MET's Institute of Engineering

Bhujbal Knowledge City, Adgaon, Nashik.

Department of Computer Engineering

"Synchronization and Concurrency Control"

Prepared By

Prof. Anand N. Gharu

(Assistant Professor)

PVGCOE Computer Dept.

CLASS : TE COMPUTER 2019

SUBJECT : SPOS (SEM-I)

UNIT : V

28 OCT 2023

CONTENTS :-

1. Concurrency: Principle and issues with Concurrency
2. Mutual Exclusion, Hardware approach, Software approach
3. Semaphore, Mutex and monitor
4. Reader writer problem, Producer Consumer problem, Dining Philosopher problem.
5. Deadlocks: Principle of Deadlock, Deadlock prevention
6. Deadlock avoidance
7. Deadlock detection
8. Deadlock recovery.

Concurrency

Introduction of Concurrency

1. Concurrency is the execution of the multiple instruction sequences at the same time.
2. It happens in the operating system when there are several process threads running in parallel.
3. The running process threads always communicate with each other through shared memory or message passing.
4. Concurrency results in sharing of resources result in problems like deadlocks and resources starvation.

Introduction of Concurrency

5. It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput..

Principle of Concurrency

Both interleaved and overlapped processes can be viewed as examples of concurrent processes, they both present the same problems.

The relative speed of execution cannot be predicted. It depends on the following:

1. The activities of other processes
2. The way operating system handles interrupts
3. The scheduling policies of the operating system

Problem in Concurrency

1. Sharing global resources –

Sharing of global resources safely is difficult. If two processes both make use of a global variable and both perform read and write on that variable, then the order in which various read and write are executed is critical.

2. Optimal allocation of resources –

It is difficult for the operating system to manage the allocation of resources optimally.

Problem in Concurrency

3. Locating programming errors –

It is very difficult to locate a programming error because reports are usually not reproducible.

4. Locking the channel –

It may be inefficient for the operating system to simply lock the channel and prevents its use by other processes.

Advantages of Concurrency

1. Running of multiple applications –

It enable to run multiple applications at the same time.

2. Better resource utilization –

It enables that the resources that are unused by one application can be used for other applications.

3. Better average response time –

Without concurrency, each application has to be run to completion before the next one can be run.

Disadvantages of Concurrency

1. It is required to protect multiple applications from one another.
2. It is required to coordinate multiple applications through additional mechanisms.
3. Additional performance overheads and complexities in operating systems are required for switching among applications.
4. Sometimes running too many applications concurrently leads to severely degraded performance.

Disadvantages of Concurrency

1. It is required to protect multiple applications from one another.
2. It is required to coordinate multiple applications through additional mechanisms.
3. Additional performance overheads and complexities in operating systems are required for switching among applications.
4. Sometimes running too many applications concurrently leads to severely degraded performance.

Issues of Concurrency

Non-atomic –

Operations that are non-atomic but interruptible by multiple processes can cause problems.

Race conditions –

A race condition occurs of the outcome depends on which of several processes gets to a point first.

Issues of Concurrency

3. Blocking –

Processes can block waiting for resources. A process could be blocked for long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.

4. Starvation –

It occurs when a process does not obtain service to progress.

5. Deadlock –

It occurs when two processes are blocked and hence neither can proceed to execute.

SYNCHRONIZATION

Synchronization in OS

On the basis of synchronization, processes are categorized as one of the following two types:

1. **Independent Process :** Execution of one process does not affects the execution of other processes.
2. **Cooperative Process :** Execution of one process affects the execution of other processes.

“The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization.”

Synchronization Mechanism

- **Race Condition :**

A **Race Condition** typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

- **Critical Section :**

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

Critical Section Problem

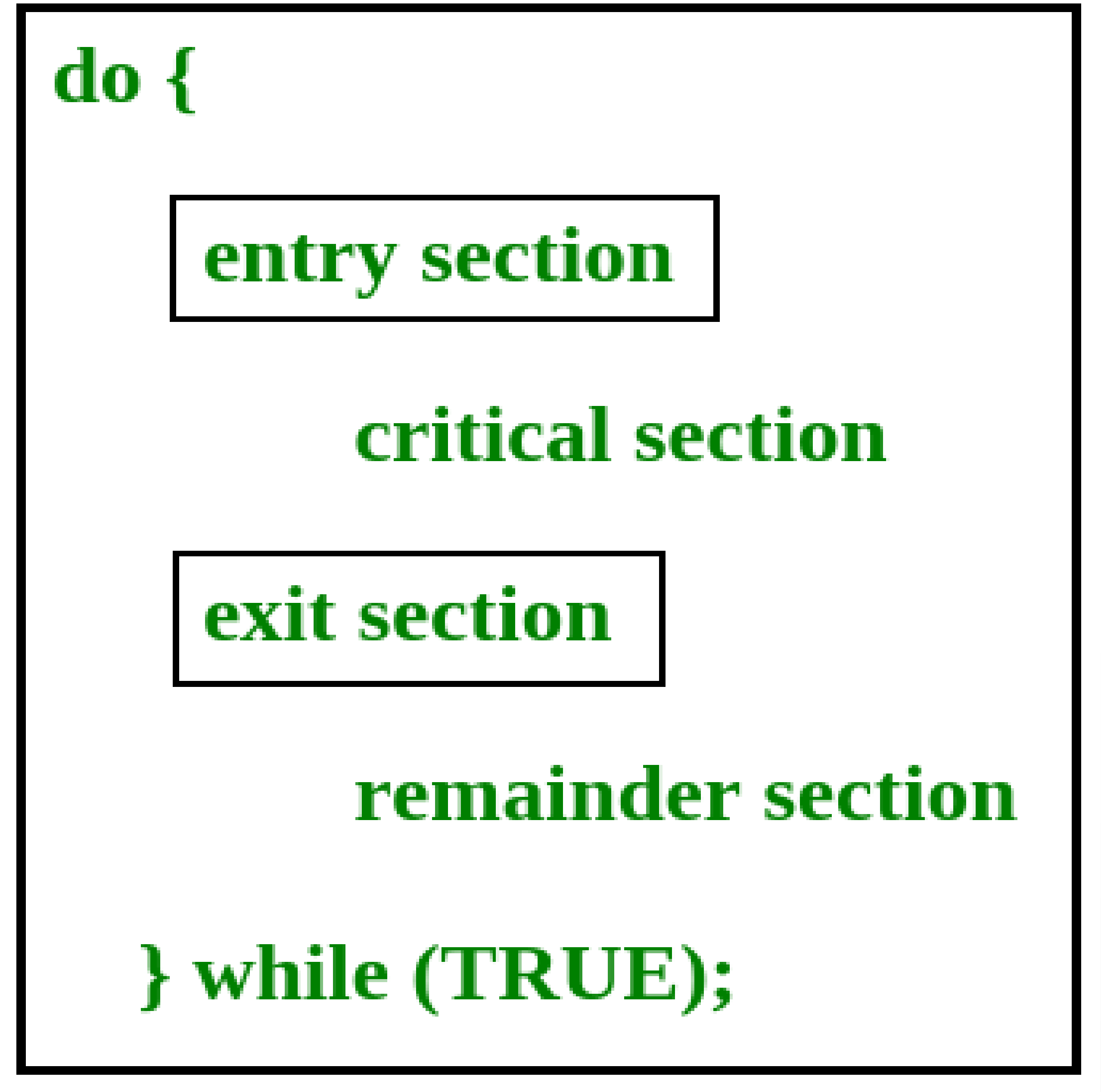
- Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

- The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

- The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Critical Section Problem

- Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.



```
do {
      entry section
            critical section
      exit section
            remainder section
   } while (TRUE);
```

Critical Section Problem

- In the entry section, the process requests for entry in the Critical Section.
- **Any solution to the critical section problem must satisfy three requirements:**

1. Mutual exclusion
2. Progress
3. Bounded waiting

Requirements of Synchronization mechanisms

- **Primary**
- **Mutual Exclusion**

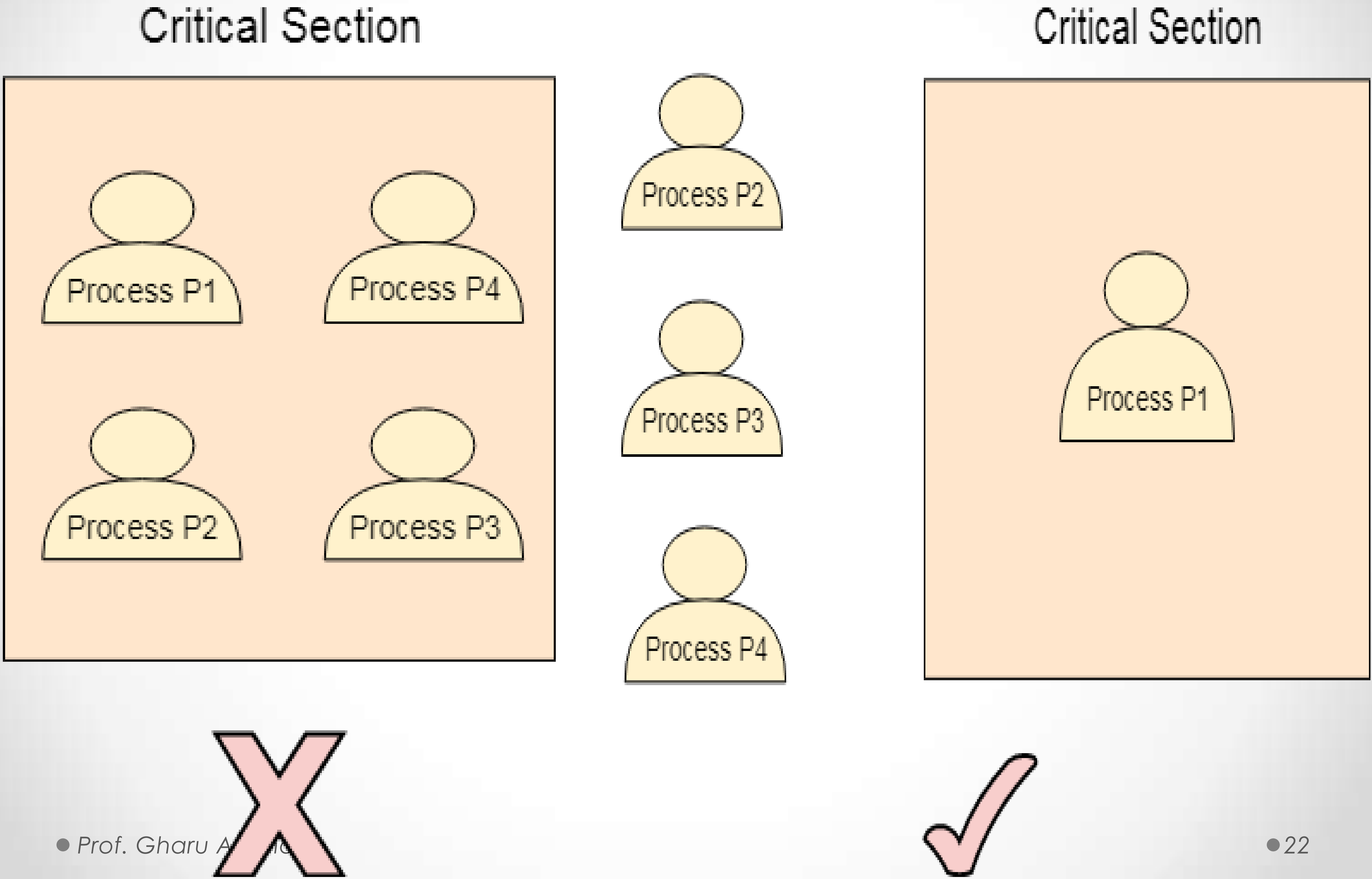
Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.

Requirements of Synchronization mechanisms

- **Primary**
- **Progress**

Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Critical Section Problem



Requirements of Synchronization mechanisms

- **Secondary**
- **Bounded Waiting**
- We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

Requirements of Synchronization mechanisms

- **Secondary**
- **Architectural Neutrality**

Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Interprocess communication

In computer science, **inter-process communication** or **interprocess communication** (**IPC**) allows communicating processes to exchange the data and information.

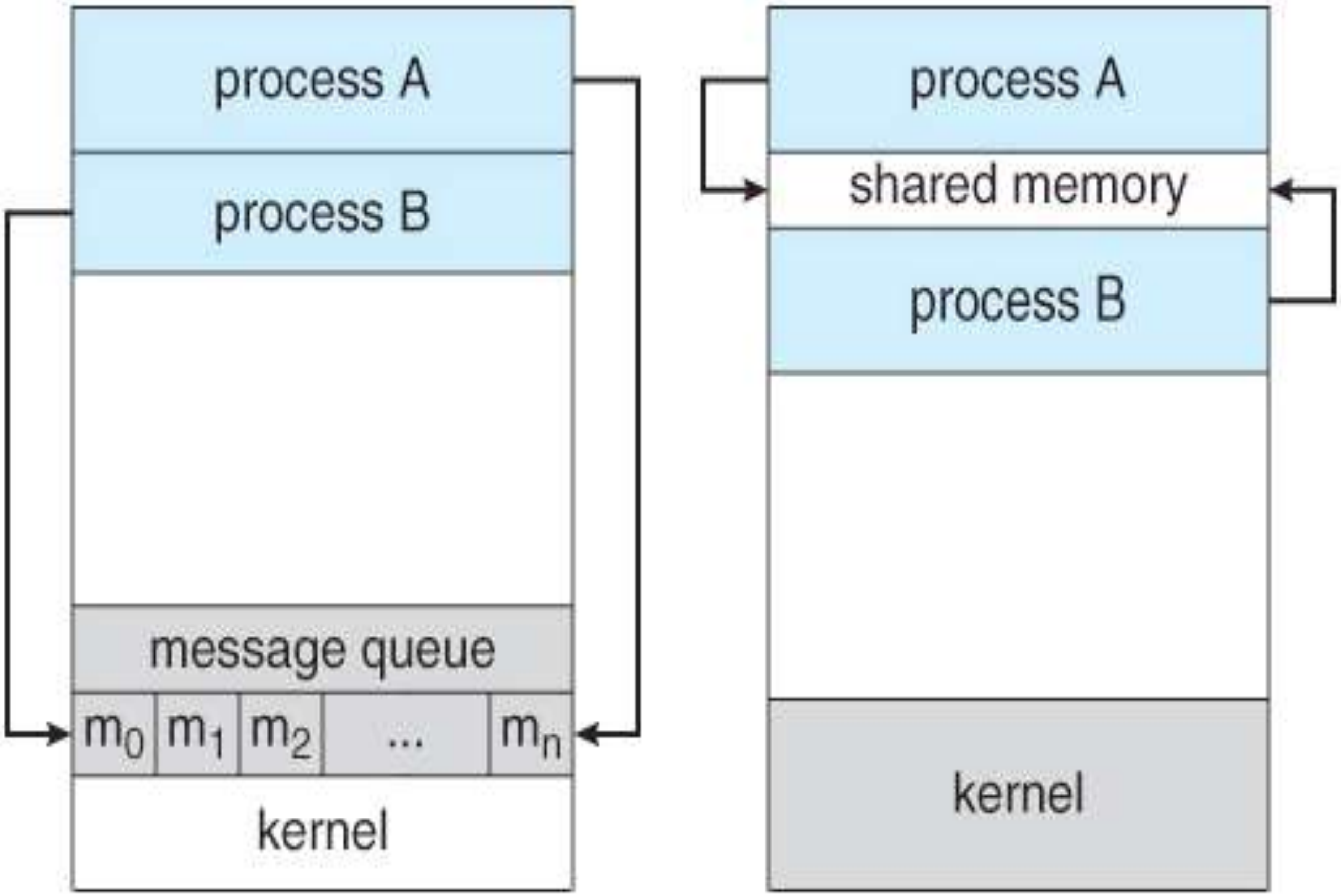
There are two methods of IPC :

1. Shared memory
2. Message passing

Interprocess communication

There are two primary models of inter process communication:

- shared memory and
- message passing.



(a) Message Passing

(b) Shared Memory

Interprocess communication

- **shared memory :**

  In this, processes are interact with each other through shared variable;

  processes are exchange information by reading & writing data using shared variable.

- **message passing** :.

  in this, instead of reading or writing, processes send and receive the messages.

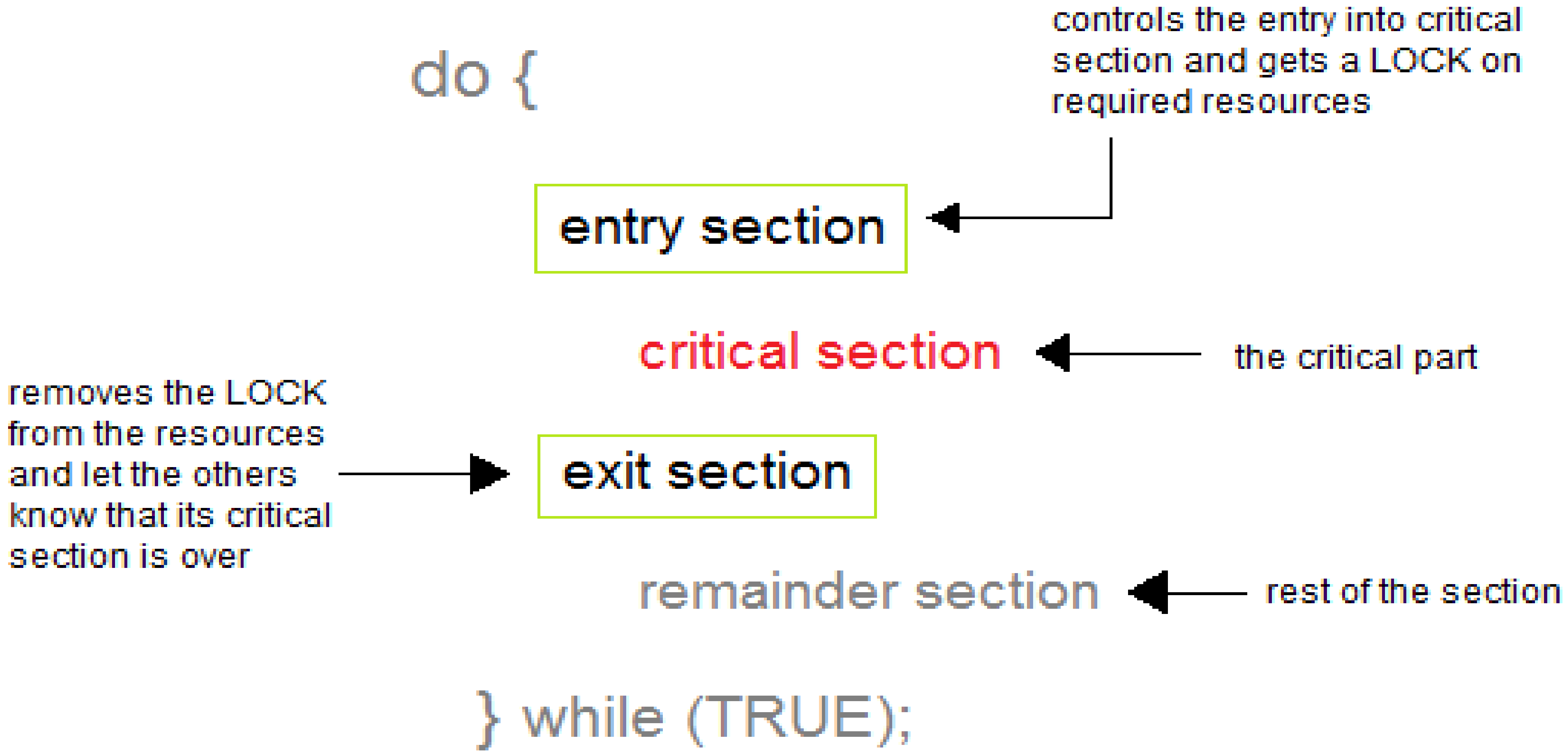
  send and receive functions are implemented in OS.

SEND (B, message)

RECEIVE (A, memory address)

Critical section

critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

controls the entry into critical section and gets a LOCK on required resources

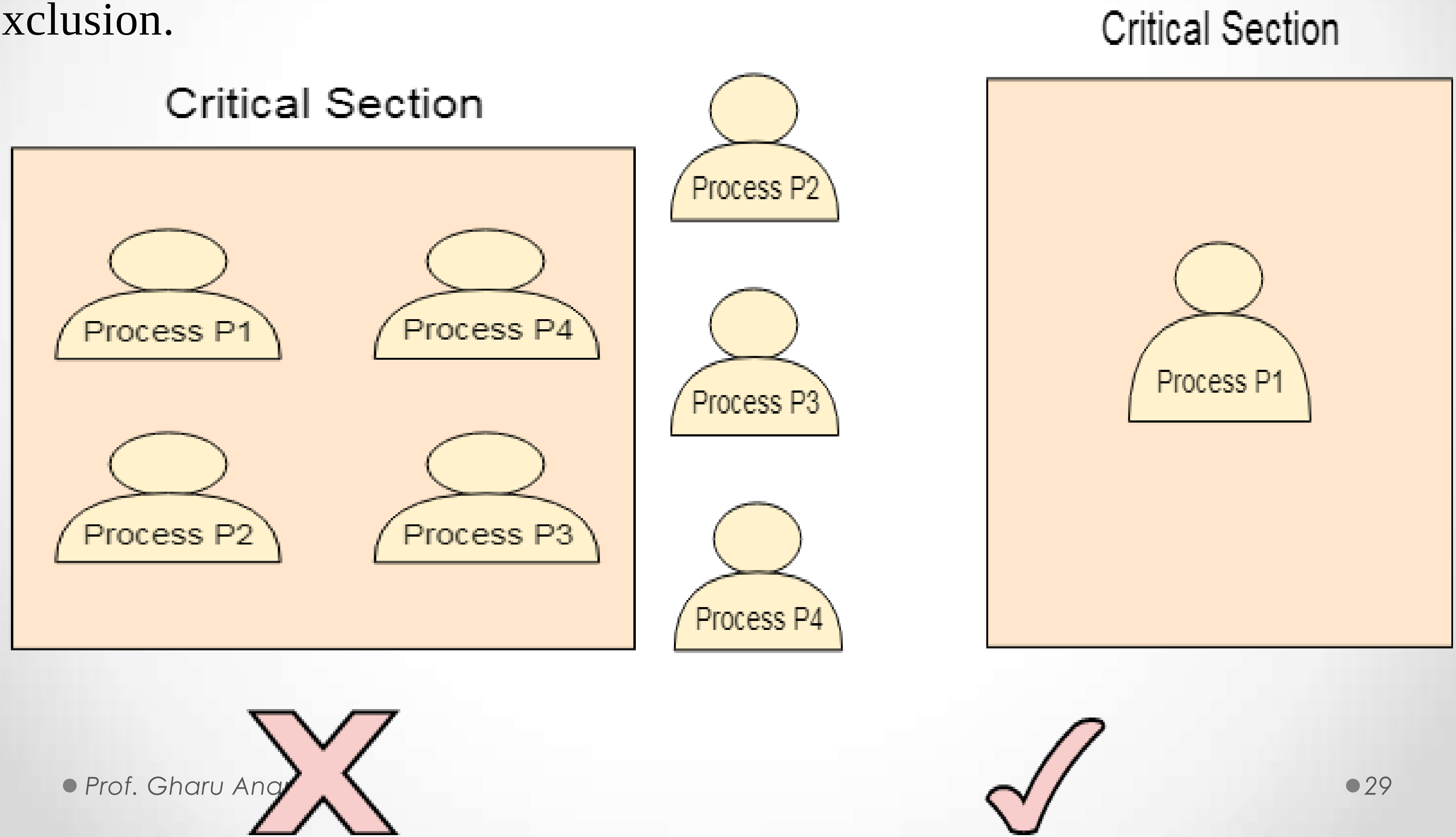
the critical part

removes the LOCK from the resources and let the others know that its critical section is over

rest of the section

Mutual Exclusion

A **mutual exclusion** (mutex) is a program in which Shared resource is not allowed to access by more than one process at same time is called mutual exclusion.



Semaphore in IPC

In computer science, a **semaphore** is a variable or abstract data type **used** to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.

- ➢ Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment

The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

Types of Semaphore

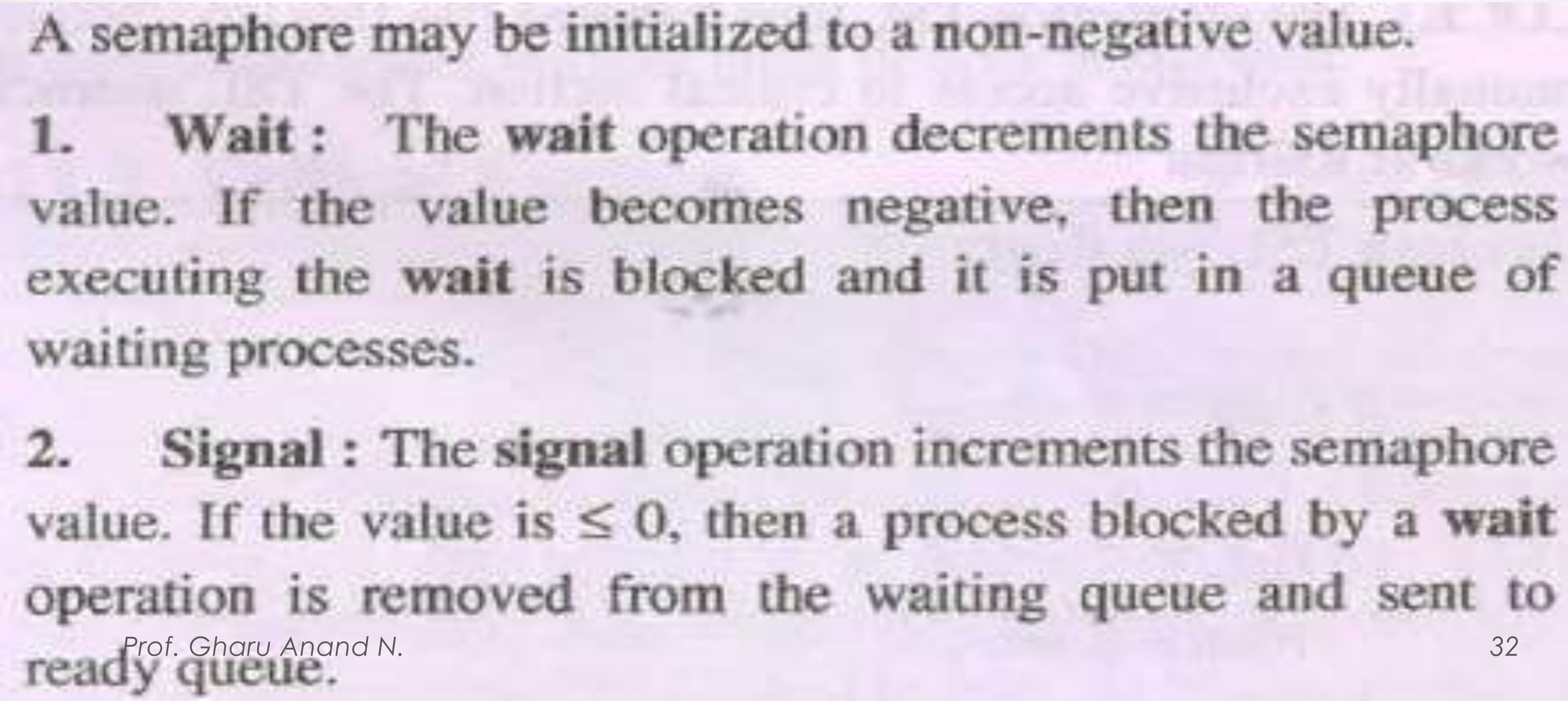
Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.

Primitives of Semaphore

There are two types of Primitives :

1. wait()

2. Signal()



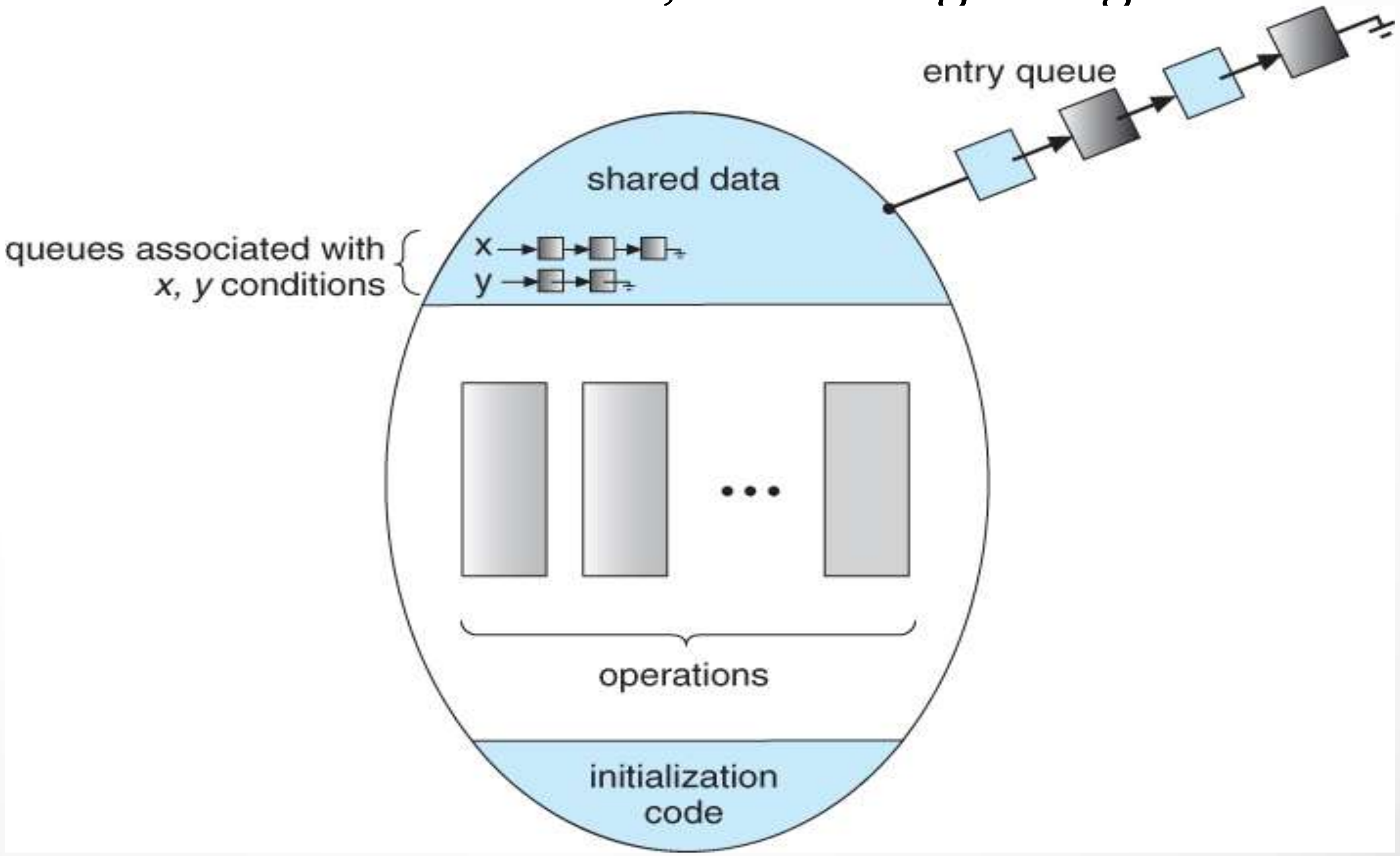
A semaphore may be initialized to a non-negative value.

