Chapter 7: Deadlocks
Objectives

- Concept of Deadlocks in Computer Systems
- Methods for Handling Deadlocks
The Deadlock Problem

Background

- Computer Systems have a finite number of Resources
- In a Multiprogramming environment, Processes compete for the limited Resources
  - Each Process Requests System Resource from the OS
  - OS allocates Resource to the Process if the requested resource is available
  - A Process transitions to a Waiting (blocked) state if the requested Resource is not currently available

Deadlock Problem

- Deadlock occurs when a set of two or more blocked Processes, each holding to a resource, and waiting to acquire a resource that is held by another process in the set
  - The event never occurs! Why?
Deadlock Problem

Examples

- Semaphores S and Q

\[
\begin{align*}
\text{P}_0 & \quad \text{wait}(S); \\
& \quad \text{wait}(Q); \\
& \quad \text{signal}(S); \\
& \quad \text{signal}(Q);
\end{align*}
\]

\[
\begin{align*}
\text{P}_1 & \quad \text{wait}(Q); \\
& \quad \text{wait}(S); \\
& \quad \text{signal}(Q); \\
& \quad \text{signal}(S);
\end{align*}
\]
Deadlocks

System Model

- **System has a finite number of resources** (CPU Cycles, Memory Space, I/O Devices)
  - **Resource Type Class:** $R_1, R_2, \ldots, R_n$
    - Each Class has $n$-identical instances (copies) of its resource; $n = 1, 2, 3$
      - A system with 2 CPUs $\rightarrow n=2$
      - But a system with 2 networked printers may not have 2 identical instances of the printers Why?

- A Process must request an instance of a Resource Type before using it
  - Requesting Process waits if the resource is currently unavailable
- A process must release a resource after use
Deadlock Characterization

Conditions

The following conditions must hold simultaneously in a system for a deadlock to occur

1. **Mutual Exclusion**
   - Process $P_i$ holds a resource. If other Processes $P_k \ (i \neq k)$ request the same resource, then $P_k$ must wait until $P_i$ releases the requested resource

2. **Hold and Wait**
   - A process $P_i$ holds one or more resources and waiting to acquire additional resources that are currently being held by other Processes $P_k \ (i \neq k)$

3. **No Preemption**
   - Resources are released only voluntarily by the process holding it after the process has completed its task

4. **Circular Wait**
   - Consider a set of waiting Processes $\{P_0, P_1, P_2, \ldots, P_{n-1}, P_n\}$
   - $P_0$ waits for resources held by $P_1$
   - $P_1$ waits for resources held by $P_2$
   - $\ldots \ldots$
   - $P_{n-1}$ waits for resources held by $P_n$
   - $P_n$ waits for resources held by $P_0$
Deadlock Characterization

Resource-Allocation Graph

Deadlocks are modeled using Directed Graphs:

- Resource Allocation Graph (RAG)

  RAG consists of

  - Set of Vertices V representing:
    - Set of two types of nodes
      - \( P = \{P_0, P_1, \ldots, P_n\} \) represents the set of active Processes
      - \( R = \{R_1, R_2, \ldots, R_m\} \) represents set of resource types in the system

  - Set of Edges, E

<table>
<thead>
<tr>
<th>Request/Assignment Edge</th>
<th>(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>When Process ( P_i ) requests an instance of ( R_j )</td>
<td>( P_i \rightarrow R_j )</td>
</tr>
<tr>
<td>When an instance of ( R_j ) is allocated to ( P_i )</td>
<td>( R_j \rightarrow P_i )</td>
</tr>
<tr>
<td>Assignment edge is deleted when Process completes usage of resources</td>
<td></td>
</tr>
</tbody>
</table>

Operating Systems CS 33211
Resource-Allocation Graph

Notations

- Process \( P_i \):
  \[ P_i \]

- Resource Type \( R_j \):
  \[ R_j \]
  - If \( R_1 \) has two instances:
    \[ R_1 \]
  - If \( R_2 \) has one instance:
    \[ R_2 \]
Given the set:

- \( P = \{ P_1, P_2, P_3 \} \)
- \( R = \{ R_1, R_2, R_3, R_4 \} \)
- Resource instances:
  - \( R_1 \) has one
  - \( R_2 \) has 2
  - \( R_3 \) has 3
  - \( R_4 \) has 4

Represent the following set of Edges

\[ E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3 \} \]
Resource-Allocation Graph
Interpretation

E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}

Describe states of P_1, P_2 and P_3

- P_1 is holding an instance of resource R_2 and waiting for an instance of R_1
- P_2 ...
- P_3 ...

What observations can we deduce about the state of processes if R_2 had only one resource instance?
Deadlock Characterization
Resource Allocation Graph with a Cycle

- If each resource type has a single instance
  - A cycle in the graph implies a deadlock has occurred
    - The processes in the cycle are deadlocked
- If each resource has several instances:
  - A cycle does not necessarily imply the occurrence of a deadlock

Suppose $P_3$ requests an instance of $R_2$

List the two cycles in the system
What is the deadlock scenario?
Handling Deadlocks

Methods

- **Deadlock prevention algorithms**
  - Methods that ensure that at least one of the necessary conditions do not hold at any given instance

- **Deadlock avoidance algorithms**
  - OS makes use of: prior info (heuristic data) about processes, currently available resources allocated to each process and resources for possible future requests, then decides whether to wait/fulfill a process request

