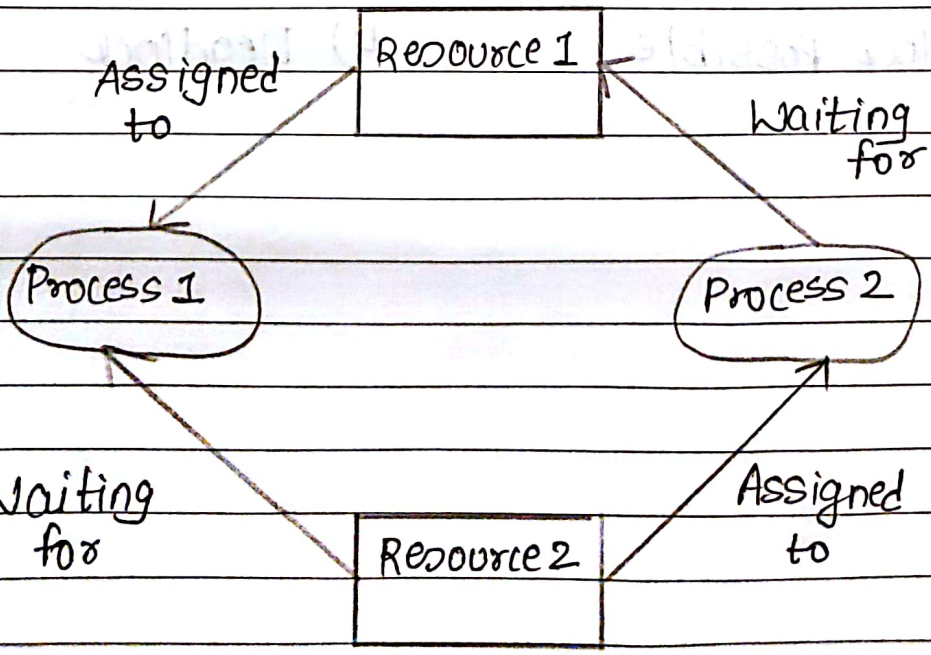
DEADLOCKS

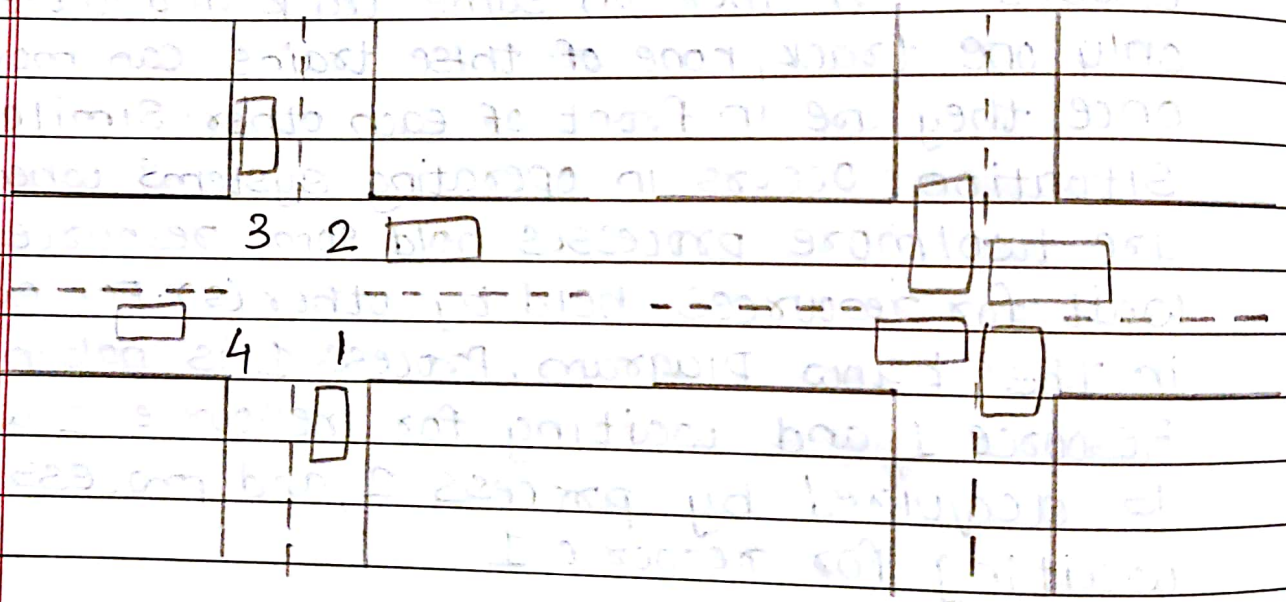
Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other resource process.

Consider an example when two trains are coming toward each other on same track and there is only one track, none of thise trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two/more processes hold some resource and wait for resources held by other(s). For example in the below Diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1



- A process is deadlocked if it is waiting for an event that will never occur. Typically, more than the process will be involved in a deadlock.
- A process is indefinitely postponed if it is delayed repeatedly over a long period of time while the attention of the system is given to other process. i.e the processed but never gets the CPU.

Illustration of Deadlock



a) Deadlock Possible

b) Deadlock

* SYSTEM MODEL

- For the purpose of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers CPUs, open files, tapes drives, CD-ROMs etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category If this is not the case (i.e if there is difference between the resources within a category), then that category needs to be further divided into categories. For example, "printers" may be seperated into laser printers and color inkjet printers
- In a normal operation a process must ~~be~~ request a resource before using it, and release it when it is done, in the foll sequence:

Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For eg:- the system calls open(), malloc(), new() and request()

Use :- The process uses the resource, eg :- prints to the print or reads from the file

Release: The process relinquishes the resource, so that it becomes available for other resources.
Eg :- close(), free(), delete() and release()

Resources can be of several types

Reusable- This are the resources that we can reuse again and again for example CPU cycles, memory space, I/O devices and files
Acquire → Use → Release

Consumable: Produced by a process, needed by a process - Eg: messages, buffers of information, interrupts
Create → acquire → use
Resource cease to exist after it has been used.

DEADLOCK PREVENTION

* Deadlock can arise if four conditions hold simultaneously (Neccessary conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at time)

Hold and wait: A process is holding atleast one resource and waiting for resources

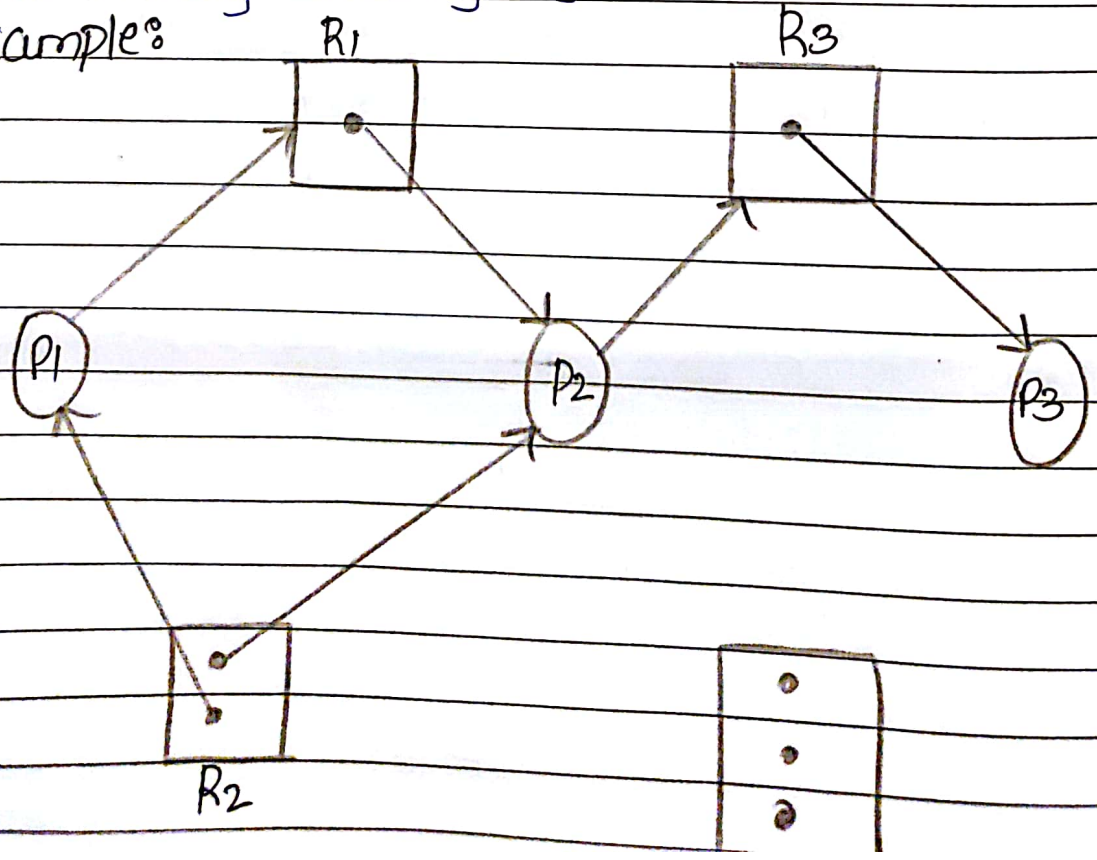
No Preemption: A resource cannot be taken from a process unless the process release the resource

