Deadlock

- B. Ramamurthy

Introduction

- Parallel operation among many devices driven by concurrent processes contribute significantly to high performance. But concurrency also results in contention for resources and possibility of deadlock among the vying processes.
- Deadlock is a situation where a group of processes are permanently blocked waiting for the resources held by each other in the group.
- Typical application where deadlock is a serious problem: Operating system, data base accesses, and distributed processing.

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

Deadlock Characterization

- 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 $P_2$, …, $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$.

Deadlock can arise if four conditions hold simultaneously.

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 consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
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 (Cont.)

- **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

- Requires that the system has some additional a priori information available.
- **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 State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- 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 + 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.

Resource-Allocation Graph Algorithm

- 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 \textit{a priori} in the system.

Banker’s Algorithm

- Multiple instances.
- Each process must \textit{a priori} claim maximum use.
- 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.

Data Structures for the Banker’s Algorithm

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 \( \text{Max}(i,j) = k \), then process \( P_i \) may request at most \( k \) instances of resource type \( R_j \).
- Allocation: \( n \times m \) matrix. If \( \text{Allocation}(i,j) = k \), then \( P_i \) is currently allocated \( k \) instances of \( R_j \).
- Need: \( n \times m \) matrix. If \( \text{Need}(i,j) = k \), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task.

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

Safety Algorithm

1. Let \( \text{Work} \) and \( \text{Finish} \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   - \( \text{Work} = \) Available
   - \( \text{Finish}(i) = \) false for \( i = 1, 2, \ldots, n \).
2. Find and \( i \) such that both:
   - (a) \( \text{Finish}(i) = \) false
   - (b) \( \text{Need}(i) \leq \text{Work} \)
3. \( \text{Work} = \text{Work} + \text{Allocation}(i) \)
   - \( \text{Finish}(i) = \) true
   - go to step 2.
4. If \( \text{Finish}(i) = \) 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 $R_i \leq k$ then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $\text{Request} \leq \text{Need}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\text{Request} \leq \text{Available}$ 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:
   - $\text{Available} = \text{Available} - \text{Request}$
   - $\text{Allocation} = \text{Allocation} + \text{Request}$
   - $\text{Need} = \text{Need} - \text{Request}$
   - 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>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example (Cont.)

- The content of the matrix. Need is defined to be $\text{Max} - \text{Allocation}$.

<table>
<thead>
<tr>
<th>Need</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>6</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.

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Example $P_i$ Request (1,0,2) (Cont.)

- Check that $\text{Request} \leq \text{Available}$ (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>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_2, P_0>$ 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?

Single Instance of Each Resource Type

- 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.
- 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.