Deadlock Handling

Potential 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);
}
```

Why “potential”?

The code may not cause deadlock if one thread grabs both locks before the other.
If both threads hold on the one lock before trying the second lock, a deadlock is ensured.

http://www.eg.bucknell.edu/~cs315/Fall13/code/deadlock/deadlock.c

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state. **(prevention and avoidance)**
- Allow the system to enter a deadlock state and then recover. **(recover)**
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; 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.
  - Low resource utilization; starvation possible.

Prevention and Avoidance

- In deadlock prevention, we try to break one of the four necessary conditions for deadlock.
  - Doing so doesn’t require any extra knowledge, we just examine the current status of the resources and requests to make a decision and take actions
- If extra knowledge is available, or extra constrains are allowed, we can avoid the deadlocks.
Deadlock Avoidance

The system has additional a priori information.

- The 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 States

- Sequence \( <P_0, P_1, \ldots, P_n> \) is safe if for each \( P_i \), the resources that \( P_i \) can still request can be satisfied by currently available resources plus the resources held by all the \( P_j \) with \( j < i \).
  - If \( P_i \) resource needs are not immediately available, then \( P_i \) can wait until all \( P_j \) have finished.
  - When \( P_i \) 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.
- The system is in a safe state if there exists a safe sequence for all processes.
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.

Safe Sequence and State Example

<table>
<thead>
<tr>
<th>Table: Resource allocation</th>
<th>Table: Total resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Needs</td>
<td>Allocated</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
</tr>
</tbody>
</table>

Safe sequence 1: \( <P_1, P_2>, <P_0, 5>, <P_2, 3> \)
Safe sequence 2: \( <P_2, P_3>, <P_0, 5>, <P_1, 2> \)
More safe sequences?

Basic Facts

- If a system is in a safe state there can be no deadlock.
- If a system is in unsafe state, there exists the possibility of deadlock.
- Avoidance strategies ensure that a system will never enter an unsafe state.

Safe, Unsafe, and Deadlock States

Resource-Allocation Graph Algorithm

Goal: prevent the system from entering an unsafe state.

- Applicable only when there is a single instance of each resource type.
- Claim edge \( P_i \rightarrow R_j \) indicates 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 a priori in the system.
- If there is no cycle as the result of allocation, the system is safe.
Resource-Allocation Graph for Deadlock Avoidance

Unsafe State In Resource-Allocation Graph

Banker’s Algorithm

- Applicable when there are multiple instances of each resource type.
- In a bank, the cash must never be allocated in a way such that it cannot satisfy the need of all its customers.
- Each process must state a priori the maximum number of instances of each kind of resource that it will ever need.
- 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.

Banker’s Algorithm: Data Structures

Let \( n \) be the number of processes, and \( m \) be the 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 \( \text{Max}[i,j] = k \), then process \( P_i \) may request at most \( k \) instances of resource type \( R_j \).
- **Allocation**: \( n \times m \) matrix. If \( \text{Allocation}[i,j] = k \) then \( P_i \) is currently allocated \( k \) instances of \( R_j \).
- **Need**: \( n \times m \) matrix. If \( \text{Need}[i,j] = k \), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task.

\[
\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]
\]

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 \( i = 0, 1, 2, \ldots, n-1 \).
2. Find \( i \) such that both:
   - (a) \( \text{Finish}[i] = \text{false} \)
   - (b) \( \text{Need}[i] \leq \text{Work} \)
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.

Resource-Request Algorithm for Process \( P_i \)

1. If \( \text{Request}[i] \leq \text{Need} \), 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:
   - \( \text{Available} = \text{Available} - \text{Request} \)
   - \( \text{Allocation} = \text{Allocation} + \text{Request} \)
   - \( \text{Need} = \text{Need} - \text{Request} \)
   - If safe \( \Rightarrow \) the resources are allocated to \( P_i \) (run safe algorithm)
   - If unsafe \( \Rightarrow \) \( P_i \) 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>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P_0$ 0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>$P_1$ 2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$ 3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_3$ 2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_4$ 0 2 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

Example (Cont.)

- The content of the matrix. Need is defined to be Max – Allocation.

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

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

Example $P_1$ Request (1,0,2)

(Cont.)

- 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>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P_0$ 0 1 0</td>
<td>7 4 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>$P_1$ 3 0 2</td>
<td>0 2 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$ 3 0 1</td>
<td>6 0 0</td>
<td></td>
</tr>
<tr>
<td>$P_3$ 2 1 1</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>$P_4$ 0 0 2</td>
<td>4 3 1</td>
<td></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?