1. **Wait :** The **wait** operation decrements the semaphore value. If the value becomes negative, then the process executing the **wait** is blocked and it is put in a queue of waiting processes.

2. **Signal :** The **signal** operation increments the semaphore value. If the value is ≤ 0, then a process blocked by a **wait** operation is removed from the waiting queue and sent to ready queue.

Monitor

- A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.
- Monitors also have a mechanism for signaling other threads that their condition has been met.
- A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition.
- Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.



Monitor

ASPECTS	SEMAPHORE	MONITOR
Basic	Semaphores is an integer variable S.	Monitor is an abstract data type.
Action	The value of Semaphore S indicates the number of shared resources available in the system	The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
Access	When any process access the shared resources it perform wait() operation on S and when it releases the shared resources it performs signal() operation on S.	When any process wants to access the shared variables in the monitor, it needs to access it through the procedures.
Condition variable	Semaphore does not have condition variables.	Monitor has condition variables.

IPC Problem (Classical Problem of Synchronization)

IPC Problem

1. Producer Consumer Problem
2. Reader Writer Problem
3. Dining Philosopher Problem
4. Sleeping Barber Problem

Producer-consumer problem

In computing, the **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer-consumer problem

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers

Reader - writer problem

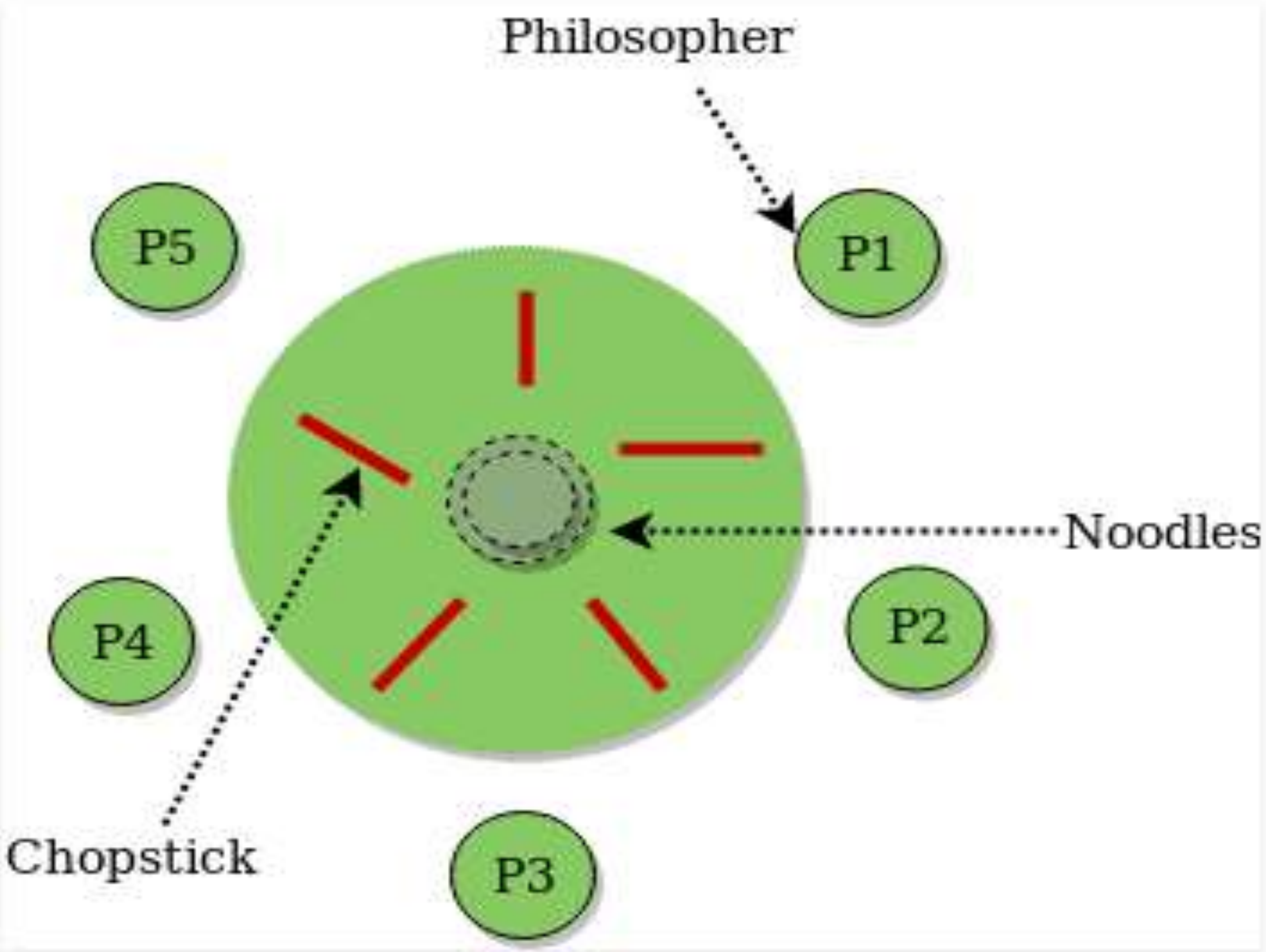
The R-W problem is another classic problem for which design of synchronization and concurrency mechanisms can be tested. The producer/consumer is another such problem; the dining philosophers is another.

Definition

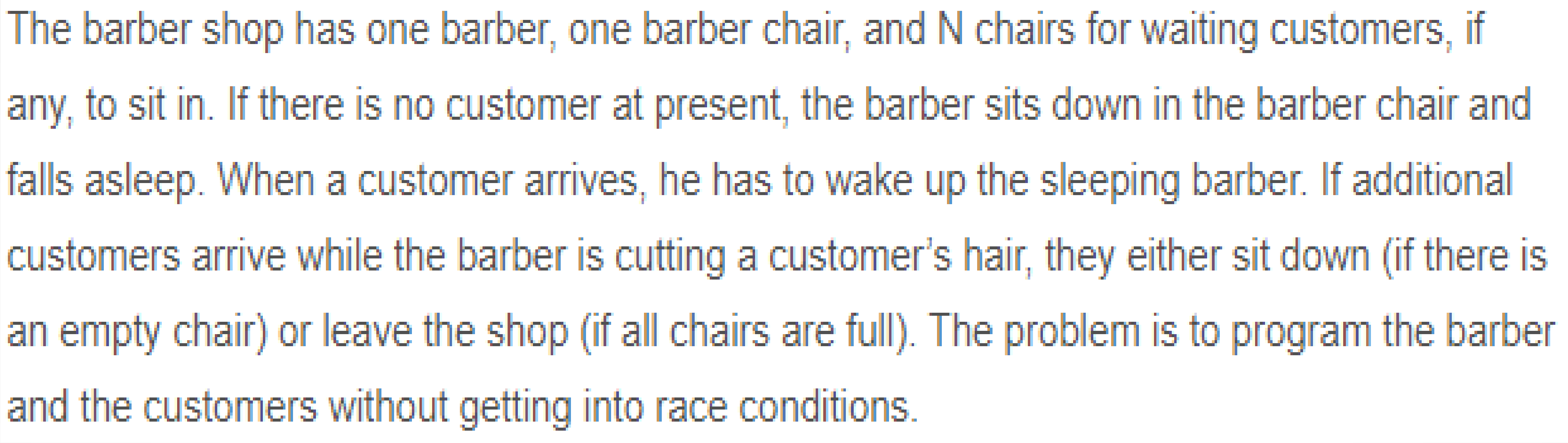
- There is a data area that is shared among a number of processes.
- Any number of readers may simultaneously write to the data area.
- Only one writer at a time may write to the data area.
- If a writer is writing to the data area, no reader may read it.
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write
- A process that reads and writes to a data area must be considered a writer (consider producer or consumer)

