

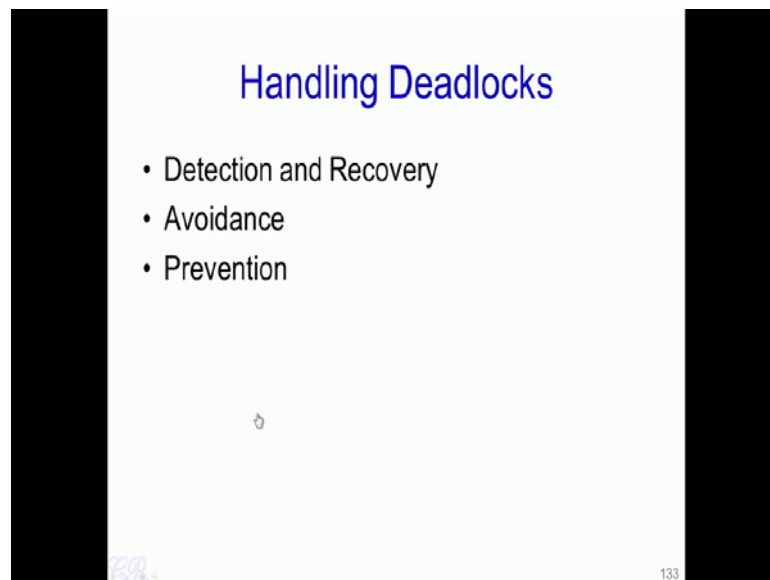
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 32
Dealing with Deadlocks

There are 3 ways to handle Deadlocks. That is by Deduction and Recovery from Deadlocks, Deadlock Avoidance and Deadlock Prevention.

In this video, we will have a look at Deadlock Avoidance and Prevention. By avoidance we mean that, the system will never go into a state which could potentially create a deadlock situation to have an example about how avoidance algorithms work we will just take very simple example.

(Refer Slide Time: 00:53)



(Refer Slide Time: 00:56)

Deadlock Avoidance

- Basic idea, always make the right choice!

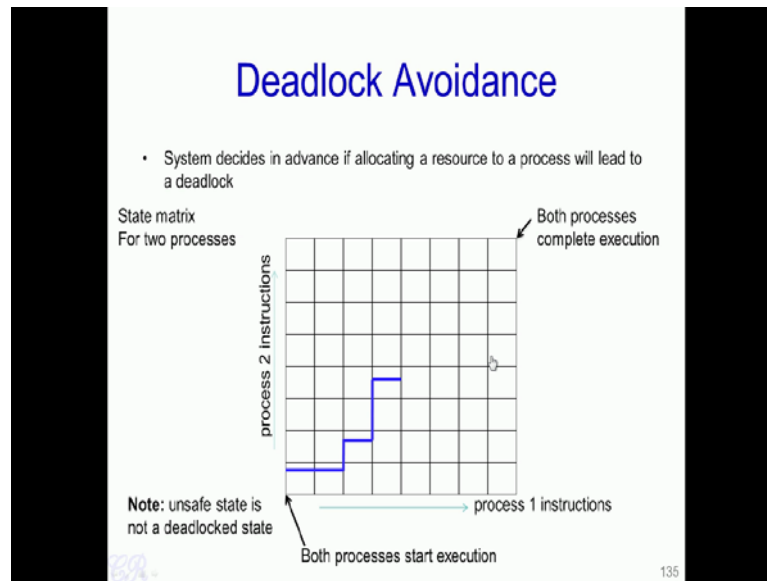


134

So, let say we want to go from this point A, this is a place to this point B and as usual there are multiple routes which could take us there. So, this is one particular route and this is another route. Now we need to make a choice or rather the right choice that will take us to this particular place B in a safe way. For example, if we may go through this route it may be dangerous or may have some particular blocks on the road.

On the other hand going through a third way may have a similar situation. So, out of the multiple ways which we can go from one point to another we need to make a choice of which path we need to go. Deadlock avoidance algorithms works in very similar ways. Let us take a particular example with two processes.

(Refer Slide Time: 02:09)



Process 1 and process 2, and we will represent this state of the system by this particular graph. Now on the x axis, are the instructions executed by process 1 and the y axis corresponds to the instructions executed by process 2, with respect to time; so both are to with respect to time. Thus, process 1 and process 2 begin to execute from the origin that is this point over here and they complete execution at the other end that is the point over here. Now, as this process executes in the system the state of the system will change, and the state changes depending on how these two processes execute with respect to each other.

