# Practical - 01

1. **Title: -** Implement and update the BIOS settings of your PC.

1. **Outcome: -** Must be able to understand more about BIOS settings in PC

1. **Objectives:** To understand how BIOS works and what we can change in it

## 4. Nomenclature, theory with self-assessment questionnaire: -

**4.1 Solution:**

Step 1: Restart your computer. Open Start, click the power icon, and click Restart.

* If your computer is locked, click the lock screen, then click the power icon in the bottom-right corner of the screen and click Restart.
* If your computer is already off, press your computers "On" switch.

Step 2: Wait for the computer's first startup screen to appear. Once the startup screen appears, you'll have a very limited window in which you can press the setup key.

* It's best to start pressing the setup key as soon as the computer begins to restart.

|  |  |  |
| --- | --- | --- |
| Del | or | F2 |

* If you see "Press [key] to enter setup" or something similar flash across the bottom of the screen and then disappear, you'll need to restart and try again. Step 3: Press and hold to enter setup. The key you're prompted to press might also be different; if so, use that key instead.

|  |
| --- |
| Fn |

* You'll typically use the "F" keys to access the BIOS. These are at the top of your keyboard, though you may have to locate and hold the key while pressing

the proper "F" key.

* You can look at your computer model's manual or online support page to confirm your computer's BIOS key.

Step 4: Wait for your BIOS to load. After successfully hitting the setup key, the BIOS will load. This should only take a few moments. When the loading is complete, you will be taken to the BIOS settings menu.

Step 5: Familiarize yourself with the BIOS controls. Since BIOS menus don't support mouse input, you'll need to use the arrow keys and other computer-specific keys to navigate the BIOS. You can usually find a list of controls in the bottom-right corner of the BIOS homepage.

Step 6: Change your settings carefully. When adjusting settings in your BIOS, be sure that you certain what the settings will affect. Changing settings incorrectly can lead to system or hardware failure.

* If you don't know what you want to change coming into the BIOS, you probably shouldn't change anything.

Step 7: Change the boot order. If you want to change what device to boot from, enter the Boot menu. From here, you can designate which device the computer will attempt to boot from first. This is useful for booting from a disc or flash drive to install or repair an operating system.

* You'll typically use the arrow keys to go over to the Boot tab to start this process.

Step 8: Create a BIOS password. You can create a password that will lock the computer from booting unless the correct password is entered.

Step 9: Change your date and time. Your BIOS’s clock will dictate your Windows clock. If you replace your computer's battery, your BIOS clock will most likely be reset.

Step 10: Change fan speeds and system voltages. These options are for advanced users only. In this menu, you can [overclock your CPU,](https://www.wikihow.com/Overclock-a-CPU) potentially allowing for higher performance. This should be performed only if you are comfortable with your computer’s hardware.

Step 11: Save and exit. When you are finished adjusting your settings, you will need to save and exit by using your BIOS' "Save and Exit" key in order for your changes to take effect. When you save and restart, your computer will reboot with the new settings.

• Check the BIOS key legend to see which key is the "Save and Exit" key.

# Practical - 02

1. **Title: -** Implement the scheduling for that where CPU give chance to complete those process first, which comes first?

1. **Outcome: -** Able to understand how FCFS algorithm works

1. **Objectives: -** we have to find average burst time, waiting time, turnaround time to implement the FCFS process.

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| proc | No. of processes |
| burst\_time | Burst time of process |
| wait\_time | Waiting time of process |
| tat | Turnaround time |
| i | Counter for loop |

**4.2 Solution:**

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first. **Algorithm:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst time

Step 4: Set the waiting of the first process as ‗0‘and its burst time as its turnaround time

Step 5: for each process in the Ready Q calculate a). Waiting time (n) = waiting time (n-1) +

Burst time (n-1) b). Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate a) Average waiting time = Total waiting Time / Number of process b)

