MODULE-I

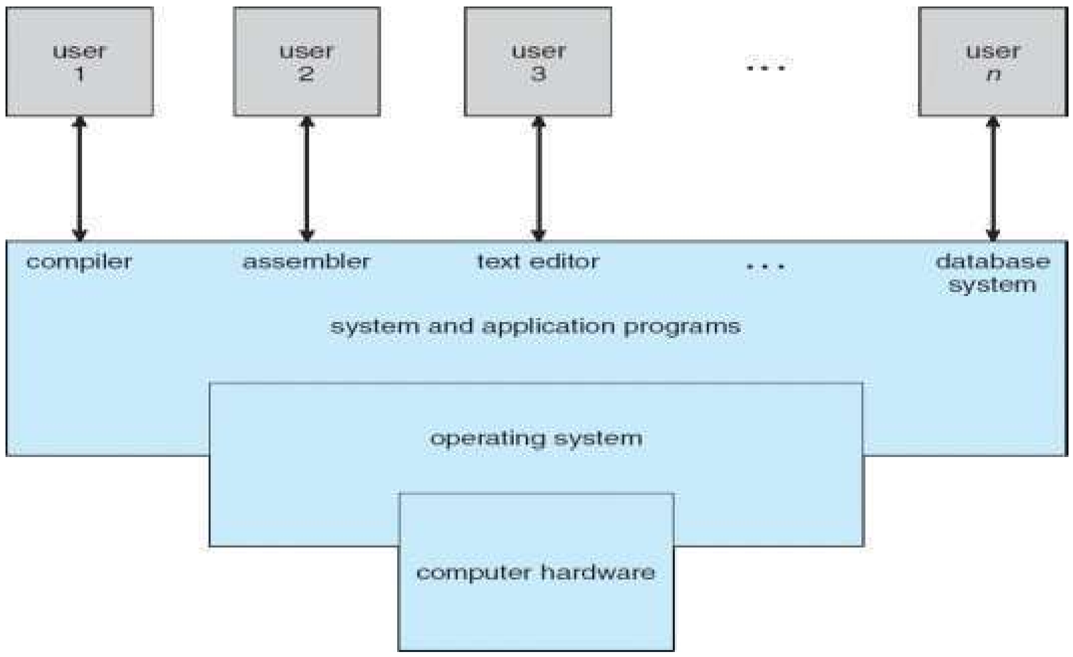
Introduction to OS

A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

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



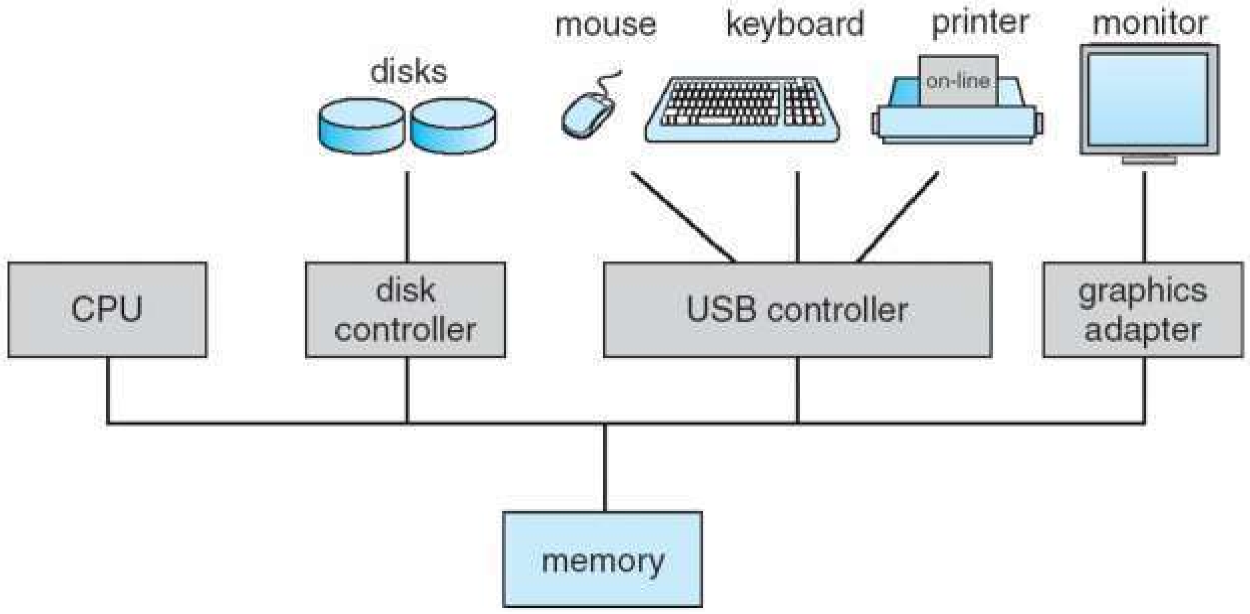
OS Definition

- 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

Computer Startup

- bootstrap program is loaded at power-up or reboot
    - Typically stored in ROM or EPROM, generally known as firmware
    - Initializes all aspects of system
    - Loads operating system kernel and starts execution

Computer System Organisation



- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles
- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*
- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*

- A *trap* is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**
- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- **polling**
- **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

I/O Structure

- After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the operating system to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity

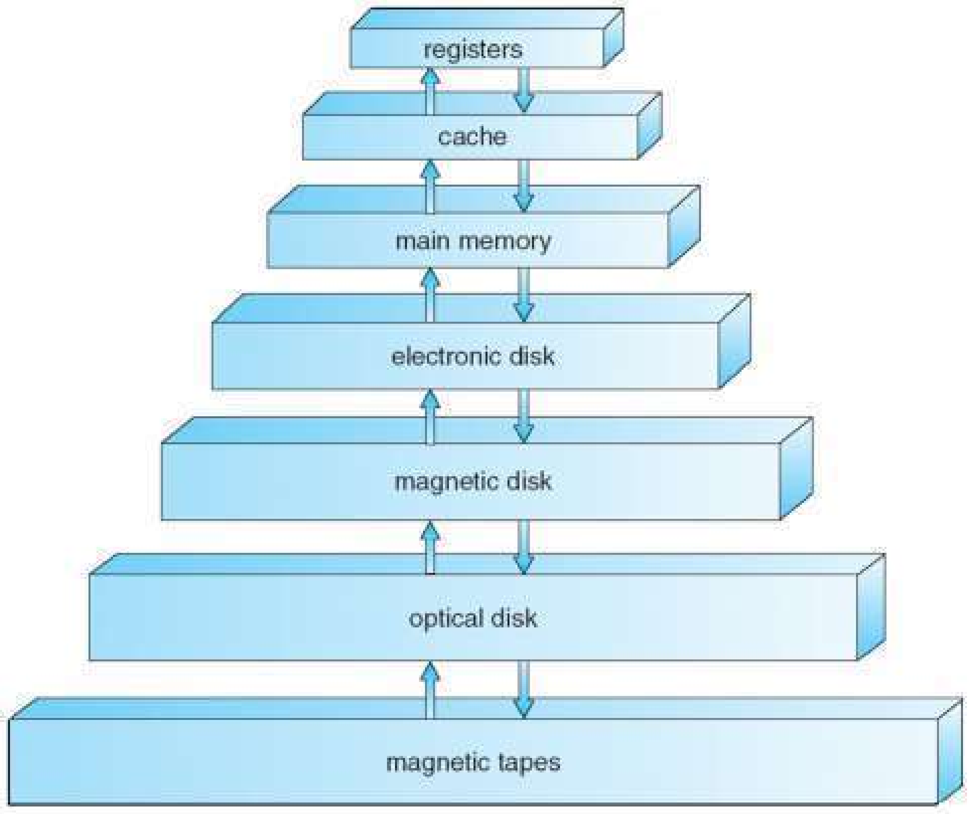
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

