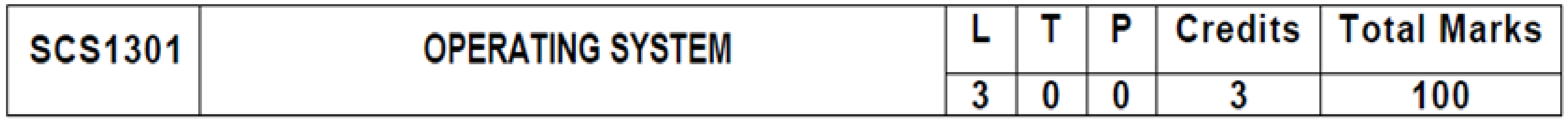
SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade "A" by NAAC | Approved by AICTE

SCS1301 - OPERATING SYSTEM

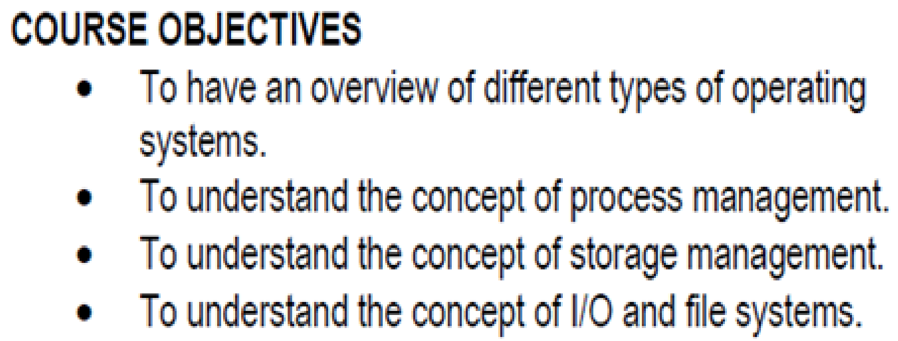
UNIT – 2

PROCESS MANAGEMENT

SYLLUBUS

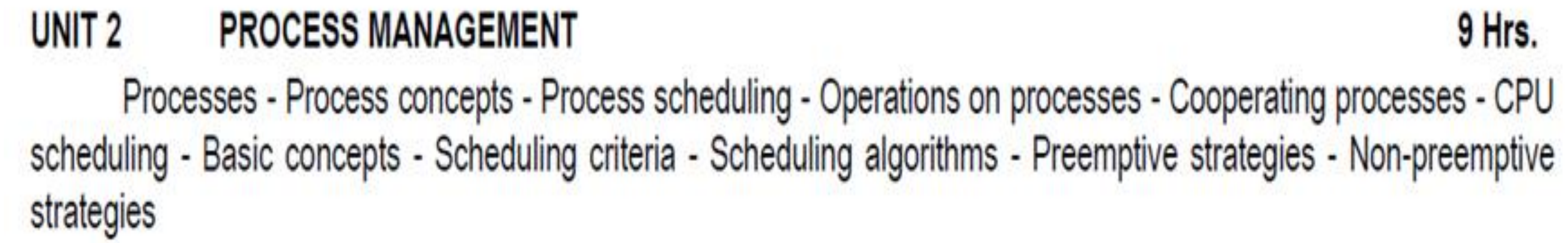


		L	T	P	Credits	Total Marks
SCS1301	OPERATING SYSTEM	3	0	0	3	100



COURSE OBJECTIVES

- To have an overview of different types of operating systems.
- To understand the concept of process management.
- To understand the concept of storage management.
- To understand the concept of I/O and file systems.



UNIT 2 PROCESS MANAGEMENT 9 Hrs.

Processes - Process concepts - Process scheduling - Operations on processes - Cooperating processes - CPU scheduling - Basic concepts - Scheduling criteria - Scheduling algorithms - Preemptive strategies - Non-preemptive strategies

COURSE OUTCOMES

SCS1301.1	Comprehend knowledge about Operating system components and services.
SCS1301.2	Apply knowledge of process scheduling algorithms for a given context.
SCS1301.3	Analyze process synchronization and deadlock conditions.
SCS1301.4	Construct the process of Mapping logical address to physical address.
SCS1301.5	Design appropriate strategies for Paging, Segmentation.
SCS1301.6	Develop real time applications Based on Linux shell programming.

INTRODUCTION TO PROCESSES

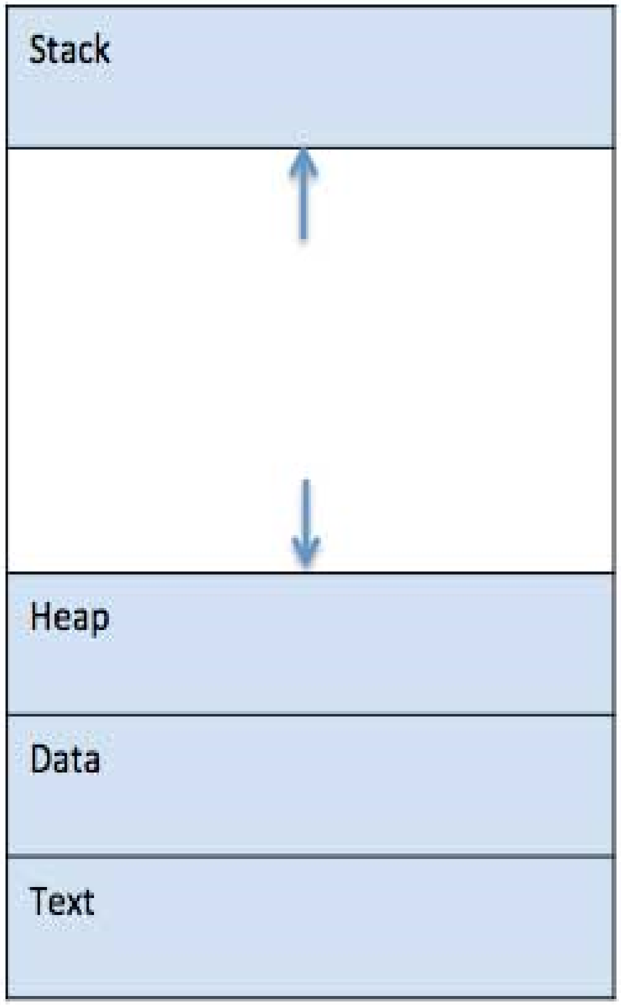
- Early computer systems allowed only one program to be executed at a time.
- **Definition :** A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.
- A process is the unit of work in most systems.

- Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
- A process is the unit of work in a modern time-sharing system.
- The more complex the operating system is, the more it is expected to do on behalf of its users.
- Systems consist of a collection of processes:
  - Operating-system processes execute system code, and
  - User processes execute user code.

PROCESSES

- A process is mainly a program in execution where the execution of a process must progress in a sequential order or based on some priority or algorithms.
- In other words, it is an entity that represents the fundamental working that has been assigned to a system.
- When a program gets loaded into the memory, it is said to as process. This processing can be categorized into 4 sections. These are:
  - Heap
  - Stack
  - Data
  - Text

PROCESSES



PROCESSES

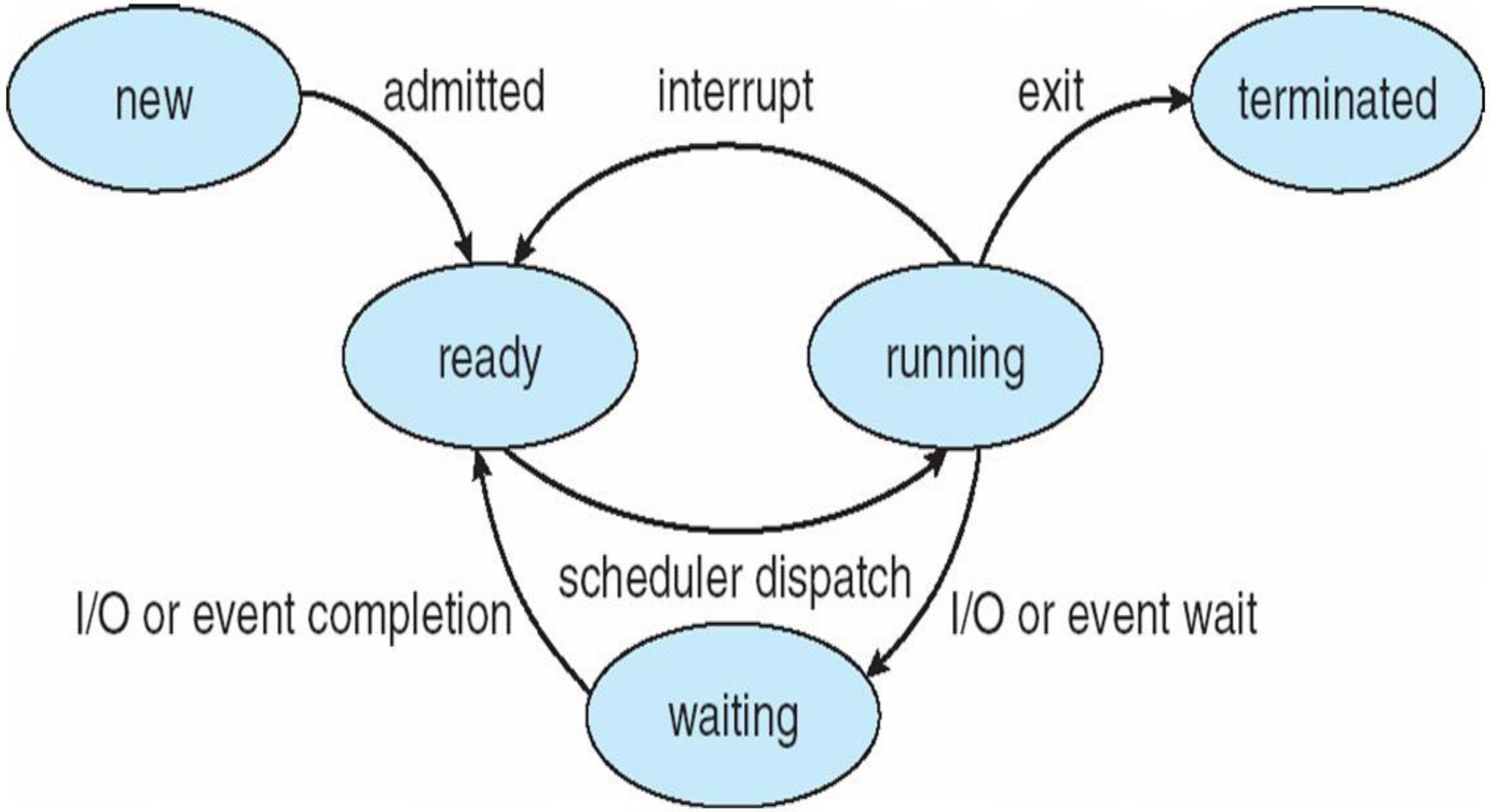
Process memory is divided into four sections for efficient working :

- **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

What is a Process?

- A process is a program in execution.
- Process is not as same as program code but a lot more than it.
- A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity.
- Attributes held by process include hardware state, memory, CPU etc.

