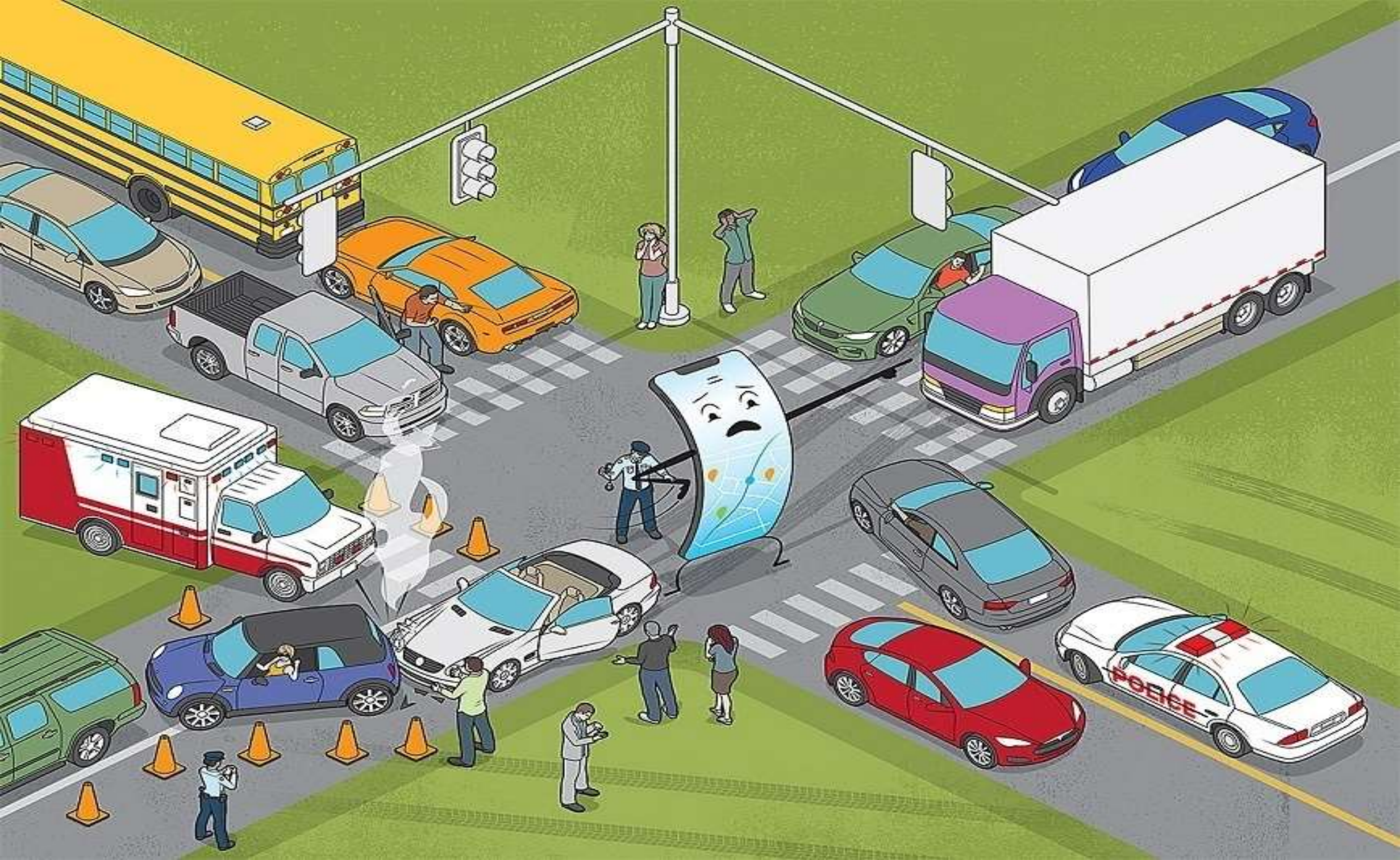
SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 OF THE UGC ACT, 1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

CSE308 Operating Systems

Deadlocks

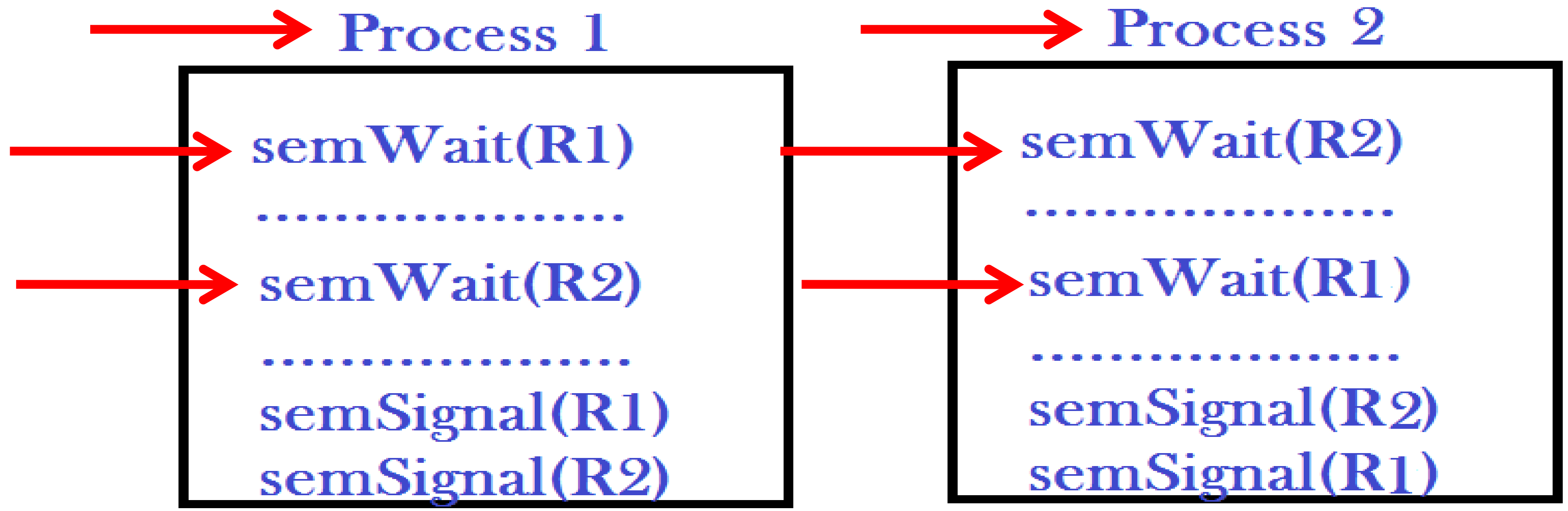
Dr S.Rajarajan
SASTRA


POLICE

What is a deadlock ?

- In a multiprogramming environment, several processes may **compete** for a **finite number of resources.**

Deadlocked



Process 1

```
semWait(R1)
....................
semWait(R2)
....................
semSignal(R1)
semSignal(R2)
```

Process 2

```
semWait(R2)
....................
semWait(R1)
....................
semSignal(R2)
semSignal(R1)
```

Blocked **Preempted**

Blocked

Scenarios that will not result in Deadlock

1. Both process make the resource requests in the same order

Process 1

```
semWait(R1)
......................
semWait(R2)
......................
semSignal(R1)
semSignal(R2)
```

Process 2

```
semWait(R1)
......................
semWait(R2)
......................
semSignal(R1)
semSignal(R2)
```

2. If processes execute sequentially without interruption

Process 1

```
semWait(R1)
....................
semWait(R2)
....................
semSignal(R1)
semSignal(R2)
```

Process 2

```
semWait(R2)
....................
semWait(R1)
....................
semSignal(R2)
semSignal(R1)
```

3. If either R1 or R2 is available in multiple instances/quantities

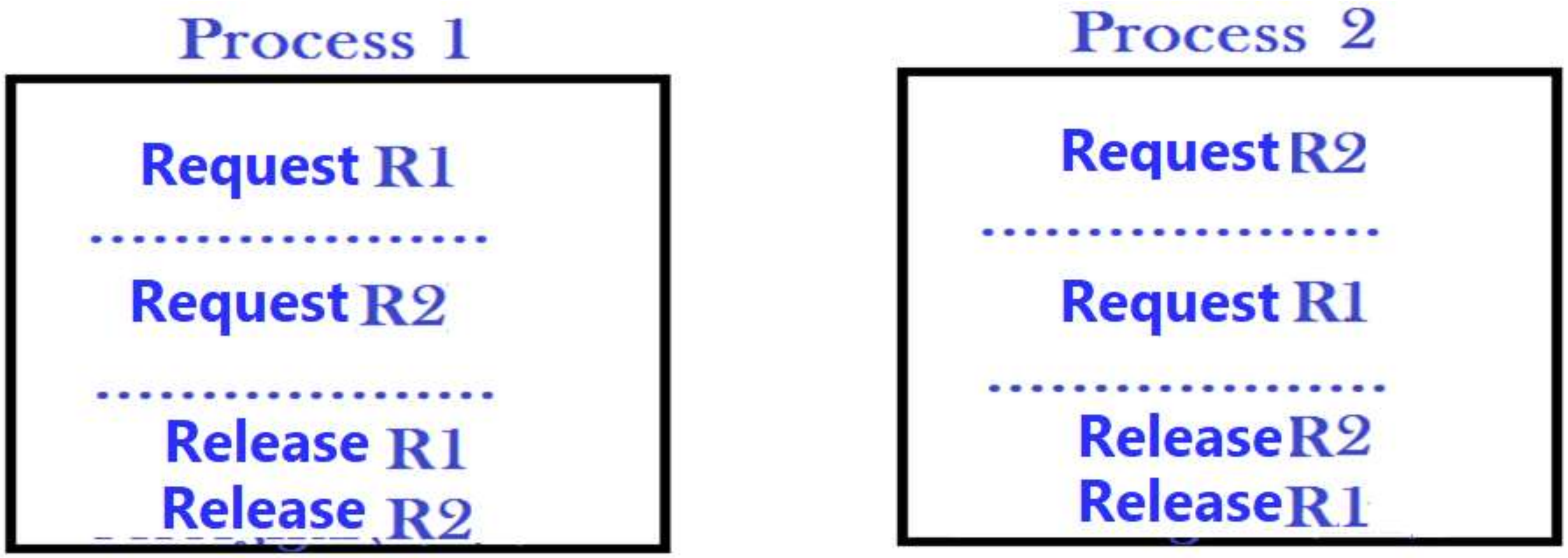
Process 1

```
semWait(R1)
.....................
semWait(R2)
.....................
semSignal(R1)
semSignal(R2)
```

Process 2

```
semWait(R2)
.....................
semWait(R1)
.....................
semSignal(R2)
semSignal(R1)
```

4. If resources can be accessed without mutual exclusion



System Model

- **Finite number of resources** to be **distributed** among a number of competing processes.
- Resources are of **several types** (or **classes**), each consisting of some number of identical **instances**.
  - **CPU** cycles, **files, Memory, Sync locks, Messages** and **I/O devices** are examples of resource types.
- If a system has **two CPUs**, then the resource type *CPU has **two*** **instances**.

Assumptions

- If a process **requests an instance** of a resource type, the allocation of **any instance** of the type should satisfy the request.
- If it does not, then the instances are **not identical**.
- A process must **request a resource before using** it and must **release the resource** after using it.
- A process **may request as many resources** as it requires to carry out its designated task.
- But the number of resources requested **can not exceed the total number of resources** available in the system.

- A process may utilize a resource in the following sequence:
- **1. Request.** The process requests the resource. If the request cannot be granted immediately (then the requesting process may be blocked.
- **2. Use.** The process can operate on the resource.
- **3. Release.** The process releases the resource.
- The request and release of resources may be **system calls**.
- Examples are the **request() and release() device**, **open()** and **close()** file, and **allocate()** and **free()** memory system calls.

- A set of processes are in a deadlocked state when every process in the set is **waiting for an event** that **can be caused only by another process** in the set.
- The resources may be
  - **physical resources (reusable resource)** (for example, printers, tape drives, memory space, and CPU cycles)
  - ‘or **logical resources(consumable resouces)** (for example, messages, signals, semaphores, mutex locks, and files).
- Deadlocks may also involve different resource types.

Messages and signals too may cause deadlocks

Blocking send & Blocking receive

- **Process A**
- receive(msg1, process B)
- Send(msg2, process B)

- **Process B**
- receive(msg2, process A)
- Send(msg1, process A)

DEADLOCK CHARACTERIZATION

Necessary Conditions for deadlock to occur

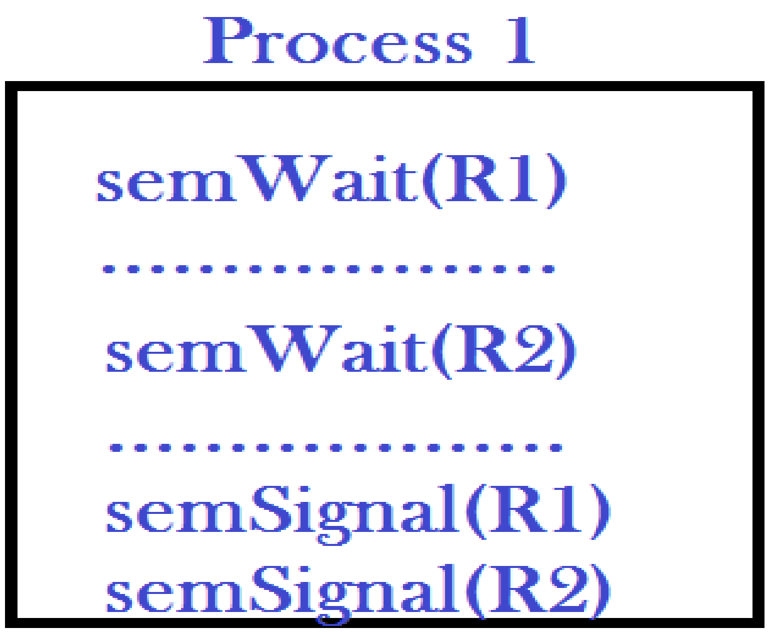
- A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **1. Mutual exclusion.** At least one resource must be held in a **non-sharable mode**; that is, **only one process at a time** can use the resource.
- 2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

- **3. 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.
- **4. Circular wait.** A set {*P0, P1, ..., Pn} of waiting processes 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.*
- **All four conditions must hold for a deadlock to occur.**

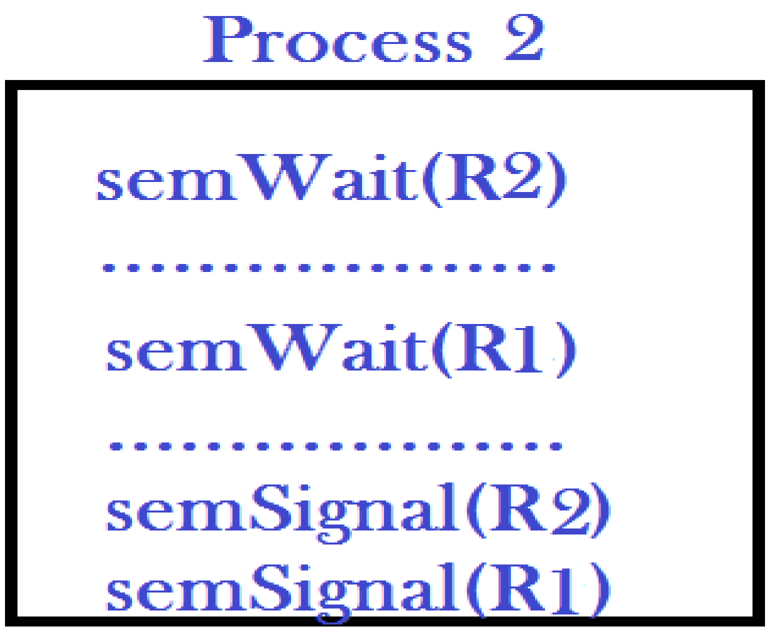
- In a multiprogramming environment, several processes may **compete** for a **finite number of resources.**

Mutual exclusion Hold and wait No preemption Circular wait



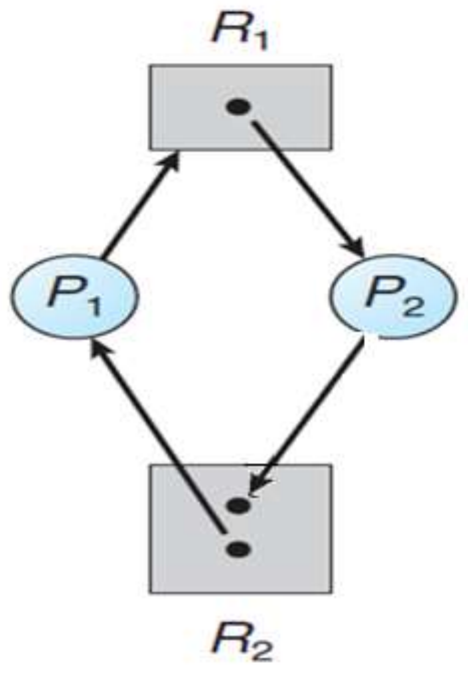
Process 1

```
semWait(R1)
....................
semWait(R2)
....................
semSignal(R1)
semSignal(R2)
```



Process 2

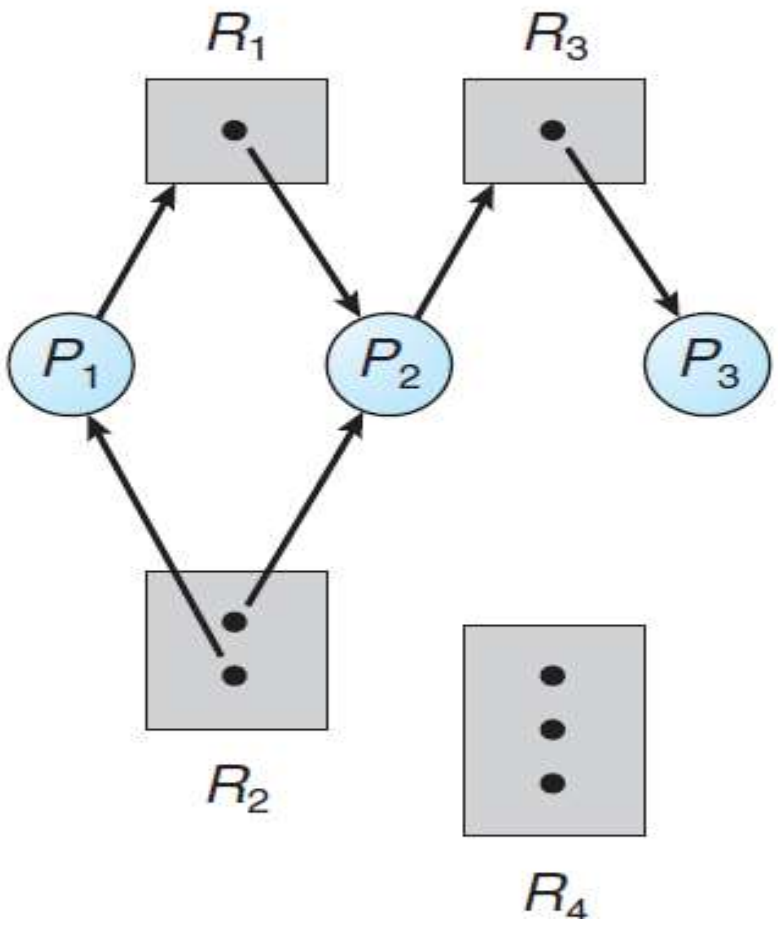
```
semWait(R2)
....................
semWait(R1)
....................
semSignal(R2)
semSignal(R1)
```

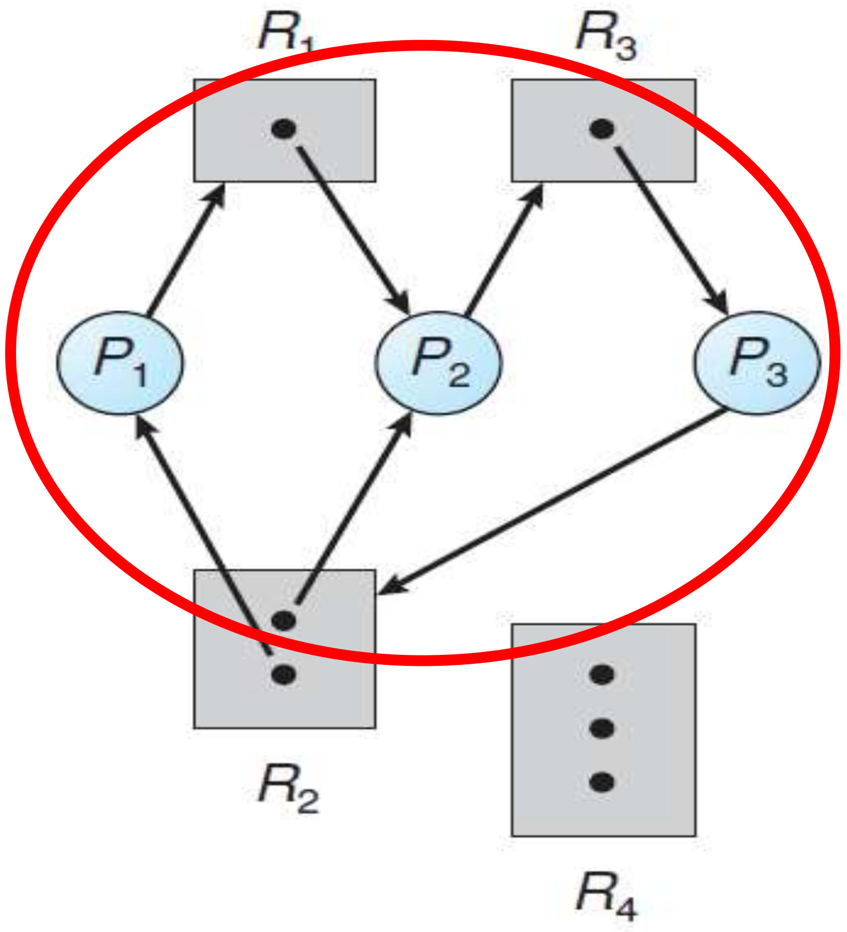


Resource-Allocation Graph

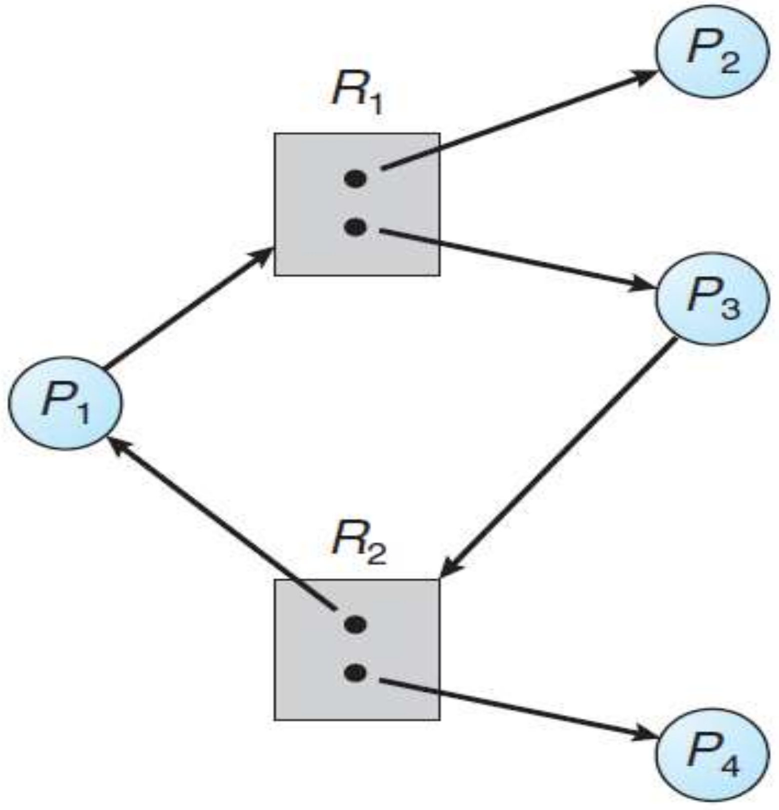
- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph.**
- A graph consists of a set of vertices *V* and a set of edges *E.*
- A directed edge from process *Pi to resource type Rj is denoted by* ***Pi → Rj*** *;* it signifies that process *Pi has requested an instance of resource type Rj and* is currently **waiting for that resource**.
- A directed edge from resource type *Rj* to process *Pi is denoted by* ***Rj → Pi****; it signifies that an instance of resource* type *Rj has been* ***allocated to process Pi***

- 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 **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.


