Concurrency and synchronization

A process can be of two types:

- Independent process.
- Co-operating process.

- **Independent Processes** operating concurrently on a system are those that can neither affect other processes or be affected by other processes.
- **Cooperating processes** are those that can affect or are affected by other processes running on the system. Cooperating processes may share data with each other.

Reasons for needing cooperating processes

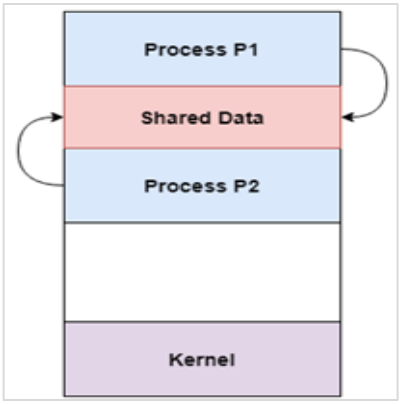
There may be many reasons for the requirement of cooperating processes. Some of these are given as follows −

- **Modularity**
  Modularity involves dividing complicated tasks into smaller subtasks. These subtasks can be completed by different cooperating processes. This leads to faster and more efficient completion of the required tasks.
- **Information Sharing**
  Sharing of information between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.
- **Convenience**
  There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.
- **Computation Speedup**
  Subtasks of a single task can be performed parallely using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

Methods of Cooperation

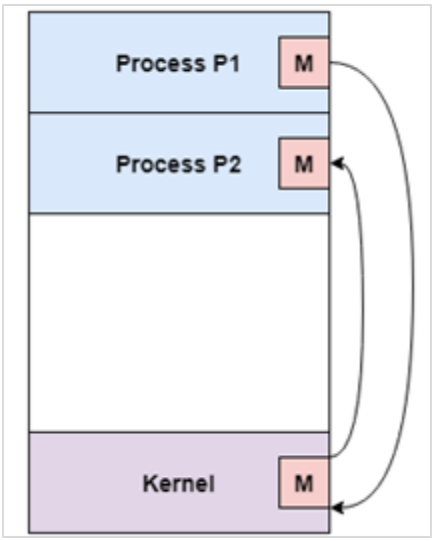
Cooperating processes can coordinate with each other using shared data or messages. Details about these are given as follows −

- **Cooperation by Sharing**
  The cooperating processes can cooperate with each other using shared data such as memory, variables, files, databases etc. Critical section is used to provide data integrity and writing is mutually exclusive to prevent inconsistent data.
  A diagram that demonstrates cooperation by sharing is given as follows −



In the above diagram, Process P1 and P2 can cooperate with each other using shared data such as memory, variables, files, databases etc.

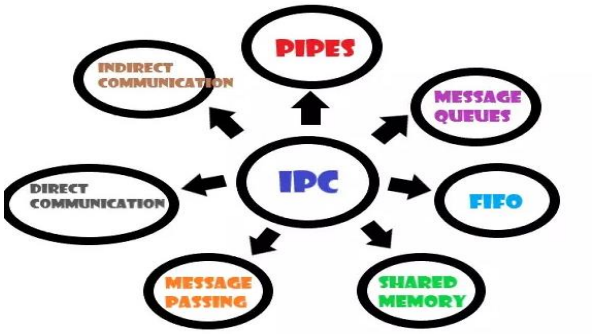
- **Cooperation by Communication**
  The cooperating processes can cooperate with each other using messages. This may lead to deadlock if each process is waiting for a message from the other to perform an operation. Starvation is also possible if a process never receives a message.
  A diagram that demonstrates cooperation by communication is given as follows −



In the above diagram, Process P1 and P2 can cooperate with each other using messages to communicate.

