BANKERS

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
	// P0, P1, P2, P3, P4 are the Process names here

	int n, m, i, j, k;
	n = 5; // Number of processes
	m = 3; // Number of resources
	int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
						{ 2, 0, 0 }, // P1
						{ 3, 0, 2 }, // P2
						{ 2, 1, 1 }, // P3
						{ 0, 0, 2 } }; // P4

	int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
					{ 3, 2, 2 }, // P1
					{ 9, 0, 2 }, // P2
					{ 2, 2, 2 }, // P3
					{ 4, 3, 3 } }; // P4

	int avail[3] = { 3, 3, 2 }; // Available Resources
	int f[n], ans[n], ind = 0;
	for (k = 0; k < n; k++) {
		f[k] = 0;
	}
	int need[n][m];
	for (i = 0; i < n; i++) {
		for (j = 0; j < m; j++)
			need[i][j] = max[i][j] - alloc[i][j];
```

```
        }
        int y = 0;
        for (k = 0; k < 5; k++) {
                for (i = 0; i < n; i++) {
                        if (f[i] == 0) {

                                int flag = 0;
                                for (j = 0; j < m; j++) {
                                        if (need[i][j] > avail[j]){
                                                flag = 1;
                                                break;
                                        }
                                }

                                if (flag == 0) {
                                        ans[ind++] = i;
                                        for (y = 0; y < m; y++)
                                                avail[y] += alloc[i][y];
                                        f[i] = 1;
                                }
                        }
                }
        }

        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
                printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);

        return (0);
}
```

PAGE REPLACEMENT ALGORITHMS

FIFO

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d",&n);
printf("\n ENTER THE PAGE NUMBER :\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n ENTER THE NUMBER OF FRAMES :");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]= -1;
j=0;
printf("\tref string\t page frames\n");
for(i=1;i<=n;i++)
{
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i])
avail=1;
if (avail==0)
{
frame[j]=a[i];
j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf("%d\t",frame[k]);
}
printf("\n");
}
printf("Page Fault Is %d",count);
return 0;
}
```

OPTIMAL

```
#include<stdio.h>

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);
```

```
printf("Enter page reference string: ");

for(i = 0; i < no_of_pages; ++i){
   scanf("%d", &pages[i]);
}

for(i = 0; i < no_of_frames; ++i){
   frames[i] = -1;
}

for(i = 0; i < no_of_pages; ++i){
   flag1 = flag2 = 0;

   for(j = 0; j < no_of_frames; ++j){
      if(frames[j] == pages[i]){
            flag1 = flag2 = 1;
            break;
         }
   }

   if(flag1 == 0){
      for(j = 0; j < no_of_frames; ++j){
         if(frames[j] == -1){
             faults++;
             frames[j] = pages[i];
             flag2 = 1;
             break;
         }
      }
   }

   if(flag2 == 0){
         flag3 =0;

      for(j = 0; j < no_of_frames; ++j){
         temp[j] = -1;

         for(k = i + 1; k < no_of_pages; ++k){
                  if(frames[j] == pages[k]){
                           temp[j] = k;
                           break;
                  }
         }
      }

      for(j = 0; j < no_of_frames; ++j){
         if(temp[j] == -1){
                  pos = j;
                  flag3 = 1;
                  break;
         }
      }

      if(flag3 ==0){
         max = temp[0];
         pos = 0;

         for(j = 1; j < no_of_frames; ++j){
                  if(temp[j] > max){
                           max = temp[j];
                           pos = j;
                  }
         }
      }

                           frames[pos] = pages[i];
                           faults++;
```

```
        }

        printf("\n");

        for(j = 0; j < no_of_frames; ++j){
            printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

LRU

```
#include<stdio.h>

int findLRU(int time[], int n){
        int i, minimum = time[0], pos = 0;

        for(i = 1; i < n; ++i){
                if(time[i] < minimum){
                        minimum = time[i];
                        pos = i;
                }
        }

        return pos;
}

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;
        printf("Enter number of frames: ");
        scanf("%d", &no_of_frames);

        printf("Enter number of pages: ");
        scanf("%d", &no_of_pages);

        printf("Enter reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

        for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == pages[i]){
                        counter++;
                        time[j] = counter;
                                flag1 = flag2 = 1;
                                break;
                        }
        }

        if(flag1 == 0){
                        for(j = 0; j < no_of_frames; ++j){
                        if(frames[j] == -1){
                                counter++;
```

