Practical No. 4

Aim: Implementation of various CPU scheduling algorithms (FCFS, SJF, Priority).

Theory:

FCFS:

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

SJF (shortest job first):

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

Priority:

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of two process is same then they will be scheduled according to their arrival time.

Program:

FCFS:

```
#include<stdio.h>
void main()
{ int n;
  printf ("Enter the no of processes : ")
  scanf ("%d", &n);

  int burst_time [n], wt[n], tat[n];
  for( int P=0; P<n ; P++)
  {
     printf ("Enter the burst time of process P%d", i);
     scanf ("%d", & burst_time [i]);
  }
  wt[0] = 0, tat [0] = burst_time [0];
  for (int i=1; i< n ; i++)
  { wt[i] = wt[i-1] + burst_time [i-1];
    tat[i] = tat [i-1] + burst_time[i];
  }

  printf("\n Process\t Waiting Time\t Turn around time\n");
  for (int i=0 ; i<n ; i++)
  {
    printf ("P%d \t %d \t\t %d \n", i, wt[i], tat[i]);
  }
}
```

O/P:

```
Enter the no of processes : 5
Enter the burst time of process no : 5
```

Enter the burst time of process P1 : 6
Enter the burst time of process P2 : 2
Enter the burst time of process P3 : 8
Enter the burst time of process P4 : 1

Process	waiting Time	Turn around time
P0	0	5
P1	5	11
P2	11	13
P3	13	21
P4	21	22

Shortest Job first :

```
#include <stdio.h>
void main()
{
  int A[100][4];
  int i, j, n, index, temp;
  printf("Enter number of process : ");
  scanf("%d", &n);
  printf("Enter Burst Time : \n");

  for (i=0; i<n; i++)
  {
     printf("P%d : ", i+1);
     scanf("%d", &A[i][1]);
     A[i][0] = i+1;
  }
```

```
for(i=0 ; i<n ;i++)
{
  index =i;
  For(j=i+1; j<n ; j++)
     if(A[j][1] < A[index] [1])
        index=j;
  temp = A[i][1];
  A[i][1] = A[ index][1];
  A[index][1] = temp;

  temp = A[i][0];
  A[i] [0] = A[index] [0];
  A[index][0) = temp;
}
A[0][2]=0;

for( i=1; i<n; i++)
{
  A[i][2] = 0;
  for (j=0; j<i ;j++)
     A[i][2] += A[j][1];
}

for( i=0; i<n; i++)
  A[i][3] = A[i][1]

printf("\n Process  BurstTime  waiting time  Turn
    around time \n");
```

```
for(i=0; i<n; i++)
{
   for( j=0 ; j<n; j++)
   {
     if(A[j][0]==i+1)
      printf("P%od \t %od \t %od \t %od \n",
          A[j][0], A[j][1], A[j][2], A[j][3]);
   }
}
```

O/P:-

Enter number of process : 5

Enter Burst time:-

P1 : 5

P2 : 6

P3 : 4

P4 : 8

P5 : 2

Process	Burst Time	waiting Time	Turn around Time
P1	5	6	11
P2	6	11	17
P3	4	2	6
P4	8	17	25
P5	2	0	2

#Priority:

```
#include <stdio.h>
void main()
{
 int A[100][5];
 int i, j, n, index, temp;
 printf ("Enter number of process :");
 scanf("%d", &n);
 printf ("Enter Burst Time :\n");

 for( i=0 ; i<n ; i++)
 {
  printf ("P%d : ", i+1);
  scanf ("%d", &A[i][1]);
  A[i][0] = i+1;
 }
 printf("Enter the priority of processes :\n");
 for ( i=0 ; i<n ; i++)
 {   printf (" P%d :", i+1);
     scanf ("%d", &A[i][2]);
 }

for( i=0; i<n; i++)
{   index = i;
    for ( j= i+1 ; j<n ; j++)
        if ( A[j][2] < A[index][2])
            index = j ;

    temp = A[j][2];
```

```
        A[i][2] = A[index][2];
        A[index][2] = temp;

        temp = A[i][0];
        A[i][0] = A[index][0];
        A[index][0] = temp;

        temp = A[i][1];
        A[i][1] = A[index][1];
        A[index][1] = temp;
    }
    A[0][3]=0;
    for(i=1; i<n; i++)
    {
        A[i][3]=0;
        for(j=0; j<i; j++)
            A[i][3] += A[j][1];
    }
    for(i=0; i<n; i++)
        A[i][4] = A[i][1]+A[i][3];

printf("\n Process Burst Time Priority waiting Time
    Turn around time\n");
for(i=0; i<n; i++)
{   for(j=0; j<n; j++)
    {   if(A[j][0]==i+1)
        printf("P%d \t %d \t %d \t %d \t %d \n",
        A[j][0], A[j][1], A[j][2], A[j][3], A[j][4]);
    }
}
```

