Concepts of Operating Systems

- Vineela

Session 3: Shell Programming

Lecture:

- Decision loops (if else, test, nested if else, case controls, while…until, for)
- Regular expressions; Arithmetic expressions
- More examples in Shell Programming

Decision loops (if else, test, nested if else, case controls, while…until, for)

1. **If Statement**

Syntax:

```
if [ condition ]
    then
    Commands
fi
```

Example:

```
if [ $age -ge 18 ]

    then

    echo "You are eligible to vote."

    fi
```

2. **if-else Statement**

Syntax:

```
if [ $marks -ge 35 ]

then

        echo "Pass"

else

        echo "Fail"
fi
```

3. if-elif-else Statement

```
if [ $marks -ge 75 ];
        then
                echo "Distinction"
        elif [ $marks -ge 35 ];
                then
                        echo "Pass"
                else
                        echo "Fail"
fi
```

4. Nested if Statement

```
if [ $age -ge 18 ];
        then
                if [ $citizen = "yes" ];
                                then
                                        echo "Eligible to vote"
                                else
                                        echo "Not a citizen"
                fi
                else
                        echo "Underage"
fi
```

test Command **&** case Statement (Switch-like)

- **test Command (Alternative to [ ])**

```
if test $a -gt $b
        then
                echo "$a is greater"
fi
```

★ [condition] is just a shortcut for test condition

- **case Statement (Switch-like)**

Syntax:

```
case word in
        pattern1)
                Statement(s) to be executed if pattern1 matches
        ;;
        pattern2)
                Statement(s) to be executed if pattern2 matches
         ;;
        pattern3)
                Statement(s) to be executed if pattern3 matches
         ;;
        *)
        Default condition to be executed
         ;;
esac
```

Example of Switch-like:

```
read -p "Enter choice (start/stop): " action
case $action in
                start)
                        echo "Starting service...";;
                stop)
                        echo "Stopping service...";;
                *)
                echo "Invalid option";;
 esac
```

Loop - while & for

- **while Loop**

cou

```
nt=1
          while [ $count -le 5 ]
          do
                echo "Count = $count"
                count=$((count + 1))
          done
```

until Loop - Executes **until the condition becomes true**.

```
      n=1
          until [ $n -gt 3 ]
      do
          echo "n = $n"
          n=$((n+1))
      done
```

- **for Loop**

List-based:

```
for name in Alice Bob Charlie

do
      echo "Hi $name"

done
```

- **C-style:**

```
for (( i=1; i<=5; i++ ))
      do
            echo "i = $i"
      done
```

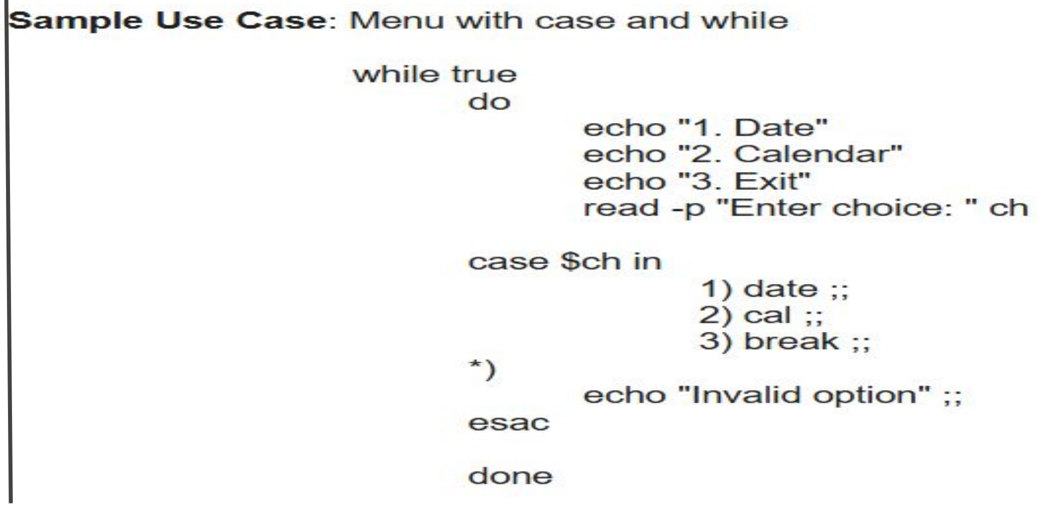
- **Loop Control Statements**

  break - Exits the loop

  continue - Skips current iteration, goes to

  next loop

  exit - Exits the script entirely



Sample Use Case: Menu with case and while

```
while true
      do
            echo "1. Date"
            echo "2. Calendar"
            echo "3. Exit"
            read -p "Enter choice: " ch

      case $ch in
                  1) date ;;
                  2) cal ;;
                  3) break ;;
      *)
            echo "Invalid option" ;;
      esac

      done
```

Regular Expressions

- **Regular expressions (regex)** are patterns **used to match character combinations in text**. In shell scripting, you commonly use them with tools like **grep, sed, and awk** for searching, replacing, and processing text.

Pattern	Meaning	Example	Matches
.	Any single character	c.t	cat, cut, c9t
*	Zero or more of previous char	lo*	l, lo, loo
^	Start of line	^Hi	Hi there
[^]	Not any of the chars inside	[^aeiou]	Any consonant
[]	Any one char inside	[aeiou]	a, e, i, etc.
$	End of line	bye$	goodbye
\	Escape special characters	\.	A literal dot .

- Use grep -E or egrep for extended regex patterns (+, {}, \|, etc.)

Arithmetic Expressions

- **Using let Command**

```
let result=5+3

          echo $result
```

Output: 8

Note: No $ before variables inside let.

- **Using Double Parenthesis ((...)) Syntax**

```
a=10
b=5
((sum = a + b))
          echo "Sum = $sum"
```

Supported Operators → +, - , * , / , % , *= , += , -=

- **Using $((...)) for Inline Evaluation**

```
a=7
b=2
echo "Multiplication: $((a * b))"
```

- **Using expr (older but widely compatible)**

```
a=20
b=6
result=`expr $a / $b`
echo "Division: $result"
```

Note:

❖ You must add spaces between operands and operators.

❖ For **decimal numbers**, always use bc.

❖ Prefer (( )) and $(( )) over expr for modern scripts.

❖ Use let when assigning values directly.

Sessions 4 & 5: Processes

Lecture:

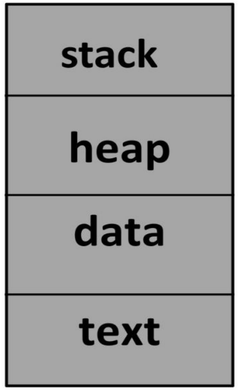
- What is process; preemptive and non-preemptive processes
- Difference between process and thread
- Process management; Process life cycle
- What are schedulers – Short term, Medium term and Long term
- Process scheduling algorithms – FCFS, Shortest Job First, Priority, RR, Queue. Belady's Anomaly
- Examples associated with scheduling algorithms to find turnaround time to find the better performing scheduler
- Process creation using fork; waitpid and exec system calls; Examples on process creation; Parent and child processes
- Orphan and zombie processes

What is Process?

- **Early computers** allowed **only one program** to be executed at a time but **contemporary computer systems** allow **multiple programs** to be loaded into memory and executed concurrently.
- A System consists of a collection of processes - **OS processes** executing system code and **user processes** executing user code
- All these processes execute concurrently with the CPU multiplexed among them.
- A process is a **program which is in execution**
- A process created by the main process is called a **child process.**
- Process management involves various tasks like creation, **scheduling**, termination of processes and a **dead lock**
- For a program to be executed, it must be mapped to absolute addresses and loaded into memory.
- As the program executes, it accesses program instructions and data from memory by generating these absolute addresses.
- Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed

Process Architecture

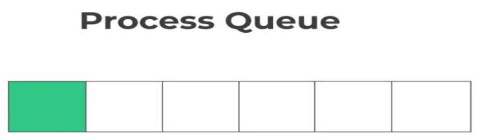
- **Stack:** The Stack stores temporary data like function parameters, returns addresses, and local variables.
- **Heap**: Allocates memory, which may be processed during run time.
- **Data:** It contains the global as well as static variable.
- **Text:** Text Section includes the current activity, which is represented by the value of the **Program Counter** (register within a CPU that holds the memory address of the next instruction to be executed)



Process Life Cycle

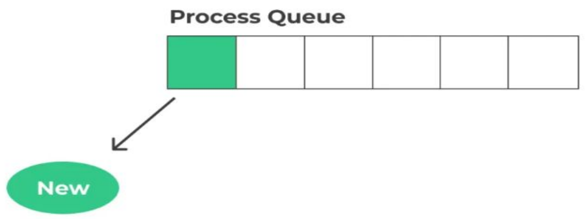
- **New** → When a process is started/created first, it is in this **New** state
- **Ready** → The process may enter this state after starting or while running, but the scheduler may interrupt it to assign the CPU to another process.
- **Running** → When the OS scheduler assigns a processor to a process, the process state gets set to running, and the processor executes the process instructions.
- **Waiting** → If a process needs to wait for any resource, such as for user input or for a file to become available, it enters the waiting state.
- **Terminated or Exit** → The process is relocated to the terminated state, where it waits for removal from the main memory once it has completed its execution or been terminated by the operating system.

- Imagine a unit process that executes a simple addition operation and prints it.



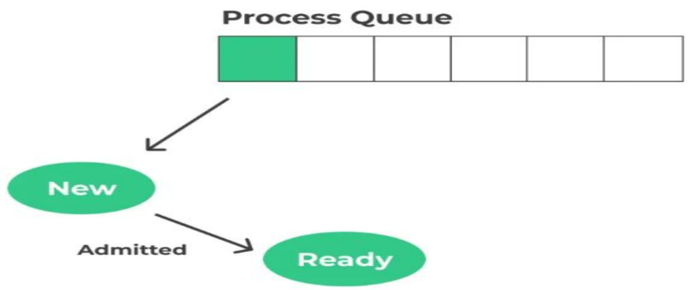
New State

- Process it submitted to the process queue, it in turns acknowledges submission.
- Once submission is acknowledged, the process is given new status.



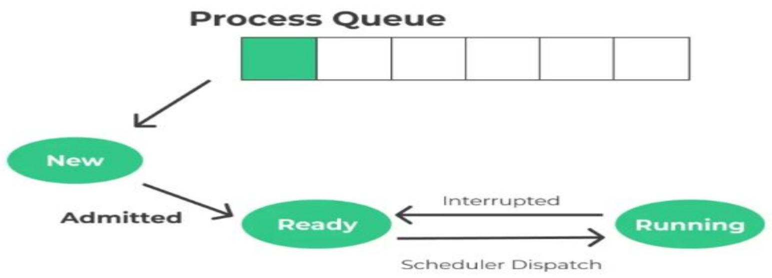
Ready State

- It then goes to Ready State, at this moment the process is waiting to be assigned a processor by the OS



Running State

- Once the Processor is assigned, the process is being executed and turns in Running State.



