Unit II (8 Hrs)

Operating Systems : Methodologies for implementation of O/S service system calls,

system programs, Interrupt mechanisms.

Process - Concept of process and threads, Process states, Process management, Context switching

Interaction between processes and OS Multithreading

Process Control, Job schedulers, Job Scheduling,

scheduling criteria, scheduling algorithms

A **system call** is how a program requests a service from an operating system's kernel.

This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling.

System calls provide an essential interface between a process and the operating system.

System calls can be roughly grouped into five major categories:

1. Process Control

    a) load b) execute c) end, abort d) create process

    e) terminate process f) get/set process attributes

    g) wait for time, wait event, signal event

    h) allocate, free memory

2. File management

   a) create file, delete file b) open, close c) read, write, reposition

   d) get/set file attributes

3. Device Management

   a) request device, release device b) read, write, reposition

   c) get/set device attributes d) logically attach or detach devices

4. Information Maintenance

   a) get/set time or date b) get/set system data

   c) get/set process, file, or device attributes

5. Communication

   a) create, delete communication connection

   b) send, receive messages

   c) transfer status information

   d) attach or detach remote devices

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.

In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process.

It is because of the critical nature of operations that the operating system itself does them every time they are needed.

e.g. for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells).

Process States

A process is a program in execution.

The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

Components of Process :-

1. Object Program :- Code to be executed.

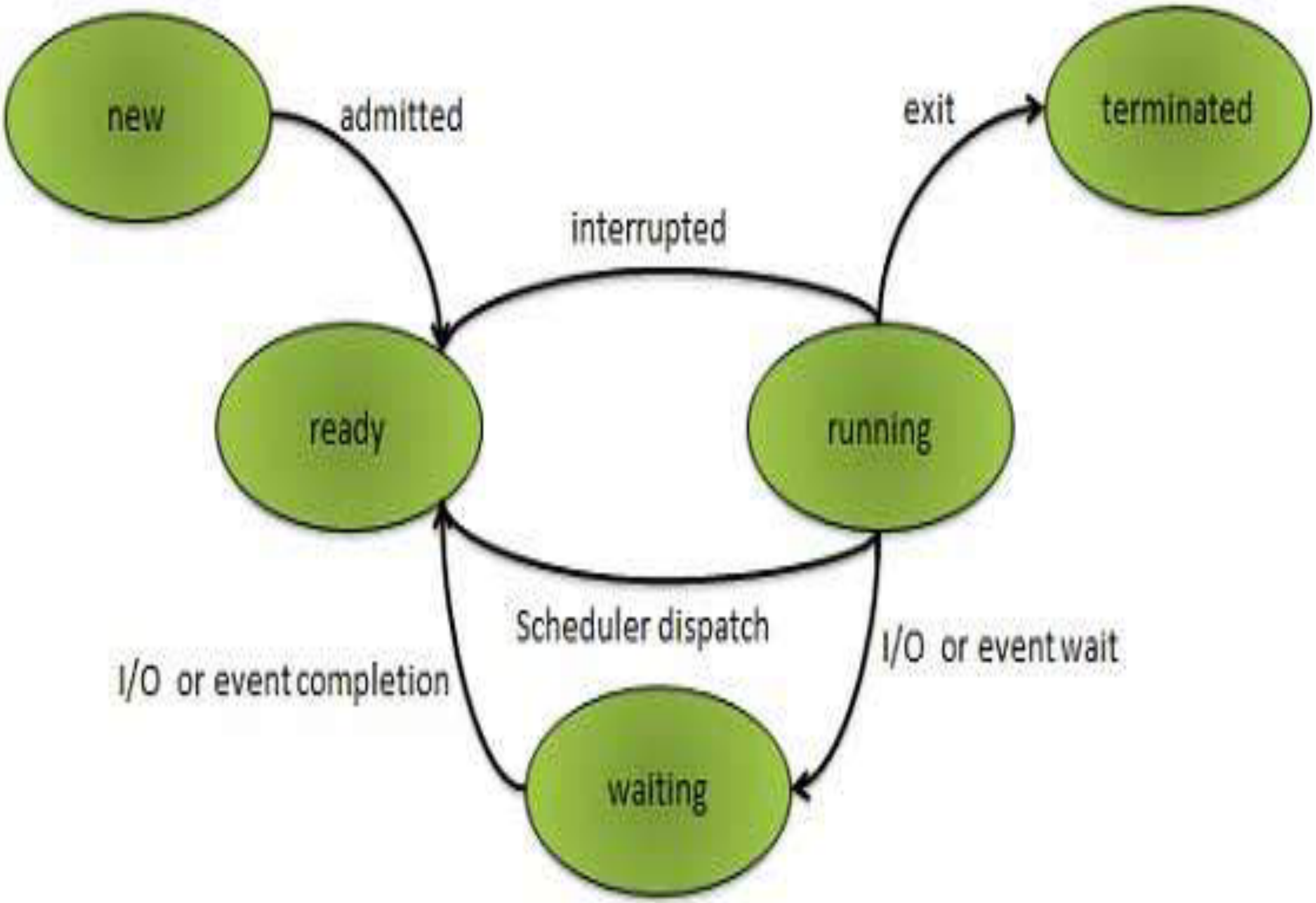
2 . Data :- Data to be used for executing the program.

3. Resources :- While executing the program, it may require some resources.

4. Status :- Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

A process goes through a series of discrete process states.

1) **New State** The process being created.
2) **Ready State** .A 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. It is runnable but temporarily stopped to let another process run. Logically, the 'Running' and 'Ready' states are similar.
3) **Running State** A process is said t be running if it currently has the CPU, i.e. - actually using the CPU at that particular instant.
4) **Blocked (waiting) State** When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.


new
admitted
exit
terminated
interrupted
ready
running
Scheduler dispatch
I/O or event wait
I/O or event completion
waiting

5. **Terminated State** The process has finished execution.

In the case of 'Ready' state, there is temporarily no CPU available for it.

The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

Process Control Block, PCB

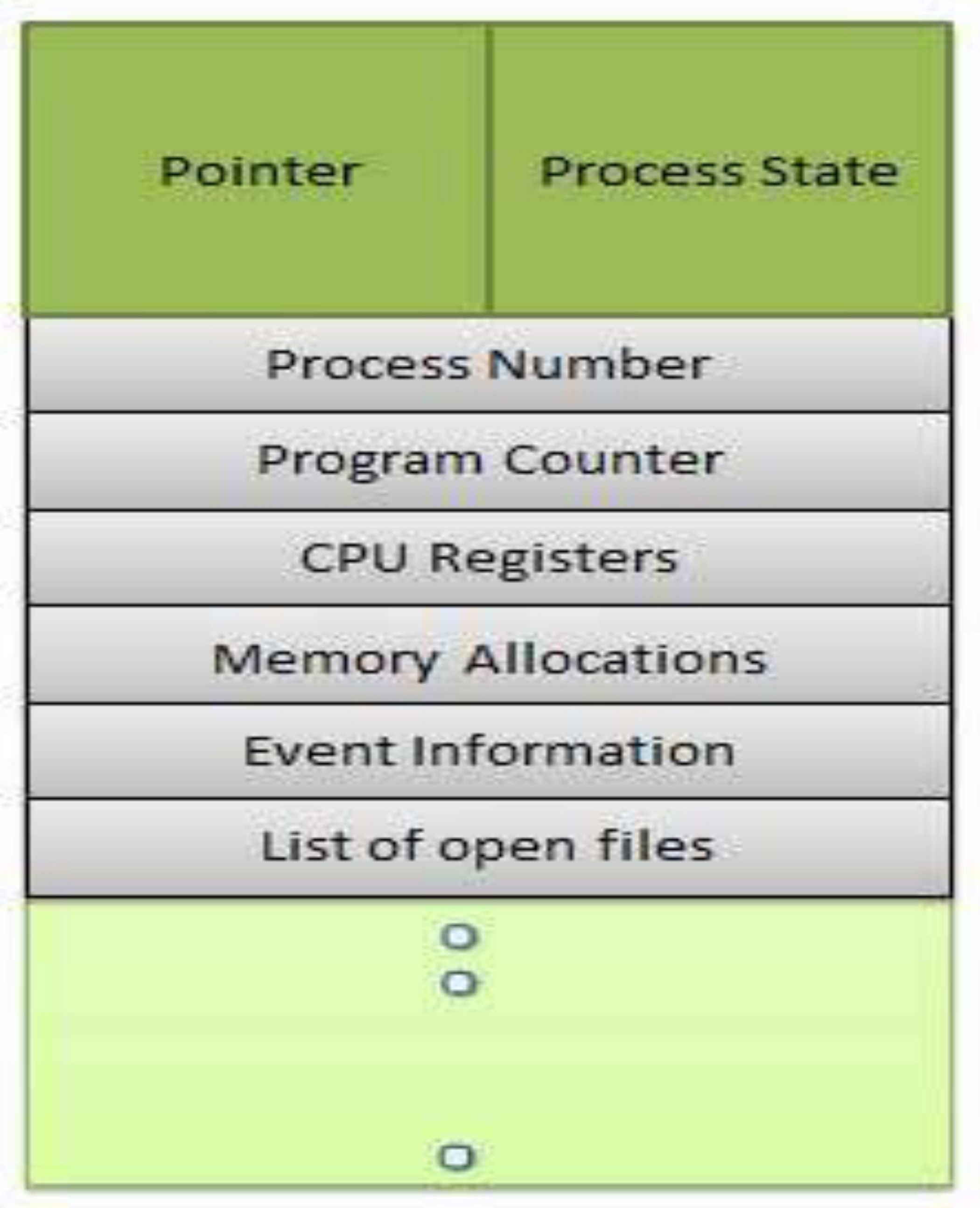
Each process is represented in the operating system by a process control block (PCB) also called a task control block.

PCB is the data structure used by the operating system.

Operating system groups all information that needs about particular process.

PCB contains many pieces of information associated with a specific process which are described below.

Process Control Block, PCB



Pointer

Process State

Process Number

Program Counter

CPU Registers

Memory Allocations

Event Information

List of open files

○
○

○

Pointer

Pointer points to another process control block. Pointer is used for maintaining the scheduling list.

Process State

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

Program Counter

Program Counter indicates the address of the next instruction to be executed for this process.

CPU registers

CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

Memory management information

This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for de-allocating the memory when the process terminates.

Accounting information

This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

Process Control Block includes :-

CPU scheduling, I/O resource management, File management information etc..

The PCB serves as the repository for any information which can vary from process to process.

Loader/linker sets flags and registers when a process is created.

If that process get suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB.

By this technique, the hardware state can be restored so that the process can be scheduled to run again.

In Short - it's a data structure holding:

- PC, CPU registers,
- memory management information,
- accounting (time used, ID, ...)
- I/O status (such as file resources),
- scheduling data (relative priority, etc.)
- Process State (so running, suspended, etc. is simply a field in the PCB).

SCHEDULING

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a **Multiprogramming operating system**. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

The act of **Scheduling a process means changing the active PCB pointed to by the CPU.** Also called a **context switch.**

A context switch is essentially the same as a process switch - it means that the memory, as seen by one process is changed to the memory seen by another process.

Scheduling Queues

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

(Process is driven by events that are triggered by needs and availability)

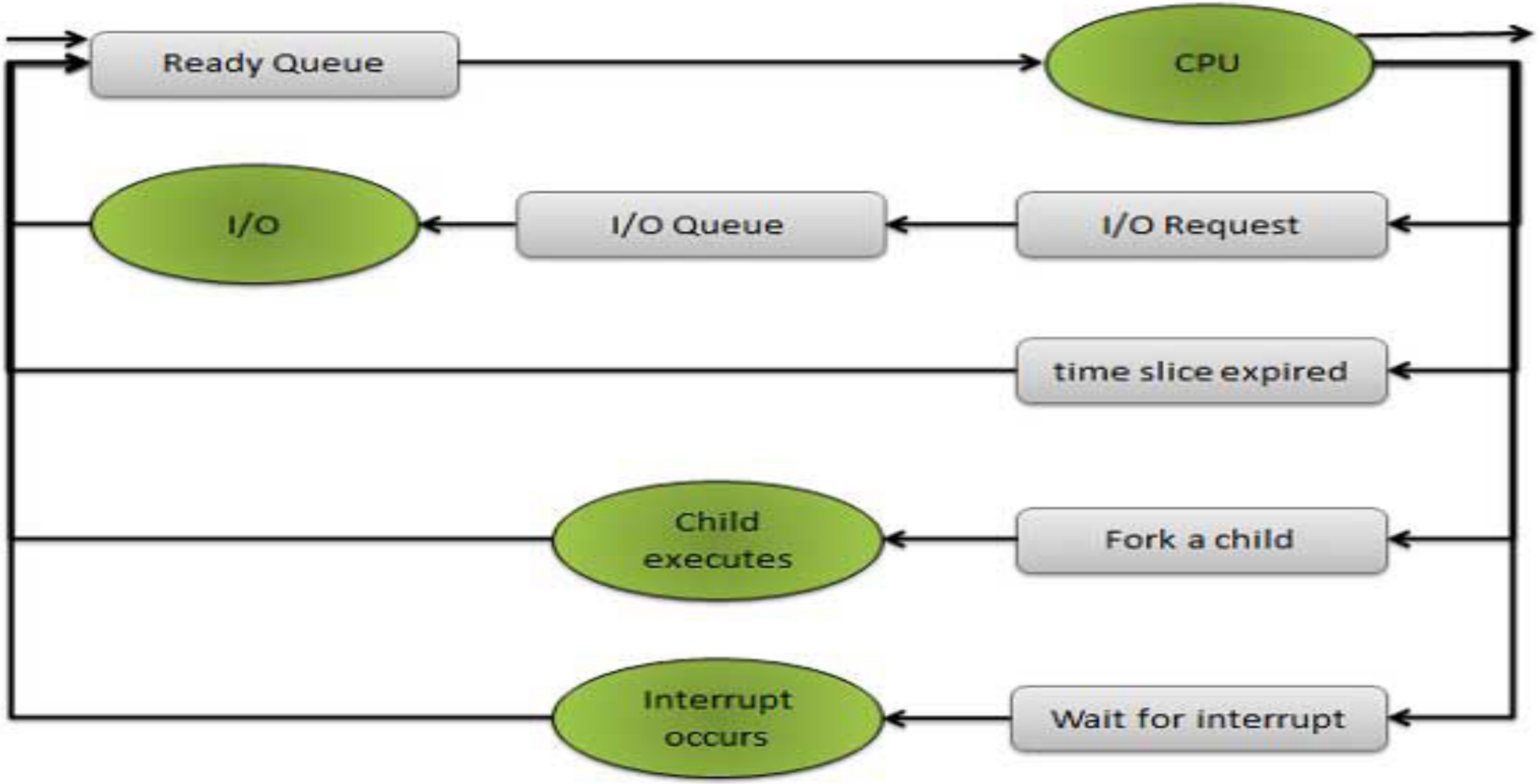
·Ready queue = contains those processes that are ready to run.

·I/O queue (waiting state) = holds those processes waiting for I/O service.

What do the queues look like? They can be implemented as single or double linked.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system



Queues are of two types :- 1) Ready queue 2) Device queue

A newly arrived process is put in the ready queue.

Processes waits in ready queue for allocating the CPU.

Once the CPU is assigned to a process, then that process will execute.

While executing the process, any one of the following events can occur.

The process could issue an I/O request and then it would be placed in an I/O queue.

The process could create new sub process and will wait for its termination.

The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Schedulers

Schedulers are special system softwares which handles 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

LONG TERM SCHEDULER

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

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready , then there is use of long term scheduler.

LONG TERM SCHEDULER …

- Run seldom (rarely) (when job comes into memory)
- Controls degree of multiprogramming
- Tries to balance arrival and departure rate through an appropriate job mix.

SHORT TERM SCHEDULER

It is also called CPU scheduler. Main objective is increasing 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 process among the processes that are ready to execute and allocates CPU to one of them.

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

SHORT TERM SCHEDULER…

Contains three functions:

· Code to remove a process from the processor at the end of its run.

a) Process may go to ready queue or to a wait state.

· Code to put a process on the ready queue –

a) Process must be ready to run.

b) Process placed on queue based on priority.

· Code to take a process off the ready queue and run that process (also called dispatcher).

a) Always takes the first process on the queue (no intelligence required)

b) Places the process on the processor.

This code runs frequently and so should be as short as possible.

MEDIUM TERM SCHEDULER

Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

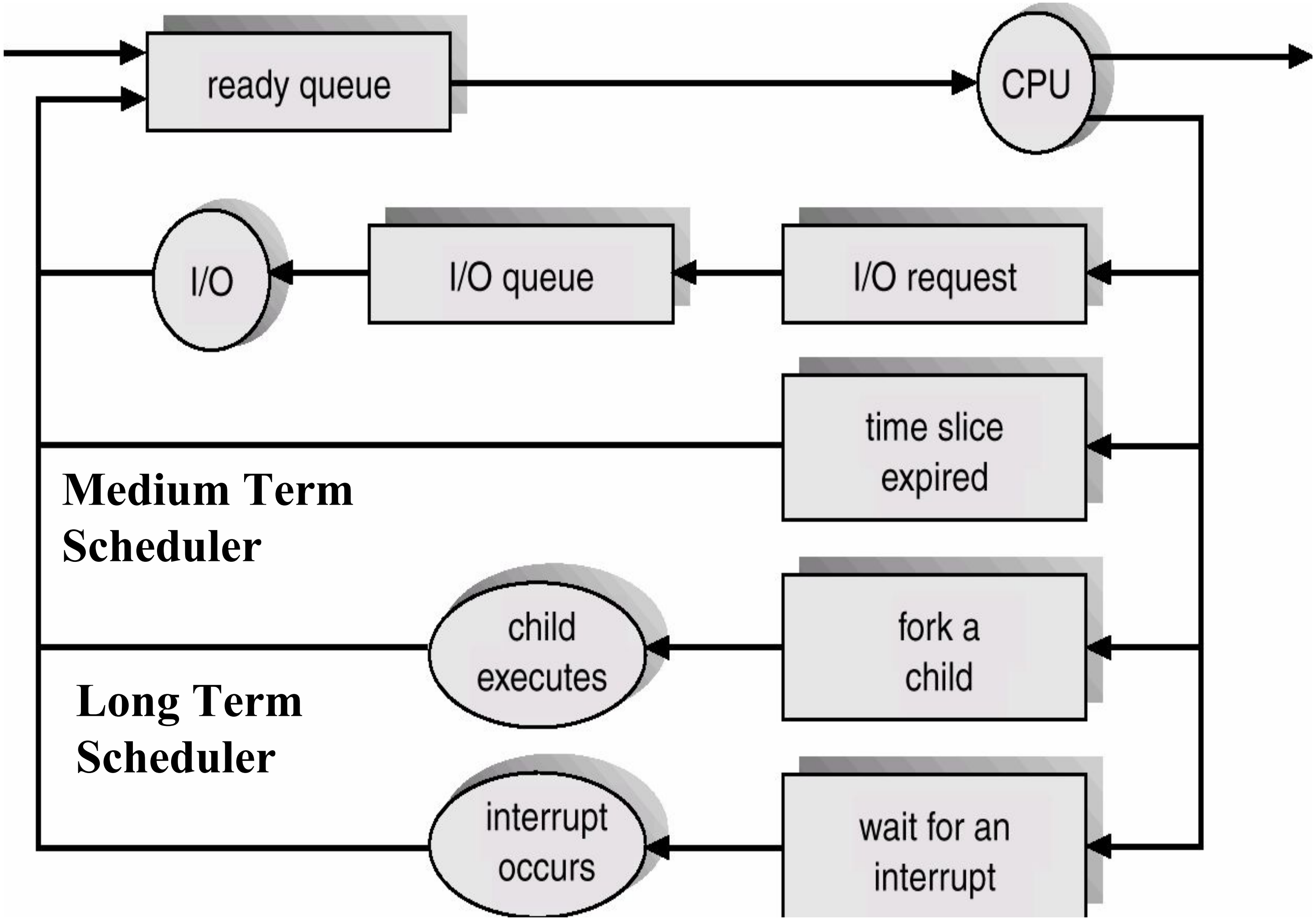
MEDIUM TERM SCHEDULER ….

Mixture of CPU and memory resource management.

Swap out/in jobs to improve mix and to get memory.

Controls change of priority.