- **Deadlock Detection algorithm**
  - Detect when deadlock has occurred and use recovery algorithms to resolve the deadlock

- **Ignore the problem** and pretend deadlock never occurred
  - The Application Developer would handle deadlocks explicitly
Ensure that at least one of the necessary conditions do not hold at any given instance

- **Mutual Exclusion**
  - if one process holds a resource, other processes requesting that resource must wait until the process releases it
  - To avoid mutual exclusion scenario:
    - Printers – Use spooling, so process do not wait on the physical printer
    - Sharable resources
      - Read-only files
      - Most files in account
    - Non-sharable resources are hard to handle
      - Networked printers
Deadlock Prevention Algorithm

- No preemption Means resources are released only voluntarily by the process holding it after the process has completed its task
  - To avoid no Preemption scenario:
    - **Allow preemption**
      - If Process \( P_1 \) requests resources that are not available, check to see who holds the resources
        - If holder is waiting on additional resources, preempt holder for resources requested by \( P_1 \)
        - Else, \( P_1 \) waits
          - So preempt some of the other resources of \( P_1 \)
            - \( P_1 \) only wakes up when it acquires new resources
      - If Process \( P_1 \) requests a resource that cannot be allocated to it, all resources held by \( P_1 \) are preempted
        - \( P_1 \) only wakes up when it acquires new resources

**Example**
- Suppose Process \( P_i \) currently holds resources \( R_i \) (\( i = 1, 2, \ldots,n \)) and
- \( P_i \) needs to wait to acquire another resource \( Q_j \)
  - **Protocol:**
    1. \( P_i \) releases all resources \( R_i \) and
    2. \( P_i \) waits for resources \( R_i \) and \( Q_j \)
    3. \( P_i \) wakes up (preempted) only when the \( R_i \) and \( Q_j \) are available
- Suppose Process \( P_i \) requests resources \( R_i \) (\( i = 1, 2, \ldots,n \))
  - **Protocol:**
    1. If resources \( R_i \) are available:
       - Allocate them
    2. Else if \( R_i \) are allocated to a waiting Process:
       - Preempt the desired resources and allocate to \( P_i \)
    3. Else:
       - \( P_i \) must wait
Deadlock Prevention Algorithm

- **Hold and Wait**
  - A process $P_i$ holds one or more resources and waiting to acquire additional resources that are currently being held by other Processes $P_k$

- To avoid Hold and Wait scenario:
  - Ensure when a process requests a resource, it does not hold any other resource
    - Request all resources at once
      - At the start of execution
      - At any point in the program
    - To get new resources, first release current resources, then acquire new and old resources at once
Circular Wait

- Consider a set of waiting Processes \{P_0, P_1, P_2, \ldots, P_{n-1}, P_n\}
- P_0 waits for resources held by P_1
- P_1 waits for resources held by P_2
- \ldots
- P_{n-1} waits for resources held by P_n
- P_n waits for resources held by P_0

To avoid circular wait scenario:

- Impose a logical order on all resources, and require process to request resources in the order
  - Disk Drive → Printer → CDROM
  - Process P_1 requests disk drive then printer
  - Process P_2 requests disk driver then printer
  - Process does not request printer, then disk drive → deadlock

- Ensure order is in logical sequence
  - Allow processes to release all resources, and start request sequence over
  - Let process request total resources at one time
Example:

- Suppose resources $R = \{R_1, R_2, R_3, R_4\}$
- Define a one-to-one mapping:
  $\{R_1, R_2, R_3, R_4\} \rightarrow \{1, 2, 3, 4\}$ for ordering resources

Protocol To Avoid Circular Wait:

- If Process $P_k$ needs an instance of a resource:
  - Request only in increasing order
- If Process needs several instances of the same request:
  - Process issues a single request for all instances
Deadlock Avoidance Algorithms

Model

- Model Requires additional information about how resources are to be requested before Processes start executing

- Deadlock-avoidance algorithms differ in the amount of info required
  - 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 a circular-wait condition can never exist.
  
  - Resource-allocation state is defined by three parameters:
    - The number of available resources
    - The number of allocated resources
    - The maximum demands of the processes.
Deadlock Avoidance Algorithms
Safe State Illustration

Consider a system with
- Processes: P₀, P₁, P₂
- Resources: 12 Tape drives

At \( t₀ \), system is in a safe state:
- \( < P₀, P₁, P₂ > \) is a safe sequence

<table>
<thead>
<tr>
<th>Processes</th>
<th>Max Needs (a priori)</th>
<th>Current Needs (( t₀ ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>10</td>
<td>5 [5] [5]</td>
</tr>
<tr>
<td>P₁</td>
<td>4</td>
<td>2 [2]</td>
</tr>
<tr>
<td>P₂</td>
<td>9</td>
<td>2 [7] [7] [7]</td>
</tr>
</tbody>
</table>

Available Resources: 3 1 5 10 12

Go to slides 7.19 – 7.21
Basic Idea:

- Given a single instance resource-allocation system
- Suppose $P_i$ requests Resources $R_j$: $P_i \rightarrow R_j$
  - If $R_i \rightarrow P_j$ does not result in a cycle in the RAG:
    - System is in **safe** state
    - **Grant $P_i$’s request**
  - Else:
    - System is in **unsafe** state
    - $P_i$ must wait until system is in safe state

To check for safety, we need a cycle detection algorithm in RAG.