ILAHIA COLLEGE OF ENGINEERING AND TECHNOLOGY

OPERATING SYSTEM-S4 CSE

Module IV

CPU Scheduling – Scheduling Criteria – Scheduling Algorithms.

Deadlocks – Conditions, Modeling using graphs. Handling – Prevention – Avoidance – DetectionRecovery.

SCHEDULING CRITERIA

Scheduling Criteria refers to those set of parameters / metrics / characteristics which are used for comparing different Scheduling Algorithms or Methods.

The following are the major criteria for process scheduling:

1) CPU UTILIZATION It measures the CPU usage in terms of how busy the processors is, or load on the processors.

2) THROUGHPUT It is the number of processes that are completed per unit time. It is dependent on the characteristics and resource requirement of the process being executed.

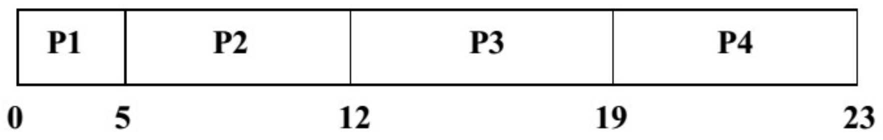
3) TURNAROUND TIME (TAT) It measures the time taken to execute a process.

4) RESPONSE TIME (RT) It is a measure of time taken to produce the first response for a process (before it is given as output to user). It is more relevant in Time-Sharing and Real-Time systems.

5) WAITING TIME (WT) It measures the time a process waits in the ready queue. It is more significant in Multiprogramming systems.

GANTT CHART

- It is used to represent the scheduling of process graphically.
- It is a rectangular time scale diagram with the X-axis depicting the timeline.
- Process IDs of processes allocated to the CPU for a particular time-period is indicated in its rectangular slot.



FEW TERMS THAT NEED TO BE FAMILIARIZED

1) REQUEST RELATED

a) ARRIVAL TIME Time when a job or process is submitted

b) ADMISSION TIME Time when the system starts considering a job or process for scheduling

c) COMPLETION TIME Time by which a job or process is completed

d) DEADLINE Time by which a job or process must be completed to meet the response requirement of a real time application

e) SERVICE TIME The total CPU and I/O time required by a job or process, or sub-request, to complete its operation

f) PREEMPTION Forced deallocation of CPU from a job or process

CPU SCHEDULING ALGORITHMS

The following are the major types of CPU Scheduling algorithms:

1) NON-PREEMPTIVE ALGORITHMS

a) FIRST-COME FIRST SERVE (FCFS) SCHEDULING

b) SHORTEST JOB FIRST (SJF) SCHEDULING

c) PRIORITY NON-PREEMPTIVE (P-NP) SCHEDULING

2) PREEMPTIVE ALGORITHMS

a) SHORTEST REMAINING TIME FIRST (SRTF) SCHEDULING

b) PRIORITY PREEMPTIVE (PRI-P) SCHEDULING

c) ROUND ROBIN SCHEDULING

3) HIGHEST RESPONSE RATIO NEXT SCHEDULING

4) MULTILEVEL QUEUE SCHEDULING

5) MULTILEVEL FEEDBACK QUEUE SCHEDULING

6) MULTIPLE-PROCESSOR SCHEDULING

7) REAL-TIME SCHEDULING

First Come First Serve (FCFS)

- The process that request first for the processor is allotted the processor first.
- The ready queue is maintained as a FIFO queue.
- When the processor becomes free it is allotted to the process at the head of the queue and when a new process arrives it is entered at the tail of the queue.
- It is the simplest algorithm and easy to implement.

Process	Burst Time
P1	6
P2	8
P3	7

P4	3

P1	P2	P3	P4

0 6 14 21
24

Average waiting time = (0+6+14+21)/4 = 10.25ms.

If there is one CPU bound process and many I/O bound processes, while CPU bound process executes all the I/O bound process will finish their job with the I/O devices and will wait for the CPU in the ready queue. During this time the I/O devices will be idle. When the CPU bound process finish its CPU burst it goes to do I/O, meantime all the I/O bound processes will finish their CPU burst since they have got very short CPU bursts, and will wait in the device queue. All the processes waiting for one big process to get off the CPU is called ***convoy effect.***

Shortest Job First Scheduling (SJF)

- Here among the processes in the ready queue, the one with the least CPU burst time will be allotted the processor first.
- If two processes have their burst time equal, then FCFS scheduling is used.
- The average waiting time is small when compared to FCFS scheduling.
- SJF is optimal algorithm.
- It is difficult to implement because knowing the length of the next CPU burst is difficult.

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

P4	P1	P3	P2

0 3 9 16 24

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

SJF can be non preemptive or preemptive. The preemptive version of SJF is called ***Shortest Remaining Time First*** scheduling. Here when a new job arrives in the ready queue, its CPU burst is compared with the remaining CPU burst of the currently running process. If the newly arrived process has got the smaller burst then the currently running process will be preempted and the new process will be allotted to the processor. The preempted process will enter the ready queue and wait for its turn to get the processor.

Process	Arrival time	Burst Time
P1	0	6
P2	1	8
P3	2	7
P4	3	3

P1	P4	P1	P3	P2	
0	3	6	9	16	24

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

Priority Scheduling

Each process is assigned a priority number (integer). The CPU is allocated to the process with the highest priority (smallest integer may be considered as highest priority). Priority scheduling can be preemptive or non preemptive. In preemptive priority scheduling, while a process is executing, if a new process with high priority than the currently running process arrives, the process is preempted and the processor is allocated to the new process. SJF is a priority scheduling where priority is the predicted next CPU burst time.

The problem with priority scheduling is that a low priority processes may have to wait indefinitely to get the processor to execute. This is called ***starvation***. The solution is to increase the priority of the process as time progresses. This is called ***aging***.

Round Robin Scheduling

Each process gets a small amount of CPU time known as time quantum. When the time elapses process is preempted and the processor is allocated to the next process at the head of the ready queue. When the time quantum is very high this scheduling behaves as FCFS and when the time quantum is very small overhead will be high. It should be large with respect to context switch.

Consider the following table (let the time quantum be 5ms)

Process	Burst Time
P1	6
P2	8

P3	7
P4	3

P1	P2	P3	P4	P1	P2	P3

0 5 10 15 18 19 22 24

Average waiting time = (0+5+10+15)/4 = 14.75ms.

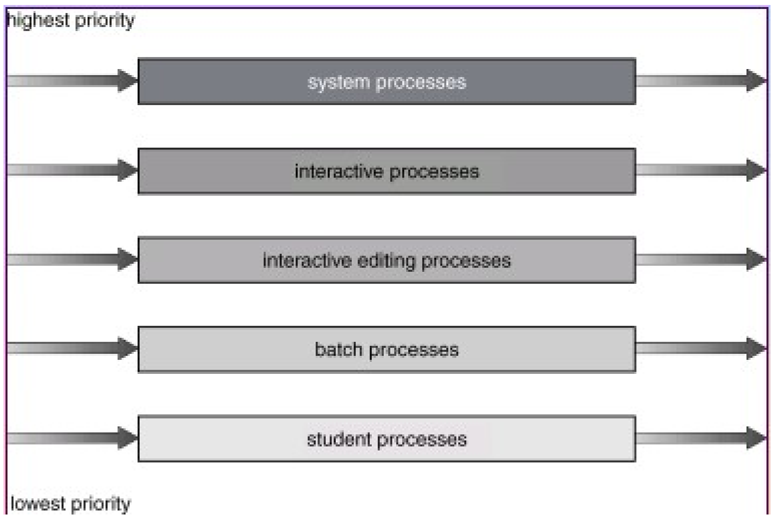
This scheduling exhibits higher waiting time and turnaround time than SJF but the response time is better.

Multilevel Queue Scheduling

Ready queue is partitioned into separate queues, for example, foreground (interactive) and background (batch).

Each queue has its own scheduling algorithm, foreground – RR, background – FCFS

Scheduling must be done between the queues. Any scheduling algorithm can be used for this purpose. If fixed priority scheduling is used for this purpose (i.e., serve all from foreground then from background). There is possibility of starvation. If RR is used each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS



Multilevel Feedback Queue Scheduling

In multilevel queue scheduling once a process enters a queue, it remains there. But in multilevel feedback queue scheduling a process can move between the various queues. Thus aging can be implemented.

