# INTER PROCESS COMMUNICATION(IPC)

# &

# PROCESS SYNCHRONIZATION

## Module 3 & 4

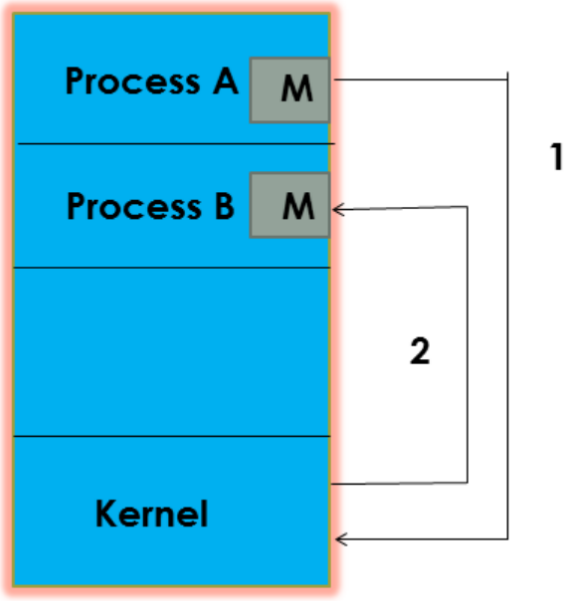
# Introduction

- Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs.
- The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.


## How to do Interprocess Communication


## 1. Message passing

- Message passing is a method of Inter Process Communication in OS.
- It involves the exchange of messages between processes, where each process sends and receives messages to coordinate its activities and exchange data with other processes.
- In message passing, each process has a unique identifier, known as a process ID, and messages are sent from one process to another using this identifier.
- When a process sends a message, it specifies the recipient process ID and the contents of the message, and the operating system is responsible for delivering the message to the recipient process. The recipient process can then retrieve the contents of the message and respond, if necessary.

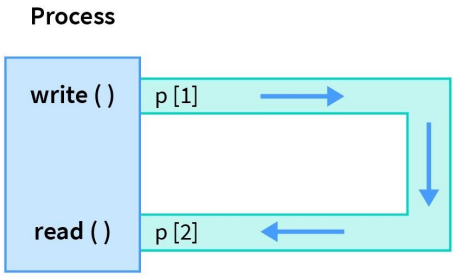



- The main advantage of message passing is that it provides a flexible and scalable method of communication between processes, as processes can communicate with each other even if they are running on different hosts or in different network domains.
- However, message passing also has some disadvantages, such as increased overhead due to the need to copy messages between address spaces, and the possibility of message loss or corruption due to network failures or other system issues. Despite these limitations, message passing remains a popular and widely used method of IPC in operating systems.

## 2. Pipes

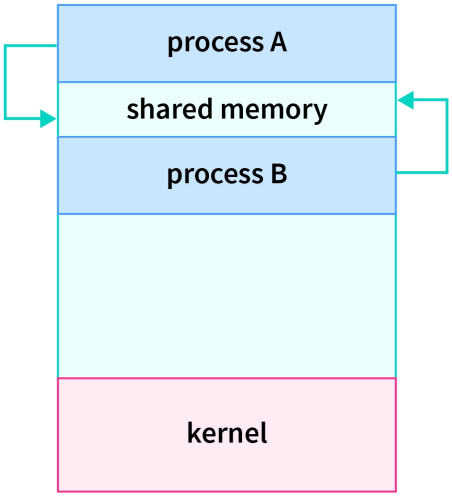
- It is a half-duplex method (or one-way communication) used for IPC between two related processes.

- It is like a scenario like filling the water with a tap into a bucket. The filling process is writing into the pipe and the reading process is retrieved from the pipe.



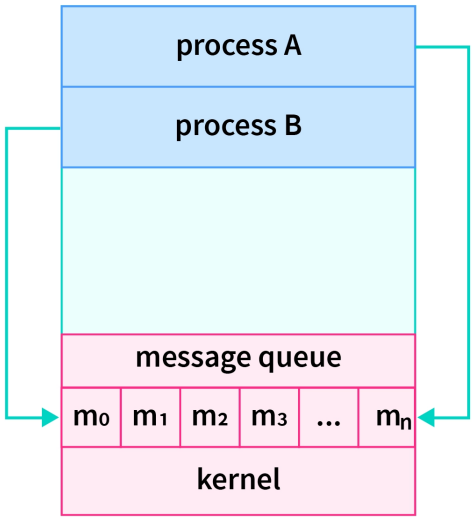

## 3. Shared Memory

- Multiple processes can access a common shared memory.
- Multiple processes communicate by shared memory, where one process makes changes at a time and then others view the change. Shared memory does not use kernel.




## 4. Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget().
- All processes can exchange information through access to a common system message queue.
- The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process.
- Each message is given an identification or type so that processes can select the appropriate message.
- Process must share a common key in order to gain access to the queue in the first place.




## 5.Direct Communication

- In this, the process that wants to communicate must name the sender or receiver.
- A pair of communicating processes must have one link between them.
- A link (generally bi-directional) is established between every pair of communicating processes

### 6. Indirect Communication

- Pairs of communicating processes have shared mailboxes.
- Link (uni-directional or bi-directional) is established between pairs of processes.
- The sender process puts the message in the port or mailbox of a receiver process and the receiver process takes out (or deletes) the data from the mailbox.

### 7. FIFO

- Used to communicate between two processes that are not related.
- Full-duplex method - Process P1 is able to communicate with Process P2, and vice versa.

## Concurrency Control

- Concurrency refers to the execution of multiple instruction sequences at the same time.
- It occurs in an operating system when multiple process threads are executing concurrently.
- These threads can interact with one another via shared memory or message passing.
- Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity.
- It aids with techniques such as process coordination, memory allocation, and execution schedule to maximize throughput.

**There are several motivations for allowing concurrent execution:**

- **Physical resource sharing :** Multiuser environment since hardware resources are limited.
- **Logical resource sharing:** Shared file(same piece of information).
- **Computation speedup:** Parallel execution
- **Modularity:** Divide system functions into separation processes.

## Issues of Concurrency

Various issues of concurrency are as follows:

**1. Non-atomic**

Operations that are non-atomic but interruptible by several processes may happen issues. A non-atomic operation depends on other processes, and an atomic operation runs independently of other processes.

**2. Deadlock**

In concurrent computing, it occurs when one group member waits for another member, including itself, to send a message and release a lock. Software and hardware locks are commonly used to arbitrate shared resources and implement process synchronization in parallel computing, distributed systems, and multiprocessing.

**3. Blocking**

A blocked process is waiting for some event, like the availability of a resource or completing an I/O operation. Processes may block waiting for resources, and a process may be blocked for a long time waiting for terminal input. If the process is needed to update some data periodically, it will be very undesirable.

### 4. Race Conditions

A race problem occurs when the output of a software application is determined by the timing or sequencing of other uncontrollable events. Race situations can also happen in multithreaded software, runs in a distributed environment, or is interdependent on shared resources.

### 5. Starvation

A problem in concurrent computing is where a process is continuously denied the resources it needs to complete its work. It could be caused by errors in scheduling or mutual exclusion algorithms, but resource leaks may also cause it.

## Process Synchronization

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner.
- It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access.
- To achieve this, **various synchronization techniques such as semaphores, monitors, and critical sections are used.**
- In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems.

- Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

**On the basis of synchronization, processes are categorized as one of the following two 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 processes also because resources are shared in Cooperative processes.

## Race Condition

- In multithreaded programs when multiple threads or processes access shared data or resources concurrently, the final outcome of the program depends on the timing or order of execution of these threads or processes.
- In other words, a race condition arises when the behavior of a program becomes unpredictable and unintended due to the unpredictable interleaving of multiple threads or processes.
- A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute.
- Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction.

**Example:**

- Let's say there are two processes P1 and P2 which share common variable (shared=10).
- Suppose, Process P1 first come under execution, and CPU store common variable between them (shared=10) in local variable (X=10) and increment it by 1(X=11) , after then when CPU read line sleep(1) ,it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in waiting state for 1 second .
- Now CPU execute the Process P2 line by line and store common variable (Shared=10) in its local variable (Y=10) and decrement Y by 1(Y=9),after then when CPU read sleep(1) , the current process P2 goes in waiting state and CPU remains idle for sometime as there is no process in ready-queue, after completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code (store the local variable (X=11) in common variable (shared=11) ),CPU remain idle for sometime waiting for any process in ready-queue ,after completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2(store the local variable (Y=9) in common variable (shared=9) ).

**Initially Shared = 5**

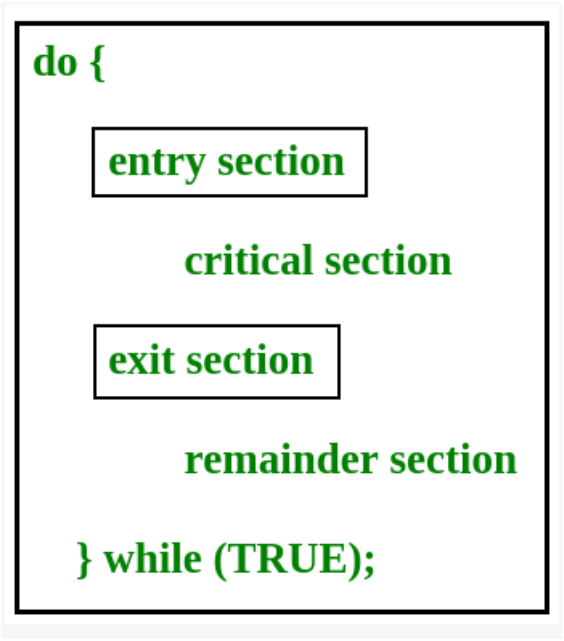
| Process 1 | Process 2 |
|---|---|
| int X = shared | int Y = shared |
| X++ | Y- - |

| | |
|---|---|
| sleep(1) | sleep(1) |
| shared = X | shared = Y |

**Critical Sections:** **Critical sections are regions of code where shared resources are accessed and modified.** To prevent race conditions, developers use synchronization techniques like locks, semaphores, or mutexes to protect critical sections, ensuring that only one thread or process can access the shared resource at a time.

## Critical Section Problem

- A critical section is a code segment that can be accessed by only one process at a time.
- The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables.
- **So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.**




In the entry section, the process requests for entry in the Critical Section.

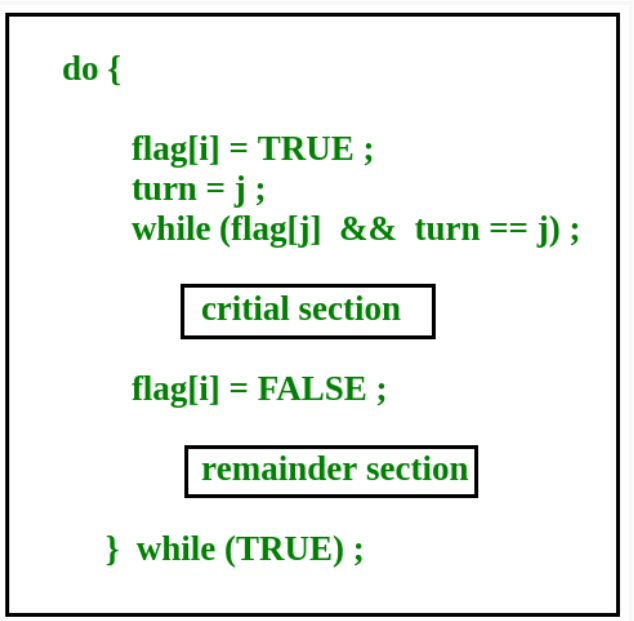
Any solution to the critical section problem must satisfy three requirements:

- **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.
- **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.

## Peterson's Solution

- Peterson's Solution is a classical software-based solution to the critical section problem.

- Peterson's solution is used to provide the operating system with a functionality called mutual exclusion, where two concurrent processes can share the same resource with any problem or collision.
- In Peterson's solution, we have two shared variables:
  - boolean **flag**[i]: Initialized to FALSE, initially no one is interested in entering the critical section
  - int **turn**: The process whose turn is to enter the critical section.




1. At first, both processes set their flag variables to indicate that they don't currently want to enter the critical section. By default, the turn variable is set to the ID of one of the processes, it can be 0 or 1. This will indicate that it is initially the turn of that process to enter the critical section.
2. Both processes set their flag variable to indicate their intention to enter the critical section.
3. Then the process, which is having the next turn, sets the turn variable to its own ID. This will indicate that it is its turn to enter the critical section. For

example, if the current value of turn is 1, and it is now the turn of Process 0 to enter, Process 0 sets turn = 0.

4. Both processes enter a loop where they will check the flag of the other process and wait if necessary. The loop continues as long as the conditions are met.
5. If both conditions are satisfied, the process waits and yields the CPU to the other process. This ensures that the other process has an opportunity to enter the critical section.
6. Once a process successfully exits the waiting loop, then it can enter the critical section. It can also access the shared resource without interference from the other process. It can perform any necessary operations or modifications within this section.
7. After completing its work in the critical section, the process resets its flag variable. Resetting is required to indicate that this process no longer wants to enter the critical section (flag[i] = false). This step ensures that the process can enter the waiting loop again correctly if needed.

## Disadvantages of Peterson's Solution

- It involves **busy waiting**.(In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- (Busy waiting, also known as spinning, or busy looping is a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution. In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

# Semaphores

- Semaphore is an integer variable used in mutually exclusive manner by various cooperative processes in order to achieve synchronization.
- To overcome the Critical section problem, we can use the synchronization tool, semaphore.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait and signal.**
- **Operations on Semaphore**
  - **wait() or down or P()** — ***Placed at Entry Code***

    The operation to reduce the value of semaphore by 1
  - **signal() or UP or V() or post or Release** — ***Placed at Exit Code***

    The operation to increase the value of semaphore by 1

**The classical definition of wait in pseudocode:**

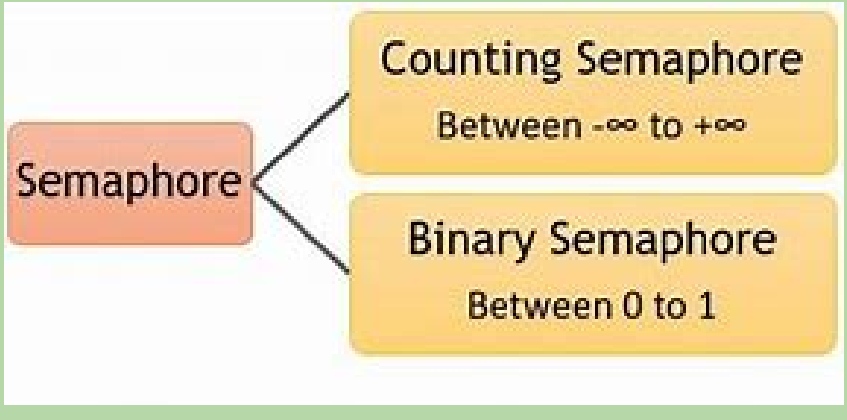
```
wait(S) {
while (S <= 0);
S --;
}
```

**The classical definition of signal in pseudocode:**

```
Signal(S){
S++;
}
```

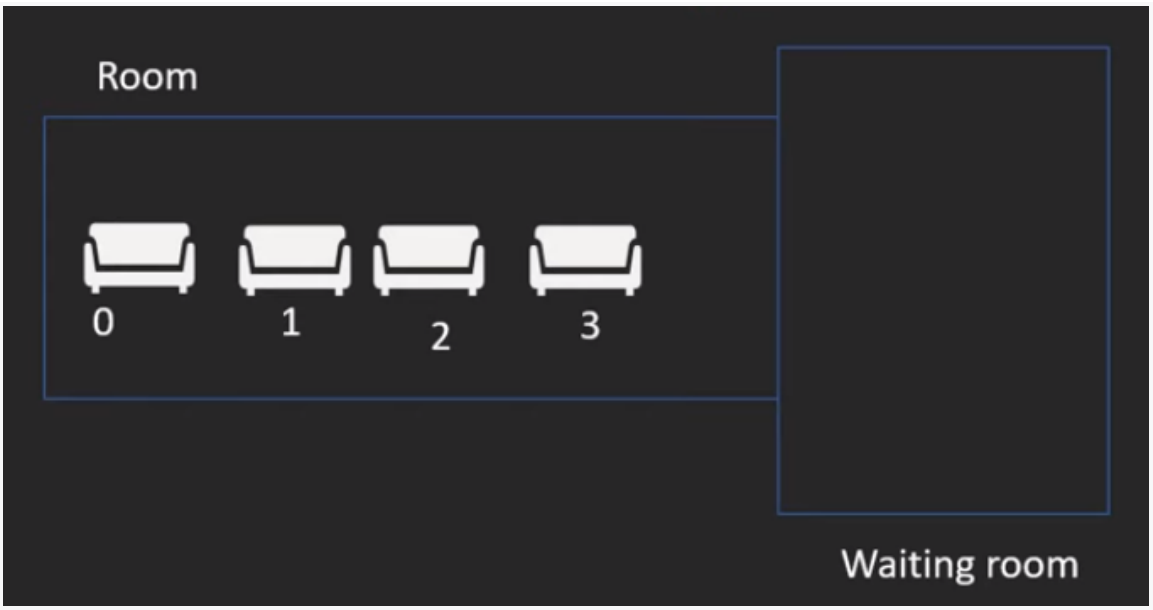
- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

## Types of semaphores




### Counting Semaphores

- Counting semaphore can be used when we need to have more than one process in the critical section at the same time.
- The value of the counting semaphore can be positive or negative.
- The positive value indicates the number of processes that can be present in the critical section at the same time.
- The negative value indicates the number of processes that are blocked in the waiting list.




- Here, the number of chairs in the room is 4.
- So, the number of people that the room can occupy is 4.
- When the 5th person comes, then he has to wait in the waiting room.
- After the use of any of the person, 1 chair will be empty, then, the 5th person can sit in the chair.

- The semaphore can be initialized to the number of instances of the resource.
- Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available.
- Then, the process can **enter** its critical section thereby **decreasing the value of the counting semaphore by 1.**
- After the process is over with the use of the instance of the resource, it can **leave** the critical section thereby **adding 1** to the number of available instances of the resource.

## Counting semaphore implementation

Entry code

```
Wait (Semaphore S)
{
        S.value=S.value-1;
        if(S.value<0)
        {
                put process(PCB) in suspend list
                sleep();
        }
        else
        return;
}
```

S= 3

exit code

```
Signal(Semaphore S)
{
        S.value=S.value+1;
        if(S.value<= 0)
        {
                Select process from suspend list
                wakeup();
        }
}
```

## Binary Semaphores

- Binary semaphore strictly provides mutual exclusion, also known as **mutex lock.**
- It allows only 1 process in the critical section(ie, mutual exclusion).
- Semaphore can only be either 0 or 1.

**Critical Section**

- Let, a process P1, wants to enter into the Critical section.
- It will check the value of Semaphore.

