Operating System
KCS – 401

Process

Dr. Pankaj Kumar
Associate Professor – CSE
SRMGPC Lucknow

Outline of the Lecture

Process Concept

Process in memory

Process state

Process Transition Diagram,

Process Control Block (PCB)

Process Concept

An operating system executes a variety of programs:

- Batch system – **jobs**
- Time-shared systems – **user programs** or **tasks**

Textbook uses the terms ***job*** and ***process*** almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

Multiple parts

Program is ***passive*** entity stored on disk (**executable file**), process is ***active***

- Program becomes process when executable file loaded into memory

Execution of program started via GUI mouse clicks, command line entry of its name, etc

One program can be several processes

- Consider multiple users executing the same program

Process in Memory

Multi Parts of OS

The program code, also called **text section**

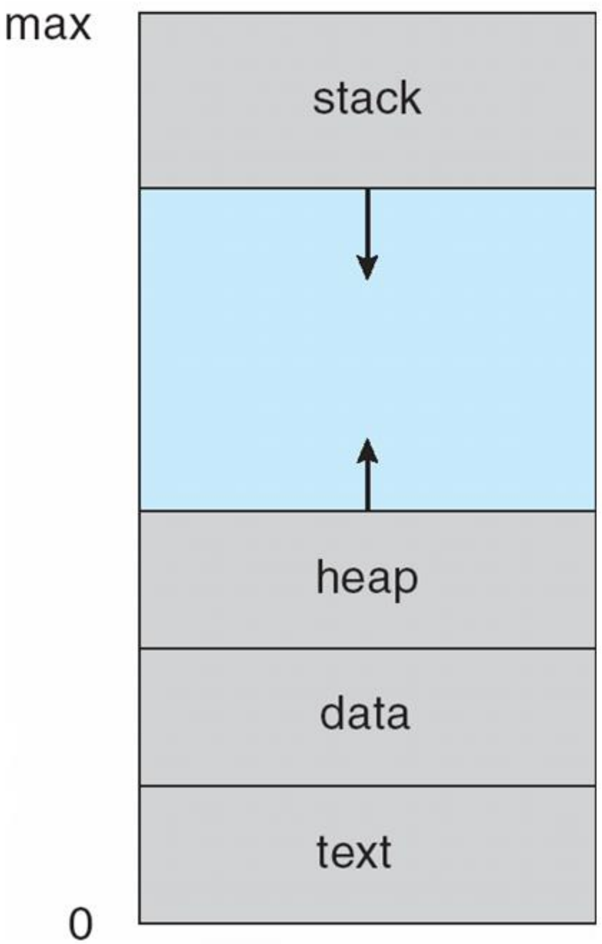
Current activity including **program counter**, processor registers

