Process

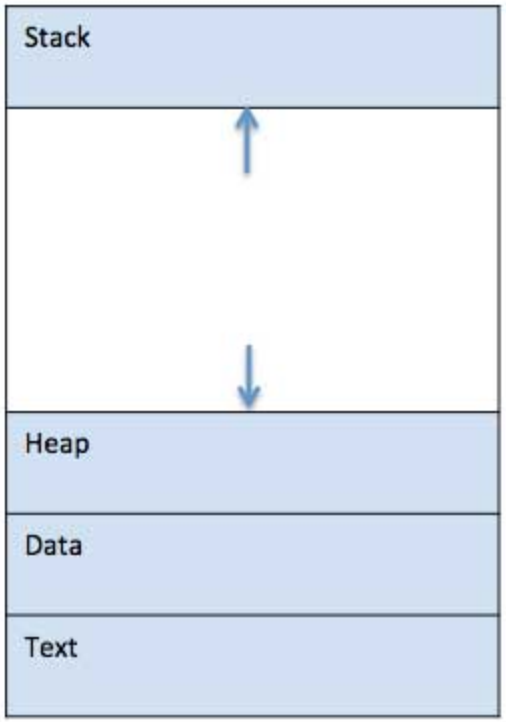
Process

A process is basically 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.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data. The following image shows a simplified layout of a process inside main memory –


Stack
Heap
Data
Text

S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

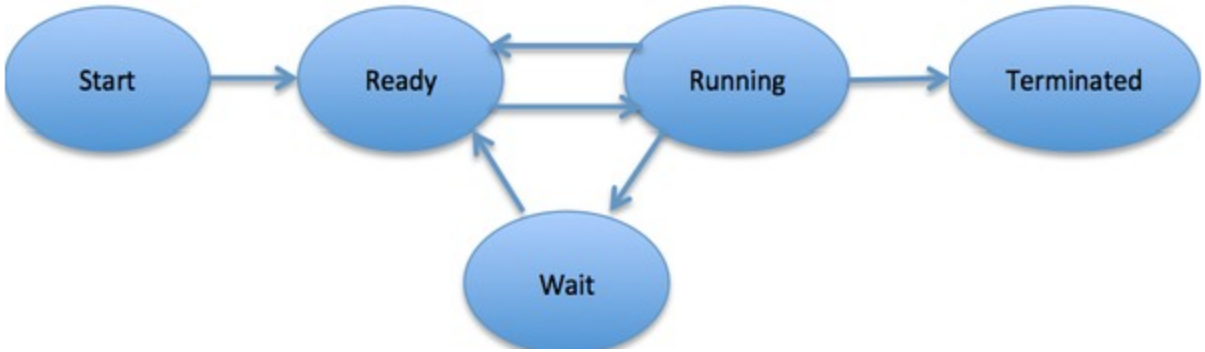
Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**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.
3	**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.

4	**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.
5	**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.



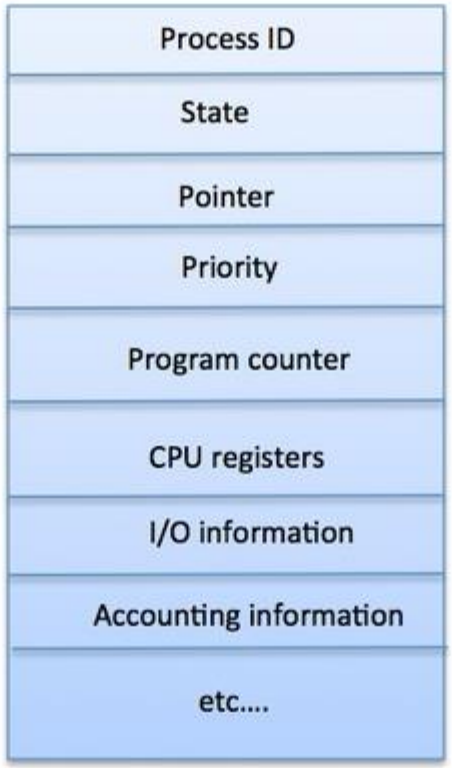
Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.

6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Definition

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.

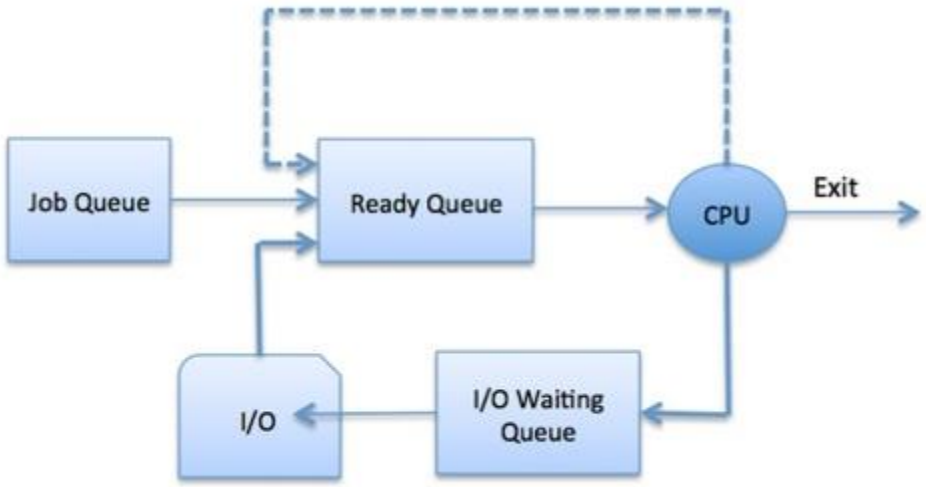
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

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. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling 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.


Job Queue
Ready Queue
CPU
Exit
I/O
I/O Waiting Queue

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

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

S.N.	State & Description
1	**Running** When a new process is created, it enters into the system as in the running state.
2	**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. 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.

Schedulers

Schedulers are special system software which handle 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 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.

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 a process changes the state from new to ready, then there is use of long-term scheduler.

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. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **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.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, 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.

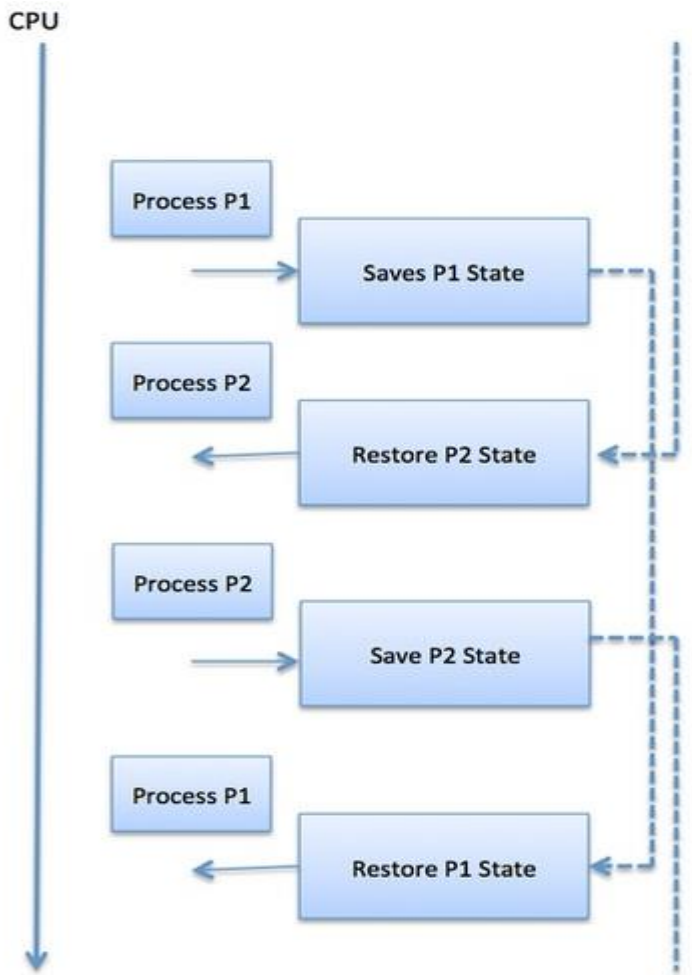
Comparison among Scheduler

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

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 state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.


CPU
Process P1
Saves P1 State
Process P2
Restore P2 State
Process P2
Save P2 State
Process P1
Restore P1 State

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

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

CPU Scheduling

CPU Scheduling

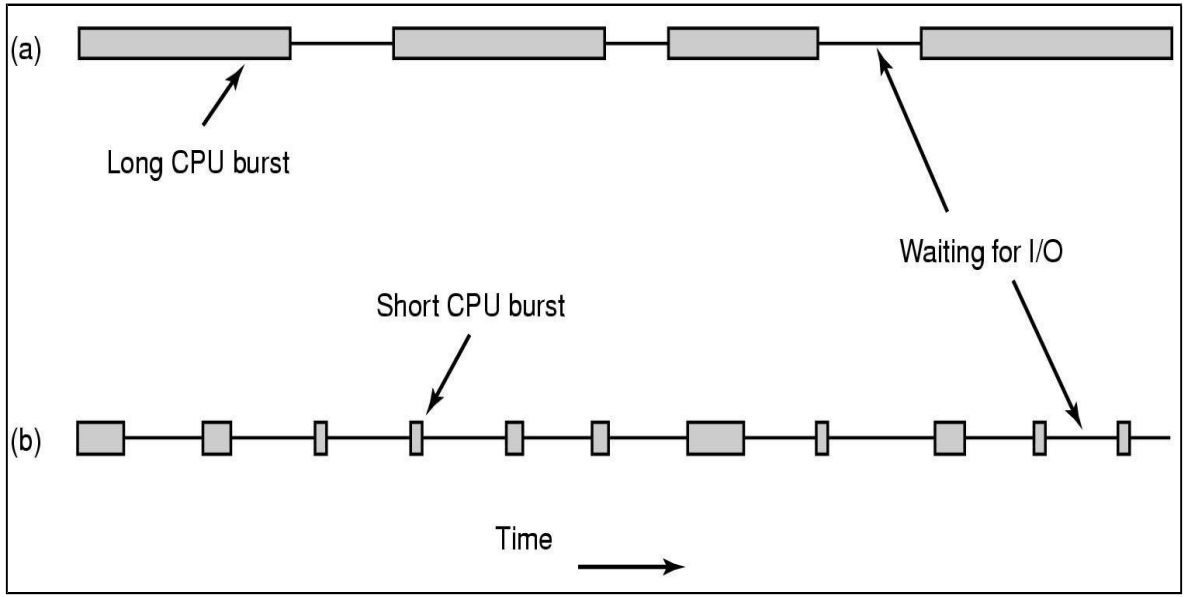
- Scheduling is a fundamental function of Operating System.
- All computer resources are scheduling before use and CPU is one of the Primary computer resource.
- Scheduling refers to a set of policies and mechanisms supported by OS that controls the order in which the work to be done.
- A scheduler is an OS program (module) that selects the next job to be admitted for execution.
- The main objectives of scheduling is to increase CPU utilization and higher throughput.

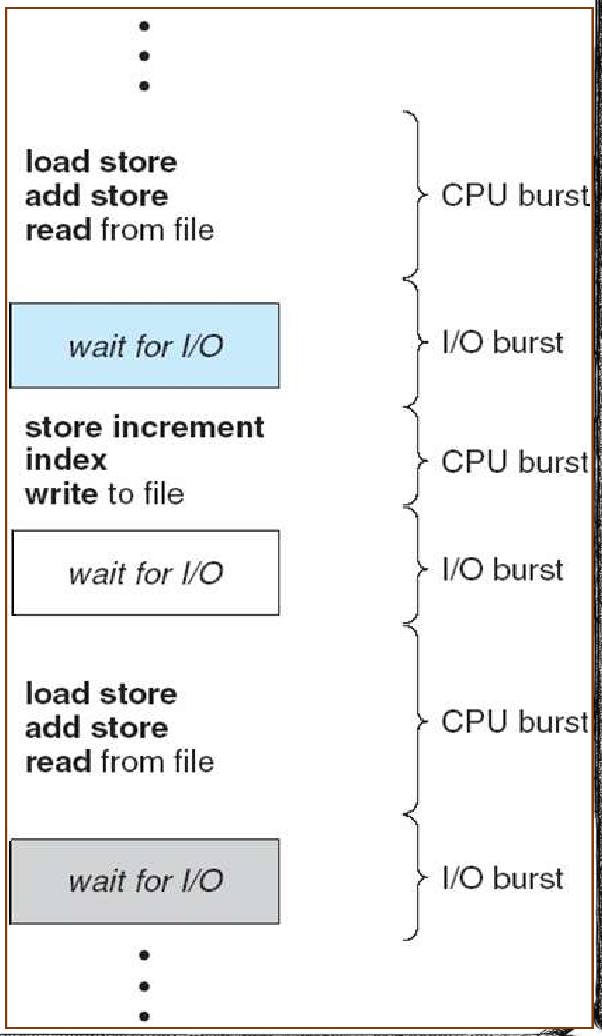
Need for Scheduling

- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.
- A typical process involves both I/O time and CPU time.
  - In a uniprogramming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time.
  - In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

CPU–I/O Burst Cycle

- 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.