**If (Value of Semaphore==1)**
**then Enter into the critical section**

- The P1 can't enter into the Critical section,

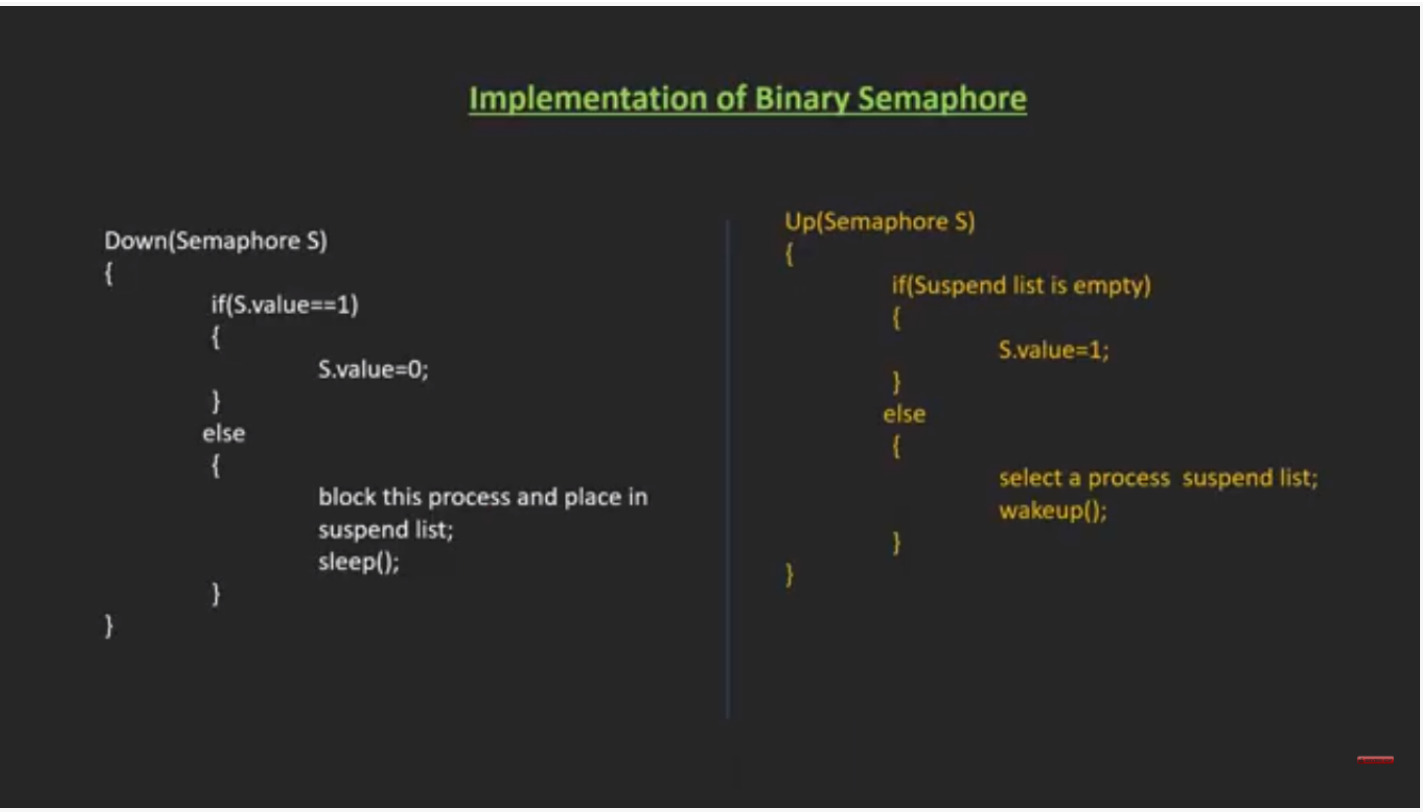
**If (Value of Semaphore==0)**

  - P1 has to wait until the lock becomes 0.

- All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0.
- Then, the process can make the mutex semaphore 1 and start its critical section.
- When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.

**Mutex**

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for Mutual Exclusion Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.




## Bakery Algorithm

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Bakery Algorithm is a critical section solution for **N** processes. The algorithm preserves the first come first serve property.

- **How does the Bakery Algorithm work?**
  In the Bakery Algorithm, each process is assigned a number (a ticket) in a lexicographical order.
- Before entering the critical section, a process receives a ticket number, and the process with the smallest ticket number enters the critical section.
- If two processes receive the same ticket number, the process with the lower process ID is given priority.
- The Bakery Algorithm ensures fairness by assigning a unique ticket number to each process based on a lexicographical order.
- This ensures that processes are served in the order they arrive, which guarantees that all processes will eventually enter the critical section.

- Before entering its critical section, the process receives a number. Holder of the smallest number enters the critical section.
- If processes Pi and Pj receive the same number,

```
if i < j
Pi is served first;
else
Pj is served first.
```

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5, ...

**Notation** –lexicographical order (ticket #, process id #) – Firstly the ticket number is compared. If same then the process ID is compared next.

# Classical IPC Problems and Solutions

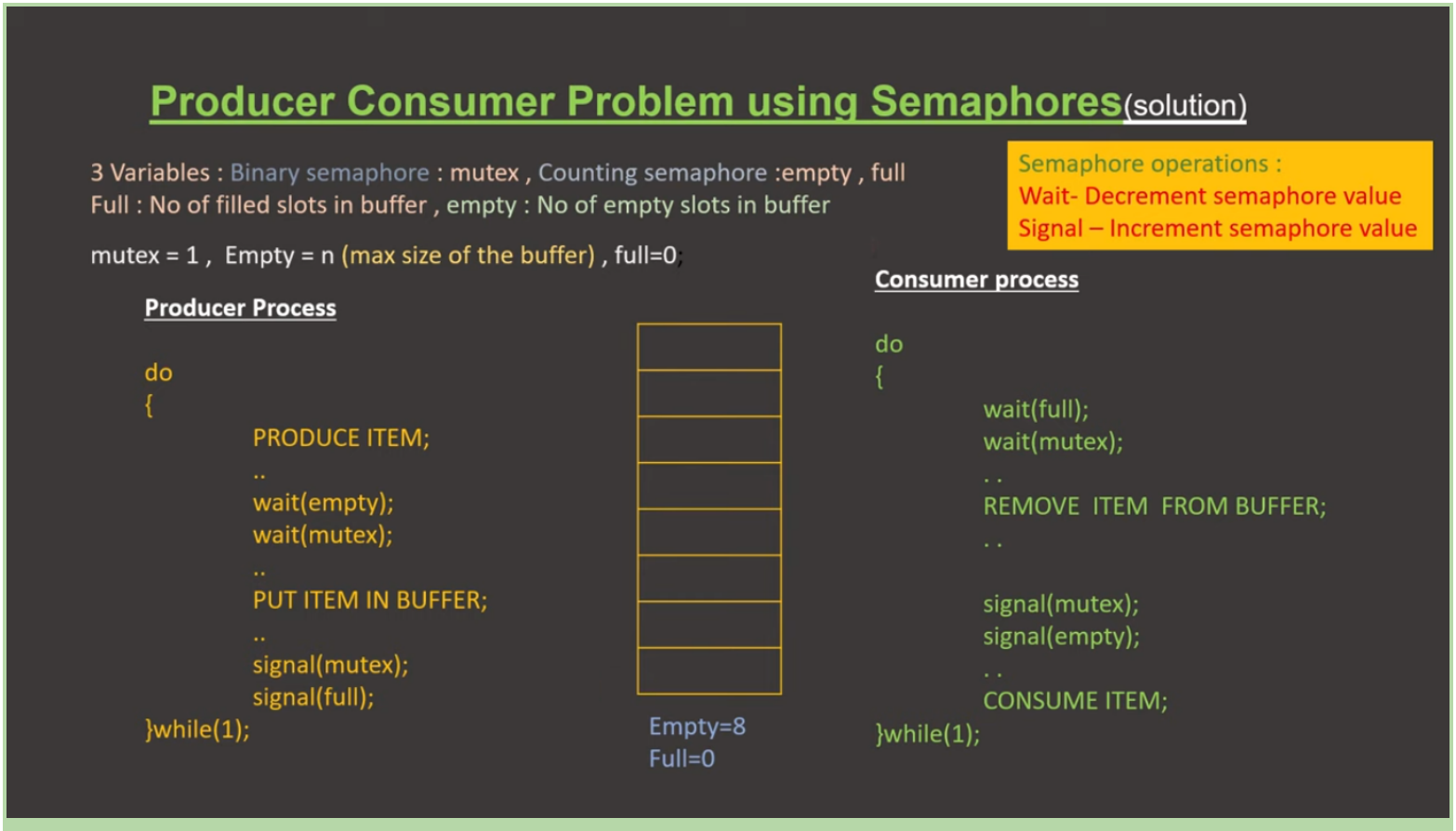
## Producer-Consumer Problem or Bounded Buffer Problem

- The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.
- The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

**Below are a few points that considered as the problems occur in Producer-Consumer:**

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

- **The solution for the Producer-Consumer problem is using semaphore.**



- **Solution:** 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 manner, 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.

**Solution for Producer –**

```
do{
    wait(empty_slots)
    wait(mutex)
    add_item_to_buffer()
    signal(mutex)
    signal(filled_slots)
}while(true)
```

**Producers follow these steps:**

- Wait on ***empty_slots*** semaphore: If there are no empty slots in the buffer (i.e., it's full), the producer will wait.
- Wait on ***mutex*** semaphore: This ensures mutual exclusion so that only one producer can access the buffer at a time.
- Add the item to the buffer.
- Release the ***mutex*** semaphore to allow other producers and consumers to access the buffer.
- Signal ***filled_slots*** semaphore to indicate that an item has been added to the buffer.

