Deadlocks: Avoidance – Detection - Recovery

Summer 2013
Cornell University
Today

- Can we avoid a deadlock? Can we detect and recover from a deadlock?
- Safe state
- Deadlock avoidance
- Banker's Algorithm
- Deadlock detection
- Deadlock recovery
Deadlock Avoidance

- The **system knows** the complete sequence of requests and releases for each process.
- The **system decides** for each request whether or not the process should wait in order to avoid a deadlock.
- Each **process declare** the maximum number of resources of each type that it may need.
- The system should always be at a **safe state**.
- Safe state $\rightarrow$ no deadlock
  - the inverse is not always true.
Safe State

- A state is said to be **safe**, if it has a process sequence
  - \{P_1, P_2, \ldots, P_n\}, such that for each \(P_i\),
  - the resources that \(P_i\) can still request can be satisfied by the currently available resources plus the resources held by all \(P_j\), where \(j < i\).

- State is safe because OS can definitely avoid deadlock
  - by blocking any new requests until safe order is executed

- This avoids **circular wait** condition
  - Process waits until safe state is guaranteed
Safe State

• Suppose there are 12 tape drives

<table>
<thead>
<tr>
<th></th>
<th>Max Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

• 3 drives remain

• Current state is safe because a safe sequence exists: \(<p1,p0,p2>\)
  • \(p1\) can complete with current resources
  • \(p0\) can complete with current + \(p1\)
  • \(p2\) can complete with current + \(p1+p0\)

• If \(p2\) requests 1 drive, then it must wait to avoid unsafe state.
Resource-Allocation Graph Algorithm

- Works only if each resource type has **one** instance.

- Algorithm:
  - Add a **claim edge**, Pi → Rj, if Pi can request Rj in the future
  - Represented by a dashed line in graph
  - A request Pi → Rj can be granted only if:
    - Adding an assignment edge Rj → Pi does not introduce cycles
      - (since cycles imply unsafe state)
Banker's Algorithm

- Applicable to resources with **multiple instances**.
- Less efficient than the resource-allocation graph scheme.
- Each process declares its needs (number of resources)
- When a process requests a set of resources:
  - Will the system be at a safe state after the allocation?
    - Yes → Grant the resources to the process.
    - No → Block the process until the resources are released by some other process.
Banker's Algorithm

n: integer  # of processes
m: integer  # of resource-types
available[1..m] available[i] is # of avail resources of type i
max[1..n,1..m] max demand of each Pi for each Ri
allocation[1..n,1..m] current allocation of resource Rj to Pi
need[1..n,1..m] max # resource Rj that Pi may still request
Banker's Algorithm

- If request[i] > need[i] then
  - error (asked for too much)
- If request[i] > available[i] then
  - wait (can't supply it now)
- Resources are available to satisfy the request
  - Let's assume that we satisfy the request. Then we would have:
    - available = available - request[i]
    - allocation[i] = allocation[i] + request[i]
    - need[i] = need[i] - request[i]
  - Now, check if this would leave us in a safe state:
    - If yes, grant the request,
    - If no, then leave the state as is and cause process to wait.
Banker's Algorithm

- Safety Algorithm

work[1..m] = available  /* how many resources are available */
finish[1..n] = false (for all i)  /* none finished yet */

**Step 1:**
Find an i such that finish[i]=false and need[i] <= work  /* find a proc that can complete*/
/* its request now */

If no such i exists, go to step 3  /* we’re done */

**Step 2:** Found an i:
finish [i] = true  /* done with this process */
work = work + allocation [i]  /* assume this process were to finish, */
/* and its allocation back to the available list */
go to step 1

**Step 3:** If finish[i] = true for all i, the system is safe. Else Not
## Banker’s Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</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>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- This is a safe state: safe sequence <P1, P3, P4, P2, P0>

- Suppose that P1 requests (1,0,2)
  - Add it to P1’s allocation and subtract it from Available.
Banker’s Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</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>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- This is still safe: safe seq <P1, P3, P4, P0, P2>
- In this new state, P4 requests (3,3,0)
  - Not enough available resources.
- P0 requests (0,2,0)
  - Let’s check resulting state...
## Banker’s Algorithm: Example

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</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>P0</td>
<td>0 3 0</td>
<td>7 5 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- This is unsafe state (why?).
- So P0’s request will be denied.
The story so far..

- We saw that you can **prevent** deadlocks.
  - By **negating** one of the four necessary conditions.
- We saw that the OS can schedule processes in a careful way so as to **avoid** deadlocks.
  - Using a resource allocation graph.
  - **Banker’s algorithm**.
- What are the downsides to these?
Deadlock Detection

- If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - **Detection**: finding out if deadlock has occurred
    - Keep track of *resource allocation* (who has what)
    - Keep track of *pending requests* (who is waiting for what)
  - **Recovery**: resolve the deadlock
Using the RAG Algorithm to detect deadlocks

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Use a **wait-for graph:**
  - Collapse resources
  - An edge Pi → Pk exists only if RAG has Pi → Rj & Rj → Pk
  - Cycle in wait-for graph → deadlock!
Detection Algorithm

- Multiple instances per resource.
- Data structures:

  n: integer  # of processes
  m: integer  # of resource-types
  available[1..m]  available[i] is # of avail resources of type i
  request[1..n,1..m]  current demand of each Pi for each Ri
  allocation[1..n,1..m]  current allocation of resource Rj to Pi
  finish[1..n]  true if Pi’s request can be satisfied

Let request[i] be vector of # instances of each resource Pi wants
Detection Algorithm

- Multiple instances per resource.

- Data structures:

  - n: integer   # of processes
  - m: integer   # of resource-types
  - available[1..m] available[i] is # of avail resources of type i
  - request[1..n,1..m] current demand of each Pi for each Ri
  - allocation[1..n,1..m] current allocation of resource Rj to Pi
  - finish[1..n] true if Pi’s request can be satisfied

Let request[i] be vector of # instances of each resource Pi wants
Detection Algorithm

- work[] = available[]
- for all i < n, if allocation[i] != 0
  - then finish[i] = false else finish[i] = true
- find an index i such that:
  - finish[i] = false;
  - request[i] <= work
- if no such i exists, go to 7.
- work = work + allocation[i]
- finish[i] = true, go to 3
- if finish[i] = false for some i,
  - then system is deadlocked with Pi in deadlock
Detection Algorithm: Example

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<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>

- The system is not in a deadlocked state.
- What will happen if P2 makes an additional request for a instance of type C?
Deadlock Recovery

- **Killing** one/all deadlocked processes
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- **Preempt** resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Rollback (partial or total)
  - Starvation (prevent a process from being executed)
Today

- Can we avoid a deadlock? Can we detect and recover from a deadlock?
- Safe state
- Deadlock avoidance
- Banker's Algorithm
- Deadlock detection
- Deadlock recovery