Module-3

Deadlocks

Contents :

Deadlocks----System Model

- WKT, every process needs some resources like main memory sapce, CPU cycles, files & I/O devices such as printers, keyboard, mouse, peripheral devices, etc
- Every system consists of finite number of resources. These resources are distributed among the many competing processes in a multiprogramming environment.
- A system may have multiple resources of the same instance. For example, if a system has 4 CPU, then The resource type CPU has 4 instances. similarly, resource type "printer” may have 3 instances.
- WKT, a running process may require files or I/O devices to complete its task.
- So, **when a process needs any resource, it should make request for that & after using, it should release the resource.**

Deadlocks----System Model

- Number of resources requested by a process should not exceed the total number resource of that type. For example, a process should not request 4 printers when there only 3 printers.
- Any process requesting for the resource should follow the sequence

1) **Request**

if a process needs any resource, then it should request for that. If the requested resource is available, immediately it will be allocated to that process.

suppose, if the requested resource is not free at that time, then the process should wait for the resource to become free

2) **Use**

Once, the resource is allocated to a process, then process can operate (use) that resource.

Deadlocks----System Model

3) **Release**

Once the process finishes its jobs then it should release the resources that it owns.

These are done through system calls like, request(), release(), open(), close() (for files), allocate() & free() (for memory space).

- In order to check the availability of the resources, OS maintains a table called **system table.**
- A system table consists of following information
  - How many resources of each type are there.
  - How many are allocated
  - How many are free
  - which are the process which requests for the particular resource
  - to which process, resources are allocated
  - if all the instances of a particular resource are already allocated, if another process request for the same, then that process will be added to the waiting queue of that resource.

Deadlocks----System Model

- Situation in which 2 or more processes / threads get into a state whereby each is controlling a resource that the other needs, is called **deadlock situation.**
- for example, suppose 2 pirates have each obtained half of a treasure map. Each pirate needs the other half of map to obtain the treasure, but neither will give up his half of the map. This is deadlock.
- In software, deadlocks occur because one process holds a resource, file A, while requesting another, file B. At the same time, another process holds the 2nd resource, file B, while requesting the first one, file A. In this situation, both the processes will never get their desired resources to complete their tasks & remains in the deadlock state for ever.
- Multithreaded programs are god candidates for deadlock because multiple threads can compete for shared resources

Deadlock characterization

- In a deadlock, processes never finish executing & system resources are tied up, preventing other jobs from starting.
- **Necessary conditions**

Deadlock situation occurs only when the following 4 events happens simultaneously in a system

1) **Mutual exclusion**

only one process should use the resource at a time. (always, resource are non-sharable).

if another process requests that resource, requesting process must be delayed until the resource has been released.

2) **Hold and Wait**

a process must be using or holding at least one resource & waiting to acquire additional resources that are currently being held by other processes

Deadlock characterization

3) No pre-emption

resources cannot be pre-empted, i.e., a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4) Circular wait

A set of waiting processes { P0,P1,P2,……..Pn} must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ……………Pn-1 is waiting for resource held by Pn and Pn is waiting for resource held by P0.

Resource Allocation Graph

- Deadlocks can be described more efficiently through a directed graph called system **resource allocation graph**
- This graph consists of set of vertices V and a set of edges E
- Set of vertices/nodes are classified into 2 types
  1) Set of active processes, P={ P1, P2, P3, ……..Pn }
  2) Set of all resources, R= _ R1, R2, R3, ……….Rm }
- If any process Pi requests any resource Ri, then it is represented as Pi→Ri, that means, when Pi requests one instance of Ri then it is represented as Pi→Ri and then Pi goes to waiting state to get one instance of Ri.
- If one instance of any resource Ri is allocated to any process Pi, then it is represented as Ri→Pi
- A directed edge Pi→Ri is called **request edge** and a directed edge Ri→Pi is called **assignment edge**
- In this graph, processes are represented as circle & resources are represented as rectangle with some dots.
- This dots represents number of instances of that particular resource

Resource Allocation Graph

The sets *P,* R and *E:*

P = {P1, P2, P3}   R= {R1, R2, R3, R4}

E = {Pl→ Rl, P2→*R3.* Rl→*P2,* R2→P2, R2→Pl, R3→P3}

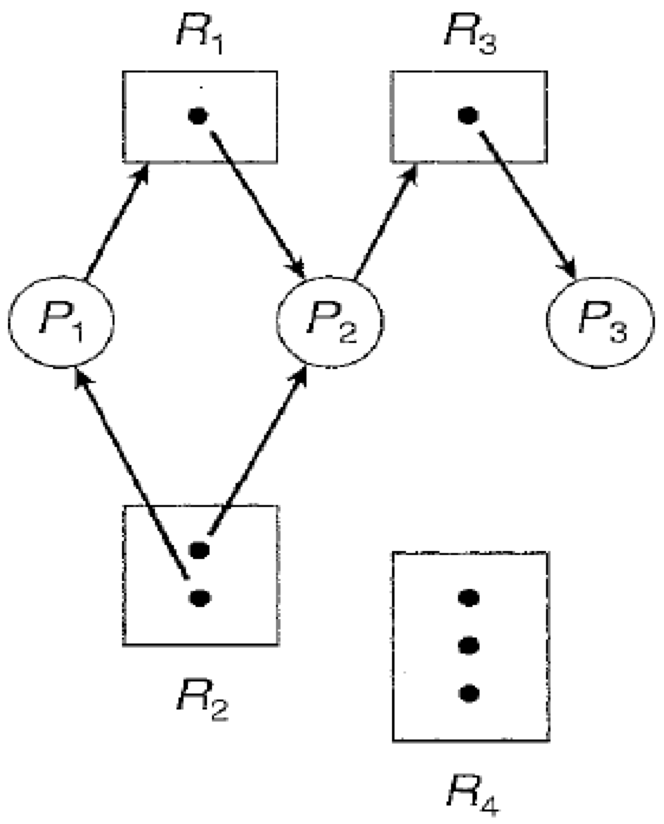
Resource instances:

One instance of resource type R1

Two instances of resource type R2

One instance of resource type R3

Three instances of resource type R4



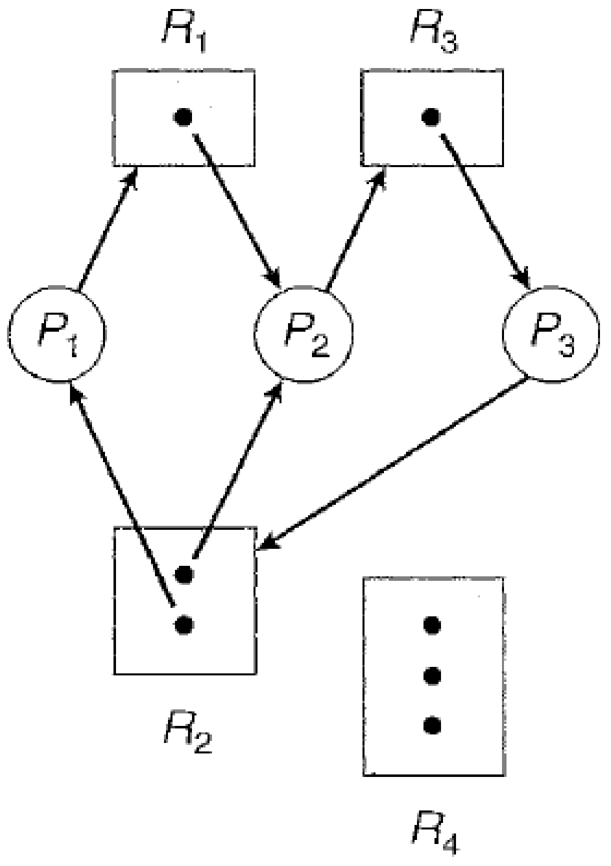
- By observing resource allocation graph, we can say processes are in Deadlock situation or not.
- If the graph contains **no cycles** then we can say there is **no deadlock**
- If the graph contains any cycle, then deadlock may exist (but it is not sure)
- If there is **only one instance of all the resources** & the graph is **having cycle**, then we can say surely that, **there is a deadlock**

Resource Allocation Graph

- Consider again example 1, with P3 requesting an instance of P2.
- *Now, its resource allocation graph is as shown*
- In this graph, there are 2 cycles

1) P1→R1→P2→R3→P3→R2→P1
2) P2→R3→--P3→R2→P2

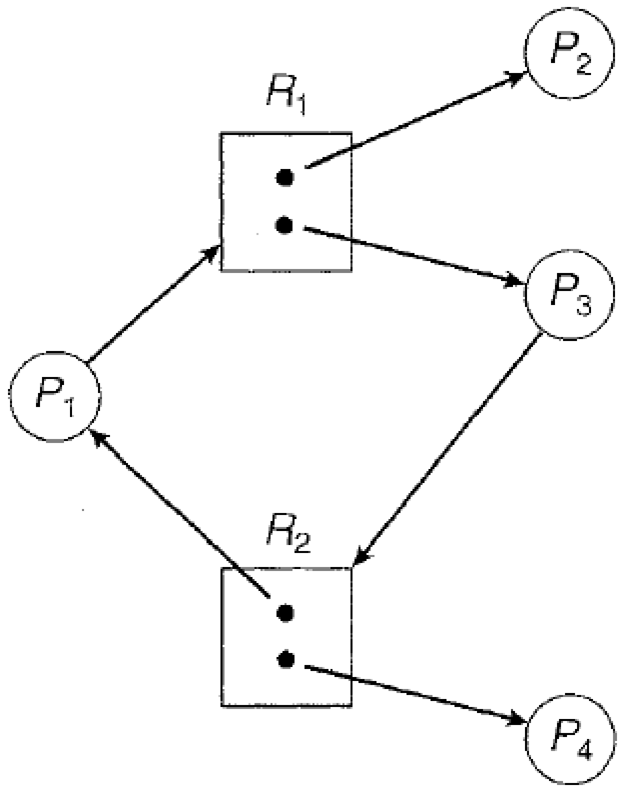
- 1st cycle indicates P1, P2 and P3 are deadlocked
- 2nd cycle indicates P2 and P3 are deadlocked
- Here deadlock occurs because no instances of any resource is free (all are allocated)
- Each process is holding one or more resource & requsting for another resource which is hold by another process



Resource Allocation Graph

- Now consider the resource allocation graph as shown
- Here, even though graph is having cycle, there is no deadlock
- SUMMARY

1) if resource allocation graph does not having any cycle, then it indicates there is no deadlock situation
2) if resource allocation graph is having cycle, deadlock may appear.
   There are 2 cases
   a) if there is a cycle & all the resource instances are still free then there is no deadlock
   b) if there is a cycle & all the instances of all the resources are already allocated (not free) then there is a deadlock situation.



Methods for handling deadlocks

- If deadlocks are not recovered, then system performance decreases, because more & more upcoming processes are keeps on requesting for the resources but all are allocated, no one is free.
- So eventually, system will stop functioning & will need to be restarted manually
- So, deadlocks have to be prevented or avoided, if occurred it has to be recovered
- To handle the deadlocks, some measures or methods has to be followed:

  1) some protocols has to be used to prevent or avoid deadlocks, ensuring that system will never enter the deadlock state

  2) if deadlock occurs, it should be detected & recovered in an efficient way

  3) if deadlock occurs, ignore the problem altogether & pretend that deadlock never occur in the system
- The 3rd solution is the one used by most of the Oss including windows and unix.

Deadlock prevention

- We know that, deadlock occurs only when all of the 4 following conditions are occurred:
  1) mutual exclusion
  2) Hold & wait
  3) No preemption
  4) Circular wait.
- Deadlocks can be prevented by avoiding any one of the above mentioned 4 events.

Mutual exclusion

- if all the resources are sharable, then there is no chance of deadlock.
- For example, Read only files are sharable resources. if several processes attempt to open a read-only file at the same time, they can be granted simultaneously. In this case, no process need to wait for sharable resources & hence no deadlock. But there are some resources which cannot be shared by several processes at the same time. For example, printer, scanner, etc.
- in general, it is not possible to prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Deadlock prevention

Hold and Wait

- There are some possibilities to avoid deadlock by avoiding this hold & wait Condition.
- A process is allowed to request for any resources only when it doesn't have (not holding) resources.
- Another way is, process is allowed to execute only when it is having all of its resources required to complete their task.
- Once process is having all of its resources, then it can be allowed to execute. In this case, there is no requesting for any other recourse & no waiting & hence no deadlock.
- So, it is the responsibility of the OS to allocate all the resources to a process but practically it is having some disadvantages like starvation & less utilization of resources.
- If all the resources are allocated to a process & it is taking long time to complete its task, other processes has to wait for long period of time which leads to starvation.

Deadlock prevention

No Pre-emption

- We can avoid deadlock by avoiding no-preemption.
- It can be done in 2 ways.

1) when a process request for any resource & it is already allocated, then this process has to wait until that requested resource becomes free.

