# Module-2

# Process Management

# Contents :

- Process Concepts
- Operations on Processes
- Inter Process Communication- IPC
- Introduction to threads
- Multithreading Models
- Threading Issues
- Basic Concepts of Process(CPU) Scheduling
- Scheduling Criterions

.

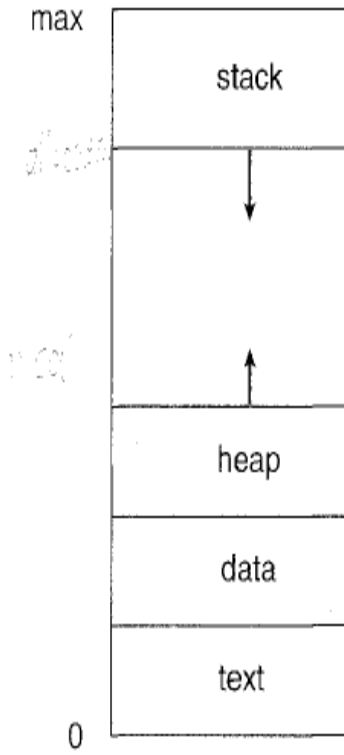
- Scheduling Algorithms:

1. FCFS
2. SJF
3. RR
4. Priority
5. Multilevel Queue
6. Multilevel Feedback Queue

# Process Concepts

- In the early computer systems, only one program is executing at a time. So, that one program had complete control over the system & its resources.
- But, today's computer systems allow multiple programs to execute at a time (parallely).
- So, some sort of mechanism is needed to create, execute & manage those programs which are running parallely.
- This is will be done by process manager (part of OS)
- "**A running program is called as process**" OR "**A program in execution is called as process**"

# Process Concepts




An operating system executes a variety of programs:

- In **Batch systems,** we use the term **jobs**
- In **Time-shared systems,** we use the term **tasks**
- In many respects, all these activities are similar & hence we call them as **processes**

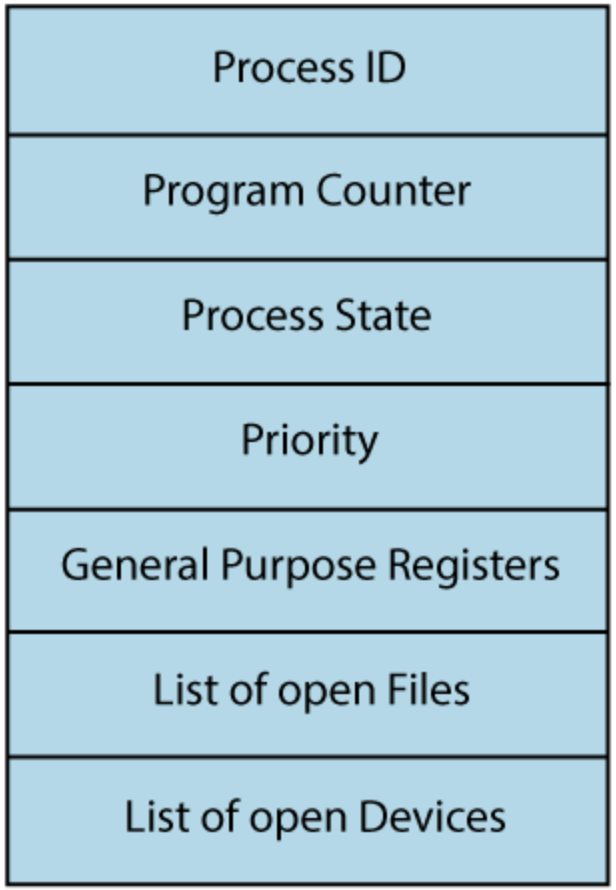
- structure of process in memory is as shown in the fig,
- **Text section** includes compiled program code which is read from secondary storage when program is launched
- Text section also includes information like a) which part of the program is currently executing
  b) which is next statement to be executed(i.e., content of PC)
  c) contents of some other processor's registers
- Each & every process has its own stack called **process stack** consists of temporary data such as function parameters, return address/values & local variables.
- **Data Section** contains global variables & static variables which are allocated & initialized before executing the progra
- **Heap** is used for dynamic memory allocation & is managed via calls to new, delete, malloc, free, etc.

# Process Concepts----Program vs Process:

| PROGRAM | PROCESS |
|---|---|
| 1) Set of instructions to perform a specific task | 1) Program in execution |
| 2) Passive entity | 2) Active entity |
| 3) One program can invoke multiple processes | 3) Cannot invoke any other processes |
| 4) Program is stored generally in the form of file (executable) | 4) Program become process only when an executable file is loaded into main memory |
| 5) Program doesn't require any resources other than storage space | 5) Process holds resources such as CPU, memory address, disk, I/O, etc |
| 6) Program has longer life span. It is stored on disc forever | 6) It has limited life span. It is created when execution starts & terminated as execution is finished |

## Attributes /Characteristics of a Process:

1) **Process Id**
2) **Process State**
3) **CPU registers**
4) **I/O status information**
5) **CPU scheduling information**




Process Attributes

# Process Control Block (PCB)

Each process is represented in OS by **PCB** also called as **Task control block (TCB)**
Every process has its own **program control block(PCB**), i.e.; each process will have a unique **PCB**
A PCB has the following information about a process:

- ***Process number:*** *unique id of a process*
- **Process state –** state may be ready, new, running, waiting or terminated
- **Program counter -** location of the next instruction to be executed
- **CPU registers –** number of registers & type of registers vary from system to system, depending upon the computer architecture. These includes accumulators, index registers, stack pointers, etc
- **CPU scheduling information-** includes process priorities, pointers to scheduling queues & some other scheduling parameters
- **Memory-management information -** includes amount of time consumed by CPU in user & kernel mode, account numbers, jobs/process numbers, etc
- **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

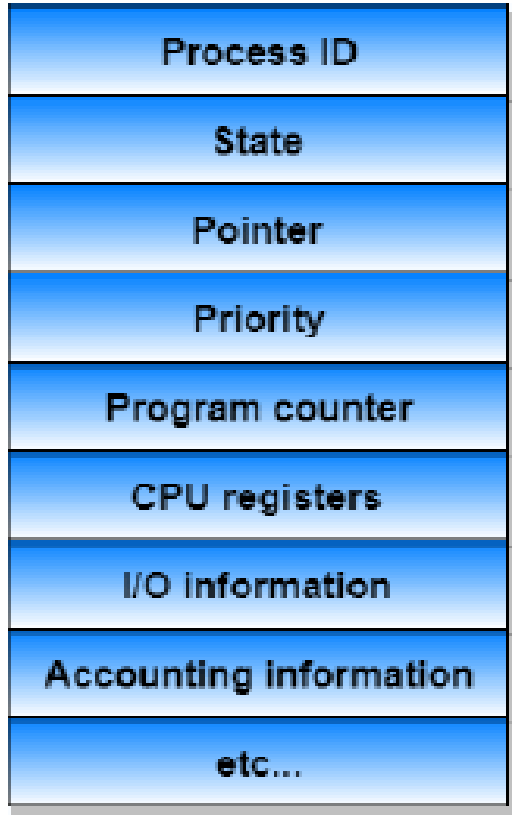
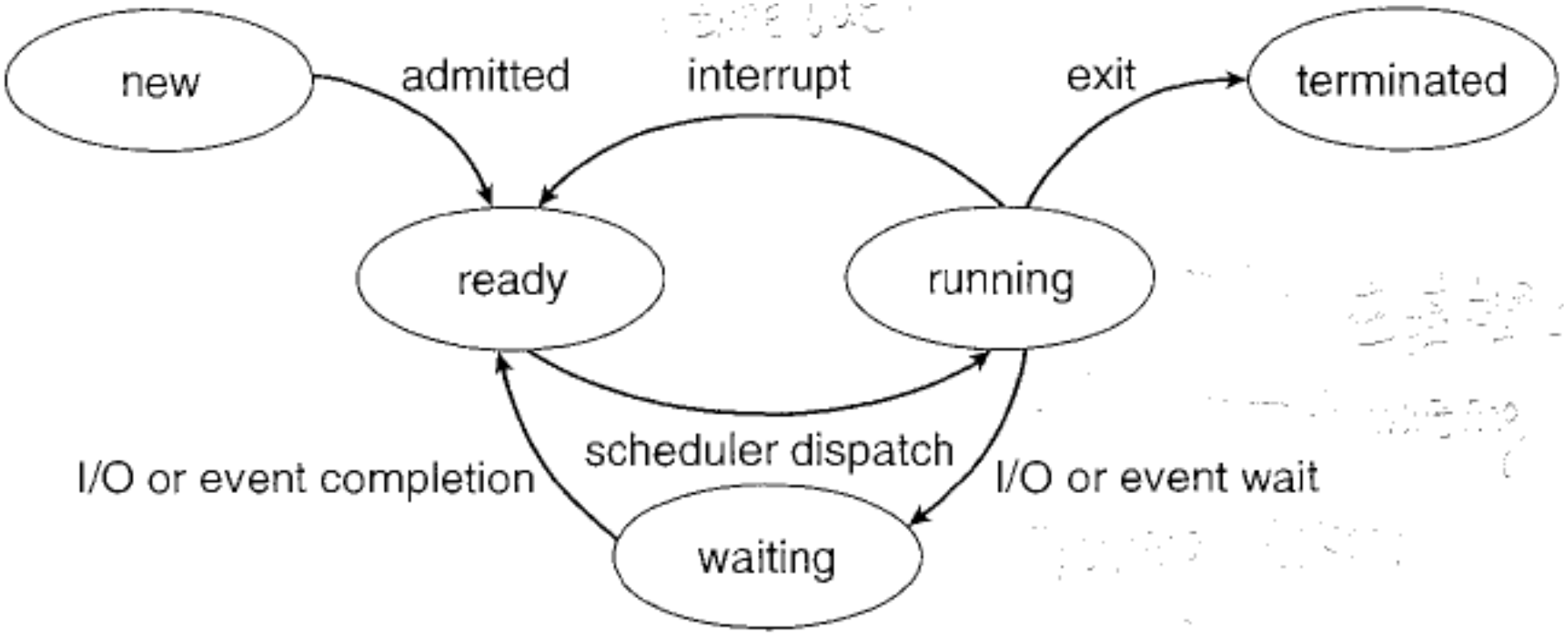



Fig.: Process Control B

# Process State Diagram:

- State of a process is defined as the current activity/status of the process.
- When a process starts executing, it keeps on changing its state from one state to another.



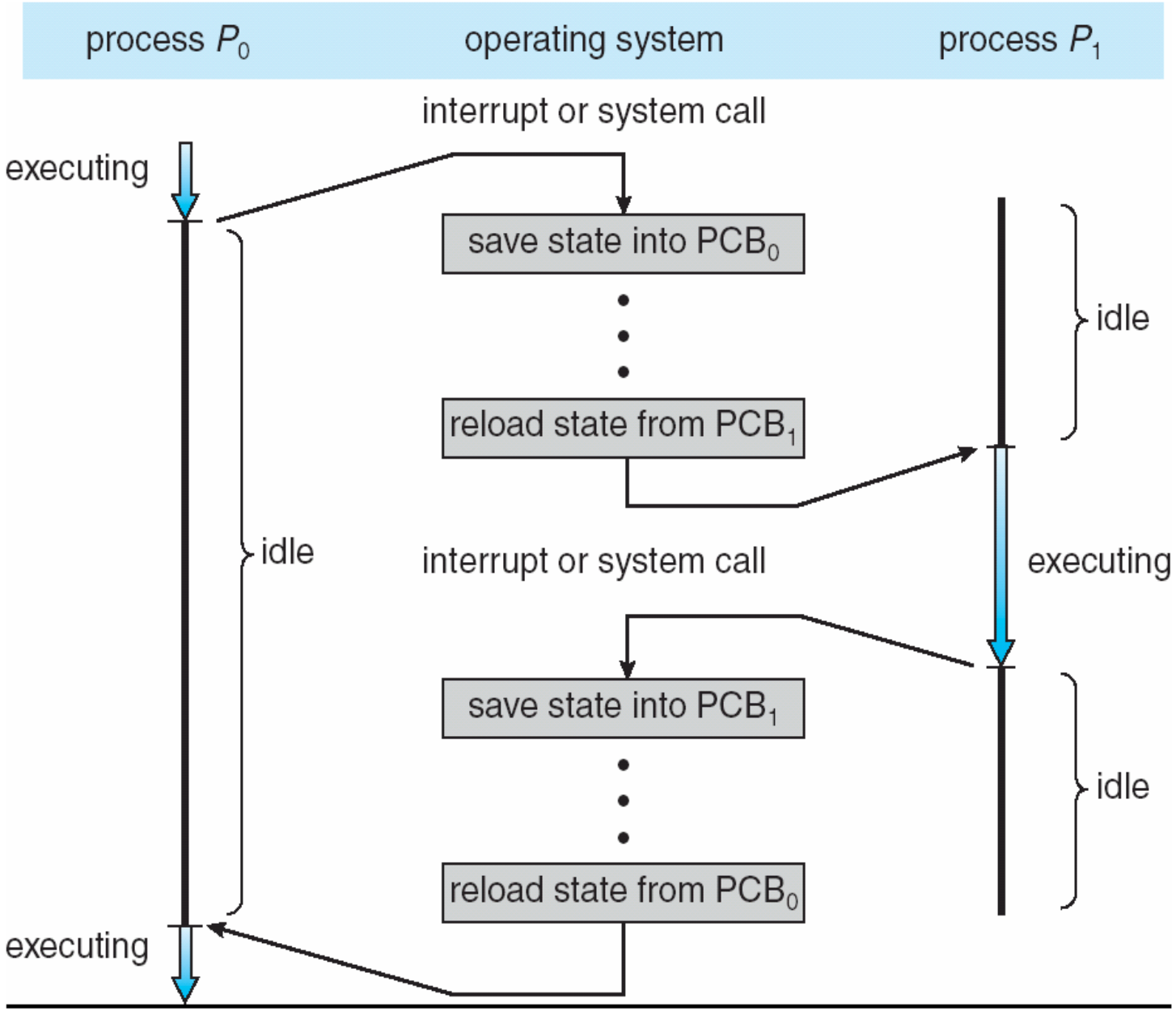
# Process States:

The various states of a process are:

1. **New:** when process is created, it will be in new state.
2. **Ready:** process has all its resources & set to execute, but CPU time is not allocated. That means, process is ready to execute or process is waiting for CPU allocation
3. **Running:** when the instructions of the process is getting executed, process will be in running state
4. **Waiting(or Block):** sometimes, a process has to wait for some resources to become available or for some event to occur. In such conditions, we can say process is in waiting state.
5. **Terminated:** The process completed its execution successfully.

Name of these 5 states may vary from one OS to another, but the concept remains same
It is important to realize that, only one process can be in running state at any time, but, many processes may be in ready & waiting state.

# What happens when the CPU Switch From a process to another process?



# Context Switch

- **Context switch** occurs when the time slice for one process has expired & a new process is to be loaded from the ready queue.
- Switching the CPU to another process requires **state-save** of the current process & **a state-restore** of different process. This task is known as **context switch**
- Saving & restoring states involves saving & restoring of all of the registers & PC's as well as the PCB
- Context switching happens very frequently & the overhead of doung the switching is just lost CPU time, so context switches(state-save & state restore) need to be as fast as possible
- Special hardware were used to speed up the context switching, such as single machine instructions for saving or restoring all registers at once.

## When does a context switch happen?

1. When a high-priority process comes to ready state (i.e. With a higher priority than the running process)
2. An Interrupt occurs
3. User and kernel mode switch (It is not necessary though)
4. Pre-emptive CPU scheduling used.

**Context switching allows multiple processes to share a single central processing unit(CPU).**

# Operations on Processes

- Process can be created, executed & deleted dynamically.
- A system must provide mechanisms for: process creation & process termination
- **Process creation**
  - A process may in turn can create other processes through appropriate system calls like fork or spawn or CreateProcess
  - Process which creates another process is called **“parent process”** & newly created process is called as **“child process”**
  - Each of these newly created processes may in turn create other processes, forming a **tree of processes**
  - Each & every process is having an unique integer number called PID.
  - Most OS’s identify the processes through PID only
  - Every process needs certain resources such as CPU time, memory, files, I/O devices in order to complete its task

# Operations on Processes----Process creation **contd**

- When a process creates sub-process, that sub-process may be able to obtain its resources directly from the OS or from its parent process
- So, parent process must partition its resources among its children or it should share the resources among its child's.
- When a parent process creates child process, it may pass some initialization data (i.e., input) to the child process.
- When a parent process creates a new process, 2 possibilities exist in terms execution

  *) parent process continues to execute concurrently with its children

  *) parent process waits until some or all of its children have terminated
- There are 2 possibilities in terms of address space of new process

*) The child process is a duplicate of the parent process (it has the same program & data as the parent).

*) The child process has a new program loaded into it.

Let us understand these concepts by considering UNIX OS as an example.

# Process Creation

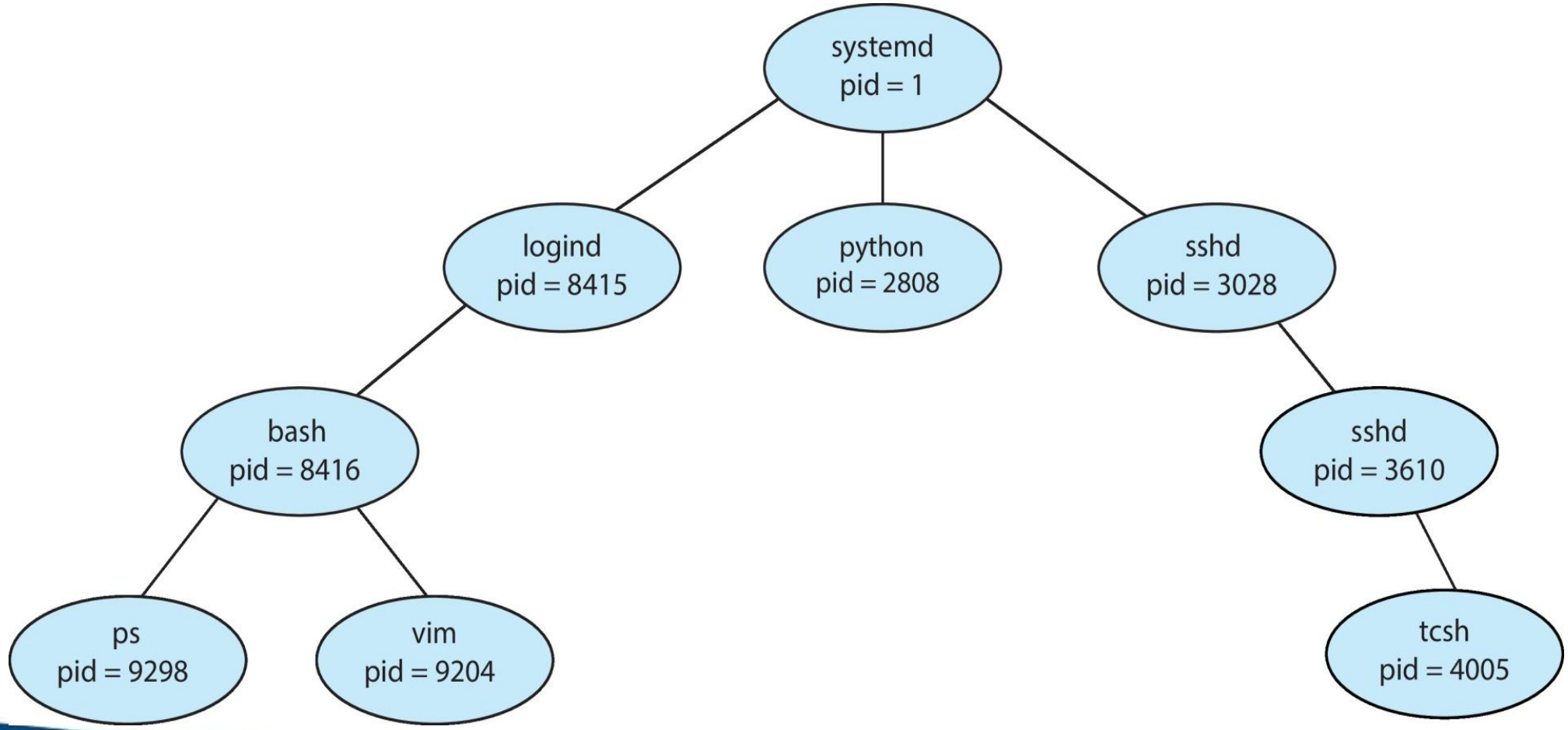



**Fig.: A Tree of Processes in Linux**

# Creating a separate process using the UNIX fork() system call

```
#include <sysltypes.h>     #include <stdio.h>    #include <unistd.h>
int main()
{
        pid_t pid;
        pid =fork();                                        /* fork a child process */
        if (pid < 0) {                                      /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {                                /* child process */
                execlp("lbinlls","ls",NULL);
                }
                else {                                      /* parent process */
                wait (NULL) ;                               /* parent will wait for the child to complete */
                printf("Child Complete");
                }
         return 0;
}
```

# Process Termination

- After executing the last statement, process request for terminating by invoking **exit()** system call.
- Terminating process should inform to its parent process about its termination via **wait()** system call
- When a process is getting terminating, it returns its status value(generally integer) to its parent process through **wait()**
- All the resources of the process, including physical & virtual memory, open files & I/O buffers are deallocated by the OS
- Process may also be terminated by the system from various reasons
  - inability of the system to deliver necessary system resources
  - parent process may kill it's children if the task assigned to them is no longer needed.
  - If the parent exits, system may or may not allow the child to may Continue without a parent. (on UNIX, orphaned processes are generally inherited by init, which then proceeds to kill them. In UNIX, nohup Command allows a child to continue after its parent has exited).
- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred as **cascaded termination,** which is normally initiated by the O.S.

