



Course name: OPERATING SYSTEMS [3 0 0 3]

Course code: DSE 3153

Introduction to OS + OS Structures

Instructors:

**Dr. Sandhya Parasnath Dubey
&
Dr. Abhilash K Pai**

Courtesy: Slides are adapted from the textbook “Operating System Concepts” by Silberschatz et al.



Course Outcomes (COs)

At the end of this course, the student should be able to:

CO1	Understand Operating System components, structure and the process concept
CO2	Analyze the requirement of threads and process scheduling.
CO3	Explore process synchronization, deadlocks avoidance, detection and recovery algorithms.
CO4	Evaluate different memory management strategies.
CO5	Analyze file Systems, secondary storage management, and system protection.



DSE 3153 OPERATING SYSTEMS SYLLABUS

Introduction: Operating System Structure and Operations, Process Management, Memory Management, Storage Management;

System Structures: Operating System Services, User Operating System Interfaces, Types of System Calls, System Programs, Operating System Structure, System Boot;

Process Concept: Overview, Process Scheduling, Operations on Processes, Inter-process Communication;

Multithreaded Programming : Multithreaded Models, Thread Libraries;

Process scheduling: Scheduling Algorithms, Thread Scheduling, Linux scheduling;



DSE 3153 OPERATING SYSTEMS SYLLABUS

Synchronization: Critical Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores,

Memory Management Strategies: Logical Versus Physical Address Space, Segmentation, Contiguous Memory Allocation, Paging, Structure of Page Table, Segmentation,

Virtual Memory Management: Demand Paging, Copy-On-Write, Page Replacement, Allocation of Frames, Thrashing,

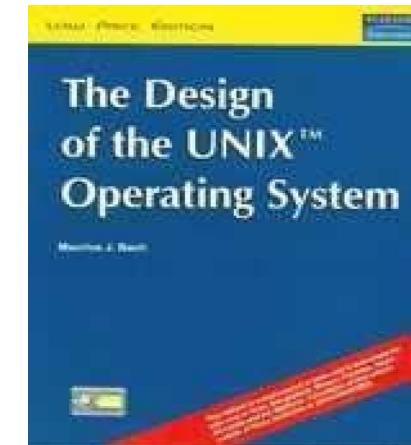
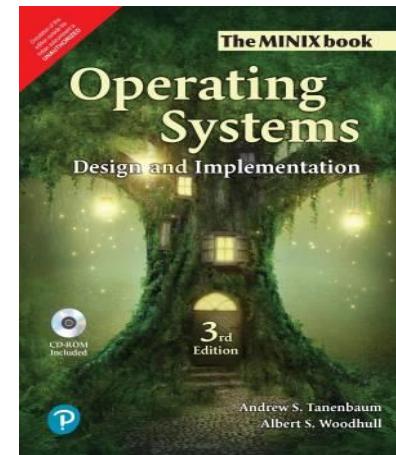
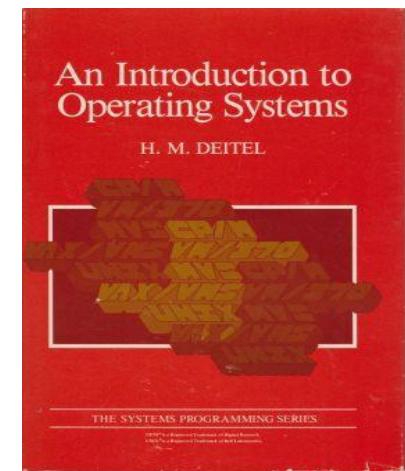
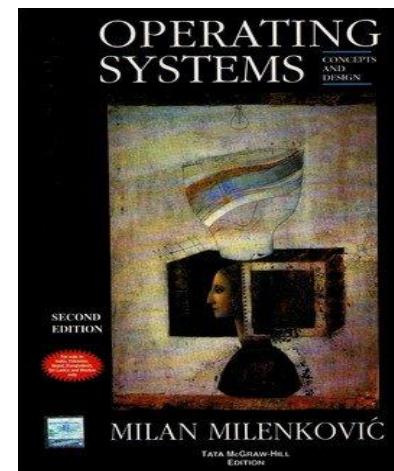
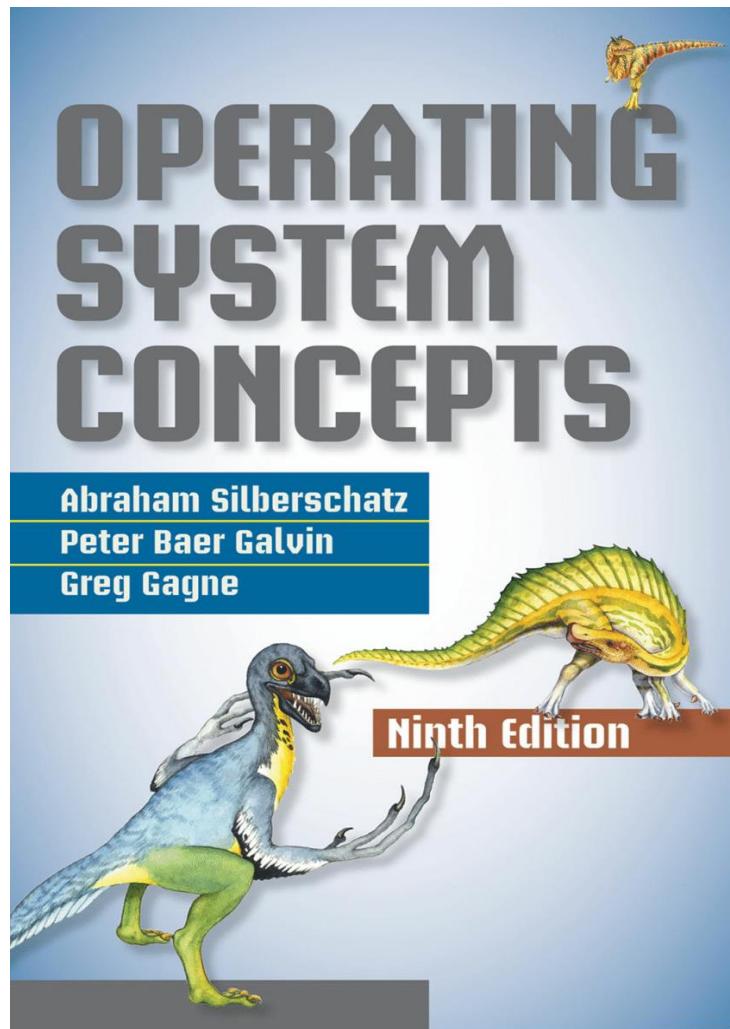
Mass-Storage Structure: Disk Scheduling, Swap-Space Management,

Deadlocks: System Model, Deadlock: Deadlock prevention, Avoidance, Detection, Recovery,

File System: File Concept, Protection.



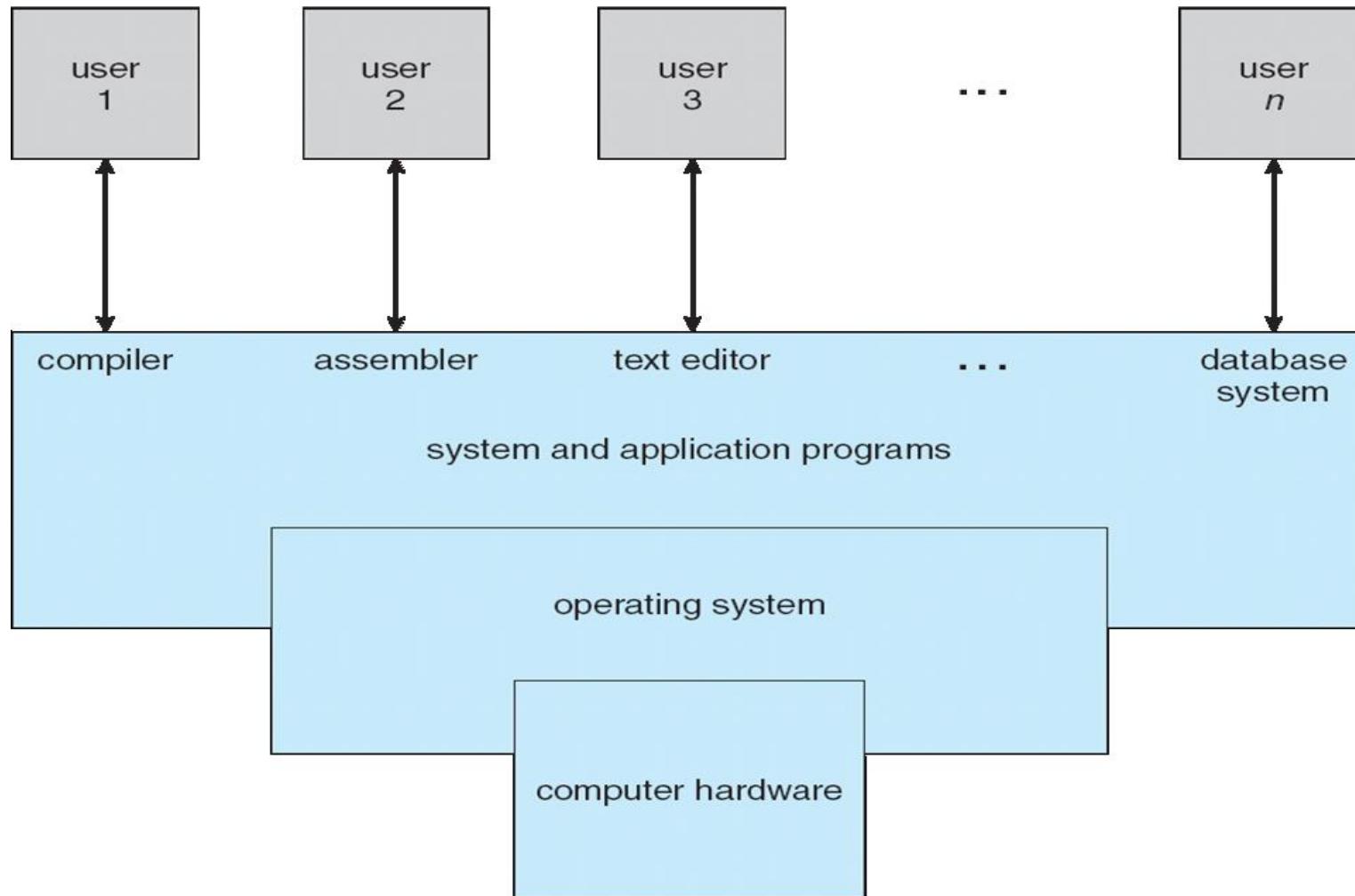
Reference Books





Introduction to OS

Four Components of a Computer System (Abstract view)





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
 - **System and 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



What is an Operating System?

- A program that acts as an intermediary between a user of a computer software and the computer hardware
- OS is a system software
- It provides an environment within which other programs can do useful work
- **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



A Simple Program

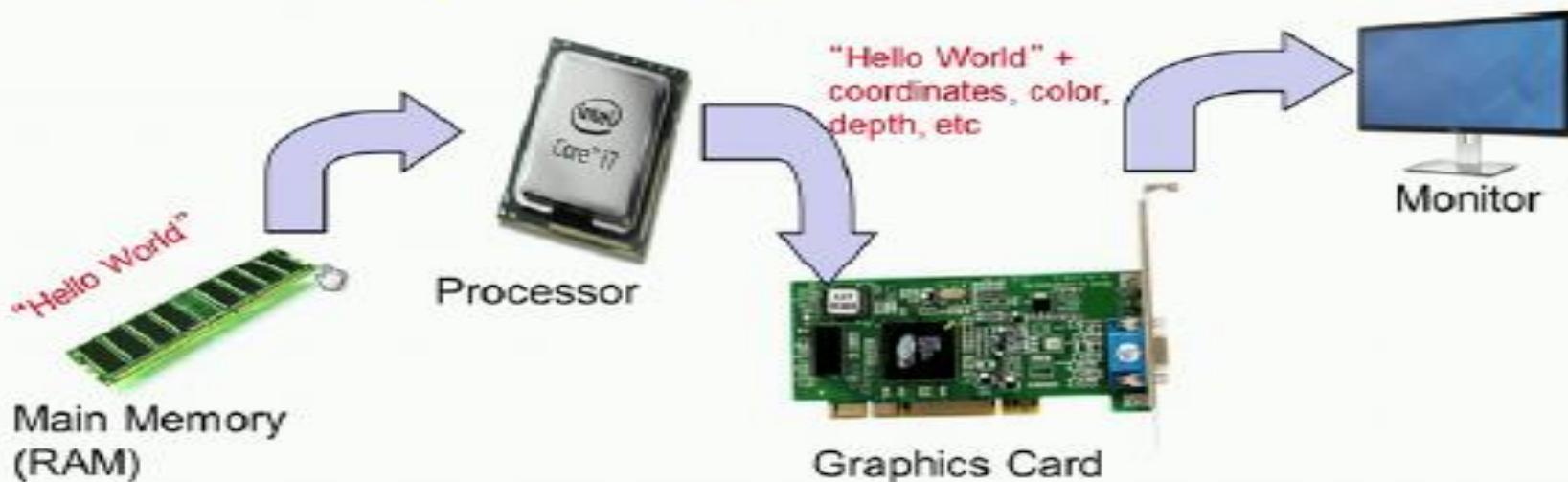
What is the output of the following program?

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

How is the string displayed on the screen?

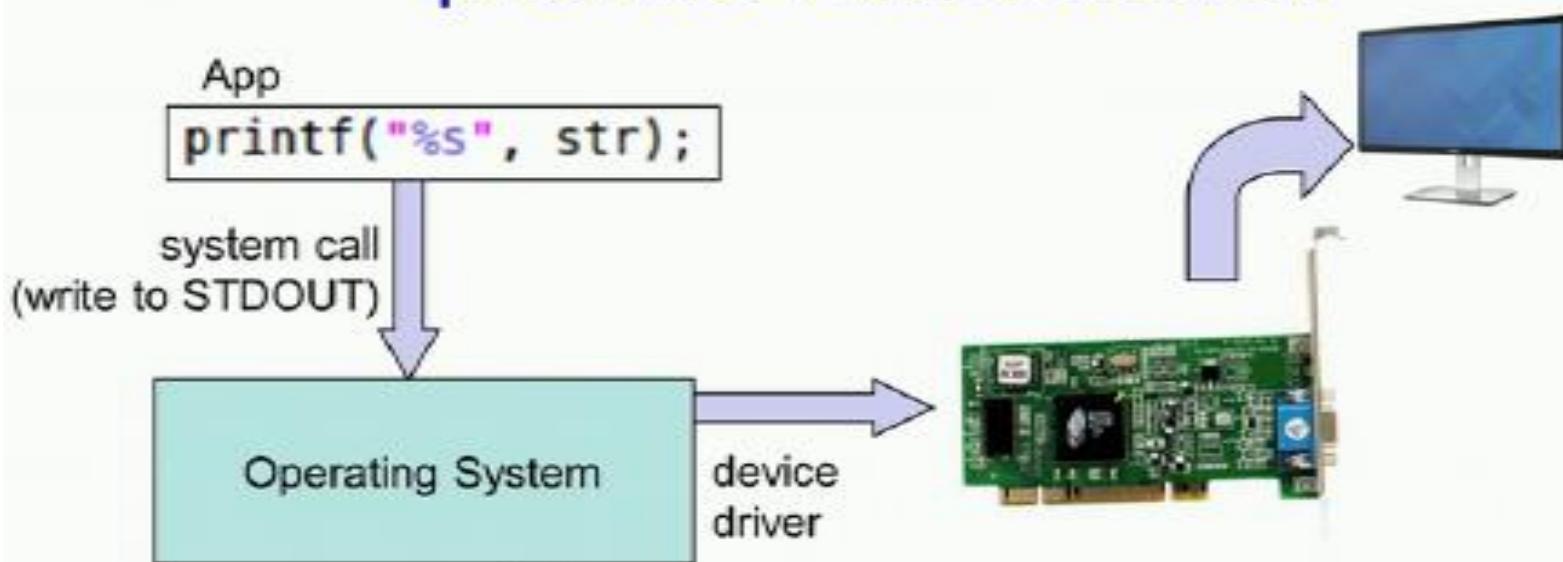
Displaying on the Screen



- Can be complex and tedious
- Hardware dependent

Without an OS, all programs need to take care of every nitty gritty detail

Operating Systems provide Abstraction



- **Easy to program apps**
 - No more nitty gritty details for programmers
- **Reusable functionality**
 - Apps can reuse the OS functionality
- **Portable**
 - OS interfaces are consistent. The app does not change when hardware changes

