Deadlock

Notice: The slides for this lecture have been largely based on those accompanying an earlier edition of the course text Operating Systems Concepts with Java, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.
A System Model

- Resource types $R_1, R_2, \ldots, R_m$
  
  \textit{(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
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **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, after that process has completed its task.
- **Circular wait**: there exists a set \( \{P_0, P_1, \ldots, P_n\} \) 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 \( P_2 \), \ldots, \( P_{n-1} \) 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

**Graph: G=(V,E)**

- The nodes in V can be of two types (partitions):
  - \( P = \{P_1, P_2, ..., P_n\} \), the set consisting of all the processes in the system.
  - \( R = \{R_1, R_2, ..., R_m\} \), the set consisting of all resource types in the system.
- request edge – directed edge \( P_i \rightarrow R_j \)
- assignment edge – directed edge \( R_j \rightarrow P_i \)
Resource Allocation Graph

Example 1
Resource Allocation Graph

Example 2

[Diagram of a resource allocation graph with processes P1, P2, P3, and P4, and resources R1 and R2.]
Basic Facts

- **If graph contains no cycles** $\Rightarrow$ no deadlock.

- **If graph contains a cycle** $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.
Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.
Deadlock Prevention

**Restrain the ways request can be made.**

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being 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.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
Deadlock Avoidance

The system has additional *a priori* information.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
Safe States

• Sequence $<P_1, P_2, \ldots, P_n>$ 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 $P_i$'s resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished.
  - When $P_i$ is finished, $P_j$ can obtain needed resources, execute, return allocated resources, and terminate.
  - When $P_i$ terminates, $P_{n+1}$ 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.
Safe, Unsafe, and Deadlock States

deadlock

unsafe

safe
Resource-Allocation Graph Algorithm

- **Goal:** not to allow the system to enter an unsafe state.
- Applicable only when there is a single instance of each resource type.

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

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

- 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 for Deadlock Avoidance
Unsafe State in Resource-Allocation Graph

02/24/2010
CSCI 315 Operating Systems Design
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, and $m =$ 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.

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

1. Let Work and Finish be vectors of length $m$ and $n$, respectively.
   Initialize:
   
   \[ \text{Work} = \text{Available} \]
   
   \[ \text{Finish}[i] = \text{false} \text{ for } i = 1, 2, \ldots, n \]

2. Find an $i$ such that both:
   
   (a) \( \text{Finish}[i] = \text{false} \)
   
   (b) \( \text{Need}_i \leq \text{Work} \)

   If no such $i$ exists, go to step 4.

3. \[ \text{Work} = \text{Work} + \text{Allocation}, \]
   
   \[ \text{Finish}[i] = \text{true} \]
   
   go to step 2.

4. If \( \text{Finish}[i] = \text{true} \) for all $i$, then the system is in a safe state.
Resource-Request Algorithm for Process $P_i$

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

1. If $Request \leq \text{Need}$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request \leq \text{Available}$, 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:
   
   - $\text{Available} = \text{Available} - \text{Request}_i$
   - $\text{Allocation} = \text{Allocation} + \text{Request}_i$
   - $\text{Need} = \text{Need} - \text{Request}_i$

   - 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>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>
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>( 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 0 1 0</td>
<td>7 4 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P_0 3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>P_1 3 0 1</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>P_2 2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P_3 0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
<tr>
<td>P_4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?