UNIT – 3

PROCESS SYNCHRONIZATION –

- **The Critical Section Problem,**
- **Synchronization Hardware,**
- **Semaphores, and Classical Problems of Synchronization,**
- **Critical Regions,**

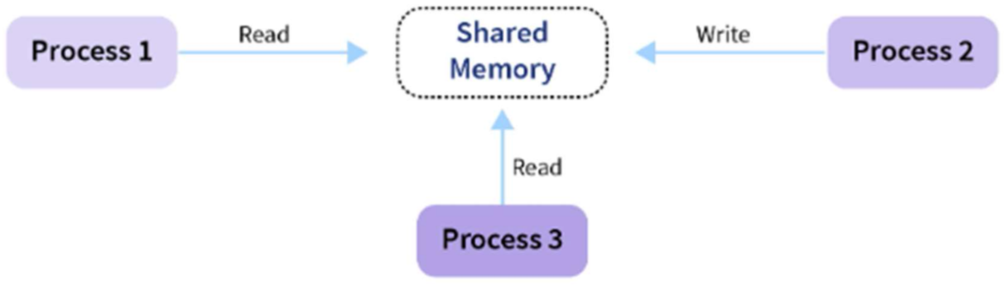
Monitors Inter process Communication Mechanisms:

- **IPC between processes on a single computer system,**
- **IPC between processes on different systems, using pipes,**
- **FIFOs,**
- **message queues,**
- **shared memory.**

3.1 Process Synchronization:

An operating system is software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

Let us take a look at why exactly we need Process Synchronization. For example, If a process1 is trying to read the data present in a memory location while another process2 is trying to change the data present at the same location, there is a high chance that the data read by the process1 will be incorrect.



3.2 Critical Section Problem:

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

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

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

1) Solutions

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.

Progress

Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

Bounded Waiting

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

2) Two general approaches are used to handle critical sections:

Preemptive kernels:

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

Non preemptive kernels:

A non-preemptive kernel does not allow a process running in kernel mode to be preempted. A kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

3.3 Hardware Synchronization:

Process Synchronization problem can be solved by software as well as a hardware solution. Peterson solution is one of the software solutions to the process synchronization problem. Peterson algorithm allows two or more processes to share a single-use resource without any conflict.

There are three algorithms in the hardware approach of solving Process Synchronization problem:

Test and Set
Swap
Unlock and Lock

1. Test and Set:

- Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true.
- The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop.
- The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured.
- Once the first process gets out of the critical section, lock is changed to false.
- So, now the other processes can enter one by one. Progress is also ensured.
- However, after the first process, any process can go in.
- There is no queue maintained, so any new process that finds the lock to be false again can enter.
- So bounded waiting is not ensured.

Test and Set Pseudocode –

```
//Shared variable lock initialized to false
boolean lock;
boolean TestAndSet (boolean &target){
    boolean rv = target;
    target = true;
    return rv;
}
while(1)
{
    while (TestAndSet(lock));
     critical section
    lock = false;
remainder section
}
```

2. Swap:

- Swap algorithm is a lot like the TestAndSet algorithm.
- Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock.
- First process will be executed, and in while(key), since key=true, swap will take place and hence lock=true and key=false.
- Again next iteration takes place while(key) but key=false, so while loop breaks and first process will enter in critical section.
- Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process).
- Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section.
- Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured.

However, again bounded waiting is not ensured for the very same reason.

Swap Pseudocode –

```
// Shared variable lock initialized to false
// and individual key initialized to false;
boolean lock;
Individual key;
void swap(boolean &a, boolean &b){
   boolean temp = a;
   a = b;
   b = temp;
}
while (1)
{
   key = true;
   while(key)
       swap(lock,key);
critical section
   lock = false;
remainder section
}
```

3. Unlock and Lock:

- Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting.
- A ready queue is maintained with respect to the process in the critical section.
- All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially.
- Once the ith process gets out of the critical section, it does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms. Instead, it checks if there is any process waiting in the queue.
- The queue is taken to be a circular queue.
- j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section.