Dining philosopher problem

The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Barber sleeping problem

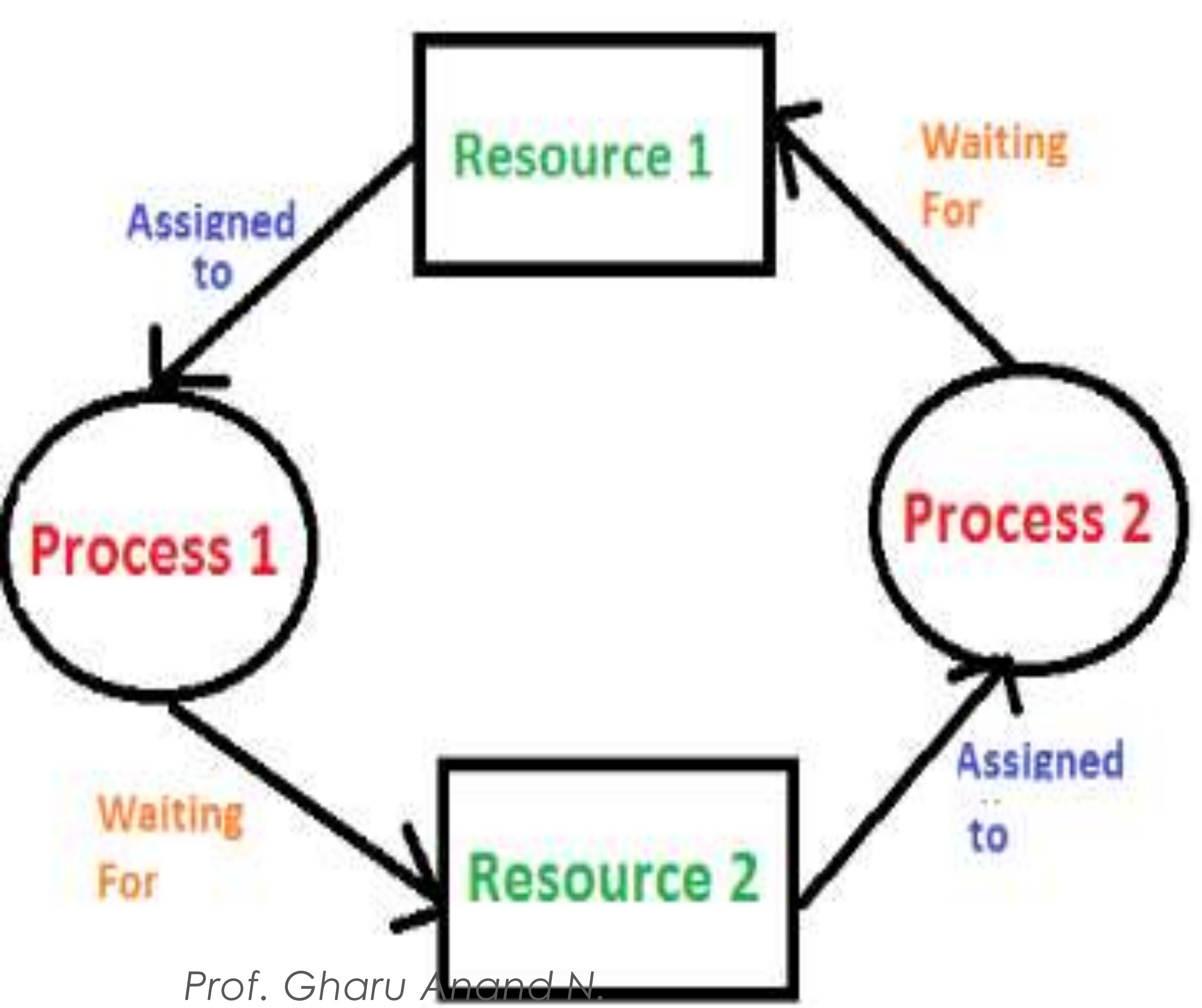


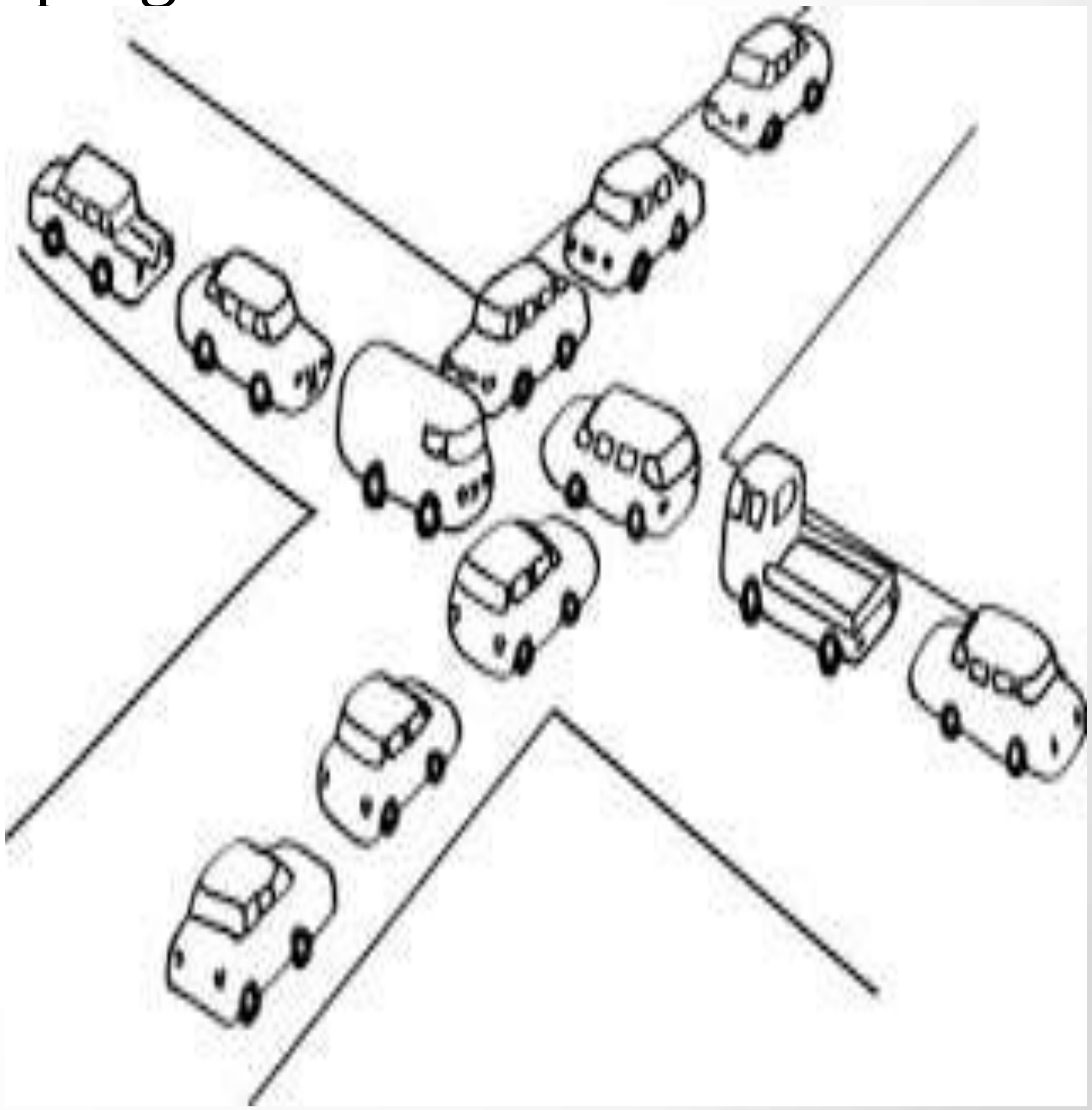
The barber shop has one barber, one barber chair, and N chairs for waiting customers, if any, to sit in. If there is no customer at present, the barber sits down in the barber chair and falls asleep. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there is an empty chair) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions.

- In a barber shop, there is one barber, some chairs and some customers
- A barber sleeps if there is no customer (not even on chairs, waiting for a haircut ☹)
- A customer wakes up the barber if it's his turn to get his haircut
- A customer waits if there is any chair left
- A customer leaves, if all the chairs are occupied

DEADLOCK

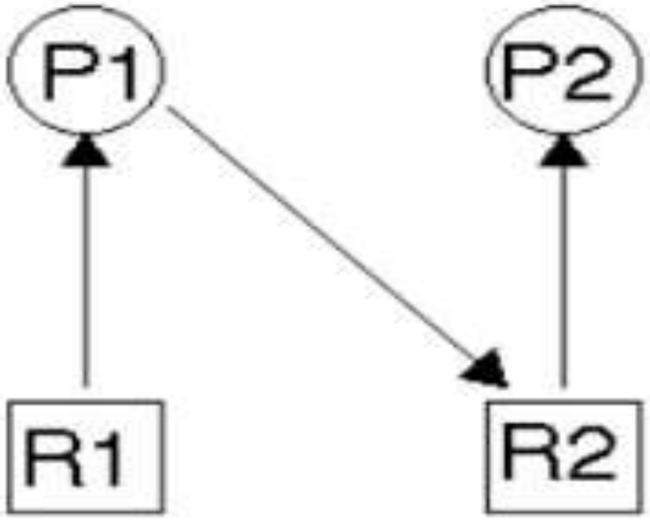
- A **deadlock** is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. The earliest computer operating systems ran only one program at a time.



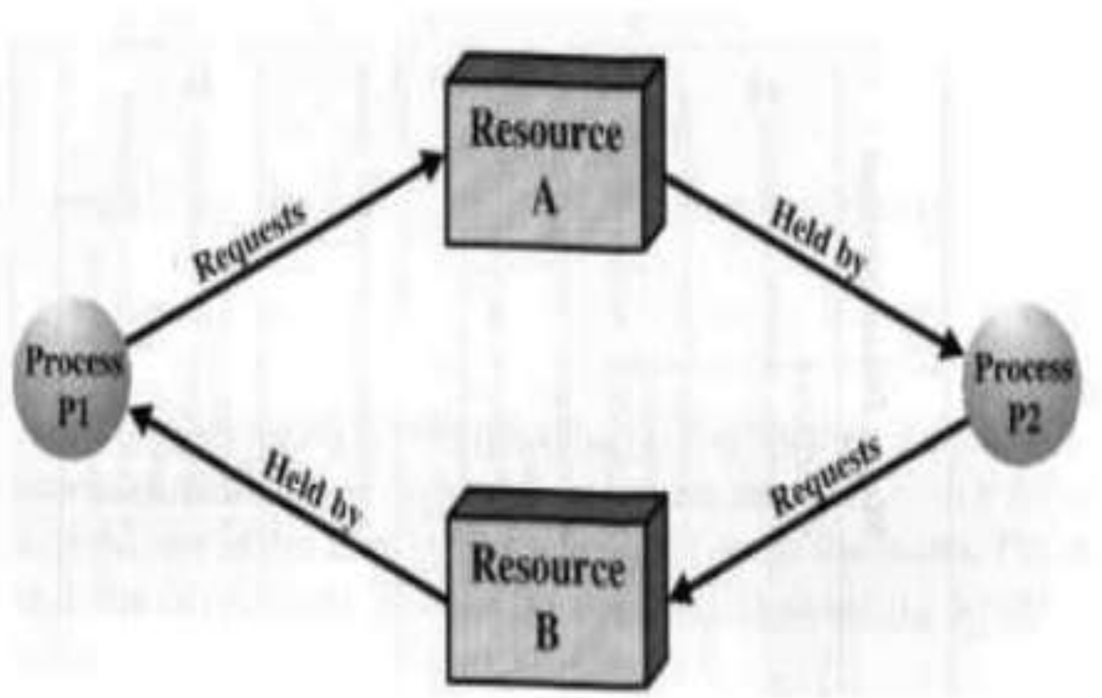


DEADLOCK condition

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

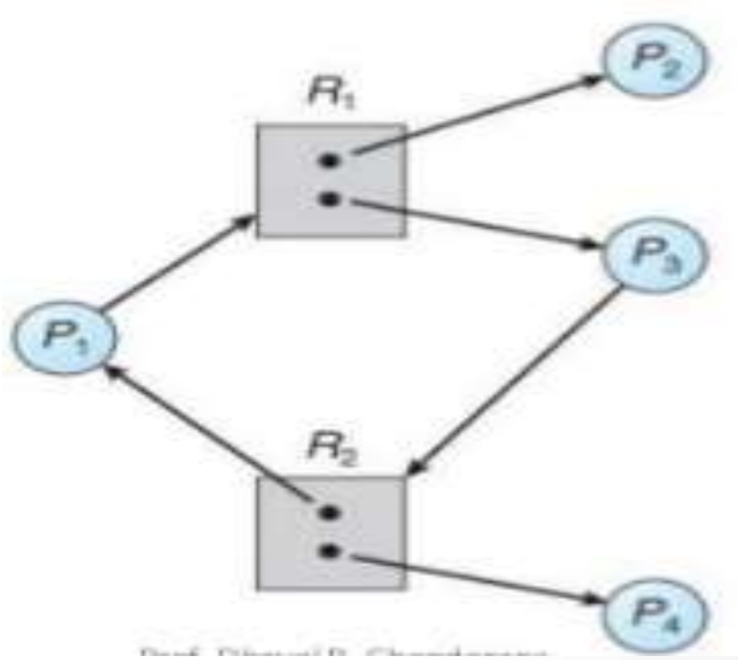


Hold and Wait: A process is holding at least one resource and waiting for resources.

