# Operating Systems (UNIT-2)

# Overview of Syllabus



Unit 1- Introduction Introduction What Operating System do, Operating System structure, Operating system Operations. System Structures Operating system services, **System Calls**, Types of System calls Process Management Process concept, Process scheduling, Operations on processes

**Unit-2 Multithreaded programming Overview, Multicore programming, Multithreading models, Thread libraries - pthreads CPU scheduling and Process Synchronization Basic concepts, scheduling criteria, scheduling algorithms-FCFS, SJF, RR, priority, Real-time CPU scheduling**

Unit 3- Process Synchronization Background, The Critical section problem, Peterson's Solution Process Synchronization hardware, Mutex locks, Semaphores, Classic problems of synchronization

Unit 4- Main Memory Management Background, Swapping, Contiguous memory allocation, Segmentation, Paging, Structure of page table. Virtual memory Background, Demand Paging, Copy-on-write, Page replacement, Allocation of frames, Thrashing

Unit -5 File Systems File Naming, File Structure, File Types, File Access, File Attributes, File Operations, An example program using File-System calls, File-System Layout, Implementing Files

# Overview –Threads

**Process : The program loaded into the memory and getting executed is called Process**

**Thread : Basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack**

**Threads aren't actually allowed to exist outside a process**

**Process may have many threads in it**

**Each and every thread belongs to one single process**

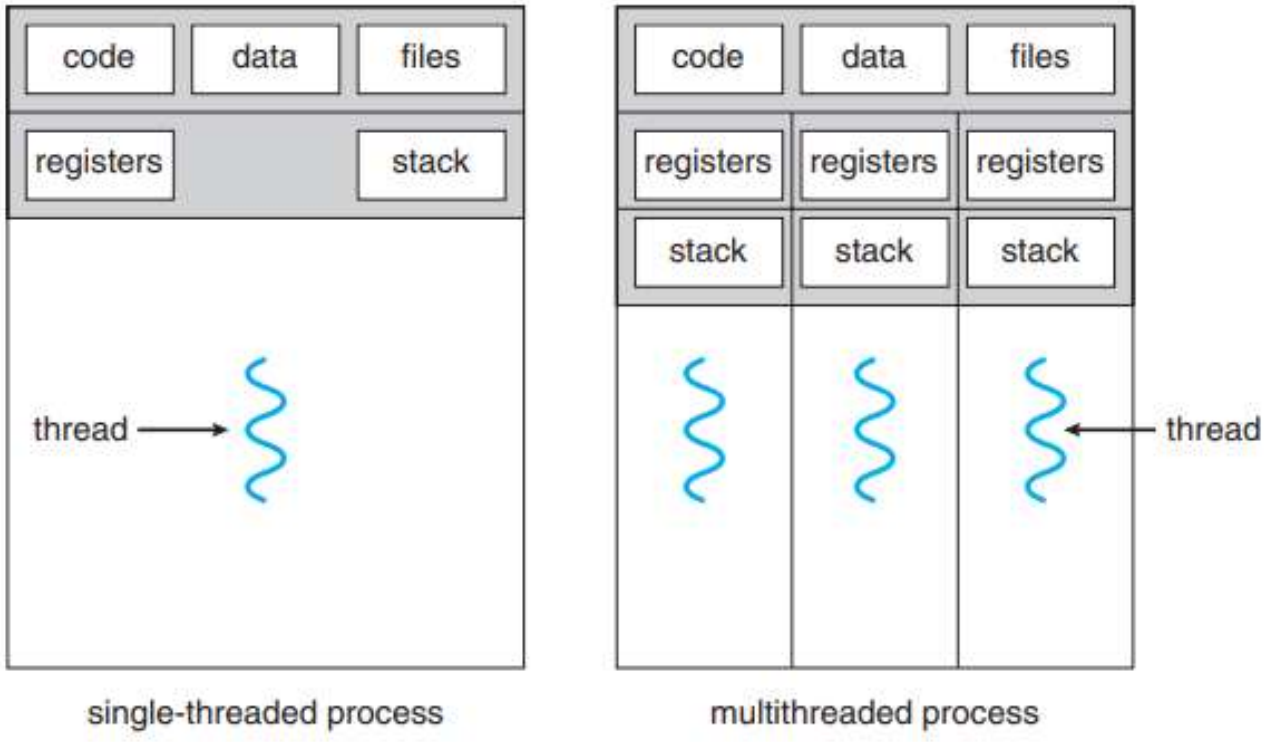
**It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.**

# Overview –Single and Multithreaded Processes

A traditional (or heavyweight) process has a single thread of control.

If a process has multiple threads of control, it can perform more than one task at a time.

**Example**
A word processor may have a thread for **displaying graphics**, another thread for **responding to keystrokes from the user**, and a third thread for **performing spelling and grammar checking in the background.**



# Overview –Single and Multithreaded Processes

**Example**

If the web server ran as a **traditional single-threaded process**, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

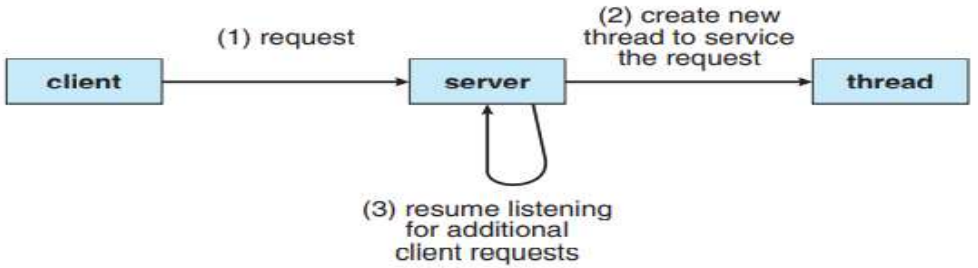
**Two Solutions to Address this Problem :**

**First : Create the New  traditional Process to service the request.**

**Second : Create the New Thread to service the request.**

Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process.

More efficient to use **one process that contains multiple threads**




Multithreaded server architecture

# Why Do We Need Thread?

- **Creating a new thread in a current process requires significantly less time than creating a new process.**
- **Threads can share common data without needing to communicate with each other.**
- **When working with threads, context switching is faster.**
- **Terminating a thread requires less time than terminating a process.**

# Benefits of Threads

- **Responsiveness –** May allow continued execution if part of process is blocked, this quality is useful to design especially important for user interfaces. If one thread fails, other thread will pitch in, so the user will not experience the lag.
- **Resource Sharing –** Processes can only share resources through techniques such as **shared memory** and **message passing**. Such techniques must be explicitly arranged by the programmer. But in case of the thread they share the resources by **default**
- **Economy –** Allocating memory and resources for process creation is costly,Cheaper than process creation, thread switching lower overhead than context switching.
- Eg : In Solaris OS , Creating Process is 30 minutes slower compared to thread process creation.
- **Scalability** – Threads can take advantage of multiprocessor architectures

# Multi Core Programming

Earlier in the history of computer design, in response to the need for more computing performance, **single-CPU systems evolved into multi-CPU systems.**

Recent Trend - in system design is to place **multiple computing cores on a single chip.**

**CPU Chips having more than one core- such systems are called as Multi Core Systems**

Each core appears as a separate processor to the operating system.

