Chapter 7: Deadlocks
Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Recovery from Deadlock
Chapter Objectives

■ To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
■ To present a number of different methods for preventing or avoiding deadlocks in a computer system
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 disk drives
  - $P_1$ and $P_2$ each hold one disk drive and each needs another one

- Example: semaphores $A$ and $B$, initialized to 1

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait (A);</td>
<td>wait(B)</td>
</tr>
<tr>
<td>wait (B);</td>
<td>wait(A)</td>
</tr>
</tbody>
</table>
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)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSs **do not** prevent or deal with deadlocks
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
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 \( \{P_0, P_1, \ldots, P_n\} \) 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 \( P_2 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Deadlocks can be described in terms of *resource-allocation graphs.*

- 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 $P_i \rightarrow R_j$
- 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

![Resource Allocation Graph Diagram]

- Processes: P₁, P₂, P₃, P₄
- Resources: R₁, R₂, R₃, R₄

Each process is connected to at least one resource, illustrating resource allocation in a system.
Resource Allocation Graph With A Deadlock
Graph With A Cycle But No Deadlock
Basic Facts

- If graph contains **no cycles** ⇒ **no deadlock**

- If graph **contains a cycle** ⇒
  - 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

- **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
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 *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, system must decide if immediate allocation leaves the system in a safe state.

- System is in **safe state** if there exists a sequence \(<P_1, P_2, \ldots, P_n>\) of ALL the processes in the systems such that 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 \(j < i\).

- That is:
  - 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.
Example

- A system with 12 tape drives and 3 processes
- \(P_0\) requires 10 tape drives, \(P_1\) requires 4 tape drives, \(P_2\) requires 9

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_0)</td>
<td>10</td>
</tr>
<tr>
<td>(P_1)</td>
<td>4</td>
</tr>
<tr>
<td>(P_2)</td>
<td>9</td>
</tr>
</tbody>
</table>

- At time \(T_0\) the system in a **safe state** with seq. \(<P_1, P_0, P_2>\).
- At time \(T_1\) suppose \(P_2\) requested an additional tape drive, then the sequence \(<P_1, P_0, P_2>\). Will lead to deadlock.
Example

Initially, free=3
At time $T_0$ the system in a safe state with seq. $<P_1, P_0, P_2>$.

<table>
<thead>
<tr>
<th></th>
<th>Max needs</th>
<th>Current needs</th>
<th>Locate more</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>4</td>
<td>2</td>
<td>10-5 .2</td>
<td>5</td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td>5</td>
<td>9-2 .5</td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td>2</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

Free= 1  Free=0  Free=3
At time $T_1$ suppose $P_2$ requested an additional tape drive, then the sequence $<P_1, P_0, P_2>$. Will lead to deadlock, now free=2

<table>
<thead>
<tr>
<th></th>
<th>Max needs</th>
<th>Current needs</th>
<th>Allocate more</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>4</td>
<td>2</td>
<td>4-2</td>
<td>4</td>
</tr>
<tr>
<td>$P_0$</td>
<td>10</td>
<td>5</td>
<td>10-5</td>
<td>4</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
<td>3</td>
<td>9-3</td>
<td>4</td>
</tr>
</tbody>
</table>

Free= 0
Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.
Avoidance algorithms

- **Single** instance of a resource type
  - Use a resource-allocation graph

- **Multiple** instances of a resource type
  - Use the banker’s algorithm
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated 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.

- Request edge converted to an assignment edge when the resource is allocated to the process.

- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed a priori in the system.
Resource-Allocation Graph
Unsafe State In Resource-Allocation Graph

\[ \begin{align*}
R_1 & \quad \rightarrow \quad P_1 \\
& \quad \rightarrow \\
& \quad \rightarrow \\
R_2 & \quad \rightarrow \\
& \quad \rightarrow \\
& \quad \rightarrow
\end{align*} \]
Resource-Allocation Graph Algorithm

- Suppose that process $P_i$ requests a resource $R_j$

- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait until the request can be granted safely.
- 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.

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$
Safety Algorithm

1. Let Work and Finish be vectors of length n.
   Initialize:
   \[ \text{Work} = \text{Available} \]
   \[ \text{Finish}[i] = \text{false} \text{ for } i = 0, 1, \ldots, n-1 \]

2. Find an \( i \) such that both:
   (a) \( \text{Finish}[i] = \text{false} \)
   (b) \( \text{Need}_i \leq \text{Work} \)
   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 \( \text{Finish}[i] = \text{true} \text{ for all } i \), then the system is in a safe state.

---

Step 1: Let Work and Finish be vectors of length n.
Step 2: find a process has not finished, but could with the given available working set.
Step 3: process i finishing up and releasing its resources back into the work pool.
Step 4: a safe sequence has been found.

---

a working copy of the available resources, which will be modified during the analysis.
a vector of Booleans indicating whether a particular process has finished so far in the analysis.
Resource-Request Algorithm for Process $P_i$

$Request_i = \text{request vector for process } P_i$. 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:

   $Available = Available - Request_i$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i - Request_i$

   - 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*

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

- 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 $(1,0,2)$

- 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>$A\ B\ C$</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?
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
End of Chapter 7