Chapter 8: Deadlocks

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.
- 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 ($P_i, P_j, \ldots, P_k$) of waiting processes such that $P_i$ is waiting for a resource that is held by $P_j$, $P_j$ is waiting for a resource that is held by $P_k$, and $P_k$ is waiting for a resource that is held by $P_i$.

Deadlock with Mutex Locks

Deadlocks can occur via system calls, locking, etc.
- See example box in text page 318 for mutex deadlock
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, ..., P_n\}, the set consisting of all the processes in the system
  - R = \{R_1, R_2, ..., 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 \( R_j \rightarrow P_i \)

Example of a Resource Allocation Graph

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:
  - Deadlock prevention
  - Deadlock avoidance
- 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 (e.g., read-only files); must hold for non-sharable 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 allocated to it.
  - 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

Deadlock Example

```c
/* 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);
}
```

Deadlock Example with Lock Ordering

```c
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);
}
```

Transitions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A.
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 a safe state if there exists a sequence \( P_1, P_2, \ldots, P_n \) 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 a safe state \( \Rightarrow \) no deadlocks.
- If a system is in an unsafe state \( \Rightarrow \) possibility of deadlock.
- Avoidance \( \Rightarrow \) ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock 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

- 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.
Unsafe State In Resource-Allocation Graph

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

Unsafe State In 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
- 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 $[i] = k$, there are $k$ instances of resource type $R_i$ 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) $Need[i] \leq Work$
   - If no such $i$ exists, go to step 4

3. $Work = Work + Allocation$
   - $Finish[i] = true$
   - 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 $P_i$

Let $Request_i$ = 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:
   - $Available = Available - Request_i$
   - $Allocation = Allocation + Request_i$
   - $Need = Need - Request_i$
   - If safe => the resources are allocated to $P_i$
   - 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>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>
</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 \( \text{Need} \) is defined to be \( \text{Max} - \text{Allocation} \)

<table>
<thead>
<tr>
<th>Need</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>7 4 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_1 )</td>
<td>1 2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td>6 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_3 )</td>
<td>0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_4 )</td>
<td>4 3 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence \( \langle P_1, P_3, P_4, P_2, P_0 \rangle \) 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>( 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 \( \langle P_1, P_3, P_4, P_2, P_0 \rangle \) satisfies safety requirement.
- Can request for (3,3,0) by \( P_2 \) 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. 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 vertices in the graph

Resource-Allocation Graph and Wait-for Graph

- 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 \( \text{Work} \) and \( \text{Finish} \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   - \( \text{Work} = \text{Available} \)
   - \( \text{Finish}[i] = \text{false} \) for all \( i \)

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

Example of Detection Algorithm

- **Example**: 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>Process</th>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>0 0 0 0</td>
<td>2 0 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>0 1 0 0</td>
<td>2 0 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3 0 3 0</td>
<td>0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2 1 1 0</td>
<td>0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0 0 2 0</td>
<td>0 0 2</td>
<td>0 0 0 0</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 \( i \).

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
- 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
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

End of Chapter 8