****

**Tribhuvan University**

**Faculty of Humanities and Social Science**

**Operating System**

**A LAB REPORT**

**Submitted To**

**Department of Computer Application**

**Shahid Smarak College**

*In partial fulfillment of the requirements in the Bachelors in Computer Application*

Submitted By :

Amir Maharjan

TU Regd.No :

Exam Roll No :

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Internal Supervisor**

**Hari Lal Chalise**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**External Supervisor**

**Kirtipur Kathmandu**

# BCA 4th semester Operating system Lab Work

1. Write a program to create process by implementing fork (), wait (), and exit () system call.
2. Write a “program to create multi-threaded process.”
3. Write C programs to simulate solutions to Classical Process Synchronization Problems: Producer-Consumer Problem.
4. Write C programs to simulate CPU scheduling algorithms: FCFS, SJF, and Robin. Also find the average turn around time and waiting time for each of the algorithm. (Quantum time (in ms): 5)
5. Write a program to simulate the following contiguous memory allocation technique. a) Worst-fit b) Best-fit c) First- fit
6. Write a program to simulate Page replacement Algorithm: FIFO, LRU & optimal
7. Write a program to simulate Disk Scheduling Algorithms: FCFS, SCAN, SSTF.

LIST OF EXPERIMENTS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.No.** | **Name of the Experiment** | **Date of Experiment** | **Date of Submission** | **Page No** | **Faculty Signature** |
| 1 | Write a program to create process by implementing fork (), wait (), and exit () system call. | 2023-07-23 |  | 1 - 2 |  |
| 2 | Write a “program to create multi-threaded process.” | 2023-07-30 |  | 3 – 3 |  |
| 3 | Write C programs to simulate solutions to Classical Process Synchronization Problems: Producer-Consumer Problem. | 2023-08-03 |  | 4 - 5 |  |
| 4 | Write C programs to simulate CPU scheduling algorithms: FCFS, SJF, and Robin. Also find the average turn around time and waiting time for each of the algorithm. (Quantum time (in ms): 5) | 2023-08-10 |  | 6 - 9 |  |
| 5 | Write a program to simulate the following contiguous memory allocation technique. a) Worst-fit b) Best-fit c) First- fit | 2023-08-16 |  | 10 – 12 |  |
| 6 | Write a program to simulate Page replacement Algorithm: FIFO, LRU & optimal | 2023-08-20 |  | 13 – 15 |  |
| 7 | Write a program to simulate Disk Scheduling Algorithms: FCFS, SCAN, SSTF. | 2023-08-22 |  | 16 - 18 |  |

# Lab No 1. / Experiment 1

**AIM:** To write the program to implement fork () system call.

# DESCRIPTION:

Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t child\_pid;

child\_pid = fork();

if (child\_pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (child\_pid == 0) {

printf("Child process with PID: %d\n", getpid());

exit(EXIT\_SUCCESS);

} else {

printf("Parent process with PID: %d\n", getpid());

printf("Child process created with PID: %d\n", child\_pid);

int status;

wait(&status);

if (WIFEXITED(status)) {

printf("Child process exited with status: %d\n", WEXITSTATUS(status));

} else {

printf("Child process did not exit normally.\n");

}

}

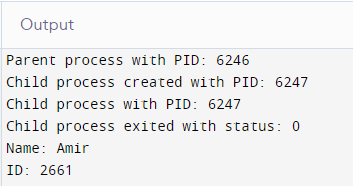
printf("Name: Amir\n");

printf("ID: 2661");

return 0;

}

**OUTPUT:**



**RESULT**: Thus, the program was executed and verified successfully.

# Lab No 2. / Experiment 2

**AIM:** To create a multithreaded process

# DESCRIPTION:

Used to create one or more than one process. Print the name of the thread. It also show the difference before the thread and after the thread.

**PROGRAM:**

include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

void \*myThreadFun(void \*vargp)

{

sleep(1);

printf("Printing Name from Thread \n");

return NULL;

}

int main()

{

pthread\_t thread\_id;

printf("Before Thread\n");

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

pthread\_join(thread\_id, NULL);

printf("After Thread\n");

exit(0);

}

**OUTPUT:**

Undefined reference pthread\_create in main();

**RESULT**: Thus, the program was not executed and could not be verified successfully.

# Lab No 3. / Experiment 3

**AIM:** To simulate solutions to Classical Process Synchronization Problems: Producer-Consumer Problem.

# DESCRIPTION:

If the buffer has some resources, then the consumer can consume the resources available in the buffer. If the buffer becomes completely empty the producer is called and does its job of filling the buffer with resources.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;

int full = 0;

int empty = 10, x = 0;

void producer() {

--mutex;

++full;

--empty;

x++;

printf( "\nProducer produces item %d", x );

++mutex;

}

void consumer() {

--mutex;

--full;

++empty;

printf( "\nConsumer consumes item %d", x );

x--;

++mutex;

}

int main() {

int n, i;

printf( "\n1. Press 1 for producer \n2. Press 2 for consumer \n3. Press 3 for Exit" );

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

printf( "\nEnter 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);

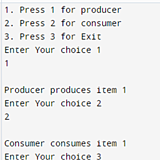
break;

}

}

}

**OUTPUT:**



**RESULT**: Thus, the program was executed and verified successfully.

# Lab No 4. / Experiment 4

**AIM:** To simulate CPU scheduling algorithms: FCFS, SJF, and Robin. Also find the average turnaround time and waiting time for each of the algorithm. (Quantum time (in ms): 5)

DESCRIPTION:

In **FCFS**, the process that arrives first gets executed first. Hence the name First Come First served.

In **SJF**, the process that has the shortest burst time gets executed first after its arrival from the queue.

In **Round Robin**, burst time and arrival time has nothing to with process execution. The process with the highest or lowest priority gets executed first.

**PROGRAM:**

**FCFS**

#include<stdio.h>

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

wt[0] = 0;

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

}

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

for (int i = 0; i < n ; i++)

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

}

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst time Waiting time Turn around time\n");

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

printf(" %d ",(i+1));

printf(" %d ", bt[i] );

printf(" %d",wt[i] );

printf(" %d\n",tat[i] );

}

float s=(float)total\_wt / (float)n;

float t=(float)total\_tat / (float)n;

printf("Average waiting time = %f",s);

printf("\n");

printf("Average turn around time = %f ",t);

}

int main()

{

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}

**SJF**

#include <stdio.h>

int main()

{

Average

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg\_wt, avg\_tat;

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

Process Id.

for (i = 0; i < n; i++) {

printf("P%d: ", i + 1);

scanf("%d", &A[i][1]);

A[i][0] = i + 1;

}

Burst Time.

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];

total += A[i][2];

}

avg\_wt = (float)total / n;

total = 0;

printf("P BT WT TAT\n");

for (i = 0; i < n; i++) {

A[i][3] = A[i][1] + A[i][2];

total += A[i][3];

printf("P%d%d%d%d\n", A[i][0],A[i][1], A[i][2], A[i][3]);

}

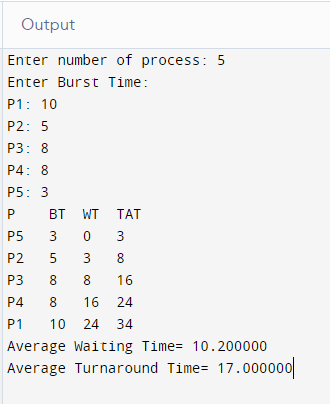
avg\_tat = (float)total / n;

printf("Average Waiting Time= %f", avg\_wt);

printf("\nAverage Turnaround Time= %f", avg\_tat);

}

**Output:**



**Round Robin**

#include<stdio.h>

int main()

{

int cnt,j,n,t,remain,flag=0,tq;

int wt=0,tat=0,at[10],bt[10],rt[10];

printf("Enter Total Process:\t ");

scanf("%d",&n);

remain=n;

for(cnt=0;cnt<n;cnt++)

{

printf("Enter Arrival Time and Burst Time for Process Process Number %d :",cnt+1);

scanf("%d",&at[cnt]);

scanf("%d",&bt[cnt]);

rt[cnt]=bt[cnt];

}

printf("Enter Time Quantum:\t");

scanf("%d",&tq);

printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");

for(t=0,cnt=0;remain!=0;)

{

if(rt[cnt]<=tq && rt[cnt]>0)

{

t+=rt[cnt];

rt[cnt]=0;

flag=1;

}

else if(rt[cnt]>0)

{

rt[cnt]-=tq;

t+=tq;

