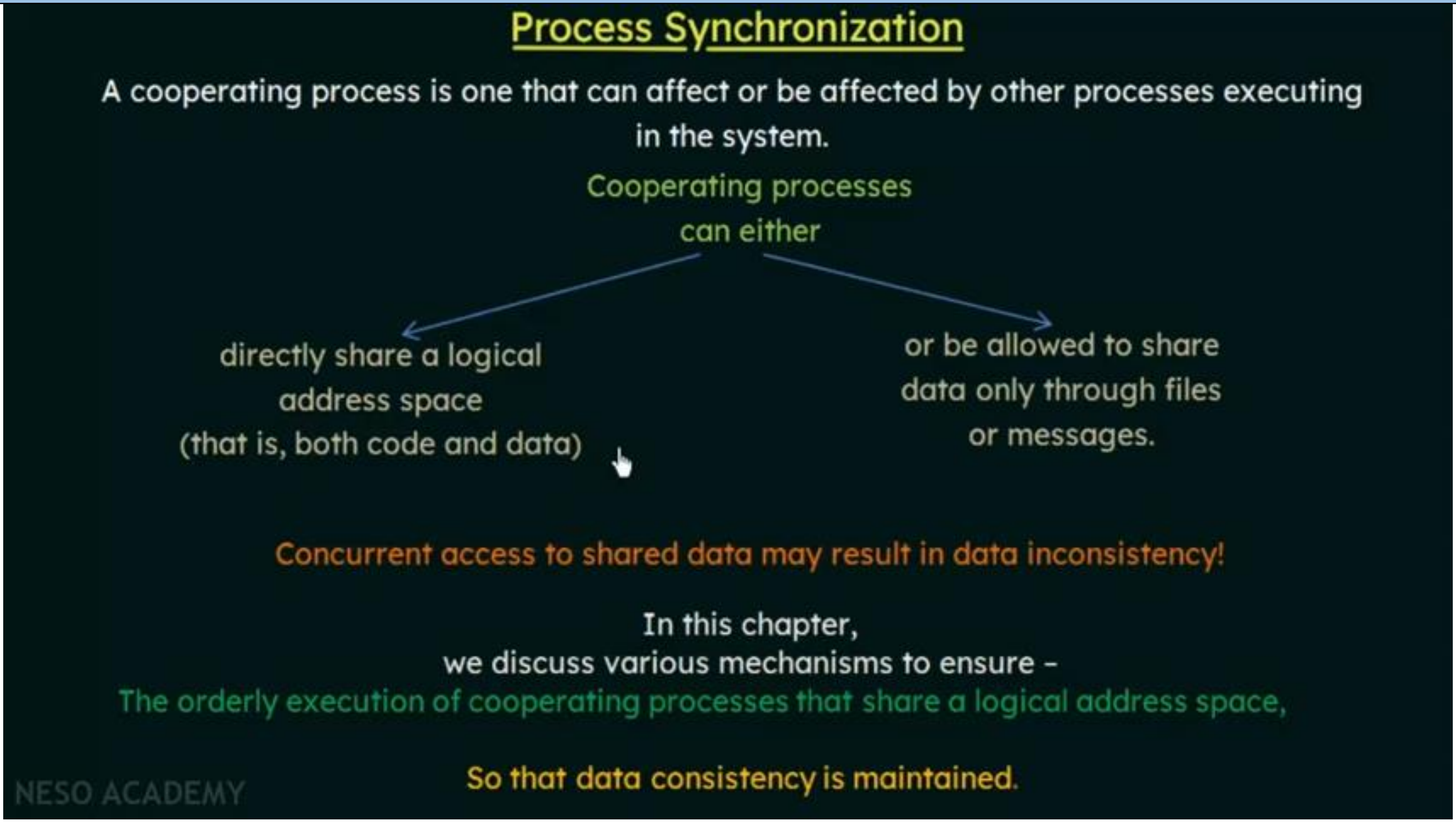
1. PROCESS SYNCHRONIZATION



Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

directly share a logical address space (that is, both code and data)

or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter, we discuss various mechanisms to ensure –
The orderly execution of cooperating processes that share a logical address space,

So that data consistency is maintained.

- 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.
- 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.

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

A. **Independent Process**:
   The execution of one process does not affect the execution of other processes.
B. **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 also because resources are shared in Cooperative processes.

Race Condition:

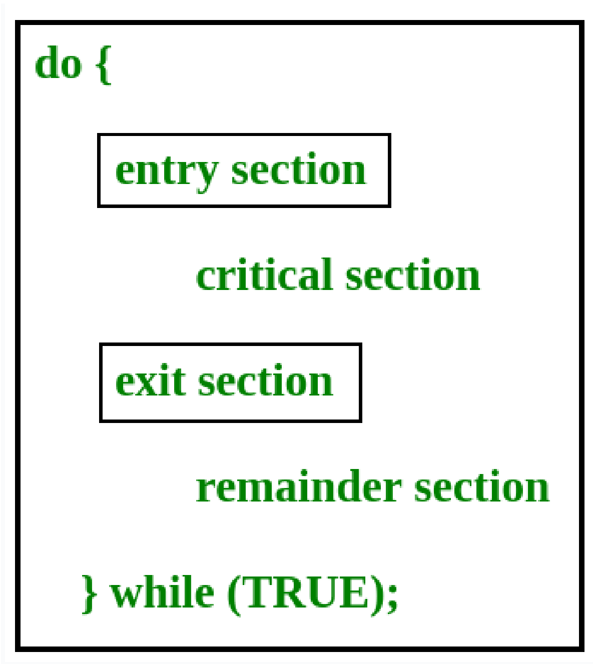
❖ **It is an unwanted condition when two or more process perform one or more operations at the same time i.e. several processes access and manipulate shared data.**

❖ **Process must be racing together to access a sharable resources/ critical section.**

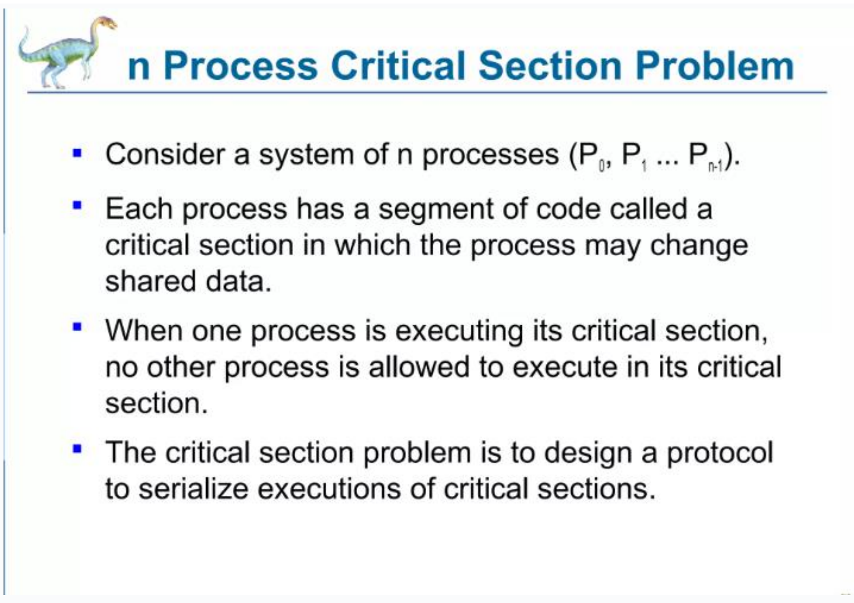
❖ **The order of execution of instruction influences the result produced.**

2. Critical Section

1. Critical Section refers to the segment of code or the program which tries to access or modify the value of the variables in a shared resource.
2. The section above the critical section is called the **Entry Section**. The process that is entering the critical section must pass the entry section.



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



n Process Critical Section Problem

- Consider a system of n processes (P_0, P_1 ... P_{n-1}).
- Each process has a segment of code called a critical section in which the process may change shared data.
- When one process is executing its critical section, no other process is allowed to execute in its critical section.
- The critical section problem is to design a protocol to serialize executions of critical sections.

3. The section below the critical section is called the **Exit Section**.
4. The section below the exit section is called the **Reminder Section** and this section has the remaining code that is left after execution.
5. The critical section is used to solve the problem of race conditions occurring due to synchronization.
6. The critical section represents the segment of code that can access or modify a shared resource.
7. There can be only one process in the critical section at a time.
8. There are many solutions to critical section problems such as Peterson's solution, semaphore, etc.

3. Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

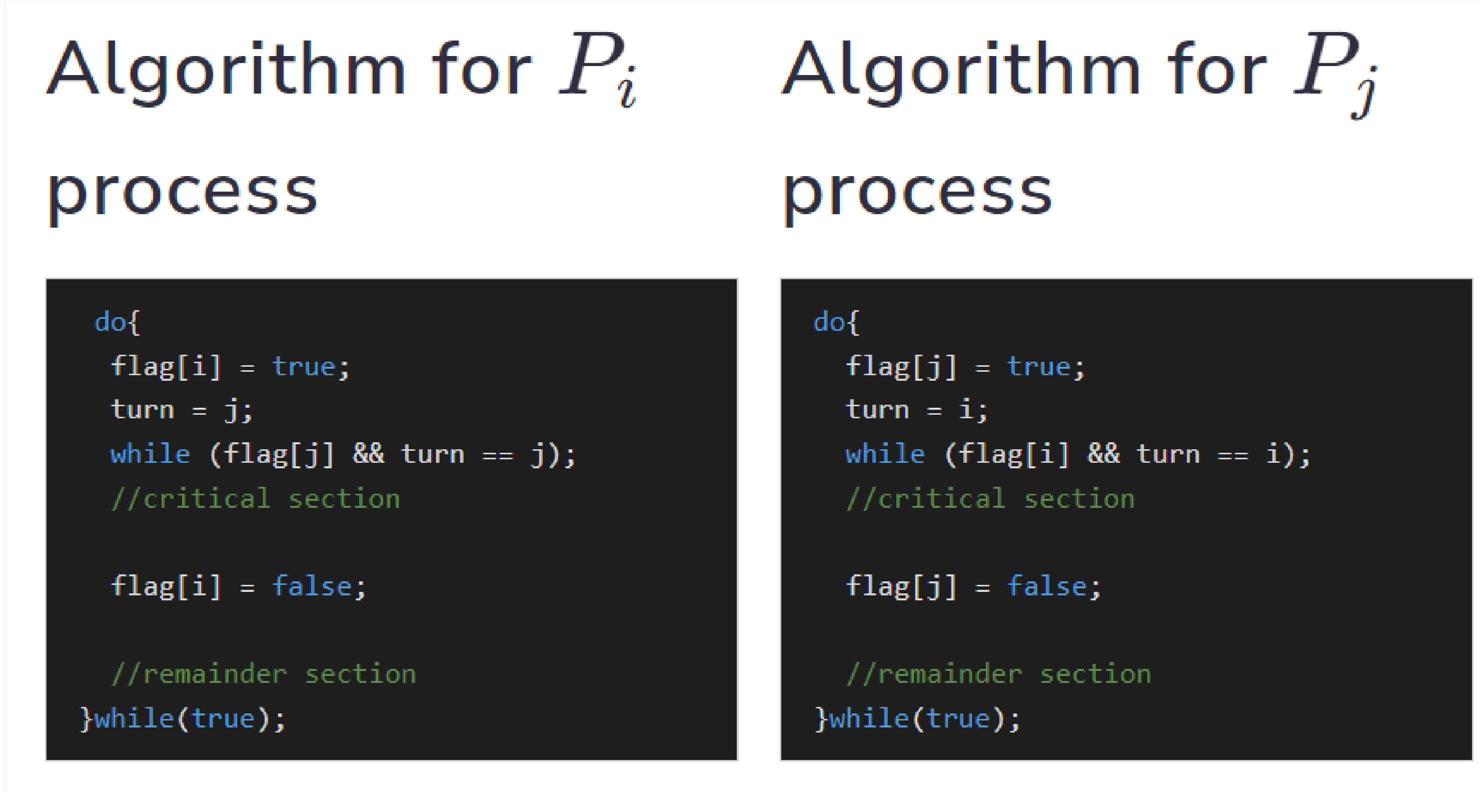
A. **Mutual Exclusion**
Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

