Chapter 7
Deadlocks

Zhi Wang
Florida State University
Contents

- Deadlock problem
- System model
- Handling deadlocks
  - deadlock prevention
  - deadlock avoidance
  - deadlock detection
- Deadlock recovery
The Deadlock Problem

- **Deadlock**: a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Examples:
  - a system has 2 disk drives, $P_1$ and $P_2$ each hold one disk drive and each needs another one
  - semaphores A and B, initialized to 1

  
<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</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 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
  - starvation is possible
- Note: most OSes do not prevent or deal with deadlocks
System Model

• Resources: $R_1, R_2, \ldots, R_m$
  
  • each represents a different **resource type**
    
    • e.g., CPU cycles, memory space, I/O devices
  
  • each resource type $R_i$ has $W_i$ **instances**.

• Each process utilizes a resource in the following pattern
  
  • request
  
  • use
  
  • release
Four Conditions of Deadlock

- **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 it has completed its task
- **Circular wait**: there exists a set of waiting processes \( \{P_0, P_1, \ldots, P_n\} \)
  - \( P_0 \) is waiting for a resource that is held by \( P_1 \)
  - \( P_1 \) is waiting for a resource that is held by \( P_2 \)
  - \( P_{n-1} \) is waiting for a resource that is held by \( P_n \)
  - \( P_n \) is waiting for a resource that is held by \( P_0 \)
Resource-Allocation Graph

• Two types of nodes:
  • \( P = \{P_1, P_2, \ldots, P_n\} \), the set of all the **processes** in the system
  • \( R = \{R_1, R_2, \ldots, R_m\} \), the set of all **resource** types in the system

• Two types of edges:
  • **request edge**: directed edge \( P_i \rightarrow R_j \)
  • **assignment edge**: directed edge \( R_j \rightarrow P_i \)
Resource-Allocation Graph

- Process
- Resource Type with 4 instances
- Pi requests instance of Rj
- Pi is holding an instance of Rj
Resource Allocation Graph

- Is there a deadlock?
Resource Allocation Graph

• Is there a deadlock?
Resource Allocation Graph

- Is there a deadlock?
  - circular wait does not necessarily lead to deadlock
Basic Facts

• If graph contains **no cycles** ➝ no deadlock

• If graph contains a cycle

  • if only **one instance per resource type**, ➝ deadlock

  • if **several instances** per resource type ➝ **possibility** of deadlock
How to Handle Deadlocks

- **Deadlock prevention**: ensure that the system will never enter a deadlock state
- **Deadlock detection and recovery**: allow the system to enter a deadlock state and then recover
- **Ignore the problem** and pretend deadlocks never occur in the system
Deadlock Prevention

- How to prevent **mutual exclusion**
  - not required for sharable resources
  - must hold for non-sharable resources

- How to prevent **hold and wait**
  - whenever a process requests a resource, it doesn’t hold any other resources
    - require process to request *all* its resources before it begins execution
    - allow process to request resources only when the process has none
  - low resource utilization; starvation possible
Deadlock Prevention

• How to handle **no preemption**
  • if a process requests a resource not available
    • release all resources currently being held
    • preempted resources are added to the list of resources it waits for
      • process will be restarted only when it can get all waiting resources

• How to handle **circular wait**
  • impose a total ordering of all resource types
  • require that each process requests resources in an increasing order
  • Many operating systems adopt this strategy for some locks.
Deadlock Avoidance

- Each process declares a max number of resources it may need.
- Deadlock-avoidance algorithm ensure there can never be a circular-wait condition.
- Resource-allocation state:
  - the number of available and allocated resources.
  - 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:
  
  • there exists a sequence \(<P_1, P_2, ..., P_n>\) of all processes in the system
  
  • for each \(P_i\), resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all the \(P_j\), with \(j < i\)

• Safe state can guarantee no deadlock
  
  • if \(P_i\)’s resource needs are not immediately available:
    
    • wait until all \(P_j\) have finished (\(j < i\))
    
    • when \(P_j\) (\(j < i\)) has finished, \(P_i\) can obtain needed resources,
    
    • when \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, and so on
Basic Facts

• If a system is in **safe state** ➞ no deadlocks

• If a system is in **unsafe state** ➞ possibility of deadlock

• **Deadlock avoidance** ➞ ensure a system *never enters an unsafe state*
Deadlock Avoidance
Deadlock Avoidance Algorithms

- Single instance of each resource type ➔ use resource-allocation graph
- Multiple instances of a resource type ➔ use the banker’s algorithm
Single-instance Deadlock Avoidance

- Resource-allocation graph can be used for **single instance resource** deadlock avoidance
  - one new type of edge: **claim edge**
    - claim edge $P_i \rightarrow R_j$ indicates that process $P_i$ *may* request resource $R_j$
    - claim edge is represented by a dashed line
  - **resources must be claimed a priori in the system**
- Transitions in between edges
  - **claim edge** converts to **request edge** when a process requests a resource
  - **request edge** converts to an **assignment edge** when the resource is allocated to the process
  - **assignment edge** reconverts to a **claim edge** when a resource is released by a process
Single-instance Deadlock Avoidance