}

if(rt[cnt]==0 && flag==1)

{

remain--;

printf("P[%d]\t|\t%d\t|\t%d\n",cnt+1,t-at[cnt],t-at[cnt]-bt[cnt]);

wt+=t-at[cnt]-bt[cnt];

tat+=t-at[cnt];

flag=0;

}

if(cnt==n-1)

cnt=0;

else if(at[cnt+1]<=t)

cnt++;

else

cnt=0;

}

printf("\nAverage Waiting Time= %f\n",wt\*1.0/n);

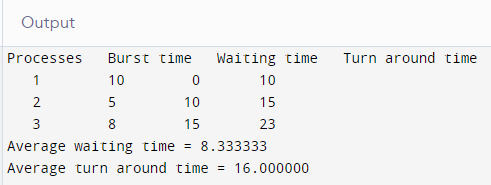
printf("Avg Turnaround Time = %f",tat\*1.0/n);

return 0;

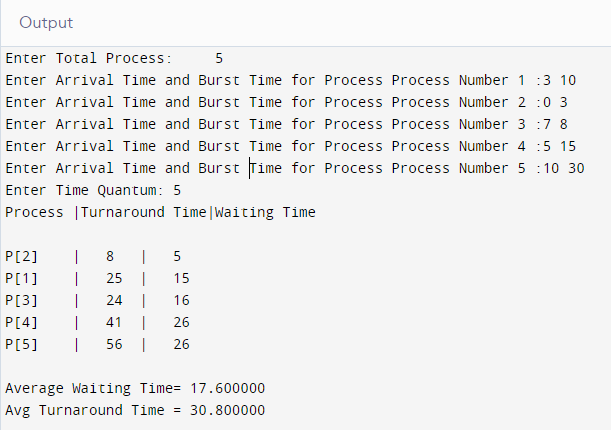
}

**OUTPUT:**

**FCFS**



**Round Robin**

****

**RESULT**: Thus, the program was executed and verified successfully.

# Lab No 5. / Experiment 5

**AIM:** To simulate the following contiguous memory allocation technique.

a) Worst-fit b) Best-fit c) First- fit

DESCRIPTION:

In **Worst fit**, if a process finds a memory space that can hold it’s size. The process is allocated in that memory space without even considered other memory spaces.

In **Best fit**, the memory space that fit best with the process size is allocated to the process.

In **First fit**, the first memory space that meets the required process size is allotted the process.

**PROGRAM:**

**For BEST FIT, FIRST FIT and WORST FIT**

#include <stdio.h>

#define MEMORY\_SIZE 1000

#define NUM\_BLOCKS 5

struct MemoryBlock {

int id;

int size;

int allocated;

};

void initializeMemory(struct MemoryBlock memory[], int numBlocks) {

for (int i = 0; i < numBlocks; i++) {

memory[i].id = -1;

memory[i].size = MEMORY\_SIZE / NUM\_BLOCKS;

memory[i].allocated = 0;

}

}

void printMemory(struct MemoryBlock memory[], int numBlocks) {

printf("Memory Block\tSize\tStatus\t\tProcess ID\n");

for (int i = 0; i < numBlocks; i++) {

printf("%d\t\t%d\t%s\t\t", i + 1, memory[i].size, (memory[i].allocated) ? "Allocated" : "Free");

if (memory[i].allocated) {

printf("%d", memory[i].id);

} else {

printf("-");

}

printf("\n");

}

}

void worstFit(struct MemoryBlock memory[], int numBlocks, int processId, int processSize) {

int worstFitIndex = -1;

int worstFitSize = -1;

for (int i = 0; i < numBlocks; i++) {

if (!memory[i].allocated && memory[i].size >= processSize) {

if (worstFitIndex == -1 || memory[i].size > worstFitSize) {

worstFitIndex = i;

worstFitSize = memory[i].size;

}

}

}

if (worstFitIndex != -1) {

memory[worstFitIndex].id = processId;

memory[worstFitIndex].allocated = 1;

memory[worstFitIndex].size = processSize;

printf("Process %d allocated to memory block %d (Worst-fit)\n", processId, worstFitIndex + 1);

} else {

printf("Process %d cannot be allocated (Worst-fit)\n", processId);