If that process is having (holding) some resources, then those resources has to be preempted (released)

These released resources are added to the list of available resources so that other processes which are waiting for these can be benefited.

process which is preempted(released) all of its resources, that has to be restarted only when it can regain its old resources & as well as new ones that it is requesting

2) when a process requests for any resource, it will be get allocated if it is free.

if the requested resource is not free, then check whether they are allocated to some other processes which are also waiting for some other resources

if they are waiting for some other resources, pre-empt(release) those & allocated to the requested process.

Deadlock prevention

Circular Wait

- One way to avoid circular wait is to assign unique numbers to all the resources & then making the processes to request the resources only in strictly increasing (or decreasing) order.
- In order to assign a natural number to every resource, one-to-one function is used which is defined as $F:R \rightarrow N$
  N is a set of natural numbers and R is set of resources
- Let R includes tape drives, disk drives & printers, then the function F might be defined as
  F(tape drive)=1 F(disk drive)=5 F(Printer)=12
- First method used to avoid circular wait is to make processes to request the resources in increasing order
- Assume, first one process request for Ri.
- Then, this process has to request for another resource say Rj if and only if F(Rj) > F(Ri)
- For example, P1 requests for disk drive, then after some time, it can request only printer because F(printer)>F(disk drive) but it cannot request for tape drive because F(tape drive) is not greater than F(disk drive)
- Once P1 is using disk drive & also printer, then it cannot request for tape drive. So, there is no possibility of circular wait

Deadlock prevention-----circular wait contd

Circular Wait

- Another method used to avoid circular wait is, to make process to request for any resource Rj only after releasing any resources Ri such that F(Ri)>=F(Rj)
- For example, assume process P1 is using tape drive & then it requests for printer. It is allowed because F(printer)>F(tape drive)
- Now, P1 is using both tape drive and printer
- Assume, P1 needs disk drive but it cannot request because F(disk drive) is not greater than F(printer)
- So, now release all the resources which are having value greater than or equal to disk drive
- Once all the resources are released, process P1 will be in no waiting state & hence there is no circukar wait.

Deadlock avoidance

Circular Wait

- Another method used to avoid circular wait is, to make process to request for any resource Rj only after releasing any resources Ri such that F(Ri)>=F(Rj)
- For example, assume process P1 is using tape drive & then it requests for printer. It is allowed because F(printer)>F(tape drive)
- Now, P1 is using both tape drive and printer
- Assume, P1 needs disk drive but it cannot request because F(disk drive) is not greater than F(printer)
- So, now release all the resources which are having value greater than or equal to disk drive
- Once all the resources are released, process P1 will be in no waiting state & hence there is no circukar wait.

Deadlock prevention-----circular wait contd

Circular Wait

- Another method used to avoid circular wait is, to make process to request for any resource Rj only after releasing any resources Ri such that F(Ri)>=F(Rj)
- For example, assume process P1 is using tape drive & then it requests for printer. It is allowed because F(printer)>F(tape drive)
- Now, P1 is using both tape drive and printer
- Assume, P1 needs disk drive but it cannot request because F(disk drive) is not greater than F(printer)
- So, now release all the resources which are having value greater than or equal to disk drive
- Once all the resources are released, process P1 will be in no waiting state & hence there is no circukar wait.

Deadlock avoidance

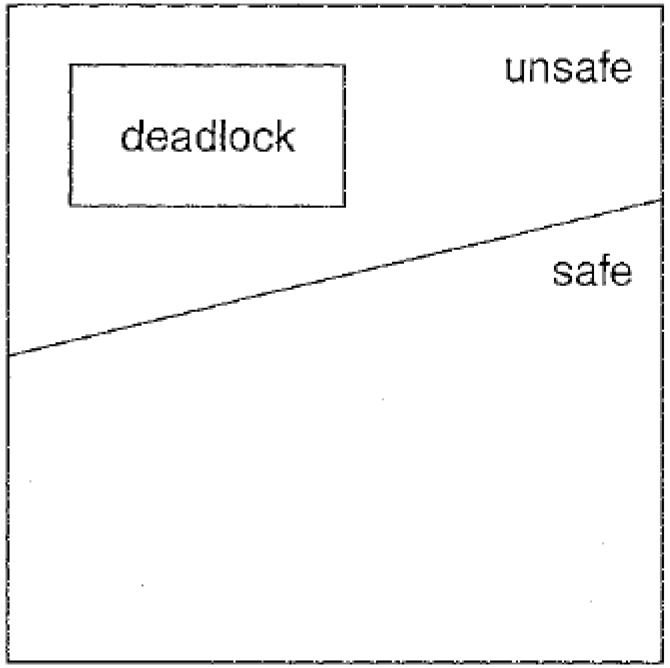
- WKT, deadlock can be avoided by avoiding any one of 4 Conditions, mutual exclusion, hold & wait, no preemption & circular wait.
- If we use this method to avoid deadlock, then it leads to
  *) less utilization of resources &
  *) reduces the system throughput.
- So we need some other methods to avoid deadlocks.
- An alternative method used to avoid deadlock is to maintain additional information about the processes & all the resources.
- with this additional information, system should decide
  *) which process is requesting which resource
  *) whether that resource is available or not
  *) whether requested resource can be allocated or allowing that requested process to wait
- There are many algorithms which uses this approach but they differ from amount & type of information required.

Deadlock avoidance

- Simplest & most useful method is that which makes process to declare maximum number of resources it requires to do it's task.
- With this prior information, an algorithm can be constructed which avoids the system from entering into a deadlock state.

Safe state

- A state is **safe**, if the system can allocate all the requested resources to a process & avoiding that process to enter into the deadlock state
- If system is in safe state, then there is no way of deadlock
- If system is in unsafe state, then it may leads to deadlock.
- But, not all unsafe states are deadlock
- As long as the state is in safe state, OS can avoid unsafe & also deadlocked state.