Wait and Termination State

- Now the process can follow the following transitions –
  - The process may have all resources it needs and may get directly executed and goes to Termination State.
  - Process may need to go to waiting state any of the following
    - Access to Input/Output device (Asking user the values that are required to be added) via console
    - Process maybe intentionally interrupted by OS, as a higher priority operation maybe required, to be completed first
    - A resource or memory access that maybe locked by another process, so current process goes to waiting state and waits for the resource to get free.
  - Once requirements are completed i.e. either it gets back the priority to executed or requested locked resources are available to use, the process will go to running state again where, it may directly go to termination state or may be required to wait again for a possible required input/resource/priority interrupt.
- Termination

Summary

1. New – New Process Created
2. Ready – Process Ready for Processor/computing power allocation
3. Running – Process getting executing
4. Wait – Process waiting for signal
5. Terminated – Process execution completed

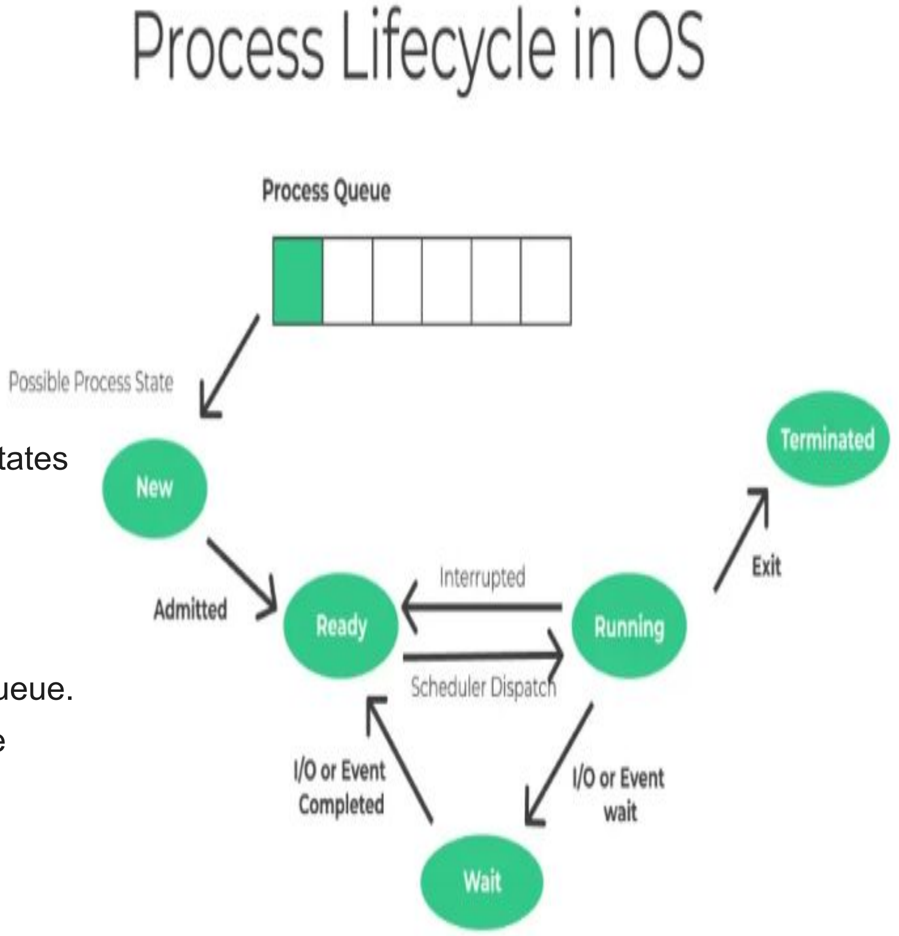
Apart from the above some new systems also propose 2 more states of process which are –

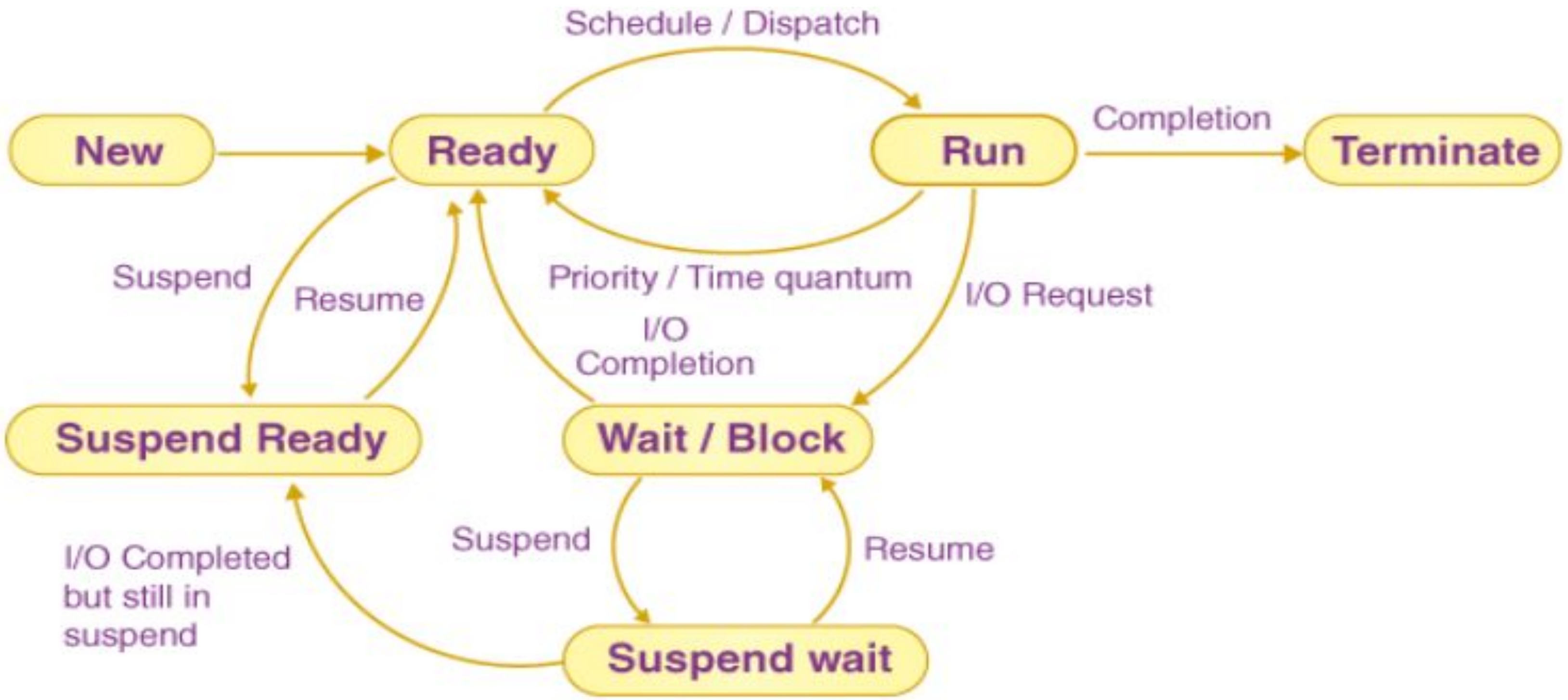
1. **Suspended Ready** –

   There maybe no possibility to add a new process in the queue. In such cases its can be said to be suspended ready state

2. **Suspended Block** –

   If the waiting queue is full

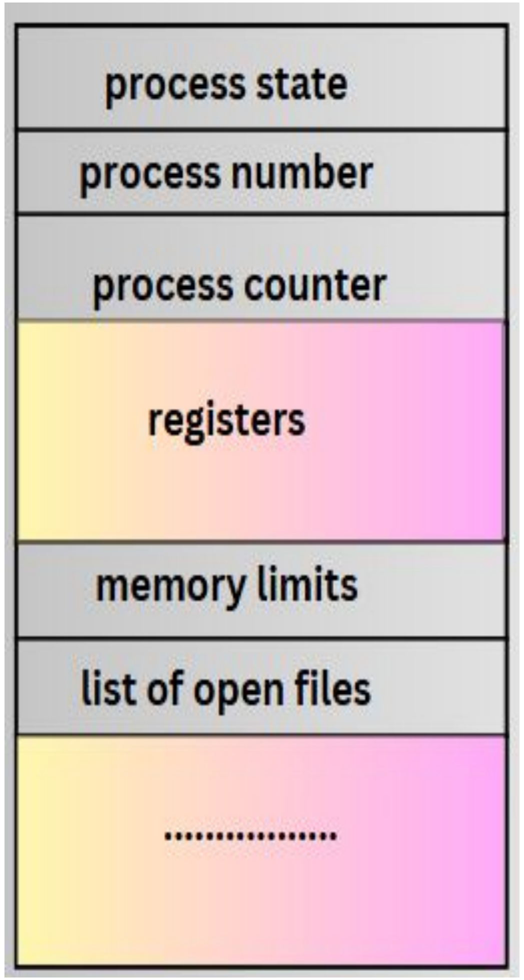




Process State Diagram

Process Control Block (PCB)

- Every process has a process control block, which is a data structure managed by the operating system.
- An integer process ID (or PID) is used to identify the PCB. Also called as **Task Control block**
- **Process state -** The process's present state may be it's new,ready, waiting,running,waiting, halted and so on.
- **Process privileges -** This is required in order to grant or deny access to system resources.
- **Process ID -** Each process in the OS has its own **unique** identifier.
- **Pointer -** It refers to a pointer that points to the **parent process**.
- **Program counter -** The program counter refers to a pointer that points to the address of the process's next instruction.
- **CPU registers -** Processes must be stored in various CPU registers for execution in the running state.



Process Control Block (PCB)

- **CPU scheduling information -** Process priority and additional scheduling information are required for the process to be scheduled
- **Memory management information -** This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.
- **Accounting information -** This comprises CPU use for process execution, time constraints, and execution ID and other things.
- **IO status information -** This section includes a list of the process's I/O devices allocated to the process, a list of open files, and so on.
- PCB serves as the repository for any information that may very from process to process
- **The PCB architecture is fully dependent on the operating system, and different operating systems may include different information**

preemptive and non-preemptive processes

Preemptive process

- In operating systems, scheduling is the method by which processes are given access the CPU
- Process scheduling is performed by the CPU to decide the next process to be executed
- There are two primary types of CPU scheduling:
  a. preemptive
  b. non-preemptive

Different Types of CPU Scheduling Algorithms

There are mainly two types of scheduling methods:

Preemptive Scheduling Method:

- Preemptive scheduling is used **when a process switches from running state to ready state or from the waiting state to the ready state**
- If something is pre-emptive, it is done before other people can act, especially to prevent them from doing something else.

Non-preemptive scheduling

- Non-preemptive scheduling algorithms refer to the class of CPU scheduling technique where once a process is allocated the CPU, **it holds the CPU till the process gets terminated or is pushed to the waiting state.**
- No process is interrupted until it runs to completion.
- The scheduler allocates another process to the CPU only after the currently allocated process terminates and relinquishes control on the CPU.

Difference Between Preemptive and Non-Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
Resources are allocated according to the cycles for a limited time.	Resources are used and then held by the process until it gets terminated.
The process can be interrupted, even before the completion.	The process is not interrupted until its life cycle is complete.
Starvation may be caused, due to the insertion of priority process in the queue.	Starvation can occur when a process with large burst time occupies the system.
Maintaining queue and remaining time needs storage overhead.	No such overheads are required.
Fair scheduling can be applied where all the processes can get equal chance for CPU access	A process may monopolize the CPU.
Deadlocks can be easily avoided.	Deadlocks may occur.

What is the Need for a CPU Scheduling Algorithm?

- It ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.
- In Multiprogramming, if the long-term scheduler selects multiple I/O binding processes then most of the time, the CPU remains idle.

