Experiment No 1

Aim : To explore usage of basic Linux Commands and system calls for file, directory and process management.

Theory:

1) To create a subdirectory named mystuff in the current directory

```
$ mkdir mystuff
```

2) To change directory, use the cd command. The syntax is cd followed by the name of the directory you want to go to.

```
$ cd  Desktop

$ pwd
 /home/student/Desktop
```

3) To show content of the given file. Cat command reads data from the file and gives their content as output

```
$ cat filename
$ cat file1 file2
```

   This will show the content of file1 and file2

4) To list contents inside a directory, use ls command

```
$ ls mystuff
```

5) To change owner of the file: use chown command

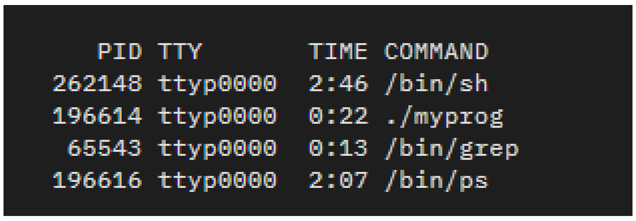
```
$ chown owner_name file_name
$ sudo chown root file1.txt
```

6) To change access mode of the file, use chmod command
   $ chmod 200 testfile

7) To change the group ownership of a file or directory, use chgrp command
   $ sudo chgrp geeksforgeeks abc.txt

8) You can use the **ps** command to display a list of your processes that are currently running and obtain additional information about those processes.

   $ ps



```
    PID TTY       TIME COMMAND
 262148 ttyp0000  2:46 /bin/sh
 196614 ttyp0000  0:22 ./myprog
  65543 ttyp0000  0:13 /bin/grep
 196616 ttyp0000  2:07 /bin/ps
```

PID

This is a **process identification number**. PID is automatically assigned to each process when it is created on a Linux operating system

TTY

The name of the controlling terminal, if any. The *controlling terminal* is the workstation that started the process.

TIME

The amount of central processor time the process has used since it began running.

COMMAND

The name of the command or program that started the process. The display indicates which directory the command or program is found in.

9) System calls serve as the interface between an operating system and a process. Programs can interact with the operating system by making a system call. When a computer programme requests something from the kernel of the operating system, it performs a system call. The most common system calls used on Linux system calls are open, read, write, close.

Program to illustrate Open and Close system call

Prg1.c

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num;
   FILE *fptr;
   // use appropriate location if you are using MacOS or Linux
   fptr = fopen("program.txt","w");
   if(fptr == NULL)
   {
          printf("Error!");
          exit(1);
   }
   printf("Enter num: ");
   scanf("%d",&num);
   fprintf(fptr,"%d\n",num);
   fclose(fptr);
   return 0;
    }
```

Program to illustrate read/write system call

```
#include<unistd.h>
int main()
{
char b[30];
read(0,b,10);
write(1,b,10);
return 0;
}
```

10) SORT command is used to sort a file, arranging the records in a particular order.

```
$ sort filename.txt
```

11) The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern.

Syntax:
grep [options] pattern [files]

Options : **i :** Ignores, case for matching

```
$ grep -i "UNix" geekfile.txt
```

The -i option enables to search for a string case insensitively in the given file. It matches the words like “UNIX”, “Unix”, “unix”.

12) The awk command is used for manipulating data and generating reports.

By default Awk prints every line of data from the specified file.

