UNIT:3 Process communication and synchronization

DATE : / /
PAGE :

Principle of Concurrency

⇒ Concurrency is the execution of a set of multiple instruction sequences at the same time. This occur when there are several process threads running in parallel.

⇒ process that co-exist on the memory at a given time are called con-current process. The concurrent process may either be independent or co-operating.

⇒ Todays technology, like multicore processors and parallel processing, allow multiple processes and threads to be executed simultaneously. multiple process and threads can be excess the same memory space, the same declared variables in odd or even read or write to the same file.

⇒ The amount of time it takes for a process to execute is not easily calculated so we are unable to predict which process is complete first, there by allow us to implement algorithm to deal with the issue that concurrency threads. The amount of time a process takes to complete depends on the following.

① The activity of other process.
② The way OS handles interrupt.
③ The scheduling policies of OS.

problems in concurrency.

① Sharing global resources.

sharing of global resources safely is difficult. If two processes both make use of a global variable and both make change to the variable's value, then the order in which various change takes are executed is critical.

DATE : __/__/__
PAGE : _____

② Optimal allocation of resources.
It is difficult for OS to manage the allocation of resources optimally.

③ Locating programming errors.
It is difficult to spot a programming error because reports are usually repeatable due to varying states of shared components each time code is executed.

④ Looking the channel
It may in-efficient for the OS to simply lock the resources and prevents its use by other process.

Advantages of concurrency.

① Running of multiple application: Having concurrency allow the resource that are not being use by one application can be used for another application.
② Better resource utilization: Having concurrency allow the resource that are not being used by one application can be used for another application.
③ Better average response time: without con-currency each application has to be run to completion before the next one can be run.
④ Better performance: Con-currency provide better performance by the OS when one application use only the processor and another application use only the disk drive then the time to concurrenily run both applications to completion will be shorter than the time to run each application consequitively.

DATE : __/__/__
PAGE : _____

Critical Section (region) problem.

A critical section is a segment of a code which can be accessed by a single process at a specific point of time. In critical section, only single process can be executed other process, waiting to execute their critical section, need to wait until the current process completes its execution.

A soln to a critical section problem must satisfy the following three condition.

① Mutual exclusion.

out of a group of co-operating processes, only one process can be in its critical section at a given point of time.

② Progress.

This soln is used when no one is in the critical section and someone wants in. Then those processes not in their remainder section should decide who should go in a finite time.

③ Bound Waiting.

When a process makes a requires for getting into the critical section, there is a specific limit about no of processes can get into the critical section. So when the limit is reached, system must allow request to the process to gets into its critical section.

Logical conditions to implement Critical region.

for current critical region, implementation the following four condition are required.

DATE : __/__/__
PAGE :

1. No two processes may be simultaneously inside their critical region.
2. No assumption may be made about speeds or the no. of cpu's.
3. No process running outside the critical region may block other processes.
4. No process should have to wait forever to enters its critical region.

Mutual exclusion.

