So far…

▸ *Discussed deadlocks, conditions for deadlocks to happened, mechanisms to prevent deadlock and mechanisms to avoid deadlocks*

   ■ Avoiding deadlocks means that you don't go into an unsafe state

Banker's Algorithm

▸ Multiple instances of resources

▸ Each process must a priori claim the maximum amounts of resources to use.

▸ When a process requests a resource it may have to wait

■ When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types

▸ *Available:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.

▸ *Max:* n x m matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.

▸ *Allocation:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.

▸ *Need:* n x m matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

Need $[i,j]$ = *Max*$[i,j]$ – *Allocation* $[i,j]$.

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish [*i*] = *false* for *i* = 0,1, …, *n-1*

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. *Work = Work + Allocation$_i$*
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j]$ = *k* then process P_i wants *k* instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;

   - λ *If safe* ⇒ *the resources are allocated to Pi.*
   - λ *If unsafe* ⇒ *Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

▸ 5 processes P_0 through P_4; 3 resource types A(10 instances), B(5 instances) and C(7 instances)

▸ Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

▸ The content of the matrix. Need is defined to be Max – Allocation.

$$Need$$

	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

▸ The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

▸ Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

▸ Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.

▸ Can request for (3,3,0) by P4 be granted?

▸ Can request for (0,2,0) by P0 be granted?

Deadlock Detection

▸ Allow system to enter deadlock state

▸ Detection algorithm

▸ Recovery scheme

Single Instance of Each Resource Type

- ▸ Maintain *wait-for* graph
  - ■ Nodes are processes.
  - ■ $P_i \rightarrow P_j$ if P_i is waiting for P_j.

- ▸ Periodically invoke an algorithm that searches for a cycle in the graph.

- ▸ An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph

(a)

(b)
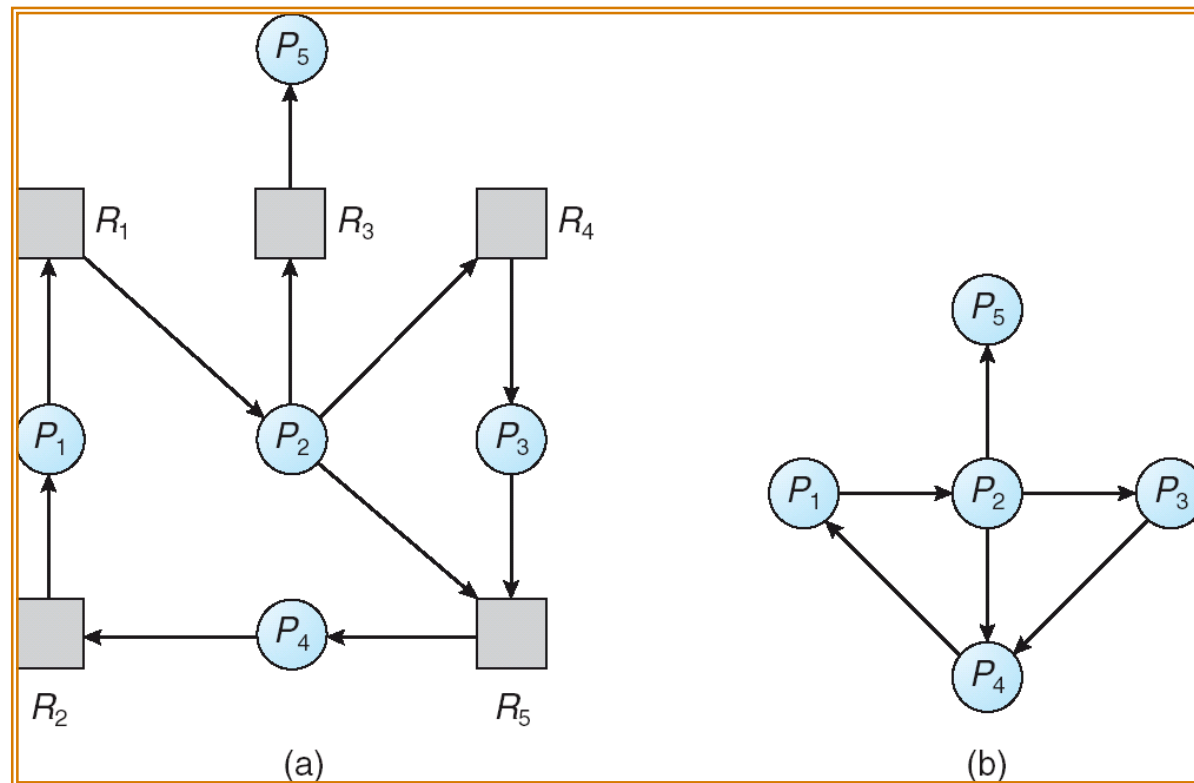
Resource-Allocation Graph   Corresponding wait-for graph

Several Instances of a Resource Type

▸ *Available:* A vector of length m indicates the number of available resources of each type.

▸ *Allocation:* An n x m matrix defines the number of resources of each type currently allocated to each process.

▸ *Request:* An n x m matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting k more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

   (a) *Work = Available*

   (b) For *i* = 0,1, …, *n-1*, if *allocation$_i$* ≠ 0, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b) *Request$_i$* ≤ *Work*

   If no such *i* exists, go to step 4.

3. *Work = Work + Allocation$_i$*
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, 1 ≤ *i* ≤ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then *P$_i$* is deadlocked.

Example of Detection Algorithm

▸ Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).

▸ Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

▸ Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[i] = true for all i

Example (Cont.)

▸ P_2 requests an additional instance of type C.

<u>*Request*</u>

A B C

P_0 0 0 0

P_1 2 0 1

P_2 0 0 1

P_3 1 0 0

P_4 0 0 2

▸ State of system?

■ Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.

■ Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

▸ When, and how often, to invoke depends on:

 ■ How often a deadlock is likely to occur?

 ■ How many processes will need to be rolled back?

  ● one for each disjoint cycle

▸ If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

▶ Abort all deadlocked processes

▶ Abort one process at a time until the deadlock cycle is eliminated

▶ In which order should we choose to abort?

- Priority of the process

- How long process has computed, and how much longer to completion

- Resources the process has used.

- Resources process needs to complete.

- How many processes will need to be terminated.

- Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

▸ Selecting a victim – minimize cost.

▸ Rollback – return to some safe state, restart process for that state.

▸ Starvation – same process may always be picked as victim, include number of rollback in cost factor.