Circular Wait: A set of processes are waiting for each other in circular form

* RESOURCE ALLOCATION GRAPH

- In some cases deadlocks can be understood more clearly through the use of Resource-Allocation Graphs, having the following properties:
  - A set of resources categories, {R1, R2 --- RN} which appear as square nodes on the graph. Dots inside the resource indicate specific instances of the resource
  - A set of processes, {P1, P2, P3, ---- PN}
  - Request Edges: A set of directed arcs from Pi to Rj, indicating that Pi has requested Rj, and is currently waiting for that resource to become available.
  - Assignment Edges: A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding Rj

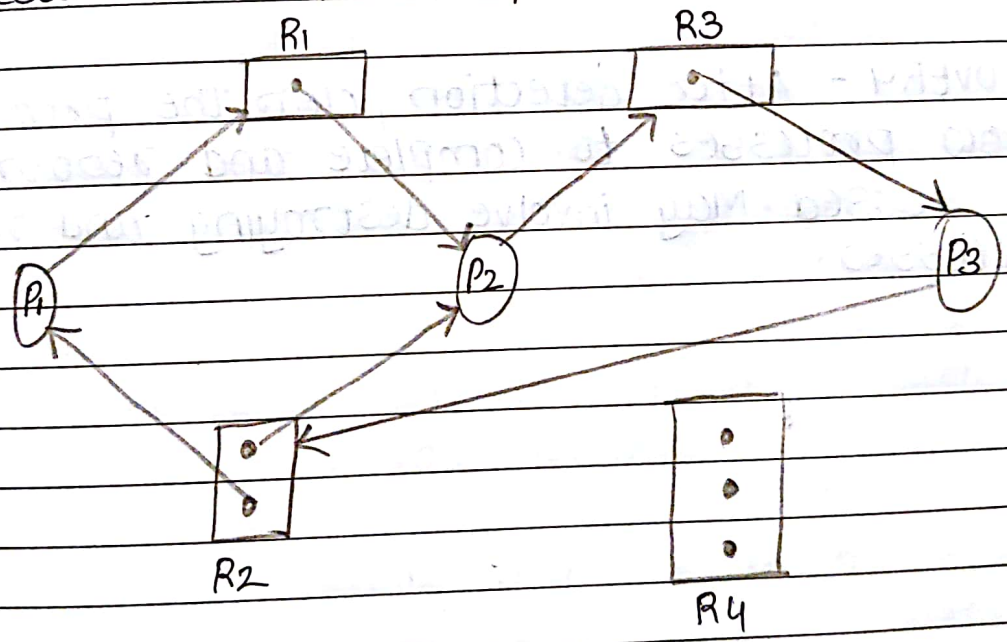
Example:



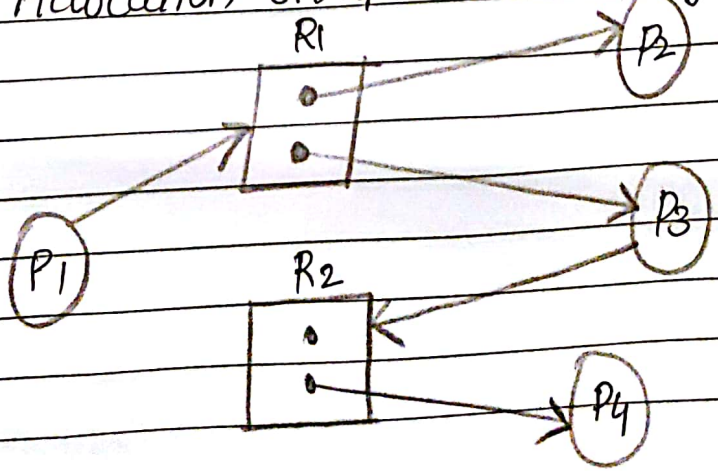
- If a resource-allocation graph contains no cycles, then the system is not deadlocked.
- If a resource allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.
- If the resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not gurantee one.

Example:

1] Resource Allocation Graph with Deadlock



Resource Allocation Graph with a cycle but no deadlock



METHODS FOR HANDLING DEADLOCKS

Deadlock prevention ~~or Avoidance~~ - Do not allow systems to get into a deadlock state

AVOIDANCE - Impose less stringent conditions than for prevention, allowing the possibility of deadlock but sidestepping as it occurs

DETECTION - Allow possibility of deadlock, determine if deadlock has occured and which processes and resources are involved

RECOVERY - After detection, clear the problem, allow processes to complete and resources to be reused. May involve destroying and restarting processes.

* DEADLOCK PREVENTION

The strategy of deadlock prevention is, simply put, to design a system in such a way that the possibility of deadlock is excluded. We can view deadlock prevention methods as falling into two classes.
An indirect method of deadlock prevention is to prevent occurence of one of the three necessary conditions (Mutual Exclusion, Hold and Wait, NO preemption)
A direct method of deadlock prevention is to ~~protect~~ prevent the occurence of circular Wait.