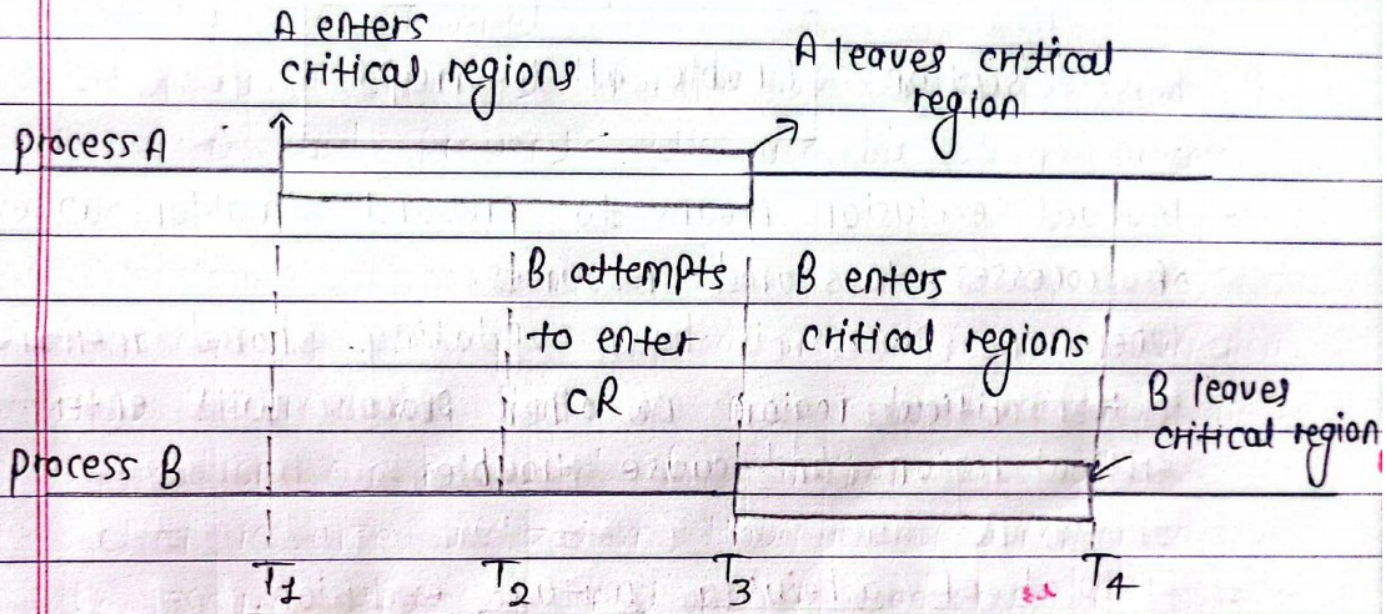


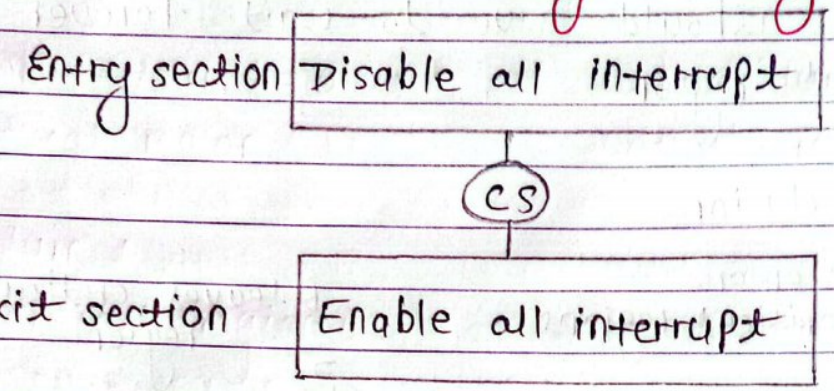
Fig Mutual exclusion using critical region.

Mutual exclusion is program object that prevents simultaneously access to a shared resource. This concept is used in concurrent programming with a critical section. A piece of code in which processes or threads access a shared resources

Here process A enters its critical region at a time T_1. A little later at a time T_2 process be attempts to enter its critical region but fails because another process is already in critical region and we allow only one at a time consequently, B temporarily

until time T_3. When A leaves the critical region, allow B to enter immediately. Eventually B leaves at T_4 and we are back original situation with no processes in their critical region.

Mutual exclusion with busy waiting.



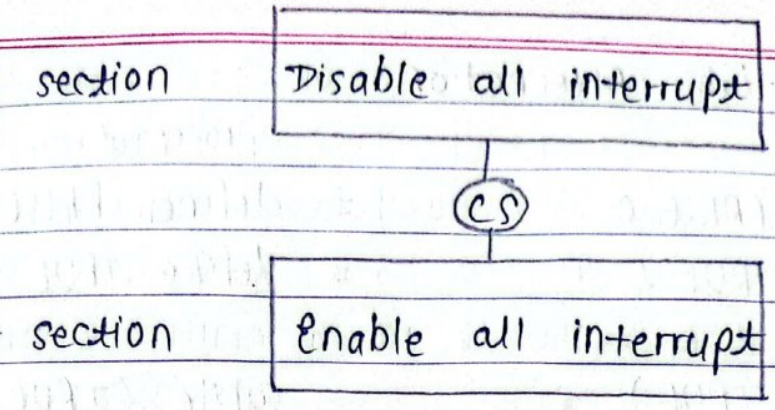
- Mutual exclusion means to prevant simultaneous excess of processes to shared resources.
- When one process is busy updating shared memory in its critical region no other process will enter its critical region and cause trouble.

Proposal of acheiving mutual exclusion.

① Disabling interrupt

- It is hardware type soln which runs in kernel mode.
- An interrupt is an event that allow the sequence in which the processor execute instruction.
- On a single processor system, the simplest soln is to have each process disable all interrupt just after entering its critical regions and re-enable them just before leaving it.

DATE : __/__/__
PAGE : _____



Problem: Suppose that one of the process disable interrupt and never turn on again then this could be end of the system.

② Lock Variable:

- It is software solution.
- Make the use of single shared (lock) variable, initially zero (0).
- Before entrying into a critical section, a process sets a shared variable locks value.
- If the value of lock is 0 then set it to 1 before entrying the critical section and enters into the critical section and set it to 0 immediately after leaving CS.
- If the value of lock is 1 then wait until it becomes 0 by some other process which is in critical section.

Problem: suppose that one process reads the lock and see that it is 0 before it can set the lock to 1, another process is scheduled, runs and sets the lock to 1. When the first process runs again, it will also set the lock to 1 and two processes will be their critical region at the same time.

DATE : __/__/__
PAGE : _____

③ Strict Variable Alternation.

process 0

```
# define FALSE 0
# define TRUE 1

while (TRUE)
{
  while (turn != 0);
  critical_region ();

  turn 1;
  non critical_region ();
}
```

process 1

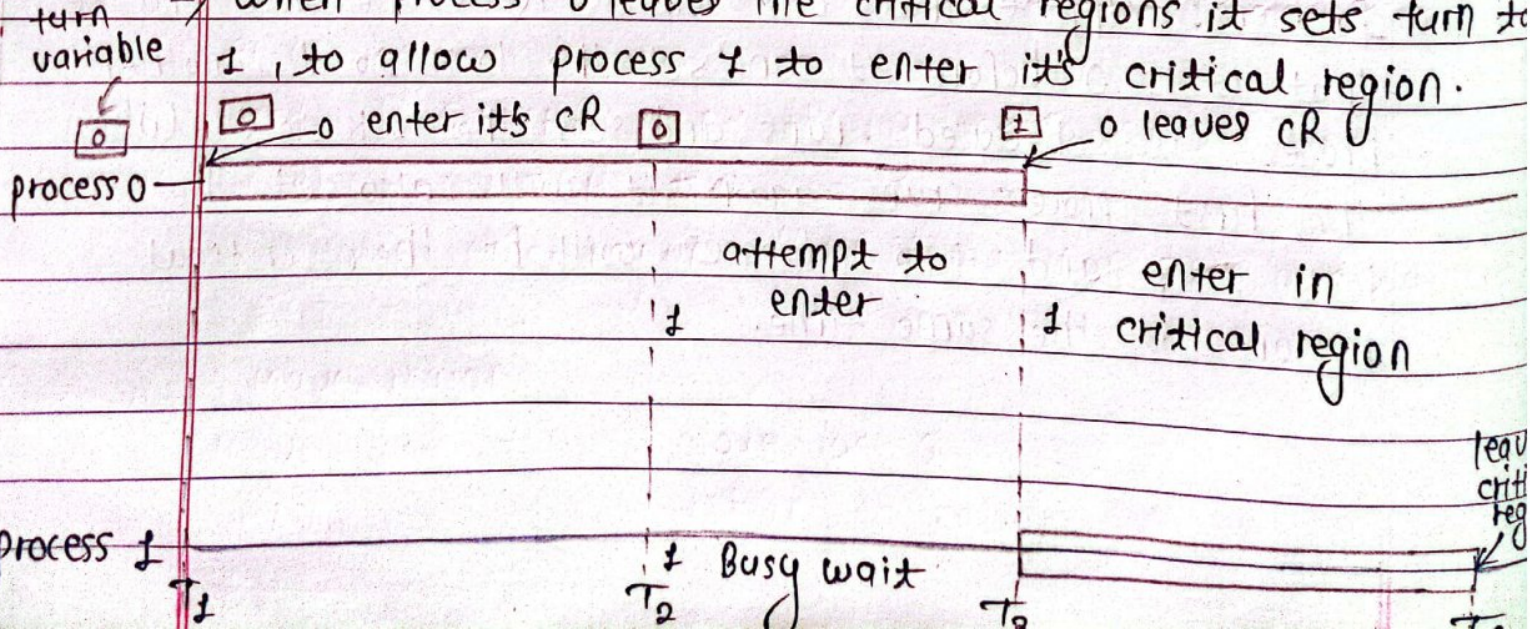
```
# define FALSE 0
# define TRUE 1

while (TRUE)
{
  while (turn != 1);
  critical_region ();

  turn 0;
  non critical_region ();
}
```

⇒ Two process solution.

⇒ The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

⇒ Initially, process 0 inspect turn, find it to be 0 and enters its critical region. process 1 also find it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.

⇒ When process 0 leaves the critical regions it sets turn to 1, to allow process 1 to enter its critical region.



DATE : __/__/__
PAGE :

problem : Suppose that process 1 finishes its critical region quickly, so that both process are their non-critical regions, with turn set to 0. Now process 0 execute its whole loop quickly, existing its critical region and setting turn to 1. At this point turn is 1 and both process are executing in their non-critical region.

④ Petersons solution.

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

   int turn;
int interested[N] = FALSE
void enter_region (int process)
 {
   int other;
other = 1 - process;
interest[process] = TRUE;
   turn = process;
while (turn == process && interested[other] == TRUE;
}
  void leave_region (int process)
  {
    interested[process] = FALSE;
  }
```

→ Before using the shared variables (ie before entering its critical region), each process calls enter_region with its own process number 0 or 1 as parameters.

DATE :__/__/__
PAGE :_____

⇒ This call will cause it to wait, if need be until it is safe to enter. After it has finished the shared variables, the process call leave_region to indicate that it is done and to allow the other process to enter, if it is so desire.

⇒ Initially neither process is in its critical region Now process 0 calls enter_region. if it indicates its interest by setting its array element and sets turn to 0.

⇒ Since process 1 is not interested ; enter_region return immediately. if process 1 now makes a call to enter_region. It will hang their until interested [0] goes to false and event that only happens when process 0 calls leave_region to exit the critical region.

⑤ Test and set lock instruction (TSL)

⇒ Test and set lock (TSL) is a synchronization mechanism

⇒ It uses a set and test instruction to provide the synchronization among the processes executing concurrent

```
while (test set lock );
        CS
  lock 0;

enter_region
TSL REGISTER , LOCK // copy lock to register and set lock
CMP REGISTER , #0
   JNE enter region // If it was non-zero, lock was set.
RET
leave_region
move lock , #0 // store lock 0
RET              // return to caller.
```

DATE : __/__/__
PAGE :

⇒ To use the TSL instruction, we will use the shared variable, lock to coordinate access to shared memory.
⇒ When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory.
⇒ When it is done, the process sets lock back to 0 using an ordinary move instruction.

Sleep and wakeup.

⇒ For mutual exclusion, one of the simplest is the pair sleep and wake up.
⇒ Sleep is a system call that causes the caller to block ie be suspended until another process wakes it up.
⇒ Wake up is a system call that wakeup the process.
⇒ The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wake up, each have one parameter, a memory address used to match up sleeps with wakeup.

Semaphores.

⇒ Semaphores is a simply a variable. This variable used to solve critical section problem and to acheive the process synchronization in the multi-processing environment.
⇒ It is a technique to manage the concurrent processes by using a simple integer value known as semaphores.
⇒ The two most common semaphore are counting semaphores and binary semaphore.
⇒ Counting semaphores can take $-\infty$ to ∞ integer values and binary semaphores can take the value 0 and 1 only.

DATE :__/__/__
PAGE :_____

→ A semaphore can only be accessed using the following operation.

Wait: The wait operation decrement the value of its argument s if it is positive, if s is negative or 0 then no operation is perform.

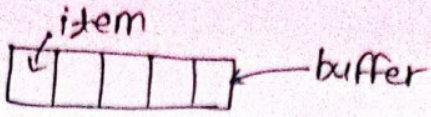
```
wait (semaphore s);
{
   while (s<=0);
      s--;
}
```

Signal: The signal operation increments the value of its argument s.