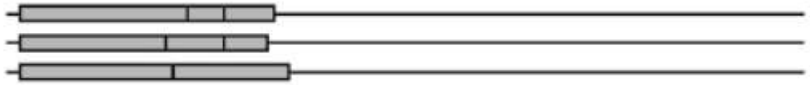
Consider an application with four threads.

**System with a single computing core-** Processing core is capable of executing only one thread at a time.

**System with multiple cores** - Threads can run in parallel, because the system can assign a separate thread to each core.

# Multi Core Programming

Concurrent, non-parallel execution



Concurrent, parallel execution

**Concurrency is the task of running and managing the multiple computations at the same time.**

**Parallelism is the task of running multiple computations simultaneously.**

**Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:

**Dividing activities :** Involves examining applications to find areas that can be divided into separate, concurrent tasks.

**Balance :** programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks.

**Data splitting :** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores

**Data dependency:** one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency

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

A. Many-to-One

B. One-to-One

C. Many-to-Many

# Many-to-One

**Many user-level threads mapped to single kernel thread**

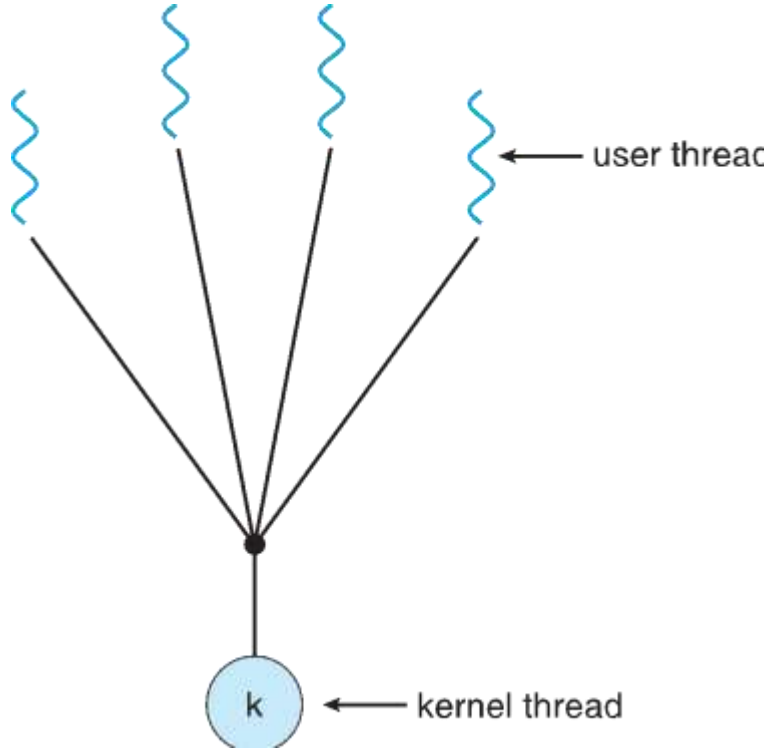
**One thread blocking causes all to block**

**Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time (It doesn't implement the multiprocessor )**

**Few systems currently use this model**

**Examples:**

- **Solaris Green Threads**
- **GNU Portable Threads**



# One-to-One

**Each user-level thread maps to kernel thread**

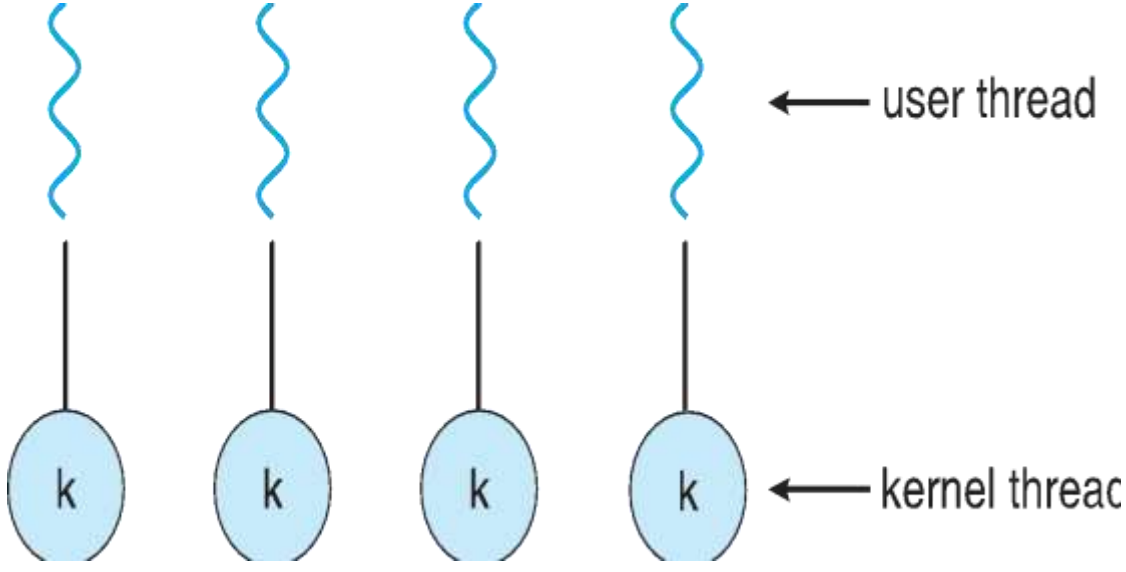
**Creating a user-level thread creates a kernel thread**

**More concurrency than many-to-one**

**Number of threads per process sometimes restricted due to overhead**

**Examples**

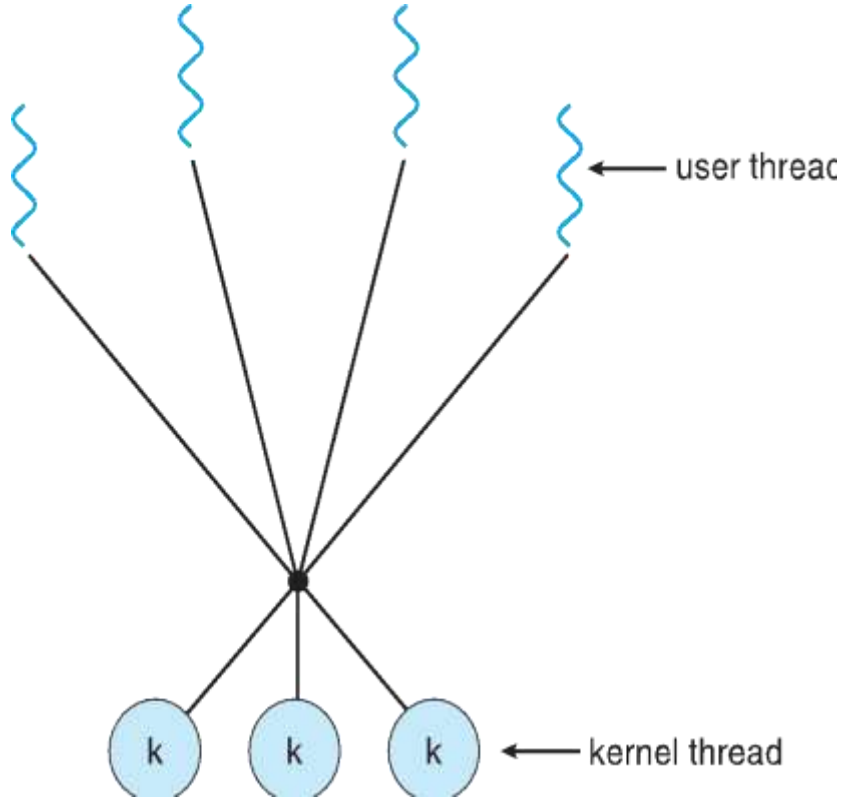
- **Windows**
- **Linux**
- **Solaris 9 and later**



