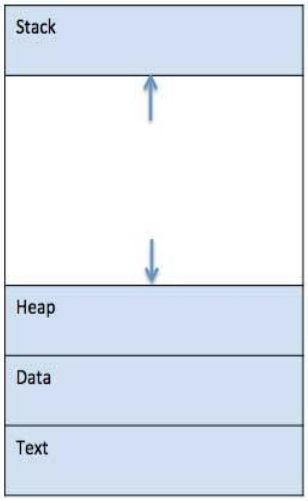
Unit-2nd – Concurrent Processes

Process Concept

A process is basically a program in execution. The execution of a process must progress in a sequential fashion. A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

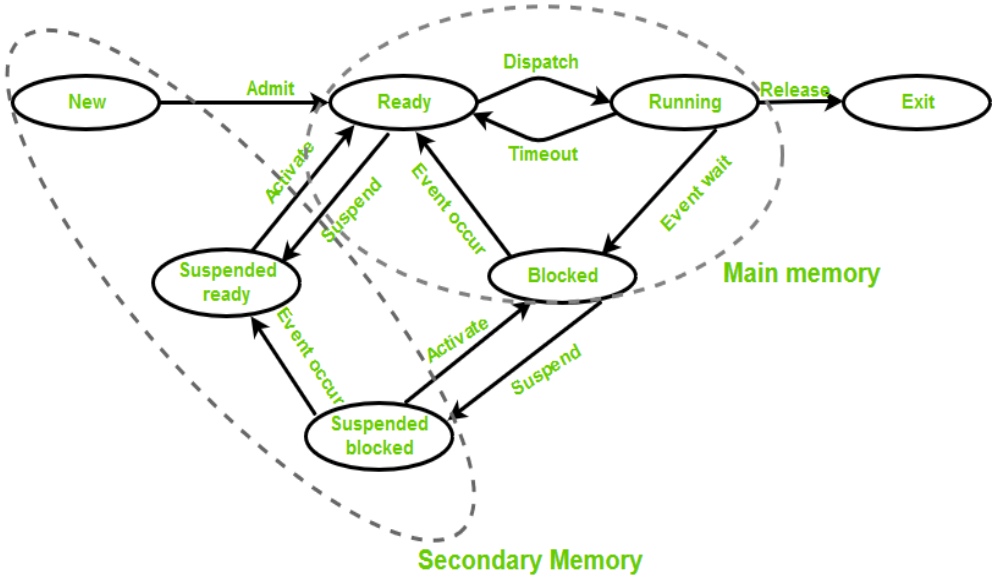
When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −



Process State: -

- **New (Create) –** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
- **Ready –** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
- **Run –** The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or wait –** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.

- **Terminated or completed –** Process is killed as well as PCB is deleted.
- **Suspend ready –** Process that was initially in the ready state but were swapped out of main memory (refer Virtual Memory topic) and placed onto external storage by scheduler are said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.
- **Suspend wait or suspend blocked –** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.



Principles of Concurrency: -

Concurrency It is the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.

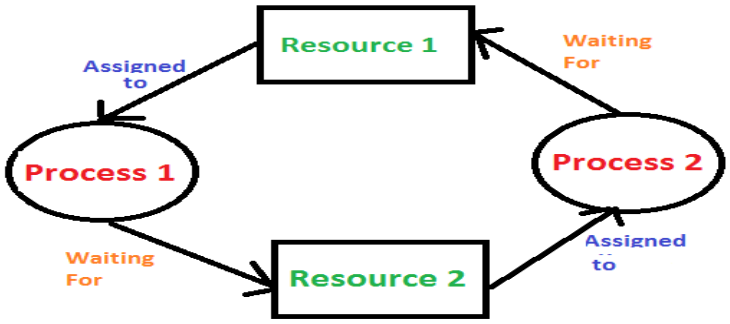
Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more **faults** within) some of its components.