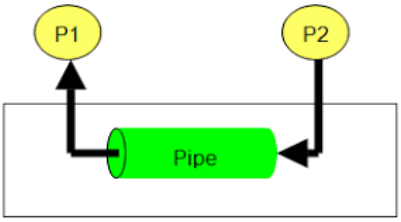
Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interfaces which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods



Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

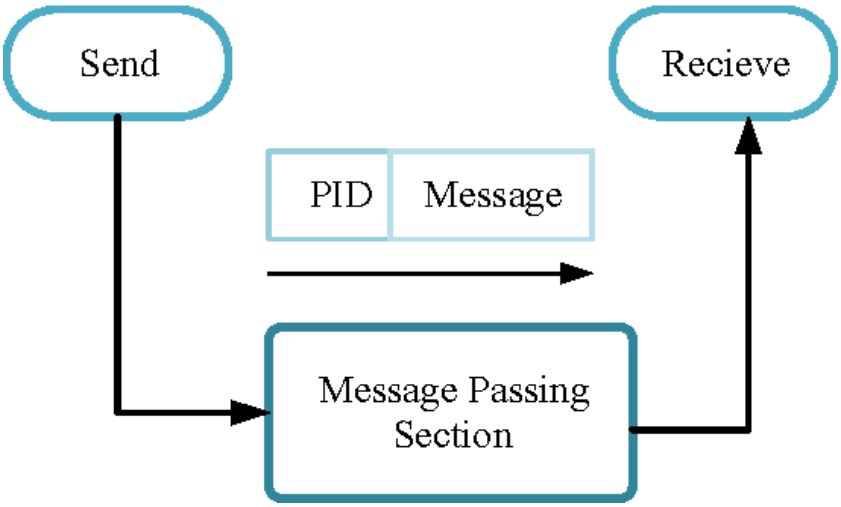


Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

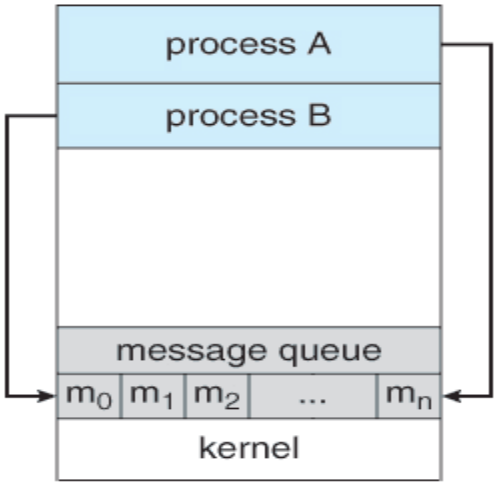
IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)



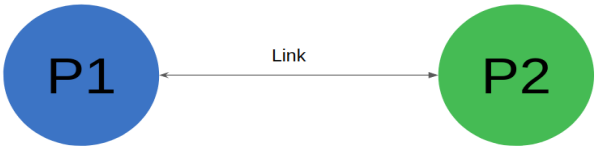
Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.



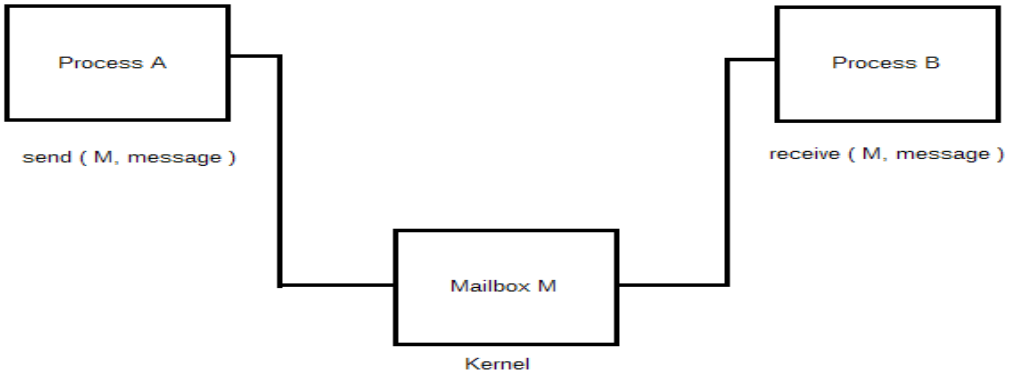
Direct Communication:

In this type of inter-process communication process, we should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.



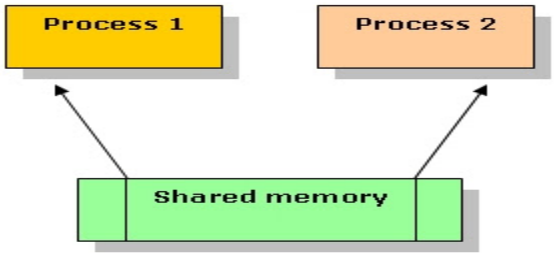
Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.



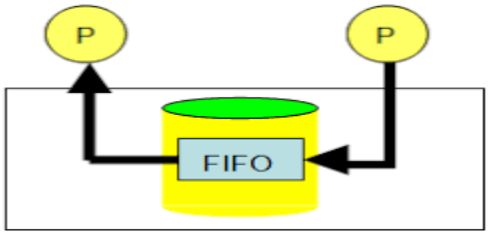
Shared Memory:

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory is protected from each other by synchronizing access across all the processes.



FIFO or named pipes

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.



Principles of concurrency

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

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

Advantages of Concurrency :

- **Running of multiple applications –**
  It enables us to run multiple applications at the same time.
- **Better resource utilization –**
  It enables the resources that are unused by one application can be used for other applications.
- **Better average response time –**
  Without concurrency, each application has to be run to completion before the next one can be run.
- **Better performance –**
  It enables better performance by the operating system. When one application uses only the processor and another application uses only the disk drive then the time to run both applications concurrently to completion will be shorter than the time to run each application consecutively.

Drawbacks of Concurrency :

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

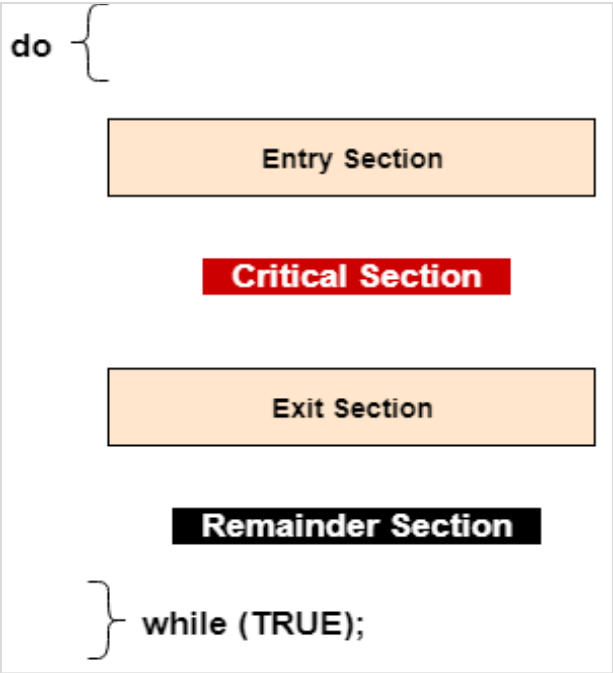
Issues of Concurrency :

- **Non-atomic –**
  Operations that are non-atomic but interruptible by multiple processes can cause problems.
- Race conditions –
  A race condition occurs if the outcome depends on which of several processes gets to a point first.
- **Blocking –**
  Processes can block waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.
- Starvation –
  It occurs when a process does not obtain service to progress.
- Deadlock –
  It occurs when two processes are blocked and hence neither can proceed to execute.

Process Synchronization

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.

A diagram that demonstrates the critical section is as follows –



In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

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 –

MUTUAL EXCLUSION:-Mutual exclusion is a property of process synchronization which states that "no two processes can exist in the critical section at any given point of time". Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

- **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. It should not wait endlessly to access the critical section

Hardware and software approaches

Process Synchronization problems occur when two processes running concurrently share the same data or same variable. The value of that variable may not be updated correctly before its being used by a second process. Such a condition is known as Race Around Condition. There are software as well as hardware solutions to this problem. In this article, we will talk about the most efficient hardware solution to process synchronization problems and its implementation.

