Chapter - 4

Deadlocks

Important Questions

1.What do you mean by Deadlocks ?

- A process request for some resources. If the resources are not available at that time , the process enters a waiting state . The resources was held by other processes .The waiting process may never able to get the resource.

This situation is called deadlock.

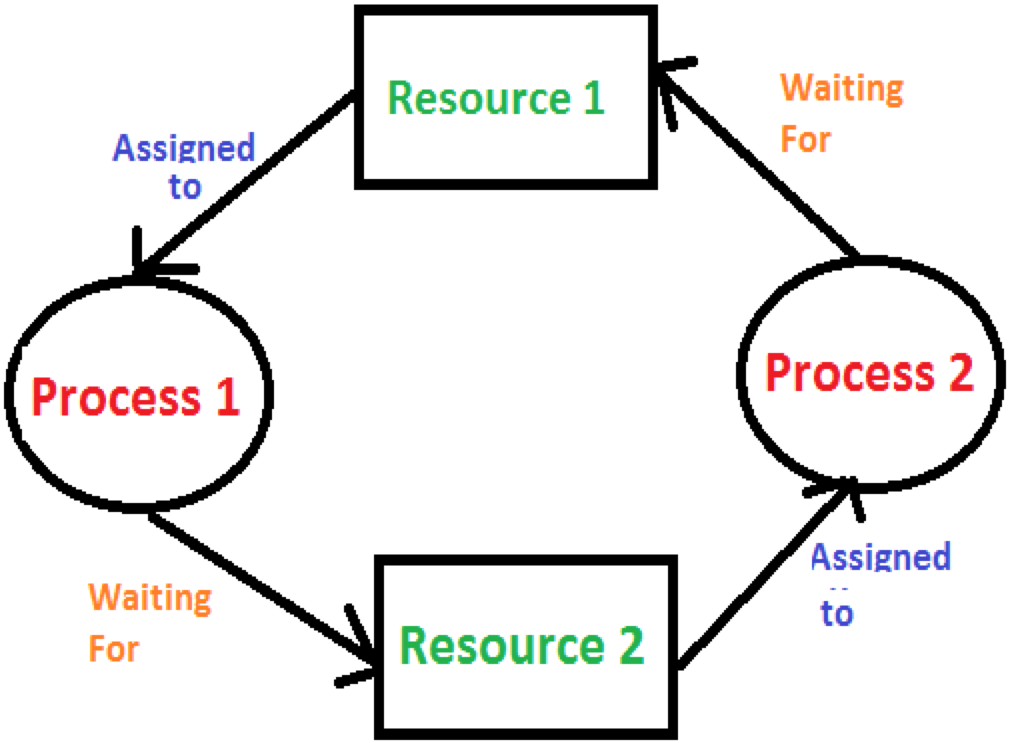
2 . What are the necessary conditions for deadlocks?

- **Mutual exclusion**: only one process at a time can use a resource. If another process requests the same resource, the requesting process must wait until the resource is released.
- **Hold and wait**: Processes currently holding resources granted earlier , can request for new resources , that are currently held by other.

- **No preemption**: a resource can be released by the process holding it only after that process has competed its task.
- **Circular wait**: The circular chain of two or more processes must exist such that each of them is waiting for a resource held by next member.

There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Circular wait

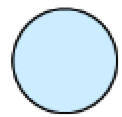


3. Explain the Resource allocation graph in detail.

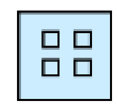
- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource allocation graph

