Operating Systems
CMPSC 473
Deadlocks
March 3, 2009 - Lecture 13
Instructor: Trent Jaeger
• Last class:
  – Synchronization

• Today:
  – Deadlocks
Definition

• A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

• An event could be:
  – Waiting for a critical section
  – Waiting for a condition to change
  – Waiting for a physical resource
Necessary Conditions for a Deadlock

- **Mutual exclusion**: The requesting process is delayed until the resource held by another is released.
- **Hold and wait**: A process must be holding at least 1 resource and must be waiting for 1 or more resources held by others.
- **No preemption**: Resources cannot be preempted from one and given to another.
- **Circular wait**: A set \( (P_0, P_1, \ldots, P_n) \) of waiting processes must exist such that \( P_0 \) is waiting for a resource held by \( P_1 \), \( P_1 \) is waiting for \( \ldots \) by \( P_2 \), \( \ldots \) \( P_n \) is waiting for \( \ldots \) held by \( P_0 \).
Resource Allocation Graph

- Vertices (V) = Processes (Pi) and Resources (Rj)
- Edges (E) = Assignments (Rj->Pi, Rj is allocated to Pi) and Request (Pi->Rj, Pi is waiting for Rj).
- For each Resource Rj, there could be multiple instances.
- A requesting process can be granted any one of those instances if available.
An example

```
R1

P1
  ↓
  ↓
R2

P2

R3

P3
  ↓
  ↓
R4
```
If there is a deadlock, there will be a cycle (Necessary Condition).
A cycle is NOT a sufficient condition
Strategies for Handling Deadlocks

• Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.

• Detect and recover after its occurrence.

• Avoidance by careful resource allocation

• Prevention by structurally negating one of the four necessary conditions
Strategies for Handling Deadlocks

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.

- Detect and recover after its occurrence.

- Avoidance by careful resource allocation

- Prevention by structurally negating one of the four necessary conditions
Deadlock Prevention

• Note that all 4 necessary conditions need to hold for deadlock to occur.

• We can try to disallow one of them from happening:
  – Mutual exclusion: This is usually not possible to avoid with many resources.
  – No preemption: This is again not easy to address with many resources. Possible for some resources (e.g. CPU)
  – Hold and Wait:
    • Allow at most 1 resource to be held/requested at any time
    • Make sure all requests are made at the same time.
    • …
– Circular Wait

• Number the resources, and make sure requests are always made in increasing/decreasing order.

• Or make sure you are never holding a lower numbered resource when requesting a higher numbered resource.
Strategies for Handling Deadlocks

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.

- Detect and recover after its occurrence.

- Avoidance by careful resource allocation

- Prevention by structurally negating one of the four necessary conditions
Deadlock Avoidance

- Avoid actions that may lead to a deadlock.
- Visualize the system as a state machine moving from 1 state to another as each instruction is executed.
- A state can be: safe, unsafe or deadlocked.
- Safe state is one where
  - it is not a deadlocked state
  - there is some sequence by which all requests can be satisfied.
- To avoid deadlocks, we try to make only those transitions that will take you from one safe state to another.
- This may be a little conservative, but it avoids deadlocks.
State Transitions

- Safe
- Unsafe
- Deadlocked

- Start
- End
Safe State

1 resource with 12 units of that resource available.

Current State: Free = (12 - (5 + 2 + 2)) = 3

<table>
<thead>
<tr>
<th></th>
<th>Max. Needs</th>
<th>Currently Allocated</th>
<th>Still Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

This state is **safe** because, there is a sequence (P1 followed by P0 followed by P2) by which max needs of each process can be satisfied. This is called the reduction sequence.
Unsafe State

What if P2 requests 1 more and is allocated 1 more?

Only P1 can be reduced. If P0 and P2 then come and ask for their full needs, the system can become deadlocked. Hence, by granting P2’s request for 1 more, we have moved from a safe to unsafe state.

Deadlock avoidance algorithm will NOT allow such a transition, and will not grant P2’s request immediately.
- Deadlock avoidance essentially allows requests to be satisfied only when the allocation of that request would lead to a safe state.
- Else do not grant that request immediately.
Banker’s algorithm for deadlock avoidance

- When a request is made, check to see if after the request is satisfied, there is a (at least one!) sequence of moves that can satisfy all possible requests. i.e. the new state is safe.
- If so, satisfy the request, else make the request wait.
Checking if a state is safe  
(Generalization for “M” resources)

N processes and M resources

Data Structures: 
MaxNeeds[N][M]; 
Allocated[N][M]; 
StillNeeds[N][M]; 
Free[M]; 
Temp[M]; 
Done[N]; 

while () { 
    Temp[j] = Free[j] for all j 
    Find an i such that 
        a) Done[i] = False 
        b) StillNeeds[i,j] <= Temp[j] 
    if so { 
        Temp[j] += Allocated[i,j] for all j 
        Done[i] = TRUE /* release Allocated[i] */ 
    } 
    else if Done[i] = TRUE for all i then state is safe 
    else state is unsafe 
} 

M*N^2 steps to detect if a state is safe!
An example

5 processes, 3 resource types A (10 instances), B (5 instances), C (7 instances)

<table>
<thead>
<tr>
<th>MaxNeeds</th>
<th>Allocated</th>
<th>StillNeeds</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 7 5 3</td>
<td>P0 0 1 0</td>
<td>P0 7 4 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P1 3 2 2</td>
<td>P1 2 0 0</td>
<td>P1 1 2 2</td>
<td></td>
</tr>
<tr>
<td>P2 9 0 2</td>
<td>P2 3 0 2</td>
<td>P2 6 0 0</td>
<td></td>
</tr>
<tr>
<td>P3 2 2 2</td>
<td>P3 2 1 1</td>
<td>P3 0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4 4 3 3</td>
<td>P4 0 0 2</td>
<td>P4 4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

This state is safe, because there is a reduction sequence <P1, P3, P4, P2, P0> that can satisfy all the requests.

