

Deadlock in Operating System

CSEN3103: 3RD YEAR: SEC-A: Deadlocks

- **System Model**
- **Deadlock Characterization**
- **Methods for Handling Deadlocks**
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**
- **Recovery from Deadlock**
- **Combined Approach to Deadlock Handling**

Approaching Deadlock.....

Let us consider the following scenario:

A process P1 is in execution. It requests for a printer. The operating system allocates the printer to process P1. A context switch happens: Process P1 is now in possession of the printer, but it has relinquished the CPU. The CPU is allotted to another process, P2. The process P2 now requests for the printer....

- ❑ In this situation, for the moment, neither P1 nor P2 can progress, because P1 has the printer but needs the CPU to generate data.
- ❑ *On the other hand P2 has the CPU and can generate data, but needs the printer to print it.*
- ❑ *We can say, both processes P1 and P2 are blocked at the moment and are in a deadlock situation.*
- ❑ **Deadlock** is a situation in a computer system, where a set of processes is blocked, waiting for an event which will never happen.
- ❑ **Starvation** can be defined as a situation where a process waits for a resource, which is, for a prolonged time, made available to other processes.

The Deadlock Problem

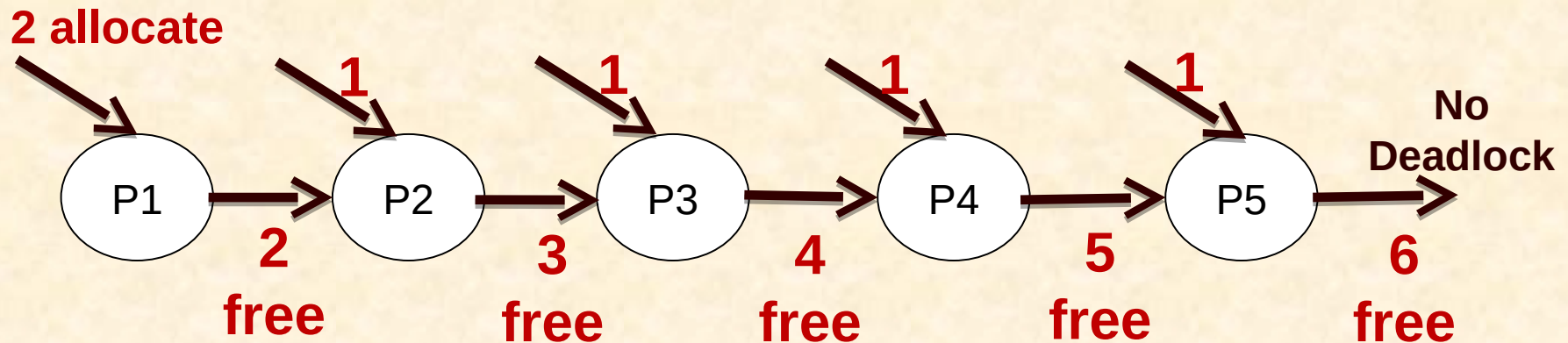
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. Couple of examples:
- Example -1
 - System has 2 printers.
 - P_1 and P_2 each hold one printer and each needs another one.
- Example-2
 - semaphores A and B , initialized to 1

P_1	P_2
<i>wait (A);</i>	<i>wait(B)</i>
<i>wait (B);</i>	<i>wait(A)</i>

DEADLOCK – Let's talk about Resource allocation to processes

- A computer system has 6 tape drives with n processes competing for them. Each process need 2 tape drives. The maximum value for n for which the system is guaranteed to be deadlock free is: a) 6 b) 5 c) 4 d) 3

5 processes can start with 1 tape drive (each). Processes are executing sequentially in the following way:



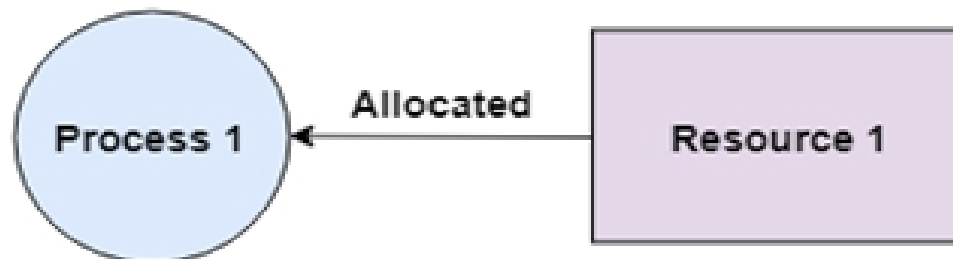
System Model

- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

Necessary conditions (Coffman Conditions) for a Deadlock to occur

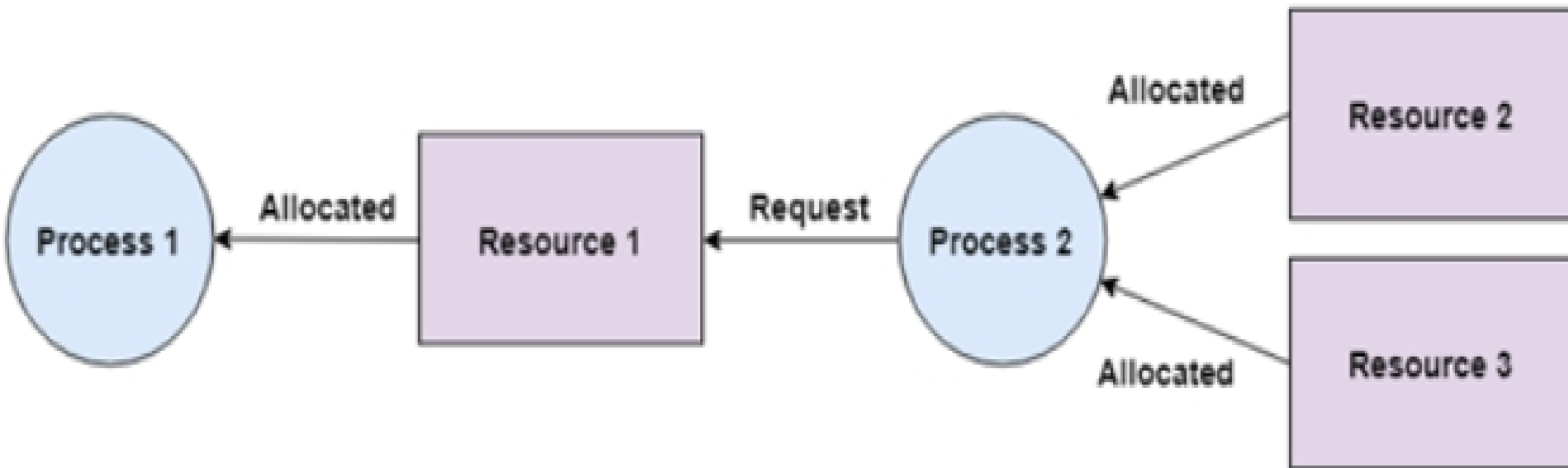
Deadlock can arise if the following four conditions hold simultaneously.