}

}

void bestFit(struct MemoryBlock memory[], int numBlocks, int processId, int processSize) {

int bestFitIndex = -1;

int bestFitSize = MEMORY\_SIZE + 1;

for (int i = 0; i < numBlocks; i++) {

if (!memory[i].allocated && memory[i].size >= processSize) {

if (memory[i].size < bestFitSize) {

bestFitIndex = i;

bestFitSize = memory[i].size;

}

}

}

if (bestFitIndex != -1) {

memory[bestFitIndex].id = processId;

memory[bestFitIndex].allocated = 1;

memory[bestFitIndex].size = processSize;

printf("Process %d allocated to memory block %d (Best-fit)\n", processId, bestFitIndex + 1);

} else {

printf("Process %d cannot be allocated (Best-fit)\n", processId);

}

}

void firstFit(struct MemoryBlock memory[], int numBlocks, int processId, int processSize) {

for (int i = 0; i < numBlocks; i++) {

if (!memory[i].allocated && memory[i].size >= processSize) {

memory[i].id = processId;

memory[i].allocated = 1;

memory[i].size = processSize;

printf("Process %d allocated to memory block %d (First-fit)\n", processId, i + 1);

return;

}

}

printf("Process %d cannot be allocated (First-fit)\n", processId);

}

int main() {

struct MemoryBlock memory[NUM\_BLOCKS];

initializeMemory(memory, NUM\_BLOCKS);

int numProcesses;

printf("Enter the number of processes: ");

scanf("%d", &numProcesses);

for (int i = 1; i <= numProcesses; i++) {

int processSize;

printf("Enter the size of process %d: ", i);

scanf("%d", &processSize);

worstFit(memory, NUM\_BLOCKS, i, processSize);

bestFit(memory, NUM\_BLOCKS, i, processSize);

firstFit(memory, NUM\_BLOCKS, i, processSize);

printMemory(memory, NUM\_BLOCKS);

printf("\n");

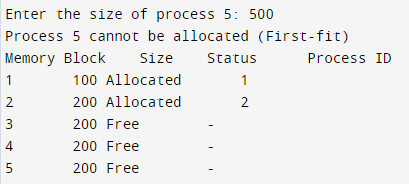
}

return 0;

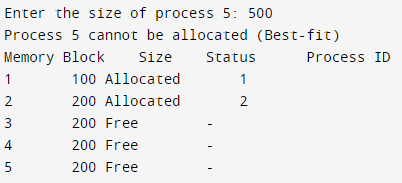
}

**OUTPUT:**

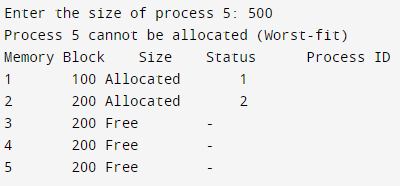
**First-Fit**

****

**Best-Fit**

****

**Worst-Fit**

****

**RESULT**: Thus, the program was executed and verified successfully.

# Lab No 6. / Experiment 6

**AIM:** To simulate Page replacement Algorithm: FIFO, LRU & optimal

# DESCRIPTION:

In FIFO, the process that came in first also gets kicked out first or executed first.

In LRU, the page reference that is used the least in the sequence is replaced.

In Optimal, the page

**PROGRAM:**

**FIFO**

#include <stdio.h>

int main() {

int n, frames;

printf("Enter the number of frames: ");

scanf("%d", &frames);

printf("Enter the number of pages: ");

scanf("%d", &n);

int pages[n];

printf("Enter the page reference sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

int fifo[frames];

int front = 0, rear = 0, page\_faults = 0;

for (int i = 0; i < frames; i++) {

fifo[i] = -1;

}

for (int i = 0; i < n; i++) {

int page = pages[i];

int found = 0;

for (int j = 0; j < frames; j++) {

if (fifo[j] == page) {

found = 1;

break;

}

}

if (!found) {

fifo[rear] = page;

rear = (rear + 1) % frames;

page\_faults++;

}

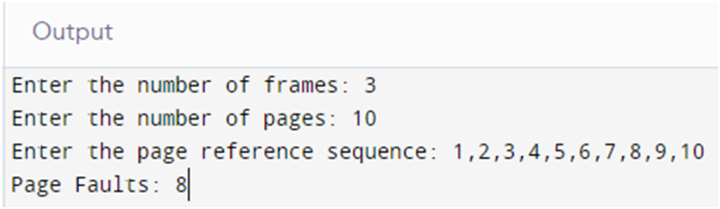
}

printf("Page Faults: %d\n", page\_faults);

return 0;