Stack containing temporary data

Function parameters, return addresses, local variables

Data section containing global variables

Heap containing memory dynamically allocated during run time

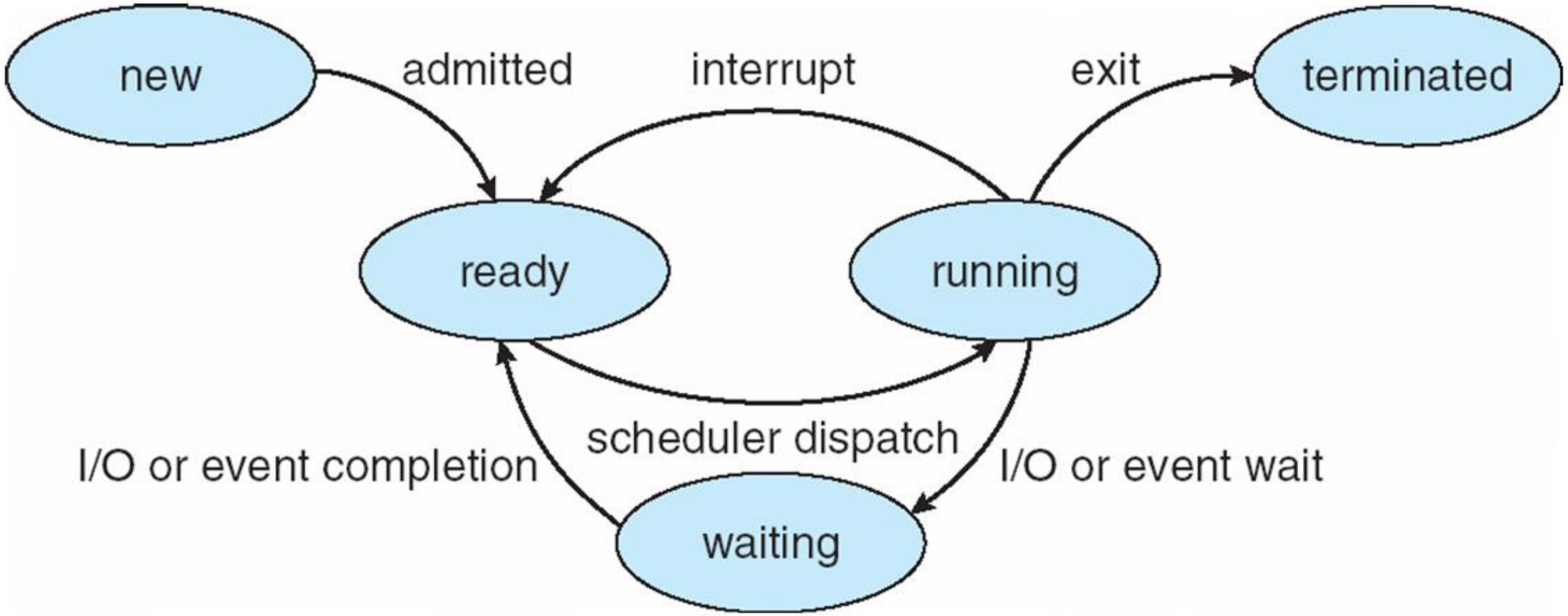


Process State

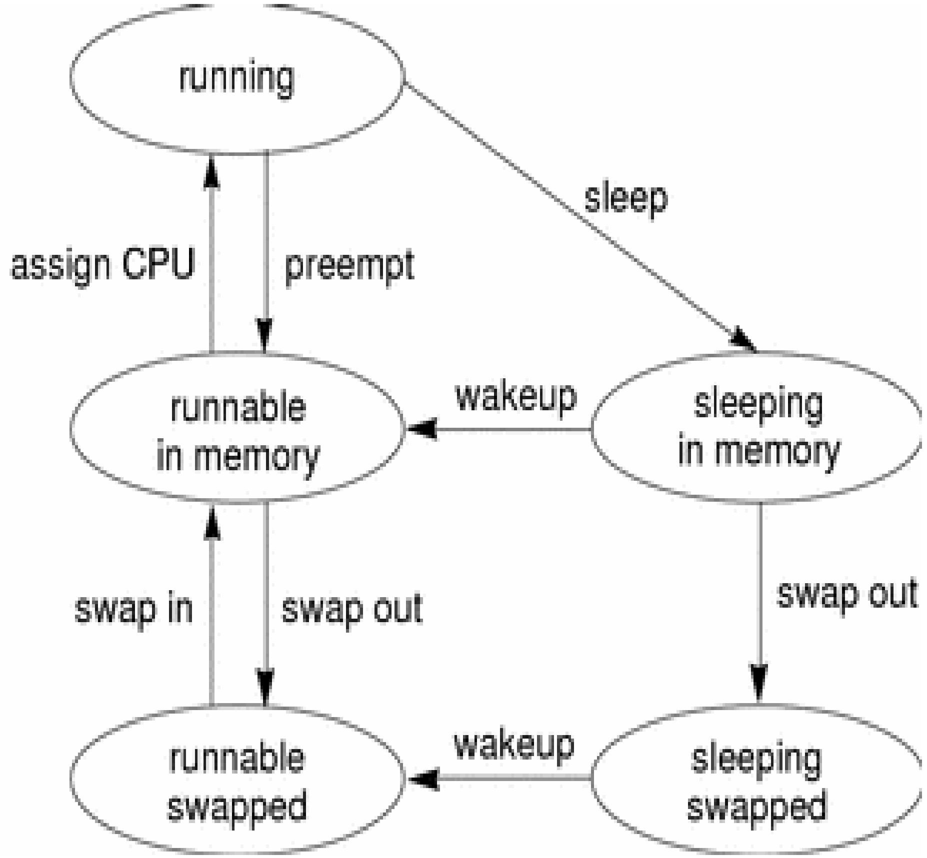
As a process executes, it changes **state**

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Process State



Process Transition Diagram



Process Control Block

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Process Id: A process id is a unique identity of a process. Each process is identified with the help of the process id.

Process State: A process can be in any state out of the possible states of a process. So, the CPU needs to know about the current state of a process, so that its execution can be done easily.

Process-Id
Process state
Process Priority
Accounting Information
Program Counter
CPU Register
PCB Pointers
....

Process Control Block

Process Control Block

Priority There is a priority associated with each process. Based on that priority the CPU finds which process is to be executed first. Higher priority process will be executed first.

Process Accounting Information This field of PCB gives the account/description of the resources used by that process. Like, the amount of CPU time, real-time used, connect time.

Program Counter The program counter is the pointer to an instruction in the program or code that is to be executed next. This field contains the address of the instruction that will be executed next in the process.

CPU Registers Whenever an interrupt occurs and there is a context switch between the processes, the temporary information is stored in the registers. So, that when the process resumes the execution it correctly gains from where it leaves. CPU registers are used to hold those temporary values or information.

Process-Id
Process state
Process Priority
Accounting Information
Program Counter
CPU Register
PCB Pointers
....

Process Control Block

Process Control Block

PCB Pointer In this field, the pointer has an address of the next PCB, whose process state is **ready**. In this way, the operating system maintains the hierarchy of all the processes so that a parent process could locate all the child processes it creates easily.

List of Open Files These are the different files that are associated with the process

I/O Status Information This information includes the list of I/O devices used by the process, the list of files etc.

Event Information This field contains the information of the event for which the certain process is in **block** state. Whenever that event occurs the operating system identifies the process awaiting for this event using this field. If the event occurred match with this field the process changes its state from blocked to ready.

Process-Id
Process state
Process Priority
Accounting Information
Program Counter
CPU Register
PCB Pointers
....

Process Control Block

Process Address Space

Address space is a space in computer memory. Every process has an address space.

Address Space can be of two types

1. Physical Address Space

2. Virtual Address Space

Process Address Space means a space that is allocated in memory for a process. The **process address space** is the set of logical **addresses** that a **process** references in its code.

Process identification information

A PID (i.e., *process identification number*) is an identification number that is automatically assigned to each *process*.

A process is an *executing* (i.e., running) instance of a program. Each process is guaranteed a unique PID, which is always a non-negative integer.

The process *init* is the only process that will always have the same PID on any session and on any system, and that PID is 1. This is because init is always the first process on the system and is the ancestor of all other processes.

A very large PID does not necessarily mean that there are anywhere near that many processes on a system. This is because such numbers are often a result of the fact that PIDs are not immediately reused, in order to prevent possible errors.

Basic Concept

Maximum CPU utilization obtained with multiprogramming

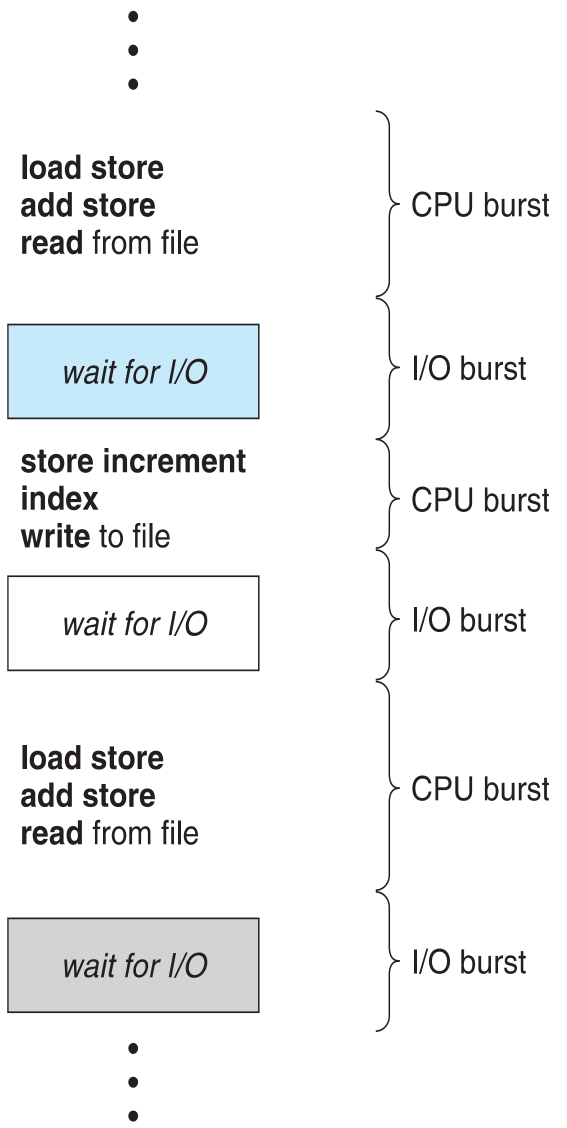
CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait

CPU burst followed by **I/O burst**

CPU burst distribution is of main concern

Burst Time/Execution Time: It is a time required by the process to complete execution. It is also called running time.

Arrival Time: when a process enters in a ready state.



CPU Scheduling

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold.

The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution.

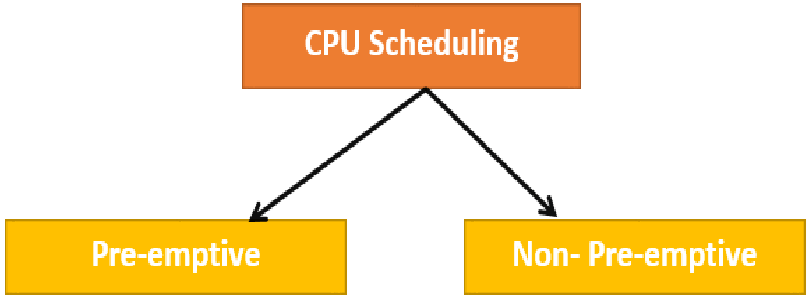
The selection process will be carried out by the ***CPU scheduler***. It selects one of the processes in memory that are ready for execution.

CPU Scheduling

Types of CPU Scheduling

Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.



Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

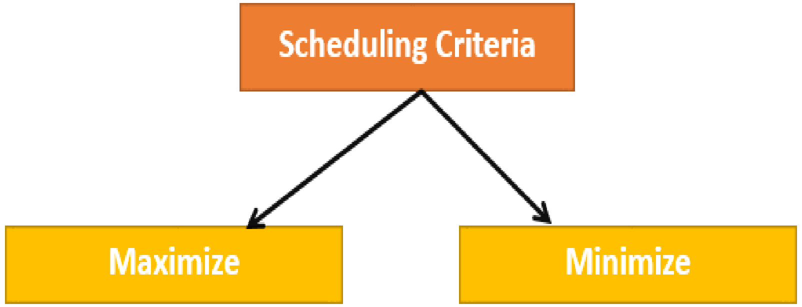
- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

Dispatch latency – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

CPU utilization: CPU utilization is the main task in which the operating system needs to make sure that CPU remains as busy as possible. It can range from 0 to 100 percent.

Throughput: The number of processes that finish their execution per unit time is known Throughput. So, when the CPU is busy executing the process, at that time, work is being done, and the work completed per unit time is called Throughput.



Maximize:
CPU Utilization
Throughput

Minimize:
Turnaround time
Waiting time
Response time

Waiting time: Waiting time is an amount that specific process needs to wait in the ready queue.

Response time: It is an amount to time in which the request was submitted until the first response is produced.

Turnaround Time: Turnaround time is an amount of time to execute a specific process. It is the calculation of the total time spent waiting to get into the memory, waiting in the queue and, executing on the CPU. The period between the time of process submission to the completion time is the turnaround time.

Scheduling Criteria

Completion Time: Time at which process completes its execution.

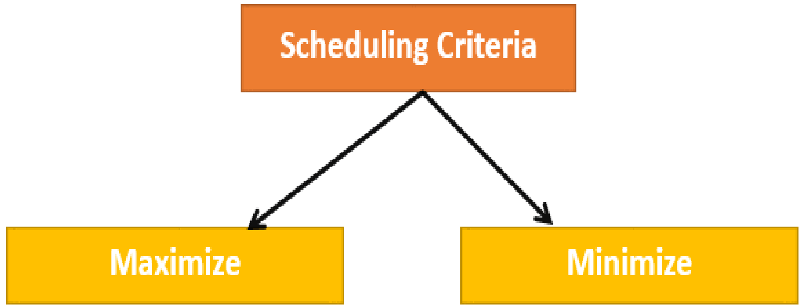
Turn Around Time: Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time

Waiting time = Start time - Arrival time



Maximize:
CPU Utilization
Throughput

Minimize:
Turnaround time
Waiting time
Response time

Scheduling Type

Types of CPU scheduling Algorithm

There are mainly six types of process scheduling algorithms

1. First Come First Serve (FCFS)
2. Shortest-Job-First (SJF) Scheduling
3. Shortest Remaining Time
4. Priority Scheduling
5. Round Robin Scheduling
6. Multilevel Queue Scheduling

First Come First Serve (FCFS)

It is the easiest and most simple CPU scheduling algorithm. In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

Characteristics of FCFS method:

- It offers non-preemptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis
- It is easy to implement and use.
- However, this method is poor in performance, and the general wait time is quite high.

First Come First Serve (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3

The Gantt Chart for the schedule is:

Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

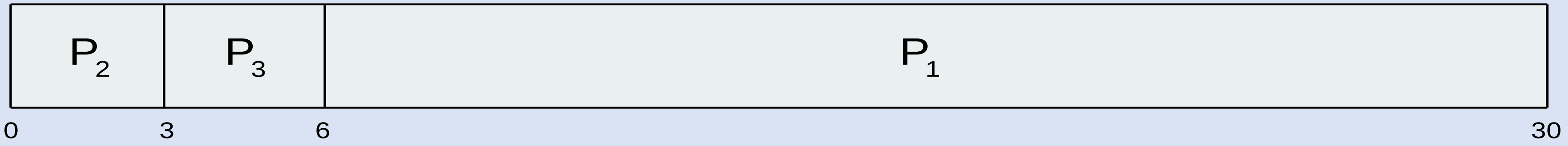
Average waiting time: $(0 + 24 + 27)/3 = 17$

First Come First Serve (FCFS)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

The Gantt chart for the schedule is:



Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$

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

Much better than previous case

Convoy effect - short process behind long process

Consider one CPU-bound and many I/O-bound processes

First Come First Serve (FCFS)

P4	P3	P1	P5	P2

0 3 11 17 21 23

Process	Burst time	Arrival time
P1	6	2
P2	3	5
P3	8	1
P4	3	0
P5	4	4

Waiting time for P_1 = 11-2 = 9

Waiting time for P_2 = 21-5 = 16

Waiting time for P_3 = 3-1 = 2

Waiting time for P_4 = 0

Waiting time for P_5 = 17-4 = 13

Average waiting time: (9 + 16 + 2+0+13)/5 = 40/5 =8

Turn Around Time = P_1 = 17-2 = 15

Turn Around Time = P_2 = 23-5 = 18

Turn Around Time = P_3 = 11-1 = 10

Turn Around Time = P_4 = 3-0 = 3

Turn Around Time = P_5 = 21-4 = 17

Average Turn Around Time : (15 + 18 + 10+3+17)/5 = 63/5 =12.6

First Come First Serve (FCFS)

0	2	3	7	10	13	14

P3		P1	P5	P2	P4

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

Waiting time for P_1 = 3 – 3 = 0

Waiting time for P_2 = 10-5 = 5

Waiting time for P_3 = 0

Waiting time for P_4 = 13-5 = 8

Waiting time for P_5 = 7-4 = 3

Average waiting time: (0 + 5 + 0 + 8 + 3)/5 = 16/5 =3.2

Turn Around Time = P_1 = 7-3 = 4

Turn Around Time = P_2 = 13-5 = 8

Turn Around Time = P_3 = 2-0 = 2

Turn Around Time = P_4 = 14-5 = 9

Turn Around Time = P_5 = 10-4 = 6

Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8

Shortest Job First Serve (SJFS)