Terminologies Used in CPU Scheduling

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.
- **Turn Around Time = Completion Time – Arrival Time**
- **Waiting Time = Turn Around Time – Burst Time**

Thread

- A thread refers to an **execution unit in the process** that has its own programme counter, stack, as well as a set of registers.
- Now, thread execution is possible within any OS's process. Furthermore, **each and every thread belongs to one single process.**
- Multiple threads can easily communicate information and important data, such as code segments or files, as well as data segments.
- Apart from that, a process can have several threads

Ex - Multiple tabs in a browser, for example, can be considered threads.

MS Word employs many threads to prepare the text in one thread, receive input in another thread, and so on.

Components of Thread

A thread has the following three components:

1. Program Counter
2. Register Set
3. Stack space

Why do we need Thread?

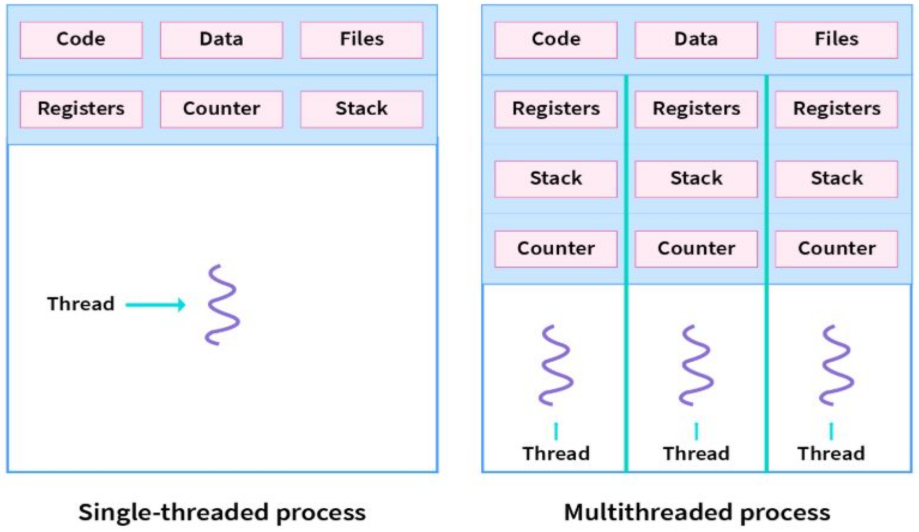
- Creating a new thread in a current process requires significantly less time than creating a new process.
- Threads can share common data without needing to communicate with each other.
- When working with threads, context switching is faster.
- Terminating a thread requires less time than terminating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

Single and Multi threaded Process

- Single threaded processes contain the execution of instructions in a single sequence. In other words, **one command is processes at a time.**
- The opposite of single threaded processes are multithreaded processes.
- These processes **allow the execution of multiple parts of a program at the same time.**
- These are lightweight processes available within the process.

Multithreaded Processes Implementation

- Multithreaded processes can be implemented as user-level threads or kernel-level threads.



User-level Threads and Kernel level Threads

User-level Threads

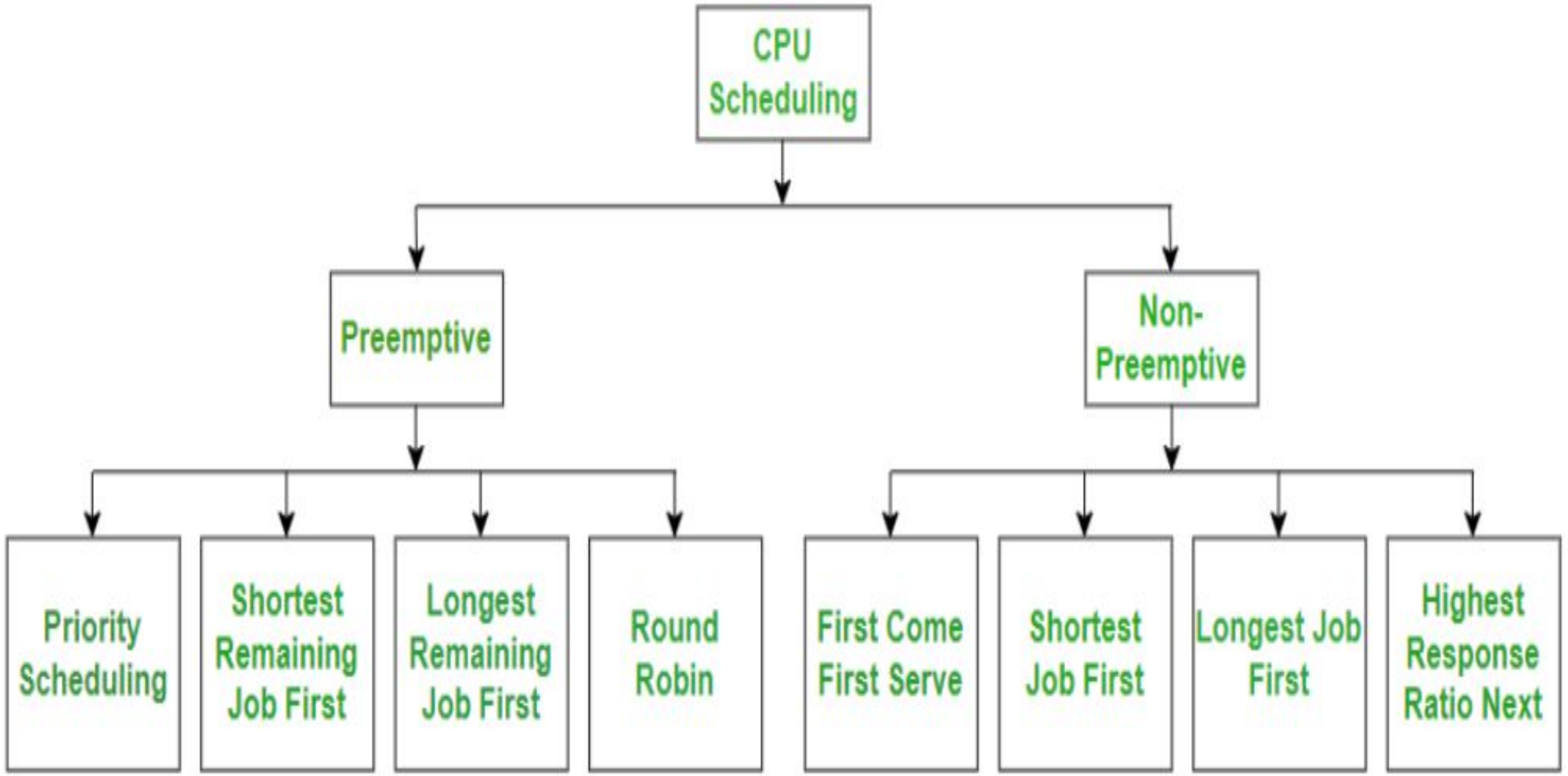
- The user-level threads are implemented by users and the kernel is not aware of the existence of these threads.
- It handles them as if they were single-threaded processes.
- User-level threads are small and much faster than kernel level threads.
- Also, there is no kernel involvement in synchronization for user-level threads.

Kernel-level Threads

- Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel.
- The context information for the process as well as the process threads is all managed by the kernel.
- Because of this, kernel-level threads are slower than user-level threads.

Process vs Thread

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread's data)
Eg: Opening two different browsers.	**Eg**: Opening two tabs in the same browser.


CPU Scheduling
Preemptive
Non-Preemptive
Priority Scheduling
Shortest Remaining Job First
Longest Remaining Job First
Round Robin
First Come First Serve
Shortest Job First
Longest Job First
Highest Response Ratio Next

What are schedulers - Short term, Medium term and Long term

Short-Term Scheduler (CPU scheduler)

- The short-term scheduler selects processes from the **ready queue** that are residing in the main memory and allocates **CPU** to one of them.
- As compared to long-term schedulers, a short-term scheduler has to be used very often i. e. **the frequency of execution of short-term schedulers is high.**
- The short-term scheduler is invoked whenever an event occurs. Such an event may lead to the **interruption of the current process** or it may provide an opportunity to preempt the currently running process in favor of another.
- The example of such events are:
  - Clock ticks (time-based interrupts)
  - I/O interrupts and 1/0 completions
  - Operating system calls
  - Sending and receiving of signals

Medium-Term Scheduler

- The medium-term scheduler is required at the times when a **suspended** or **swapped-out process is to be brought into a pool of ready processes.**
- This is done because there is a limit on the number of active processes that can reside in the main memory.
- The medium-term scheduler is **in-charge of handling the swapped-out process**.
- It has nothing to do with when a process remains suspended.
- However, once the suspending condition is removed, the medium terms scheduler attempts to allocate the required amount of main memory and swap the process in and make it ready.
- Thus, the medium-term scheduler plans the CPU scheduling for processes that have been waiting for the completion of another process or an I/O task.

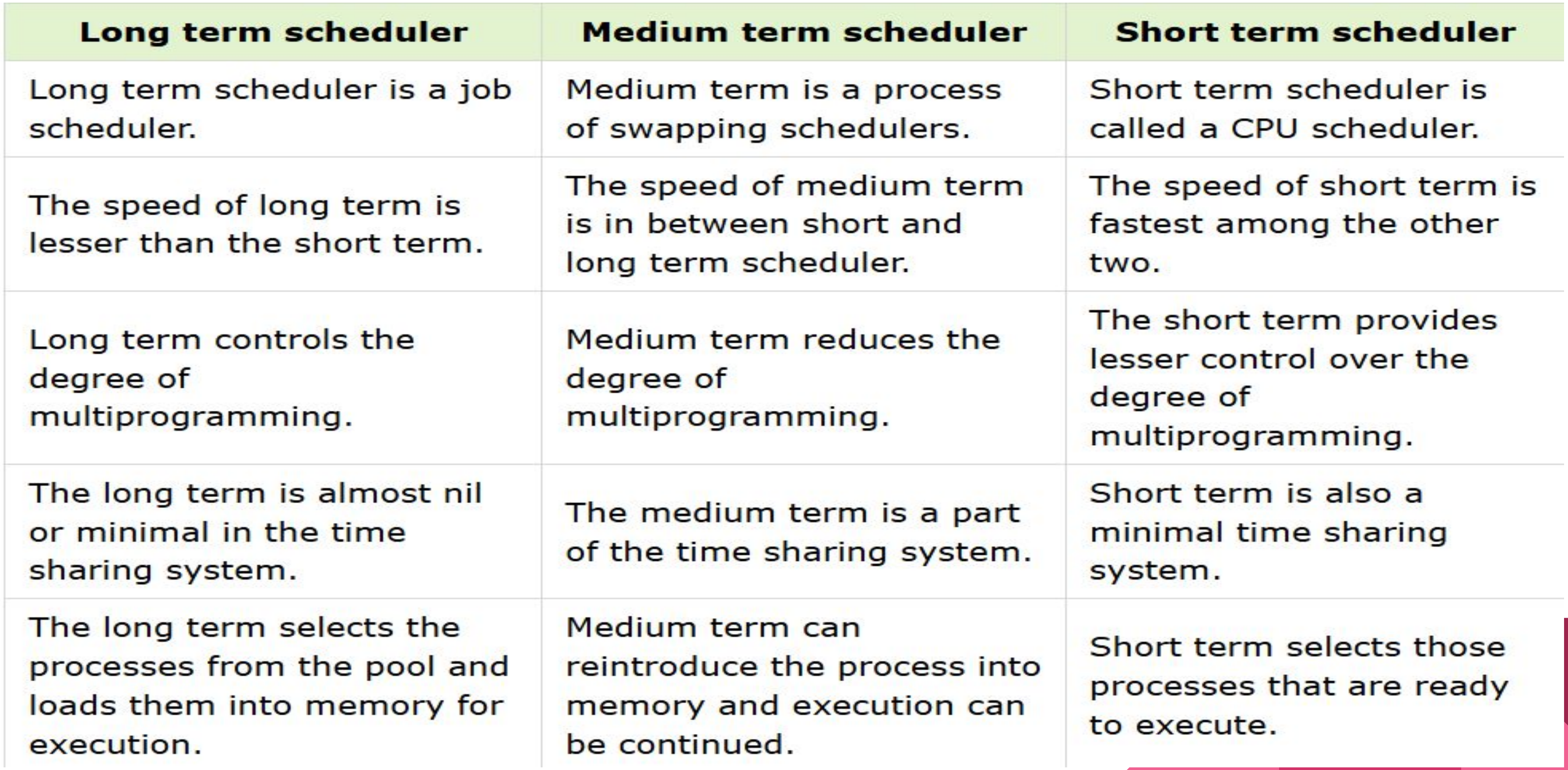
Long-Term Scheduler (job scheduler)

