V22.0202-001
Computer Systems Organization II (Honors)
(Introductory Operating Systems)

Lecture 12
Process Deadlocks (cont’d)

March 24, 2004

Outline

• Announcements
  – Lab 4 due back on April 5th
    • Demos on April 5th and 6th
  – Questions?

• Process Deadlocks
  – System model
  – Characteristics of deadlocks
  – Methods for handling deadlocks
    • Deadlock prevention

[Silberschatz/Galvin/Gagne, Sections 8.2 – 8.8]

---

(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 \( p_{i+1} \)

---

(Review) Graph Representations of Deadlocks

A Resource Allocation Graph (RAG)

• Two types of nodes
  – Processes and
  – Resources

• Three types of directed edges between Processes and Resources
  – request edge: a solid edge from \( P \) to \( r \), indicating that \( P \) has requested \( r \)
  – assignment edge: a solid edge from \( r \) to \( P \), indicating that the OS has already allotted resource \( r \) to process \( P \)
  – claim edge: a dotted edge from a process node \( P \) to a resource node \( r \), indicating that \( P \) may request \( r \) at some point in the future

• We shall focus only on requests for exclusive access to a resource
  – handling of mixed access types is slightly complicated
(Review) Resource Allocation Graphs: Example

• Assignment edges originate from a resource instance
• A request edge is instantaneously transformed to an assignment edge if resources are available

Deadlocks in RAGs with Single Resource Instances

• A cycle in the graph is a necessary and sufficient condition for the existence of a deadlock

Deadlocks in RAGs with 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

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?

Deadlock Prevention: Circular Wait (cont’d)

• Approach: Impose an order on resource acquisition
  – all the N types of resources in the system are linearly ordered
    • each is given a number, called rank, in the range 1, 2, ..., N
    • the resources of the same type all have the same rank
    • different types of resources get distinct ranks
  – processes are required to sequence their resource requests in strictly increasing order of rank
    • i.e., they ask for all the “smaller” rank resources first
  – In our example
    • rank(memory) = 1, rank(disk) = 2, rank(printer) = 3
  – Why does this work?

Deadlock Prevention: Circular Wait (cont’d)

• 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 ≥ 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

- Deadlock occurs because processes are waiting on each other to release resources
- Main idea of deadlock avoidance:
  - request additional information about how resources are to be requested
  - before allocating request, check if system will enter a deadlock state
    - F (resources available, resources allocated, future requests/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 iff there exists a safe sequence
- A sequence <P₁, P₂, ..., Pₙ> is a safe sequence for the current allocation if, for each Pᵢ, the resources that Pᵢ can still request can be satisfied by the currently available resources plus resources held by all the Pⱼ for 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
    - P₁: max need = 10, current need = 5
    - P₂: max need = 4, current need = 2
    - P₃: max need = 9, current need = 2
    - This is a safe state, since a safe sequence <P₂, P₁, P₃> exists
    - P₃ requests an additional unit. Should this request be granted?
      - No, because this would put the system in an unsafe state
        - P₁, P₂, P₃ will then hold 5, 2, and 3 resources (2 units are available)
        - P₂’s future needs can be satisfied, but no way to satisfy P₁’s and P₃’s needs
- Avoidance algorithms prevent the system from entering an unsafe state

Deadlock Avoidance: Single Resource Instances

- Deadlock = 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 diagram:

- <P₁, P₂, P₃> is a safe sequence
- P₃’s future request can be satisfied because R₃ is available
- P₂’s future request can be satisfied by P₃ yielding R₂
- P₁’s future requests can be satisfied by P₂ giving up R₁ and available R₃

Example scenario:

- Say P₁ requests R₃: should this be granted?
- No, because an assignment edge from R₃ to P₁ would create a cycle in the RAG.

Additional note:

- Does this always imply a deadlock?
- No, because P₁ can release R₃ before requesting R₁
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

resource availability
Available[1..m]
maximum demand
Max[1..n, 1..m]
current allocation
Allocation[1..n, 1..m]
potential need
Need[1..n, 1..m]

1. If Requestᵢ ≤ Needᵢ goto Step 2, else flag error
2. If Requestᵢ ≤ Availableᵢ goto Step 3, else wait
3. Allocate the resources
   Available := Availableᵢ - Requestᵢ;
   Allocationᵢ := Allocationᵢ + Requestᵢ;
   Needᵢ := Needᵢ - Requestᵢ;
   Check if this is a safe state
   If not: undo the allocation and wait

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]]

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 earlier in the lecture)

Detection: Multiple Resource Instances (cont’d)

• 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

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]</td>
<td>[0, 0, 0]</td>
<td>[3, 1, 3]</td>
</tr>
<tr>
<td>P1 [2, 0, 0]</td>
<td>[2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td>P2 [0, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 1, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P4 [0, 0, 2]</td>
<td>[0, 0, 2]</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]</td>
<td>[0, 0, 0]</td>
<td>[0, 1, 0]</td>
</tr>
<tr>
<td>P1 [2, 0, 0]</td>
<td>[2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td>P2 [3, 0, 3]</td>
<td>[0, 0, 1]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 1, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P4 [0, 0, 2]</td>
<td>[0, 0, 2]</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 and 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 combined deadlock solution
  - process control blocks: use resource ordering (prevention) Why?
  - user process memory: use pre-emption (detection/recovery)
  - job resources: require prior claims (avoidance)
  - swappable space: preallocate; no hold and wait (prevention)