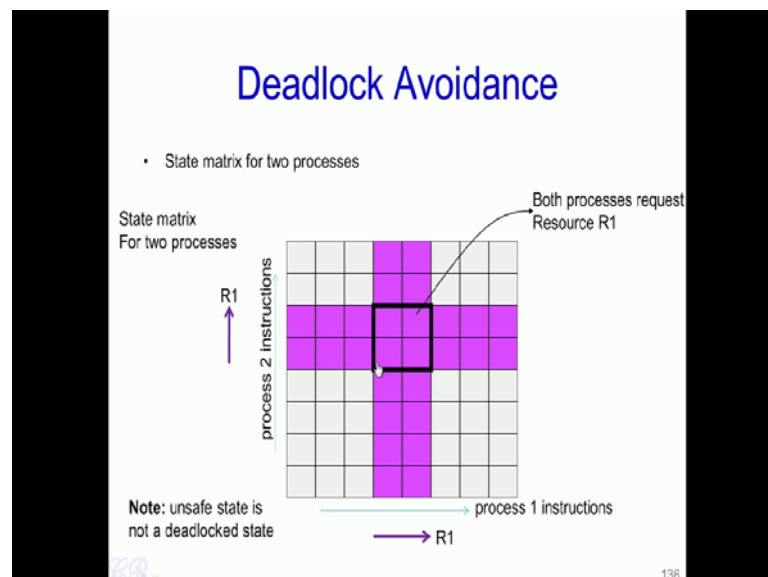
Let us say this state is represented by each of these blocks over here. Now, this line over here indicates the execution path for example, over here the line is horizontal indicating that instructions corresponding to process 1 gets executed. So, as a result of process 1 executing while process 2 not executing, we get a state of the system over here which is shown by this small square, then the state is shifted to this as shown over here and then at this particular point there is a context switch and process 2 executes.

As a result of process 2 being executed, the state of the system gets shifted to this point. Then process 1 executes again and the state changes, then process 2 gets executed process 1 and so on. And at the end we will have both process 1 as well as process 2

reaching this particular point; essentially both processes have completed execution.

Now, you see depending on how the operating system schedules, how various resources are allocated or de-allocated to process 1 and process 2. This particular trace or rather the execution trace of the process 1 with respect to process 2 would vary. For example, in the worse it could be the process 1 executes continuously did it completes and then process 2 executes completely till it completes or another technique could be the process 1 executes a few instructions, then process 2 executes the few instructions, then process 1, process 2, process 1 and so on, till both of them complete.

(Refer Slide Time: 05:08)

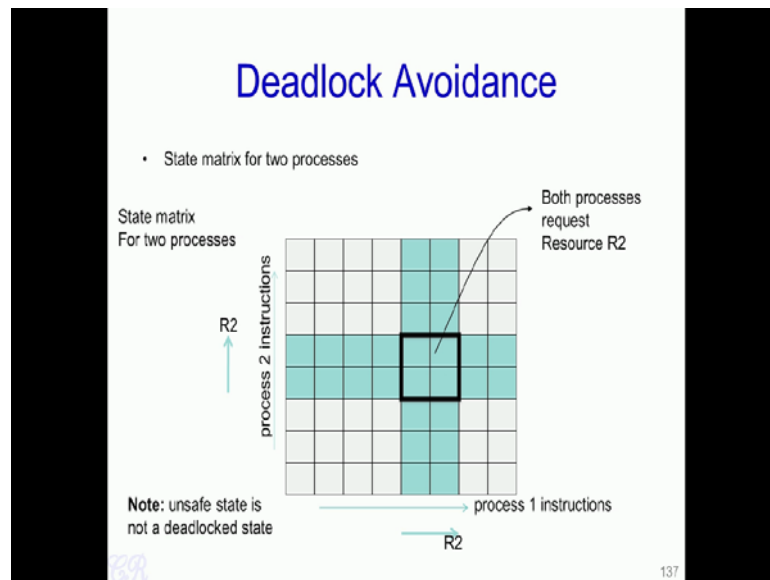


Now, let us also state that process 1 and process 2 requires a resource R1 during its execution. So, process 1 requires the resource R1 at this particular time. So, in this particular time over here which is shown at this point. So, these instructions in process 1 require the use of resource R1. So, this has been highlighted by this purple block over here.

