PROCESS MANAGEMENT

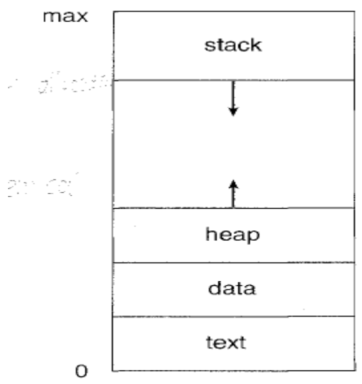
Syllabus:

Define process - process control block (PCB) and its general structure - different states of a process with the help of state diagram.- Define a thread – Comparison between threads and processes. - Multithreading.
Schedulers – long, medium and short term- queuing diagrams. - context switching. CPU and I/O burst cycles - CPU bound and I/O bound processes- difference between preemptive and non-preemptive scheduling- Various scheduling criteria - FCFS, SJF, Priority, and RR scheduling algorithms and their Gantt charts - Multilevel queue and Multilevel feedback queue scheduling
Process synchronization - Co-operating processes - Race condition - Critical section of processes – Critical Section Problem and its solutions- Resource allocation graphs - Deadlock and its causes – Deadlock avoidance, prevention and detection &recovery.

Processes and Threads

PROCESS

- ❖ Process is a program in execution
- ❖ Process execution must progress in sequential fashion.
- ❖ A process includes:
    - program counter : Used to represent current activity
    - stack : Contains temporary data
    - data section : Contains Global variables
    - Heap: dynamic memory allocated during process run time.
- ❖ Program is a passive entity, it won't request for any resources.
- ❖ When user wants to execute a program, it is loaded to main memory
- ❖ At once the program is loaded into main memory, it becomes a process. Now it can request for resources
- ❖ Diagrammatic representation of process in memory:



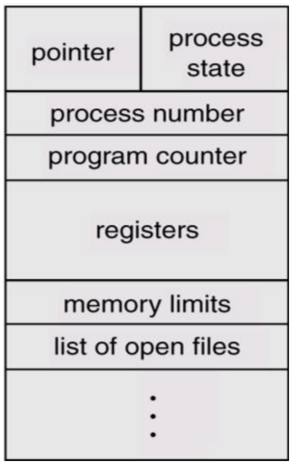
PROCESS CONTROL BLOCK (PCB)

- ❖ Each process is represented in the OS by Process Control Block

- ❖ It contains certain information about the process Such as :
    1. **Process state :** State may be New, ready, running, waiting, halted etc

2. **Program Counter : I**ndicates the address of next instruction to be executed.
3. **CPU registers:** Includes Accumulators, index registers, stack pointers, and general purpose registers and condition code informations.
4. **CPU scheduling information :** Includes process priority, pointers to scheduling queues and other scheduling parameters
5. **Memory-management information :** Includes information such as value of the base and limit registers, page tables or segment tables depending upon the memory system used by the OS
6. **Accounting information :** Information includes the amount of CPU and real time used, time limits, process numbers, account numbers etc
7. **I/O status information:** Information includes list of I/O devices allocated to the process, list of open files etc

❖ Diagrammatic representation process control block as follows:



PROCESS STATES

❖ When a process is in execution, its state changes
❖ The ***state*** of a process is defined by the current activity of that process

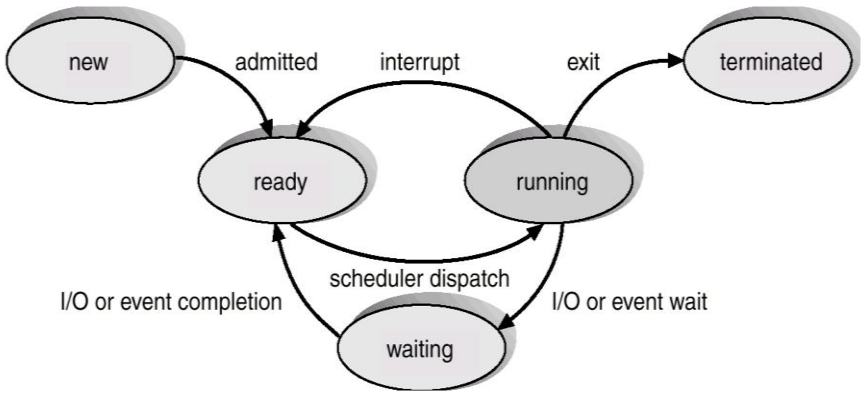


Diagram of process state

- **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.