# Many-to-Many Model

**Allows many user level threads to be mapped to many kernel threads**

**Allows the  operating system to create a sufficient number of kernel threads**

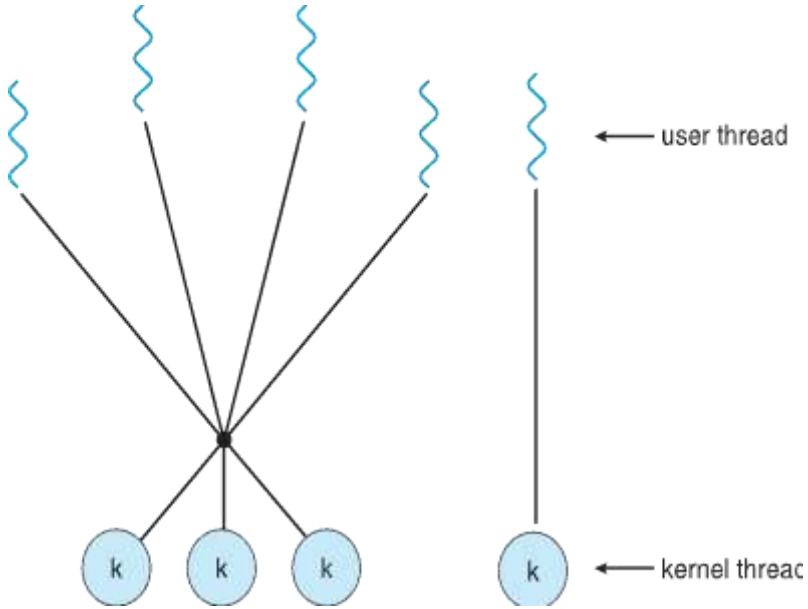


# Two-level Model

**Similar to M:M, except that it allows a user thread to be bound to kernel thread**

**Examples**

- **IRIX**
- **HP-UX**
- **Tru64 UNIX**
- **Solaris 8 and earlier**

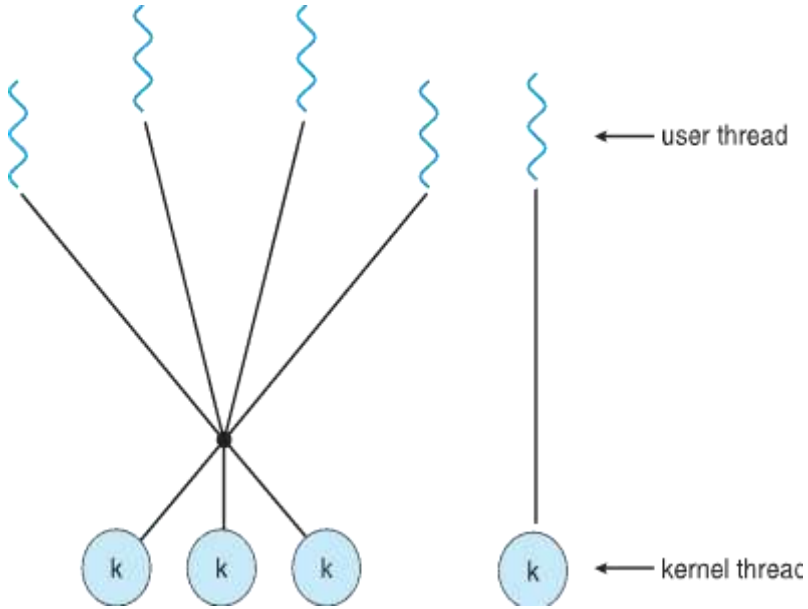


# Two-level Model

**Similar to M:M, except that it allows a user thread to be bound to kernel thread**

**Examples**

- **IRIX**
- **HP-UX**
- **Tru64 UNIX**
- **Solaris 8 and earlier**



# Types of Parallelism

## *Data parallelism*

Focuses on distributing **subsets of the same data across multiple computing cores** and performing the same operation on each core.

**Example:**
Summing the contents of an array of size N.
thread 1, running on core 1, could sum the elements [0] . . . [N/2 − 1] while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1].

## *Task parallelism*

- It involves distributing of the Tasks(Threads) across the multiple computing cores.
- Each thread is performing a unique operation on the data.

# Thread Libraries

- A thread library provides the programmer with an API for creating and managing threads.
- There are two primary ways of implementing a thread library.
- *Library entirely in user space*
- The first approach is to provide a library entirely in user space with no kernel support.
- All code and data structures for the library exist in user space.
- This means that invoking a function in the library results in a local function call in user space and not a system call.
- *Library entirely in Kernel space*
- Kernel-level library supported directly by the operating system.
- In this case, code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel

# Pthreads

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.

**POSIX**

- The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

- This is a specification for thread behaviour, not an implementation.

- API specifies behavior of the thread library, implementation is up to developer

- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# Pthreads – Example

- **Thread Program to calculate the sum of first N Numbers**

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

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

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

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

- **Include the Libraries**
- **Define the Global Variable Sum**
- **Define the Pointer Runner which access Void Pointer**
- **Void*parameter : Function Declaration**
- **Define the command line arguments**
- **Argc- Argument Counts (Int Type)**
- **Argv – Stores the argument we have passed**
- **Eg : Let the file name is sum.c, then argv[0]**

**contains the file name and argv[1] contains the N number**

**Tid : Thread Identification Number**

**Attr_t == Data Type and attr == Is the Variable (Using these variables, we can give some new attributes to the thread)**

