Chapter 10  

Deadlock

What is Deadlock?
- Two or more entities need a resource to make progress, but will never get that resource
- Examples from everyday life:
  - Gridlock of cars in a city
  - Class scheduling: Two students want to swap sections of a course, but each section is currently full.
- Examples from Operating Systems:
  - Two processes spool output to disk before either finishes, and all free disk space is exhausted
  - Two processes consume all memory buffers before either finishes

Deadlock Illustration

A set of processes is in a DEADLOCK state when every process is waiting for an event initiated by another process in the set.

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request X</td>
<td>Request Y</td>
</tr>
<tr>
<td>Request Y</td>
<td>Request X</td>
</tr>
<tr>
<td>Release X</td>
<td>Release Y</td>
</tr>
<tr>
<td>Release Y</td>
<td>Release X</td>
</tr>
</tbody>
</table>

Deadlock Illustration

- A requests & receives X
- B requests & receives Y
- A requests Y and blocks
- B requests X and blocks

The “Deadly Embrace”

Terminology

- Preemptible vs. Non-preemptible
- Shared vs. Exclusive resource
  - Example of Shared resource: File
  - Example of Exclusive resource: Printer

Terminology ...

- Indefinite postponement
  - Job is continually denied resources needed to make progress
  
  Example: High priority processes keep CPU busy 100% of time, thereby denying CPU to low priority processes
Three Solutions to Deadlock

#1: Mr./Ms. Conservative (Prevention)

"We had better not allocate if it could ever cause deadlock"

Process waits until all needed resource free
Resources underutilized

#2: Mr./Ms. Prudent (Avoidance)

"If resource is free and with its allocation we can still guarantee that everyone will finish, use it."
Better resource utilization
Process still waits

#3: Mr./Ms. Liberal (Detection/Recovery)

"If it’s free, use it -- why wait?"
Good resource utilization, minimal process wait time
Until deadlock occurs...

Names for Three Methods on Last Slide

1) Deadlock Prevention
   - Design system so that possibility of deadlock is avoided a priori
2) Deadlock Avoidance
   - Design system so that if a resource request is made that could lead to deadlock, then block requesting process.
   - Requires knowledge of future requests by processes for resources.
3) Deadlock Detection and Recovery
   - Algorithm to detect deadlock
   - Recovery scheme

4 Necessary Conditions for Deadlock

- Mutual Exclusion
  - Non-sharable resources
- Hold and Wait
  - A process must be holding resources and waiting for others
- No pre-emption
  - Resources are released voluntarily
- Circular Wait

Deadlock Prevention

Deny one or more of the necessary conditions

- Prevent “Mutual Exclusion”
  - Use only sharable resources
  - Impossible for practical systems
Deadlock Prevention ...
- Prevent "Hold and Wait"
  (a) Preallocation: process must request and be allocated all of its required resources before it can start execution
  (b) Process must release all of its currently held resources and re-request them along with request for new resources
  => Very inefficient
  => Can cause "indefinite postponement": jobs needing lots of resources may never run

Deadlock Prevention ...
- Allow "Resource Preemption"
  - Allowing one process to acquire exclusive rights to a resource currently being used by a second process
  => Some resources cannot be preempted without detrimental implications (e.g., printers, tape drives)
  => May require jobs to restart

Deadlock Prevention ...
- Prevent Circular Wait
  - Order resources and
  - Allow requests to be made only in an increasing order

Preventing Circular Wait

Impose an ordering on Resources:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>2</td>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>4</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Request: W X Y Z A B C D

After first 4 requests:

A / W
B / X
C / Y

Process D cannot request resource W without voluntarily releasing Z first

Problems with Linear Ordering Approach

(1) Adding a new resource that upsets ordering requires all code ever written for system to be modified!
(2) Resource numbering affects efficiency
  => A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending sequence

Deadlock Avoidance
- OS never allocates resources in a way that could lead to deadlock
  => Processes must tell OS in advance how many resources they will request
Banker's Algorithm

- Banker's Algorithm runs each time:
  - a process requests resource - Is it Safe?
  - a process terminates - Can I allocate released resources to a suspended process waiting for them?
- A new state is safe if and only if every process can complete after allocation is made
  $\Rightarrow$ Make allocation, then check system state and de-allocate if safe/unsafe

Definition: Safe State

- State of a system
  - An enumeration of which processes hold, are waiting for, or might request which resources
- Safe state
  - No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.
  - Alternatively,
    - No process is deadlocked, and the current state will not lead to a deadlocked state

Deadlock Avoidance

Safe State:

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Process 2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Process 3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Available = 2

Unsafe State:

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Process 2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Process 3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Available = 1

Safe to Unsafe Transition

Current Safe State:

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Maximum Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Process 2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Process 3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Available = 2

Suppose Process 3 requests and gets one more resource

<table>
<thead>
<tr>
<th>User</th>
<th>Current Loan</th>
<th>Maximum Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>User 2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>User 3</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Available = 1

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

\[
\text{Need}[j] = \text{Max}[j] - \text{Allocation}[j].
\]

1. Let Work and Finish be vectors of length \( m \) and \( n \), respectively. Initialize:

\[
\text{Work} = \text{Available} \\
\text{Finish}[j] = \text{false} \text{ for } i = 1, 2, 3, \ldots, n.
\]

\( i = 1; \)

while (\( i <= n \)) Do {

\[
\text{if } (\text{Finish}[j] \land \text{Need}[j] <= \text{Work}) \{
\text{Finish}[j] = \text{true};
\text{Work} = \text{Work} + \text{Allocation};
\}
\]

else \( i++; \)

