Deadlock (CSE 422S)

Ken Wong
Washington University
kenw@wustl.edu
www.arl.wustl.edu/~kenw

Deadlock

- Definition
  » The permanent blocking of a set of processes

- Necessary Conditions
  » Mutual Exclusion: Only one process at a time can use a resource
  » Hold and Wait: A process holding one resource is waiting to acquire additional resources held by other processes
  » No Preemption: No resource can be forcibly taken from a processing holding it

- Sufficient Condition
  » Circular Waiting: There exists a permanent, circular sequence of processes such that each process is waiting for a resource held by the preceding process

Approaches for Handling Deadlock

- Ignore Deadlock
- Deadlock Prevention
  » Prevent one of the necessary/sufficient conditions

- Deadlock Avoidance
  » Allow the 3 necessary conditions
  » Dynamically make choices (based on knowledge of future requests) to avoid deadlock (i.e., find a safe path)

- Deadlock Detection
  » Periodically run an algorithm to detect circular waiting
  » After detecting deadlock, run a recovery algorithm to remove the deadlock

Resource Allocation Graph

- Resource-Process Arc: Resource has been requested, granted and is being held
- Process-Resource Arc: Process is blocked waiting for resource

Interpretation
» No Cycle ➔ There is no deadlock
» Cycle ➔ There may be deadlock

Processes
Held
Requests
Files
Held
F1
F2
A
B
Deadlock Prevention

- Mutual Exclusion
  - Not required for sharable resources, but must hold for non-sharable ones
- Never Hold and Wait
  - Block a process until it can acquire all of its requested resources at once
- Allow Preemption
  - Process must release all resources when it is denied a resource request
- Prevent Circular Waiting
  - Define a total ordering when allocating resource types
  - Require that resources be requested in increasing order

Never Hold and Wait
Block a process until it can acquire all of its requested resources at once

Deadlock Avoidance (Banker's Alg.)

- Each process must state resource demands a priori
  - Bank lends out fixed amount of money
  - Customer gets fixed line of credit; borrows and pays back part of loan over time
- Safe State: A state in which there is at least one resource request sequence in which all processes can run to completion
  - Unsafe State: There is only a potential for deadlock
- Always ensure the system is in a safe state
  - When there is a request, update the system state as if it is granted
  - If the state is safe, grant the request; else block the process until it is safe to grant the request
- When a process gets all of its resources, it must return them in finite time

Example (Banker's Algorithm)

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Current State

<table>
<thead>
<tr>
<th>Quantity [Q(j)]:</th>
<th>Unallocated [U(j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Max Demand [D(i,j)]:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation [A(i,j)]:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

An Unsafe State

| Each process still needs 1 unit of R1 |

<table>
<thead>
<tr>
<th>Quantity [Q(j)]:</th>
<th>Unallocated [U(j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand [D(i,j)]:</th>
<th>Allocation [A(i,j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
</tr>
</tbody>
</table>

Only potential for deadlock... not certainty
Example of Banker's Algorithm

**Is this a safe state?**

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
</tr>
<tr>
<td>9   3   6</td>
<td>0   1   1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
</tr>
<tr>
<td>P1  3   2   2</td>
<td>P1  1   0   0</td>
</tr>
<tr>
<td>P2  6   1   3</td>
<td>P2  6   1   2</td>
</tr>
<tr>
<td>P3  3   1   4</td>
<td>P3  2   1   1</td>
</tr>
<tr>
<td>P4  4   2   2</td>
<td>P4  0   0   2</td>
</tr>
</tbody>
</table>

Example of Banker's Algorithm

**Safe Order:** P2, P1, P3, P4

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
</tr>
<tr>
<td>9   3   6</td>
<td>0   1   1</td>
<td>0   0   0</td>
<td>0   0   0</td>
</tr>
</tbody>
</table>

| P1  3   2   2    | P2  6   0   0    | P4  0   0   2       |
| P3  3   1   4    | P3  2   1   1    | P2  6   0   0       |
| P4  4   2   2    | P1  1   0   0    | P3  2   1   1       |

The Banker's Algorithm

**Process i requests R(*) resources of each type**

if (A(i,*) + R(*) > D(i,*)) ... Error: Max demand exceeded else {
  U’(*) = U(*) - R(*); // trial allocation
  A’(i,*) = A(i,*) + R(*);
  if (isSafe(U’, D, A’)) { // allocate
    A(i,*) = A’(i,*);
    U(*) = U’(*);
  } else . . . Restore old state and suspend process . . .
}

**isSafe(U’,D,A’)**

- True if there is a possible sequence (path) of process completions that includes all processes

P2 Can Complete

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
</tr>
<tr>
<td>9   3   6</td>
<td>0   1   1</td>
<td>0   0   0</td>
</tr>
</tbody>
</table>

| P1  3   2   2    | P2  6   0   0    | P3  2   1   1       |

- P2 returns resources and demand goes to 0

P2 Can Complete

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
<td>R1  R2  R3</td>
</tr>
<tr>
<td>9   3   6</td>
<td>0   1   1</td>
<td>0   0   0</td>
</tr>
</tbody>
</table>

| P1  3   2   2    | P2  6   0   0    | P3  2   1   1       |

- P2 returns resources and demand goes to 0
**The Test for Safety**

```java
boolean isSafe(U, D, A) {
    Utmp(*) = U(*);
    Rest = . . . Set of all processes . . .
    safePath = True;
    while (safePath) {
        if (there is a process i such that
            D(i,*) - A(i,*) <= Utmp(*)) {
            Utmp(*) = Utmp(*) + A(i,*);
            Rest = Rest - { i };   
        } else safePath = False;
    }
    return (isEmpty(Rest));
}
```

---

**Pros and Cons of Deadlock Avoidance**

**Pros**
- Less restrictive than deadlock prevention
- Not necessary to preempt/rollback processes

**Cons**
- Must state in advance the max resource demand D(*,*)
- There must be a fixed amount of resources Q(*,*) to allocate
- Execution order of the process under consideration must not be constrained by synchronization requirements of other processes
- Ability to finish is completely determined by resource demands

---

**Deadlock Detection**

**Idea**
- Determine if all requests can eventually be satisfied
- Simulate request grants and completing processes (that return resources)
- Mark processes that can complete (are NOT deadlocked)
- A deadlock exists if and only if there are unmarked processes at the end of the algorithm

**Comments**
- Does not guarantee that deadlock will not occur (depends on order in which requests are granted)
- Just determines if deadlock currently exists
- Modify isSafe( ) function to get isDeadlocked( )

---

**Deadlock Detection Algorithm**

**Process i requests R(i,*) additional resources**

```
Utmp(*) = U(*);
Rest = . . . Set of all processes with A(i,*) != 0 . . .
safePath = True;
while (safePath) {
    // find a process whose request can be satisfied
    if (there exists a process i such that
        R(i,*) <= Utmp(*)) {
        // simulate process i completing
        Utmp(*) = Utmp(*) + A(i,*)
        Rest = Rest - { i };
    } else safePath = False;
}
```

... Each process in Rest is deadlocked ...

**Looks like isSafe( ) function!**