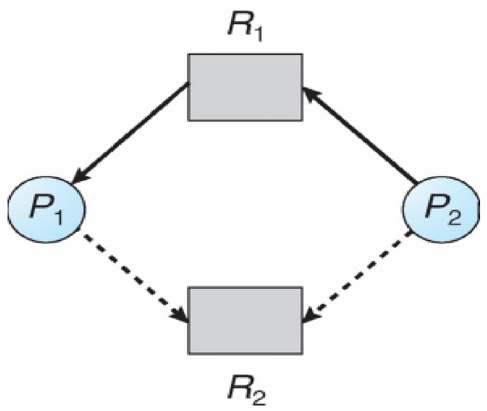


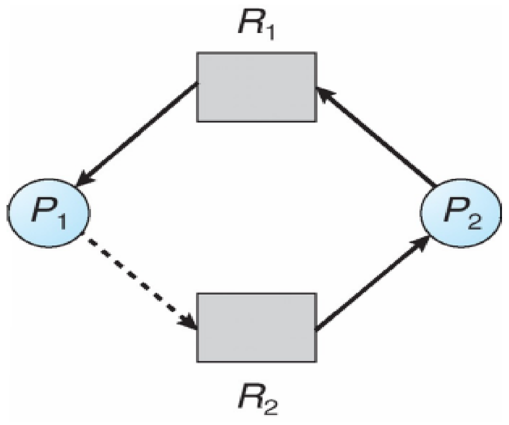
Resource allocation graph algorithm

- From resource-allocation graph, it is Known that, if there is only one instance of all the resources & the graph is having cycle, then it indicates deadlock situation.
- we also know, about request edge (Pi→Rj) and assignment edge (Rj→ Pi).
- In this resource-allocation graph algorithm, we use one more type of edge called **claim edge**.
- A claim edge is a dotted edge Pi - - - - - > Rj which indicates that process Pi may request resource Rj in future.
- When a process requests resource, then claim edge is Converted into request edge. (as Pi- - - - > Rj into Pi→ Rj)
- Similarly, when a process releases the resource, then assignment edge is converted into claim edge (ie, Rj → Pi into Pi- - - - >Rj).
- In this, resource allocation graph algorithm, all the claim edges should be included in the graph before any process starts executing.
- But, relaxation can be given to this condition, by allowing a claim edge Pi- - - - >Rj to be added to the graph only if all the edges associated with Pi are claim edges.
- "A sequent edge Pi→ Rj can be converted into arrignment edge Rf→ Pi only when this arrignment edge does not form the cycle in this resource alteration graph.If cycle is found, it indicates that rystem is in unrafe state & hence indicates deadlock Go situation.So, in this way we can avoid the deadlock.

Resource allocation graph algorithm

- A request edge Pi→ Rj can be converted into assignment edge Rj→ Pi only when this assignment edge does not form the cycle in this resource alteration graph.
- If cycle is found, it indicates that system is in unsafe state & hence indicates deadlock situation.
- So, in this way we can avoid the deadlock.





Unsafe State In Resource-Allocation Graph

Suppose that P2 requests R2. although R2 is currently free, we can not allocate it to P2, since this action may create a cycle if P1 requests R2 as well.

Bankers algorithm

1. For resource types with Multiple instances, Banker’s algorithm is to be used

2. Each process must a priori claim maximum use.

3. When a process requests a resource it may have to wait.

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

Bankers 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: $n \times m$ 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.

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

Bankers algorithm

Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

<u>Safety Algorithm</u>

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

```
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 and i such that both:
        (a) Finish [i] = false
        (b) $Need_i \le Work$
        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.
```

Bankers algorithm

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:

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

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

Bankers algorithm

Example 1

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	

Bankers algorithm

Example 1(contd.)

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.

Bankers algorithm

Example 1(contd.)

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.

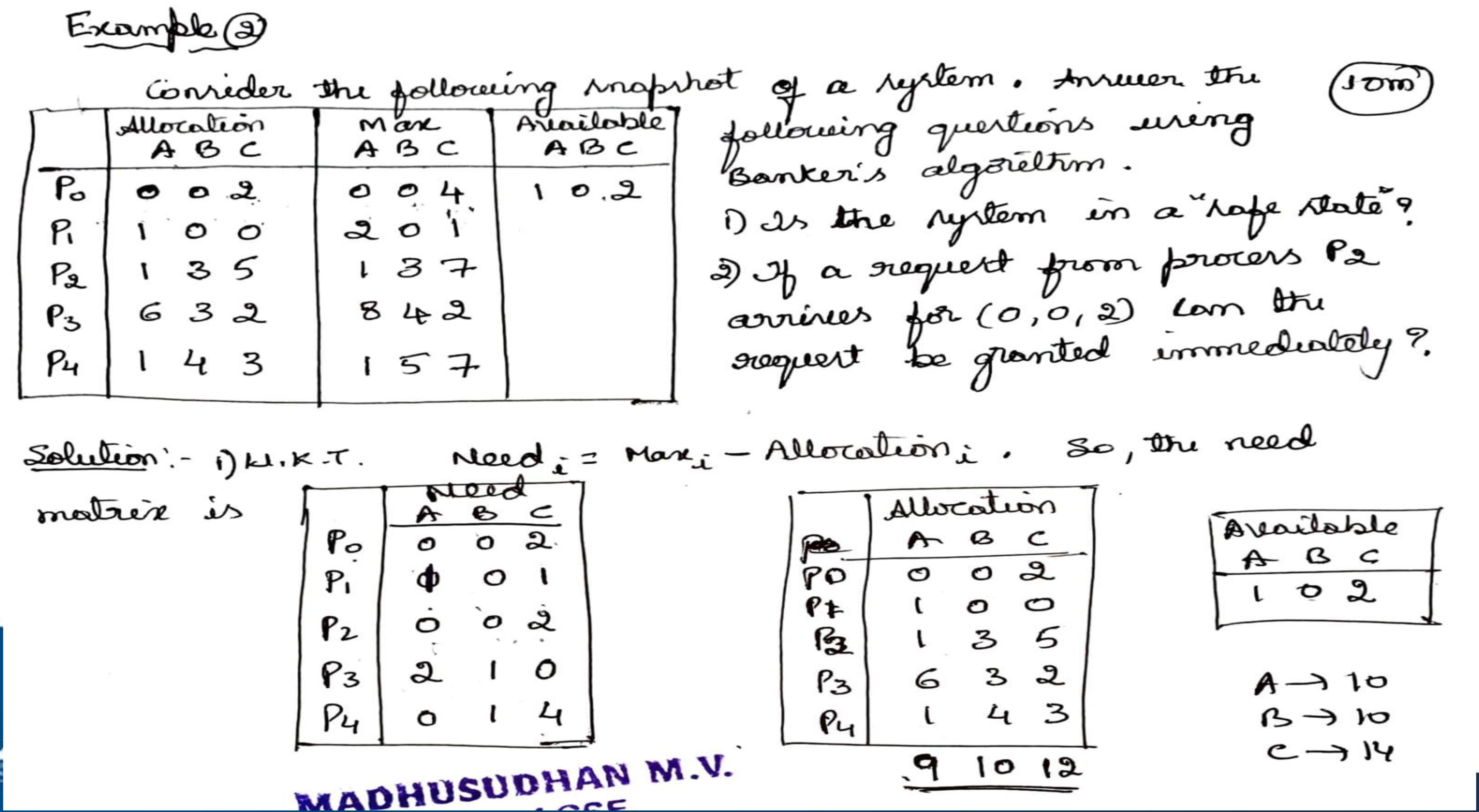
Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Bankers algorithm

More Practice Problems on Banker’s Algorithm

Bankers algorithm



Example ②

Consider the following snapshot of a system. Answer the following questions using Banker's algorithm. (10m)

	Allocation A	Allocation B	Allocation C	Max A	Max B	Max C	Available A	Available B	Available C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

1) Is the system in a "safe state"?

