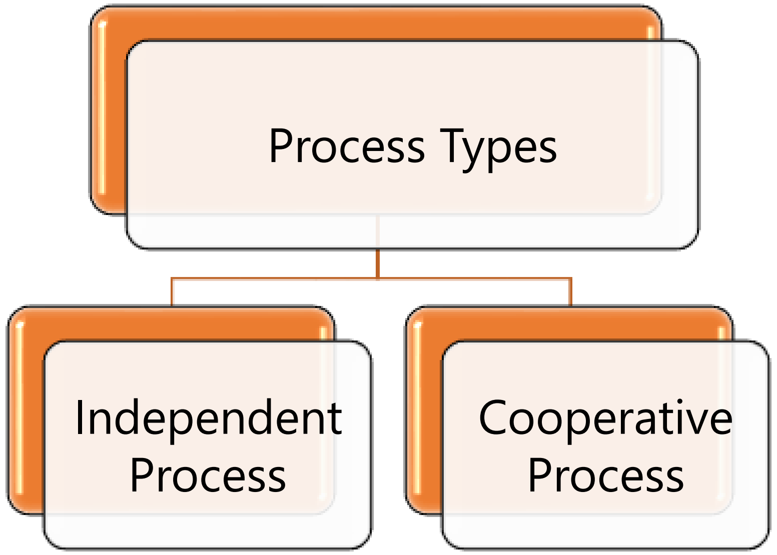
Process Synchronization

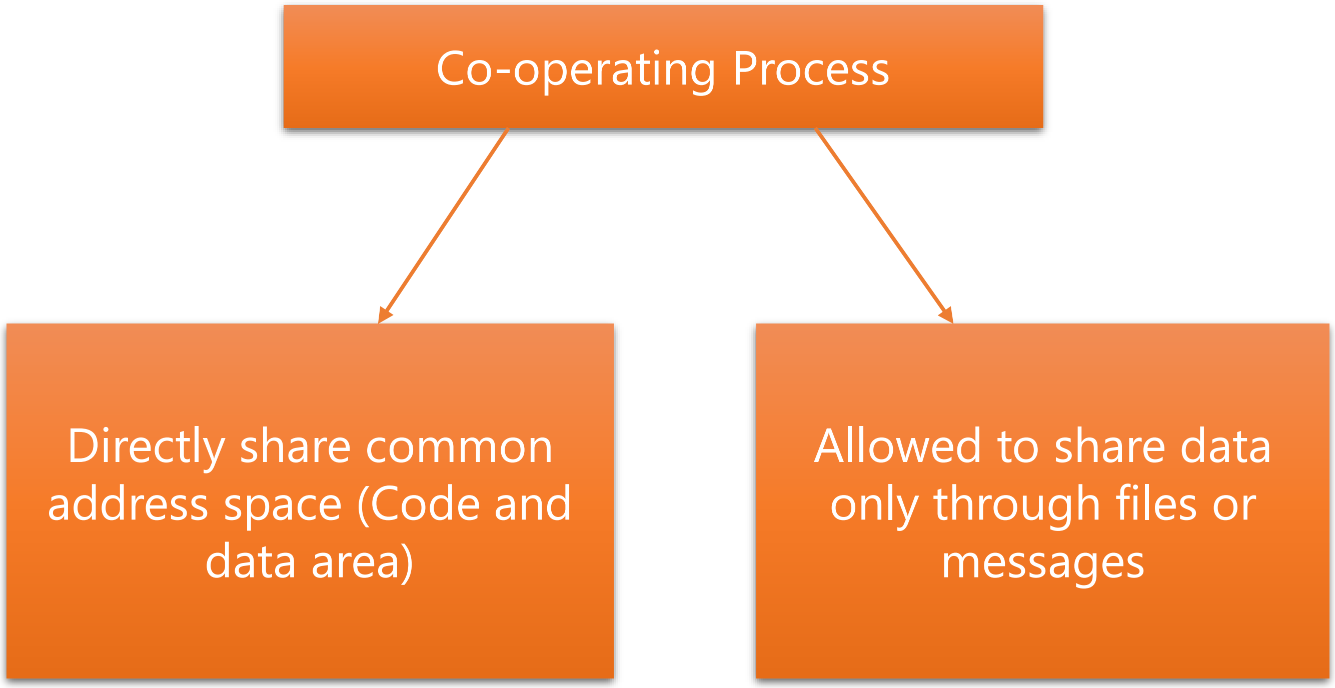
Process Types



- ✓ **Independent Process**:
  - ✓ The execution of one process does not affect the execution of other processes.
- ✓ **Cooperative Process**:
  - ✓ A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative process because resources are shared in Cooperative processes.