```
$ awk '{print}' employee.txt
$ awk '/manager/ {print}' employee.txt
$ awk '{print $1,$4}' employee.txt
```

Conclusion :

Thus we have studied some basic Linux commands.

Experiment No 2

Aim : To write a C program to simulate basic Linux commands like ls, copy, rm etc.,

Theory:

Linux commands can be simulated in the high level language using Unix system calls and APIs available in the language. In addition, programming language construct can also be used to avail the Linux commands.

Exercises :

1. Siumulation of ls command

Algorithm :

1. Include necessary header files for manipulating directory.

2. Declare and initialize required objects.

3. Read the directory name form the user

4. Open the directory using opendir() system call and report error if the directory is

not available

5. Read the entry available in the directory

6. Display the directory entry (ie., name of the file or sub directory.

7. Repeat the step 6 and 7 until all the entries were read.

/* 1. Siumulation of ls command */

```
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
int main()
{
char dirname[10];
DIR*p;
struct dirent *d;
printf("Enter directory name\n");
scanf("%s",dirname);
p=opendir(dirname);
if(p==NULL)
{
perror("Cannot find directory1");
exit(-1);
}
while(d=readdir(p))
printf("%s\n",d->d_name);
return 0;
}
```

/* 2. Simulation of mv command */

```
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
int main(int argc, char **argv)
{
if(argc>3 || argc<3)
{
printf("Please Provide two arugments \n");
}
else{
int fd1,fd2;
int n,count=0;
if(access(argv[1],F_OK)<0)
{
printf("%s not found \n ",argv[1]);
}
if(rename(argv[1],argv[2])==0)
printf(" %s is movied or renamed to %s \n successfully
\n",argv[1],argv[2]);
return (0);
}
}
```

Conclusion:

Thus C program was written to simulate some Linux Commands

Experiment No 3

Aim : Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.

Theory:

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.

Both getppid() and getpid() are inbuilt functions defined in unistd.h library.
Getppid() : returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for init process.
getpid() : returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

/* 1. Fork System command */

```
include<iostream>
#include<unistd.h>
using namespace std;
int main()
{
int pid;
pid=fork();
if(pid==0)
{
cout<<"\nParent Process id: "<<getpid()<<endl;
cout<<"Child Process id: "<<getppid()<<endl;
cout<<"User of calling process: "<<getuid()<<endl;
cout<<"Effective User id of calling process: "<<geteuid()<<endl;
cout<<"Group id of calling process: "<<getgid()<<endl;
cout<<"Effective group id of calling process:"<<getegid()<<endl;
}
return 0;
}
```

```
Output:
Parent Process id: 9036
Child Process id: 1504
User of calling process: 1001
```

```
Effective User id of calling process: 1001
Group id of calling process: 1001
Effective Group id of calling process:1001
```

Experiment No 4

Aim : Write a program to demonstrate the concept of non-preemptive scheduling algorithms.

Theory:

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. The simplest non-preemptive scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the
CPU first is allocated the CPU first.

Algorithm for FCFS:

1. Start the program
2. Read the processes and its burst time
3. Schedule the process in FCFS order
4. Calculate the waiting time and turnaround time for each process
5. Display the scheduled process
6. Stop the program

/* 1. FCFS algorithm */

```
#include<iostream>
using namespace std;

// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
                          int bt[], int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int  i = 1; i < n ; i++ )
        wt[i] =  bt[i-1] + wt[i-1] ;
}

// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n,
                  int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int  i = 0; i < n ; i++)
```

```
        tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);

    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    //Display processes along with all details
    cout << "Processes  "<< " Burst time  "
         << " Waiting time  " << " Turn around time\n";

    // Calculate total waiting time and total turn
    // around time
    for (int  i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << "   " << i+1 << "\t\t" << bt[i] <<"\t    "
            << wt[i] <<"\t\t  " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}

// Driver code
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    //Burst time of all processes
    int  burst_time[] = {10, 5, 8};

    findavgTime(processes, n,  burst_time);
    return 0;
}
```

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. The SJF algorithm can be either preemptive or non-preemptive.

Algorithm:

1. Start the program
2. Read the process and the burst time
3. Schedule the process in the SJF order
4. Calculate the waiting time and turn around time for each processs.
5. Display the scheduled process
6. Stop the program

/* 2. SJF algorithm */

