Unit-3

Q) Define critical section problem and 3 conditions to be satisfied by solutions to critical section problem

Consider a system consisting of n processes {$P_0, P_1, P_2 \ldots P_{n-1}$} each process has a segment of code called critical section, in which processes may change common variables, tables, files so that their values will not become inconsistent.

only one process can execute code in critical section. To enter critical section, process requests permission in entry section then enters critical section. After coming out of critical section process executes exit section

```
do {
    Entry section
    Critical section
    exit section
    Remainder section
} while (true);
```

Structure of process P_i

Solution to critical section problem should satisfy the following 3 requirements

1) **mutual exclusion**: If a process is in critical section then no other process can be in critical section.

Scanned by CamScanner

(2) Progress : A process that have just completed Critical section and are in remainder section cannot again enter Critical section.

(3) Bounded waiting : There is After a process P_i has requested to enter Critical section there is a limit on number of times other processes can enter critical section before giving chance to P_i.

Q2 Two process solution for Critical section problem or Petersons ~~problem~~ solution

Petersons solution provides solution to critical section Problem works only for two processes. Here we use two data items.

Two processes are P_0 & P_1

```
int turn;
boolean flag[2];
```

if turn = 0 P_0 is allowed to enter Critical section if flag[0] = true

turn = 1 P_1 is allowed to enter Critical section if flag[1] = true.

if flag[0] = true = P_0 wants to enter CS
flag[1] = true = P_1 wants to enter CS

Mutual exclusion : Now if both P_0 P_1 wants to enter Critical sec at same time one sets turn to 0 then immediately P_1 will set turn to 1 finally P_1 will enter CS. ∴ mutual exclusion is preserved i.e either P_0 or P_1 only will be in critical section.

Scanned by CamScanner

(1) Structure of Process P_i in peterson's solution

```
do
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)      entry section
        wait;

    critical section

    flag[i] = false;                  exit section

    remainder section;
} while (true);
```

Progress: A process after exiting from Critical section changes its flag value to false thereby allowing waiting process to enter into critical section.

bounded waiting now if P_0 come of CS and sets its flag[0] = false, now P_1 which has its flag[1] = true and turn = 1 will enter critical section. ∴ after P_0 enters CS, next P_1 will get chance to enter CS if it wants.

Note: locks are used as solution to Critical section problem. Below we have code for it.

```
do
{
    acquire lock
        Critical section
    release lock
        remainder section
}while(true);
```

To enter into Critical section we should acquire lock and after exiting from critical section we should release locks so that next waiting process can enter into Critical section.

Scanned by CamScanner

Q3 Explain Process Synchronization Hardware (For multiple processes)
mutual exclusion can be achieved in two ways → a) using Testandset Instruction
b) using swap instruction

Q) Achieving mutual exclusion using Testandset

Code for Testandset instruction.

```
boolean Testandset (boolean &lock)
{
    boolean returnvalue;
    returnvalue = lock;
    lock = true;
    return (returnvalue);
}
```

→ Test & set is atomic instruction. If any process starts this function it should be executed completely. If two processes will execute Testset at same time they will execute one after other only.

Code to achieve mutual exclusion using Test & set —

```
do {
    while(Testandset(lock));      entry section
        Critical section
    lock = false;                 exit section
        remainder section
} while(true);
```

First lock will be initialized to ~~true~~ false;

lock = false;

Above code is for any process Pi.

Above solution to critical section problem satisfied mutual exclusion. (i.e only one process will be in critical section at a time) but doesn't satisfy bounded wait i.e a process which has come just now out of critical section will again compete to enter Critical section as lock is false.

Scanned by CamScanner

(b) Achieving mutual exclusion using Swap Instruction

```
void swap (boolean &a, boolean &b)
{ boolean temp;
  temp=a;
  a=b;
  b=temp;
}
```

Swap Instruction is atomic. If any process starts executing it they should complete it before another process ~~executes~~ starts execution of swap again. First lock = false.

mutual exclusion implementation using SwapInstruction

```
do {
    key=true;
    while (key == true)
        swap (lock, key);

        Critical section

    lock = false

        Remainder section
} while (true);
```

Entry section

The above solution to Critical section Problem satisfies mutual exclusion but doesnot satisfy Bounded wait condition.

(3) The Below code satisfies both mutual exclusion condition & bounded wait condition for Critical section problem using Test and set Instruction

boolean waiting[n]; } These are global
boolean lock; } variables initialized
waiting[0] = false ..... waiting[n-1] = false } to false initially.
lock = false;

For a process to enter into Critical section either waiting(i) should be false or key should be false.

```
do
{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = Testandset(lock);
    waiting[i] = false;

    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
} while (true);
```

Scanned by CamScanner

The above solution satisfies mutual exclusion i.e only one process will be in critical section at a time.