Storage Hierarchy

- Storage systems organized in hierarchy
    - Speed
    - Cost
    - Volatility



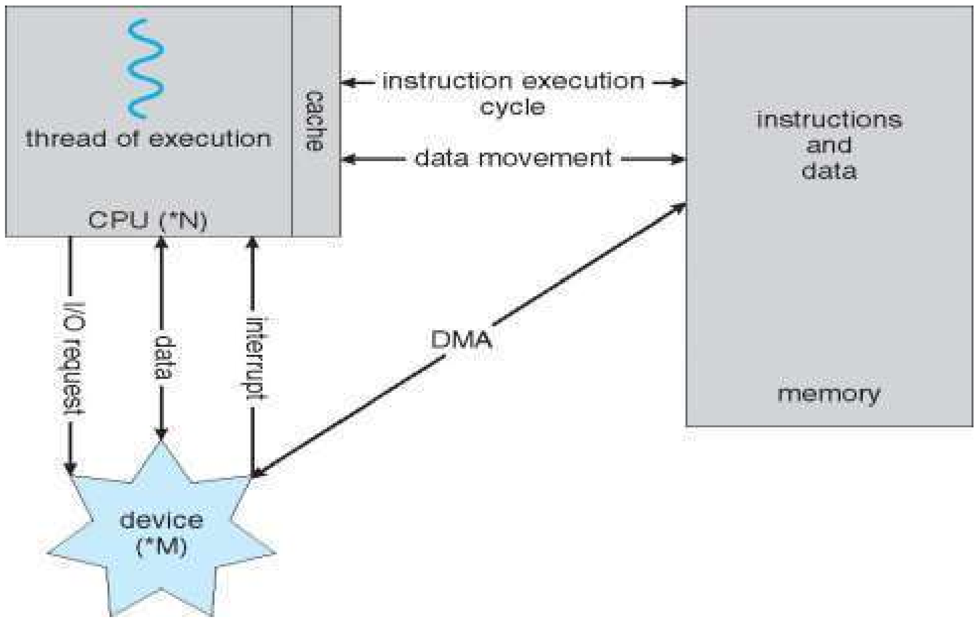
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (fast)
    - If not, data copied to cache and used there
- Cache smaller than storage being cached
    - Cache management important design problem
    - Cache size and replacement policy
    - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
    - The **disk controller** determines the logical interaction between the device and the computer

Computer System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
    - Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
    - Also known as parallel systems, tightly-coupled systems
    - Advantages include
        - Increased throughput
        - Economy of scale
        - Increased reliability – graceful degradation or fault tolerance
    - Two types
        - Asymmetric Multiprocessing

- Symmetric Multiprocessing



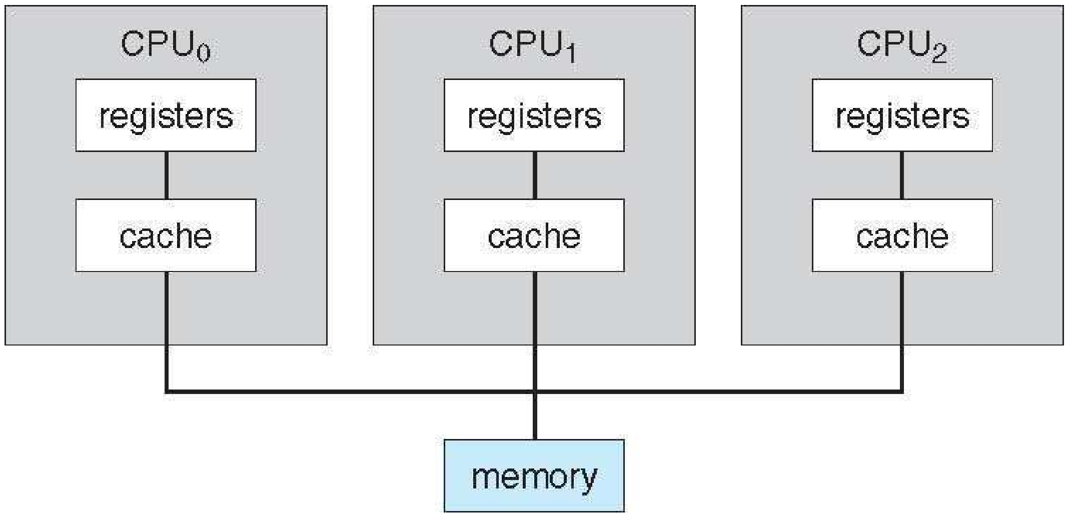


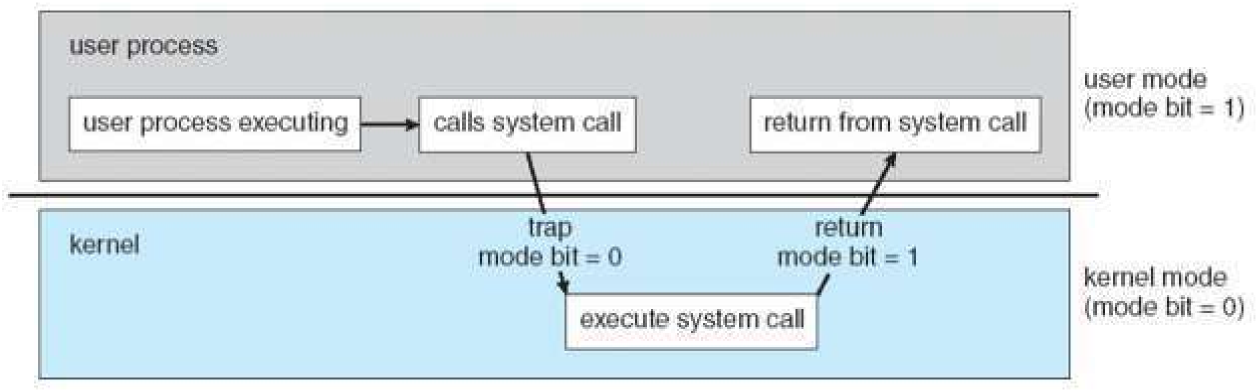
Fig: Symmetric multiprocessing architecture

Operating System Structure

- **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute

  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**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
- **Operating System Operation**
- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **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
- Timer to prevent infinite loop / process hogging resources

- Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time

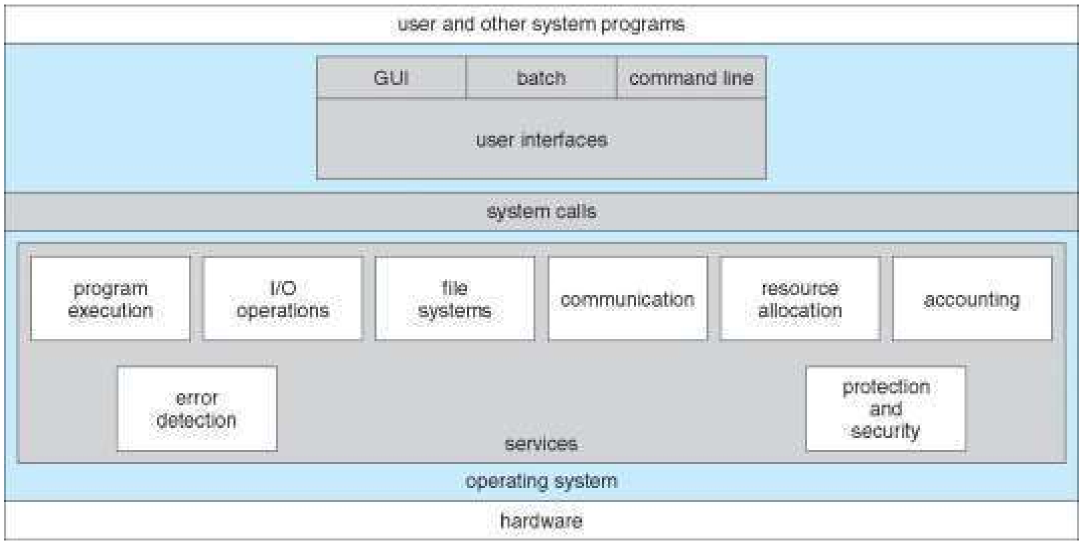


OS Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- One set of operating-system services provides functions that are helpful to the user (Cont):
  - Communications – Processes may exchange information, on the same computer or between computers over a network

        - Communications may be via shared memory or through message passing (packets moved by the OS)
    - Error detection – OS needs to be constantly aware of possible errors
        - May occur in the CPU and memory hardware, in I/O devices, in user program
        - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
        - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- 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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
    - **Accounting -** To keep track of which users use how much and what kinds of computer resources
    - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
        - **Protection** involves ensuring that all access to system resources is controlled
        - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
        - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

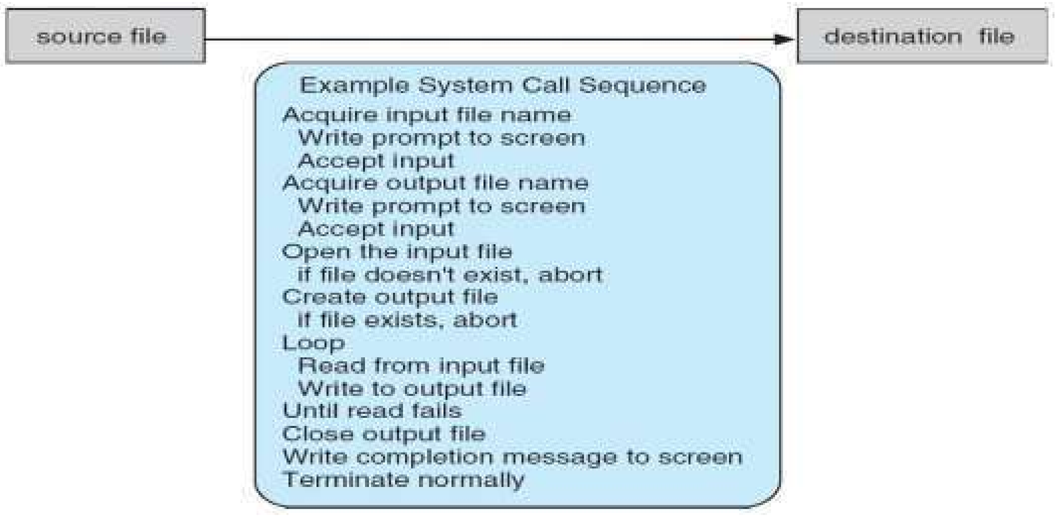


System Call

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program 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)