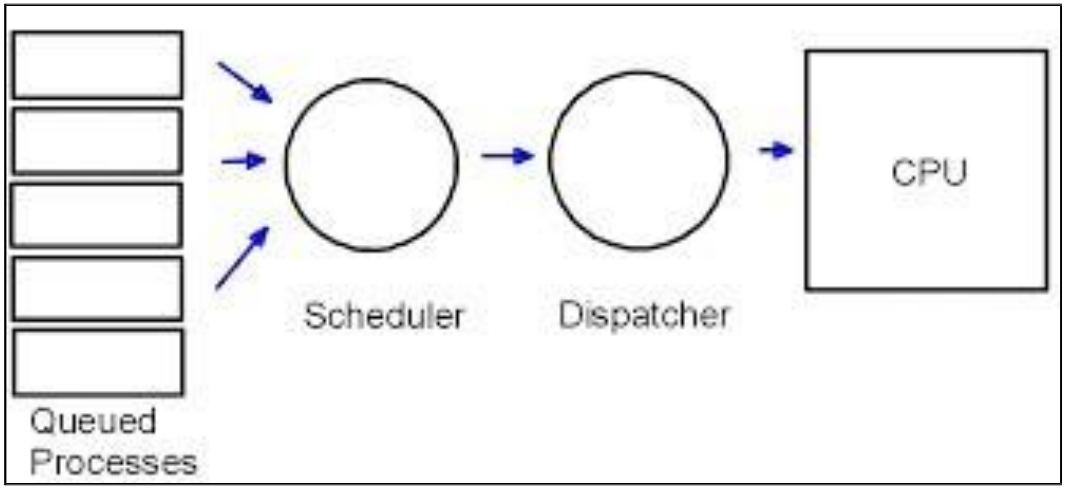


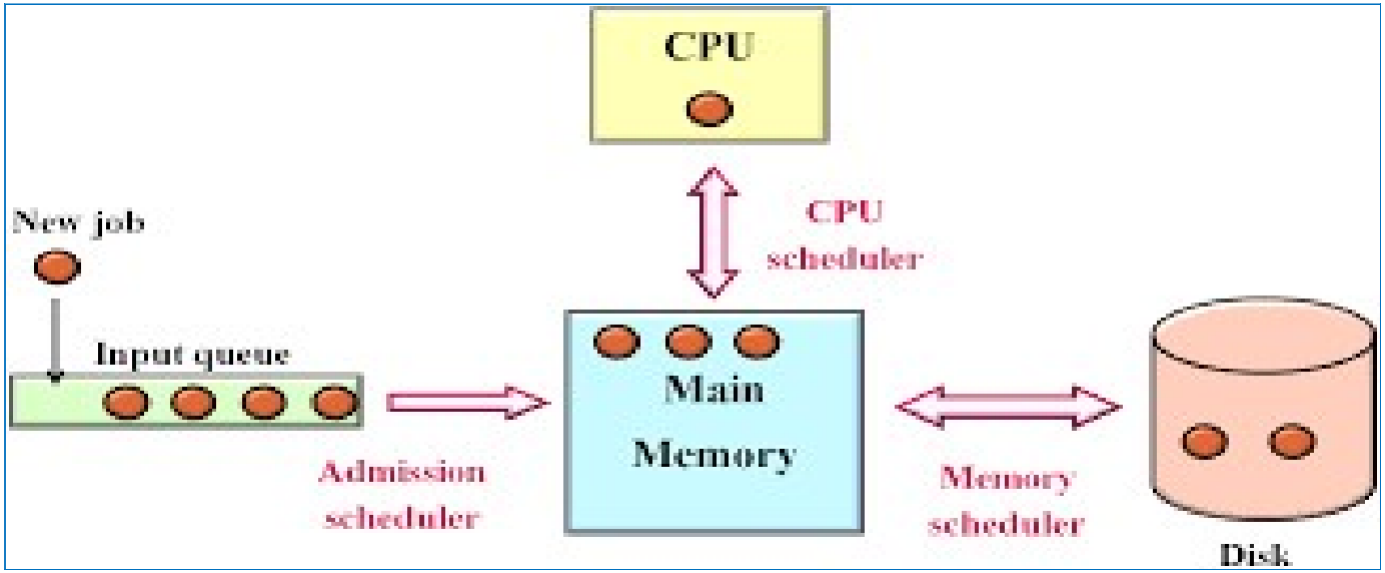


- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts.

CPU Scheduler

- The short-term scheduler, or CPU scheduler selects a process from the processes in memory that are ready to execute and allocates the CPUto that process.





Preemptive Scheduling

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

1. *When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process).*
2. *When a process switches from the running state to the ready state (for example, when an interrupt occurs)*
3. *When a process switches from the waiting state to the ready state (for example, at completion of I/O)*
4. *When a process terminates*

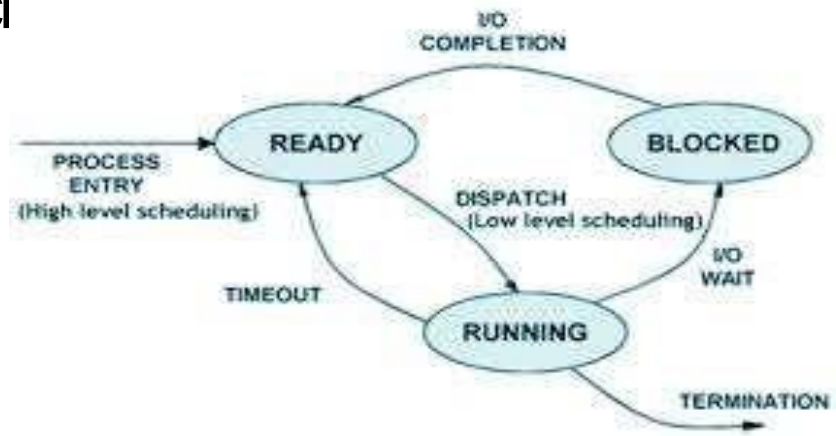
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be non-preemptive, or cooperative. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be preemptive.

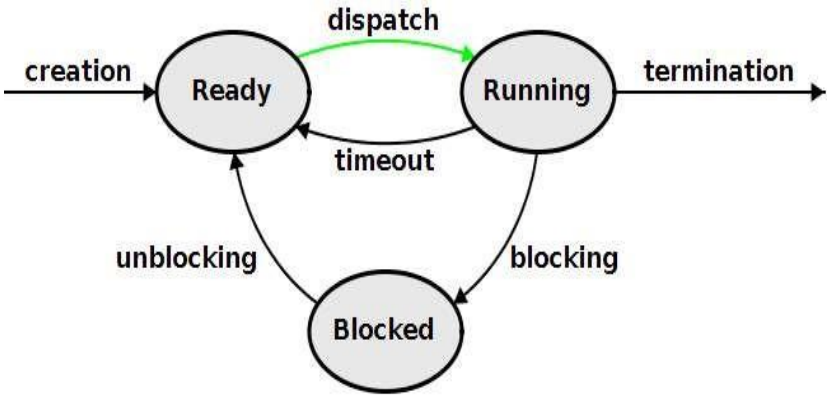
Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the scheduler. This function involves:

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the newly loaded program.

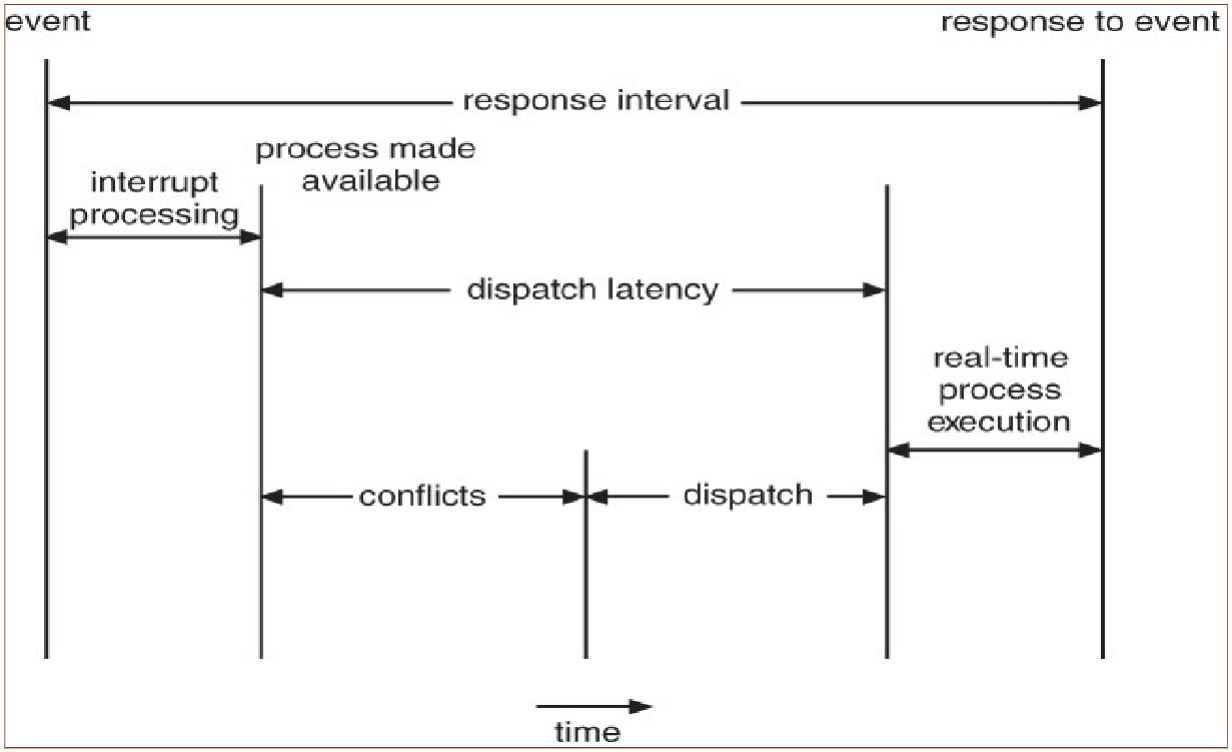
The dispatcher needs to be as fast as possible, as it is run on every context switch. Dispatch latency is the amount of time required for the scl art another.





Scheduling Criteria

- Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.



There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

- **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- **Throughput** - Number of processes completed per unit time. May range from 10/second to 1/hour depending on the specific processes.
- **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
- **Waiting time** – is the sum of the times, processes spend in the ready queue waiting their turn to get on 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).

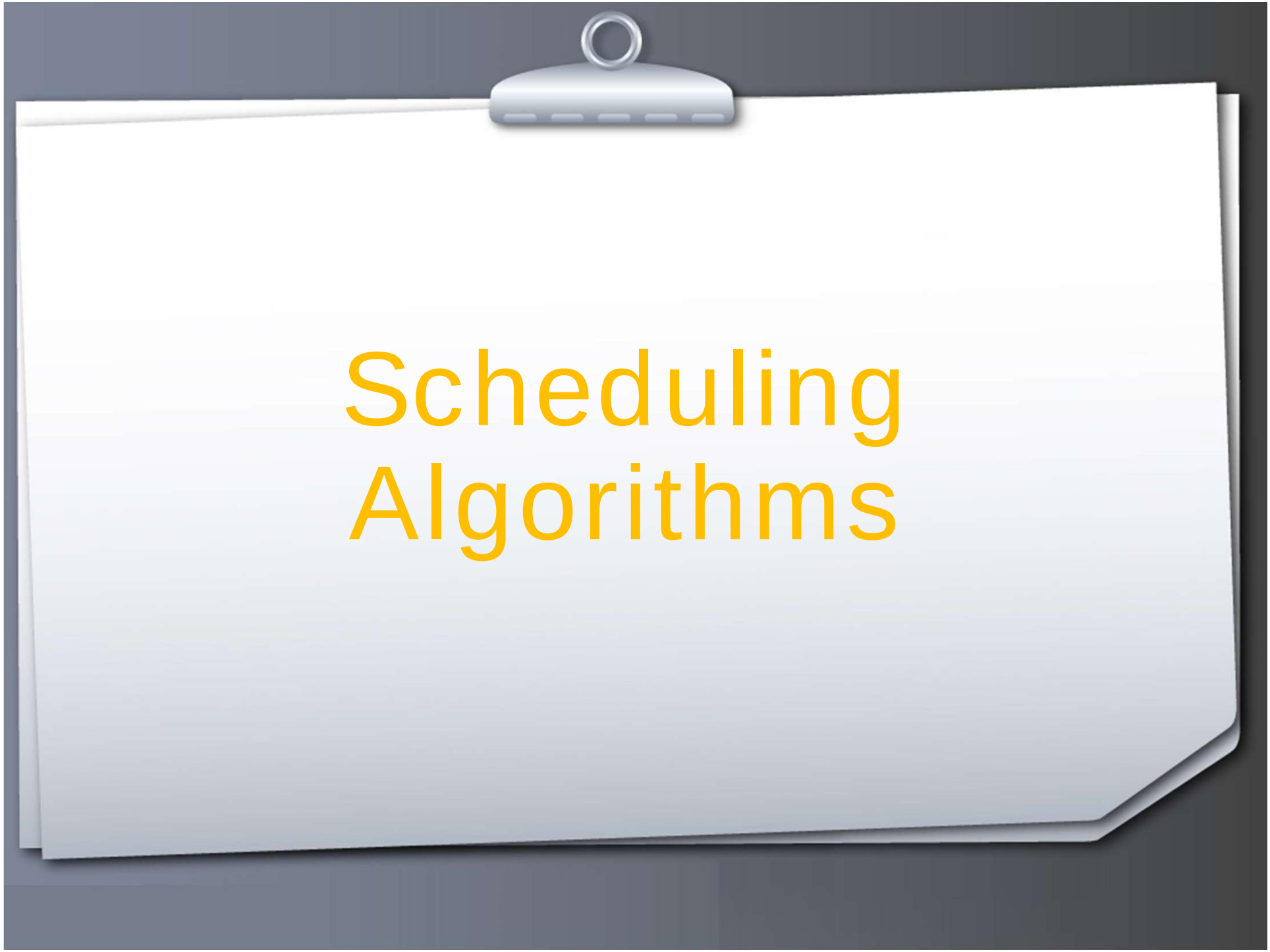
- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as to minimize the maximum response time.

- Sometimes it is most desirable to minimize the variance of a criteria than the actual value. i.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

Scheduling Types

- These algorithms are either **Preemptive** or **Non-Preemptive**.

- **Preemptive Algorithm:**
  - It is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.
- **Non-Preemptive Algorithms:**
  - These are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time.


Scheduling
Algorithms

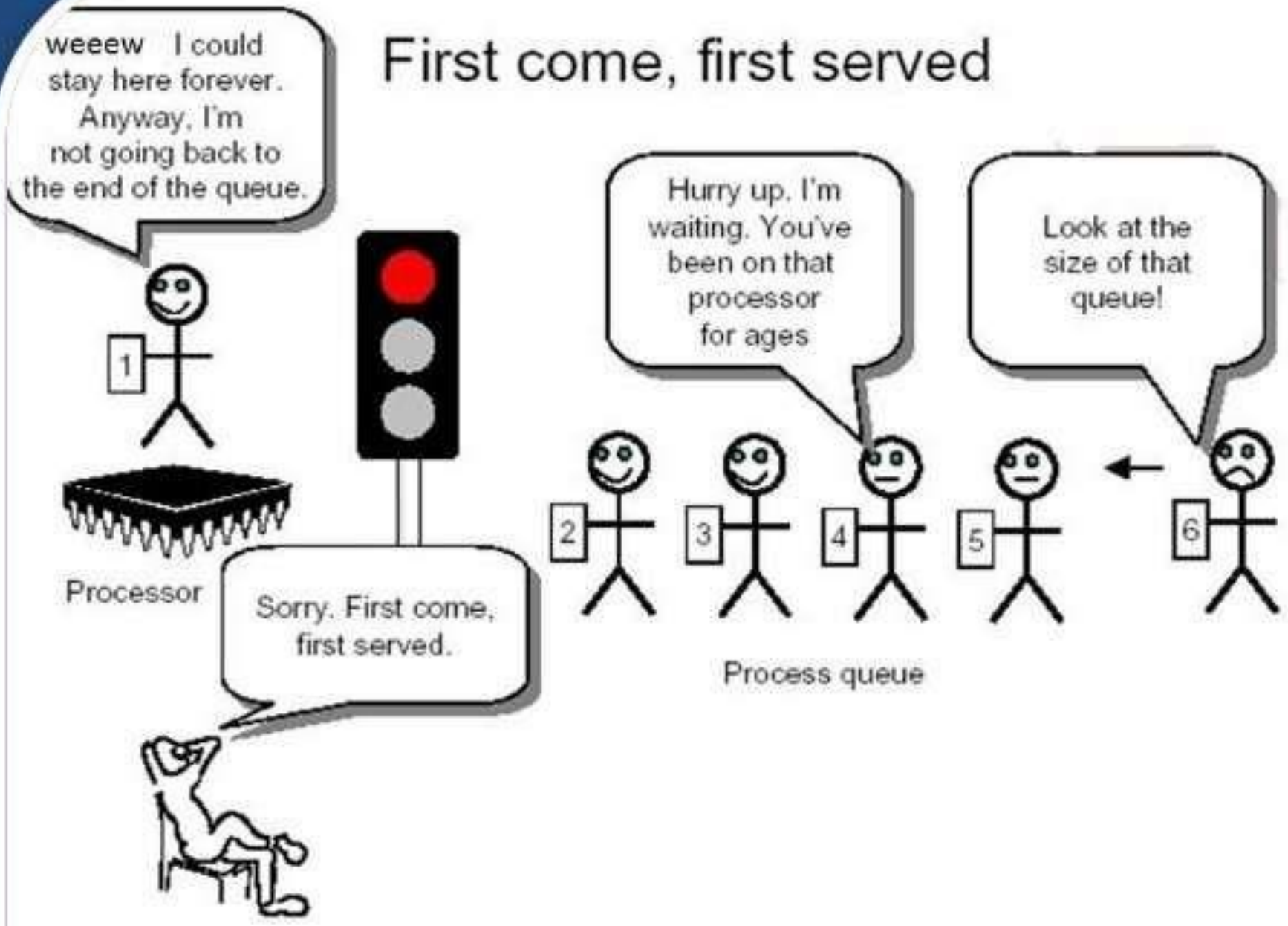
Scheduling Algorithms

- There are six popular process scheduling algorithms which we are going to discuss in this chapter:
  - First-Come First-Served (FCFS) Scheduling
  - Shortest-Job-First (SJF) Scheduling
  - Priority Scheduling
  - Shortest Remaining Time
  - Round Robin(RR) Scheduling
  - Multiple-Level Queues Scheduling

First-Come, First-Served Scheduling

- The first-come, first-served(FCFS) is the simplest scheduling algorithm.
- 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.
- On the negative side, the average waiting time under the FCFS policy is often quite long.

First come, first served



EXAMPLE

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

A Gantt chart is a horizontal bar chart developed as a production control tool in 1917 by Henry L. Gantt, an American engineer and social scientist.

Wait time of each process is as follows –

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 = 5.75

Turn arround Time

Turn arround time=Dispatch time-arrival time
P0=5-0=5
P1=8-1=7
92=16-2=14
P3=22-3=19

AVG TAT=(5+7+14+19)/4

11.2

EXAMPLE

Consider the following three processes

Process	Burst Time
P1	24
P2	3
P3	3

In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.

In the second Gantt chart below, the same three processes have an average wait time of
(0 + 24 + 27) / 3 = 17 ms. This reduction is substantial.

Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

P1=0
P2=24
P3=27
AVG WT =(0+24+27)/3
51/3
=17
TAT
P1=24
P2=27
P3=30
AVG TAT=(24+27+30)/3
81/3
=27

- FCFS can also block the system in a busy dynamic system in another way, known as the convoy effect.
  - § When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle.
  - § When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.
- The FCFS scheduling algorithm is nonpreemptive.
  - § 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.

Shortest-Job-First Scheduling

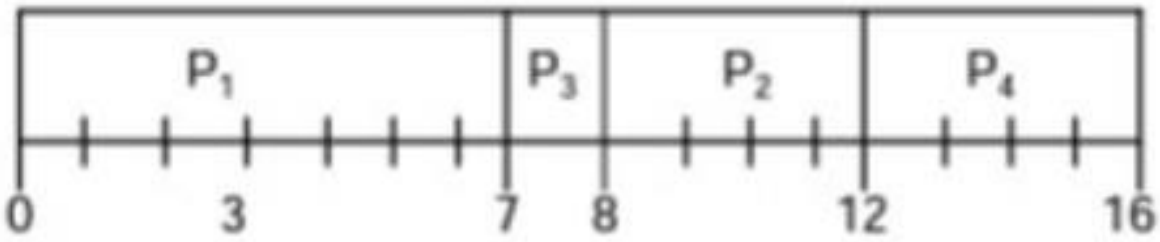
- Shortest-job-first (SJF) scheduling algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.

EXAMPLE

Consider the following processes

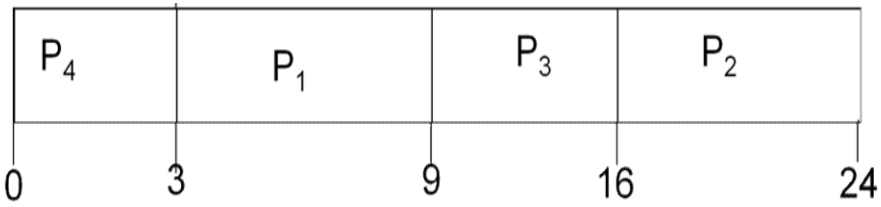
Example: Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Gantt Chart representation is:

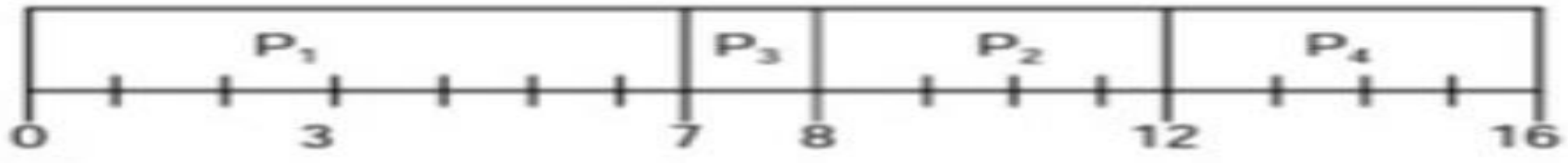


Average waiting time = (0 + 6 + 3 + 7) / 4 = 4

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	0.0	8
P_3	0.0	7
P_4	0.0	3



The average waiting time is (3 + 16 + 9 + 0) / 4 = 7 milliseconds



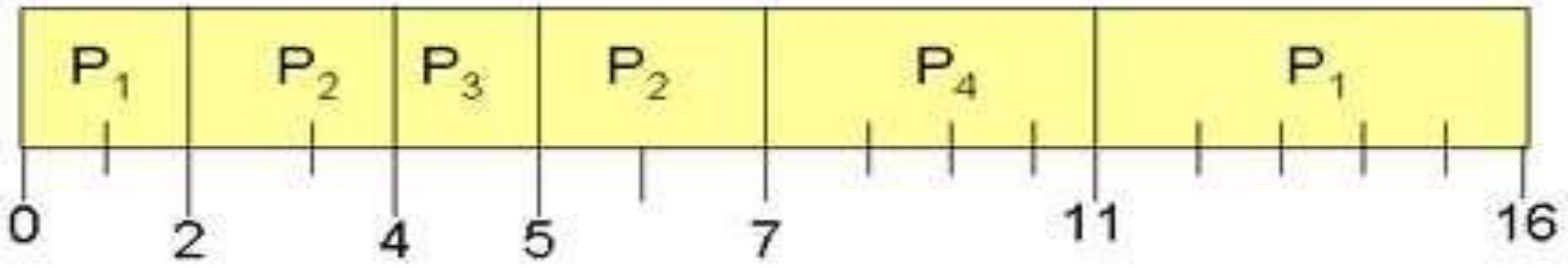
Wt p1=0-0=0
P2=8-2=6
P3=7-4=3
P4=12-5=7
AWT=(0+6+3+7)/4
=4

TAT
P1=7-0=7
P2=12-2=10
P3=8-4=4
P4=16-5=11
ATAT=(7+10+4+11)/4
=32/4
=8

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.

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

- SJF (preemptive)



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

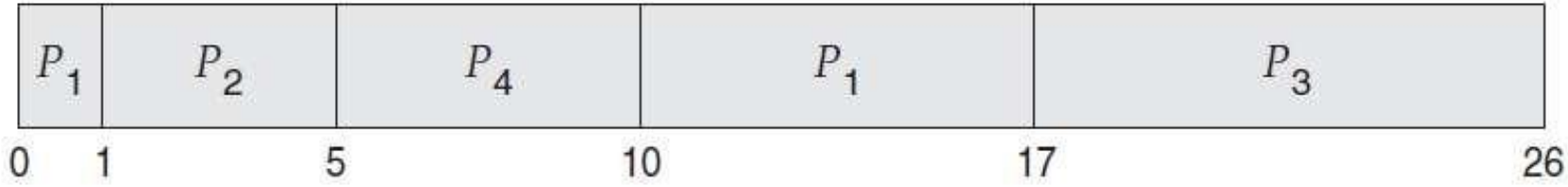
Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling (SRTF)

EXAMPLE

Consider the following processes

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Gantt Chart representation is:



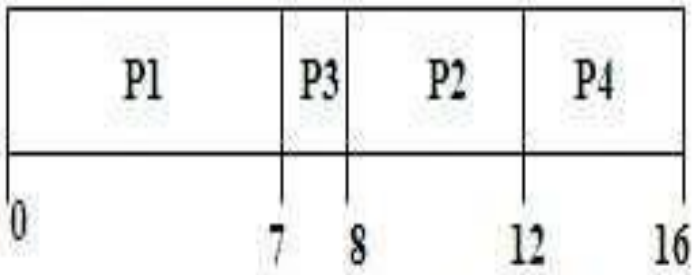
- Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds.
- A nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

EXAMPLE

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-Preemptive SJF Scheduling

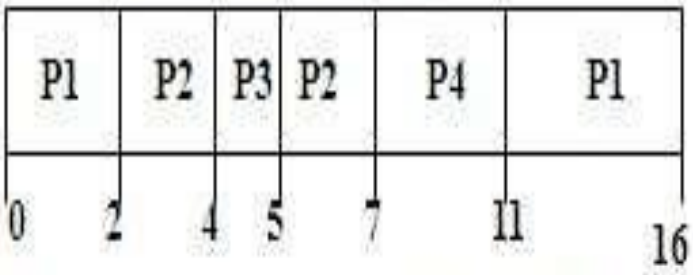
Gantt Chart for Schedule



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

Preemptive SJF Scheduling

Gantt Chart for Schedule



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

- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?
- For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.
- Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.
- A more practical approach is to predict the length of the next burst, based on some historical measurement of recent burst times for this process. One simple, fast, and relatively accurate method is the exponential average of the measured lengths of previous CPU bursts.

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.
- In practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers.

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

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

Gantt Chart representation is:

Average waiting time = (0 + 1 + 6 + 16 + 18) / 5 = 8.2

The average waiting time is 8.2 milliseconds

TAT

P1=16

P2=1

P3=18

P4=19

P5=6

Avg TAT=(16+1+18+19+6)/5

=60/5

=12

EXAMPLE

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

Try this!!!!

The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Now Try this!!!!

Process	Burst Time (mills.)	Priority	Arrival Time (mills)
P1	9	5	0
P2	4	3	1
P3	5	1	2
P4	7	2	3
P5	3	4	4
Total	28		

The average waiting time is 9.6 milliseconds

- Priorities can be assigned either internally or externally.
  - Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel.
  - External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
  - When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
  - A 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 can suffer from a major problem known as **indefinite blocking, or starvation**, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
  - If this problem is allowed to occur, then processes will either run eventually when the system load lightens, or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
  - One common solution to this problem is aging, in which priorities of jobs increase the longer they wait.
  - Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

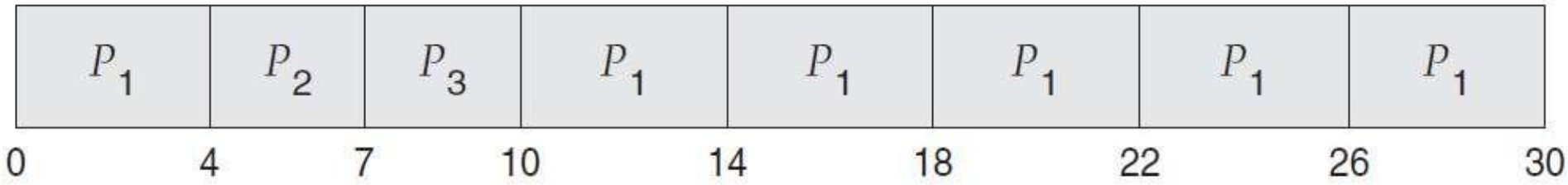
Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called time quantum.
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
- If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
- If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.

- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms.

