Deadlocks

Chapter Objectives

- What causes deadlock
- Methods for preventing, avoiding, or detecting deadlocks in a computer system

System Model

Resource types $R_1, R_2, . . ., R_m$ CPU *cycles, memory space, I/O devices*

Each resource type R_i has *1 or more* instances

While accessing each process must follow following sequence:

- request .... Use system call
- Use
- Release------------- use system call

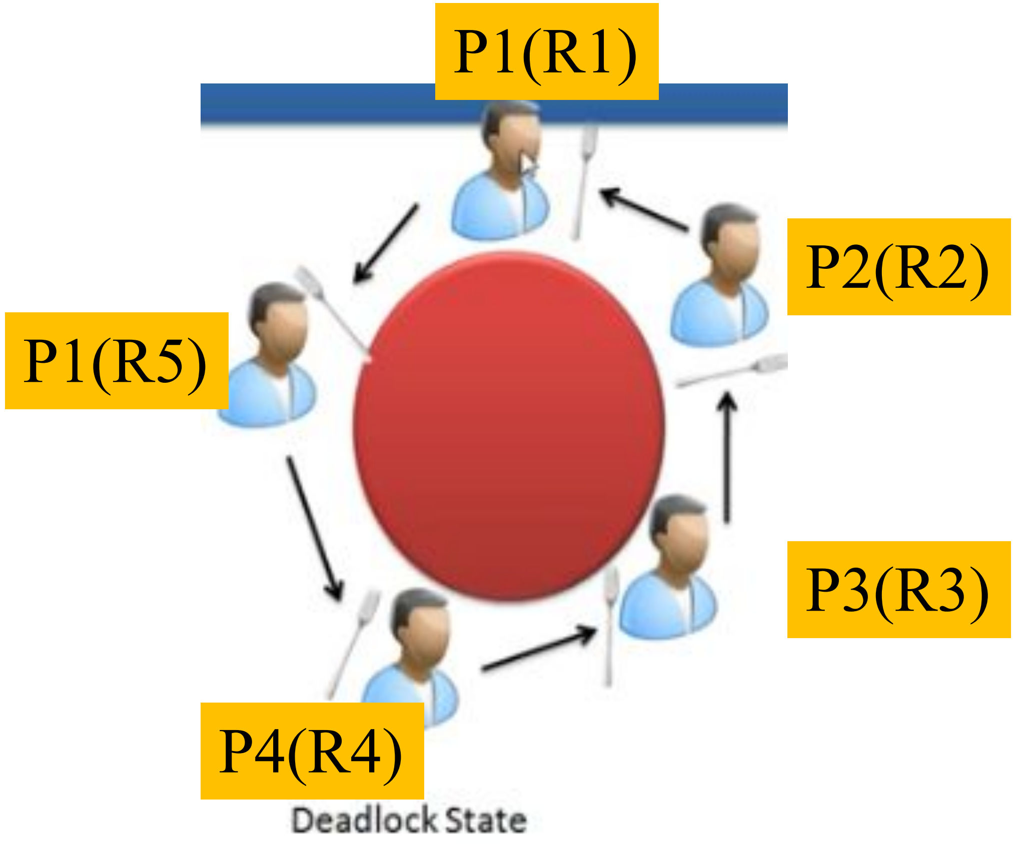
- For the resource that is not managed by system call, process uses semaphore or locks

- Resource allocation depends on requirement and availability.

- Allocation information is maintained by the system

Deadlock

- The set of processes are said to be in deadlock state, if each process in a set is holding some resource and is **waiting for the resource** that is held by another process in a set.



System Model

- A deadlock consists of a **set of blocked processes**, each holding a resource and waiting to acquire a resource held by another process in the set
- Deadlock may involve different resource type

Proc1:

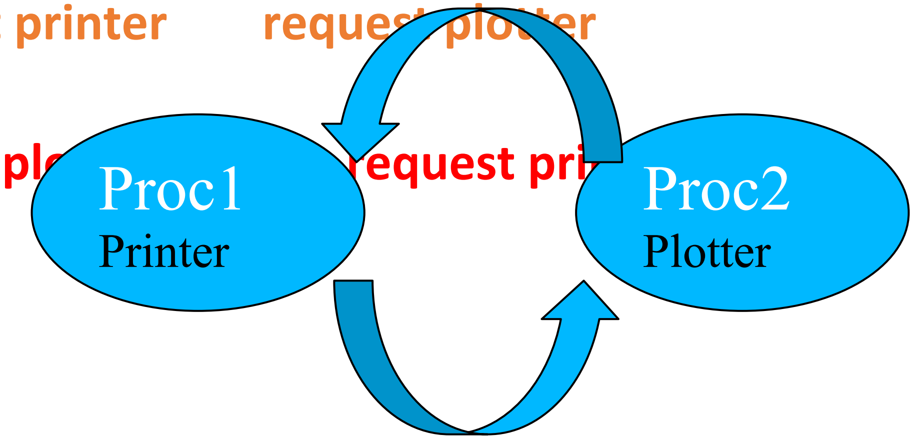
request printer

request pl

Proc2:

request plotter

request pri

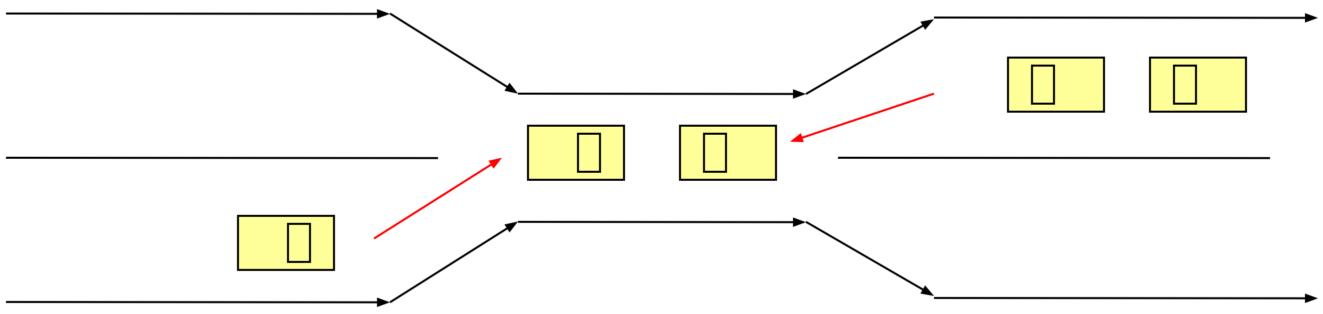


Deadlock in semaphore

- Semaphores *A* and *B*, initialized to 1

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

Bridge Crossing Example



- Multiprogramming system:
    - Multiple processes (vehicles)
    - Scanty resource (one_lane bridge)
    - Traffic only in one direction
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- **Several cars may have to be backed up if a deadlock occurs**
- Starvation is possible

Problems due to Deadlock

- **Processes** never complete their execution
- **Resources** are tied up, preventing other processes from starting

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** Resources cannot be preempted; that is a resource can be released only voluntarily by the process holding it after that process has completed its task
- **Circular wait:** A set { P0 , Pl, ... , P11 } of waiting processes must exist such that Po is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by Po

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consists of 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

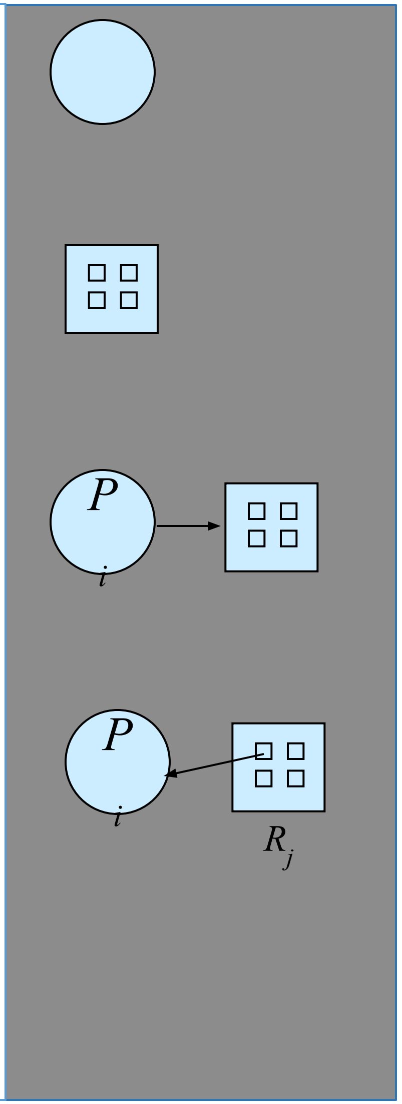
- **request edge** – directed edge $P_i \rightarrow R_j$

- **assignment edge** – directed edge $R_j \rightarrow P_i$

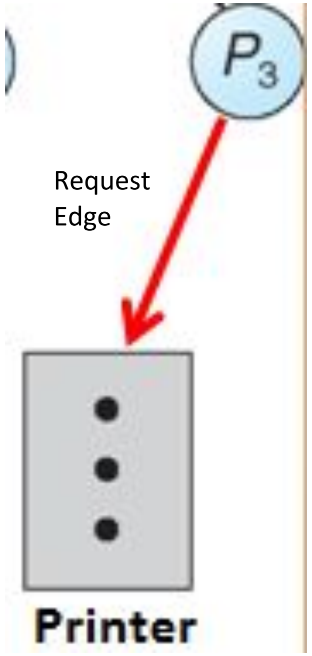
Resource-Allocation Graph (Cont.)