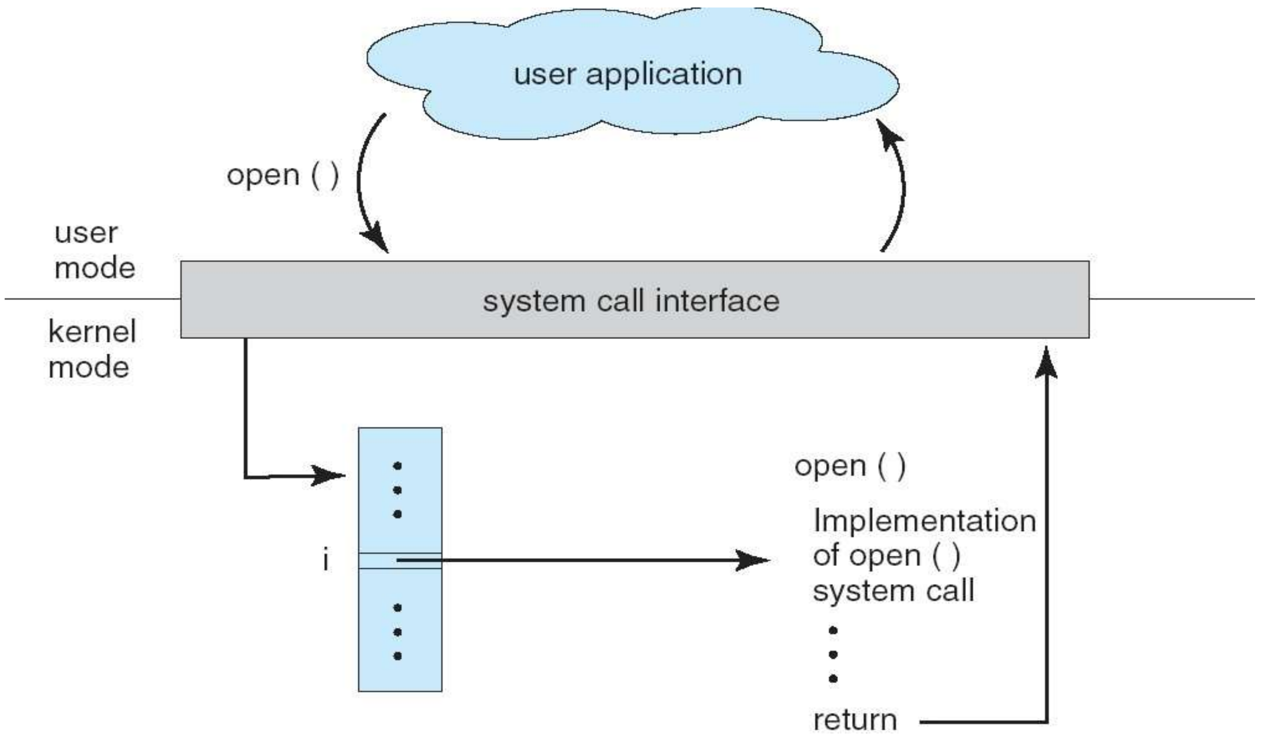
  Example

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



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

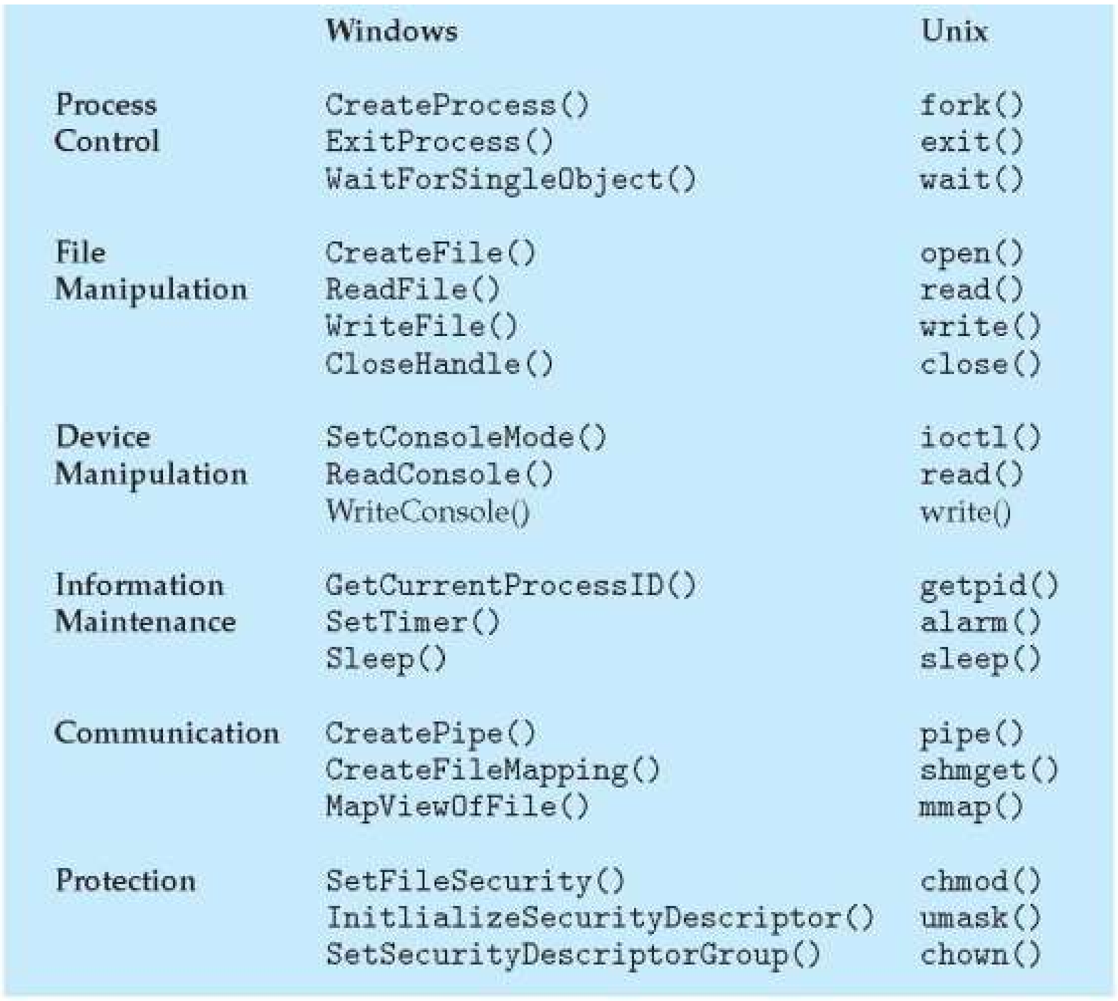
- Typically, a number associated with each system call
    - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about 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)