OS as a Resource Manager

- Multiple apps but limited hardware

Apps



```
#include <stdio.h>
int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

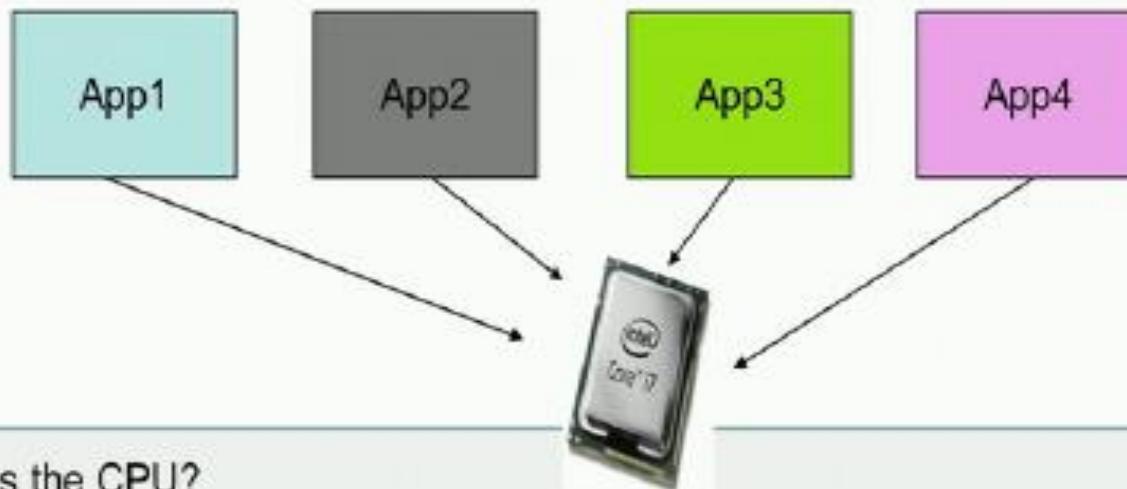


Operating Systems



Allows sharing of hardware!!

Sharing the CPU



Who uses the CPU?

App1

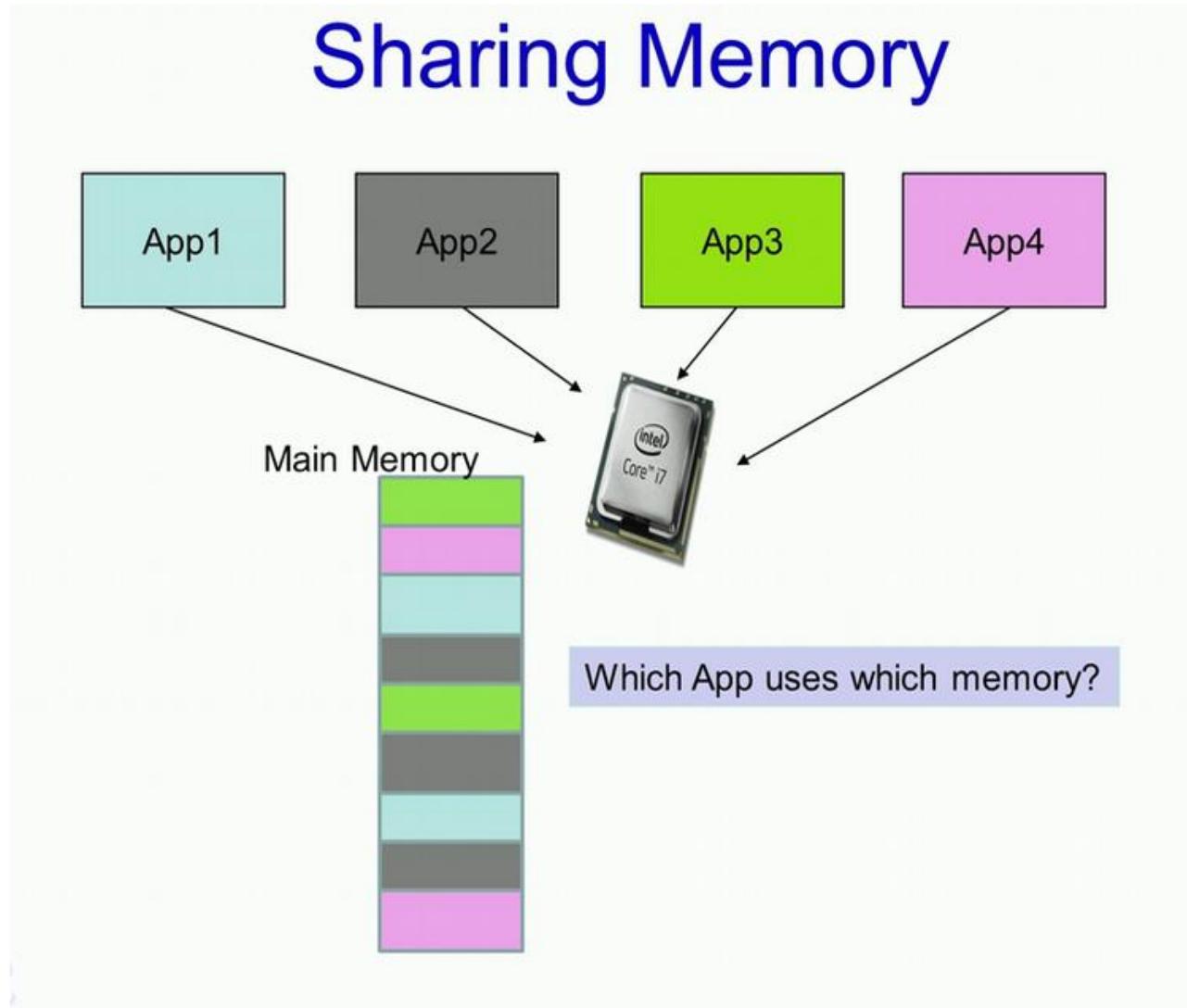
App2

App3

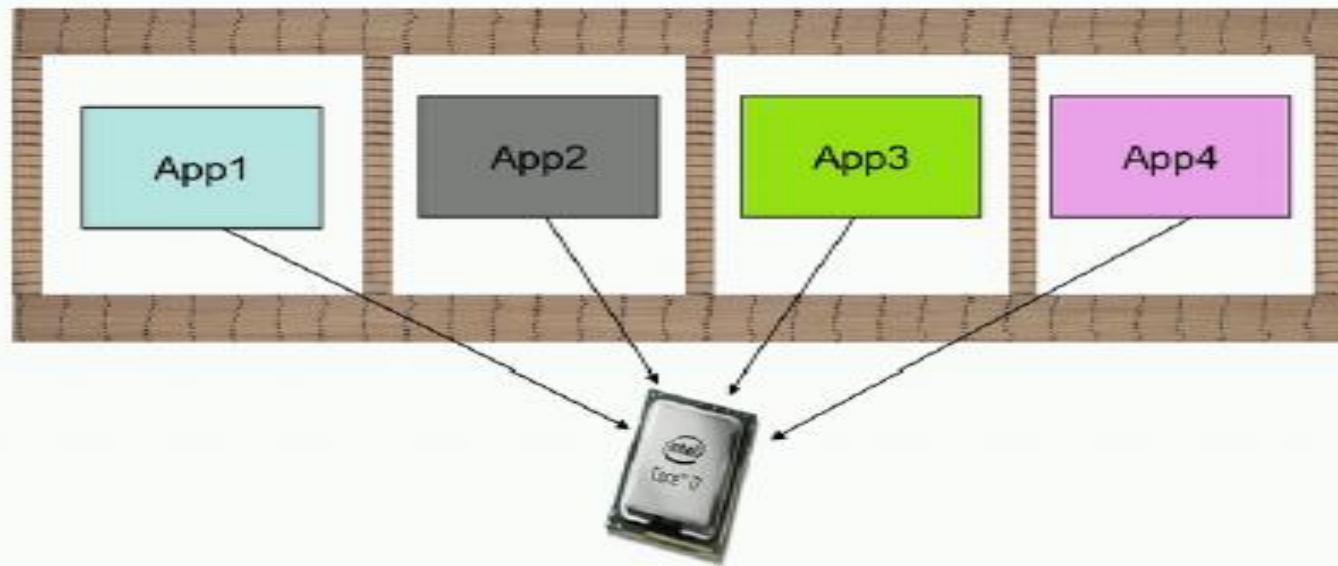
App4

time

Sharing Memory



Share but Isolate



Share resources but keep applications isolated from each other

R



What Operating Systems Do

User View: The user's view of the computer varies according to the interface being used

- **Single User Computers (e.g., PC, workstations):** Users want convenience, **ease of use and good performance**
 - Don't care about **resource utilization**
- **Multi User Computers:** shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- **Handheld computers (e.g., smartphones and tablets):** are resource poor, optimized for usability and battery life
- **Embedded Computers (e.g., Computers in home devices and automobiles):** Some computers have little or no user interface, such as embedded computers in devices and automobiles



What Operating Systems Do

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. There are two different views:

- OS is a **resource allocator**
 - Manages all resources
 - 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



Types of Systems and Roles of OS within each

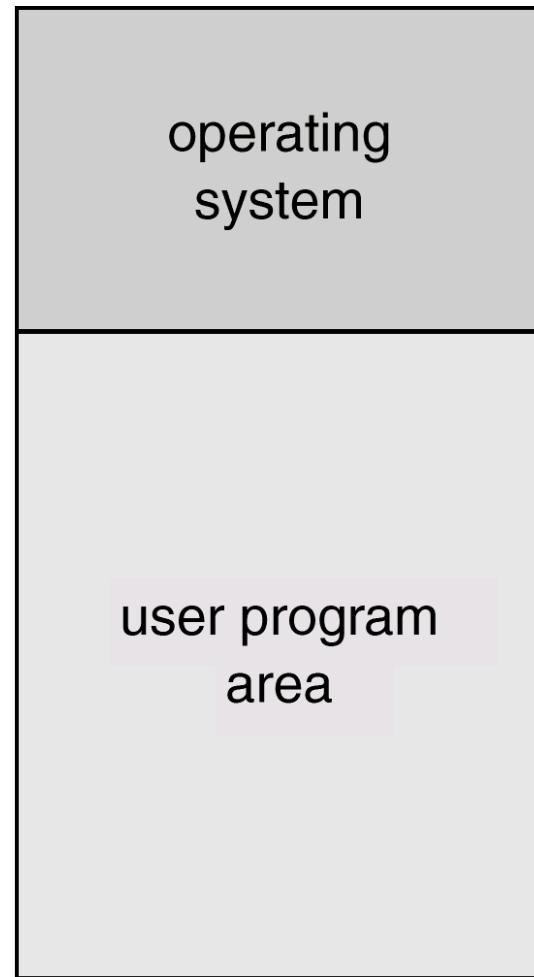
- Simple Batch Systems
- Multiprogramming Batched Systems
- Time-Sharing Systems
- Personal-Computer Systems
- Parallel Systems
- Distributed Systems
- Real -Time Systems



Simple Batch Systems

- Hire an operator
- User ≠ operator
- Add a card reader
- Reduce setup time by batching similar jobs
- Automatic job sequencing – automatically transfers control from one job to another. **First rudimentary operating system.**
- Resident monitor
 - initial control in monitor
 - control transfers to job
 - when job completes control transfers back to monitor

Memory Layout for a Simple Batch System



Spooling (simultaneous peripheral operation on-line)

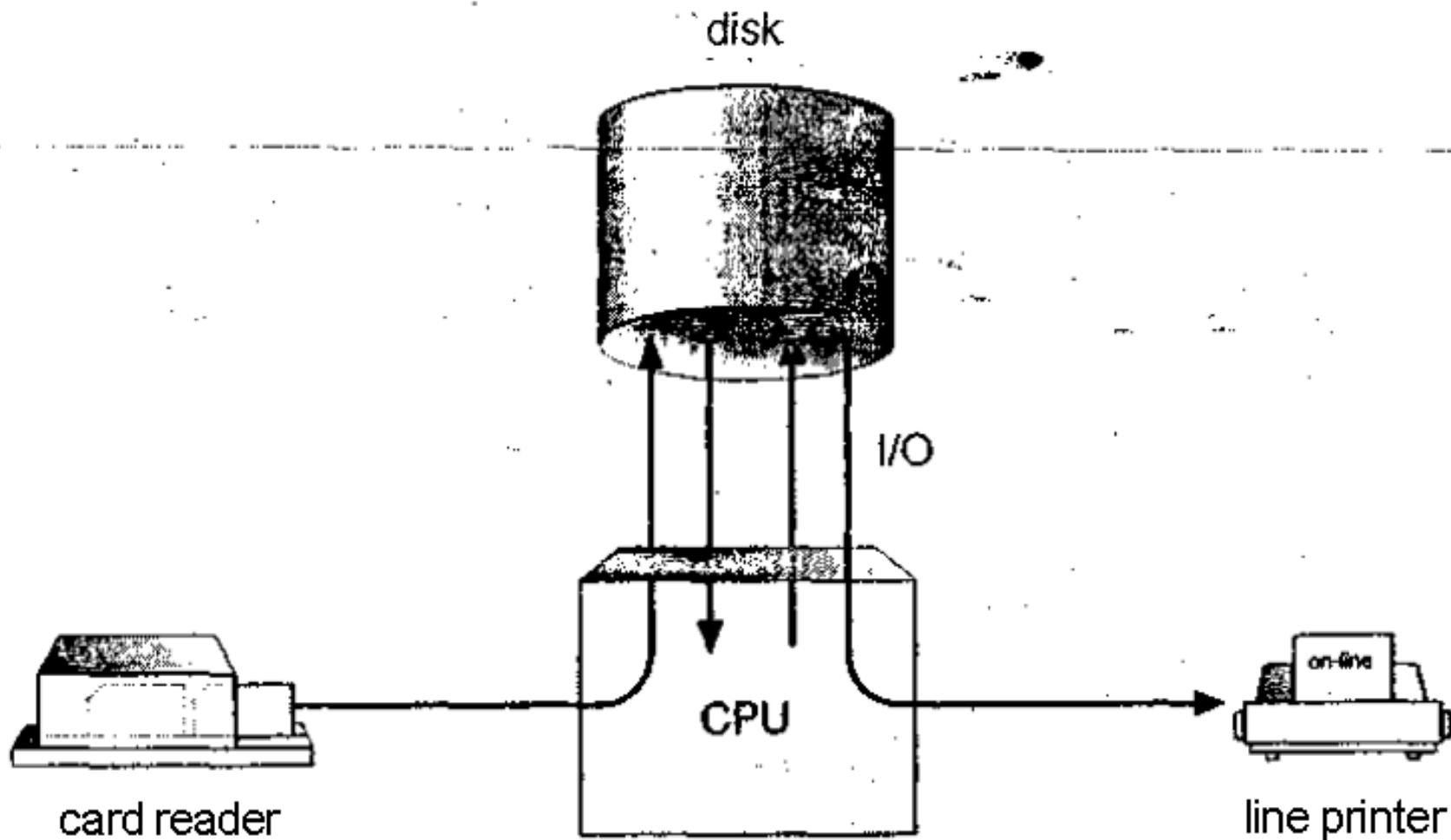


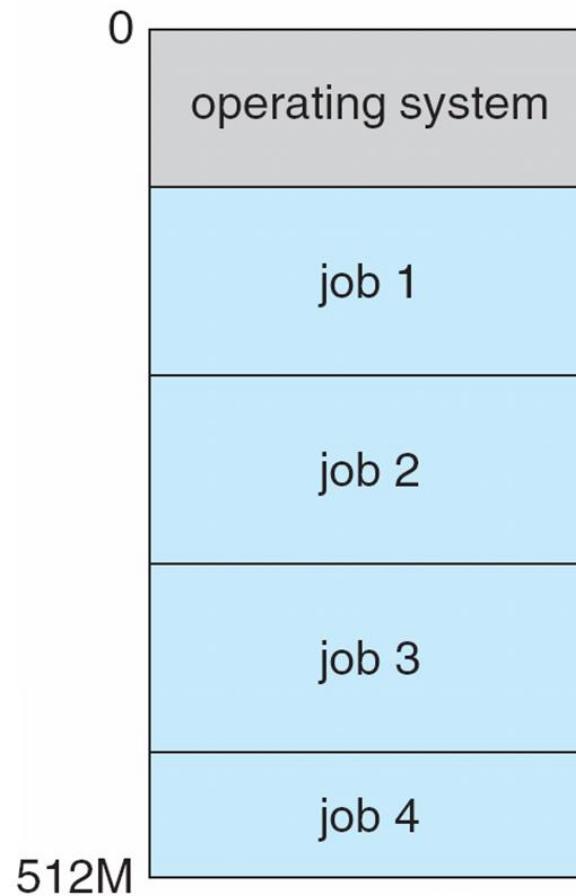
Figure 1.3 Spooling.



• **Multiprogramming**

- One of the most important aspects of operating systems is the **ability to multiprogram**.
- A single program cannot, keep either the CPU or the I/O devices busy at all times
- **Multiprogramming** increases **CPU utilization** by organizing jobs (code and data) so that the CPU always has one to execute.
- Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but
- **they do not provide for user interaction with the computer system.**

Memory Layout for Multiprogrammed System





OS Features Needed for Multiprogramming

- **Job scheduling:** If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling**
- **Memory management:** the system must allocate the memory to several jobs.
- **CPU scheduling:** the system must choose among several jobs ready to run.
- **Degree of Multiprogramming:** Number of processes in the main memory



- **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)

logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

- Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system.
 - **Response time** should be < 1 second
 - Each user has at least one program executing in memory ⇒ **process**
 - If several jobs ready to run at the same time ⇒ **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



Time-Sharing Systems—Interactive Computing

- The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory).
- A job is swapped in and out of memory to the disk.
- Time-sharing systems provide a mechanism for concurrent execution, which requires sophisticated CPU scheduling schemes.
- To ensure orderly execution, the system must provide mechanisms for **job synchronization** and **communication** and must ensure that jobs do not get stuck in a **deadlock**, forever waiting for one another



Parallel Systems

- **Multiprocessor systems** with more than one CPU in close communication.
- *Tightly coupled system* – processors share memory and a clock; communication usually takes place through the shared memory.
- Advantages of parallel system:
- **Increased throughput:** By increasing the number of processors, we get more work done in a shorter period of time.
- **Economical :** The processors can share peripherals, cabinets, and power supplies. If several programs are to operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, rather than to have many computers with local disks and many copies of the data.
- **Increased reliability:** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down.
- The ability to continue providing service proportional to the level of surviving hardware is called ***graceful degradation***. Systems that are designed for graceful degradation are also *called fault-tolerant*.



Real-Time Systems

- This system is used when there **are rigid time requirements on the operation of a processor or the flow of data**
- Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs.
- Often used as a **control device in a dedicated application** such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
- **Well-defined fixed-time constraints.**

Hard real-time system

- It guarantees that critical tasks complete on time.
- This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it.
- It is used as a control device in a dedicated application. A hard real-time operating system has well-defined, fixed time constraints.
- Processing *must* be done within the defined constraints, or the system will fail.
- Secondary storage limited or absent, data stored in short-term memory, or read-only memory (ROM)



Real-Time Systems

- *Soft real-time system*
 - Limited utility in industrial control or robotics
 - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.
- Soft real-time systems have less stringent timing constraints, and do not support deadline scheduling.



Operating-System Operations

- Modern operating systems are **interrupt driven**.
- Events are almost always signaled by the occurrence of an interrupt or a trap.
- A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.
- Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running



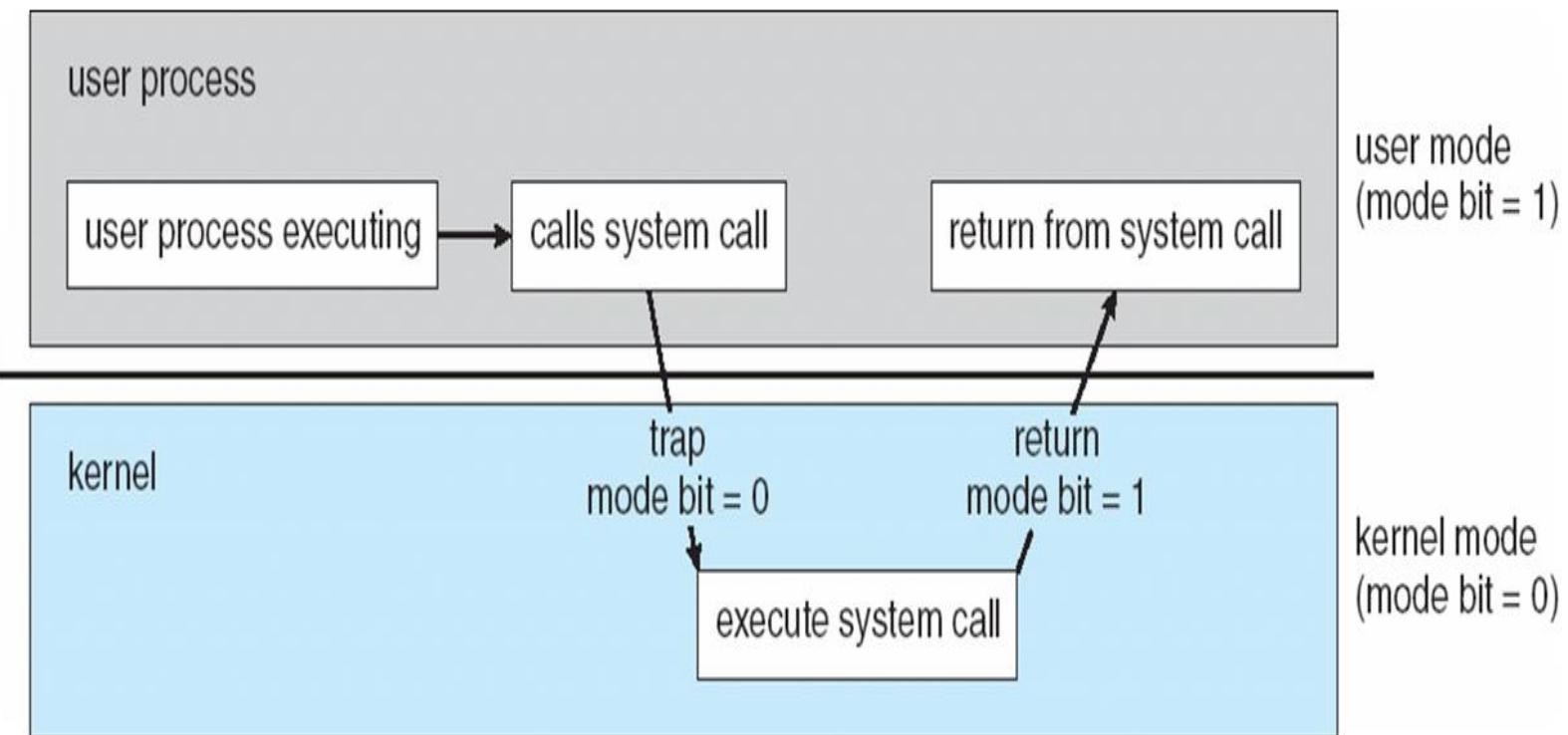
Operating-System Operations

- **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**



Operating-System Operations (cont.)

Transition from User to Kernel Mode





Operating-System Operations (cont.)

- 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



Components/Functions of OS

- Process Management
- Main Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management
- Protection System
- Networking
- Command-Interpreter System



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**
- 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
 - Varies between WORM (write-once, read-many-times) and RW (read-write)



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



Protection and Security

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



Operating System Structures

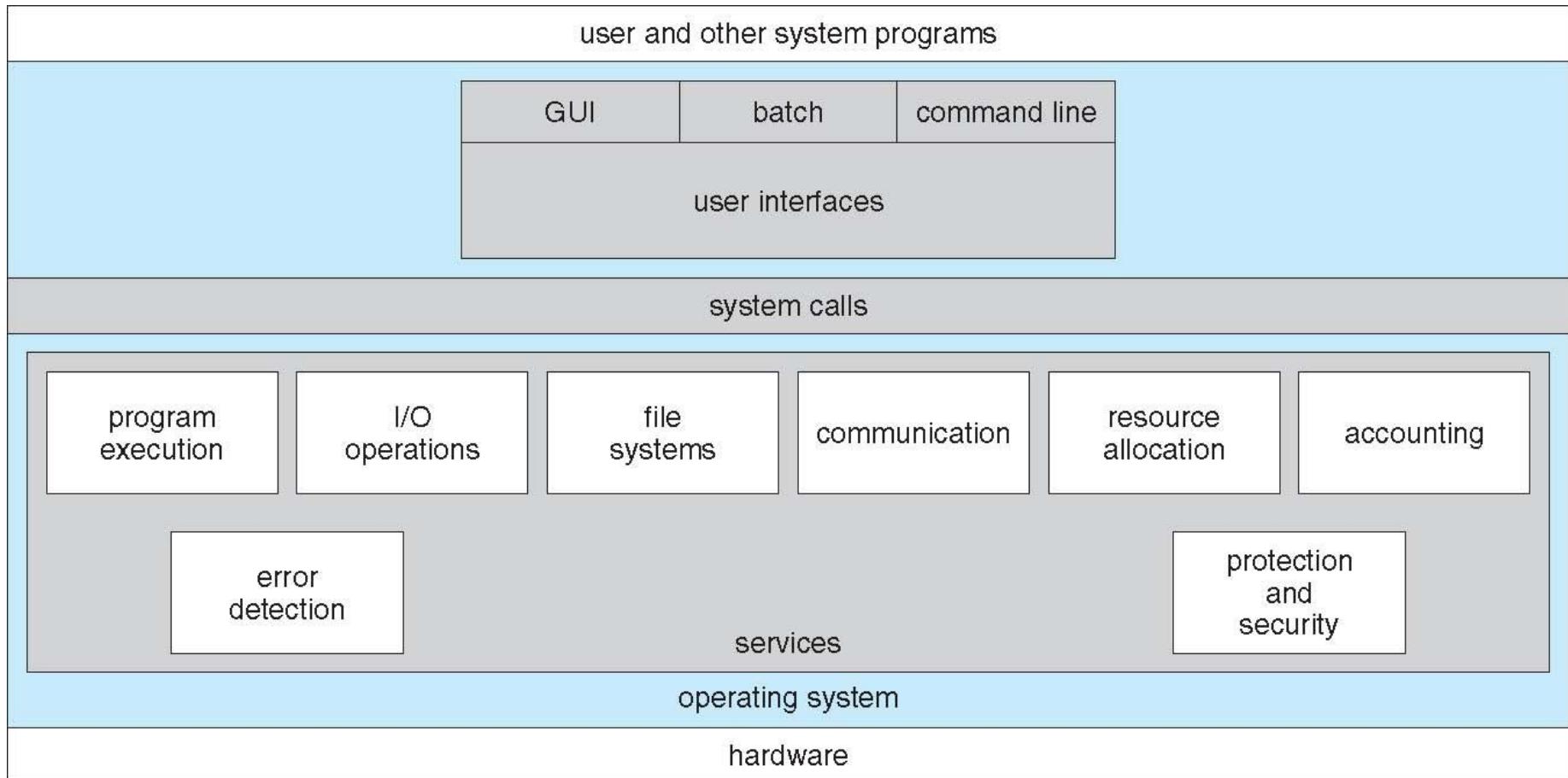


Operating System Services

- Operating systems provide an environment for execution of programs
 - Provides certain services to
 - **Programs** and
 - **Users of those programs**
- Basically two types of services:
- ✓ Set of OS services provides **functions** that are helpful to the user
 - ✓ Set of OS functions for ensuring the **efficient operation** of the system itself **via resource sharing**



A View of Operating System Services





Operating System Services

- 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)



Operating System Services

I/O operations - A running program may require I/O, which may involve a file or an I/O device

File-system manipulation - 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)



Operating System Services

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



Operating System Services

- 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



Operating System Services

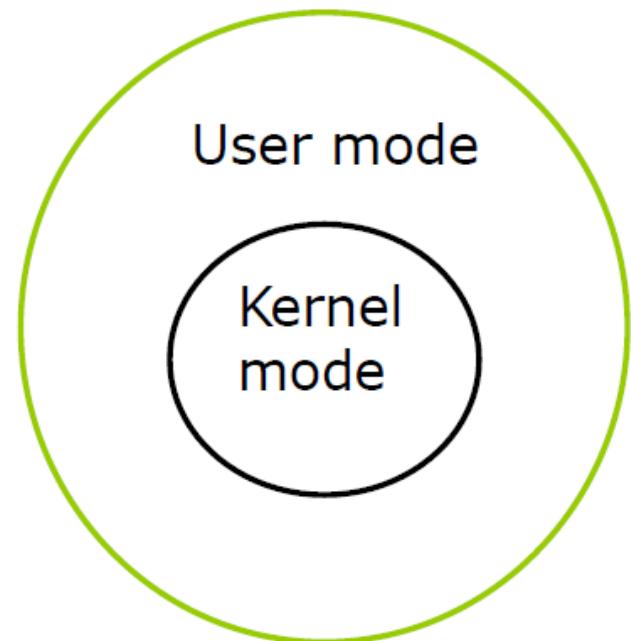
Protection and security - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

Protection involves ensuring that all access to system resources is controlled

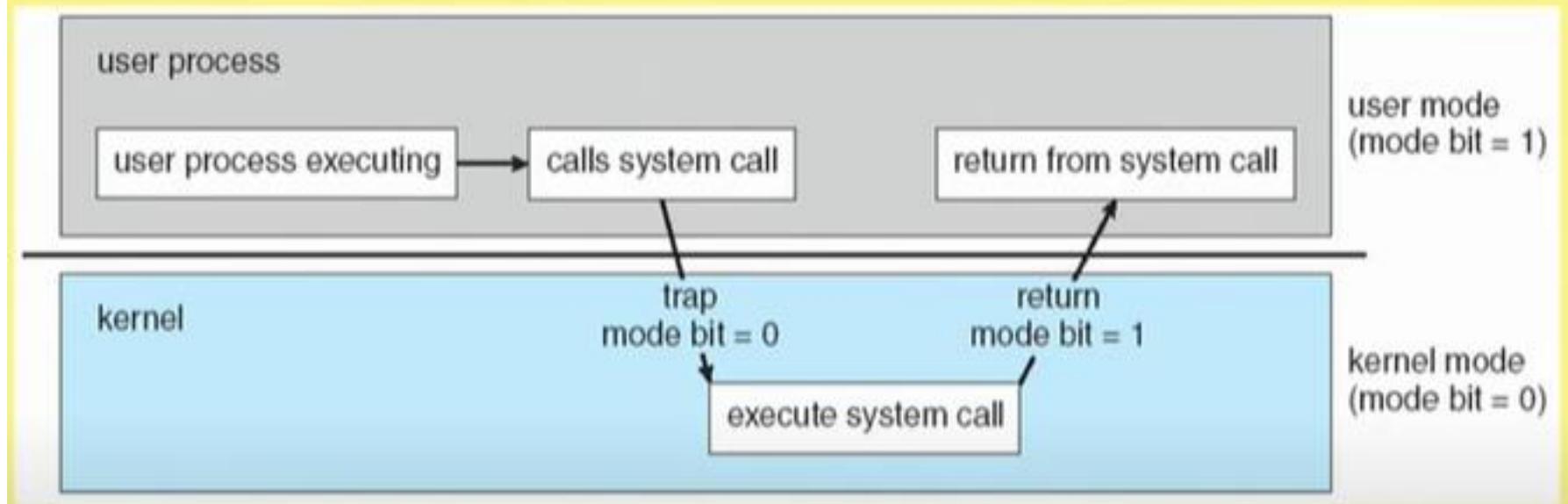
Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

System Calls

- **It provides an interface to the services made available by an OS.** (Available as routines in C,C++,assembly languages)
- **Mode of Operation**
 - **User Mode**
 - Safer mode of operation
 - **Kernel mode**
 - Privileged mode
 - Access to all H/W resources



Transition from User to Kernel Mode

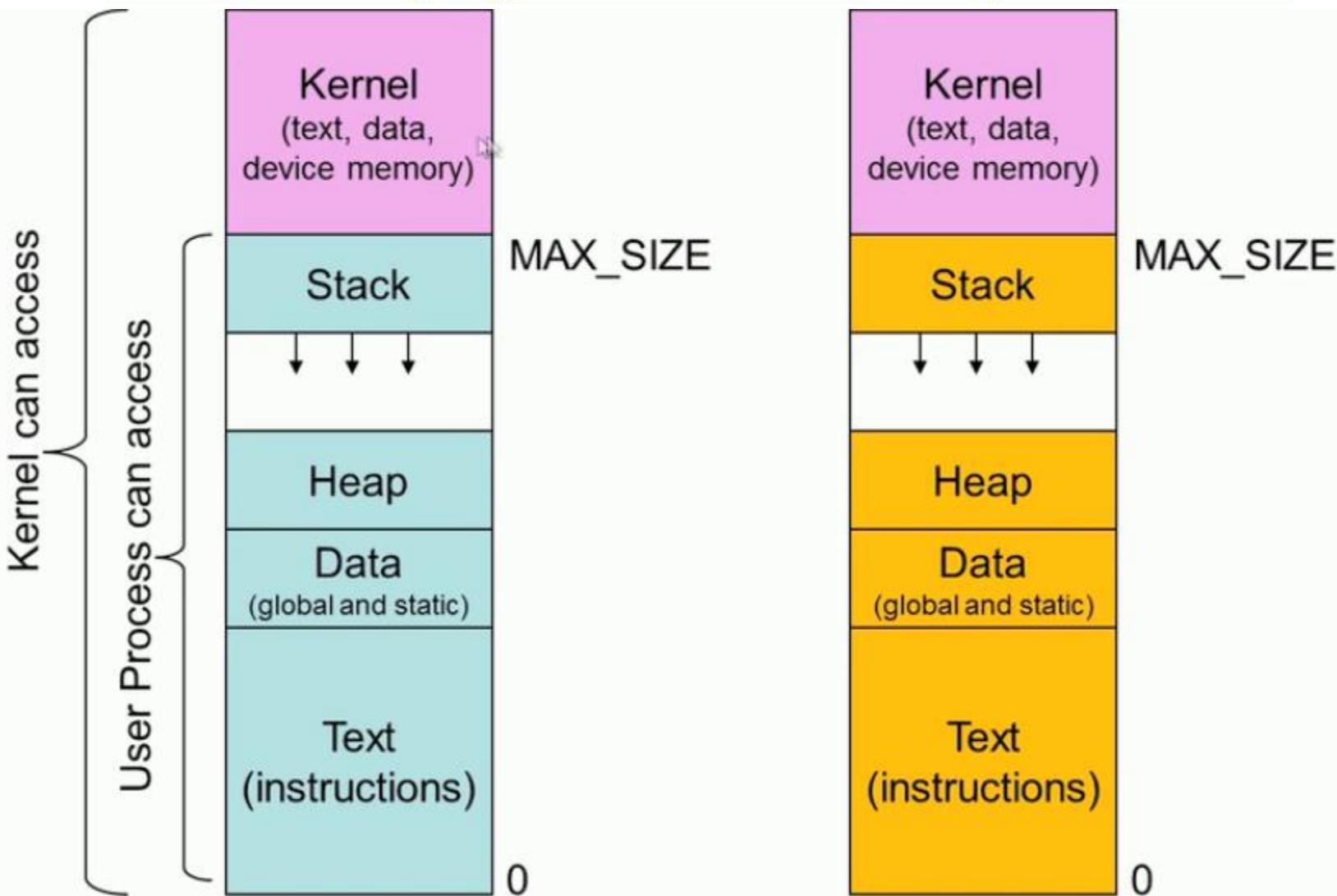




System Calls

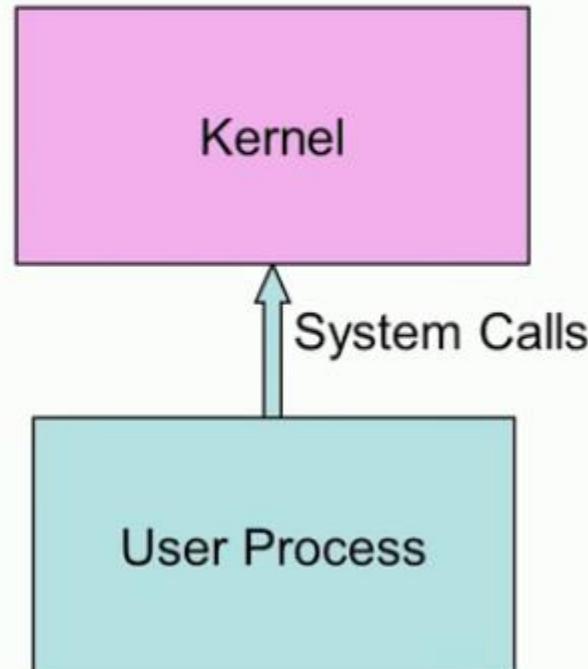
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are
 - Win32 API for Windows,
 - POSIX API for POSIX-based systems (including virtually all versions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

Communicating with the OS (System Calls)



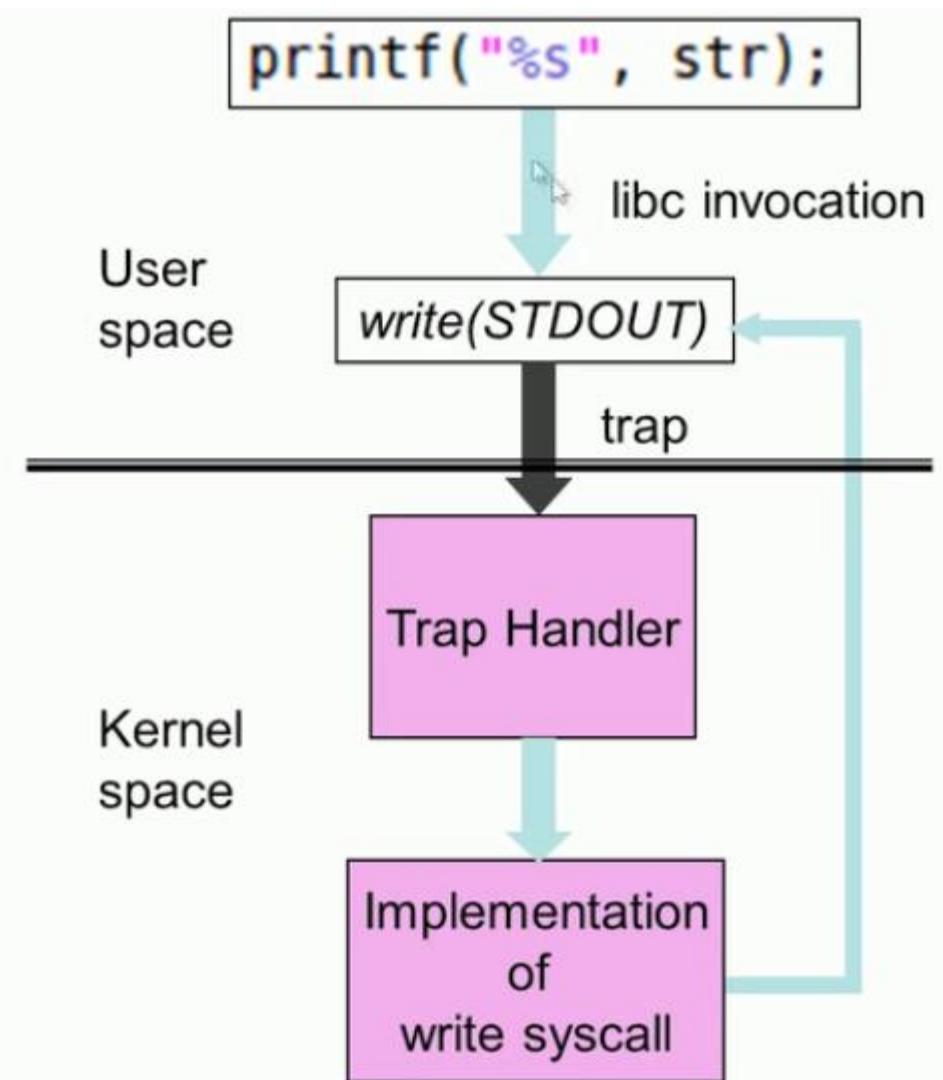
Communicating with the OS (System Calls)

- System call invokes a function in the kernel using a Trap
- This causes
 - Processor to shift from user mode to privileged mode
- On completion of the system call, execution gets transferred back to the user process





Example (write system call)



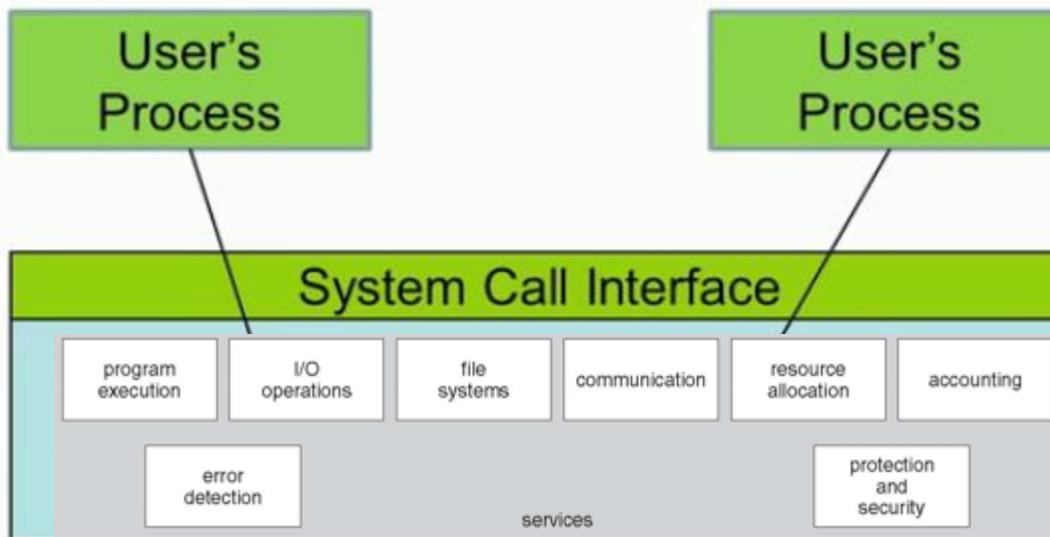


System Call vs Procedure Call

System Call	Procedure Call
Uses a TRAP instruction (such as int 0x80)	Uses a CALL instruction
System shifts from user space to kernel space	Stays in user space ... no shift
TRAP always jumps to a fixed address (depending on the architecture)	Re-locatable address

System Call Interfaces

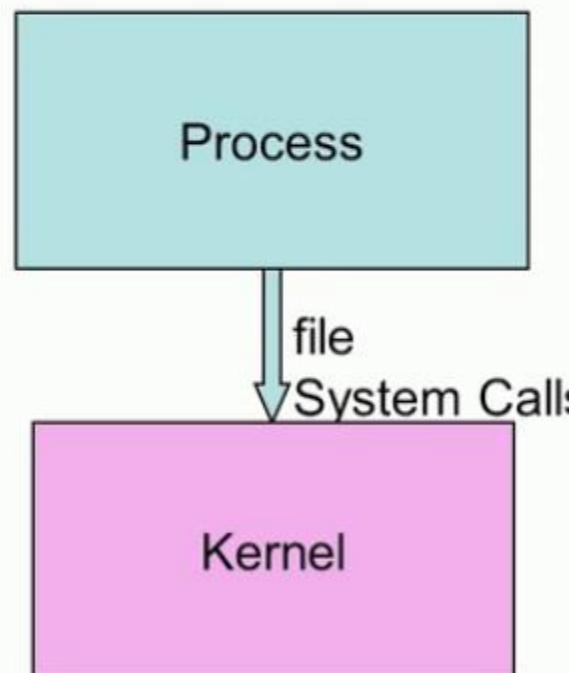
- System calls provide users with interfaces into the OS.
- What set of system calls should an OS support?
 - Offer sophisticated features
 - But yet be simple and abstract whatever is necessary
 - General design goal : rely on a few mechanisms that can be combined to provide generality



File System Calls

(How to design?)

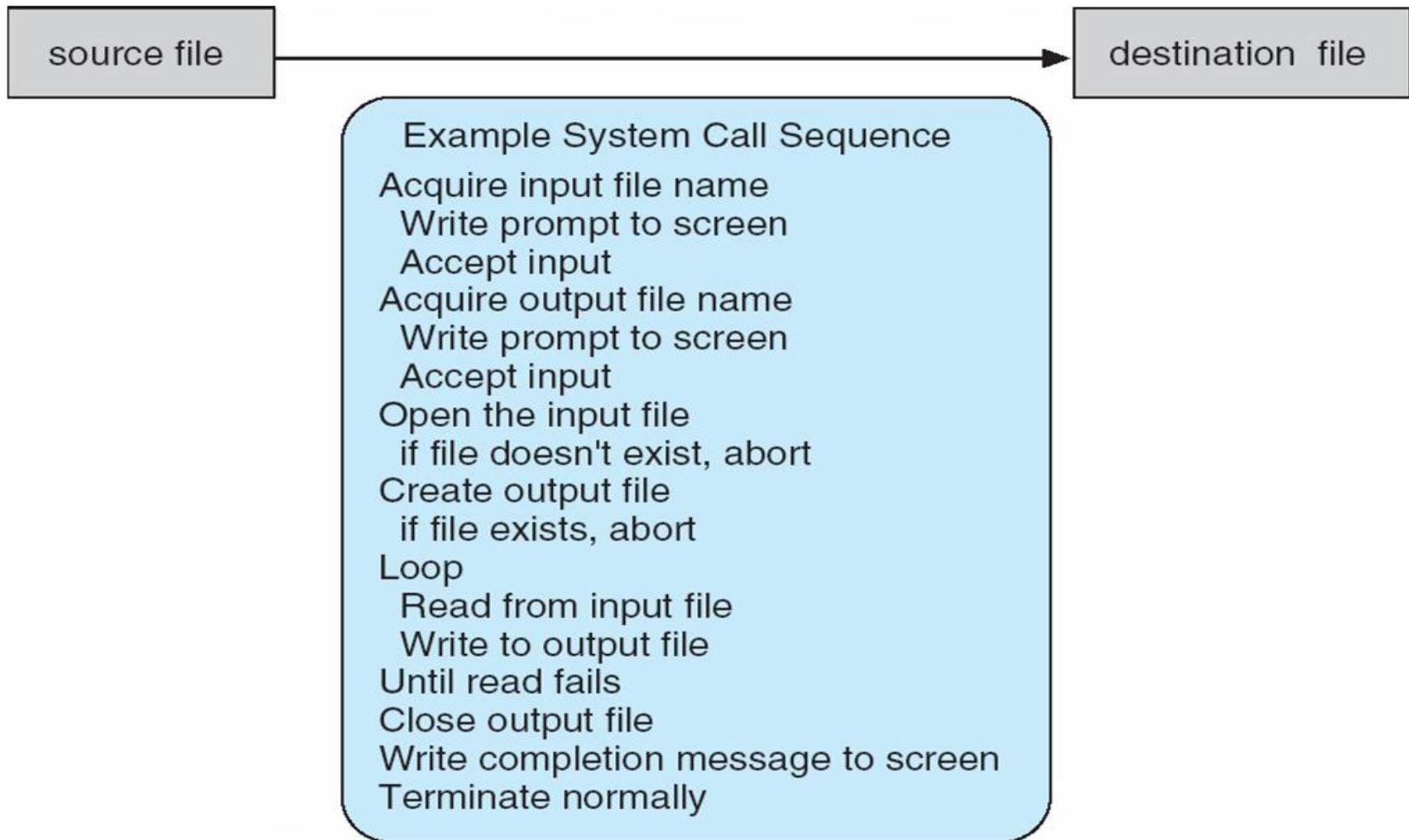
- Files: data persistent across reboots
- What should the file system calls expose?
 - Open a file, read/write file, creation date, permissions, etc.
 - Permission for multiple access to files
 - More sophisticated options like seeking into a file, linking, etc.
- What should the file system calls hide?
 - Details about the storage media.
 - Exact locations in the storage media.





Example of System Calls

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

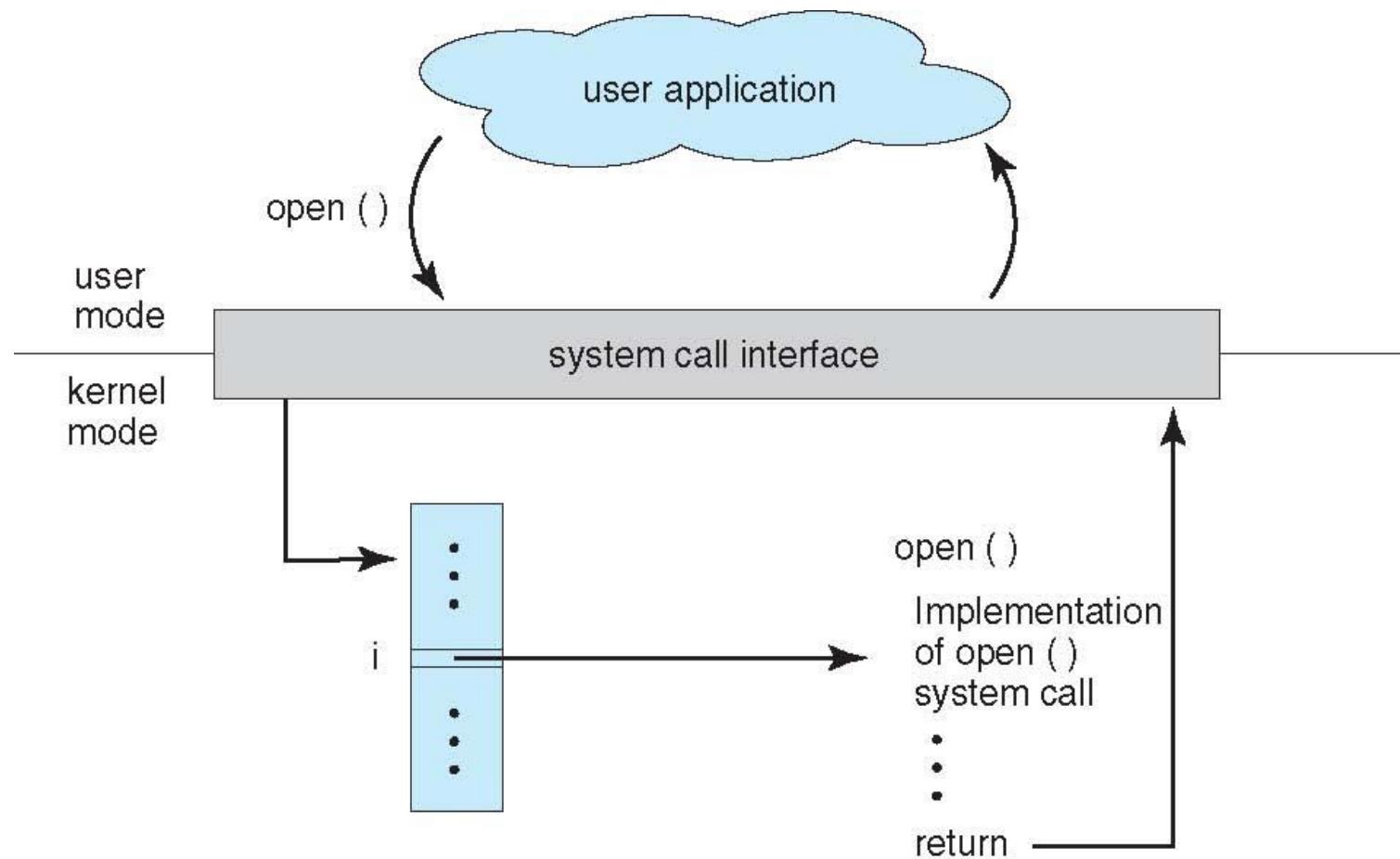




System Call Implementation

- 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)

System Call – OS Relationship



Available as routines written in C/C++



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
 - **Debugger** for determining **bugs, single step** execution
 - **Locks** for managing access to shared data between processes



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
 - create, delete communication connection
 - send, receive messages if **message passing model** to **host name** or **process name**
 - From **client** to **server**
 - **Shared-memory model** create and gain access to memory regions
 - transfer status information
 - attach and detach remote devices



Types of System Calls (Cont.)

- Protection
 - Control access to resources
 - Get and set permissions
 - Allow and deny user access



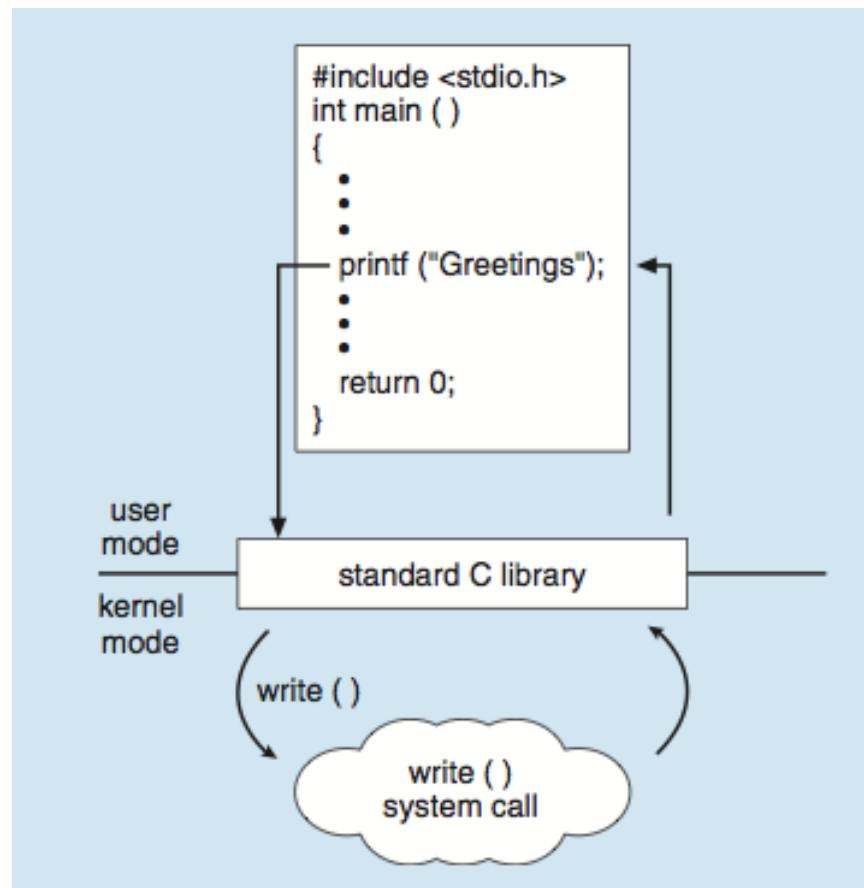
Examples of Windows and Unix System Calls

	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



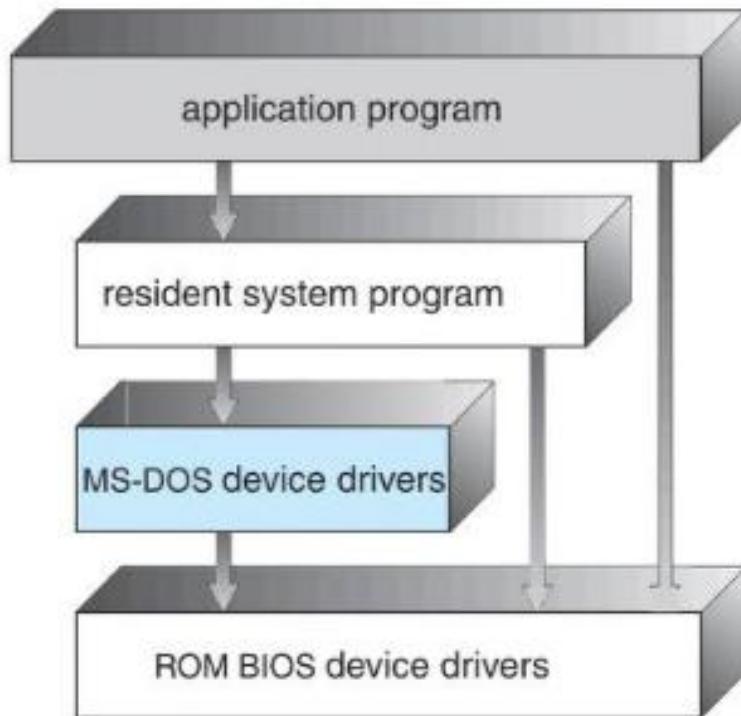


Operating System Structure

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



Simple/Monolithic Structure –MS DOS



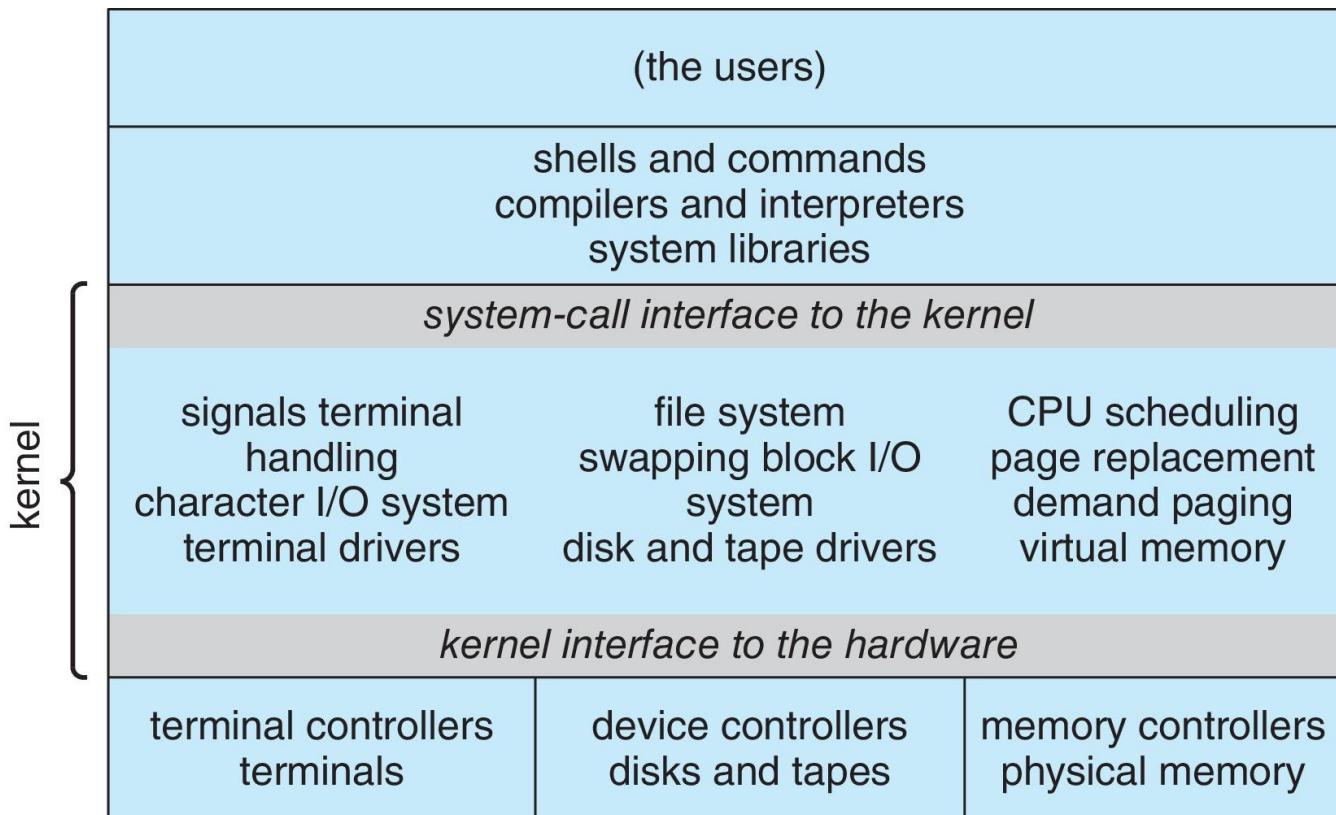


Monolithic Structure – Original UNIX

- 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

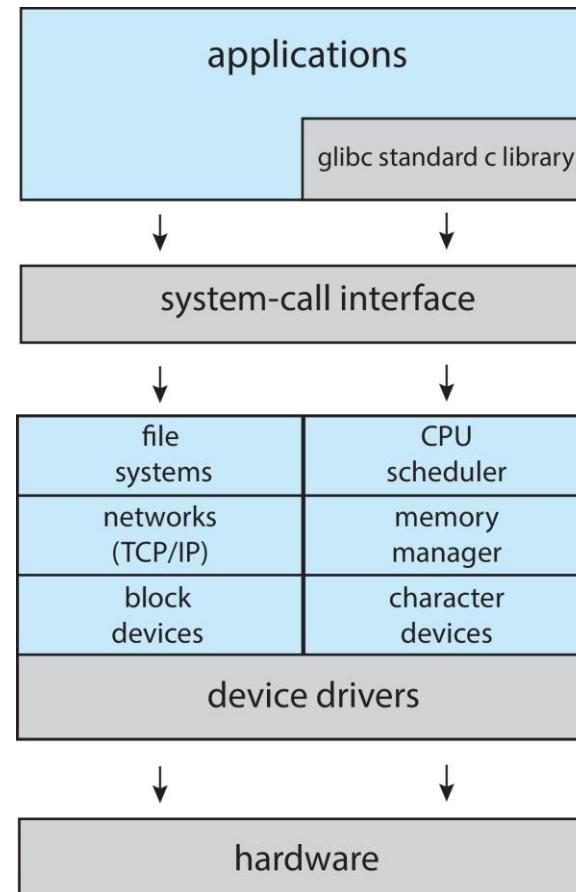
Traditional UNIX System Structure

Beyond simple but not fully layered



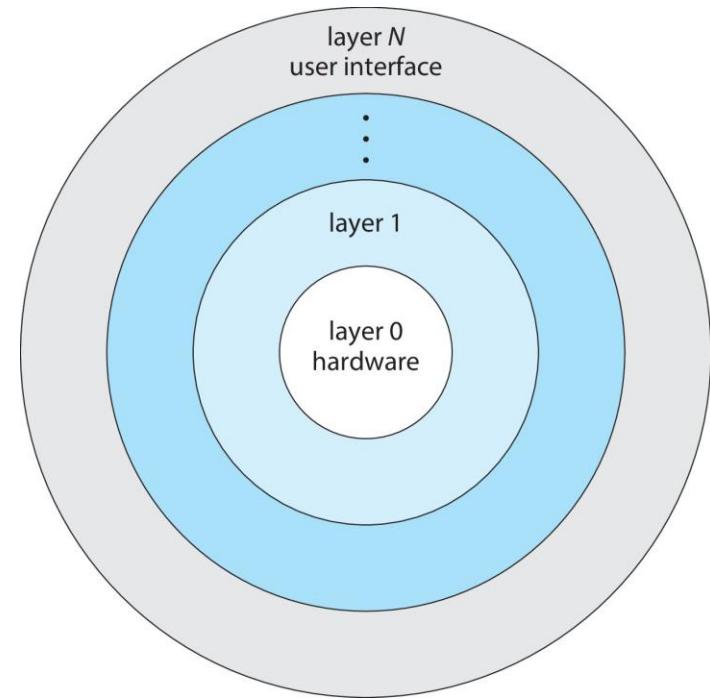
Linux System Structure

Monolithic plus modular design



Layered Approach

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

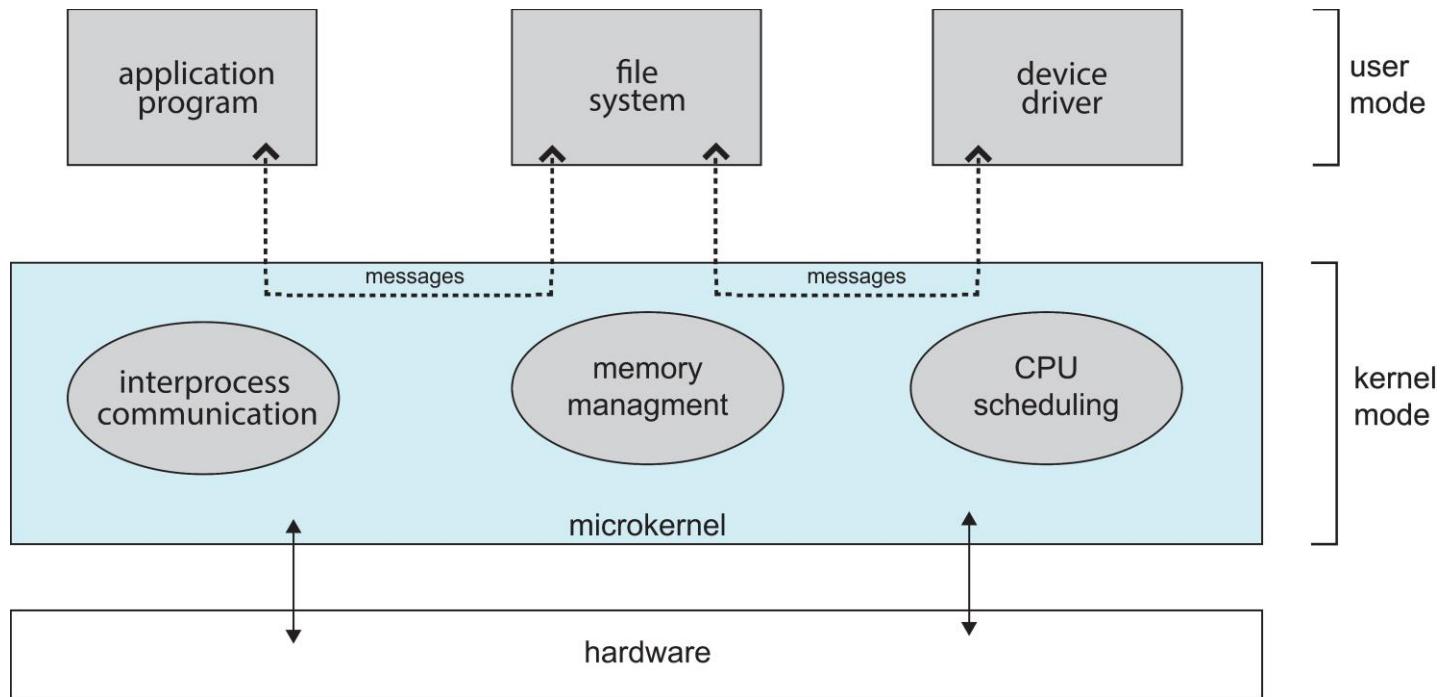




Microkernels

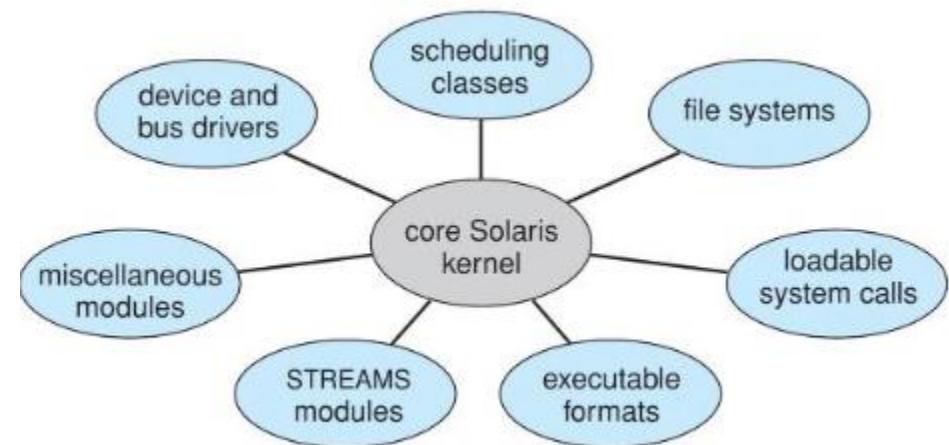
- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
 - Easier to extend a microkernel
 - Easier to port the operating system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

Microkernel System Structure



Modules

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

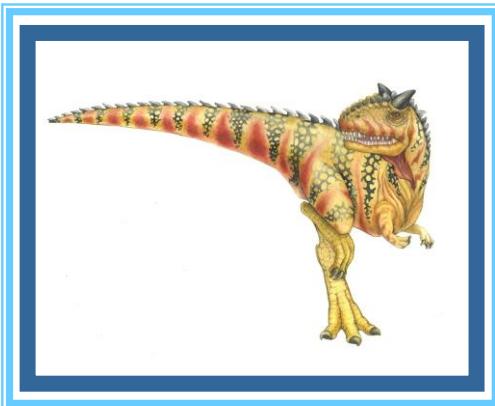




Hybrid Systems

- Most modern operating systems are not one pure model
 - Hybrid combines multiple approaches to address performance, security, usability needs
 - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
 - Windows mostly monolithic, plus microkernel for different subsystem **personalities**
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
 - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Process Concepts





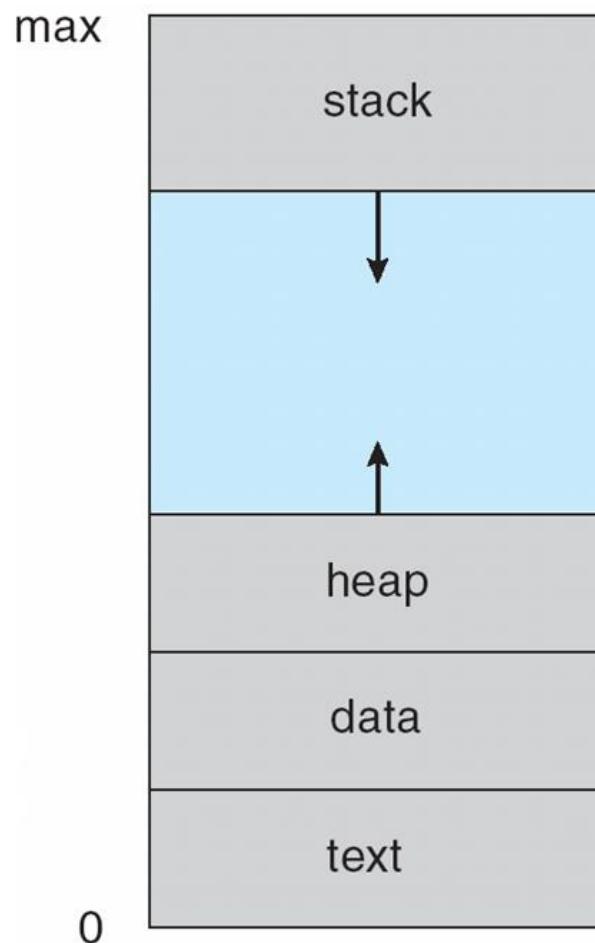
Process Concept

- **Process** – a program in execution;
 - A process will need certain resources—such as CPU time, memory, files, and I/O devices to accomplish its task.
 - These resources are allocated to the process either when it is created or while it is executing
- 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 in Memory



A process is the unit of work in a modern time-sharing system



Process Memory Map

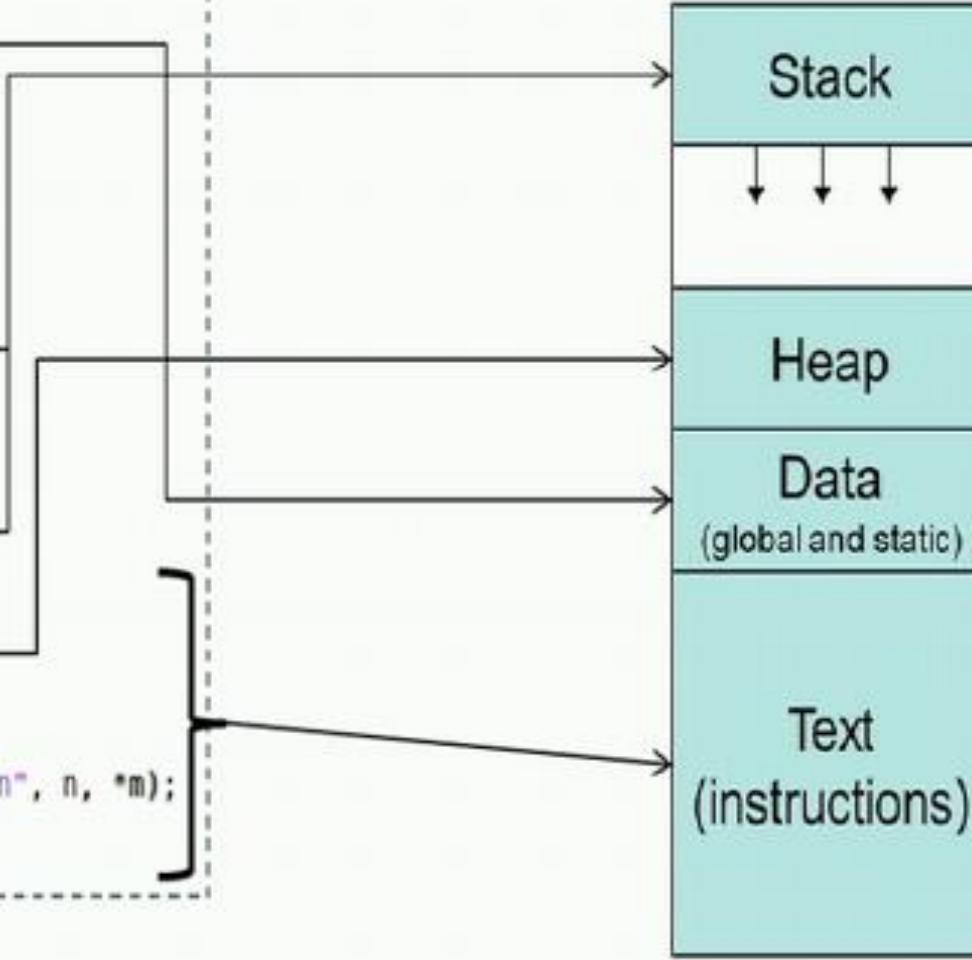
```
#include <stdio.h>
#include <stdlib.h>

int calls;

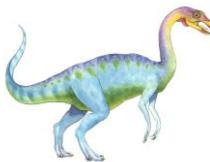
void fact(int a, int *b)
{
    calls++;
    if (a == 1) return;
    *b = *b * a;
    fact(a - 1, b);
}

int main(){
    int n, *m;

    scanf("%d", &n);
    m = malloc(sizeof(int));
    *m = 1;
    fact(n, m);
    printf("Factorial(%d) is %d\n", n, *m);
    free(m);
}
```



MAX_SIZE
0xc0000000
in Linux
0x80000000
in xv6



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
- Operating system processes executing system code and user processes executing user code.
- Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them





Process State

- As a process executes, it changes **state**
- The state of a process is defined in part by the current activity of that process
- A process may be in one of the following states:
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal)
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution
- Only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*





Diagram of Process State

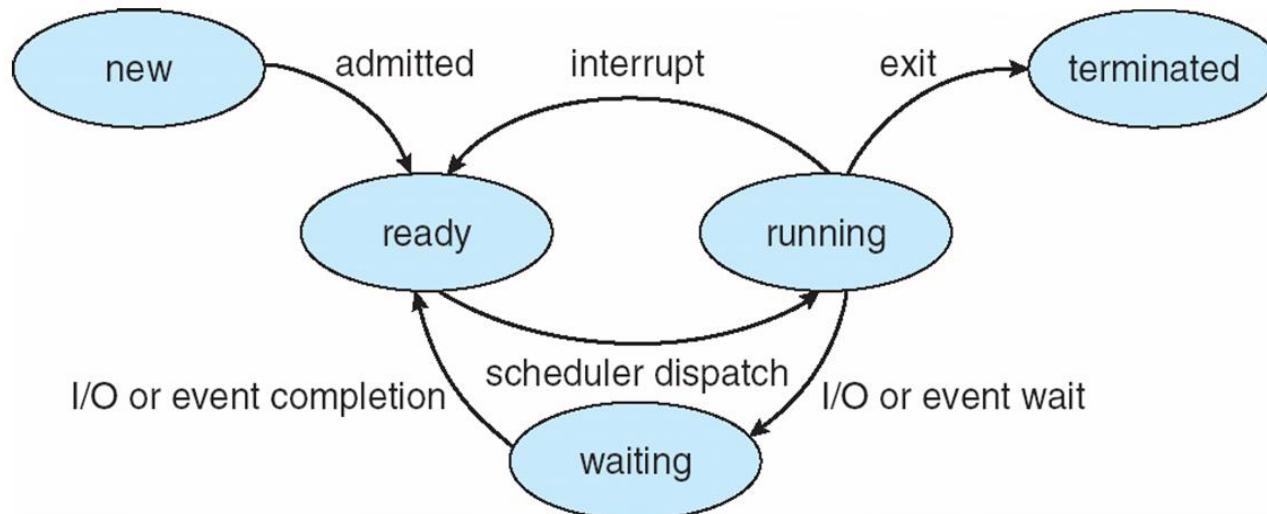
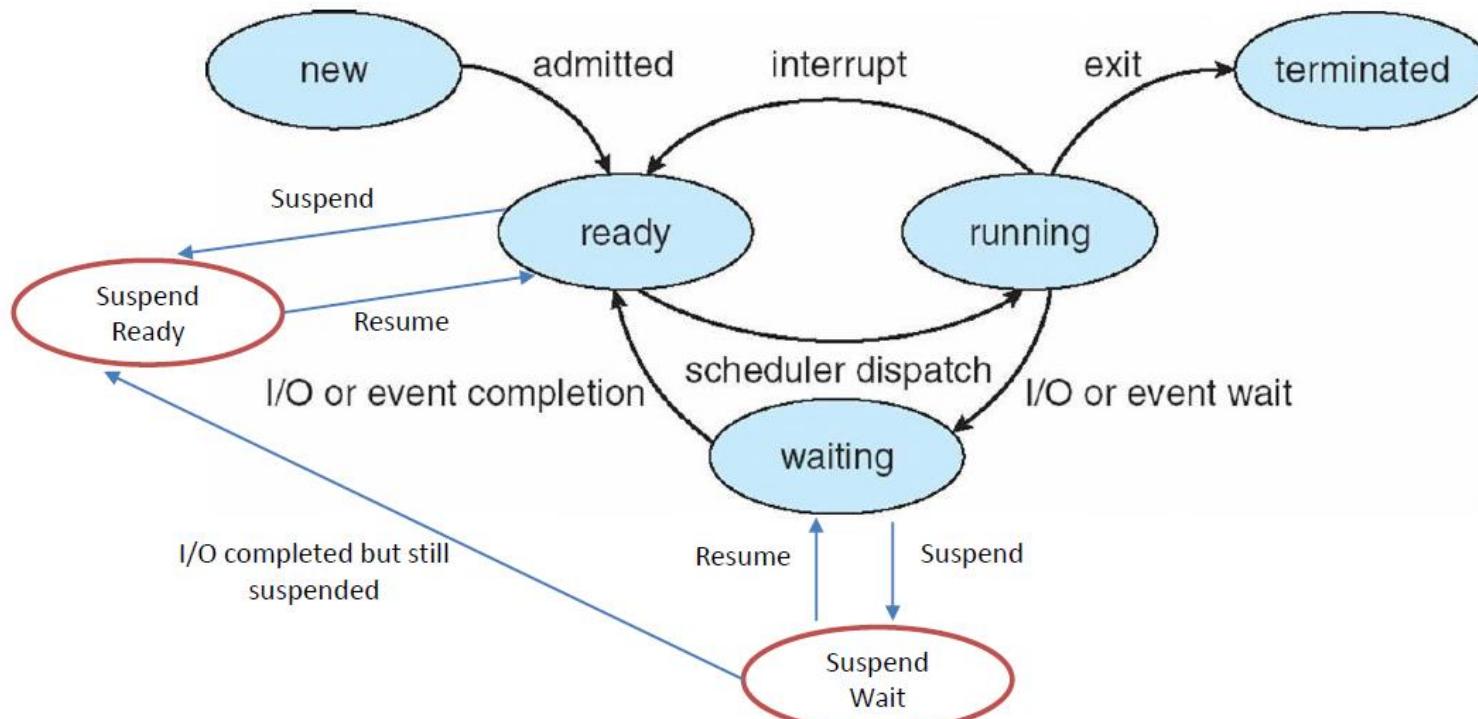
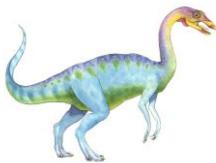




Diagram of Process State





Process Control Block (PCB)

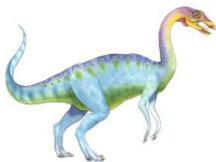
Information associated with each process (also called **task control block**)

- **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



Fig:- Process Control Block (PCB)

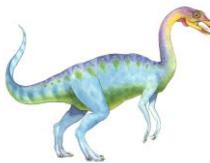




Process Control Block (PCB)

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**
- It contains many pieces of information associated with a specific process, including these:

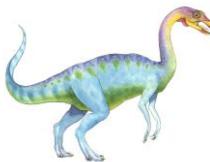




Process Control Block (PCB)

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. **Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.**
- **CPU-scheduling information.** This information includes a **process priority**, **pointers to scheduling queues**, and any **other scheduling parameters**.





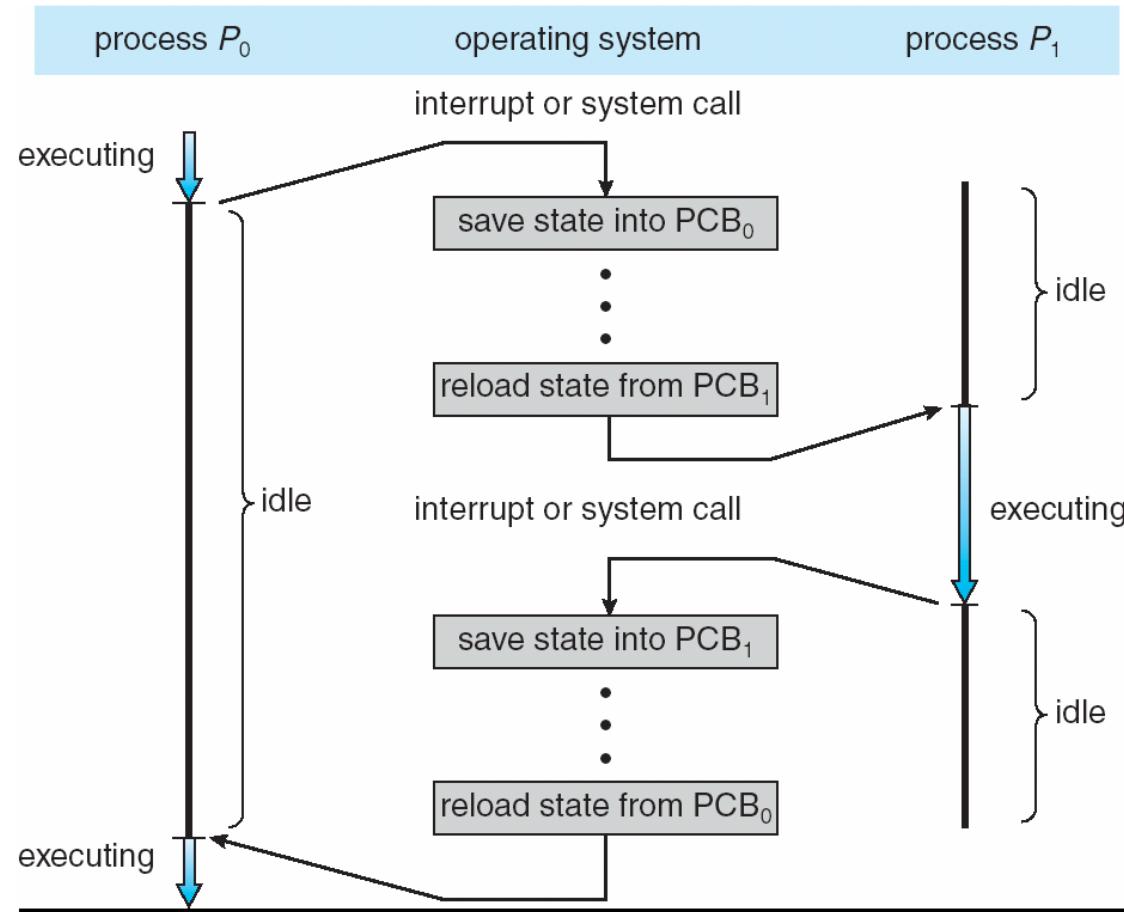
Process Control Block (PCB)

- **Memory-management information.** This information may include such items as the **value of the base and limit registers** and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.





CPU Switch From Process to Process





Process Scheduling

- **WHY:**

- Several Processes competing at a time to get the CPU for their execution

- **Scheduling**

- strategy and methods used by OS to decide which process is going to be allocated to CPU next among the several process in the queue for CPU time

- The objective **of multiprogramming** is

- to have some process running at all times, to maximize CPU utilization

- Time sharing multiprogramming system





Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process **gives** up the CPU under two conditions:
 - I/O request
 - After N units of time have elapsed
- Once a process gives up the CPU it is added to the **ready queue**
- **Process scheduler** selects among available processes for next execution on CPU





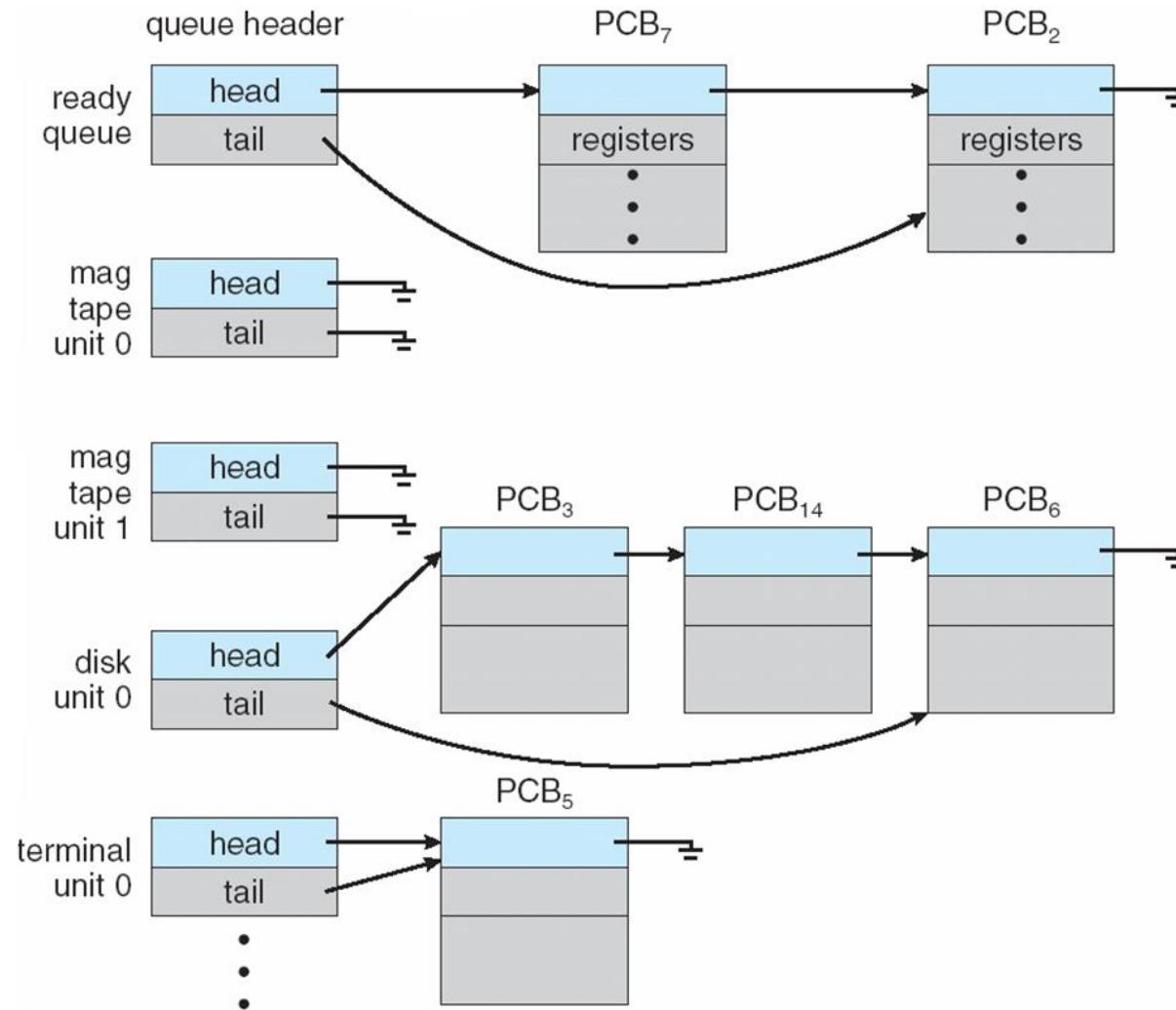
Process Scheduling

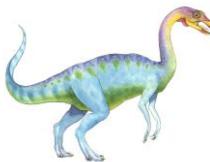
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing 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





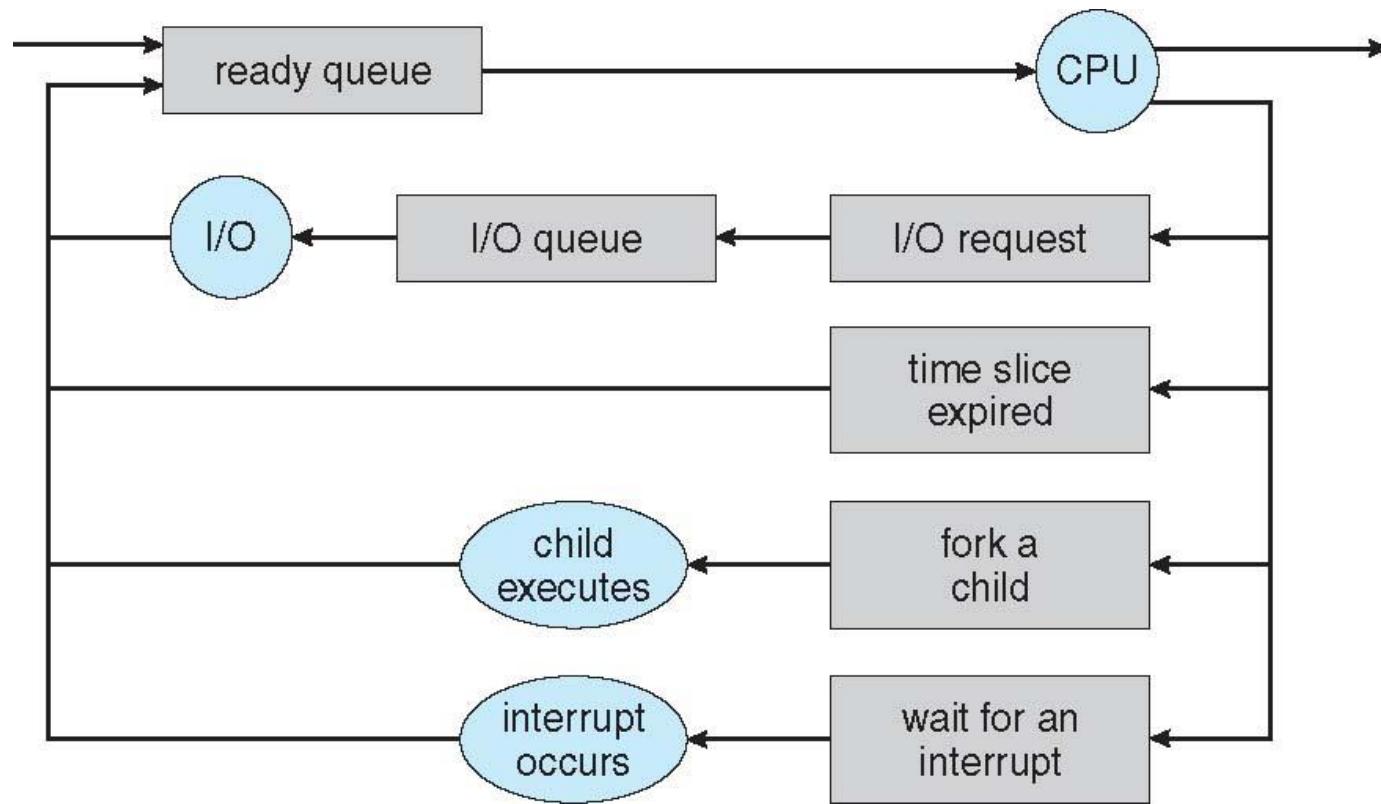
Ready Queue And Various I/O Device Queues





Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows





Schedulers

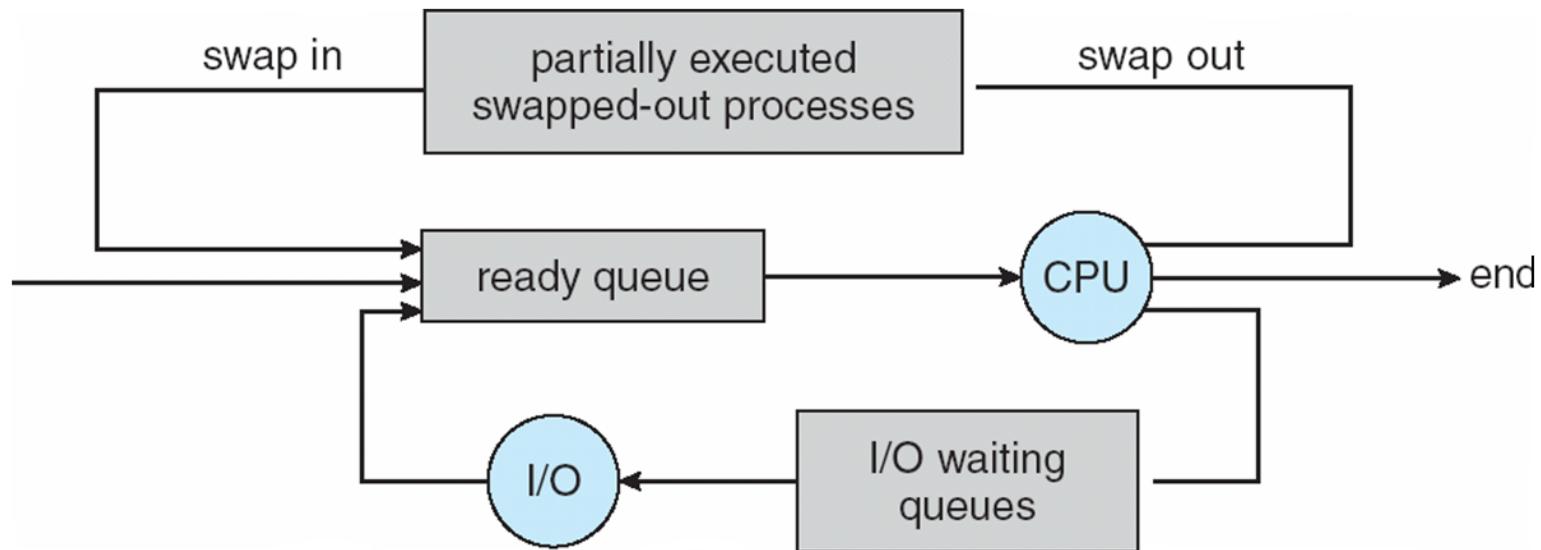
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which 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**

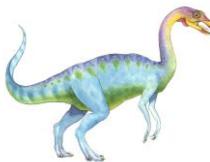




Context Switch

- 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

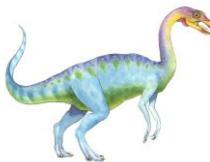




Operations on Processes

- During the course of execution, a process may create several new processes
 - The creating process is called a parent process, and
 - the new processes are called the children of that process
- System must provide mechanisms for:
 - process creation,
 - process termination,





Process Creation

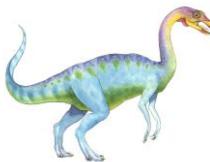
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Process identified and managed via a process identifier (pid)
 - PID is an integer number
 - The PID provides a unique value for each process in the system, and
 - it can be used as an index to access various attributes of a process within the kernel.
- **getpid()**

This function returns the process identifiers of the calling process.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void); // this function returns the process identifier (PID)
pid_t getppid(void); // this function returns the parent process identifier (PPID)
```

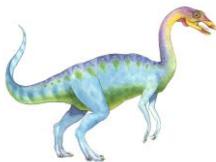




Process Creation

```
sandhya@telnet:~/DSEOS2022/process$ cat pid.c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
    pid_t getpid(void); //return PID
    pid_t getppid(void); //return PPID
    printf("PID is %d",getpid());
    printf("\n PPID is %d",getppid());
    return 0;
}
sandhya@telnet:~/DSEOS2022/process$ ./a.out
PID is 2682
PPID is 2554sandhya@telnet:~/DSEOS2022/process$ ps
  PID  TTY          TIME CMD
2554 pts/0    00:00:00 bash
 2683 pts/0    00:00:00 ps
sandhya@telnet:~/DSEOS2022/process$
```





Process Creation

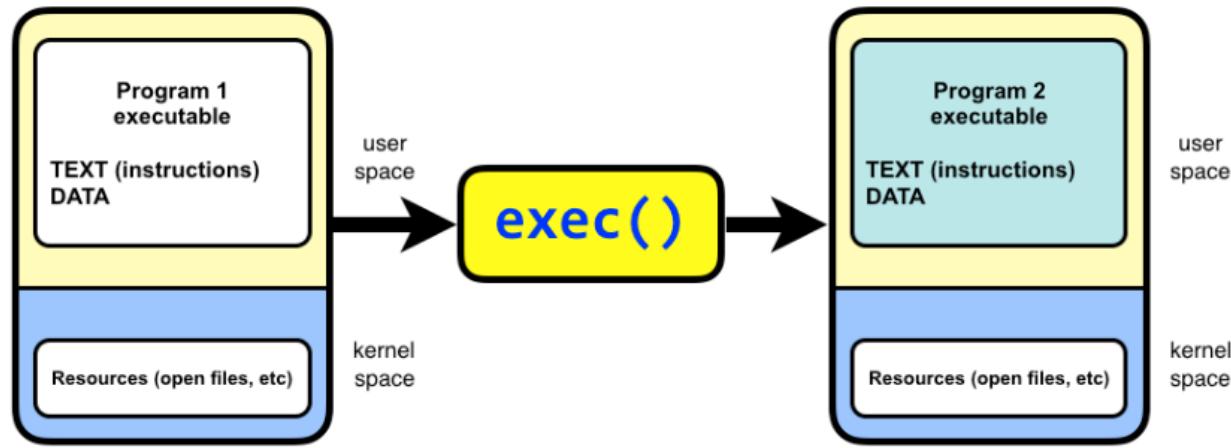
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
 - ▶ ***Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes***

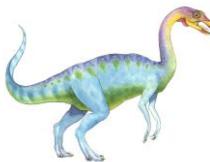




Process Creation

- When a process creates a new process, two possibilities for execution exist:
 - Parent and children execute concurrently
 - Parent waits until children terminate
- There are also two address-space possibilities for the new process:
 - 1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
 - 2. The child process has a new program loaded into it.





Process Creation: fork()

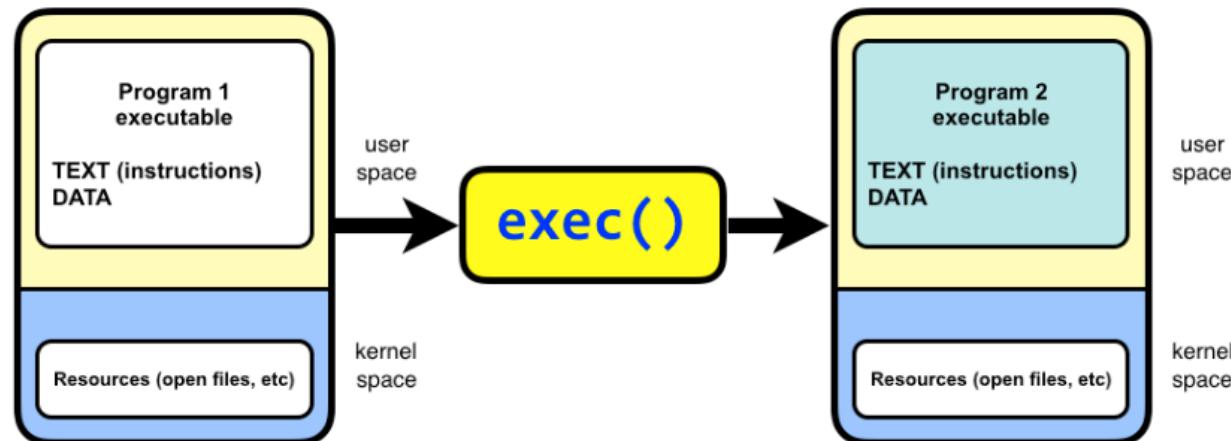
- A new process is created by the **fork()** system call
 - A new process is created by calling fork.
 - This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process.
 - The new process is almost identical to the original, executing the same code **but with its own data space, environment, and file descriptors**.
 - After a new child process is created, both processes will execute the **next instruction following the fork() system call**.





Process Creation: fork()

- Different values returned by fork():
 - **Negative Value**: creation of a child process was unsuccessful.
 - **Zero**: Returned to the newly created child process.
 - **Positive value**: Returned to parent or caller. The value contains process ID of newly created child process.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program
- The exec() system call loads a binary file into memory and starts its execution
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child





Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    int sharedVar = 10;
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == -1) {
        perror("fork");
        return 1;
    }

    if (child_pid == 0) {
        // This code will be executed by the child process
        sharedVar = 20;
        printf("Child - sharedVar: %d\n", sharedVar);
    }
    else {
        // This code will be executed by the parent process
        printf("Parent - sharedVar: %d\n", sharedVar);
    }

    return 0;
}
```

Output

- If Parent process executes first:

Parent - sharedVar: 10
Child - sharedVar: 20

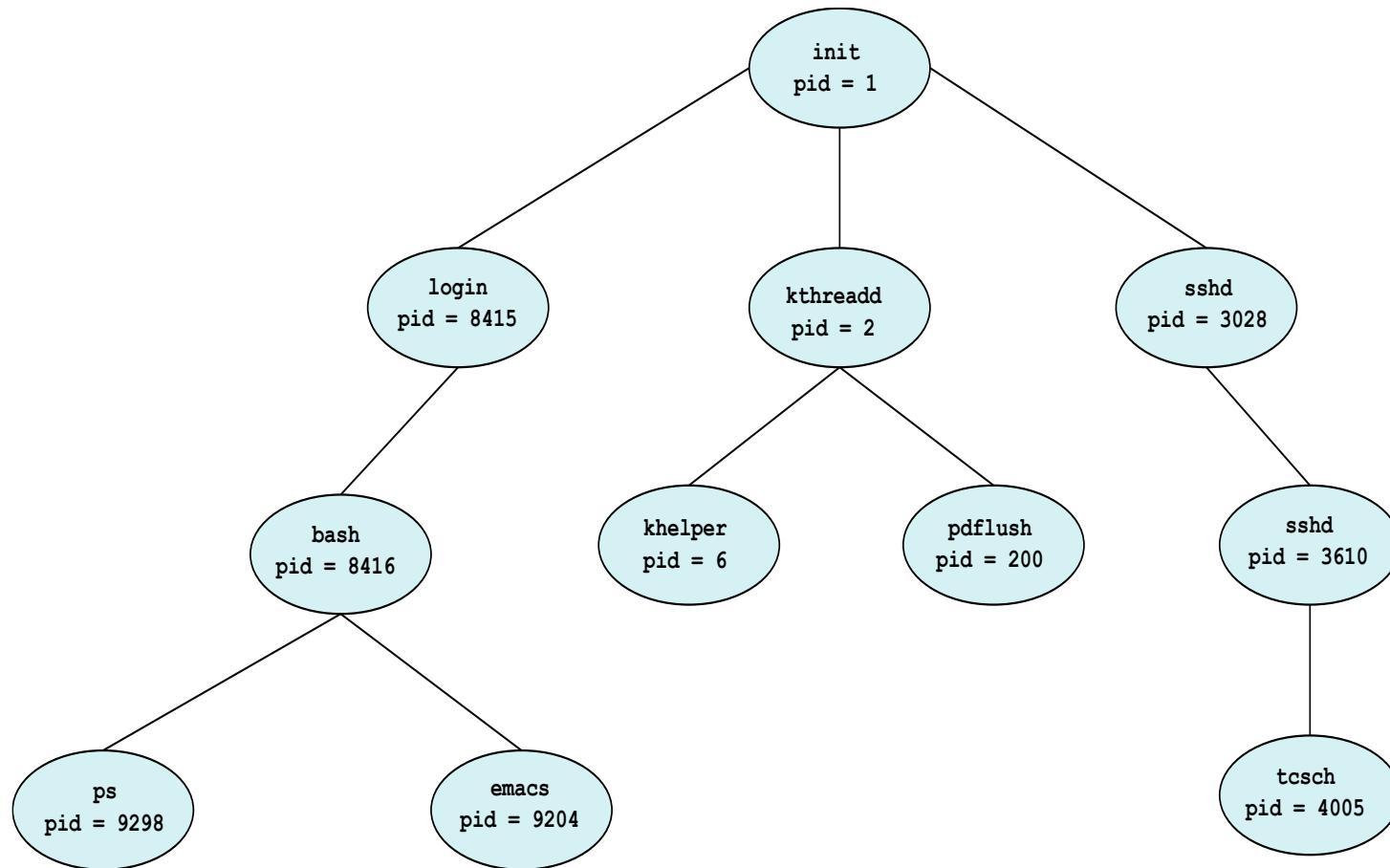
- If Child process executes first:

Child - sharedVar: 20
Parent - sharedVar: 10





A Tree of Processes in Linux

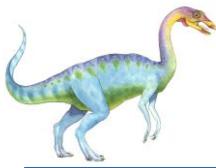




A Tree of Processes in Linux

- The **init process** (which always has a pid of 1) serves as the root parent process for all user processes.
- The **kthreadd process** is responsible for creating additional processes that perform tasks on behalf of the kernel
- The **sshd process** is responsible for managing clients that connect to the system by using ssh (Secure Shell)
- The **login process** is responsible for managing clients that directly log onto the system.
- In this example, a client has logged on and is using the **bash** shell, which has been assigned pid 8416.
- Using the **bash** command-line interface, this user has created the process **ps** as well as the **emacs** editor.
- Parent waits until children terminate
- **Pdflush: a set of kernel threads which are responsible for writing the dirty pages to disk**





A Tree of Processes in Linux

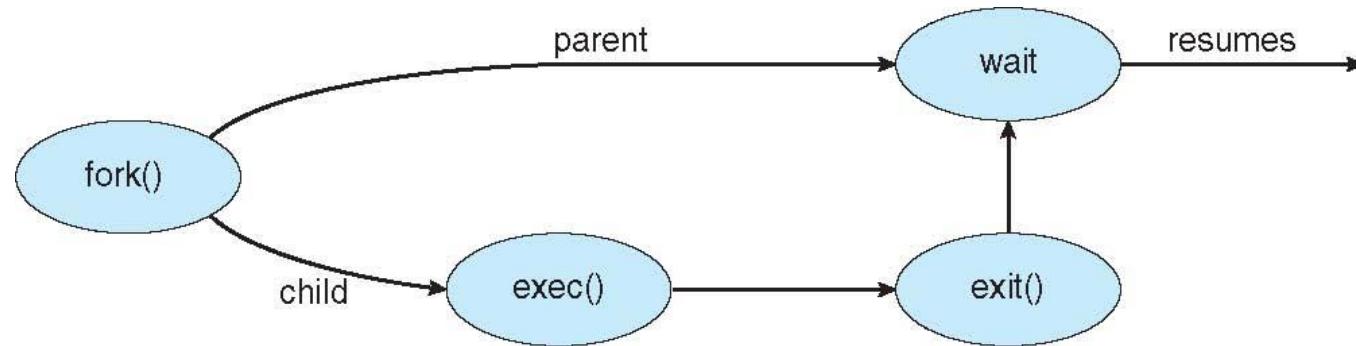
- On UNIX and Linux systems, the `ps` command is used to list all the user process
 - For example, `ps -el`
 - ▶ will list complete information for all processes currently active in the system.

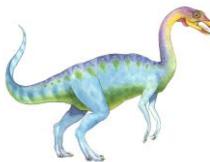




Process Creation (Cont.)

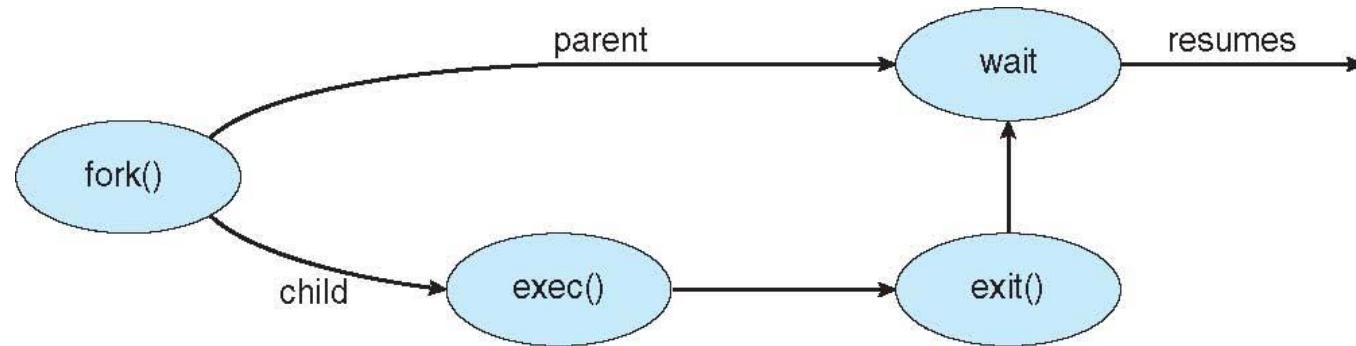
- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - `fork()` system call creates new process
 - `exec()` system call used after a `fork()` to replace the process' memory space with a new program





Process 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

1. 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
2. A process can cause the termination of another process via an appropriate system call
 - Usually, such a system call can be invoked only by the parent of the process.
 - Otherwise, users could arbitrarily kill each other's jobs.
 - Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

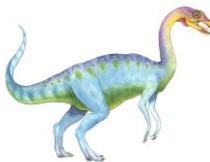




Process Termination

- 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
- 1. Some operating systems do not allow child to exists 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.





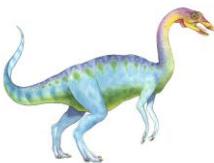
Process Termination

- 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**
- Linux and UNIX address this scenario by assigning the **init process** as the new parent to orphan processes





Example

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    if (fork() == 0)
        printf("HC: hello from child\n");
    else
    {
        printf("HP: hello from parent\n");
        wait(NULL);
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");

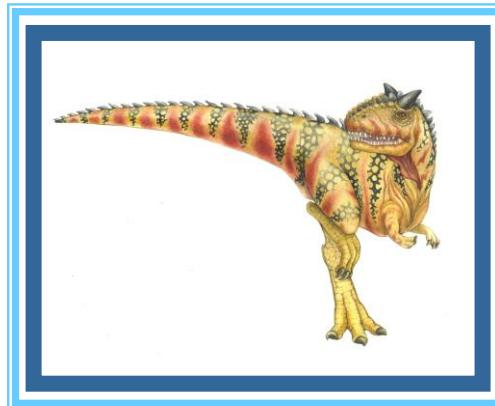
    return 0;
}
```

One Possibility of Output

HC: hello from child
Bye
HP: hello from parent
CT: child has terminated
Bye



CPU Scheduling





Objectives

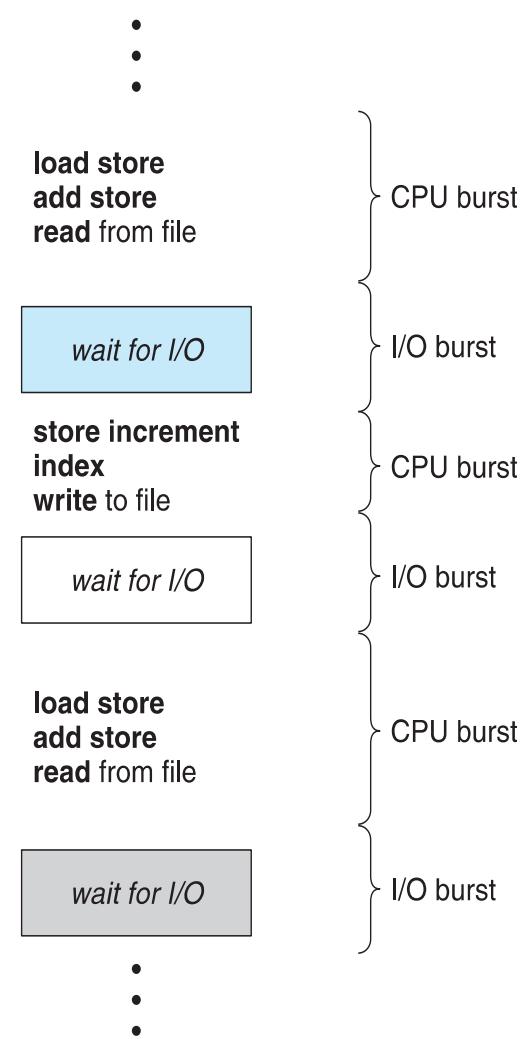
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
 - ▶ To understand basic concepts, CPU / i/o burst cycle, cpu schedulers, dispatcher
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To describe various CPU-scheduling algorithms





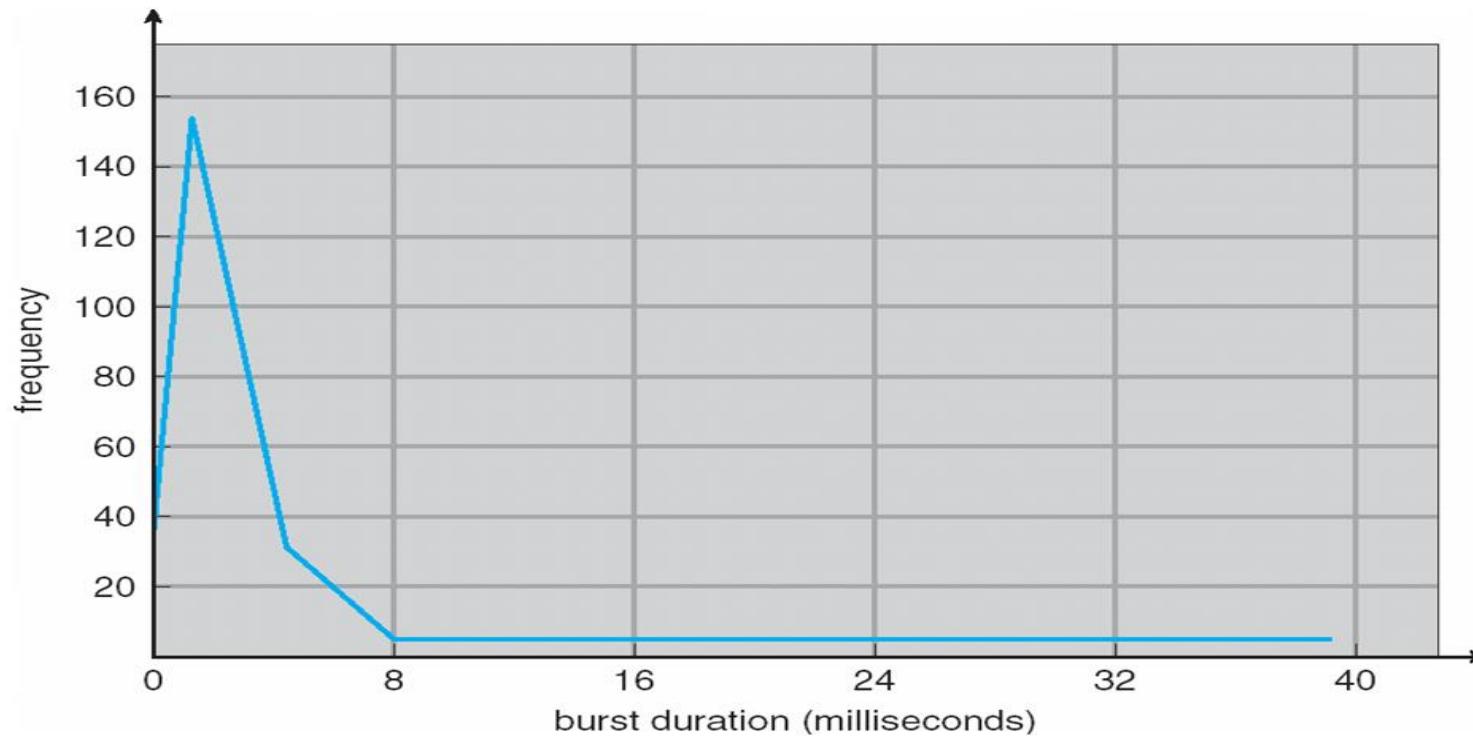
Basic Concepts

- The success of CPU scheduling depends on an observed property of processes:
 -
 -
 -
- 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**
 - *Size of cpu and i/o burst may vary for different processes.*
 - *CPU burst distribution is of main concern*
 - *I/O burst size does not affect the cpu scheduling*





Histogram of CPU-burst Times



- ✓ The curve is generally characterized as exponential or hyperexponential
 - 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.





CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 - 1. Switches from running to waiting state
 - ▶ for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process
 - 2. Switches from running to ready state
 - when an interrupt occurs
 - 3. Switches from waiting to ready
 - at completion of I/O
 - 4. Terminates



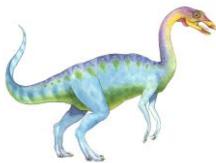


Non preemptive

- For situations 1 and 4 , no choice in terms of scheduling (a new process(if exists in ready queue) must be selected for execution) ----
→ **non preemptive/ cooperative**
- There is a choice for situations 2 and 3 → **preemptive**

- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU
 - Either **by terminating**
 - Or **by switching to the waiting state.**
 - ▶ Ex:- used by Microsoft Windows 3.x





Problems of preemptive

Preemptive scheduling can result in race condition when data are shared among several processes

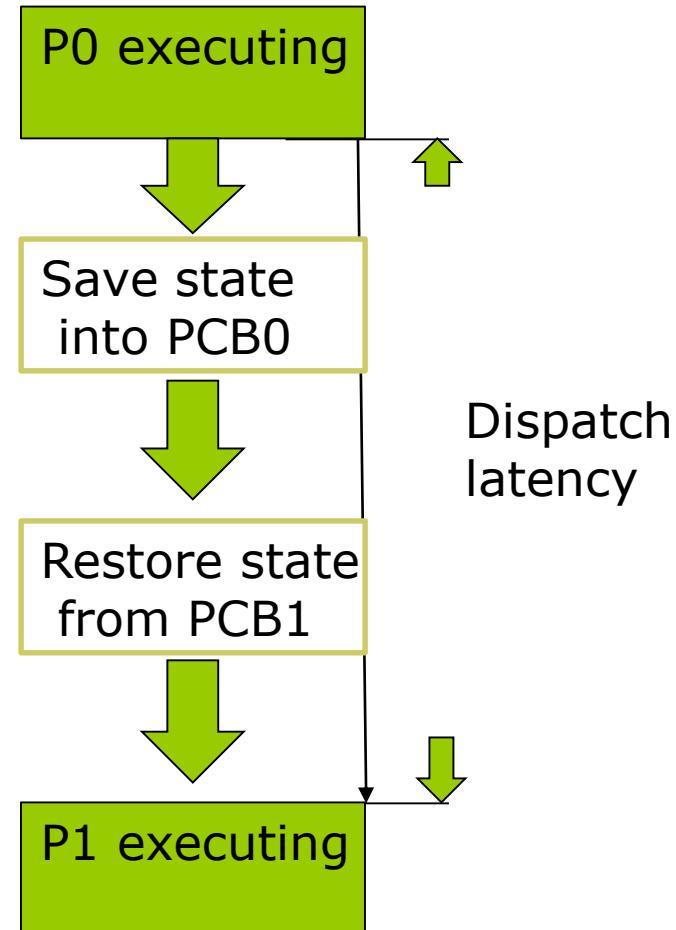
- Consider access to shared data
- Consider preemption while in kernel mode
- Consider interrupts occurring during crucial OS activities
 - While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state
- Preemptive introduced by windows 95
- All modern OS(Windows, Mac OS X, Linux, Unix uses preemptive scheduling algorithms.





Dispatcher

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





Scheduling Criteria

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

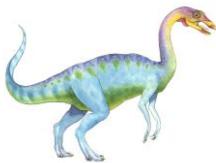




Scheduling Algorithm Optimization Criteria

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





Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.





First- Come, First-Served (FCFS) Scheduling

- It is a non preemptive policy

<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: $(0 + 24 + 27)/3 = 17$





First- Come, First-Served (FCFS) Scheduling

- It is a non preemptive policy

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

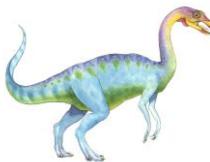
- Suppose that the processes arrive in the order: P_1, P_2, P_3

The Gantt Chart for the schedule is:



- TAT for $P_1 = 24$; $P_2 = 27$; $P_3 = 30$
- Average TAT: $(24+27+30)/3$





FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

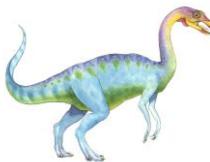
$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



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





FCFS

PID	A	B	C	D	E
AT	3	5	0	5	4
BT	4	3	2	1	3

GANTT CHART



PID	A	B	C	D	E
AT	3	5	0	5	4
BT	4	3	2	1	3
WT	0	5	0	8	3
TAT	4	8	2	9	6





Shortest-Job-First (SJF) Scheduling

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

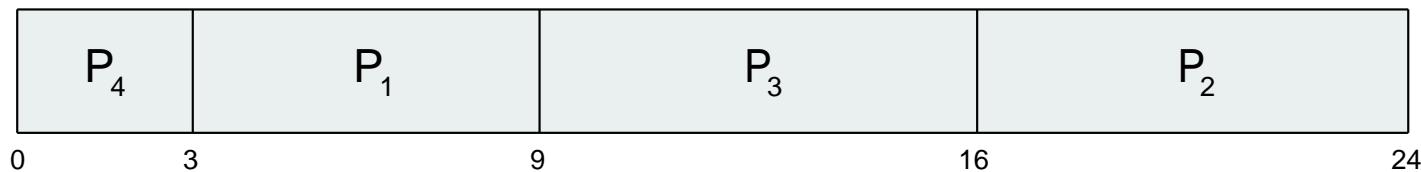




Example of SJF(non preemptive)

PID	BT
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



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

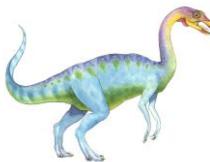




Determining Length of Next CPU Burst

- Preemptive version called
shortest-remaining-time-first





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 =



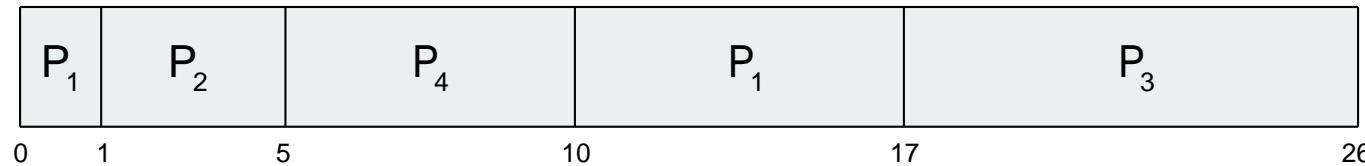


Example of Shortest-remaining-time-first

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

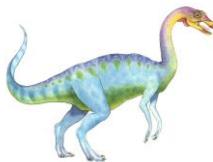
<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*



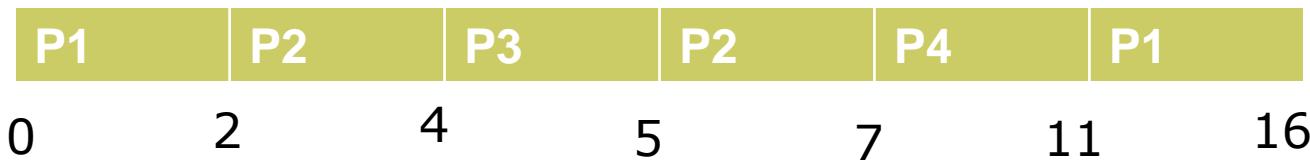
- Average waiting time = $[(10-0-1)+(1-1-0)+(17-2-0)+5-3-0]/4 = 26/4 = 6.5$ msec
- WT=FinalCUPALLOCTIME-ARRIVALTIME-PREVIOUSEXECUTIONTIME**





SJF(preemptive(SRTF))

PID	P1	P2	P3	P4
AT	0	2	4	5
BT	7	4	1	4



PID	P1	P2	P3	P4
AT	0	2	4	5
BT	7	4	1	4
WT	11-0-2=9	5-2-2=1	4-4-0=0	7-5-0=2
TAT	9+7=16	1+4=5	0+1=1	2+4=6





Practice Exercises

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

- What is the average turnaround time for these processes with the FCFS scheduling algorithm?
- What is the average turnaround time for these processes with the SJF scheduling algorithm?
- The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be called future-knowledge scheduling.





Practice Exercises

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

What is the average turnaround time for these processes with the FCFS scheduling algorithm?

FCFS





Practice Exercises

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

What is the average turnaround time for these processes with the SJF scheduling algorithm?

SJF





Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1



The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. **This algorithm could be called future-knowledge scheduling.**





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

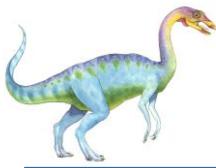
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



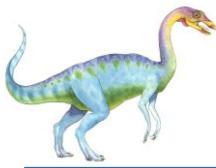
- Average waiting time = 8.2 msec



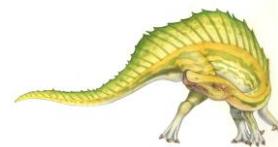


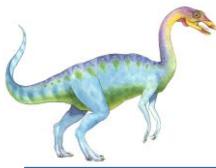
Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105



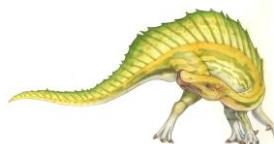


Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105



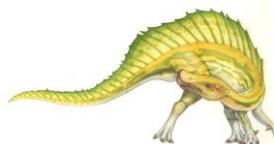


PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
P*	3	1	4	5	2





PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
P*	3	1	4	5	2





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

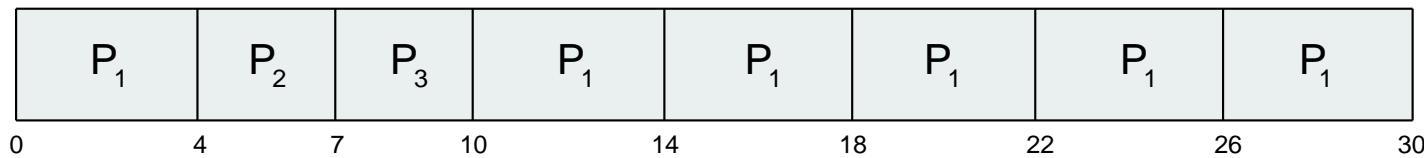




Example of RR with Time Quantum = 4

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

- The Gantt chart is:

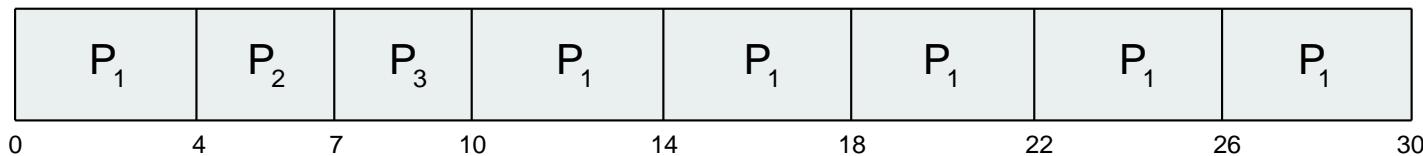




Example of RR with Time Quantum = 4

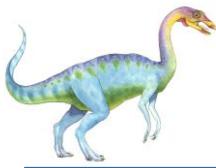
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

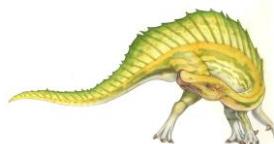


- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



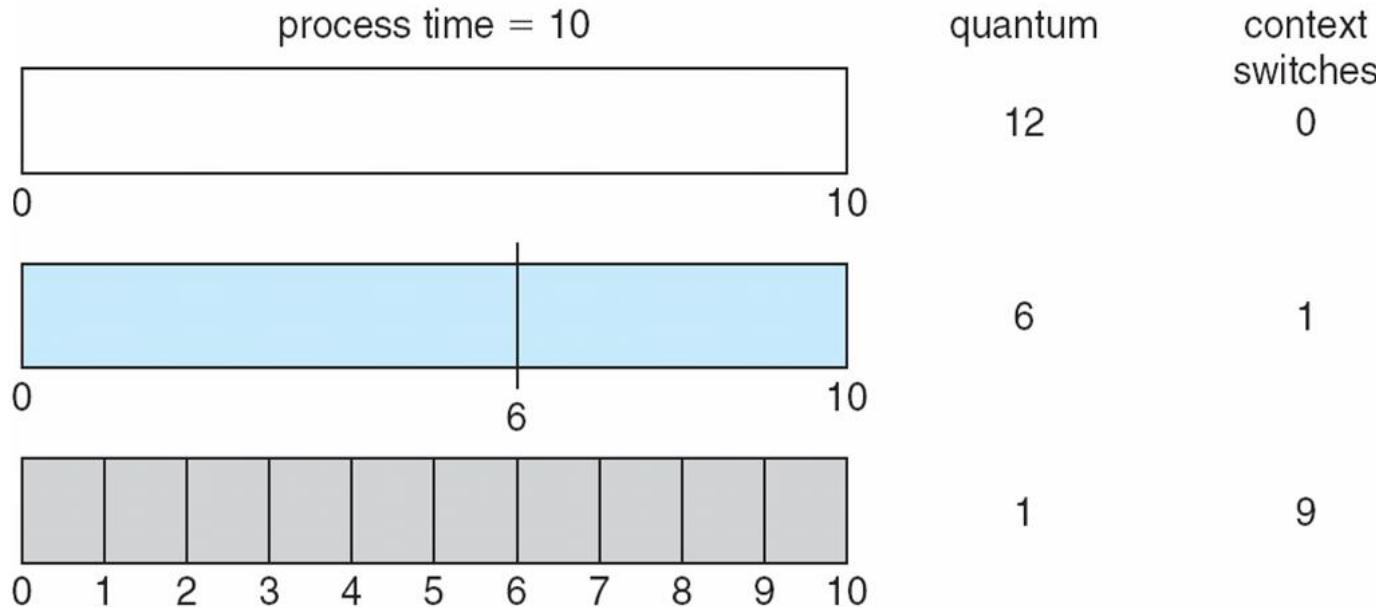


PID	P1	P2	P3	P4	P5
BT	2	1	8	4	5
P*	2	1	4	2	3



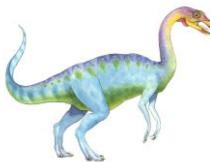


Time Quantum and Context Switch Time

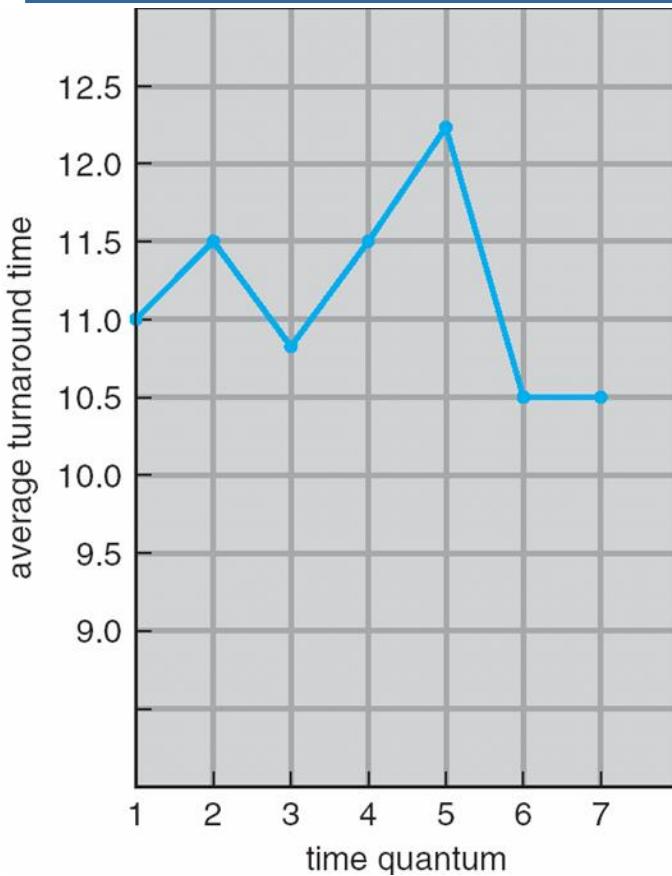


- ✓ Thus, we want the time quantum to be large with respect to the context switch time.
- ✓ If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
- ✓ In practice, most modern systems have time quanta ranging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





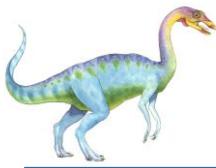
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q





Multilevel Queue Scheduling

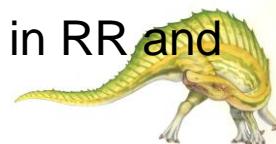
Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.





Multilevel Queue Scheduling

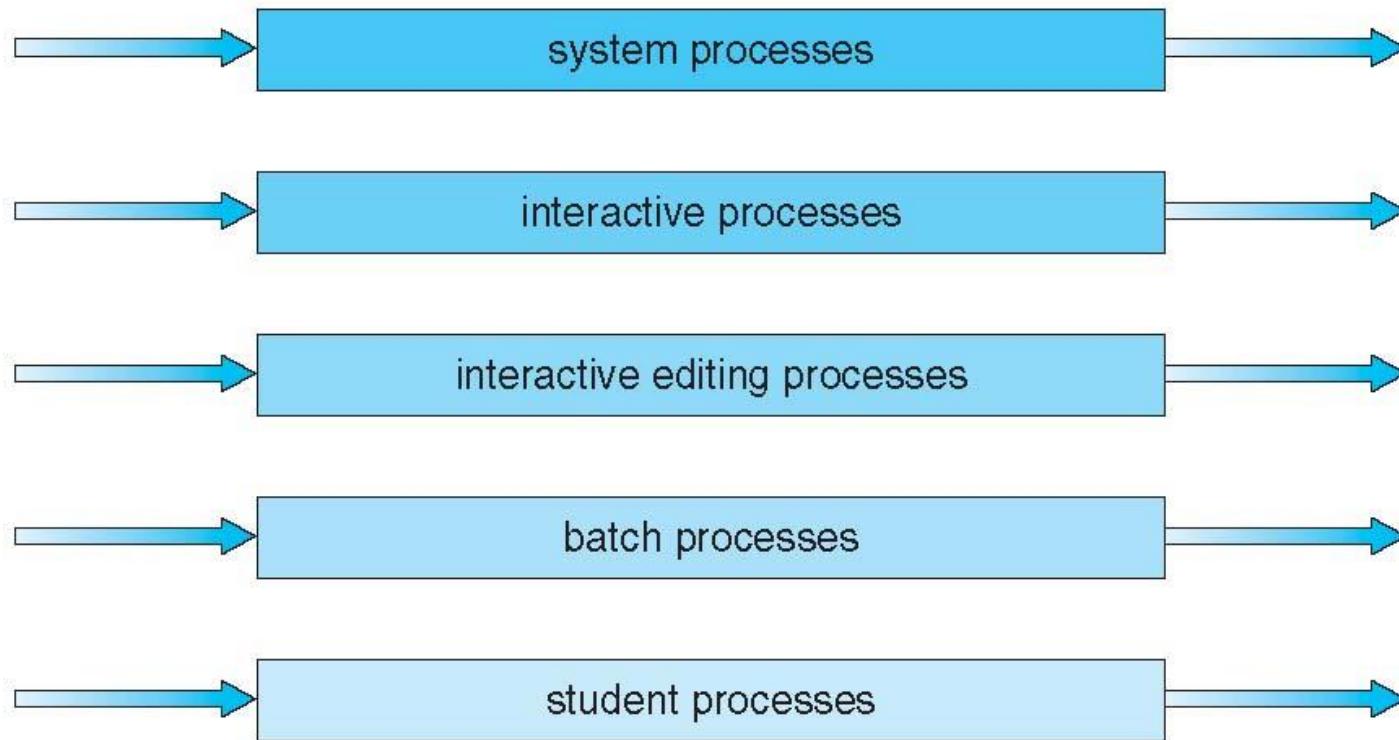
- Ready queue is partitioned into separate queues, eg:
 - **foreground** (interactive)
 - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues:
 - Fixed priority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





Multilevel Queue Scheduling

highest priority



lowest priority





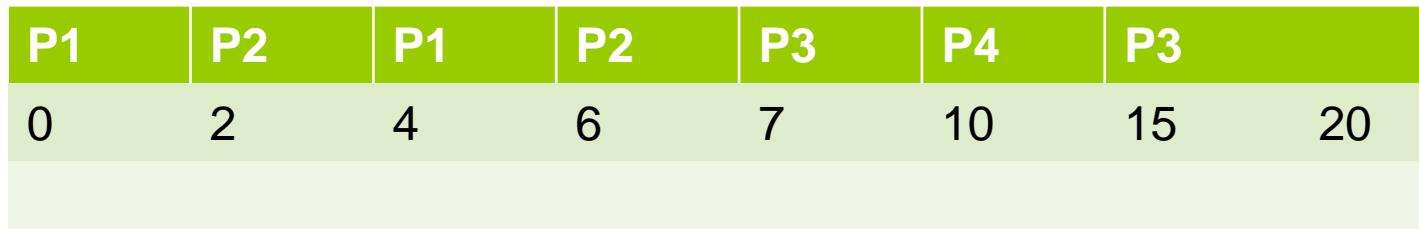
Multilevel Queue Scheduling

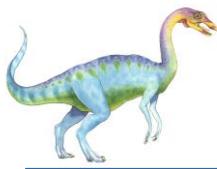
Consider below data of four processes under multilevel queue scheduling, Q No. denotes the queue of the process

PID	Arrival Time	Burst Time	Q No.
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

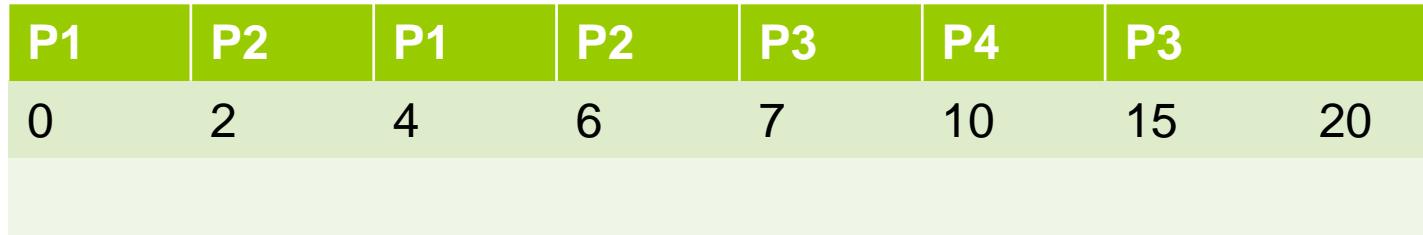
Priority of queue 1 is greater than queue 2. Queue 1 uses RR (TQ=2) and queue 2 uses FCFS.

Draw the Gantt chart for above data and Calculate AWT, TAT





Multilevel Queue Scheduling





Multilevel Feedback Queue

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





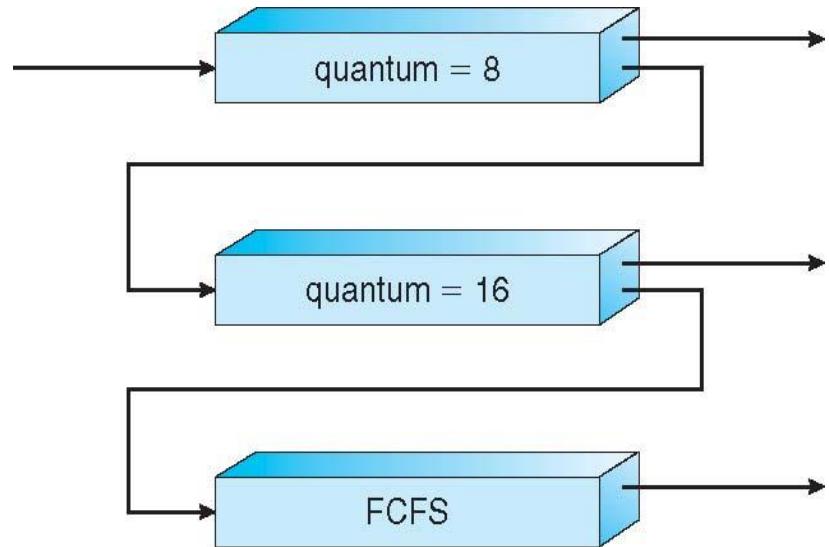
Example of Multilevel Feedback Queue

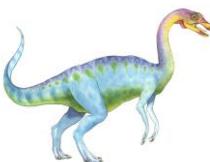
■ Three queues:

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

■ Scheduling

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





Example

- Consider the following set of processes with the length of the CPU burst time given in millisecond
- The processes are assumed to have arrived in the order of P1,P2,P3,P4, P5, all at time 0.
- Draw four Gantt chart illustrating the executing of these process using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority) and RR ($q=1$) scheduling.

PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
Priority	3	1	3	4	2

Algo.	FCFS	SJF	PRIORITY	RR
AWT				
ATT				





FCFS

PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
Priority	3	1	3	4	2





SJF

PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
Priority	3	1	3	4	2





NP Priority

PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
Priority	3	1	3	4	2

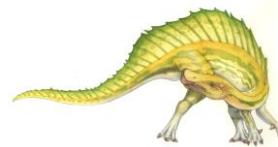
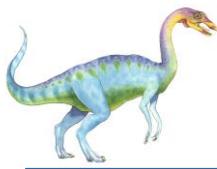




RR, q=1

PID	P1	P2	P3	P4	P5
BT	10	1	2	1	5
Priority	3	1	3	4	2







Scheduling Algorithm Optimization Criteria

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





Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms

- **Deterministic modeling**
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

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



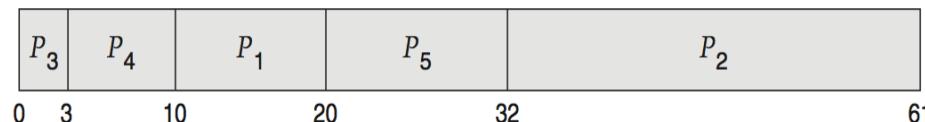


Deterministic Evaluation

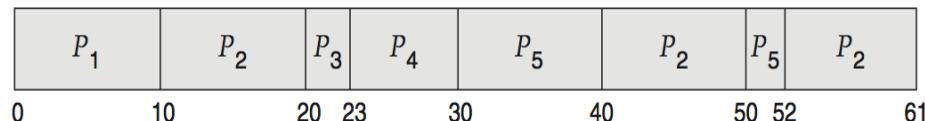
- For each algorithm, calculate **minimum average waiting time**
- Simple and fast, but requires exact numbers for input, applies only to those inputs
 - FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





Queueing Models

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





Little's Formula

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





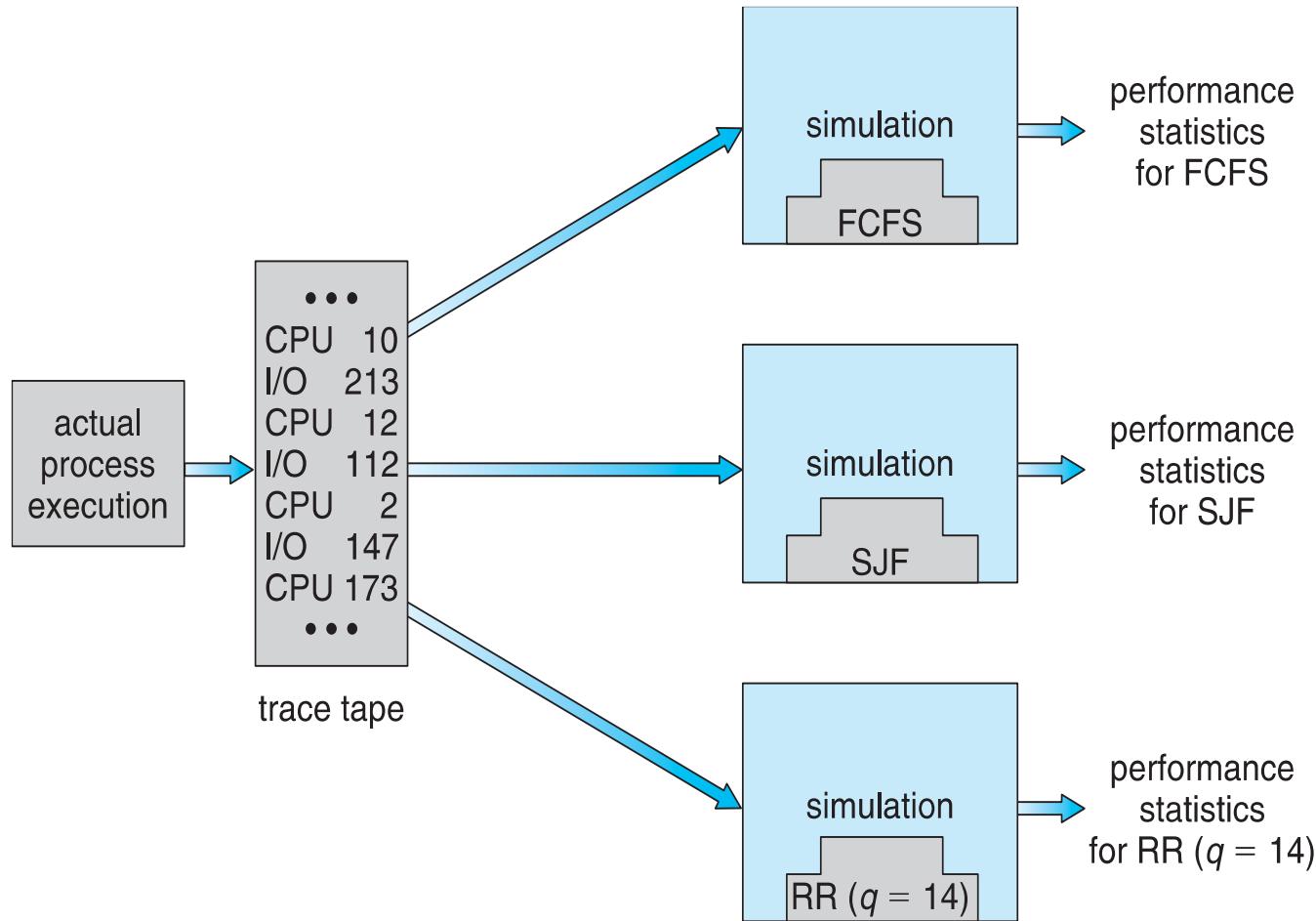
Simulations

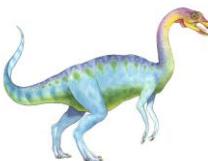
- Queueing models limited
- **Simulations** more accurate
 - Programmed model of computer system
 - Clock is a variable
 - Gather statistics indicating algorithm performance
 - Data to drive simulation gathered via
 - ▶ Random number generator according to probabilities
 - ▶ Distributions defined mathematically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





Evaluation of CPU Schedulers by Simulation





Implementation

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





Exercises

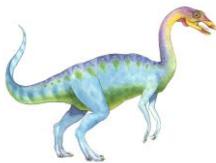
Consider the following set of processes, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
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, nonpreemptive 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)?

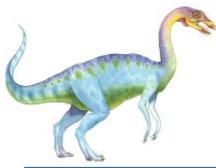




Exercises

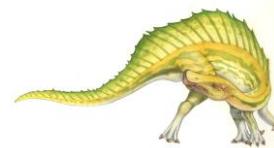
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	
P_1	2	2	
P_2	1	1	
P_3	8	4	
P_4	4	2	
P_5	5	3	

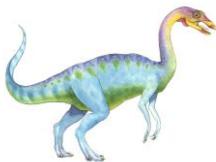




Exercises

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

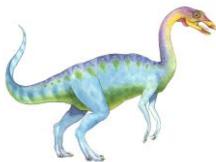




Exercises

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	
P_1	2	2	
P_2	1	1	
P_3	8	4	
P_4	4	2	
P_5	5	3	





Exercises

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	
P_1	2	2	
P_2	1	1	
P_3	8	4	
P_4	4	2	
P_5	5	3	





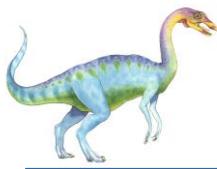
Exercises

The following processes are being scheduled using a preemptive, roundrobin 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 *Pidle*). 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

- Show the scheduling order of the processes using a Gantt chart.
- What is the turnaround time for each process?
- What is the waiting time for each process?

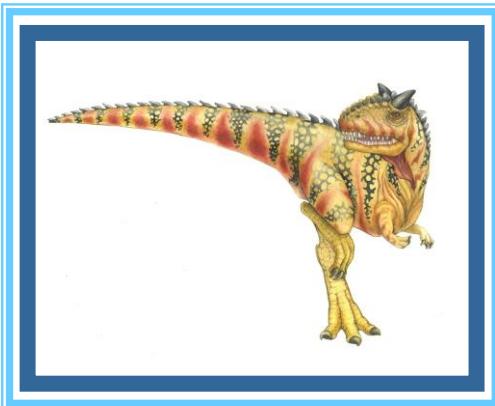




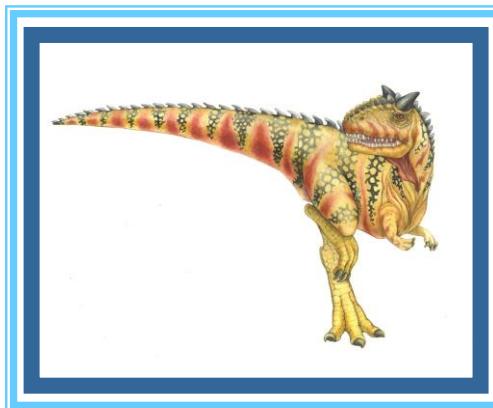
Summary



End of Chapter 6



Interprocess Communication





Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- A process is *independent* if it **cannot affect or be affected** by the other processes executing in the system.
 - Any process that does not share data with any other process is independent.
- A process is *cooperating* if it **can affect or be affected** by the other processes executing in the system.
 - Any process that shares data with other processes is a cooperating process.
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience





Cooperating Processes

- Advantages of process cooperation

- Information sharing

- ▶ Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- Computation speed-up

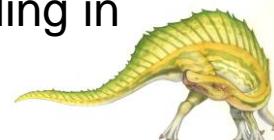
- ▶ If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores

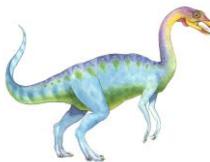
- Modularity

- ▶ We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

- Convenience

- ▶ Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel

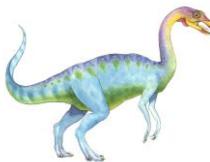




Interprocess Communication

- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them **to exchange data and information**
- Two models of IPC
 - **Shared memory**
 - ▶ a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region
 - **Message passing**
 - ▶ Communication takes place by means of messages exchanged between the cooperating processes
 - ▶ Message passing is useful for exchanging smaller amounts of data, because **no conflicts need be avoided**.
 - ▶ Message passing is also easier to implement in a distributed system than shared memory.





Interprocess Communication

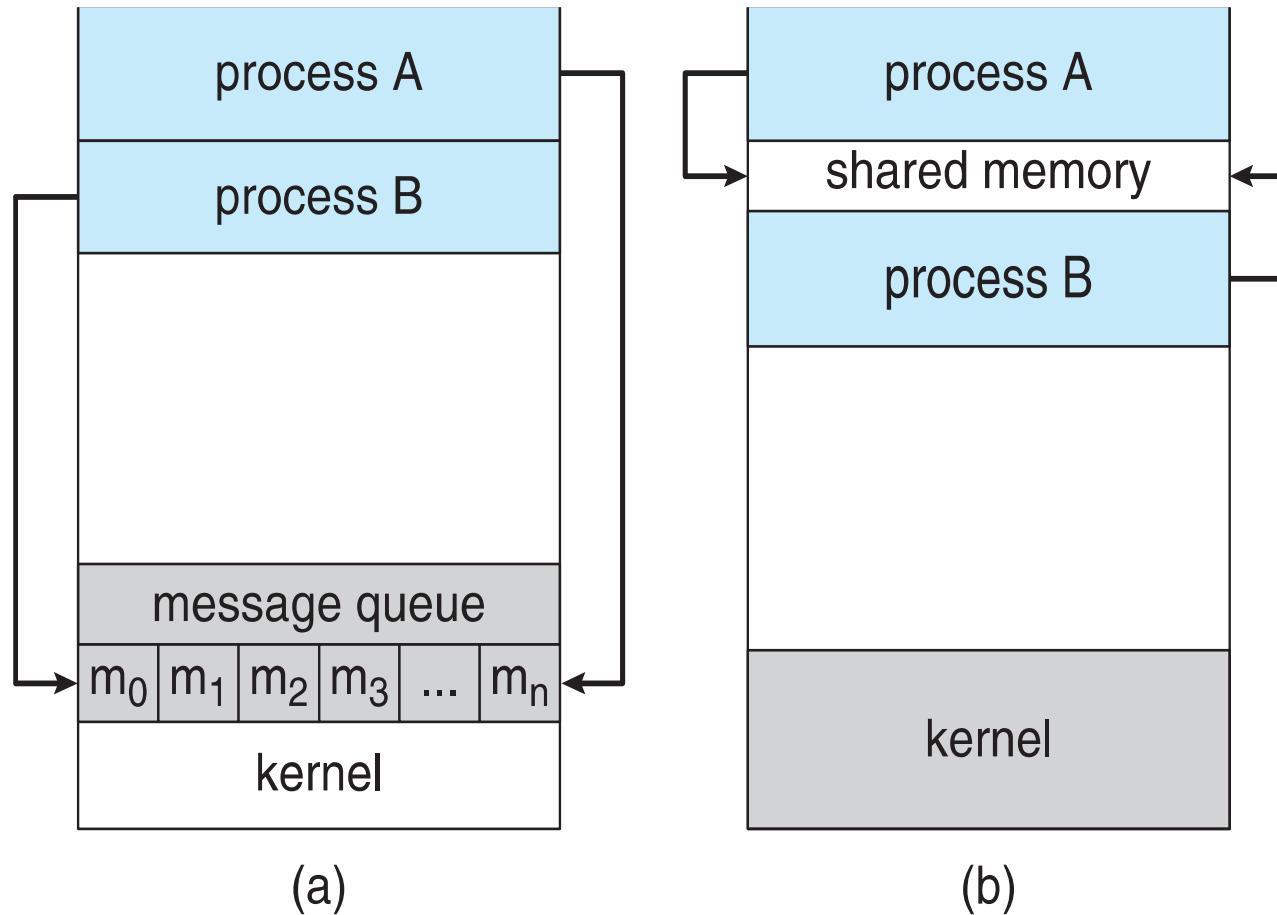
- Shared memory can be **faster** than message passing, since message-passing systems are typically implemented using **system calls** thus require the **more time-consuming task of kernel intervention**.
- In shared-memory systems, system calls are required **only to establish shared memory regions**. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.
- Recent research on systems with **several processing cores** indicates that **message passing provides better performance** than shared memory on such systems. **Shared memory suffers from cache coherency issues**, which arise because shared data migrate among the several caches. As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.





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
 - For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
 - The producer–consumer problem also provide metaphor for the client-server paradigm. Ex. a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.
 - **Unbounded-buffer** places no practical limit on the size of the buffer
 - ▶ The consumer may have to wait for new items, but the producer can always produce new items
 - **Bounded-buffer** assumes that there is a fixed buffer size
 - ▶ the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full





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.
- Synchronization is discussed in great details in Chapter 5.

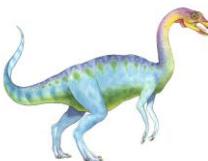




Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
 - 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
 - If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult
 - Conversely, variable-sized messages require a more complex system level implementation, but the programming task becomes simpler.





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
- This scheme exhibits **symmetry** in addressing; that is, both the sender process and the receiver process must name the other to communicate.
- In **asymmetry** addressing, only the sender names the recipient; the recipient is not required to name the sender
 - **send(P, message)**—Send a message to process *P*.
 - **receive(id, message)**—Receive a message from any process.





Indirect Communication

- **Indirect communication:** the messages are sent to and received from **mailboxes**, or **ports**
 - A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
 - ▶ Each mailbox has a unique identification.
 - ▶ A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.
- 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

- 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

- 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.

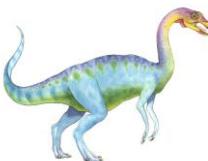




Synchronization

- 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 continues
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message
- Different combinations of send() and receive() are possible
 - If both send and receive are blocking, we have a **rendezvous**





Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue
 - Queue of messages attached to the link.
- implemented in one of three ways
 - **Zero capacity** – The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message (rendezvous)
 - **Bounded capacity** – finite length of n messages. the sender can continue execution without waiting. Sender must wait if link full
 - **Unbounded capacity** – infinite length. Sender never waits
- The zero-capacity case is sometimes referred to as a message system with no buffering.
- The other cases are referred to as systems with automatic buffering





Summary

- A process is a program in execution. As a process executes, it changes state.
- The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated.
- Each process is represented in the operating system by its own process control block (PCB).
- There are two major classes of queues in an operating system: I/O request queues and the ready queue.
- Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU.
- Short-term (CPU) scheduling is the selection of one process from the ready queue.
- Operating systems must provide a mechanism for parent processes to create new child processes
- There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.



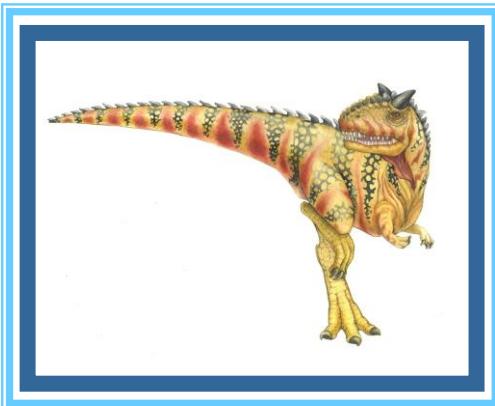


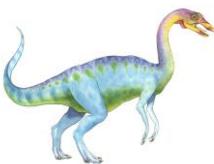
Summary

- The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an interprocess communication mechanism to communicate with each other.
- In a shared-memory system, the responsibility for providing communication rests with the application programmers;
 - the operating system needs to provide only the shared memory.
- The message-passing method allows the processes to exchange messages.
 - The responsibility for providing communication may rest with the operating system itself.
 - These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.



