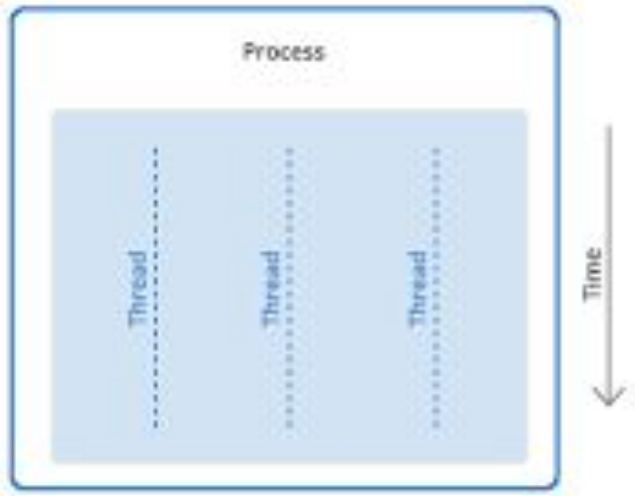
MODULE 2:
PROCESS MANAGEMENT

PROCESS

- The most central concept in any operating system is the process: an abstraction of a running program. Everything else hinges on this concept.
- A process is a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices— to accomplish its task.
- These resources are typically allocated to the process while it is executing.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code.
- All these processes may execute concurrently
- A process is just an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process.

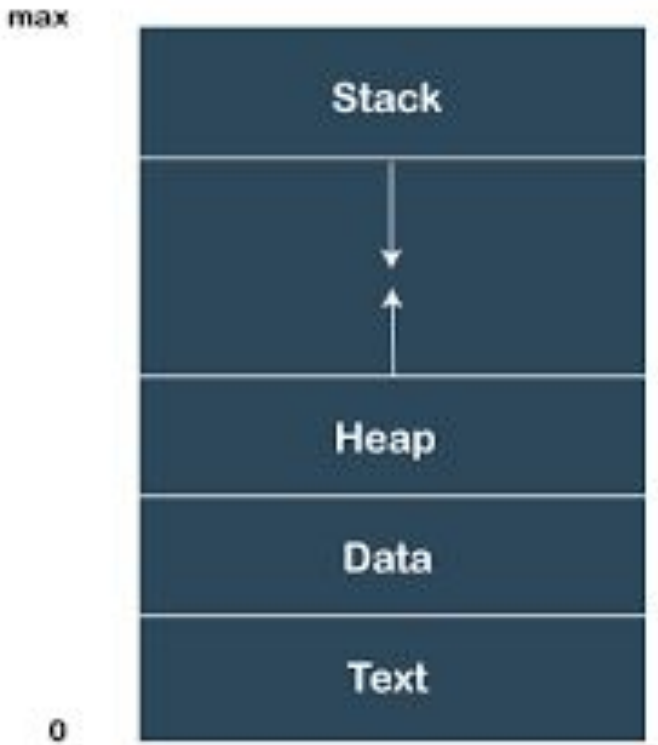
PROCESS & THREAD

- Modern operating systems support processes having multiple threads of control.
- On systems with multiple hardware processing cores, these threads can run in parallel.
- One of the most important aspects of an operating system is how it schedules threads onto available processing cores.
- So threads are units of a Process that is under execution.



PROCESS

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data.
- The following image shows a simplified layout of a process inside main memory
- **Stack**-The process Stack contains the temporary data such as method/function parameters.
- **Heap**-This is dynamically memory.
- **Data**- This section contains the global and static variables.
- **Text**-This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.



Process in the Memory

PROCESS STATES

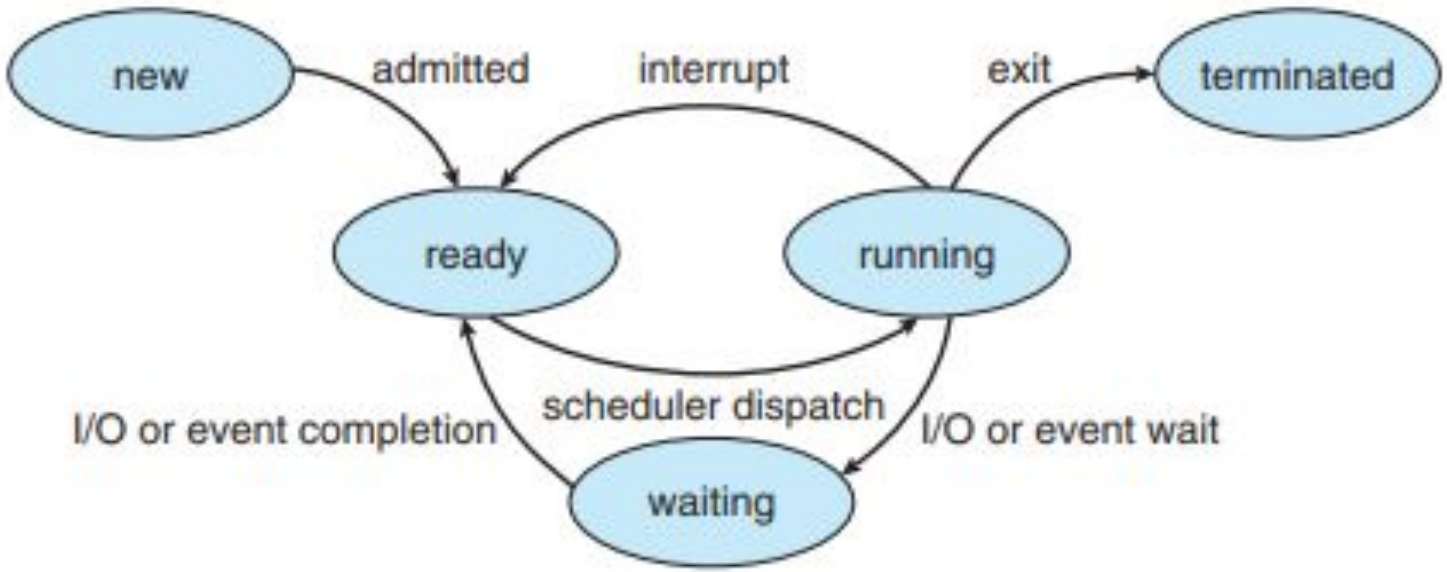


Figure 3.2 Diagram of process state.

PROCESS STATES

- **New**: The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

OPERATIONS ON PROCESS

- **PROCESS CREATION**
- **PROCESS SCHEDULING**
- **PROCESS TERMINATION**

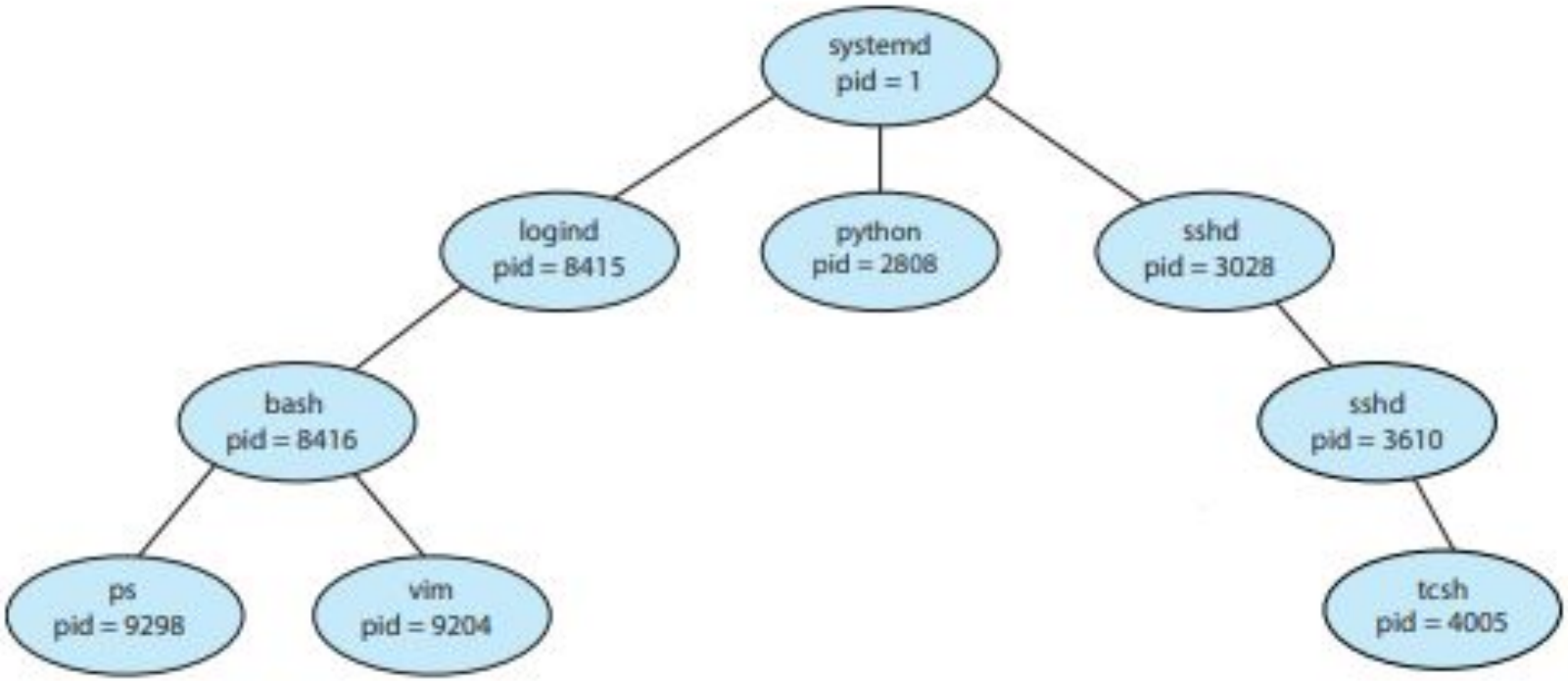
PROCESS CREATION

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

PROCESS CREATION

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

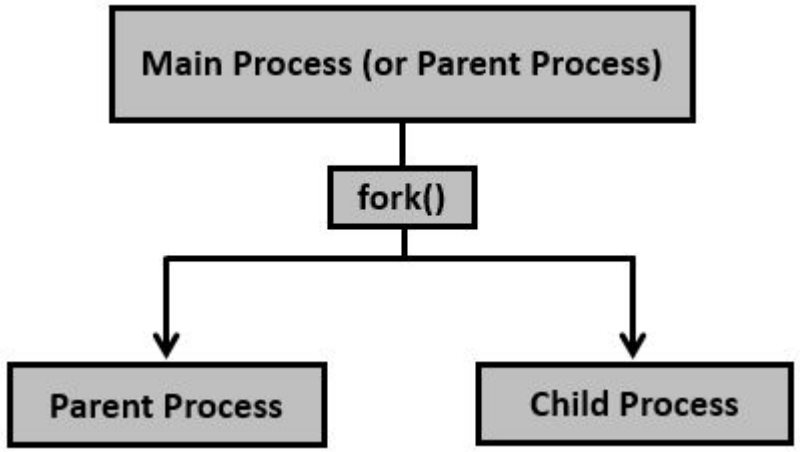
A tree of processes on a LINUX system



PROCESS CREATION

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

  Windows- createProcess()



PROCESS TERMINATION

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent
  - Process' resources are deallocated by operating system
  -

Termination can occur in following cases:

- A process can cause termination of another process via an appropriate system call. Usually such system call is invoked by the parent of the process who is to be terminated. Other users could arbitrarily kill each others processes.
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
  - Some operating system do not allow child to continue if its parent terminates. All children terminated - *cascading termination*

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

PROCESS CONTROL BLOCK

Figure 3.3 Process control block (PCB).

PROCESS CONTROL BLOCK

- Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure 3.3.
- It contains many pieces of information associated with a specific process, including these:

 Process state-The state may be new, ready, running, waiting, halted, and so on.

 Program counter-The counter indicates the address of the next instruction to be executed for this process

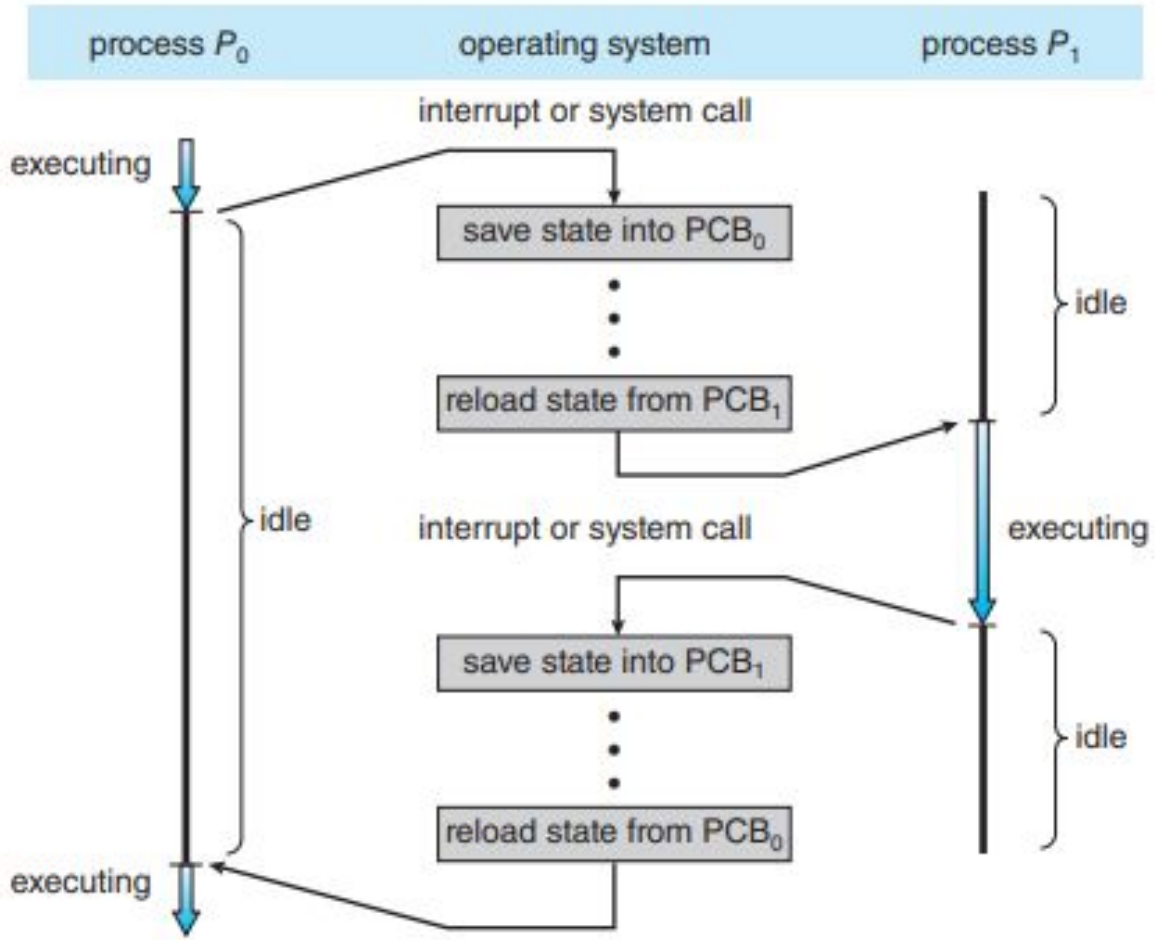
PROCESS CONTROL BLOCK

- CPU registers-The registers vary in number and type & may include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- CPU-scheduling information- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information- This information may include such items as the value of the registers and the page tables, or the segment tables.
- Accounting information-This information includes the amount of CPU and real time used resources, time limits, account numbers, job or process numbers, and so on.
- I/O status information-This information includes the list of I/O devices allocated to the process, a list of open files, and so on

CONTEXT SWITCHING

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process.
- Generically, we perform a state save of the current state of the CPU core, be it in kernel or user mode, and then a state restore to resume operations.
- Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process.
- This task is known as a context switch

CONTEXT SWITCH FROM PROCESS TO PROCESS



THREADS

- A thread is a path of execution within a process.
- A process can contain multiple threads.
- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional 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

Benefits of Multithreads:

- Responsiveness
- Resource Sharing
- Economy
- Stability

THREADS

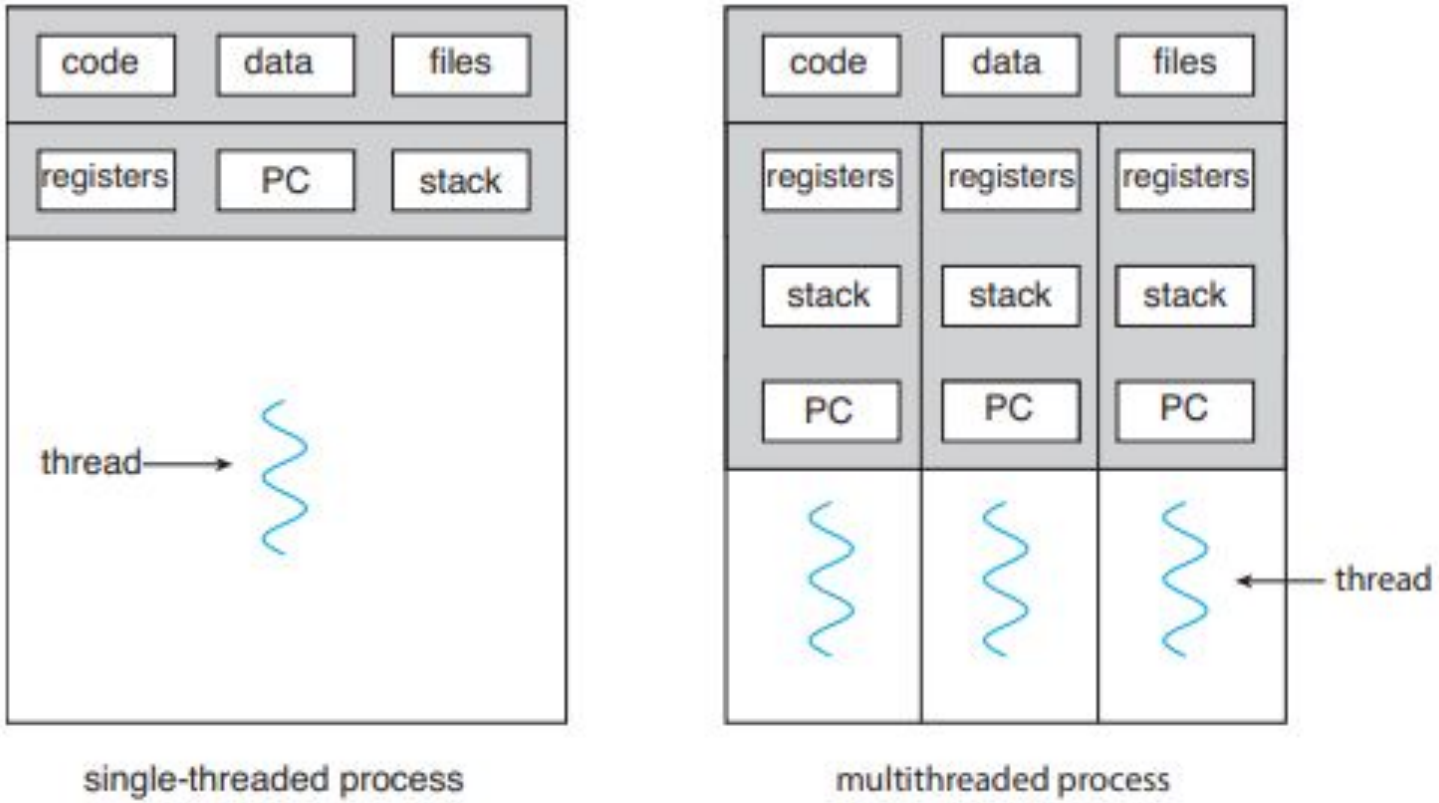


Figure 4.1 Single-threaded and multithreaded processes.

DIFFERENCE PROCESS / THREAD

Process	Thread
Program in execution	It is the part of process
It is heavy weight process	It is light weight process
Process context switch takes more time	Thread context switch takes less time
New process creation, termination takes more time	New Thread creation, termination takes lesse time
Each process executes the same code but has its own memory ad file resources	All thread can share same set of open files, child process
In process based implementation if one process is blocked , no other server process can execute until the first process unblocked	In multithreaded server implementation, if one thread blocked and waiting, second thread in the same process could execute
Multiple redundant process use more resources	Multiple threaded process use fewer resources

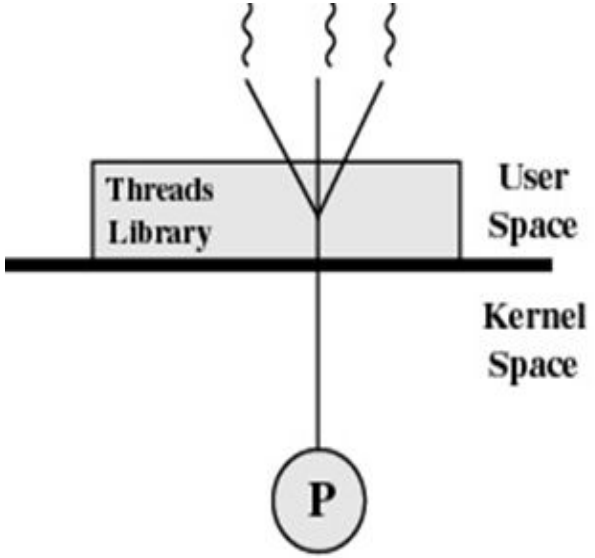
Multithreading: Types of threads

1. User level threads
2. Kernel level threads

Multithreading: Types of threads

User level threads

- Threads are implemented at user level by Thread Library: creates, scheduling and management of threads without kernel involvement.
- Hence fast to create & manage.
- Status of information table is maintained within Thread Library hence better scalability.



User level threads

❖ ADVANTAGES

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.
- Supported above the kernel, via a set of library calls at the user level
- Fast switching among threads: Threads do not need to call OS and cause interrupts to kernel.
- User-level threads does not require modification to operating systems

User level threads

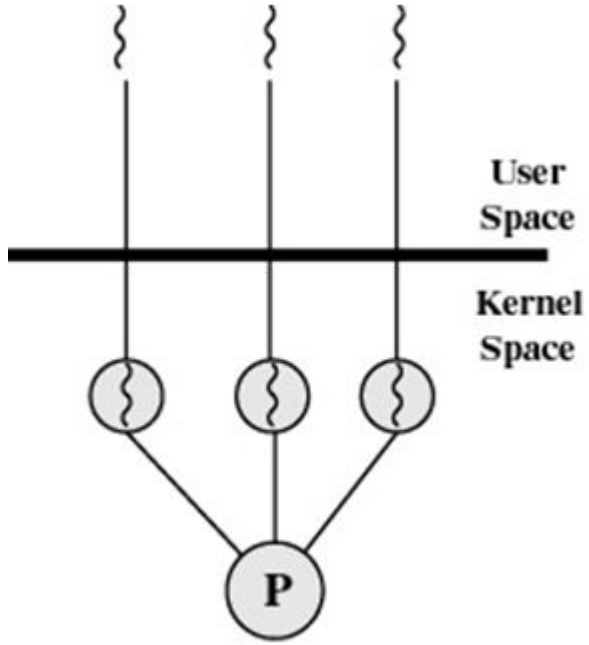
❖ Disadvantages

- When a user level thread executes a blocking system call, not only that thread is blocked, but also all of the threads within the process are blocked.
- There is a lack of coordination between threads and operating system kernel.
- Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within.
- It is up to each thread to relinquish control to other threads
- Multithreaded application cannot take advantage of multiprocessing

Multithreading: Types of threads

Kernel Threads

- Kernel maintains process table and keeps track of all processes.
- Kernel threads are slow & insufficient because it requires a full TCB(Thread Control Block) for each thread to manage and schedule.



Kernel Threads

❖ ADVANTAGES

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can multithreaded.

❖ DISADVANTAGES

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Kernel Threads

- Disadvantages – The kernel-level threads are slow and inefficient .
- For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes.
- It require a full thread control block (TCB) for each thread to maintain information about threads.
- As a result there is significant overhead and increased in kernel complexity

Difference between User Level & Kernel Level Thread

User Level thread	Kernel Level Thread
User level threads are faster to create and manage	Kernel level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads
User level thread is generic and can run on any operating system	Kernel level thread is specific to the operating system
Multi-threaded application cannot take advantage of multiprocessing	Kernel routines themselves can be multithreaded.