Mutual Exclusion is a property of concurrency control, which is instituted for the purpose of preventing *race conditions**; it is the requirement that one thread of execution never enter its *critical section*** at the same time that another concurrent thread of execution enters its own critical section.

***Race condition** or **Race hazard** is the behavior of electronic, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended.

****Critical Section** is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.

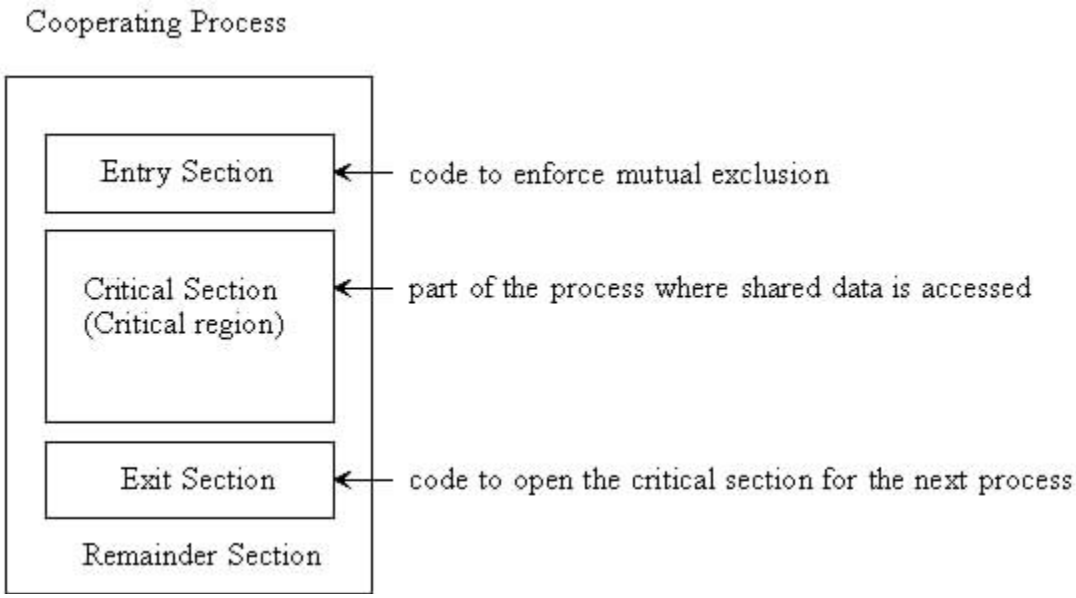
*** **Deadlocks**- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



Deadlock freedom can be expanded to implement one or both of these properties:

•**Lockout-freedom** guarantees that any process wishing to enter the critical section will be able to do so eventually. This is distinct from deadlock avoidance, which requires that some waiting process be able to get access to the critical section, but does not require that every process gets a turn. If two processes continually trade a resource between them, a third process could be locked out and experience resource starvation, even though the system is not in deadlock. If a system is free of lockouts, it ensures that every process can get a turn at some point in the future.

•**A k-bounded waiting property** gives a more precise commitment than lock out freedom. Lockout-freedom ensures every process can access the critical section eventually: it gives no guarantee about how long the wait will be. In practice, a process could be overtaken an arbitrary or unbounded number of times by other higher-priority processes before it gets its turn. Under a k-bounded waiting property, each process has a finite maximum wait time. This works by setting a limit to the number of times other processes can cut in line, so that no process can enter the critical section more than k times while another is waiting.



Block diagram of mutual exclusion (STEP WISE)

Producer–consumer problem: -

The **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, who 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. The problem is 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.