# Pthreads – Example

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

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

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

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

   pthread_exit(0);
}
```

- **Init – Initialization and & attr is used to store the address of the variable.**

**Creation of Threads : It contains four arguments (thread identification number, attributes, function and argument)**

**Join : Blocks the main function, until the runner function executes**

(Parent thread will wait for function to terminate, and this is done by Join function)

The atoi() function **converts a character string to an integer value**

The summation thread will terminate when it calls the function pthread exit().

# CPU scheduling and Process Synchronization

CPU scheduling is the basis of Multiprogrammed operating systems.

Goal : Computer more productive

***Basics***

In a single-processor system, only one process can run at a time.

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

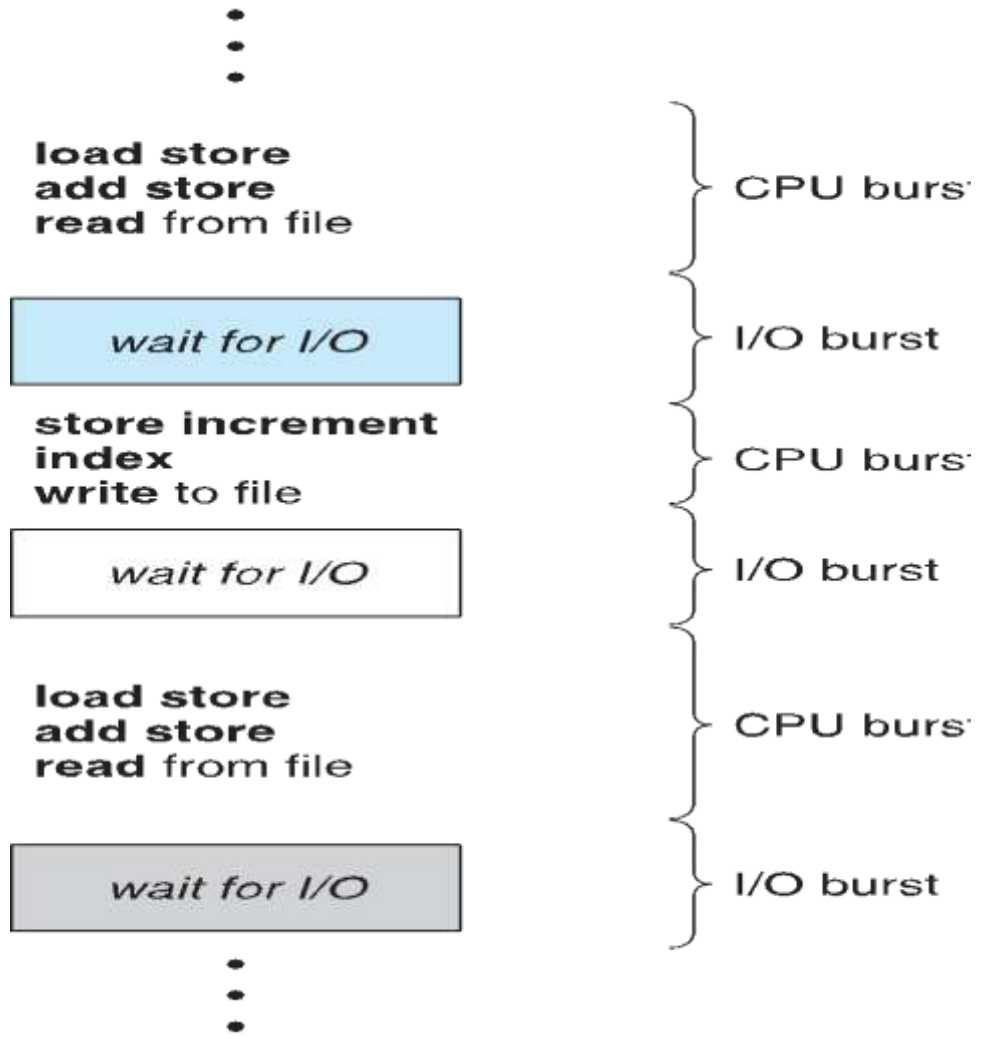
Several processes are kept in memory at one time

# CPU –I/O Burst Cycle

**Process Execution consist of two states i.e. in CPU execution and I/O wait**

**CPU Burst : Time in which the process is in CPU execution**

**I/O Burst : Time in which the process is in I/O wait**



# Scheduling Criteria

**CPU utilization** – Keep the CPU as busy as possible. (In lightly loaded system,

**Throughput** – (Measure of the work done by the CPU) i.e Number of the of processes that complete their execution per time unit.

**Turnaround time** – amount of total time to execute a particular process (Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O)

**Waiting time** – Amount of time a process has been waiting in the ready queue.

**Response time** – Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

# Scheduling Algorithm Optimization Criteria

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

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

Simplest CPU-scheduling algorithm

The process that requests the CPU first is allocated the CPU first.

It is easily implemented by 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.

Drawback : the average waiting time under the FCFS policy is often quite long

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

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

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


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

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



Suppose that the processes arrive in the order:

$P_2$ , $P_3$ , $P_1$

- The Gantt chart for the schedule is:


- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - All the other processes wait for the one big process to get off the CPU