Cooperating Processes


Co-operating Process
Directly share common address space (Code and data area)
Allowed to share data only through files or messages

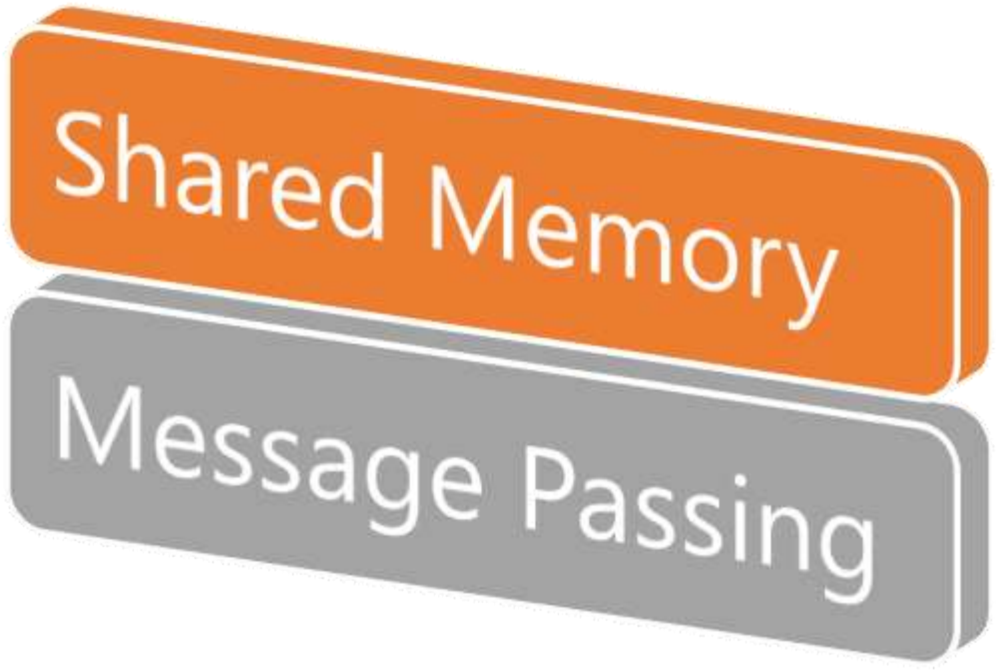
- ✓ Concurrent access to shared data may result in data inconsistency.
- ✓ Maintaining data consistency requires mechanisms to ensure the **orderly execution** of **cooperating processes**

Inter Process Communication (IPC)