Types of system call

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

- Communications
- Protection



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

OS Structure

n MS-DOS – written to provide the most functionality in the least space

l Not divided into modules

l Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

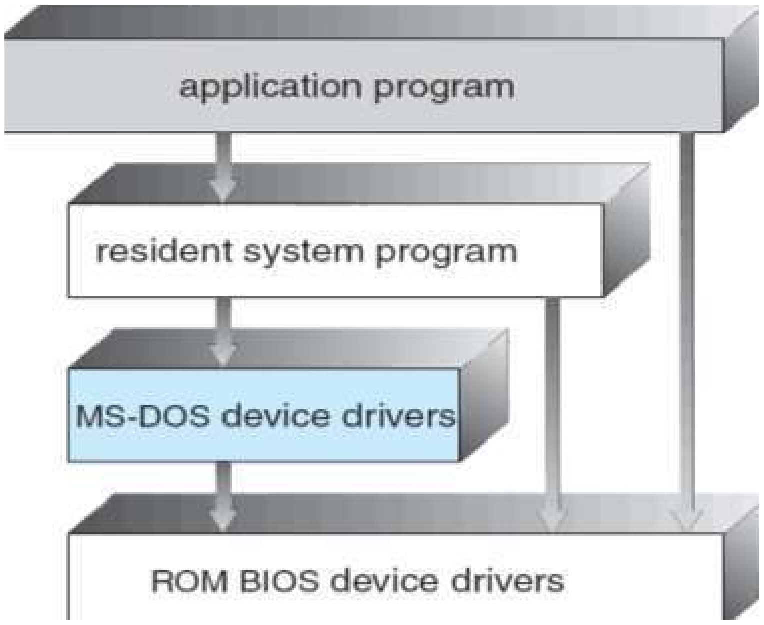


Fig: MS Dos structure

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

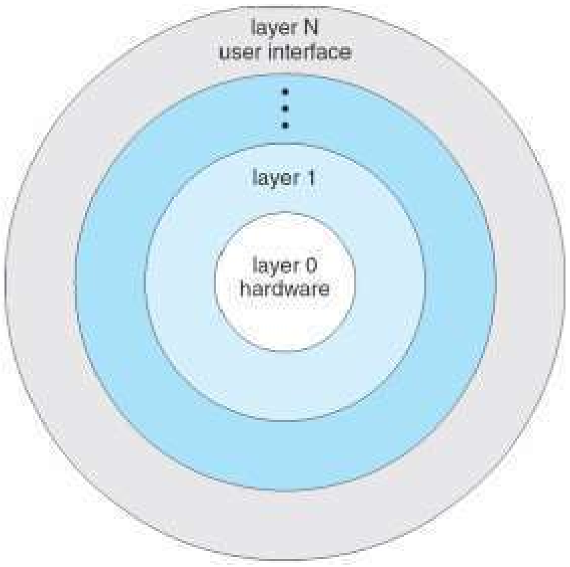


Fig: Layered System

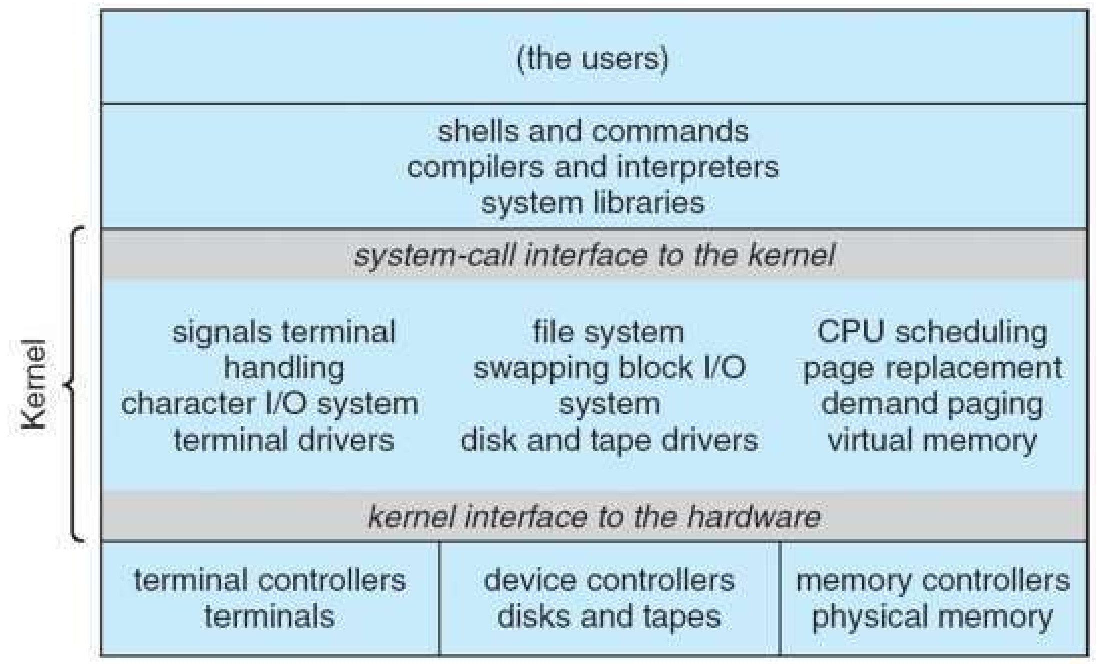


Fig: UNIX system structure

Micro Kernel Sructure

- Moves as much from the kernel into "*user*" space
- 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

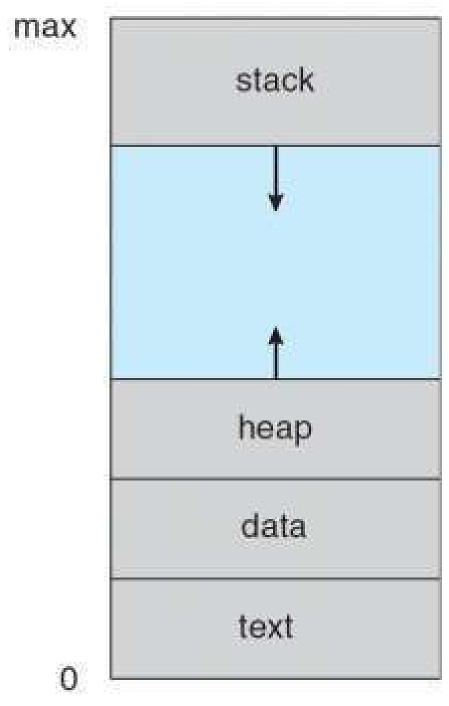
Virtual Machne

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware

- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

Process Management

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section



Process State

As a process executes, it changes *state*

- new: The process is being created
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution

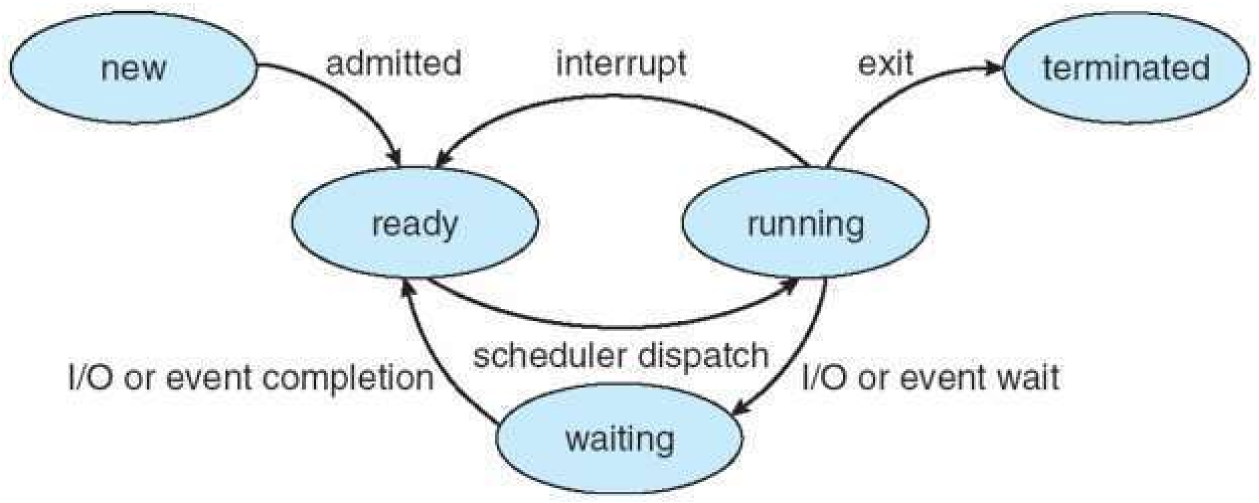


Fig: Process Transition Diagram

PCB: Process Control Block

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

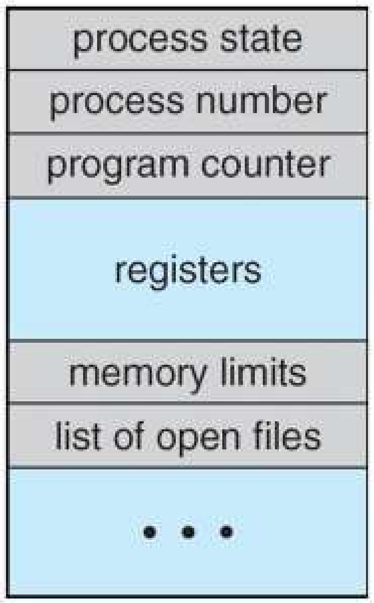


Fig: PCB

Context Switching

- 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
- Time dependent on hardware support

