Unit 4 : Deadlock

Deadlock is a potential problem in any operating system. It occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again.

Deadlock occurs in many guises in the real world, from processing documents in an office to automobile traffic patterns in large cities. In some cases, deadlock may be purposely defined into a system so that a human operator of the system can intervene to resolve a conflict that is too difficult for the system to solve; this is common in office procedures. In this unit, different methods of handling problems like deadlock modeling, deadlock prevention, deadlock avoidance and deadlock detection will be discussed.

Lesson 1 : Introduction to Deadlock

1.1. Learning Objectives

On completion of this lesson you will be able to know:

- deadlock
- the conditions for deadlock
- various types of resources an OS manages.

1.2. Resources

Before discussing deadlock, we have to know what a resource is. OS is basically a resource manger. Deadlock can occur when processes have exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as resources. A resource can be hardware device (e.g., a tape drive memory) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can only be used by a single process at any instant of time.

Resources come in the following types :

Preemptible Resources

Preemptible Resources

Resources the OS can remove from a process (before it is completed) and give to another process.

Non-preemptible Resources

Resources a process must hold from when they are allocated to when they are released.

Shared Resource

A resource which may be used by many processes simultaneously.

Dedicated Resource

A resource that belongs to one process only.

In general, deadlock involves non-preemptive resources.

Under the normal mode of operation, the sequence of events required to use a resource is :

1. ***Request :*** If the request cannot be immediately granted (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. ***Use :*** The process can operate on the resource (for example, if the resource is a line printer, the process can print on the printer).

3. ***Release :*** The process releases the resource.

A process must request a resource before using it and release the resources after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total available in the system. In other words, a process cannot request three printer if the system only has two.

1.3. Deadlock

Memory is a resource, a tape drive is a resource and access to a shared variable is a resource.

Deadlock

Computer systems are full of resources that can only be used by one process at a time Having two processes simultaneously writing to the printer leads to gibberish. All operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

Deadlock problems may be a part of our daily environment. Consider the problem of crossing a river that has a number of stepping stones (Fig. 4.1). At most one foot can be on each stepping stone at a time. To cross the river, a person must use each of the stepping stones. We can view each person crossing the river as a process and

each stepping stone as a resource. A deadlock occurs when two people start crossing the river from opposite sides and meet in the middle (Fig. 4.2).

Stepping on a stone can be viewed as acquiring the resource, while removing the foot corresponds to releasing the resource. A deadlock occurs when two people try to step on the same stone. The deadlock can be resolved if either person retreats to the side of the river from which they started. In operating system terms, this retreat is called a rollback.

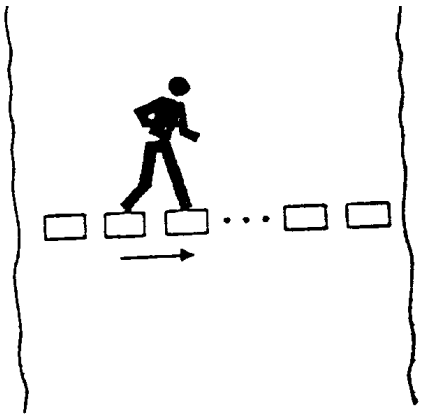

Fig. 4.1 : River crossing.

The only way to ensure that a deadlock will not occur is to require each person crossing the river to follow an agreed-upon protocol. One such protocol would require each person who wants to cross the river to find out whether someone else is crossing from the other side.

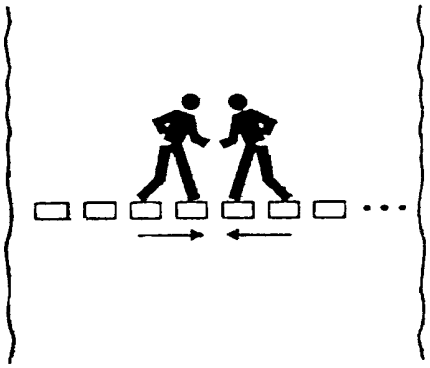

Fig. 4.2 : Deadlock situation in river crossing.

A set of process is in a deadlock state when every process in the set is waiting for an event that can be caused by another process in the set.