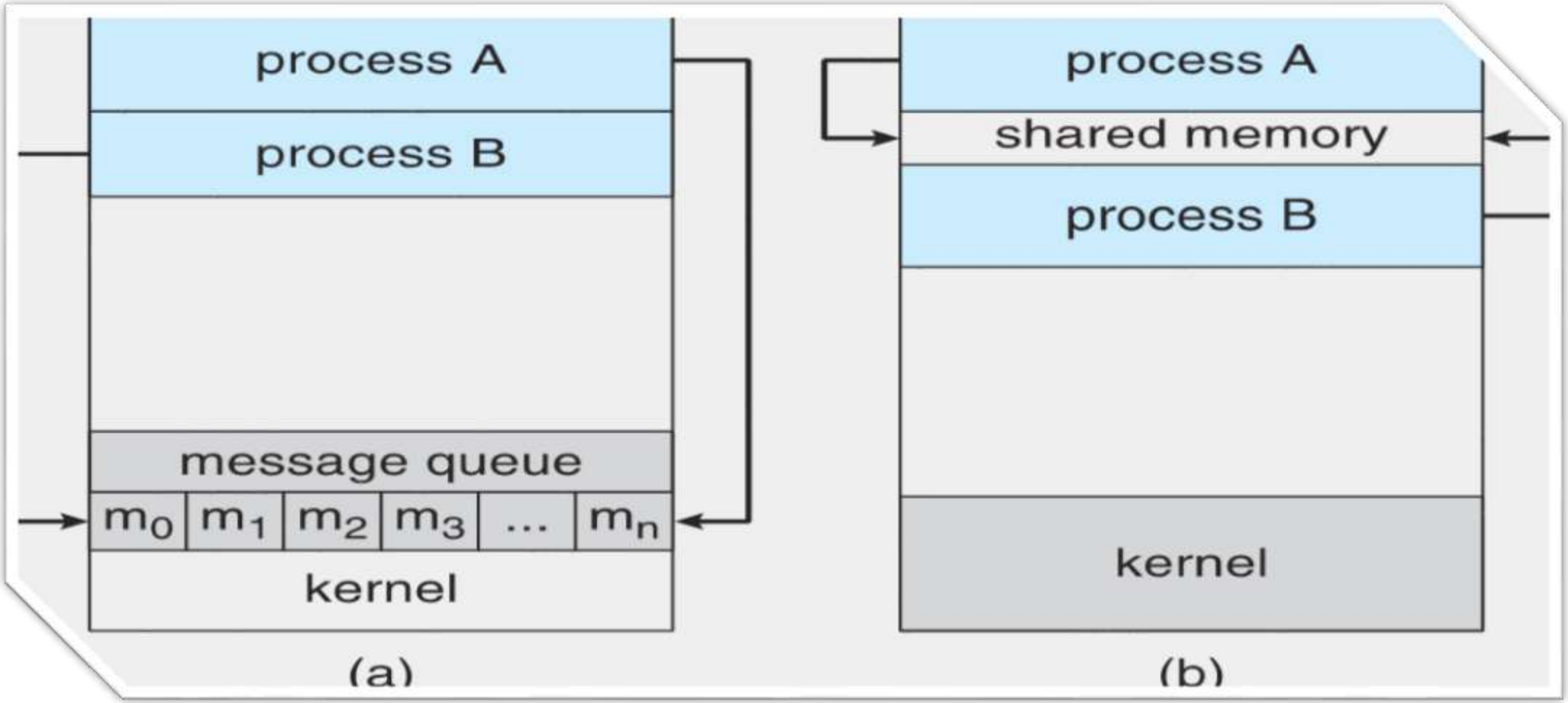
- ✓ Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions.
- ✓ The communication between these processes can be seen as a method of co-operation between them.

Inter Process Communication (IPC)

- ✓ Useful in a distributed environment
  - ✓ The processes may reside on different computers connected with a network.
- ✓ A chat program used on the World Wide Web.