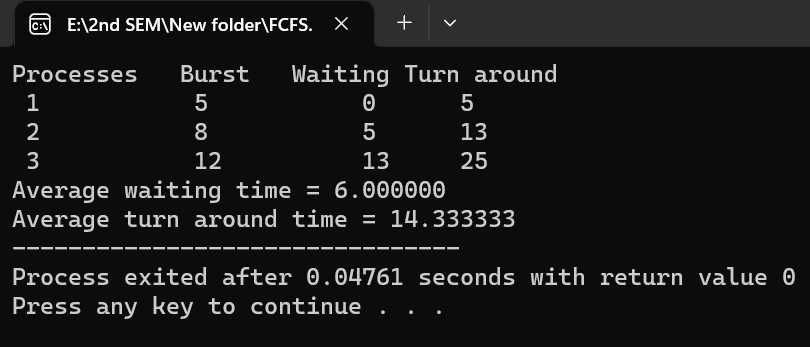
Average Turnaround time = Total Turnaround Time / Number of process Step 7: Stop the process

## 4.3 Code/ Pseudo Code

|  |
| --- |
| **#include <stdio.h>**  **int waitingtime(int proc[], int n, int burst\_time[], int wait\_time[])**  **{ wait\_time[0] = 0;**  **for (int i = 1; i < n ; i++ )**  **wait\_time[i] = burst\_time[i-1] + wait\_time[i-1] ; return 0;**  **}**  **int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[])**  **{ int i;**  **for ( i = 0; i < n ; i++)**  **tat[i] = burst\_time[i] + wait\_time[i]; return 0;**  **}**  **int avgtime( int proc[], int n, int burst\_time[])**  **{**  **int wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0; int i;**  **waitingtime(proc, n, burst\_time, wait\_time); turnaroundtime(proc, n, burst\_time, wait\_time, tat); printf("Processes Burst Waiting Turn around\n"); for ( i=0; i<n; i++)**  **{ total\_wt = total\_wt + wait\_time[i]; total\_tat = total\_tat + tat[i];**  **printf(" %d\t %d\t\t %d \t%d\n", i+1, burst\_time[i],wait\_time[i], tat[i]);**  **}**  **printf("Average waiting time = %f\n", (float)total\_wt /**  **(float)n);**  **printf("Average turn around time = %f", (float)total\_tat /**  **(float)n); return 0; } int main()**  **{**  **int proc[] = { 1, 2, 3};**  **int n = sizeof proc / sizeof proc[0]; int burst\_time[] = {5, 8, 12}; avgtime(proc, n, burst\_time); return 0; }** |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the FCFS algorithm

4.4.2.2 Issues: NA

# Practical – 03

1. **Title: -** Implement Non-Preemptive Shortest Job first CPU Scheduling

1. **Outcome: -** Able to understand how SJF algorithm works

1. **Objectives: -** we have to find average burst time, waiting time, turnaround time to implement the SJF process.

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| proc | No. of processes |
| burst\_time | Burst time of process |
| wait\_time | Waiting time of process |
| tat | Turnaround time |
| i | Counter for loop |
| pos | position |

4.2 **Solution:**

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

**Algorithm:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time. Step 4: Start the Ready Q according to the shortest Burst time by sorting according to lowest to highest burst time. Step 5: Set the waiting time of the first process as ‗0‘and its turnaround time as its burst time. Step 6: Sort the processes names based on their Burt time.

Step 7: For each process in the ready queue, calculate.

1. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
2. Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

1. Average waiting time = Total waiting Time / Number of process

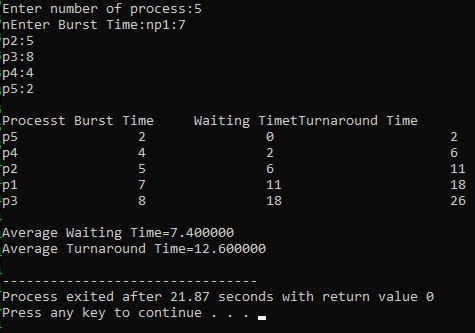
d)Average Turnaround time = Total Turnaround Time / Number of process Step 9: Stop the process.