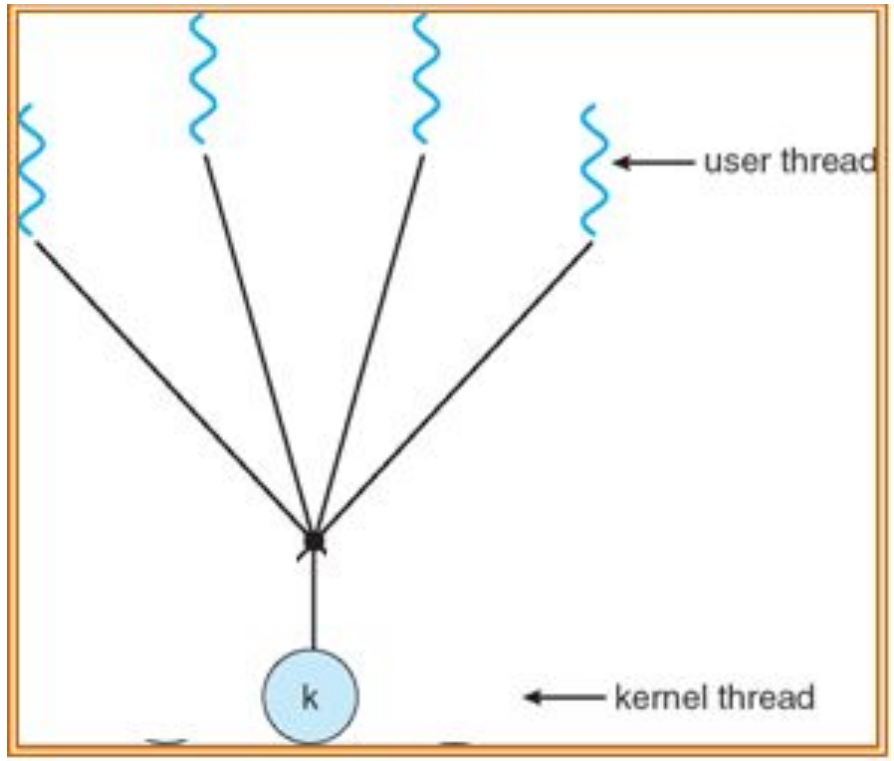
Multithreading: Thread Models

- Many-to-One Model
- One-to-one Model
- Many-to-Many Model

Multithreading: Many-to-One Model

Many-to-One: maps many user level threads into one kernel level thread.

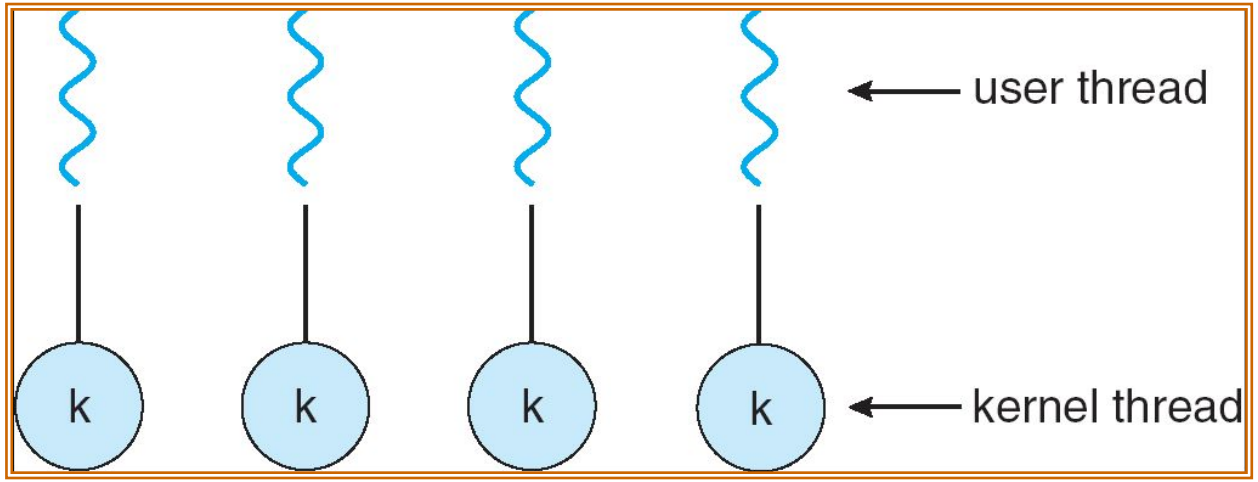
- Thread management is done by thread library in the user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call
- Since only one thread can access kernel at a time, multiple threads cannot run concurrently and thus cannot make use of multiprocessors.



Multithreading: One-to-One Model

One-to-One: Maps each user level thread to a kernel level thread.

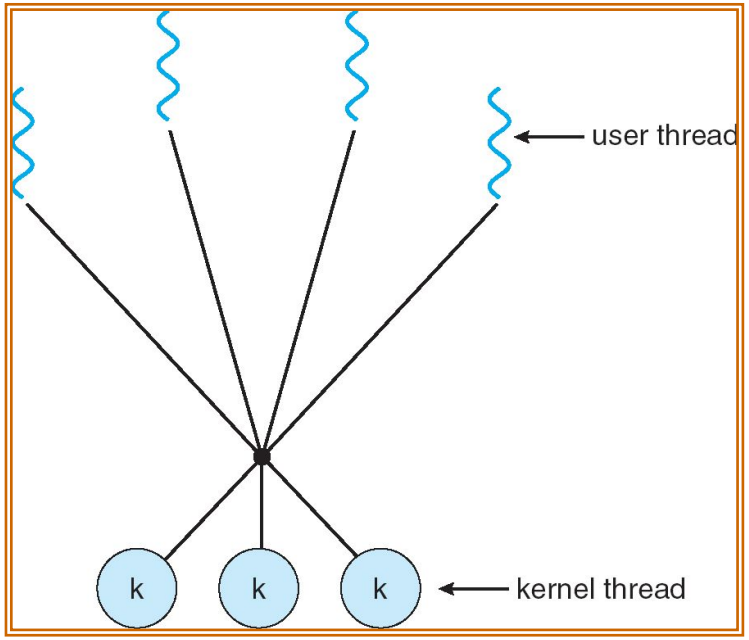
- Provides more concurrency than many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Also allows multiple threads to run in parallel on multiple processors.
- Creating a user level thread results in creating a kernel thread.
- More overhead and allows parallelism.
- Because of the overhead most implementations limit the number of kernel threads created



Multithreading: Many-to-Many Model

Many-to-Many: Multiplexes many user level threads to a smaller or equal number of kernel threads

- Has the advantages of both the many-to-one and one-to-one model.
- The number of kernel threads maybe specific to either a particular application or a particular machine.
- Developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also when a user threads performs a blocking system calls, the kernel can schedule another thread for execution.



Process Scheduling

- The objective of multiprogramming is to have some process running at all times.
- To maximize the CPU utilization scheduling is done among various processes.

 Job queue – set of all processes in the system

 Ready queue – set of all processes residing in main memory, ready and waiting to execute .

o A ready queue header contains pointer to first & last PCBs in the linked list.

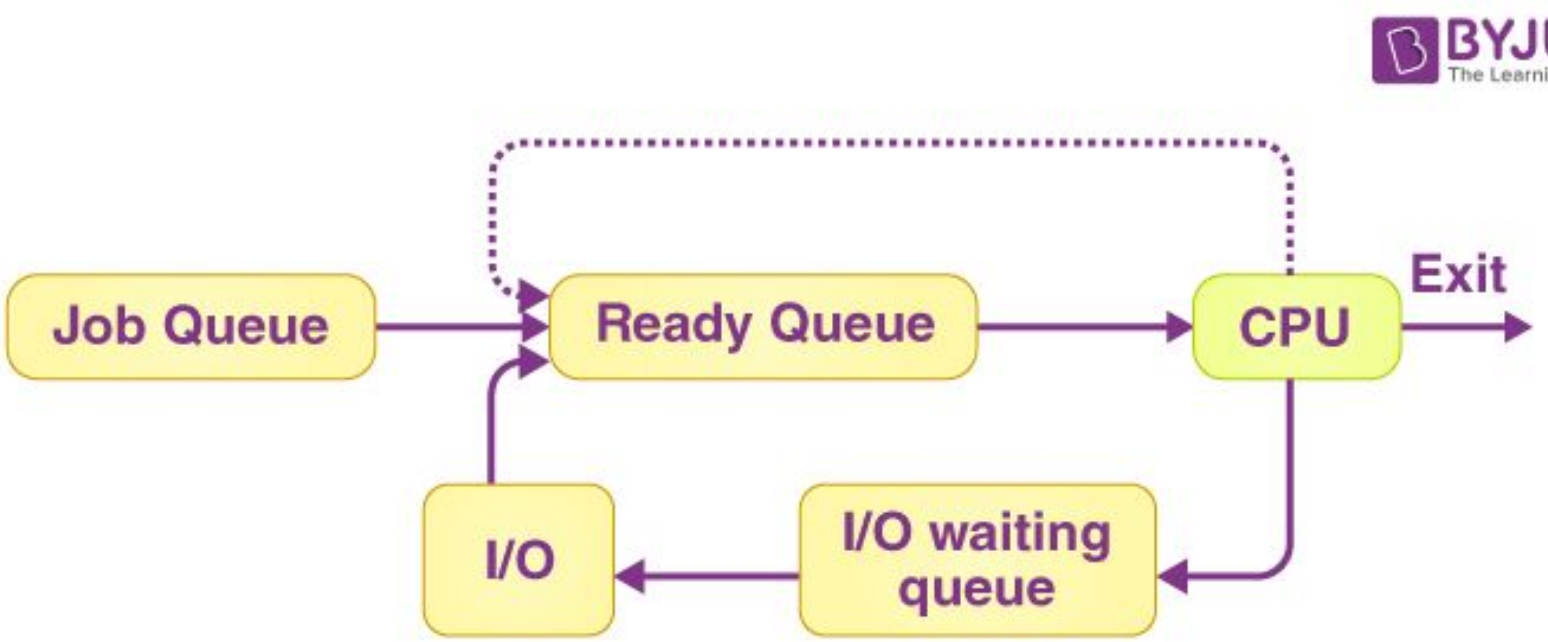
o Each PCB has pointer, points to next process in queue.

 Device queues – set of processes waiting for an I/O device

- Processes migrate among the various queues

Process Scheduling

Queuing representation of process scheduling



TYPES OF SCHEDULERS

- In general, (job) scheduling is performed in three stages: short-, medium-, and long-term. The activity frequency of these stages are implied by their names.
- Long-term (job) scheduling is done when a new process is created. It initiates processes and so controls the degree of multi-programming (number of processes in memory).
- Medium-term scheduling involves suspending or resuming processes by swapping (rolling) them out of or into memory.
- Short-term (process or CPU) scheduling occurs most frequently and decides which process to execute next.

TYPES OF SCHEDULERS

- **Long-term scheduler (or job scheduler)**

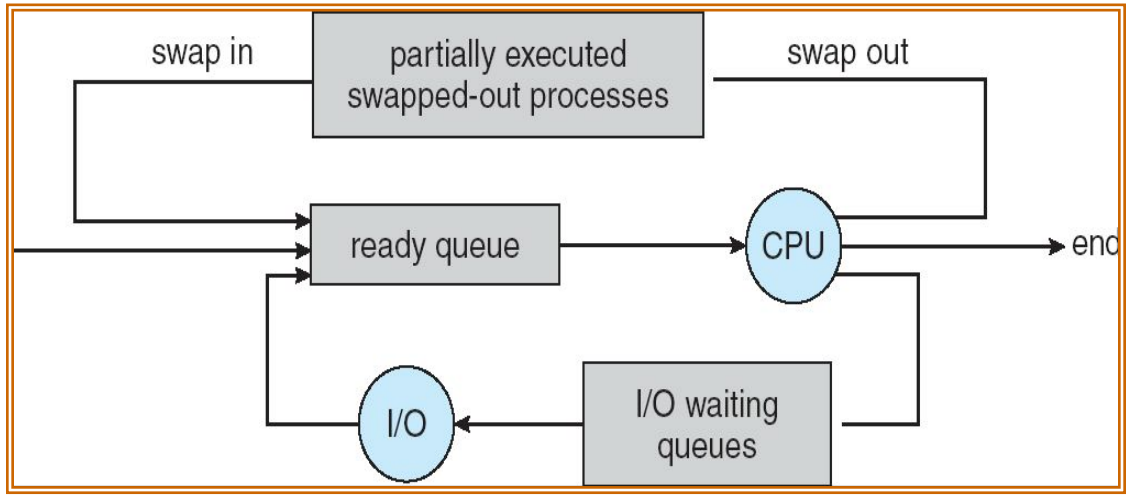
  ✔ selects which processes from secondary storage should be brought into the ready queue.

- **Short-term scheduler (or CPU scheduler)**

  ✔ selects which process should be executed next(from ready queue) and allocates CPU.

Medium term scheduler

- It removes processes from memory and swaps to disk to reduce the degree of multiprogramming to increase throughput.



Long term scheduler	Short term scheduler	Medium term scheduler
Selects the process from the disk and loads them into main memory for execution, puts in ready queue	Chooses the process from ready queue and assigns it to CPU	Swaps in and out the process from memory
Speed is less	Speed is fast	Speed is moderate
Transition of process from New to Ready	Transition of process from Ready to executing	No process state transtition
Not present in time sharing system	Minimal in Time sharingsystem	Present in Time sharing system
Supply a reasonable mix of jobs, such as I/O bound and CPU bound	Select a new process to allocate to CPU frequently	Process are swapped in and out for balanced process mix

SCHEDULING CRITERIA

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

OPTIMIZATION CRITERIA

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

TYPES OF SCHEDULING

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state.
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).
4. When a process terminates.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
However, there is a choice for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

TYPES OF SCHEDULING

NON-PREEMPTIVE

- Simple to implement.
- No timers, process gets the CPU for as long as desired
- Open to denial-of-service.
- Malicious or buggy process can refuse to yield
- Typically includes an explicit yield system call or similar, plus implicit yields, e.g., performing IO, waiting
-

PREEMPTIVE

- Solves denial-of-service.
- OS can simply preempt long-running process
- More complex to implement: Timer management, concurrency issues

Types of Scheduling algorithm

- First Come First Server(FCFS)
- Shortest Job First (SJF)
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

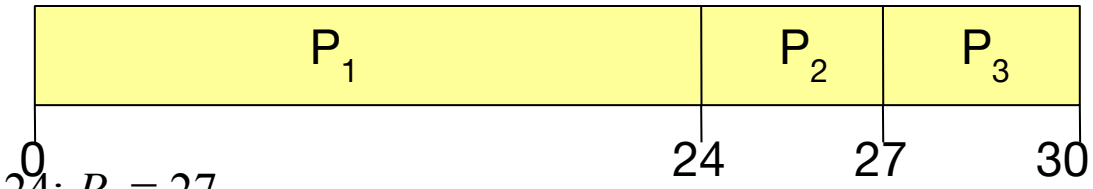
FCFS Scheduling

- Case #1 is an example of the **convoy effect;** all the other processes wait for one long-running process to finish using the CPU
  - This problem results in lower CPU and device utilization; Case #2 shows that higher utilization might be possible if the short processes were allowed to run first
- The FCFS scheduling algorithm is **non-preemptive**
  - Once the CPU has been allocated to a process, that process keeps the CPU until it releases it either by terminating or by requesting I/O
  - It is a troublesome algorithm for time-sharing systems

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- With FCFS, the process that requests the CPU first is allocated the CPU first
- Case #1: Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- Gantt Chart :

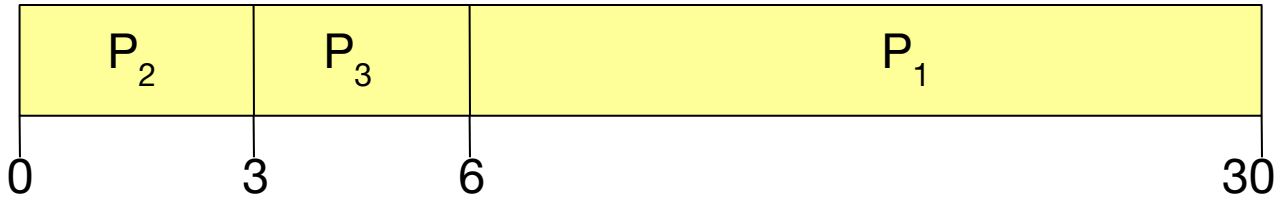


- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17
- Average turn-around time: (24 + 27 + 30)/3 = 27

FCFS Scheduling

- **Case #2: Suppose that the processes arrive in the order: P_2 , P_3 , P_1**

Gantt chart:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3 (Much better than Case #1)
- Average turn-around time: (3 + 6 + 30)/3 = 13

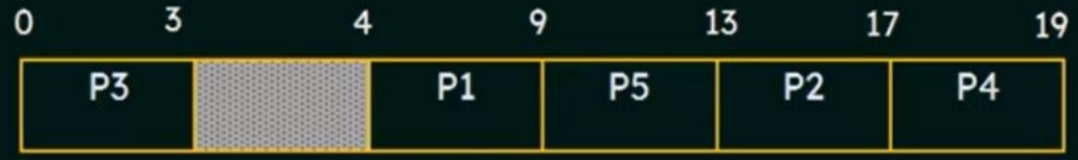
Consider the set of 5 processes whose arrival time and burst time are given below:

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 average turnaround time,
if FCFS Scheduling Algorithm is followed.

Solution:

Gantt Chart:



The shaded box represents the idle time of CPU

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

Turn Around time = Completion time – Arrival time

Waiting time = Turn Around time – Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	9	9 – 4 = 5	5 – 5 = 0
P2	17	17 – 6 = 11	11 – 4 = 7
P3	3	3 – 0 = 3	3 – 3 = 0
P4	19	19 – 6 = 13	13 – 2 = 11
P5	13	13 – 5 = 8	8 – 4 = 4

Now,

Average Turn Around time = (5 + 11 + 3 + 13 + 8) / 5
= 40 / 5
= 8 units

Average waiting time = (0 + 7 + 0 + 11 + 4) / 5
= 22 / 5
= 4.4 units

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **nonpreemptive –** once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **preemptive –** if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Shortest-Job-First (SJF) Scheduling

Scheduling Algorithms

(Shortest-Job-First Scheduling)

- This algorithm associates with each process the length of the process's next CPU burst.
- 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.

The SJF algorithm can be either preemptive or nonpreemptive

A 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 its total length.

NON PRE-EMPTIVE SJF

SJF (Shortest Job first) Scheduling

Criteria- Burst time
mode - Non Preemptive

Process	AT	BT
P_1	0	4
P_2	1	3
P_3	2	1
P_4	3	2
P_5	4	6

Gantt Chart:

Waiting Time:-

P_1 = 0-0 = 0ms
P_2 = 7-1 = 6ms
P_3 = 4-2 = 2ms
P_4 = 5-3 = 2ms
P_5 = 10-4 = 6ms

TWT = 16ms

AWT = $\frac{16}{5}$ = 3.2ms

Turn around time:-

P_1 = 4-0 = 4ms
P_2 = 10-1 = 9ms
P_3 = 5-2 = 3ms
P_4 = 7-3 = 4ms
P_5 = 16-4 = 12ms

TAT = 32ms

ATAT = $\frac{32}{5}$ = 6.4ms

Example 1: Preemptive SJF (Shortest-remaining-time-first)

Example of SJF Scheduling (Premptive)

Consider the following four processes, with the length of the CPU burst given in milliseconds and the processes arrive at the ready queue at the times shown:

Process ID	Arrival Time	Burst Time
P1	0	~~8~~ 7
P2	1	4
P3	2	9
P4	3	5

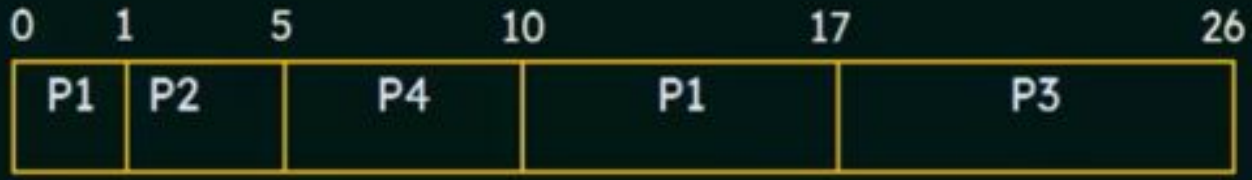
Waiting Time for P1 = (10 – 1 - 0) = 9 ms

Waiting Time for P2 = (1 – 0 - 1) = 0 ms

Waiting Time for P3 = (17 – 0 - 2) = 15 ms

Waiting Time for P4 = (5 – 0 - 3) = 2 ms

Gantt Chart:



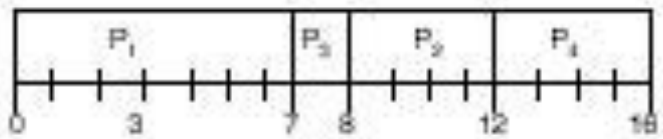
Average Waiting Time

= (9 + 0 + 15 + 2)/4 = 6.5 ms

Waiting Time = Total waiting Time – No. of milliseconds Process executed – Arrival Time

Example of Non-Preemptive SJF

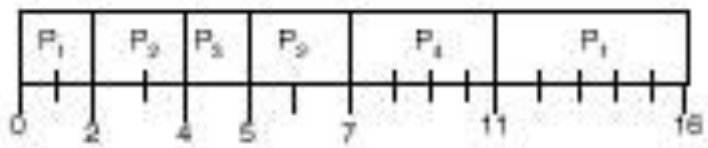
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



average waiting time = (0 + 6 + 3 + 7)/4 = 4

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



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

Practice Sum for Non Preemptive SJF

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	4
P2	1	3
P3	2	1
P4	3	2
P5	4	5

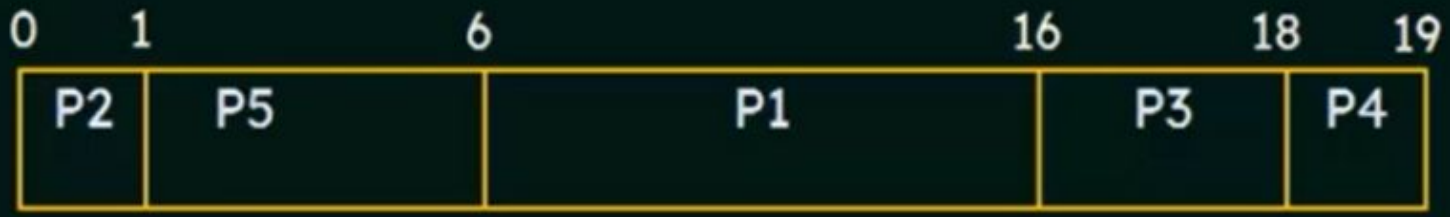
PRIORITY SCHEDULING

- A priority number (integer) is associated with each process.
- Priority scheduling is a form of preemptive scheduling where priority is the basis of preemption.
- It can be non-preemptive also. Means the newly arrived process will be kept at head of the ready queue
- (smallest integer ≡ highest priority)
- **Problem ≡ Starvation/Indefinite Blocking** – low priority processes may never execute.
- **Solution ≡ Aging** – as time progresses increase the priority of the process.

Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds:

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using Priority Scheduling, we would schedule these processes according to the following Gantt Chart:



Waiting Time for P1 = 6 ms

Waiting Time for P2 = 0 ms

Waiting Time for P3 = 16 ms

Waiting Time for P4 = 18 ms

Waiting Time for P5 = 1 ms

Average Waiting Time

= (6 + 0 + 16 + 18 + 1) / 5

= 41 / 5 ms

= 8.2 ms

Priority Scheduling

Solved Problem -1

GATE 2017

Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

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

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is_____.