1.3.1. Deadlock Definition

The deadlock can be defined formally as follows *: A set of process is in a deadlock state when every process in the set is waiting for an event that can be caused by another process in the set.* The events with which we are mainly concerned here are resource acquisition and release.

To illustrate a deadlock state, consider a system with three tape drives. Suppose that there are three processes, each holding one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released", which can only be caused by one of the other waiting processes. This example illustrates a deadlock involving processes completing for the same resource type.

In other words deadlock is a state where one or more processes are blocked, all waiting on some event that will never occur.

For Example,
There are two process P1 and P2 and two unshakeable resources R1 and R2. Given the following code for process P1 and P2.

P1	**P2**
lock R1	lock R2
lock R2	lock R1
calculate	calculate

It is possible for the processes to be deadlocked if P1 has locked R1 and P2 has locked R2. Now process one attempts to execute its second instruction and obtain a lock on R2. However it must block the only copy of R2 is locked by P2.

P2 can now execute and it will attempt to lock R1. However it can't! It must block waiting on P1 to release R1.

Neither process can progress until event they are waiting for occurs. P1 is waiting for P2 to release R2 and P2 is waiting for P1 to release R1. Neither event will ever happen. These processes are deadlocked.

1.3.2. Conditions for Deadlock

Conditions for Deadlock

Coffman et. al. (1971) showed that a deadlock situation can arise if and only if the following four conditions hold simultaneously in a system.

Mutual Exclusion Condition

There exist shared resources that are used in a mutually exclusive manner. At least one resource is held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and Wait Condition

Processes hold onto resources they already have while waiting for the allocation of other resources. For example, a process obtaining a lock holds onto that lock while asking for another.

No preemption condition

Resources cannot be preempted; i.e. can only be resources cannot be removed from a process until hat process releases.

Circular wait

A circular chain of processes exists in which each process holds resources wanted by the next process in the chain.

We can see these four conditions in our river-crossing example. A deadlock occurs if and only if two people from opposite sides of the river meet in the middle. The mutual-exclusion condition obviously holds, since at most one person can be stepping on a stone at one time. The hold-and-wait condition is satisfied, since each person is stepping on one stone and waiting to step on the next one. The no-preemption condition holds, since a stepping stone cannot be forcibly removed from the person stepping on it. Finally, the circular-wait condition holds, since the person coming from the east is waiting on the person coming from the west, while the person coming from the west is waiting on the person coming from the east. Neither of the two intern can proceed, and each is waiting for the other to remove their foot from one of the stepping stones.

Deadlock can be handled in a number of ways by an operating system :

- it can be ignored,
- it can be prevented by denying one of the above necessary conditions,
- it can be avoided,
- or it can be detected and recovered from.

1.4. Exercise

1.4.1. Multiple choice questions

1. Which is true for dedicated resource?

i) A resource that belongs to one process only
ii) A resource which may be used by many processes simultaneously
iii) A resource which is given to another resource
iv) None of the above.

1.4.2. Questions for short answers

a) Explain what deadlock is? Give an example of deadlock.
b) How can deadlock be handled by an operating system?
c) What do understood by a resource?
d) What are the various types of resources an OS manages?
e) Describe the sequence of events required to use a resource.
f) List some examples of deadlocks which are not related to a computer system environment.

1.4.3. Analytical questions

a) What are the necessary conditions for deadlock?
b) Prove that the four conditions for deadlock must hold in river crossing example.

Lesson 2 : Deadlock Modeling

2.1. Learning Objectives

On completion of this lesson you will be able to know:

- resource allocation graph
- the methods for preventing deadlock.

2.2. Deadlock Modeling

Deadlock Modeling

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices is partitioned into two types P = {p_1, p_2, ..., p_n}, the set consisting of all the processes in the system, and R = {r_1, r_2, ..., r_m}, the set consisting of all resource types in the system.

An edge (p_i, r_j) is called a request edge, while an edge (r_j, p_i) is called and assignment edge.

Pictorially, we represent each process, p_i as a circle and each resource type, r_j as a square. Since resource type r_j may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge only points to the square r_j, while an assignment edge must also designate one of the dots in the square.

When process p_i requests an instance of resource type r_j, a request edge is inserted in the resource allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process later releases the resource, the assignment edge is deleted.