}

**Output**



**LRU**

#include <stdio.h>

int main() {

int n, frames;

printf("Enter the number of frames: ");

scanf("%d", &frames);

printf("Enter the number of pages: ");

scanf("%d", &n);

int pages[n];

printf("Enter the page reference sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

int lru[frames];

int page\_faults = 0;

for (int i = 0; i < frames; i++) {

lru[i] = -1;

}

for (int i = 0; i < n; i++) {

int page = pages[i];

int found = 0;

int empty\_slot = -1;

for (int j = 0; j < frames; j++) {

if (lru[j] == page) {

found = 1;

break;

}

if (lru[j] == -1) {

empty\_slot = j;

}

}

if (!found) {

if (empty\_slot != -1) {

lru[empty\_slot] = page;

} else {

int min\_index = 0;

for (int j = 1; j < frames; j++) {

if (lru[j] < lru[min\_index]) {

min\_index = j;

}

}

lru[min\_index] = page;

}

page\_faults++;

}

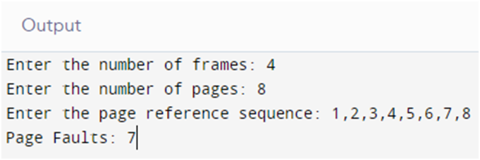
}

printf("Page Faults: %d\n", page\_faults);

return 0;

}

**Output**

****

**Optional replacement Algorithm**

#include <stdio.h>

#include <limits.h>

int main() {

int n, frames;

printf("Enter the number of frames: ");

scanf("%d", &frames);

printf("Enter the number of pages: ");

scanf("%d", &n);

int pages[n];

printf("Enter the page reference sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

int optimal[frames];

int page\_faults = 0;

for (int i = 0; i < frames; i++) {

optimal[i] = -1;

}

for (int i = 0; i < n; i++) {

int page = pages[i];

int found = 0;

int empty\_slot = -1;

for (int j = 0; j < frames; j++) {

if (optimal[j] == page) {

found = 1;

break;

}

if (optimal[j] == -1) {

empty\_slot = j;

break;

}

}

if (!found) {

if (empty\_slot != -1) {

optimal[empty\_slot] = page;

} else {

int farthest\_index = -1;

int farthest\_distance = INT\_MIN;

for (int j = 0; j < frames; j++) {

int future\_index = -1;

for (int k = i + 1; k < n; k++) {

if (optimal[j] == pages[k]) {

future\_index = k;

break;

}

}

if (future\_index == -1) {

farthest\_index = j;

break;

} else if (future\_index > farthest\_distance) {

farthest\_distance = future\_index;

farthest\_index = j;

}

}

optimal[farthest\_index] = page;

}

page\_faults++;

}

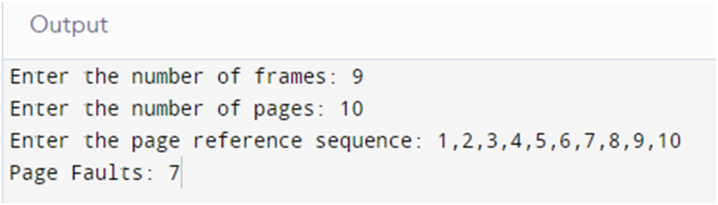
}

printf("Page Faults: %d\n", page\_faults);

return 0;

}

**OUTPUT:**



**RESULT**: Thus, the program was executed and verified successfully.

# Lab No 7. / Experiment 7

**AIM:** To simulate Disk Scheduling Algorithms: FCFS, SCAN, SSTF

# DESCRIPTION:

In **FCFS**, the read-write (R/W) head tracks the elements in a sequence of first to last.

In **SCAN**, a direction is given and the elements that in the given direction are executed first. After all the elements in one direction is over, heads to zero node and continues.

In **SSTF**, a direction is meaningless. The element that has the lowest number gets executed first.