B. **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..

C. **Bounded Waiting**
Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

4. Explanation of Peterson's Algorithm in OS

1. Peterson's solution is one of the classical solutions to solve the critical-section problem in OS.
2. It follows a simple algorithm and is limited to two processes simultaneously.



Algorithm for P_i process

```
do{
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  //critical section

  flag[i] = false;

  //remainder section
}while(true);
```

Algorithm for P_j process

```
do{
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  //critical section

  flag[j] = false;

  //remainder section
}while(true);
```

a) There are total N processes, each with a variable flag set to false on initialization. This variable flag indicates if a process is ready to enter the critical region.
b) The turn variable denotes which process its turn is now to enter the critical region.
c) Then, every process will enter the entry section in which we define j, which denotes another process that came before process i.
d) We allow process I to enter the critical section and indicate that j process is now turned to enter the critical section using turn=j.
e) Then we check whether the process j is in the critical region using the conditions flag[j]==true && turn=j. If process j is in the critical region, the while loop runs continuously, and stalls process *i* from entering the region until process j exits out of the critical region.
f) The process which has exited the critical region is marked by flag[i]=false;, where I denote the process exiting from the critical region.

Advantages of Peterson's Solution

1) Peterson's solution allows multiple processes to share and access a resource without conflict between the resources.
2) Every process gets a chance of execution.
3) It is simple to implement and uses simple and powerful logic.
4) It can be used in any hardware as it is completely software dependent and executed in the user mode.
5) Prevents the possibility of a deadlock.

Disadvantages of Peterson's Solution

1) The process may take a long time to wait for the other processes to come out of the critical region. It is termed as **Busy waiting**.
2) This algorithm may not work on systems having multiple CPUs.
3) The Peterson solution is restricted to only two processes at a time.

5. Semaphores

They are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization..

The definitions of wait and signal are as follows –

1) Wait

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

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

2) Signal

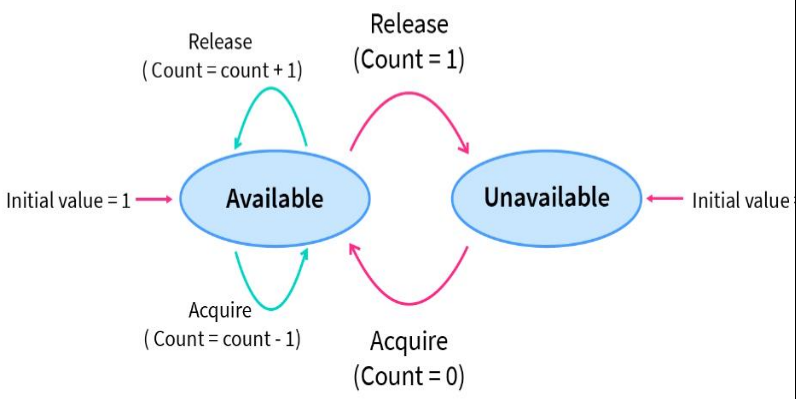
The signal operation increments the value of its argument S.

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

TYPES OF SEMAPHORES

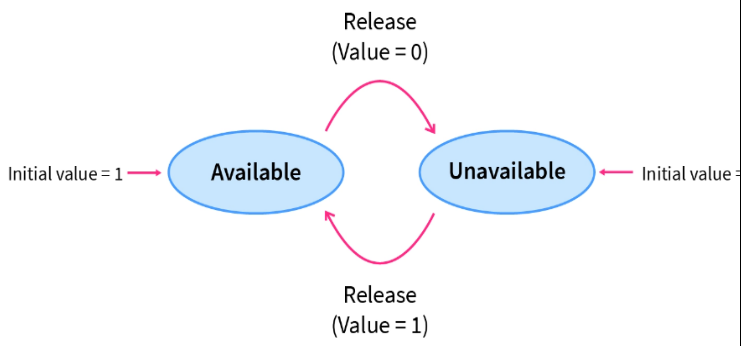
1) Counting Semaphores

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.



2) Binary Semaphores

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.



- There are two types of semaphores:
  - Binary - take on values 0 or 1
  - Counting - take on any integer value
- There are mainly two operations of semaphores:
  - Wait - decrements the value of semaphore
  - Signal - increments the value of semaphore

S.No	Mutex	Semaphores
1.	It is a locking mechanism used to synchronize access to a resource. A thread needs to lock the resource and unlock it as well.	It is a signaling mechanism. It uses wait() and signal() calls.
2.	It is an object.	It is an integer variable.
3.	It allows multiple threads to access the same resource but not concurrently.	It allows multiple processes to access the finite instance of resources.
4.	It can be released only by the thread that has locked it.	It can be released by any process acquiring or releasing the resource.
5.	It does not have different categories.	Semaphores are of two types: Binary and counting semaphores.

6. Classical Problems of Synchronization

Process synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources

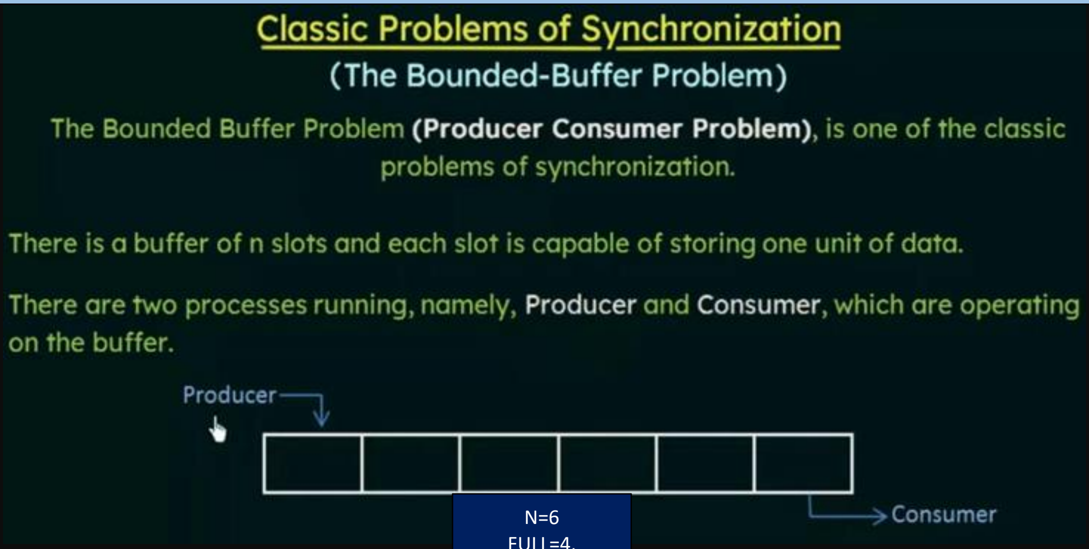
Three classical problem depicting flaws of process synchronaization in systems where cooperating processes are present are

1) Producer/Consumer Problem or Bounded-Buffer Problem

2) Readers and Writers Problem

3) Dining-Philosophers Problem

1. PRODUCER/CONSUMER OR BOUNDED-BUFFER PROBLEM



Classic Problems of Synchronization

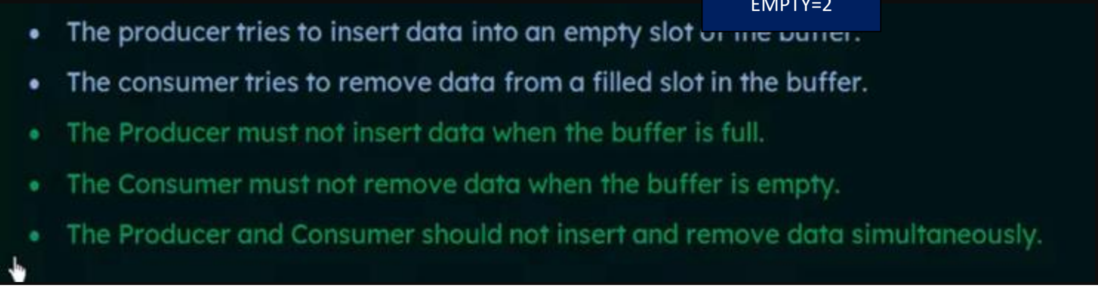
(The Bounded-Buffer Problem)

The Bounded Buffer Problem **(Producer Consumer Problem)**, is one of the classic problems of synchronization.

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.

N=6
FULL=4,
EMPTY=2



- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Solution

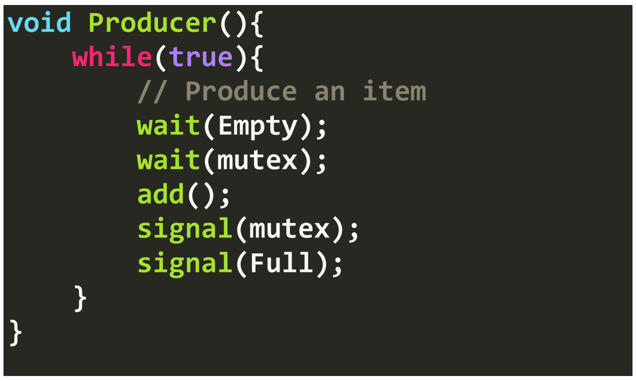
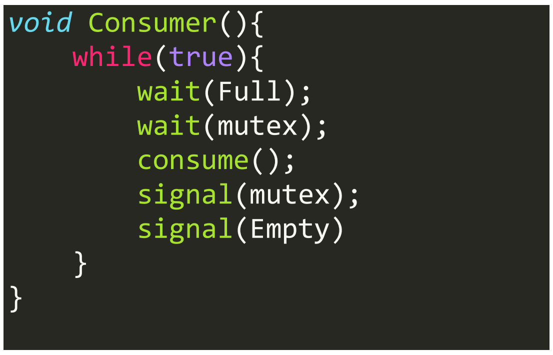
The solution to the Producer-Consumer problem involves three *semaphore* variables.

A. **semaphore Full**:
Tracks the space filled by the Producer process. It is initialized with a value of 00 as the buffer will have 00 filled spaces at the beginning

B. **semaphore Empty**:
Tracks the empty space in the buffer. It is initially set to **buffer_size** as the whole buffer is empty at the beginning.

C. **semaphore mutex**:
Used for mutual exclusion so that only one process can access the shared buffer at a time.

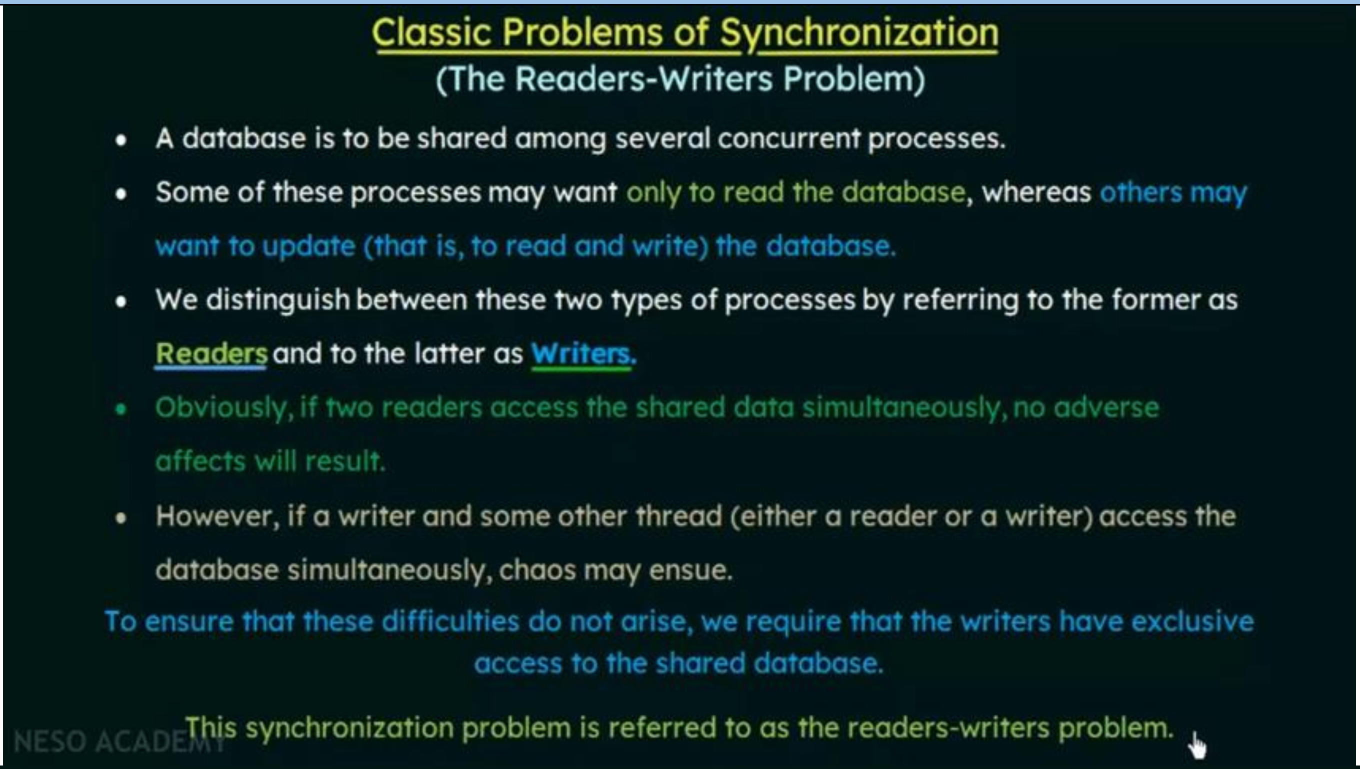
Using the signal() and wait() operations on these semaphores, we can arrive at a solution.

Let's look at the code for the Producer processes.	**The code for the Consumer process is as follows.**
	

```
void Producer(){
    while(true){
        // Produce an item
        wait(Empty);
        wait(mutex);
        add();
        signal(mutex);
        signal(Full);
    }
}
```

```
void Consumer(){
    while(true){
        wait(Full);
        wait(mutex);
        consume();
        signal(mutex);
        signal(Empty)
    }
}
```

2. Readers and Writers Problem

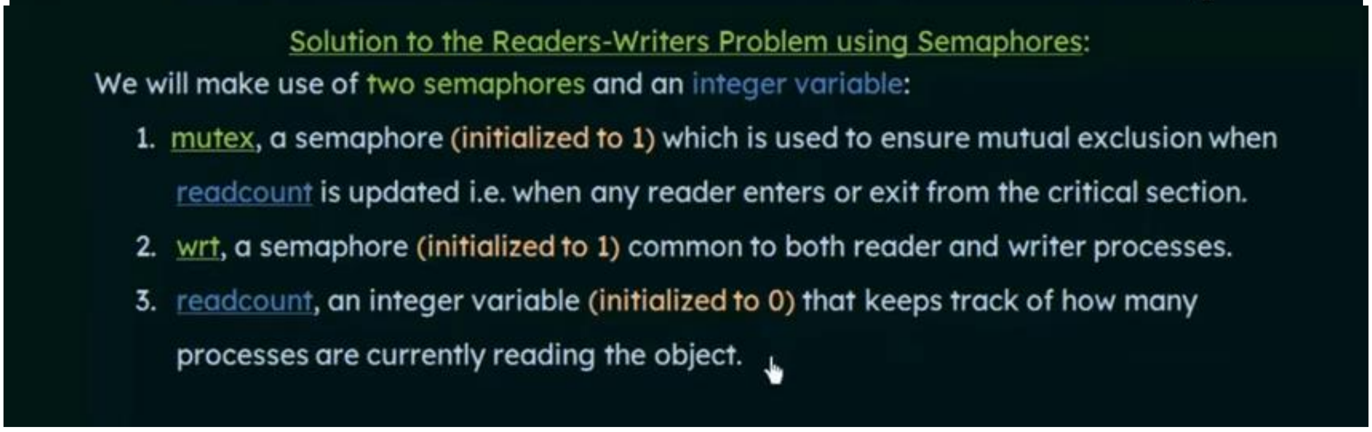


Classic Problems of Synchronization
(The Readers-Writers Problem)

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as Readers and to the latter as Writers.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

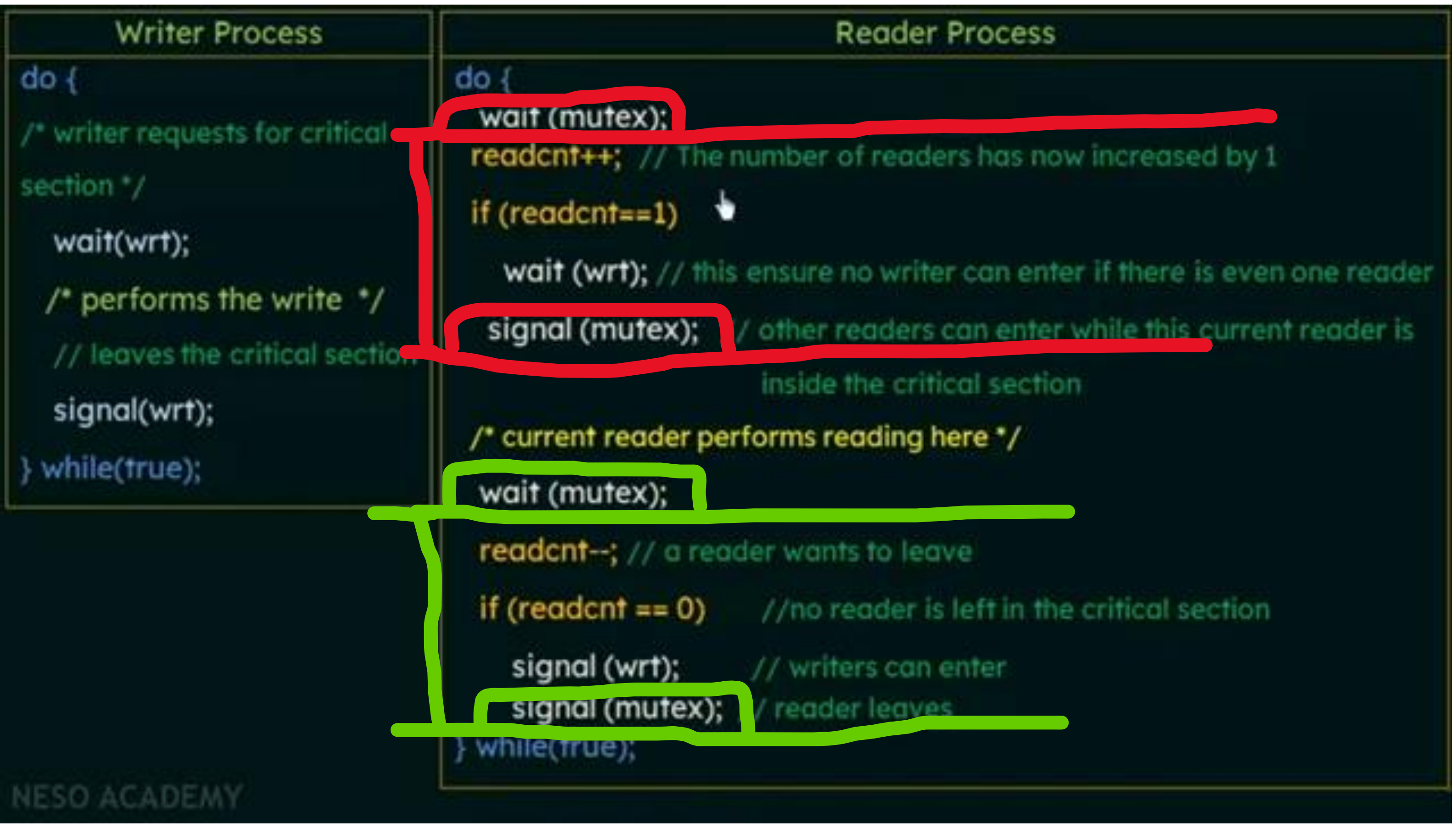
This synchronization problem is referred to as the readers-writers problem.



Solution to the Readers-Writers Problem using Semaphores:

We will make use of two semaphores and an integer variable:

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
2. wrt, a semaphore (initialized to 1) common to both reader and writer processes.
3. readcount, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.



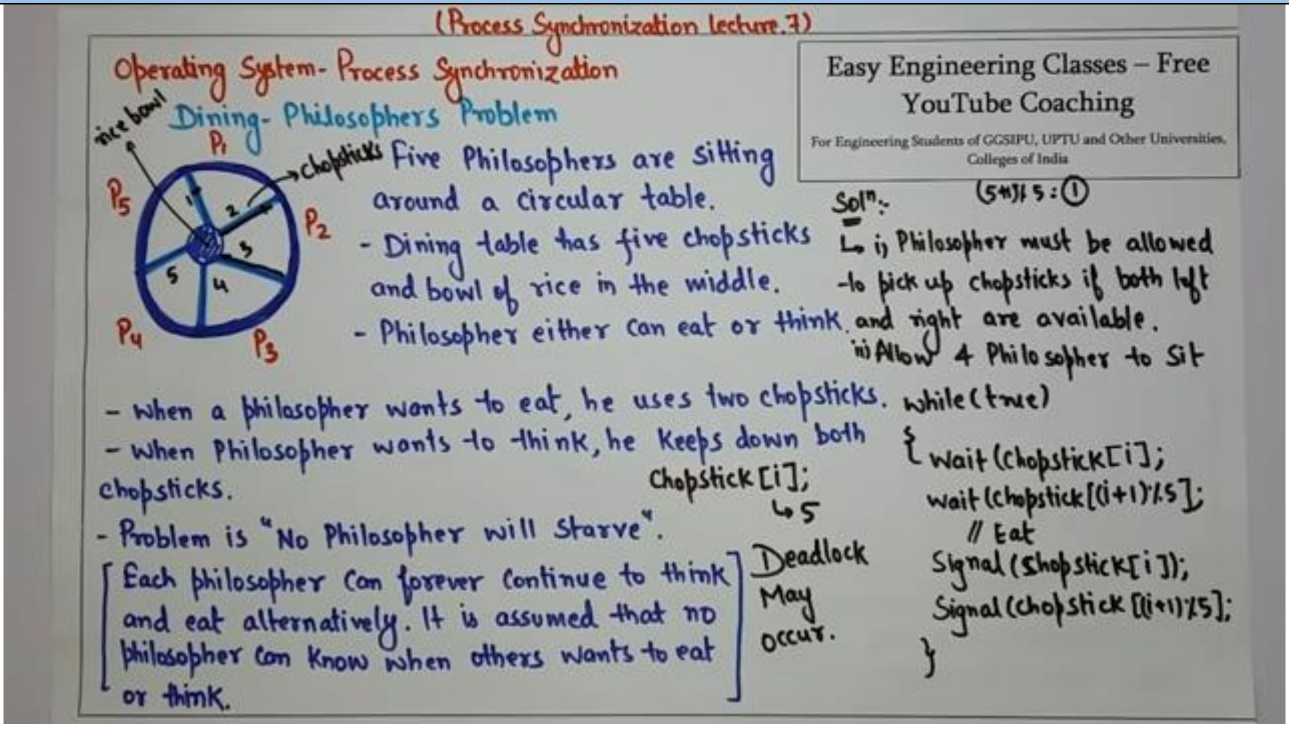
Writer Process

```
do {
/* writer requests for critical
section */
   wait(wrt);
  /* performs the write */
   // leaves the critical sectio
   signal(wrt);
} while(true);
```

Reader Process

```
do {
   wait (mutex);
  readcnt++;  // The number of readers has now increased by 1
  if (readcnt==1)
     wait (wrt); // this ensure no writer can enter if there is even one reader
    signal (mutex);  // other readers can enter while this current reader is
                       inside the critical section
  /* current reader performs reading here */
   wait (mutex);
   readcnt--; // a reader wants to leave
   if (readcnt == 0)      //no reader is left in the critical section
      signal (wrt);       // writers can enter
      signal (mutex);     // reader leaves
} while(true);
```

3. Dining-Philosophers Problem


rice bowl
chopsticks
P1
P2
P3
P4
P5
1
2
3
4
5

(Process Synchronization lecture.7)

Operating System- Process Synchronization

Dining- Philosophers Problem

Easy Engineering Classes – Free YouTube Coaching

For Engineering Students of GGSIPU, UPTU and Other Universities, Colleges of India

Five Philosophers are sitting around a circular table.

- Dining table has five chopsticks and bowl of rice in the middle.
- Philosopher either can eat or think.
- when a philosopher wants to eat, he uses two chopsticks.
- when Philosopher wants to think, he keeps down both chopsticks.
- Problem is "No Philosopher will Starve".

[Each philosopher can forever continue to think and eat alternatively. It is assumed that no philosopher can know when others wants to eat or think.]

