1 (a) Explain Dining-Philosopher's problem using monitors.

1. Shared data

- Bowl of rice (data set)

- Semaphore chopstick [5] initialized to 1

2. The structure of Philosopher *i*:

```
While (true) {
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5] );
// eat
signal ( chopstick[i] );
signal (chopstick[ (i + 1) % 5] );
// think
}
```

Problems with Semaphores

1. Correct use of semaphore operations:

- signal (mutex) …. wait (mutex)

- wait (mutex) … wait (mutex)

- Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

1. high-level abstraction that provides a convenient and effective mechanism for process synchronization

2. Only one process may be active within the monitor at a time

```
monitor monitor-name
{
// shared variable declarations
procedure P1 (…) { …. }
…
procedure Pn (…) {……}
Initialization code ( ….) { … }
…
}
}
```

Solution to Dining Philosophers

```
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 left and right neighbors
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 () ;
}
}
initialization_code() {
for (int i = 0; i < 5; i++)
state[i] = THINKING;
}
}
```

->Each philosopher *I* invokes the operations pickup()
and putdown() in the following sequence:

```
dp.pickup (i)
EAT
dp.putdown (i)
```

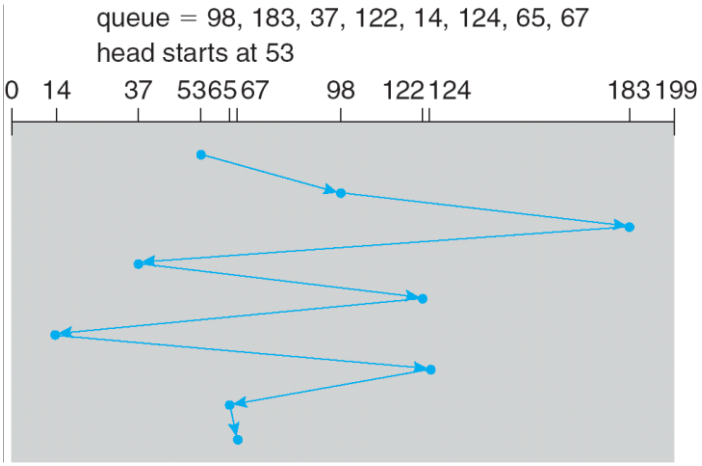
2 (a) What is disk scheduling? Discuss different disk scheduling algorithms with example.

1. **FCFS scheduling algorithm**:-

This is the simplest form of disk scheduling algorithm. This services the request in the order they are received. This algorithm is fair but do not provide fastest service.

It takes no special time to minimize the overall seek time.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 67 for a total head movement of 640 cylinders. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.

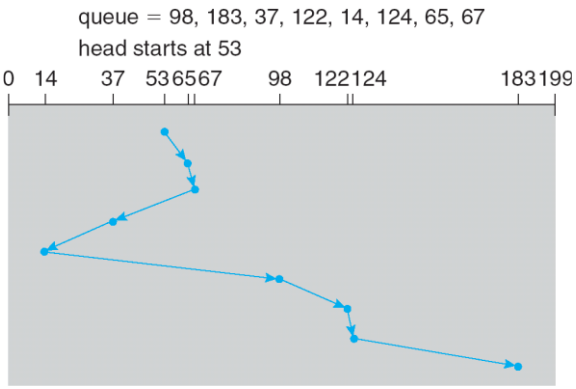
If the requests for cylinders 37 and 14 could be serviced together before or after 122 and 124 the total head movement could be decreased substantially and performance could be improved.

2. **SSTF** (Shortest seek time first) **algorithm**:-

This selects the request with minimum seek time from the current head position.

Since seek time increases with the number of cylinders traversed by head, SSTF chooses the pending request closest to the current head position.

Eg:- *:-* consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



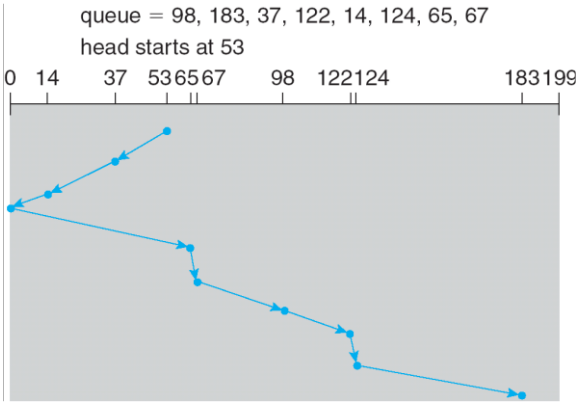
If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer then 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183.

The total head movement is only 236 cylinders. SSTF is essentially a form of SJF and it may cause starvation of some requests. SSTF is a substantial improvement over FCFS, it is not optimal.