## 4.3 Code/ Pseudo Code

|  |
| --- |
| **#include<stdio.h> int main()**  **{ int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp; float avg\_wt,avg\_tat;**  **printf("Enter number of process:"); scanf("%d",&n);**  **printf("Enter Burst Time:"); for(i=0;i<n;i++)**  **{ printf("p%d:",i+1); scanf("%d",&bt[i]); p[i]=i+1;**  **} for(i=0;i<n;i++)**  **{ pos=i;**  **for(j=i+1;j<n;j++)**  **{**  **if(bt[j]<bt[pos]) pos=j;**  **} temp=bt[i]; bt[i]=bt[pos]; bt[pos]=temp; temp=p[i]; p[i]=p[pos]; p[pos]=temp;**  **} wt[0]=0;**  **for(i=1;i<n;i++)**  **{ wt[i]=0; for(j=0;j<i;j++)**  **wt[i]+=bt[j]; total+=wt[i];**  **}**  **avg\_wt=(float)total/n; total=0;**  **printf("\nProcesst Burst Time \tWaiting TimetTurnaround Time");**  **for(i=0;i<n;i++)**  **{** |
| **tat[i]=bt[i]+wt[i]; total+=tat[i];**  **printf("\np%d\t\t %d\t\t**  **%d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);**  **}**  **avg\_tat=(float)total/n;**  **printf("\nAverage Waiting Time=%f",avg\_wt); printf("Average Turnaround Time=%fn",avg\_tat); }** |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the SJF algorithm

4.4.2.2 Issues: NA

# Practical - 04

1. **Title: -** Implement the scheduling for that where CPU give chance to complete those process first, which comes first?

1. **Outcome: -** Able to understand how Priority algorithm works

1. **Objectives: -** we have to find average burst time, waiting time, turnaround time to implement the Priority algorithm process.

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| proc | No. of processes |
| burst\_time | Burst time of process |
| wait\_time | Waiting time of process |
| tat | Turnaround time |
| i | Counter for loop |
| pos | position |
| temp | Temporary variable |
| avg\_wt | Average waiting time |
| avg\_tat | Average turnaround time |

**4.2 Solution:**

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process.

Step 2: Accept the number of processes in the ready Queue.

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time.

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as ‗0‘and its burst time as its turnaround time.

Step 6: Arrange the processes based on process priority. Step 7: For each process in the Ready Q calculate Step 8: for each process in the Ready Q calculate

1. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
2. Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate,

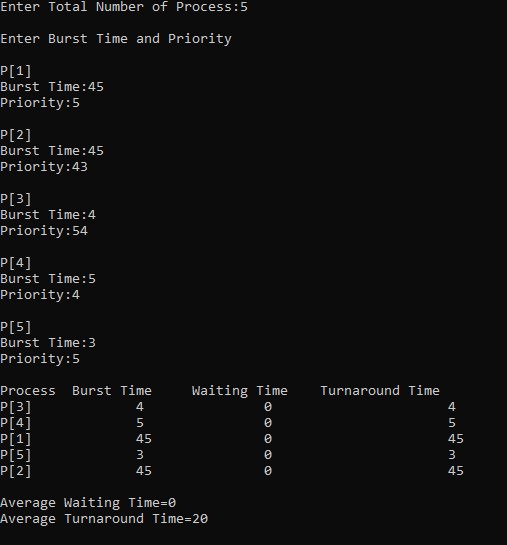
1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Print the results in an order. Step10: Stop

## 4.3 Code/ Pseudo Code

|  |
| --- |
| #include<stdio.h>  int main()  {  int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat; printf("Enter Total Number of Process:"); scanf("%d",&n); printf("\nEnter Burst Time and Priority\n"); for(int i=0;i<n;i++)  {  printf("\nP[%d]\n",i+1); printf("Burst Time:"); scanf("%d",&bt[i]); printf("Priority:"); scanf("%d",&pr[i]); p[i]=i+1;  }    for(int i=0;i<n;i++)  {  pos=I; for(j=i+1;j<n;j++)  {  if(pr[j]<pr[pos]) pos=j;  }  temp=pr[i]; pr[i]=pr[pos]; |
| pr[pos]=temp; temp=bt[i]; bt[i]=bt[pos]; bt[pos]=temp; temp=p[i]; p[i]=p[pos]; p[pos]=temp;  }  wt[0]=0;  for(int i=1;i<n;i++)  {  wt[i]=0; for(j=0;j<I;j++) wt[i]+=bt[j]; total+=wt[i];  }  avg\_wt=total/n; total=0; printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time"); for(int i=0;i<n;i++)  {  tat[i]=bt[i]+wt[i]; total+=tat[i];  printf("\nP[%d]\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);  }  avg\_tat=total/n; //average turnaround time printf("\n\nAverage Waiting Time=%d",avg\_wt); printf("\nAverage Turnaround Time=%d\n",avg\_tat); return 0;  } |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the Priority algorithm

4.4.2.2 Issues: NA

# Practical - 05

1. **Title: -** To simulate the CPU scheduling algorithm round-robin

1. **Outcome: -** Able to understand how Round robin algorithm works

1. **Objectives: -** we are finding the waiting time, turnaround time and burst time of the process to implement the round robin algorithm.

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| proc | No. of processes |
| burst\_time | Burst time of process |
| wait\_time | Waiting time of process |
| tat | Turnaround time |
| i | Counter for loop |

4.2 **Solution:** To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not, it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assigns the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed. Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process

(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

* 1. Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1 ) + the time difference in getting the CPU from process(n-1)
  2. Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

* 1. Average waiting time = Total waiting Time / Number of process
  2. Average Turnaround time = Total Turnaround Time /

Number of process

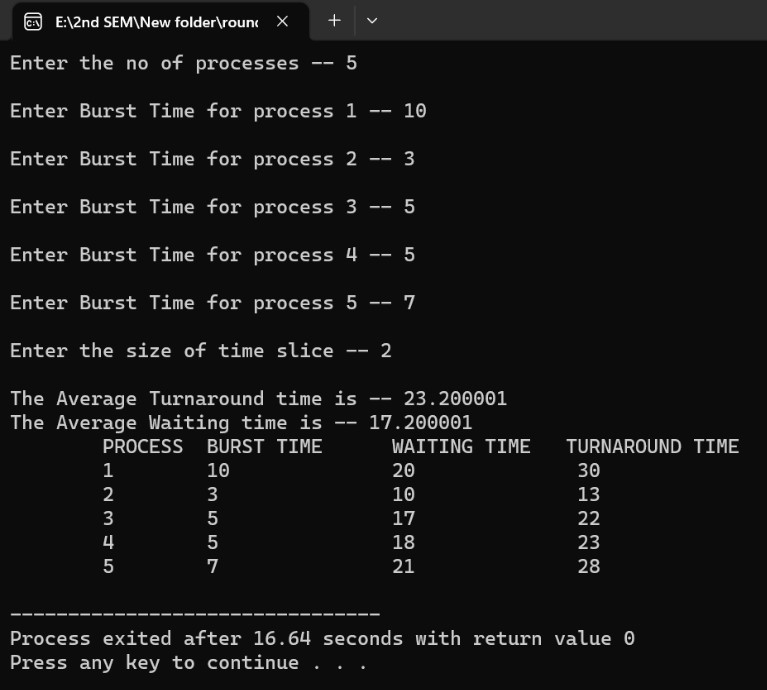
Step 8: Stop the process

## 4.3 Code/ Pseudo Code

|  |
| --- |
| **#include<stdio.h> int main()**  **{**  **int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float awt=0,att=0,temp=0;**  **printf("Enter the no of processes -- "); scanf("%d",&n); for(i=0;i<n;i++)**  **{**  **printf("\nEnter Burst Time for process %d -- ", i+1); scanf("%d",&bu[i]); ct[i]=bu[i];**  **}**  **printf("\nEnter the size of time slice -- "); scanf("%d",&t); max=bu[0]; for(i=1;i<n;i++) if(max<bu[i]) max=bu[i];**  **for(j=0;j<(max/t)+1;j++) for(i=0;i<n;i++) if(bu[i]!=0) if(bu[i]<=t)**  **{ tat[i]=temp+bu[i]; temp=temp+bu[i]; bu[i]=0;**  **} else { bu[i]=bu[i]-t; temp=temp+t;**  **}** |
| **for(i=0;i<n;i++){ wa[i]=tat[i]- ct[i]; att+=tat[i]; awt+=wa[i];} printf("\nThe Average Turnaround time is -- %f",att/n); printf("\nThe Average Waiting time is -- %f ",awt/n); printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n"); for(i=0;i<n;i++)**  **printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]); return 0;**  **}** |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the round robin algorithm

4.4.2.2 Issues: NA

# Practical - 06

1. **Title: -** To Write a C program to simulate producer-consumer problem using Semaphores

1. **Outcome: -** Able to understand how Semaphores technique works

1. **Objectives: -** It is a consumer producer-based technique.