Sr. No.	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

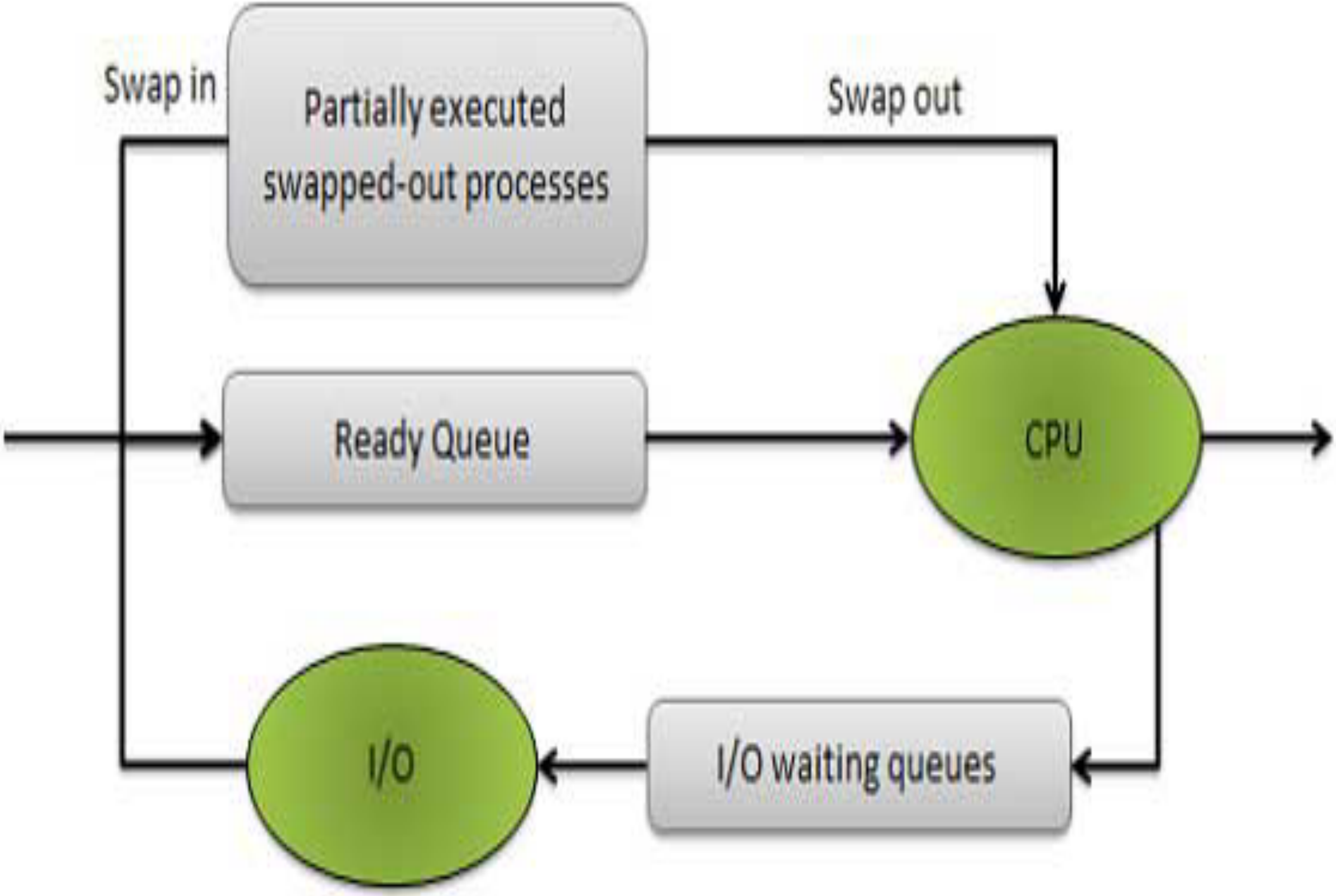


Short Term Scheduler

Medium Term Scheduler

Long Term Scheduler

Interrupt Handler


Swap in
Partially executed swapped-out processes
Swap out
Ready Queue
CPU
I/O
I/O waiting queues

Two State Process Model

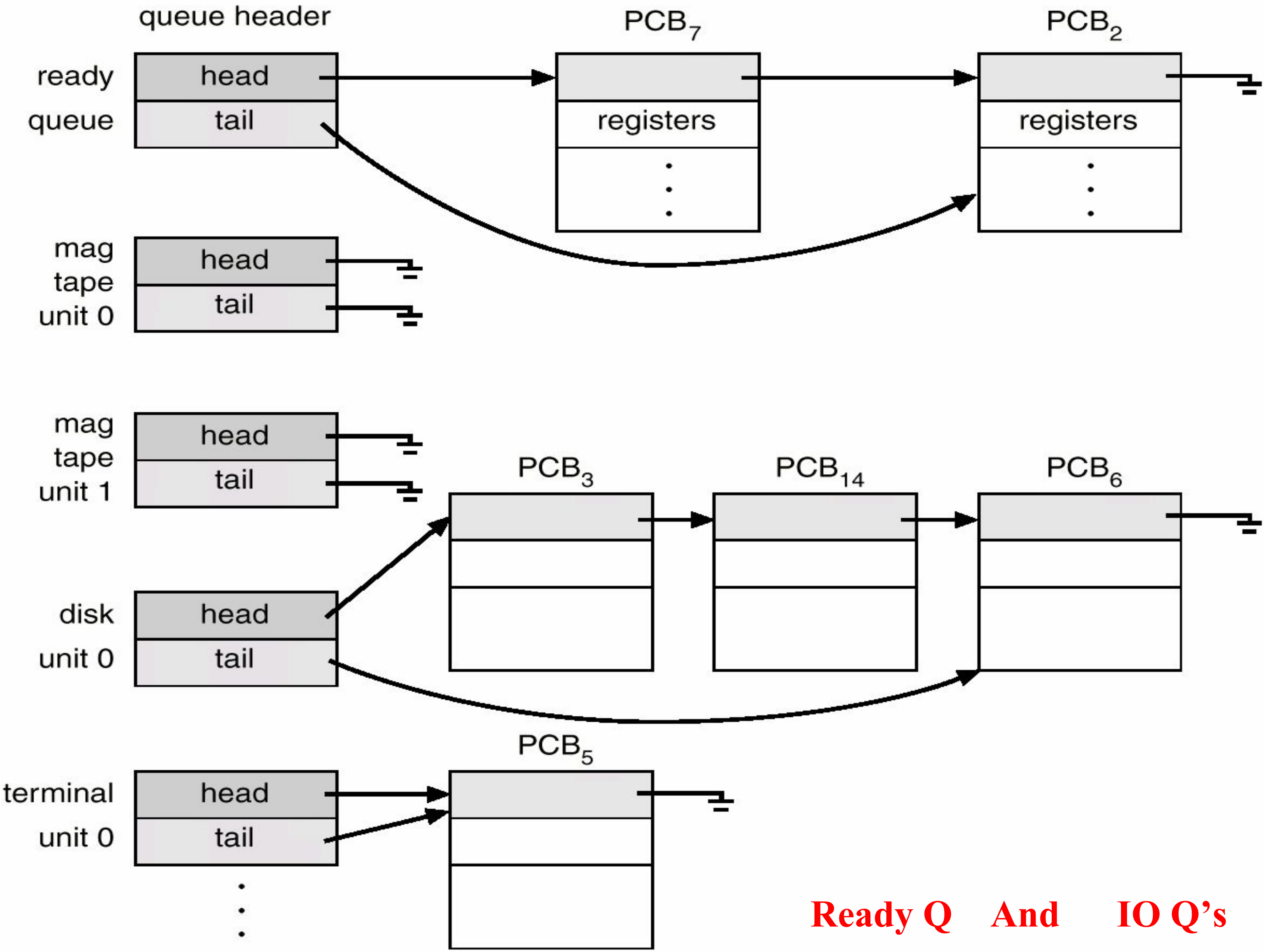
Two state process model refers to running and non-running states which are described below.

Running

1 When new process is created by Operating System that process enters into the system as in the running state.

Not Running

2 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. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.



Ready Q And IO Q’s

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.

Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

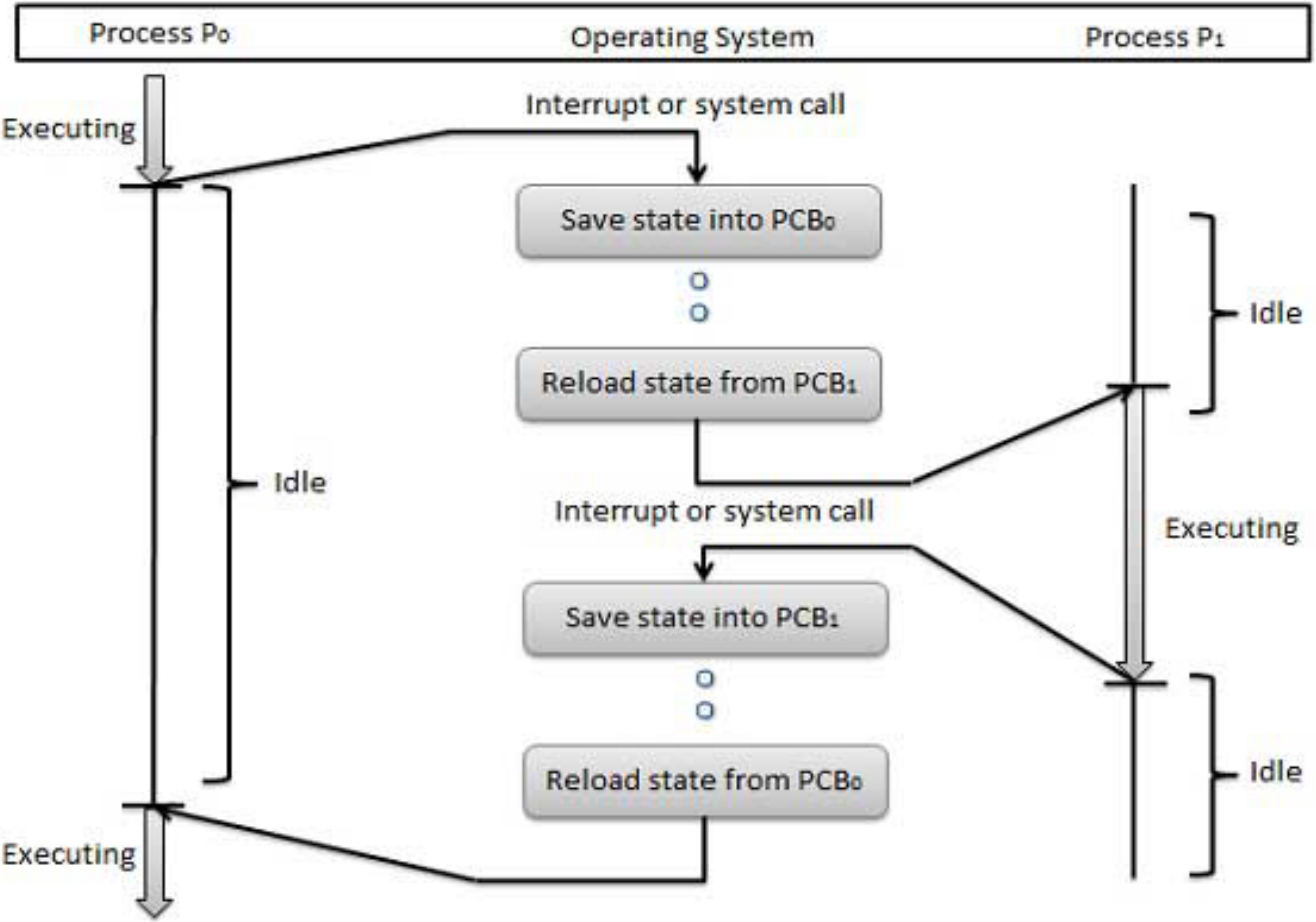
When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support.

Context switch requires (n + m) bxK time units to save the state of the processor with **n** general registers, assuming b are the store operations are required to save **n** and **m** registers of two process control blocks and each store instruction requires K time units.

Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting



The CPU switching from one process to another.