Process ID	Arrival Time	Burst Time	Priority
P1	0	~~11~~ ~~9~~ -	2
P2	5	~~28~~	0
P3	12	~~2~~ -	3
P4	2	~~10~~ ~~7~~ -	1
P5	9	~~16~~	4

Waiting Time for P1 = (40 – 2 - 0) = 38 ms

Waiting Time for P2 = (5 – 0 - 5) = 0 ms

Waiting Time for P3 = (49 – 0 - 12) = 37 ms

Waiting Time for P4 = (33 – 3 - 2) = 28 ms

Waiting Time for P5 = (51 – 0 - 9) = 42 ms

Average Waiting Time

= (38 + 0 + 37 + 28 + 42) / 5

= 145/5 ms

= 29 ms

Solution:

Gantt Chart:

Waiting Time = Total waiting Time – No. of milliseconds Process executed – Arrival Time

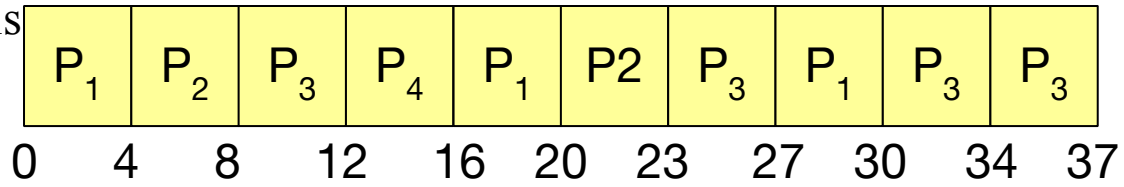
ROUND ROBIN (RR)

- 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, No process waits more than $(n\text{-}1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large w.r.t. context switch, otherwise overhead is too high.
- One rule of thumb is that 80% of the CPU bursts should be shorter than the time quantum

Example of RR with Time Quantum = 4 ms

Process	Burst Time
P_1	11
P_2	7
P_3	15
P_4	4

- The Gantt chart is



- Typically, higher average turnaround than SJF, but better *response time*
- Average turn-around time = (30-0)+(23-0)+(37-0)+(16-0)
- =30 + 23 + 37 + 16) / 4 = 26.5
- Average waiting time = [0+(16-4)+(27-20)]+[4+(20-8)]+[8+(23-12)+(30-27)]+[12]
- = (19+4+22+12)/4

  =14.25

Round Robin Scheduling

Solved Problem

(Part - 2)

Consider the set of 5 processes whose arrival time and burst time are given below:

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

If the CPU scheduling policy is Round Robin with time quantum = 2 units, calculate the average waiting time and average turn around time.

Gantt Chart:

Turn Around time = Completion time – Arrival time

Waiting time = Turn Around time – Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

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

Average Turn Around time

= (13 + 11 + 3 + 6 + 10) / 5

= 43 / 5 = **8.6 units**

Average waiting time

= (8 + 8 + 2 + 4 + 7) / 5

= 29 / 5 = **5.8 units**

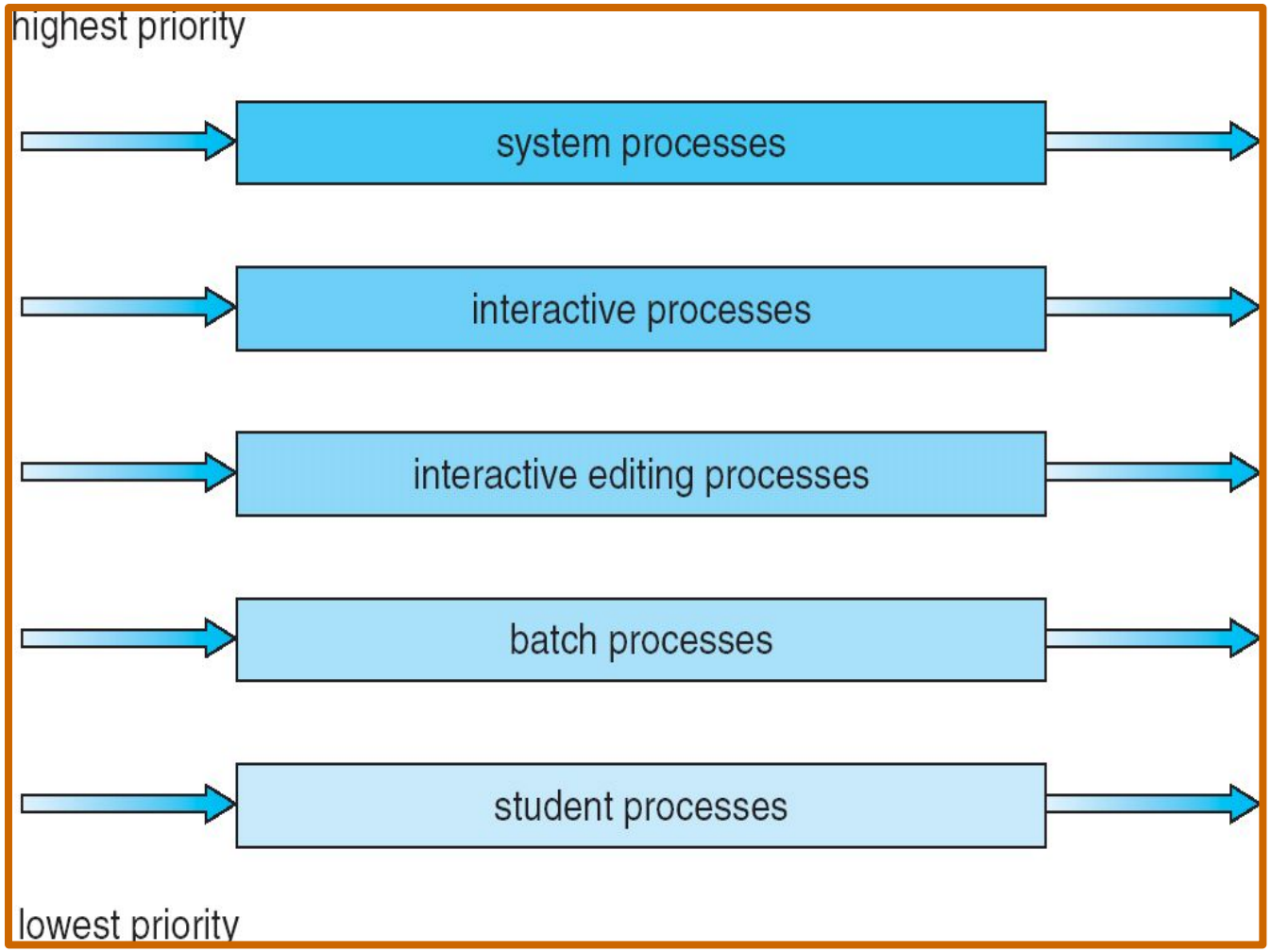
MULTI-LEVEL QUEUE SCHEDULING

- Multi-level queue scheduling is used when processes can be classified into groups
- For example, **foreground** (interactive) processes and **background** (batch) processes
  - The two types of processes have different response-time requirements and so may have different scheduling needs
  - Also, foreground processes may have priority (externally defined) over background processes
- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues
- The processes are permanently assigned to one queue, generally based on some property of the process such as memory size, process priority, or process type

MULTI-LEVEL QUEUE SCHEDULING

- Each queue has its own scheduling algorithm
  - The foreground queue might be scheduled using an RR algorithm
  - The background queue might be scheduled using an FCFS algorithm
- In addition, there needs to be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling
  - The foreground queue may have absolute priority over the background queue
- One example of a multi-level queue are the five queues shown below
- Each queue has absolute priority over lower priority queues
- For example, no process in the batch queue can run unless the queues above it are empty
- However, this can result in starvation for the processes in the lower priority queues

MULTI-LEVEL QUEUE SCHEDULING

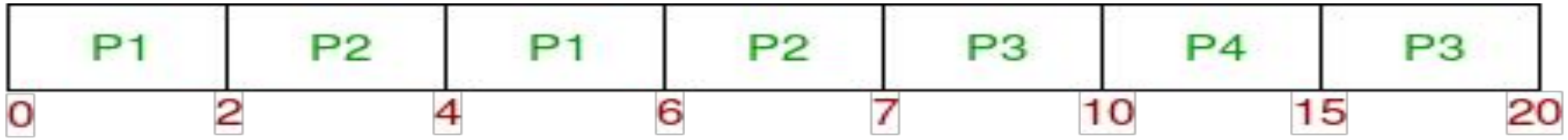


MULTI-LEVEL QUEUE SCHEDULING

- Another possibility is to time slice among the queues
- Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes
    - The foreground queue can be given 80% of the CPU time for RR scheduling
    - The background queue can be given 20% of the CPU time for FCFS scheduling