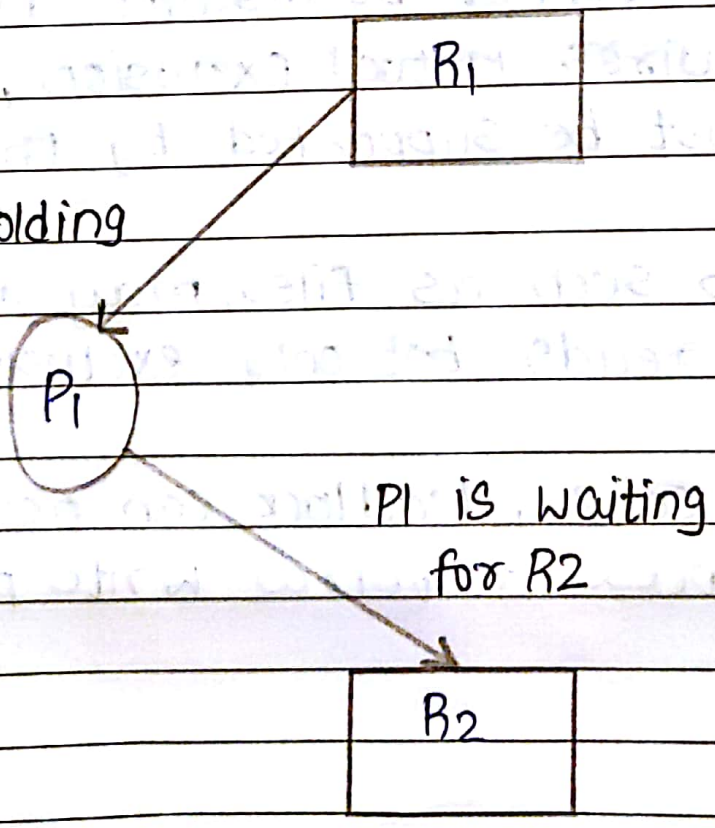
1] Mutual Exclusion

Mutual Exclusion cannot be disallowed. If access to a resource requires Mutual Exclusion, then mutual Exclusion must be supported by the operating System.
Some resources such as files, may allow multiple accesses for reads but only exclusive access for writes.
Even in this case, deadlock can occur if more than one process required write permission.

HOLD AND WAIT

Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. For example, if a resource requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution

Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation



3] NO Preemption

- This condition can be prevented in several ways.
- First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with additional resources
- Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.
- This latter scheme would prevent deadlock only if no two processes possessed the same priority.
- This process is practical only when applied to resources whose state can be easily saved and restored later, as it case with a processor

4] CIRCULAR WAIT

The circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resource of R, then it may subsequently request only those resources of type following R in the ordering

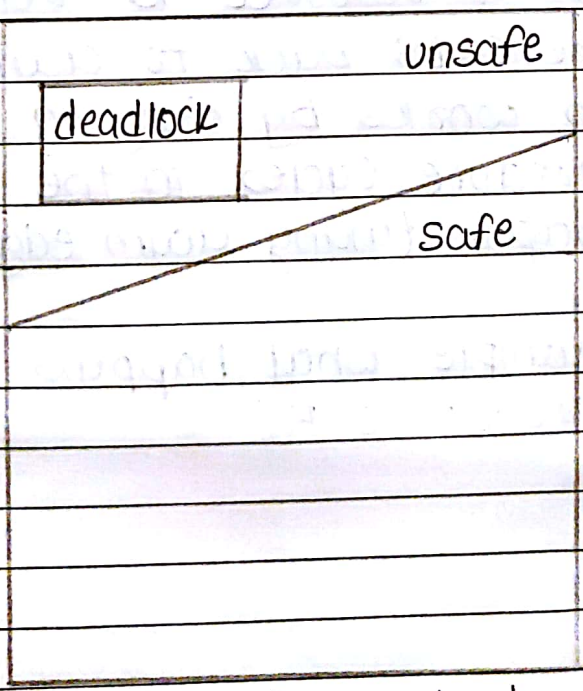
To see that this strategy works, let us associate an index with each resource type. Then resource Ri preceeds Rj in the ordering if i<j. Now suppose that two processes, A and B are deadlocked because A has acquired Ri and requested Rj, and B has acquired Rj and requested Ri. This condition is impossible because it implies i<j and j<i.

DEADLOCK AVOIDANCE

- The general idea behind deadlock avoidance is to prevent dead locks from ever happening, by preventing at leaot one of the aforementioned conditions
- This requires more information about each process AND tends to lead to low device utilization (i.e it is a conservative approach)
- In some algorithms the scheduler only needs to know the maximum number of each ~~process~~ th resource that a process might potentiaay use. In more complex algorithm the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order
- When a scheduler see that starting a process or granting resource requests may lead to ~~&~~ future deadlocks, then that process is not just started or the request is not granted
- A resource allocation state is defined by the number of processes available and allocated resources, and the maximum requirements of all processes in the system,

SAFE STATE

- A state is a safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock.
- More formally, a state is safe if there exists a safe sequence of processes {P1, P2, ----, PN} such that all of resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j< i (i.e if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they freed up)
- If a safe sequence does not exists, then the system is in unsafe state, which MAY lead to deadlock

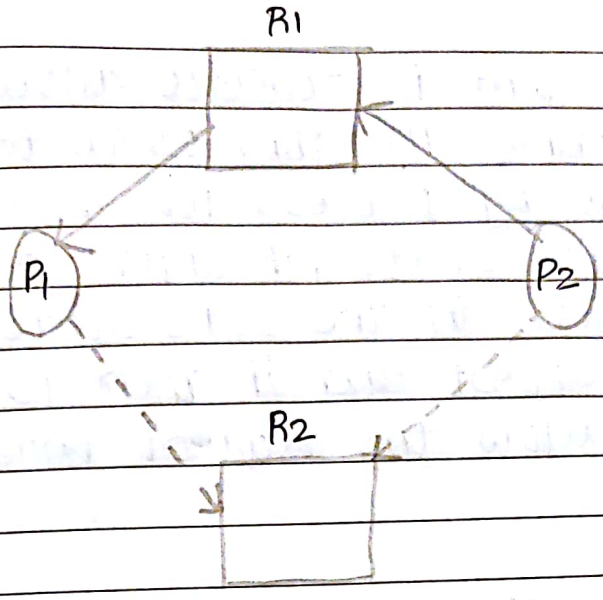


Safe, Unsafe, and deadlock state spaces

* RESOURCE-ALLOCATION GRAPH ALGORITHM

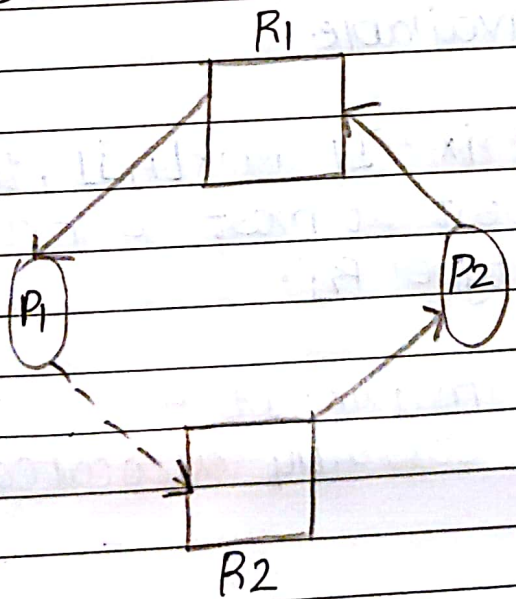
- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim-edges, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular resources before that process is allowed to request any resources.
- When a process may request, the claim edge $P_i \rightarrow R_j$ is converted to ~~the~~ a request edge. Similarly when a resource is released, the assignment reverts back to claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process R2 requests resource R2.

RESOURCE ALLOCATION GRAPH FOR DEADLOCK AVOIDANCE



The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted

An unsafe state in Resource Allocation Graph



* BANKER'S ALGORITHM

Banker's Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by process for resource, it checks for safe state, if after granting request system remains in the safe state and it allows the request and if their is no safe state don't allow the request made by the process

Data structure used for Banker's Algorithm

Let n = no. of process

m = no. of resource types

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

Max: n x m matrix. If Max [i, j], then process Pi may request at most k instances of resource type Rj.

Allocation: n x m matrix. If Allocation [i, j] = k then Pi is currently allocated k instances of Rj

Need: n x m matrix. If Need [i, j] = k, then Pi may need k more instances of Rj to complete the task

Need [i, j] = Max [i, j] - Allocation [i, j]

SAFTEY ALGORITHM

1. Let Work and Finish be vectors of length m and n, respectively. Intialize:
   Work = Available
   Finish[i] = false for i = 0, 1, 2, - - - , n-1

2. Find an i such that both:
   a) Finish [i] = false
   b) Need i ≤ Work
   If no such i exists, go to step 4

3. Work = Work + Allocation i
   Finish[i] = true
   go to ~~set~~ 2 step 2

4. If finish[i] = = true ∀ i, then the system is in safe state

Request will only be granted under below condition:

- If request made by process is less than equal to max need to that process.
- If request made by process is less than equal to freely available resource in the system.

