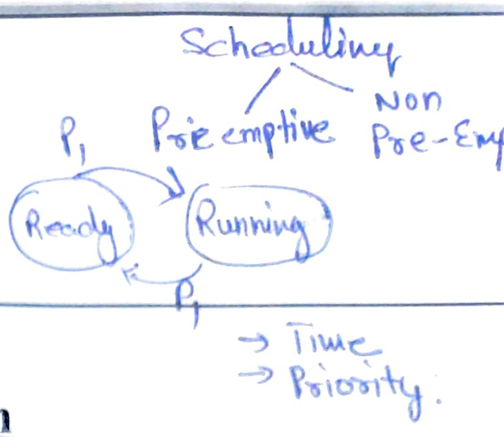
UNIT-2



Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:
Independent Process- Execution of one process does not affect the execution of other processes.
Cooperative Process- Execution of one process affects the execution of other processes.

Race Condition- Several processes access and process the manipulations over the same data concurrently, and then the outcome depends on the particular order in which the access takes place.

Example1-Let P1 and P2 are cooperative processes and x is a shared variable.

x=5← Shared variable

P1	P2
A=x;	B=x;
A++;	B--;
sleep (1);	sleep (1);
x=A;	x=B;

We are expecting the final value of xwill be 5 but due to race condition final value of x will be either 4 or 6. ✓

Example2-Let P1 and P2 are cooperative processes and x is a shared variable.

x=5← Shared variable

P1	P2
A=2*x	B=x+2
Print(A)	Print(B)

{10,7} and {7,10}

✓ We can get more than one output like ~~{10,12},{7,14},{12,10}~~ and ~~{14,7}~~ due to race condition for above cooperative processes.

Critical Section

The critical section is a code segment where the shared variables can be accessed. 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.

The critical section is given as follows:

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

Entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. In the entry section, the process requests for entry in the **Critical Section.**
Exit section handles the exit from the critical section. It releases the resources and also informs the other processes that critical section is free.

Requirements of solution to critical section Problem-

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

1. **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. It is essential requirement that must be satisfied by the solution of critical section problem.
2. **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. It is essential requirement that must be satisfied by the solution of critical section problem.
3. **Bounded Waiting**
   A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
4. No assumption related to H/w, speed etc.

Lock 0 → vacant
1 → full

ritical Section Solution using lock:-   (lock != 0)

```
do {
    acquire lock
        CS
    release lock
}
```

```
1. While (lock == 1);      entry
2.   Lock = 1;             code
3. Critical Section
4. lock = 0                Exit code
```

Dekker's algorithm

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming.If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes. Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvationOne advantage of this algorithm is that it doesn't require special test-and-set (atomic read/modify/write) instructions and is therefore highly portable between languages and machine architectures. One disadvantage is that it is limited to two processes.

1st Version of Dekker's Solution-

P1	**P2**
while(true)	**while(true)**
{	{
while(turn!=0);	**while(turn!=1);**
$\boxed{\text{CS}}$	$\boxed{\text{CS}}$
turn=1;	**turn=0;**
$\boxed{\text{RS}}$	$\boxed{\text{RS}}$
}	}

Mutual Exclusion is assured but Progress is not assured in this solution.

2nd Version of Dekker's Solution-

P1	P2
while(true)	while(true)
{	{
flag[0]=true;	flag[1]=true;
while(flag[1])	while(flag[0])
$\boxed{\text{CS}}$	$\boxed{\text{CS}}$
flag[0]=false;	flag[1]=false;
$\boxed{\text{RS}}$	$\boxed{\text{RS}}$
}	}

Mutual Exclusion is assured and Progress is assured in this solution but deadlock may occur in this solution.

Turn variable (strict Alteration)

$\boxed{P_0}$ P_1 X
CS

$\boxed{P_1}$ P_0 X

	Process P_0	Process P_1
	No CS	No CS
entry code →	while (turn != 0);	while (turn != 1);
	$\boxed{\text{Critical Section}}$	$\boxed{\text{Critical Section}}$
exit code →	turn = 1;	turn = 0;

Peterson's Solution

- ❖ Peterson's Solution is a classical **software based solution** to the critical section problem.
- ❖ In Peterson's solution, we have two shared variables:
  - boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
  - int turn : The process whose turn is to enter the critical section.

P1	P2
while(true)	**while(true)**
{	{
flag[0]=true;	**flag[1]=true;**
turn=1;	**turn=0;**
while(turn==1&&flag[1]==true);	**while(turn==0&&flag[0]==true);**
CS	CS
flag[0]=false;	**flag[1]=false;**
RS	RS
}	}

Peterson's Solution preserves all three conditions:

- ❖ Mutual Exclusion is assured as only one process can access the critical section at any time.
- ❖ Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- ❖ Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- ❖ It involves Busy waiting
- ❖ It is limited to 2 processes.