- **Mutual exclusion:** A major cause of deadlock is that some resources are not shareable. Therefore, only one process can access such a resource at a time. This is a case of mutual exclusion, where, competing processes mutually exclude each other for the usage of the resource. Example: the chopstick is a non-shareable resource and therefore no two philosophers can simultaneously use the same chopstick.
- There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



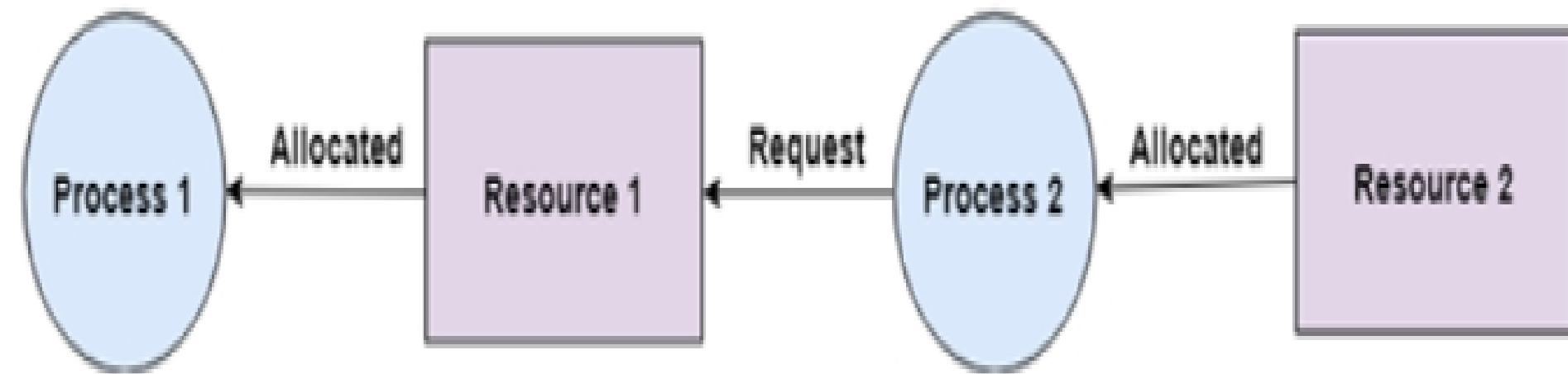
Necessary conditions (Coffman Conditions) for a Deadlock to occur

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes. This means a process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



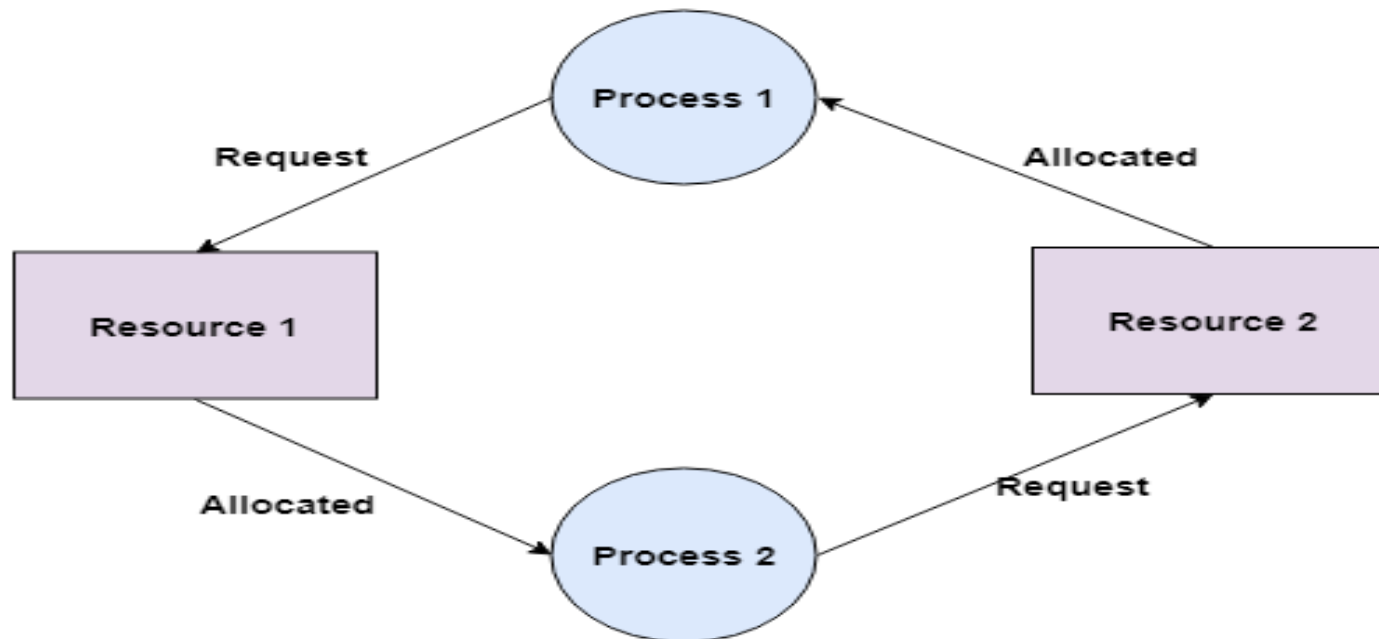
Necessary conditions (Coffman Conditions) for a Deadlock to occur

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task. A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Necessary conditions (Coffman Conditions) for a Deadlock to occur

Circular wait: there exists a set $\{P_0, P_1, \dots, P_n\}$ 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 .