PROCESS STATE / PROCESS LIFE CYCLE



PROCESS STATE

- When a process executes, it passes through different states.
- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur 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

PROCESS STATE

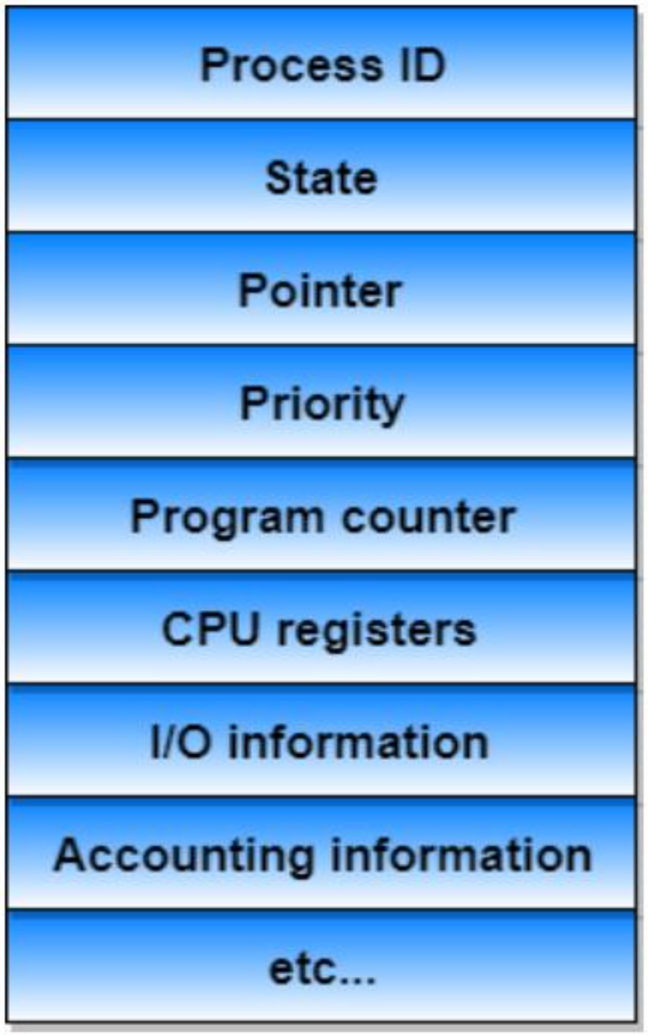
- **START / NEW**
  - This is the initial state when a process is first started / created.
- **READY**
  - The process is waiting to be assigned to a processor.
  - Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
  - Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
- **RUNNING**
  - Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

PROCESS STATE

- **WAITING**
  - Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **TERMINATED OR EXIT**
  - Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

These names are arbitrary, and they vary across operating systems.

PROCESS CONTROL BLOCK (PCB)



Information associated with each process
(also called **task control block**)

PCB

- There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:
- **Process ID**
  - Unique identification for each of the process in the operating system.
- **Process State**
  - It can be running, waiting etc.
- **Process Privileges**
  - This is required to allow/disallow access to system resources.

PCB

- **Pointer**
  - A pointer to parent process.
- **Program Counter**
  - Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU Registers**
  - Various CPU registers where process need to be stored for execution for running state.
- **CPU Scheduling Information**
  - Process priority and other scheduling information which is required to schedule the process.

PCB

- **Memory Management Information**
  - This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **Accounting Information**
  - This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO Status Information**
  - This includes a list of I/O devices allocated to the process.

PCB

- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. (shown in the diagram)
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

PROCESS SCHEDULING

- The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.
- **AIM :** The process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs.
- For achieving this, the scheduler must apply appropriate rules for swapping processes **IN** and **OUT** of CPU

PROCESS SCHEDULING

- Scheduling fell into one of the two general categories:
- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

What are Scheduling Queues?

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.
- The OS maintains all PCBs in Process Scheduling Queues.
- The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue.

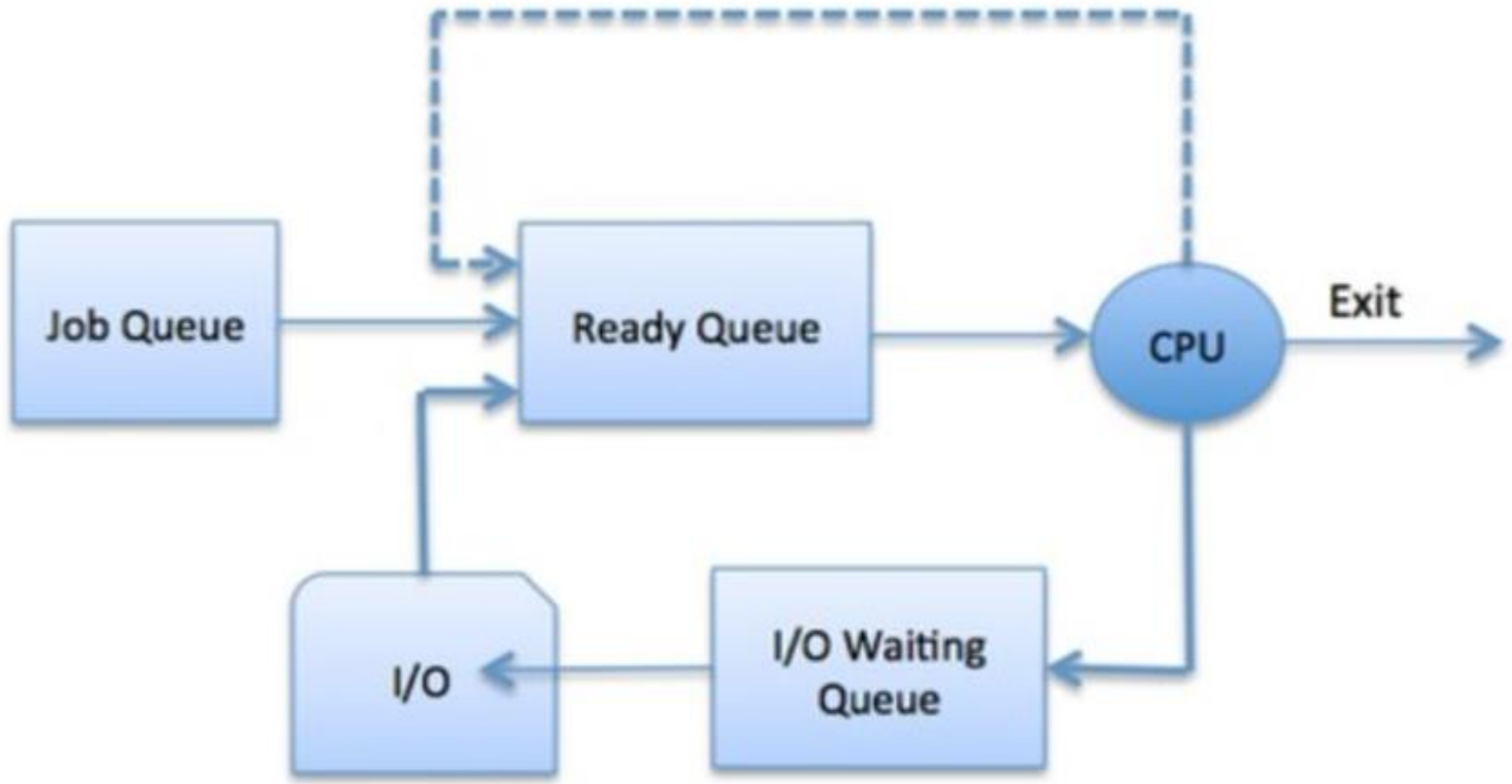
Scheduling Queues?

- Once the process is assigned to the CPU and is executing, one of the following several events can occur:
  - The process could issue an I/O request, and then be placed in the I/O queue.
  - The process could create a new sub-process and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Scheduling Queues?

- Process migration between the various queues:
  - Job queue
  - Ready queue
  - Device queues
- **JOB QUEUE –** This queue keeps all the processes in the system.
- **READY QUEUE –** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **DEVICE QUEUES –** The processes which are blocked due to unavailability of an I/O device constitute this queue.

Scheduling Queues?

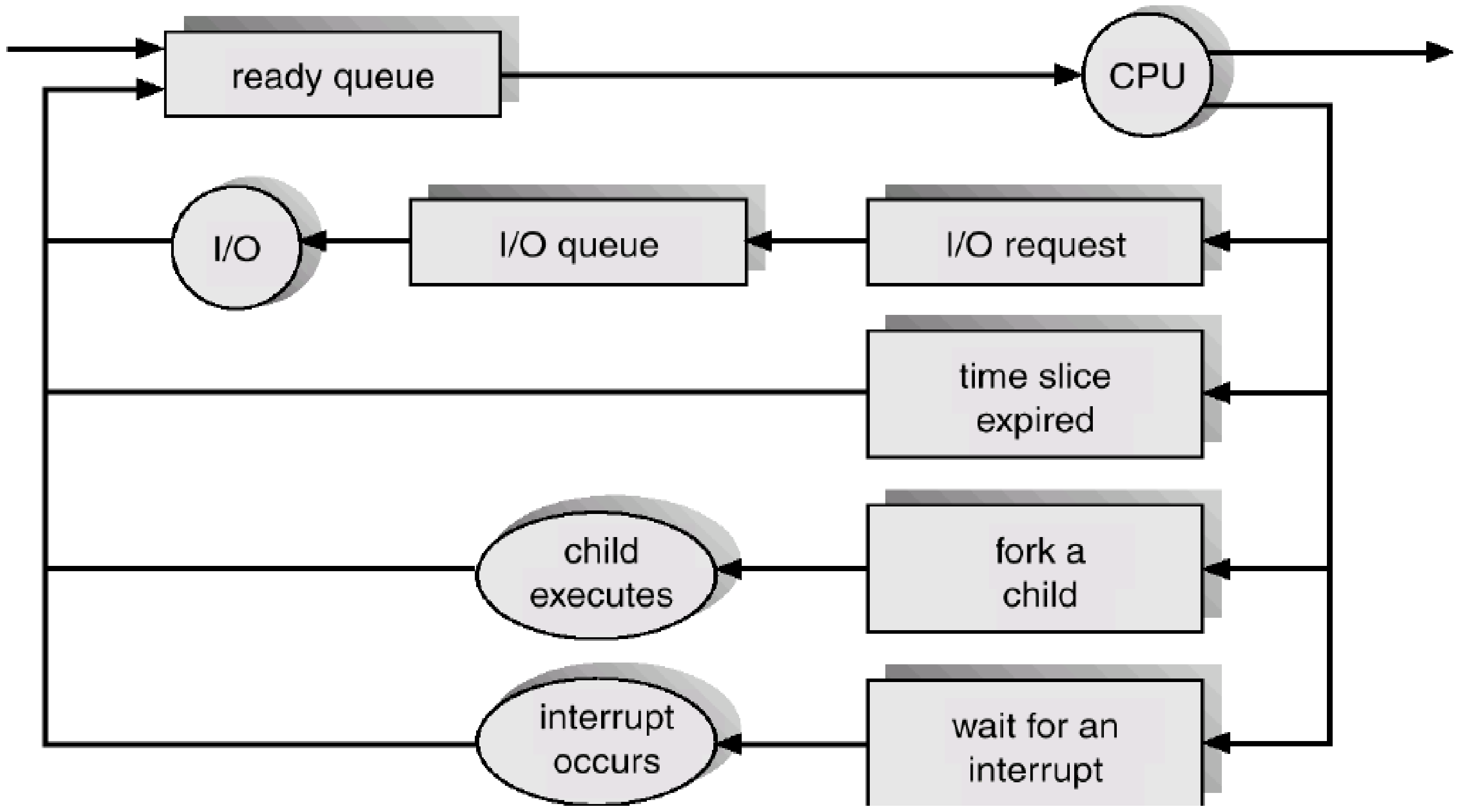


- The OS can use different policies to manage each queue **(FIFO, Round Robin, Priority, etc.)**

TWO STATE PROCESS MODEL

- Two-state process model refers to running and non-running states which are described below
- **RUNNING**
  - When a new process is created, it enters into the system as in the running state.
- **NOT RUNNING**
  - Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list.

QUEUING DIAGRAM REPRESENTATION FOR PROCESS SCHEDULING



PROCESS SCHEDULING

- Each rectangular box represents a queue.
- Two types of queues are present:
  - the ready queue and
  - a set of device queues
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or is dispatched.

