Operating System

Tutorial 10

Deadlock Avoidance

(Dijkstra/Banker's Algorithm)

Deadlock

- A set of blocked processes each holding some resources and waiting to acquire some resources held by another process in the set.
- Example (hardware resources):
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example (“software” resource:):
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	***wait(B)***
wait (B);	***wait(A)***
signal(B);	***signal(A)***
signal(A);	***signal(B)***

The (4) Conditions for Deadlock

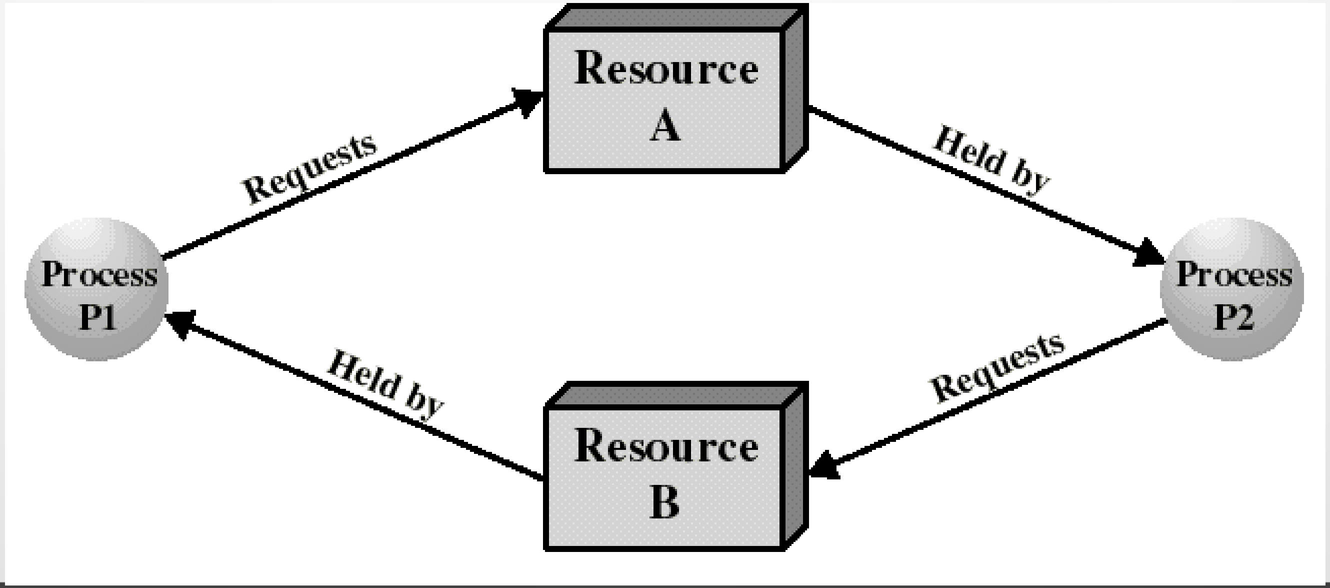
There are three policy conditions that must hold for a deadlock to be possible (the "necessary conditions"):

1. Mutual Exclusion
   - only one process at a time may use a resource
2. Hold-and-Wait
   - a process may hold allocated resources while awaiting assignment of others
3. No Preemption
   - a resource can be released only voluntarily by the process holding it, after that process has completed its task