- If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section.
- If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section.
- This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm.

Unlock and Lock Pseudocode –

```
// Shared variable lock initialized to false
// and individual key initialized to false
boolean lock;
Individual key;
Individual waiting[i];
while(1){
   waiting[i] = true;
   key = true;
   while(waiting[i] && key)
      key = TestAndSet(lock);
   waiting[i] = false;
critical section
   j = (i+1) % n;
   while(j != i && !waiting[j])
       j = (j+1) % n;
   if(j == i)
       lock = false;
   else
       waiting[j] = false;
remainder section
}
```

3.4 Semaphores:

Semaphores refer to the integer variables that are primarily used to solve the critical section problem via combining two of the atomic procedures, wait and signal, for the process synchronization.

The definitions of signal and wait are given below:

Wait

It decrements the value of its A argument in case it is positive. In case S is zero or negative, then no operation would be performed.

```
wait(A)
{
        while (A<=0);
        A–;
}
```

Signal

This operation increments the actual value of its argument A.

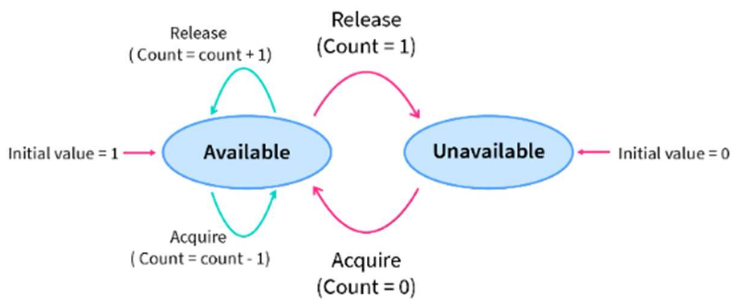
```
signal(A)
{
        A++;
}
```

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

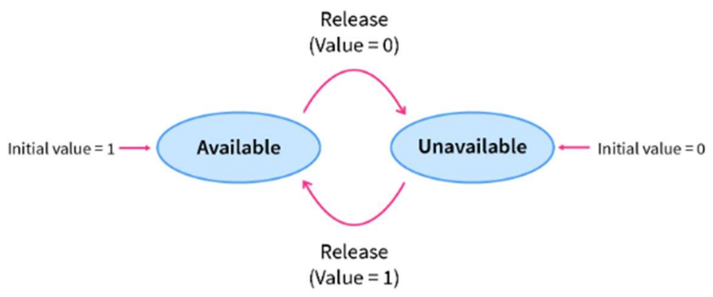
Counting Semaphores

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Its value can range over an unrestricted domain.



Binary Semaphores

This is also known as a mutex lock, as they are locks that provide mutual exclusion. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes and a single resource.



3.5 Classical Problems of synchronization:

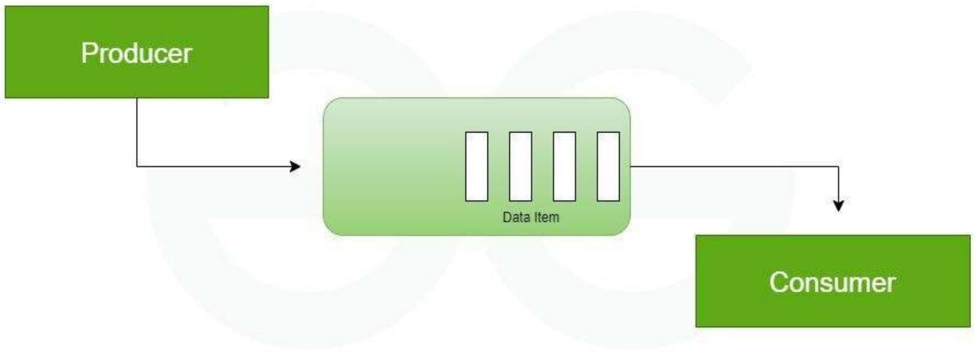
We will discuss the following three problems:

Bounded Buffer (Producer-Consumer) Problem
Dining Philosophers Problem
The Readers Writers Problem

Bounded Buffer Problem

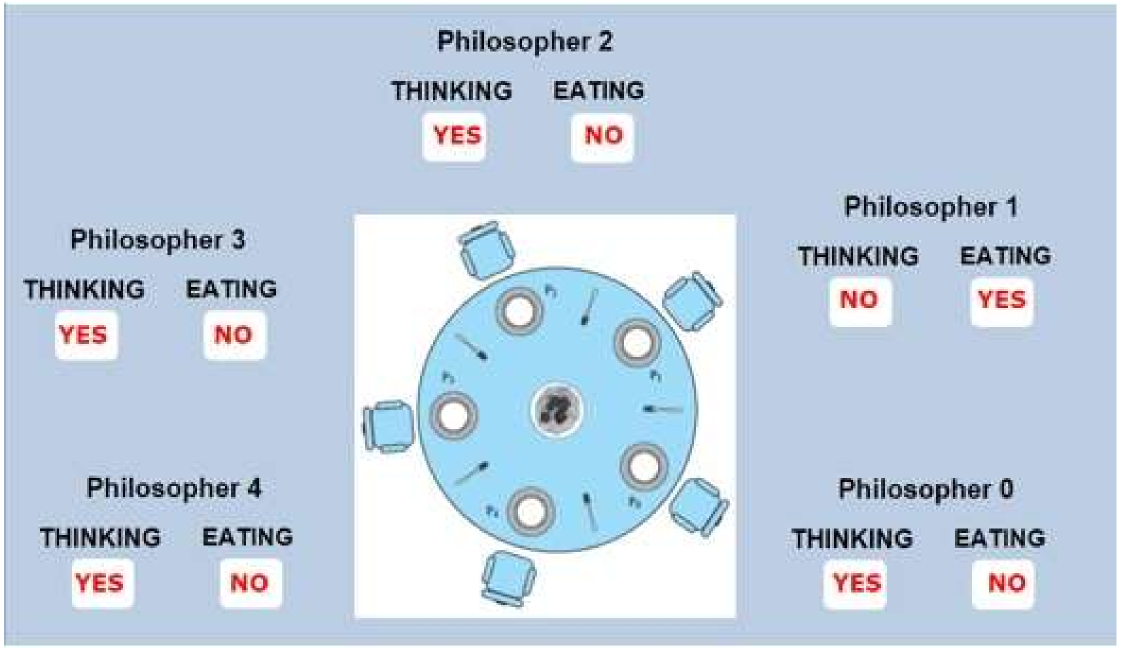
- Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.
- This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

- In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.