SCHEDULERS

- Schedulers are special system software which handles the process scheduling in various ways.
- Their main task is to select the jobs to be submitted into the system and to decide which process to run.
- Schedulers are of three types –
  - Long-Term Scheduler
  - Short-Term Scheduler
  - Medium-Term Scheduler

SCHEDULERS

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

LONG TERM SCHEDULER

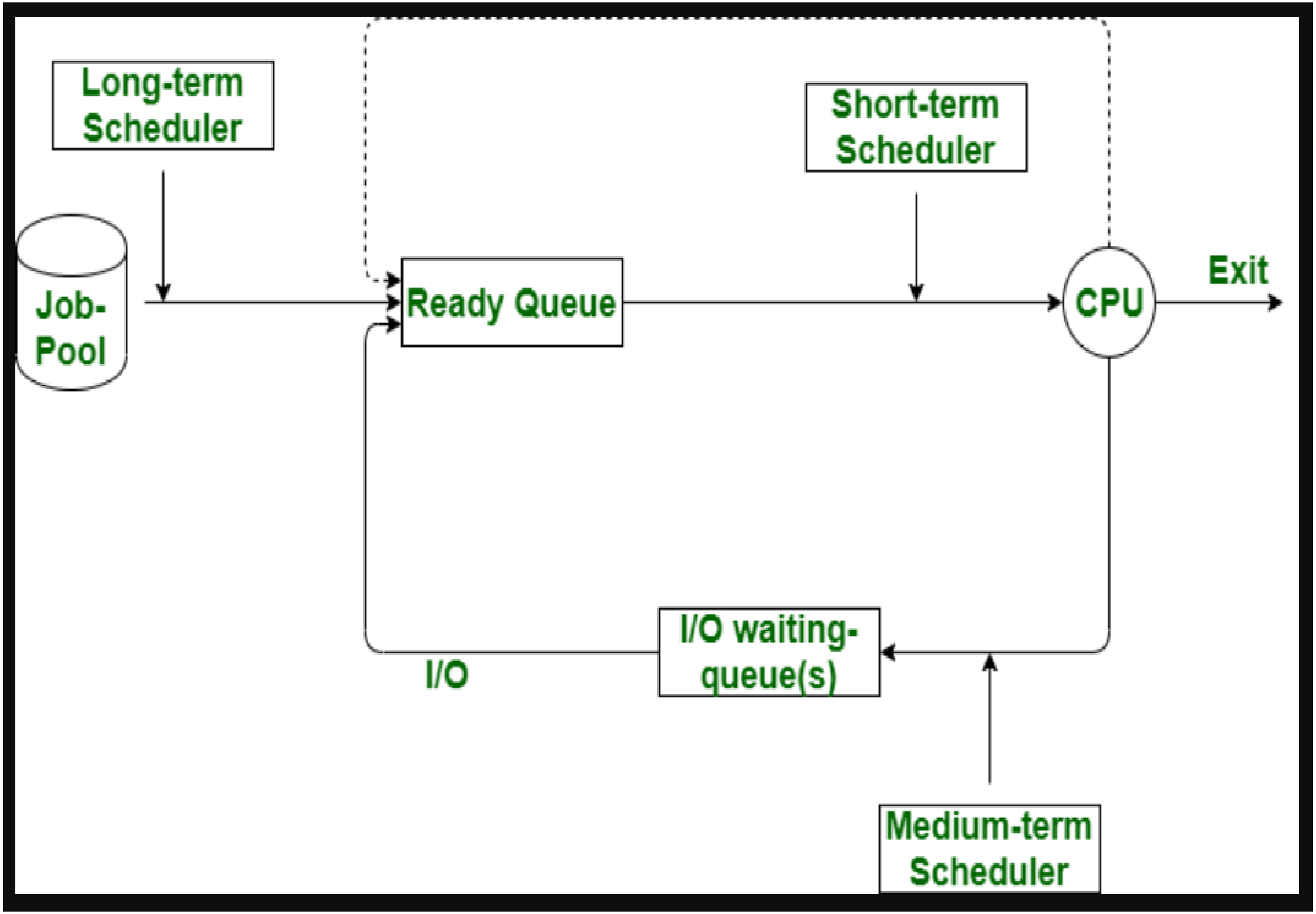
- It is also called a job scheduler.
- A long-term scheduler determines which programs are admitted to the system for processing.
- It selects processes from the queue and loads them into memory for execution.
- Process loads into the memory for CPU scheduling.

SHORT TERM SCHEDULER

- It is also called as CPU scheduler.
- Its main objective is to increase system performance in accordance with the chosen set of criteria.
- It is the change of ready state to running state of the process.
- CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next.

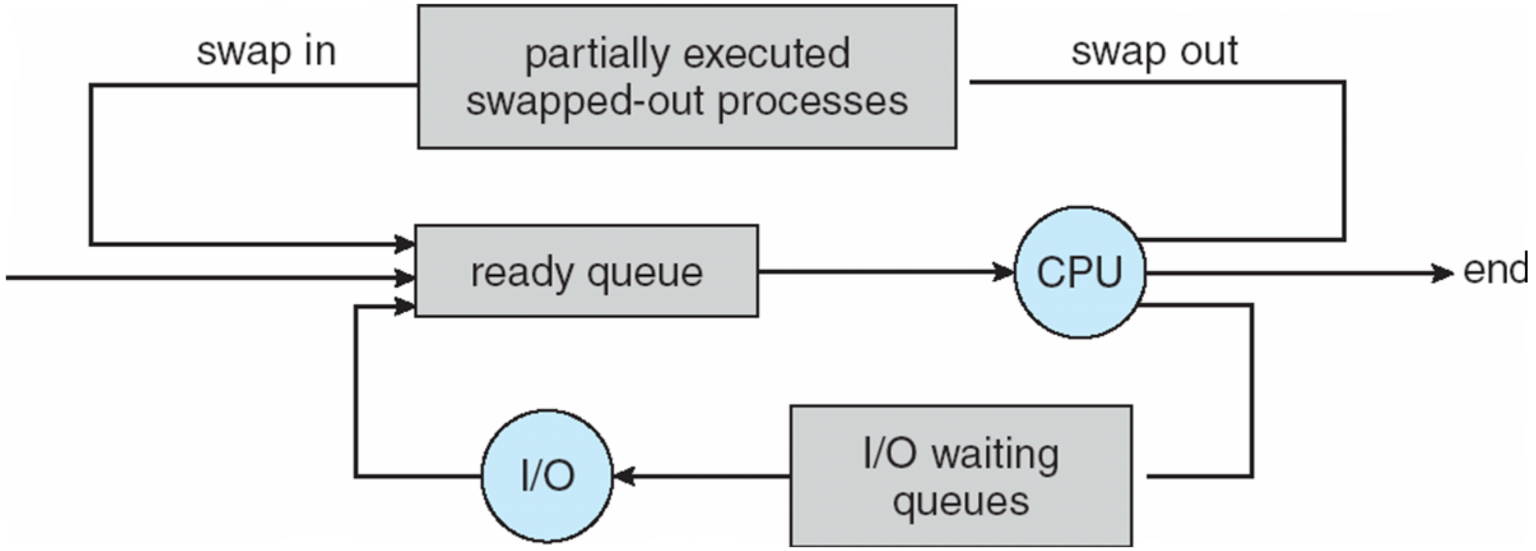
MEDIUM TERM SCHEDULER

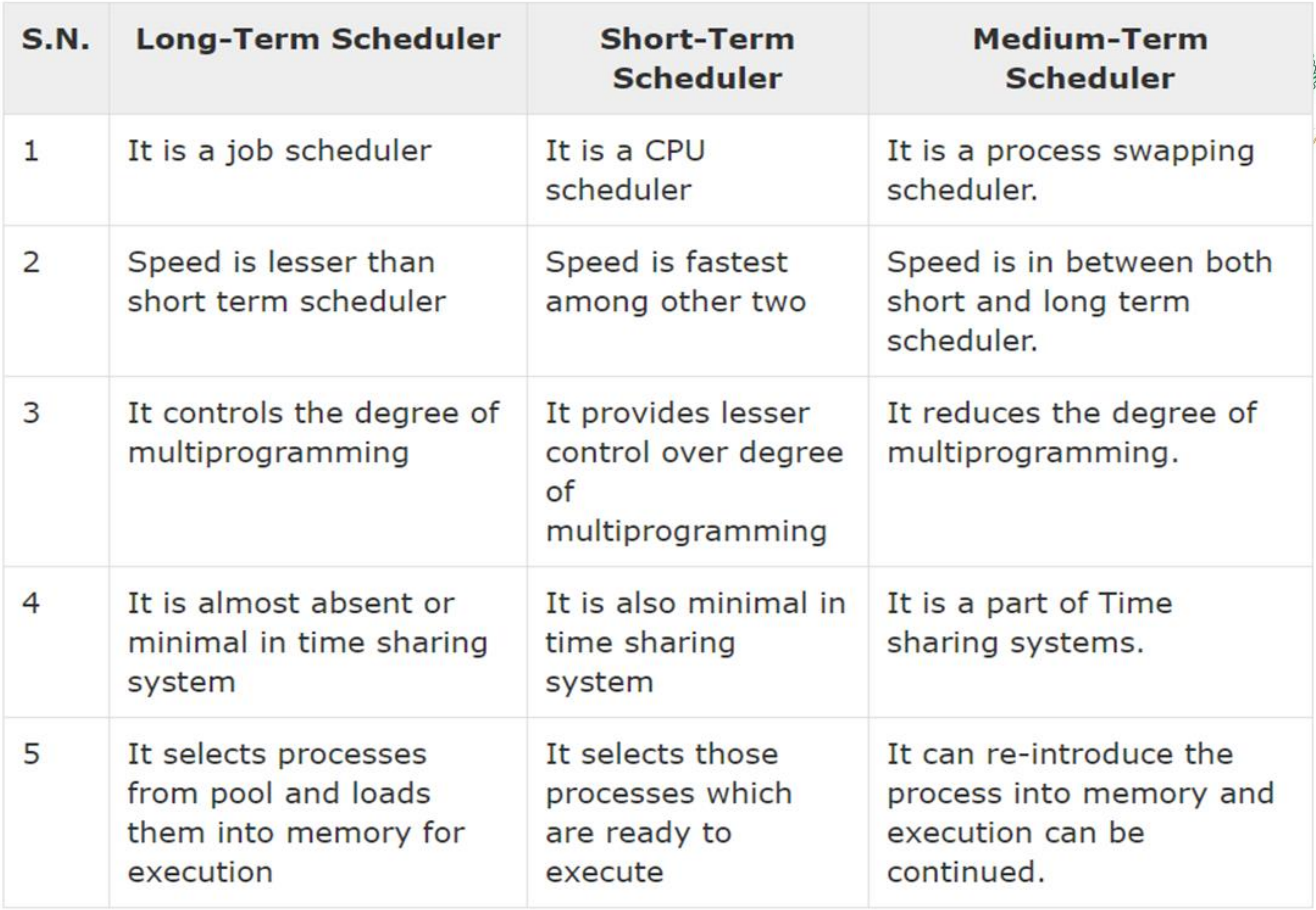
- Medium-term scheduling is a part of swapping.
- It removes the processes from the memory.
- The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request.
- A suspended process cannot make any progress towards completion.


