Deadlocks
The Deadlock Problem

• A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

• Example
  - System has 2 disk drives.
  - P1 and P2 each hold one disk drive and each needs another one.

• Example
  - semaphores A and B, initialized to 1
    
    | P0  | P1 |
    |-----|----|
    | wait (A); | wait(B) |
    | wait (B); | wait(A) |
System Model

- Resource types $R_1, R_2, \ldots, R_m$.
  - CPU cycles, memory space, I/O devices.
- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - Request.
  - Use.
  - Release.
Necessary Conditions for Deadlock

- **Mutual exclusion**: only one process at a time can use a resource.
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption**: a resource can be released only voluntarily by the process holding it.
- **Circular wait**: there exists a set \( \{ P_0, P_1, ..., P_n \} \) of waiting processes such that \( P_0 \) is waiting for a resource that is held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), ..., \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Resource Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of processes.
  - $R = \{R_1, R_2, \ldots, R_m\}$, a set consisting of all resource types.
- Request edge – directed edge $P_i \rightarrow R_j$.
- Assignment edge – directed edge $R_j \rightarrow P_i$. 
Resource Allocation Graph

- Process.
- Resource Type with 4 instances.
- \( P_i \) requests instance of \( R_j \).
- \( P_i \) is holding an instance of \( R_j \).
Deadlock Rules

• If graph contains no cycles then no deadlock.
• If graph contains a cycle then,
  – if only one instance per resource type, deadlock.
  – if several instances per resource type, possibility of deadlock.
Resource Allocation Graph
(Deadlock?)
Graph With Cycle (Deadlock?)
Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
  - Deadlock prevention.
- Allow the system to enter a deadlock state and then recover.
  - Deadlock detection.
- Ignore the problem and pretend that deadlocks never occur in the system.
  - Used by most operating systems, including Linux.
  - “The Ostrich Algorithm”.
  - Why might this make sense?
Deadlock Prevention

- Recall our preconditions for deadlock:
  - Mutual exclusion.
  - Hold and wait.
  - No preemption.
  - Circular wait.
- If we can exclude any one, we can prevent deadlock.
- Mutual Exclusion?
Hold and Wait

- Require processes to request all resources at startup (atomically).
- Require processes to release all resources before requesting new ones.
- Drawbacks?
No Preemption 1

- If a process that is holding some resources requests another resource that cannot be allocated to it, then all resources held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
No Preemption 2

- If a process requests resources that are held, check to see if the holding process is waiting.
- If it is waiting, preempt its resources.
- If resources are held by a non-waiting process, requesting process must wait.
- Any drawbacks to the two preemption schemes?
Circular Wait

• Impose a total ordering of all resource types, and require that each process requests resources in increasing order of enumeration.

• Why this works:
  – It is not possible to have a set of processes such that every process is waiting for a resource with a higher number than the number of a currently held resource.

• Any drawbacks to this algorithm?
Deadlock Avoidance

- Our deadlock prevention methods put restrictions on how processes could request resources.
- Deadlock avoidance mechanisms require the OS to keep track of requests and resources to make sure that no deadlock can occur.
- This will require the system to have some a-priori information about the requests that a process could make.
Deadlock Avoidance

- Each process declares the max number of resources of each type it may need.
- System examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - If a request could lead to a circular wait, the process must wait until the state changes.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
Safe States

- System is in **safe state** if there exists a sequence \(<P_1, P_2, ..., P_n>\) of ALL the processes is the systems such that for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all the \(P_j\), with \(j < i\).

- i.e. There remains some way to grant everyone's maximum request.

- (Assumption is that all processes terminate eventually.)
Safe State Example

- Example from the book:
  - Assume that a system has 12 tape drives.
  - This is a safe state:

<table>
<thead>
<tr>
<th></th>
<th>MAX</th>
<th>ALLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

- The sequence (P1, P0, P2) works.
Safe State Example

- This is *not* a safe state:

<table>
<thead>
<tr>
<th></th>
<th>MAX</th>
<th>ALLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

- P1 is the only process who's max request could still be granted.
- When it exits, only 4 free drives.
- If both P0 and P2 then request their max drives, deadlock occurs.

• An unsafe state does not guarantee deadlock.
Resource Allocation Graph Algorithm

- Only works if there is at most one resource of each type.

- **Claim edge** $P_i \rightarrow R_j$ indicates that process $P_j$ may request resource $R_j$; represented by a dashed line.

- Claim edge converts to request edge when a process requests a resource.

- Request edge converted to an assignment edge when the resource is allocated to the process.

- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed *a priori* in the system.
Resource Allocation Graph Algorithm

- Suppose that process $P_i$ requests a resource $R_j$.
- Grant request only if converting the request edge to an assignment edge does not result in the formation of a cycle.
Banker's Algorithm

- Multiple resource instances.
- Each process must declare maximum use.
- Any process that makes a request that would lead to an unsafe state must wait.
- When a process gets all its resources it must return them in a finite amount of time.
Banker's Algorithm Data Structures

- $n$ processes, $m$ resource types.
- **Available**: Vector of length $m$. Available[$j$] = number of available resources of type $R_j$.
- **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.

$$\text{Need} \ [i,j] = \text{Max} [i,j] - \text{Allocation} \ [i,j].$$
## Banker's Algorithm: Checking For Safety

1. Let **Work** and **Finish** be vectors of length \(m\) and \(n\), respectively. Initialize:
   
   \[
   \begin{align*}
   & Work = Available \\
   & Finish[i] = false \text{ for } i = 0, 1, ..., n-1.
   \end{align*}
   \]

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. \[
   \begin{align*}
   & Work = Work + Allocation_i \\
   & Finish[i] = true
   \end{align*}
   \]

   go to step 2.

4. If \(Finish[i] == true\) for all \(i\), then the system is in a safe state.
Banker's Algorithm: Requesting Resources

\( \text{Request}_i = \text{request vector for process } P_i. \)

1. If \( \text{Request}_i \leq \text{Need}_i \) go to step 2. Otherwise, raise error condition.

2. If \( \text{Request}_i \leq \text{Available} \), 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*}
   \text{Available} &= \text{Available} - \text{Request}; \\
   \text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\
   \text{Need}_i &= \text{Need}_i - \text{Request}_i;
   \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.
Banker's Algorithm Analysis

- Thoughts on running time of safety check?
- Drawbacks to the bankers algorithm?
Deadlock Detection: Single Instance of Each Resource

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

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

- 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.
Wait-for Graph

- Why bother collapsing the graph?
Deadlock Detection With Multiple Resource Instances

• Big picture:
  – Periodically we check the system for deadlocks.
  – The deadlock check algorithm is very similar to the safety check algorithm we saw before.
  – Instead of looking for unsafe states, we look for deadlocked states.
Deadlock Detection Data Structures

- **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 $Request[i_j] = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$. 
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 $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.

2. Find an index $i$ such that both:
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$

If no such $i$ exists, go to step 4.
3. \( Work = Work + Allocation_i \)
   \( Finish[i] = true \)
   go to step 2.

4. If \( Finish[i] == \) 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.
Deadlock Recovery: 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?
Deadlock Recovery: Resource Preemption

- Selecting a victim – minimize some cost measure.
- Rollback – return to some safe state, restart process for that state.
  - Can be tricky.
- Starvation – same process may always be picked as victim
  - Include number of rollbacks in cost factor.
Acknowledgments

• Portions of these slides are taken from Power Point presentations made available along with:

• Original versions of those presentations can be found at: