Lecture - XII
Deadlocks & Main Memory Management

Tevfik Koşar

University at Buffalo
October 11th, 2011

Roadmap

- Deadlocks
  - Resource Allocation Graphs
  - Deadlock Detection
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Recovery
- Main Memory Management
Deadlock Avoidance

Deadlock Prevention: prevent deadlocks by restraining resources and making sure one of 4 necessary conditions for a deadlock does not hold. (system design)

--> possible side effect: low device utilization and reduced system throughput

Deadlock Avoidance: Requires that the system has some additional \textit{a priori} information available. (dynamic request check)

i.e. request disk and then printer..

or request at most n resources

--> allows more concurrency

• Similar to the difference between a traffic light and a police officer directing the traffic!

Deadlock Avoidance

• Simplest and most useful model requires that each process declare the \textit{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 \textit{state} is defined by the number of available and allocated resources, and the maximum demands of the processes.
Example

P1:
Request Disk
Request Printer

....
Release Printer
Release Disk

P2:
Request Printer
Request Disk

....
Release Disk
Release Printer

Safe State

- A state is safe if the system can allocate resources to each process (upto its maximum) in some order and can still avoid a deadlock.

- 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.
Safe State

- 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 + 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 \(P_j\) 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.
- If no such sequence exists, the state is **unsafe!**

Basic Facts

- If a system is in safe state \(\Rightarrow\) no deadlocks.
- If a system is in unsafe state \(\Rightarrow\) possibility of deadlock.
- **Avoidance** \(\Rightarrow\) ensure that a system will never enter an unsafe state.
### Safe, Unsafe, Deadlock State

![Diagram](image)

### Example

Consider a system with 3 processes and 12 disks.

At $t = t_0$;

<table>
<thead>
<tr>
<th>Process</th>
<th>Maximum Needs</th>
<th>Current Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>
Example (cont.)

Consider a system with 3 processes and 12 disks.

At $t = t_1$;

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
</tbody>
</table>

Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicated 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.

- 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.
Resource-Allocation Graph For Deadlock Avoidance

Unsafe State In Resource-Allocation Graph
Banker’s Algorithm

- Works for multiple resource instances.
- Each process declares maximum # of resources it may need.
- When a process requests a resource, it may have to wait if this leads to an unsafe state.
- 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 $Max [i,j] = k$, 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.

Safety Algorithm

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

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. \(\text{Work} = \text{Work} + \text{Allocation}_i\)
   \(\text{Finish}[i] = \text{true}\)
   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 \(P_i\)

Let Request\(_i\) be the 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 \(\text{Request}_i \leq \text{Need}_i\) go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

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:
   \[\text{Available} = \text{Available} - \text{Request}_i;\]
   \[\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;\]
   \[\text{Need}_i = \text{Need}_i - \text{Request}_i;\]

   • 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>Process</th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<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 of Banker’s Algorithm

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

$$
\begin{align*}
\text{Need} \\
A & B & C \\
P_0 & 7 & 4 & 3 \\
P_1 & 1 & 2 & 2 \\
P_2 & 6 & 0 & 0 \\
P_3 & 0 & 1 & 1 \\
P_4 & 4 & 3 & 1 \\
\end{align*}
$$
### Example of Banker’s Algorithm

- **Snapshot at time** $T_0$:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<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>

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

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

\[
\begin{array}{ccc|ccc|ccc}
\text{Allocation} & \text{Need} & \text{Available} \\
ABCD & ABC & ABC \\
P_0 & 010 & 743 & 230 \\
P_1 & 302 & 020 & \\
P_2 & 301 & 600 & \\
P_3 & 211 & 011 & \\
P_4 & 002 & 431 & \\
\end{array}
\]

- 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?

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes. \(\rightarrow\) expensive

- Abort one process at a time until the deadlock cycle is eliminated. \(\rightarrow\) overhead of deadlock detection alg.

- 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.

Main Memory Management
Memory Management Requirements

- The O/S must fit multiple processes in memory
  - memory needs to be subdivided to accommodate multiple processes
  - memory needs to be allocated to ensure a reasonable supply of ready processes so that the CPU is never idle
  - memory management is an optimization task under constraints

Fitting processes into memory is like fitting boxes into a fixed amount of space

Memory Allocation

- Fixed-partition allocation
  - Divide memory into fixed-size partitions
  - Each partition contains exactly one process
  - The degree of multi programming is bound by the number of partitions
  - When a process terminates, the partition becomes available for other processes

<table>
<thead>
<tr>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>process 5</td>
</tr>
<tr>
<td>process 9</td>
</tr>
<tr>
<td>process 10</td>
</tr>
<tr>
<td>process 2</td>
</tr>
</tbody>
</table>

⇒ no longer in use
Memory Allocation (Cont.)

- Variable-partition Scheme (Dynamic)
  - When a process arrives, search for a hole large enough for this process
  - Hole - block of available memory; holes of various size are scattered throughout memory
  - Allocate only as much memory as needed
  - Operating system maintains information about:
    - allocated partitions
    - free partitions (hole)

Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

*First-fit* is faster.

*Best-fit* is better in terms of storage utilization.

*Worst-fit* may lead less fragmentation.
Example

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

Summary

• Deadlocks
  - Resource Allocation Graphs
  - Deadlock Detection
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Recovery
• Main Memory Management
Acknowledgements


- “Modern Operating Systems” book and supplementary material by A. Tanenbaum

- R. Doursat and M. Yuksel from UNR