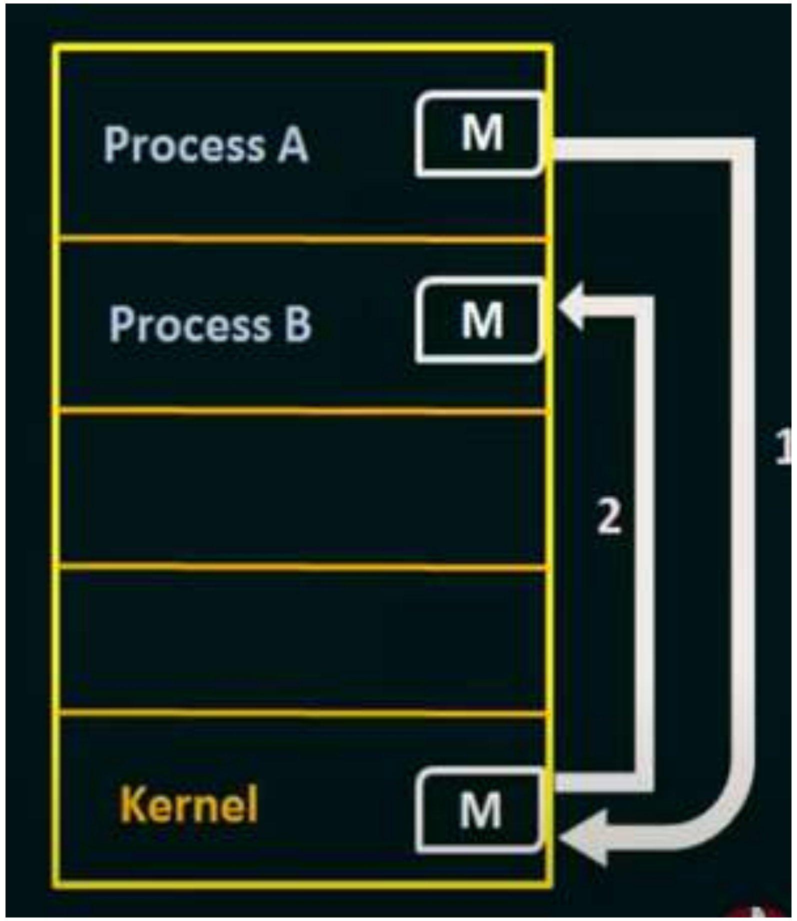


Shared Memory Vs Message Passing



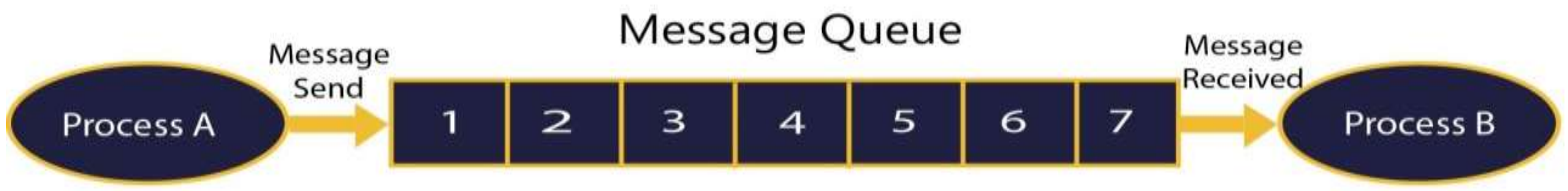
Message Passing

- Communicate and synchronous without sharing the same address space.
- Suitable if communicating processes are in different systems connected through network.



Message Passing

- Send Message
- Receive Message



Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

send(P, message) – Send a message to process P

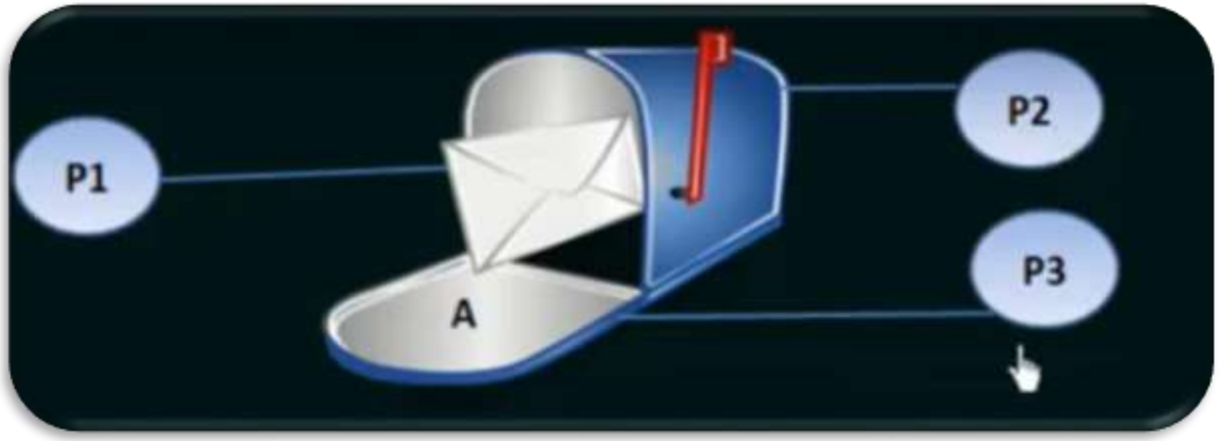
receive(Q, message) – Receive a message from process Q

Communication link in this scheme has the following properties:

A link is established automatically between every pair of processes that want to communicate.

