PRACTICAL - 3

DPES
Dhole Patil Education Society

CPU Scheduling

AIM: Implement the C program for CPU scheduling algorithms :-
Shortest Job first (Preemptive) and Round Robin with different arrival time

OBJECTIVE :-

To study : CPU scheduling algorithm
1. Shortest Job first
2. Round- Robin.

THEORY :

In multiprogramming CPU scheduling is necessary to have maximum utilization / throughput of the processor so that multiple processes can handle efficiently without leaving the CPU idle.
It selects one of the processes in ready queue to be executed, and allocates the CPU to one of them. CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to Ready state.
3. Switches from Waiting to Ready
4. Terminates.

CPU Scheduling Algorithms :- Scheduling of processes / work is done to finish the work on time.
Below are different times with respect to a process.

- Arrival Time :- Time at which the process arrives in the ready queue.
- Completion Time : Time at which process complete its execution.
- Burst Time :- Time require by a process for CPU execution.
- Turn Around Time : Time difference between completion time and arrival time

  TAT = CT - AT.

- Waiting Time : Time difference between turn around time and burst time

  WT = TAT - BT.

* Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Max turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in Ready queue]
- Min Response time [Time when a process produces first Response]

* Different Scheduling Algorithms :-

There are two main strategies used for CPU scheduling

- Non preemptive : once the CPU has been allocated to a process, it keeps the CPU until process terminates or by switching to the wait state. Eg. MS Windows 3.X

- Preemptive : The process can be preempted when a higher priority process is ready for the execution.
Eg. Windows 95, Mac OS.

1. First come first serve (FCFS):
- Simplest scheduling algorithm that schedules according to arrival times of processes.
- First come first serve scheduling algorithm process that request the CPU first is allocated the CPU first.
- It is implemented by using FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

2. Shortest Job first (SJF):
- Process which has the shortest burst time is scheduled first.
- If two processes have the same burst time then FCFS is used to break the tie.
- It is a non-preemptive Scheduling algorithm.

* Round Robin :
- Each process is assigned a fixed time (Time Quantum / Time slice) in cyclic way.
- The CPU scheduler goes round in the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum & dispatches the process.
- one of two thing will then happen.
- The processor may have a CPU burust of less than 1-time quantum.
- The CPU will then select the next process in the ready queue.

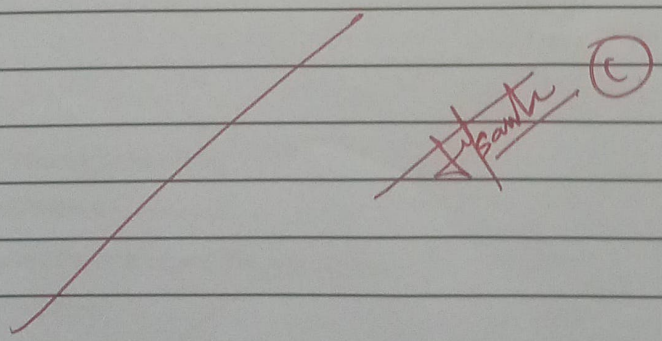
4. Priority Scheduling :-
In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first.
If priorities of two processes match, then schedule according to arrival time. Here Starvation of process is possible

5. Multilevel Queue Scheduling :-
According to the priority of process, processes are placed in the different queue. Generally high priority process is placed in the top

top level queue only after completion of process from top level queue, lower level queued processes are scheduled. It can suffer from starvation.

6. Multi level feedback queue Scheduling :
It allocas the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to lower-priority queue.

CONCLUSION : Thus, we have implemented preemptive shortest path algorithm and Round Robin Algorithm with 'C' in Linux.



PRACTICAL - 4A

ज्ञानं सर्विज्ञानम्
DPES
Dhole Patil Education Society

Thread Synchronization.

AIM : Thread Synchronization using counting semaphores. Application to demonstrate producer - consumer problem with counting semaphore & mutex.

OBJECTIVE :- To study
- Semaphore
- Mutex.
- producer - Consumer problem.

THEORY:

* Semaphores :-
Semaphores is an integer value used for signaling among processes. Only three Operation may be performed on a semaphores all of which are atomic : initialize, decrement and increment. It is known as a counting semaphore or a general semaphore.
Semaphore are the OS tools for Synchronization
Two types :
1. Binary Semaphore
2. Counting semaphore

2. Sem_wait()
It decrement the semaphore pointed by them. If a semaphore's value is greater than zero, then the decrement proceeds, and the function return, immediately.

```
int sem_wait (sem_t * sem);
```

3. Sem_post()
It increment the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a Sem_wait (3).

* Mutex
Mutexes are a method used to be sure two thread, including the parent thread, do not attempt to access shared resources at a same time.

1. Pthread_mutex_init()
The function shall initialize the mutex referenced by mutex with attribute specified by attr. If attr is NULL, the defaut mutex attributes object.

```
int pthread_mutex_init(pthread_mutex_t
* restrict mutex, cons pthread_mutexarr
_t * restrict attr)
```

```
void consumer ()
{
   while (true) {
      SemWait (n);
      SemWait (s);
      take ();
      SemSignal (s);
      SemSignal (e);
      Consume();
   }
}
void main () {
   parbegin [producer, consumer);
}
```

* POSTX Semaphores

POSIX semaphore allow processes and threads to synchronize their action. A semaphore is an integer whose value is never allowed to fall below zero. Two operation can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)).

* Semaphore functions:

1] Sem_init()

It initializes the unnamed semaphore at the address pointed to by Sem. The value arguement specifies the initial value for semaphore.

Figure illustrate the structure of buffer b. The producer can generate items & store them in buffer at its own pace. Each time an index (n) into the buffer is incremented.

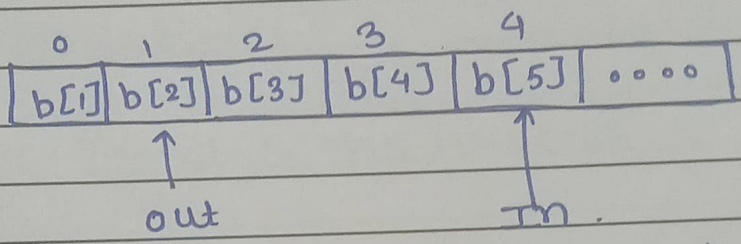


fig: figure infinite buffer for producer/consumer problem

* Solution:

```
Const int sizeofbuffer = Semaphore s=1, n=0,
  e= sizeofbuffer;
  void producer ()
{
  while (true) {
     produce ();
     SemWait (e);
     SemWait (s);
     append();
     SemSignal (s);
     SemSignal (n);
  }
}
```

```
struct semaphore {
    int count;
    queueType queue;
};
void SemWait (semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* block the process
    }
}
void SemSignal (semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process
    }
}
```

* The Producer / Consumer Problem

There are one or more producer generating some type of data & placing these in a buffer. There is a single consumer that is taking item out of the buffer one at a time. That is only one agent may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full & that is the consumer won't try to remove data from an empty buffer.

1. Binary Semaphores :-
Semaphore which are restricted to the values 0 and 1 are called binary semaphore and are used to implement locks.
It is means of suspending active processes which are later to be reactivated at such time, condition are right to be continue.

2. Counting Semaphores :-
Semaphore which allow an arbitrary resource count are called counting semaphores.

- A counting semaphore can be implemented as follows:

★ initialize :- initialize to non negative integer.

★ Decrement (SemWait)
- Process executes this is receive a signal
- If signal is not transmitted, process is suspended.
- Decrement semaphore value.
- If value become negative, process is blocked.
- Otherwise it continues execution.

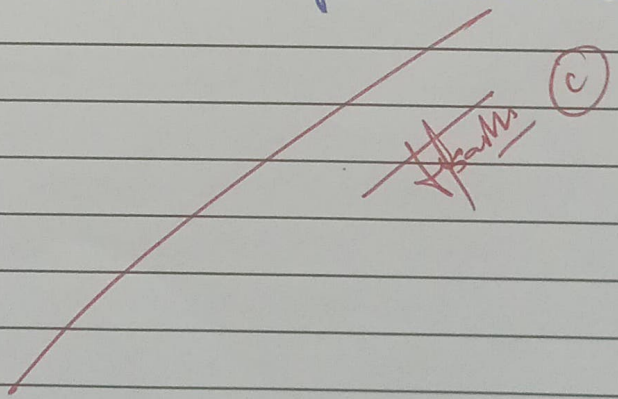
★ Increment
- Process executes it to transmit a signal.
- Increment semaphore value.
- If value is less than or equal to zero, process blocked by SemWait is unblocked.

2. pthread_mutex_destroy()
The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect uninitialized.
int pthread_mutex_destroy
(pthread_mutex_t * mutex);

3. pthread_mutex_unlock()
The function shall release the mutex object referened by mutex. The manner in which a mutux released is dependent upon the mutex's type attribute.

CONCLUSION:

Thus, we have implemented producer - consumer problem using 'C' in linux.



PRACTICAL - 4B

ज्ञानं सविज्ञानम्
DPES
Dhole Patil Education Society

AIM: Thread Synchronization and mutual exclusion using mutex
Application to demostrate:
Reader - Writer problem with reader priority.

THEORY:

* **Semaphore :-**

Semaphore is an integer value used for signaling among processes. Only three operation may be performed on a semaphore, all of which are atomic: initialize, decrement and increment.

Two Types:

1. Binary
2. counting

* **Binary semaphore :-**

Semaphore which are restricted to the values 0 and 1 are called binary semaphores and are used to implement locks. It is a mean of suspending active processes which are later to be reactivated at a such time condition are right for it to continue.

* Counting Semaphore :-
Semaphore which allows an arbitary resource count are called counting semaphore.
A counting semaphore comprises:
A integer variable, initialized to value K. During operation it can assume any value <=K, a pointer to a process queue.

* The Reader Writer Problem *
In dealing with the design of synchronization and concurrency mechanism, it is useful to be able to relate the problem at hand to known problem and to able to test any solution in term the reader / writers problem is defined as follows:
There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor register. There are a number of processes that only read the data area and a number that only writes to the data area (writer). The conditions that must be ~~it~~ satisfied as follows:

1. Any number of reader may simultaneously read file.
2. Only one writer at a time may writes to a file.
3. If a writer is writing to the file, no reader may read it

* Reader Have Priority

Figure is a solution using semaphores, showing one instances each of a read and a write; the solution does not changes for multiple readers and writers. The writer process is simple. The semaphore wsem is used to enforce mutual exclusion. The global variable readcount is used to keep the track of the number of readers, and the semaphores x is used to assure that read count is updated properly.

* POSIX Semaphores.

POSIX semaphores allow processes and thread to synchronize their action. A Semaphore is an integer whose value is never allowed to fall below zero. Two operation can be performed on semaphores: increment the semaphore value by one. ; and decrement the semaphore value by one.

Semaphore functions:

. Sem_init()

It initialize the unamed semaphore at the address pointed to by sem. The value arguement specifies the initial value for the semaphore.

```
instem_init(sem_t *sem, intpshared,
                    unsigned int value);
```

- Sem wait()

It decrement the semaphore pointed to by Sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function return immediately.

intSem_wait(Sem_t * Sem)

- Sem post()

It increment (unlock) the semaphore pointed to by Sem. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it

intSem_unlink(const char * name)

All the above function return

0: success

-1: Error

Mutex

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resources at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes

- pthread_mutex_init() :-
The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as ~~processing~~ passing the address of a default mutex attributes object.

int pthread_mutex_init(pthread_mutex_t * restrict mutex, const pthread_mutexattr_t * restrict attr);

- pthread_mutex_unlock()
The function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute.
int pthread_mutex_unlock(pthread_mutex_t *mutex)

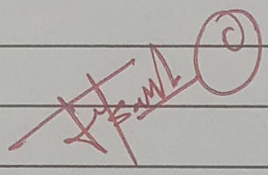
- pthread_mutex_destroy()
The function shall destroy the mutex object referenced by mutex; the mutex object becomes in effect, uninitialized.

int pthread_mutex_destroy(pthread_mutex_t *mutex);

DPCOE, WAGHOLI, PUNE

CONCLUSION:

Thus, we have implemented producer- consumer problem using "C" in linux



PRACTICAL - 5

ज्ञानं सर्वज्ञानम्
DPES
Dhole Patil Education Society

AIM :- Implement the C program for Deadlock Avoidance Algorithm : Banker's Algorithm.

OBJECTIVES : To study Banker's algorithm for deadlock avoidance.

THEORY :

* **Banker's Algorithm :**
Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in OS. It ensures that a system remains in a safe state by carefully allocating resources to processes while avoiding unsafe state that could lead to deadlocks.

- The Banker's Algorithm is a smart way for computer system to manage how programs use resources, like memory or CPU time.
- It helps prevent situations where programs get stuck and can not finish their tasks. This condition is known as deadlock.
- By keeping track of what resources each program needs and what's available, the banker algorithm makes sure that programs only get what they need in a safe order.

* Components of the Banker's Algorithm.

The following Data structure are used to implement the Banker's Algorithm:
Let 'n' be the number of processes in the system and 'm' be the number of resource types.

1) Available
- It is 1-D array of size 'm' indicating the number of available resources of each type.
- Available [j] = K means there are 'K' instances of resources type Rj.

2) Max
- It is a 2D array of size 'm * n' that defines the maximum demand of each process in a system.
- Max [i, j] = K means process Pi may request at most 'K' instances of resource type Rj.

3) Allocation:
- It is a 2-d array of size 'n * m' that defines the number of resources of each type currently allocated to each process.
- Allocation [i, j] = K means process Pi is currently allocated 'K' instances of resources type Rj.

4) Need

- It is a 2-d array of size 'n * m' that indicate the remaining resources need of each process.
- Need [i, j] = K means process Pi currently needs 'K' instances of resources type Rj
- Need [i, j] = Max [i, j] - Allocation [i, j]

* Key concept in Banker's Algorithm:-

- Safe State:
  There exists at least one sequence of processes such that each process can obtain the needed resource, complete its execution, release its resource, and thus allow other processes to eventually complete without entering a deadlock.

- unsafe State:
  Even though the system can still allocate resources to some processes, there is no guarantee that all processes can finish without potentially causing a deadlock.

Example:-

Process	Allocation			Max			Available			Remaining		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	2	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
P4	2	1	1	4	2	2	7	4	5	2	1	1
P5	0	0	2	5	3	3	7	5	5	5	3	1

For A, Total [0+2+3+2+0] = 7
Available = [10 - 7] = 3
For B, Total [1+0+0+1+0] = 2
Available [5-2] = 3
For C, Total [0+0+2+1+2] = 5
Available [7-5] = 2

For 1st iteration,
P1 needs 7 4 and 3 but available are 3 3 2.
P2 needs 1, 2, and 2 but available are 3 3 2.
It can be used.
So and P2 as the first process in the sequence <P2>
new available =
allocation of P2 + available
(2+3) (0+3) (0+2) = [5, 3, 2]

2nd iteration
P3 needs 6, 0, 0 but available 5, 3, 2.
P4 needs 2, 1, 1, available 5, 3, 2.

P4 can use resources, Add P4 to sequence
<P2, P4>

Now available using P4 = (5+2)(3+1)(2+1) = 7,4,3.

P5 needs 5,3,1 and available 7,4,3
So P5 can utilize the resources.
Add P5 to the sequence <P2, P4, P5>
Now available P5 = (0+7)(0+4)(2+3) = (7,4,5)

For P1, available in 7,4,5 & required 7,4,3
Hence P1 can utilize the resource
Add P1 to the sequence <P2, P4, P5, P1>
Now available P1 = (0+7)(1+4)(0+5) = (7,5,5)

For P3 available is 7,5,5 & required 6,0,0.
Hence it can use the resources.
Add P3 to the sequence <P2, P4, P5, P1, P3>

Hence, the order of processes to occur to avoid deadlock is <P2, P4, P5, P1, P3>.

CONCLUSION :- From this practical, I learnt how to implement the C program for deadlock avoidance algorithm : Banker's algorithm.

PRACTICAL-6

ज्ञानं सर्वविज्ञानम्
DPES
Dhole Patil Education Society

AIM :- Implement the C program for page replacement Algorithm : FCFS, LRU and optimal for frame size as minimum three

OBJECTIVES :- To study page replacement algorithm:

1) FCFS
2) LRU
3) Optimal

THEORY :-

* **Page Replacement Algorithm :-**

Page replacement algorithm are needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and operating system replaces one of the existing pages with newly needed page. Different page replacement algorithm suggest different ways to decide which page to replace. The target for all algorithm is to reduce number of page fault.

1) First In First Out (FIFO) page Replacement algorithm :-

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue. When a page needs to be replaced page in the front of the queue is selected for removal.

* Algorithm :-

1) Start the process
2) Read number of pages n.
3) Read number of pages no.
4) Read page number into an array a [i]
5) Initialize avail [i] = 0. to chek page hit
6) Replace the page with circular queue, while re-placing check page availability in the frame place avail [i] = 1 if page is placed in the frame count page fault.
7) Print the result
8) Stop the process.

* Example :-

consider page reference string 1, 3, 0, 3, 5, 6 and page slot 3.

String : 1, 3, 0, 3, 5, 6

		0	0	0	0
	3	3	3	3	6
1	1	1	1	5	5
*	*	*	H	*	*

∴ total page faut = 5.

2) Least Recently used (LRU) page replacement algorithm :-

In Least Recently used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely.

* Algorithm :-

1) Start the process.
2) Declare the size.
3) Get the number of pages to be inserted.
4) Get the value.
5) Declare counter and stack.
6) Select the LRU page by counter value.
7) Stack them according the selection.
8) Display the values
9) Stop the process.

* Example :-

Page Reference String : 7 0 1 2 0 3 0 4 2 3 0 3 2

frame Size : 4

			2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
			*	H	*	*	*	H	H	H	H	H	*	H	H	H	H	*	H	H

page hit = 12

page fault = 8

3) Optimal page replacement Algorithm :-

- In this algorithm of page replacement, pages are ~~placed~~ replaced which would not be used for the longest duration of time in the future.
- In optimal page replacement for every reference we do following :-

i) If referred page is already present, increment hit count

ii) If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

* Algorithm.

1) Start program.
2) Read Number of pages and frames.
3) Read Each page value.
4) Search for page in the frames.
5) If not available allocate free frame.
6) If No frames is free replace the page with the page that is leastly used
7) Print Page number of page fault
8) Stop process.

* Example :-

String :- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3
frame Size :- 4.

			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
*	*	*	*	H	*	H	*	H	H	H	H	H	H

Page hit = 8
page fault = 6

CONLUSION :- From this practical, I learnt to implement C program for page replacement algorithm :- FCFS, LRU and optimal for frame size as minimum three.

PRACTICAL - 7A

Inter Process communication using FIFO.

AIM : Full duplex communication between two independent process First process accepts sentences and writes on one pipe to be read by second process & second process count number of characters, number of words and number of lines in accepted sentences, writes this output in text file and writes the content of the file on second pipe to be read by first process & displays on standard output.

OBJECTIVES : To Study

- FIFO
- FIFO operation
- use of FIFO for inter process communication.

THEORY :

FIFO :-

A FIFO (first In first out) is a one way flow of data FIFO have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. Any process can open or close the FIFO. FIFO are also called named pipes.

* Properties:

1. After a FIFO is created, it can be opened for read or write.
2. Normally, opening a FIFO for read or write, it blocks until another process opens it for write or read.
3. A read gets as much data is request or as much data as the FIFO has, whichever is less.
4. A write to a FIFO is atomic, as long as the write does not exceed the capacity of the FIFO.
5. FIFO must be opened by two processes :, one opens it as Header on one end, the other opens it as Sender on the other end.

* Creating a FIFO

A FIFO is created by the mkfifo function. Specify the path to the FIFO on command line.