The above solution also satisfies bounded wait- because after i next process i only will get chance to enter critical section $j = (i+1)\%n$.

If next process of i is not ready to enter Critical section then next process of j will get chance to enter CS.

If (!waiting[j])

$j = (j+1)\%n$ satisfied.

Progress and bounded wait conditions are also satisfied. After all processes 0 to n-1 completed entering CS Process 0 will again get chance to enter CS.

Initially all waiting[i] = false & lock = false

If Process i wants to enter critical section then it changes it's waiting[i] = true.

Scanned by CamScanner

Q4 Explain about Semaphores, usage and Implementation (4)

Semaphore is an integer variable used to for process synchronization to as a solution to Critical section problem.

Semaphore can be accessed only through two atomic operations wait, signal.

Code for wait(t)

```
wait(s)        /* S is semaphore */
{ while(s<=0)
      ; /* wait or no operation */
  s--;
}
```

Code for signal

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

once wait & signal r being executed by an process they should be completed fully. No other process can enter until they come out of wait & signal.

Semaphores usage :- They are used as solution to n-process critical section problem.

n processes share only one semaphore.

Example:
Suppose we have process P_1 with one stmt S_1 and process P_2 with one stmt S_2. I want S_2 to be executed only after S_1 then I take a semaphore with name

Scanned by CamScanner

Synch and initialize it with 0 and write below code

For process P1 :- Synch = 0;

```
Process P1
{
  S1;
  signal(Synch);
}
```

```
Process P2
{
  wait(Synch);
  S2;
}
```

First Synch = 0 Now if P_2 wants to execute before P_1 it executes wait(synch) ∴ Synch == 0 in wait operation it enters infinity loop

```
while(S<=0)
  ; /* wait */
```

so only if P_1 executes first and changes S from 0 to 1 by executing signal() operation doing S++ P_2 cannot move to start S_2.

example 1 over

Example 2

Suppose our semaphore name is mutex and it's initialized to value 1 and code for process P_i is given below to achieve mutual exclusion

```
do
{
  wait(mutex);
  Critical section;
  signal(mutex);
  remainder section;
} while(true);
```

Scanned by CamScanner

Implementation of semaphores :- (5)

Problem with some semaphores is when one process is in Critical section if other process wants to enter Critical section it will continuously execute code in entry section indefinitely until it gets a chance to enter into Critical section. This wastes lot of time of CPU i.e it makes CPU busy waiting. This type of semaphores are called spinlocks. But advantage of spinlocks are they require no context switching b/w processes.

Below we define semaphore in such a way that when one process is in Critical section, if other processes want to enter Critical section they will be blocked and all of them are stored in a queue in waiting state. When process i.e already in Critical section comes out of it then one of the processes in waiting state in queue will be changed to ready state & executed using wait & signal operation. But disadv of this process is they require context switching.

Definition of semaphore

```
typedef struct {
        int value;          /* value of semaphore */
        struct process *L;  /* List of processes waiting on this semaphore */
} semaphore;

semaphore S;
```

S.value => Value of semaphore (if it is -ve i.e no. of Processes waiting on this semaphore)

S.L => names of all processes waiting on this semaphore.

Scanned by CamScanner

The wait (Semaphore) Code

```
Void wait (Semaphore S)
{
    S.value--;
    if (S.value < 0)
    {
        add this process to S.L;
        block(); /* block this process to wait state */
    }
}
```

The signal (Semaphore) Code

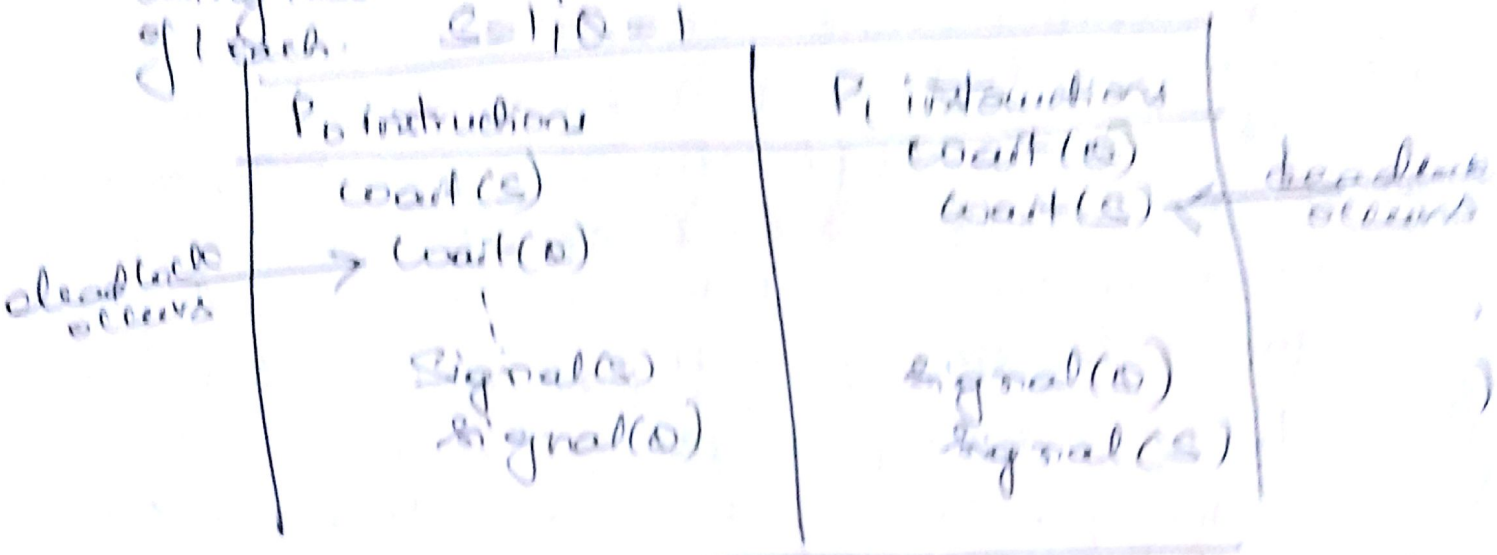
```
Void signal (Semaphore S)
{
    S.value++;
    if (S.value <= 0)
    {
        remove a process from S.L;
        /* or move process from waiting queue */
        wakeup(P); /* restart blocked process from queue */
    }
}
```

The above semaphore S is Counting semaphore i.e it has values 0 to n. Binary semaphore has only two values 0 and 1.

Scanned by CamScanner

Deadlocks & starvation problems in semaphores:

Deadlocks occurs in semaphores also, consider below example: P_0 & P_1 are two processes each using two semaphores S and Q with initial values of 1 each. S=1, Q=1



In above examples P_0 ~~will~~ not executes wait(S) P_1 executes wait(Q) now no deadlock is there. But when P_0 executes wait(Q) P_0 is blocked until signal(Q) will be executed by P_1, but P_1 cannot goto signal(Q) instruction as it by itself is blocked by executing wait(S) instruction.

Set of processes are in deadlock state if every process in set is waiting for an event to be caused by another process, which is also in same state.

A process is starving or indefinitely blocked if it's waiting for a long period of time in entry section of semaphore.

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

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

Scanned by CamScanner

Priority Inversion of processes

⇒ L, M, & H are 3 processes with low, middle and high priority. Now if L is running & resources used by L are now needed by H for its execution, what to do now? Generally ∵ M and H are of high priority than L ~~the~~ next M may get chance to execute ~~after~~ even if we preempt L from its execution. ∴ what we do is we allow L to finish execution by giving L priority of H person. After L finishes H gains back its priority & L again gets least priority. So out of L, M, H, H will be executed next but not M ∵ H has higher priority.

Scanned by CamScanner

Q5 Classical problems of synchronization.

a) Bounded buffer problem
b) Readers-writers problem
c) Dining philosophers problem

a) Bounded buffer problem.

Suppose we have n buffers, each buffer can hold one item. → mutex is used to provide mutual exclusion (i.e it says that either producer is items or consumer is items it in critical section)

→ empty semaphore maintains count of empty buffers (i.e) no empty buffers are there producer should wait

→ Full semaphore maintains count of number of full buffers (i.e) no buffer is full consumer should wait

Initialization: mutex = 1, empty = n, full = 0;

Structure of Producer process

```
do
{
    // produce an item in nextp
    wait(empty);   } entry section
    wait(mutex);
    . . .
    // add nextp to buffer   critical section
    . . .
    signal(mutex);   } exit section
    signal(full);
} while (true);
```

Q5 Classical problems of synchronization.

a) Bounded buffer problem
b) Readers-writers problem
c) Dining philosophers problem

a) Bounded buffer problem.