- The long-term scheduler works with the **batch queue** and selects the **next batch job** to be executed.
- Processes, which are resource intensive and have a low priority are called **batch jobs.**
- **For example, a user requests for printing a bunch of files.**
- We can also say that a **long-term scheduler selects the processes or jobs from secondary storage device**
- **eg, a disk and loads them into the memory for execution**.
- The long-term scheduler is called “**long-term**” **because the time for which the scheduling is valid is long**.
- This scheduler shows the **best performance** by selecting a good process mix of I/O-bound and CPU-bound processes.

Summary of Short term, Medium term and Long term Schedulers

- **Short Term Scheduler** decides which process to execute next, responsible for CPU scheduling, or It carry process from ready queue to running state.
- **Medium Term Scheduler** handles process swapping between main memory and secondary storage, manages the degree of multiprogramming.
- **Long Term Scheduler** decide which processes to admit to the system, or it selects the processes or jobs from secondary storage device eg, a disk and loads them into the main memory for execution.

Short term VS Medium term VS Long term Schedulers



Long term scheduler	Medium term scheduler	Short term scheduler
Long term scheduler is a job scheduler.	Medium term is a process of swapping schedulers.	Short term scheduler is called a CPU scheduler.
The speed of long term is lesser than the short term.	The speed of medium term is in between short and long term scheduler.	The speed of short term is fastest among the other two.
Long term controls the degree of multiprogramming.	Medium term reduces the degree of multiprogramming.	The short term provides lesser control over the degree of multiprogramming.
The long term is almost nil or minimal in the time sharing system.	The medium term is a part of the time sharing system.	Short term is also a minimal time sharing system.
The long term selects the processes from the pool and loads them into memory for execution.	Medium term can reintroduce the process into memory and execution can be continued.	Short term selects those processes that are ready to execute.

FCFS - First Come First Serve CPU Scheduling

- The processes are attended to in the order **in which they arrive in the ready queue**, much **like customers lining up at a grocery store.**

How Does FCFS Work?

1. **Arrival:** Processes enter the system and are placed in a queue in the order they arrive.
2. **Execution:** The CPU takes the first process from the front of the queue, executes it until it is complete, and then removes it from the queue.
3. **Repeat:** The CPU takes the next process in the queue and repeats the execution process.

This continues until there are no more processes left in the queue.

FCFS CPU Scheduling:

Example of FCFS CPU Scheduling:

To understand the First Come, First Served (FCFS) scheduling algorithm effectively, we'll use two examples –

- one where all processes arrive at the same time,
- another where processes arrive at different times.

We'll create **Gantt charts** for both scenarios and **calculate the turnaround time and waiting time for each process**.

Scenario 1: Processes with Same Arrival Time

Consider the following table of arrival time and burst time for three processes P1, P2 and P3

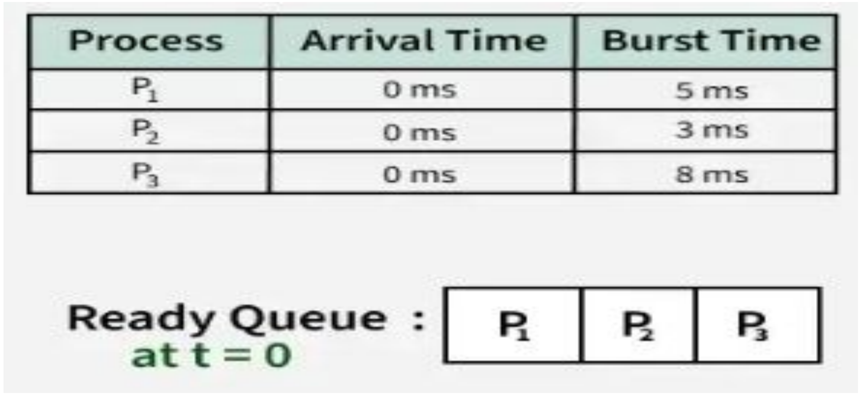
process	Arrival Time	Burst Time
P1	0	5
P2	0	3
P3	0	8

Step-by-Step Execution:

1. **P1** will start first and run for 5 units of time (from 0 to 5).
2. **P2** will start next and run for 3 units of time (from 5 to 8).
3. **P3** will run last, executing for 8 units (from 8 to 16).

FCFS Scheduling

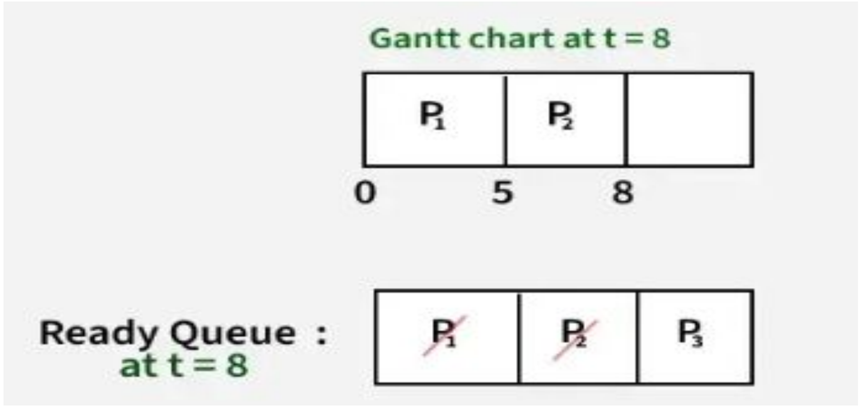
Step No 1



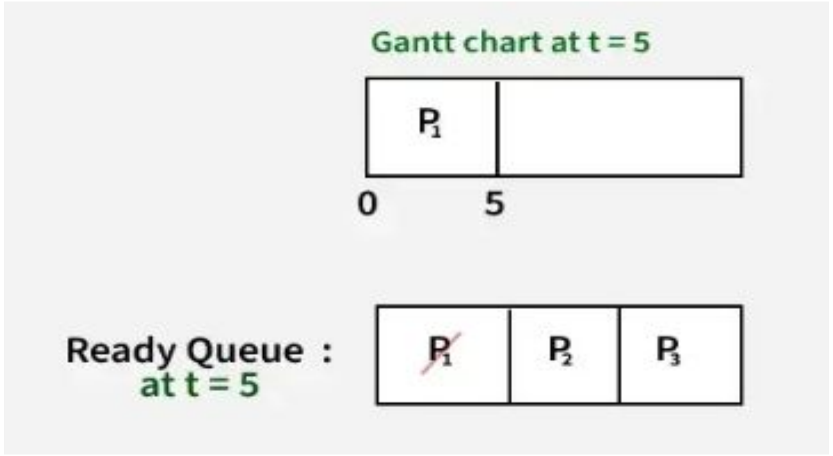
Process	Arrival Time	Burst Time
P_1	0 ms	5 ms
P_2	0 ms	3 ms
P_3	0 ms	8 ms

Ready Queue : at t = 0
P_1
P_2
P_3

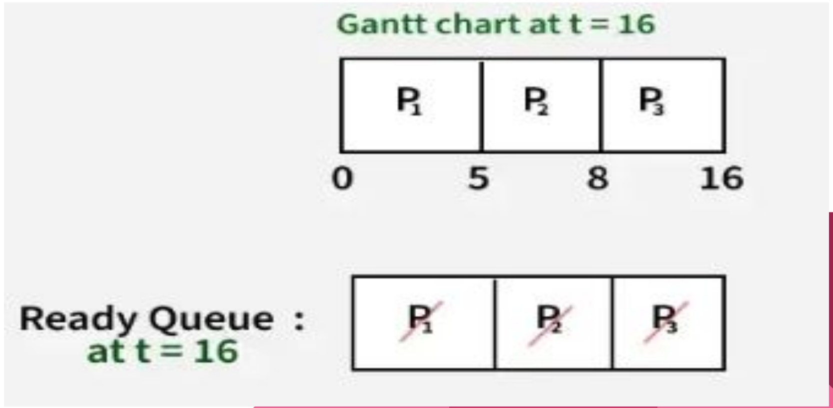
Step No 2



Step No 3



Step No 4



Turnaround Time = Completion Time - Arrival Time

Waiting Time = Turnaround Time - Burst Time

AT : Arrival Time

BT : Burst Time or CPU Time

TAT : Turn Around Time

WT : Waiting Time

- **Average Turn around time** = 9.67
- **Average waiting time** = 4.33

Processes	AT	BT	CT	TAT	WT
P1	0	5	5	5-0 = 5	5-5 = 0
P2	0	3	8	8-0 = 8	8-3 = 5
P3	0	8	16	16-0 = 16	16-8 = 8

Scenario 2: Processes with Different Arrival Times

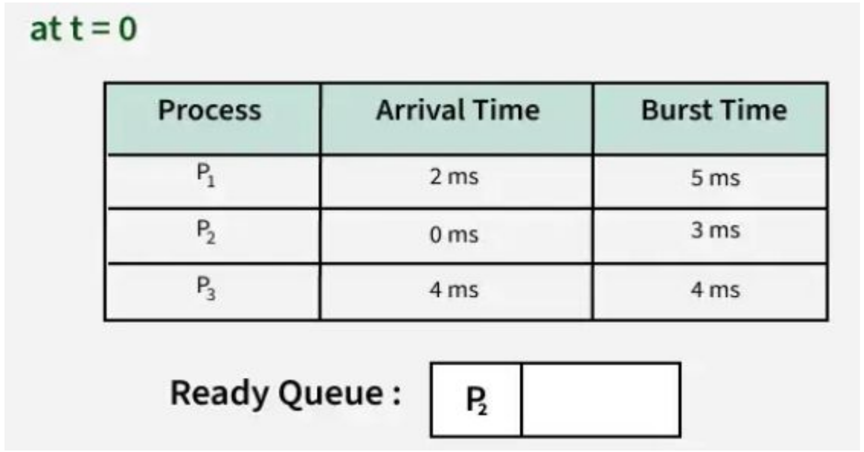
Consider the following table of arrival time and burst time for three processes P1, P2 and P3