❖ Only One ***Process*** can be in running state on any processor at any instant of time , although many process can be in ***ready*** and ***waiting***

THREADS

❖ A ***thread*** **(or *lightweight process*)** is a basic unit of CPU utilization.

❖ It consists of:

- Thread ID
- program counter
- register set
- stack space

❖ It shares with other threads belonging to same process its:

- code section
- data section
- operating-system resources

❖ Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.

❖ Single threaded process will be able to service only one client at a time

Comparison between THREADS and PROCESS

PROCESS	THREADS
Heaviest Unit of OS	Lightest Unit of OS
Have Own Resources allocated by the OS	Don’t have Own Resources
Communication between Independent Process need the help of OS	Threads communicate4 each other without the help of OS

Advantages of Threads

❖ **Foreground and Background Work :** More than one operation can be performed at a time with the help of threads

❖ **Speeds Execution:** Multithreaded programs increases the speed of execution

❖ **Easy to create and Destroy :** Since it don’t have any resources associated with it, easy to create and destroy

MULTI THREADING

❖ ***Multithreading*** refers to the ability of an OS to support multiple, concurrent paths of execution within a single process

- Many Multithreading models are available :
  - Many-to-one Model
  - One-to-One Model
  - Many-to-Many Model

CPU SCHEDULING

PROCESS SCHEDULING

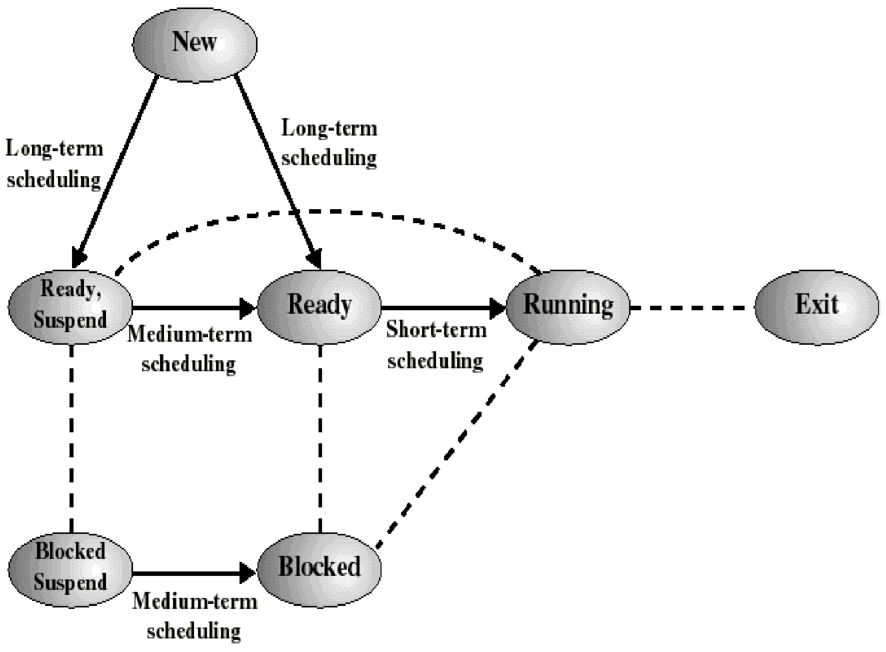
- ***Process Scheduling*** refers to a set of policies and mechanisms built into the OS to control and manage the order of work done by the System.

Objectives of Process Scheduling

- **Maximize System Performance**
- **High Throughput :** Number of processes completed per of unit time
- **Low Response Time :** Time elapsed from the submission of a request until the first response is produced
- **High processor utilization**

SCHEDULERS

- Process scheduling is taken care by a unit of OS called ***Schedulers***
- A Process has to transfer between different queues throughout its life time.
- The selection of process from these queues is done by schedulers
- There are ***Three*** types of Schedulers
  - **Long term Scheduler :** Determines which process to Admit
  - **Medium term Scheduler :** Determines which process to Swap in or Out
  - **Short term Scheduler :** Determines which ready process to execute Next



1. **Long term Schedulers**

- All the User programs are stored in the Secondary memory named ***Job Pool***
- Long term schedulers select programs from a job pool and loads into main memory for execution
- There are ***Two*** decisions involved here:

a) Scheduler must decide when the OS can take one or more additional process
b) Scheduler must decide which job to accept and turn into process