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| choice | Select the choice |
| bufsize | Buffer size |
| i | Counter for loop |
| In | In=0 |
| Out | Out=0 |

4.2 **Solution:**

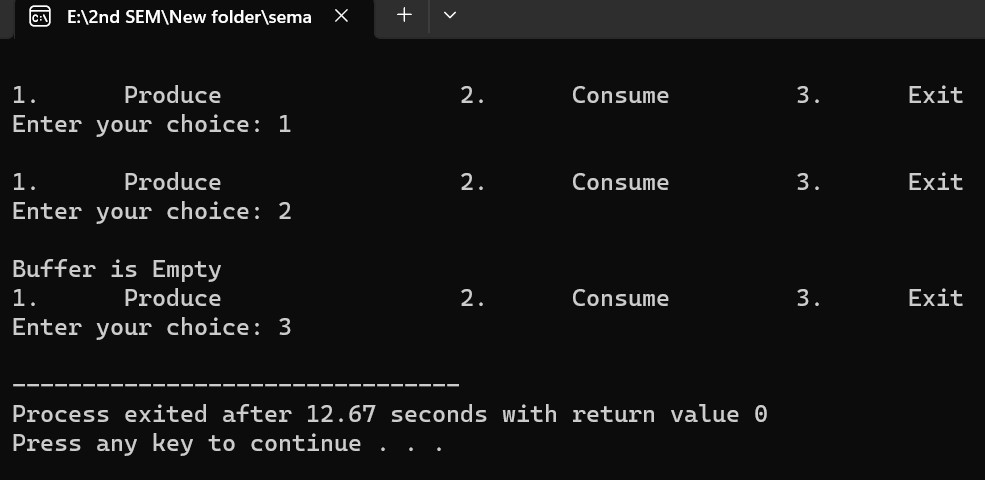
Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer- consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

## 4.3 Code/ Pseudo Code

|  |
| --- |
| **#include<stdio.h> int main()**  **{**  **int buffer[10], bufsize, in, out, produce, consume, choice=0; in = 0; out = 0; bufsize = 10; while(choice !=3)**  **{**  **printf("\n1. Produce \t 2. Consume \t3.**  **Exit"); printf("\nEnter your choice: "); scanf("%d",&choice); switch(choice)**  **{ case 1: if((in+1)%bufsize==out) printf("\nBuffer is Full");**  **else**  **{**  **} break;**  **printf("\nEnter the value: "); scanf("%d", &produce); buffer[in] = produce;**  **in = (in+1)%bufsize;**  **case 2: if(in == out)**  **printf("\nBuffer is Empty");**  **}**  **}**  **return 0;**  **}** |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.3 Advantages: To know more about the semaphores

4.4.2.4 Issues: NA

# Practical - 07

1. **Title: -** To Write a C program to simulate the following contiguous memory allocation techniques

1. **Outcome: -** Able to understand how memory allocation technique works

1. **Objectives: -** It is done in three phases worst fit, best fit, first fit

## 4. Nomenclature, theory with self-assessment questionnaire: -

**4.1 Nomenclature:**

|  |  |
| --- | --- |
| frag | Fragment |
| ff | First fit |
| bf | Best fit |
| nb | No. of blocks |
| i | Counter for loop |
| nf | No. of files |

**4.2 Solution:**

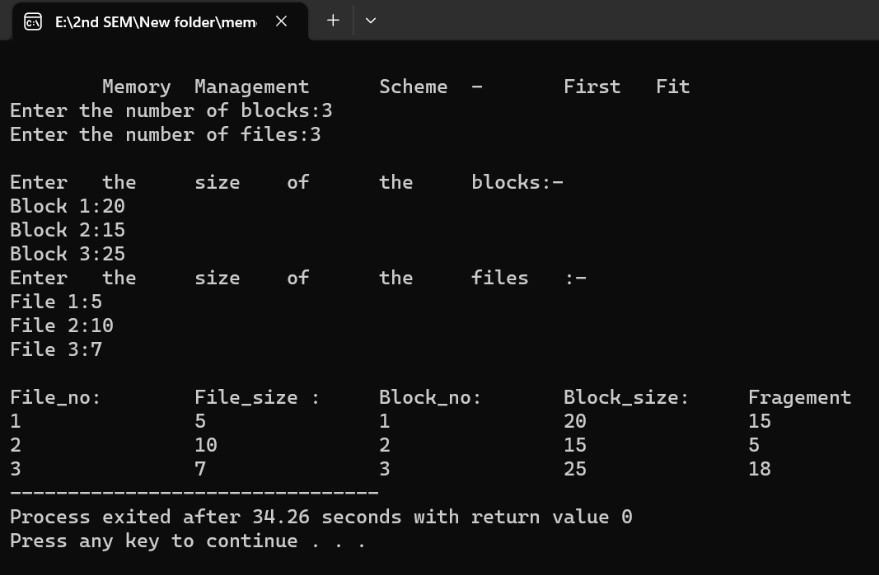
One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