For example :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char * pathname,
            mode_t mode);
```

Pathname : a UNIX pathname. (path and filename)
The name of FIFO.

mode : The file permission bits. It specifies the pipe's owner, group & world permission.

* Accessing a FIFO:-

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O function like open, write, read, close or C library I/O function (fopen, fprintf, fscanf, fclose, & so on) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONGLY);
write (fd, data, data_length);
close (fd) ;
```

To read a string from the FIFO using C library I/O function, you could use this code:

```
FILE * fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to maximum size of PIPE_BUF (4KB on linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

* close : To close an open FIFO use close ()
* unlink : To delete a created FIFO, use unlink().

CONCLUSION :

Thus, we studied inter process communication using FIFOs

PRACTICAL - 7B

ज्ञानं सर्वज्ञानम्
DPES
Dhole Patil Education Society

Inter-process communication using Shared memory using System v.

AIM: Application to demonstrate : client and server program in which server process creates a shared memory segment and writes the message to the shared memory segment. client process reads the message from the shared memory segment & display it to the screen.

OBJECTIVE :-

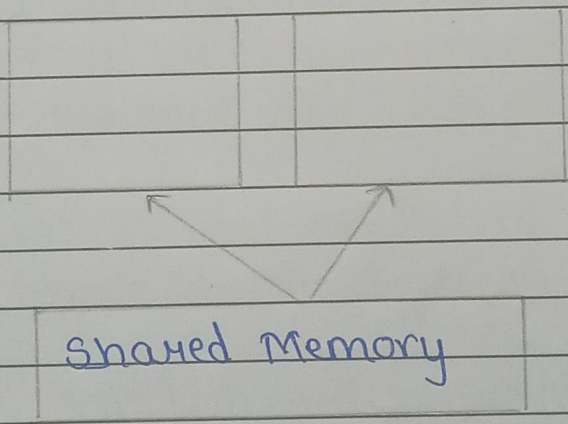
The aim of this laboratory is to show you how the processes can communicate among themselves using the shared memory region. Shared Memory is an efficient means of passing data between program A shared memory segment is described by a control structure with a unique ID that point to an area of physical memory. In this lab the following issues related to shared memory utilization are discussed :

- creating a shared memory segment
- controlling a shared memory segment
- Attaching and Detaching a shared memory segment

THEORY :

* What is Shared Memory ?

A shared memory is an extra piece of memory that is attached to some address space for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race condition may occur if memory accesses are not handled properly.



Shared memory is a features supported by UNIX System V, including Linux, SunOS and Solaris.

one process must explicitly ask for a area, using a key, to be shared by other processes. This process will be called the server. All

other processes, the clients that know the shared area access it.
A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type shmid_ds in header file sys/shm.h

* A general scheme of using shared memory is the following:
For a server, it should be started before any client. The server should perform the following tasks:

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call shmget().
2. Attach this shared memory to the server's address space with system call shmat().
3. Initialize the shared memory, if necessary.
4. Do something and wait for all client's completion.
5. Detach the shared memory with system shmdt()
6. Remove the shared memory with System call shmctl().

* For the client part, the procedure is almost the same:

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client's address space.
3. use the memory.
4. Detach all shared memory segment, if necessary
5. Exit.

* Asking for shared Memory Segment - Shmget() :-

The system call that requests a shared memory segment is shmget().
It is defined as follows :

```
shm_id = shmget (
        key_t K,
        int size,
        int flag, );
```

If Shmget() can successfully get the requested shared memory it function value is non negative integer, the shared memory ID,

the Shared memory ID; otherwise the function value is negative.

* Keys:
UNIX requires a key of type key_t defined in file sys/types.h for requesting resources such as shared memory segments, message queues and semaphores.
A key is simply an integer of type key_t. however, you should not use int or long, since the length of a key is system dependent.

There are three different ways of using keys :-
1. A specific integer value (eg. 123456)
2. A key generated with function ftok()
3. A uniquely generated key using IPC_PRIVATE (i.e., a private key)

* Detaching and Removing a Shared Memory Segment - Shmdt() and Shmctl() :-

System call shmdt() is used to detach a shared memory. After a shared memory is detached, it cannot be used.
However, it is still there & can be re-attached back to a process's address space, perhaps at a different address.
To remove a shared memory use shmctl().

- The only arguement to shmdt() is the shared memory address returned by shmat().

Syntax:

```
shmdt(shm_ptr);
```

where, shm_ptr is the pointer to the shared memory. This pointer is returned by shmat()
If the detach operation fails, the returned function value is non-zero.

- To remove a shared memory segment, use the following code:

```
shmctl(shm_id, IPC_RMID, NULL);
```

Where shm_id is the shared memory ID.
IPC_RMID indicates this a remove operation.
NOTE: After removal of shared memory ~~management~~ segment, if you want to use it again, you should use shmget() followed by shmat()

CONCLUSION :-

Thus, we studied inter-process communication using shared memory using systemv.

PRACTICAL -8

DPES
Dhole Patil Education Society

TITLE : Implement the C program for Disk Scheduling algorithm: SSTF, SCAN, C-LOOK considering the initial head position moving away from the Spindle

OBJECTIVES :

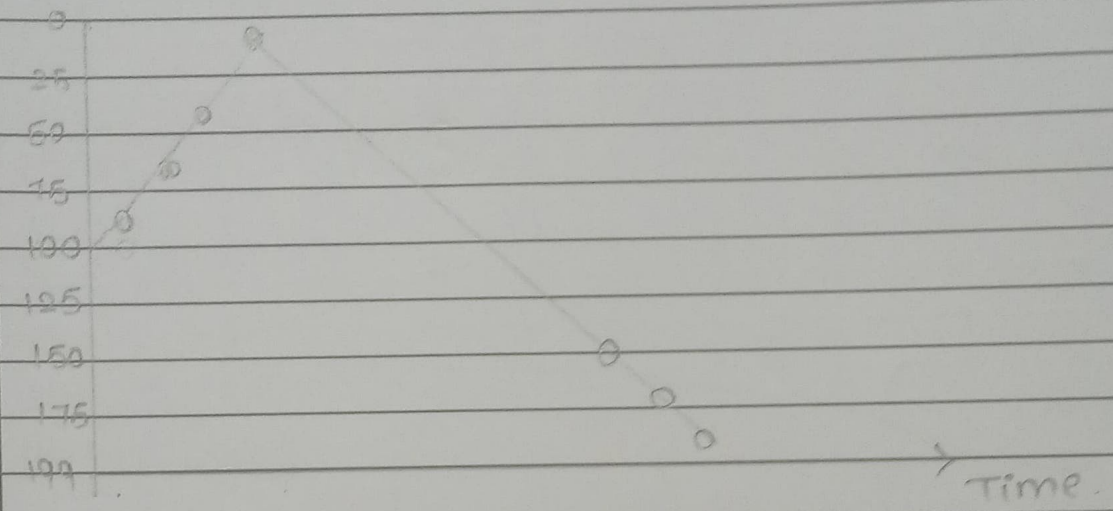
- To Study the concept of disk scheduling in linux
- learn to implement disk scheduling algorithm in the linux

THEORY :

* **Shortest Service Time first (SSTF):**
Select the disk I/O request that requires the least movement of the disk arm from its current position
Always choose the minimum seek time.
Selects the request with the minimum seek time from the current head position. SSTF scheduling is a form of SJF scheduling ; may cause starvation of some request.

Example :
To compare various schemes, consider a disk head is initially located at track 100. Assume a disk with 200 tracks and that the disk request queue has random ~~qu~~ request it. The requested tracks, in the order

received by the disk scheduler, are:
55, 58, 39, 18, 90, 160, 150, 38, 184



b) SSTF

* SCAN

Arm moves in one direction only, satisfying all outstanding request until it reaches the last track in that direction is reversed.

The disk arm start at one end of the disk and moves toward the other end, servicing request until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

SCAN algorithm sometimes called the elevator algorithm.

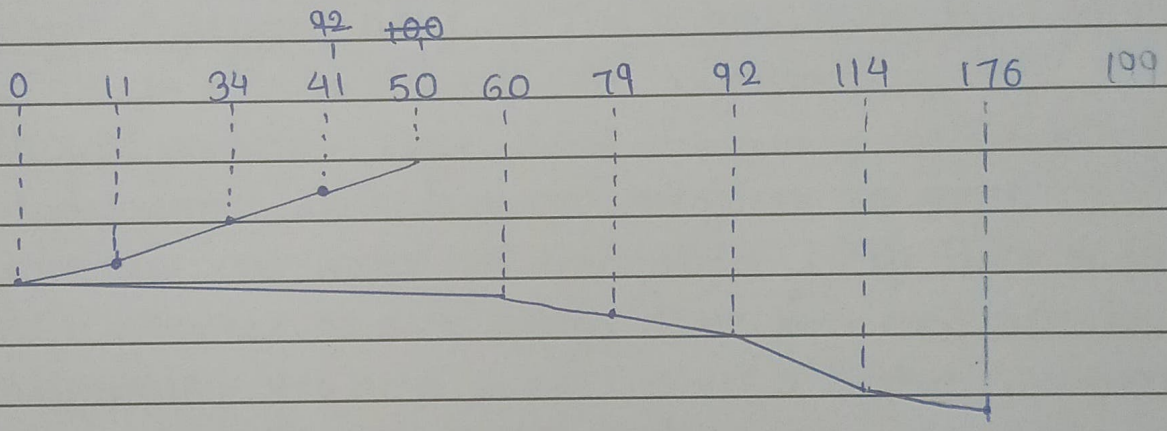
Example:

Sequence = {176, 79, 34, 60, 92, 11, 41, 114}

head position = ~~100~~ 50

Direction = left

→



= (50-41) + (41-34) + (34-11) + (11-0) + (60-0) + (79-60) + (92-79)
(114-92) + (176-114)

= 226

= (50-0) + (176-0) (from right to left)

= 50+176

= 226

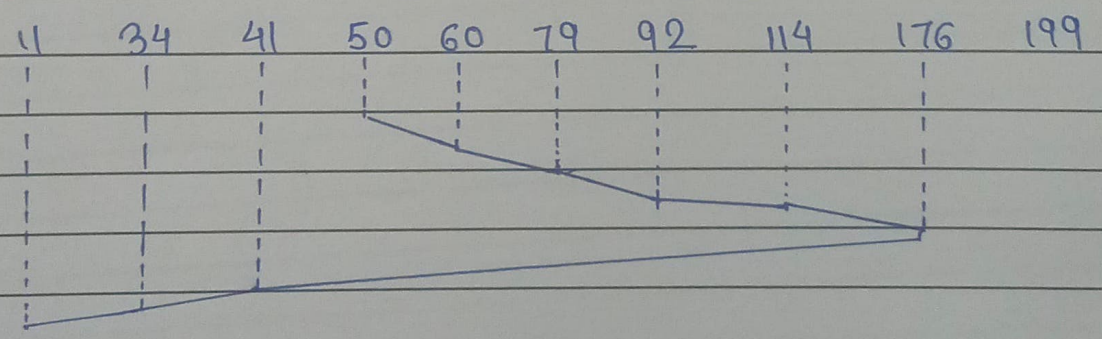
= 226/8

= 28.26

* C-LOOK

C-look is an enhanced version of both SCAN as well as look disk scheduling algorithm. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm.
In this algorithm, the head service request only in one direction until all the request in this direction are not serviced and then jump back to the farthest request on the other direction and service the remaining request which gives a better uniform servicing as well as avoids wasting seek time forgoing till the end of the disk.

Examples:
Sequence = {176, 79, 34, 60, 92, 11, 41, 114}
head position = 50
Direction = right



= (60-50) + (79-60) + (92-79) + (114-92) + (176-114) + (176-11) + (34-11) + (41-34)
= 321

CONCLUSION : Thus we have studied and implemented disk scheduling algorithm : SSTF, SCAN, C-LOOK considering the initial head position moving away from the spindle