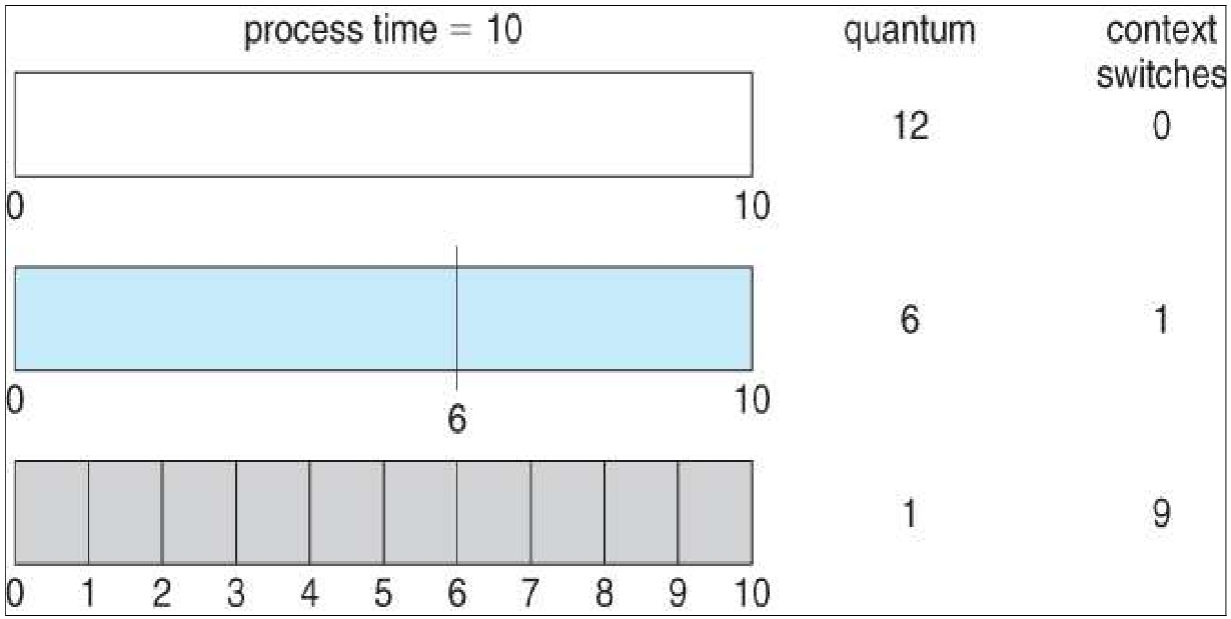
Consider this example of three processes and a time quantum of 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



The average waiting time is calculated for this schedule. P1 waits for 6 milliseconds (10 - 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.
- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- BUT, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are.
- Turnaround time also depends on the size of the time quantum. In general, turnaround time is minimized if most processes finish their next cpu burst within one time quantum.



- The way in which a smaller time quantum increases context switches.
- A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Advantages

- Jobs get fair share of CPU
- Shortest jobs finish relatively quickly

Disadvantages

- Poor average waiting time with similar job lengths
  - Example: 10 jobs each requiring 10 time slices
  - RR: All complete after about 100 time slices
  - FCFS performs better!
- Performance depends on length of time quantum

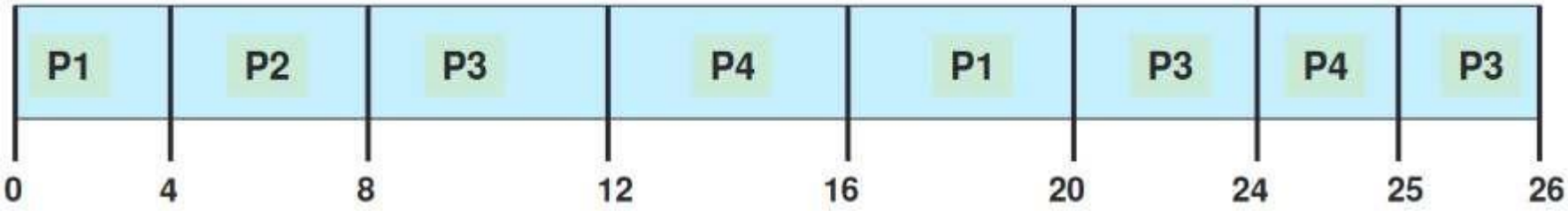
EXAMPLES

Process	Arrival Time	Service Time
1	0	8
2	1	4
3	2	9
4	3	5

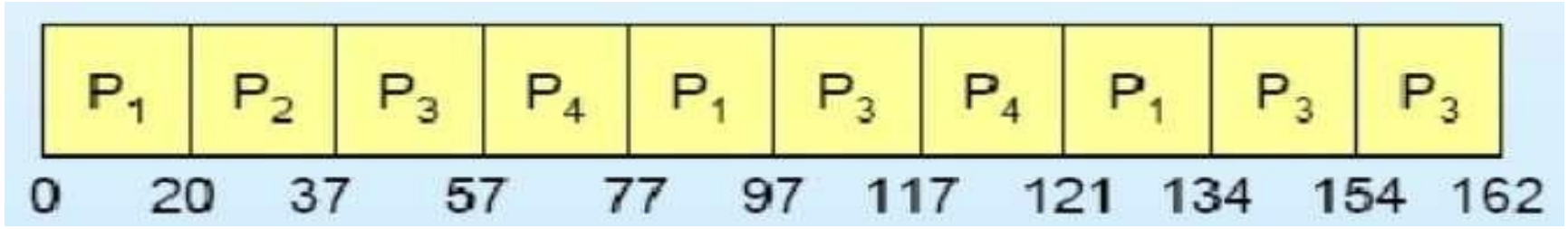
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Time quantum : 20

Round Robin, quantum = 4



Average wait = ((20-0) + (8-1) + (26-2) + (25-3))/4 = 74/4 = 18.5



Average Waiting Time = (81+20+94+97) / 4 = 73

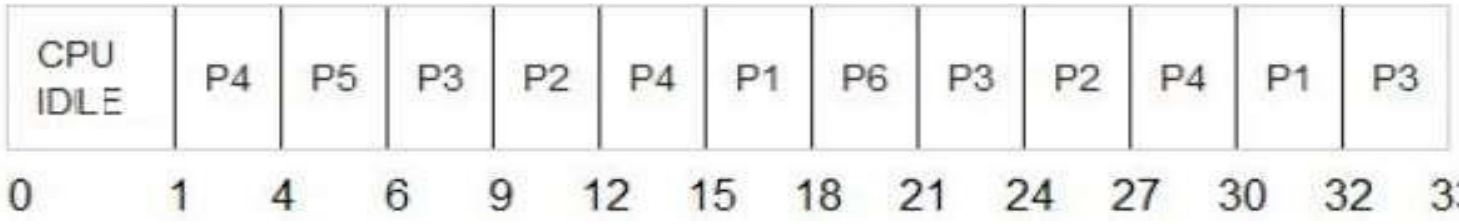
Practice Problem

Q). Consider the following processes with arrival time and burst time. Calculate average turnaround time, average waiting time and average response time using round robin with time quantum 3?

Process id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

Solution

Queue: P4,P5,P3,P2,P4,P1,P6,P3,P2,P4,P1,P3



Process id	Arrival time	Burst time	Completion time	Turnaround time	Waiting time	Response time
P1	5	5	32	27	22	10
P2	4	6	27	23	17	5
P3	3	7	33	30	23	3
P4	1	9	30	29	20	0
P5	2	2	6	4	2	2
P6	6	3	21	15	12	12

Average turnaround time= $\frac{(27+23+30+29+4+15)}{6}$ =21.33

Average waiting time= $\frac{(22+17+23+20+2+12)}{6}$ =16

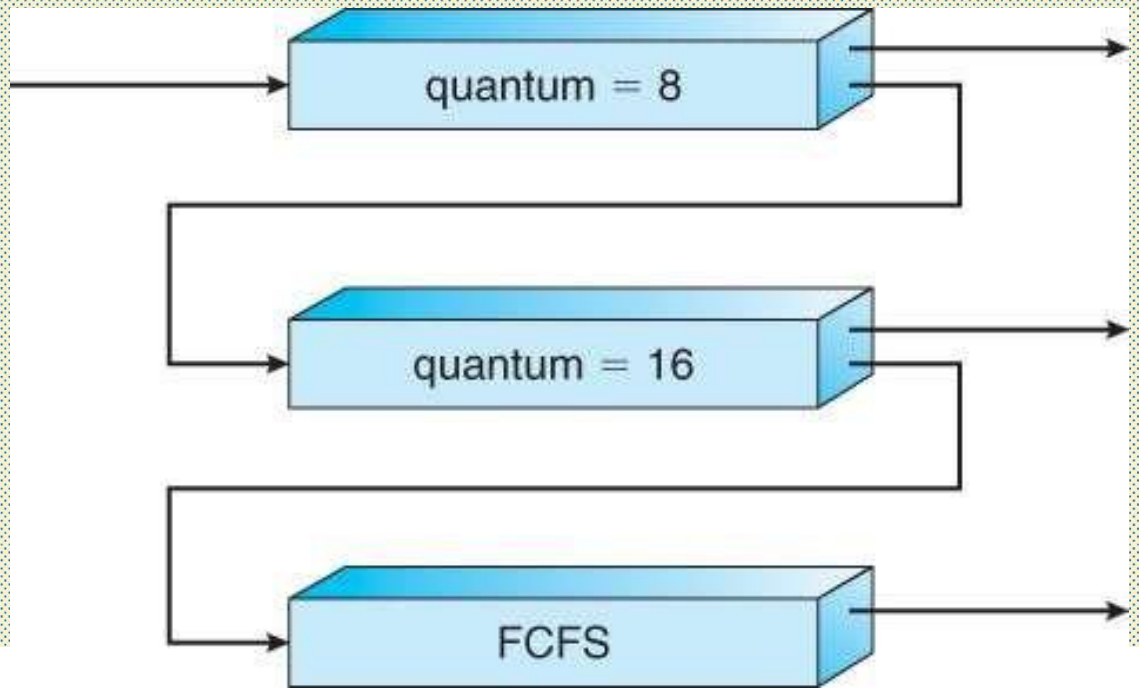
Average response time= $\frac{(10+5+3+0+2+12)}{6}$ =5.33

Multilevel Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

Multilevel Feedback-Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - *If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.*
  - *Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.*
- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
  - *The number of queues.*
  - *The scheduling algorithm for each queue.*
  - *The methods used to upgrade or demote processes from one queue to another. (Which may be different.)*
  - *The method used to determine which queue a process enters initially.*



Three queues:

- $Q0$ – RR with time quantum 8 milliseconds
- $Q1$ – RR time quantum 16 milliseconds
- $Q2$ – FCFS

Scheduling

A new job enters queue *Q0* which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q1$.

At $Q1$ job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q2$.

Memory Management: Introduction

- Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.
- Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free.
- It checks how much memory is to be allocated to processes.
- It decides which process will get memory at what time.
- It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Memory Management: Introduction

- The utilization and performance of several processes to single processor.
- This is achieved by keeping all the processes in primary memory.
- The sharing of memory is very essential to improving the performance of CPU.
- No process can ever run before a certain amount of memory is allocated to it.
- The overall resource utilization and other performance criteria of a computer system are largely affected by performance of memory of management

Memory Management: Features

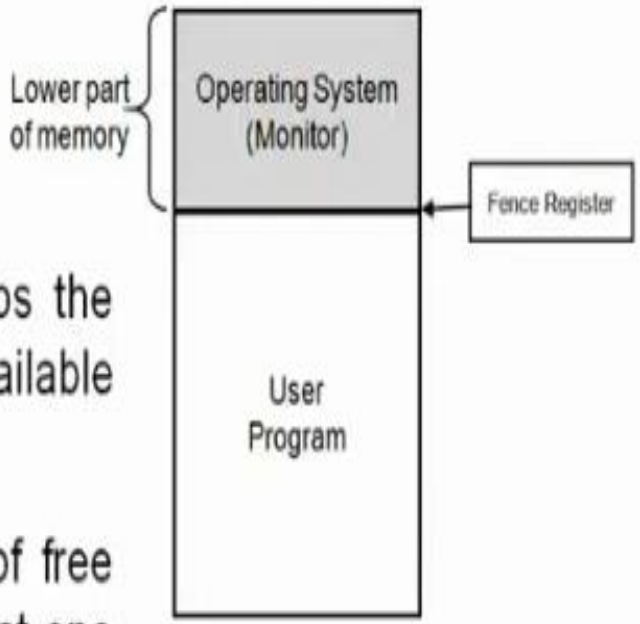
- Two important features of memory management are
  - Protection
  - Sharing

- Protection:
  - In order to protect one process from another, their address space must be safe by memory management scheme.
  - An active process should never attempt to access wrongly or maliciously and destroy the content of each other's address space.

- Sharing
  - Apart from protection, memory management scheme must support sharing of common data or data structure such as symbol table in compiler or assembler

Single Process Monitor

- This is the simple memory management approach.
- The memory is divided into two contiguous sections.
  - First section contains operating system program also called monitor.
  - Second section is for user program.



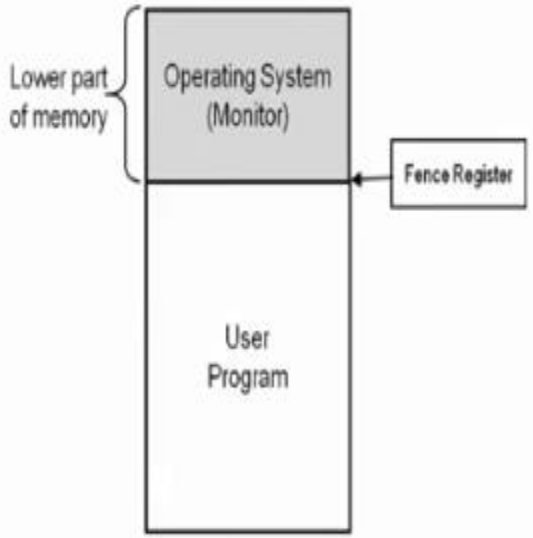
- In this type of approach, OS only keeps the track of the first and the last location available for allocation of user programs.
- In order to provide a contiguous area of free storage for user program, OS is loaded at one extreme end, either at the bottom or at the top.

Single Process Monitor

- A new program (user process) is loaded only when the OS passes a control to it.
- After receiving a control by the process, it starts running until its completion or termination due to I/O or some error.
- When this program is completed or terminated, the OS may load another program for execution.
- Two important issues such as protection and sharing of code must be addressed while designing any memory management scheme.
  - Sharing of code and data in a single process environment does not make much sense because only one process (program) reside in memory at a time.
  - Protection is also hardly supported by a single process monitor because only one process is memory.
- Protection of OS program from user code is must otherwise it may crash

Single Process Monitor

- The protection is supported through hardware mechanism such as a dedicated register are as follows:
  - OS codes usually resides in low memory area.
  - A register is set to the highest address occupied by OS code.
  - A memory address generated by user program to access certain memory location is first compared with fence register's content.
  - If the address is generated is below the fence, it will be trapped and denied permission.



Single Process Monitor: Advantages & Disadvantages

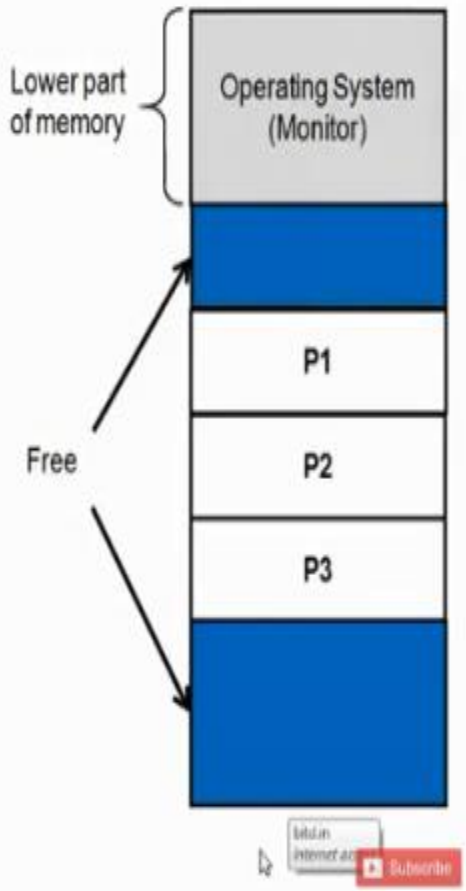
- It is a simple memory management approach.
- Due to lack of support of multiprogramming results in lower utilization of CPU and memory capacity.
- Since only one program is residing at a time in memory, it may not occupy whole memory, therefore memory is under utilized.
- CPU will be sitting idle during the period when a running program requires some I/O.

Multiprogramming

- In multiprogramming environment several program reside in primary memory at a time and the CPU passes its control rapidly between these program.
- One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process.
- There may be two types of memory partitioning:
  - **Static:** In this the division of memory into number of predefined partition and its size is made in the beginning and remain fixed thereafter.
  - **Dynamic:** The size and the number of partitions are decided during the run time by the OS.

Multiprogramming with Fixed Partition

- Basic approach is to divide the memory into several fixed size partitions where each partition will accommodate only program for execution.
- The number of programs residing in memory will be bound by the number of partitions.
- When a program terminates, that partition is free for another program waiting in a queue.



Fixed Partition Table

- Once partitions are defined, OS system keeps track of status of memory location either allocated or free.
- This is done through a data structure called Partition Description Table.

Sr. No.	Starting Address of Partition	Size of Partition	Partition Status
1	0K	200K	Allocated
2	200K	200K	Free
3	400K	200K	Allocated
4	600K	300K	Allocated
5	900K	100K	Allocated
6	1000K	100K	Allocated

Multiprogramming with Fixed Partition

- The two most common strategies to allocate free partition to ready process are:
  - **First-fit:**
    - ✓ It is to allocate the first free partition large enough to accommodate the process.
  - **Best-bit:**
    - ✓ It is allocates the smallest free partition that meets the requirement of the process
- Both strategies require to scan the partition description table to find out free partitions.
- **Conclusion:** The **first-fit** terminates after finding the first such partition whereas the **best-fit** continues searching for the near exact size.

Multiprogramming with Fixed Partition: Process Swaping

- Whenever a new process is ready to be loaded into memory and if no partition is free, swapping of process between main memory and secondary storage Is done.
- Swapping helps in CPU utilization by replacing suspendable process but residing into main memory with ready to execute process from secondary storage.
- When the scheduler admits a new process of high priority for which no partition is free, a memory manager is invoked to make a partition free to accommodate the process.
- The memory manager performs this task by swapping out low priority process suspended for a comparatively long time in order to load and execute the higher priority process.
- When the higher priority process is terminated, the lower priority process can be swapped back and continued

Fixed Partition Restriction

1. No single program/process may exceed the size of the largest partition in a given system.

2. It does not support a system having dynamically data structure such as stack, queue, heap etc.

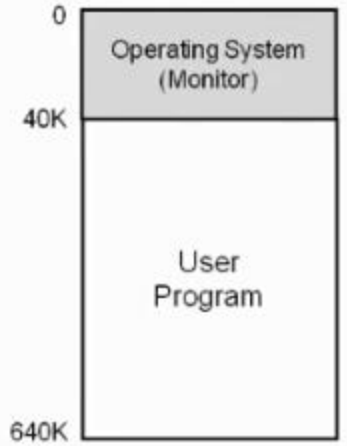
3. It limits the degree of multiprogramming which in turn may reduce the effectiveness of short-term scheduling.

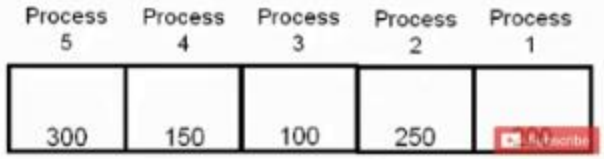
Multiprogramming with Dynamic/Variable Partition

- Creates partitions dynamically to meet the requirements of each requesting process.
- When the process terminates to becomes swapped-out, the memory manager can return the vacated space to the pool of free memory areas from which partition allocation are made.
- In this partition, neither the size nor the number of dynamically allocated partition need be limited at any other time.
- Memory manager continues creating and allocating partitions to requesting process until all physical memory is exhausted or maximum allowable degree of multiprogramming is reached.
- Dynamic partition approach improves memory utilization but it also complicates the process of allocation and deallocation of memory.

Multiprogramming with Dynamic Partition: Example

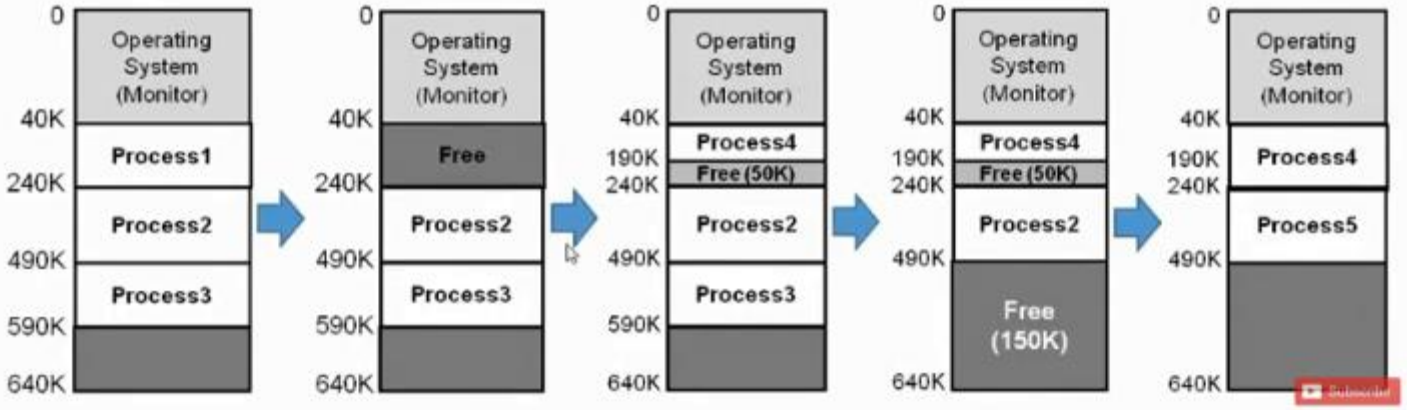
- Suppose a system have 640K main memory in which 40K is occupied by OS program.
- There are 5 jobs waiting for memory allocation in a job queue.
- Applying FCFS scheduling policy, Process1, Process2, and Process3, immediately allocated in memory.
- Process4 cannot be accommodated because there is only 600-550 = 50K left for it.





Memory Allocation and Job Scheduling

- Let us assume that after some time Process1 is terminated, releasing 200K memory space.
- Next Process4 is swapped-in in the memory which is shown in the figure.
- After Process1, Process3 gets terminated releasing 100K memory but Process cannot be accommodated due to external fragmentation.
- After the swapping out of Process2 due to termination, Process5 will be loaded for execution.



Variable Partition Problem

- The main problem is external fragmentation.
- It exists when the size of memory is large enough for a requesting process, but it cannot satisfy a request because it is not contiguous.
- Storage is fragmented into a small number of free spaces.
- Depending upon the total size of memory and number and size of a program, external fragmentation may be either a minor or a major problem.
- One solution to this problem is Compaction.
  - It is possible to combines the free spaces into a large block by pushing all the processes downwards as far as possible.

Variable Partition

Advantages

- Memory utilization is generally better than fixed size partitions, since partition are created accordingly to the size of process.
- Support processes whose memory requirement increase during their execution.

Disadvantages

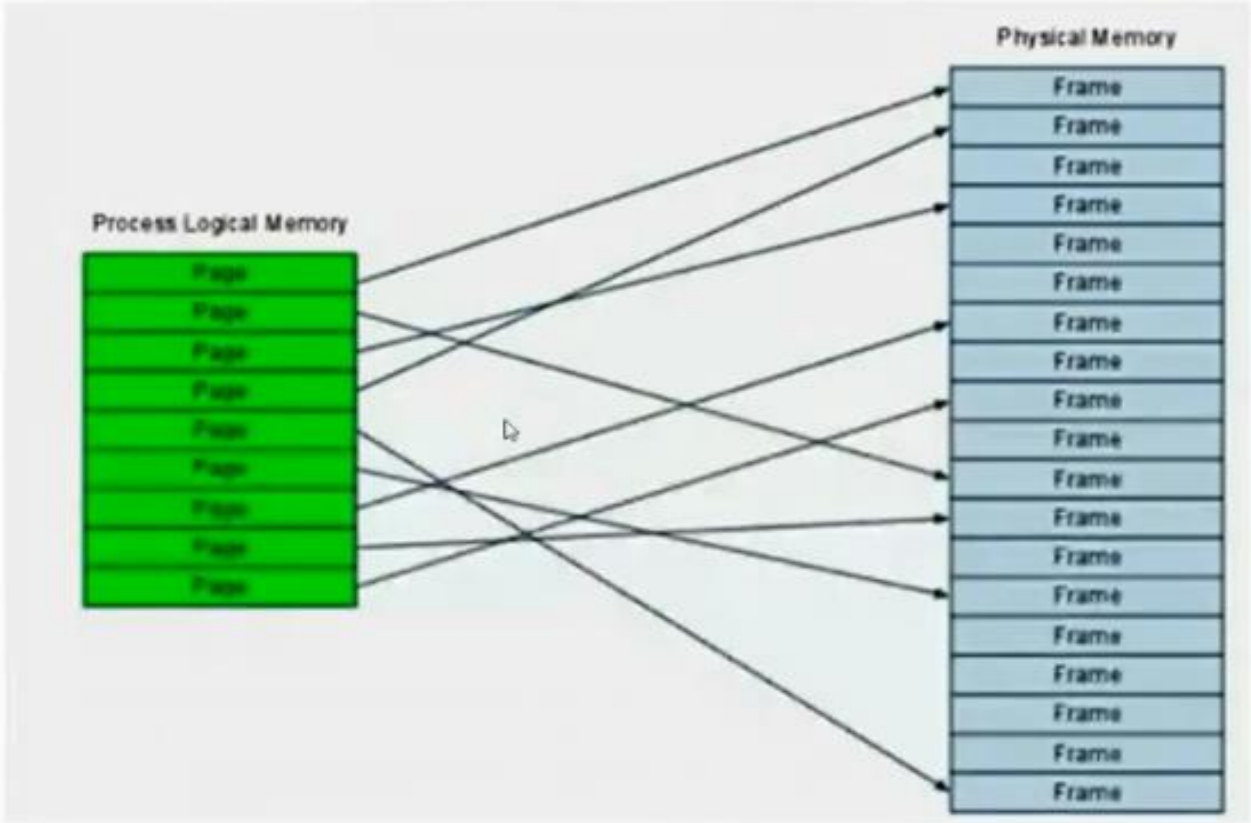
- Requires lots of operating system space, time, complex memory management algorithm.
- Compaction time is very high. Although internal fragmentation is negligible, external fragmentation may be strict problem imposing a time penalty for compaction.

Paging

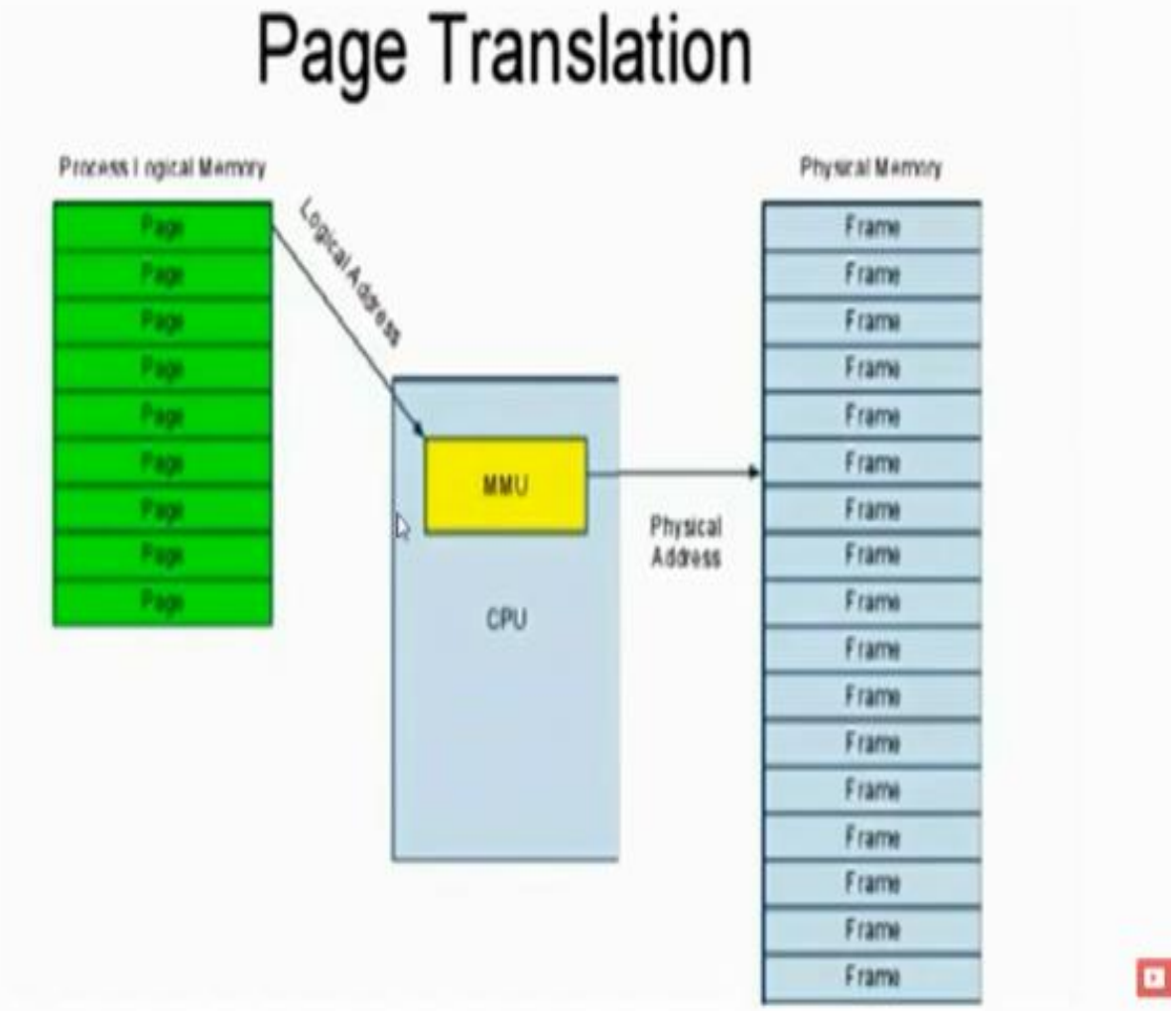
Basic Definition

- Memory-management technique that permits the physical address space of a process to be noncontiguous.
- Physical memory is divided into fixed size blocks called FRAMES.
- Logical memory is divided into blocks of the same size called PAGES.
- A FRAME
  - It has the same size as a page
  - It is a place where a (logical) page can be (physically) placed

