Announcements

- Reading 7 (7.5-7.9)
- Midterm #1 is March 10 in class
  - covers material through and including lecture 11
    - problems at the end of the chapters
    - synchronization problems
    - questions about the project

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 is a vector of length m (resources)
- Finish is a vector of length n (processes)

1. Work = Available; Finish = false
2. Find an *i* such that Finish[i] = false and $Need_i$ <= Work if no such i, go to 4
3. Work += $Allocation_i$; Finish[i] = true; goto step 2
4. If Finish[i] = true for all i, system is in a safe state

all elements in the vector are <=

Note this requires m x n^2 steps

Banker's Algorithm - Example

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

Consider the snapshot of the system at this time

	Alloc	Max	Avail	Need (Max - alloc)
	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.

Deadlock Detection

- Resource Allocation Graph
  - Graph consists of vertices
    - type $P = \{P_1,..,P_n\}$ represent processes
    - type $R = \{R_1,..,R_m\}$ represent resources
  - Directed edge from process P_i to resource type R_j signifies that a process i has requested resource type j
  - *request edge*
  - A directed edge from R_j to P_i indicates that resource R_j has been allocated to process P_i
  - *assignment edge*

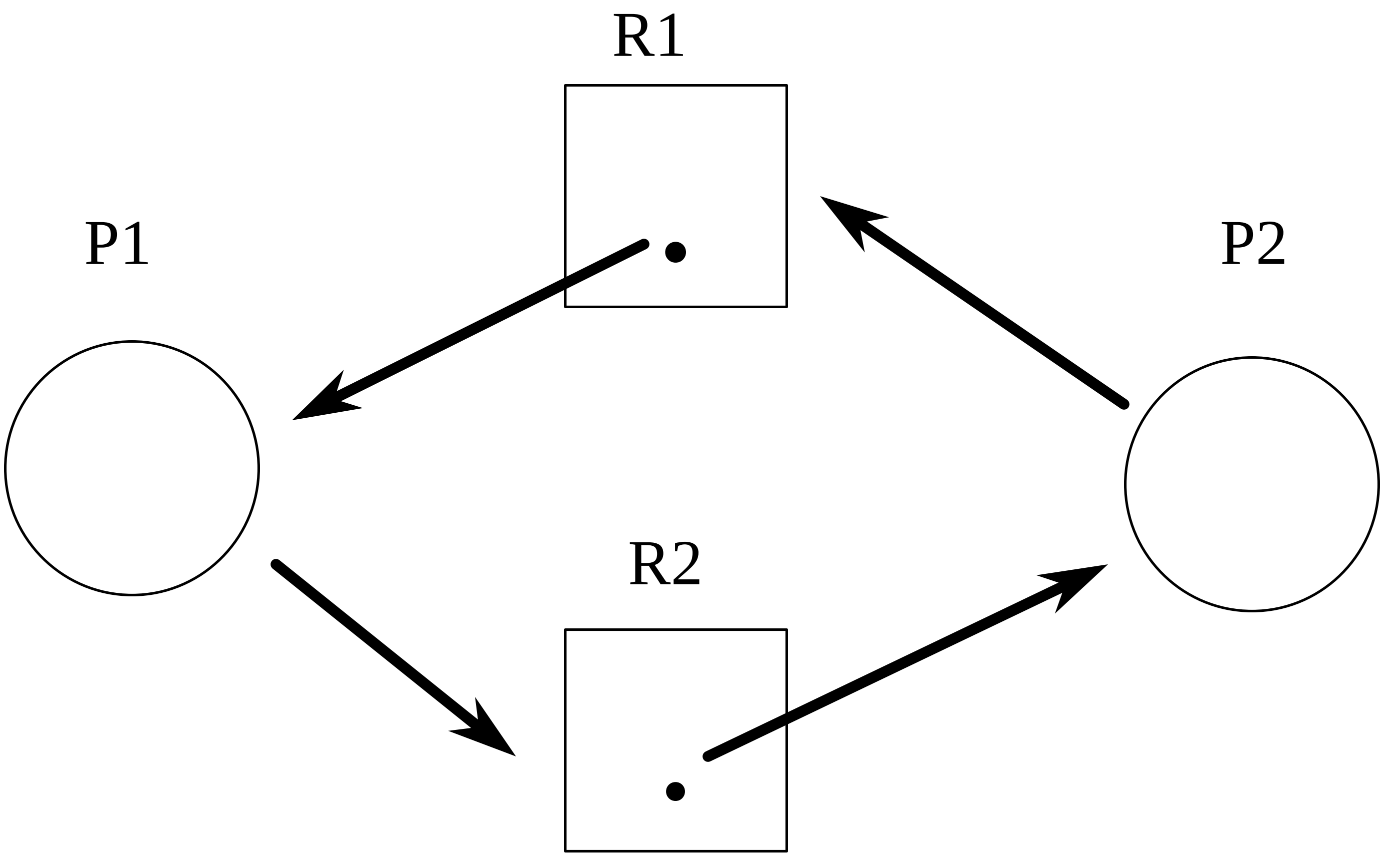
Deadlock Detection (cont.)

- Resource types may have more than one instance
- Each resource vertex represents a resource type.
- Each resource instance is of a unique resource type, each resource instance is represented by a "subvertex" associated with a resource vertex
  - (Silverschatz represents resource vertices by squares, resource instance "subvertices" by dots in the square. Process vertices are represented by circles)
- A request edge points to a resource vertex
- An assignment edge points from a resource "subvertex" to a process vertex

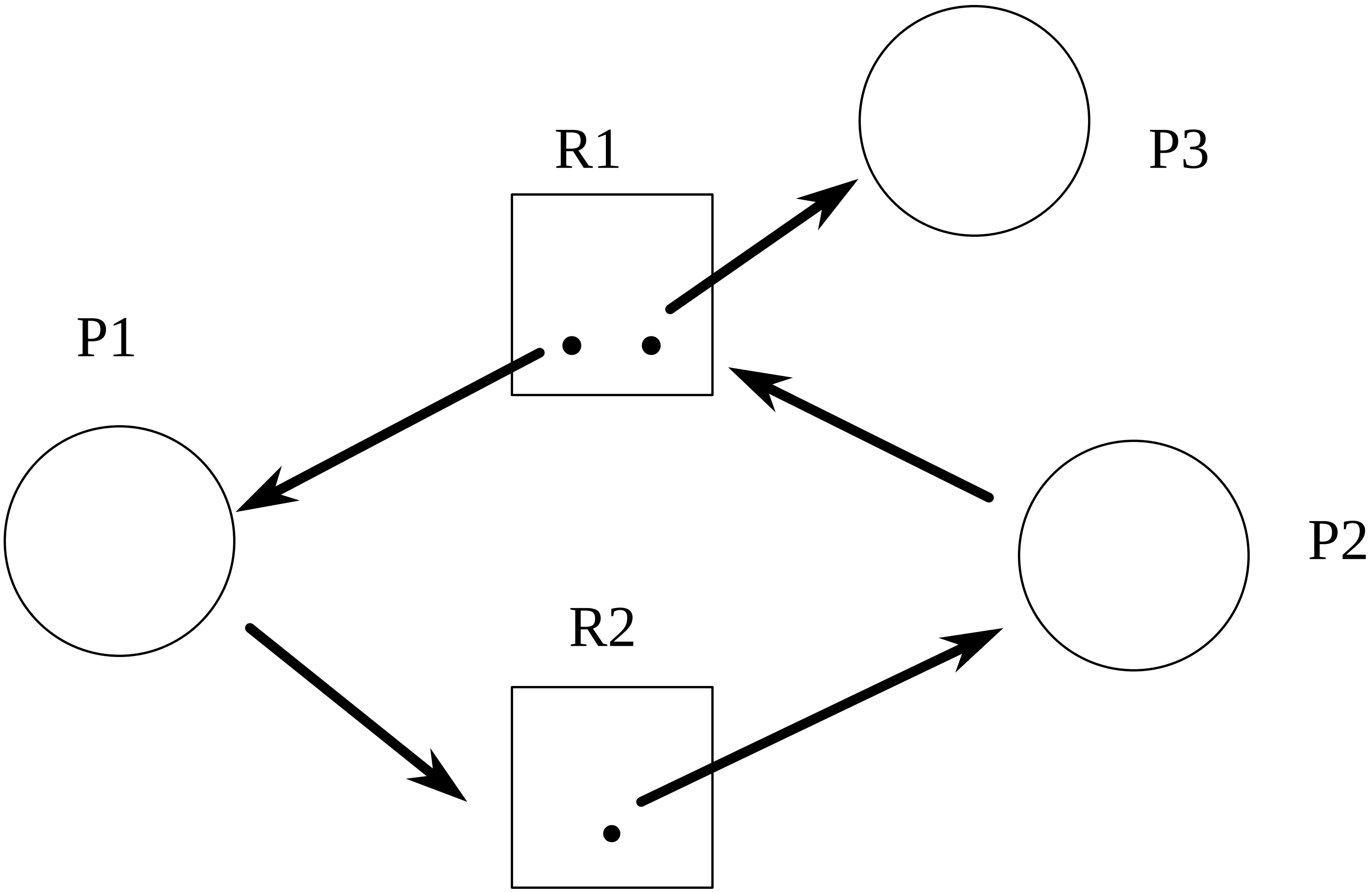
Resource Allocation Graph

- When a process P_i requests an instance of resource type R_j, a request edge is inserted into the resource allocation graph
- When the request can be fulfilled, the request edge is transformed into an assignment edge
- When the process is done using the resource, the assignment edge is deleted
- If the graph contains no cycles, no deadlock can exist

Deadlock!

