Outline

• Announcements
  – Lab 3 due February 28th, 5:00pm
    • Briefly describe your strategy in the write up
  – Demo times: Thursday 6:00pm – 8:00pm, Friday 10:30am – 12:30pm
    – follow “Demo Scheduling” link on web site
  – Midterm exam on March 5th (closed book)
    • review session on February 28th
  – Questions?

• Process deadlocks (contd.)
  – deadlock avoidance
  – deadlock detection
  – deadlock recovery
  – combined approach to deadlock handling

Silberschatz/Galvin: Sections 7.5 – 7.9

Review: Deadlock Prevention

• 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 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, with j<i

P₁’s future requests can be satisfied by P₂ giving up R₁ and available R₃
P₂’s future request can be satisfied by P₃ yielding R₂
P₃’s future request can be satisfied because R₃ is available

<P₃, P₂, P₁> is a safe sequence

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)
      - P₃’s future needs can be satisfied, but no way to satisfy P₁’s and P₂’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

<P₃, P₂, P₁> is a safe sequence

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

1. If Requestᵢ ≤ Needᵢ, goto Step 2, else flag error
2. If Requestᵢ ≤ Availableᵢ, goto Step 3, else wait
3. Allocate the resources
   Availableᵢ := Availableᵢ - Requestᵢ;
   Allocationᵢ := Allocationᵢ + Requestᵢ;
   Needᵢ := Needᵢ - Requestᵢ;
   Check if this is a safe state.
   If not: undo the allocation and wait
4. If Finishᵢ = 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, 0, 2, 1, 0, 0]
  - Need = [[0, 0, 2, 1, 0, 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, 0, 1, 1, 1, 0]
  - Need = [[0, 0, 1, 1, 1, 0], [0, 0, 1, 1, 1, 0]]

Initially, Work = [0, 1, 0]
Need₃ ≤ 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 (contd.)

- 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] [0, 0, 0] [3, 1, 3]</td>
<td>✓ P1 [2, 0, 0] [2, 0, 0]</td>
<td>✓ P2 [3, 0, 3] [0, 0, 0]</td>
</tr>
<tr>
<td>✓ P3 [2, 1, 1] [1, 0, 0]</td>
<td>✓ P4 [0, 0, 2] [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] [0, 0, 0] [0, 1, 0]</td>
<td>P1 [2, 0, 0] [2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td>P2 [3, 0, 3] [0, 0, 1]</td>
<td>P3 [2, 1, 1] [1, 0, 1]</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 7.7 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 preemption 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 Three Classes

- **February 28th**: Review questions for midterm

- **March 5th**: Midterm Exam
  - Silberschatz/Galvin (5th Edition) Chap. 1 – 7 (except Sec. 5.4, 5.6, 6.9)
  - Lectures 1 – 11
  - Nachos Labs 1 – 3 (including mailing list commentary)

- **March 7th**: Memory Management
  - logical versus physical address space
  - swapping
  - allocation
  - paging, segmentation, and hybrids

Reading
- Silberschatz/Galvin: Chapter 8