Now, in a similar way during this time from this point to this point, process 2 requires the use of resource 1, so this has been I highlighted over here. Now this intersecting square is the region where both process 1 as well as process 2, require resource 1. So, this

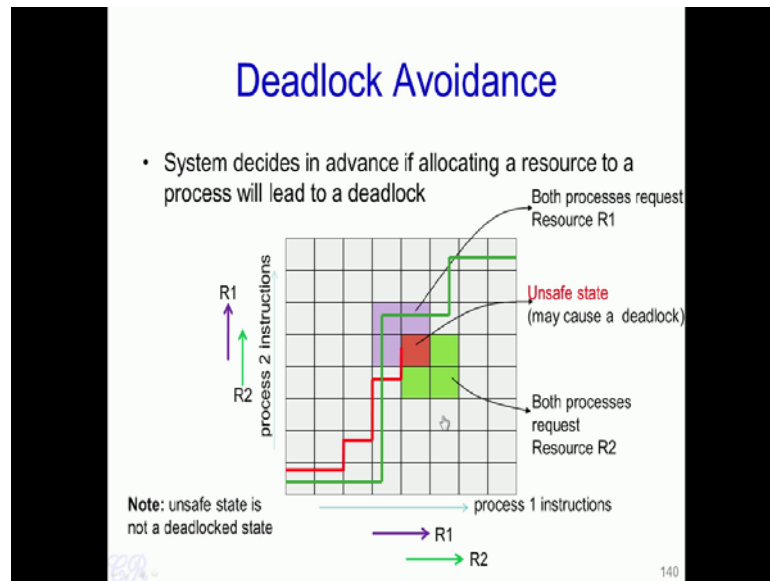
intersecting square represented by the intersection of these two regions corresponds to the instructions in both processes that request resource R1.

(Refer Slide Time: 06:15)



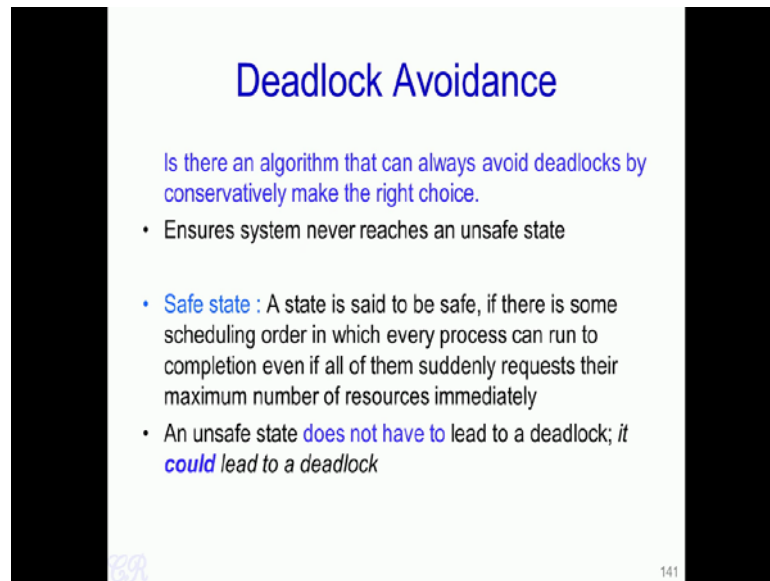
Similarly, let us say that there is another resource R2 which is required in these particular instructions by process 2 and these particular instructions in process 1, and correspondingly obtained intersecting square as shown here and this square represents the part where both processes p 1 and p 2 request resource R2. Now, what we will see is the intersecting part of R1 and R2 resources. So, if we actually intersect this graph with this graph, we get the points where both processes request R1 that is this purple part here, and this part is where the both processes require R2, and the intersecting area is where both processes require both resources R1 and R2.

(Refer Slide Time: 06:58)



Now, this area is which may potentially cause a deadlock. So, we call this as an unsafe state. So, an unsafe state is not a deadlock state, but rather it is a state which may potentially cause a deadlock. Thus, if the OS was to schedule the process 1 and process 2 in such a way that the execution path ends up in this unsafe state then, we may potentially have a deadlock. On the other hand, if the OS schedules process 1 and process 2 execution in such a way that this unsafe state is avoided then the deadlock is avoided. So, this is the essential technique used to avoid deadlocks in systems.

