UNIT-II

Concurrent Processes

Process :-

Process is a program in execution which also include current activity as represented by the value of the program counter, content of processor's registers, process stack, temprary data and a data section containing global variables.

Principle of concurrency :-

On a single processor mic the os support for concurrency allows multiple applications to share resources in such a way that appln appear to run at the same time.

Type of processes :-

→ A process is independent if it can not affect or be affected by the other processes executing in the sys. (ie process does not share any data).

→ A process is co-operating if it can affect or be affected by the other process executing in the sys. (ie process sharing data).

Reason for allowing process co-operation —

→ Info sharing
→ Computation speed up (break task into subtask & do II execution)
→ Modularity
→ Convenience

Producer Consumer Problem

A producer process produces info that is consumed by a consumer process.

eg:- print prog. produces character & consumed by printer driver.

→ A buffer is required which will be filled by data produced by producer process & consumed by consumer process.

→ Both must be synchronized so that the consumer does not try to consume an item that has not yet been produced.

→ Unbounded buffer → No limit on size
Bounded buffer → fixed size buffer.

Shared m/m Sol^n to the bounded buffer Problem

variable :-

var n;
in, out; (0 to n-1)

in → points to next free position in buffer
out → " " first full " " "

in = out → Buffer is empty
in+1 mod n = out → Buffer is full.

```
Producer process (nextp a local variable)
{
Repeat
produce an item in nextp;
while in+1 mod n = out  do no_op^n;
buffer [in] := nextp;
in = in +1 mod n;
until false;
}

Consumer process (local var nextc)
{
Repeat
while in = out do no_op^n;
nextc = buffer [out];
out = out +1 mod n;
---
consume the item in nextc;
---
until false
}
```

→ A situation where processes access and manipulate the same data concurrently and the outcome of the execution depends on particular order in which the access takes place is called Race Condition.

Mutual Exclusion :-

If more than one process tries to access the variable at the same time, it will lead to inconsistant data. The problem can be solved by giving each process exclusive access to the shared variable.
Each process accessing the shared data excludes all others from doing so simultaneously, this is called Mutual exclusion.

Critical Section :-

When a process is accessing shared modifiable data, the process is said to be in a ~~con~~ critical section.
Thus, when one process is in critical section, all other process are excluded from their own CS but they may continue executing outside their CS.

There are three requirements that must stands for correct solution :-

* Mutual Exclusion
* Progress (Only that process compute which want to)
* Bound wait (logical bound that every process will get chance)

→ (If no process is in its CS, & if one or more threads wants to execute their CS then any one ~~of~~ of these must be allowed to get into its CS.)

Dekker's Solution

```
boolean flag[2]; // flag[1] is flag of P1 and flag[2] is of P2.
int turn;        // turn = 1 means favour is first process and
                 //        turn = 2 "  "  "  "  second  "
void P1()
{
    while(true){
        flag[1]=true;
        while(flag[2]){
            if(turn==2){
                flag[1]=false;
                while(turn==2);     // do nothing
                flag[1]=true;
            }
        }
        CS1;
        turn=2;
        flag[1]=false;
        otherstuff;
    }
}
void P2()
{
    while(true){
        flag[2]=true;
        while(flag[1]){
            if(turn==1){
                flag[2]=false
                while(turn==1);     // do nothing
                flag[2]=true;
            }
        }
        CS;
        turn=1;
        flag[2]=false;
        otherstuff;
    }
}
```

```
void main()
{
flag[1]=false;
flag[2]=false;
turn=1;
parbegin(P1,P2);
}
```

Case1 :- If flag[1] = T, flag[2] = F, skip outer loop
Process 1 in CS.

Case2 :- If Flag[1] = T, flag[2] = 2, if turn = 1 outer while execute repeatedly, untill P_2 make its flag off.

Case3 :- If flag[1] = T, flag[2] = T, if turn = 2, then make flag[1] = F and looping in inner while untill turn = 2. → (So p2 in CS)

Case4 :- When P_2 comes out from CS, it make turn = 1 | flag[2] = F
Now P_1 passes inner while and set flag[1] = T
and come to outer while
if flag[2] still false then P_1 in CS.
and if P_2 makes flag[2] = T and turn = 1 Then P_1 loops within outer while untill P_2 set flag[2] = F.

Case5 :- As P_1 comes out of inner busy loop, it loose the processor before executing flag[1] = T, at this time —
flag[1] = F, Flag[2] = T, turn = 1
P_2 will re-enter in its CS.

When P1 gets processor back it set flag[1] = T | turn = 1
Then if P_2 tries to entu CS then set
flag[2] = F and forced in inner busy wait.
So P_1 will entu its CS.
So this will not result in idefinite postponement.