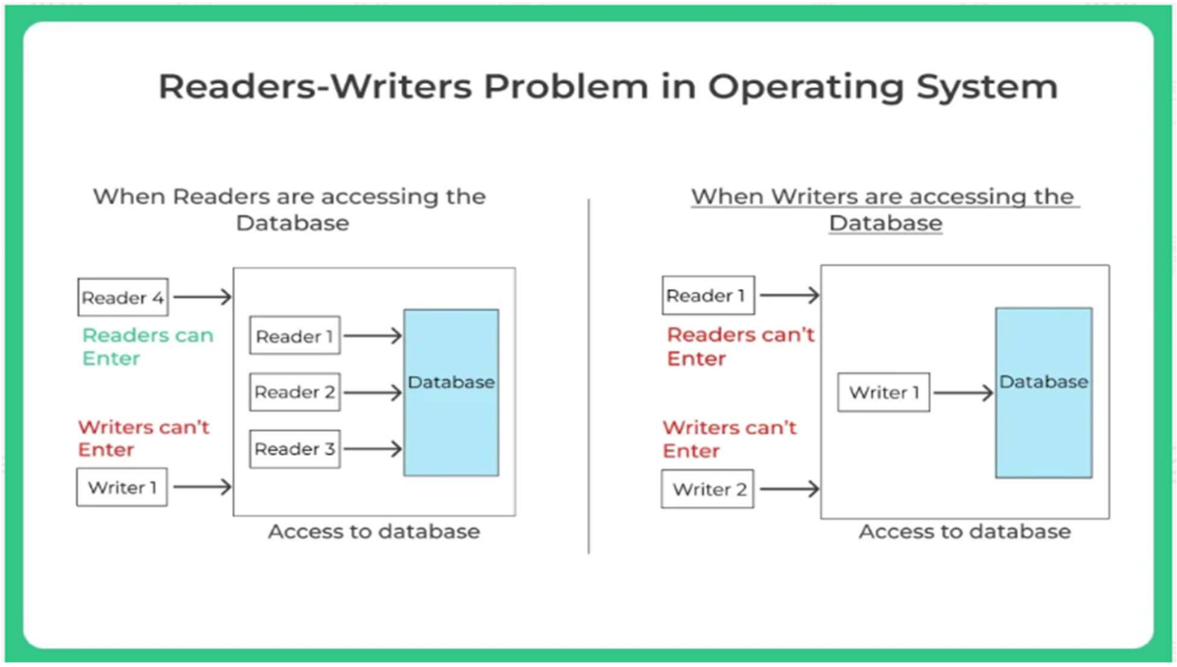
Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.



The Readers Writers Problem

- In this problem there are some processes (called readers) that only read the shared data, and never change it, and there are other processes (called writers) who may change the data in addition to reading, or instead of reading it.
- There is various type of readers-writers problem, most centered on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.

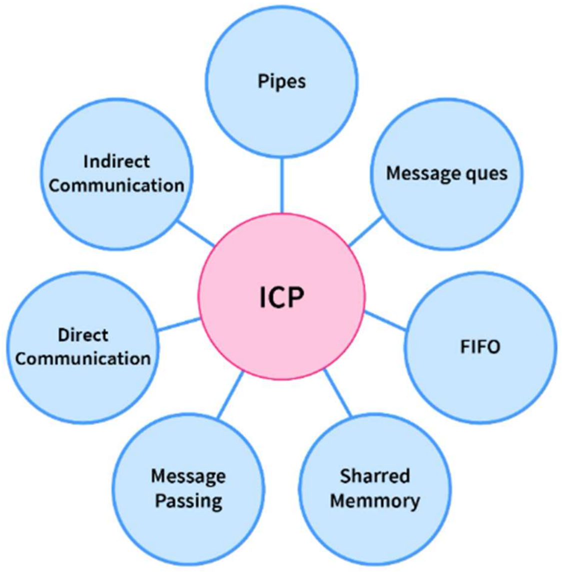


3.6 Inter Process communication:

Interprocess Communication or IPC provides a mechanism to exchange data and information across multiple processes, which might be on single or multiple computers connected by a network. This is essential for many tasks, such as:

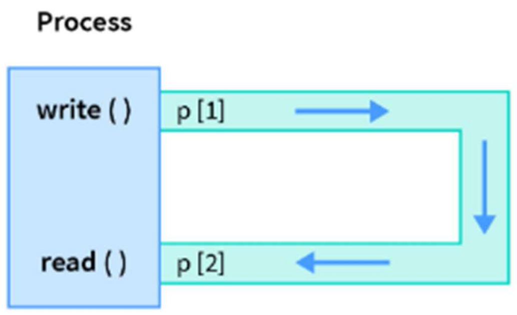
- Sharing data
- Coordinating activities
- Managing resources
- Achieving modularity

