Frequently asked questions from the previous class survey

- Prevention or avoidance: Which is better?
- Does numbering prevent starvation in the Dining Philosopher’s problem?
  - Deadlock implies starvation
  - Starvation does not imply deadlock

Topics covered in this lecture

- Deadlock Avoidance
  - Banker’s Algorithm
- Deadlock Detection
  - And ... recovery
- Other issues relating to deadlocks

Banker’s Algorithm

- Designed by Dijkstra in 1965
- Modeled on a small-town banker
  - Customers have been extended lines of credit
  - Not ALL customers will need their maximum credit immediately
- Customers make loan requests from time to time

Crux of the Banker’s Algorithm

- Consider each request as it occurs
  - See if granting it is safe
  - If safe: grant it,  If unsafe: postpone
- For safety banker checks if he/she has enough to satisfy some customer
  - If so, that customer’s loans are assumed to be repaid
  - Customer closest to limit is checked next
  - If all loans can be repaid; state is safe: loan approved
Banker’s Algorithm: Managing the customers. Banker has only reserved 10 units instead of 22

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 10

SAFE

UNSAFE

A customer may not need the entire credit line. But the banker cannot count on this behavior.

There is ONLY ONE resource: Credit

March 23, 2017

CS370: Operating Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

---

Banker’s algorithm: Crux

- Declare maximum number of resource instances needed
- Cannot exceed resource thresholds
- Determine if resource allocations leave system in a safe state

Data Structures: \( n \) is the number of processes and \( m \) is the number of resource types

- Available: Vector of length \( m \)
  - Number of resources for each type
    - Available\([i]\) = \( k \)

- Max: \( n \times m \) matrix
  - Maximum demand for each process (in each row)
  - Max\([i,j]\)= \( k \)
  - Process \( P_i \) may request at most \( k \) instances of \( R_j \)

- Need: \( n \times m \) matrix
  - Resource instances needed for each process (each row)
  - Need\([i,j]\)=k
  - Process \( P_i \) may need \( k \) more instances of \( R_j \)

Vectors identifying a process’ resource requirements: Rows in the matrices

- Allocation\(_i\)
  - Resource instances allocated for process \( P_i \)
- Need\(_i\)
  - Additional resource instances process \( P_i \) may still request

Banker’s algorithm: Notations

- \( X \) and \( Y \) are vectors of length \( m \)
- \( X \leq Y \) if-and-only-if
  \( X[i] \leq Y[i] \) for all \( i=1,2,\ldots,m \)
- \( X = (1,7,3,2) \) and \( Y = (0,3,2,1) \)
  - So, \( Y \leq X \)
  - Also \( Y < X \) if \( Y \leq X \) and \( Y \neq X \)
Banker’s Algorithm: Resource-request

- Request$_i$: Request vector for process P$_i$
- Request$_i$$[j]=k$
- Process P$_i$ wants k instances of R$_j$

Bankers Algorithm: Resource-request

- Request$_i \leq$ Need$_i$
- Request$_i \leq$ Available
- Available = Available – Request$_i$
- Allocation$_i$ = Allocation$_i$ + Request$_i$
- Need$_i$ = Need$_i$ – Request$_i$

Bankers Algorithm: Safety

Initialize Work = Available

Find i such that:
- Finish[i]=false & Need$_i$ \leq Work

Work = Work + Allocation$_i$
- Finish[i]=true

for all i
- if {Finish[i] = true}
  YES

Safe state

NO Unsafe state

Bankers Algorithm: Example

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

Suppose process P1 requests 1 A, and 2 C: Request$_1=$ (1,0,2)
- Request$_1 \leq$ Available
- Pretend request was fulfilled
- <P1, P3, P4, P2, P0> satisfies safety criteria

Request$_4$ = (3,0,0) from process P4 cannot be granted: resources unavailable
- Few systems use this for avoiding deadlocks
Single instance of EACH resource type

- Use wait-for graph
- Variant of the resource allocation graph
- Deadlock exists if there is a cycle in the graph
- Transformation
  1. Remove resource nodes
  2. Collapse appropriate edges