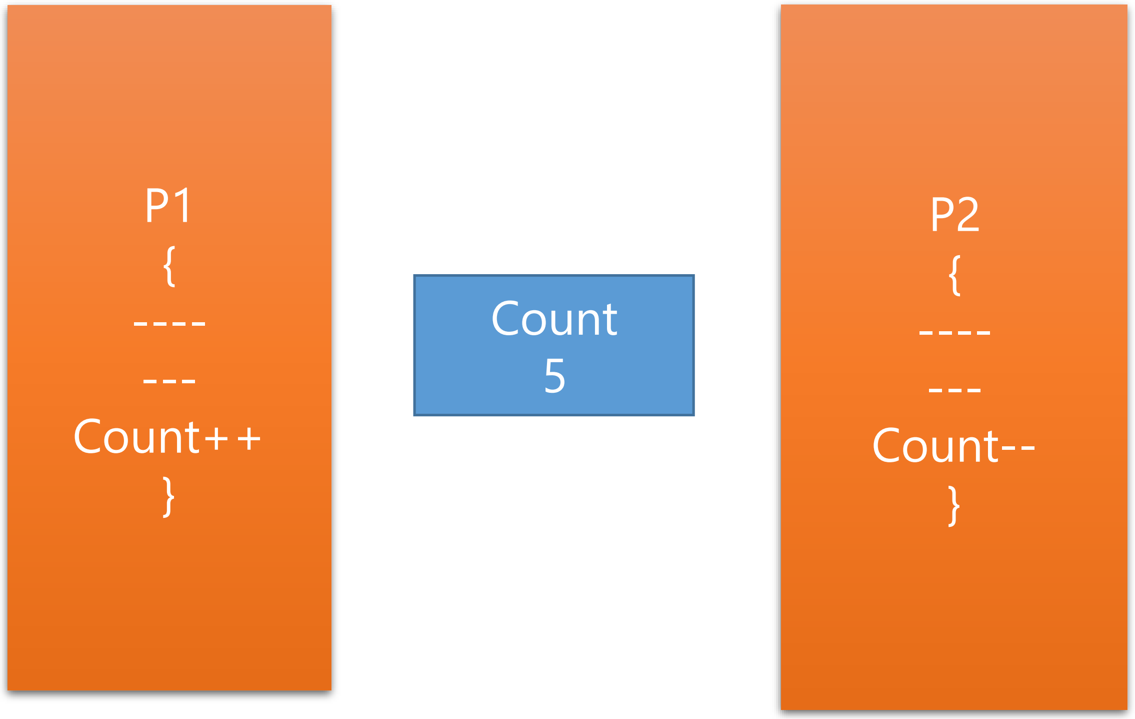
Indirect Communication

- The messages are sent to and received from mailboxes, or ports which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- A process can communicate with some other process via a number of different mailboxes.



A **mailbox** may be **owned** either by a **process** or by the **operating system.**

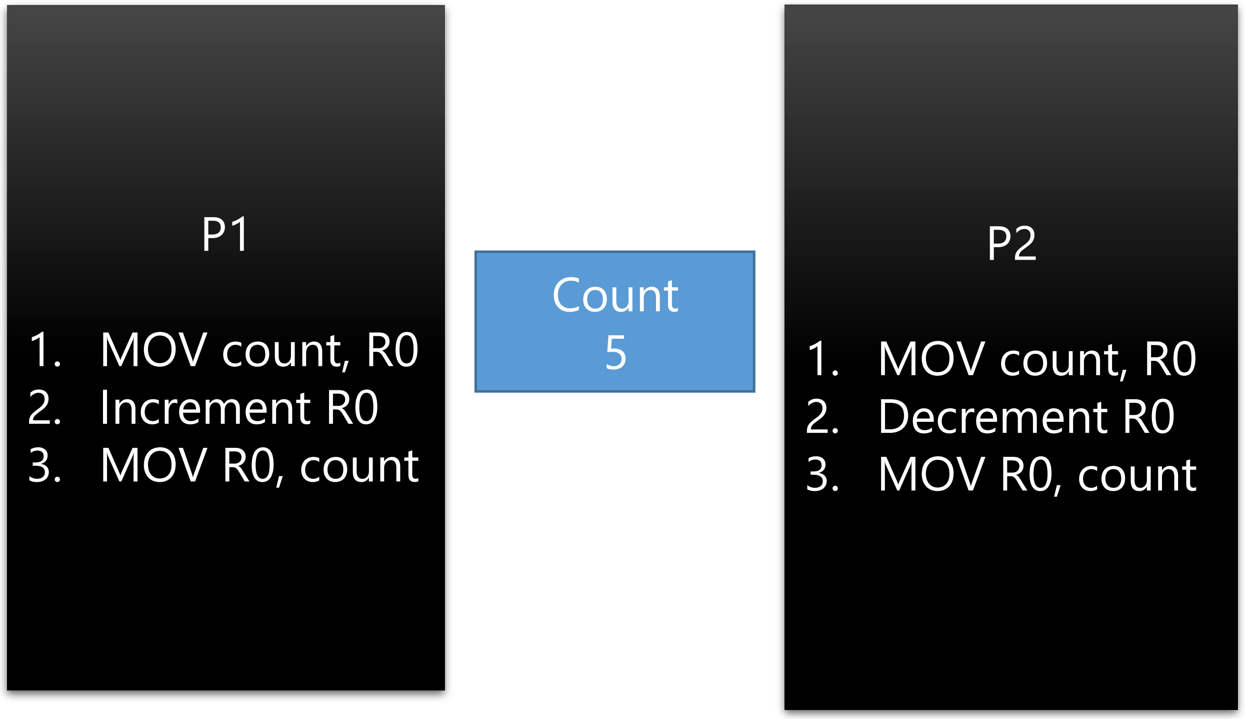
Race Condition



```
P1
{
----
---
Count++
}
```