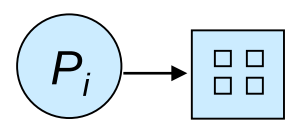
- Process



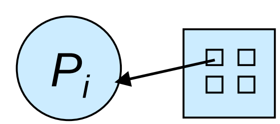
- Resource Type with 4 instances



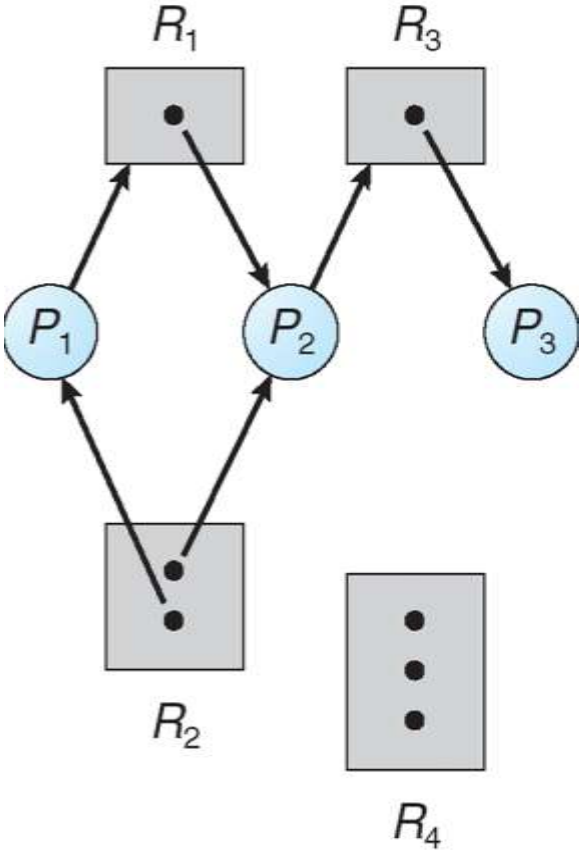
- P_i requests instance of R_j



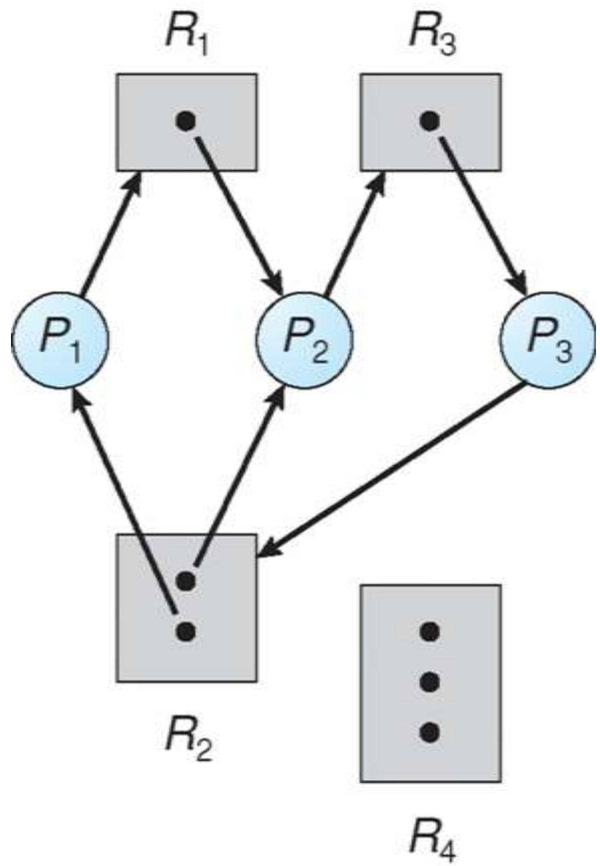
- P_i is holding an instance of R_j



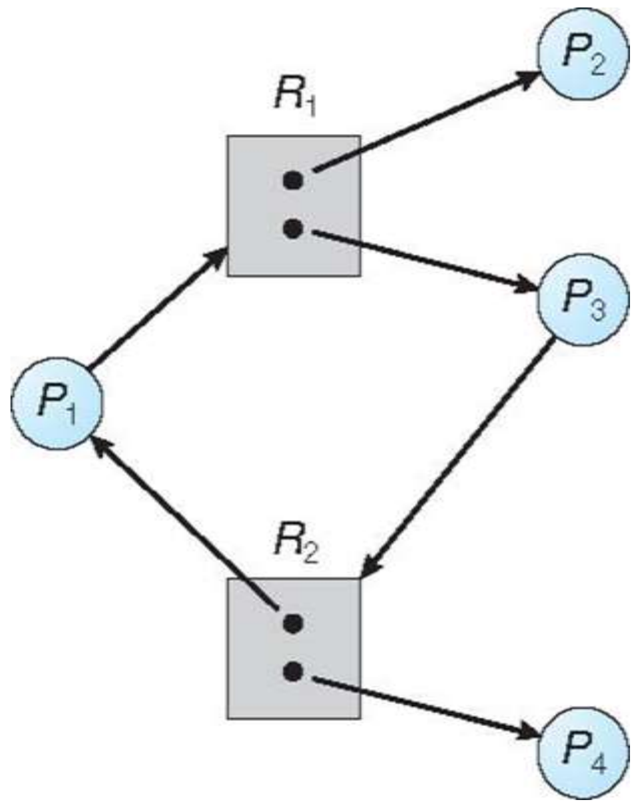
Example for Resource allocation graph



Resource allocation graph with deadlock



Graph With A Cycle



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

4. Explain the concept of Deadlock prevention in detail

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

Mutual Exclusion

Not always possible to prevent deadlock by preventing mutual exclusion (making all resources shareable) as certain resources are cannot be shared safely.

Hold and Wait

A process can get all required resources before it start execution. This will avoid deadlock, but will result in reduced throughputs as resources are held by processes even when they are not needed. They could have been used by other processes during this time.

Second approach is to request for a resource only when it is not holding any other resource.

- **No preemption**

We will see two approaches here. If a process request for a resource which is held by another process, then the resource may be preempted from the other process. In the second approach, if a process request for a resource which are not readily available, all other resources that it holds are preempted.

Circular wait

- To avoid circular wait, resources may be ordered and we can ensure that each process can request resources only in an increasing order of these numbers. The algorithm may itself increase complexity and may also lead to poor resource utilization.

5. Explain Banker's Algorithm in detail

- Multiple instances
- 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

- Banker’s algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that is applicable to resource allocation systems with multiple instances of each resource type.
- The Bankers algorithm is run by operating system whenever a process requests resources. The algorithm must determine whether allocation of these resources will put the system in a safe state. If true , the resources will be allocated. Otherwise ,the process must wait until some other process runs to completion and releases enough resources

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

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) $\boldsymbol{Need_i \leq Work}$
   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   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

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i} \leq \boldsymbol{Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i} \leq \boldsymbol{Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   $\boldsymbol{Available = Available - Request_i;}$
   $\boldsymbol{Allocation_i = Allocation_i + Request_i;}$
   $\boldsymbol{Need_i = Need_i - Request_i;}$

   - If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
   - If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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* (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	

- 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

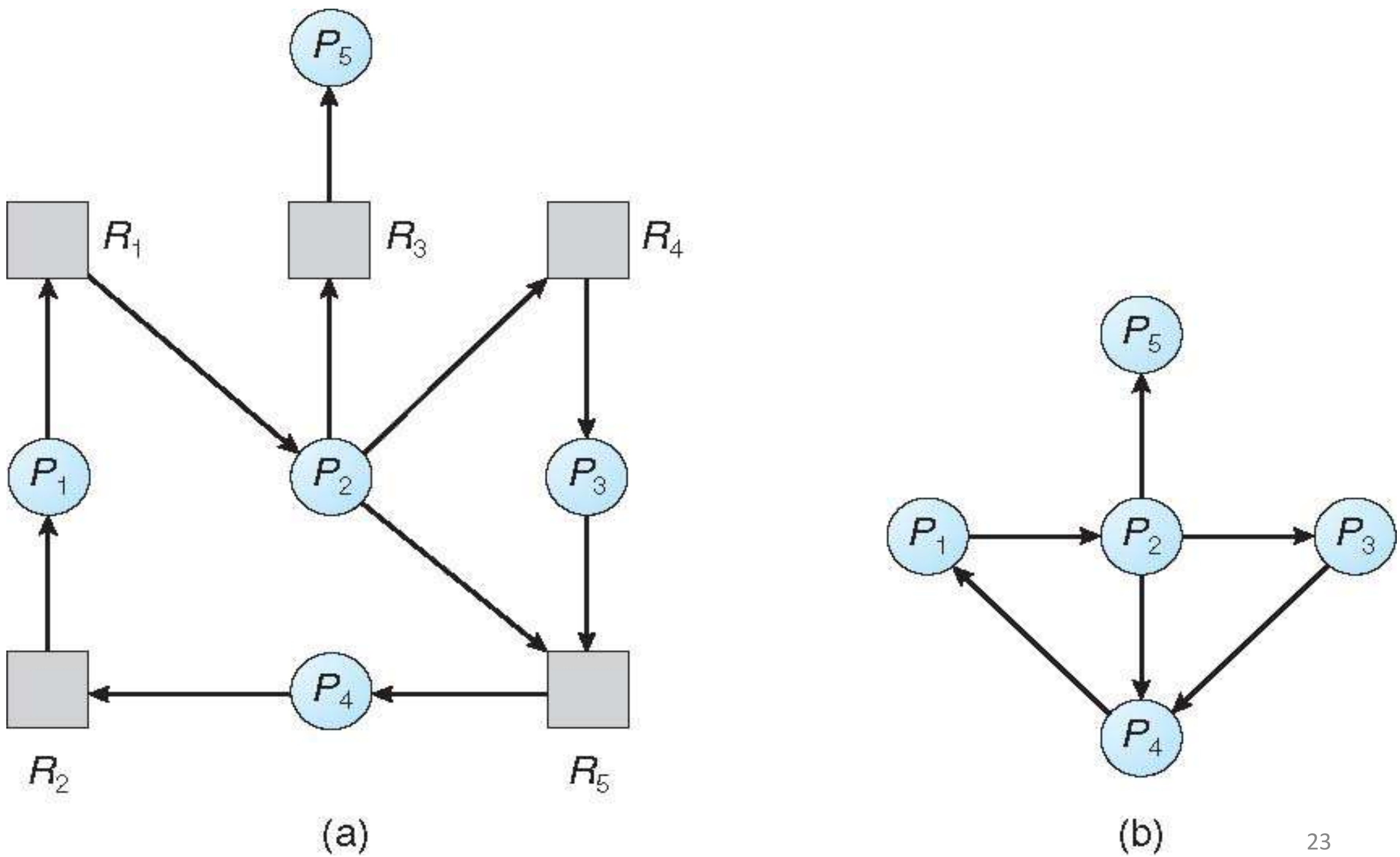
6. Explain the Deadlock detection algorithm for single and multiple instance type

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. If there is a cycle, there exists a deadlock

Resource-Allocation Graph and 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* = 1,2, …, *n***, if $\boldsymbol{Allocation_i} \neq \mathbf{0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i} \leq \boldsymbol{Work}$

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

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

4. If ***Finish[i] == false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state.

7. Explain the concept of Deadlock recovery

- Process termination
- Resource pre-emption
- Check point / roll back mechanism

Process termination

- Abort all deadlocked process
- Successively abort each deadlocked process until the deadlock no longer exists.

Resource pre-emption

Roll back

A process that has a resource pre-empted from it must be roll back to the point to its acquiring of that resource.

Total roll back – Abort the process and restart it.

Check point

- Keep checkpointing periodically
- When a deadlock is detected , see which resource is needed
- Take away the resource from the process currently having it
- Restart the process from the checkpointed state.

8.What is safe state?

- **Safe State**
  - state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.