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

FCFS is the Non Preemptive Scheduling.

In FCFS Scheduling , once the CPU has allocated the process, that process keeps

the CPU until its executed completely.

Drawback : This will not work in Time Sharing Systems, and doesn’t support the

Context Switch

# FCFS Problem

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 4 | 5 |
| P2 | 6 | 4 |
| P3 | 0 | 3 |
| P4 | 6 | 2 |
| P5 | 5 | 4 |

**Calculate the Average Waiting Time and Turn around time**

**Turn Around Time == Completion time – Arrival Time**

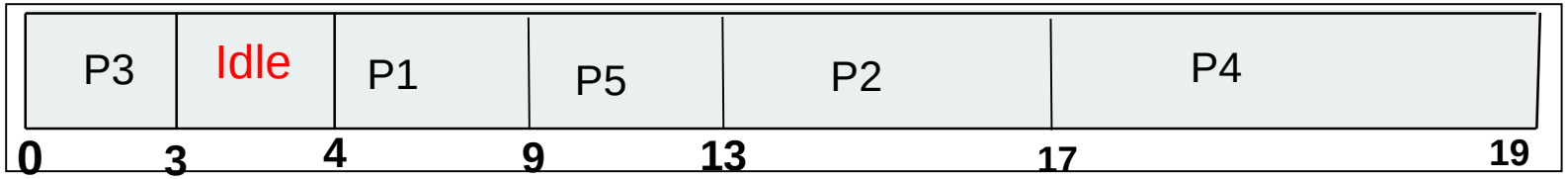
**Waiting Time == Turn around time - Burst Time**

**Note : If the two process have same arrival time, Lowest Process id will be having the Highest priority**

# FCFS Problem



| Process ID | Arrival Time | Burst Time | Turn around Time | Waiting Time |
|---|---|---|---|---|
| P1 | 4 | 5 | 5 | 0 |
| P2 | 6 | 4 | 11 | 7 |
| P3 | 0 | 3 | 3 | 0 |
| P4 | 6 | 2 | 13 | 11 |
| P5 | 5 | 4 | 8 | 4 |




**Average Turn around time = (Total Turn Around Time )/ (Total Number of Process) =(40/5)= 8 units**

**Average Waiting Time == = (Total Waiting Time )/ (Total Number of Process) = 22/5 = 4.4 units**

# FCFS Problem

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 3 |
| P2 | 1 | 2 |
| P3 | 2 | 1 |
| P4 | 3 | 4 |
| P5 | 4 | 5 |
| P6 | 5 | 2 |

**Calculate the Efficiency of the algorithm, by applying the FCFS Process and consider 1 unit of overhead in scheduling the process**

# Shortest JoB First (SJF)

- **When the CPU is available, it is assigned to the process that has the smallest next CPU burst**

- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie

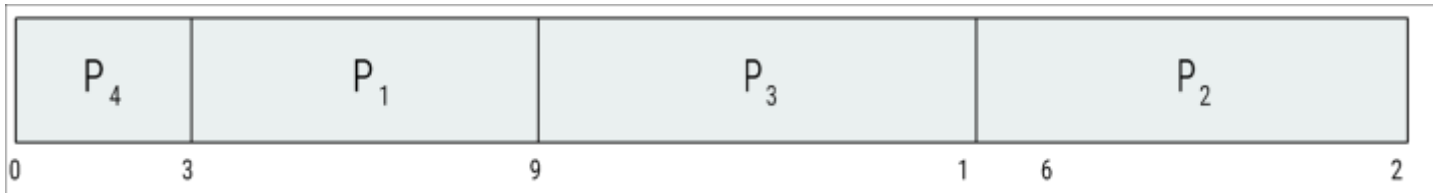
- **SJF Algorithm is Pre-emptive or Non Premptive (If the two process have same burst time, Lowest Process**
- **id will be having the Highest priority)**

- More appropriate term for this scheduling method would be the **shortest-next CPU-burst algorithm**,
- because scheduling depends on the length of the next CPU burst of a process rather than rather than **its total length**

# Shortest JoB First (SJF)

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- SJF scheduling chart

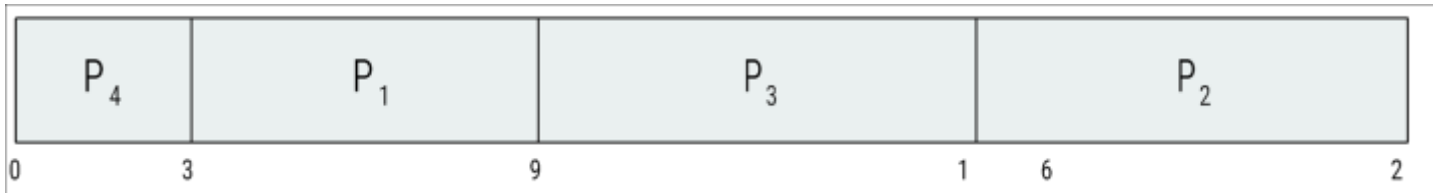



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- *Apply FCFS and compare the average waiting time*

# Shortest JoB First (SJF)- Example -Premptive

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- SJF scheduling chart

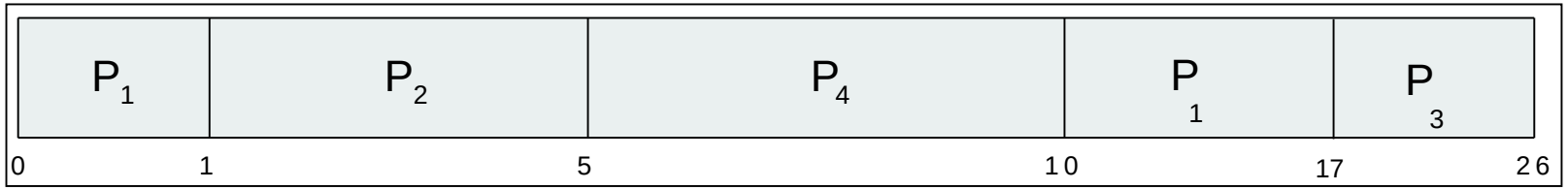



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- *Apply FCFS and compare the average waiting time*

# Shortest JoB First (SJF)



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




*Waiting Time = Total waiting time- No of Milliseconds Process executed – Arrival time*

*P1 = 10-01-00 =9*

*P2 = 1-0-1 = 0*

*P3 = 17-0-2 = 15*

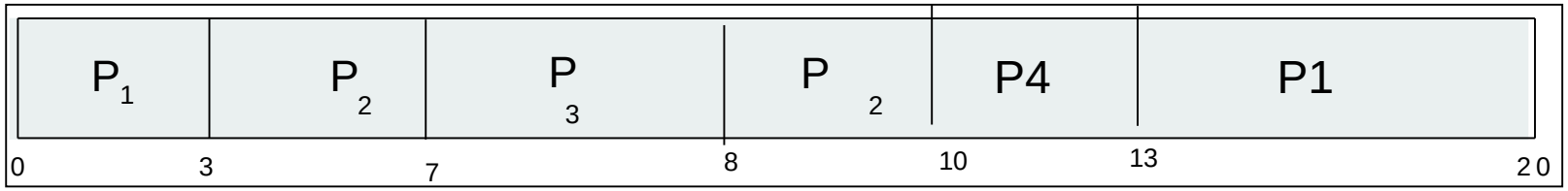