Threads





Threads

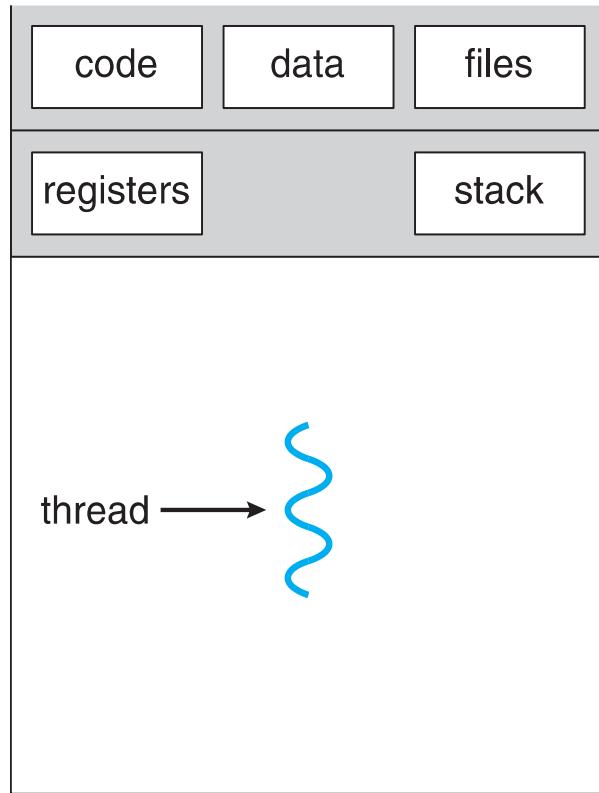
- A thread is a basic unit of CPU utilization;
 - it comprises
 - ▶ a **thread ID**,
 - ▶ a **program counter**,
 - ▶ a **register set, and**
 - ▶ a **stack**.
 - It shares with other threads belonging to the same process its **code section, data section, and other operating-system resources**, such as open files and signals.



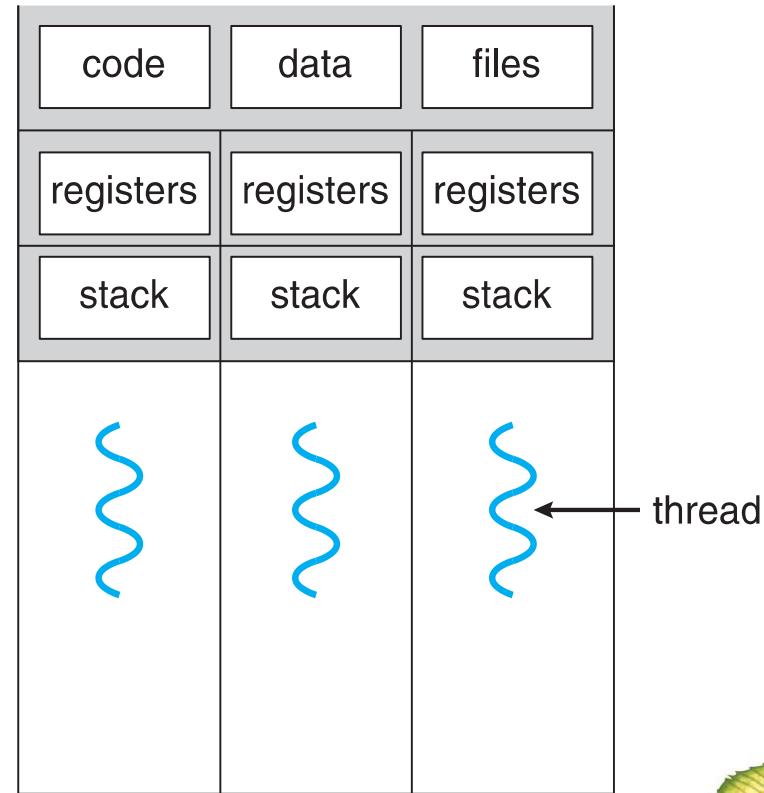


Single and Multithreaded Processes

- ✓ A traditional (or **heavy weight**) process has a single thread of control.
- ✓ If a process has multiple threads of control, it can perform more than one task at a time.



single-threaded process



multithreaded process

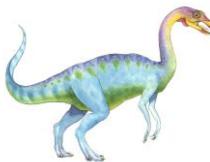




Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display-a thread for displaying graphics
 - Fetch data-another thread for responding to keystrokes from the user
 - Spell checking-a third thread for performing spelling and grammar checking in the background
- Applications can also be designed to leverage processing capabilities on multicore systems.
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded





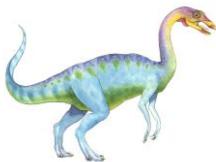
Simple Thread Program

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

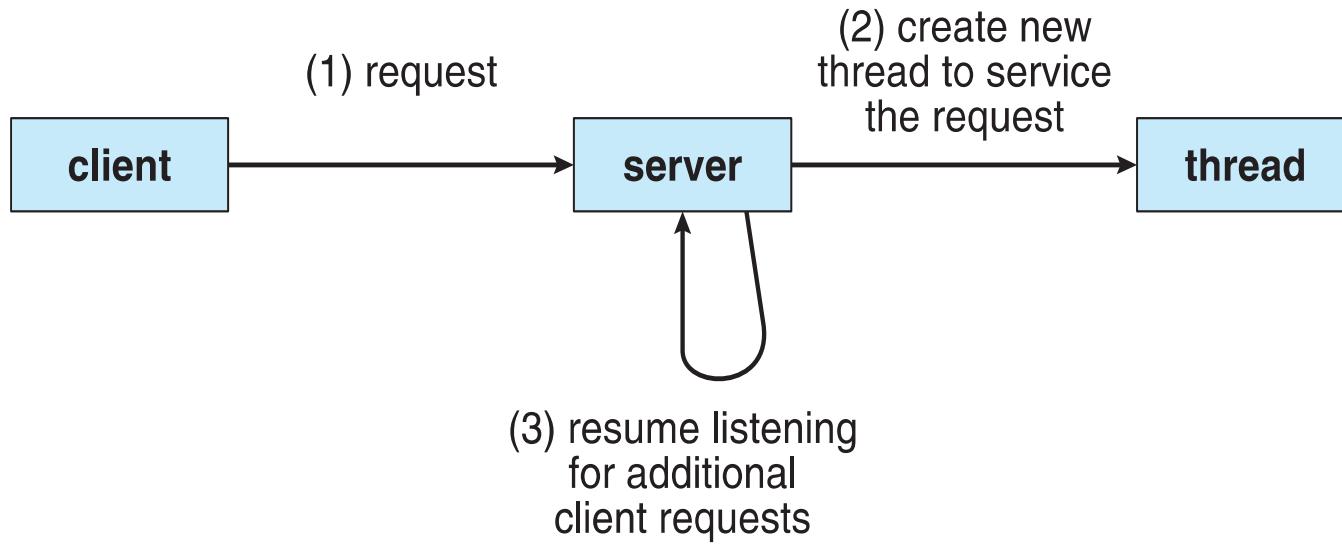
void* thread_code( void * param )
{
    printf( "In thread code\n" );
}

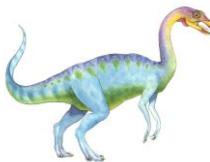
int main()
{
    pthread_t thread;
    pthread_create(&thread, 0, &thread_code, 0 );
    printf("In main thread\n");
}
```





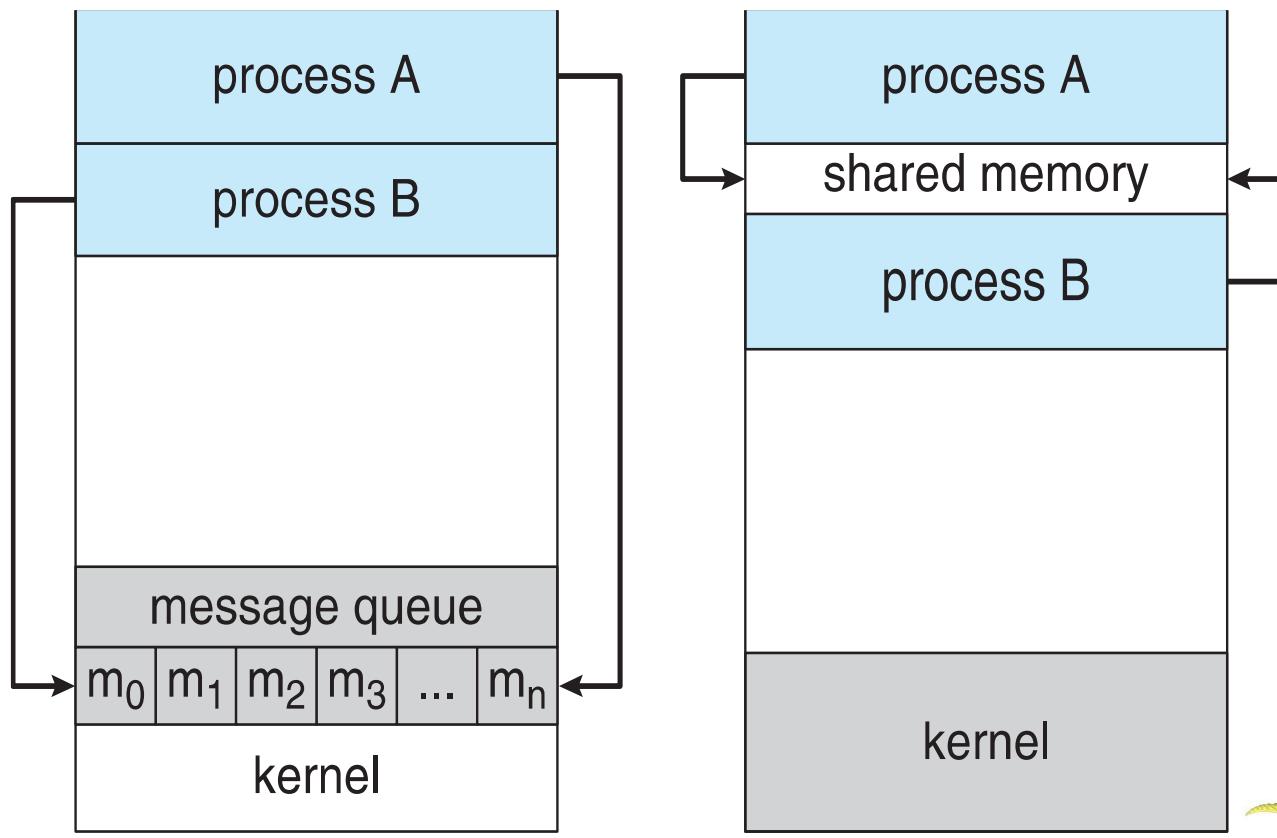
Multithreaded Server Architecture





Benefits

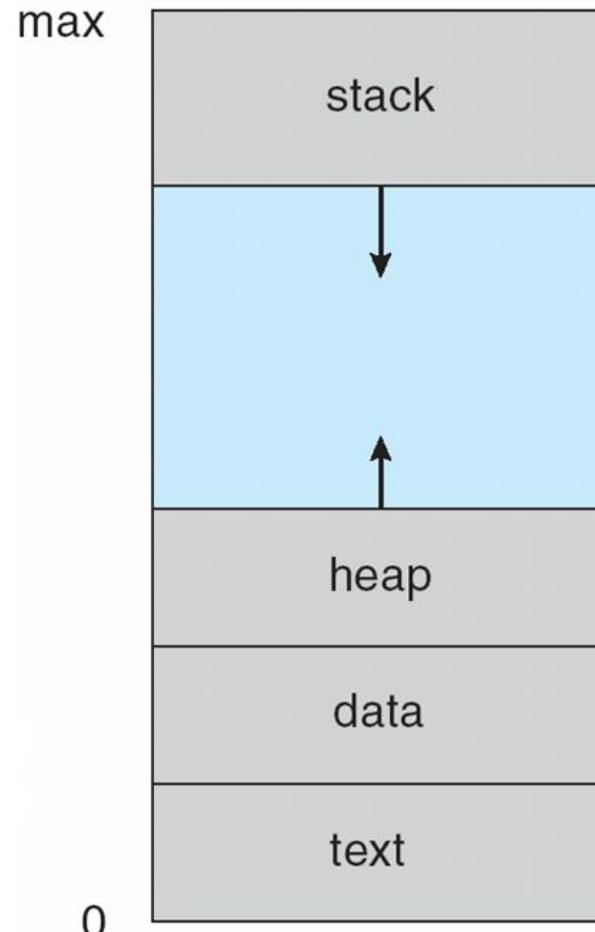
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing





Benefits

- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures





User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
 - managed without kernel support
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
 - managed directly by the operating system
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X
- **A relationship must exist between user threads and kernel threads.**





Multithreading Models

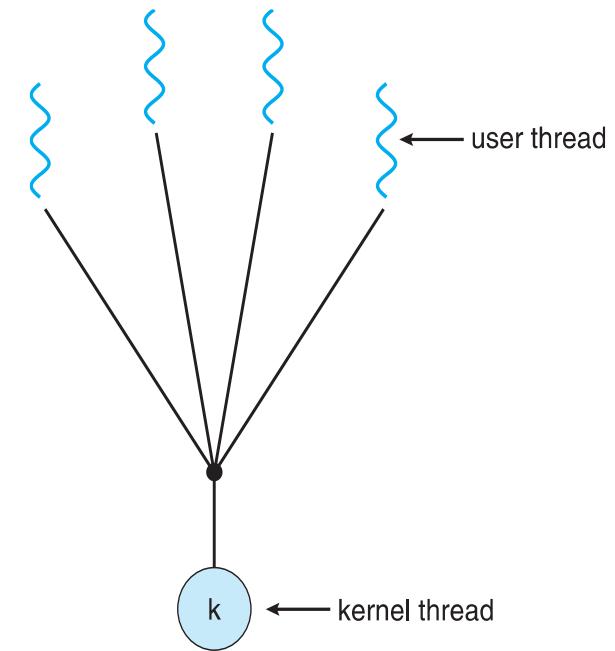
- Many-to-One
- One-to-One
- Many-to-Many

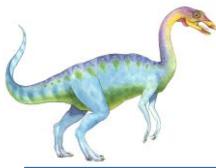




Many-to-One

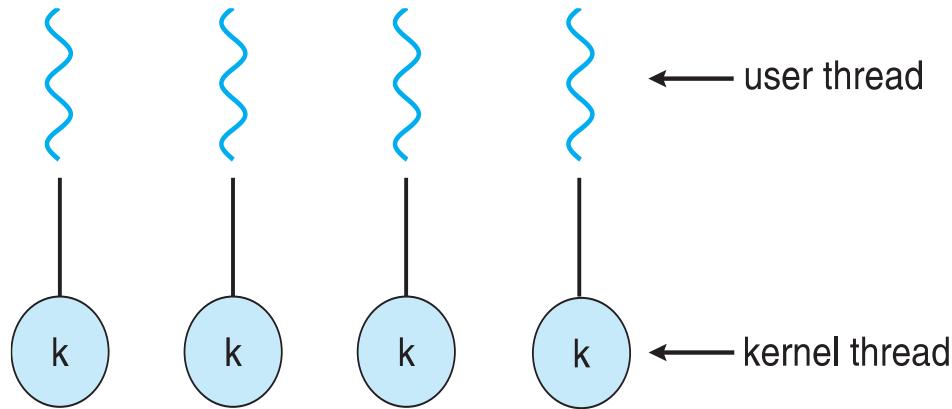
- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space, **so it is efficient**
- **One thread blocking causes all to block**
- **Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time**
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
- very few systems continue to use the model because of its **inability to take advantage of multiple processing cores**





One-to-One

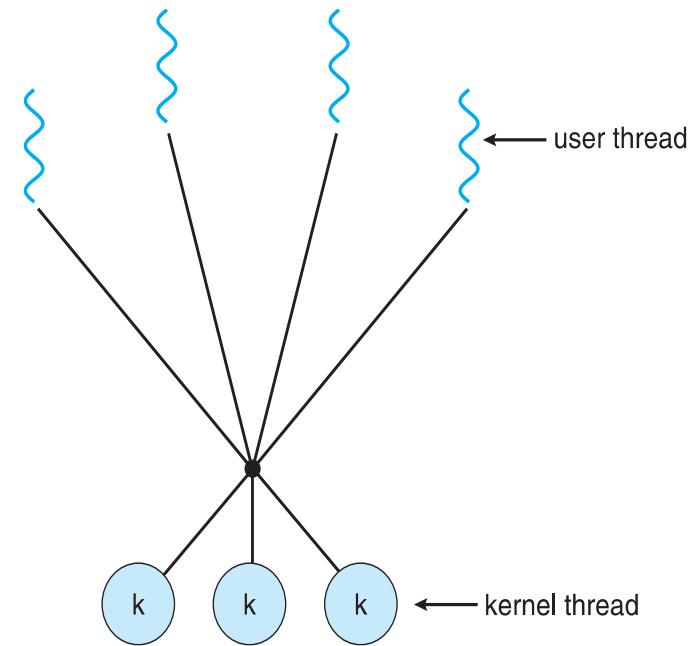
- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- **More concurrency than many-to-one:** by allowing another thread to run when a thread makes a blocking system call
- It also allows multiple threads to run in parallel on multiprocessors
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later

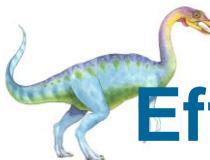




Many-to-Many Model

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





Effect of Multithreaded Models on Concurrency

- The many-to-one model allows the developer to create as many user threads as he/she wishes, it does not result in true concurrency, because **the kernel can schedule only one thread at a time.**
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
- The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.





Microsoft Windows [Version 10.0.19044.1889]
(c) Microsoft Corporation. All rights reserved.

```
C:\Users\MAHE>wmic  
wmic:root\cli>CPU Get NumberOfCores  
NumberOfCores  
4  
  
wmic:root\cli>CPU Get NumberOfCores, NumberOfLogicalProcessors  
NumberOfCores  NumberOfLogicalProcessors  
4                8  
  
wmic:root\cli>
```





Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing a thread libraries
 - Library entirely in user space with no kernel support
 - ▶ All code and data structures for the library exist in user space.
 - ▶ This means that invoking a function in the library results in a local function call in user space and not a system call
 - Kernel-level library supported by the OS
 - ▶ Code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel
- Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

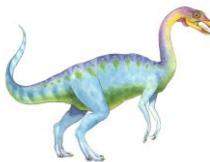




PThreads

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





Windows, and Java Thread

- 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.
- However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
- This means that on Windows systems, Java threads are typically implemented using the Windows API;
- UNIX and Linux systems often use Pthreads.





Threading Issues

- Some of the issues to consider in designing multithreaded programs
- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





Semantics of fork() and exec()

- The semantics of the fork() and exec() system calls change in a multithreaded program
- Issue
 - If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
 - ▶ Does `fork()` duplicate only the calling thread or all threads?
- Solution
 - Some UNIX systems have chosen to have two versions of `fork()`,
 - ▶ one that duplicates all threads and
 - ▶ another that duplicates only the thread that invoked the `fork()` system call.
- **But which version of `fork()` to use and when?**





Semantics of fork() and exec()

- If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.
- Depends on the application
 - If **exec()** is called immediately after forking,
 - ▶ Then duplicating all threads is unnecessary, as the program specified in the parameters to **exec()** will replace the process. In this instance, duplicating only the calling thread is appropriate.
 - If the separate process does not call **exec()** after forking,
 - ▶ Then separate process should duplicate all threads.





Signal Handling

Signals are used in UNIX systems **to notify a process that a particular event has occurred.**

- ❑ A signal may be received either **synchronously or asynchronously** depending on the **source** of and the **reason for the event** being signaled.
- ❑ All signals, whether synchronous or asynchronous, follow the same pattern:
- ❑ A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Once delivered, the signal must be handled
 4. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined





Signal Handling (Cont.)

Every signal has **default handler** that kernel runs when handling that signal

- ❑ **User-defined signal handler** can override default action
- ❑ Signals are handled in different ways.
 - ❑ Some signals (such as changing the size of a window) are simply **ignored**; others (such as an illegal memory access) are handled by **terminating the program**.
 - ❑ Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. **Where, then, should a signal be delivered?**
 - ❑ Deliver the signal to the thread to which the signal applies
 - ❑ Deliver the signal to every thread in the process
 - ❑ Deliver the signal to certain threads in the process
 - ❑ Assign a specific thread to receive all signals for the process





Signal Handling (Cont.)

- The **method for delivering** a signal depends on the **type of signal generated**.
 - Synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
 - Asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.
- The standard UNIX function for delivering a signal is
 - `kill(pid_t pid, int signal)`
 - `Pthread_kill(pthread_t tid, int signal)`

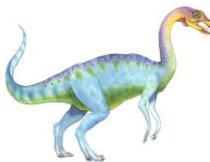




Thread Cancellation

- Terminating a thread before it has finished
 - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
 - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation:** One thread terminates the target thread immediately
 - **Deferred cancellation:** allows the target thread to periodically check if it should be cancelled



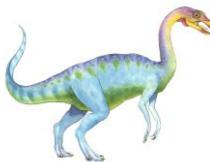


Thread Cancellation

- The difficulty with cancellation occurs in situations where **resources have been allocated to a canceled thread** or where a thread is canceled while **in the midst of updating data it is sharing with other threads**. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously **may not free a necessary system-wide resource**.
- With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
.  
.  
  
/* cancel the thread */  
pthread_cancel(tid);
```





Thread Cancellation (Cont.)

- Invoking thread cancellation requests, cancellation, but actual cancellation depends on thread state
- Pthreads supports three cancellation modes. Each mode is defined as a state and a type

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**
 - ▶ i.e. `pthread_testcancel()`
 - ▶ Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals





Thread-Local Storage

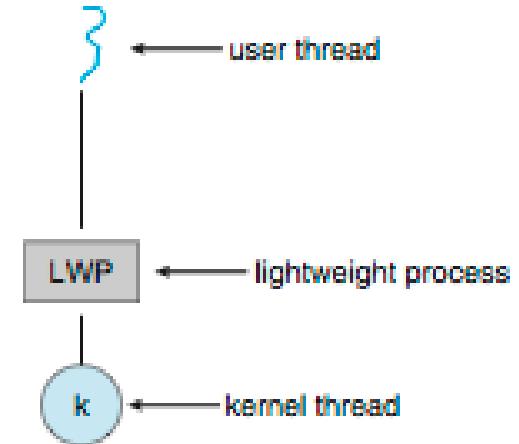
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to **static** data
 - TLS is unique to each thread

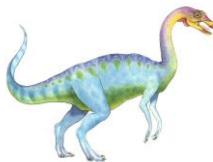




Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to kernel thread
 - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Operating System Examples

- Windows Threads
- Linux Threads





Linux Threads

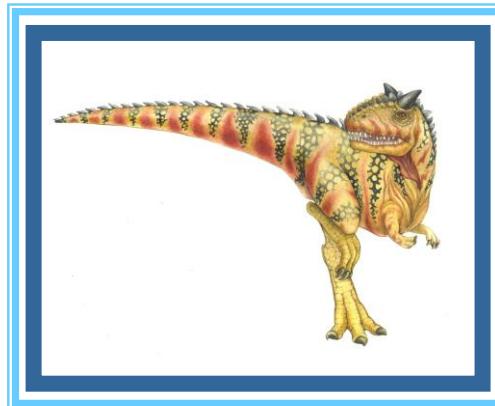
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
 - Flags control behavior

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

- `struct task_struct` points to process data structures (shared or unique)



Process Synchronization





Background

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





Producer

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





Consumer

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





Race Condition

- A situation where several processes accesses 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 **race condition**.
- This condition can be avoided using the technique **process synchronization**
- Here, **ensure only one process manipulate shared data at a time.**





Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

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

S0: producer execute <code>register1 = counter</code>	{register1 = 5}
S1: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute <code>register2 = counter</code>	{register2 = 5}
S3: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute <code>counter = register1</code>	{counter = 6 }
S5: consumer execute <code>counter = register2</code>	{counter = 4}

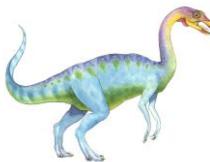




Critical Section Problem

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





Critical Section

- General structure of process P_i

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





Solution to Critical-Section Problem

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

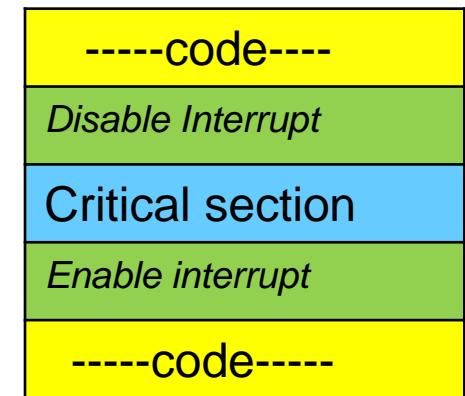




Hardware Solution

- **Entry section – first action is “disable interrupts”**
- **Exit section – last action is “enable interrupts”**
- Must be done by OS. Why?
- Implementation issues:
 - **Uniprocessor systems**
 - Currently running code would execute without preemption
 - **Multiprocessor systems**
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- **Is this an acceptable solution?**

This is impractical if the critical section code is taking long time to execute





Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ▶ Essentially free of race conditions in kernel mode





Algorithm 1: Software Solution for Process Pi

Keep a variable “turn” to indicate which process next

```
do {  
    while (turn != i);  
        critical section  
    turn = j;  
        remainder section  
    } while (true);
```

- Algorithm is correct. Only one process at a time in the critical section.
- What if turn = j, Pi wants to enter in critical section and Pj does not want to enter in critical section?





Algorithm 1: Software Solution for Process Pi

```
do {  
    while (turn != i);  
        critical section  
    turn = j;  
        remainder section  
    } while (true);
```

- What if turn = j, Pi wants to enter in critical section and Pj does not want to enter in critical section?





Algorithm 2 for Process P_i

- 2 Process solution using **Flag variable**
- Flag is Boolean variable with value T & F

```
While (1)
```

```
{
```

```
    Flag[i]=T //  $P_i$  wish to go in CS
```

```
    while(Flag[j]); Check weather  $P_j$  wants to go to CS
```

```
        CS
```

```
        Flag[i]=F
```

```
}
```

CS	P0	P1
	F	F
P0	T	F
P1	F	T
	F	F
P0	T	F
	F	F
P0	T	F
	F	F
	T	T





Algorithm 3: Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- 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 P_i is ready!





Algorithm 3: Algorithm for Process P_i

```
do {  
    flag[i] = true;  
  
    turn = j;  
  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```

FLAG





Algorithm 3: Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

```
while (flag[j] && turn == j);
```

FLAG

P0	P1
T	F
T	T
F	T

- But results in **busy waiting**.

TURN:1





2 Process solution using Turn variable

Algorithm for Process P_i:

```
do {  
  
    while (turn != i);  
  
        critical section  
    turn = j;  
  
        remainder section  
} while (true);
```





2 Process solution using Flag variable

- Flag is Boolean variable with value T & F
- Algorithm for Process P_i :

```
While (1)
{
    Flag[i]=T //  $P_i$  wish to go in CS
    while(Flag[j]); // Check whether  $P_j$  wants to go to CS
    CS
    Flag[i]=F
}
```





Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- 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 P_i is ready!





Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```





P₀

```
do {  
    flag[0] = true;  
    turn = 1;  
    while (flag[1] && turn == 1);  
        critical section  
    flag[0] = false;  
        remainder section  
} while (true);
```

P₁

```
do {  
    flag[1] = true;  
    turn = 0;  
    while (flag[0] && turn == 0);  
        critical section  
    flag[1] = false;  
        remainder section  
} while (true);
```

0	1
T	T

0





Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met





Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too 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);
```

- Lock is provided by the hardware (Hardware designers must provide such a functionality).
- Two processes cannot have the lock simultaneously.





test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.





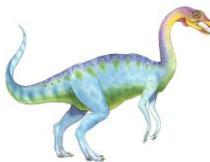
Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock)); /* do nothing */  
          /* critical section */  
  
    lock = false;  
          /* remainder section */  
  
} while (true);
```

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





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;  
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” **the value of the passed parameter “new_value” only if “value” ==“expected”**. That is, the swap takes place only under this condition.





Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

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

    lock = 0;
    /* remainder section */

} while (true);
```

If previous value of lock was 0 then new value
is set to 1



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

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

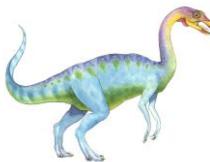




Mutex Locks

- ❑ Previous solutions are complicated (are m/c level instructions) 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
 - ❑ Boolean variable indicating if lock is available or not





acquire() and release()

```
② acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false; ;  
}  
  
③ release() {  
    available = true;  
}  
  
④ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

- 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**





Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two **indivisible (atomic) operations**
 - **wait()** and **signal()**
 - ▶ Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)          //testing of integer value S  
        ; // busy wait  
    S--;  
}
```

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

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





Semaphore Usage

- Can solve various synchronization problems
- A solution to the CS problem:- Create a semaphore “**sync** initialized to 1”

wait (sync);

CS

signal(sync);

- Consider **P₁** and **P₂** that require **S₁** to happen before **S₂**
 - » Create a semaphore “**synch**” initialized to 0

P1:

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

P2:

```
wait(synch);  
S2;
```





Types of semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can implement a counting semaphore **S** as a binary semaphore





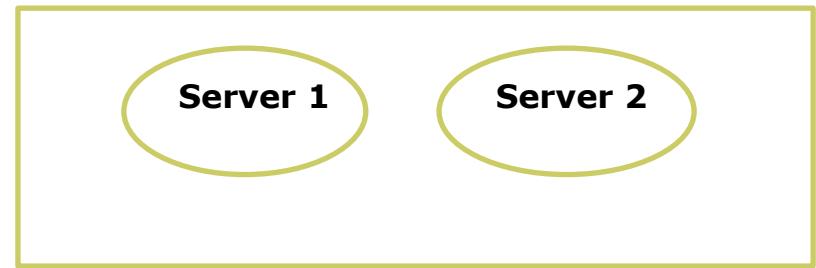
Counting semaphore example

- Allows at most two processes to execute in critical section
- Create a semaphore “sync initialized to 2 ” (assume two servers)
ie: 2 processes can execute simultaneously

wait (sync);

CS

signal(sync);





Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
 - This implementation is based on **busy waiting** in critical section implementation(ie; code for wait() and signal())
 - ▶ But implementation code is short
 - ▶ Little busy waiting if critical section rarely occupied
- Can we implement semaphore with no busy waiting ?

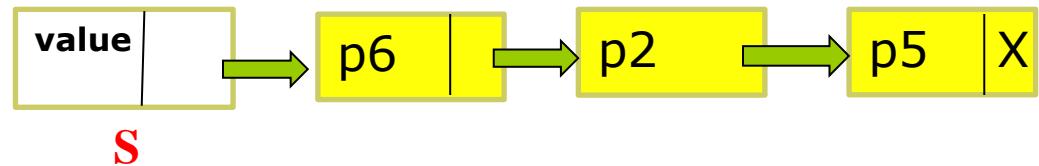




Semaphore Implementation with no Busy waiting

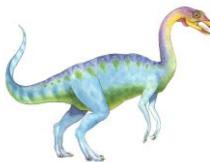
- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list

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



- 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





Implementation with no Busy waiting (Cont.)

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





Implementation with no Busy waiting (Cont.)

- Does this implementation ensure the progress requirement?
- Does this implementation ensure the Bounded waiting requirement?





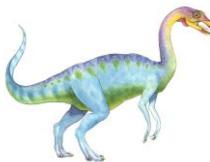
Deadlock and Starvation

- **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	P_1
<code>wait(S) ;</code>	<code>wait(Q) ;</code>
<code>wait(Q) ;</code>	<code>wait(S) ;</code>
...	...
<code>signal(S) ;</code>	<code>signal(Q) ;</code>
<code>signal(Q) ;</code>	<code>signal(S) ;</code>

- **Starvation – indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**





Problems with semaphore

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

- Deadlock and starvation are possible.
- Solution → create high level programming language constructs.

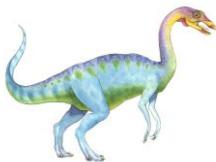




Classical Problems of Synchronization

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

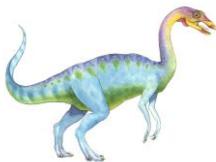




Bounded-Buffer Problem

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





Bounded Buffer Problem (Cont.)

□ The structure of the producer process

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





Bounded Buffer Problem (Cont.)

□ The structure of the consumer process

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





Underflow/ overflow condition

- **To check underflow:- Consumer is trying to read from an empty buffer**

n=5,

mutex=1,

full=0,

empty=5

- **To check overflow:- Producer is trying to produce an item when buffer if full.**

n=5,

mutex=1,

full=5,

empty=0





Readers-Writers Problem

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





Readers-Writers Problem (Cont.)

□ The structure of a writer process

```
do  {
    wait(rw_mutex);

    ...
    /* writing is performed */

    ...

    signal(rw_mutex);
} while (true);
```





Readers-Writers Problem (Cont.)

□ The structure of a reader process

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





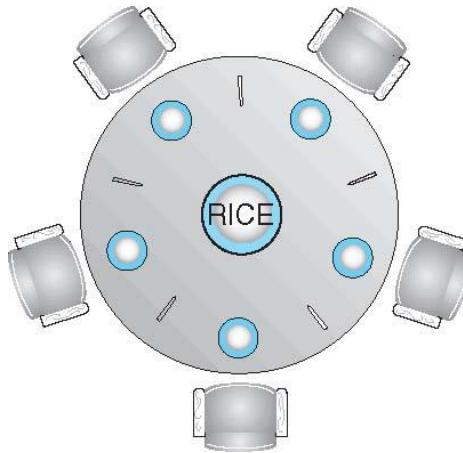
Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks





Dining-Philosophers Problem



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





Dining-Philosophers Problem Algorithm

- The structure of Philosopher i :

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

- What is the problem with this algorithm?





Dining-Philosophers Problem Algorithm (Cont.)

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





Readers-Writers Problem

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



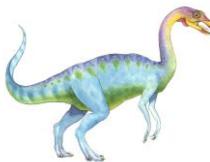


Dining-Philosophers Problem



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





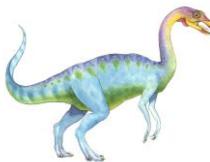
Dining-Philosophers Problem Algorithm

- The structure of Philosopher i :

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

- What is the problem with this algorithm?



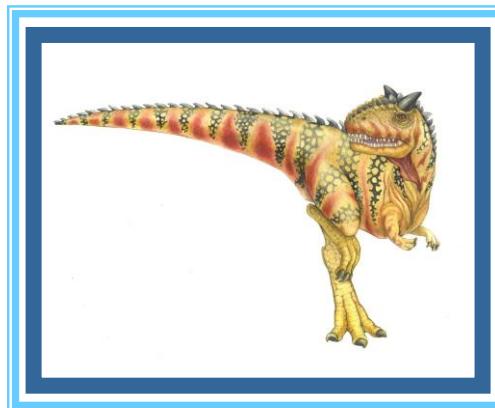


Dining-Philosophers Problem Algorithm (Cont.)

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



Main Memory Management





Background

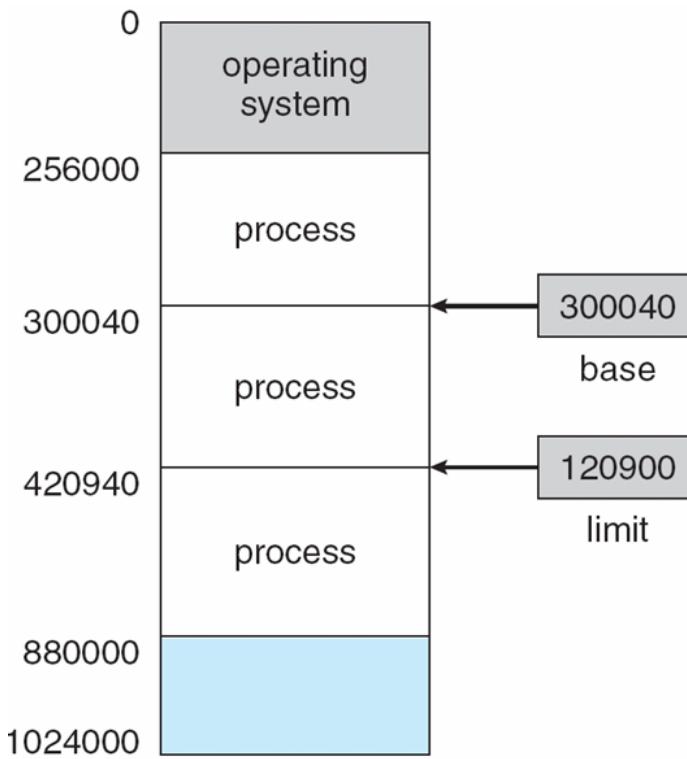
- Program must be brought (from disk) into memory and placed within a process for it to be run
 - *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
 - User programs go through several steps before being executed.
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees
 - a stream of addresses + read requests,
 - or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

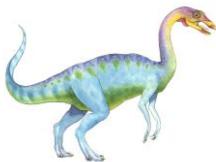




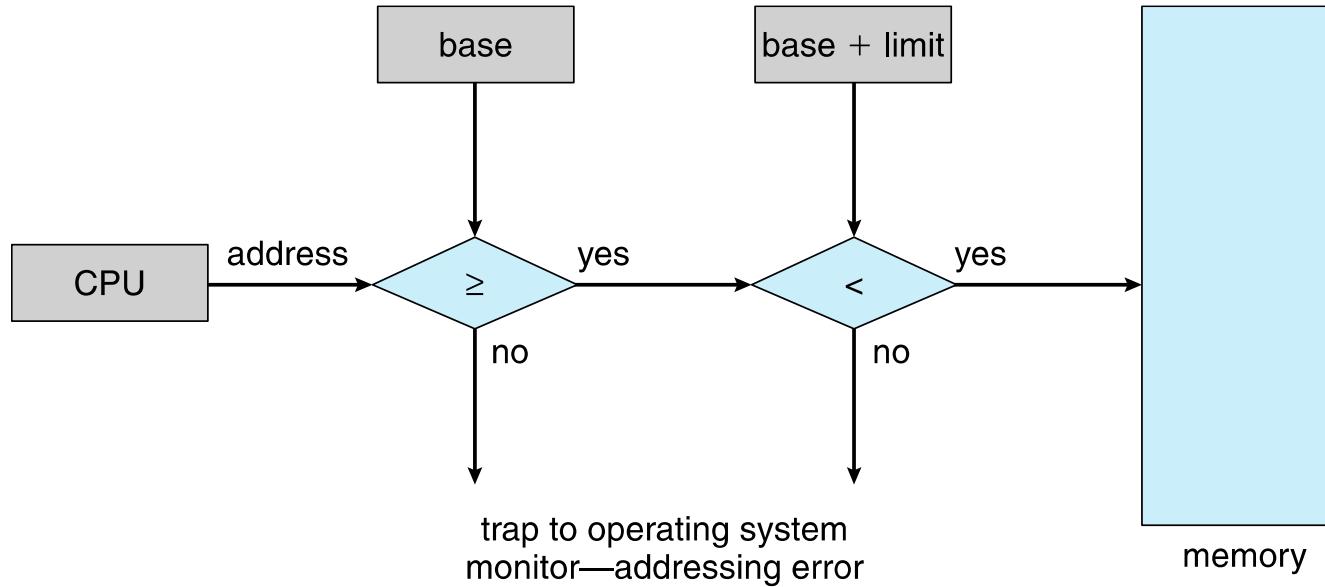
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user





Hardware Address Protection





Address Binding

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





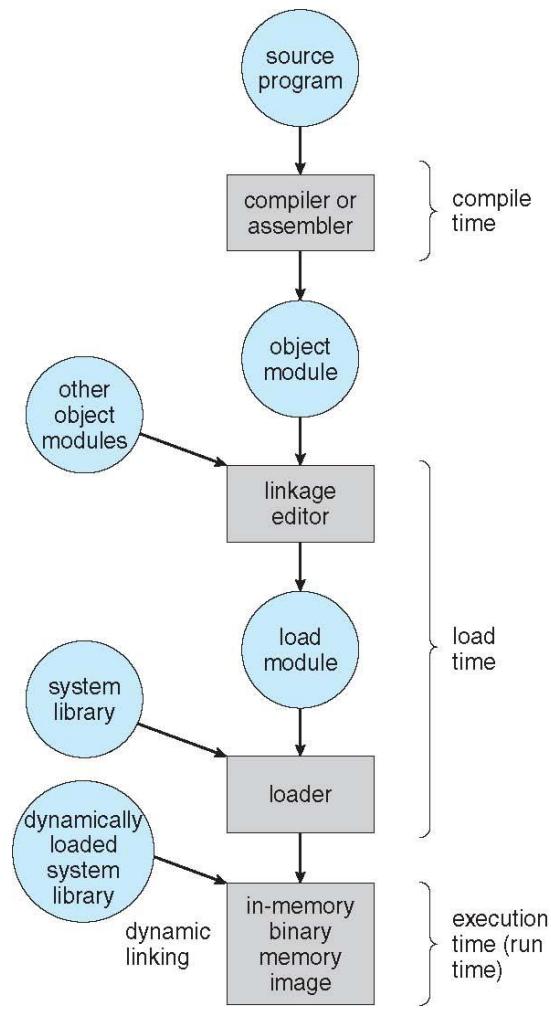
Binding of Instructions and Data to Memory

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





Multistep Processing of a User Program





Logical vs. Physical Address Space

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

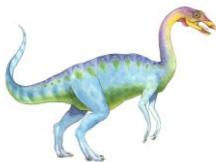




Memory-Management Unit (MMU)

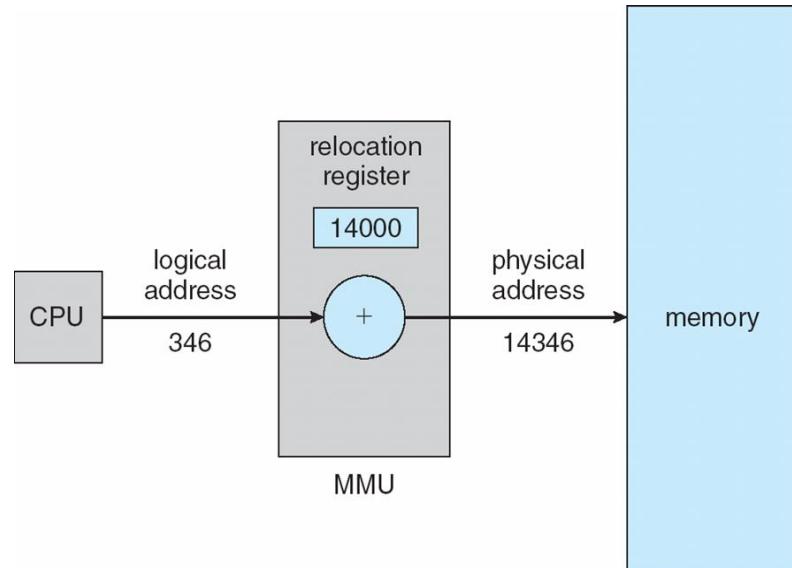
- Hardware device that at run time maps virtual to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base register now called **relocation register**
 - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses





Dynamic relocation using a relocation register

- 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 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

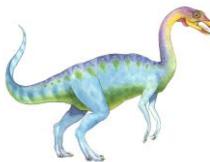




Swapping

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





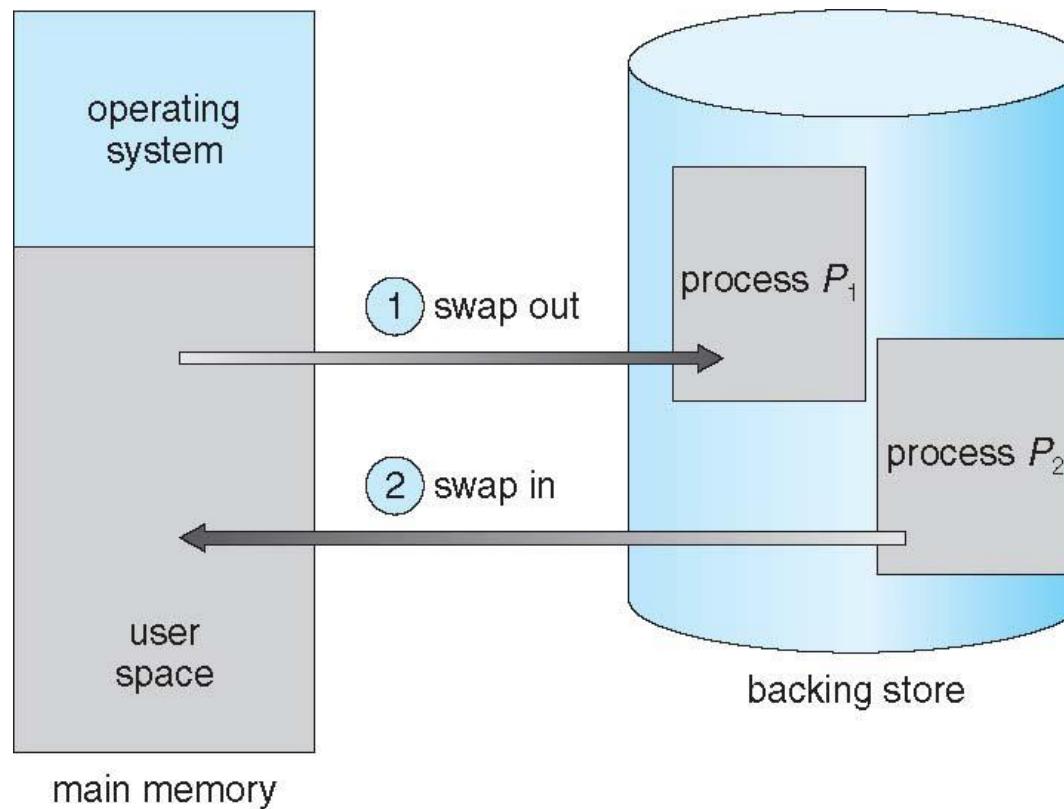
Swapping (Cont.)