Queue 1-Round Robin Scheduling Time Quantum=2ms
Queue 2- FCFS Scheduling
Q1>Q2

PROCESS	QUEUE	AT	BT	CT	TAT	WT
P1	1	0	4	6	6	2
P2	1	0	3	7	7	4
P3	2	0	8	20	20	12
P4	1	10	5	15	5	0

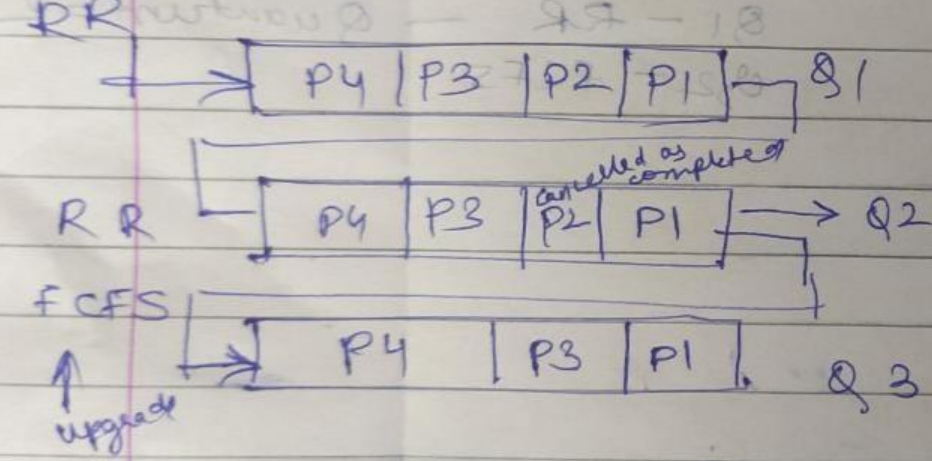


MULTILEVEL FEEDBACK QUEUE SCHEDULING

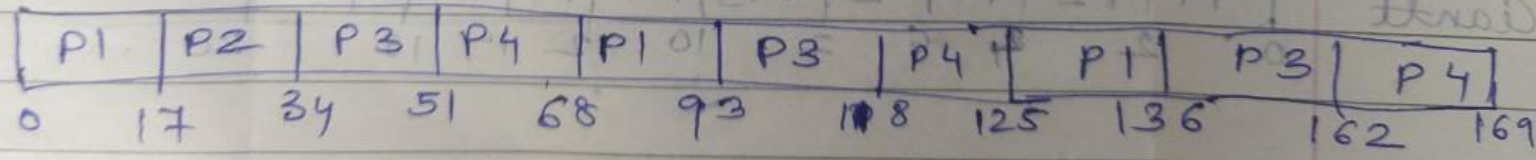
- In multilevel feedback queue scheduling, a process can move between the various queues; aging can be implemented this way
- A multilevel-feedback-queue scheduler is defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to promote a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

Process	B.T	A.T	C.T	TAT	WT
P1	53	0			
P2	17	0			
P3	68	0			
P4	24	0			

Q1 = RR (q = 17)
Q2 = RR (q = 25)
Q3 = FCFS

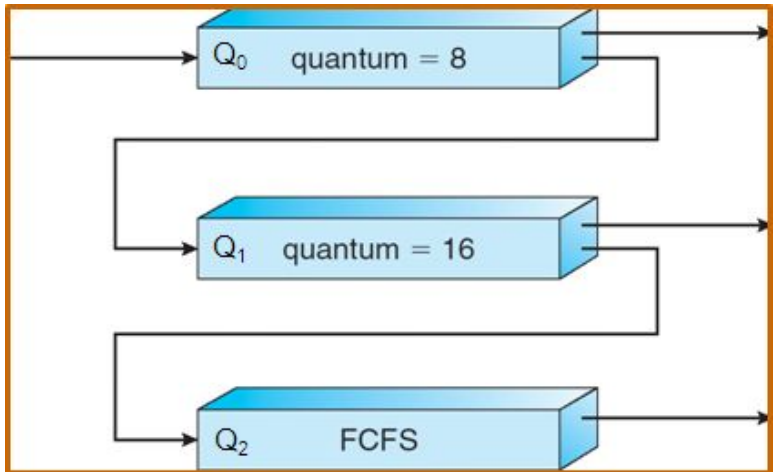


Gantt chart



EXAMPLE OF MULTILEVEL FEEDBACK QUEUE

- Scheduling
  - A new job enters queue Q_0 *(RR)* and is placed at the end. When it gains the CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the end of queue Q_1.
  - A Q_1 (RR) job receives 16 milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 (FCFS).



i) FCFS

ii) Pre-emptive and non pre-emptive SJF

iii) Pre-emptive priority

Process	Arrival Time	Burst Time	Priority
P1	0	8	3
P2	1	1	1
P3	2	2	2
P4	3	3	3
P5	4	6	4

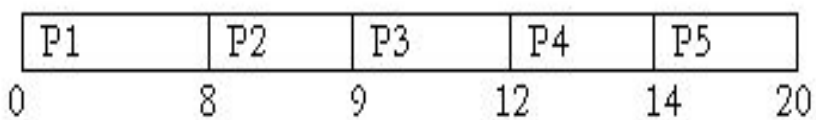
FCFS

Process	Arrival Time	Burst Time	Priority	Waiting Time	Turn Around Time
P1	0	8	3	0	8
P2	1	1	1	7	8
P3	2	3	2	7	10
P4	3	2	3	9	11
P5	4	6	4	10	16

Average WT: Total Waiting Time / No of processes: 0+7+7+9+10/5 = 6.6

Average TAT: Total Turn Around Time / No of processes: 8+8+10+11+16/5 = 10.6

Gantt Chart:



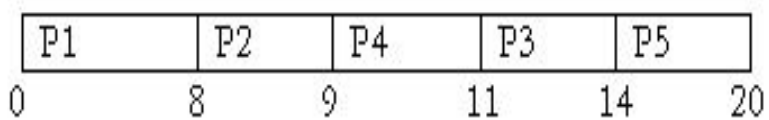
Non Pre-emptive SJF:

Process	Arrival Time	Burst Time	Waiting Time	Turn Around Time
P1	0	8	0	8
P2	1	1	7	8
P3	2	3	9	12
P4	3	2	6	8
P5	4	6	10	16

Average WT: Total Waiting Time / No of processes = 6.4

Average TAT: Total Turn Around Time / No of processes = 10.4

Gantt Chart:



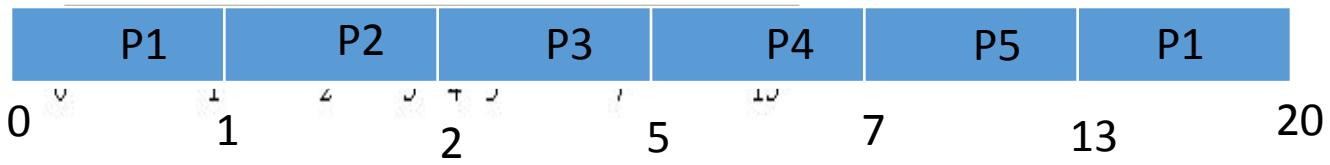
Pre-emptive SJF:

Process	Arrival Time	Burst Time	Waiting Time	Turn Around Time
P1	0	8	12	20
P2	1	1	0	1
P3	2	3	0	3
P4	3	2	2	4
P5	4	6	3	9

Average WT: Total Waiting Time / No of processes = 3.4

Average TAT: Total Turn Around Time / No of processes = 7.4

Gantt chart:



Pre-emptive Priority:

Process	Arrival Time	Burst Time	Waiting Time	Turn Around Time
P1	0	8	3	12
P2	1	1	1	1
P3	2	3	2	3
P4	3	2	3	11
P5	4	6	4	16

Average WT: Total Waiting Time/No of processes = 4.6

Average TAT: Total Turn Around Time/No of processes = 8.6

Gantt Chart :