O/P:

Enter the number of process : 5

Enter Burst Time :

P1 : 3

P2 : 5

P3 : 2

P4 : 8

P5 : 4

Enter the priority of processes :

P1 : 2

P2 : 5

P3 : 1

P4 : 4

P5 : 3

Process	Burst time	Priority	waiting time	Turnaround time
P1	3	2	2	5
P2	5	5	17	22
P3	2	1	0	2
P4	8	4	9	17
P5	4	3	5	9

Conclusion: We have successfully implemented the FCFS, SJF and priority scheduling algorithm.

Practical No 5

Aim:- Implementation of various page replacement algorithm (FIFO, Optimal, LRU).

Theory:-

In operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page replacement algorithms:

① First In First Out [FIFO]:- This is the simplest algorithm. In this algorithm, OS keeps track of all pages in the memory in queue. Oldest page is in front of the queue. When a page needs to be replaced, page in the front of the queue is selected for removal.

eg.

1	3	0	3	5	6
1	1	1	1	5	5
	3	3	3	3	6
		0	0	0	0
x	x	x	Hit	x	x

5 Page fault

② * Belady's Anomaly:-

It proves that it is possible to have more page faults when increasing the number of page frames while using first in first out [FIFO] page replacement algorithm.

eg. we consider reference string 3,2,1,0,3,2,4,3,2,1,0,4. and 3 frame, we get 9 total 9 page faults, but if we increase frames to 4 we get 10 page faults.

(11) Optimal Page Replacement :

In this algorithm pages we replaced which would not be used for the longest duration of time in the future.

Let us consider page reference string 7,0,1,2,0,3,0,4,2,3,0,3,2 & 3 page.

1. Initially all slots are empty, so when 7,0,1,2 are allocated empty slots → 4 page faults

Now for the future page reference string → 0 page fault because the already available in the memory.

(11) Least Recently used (LRU) :

In this algorithm the page will be replaced which is least recently used.

Let say the page reference string is 7,0,1,2,0,3,0,4,2,3,0,3,2. Initially we have 4 frame.

When 3 come it will take the place of 7 because it is least recently used → 1 page fault

0 is already in memory → 0 page fault

4 will take place of 1 → 1 page fault. Now the further page reference string → 0 page fault because they are already available in the memory.

Program :

```
#include <stdio.h>
int main()
{
  int size, m, n, s, pages, pagefaults = 0, frames;
  printf ("Enter the no of pages : ");
  scanf ("%d", &size);
  printf("Enter the pages: \n");
  int refstring[size];
  for(int i=0 ; i< size ; i++)
  scanf("%d", &refstring[i]);

  printf ("Enter no of frames : ");
  scanf ("%d", &frames);

  pages = sizeof(refstring) / sizeof(refstring[0]);

  printf ("\n Incoming \t");frame
  for( int i=0;  i< frames ; i++)
   printf ("fram %d\t ", i+1);

  int temp[frames];
  for( m=0; m < frames ; m++)
     temp[m] = -1;

  for(m=0; m <pages ; m++)
  {
     s=0;
     for (n=0 ; n < frames ; n++)
```

```
& if(refstring[m] == temp[n])
  {
    s++;
    pagefaults--;
  }
}
  pagefaults++;
if((pagefaults<=frames) && (s==0))
   temp[m] = refstring[m];
else if (s==0)
   temp[(pagefaults-1) % frames] = refstring[m];

printf("\n");
printf("%d\t\t", refstring[m]);
for(n=0; n<frames; n++)
{
   if(temp[n] != -1)
     printf("%d\t\t", temp[n]);
   else
     printf("-\t\t");
  }
}

printf("\n\n Total Page faults : %d\n", pagefaults);
return 0;
}
```

O/P :-

Enter the no of pages : 7
Enter the pages :
2
4
1
3
2
5
3
Enter the no of frames : 3

Incoming	Frame1	Frame2	Frame3
2	2	-	-
4	2	4	-
1	2	4	1
3	3	4	1
2	3	2	1
5	3	2	5
3	3	2	5

Total Page faults : 6

Conclusion :- We have successfully implement the first ~~coming~~ first out page replacement algorithm.

Practical No 6

Aim: Concurrent programming ; use of threads and processes ; system calls (fork and V-fork).

Theory:

system call : A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the OS.

services provided by system calls:

1) Process creation and management
2) Main memory management
3) File access, directory and file system management
4) Device management
5) Protection
6) Information maintainance
7) Communication

fork() in C :-

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

A child proces uses the same pc (program counter

same CPU registers, same open files which use in the parent process.

It takes no parameter and returns an integer value. Below are different values returned by fork()

Negative value: creation of child process was unsuccessfull.

Zero: Returned to the newly created child Process.

positive value: Returned to parent or caller. The value contains the process ID of newly created child process.

Vfork():

vfork() is also system call which is used to create new process. New process created by vfork() system call is called child process and process that invoked vfork() system call is called parent process. code of child process is same as code of its parent process. child process suspends execution of parent process until child process completes its execution as both processes share the same address space.

Program:

fork:

```
#include <stdio.h>
#include < sys/ types.h)
Int main()
{
```

```
    fork();
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

Output:

Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello

Here we get 8 times Hello as we called fork 3 time. (2^3).

vfork():

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    vfork();
    vfork();
    vfork();
```

```
    printf("Vfork \n");
    return 0;
}
```

Output :-

Vfork
Vfork
Vfork
Vfork
Vfork
Vfork
Vfork
Vfork

Conclusion :- We have successfully implement the program for system call fork() and vfork().

Chhatrapati Shahu Maharaj Shikshan Sanstha's
CHH. SHAHU COLLEGE OF ENGINEERING
Kanchanwadi, Paithan Road, Aurangabad.

Date :

Practical No 7

Aim: Study pthreads and implement the following: write a program which shows the performance.

Theory:

The POSIX thread libraries are C/C++ thread API based on standards. It enables the creation of a new concurrent process flow. It works well on multi-processor or multicore systems, where the process flow may be scheduled to execute on another process increasing speed through parallel or distributed processing. Because the system does not create a new system ~~does not create a new system~~, virtual memory space and environment for the process, threads needles overhead than "Forking" or creating a new process.

To utilise the Pthread interfaces, we must include the header pthread.h at the start of the CPP script.

```
#include<pthread.h>
```

PThreads is a highly concrete multithreading system that is the UNIX systems default standard. PThreads is an abbreviation for POSIX threads, and POSIX is an abbreviation for Portable operating system Interface, which is a type of interface that the operating system must implement. PThreads in POSIX outline the threading APIs that the operating system must provide.

Why is Pthreads used?

- The fundamental purpose for adopting pthreads is to improve programme performance.
- When compared to the expense of starting and administering a process, a thread requires far less operating system overhead. Thread management takes fewer system resources than process management.
- A process's threads all the shares the same address space. Inter thread communication is more efficient and, in many circumstances, more user-friendly than inter process communication.

Program :

```
#include <stdio.h>
#include <pthread.h>

void * func (void * arg)
{
    pthread_detach (pthread_self());
    printf(" Inside the thread \n");
    pthread_exit (NULL);
}

int main()
{
    pthread_t ptid;
    pthread_create (&ptid, NULL, &func, NULL);
```

```
printf("This line may be printed before thread terminate")
if(pthread_equal(ptid, pthread_self()))
    printf("Threads are equal\n");
else
    printf("Threads are not equal\n");

pthread_join(ptid, NULL);
printf("This line will be printed after thread
    ends\n");

pthread_exit(NULL);
return 0;
}
```

Output :-

1st :- This line may be printed before thread terminates.
Inside the thread
Threads are not equal
This line will be printed after thread ends.

2nd time :-
This line may be printed before thread terminates
Threads are not equal
This line will be printed after thread ends
Inside the thread.

Conclusion :- We have successfully implement the program to demonstrate pthreads.

Practical No. 8

Aim: Implementation of synchronization primitives - semaphore, locks and conditional variables.

Theory:

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.

Semaphore is simply an integer variable that is shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types

1. **Binary semaphore:**

This is also known as mutex lock. It can have only two values - 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

2. **Counting semaphore:**

It's values can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

We can have two operations that can be used to access and change the value of the semaphore variable.