Chopstick [i];
↳ 5

Deadlock May occur.

Soln:-
↳ i) Philosopher must be allowed to pick up chopsticks if both left and right are available.
ii) Allow 4 Philosopher to Sit

(5+1)%5 : ①

```
while(true)
{ wait (chopstick[i];
  wait (chopstick[(i+1)%5];
    // Eat
  Signal (shopstick[i]);
  Signal (chopstick [(i+1)%5];
}
```

1. The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively.
2. There is a bowl of rice for each of the philosophers and 5 chopsticks.
3. A philosopher needs both their right and left chopstick to eat.
4. A hungry philosopher may only eat if there are both chopsticks available.
5. Otherwise a philosopher puts down their chopstick and begin thinking again.

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows –

```
do {
   wait( chopstick[i] );
   wait( chopstick[ (i+1) % 5] );
   . .
   . EATING THE RICE
   .
   signal( chopstick[i] );
   signal( chopstick[ (i+1) % 5] );
   .
   . THINKING
   .
} while(1);
```

a. In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5].
b. This means that the philosopher i has picked up the chopsticks on his sides.
c. Then the eating function is performed.

d. After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5].
e. This means that the philosopher i has eaten and put down the chopsticks on his sides.
f. Then the philosopher goes back to thinking.

What is a monitor in OS?

1. Monitors are a programming language component that aids in the regulation of shared data access.
2. The Monitor is a package that contains shared data structures, operations, and synchronization between concurrent procedure calls.
3. Therefore, a monitor is also known as a synchronization tool. **Java, C#, Visual Basic, Ada, and concurrent Euclid** are among some of the languages that allow the use of monitors.

Components of Monitor in an operating system

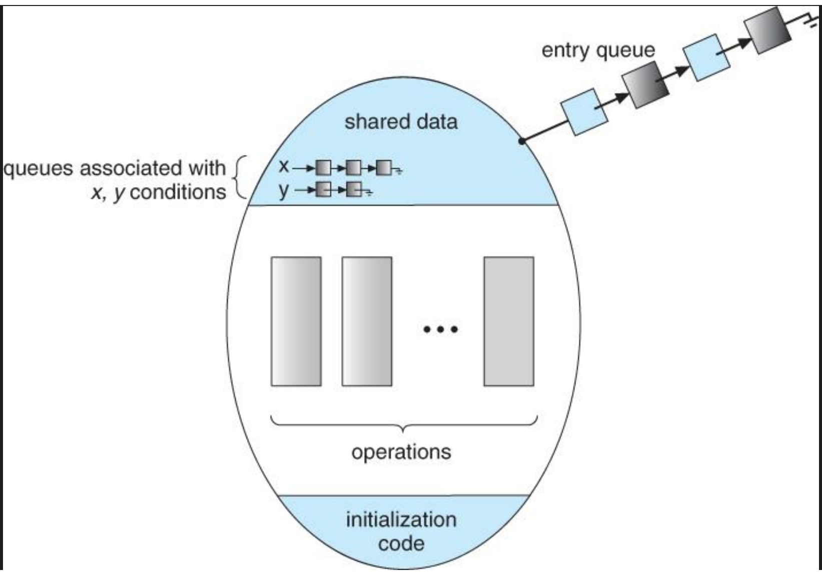
The monitor is made up of four primary parts:

```
Monitor monitorName{
    variables_declaration;
    condition_variables;

    procedure p1{ ... };
    procedure p2{ ... };
    ...
    procedure pn{ ... };

    {
        initializing_code;
    }

}
```



1. **Initialization:**
   The code for initialization is included in the package, and we just need it once when **creating the monitors**.
2. **Private Data:**
   It is a feature of the monitor in an operating system to make the data private. It holds all of the monitor's secret data, which includes private functions that may only be utilized within the monitor. As a result, private fields and functions are not visible outside of the monitor.
3. **Monitor Procedure:**
   Procedures or functions that can be invoked from outside of the monitor are known as **monitor procedures**.
4. **Monitor Entry Queue:**
   Another important component of the monitor is the Monitor Entry Queue. It contains all of the threads, which are commonly referred to as procedures only.

Condition Variables

There are two sorts of operations we can perform on the monitor's condition variables:

1. Wait
2. Signal

   Consider a condition variable (y) is declared in the monitor:

- **y.wait():**
  The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.
- **y.signal():**
  If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

Characteristics of Monitors in OS

A monitor in os has the following characteristics:

- We can only run one program at a time inside the monitor.
- Monitors in an operating system are defined as a group of methods and fields that are combined with a special type of package in the os.
- A program cannot access the monitor's internal variable if it is running outside the monitor. Although, a program can call the monitor's functions.
- Monitors were created to make synchronization problems less complicated.
- Monitors provide a high level of synchronization between processes.

Advantages of Monitor in OS

- Monitors offer the benefit of making concurrent or parallel programming easier and less error-prone than semaphore-based solutions.
- It helps in process synchronization in the operating system.
- Monitors have built-in mutual exclusion.
- Monitors are easier to set up than semaphores.
- Monitors may be able to correct for the timing faults that **semaphores cause**.

Disadvantages of Monitor in OS

- Monitors must be **implemented** with the programming language.
- Monitor increases the compiler's workload.
- The monitor requires to understand what **operating system** features are available for **controlling crucial** sections in the parallel procedures.