Count
5

```
P2
{
----
---
Count--
}
```



P1

```
1. MOV count, R0
2. Increment R0
3. MOV R0, count
```

Count
5

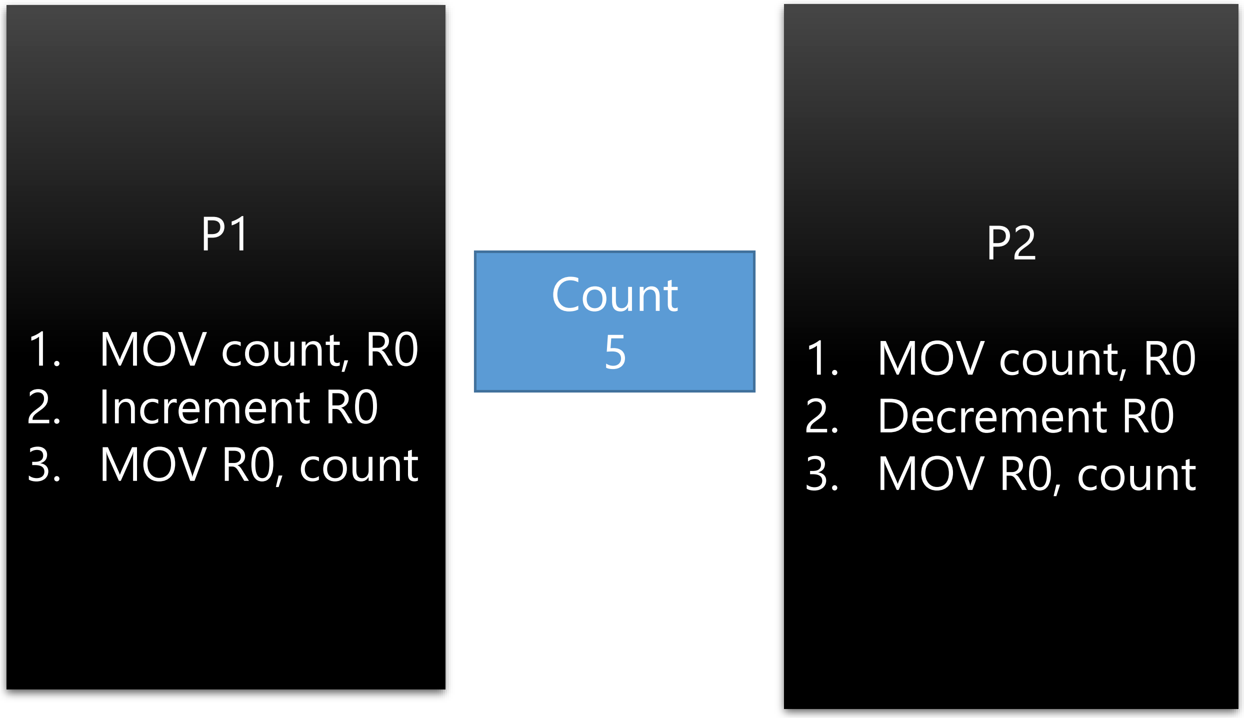
P2

```
1. MOV count, R0
2. Decrement R0
3. MOV R0, count
```