# Inter-Process Communication(IPC)

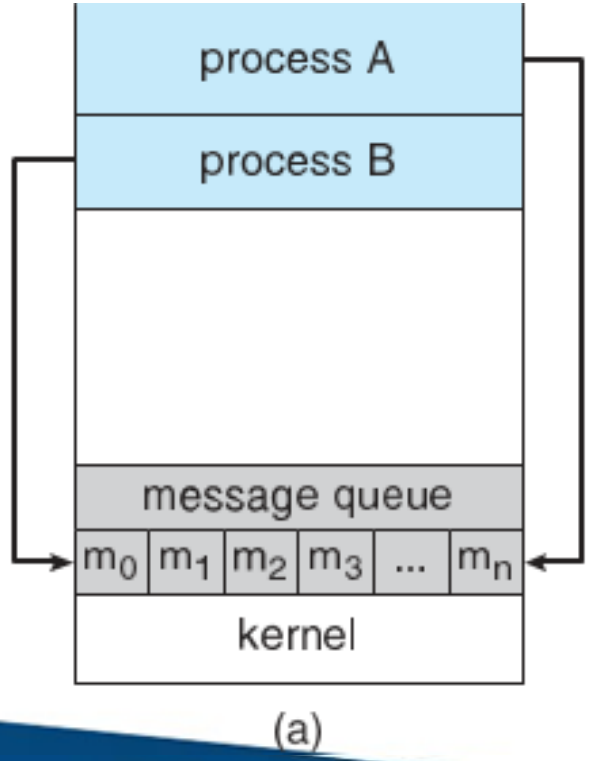
- Processes executing concurrently in the OS may be either ***independent*** or ***cooperating process***
- **Independent processes** are the processes which does not share any data with any other processes. These independent processes doesn't affects other processes or does not get affected by other processes.
- Cooperating processes can affect or be affected by other processes, e.g., while sharing data or resource/s.
- There are several reasons for allowing processes to co-operate:
    - **Information sharing**:- several users may be interested in the same piece of information(shared file, pipeline)
    - **Computation speed-up:-** solution to a problem can be solved faster if the problem can be broken down into sub-problems & solved parallely(possible only when multiple processors)
    - **Modularity:-**it is always efficient to build the system in a modular fashion by dividing system functions into separate processes or threads
    - **Convenience:-** even single user may work on multiple tasks at the same time(such as editing, printing & computing)

# Inter-Process Communication(IPC)

- When co-operating processes are sharing some data, OS should provide some sort of communication between these co-operating processes -------IPC
- So, co-operating processes requires IPC, to exchange data & information.
- There are 2 fundamental models of IPC
  - 1) Shared Memory
  - 2) Message Passing

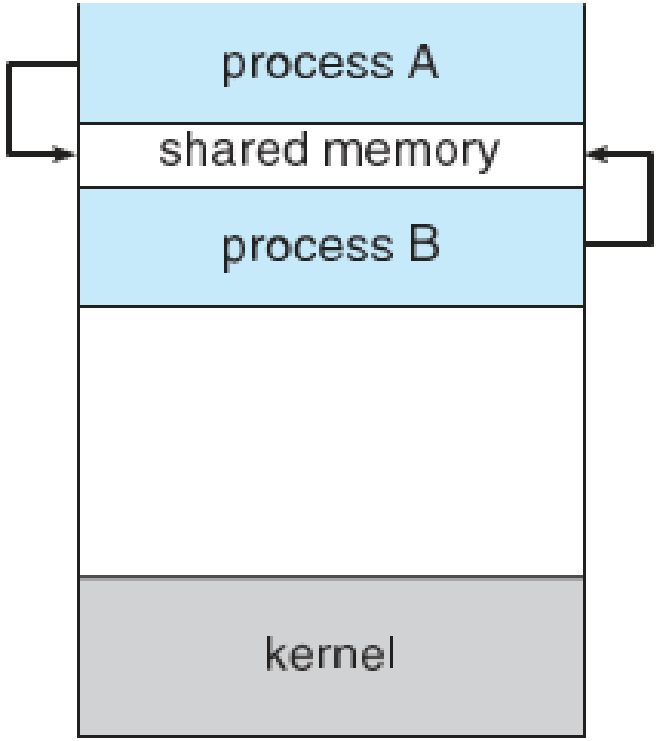
# Communication Models

(a) Message passing




(a)

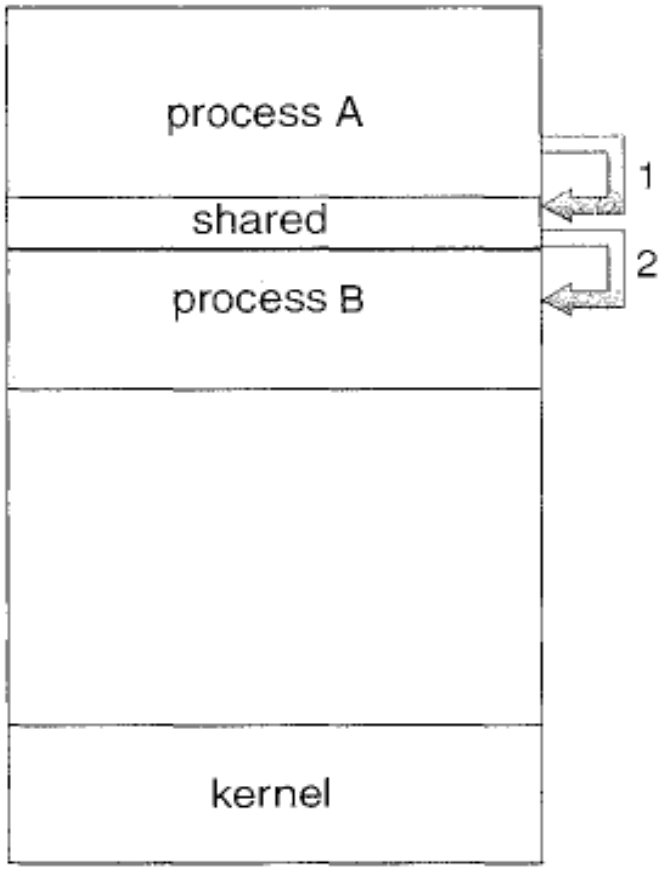
(b) shared memory




(b)

# Shared Memory Systems

- in this model, region of memory that is to be shared by co-operating processes is established. Processes can then exchange information by reading & writing data to the shared region
- Shared memory is faster once it is established because no system calls are required & access occurs at normal memory speeds. However, it is more complicated to establish initially.
- Shared memory is generally preferable when large amount of data to be shared quickly between the processes within a system.
- But, it doesn't work between computers.
- This shared memory region normally in the address space of the process which establishes the shared memory region.
- We know that, OS prevents one process from accessing another process's memory, processes that are using this shared memory should remove this restriction.
- Once the processes remove this restriction, then they can exchange data by reading & writing in the shared areas.
- Processes should also ensure that they are not writing into this shared region at the same time,



# Message Passing Systems

- In order to use shared memory concept, application program's. has to write separate code for providing the facility of accessing & manipulating the shared memory.
- In **message passing** model, communication takes place by means of messages exchanged between the cooperating processes.
- message passing is simpler to establish (set-up) & works will between the computers & as well as within the computer system.
- Generally, it is preferable when the amount of data transfer is small & when multiple computers are involved.
- Message passing requires system calls for every message transfer & hence it is slower.
- Massage passing provides a mechanism to allow processes to communicate & to synchronize their actions without sharing the same address space to is particularly useful when communication to be happen between the systems in a network. Best example is chating.
- message passing facility provides 2 operations
  1) Send (message)
  2) Recieve (message)
- Messages sent by a process may be fixed or variable size. If **only fixed size** messages can be sent, system- level *implementation is straightforward*, but it makes task of *programming more difficult*.

# Message Passing Systems

- **Variable sized** messages requires *more complex implementation* (at the system level) but *programming task becomes simpler*
- If processes P & Q want to Communicate, they must rend messages to & receive messages from each other.
- In order to do this, a communication channel/link has to be established.
- In order to setup the communication link, there are several methods
  - Direct or indirect communication
  - Synchronous or Asynchronous Communication
  - Automatic or explicit buffering
- **Naming:** Processes that wants to Communicate must have a way to refer to each other. They can use either direct or indirect Communication.
- Under **direct Communication**, each process should know the name of other process with which it is communicating.
  In this scheme, **Send()** & **Receive()** are defined as send (P, message)------to send a message to P
  receive(Q, message) ---receive a message from Q
  In this scheme, there should be dedicated link between sender-receiver pair.
- For **symmetric Communication**, both sender & receiver should know the name of each other.
- For asymmetric Communication, only render should know the name reciever but for receiver no need to know render name,Send (P, merrage) Send a to PRecents (id, merrage) ""id" is net to the name of process with which Communication has taken place Recieve (Q, Merage)

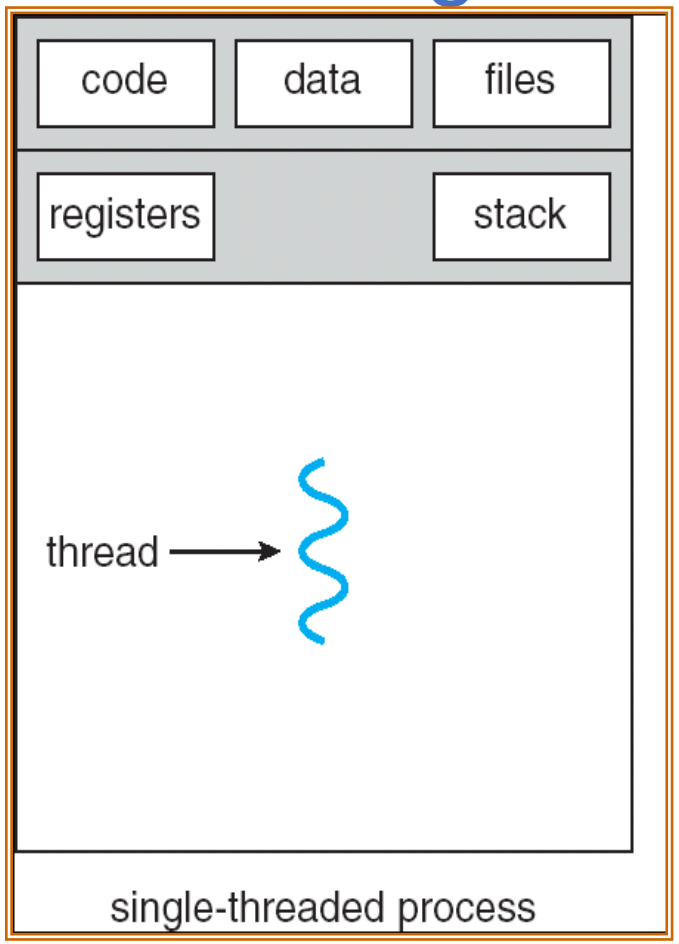
# Message Passing Systems

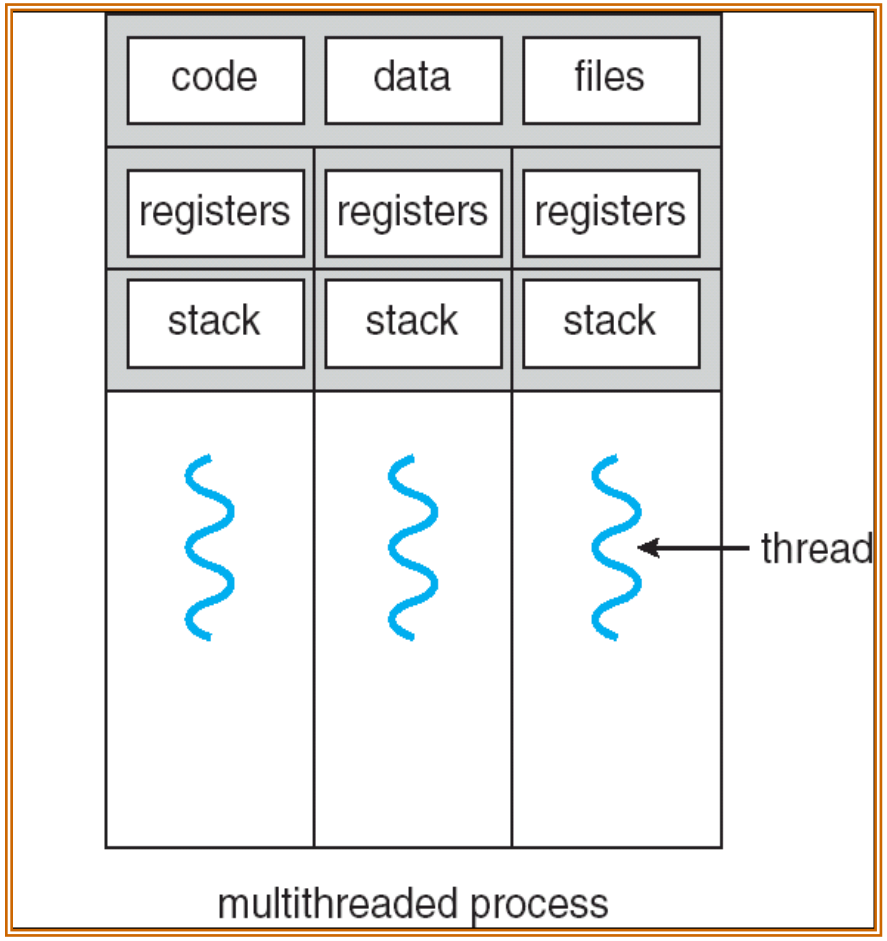
- For **asymmetric Communication**, only sender should know the name of receiver but for receiver no need to know sender name, Send (P, message)-------- Send a to P
  Receive(id, message)------ "id" is set to the name of process with which Communication has taken place.
- With **indirect Communication**, messages are sent to mailboxes or ports.
- Each mailbox has a unique identification. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which, messages can be removed.
- In this scheme, a process can communicate with some other process via number of different mailboxes.
  send(A, message) --------send a message to mailbox A
  receive(A, message) ------receive a message from mailbox A.
- Here, multiple processes can share the same mailbox & ports. OS must provide system calls to create & delete mailboxes & to send & receive messages to / from mailboxes.
- A mailbox may be owned either by a process or by OS.
- If the mailbox is owned by a process, then we need to distinguish between the owner & the user.
- When a process that owns a mailbox terminates, mailboxes disappears