• Is this state safe?
Single-instance Deadlock Avoidance

- Is this state safe?
Single-instance Deadlock Avoidance

• 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
  
  • no cycle $\Rightarrow$ safe state
Banker’s Algorithm

• Banker’s algorithm is for **multiple-instance resource deadlock avoidance**
  
  • each process must a priori claim **maximum** use of each resource type
  
  • when a process requests a resource it may have to wait
  
  • when a process gets all its resources it must release them in a finite amount of time
Data Structures for the Banker’s Algorithm

- \( n \) processes, \( m \) types of resources
  - **available**: an array of length \( m \), instances of available resource
    - available\([j]\) = \( k \): \( k \) instances of resource type \( R_j \) available
  - **max**: a \( n \times m \) matrix
    - \( \text{max} [i,j] = k \): process \( P_i \) may request at most \( k \) instances of resource \( R_j \)
  - **allocation**: \( n \times m \) matrix
    - allocation\([i,j]\) = \( k \): \( P_i \) is currently allocated \( k \) instances of \( R_j \)
  - **need**: \( n \times m \) matrix
    - need\([i,j]\) = \( k \): \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task
    - need \([i,j]\) = \( \text{max}[i,j] – \text{allocation } [i,j] \)
Banker’s Algorithm: **Safe State**

- Data structure to compute whether the system is in a safe state
  - use **work** (a vector of length \( m \)) to track allocatable resources
    - unallocated + released by finished processes
  - use **finish** (a vector of length \( n \)) to track whether process has finished
  - initialize: **work** = available, **finish**[i] = false for \( i = 0, 1, \ldots, n-1 \)
- Algorithm:
  - find an \( i \) such that **finish**[i] = false && need[i] \( \leq \) work if no such \( i \) exists, go to step 3
  - **work** = **work** + allocation[i], **finish**[i] = true, go to step 1
  - if **finish**[i] == true for all \( i \), then the system is in a safe state
Bank’s Algorithm: **Resource Allocation**

- Data structure: request vector for process $P_i$
  - $\text{request}[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$
- Algorithm:
  1. If $\text{request}_i \leq \text{need}[i]$ go to step 2; otherwise, raise error condition (the process has exceeded its maximum claim)
  2. If $\text{request}_i \leq \text{available}$, go to step 3; otherwise $P_i$ must wait (not all resources are not available)
  3. Pretend to allocate requested resources to $P_i$ by modifying the state:
     
     $\text{available} = \text{available} - \text{request}_i$
     
     $\text{allocation}[i] = \text{allocation}[i] + \text{request}_i$
     
     $\text{need}[i] = \text{need}[i] - \text{request}_i$
  4. Use previous algorithm to test if it is a safe state, if so $\Rightarrow$ allocate the resources to $P_i$
  5. If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored
Banker’s Algorithm: Example

- System state:
  - 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>
Banker’s Algorithm: Example

- need = max – allocation

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

- The system is in a safe state since the sequence < P₁, P₃, P₄, P₂, P₀> satisfies safety criteria
Banker’s Algorithm: Example

• Next, $P_1$ requests $(1, 0, 2)$, try the allocation. The updated state is:

<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>0 1 0</th>
<th>7 4 3</th>
<th>2 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

• 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?
Deadlock Detection

- Allow system to enter deadlock state, but detect and recover from it
- Detection algorithm and recovery scheme
Deadlock Detection: **Single Instance Resources**

- Maintain a 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
Wait-for Graph Example

Resource-allocation Graph

wait-for graph
Deadlock Detection: **Multi-instance Resources**

- Detection algorithm similar to Banker’s algorithm’s safety condition
  - to prove it is **not possible** to enter a **safe state**

- Data structure
  - **available**: a vector of length $m$, 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
    - request $[i, j] = k$: process $P_i$ is requesting $k$ more instances of resource $R_j$
  - **work**: a vector of $m$, the allocatable instances of resources
  - **finish**: a vector of $m$, whether the process has finished
    - if $\text{allocation}[i] \neq 0 \implies \text{finish}[i] = \text{false}$; otherwise, $\text{finish}[i] = \text{true}$
Deadlock Detection: Multi-instance

- Find a process $i$ such that $\text{finish}[i] == \text{false} && \text{request}[i] \leq \text{work}$
  - if no such $i$ exists, go to step 3
- $\text{work} = \text{work} + \text{allocation}[i]; \text{finish}[i] = \text{true}$, go to step 1
- If $\text{finish}[i] == \text{false}$, for some $i$, the system is in deadlock state
  - if $\text{finish}[i] == \text{false}$, then $P_i$ is deadlocked
Example of Detection Algorithm

- System states:
  - 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></th>
<th>allocation</th>
<th>request</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>0 0 2</td>
<td></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.)

- P2 requests an additional instance of type C

  request
  
  A B C

  \[ \begin{array}{c|ccc}
  \text{P}_0 & 0 & 0 & 0 \\
  \text{P}_1 & 2 & 0 & 2 \\
  \text{P}_2 & 0 & 0 & 1 \\
  \text{P}_3 & 1 & 0 & 0 \\
  \text{P}_4 & 0 & 0 & 2 \\
  \end{array} \]

- State of system?
  - can reclaim resources held by process \( \text{P}_0 \), but insufficient resources to fulfill other processes; requests
  - deadlock exists, consisting of processes \( \text{P}_1, \text{P}_2, \text{P}_3, \) and \( \text{P}_4 \)
Deadlock Recovery

• Terminate deadlocked processes. options:
  • 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?
End of Chapter 7