Deadlock
Safe States

- Sequence \(<P_1, P_2, ..., P_n>\) is safe if for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources plus the resources held by all the \(P_j\) with \(j < i\).
  - If \(P_i\) resource needs are not immediately available, then \(P_i\) can wait until all \(P_j\) have finished.
  - When \(P_i\) is finished, \(P_i\) can obtain needed resources, execute, return allocated resources, and terminate.
  - When \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, and so on.

- The system is in a safe state if there exists a safe sequence for all processes.
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.
Basic Facts

- If a system is in a safe state there can be no deadlock.

- If a system is in unsafe state, there exists the **possibility** of deadlock.

- **Avoidance** strategies ensure that a system will never enter an unsafe state.
Banker’s Algorithm

- Applicable when there are multiple instances of each resource type.
- In a bank, the cash must never be allocated in a way such that it cannot satisfy the need of all its customers.
- Each process must state a priori the maximum number of instances of each kind of resource that it will ever need.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
Banker’s Algorithm: Data Structures

Let $n$ = number of processes, $m$ = number of resources types.

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

Note that:

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$
Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   \[ \text{Work} = \text{Available} \]
   \[ \text{Finish}[i] = \text{false} \text{ for } i = 1, 3, \ldots, n. \]

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.
Resource-Request Algorithm for Process $P_i$

Request = request vector for process $P_i$. If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available.

3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:
   
   \[
   \begin{align*}
   Available &= Available - Request, \\
   Allocation &= Allocation + Request, \\
   Need &= Need - Request,
   \end{align*}
   \]

   - If safe $\Rightarrow$ the resources are allocated to $P_i$.
   - If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored.
Example of Banker’s Algorithm

- 5 processes $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).
- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Example (Cont.)

- The content of the matrix. Need is defined to be Max – Allocation.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Need</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria.
Example $P_1$ Request (1,0,2) (Cont.)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2)$) $\Rightarrow$ true.

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

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for $(3,3,0)$ by $P_4$ be granted?
- Can request for $(0,2,0)$ by $P_0$ be granted?
When Deadlock Happens

- Another way to deal with deadlock is not to use either prevention or avoidance. The system may enter a deadlock state; the OS will deal with that when [if] it happens.

- What is needed in such a system:
  - a detection algorithm to determine when deadlock states are entered, and
  - a recovery scheme to get the system back on a safe state.
Single Instance of Each Resource Type

- Maintain a *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if $P_j$ is waiting for $P_i$.

- Periodically invoke an algorithm that searches for a cycle in the graph.

- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph.
Resource-Allocation Graph and Wait-for Graph

Resource-Allocation Graph

Corresponding wait-for graph

03/01/2010
CSCI 315 Operating Systems Design
Several Instances of a Resource Type

- **Available:** A vector of length $m$ indicates the number of available resources of each type.

- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

- **Request:** An $n \times m$ matrix indicates the current request of each process. If $Request_{ij} = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$. 
Detection Algorithm

1. Let Work and Finish be vectors of length $m$ and $n$, respectively. Initialize:
   (a) Work = Available
   (b) For $i = 1, 2, \ldots, n$, if Allocation, $\neq 0$, then

2. Find an index $i$ such that both:
   (a) Finish[$i$] == false
   (b) Request, $\leq$ Work
   If no such $i$ exists, go to step 4.

3. Work = Work + Allocation,
       Finish[$i$] = true
   Go to step 2.

4. If Finish[$i$] == false, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[$i$] == false, then $P_i$ is deadlocked.
Example of Detection Algorithm

- Five processes $P_0$ through $P_4$; three resource types
  A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in $Finish[i] = true$ for all $i$. 

03/01/2010 CSCI 315 Operating Systems Design
Example (Cont.)

- $P_2$ requests an additional instance of type C. 

  \[
  \begin{array}{c|ccc}
  \text{Request} & A & B & C \\
  \hline
  P_0 & 0 & 0 & 0 \\
  P_1 & 2 & 0 & 1 \\
  P_2 & 0 & 0 & 1 \\
  P_3 & 1 & 0 & 0 \\
  P_4 & 0 & 0 & 2 \\
  \end{array}
  \]

- State of the system?
  - Can reclaim resources held by process $P_0$, but have insufficient resources to fulfill the requests of other processes.
  - Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$, and $P_4$. 

03/01/2010  CSCI 315 Operating Systems Design  17
Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    (one for each disjoint cycle)

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost.

- **Rollback** – return to some safe state, restart process for that state.

- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor.
Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection
  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.