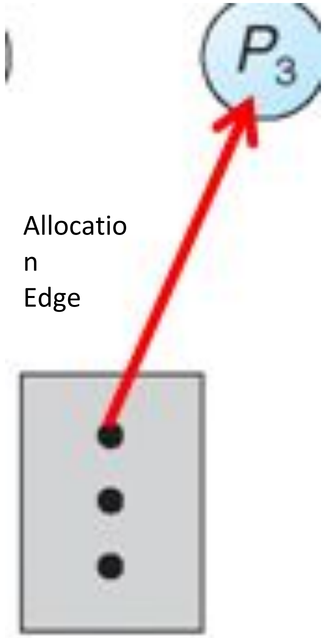
- Process
- Resource Type with 4 instances
- P_i requests instance of resource type R_j
- P_i is holding an instance of R_j

R_j



Example of a Resource Allocation Graph

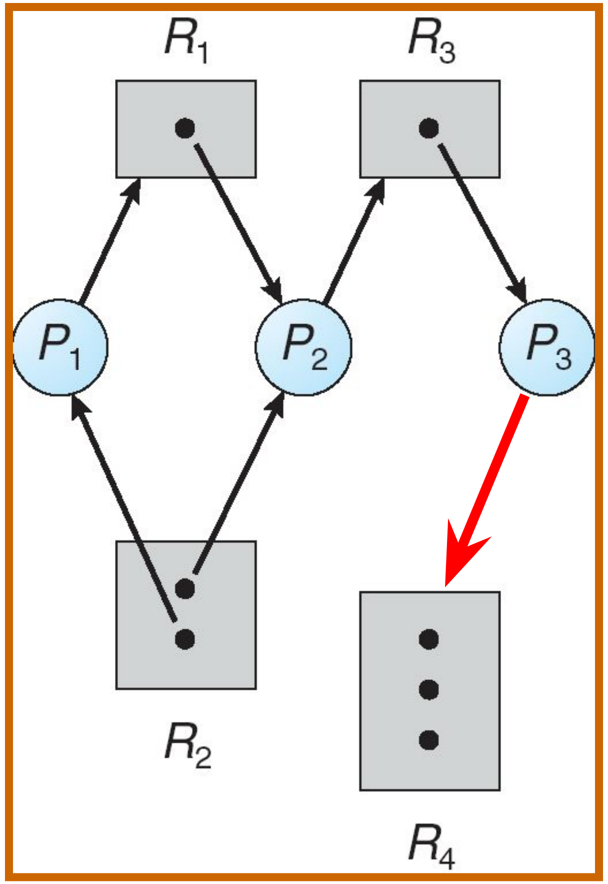




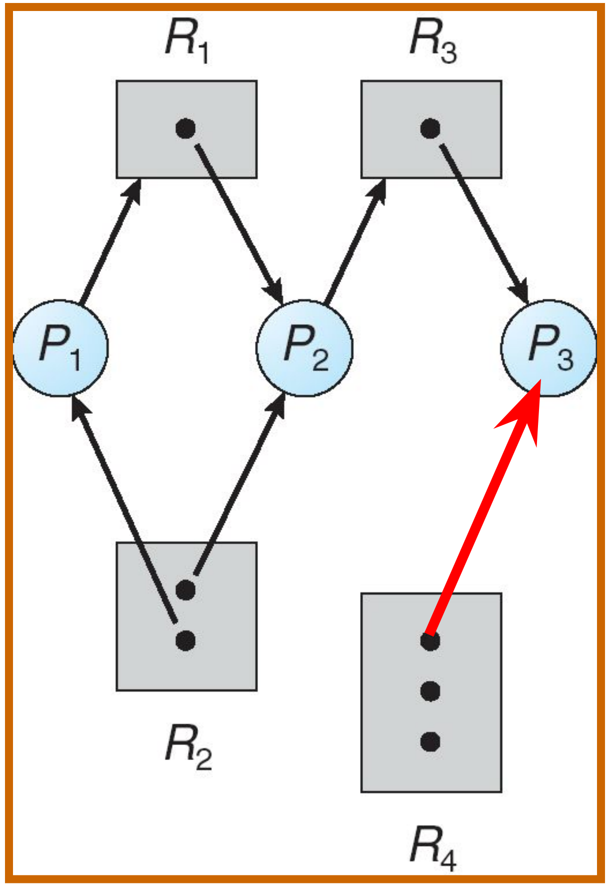
Request

Allocation

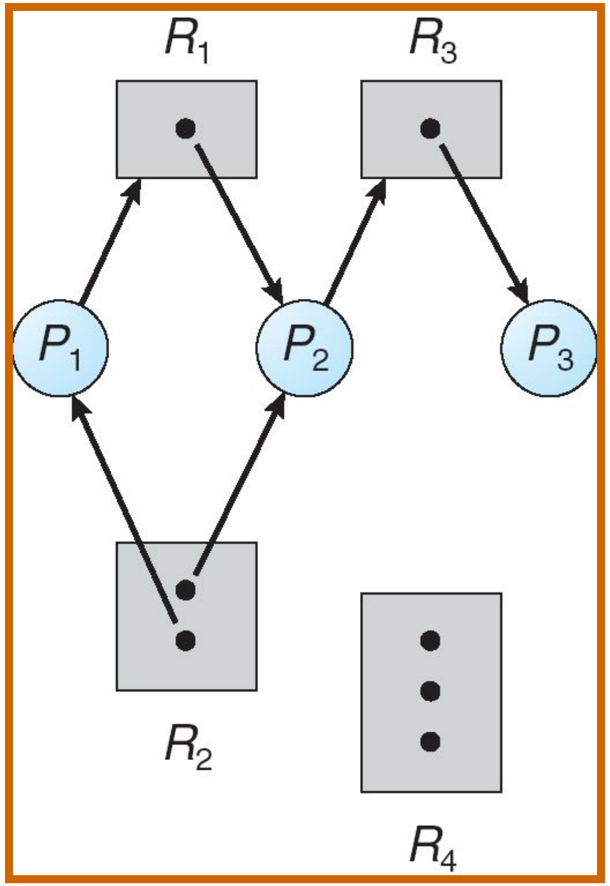
System Resource-Allocation Graph (contd..)



request

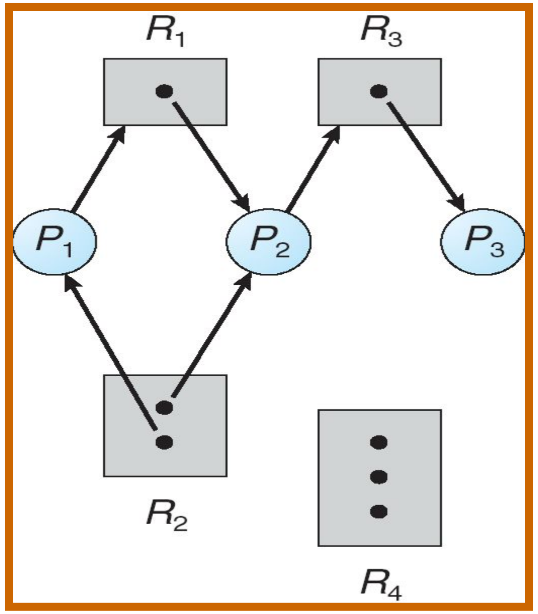


Assign



release

System Resource-Allocation Graph (contd..)



If cycle then deadlock
Else no deadlock

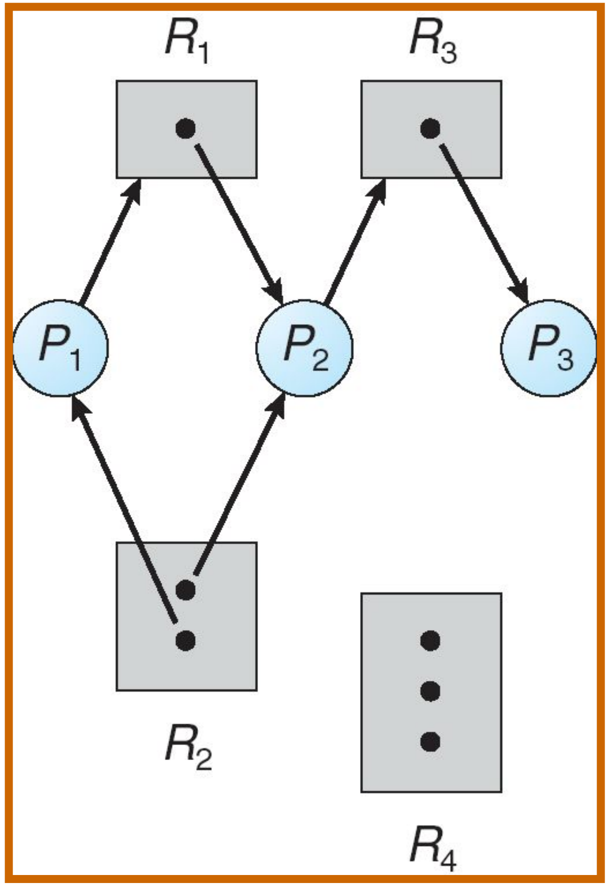
- **Current status**
  - P={P1,P2,P3}
  - R={R1,R2,R3,R4}
  - E={P1→R1,P2→R3,R1→P2,R2→P2,R2→P1,R3→P3}
- **Resource instances:**
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4
- **Process states:**
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3 .

Resource-allocation graph.

- Given the definition of a resource-allocation graph, if the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
- Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
- To illustrate this concept, we return to the resource-allocation graph depicted in Figure 7.2.

Resource Allocation Graph With A Deadlock

Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2

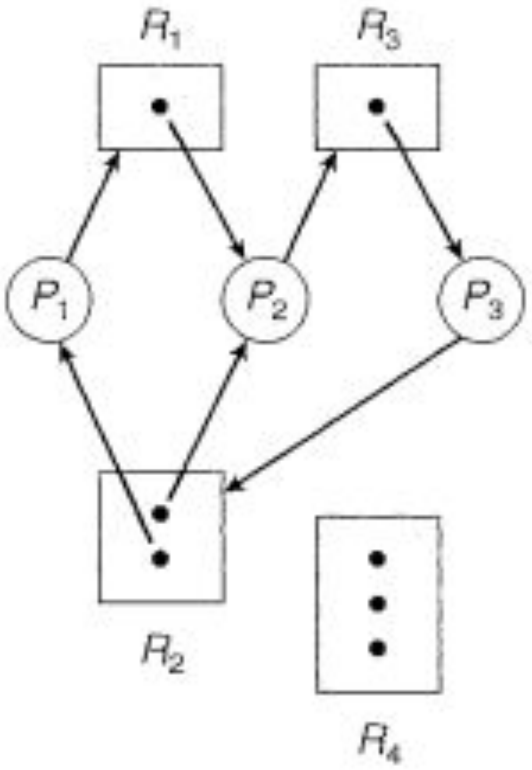


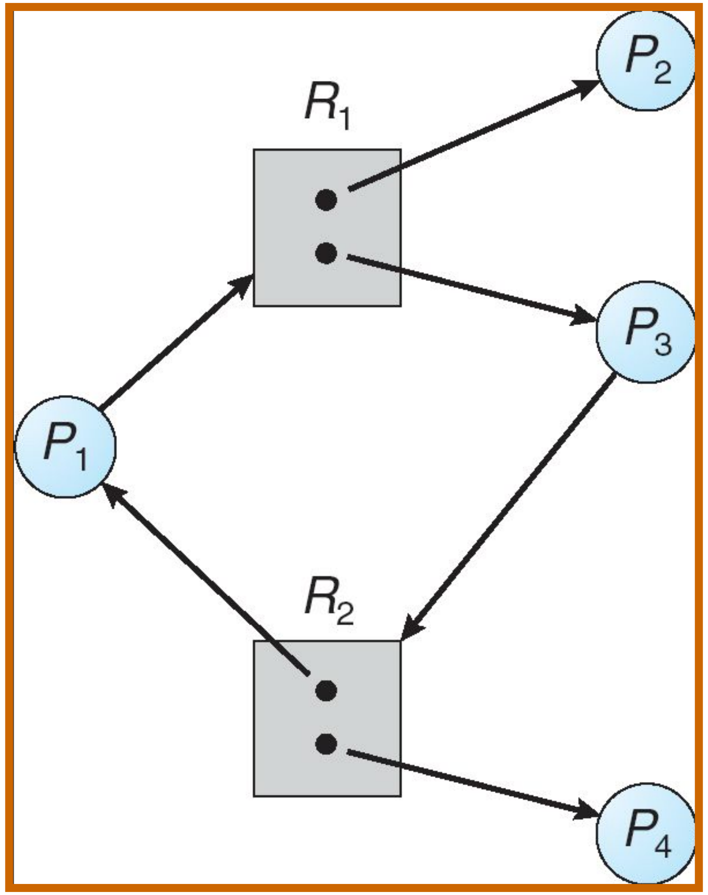
Figure 7.3 Resource-allocation graph with a deadlock.

Resource Allocation Graph With A Deadlock

- Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 R2 is added to the graph (Figure 7.3). At this point, two minimal cycles exist in the system:
- P1R1P2R3P3R2P1
- P2R3P3R2P2

- Processes P1, P2, and P3 are deadlocked.
- Process P2 is waiting for the resource R3, which is held by process P3.
- Process P3 is waiting for either process P1 or process P2 to release resource R2.
- In addition, process P1 is waiting for process P2 to release resource R1.

Graph with a cycle but no deadlock



Now consider the resource-allocation graph in Figure 7.4.
In this example,
we also have a cycle:
P1R1P3R2P1

However, there is no deadlock.
Observe that process P4 may release its instance of resource type R2.
That resource can then be allocated to P3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.
If there is a cycle, then the system may or may not be in a deadlocked state.
This observation is important when we deal with the deadlock problem.

Relationship of cycles to deadlocks

- If a resource allocation graph contains **no cycles** $\Rightarrow$ **no deadlock**

- If a resource allocation graph contains a **cycle and if only one instance exists per resource type** $\Rightarrow$ **deadlock**

- If a resource allocation graph contains a **cycle and and if several instances exists per resource type** $\Rightarrow$ **possibility of deadlock**

Que: Consider a system with four processes P1, P2, P3, and P4, and two resources, R1, and R2,

respectively.

Each resource has two instances. Furthermore:

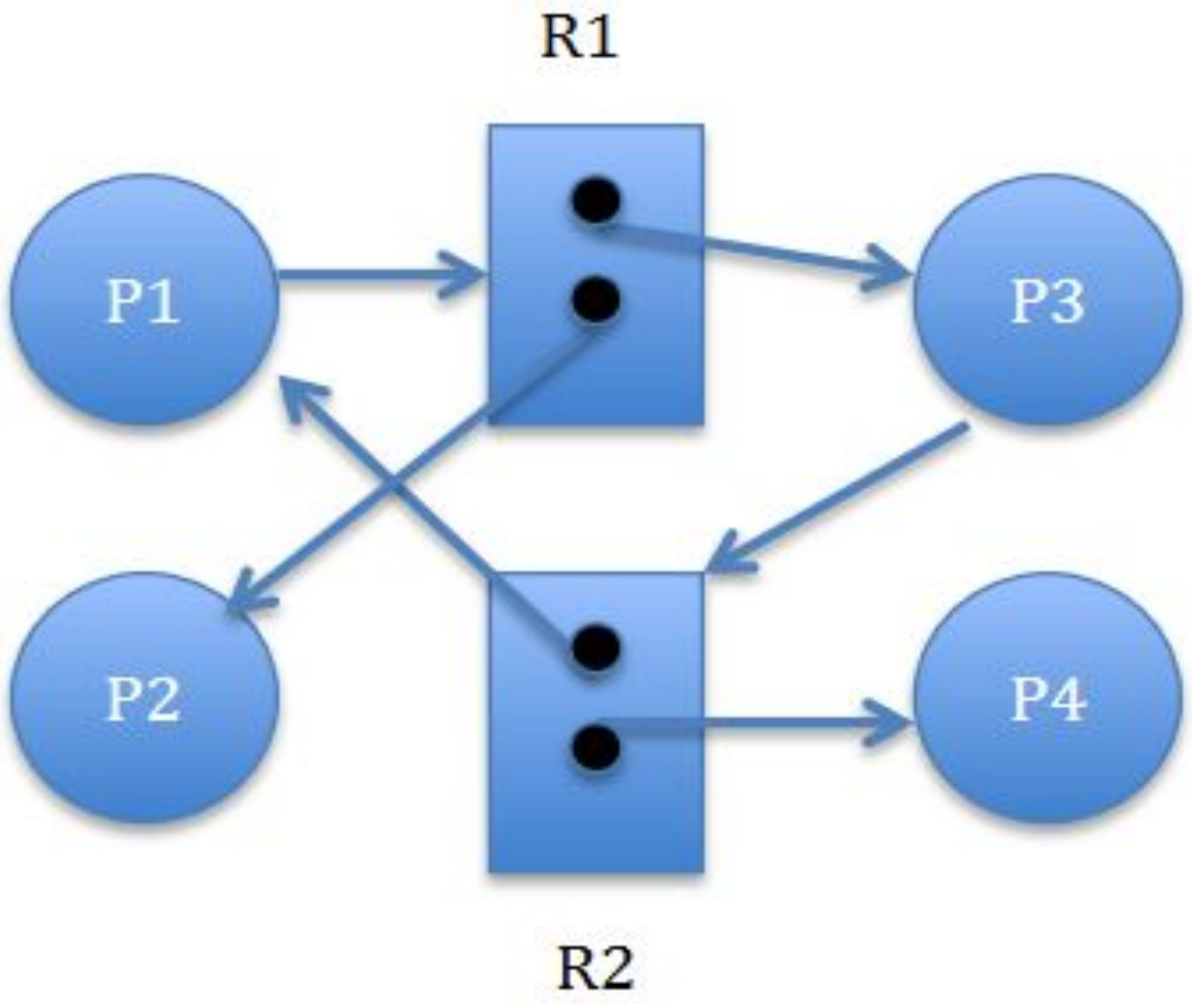
- P1 allocates an instance of R2, and requests an instance of R1
- P2 allocates an instance of R1, and doesn’t need any other resource
- P3 allocates an instance of R1 and requires an instance of R2
- P4 allocates an instance of R2, and doesn’t need any other resource

Draw the resource allocation graph.

Is there a cycle in the graph? If yes name it.
P1 R1 P3 R2 P1

Solution:

Resource allocation graph



Que: A system has three types of resources R1 R2 R3 and their number of units are 3, 2, and 2 respectively. Four processes P1 P2 P3 and P4 are currently competing for these resources in following number.

i) P1 is holding one unit of R1 and is requesting for one unit of R2.

ii) P2 is holding two units of R2 and is requesting for one unit each of R1 and R3.

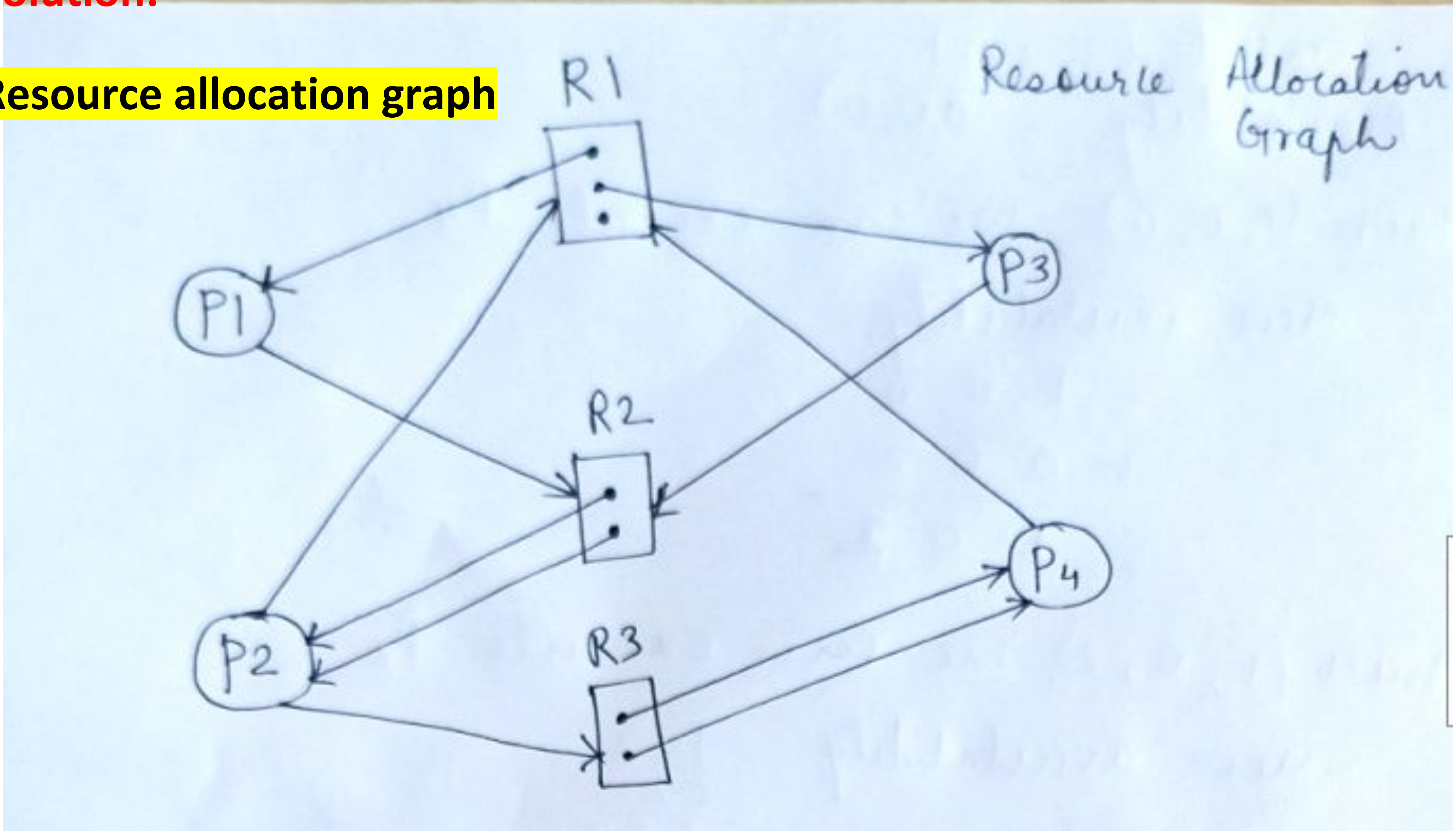
iii) P3 is holding one unit of R1 and is requesting for one unit of R2.

iv) P4 is holding two units of R3 and requesting for one unit of R1.

Draw the recourse allocation graph for above. Determine if any of the processes are in deadlock state?

Solution:

Resource allocation graph



Cycle but no deadlock

There are cycles in the graph

$P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_4 \rightarrow R_1 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_4 \rightarrow R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2$

$P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_4 \rightarrow R_1 \rightarrow P_3$

$P_4 \rightarrow R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_4$

Now, we will ~~fou~~ find out whether deadlock exists or no

Process	Allocation R_1	Allocation R_2	Allocation R_3	Request R_1	Request R_2	Request R_3
P_1	1	0	0	0	1	0
P_2	0	2	0	1	0	1
P_3	1	0	0	0	1	0
P_4	0	0	2	1	0	0

No processes are in deadlock state.

