Lesson 5: Deadlocks
Memory management (Part 1)

Contents

- The Concept of Deadlock
- Resource-Allocation Graph
- Approaches to Handling Deadlocks
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

==============

- Memory Management Background
- Logical vs. Physical Address Space
- Swapping
- Contiguous Memory Allocation
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 tape drives.
- \( P_1 \) and \( P_2 \) each hold one tape drive and each needs another one.

Example
- Semaphores \( A \) and \( B \), initialized to 1 (mutexes)
  
  \[
  \begin{align*}
  P_0 & \quad P_1 \\
  \text{wait}(A); & \quad \text{wait}(B); \\
  \text{wait}(B); & \quad \text{wait}(A); \\
  \vdots & \quad \vdots 
  \end{align*}
  \]

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.
System Model

- Resource types $R_1, R_2, \ldots, R_m$
  - memory space, I/O devices, records in a file
- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can occur if all 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 $P_2$, $P_0$ is waiting for a resource that is held by $P_n$, and $P_0$ is waiting for a resource that is held by $P_0$.

**NECESSARY** condition! (not sufficient)
Coffman’s conditions [E. G. Coffman, 1971]
A set of vertices $V$ and a set of edges $E$.

$V$ is partitioned into two types (bipartite graph):

- $P = \{P_1, P_2, \ldots, P_n\}$, 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 $P_1 \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

Process

Resource Type with 4 instances

$P_i$ requests an instance of $R_j$

$P_i$ is holding an instance of $R_j$
Can Scheduling Avoid Deadlocks?

- Consider an example:
  - Processes A, B, C compete for 3 single-instance resources R, S, T

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A requests R</td>
<td>B requests S</td>
<td>C requests T</td>
</tr>
<tr>
<td>2.</td>
<td>B requests S</td>
<td>C requests T</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>C requests T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>A requests S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>B requests T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>C requests R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Deadlock
- No deadlock results

- Can a careful scheduling avoid deadlocks?
  - What are the conditions?
  - What algorithm to use?
Approaches to Handling Deadlocks

- **Ostrich approach**: Ignore the problem and pretend that deadlocks never occur in the system
  - Used by many operating systems, including the majority of UNIX implementations

- **Deadlock Prevention**: Take such precautions that deadlock state is unlikely

- **Deadlock Avoidance**: Ensure that the system will never enter a deadlock state

- **Detect & Recover**: Allow the system to enter a deadlock state and then recover

- **Managing deadlock**
  - Try to break at least one of the Coffman conditions

---

**Deadlock Prevention**

Restrain the ways how a resource request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
  - Sharable resource: e.g., read-only file
  - Non-sharable resource: e.g., printer

- **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
  - Consequence: Extremely low resource utilization; danger of starvation
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 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 safe sequence of all processes.
- **Sequence** of processes \(<P_1, P_2, \ldots, P_n>\) is **safe** if for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all the \(P_k\), with \(k<i\).
  - If \(P_i\) resource needs are not immediately available, then \(P_i\) can wait until all \(P_k\) have finished.
  - When \(P_k\) 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.
**Resource-Allocation Graph Algorithm**

- **Claim edge** $P_i \rightarrow R_j$ indicates that process $P_j$ may request resource $R_j$.
  - represented by a dashed line.
- Claim edge changes to a request edge when the process actually requests the resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed in the system *a priori*.

![Resource-Allocation Graph For Deadlock Avoidance](image1)

![Unsafe State In Resource-Allocation Graph](image2)

**Banker’s Algorithm**

- Banker’s behavior (example of one resource type with many instances):
  - Clients are asking for loans up-to an agreed limit
  - The banker knows that not all clients need their limit simultaneously
  - All clients must achieve their limits at some point of time but not necessarily simultaneously
  - After fulfilling their needs, the clients will pay-back their loans

  - Example:
    - The banker knows that all 4 clients need 22 units together, but he has only total 10 units

<table>
<thead>
<tr>
<th>Client</th>
<th>Used</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Eve</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Joe</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Mary</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Available: 10
State (a)

<table>
<thead>
<tr>
<th>Client</th>
<th>Used</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Eve</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Mary</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Available: 2
State (b)

<table>
<thead>
<tr>
<th>Client</th>
<th>Used</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Eve</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Mary</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Available: 1
State (c)
Banker’s Algorithm (cont.)