*P4 = 5-0-3 = 2*

*Average waiting time = 26/4 = 6.5ms*

# Shortest JoB First (SJF)

Consider the following process with the arrival time and length of CPU burst given in milliseconds. The scheduling algorithm is **Premptive SJF. Calculate average Turn around time**

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 10 |
| P2 | 3 | 6 |
| P3 | 7 | 1 |
| P4 | 8 | 3 |



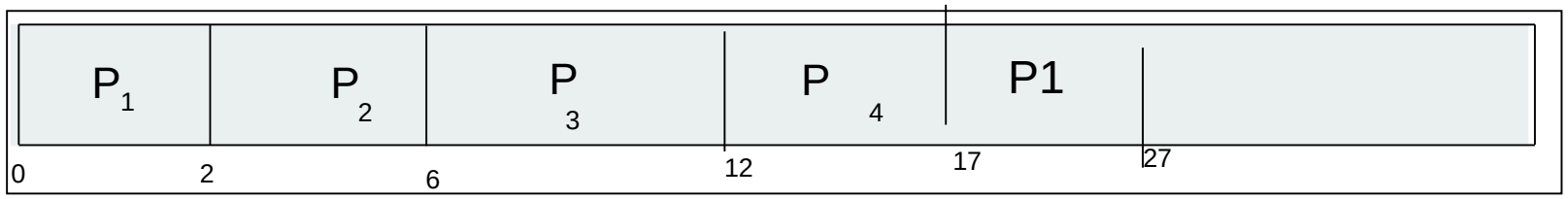

*Turn Around Time = Completion Time – Arrival Time*
*P1 = 20-0 =20*
*P2 = 10-3 = 7*
*P3 = 8-7 = 1*
*P4 = 13-8 = 5*
*Average waiting time = 33/4 = 8.2ms*

*Calculate the Average waiting time*

# Shortest JoB First (SJF)

Consider the following process with the arrival time and length of CPU burst given in milliseconds. The scheduling algorithm is **Premptive SJF. Calculate average Turn around time**

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 12 |
| P2 | 2 | 4 |
| P3 | 3 | 6 |
| P4 | 8 | 5 |




*Calculate the Average waiting time and Turn around time*

# Priority Scheduling

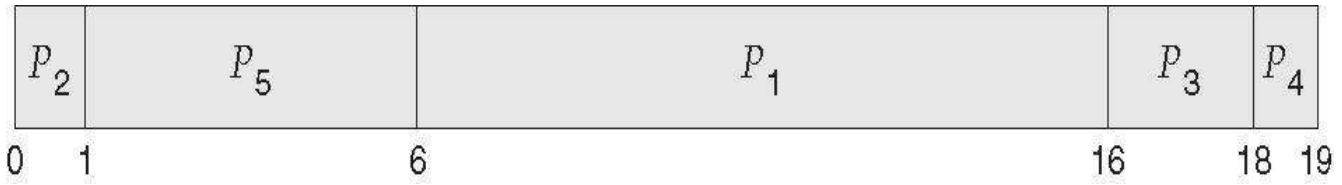
- The SJF algorithm is a special case of the general priority-scheduling algorithm
- A priority is associated with each process, and the CPUis allocated to the process with the highest priority (smallest integer ≡ highest priority).
- Equal-priority processes are scheduled in FCFS order.
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time. (Larger the CPU Burst time, lower the priority and vice versa).
- Priority Scheduling algorithm is Premptive or Non Premptive

# Priority Scheduling

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ⋯, P5, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

- Priority scheduling Gantt Chart



# Priority Scheduling

- Major problem with priority scheduling algorithms is indefinite blocking, or starvation.
- A process that is ready to run but waiting for the CPU can be considered **blocked**
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

# Priority Scheduling Solved Pbm-1(Gate-2017)

| Process ID | Arrival Time | Burst Time | Priority |
|---|---|---|---|
| P1 | 0 | 11 | 2 |
| P2 | 5 | 28 | 0 |
| P3 | 12 | 2 | 3 |
| P4 | 2 | 10 | 1 |
| P5 | 9 | 16 | 4 |

***Consider the Set of the Process and find the average waiting time by applying Premptive Priority Scheduling***

*Ans:*

*Waiting Time = Total waiting time- No of Milliseconds Process executed – Arrival time*

*P1 = 40-02-00 =38*

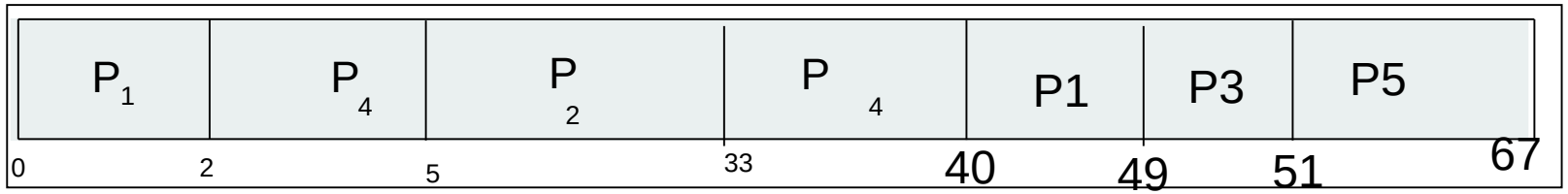
*P2 = 5-0-5= 0*

*P3 = 49-0-12 = 37*

*P4 = 33-3-2= 28*

*P5 = 51-0-9 = 42*

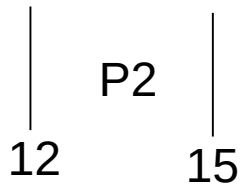
*Average waiting time = 145/5 = 29ms*



# Priority Scheduling Solved Pbm-2

| Process ID | Arrival Time | Burst Time | Priority |
|---|---|---|---|
| P1 | 0 | 4 | 2 |
| P2 | 1 | 3 | 3 |
| P3 | 2 | 1 | 4 |
| P4 | 3 | 5 | 5 |
| P5 | 4 | 2 | 5 |

