Chapter 7: Deadlocks
Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- 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
System Model

- System consists of resources
- 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
- A lock mechanism (e.g., mutex, semaphore) is assigned its own resource class; locks are also system resources
- Each process utilizes a resource as follows:
  - request
  - use
  - release
- request and release may be system calls; a system table records if a resource is free or allocated, and if so, to which process; a queue of waiting processes for one resource
Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example box in textbook (pages 317-318) for mutex deadlock
  - a first thread requests mutex1 and mutex2
  - a second thread requests mutex2 and mutex1
  - they each acquire their first mutex and then wait for their second mutex, acquired by the other
Deadlock Characterization

Deadlock can arise if all four conditions hold simultaneously

- **Mutual exclusion**
  - only one process at a time can use a resource; other processes requesting this resource must be delayed

- **Hold and wait**
  - a process holding at least one resource is waiting to acquire additional resources held by other processes
Deadlock Characterization (cont.)

Deadlock can arise if all **four** conditions hold **simultaneously**

- **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 \( P_2 \), and so forth,
    - \( P_n-1 \) is waiting for a resource that is held by \( P_n \),
    - and \( P_n \) is waiting for a resource that is held by \( P_0 \)
Resource-Allocation Graph

- Directed graph: a set of vertices $V$ and a set of edges $E$
- $V$ is partitioned into two sets:
  - $P = \{ P_1, P_2, \ldots, P_n \}$, set of all the processes in the system
  - $R = \{ R_1, R_2, \ldots, R_m \}$, set 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 $R_j$

- $P_i$ holds an instance of $R_j$
Example of a Resource Allocation Graph
Resource Allocation Graph with a Deadlock

two minimal cycles
Graph with a Cycle But No Deadlock
Basic Facts

- If graph contains no cycles ⇒ no deadlocks

- If graph contains a cycle ⇒ a deadlock *may or may not* exist
  - if only one instance per resource type, then deadlock
    - necessary and sufficient condition
  - if several instances per resource type, possibility of deadlock
    - necessary but not sufficient condition
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 and Windows
Deadlock Prevention

- Restrain the ways request can be made (release at least 1 of the 4 necessary conditions)
  - **Mutual Exclusion** – not required for sharable resources, but must hold for nonsharable resources
  - **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
    1. Require process to request and be allocated all its resources *before* it begins execution, or
    2. Allow process to request resources *only* when the process has *none*
      - it must release its current resources before requesting them back again

- Disadvantages:
  - low resource utilization: unnecessarily unused
  - possible starvation: always waiting...
Deadlock Prevention (cont.)

- **No Preemption** –
  - If a process, that is holding some resources, does request another resource that cannot be immediately allocated to it, then all its 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
  - require that each process requests resources in an increasing order of enumeration, or release resources of *higher or equal* order.
    - several instances of one resource type must be requested in a single request.
  - a *witness* may detect a wrong order and emit a warning.
Deadlock Example

/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /* Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /* Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
    release(lock2);
    release(lock1);
}
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 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 \(< P_1, P_2, ..., P_n >\) of ALL the processes in the system 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.
Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks

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

- Avoidance $\Rightarrow$ 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** $P_i \rightarrow R_j$ indicates that process $P_j$ *may* request resource $R_j$; represented by a dashed line.

- Claim edge is converted to request edge when a process requests a resource.

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

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

- Resources must be claimed *a priori* in the system, i.e., from the start, claim edges must be entered.
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 (an $O(n^2)$ task)

![Diagram showing resource allocation between processes]

- If $P_1$ requests $R_2$, unsafe state
- Allocate $R_2$ to $P_2$
Banker’s Algorithm

- Multiple instances
- Less efficient than the resource-allocation graph algorithm
- Each process must *a priori* claim maximum number of instances of each resource type
- 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

\[ Need[i, j] = Max[i, j] – Allocation[i, j] \]
Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize:
   
   $Work = Available$
   
   $Finish[i] = false$ for $i = 0, 1, ..., n - 1$

2. Find an $i$ such that both:
   
   (a) $Finish[i] == false$
   
   (b) $Need_i \leq Work$ ($P_i$ needs less resources than still available)

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

3. $Work = Work + Allocation_i$ (release resources of $P_i$ back into available)
   
   $Finish[i] = true$

   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 $P_i$

$Request = \text{request vector for process } P_i$

If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

$P_i$ requests resources:

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:

   \[ Available = Available - Request \]
   \[ Allocation_i = Allocation_i + Request_i \]
   \[ Need_i = Need_i - Request_i \]

   - 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>Allocation</th>
<th>Max</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 5 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>
Example (cont.)

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

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

<table>
<thead>
<tr>
<th></th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P₁</strong> ok; releases 2 / 0 / 0</td>
<td>3 / 3 / 2</td>
</tr>
<tr>
<td><strong>P₃</strong> ok; releases 2 / 1 / 1</td>
<td>5 / 3 / 2</td>
</tr>
<tr>
<td><strong>P₄</strong> ok; releases 0 / 0 / 2</td>
<td>7 / 4 / 3</td>
</tr>
<tr>
<td><strong>P₂</strong> ok; releases 3 / 0 / 2</td>
<td>7 / 4 / 5</td>
</tr>
<tr>
<td><strong>P₀</strong> ok; releases 0 / 1 / 0</td>
<td>10 / 4 / 7</td>
</tr>
<tr>
<td><strong>P₀</strong> ok; releases 0 / 1 / 0</td>
<td>10 / 5 / 7</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)

- 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 2</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? no, (2,3,0) $< (3,3,0)$

- Can request for (0,2,0) by $P_0$ be granted? no, further state unsafe
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 (collapse resources nodes in resource-allocation graph)
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$ (to release a requested resource)

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

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

Resource-allocation graph (a) and Corresponding wait-for graph (b)

Resource-allocation graph

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 \textit{Work} and \textit{Finish} be vectors of length \( m \) and \( n \), respectively. Initialize:
   
   (a) \textit{Work} = \textit{Available}
   
   (b) For \( i = 1, 2, \ldots, n \), if \( \textit{Allocation}_i \neq 0 \), then
       \[
       \textit{Finish}[i] = \text{false}; \text{ otherwise, } \textit{Finish}[i] = \text{true}
       \]

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

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

3. \( \text{Work} = \text{Work} + \text{Allocation}_i \) (hoping \( P_i \) will return soon its resources)
   \( \text{Finish}[i] = \text{true} \)
   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.)

- $P_2$ 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?
  - Can reclaim resources held by process $P_0$, but insufficient resources to fulfill requests of the other processes
  - Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$, and $P_4$
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
- Ask the operator to resolve manually the deadlock
- Abort one process at a time until the deadlock cycle is eliminated

In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. 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, but must keep data on the states of running processes

- **Starvation** – same process may always be picked as victim, include number of rollbacks in cost factor
End of Chapter 7