G22.3250
Honors Operating Systems

Lecture 11: Deadlocks
Avoidance, Detection, and Recovery

Outline

• Process deadlocks (contd.)
  – deadlock avoidance
  – deadlock detection
  – deadlock recovery
  – combined approach to deadlock handling
  – extended resource allocation graphs
    • consumable and reusable resources

Silberschutz/Galvin: Sections 7.5 - 7.9

Review: Deadlock Prevention

• Main idea: Prevent one of the four necessary conditions
  – mutual exclusion
  – hold-and-wait
  – no preemption
  – circular wait

• Limitations
  – inefficient
    • static allocation of resources reduces concurrency
    • a process may need to be preempted even when there is no deadlock
  – restrictive
    • requires allocation of future resource requirements before it starts executing

• Alternative approaches?

Deadlock Avoidance

• Main idea:
  – request additional information about how resources are to be requested
  – prior to allocating a request, verify that the system will not enter a deadlock state
    \[ f(\text{resources currently available, resources currently allocated, future requests and releases}) \]
    • if not: grant the request
    • if yes: block the process

• Algorithms differ in amount and type of information
  – simplest (also most useful) model: \textit{maximum number of resources}
  – other choices
    • sequence of requests and releases
    • alternate request paths

• How can we find out if a system will enter a deadlock state?
Deadlock Avoidance: Notion of a Safe State

A system is in a **safe state** if there exists a **safe sequence**

A sequence \(<P_1, P_2, \ldots, P_n>\) is a **safe sequence** for the current allocation if, for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus resources held by all the \(P_j\) with \(j<i\)

- **Properties of safe states**
  - a safe state is not a deadlock state
  - an unsafe state may lead to deadlock
  - it is possible to go from a safe state to an unsafe state
    - e.g., system has 12 units of a resource, and process requirements are
      - \(P_1\): maximum need = 10, current need = 5
      - \(P_2\): maximum need = 4, current need = 2
      - \(P_3\): maximum need = 9, current need = 2
    - system enters an unsafe state if \(P_3\)'s request for an additional unit is granted

- **Avoidance algorithms prevent the system from entering an unsafe state**

---

Deadlock Avoidance Algorithms - 1

- **Single resource instances**
  - detect cycles in the resource allocation graph
  - \(O(V+E)\) operations
  - a request is converted to an assignment edge iff it does not result in a cycle

\(<P_3, P_2, P_1>\) is a safe sequence

Say \(P_1\) requests \(R_3\):

- should this be granted?

  No, because an assignment edge from \(R_3\) to \(P_1\) would create a cycle in the RAG.

  Does this always imply a deadlock?

  No, because \(P_1\) can release \(R_3\) before requesting \(R_1\)

---

Deadlock Avoidance Algorithms - 2

- **Multiple resource instances: Banker’s Algorithm**
  - upon entering the system, a process declares the **maximum** number of instances of each resource type that it may need
  - the algorithm decides, for each request, whether granting it would put the system in an unsafe state

<table>
<thead>
<tr>
<th>Resource availability</th>
<th>(Available[1..m])</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum demand</td>
<td>(Max[1..n, 1..m])</td>
</tr>
<tr>
<td>current allocation</td>
<td>(Allocation[1..n, 1..m])</td>
</tr>
<tr>
<td>potential need</td>
<td>(Need[1..n, 1..m])</td>
</tr>
</tbody>
</table>

1. If \(Request_i \leq Need_i\),
   goto Step 2, else flag error
2. Find an \(i\) such that
   - \(Finish[i] = false\), and
   - \(Need_i \leq Work\)
   if no such \(i\), goto Step 4
3. If \(Finish[i] = true\),
   goto Step 2;
4. If \(Need_i \leq Work\),
   then the system is in a safe state

1. **Work** := \(Available\); \(Finish[i] := false\), for all \(i\);
2. Find an \(i\) such that
   - \(Finish[i] = false\), and
   - \(Need_i \leq Work\)
   if no such \(i\), goto Step 4
3. **Work** := \(Work + Allocation\);
   \(Finish[i] := true\);
   goto Step 2;
4. If \(Finish[i] = true\) for all \(i\),
   then the system is in a safe state

<table>
<thead>
<tr>
<th>Capacity</th>
<th>([2, 4, 3])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Max)</td>
<td>([{1, 2, 2}, {1, 2, 1}, {1, 1, 1}])</td>
</tr>
<tr>
<td>(Allocation)</td>
<td>([{1, 2, 0}, {0, 1, 1}, {1, 0, 1}])</td>
</tr>
<tr>
<td>(Available)</td>
<td>([0, 1, 1])</td>
</tr>
<tr>
<td>(Need)</td>
<td>([{0, 0, 2}, {1, 1, 0}, {0, 1, 0}])</td>
</tr>
</tbody>
</table>

- \(P_i\) requests \([0, 0, 1]\)
  Should this be granted?