(Refer Slide Time: 08:33)



Deadlock Avoidance

Is there an algorithm that can always avoid deadlocks by conservatively make the right choice.

- Ensures system never reaches an unsafe state
- **Safe state** : A state is said to be safe, if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources immediately
- An unsafe state **does not have to** lead to a deadlock; it **could** lead to a deadlock

141

The next question is - is there an algorithm that can always avoid deadlocks by conservatively making the right choice. That is can be ensure that the system never reaches an unsafe state by making some choices in the way processes execute or the way resources are allocated and so on.

So, one way that we can build such an algorithm is by using what is known as the Banker's algorithm. So, let us see an example with this.


(Refer Slide Time: 09:12)

Example with a Banker

- Consider a banker with 3 clients (A, B, C).
 - Each client has certain credit limits (totaling 20 units)
 - The banker knows that max credits will not be used at once, so he keeps only 10 units

	Has	Max
A	3	9
B	2	4
C	2	7

Total : 10 units
free : 3 units



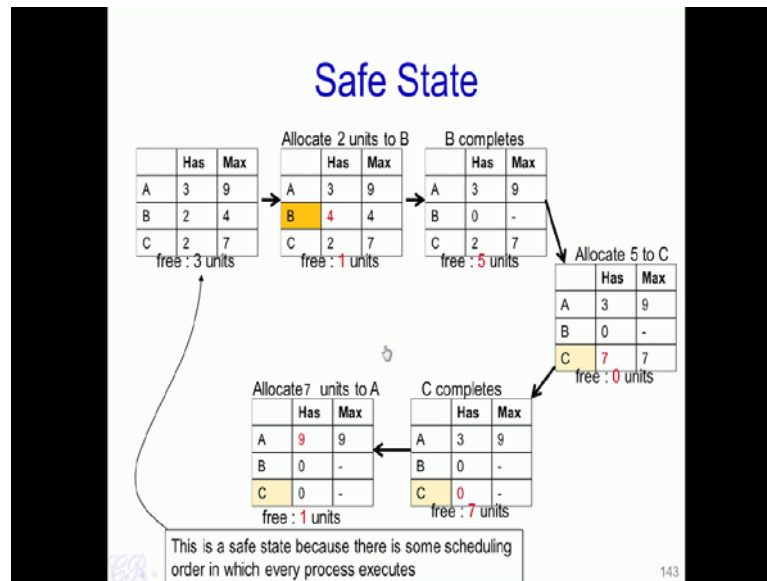
- Clients declare **maximum** credits in advance. The banker can allocate credits provided no unsafe state is reached.

142

Now, consider a banker with 3 clients A, B and C. Each client has certain credit limits and the sum of all these credit limits totals to 20 units. Now the banker knows that the maximum credits of 20 units will not be used all at once. So, he keeps only 10 units to see what this means let us say we consider these table with the 3 clients A, B and C and each client has in his account some units. So, for instance A has got 3 units, B has got 2 units and C has got 2 units, So the total of 4 5 6 7 and since the banker only keeps 10 units, so the remaining 3 units are free and not allocated to any of the clients.

On the other hand, the maximum credits for each client is 9, 4 and 7 respectively. So, you note that 9 plus 4 plus 7 is the maximum credit limits which totals to 20 units. And this maximum credit limits is declared by the client in advance. Now, given this particular table the banker should be able to allocate the maximum credits to users in such a way that no unsafe state is reached. So, let us see this with examples.

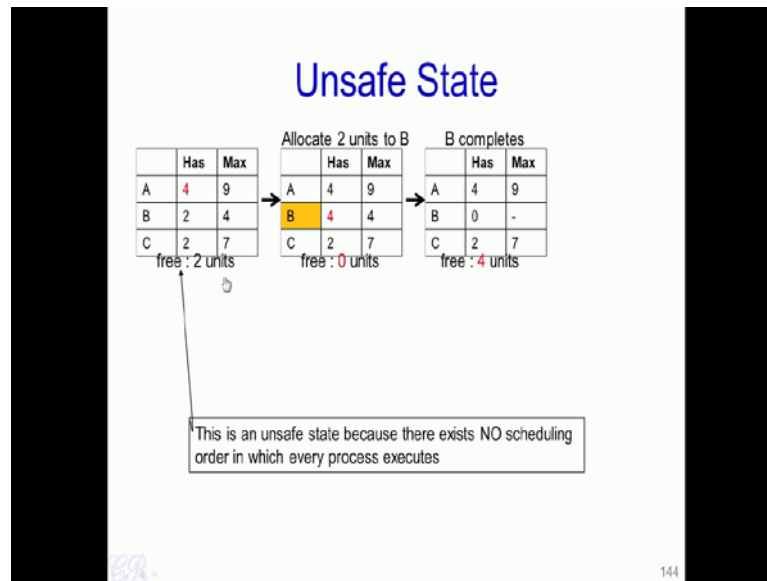
(Refer Slide Time: 11:07)