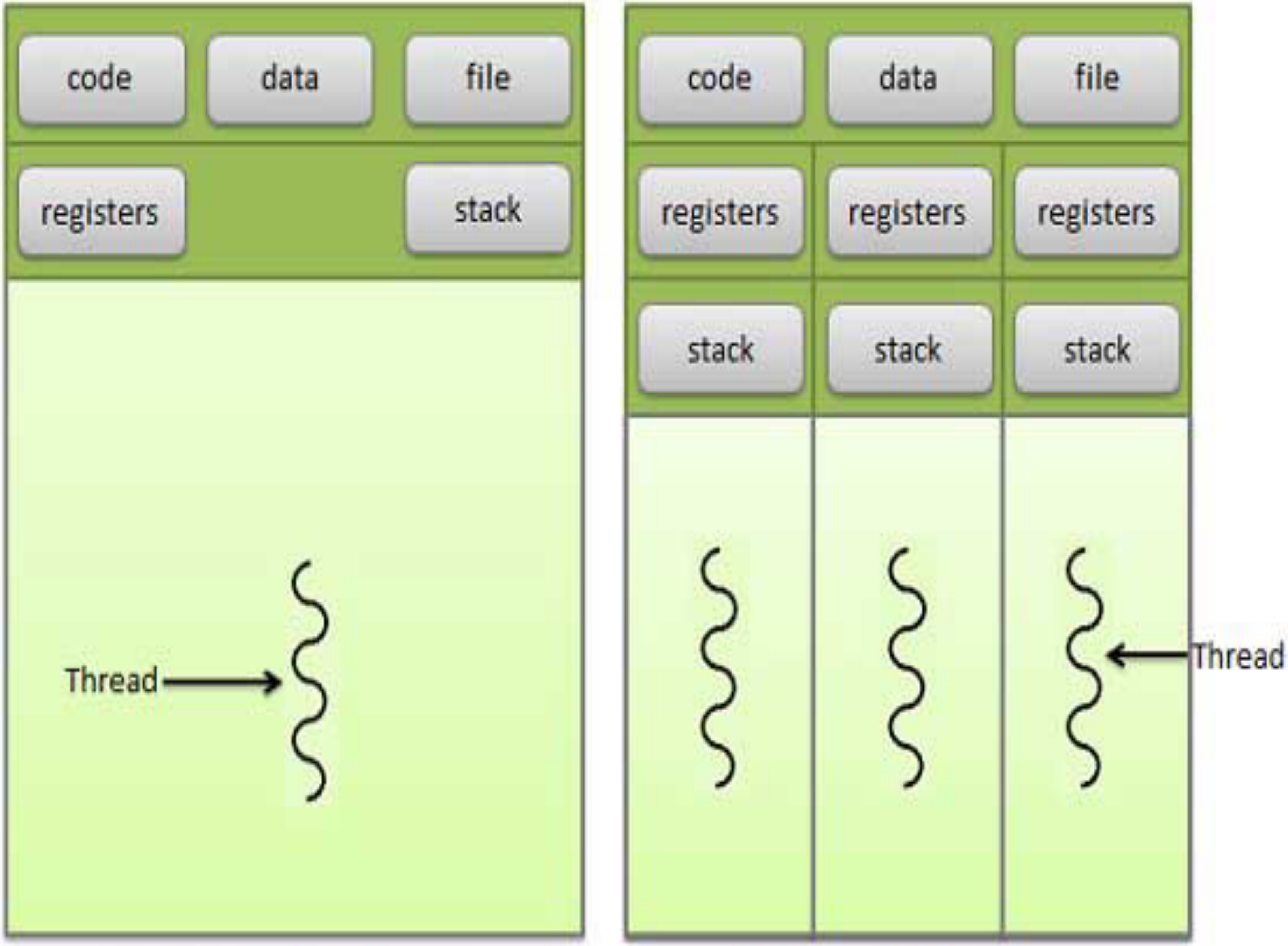
Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

What is Thread?

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Following figure shows the working of the single and multithreaded processes.



Single threaded Process

Multi-threaded Process

Sr. No.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
1	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
1	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
1	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
1	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
1	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Adntages of Thread

- Thread minimize context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways

- **User Level Threads** -- User managed threads
- **Kernel Level Threads** -- Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, application manages thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

Advantages

Thread switching does not require Kernel mode privileges.

User level thread can run on any operating system.

Scheduling can be application specific in the user level thread.

User level threads are fast to create and manage.

Disadvantages

In a typical operating system, most system calls are blocking.

Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

Kernel can simultaneously schedule multiple threads from the same process on multiple processes.

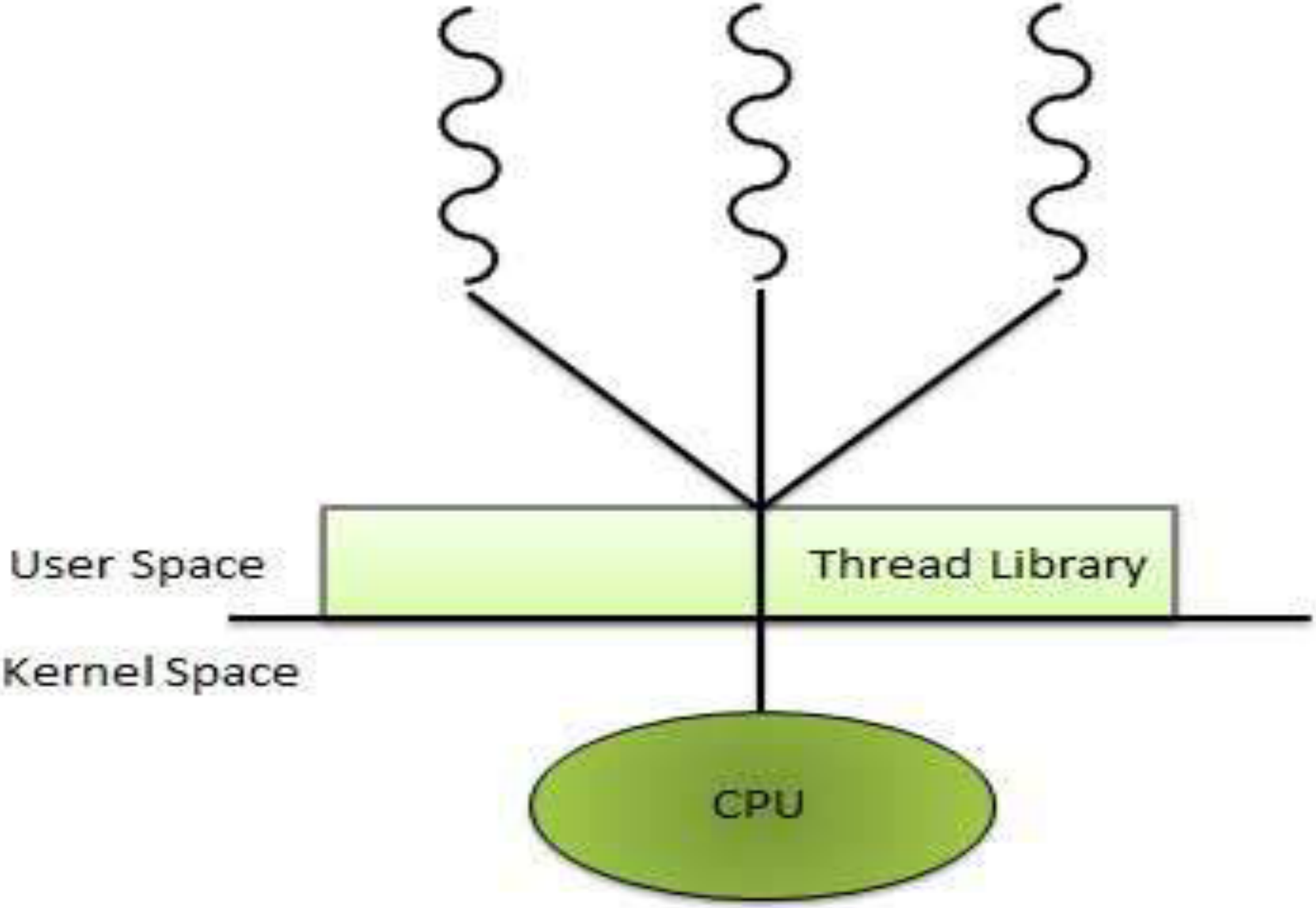
If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

Kernel routines themselves can multithreaded.

Disadvantages

Kernel threads are generally slower to create and manage than the user threads.

Transfer of control from one thread to another within same process requires a mode switch to the Kernel.


User Space
Thread Library
Kernel Space
CPU

Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

Many to many relationship.

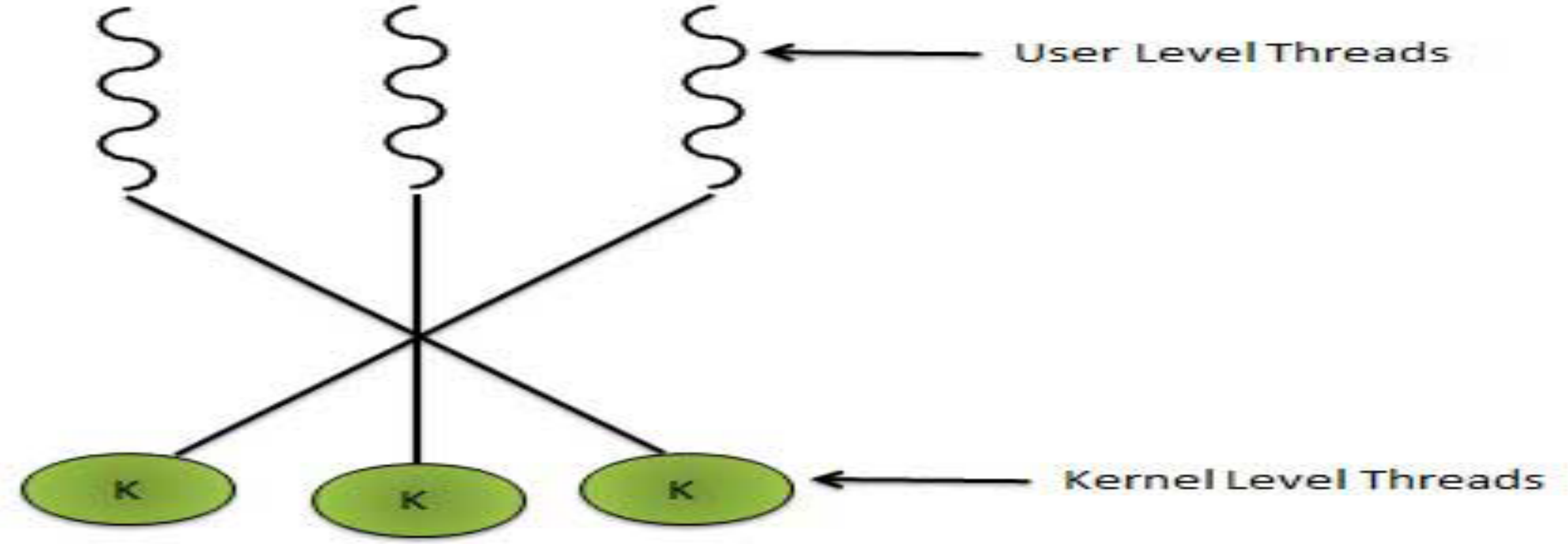
Many to one relationship.