2) If a request from process P_2 arrives for (0, 0, 2) can the request be granted immediately?

Solution:- i) W.K.T. $Need_i = Max_i - Allocation_i$. So, the need matrix is

	Need A	Need B	Need C
P_0	0	0	2
P_1	1	0	1
P_2	0	0	2
P_3	2	1	0
P_4	0	1	4

	Allocation A	Allocation B	Allocation C
P0	0	0	2
P1	1	0	0
P2	1	3	5
P3	6	3	2
P4	1	4	3
	9	10	12

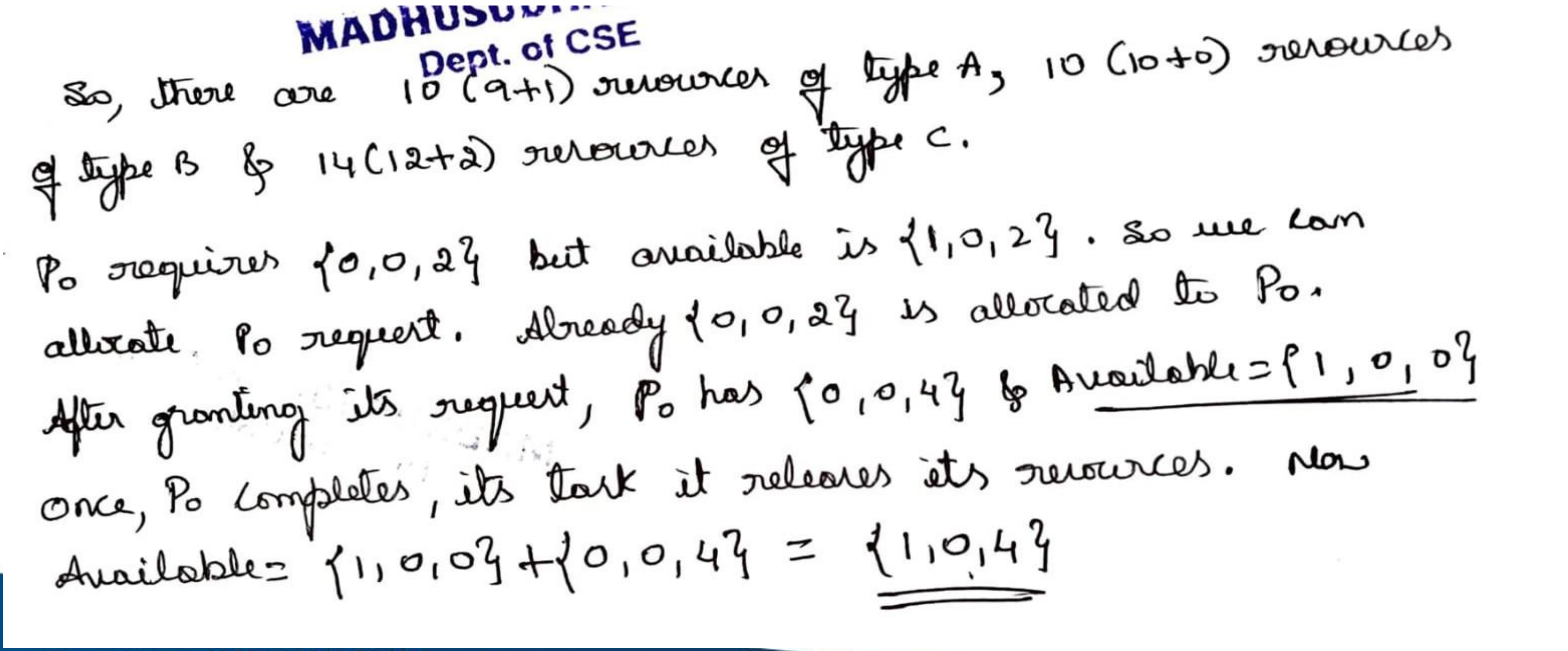
Available A	Available B	Available C
1	0	2

$A \rightarrow 10$

$B \rightarrow 10$

$C \rightarrow 14$

Bankers algorithm



MADHUSU

Dept. of CSE

So, there are 10 (9+1) resources of type A, 10 (10+0) resources of type B & 14 (12+2) resources of type C.

P_0 requires {0,0,2} but available is {1,0,2}. So we can allocate P_0 request. Already {0,0,2} is allocated to P_0. After granting its request, P_0 has {0,0,4} & Available = {1,0,0}

Once, P_0 completes, its task it releases its resources. Now

Available = {1,0,0} + {0,0,4} = {1,0,4}

	Already Allocated			Needed			Available			
	A	B	C	A	B	C	A	B	C	
P_0	0	0	2	0	0	2	1	0	2	Since, Need ≤ Available, we can grant the request.
	0	0	4	——			1	0	0	
Once P_0 completes, it releases all its allocated resources;							$\boxed{1}$	$\boxed{0}$	$\boxed{4}$	
P_1	1	0	0	1	0	1	1	0	4	Since, Need ≤ Available, we can grant the request.
	2	0	1	——			0	0	3	
Once P_1 completes, it releases all its allocated resources							$\boxed{2}$	$\boxed{0}$	$\boxed{4}$	
P_2	1	3	5	0	0	2	2	0	4	Since, Need ≤ Available, we can grant the request
	1	3	7	——			2	0	2	
Once P_2 completes, its releases all its allocated resources							$\boxed{3}$	$\boxed{3}$	$\boxed{9}$	
P_3	6	3	2	2	1	0	3	3	9	Since, Need ≤ Available, we can grant the request
	8	4	2	——			1	2	9	
Once P_3 completes, it releases all its allocated resources							$\boxed{9}$	$\boxed{6}$	$\boxed{11}$	
P_4	1	4	3	0	1	4	9	6	11	Since, Need ≤ Available, we can grant the request
	1	5	7	——			9	5	7	
once P_4 completes, it releases all its allocated resources							$\boxed{10}$	$\boxed{10}$	$\boxed{14}$	

Since, we successfully completed all the 5 processes in the sequence {P_0, P_1, P_2, P_3, P_4}, we can say <u>system is in safe state</u> & safe sequence is {P_0, P_1, P_2, P_3, P_4}.

