Deadlock avoidance:

When processes begin executing, they must tell system what resources they might request at some time in their execution.

System then evaluates every resource request to see whether there is any future set of requests by the other processes that could result in deadlock if this request is granted.

If future deadlock is possible, resource request is denied (state of system if request were granted is called an unsafe state).

If future deadlock is not possible, resource request is granted (state after request is granted is called a safe state).

Note that an unsafe state is not the same as deadlock, rather an unsafe state is one that might eventually lead to deadlock.

A safe state can never lead to deadlock; this is what it means to be safe.

Banker's Algorithm

An algorithm to decide whether a request can be granted by checking for unsafe states.

\[ n = \# \text{ of processes } (p_1, p_2, \ldots, p_n) \]

\[ m = \# \text{ of resource types } \]
Data structures:

Available \([1 \ldots m]\) : Available \([j]\) gives the \# of resources of type \(j\) that are currently available

Max \([1 \ldots n, 1 \ldots m]\) : Max \([i, j]\) gives the max. \# of resources of type \(j\) that process \(i\) will need at any time

Allocation \([1 \ldots n, 1 \ldots m]\) : Allocation \([i, j]\) gives the \# of resources of type \(j\) currently allocated to process \(i\)

Need \([1 \ldots n, 1 \ldots m]\) : Need \([i, j]\) gives the remaining resources of type \(j\) that process \(i\) might request

\[\text{Need} \ [i, j] = \text{Max} \ [i, j] - \text{Allocation} \ [i, j]\]

Request \(_i\) \([1 \ldots m]\) : A resource request made by process \(i\)

\[\text{Request}_i \ [j] \text{ gives the \# of resources of type } j \text{ being requested by process } i\]

Upon receiving Request \(_i\), the system does the following:

1) If \(\text{Request}_i \ [j] \leq \text{Need} \ [i, j], \forall j\), then go to Step 2.
   Else error (\(i\) is requesting more than its stated max. for some resource type).

2) If \(\text{Request}_i \ [j] \leq \text{Available} \ [j], \forall j\), then go to Step 3.
   Else process \(i\) must wait (some resource is not available currently).

3) Pretend to grant request:

\[\text{Available} \ [j] \leftarrow \text{Available} \ [j] - \text{Request}_i \ [j], \forall j\]
\[\text{Allocation} \ [i, j] \leftarrow \text{Allocation} \ [i, j] + \text{Request}_i \ [j], \forall j\]
\[\text{Need} \ [i, j] \leftarrow \text{Need} \ [i, j] - \text{Request}_i \ [j], \forall j\]

Go to Step 4.
4) If current state unsafe, then:

\[
\text{available}[j] \leftarrow \text{available}[j] + \text{request}_i[j], \forall j \\
\text{allocation}[i,j] \leftarrow \text{allocation}[i,j] \cup \text{request}_i[j], \forall j \\
\text{need}[i,j] \leftarrow \text{need}[i,j] + \text{request}_i[j], \forall j \\
\text{deny request}
\]

Else grant request.

To check if current state is unsafe, do the following:

1) Initialize:
\[
\text{work}[j] \leftarrow \text{available}[j], \forall j \\
\text{finish}[i] \leftarrow \text{false}, \forall i
\]

Go to step 2.

2) Find \( i \) such that \( (\text{finish}[i] = \text{false}) \text{ AND } (\text{need}[i,j] \leq \text{work}[j], \forall j) \)

If no such \( i \) exists, then go to step 4.
Else go to step 3.

3) \( \text{work}[j] \leftarrow \text{work}[j] + \text{allocation}[i,j], \forall j \).
\( \text{finish}[i] \leftarrow \text{true} \).

Go to step 2.

4) If \( (\text{finish}[i] \neq \text{true}, \forall i) \) then state is unsafe.

\begin{itemize}
\item there is no order in which processes can request maximum resources that will allow all processes to finish
\item deadlock inevitable from this state
\item if all processes use maximum resources
\end{itemize}
Example of Banker's Algorithm Execution

Processes: P1, P2, P3, P4, P5
Resource Types: A, B, C
(10) (5) (7)

Current state:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>753</td>
<td>3 3 2</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P2</td>
<td>2 0 0</td>
<td>322</td>
<td></td>
<td>1 2 2</td>
</tr>
<tr>
<td>P3</td>
<td>3 0 2</td>
<td>902</td>
<td></td>
<td>6 0 0</td>
</tr>
<tr>
<td>P4</td>
<td>2 1 1</td>
<td>222</td>
<td></td>
<td>0 1 1</td>
</tr>
<tr>
<td>P5</td>
<td>0 0 2</td>
<td>433</td>
<td></td>
<td>4 3 1</td>
</tr>
<tr>
<td>Total</td>
<td>7 2 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Safe state?

YES, because if requests come in order P2, P4, P5, P3, P1, all processes will finish.

Need | Available | Need ≤ Available?
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1 2 2</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P4</td>
<td>0 1 1</td>
<td>5 3 2</td>
</tr>
<tr>
<td>P5</td>
<td>4 3 1</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P3</td>
<td>6 0 0</td>
<td>7 4 5</td>
</tr>
<tr>
<td>P1</td>
<td>7 4 3</td>
<td>10 4 7</td>
</tr>
</tbody>
</table>
suppose P₂ requests \((1, 0, 2)\):

1) Is \((1, 0, 2) \leq (1, 2, 2)\) ?
   YES, Go to Step 2.

2) Is \((1, 0, 2) \leq (3, 3, 2)\) ?
   YES, Go to Step 3.

3) Pretend to grant request:
   Allocation \([2, \star] \leftarrow (3, 0, 2)\)
   Need \([2, \star] \leftarrow (0, 2, 0)\)
   Available \(\leftarrow (2, 3, 0)\)

4) Check if state is safe
   P₂ can finish \(\Rightarrow\) Available \(\leftarrow (5, 3, 2)\)
   P₄ can finish \(\Rightarrow\) Available \(\leftarrow (7, 4, 3)\)
   all can finish \(\Rightarrow\) state is safe \(\Rightarrow\) request granted

from previous state, suppose P₁ requests \((0, 0, 2)\):

1) Is \((0, 0, 2) \leq (7, 4, 3)\) ?
   YES, Go to Step 2.

2) Is \((0, 0, 2) \leq (3, 3, 2)\) ?
   YES, Go to Step 3.

3) Pretend to grant request:
   Allocation \([1, \star] \leftarrow (0, 1, 2)\)
   Need \([1, \star] \leftarrow (7, 4, 3)\)
   Available \(\leftarrow (3, 3, 0)\)
4) Check if state is safe
   Can any process finish?
   NO, because Need \( [i, *] > \) Available, \( \forall i \)
   \( \Rightarrow \) state is unsafe \( \Rightarrow \) request denied
   Allocation, Need, Available are reset to previous values

---

**Deadlock Detection and Recovery**

Idea is to wait until deadlock happens, detect it, and then break the deadlock periodically. The system checks for deadlock (this can be done by a simple variation of the Banker's algorithm).

If deadlock is detected, a recovery action is taken to break the deadlock.

---

**Deadlock recovery:**

1) process termination - abort some processes until deadlock is broken
2) resource preemption - force processes to release some resources