Announcements

- Assignment 1:
  - Average: 80.5
  - Standard Deviation: 24

- Important clarification for assignment 3 online regarding the extra credit part (fairness policy).
Chapter 8: Deadlocks

- Background.
- System Model.
- Deadlock Conditions.
- Methods for Handling Deadlocks.
- Deadlock Prevention.
- Deadlock Avoidance.
- Deadlock Detection.
- Recovery from Deadlock.
Background

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Example:
- System has 2 tape drives.
- $P_1$ and $P_2$ each hold one tape drive and each needs another one.

Example:
- Semaphores $A$ and $B$, initialized to 1:
  
  $P_0$
  
  - Execute first $\rightarrow \text{wait}(A)$
  - Execute third $\rightarrow \text{wait}(B)$

  $P_1$
  
  - $\text{wait}(B) \leftarrow \text{Execute second}$
  - $\text{wait}(A) \leftarrow \text{Execute fourth}$
Traffic Deadlock

- Very heavy fines in US if you block an intersection. If you can’t cross to the other side, don’t enter intersection.
System Model

- Resource *types* $R_1, R_2, \ldots, R_m$: CPU cycles, memory space, I/O devices, printers.
- Each resource type $R_i$ has $W_i$ *instances*: 2 printers, 5 item containers in a buffer.
- Each process utilizes a resource as follows:
  1. Allocate (request/acquire).
  2. (Hold/own and) use.
- A single, *atomic* allocation request can be for:
  - Any one instance of one resource.
  - Any number of instances of one resource.
  - Any number of instances of many resources (different number for each resource).
Deadlock Conditions

- **Mutual exclusion**: only one process at a time can use a resource instance, e.g. a printer.

- **Hold and wait**: a process holding at least one resource instance is waiting to acquire additional resource instances, which are currently held by or accumulated for other processes:
  - Held by: another process is using the resource instances.
  - Accumulated for: nobody is using them, but somebody else has requested many instances a long time ago and we are setting aside instances a few at a time to satisfy that big request (*fairness policy*).

- **No preemption**: a resource instance can be released only voluntarily by the process holding it, after that process has completed its task.

- **Circular wait**: there exists a set \( \{ P_0, P_1, \ldots, P_n \} \) of waiting processes such that
  - \( P_0 \) is waiting for an instance of a resource and \( P_1 \) is holding instances of that resource,
  - \( P_1 \) is waiting for ... and \( P_2 \) is holding ...
  - ...,
  - \( P_{n-1} \) is waiting for ... and \( P_n \) is holding ...
  - \( P_n \) is waiting for ... and \( P_0 \) is holding ...

**Circular wait is necessary condition, but not sufficient!**
A set of vertices $V$ and a set of edges $E$.

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$: all the processes.
  - $R = \{R_1, R_2, \ldots, R_m\}$: all resource types.

Instances are squares inside each resource type vertex.

- Request edge: directed edge $P_i \rightarrow R_j$: $P_i$ is waiting to acquire one or more instances of $R_j$.

- Assignment edge: directed edge $R_j \rightarrow P_i$: $P_i$ is holding one or more instances of $R_j$. 
Resource-Allocation Graph (Cont.)

![Resource-Allocation Graph Diagram]
Resource-Allocation Graph With Cycles

Deadlock

No deadlock: $P_4$ or $P_2$ can finish, enabling $P_1$ and $P_3$ to finish.
If graph contains no cycles then no deadlock: no circular wait.

If graph contains a cycle then there is circular wait. Also:
- If only one instance per resource type, deadlock has occurred. (Circular wait is necessary and sufficient condition.)
- If several instances per resource type, deadlock is possible but not certain.
Methods for Handling Deadlocks

1. *Prevention & avoidance:* ensure that the system will *never* enter a deadlock state.

2. *Detection:* allow the system to enter a deadlock state and then recover: CenterRun plan execution engine:
   - Host update plan starts on one host (WebLogic slave), may move to another (WebLogic master) temporarily before finishing with first.
   - Hosts locked by a plan while it runs on them.
   - Multiple plans run at the same time. Plan that would cause deadlock gets aborted as soon as it attempts to lock a host.

3. Ignore the problem and pretend that deadlocks never occur in the system: used by most operating systems, including UNIX.
Deadlock Prevention

Restrain the ways request can be made: ensure that at least one of the four conditions cannot be met.

- **Mutual exclusion**: make sure system doesn’t have nonsharable resources.
  - For example, create printer spooler to provide infinite virtual printers; hence printer never needs to be locked.
  - Some resources inherently nonshareable, e.g. full-screen monitor mode.

- **Hold and wait**: whenever a process requests a resource instance, it should not hold any other resources instances.
  - Require process to request and be allocated all its resources before it begins execution in a single atomic allocation request, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.
Deadlock Prevention (Cont.)