Address Mapping

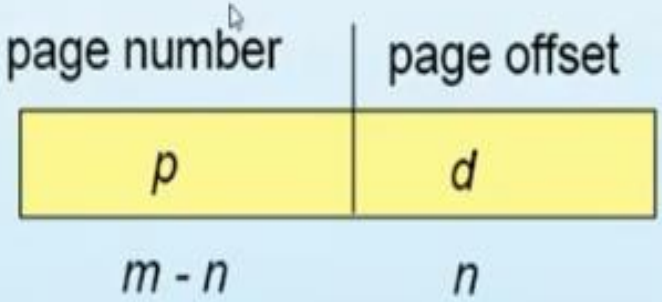


Variable Partition Problem



Paging Hardware

- Every address generated by the **CPU** is divided into two parts: Page number (p) and Page offset (d)
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

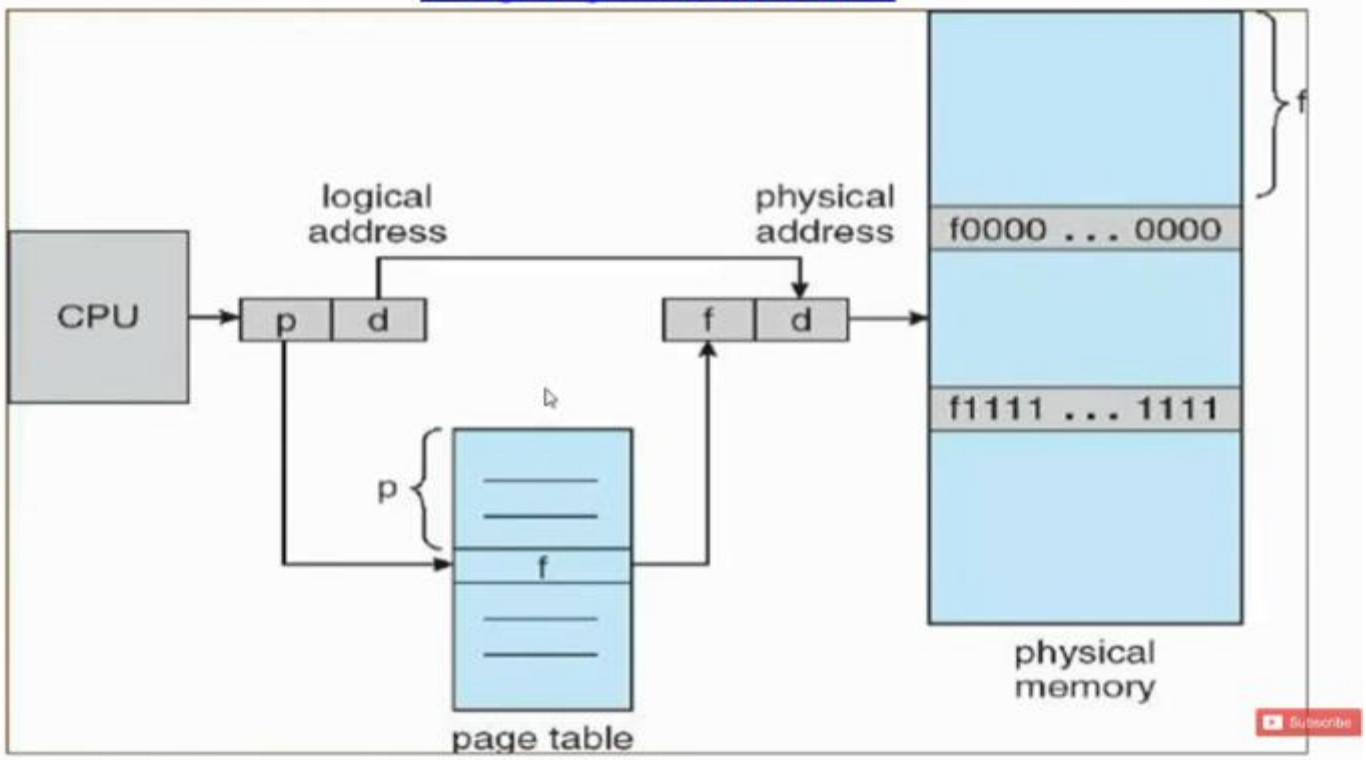


  - For given logical address space 2^m *and page size* 2^n

Paging Hardware

- The page number is used as an index into a Page Table
  - The page size is defined by the hardware
  - The size of a page is typically a power of 2, varying between 512 bytes and 16MB per page
  - **Reason:** If the size of logical address is 2^m and page size is 2^n, then the high-order m-n bits of a logical address designate the page number

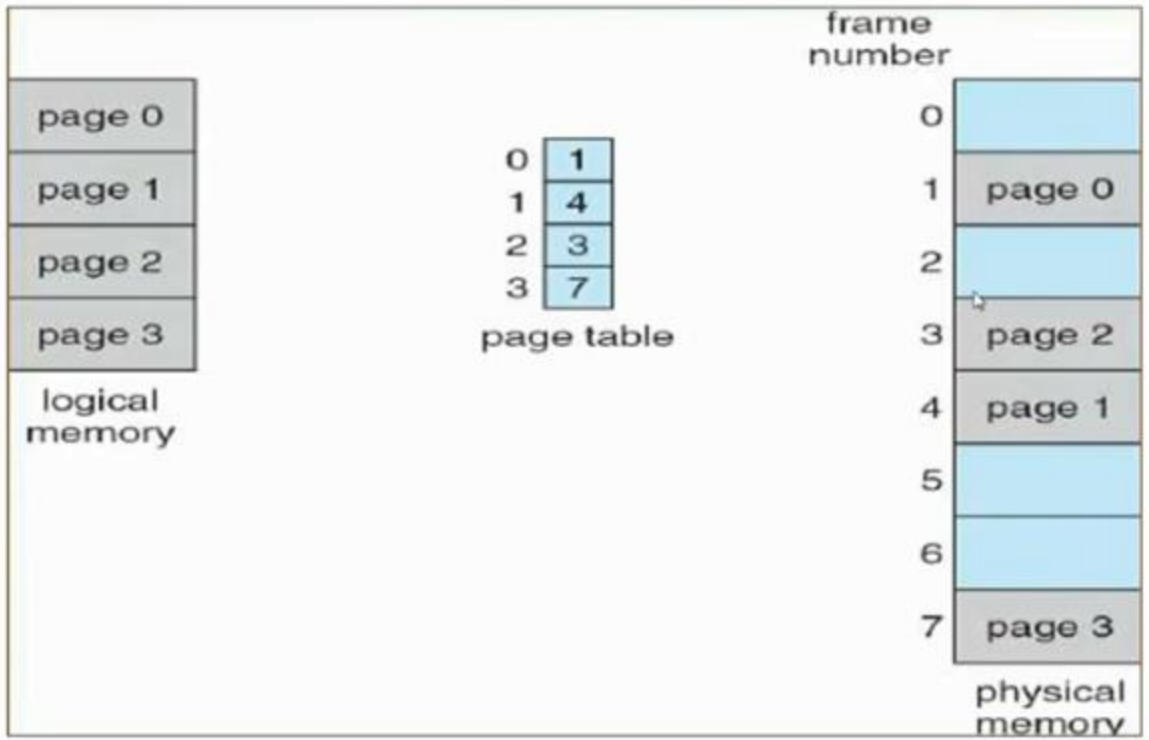
Paging Hardware



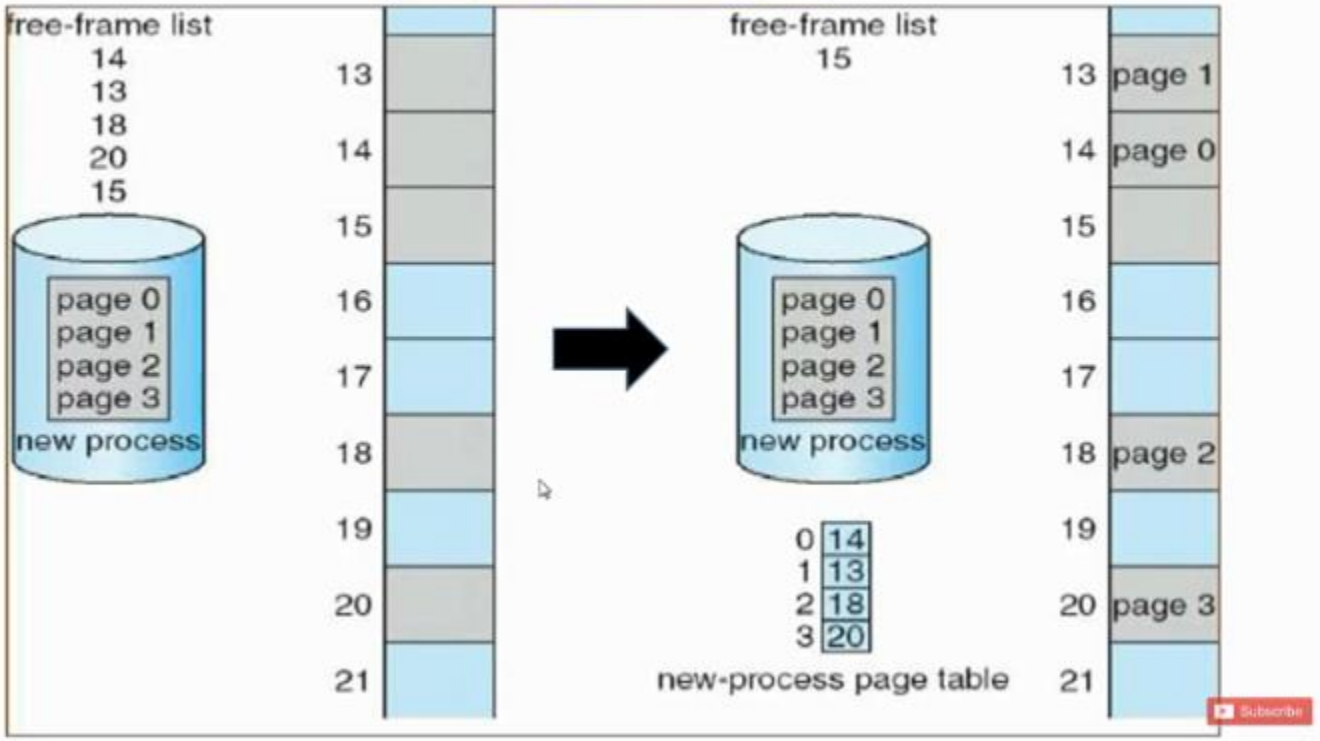
Paging Example

- When we use a paging scheme, we have no external fragmentation:
  - ANY free frame can be allocated to a process that needs it.
  - However, we may have internal fragmentation
  - If the process requires n pages, at least n frames are required
  - The first page of the process is loaded into the first frame listed on free-frame list, and the frame number is put into page table

Paging Example



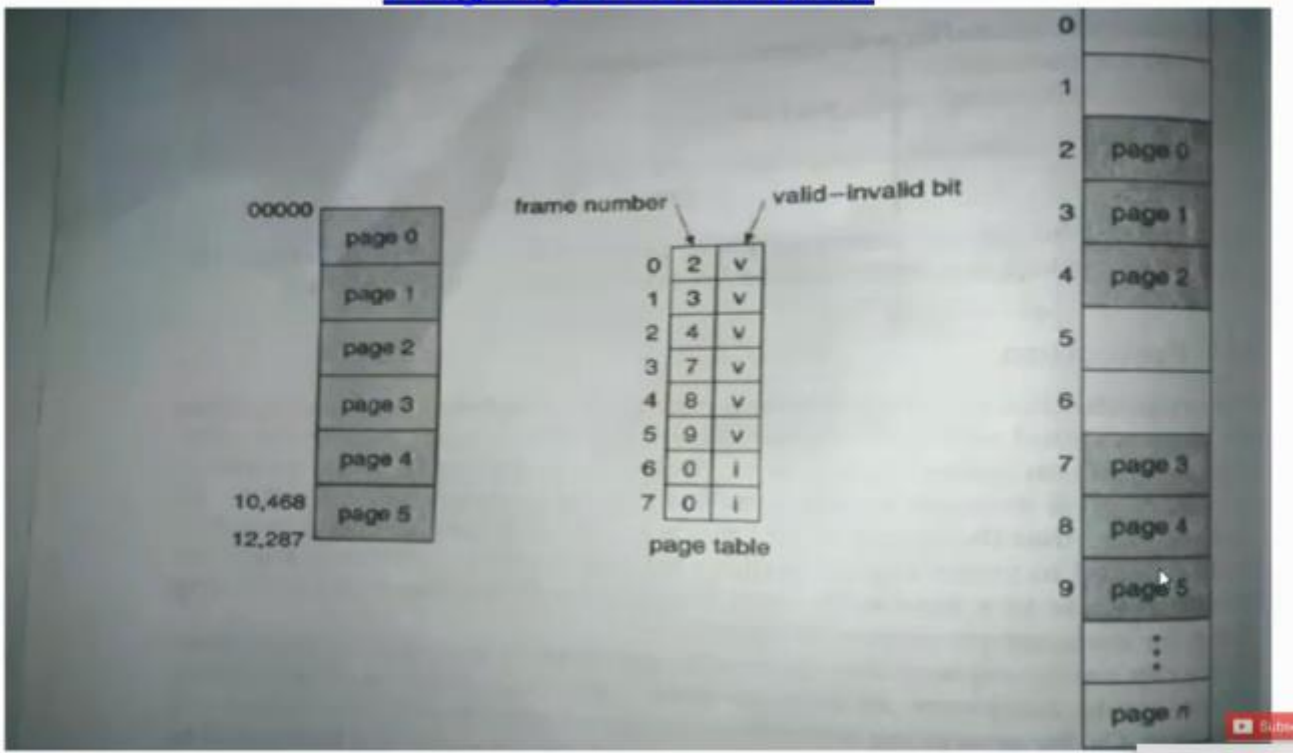
Paging Example



Implementing Page Table

- To implement paging, the simplest method is to implement the page table as a set of registers
  - However, the size of register is limited and the size of page table is usually large
  - Therefore, the page table is kept in main memory
- If we want to access any location, we must first index into the page table
- This requires at least one memory access
- The standard solution is to use a special, small, fast cache, called **Page Table**

Paging Protection

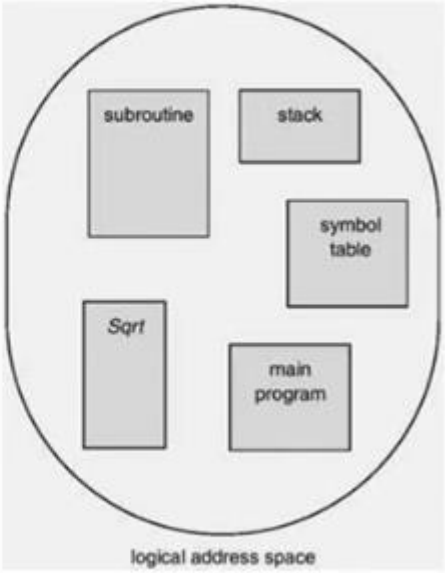


Advantages and Disadvantages

- **ADVANTAGES**
  - No external Fragmentation
  - Simple memory management algorithm
  - Swapping is easy (Equal sized Pages and Page Frames)
  - Share common code especially in a time-sharing environment
- **DISADVANTAGES**
  - Internal fragmentation
  - Page tables may consume more memory.
  - Multi level paging leads to memory reference overhead.

Segmentation: Introduction

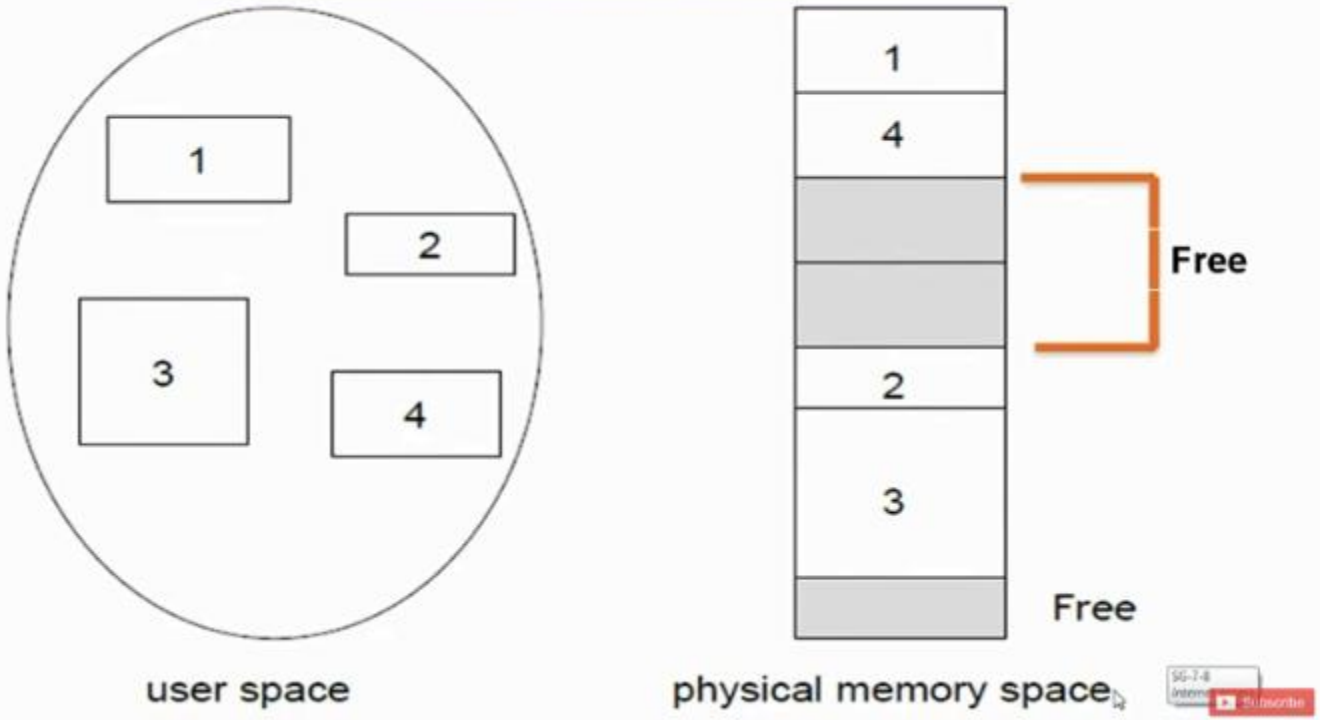
- Segmentation is one of the most common ways to achieve memory protection.
- Because internal fragmentation of pages takes place, the **user's view of memory is lost**
- The user will view the memory as a combination of segments
- In this type, memory addresses used are not contiguous
- Each memory segment is associated with a specific length and a set of permissions.
- When a process tries to access the memory it is first checked to see whether it has the **required permission to access the particular memory** segment and whether it is within the **length specified by that particular** memory segment.



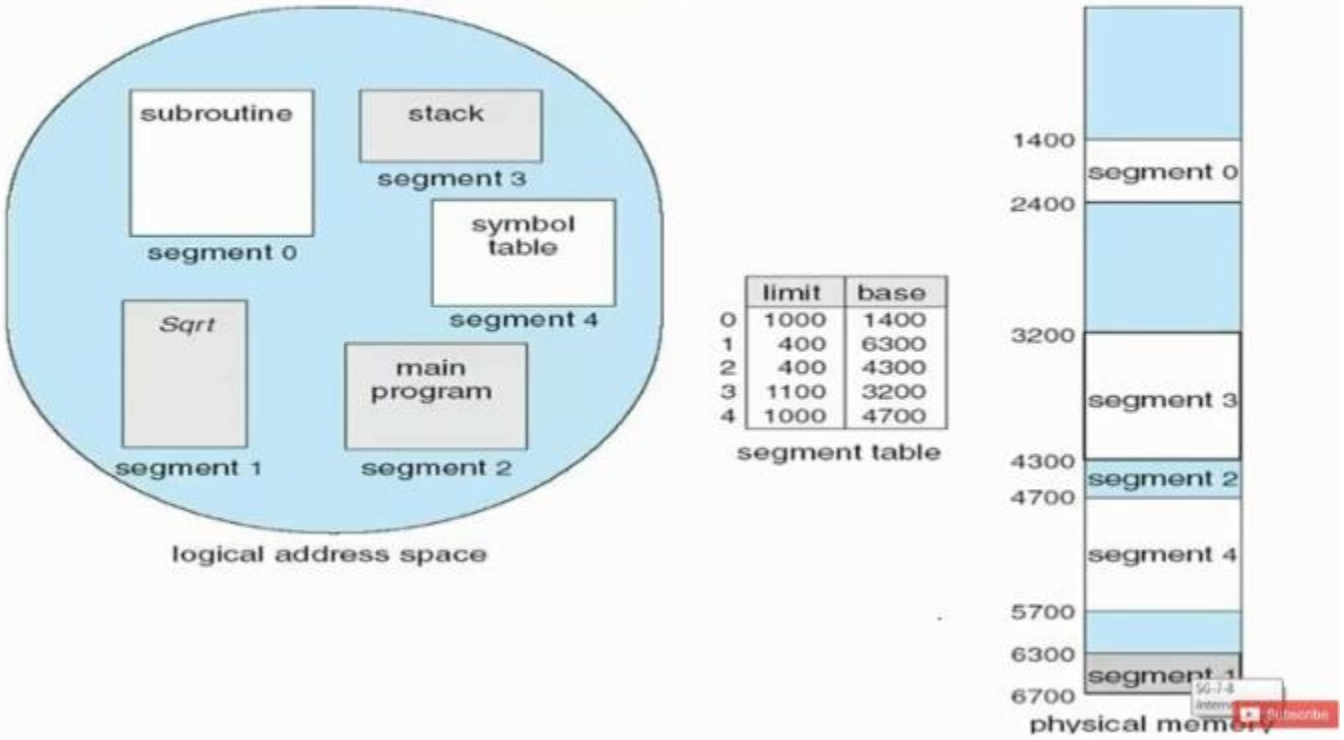
Variable Partition Problem

- Segments are variable-sized
  - Dynamic memory allocation required (first fit, best fit, worst fit).
- External fragmentation
  - In the worst case the largest hole may not be large enough to fit in a new segment. Note that paging has no external fragmentation problem.
- Each process has its own segment table
  - like with paging where each process has its own page table. The size of the segment table is determined by the number of segments, whereas the size of the page table depends on the total amount of memory occupied.
- Segment table located in main memory
  - as is the page table with paging
- Segment table base register (STBR)
  - Points to current segment table in memory
- Segment table length register (STLR)
  - indicates number of segments

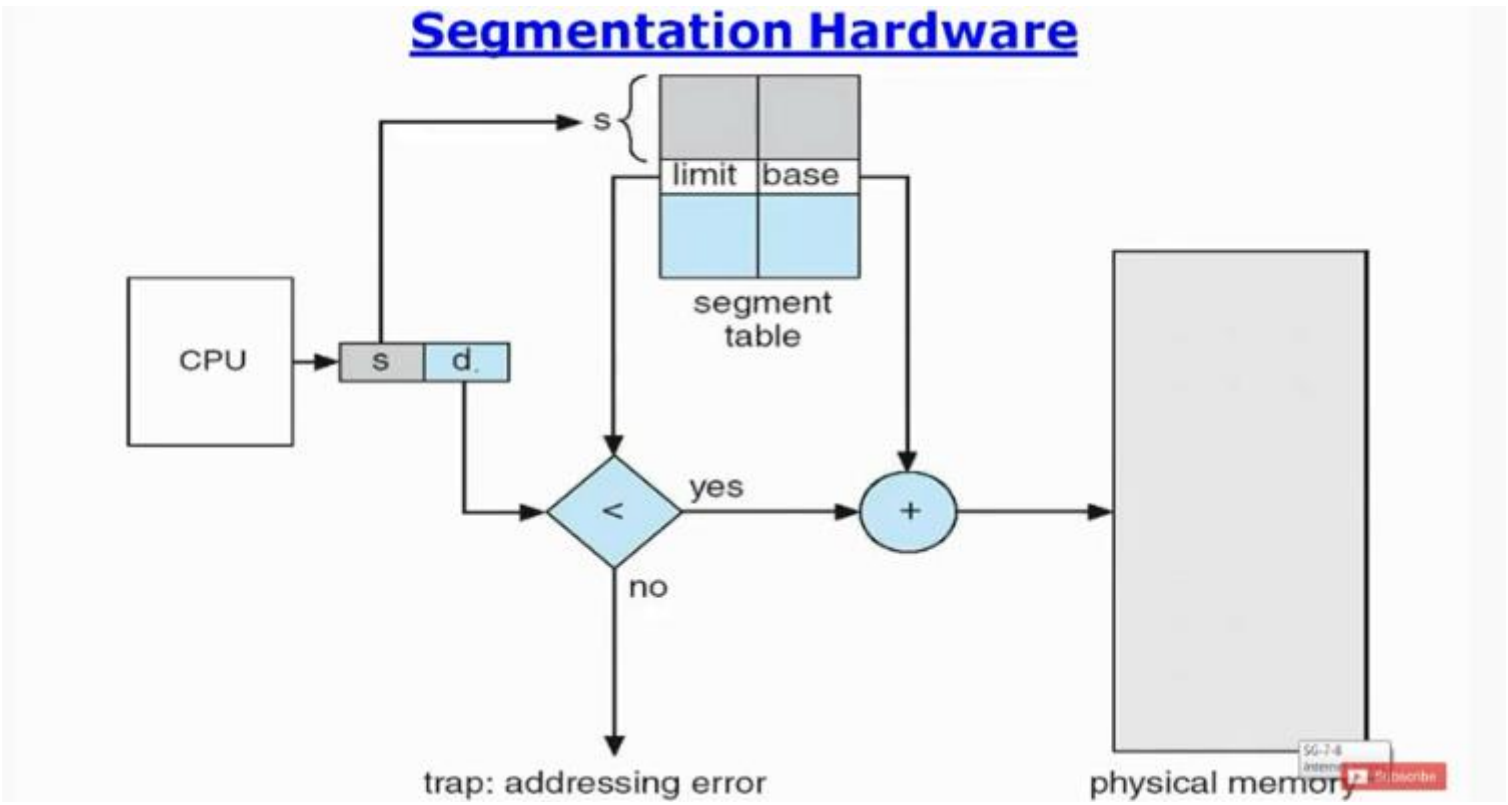
Logical View of Segmentation



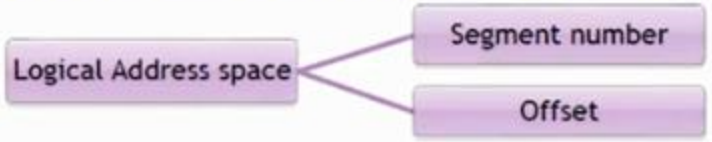
Example of Segmentation



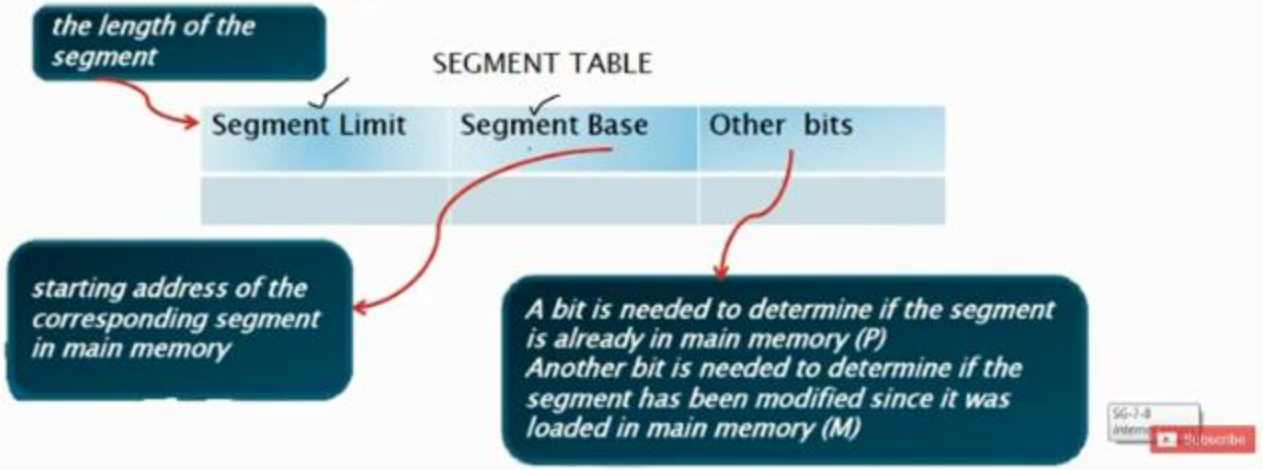
Segmentation Hardware



Logical Addressing in Segmentation



- The mapping of the logical address to the physical address is done with the help of the segment table.



Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes
- Two processes that need to share access to a single segment would have the same segment name and address in their segment tables.

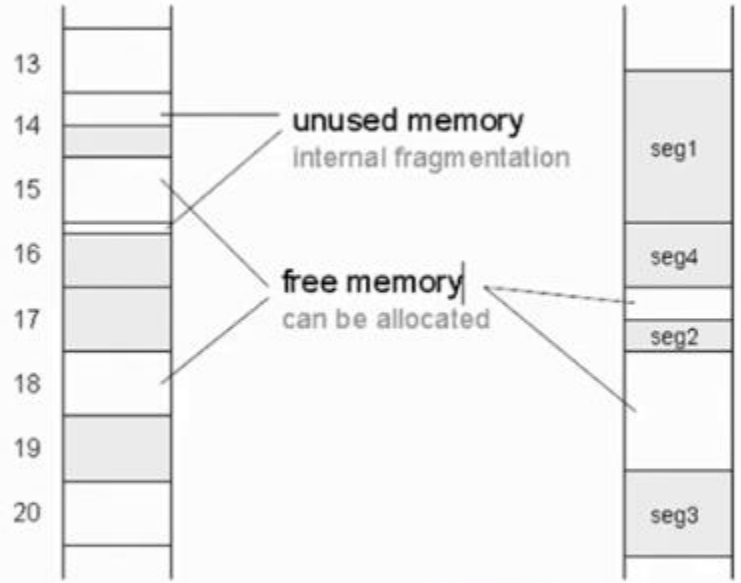
Advantages

- No internal fragmentation
- Segment tables consume less memory than page tables (only one entry per actual segment as opposed to one entry per page in Paging method)
- Because of the small segment table, memory reference is easy
- Lends itself to sharing data among processes.
- Lends itself to protection.
- As the individual lines of a page do not form one logical unit, it is not possible to set a particular access right to a page.
- Note that each segment could be set up an access righ

Disadvantages

- External fragmentation.
- Costly memory management algorithm
- Unequal size of segments is not good in the case of swapping.

Paging vs Segmentation



Paging is based on fixed-size units of memory (frames)

Segmentation is based on variable-size units of memory (segments)

Paging vs Segmentation

- With paging physical memory is divided into fixed size frames. When memory space is needed, as many free frames are occupied as necessary. These frames can be located anywhere in memory, the user process always sees a logical contiguous address space
- With segmentation the memory is not systematically divided. When a program needs k segments (usually these have different sizes), the OS tries to place these segments in the available memory holes. The segments can be scattered around memory. The user process does not see a contiguous address space, but sees a collection of segments (of course each individual segment is contiguous as is each page or frame).

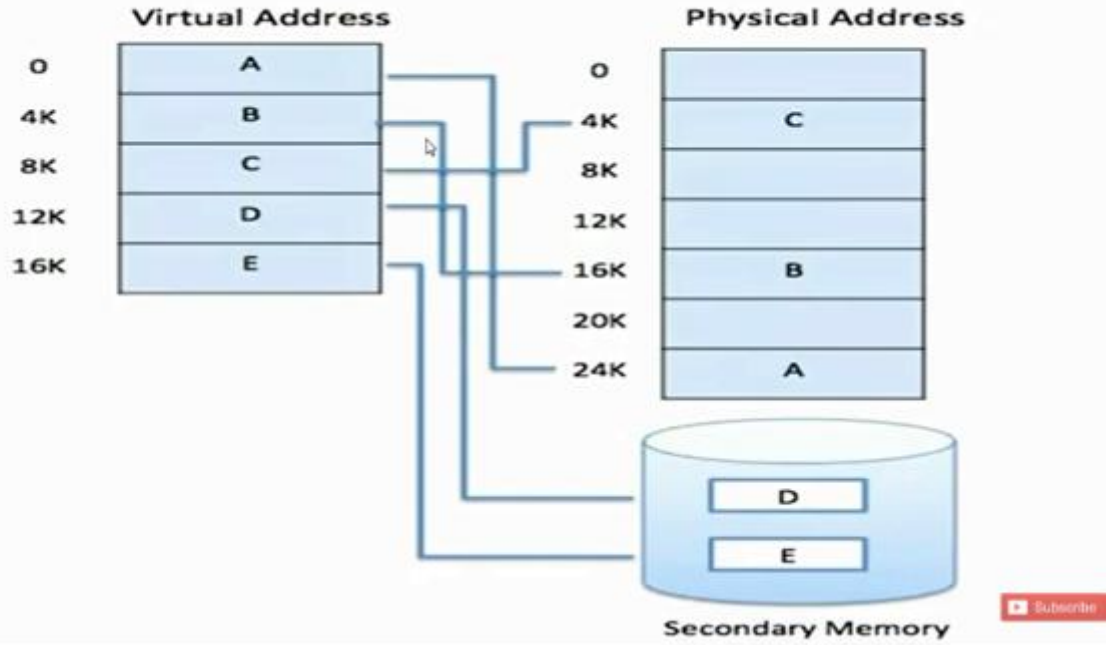
Paging vs Segmentation

Paging	Segmentation
Each process is assigned its page table.	Each process is assigned a segment table
Page table size proportional to allocated Memory	Segment table size proportional to number of segments
Often large page tables and/or multi-level Paging	Usually small segment Tables
Internal fragmentation	External fragmentation.
Free memory is quickly allocated to a process	Lengthy search times when allocating memory to a proc

Virtual Memory: Introduction

- Virtual memory is a common part of operating system on desktop computers.
- **Virtual Memory** is a storage scheme that provides user an illusion of having a very big main memory.
- The term virtual memory refers to something which appears to be present but actually it is not.
- The virtual memory technique allows users to use more memory for a program than the real memory of a computer.
- The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

Virtual Memory: Example



NEED OF VIRTUAL MEMORY

- Virtual memory is a imaginary memory which we are assuming. If we have a material that exceed your memory at that time we need to use the concept of virtual memory.
- virtual memory is temporary memory which is used along with the ram of the system.

Importance of Virtual Memory

- When your computer runs out of Physical Memory (**RAM**) it writes what it needs to remember to the hard disc in a swap file as Virtual Memory.

- If a computer running Windows requires more memory/RAM then there is installed in the system to run a program, etc, it uses a small section of the hard drive for this purpose

Advantages of Virtual Memory

- Allows processes whose aggregate memory requirement is greater than the amount of physical memory, as infrequently used pages can reside on the disk.
- Virtual memory allows speed gain when only a particular segment of the program is required for the execution of the program.
- This concept is very helpful in implementing multiprogramming environment.

Disadvantages of Virtual Memory

- Applications run slower if the system is using virtual memory.
- It Takes more time to switch between applications.
- Less hard drive space for your use.
- It reduces system stability.

Virtual Memory: Implementation

- Virtual memory is commonly implemented by demand paging.
- It can also be implemented in a segmentation system.
- Demand segmentation can also be used to provide virtual memory.

Address Space and Memory Space

- Virtual Memory is the address used by the programmer and the set of such addresses is called ***Address Space***.
- An address in Main Memory (**RAM**) is called a Physical Address. The set of such locations in main memory is called the Memory Space.
- The memory space consist of the actual main memory locations directly addressable for processing.

Demand Paging

- A **Demand Paging** system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.
- When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory
- Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

Virtual Memory: Page Fault

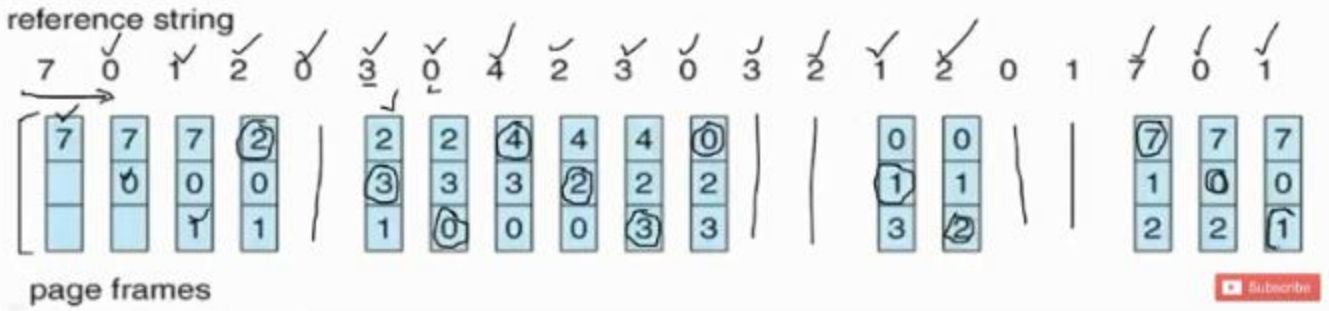
- While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a ***page fault*** and transfers control from the program to the operating system to demand the page back into the memory.

Page Replacement Algorithm

- In a computer operating system that uses paging for virtual memory management, *Page Replacement algorithm* decide which memory pages to page out. When a page of memory need to be allocated.
- Paging happens when a ***page fault*** occurs.
- When a program starts execution, one or more pages are transferred into main memory.
  - FIFO (first in first out)
  - LRU (Least Recently used)
  - OPT (Optimal)

Page Replacement: FIFO

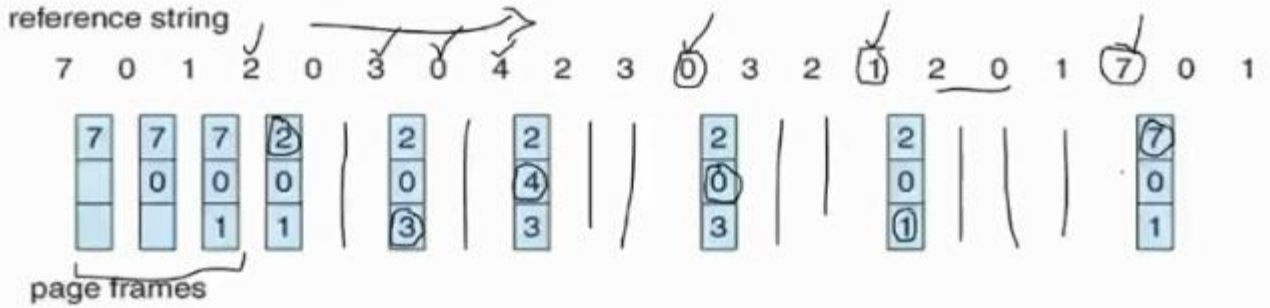
- First In First Out (FIFO) is very easy to implement
- The FIFO algorithm select the page for replacement that has been in memory the longest time
- The page to be removed is easily determined because its identification number is at the top of the FIFO stack.
- It has the disadvantage that under certain circumstances pages are removed and loaded from memory to frequently.



15 page fault

Page Replacement: Optimal

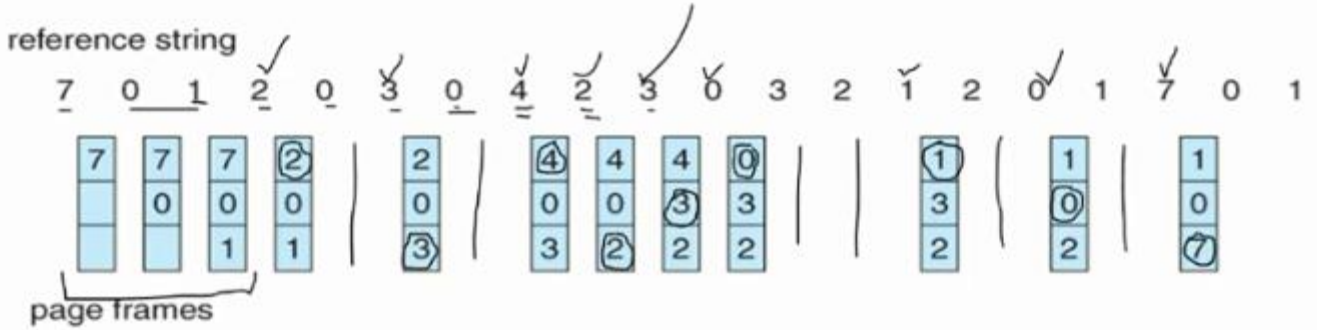
- Replace the page that will not be used for the longest time in the future.



9 page fault

Page Replacement: LRU

- ***Least Recently Used (LRU),*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future.



12 page fault