Chapter 7
Deadlocks
Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
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
  - System has 2 tape drives.
  - $P_1$ and $P_2$ each hold one tape drive and each needs another one.
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
System resources are utilized in the following way:

- **Request**: If a process makes a request to use a system resource which cannot be granted immediately, then the requesting process blocks until it can acquire the resource.

- **Use**: The process can operate on the resource.

- **Release**: The process releases the resource.

**Deadlock**: A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.
Deadlock Characterization

For a deadlock to occur, each of the following four conditions must hold.

- **Mutual exclusion**: only one process at a time can use a resource.
- **Hold and wait**: A process must be holding a resource and waiting for another.
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait**: A waits for B, B waits for C, C waits for A.
Four conditions? How to prevent this?
Resource-Allocation Graph

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

\[\text{Diagram showing resource allocation between processes and resources.}\]
Resource Allocation Graph With A Deadlock
Resource Allocation Graph With A Cycle But No Deadlock
Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.

- If 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

- Ensure that the system will *never* enter a deadlock state.
  - **Prevention**: Ensure one of the four conditions fails.
  - **Avoidance**: The OS needs more information so that it can determine if the current request can be satisfied or delayed.

- Allow the system to enter a deadlock state, detect it, and recover.

- Ignore the problem and pretend that deadlocks never occur in the system.
Deadlock Prevention: Mutual Exclusion

- By ensuring that at least one of the four conditions cannot hold, we can prevent the occurrence of a deadlock.

- Mutual Exclusion: Some sharable resources must be accessed exclusively (e.g., printer), which means we cannot deny the mutual exclusion condition.
Deadlock Prevention: Hold and Wait

- No process can hold some resources and then request for other resources.

- Two strategies are possible:
  - A process must acquire all resources before it runs.
  - When a process requests for resources, it must hold none (i.e., returning resources before requesting for more).

- **Resource utilization** may be low, since many resources will be held and unused for a long time.

- **Starvation** is possible. A process that needs some popular resources may have to wait indefinitely.
Deadlock Prevention: 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.

- If the requested resources are not available:
  - If they are being held by processes that are waiting for additional resources, these resources are preempted and given to the requesting process.
  - Otherwise, the requesting process waits until the requested resources become available. While it is waiting, its resources may be preempted.
Deadlock Prevention: Circular Wait

- To break the circular waiting condition, we can order all resource types (e.g., tapes, printers).
- A process can only request resources higher than the resource types it holds.
- Suppose the ordering of tapes, disks, and printers are 1, 4, and 8. If a process holds a disk (4), it can only ask a printer (8) and cannot request a tape (1).
- A process must release some higher order resources to request a lower order resource. To get tapes (1), a process must release its disk (4).
- In this way, no deadlock is possible. Why?
Review on Deadlock Prevention

- Four conditions for deadlock
- How to violate these conditions
  - No Mutual exclusion
  - No Hold and wait (example)
  - No No preemption (example)
  - No Circular wait (example) -- special case of no hold and wait?
Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- 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.
Safe State

- When a process requests an available resource, the 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 $<P_1, P_2, \ldots, P_n>$ is safe if for each $P_i$, the resources that $P_i$ can still request can be satisfied by currently available resources + 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 example

For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence? (P1, P0, P2)

<table>
<thead>
<tr>
<th></th>
<th>Maximum Needs</th>
<th>Current Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

What happens to the above table if process P2 requests and is granted one more tape drive?
Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks. (WHY?)

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
Safe, Unsafe, Deadlock State
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 *apriori* in the system.
Resource-Allocation Graph For Deadlock Avoidance

- Process $P_1$ requests $R_1$ and $R_2$
- Process $P_2$ requests $R_1$ and $R_2$

$R_1$ and $R_2$ are mutually exclusive resources.

Dashed arrows indicate the request order relationship.
Unsafe State In Resource-Allocation Graph

\[ R_1 \]

\[ R_2 \]

\[ P_1 \]

\[ P_2 \]
Banker’s Algorithm

- Multiple instances.
- Each process must apriori 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 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 Allocation\([i,j] = k\) then \( P_i \) is currently allocated \( k \) instances of \( R_j \).

■ **Need**: \( n \times m \) matrix. If Need\([i,j] = k\), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task.
Safety Algorithm

1. Let Work and Finish be vectors of length \( m \) and \( n \), respectively. Initialize:

   \[
   Work = \text{Available} \quad Finish[i] = false \text{ for } i = 0, 1, \ldots, n.
   \]

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

   If no such \( i \) exists, go to step 4.

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

   go to step 2.

4. If \( Finish[i] == true \) for all \( i \), then the system is in a safe state.
Complexity $O(n^2 m)$-why?

BOOLEAN function SAFESTATE is -- Determines if current state is safe

{ NOCHANGE : boolean;
  WORK : array[1..m] of INTEGER = AVAILABLE;
  FINISH : array[1..n] of boolean = [false, ..,false];
  I : integer;

repeat
  NOCHANGE = TRUE;
  for I = 1 to N do
    if ((not FINISH[I]) and 
      NEEDI <= WORK) then {
      WORK = WORK + ALLOCATION_i;
      FINISH[I] = true;
      NOCHANGE = false;
    }
  until NOCHANGE;
return (FINISH == (true, .., true));}
Resource-Request Algorithm for Process $P_i$

Request = request vector for process $P_i$. If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, 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 state as follows:

\[
\text{Available} = \text{Available} - \text{Request}_i;
\]

\[
\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;
\]

\[
\text{Need}_i = \text{Need}_i - \text{Request}_i;
\]

- If safe $\Rightarrow$ the resources are allocated to $P_i$.
- If unsafe $\Rightarrow$ $P_i$ 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>$P_0$</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>
Example (Cont.)

- The content of the matrix. Need is defined to be Max – Allocation.

\[
\begin{array}{cccc}
& A & B & C \\
Need & 7 & 4 & 3 \\
P_0 & 1 & 2 & 2 \\
P_1 & 6 & 0 & 0 \\
P_2 & 0 & 1 & 1 \\
P_3 & 4 & 3 & 1 \\
P_4 & \\
\end{array}
\]

- The system is in a safe state since the sequence \(< P_1, P_3, P_4, P_2, P_0 >\) satisfies safety criteria.
Example $P_1$ Request (1,0,2) (Cont.)

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

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$ 0 1 0</td>
<td>7 4 3 2 3 0</td>
<td></td>
</tr>
<tr>
<td>$P_1$ 3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$ 3 0 1</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$ 2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$ 0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>
Example $P_1$ Request (1,0,2) (Cont.)

- 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?
Traffic Example

- Is it safe state with only 3 crossings occupied?
- Is it OK if there is another queue is coming to occupy the 4th crossing?
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 $\text{Request}[i,j] = k$, 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 $i = 1, 2, \ldots, n$, if $Allocation_i \neq 0$, 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.
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.
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 Request Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_3$</td>
</tr>
<tr>
<td>$P_4$</td>
</tr>
</tbody>
</table>

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

State of system?
Traffic Example - detection

- Detection with 4 crossings occupied?
- Detection with 3 crossings occupied?
Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?

- 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.
- 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.