- 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
 - Started if more than threshold amount of memory allocated
 - Disabled again once memory demand reduced below threshold





Schematic View of Swapping





Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap out time of 2000 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
 - System calls to inform OS of memory use via `request_memory()` and `release_memory()`





Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
 - Pending I/O – can't swap out as I/O would occur to wrong process
 - Or always transfer I/O to kernel space, then to I/O device
 - ▶ 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

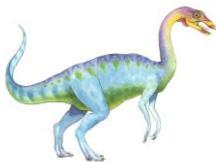




Contiguous Allocation

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





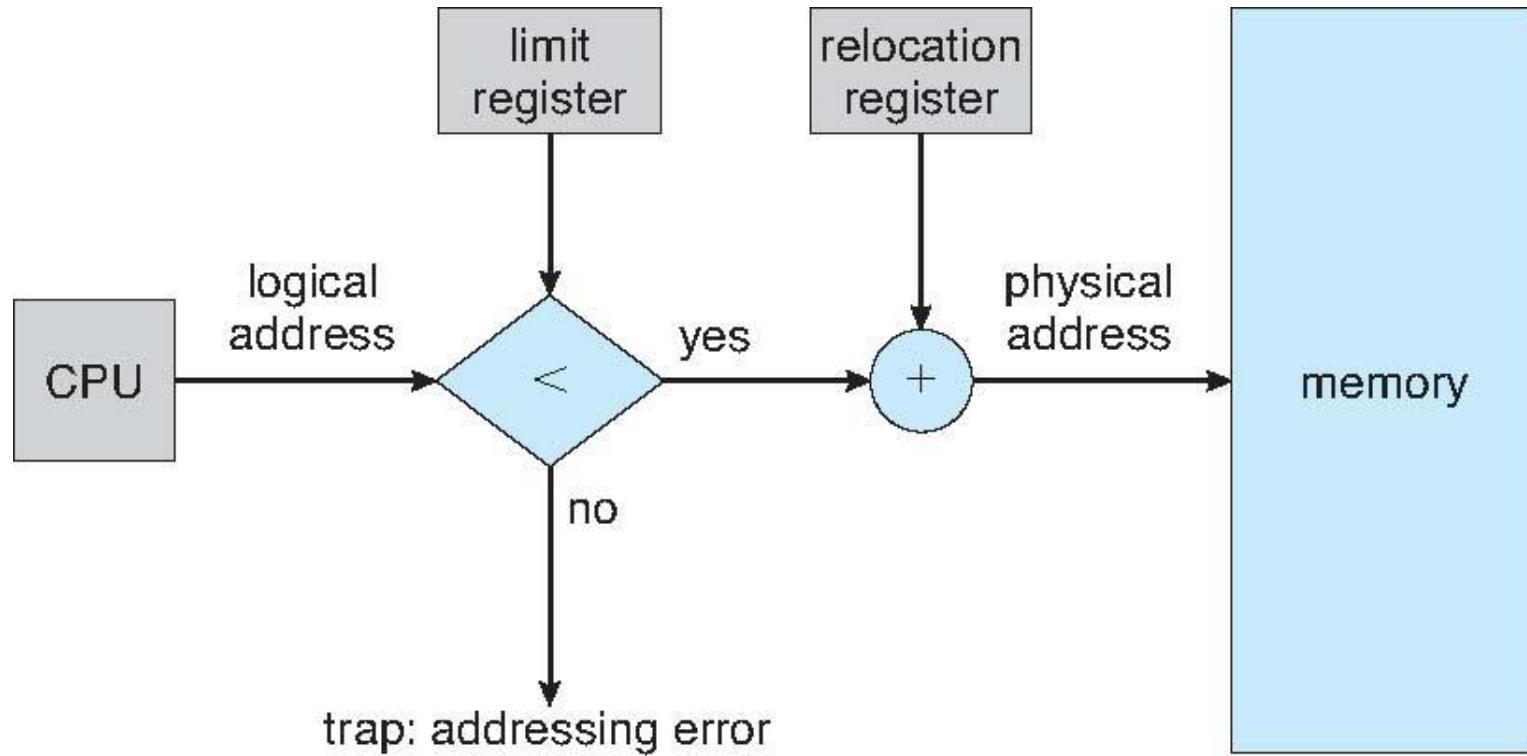
Contiguous Allocation (Cont.)

- 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*



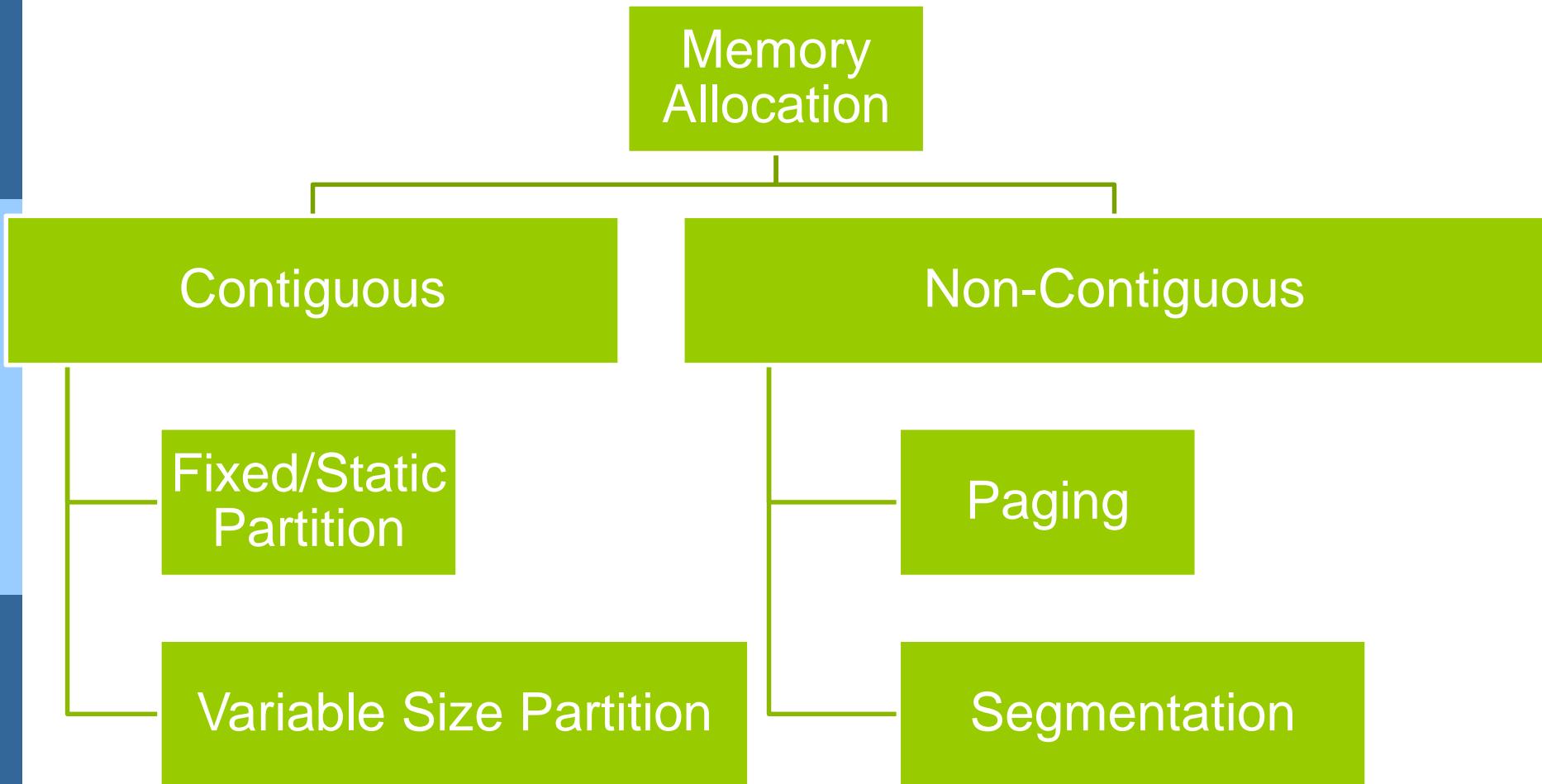


Hardware Support for Relocation and Limit Registers





Memory Mapping Techniques



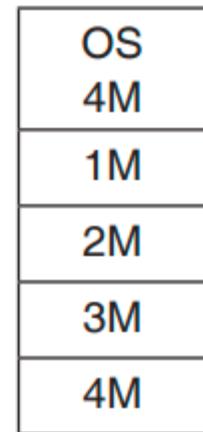


Contiguous Storage Allocation

- A memory resident program occupies a single contiguous block of memory.
- Fixed/Static Partitioning
 - The main memory is divided into a number of static partitions at system generation time.
 - Moreover, a process may be loaded into a partition of equal or greater size.
 - Partition size: Two alternatives of fixed partition are as follows:
 - ▶ 1. Equal-size partitions
 - ▶ 2. Unequal-size partitions



Equal size



Unequal size



Contiguous Storage Allocation

- Equal-size partitions:
- Any process whose size is less than or equal to the partition size can be loaded into any available partition.
- Two problems with this technique are as follows:
 - 1. A program may be too big to fit into a partition.
 - ▶ Use overlaying to solve this problem.
 - 2. Main memory utilization is extremely inefficient, as there is a possibility of internal fragmentation.
- In internal fragmentation, there is a space wastage internal to a partition due to the fact that the block of data loaded is smaller than the partition.

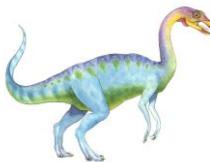




Contiguous Storage Allocation

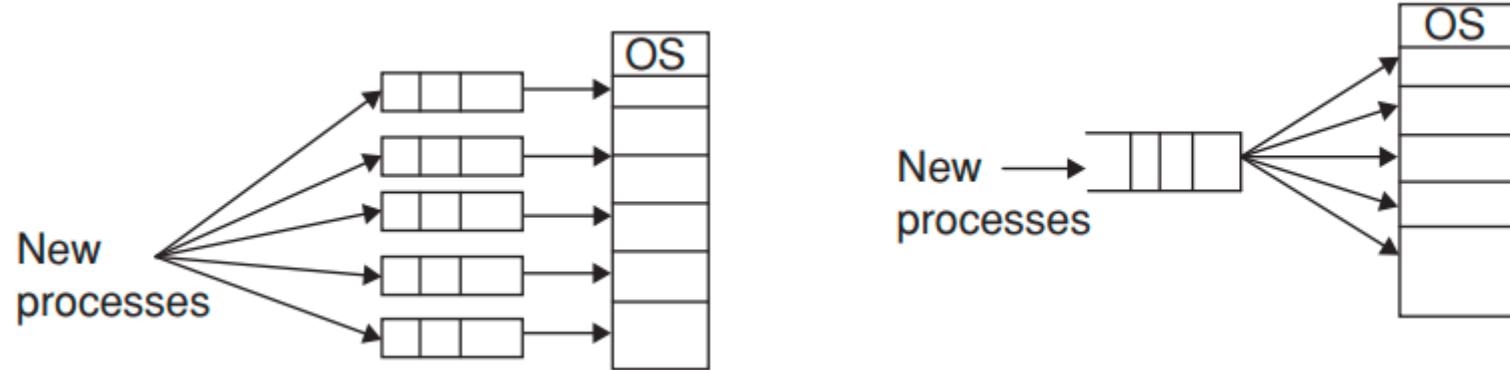
- Unequal-sized partition:
- Both the problems with equal-size partition can be lessened by using unequal-sized partitions.
- Placement algorithm: With equal-size partitions, the placement of processes in memory is trivial.
- As all partitions are of equal size, it doesn't matter which partition is used.
- With Unequal-size partitions, there are two possible ways to assign processes to partitions:
 - 1. Assign each process to the smallest partition within which it will fit.
 - 2. Employ a single queue for all processes





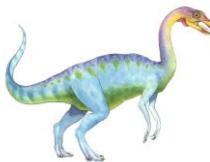
Contiguous Storage Allocation

- 1. Assign each process to the smallest partition within which it will fit.



- Figure shows one process queue for partition.
 - Minimized internal fragmentation.
 - Possibility of unused partitions.
- 2. Employ a single queue for all processes
- When it is time to load a process into main memory, the smallest available partition that will hold the process is selected.





Contiguous Storage Allocation

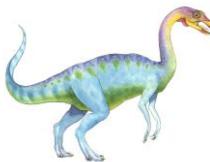
□ Advantages

- 1. Simple to implement.
- 2. Little OS overhead.

□ Disadvantages

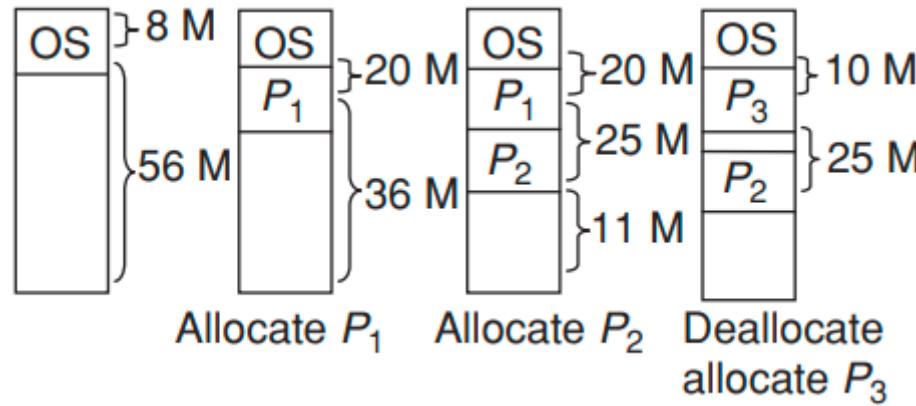
- 1. Inefficient use of memory due to internal fragmentation.
- 2. Maximum number of active processes is fixed.





Contiguous Storage Allocation

- Dynamic Partitioning
- With dynamic partitioning, the partitions are of variable length and number.
- When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

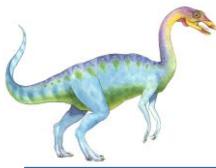




Contiguous Storage Allocation

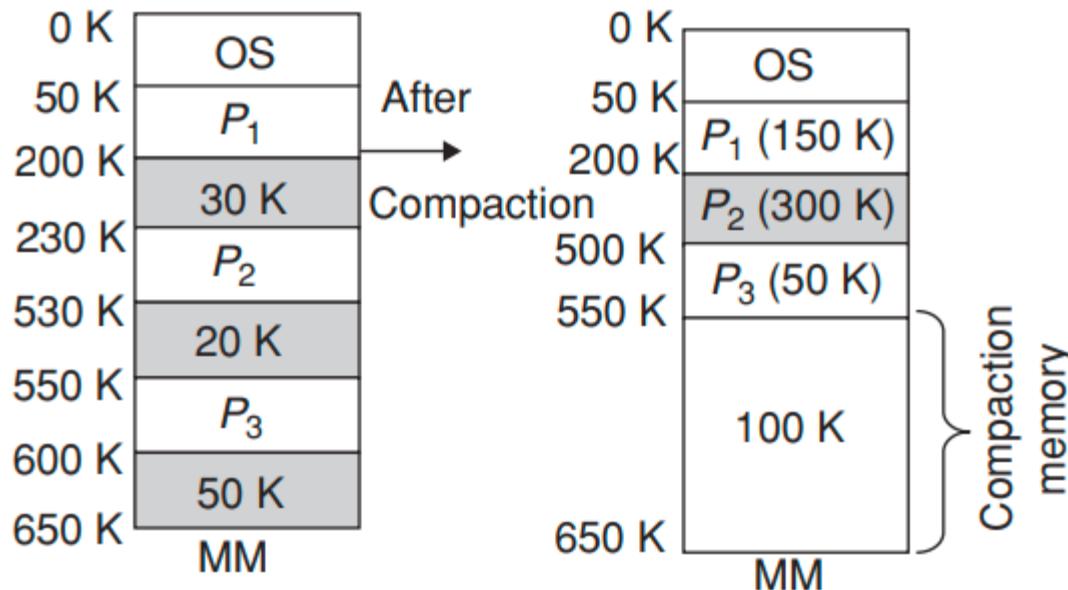
- This method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory.
- As time goes on, memory becomes more and more fragmented and memory utilization declines. This phenomenon is referred to as external fragmentation.
- It indicates the memory that is external to all partitions becomes increasingly fragmented.



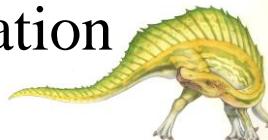


Compaction

- Compaction is a technique by which the resident programs are relocated in such a way that the small chunks of free memory are made contiguous to each other and clubbed together into a single free partition that may be big enough to accommodate more programs.



It should be noted that compaction involves dynamic relocation of a program.





Placement algorithm

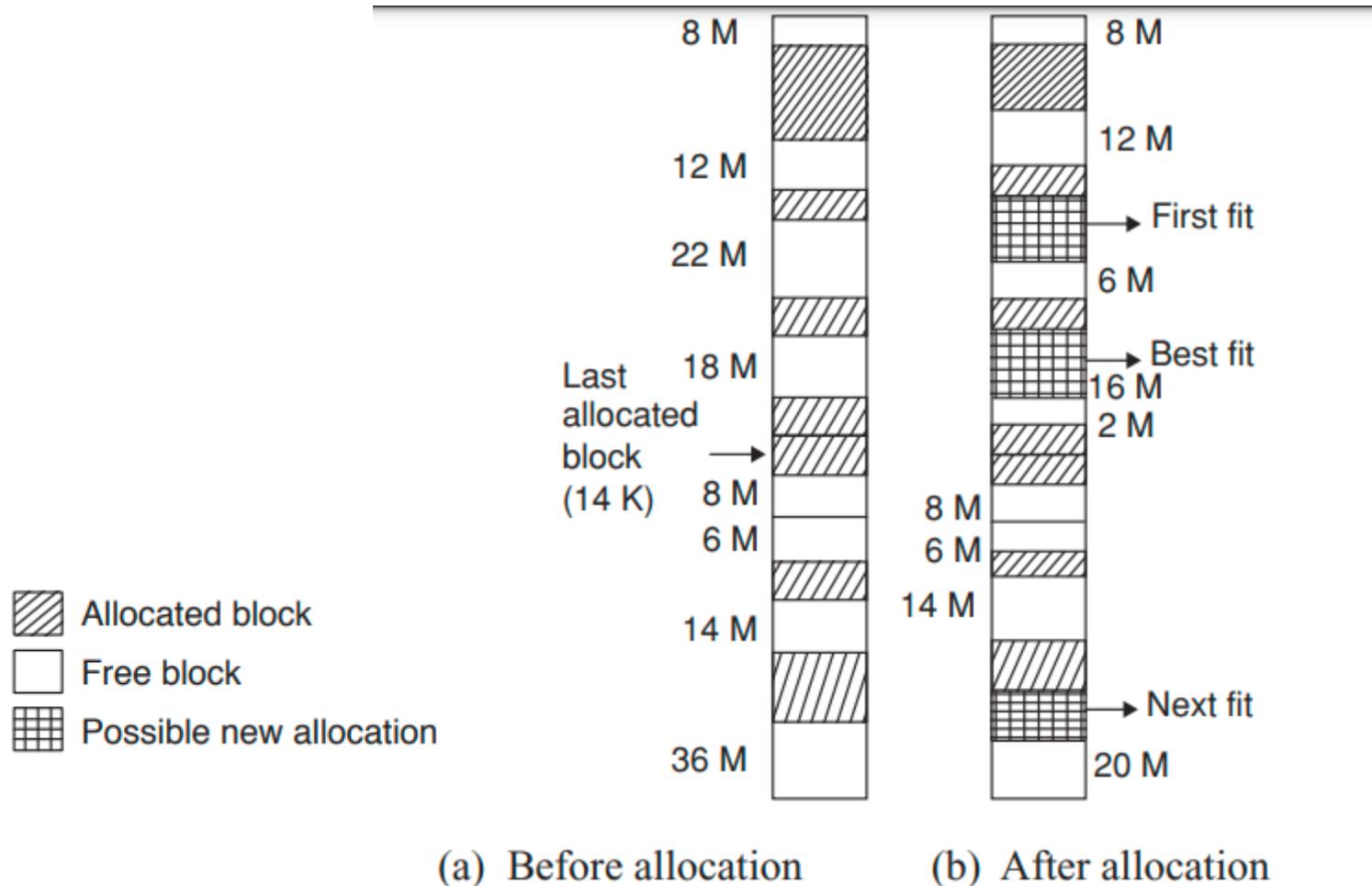
- Memory compaction is a time-consuming process, and hence the OS uses some placement algorithms.
- The three most common strategies to allocate free partitions to the new processes are as follows:
 - 1. First fit: Allocate the first free partition, large enough to accommodate the process. It executes faster.
 - 2. Best fit: Allocate the smallest free partition that meets the requirement of the process. It achieves higher utilization of memory by searching smallest free partition.
 - 3. Worst fit: Allocate the largest available partition to the newly entered process in the system.
 - 4. Next fit: Start from current location in the list.





Placement algorithm

- Example: Consider the following memory configuration after a number of placement and swapping out operations. The last block that was used was a 22 MB block from which a 14 MB partition was created. The figure (b) shows 16 MB allocation request.





Placement algorithm

- Advantages of dynamic partitioning
 - 1. Memory utilization is generally better as partitions are created dynamically.
 - 2. No internal fragmentation as partitions are changed dynamically.
 - 3. The process of merging adjacent holes to form a single larger hole is called coalescing.
- Disadvantages
 - 1. Lots of OS space, time, complex memory management algorithms are required.
 - 2. Compaction time is very high.

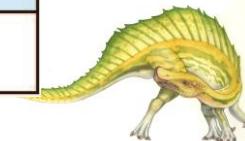
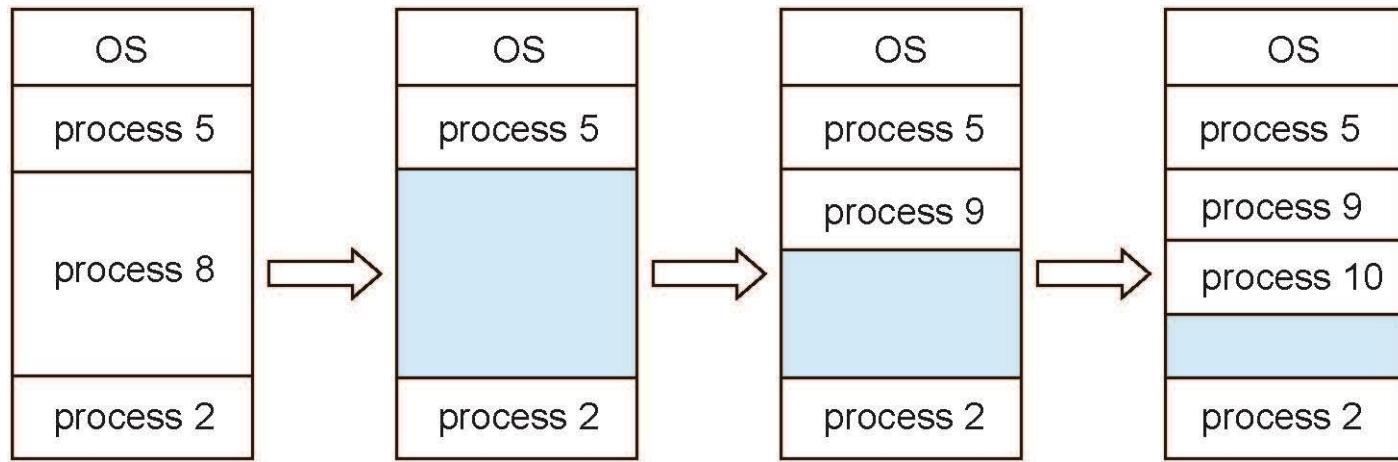




Multiple-partition allocation

□ Multiple-partition allocation

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





Dynamic Storage-Allocation Problem

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

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

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





Fragmentation

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





Fragmentation (Cont.)

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





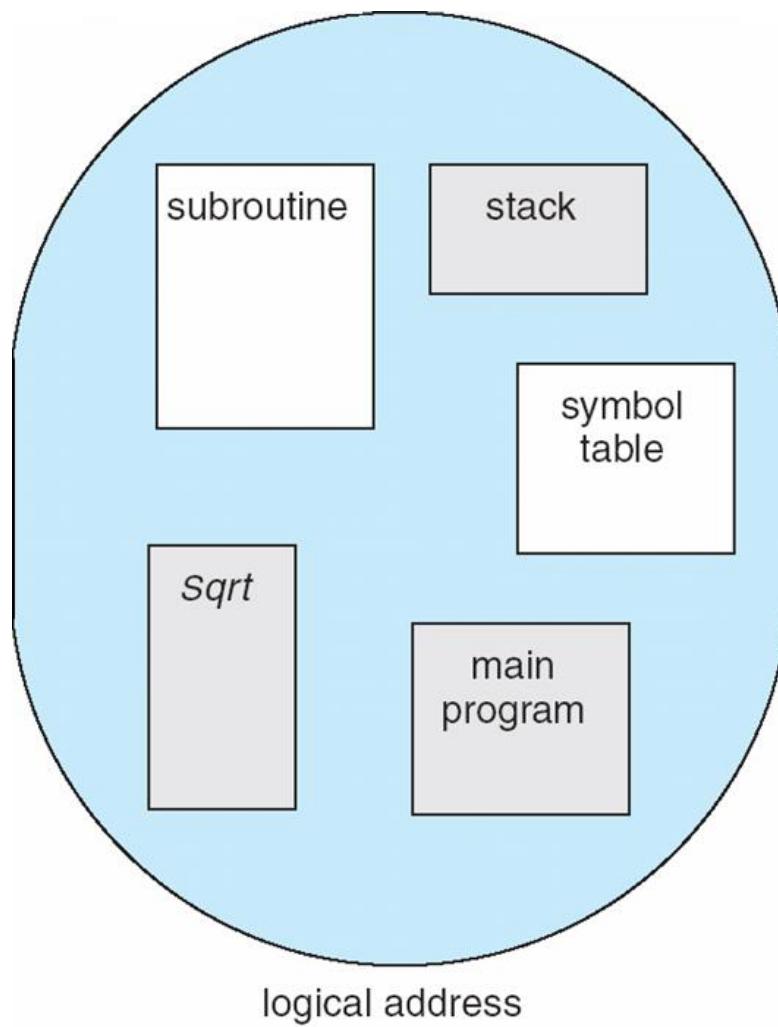
Segmentation

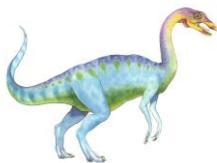
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays



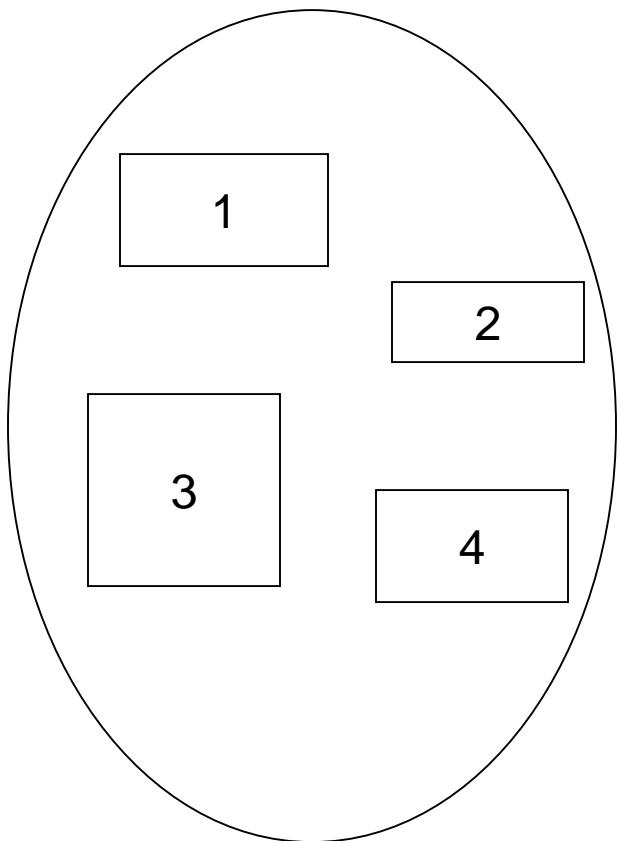


User's View of a Program

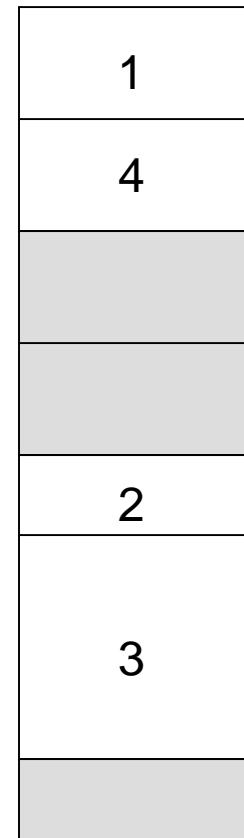




Logical View of Segmentation



user space



physical memory space





Segmentation

- Each process is divided into a number of unequal-size segments.
- A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.
- The logical address using segmentation consists of two parts:
 - **segment number** and an **offset**.
- It makes use of a segment table for each process and a list of free blocks of main memory.
 - Each segment table entry would have to give the starting address in main memory of the corresponding segment.
 - It also contains the length of the segment.

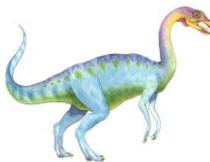




Segmentation

- Steps for address translation:
 - Extract the segment number from the logical address.
 - Use the segment number as an index into the process segment table to find the starting physical address of the segment.
 - Compare the offset to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
 - The desired physical address is the sum of the starting physical address of the segment plus the offset.





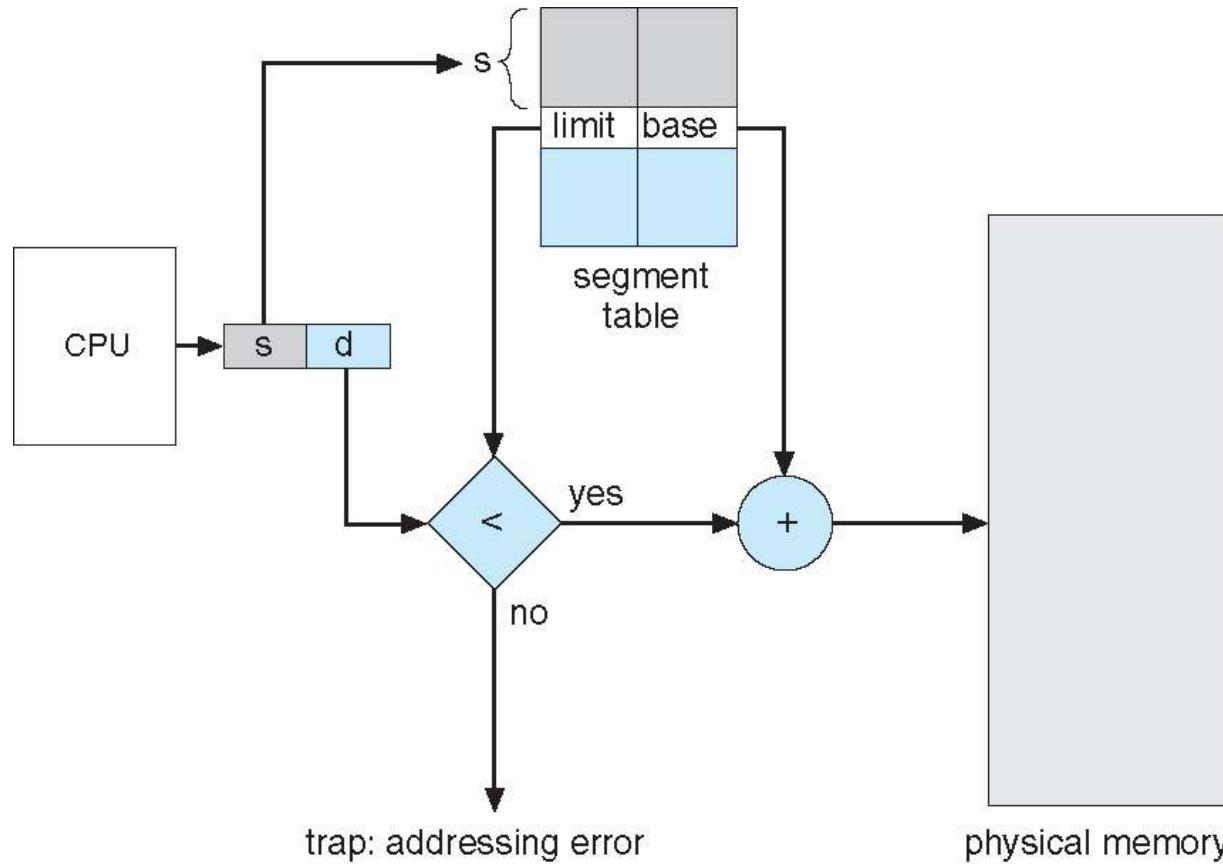
Segmentation Architecture

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





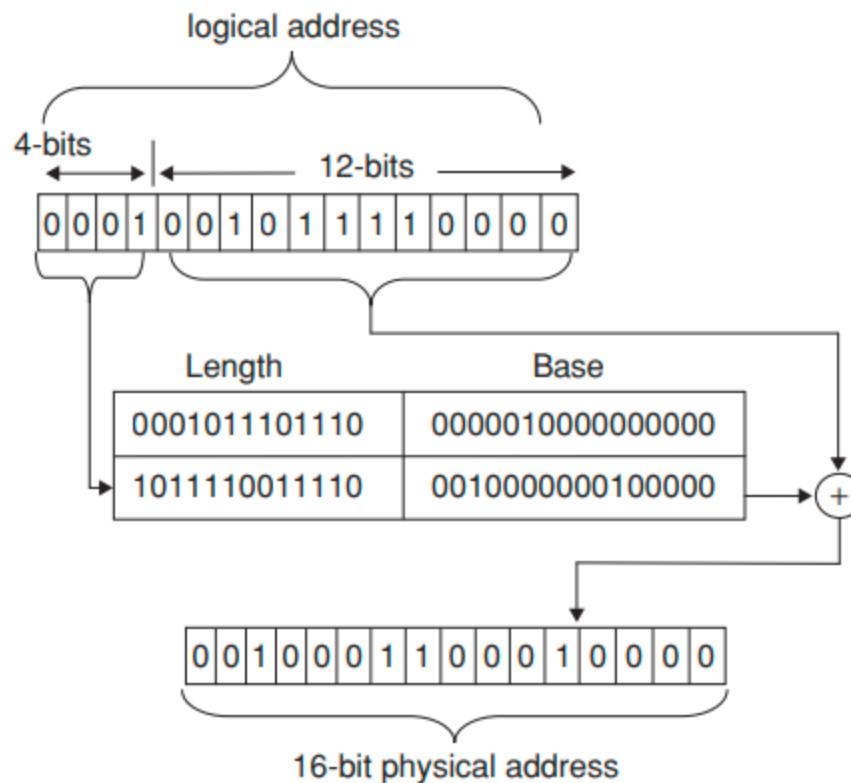
Segmentation Hardware





Segmentation

- Example: Consider the logical address 0001001011110000. Let the segment number consists of 4-bits. Then segment number = 0001 = 1
Offset = 001011110000 = 752.





Segmentation

- Advantages
 - No internal fragmentation
 - Improved memory utilization.
- Disadvantage
 - External fragmentation.

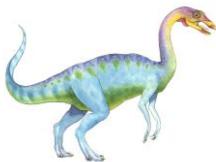




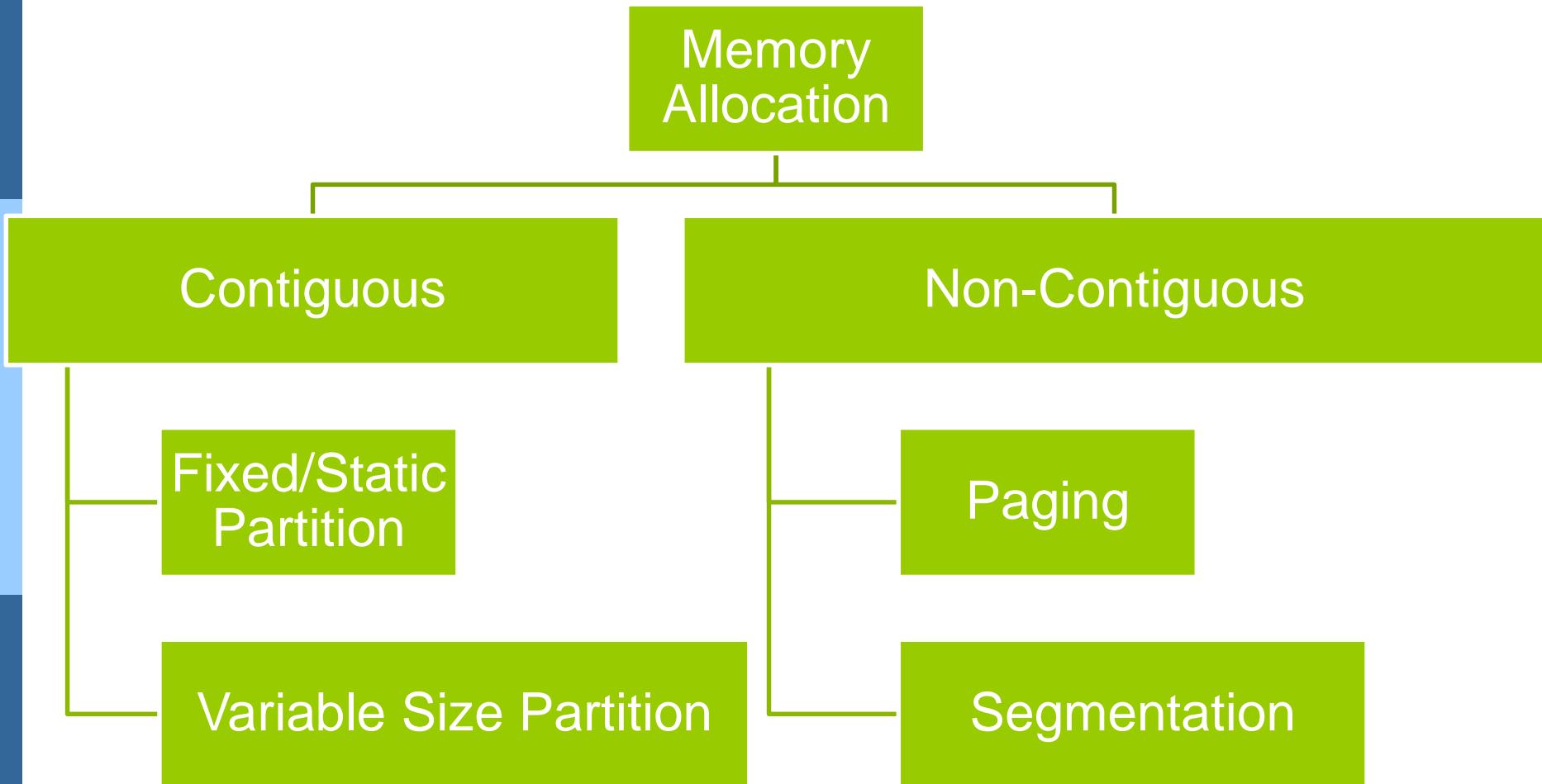
Segmentation Architecture (Cont.)

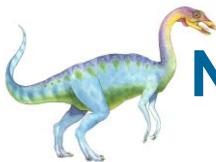
- Protection
 - With each entry in segment table associate:
 - ▶ validation bit = 0 \Rightarrow illegal segment
 - ▶ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem





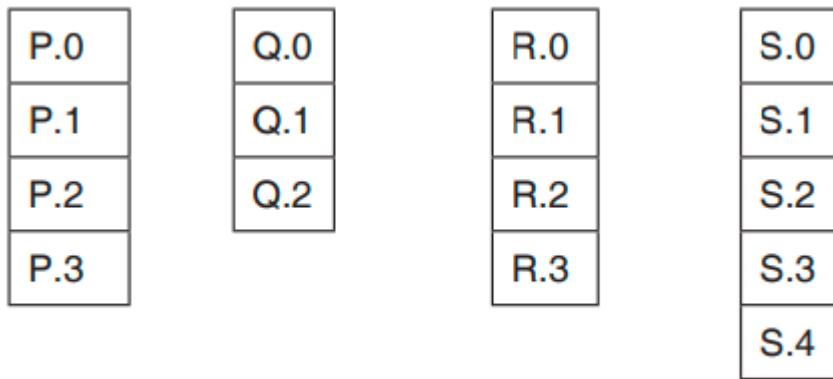
Memory Mapping Techniques





Non-contiguous Storage Allocation Methods-Paging

- Permits the physical address space of a process to be noncontiguous
- The main memory is divided into a number of equal-size frames.
- Each process is divided into a number of equal-size frames.
- The chunks of processes are referred as pages.
- A process is loaded by loading all of its pages into available, not necessarily contiguous frames.
- Example: At a point in time, some of the frames in memory are in use and some are free. A list of free frames is maintained by the OS. Consider four processes with their pages as displayed below:



Process P

Process Q

Process R

Process S

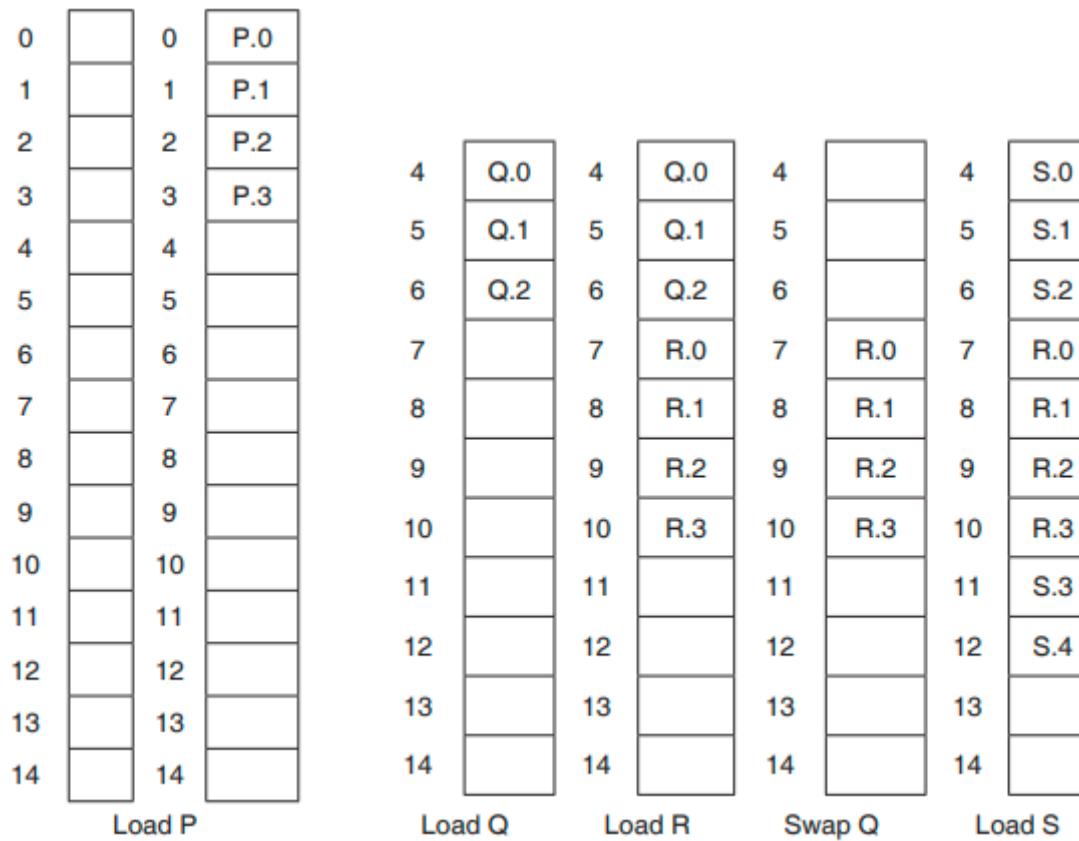
Paging avoids external fragmentation and the need for compaction





Non-contiguous Storage Allocation Methods-Paging

- Let the main memory consist of 15 frames: Main memory





Non-contiguous Storage Allocation Methods-Paging

- The OS maintains a page table for each process.
- The page table shows the frame location for each page of the process.
- Within a program, each logical address consists of **a page number** and an **offset** within the page.
- Here a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address.
- For this, the processor must know the following details:
 - Logical address: Consists page number and offset.
 - Page table: Used to produce physical address (Frame number, offset).