So, the problem we are trying to solve here is whether these allocations maximum of 9, 4 and 7 can be made to A B and C in such a way that this state is safe or rather such an allocation can be made. So, let say we start with two units being allocated to B. So, we have 3 units in the free store, out of these 3 units - 2 are allocated to B, therefore B has now 2 plus 2 that is 4 units, and the free store now has 3 minus 2 that is 1 unit free. Now, after B completes all the 4 units are returned to the free store, so 4 plus 1 will be 5 units present in the free store and these 5 units can then be allocated to C. So, C will get it is maximum credit limit of 5 plus 2 that is 7 units as seen over here. So, the amount present in the free store is 0 units.

Now, after C completes all the 7 units of C get returned back to the free store. Now the banker could allocate the resources required by A that is the maximum resources required by A that is of 7 more units to get it is maximum credit limit of 9 units. So, we have 9 units are allocated to A and 1 unit which is free. Thus we see with this particular start point or this particular state of the system where there are 3 units free and there is maximum of 9, 4 and 7 units as the credit limit of each client can have. We see that with this particular state there is a schedule which is possible so that all requests and all maximum requests by the clients can be fulfilled. So, we call such a state as a Safe State and the banker could go ahead and make such allocations.

(Refer Slide Time: 13:35)



Now, let us look at another example where the state we start with is that A has 4 units with it, B has 2 and C has 2, so a total of 8 units are present with A, B and C respectively. And the number of free units is 2 and the maximum credit limit of each client is as before 9, 4 and 7 respectively. Now, we see that irrespective of how the banker tries to allocate these remaining units to each of its clients, there is no schedule which is feasible that could fulfill these requirements.

For example suppose B were allocated its 2 more units and you see that given that there are only 2 units free, so B is the only client which can be serviced therefore, B gets 4 units while there are 0 free units present. Now after B completes we have 4 units free, but you see that neither A which requires 5 more units, nor C which requires 5 more units as well can be serviced with 4 units which is present in the free store. Therefore, we that this state is an Unsafe State because there is no scheduling order which can cater to these requirements. So, this is an unsafe state and the banker should ensure that such a state is not reached at any point.

(Refer Slide Time: 15:33)

Banker's Algorithm (with a single resource)

When a request occurs

- If(is_system_in_a_safe_state)
 - Grant request
- else
 - postpone until later

Deadlock	unsafe
	safe

Please read Banker's Algorithm with multiple resources from
Modern Operating Systems, Tanenbaum

145

A similar mechanism can be also implemented in the operating system, where corresponding to each request the operating system will determine if the system is in a safe state, and if indeed the system is in a safe state indicating that there is a schedule of the resources so that all request can be granted, then the requests are allocated. Otherwise the request is postponed until later.

(Refer Slide Time: 16:08)

Deadlock Prevention

- Deadlock avoidance not practical, need to know maximum requests of a process
- Deadlock prevention
 - Prevent at-least one of the 4 conditions
 1. Mutual Exclusion
 2. Hold and wait
 3. No preemption
 4. Circular wait

146

The third way to prevent deadlocks is what is known as Deadlock Prevention. Essentially we prevent deadlocks by designing systems where one of the 4 conditions is not satisfied. So, we have seen these 4 conditions and these are the conditions which are essential for a deadlock to occur. By preventing one of these conditions from holding in the system we can therefore prevent deadlocks. For example, if we design a system where hold and wait condition cannot be satisfied, that is a process cannot hold a resource while waiting for another resource then such a system will not have a deadlock.

So, let us see various ways in which we can actually prevent one of these conditions from happening.

