Lecture - X
Deadlocks - II

Tevfik Koşar

Louisiana State University
February 21st, 2008

Roadmap

• Deadlocks
  - Deadlock Prevention
  - Deadlock Detection
Exercise

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possibility of a deadlock in this implementation.

b. Modify the order of some of the get requests to prevent the possibility of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

<table>
<thead>
<tr>
<th>void P0()</th>
<th>void P1()</th>
<th>void P2()</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (true) {</td>
<td>while (true) {</td>
<td>while (true) {</td>
</tr>
<tr>
<td>get(A);</td>
<td>get(D);</td>
<td>get(C);</td>
</tr>
<tr>
<td>get(B);</td>
<td>get(E);</td>
<td>get(F);</td>
</tr>
<tr>
<td>get(C);</td>
<td>get(B);</td>
<td>get(D);</td>
</tr>
<tr>
<td>// critical region:</td>
<td>// critical region:</td>
<td>// critical region:</td>
</tr>
<tr>
<td>// use A, B, C</td>
<td>// use D, E, B</td>
<td>// use C, F, D</td>
</tr>
<tr>
<td>release(A);</td>
<td>release(D);</td>
<td>release(C);</td>
</tr>
<tr>
<td>release(B);</td>
<td>release(E);</td>
<td>release(F);</td>
</tr>
<tr>
<td>release(C);</td>
<td>release(B);</td>
<td>release(D);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
  ➔ deadlock prevention or avoidance
- 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
  ➔ Programmers should handle deadlocks (UNIX, Windows)
Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- Restrain the ways request can be made.

- **Mutual Exclusion** - not required for sharable resources; must hold for nonsharable resources.
  - Eg. read-only files

- **Hold and Wait** - must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.

  Example: Read from DVD to memory, then print.
  1. holds printer unnecessarily for the entire execution
     - Low resource utilization
  2. may never get the printer later
     - 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 Detection

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

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.
Resource-Allocation Graph and Wait-for Graph

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 $Request[i, j] = k$, 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, ..., *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.

Detection Algorithm (Cont.)

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.

Algorithm requires an order of \(O(m \times n^2)\) operations to detect whether the system is in deadlocked state.
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>Allocation</th>
<th>Request</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>0 0 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 3</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

Example (Cont.)

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

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
</tr>
<tr>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_3$</td>
</tr>
<tr>
<td>$P_4$</td>
</tr>
</tbody>
</table>

- State of system?
  - Can reclaim resources held by process $P_0$, but insufficient
Summary

- Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

- **Next Lecture: Main Memory - I**
- **Reading Assignment: Chapter 7 from Silberschatz.**

Acknowledgements

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR