**Institute Of Science and Technology**

**Tribhuwan University**

**Shape, arrow

Description automatically generated**

**Lab Report**

**On**

**Operating Systems (CSC 259)**

**Submitted to:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Department Of Computer Science And Information Teachnology**

**Prithivi Narayan Campus , Pokhara**

**Submitted by :**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Roll No: \*\***

***Third* Batch**

**Lab Index**

**Name: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Roll. No: \*\***

**Level: Bachelor Year/Sem: Fourth**

**Course Title: Operating Systems Course No.: CSC 259**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Title** | **Submitted On:** | **Remarks** |
| **1.** | **FCFS disk scheduling algorithm** |  |  |
| **2.** | **SSTF disk scheduling algorithm** |  |  |
| **3.** | **FIFO page replacement algorithm** |  |  |
| **4.** | **LRU page replacement algorithm** |  |  |
| **5.** | **FCFS process scheduling algorithm** |  |  |
| **6.** | **SJF process scheduling algorithm** |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**LAB NO. 1**

**TITLE: FCFS DISK SCHEDULING ALGORITHM**

1. **THEORY**

This is a non-preemptive disk scheduling algorithm. This is the simplest scheduling algorithm. Jobs are scheduled in the order they were received in disk queue. Implementation of this technique is easy and based on FIFO queue. It has poor performance as average waiting time is high. This policy aims to minimize response time with little regard for throughput.

FEATURES:

* Perform operations in order requested
* No reordering of work queue
* No starvation: every request is serviced
* Poor performance

1. **SOURCE CODE**

*#include<stdio.h>*

*#include<stdlib.h>*

*int main()*

*{*

*int queue[20],n,head,i,j,k,seek=0,max,diff;*

*float avg;*

*printf("Enter the max range of disk \n");*

*scanf("%d",&max);*

*printf("Enter the size of queue request \n");*

*scanf("%d",&n);*

*printf("Enter the queue of disk positions to be read\n");*

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

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

*printf("Enter the initial head position \n");*

*scanf("%d",&head);*

*queue[0]=head;*

*for(j=0;j<=n-1;j++)*

*{*

*diff= abs(queue[j+1]-queue[j]);*

*seek+=diff;*

*printf("Diskhead moves from %d to %d with seek %d\n", queue[j], queue[j+1], diff);*

*}*

*printf("Total seek time is : %d\n",seek);*

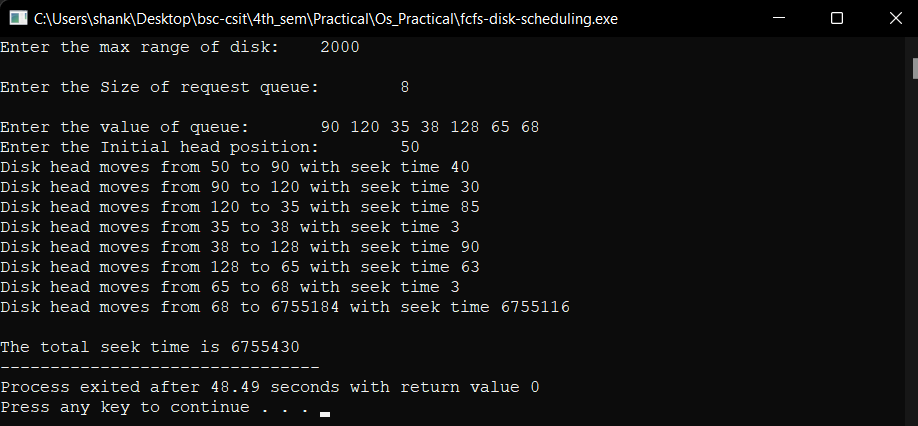
*avg=seek/(float)n;*

*printf("Average seek time is %f \n",avg);*

*return 0;*

*}*

1. **OUTPUT**



**LAB NO. 2**

**TITLE: SSTF DISK SCHEDULING ALGORITHM**

1. **THEORY**

In SSTF, requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it.