Total (3, 2, 2)

Availability = (1, 0, 0)

with (1, 0, 0), we can execute P_4

New availability

$$\begin{array}{r} = 1, 0, 0 \\ +\ 0\ \ 0\ \ 2 \\ \hline 1\ \ 0\ \ 2 \end{array}$$

with (1, 0, 2), we can execute P_2

New availability

$$\begin{array}{r} 1\,0\,2 \\ 0\,2\,0 \\ \hline 1\,2\,2 \end{array}$$

with (1, 2, 2) we can execute P_1

New availability

$$\begin{array}{r} 1\ 2\ 2 \\ 1\ 0\ 0 \\ \hline 2\ 2\ 2 \end{array}$$

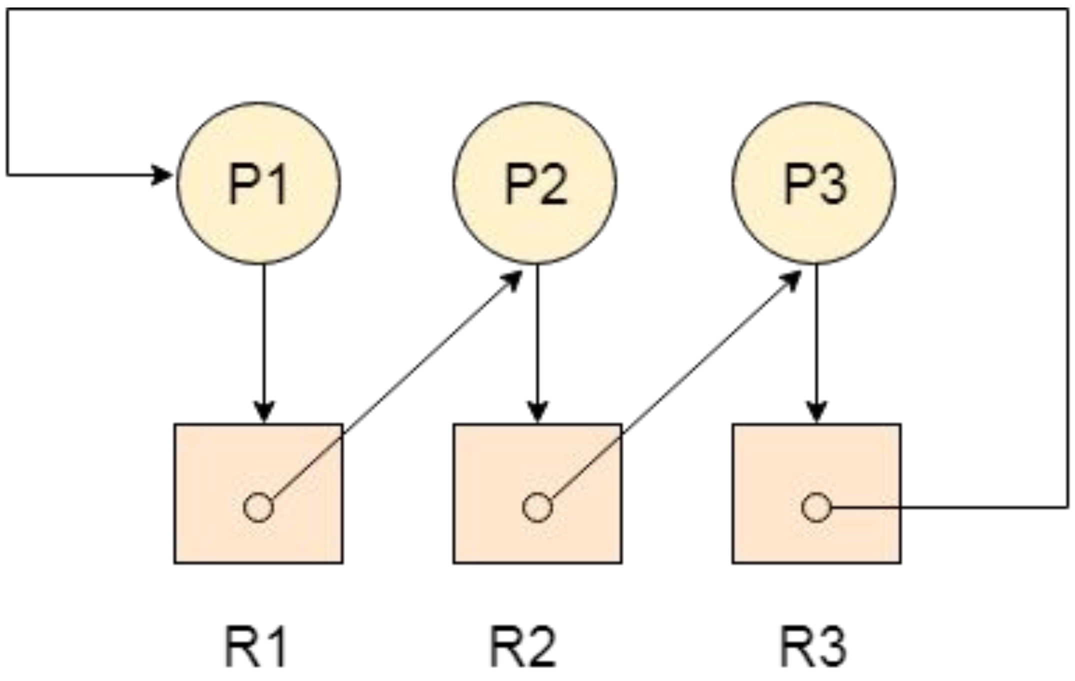
with (2, 2, 2), we can execute P_3

$$\begin{array}{r} 2\ 2\ 2 \\ 1\ 0\ 0 \\ \hline 3\ 2\ 2 \end{array}$$

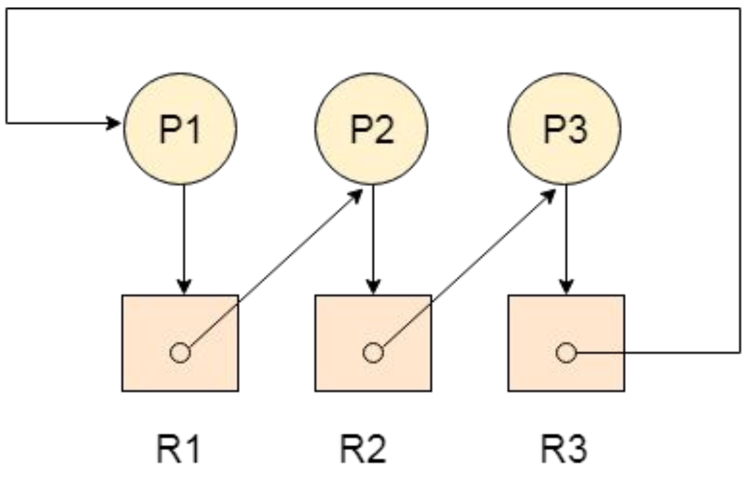
All the processes have been executed.

Hence no Process is in deadlock state.

Que.
The following example contains three processes P1, P2, P3 and three resources R1, R2, R3. All the resources are having single instances each.



Solution:



Allocation Matrix

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix

Process	R 1	R 2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

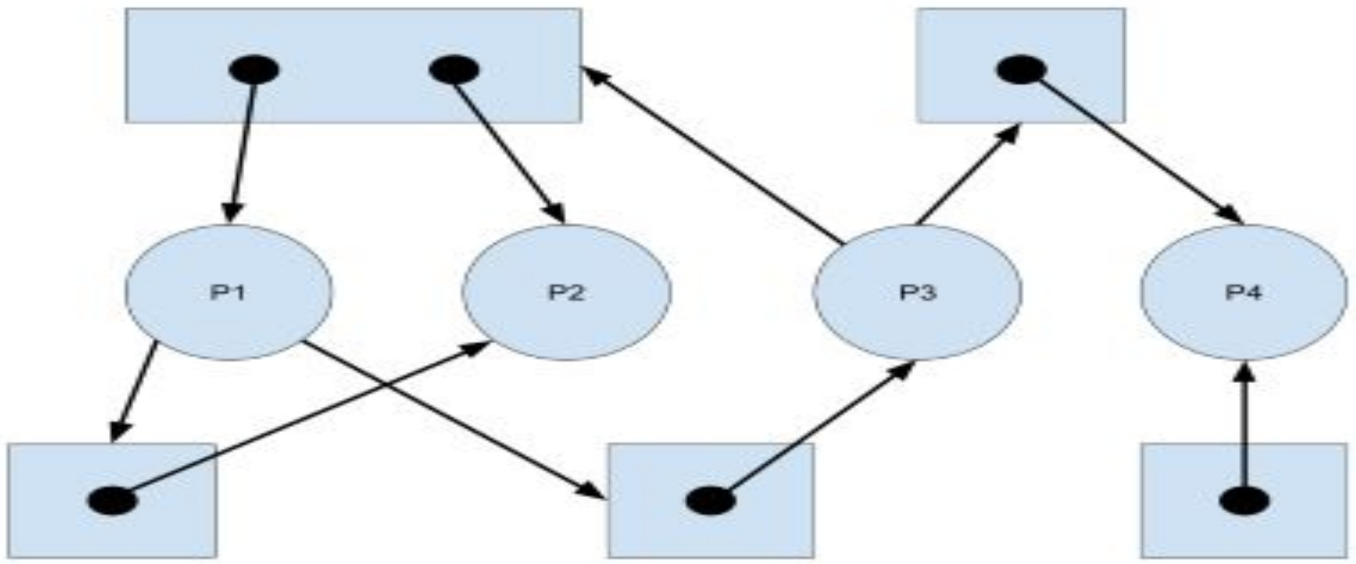
Neither we are having any resource available in the system nor a process going to release.

Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.

We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

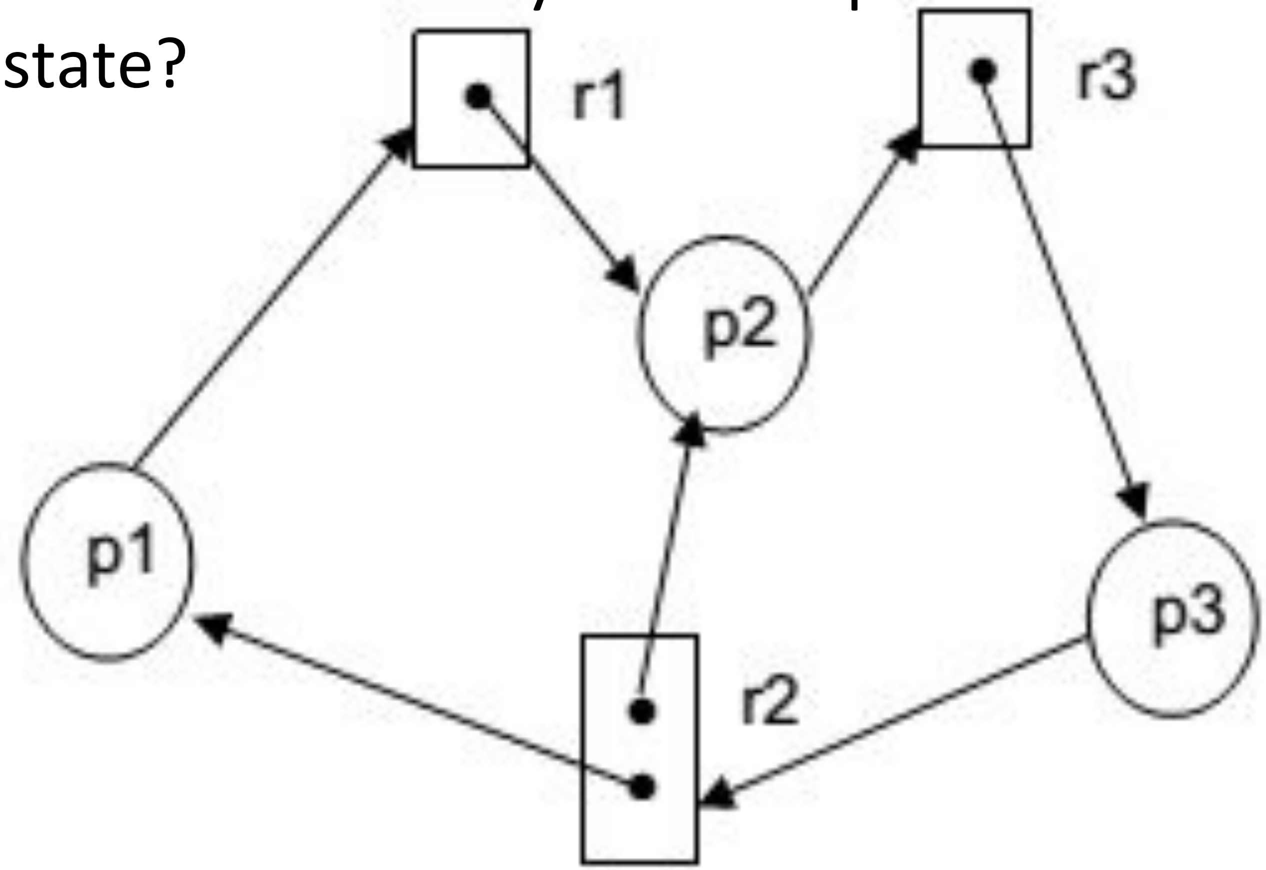
<u>**Objective**</u>: In solving this question, you be able to interpret a resource allocation graph as well as a Wait-for graph. By describing a resource allocation graph, you will see how processes can request and hold multiple resources. Understanding this is essential to follow the process order of completion and determine if deadlock is existent in the system.

1. Describe the following resource allocation graph.
   EX: P1 -> R1: Process P1 is requesting instance of R1.

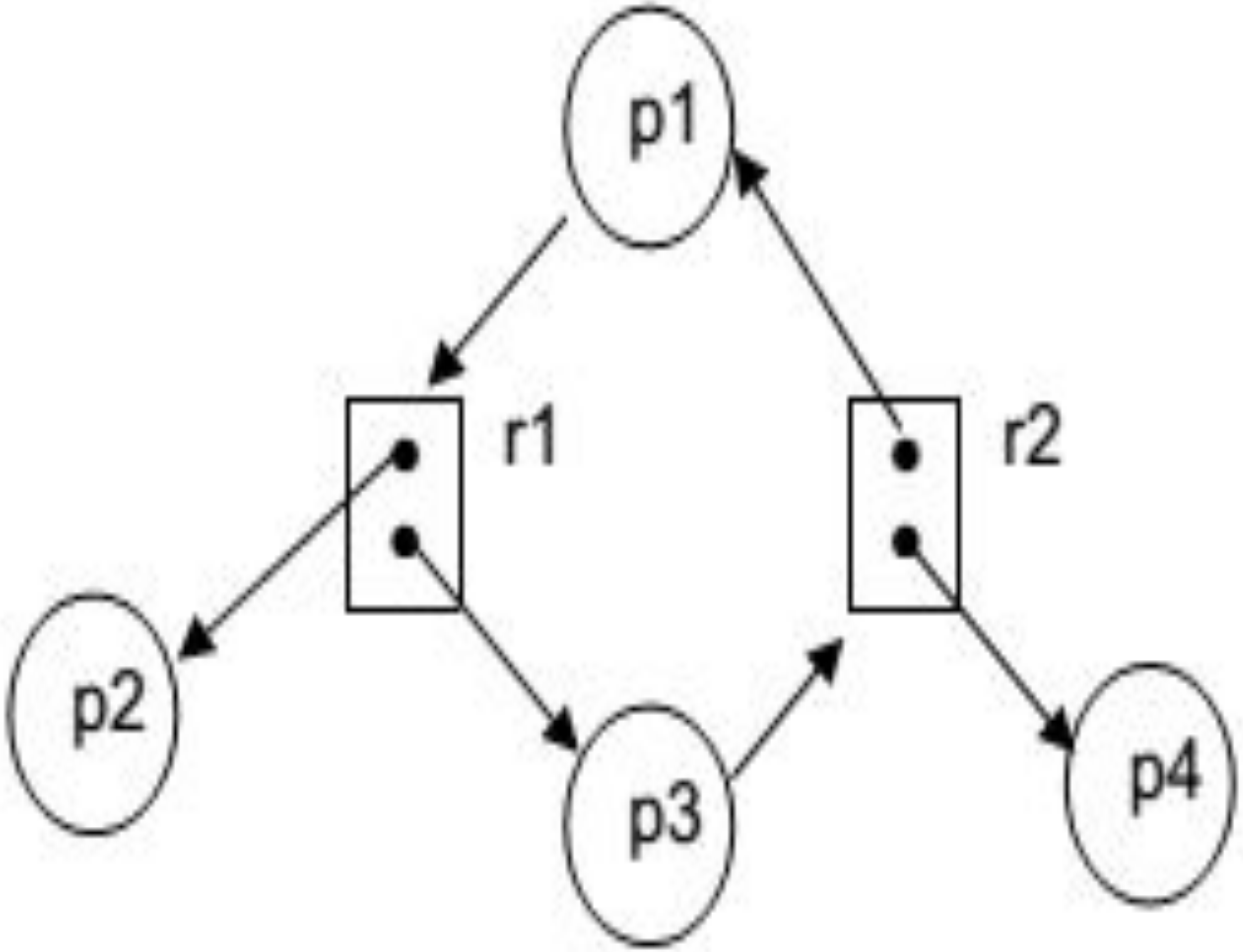


Determine if any of the processes are in deadlock state?

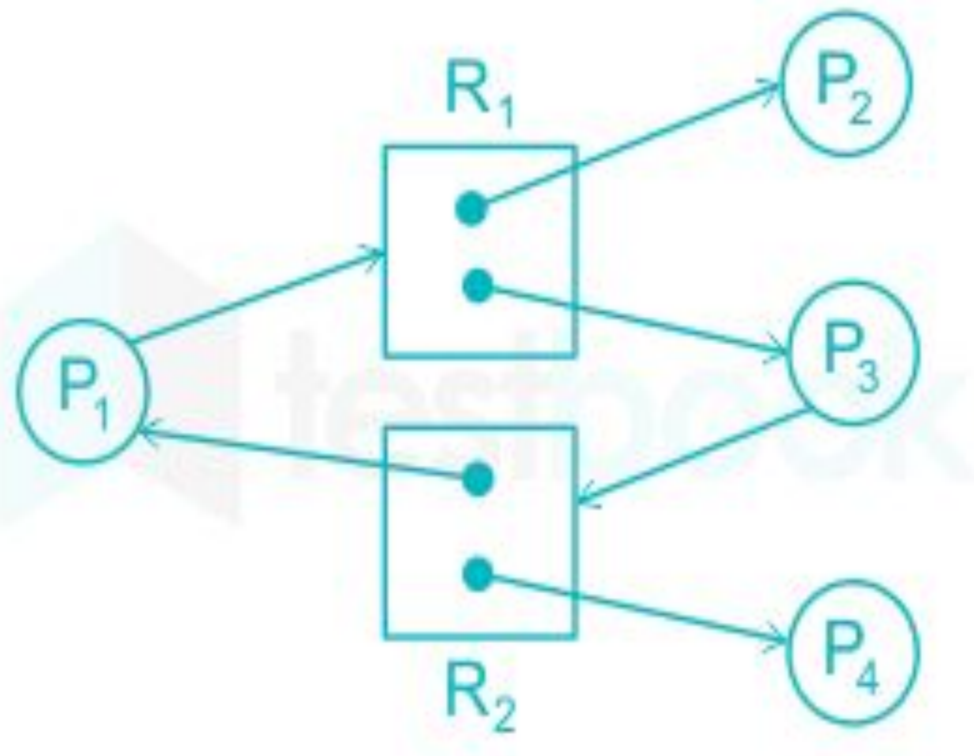
Determine if any of the processes are in deadlock state?



Determine if any of the processes are in deadlock state?

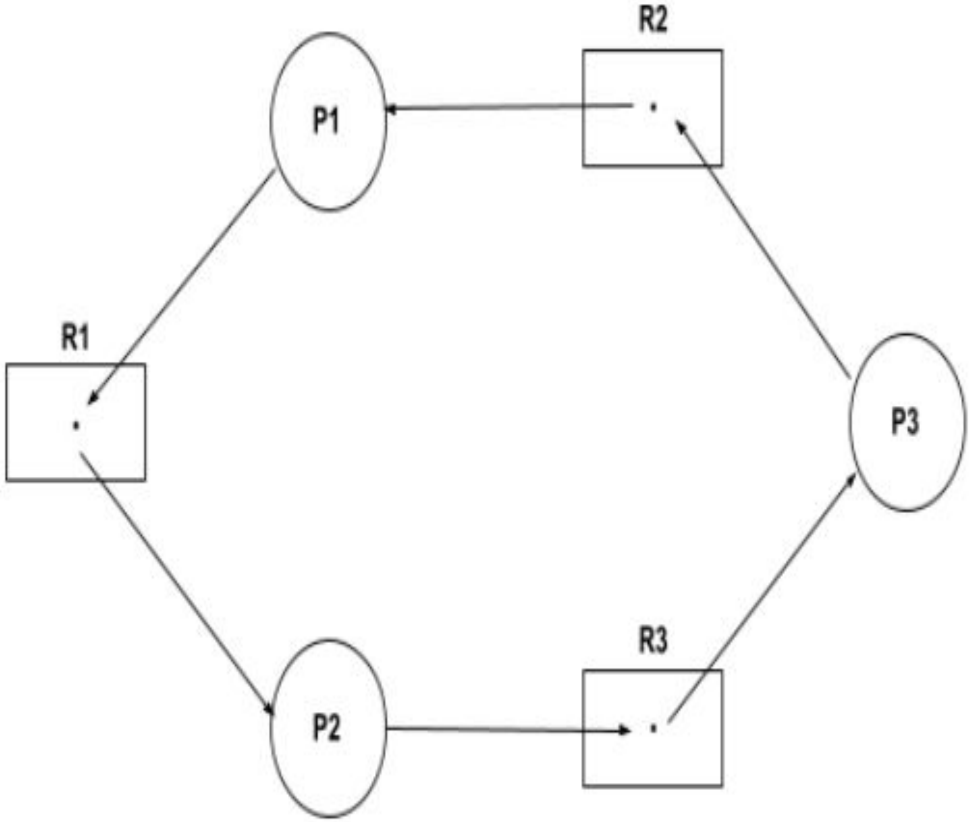


Determine if any of the processes are in deadlock state?



