Operating Systems (OS) Concepts - IT 308: Lecture 15
Deadlock avoidance--II

**Batch:** B.Tech III year

**Instructor:** Rahul Muthu

DA-IICT
Behaviour in an unsafe state

- As long as the current state is safe, the operating system can avoid unsafe states, and thereby deadlocks.
Behaviour in an unsafe state

- As long as the current state is safe, the operating system can avoid unsafe states, and thereby deadlocks.

- In an unsafe state a deadlock may or may not occur. The system cannot service resource requests and avoid deadlocks in an unsafe state.
Behaviour in an unsafe state

- As long as the current state is safe, the operating system can avoid unsafe states, and thereby deadlocks.
- In an unsafe state a deadlock may or may not occur. The system cannot service resource requests and avoid deadlocks in an unsafe state.
- In an unsafe state, the behaviour of the processes determines whether or not a deadlock occurs.
As long as the current state is safe, the operating system can avoid unsafe states, and thereby deadlocks.

In an unsafe state a deadlock may or may not occur. The system cannot service resource requests and avoid deadlocks in an unsafe state.

In an unsafe state, the behaviour of the processes determines whether or not a deadlock occurs.

It is possible to go from a safe state to an unsafe state. This may occur by the system following a resource allotment not conforming to a safe sequence.
Behaviour in an unsafe state

- As long as the current state is safe, the operating system can avoid unsafe states, and thereby deadlocks.
- In an unsafe state a deadlock may or may not occur. The system cannot service resource requests and avoid deadlocks in an unsafe state.
- In an unsafe state, the behaviour of the processes determines whether or not a deadlock occurs.
- It is possible to go from a safe state to an unsafe state. This may occur by the system following a resource allotment not conforming to a safe sequence.
- Allotment out of sequence is acceptable, if as a result the system continues to stay in a safe state.
The deadlock avoidance scheme works by starting in a safe state, and never making allocations which cause the system to enter an unsafe state.
The deadlock avoidance scheme works by starting in a safe state, and never making allocations which cause the system to enter an unsafe state.

In order to avoid entering an unsafe state, the system sometimes does not allocate a process a request, even if the resource is available.
In a system with only single instances of each resource-type, a variant of the resource-allocation graph can be used to design a deadlock avoidance algorithm.
In a system with only single instances of each resource-type, a variant of the resource-allocation graph can be used to design a deadlock avoidance algorithm.

In addition to the assignment and request edges there is a third class of edges called claim edges. A claim edge is directed from a process to a resource.
In a system with only single instances of each resource-type, a variant of the resource-allocation graph can be used to design a deadlock avoidance algorithm.

In addition to the assignment and request edges there is a third class of edges called *claim edges*. A claim edge is directed from a process to a resource.

A claim edge from $P_i$ to $R_j$ indicates that process $P_i$ may request resource $R_j$ in the future.
In a system with only single instances of each resource-type, a variant of the resource-allocation graph can be used to design a deadlock avoidance algorithm.

In addition to the assignment and request edges there is a third class of edges called \textit{claim edges}. A claim edge is directed from a process to a resource.

A claim edge from \( P_i \) to \( R_j \) indicates that process \( P_i \) may request resource \( R_j \) in the future.

A claim edge is like a request edge, because it is directed from a process to a resource. We distinguish the two by using a \textit{dashed line} for claim edges.
When a process requests a particular resource the claim edge is replaced by a request edge. Similarly, when a process releases a resource, the assignment edge is replaced by a claim edge.
When a process requests a particular resource the claim edge is replaced by a request edge. Similarly, when a process releases a resource, the assignment edge is replaced by a claim edge.

Under deadlock avoidance we require all processes to declare their maximum requirements a priori. In this specific scheme, we relax this and allow processes to add claim edges only if all edges currently associated with it are claim edges.
- When a process requests a particular resource the claim edge is replaced by a request edge. Similarly, when a process releases a resource, the assignment edge is replaced by a claim edge.

- Under deadlock avoidance we require all processes to declare their maximum requirements a priori. In this specific scheme, we relax this and allow processes to add claim edges only if all edges currently associated with it are claim edges.

- As we have seen earlier, there can be a deadlock only if there is a directed cycle in the original resource-allocation graph.
- When a process requests a particular resource the claim edge is replaced by a request edge. Similarly, when a process releases a resource, the assignment edge is replaced by a claim edge.

- Under deadlock avoidance we require all processes to declare their maximum requirements a priori. In this specific scheme, we relax this and allow processes to add claim edges only if all edges currently associated with it are claim edges.

- As we have seen earlier, there can be a deadlock only if there is a directed cycle in the original resource-allocation graph.

- Thus a request is granted only if changing the claim edge to an assignment edge does not create a cycle in the resource-allocation graph.
The resource-allocation graph based algorithm described above is not applicable here, since a cycle does not imply a deadlock.
The resource-allocation graph based algorithm described above is not applicable here, since a cycle does not imply a deadlock.

The algorithm used here is called *banker’s algorithm*. The name is used because a similar procedure could be used by a banking system to ensure that the bank never allocates its available cash in a way that it can no longer satisfy its customers’ needs.
The resource-allocation graph based algorithm described above is not applicable here, since a cycle does not imply a deadlock.