# Threads

- Today's most of the OS's provides some features for process's to contain multiple threads of control
- "A **Thread** is a basic unit of CPU utilization, consisting of program counter (PC), thread ID, set of registers & stack"
- Traditional process (heavyweight) has a single thread of control. That means, generally each & every process should have one thread of control.
- Every process has its own **code segment**, **data segment** & **some OS resources** like open files & signals
- Single thread can perform only one specific task.
- Multiple threads can perform multiple tasks at a time
- If a process has multiple threads, it can perform multiple tasks
- A thread shares its code & data segments, OS resources with other threads belonging to the same process

# Single and Multithreaded Processes



# Four Primary Benefits of Threading

- **Responsiveness**
  - By making an Interactive application as multithreading, we can allow it to continue running even if part of it is blocked or performing lengthy computations, thereby increasing responsiveness to the user
  - For example, multithreaded web browser could allow user interaction in one thread while an image/document was being uploaded/downloaded in another thread
- **Resource Sharing**
  - Processes may only share resources through shared memory or message passing techniques.
  - However, Threads share memory & the resources of the process to which they belongs by default.
  - Benefit of sharing code & data is that, it allows an application to have several different threads within same address space
- **Economy/Efficiency**
  - Resource allocation is costly.
  - Sharing reduces overhead and reduces the cost of context switching during CPU scheduling
- **Utilization of Multiprocessor Architectures**
  - Can leverage multi-CPU architectures for genuine parallelism

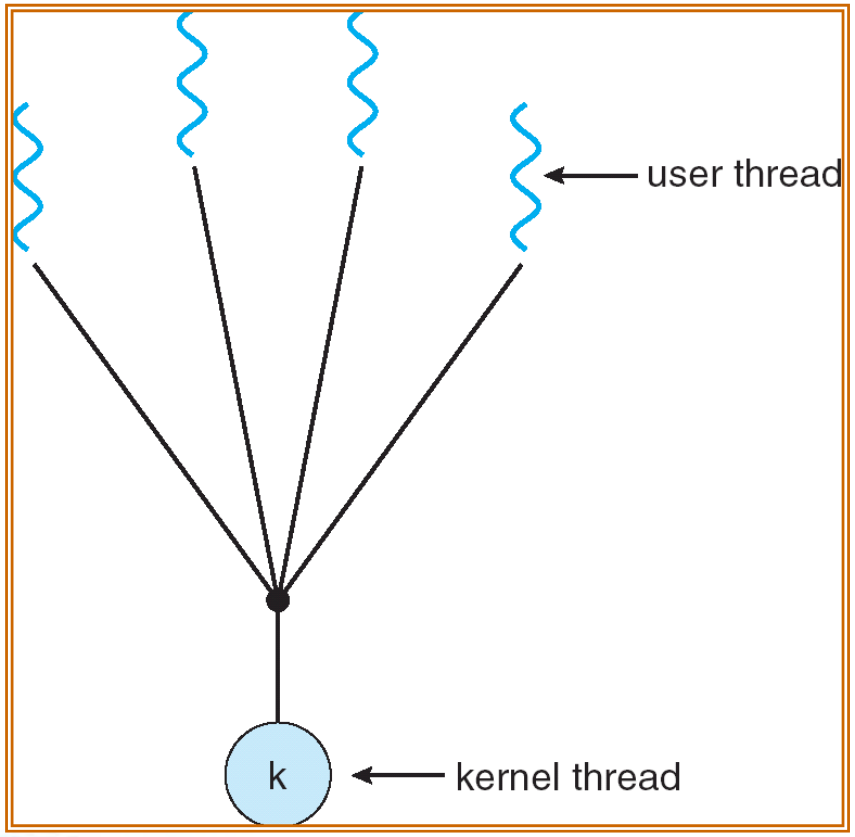
# Four Primary Benefits of Threading

- **Scalability**
  - Benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors
  - A single-threaded process can only run on one processor, regardless how many are available
  - Multithreading on a multi-CPU machines increases parallelism

# Multithreading Models

- 2 types of threads to be managed in a modern system
  - User threads
  - Kernel threads
- User threads are supported above the kernel & are managed without kernel support
- Kernel threads are supported & managed directly by OS. Kernel threads are supported within the kernel of the OS itself
- A relationship must exist between user and kernel threads
- There are 3 common ways to establish relationship between user & kernel thread
  - Many-to-One
  - One-to-One
  - Many-to-Many

# Many-to-One Model

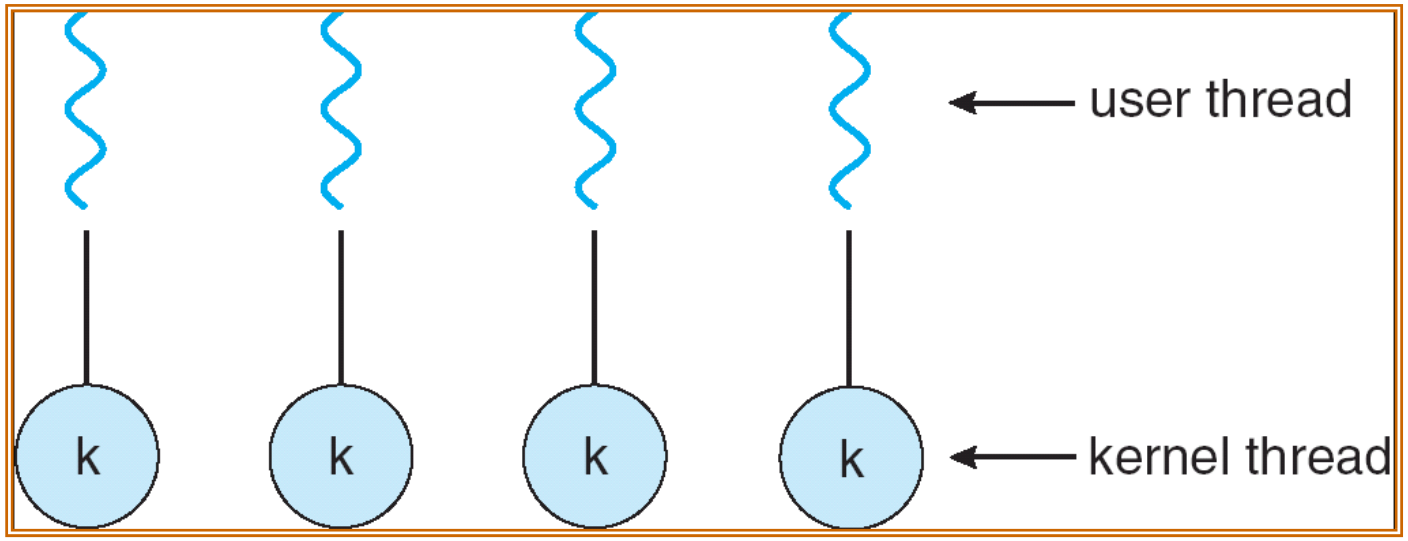



Many user-level threads mapped to single kernel thread

# Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is handled by the thread library in user space, which is very efficient.
- If any user thread makes a blocking system call, then entire process will be blocked. Due to this, other threads also cannot continue.
- Since, only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- Example:
  - Solaris Green Threads

# One-to-one Model

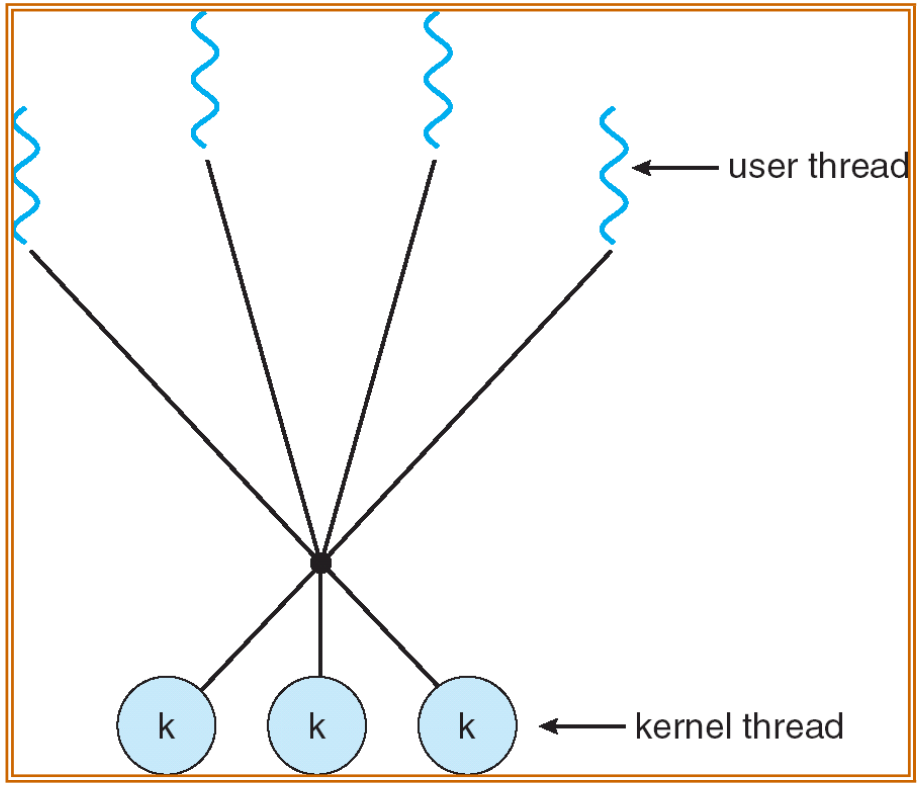



Each user-level thread maps to a unique kernel thread

# One-to-One

- Each user-level thread maps to a unique kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
- It provides concurrency(parallelism) more than the many-to-one by allowing another thread to run when a thread makes a blocking system call
- Allows multiple threads to run in parallel on multi-processors
- Cons
  - Creating user thread requires creating a unique kernel thread
  - Overhead of kernel-thread creation can burden performance & hence system becomes slower
  - Number of user threads bounded by number of kernel threads

