## Deadlocks

- Develop a description of deadlocks
- Present different methods for preventing/avoiding deadlocks in a computer system

### 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.
  - P1 and P2 each hold one tape drive and each needs another one.

  **Example:**
  - Semaphores A and B, initialized to 1
  - P0: `wait(A); wait(B)`
  - P1: `wait(B); wait(A)`

### System Model

- Resource types R1, R2, ..., Rm
  - Example: CPU cycles, memory space, I/O devices, ...
- Each resource type R has W_i instances.
- Each process utilizes a resource as follows:
  + Request
  + Use
  + Release

### Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously.
  1. Mutual exclusion: only one process at a time can use a resource.
  2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
  3. No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  4. Circular wait: there exists a set (P0, P1, ..., Pn) of waiting processes such that
     - P0 is waiting for a resource that is held by P1
     - P1 is waiting for a resource that is held by P2
     - Pn is waiting for a resource that is held by P0

Condition 4 implies condition 2, i.e., conditions are not independent.

### System Model

- Resource types R1, R2, ..., Rm
  - Example: CPU cycles, memory space, I/O devices, ...
- Each resource type R has W_i instances.
- Each process utilizes a resource as follows:
  + Request
  + Use
  + Release

### Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously.
  1. Mutual exclusion: only one process at a time can use a resource.
  2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
  3. No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  4. Circular wait: there exists a set (P0, P1, ..., Pn) of waiting processes such that
     - P0 is waiting for a resource that is held by P1
     - P1 is waiting for a resource that is held by P2
     - Pn is waiting for a resource that is held by P0

Condition 4 implies condition 2, i.e., conditions are not independent.
Resource-Allocation Graph

A set of vertices $V$ and a set of edges $E$.

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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 $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

- Process
- Resource Type with 4 instances
- $P_i$ requests instance of $R_j$
- $P_i$ is holding an instance of $R_j$

Example: Resource Allocation Graph

Graph with a Cycle but No Deadlock

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - 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.
- 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 requests can be made.

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable 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.
  - 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.

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 <P₁, P₂, ..., Pₙ> is safe if for each Pᵢ the resources that Pᵢ can still request can be satisfied by currently available resources + resources held by all the Pⱼ, with j < i.
  - If Pᵢ resource needs are not immediately available, then Pᵢ can wait until all Pⱼ have finished.
  - When Pᵢ is finished, Pᵢ can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pᵢ terminates, Pᵢ₊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.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.
Resource Allocation Graph Algorithm

- 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.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed \textit{a priori} in the system.

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.

Safety 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 \( i = 1, 3, \ldots, n \).
2. Find an \( i \) such that both:
   (a) \( \text{Finish}[i] = \text{false} \)
   (b) \( \text{Need}_i \leq \text{Work} \)
   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 \( \text{Finish}[i] = \text{true} \) for all \( i \), then the system is in a safe state.
   This algorithm may require an order of \( mn^2 \) operations in order to decide whether a state is safe.

Resource Request Algorithm for Process \( P_i \)

\( \text{Request}_i \) = request vector for process \( P_i \). If \( \text{Request}_i[j] = k \) then 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 \( \text{Need}_i \leq \text{Available} \), 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:
   \( \text{Available} := \text{Available} - \text{Request}_i \)
   \( \text{Allocation}_i := \text{Allocation}_i + \text{Request}_i \)
   \( \text{Need}_i := \text{Need}_i - \text{Request}_i \)
   * 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 P0 through P4; 3 resource types: A (10 instances), B (5 instances), and C (7 instances).

- Snapshot at time T0:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- The matrix Need is defined as: Max – Allocation.

<table>
<thead>
<tr>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P4</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence <P1, P3, P4, P2, P0> satisfies safety criteria. (Checked with safety algorithm)

Example (Cont.)

1. Work := [3 3 2]
2. Need ≤ Work := [3 3 2] + [2 0 0] = [5 3 2]
3. Need ≤ Work := [5 3 2] + [2 1 1] = [7 4 3]
4. Need ≤ Work := [7 4 3] + [1 0 0] = [8 4 4]
5. Need ≤ Work := [8 4 4] + [0 1 0] = [9 4 5]
6. Need ≤ Work := [9 4 5] + [0 0 2] = [10 4 7]

P1, P3, P4, P2 is a safe sequence

Example: P3 requests + (1, 0, 2)

- Check that Request ≤ Available (that is, (1, 0, 2) ≤ (3, 3, 2)) is true.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>0 2 0</td>
</tr>
<tr>
<td>P2</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence <P2, P4, P0, P3> satisfies safety requirement.

- Can request for (3, 3, 0) by P2 be granted?
- Can request for (0, 2, 0) by P4 be granted?

Example of Banker’s Algorithm!

1. Work := [3 3 2]
2. Need ≤ Work := [3 3 2] + [2 0 0] = [5 3 2]
3. Need ≤ Work := [5 3 2] + [2 1 1] = [7 4 3]
4. Need ≤ Work := [7 4 3] + [1 0 0] = [8 4 4]
5. Need ≤ Work := [8 4 4] + [0 1 0] = [9 4 5]
6. Need ≤ Work := [9 4 5] + [0 0 2] = [10 4 7]

P1, P3, P4, P2 is also a safe sequence

Example of Banker's Algorithm

Grant Request P0 (0, 2, 0) Immediately?

1. Work := [3 3 2] - [0 2 0] = [3 1 2]
2. Need := [3 1 2] + [2 1 1] = [5 3 3]
3. Need ≤ Work := [5 3 3] + [1 0 0] = [6 3 4]
4. Need ≤ Work := [6 3 4] + [0 1 0] = [7 4 5]
5. No safe sequence! Unsafe state!
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 and Wait-for Graph

![Resource Allocation Graph and Wait-for Graph](image)

Resource Allocation Graph

- 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 \( \text{Request}_{ij} = k \), 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
   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.

Detection Algorithm (Cont.)

- Complexity of the detection algorithm: \( O(m \times n^2) \)
Example of Detection Algorithm

- Five processes $P_i$ 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>$P_0$</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0</td>
<td>0 0 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0</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.)

- $P_j$ requests an additional instance of type $C$.

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

- State of system?
  - 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?
  - E.g., CPU utilization drops below a threshold, or once every half hour.

- 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

- Recovery by operator (manually) or system (automatically)
  - Abort all deadlocked processes.
  - Cost of re-executing processes
  -Abort one process at a time until the deadlock cycle is eliminated.
  -Cost of invoking deadline detection algorithm

- 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

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process from that state.
- Starvation – same process may always be picked as victim, include number of rollbacks in cost factor.

Recommended Reading and Exercises

- Reading
  - [SGG7] Chapter 7
  - Chapter 8 (sixth edition)

- Exercises
  - All

- No project