The UNIVERSITY of VERMONT

Benjamin Fenster
CS-201 (Arslan)
11 December 2005

Notes – Deadlocks

I. Definition
   a. A set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.
   b. Example: P_1 has A and needs B. P_2 has acquired B and needs A.
   c. Bridge-Crossing Example:
      i. Have a one-lane bridge
      ii. A car enters from each end so they end up facing off in the middle.
      iii. One car has to back up and let traffic from the other side go through.
   d. Necessary Conditions for a Deadlock
      i. Mutual exclusion: Only one process may use resource R at a time.
      ii. Hold and Wait: A process is holding at least one resource and is waiting for other resources (can't deadlock if you only get to use one thing at a time)
      iii. No preemption: A resource can be released voluntarily only by the process holding it. (If the system can preempt there won't be a deadlock for long.)
      iv. Circularity: There's a circular wait relationship (to be explored in detail later)

II. Resource Allocation Graph
   a. Each process is a vertex
   b. Each resource is a vertex.
   c. Arrows from processes to resources indicate requests.
   d. Arrows from resources to processes indicate what's currently being held.
   e. We don't really distinguish between instances of a resource; just draw from one vertex to another.

III. Deadlock Avoidance
   a. More time spent trying to avoid deadlocks = more overhead
   b. None of these algorithms is 100% accurate, just conservative avoidance techniques
   c. The Four Conditions
      i. Mutual Exclusion: We can't do anything about this. We need it for non-shared resources (though sharable resources don't require mutual exclusion)
      ii. Hold and Wait
         1. Could restrict processes to holding no more than one resource (not very realistic)
         2. Could require processes to request and acquire all resources before beginning execution
            a. Better!
            b. Like two-phase locking
            c. Imposing this guarantees no deadlocks, since no processes can hold some resources and then request others.
            d. Doesn't yield an efficient use of resources though. It prevents processes from executing just to avoid the chance of a deadlock!
            e. Some processes may starve too – one needs lots of resources but other little processes continually hold some of them
         3. No Preemption
            a. Just allow preemption!
            b. If a process holding some resources requests a new one that's not available, force it to release everything it's holding.
            c. This certainly eliminates deadlocks. Whenever there's a chance of a deadlock, just bail out!
         4. Circular Wait
            a. Create a total ordering of all resources.
            b. A process needs to request resources in increasing order.
            c. Certainly guarantees no deadlocks.
            d. Also creates poor CPU utilization – works much like the solution for "hold and wait."

d. Safe State
    i. The system is in a safe state if there's some execution order by which all processes will finish at the end.
    ii. Depends really on the order in which resources are requested and released.

IV. Banker's Algorithm
a. Processes declare maximum need for resources.
b. If a request cannot be granted, the process must wait.
c. When a process completes, resources are all freed.
d. Terms:
    i. n: number of processes
    ii. m: number of resource types
    iii. Available[J] = k (k available instances of resource type J)
    iv. Max[i, J] = k (process P_i requested maximum k of resource J at)
    v. Allocation[i, J] = k (process P_i has already gotten k instances of J allocated)
    vi. Need[i, J] = k (process P_i needs k additional instances of J).
    vii. So Need[i, J] = Max[i, J] – Allocation[i, J]
e. Safety Algorithm
    i. Let work and finish be n x m matrices.
        1. work = Available
        2. finish[i] = false for all processes
    ii. Find an i such that:
        1. finish[i] = false
        2. Need[i] ≤ Work (one whole row, i.e. all resources)
        3. If not found, go to step 4.
    iii. Work = Work + Alocation[i]
        1. finish[i] = true
        2. go to step 2
    iv. if finish[i] == true for all i then the system is in a safe state, otherwise it's not.
    v. The Idea:
    vi. Find a process whose demand for resources can be met right now.
    vii. If it's found, execute and release its resources.
    viii. If no such process was found…
        1. If it's because the processes are all done, okay.
        2. If it's because all processes demand too much, then it's not a safe state
f. Resource Request Algorithm
    i. Assume process P_i requests resources
    ii. Let Request[i] be the request vector such that: If Request[i,J] = k, P_i requests an additional k instances of type J
    iii. Should the request be granted?
        1. If Request[i] ≤ Need[i] go to step 2 (otherwise it's a violation; halt!)
        2. If Request[i] ≤ Available[i]) go to step 3 (otherwise P_i must wait)
        3. Pretend to allocate requested resources to P_i
            a. Available = Request[i]
            b. Allocation[i] += Request[i]
            c. Need[i] -= Request[i]
        4. Run safety algorithm
            a. If safe, really allocate resources to P_i
            b. If unsafe, P_i must wait. Restore old Available, Allocation, and Need (no more pretending)

V. Deadlock Detection
a. Allow deadlocks to occur, but then detect them.
b. Steps
    i. Step 1
        1. Let work be a vector of size m
        2. Let finish be a vector of size n

3. work = Available
4. If Allocation[i] == 0 then finish[i] = true else finish[i] = false

ii. Step 2: Find an i such that finish[i] = false for all i and request[i] ≤ work. If none found, go to step 4.

iii. Step 3
1. Work += Allocation[i] (finishes, returns resources)
2. finish[i] = true
3. Go to step 2.

iv. Step 4: If finish[i] = false for some i then the system is deadlocked. (P_i is deadlocked).

c. This algorithm correctly assesses whether or not the system is in deadlock.

VI. Recovery from Deadlocks

a. Once we've detected a deadlock, select a "victim" process and roll it back.
b. Want a low-priority process to roll back if possible.
c. Also want a victim that hasn't been running long (so we'll have less work to do in rolling it back)
d. Don't want any one process to be selected repeatedly as a victim or it'll starve.
e. This algorithm can run *much* less frequently than deadlock avoidance since there's no real hurry – the deadlock will be there forever if we don't get around to checking quickly.