```
#include<stdio.h>
void main()
{
float aw,at;
int a[10],tp,tb,i,j,b[10],w[10],wt,tt,n,t[10];
printf("Enter the number of process\n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
a[i]=i;
printf("Enter the burst time for process %d\n",a[i]);
scanf("%d",&b[i]);
}
for(i=1;i<=n;i++)
{
for(j=i+1;j<=n;j++)
{
if(b[i]>b[j])
{
tb=b[i];
tp=a[i];
a[i]=a[j];
a[j]=tp;
b[i]=b[j];
b[j]=tb;}
tb=b[i];
tp=a[i];}}
wt=0;
tt=0;
t[0]=0;
printf("\nProcess no \t burst time \t waiting time\t turn arouond time");
for(i=1;i<=n;i++){
w[i]=t[i-1];
t[i]=w[i]+b[i];
wt=wt+w[i];
tt=tt+t[i];
printf("\n %d \t\t%d\t\t%d\t\t%d",a[i],b[i],w[i],t[i]);}
aw=wt/n;
at=tt/n;
printf("\n avg waiting time =%f\n,avg turn around time=%f \n",aw,at);}
```

OUTPUT:

```
Enter the number of process
4
Enter the burst time for process 1
3
Enter the burst time for process 2
6
Enter the burst time for process 3
4
Enter the burst time for process 4
2
 Process no        burst time        waiting time      turn around time
 4                 2                 0                 2
 1                 3                 2                 5
 3                 4                 5                 9
 2                 6                 9                 15
 avg waiting time =4.000000
,avg turn around time=7.000000
```

Conclusion :

Thus, we have implemented CPU scheduling algorithms successfully.

Experiment No 5

Aim : Write a C program to implement solution of Producer consumer problem through Semaphore.

Theory:

The Producer-Consumer problem is a classic synchronization problem in operating systems. The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process. The Producer process creates an item and adds it to the shared buffer. The Consumer process takes items out of the shared buffer and "consumes" them. Certain conditions must be met by the Producer and the Consumer processes to have consistent data synchronization:

1. The Producer process must not produce an item if the shared buffer is full.
2. The Consumer process must not consume an item if the shared buffer is empty.
3. Access to the shared buffer must be mutually exclusive; this means that at any given instance, only one process should be able to access the shared buffer and make changes to it.

The solution to the Producer-Consumer problem involves three *semaphore* variables.

- **semaphore Full**: Tracks the space filled by the Producer process. It is initialized with a value of 00 as the buffer will have 00 filled spaces at the beginning
- **semaphore Empty**: Tracks the empty space in the buffer. It is initially set to **buffer_size** as the whole buffer is empty at the beginning.
- **semaphore mutex**: Used for mutual exclusion so that only one process can access the shared buffer at a time.

Using the signal() and wait() operations on these semaphores, we can arrive at a solution.

Algorithm:

1. Start the program
2. Create three semaphore mutex empty full using segment system call.

3. Create shared memory address space using segment system call.
4. Produce an item, store an item in buffer.
5. Consume an item from buffer
6. Display the buffer
7. Stop the program

/* 1. Producer-Consumer Problem */

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

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;

    // Increase the number of full
    // slots by 1
    ++full;

    // Decrease the number of empty
    // slots by 1
    --empty;

    // Item produced
    x++;
    printf("\nProducer produces"
           "item %d",
           x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;
    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
           "item %d",
           x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}

// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can  give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
#pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
            // is full
            else {
                printf("Buffer is full!");
            }
            break;
        case 2:

            // If mutex is 1 and full
            // is non-zero, then it is
            // possible to consume
            if ((mutex == 1)
                && (full != 0)) {
                consumer();
            }

            // Otherwise, print Buffer
            // is empty
            else {
                printf("Buffer is empty!");
            }
            break;

        // Exit Condition
        case 3:
            exit(0);
            break;
        }
    }
}
```

Conclusion:

Thus C program to solve producer consumer is executed successfully.