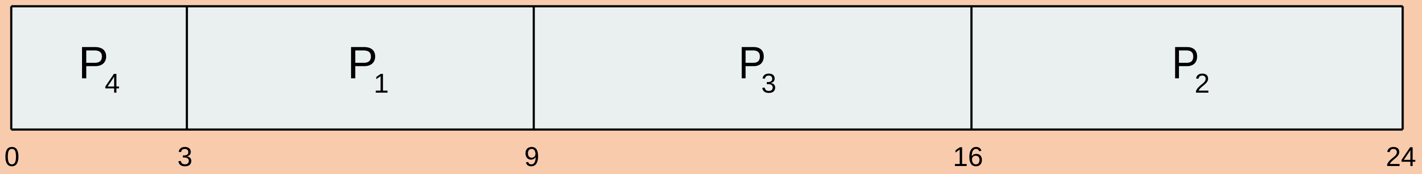
1. It is a Greedy Algorithm.
2. Sort all the process according to the arrival time.
3. Then select that process which has minimum arrival time and minimum Burst time.
4. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Shortest Job First Serve (SJFS)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

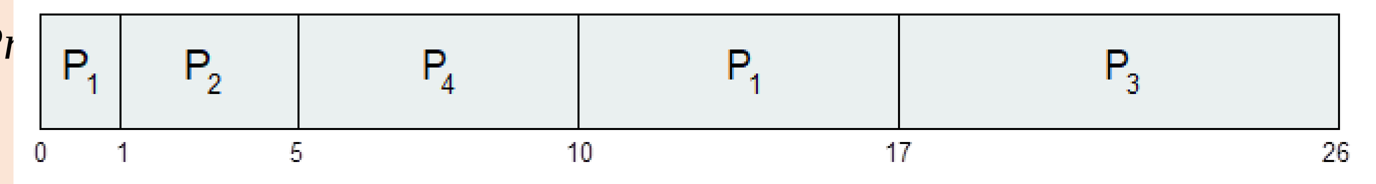
Shortest Job First Serve (SJFS)

Preemptive mode of Shortest Job First is called as **Shortest Remaining Time First (SRTF)**.

- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Pr*

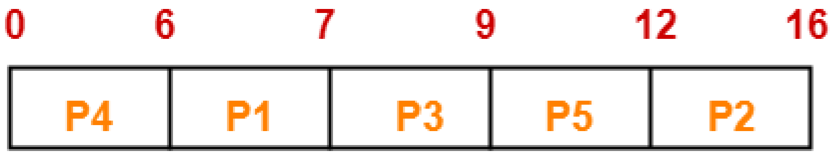


- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Shortest Job First Serve (SJFS)

set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3



Gantt Chart

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Shortest Job First Serve (SJFS)

Advantages-

- SJFS is optimal and guarantees the minimum average waiting time.
- It provides a standard for other algorithms since no other algorithm performs better than it.

Disadvantages-

- It can not be implemented practically since burst time of the processes can not be known in advance.
- It leads to starvation for processes with larger burst time.
- Priorities can not be set for the processes.
- Processes with larger burst time have poor response time.

Priority Scheduling

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

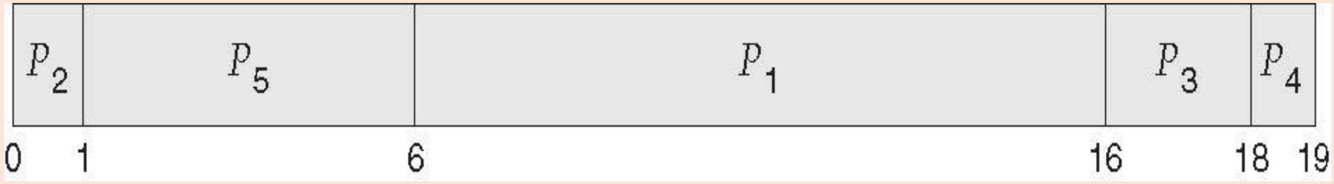
Characteristics of Priority Scheduling

- It used in Operating systems for performing batch processes.
- If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.
- In priority scheduling, a number is assigned to each process that indicates its priority level.
- Lower the number, higher is the priority.
- In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

Priority Scheduling

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

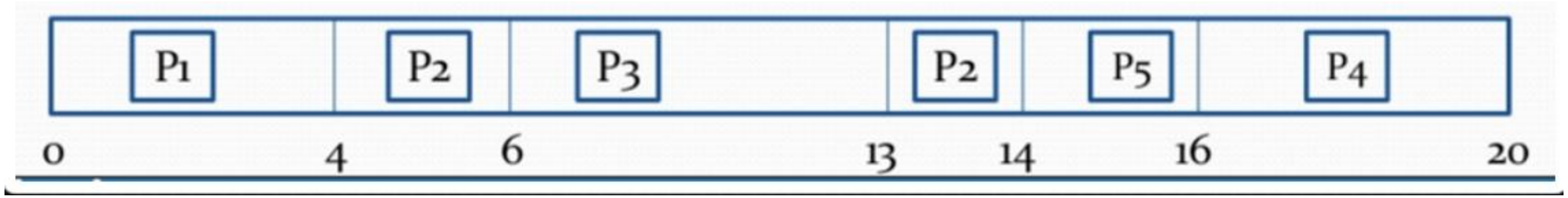
Priority scheduling Gantt Chart



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

Priority Scheduling

Process	Priority	Burst time	Arrival time
P1	1	4	0
P2	2	3	0
P3	1	7	6
P4	3	4	11
P5	2	2	12



Average Waiting time = (0+11+0+5+2)/5 = 18/5= 3.6

Priority Scheduling

Advantages

Processes are executed on the basis of priority so high priority does not need to wait for long which saves time

Suitable for applications with fluctuating time and resource requirements.

Disadvantages of priority scheduling

Starvation – low priority processes may never execute

Solution ≡ **Aging** – as time progresses increase the priority of the process

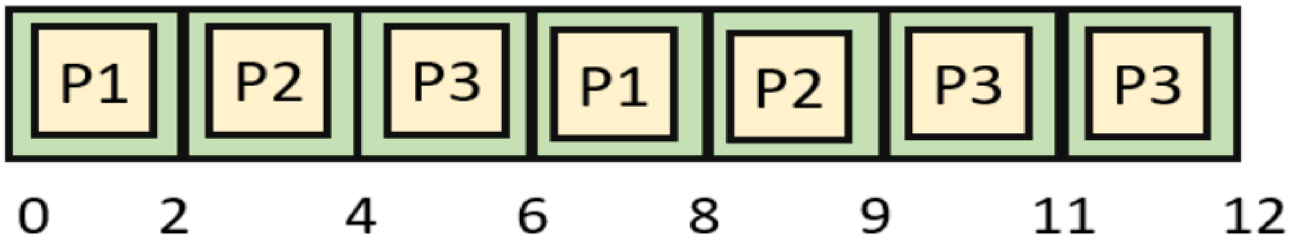
Round-Robin Scheduling

- Round robin is a pre-emptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time **quantum/time slice.**
- The process that is preempted is added to the end of the queue.
- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
- It is a real time algorithm which responds to the event within a specific time limit.
- Round robin is one of the oldest, fairest, and easiest algorithm.
- Widely used scheduling method in traditional OS.