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

For example let us consider three queues:

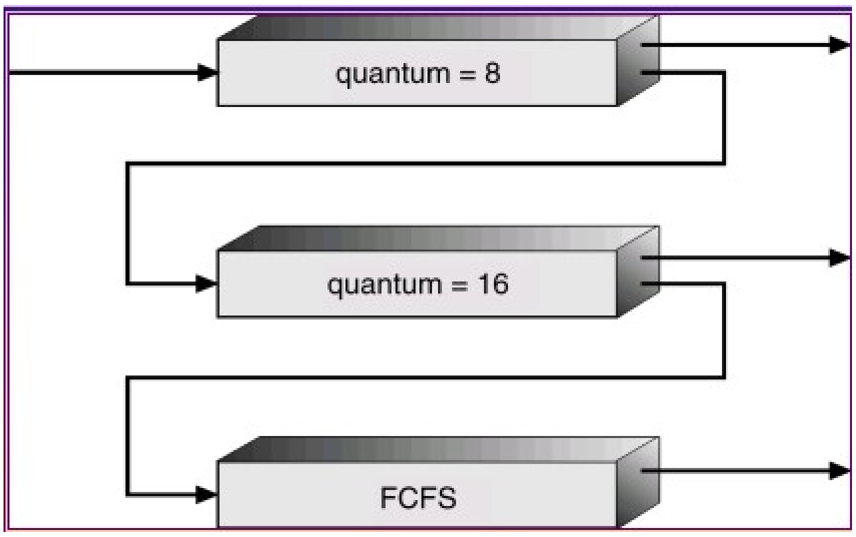
Q_0 – time quantum 8 milliseconds

Q_1 – time quantum 16 milliseconds

Q_2 – FCFS

Scheduling

A new job enters queue Q_0 is served using 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. In

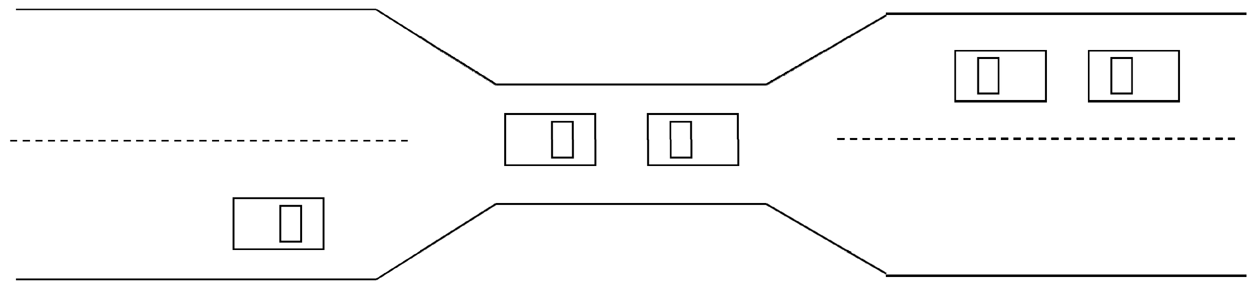


Deadlocks

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- If a system has two cpus, then the resource type cpu has two instances
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
    1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
    2. Use - The process uses the resource, e.g. prints to the printer or reads from the file.
    3. Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).

Deadlock Characterization

Necessary Conditions For Deadlocks

A deadlock occurs in a system if the following four conditions hold simultaneously:

1. Mutual exclusion: At least one of the resources is non-sharable that is only one process at a time can use the resource.
2. Hold and wait: A process exists that is holding on to at least one resource and waiting for an additional resource held by another process.
3. No preemption: Resources cannot be preempted, i.e. a resource is released only by the process that is holding it,after that process has completed its task
4. Circular wait: There exist a set of processes P_0, P_1, ….., P_n of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ….., P_{n-1} is waiting for a resource held P_n and P_n is in turn waiting for a resource held by P_0.

Resource Allocation Graph

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource ($P_i \rightarrow R_j$) signifies a request from a process P_i for an instance of the resource R_j and P_i is waiting for R_j. A directed edge from a resource to a process ($R_j \rightarrow P_i$) indicates that an instance of the resource R_j has been allotted to process P_i. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted. Consider the following system:

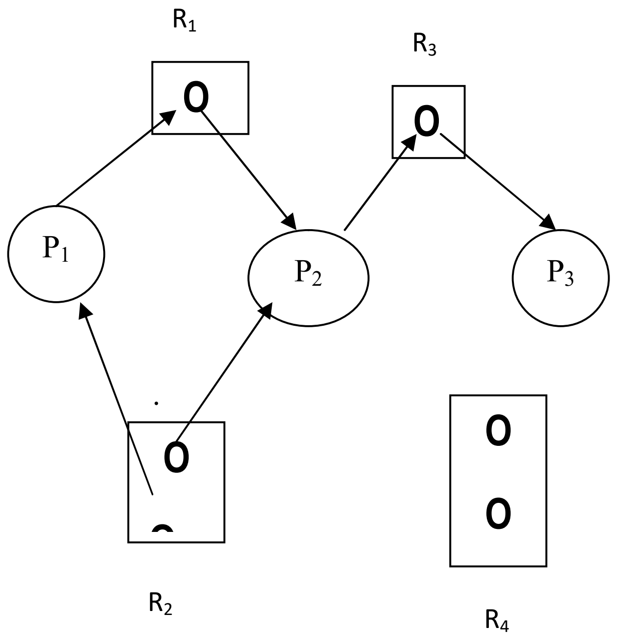
There are 3 processes P_1, P_2 and P_3.

Resources R_1, R_2, R_3 and R_4 have instances 1, 2, 1, and 3 respectively.

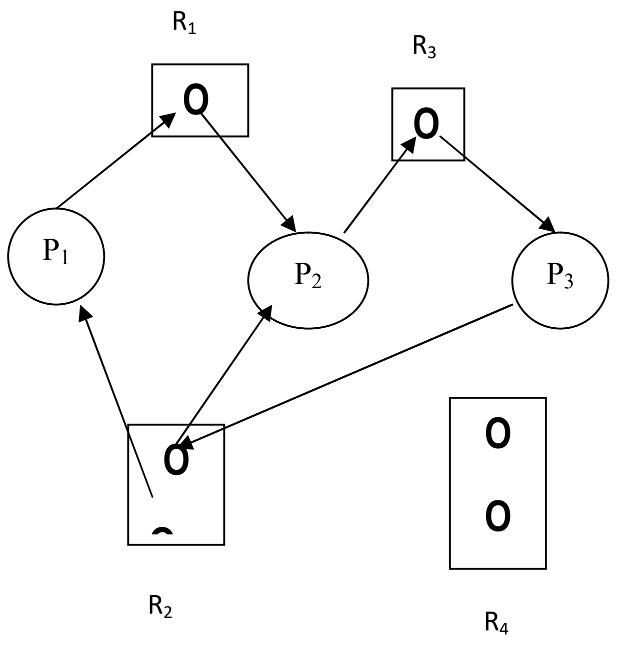
P_1 is holding R_2 and waiting for R_1.

P_2 is holding R_1, R_2 and is waiting for R_3.
P_3 is holding R_3.

The resource allocation gragh for a system in the above situation is as shown below



If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. **If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for existence of a deadlock.**



Here two cycles exist:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_0, P_1 and P_3 are deadlocked and are in a circular wait. P_2 is waiting for R_3 held by P_3. P_3 is waiting for P_1 or P_2 to release R_2. So also P_1 is waiting for P_2 to release R_1**. If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock.**

Deadlock Handling

Different methods to deal with deadlocks include

- methods to ensure that the system will never enter into a state of deadlock,]
- methods that allow the system to enter into a deadlock and then recover or
- just ignore the problem of deadlocks.

To ensure that deadlocks never occur, deadlock prevention / avoidance schemes are used. The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. Deadlock prevention ensures that at least one of the four necessary conditions for deadlocks do not hold. To do this the scheme enforces constraints on requests for resources. Dead lock avoidance scheme requires the operating system to know in advance, the resources needed by a process for its entire lifetime. Based on this a priori information, the process making a request is either made to wait or not to wait in case the requested resource is not readily available. If none of the above two schemes are used, then deadlocks may occur. In such a case, an algorithm to recover from the state of deadlock is used.

If the problem of deadlocks is ignored totally that is to say the system does not ensure that a deadlock does not occur and also does not provide for recovery from deadlock and such a situation arises, then there is no way out of the deadlock. Eventually the system may crash because more and more processes request for resources and enter into deadlock.

Deadlock Prevention

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus prevention of deadlock is possible by ensuring that at least one of the four conditions cannot hold.

1. Mutual exclusion: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneous access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.

2. Hold and wait: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:

1. A process requests for and gets allocated all the resources it uses before execution begins.
2. A process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems to be applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

3. No preemption: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource, if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true that is the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.

Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.

Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.

4. Circular wait: Resource types need to be ordered and processes requesting for resources will do so in increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus $F: R \rightarrow N$ is a 1:1 function that maps resources to numbers. For example:

F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

To ensure that deadlocks do not occur, each process can request for resources only in increasing order of these numbers. A process to start with in the very first instance can request for any resource say R_i. There after it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.

The mapping function F should be so defined that resources get numbers in the usual order of usage.

Deadlock Avoidance

Deadlock prevention algorithms ensure that at least one of the four necessary conditions for deadlocks namely mutual exclusion, hold and wait, no preemption and circular wait do not hold. The disadvantage with prevention algorithms is poor resource utilization and thus reduced system throughput.

An alternate method is to avoid deadlocks. In this case additional a priori information about the usage of resources by processes is required. This information helps to decide on whether a process should wait for a resource or not. Decision about a request is based on all the resources available, resources allocated to processes, future requests and releases by processes.

A deadlock avoidance algorithm requires each process to make known in advance the maximum number of resources of each type that it may need. Also known is the maximum number of resources of each type available. Using both the above a priori knowledge, deadlock avoidance algorithm ensures that a circular wait condition never occurs.

Safe State

A system is said to be in a safe state if it can allocate resources up to the maximum available and is not in a state of deadlock. A safe sequence of processes always ensures a safe state. A sequence of processes $< P_1, P_2, \ldots, P_n >$ is safe for the current allocation of resources to processes if resource requests from each P_i can be satisfied from the currently available resources and the resources held by all P_j where $j < i$. If the state is safe then P_i requesting for resources can wait till P_j's have completed. If such a safe sequence does not exist, then the system is in an unsafe state.

A safe state is not a deadlock state. Conversely a deadlock state is an unsafe state. But all unsafe states are not deadlock states as shown below:

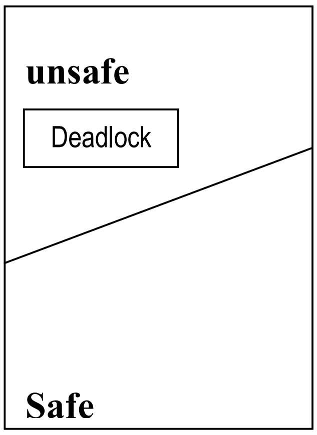


Figure : Safe, unsafe and deadlock state spaces.

If a system is in a safe state it can stay away from an unsafe state and thus avoid deadlock. On the other hand, if a system is in an unsafe state, deadlocks cannot be avoided.

Illustration: A system has 12 instances of a resource type and 3 processes using these resources. The maximum requirements for the resource by the processes and their current allocation at an instance say t_0 is as shown below:

Process	Maximum	Current
P0	10	5
P1	4	2
P3	9	2

At the instant t_0, the system is in a safe state and one safe sequence is < P_1, P_0, P_2 >. This is true because of the following facts:

Out of 12 instances of the resource, 9 are currently allocated and 3 are free. P_1 needs only 2 more, its maximum being 4, can be allotted 2. Now only 1 instance of the resource is free. When P_1 terminates, 5 instances of the resource will be free. P_0 needs only 5 more, its maximum being 10, can be allotted 5. Now resource is not free. Once P_0 terminates, 10 instances of the resource will be free. P_3 needs only 7 more, its maximum being 9, can be allotted 7. Now 3 instances of the resource are free. When P_3 terminates, all 12 instances of the resource will be free.

Thus the sequence < P_1, P_0, P_3 > is a safe sequence and the system is in a safe state. Let us now consider the following scenario at an instant t_1. In addition to the allocation shown in the table above, P_2 requests for 1 more instance of the resource and the allocation is made. At the instance t_1, a safe sequence cannot be found as shown below:

Out of 12 instances of the resource, 10 are currently allocated and 2 are free. P_1 needs only 2 more, its maximum being 4, can be allotted 2. Now resource is not free. Once P_1 terminates, 4 instances of the resource will be free. P_0 needs 5 more while P_2 needs 6 more. Since both P_0 and P_2 cannot be granted resources, they wait. The result is a deadlock.

Thus the system has gone from a safe state at time instant t_0 into an unsafe state at an instant t_1. The extra resource that was granted to P_2 at the instant t_1 was a mistake. P_2 should have waited till other processes finished and released their resources.