Resource-Allocation Graph

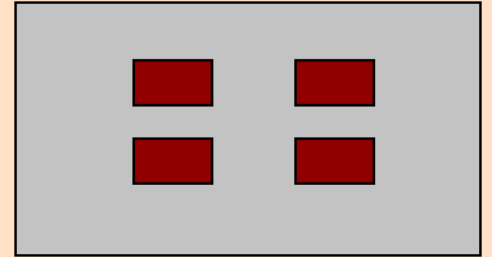
A set of vertices V and a set of edges E .

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

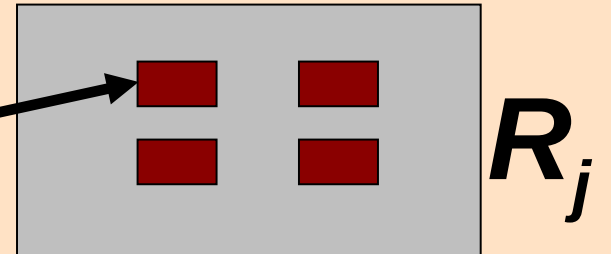
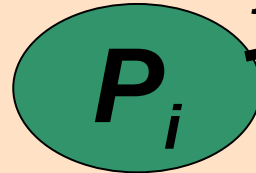
Resource-Allocation Graph (Cont.)

- Process 

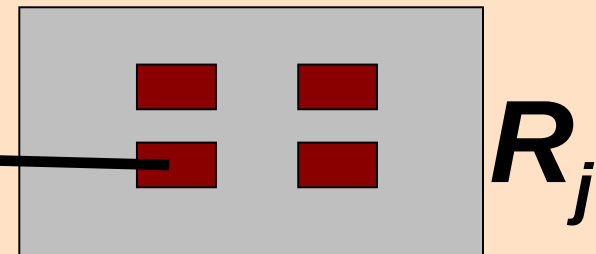
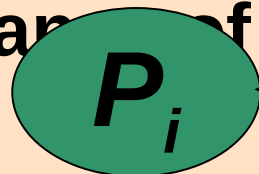
Resource Type with 4 instances



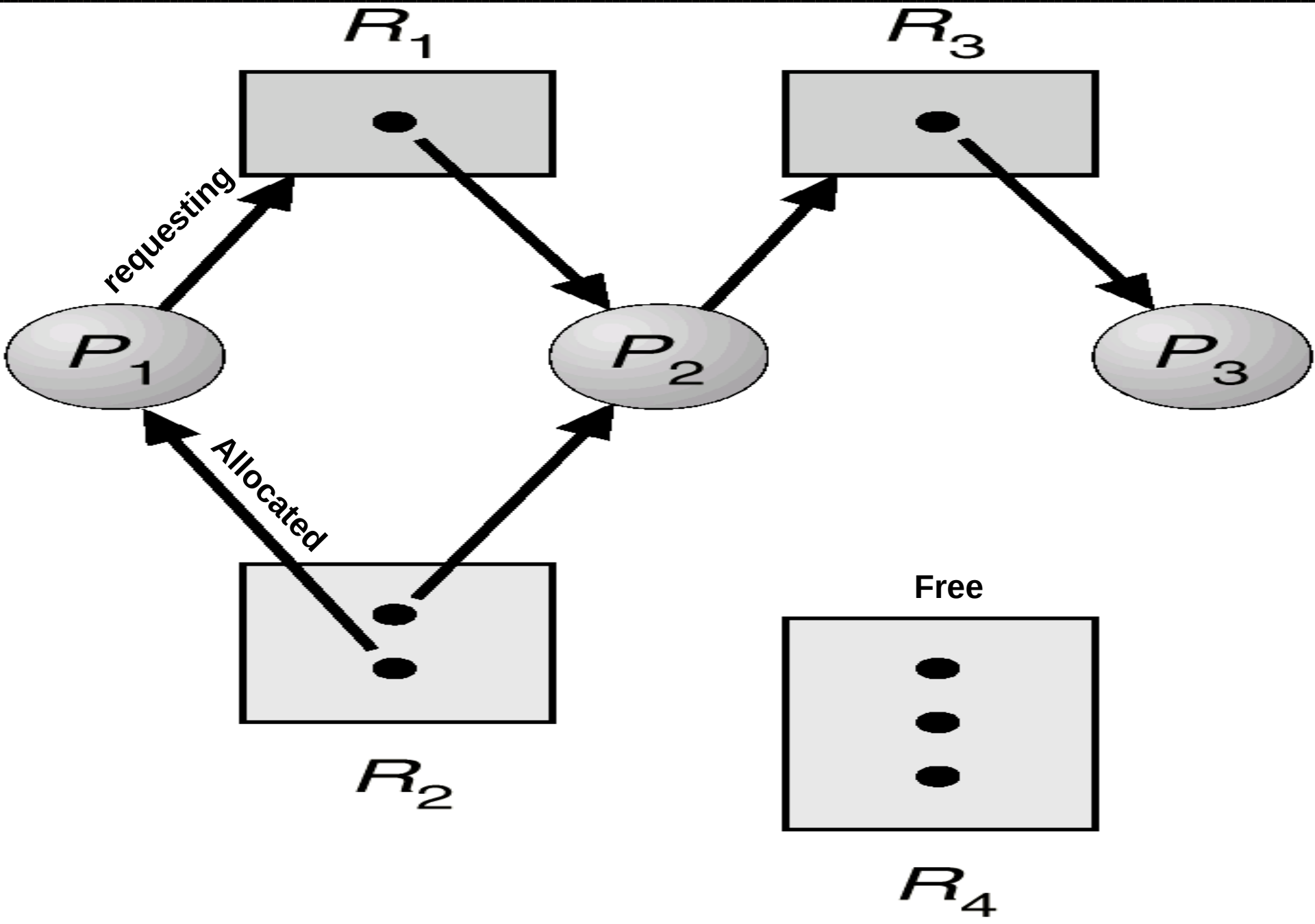
- P_i requests instance of R_j



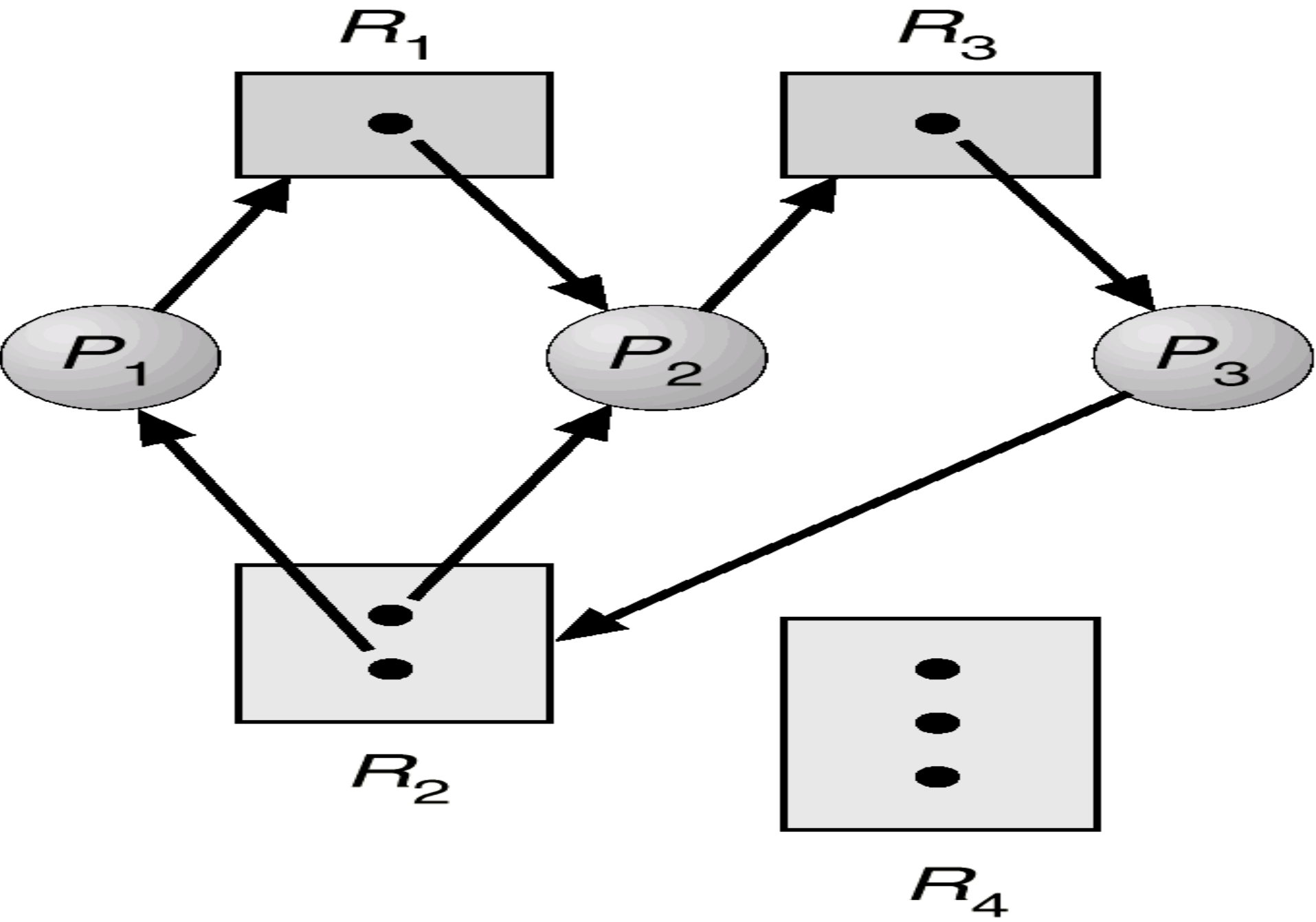
- P_i is holding an instance of R_j



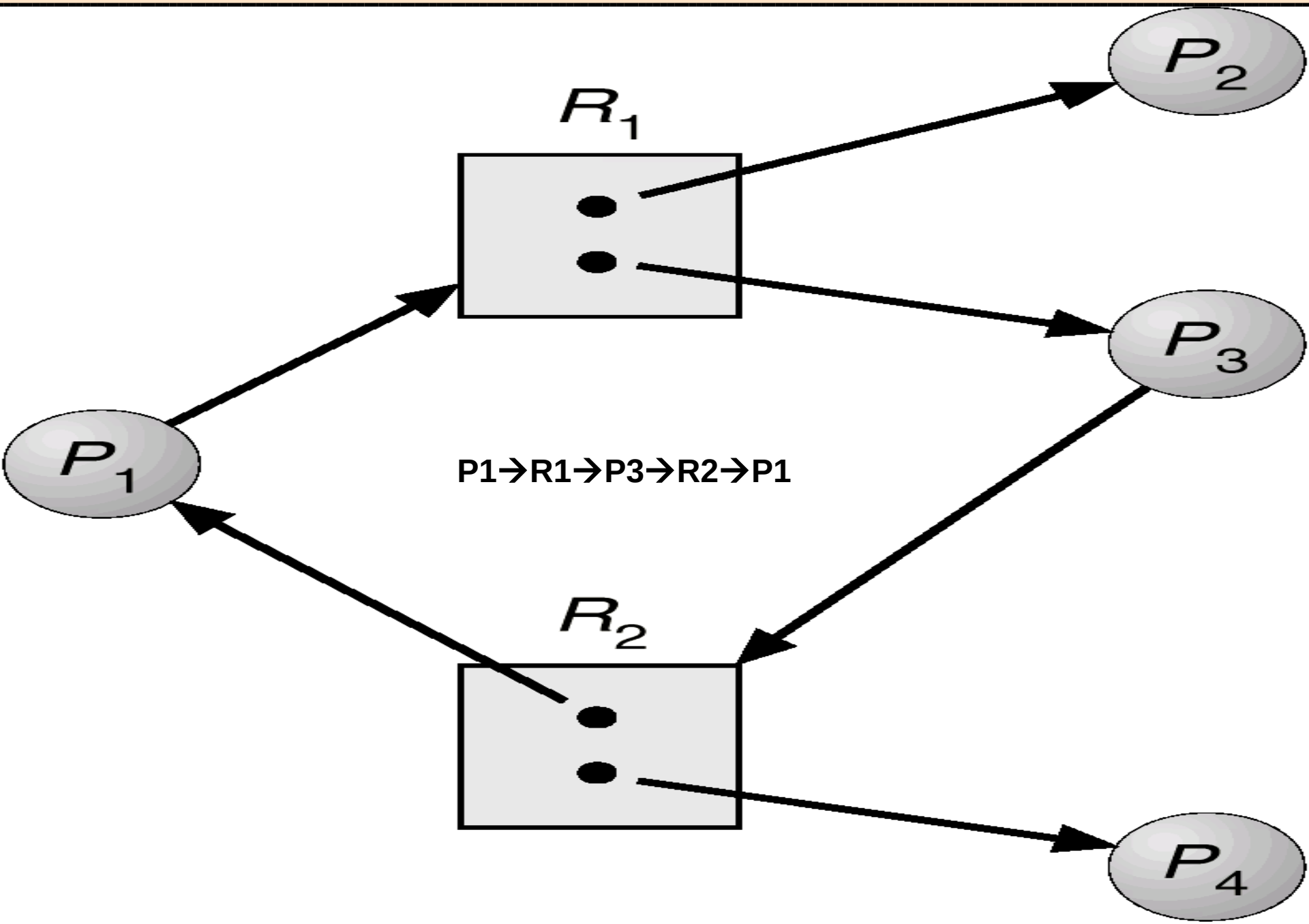
Example of a Resource Allocation Graph (Deadlock???)



Resource Allocation Graph (Deadlock ???)



Resource Allocation Graph (Deadlock ???)



Basic Facts

- Cycle is there: $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$
- But, there is no deadlock. Process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.
- 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.

Methods for Handling Deadlocks

- Ensure that the system will **never enter a deadlock state.**
- Allow the system to ***enter a deadlock state and then recover.***
- ***Ignore the problem and pretend that deadlocks never occur in the system;*** used by most operating systems, including UNIX.

Deadlock Strategies

- **The deadlock problem is handled by using the following strategies:**
 - **Deadlock detection**
 - **Deadlock prevention**
 - **Deadlock avoidance**

Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods:

1. All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
2. Resources can be preempted from some processes and given to others till the deadlock is resolved.

Deadlock Prevention

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources example. printer.
- **Hold and Wait** – must guarantee that whenever a *process requests a resource, it does not hold any other resources*.
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
 - 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.
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
 - *Adopt a policy wherein the resources allocated to a process can be preempted by the operating system. However, sometimes cost of preemption becomes very high. For instance, the resources like printers, plotters, etc., should not be forcefully taken by the OS from an executing process. In contrast, preempting a CPU for a short duration is tolerable.*
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

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

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

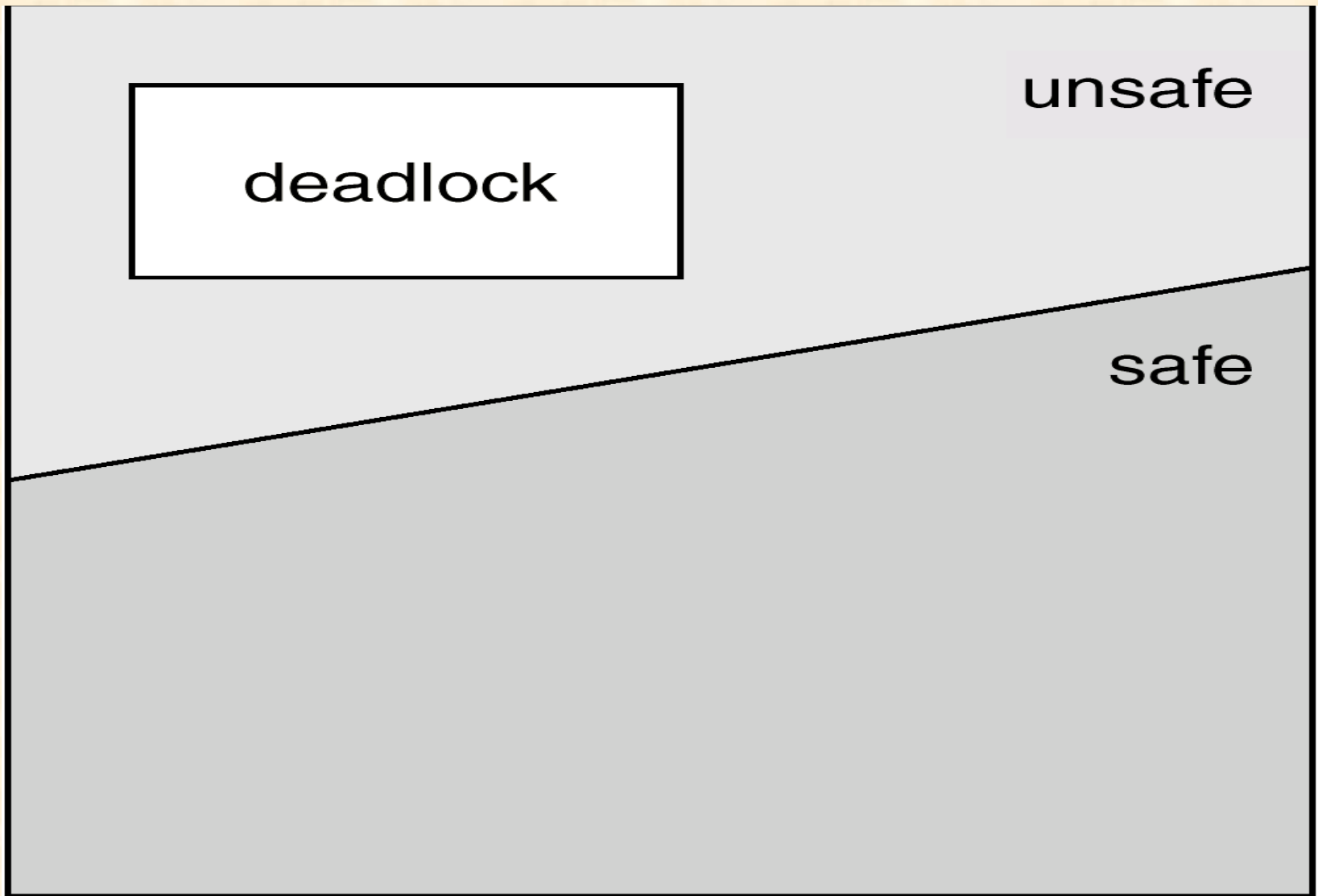
Safe State

- *When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.*
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that P_i can still request can be satisfied by **currently available resources + resources held by all the P_j , with $j < i$.**
 - If P_i 's 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, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- **If a system is in safe state \Rightarrow no deadlocks.**
- **If a system is in unsafe state \Rightarrow possibility of deadlock.**
- **Avoidance \Rightarrow ensure that a system will never enter an unsafe state.**

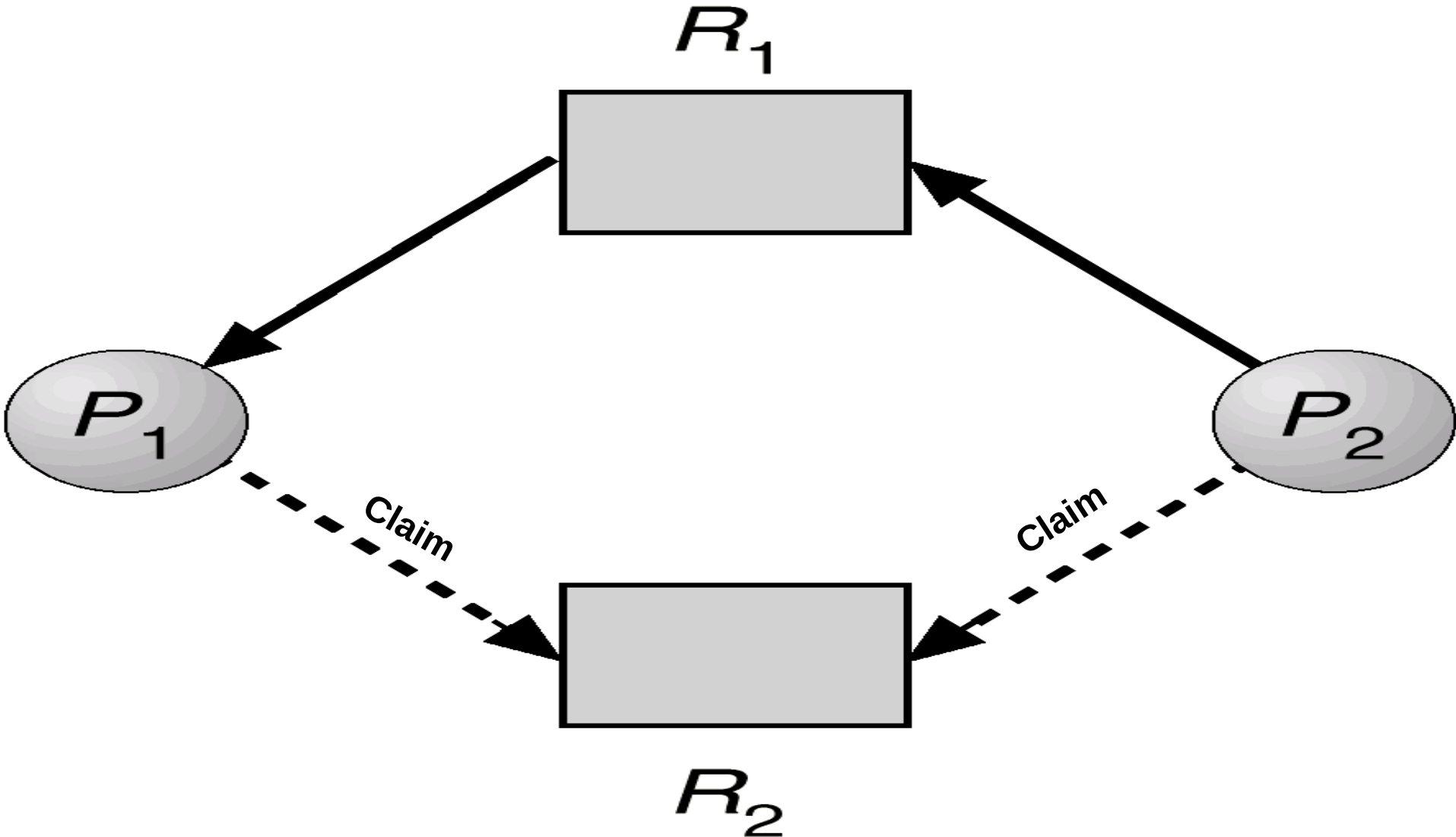
Safe, unsafe , deadlock state spaces



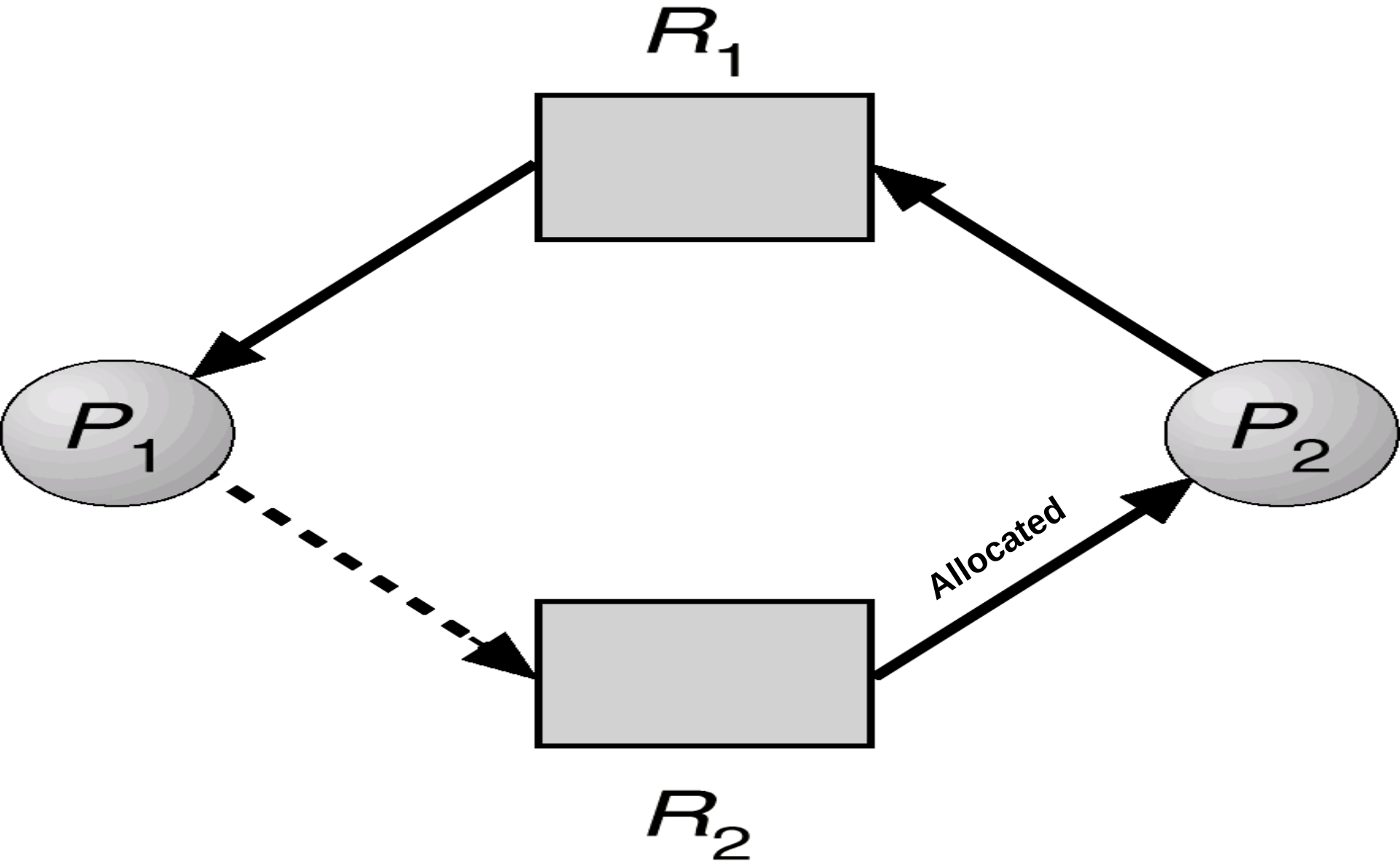
Resource-Allocation Graph Algorithm

- **Claim edge** $P_i \text{ --- } R_j$ indicated that process P_i *may request* resource R_j ; represented by a dashed line.
- **Claim edge converts to request edge** when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In A Resource-Allocation Graph



Banker's Algorithm (the deadlock avoidance algorithm)

- **Multiple instances.**
- **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 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:** $n \times m$ matrix. If $Max [i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j .
- **Need:** $n \times m$ 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

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

Work := *Available*

Finish [*i*] = *false* for *i* = 1, 2, 3, ..., *n*.

2. Find an *i* such that both:

(a) *Finish* [*i*] = *false*

(b) $Need_i \leq Work$

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*] = *true* for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

$Request_i$ = 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:

$Available := Available - Request_i;$

$Allocation_i := Allocation_i + Request_i;$

$Need_i := Need_i - Request_i;$

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

Example of Banker's Algorithm

Q : Suppose we have two resources, A, and B. A has 6 instances and B has 3 instances. Can the system execute the following processes without deadlock occurring?

**Already
Allocated**

Process	Allocate		Maximum need	
	A	B	A	B
P1	1	1	2	2
P2	1	0	4	2
P3	1	0	3	2
P4	0	1	1	1
P5	2	1	6	3

**Sum of max. need
per row.**

4
6
5
2
9

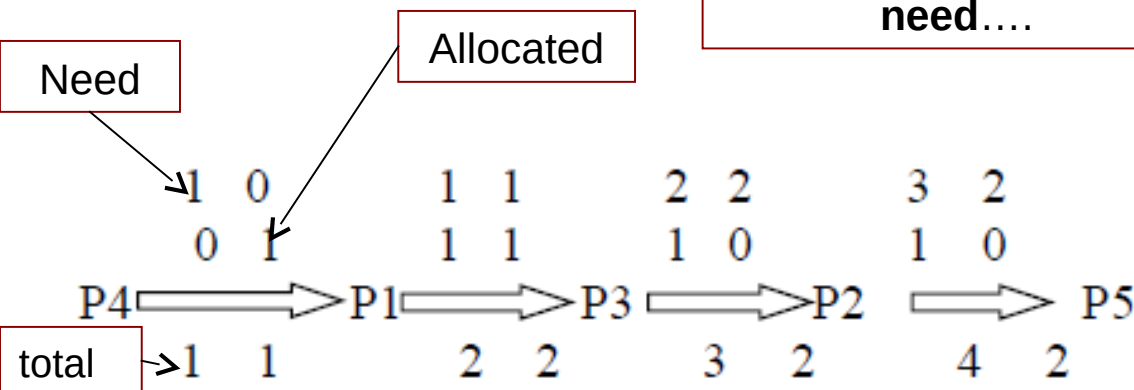
Min.

Answer

Available: A \rightarrow 6 instances,
Already allocated \rightarrow 5;
available $\rightarrow 6 - 5 = 1$; For B \rightarrow
0, no instances are available

Total freed up resources
can be allocated to
awaiting processes..
Starting from P4 \rightarrow P1
 \rightarrow P3 \rightarrow P2 \rightarrow P5....
Total can fulfill the
need....

Process	Need	
	A	B
P1	1	1
P2	3	2
P3	2	2
P4	1	0
P5	4	2



We can execute the processes in the sequence $\langle P4, P1, P3, P2, P5 \rangle$ without deadlock.

Q : Consider we have five processes P0, P1, . . . P5 and three resources A, B, and C. Is the executing the following processes in the safe state?

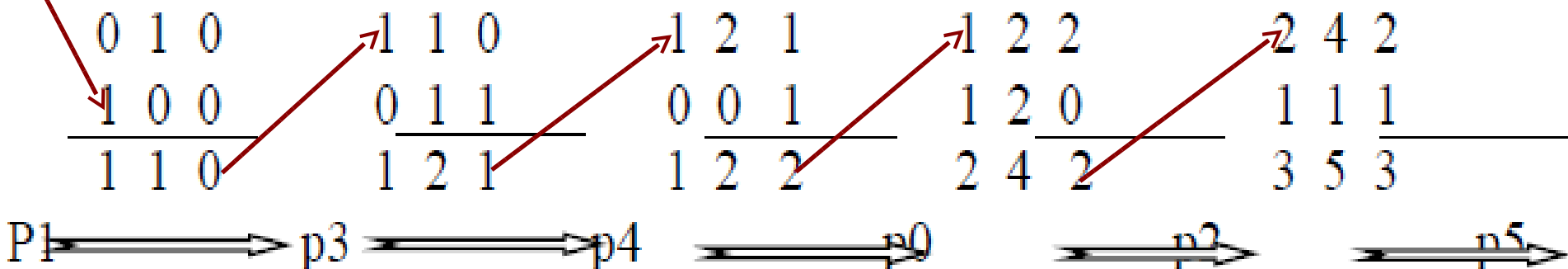
Process	Allocation			Maximum need			Available		
	A	B	C	A	B	C	A	B	C
P0	1	2	0	2	2	2	0	1	0
P1	1	0	0	1	1	0			
P2	1	1	1	1	4	3			
P3	0	1	1	1	1	1			
P4	0	0	1	1	2	2			
P5	1	0	0	1	5	1			

Answer

We find the resources that need for each process

Process	Need		
P0	1	0	2
P1	0	1	0
P2	0	3	2
P3	1	0	0
P4	1	2	1
P5	0	5	1

Allocation



Q. : Suppose we have five processes and three resources, A, B, and C. A has 2 instances, B has 5 instances and C has 4 instances. Can the system execute the following processes without deadlock occurring, where we have the following?

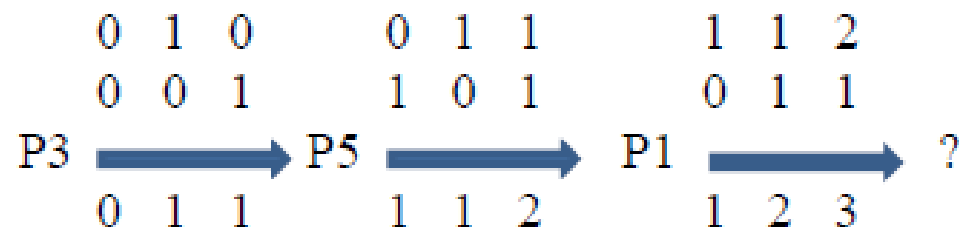
Process	Maximum need			Allocation		
	A	B	C	A	B	C
P1	1	2	3	0	1	1
P2	2	2	0	0	1	0
P3	0	1	1	0	0	1
P4	3	5	3	1	2	1
P5	1	1	2	1	0	1

Answer

The available is A=0, B=1, C=0.

The current need is

Process	Current need		
	A	B	C
P1	1	1	2
P2	2	1	0
P3	0	1	0
P4	2	3	2
P5	0	1	1



The deadlock is occurred since the available resources is less than the needs of P2 and P4.

Example of Banker's Algorithm

- 5 processes P_0 through P_4 ; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- **Need** 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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example (Cont.): P_1 request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$).

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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?

START