**Solution for Consumer –**

```
do{
wait(filled_slots)
wait(mutex)
retrieve_item_from_buffer()
```

```
signal(mutex)
signal(empty_slots)
consume_item()
}while(true)
```

- The ***mutex*** semaphore ensures that only one producer or consumer can access the buffer at any given time, preventing race conditions.
- The ***empty_slots*** semaphore initially represents the available buffer space, preventing producers from overfilling the buffer.
- The ***filled_slots*** semaphore represents the number of items in the buffer, ensuring that consumers wait when the buffer is empty and allowing them to proceed when there's something to consume.
- The ***wait*** operation decrements a semaphore value, and the signal operation increments it.

## Dining Philosophers Problem

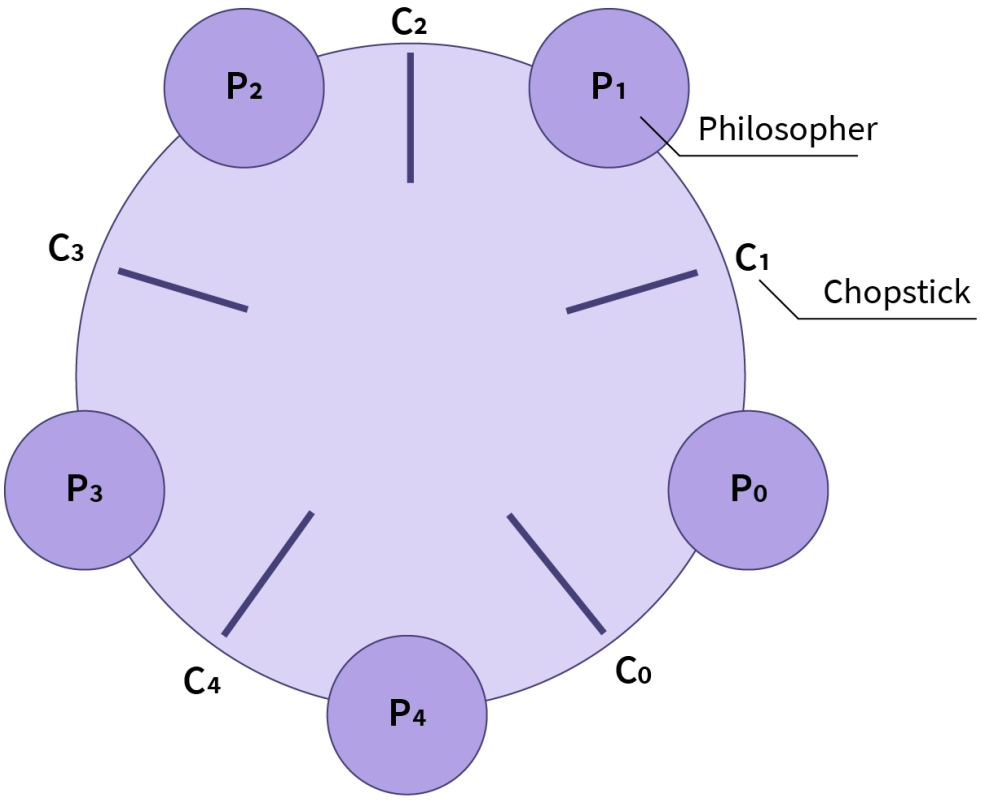
- Dining Philosophers Problem in OS is a classical synchronization problem in the operating system.
- With the presence of more than one process and limited resources in the system the synchronization problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency.

### Problem statement

- Five silent philosophers sit at a round table with bowls of spaghetti.
- Chopsticks/Forks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat.
- However, a philosopher can only eat spaghetti when they have both left and right forks.

- Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher.
- After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others.
- A philosopher can only take the fork on their right or the one on their left as they become available and they cannot start eating before getting both forks.
- Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

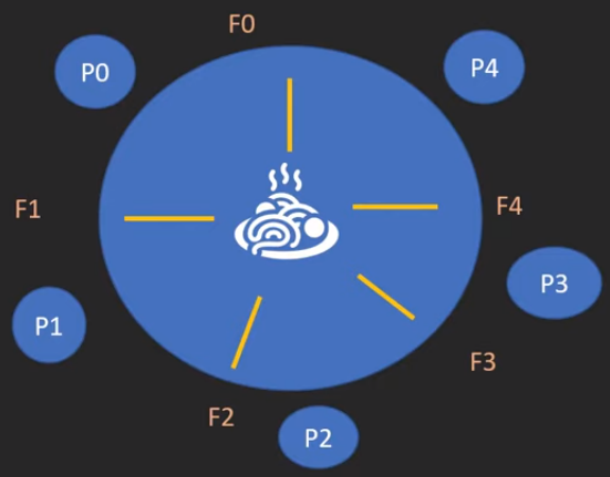
**The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.**



- Dining Philosopher Problem states that 5 philosophers seated around a circular table with one fork between each pair of philosophers
- There is one fork between each philosopher. A philosopher may eat if he can pickup the two forks adjacent to him.
- Every philosopher has 2 states, thinking and Eating
- The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

```
Void philosopher (void)
{
        while(true)
        {
                Thinking()
                take_fork(i); // Left fork
                take_fork((i+1)% N); //  Right Fork
                EAT();
                put_fork(i);
                put_fork((i+1)% N);


        }
}
```

