Deadlocks

Chapter 7

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

Bridge Crossing Example

- Narrow bridge: Traffic only in one direction at a time.
- 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).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

The Deadlock Problem

- A set of blocked processes each holding a shared resource and waiting to acquire a resource held by another process in the set.
- Example:
  - System has 2 tape drives.
  - P1 and P2 each hold one tape drive and each needs another one.
- Example:
  - Semaphores A and B, initialized to 1

Deadlock Characterization

Deadlock can arise only 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 of resources: 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 \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that
  - \( P_n \) is waiting for a resource that is held by \( P_{n-1} \),
  - \( P_{n-1} \) is waiting for a resource that is held by \( P_{n-2} \),
  - \( P_0 \) is waiting for a resource that is held by \( P_n \) and
  - \( P_1 \) is waiting for a resource that is held by \( P_0 \).

System Model

- Resource types \( R_1, R_2, \ldots, R_m \)
  - CPU cycles, memory space, I/O devices
- Each resource type \( R_i \) has \( W_i \) instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release
Resource-Allocation Graph

A set of vertices $V$ and a set of edges $E$.

$V$ partitioned into 2 types of vertices:
- $P = \{P_1, P_2, \ldots, P_n\}$, the set of all processes in the system.
- $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system.

$E$ has two types of edges:
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Example of a Resource Allocation Graph

Example of a Resource Allocation Graph With A Deadlock

Example of a Resource Allocation Graph With A Cycle But No Deadlock

Basic Facts

- Graph contains no cycles $\Rightarrow$ no deadlock.
- Graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Deadlock Prevention and Deadlock Avoidance:
  Ensure that the system will never enter a deadlock state.
- Deadlock Detection and Deadlock Recovery:
  Allow the system to enter a deadlock state and then recover.
- Ignore the problem
  and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX, Windows
  - responsibility lifted to the user / programmer.
Deadlock Prevention

Ensure that at least one of the four necessary conditions for deadlock occurrence does not hold.

- **Mutual Exclusion** – needed only for limited shared resources
  - Example: Read-only-file access by arbitrarily many readers

- **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 as a whole before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible; not flexible.

Deadlock Prevention (Cont.)

- **No Preemption of Resources** –
  - If a process 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 a priori information available about the needed resources.

- **Simplest model:** Each process must declare the maximum number of resources of each type that it may need.

- **Resource-allocation state** is defined by
  - the number of available and allocated resources, and
  - the maximum demands of the processes.

- **State transition** = granting a request or releasing resource

- **Deadlock-avoidance algorithm** dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Safe Resource Allocation 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 (i.e., completion sequence) of all processes.

- Sequence \( P_1, P_2, \ldots, P_n \) is safe if, for each \( P_i \), the resources that \( P_i \) can still request can be satisfied by currently available resources + the resources held by all the \( P_j \) with \( j < i \).
  - If \( P_i \)'s 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 \( \Rightarrow \) no deadlocks.
- If a system is in unsafe state \( \Rightarrow \) possibility of deadlock.

Avoidance: ensure that a system will never enter an unsafe state.

Deadlock Avoidance Algorithms

Avoidance Algorithms for 2 Cases:

- **Case 1:** All resource types have 1 instance only
  - Resource Allocation Graph Algorithm

- **Case 2:** 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 the process requests the resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Unsafe State In Resource-Allocation Graph

As this claim turned into a request, we have a deadlock!

Deadlock State In Resource-Allocation Graph

As this claim turns into a request, then we have a deadlock!

Banker’s Algorithm

- Supports multiple instances of a resource type.
  - Current state: held resources of each type per process
  - State transition: grant a request or get back a resource
  - 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.
  - **Principle**: System is always in a safe state.
- At each new resource request:
  1. Assume the request is granted
  2. Check if the resulting system is in a safe state
  3. If so – grant the request, otherwise delay it.