The solution for 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. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

To solve the problem, some programmer might come up with a solution shown below. In the solution two library routines are used, sleep and wakeup . When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable itemCount holds the number of items in the buffer.

```
int itemCount = 0;
Operating System
procedure producer()
{
while (true)
{
item = produceItem();
if (itemCount == BUFFER_SIZE)
{
sleep();
}
putItemIntoBuffer(item);
itemCount = itemCount + 1;
if (itemCount == 1)
{
wakeup(consumer);
}
}
}
procedure consumer()
{
while (true)
{
if (itemCount == 0)
{
sleep();
}
item = removeItemFromBuffer();
itemCount = itemCount - 1;
if (itemCount == BUFFER_SIZE - 1)
{
wakeup(producer);
}
consumeItem(item);
}
}
```

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”. The term was first coined by **Dijkstra**. 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. To understand mutual exclusion, let’s take an example.

Example:

In the clothes section of a supermarket, two people are shopping for clothes.

Boy A decides upon some clothes to buy and heads to the changing room to try them out. Now, while boy A is inside the changing room, there is an ‘occupied’ sign on it – indicating that no one else can come in. Girl B has to use the changing room too, so she has to wait till boy A is done using the changing room.

Once boy A comes out of the changing room, the sign on it changes from ‘occupied’ to ‘vacant’ – indicating that another person can use it. Hence, girl B proceeds to use the changing room, while the sign displays ‘occupied’ again.

The changing room is nothing but the critical section, boy A and girl B are two different processes, while the sign outside the changing room indicates the process synchronization mechanism being used.

The Critical Section Problem: -

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device. The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section. The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

Requirements of Synchronization mechanisms

Primary

1. **Mutual Exclusion: -** Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.

2. 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.

Secondary

1. Bounded Waiting: - We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

2. Architectural Neutrality: - Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Dekker's Algorithm: -

Dekker's algorithm is the first solution of critical section problem. There are many versions of this algorithms, the 5th or final version satisfies the all the conditions below and is the most efficient among all of them.

The solution to critical section problem must ensure the following three conditions:

- Mutual Exclusion
- Progress
- Bounded Waiting

1. First version

- Dekker's algorithm succeeds to achieve mutual exclusion.
- It uses variables to control thread execution.
- It constantly checks whether critical section available.

Problem

The problem of this first version of Dekker's algorithm is implementation of lockstep synchronization. It means each thread depends on other to complete its execution. If one of the two processes completes its execution, then the second process runs. Then it gives access to the completed one and waits for its run. But the completed one would never run and so it would never return access back to the second process. Thus the second process waits for infinite time.

2. Second Version

In Second version of Dekker's algorithm, lockstep synchronization is removed. It is done by using two flags to indicate its current status and updates them accordingly at the entry and exit section.

Problem

Mutual exclusion is violated in this version. During flag update, if threads are preempted then both the threads enter into the critical section. Once the preempted thread is restarted, also the same can be observed at the start itself, when both the flags are false.

3. Third Version

In this version, critical section flag is set before entering critical section test to ensure mutual exclusion.

Problem

This version failed to solve the problem of mutual exclusion. It also introduces deadlock possibility, both threads could get flag simultaneously and they will wait for infinite time.

4. Fourth Version

In this version of Dekker's algorithm, it sets flag to false for small period of time to provide control and solves the problem of mutual exclusion and deadlock.

Problem

Indefinite postponement is the problem of this version. Random amount of time is unpredictable depending upon the situation in which the algorithm is being implemented; hence it is not acceptable in case of business critical systems.

5. Fifth Version (Final Solution)

In this version, flavored thread motion is used to determine entry to critical section. It provides mutual exclusion and avoiding deadlock, indefinite postponement or lockstep synchronization by resolving the conflict that which thread should execute first. This version of Dekker's algorithm provides the complete solution of critical section problems.

Peterson Solution: -

This is a software mechanism implemented at user mode. It is a busy waiting solution can be implemented for only two processes. It uses two variables that are turn variable and interested variable.