Suppose we have n buffers, each buffer can hold one item. → mutex is used to provide mutual exclusion (i.e it sees that either producer of items or consumer of items is in critical section)

→ empty semaphore maintains count of empty buffers (if no empty buffers are there producer should wait)

→ Full semaphore maintains count of number of full buffers (if no buffer is full consumer should wait)

Initialization → mutex = 1, empty = n, full = 0;

Structure of Producer process

```
do
{
    // produce an item in nextp
    wait(empty);
    wait(mutex);
    . . .
    // add nextp to buffer
    . . .
    signal(mutex);
    signal(full);
} while(true);
```

entry section

critical section

exit section

Structure of Consumer process.

```
do
{
    wait(full);
    wait(mutex);
        .  .  .
    // remove an item from buffer to nextc
    signal(mutex);
    signal(empty);
        .  .  .
    // consume item in nextc
} while(true);
```

entry section.

Critical section

exit section

buffers accessing is done in critical section only.

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

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

Here mutex initialized to 1

empty = n [∵ initially all buffers are empty]

full = 0 [initially no buffer is full].

Scanned by CamScanner

(b) Readers-writers problem

Two readers can access the shared data simultaneously, even then no problem occurs. But if one reads data at the time other is writing its value (updation) then reader may get unupdated value. Similarly two writers cannot change data at same time simultaneously. Therefore writers should have exclusive access to shared data when writing to database. This is called readers-writers problem.

First readers-writers problem: It requires that no reader be kept waiting unless a writer has already obtained permission to use shared object. i.e when one process is reading other process can also read unless a process has already obtained permission to write to that object.

Second reader-writers problem: If a writer is waiting to access object no new reader can start reading.

Solution to First reader-writers problem.

```
semaphore mutex, wrt;
int readCount.

mutex = 1; wrt = 1;
readCount = 0;
```

} Initializations.

wrt is semaphore is common to both reader & writer.
Mutex semaphore gives mutual exclusion to use readCount (only one process can change readCount at a time)

Scanned by CamScanner

wrt is ~~used as~~ semaphore to provide mutual exclusion for writers. wrt is used by first and last readers that enters & exits critical section.

If a writer is in critical section and n readers are waiting to enter into CS then one reader waits on wrt and (n-1) readers are queued on mutex.

Structure of ~~reader~~ writer process

```
do {
    wait(wrt);          entry section
        // writing is performed
    signal(wrt)         exit section
} while(true);
```

Structure of writer process

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
```

entry section

reading is done

```
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
} while(true);
```

exit section.

To see that readcount is used only by one process at a time.

Scanned by CamScanner

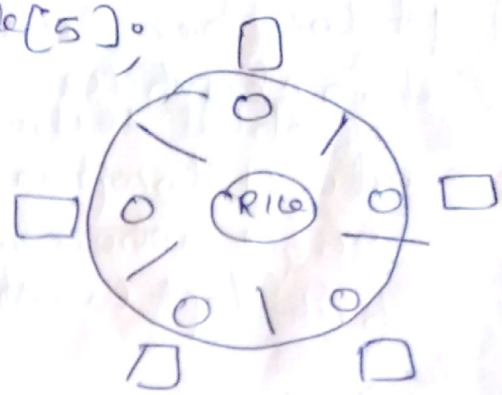
(c) The Dining-philosophers problem

In Dining philosophers problem coe have circular table soorounded by five chairs & bowl of rice in middle of table. Each philosopher has a plate, between ~~to~~ two philosophers coe have a chop stick. To eat each philosopher should pick chopsticks on left & right.

We ~~It~~ use five semaphore for solution to this problem

Semaphore chopstick[5];

Initialization
- chopstick[0] = 1
- chopstick[1] = 1
- chopstick[4] = 1



The below solution guaranteer that no two neighbours eat simultaneously.

```
do
{
    wait(chopstick[i]);                    entry section
    wait(chopstick[(i+1)%5]);