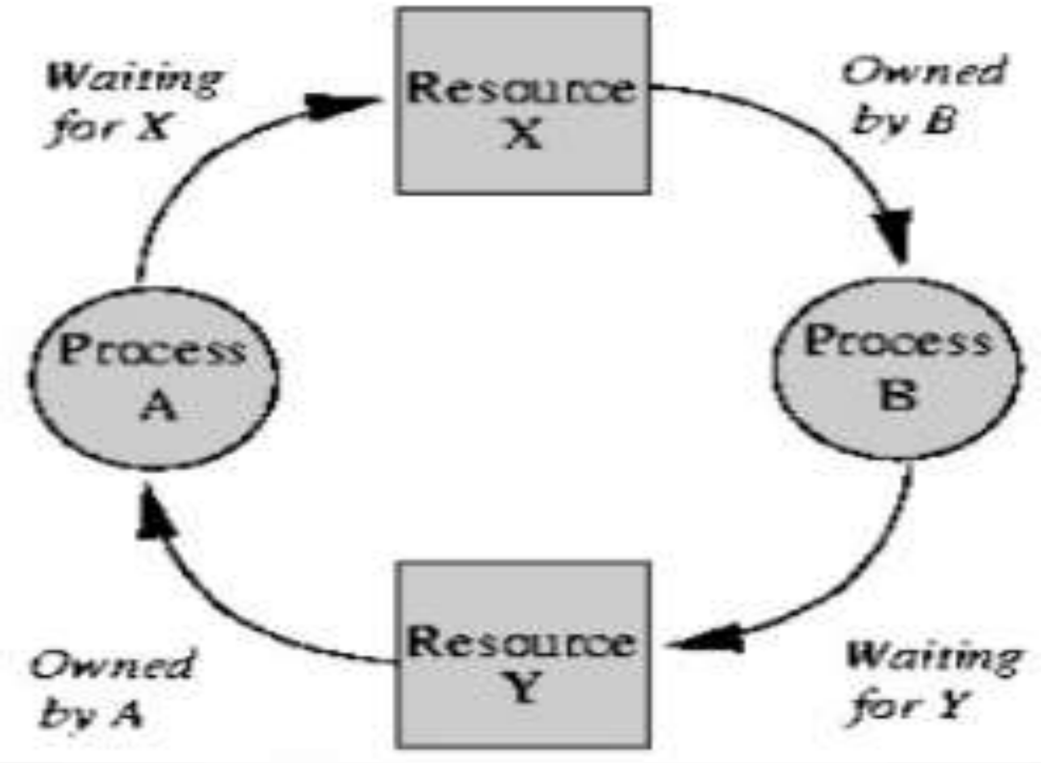


DEADLOCK condition

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



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



Methods for handling deadlock

- There are three ways to handle deadlock

  1) **Deadlock prevention or avoidance**: The idea is to not let the system into deadlock state.
- 2) **Deadlock detection and recovery**: Let deadlock occur, then do preemption to handle it once occurred.
- 3) **Ignore the problem all together**: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Deadlock recovery

- **Preemption** We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

- **Rollback** In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

- **Kill one or more processes**This is the simplest way, but it works.

Deadlock prevention

We can prevent Deadlock by eliminating any of the above four condition.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tap drive and printer, are inherently non-shareable.

Eliminate Hold and wait

1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.
2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.

Eliminate No Preemption

Preempt resources from process when resources required by other high priority process.

Eliminate Circular Wait

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

Banker’s Algorithm

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource type that it needs.

Deadlock Avoidance

Let us assume that there are **n** processes and **m** resource types. Some data structures are used to implement the banker's algorithm. They are:

Available: It is an array of length **m**. It represents the number of available resources of each type. If Available[j] = k, then there are **k** instances available, of resource type **Rj**.

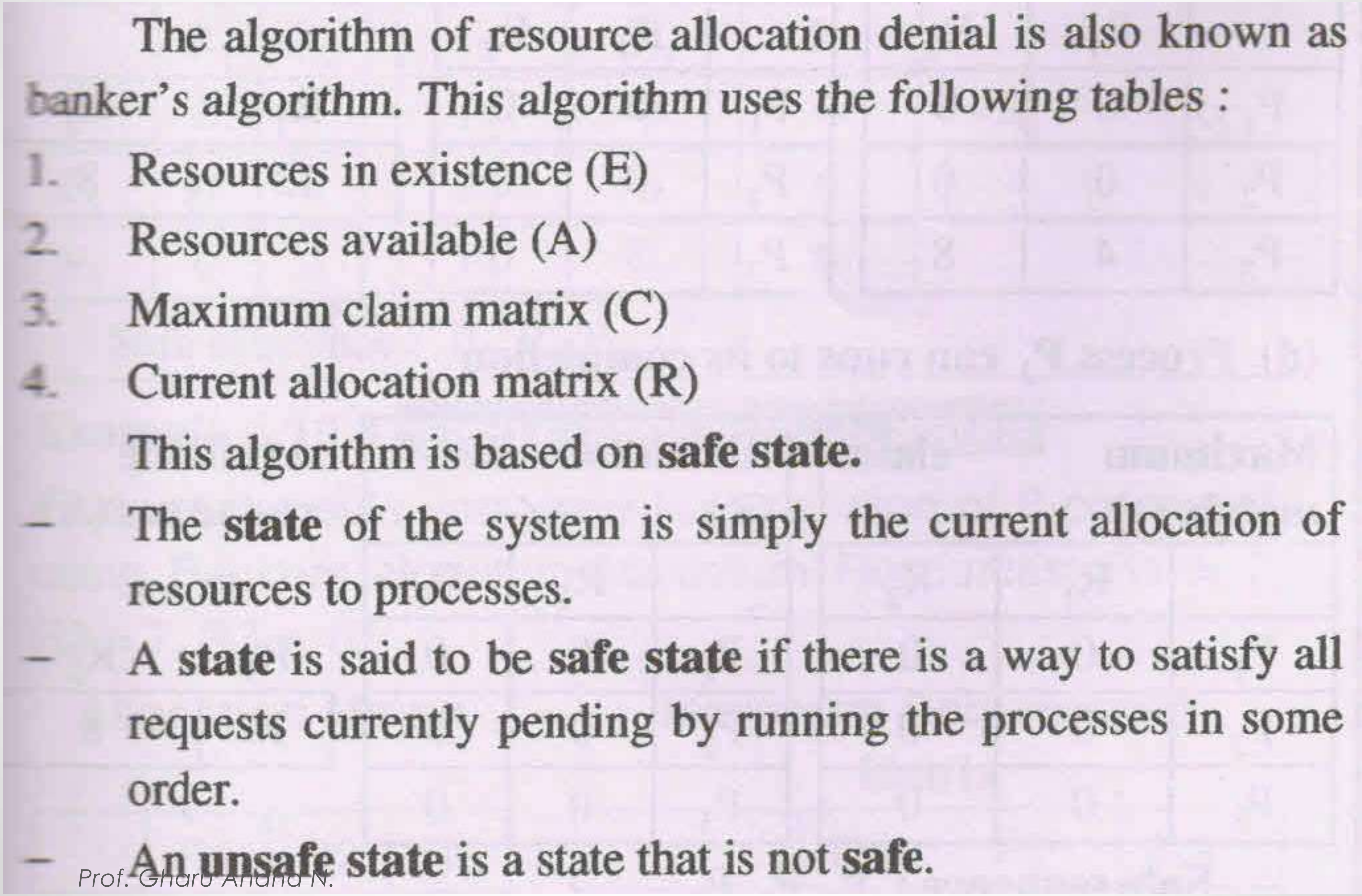
Max: It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process **Pi** can request atmost **k** instances of resource type **Rj**.3

Allocation: It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process **Pi** is currently allocated **k** instances of resource type **Rj**.

Need: It is an **n x m** matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