- Always keep so many resources that satisfy the needs of at least one client.
- 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 resource 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:
   \[
   \text{Work} = \text{Available} \\
   \text{Finish}[i] = \text{false} \text{ for } i = 1, 3, \ldots, n.
   \]
2. Find and \( i \) such that both:
   \begin{enumerate}
   \item \( \text{Finish}[i] = \text{false} \)
   \item \( \text{Need}_i \leq \text{Work} \)
   \end{enumerate}
   If no such \( i \) exists, go to step 4.
3. \( \text{Work} = \text{Work} + \text{Allocation}_i \)
   \[
   \text{Finish}[i] = \text{true} \\
   \text{go to step 2.}
   \]
4. If \( \text{Finish}[i] == \text{true} \) for all \( i \), then the system is in a safe state.

Resource-Request Algorithm for Process \( P_i \)

\( \text{Request}_i \) = request vector for process \( P_i \).
\( \text{Request}_i[j] = k \) means that process \( P_i \) wants \( k \) instances of resource type \( R_j \).

1. If \( \text{Request}_i \leq \text{Need}_i \), go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If \( \text{Request}_i \leq \text{Available} \), 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:
   \[
   \begin{align*}
   \text{Available} &= \text{Available} - \text{Request}_i; \\
   \text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\
   \text{Need}_i &= \text{Need}_i - \text{Request}_i;
   \end{align*}
   \]
   \begin{itemize}
   \item If safe \( \Rightarrow \) the resources are allocated to \( P_i \).
   \item If unsafe \( \Rightarrow \) \( P_i \) must wait, and the old resource-allocation state is restored.
   \end{itemize}
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>Need</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
<td></td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria.

Example (Cont.): $P_1$ requests (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>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</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?
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.
- 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](image)

Corresponding wait-for graph

(a)

(b)
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 $Request[i][j] == k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$.

Detection Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize:
   (a) $Work = Available$
   (b) For $i = 1, 2, ..., n$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.

2. Find an index $i$ such that both:
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$
   If no such $i$ exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i] == false$, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then $P_i$ is deadlocked.

The 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>Total</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 $C$. The Request matrix changes

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P_0$ 0 0 0</td>
</tr>
<tr>
<td>$P_1$ 2 0 1</td>
</tr>
<tr>
<td>$P_2$ 0 0 1</td>
</tr>
<tr>
<td>$P_3$ 1 0 0</td>
</tr>
<tr>
<td>$P_4$ 0 0 2</td>
</tr>
</tbody>
</table>

- State of system?
  - System would now get deadlocked
  - Can reclaim resources held by process $P_0$, but insufficient resources to fulfill other processes’ requests.
  - 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
  - Very expensive

- 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

- Select 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.

Questions? on Deadlocks
Memory Management Background

- Program must be brought into memory and placed within a process memory space for it to be executed
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs (Applications) go through several steps before being run
- Applications' view of the Memory:

![Memory Map Diagram]

Binding of Instructions and Data to Memory

- **Compile time**: If memory location is known a priori, *absolute code* can be generated; must recompile code if starting location changes
- **Load time**: Must generate *relocatable code* if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit* registers).
Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as *virtual address*
  - **Physical address** – the address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical (virtual) and physical addresses differ within the execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real physical* addresses
Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required; implemented through program design (overlays)

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, stub, placed instead of the real procedure call – used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system support needed to check if the routine is in memory and addressable by the process
- Dynamic linking is particularly useful for libraries
Swapping

- A process' memory can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

- Major part of **swap time** is transfer time; total transfer time is directly proportional to the amount of memory swapped.

Contiguous Allocation

- Main memory is usually split into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.

- **Single-partition allocation**
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

A base and a limit register define a logical address space.
Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - *Hole* – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions    b) free partitions (holes)

Questions?

on Memory Management (Part 1)

End of Lesson 5