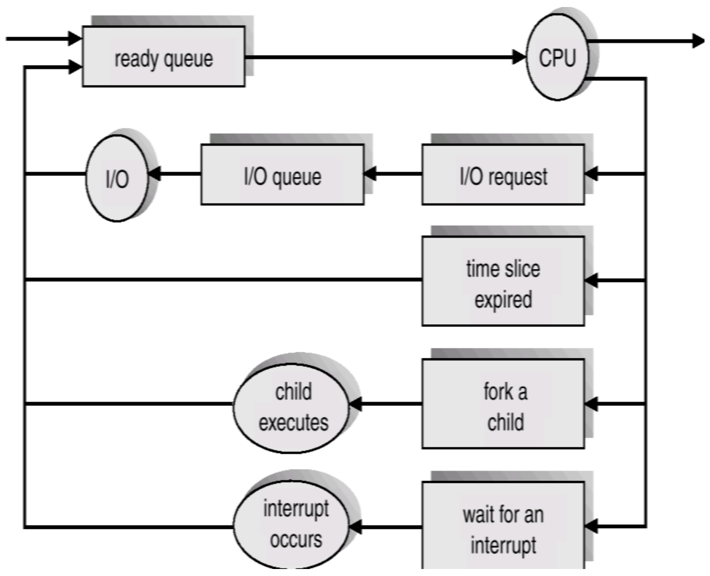
2. **Medium Term Schedulers**

- Controls which remains resident in memory and which jobs must be swapped out to reduce degree of multiprogramming
- Swapping means removing a process from memory temporarily and then bring back to main memory for continued execution
- Medium term schedulers do the job of Swap in and Swap out

3. **Short Term Schedulers**

- Short term scheduling selects a program from main memory and allocates CPU to one of them
- The frequency of execution of the short term schedulers is higher than that of long term schedulers because a process may execute for only a few milliseconds and may wait for an event to occur

Scheduling Queues

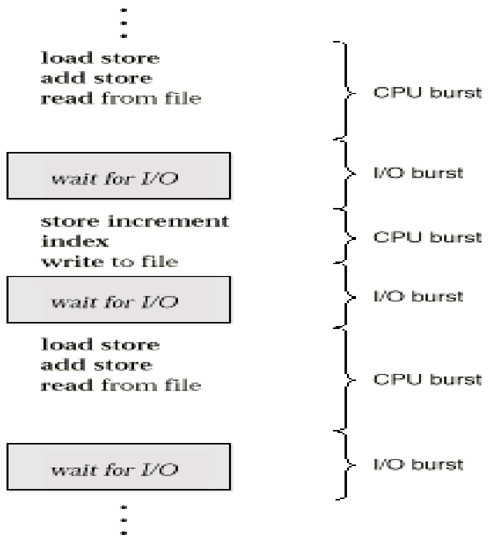


- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** – set of processes waiting for an I/O device.
- Process migrates between the various queues.

CONTEXT SWITCHING

- ❖ Switching of CPU from one process to another is called Context switching
- ❖ The ***Context*** of a process is represented by PCB of a process
- ❖ When Context switch occurs, the kernel saves the context of the old process in its PCB and loads the saves context of the new process
- ❖ Context Switch sometimes highly depend upon Hardware support

CPU and I/O Burst Cycles



- The success of CPU scheduling depends on the following property:
    - Process execution consists of a cycle of CPU execution and I/O wait
    - Processes alternate between these two states
    - Process execution begins with CPU Burst, followed by I/O Burst; then another CPU burst, I/O burst and so on..
    - The Last CPU burst will end with a system request to terminate execution

SCHEDULING CRITERIA

- **CPU utilization** – keeps the CPU as busy as possible
- **Throughput** – Number of processes that complete their execution per unit time
- **Turn Around Time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

TYPES OF SCHEDULING ALGORITHMS

- There are ***two*** types of scheduling algorithms
    - **Pre-Emptive Scheduling**
    - **Non Pre emptive Scheduling**
- 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.