The algorithm used here is called banker’s algorithm. The name is used because a similar procedure could be used by a banking system to ensure that the bank never allocates its available cash in a way that it can no longer satisfy its customers’ needs.

A process entering the system must declare the maximum number of instances of each resource type that it may need.
The resource-allocation graph based algorithm described above is not applicable here, since a cycle does not imply a deadlock.

The algorithm used here is called *banker’s algorithm*. The name is used because a similar procedure could be used by a banking system to ensure that the bank never allocates its available cash in a way that it can no longer satisfy its customers’ needs.

A process entering the system must declare the maximum number of instances of each resource type that it may need.

If a process requests some resources, the system determines if granting the request will leave the system in a safe state. The allocation is made only if the it will, otherwise, the process must wait for some other processes to release resources before it is granted.
A number of data structures need to be maintained. These data structures encode the state of the resource allocation system. The number of processes in the system is denoted by $n$ and the number of resource-types by $m$.

These data structures vary in both value and size over time.

- **Available:** A vector of length $m$ with an entry for each resource-type.
A number of data structures need to be maintained. These data structures encode the state of the resource allocation system. The number of processes in the system is denoted by $n$ and the number of resource-types by $m$.

These data structures vary in both value and size over time.

- **Available**: A vector of length $m$ with an entry for each resource-type.
- **Max**: An $nxm$ matrix indicating the maximum request for each resource type by each process.
A number of data structures need to be maintained. These data structures encode the state of the resource allocation system. The number of processes in the system is denoted by $n$ and the number of resource-types by $m$. These data structures vary in both value and size over time.

- **Available**: A vector of length $m$ with an entry for each resource-type.
- **Max**: An $nxm$ matrix indicating the maximum request for each resource type by each process.
- **Allocation**: It is an $nxm$ matrix indicating the number of units of each resource type currently allocated to each process.
A number of data structures need to be maintained. These data structures encode the state of the resource allocation system. The number of processes in the system is denoted by $n$ and the number of resource-types by $m$. These data structures vary in both value and size over time.

- **Available**: A vector of length $m$ with an entry for each resource-type.
- **Max**: An $nxm$ matrix indicating the maximum request for each resource type by each process.
- **Allocation**: It is an $nxm$ matrix indicating the number of units of each resource type currently allocated to each process.
- **Need**: This is again, an $nxm$ matrix with each entry representing the difference between the max requirement of the resource type by the process and the current allocation.
Work and Finish are vectors of length \( m \) and \( n \) respectively. The following initialisations are performed: Work \( \leftarrow \) Available and Finish \( \leftarrow \) false, \( \forall i \in \{1, \ldots, n\} \).
Safety algorithm

1. Work and Finish are vectors of length m and n respectively. The following initialisations are performed: Work ← Available and Finish ← false, ∀i ∈ {1, . . . , n}.

2. Find an i such that both
   a. Finish[i] = false
   b. Needi ≤ Work.
   If no such i exists go to Step 4.
Safety algorithm

1. Work and Finish are vectors of length $m$ and $n$ respectively. The following initialisations are performed: $Work \leftarrow Available$ and $Finish \leftarrow false, \forall i \in \{1, \ldots, n\}$.

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

If no such $i$ exists go to Step 4.

3. $Work \leftarrow Work + Allocation_i$;
   $Finish[i] \leftarrow true$
   Go to step 2.
1. Work and Finish are vectors of length $m$ and $n$ respectively. The following initialisations are performed: Work ← Available and Finish ← false, $\forall i \in \{1, \ldots, n\}$.

2. Find an $i$ such that both
   a. Finish[$i$] = false
   b. Need[$i$] $\leq$ Work.
   If no such $i$ exists go to Step 4.

3. Work ← Work + Allocation[$i$];
   Finish[$i$] ← true
   Go to step 2.

4. If Finish[$i$] = true for all $i$, then the system is in a safe state.
If $Request_i \leq Need_i$, go to step 2. Otherwise, declare claim-exceeded error.
Resource-Request Algorithm

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, declare claim-exceeded error.

2. If $Request_i \leq Available$ go to Step 3. Otherwise $P_i$ must wait for the resources to become available.
**Resource-Request Algorithm**

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, declare claim-exceeded error.

2. If $\text{Request}_i \leq \text{Available}$ go to Step 3. Otherwise $P_i$ must wait for the resources to become available.

3. Modify the variables as follows, without actually executing.
   
   $\text{Available} \leftarrow \text{Available} - \text{Request}_i$
   
   $\text{Allocation}_i \leftarrow \text{Allocation}_i + \text{Request}_i$
   
   $\text{Need}_i \leftarrow \text{Need}_i - \text{Request}_i$

   Perform this last step only if it results in a safe state. Otherwise the allocation is deferred.
Example state

Resource types $A$:10 instances, $B$:5 instances, $C$:7 instances.

**Table:** Multiple instances of resource-types

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

### The Need Vector

<table>
<thead>
<tr>
<th>Process</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>7 4 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1 2 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>6 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0 1 1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>
Process $P_1$ makes a request for one unit of resource type $A$, and two instances of resource type $C$.

<table>
<thead>
<tr>
<th>Process</th>
<th>Need</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>