Long-term Scheduler
Short-term Scheduler
Job-Pool
Ready Queue
CPU
Exit
I/O waiting-queue(s)
I/O
Medium-term Scheduler

Medium-term scheduler

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



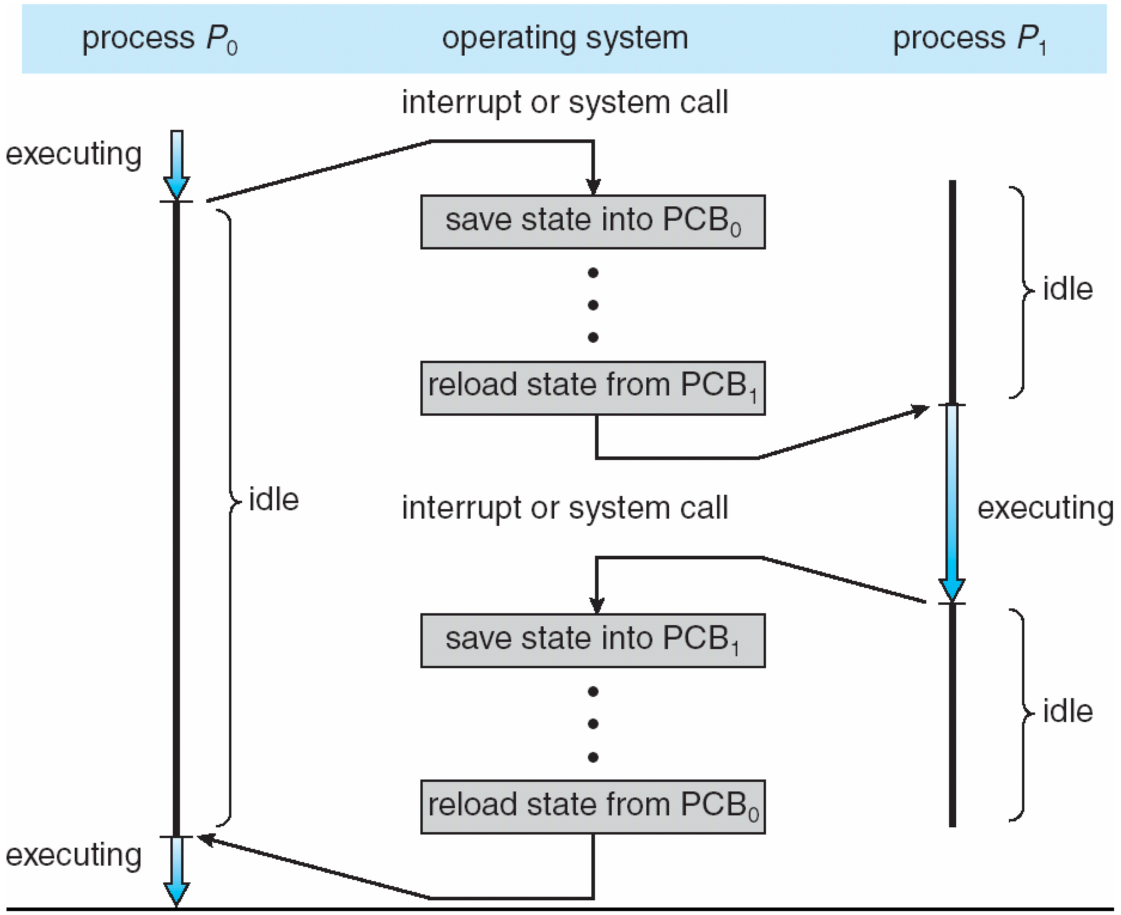


S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

CONTEXT SWITCH

- Switching the CPU to another process requires **saving** the state of the old process and **loading** the saved state for the new process. This task is known as a **Context Switch**.
- The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information.
- Context switch time is **pure overhead**, because the **system does no useful work while switching**. Its speed varies from machine to machine, depending on the memory speed.

CPU Switch From Process to Process



OPERATIONS ON PROCESSES

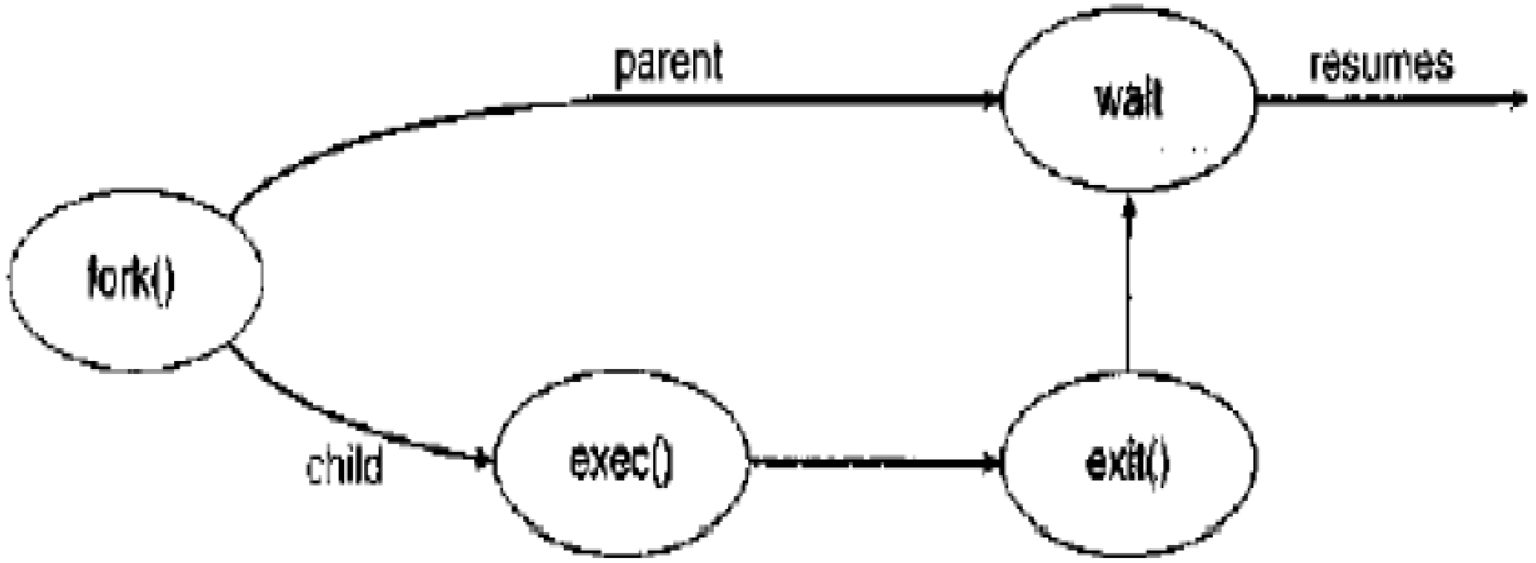
- The processes in the system can execute concurrently, and they must be created and deleted dynamically
- The two major operation **Process Creation** and **Process Termination.**

PROCESS CREATION

- Through appropriate system calls, such as fork or spawn, processes may create other processes.
- The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.
- Parent process creates 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.

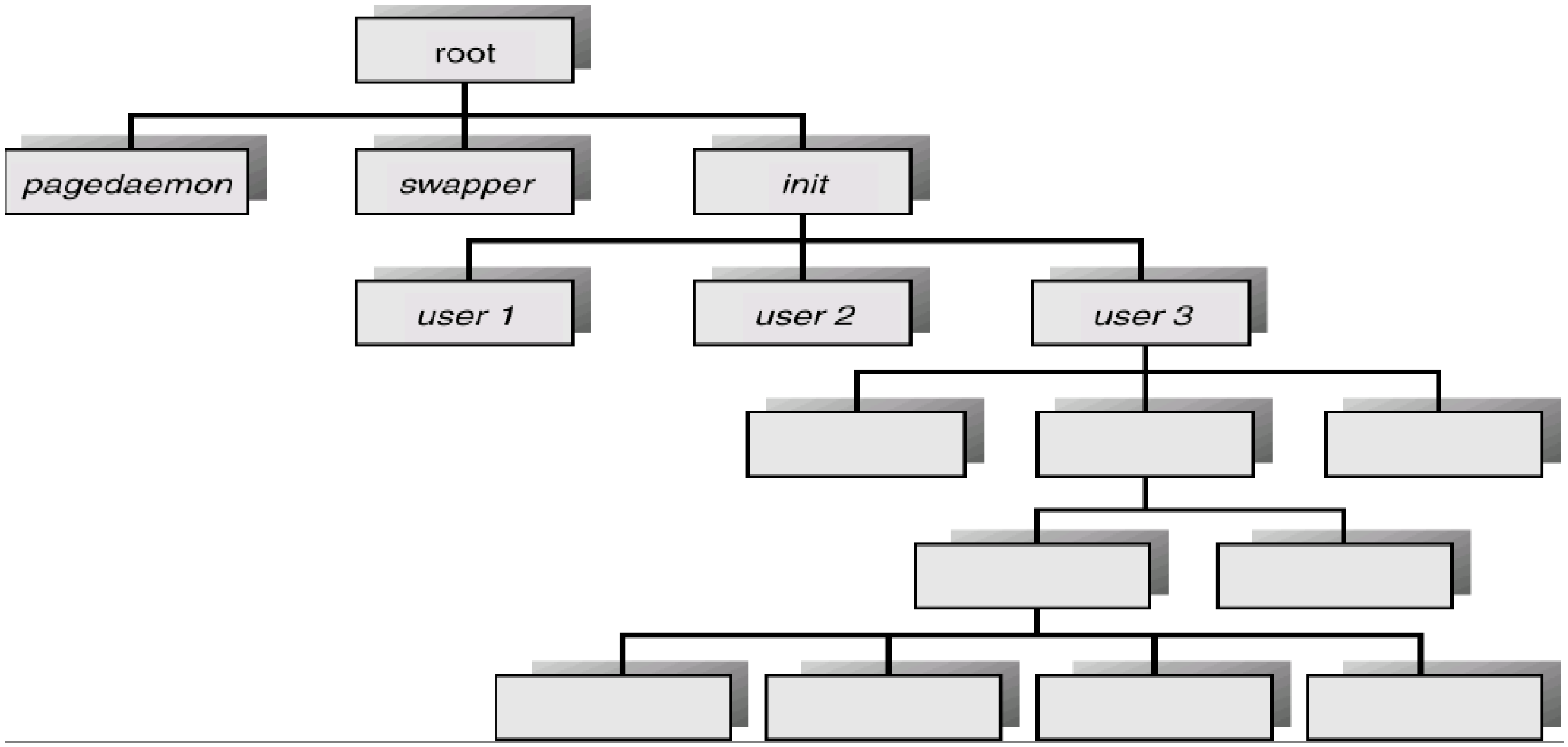
PROCESS CREATION

`fork()` system call creates new process
`exec()` system call used after a **`fork()`** to replace the process' memory space with a new program