Since resources available should not be allocated right away as the system may enter an unsafe state, resource utilization is low if deadlock avoidance algorithms are used.

1.Resource Allocation Graph Algorithm

A resource allocation graph could be used to avoid deadlocks. If a resource allocation graph does not have a cycle, then the system is not in deadlock. But if there is a cycle then the system may be in a deadlock. If the resource allocation graph shows only resources that have only a single instance, then a cycle does imply a deadlock. An algorithm for avoiding deadlocks where resources have single instances in a resource allocation graph is as described below.

The resource allocation graph has request edges and assignment edges. Let there be another kind of edge called a claim edge. A directed edge $P_i \rightarrow R_j$ indicates that P_i may request for the resource R_j some time later. In a resource allocation graph a dashed line represents a claim edge. Later when a process makes an actual request for a resource, the corresponding claim edge is converted to a request edge $P_i \rightarrow R_j$. Similarly when a process releases a resource after use, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Thus a process must be associated with all its claim edges before it starts executing.

If a process Pi requests for a resource R_j, then the claim edge $P_i \rightarrow R_j$ is first converted to a request edge $P_i \rightarrow R_j$. The request of P_i can be granted only if the request edge when converted to an assignment edge does not result in a cycle.

If no cycle exists, the system is in a safe state and requests can be granted. If not the system is in an unsafe state and hence in a deadlock. In such a case, requests should not be granted. This is illustrated below (Figure 5.5a, 5.5b).

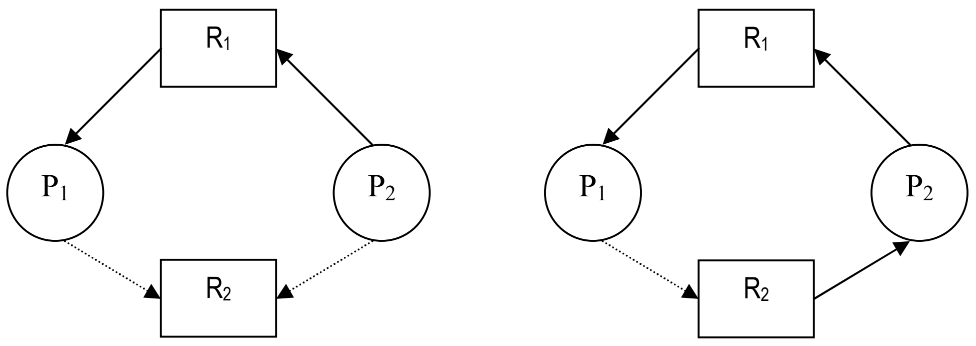


Figure : Resource allocation graph showing safe and deadlock states.

Consider the resource allocation graph shown on the left above. Resource R_2 is currently free. Allocation of R_2 to P_2 on request will result in a cycle as shown on the right. Therefore the system will be in an unsafe state. In this situation if P_1 requests for R_2, then a deadlock occurs.

2.Banker's Algorithm

The resource allocation graph algorithm is not applicable where resources have multiple instances. In such a case Banker's algorithm is used.

A new process entering the system must make known a priori the maximum instances of each resource that it needs subject to the maximum available for each type. As execution proceeds and requests are made, the system checks to see if the allocation of the requested resources ensures a safe state. If so only are the allocations made, else processes must wait for resources.

The following are the data structures maintained to implement the Banker's algorithm:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant. Available[j] = k means to say there are k instances of the j^{th} resource type R_j.
4. Max: is a demand vector of size n x m. It defines the maximum needs of each resource by the process. Max[i][j] = k says the i^{th} process P_i can request for at most k instances of the jth resource type R_j.
5. Allocation: is an n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes. If Allocation[i][j] = k then i^{th} process P_i is currently holding k instances of the j^{th} resource type R_j.
6. Need: is also an n x m vector which gives the remaining needs of the processes. Need[i][j] = k means the ith process P_i still needs k more instances of the j^{th} resource type R_j. Thus Need[i][j] = Max[i][j] – Allocation[i][j].

2.1 Safety Algorithm

Using the above defined data structures, the Banker's algorithm to find out if a system is in a safe state or not is described below:

1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize Work = Available and Finish[i] = false for i = 1, 2, ..., n.
3. Find an i such that

a. Finish[i] = false and

b. $Need_i$ <= Work ($Need_i$ represents the ith row of the vector Need).
If such an i does not exist , go to step 5.