1. **SOURCE CODE**

*#include<stdio.h>*

*#include<conio.h>*

*struct di{*

*int num;*

*int flag;*

*};*

*int main()*

*{*

*int i,j,sum=0,n,min,loc,x,y;*

*struct di d[20];*

*int disk;*

*int ar[20],a[20];*

*printf("Enter the number of location \t");*

*scanf("%d",&n);*

*printf("Enter the position of head\t");*

*scanf("%d",&disk);*

*printf("Enter elements of disk queue\n");*

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

*{*

*scanf("%d",&d[i].num);d[i].flag=0;*

*}*

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

*{*

*x=0;min=0;loc=0;*

*for(j=0;j<n;j++){*

*if(d[j].flag==0){*

*if(x==0)*

*{*

*ar[j]= disk-d[j].num;*

*if(ar[j]<0){*

*ar[j]=d[j].num-disk;*

*}*

*min= ar[j];*

*loc=j;*

*x++;*

*}*

*else{*

*ar[j]=disk-d[j].num;*

*if(ar[j]<0){*

*ar[j]=d[j].num-disk;*

*}*

*}*

*if(min>ar[j]){ min=ar[j];loc=j; }*

*}*

*}*

*d[loc].flag=1;*

*a[i]=d[loc].num-disk;*

*if(a[i]<0){a[i]=disk-d[loc].num;}*

*disk=d[loc].num;*

*}*

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

*sum=sum+a[i];*

*}*

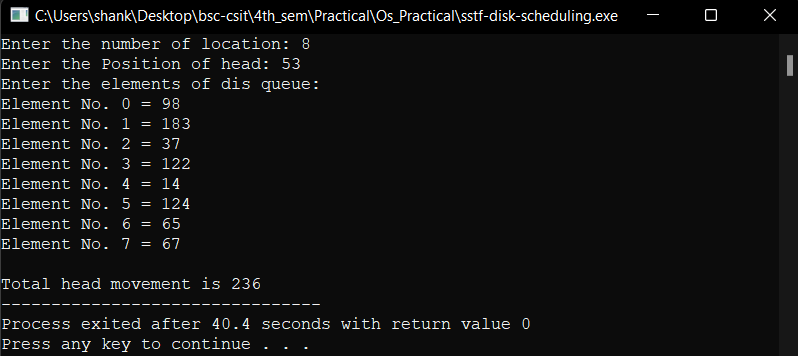
*printf("\n Movement of total cylinders %d",sum);*

*getch();*

*return 0;*

*}*

1. **OUTPUT**



**LAB NO. 3**

**TITLE: FIFO PAGE REPLACEMENT ALGORITHM**

1. **THEORY**

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all the pages in the memory in a queue; 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. It is likely to replace heavily or constantly used pages and they are still needed for further processing. It sometimes suffers from Belady’s anomaly.

1. **SOURCE CODE**

*#include<stdio.h>*

*int main()*

*{*

*int refString[10],pagefaults=0,m,n,s,pages,frames;*

*printf("\n Enter the number of pages: \t");*

*scanf("%d",&pages);*

*printf("\n Enter reference string values:\n");*

*for(m = 0; m < pages; m++)*

*{*

*printf("Value No.[%d]:\t",m+1);*

*scanf("%d",&refString[m]);*

*}*

*printf("\n Enter the total number of frames:\t");*

*{*

*scanf("%d",&frames);*

*}*

*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");*

*for(n=0; n<frames;n++)*

*{*

*printf("%d\t",temp[n]);*

*}*

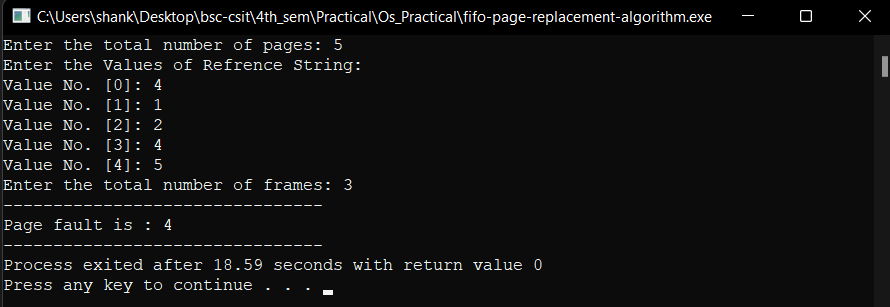
*}*

*printf("\n TotalPage Faults:\t%d\n",pagefaults);*

*return 0;*

*}*

1. **OUTPUT:**

****

**LAB NO. 4**

**TITLE: LRU PAGE REPLACEMENT ALGORITHM**

1. **THEORY**

In this algorithm, the page that has not been used for the longest period has to be replaced. If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period. This approach is the least recently used (LRU) algorithm. LRU replacement associates with each page the time of that page’s last use. When a page must be replaced, LRU chooses that page that has not been used recently. This strategy is the optimal page replacement algorithm looking backward in time, rather than forward.