\}

if (Finish[j] = = true for all \( j \)) return (SAFE);
else return (UNSAFE);
**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 \(T_0\):
  
  \[
  \begin{array}{c|c|c|c}
  Allocation & Max & Available \\
  \hline
  A & B & C & A & B & C \\
  P_0 & 0 & 1 & 0 & 7 & 5 & 3 \\
  P_1 & 2 & 0 & 0 & 3 & 2 & 2 \\
  P_2 & 3 & 0 & 2 & 9 & 0 & 2 \\
  P_3 & 2 & 1 & 1 & 2 & 2 & 2 \\
  P_4 & 0 & 0 & 2 & 4 & 3 & 3 \\
  \end{array}
  \]

**Example (Cont.)**

- The content of the matrix. **Need** is defined to be **Max** – **Allocation**.
  
  Need
  
  \[
  \begin{array}{c|c|c|c}
  A & B & C \\
  P_0 & 7 & 4 & 3 \\
  P_1 & 1 & 2 & 2 \\
  P_2 & 6 & 0 & 0 \\
  P_3 & 0 & 1 & 1 \\
  P_4 & 4 & 3 & 1 \\
  \end{array}
  \]

- The system is in a safe state since the sequence
  
  \(<P_4, P_3, P_2, P_0>\)
  
  satisfies safety criteria.

**Example \(P_i\) Request (1,0,2) (Cont.)**

- Check that **Request** ≤ **Available** (that is, \((1,0,2) \leq (3,3,2)\)
- True.

  Allocation Need Available
  
  \[
  \begin{array}{c|c|c|c}
  A & B & C & A & B & C \\
  P_0 & 0 & 1 & 0 & 7 & 4 & 3 \\
  P_1 & 3 & 0 & 2 & 0 & 2 & 0 \\
  P_2 & 3 & 0 & 1 & 6 & 0 & 0 \\
  P_3 & 2 & 1 & 1 & 0 & 1 & 1 \\
  P_4 & 0 & 0 & 2 & 4 & 3 & 1 \\
  \end{array}
  \]

- Executing safety algorithm shows that sequence
  
  \(<P_4, P_3, P_0, P_1>\)
  
  satisfies safety requirement.
- Can request for \((3,3,0)\) by \(P_4\) be granted?
- Can request for \((0,0,2)\) by \(P_0\) be granted?

**Banker's Algorithm: Summary**

(+) PRO's:
- Deadlock never occurs.
- More flexible & more efficient than deadlock prevention. (Why?)

(-) CON's:
- Must know max use of each resource when job starts.
  
  => No truly dynamic allocation
- Process might block even though deadlock would never occur

**Resource Graphs**

- Graphical model of deadlock
- Nodes:
  
  1) Processes
  
  \[
  \begin{array}{c|c}
  P_I & \rightarrow P_R \\
  \end{array}
  \]

- Edges:
  
  1) Request
  
  \[
  \begin{array}{c|c|c|c}
  P_I & \rightarrow P_R \\
  \end{array}
  \]

- 2) Allocate
  
  \[
  \begin{array}{c|c|c|c}
  P_I & \rightarrow P_R \\
  \end{array}
  \]

**Deadlock Detection**

- Allow deadlock to occur, then recognize that it exists
- Run deadlock detection algorithm whenever locked resource is requested
- Could also run detector in background
Resource Graphs: Example

- P1 holds 2 units of R1
- P1 holds 1 unit of R2
- R1 has a total inventory of 4 units
- P2 holds 1 unit of R1
- P2 requests 1 unit of R2 (and is blocked)

Operations on Resource Graphs: An Overview

1) Process requests resources: Add arc(s)
2) Process acquires resources: Reverse arc(s)
3) Process releases resources: Delete arc(s)

Graph Reductions

- A graph is reduced by performing operations 2 and 3 (reverse, delete arc)
- A graph is completely reducible if there exists a sequence of reductions that reduce the graph to a set of isolated nodes
- A process P is not deadlocked if and only if there exists a sequence of reductions that leave P unblocked
- If a graph is completely reducible, then the system state it represents is not deadlocked

Operations on Resource Graphs: Details

1) P requests resources (Add arc)
   - Precondition:
     - P must have no outstanding requests
     - P can request any number of resources of any type
   - Operation:
     - Add one edge (P, Rj) for each resource copy Rj requested

2) P acquires resources (Reverse arc)
   - Precondition:
     - Must be available units to grant all requests
     - P acquires all requested resources
   - Operation:
     - Reverse all request edges directed from P toward resources

Operations on Resource Graphs: Details ...

3) P releases resources (Delete arc)
   - Precondition:
     - P must have no outstanding requests
     - P can release any subset of resources that it holds
   - Operation:
     - Delete one arc directed away from resource for each released resource

Resource Graphs

- NO.... One sequence of reductions:
  1) P1 acquires 1 unit of R1
  2) P1 releases all resources (finishes)
  3) P2 acquires 2 units of R2
  4) P2 releases all resources (finishes)
Resource Graphs ...

NO.... One sequence of Reductions:
1)  P2 acquires 2 units of R2
2)  P2 releases all resources (finishes)
3)  P1 acquires 2 units of R1
4)  P1 releases all resources (finishes)

Recovering from Deadlock

Once deadlock has been detected, the system must be restored to a non-deadlocked state

1)  Kill one or more processes
   - Might consider priority, time left, etc. to determine order of elimination
2)  Preempt resources
   - Preempted processes must rollback
   - Must keep ongoing information about running processes