4. Work = Work + $Allocation_i$
Finish[i] = true

   Go to step 3.

5. If finish[i] = true for all i, then the system is in a safe state.

2.2 Resource-Request Algorithm

Let $Request_i$ be the vector representing the requests from a process P_i. Requesti[j] = k shows that process P_i wants k instances of the resource type R_j. The following is the algorithm to find out if a request by a process can immediately be granted:

1. If $Request_i$ <= $Need_i$, go to step 2.
   else Error "request of P_i exceeds Max_i".

2. If $Request_i$ <= $Available_i$, go to step 3.
   else P_i must wait for resources to be released.

3. An assumed allocation is made as follows:
   Available = Available – $Request_i$

   $Allocation_i$ = $Allocation_i$ + $Request_i$

   $Need_i$ = $Need_i$ – $Request_i$

If the resulting state is safe, then process P_i is allocated the resources and the above changes are made permanent. If the new state is unsafe, then P_i must wait and the old status of the data structures is restored.

Illustration: n = 5 < P_0, P_1, P_2, P_3, P_4 >

M = 3 < A, B, C >

Initially Available = < 10, 5, 7 >

At an instant t_0, the data structures have the following values:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

To find a safe sequence and to prove that the system is in a safe state, use the safety algorithm as follows:

Step	Work	Finish	Safe sequence
0	3 3 2	F F F F F	< >
1	5 3 2	F T F F F	$< P_1 >$
2	7 4 3	F T F T F	$< P_1, P_3 >$
3	7 4 5	F T F T T	$< P_1, P_3, P_4 >$
4	7 5 5	T T F T T	$< P_1, P_3, P_4, P_0 >$
5	10 5 7	T T T T T	$< P_1, P_3, P_4, P_0, P_2 >$

Now at an instant t_1, $Request_1 = < 1, 0, 2 >$. To actually allocate the requested resources, use the request-resource algorithm as follows:

$Request_1 < Need_1$ and $Request_1 < Available$ so the request can be considered. If the request is fulfilled, then the new the values in the data structures are as follows:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0

P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

Use the safety algorithm to see if the resulting state is safe:

Step	Work	Finish	Safe sequence
0	2 3 0	F F F F F	< >
1	5 3 2	F T F F F	$< P_1 >$
2	7 4 3	F T F T F	$< P_1, P_3 >$
3	7 4 5	F T F T T	$< P_1, P_3, P_4 >$
4	7 5 5	T T F T T	$< P_1, P_3, P_4, P_0 >$
5	10 5 7	T T T T T	$< P_1, P_3, P_4, P_0, P_2 >$

Since the resulting state is safe, request by P_1 can be granted.

Now at an instant t_2 $Request_4$ = < 3, 3, 0 >. But since $Request_4$ > Available, the request cannot be granted. Also $Request_0$ = < 0, 2, 0> at t_2 cannot be granted since the resulting state is unsafe as shown below:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Using the safety algorithm, the resulting state is unsafe since Finish is false for all values of i and we cannot find a safe sequence.

Step	Work	Finish	Safe sequence

0 2 1 0 F F F F F < >

Deadlock Detection

If the system does not ensure that a deadlock cannot be prevented or a deadlock cannot be avoided, then a deadlock may occur. In case a deadlock occurs the system must

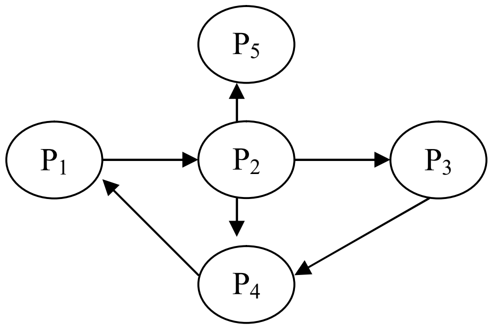
1. detect the deadlock
2. recover from the deadlock

Single Instance Of A Resource

If the system has resources, all of which have only single instances, then a deadlock detection algorithm, which uses a variant of the resource allocation graph, can be used. The graph used in this case is called a **wait-for** graph.