- **Allocate and check if system is in a safe state**

1. **Work** := \(Available\);
   \(Finish[i] := false\), for all \(i\);
2. Find an \(i\) such that
   - \(Finish[i] = false\), and
   - \(Need \leq Work\)
   if no such \(i\), goto Step 4
3. **Work** := \(Work + Allocation\);
   \(Finish[i] := true\);
   goto Step 2;
4. If \(Finish[i] = true\) for all \(i\),
   then the system is in a safe state

Initially, \(Work = [0, 1, 0]\)

\(Need_i \leq Work\), so \(P_3\) can finish

\(Work = [1, 1, 1]\)
Now, both \(P_1\) and \(P_2\) can finish
Limitations of Deadlock Avoidance

- Requires specifying future needs
  - not generally known for OS processes
  - more applicable to specialized situations
    • programming language constructs (e.g., transaction-based systems)
    • known OS components (e.g., Unix "exec")
- Most general solution: Deadlock detection and recovery

Deadlock Detection

- Go back to using a resource allocation graph in which only
  - request and assignment edges are defined
  - future (potential) requests are not relevant to "is there deadlock now?"
- Two cases
  - single instance of each resource
    • can detect deadlock by looking for a cycle
    • need only look at the wait-for graph
      - obtained by removing resource nodes and collapsing the appropriate edges
  - multiple instances of each resource
    • cycles do not imply deadlock

Deadlock Detection: Multiple Resource Instances

- A cycle in the graph is a necessary but not sufficient condition for the existence of a deadlock
  - if a cycle does not exist: no deadlock
  - if a cycle exists: there may or may not be a deadlock

Deadlock Detection Algorithm

- A new use for the Bankers’ algorithm
  - detect if the current set of requests are such that satisfying any of them will put the system in an unsafe state

1. Work := Available;
   Finish[i] := false, for all i;
2. Find an i such that
   a. Finish[i] = false, and
   b. Request[i] ≤ Work
   if no such i, goto Step 4
3. Work := Work + Allocation[i];
   Finish[i] := true;
   goto Step 2;
4. If Finish[i] = false for some i,
   then the system is in a deadlock state
5. Work := Available;
   Finish[i] := false, for all i;
6. Find an i such that
   a. Finish[i] = false, and
   b. Need[i] ≤ Work
   if no such i, goto Step 4
7. Work := Work + Allocation[i];
   Finish[i] := true;
   goto Step 2;
8. If Finish[i] = true for all i,
   then the system is in a safe state
Deadlock Recovery

- Only general principles known

Two options
- Break the cyclic waiting by terminating some of the processes
  - choice 1: abort all deadlocked processes
  - choice 2: abort one process at a time till deadlock resolved
- Enable at least one of the processes to make progress
  (by preemitting resources from another)
  - issue 1: how is the victim process selected?
  - issue 2: can the process handle resource preemption?
    - in general, might require rollback & restart
  - issue 3: how does one prevent starvation?
    - bound the number of rollbacks/preemptions for a particular process

Combined Approaches

- Using only a single approach (prevention, avoidance, or detection + recovery) in isolation is not very effective

- Combination is superior
  - general idea: classify resources, use different approach for each
    - process control blocks: use resource ordering
    - user process memory: use pre-emption
    - assignable devices: avoidance; require prior claims
    - swappable space: preallocate; no hold & wait

Generalizing the Resource Allocation Graph

- Resources partitioned into two sets:
  - reusable resources: what we have been talking about so far
  - consumable resources: produced and consumed by processes
    - e.g., messages, interrupt signals
    - each consumable resource has a nonempty set of producers
      - a producer can produce any number of instances of that resource

Operations on the General RAG

- Request
  - adds edges between processes and resource nodes
  - a request for multiple instances adds multiple edges

- Grant (modulo availability of sufficient number of instances)
  - reusable resources: request edges are converted into assignment edges
  - consumable resources: request edges are deleted

- Release
  - reusable resources: assignment edges are deleted
    - the instance count of the resource is incremented for each such edge
  - consumable resources: no change in producer edges
    - the instance count can increase by an arbitrary amount
The Graph Reduction Method

- An optimistic method which assumes that a process will not make any additional requests
  - corresponds to the best set of operations to unblock any blocked processes
- A general RAG can be reduced by a process \( P_i \) that is not blocked
  - reusable resources: delete all request and assignment edges
    - for each deleted assignment edge, increment the instance count of the resource
  - consumable resources:
    - delete all request edges and decrement the instance count for each such edge
    - if \( P_i \) is a producer, delete the producer edge and set instance count to infinity

A graph is completely reducible if a sequence of reductions deletes all edges in the graph

A system-state is deadlock free if its general RAG is completely reducible

Deadlock: Necessary and Sufficient Conditions

- While a completely reducible graph implies deadlock-freedom, the reverse is not true
  - does not give a sufficient condition for the presence of a deadlock
  - moreover, very inefficient approach for verifying deadlock-freedom

A state is an expedient state if all processes having outstanding requests are blocked
  - i.e., all possible grantable requests have been granted

A knot \( K \) in a graph is a nonempty set of nodes such that for every node \( x \) in \( K \), all nodes in \( K \) and only the nodes in \( K \) are reachable from \( x \)

In a general RAG
  - a cycle is a necessary condition for a deadlock
  - if the graph is expedient, then a knot is a sufficient condition for a deadlock

Deadlock Conditions: Special Cases

- Systems with single-unit requests
  An expedient general RAG with single-unit requests represents a deadlock state \( \text{iff} \) it contains a knot
    - how can we check for knots?
    - make the ancestor of each sink node a sink
      a general RAG does not contain a knot \( \text{iff} \) all nodes become sinks

- Systems with only reusable resources
  If different sequences of reductions applied to a starting state \( S \) result in states that cannot be reduced, then all of these resulting states are identical
  A state \( S \) is not a deadlock state \( \text{iff} \) \( S \) is completely reducible

Next Lecture

- Memory Management
  - logical versus physical address space
  - swapping
  - allocation
  - paging, segmentation, and hybrids

Reading
- Silberschatz/Galvin: Chapter 8