Experiment No 6

Aim : Write a program to demonstrate the concept of deadlock avoidance through Banker"s Algorithm.

Theory:

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. Deadlock Avoidance is used by the Operating System to Avoid Deadlock in the System. The processes need to specify the maximum resources needed by the Operating System so that the Operating System can simulate the allocation of available resources to the requesting processes and check if it is possible to satisfy the need of all the processes' requirements.

/* 1. Deadlock Avoidance */

```
#include<stdio.h>

int main() {
  /* array will store at most 5 process with 3 resoures if your
process or
  resources is greater than 5 and 3 then increase the size of array
*/
  int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3],
safe[5], available[3], done[5], terminate = 0;
  printf("Enter the number of process and resources");
  scanf("%d %d", & p, & c);
  // p is process and c is diffrent resources
  printf("enter allocation of resource of all process %dx%d matrix",
p, c);
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      scanf("%d", & alc[i][j]);
    }
  }
  printf("enter the max resource process required %dx%d matrix", p,
c);
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      scanf("%d", & max[i][j]);
    }
  }
  printf("enter the  available resource");
  for (i = 0; i < c; i++)
    scanf("%d", & available[i]);

  printf("\n need resources matrix are\n");
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      need[i][j] = max[i][j] - alc[i][j];
      printf("%d\t", need[i][j]);
    }
    printf("\n");
  }
  /* once process execute variable done will stop them for again
execution */
  for (i = 0; i < p; i++) {
    done[i] = 0;
  }
  while (count < p) {
    for (i = 0; i < p; i++) {
      if (done[i] == 0) {
        for (j = 0; j < c; j++) {
          if (need[i][j] > available[j])
            break;
        }
        //when need matrix is not greater then available matrix then
if j==c will true
        if (j == c) {
          safe[count] = i;
          done[i] = 1;
          /* now process get execute release the resources and add
them in available resources */
          for (j = 0; j < c; j++) {
            available[j] += alc[i][j];
          }
          count++;
          terminate = 0;
        } else {
          terminate++;
        }
      }
    }
    if (terminate == (p - 1)) {
      printf("safe sequence does not exist");
      break;
    }

  }
  if (terminate != (p - 1)) {
    printf("\n available resource after completion\n");
    for (i = 0; i < c; i++) {
      printf("%d\t", available[i]);
    }
    printf("\n safe sequence are\n");
    for (i = 0; i < p; i++) {
      printf("p%d\t", safe[i]);
    }
  }

  return 0;
}
```

Conclusion:

Thus C program for deadlock avoidance is executed successfully.

Experiment No 7

Aim : Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.

Theory:

There are various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.

Best Fit: The Best Fit algorithm tries to find out the smallest hole possible in the list that can accommodate the size requirement of the process.

Using Best Fit has some disadvantages.

1. 1. It is slower because it scans the entire list every time and tries to find out the smallest hole which can satisfy the requirement the process.
2. Due to the fact that the difference between the whole size and the process size is very small, the holes produced will be as small as it cannot be used to load any process and therefore it remains useless. Despite of the fact that the name of the algorithm is best fit, It is not the best algorithm among all.

First Fit: First Fit algorithm scans the linked list and whenever it finds the first big enough hole to store a process, it stops scanning and load the process into that hole. This is the simplest to implement among all the algorithms and produces bigger holes as compare to the other algorithms.

Worst Fit: The worst fit algorithm scans the entire list every time and tries to find out the biggest hole in the list which can fulfill the requirement of the process. Despite of the fact that this algorithm produces the larger holes to load the other processes, this is not the better approach due to the fact that it is slower because it searches the entire list every time again and again.