Exercise: Formally go through each of the steps that update these matrices for the reduction sequence.
If P1 requests 1 more instance of A and 2 more instances of C can we safely allocate these? - Note these are all allocated together and we denote this set of requests as (1,0,2)

If allocated the resulting state would be:

<table>
<thead>
<tr>
<th>MaxNeeds</th>
<th>Allocated</th>
<th>StillNeeds</th>
<th>Free</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>P0</td>
<td>7  5  3</td>
<td>0  1  0</td>
<td>7  4  3</td>
</tr>
<tr>
<td>P1</td>
<td>3  2  2</td>
<td>3  0  2</td>
<td>0  2  0</td>
</tr>
<tr>
<td>P2</td>
<td>9  0  2</td>
<td>3  0  2</td>
<td>6  0  0</td>
</tr>
<tr>
<td>P3</td>
<td>2  2  2</td>
<td>2  1  1</td>
<td>0  1  1</td>
</tr>
<tr>
<td>P4</td>
<td>4  3  3</td>
<td>0  0  2</td>
<td>4  3  1</td>
</tr>
</tbody>
</table>

This is still safe since there is a reduction sequence <P1,P3,P4,P0,P2> to satisfy all the requests. (work this out!)
Hence the requested allocations can be made.
After this allocation, P0 then makes a request for (0,2,0). If granted the resulting state would be:

<table>
<thead>
<tr>
<th>MaxNeeds</th>
<th>Allocated</th>
<th>StillNeeds</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 7 5 3</td>
<td>P0 0 3 0</td>
<td>P0 7 2 3</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P1 3 2 2</td>
<td>P1 3 0 2</td>
<td>P1 0 2 0</td>
<td></td>
</tr>
<tr>
<td>P2 9 0 2</td>
<td>P2 3 0 2</td>
<td>P2 6 0 0</td>
<td></td>
</tr>
<tr>
<td>P3 2 2 2</td>
<td>P3 2 1 1</td>
<td>P3 0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4 4 3 3</td>
<td>P4 0 0 2</td>
<td>P4 4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

This is an UNSAFE state.

So this request should NOT be granted.
Strategies for Handling Deadlocks

• Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.

• Detect and recover after its occurrence.

• Avoidance by careful resource allocation

• Prevention by structurally negating one of the four necessary conditions
Deadlock Detection and Recovery

- If there is only 1 instance of each resource, then a cycle in the resource-allocation graph is a “sufficient” condition for a deadlock, i.e. you can run a cycle-detection algorithm to detect a deadlock.

- With multiple instances of each resource, ???
Detection Algorithm

N processes, M resources

Data structures:
  Free[M];
  Allocated[N][M];
  Request[N][M];
  Temp[M];
  Done[N];

Basic idea is that there is at least 1 execution that will unblock all processes.

M*N^2 algorithm!

1. Temp[i] = Free[i] for all i
   Done[i] = FALSE unless there is no resources allocated to it.

2. Find an index i such that both
   (a) Done[i] == FALSE
   (b) Request[i] <= Temp (vector comp.)
   If no such i, go to step 4.

3. Temp = Temp + Allocated[i] (vector add)
   Done[i]= TRUE; /* release Allocated[i] */
   Go to step 2.

4. If Done[i]=FALSE for some i, then there is a deadlock.
**Example**

5 processes, 3 resource types A (7 instances), B (2 instances), C (6 instances)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This state is **NOT** deadlocked.

By applying algorithm, the sequence <P0, P2, P3, P1, P4> will result in Done[i] being TRUE for all processes.
If on the other hand, P2 makes an additional request for 1 instance of C

State is:

<table>
<thead>
<tr>
<th>Allocated</th>
<th>Request</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 0 1 0</td>
<td>P0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>P1 2 0 0</td>
<td>P1 2 0 2</td>
<td></td>
</tr>
<tr>
<td>P2 3 0 3</td>
<td>P2 0 0 1</td>
<td></td>
</tr>
<tr>
<td>P3 2 1 1</td>
<td>P3 1 0 0</td>
<td></td>
</tr>
<tr>
<td>P4 0 0 2</td>
<td>P4 0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

This is deadlocked!
Even though P0 can proceed, the other 4 processes are deadlocked.
Recovery

• Once deadlock is detected what should we do?
  – Preempt resources (whenever possible)
  – Kill the processes (and forcibly remove resources)
  – Checkpoint processes periodically, and roll them back to last checkpoint (relinquishing any resources they may have acquired since then).
Summary

• Deadlocks
  – Necessary and sufficient conditions
    • Resource allocation graph
  – Strategies
    • Ignore
    • Prevention
      – Safe States
    • Avoidance
    • Detection and recovery
• Next time: Memory