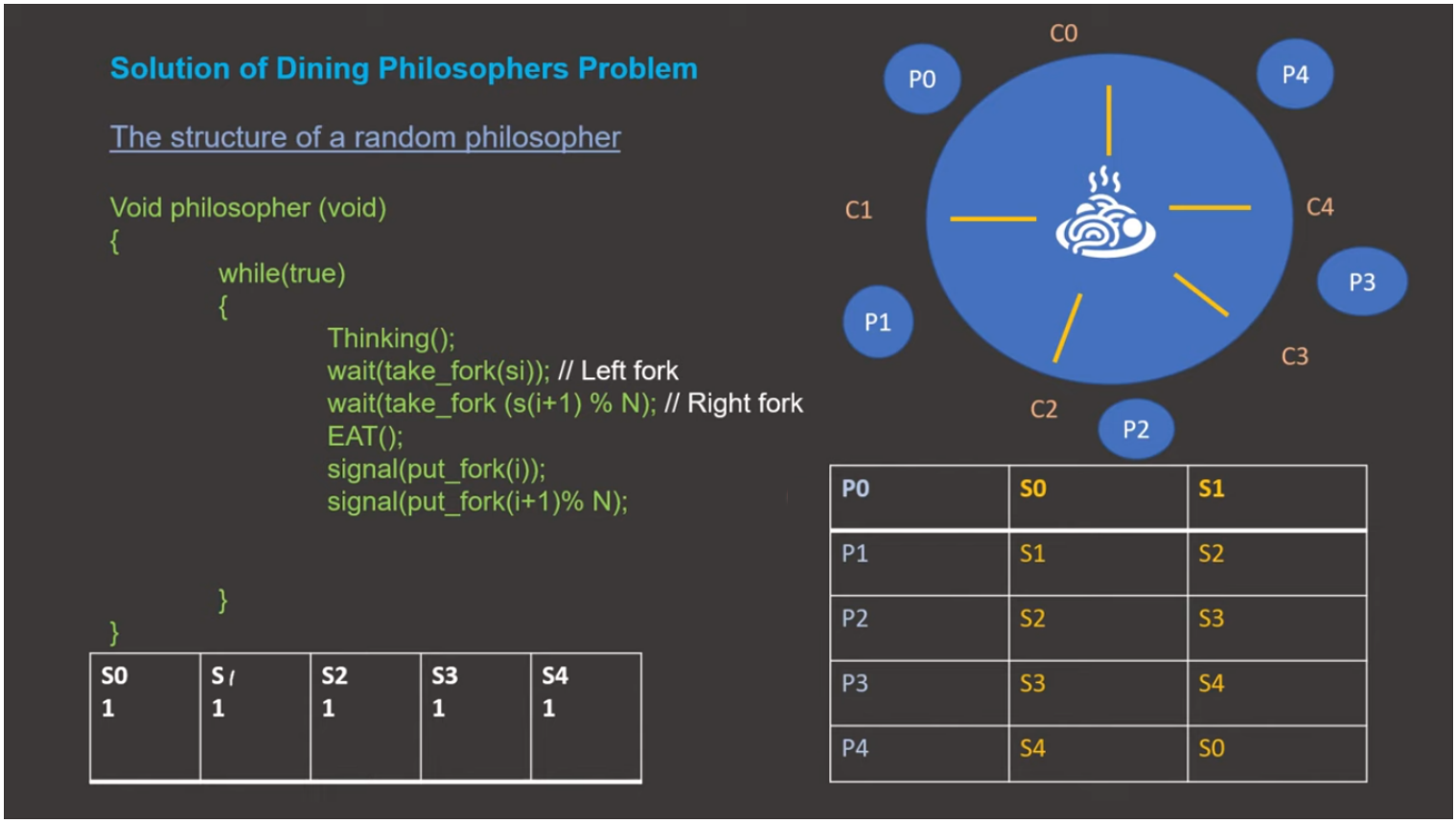



- Let, Initially the philosopher P0 is Thinking.
- After that, P0 can take the forks F0 and F1.
- First  take  the left fork F0.
- Then take the right fork F1.
- Then P0 will eat.
- After eating the P0 will first put down the left fork and then put down the right fork.
- Then the next philosopher P1 will come and continue the same.
- **Now there won't be any issues with the flow of the processes.**

## Problem Arises

- In the above case, F0 is allocated to P0.
- But when it finds for F1, it was already allocated to the philosopher P1.
- So P0 has to wait for the fork from P1.

# Solution




- Here the code is modified with Semaphore.
- For the 5 forks, an array of 5 semaphores S0,S1,S2,S3,S4 are used.
- Each semaphore assigned a value 1 initially.
- Each Philosopher needs 2 semaphores.So, P0 associated with S0 and S1.(Refer above table).
- When P0 comes, it will think first.

- Then set the value of S0 and S1 as 0.
- Now in between P1 comes, the semaphore associated with P1 is S1. But S1 has value=0.
- So P1 can't access the critical section. And it will wait.

**There will be a deadlock situation that occurs when all the Philosophers took the left fork at the same time.**

- So all semaphores value will be 0. There won't be any option available to take the right fork.
- So, inorder to solve the problem, philosophers from P0 to P3 will take the Left fork first and P4 will take the right one first.
- So the previous processes will get their forks.
- So deadlock can be avoided.

## Readers-Writers Problem

- The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are **readers** i.e. they only **want to read the data from the object** and some of the processes are **writers** i.e. they **want to write into the object.**
- The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.
- **To solve this situation,** a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.
- **This can be implemented using semaphores.**

**Problem**

➢ There is a file that is shared between multiple processes.

- ➢ Some of these processes are readers(read data from file) and others writers(write data to the file).
- ➢ Multiple readers can access the file at the same time.
- ➢ If two writers or a reader and writer access the object at the same time, there may be problems.

**Solution**

- The problem can be solved using binary semaphore

## Reader and writers problem using Semaphore

**3 variables**

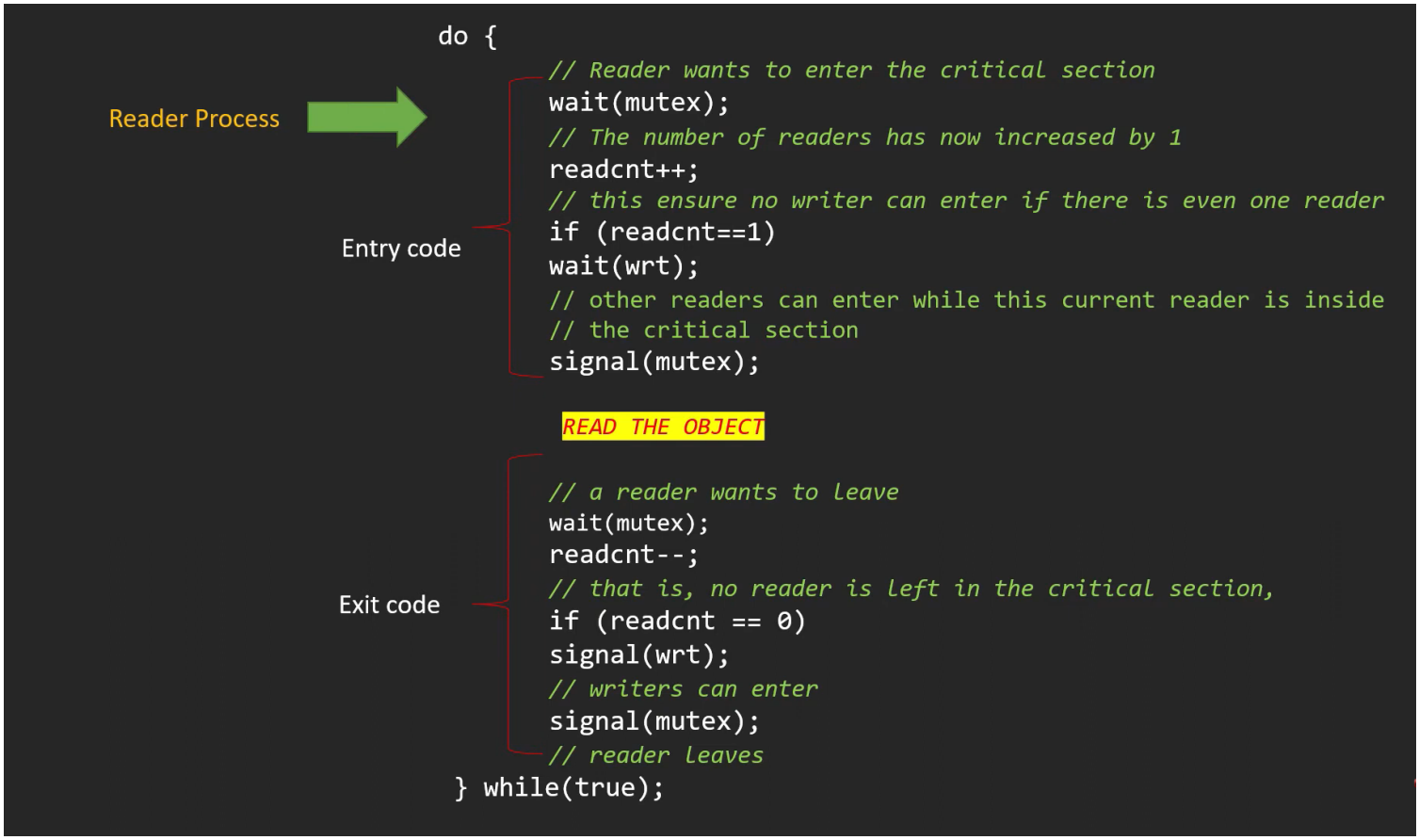
Mutex=1 : Ensure mutual exclusion
wrt =1: Writing mechanism common to reader and writer
readcnt =0: No of readers accessing the file