/* 1. First Fit Placement Algorithm */

```
#include<bits/stdc++.h>
using namespace std;

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m,
              int processSize[], int n)
```

```
{
    // Stores block id of the
    // block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                break;
            }
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i+1 << "\t\t"
             << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

// Driver code
int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

Conclusion:

Thus C++ program for first fit placement algorithm is executed successfully.

Experiment No 8

Aim : Write a program to demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU etc

Theory:

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

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

Least Recently Used (LRU): In this algorithm, page will be replaced which is least recently used.

Optimal Page replacement: In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

/* 1. FIFO Replacement Algorithm */

```
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
    // To represent set of current pages. We use
    // an unordered_set so that we quickly check
    // if a page is present in set or not
    unordered_set<int> s;

    // To store the pages in FIFO manner
    queue<int> indexes;

    // Start from initial page
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        // Check if the set can hold more pages
        if (s.size() < capacity)
        {
            // Insert it into set if not present
            // already which represents page fault
            if (s.find(pages[i])==s.end())
            {
                // Insert the current page into the set
                s.insert(pages[i]);
                // increment page fault
                page_faults++;

                // Push the current page into the queue
                indexes.push(pages[i]);
            }
        }

        // If the set is full then need to perform FIFO
        // i.e. remove the first page of the queue from
        // set and queue both and insert the current page
        else
        {
            // Check if current page is not already
            // present in the set
            if (s.find(pages[i]) == s.end())
            {
                // Store the first page in the
                // queue to be used to find and
                // erase the page from the set
                int val = indexes.front();

                // Pop the first page from the queue
                indexes.pop();

                // Remove the indexes page from the set
                s.erase(val);

                // insert the current page in the set
                s.insert(pages[i]);

                // push the current page into
                // the queue
                indexes.push(pages[i]);

                // Increment page faults
                page_faults++;
            }
        }
    }

    return page_faults;
}

// Driver code
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                   2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

Conclusion:

Thus C++ program for FIFO replacement algorithm is executed successfully.

Experiment No 9

Aim : Write a program in C to do disk scheduling - FCFS, SCAN, C-SCAN

Theory:

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O Scheduling.

First-in, first-out (FIFO): The FIFO scheduling process the requests sequentially. The request arrived first is served first. It is fair to all processes and approaches random scheduling in performance if there are many processes.

Shortest Service Time First (SSTF): The SSTF scheduling selects the disk I/O request that requires the least movement of the disk arm from its current position.

SCAN: In SCAN scheduling arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction. Then direction is reversed

C-SCAN: C-SCAN scheduling restricts scanning to one direction only when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.

/* 1. FIFO Disk Scheduling Algorithm */

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 50, 89, 52, 61, 87, 25

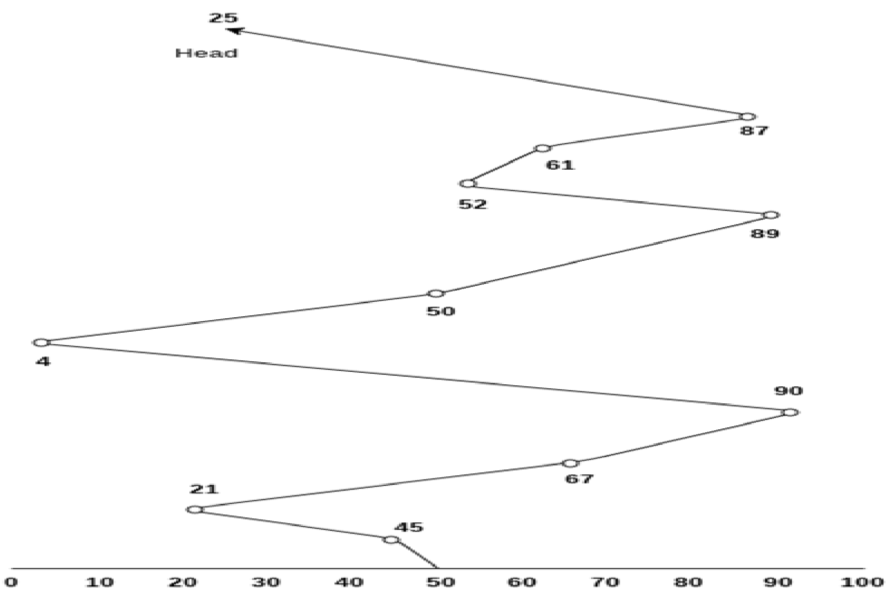
The head pointer starts at 50 and moves in left direction. Find the number of head movements in cylinders using FIFO scheduling.