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

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

Program :-

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;
void * ~~routh~~thread (void *arg)
{
    //wait
    sem_wait(&mutex);
    printf("\n Entered ~~thread~~\n");

    //critical section
    sleep(4);
```

```
    //signal
    printf("\n Exit thread\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread, NULL);
    sleep(2);
    pthread_create(&t2, NULL, thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&mutex);
    return 0;
    return 0;
}
```

Output :-

```
Entered thread
Exit thread
Entered thread
Exit thread
```

- If we would not have used semaphore, the output would have been as follows due to context switching;

```
Entered thread
Entered thread
```

Exit thread
Exit thread

Conclusion:- We have successfully implement the program to demonstrate the use of semaphore.

Practical No 9

Aim: Implementation of Producer-consumer problem, Bankers algorithm.

Theory: The producer consumer problem is a sychronisation problem. There is a fixed size buffer and the buffer produces item and enter them into the buffer. The consumer removes the items from the buffer and consume them.

The producer-consumer problem resolved using semaphores. The code for producer and consumer process are given below:

Producer process:

```
do{ ------
    /* produce an item in next_produced */
    ------
    wait(empty);
    wait(mutex);
    ---
    /* add next_produced to the buffer */
    ----
    signal(mutex);
    signal(full);
} while(true);
```

* Structure of producer process

Consumer process:

```
do {
    wait (full);
    wait (mutex);
    ------
    /* remove an item from buffer to next-consumed */
    ------
    signal (mutex);
    signal (empty);
    -----
    /* consume the item in next-consumed */
    -----
} while (true);
```

* The structure of consumer process.

defination of wait operation:

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

defination of signal operation:

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

```
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = n;
```

Above semaphors are shared between the producer and consumer process.

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

Bankers Algorithm:-

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "is-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following data-structures are used to implement Banker's Algorithm:-

(i) Available (vector) :- It indicates the no of available resources of each type.

(ii) Max :- It is n×n matrix defining the maximum demand of each process

(iii) Allocation :- It is a n×n matrix defining the no of resources of each type currently allocated to each process.

(iv) Need :- It is n×n matrix indicating the remaining resource need of each resources.

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Banker's algorithm consists of safety algorithm and resource request algorithm.

Safety Algorithm :

It is use for finding out whether or not a system is in safe state.

1. Let work and finish be the two vectors of length m and n respectively.
   Initialize : work = available
   finish[i] = false , where i = 0, 1, ... n-1.
2. Find an index i such that both
   (a) finish[i] = false
   (b) Needi ≤ work
   If no such i exists goto step (4)
3. work = work + Allocation[i]
   finish[i] = true
   goto step (2)
4. If finish[i] = true for all i
   then the system is in safe state.

Resource-Request Algorithm :

Let Requesti be the request array for process Pi. Requesti[j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

(1) If Requesti <= Needi
Goto step(2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

② If Request i <= Available
Goto step(3); otherwise, Pi must wait, since the resources are not available.

③ Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available - Requesti
Allocationi = Allocationi + Requesti
Needi = Needi - Requesti

Program:

```
#include <stdio.h>
#include <stdlib.h>

int mutex=1;
int full=0;
int empty=10, n=0;

void producer()
{
    --mutex;
    ++full;
    --empty;
    n++;
    printf("\n Producer produces Item %d", x);
    ++mutex;
}
```

```
void consumer()
{
  --mutex ;
  ++full;
  ++empty;
  x++;
  printf("\n Consumer consumes item %d", x);
  x--;
  ++mutex;
}

int main()
{
  int n,i;
  printf("\n1. Press 1 for Producer"
         "\n2. Press 2 for consumer"
         "\n 3. Press 3 for Exit");

# pragma omp critical
   for( i=1; i>0; i++)
   {
    printf("\n Enter your choice :");
    scanf ("%d", &n);

    switch (n)
    {
     case 1:
          if((mutex==1) && (empty!=0))
           producer();
```

```
        else
            printf("Buffer is Full!");
        break;
    case 2:
        if((mutex==1) && (full!=0))
            consumer();
        else
            printf("Buffer Is empty!");
        break;
    case 3:
        exit(0);
        break;
    }
  }
}
```

Output :-

1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit.

Enter your choice : 1

Producer produces Item 1
Enter your choice : 1

Producer produces item 2
Enter your choice : 2

Consumer consumes item 2
Enter your choice : 1

producer produces item 2
Enter ayour choice:3

Conclusion:- We have successfully implement the program for producer - consumer problem.

Practical No. 10

Aim: Implementation of various memory allocation algorithms, (First fit, Best fit and Worst fit), Disk scheduling algorithms (FCFS, SCAN, SSTF, C-SCAN).

Theory:

One of the simplest method for memory allocation is to divide memory into several fixed partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input from the input queue is loaded into the free partition. When a process terminates, the partition becomes available for another process. The OS keeps a table indicating which parts of memory are available and which occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided.

① Best fit:

The best fit allocates the process to a partition which is the smallest sufficient partition among the free available partition. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequated. It then tries to find a hole which is ~~the~~ least to actual process size needed.

Advantage: Memory utilization is much better than

first fit as it searches the smallest free partition first available.

Disadvantage:- It is slowest and may even tend to fill up memory with tiny ~~holes~~ useless holes.

(ii) First fit:-

In the first fit approach is to allocate the first free partition or hole large enough which can accomodate the process. It finishes after finding the first suitable free partition.

Advantage:- Fastest algorithm because it searches as little as possible.

(iii) Worst fit:-

Worst fit allocates a process to the partition which is largest sufficient among a freely available partitions available in main memory.

If a large process comes at a later stage, then memory will not have space to among fit.

Disk Scheduling Algorithms:-

It is done by os to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling. There are many disk scheduling algorithms. ~~but~~ some of them are following.

(1) FCFS (First come first serve):-

FCFS is the smallest of all the disk scheduling algorithms. In FCFS the requests are addressed in the order they arrive in the disk queue.

(ii) SCAN:

In SCAN algorithm the disk arm moves into a particular direction and serve the request coming in its path and after reaching the end of disk it will reverse its direction.

As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

(iii) SSTF:

Shortest seek time first algorithm selects disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

It allows the head to move to the closest track in the service queue.

(iv) C-SCAN:

In this, the disk arm again scan the path that has been selected, after reversing its direction. So it may be possible that too many requests are waiting at the other end or there may be zero.

Program :-

First fit memory allocation :-

```
#include<stdio.h>

int main()
{
    int m, n, i, j ;
    printf("Enter number of blocks in the memory :");
    scanf("%d", &m);
    printf("Enter number of processes in the input queue");
    scanf("%d", &n);

    int blockSize[m], processSize[n], allocation[n];
    printf("Enter the block size in the memory one
            by one : \n");
    for (i=0; i<m; i++)
        scanf("%d", &blockSize[i]);
    printf("Enter the memory require by process
            one by one :\n");
    for(i=0; i<n; i++)
        scanf("%d", &processSize[i]);

    for(i=0; i<n; i++)
        allocation[i] = -1;

    for(i=0; i<n; i++){
    {
        for (j=0; j<m; j++)
        {
```

```
        if(blockSize[j] >= processSize[i])
        {
            allocation[i] = j;
            blockSize[j] -= processSize[i];
            break;
        }
    }
}
printf("Enter \nProcess No. \t Process Size \t Block no\n");
for(int i=0; i<n; i++)
{
    printf("%d \t\t", i+1);
    printf("%d \t\t\t", processSize[i]);
    if (allocation[i] != -1)
        printf("%d", allocation[i]+1);
    else
        printf("Not allocated");
    printf("\n");
}
return 0;
}
```

Output :-

```
Enter number of blocks in the memory : 5
Enter number of processes in the input queue: 4
Enter the block size in the memory one by one:
100
500
```

200
300
600
Enter the memory required by process one by one :
121
417
212
426

PROCESS NO	Process size	Block no
1	121	2
2	417	5
3	212	2
4	426	not allocated

* RCFS disk scheduling

```
#include<stdio.h>
#include<math.h>
int main()
{
    int size, head, seek_count=0, distance, cur_track;
    printf("Enter the size of requests :");
    scanf("%d", &size);
    int arr[size];
    printf("Enter requested queue one by one : \n");
    for (int i=0; i<size; i++)
        scanf("%d", &arr[i]);

    printf("Enter position of read write head :");
    scanf("%d", &arr[i]);
```

```
for(int i=0; i< size; i++)
{
   cur_track = arr[i];
   distance = abs(cur_track - head);
   seek_count += distance;
   head = cur_track;
}

printf("\n Total number of seek operations = %d\n",
      seek_count);
printf("Seek sequence is\n");
for (int i=0; i < size; i++)
{
   printf("%d\n", arr[i]);
}
return 0;
}
```

Output :-

```
Enter the size of request : 7
Enter requested queue one by one!
82
170
43
140
24
16
190
```

Enter the position of read write head : 50

Total number of seek operations = 642
Seek sequence Ps :
82
170
43
140
24
16
190

Conclusion :- We have successfully implement the First fit memory allocation and first come first serve disk scheduling algorithms.