[Limitation :- If P_1 again preempt to P_2 before P_2 set true P_1 ~~will~~ may execute again & again]

```
Program peterson_algo
int turn;
boolean flag[2];
procedure P1()          // first process
{
 while(true) {
        flag[1] = true;
        turn = 2;
            while(flag[2]=true and turn==2);
            CS;
            flag[1] = false;
            otherstuff;
    }
  }
procedure P2()          // second process.
{
  while(true){
        flag[2] = true;
        turn = 1;
          while(flag[1] and turn==1);
        CS;
        flag[2] = false;
        otherstuff;
    }
}
main()                  // main Program.
 {
  flag[1] = false;
  flag[2] = false;
  parbegin(P1, P2);
  }
```

Semaphores :- (S/w Soln)

It is a synchronization tool to genrilize critical section problem.

A semaphore S in an intiger variable that, apart from initialization, is accessed only through two standard atomic operations : wait and signal.

```
wait(S): while S≤0 do no-op";
         S := S-1;

Signal(S): S:=S+1;
```

→ Modifications to the intiger value of the semaphores in the wait and signal operations must be executed indivisibly. (when one process modifies the value no other process should modify it)

→ In case of wait(S), the testing of the intiger value of S (S≤0), & its modification, must also be executed without interruption.

Usage :-

We can use semaphores to deal with the n-procus critical section problem.

The n processes share a semaphore, mutex (mutual-exclusion), initialized to 1. Each process is organized as:

m = 0

$P_1 | P_2 - P_n$

```
repeat
    wait(mutex);
        → critical section
    Signal(mutex);
        remainder section
Until false;
```

Use of Semaphore to Solve various synchronization Problem

→ P1 process with statement S1.
→ P2 " " " S2;
→ common semaphore "synch" initialized to zero.
→ Now if we want to execute S2 after S1 has completed then code —

P1
S1;
signal (synch);

P2
wait (synch);
→ S2;

Now P2 will execute S2 only after P1 has invoked signal (synch) which is after S1.

Test and Set Operations

In uniprocessor environment, the problem of CS can be solved if we could disallow interrupts to occur while a shared variable is being modified. This soln is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the msg is passed to all the processors which causes delay & system efficiency decreases.
Many m/c use special H/W instructions to solve the CS problem.
Test and set instruction can be defined as:-

```
function Test_and_set (var target : boolean)
begin
    Test_and_set := target;
    target := true;
end;
```

Use of Semaphore to Solve various Synchronization Problem

This instruction is executed atomically, Thus if two test_and_set inst. are exe. simultaneously (on diff CPU), they will be executed sequentially.

Test: It tests the value of a m/m byte such that the condition code indicates whether the value was zero or non-zero.

Set: It sets all bits of m/m byte to 1.

Bounded-Waiting mutation exclusion with test-&-set

```
var j ; (0 to n-1)                    array waiting[0 to n-1];
boolean key;                          boolean ~~waiting~~ lock;
~~boo~~ boolean lock = false;                 (initialized to false)

repeat
    waiting[i] = true;
        key = true;
    while (waiting[i] and key)
        do key = test_and_set (&lock);
    waiting[i] = false;

    Critical section

    j = i+1 mod n
    while ((j ≠ i) and (not waiting[j]))
        j = j+1 mod n;
    if (j = i) then lock = false
    else waiting[j] = false;

    remaining section
until false;
```

* Target (in test & set) and lock (in algo) are same.
{ie. call by reference}

→ For ME ⇒ Process P_i can enter its cs only if either waiting[i] = false or key = false;

→ key is false only if Test-and-set is executed.
First process to execute test-and-set must find key = false, all other must wait.

→ For bound-wait – when a process leaves it cs, it scan the array waiting in cyclic order (i, i+1, i+2)
Thus any process will enter its cs within n-1 turns.

Binary Semaphore

< Down / P / wait
Up / V / signal

Binary Semaphore is also known as mutex lock.
It can have only two values 0 and 1.
Its value is initialized to 1.

Counting Semaphore :-

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instance.

In case of binary Semaphore we can write code of wait and signal as:—

```
wait (semaphore s)
{
    if (s.value == 1)
      s.value = 0;
    else
    { Block the process
      place in suspend
      list. (sleep);
    }
}
```

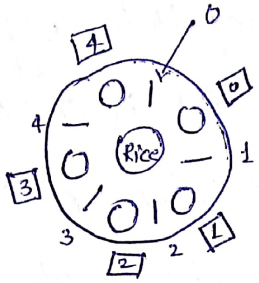
```
signal (semaphore s)
{
  if (suspend list is empty)
  {
    s.value = 1;
  }
  else
  {select a process from
    suspend list & wakeup();
  }
}
```

When P1 goes to sleep() then it was placed in suspended list. When it wakeup, it will not execute the whole wait operation again but will resume from the point of sleep() part and directly enters in CS.

Classical Problems on Synchronization

① The Dining Philosophers Problems

Consider five philosophers who spend their lives thinking and eating. They share common circular table surrounded by five chairs, each belongs to one philosopher.



(situation of dinining philosopher)

In centu of the table there is a bowl of rice and the table is laid with five single chopsticks.

Then the philosopher thinks, he does not interact with his calleagues. From time to time, a philosopher gets hungry and treis to pick up two chopsticks that are closest to him (from left & right). A philosopher pick only one chopstick at a time. When hungry philosophus has both the chopsticks at the same time he eats without releasing his chopsticks. Whe he finished eating, he put down both of his chopstick and start thinking again.

This problem is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

Solution :-

Represent each chopstick by a semaphore.

→ variable chopstick is array[0..4] of semaphore. all initialized with one.

→ Philosopher graps chopstick by executing wait opⁿ and release by executing signal opⁿ on appropriate semaphore.

Structure of philosopher i

```
Repeat
    wait (chopstick[i]);
    wait (chopstick[i+1 mod 5]);
    ...
    Eat;
    ...
    signal (chopstick[i]);
    signal (chopstick[i+1 mod 5]);
    ...
    Think;
Until false;
```

→ It guaranties that no two neighbours are eating simultaneously but it has possibility of deadlock if all five become hungry.

Possible Remedies to deadlock :—

→ Allow atmost four philosopher to be sitting simultaneously at the table.

→ Allow a philosopher to pick up chopsticks only if both are free.

→ Use asymmetric soln ie :-
An odd philosopher pick first his left chopstick & then his Right chopstick.
Whore as even philosopher pick up his right chopstick & then the left one

---

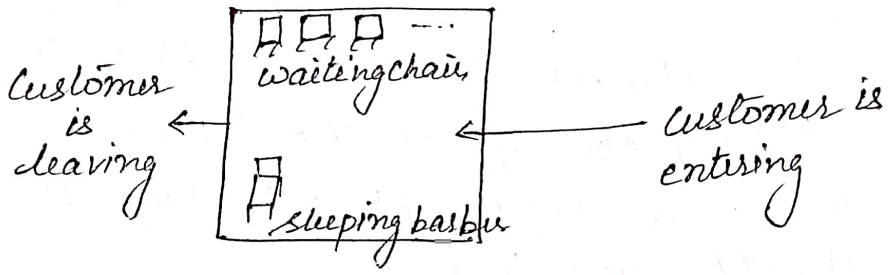
(2) Sleeping Barber Problem

Problem: 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. ~~customers if there are no~~

→ If there is no customer, then barber sleep 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 chair, then the remaining customers either wait if there are empty chair in the waiting room or leave the room if no chair is empty.



Solution :- The solution to this problem includes three semaphores —

① customer - to count no of customer in waiting room (not on barber seat).

② Barber — Used to tell whether the barber is idle or is working (0 or 1).

③ Mutex — Used to provide the mutual exclusion

Need Of Mutex (Because of Scheduling Problem)

Understand with example —

(i) A customer may arrive and observe that barber is cutting hair, so he goes to the waiting room. While he is on his way, barber finishes cutting and goes to check the waiting room. Since there is noone in the room, he goes back to his chair and sleeps.

Now barber is waiting for customer and the customer is waiting for the barber.

(ii) Two customers may arrive at the same time, observe that the barber is cutting hair and their is a single seat in waiting room, then both will attempt to occupy the single chair.

Solution :-

[Deadlock free but could lead to starvation]

```
semaphore customer = 0          // count waiting customer
semaphore barber   = 0          // check status of barber
semaphore accessseat (mutex) = 1
int no_of_free_seat = n

Barber()
 {
  wait (customer);
  wait (access seat);
  no_of_free_seat ++;
  signal (Barber);
  signal (accessseat);
 }

customer ()
 {
  wait (accessseat)
  if (no_of_freeseat > 0)
    {no_of_freeseat --;
     signal (customer);
     signal (accessseat);
```

```
    wait(barber);
  }
else
    signal(accessseat);
}
```

Readers - Writer Problem

There is a shared resource which should be accessed by multiple processes. There are two type of processes —

Writer & reader

Any no of reader can read from resource simultaneously but only one writer can write. When writer is writing data, no other process can access. A writer can not write if there are non zero no of readers to accessing the resource at that time.

Solution :-

Write Process

```
while (T)
{
wait (w);
Write opn;
Signal (w);
}
```

Reader opn

```
while (T)
{
wait (mutex);
readcount++;
if (readcount == 1)
wait (w);
Signal (mutex);
Read opn;
wait (mutex)
readcount--;
if (readcount == 0)
signal (w);
signal (mutex);
}
```