What the edges in the wait-for graph imply

- $P_i \rightarrow P_j$ implies $P_i$ is waiting for a resource held by $P_j$
- $P_i \rightarrow P_j$ only if resource allocation graph has
  1. $P_i \rightarrow R_q$ and
  2. $R_q \rightarrow P_j$ for some resource $R_q$

Transforming a resource allocation graph into a wait-for graph
Deadlock detection for multiple instances of a resource type

- Wait-for graph is not applicable
- Approach uses data structures similar to Banker’s algorithm

Data Structures: n is number of processes, m is number of resource types

- Available: Vector of length m
  - Number of resources for each type
- Allocation: n x m matrix
  - Resource instances allocated for each process
- Allocation[i,j]=k
  - Process Pi currently allocated k instances of Rj
- Request: n x m matrix
  - Current request for each process
- Request[i,j]=k
  - Process Pi requests k more instances of Rj

Deadlock detection: Initialization

Work and Finish are vectors of length m & n

Work = Available
if (Allocation[i] ≠ 0) {
  Finish[i] = false;
} else {
  Finish[i] = true;
}

Deadlock detection: Usage

- How often will the deadlock occur?
- How many processes will be affected when it happens?
Frequency of invoking deadlock detection

- Resources allocated to deadlocked process idle
- Until the deadlock can be broken
- Deadlocks occur only when process makes a request
- Significant overheads to run detection per request
- Middle ground: Run at regular intervals

Recovery from deadlock

- Automated or manual

  OPTIONS
  - Break the circular wait: Abort processes
  - Preempt resources from deadlocked process(es)

Breaking circular wait: Process termination

- Abort all deadlocked processes
- Abort processes one at a time
  - After each termination, check if deadlock persists
- Reclaim all resources allocated to terminated process

Terminating a Process

- Process may be in the midst of something
  - Updating files, printing data etc
- Abort process whose termination will incur minimum costs
  - Policy decision similar to scheduling decisions

Factors determining process termination

- Priority
- How long has the process been running?
  - How much longer?
- Number and types of resources used
  - How many more needed?
- Interactive or batch
Deadlock recovery: Resource preemption

For a set of deadlocked processes:

- **Preempt resources from some process**
- **Give resources to some other process**

Deadlock persists

- **Preempt resources from some process**
- **Give resources to some other process**

Deadlock broken

DONE

Deadlock recovery through rollbacks

- **Checkpoint** process periodically
- **Contains memory image and resource state**
- **Deadlock detection tells us which resources are needed**
- **Process owning a needed resource**
  - **Rolled back** to before it acquired needed resource
  - Work done since rolled back checkpoint discarded
  - **Assign resource to deadlocked process**

Resource preemption: Issues

- **Selecting a victim**
  - Which resource and process
  - Order of preemption to minimize cost
- **Starvation**
  - Process can be selected for preemption *finite* number of times

Other issues

- **Two-phase locking**
  - Used in database systems
  - Operation involves requesting locks on several records and updating all the locked records
  - When multiple processes are running?
    - Possibility of deadlocks

- **First phase**
  - Process tries to acquire all the locks it needs, one at a time
  - If successful: start second-phase
  - If some record is already locked?
    - Release all locks and start the first phase all over

- **Second-phase**
  - Perform updates and release the locks
Communication Deadlocks

- Process A sends a request message to process B
  - Blocks until B sends a reply back
- Suppose, that the request was lost
  - A is blocked waiting for a reply
  - B is blocked waiting for a request to do something
  - Communication deadlock

Communication deadlocks

- Cannot be prevented by ordering resources (there are none)
  - Or avoided by careful scheduling (no moments when a request can be postponed)
- Solution to breaking communication deadlocks?
  - Timeouts
    - Start a timer when you send a message to which a reply is expected.

Livelocks

- Polling (busy waits) used to enter critical section or access a resource
  - Typically used for a short time when overhead for suspension is considered greater
- In a livelock two processes need each other’s resource
  - Both run and make no progress, but neither process blocks
  - Use CPU quantum over and over without making progress

Livelocks do occur

- If fork fails because process table is full
  - Wait for some time and try again
- But there could be a collection of processes each trying to do the same thing

The contents of this slide-set are based on the following references