Data Structures for the Banker’s Algorithm

$n =$ number of processes, $P_i$

$m =$ number of resource types, $R_j$

- **Available**: Vector of length $m$.
  - Available[$j$] = $k$ iff $k$ instances of resource type $R_j$ available
- **Max**: $n \times m$ matrix.
  - Max[$i$,$j$] = $k$ ⇒ $P_i$ may request at most $k$ instances of $R_j$
- **Allocation**: $n \times m$ matrix.
  - Allocation[$i$,$j$] = $k$ iff $P_i$ is currently allocated $k$ instances of $R_j$
- **Need**: $n \times m$ matrix.
  - $Need[i] = k$ ⇒ $P_i$ may need $k$ more instances of $R_j$
  - $Need[i] = Max[i] - Allocation[i]$

Safety Algorithm: Is the system in a safe state?

1. Vectors Work[$1..m$] and Finish[$1..n$], initialized to:
   - Work = Available
   - Finish[$i$] = false for $i = 0, 1, ..., n-1$
2. Find an $i$ such that both:
   - (a) Finish[$i$] = false
   - (b) Need[$i$] ≤ Work
     If no such $i$ exists, go to step 4.
3. Work = Work + Allocation,
   Finish[$i$] = true
goto step 2.
4. If Finish[$i$] = true for all $i$
   then the system is in a safe state.

Time required: $O(mn^3)$
**Resource-Request Algorithm for Process Pi**

Request: request vector for process Pi.
Request[i] = k iff Pi wants k instances of resource type R_i.

1. If Request[i] ≤ Need[i] go to step 2.
   Otherwise, error("process has exceeded its maximum claim")

2. If Request[i] ≤ Available go to step 3.
   Otherwise Pi must wait, since resources are not available.

3. Pretend to allocate requested resources to Pi by modifying the state as follows:
   Available = Available – Request[i];
   Allocation[i] = Allocation[i] + Request[i];
   Need[i] = Need[i] – Request[i];

   If safe ⇒ the resources are allocated to Pi.
   If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored.

**Example of Banker's Algorithm**

- 5 processes P_0 … P_4
- 3 resource types
  - A (10 instances), B (5 instances), C (7 instances)
- Snapshot at time T_0:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Available</th>
<th>Need</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 0 1 0</td>
<td>7 5 3 3 2</td>
<td>P_1 7 4 3</td>
</tr>
<tr>
<td>P_1 2 0 0</td>
<td>3 2 2</td>
<td>P_2 1 2 2</td>
</tr>
<tr>
<td>P_2 3 0 2</td>
<td>9 0 2</td>
<td>P_3 0 1 1</td>
</tr>
<tr>
<td>P_3 2 1 1</td>
<td>2 2 2</td>
<td>P_4 0 0 2</td>
</tr>
<tr>
<td>P_4 0 0 2</td>
<td>4 3 3</td>
<td>P_5 4 3 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 > satisfies safety criteria.

**Example Continued:**

- Stated: The system is in a safe state since the sequence < P_1, P_3, P_0, P_2, P_4 > satisfies safety criteria.
- Check:

<table>
<thead>
<tr>
<th>Need</th>
<th>Alloc</th>
<th>Work := 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 7 4 3 0 1 0</td>
<td>not enough</td>
<td>ok... --</td>
</tr>
<tr>
<td>P_1 1 2 2 2 0 0</td>
<td>ok, use and release [2 0 0]</td>
<td>ok... --</td>
</tr>
<tr>
<td>P_2 6 0 0 3 0 2</td>
<td>not ok</td>
<td>ok... --</td>
</tr>
<tr>
<td>P_3 0 1 1 2 1 1</td>
<td>ok... --</td>
<td>ok...</td>
</tr>
<tr>
<td>P_4 4 3 1 0 0 2</td>
<td>ok...</td>
<td></td>
</tr>
</tbody>
</table>

- Scan for a Pi where Need[i] ≤ Work.

- At each "ok, use and release" the resources that a Pi has (Alloc) are returned to Work (i.e., get available).

**Alternative Example:**

Often several possible sequences for a safe state:

- Previous: < P_1, P_3, P_4, P_2 >
- Also: < P_1, P_3, P_4, P_2 >
- As stated in the book!