Process creation

PROCESS CREATION



A Tree of Processes On A Typical UNIX System

PROCESS TERMINATION

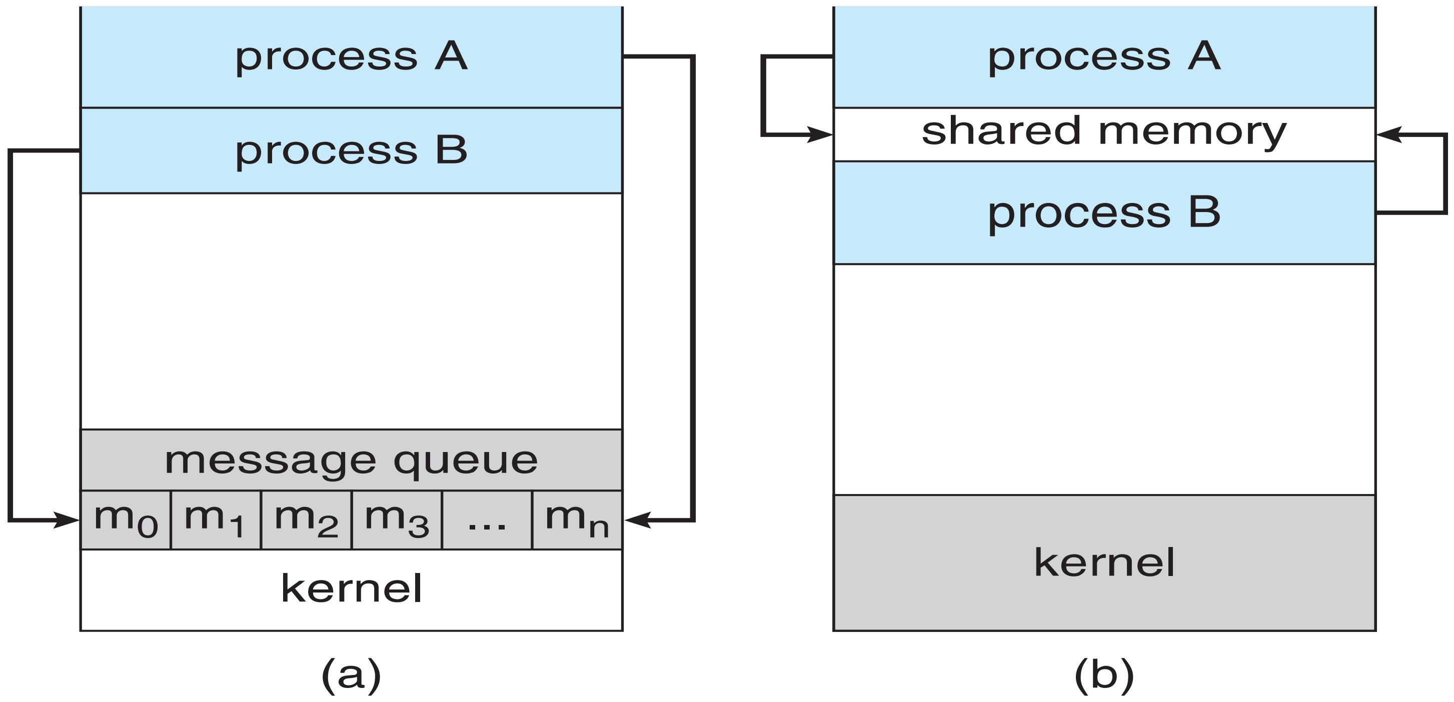
- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing (b) shared memory



COOPERATING PROCESSES

- There are several reasons for providing an environment that allows process cooperation:
  - **INFORMATION SHARING**
  - **COMPUTATION SPEEDUP**
  - **MODULARITY**
  - **CONVENIENCE**
  - **BUFFER**

COOPERATING PROCESSES

- **INFORMATION SHARING:**
  - Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **COMPUTATION SPEEDUP:**
  - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
- **MODULARITY:**
  - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
- **CONVENIENCE**
  - Even an individual user may work on many tasks at the same time

Producer-Consumer Problem

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

Bounded-Buffer – Shared-Memory Solution

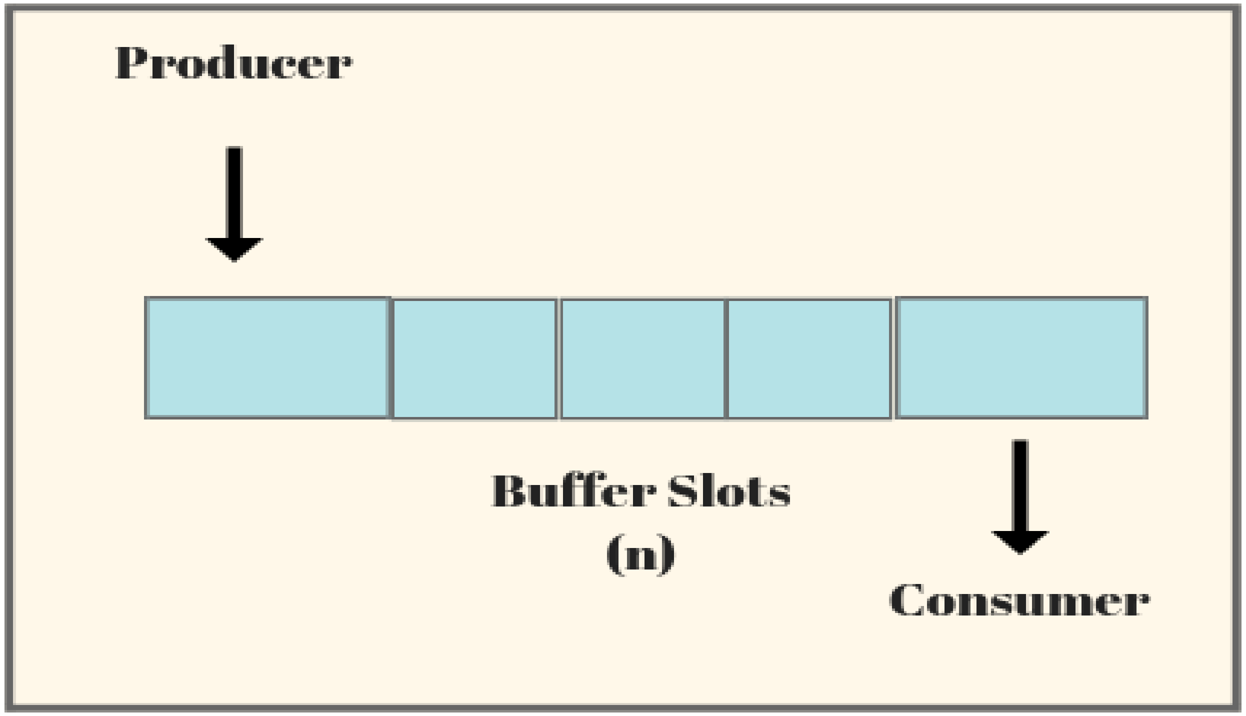
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

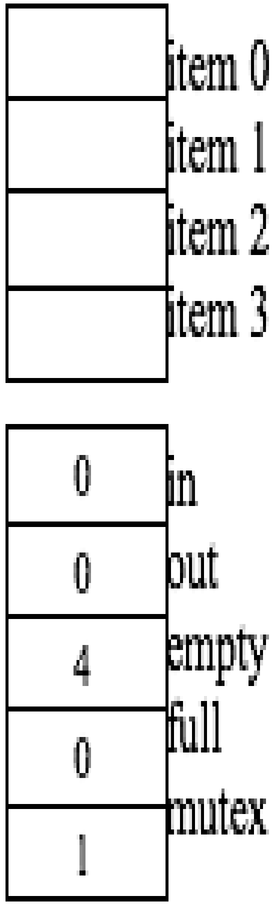
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded Buffer Problem



```
/* structure of P1 & P2*/
do {
    __
    produce an item in nextp
    __
    wait(empty)
    wait(mutex)
    buffer(in)= nextp
    in = (in + 1) % n
    signal(mutex)
    signal(full)
}while(1)
```

```
/* structure of P3 */
do {
    wait(full)
    wait(mutex)
    nextp = buffer(out)
    out = (out + 1) % n
    signal(mutex)
    signal(empty)
    ____..
    consume item in nextp
    ___..
}while(1)
```



Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next consumed 
*/
}
```

https://www.youtube.com/watch?v=LRiN3DJdskA

CPU SCHEDULING

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state)
- This is due to unavailability of any resource like I/O etc, thereby making full use of CPU.
- The aim of CPU scheduling is to make the system efficient, fast and fair.
- Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed.

CPU SCHEDULING: DISPATCHER

- Another component involved in the CPU scheduling function is the **Dispatcher**.
- The dispatcher is the module that gives control of the CPU to the process selected by the **short-term scheduler**.
- This function involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program from where it left last time.

CPU SCHEDULING: DISPATCHER

- The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**.
- Types of CPU Scheduling
  - When a process switches from the **running** state to the **waiting** state (for I/O request or invocation of wait for the termination of one of the child processes).
  - When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs).
  - When a process switches from the **waiting** state to the **ready** state (for example, completion of I/O).
  - When a process **terminates**.

CPU SCHEDULING: SCHEDULING CRITERIA

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- Many criteria have been suggested for comparing CPU scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.
  - **CPU UTILIZATION**
  - **THROUGHPUT**
  - **TURNAROUND TIME**
  - **WAITING TIME**
  - **LOAD AVERAGE**
  - **RESPONSE TIME**

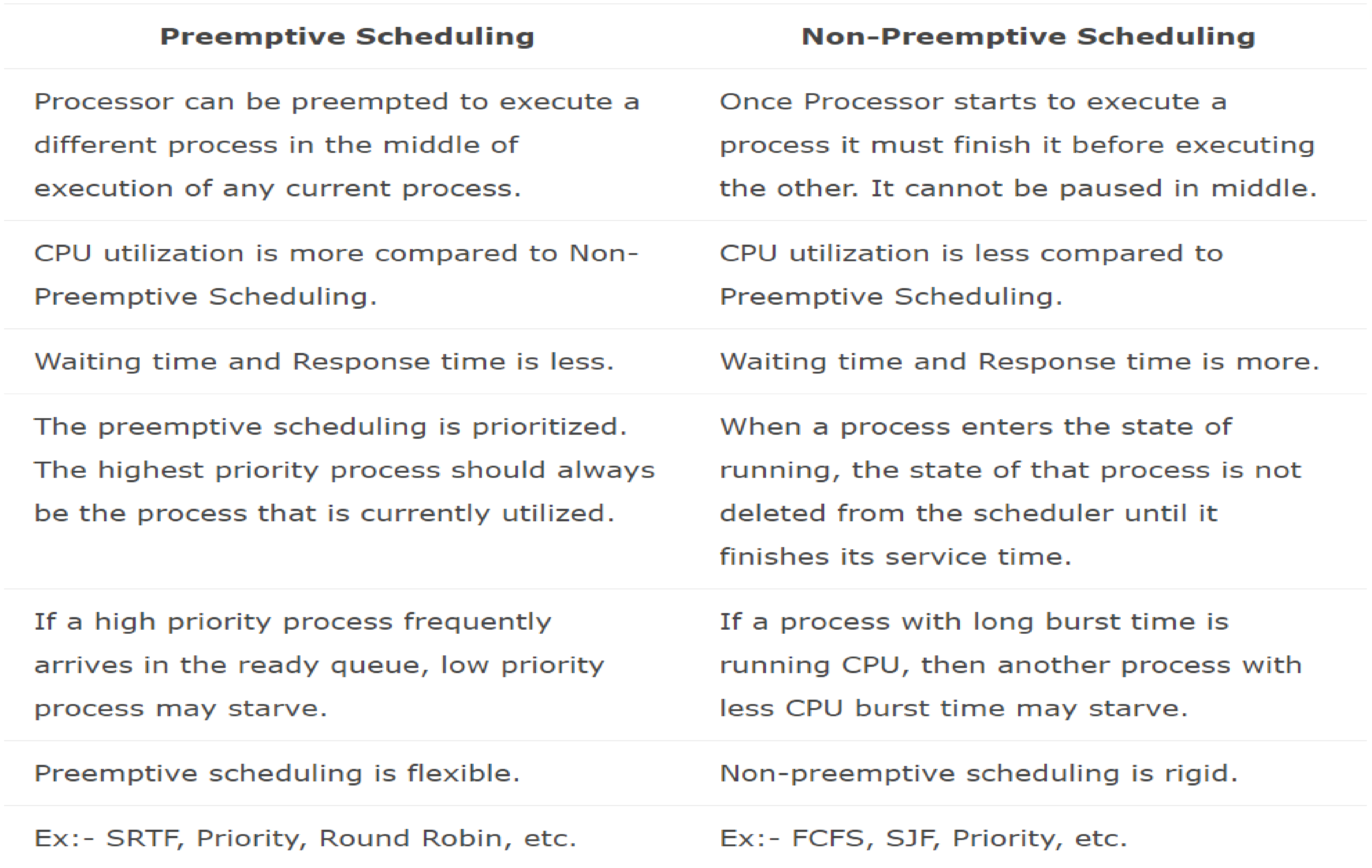
CPU SCHEDULING: SCHEDULING CRITERIA

- **CPU UTILIZATION**
  - We want to keep the CPU as busy as possible.
  - Conceptually, CPU utilization can range from 0 to 100 percent.
- **THROUGHPUT**
  - It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time.
- **TURNAROUND TIME**
  - It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process (Wall clock time).

CPU SCHEDULING: SCHEDULING CRITERIA

- **WAITING TIME**
  - The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- **LOAD AVERAGE**
  - It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- **RESPONSE TIME**
  - Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

PREEMPTIVE AND NON – PREEMPTIVE



Preemptive Scheduling	Non-Preemptive Scheduling
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more compared to Non-Preemptive Scheduling.	CPU utilization is less compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.	When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Ex:- SRTF, Priority, Round Robin, etc.	Ex:- FCFS, SJF, Priority, etc.

CPU Scheduling

- CPU scheduling is the basis of **Multiprogrammed** operating systems.
- By switching the CPU among processes, the operating system can make the **computer more productive**.
- However, the terms **Process scheduling** and **Thread scheduling** are often used interchangeably.

Basic concepts

- In a **single-processor system**, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to **maximize CPU utilization**.
- The idea of multiprogramming is relatively simple.
- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then just sits idle; all this waiting time is wasted; no useful work is accomplished.
- With multiprogramming, we try to use this time productively.
- Several processes are kept in memory at one time.

Basic concepts

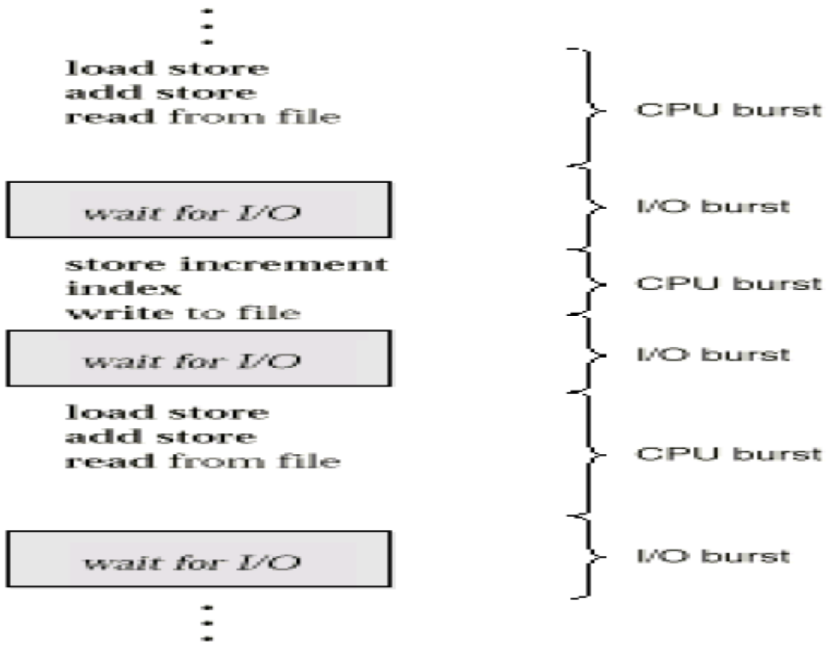
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.
- Every time one process has to wait, another process can take over use of the CPU.
- **Scheduling** of this kind is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.
- The **CPU** is, of course, one of the primary computer resources.
- Thus, its scheduling is central to **Operating System design**

CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes:

- Process execution consists of a cycle of **CPU execution and I/O wait**.
- Processes alternate between these two states. Process execution begins with a CPU burst.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution (Fig a).
- The durations of CPU bursts have been measured extensively.

Alternating sequence of CPU and I/O bursts



CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler** (or CPU scheduler).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- It can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally **process control blocks (PCBs)** of the processes.

Preemptive Scheduling

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

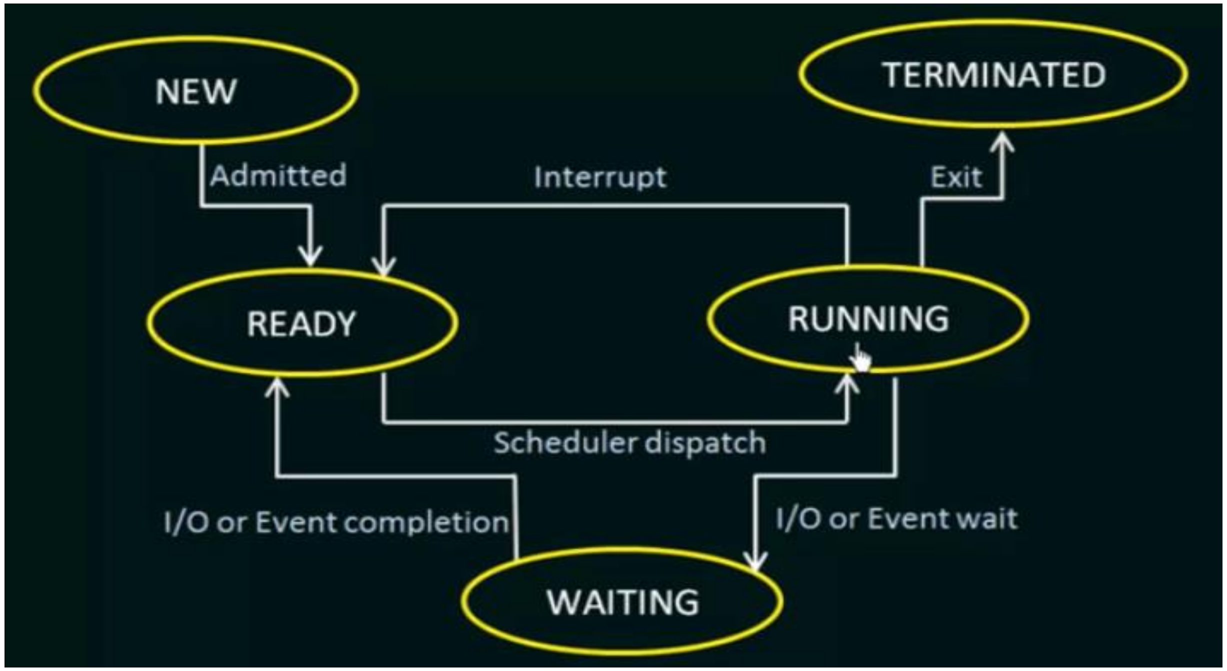
i. When a process switches from the **running state to the waiting state** (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)**.**

ii. When a process switches from the **running state to the ready state** (for example, when an interrupt occurs).

iii. When a process switches from the **waiting state to the ready state** (for example, at completion of I/O).

iv. When a process terminates.

Process States



Preemptive Scheduling

- 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.
- There is a choice, however, for situations **2** and **3**.
- When scheduling takes place only under circumstances **1** and **4**, we say that the scheduling scheme is **non preemptive** or **cooperative;** otherwise, it is **preemptive.**
- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Preemptive Scheduling

- This scheduling method is used by the Microsoft Windows 3.1 and Apple Macintosh Operating systems.
- Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.
- Unfortunately, preemptive scheduling incurs a cost associated with access to shared data.
- Consider the case of two processes that share data.
- While one is updating the data, it is preempted so that the second process can run.
- The second process then tries to read the data, which are in an inconsistent state,
- In such situations, we need new mechanisms to coordinate access to shared data.

Dispatcher

- Another component involved in the CPU-scheduling function is the **Dispatcher.**
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves :
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the **Dispatch latency.**

Scheduling Criteria

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.
- The criteria include the following:

CPU utilization

- We want to keep the **CPU as busy as possible**.
- Conceptually, CPU utilization can range from 0 to 100 percent.
- In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput

- If the CPU is busy executing processes, then work is being done.
- One measure of work is the **number of processes that are completed per time unit**, called throughput.
- For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

Turnaround time