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

1. Test and Set
2. Swap
3. Unlock and Lock

Hardware instructions in many operating systems help in effective solution of critical section problems.

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. So, again, when a process is in the critical section, no other process gets to enter it as the value of lock is true. 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(booelan &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.

```
// 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);

    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

}
```

Software Approach –

In Software Approach, Some specific Algorithm approach is used to maintain synchronization of the data. Like in Approach One or Approach Two, for a number of two process, a temporary variable like (turn) or boolean variable (flag) value is used to store the data. When the condition is True then the process in the waiting State, known as Busy Waiting State. This does not satisfy all the Critical Section requirements.

Another Software approach known as Peterson's Solution is best for Synchronization. It uses two variables in the Entry Section so as to maintain consistency, like Flag (boolean variable) and Turn variable(storing the process states). It satisfy all the three Critical Section requirements

Peterson's solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

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

The structure of process Pi in Peterson's solution. This solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

Semaphore

process synchronization tool, Semaphore is an integer variable S. This integer variable S is initialized to the number of resources present in the system. The value of semaphore S can be modified only by two functions wait() and signal() apart from initialization.

The wait() and signal() operation modifies the value of the semaphore S indivisibly. Which means when a process is modifying the value of the semaphore, no other process can simultaneously modify the value of the semaphore. Further, the operating system distinguishes the semaphore in two categories Counting semaphores and Binary semaphore.

In Counting Semaphore, the value of semaphore S is initialized to the number of resources present in the system. Whenever a process wants to access the shared resources, it performs wait() operation on the semaphore which decrements the value of semaphore by one. When it releases the shared resource, it performs a signal() operation on the semaphore which increments the value of semaphore by one.

When the semaphore count goes to 0, it means all resources are occupied by the processes. If a process need to use a resource when semaphore count is 0, it executes wait() and get blocked until a process utilizing the shared resources releases it and the value of semaphore becomes greater than 0.

In Binary semaphore, the value of semaphore ranges between 0 and 1. It is similar to mutex lock, but mutex is a locking mechanism whereas, the semaphore is a signalling mechanism. In binary semaphore, if a process wants to access the resource it performs wait() operation on the semaphore and decrements the value of semaphore from 1 to 0.

When process releases the resource, it performs a signal() operation on the semaphore and increments its value to 1. If the value of the semaphore is 0 and a process want to access the resource it performs wait() operation and block itself till the current process utilizing the resources releases the resource.

Monitor

To overcome the timing errors that occur while using semaphore for process synchronization, the researchers have introduced a high-level synchronization construct i.e. the monitor type. A monitor type is an abstract data type that is used for process synchronization.

An abstract data type monitor type contains the shared data variables that are to be shared by all the processes and some programmer-defined operations that allow processes to execute in mutual exclusion within the monitor. A process can not directly access the shared data variable in the monitor; the process has to access it through procedures defined in the monitor which allow only one process to access the shared variables in a monitor at a time.

The syntax of monitor is as follow:

```
monitor monitor_name

{

//shared variable declarations

procedure P1 ( . . . ) {

}

procedure P2 ( . . . ) {

}

procedure Pn ( . . . ) {
```

```
}

initialization code ( . . . ) {

}

}
```

A monitor is a construct such as only one process is active at a time within the monitor. If another process tries to access the shared variable in the monitor, it gets blocked and is lined up in the queue to get the access to shared data when previously accessing the process releases it.

Conditional variables were introduced for additional synchronization mechanisms. The conditional variable allows a process to wait inside the monitor and allows a waiting process to resume immediately when the other process releases the resources.

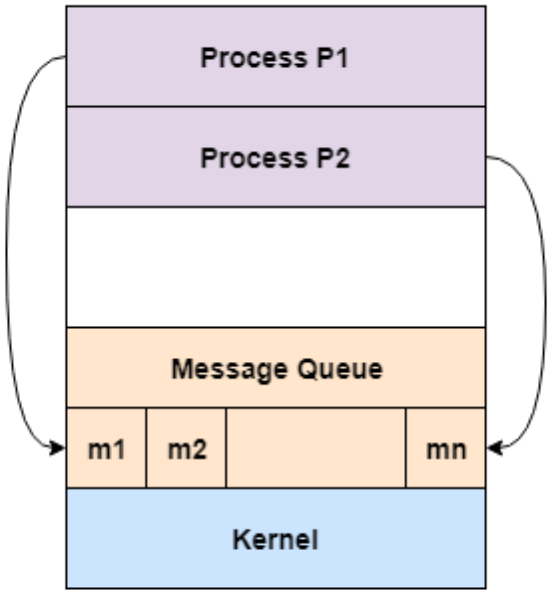
The conditional variable can invoke only two operations: wait() and signal(). Where if a process P invokes a wait() operation it gets suspended in the monitor till another process Q invokes signal() operation i.e. a signal() operation invoked by a process resumes the suspended process.

Message passing model

Process communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another. One of the models of process communication is the message passing model.

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message passing model of process communication is given as follows –



Message Passing Model

In the above diagram, both the processes P1 and P2 can access the message queue and store and retrieve data.

Advantages of Message Passing Model

Some of the advantages of message passing model are given as follows −

- The message passing model is much easier to implement than the shared memory model.
- It is easier to build parallel hardware using message passing model as it is quite tolerant of higher communication latencies.

Disadvantage of Message Passing Model

The message passing model has slower communication than the shared memory model because the connection setup takes time.

classical problems of synchronization

These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

1. Bounded-buffer (or Producer-Consumer) Problem,

2. Dining-Philosophers Problem,

3. Readers and Writers Problem,

producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

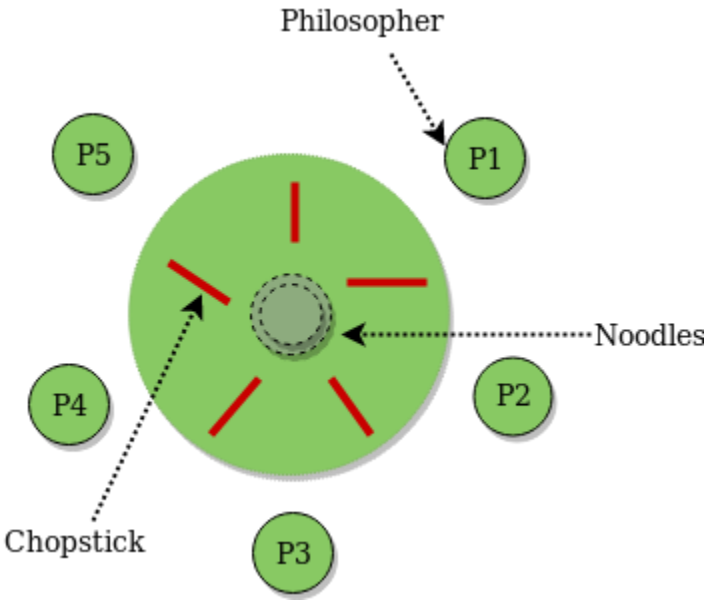
- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem : To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

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 way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

https://youtu.be/L-miVh3xUnQ

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



Semaphore Solution to Dining Philosopher –

Each philosopher is represented by the following pseudocode:

There are three states of the philosopher: **THINKING, HUNGRY, and EATING**. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or put down at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors

https://youtu.be/g__anK9O75k

Readers and Writers Problem

.Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

https://youtu.be/XWEn71mTShs

Functions for sempahore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

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

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.

- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {

  // Reader wants to enter the critical section
  wait(mutex);

  // The number of readers has now increased by 1
  readcnt++;

  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;
```

```
  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);          // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.