- When Scheduling takes place only under circumstances 1 and 4 it is **N*on pre-emptive* Scheduling**
- All other scheduling is ***preemptive.***

SCHEDULING ALGORITHMS

1. **First-Come, First-Served Scheduling (FCFS)**
2. **Shortest-Job-First Scheduling (SJF) or Shortest Process Next(SPN)**
3. **Priority Scheduling**
4. **Round-Robin Scheduling (RR Scheduling)**
5. **Multilevel Queue Scheduling**
6. **Multilevel Feedback Queue Scheduling**

1. FIRST COME FIRST SERVED (FCFS)

- Simplest scheduling algorithm
- Non pre-emptive algorithm
- Also known as first in, first out(FIFO).
- Jobs arriving are placed at the end of the queue.

- Ready queue is a FIFO queue.
- Processes are dispatched according to their arrival time on the ready queue.

Characteristics:

- The process that requests the CPU, first is allocated the CPU first.
- This can easily be implemented using a queue.
- FCFS is not, preemptive. Once a process has the CPU, it will occupy the CPU until the process completes or voluntarily enters the wait state.
- All the processes wait for the big one processes to get off the CPU, since CPU utilization is low.
- Throughput can be low, since long processes can control the CPU.
- Response turnaround time , waiting time and response time can be high.
- No prioritization.

<u>Example1:</u>

A	**B**	**C**	**D**
10	5	7	6

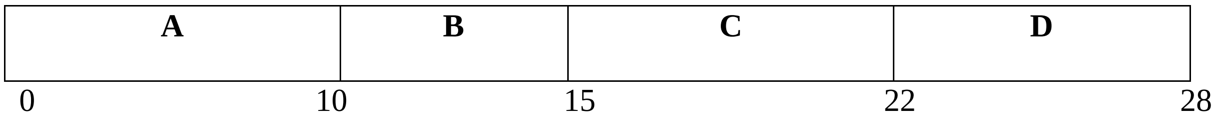
- Four jobs A,B,C,D costem inme into the system this order at about the same time

Process	**Start**	**Running**	**End**
A	0	10	10
B	10	5	15
C	15	7	22
D	22	6	28

Average waiting time = (0+10+15+22)/4 = 47/4
= 11.8

Average turn around time = (10+15+22+28)/4 = 75/4
= 18.8

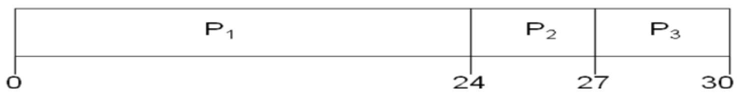
The **Gantt Chart** for the schedule is:



<u>Example2:</u>

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

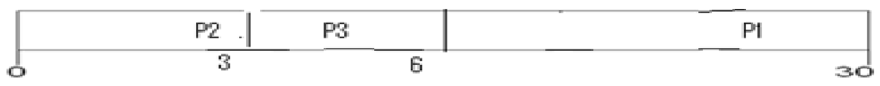
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- The **Gantt Chart** for the schedule is:



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

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

- 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 = 6ms; P_2 = 0 ms; P_3 = 3 ms

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

- ***Much better than previous case.***

2. SHORTEST JOB FIRST SCHEDULING

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
    - Nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
    - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.
- Pre-emptive SJF scheduling is sometimes called as ***Shortest Remaining Time First Scheduling***

Example of Non-Preemptive SJF

Consider the following set of processes, with length of the CPU service time in milliseconds

Process	Service Time(ms)
P1	24
P2	3
P3	2

Let us assume that all the processes arrive at 0 time. The result of the scheduling is shown in the following Gantt chart

P3	P2	P1

0 2 5 29

Waiting Time:

Process P3 = 0 ms; P2=2 ms; P1 = 5ms

Avg Waiting Time =(5+2+0)/3 = 2.337 ms

Example of Preemptive SJF

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

The result of the scheduling is shown in the following Gantt chart

P1	P2	P4	P1	P3

0 1 5 10 17 26

Waiting Time:

Process P1 = (10-1)= 9 ms; P2=(1-1)=0 ms; P3 = (17-2)=15ms P4=(5-3)= 2 ms

Avg Waiting Time =(9+0+15+2)/3 =6.5 ms

- The average waiting time is minimum
- The only difficulty is, it is hard to find the next CPU time and Possibility of starvation for longer processes
- CPU bursts can be predicted using exponential averaging.

$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

1. $t_n =$ actual lenght of n^{th}CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$

3. PRIORITY SCHEUDLING

- Priority is assigned to each process
- CPU is allocated to the process with the highest priority
- IF 2 processs have same priority, FCFS scheduling is used to break the tie
- There are ***two*** scheduling algorithms. They are:
  1. **Non pre emptive**
  2. **Pre emptive**

<u>Non Pre emptive Scheduling</u>

Consider the following set of processes with priority and length of CPU service time in Milliseconds

Process	**Service time**	**Priority**
P1	24	3
P2	3	1
P3	2	5

Let us assumes that all the processes arrives at 0 time and the lower number represents higher priority

The result of the scheduling is shown in the Gantt Chart

P2	P1	P3
0	3	27

Waiting Time:

Process P1 =3 ms; P2 = 0 ms ; P3=27 ms

Avg Waiting Time= (3+0+27)/3 = 10 ms

<u>Pre emptive scheduling</u>

Assume that processes arrives at different times

Process	Arrival Time	service Time	Priority
P1	0	24	3
P2	1	3	1
P3	2	2	5

Execution Procedure :

i) At time= 0 , CPU is given to process P1

ii) When time = 1, a process p2 with higher priority than the currently running process arrives. So the currently running low priority process is taken out(preempted) of CPU and process P2 gains CPU

iii) When time=2 , a process P3 with low priority than the currently running process arrives. So nothing happens.

iv) When process P2 completes, CPU is given to next higher priority process P1 and then P3

The result of the scheduling is as shown below

P1	P2	P1	P3

0 1 4 27 29

Process P1 = (4-1)=3 ms ; P2= (1-1)= 0 ms ; P3=(27-2)= 25 ms
Average Waiting Time = (3+0+25)/3=9.3 ms

➢ The disadvantage of priority scheduling is Starvation – low priority processes may never execute or want to wait infinitely

4. ROUND ROBIN ALGORITHM

❖ Scheduling is designed for time sharing systems
❖ Similar to FCFS with pre emption to switch between process
❖ Each process gets a small unit of CPU time (***time quantum or Time slice***). After this time has elapsed, the process is preempted and added to the end of the ready queue.
❖ The ready queue is treated as a circular queue.New process are added to the tail of the circular queue
❖ If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
❖ If the process completes its execution within the time slice ,it will come out of CPU
❖ IF not, after the time period, the CPU is assigned to other processes waiting in queue and repeat the procedure

Consider the following setoff process arrives at 0 time.
Time slice is 4 ms

Process	Service time
P1	24
P2	3
P3	2

Since the timeslice is fixed as 4 ms, the following happens to process :

1. CPU is given to P1.Eventhough it needs 24 ms, it is given only 4 ms and is taken back out of the CPU and is placed in the tail of the circular ready queue
2. Then P2 will gain CPU.Since it does not need 4 ms, it will quit before the time slot expires
3. Then P3 will gain CPU

4. After all the processes are given 1 time slice, the CPU is given to P1 and so on.

❖ The Gantt chart is as follows

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 9 13 17 21 25 29

Waiting Time :

P1 = 5 ms ; P2 =4 ms ; P3 = 7 ms

Avg Waiting Time = (5+4+7)/3= 5.33 ms

❖ Performance of RR depends on the Time slice
❖ If it is very large, it will be same as FCFS
❖ It it is too small, it will be effective

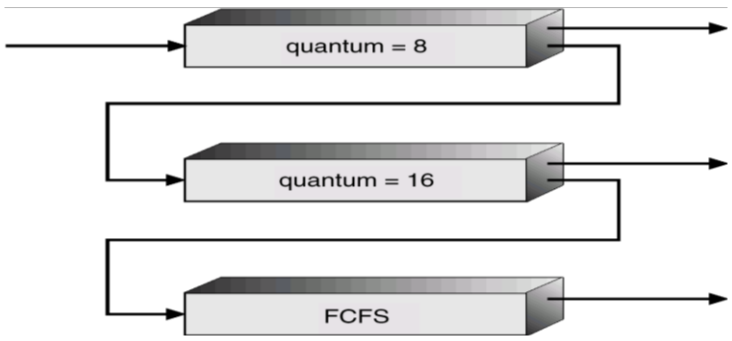
5. MULTI LEVEL FEEDBACK QUEUE SCHEDULING

❖ A process can move between the various queues;
❖ Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