One to one relationship.

Many to Many Model

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

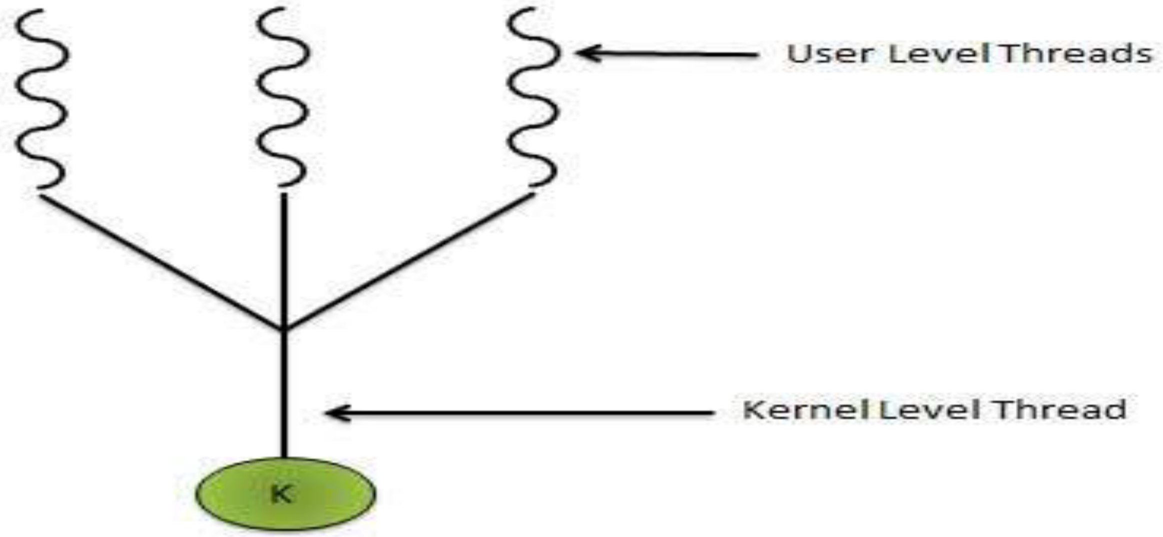
Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

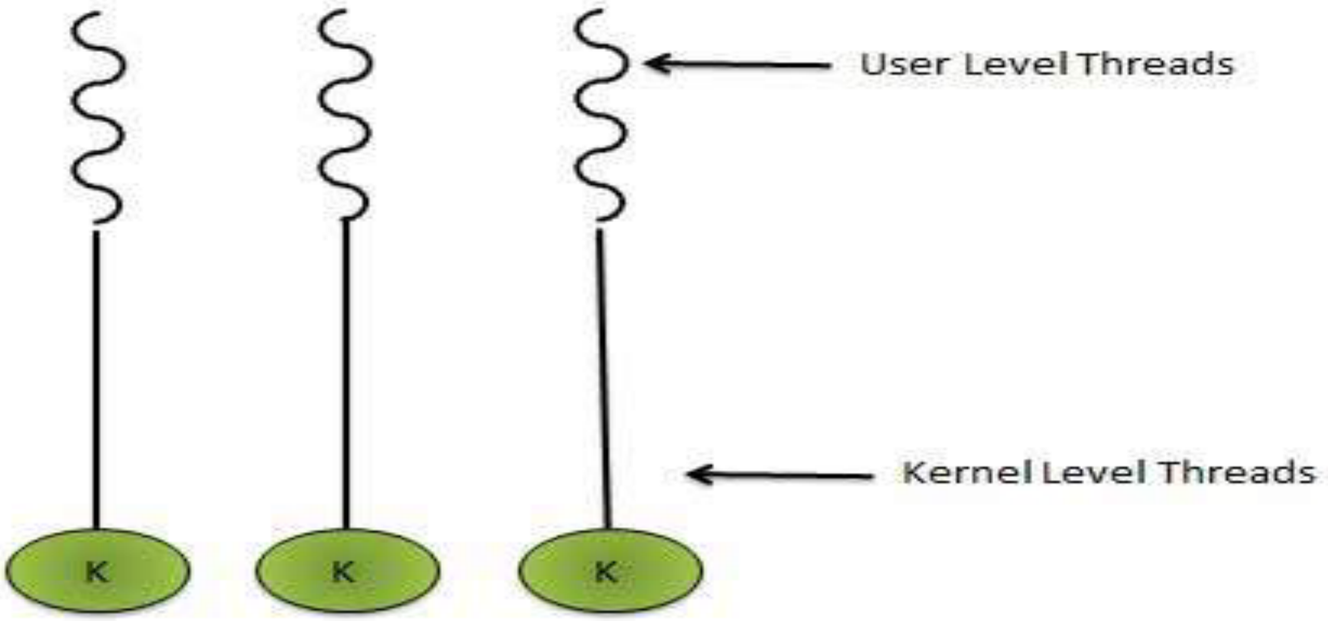
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes



One to One Model

There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model



Difference between User Level & Kernel Level Thread

Sr. No.	User Level Threads	Kernel Level Thread
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User level thread is generic & can run on any OS	Kernel level thread is specific to the operating system.
4	Multi-threaded application can't take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Scheduling Criteria

CPU utilization – keep the CPU as busy as possible

Throughput – No of processes that complete their execution per time unit .

Turnaround Time – amount of time to execute a particular process

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

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

Optimization Criteria

Max CPU utilization

Max throughput

Min turnaround time

Min waiting time

Min response time

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.

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 the 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 periods spend 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.

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. Investigators have suggested that, for interactive systems, it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance

Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

Shortest-Job-First (SJR) Scheduling

Priority Scheduling

Round Robin (RR)

Multilevel queue scheduling

How to choose a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above. For example, Windows NT/XP/Vista uses a Multilevel feedback queue, a combination of fixed priority preemptive scheduling, round-robin, and first in first out.

In this system, processes can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling amongst the high priority processes and FIFO among the lower ones. In this sense, response time is short for most processes, and short but critical system processes get completed very quickly. Since processes can only use one time unit of the round robin in the highest priority queue, starvation can be a problem for longer high priority processes.

Types of Scheduling Algorithm

Circumstances that scheduling may take place :-

A process switches from the running state to the waiting state (e.g., doing for I/O, invocation of wait for the termination of one of the child processes)

A process switches from the running state to the ready state

(e.g., an interrupt occurs)

A process switches from the waiting state to the ready state (e.g., I/O completion)

A process terminates

Non-preemptive scheduling :-

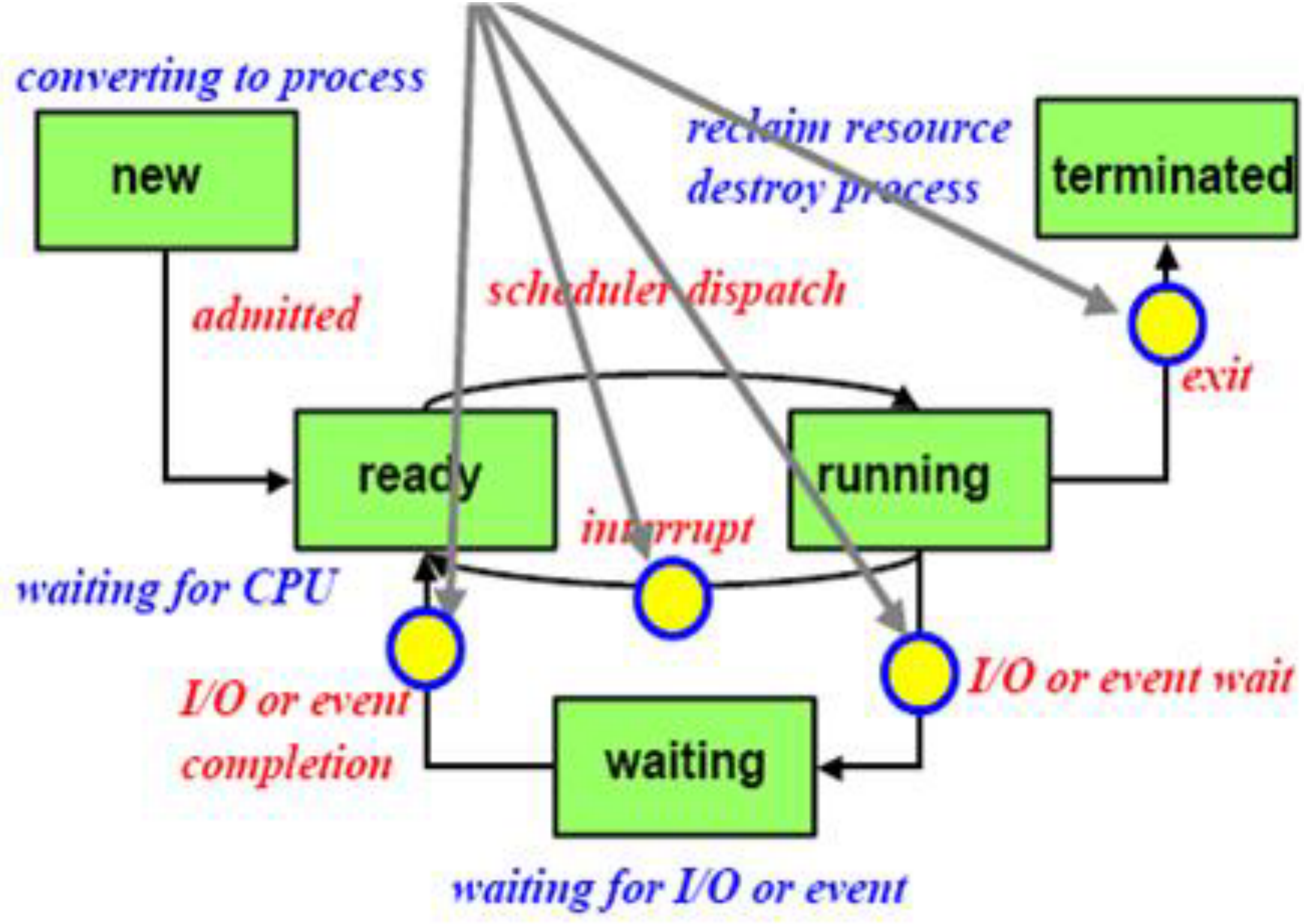
Scheduling occurs only when a process voluntarily enter the wait state or terminates

Simple, but very inefficient.

It 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.

Preemptive Scheduling :- scheduling occurs in all possible cases. What if the kernel is in its critical section modifying some important data? Mutual exclusion may be violated.

