Outline

- Announcements
  - Write-up 2 due today
  - Lab 3 due March 3rd
  - Midterm review on March 1st
  - Midterm exam on March 6th (closed book)
  - 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
  - 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

  \[ F \] (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: 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 \(<P_1, P_2, \ldots, P_n>\) 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 \(j<i\).

Safe States (contd.)

- 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
  - \(<P_2, P_1, P_3>\) is a safe sequence
  - the system enters an unsafe state if \(P_1\)'s request for an additional unit is granted
    - \(P_1, P_2, P_3\) will then hold 5, 2, and 3 resources (2 units are available)
    - \(P_3\)'s future needs can be satisfied, but no way of satisfying \(P_1\)'s and \(P_2\)'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.

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.

\[
\begin{align*}
1. & \text{if } \text{Request} \leq \text{Need}, \text{goto Step 2, else flag error} \\
2. & \text{if } \text{Request} \leq \text{Available}, \text{goto Step 3, else wait} \\
3. & \text{Allocate the resources} \\
& \text{Available} := \text{Available} - \text{Request}; \\
& \text{Allocation} := \text{Allocation} + \text{Request}; \\
& \text{Need} := \text{Need} - \text{Request}; \\
& \text{Check if this is a safe state} \\
& \text{If not: undo the allocation and wait} \\
\end{align*}
\]
Banker’s Algorithm: Example

- Three resource types and three processes \( (P_1, P_2, P_3) \)
- 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 = \([[0, 0, 2], [1, 1, 0], [0, 1, 0]]\)

- \( P_1 \) requests \([0, 0, 1]\)
  Should this be granted?

- Allocate and check if system is in a safe state

Initially, \( Work = [0, 1, 0] \)
\( Need_3 \leq Work \), 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"

- 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 \( \equiv \) 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
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;
   \text{Finish[i]} := false, for all \text{i};
2. Find an \text{i} such that
   a. \text{Finish[i]} = false, and
   b. Request \text{i} \leq Work
   if no such \text{i}, goto Step 4
3. Work := Work + Allocation\text{i};
   \text{Finish[i]} := true;
   goto Step 2;
4. If \text{Finish[i]} = false for some \text{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>\checkmark \text{P0} [0, 1, 0]</td>
<td>[0, 0, 0]</td>
<td>[3, 1, 3]</td>
</tr>
<tr>
<td>\checkmark \text{P1} [2, 0, 0]</td>
<td>[2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td>\checkmark \text{P2} [3, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>\checkmark \text{P3} [2, 1, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>\checkmark \text{P4} [0, 0, 2]</td>
<td>[0, 0, 2]</td>
<td></td>
</tr>
</tbody>
</table>

- What about the following?

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

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 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: can 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

- **March 1st**: Review questions for midterm

- **March 6th**: 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 8th**: Memory Management
  - logical versus physical address space
  - swapping
  - allocation
  - paging, segmentation, and hybrids

*Reading*

- Silberschatz/Galvin: Chapter 8