2) If a request from P2 arrives for {0,0,2}, can the request be granted immediately?

	Allocation A B C	Max A B C	Available A B C	Need A B C
P0	0 0 2	0 0 4	1 0 0	0 0 2
P1	1 0 0	2 0 1		1 0 1
P2	1 3 7	1 3 7		0 0 0
P3	6 3 2	8 4 2		2 1 0
P4	1 4 3	1 5 7		0 1 4

	Already allocated A B C	Needed A B C	Available A B C	
× P0	0 0 2	0 0 2	1 0 0	Since, $Need_0$ > Available, not possible to grant.
× P1	1 0 0	1 0 1	1 0 0	Since, $Need_1$ > Available, not possible to grant
✓ P2	1 3 7	0 0 0	1 0 0	Since, P2 got all its resources, it Completes its task & releases all its resources.

After P2 releases its resources $\boxed{2\ 3\ 7}$

✓ P_3	6 3 2	2 1 0	2 3 7
	8 4 2	—	0 2 7

Since, P3 Need ≤ Available, we can grant the request

once P3 Completes, it releases all its resources $\boxed{8\ 6\ 9}$

✓ P_4	1 4 3	0 1 4	8 6 9
	1 5 7	—	8 5 5

Since, Need ≤ Available, we can grant the request

once P4 Completes, it releases all its resources $\boxed{9\ 10\ 12}$

✓ P_0	0 0 2	0 0 2	9 10 12
	0 0 4	—	9 10 10

Need ≤ Available

After P0 Completes its task, it releases all its resources $\boxed{9\ 10\ 14}$

✓ P_1	1 0 0	1 0 1	9 10 14
	2 0 1	—	8 10 13

Need ≤ Available

After P1 releases its resources $\boxed{10\ 10\ 14}$

We completed all 5 processes successfully even after granting the additional request of P2, we can say system is in safe state & safe sequence is {P2, P3, P4, P0, P1}.

Hence, ~~*~~ additional request of P2 can be granted immediately.

Example 3

6m

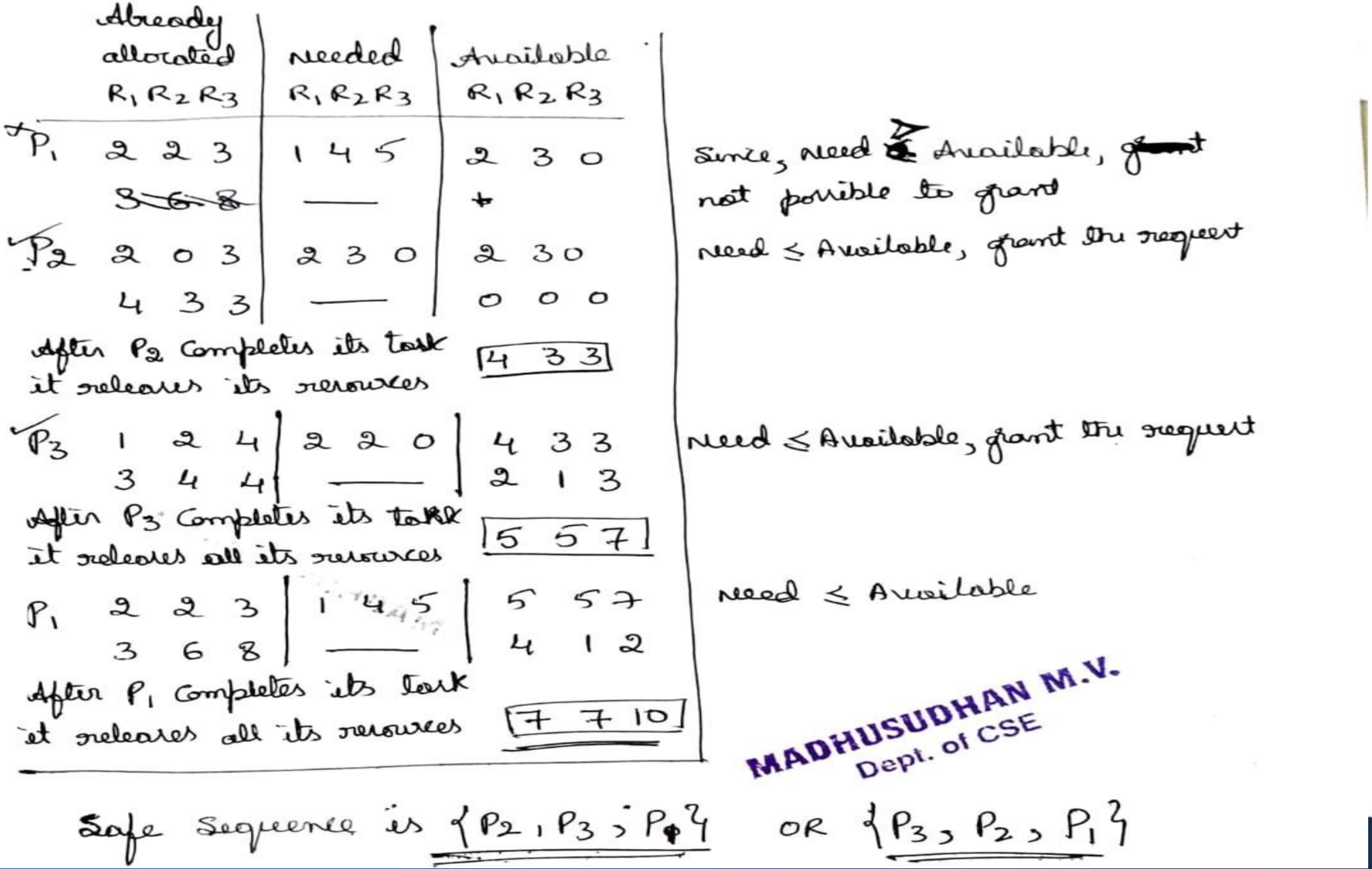
For the following snapshot, find the safe sequence using Banker's algorithm? Number of resource units are R1, R2 & R3 which are 7, 7, 10 respectively.