***Writer Process***

```
do
{
        // writer requests for critical     section
        wait(wrt);
        . . .

        WRITE INTO FILE
        . . .
        // leaves the critical section
        signal(wrt);
} while(true);
```

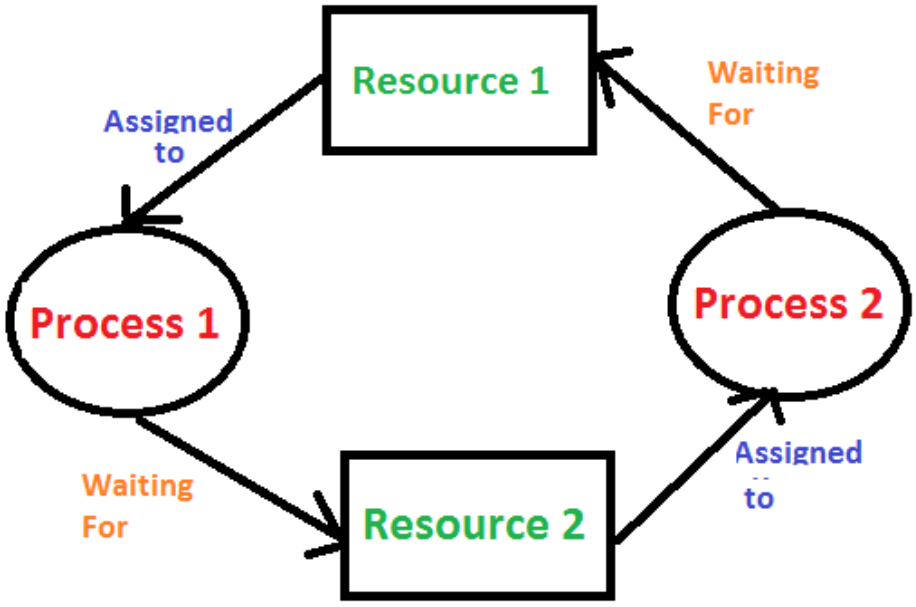
- If a writer wants to access the object, wait operation is performed on wrt.
- After that no other writer can access the object.
- When a writer is done writing into the object, signal operation is performed on wrt.




- In the above code, mutex and wrt are semaphores that are initialized to 1.
- Also, readcnt is a variable that is initialized to 0.
- The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

# Deadlock

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.




In the diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

## Difference between Starvation and Deadlock

| Sr. | Deadlock | Starvation |
|---|---|---|
| 1 | Deadlock is a situation where no process got blocked and no process proceeds | Starvation is a situation where the low priority process got blocked and the high priority processes proceed. |

| | | |
|---|---|---|
| 2 | Deadlock is an infinite waiting. | Starvation is a long waiting but not infinite. |
| 3 | Every Deadlock is always a starvation. | Every starvation need not be deadlock. |
| 4 | The requested resource is blocked by the other process. | The requested resource is continuously be used by the higher priority processes. |
| 5 | Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously. | It occurs due to the uncontrolled priority and resource management. |

## Necessary conditions for Deadlocks

### 1. Mutual Exclusion

- A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

### 2. Hold and Wait

- A process waits for some resources while holding another resource at the same time.

### 3. No preemption

- The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

### 4. Circular Wait

- All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

## Strategies for handling Deadlock

- Deadlock Ignorance
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

## Deadlock Ignorance

- Deadlock Ignorance is the most widely used approach among all the mechanisms.
- This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock.
- This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.
- There is always a tradeoff between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses a deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.
- In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

### Safe State

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

## Deadlock prevention

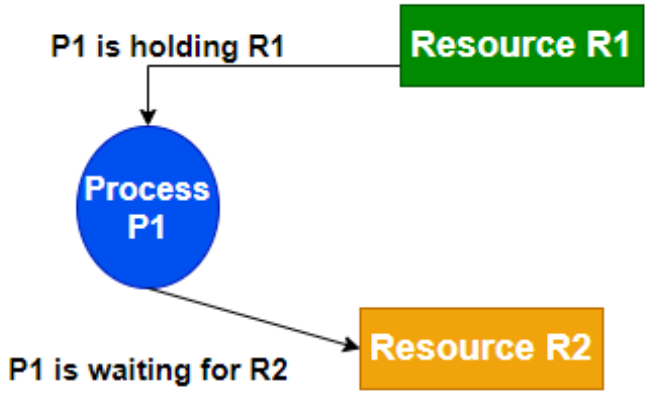
- We can **prevent a Deadlock** by **eliminating any of the four conditions.**

### 1. Eliminate Mutual Exclusion

- This condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes.
- In contrast, Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- A good example of a sharable resource is Read-only files because if several processes attempt to open a read-only file at the same time, then they can be granted simultaneous access to the file.
- A process need not to wait for the sharable resource. Generally, deadlocks cannot be prevented by denying the mutual exclusion condition because there are some resources that are intrinsically non-sharable.

### 2. Solve hold and Wait

- Hold and wait condition occurs when a process holds a resource and is also waiting for some other resource in order to complete its execution.
- Thus if we did not want the occurrence of this condition then we must guarantee that when a process requests a resource, it does not hold any other resource.




**Hold and wait condition**

**There are some protocols that can be used in order to ensure that the Hold and Wait condition never occurs:**

- According to the first protocol; Each process must request and gets all its resources before the beginning of its execution.
- The second protocol allows a process to request resources only when it does not occupy any resource.

**Disadvantages of Both Protocols**

- Utilization of resources may be low, since resources may be allocated but unused for a long period. In the above-given example, for instance, we can release the DVD drive and disk file and again request the disk file and printer only if we can be sure that our data will remain on the disk file. Otherwise, we must request all the resources at the beginning of both protocols.
- There is a 0 A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

### 3. Allow preemption

The third necessary condition for deadlocks is that there should be no preemption of resources that have already been allocated. In order to ensure that this condition does not hold the following protocols can be used :

- According to the First Protocol: "If a process that is already holding some resources requests another resource and if the requested resources cannot be allocated to it, then it must release all the resources currently allocated to it."
- According to the Second Protocol: "When a process requests some resources, if they are available, then allocate them. If in case the requested resource is not available then we will check whether it is being used or is allocated to some other process waiting for other resources. If that resource is not being used, then the operating system preempts it from the waiting process and allocate it to the requesting process. And if that resource is being used, then the requesting process must wait".

The second protocol can be applied to those resources whose state can be easily saved and restored later for example CPU registers and memory space, and cannot be applied to resources like printers and tape drivers.

### 4. Circular wait

The Fourth necessary condition to cause deadlock is circular wait, In order to ensure violate this condition we can do the following:

- Assign a priority number to each resource. There will be a condition that any process cannot request for a lesser priority resource. This method ensures that not a single process can request a resource that is being utilized by any other process and due to which no cycle will be formed.
- Example: Assume that R5 resource is allocated to P1, if next time P1 asks for R4, R3 that are lesser than R5; then such request will not be granted. Only the request for resources that are more than R5 will be granted.