Approaches for Inter-Process Communication



a) Pipes

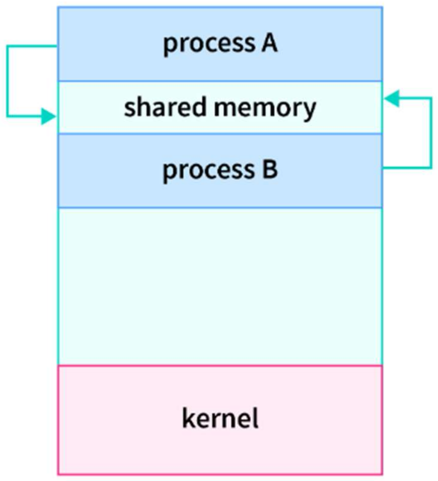
- Pipes are a simple form of shared memory that allows two processes to communicate with each other.
- It is a half-duplex method (or one way communication) used for IPC between two related processes.
- One process writes data to the pipe, and the other process reads data from the pipe.
- It is like a scenario like filling water with a tap into a bucket. The filling process is writing into the pipe and the reading process is retrieving from the pipe.
- Pipes can be either named or anonymous, depending on whether they have a unique name or not.
- Named pipes are a type of pipe that has a unique name, and can be accessed by multiple processes. Named pipes can be used for communication between processes running on the same host or between processes running on different hosts over a network.
- Anonymous pipes, on the other hand, are pipes that are created for communication between a parent process and its child process. Anonymous pipes are typically used for one-way communication between processes, as they do not have a unique name and can only be accessed by the processes that created them.



b) Shared Memory

- Shared memory is a region of memory that is accessible to multiple processes. This allows processes to communicate with each other by reading and writing data from the shared memory region.
- Shared memory is a fast and efficient way for processes to communicate, but it can be difficult to use if the processes are not carefully synchronized.
- There are two main types of shared memory:
    - **Anonymous shared memory:** Anonymous shared memory is not associated with any file or other system object. It is created by the operating system and is only accessible to the processes that created it.
    - **Mapped shared memory:** Mapped shared memory is associated with a file or other system object. It is created by mapping a file into the address space of one or more processes.

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.

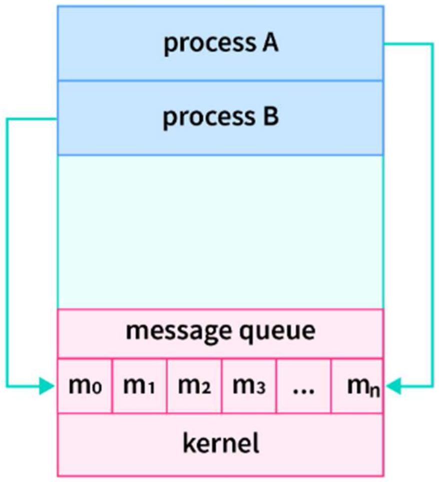


c) Message Passing

- 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.
- Processes can communicate without any shared variables, therefore it can be used in a distributed environment on a network.
- 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.

d) Message Queues:

- Message queues are a more advanced form of pipes.
- They allow processes to send messages to each other, and they can be used to communicate between processes that are not running on the same machine.
- Message queues are a good choice for communication between processes that need to be decoupled from each other.
- In Message Queue IPC, each message has a priority associated with it, and messages are retrieved from the queue in order of their priority.
- This allows processes to prioritize the delivery of important messages and ensures that critical messages are not blocked by less important messages in the queue.
- Message Queue IPC provides a flexible and scalable method of communication between processes, as messages can be sent and received asynchronously, allowing processes to continue executing while they wait for messages to arrive.



e) FIFO

- FIFO (First In First Out) is a type of message queue that guarantees that messages are delivered in the order they were sent.
- It involves the use of a FIFO buffer, which acts as a queue for exchanging data between processes.
- Used to communicate between two processes that are not related.
- In the FIFO method, one process writes data to the FIFO buffer, and another process reads the data from the buffer in the order in which it was written.
- Full-duplex method - Process P1 is able to communicate with Process P2, and vice versa.
- The main advantage of the FIFO method is that it provides a simple way for processes to communicate, as data is exchanged sequentially, and there is no need for processes to coordinate their access to the FIFO buffer.
- However, the FIFO method can also introduce limitations, as it may result in slow performance if the buffer becomes full and data must be written to the disk, or if the buffer becomes empty and data must be read from the disk.