# Many-to-Many Model




Many user-level threads mapped to single kernel thread

# Many-to-Many Model

- Many user level threads are mapped to an equal or smaller number of kernel threads
- Combines the best features of one-to-one and many-to-one models
- Allows OS to create a sufficient number of kernel threads
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package
- Advantages:
  - Developers can create as many user threads as necessary(no restriction)
  - Corresponding kernel threads can execute in parallel
  - Blocking kernel system calls do not block the entire process

# Threading Issues- 1) fork() & exec()

Issues to be considered while using multithreaded programs

**1) fork()** and **exec()** system calls

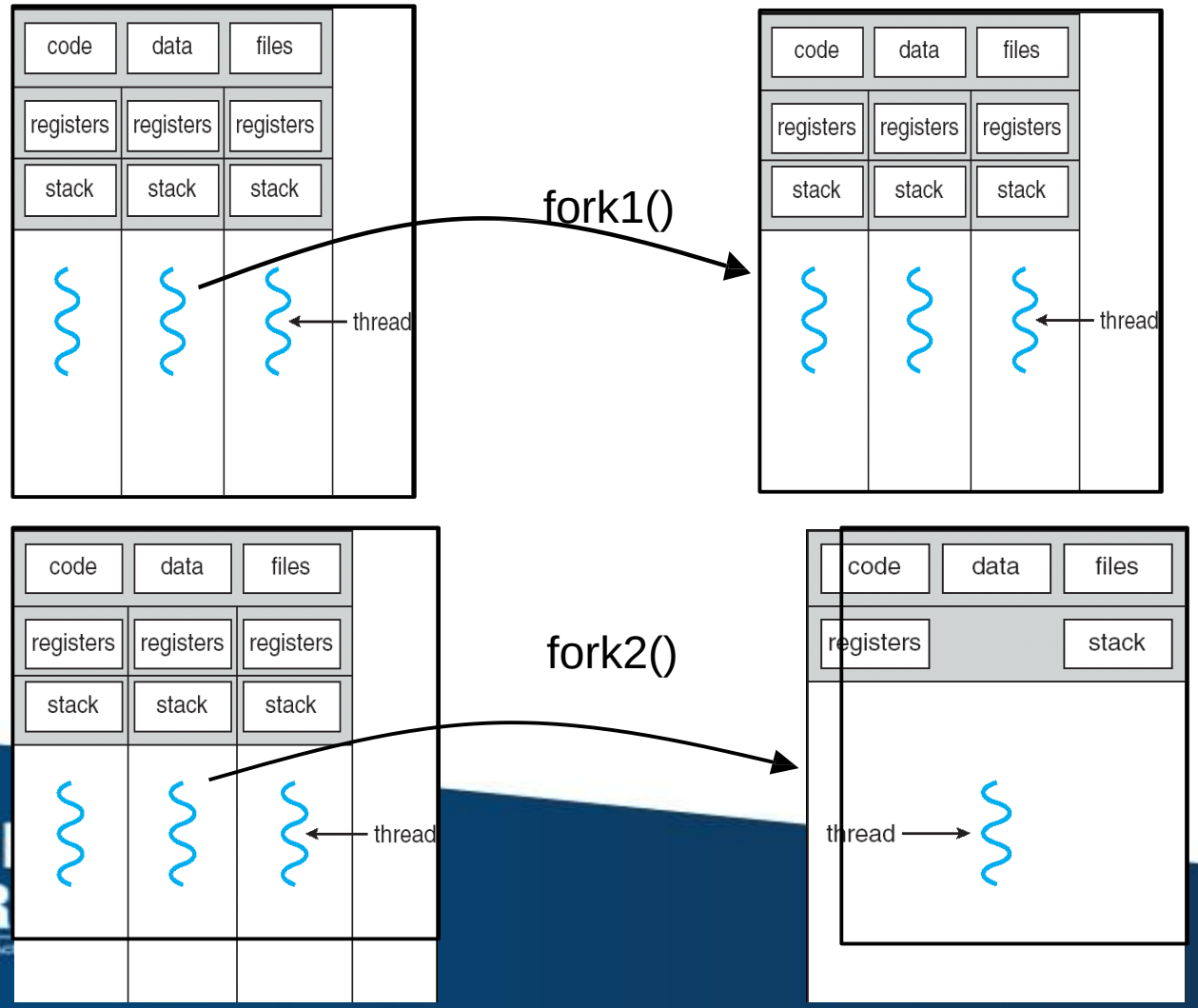
- Fork() is used to create a separate, duplicate process
- Now the question is, if one thread in a program calls fork(), whether the entire process is copied or new process is created only for that single thread?
- Actually it is system dependent.
- In some systems(Unix), it duplicates all threads(creates entirely new process) & as well as duplicates only the thread that invokes fork()

# Threading Issues

- **exec()** system calls
  - exec() is used to run an executable file in the context of an already existing process, replacing the previous executable
  - If a thread invokes exec(), the program specified in the parameter to exec() will replace the entire process, including all threads
  - Challenging is to decide, which of the 2 versions of fork() is to be used.
  - 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

# Semantics of fork()and exec()

- Does fork()duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of fork()
- exec()usually replaces all threads with new program



# Threading Issues- 2)Thread Cancellation

- **Thread cancellation** is the task of terminating a thread before it completes its action.
- A thread that is to be cancelled is referred as **target thread**.
- Threads that are no longer needed may be cancelled by another thread in one of 2 way's
  - Asynchronous Cancellation
  - Deferred Cancellation
- In **asynchronous cancellation**, one thread immediately terminates the target thread.
- Threads which are sharing the resources & as well as threads involved in inter-thread data transfer with the target thread gets affected.
- To overcome these disadvantages, deferred cancellation is preferred.
- In **deferred cancellation**, target thread itself sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically & exit nicely when it sees the flag set.

# Threading Issues- 3) Signal Handling

- Signals are used to inform processes indicating some events are occurred
- Signals are used only in the unix
- A signal may be received either synchronously or asynchronously
- Now the question is, when a multithreaded process receives a signal, to which thread should this signal be delivered?
- For this, there are 4 options:
  1) Deliver the signal to the thread to which that signal applies
  2) Deliver the signal to every thread in the process
  3) Deliver the signal to certain threads in the process
  4) Assign a specific thread to receive all signals to the process
- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting & which they are ignoring
- However, signals can be delivered to only one thread, which is generally the first thread that is accepting that particular signal
- UNIX provides 2 separate system calls **kill(pid,signal) & pthread_kill(tid,signal)**
- Windows doesn't support signals but they can be emulated using Asynchronous Procedure Calls(APC)

# Threading Issues- 4) Thread Pools

- Creating new threads everytime when it is needed & deleting after it is done, can be inefficient & can also lead to a (unlimited) number of threads being created.
- An alternative solution is to create a number of threads when the process first starts & put those threads into a Thread pool.
- Threads are allocated from the pool as needed & then returned back to the pool when no Longer needed.
- When no threads are available in the pool, the process may have to wait until one becomes fre/available
- Win32 provides thread pool through "**PoolFunction()**" function
- Java provides through **java.util.concurrent** package

# Threading Issues- 5) Thread-specific Data

- A process may Contain one thread (single threaded) or multiple threads (multi threaded).
- Threads belonging to process shares the data of that process .Actually, this sharing of data is one benefit of multithreaded programming.
- But in some situations, each thread might need it's own copy of Certain data. Such data is called **thread specific data**.
- If data is sharing, that cannot be modified by any thread.
- Sometimes, a thread needs some data & later it has to perform some Calculation/processing on that. For this reason, thread needs its own copy of specific data.

# Threading Issues- 6) Scheduler activations

- Final issue to be considered in an multithreaded programming is the communication between the kernel & the thread library
- Actually, some interface is required between the user thread & the kernel thread, particularly for many-to-many model
- This interface is an data structure typically known as Light Weighted Process (LWP).
- Generally, this interface is an virtual processor called Lightweight process
- There is one-to-one correspondence between LWP's & kernel threads

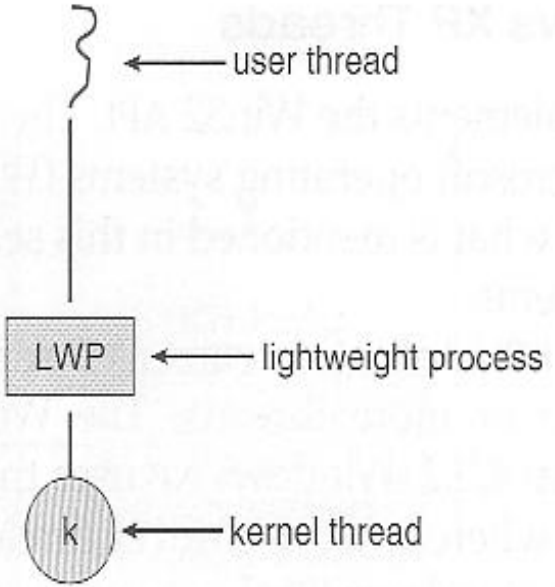



Figure 4.9 Lightweight process (LWP.)

# Basic Concepts of CPU Scheduling

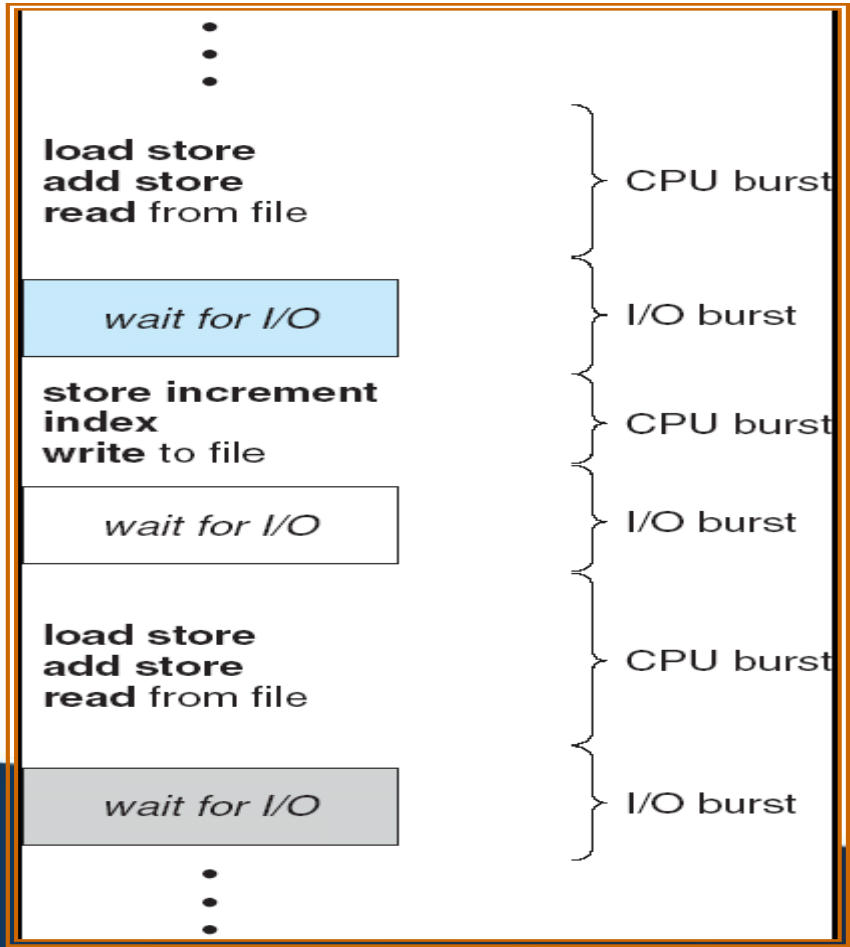
3.3 Basic Concepts of CPU Scheduling

# Basic Concepts

- CPU scheduling (Process Scheduling) is the basis of multiprogrammed OSs
- By switching the OS among processes the OS can make the computer more productive
- In a single-processor system, only one process can run at a time; all the other processes in the queue has to wait until the CPU is free.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- Assume, one process is running & some other processes are waiting for CPU. CPU becomes free after completing the task of already running process.
- Sometimes, running process may need some I/O resources to complete the task & they request for I/O. But, at that time, I/O is busy. So, the running process go to wait state & now CPU is idle.
- Instead of making CPU idle, we can allocate CPU to one of the waiting processes which are waiting for CPU.
- So, every time one process has to wait, another process can take over the use of CPU.
- Allocating CPU from one process to another is done by **scheduler** & the procedure is called **process scheduling**.

