Lecture 11
Deadlock Prevention, Avoidance, Detection and Recovery
March 20, 2002

Outline

• Announcements
  – Lab 3 due date extended to Monday, March 25th (12 noon)
  – Lab demos on Monday (March 25th) and Tuesday (March 26th)

• Deadlocks (cont’d)
  – (Review) deadlock characterization
  – methods for handling deadlocks
    – deadlock prevention
    – deadlock detection
    – deadlock recovery
    – combined approach to deadlock handling

[Silberschatz/Galvin/Gagne: 8.3 – 8.7]

(Review) Conditions for Deadlock

• Deadlocks involve a set of processes contending for a set of resources

• All of the following conditions must hold for deadlock to occur
  – Mutual Exclusion
    • at least one resource can only be used by one process at any one time
  – Hold and Wait
    • there must exist at least one process that is holding at least one resource, and is waiting to acquire additional resources currently held by other processes
  – No Preemption
    • processes cannot be forced to give up resources
  – Circular Wait
    • there is a sequence of processes $p_1, p_2, ..., p_n, p_1$ such that $p_i$ is waiting for a resource held by $p_{i+1}$

Methods for Handling Deadlocks

Three approaches with different cost-performance tradeoffs

• Prevention
  – deadlock cannot possibly occur

• Avoidance
  – deadlock can occur, but there are algorithms to avoid it
  – relies on the OS having an advance model of possible resource requests from processes

• Detection and Recovery
  – deadlock may occur, but there are ways of detecting it and recovering
  – this method is preferable when deadlocks happen rarely
Deadlock Prevention

• Approach: Ensure that the necessary conditions for deadlocks are never satisfied

• Prevent one of the following from becoming true
  – Mutual Exclusion
  – Hold and Wait
  – No Preemption
  – Circular Wait

Deadlock Prevention (1): Mutual Exclusion

• Mutual exclusion is not a problem for sharable resources
  – an example is a “read-only” file which is a resource that can be accessed simultaneously

• Problem: Some resources are inherently not sharable
  – so, denying the mutual exclusion condition cannot be enforced in general

Deadlock Prevention (2): Hold-and-Wait

• Approach: Guarantee that when a process requests a resource it does not hold any other resources
  – Choice 1: A process requests and is allocated all of its resources before it begins execution
    • require system calls requesting resources to precede all other system calls
    – Need all resources to be requested with a single call
  – Choice 2: A process releases any resources it is holding before it requests for new ones
  – Choice 3: A process that is “holding” a resource immediately releases it if another of its requests cannot be satisfied currently

• Limitations
  – inefficient
  – lowered resource utilization
  – starvation

Deadlock Prevention (3): No Preemption

• Approach: Take away resources from a process (preemption) and give them to another waiting process
  – some resources are preemptible
    • e.g., memory space, disk space (on a particular disk)
    • these can be taken away from a process
  – examples of non-preemptible resources?

• Choices
  – protocol 1: if a process is holding some resources and requests other resources that cannot be granted to it, all of its resources are taken away
  – protocol 2: when a process requests additional resources, see if these resources are being held by a process that is itself waiting for new resources. In this situation, preempt the second process

• Limitation: Cost of preemption
  – a process may get preempted even when there is no deadlock
Deadlock Prevention (4): Circular Wait

- Example:
  - Processes need three resources: memory, disk, printer
  - Consider two cases:
    - Case 1: processes pick own order in which to ask for resources
    - Case 2: each process asks first for memory, then disk, then the printer
  - Which of these cases can result in deadlock?

- In our example
  - rank(memory) = 1, rank(disk) = 2, rank(printer) = 3

Why it works
- suppose the circular wait consists of processes P1, P2, ..., Pn, P1
- suppose P(i) is waiting on a resource held by P(i+1) of rank R(j)
  - P(i+1) must have been granted all resources it needs of rank R(j)
  - it must therefore be waiting for a resource of rank \( \geq R(j) + 1 \)
- since a cycle of strictly increasing ranks cannot exist, there can exist no such cycle.

Two related points
- an equivalent strategy is one where a process, when it requests a resource of a particular rank, releases all those with a higher rank
- typical rank orders are based on natural usage
  - e.g., since storage devices are used “before” printers, they get smaller ranks

Deadlock Prevention: Summary

- Main idea: Prevent one of the four necessary conditions
  - mutual exclusion
  - hold-and-wait: ask for all resources at start
  - no preemption
  - circular wait: resource ranking scheme

- 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
- before allocating request, verify that system will not enter a deadlock state
  \( F(\text{resources currently available, resources currently allocated, future requests and releases}) \)
  - if no: grant the request
  - if yes: block the process

Algorithms differ in amount and type of information
- simplest (also most useful) model: 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 \(<P_1, P_2, ..., P_n>\) 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