The resource allocation graph in Fig. 4.3 depicts the following situation.

- The sets P, R and E :

$P = \{p_1, p_2, p_3\}$
$R = \{r_1, r_2, r_3, r_4\}$
$E = \{p_1, r_1), (p_2, r_3), (r_1, p_2), (r_2, p_2), (r_2, p_1), r_3, p_3)\}$

- Resource instances :
  - ◊ One instance of resource type r_1.
  - ◊ Two instances of resource type r_2.
  - ◊ One instances of resource type r_3.
  - ◊ Two instances of resources type r_4.

- Process states :
  - ◊ Process p_1 is holding and instance of resource type r_2 and is waiting for an instance of resource type r_1.

- ◊ Process p_2 is holding an instance of r_1 and r_2 and is waiting for an instance of resource type r_3.
- ◊ Process p_3 is holding an instance of r_3.

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

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involve only a set of resource types, each of which have only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. So, a cycle in the graph is both a necessary and sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, let us return to the resource allocation graph depicted in (Fig. 4.3). Suppose that process p_3 requests an instance of resource type r_2. Since no resource instance is available, a request edge (p_3, r_2) is added to the graph (Fig. 4.4). At this point two minimal cycles exist in the system.

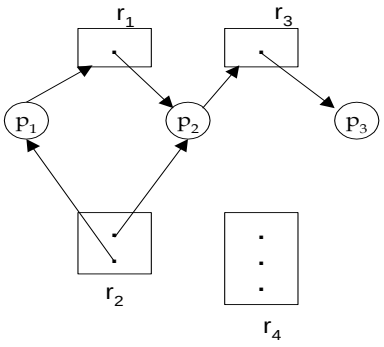

Fig. 4.3 : Resource allocation graph.

$p_1 \rightarrow r_1 \rightarrow p_2 \rightarrow r_3 \rightarrow p_3 \rightarrow r_2 \rightarrow p_1$

$p_2 \rightarrow r_3 \rightarrow p_3 \rightarrow r_2 \rightarrow p_2$

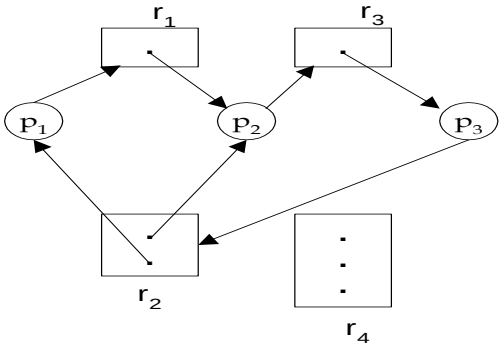

Fig. 4.4 : Resource allocation graph with a deadlock.

Processes, p_1, p_2 and p_3 are deadlock. Process p_2 is waiting for the resource r_3, which is held by process p_3. Process p_3 is waiting for either process p_1 or p_2 to release r_2. Meanwhile, process p_2 waiting on process p_3. In is Process p_1 is waiting for process p_2 to release resource r_1.

Now consider Fig.4.5. In this example, we also have a cycle.

$p_1 \rightarrow r_1 \rightarrow p_3 \rightarrow r_2 \rightarrow p_1$

However, there is no deadlock. Observe that process p_4 may release its instance of resource type r_2. That resource can be allocated to p_3 breaking the cycle.

To summarize, if a resource allocation graph does not have a cycle then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important in dealing with the deadlock problem.

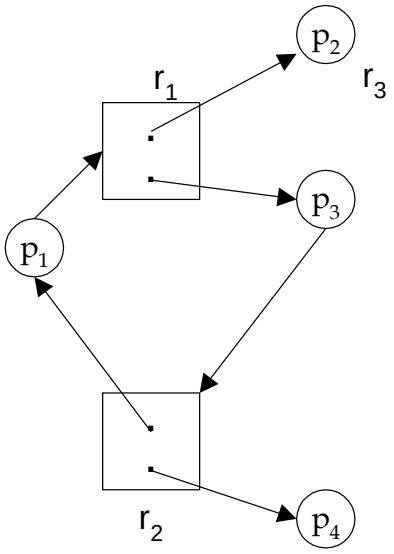

Fig. 4.5 : Resource allocation graph with a cycle but no deadlock.