***Consider the Set of the Process and find the average waiting time by applying Non Premptive Priority Scheduling***
***(Note: Higher Number Represents Higher Priority )***



# Priority Scheduling Solved Pbm-2

| Process ID | Arrival Time | Burst Time | Priority |
|---|---|---|---|
| P1 | 0 | 4 | 2 |
| P2 | 1 | 3 | 3 |
| P3 | 2 | 1 | 4 |
| P4 | 3 | 5 | 5 |
| P5 | 4 | 2 | 5 |

***Consider the Set of the Process and find the average waiting time by applying Non Premptive Priority Scheduling (Note: Higher Number Represents Higher Priority )***

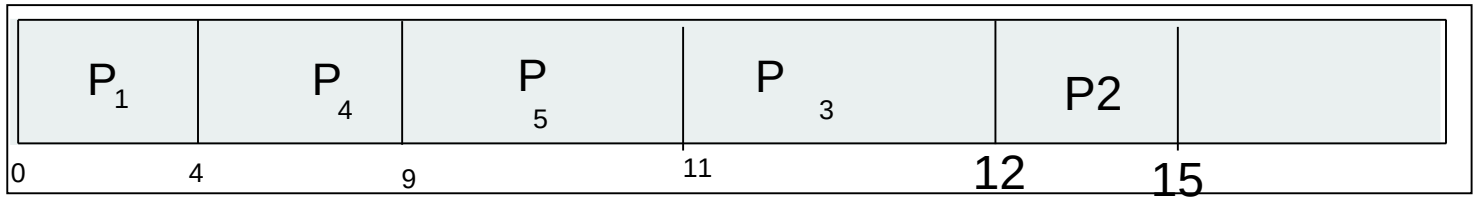
***Ans:***

**Turn Around Time == Completion time – Arrival Time**

**P1= 4-0 = 4 , P2=15-1 = 14, P3 = 12-2 = 10, P4 = 9-3 = 6, P5 = 11-4 = 7**

**Waiting Time == Turn around time - Burst Time**

**P1 = 4-4 = 0, P2= 14-3 = 11,P3 = 9,P4 = 1, P5 = 5**



# Round Robin (RR)

**The round-robin (RR) scheduling algorithm is designed especially for time sharing systems.**

**It is similar to FCFS scheduling, but pre-emption is added to enable the system to switch between processes.**

**A small unit of time, called a time quantum or time slice, is defined. (**A time quantum is generally from 10 to 100 milliseconds in length)

**The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.**

**The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.**

**The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.**

**If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue.**

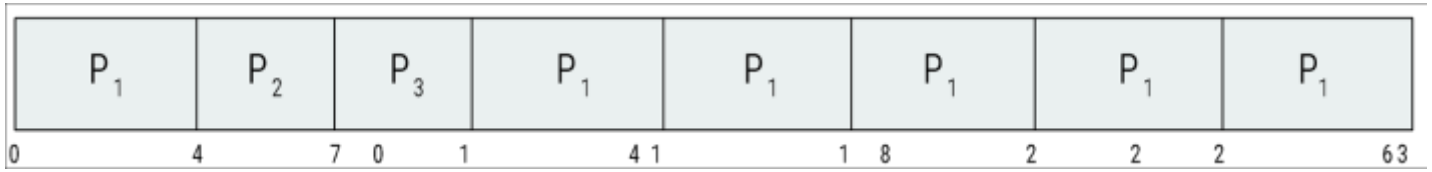
**The CPU scheduler will then select the next process in the ready queue**.

# Example of RR with Time Quantum = 4



| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

The Gantt chart is:




**Turn Around Time == Completion time – Arrival Time**

**P1= 30-0 = 30, P2 = 7,P3 = 10**

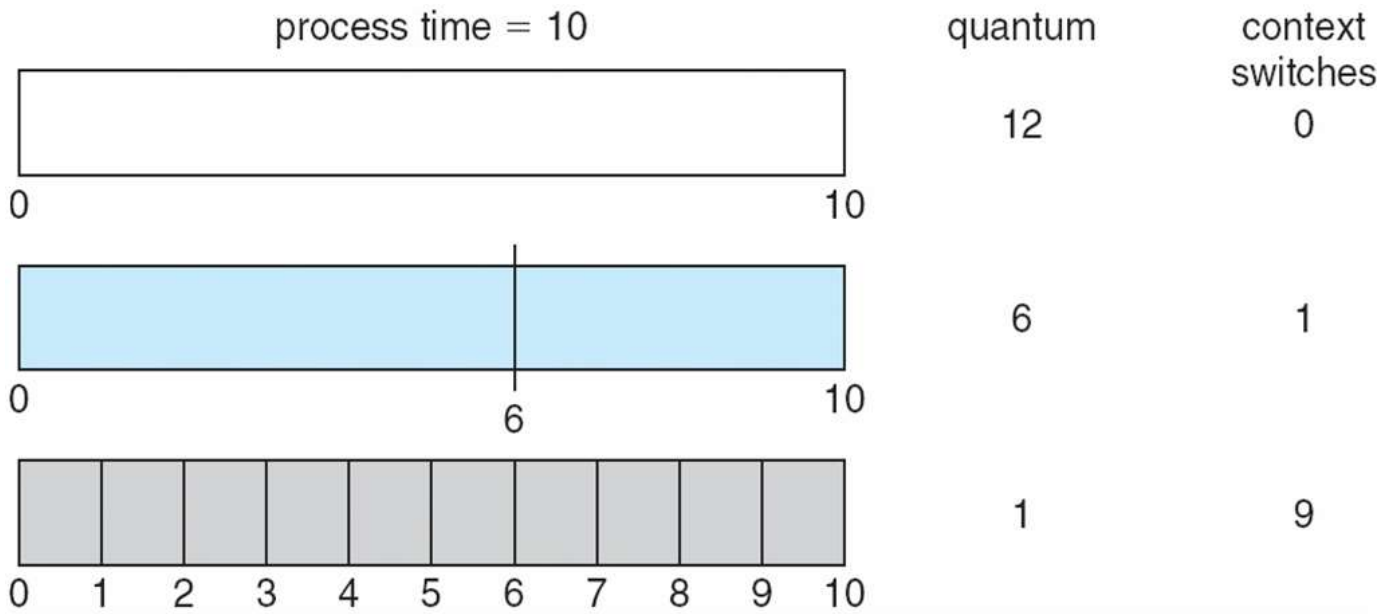
**Waiting Time = Turn Around time – Burst Time**

**P1= 6 P2 =4 P3 = 7**

# Round Robin Continued

The performance of the RR algorithm depends heavily **on the size of the time quantum**. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.

If the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches



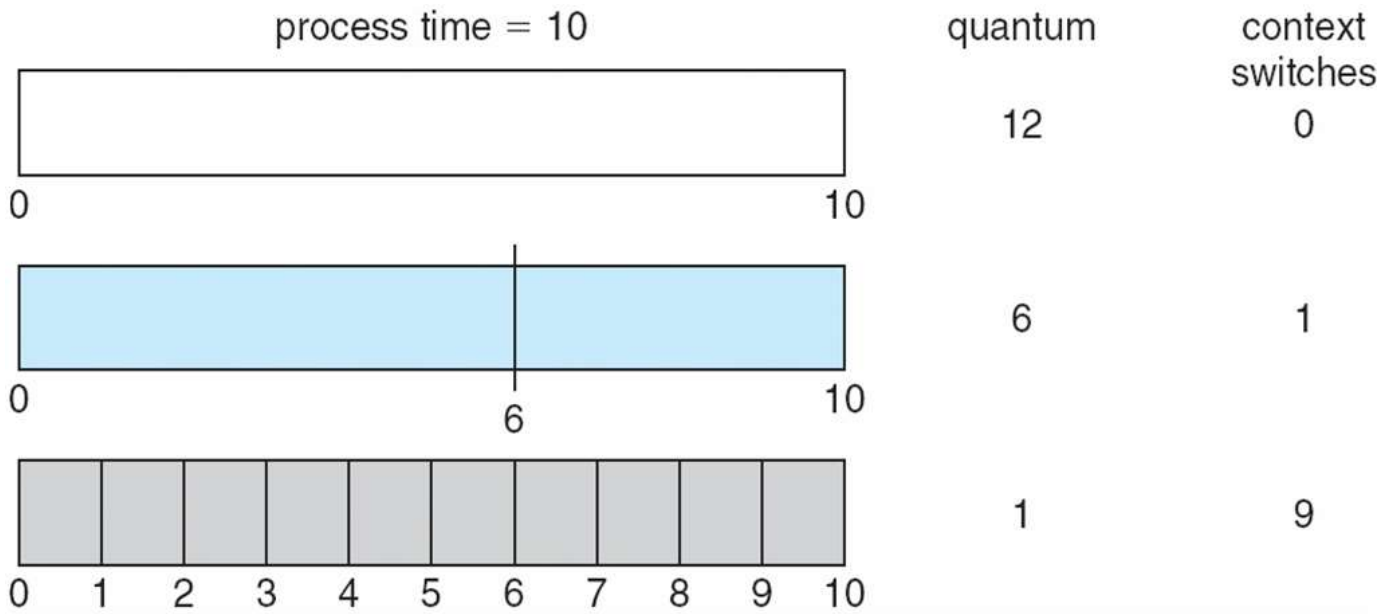

A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

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

# Round Robin Continued

The performance of the RR algorithm depends heavily **on the size of the time quantum**. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.

If the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches




A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

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

# Round Robin Scheduling - Problem

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 1 |
| P4 | 3 | 2 |
| P5 | 4 | 3 |

***Consider the Set of the Process and find the average waiting time and turn around time by applying Round Robin Scheduling, consider the time quantum of 2ms***

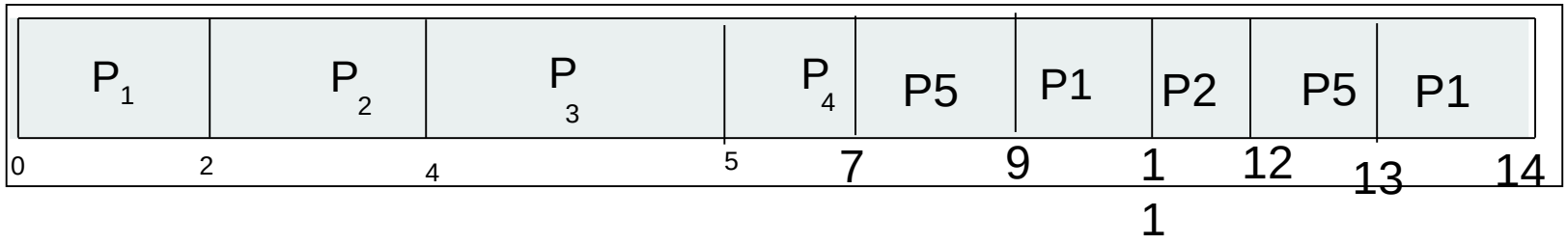