3. **SCAN algorithm**:-

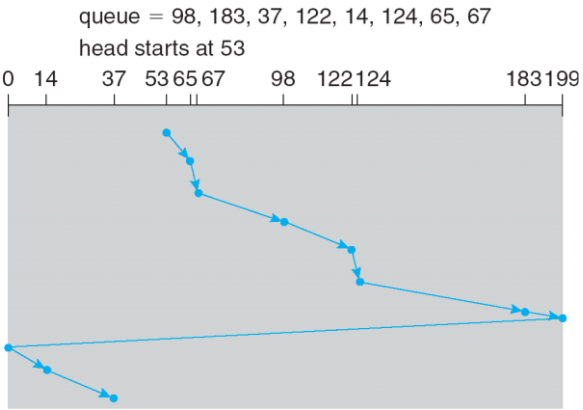
In this the disk arm starts at one end of the disk and moves towards the other end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues.

Eg:- :- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

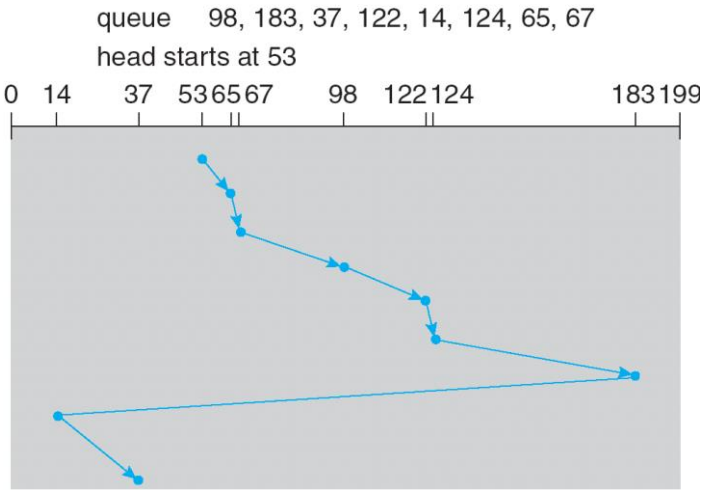


If the disk head is initially at 53 and if the head is moving towards 0, it services 37 and then 14. At cylinder 0 the arm will reverse and will move towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.
If a request arrives just in from of head, it will be serviced immediately and the request just behind the head will have to wait until the arms reach other end and reverses direction.
The SCAN is also called as elevator algorithm.

4. **C-SCAN** (Circular scan) **algorithm**:-
C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.
The C-SCAN treats the cylinders as circular list that wraps around from the final cylinder to the first one.
Eg:-



4. **Look Scheduling algorithm**:-
Both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice neither of the algorithms is implemented in this way.
The arm goes only as far as the final request in each direction. Then it reverses, without going all the way to the end of the disk. These versions of SCAN and C-SCAN are called Look and C-Look scheduling because they look for a request before continuing to move in a given direction.
Eg:-



Selection of Disk Scheduling Algorithm:-

1. SSTF is common and it increases performance over FCFS.

2. SCAN and C-SCAN algorithm is better for a heavy load on disk.

3. SCAN and C-SCAN have less starvation problem.

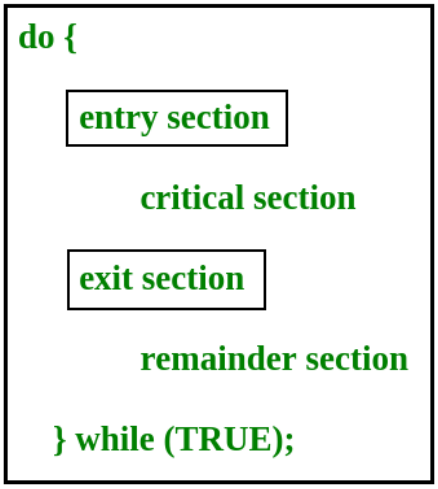
4. SSTF or Look is a reasonable choice for a default algorithm.

3 (a) Explain different IPC mechanisms available in Linux in detail.

4 (a) What is a critical section problem? What requirements should a solution to critical section problem satisfy? State Peterson's solution and indicate how it satisfies the above requirements.

Critical Section

The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results,



one must identify codes in *Critical Sections* in each thread. The characteristic properties of the code that form a *Critical Section* are

 Codes that reference one or more variables in a “read-update-write” fashion while any of those variables is possibly being altered by another thread.

 Codes that alter one or more variables that are possibly being referenced in “read-updata-write” fashion by another thread.

 Codes use a data structure while any part of it is possibly being altered by another thread.

 Codes alter any part of a data structure while it is possibly in use by another thread. Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

Solution to Critical-Section Problem

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
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

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

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.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

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 blocks 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.
-