2.3. Deadlock Prevention

We know that for the occurrence of a deadlock, each of the four necessary conditions must hold. By preventing on of the 4 necessary conditions, we can prevent the occurrence of a deadlock. Let's look on this approach by examining each of the four necessary conditions.

2.3.1. Mutual Exclusion Condition

A printer cannot be simultaneously shared by several processes and sharable resources on the other hand, do not require mutually exclusive access, cannot be

involved in a deadlock. Read only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. So, it is not possible to prevent deadlocks by denying the mutual-exclusion condition.

2.3.2. Hold and Wait Condition

If we can prevent processes that hold resources from waiting for more resources we can eliminate deadlock. One way to achieve this goal is to require all processes to request all their resources before staring execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

Another way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

There are two main disadvantages to these protocols. *First* resource utilization may be very low, since many of the resources may be allocated but unused for a long period of time. In the example given below, will make it clear. A process that reads data from an input tape, analyzes it for an hour and then writes an output tape as well as plotting the results. If all the resources must be requested in advance, the process will up the output tape drive tie and the plotter for an hour.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely while at least one of the resources that it needs is always allocated to some other process.

advantage

It is easy to code.

2.3.3. No Preemption Condition

If we can severe that no preemption does not hold, we can eliminate deadlocks. The protocol is as follows :

If a process that is holding some resources requests another resource that cannot be immediately allocated to it (that is the process must wait) then all resources currently being held are preempted. That is, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will only be restarted when it can regain its old resources, as well as the new ones that it is requesting.

2.3.4. Circular wait Conditions

By preventing the circular wait from occurring, we can eliminate deadlock. For this, resources are uniquely numbered (Fig. 4.6(a)) and processes can only request resources in linear ascending order.

Now the protocol is this : processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request a first card reader and then a tape drive, but it may not request first a printer and then disk drive.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 4.6(b). We can get a deadlock only if C request resource j and D requests resource i. Assuming i and j are distinct resources, they will have different numbers. If i ➤ j the C is not allowed to request j. If i ◄ j then C is not allowed to request i. Thus, dead look is impossible.

1. card reader
2. disk drive
3. tape drive
4. printer.

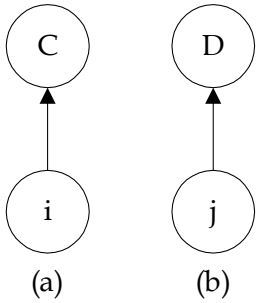

Fig. 4.6 : (a) Numerically ordered resources (b) A resource graph.

2.4. Exercise

2.4.1. Questions for short answers

a) What do you understood by resource allocation graph?
b) What is a request edge and assignment edge?

2.4.2. Analytical questions

1. Explain that
a) a cycle in the graph is both a necessary and sufficient condition for existence of deadlock.
b) a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
2. What do you know about deadlock prevention?

Lesson 3 : Deadlock Avoidance

3.1. Learning Objectives

On completion of this lesson you will be able to :

- explain deadlock avoidance
- define safe state and unsafe state
- know banker's algorithm.

3.2. Deadlock Avoidance

Avoidance strategies are designed to allocate only when it is certain that deadlock can not occur as a result of the allocation. The basic observation behind avoidance strategies is that the set of all states can be partitioned into safe states and unsafe state(states that are not safe).

3.2.1. Safe State and Unsafe State

The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish, whereas from an unsafe state, no such guarantee can be given.

A safe state is one in which it can be determined that the system can allocate resources to each process (up to its maximum) in some order (for all pending requests) and still avoid deadlock. A safe state is not a deadlock state. A deadlock state is an unsafe state. Not all unsafe states are deadlocks but an unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (deadlock) states. Process that are holding resources in an unsafe state may subsequently release them rather than ask for more; the system would transition from an unsafe state back into a safe state. The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish, whereas from an unsafe state, no such guarantee can be given. It will be more clear to illustrate safe state and unsafe state by an example.

Suppose a system has 12 disk drive and three processes : P_0, P_1 and P_2. Process P_0 require ten disk drives, process P_1 may need four, and process P_2 may need up to nine disk drives. At time T_0 process P_0 is holding five disk drives, process P_1 is holding two, and process P_2 is holding two. So, there are three free disk drives (Fig. 4.7(a)).