Order of execution→P1:1,2,3 P2: 1,2,3

Order of execution→P1:1,2 P2: 1,2 P1: 3 P2:3

Race Condition



P1:

```
1. MOV count, R0
2. Increment R0
3. MOV R0, count
```

P2:

```
1. MOV count, R0
2. Decrement R0
3. MOV R0, count
```

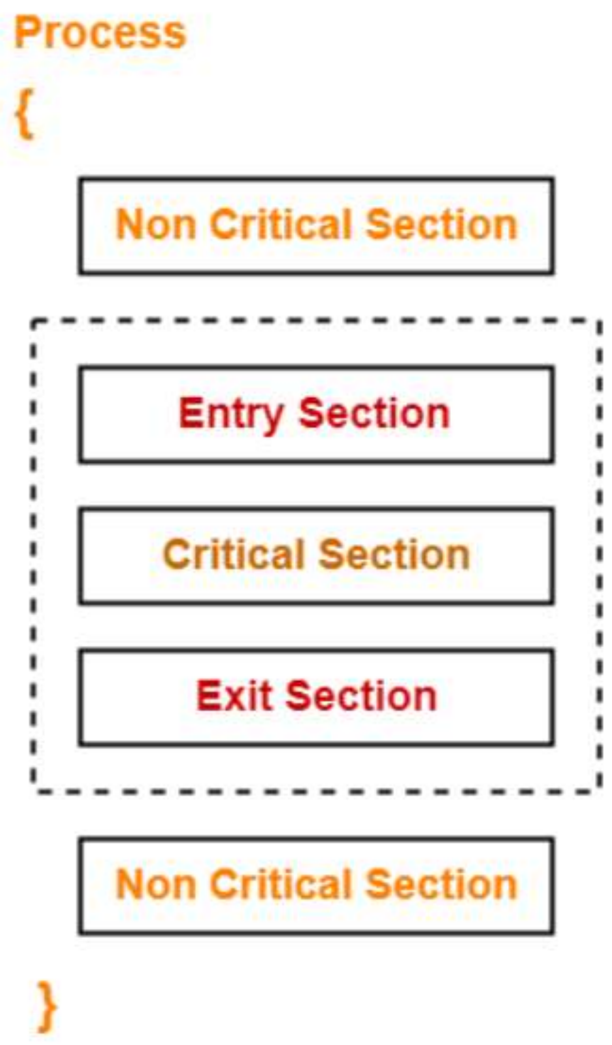
Order of execution→P1:1,2,3   P2: 1,2,3

Order of execution→P1:1,2   P2: 1,2   P1: 3   P2:3

- ✓ Final output produced depends on the order of execution.
- ✓ A situation like this, where several processes access and manipulate the same data concurrently and outcome of execution depends on the order of execution , is called **race condition**

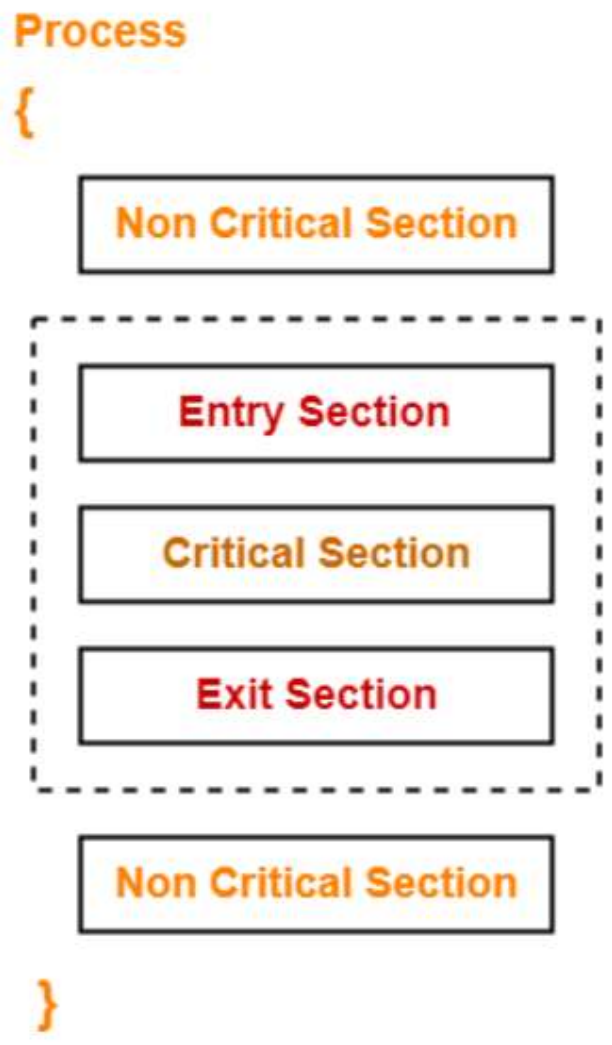
Critical Section Problem

- ✓ Consider a system containing processes {P1,P2, ….Pn}
- ✓ Each process has a segment of code called critical section.
- ✓ The code segment in critical section is responsible for making a change in common shared variables, writing to a file, updating a table etc…



Critical Section Problem

- ✓ Each process must request permission to enter its critical section.
- ✓ The section of code implementing this request is the entry section.
- ✓ Critical section is then followed by an exit section.
- ✓ The remaining code is the remainder section.