**Grant Request P_0(0,2,0) ????**

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need: [7 4 3] [0 2 0] [6 0 0] [0 1 1] [4 3 1]</td>
<td>Available: [2 3 0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alloc: [0 1 0] [3 0 2] [3 0 2] [2 1 1] [0 0 2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Tentatively, grant the request P_0(0,2,0):

  - Scan for a Pi where the Need ≤ Available...
  - After finishing, Available...

- Is this new state safe?
- We have an unsafe state – and therefore should not grant the request!
Weaknesses of the Banker’s Algorithm

- Assumes a fixed number of resources
  - not realistic – number of resources can vary over time
- Assumes a fixed population of processes
  - not realistic for interactive systems
- Assumes that processes state maximum needs in advance
  - often not known (depend e.g. on input data or user commands)
- Waiting for completion of one or several processes may take very long / unpredictable time before a request is granted

Remark:
The Banker’s Algorithm was implemented in the THE OS (late 1960s) but is not used in modern OS due to the above limitations.

Deadlock Detection and Recovery

- Allow system to enter deadlock state
- Detection algorithm
  - Single instance of each resource type
  - Multiple instances
- Recovery scheme

Deadlock Detection, Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes.
  - \( P_i \rightarrow P_j \) iff \( P_i \) 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 with \( O(n^2) \) edges (e.g., DFS-based) requires \( O(n^2) \) operations, where \( n \) = number of vertices in the graph.

Detection Algorithm

1. Vectors Work\([1..m]\), Finish\([1..n]\] initialized by:
   \[\text{Work} = \text{Available}\]
   for \( i = 1, 2, ..., n\), if \( \text{Allocation}_i = 0 \) then Finish\(i\) = false
   otherwise Finish\(i\) = true

2. Find an index \( i \) such that both:
   (a) Finish\(i\) = false
   (b) Request\(i\) \leq\ 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\(i\) = false, for some \( i, 1 \leq i \leq n\),
   then the system is in deadlocked state.
   Specifically, if Finish\(i\) = false, then \( P_i \) is deadlocked.

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

Deadlock Detection, Several Instances of a Resource Type

- Available: vector of length \( m \) indicates the number of available resources of each type.
- Allocation: \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.
- Request: \( n \times m \) matrix indicates the currently pending requests of each process.
  \( Request[i, j] = k \) iff \( P_i \) is requesting \( k \) more instances of \( R_j \)

Example of Detection Algorithm

- 5 processes \( P_0 \ldots P_4 \)
- 3 resource types:
  - A (7 instances), B (2 instances), 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 )</td>
<td>( B )</td>
<td>( C )</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>2 0 0</td>
<td>2 0 2</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>2 1 1</td>
<td>1 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 2</td>
<td>0 0 2</td>
<td>0 0</td>
</tr>
</tbody>
</table>

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

- Process P₂ requests an additional instance of type C.

<table>
<thead>
<tr>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P₀ 0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P₁ 2 0 1</td>
<td></td>
</tr>
<tr>
<td>P₂ 0 0 1</td>
<td></td>
</tr>
<tr>
<td>P₃ 1 0 0</td>
<td></td>
</tr>
<tr>
<td>P₄ 0 0 2</td>
<td></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, 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
- Invocation at every resource request?
  - Too much overhead

Occasional invocation?
(e.g., once per hour, or whenever CPU utilization below 40%)

- 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.
- 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 the process needs to complete.
  - How many processes will need to be terminated.

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 rollbacks in cost factor.

Summary

- Deadlock characterization
  - 4 necessary conditions
  - Resource allocation graph

- Deadlock prevention
  - Prohibit one of the four necessary conditions

- Deadlock avoidance
  - 1 instance-resources: Resource allocation graph algorithm
  - Banker’s algorithm (state safety, request granting)

- Deadlock detection and recovery
  - 1 instance-resources: Find cycles in Wait-for graph
  - Several instances: Deadlock detection algorithm

- Do nothing – lift the problem to the user / programmer