Process Scheduling Queues

- 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

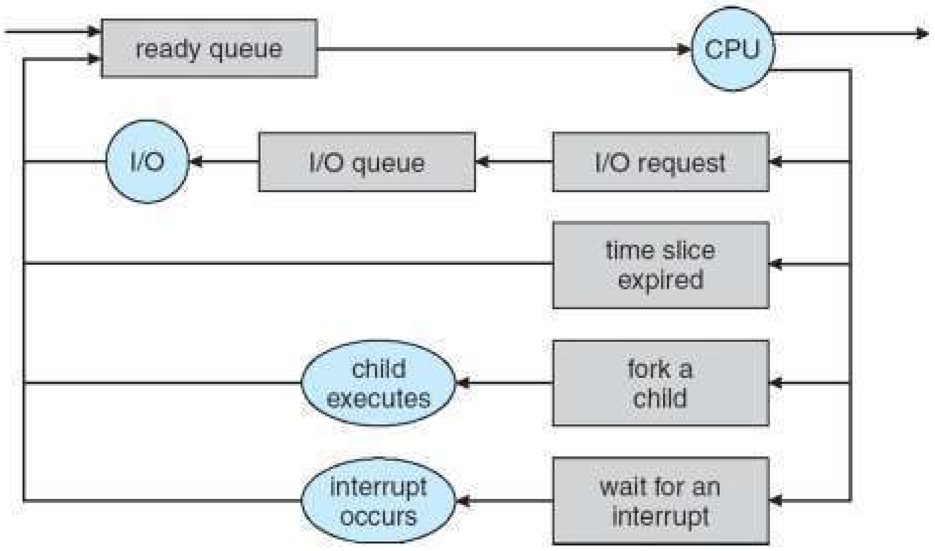
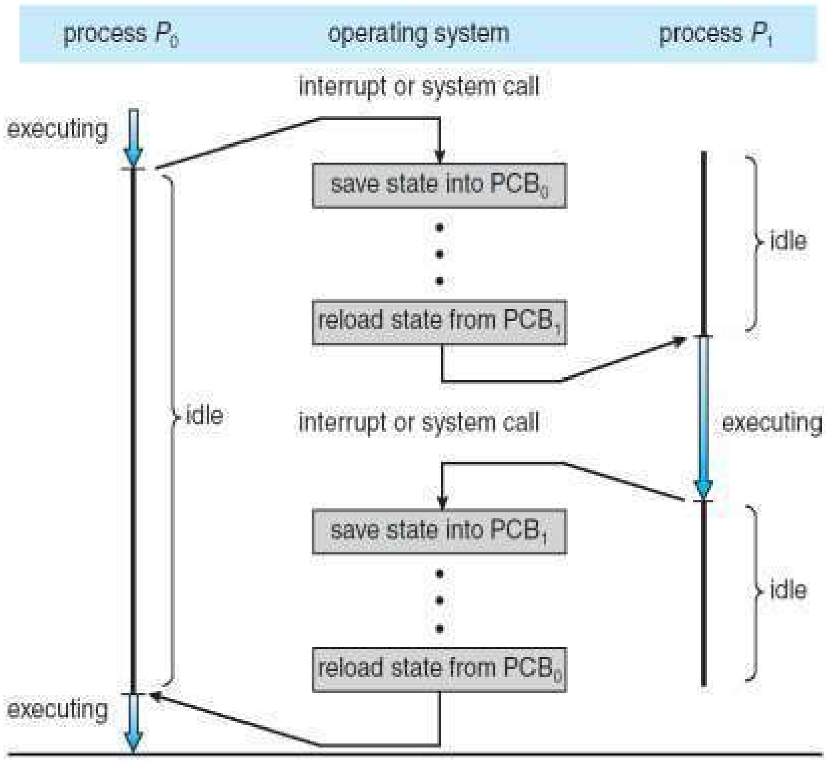


Fig: Process Scheduling



Schedulers

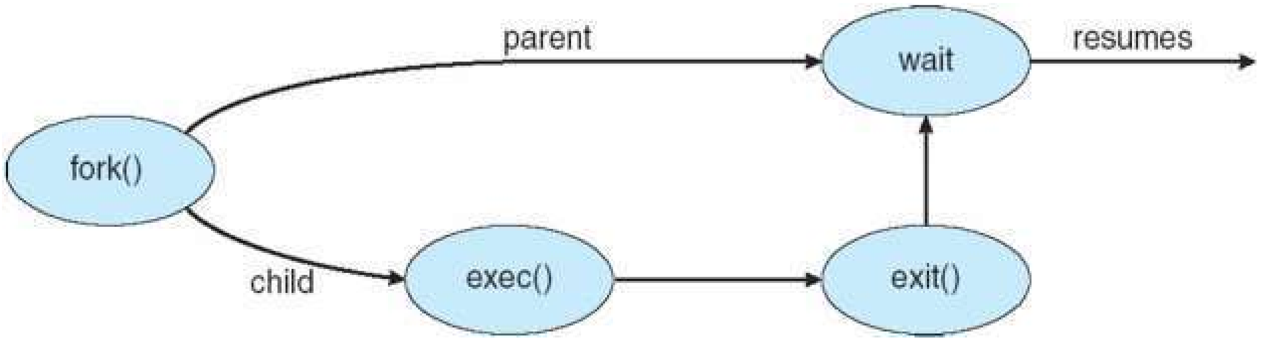
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

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

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources
- Execution
    - Parent and children execute concurrently
    - Parent waits until children terminate
- Address space
    - Child duplicate of parent
    - Child has a program loaded into it
- UNIX examples
    - **fork** system call creates new process

- **exec** system call used after a **fork** to replace the process' memory space with a new program



Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
    - Output data from child to parent (via wait)
    - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting
        - Some operating system do not allow child to continue if its parent terminates
            - All children terminated - cascading termination

Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
    - Information sharing
    - Computation speedup
    - Modularity

- Convenience

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

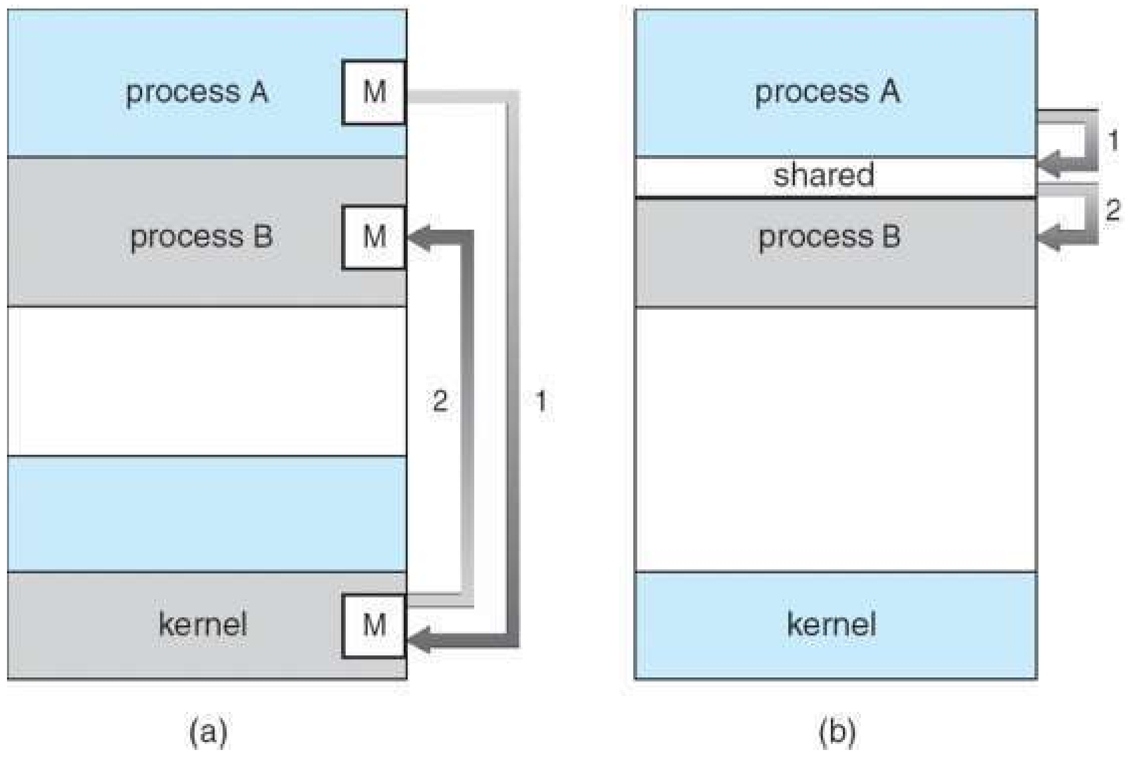


Fig:a- Message Passing, b- Shared Memory

Cooperating Process

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity

- Convenience

Producer Consumer Problem

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

```
        while (true) {
 /* Produce an item */

     while (((in = (in + 1) % BUFFER SIZE count)  == out)

           ;   /* do nothing -- no free buffers */

          buffer[in] = item;

          in = (in + 1) % BUFFER SIZE;

   }
```

Fig: Producer Process

```
                 while (true) {

      while (in == out)

           ; // do nothing -- nothing to consume

             // remove an item from the buffer

             item = buffer[out];

             out = (out + 1) % BUFFER SIZE;
```

IPC-Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:

  - send(*message*) – message size fixed or variable
  - receive(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Direct Communication

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

Indirect Communication

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

- Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - 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
- 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.

Synchronisation

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full

3. Unbounded capacity – infinite length
Sender never waits

Thread

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

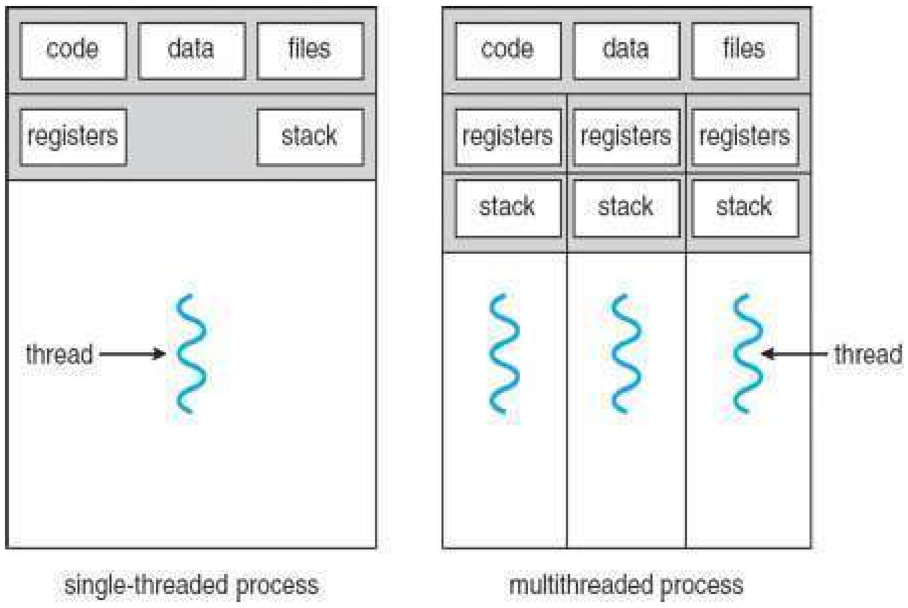


Fig: Single threaded vs multithreaded process

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

User Threads

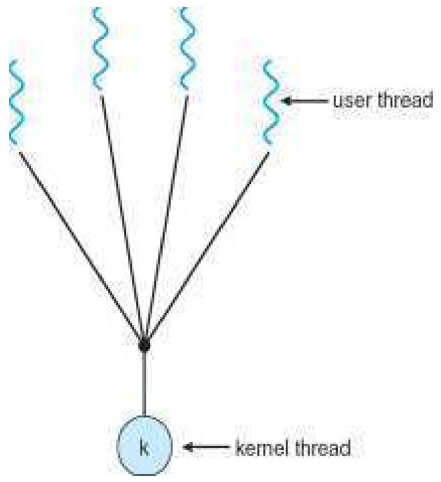
- Thread management done by user-level threads library
- Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Kernel Thread

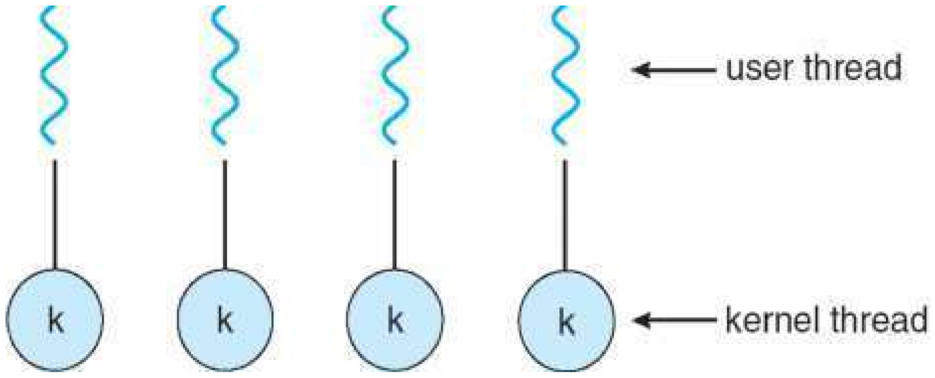
- Supported by the Kernel
- Examples
    - Windows XP/2000
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X

Multithreading Models

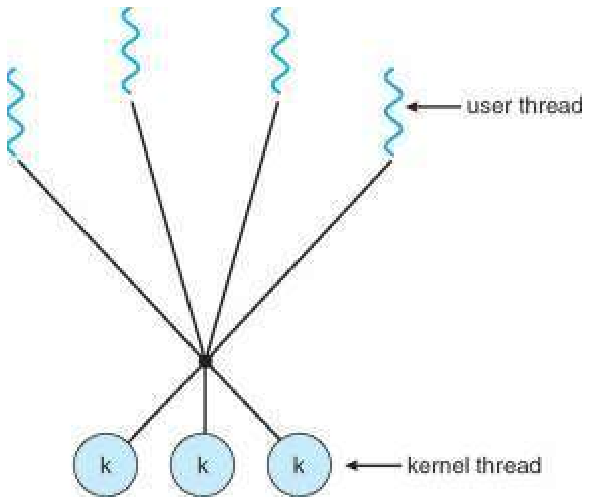
- Many-to-One
    - Many user-level threads mapped to single kernel thread
    - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



- One-to-One
  - Each user-level thread maps to kernel thread
  - Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



- Many-to-Many
  - Allows many user level threads to be mapped to many kernel threads
  - Allows the operating system to create a sufficient number of kernel threads
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package