Round-Robin Scheduling

quantum/time slice = 2

Process Queue	Burst time
P1	4
P2	3
P3	5



Waiting Time
P1= 8-4= 4
P2= 9-3= 6
P3= 12-5= 7

Turnaround Time
P1= 8
P2= 9
P3= 12

Average Waiting time = (4+6+7)/3 = 5.66

Round-Robin Scheduling

P1-P2-P3-P4-P5-P1-P6-P2-P5

P1	P2	P3	P4	P5	P1	P6	P2	P5	
0	4	8	11	12	16	17	21	23	24

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

Process ID	Completion Time	Turn Around Time	Waiting Time
1	17	17	12
2	23	22	16
3	11	9	6
4	12	9	8
5	24	20	15
6	21	15	11

Avg Waiting Time = (12+16+6+8+15+11)/6 = 76/6 units

Round-Robin Scheduling

Disadvantages

- If slicing time of OS is low, the processor output will be reduced.
- This method spends more time on context switching
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.
- Round-robin scheduling doesn't give special priority to more important tasks.
- Lower time quantum results in higher the context switching overhead in the system.
- Finding a correct time quantum is a quite difficult task in this system.

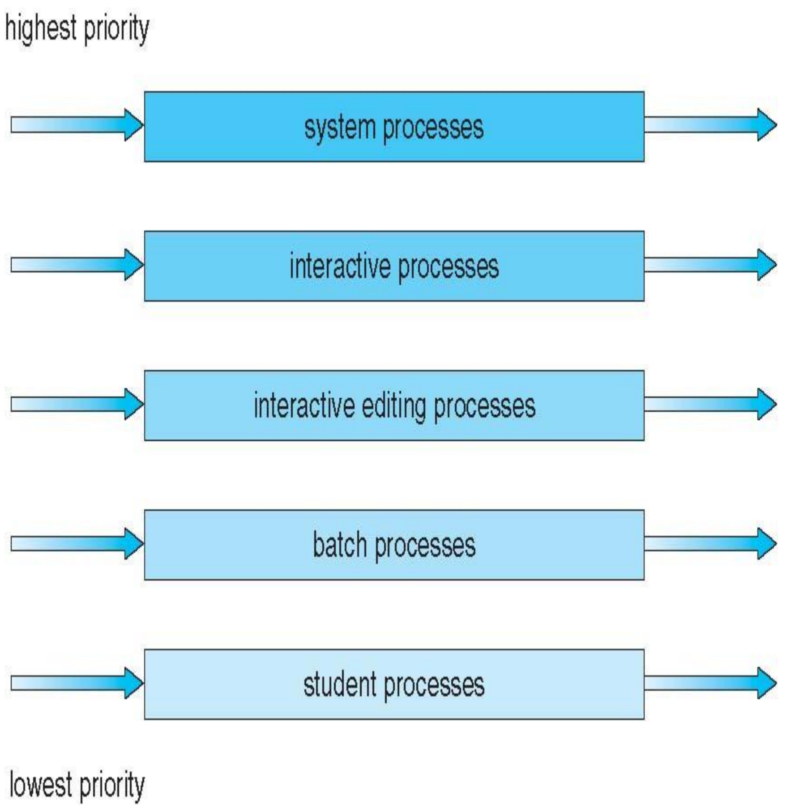
Multilevel Queue Scheduling

Multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

A common division is made between foreground (or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

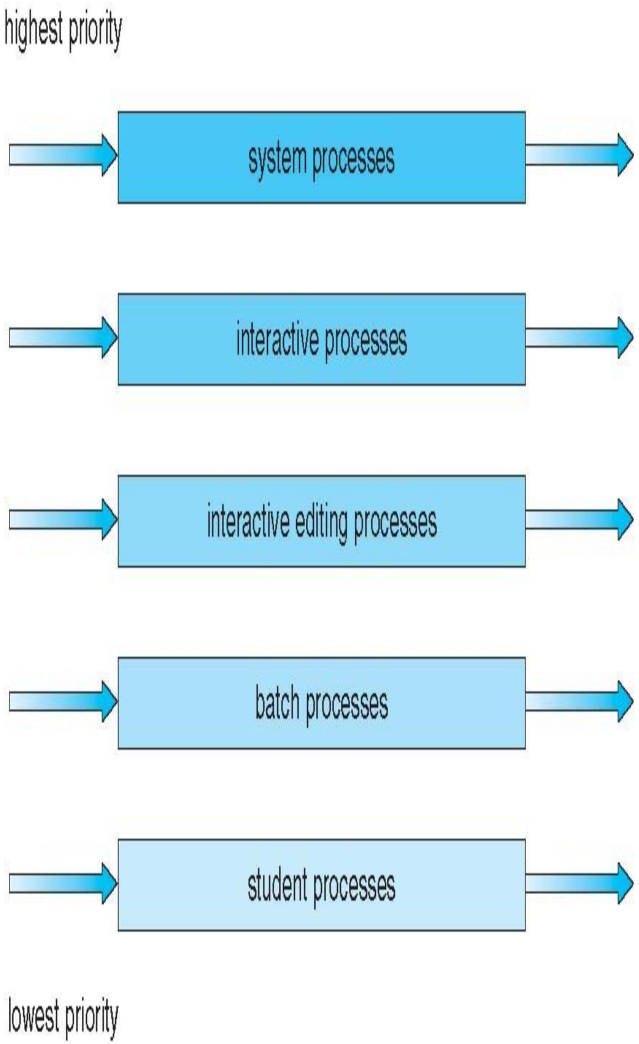
Multilevel queue scheduling has the following characteristics:

- Each queue has its own scheduling algorithm
- Processes are divided into different queues based on their process type, memory size and process priority.



Multilevel Queue Scheduling

- **System Process** The Operating system itself has its own process to run and is termed as System Process.
- **Interactive Process** The Interactive Process is a process in which there should be the same kind of interaction (e.g. online game).
- **Batch Processes** Batch processing is basically a technique in the Operating system that collects the programs and data together in the form of the **batch** before the **processing** starts.
- **Student Process** The system process always gets the highest priority while the student processes always get the lowest priority.