(Refer Slide Time: 17:10)

Prevention

- 1. Preventing Mutual Exclusion**
 - Not feasible in practice
 - But OS can ensure that resources are optimally allocated
- 2. Hold and wait**
 - One way is to achieve this is to require all processes to request resources before starting execution
 - May not lead to optimal usage
 - May not be feasible to know resource requirements
- 3. No preemption**
 - Pre-empt the resources, such as by virtualization of resources (eg. Printer spools)
- 4. Circular wait**
 - One way, process holding a resource cannot hold a resource and request for another one
 - Ordering requests in a sequential / hierarchical order.

147

Preventing Mutual Exclusion, so in practice this is not feasible we will not be able to always prevent mutual exclusion. For example, take the case of a printer resource. So, it always needs to work or print corresponding to a particular process. So, it cannot be simultaneously shared with two processes unless it has a spool present in it. However, this being said the operating system can ensure that the resources are optimally allocated.

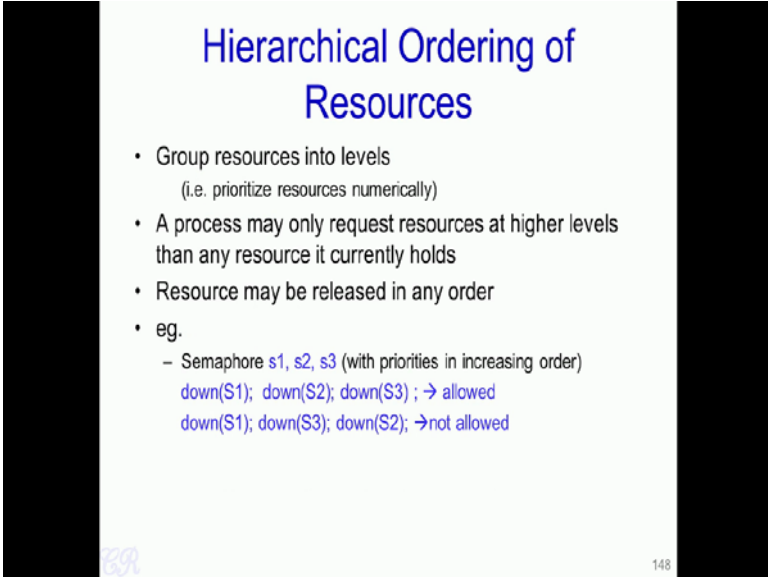
Let see the Hold and wait. So, one way to achieve this is to require all processes to request their resources before they start executing. So, this obviously is not an optimal

usage of the resources and also may not be feasible to know the resource requirements during the start of execution.

Now, let us look at the third way there is no preemption. So, Pre-empt the resources such as by virtualization of resources example by printer spools. So, in a printer spool we could have several processes sending documents to be printed at exactly the same time. So, all these documents are spooled or buffered in the printer and eventually each document is then printed one after the other.

The final technique we can target is the Circular wait. One way to prevent this is that process holding a resource cannot hold a resource and request for another one at the same time. Second way to prevent the circular wait is by ordering requests in a particular order - either sequential or hierarchical order.

(Refer Slide Time: 19:20)



Hierarchical Ordering of Resources

- Group resources into levels
(i.e. prioritize resources numerically)
- A process may only request resources at higher levels than any resource it currently holds
- Resource may be released in any order
- eg.
 - Semaphore s_1, s_2, s_3 (with priorities in increasing order)
 $\text{down}(S_1); \text{down}(S_2); \text{down}(S_3); \rightarrow$ allowed
 $\text{down}(S_1); \text{down}(S_3); \text{down}(S_2); \rightarrow$ not allowed

148

Let us look at the hierarchical order of resources. Here resources are grouped into a level that is they are prioritized by some numeric value. A process may only request resources at a higher level than any resource it currently holds, and resources may be released in any order. For example, let us say we have semaphores s_1, s_2 and s_3 with priorities in increasing order that is s_3 is the highest priority, while s_1 is the lowest priority. So, let us

say a process makes this particular sequence of semaphore requests S1, S2 and S3. So, we see that an ordering is followed that is S2 is greater than S1, and S3 is greater than S2. So, this kind of allocation should be allowed.

However, in this case where we have the first S1 then S3 and then S2, so in this particular case the allocation is not allowed because the ordering is not maintained.

Thank you.