- **No preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
  - Doesn’t apply to some resource types: can’t stop printer half-way, let other process go, and then return.

- **Circular wait:** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration:
  - E.g. scanner must always be requested before printer.
  - $P_0$ \#wait$(S)$ \#wait$(P)$ $P_1$ \#wait$(P)$ \#wait$(S)$ Illegal order.
  - Interactive programs are unpredictable.
Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declares the *maximum number* of instances of each resource type that it may need during its execution.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
- We don’t care who is currently waiting! Algorithms assume worst-case: we treat each process as if it is blocked waiting for whatever it still doesn’t have allocated.
Safe State

- When a process issues allocation request, system must decide if immediate allocation leaves the system in a *safe state*. Who is actually waiting is irrelevant.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence \(<P_1, P_2, ..., P_n>\) is safe if for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all the \(P_j\), with \(j<i\).
  - If \(P_i\) resource needs are not immediately available, then \(P_i\) can wait until all \(P_j\) have finished.
  - When \(P_j\) is finished, \(P_i\) can obtain needed resources, execute, return allocated resources, and terminate.
  - When \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, and so on.
Safe State (Cont.)

- If a system is in safe state then deadlock is impossible.
- If a system is in unsafe state then deadlock is possible (but not certain; see two slides ahead).
- Avoidance: ensure that a system will never enter an unsafe state, hence keeping deadlock impossible:
  - One instance per resource type: resource-allocation graph algorithm.
  - Multiple instances per resource type: banker’s algorithm.
Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicates that process $P_i$ may request resource $R_j$; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed \textit{a priori} in the system.
- System is in safe state as long as the graph doesn’t have a cycle.
  - We treat claim edges as if they are request edges, i.e. as if resource is already waiting for those resource instances.
Resource-Allocation Graph Algorithm (Cont.)

Safe state

Unsafe state: no deadlock since $P_1$ may terminate before it asks for $R_2$.

$P_2$ requests $R_2$ and receives it (it won’t if system runs deadlock avoidance).

$P_2$ releases $R_2$. 
Banker’s Algorithm: Data Structures

- \( n \) = number of processes.
- \( m \) = number of resources types.
- **Available**: \( m \)-element vector.
  - \( \text{Available}[j] = k \): there are \( k \) instances of resource type \( R_j \) available.
- **Max**: \( n \times m \) matrix.
  - \( \text{Max}[i,j] = k \): process \( P_i \) may request at most \( k \) instances of resource type \( R_j \).
- **Allocation**: \( n \times m \) matrix.
  - \( \text{Allocation}[i,j] = k \): \( P_i \) is currently allocated \( k \) instances of \( R_j \).
- **Need**: \( n \times m \) matrix.
  - \( \text{Need}[i,j] = k \): \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task:
    \[
    \text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j].
    \]
**Banker’s Algorithm: Safety Check**

Intuition: we look for a safe sequence. If we find one, the state is safe. If not, it is unsafe.

1. Let Work and Finish be vectors of length $m$ and $n$, respectively. Initialize:
   
   $$Work = \text{Available}$$
   
   $$Finish[i] = (\text{Allocation}_i == 0) \text{ for } i = 1, 2, \ldots, n.$$  

2. Find an $i$ such that both:
   
   (a) $Finish[i] == false$ and
   
   (b) $\text{Need}_i \leq Work$ (All elements are $\leq$).

   If no such $i$ exists, go to step 4.

3. $Work = Work + \text{Allocation}_i$
   
   $Finish[i] = true$
   
   go to step 2.

4. If $Finish[i] == true$ for all $i$, then the system is in a safe state.

5. $P_i$ can become the $k+1^{st}$ process in the safe sequence.

6. Otherwise, we built a partial safe sequence, which means the remaining processes can possibly deadlock.
Example of Banker’s Algorithm

- 5 processes $P_0$ through $P_4$
- Resource types:
  - $A$ (10 total instances),
  - $B$ (5 total instances), and
  - $C$ (7 total instances).

- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Max</th>
<th>Available</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>$P_0$</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>
The *Need* matrix.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The system is in a safe state since the sequence

$< P_1, P_3, P_4, P_2, P_0 >$

is a safe sequence.
Resource-Request Algorithm for Process $P_i$

- **Request**: $m$-element vector.
  - $Request[j] = k$: process $P_i$ wants $k$ instances of resource type $R_j$.

- Resource allocator algorithm, i.e. what to do with this request (grant, enqueue, deny):
  1. If $Request \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  2. If $Request \leq Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available.
  3. *Pretend* to allocate requested resources to $P_i$ by modifying the resource-allocation state as follows:
    
    $Available = Available - Request$
    
    $Allocation_i = Allocation_i + Request$
    
    $Need_i = Need_i - Request$
    
    - If safe then the resources are allocated to $P_i$ and make the temporary changes to the state permanent.
    - If unsafe then $P_i$ must wait and the temporary changes are discarded.
Example of Banker’s Algorithm (Cont.)