For System Processes: First Come First Serve(FCFS) Scheduling.

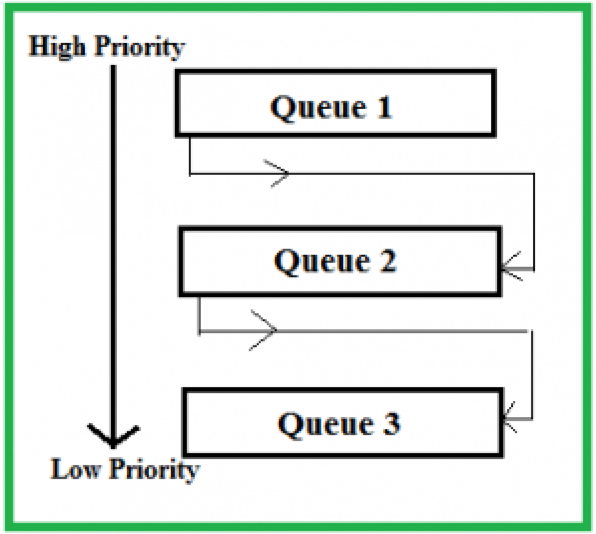
For Interactive Processes: Shortest Job First (SJF) Scheduling.

For Batch Processes: Round Robin(RR) Scheduling

For Student Processes: Priority Scheduling

Multilevel Feedback Queue Scheduling

This Scheduling is like Multilevel Queue (MLQ) Scheduling but in this process can move between the queues. **Multilevel Feedback Queue Scheduling (MLFQ)** keep analyzing the behavior (time of execution) of processes and according to which it changes its priority.

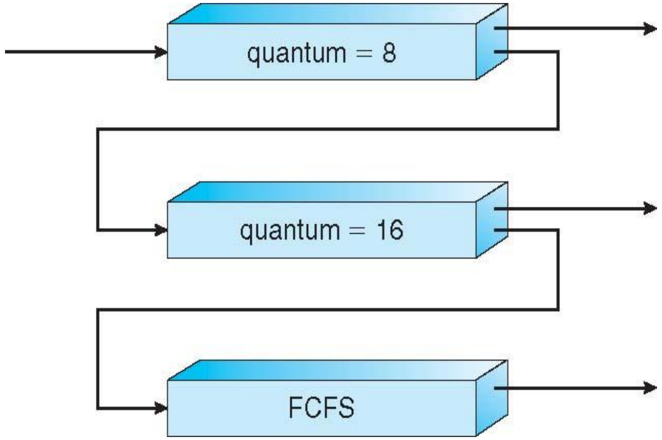


Three queues:

- Q_1 – RR with time quantum 8 milliseconds
- Q_2 – RR time quantum 16 milliseconds
- Q_3 – FCFS

Scheduling

- A new job enters queue Q_0 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 Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2



Multiprocessor Scheduling

Multiple processor scheduling or multiprocessor scheduling focuses on designing the scheduling function for the system which is consist of 'more than one processor'. With multiple processors in the system, the load sharing becomes feasible but it makes scheduling more complex.

CPU scheduling more complex when multiple CPUs are available

Homogeneous processors within a multiprocessor

Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing

Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

Multiprocessor Scheduling

Processor affinity – process has affinity for processor on which it is currently running

1. **Soft Affinity** – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity** – Hard Affinity allows a process to specify a subset of processors on which it may run.

If SMP, need to keep all CPUs loaded for efficiency

Load balancing attempts to keep workload evenly distributed

Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

Pull migration – idle processors pulls waiting task from busy processor

System Model

System consists of finite no of resources to be distributed among a number of competing process.

Resources are partitioned into several $R_1, R_2, . . ., R_m$ consisting number of identical instances.

CPU cycles, memory space, I/O devices (such as printers…)

Each resource type R_i has W_i instances.

Each process utilizes a resource as follows:

request

use

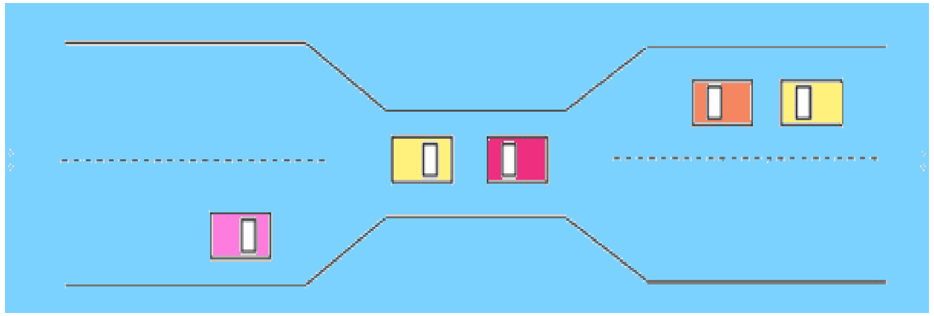
release

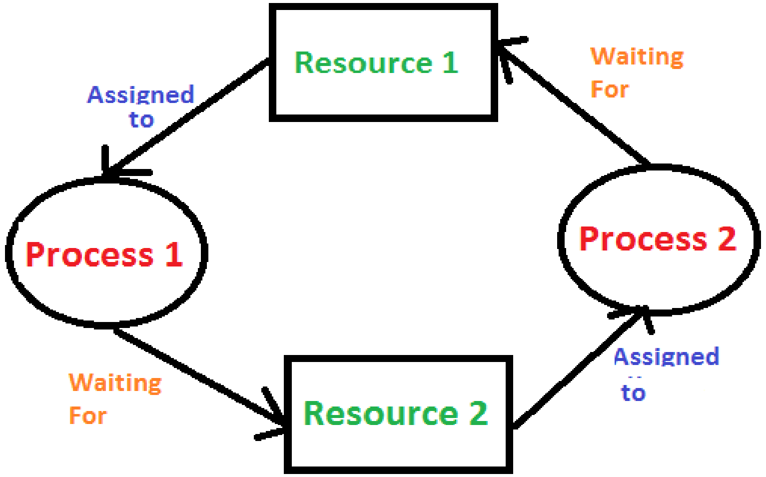
Basic Concept - Deadlock

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

or

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

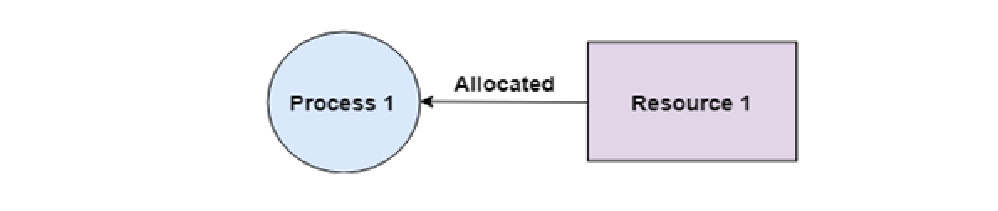




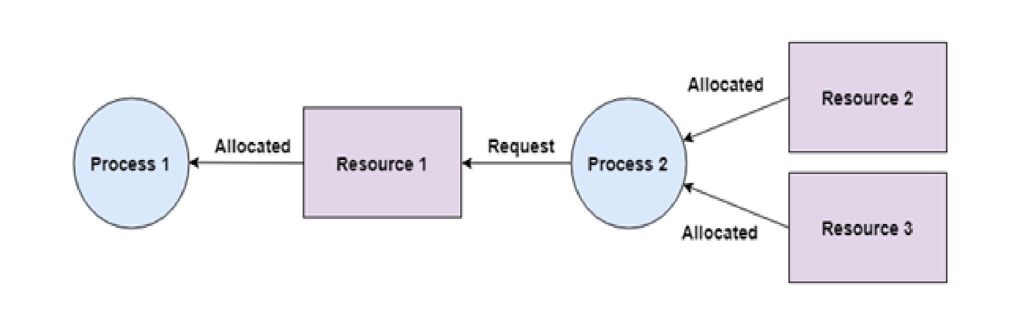
Deadlock Characterisation

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion: only one process at a time can use a resource. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.

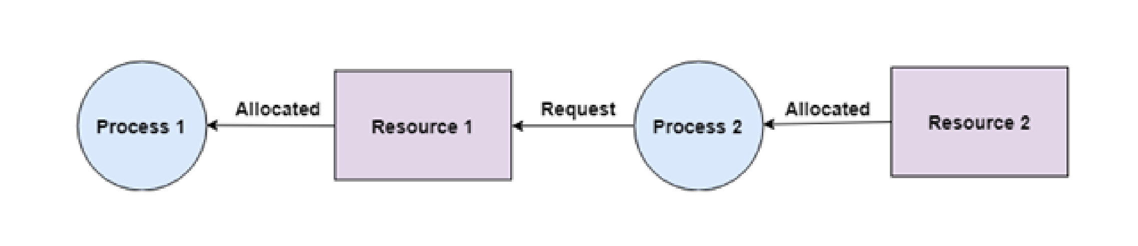


Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes. Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.

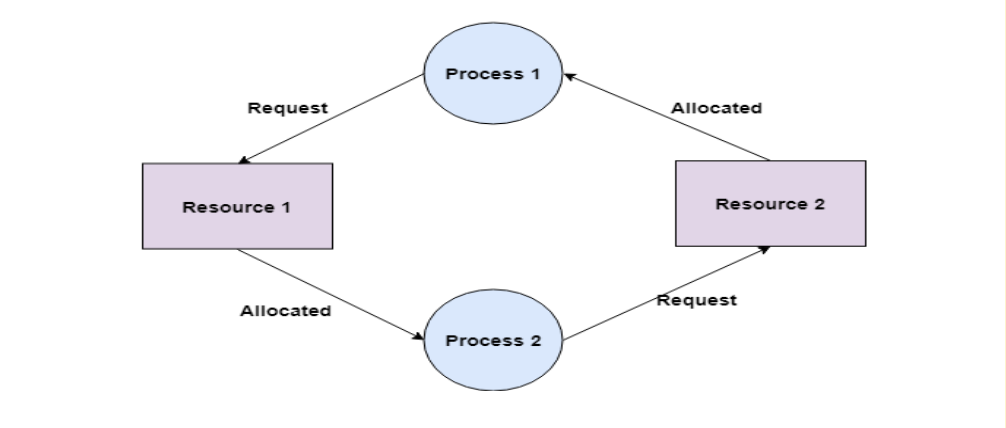


Deadlock Characterisation

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular wait: there exists a set {P_0, P_1, …, P_n} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



Deadlock Characterisation

Resource-Allocation Graph

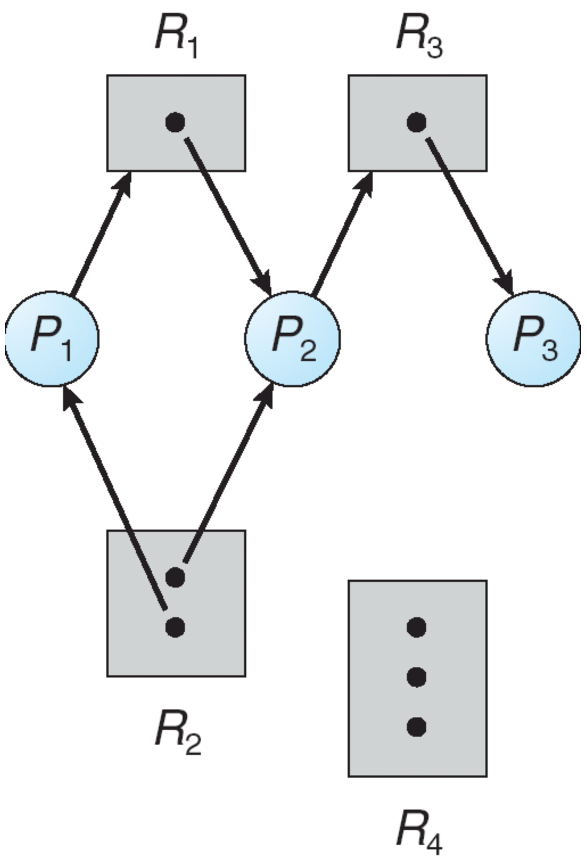
V is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

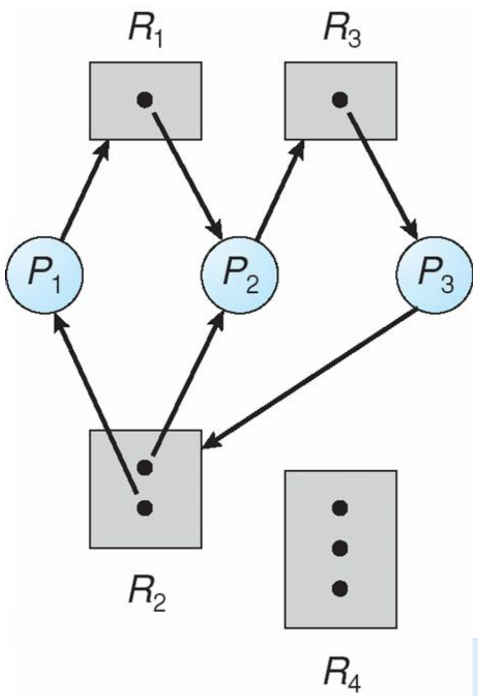
request edge – directed edge $P_i \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$