❖ Eg : Consider Three queues:
    Q_0 – time quantum 8 milliseconds
    Q_1 – time quantum 16 milliseconds
    Q_2 – FCFS Scheduling

<u>Scheduling Procedure</u>

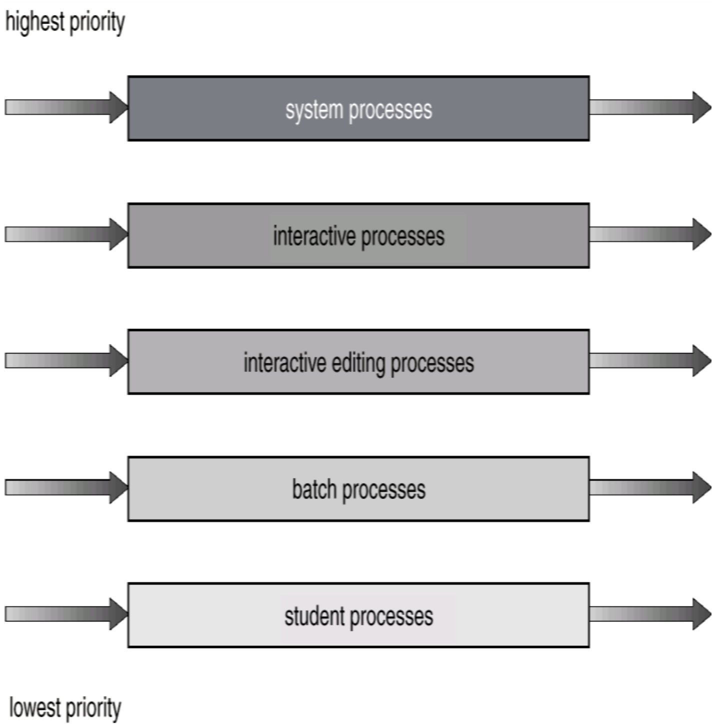
❖ A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
❖ At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



6. MULTI LEVEL QUEUE SCHEDULING

- ❖ Ready queue is partitioned into several separate queues:
  - foreground process (interactive)
  - background process(batch)
- ❖ The processes are permanently assigned to one queue based on some property of the process such as memory size, process type etc
- ❖ Each queue has its own scheduling algorithm,

  Eg: foreground – RR
  background – FCFS
- ❖ Scheduling among queues are implemented as Fixed priority scheduling
- ❖ Fixed priority is shown below:



<u>Process synchronization</u>

<u>Co Operating Processes</u>

- ❖ Process that share data with other processes are called Co operating process
- ❖ ***Cooperating*** process can affect or be affected by the execution of another process

- Advantages of process cooperation are:
    - **Information sharing :** several users are allowed to share a same piece of information
    - **Computation speed-up :** If the system has more than one CPU, one job can be divided into sub jobs and can run parallel with others
    - **Modularity :** Dividing system function into separate threads or process
    - **Convenience :** many tasks can be done at same time

RACE CONDITION

- If more than one process shares a common variable concurrently the result of the execution will be different depending upon the order of execution of the process. This situation is called ***race Condition***
- The order of execution of the co operating process is determined by the OS

Eg : 1
Suppose that two process P1 and P2 share a global variable “a”. At some point in its execution P2 updates “a” to value 2. Thus the two tasks are in race to write the variable “a”.In this example the “loser”of the race(process that updates last) determines the value of a

Eg: 2
Consider two process P3 and P4 that share global variables “b” and “c”, with initial values b=1 and c=2. At some point of the execution P3 executes the assignment b=b+c, and at some point in its execution P4 executes the assignment c=b+c. The final value of the two variables depends on th order of process execution. If P3 executes first, then Final values are b=3 and c=5.
If P4 executes first, values are b=4 and c=3

CRITICAL SECTION

- To avoid the race condition, we must make sure that no two processes share a variable or a file at same time
- This is achieved by ***critical section***
- ***Critical section*** is a code segment in process, in which shared variables are accessed
- The structure of process is as below :

```
do{
  Entry Section
      Critical section
    Exit section

       Remainder section
} While(1);
```

- ❖ To avoid race condition we must make sure that if one process is executing in critical section, no other process is allowed to enter into its critical section
- ❖ Each process must request permission to enter critical section
- ❖ Different sections in Process are :
  - **Entry section :** Contains codes for giving permission for processes to enter into the critical section
  - **Remainder Section :** Section contains codes other than the codes in critical section
  - **Exit Section :** Section contains codes to come out of the critical section.This section must be next to critical section

