Announcements

- Midterm is Thursday (3/6/14)
  - Covers up through definition of deadlock (last Th lecture)
  - Summary of reading assignments on web
- Project #2 is due today at 5:00 PM
- Project #1grades are now posted in grades
  - Re-grade request deadline is 3/11/14

Detecting Deadlock Algorithm

- Variables:

  n is the number of processes

  m is the number of resource types

  – Available - vector of length m indicating the number of available resources of each type
  – Work - vector of length m indicating the number of currently available resources of each type
  – Allocation - n by m matrix defining number of resources of each type currently allocated to each process
  – Request is an m x n matrix indicating the number of additional resources requested by each process
  – Finish is a vector of length n (processes) indicating if we are finished checking that process

Detecting Deadlock

1. Work = Available;

   foreach i in n

      if any of Allocation[i,*] != 0 Finish[i] = false

      else Finish[i] = true;

2. Find an *i* such that Finish[i] = false and

      Request[I,*] <= Work[i,*] if no such i, go to 4

3. Work[i,*] += Allocation[i,*] ;

   Finish[i] = true;

   goto step 2

4. If Finish[i] = false for some i, system is in deadlock

Note: this requires m x n^2 steps

Recovery from deadlock

- Must free up resources by some means
- Process termination
  - kill all deadlocked processes
  - select one process and kill it
    - must re-run deadlock detection algorithm again to see if it is freed.
- Resource Preemption
  - select a process, resource and de-allocate it
  - rollback the process
    - needs to be reset the process to a safe state
    - this requires additional state
  - starvation
    - what prevents a process from never finishing?

Deadlock Prevention

Ensure that:

one or more of the necessary conditions for deadlock do not hold

- Hold and wait
  - guarantee that when a process requests a resource, it does not hold any other resources
  - Each process could be allocated all needed resources before beginning execution
  - Alternately, process might only be allowed to wait for a new resource when it is not currently holding any resource

Deadlock Prevention

- Mutual exclusion
  - Sharable resources do not require mutually exclusive access and cannot be involved in a deadlock.
- Circular wait
  - Impose a total ordering on all resource types and make sure that each process claims all resources in increasing order of resource type enumeration
- No Premption
  - virutalize resources and permit them to be prempted. For example, CPU can be prempted.

Deadlock Avoidance

- Require additional information about how resources are to be requested - decide to approve or disapprove requests on the fly
- Assume that each process lets us know its maximum resource request
- Safe state:
  - system can allocate resources to each process (up to its maximum) in *some order* and still avoid a deadlock
  - A system is in a safe state if there exists a *safe sequence*

Safe Sequence

- Sequence of processes <P_1, .. P_n> is a safe sequence if for each P_i, the resources that P_i can request can be satisfied by the currently available resources plus the resources held by all P_j, $j<i$
- If the necessary resources are not immediately available, P_i can always wait until all P_j, $j<i$ have completed

Banker's Algorithm

- Each process must declare the maximum number of instances of each resource type it may need
- Maximum can't exceed resources available to system
- Variables:

  n is the number of processes

  m is the number of resource types

  – Available - vector of length m indicating the number of available resources of each type
  – Max - n by m matrix defining the maximum demand of each process
  – Allocation - n by m matrix defining number of resources of each type currently allocated to each process
  – Need: n by m matrix indicating remaining resource needs of each process
  – Work: a vector of length m (resources)
  – Finish: a vector of length n (processes)

Safe State Predicate

1. Work = Available; Finish[*] = false
2. Find an *i* such that Finish[i] = false

   and Need[i,*] <= Work[i,*] if no such i, go to 4

   all elements in the vector are <=
3. Work[i,*] += Allocation[i,*];

   Finish[i] = true;

   goto step 2
4. If Finish[i] = true for all i, system is in a safe state

Note this requires m x n^2 steps

Safe State Predicate - Example

Three resources: A, B, C (10, 5, 7 instances each)

Consider the snapshot of the system at this time

	Alloc	Max	Avail	Max - alloc Need
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

System is in a safe state, since the sequence <P1, P3, P4, P2, P0> satisfy the safety criteria.

Resource Request Algorithm

(1) If $Request_i$ <= $Need_i$ then goto 3

- otherwise - the process has exceeded its maximum claim

(2) If $Request_i$ <= Available then goto 3

- otherwise process must wait since resources are not available

(3) Check request by having the system pretend that it has allocated the resources by modifying the state as follows:

- Available = Available - $Request_i$
- Allocation = Allocation + $Request_i$
- $Need_i$ = $Need_i$ - $Request_i$

- Find out if resulting resource allocation state is safe, otherwise the request must wait.