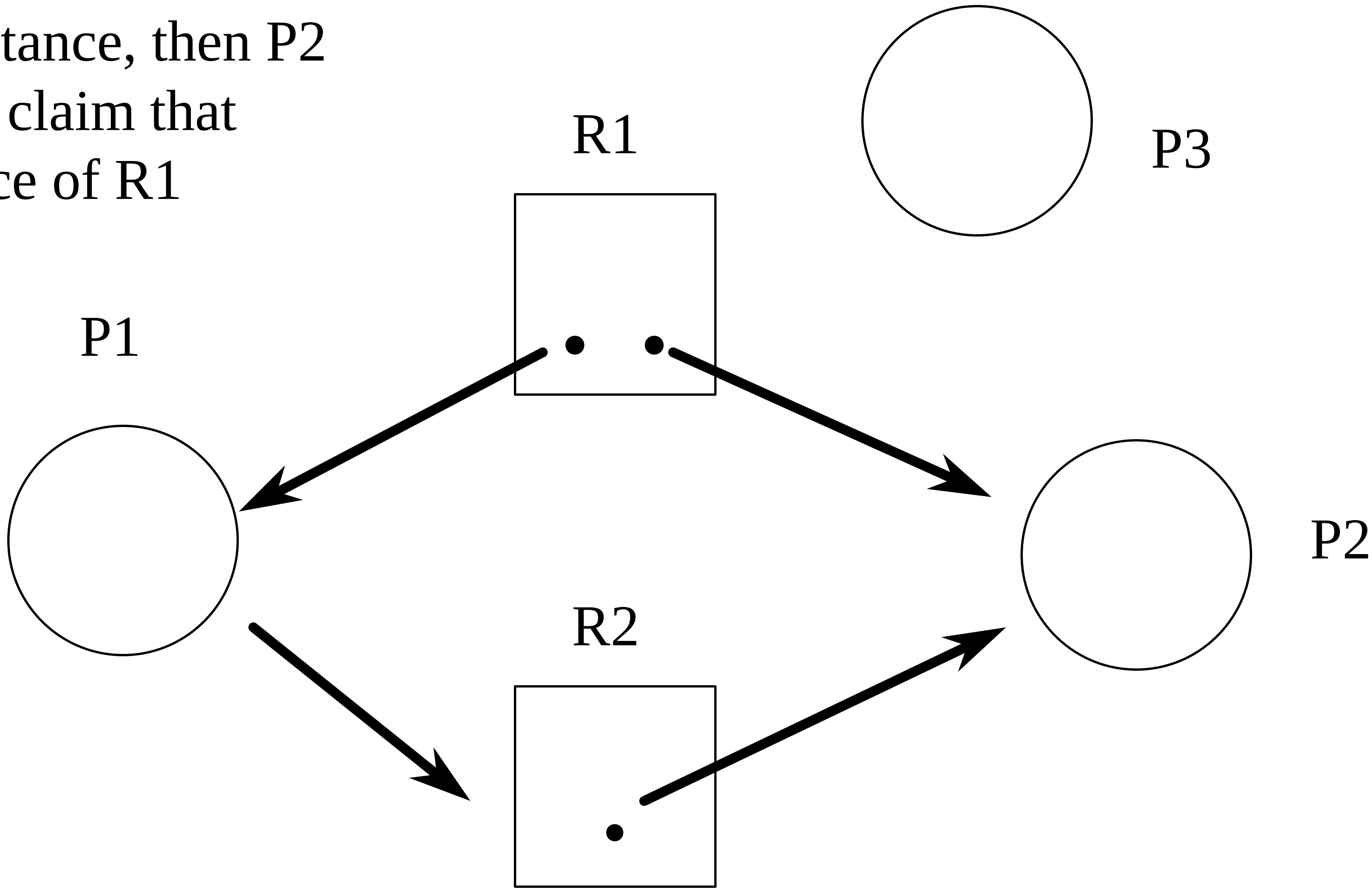


Deadlock??