	Allocated resources			Maximum requirements		
	R1	R2	R3	R1	R2	R3
P1	2	2	3	3	6	8
P2	2	0	3	4	3	3
P3	1	2	4	3	4	4

Solution :-

	Allocated resources R_1 R_2 R_3	Max requirements R_1 R_2 R_3	Need R_1 R_2 R_3	Available R_1 R_2 R_3
P_1	2 2 3	3 6 8	1 4 5	2 3 0
P_2	2 0 3	4 3 3	2 3 0	
P_3	1 2 4	3 4 4	2 2 0	
	5 4 10			

Available = {7, 7, 10} − {5, 4, 10} = 2, 3, 0



	Already allocated $R_1 R_2 R_3$	Needed $R_1 R_2 R_3$	Available $R_1 R_2 R_3$	
✓P_1	2 2 3	1 4 5	2 3 0	Since, Need > Available, ~~grant~~ not possible to grant
	~~3 6 8~~	——	*	
✓P_2	2 0 3	2 3 0	2 3 0	Need ≤ Available, grant the request
	4 3 3	——	0 0 0	

After P_2 Completes its task it releases its resources $\boxed{4\ 3\ 3}$

✓P_3	1 2 4	2 2 0	4 3 3	Need ≤ Available, grant the request
	3 4 4	——	2 1 3	

After P_3 Completes its task it releases all its resources $\boxed{5\ 5\ 7}$

P_1	2 2 3	1 4 5	5 5 7	Need ≤ Available
	3 6 8	——	4 1 2	

After P_1 Completes its task it releases all its resources $\boxed{7\ 7\ 10}$

MADHUSUDHAN M.V.
Dept. of CSE

Safe Sequence is $\{P_2, P_3, P_1\}$ OR $\{P_3, P_2, P_1\}$

Example (4) (8m)

Consider the following snapshot of resource allocation at time T_1

	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			

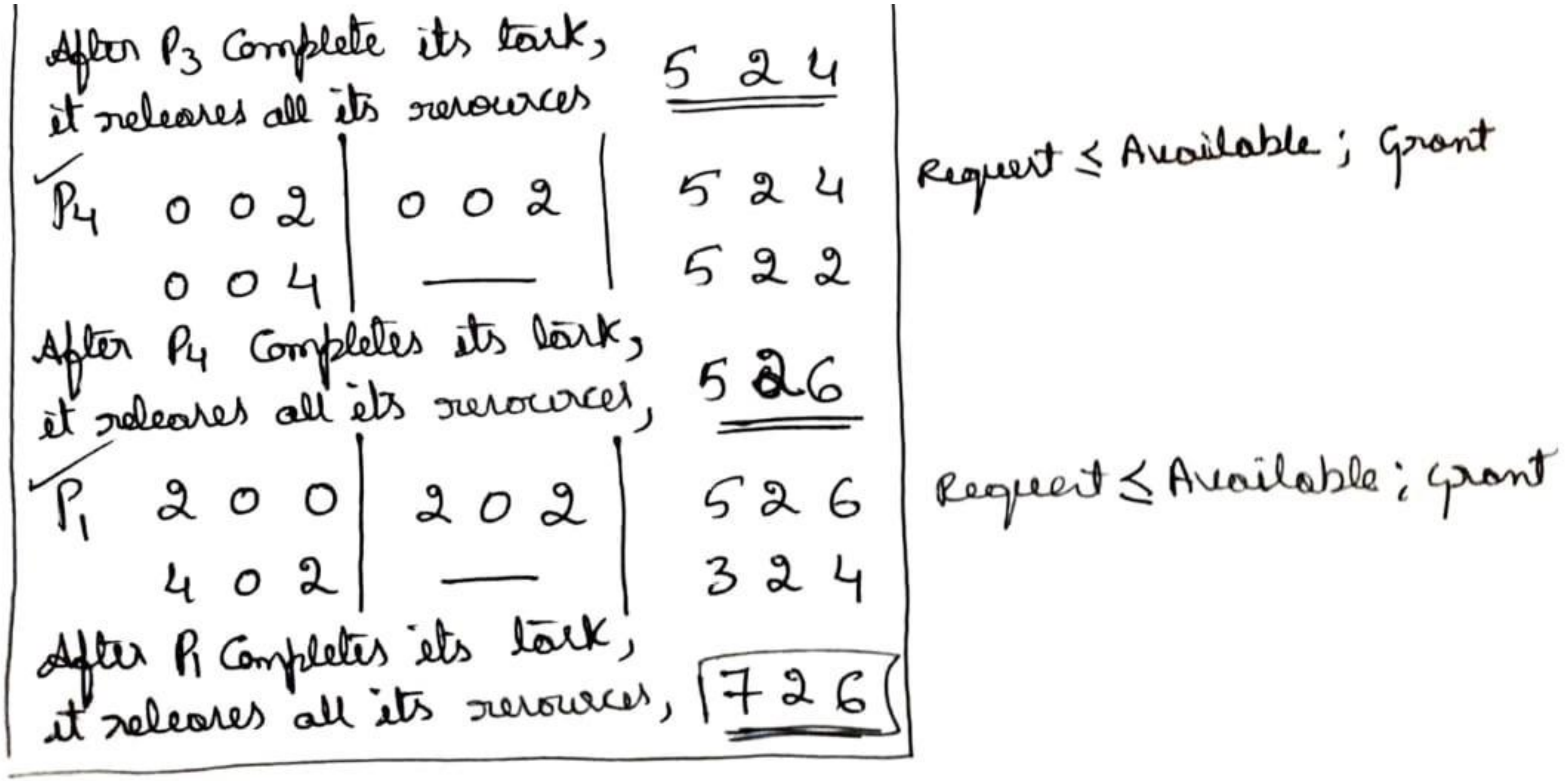
a) Show that, system is not deadlocked by generating one safe sequence

b) At interval t_2, P_2 makes one additional request for ~~an~~ instance of type C. Show that, system is deadlocked if the request is granted. Write down the deadlocked processes.

