Page No.
Date

Q1. Semaphore S is an integer variable that, apart from initialisation is accessed only through 2 standard atomic operations wait() & signal().

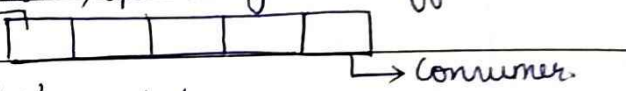
Definition of wait():
```
P(Semaphore S){
  while(S<=0)
    ; //no operatn
    S--;
}
```

Definition of signal():
```
V(Semaphore S){
    S++;
}
```

Q2. Classical Problems of Synchronisation:

① Bounded-Buffer Problem / Producer-Consumer Problem:

- Buffer of n slots, each slot capable of storing 1 unit of data.
- 2 Processes running: Producer & Consumer, operating on buffer.



i) m(mutex): semaphore initialised to '1', used to acquire and release the lock.

ii) empty: semaphore initialised to 'n' which is no. of slots in buffer

iii) full: semaphore initialised to '0'.

- Producer:

```
do {
  wait(empty); //wait unit empty >0. & then decrement empty
  wait(mutex); //acquire lock.

  /* add data to buffer */
  signal(mutex); //release lock.
  signal(full); //increment 'full'
} while(TRUE)
```

- Consumer:

```
do {
    wait(full); // wait until full > 0 & then decrement full.
    wait(mutex); // acquire lock.

    /* remove data from buffer */
    signal(mutex);   // release lock.
    signal(empty);   // increment 'empty'
} while (TRUE)
```

② Readers-Writers Problem (done in Assignment)

③ Dining Philosopher's Problem:

① Solution using Semaphore:

all elements of chopstick are initialized to 1

```
do {
 wait(chopstick[i]);
 wait(chopstick[(i+1)%5]);
 . . .
 //eat
 signal(chopstick[i]);
 signal(chopstick[(i+1)%5]);
 //think
} while (TRUE);
```

- It could create deadlock.
- If all 5 get hungry, all pick left chopstick ∴ chopstick = 0.
- When each go to grab right chopstick, delayed forever.

Remedies to avoid Deadlock:

- Allow atmost 4 to sit simult. at table.
- Allow one to pick chopstick only if both availab. (pick in Critical Sol)
- Use asymmetric sol^n. i.e. odd one picks up his left & then right. & even one picks up his right & left.

Page No.
Date

② Solution using Monitors : (deadlock free)

- new DS : enum { thinking, eating, hungry } state [5] ;
  to distinguish 3 states.
- philoso. i can set variable state [i] = eating only if 2 neigh are not eating : (state [(i+4) % 5] != eating) & (state [(i+1) % 5] != eating)
- we need to declare condition self [5]; where philo i can delay himself when he hungry but unable to obtain chopstick he need.

```
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) && (state [i] == HUNGRY) &&
        (state [(i+1) % 5] != EATING)) {
      state [i] = EATING;
      self [i]. signal();
    }
  }
```

Page No.
Date / /

```
initialization-code() {
    for (int i = 0; i < 5; i++)
        State [i] = THINKING;
    }
}
```