**PROGRAM:**

**FCFS**

#include <stdio.h>

#include <math.h>

int size = 8;

void FCFS(int arr[],int head)

{

int seek\_count = 0;

int cur\_track, distance;

for(int i=0;i<size;i++)

{

cur\_track = arr[i];

distance=fabs(head-cur\_track);

seek\_count += distance;

head = cur\_track;

}

printf("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]);

}

}

int main()

{

int arr[8] = { 176, 79, 34, 60, 92, 11, 41, 114 };

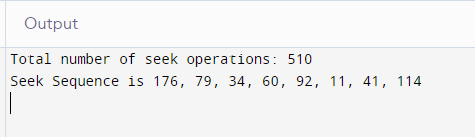
int head = 50;

FCFS(arr,head);

return 0;

}

**Output**

****

**SCAN**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define size 10

#define disk\_size 200

int comp(const void \* l, const void \* n) {

return (\*(int\*)l - \*(int\*)n);

}

void SCAN(int arr[], int head, char\* dn){

int seek\_num = 0;

int dt, cur\_track;

int leftside[size], rightside[size];

int seek\_seq[size + 3];

int m\_scan = 0, s\_scan = 0;

if { (strcmp(dn, "leftside") == 0)

leftside[m\_scan++] = 0;

}else if {(strcmp(dn, "rightside") == 0)

rightside[s\_scan++] = disk\_size - 1;}

for (int p\_s = 0; p\_s < size; p\_s++) {

if (arr[p\_s] < head)

leftside[m\_scan++] = arr[p\_s];

if (arr[p\_s] > head)

rightside[s\_scan++] = arr[p\_s];

}

qsort(leftside, m\_scan, sizeof(int), comp);

qsort(rightside, s\_scan, sizeof(int), comp);

int go = 2;

int ind = 0;

while (go--) {

if (strcmp(dn, "leftside") == 0) {

for (int p\_s = m\_scan - 1; p\_s >= 0; p\_s--) {

cur\_track = leftside[p\_s];

seek\_seq[ind++] = cur\_track;

dt = abs(cur\_track - head);

seek\_num += dt;

head = cur\_track;

}

dn = "rightside";

}

else if (strcmp(dn, "rightside") == 0) {

for (int p\_s = 0; p\_s < s\_scan; p\_s++) {

cur\_track = rightside[p\_s];

seek\_seq[ind++] = cur\_track;

dt = abs(cur\_track - head);

seek\_num += dt;

head = cur\_track;

}

dn = "leftside";

}

}

printf("Num of seek process = %d", seek\_num);

printf("Sequence is");

for (int p\_s = 0; p\_s < ind; p\_s++) {

printf("%d", seek\_seq[p\_s]);

}

}

int main(){

int arr[size] = { 126, 90, 14, 50, 25, 42, 51, 78, 102, 100 };

int head = 42;

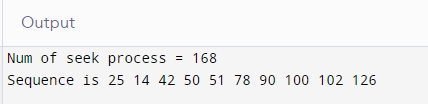
char dn[] = "leftside";

SCAN(arr, head, dn);

return 0;

}

**Output:**



**SSTF:**

#include<stdio.h>

#include<math.h>

#include<stdlib.h>

void main()

{

int queue[100],t[100],head,seek=0,n,i,j,temp;

float avg;

// clrscr();

printf("\*\*\* SSTF Disk Scheduling Algorithm \*\*\*\n");

printf("Enter the size of Queue\t");

scanf("%d",&n);

printf("Enter the Queue\t");

for(i=0;i<n;i++)

{

scanf("%d",&queue[i]);

}

printf("Enter the initial head position\t");

scanf("%d",&head);

for(i=1;i<n;i++)

t[i]= abs(head-queue[i]);

for(i=0;i<n;i++)

{

for(j=i+1;j<n;j++)

{

if(t[i]>t[j])

{

temp=t[i];

t[i]=t[j];

t[j]=temp;

temp=queue[i];

queue[i]=queue[j];

queue[j]=temp;

}

}

}

for(i=1;i<n-1;i++)

{

seek=seek+abs(head-queue[i]);

head=queue[i];

}

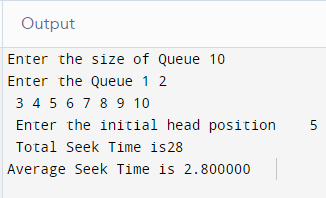
printf("\nTotal Seek Time is%d\t",seek);

avg=seek/(float)n;

printf("\nAverage Seek Time is %f\t",avg);

}

**Output:**

****

**RESULT**: Thus, the program was executed and verified successfully.