Number of cylinders moved by the head

=(50-45)+(45-21)+(67-21)+(90-67)+(90-4)+(50-4)+(89-50)+(61-52)+(87-61)+(87-25)

= 5 + 24 + 46 + 23 + 86 + 46 + 49 + 9 + 26 + 62

= 376



```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i,n,req[50],mov=0,cp;
    printf("enter the current position\n");
    scanf("%d",&cp);
    printf("enter the number of requests\n");
    scanf("%d",&n);
    printf("enter the request order\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&req[i]);
    }
    mov=mov+abs(cp-req[0]);  // abs is used to calculate the absolute
value
    printf("%d -> %d",cp,req[0]);
    for(i=1;i<n;i++)
    {
        mov=mov+abs(req[i]-req[i-1]);
        printf(" -> %d",req[i]);
    }
    printf("\n");
    printf("total head movement = %d\n",mov);
}
```

/* 2. SCAN Disk Scheduling Algorithm */

```
#include<conio.h>
#include<stdio.h>
int main()
{
 int i,j,sum=0,n;
 int d[20];
 int disk;   //loc of head
 int temp,max;
 int dloc;   //loc of disk in array
 clrscr();
 printf("enter number of location\t");
 scanf("%d",&n);
 printf("enter position of head\t");
 scanf("%d",&disk);
 printf("enter elements of disk queue\n");
 for(i=0;i<n;i++)
 {
 scanf("%d",&d[i]);
 }
 d[n]=disk;
 n=n+1;
 for(i=0;i<n;i++)    // sorting disk locations
 {
  for(j=i;j<n;j++)
  {
    if(d[i]>d[j])
    {
    temp=d[i];
    d[i]=d[j];
    d[j]=temp;
    }
  }
 }
 max=d[n];
 for(i=0;i<n;i++)   // to find loc of disc in array
 {
 if(disk==d[i]) { dloc=i; break;  }
 }
 for(i=dloc;i>=0;i--)
 {
 printf("%d -->",d[i]);
 }
 printf("0 -->");
 for(i=dloc+1;i<n;i++)
 {
 printf("%d-->",d[i]);
 }
 sum=disk+max;
       printf("\nmovement of total cylinders %d",sum);
 getch();
 return 0;
}
```

Conclusion:

Thus C program for FIFO and SCAN disk scheduling is executed successfully.

Experiment No 10

Aim : Display current shell, home directory, operating system type, current path setting, current working directory.

Theory:

The shell is the Linux command line interpreter. It provides an interface between the user and the kernel and executes programs called commands.

The $HOME is a shell environmental variable containing a full path to user directory. The $HOME variable is set automatically by the system upon its installation and is usually set to /home/username . However, it is possible to set the $HOME variable to any custom path as required.

The uname (UNIX name) command in Linux is a simple yet powerful tool that offers information about a Linux machine's operating system and hardware platform.

It instructs the shell where to look for these executable files and specifies a list of directories that house various executables on the system

/* 1. Display shell, Home directory, Operating system type, Current Path setting and Current working directory*/

```
echo “Your current shell”

echo $SHELL

echo "Your home directory"

echo $HOME

echo "Your operating system type"

uname

echo "Your current path setting"

echo $PATH

echo "Your current working directory"

pwd
```

Conclusion:

Thus all Linux commands are executed successfully.