Second Algo also

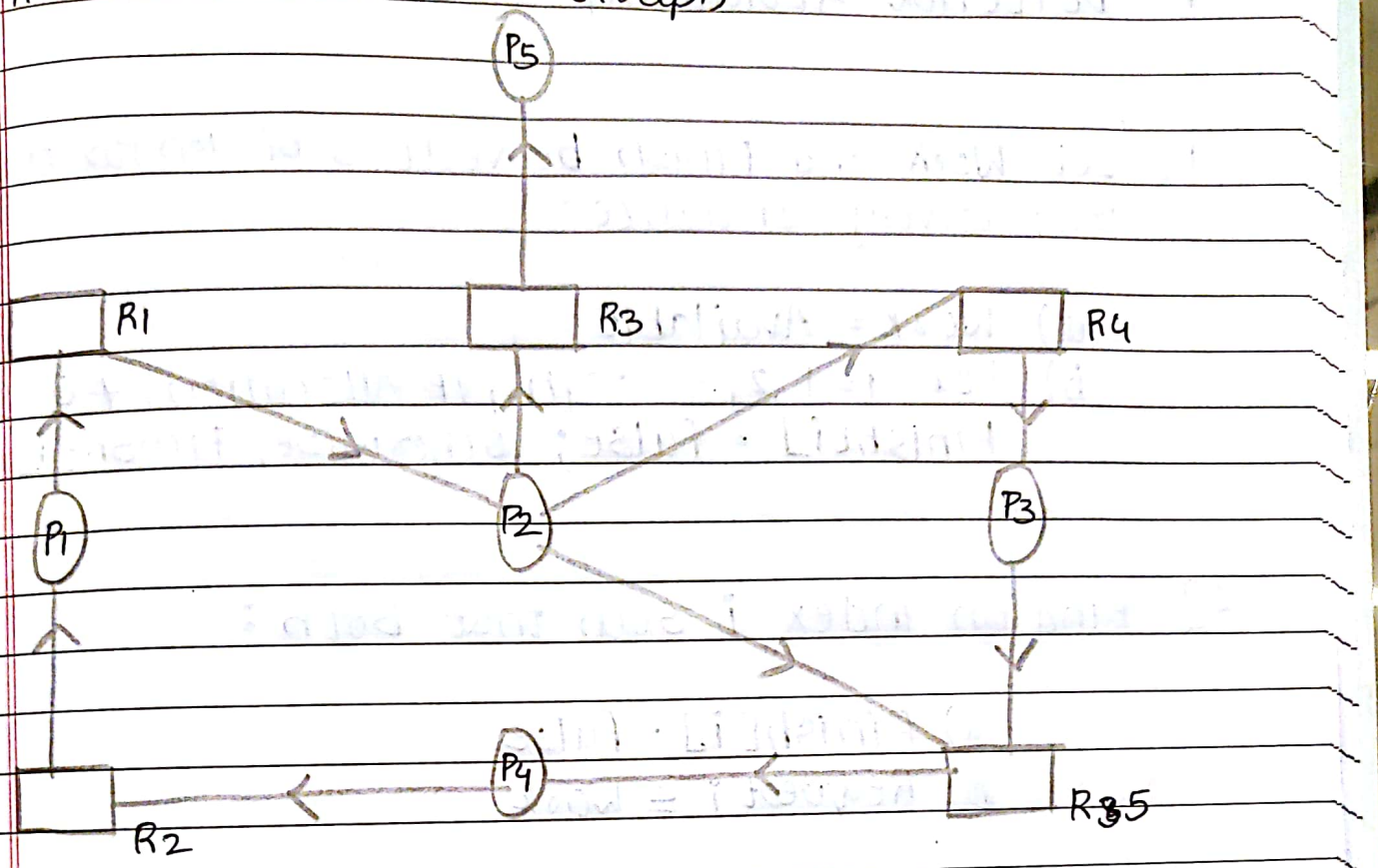
* DEADLOCK DETECTION

- If deadlocks are not avoided, then another approach is to detect when they have occured and recover somehow
- In addition to the performance hit of constantly checking for deadlock, a policy/algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted to have their resource preempted.