Non-contiguous Storage Allocation Methods-Paging

- In the previous example, the page tables of each process will be:

0		0	P.0
1		1	P.1
2		2	P.2
3		3	P.3
4			
5		4	Q.0
6		5	Q.1
7		6	Q.2
8		7	
9		8	
10		9	
11		10	
12		11	
13		12	
14		13	
		14	

Load P

4		4	Q.0
5		5	Q.1
6		6	Q.2
7		7	R.0
8		8	R.1
9		9	R.2
10		10	R.3
11		11	
12		12	
13		13	
14		14	

Load Q

4		4	Q.0
5		5	Q.1
6		6	Q.2
7		7	R.0
8		8	R.1
9		9	R.2
10		10	R.3
11		11	
12		12	
13		13	
14		14	

Load R

4		4	S.0
5		5	S.1
6		6	S.2
7		7	R.0
8		8	R.1
9		9	R.2
10		10	R.3
11		11	
12		12	
13		13	
14		14	

Swap Q

4		4	S.0
5		5	S.1
6		6	S.2
7		7	R.0
8		8	R.1
9		9	R.2
10		10	R.3
11		11	
12		12	
13		13	
14		14	

Load S

0	0	0	-	0	7
1	1	1	-	1	8
2	2	2	-	2	9
3	3	3		3	10

Process P

page table

0	0	0	-	0	7
1	1	1	-	1	8
2	2	2	-	2	9
3	3	3		3	10

Process Q

page table

0	0	0	-	0	7
1	1	1	-	1	8
2	2	2	-	2	9
3	3	3		3	10

Process R

page table

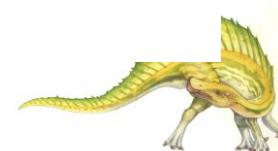
0	4
1	5
2	6
3	11
4	12

Process S

page table

13
14

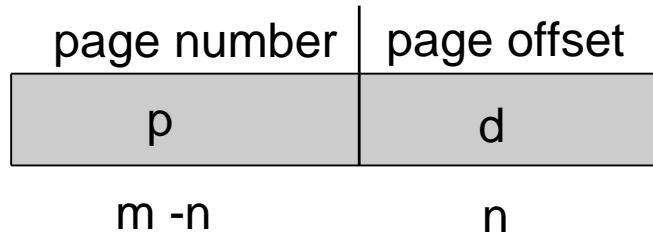
Free frame list.





Address Translation Scheme

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

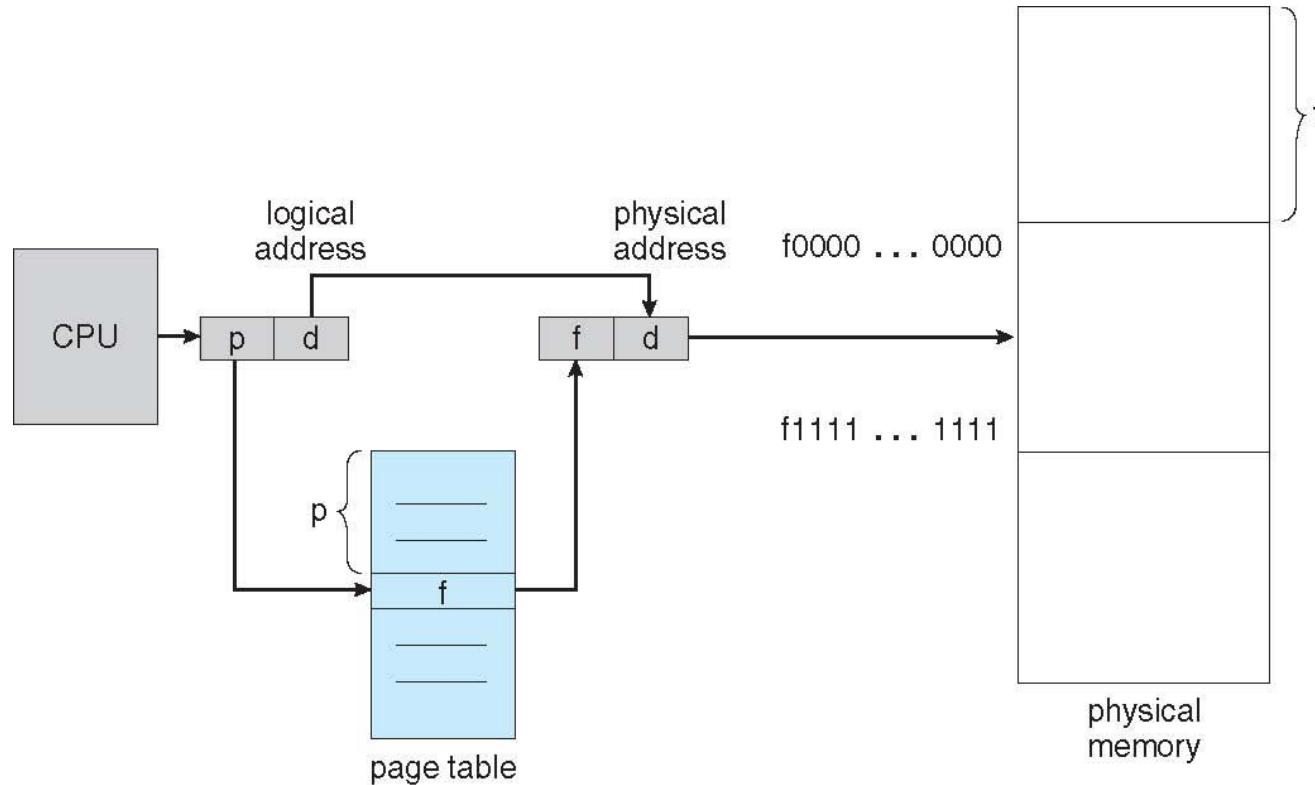


- For given logical address space 2^m and page size 2^n



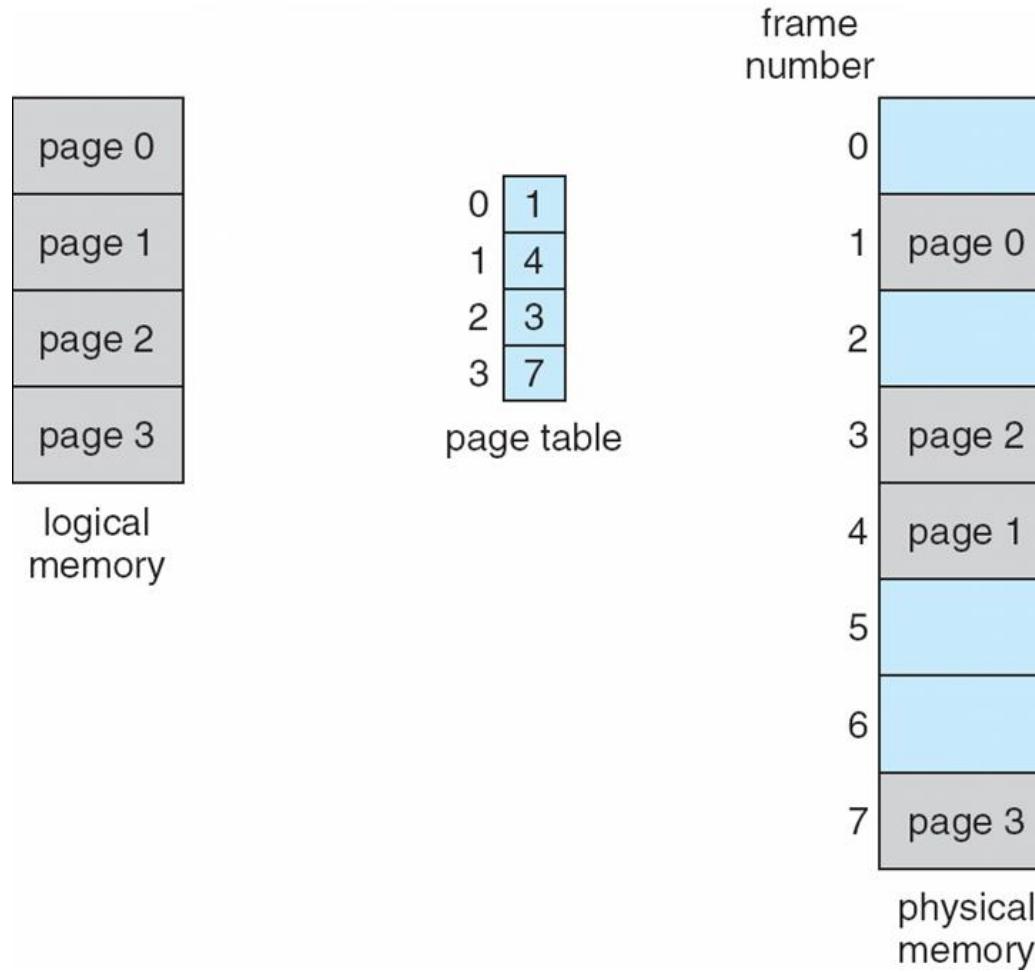


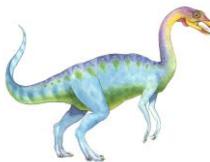
Paging Hardware





Paging Model of Logical and Physical Memory





Paging Example

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

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	
8	
12	
16	
20	a
21	b
22	c
23	d
24	e
25	f
26	g
27	h
28	

physical memory

$n=2$ and $m=5$ 32-byte memory and 4-byte pages





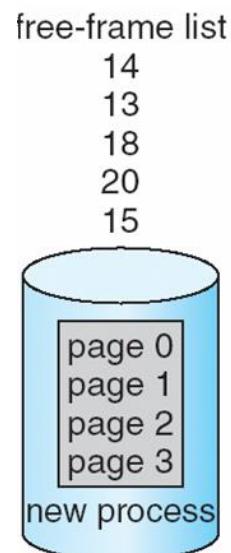
Paging (Cont.)

- Calculating internal fragmentation
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of $2,048 - 1,086 = 962$ bytes
 - Worst case fragmentation = 1 frame – 1 byte
 - On average fragmentation = 1 / 2 frame size
 - So small frame sizes desirable?
 - But each page table entry takes memory to track
 - Page sizes growing over time
 - ▶ **Solaris supports two page sizes – 8 KB and 4 MB**
- By implementation process can only access its own memory

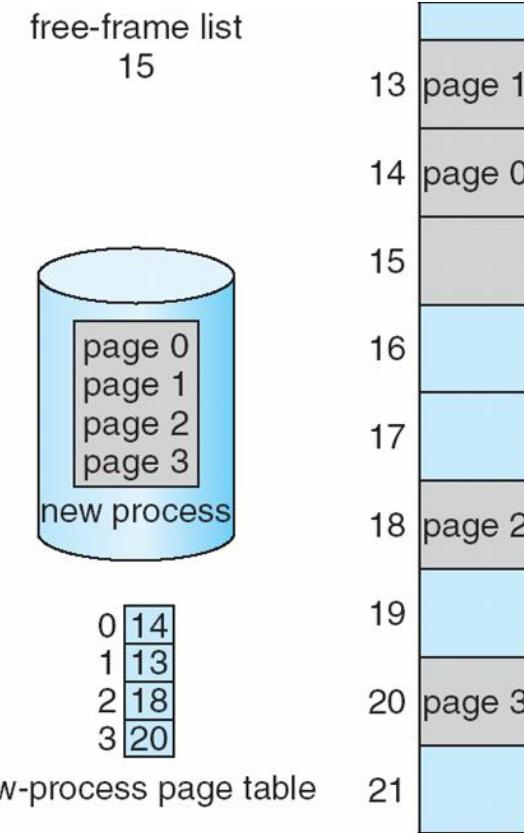
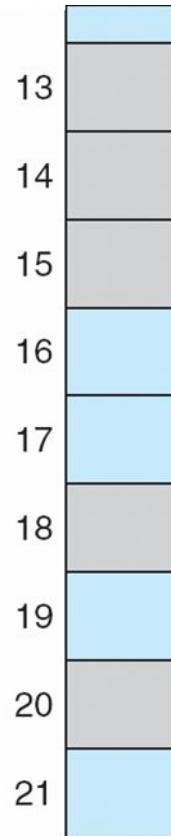




Free Frames



(a)

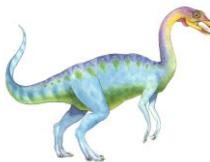


(b)

Before allocation

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

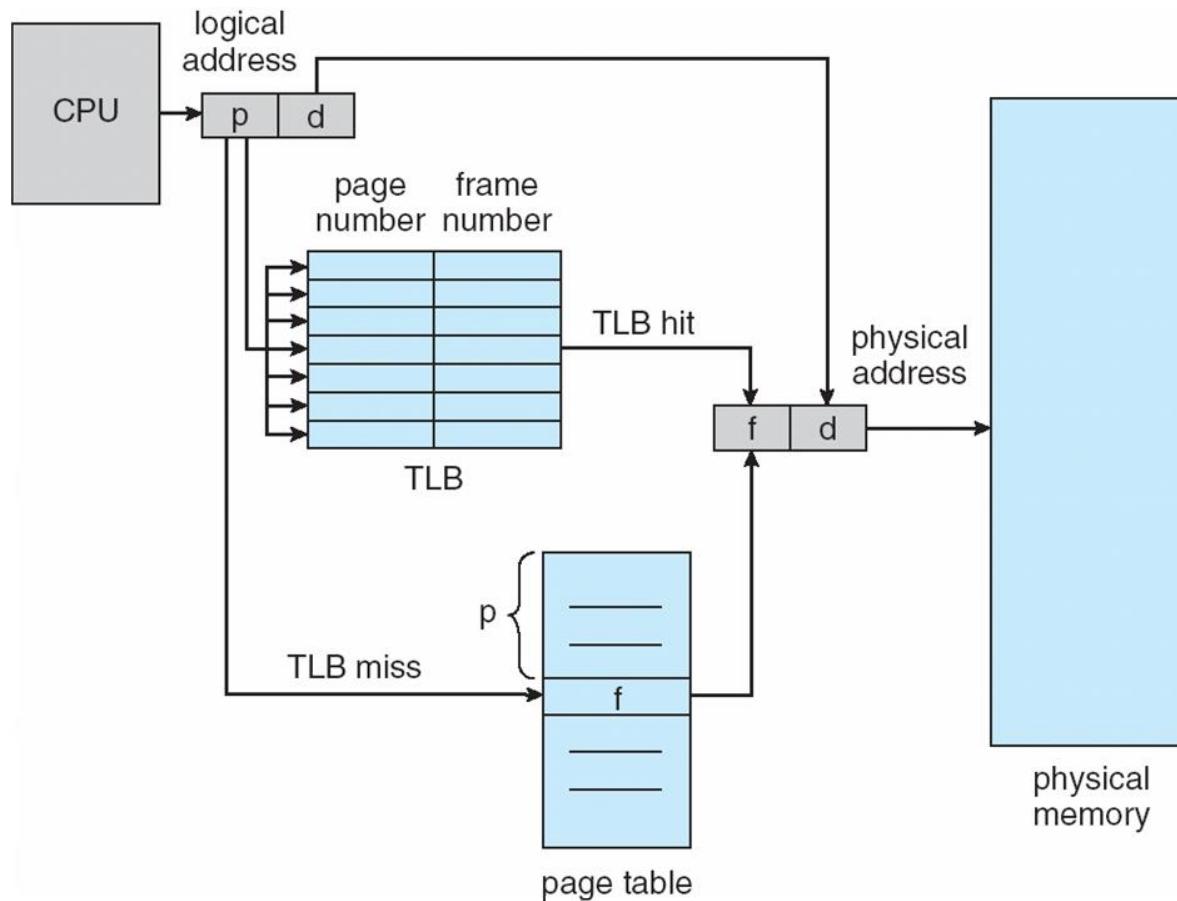
Page #	Frame #

- 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





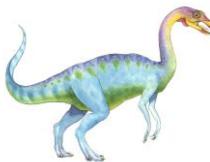
Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more)
- **Effective Access Time (EAT)**

$$\begin{aligned}\text{effective access time} &= 0.80 \times 100 + 0.20 \times 200 \\ &= 120 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).





Steps for address translation

- Extract the page number from the logical address.
- Use the page number as an index into the process page table to find the frame number.
- The physical address will be constructed by appending the frame number to the offset.
- Advantage
 - There is no external fragmentation.
- Disadvantage
 - There is a small amount of internal fragmentation.





Memory Protection

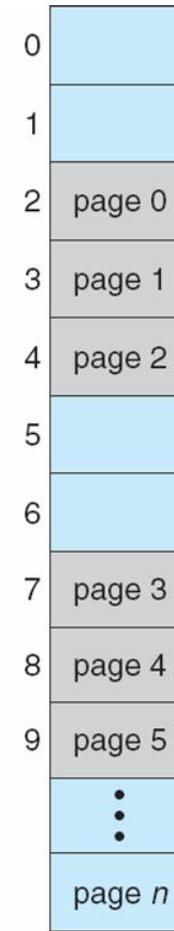
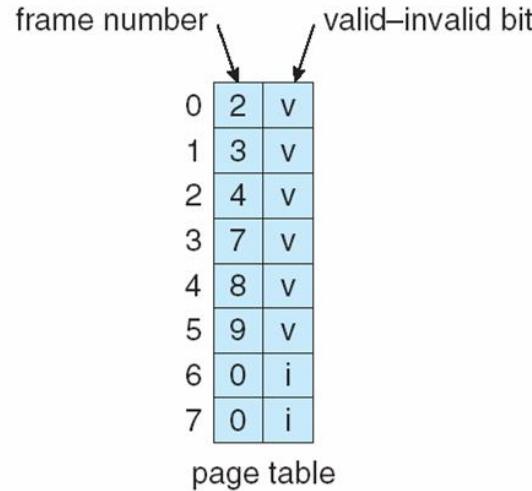
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated 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

□ 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

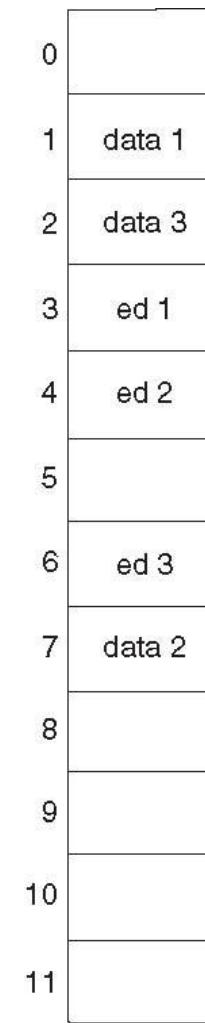
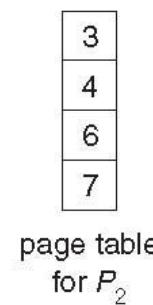
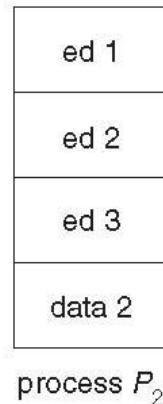
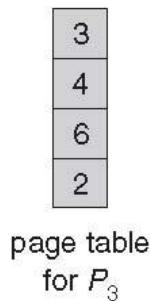
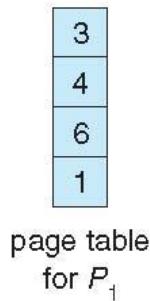
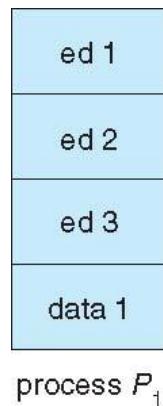
□ Private code and data

- 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





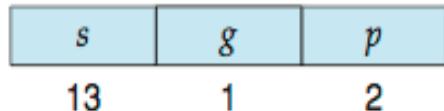
Shared Pages Example





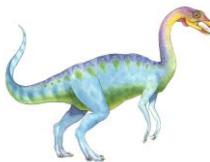
Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
 - Selector given to segmentation unit
 - ▶ Which produces linear addresses

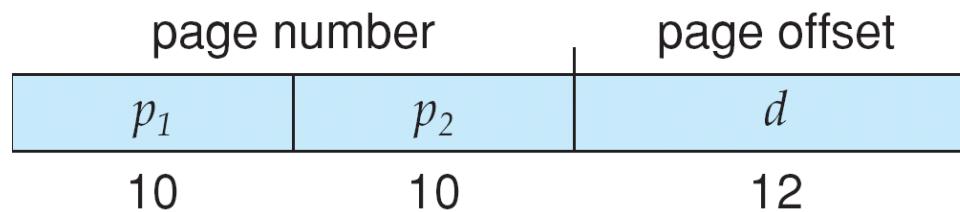
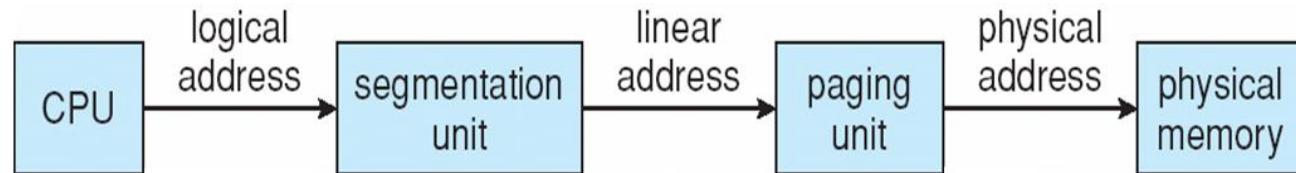


- Linear address given to paging unit
 - ▶ Which generates physical address in main memory
 - ▶ Paging units form equivalent of MMU
 - ▶ Pages sizes can be 4 KB or 4 MB



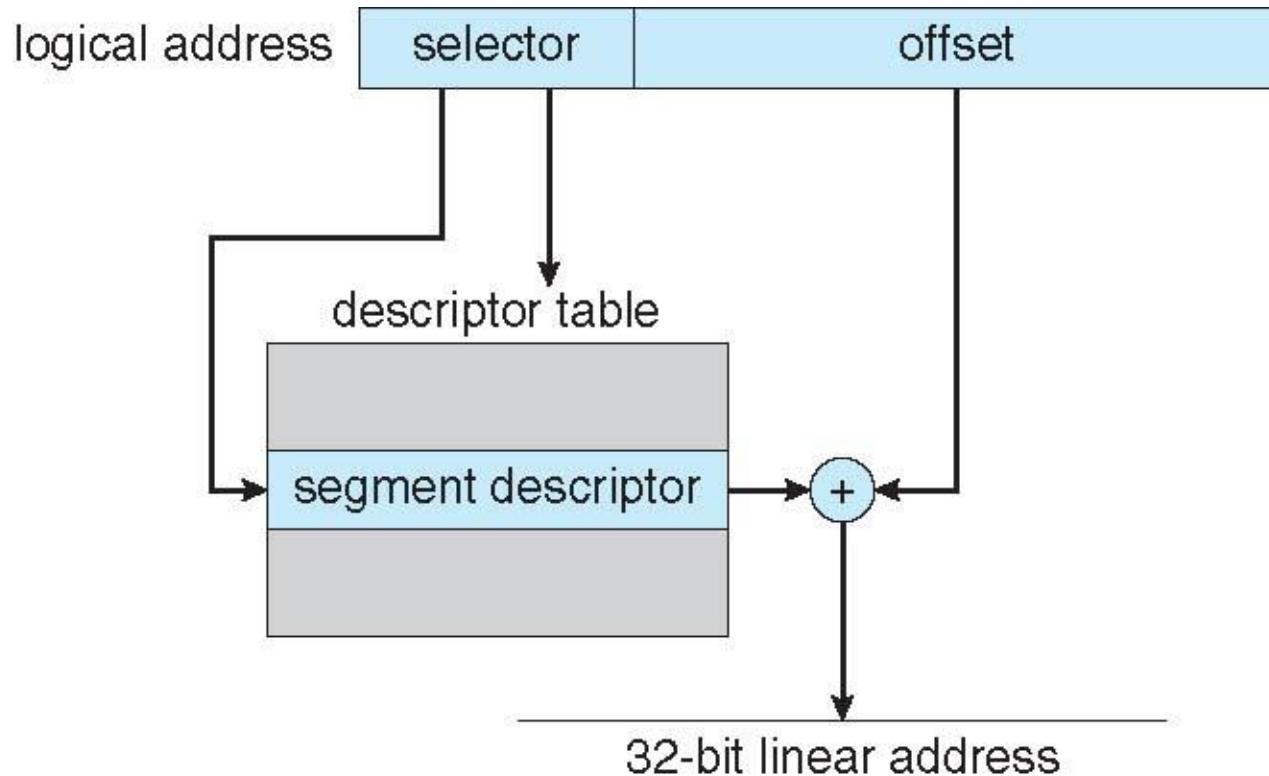


Logical to Physical Address Translation in IA-32



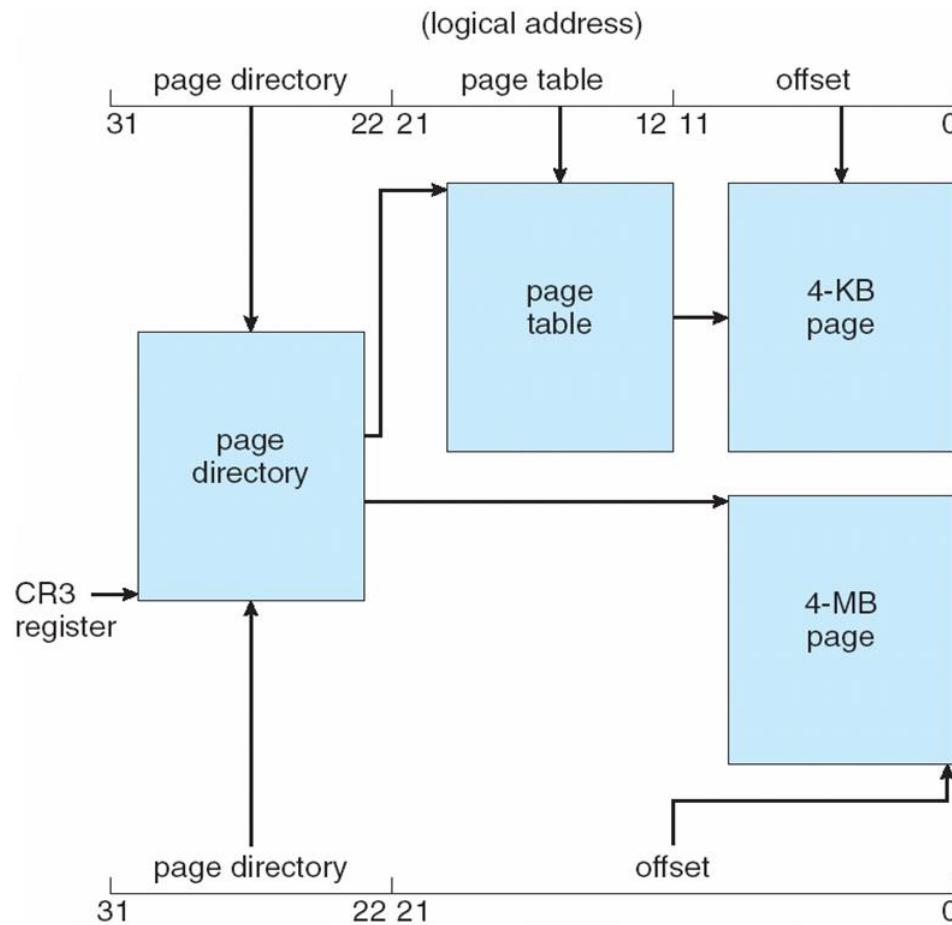


Intel IA-32 Segmentation

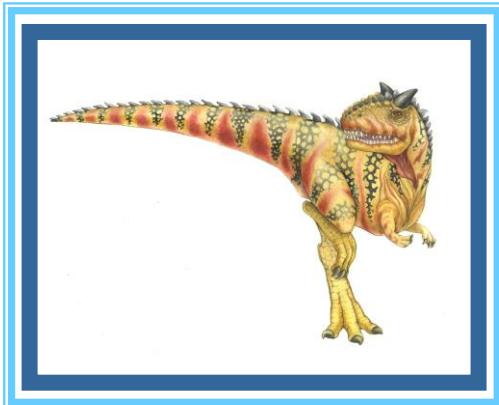




Intel IA-32 Paging Architecture



Virtual Memory





Background

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory -> each user program runs faster

Thus, running a program that is not entirely in memory would benefit both the system and the user





Background (Cont.)

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





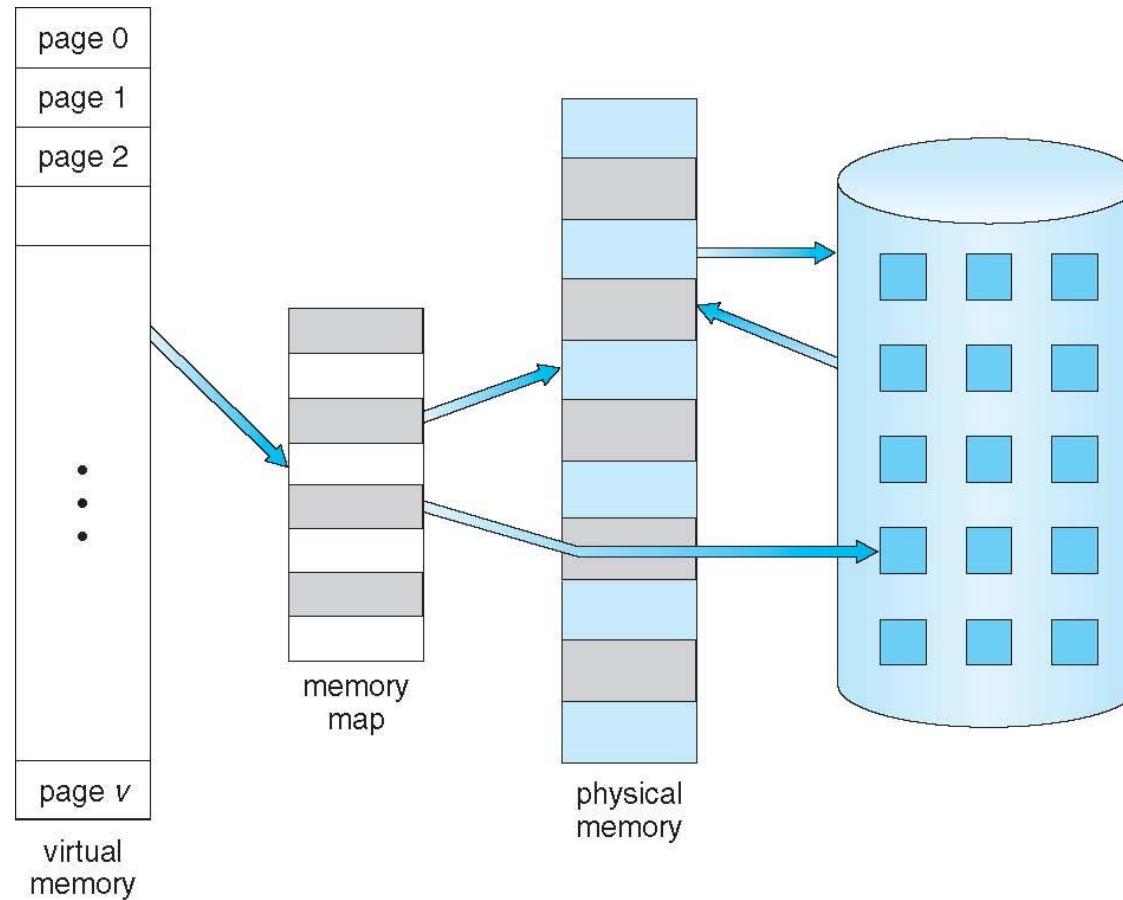
Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation





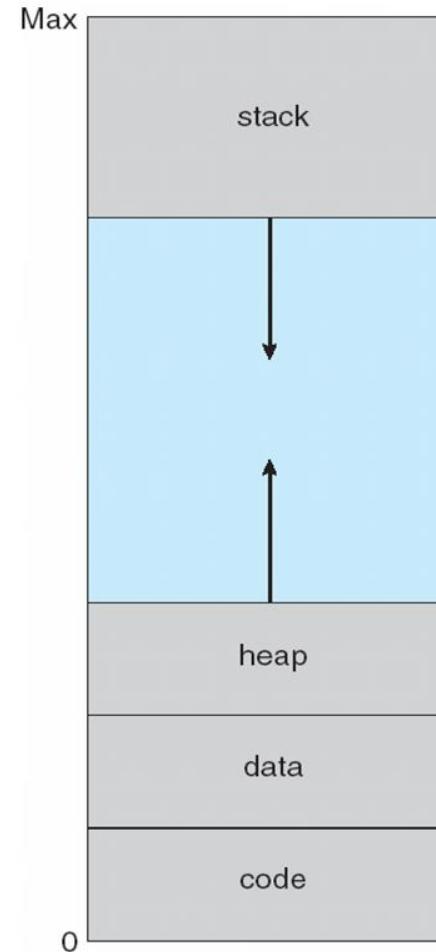
Virtual Memory That is Larger Than Physical Memory

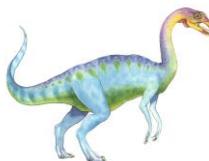




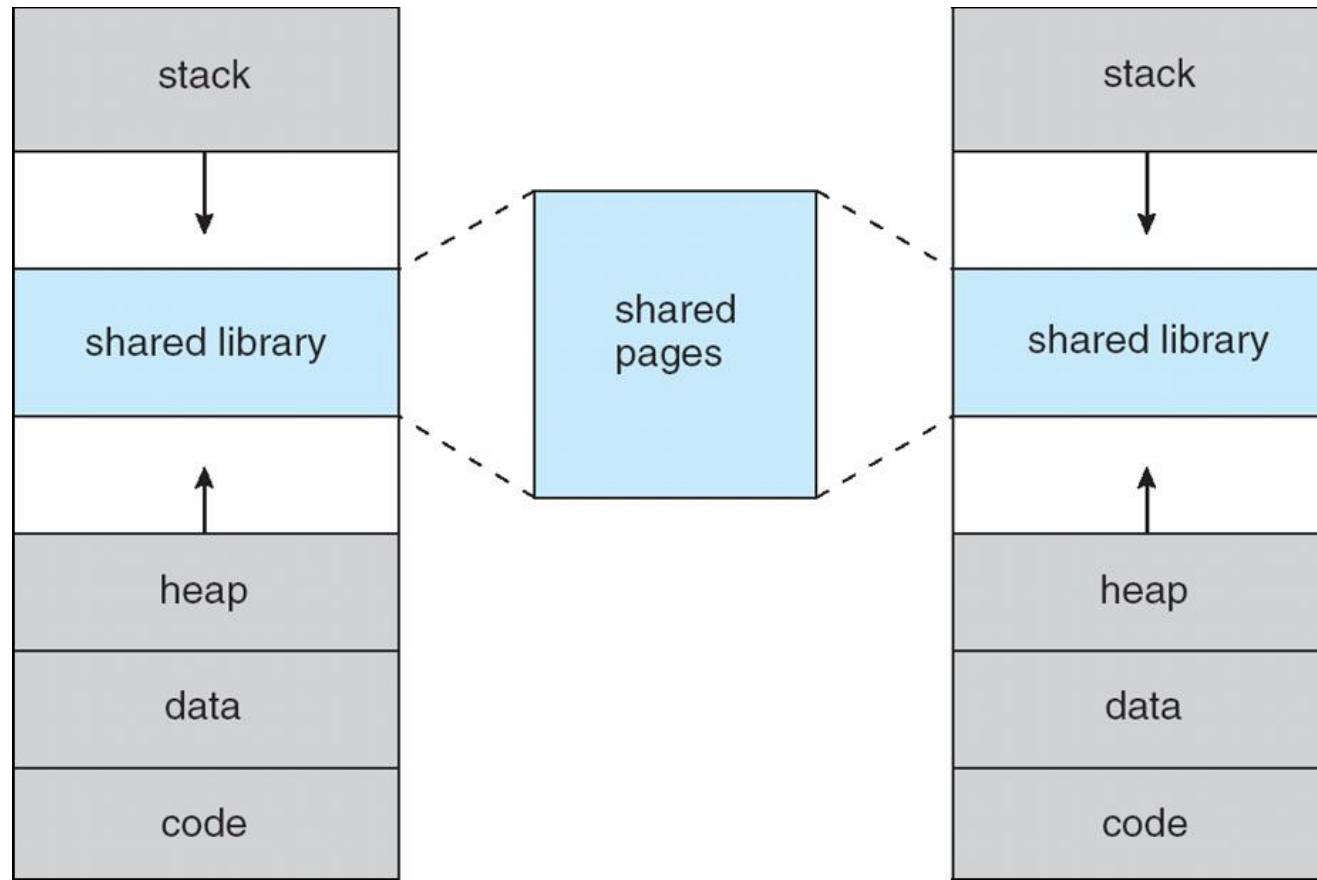
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is hole
 - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation





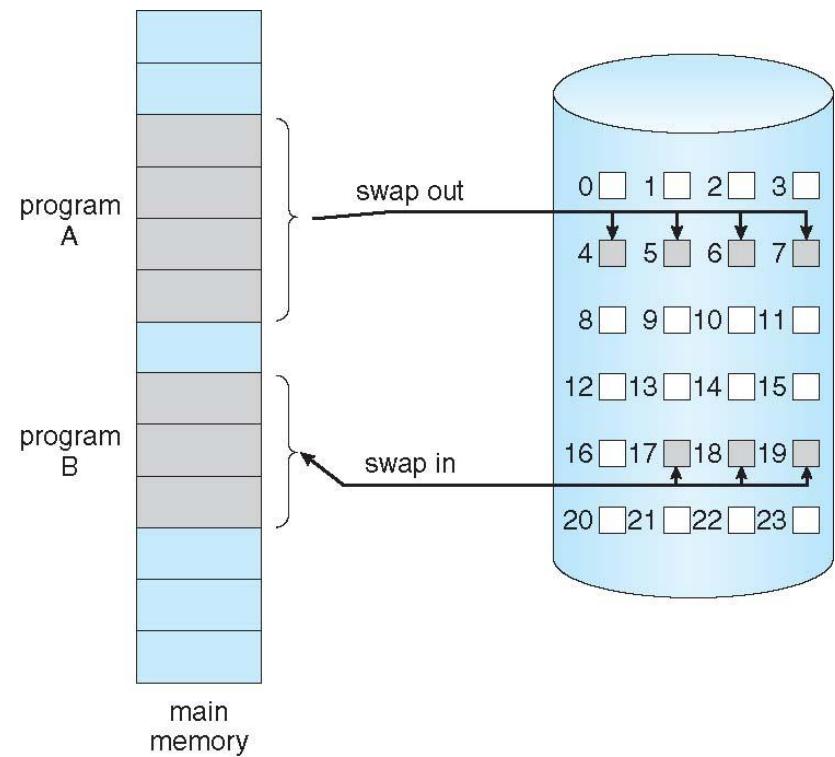
Shared Library Using Virtual Memory





Demand Paging

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

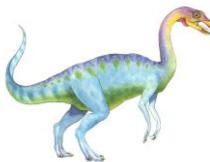




Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - ▶ Without changing program behavior
 - ▶ Without programmer needing to change code





Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

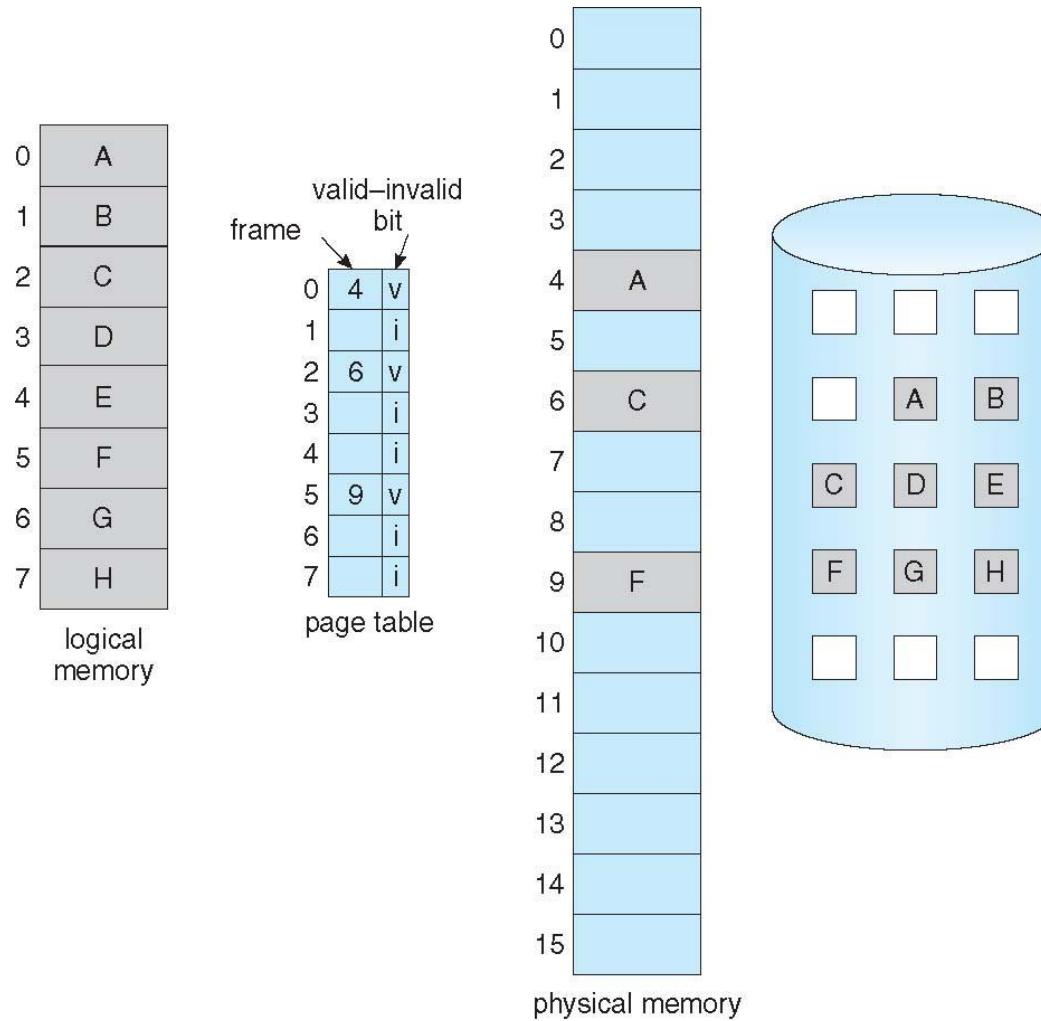
page table

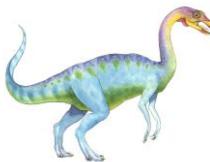
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault





Page Table When Some Pages Are Not in Main Memory





Page Fault

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

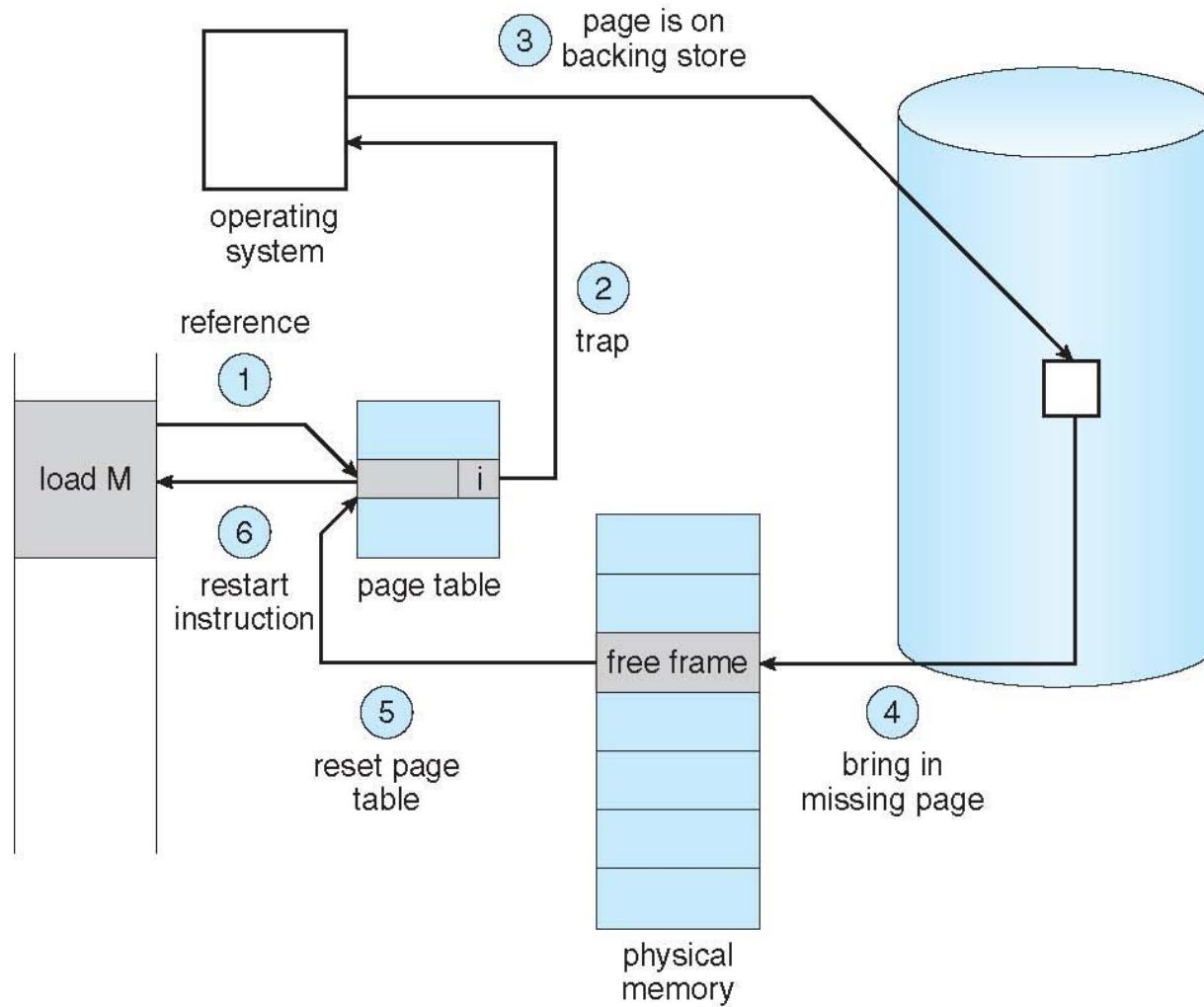
page fault

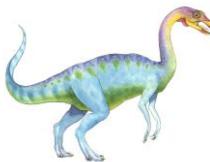
1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
Set validation bit = **V**
5. Restart the instruction that caused the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





Steps in Handling a Page Fault





Performance of Demand Paging

□ Stages in Demand Paging (worse case)

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





Performance of Demand Paging (Cont.)