***Ans:***

**Turn Around Time == Completion time – Arrival Time**

**P1= 14-0 = 14 , P2=12-1 = 13, P3 = 5-2 = 3, P4 = 7-3 = 4, P5 = 13-4 = 9**

**Waiting Time == Turn around time - Burst Time**

**P1 = 9 , P2= 10,P3 = 2,P4 = 2, P5 = 6**



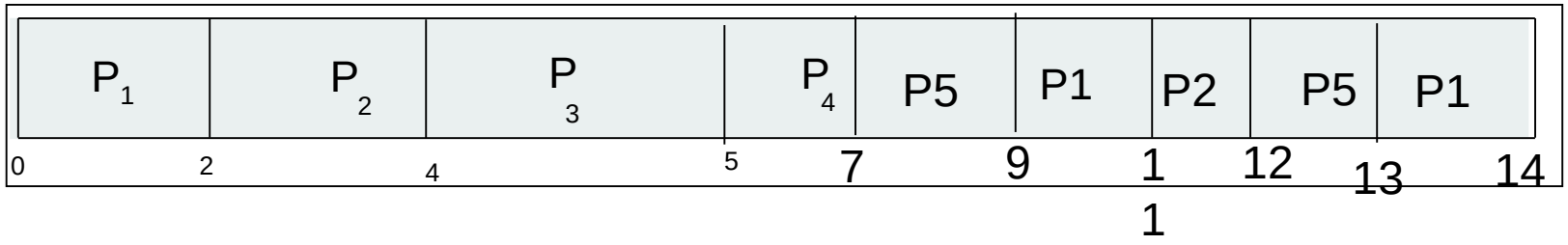
# Round Robin Scheduling – Problem - 2-GATE 2019

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 1 |
| P4 | 3 | 2 |
| P5 | 4 | 3 |

***Consider the Set of the Process and find the average waiting time and turn around time by applying Round Robin Scheduling, consider the time quantum of 2ms***

*Ans:*

*Just Check whether the gannt chart given below is correct or wrong*



# Round Robin Scheduling – Problem - 2-GATE 2019

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 1 |
| P4 | 3 | 2 |
| P5 | 4 | 3 |

***Consider the Set of the Process and find the average waiting time and turn around time by applying Round Robin Scheduling, consider the time quantum of 2ms***

*Ans:*

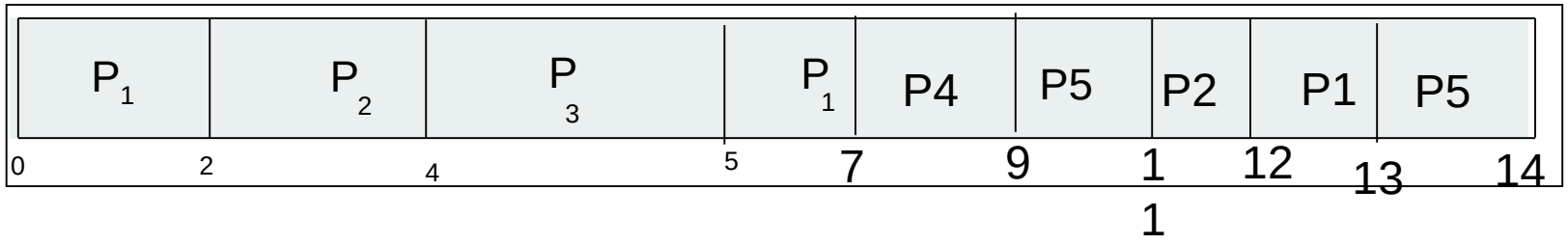
*The gannt chart given in the last slide is wrong, the correct one is below*

**Turn Around Time == Completion time – Arrival Time**

**P1= 13-0 = 13 , P2=12-1 = 11, P3 = 5-2 = 3, P4 = 9-3 = 6, P5 = 14-4 = 10**

**Waiting Time == Turn around time - Burst Time**

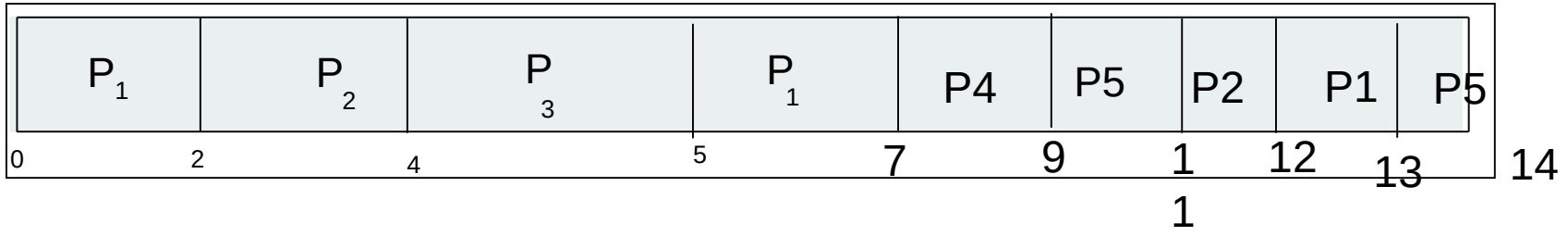
**P1 = 8 , P2= 8,P3 = 2,P4 = 4, P5 = 7**



# Round Robin Scheduling – Problem - 2

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 1 |
| P4 | 3 | 2 |
| P5 | 4 | 3 |

***Oms – P1***
***1ms – P2 has arrived***
***2ms – P3 has arrived & P1 shifted to P2***
***3ms – P4 arrived***
***4ms – P5 has arrived & P2-P3***
***5ms – P3 executes & P3-P1***
***6ms – P1 is executing***
***7ms – P1 –P4***
***8ms = P4 is executing***
***9ms = P4 finishes --- P5***
***10ms = P5 is executing***
***11ms = P5—P2***
***12ms – P2 finishes***
*13ms =P1 finishes*
*14ms= P5 finishes*



# Real-Time CPU Scheduling

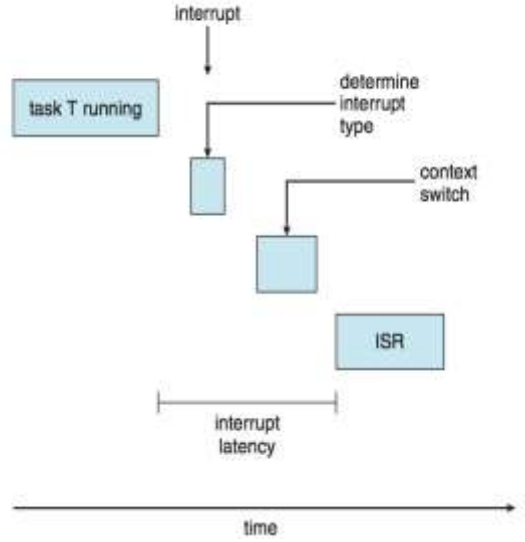
**Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled, They guarantee only that the process will be given preference over noncritical processes

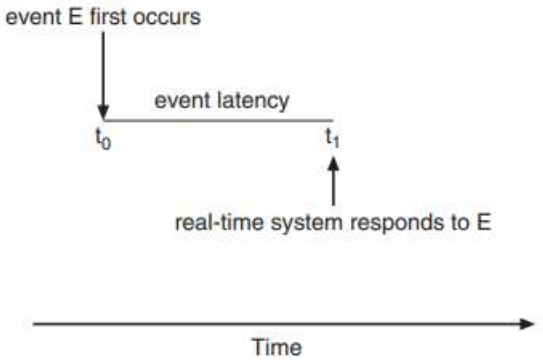
**Hard real-time systems** – task must be serviced by its deadline, service after the deadline has expired is the same as no service at all.

as the amount of time that elapses from when an event occurs to when it is serviced –**Event Latency**

**Two types of latencies affect performance**

1. **Interrupt latency** – refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt(operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention)



# Real-Time CPU Scheduling

**1.Dispatch latency** – time for schedule to take current

process off CPU and switch to another

Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode

1. Release by low-priority process of resources needed by high-priority processes