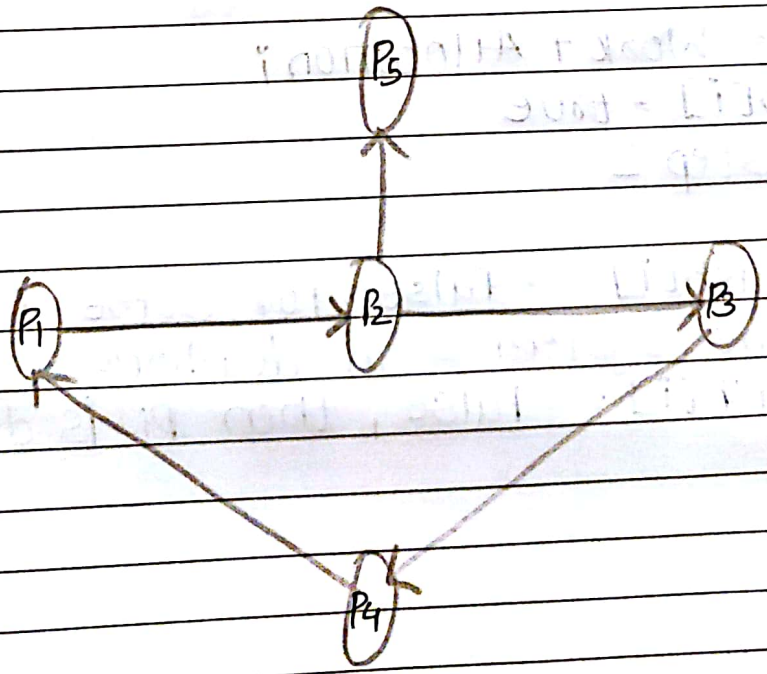
SINGLE INSTANCE OF EACH RESOURCE TYPE

- If each resource category has a single instance, then we can use a variation of the resource-allocation known as wait-for graph.
- A wait-for graph can be constructed from a resource allocation graph by eliminating the resources and collapsing the associated edges, as shown in fig below
- An arc from Pi to Pj in a wait-for-graph indicates that process Pi is waiting for a resource that process Pj is currently holding
- As before, cycles in deadlock graph indicates deadlocks.

Resource Allocation Graph



Corresponding wait-for graph



* DETECTION ALGORITHM

1.] Let Work and Finish be vectors of length m and n respectively Intialize:

a) Work = Available
b) For i = 1, 2, ----, n, 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 Pi is deadlocked

RECOVERY FROM DEADLOCK

- There are 3 basic approaches to recovery from deadlock.
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one/more processes involved in the deadlock
  3. Preempt Resources

Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock This definitely solves the deadlock, but at the expense of terminating more processes that would be absolutely neccessary.
  - Terminate process one by one until deadlock is broken. This is more conservative, but require deadlock detection after each step.

Resource Preemption

Selecting a Victim: Deciding which resources to preempt from which processes involves many of the same decision criteria. ~~outlin~~

Rollback: Ideally one would like to roll back a preempted process to a safe state prior to the point at which the resource was originally allocated to the process. Unfortunately it can be

difficult or impossible to determine what such a safe state is, and so the only safe roll back is to rollback to the beginning.

3.] Starvation: How do you gurantee that a process won't starve because its resources are constantly being preempted? one option would be to use a priority system, and increase the priority of a process every time its resources gets preempted. Eventually it would get a high enough priority that it won't get preempted any more.

Dining Philosophy - 2 pseudo code
- Problems
- Solution

monitors - Struct
P/C Solotion