- From the point of view of a particular process, the important criterion is how long it takes to execute that process.
- The interval from the **time of submission of a process to the time of completion** is the *turnaround time.*
- Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time

- The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
- *Waiting time* is the **sum of the periods spent waiting in the ready queue**.

Response time

- In an interactive system, turnaround time may not be the best criterion.
- Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
- Thus, another measure is the time from the submission of a request until the first response is produced.
- This measure, called *response time,* is the **time it takes to start responding**, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

Conclusion

- It is desirable to **maximize CPU utilization** and **throughput** and to **minimize turnaround time, waiting time, and response time**. In most cases, we optimize the average measure.
- However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.
- For example, to guarantee that all users get good service, we may want to minimize the maximum response time.
- Some analysts have suggested that, for interactive systems (such as timesharing systems), it is more important to minimize the *variance* in the response time than to minimize the average response time.

Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU scheduling algorithms:

i. First-come, First served scheduling
ii. Shortest-Job-First scheduling
iii. Shortest-remaining-time-first scheduling
iv. Priority scheduling
v. Round-Robin scheduling
vi. Multilevel Queue scheduling
vii. Multilevel Feedback Queue scheduling

First-come, First served scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.**

- The **FCFS** scheduling algorithm is **non preemptive**.
- With this scheme, **the process that requests the CPU first is allocated the CPU first**.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- The code for FCFS scheduling is **simple to write and understand**.
- The **average waiting time under the FCFS policy, however, is often quite long**.

First-come, First served scheduling

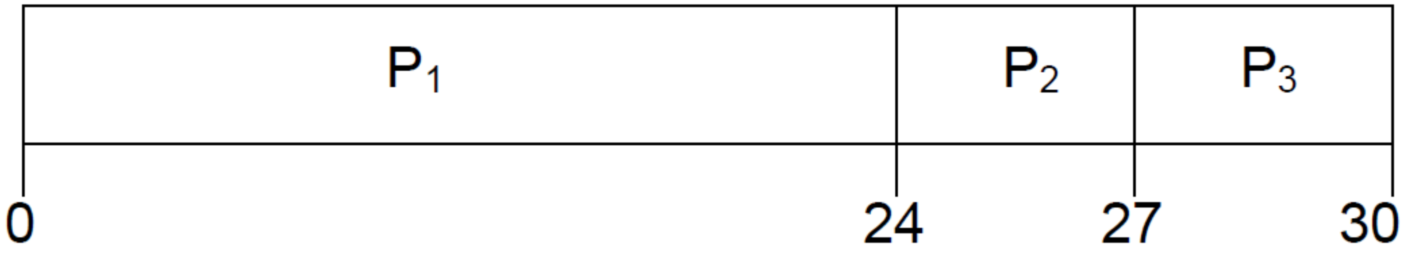
Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Example 1:

Process	**Burst Time**
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order : *P1 , P2 , P3 and are served in FCFS order,*

The **Gantt Chart** for the schedule is:



Waiting time:

- The waiting time is 0 milliseconds for process P1,
- 24 milliseconds for process *P2,* and
- 27 milliseconds for process *P3.*
- Thus, the **average waiting time** is (0 + 24 + 27)/3 = 17 milliseconds.

Turnaround time:

- Turnaround time for P1 = 24
- Turnaround time for P2 = 24 + 3
- Turnaround time for P3 = 24 + 3 + 3

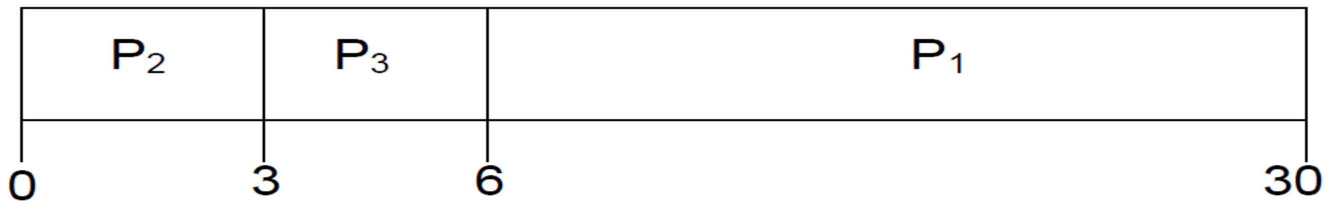
- Average Turnaround time = (24+24+3+24+3+3) / 3
  = 27 milliseconds.

Example 2:

- If the processes arrive in the order *P2,* P3, P1.

Process	**Burst Time**
P2	3
P3	3
P1	24

- The Gantt chart for the schedule is:



Waiting time:

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- **Average waiting time**: $(6 + 0 + 3)/3 = 3$milliseconds.

Turnaround time:

- Turnaround time for P2 = 3
- Turnaround time for P3 = 3 + 3
- Turnaround time for P1 = 3 + 3+24

First-come, First served scheduling

- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Shortest-Job-First scheduling

- A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.**
- 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.
- Note that 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.
- We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

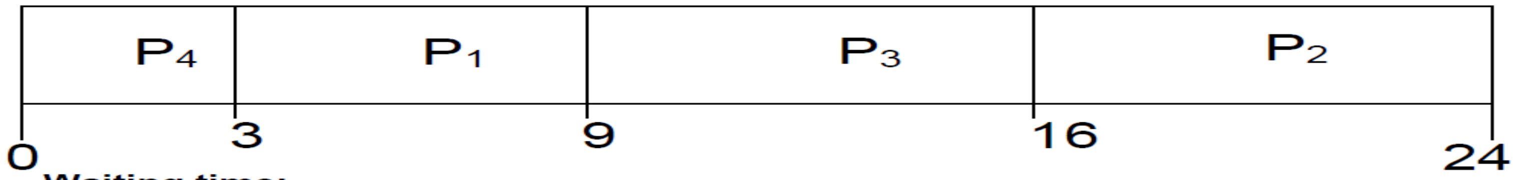
Shortest-Job-First scheduling

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Example:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt chart:



Waiting time:

- The waiting time is 3 milliseconds for process *P1,*
- 16 milliseconds for process *P2,*
- 16 milliseconds for process *P3,* and
- 0 milliseconds for process P4.
- Thus, the ***average waiting time*** =(3 + 16 + 9 + 0)/4
  = 7 milliseconds.

Turnaround time:

- Turnaround time for P1 = 9
- Turnaround time for P2 = 24
- Turnaround time for P3 = 16
- Turnaround time for P4 = 3
- Average Turnaround time = (9+24+16+3)/ 4
  = 13 milliseconds.

Shortest-Job-First scheduling

- The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the *average* waiting time decreases. The SJF algorithm can be either **preemptive** or **nonpreemptive**.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Shortest- Remaining-Time- First

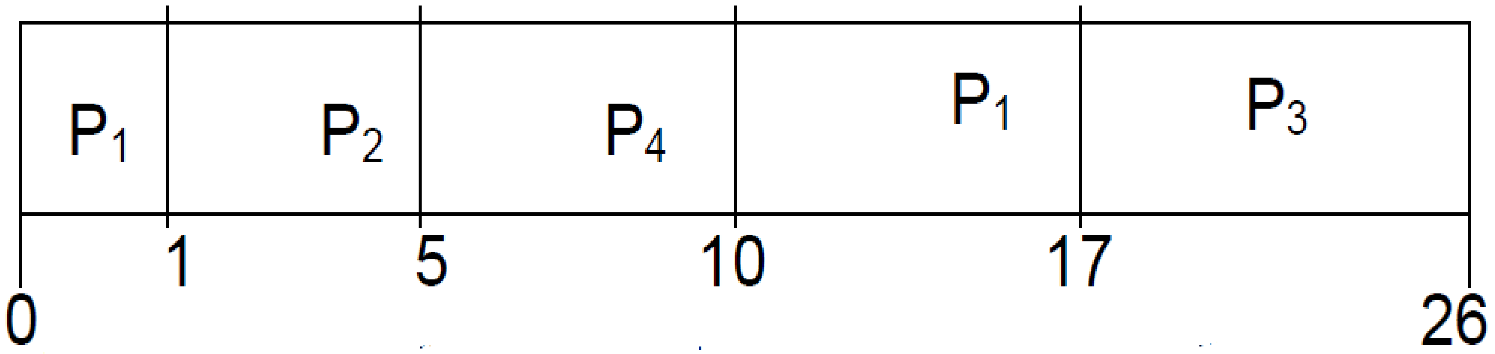
Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**.
As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Example:

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

Example:

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



Shortest- Remaining-Time- First

- Process *P1* is started at time 0, since it is the only process in the queue
- Process *P2* arrives at time 1.
- Among *P1* and *P2* as P2 is having shortest time, P2 will get executed first.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- After completion of P2, P1 is scheduled.
- And then in the last P4 which is having larger time is sheduled.

Waiting Time=Last entry into the CPU – Arrival Time – How much time it spend in CPU before last entry

Shortest- Remaining-Time- First

Waiting time:

- Waiting time for process p1 = (10 - 1)
- Waiting time for process p2 = (1-1)
- Waiting time for process p3 = (17-2)
- Waiting time for process p4 = (5-3)

- The average waiting time = ((10 - 1) + (1 - 1) + (17-2) + (5-3))/4
  = 26/4
  = 6.5 milliseconds.

- Nonpreemptive **SJF** scheduling would result in an average waiting time of 7.75 milliseconds.

Turnaround time:

- Turnaround time for P1 = (0+8)=8
- Turnaround time for P2 = (7 + 4)
- Turnaround time for P3 = (15+9)
- Turnaround time for P4 = (9 + 5)

- Average Turnaround time = (8+11+24+14) / 4
  = 14.25 milliseconds.

Priority scheduling

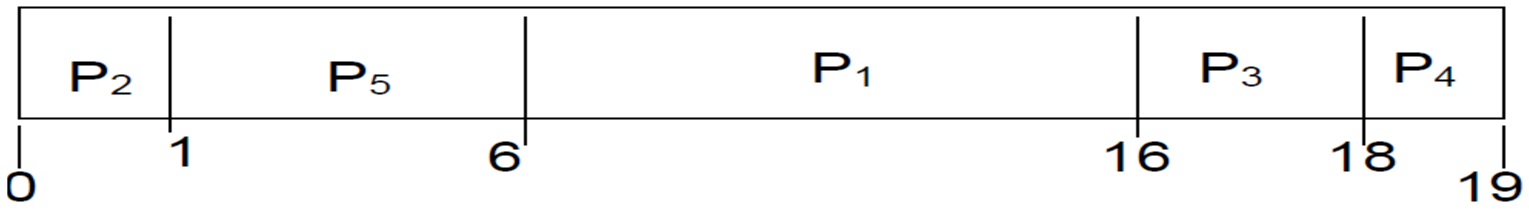
- The SJF algorithm is a special case of the general **priority scheduling algorithm.**
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.
- The **larger the CPU burst, the lower the priority**, and vice versa.
- Note that we discuss scheduling in terms of *high* priority and *low* priority.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion.
- Here, we assume that low numbers represent high priority.

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

Example:

Process	**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:



Priority scheduling

- Process *P2* is started at time 0, since it is having the highest priority among the processes in the queue.