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

Bankers Algorithms



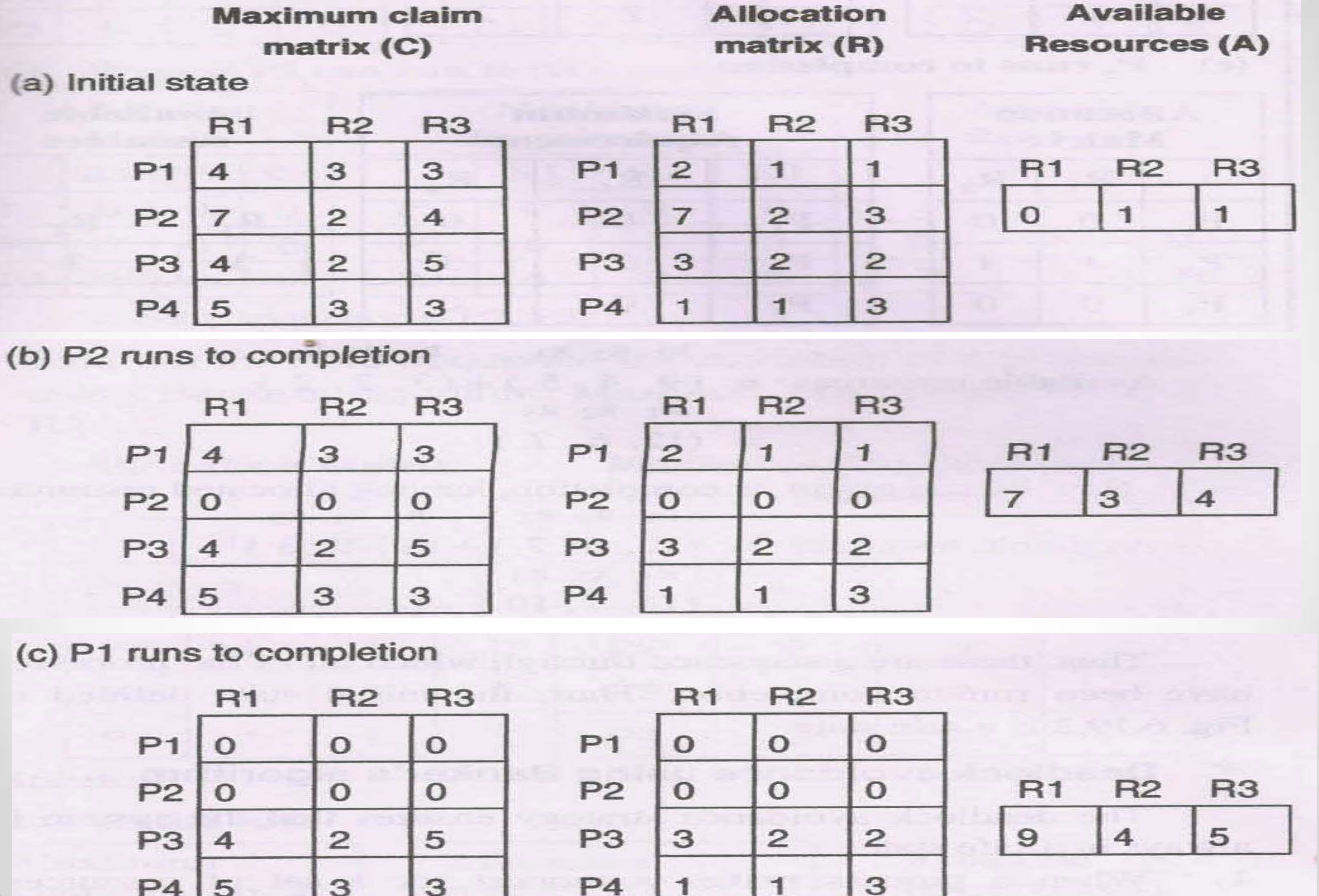
The algorithm of resource allocation denial is also known as banker's algorithm. This algorithm uses the following tables :

1. Resources in existence (E)
2. Resources available (A)
3. Maximum claim matrix (C)
4. Current allocation matrix (R)

This algorithm is based on **safe state.**

- The **state** of the system is simply the current allocation of resources to processes.
- A **state** is said to be **safe state** if there is a way to satisfy all requests currently pending by running the processes in some order.
- An **unsafe state** is a state that is not **safe.**

Bankers Algorithms



Maximum claim matrix (C) | Allocation matrix (R) | Available Resources (A)

(a) Initial state

Maximum claim matrix (C)

	R1	R2	R3
P1	4	3	3
P2	7	2	4
P3	4	2	5
P4	5	3	3

Allocation matrix (R)

	R1	R2	R3
P1	2	1	1
P2	7	2	3
P3	3	2	2
P4	1	1	3

Available Resources (A)

R1	R2	R3
0	1	1

(b) P2 runs to completion

	R1	R2	R3
P1	4	3	3
P2	0	0	0
P3	4	2	5
P4	5	3	3

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

R1	R2	R3
7	3	4

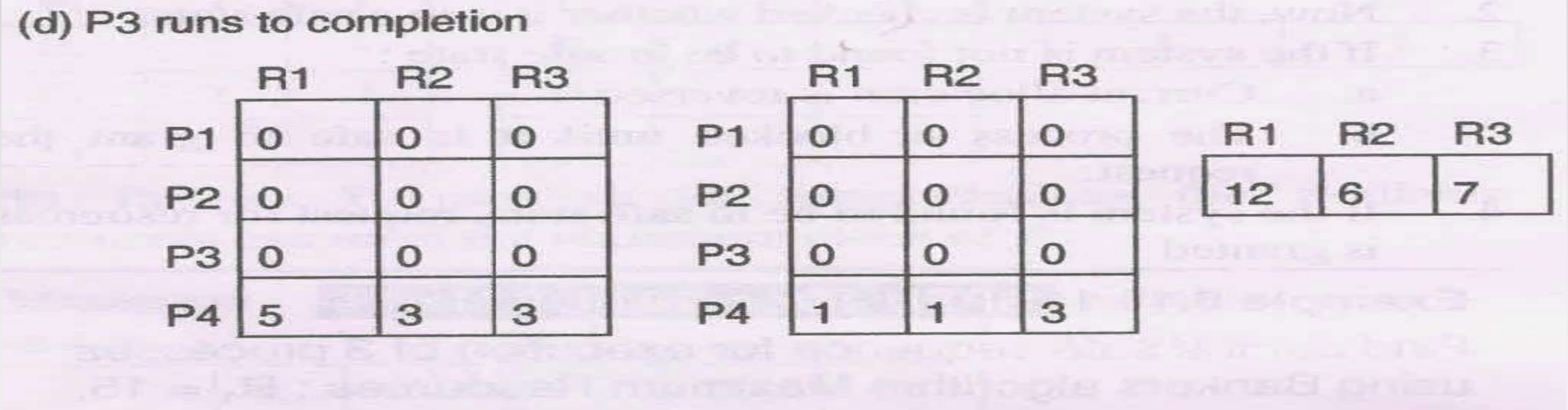
(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	4	2	5
P4	5	3	3