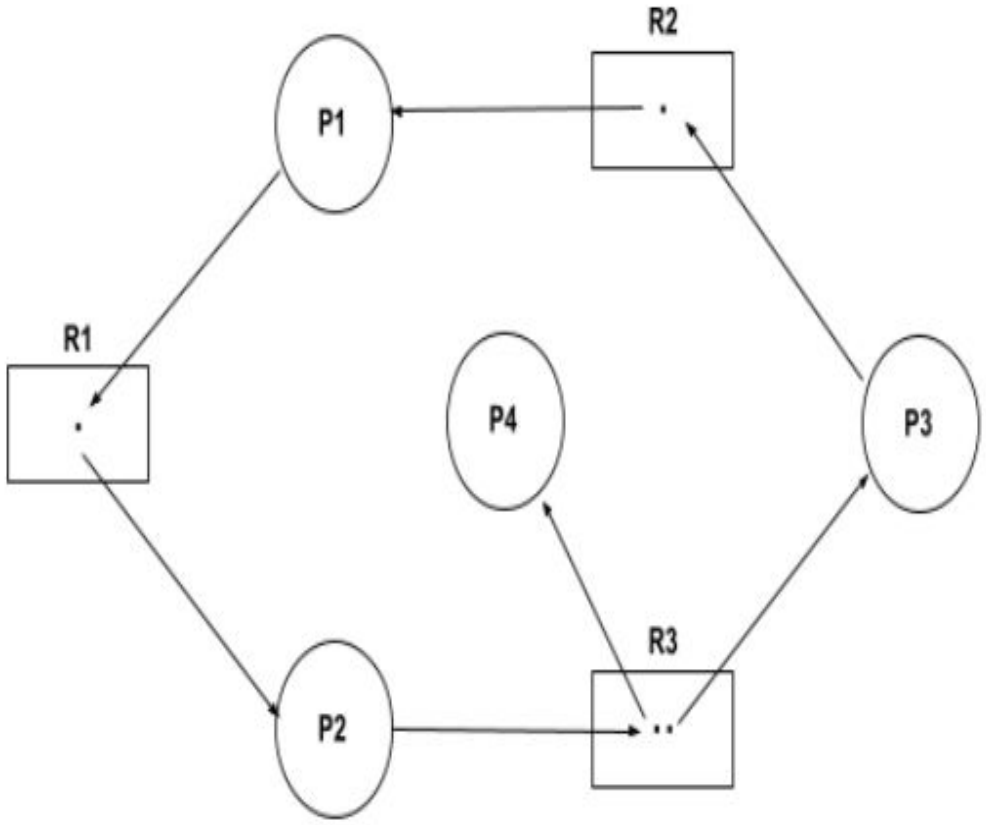
	Allocation resources		Request resources	
	R_1	R_2	R_1	R_2
P_1	0	1	1	0
P_2	1	0	0	0
P_3	1	0	0	1
P_4	0	1	0	0

Determine if any of the processes are in deadlock state?

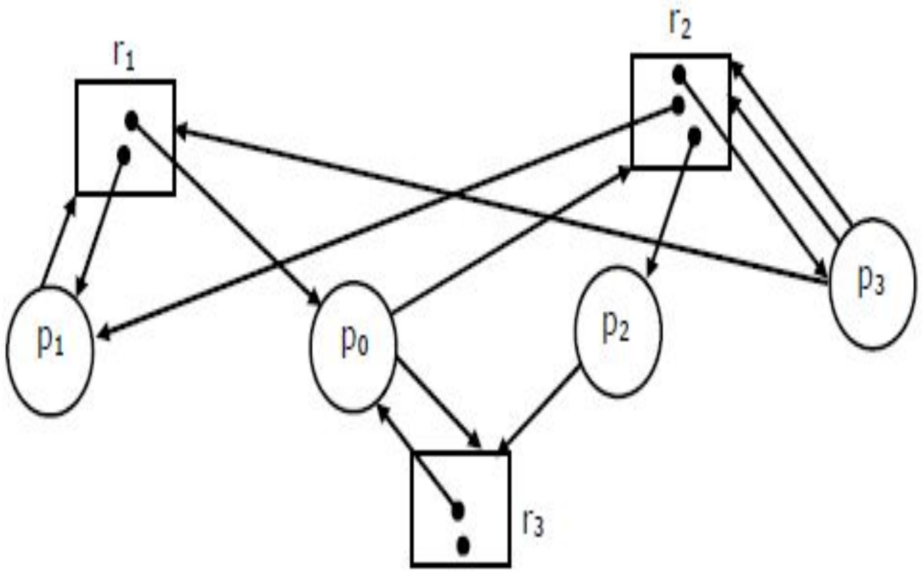


Process	Allocation			Request		
	Resource			Resource		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	1	0	0
P2	1	0	0	0	0	1
P3	0	0	1	0	1	0

Determine if any of the processes are in deadlock state?



Process	Allocation Resource			Request Resource		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	1	0	0
P2	1	0	0	0	0	1
P3	0	0	1	0	1	0
P4	0	0	1	0	0	0



(a) Find if the system is in a deadlock state.

Methods for Handling Deadlocks

- **Use of protocol to prevent / Avoid deadlock**
  - Ensure that the system will *never* enter a deadlock state
- **Detection** & Recovery
  - Allow the system to enter a deadlock state and then recover
- **Do Nothing**
  - Ignore the problem and pretend that deadlock never occur in a system.
    - and let the user or system administrator handle the problem; used by most operating systems, including **Windows and UNIX**

Deadlock Prevention

Avoid one of the necessary condition.

- **Mutual Exclusion** –
  - The mutual-exclusion condition must hold for nonsharable resources.
  - For example, a printer cannot be simultaneously shared by several processes.
  - Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.
  - Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.
- **Hold and Wait –**
  - To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that whenever a process **requests** a resource, it **does not hold** any other resources
  - Require a process to request and be allocated all its resources **before it begins execution,**
  - or allow a process to request resources **only when the process has none**
    - **Example CD ROM – COPY ON DISK & sort -- PRINT**
  - Result: Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** – **(release & regain)**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - A process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait-**
- The fourth and final condition for deadlocks is the circular-wait condition.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- To illustrate, we let R = { R1, R2, ... , Rm} be the set of resource types.

- We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Formally, we define a one-to-one funtion F: R ⍰N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
- F (tape drive) = 1
- F (disk drive) = 5
- F (printer) = 12
- We can now consider the following protocol to prevent deadlocks:
- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of a resource type -say, Ri. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) >=F(Rj). It must also be noted that if several instances of the same resource type are needed, a single request for all of them must be issued.
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Prevention Problems

- Thus deadlock can be prevented by making one the required condition false.

- By restricting process to access resource in particular order.

- Problems
  - Low resource utilization
  - Reduced system throughput

Deadlock Avoidance

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

- Set of processes, Resources, Ordering of resources, request and release of resource of each process.

- By using this information, for each request system identify whether there will be deadlock in future or not.

- Different algorithm require different information.

Deadlock Avoidance

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

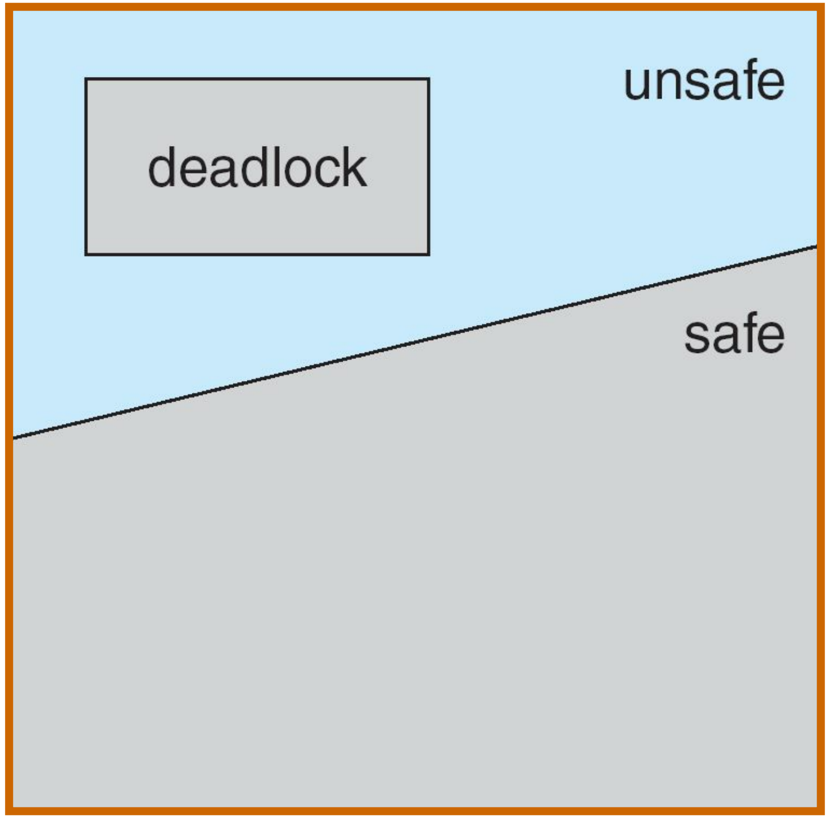
Safe, Unsafe , Deadlock State

Safe State:

All processes can Complete the task.

Safe sequence

No deadlock



Unsafe State:

May lead to Deadlock state

- Deadlock Avoidance algorithm ensure that a system will <u>never</u> enter an <u>unsafe</u> state

Safe State (continued)

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

Safe State

- When a process **requests** an available resource, the system must decide if **immediate allocation leaves the system in a safe state**

- **In safe state system can allocate resources to each process (upto its maximum) and still avoid deadlock.**

- **Resources are allocated in** a **safe sequence.**

Safe State

- A sequence of processes $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make, can be satisfied by currently available resources plus resources held by all P_j, with $j < i$.

- That is:
  - If the 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, $\boldsymbol{P_{i+1}}$ **can obtain** its needed resources, and so on

Number of Tape Drives = 12

Process	Maximum Need	Allocation	Current Need	Available
P0	10	5	5	3
P1	4	2	2	
P2	9	2	2	