Test and Set

Test and Set are a **hardware solution** to the synchronization problem. In Test and Set, we have a shared lock variable which can take either of the two values, 0 or 1.
0 Unlock
1 Lock
Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting till it become free and if it is not locked, it takes the lock and executes the critical section.
In Test and Set, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

The **test-and-set** instruction is an instruction used to write 1 (set) to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation. If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished. A CPU may use a test-and-set instruction offered by another electronic component
A lock can be built using an atomic test-and-set instruction as follows:

```
function Lock(boolean *lock) {
  while (test_and_set(lock) == 1); }
```

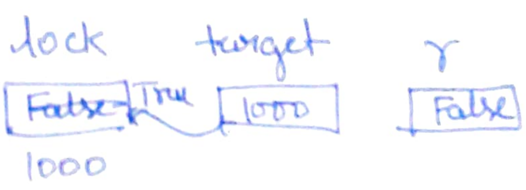
The calling process obtains the lock if the old value was 0, otherwise the while-loop spins waiting to acquire the lock.

4 Conditions.
1. Mutual Exclusion } must
2. Progress } must
3. Bounded wait
4. No assumption related to H/w, speed etc. } optional

```
while (test_and_set(&lock));
    [CS]
  lock = False;
```

$\Longleftrightarrow$

```
boolean test_and_set(boolean *target)
{
    boolean r = *target
    *target = True;
    return r;
}
```



Semaphore

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal (). The wait () operation is also called P and signal () is also called V. Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronization.

Definition of wait () Down wait

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

Definition of signal () Up signal

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

```
Down (Semaphore S)
{
S.value = S.value-1;
 if (S.value < 0)
 {
 Put Process(PCB)
 in Suspended list
 } Sleep();
else
 { S return };
```

```
Up (Semaphore S)
{
 S.value = S.value+1;
 if (S.value <= 0)
 {
   Select a Process from
 suspended list.
    wake up();
 }
}
```

All modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously

$\boxed{S=0} \xrightarrow{\text{means}}$ No more process

$\boxed{S=-3} \xrightarrow{\text{means}}$ 3 process in blocked li

Application of Semaphore

1. Solving Multi process Critical Section Problem
2. Resource allocation among various processes
3. Ordering Execution of processes.

Types of Semaphore

1. **Binary Semaphore -** This is also known as **mutex lock**. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement solution of critical section problem with multiple processes. 0 1
2. **Counting Semaphore -** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances. → −∞ to ∞

```
Down (Semaphore S)
{
  if (S.value == 1)
  {
    S.value = 0;
  }   CS
  else
  {
   Block the process
  and place in suspended
  list, sleep();
  }
}
```

```
Up (Semaphore S)
{ if (suspended list is empty)
  {
       S.value = 1;
  }
  else
  {
    select a process from suspen
  list and wake up();
  }
```

―

Producer Consumer Problem

Problem Statement -We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores -
mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer -

```
do{

        //produce an item

        wait(empty);
        wait(mutex);

        //place in buffer

        signal(mutex);
        signal(full);

}while(true);
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

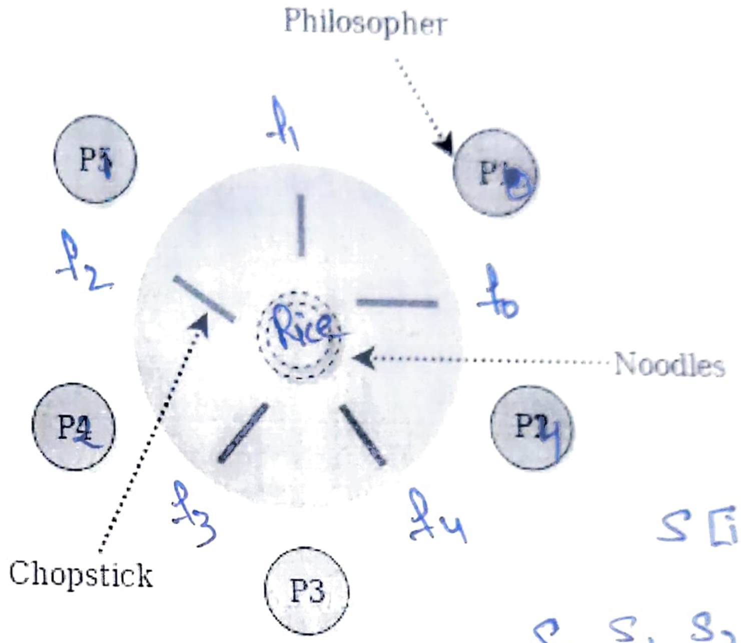
Solution for Consumer -

```
do{

        wait(full);
        wait(mutex);

        // remove item from buffer

        signal(mutex);
        signal(empty);

        // consumes item

}while(true);
```

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

The Dining Philosopher Problem

Problem Statement- 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.



[5 philosopher
5 fork]

S[i] five semaphore

S_0 S_1 S_2 S_3 S_4
1 1 1 1 1

Semaphore Solution to Dining Philosopher –
Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
 {        THINK;
          PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
          EAT;
          PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

P_0	S_0	S_1
P_1	S_1	S_2
P_2	S_2	S_3
P_3	S_3	S_4
P_4	S_4	S_5

Deadlock

$S_5 \rightarrow$ reverse order S_5 S_4

Solution

There are three states of 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 putdown 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.

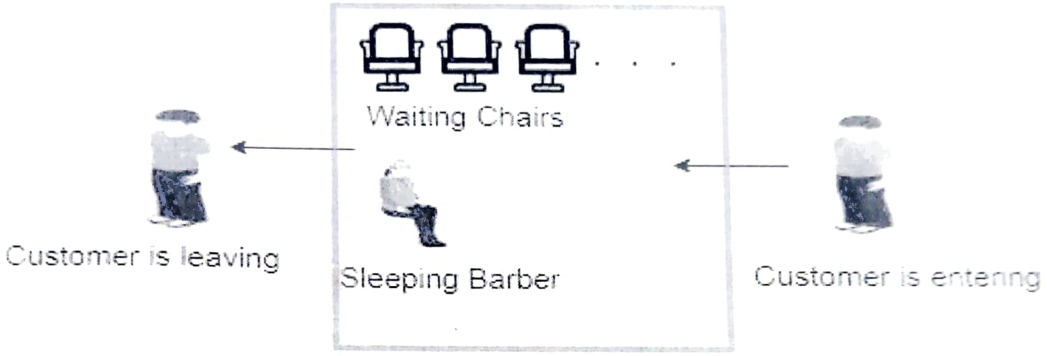
```
void philosopher (void)
{
 while (true)
 {
   Thinking();
entry { wait (take fork (S_i));
      { wait ( take fork (S_(i+1) mod N)
   EAT();
   signal (put fork (i));
   signal (put fork (i+1)% N);
 }
}
```

In Computer System, the sleeping barber problem is a classic IPC & Synchronization problem between multiple OS process.

Sleeping Barber Problem

Problem Statement- The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



Solution - The solution to this problem includes three semaphores. First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.

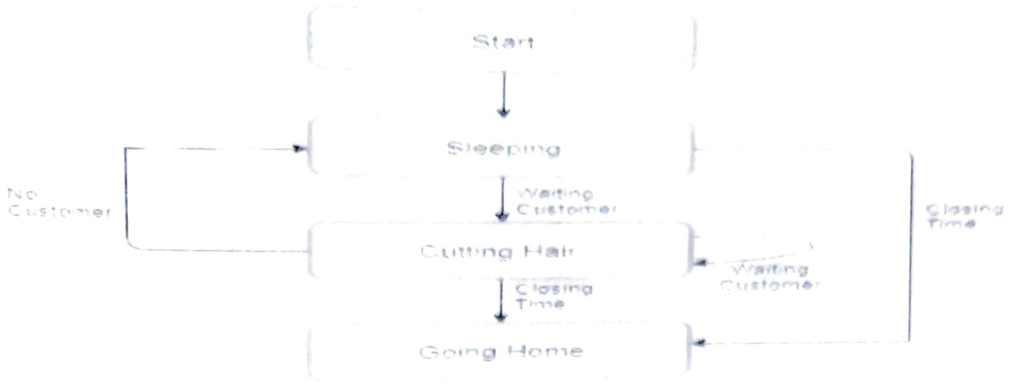
When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less than the number of chairs then he sits otherwise he leaves and releases the mutex.

If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping.

At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

Conditions:-

- 1 waiting room with N chairs
- 1 barber room with 1 barber chair



Inter process communication

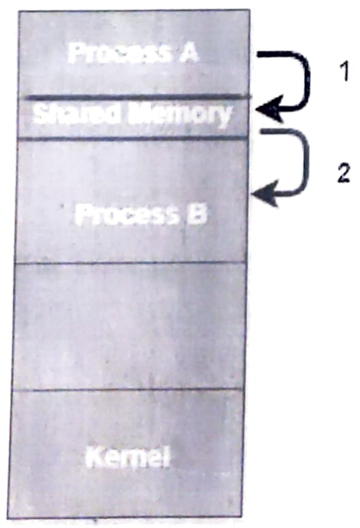
Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. Processes can communicate with each other using these two ways:

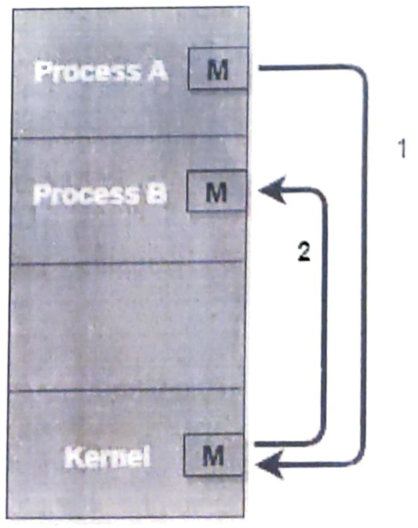
1. Shared Memory
2. Message passing

Shared Memory Method

Shared memory requires processes to share some variable and it completely depends on how programmer will implement it.

One way of communication using shared memory can be- Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.





Messaging Passing Method

In this method, processes communicate with each other without using any kind of of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

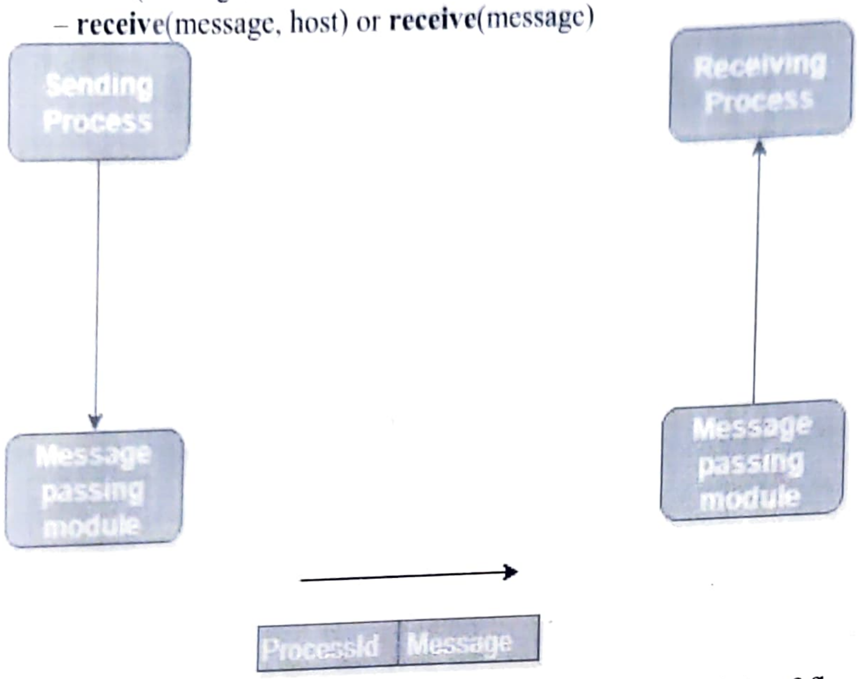
- Establish a communication link (if a link already exists, no need to establish it again.)

- Start exchanging messages using basic primitives.

We need at least two primitives:

– **send**(message, destinaion) or **send**(message)

– **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.** The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

sleeping barber Sol.

Variables :- Shared data

```
        semaphore customer = 0;
        semaphore barber = 0;
        access seat mutex = 1;      int Number of free Seats = N;
barber                            | customer :
  1. while (true)                 | while(1)
  2. { wait (customer); // wait for customer |   { sem_wait(access seat);
                        (asleep)  |     if ( Number of free Seat >0)
  3. wait (mutex); // whenever wait(1) |   { Number of free Seat --; // sitting
         is executes it           |                                    down
         decrease value to 0      |   sema_post (customers); // notify the barb
         in mutex to protect      |   sema-post (access-post); // realease the lo
         the no. of available     |   Sema-wait (barber); // wait in the
         seat.                    |                          waiting room i
                 free             |      customer is having   barber is busy
  4. Number of Seat ++;           |      hair cut
           // a chair             |   } else
  (signal)  gets free             |
  5. Sem-post (barber);           |
  . Sem-post (Mutex); // realease the mutex
                        on the chair
       // barber is cutting hair
  }
```