Solution :-

	Allocation	(Need) Request	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	0 0 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		

	Already allocated	Request	Available	
	A B C	A B C	A B C	
✓ P_0	0 1 0	0 0 0	0 0 0	~~need~~ Request ≤ Available ; grant
	0 1 0	——	0 0 0	
After P_0 Completes its task it releases all its resources			$\underline{0\ 1\ 0}$	
× P_1	2 0 0	2 0 2	0 1 0	Request > Available ; not possible to grant
✓ P_2	3 0 3	0 0 0	0 1 0	Since P_2 doesn't requires anything it Completes & releases its resources
			$\underline{3\ 1\ 3}$	
✓ P_3	2 1 1	1 0 0	3 1 3	Request ≤ Available ; Grant
	3 1 1	——	2 1 3	



After P_3 Complete its task, it releases all its resources $\underline{\underline{5\ 2\ 4}}$

✓P_4	0 0 2	0 0 2	5 2 4	Request ≤ Available ; Grant
	0 0 4	—	5 2 2	

After P_4 Completes its task, it releases all its resources, $\underline{\underline{5\ 2\ 6}}$

✓P_1	2 0 0	2 0 2	5 2 6	Request ≤ Available ; Grant
	4 0 2	—	3 2 4	

After P_1 Completes its task, it releases all its resources, $\boxed{\underline{7\ 2\ 6}}$

Since, we completed all the ~~tasks~~ processes successfully, we can say system is in safe state (not deadlocked) & the safe sequence is $\{P_0, P_2, P_3, P_4, P_1\}$.

b) At time t_2, if P_2 makes request for instance of type C, then the snapshot is as follows

	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	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	Already Allocated			Request			Available			
	A	B	C	A	B	C	A	B	C	
✓P_0	0	1	0	0	0	0	0	0	0	Since P_0 doesn't request anything, it
							0	1	0	Completes its task & releases resources
×P_1	2	0	0	2	0	2	0	1	0	Request > Available; Not possible to grant
×P_2	3	0	3	0	0	1	0	1	0	Request > Available; Not possible to grant
×P_3	2	1	1	1	0	0	0	1	0	Request > Available; Not possible to grant
×P_4	0	0	2	0	0	2	0	1	0	Request > Available; Not possible to grant

Here, only process P_0 can be complete & all remaining 4 processes P_1, P_2, P_3 & P_4 are in deadlocked state.

Deadlock Detection

1. Allow system to enter deadlock state

2. Detection algorithm

3. Recovery scheme

Deadlock Detection

Single Instance of each Resource type

- Maintain *wait-for* graph

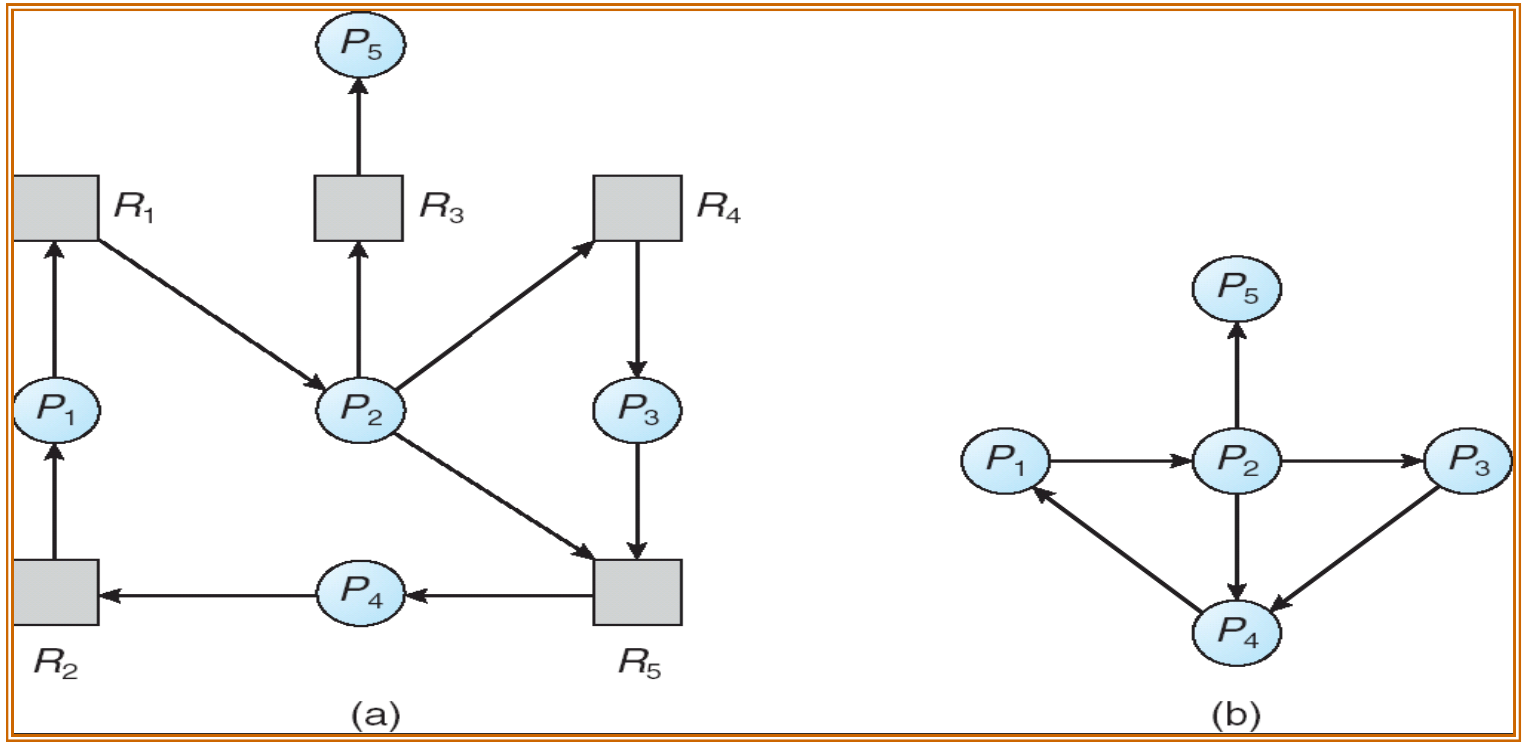
  Nodes are processes.

  $P_i \rightarrow P_j$ 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.

- 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.

Deadlock Detection



Resource Allocation GraphCorresponding Wait-for-Graph

Deadlock Detection

Several instances of a Resource type

Detection Algorithm

***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.

Deadlock Detection

Detection Algorithm

1. Let *Work* and *Finish* be two arrays of length *m* and *n*, respectively Initialize:
   (a) *Work = Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   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*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Deadlock Detection

Example

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*.

Deadlock Detection

Example(contd)

P_2 requests an additional instance of type *C*.

	Request
	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.

Deadlock Detection

Detection Algorithm Usage

- When, and how often, to invoke depends on:
  1. How often a deadlock is likely to occur?
  2. 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?
  1. Priority of the process.
  2. How long process has computed, and how much longer to completion.
  3. Resources the process has used.
  4. Resources process needs to complete.
  5. How many processes will need to be terminated.
  6. 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.