The Critical Section Problem

- ✓ **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

- ✓ **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.

The Critical Section Problem

- ✓ **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

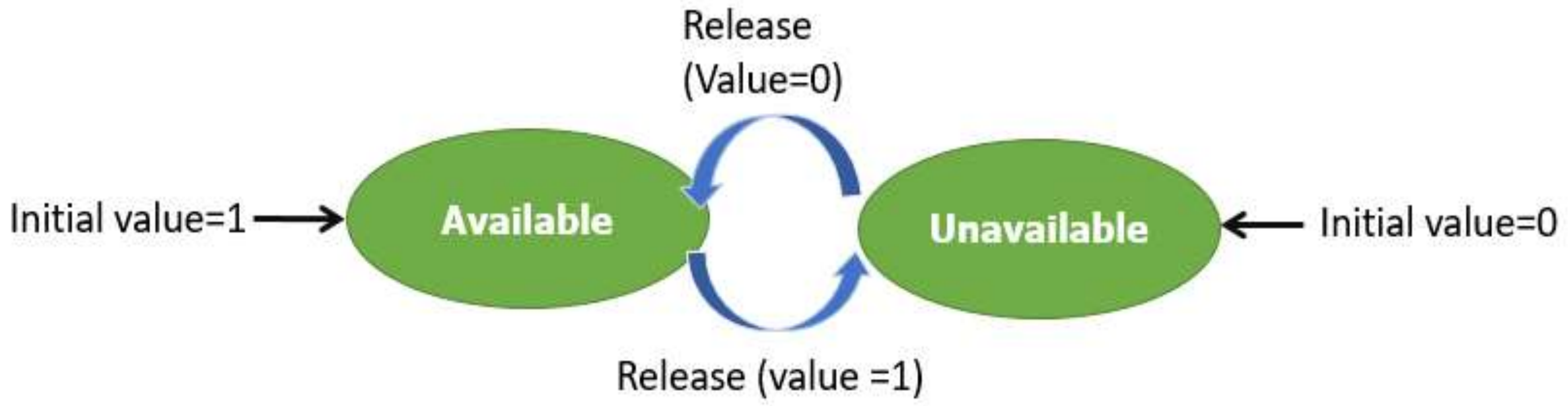
Semaphore

- ✓ Proposed by Edsger Dijikstra
- ✓ To manage concurrent processes using a single integer value.
- ✓ A non negative integer variable shared between threads/processes.
- ✓ A semaphore **'S'** is a variable and its value can be accessed only by two standard atomic operations called wait() and signal()

- ✓ wait () →P → to test
- ✓ Signal() → V → to increment

Types of semaphore

- ✓ BINARY Semaphore
  - ✓ In these type of Semaphores the integer value of the semaphore can only be either 0 or 1.



Types of semaphore

- ✓ COUNTING Semaphore
  - ✓ **Counting semaphores** are signalling integers that can take on any integer value.
  - ✓ Using these Semaphores we can coordinate access to resources and here the Semaphore count is the number of resources available.
  - ✓ **Counting semaphores are generally used when the number of instances of a resource are more than 1, and multiple processes can access the resource**.

Semaphore

- ✓ Definition of wait()

```
P ( semaphore s)
{
while (s <=0);
s--;
}
```

- ✓ Definition of signal()

```
V ( semaphore s)
{
s++;
}
```

Semaphore

- ✓ sem_t s; To declare a semaphore variable
- ✓ sem_init(sem_t *s,int pshared,unsigned int value) → To initialize a semaphore variable. (address of semaphore variable, pshared =0 if sharing in threads, 1 if sharing in processes, initial value of variable)
  - ✓ Eg: Sem_init(&s,0,1)
- ✓ sem_wait(&s) – to implement wait(s)
- ✓ sem_post(&s) - to implement dignal(s)
- ✓ Sem_destroy(&s) → to destroy semaphore variable