Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Silberschatz and Galvin©1999

Safe State

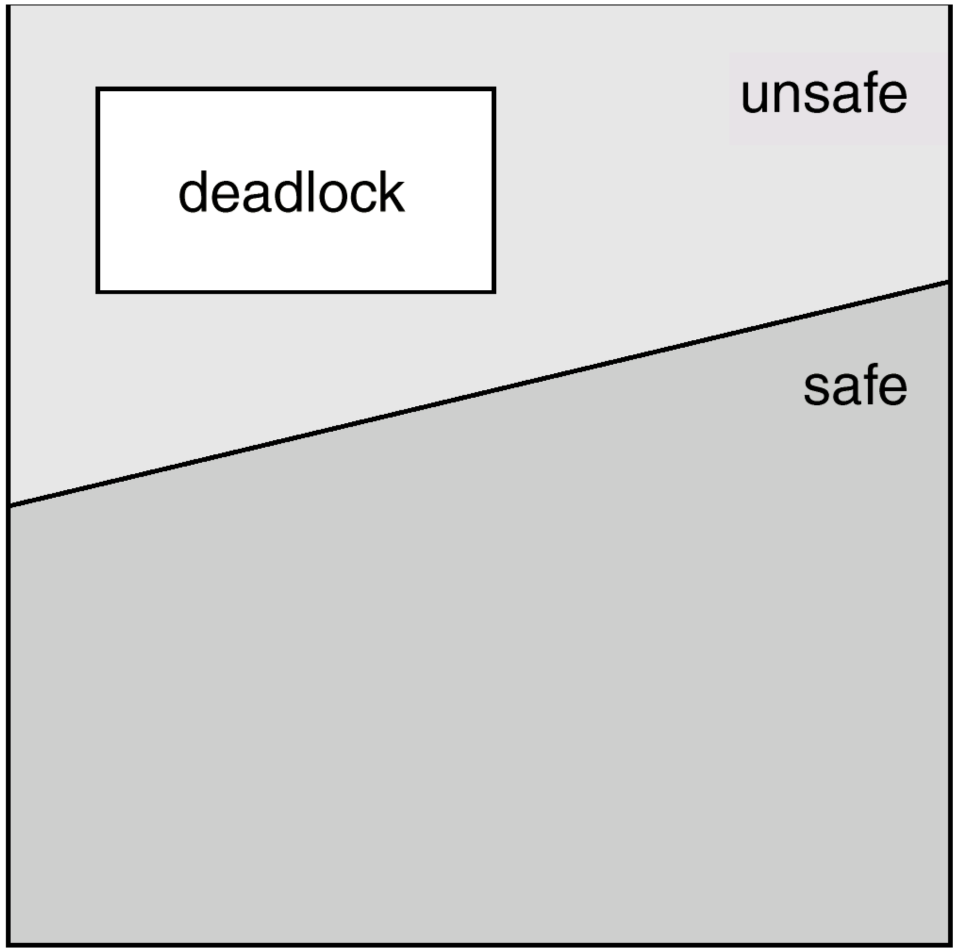
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P_1, P_2, …, P_n> is safe if for each P_{i}, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.
  - If $\mathrm{P_i}$ resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_{i} can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Silberschatz and Galvin©1999