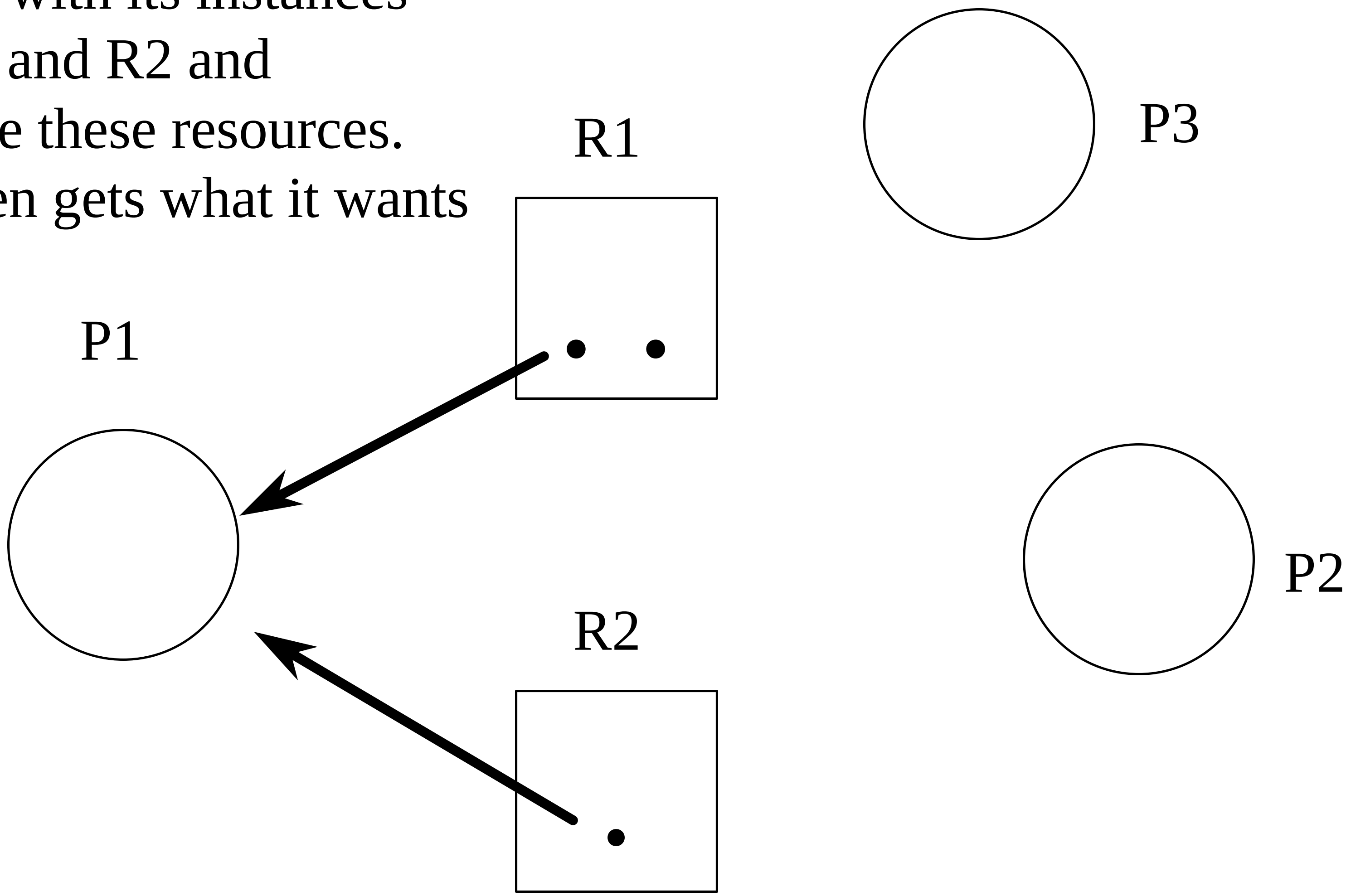


No!!

P3 could finish with its instance of R1, release the instance, then P2 would claim that instance of R1



Then, P2 could finish with its instances of R1 and R2 and release these resources. P1 then gets what it wants



Detecting Deadlock

Work is a vector of length m (resources)

Finish is a vector of length n (processes)

- Allocation is an n x m matrix indicating the number of each resource type held by each process
- Request is an m x n matrix indicating the number of additional resources requested by each process

1. Work = Available;

   This is the difference from the Banker's algorithm.

   if Allocation[i] != 0 Finish = false else Finish = true;

2. Find an *i* such that Finish[i] = false and $Request_i$ <= Work if no such i, go to 4
3. Work += Allocation ; 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?