Process	Hold	Max
P_0	5	10
P_1	2	4
P_2	2	9

Free : 3

(a)

Process	Hold	Max
P_0	5	10
P_1	4	4
P_2	2	9

Free :1

(b

Process	Hold	Max
P_0	5	10
P_1	0	-
P_2	9	2

Free: 5

(c)

Process	Hold	Max

Process	Hold	Max

Process	Hold	Max

P_0	10	10
P_1	0	-
P_2	2	9

Free : 0
(d)

P_0	0	-
P_1	0	-
P_2	2	9

Free : 10
(e)

P_0	0	-
P_1	0	-
P_2	9	9

Free : 3
(f)

Process	Hold	Max
P_0	0	-
P_1	0	-
P_2	0	-

Free : 12
(g)

Fig. 4.7 : Demonstration that the state is safe.

The system is in a safe state. The sequence <p_1, p_0, p_2> satisfies the safety condition, since process p_1 can immediately be allocated all of its disk drives (Fig. 4.7(b)) and return them and the system will then have five available disk drives (Fig. 4.7(c)). Process p_0 can get all of its disk drives and return them, the system will then have ten available disk drives (Fig. 4.7(e)) and finally process p_2 could get all of its disk drives and return them. The system will then have all twelve disk drives available (Fig. 4.7(g)).

Let's look how to go from a safe state to an unsafe state. At time t_1, process p_2 requests and is allocated one more disk drive (Fig. 4.8(b)). The system is no longer in a safe state. At this point, only process p_1, can be allocated all of its disk drives. When it returns them, the system have only four available disk drives (Fig. 4.8(d)). Since process p_0 is allocated five tape drives, but has a maximum of ten, it may then request five more. Since they are unavailable, process p_0 must wait. Similarly, process p_2 may request an additional six tape drives and have to wait, resulting in a deadlock.

Process	Hold	Max
P_0	5	10
P_1	2	4
P_2	2	9

Free : 3
(a)

Process	Hold	Max
P_0	5	10
P_1	2	4
P_2	3	9

Free : 2
(b)

Process	Hold	Max
P_0	5	10
P_1	4	4
P_2	3	9

Free : 0
(c)

Process	Hold	Max
P_0	5	10
P_1	0	-
P_2	3	9

Free : 4
(d)

Fig. 4.8 : Proving that the state (b) is not safe.

Given the concept of a safe state, we can define avoidance algorithms which ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide if the resource can be immediately allocated or if the process must wait. The request is granted only if it leaves the system in a safe state.

3.3. Bankers Algorithm

Bankers algorithm

The scheduling algorithm for avoiding deadlock is known as Dijkstra's banker's algorithm. It is modeled after the lending policies often employed in banks. The algorithm could be used in a banking system to ensure that the bank never allocates its available cash in such a way that it can no longer satisfy the needs of all its customers.

At the time of entering a new process to the system, it must declare the needed maximum number of instances of each resource type may not exceed the total number of resources in the system. When a user requests a set of resources, it must be determined whether the allocation of these resources will leave the system in a safe state. If so, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

The following data structures must be maintained to implement the banker's algorithm.

3.3.1. Data Structures

Available : A vector of length l indicating the number of available resources of each type. If *Available [j] = k*, there are k instances of resources type r_j available.

Max : An l x m matrix defining he maximum demand of each process. If *Max (i,j) = k*, then process p_j may request at most k instances of resource type r_j, where m is the No. of process, l is the No. of resource types.

Allocation : An l x m matrix defining the number of resources of each type currently allocated to each process. If *Allocation [i,j] = k,* then process p_i is currently allocated k instances of resource type r_j.