Thread Library

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

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- 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)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
    - Extending Thread class
    - Implementing the Runnable interface

Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation of target thread
    - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Thread Cancellation

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

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
    - Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Difference between Process and Thread

Process	**Thread**
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Process Scheduling

- 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** distribution

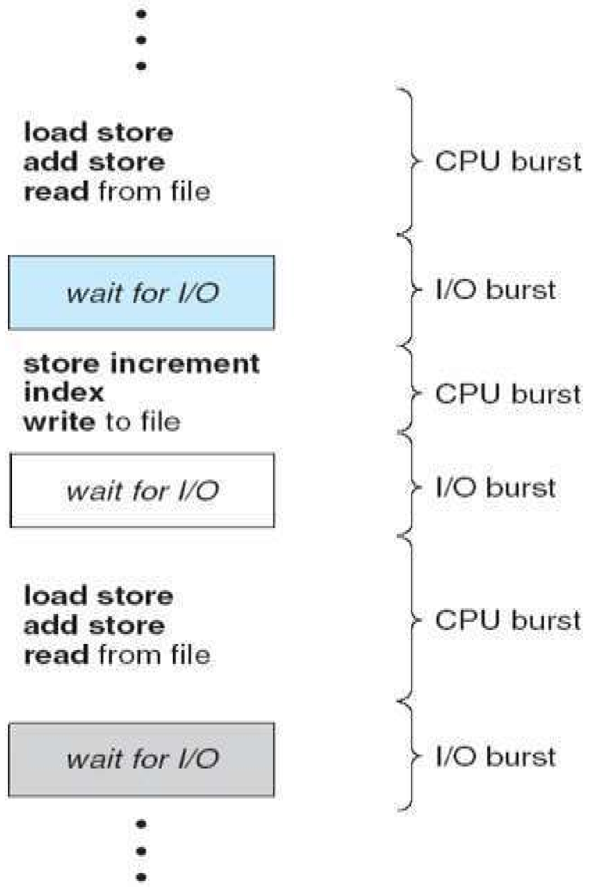


Fig: CPU burst and I/O burst

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

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

CPU Scheduling Criteria

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

CPU Scheduling Algorithms

A. First Come First Serve Scheduling

- Schedule the task first which arrives first
- Non preemptive In nature

B. Shortest Job First 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

Priority Scheduling

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging –** as time progresses increase the priority of the process

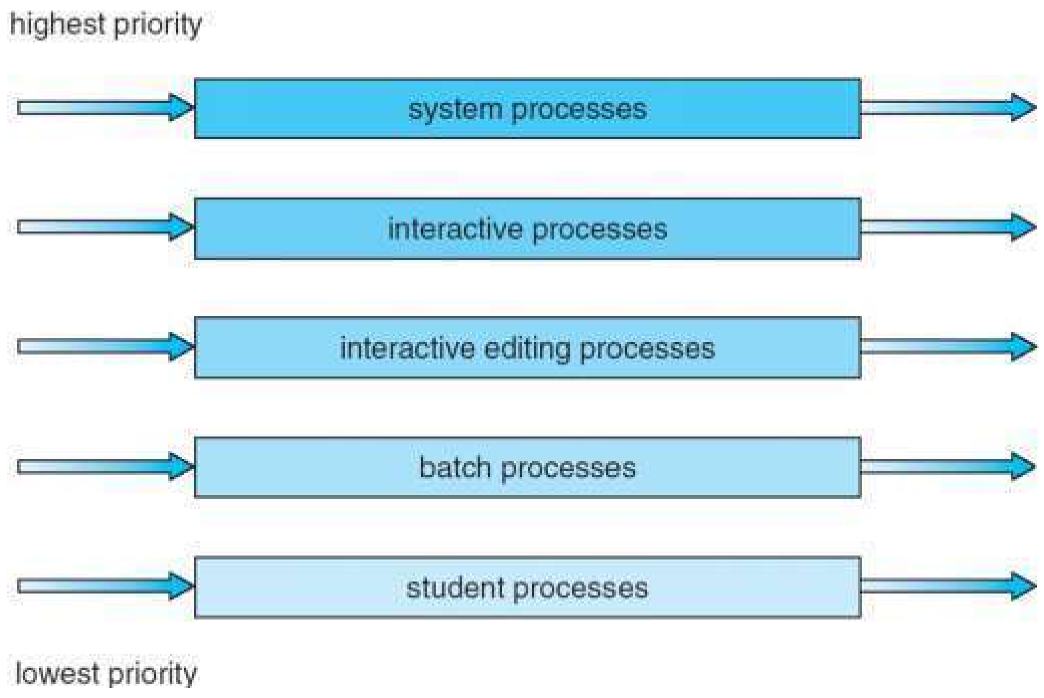
Round Robin Scheduling

- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS

- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- 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

MODULE-II

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Pseudocode for Producer Process	Pseudocode for Consumer Process

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

data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

- count++ could be implemented as

```
register1 = count
```

```
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

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

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

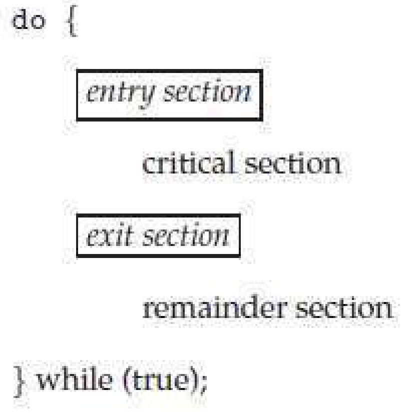
```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

Critical Section Problem

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

- Entry Section Code requesting entry into the critical section.
- Critical Section Code in which only one process can execute at any one time.
- Exit Section The end of the critical section, releasing or allowing others in.
- Remainder Section Rest of the code AFTER the critical se



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

General structure of a typical process P_i.

-

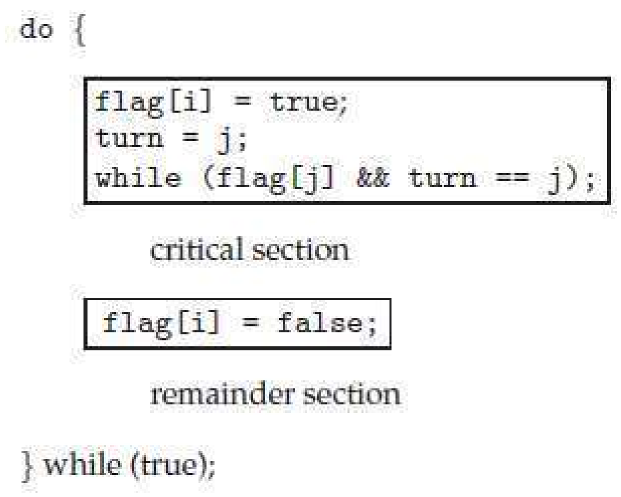
- Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this
- request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

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

Peterson’s Solution

- Two process solution
- Assume that the LOAD and STORE 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!



```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Hardware Synchronization

- Many systems provide hardware support for critical section code
- 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-interruptable

  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical Section Problem using Lock

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

TestAndndSet Instruction

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

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

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