Safe sequence: P1,P0,P2

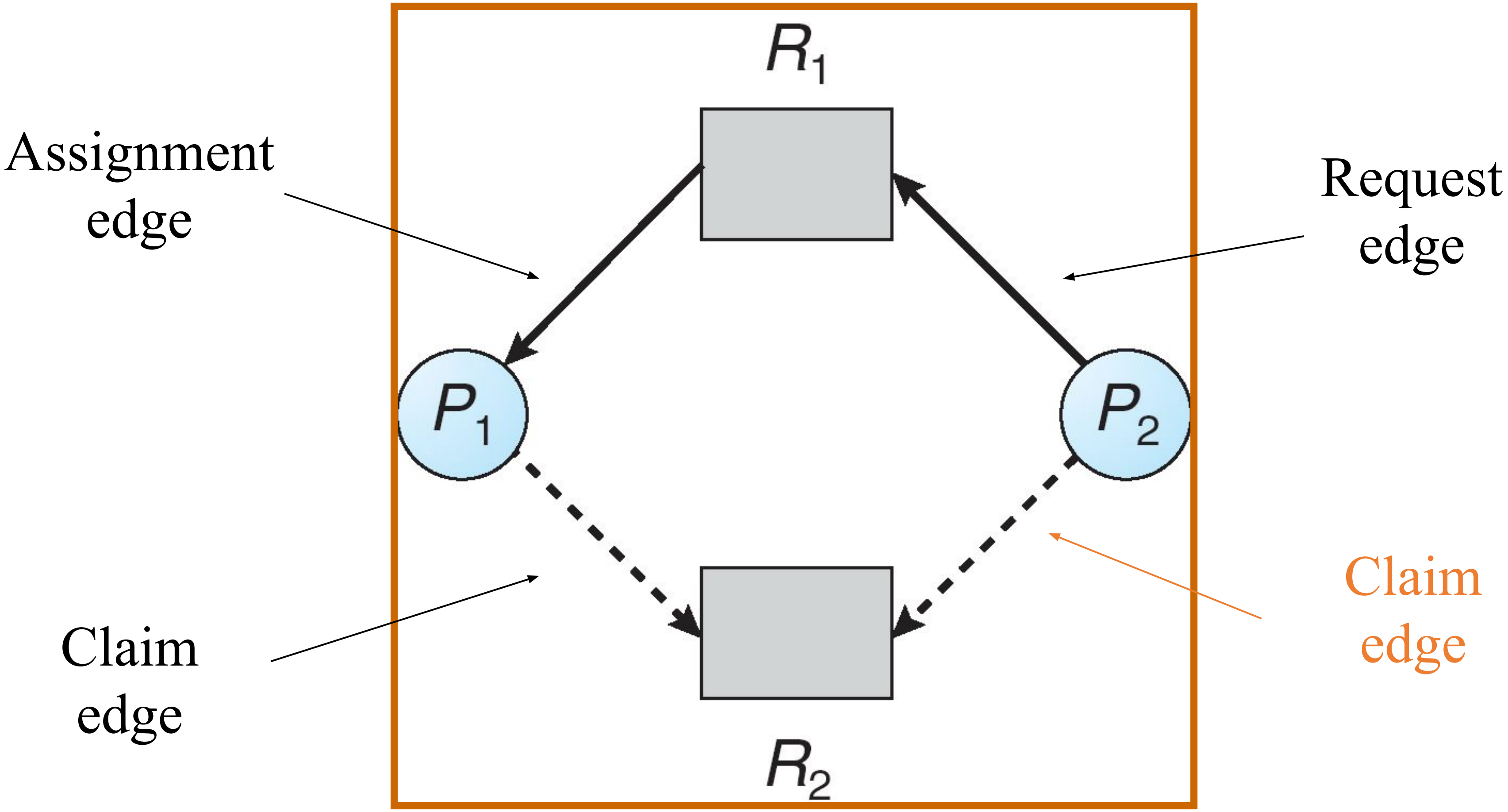
Deadlock Avoidance algorithms

- **resource-allocation graph**
  - For a <u>single</u> instance of each resource type.
    - Claim edge : process may request resource
- **Banker's algorithm**
  - For <u>multiple</u> instances of each resource type.

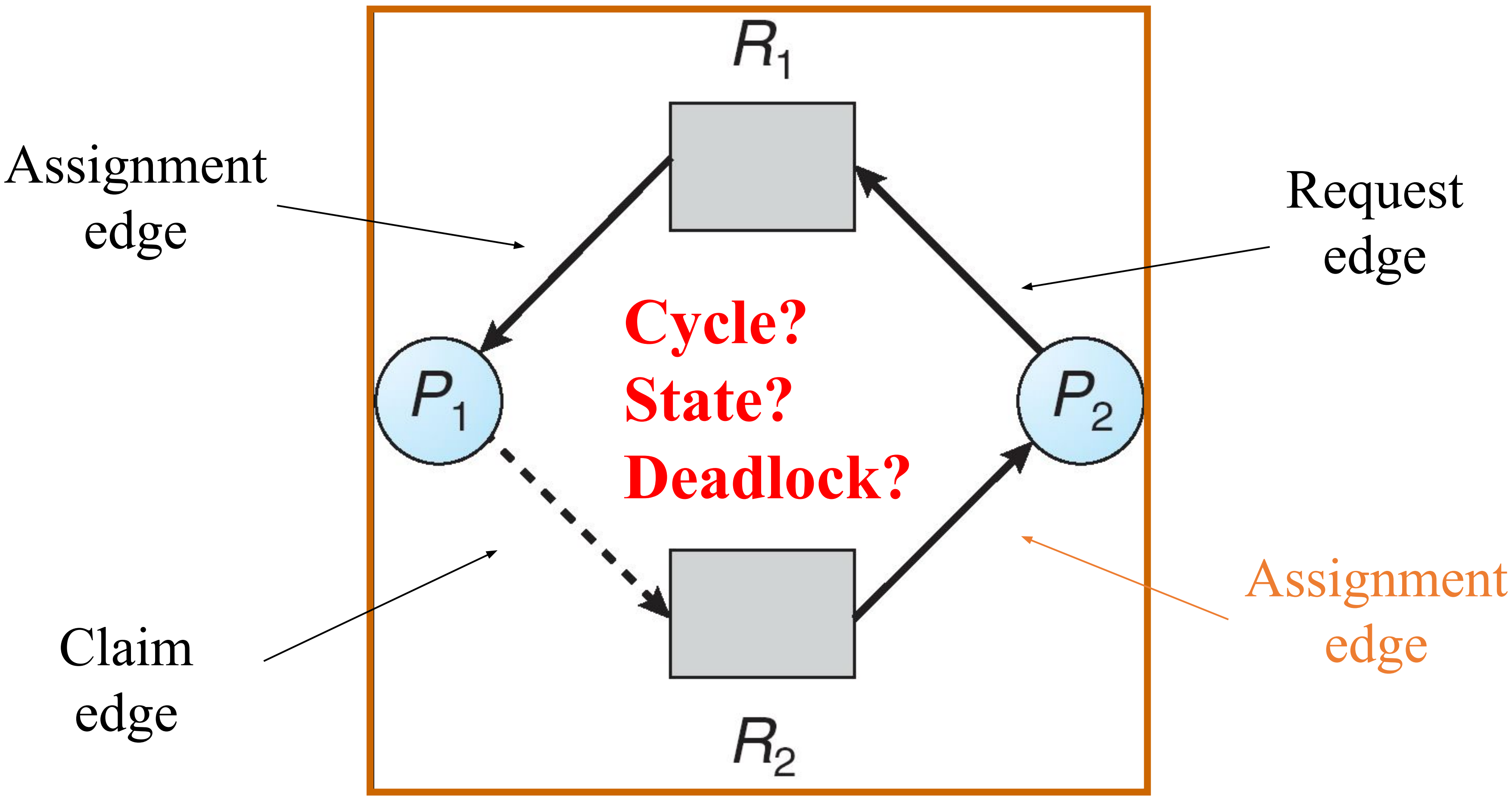
Resource-Allocation Graph Algorithm

- Introduce a new kind of edge called a <u>claim edge</u>
- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line
- A <u>claim edge</u> converts to a <u>request edge</u> when a process **requests** a resource
- A <u>request edge</u> converts to an <u>assignment edge</u> when the resource is **allocated** to the process
- When a resource is **released** by a process, an <u>assignment edge</u> reconverts to a <u>claim edge</u>
- Resources must be **claimed *a priori*** in the system

Resource-Allocation Graph Algorithm with Claim Edges
cycle detection algorithm



Unsafe State In Resource-Allocation Graph

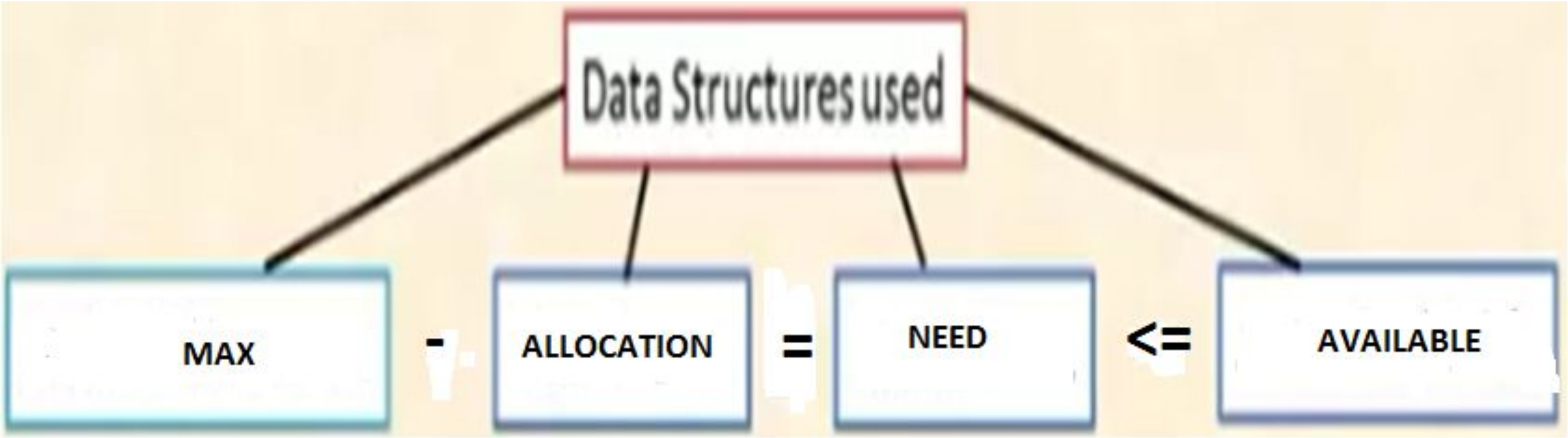


Banker’s Algorithm

Banker’s Algorithm

- Used when there exists **multiple** instances of a resource type
- 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

Data structures used



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]

1. Safety Algorithm

- 1. Let **Work and Finish** be vectors of length m and n respectively

  Initialize **Work=Available** and **Finish[i]=false** for i=0,1,...n-1

- 2. Find an index **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

  Goto step 2.

- 4. if Finish[i]==true for all i, then the system is in a safe

  state.

- Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	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	

- a.What will be the content of the Need matrix?
- b.Is the system in a safe state? If Yes, then what is the safe sequence?
- c.What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

Example 1

A=10,B=5,C=7

Max Needs = allocated + need

process	Allocation				MAX				Need			<=	Available			T/F
	A	B	C		A	B	C		A	B	C		A	B	C	
P0	0	1	0		7	5	3		7	4	3	<=	3	3	2	F
P1	2	0	0		3	2	2		1	2	2	<=	3	3	2	T
P2	3	0	2		9	0	2		6	0	0	<=	5	3	2	F
P3	2	1	1		2	2	2		0	1	1	<=	5	3	2	T
P4	0	0	2		4	3	3		4	3	1	<=	7	4	3	T
													7	4	5	

<=745 T

<=755 T

10 5 7

IS THE SYSTEM IN SAFE STATE?

If Need <= Available…….then new available
P0)743<=332….False… new available= 332
P1)122<=332….True… new available= 332+200=532
P2)600<=532…False… new available=532
P3)011<=532…True…new available=532+211=743
P4)431<=743…True…new available=743+002=745
P2)600<=745…True…new available=745+302=10 4 7
P0)743<=10 4 7…True…new available=10 4 7 + 010=10 5 7
Process sequence is P1,p3,p4,p2,P0

1)Identify Need Matrix
2)is the system in safe state?

Resource Request Algorithm

- Used for determining whether request can be safely granted
- Assumptions:
  - Request[i]be the request vector for process Pi
  - If $Request_i[j]==k$, then P[i] wants k instances of resource type R[j]
  - When a request for resources is made by P[i], the following actions are taken

- [P.T.O.]

Resource Request Algorithm ..CONTD..