### 4.3 Code/ Pseudo Code

|  |
| --- |
| **#include<stdio.h> #define max 25 int main()**  **{**  **int frag[max],b[max],f[max],i,j,nb,nf,temp; static int bf[max],ff[max];**  **printf("\n\tMemory Management Scheme - FirstFit"); printf("\nEnter the number of blocks:"); scanf("%d",&nb);**  **printf("Enter the number of files:"); scanf("%d",&nf);**  **printf("\nEnter the size of the blocks:-**  **\n");**  **for(i=1;i<=nb;i++)**  **{ printf("Block %d:",i); scanf("%d",&b[i]);**  **}**  **printf("Enter the size of the files :-\n"); for(i=1;i<=nf;i++)**  **{ printf("File %d:",i); scanf("%d",&f[i]);**  **}**  **for(i=1;i<=nf;i++)**  **{**  **for(j=1;j<=nb;j++)**  **{**  **if(bf[j]!=1)**  **{**  **temp=b[j]-f[i]; if(temp>=0)**  **{**  **ff[i]=j; break;**  **}**  **} }**  **frag[i]=temp; bf[ff[i]]=1;**  **}**  **printf("\nFile\_no:\tFile\_size**  **:\tBlock\_no:\tBlock\_size:\tFragement"); for(i=1;i<=nf;i++)**  **printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],f rag[i]); return 0;**  **}** |

### 4.4 Results

#### 4.4.1 Test Case



#### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the memory allocation techniques

4.4.2.2 Issues: NA

# Practical - 08

1. **Title: -** To Write a C program to simulate disk scheduling algorithms

1. **Outcome: -** Able to understand how simulate disk scheduling algorithm works

1. **Objectives: -** it is done in three phases FCFS, SCAN, C-SCAN

1. **Nomenclature, theory with self-assessment questionnaire: - 4.1 Nomenclature:**

|  |  |
| --- | --- |
| arr | Array name |
| seek\_count | Seek count |
| distance | To calculate distance |
| cur\_track | Current track |
| i | Counter for loop |
| head | head |

* 1. **Solution:**

One of the responsibilities of the operating system is to use the hardware efficiently.

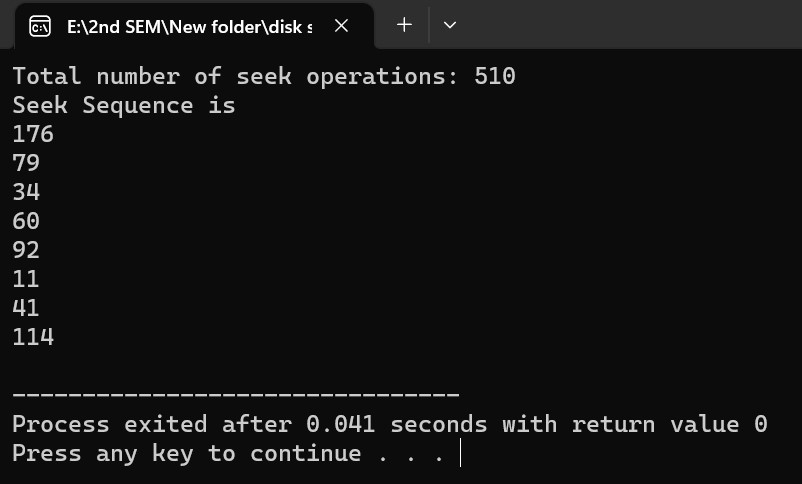
For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. CSCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

* 1. **Code/ Pseudo Code**

|  |
| --- |
| **#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;**  **}** |

## 4.4 Results

### 4.4.1 Test Case



### 4.4.2 Result Analysis

4.4.2.1 Advantages: To know more about the simulate disk scheduling techniques

4.4.2.2 Issues: NA