The kernel must pay special attention to this situation and, hence, is more complex.


converting to process
new
admitted
scheduler dispatch
reclaim resource
destroy process
terminated
exit
ready
running
interrupt
waiting for CPU
I/O or event
completion
I/O or event wait
waiting
waiting for I/O or event

First Come First Serve

First Come, First Served (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.

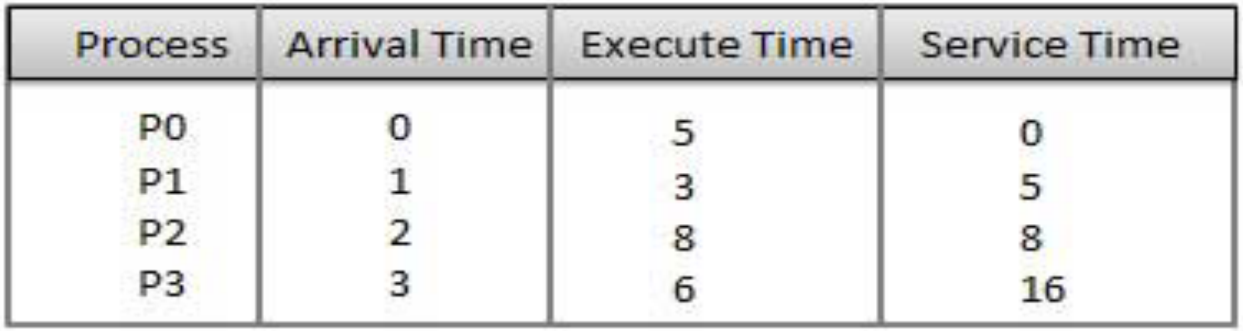
Throughput can be low, since long processes can hog the CPU

Turnaround time, waiting time and response time can be high for the same reasons above

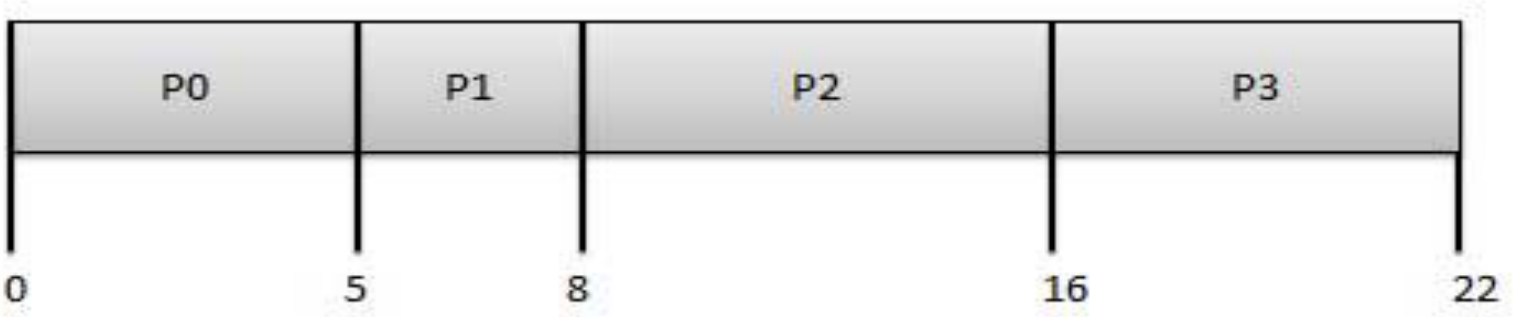
No prioritization occurs, thus this system has trouble meeting process deadlines.

The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.

It is based on Queuing



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is following

Process	**Wait Time : Service Time - Arrival Time**
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.55

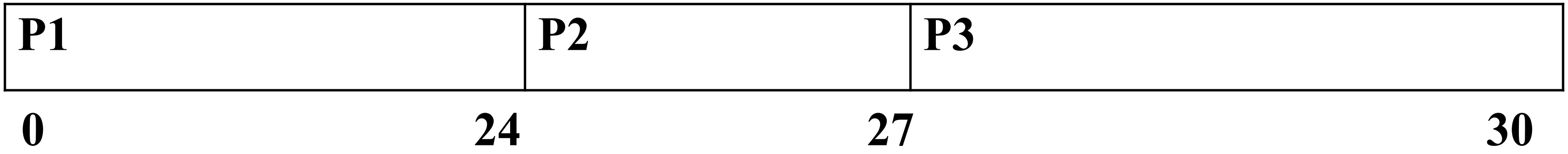
First-Come, First-Served (FCFS) Scheduling

Process	Burst Time (time required for completion)
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order : ***P1* , *P2* , *P3***

The Gantt Chart for the schedule is:

a chart in which a series of horizontal lines shows the amount of work done or production completed in certain periods of time in relation to the amount planned for those periods.

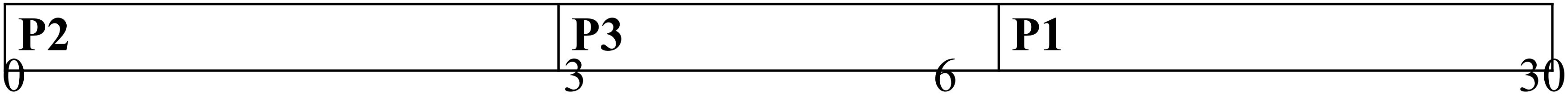


Waiting time for *P1* = 0; *P2* = 24; *P3* = 27

Average waiting time: (0 + 24 + 27)/3 = 51/3 = 17

Suppose that the processes arrive in the order :- *P2* , *P3* , *P1* .

The Gantt chart for the schedule is:



P2 P3 P1

0 3 6 30

Waiting time for *P1* = 6*;* *P2* = 0*;* *P3* = 3

Average waiting time: (6 + 0 + 3)/3 = 9/3 = 3

Much better than previous case.

Due to Convoy effect short process behind long process

Process	**Burst Time (time required for completion)**
P2	03
P3	03
P1	24

Convoy effect

Consider : $P1$: **CPU-bound** $P2, P3, P4$: **I/O-bound**

$P2$, $P3$ and $P4$ could quickly finish their IO request → ready queue, waiting for CPU.

Note : IO devices are idle then.

then $P1$ finishes its CPU burst and move to an IO device.

$P2$, $P3$, $P4$, which have short CPU bursts, finish quickly → back to IO queue.

Note: CPU is idle then.

$P1$ moves then back to ready queue is gets allocated CPU time.

Again $P2$, $P3$, $P4$ wait behind $P1$ when they request CPU time.

One cause : FCFS is non-preemptive

$P1$ keeps the CPU as long as it needs

Shortest-Job-First (SJR) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

Two schemes:

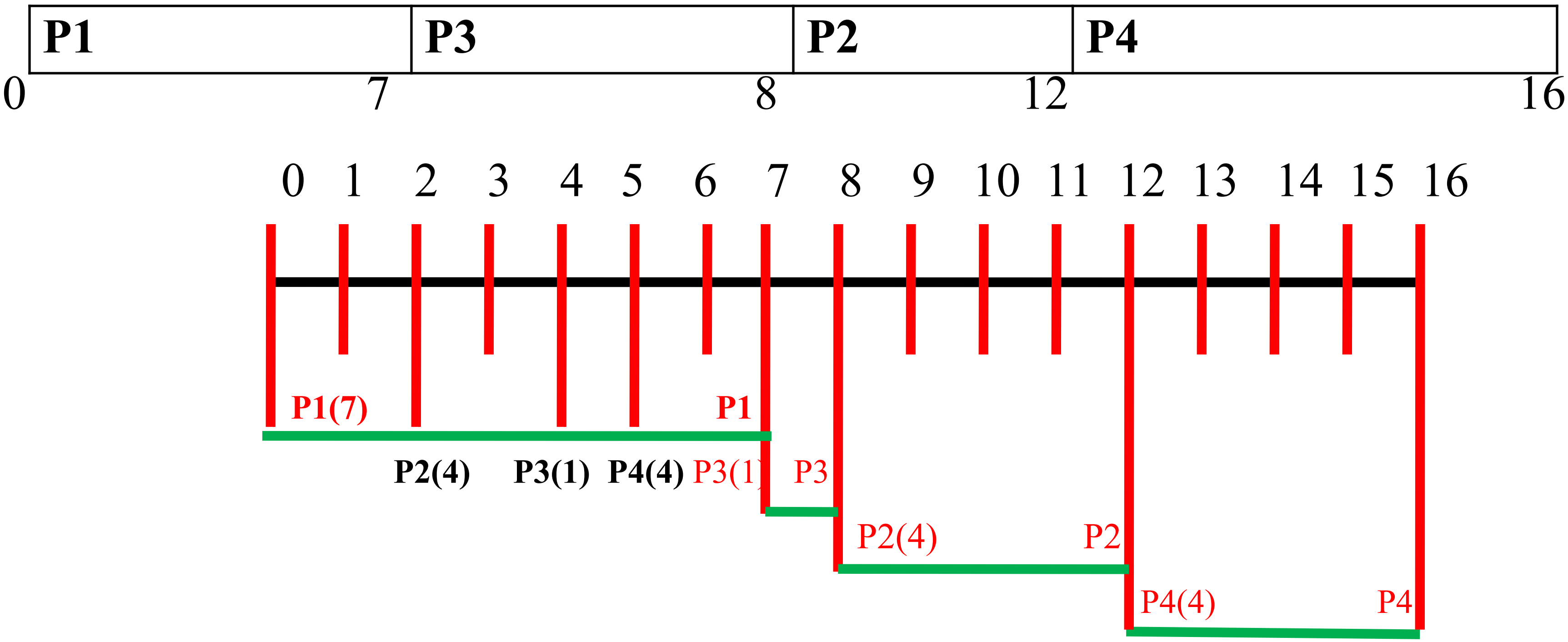
1. non pre- emptive – once CPU given to the process it cannot be preempted until completes its CPU burst.

2. preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

SJF is optimal – gives minimum average waiting time for a given set of processes.