Safe, unsafe , deadlock state spaces



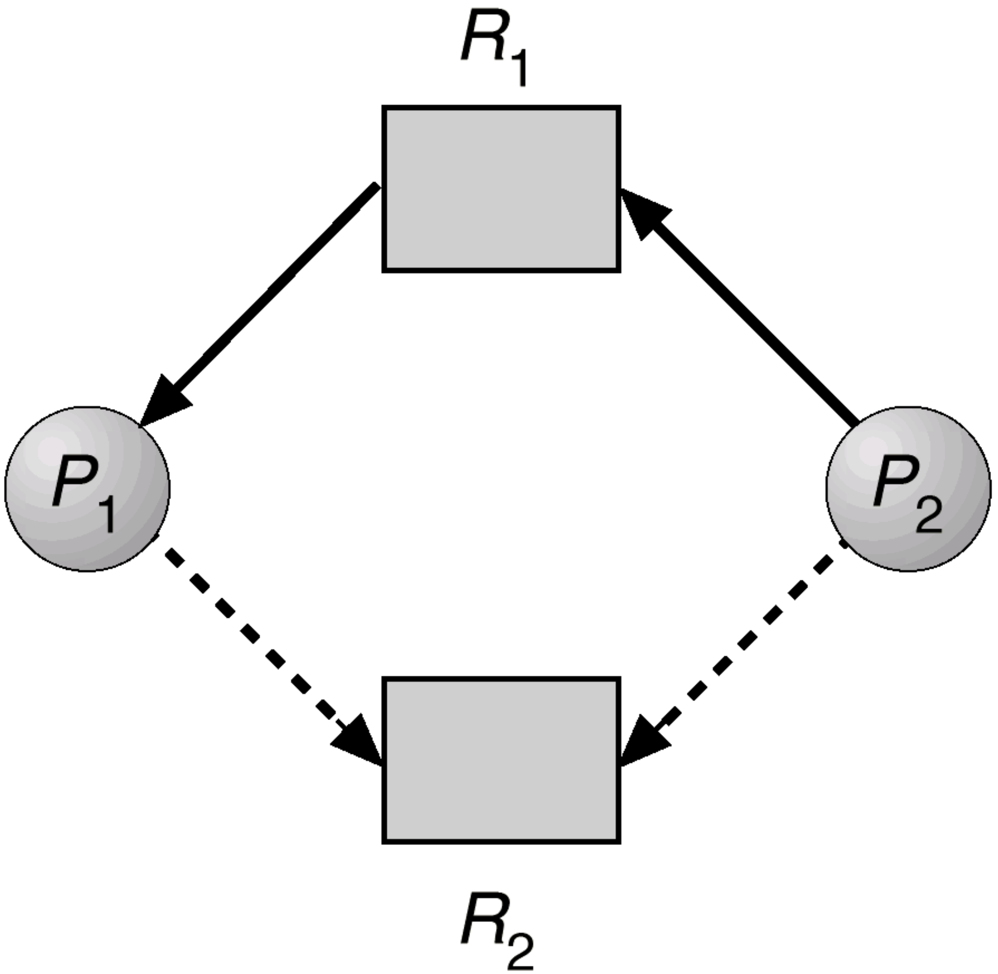
Silberschatz and Galvin©1999

Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

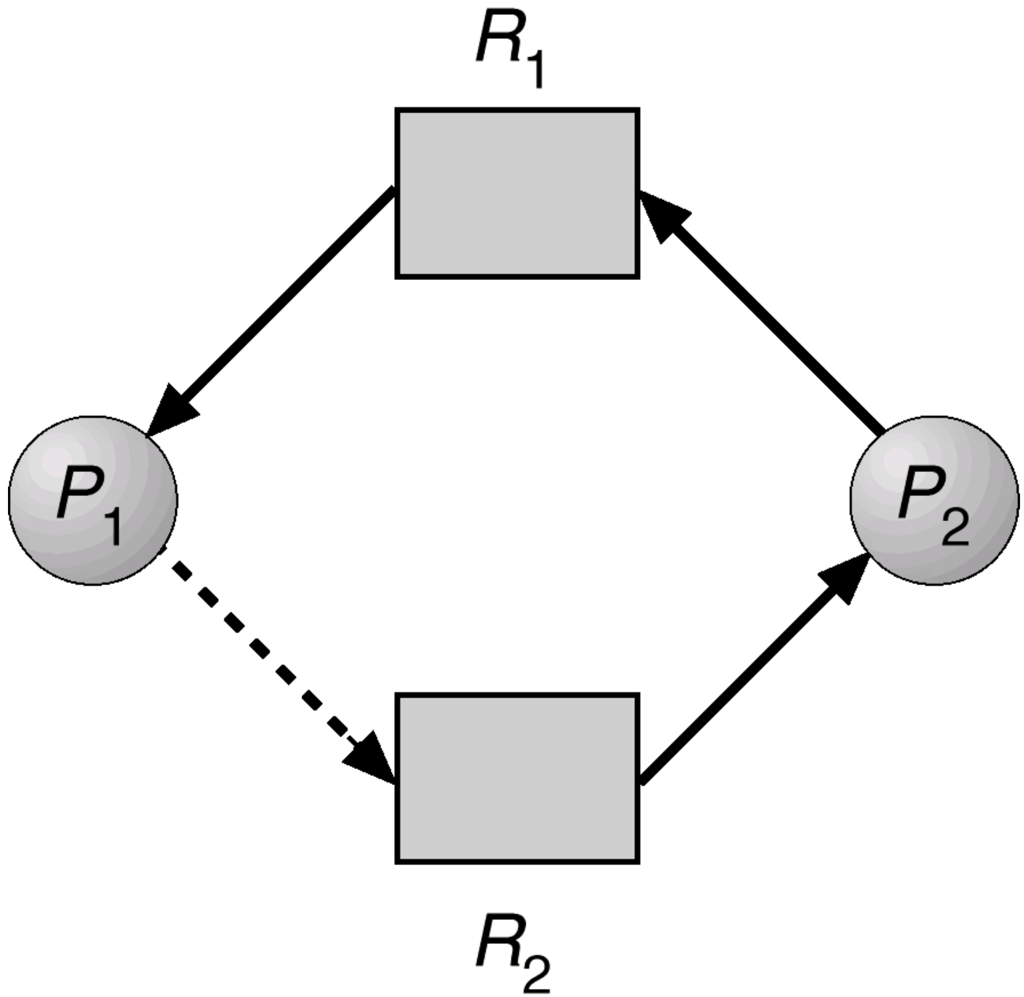
Silberschatz and Galvin©1999

Resource-Allocation Graph For Deadlock Avoidance



Silberschatz and Galvin©1999

Unsafe State In A Resource-Allocation Graph



Silberschatz and Galvin©1999

Banker’s Algorithm

- Multiple instances.
- Each process must a priori claim maximum 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.

Silberschatz and Galvin©1999

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.

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j].$$

Silberschatz and Galvin©1999

Safety Algorithm

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

   Work := *Available*

   Finish [*i*] = *false* for *i* - 1,3, …, *n.*

2. Find and *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.

Silberschatz and Galvin©1999

Resource-Request Algorithm for Process P_i

Request$_i$ = 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:

$$Available := Available = Request_i;$$

$$Allocation_i := Allocation_i + Request_i;$$

$$Need_i := Need_i - Request_{i;;}$$

- *If safe $\Rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored*

Silberschatz and Galvin©1999

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5instances, 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	

Silberschatz and Galvin©1999

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.

Silberschatz and Galvin©1999

Example (Cont.): P_1 request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ *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 $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Silberschatz and Galvin©1999