- After execution of P2 process, process *P5* is started at time 1.
- Then *P1,P3* and *P4* as P2 processes are executed. is having shortest time, P2 will get executed first.

Waiting time:

Waiting time for process p1 = 6
Waiting time for process p2 = 0
Waiting time for process p3 = 16
Waiting time for process p4 = 18
Waiting time for process p5 = 1

The average waiting time = (6+0+16+18+1)/5
= 41/5
= 8.2 milliseconds.

Priority scheduling

Turnaround time:

Turnaround time for P1 = 6 + 10
Turnaround time for P2 = 0 + 1
Turnaround time for P3 = 16 + 2
Turnaround time for P4 = 18 + 1
Turnaround time for P5 = 1 + 5

Average Turnaround time = (16+1+18+19+6) / 5

= 12 milliseconds

Priority scheduling

- Priorities can be defined either *internally* or *externally*.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
- External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Priority scheduling can be either **preemptive** or **nonpreemptive.**
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- *preemptive* priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A *nonpreemptive* priority scheduling algorithm will simply put the new process at the head of the ready queue.

Priority scheduling

- A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.**
- A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade "A" by NAAC | Approved by AICTE

Priority scheduling

- A solution to the problem of indefinite blockage of low-priority processes is **aging.**
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.
- In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

Round-Robin Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a **time quantum** or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds.
- *The ready queue is treated as a circular queue*.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.

Round-Robin Scheduling

- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen:

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

  Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.
  - A *context switch* will be executed, and the process will be put at the **tail** of the ready queue.
  - The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long.

Round-Robin Scheduling

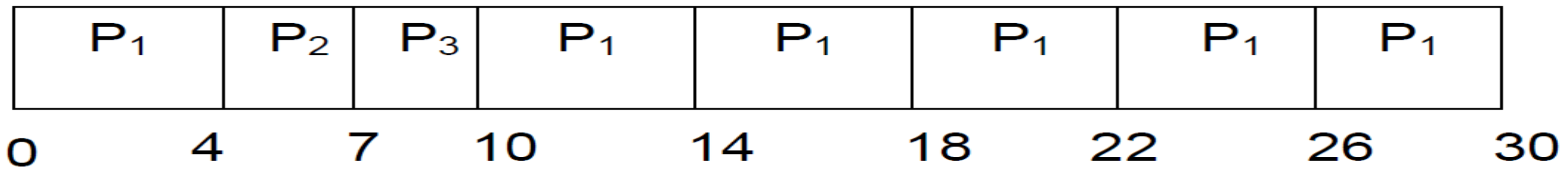
Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Example 1:

Process	**Burst Time**
P1	24
P2	3
P3	3

Time quantum = 4 ms

The Gnatt chart is:



Round-Robin Scheduling

- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2*
- Since process P2 does not need 4 milliseconds, it quits before its time quantum expires
- The CPU is then given to the next process, process P3
- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is:

Waiting time:

- Waiting time for process p1 = (10 - 4)
- Waiting time for process p2 = 4
- Waiting time for process p3 = 7
- The average waiting time = ((10-4)+4+7) /3

= 17/3 = 5.66 milliseconds.

Round-Robin Scheduling

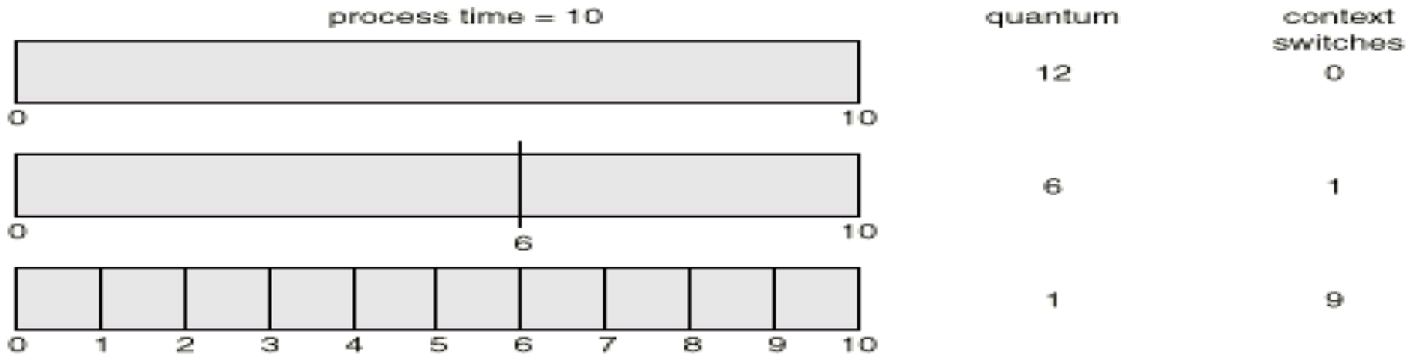
Turnaround time:

- Turnaround time for P1 = (10-4) + 24
- Turnaround time for P2 = 4 + 3
- Turnaround time for P3 = 7 + 3
- Average Turnaround time = (30+7+10) / 3

= 15.6 milliseconds

- ➢ The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy
- ➢ If the time quantum is extremely small (say, 1 millisecond), the RR approach is called **processor sharing**

Round-Robin Scheduling-Showing how a smaller time quantum increases context switches



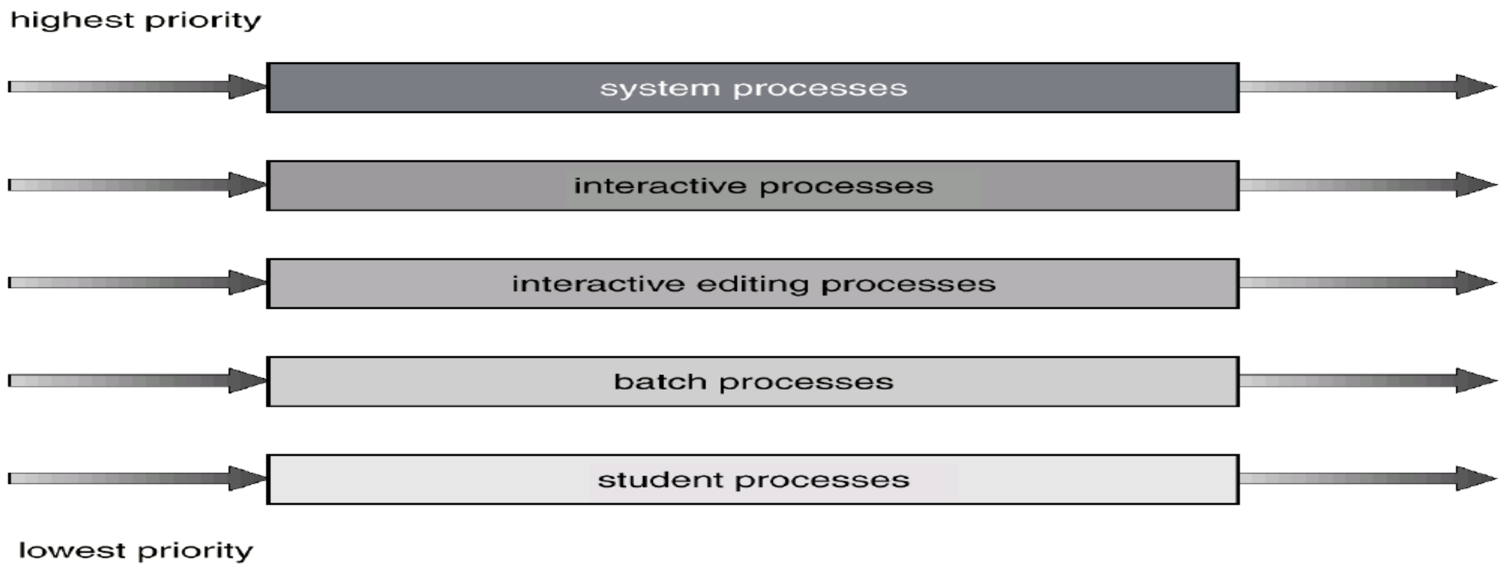
Multilevel Queue Scheduling

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- A **multilevel 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
- Each queue has its own scheduling algorithm.

Multilevel Queue Scheduling

- For example, separate queues might be used for foreground and background processes
  - The foreground queue might be scheduled by an **RR** algorithm,
  - while the background queue is scheduled by an **FCFS** algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
- For example, the foreground queue may have absolute priority over the background queue.

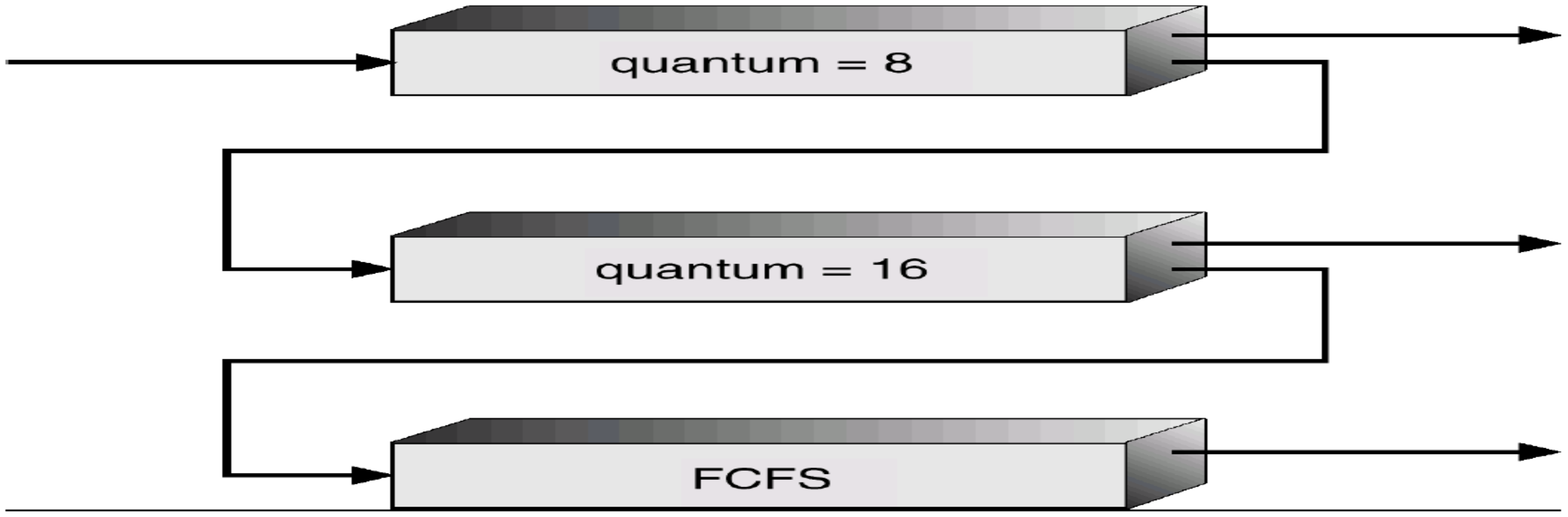
Multilevel Queue Scheduling



Multilevel Feedback Queue Scheduling

- Normally, in the multilevel queue scheduling algorithm, processes are permanently assigned to a queue when they enter to the system.
- Processes do not move between queues.
- For example, if there are separate queues for foreground and background processes, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
- The **multilevel feedback-queue scheduling algorithm,** in contrast, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of *aging* prevents **starvation.**

Multilevel Feedback Queue Scheduling



Multilevel feedback queues

Multilevel Feedback Queue Scheduling

For example, consider a multilevel feedback-queue scheduler with ***three queues***, numbered from 0 to 2 (Figure).

- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process that arrives for queue 0 will in turn, preempt a process queue 1.

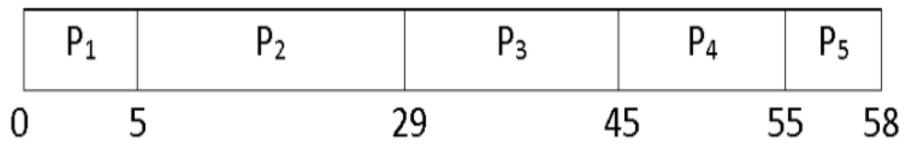
Multilevel Feedback Queue Scheduling

- A process entering the ready queue is put in queue 0.
- A process in queue 0 is given a time quantum of 8 milliseconds.
- If it does not finish within this time, it is moved to the tail of queue 1.
- If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
- If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.
- Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
- Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

FCFS Scheduling-Arrival Time =0ms

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

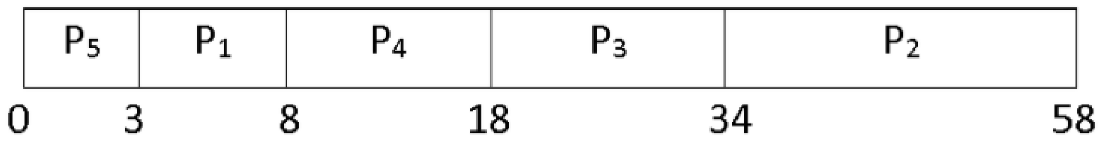
FCFS Scheduling



SJF Scheduling-Arrival Time=0ms

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

SJF Scheduling

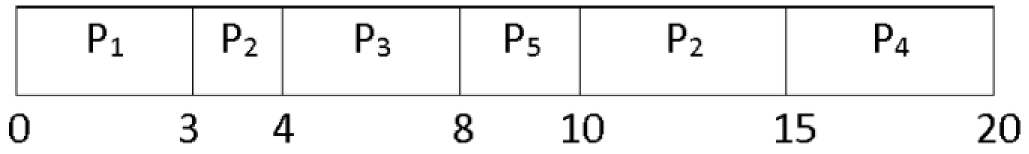


SRTF Scheduling

Arrival time	0	1ms	2ms	3ms	4ms	5ms	6ms	7ms	8ms
P1	3	2	1	0					
P2			6	6	5	5	5	5	5
P3					4	3	2	1	0
P4							5	5	5
P5									2

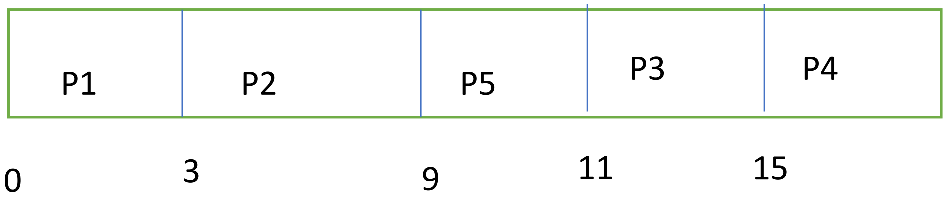
Process	Burst Time(CPU)	Arrival Time(ms)
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8

SRTF Scheduling



Non Preemptive SJF Scheduling

Arrival time	0	1ms	2ms	3ms	4ms	5ms	6ms	7ms	8ms	9ms
P1	3	2	1	0						
P2			6	6	5	4	3	2	1	0
P3					4	4	4	4	4	4
P4							5	5	5	5
P5									2	2

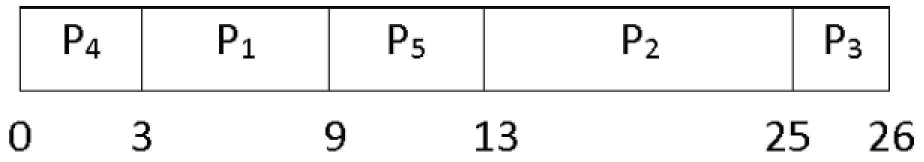


Process	Burst Time(CPU)	Arrival Time(ms)
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8

Priority Scheduling-Arrival Time=0ms

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

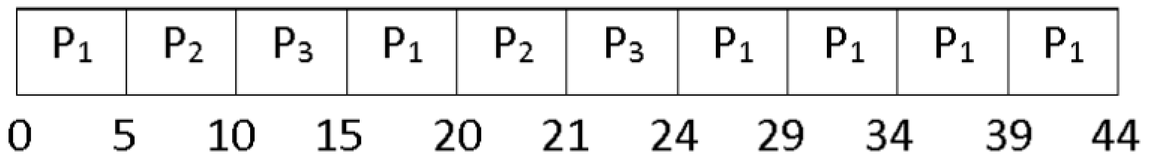
Priority Scheduling



Round Robin Scheduling-Time Quantum=5ms

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Round Robin Scheduling


P_1
P_2
P_3
P_1
P_2
P_3
P_1
P_1
P_1
P_1
0
5
10
15
20
21
24
29
34
39
44