Need : An l x m matrix indicating the remaining resource need of each process. If *Need (i,j) = k*, then process p_i may need k more instances of resource type r_j, in order to complete its task. Note that *Need(i,j) = Max(i,j) - Allocation(i,j).*

3.3.2. Algorithm

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

1. If $Request_r \leq Need_i$ then go to step 2. Otherwise error.

2. If $Request_i \leq Available$ then go to step 3. Otherwise the resources are not available, and p_i must wait.
3. Modify the state as follows.

$$Available := Available - Request_i$$
$$Allocation_i := Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

If the state is safe, the transaction is completed and process p_i is allocated its resources. However, if the new state is unsafe, then p_i must wait for $Request_i$ and the old resource allocation state is restored. The algorithm for finding out whether a system is in a safe state or not can be described as follows :

3.3.3. Safety Algorithm

1. Let *work* and *Finish* be vectors of length m and n, respectively. Initialize *Work :* = *Available* and *Finish [i] : = false* for i = 1, 2, ..., n.
2. Find an i such that :

   a. *Finish[i] = false,* and
   b. $Need_i \leq Work$.

If no such i exists, go to step 4.

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

4. If *Finish[i] = true* for all i, then the system is in a safe state.

Example,
Consider a system with five processes {p_o, p_1 ..., p_4} and three resource types. Resource type r_1 has 10 instances, resource type r_2 has 5 instances, and resource type r_3 has 7 instances. Suppose that at time T_0 the following snapshot of the system has been taken.

	Allocation	Max	Available
	r_1 r_2 r_3	r_1 r_2 r_3	r_1 r_2 r_3
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	

The content of the matrix *Need* is defined to be *Max - Allocation* and is :

	Need
	r_1 r_2 r_3
p_0	7 4 3

p_1	1 2 2
p_2	6 0 0
p_3	0 1 1
p_4	4 3 1

The system is currently in a safe state and the sequence $<p_1, p_3, p_4, p_2, p_0>$ satisfies the safety criteria.

Now the process p_1 requests one additional instance of resource type r_1 and two instances of resources of resource type r_2, so $Request_1 = (1, 0, 2)$. In order to decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ $(1, 0, 1) \leq (3, 2, 2)$ which is true. So, this request has been fulfilled and arrive at the following new state :

	Allocation	Need	Available
	$r_1\ r_2\ r_3$	$r_1\ r_2\ r_3$	$r_1\ r_2\ r_3$
p_0	0 1 0	7 5 3	2 3 0
p_1	3 0 2	0 2 0	
p_2	3 0 2	6 0 0	
p_3	2 1 1	0 1 1	
p_4	0 0 2	4 3 1	

For checking whether this new system state is safe, we have to execute our safety algorithm and find out that the sequence $<p_1, p_3, p_4, p_0, p_2>$ satisfies our safety requirement. So a request for (3, 3, 0) by p_4 cannot be granted since the resources are not available. A request for (0, 2, 0) by p_0 cannot be granted even though the resources are available, since the resulting state is unsafe.

3.4. Exercise

3.4.1. Questions for short answers

a) What do you understand by safe state?
b) What do you understand by unsafe state?
c) Describe safety algorithm.

3.4.2. Analytical questions

a) What do you understand by safe state and unsafe state? Illustrate with example.
b) Given a system that uses the banker's algorithm for avoiding deadlock and the resource state shown below.

	Allocation	Max	Available
p_0	0 0 1 2	0 0 1 2	1 5 2 0
p_1	1 0 0 0	17 5 0	
p_2	1 3 5 4	23 5 6	
p_3	0 6 3 2	0 6 5 2	
p_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm :

a) What is the content of the array *Need*?
b) Is the system in a *safe* state?
c) If a request from process p_1 arrives for (0, 4, 2, 0), immediately granted?

Lesson 4 : Deadlock Recovery

4.1. Learning Objectives

On completion of this lesson, you will be able to know:

- how to recover from deadlock
- the combined approach of handling deadlock.

4.2. Deadlock Recovery

Detection is only one part of he solution of deadlock problem. After detecting deadlock with the help of detection algorithm, what should be done next. There should be some way in the system needed to recover from deadlock and get the system going again. There are two ways for breaking a deadlock :

- Recovery through killing process.
- Recover through resource preemption.

4.2.1. Recovery Through Killing Process

For eliminating the deadlock by killing a process the following two methods can be used:

- Kill all Deadlock Processes
- Kill one process at a time until the deadlock cycle is eliminated.

Kill all Deadlock Processes

By killing all deadlocked processes will break the deadlock cycle. These processes many have computed for long time and result of these partial computation must be discarded and can be re-computed later.

