CS162 Discussion section
Week 4
Overview

• Project 1 submissions
• SVN repository setup
• Synchronization in Java
• Monitors
• Deadlock
Project 1

• Grading
  – Individual portion (40%)
  – Group portion (60%)

• Group portion breakdown
  – First draft [10 points]
  – Design Review [5 points]
  – Final design doc [25 points]
  – Code [60 points]
Design document

• Should include...
  – Introduction to the overall behavior of the system
  – Technical challenges and solutions
    • Solution overview (2-4 lines)
    • Correctness constraints
    • Data structures
    • Design decisions
    • Description of java classes and methods
    • Pseudocode for key algorithms
    • Test plan and test cases
Design document (2)

• First draft should be between 1-2k lines
  – Or about 5-10 pages
• Final draft should be 2-4k lines
  – 10-20 pages
• Include diagrams that show interaction between components and examples
Four requirements for Deadlock

• **Mutual exclusion**
  – Only one thread at a time can use a resource.

• **Hold and wait**
  – Thread holding at least one resource is waiting to acquire additional resources held by other threads

• **No preemption**
  – Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

• **Circular wait**
  – There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
    • \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    • \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    • ...
    • \( T_n \) is waiting for a resource that is held by \( T_1 \)
Resource-Allocation Graph

• System Model
  – A set of Threads $T_1, T_2, \ldots, T_n$
  – Resource types $R_1, R_2, \ldots, R_m$
    
    $CPU$ cycles, memory space, $I/O$ devices
  – Each resource type $R_i$ has $W_i$ instances.
  – Each thread utilizes a resource as follows:
    - Request() / Use() / Release()

• Resource-Allocation Graph:
  – $V$ is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  – request edge – directed edge $T_1 \rightarrow R_j$
  – assignment edge – directed edge $R_j \rightarrow T_i$
Resource Allocation Graph
Examples

• Recall:
  – request edge – directed edge $T_1 \rightarrow R_j$
  – assignment edge – directed edge $R_j \rightarrow T_i$
# Banker’s algorithm

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>What it stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>avail</td>
<td>num resources x 1</td>
<td>avail[i] is the number of available and indistinguishable instances of the resource</td>
</tr>
<tr>
<td>Max</td>
<td>num threads x num resources</td>
<td>max[i][j] is the maximum number of instances of resource j that thread i will request</td>
</tr>
<tr>
<td>Allocation</td>
<td>num threads x num resources</td>
<td>allocation[i][j] is the number of instances of resource j that have been allocated to thread i</td>
</tr>
<tr>
<td>Need</td>
<td>num threads x num resources</td>
<td>need[i][j] is the max number of instances of resource j that thread i still needs to complete its task. Notice that need[i][j] = max[i][j] - allocation[i][j] when we don’t overestimate the amount of resources we need</td>
</tr>
</tbody>
</table>
Banker’s algorithm(2)
Safety Check

Algorithm 1 Algorithm for finding out if a system is in a safe

state let work = an array of length num resources
let finish = an array of length num threads
work = avail
for each thread i set finish[i] to false
while there exists a thread i such that finish[i] == false and need[i][:] ≤ work do
    work = work + allocation[i][:]
    finish[i] = true
end while
if finish[i] == true ∀i then
    system is safe
else
    deadlock is possible
end if
Banker’s algorithm(3)
Request Granting

Algorithm 2 algorithm for requesting a resource
allocation let $request[i][]$ be the request vector for thread $i$

if $request[i][] \leq need[i][]$ then
    if $request[i][] \leq avail[i][]$ then
        wait because there aren’t enough free resources
    else
        pretend to modify the system
        $avail = avail - request[i][]$
        $allocation[i][] = allocation[i][] + request[i][]$
        $need[i][] = need[i][] - request[i][]$
    end if
else
    undo the changes to avail, allocation, and
    need wait since allocation could cause
deadlock and
    try the request later once resources have cleared up
endif
end if
else
    signal an error because we have requesting more than our max possible requests
endif