The Code of the solution is given below

```
# define N 2
# define TRUE 1
# define FALSE 0
```

```
int interested[N] = FALSE;
int turn;
voidEntry_Section (int process)
{
   int other;
   other = 1-process;
   interested[process] = TRUE;
   turn = process;
   while (interested [other] =True && TURN=process);
}
voidExit_Section (int process)
{
   interested [process] = FALSE;
}
```

Till now, each of our solution is affected by one or the other problem. However, the Peterson solution provides you all the necessary requirements such as Mutual Exclusion, Progress, Bounded Waiting and Portability.

Analysis of Peterson Solution

```
voidEntry_Section (int process)
{
   1. int other;
   2. other = 1-process;
   3. interested[process] = TRUE;
   4. turn = process;
   5. while (interested [other] =True && TURN=process);
}
```

Critical Section

```
voidExit_Section (int process)
{
   6. interested [process] = FALSE;
}
```

This is a two process solution. Let us consider two cooperative processes P1 and P2. The entry section and exit section are shown below. Initially, the value of interested variables and turn variable is 0.

Initially process P1 arrives and wants to enter into the critical section. It sets its interested variable to True (instruction line 3) and also sets turn to 1 (line number 4). Since the condition given in line number 5 is completely satisfied by P1 therefore it will enter in the critical section.

P1 → 1 2 3 4 5 CS

Meanwhile, Process P1 got preempted and process P2 got scheduled. P2 also wants to enter in the critical section and executes instructions 1, 2, 3 and 4 of entry section. On instruction 5, it got stuck since it doesn't satisfy the condition (value of other interested variable is still true). Therefore it gets into the busy waiting.

P2 → 1 2 3 4 5

P1 again got scheduled and finish the critical section by executing the instruction no. 6 (setting interested variable to false). Now if P2 checks then it are going to satisfy the condition since other process's interested variable becomes false. P2 will also get enter the critical section.

P1 → 6

P2 → 5 CS

Any of the process may enter in the critical section for multiple numbers of times. Hence the procedure occurs in the cyclic order.

Mutual Exclusion: - The method provides mutual exclusion for sure. In entry section, the while condition involves the criteria for two variables therefore a process cannot enter in the critical section until the other process is interested and the process is the last one to update turn variable.

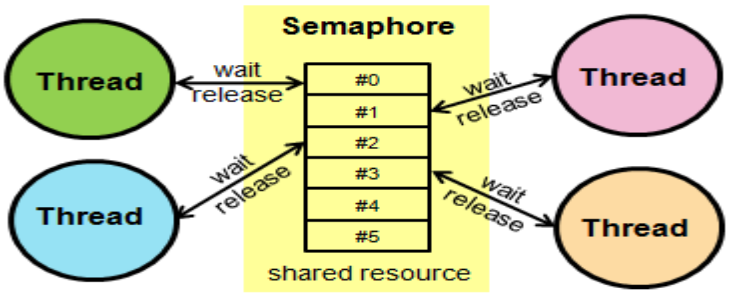
Progress: - An uninterested process will never stop the other interested process from entering in the critical section. If the other process is also interested then the process will wait.

Bounded waiting: - The interested variable mechanism failed because it was not providing bounded waiting. However, in Peterson solution, A deadlock can never happen because the process which first sets the turn variable will enter in the critical section for sure. Therefore, if a process is preempted after executing line number 4 of the entry section then it will definitely get into the critical section in its next chance.

Portability: - This is the complete software solution and therefore it is portable on every hardware.

Semaphore: -

A semaphore is a variable used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.



An important semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions. The variable is then used as a condition to control access to some system resource.

A useful way to think of a semaphore as used in the real-world systems is as a record of how many units of a particular resource are available, coupled with operations to adjust that record *safely* (i.e. to avoid race conditions) as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks*.

***Locks**: They are methods of synchronization used to prevent multiple threads from accessing a resource at the same time.

When used to control access to a pool of resources, a semaphore tracks only *how many* resources are free; it does not keep track of *which* of the resources are free. Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

The success of the protocol requires applications follow it correctly. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:

a. Requesting a resource and forgetting to release it;
b. Releasing a resource that was never requested;
c. Holding a resource for a long time without needing it;
d. Using a resource without requesting it first (or after releasing it).