Example: A system with 12 units of a resource
- Three processes
  - \(P_1\): max need = 10, current need = 5
  - \(P_2\): max need = 4, current need = 2
  - \(P_3\): max need = 9, current need = 2
- This is a safe state, since a safe sequence \(<P_2, P_1, P_3>\) exists
- \(P_3\) requests an additional unit. Should this request be granted?
  - No, because this would put the system in an unsafe state
    - \(P_1, P_2, P_3\) will then hold 5, 2, and 3 resources (2 units are available)
    - \(P_2\)'s future needs can be satisfied, but no way to satisfy \(P_1\)'s and \(P_3\)'s needs

Avoidance algorithms prevent the system from entering an unsafe state

Deadlock Avoidance: Single Resource Instances

- Deadlock \(\equiv\) Cycle in the resource allocation graph
- A request is granted iff it does not result in a cycle
  - cycle detection: \(O(V + E)\) operations

Example: A system with 3 units of a resource
- \(<P_3, P_2, P_1>\) is a safe sequence
  - \(P_3\)'s future request can be satisfied because \(R_3\) is available
  - \(P_2\)'s future request can be satisfied by \(P_3\) yielding \(R_2\)
  - \(P_1\)'s future requests can be satisfied by \(P_2\) giving up \(R_1\) and available \(R_3\)

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.
  - No safe sequence exists
  - Does this always imply a deadlock?
- No, because \(P_1\) can release \(R_3\) before requesting \(R_1\)
Deadlock Avoidance: 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

1. If Request ≤ Need, go to Step 2, else flag error
2. If Request ≤ Available
   goto Step 3, else wait
3. Allocate the resources
   Available := Available - Request;
   Allocation := Allocation + Request;
   Need := Need - Request;
   Check if this is a safe state
   If not: undo the allocation and wait
4. If Finish[i] = true for all i, then the system is in a safe state

Banker’s Algorithm: Example

- Three resource types and three processes (P₁, P₂, P₃)
  - Capacity = [2, 4, 3]
  - Max = [[1, 2, 2], [1, 2, 1], [1, 1, 1]]
  - Allocation = [[1, 2, 0], [0, 1, 1], [1, 0, 1]]
  - Available = [0, 1, 1]
  - Need = [[0, 0, 2], [1, 1, 0], [0, 1, 0]]

P₁ requests [0, 0, 1]
Should this be granted?

Allocate and check if system is in a safe state
Allocation = [[1, 2, 1], [0, 1, 1], [1, 0, 1]]
Available = [0, 1, 0]
Need = [[0, 0, 1], [1, 1, 0], [0, 1, 0]]

1. Work := Available;
   Finish[i] := false, for all i;
2. Find an i such that:
   a. Finish[i] = false, and
   b. Need[i] ≤ 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, ≤ Work, so P₃ can finish
Work = [1, 1, 1]
Now, both P₁ and P₂ can finish

Limitations of Deadlock Avoidance

- Deadlock avoidance vs. deadlock prevention
  - Prevention schemes work with local information
    - What does this process already have, what is it asking
  - Avoidance schemes work with global information
    - Therefore, are less conservative

- However, avoidance schemes require specification of 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”)

- More general solution: Deadlock detection and recovery

Deadlock Detection: Single Resource Instances

- 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?”

- Deadlock = Cycle in the RAG
  - need only look at the wait-for graph
    - obtained by removing resource nodes and collapsing the appropriate edges
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

(Examples from last lecture)

Detection: Multiple Resource Instances

- 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. \(Request[i] \leq Work\)
   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

Detection: Multiple Resource Instances (Example)

- System with three resource types and five processes

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅ P0 [0, 1, 0] [0, 0, 0] [3, 1, 3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P1 [2, 0, 0] [2, 0, 2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P2 [2, 0, 3] [0, 0, 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P3 [2, 1, 1] [1, 0, 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P4 [0, 0, 2] [0, 0, 2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No deadlock!

- What about the following?

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅ P0 [0, 1, 0] [0, 0, 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P1 [2, 0, 0] [2, 0, 2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P2 [3, 0, 3] [0, 0, 1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P3 [2, 1, 1] [1, 0, 0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✅ P4 [0, 0, 2] [0, 0, 2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deadlock!

Deadlock Recovery

- Only general principles known (read Section 8.7 for details)

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 preempting 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
- Example: Consider a system with four classes of resources
  - internal resources (e.g., PCBs)
  - main memory
  - job resources (e.g., tape drives, files)
  - swappable space
- A mixed deadlock solution
  - process control blocks: use resource ordering (prevention) Why?
  - user process memory: use pre-emption (detection/recovery)
  - job resources: require prior claims (avoidance) Why?
  - swappable space: preallocate; no hold & wait (prevention)

Next Lecture

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

Reading
- Silberschatz/Galvin/Gagne: Chapter 9