Process	Burst Time (BT)	Arrival Time (AT)
P1	5 ms	2 ms
P2	3 ms	0 ms
P3	4 ms	4 ms

Step-by-Step Execution:

- **P2** arrives at time 0 and runs for 3 units, so its completion time is:
  Completion Time of P2 = 0 + 3 =3
- **P1** arrives at time 2 but has to wait for **P2** to finish. **P1** starts at time 3 and runs for 5 units. Its completion time is:
  Completion Time of P1 = 3 + 5 = 8
- **P3** arrives at time 4 but has to wait for **P1** to finish. **P3** starts at time 8 and runs for 4 units. Its completion time is:
  Completion Time of P3 = 8 + 4 = 12

Step No 1

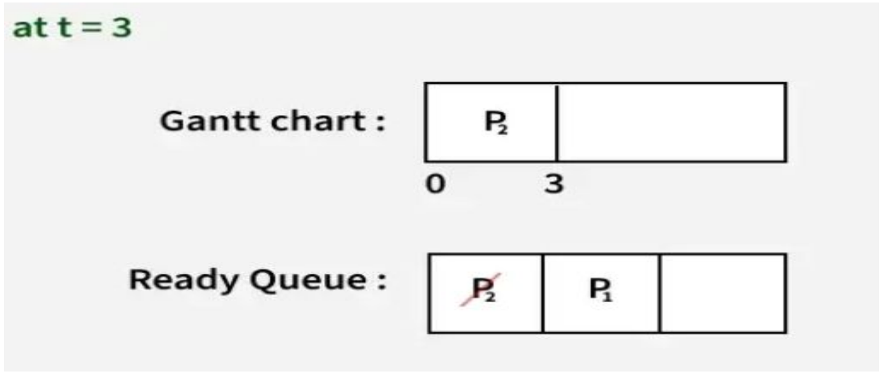


at t = 0

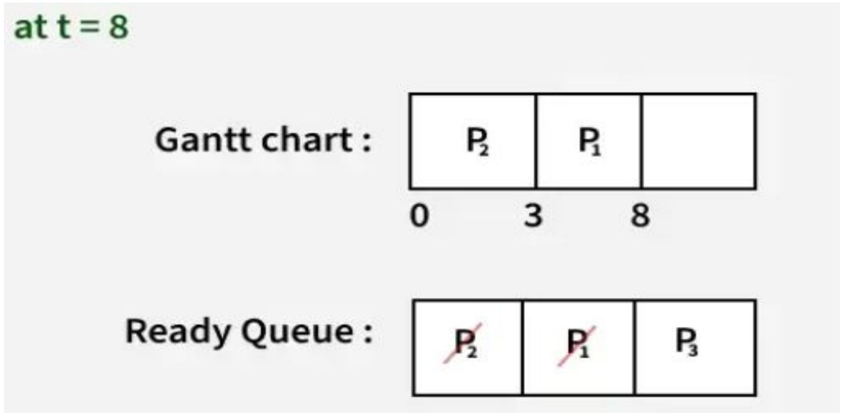
Process	Arrival Time	Burst Time
P_1	2 ms	5 ms
P_2	0 ms	3 ms
P_3	4 ms	4 ms

Ready Queue : P_2

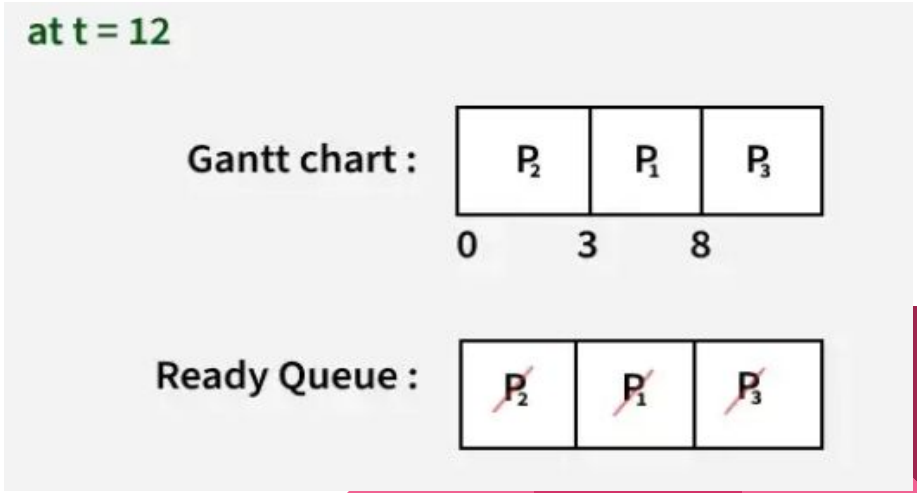
Step No 2



Step No 3



Step No 4



Now, lets calculate average waiting time and turn around time:

- **Average Turnaround time** = 5.67
- **Average waiting time** = 1.67

Process	Completion Time (CT)	Turnaround Time (TAT = CT – AT)	Waiting Time (WT = TAT – BT)
P2	3 ms	3 ms	0 ms
P1	8 ms	6 ms	1 ms
P3	12 ms	8 ms	4 ms

Advantages and Disadvantages of FCFS

Advantages of FCFS

- The simplest and basic form of CPU Scheduling algorithm
- Every process gets a chance to execute in the order of its arrival.
- Easy to implement, it doesn’t require complex data structures.
- It is well suited for batch systems where the longer time periods for each process are often acceptable.

Disadvantages of FCFS

- FCFS can result in long waiting times, especially if a long process arrives before a shorter one. This is known as the **convoy effect**, where shorter processes are forced to wait behind longer processes, leading to inefficient execution.
- The average waiting time in the FCFS is much higher than in the others
- Processes that are at the end of the queue, have to wait longer to finish.
- It is not suitable for time-sharing operating systems where each process should get the same amount of CPU time.

SJF (SHORTEST JOB FIRST) Scheduling or Shortest Job Next (SJN)/Shortest Remaining Time First (SRTF)

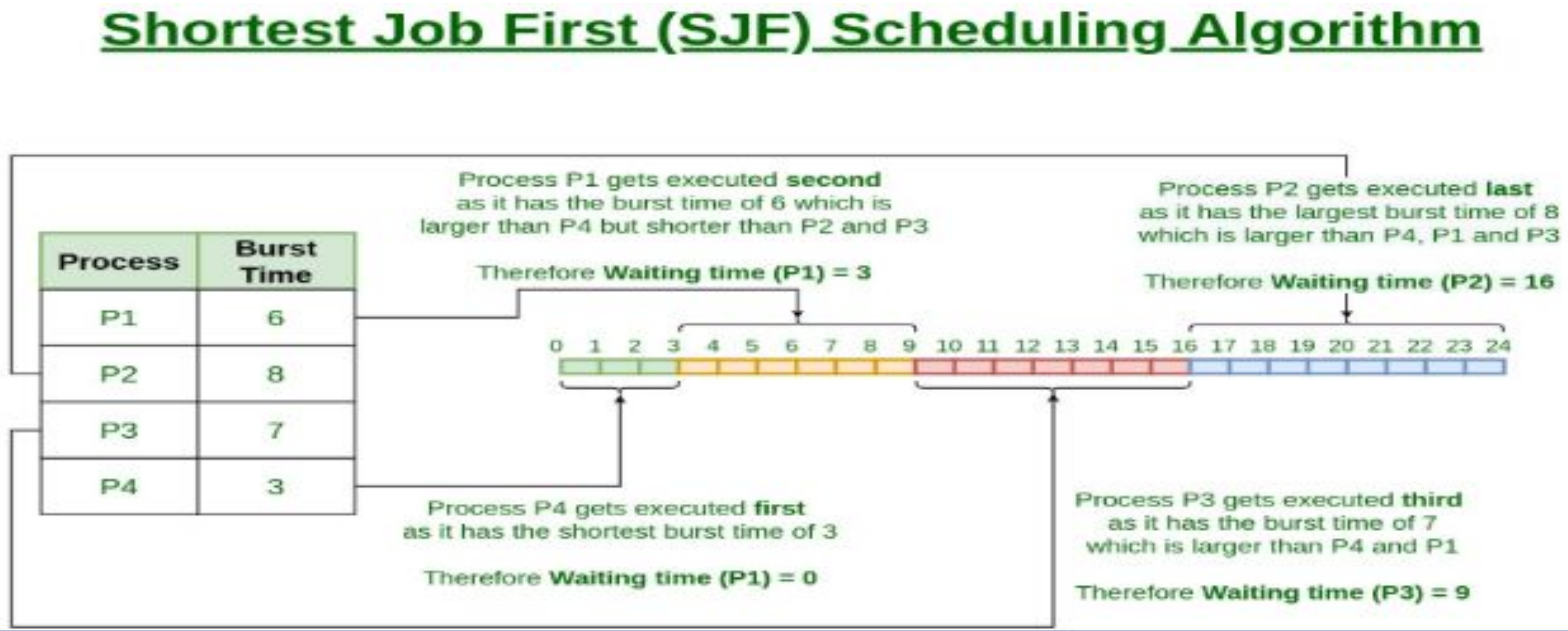
- In the Shortest Job First scheduling algorithm, the processes are **scheduled in ascending order of their CPU burst times**
- Here, if a short process enters the ready queue while a longer process is executing, process switch occurs by which the executing process is swapped out to the ready queue while the newly arrived shorter process starts to execute.
- Thus the short term scheduler is invoked either when a new process arrives in the system or an existing process completes its execution.

Features of SJF Algorithm

- SJF allocates CPU to the process with shortest execution time.
- In cases where two or more processes have the same burst time, arbitration is done among these processes on first come first serve basis.
- There are both preemptive and non-premptive
- It minimises the average waiting time of the processes.

Implementation of SJF Scheduling

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes (a ready queue) that arrives afterward till the completion of the previous process and select that process in that queue which is having minimum Burst time.



Shortest Job First (SJF) Scheduling Algorithm

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Process P1 gets executed **second** as it has the burst time of 6 which is larger than P4 but shorter than P2 and P3

Therefore **Waiting time (P1) = 3**

Process P2 gets executed **last** as it has the largest burst time of 8 which is larger than P4, P1 and P3

Therefore **Waiting time (P2) = 16**

Process P4 gets executed **first** as it has the shortest burst time of 3

Therefore **Waiting time (P1) = 0**

Process P3 gets executed **third** as it has the burst time of 7 which is larger than P4 and P1

Therefore **Waiting time (P3) = 9**

Generalized Activity Normalization Time Table (GANTT) chart

- It is a production control tool and horizontal bar chart used for graphical representation of schedule that helps to plan in an efficient way, coordinate, and track some particular tasks in project.

Advantages of SJF Scheduling

- SJF reduces the average waiting time.
- SJF is generally used for long term scheduling.
- It is suitable for the jobs running in batches, where run times are already known.
- SJF is probably optimal in terms of average Turn Around Time (TAT).

Disadvantages of SJF Scheduling

- In SJF job completion time must be known earlier.
- Many times it becomes complicated to predict the length of the upcoming CPU request.

Priority Scheduling

- **Priority scheduling** is one of the **most common scheduling algorithms** used by the operating system to schedule processes based on their priority. Each process is assigned a priority.
- The process with the **highest priority is to be executed first** and so on.
- **Processes with the same priority are executed on a first-come first served basis**.

Ways to decide the Priority

- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- Also priority can be decided on the ratio of average I/O to average CPU burst time.