## Deadlock Avoidance

- A deadlock avoidance policy grants a resource request only if it can establish that granting the request cannot lead to a deadlock either immediately or in the future.
- The kernal lacks detailed knowledge about future behavior of processes, so it cannot accurately predict deadlocks.
- To facilitate deadlock avoidance under these conditions, it uses the following conservative approach: Each process declares the maximum number of resource units of each class that it may require.
- The kernal permits a process to request these resource units in stages- i.e. a few resource units at a time- subject to the maximum number declared by it and uses a worst case analysis technique to check for the possibility of future deadlocks.
- A request is granted only if there is no possibility of deadlocks; otherwise, it remains pending until it can be granted.
- **This approach is conservative because a process may complete its operation without requiring the maximum number of units declared by it.**

### Resource Allocation Graph

The resource allocation graph (RAG) is used to visualize the system's current state as a graph. The Graph includes all processes, the resources that are assigned to them, as well as the resources that each Process requests. Sometimes, if there are fewer processes, we can quickly spot a deadlock in the system by looking at the graph rather than the tables we use in Banker's algorithm. **Deadlock avoidance can also be done with Banker's Algorithm.**

## Banker's Algorithm

Bankers's Algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, and after granting a request system remains in the safe state it allows the

request, and if there is no safe state it doesn't allow the request made by the process.

**Inputs to Banker's Algorithm**

1. Max needs of resources by each process.
2. Currently, allocated resources by each process.
3. Max free available resources in the system.

**The request will only be granted under the below condition**

1. If the request made by the process is less than equal to the max needed for that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

**Timeouts:** To avoid deadlocks caused by indefinite waiting, a timeout mechanism can be used to limit the amount of time a process can wait for a resource. If the help is unavailable within the timeout period, the process can be forced to release its current resources and try again later.

**Example:**

**Total resources in system:**

A B C D

6 5 7 6

**Available system resources are:**

A B C D

3 1 1 2

**Processes (currently allocated resources):**

| | A | B | C | D |
|---|---|---|---|---|
| P1 | 1 | 2 | 2 | 1 |
| P2 | 1 | 0 | 3 | 3 |
| P3 | 1 | 2 | 1 | 0 |

**Maximum resources we have for a process**

Processes (maximum resources):

| | A | B | C | D |
|---|---|---|---|---|
| P1 | 3 | 3 | 2 | 2 |
| P2 | 1 | 2 | 3 | 4 |
| P3 | 1 | 3 | 5 | 0 |

**Need = Maximum Resources Requirement – Currently Allocated Resources.**

**Processes (need resources):**

| | A | B | C | D |
|---|---|---|---|---|
| P1 | 2 | 1 | 0 | 1 |
| P2 | 0 | 2 | 0 | 1 |
| P3 | 0 | 1 | 4 | 0 |

# Deadlock Detection And Recovery

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system. A deadlock occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system, where no process can make progress.

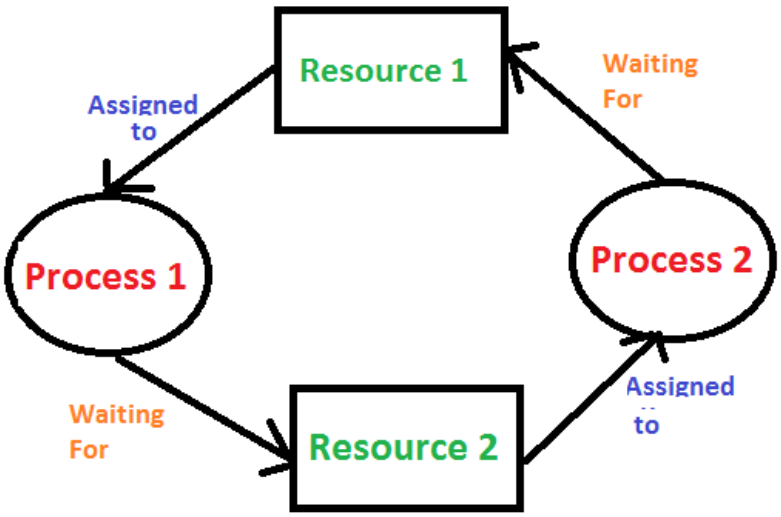
**There are two main approaches to deadlock detection and recovery:**

1. **Prevention**: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them.
   - **Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks.**
   - **Recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.**

**Deadlock Detection :**

1. **If resources have a single instance –**

- In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph.
- **The presence of a cycle in the graph is a sufficient condition for deadlock.**




In the above diagram, resource 1 and resource 2 have single instances.
**There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.**

**2. If there are multiple instances of resources –**
Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

### Wait-For Graph Algorithm

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

## Deadlock Recovery

A traditional operating system such as Windows doesn't deal with deadlock recovery as it is a time and space-consuming process. Real-time operating systems use Deadlock recovery.

**1. Manual Intervention:**

When a deadlock is detected, one option is to inform the operator and let them handle the situation manually. While this approach allows for human judgment and decision-making, it can be time-consuming and may not be feasible in large-scale systems.

**2. Automatic Recovery:**

- An alternative approach is to enable the system to recover from deadlock automatically.
- This method involves breaking the deadlock cycle by either aborting processes or preempting resources.
- Let's delve into these strategies in more detail.

**Recovery from Deadlock: Process Termination:**

**Abort all deadlocked processes:**

This approach breaks the deadlock cycle, but it comes at a significant cost. The processes that were aborted may have executed for a considerable amount of time, resulting in the loss of partial computations. These computations may need to be recomputed later.

**Abort one process at a time**

Instead of aborting all deadlocked processes simultaneously, this strategy involves selectively aborting one process at a time until the deadlock cycle is eliminated. However, this incurs overhead as a deadlock-detection algorithm must be invoked after each process termination to determine if any processes are still deadlocked.

## Recovery from Deadlock: Resource Preemption

### Selecting a victim

Resource preemption involves choosing which resources and processes should be preempted to break the deadlock. The selection order aims to minimize the overall cost of recovery. Factors considered for victim selection may include the number of resources held by a deadlocked process and the amount of time the process has consumed.

### Rollback

If a resource is preempted from a process, the process cannot continue its normal execution as it lacks the required resource. Rolling back the process to a safe state and restarting it is a common approach. Determining a safe state can be challenging, leading to the use of total rollback, where the process is aborted and restarted from scratch.

### Starvation prevention

To prevent resource starvation, it is essential to ensure that the same process is not always chosen as a victim. If victim selection is solely based on cost factors, one process might repeatedly lose its resources and never complete its designated task. To address this, it is advisable to limit the number of times a process can be chosen as a victim, including the number of rollbacks in the cost factor.

## Advantages of Deadlock Detection and Recovery in Operating Systems

1. **Improved System Stability:** Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.

2. **Better Resource Utilization:** By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. **Better System Design**: Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.

## Disadvantages of Deadlock Detection and Recovery in Operating Systems

1. **Performance Overhead:** Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action to resolve them.
2. **Complexity:** Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.
3. **False Positives and Negatives:** Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
4. **Risk of Data Loss**: In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Overall, the choice of deadlock detection and recovery approach depends on the specific requirements of the system, the trade-offs between performance, complexity, and accuracy, and the risk tolerance of the system. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.