Process	**Arrival Time**	**Burst Time**
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

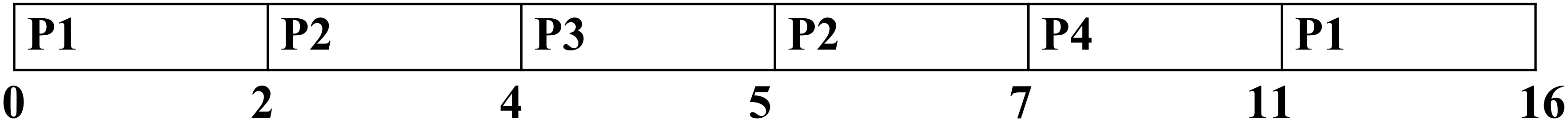
SJF (non-preemptive)



Process	Arrival Time	Burst Time	Waiting Time
P1	0.0	7	0
P2	2.0	4	08 - 02 = 6
P3	4.0	1	07 - 04 = 3
P4	5.0	4	12 - 05 = 7

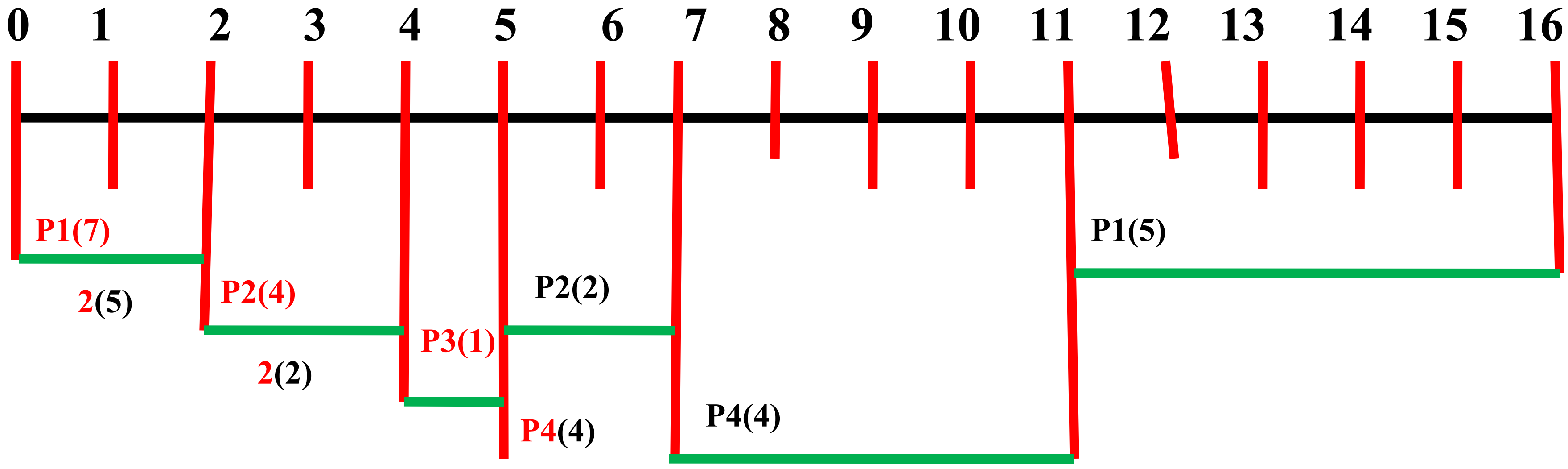
Average waiting time = [0 +(8-2) + (7-4) + (12-5)] /4 = 16/4 = 4

Example of Preemptive SJF



Shortest Job First (SJF)

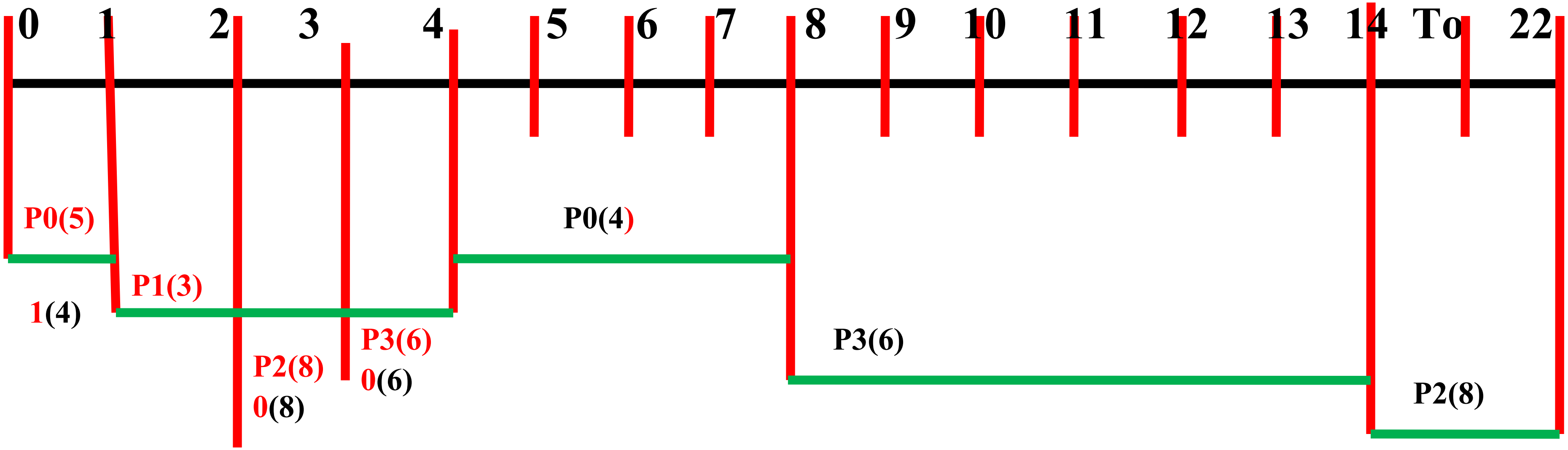
- Best approach to minimize waiting time.
- Impossible to implement
- Processer should know in advance how much time process will take.

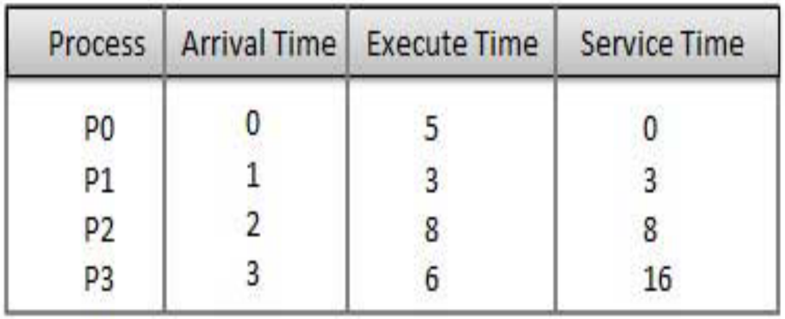


Process	Arrival Time	Burst Time	Waiting Time
P1	0.0	7	11 - 02 = 09
P2	2.0	4	05 – 02 = 03 – 02= 01
P3	4.0	1	04 – 04 = 00
P4	5.0	4	07 – 05 = 02

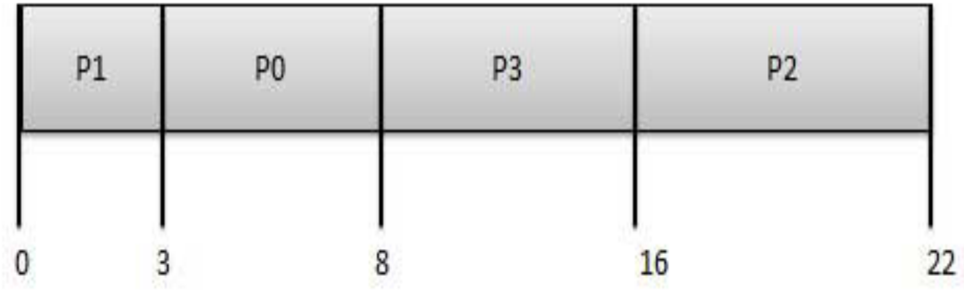
SJF (preemptive)

Average waiting time = (9 + 1 + 0 +2) /4 = 12/4 = 3





Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	3
P2	2	8	8
P3	3	6	16



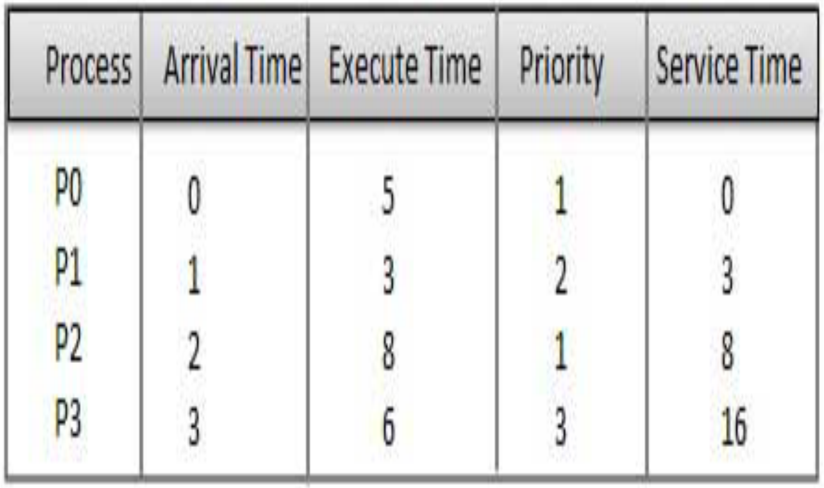
Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3 (4 – 1 = 3)
P1	0 - 0 = 0 (1-1 = 0)
P2	14 -02 = 12
P3	8 - 3 = 5

Average waiting time = (3 + 0 + 12 +5) /4 = 20/4 = 5

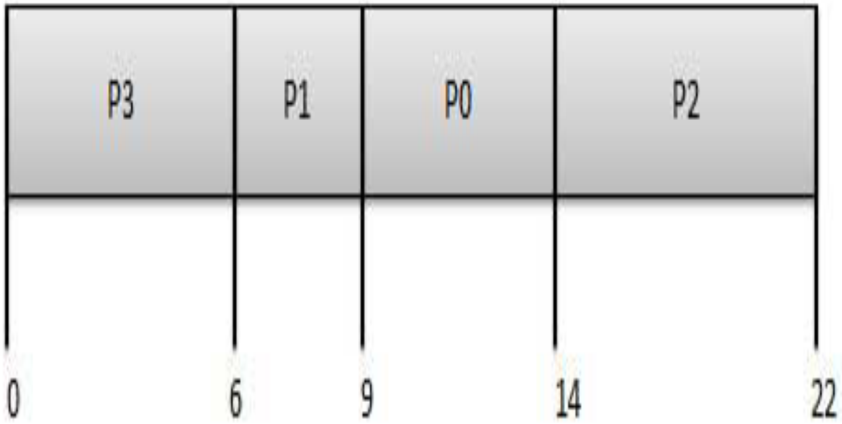
Priority Based Scheduling

- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first serve basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Wait time of each process is following



Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	3
P2	2	8	1	8
P3	3	6	3	16



Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

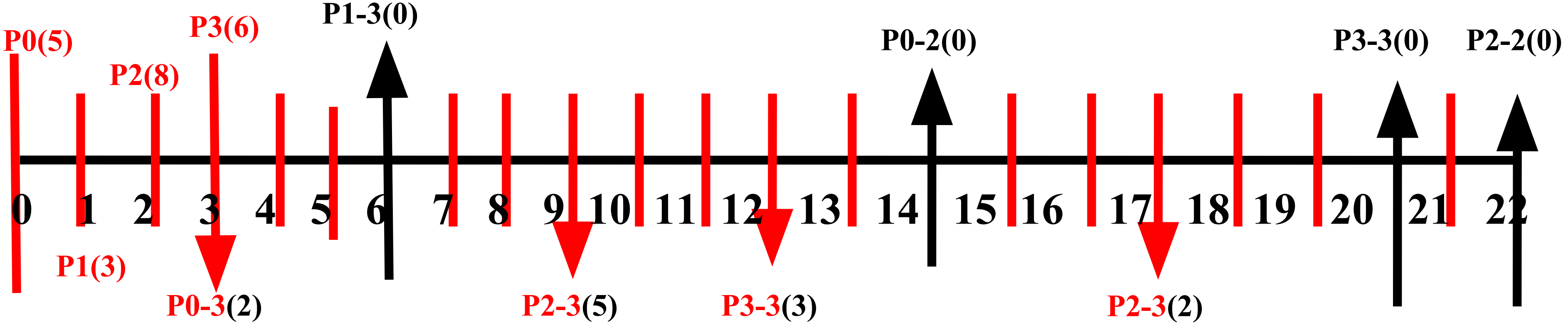
Average Wait Time : (9+5+12+0) / 4 = 6.5

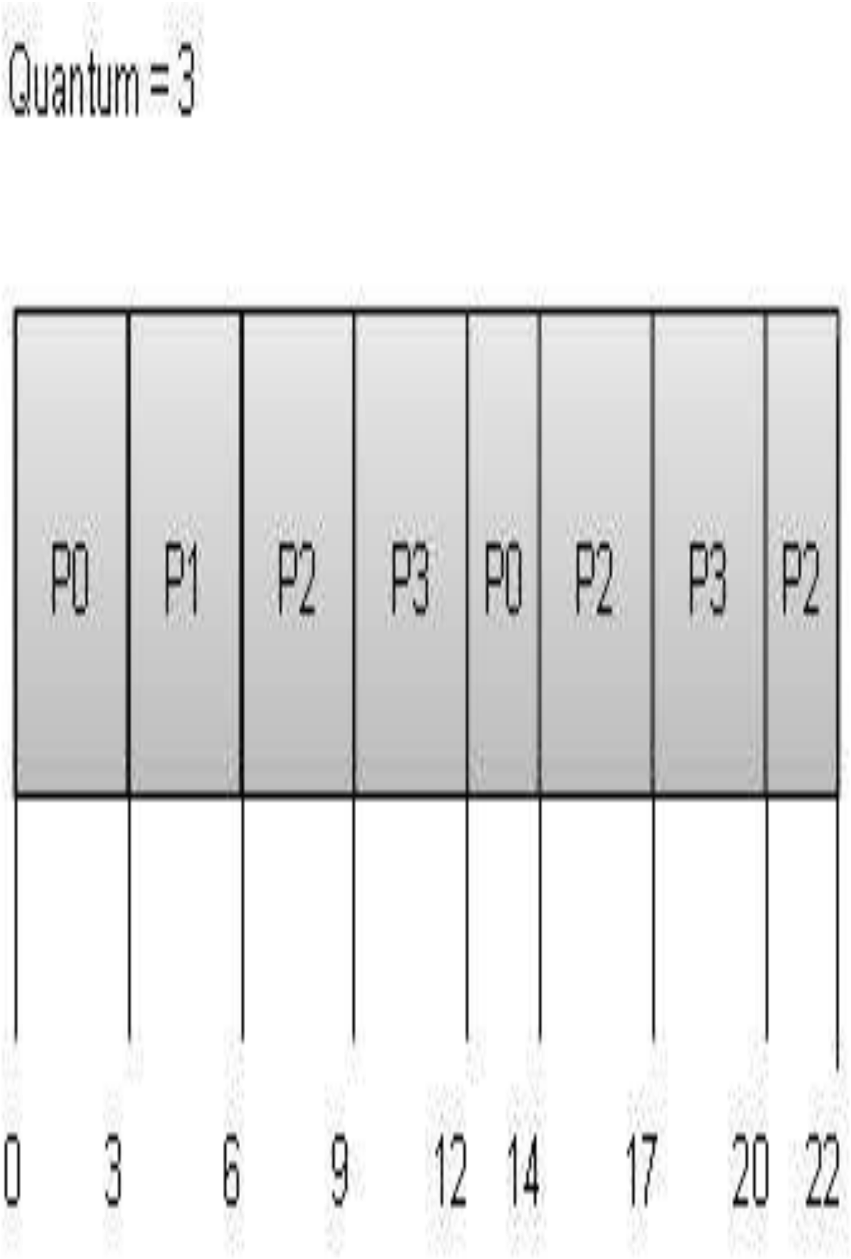
Round Robin Scheduling

- Each process is provided a fix time to execute called quantum.
- Once a process is executed for given time period. Process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.

Wait time of each process is following

Average Wait Time: (9+2+12+11) / 4 = 8.5

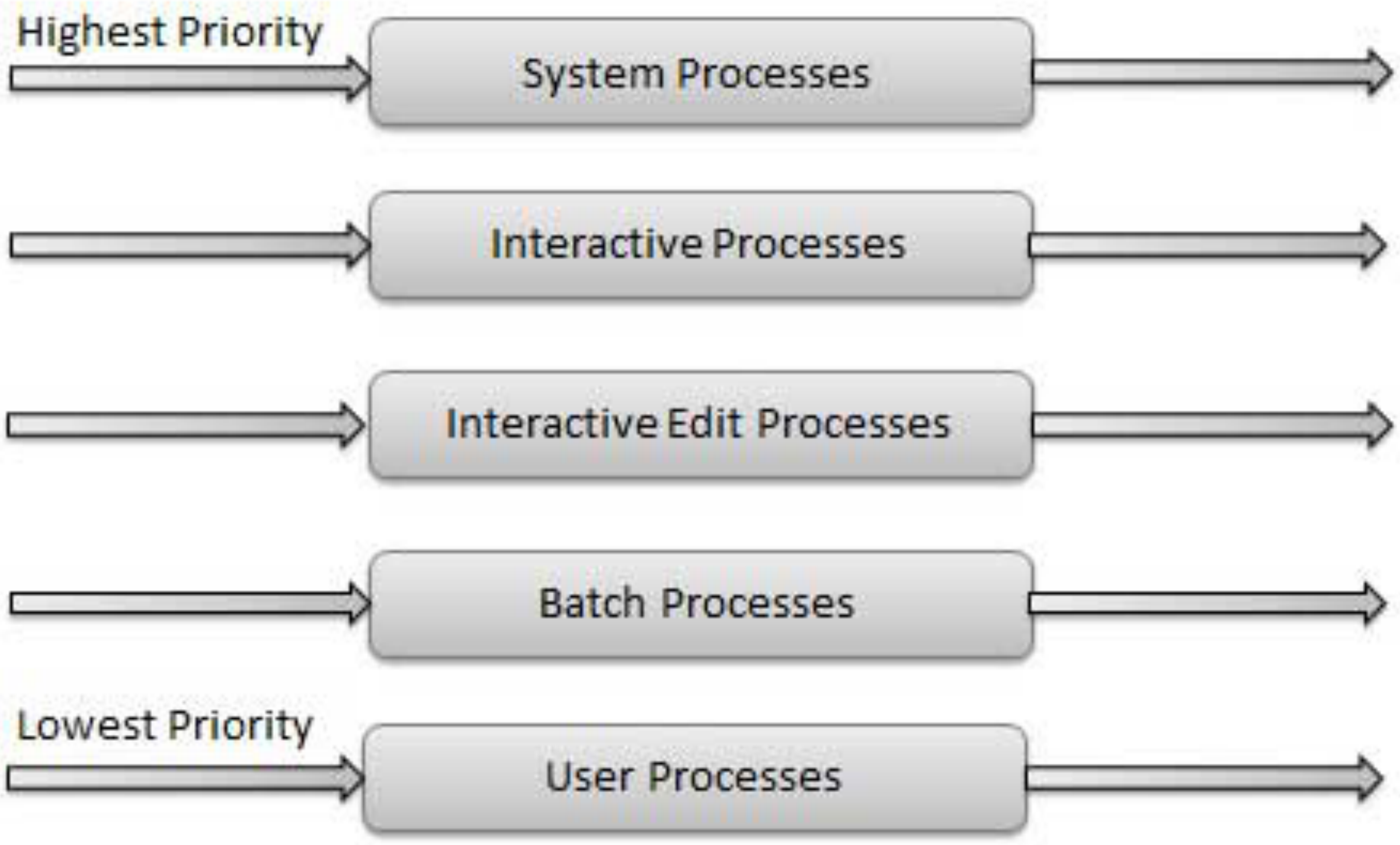




Process	Wait Time : Service Time - Arrival Time
P0	(0-0) + (12-3) = 9
P1	(3-1) = 2
P2	(6-2) + (14-9) + (20-17) = 12
P3	(9-3) + (17-12) = 11

Multi Queue Scheduling

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.



Multitasking

Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory). At any time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *process* or *thread context switching*). There are subtle differences between multitasking and multiprogramming. A *task* in a multitasking operating system is not a whole application program but it can also refer to a “thread of execution” when one process is divided into sub-tasks. Each smaller task does not hijack the CPU until it finishes like in the older multiprogramming but rather a fair share amount of the CPU time called quantum.

Just to make it easy to remember, both multiprogramming and multitasking operating systems are **(CPU) time sharing** systems. However, while in multiprogramming (older OSs) one program as a whole keeps running until it blocks, in multitasking (modern OSs) time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

Multithreading

Up to now, we have talked about multiprogramming as a way to allow multiple programs being resident in main memory and (apparently) running at the same time. Then, multitasking refers to multiple tasks running (apparently) simultaneously by sharing the CPU time. Finally, multiprocessing describes systems having multiple CPUs. So, where does multithreading come in? Multithreading is an execution model that allows a single process to have multiple code segments (i.e., *threads*) run concurrently within the "context" of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system Why should we need to have multiple threads of execution within a single process context?

Well, consider for instance a GUI application where the user can issue a command that require long time to finish (e.g., a complex mathematical computation). Unless you design this command to be run in a separate execution thread you will not be able to interact with the main application GUI (e.g., to update a progress bar) because it is going to be unresponsive while the calculation is taking place. Of course, designing multithreaded/concurrent applications requires the programmer to handle situations that simply don't occur when developing single-threaded, sequential applications. For instance, when two or more threads try to access and modify a shared resource (*race conditions*), the programmer must be sure this will not leave the system in an inconsistent or deadlock state. Typically, this thread synchronization is solved using OS primitives, such as **mutexes** and **sempaphores**.

Multiprogramming

In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn. The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.

Multiprocessing

Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. This might be misleading because we have already introduced the term “multiprogramming” to describe that before. In fact, multiprocessing refers to the *hardware* (i.e., the CPU units) rather than the *software* (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. Several variations on the basic scheme exist, e.g., multiple cores on one die or multiple dies in one package or multiple packages in one system. Anyway, a system can be both multiprogrammed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.