Chapter 8: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling
The Deadlock Problem

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

Example 1
- System has 2 tape drives.
- $P_1$ and $P_2$ each hold one tape drive and each needs another one.

Example 2
- semaphores A and B, initialized to 1

\[
\begin{align*}
&P_0 &\quad P_1 \\
&\text{wait (A);} &\text{wait(B)} \\
&\text{wait (B);} &\text{wait(A)}
\end{align*}
\]
Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - why is this hard to do?
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.
System Model

- Resource types $R_1$, $R_2$, ..., $R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type $R_i$ has $W_i$ instances.
  - $W = 1, 2, 3, ...$
- Each process utilizes a resource as follows:
  - request
  - use
  - release
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- Mutual exclusion: only one process at a time can use a resource.
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption: a resource 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, …, P_n\} of waiting processes such that \(P_0\) is waiting for a resource that is held by \(P_1\), \(P_1\) is waiting for a resource that is held by \(P_2\), …, \(P_{n-1}\) is waiting for a resource that is held by \(P_n\), and \(P_n\) is waiting for a resource that is held by \(P_0\).
A set of vertices $V$ and a set of edges $E$.

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

- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
Resource-Allocation Graph (Cont.)

- Process

- Resource Type with 4 instances

- \( P_i \) requests instance of \( R_j \)

- \( P_i \) is holding an instance of \( R_j \)
Example of a Resource Allocation Graph

The diagram illustrates a resource allocation graph with nodes representing processes and resources. The processes are labeled as $P_1$, $P_2$, $P_3$, and the resources as $R_1$, $R_3$, $R_2$, and $R_4$. The graph shows the dependencies and allocations between processes and resources.
Resource Allocation Graph With A Deadlock

![Resource Allocation Graph]

- **$P_1$, $R_1$, $P_2$, $R_3$, $P_3$, $R_4$**

\[ \text{Diagram showing resource allocation with potential deadlock.} \]
Resource Allocation Graph With A Cycle But No Deadlock
Basic Facts

- If graph contains no cycles, then no deadlock.

- If graph contains a cycle, then
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.
Methods for Handling Deadlocks

- **Ensure** that the system will never enter a deadlock state.
  - How?

- Allow the system to enter a deadlock state and then recover.
  - How?

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
  - well...not really.
  - what if P(deadlock) = .000000001?
Deadlock Prevention

Restrain the ways request can be made.

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.

- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, 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.

- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
Deadlock Avoidance

Requires that the system has some additional \textit{a priori} information available.

- Simplest and most useful model requires that each process \textit{declare} the maximum number of resources of each type that it may need.
  - \textit{why is this difficult?}

- The deadlock-avoidance algorithm dynamically \textit{examines} the resource-allocation state to ensure that there can never be a circular-wait condition.
  - \textit{why is this difficult?}

- Resource-allocation \textit{state} is defined by the number of available and allocated resources, and the maximum demands of the processes.
  - \textit{why is this difficult?}
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- 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 \(P_i\) 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.
Basic Facts

- If a system is in safe state then no deadlocks.
  - but only if we know the state

- If a system is in unsafe state then possibility of deadlock.

- Avoidance: ensure that a system will never enter an unsafe state.
Safe, Unsafe, Deadlock State

- Safe
- Unsafe
- Deadlock
Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicated that process $P_j$ 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.
Resource-Allocation Graph For Deadlock Avoidance
Unsafe State In Resource-Allocation Graph
Banker’s Algorithm

- Multiple instances of each resource type.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
Data Structures for the Banker’s Algorithm

Let $n =$ number of processes, and $m =$ number of resources types.

- **Available**: Vector of length $m$. If available $[j] = k$, there are $k$ instances of resource type $R_j$ available.
- **Max**: $n \times m$ matrix. If $\text{Max} [i,j] = k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$.
- **Allocation**: $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then $P_i$ is currently allocated $k$ instances of $R_j$.
- **Need**: $n \times m$ matrix. If $\text{Need}[i,j] = k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task.

\[
\text{Need} [i,j] = \text{Max}[i,j] - \text{Allocation} [i,j].
\]
1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   \[
   \text{Work} = \text{Available} \\
   \text{Finish}[i] = \text{false for } i = 1, \ldots, n.
   \]

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

   If no such i exists, go to step 4.

