Handling Deadlocks

Table 8.3 Deadlock Handling Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock detection and resolution</td>
<td>The kernel analyzes the resource state to check whether a deadlock exists. If so, it aborts some process(es) and allocates the resources held by them to other processes so that the deadlock ceases to exist.</td>
</tr>
<tr>
<td>Deadlock prevention</td>
<td>The kernel uses a resource allocation policy that ensures that the four conditions for resource deadlocks mentioned in Table 8.2 do not arise simultaneously. It makes deadlocks impossible.</td>
</tr>
<tr>
<td>Deadlock avoidance</td>
<td>The kernel analyzes the allocation state to determine whether granting a resource request can lead to a deadlock in the future. Only requests that cannot lead to a deadlock are granted, others are kept pending until they can be granted. Thus, deadlocks do not arise.</td>
</tr>
</tbody>
</table>

Figure 8.1 Approaches to deadlock prevention.
All Resources Together

• Simplest of all deadlock prevention policies
• Process must ask for all resources it needs in a single request
  – Kernel allocates all of them together
  • A blocked process does not hold any resources, so hold-and-wait condition is never satisfied
• Has one practical drawback:
  – Adversely influences resource efficiency

Resource Ranking

• Resource rank associated with each resource class
• Upon resource request, kernel applies a validity constraint to decide if it should be considered
  – Rank of requested resource must be larger than rank of highest ranked resource allocated to the process
  – Reject the request and abort the process if validity constraint is violated
• Result: absence of circular wait-for relationships
• Works best when all processes require their resources in the order of increasing resource rank
  – In worst case, policy may degenerate into the “all resources together” policy of resource allocation
Deadlock Avoidance

- Grants a resource request only if granting the request cannot lead to a deadlock either immediately or in the future
- Banker's algorithm
  - Each process declares the maximum number of resource units of each class that it may require
  - The kernel admits process $P_j$ only if $\text{Max\_need}_{j,k} \leq \text{Total\_resources}_k$ for all $k$
  - Uses notion of a safe allocation state
    - When system is in such a state, all processes can complete their operation without possibility of a deadlock
  - Deadlock avoidance implemented by taking system from one safe allocation state to another

### Table 8.4 Notation Used in the Banker's Algorithm

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Requested_resources}_{j,k}$</td>
<td>Number of units of resource class $R_k$ currently requested by process $P_j$</td>
</tr>
<tr>
<td>$\text{Max_need}_{j,k}$</td>
<td>Maximum number of units of resource class $R_k$ that may be needed by process $P_j$</td>
</tr>
<tr>
<td>$\text{Allocated_resources}_{j,k}$</td>
<td>Number of units of resource class $R_k$ allocated to process $P_j$</td>
</tr>
<tr>
<td>$\text{Total_alloc}_k$</td>
<td>Total number of allocated units of resource class $R_k$, i.e., $\sum \text{Allocated_resources}_{j,k}$</td>
</tr>
<tr>
<td>$\text{Total_resources}_k$</td>
<td>Total number of units of resource class $R_k$ existing in the system</td>
</tr>
</tbody>
</table>

**Definition 8.2 Safe Allocation State**
An allocation state in which it is possible to construct a sequence of process completion, resource release, and resource allocation events through which each process $P_j$ in the system can obtain $\text{Max\_need}_{j,k}$ resources for each resource class $R_k$ and complete its operation.
Deadlock Avoidance (continued)

Outline of the approach:
1. When a process makes a request, compute projected allocation state
   – This would be the state if the request is granted
2. If projected allocation state is safe, grant request by updating
   Allocated_resources and Total_alloc; otherwise, keep request pending
   – Safety is checked through simulation
   – A process is assumed to complete only if it can get its maximum
     requirement of each resource satisfied simultaneously
3. When a process releases any resource(s) or completes its
   operation, examine pending requests and allocate those that would
   put the system in a new safe allocation state

Deadlock Avoidance (continued)

• When the following condition is satisfied for process $P_i$, the algorithm
  simulates completion of process $P_i$ and release of all resources allocated to it by
  updating $Total_{alloc}$ for each $R_k$

  $\forall R_k: Total_{resources_k} - Total_{alloc_k} \geq Max_{need_{i,k}} - Allocated_{resources_{i,k}}$
Example: Banker’s Algorithm for a Single Resource Class

- The algorithm will grant the request by $P_1$
- Now consider the following requests:
  1. $P_1$ makes a request for 2 resource units
  2. $P_2$ makes a request for 2 resource units
  3. $P_3$ makes a request for 2 resource units
     - Requests by $P_1$ and $P_2$ do not put the system in safe allocation states, hence they will not be granted
     - Request by $P_3$ will be granted

![Figure 8.7](image)

**Algorithm 8.2** Banker’s Algorithm

**Inputs**
- $n$ : Number of processes;
- $r$ : Number of resource classes;
- Blocked : set of processes;
- Running : set of processes;
- Requesting_process : Process making the new resource request;
- Max_need : array $[1..n, 1..r]$ of integer;
- Allocated_resources : array $[1..n, 1..r]$ of integer;
- Requested_resources : array $[1..n, 1..r]$ of integer;
- Total_alloc : array $[1..r]$ of integer;
- Total_resources : array $[1..r]$ of integer;

**Data structures**
- Active : set of processes;
- feasible : boolean;
- New_request : array $[1..r]$ of integer;
- Simulated_allocation : array $[1..n, 1..r]$ of integer;
- Simulated_total_alloc : array $[1..r]$ of integer;

1. $Active := Running \cup Blocked$;
   for $k = 1..r$
   $New\_request[k] := Requested\_resources[requesting\_process, k];$
Example: Banker's Algorithm for Multiple Resource Classes

(a) State after Step 1

(b) State before while loop of Step 4

Figure 8.12 Operation of the banker’s algorithm for Example 8.11.
Deadlock Handling in Practice

- Deadlock detection-and-resolution and deadlock avoidance are unattractive in practice (overhead)
  - OS uses deadlock prevention approach or simply does not care about possibility of deadlocks
- OSs tend to handle deadlock issues separately for each kind of resource
  - Memory: preemptible → explicit deadlock handling is unnecessary
  - I/O devices: Resources are not limited (virtual devices)
  - Files: Deadlocks are handled by processes, not OS
  - Control blocks: Resource ranking or all-resources-together
Summary

- **Deadlock**: set of processes wait indefinitely for events because each of the events can be caused only by other processes in the set.
- Resource deadlock arises when:
  - Resources are nonshareable and nonpreemptible
  - Hold-and-wait
  - Circular wait exists
- OS can discover a deadlock by analyzing the allocation state of a system
  - Use RRAG, WFG, or *matrix model*
- Deadlocks can be detected, prevented, and avoided