```
signal (semaphore s);
{
   s++;
}
```

Producer and consumer problem solving

The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that. well, the problem arises in the case when the consumer got preempted just before it was about to sleep Now the consumer is neither sleeping nor consuming.



DATE : / /
PAGE :

```
producer()
{
  int item;
while (TRUE)
 {
  item = producer_item();
 if(count == N)
   sleep();
insert_item(item);
   count++;
if (count==1)
 wakeup(consumer);
}
   }
```

```
consumer()
{
  int item;
 while (TRUE)
 {
 if (count==0)
   sleep();
item = remove_item();
   count--;
if(count==N-1)
 wakeup(producer);
consumer_item();
}
  }
```

Advantage of semaphore.

- Semaphores donot allow multiple process to enter in the critical section.
- They allow more than one thread to access the critical section.
- No wastage of resource in semaphores because of busy waiting in semaphores as procesor time is not wasted unnecessarily to check if any condition is fullfilled in order to allow a process to access the critical section.

Disadvantage of semaphore.

- Semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the CS later.

DATE : __/__/__
PAGE :

- Wait and signal operation are required to be executed in the correct order.

Mutex :

```
Mutex_lock
TSL REGISTER, MUTEX  // copy mutex to register and set it to 1
CMP REGISTER, #0     // was register 0
JZE OK               // If it was 0, mutex was unlock so return
CALL Thread_yield    // mutex is busy, schedule another Thread
JMP Mutex_lock       // Thread try again.
OK: RET              // return to caller; critical region ent
Mutex_unlock
Move mutex, #0       // store a 0 in mutex
RET                  // return to caller.
```

Mutex is the shortform for mutual exclusion object. A mutex is a shared variable that can be in one of two states unlocked and locked. Consequently only one bit is required to represent it, but in practice an integer often is used with zero meaning unlocked and other value meaning locked. Two procedures are used with mutex. When a Thread (or process) needs access to a critical region, it calls mutex_lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeed and the calling Thread is free to enter the critical region. On the other hand, if the mutex is already locked, the calling Thread is blocked until the thread in the critical region is finished and calls mutex_unlock. If multiple Threads are blocked on the mutex, one of them is choosen at

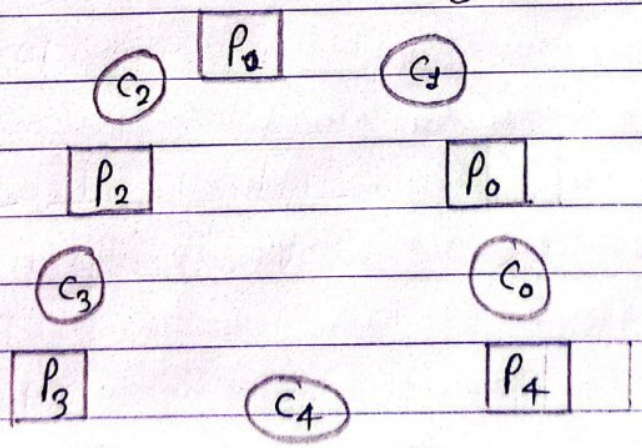
random and allow to acquire the lock.

The Dining philosophers problem.

The dining philosophers problem is the classical problem of synchronization which says that five philosophers are sitting around around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at center of a table along with five chopsticks for each philosophers. To eat, a philosopher needs both their right and a left chop-stick. A philosopher can only eat if both immediate left and right chopsticks of philosopher is available. In case of both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopsticks and start thinking again.

The dining philosopher demonstrates a large class of concurrency control problem hence it's a classic synchronization problem.

Lets ~~an~~ dining philosopher problem with code. Here, the five philosopher are represented as P_0, P_1, P_2, P_3, P_4 and five chopsticks by C_0, C_1, C_2, C_3 and C_4.



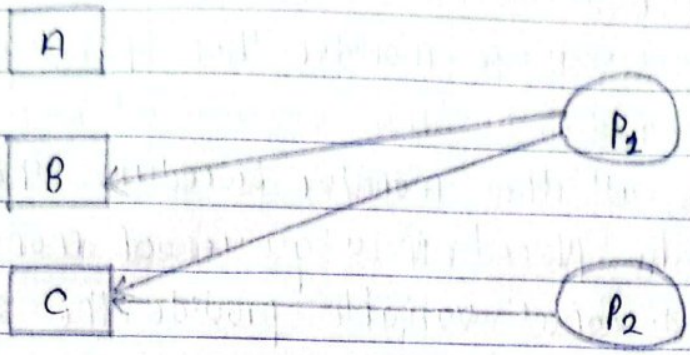
P_F Philosopher
C = chopsticks