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

R1	R2	R3
9	4	5

Bankers Algorithms

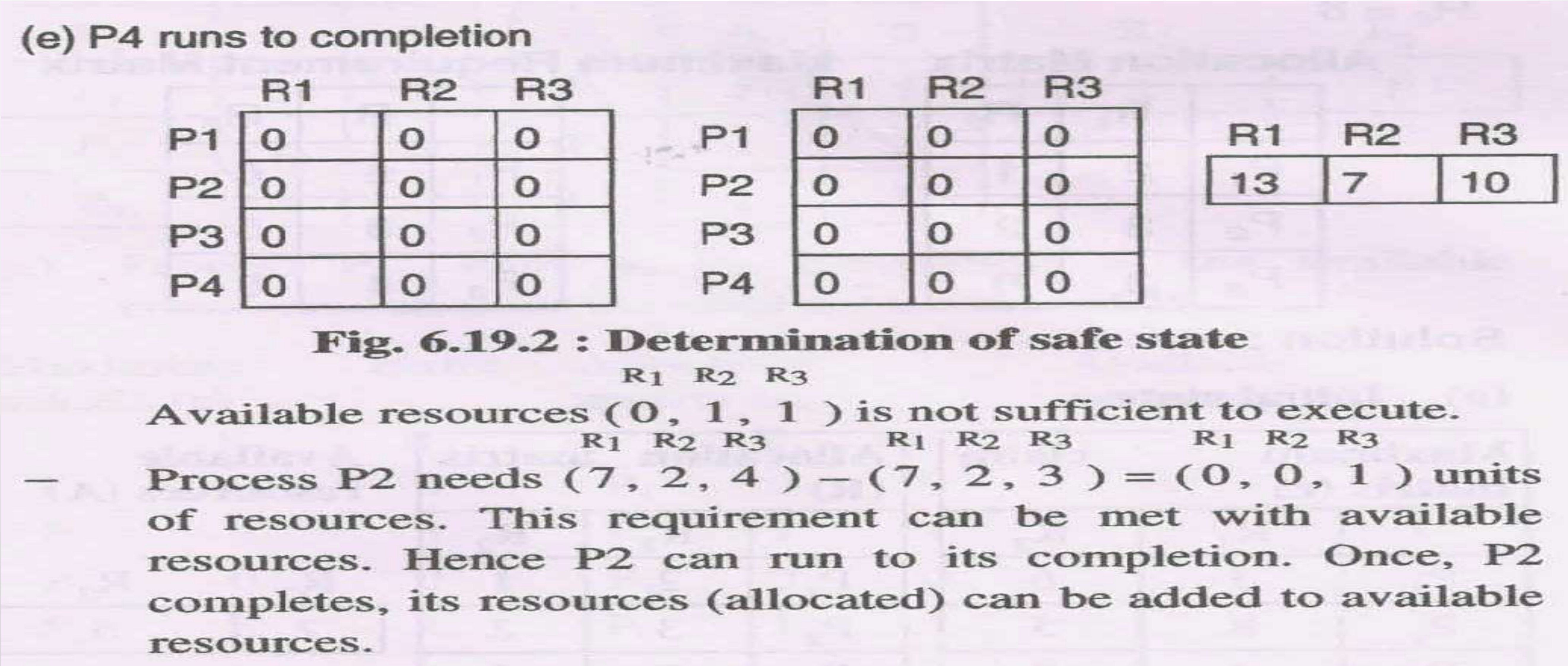


(d) P3 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	5	3	3

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

R1	R2	R3
12	6	7



(e) P4 runs to completion

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

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

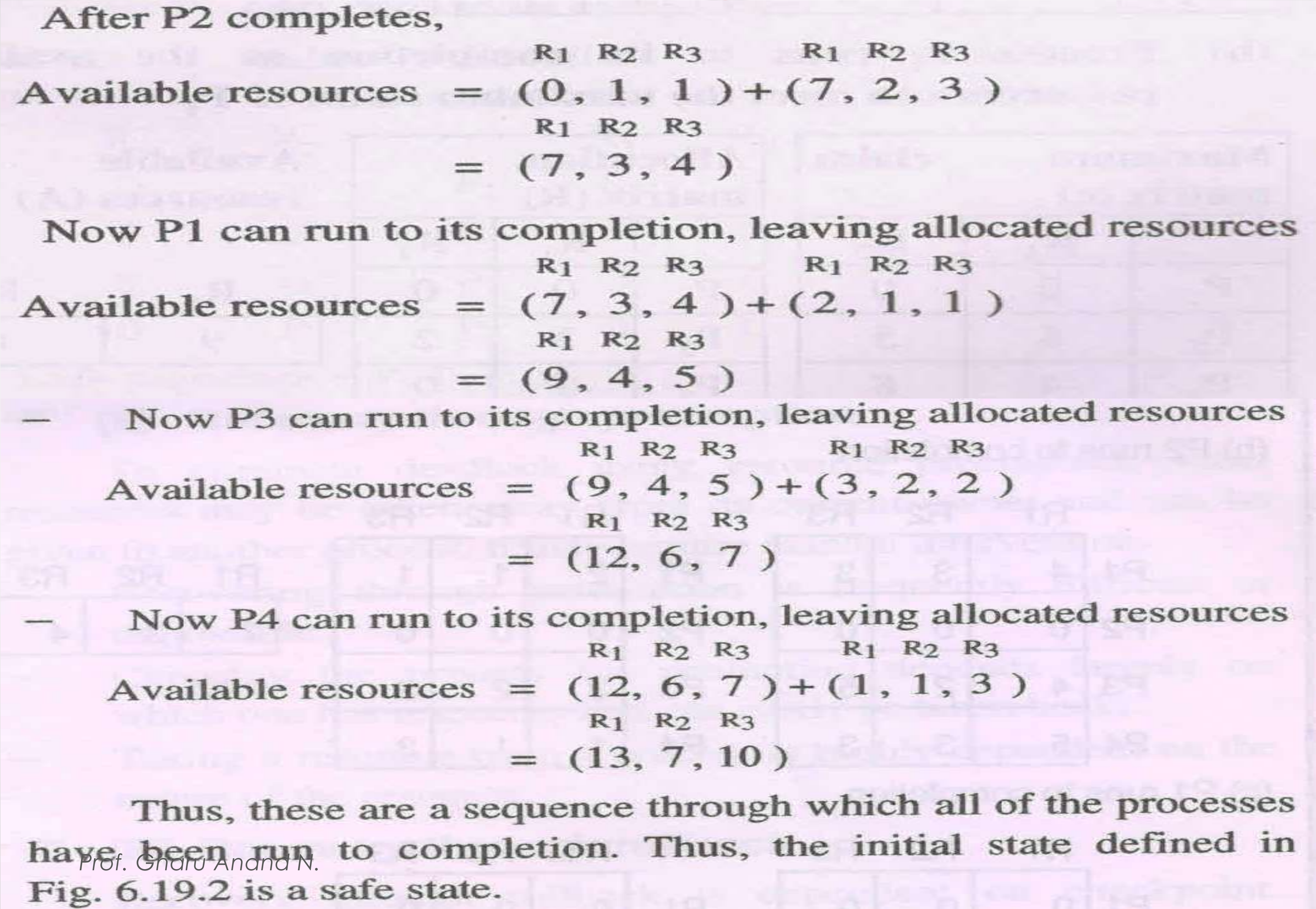
R1	R2	R3
13	7	10

Fig. 6.19.2 : Determination of safe state

Available resources $(\overset{R1}{0}, \overset{R2}{1}, \overset{R3}{1})$ is not sufficient to execute.

— Process P2 needs $(\overset{R1}{7}, \overset{R2}{2}, \overset{R3}{4}) - (\overset{R1}{7}, \overset{R2}{2}, \overset{R3}{3}) = (\overset{R1}{0}, \overset{R2}{0}, \overset{R3}{1})$ units of resources. This requirement can be met with available resources. Hence P2 can run to its completion. Once, P2 completes, its resources (allocated) can be added to available resources.

Bankers Algorithms



After P2 completes,

$$\text{Available resources} = \begin{array}{ccc} R_1 & R_2 & R_3 \\ (0, & 1, & 1\ ) \end{array} + \begin{array}{ccc} R_1 & R_2 & R_3 \\ (7, & 2, & 3\ ) \end{array}$$

$$= \begin{array}{ccc} R_1 & R_2 & R_3 \\ (7, & 3, & 4\ ) \end{array}$$

Now P1 can run to its completion, leaving allocated resources

$$\text{Available resources} = \begin{array}{ccc} R_1 & R_2 & R_3 \\ (7, & 3, & 4\ ) \end{array} + \begin{array}{ccc} R_1 & R_2 & R_3 \\ (2, & 1, & 1\ ) \end{array}$$

$$= \begin{array}{ccc} R_1 & R_2 & R_3 \\ (9, & 4, & 5\ ) \end{array}$$

— Now P3 can run to its completion, leaving allocated resources

$$\text{Available resources} = \begin{array}{ccc} R_1 & R_2 & R_3 \\ (9, & 4, & 5\ ) \end{array} + \begin{array}{ccc} R_1 & R_2 & R_3 \\ (3, & 2, & 2\ ) \end{array}$$

$$= \begin{array}{ccc} R_1 & R_2 & R_3 \\ (12, & 6, & 7\ ) \end{array}$$

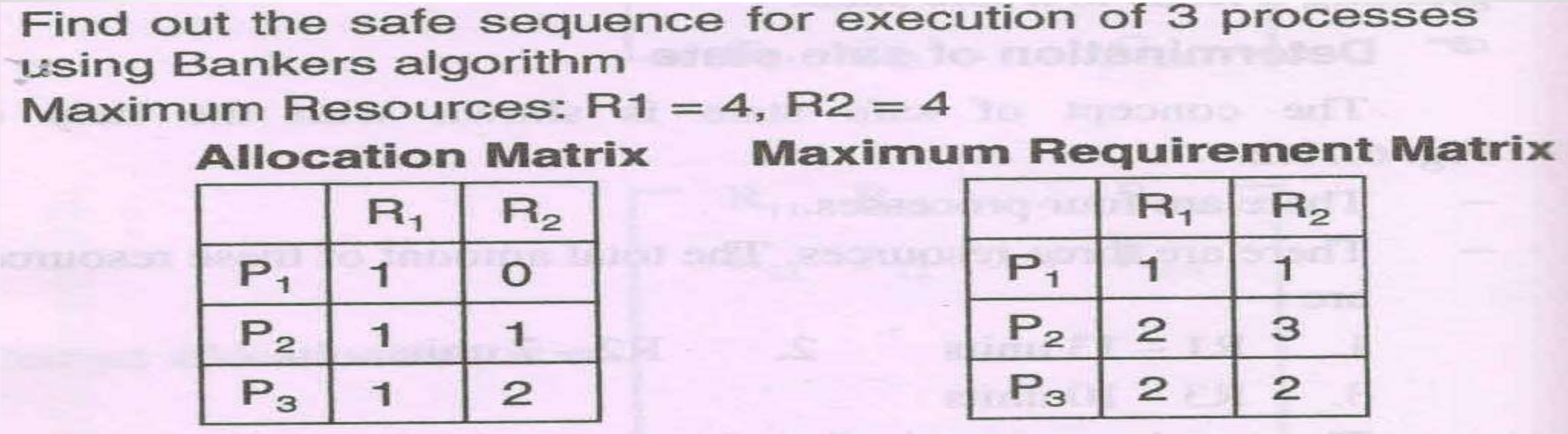
— Now P4 can run to its completion, leaving allocated resources

$$\text{Available resources} = \begin{array}{ccc} R_1 & R_2 & R_3 \\ (12, & 6, & 7\ ) \end{array} + \begin{array}{ccc} R_1 & R_2 & R_3 \\ (1, & 1, & 3\ ) \end{array}$$

$$= \begin{array}{ccc} R_1 & R_2 & R_3 \\ (13, & 7, & 10\ ) \end{array}$$

Thus, these are a sequence through which all of the processes have been run to completion. Thus, the initial state defined in Fig. 6.19.2 is a safe state.

Bankers Algorithms



Find out the safe sequence for execution of 3 processes using Bankers algorithm

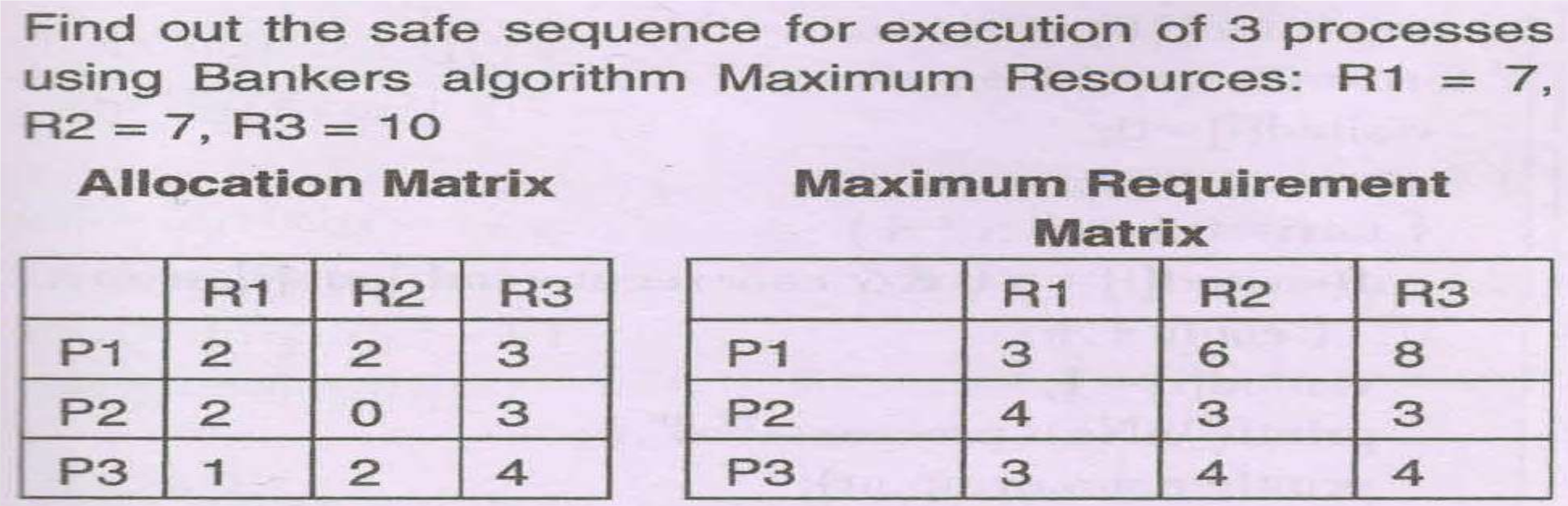
Maximum Resources: R1 = 4, R2 = 4

Allocation Matrix

	R_1	R_2
P_1	1	0
P_2	1	1
P_3	1	2

Maximum Requirement Matrix

	R_1	R_2
P_1	1	1
P_2	2	3
P_3	2	2



Find out the safe sequence for execution of 3 processes using Bankers algorithm Maximum Resources: R1 = 7, R2 = 7, R3 = 10

Allocation Matrix

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

Maximum Requirement Matrix

	R1	R2	R3
P1	3	6	8
P2	4	3	3
P3	3	4	4

THANK YOU!!!

My Blog : https://anandgharu.wordpress.com/

Email : gharu.anand@gmail.com