- Three major activities
 - Service the interrupt – careful coding means just several hundred instructions needed
 - Read the page – lots of time
 - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in}) \end{aligned}$$





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p \text{ (8 milliseconds)} \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then
 $\text{EAT} = 8.2 \text{ microseconds.}$
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
 - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses

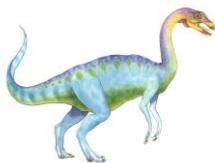




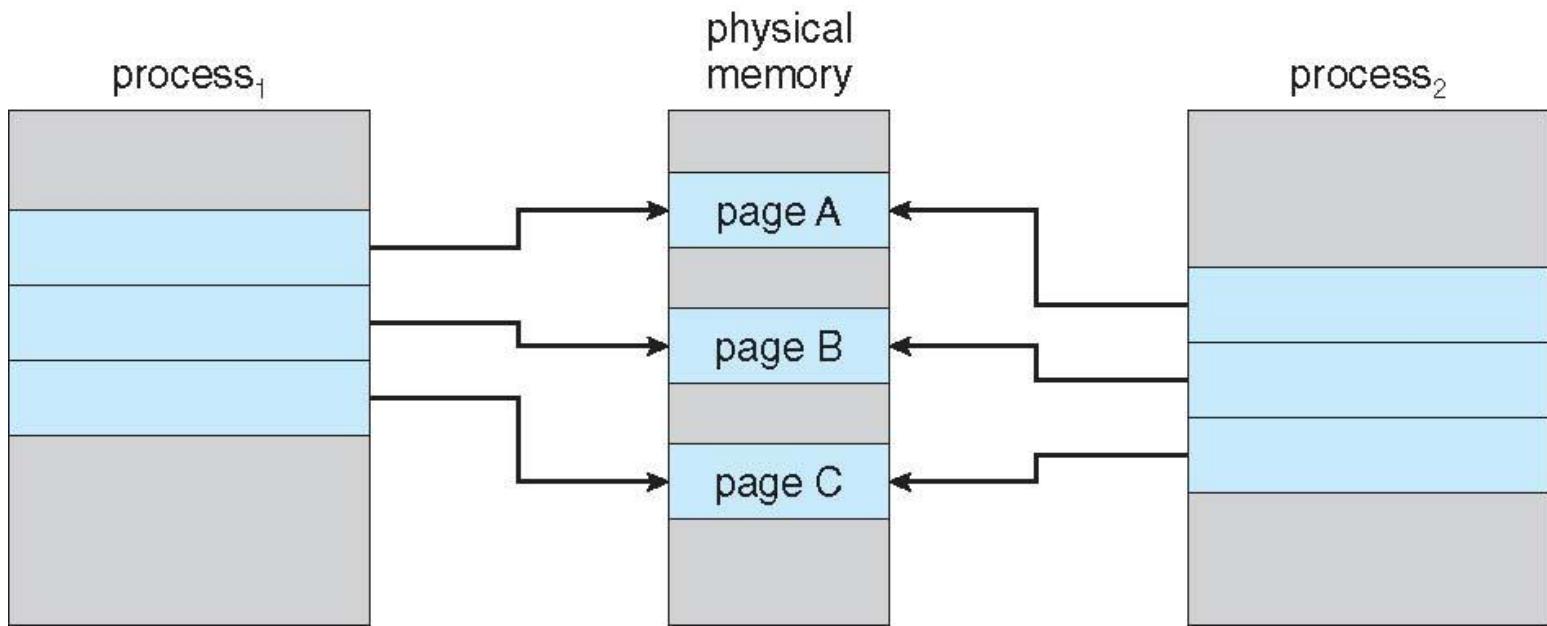
Copy-on-Write

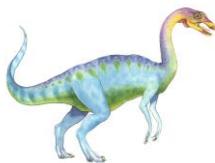
- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
 - Pool should always have free frames for fast demand page execution
 - ▶ Don't want to have to free a frame as well as other processing on page fault
 - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
 - Designed to have child call `exec()`
 - Very efficient



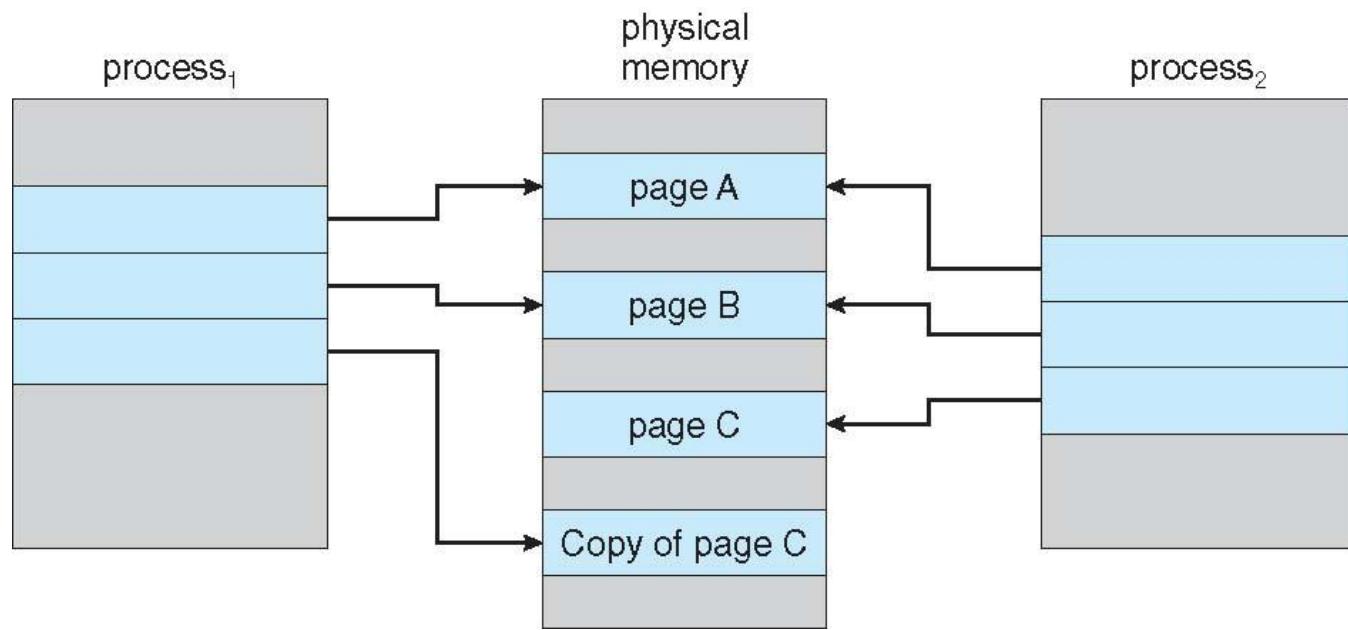


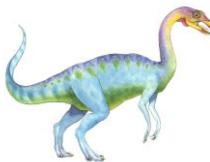
Before Process 1 Modifies Page C





After Process 1 Modifies Page C





What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times





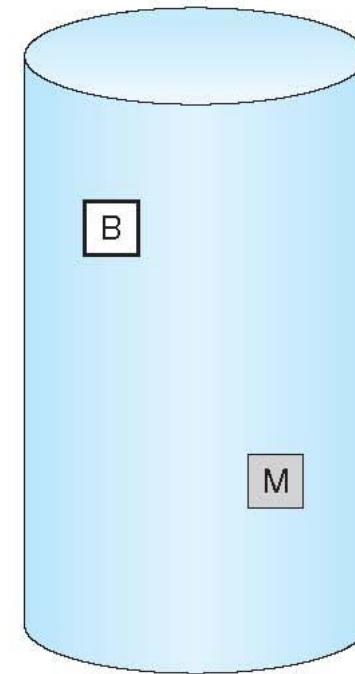
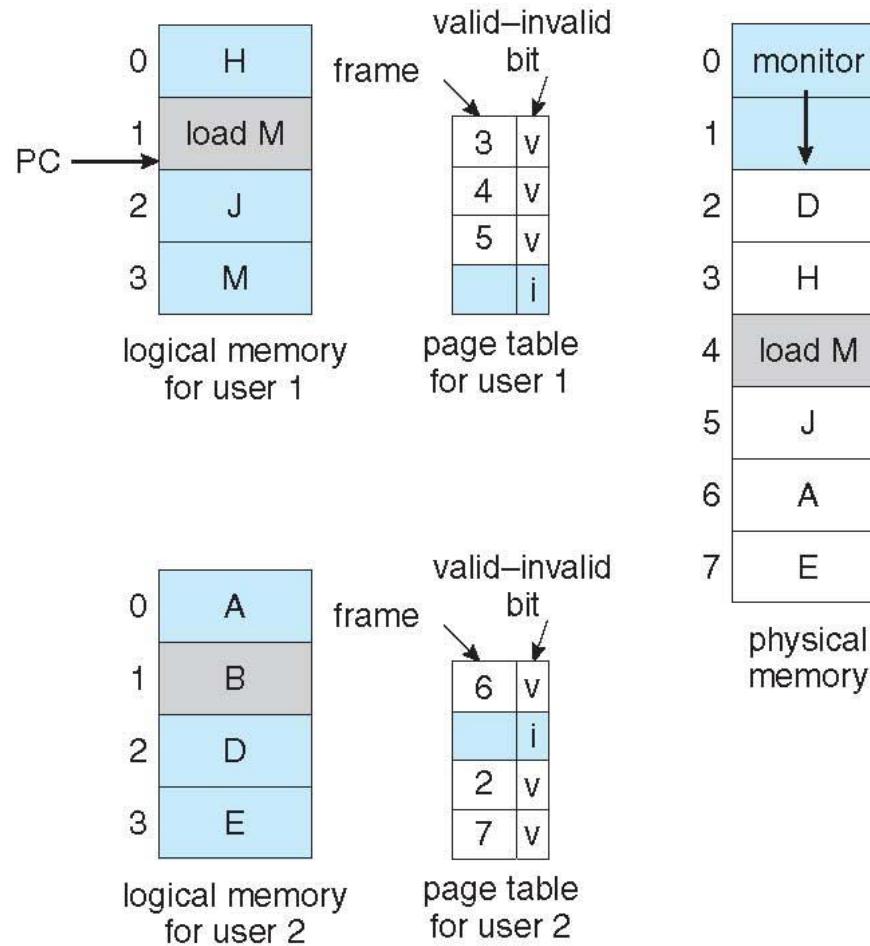
Page Replacement

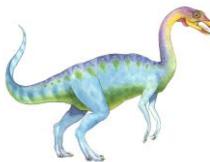
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory





Need For Page Replacement





Basic Page Replacement

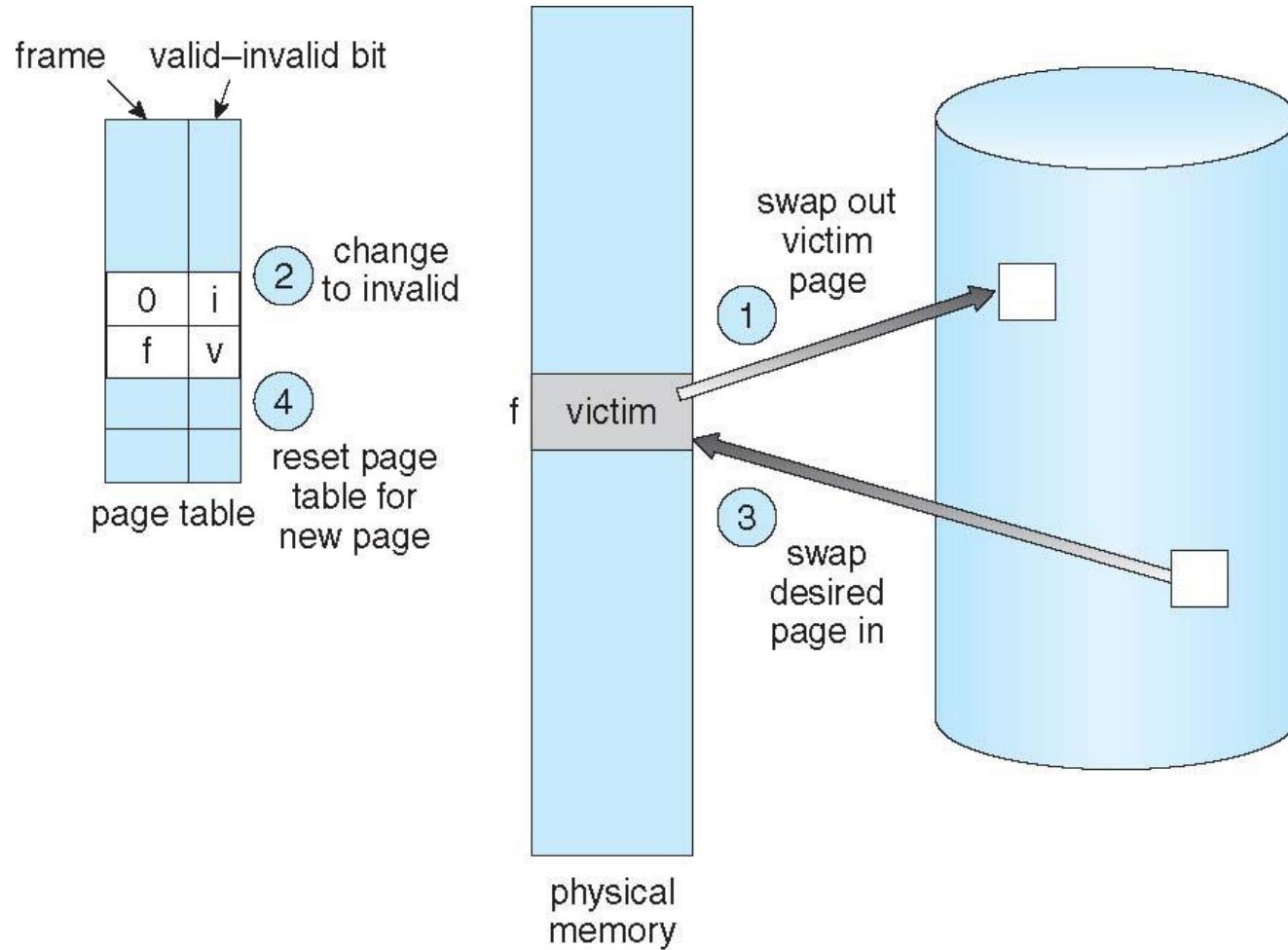
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the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

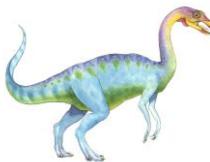
Note now potentially 2 page transfers for page fault – increasing EAT





Page Replacement





Page and Frame Replacement Algorithms

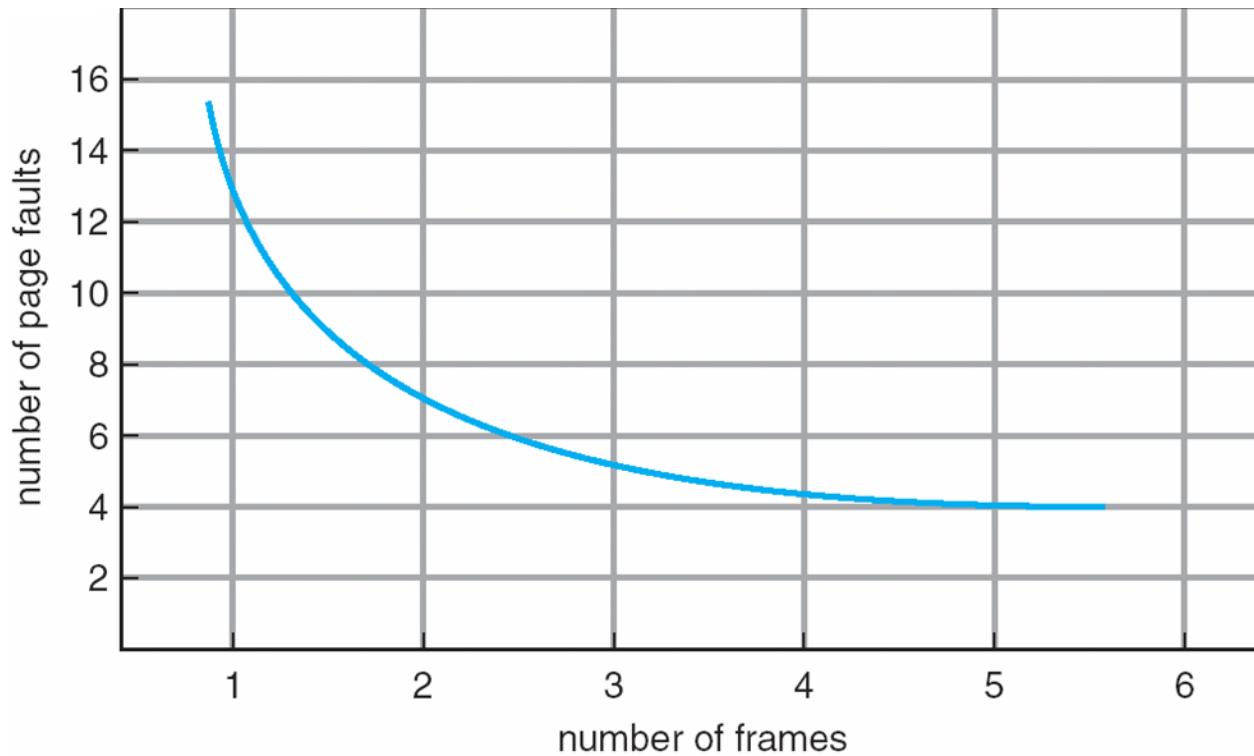
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

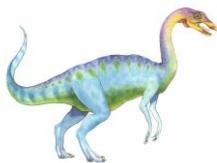
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1





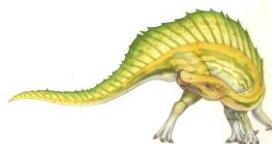
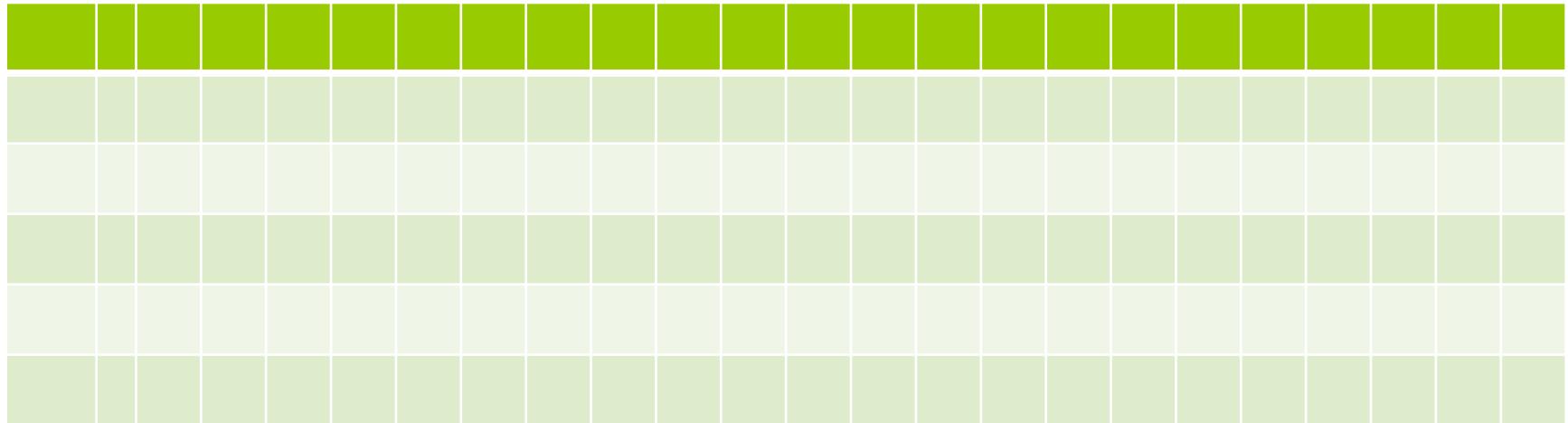
Graph of Page Faults Versus The Number of Frames





First-In-First-Out (FIFO) Algorithm

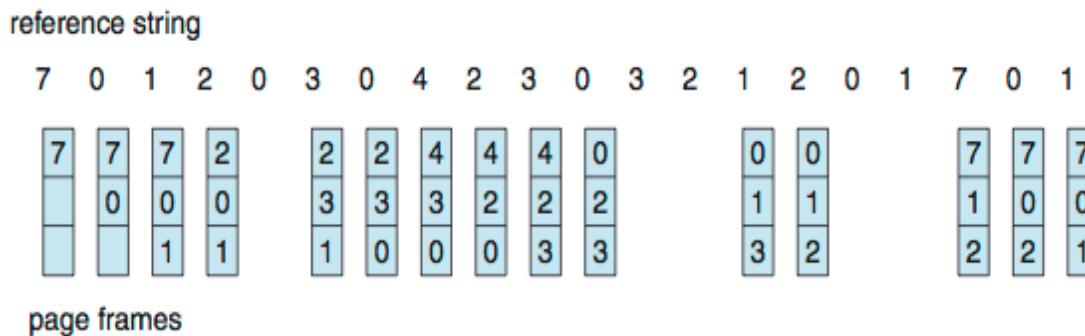
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)





First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



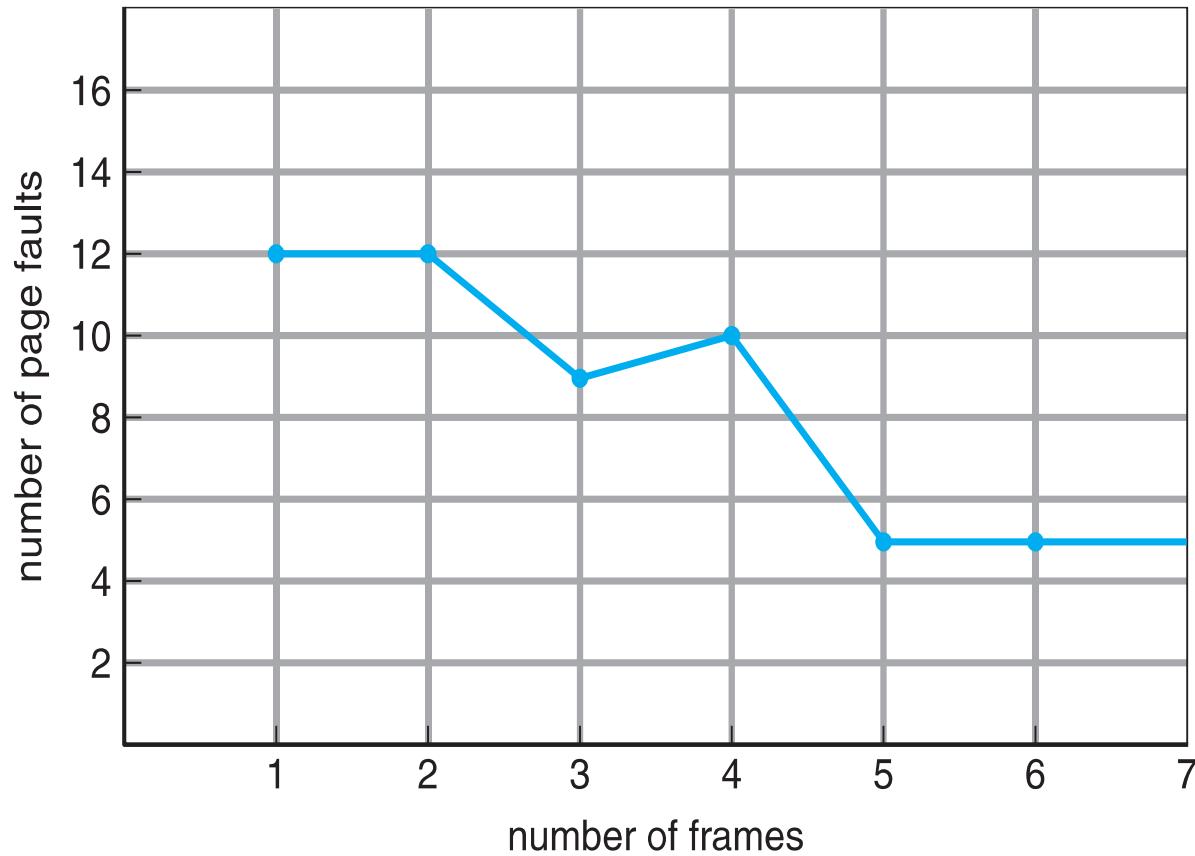
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - ▶ **Belady's Anomaly**
 - How to track ages of pages?
 - Just use a FIFO queue





FIFO Illustrating Belady's Anomaly



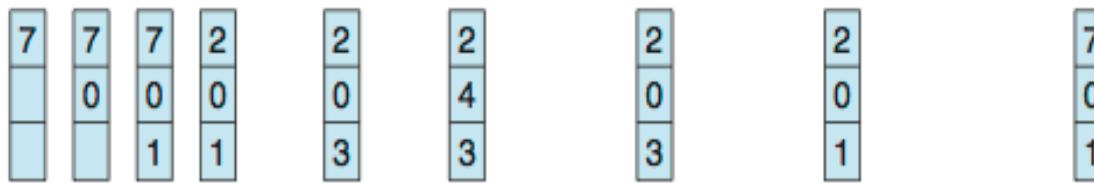


Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames



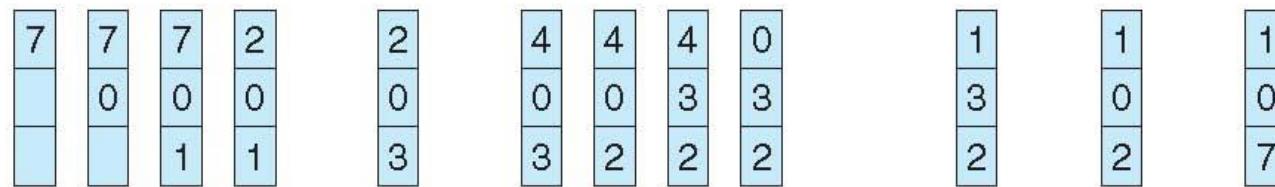


Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





LRU Algorithm (Cont.)

- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value
 - ▶ Search through table needed
- Stack implementation
 - Keep a stack of page numbers in a double link form:
 - Page referenced:
 - ▶ move it to the top
 - ▶ requires 6 pointers to be changed
 - But each update more expensive
 - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

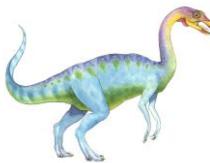




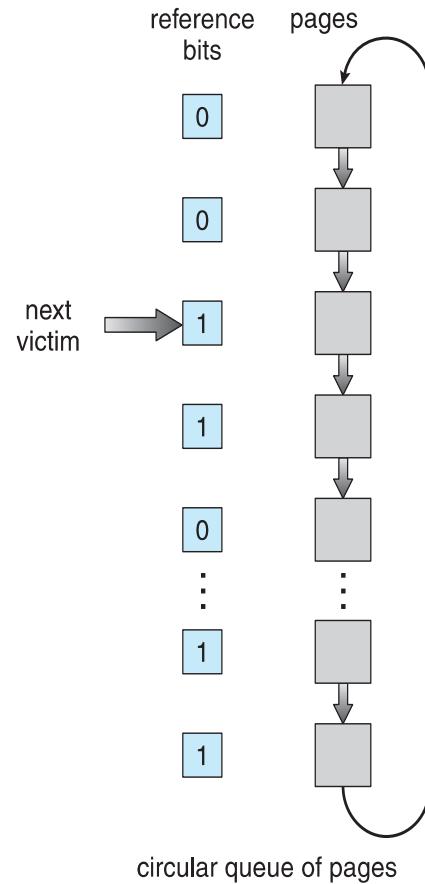
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order, however
- **Second-chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - **Clock** replacement
 - If page to be replaced has
 - ▶ Reference bit = 0 -> replace it
 - ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

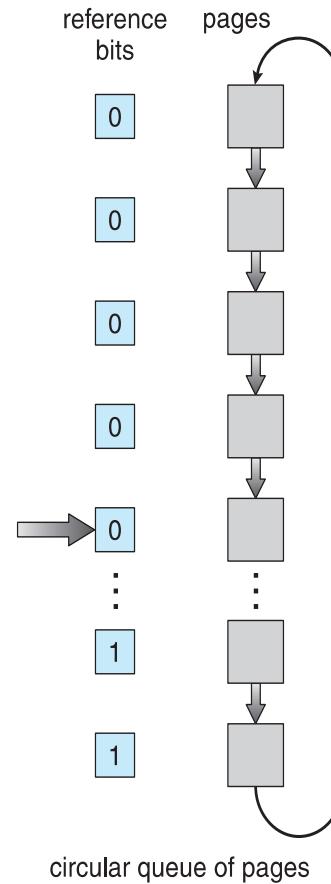




Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)





Second-Chance (clock) Page-Replacement Algorithm

Reference string: **2,3,2,1,5,2,4,5,3,2,3,5**

3 frames (3 pages can be in memory at a time per process)





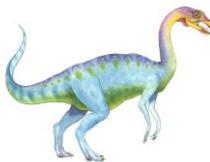
Second-Chance (clock) Page-Replacement Algorithm

Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

3 frames (3 pages can be in memory at a time per process)

7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2	7	7	7
1	1	1	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1	1



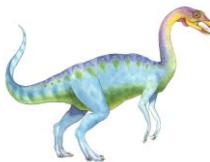


LRU counter implementation

Reference string: **1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6**
4 frames (4 pages can be in memory at a time per process)

	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	2	3	4	4	1	2	3	4	1	2	3								
2	0	1	2	3	1	2	3	4	1	2	1	2								
3	0	1	2	3	4	0	0	0	0	0	0	1								
4	0		1	2	3	4	0	0	0	0	0	0								
5	0			1	2	3	4	4	4	0										
6						1	2	3	3	4										
7																				
h/ m																				





Counting Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **Least Frequently Used (LFU) Algorithm:** replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used





Allocation of Frames

- Each process needs **minimum** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- **Maximum** of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations





Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
 - Dynamic as degree of multiprogramming, process sizes change
 - s_i = size of process p_i
 - $S = \sum s_i$
 - m = total number of frames
 - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$





Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
 - select for replacement one of its frames
 - select for replacement a frame from a process with lower priority number





Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory

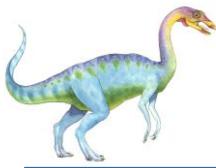




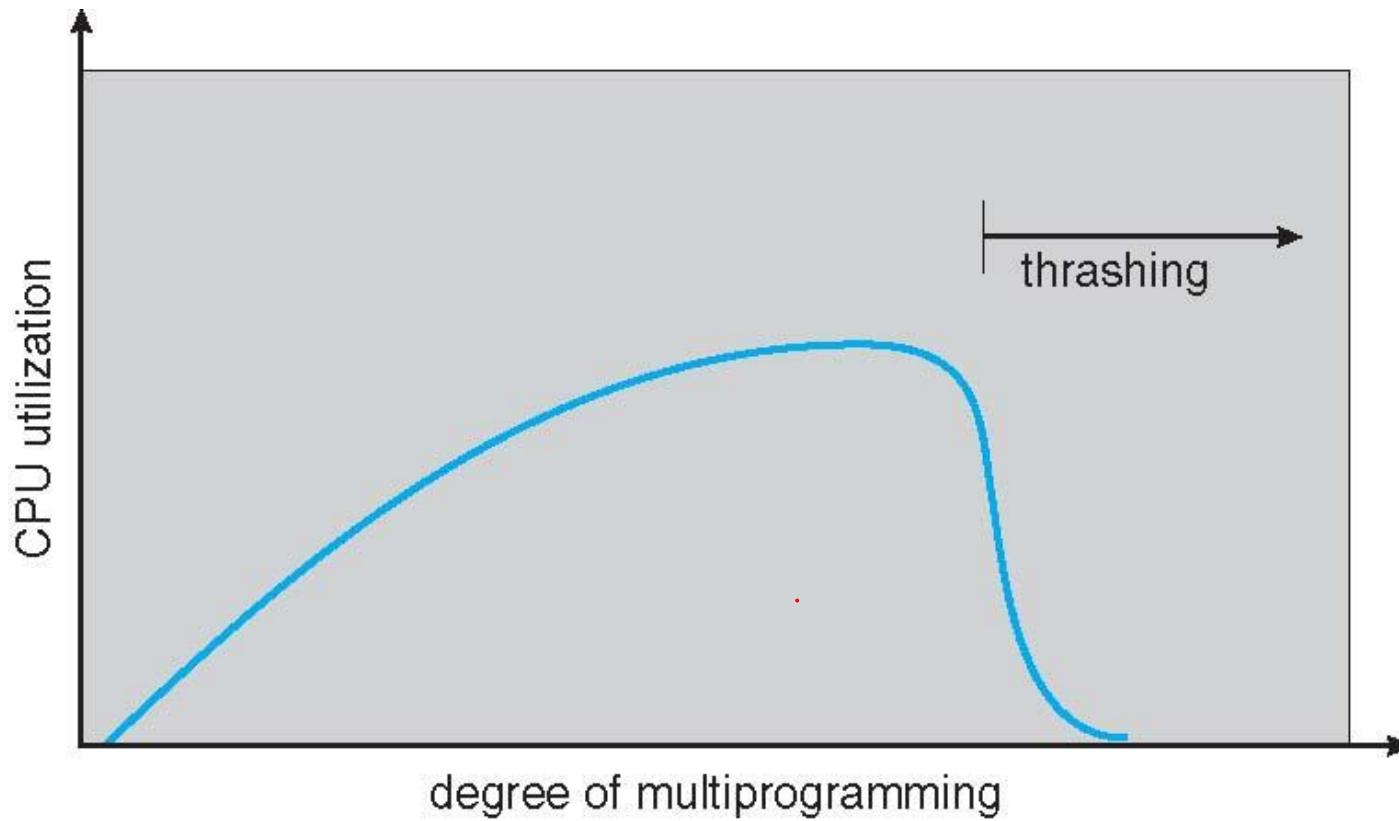
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
 - This leads to:
 - ▶ Low CPU utilization
 - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
 - ▶ Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out





Thrashing (Cont.)





Demand Paging and Thrashing

- Why does demand paging work?

Locality model

- Process migrates from one locality to another
 - Localities may overlap

- Why does thrashing occur?

Σ size of locality > total memory size

- Limit effects by using local or priority page replacement



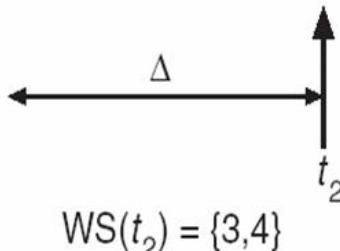
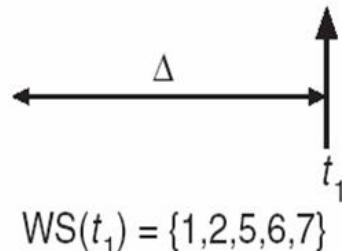


Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

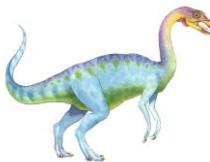




Keeping Track of the Working Set

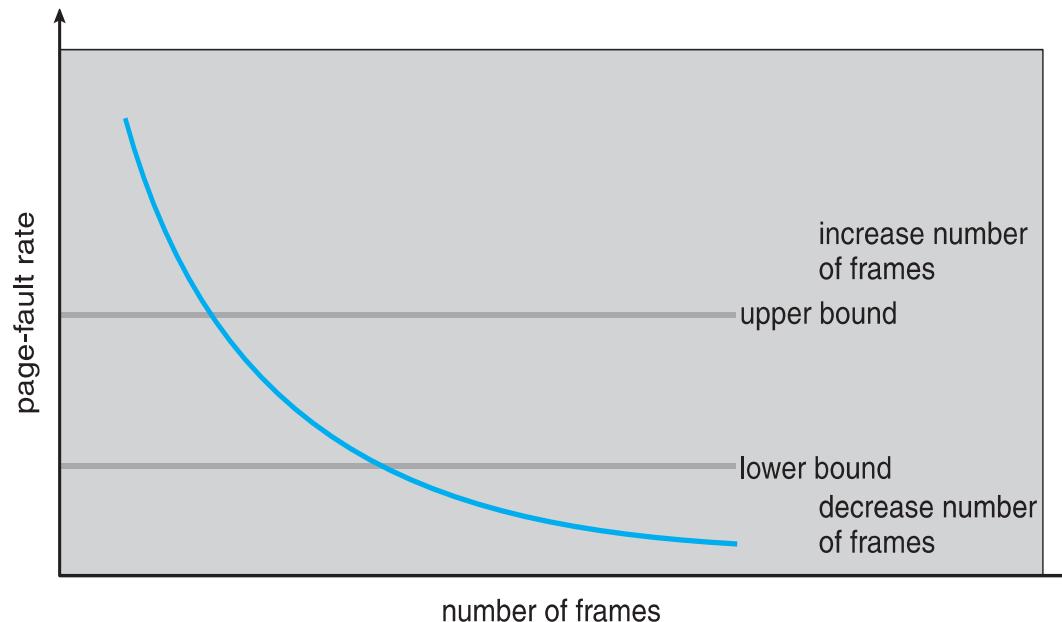
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

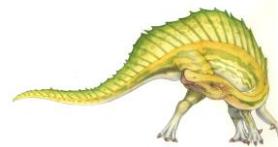
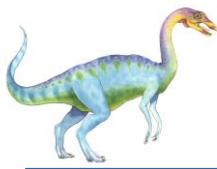




Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame

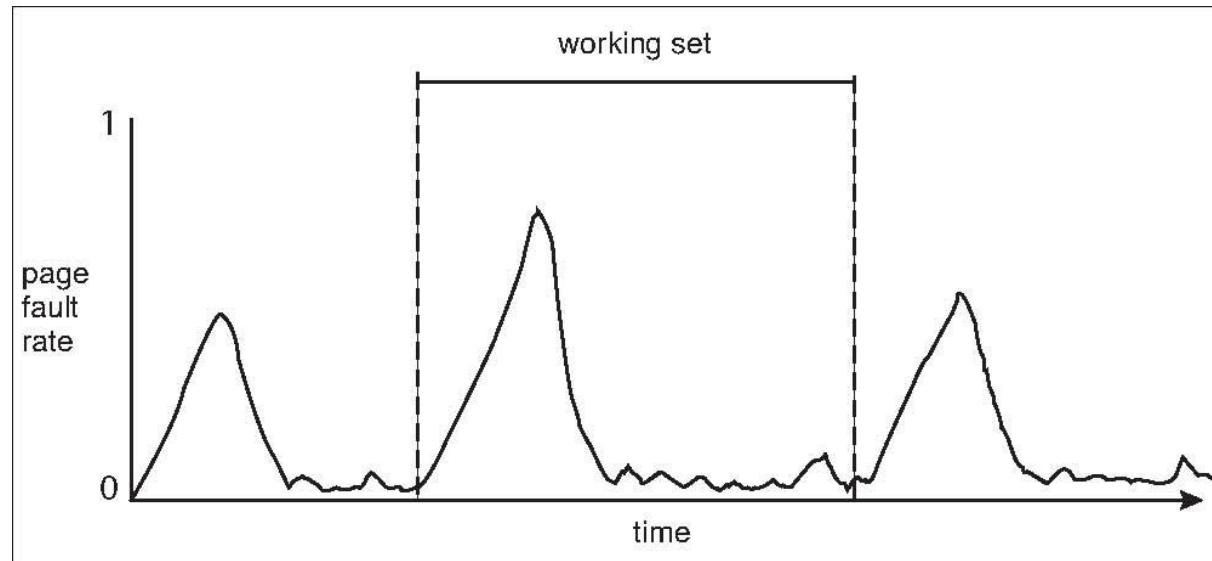






Working Sets and Page Fault Rates

- n Direct relationship between working set of a process and its page-fault rate
- n Working set changes over time
- n Peaks and valleys over time





Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

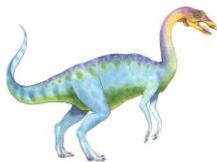




Solaris

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to begin swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages





Solaris 2 Page Scanner