- ..and a fourth condition which must actually arise to make a deadlock happen

The Conditions for Deadlock

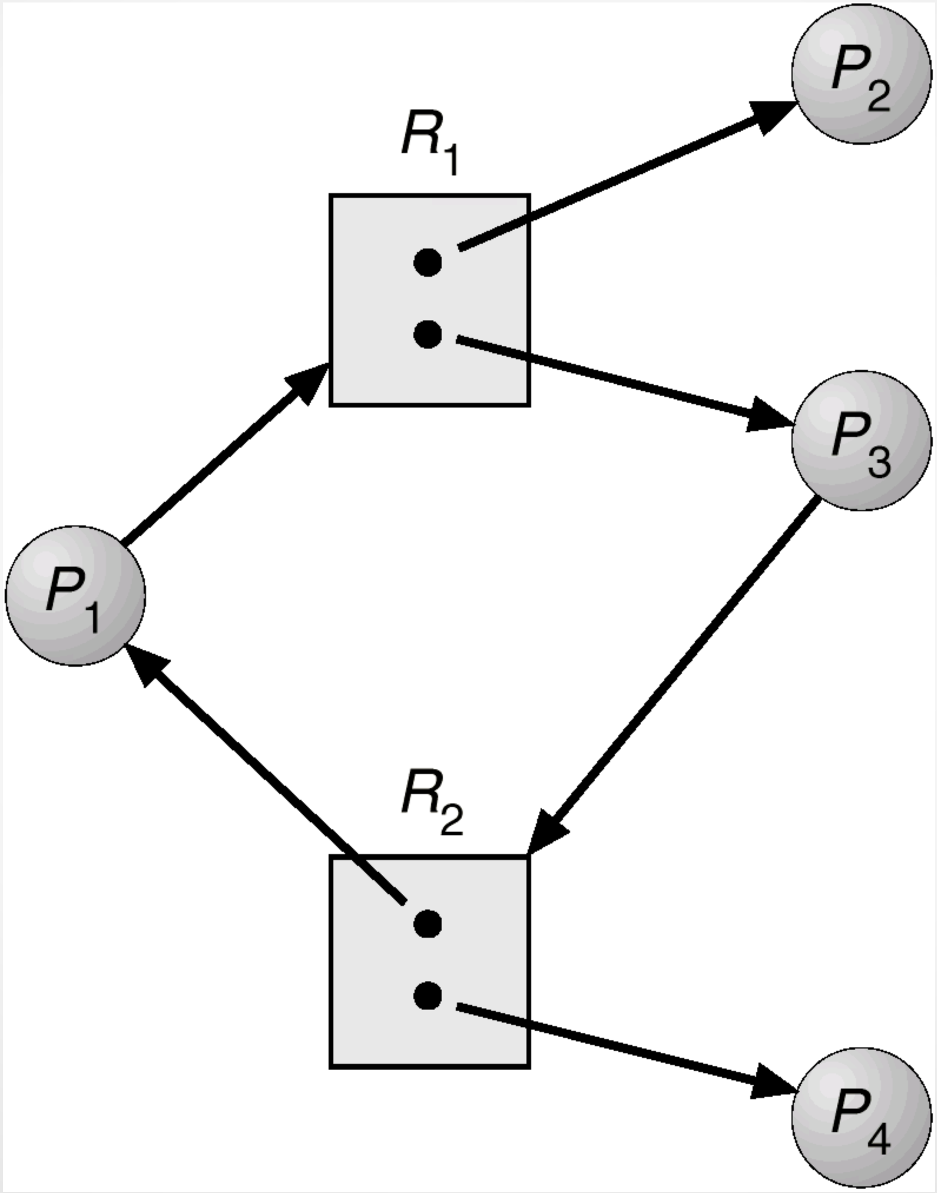
- In the presence of these necessary conditions,one more condition must arise for deadlock to actually occur:
- 4. Circular Wait
  - a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



The Conditions for Deadlock

- Deadlock occurs if and only if the circular wait condition is unresolvable
- The circular wait condition is unresolvable if the first 3 policy conditions hold
- So the 4 conditions taken together constitute necessary and sufficient conditions for deadlock

Resource Request Allocation Graph:



Multiple instances of resources R_1 and R_2: P_2 and P_4 can complete, freeing up resources for

Approaches to Handling Deadlocks

- Deadlock Prevention
  - disallow 1 of the 3 necessary conditions of deadlock occurrence, or prevent the circular wait condition from happening
- Deadlock Avoidance
  - do not grant a resource request if this allocation might lead to deadlock
- Deadlock Detection and Recovery
  - grant resource requests when possible, but periodically check for the presence of deadlock and then take action to recover from it

Deadlock Avoidance

- Two approaches:
  - do not start a process if its total demand might lead to deadlock: ("Process Initiation Denial"), or
  - do not grant an incremental resource request if this allocation could lead to deadlock: ("Resource Allocation Denial")
- In both cases: maximum requirements of each resource must be stated in advance

Resource Allocation Denial

- A Better Approach:
  - Grant incremental resource requests if we can prove that this leaves the system in a state in which deadlock cannot occur.
  - Based on the concept of a “safe state”.
- Banker’s Algorithm: (developed by Edsger Dijkstra)
  - Tentatively grant each resource request
  - Analyze resulting system state to see if it is “safe”.
  - If safe, grant the request
  - if unsafe refuse the request (undo the tentative grant)
  - block the requesting process until it is safe to grant it.

Data Structures for the Banker's Algorithm

Let n = number of processes,

m = number of resource types

- Available: Vector of length m. If Available $[j] = k$, there are k instances of resource type R_j currently available
- *Max*: n x m matrix. If *Max* $[i,j] = k$, then process P_i will request at most k instances of resource type R_j.
- *Alloc*: n x m matrix. If Alloc$[i,j] = k$ then P_i is currently allocated (i.e. holding) 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.

$$Need\ [i,j] = Max[i,j] – Alloc\ [i,j].$$

Safety Algorithm

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

   Work := *Available*

   Finish [i] == *false* for *i* = 1,2, …, *n*.

2. Find an *i* such that both:

   Finish [i] == *false*

   $Need_i \leq Work$

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

3. $Work := Work + Allocation_i$

   (Resources freed when process completes!)

   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

$Request_i$ = request vector for P_i .

$Request_i [j] = k$ means process P_i wants k instances of resource type R_j.

1. **If $Request_i \le Need_i$ go to step 2. Otherwise, error (process exceeded its maximum claim).**
2. **If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, (resources not available).**
3. **"Allocate" requested resources to P_i as follows:**

    $Available := Available - Request_i$

    $Alloc_i := Alloc_i + Request_i$

    $Need_i := Need_i – Request_i$

    If safe ⇒ the resources are allocated to P_i.

    If unsafe ⇒ restore the old resource-

Example of Banker's Algorithm

5 processes P_0 through P_4

3 resource types *A* (10 units), *B* (5 units), and *C* (7 units).

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)

Need = 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.

Now P_1 requests (1,0,2)

Check that Request ≤ Available

(that is, $(1,0,2) \le (3,3,2)$) ⇒ *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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	