Operating System: Methods of Handling Deadlocks

Prepared By:

Dr. Sanjeev Gangwar

Assistant Professor,

Department of Computer Applications,

VBS Purvanchal University, Jaunpur

METHODS FOR HANDLING DEADLOCK /DETECTION

There are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system.

To ensure that deadlock never occur the system can use either a deadlock prevention or deadlock avoidance scheme.

Deadlock Prevention- Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions cannot hold.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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 Low resource utilization; starvation possible
- **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- Requires additional information about how resources are to be used. 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.

Safe State- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes is a safe sequence for the current allocation state if, for each Pi the resources that Pj can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with $j < i$. In this situation, if

the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When Pi terminates, Pi + 1 can obtain its needed resources, and so on.

Banker's Algorithm- This algorithm can be used in banking system to ensure that the bank never allocates all its available cash such that it can no longer satisfy the needs of all its customers. This algorithm is applicable to a system with multiple instances of each resource type. When a new process enters in to the system it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. Several data structure must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let 'n' be the number of processes in the system and 'm' be the number of resource types. We need following data structures:

- **Available:** Vector of length m. If Available[j] = k, there are k instances of resource type Rj available.
- **Max:** n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
- **Allocation:** n x m matrix. If Allocation [i,j] = k then Pi is currently allocated k instances of Rj.
- **Need:** n x m matrix. If Need [i,j] = k, then Pi may need k more instances of Rj to complete its task. Need [i,j] = Max[i,j] – Allocation [i,j].

Safety Algorithm- the algorithm for finding out whether or not a system is in a safe state can be described as follows:

(1) Let work and finish be vectors of length m and n respectively.

Initialize: Work = Available

Finish [i] = False; for i= 1, 2,……,n

(2) Find an I such that both
   (a) Finish[i] =false
   (b) $Need_i \leq Work$

   If no such I exist go to step (4)

(3) Work = Work + Allocationi
   Finish [i] = true
   goto step 2

(4) If Finish [i] = true for all i,

Then the system is in safe state.

This algorithm will require an order of m x n^2 operations to decide whether a state is safe.

Resource Allocation Algorithm Request

Let Requesti be the request vector for process Pi. If Requesti [j] = k then process Pi wants k instances of resource type Rj. When a request for resources is made by process P1, the following actions are taken:

(1) If Requesti ≤ Needi
go to step 2; Otherwise,
raise an error condition, since process has exceeded its maximum claim.
(2) If Requesti ≤ Available
go to step 3; Otherwise,
Pi must wait, since resources are not available.
(3) Have the system pretend to have allocated the requested resources to Pi by modifying the state as follows:
Available = Available – Request;
Allocationi= Allocationi + Requesti ;
Needi=Needi – Requesti ;

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

Example: Considering a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 Instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	Allocation A B C	Max A B C	Available A B C
P0	**0 1 0**	**7 5 3**	**3 3 2**
P1	**2 0 0**	**3 2 2**	
P2	**3 0 2**	**9 0 2**	
P3	**2 1 1**	**2 2 2**	
P4	**0 0 2**	**4 3 3**	

(a) What will be the content of need matrix?
(b) Is the system in safe state? If yes, then what is the safe sequence?
(c) What will happen if process Pi requests one additional instance of resource type A and two instances of resource type C.
(d) If a request (3, 3, 0) by process P4 arrives in the state defined by (c), can it be granted immediately?
(e) If a request (0, 2, 0) by process P0 arrives then check whether whether it is granted or not?

Solution- As we know that

(a) Need [i, j] = Max [i j]- Allocation [i, j]

So the content of Need matrix is

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

(b) Applying safety algorithm on the given system.

For Pi, if Needi ≤ Available

Then Pi is in safe sequence

Available = Available + Allocationi

So for P0 Need= 7, 4, 3

Available = 3, 3, 2

Condition is false so P0 must wait

Now for P1 Need= 1, 2, 2

Available = 3, 3, 2

1, 2, 2 ≤ 3, 3, 2

So P1 will be kept in safe sequence.

Now Available = 3, 3, 2 + 2, 0, 0 = 5, 3, 2

Now for P2 Need= 6, 0, 0

Available= 5, 3, 2

Condition is again false so P2 must wait.