Even if all processes follow these rules, *multi-resource deadlock* may still occur when there are different resources managed by different semaphores and when processes need to use more than one resource at a time.

Counting semaphores are equipped with two operations, historically denoted as P and V. Operation V increments the semaphore *S*, and operation P decrements it.

The value of the semaphore *S* is the number of units of the resource that are currently available. The P operation wastes time or sleeps until a resource protected by the semaphore becomes available, at which time the resource is immediately claimed. The V operation is the inverse: it makes a resource available again after the process has finished using it. One important property of semaphore *S* is that its value cannot be changed except by using the V and P operations.

A simple way to understand wait (P) and signal (V) operations is:

• **Wait (*sleep*):** If the value of semaphore variable is not negative, decrement it by 1. If the semaphore variable is now negative, the process executing wait is blocked (i.e., added to the semaphore's queue) until the value is greater or equal to 1. Otherwise, the process continues execution, having used a unit of the resource.

• **Signal (*wake up*):** Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.

Examples of semaphore

Login queue: - Consider a system that can only support ten users (S=10). Whenever a user logs in, P is called, decrementing the semaphore *S* by 1. Whenever a user logs out, V is called, incrementing *S* by 1 representing a login slot that has become available. When *S* is 0, any users wishing to log in must wait until ($S > 0$) and the login request is enqueued onto a FIFO queue; mutual exclusion is used to ensure that requests are enqueued in order. Whenever *S* becomes greater than 0 (login slots available), a login request is dequeued, and the user owning the request is allowed to log in.

Test and Set Operation: -

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
}
```

Classical Problem in Concurrency

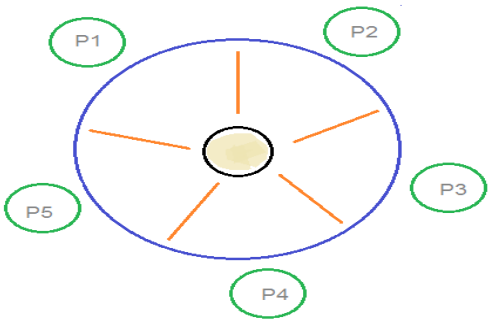
1. Dining Philosophers Problem

The dining philosopher's problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Problem: - Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

At any instant, a philosopher is either eating or thinking. 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.

Solution: - From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating. An array of five semaphores, **stick[5]**, for each of the five chopsticks.



When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down. But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

• A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

• Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

2. Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem: - There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource.

Solution: - From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

3. Sleeping Barber Problem

There is one barber in the barber shop, one barber chair and n chairs for waiting customers. If there are no customers, the barber sits down in the barber chair and takes a nap. An arriving customer must wake the barber. Subsequent arriving customers take a waiting chair if any are empty or leave if all chairs are full. This problem addresses race conditions.

This solution uses three semaphores, one for customers (counts waiting customers), one for the barber (idle - 0 or busy - 1) and a mutual exclusion semaphore, mutex. When the barber arrives for work, the barber procedure is executed blocking the barber on the customer semaphore until a customer arrives.

When a customer arrives, the customer procedure is executed which begins by acquiring mutex to enter a critical region. Subsequent arriving customers have to wait until the first customer has released mutex. After acquiring mutex, a customer checks to see if the number of waiting customers is less than the number of chairs.

If not, mutex is released and the customer leaves without a haircut. If there is an available chair, the waiting counter is incremented, the barber is awaken, the customer releases mutex, the barber grabs mutex, and begins the haircut. Once the customer's hair is cut, the customer leaves. The barber then checks to see if there is another customer. If not, the barber takes a nap.

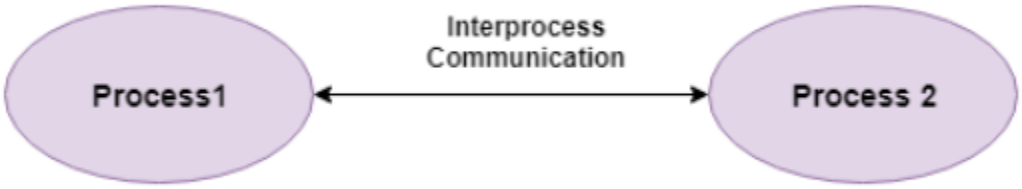
```
#define CHAIRS 5
Typedef int semaphore;
semaphore customers = 0;
```