- $P_1$ issues request (1,0,2).
- Is Request $\leq$ Need? Yes: $(1,0,2) \leq (1,2,2)$.
- Is Request $\leq$ Available? Yes: $(1,0,2) \leq (3,3,2)$.
- Modify state temporarily, run banker’s algorithm:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 1</td>
<td>6 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

- Sequence $<P_1, P_3, P_4, P_0, P_2>$ is safe. Request is granted.
- Requests enqueued (waiting):
  - $P_4$ for (3,3,0) because resources unavailable.
  - $P_0$ for (0,2,0) because state unsafe.
Deadlock Detection

- Processes do not state their maximal needs.
- But then a process that starts waiting may wait forever: system may enter deadlock state whenever a new process starts waiting.

Need:

- Detection algorithm to identify such states, and
- Recovery scheme to remove deadlock.

Detection/recovery executed either:

- At regular intervals, if complex. More later.
- Or when process issues request for resource instances and process has to wait. Before we enqueue process we check whether doing so would cause deadlock. If so, request fails with error:

  - Recovery is simple: request is denied.
  - Assignment 3.
Single Instance of Each Resource Type

- Maintain wait-for graph:

  Search for a cycle in the graph.
  - Same as avoidance if claim edges are redrawn as active request edges.
  - An algorithm to detect a cycle in a graph requires $O(ne)$ operations ($n$ vertices, $e$ edges). But since each process can wait for at most one other, $e \leq n$ hence $O(n^2)$. 

Depth-first search.
Several Instances of a Resource Type

- Same as banker’s algorithm but we don’t assume anything about future needs. Instead, we look at currently waiting processes (pending requests).
- Assignment 3.
- $n = \text{number of processes}$.
- $m = \text{number of resources types}$.
- Available: $m$-element vector.
  - `Available[j] = k`: there are $k$ instances of resource type $R_j$ available.
- Request: $n \times m$ matrix.
  - `Request[i,j] = k`: process $P_i$ is blocked on a request for $k$ (additional) instances of resource type $R_j$.
- Allocation: $n \times m$ matrix.
  - `Allocation[i,j] = k`: $P_i$ is currently allocated $k$ instances of $R_j$. 
Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   
   \[
   \text{Work} = \text{Available}
   \]
   
   \[
   \text{Finish}[i] = (\text{Allocation}_i == 0) \text{ for } i = 1,2, \ldots, n.
   \]

2. Find an \( i \) such that both:
   
   (a) \( \text{Finish}[i] == \text{false} \) and
   
   (b) \( \text{Request}_i \leq \text{Work} \).
   
   If no such \( i \) exists, go to step 4.

3. \( \text{Work} = \text{Work} + \text{Allocation}_i \)
   
   \( \text{Finish}[i] = \text{true} \)
   
   go to step 2.

4. If \( \text{Finish}[i] == \text{true} \) for all \( i \), then the system is in a good state (no deadlock).

Algorithm requires \( O(mn^2) \) operations to detect whether the system is in a deadlocked state.
Example of Detection Algorithm

- 5 processes $P_0$ through $P_4$
- Resource types:
  - $A$ (7 total instances),
  - $B$ (2 total instances), and
  - $C$ (6 total instances).

- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ discovered. No deadlock.
Example of Detection Algorithm (Cont.)

- $P_2$ requests an additional instance of type $C$.

<table>
<thead>
<tr>
<th>Request</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- No sequence discovered: partial sequence $<P_0>$ means
  - Can reclaim resources held by process $P_0$ when it dies, but insufficient resources to fulfill other pending requests.
  - Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$ and $P_4$ which are mutually dependent.
Detection Algorithm Usage

- Assume regular execution of detection algorithm (by contrast to execution for each request).

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - At least one for each disjoint deadlock cycle.

- If detection algorithm is invoked rarely, there may be many cycles and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

- If deadlock is detected, we need to recover by either:
  - Terminating processes.
  - Forcing them to release resource instances (preemption).
Recovery from Deadlock: Process Termination

- Abort all processes that cannot finish (deadlocked, or stuck behind deadlocked ones): the ones with $Finish[l] = false$.
- Abort one process at a time until all deadlock cycles are eliminated.
- In which order should we choose to abort?
  - Our options may be limited by detection algorithm.
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
Recovery from Deadlock: Resource Preemption

- **Algorithm:**
  - Select a victim process which will lose some of its resources. Process chosen based on some cost function (priority, resource instances used, etc.).
  - Rollback process: return victim to some safe state, restart process for that state. Usually means termination and restart (previous slide).

- **Starvation:** same process may always be picked as victim. To avoid, include number of rollbacks in cost factor.