Topics covered in this lecture

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

A deadlock-prone system can be in one of three states: safe, unsafe, and deadlocked

- Safe state: For any possible sequence of resource requests, there is at least one safe sequence of processing the requests that eventually succeeds in granting all pending and future requests
- Unsafe state: There is at least one sequence of future resource requests that leads to deadlock
- In a deadlocked state, the system has at least one deadlock

A system in a safe state controls its own destiny

- For any workload, it can avoid deadlock by delaying the processing of some requests
- Once the system enters an unsafe state, it may not be able to avoid deadlock
- In particular, the Banker's Algorithm (that we will look at next) delays any request that takes it from a safe to an unsafe state.

Frequently asked questions from the previous class survey

- Prevention or avoidance: Which is better?
- Delay durations when “stalling” requests: how long is too long?
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

Banker’s Algorithm: Managing the customers. Banker has only reserved 10 units instead of 22

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Available</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

There is ONLY ONE resource – Credit

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

Crux of the Banker’s Algorithm: Data Structures [Overview]

- Available: Vector of length $m$
  - Number of resources for each type
  - $\text{Available}(i) = k$
- Max: $n \times m$ matrix
  - Maximum demand for each process [in each row]
  - $\text{Max}(i, j) = k$
  - Process $P_i$ may request at most $k$ instances of $R_j$

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

- **Allocation**: \( n \times m \) matrix
  - Resource instances allocated for each process (each row)
  - \( \text{Allocation}[i,j]=k \) => Process \( P_i \) currently allocated \( k \) instances of \( R_j \)

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

---

**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 \( i=1,2,...,m \)
- \( X = \{1,7,3,2\} \) and \( Y = \{0,3,2,1\} \)
  - \( X \leq Y \)
  - Also \( Y < X \) if \( Y \leq X \) and \( Y \neq X \)

---

**Banker’s Algorithm: Resource-request**

- **Request**:
  - Request vector for process \( P_i \)
  - \( \text{Request}[j]=k \) => Process \( P_i \) wants \( k \) instances of \( R_j \)

- **Availability**:
  - \( \text{Available} = \text{Available} - \text{Request} \)
  - \( \text{Allocation} = \text{Allocation} + \text{Request} \)
  - \( \text{Need} = \text{Need} - \text{Request} \)

- **Safety**:
  - Initialize \( \text{Work} = \text{Available} \)

  - Find \( i \) such that:
    - \( \text{Finish}[i] = \text{false} \) and \( \text{Need}[i] \leq \text{Work} \)

  - Work = Work + Allocation, \( \text{Finish}[i] = \text{true} \)

  - For all \( i \), if \( \text{Finish}[i] = \text{true} \)

  - Unsafe state

  - Safe state
Bankers 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>

<P1, P3, P4, P2, P0> satisfies safety criteria

Suppose process P1 requests 1 A, and 2 C. Request1 = (1,0,2)
Request1 ≤ Available
True; request was fulfilled

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.18

Bankers 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>1 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>

<P1, P3, P4, P0> satisfies safety criteria

Suppose process P4 requests 3 A, 0 B, 0 C. Request4 = (3,0,0)
Process P4 cannot be granted resources unavailable

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.38

Bankers 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 3 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>1 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>

None of the processes can now satisfy their max resource needs.

Request2 = (0,2,0) from process P0 cannot be granted: unsafe state

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.55

Bankers Algorithm: Practical implications

Understanding the Banker’s Algorithm can help in designing simple solutions for specific problems

Banker’s Algorithm to devise a rule for thread safe acquisition of a pair of locks, A and B, with mutually recursive locking?

- Suppose a thread needs to acquire locks A and B, in that order, while another thread needs to acquire lock B first, then A

**RULE:** A thread is always allowed to acquire its second lock provided the other thread does not already hold its first lock

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.53

Bankers Algorithm: Practical implications

- Processes rarely know in advance about their maximum resource needs
- Number of processes managed by the kernel is not fixed
- Varies dynamically
- Resources thought to be available can vanish

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.53

DEADLOCK DETECTION

October 18, 2018
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L18.24
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

- \( P_i \rightarrow P_j \)
  - Process \( P_i \) 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: \( n \times m \) matrix
- Resource instances allocated for each process
- Allocation\( [i,j] = k \) : \( i \) instances of \( R_j \)

- Request: \( n \times m \) matrix
- Current request for each process
- Request\( [i,j] = k \) : \( i \) requests \( k \) more instances of \( R_j \)

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
- Middleground: Run at regular intervals

RECOVERY FROM DEADLOCK

- 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
- Middleground: 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 processes

**Deadlock persists**

**Unbroken**

**Solved**

### 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
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

---

**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

---

Two-Phase Locking

- **First phase**
  - Process tries to acquire all the locks it needs, one at a time
  - 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