For P3 Need= 0, 1, 1

Available= 5, 3, 2

Condition is true so P3 will be in safe sequence

Then Available = 5, 3, 2 + 2, 1, 1 = 7, 4, 3

For P4 Need= 4, 3, 1

Available= 7, 4, 3

Condition is true so P4 will be in safe sequence

Then Available= 7, 4, 3 + 0, 0, 2 = 7, 4, 5

Now we have two processes P0 and P2 in waiting state. As current available either P0 or P2 can be kept in safe sequence

Firstly we take P2 whose Need= 6, 0, 0

Available = 7, 4, 5

6, 0, 0 ≤ 7, 4, 5 so P4 will be in safe sequence

Then Available = 7, 4, 5 + 3, 0, 2= 10, 4, 7

Next P0 whose Need = 7, 4, 3

Available = 10, 4, 7

7, 4, 3 ≤ 10, 4, 7 is true the P0 comes in safe state

Available = 10, 4, 7 + 0, 1, 0 = 10, 5, 7

So the safe sequence is < P1, P3, P4, P2, P0 >

and the system is in safe state.

(c) Since P1 requests some additional instances of resources such that :

Requesti = (1, 0, 2)

To decide that whether this request is immediately granted we first check that

Requesti ≤ Available

i.e. (1, 0, 2) ≤ (3, 3, 2) which is true

so the request may be granted

To confirm that this request is granted we check the new state by applying safety algorithm that our system is in safe state or not.

If the new state is in safe state then only this request is granted otherwise not.

To define the new state of the system because of the arrival of request of P1 we follow the Resource-Request algorithm which results as-

Process	Allocation A B C	Max A B C	Available A B C
P0	**0 1 0**	**7 5 3**	**3 3 2**
P1	**3 0 2**	**3 2 2**	
P2	**3 0 2**	**9 0 2**	
P3	**2 1 1**	**2 2 2**	
P4	**0 0 2**	**4 3 3**	

We must determine whether this new system state is safe. To do so, we again execute our safety algorithm and find the safe sequence as < P1, P3, P4, P0, P2 > which satisfies our safety requirements. Hence we can immediately grant the request for Process P1.

(d) The request for (3,3,0) by P4 cannot be granted because
Request= (3, 3, 0)
Available= (2, 3, 0)

In this situation the condition Request< available is false

So it is not granted since resources are not available.

(e) The request for (0,2,0) by P0
Request= (0, 2, 0)
Available= (2, 3, 0)

In this situation the condition Request< available is true

So it may be granted. If it is granted then the new state of the system is defined as

Available = Available – Request

= (2, 3, 0) – (0, 2, 0)

= (2, 1, 0)

Allocation = Allocation + Request

= (0, 1, 0) + (0, 2, 0)

= (0, 3, 0)

Need = Need - Request

= (7, 4, 3) + (0, 2, 0)

= (7, 2, 3)

Process	Allocation A B C	Max A B C	Available A B C
P0	**0 3 0**	**7 2 3**	**2 1 0**
P1	**3 0 2**	**0 2 0**	
P2	**3 0 2**	**6 0 0**	
P3	**2 1 1**	**0 1 1**	
P4	**0 0 2**	**4 3 1**	

Applying safety algorithm on this new state-

All five processes are in waiting state as none is able to satisfy the condition

Needi ≤ Available

So the state represents an unsafe state.

So request for P0 is not granted through the resources are available, but the resulting state is unsafe.

Deadlock Detection- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to remove the deadlock is applied either to a system which pertains single in instance each resource type or a system which pertains 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 the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

Several Instances of a Resource type- The wait-for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. The algorithms used are

- **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, is requesting k more instances of resource type Rj

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

Recovery from Deadlock- When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination- To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes-** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

- **Abort one process at a time until the deadlock cycle is eliminated-** This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

- **Resource Preemption-** To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed.

- **Selecting a victim-** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the numbers of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

- **Rollback-** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must rollback the process to some safe state, and restart it from that state. •

- **Starvation-** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a small finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

References:

(1) Abraham Silberschatz, Galvin & Gagne, Operating System Concepts, John Wiley & Sons, INC.

(2) Harvay M.Deital, Introduction to Operating System, Addition Wesley Publication Company.

(3) Andrew S.Tanenbaum, Operating System Design and Implementation, PHI

(4) Vijay Shukla, Operating System, S.K. Kataria & Sons