Kill one process at a time until the deadlock cycle is eliminated

After each process is killed, the other processes will be able to continue and a deadlock detection algorithm must be involved to determine whether any processes are still deadlocked.

For partial computation, we have to determine which process should be terminated to try to break the deadlock. Many factors are related to determine which process is chosen.

- Priority of the process.
- Computing time.
- Number and type of resources used by process.
- Number of extra resources for completion.
- Number of process involved in the deadlock.

It is best to kill a process that can be rerun. From the beginning with no ill effects. As for example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed part way through, the first run has no inference on the second run.

4.2.2. Recovery Through Preemption

For eliminating deadlock using resource preemption, we can preempt resources from processes and give these resources to other processes until the deadlock cycle is broken.

The following issues are related to preemption : ***Selection of a Victim*** and ***Rollback***

Selection of a Victim

We have to determine the resources and processes to be preempted and determine the order of preemption cost. Consider deadlock situation involving two people A and B.

Case 1: A has a higher priority than B. So, B will have to back up.

Case 2: A needs only 2 more stepping stones to cross the river (i.e. A has already used 98 stepping stones). So, B have to back up.

Case 3: A and B deadlock in the middle of the river. There are ten people behind A. So it would be more reasonable to require B have to back up. Otherwise, eleven people will have to back up.

Rollback

The solution of a rollback is to abort the process and then restart it. Let us go through river crossing example. If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. So, We have to roll the process back to some safe state and restart it from that state.

An effective solution of rollback is to place several additional stepping stones in the river, so that one of the people involved in the deadlock may step aside to break the deadlock.

4.3. Combined Approach

It has been argued that none of the presented approaches is suitable for use as an exclusive method of handling deadlocks in a complex system. The presented approaches i.e. deadlock prevention, avoidance and detection can be combined for maximum effectiveness. The proposed method is based on the notion that resources can be partitioned into classes that are hierarchically ordered. A resource ordering technique can be applied to the classes. Within each class the most appropriate technique for handling deadlock is used.

Consider a system that consists of the four classes of resources described below.

Classes of resources

- *Swapping space*, an area of secondary storage designated for backing up blocks of main memory.
- *Job resources and assignable devices*; such as printers and drives with removable media (e.g., tapes, cartridge disks, floppies)
- *Main memory* assignable on a block basis, such as pages or segments.
- *Internal resources*, such as I/O channels and slots of the pool of dynamic memory.

The basic idea is to provide deadlock prevention between the four classes of resources by linear ordering of requests in the presented order. The following strategies may be applied to individual resource classes.

Swapping Space **:** Prevention of deadlocks by means of advanced acquisition of all needed space is a possibility for the swapping space. Deadlock avoidance is also possible, but deadlock detection is not, since there is no backup of the swapping store.

Strategies of resources classes.

Job Resources **:** Avoidance of deadlock is facilitated by the proclaiming of resource requirements which is customarily done for jobs by means of the job-control statements. Deadlock prevention through resource ordering is also a probability, but detection and recovery are undesirable due to the possibility of modification of files that belong to this class of resources.

Main Memory **:** With swapping, prevention through preemption is a reasonable choice. That allows support for run-time growth (and shrinking) of memory allocated to resident processes. Avoidance is undesirable because of its run-time overhead and tendency to underutilized resources. Deadlock detection is possible but undesirable due to either the return-time overhead for frequent detection or the unused memory held by deadlocked processes.

Internal System Resources **:** Due to frequent requests and releases of resources and the resulting frequent changes of state, the run-time overhead of deadlock avoidance and detection can hardly be tolerated. Prevention through resource ordering is probably the best choice.

4.4. Exercises

4.4.1. Multiple choice questions

1. How many ways for breaking a deadlock?

i) 2
ii) 3
iii) 4
iv) None of the above.

2. Killing all deadlocked processes

i) Will not break the deadlock cycle.
ii) Will break the deadlock cycle .
iii) Cause the deadlock problem.
iv) None of the above.

4.4.2. Questions for short answers

a) List the factors for choosing the process.
b) What do you understood by rollback? What is the effective solution to rollback?

4.4.3. Analytical questions

a) Describe the different methods of deadlock recovery.
b) Why combined approach for handling deadlock is used? Describe combined approach.