3. Work = Work + 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.
Resource-Request Algorithm for Process $P_i$

Request = request vector for process $P_i$. If $\text{Request}_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $\text{Request}_i \leq \text{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $\text{Request}_i \leq \text{Available}$, 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 state as follows:

   \[
   \begin{align*}
   \text{Available} &= \text{Available} - \text{Request}_i; \\
   \text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\
   \text{Need}_i &= \text{Need}_i - \text{Request}_i;
   \end{align*}
   \]

   1. If safe, the resources are allocated to $P_i$.
   2. If unsafe, $P_i$ must wait, and the old resource-allocation state is restored.
Example of Banker’s Algorithm

- 5 processes $P_0$ through $P_4$
- 3 resource types A (10 instances), B (5 instances), and C (7 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>A</td>
<td>B</td>
<td>C</td>
<td>A</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 content of the matrix. Need is defined to be Max – Allocation.

<table>
<thead>
<tr>
<th>Need</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P₂</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P₃</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The system is in a safe state since the sequence < P₁, P₃, P₄, P₂, P₀ > satisfies safety criteria.
Example $P_1$ Request $(1,0,2)$ (Cont.)

Check that Request $\notin$ Available (that is, $(1,0,2) \notin (3,3,2)$ $\notin$ true.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</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>$P_0$</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>

Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.

Can request for $(3,3,0)$ by $P_4$ be granted?
Can request for $(0,2,0)$ by $P_0$ be granted?
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$.

- Periodically invoke an algorithm that searches for a cycle in the graph.

- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph.
Resource-Allocation Graph and Wait-for Graph

(a) Resource-Allocation Graph

(b) Corresponding wait-for graph
Several Instances of a Resource Type

- **Available:** A vector of length \( m \) indicates the number of available resources of each type.

- **Allocation:** An \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.

- **Request:** An \( n \times m \) matrix indicates the current request of each process. If Request \([i,j] = k\), then process \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).
Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize:
   (a) Work = Available
   (b) For i = 1, 2, ..., n, if Allocation_i ≠ 0, then Finish[i] = false; otherwise, Finish[i] = true.

2. Find an index i such that both:
   (a) Finish[i] == false
   (b) Request_i ≤ Work

   If no such i exists, go to step 4.
Detection Algorithm (Cont.)

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

4. If \( \text{Finish}[i] == \text{false} \), for some \( i, 1 \leq i \leq n \), then the system is in deadlock state. Moreover, if \( \text{Finish}[i] == \text{false} \), then \( P_i \) is deadlocked.

Algorithm requires an order of \( O(m \times n^2) \) operations to detect whether the system is in deadlocked state.
Example of Detection Algorithm

- Five processes $P_0$ through $P_4$.
- Three resource types
  - A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time $T_0$:

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

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in $\text{Finish}[i] = \text{true}$ for all $i$. 
Example (Cont.)

- P₂ requests an additional instance of type C.

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

- State of system?
  - Can reclaim resources held by process P₀, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P₁, P₂, P₃, and P₄.
Detection-Algorithm Usage

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?
  - one for each disjoint cycle

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
  - why is this a bad idea?

- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - 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

- Selecting a **victim** – minimize cost.

- Rollback – return to some safe state, restart process for that state.

- Starvation – same process may always be picked as victim, include number of rollback in cost factor.
Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection
- allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.
Traffic Deadlock for Exercise 8.3
Pg 312, problem 8.12:

Can a system detect that some of its processes are starving? If you answer “yes”, explain how it can. If you answer “no”, explain how the system can deal with the starvation problem.
Pg 312, problem 8.12:

- Can a system detect that some of its processes are starving? If you answer “yes”, explain how it can. If you answer “no”, explain how the system can deal with the starvation problem.

- define “starvation”
  - waiting “unreasonable amount of time” T for a resource
  - process predetermines T
  - or, allocate resources based on longest waiting time
    - allocate to starving processes first