R_1
R_3
P_1
P_2
P_3
R_2
R_4



Processes *P1, P2, and P3 are deadlocked*



P1 & P3 are in a cycle but no deadlock.

Methods for Handling Deadlocks

- We can deal with the deadlock problem in one of three ways:
  - **Prevention:** We can design the OS by eliminating the causes of deadlock
  - **Avoidance:** We may develop a protocol for resource allocation ensuring that the system will never enter a deadlocked state.
  - **Detection:** Let deadlocks happen, but detect and recover from it.
  - **Ignore:** We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- The fourth solution is the one used by most operating systems, including **Linux and Windows.**
- It is **up to the application developer** to write programs that **handle deadlocks**

Deadlock Prevention

- We saw four conditions as necessary conditions for deadlock to occur.
- We may **attempt to prevent any one** of the four conditions from occurring and thereby deadlock will be prevented.
- So the aim of deadlock prevention is **to stop mutual exclusion or hold and wait or no preemption or circular wait.**

1. Mutual Exclusion

- The mutual exclusion condition **must hold**.
- Non-sharable resources need mutual exclusion to be imposed.
- For example, the buffer in the producer-consumer problem can not be accessed by both producer and consumer.
- So we **can not prevent** mutual exclusion.

2. Hold and Wait

- There are two options:
- 1). Each process **must request and be allocated all its resources** before it begins execution (like forks in case of dinning philosopher problem).
- 2). A process to **request resources only when it has none**. Before it can request any additional resources, it **must release** the resource that it is currently allocated.

Disadvantages.

- First, **resource utilization may be low**, since resources may be allocated but unused for a long period.
- Second, **starvation is possible**. A process that needs several resources may have to wait indefinitely, because at least one of the resources that it needs is a allocated to some other process.
- Third, sometimes the process may have to use **two resources simultaneously**, for example to copy a file content into another.

3. No Preemption

- The third necessary condition for deadlocks is that there be **no preemption of resources** that have already been allocated.
- If a process is **holding some resources** and requests another **resource that cannot be immediately allocated** to it then all resources the process is currently holding are preempted.
- The **preempted resources are added** to the list of resources for which the process is waiting.
- This protocol is often applied to resources whose **state can be easily saved and restored later**, such as CPU registers and memory space.

4. Circular Wait

- 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.
- We let *R = {R1, R2, ..., Rm} be the set of resource types.*
- We assign to each resource type a unique integer number in sequence (e.g 1,2,3,4 …)

$$F(\text{tape drive}) = 1$$
$$F(\text{disk drive}) = 5$$
$$F(\text{printer}) = 12$$

- 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.***
- *But after that, the process can request instances of resource type* ***Rj*** only if ***F(Rj) > F(Ri).***

Process P1	***Process P2***
SemWait(R1)	*SemWait(R2)*
SemWait(R2)	*SemWait(R1)*

DEADLOCK AVOIDANCE USING BANKER’S ALGORITHM

Deadlock Avoidance

- An alternative method for handling deadlocks is to **regulate the way resources are allocated** to processes.
- When a **new process** enters the system, it **must declare the maximum number of instances** of each resource type that it may need
- This number may **not exceed the total number of resources** in the system.
- When a **user requests a set of resources**, the system must determine **whether the allocation** of these resources will leave the system in a **<u>safe state</u>**.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources

- A deadlock-avoidance algorithm **dynamically examines** the resource-allocation state to ensure that a **circular-wait condition** can never occur.
- The resource allocation ***state is defined by the number of available and allocated resources*** and the maximum demands of the processes.
- The state of the system is either **safe** or **unsafe**.

Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with **only one instance of each resource type**, we can use a variant of the resource-allocation graph.
- In addition to the **request and assignment edges** already described, we introduce a new type of edge, called a **claim edge.**
- A claim edge *Pi* → *Rj indicates that process* ***Pi may request resource Rj*** *at* some time in the future.

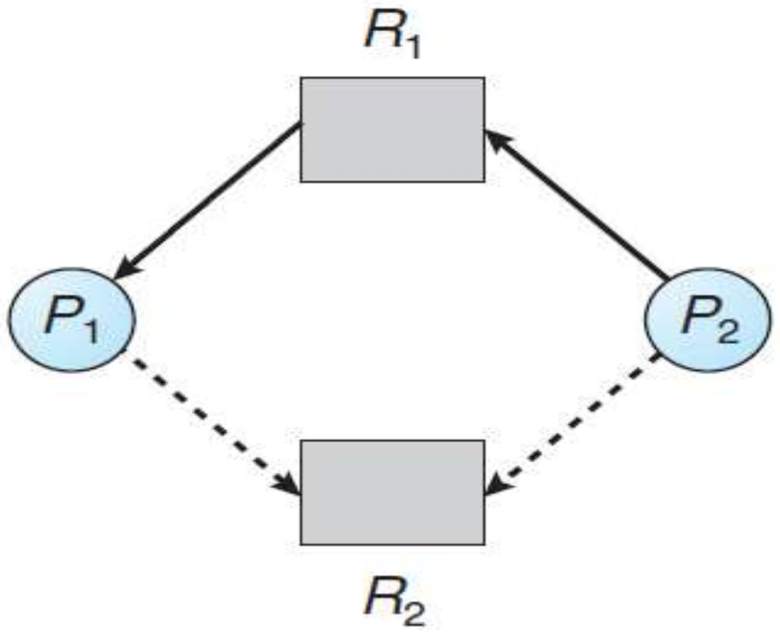
• This edge resembles a request edge in direction but is represented in the graph by **a dashed line**.

• When process *Pi requests resource Rj, the claim edge Pi → Rj is* ***converted to a request edge****.*

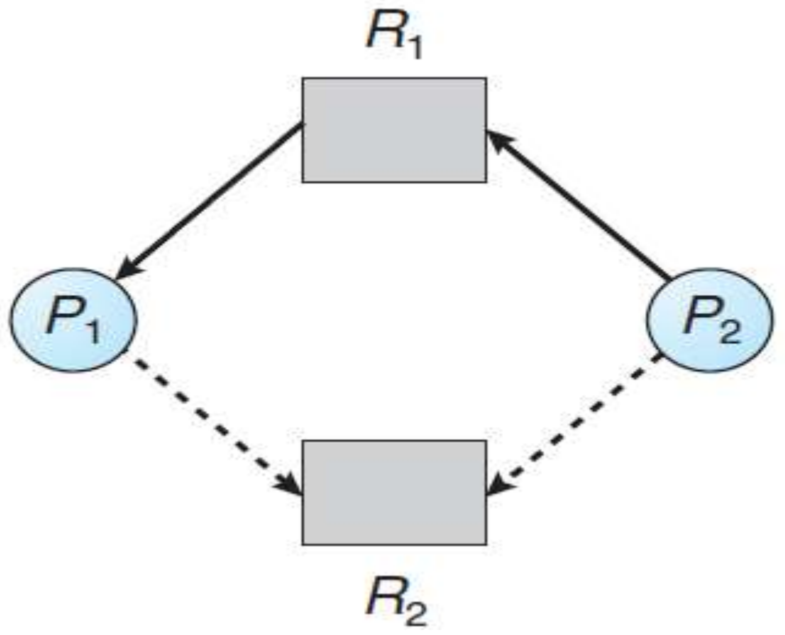
•*Similarly, when a* **resource *Rj is released by Pi****, the assignment edge* ***Rj → Pi*** *is* ***reconverted to a*** **claim edge *Pi → Rj*** *.*

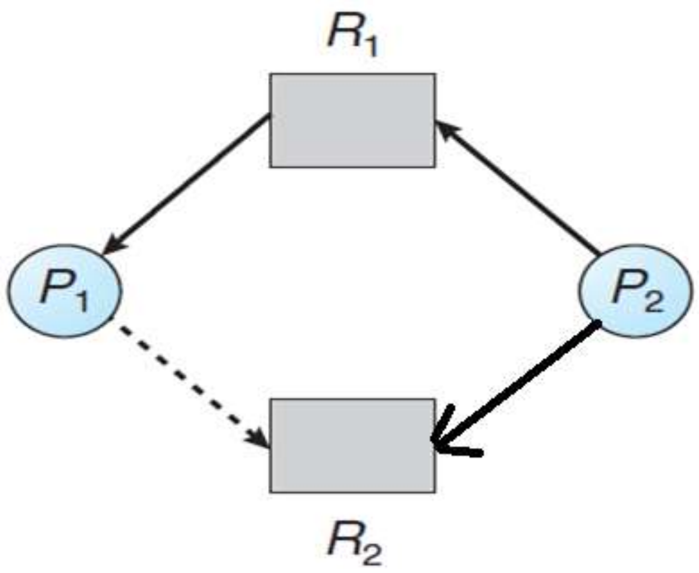
•Before process *Pi starts executing,* ***all its claim edges must already appear*** *in* the resource-allocation graph.

•We can relax this condition by allowing a claim edge *Pi → Rj to be added to the graph only if all the edges associated with* process *Pi are claim edges.*

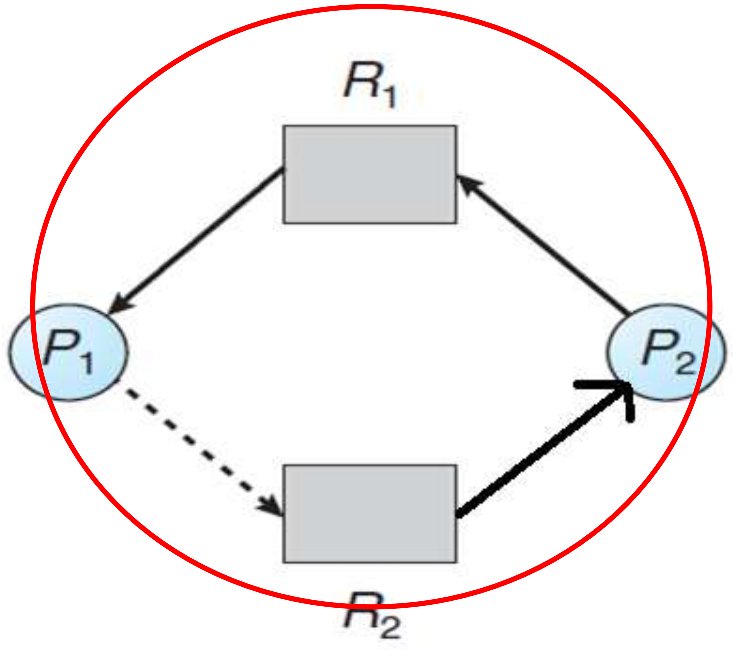

R_1
P_1
P_2
R_2

- Now suppose that process *Pi requests resource Rj, the request can be* granted only if converting the request edge *Pi → Rj to an assignment edge Rj → Pi does not result in the* ***formation of a cycle.***
- If **no cycle exists**, then the allocation of the resource will leave the system in a **safe state.**
- If a **cycle is found**, then the allocation will put the system in an **unsafe state**.
- We check for safety by using a **cycle-detection algorithm**.
- An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system



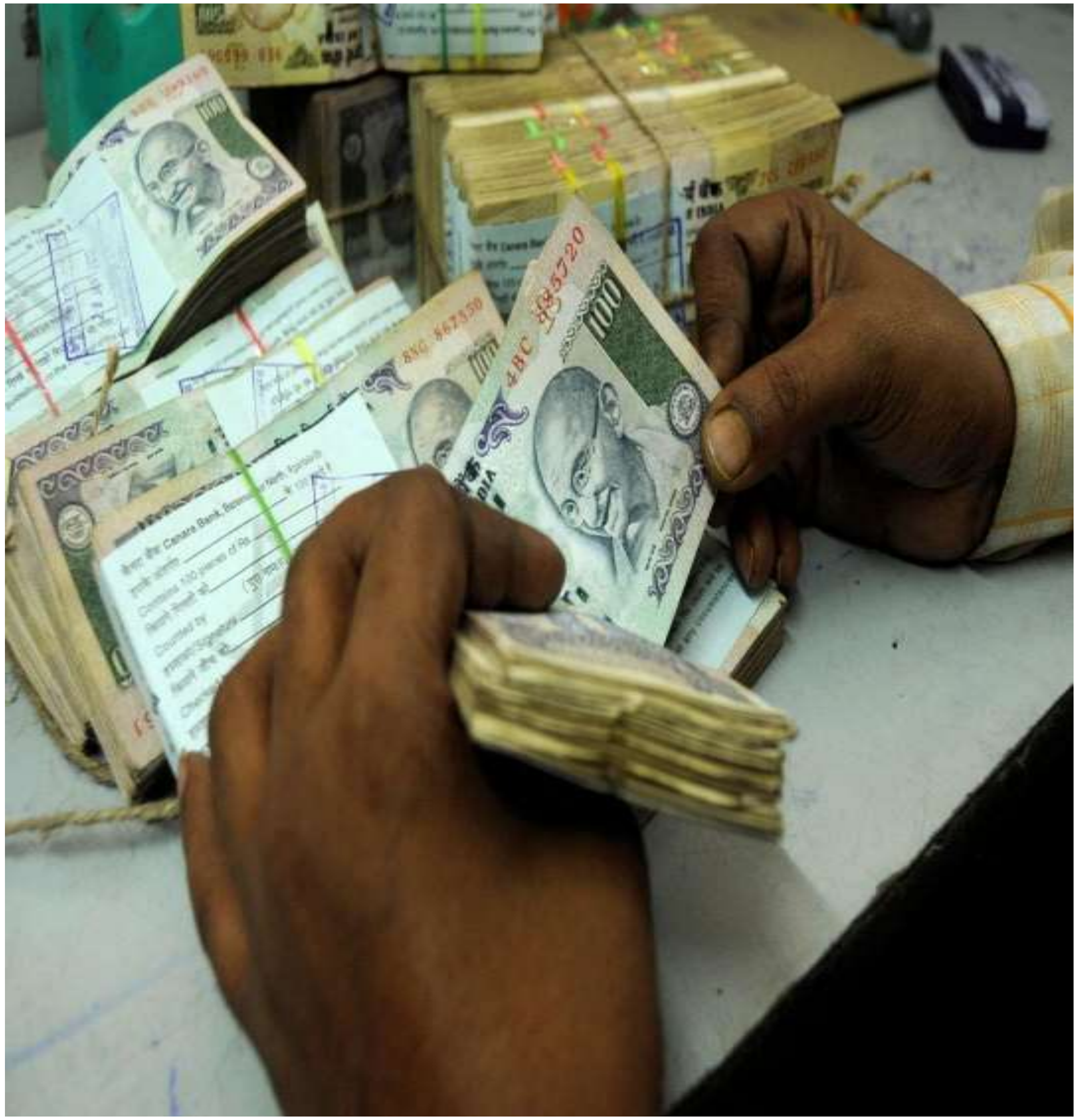


Don’t allocate R2 to P2



BANKER’S ALGORITHM FOR DEADLOCK AVOIDANCE

Bank



- Knowledge of next day money withdrawals from saving accounts and loans is essential to have sufficient cash
- Customers must inform manager about their money demand on the previous day
- Manager will carry a limited money based on the reported requests and possible deposits by customers
- No customer will be allowed to withdraw more than what he claimed yesterday
- Even for valid requests, Manager will make a judicious decision whether to allow the withdrawal or not considering the pending transactions of other customers

Possible decisions when a customer requests money

- **Invalid request** – either not informed already or exceeding the amount informed
- **Valid request but rejected** – considering the money in hand and the remaining requests by others, giving the money may cause difficulty in handling other requests
- **Valid request and accepted** - considering the money in hand and the remaining requests by others, it will be still possible to fulfill the requests of other customers

Overview of Banker’s algorithm

- Every process must declare their resource needs when they come for execution
- Whenever a process makes a request for one or more resources , Kernel must decide whether to accept the request or to decline it
- It will reject the request
  - If request violates claim
  - If requested resources are not currently available
  - if it determines that it wont be able to fulfill the demands of all the other processes if the request is accepted and the resources are allocated- deadlock
- It will accept the request if it determines that it will be able to fulfill the demands of all the other processes even if the request is accepted and the resources are allocated

Data needed

- Several resources are available in different instances. It is represented in a **Resource vector**
- Out of the total resources, some of them are already allocated to processes. The remaining resources are represented in a **Available vector**
- Processes submit their total requirements of various resources when the commence execution. This is represented in a **Claim or Max matrix**
- Some of the needed resources are already allocated to processes. This is represented in **Allocation matrix**
- Pending requirements are calculated as Need – Allocation. This is represented in **Need matrix**

Safe State

- A state is *safe if the system can allocate resources to each process as per their maximum demand* **in some order** without causing a deadlock.
- A system is in a **safe state** only if there exists a **safe sequence** to complete all processes.
- When a process is allocated all it resources as per its demand then it is expected to **complete and release** all the resources
- A sequence of processes *<P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi may make till its completion can be satisfied by the currently* available resources plus the resources held by all processes

- A **safe state** is free from deadlocks
- Conversely, a state that may lead to a deadlock is an **unsafe state**.
- Not all unsafe states are going to result in deadlocks, however an unsafe state has the **potential *to lead to a deadlock but not guaranteed to lead to deadlock.***
- As long as the state is safe, the operating system can avoid deadlock.

Data Structures needed

- **Available.** A vector of length *m indicates the **number of available resources*** **of each type of resource from 1 to m**.
- If ***Available[j]*** *equals* **k***, then k instances of resource type Rj* are available.
- **Max/Claim/Need.** An ***n × m matrix*** *defines the **maximum demand of each process**. Rows represent processes and columns represent resources*
- If *Max[i][j] equals k, then process Pi may require k instance of Rj*
- **Allocation.** An *n × m matrix defines the number of resources of each type* currently allocated to each process. If *Allocation[i][j] equals k, then process. Pi is currently allocated k instances of resource type Rj . uest at most k instances of* resource type *Rj* .

- **Need.** An *n × m matrix indicates the **remaining resource need** of each* process.
- If *Need[i][j] equals k, then process Pi may need k more instances* of resource type *Rj to complete its task. Note that Need[i][j] equals Max[i][j] – Allocation[i][j].*

[Resource = **R** = $(R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = **V** = $(V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = **C** = $\begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process *i* for resource *j*
Allocation = **A** = $\begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process *i* of resource *j*

Need = Max - Allocation

Available

A	B	C	D
3	3	2	1

Max

	A	B	C	D
P1	4	2	1	2
P2	5	2	5	2
P3	2	3	1	6
P4	1	4	2	4
P5	3	6	6	5

-

Allocation

A	B	C	D
2	0	0	1
3	1	2	1
2	1	0	3
1	3	1	2
1	4	3	2

9

=

Need

A	B	C	D
2	2	1	1
2	1	3	1
0	2	1	3
0	1	1	2
2	2	3	3

Resources

9+3

A	B	C	D
12	12	8	10

Allocation + Available

Current state is safe or unsafe ?

- Suppose there are 10 instances of a resource available

	Max	Allotted	M - A
P0	10	0	10
P1	4	1	3
P2	9	4	5

- Resources Available - 5
- Possible sequence of execution
- (P1, P2, P0), (P2,P1,P0) , (P0, P1, P2)

- If P1 given all it's resources

	Max	Allotted	M - A
P0	10	0	10
P1	4	4	0
P2	9	4	5

- Resources Available - 2
- When P1 completes

	Max	Allotted	M - A
P0	10	0	10
P1	4	0	0
P2	9	4	5

- Resources Available - 6

- If P2 given all it's resources

	Max	Allotted	M - A
P0	10	0	10
P1	4	0	0
P2	9	9	0

- Resources Available - 1
- When P2 completes

	Max	Allotted	M - A
P0	10	0	10
P1	4	0	0
P2	9	0	0

- Resources Available - 10

Since all the processes can be completed it is a safe sequence and the system is in safe state

	Max	**Allotted**	**M - A**
P0	10	0	10
P1	4	1	3
P2	9	4	5

- Another safe sequence (P2,P1,P0)
- Unsafe sequence (P0, P1, P2)

Whether request of P0 can be accepted or should be declined?

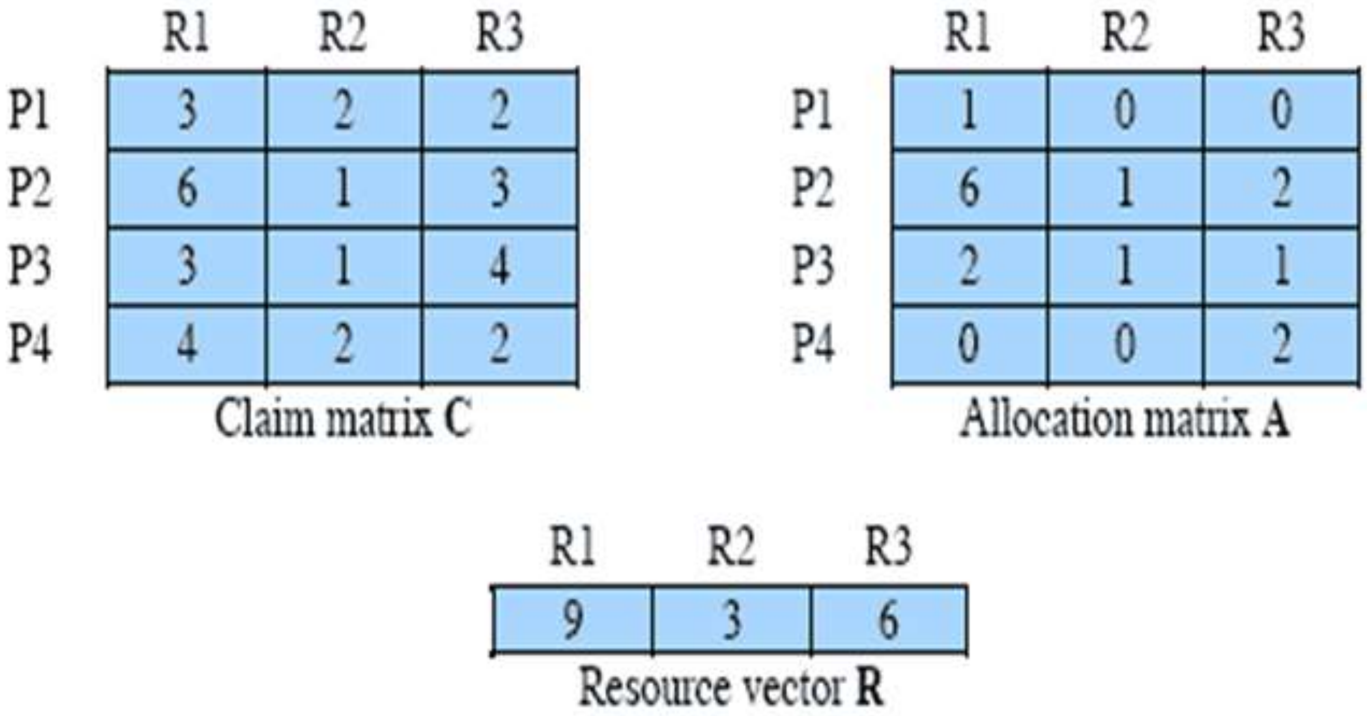
	Max	Allotted	M - A
P0	10	0	10
P1	4	1	3
P3	9	4	5

- P0 requests 3 resources – whether allocating them will lead to **unsafe state** ?

	Max	Allotted	M - A
P0	10	3	7
P1	4	1	3
P3	9	4	5

- After allocation to P0 Resources Available = 2
- It will lead to unsafe state because after allocating 3 resources to P0, no process will get their needed resources and so it might lead to a deadlock
- Decline the request of P0 and block it

Find whether the given system's state is safe or unsafe



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

R1	R2	R3
9	3	6

Resource vector R

Steps

1. Calculate the **Need** matrix as

```
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        Need[i][j] = Claim[i][j] – Allocation[i][j]
    }
}
```

2. Calculate the **Available** vector as

```
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        sum+=Allocation[j][i]
    }
    Available[i]=Resources[i]-sum
    sum=0;
}
```

Need Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

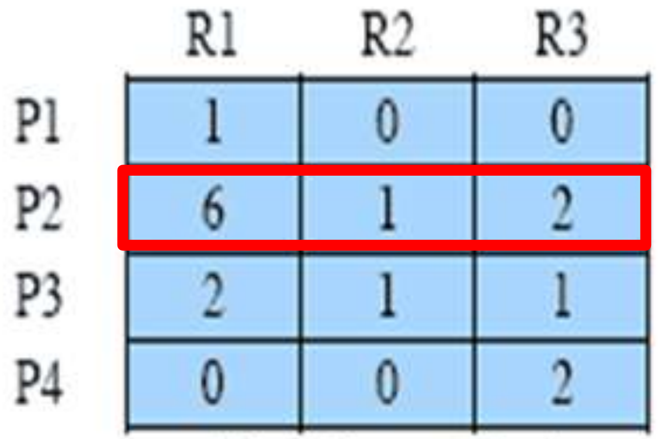
R1	R2	R3
0	1	1

Available vector **V**

- 3. Find a sequence such that all the processes can be completed with the available resources

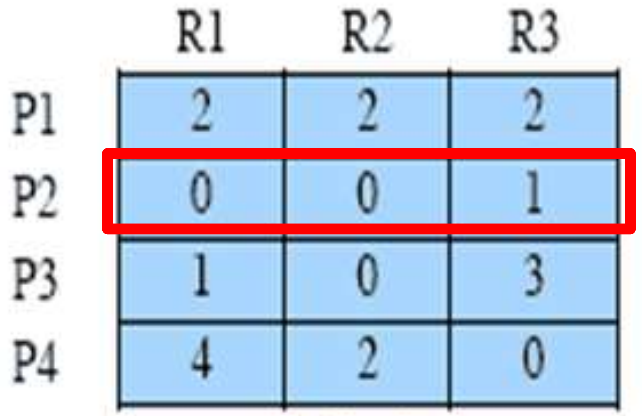
```
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        if(Need[i][j] < Available[j])
        {   pr_count++; return i;   }
    }
}
return -1
```

Need Matrix



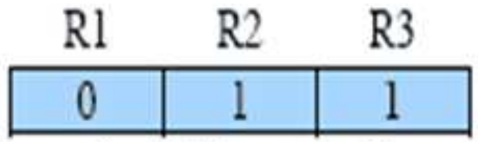
	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A



	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

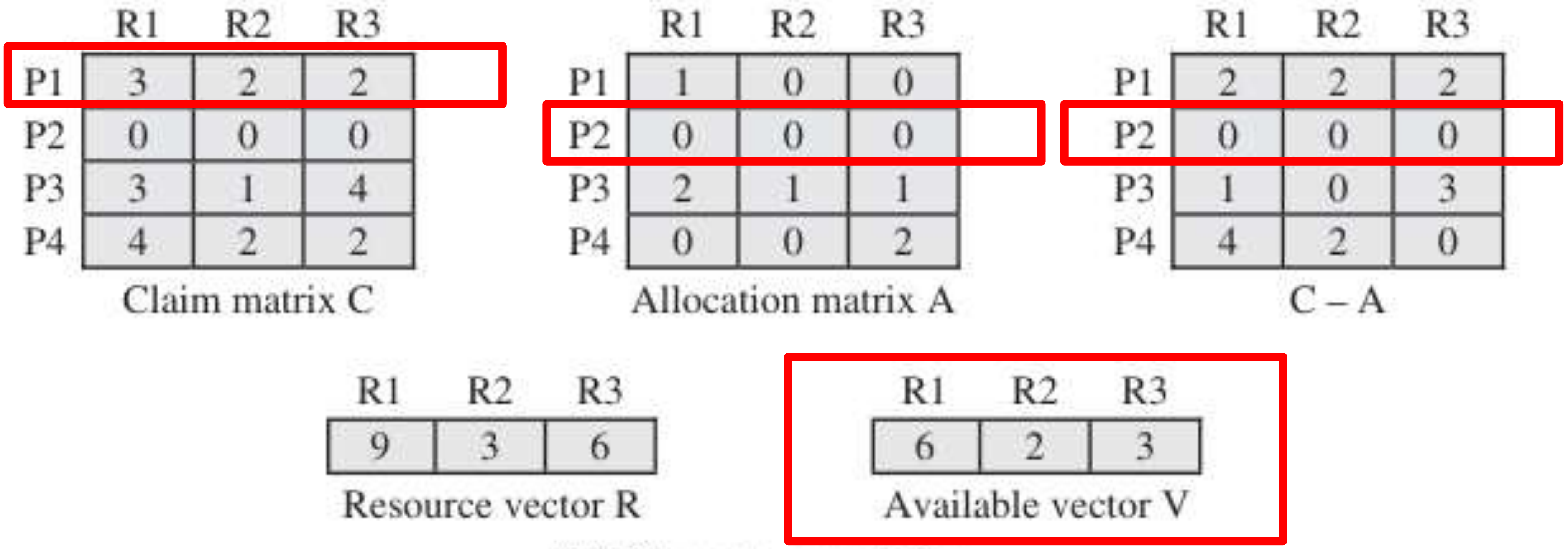
C – A



R1	R2	R3
0	1	1

Available vector V

If P2 given its needed resources



	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

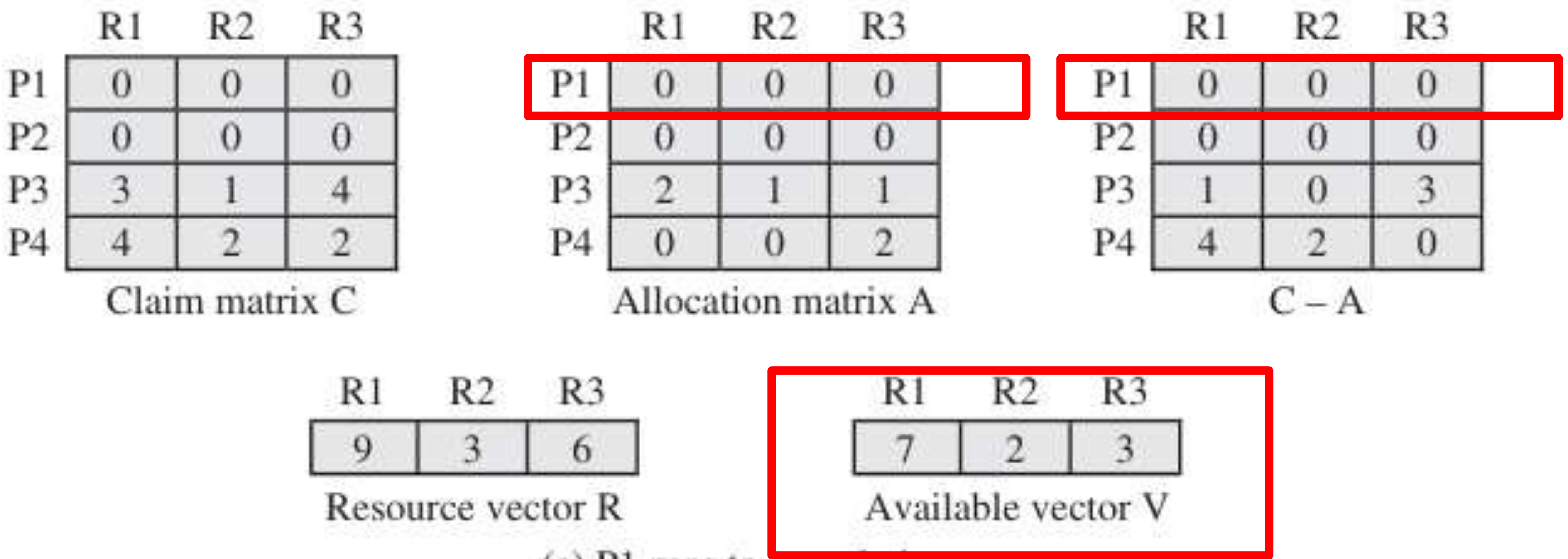
Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

If P1 given its resources



	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

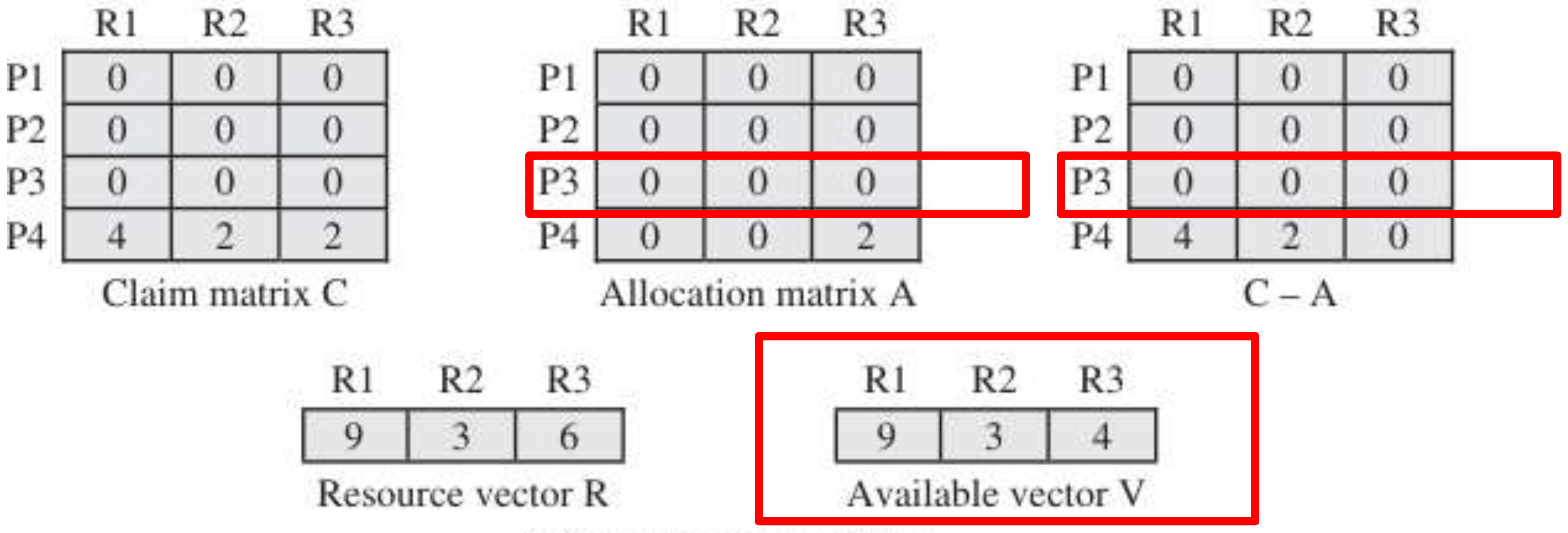
Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

If P3 given its resources



	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

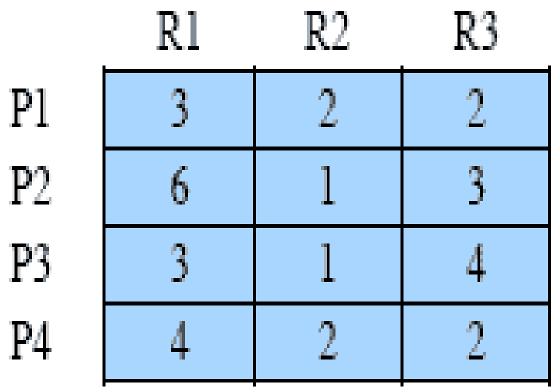
(d) P3 runs to completion

Since there is a safe sequence P2, P1, P3, P4 to complete all the processes, the system is in Safe state

New request for additional resource

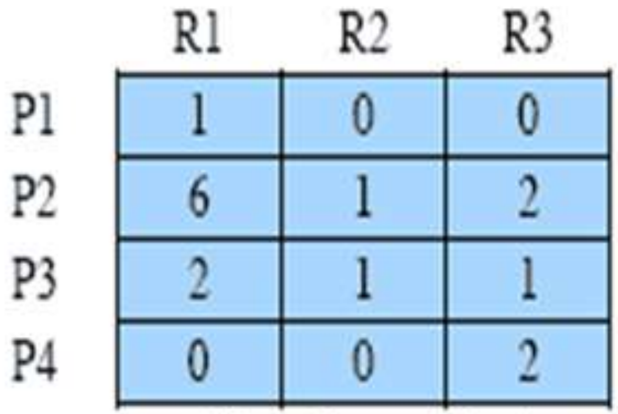
- Suppose P2 makes a request for 1 additional unit of R3, find whether it can be accepted or to be rejected as per banker’s algorithm
  - P2(0,0,1)

If P2 allotted the resource R3



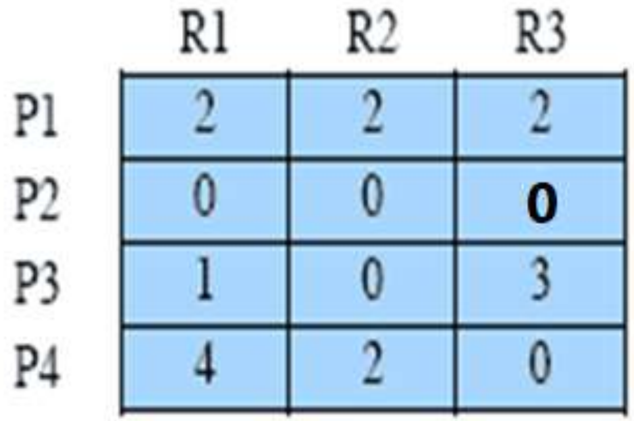
	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C



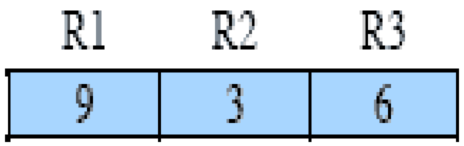
	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A



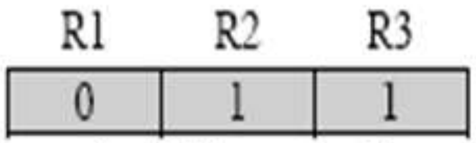
	R1	R2	R3
P1	2	2	2
P2	0	0	**0**
P3	1	0	3
P4	4	2	0

C – A



R1	R2	R3
9	3	6

Resource vector R



R1	R2	R3
0	1	1

Available vector V

Since P2 has met its resource requirements it eventually completes and release all its resources

Next P1 can be allocated all its resources

	R1	R2	R3
P1	3	2	2
P2	~~0~~	~~0~~	~~0~~
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	~~0~~	~~0~~	~~0~~
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	~~0~~	~~0~~	~~0~~
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

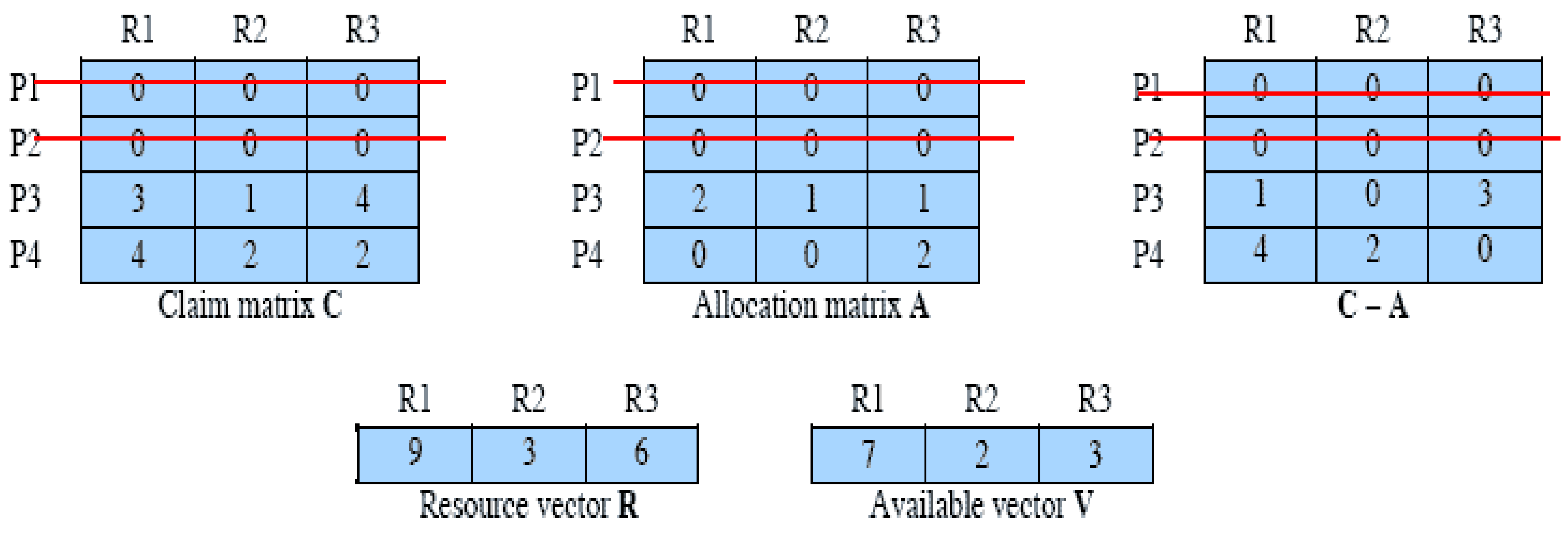
Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) **P2** runs to completion

Next P3 can be allocated all its resources



	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

Next P3 can be allocated all its resources

	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
P3	1	0	3
P4	4	2	0

C – A

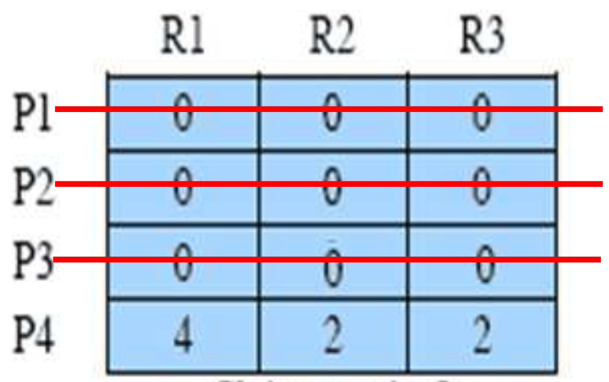
R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

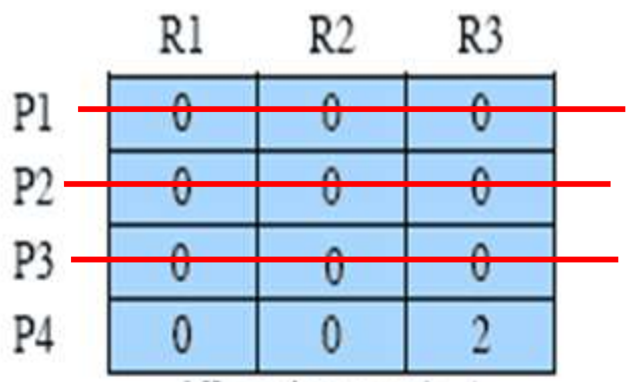
Available vector V

(c) P1 runs to completion



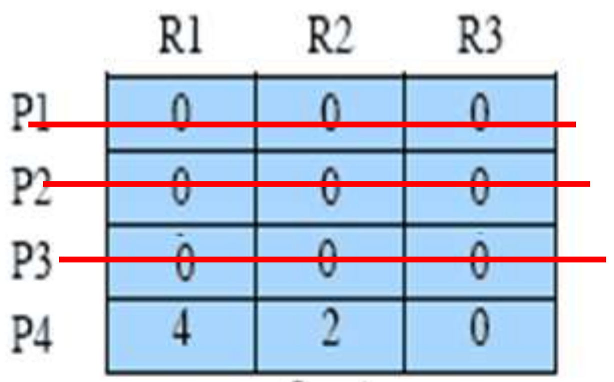
	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
~~P3~~	~~0~~	~~0~~	~~0~~
P4	4	2	2

Claim matrix C



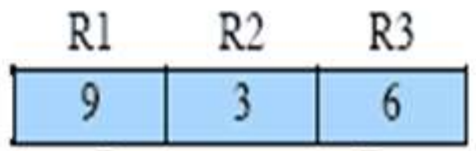
	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
~~P3~~	~~0~~	~~0~~	~~0~~
P4	0	0	2

Allocation matrix A



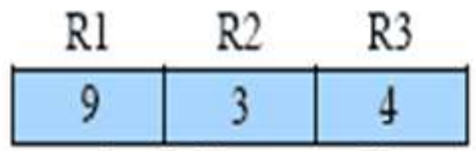
	R1	R2	R3
~~P1~~	~~0~~	~~0~~	~~0~~
~~P2~~	~~0~~	~~0~~	~~0~~
~~P3~~	~~0~~	~~0~~	~~0~~
P4	4	2	0

C – A



R1	R2	R3
9	3	6

Resource vector R



R1	R2	R3
9	3	4

Available vector V

Since all the processes can be completed it is safe to allocate resource to P2 of R3

Final state of the system after P2 allocated R3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	**3**
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	**0**
P3	1	0	3
P4	4	2	0

C – A

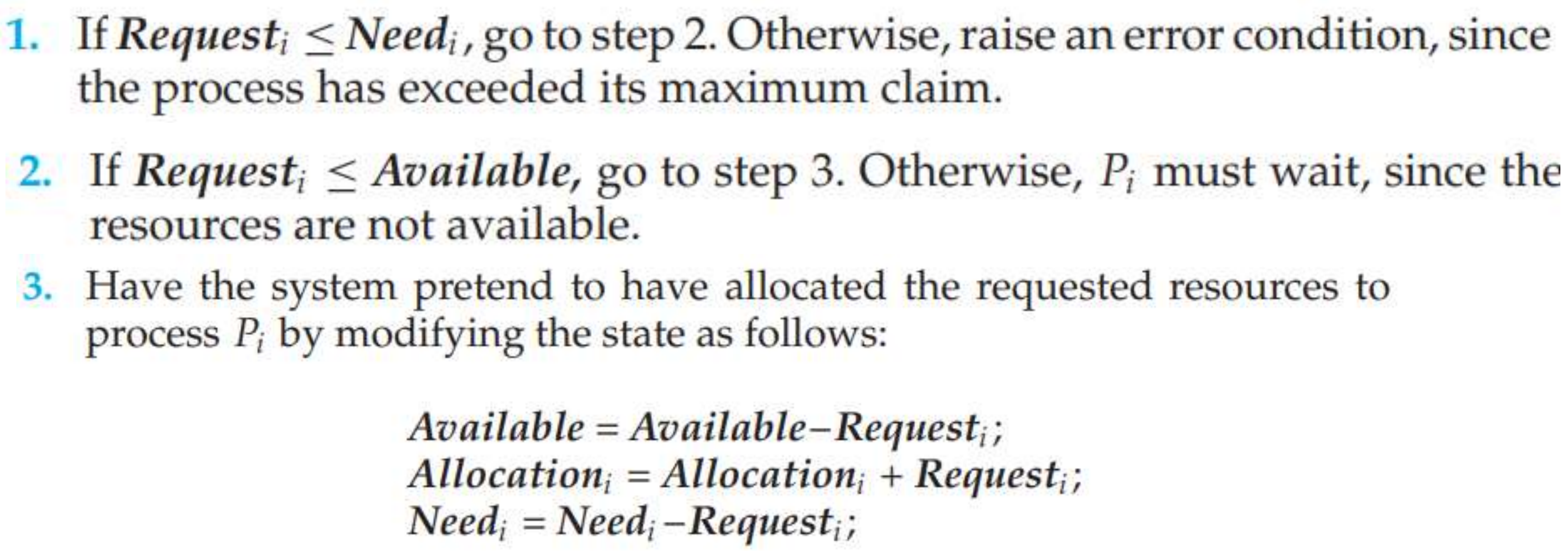
R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	**0**

Available vector V

Resource-Request Algorithm



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

$$\begin{aligned}
&\boldsymbol{Available} = \boldsymbol{Available} - \boldsymbol{Request_i};\\
&\boldsymbol{Allocation_i} = \boldsymbol{Allocation_i} + \boldsymbol{Request_i};\\
&\boldsymbol{Need_i} = \boldsymbol{Need_i} - \boldsymbol{Request_i};
\end{aligned}$$

If the resulting resource-allocation state is **safe** (using **safety algorithm**), the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then Pi must wait for $Request_i$, and the old resource-allocation state is restored

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. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.

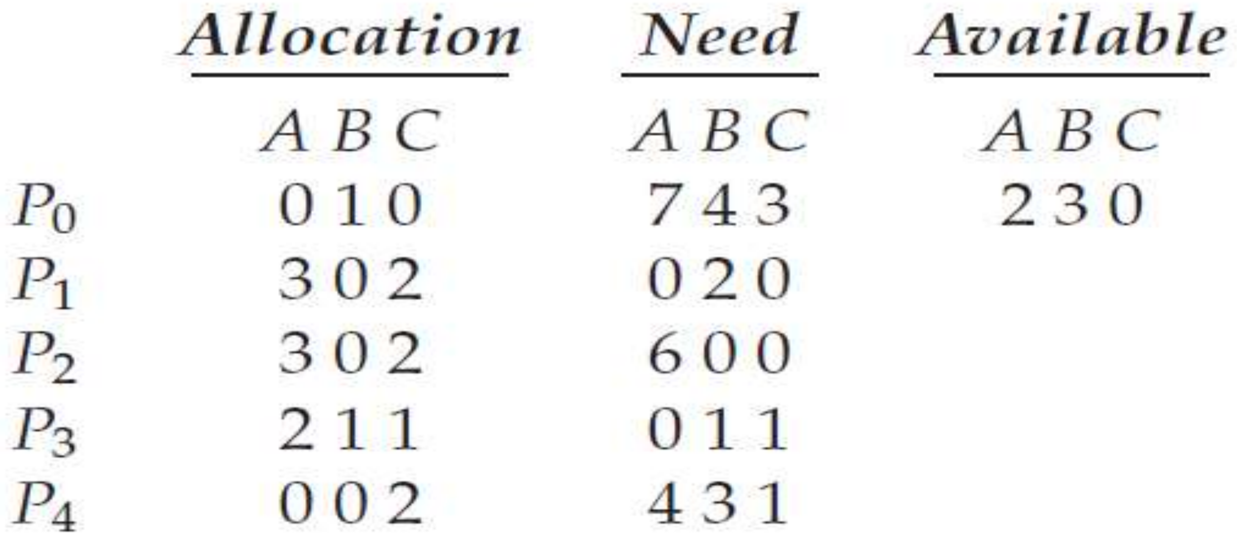
4. If $Finish[i] == true$ for all i, then the system is in a safe state.

	Allocation	***Max***	***Available***
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

	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

- We claim that the system is currently in a safe state.
- Indeed, the sequence ***<P1, P3, P4, P2, P0>*** *satisfies the safety criteria.*
- Suppose now that process ***P1*** *requests one additional instance of resource type A and two instances of* resource type *C, so Request1 = P1 (1,0,2).*
- *To decide whether this request can be* immediately granted, we first check that ***Request P1 ≤ Available—that is, that*** **(1,0,2) ≤ (3,3,2)**, which is true.
- We then pretend that this request has been fulfilled, and we arrive at the following new state:



	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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe. To do so, we execute our **safety algorithm** and find that the sequence <*P1, P3, P4, P0, P2*>satisfies the safety requirement.
- Hence, we can immediately grant the request of process *P1.*

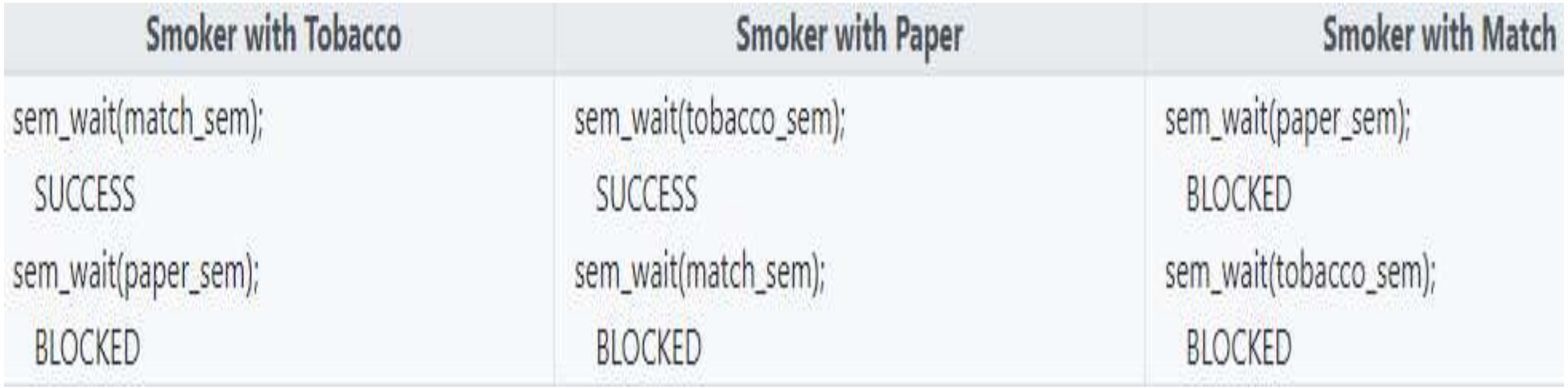
- You should be able to see, however, that when the system is in this state, a request for (3,3,0) by *P4 cannot be granted, since the resources are not available.*
- Furthermore, a request for (0,2,0) by *P0 cannot be granted, even though the* resources are available, since the resulting state is unsafe

Deadlock Avoidance

- **Advantage**
- No need to restrict the OS by eliminating any of the four conditions from occuring
- **Limitations**
- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

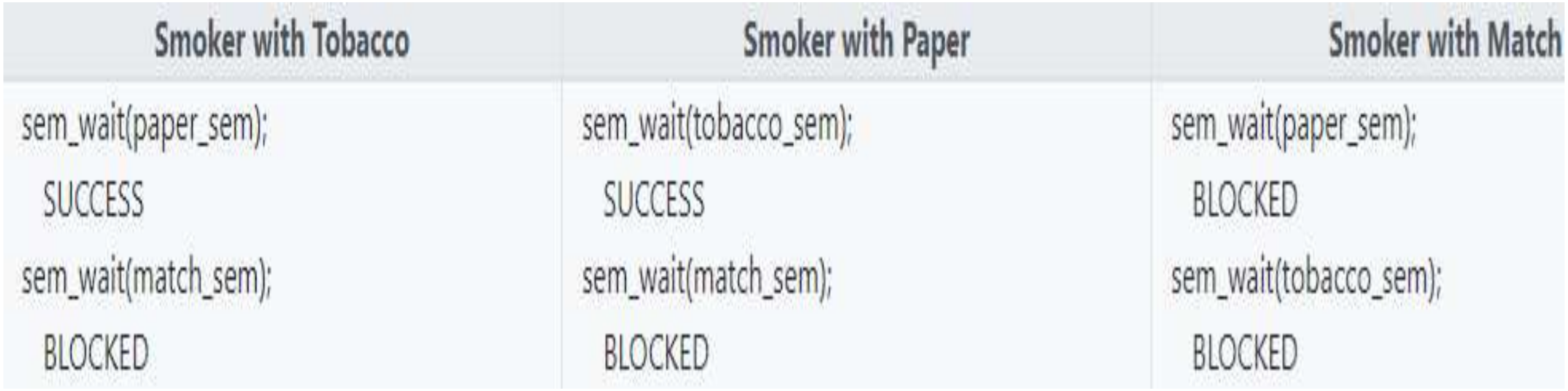
Cigarette Smokers Problem

- There four threads – **three smokers and one agent**
- The smoker needs to acquire **three items –tobacco, paper and match**
- Once a smoker has all the three they combine the paper, tobacco and light it using the match box
- The cigarette smokers problem states that it is ***impossible*** **to create the smoker threads** that would **avoid deadlock.**



Smoker with Tobacco	Smoker with Paper	Smoker with Match

sem_wait(match_sem);
SUCCESS
sem_wait(paper_sem);
BLOCKED
	sem_wait(tobacco_sem);
	SUCCESS
	sem_wait(match_sem);
	BLOCKED
		sem_wait(paper_sem);
		BLOCKED
		sem_wait(tobacco_sem);
		BLOCKED



Smoker with Tobacco	Smoker with Paper	Smoker with Match
sem_wait(paper_sem);	sem_wait(tobacco_sem);	sem_wait(paper_sem);
SUCCESS	SUCCESS	BLOCKED
sem_wait(match_sem);	sem_wait(match_sem);	sem_wait(tobacco_sem);
BLOCKED	BLOCKED	BLOCKED

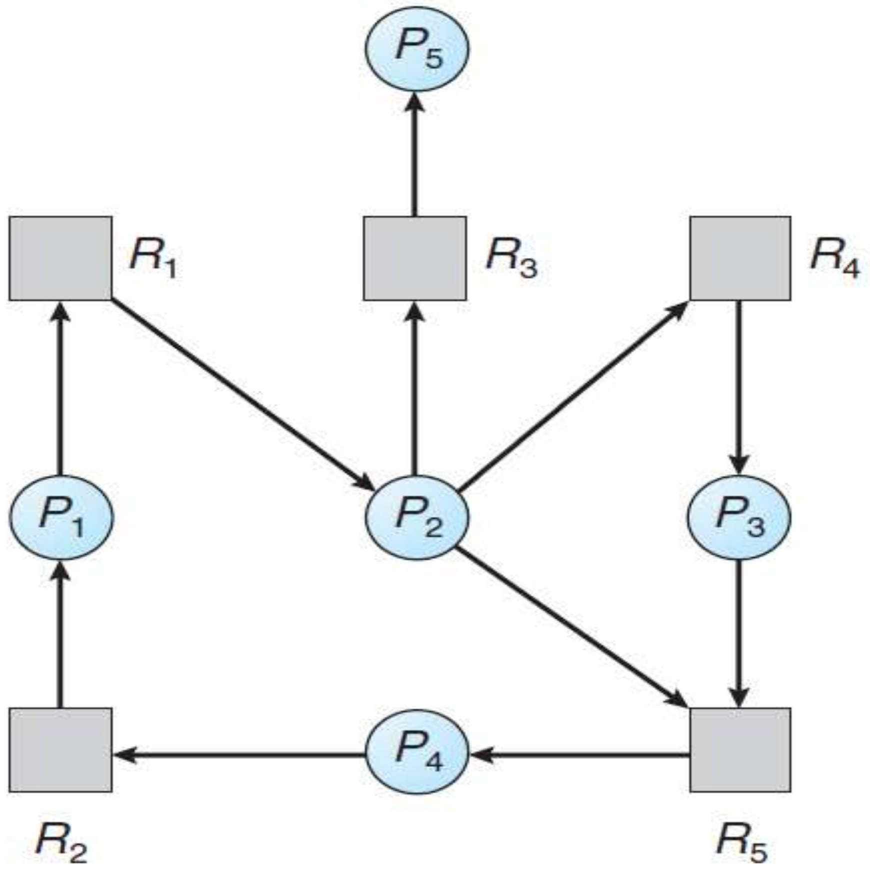
Deadlock Detection and Recovery

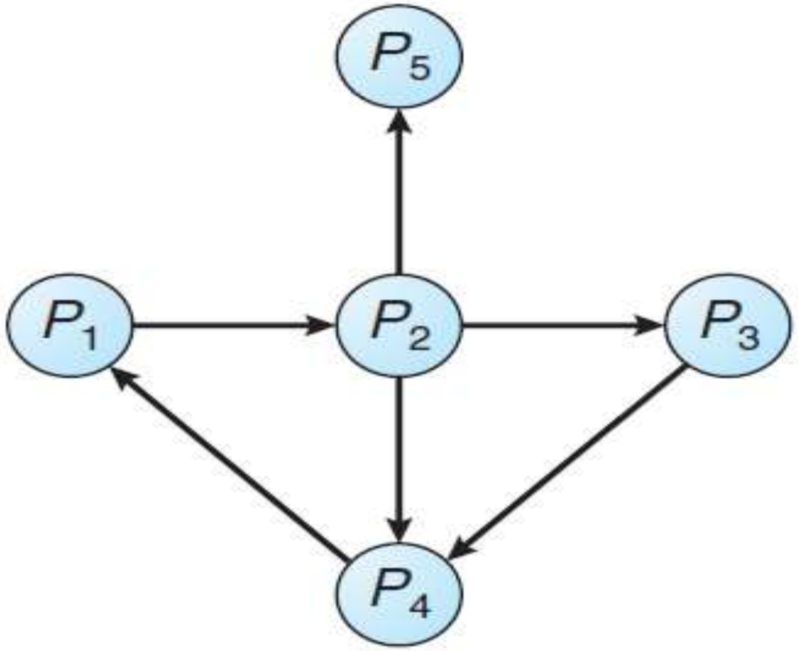
Deadlock Detection

- If a system **does not employ** either a **deadlock-prevention or a deadlock avoidance**, then a deadlock situation may occur.
- In this environment, the system may provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - An algorithm to recover from the deadlock

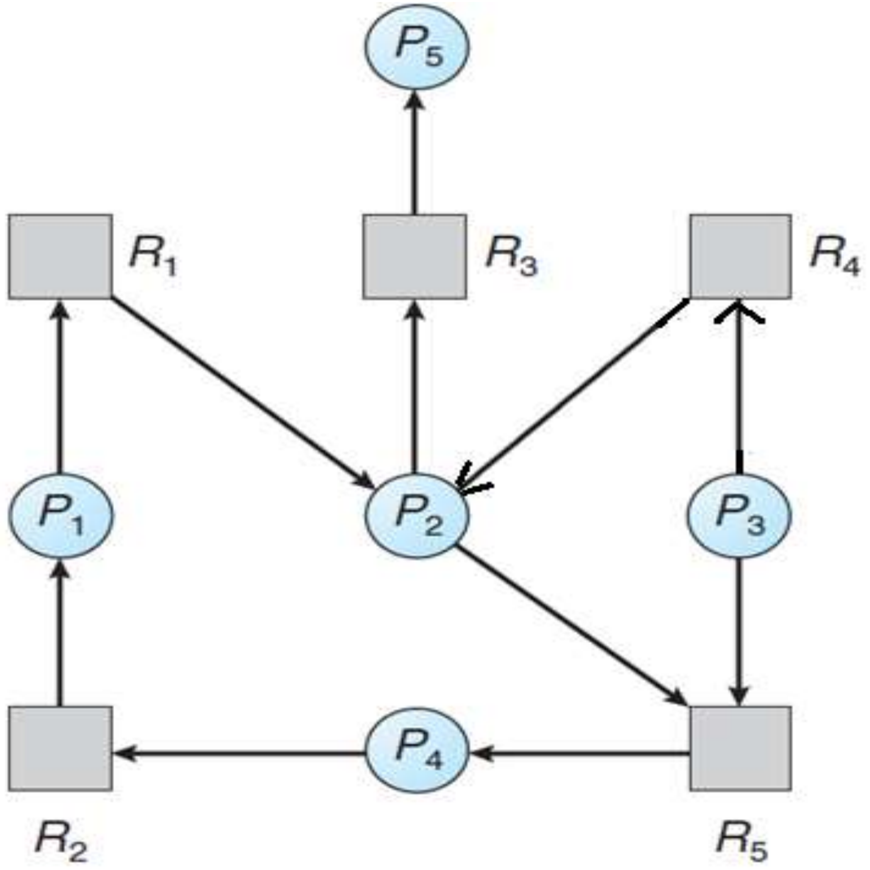
Single Instance of Each Resource Type

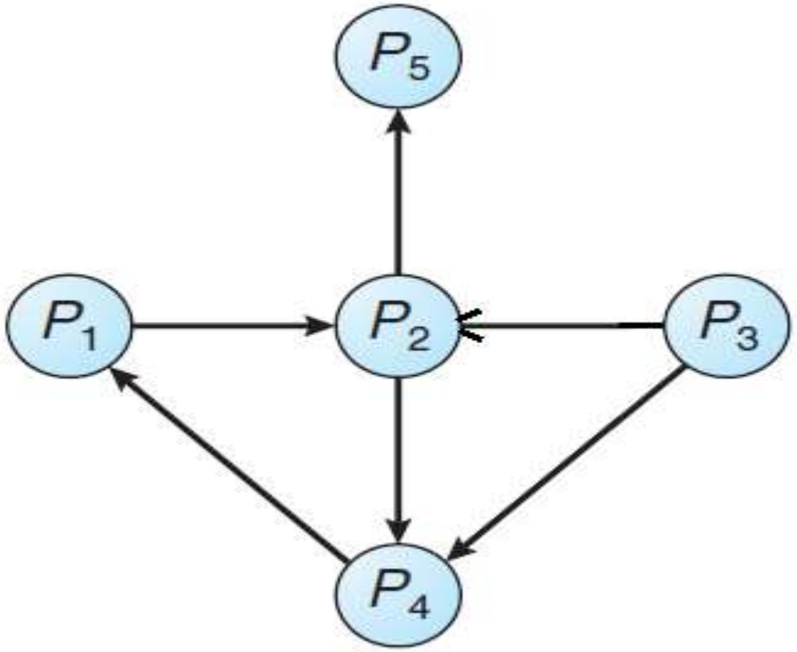
- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph.**
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from *Pi to Pj in a wait-for graph implies that* process *Pi is waiting for process Pj to release a resource that Pi needs.*
- *An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation* graph contains two edges *Pi → Rq and Rq → Pj for some resource Rq .*


P_5
R_1
R_3
R_4
P_1
P_2
P_3
P_4
R_2
R_5


P_5
P_1
P_2
P_3
P_4

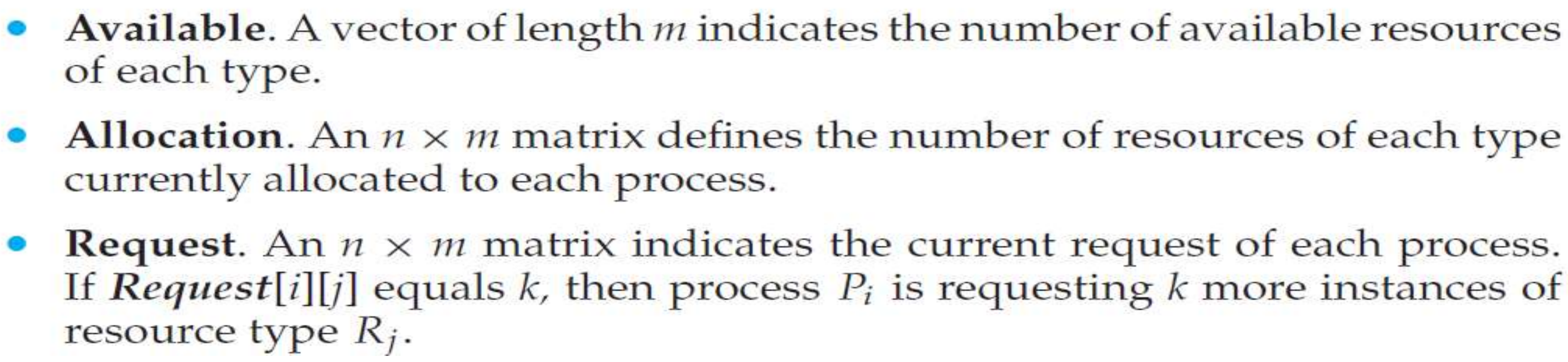
- A deadlock exists in the system if and only if the wait-for graph contains a **cycle.**
- To detect deadlocks, the system needs to ***maintain the wait for*** **graph** and **periodically** ***invoke*** *an algorithm that searches for a cycle in* the graph.


P_5
R_1
R_3
R_4
P_1
P_2
P_3
P_4
R_2
R_5


P_5
P_1
P_2
P_3
P_4

Deadlock Detection with Several Instances of a Resource Type

- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm



- **Available**. A vector of length m indicates the number of available resources of each type.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An $n \times m$ matrix indicates the current request of each process. If ***Request***$[i][j]$ equals k, then process P_i is requesting k more instances of resource type R_j.

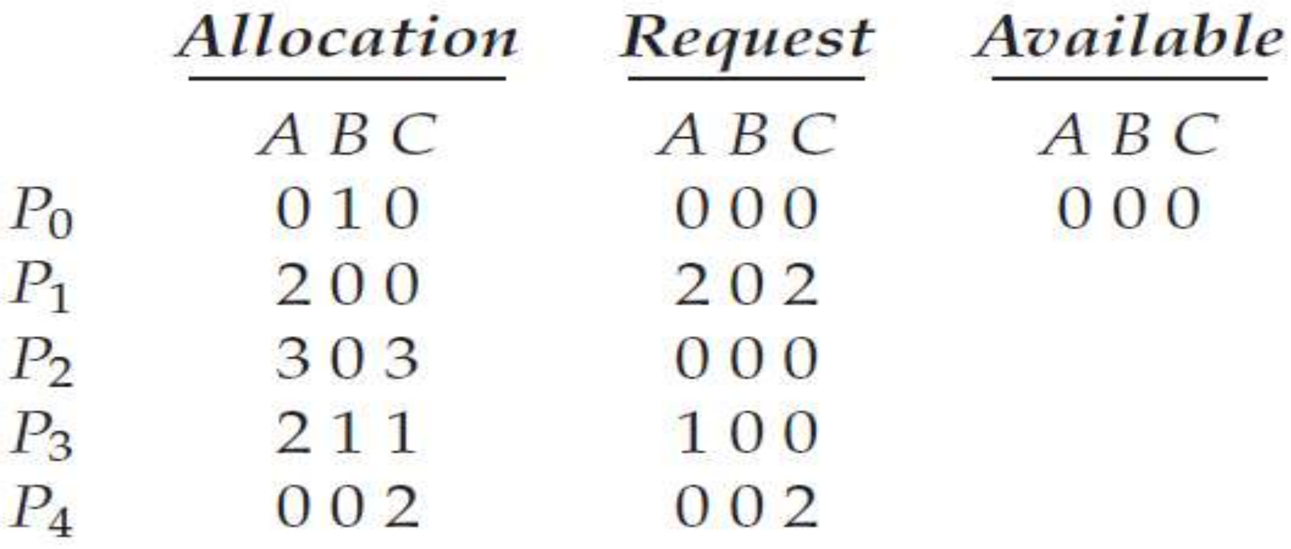
Deadlock Detection Vs Avoidance

- **Avoidance:** Every time a process requests for a resource, the OS invokes the Banker's algorithm to determine whether allocation of the resource would result in unsafe state. If yes then it avoids allocating the resource.
- **Detection:** The system follows no precaution while allocating resources, a process requesting a resource is allocated if the resource is available. But periodically detection algorithm is run to check whether any deadlock exist.

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***
   c. 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, then the system is in deadlocked state.*** Moreover, if ***Finish[i] == false, then process Pi is deadlocked.***

Deadlock detection

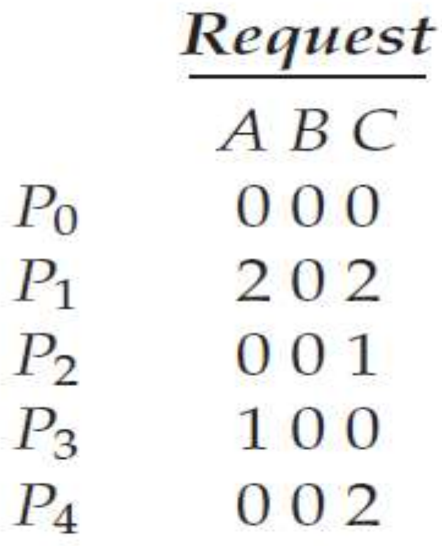


	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

•The system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence *<P0, P2, P3, P1, P4> results in**Finish[i] == true for all i***

Suppose now that process *P2 makes one additional request for an instance* of type *C.*

Request matrix is modified as follows:



	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

The system is now deadlocked.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
- **1. How *often is a deadlock likely to occur?***
- **2. How *many processes will be affected by deadlock when it happens?***
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
- In addition, the number of processes involved in the deadlock cycle may grow.

- Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time.
- A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent

Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods.
- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later
- **Abort processes successively (one by one) until deadlock is resolved**

Summary

- Deadlocks are result of overlapped resource requests by two or more processes.
- Resource allocation graphs useful tools.
- Four necessary conditions are required for deadlock to occur.
- OS may prevent or avoid or detect or ignore deadlocks.
- Bankers algorithm is used for avoidance.
- Once a deadlock is detected system need to be recovered.