1. **SOURCE CODE**

*/\**

*Least Recently Used Page Replacement Algorithm*

*\*/*

*#include <stdio.h>*

*int main(){*

*int frames[30], temp[30], pages[30];*

*int total\_pages, m, n, position, k, l, total\_frames;*

*int a = 0;*

*int b = 0;*

*int page\_fault = 0;*

*printf("Enter the total number of Frames: ");*

*scanf("%d", &total\_frames);*

*for(m = 0; m < total\_frames; m++) {*

*frames[m] = -1;*

*}*

*printf("Enter the total number of Pages: ");*

*scanf("%d", &total\_pages);*

*printf("Enter the value of refrence string:\n");*

*for(m = 0; m < total\_pages; m++) {*

*printf("Value No [%d]:\t", m+1);*

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

*}*

*for(n = 0; n < total\_pages; n++) {*

*a = 0;*

*b = 0;*

*for(m = 0; m < total\_frames; m++) {*

*if(frames[m] == pages[n]) {*

*a = 1;*

*b = 1;*

*break;*

*}*

*}*

*if(a == 0) {*

*for(m = 0; m < total\_frames; m++) {*

*if(frames[m] == -1) {*

*frames[m] = pages[n];*

*b = 1;*

*break;*

*}*

*}*

*}*

*if(b == 0) {*

*for(m = 0; m < total\_frames; m++) {*

*temp[m] = 0;*

*}*

*for(k = n-1, l = 1; l < total\_frames - 1; l++, k--) {*

*for(m = 0; m < total\_frames; m++) {*

*if(frames[m] == pages[k]) {*

*temp[m] = 1;*

*}*

*}*

*}*

*for(m = 0; m < total\_frames; m++) {*

*if(temp[m] == 0) {*

*position = m;*

*}*

*}*

*frames[position] = pages[n];*

*page\_fault++;*

*}*

*printf("\n");*

*for(m = 0; m < total\_frames; m++) {*

*printf("%d\t", frames[m]);*

*}*

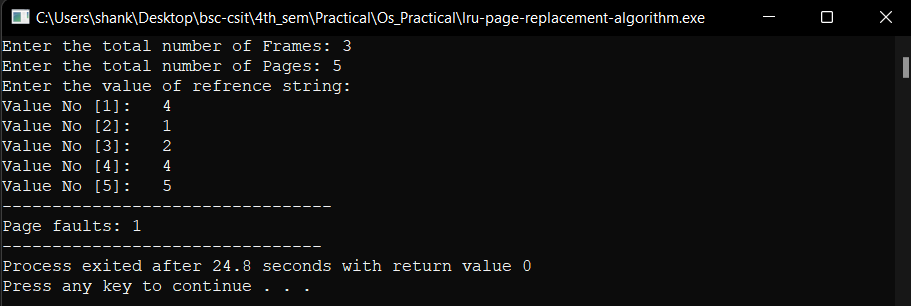
*}*

*printf("\nTotal number of page faults: \t %d\n", page\_fault);*

*return 0;*

*}*

1. **OUTPUT**

****

**LAB NO. 5**

**TITLE: FCFS PROCESS SCHEDULING ALGORITHM**

1. **THEORY**

FCFS provides an efficient, simple and error-free process scheduling algorithm that saves valuable CPU resources. It uses non-preemptive scheduling in which a process is automatically queued and processing occurs according to an incoming request or process order. FCFS derives its concept from real life customer service. FCFS may not be the fastest process scheduling algorithm, as it doesn’t check for priorities associated with processes. These priorities may depend on the processes’ individual execution times.

FEATURES

* Processes are scheduled in the order they are received
* Once the process has the CPU, it runs to completion
* Easily implemented, by managing a simple queue or by storing time the process was received
* Fair to all processes.

1. **SOURCE CODE**

*/\**

*first come first serve process scheduling algorithm*

*Example:*

*processes Burst*

*1 5*

*2 8*

*3 12*

*\*/*

*#include <stdio.h>*

*// Function to calculate waiting time*

*int waitingtime(int proc[], int n, int burst\_time[], int wait\_time[]) {*

*// waiting time for first process is 0*

*wait\_time[0] = 0;*

*// calculating waiting time*

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

*wait\_time[i] = burst\_time[i-1] + wait\_time[i-1] ;*

*return 0;*

*}*

*// Function to calculate turn around time*

*int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[]) {*

*// calculating turnaround time by adding*

*// burst\_time[i] + wait\_time[i]*

*int i;*

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

*tat[i] = burst\_time[i] + wait\_time[i];*

*return 0;*

*}*

*//Function to calculate average time*

*int avgtime( int proc[], int n, int burst\_time[]) {*

*int wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0;*

*int i;*

*//Function to find waiting time of all processes*

*waitingtime(proc, n, burst\_time, wait\_time);*

*//Function to find turn around time for all processes*

*turnaroundtime(proc, n, burst\_time, wait\_time, tat);*

*//Display processes along with all details*

*printf("Processes Burst Time Waiting Time Turnaround Time \n");*

*printf("-------------------------------------------------------\n");*

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

*total\_wt = total\_wt + wait\_time[i];*

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

*printf(" %d\t\t%d\t\t%d \t\t%d\n", i+1, burst\_time[i], wait\_time[i], tat[i]);*