```
semaphore barbers=0;
semaphore mutex=1;
int waiting=0;
void barber(void)
{
while (TRUE) {
P(customers); /*go to sleep if no customers*/
P(mutex);
waiting=waiting-1;
V(barbers);
V(mutex);
cut_hair();
}
}
void customer(void)
{
P(mutex);
if (waiting less than CHAIRS) {
waiting=waiting+1;
V(customers);
V(mutex);
P(barbers);
get_haircut();
} else {
V(mutex);
}
}
```

Interprocess Communication: -

Interprocess 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.

A diagram that illustrates interprocess communication is as follows −



The models of interprocess communication are as follows −

1. **Shared Memory Model: -** Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

Advantage: - Memory communication is faster on the shared memory model as compared to the message passing model on the same machine.

Disadvantages: -

- All the processes that use the shared memory model need to make sure that they are not writing to the same memory location.
- Shared memory model may create problems such as synchronization and memory protection that need to be addressed.

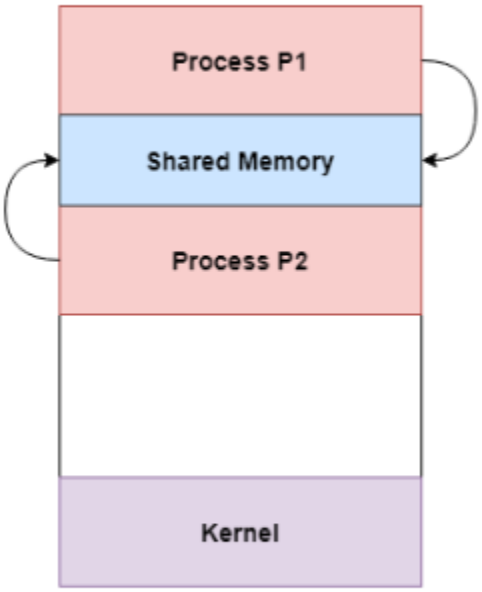
2. **Message Passing Model: -** Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

Advantage: - The message passing model is much easier to implement than the shared memory model.

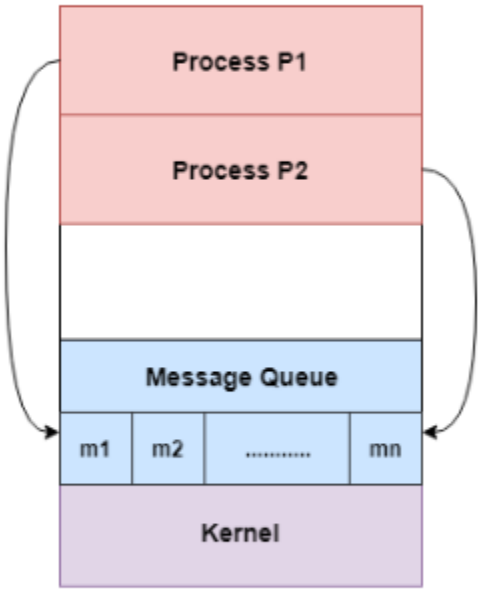
Disadvantage: - The message passing model has slower communication than the shared memory model Because the connection setup takes time.

A diagram that demonstrates the shared memory model and message passing model is given as follows

Models of Interprocess Communication



Shared Memory Model



Message Passing Model