ECS150 Discussion Section

Sophie Engle
(February 11/13 2004)
Announcements

- Website updated!
  - Added information on compiling the Minix kernel
  - Added more hints & tips for using Minix
  - Added more common compile errors
  - Updated grades

- Will have to know how to compile the Minix kernel for next programming assignment!
Deadlocks

- Resources
  - Tanenbaum p75 – 77
  - Tanenbaum p166 – 179
Resources

- Resource
  - Anything that can only be used by a single process at any instant

- Types
  - **Preemptable**
    - Resource can be taken away
    - Example: Memory
  - **Nonpreemptable**
    - Resource can NOT be taken away
    - Example: Printer
Nonpreemptable Resources

1. Request
2. Use
3. Release
Definition:
- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Example:
- Process 1 is holding resource A, needs B
- Process 2 is holding resource B, needs A
Dining Philosophers

- Must have 2 forks to eat
- Everyone picks up left fork at same time and waits for right fork
- Nobody is able to eat, deadlock occurs

Figure 2-16. Lunch time in the Philosophy Department.
Deadlock Conditions

- Necessary Conditions:
  - Mutual exclusion condition
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition
Necessary Conditions:

- Mutual exclusion condition
  - Resource assigned to exactly one process (or none)
- Hold and wait condition
  - Can request resource at any time
- No preemption condition
  - Granted resources can not be forcibly taken away
- Circular wait condition
  - Must be cycle of processes waiting on each other
Nodes:
- Processes: Circle
- Resources: Square

Arcs:
- Resource A → Process 1: P1 holds A
- Process 1 → Resource A: P1 waiting for A
### Deadlock Example

<table>
<thead>
<tr>
<th>Process #</th>
<th>Required Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>A and B</td>
</tr>
<tr>
<td>Process 2</td>
<td>B and C</td>
</tr>
<tr>
<td>Process 3</td>
<td>C and A</td>
</tr>
</tbody>
</table>
Events:
P1 requests A
Events:
P1 requests A
P2 requests B
Deadlock Example

Events:
P1 requests A
P2 requests B
P3 requests C
Deadlock Example

Events:
P1 requests A
P2 requests B
P3 requests C
P1 requests B

(notice example of hold and wait)
Deadlock Example

Events:
- P1 requests A
- P2 requests B
- P3 requests C
- P1 requests B
- P2 requests C
Events:
P1 requests A
P2 requests B
P3 requests C
P1 requests B
P2 requests C
P3 requests A

(notice example of circular wait)
Handling Deadlocks

- Possible responses:
  - Ignore
  - Detect (and recover)
  - Avoid
  - Prevent
Handling Deadlocks

Possible responses:

- Ignore
  - Are deadlocks such a bad problem?

- Detect (and recover)
  - Watch for deadlocks, fix when happens

- Avoid
  - Make good decisions!

- Prevent
  - Eliminate one condition required for deadlocks
Why ignore deadlocks?
- How often will deadlocks occur?
- How often will system crash anyway?
- Is deadlock detection, prevention, or avoidance really cost effective?

Ostrich Algorithm
- Ignore problem of deadlocks completely
## Detection and Recovery

<table>
<thead>
<tr>
<th>Method</th>
<th>Detection</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Keep resource graph, check for cycles</td>
<td>Kill a process in the cycle until cycle is broken</td>
</tr>
<tr>
<td>2</td>
<td>Check how long process has been continuously blocked</td>
<td>Kill process that has been continuously blocked for long time</td>
</tr>
</tbody>
</table>
Avoidance

- Decide if resource should be granted
  - Want to keep system in a “safe” state
  - Requires certain information in advance
  - Great in theory, but often impractical

- Banker’s Algorithm for Multiple Resources
  - Often discussed avoidance algorithm
  - Rarely used in practice
Banker’s Algorithm

- Tracks current state of system
- Prevents state from becoming unsafe
  - A state is safe if there exists some sequence that allows every process to run to completion
Banker’s Algorithm

- State tracked by:
  - 2 Process by Resource Matrices
    - Resources assigned
    - Resources still needed
  - 3 Resource Vectors
    - (E)xisting resources
    - (P)ossessed resources
    - (A)vailable resources
To check if initial state safe:

1. Find process whose needs are less than the available resources
   - If none exist, then state not safe (deadlock possible)

2. Assume process completes, and add its resources as available

3. Repeat steps 1 and 2 until all processes are terminated
   - If possible, then initial state safe
# Banker’s Algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>existing</th>
<th>possessed</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>tape drives</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>plotters</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>printers</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>cd-roms</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Process = D

<table>
<thead>
<tr>
<th></th>
<th>assigned</th>
<th></th>
<th>still needed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A  B</td>
<td>C  D</td>
<td>E</td>
<td>A  B</td>
</tr>
<tr>
<td>tape drives</td>
<td>3  0  1  1  1  0</td>
<td>1  0  3  0  2  2</td>
<td>existing</td>
<td>6  5  1</td>
</tr>
<tr>
<td>plotters</td>
<td>0  1  1  1  1  0</td>
<td>1  1  1  0  1  1</td>
<td>possessed</td>
<td>3  3  0</td>
</tr>
<tr>
<td>printers</td>
<td>1  0  1  0  0  0</td>
<td>0  1  0  1  1  1</td>
<td>available</td>
<td>4  2  2</td>
</tr>
<tr>
<td>cd-roms</td>
<td>1  0  0  1  0  0</td>
<td>0  2  0  0  0  0</td>
<td></td>
<td>2  2  0</td>
</tr>
</tbody>
</table>
### Banker’s Algorithm

Process = A

<table>
<thead>
<tr>
<th></th>
<th>assigned</th>
<th>still needed</th>
<th>existing</th>
<th>possessed</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E</td>
<td>A B C D E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tape drives</td>
<td>3 0 1 1 0</td>
<td>1 0 3 0 2</td>
<td>6 4 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>plotters</td>
<td>0 1 1 1 0</td>
<td>1 1 1 0 1</td>
<td>3 2 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printers</td>
<td>1 0 1 0 0</td>
<td>0 1 0 1 1</td>
<td>4 2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cd-roms</td>
<td>1 0 0 1 0</td>
<td>0 2 0 0 0</td>
<td>2 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eventually, all will be able to run. Initial state = safe

<table>
<thead>
<tr>
<th></th>
<th>assigned</th>
<th>still needed</th>
<th>existing</th>
<th>possessed</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>tape drives</td>
<td>A B C D E</td>
<td>A B C D</td>
<td>3 0 1 1 0</td>
<td>1 0 3 0 2</td>
<td>6 2 4</td>
</tr>
<tr>
<td>plotters</td>
<td>0 1 1 1 0</td>
<td>1 1 1 0 1</td>
<td>3 2 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printers</td>
<td>1 0 1 0 0</td>
<td>0 1 0 1 1</td>
<td>4 2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cd-roms</td>
<td>1 0 0 1 0</td>
<td>0 2 0 0 0</td>
<td>2 0 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deadlock Prevention

- Impose restrictions that make deadlocks structurally impossible
  - Prevent at least one of four conditions required for deadlock to occur

- How prevent deadlock?
  - Prevent mutual exclusion?
  - Prevent hold and wait condition?
  - Prevent no preemption condition?
  - Prevent circular wait condition?
Deadlock Conditions

- **Necessary Conditions:**
  - Mutual exclusion condition
    - Resource assigned to exactly one process (or none)
  - Hold and wait condition
    - Can request resource at any time
  - No preemption condition
    - Granted resources can not be forcibly taken away
  - Circular wait condition
    - Must be cycle of processes waiting on each other