1. If Request[i]<= Need[i], go to step 2, otherwise raise error since process has exceeded maximum claim.
2. If Request[i] <= Available, go to step 3, otherwise P[i] must wait
3. Have the system pretend to have allocated the requested resources to process P[i] by modifying the state as follows:
    1. Available = Available – Request[i]
    2. Allocation[i] = Allocation[i]+ Request[i]
    3. Need[i]= Need[i]– Request[i]

If the resulting resource-allocation state is safe, the transaction is completed and Pi is allocated to its resources

- If there is additional request of P1 for (102) can the request granted immediately? [PTO]

Additional Resource-Request

- Suppose P1 request one additional instance of A and two instances of C, i.e.Request[1]=(1,0,2)
- Decide request can be immediately granted?
- check request with need and available

i.e (1,0,2)<= (1,2,2)& (1,0,2) <= (3,3,2)which is true

- We then pretend that this request can be fulfilled and we arrive at new state

process	Allocation				Need			<=	Available					
	A	B	C		A	B	C		A	B	C			
P0	0	1	0		7	4	3	<=	**2**	**3**	**0**	**F**	**<=**	**745**
P1	**3**	**0**	**2**		**0**	**2**	**0**	<=	2	3	0	T		
P2	3	0	2		6	0	0	<=	5	3	2	**F**		755
P3	2	1	1		0	1	1	<=	5	3	2	T		
P4	0	0	2		4	3	1	<=	7	4	3	T		
									7	4	5			1057

Now for P1

Available=(332)-(102) = (230)
need=(122)-(102)=(020)
Allocation=(200)+(102)=(302)

safe sequence
P1,P3,P4,P0,P2

Problem with deadlock avoidance algorithm

- Overhead of maintaining entire information

Example 2

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

Banker's Algorithm

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

Available + Allocation = New Available

Banker's Algorithm

Available + Allocation = New Available

	Allocation	Max	Available		Need	
	A B C D	A B C D	A B C D		A B C D	
P0	0 0 1 2	0 0 1 2	1 5 2 0	P0	0 0 0 0	<= <1 5 2 0> = T
P1	1 0 0 0	1 7 5 0		P1	0 7 5 0	<= <1 5 3 2> =
P2	1 3 5 4	2 3 5 6		P2	1 0 0 2	
P3	0 6 3 2	0 6 5 2		P3	0 0 2 0	
P4	0 0 1 4	0 6 5 6		P4	0 6 4 2	

<1 5 2 0> + <0 0 1 2> = <1 5 3 2>

Banker's Algorithm

Available + Allocation = New Available

	Allocation A B C D	Max A B C D	Available A B C D		Need A B C D	
P0	0 0 1 2	0 0 1 2	1 5 2 0	P0	0 0 0 0	<= <1 5 2 0> = T
P1	1 0 0 0	1 7 5 0		P1	0 7 5 0	<= <1 5 3 2> = F
P2	1 3 5 4	2 3 5 6		P2	1 0 0 2	<= <1 5 3 2> = T
P3	0 6 3 2	0 6 5 2		P3	0 0 2 0	<= <2 8 8 6> = T
P4	0 0 1 4	0 6 5 6		P4	0 6 4 2	<= <2 14 11 8> = T

<1 5 2 0> + <0 0 1 2> = <1 5 3 2>
<1 5 3 2> + <1 3 5 4> = <2 8 8 6>
<2 8 8 6> + <0 6 3 2> = <2 14 11 8> New Available

Banker's Algorithm

Available + Allocation = New Available

	Allocation A B C D	Max A B C D	Available A B C D		Need A B C D	
P0	0 0 1 2	0 0 1 2	1 5 2 0	P0	0 0 0 0	<= <1 5 2 0> = T
P1	1 0 0 0	1 7 5 0		P1	0 7 5 0	<= <2 14 12 12> = T
P2	1 3 5 4	2 3 5 6		P2	1 0 0 2	<= <1 5 3 2> = T
P3	0 6 3 2	0 6 5 2		P3	0 0 2 0	<= <2 8 8 6> = T
P4	0 0 1 4	0 6 5 6		P4	0 6 4 2	<= <2 14 11 8> = T

<1 5 2 0> + <0 0 1 2> = <1 5 3 2>
<1 5 3 2> + <1 3 5 4> = <2 8 8 6>
<2 8 8 6> + <0 6 3 2> = <2 14 11 8>
<2 14 11 8> + <0 0 1 4> = <2 14 12 12> N.Available

The Safe Sequence:

P0 P2 P3 P4 P1

If there is additional request of P1 for (102) can the request granted immediately?

Deadlock Detection

Deadlock Detection and Recovery

- If we fail to employ preventive measures for deadlock prevention or avoidance, deadlock may occur.
- System periodically invokes the deadlock detection algorithm.
- If algorithm detects deadlock, it executes recovery algorithm.
- But data loss can be there during recovery.
- This algorithm can be used for single as well as multiple instances.

Deadlock Detection

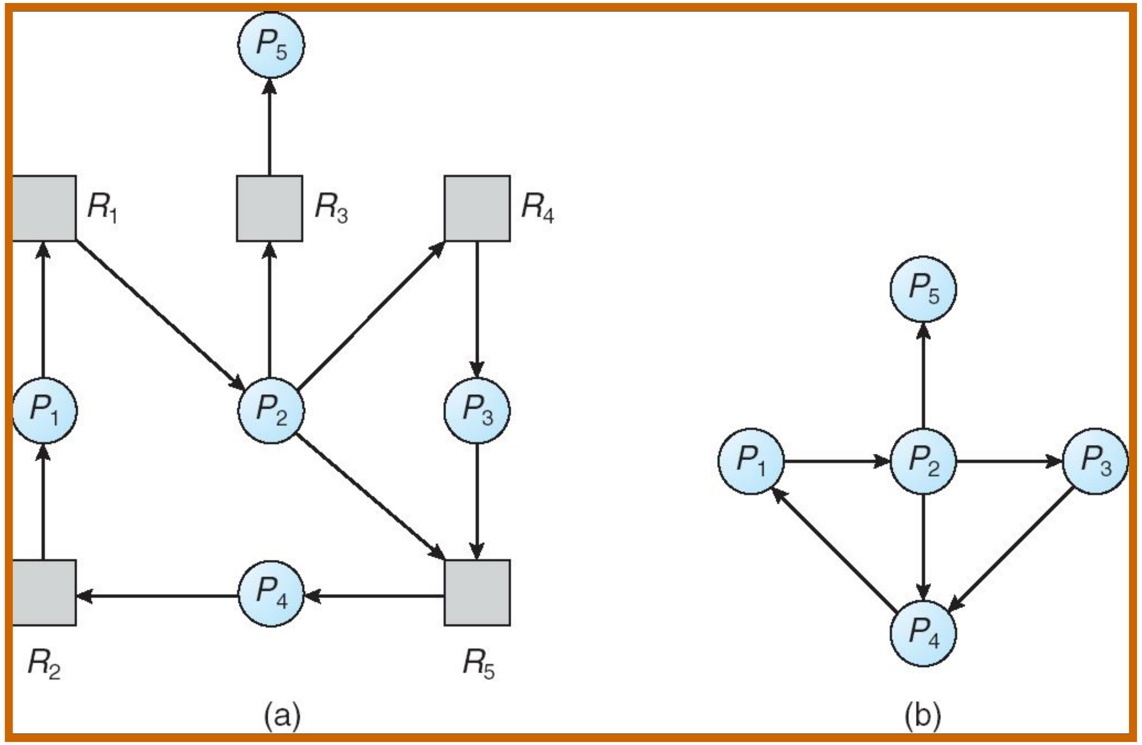
- For deadlock detection, the system must provide
  - An algorithm that examines the state of the system to **detect** whether a deadlock has occurred
  - And an algorithm to **recover** from the deadlock
- A detection-and-recovery scheme requires various kinds of overhead
  - Run-time costs of maintaining necessary information and executing the detection algorithm
  - Potential losses inherent in recovering from a deadlock

Single Instance of Each Resource Type

- Requires the creation and maintenance of a ***wait-for* graph**
  - Consists of a variant of the resource-allocation graph
  - The graph is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
  - Consequently; all 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

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Multiple Instances of a Resource Type

Required data structures:

- ***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

1. Initialize work[] = available[]
   For i = 1,2,...n, if allocation[i] != 0 then
   finish[i] = false; otherwise, finish[i] = true;

2. Find an i such that:
   finish[i] == false and request[i] <= work

   If no such i exists, go to step 4.

3. work = work + allocation[i]
   finish[i] = true
   goto step 2

4. if finish[i] == false for some i, then the system is in deadlock state.
   IF finish[i] == false, then process p[i] is deadlocked.

Deadlock detection

EXAMPLE

We have three resources, A, B, and C. A has 7 instances, B has 2 instances, and C has 6 instances. At this time, the allocation, etc. looks like this:

Is there a sequence that will allow deadlock to be avoided?

Is there more than one sequence that will work?

7 2 6

	←	Alloc	→	←	Req	→	←	Avail	→
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

After execution of algorithm it is found that
P0,P2,P3,P4,P1 results in Finish[i]==true for all i So system is not in deadlock state

Deadlock detection contd....

- Suppose P2 makes one additional request for C , can system in deadlock state?
- Then the state will be as follows:

	Request		
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

system will be in deadlock state
Although we claim the resources held by P0,
Number of resources available is not sufficient
To fulfil the request of other processes.
Deadlock exists, consisting of processes P1,P2,P3,P4.

Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on: **(frequency / resources not available / cpu usage fall below 40%)**
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?

- If the detection algorithm is invoked arbitrarily, there may be **many** cycles in the resource graph and so we would not be able to tell **which one** of the many deadlocked processes "caused" the deadlock
- If the detection algorithm is invoked for every resource request, such an action will incur a considerable **overhead** in computation time
- A less expensive alternative is to invoke the algorithm when CPU utilization drops **below 40%**, for example
  - This is based on the observation that a deadlock eventually cripples system throughput and causes CPU utilization to drop

Recovery From Deadlock

Recovery from Deadlock

- Two Approaches
  - Manual by operator / Automatic by system
    - Automatic
      - Process termination by Abort all or each process
      - Resource preemption

Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**
  - This approach will break the deadlock, but at great expense

- **Abort one process at a time until the deadlock cycle is** eliminated
  - This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked

- **Many factors may affect which process is chosen for termination**
  - What is the **priority** of the process?
  - **How long has the process run** so far and how much longer will the process need to run before completing its task?
  - How many and what **type of resources** has the process **used?**
  - How many **more resources** does the process **need** in order to finish its task?
  - How many **processes will need to be terminated**?
  - Is the process **interactive or batch**?

Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
  - **Selecting a victim** – Which resources and which processes are to be preempted?
  - **Rollback** – If we preempt a resource from a process, what should be done with that process?
    - Rollback the process to safe state and restart from that state.
  - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

COMBINED APPROACH TO DEADLOCK HANDLING:

- Type of resource may dictate best deadlock handling. Look at ease of implementation, and effect on performance.
- In other words, there is no one best technique.
- Cases include:

  Preemption for memory,

  Preallocation for swap space,

  Avoidance for devices (can extract Needs from process.)