```
Void philosopher
{
  While (1)
  {
    take_chopstick [i]
take_chopstick [(i+1] %05];

EATING THE NOODLES

put_chopstick [i];
put_chopstick [(i+1)%05];

THINKING
  }
}
```

Monitor

DATE : __/__/__
PAGE :

→ When multiple process access to shared data simultaneously it leads to the problem of Race condition.



⇒ Monitor is programming language construct that control access to shared data.

⇒ It is a collection of procedures, variables and data structure that are all grouped together in a special kind of module or package.

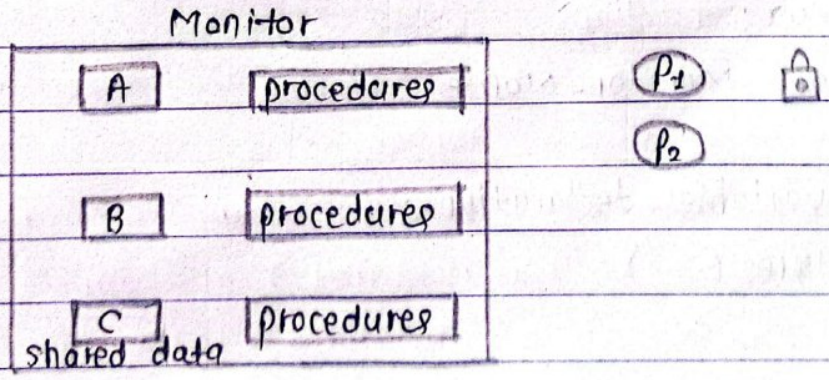
⇒ Monitor is a module that encapsulate.

- Shared data structure: procedures that operates on shared data structure and
- Synchronization betn concurrent procedures in vacations.

⇒ In monitor when a process wants to access shared data, they cannot access directly. process have to call procedure and those procedure in turn will allow access to share data.

⇒ only one process can enter into monitor at a time.

example



⇒ Here when process P_1 call the procedure in order to access shared data, at first it is checked if any other process is in monitor or not.

⇒ If no other process is in monitor then P_1 acquires lock and enters the monitor.

⇒ When process P_2 call the monitor procedure, then it's gets blocked and will be placed in a queue of monitor.

⇒ In monitor conditional variable provide the synchronization on both the processes.

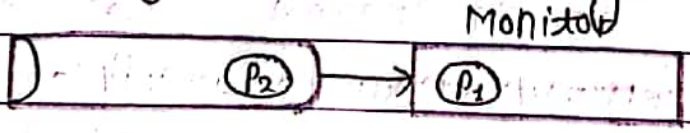
⇒ Three operation can be performed on conditional variable:
- wait()
- Signal()
- Broadcast()

⇒ wait(): When process call wait() operation, that process is placed on a queue to enter to the monitor.

⇒ Signal: When a process want to exit from monitor it calls signal() operation.

⇒ When a process call a signal() operation, it causes one of the waiting processes in the queue to enter monitor.

⇒ Broadcast: It signals all the waiting process in the queue to wait.



Monitor syntax:

```
Monitor Monitor_Name
{
  local_variable_declaration ;
  procedures (---)
  {
    // statements;
```

```
}
    procedure2(...)
    {
    // statements;
    }
    ----
}
```

Difference betn Monitor and semaphores.

Monitor	semaphore
⇒ We can use condition variables only in monitor	In semaphore, we can use condition variables anywhere in the program but we cannot use conditions variable in a semaphore.
⇒ In monitor, wait always block the caller	In semaphore, wait doesnot always block the caller.
⇒ The monitor are comprised of the shared variables and the procedures which operates the shared variable.	The semaphore's value means the no.of shared resources that are present in system.
⇒ Condition variable are present in the monitor	Condition variable are not present in the semaphore.

Sleeping Barber problem.

- Dijkstra introduced the sleeping barber problem in 1965. This problem is based on a hypothetical scenario where there is a barbershop with one barber. The barbershop is divided into two rooms the waiting room and the workroom. The waiting room has 'n' chairs for waiting customers and the workroom only has a barber chair

- Now if there is no customer then the barber sleep in his own chair (barber chair). whenever a customer arrives, he has to wakeup the barber to get his haircut. If there are multiple customers and the barber is cutting a customer's hair then the remaining customers wait in the waiting room with "n" chairs (if there are empty chairs) or they leave if there are no empty chairs in the waiting room.
- The sleeping barber problem may lead to a race condition. This problem has occurred because of actions of both barber and customer.

Example to explain the problem.

- Suppose a customer arrives and notices that the barber is busy cutting the hair of another customer, so he goes to waiting room. While he is on his way to waiting room the barber finishes his job and sees the waiting room for other customer. But (the barber) he finds noone in the waiting room (as the customer has yet not arrived in the waiting room), so he sits down his chairs (barber chair) and sleeps. Now the barber is waiting for new customers to wakeup him, and the customer is waiting as he think the barber is busy.
- Here, both of them are waiting for eachother, which leads to race conditions.

Solution of sleeping Barber problem:

The following soln uses three semaphores, one for customer (for counts of waiting for customer), one for barber (a binary semaphore denoting the state of the barber ie 0 for idle and 1 for busy) and a mutual exclusion.

DATE :__/__/__
PAGE :_____

Semaphore, mutex for seats:

```
Semaphore customer = 0; // semaphore for count of customer.
Semaphore Barber = 0; // Semaphore denoting the status of
                        barber 0 for idle and 1 for busy.
Mutex seats = 1;
int freeseats = N;
Barber
    {
      While (true)
      {
        // while waits for a customer (while barber is asleep)
              down (customer);
        // mutex semaphore to protect the no. of available seats
          down (seats);
        // a chair gets free
          freeseats ++;
        // this will bring customer for hair cut.
          up (Barber);
        // releasing the mutex (semaphores) on the chair.
          up (seats);
        // now barber is cutting the hairs.
      }
}
  customer
        {
          While (true)
          }
// protect seats so that only 1 customer tries to sit in a
chair if that's the case.
  down (seats);
```

```
// This line should not be there.
    if (freeseats > 0)
    {
        // sitting down
        freeseats--;
        // notifying the barber
        up(customer);
        // releasing the lock.
            up(seats);
        // customer wait in the waiting room if the barber is busy.
            down(Barber);
        // customer is having hair cut.
    }
    else
        { // releasing the lock
            up(seats);
        // now customer leaves
        }
    }
}
```

Analysis : When a barber arrives for work, the barber procedure is executed and he see it any customer is ready or not. If anyone is ready, pickup the customer for haircut and block the customers semaphore. If no one is ready for haircut, the barber sleeps.

If there is an available chair, the waiting counter is incremented