*}*

*printf("\n-------------------------------------------------------\n");*

*printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);*

*printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);*

*return 0;*

*}*

*// main function*

*int main() {*

*//process id's*

*int process[] = { 1, 2, 3, 4};*

*int n = sizeof process / sizeof process[0];*

*//Burst time of all processes*

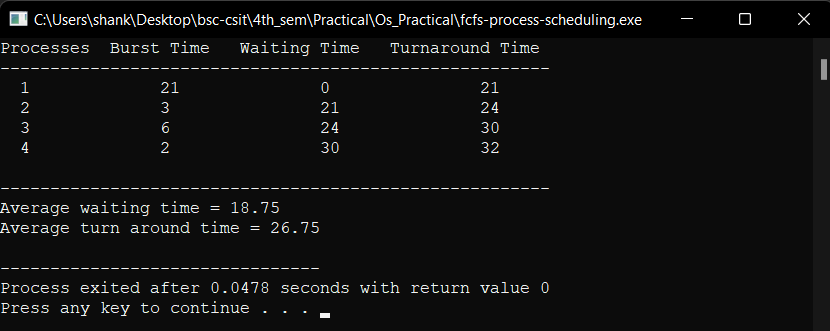
*int burst\_time[] = {21, 3, 6, 2};*

*avgtime(process, n, burst\_time);*

*return 0;*

*}*

1. **OUTPUT**



**LAB NO. 6**

**TITLE: SJF PROCESS SCHEDULING ALGORITHM**

1. **THEORY**

Shortest Job First (SJF) or shortest job next is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJF is a non-preemptive algorithm. Shortest Job First has the advantage of having minimum average waiting time among all the scheduling algorithms. It is a Greedy algorithm, which may cause starvation if shorter processes keep coming. It is practically infeasible as OS may not know brust time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

1. **SOURCE CODE**

*#include<stdio.h>*

*#include<conio.h>*

*#include<string.h>*

*#include <limits.h>*

*// Function to calculate turn around time*

*int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[]) {*

*// calculating turnaround time by adding*

*// burst\_time[i] + wait\_time[i]*

*int i;*

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

*tat[i] = burst\_time[i] + wait\_time[i];*

*return 0;*

*}*

*// Function to calculate waiting time*

*int waitingtime(int proc[], int n, int burst\_time[], int arrival\_time[], int wait\_time[]) {*

*int rt[n];*

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

*rt[i] = burst\_time[i];*

*int complete = 0, t = 0, minm = INT\_MAX;*

*int shortest = 0, finish\_time;*

*bool check = false;*

*while (complete != n) {*

*for (int j = 0; j < n; j++) {*

*if ((arrival\_time[j] <= t) && (rt[j] < minm) && rt[j] > 0) {*

*minm = rt[j];*

*shortest = j;*

*check = true;*

*}*

*}*

*if (check == false) {*

*t++;*

*continue;*

*}*

*// decrementing the remaining time*

*rt[shortest]--;*

*minm = rt[shortest];*

*if (minm == 0){*

*minm = INT\_MAX;*

*// If a process gets completely executed*

*if (rt[shortest] == 0) {*

*complete++;*

*check = false;*

*finish\_time = t + 1;*

*// Calculate waiting time*

*wait\_time[shortest] = finish\_time - burst\_time[shortest] - arrival\_time[shortest];*

*if (wait\_time[shortest] < 0){*

*wait\_time[shortest] = 0;*

*}*

*}*

*// Increment time*

*t++;*

*}*

*}*

*return 0;*

*}*

*//Function to calculate average time*

*int avgtime( int proc[], int n, int burst\_time[], int arrival\_time[]) {*

*int wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0, I;*

*//Function to find waiting time of all processes*

*waitingtime(proc, n, burst\_time, arrival\_time, wait\_time);*

*//Function to find turn around time for all processes*

*turnaroundtime(proc, n, burst\_time, wait\_time, tat);*

*//Display processes along with all details*

*printf("Processes Burst Time Arrival Time Waiting Time Turnaround Time \n");*

*printf("-------------------------------------------------------\n");*

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

*total\_wt = total\_wt + wait\_time[i];*

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

*printf(" %d\t\t%d\t\t%d \t\t%d \t%d\n", proc[i], burst\_time[i], arrival\_time[i], wait\_time[i], tat[i]);*

*}*

*printf("\n-------------------------------------------------------\n");*

*printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);*

*printf("Average turn around time = %.2f\n", (float)total\_tat / (float)n);*

*return 0;*

*}*

*// main function*

*int main() {*

*//process id's*

*int process[] = { 1, 2, 3, 4};*

*// process arrival time*

*int arrival\_time[] = {1, 2, 3, 4};*

*int n = sizeof process / sizeof process[0];*

*//Burst time of all processes*

*int burst\_time[] = {21, 3, 6, 2};*

*avgtime(process, n, burst\_time, arrival\_time);*

*return 0;*

*}*

1. **OUTPUT**