    eat

    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);            exit section

    // think
} while(true);
```

Scanned by CamScanner

But above solution causes deadlock if all 5 philosophers try to eat at sametime & execute above code & gets first chopstick each but indefinitely wait for second chopstick which other philosopher doesnot give as he is also waiting for second chopstick

Possible remedies to deadlock problem are

1) Allow atmost only 4 philosophers to eat at a time
2) philosopher can pickup chopsticks only if he gets two chopsticks. (So chopsticks should be picked up in critical section).
3. odd philosopher picks up left chopstick first & then right where as even philosophor picks right first & left chopstick second.

Scanned by CamScanner

Q) Explain about monitors. Give syntax of monitor (10)

A monitor provides solution to critical section problem.

A monitor has set of shared variables, local variables, operations. The operations (or Procedures/functions) operate on variables to provide mutual exclusion to Processes in the monitor.

→ only one process can be active out of all the processes in the monitor.

→ we define extra variables called condition Constructs or variables to provide mutual exclusion.

→ Condition x, y;

→ Two operations on Condition variables are

x.wait();

process who executed this operation should wait until already started process stops.

x.signal();

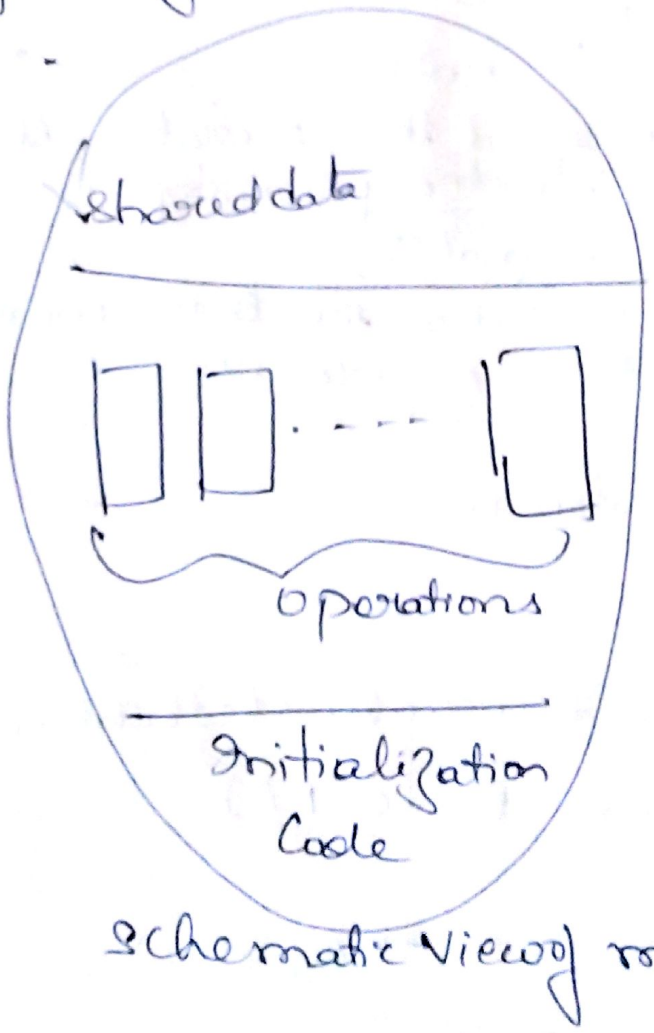
The above operation starts on waiting processes of all suspended processes.

```
monitor monitorname
{
  // shared Variable declarations
  Procedure Process1()
  {
  }
  Procedure Process2()
  {
  }
```

Scanned by CamScanner

```
:
Procedure Process()
{
  ...
}
Initialization code(...){
  {
  ...
  }
}
```

Syntax of monitor



Schematic View of monitor.

Scanned by CamScanner

Signal & wait : P waits until Q leaves the monitor or waits for another condition

Signal and continue : Q waits until P leaves the monitor or waits for another condition.

Scanned by CamScanner

Dining philosophers problem solution using monitors

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  Condition self[5];

  void pickup(int i)
  { state[i] = hungry;
    test(i);
    if (state[i] != eating)
        self[i].wait();
  }

  void putdown(int i)
  { state[i] = thinking;
    test((i+4)%5);
    test((i+1)%5);
  }

  void test(int i)
  {
    if ((state[(i+4)%5] != eating) and
        (state[(i+1)%5] != eating) and
        (state[i] == eating))
    {
      state[i] = eating;
      self[i].signal();
    }
  }
```

Scanned by CamScanner

```
initializationCode()
{
    for(int i=0; i<5; i++)
        state[i] = thinking;
}
```

```
DiningPhilosopher.pickup(i)
    eat
DiningPhilosophers.putdown(i);
```

Here mutual exclusion is preserved, Progress bounded wait are also satisfied. Deadlock will not occur.

Each philosopher executes above code, when we call pickup(0) → we are calling monitor dp pickup function for philosopher 0. pickup() function changes state of philosopher 0 to hungry and see if philosopher 1 and philosopher 4 neighbours of philosopher 0 are not eating. Because if 4 and 1 philosophers are eating philosopher 0 cannot get chopsticks to eat.

After eating philosopher calls putdown function, in which philosopher 0 tests philosopher 1 and 4 to call test function & start eating if 2 chopsticks are available. Here deadlock will not occur as no two neighbour Philosophers eat at same time.