Solution to Critical-Section Problem

- ❖ Solution to critical section should satisfy the following :
  1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

  2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only the process that are not executing in their remaining section cannot participate, and this selection of the processes that will enter the critical section next cannot be postponed indefinitely.
  3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

TWO PROCESS SOLUTION

- We restrict our attention to algorithms that are applicable to only 2 processes at a time.
- The process are numbered P_i and P_j where j==1-i

Algorithm 1

- Let the two processes share a common integer variable ***turn*** initialized to 0 or 1
- If turn==i, then process Pi is allowed to execute in its critical section .
- The structure of process Pi is shown as below

```
do
{
   While (turn != i);
   Critical section
   turn = j ;
```

```
                Remainder section

          } while(1);
```

- The solution ensures that only one process is executing the critical section at a time.
- It does not satisfy the progress requirement

Algorithm 2

- Algorithm 1 does not retain sufficient information about the state of each process
- Here we will replace the variable *turn* with following array
    - Boolean flag[2];
- The elements of array are initialized to false.
- If flag[i] is true, this value indicates that Pi is ready to enter critical section.
- The structure of process Pi is as shown:

```
do {
    Flag [i] = true ;

    While (flag[j]);
      Critical section
    Flag [i] =  false ;

      Remainder section

      } while (1);
```

Algorithm Description

- ✓ In this algorithm, process Pi first sets flag[i] to be true, signaling that it is ready to enter the critical section.
- ✓ Then Pi checks to verify Pj is not also ready to enter its critical section . If Pj were ready , then Pi would wait until Pj had indicated that it no longer needed to be in the critical section (that is until flag [j]=false)
- ✓ At this point Pi would enter the critical section . On existing the critical section, Pi would set flag[i] to be false, allowing the other process to enter critical section
- Here mutual exclusion is satisfied but progress requirement is not met

Algorithm 3

- ❖ By combining the key ideas of algorithm 1 and algorithm 2 , we obtain a correct solution to critical section problem
- ❖ It have 2 variables

```
      boolean  flag [2] ;
      int turn;
```

- ❖ Initially flag [0] = flag [1] = false;

- ❖ Value of turn is immaterial(but is either 0 or 1)
- ❖ The structure of process Pi is shown as below

```
do {
        Flag [i] = true ;

        Turn = j;

        While (flag [j] && turn == j);

              critical section

            Flag [i] = false;

          Remainder section

} while (1);
```

<u>Algorithm Description</u>

- ❖ To enter the critical section, process Pi first sets flag [i] to be true and set turn to the value j, there by asserting that if the other process wishes to enter the critical section it can do so.
- ❖ If both process try to enter at same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur,but will be overwritten immediately. The value of turn decides which of the process is allowed to enter its critical section

<u>DEAD LOCKS</u>

- ❖ A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- ❖ Under the normal mode of operation, each process utilizes a resource as follows:
    - request
    - use
    - release
- ❖ Each resource type R_i has W_i instances.
- ❖ Deadlock can arise if **four conditions hold simultaneously.**

    - **Mutual exclusion:** only one process at a time can use a resource.
    - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
    - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
    - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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.

Resource-Allocation Graph

- ❖ Deadlocks can be described in terms of directed grap called ***System Resource Allocation Graph***
- ❖ It consists of set of vertices *V* and a set of edges *E*.

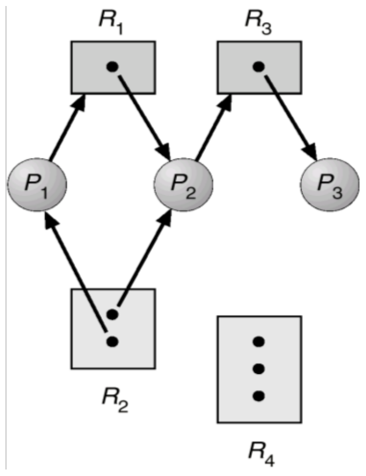
  V is partitioned into two types:

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

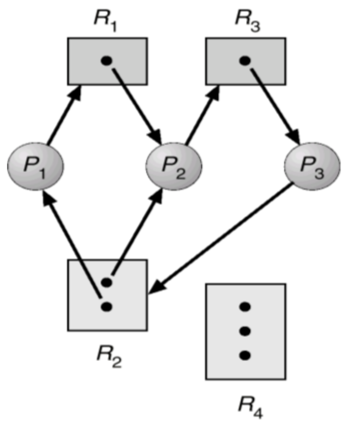
  $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

- ❖ request edge – directed edge $P_1 \rightarrow R_j$
- ❖ assignment edge – directed edge $R_j \rightarrow P_i$

Example of a Resource Allocation Graph



Example of a Resource Allocation Graph with Deadlock



- The sets P, R and E:
  - ✓ P = {P1, P2, P3}
  - ✓ R = {R1, R2, R3, R4}
  - ✓ $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - ✓ One instance of resource type R1
  - ✓ Two instances of resource type R2
  - ✓ One instance of resource type R3
  - ✓ Three instances of resource type R4

- If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Methods for Handling Deadlocks

❖ We can deal with deadlock Problems in 3 ways :
  1. Ensure that the system will *never* enter a deadlock state.
  2. Allow the system to enter a deadlock state and then recover.
  3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

❖ For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock
  - **Mutual Exclusion** – not required for sharable resources; 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. It contains the following 2 protocols
    1. Require process to request and be allocated all its resources before it begins execution,
    2. Allow process to request resources only when the process has none.

- **Circular Wait** - impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

- **No Preemption -** To ensure that this condition does not hold, we can use the following protocol.

  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are pre-empted

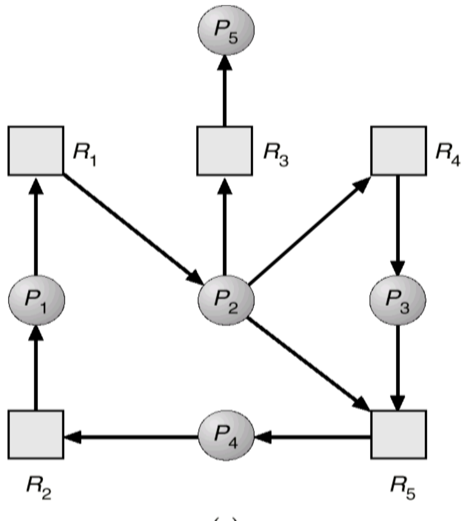
Deadlock Detection

❖ System must provide the :
  - Algorithm that examines the state of the system to determine whether a deadlock has occurred
  - An algorithm to recover from the deadlock

❖ We have two algorithms for this.
  1. Single Instance of Each Resource Type
  2. Several Instances of a Resource Type

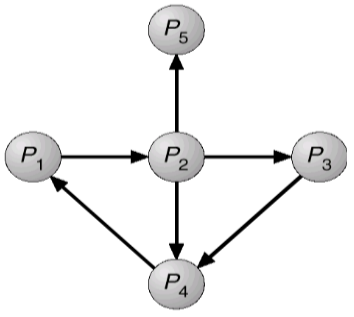
Single Instance of Each Resource Type

- ❖ If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of resource allocation graph, called ***wait- for graph***
- ❖ We obtain this graph from resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges
- ❖ An edge from Pi to Pj in a wait for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- ❖ An edge Pi $\longrightarrow$ Pj exits in wait for graph if and only if the corresponding resource allocation graph contains two edges Pi $\longrightarrow$ Rq and Rq$\longrightarrow$ Pj for some resource Rq
- ❖ Periodically invokes an algorithm that searches for a cycle in the graph.
- ❖ An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



(a)

Resource Allocation Graph



(b)

Wait for graph

Several Instances of a Resource Type

- ❖ This algorithm employs several time varying data structures
  - ***Available**:* A vector of length *m* indicates the number of available resources of each type.
  - ***Allocation**:* An *n x m* matrix defines the number of resources of each type currently allocated to each process.
  - ***Request**:* An *n x m* matrix indicates the current request of each process. If *Request* [*i j*] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize *Work* :=*Available* For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] := false; otherwise, *Finish*[i] := *true*.

2. Find an index *i* such that both:

(a) *Finish*[*i*] = *false*

(b) $Request_i \leq Work$

If no such *i* exists, go to step 4.

. 3. $Work := Work + Allocation_i$
Finish[*i*] := *true*
go to step 2.

4. If *Finish*[*i*] = false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] = *false,* then P_i is deadlocked.