# CPU/IO Burst Cycle

- Almost every process needs CPU for some time to complete their calculations and /or computations.
- Meanwhile, every process needs I/O services in order to make some data transfer.
- When process is using CPU time, then it is referred as **CPU-burst**
- When process is using I/O, then it is referred as **I/O-burst**
- Processes alternate between these 2 states.
- Always, process begins & ends with CPU burst



# CPU Scheduler

- Whenever the CPU becomes idle, it is the job of CPU scheduler to select the next process to be executed, which are in the ready queue
- **CPU scheduler (or short term scheduler)** selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- Conceptually all the processes in the ready queue are lined up waiting for a chance to run on the CPU
  - Records in the queue are generally the PCB of the processes

**Pre-emptive scheduling**

CPU scheduler selects the next process in 4 situations

1) when a process switches from the running state to the waiting rate (such as, for an I/O request or nvocation of wait() system call)

2) when a process switches from running state to the ready state (for ex, in response to an interrupt).

3) When a process switches from waiting state to ready state (say, at Completion of I/0 or a return from wait()

4) when a process terminates

# CPU Scheduler

- For situations 1 & 4, there is no choice, CPU scheduler has to select new process
- For situations 2 & 3, there is a choice - either continue running the current process or select a new process
- If scheduling takes placeonly under situations 1 & 4, then the scheduling is called **non-preemptive scheduling** or also called as **co-operative**
- In non-preemptive scheduling, once a process starts running, it keeps running until it either voluntarily blocks or until it finishes.
- If scheduling takes place only under situations 2 & 3, then scheduling is called **preemptive scheduling**.
- Preemptive scheduling is only possible on the hardware that supports a timer interrupt.
- In preemptive scheduling, Currently running process is (stopped or) put it into the waiting state & new process is selected if that new process is having highest priority than the currently running.
- upto windows 3.x, non-preemptive is used & from windows 95 preemptive is using.
- But, there are some few problems / disadvantages associated with preemptive, they are

  *) when 2 process are sharing the data & one process is updating that data & assume it get preempted before updating. Then when another process access this shared data, it will get inconsistent data.

  *) preemption affects the design of OS Kernel. When OS Kernel is executing (say implementing / servicing system call) on behalf of process & that process is preempted, then it affects OS kernel.

# Dispatcher

- **Dispatcher** is the module that allocates 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
- The dispatcher needs to be as fast as possible, as it is run on every context switch.
- The time consumed by the dispatcher to do the above said functions is known as **dispatch latency.**
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

# Scheduling Criteria

- There are many different scheduling algorithms. Deciding which is the best scheduling algorithm in particular situation is difficult.
- There are several different criteria to consider, when trying to select the best scheduling algorithm for a particular situation & environment.
- Those criteria's are

* **CPU utilization :** Always CPU should be kept busy always as possible. Generally, CPU utilization can range from 0 to 100%. On a real system, CPU usage should range from 40% (lightly loaded system) to 90% (heavily loaded system).

* **Throughput** :In order to measure, amount of work done by CPU, throughput is used. Number of processes that are Completed per unit of time is called as throughput.

For long processes, throughput may be one process per hour & for short processes, it may be 10 processes per second.

# Scheduling Criteria (2)

*) **Turnaround time** – amount of time to execute a particular process. Interval from the time of submission of a process to the time of completion

turnaround time is the sum of the periods spent waiting to get into main memory, waiting in the ready queue, executing on the CPU & doing I/O.

*) **Waiting time** – amount of time a process has been waiting in the ready queue

*) **Response time** – In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early & can continue computing new results while previous results are being output to the user.

thus, another measure is the time from the submission of a request until the first response is produced. This measure, Called **response time**, is the time it takes to start responding, not the time it takes to output the response.

# Optimization Criteria

CPU utilization

- Maximize

Throughput

- Maximize

Turnaround time

- Minimize

Waiting time

- Minimize

Response time

- Minimize

# Scheduling Algorithms

- CPU should select one process out of many processes which are in ready queue after completing the current process
- In order to select, there are so many scheduling algorithms
- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Priority scheduling
- Round Robin scheduling

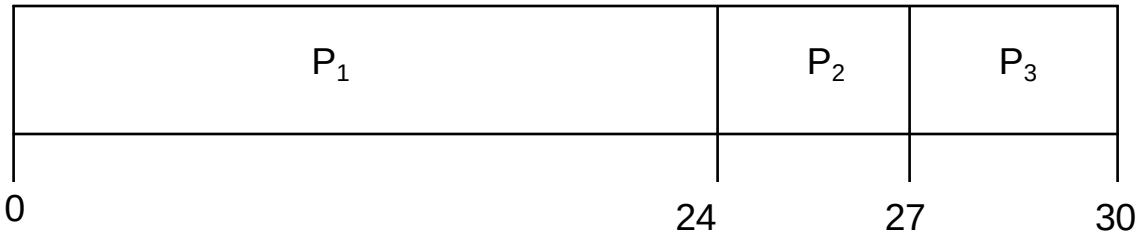
# First-Come, First-Served Scheduling (FCFS)

- **Simplest CPU scheduling algorithm**
- **Process that requested the CPU first is allocated the CPU first**
- **Implementation managed by a First In First Out (FIFO) queue**
  - When a process enters the ready queue its PCB is linked onto the tail of the queue
  - When the CPU is free, it is allocated to the process at the head of the queue and the running process is removed from the queue
- **Disadvantage:**
  - If the currently running process (or the first process in the ready queue) takes much time then all other processes has to wait for long time
  - Average waiting time under FCFS policy is often quite long
- **Nonpreemptive algorithm** → once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU

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

| Process | CPU Burst Time (ms) |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

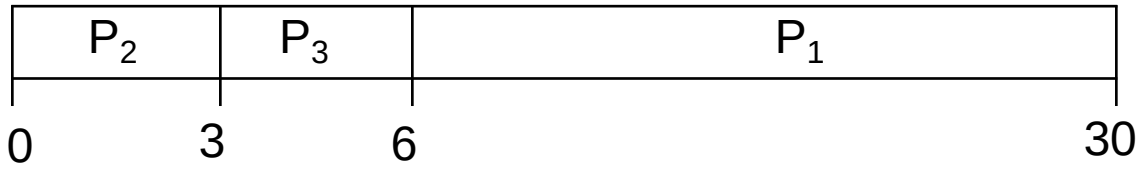
- Suppose that the processes arrive (at time 0) in the order: $P_1$ , $P_2$ , $P_3$
  The Gantt Chart for the schedule is:




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

# FCFS Scheduling (Cont.)

Suppose if same 3 processes arrives in the order $P_2$ , $P_3$ , $P_1$

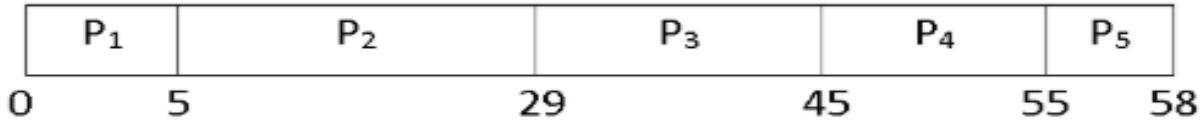



- The Gantt chart for the schedule is:
- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3m sec
- Much better than previous case
- Thus, average waiting time of FCFS keeps on changing according to their arrival & their burst time
- While processes wait in the ready queue, I/O devices are idle. At the same time, CPU-bound process finishes its CPU burst & moves to an I/O device.
- All the I/O bound processes, which have short CPU bursts, execute quickly & move back to the I/O queues. At this point, CPU sits idle.
- CPU-bound process will then move back to the ready queue & be allocated the CPU.
- Again, all I/O processes end up waiting in the ready queue until the CPU bound process is done.
- There is a **Convey Effect** as all the other processes wait for the one big process to get off The CPU.

# FCFS contd.

| Process | Burst Time(ms) |
|---|---|
| $P_1$ | 5 |
| $P_2$ | 24 |
| $P_3$ | 16 |
| $P_4$ | 10 |
| $P_5$ | 3 |

calculate the average waiting time, average turnaround time and throughput.




- **Average Waiting Time**

*Waiting Time = Starting Time - Arrival Time*

Waiting time of P1 = 0 P2 = 5 - 0 = 5 ms

P3 = 29 - 0 = 29 ms

P4 = 45 - 0 = 45 ms

P5 = 55 - 0 = 55 ms

*Average Waiting Time = Waiting Time of all Processes / Total Number of Process*

Therefore, average waiting time = (0 + 5 + 29 + 45 + 55) / 5 = 25 ms

- **Average Turnaround Time**
- *Turnaround Time = Waiting time in the ready queue + executing time + waiting time in waiting-queue for I/O*

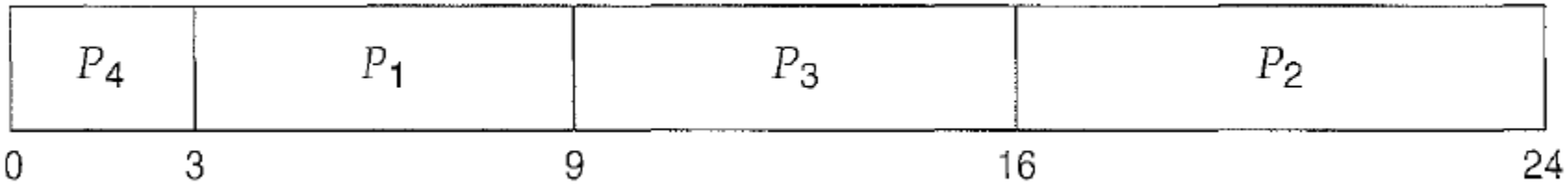
- Turnaround time of P1 = 0 + 5 + 0 = 5ms
  P2 = 5 + 24 + 0 = 29ms
  P3 = 29 + 16 + 0 = 45ms
  P4 = 45 + 10 + 0 = 55ms
  P5 = 55 + 3 + 0 = 58ms
  Total Turnaround Time = (5 + 29 + 45 + 55 + 58)ms = 192ms
  Average Turnaround Time = (Total Turnaround Time / Total Number of Process) = (192 / 5)ms = 38.4ms

- **Throughput**
- Here, we have a total of five processes. Process P1, P2, P3, P4, and P5 takes 5ms, 24ms, 16ms, 10ms, and 3ms to execute respectively. Throughput = (5 + 24 +16 + 10 +3) / 5 = 11.6ms
  It means one process executes in every 11.6 ms.