The wait-for graph is a directed graph having vertices and edges. The vertices represent processes and directed edges are present between two processes one of which is waiting for a resource held by the other. Two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ in the resource allocation graph are replaced by one edge $P_i \rightarrow P_j$ in the wait-for graph. Thus the wait-for graph is obtained by removing vertices representing resources and then collapsing the corresponding edges in a resource allocation graph.

An Illustration is shown below:



Wait-for graph

As in the previous case, a cycle in a wait-for graph indicates a deadlock. Therefore the system maintains a wait-for graph and periodically invokes an algorithm to check for a cycle in the wait-for graph.

Multiple Instances Of A Resource

A wait-for graph is not applicable for detecting deadlocks where there exist multiple instances of resources. This is because there is a situation where a cycle may or may not indicate a deadlock. If this is so then a decision cannot be made. In situations where there are multiple instances of resources, an algorithm similar to Banker's algorithm for deadlock avoidance is used.

Data structures used are similar to those used in Banker's algorithm and are given below:

1. n: Number of processes in the system.
2. m: Number of resource types in the system.
3. Available: is a vector of length m. Each entry in this vector gives maximum instances of a resource type that are available at the instant.
4. Allocation: is an n x m vector which at any instant defines the number of resources of each type currently allocated to each of the m processes.
5. Request: is also an n x m vector defining the current requests of each process. Request[i][j] = k means the i^{th} process P_i is requesting for k instances of the j^{th} resource type R_j.

ALGORITHM

```
1. Define a vector Work of length m and a vector Finish of length n.
2. Initialize    Work = Available and
   For i = 1, 2, ….., n

   If $Allocation_i$ != 0

         Finish[i] = false

   Else

         Finish[i] = true

3. Find an i such that
     a. Finish[i] = false and
     b. $Request_i$ <= Work
          If such an i does not exist , go to step 5.

4. Work = Work + $Allocation_i$
    Finish[i] = true

    Go to step 3.

5. If finish[i] = true for all i, then the system is not in deadlock.
   Else the system is in deadlock with all processes corresponding to Finish[i] = false being
   deadlocked.
```

Illustration: n = 5 $< P_0, P_1, P_2, P_3, P_4 >$

M = 3 < A, B, C >

Initially Available = < 7, 2, 6 >

At an instant t0, the data structures have the following values:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

To prove that the system is not deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	< P_0 >
2	3 1 3	T F T F F	< P_0, P_2 >
3	5 2 4	T F T T F	< P_0, P_2, P_3 >
4	5 2 6	T F T T T	< P_0, P_2, P_3, P_4 >
5	7 2 6	T T T T T	< P_0, P_2, P_3, P_4, P_1 >

Now at an instant t_1, $Request_2$ = < 0, 0, 1 > and the new values in the data structures are as follows:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	

P_4	0 0 2	0 0 2

To prove that the system is deadlocked, use the above algorithm as follows:

Step	Work	Finish	Safe sequence
0	0 0 0	F F F F F	< >
1	0 1 0	T F F F F	$< P_0 >$

The system is in deadlock with processes P_1, P_2, P_3, and P_4 deadlocked.

WHEN TO INVOKE?

The deadlock detection algorithm takes $m \times n^2$ operations to detect whether a system is in deadlock. How often should the algorithm be invoked? This depends on the following two main factors:

1. Frequency of occurrence of deadlocks
2. Number of processes involved when it occurs

If deadlocks are known to occur frequently, then the detection algorithm has to be invoked frequently because during the period of deadlock, resources are idle and more and more processes wait for idle resources.

Deadlocks occur only when requests from processes cannot be immediately granted. Based on this reasoning, the detection algorithm can be invoked only when a request cannot be immediately granted. If this is so, then the process causing the deadlock and also all the deadlocked processes can also be identified.

But invoking the algorithm an every request is a clear overhead as it consumes CPU time. Better would be to invoke the algorithm periodically at regular less frequent intervals. One criterion could be when the CPU utilization drops below a threshold. The drawbacks in this case are that deadlocks may go unnoticed for some time and the process that caused the deadlock will not be known.

Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

1 Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:

- Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
- Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

- In the latter case there are many factors that can go into deciding which processes to terminate next:

1. Process priorities.
2. How long the process has been running, and how close it is to finishing.
3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
4. How many more resources does the process need to complete.
5. How many processes will need to be terminated
6. Whether the process is interactive or batch.
7. (Whether or not the process has made non-restorable changes to any resource.)

2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
    1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
    2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
    3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.