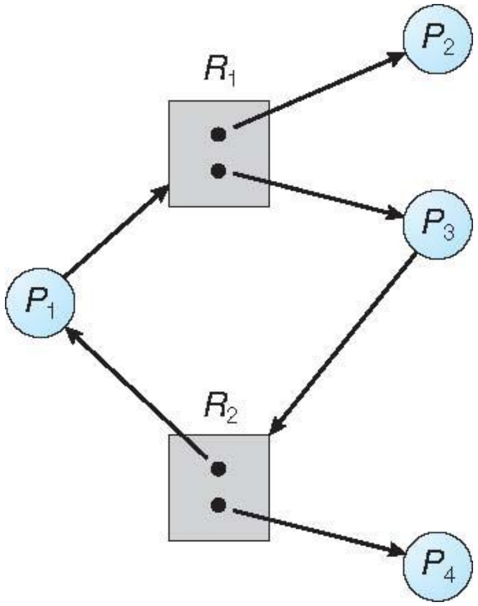


Deadlock Characterisation

Resource-Allocation Graph with Deadlock



Resource-Allocation Graph with no Deadlock



If graph contains no cycles $\Rightarrow$ no deadlock

If graph contains a cycle $\Rightarrow$

- if only one instance per resource type, then deadlock
- if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

To ensure that the system will ***never*** enter a deadlock state, the system can use:

Deadlock prevention

Deadlock avoidance

Allow the system to enter a deadlock state and then recover

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Deadlock Prevention is s a set of method for ensuring that at least one of the necessary condition can not hold.

Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
- Low resource utilization; starvation possible

Deadlock Prevention

No Preemption –

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

Preempted resources are added to the list of resources for which the process is waiting

Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

The deadlock Avoidance method is used by the operating system in order to check whether the system is in a ***safe state*** or in an ***unsafe state*** and in order to avoid the deadlocks, the process must need to tell the operating system about the maximum number of resources a process can request in order to complete its execution.

Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need

The ***deadlock-avoidance algorithm*** dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock Avoidance

Safe State

A state is safe if the system can allocate resources to each process(up to its maximum requirement) in some order and still avoid a deadlock. Formally, a system is in a safe state only, if there exists a ***safe sequence***.

System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

That is:

If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished

When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate

When P_i terminates, P_{i+1} can obtain its needed resources, and so on

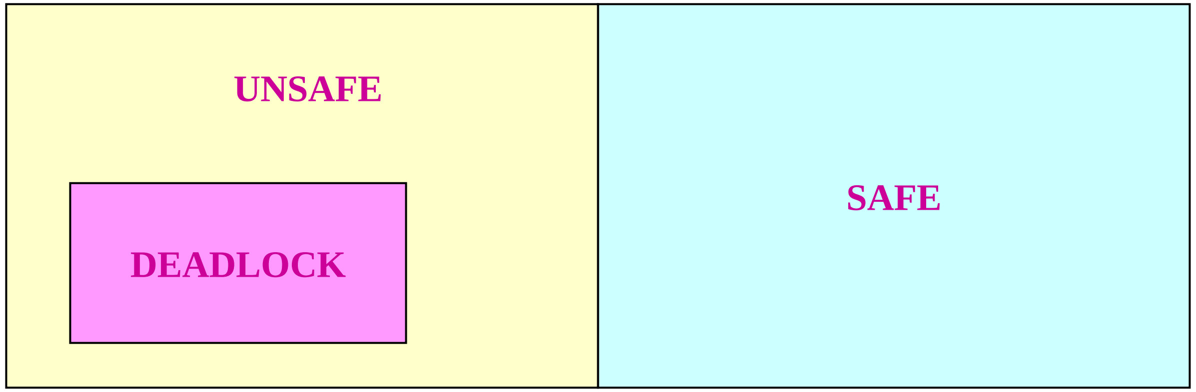
Deadlock Avoidance

In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.

If a system is in safe state $\Rightarrow$ no deadlocks

If a system is in unsafe state $\Rightarrow$ possibility of deadlock

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Only with luck will processes avoid deadlock.

O.S. can avoid deadlock.

Deadlock Avoidance

Let's assume a very simple model: each process declares its maximum needs. In this case, algorithms exist that will ensure that no unsafe state is reached.

EXAMPLE:

There exists a total of 12 tape drives. The current state looks like this:

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	9	2	7

So at time t0, the system is in a safe state. The sequence is <P2,P1,P3> satisfies the safety condition. Process P2 can immediately be allocated all its tape drives and then return them. After the return the system will have 5 available tapes, then process P1 can get all its tapes and return them (the system will then have 10 tapes); finally, process P3 can get all its tapes and return them (The system will then have 12 available tapes).

Suppose p2 requests and is given one more tape drive. What happens then???????????

Deadlock Avoidance

Banker's Algorithm

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

The bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Multiple instances

Each process must a priori claim maximum use

When a process requests a resource it may have to wait

When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm

Data Structure for Banker's Algorithm

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available

Max: n x m matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j

Allocation: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j

Need: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$$

Banker’s Algorithm

Safety Algorithm

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) $Need_i$ <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i then the system is in a safe state

Banker's Algorithm

Resource Request Algorithm

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Banker's Algorithm

Example

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Banker's Algorithm

Snapshot at time T_0:

Resource	A	B	C
Instance	10	5	7
Total Allocation	7	2	5
Available	3	3	2

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Need [i] = Max [i] - Allocation [i]
Need for P0: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P1: (3, 2, 2) - (2, 0, 0) = 1, 2, 2
Need for P2: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P3: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P4: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Work = available = 3 3 2
Finish(i) = false for i=0,1,2,3,4

For Process P1:
Need <= work
1, 2, 2 <= 3, 3, 2 condition **true**
New work = work + Allocation
(3, 3, 2) + (2, 0, 0) => 5, 3, 2
Finish(1) = true

For Process P3:
Need <= work
0, 1, 1 <= 5, 3, 2 condition **true**
New work = work + Allocation
(5, 3, 2) + (2, 1, 1) => 7, 4, 3
Finish(3) = true

For Process P4:
Need <= work
4, 3, 1 <= 7, 4, 3 condition **true**
New work = work + Allocation
(7, 4, 3) + (0, 0, 2) => 7, 4, 5
Finish(4) = true

For Process P0:
Need <= work
7, 4, 3 <= 7, 4, 5 condition **true**
New work = work + Allocation
(7, 4, 5) + (0, 1, 0) => 7, 5, 5
Finish(0) = true

For Process P2:
Need <= work
6, 0, 0 <= 7, 5, 5 condition **true**
New work = work + Allocation
(7, 5, 5) + (3, 0, 2) => 10, 5, 7
Finish(2) = true

safe state and the safe sequence is P1 P3 P4 P0 P2

Banker's Algorithm

What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Resource	A	B	C
Instance	10	5	7
Total Allocation	7	2	5
Available	3	3	2

For granting the Request (1, 0, 0),

first we have to check that **Request <= Available**, that is (1, 0, 0) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.