# Scheduling Algorithms: SJF and SRTF

- CPU is assigned to the process that has the smallest next CPU burst
- If the 2 processes are having same CPU burst, tie can be broken using FCFS
- This SJF is also called as **Shortest next CPU burst algorithm**
- Consider the following set of processes with the length of the CPU burst given in millisec

| Process | Burst Time |
|---|---|
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |




Waiting time of P4=0msec
Waiting time of P1=3msec
Waiting time of P3=9msec
Waiting time of P2=16msec

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

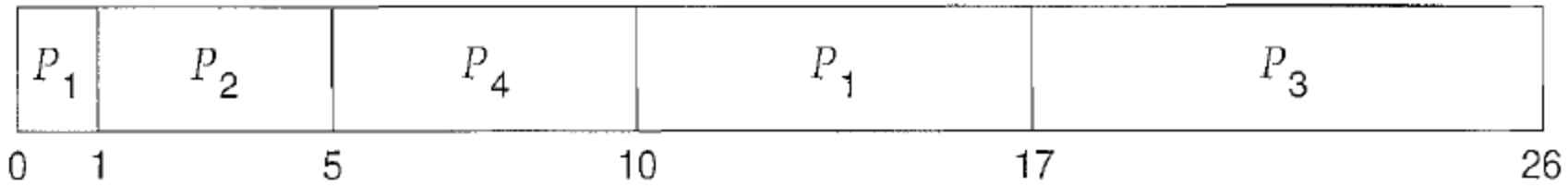
# Shortest-Job-First (SJF)

- SJF is an optimal scheduling algorithm when compared to FCFS.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- The real difficulty with the SJF algorithm is knowing length of the next CPU request.
- SJF algorithm can be either **preemptive & non-preemptive**.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest- remaining-time-first**

# SJF

Consider the following 4 processes, with the length of CPU burst given in milliseconds.
If the 4 Processes arrives in the order shown in the table

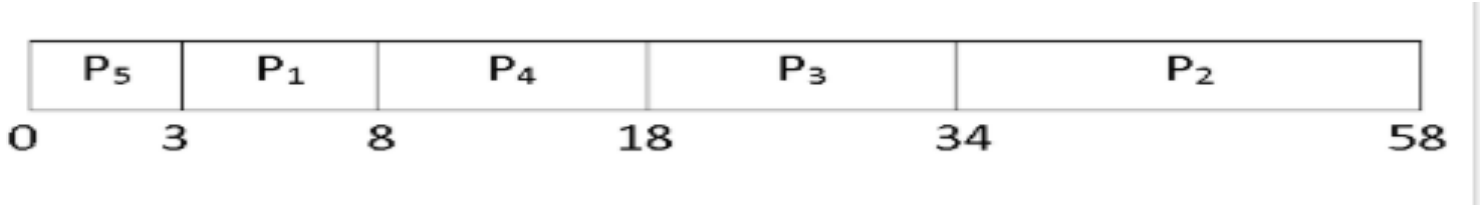
| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |




Waiting time for P1=0 + (10-1)= 9 msec
Waiting time for P2=0
Waiting time for P3=17 – 2 = 15 msec
Waiting time for P4=5 – 3 = 2 msec
Average waiting time =(9+0+15+2)/4= 6.5 msec

# SJF (Shortest Job First) Scheduling

| Process | Burst Time(ms) |
|---|---|
| $P_1$ | 5 |
| $P_2$ | 24 |
| $P_3$ | 16 |
| $P_4$ | 10 |
| $P_5$ | 3 |




Assuming arrival time is 0 for all the processes
Waiting time for P1 = 3 – 0 = 3 msec
Waiting time for P2= 34 – 0 = 34 msec
Waiting time for P3= 18 – 0 = 18 msec
Waiting time for P4= 8 – 0 = 8 msec
Waiting time for P5= 0 msec
Average Waiting Time = (3 + 34 + 18 + 8 + 0) / 5 = 12.6ms

Turnaround Time of
P1 = 3 + 5 = 8ms
P2 = 34 + 24 = 58ms
P3 = 18 + 16 = 34ms
P4 = 8 + 10 = 18ms
P5 = 0 + 3 = 3ms

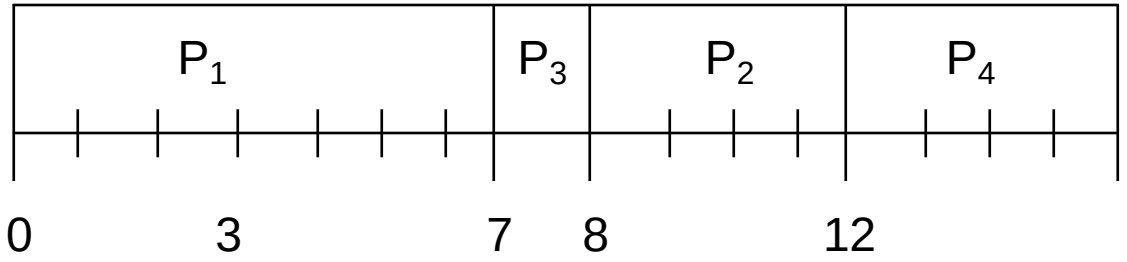
Average Turnaround Time = (8 + 58 + 34 + 18 + 3) / 5 = 24.2ms

Here, we have a total of five processes. Process P1, P2, P3, P4, and P5 takes 5ms, 24ms, 16ms, 10ms, and 3ms to execute respectively
Therefore, Throughput will be same as above problem i.e., 11.6ms for each process

# Example of Non-Preemptive SJF

| Processes | Arrival Time | Burst Time (ms) |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

- SJF (non-preemptive)

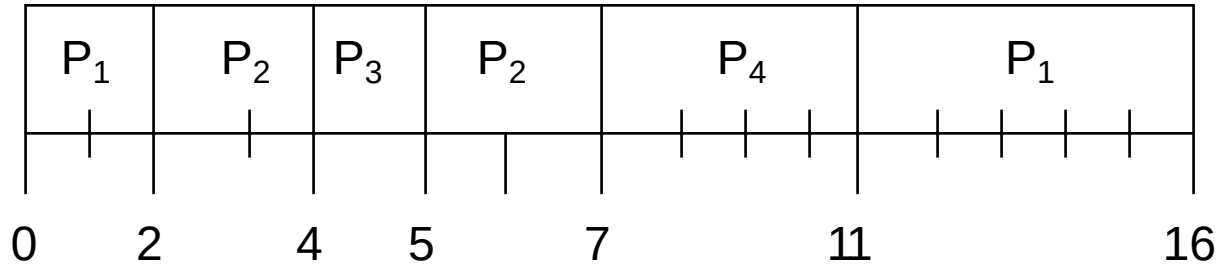



- Average waiting time = (0 + 6 + 3 + 7)/4= 4msec

# Example of Preemptive SJF

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

- SJF (preemptive)

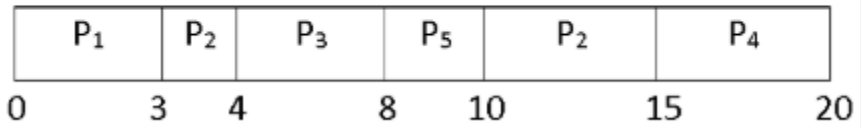



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

# SRTF

## Shortest Remaining Time First(SRTF)

| Process | Burst Time(CPU) | Arrival Time(ms) |
|---|---|---|
| $P_1$ | 3 | 0 |
| $P_2$ | 6 | 2 |
| $P_3$ | 4 | 4 |
| $P_4$ | 5 | 6 |
| $P_5$ | 2 | 8 |



# Scheduling Algorithms: Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (in this text, smallest integer ≡ highest priority)
  - Equal priority processes are scheduled using FCFS
  - Priority scheduling can be:
    - preemptive: preempt CPU if priority of newly arrived process is higher than priority of currently running process
    - nonpreemptive: put the new highest priority process at the head of the ready queue
- SJF is a priority scheduling where priority is the inverse of the predicted next CPU burst time (largest CPU burst → lowest priority)
- External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc

# Priority Scheduling

- Problem ≡ Starvation – low priority processes is indefinitely blocked and may never execute
  - In a heavily loaded system a steady stream of higher priority processes can prevent a lower priority process from ever getting the CPU
- Solution ≡ Aging – as time progresses gradually increase the priority of processes that have been waiting for a long time

# Priority Scheduling

## Priority Scheduling Example

| Process | CPU Burst Time | Priority |
|---|---|---|
| $P_1$ | 6 | 2 |
| $P_2$ | 12 | 4 |
| $P_3$ | 1 | 5 |
| $P_4$ | 3 | 1 |
| $P_5$ | 4 | 3 |

**Gantt Chart**

| $P_4$ | $P_1$ | $P_5$ | $P_2$ | $P_3$ |
|---|---|---|---|---|

### Average Waiting Time

First of all, we have to find out the waiting time of each process.

Waiting Time of process

P1 = 3ms

P2 = 13ms

P3 = 25ms

P4 = 0ms

P5 = 9ms

Therefore, Average Waiting Time = (3 + 13 + 25 + 0 + 9) / 5 = 10ms

### Average Turnaround Time

First finding Turnaround Time of each process.

Turnaround Time of process

P1 = (3 + 6) = 9ms

P2 = (13 + 12) = 25ms

P3 = (25 + 1) = 26ms

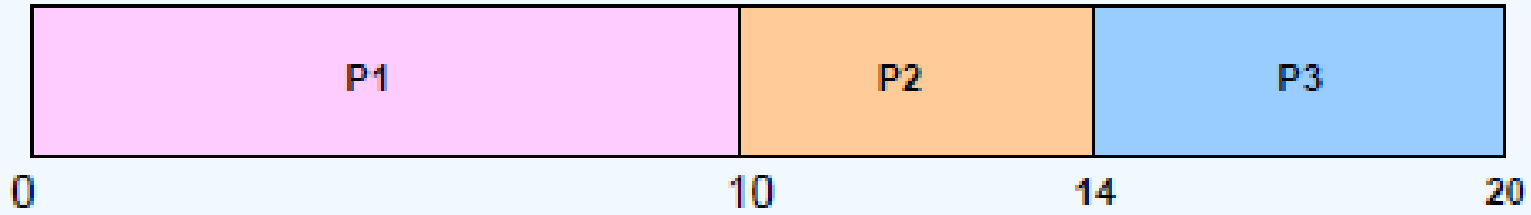
P4 = (0 + 3) = 3ms

P5 = (9 + 4) = 13ms

Therefore, Average Turnaround Time = (9 + 25 + 26 + 3 + 13) / 5 = 15.2ms

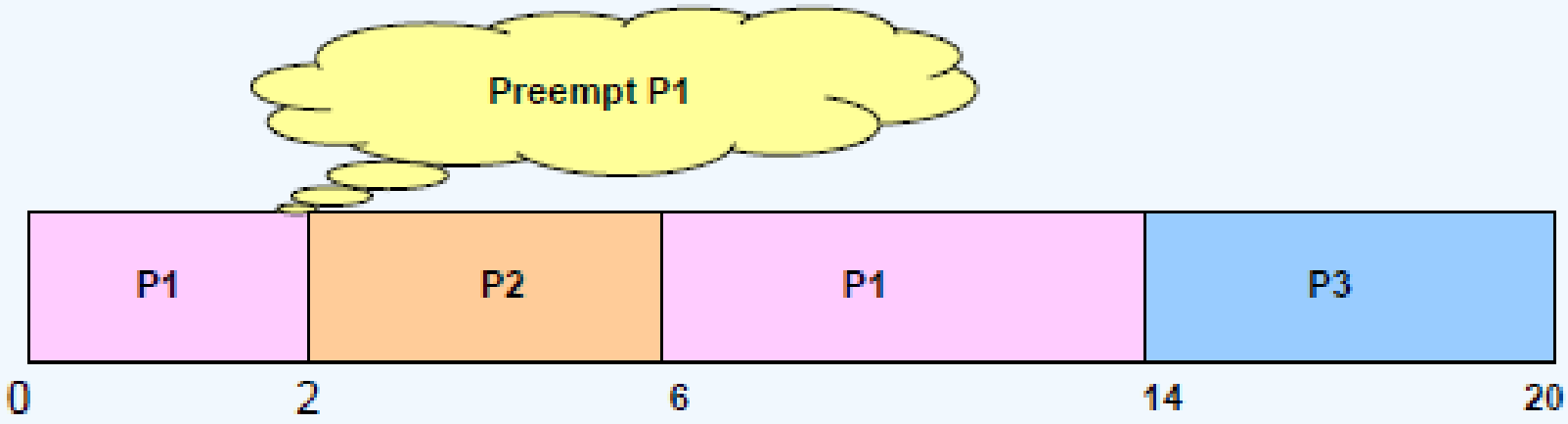
# Priority Scheduling (Non Preemptive method)

| Process | Execution Time | Priority | Arrival time in msec |
|---|---|---|---|
| P1 | 10 | 2 | 0 |
| P2 | 4 | 1 | 2 |
| P3 | 6 | 3 | 0 |



# Priority Scheduling (Preemptive method)

| Process | Execution Time | Priority | Arrival time in msec |
|---|---|---|---|
| P1 | 10 | 2 | 0 |
| P2 | 4 | 1 | 2 |
| P3 | 6 | 3 | 0 |



# Scheduling Algorithms: Round Robin (RR)

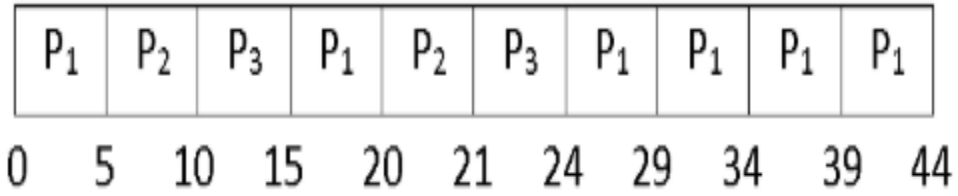
- This round robin scheduling algorithm is designed especially for time-sharing systems.
- RR is similar to FCFS scheduling, except that CPU bursts are assigned with a small unit of time called **time quantum** or **time slice**
- Each process gets a small unit of CPU time (*time quantum*), usually 10- 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) x *q* time units until its next quantum.
- Performance depends on size of q
  - *q* large ⇒ FIFO
  - *q* small ⇒ RR approach is called processor sharing and creates appearance that each of the *n* processes has its own processor running at speed 1/n
  - q must be large with respect to context switch, otherwise overhead is too high

# Round Robin

time quantum 5ms

| Process | CPU Burst Time |
|---|---|
| $P_1$ | 30 |
| $P_2$ | 6 |
| $P_3$ | 8 |

**Gantt Chart**




**Average Waiting Time**

For finding Average Waiting Time, we have to find out the waiting time of each process.

Waiting Time of

P1 = 0 + (15 - 5) + (24 - 20) = 14ms

P2 = 5 + (20 - 10) = 15ms

P3 = 10 + (21 - 15) = 16ms

Therefore, Average Waiting Time = (14 + 15 + 16) / 3 = 15ms

**Average Turnaround Time**

Same concept for finding the Turnaround Time.

Turnaround Time of

P1 = 14 + 30 = 44ms

P2 = 15 + 6 = 21ms

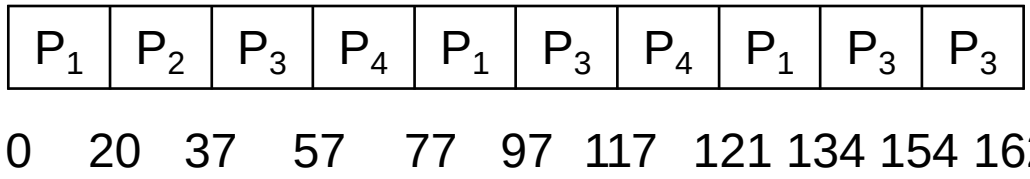
P3 = 16 + 8 = 24ms

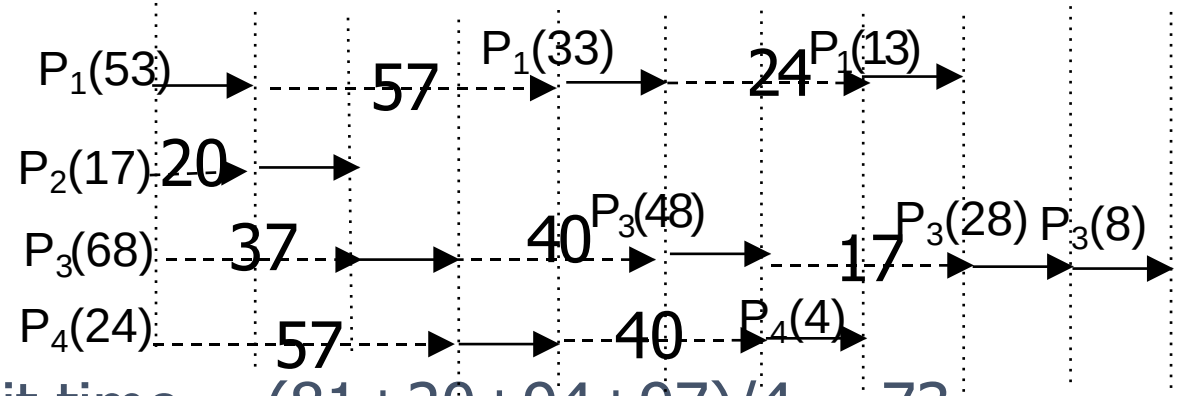
Therefore, Average Turnaround Time = (44 + 21 + 24) / 3 = 29.66ms

# Round Robin Scheduling

- Each process is given CPU time in turn, (i.e. time quantum:usually 10-100 milliseconds), & thus waits no longer than ( n – 1 ) * time quantum
- time quantum = 20

| Process | Burst Time | Wait Time |
|---|---|---|
| $P_1$ | 53 | 57 +24 = 81 |
| $P_2$ | 17 | 20 |
| $P_3$ | 68 | 37 + 40 + 17= 94 |
| $P_4$ | 24 | 57 + 40 = 97 |

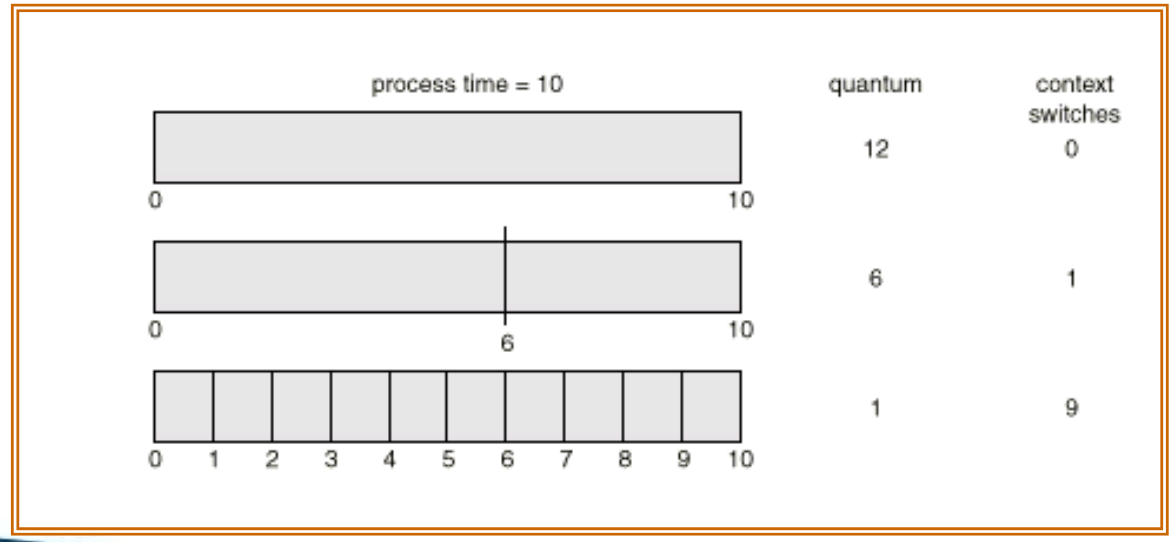



Average wait time = (81+20+94+97)/4 = 73

# Round Robin Scheduling

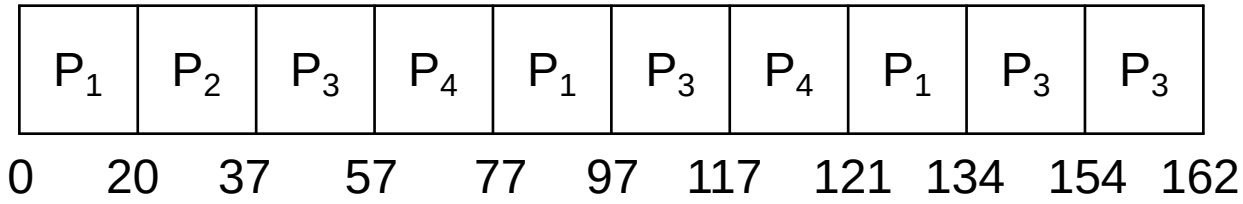
- Typically, higher average turnaround than SJF, but better *response*.
- Performance
  - *q* large ⇒ FCFS
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.



# Example of RR with Time Quantum = 20

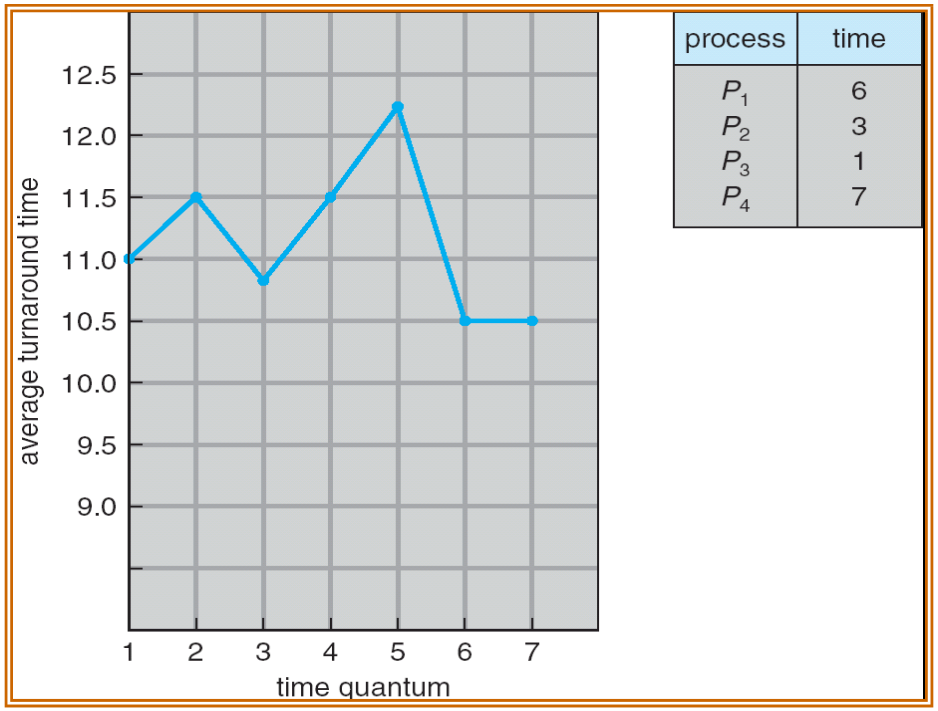
| Process | Burst Time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

- The Gantt chart is:




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

# Turnaround Time Varies With The Time Quantum




| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

Average turnaround time does not necessarily improve as size of q increases

Generally, average turnaround time can be improved if most processes finish their CPU burst in a single time quantum

# Thank You