Priority Scheduling can be implemented in two ways: Non-Preemptive Priority Scheduling and Preemptive Priority Scheduling

Non-Preemptive Priority Scheduling

- In Non-Preemptive Priority Scheduling, the CPU is not taken away from the running process. Even if a higher-priority process arrives, the currently running process will complete first.

Ex: A high-priority process must wait until the currently running process finishes.

Example of Non-Preemptive Priority Scheduling:

Consider the following table of arrival time and burst time for three processes P1, P2 and P3:

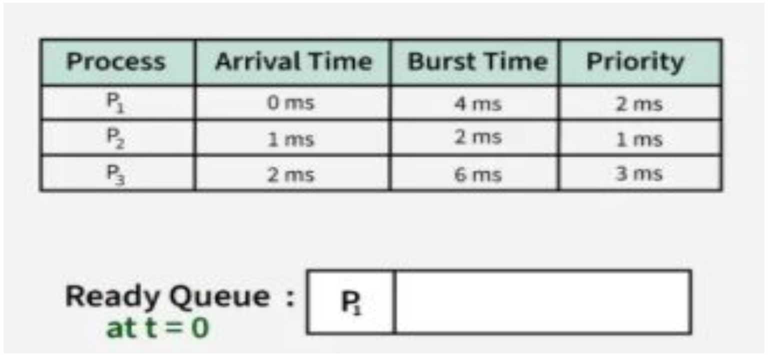
Note: Lower number represents higher priority.

Step-by-Step Execution:

- **At Time 0:** Only P1 has arrived. P1 starts execution as it is the only available process, and it will continue executing till t = 4 because it is a non-preemptive approach.
- **At Time 4:** P1 finishes execution. Both P2 and P3 have arrived. Since P2 has the highest priority (Priority 1), it is selected next.
- **At Time 6:** P2 finishes execution. The only remaining process is P3, so it starts execution.
- **At Time 12:** P3 finishes execution

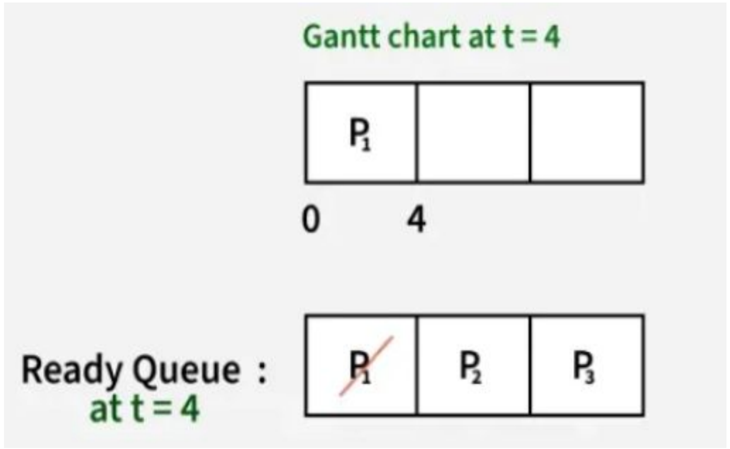
Process	Arrival Time	Burst Time	Priority
P1	0	4	2
P2	1	2	1
P3	2	6	3

Step No 1

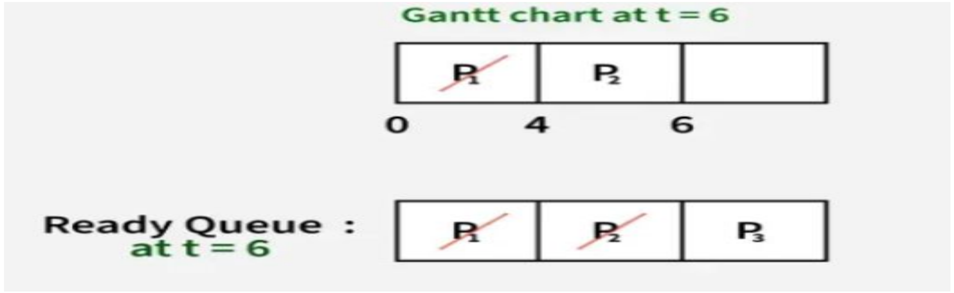


Process	Arrival Time	Burst Time	Priority
P_1	0 ms	4 ms	2 ms
P_2	1 ms	2 ms	1 ms
P_3	2 ms	6 ms	3 ms

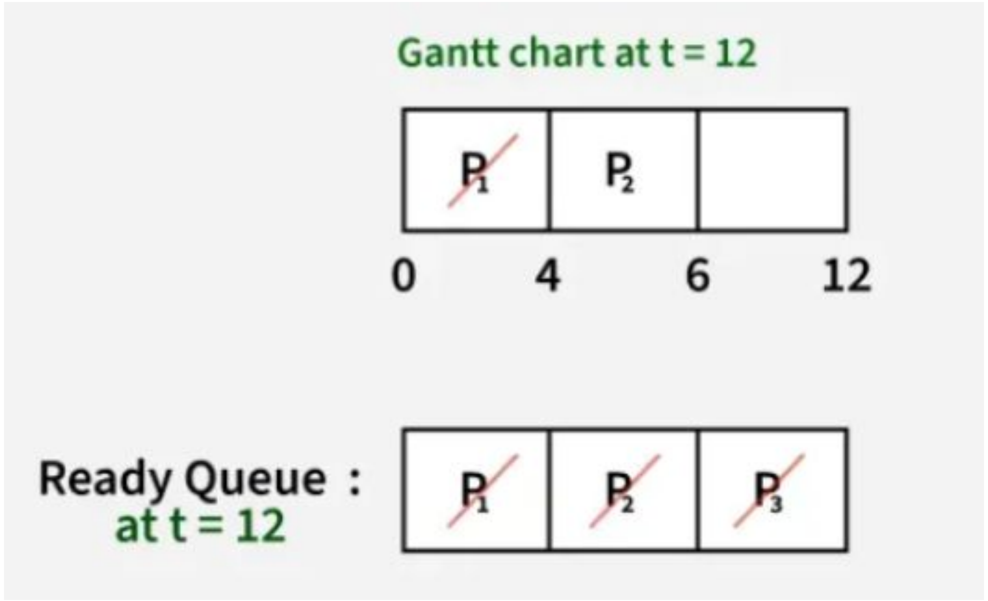
Step No 2



Step No 3



Step No 4



Advantages and Disadvantages of Priority Scheduling

Advantages of Priority Scheduling

- Implementation is simple, since scheduler doesnot require doing any prior calculations.
- Once CPU defines the relative relevance (priorities) of the processes, the order of execution is easily predictable.
- Higher priority processes are almost served immediately.
- Priority scheduling is particularly helpful in systems that have variety of processes each with their own needs.

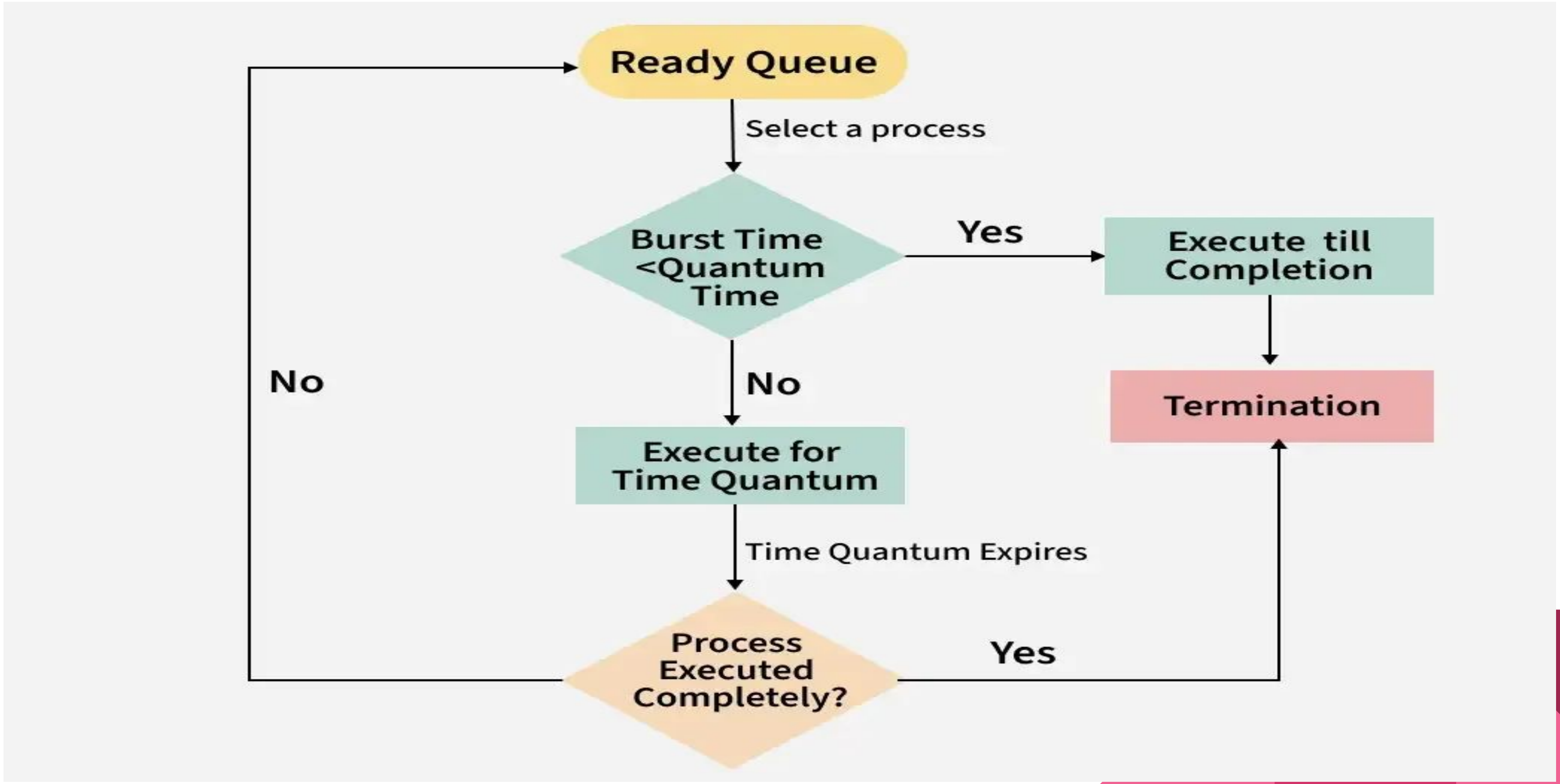
Disadvantages of Priority Scheduling

- In static priority systems, lower priority processes may need to wait indefinitely, this results in stagnation.
- In non-preemptive priority scheduling, often a large process keeps shorter processes waiting for long time.
- In preemptive priority scheduling, a low priority process may be repeatedly pre-empted by intermittent streams of high priority processes requiring frequent context switches.

Round Robin (RR) Scheduling

- **Round Robin Scheduling is one of the most efficient** and the most widely used **not only in process scheduling in operating systems but also in network scheduling.**
- This scheduling strategy derives its name from an age **old round-robin principle which advocated that all participants are entitled to equal share of assets or opportunities in a turn wise manner.**
- In RR scheduling, **each process gets equal time slices (or time quanta) for which it executes in the CPU in turn wise manner.**
- When a process gets its turn, it executes for the assigned time slice and then relinquishes the CPU for the next process in queue.
- If the process has burst time left, then it is sent to the end of the queue. Processes enter the queue on first come first serve basis.
- Round Robin scheduling is **preemptive**, which means that a running process can be interrupted by another process and sent to the ready queue even when it has not completed its entire execution in CPU.
- It is a **preemptive version of First Come First Serve (FCFS) scheduling algorithm**.

Working of Round Robin Scheduling

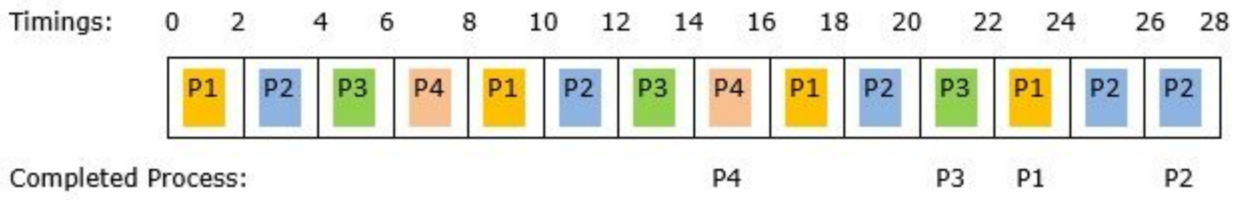


Example of Round Robin Scheduling

- Let us consider a system that has four processes which have arrived at the same time
- in the order P1, P2, P3 and P4.
- Let us consider time quantum of 2ms and perform RR scheduling on this.
- We will draw GANTT chart and find the average turnaround time and average waiting time.

Process	CPU Burst Times in ms
P1	8
P2	10
P3	6
P4	4

GANTT Chart with time quantum of 2ms



Timings: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28
P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P1 P2 P2
Completed Process: P4 P3 P1 P2

Average Turnaround Time

- Average TAT = Sum of Turnaround Time of each Process **/** Number of Processes

  (TATP1+TATP2+TATP3+TATP4) / 4 = (24 + 28 + 22+ 16) / 4 = **22.5 ms**

- In order to calculate the waiting time of each process, we multiply the time quantum with the number of time slices the process was waiting in the ready queue.

Average Waiting Time

- **Average WT** = Sum of Waiting Time of Each Process **/** Number of processes

  = (WTP1+WTP2+WTP3+WTP4)/4

  = (8*2 + 9*2+ 8*2+ 6*2) / 4 = **15.5 ms**

- **Average Waiting Time** = Sum of Waiting Time of Each Process / Number of processes

  = (WTP1 + WTP2 + WTP3 + WTP4) / 4

  = (0 + 2 +15 + 8) / 4 = **6.25 ms**

Advantages of Round Robin Scheduling

- Round Robin scheduling is the most a fair scheduling algorithm
- Starvation is totally eliminated in RR scheduling.
- It does not require any complicated method to calculate the CPU burst time
- It is pretty simple to implement and so finds application in a wide range of situations.
- Convoy effect does not occur in RR scheduling

Disadvantages of Round Robin Scheduling

- The performance is highly dependent upon the chosen time quantum. This requires prudent analysis before implementation, failing which required results are not received.
- RR scheduling does not give any scope to assign priorities to processes. So, system processes which need high priority gets the same preference as background processes. This may often hamper the overall performance of a system.

Types of Scheduling Queues

Job Queue (In Disk)

- This queue contains all the processes or jobs in the list that are waiting to be processed.
- When a job is created, it goes into the job queue and waits until it is ready for processing.
  - Contains all submitted jobs.
  - Processes are stored here in a wait state until they are ready to go to the execution stage.
  - This is the first and most basic state that acts as a default storage of new jobs added to a scheduling system.
  - Long Term Scheduler Picks a process from Job Queue and moves to ready queue.

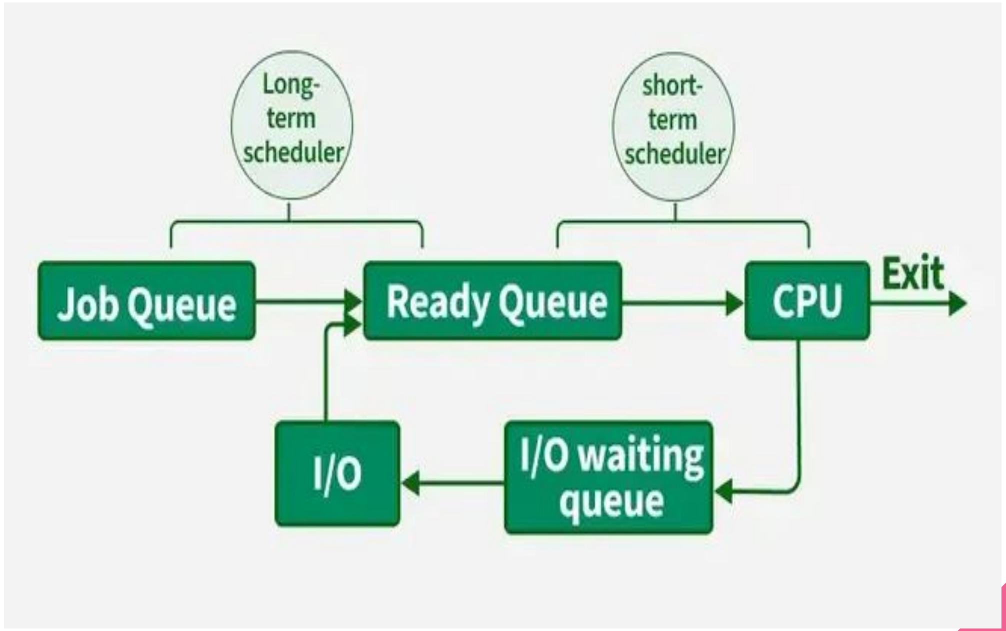
Ready Queue (In Main Memory)

- This queue contains all the processes ready to be fetched from the memory, for execution.
- When the process is initiated, it joins the ready queue to wait for the CPU to be free.
- The operating system assigns a process to the executing processor from this queue based on the scheduling algorithm it implements.
  - Contains processes (mainly their PCBs) waiting for the CPU to execute various processes it contains.
  - They are controlled using a scheduling algorithm like FCFS, SJF, or Priority Scheduling.
  - Short Term Scheduler picks a process from Ready Queue and moves the selected process to running state.

Block or Device Queues (In Main Memory)

- The processes which are blocked due to unavailability of an I/O device are added to this queue.
- Every device has its own block queue.

Flow of Movement of processes in the above different Queues



Belady’s Anomaly

- Belady's Anomaly can be **seen in the concept of Page replacement**.
- Any process is divided into pages and put in the frames of the main memory.
- But the number of frames is fixed, i.e.- they are not infinite. After some time, when new pages are needed to load, the old pages are removed or swapped from the memory.
- When a process is initialized, it requests the allocation of frames into the memory.
- There will be two situations:
    - First, if there is free space in frames for pages to be loaded
    - Second if there is no free space in frames, then the frame that is for the longest time is replaced by new frame.
- When the number of frames is manipulated, there comes a situation when number of faults increases as we increase the number of frames.
- Normally, as more page frames are available, the operating system has more flexibility to keep the necessary pages in memory, which should reduce the number of page faults. However, in the case of Belady’s Anomaly, this intuition fails, and we observe an unexpected increase in page faults with more available frames.
- **Page fault**- When there is no required page present in RAM (secondary memory) on calling from the CPU, this is known as a "Page Fault".

- Till now we know that whenever we execute a program, then a process is created and would be terminated after the completion of the execution.
- What if we need to create a process within the program and may be wanted to schedule a different task for it.

Can this be achieved?

- Till now we know that whenever we execute a program, then a process is created and would be terminated after the completion of the execution.
- What if we need to create a process within the program and may be wanted to schedule a different task for it.

Can this be achieved?

❖ Yes, obviously through process creation.
❖ Of course, after the job is done it would get terminated automatically or you can terminate it as needed.

Process creation using fork

- Process creation is achieved through the **fork() system call**.
- The newly created process is called the **child process** and the process that initiated it (or the process when execution is started) is called the **parent process**.
- **After the fork() system call, now we have two processes - parent and child processes.**

How to differentiate them? - it is through their return values.

- After creation of the child process, let us see the fork() system call details.

```
#include <sys/types.h>
#include <unistd.h>          →          Creates the child process
pid_t fork(void);
```

- After this call, there are two processes, **the existing one is called the parent process and the newly created one is called the child process**.
- The fork() system call returns either of the three values −
  - ➔ Negative value to indicate an error, i.e., unsuccessful in creating the child process.
  - ➔ Returns a zero for child process.
  - ➔ Returns a positive value for the parent process. This value is the process ID of the newly created child process.

Process creation using fork

Simple program

File name: basicfork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
        fork();
        printf("Called fork() system call\n");
        return 0;
    }
```

Execution/Output

```
Called fork() system call
Called fork() system call
```

Note − Usually after fork() call, the child process and the parent process would perform different tasks. If the same task needs to be run, then for each fork() call it would run 2 power n times, where **n** is the number of times fork() is invoked.

In the above case, fork() is called once, hence the output is printed twice (2 power 1).

Process creation using fork

File name: pids_after_fork.c

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

        pid_t pid, mypid, myppid;

        pid = getpid();

        printf("Before fork: Process id is %d\n", pid);

        pid = fork();

       if (pid < 0) {

                perror("fork() failure\n");

                return 1;

             }
```

```
// Child process

   if (pid == 0) {

                printf("This is child process\n");

                mypid = getpid();

                myppid = getppid();

                printf("Process id is %d and PPID is %d\n", mypid, myppid);

   } else {

// Parent process

     sleep(2);

     printf("This is parent process\n");

     mypid = getpid();

     myppid = getppid();

     printf("Process id is %d and PPID is %d\n", mypid, myppid);

     printf("Newly created process id or child pid is %d\n", pid);

   }

   return 0;

}
```

Compilation and Execution Steps

```
Before fork: Process id is 166629
This is child process
Process id is 166630 and PPID is 166629
Before fork: Process id is 166629
This is parent process
Process id is 166629 and PPID is 166628
Newly created process id or child pid is 166630
```

How to the process be terminated?

- A process can terminate in either of the two ways −
    - Abnormally, occurs on delivery of certain signals, say terminate signal.
    - Normally, using _exit() system call (or _Exit() system call) or exit() library function.
- The difference between _exit() and exit() is mainly the cleanup activity.
- The **exit()** does some cleanup before returning the control back to the kernel, while the **_exit()** (or _Exit()) would return the control back to the kernel immediately.

- What happens if the parent process finishes its task early than the child process and then quits or exits?
- Now who would be the parent of the child process?

init process

- Let us consider an example program, where the parent process does not wait for the child process, which **results into init process becoming the new parent for the child process**.

File name: parentprocess_nowait.c

```
#include<stdio.h>
int main() {
        int pid;
        pid = fork();

        // Child process

        if (pid == 0) {
                system("ps -ef");
                sleep(10);
                system("ps -ef");
                }
        else {
            sleep(3);
           }
            return 0;
    }
```

- The parent of the child process is init process, which is the very first process initiating all the tasks.

- To monitor the child process execution state, to check whether the child process is running or stopped or to check the execution status, etc. the wait() system calls and its variants is used.

wait() , waitpid() & waitid()

- The variants of system calls to monitor the child process/es
  - ❖ wait()
  - ❖ waitpid()
  - ❖ waitid()

- **wait()** system call would wait for one of the children to terminate and return its termination status in the buffer as explained below.

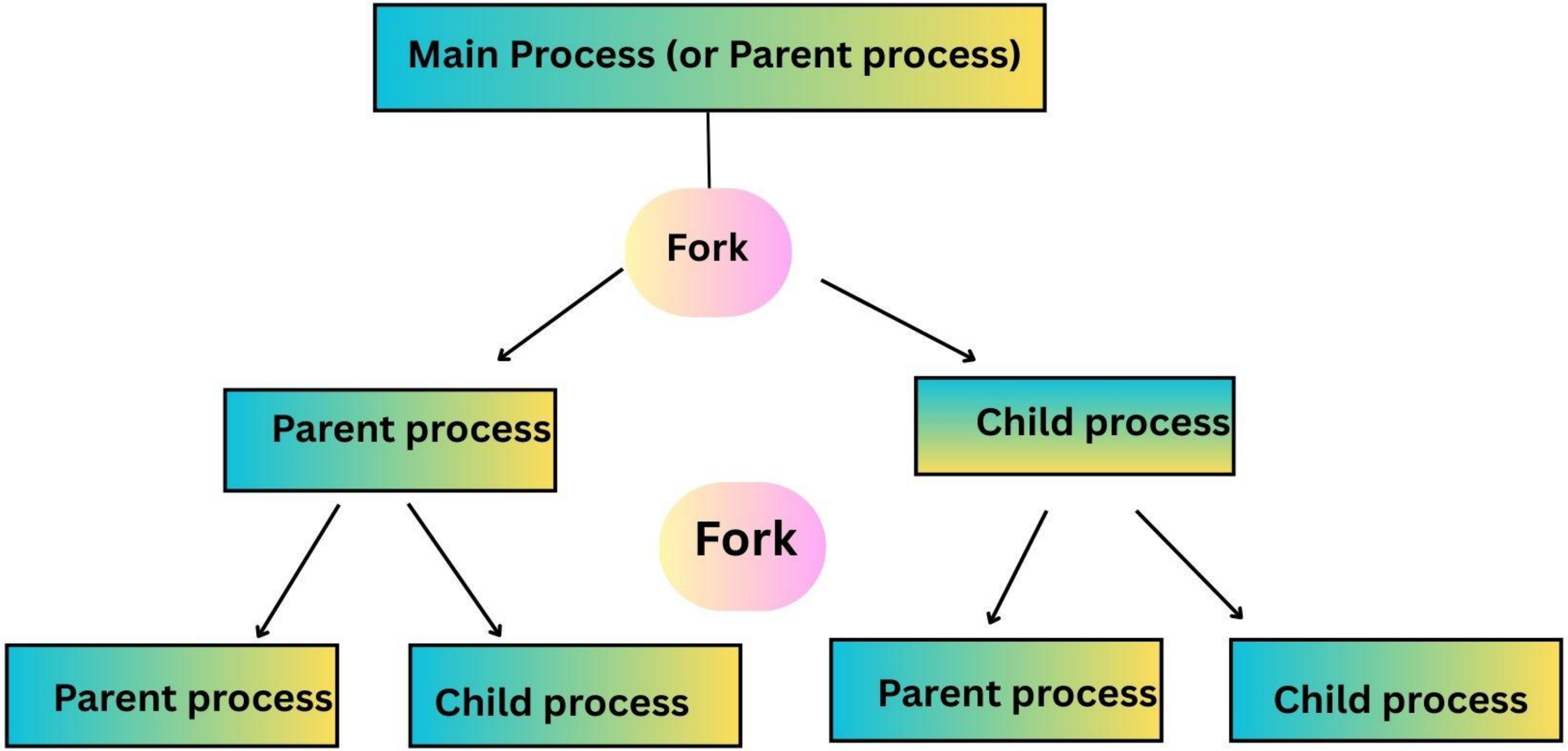
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- This call returns the process ID of the terminated child on success and -1 on failure.
- The wait() system call suspends the execution of the current process and waits indefinitely until one of its children terminates.
- The termination status from the child is available in status.

/* File name: parentprocess_waits.c */

```
#include<stdio.h>
int main() {
            int pid;
            int status;
            pid = fork();

   // Child process
            if (pid == 0) {
                        system("ps -ef");
                        sleep(10);
                        system("ps -ef");
                        return 3;
            //exit status is 3 from child process
            } else {
                  sleep(3);
                  wait(&status);
                  printf("In parent process: exit status
from child is decimal %d, hexa %0x\n", status, status);
                  }
                  return 0;
            }
```


Main Process (or Parent process)
Fork
Parent process
Child process
Fork
Parent process
Child process
Parent process
Child process

wait() , waitpid() & waitid()

- wait() system call has limitation such as it can only wait until the exit of the next child.

When to use waitpid()?

- If we need to wait for a specific child it is not possible using wait(), however, it is possible using waitpid() system call.
- The **waitpid() system call would wait for specified children to terminate and return its termination status in the buffer** as explained below.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- The above call returns the process ID of the terminated child on success and -1 on failure.
- The **waitpid() system call suspends the execution of the current process and waits indefinitely until the specified children (as per pid value) terminates.**
- The termination status from the child is available in the status.

wait() , waitpid() & waitid()

- The value of pid can be either of the following −

  <-1 − Wait for any child process whose process group ID is equal to the absolute value of pid.

  -1 − Wait for any child process, which equals to that of wait() system call.

  0 − Wait for any child process whose process group ID is equal to that of the calling process.

  >0 − Wait for any child process whose process ID is equal to the value of pid
- By default, **waitpid() system call waits only for the terminated children but this default behavior can be modified using the options argument**.

  let us check for waitid() system call. This system call waits for the child process to change state.

  ```
  #include <sys/wait.h>
  int waitpid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
  ```
- The above system call waits for the child process to change the state and this call suspends the current/calling process until any of its child process changes its state.
- The argument infop is to record the current state of the child.
- This call returns immediately, if the process has already changed its state.
- The value of idtype can be either of the following −

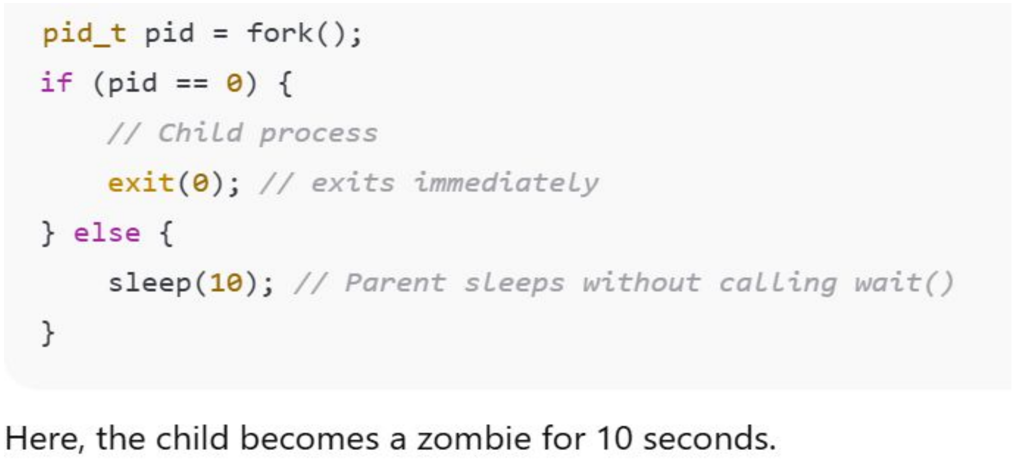
  P_PID − Wait for any child process whose process ID is equal to that of id.

Difference between fork() and exec()

SNO	fork()	exec()
1.	It is a system call in the C programming language	It is a system call of operating system
2.	It is used to create a new process	exec() runs an executable file
3.	Its return value is an integer type	It does not creates new process
4.	It does not takes any parameters.	Here the Process identifier does not changes
5.	It can return three types of integer values	In exec() the machine code, data, heap, and stack of the process are replaced by the new program.

Zombie processes

- A **Zombie Process** is a process that has **completed its execution**, but it's **entry still remains in the process table** to allow the parent process to read its exit status.
- It is created when a **child process finishes**, but the **parent has not yet called wait()**.
- It still occupies an entry in the **process table**.
- It's **not actually running** — just waiting for the parent to collect the exit info.
- If too many zombies accumulate, they can **exhaust system resources**.



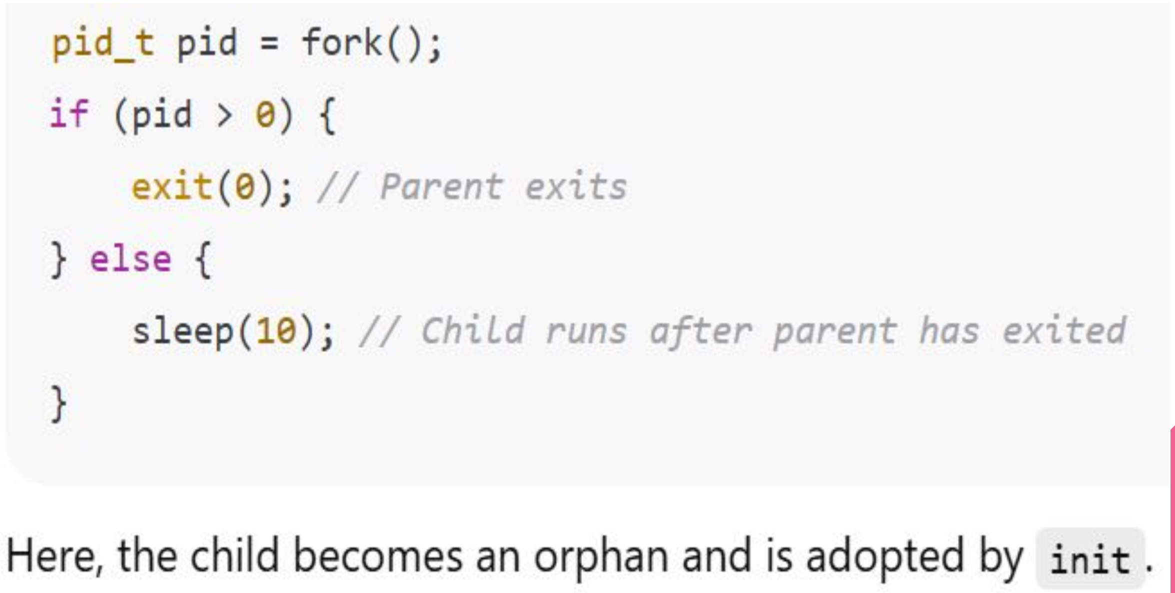
```c
pid_t pid = fork();
if (pid == 0) {
    // Child process
    exit(0); // exits immediately
} else {
    sleep(10); // Parent sleeps without calling wait()
}
```

Here, the child becomes a zombie for 10 seconds.

Orphan Process

An **Orphan Process** is a **child process whose parent has terminated** before the child finishes execution.

- The **init process (PID 1)** adopts the orphan process.
- The orphan continues to run normally.
- The OS ensures the child is not left unmanaged.



```c
pid_t pid = fork();
if (pid > 0) {
    exit(0); // Parent exits
} else {
    sleep(10); // Child runs after parent has exited
}
```

Here, the child becomes an orphan and is adopted by `init`.