```
                                        faults++;
                                        frames[j] = pages[i];
                                        time[j] = counter;
                                        flag2 = 1;
                                        break;
                                }
                        }
                }

                if(flag2 == 0){
                        pos = findLRU(time, no_of_frames);
                        counter++;
                        faults++;
                        frames[pos] = pages[i];
                        time[pos] = counter;
                }

                printf("\n");

                for(j = 0; j < no_of_frames; ++j){
                        printf("%d\t", frames[j]);
                }
                }

                printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

RR

```
#include<stdio.h>

int main()
{

  int count,j,n,time,remain,flag=0,time_quantum;
  int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
  printf("Enter Total Process:\t ");
  scanf("%d",&n);
  remain=n;
  for(count=0;count<n;count++)
  {
    printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);
    scanf("%d",&at[count]);
    scanf("%d",&bt[count]);
    rt[count]=bt[count];
  }
  printf("Enter Time Quantum:\t");
  scanf("%d",&time_quantum);
  printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
  for(time=0,count=0;remain!=0;)
  {
    if(rt[count]<=time_quantum && rt[count]>0)
    {
      time+=rt[count];
      rt[count]=0;
      flag=1;
    }
    else if(rt[count]>0)
    {
      rt[count]-=time_quantum;
      time+=time_quantum;
    }
    if(rt[count]==0 && flag==1)
    {
      remain--;
      printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
      wait_time+=time-at[count]-bt[count];
      turnaround_time+=time-at[count];
      flag=0;
    }
    if(count==n-1)
      count=0;
    else if(at[count+1]<=time)
      count++;
    else
      count=0;
  }
  printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
  printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

  return 0;
}
```

LRTF

```
#include <bits/stdc++.h>



using namespace std;
```

```
// creating a structure of a process
struct process {
	int processno;
	int AT;
	int BT;

	// for backup purpose to print in last
	int BTbackup;
	int WT;
	int TAT;
	int CT;
};

// creating a structe of 4 processes
struct process p[4];

// variable to find the total time
int totaltime = 0;
int prefinaltotal = 0;

// comparator function for sort()
bool compare(process p1, process p2)
{
	// compare the Arrival time of two processes
	return p1.AT < p2.AT;
}

// finding the largest Arrival Time among all the available
// process at that time
int findlargest(int at)
{
```

```
        int max = 0, i;
        for (i = 0; i < 4; i++) {
                if (p[i].AT <= at) {
                        if (p[i].BT > p[max].BT)
                                max = i;
                }
        }

        // returning the index of the process having the largest BT
        return max;
}

// function to find the completion time of each process
int findCT()
{

        int index;
        int flag = 0;
        int i = p[0].AT;
        while (1) {
                if (i <= 4) {
                        index = findlargest(i);
                }

                else
                        index = findlargest(4);
                cout << "Process executing at time " << totaltime
                        << " is: P" << index + 1 << "\t";

                p[index].BT -= 1;
                totaltime += 1;
```

```
                i++;

                if (p[index].BT == 0) {
                        p[index].CT = totaltime;
                        cout << " Process P" << p[index].processno
                                << " is completed at " << totaltime;
                }
                cout << endl;

                // loop termination condition
                if (totaltime == prefinaltotal)
                        break;
        }
}

int main()
{

        int i;

        // initializing the process number
        for (i = 0; i < 4; i++) {
                p[i].processno = i + 1;
        }

        // cout<<"arrival time of 4 processes : ";
        for (i = 0; i < 4; i++) // taking AT
        {
                p[i].AT = i + 1;
        }
```

```
// cout<<" Burst time of 4 processes : ";
for (i = 0; i < 4; i++) {

        // assigning {2, 4, 6, 8} as Burst Time to the processes
        // backup for displaying the output in last
        // calculating total required time for terminating
        // the function().
        p[i].BT = 2 * (i + 1);
        p[i].BTbackup = p[i].BT;
        prefinaltotal += p[i].BT;
}

// displaying the process before executing
cout << "PNo\tAT\tBT\n";

for (i = 0; i < 4; i++) {
        cout << p[i].processno << "\t";
        cout << p[i].AT << "\t";
        cout << p[i].BT << "\t";
        cout << endl;
}
cout << endl;

// soritng process according to Arrival Time
sort(p, p + 4, compare);

// calculating initial time when execution starts
totaltime += p[0].AT;

// calculating to terminate loop
prefinaltotal += p[0].AT;
```

```
findCT();
int totalWT = 0;
int totalTAT = 0;
for (i = 0; i < 4; i++) {
        // since, TAT = CT - AT
        p[i].TAT = p[i].CT - p[i].AT;
        p[i].WT = p[i].TAT - p[i].BTbackup;

        // finding total waiting time
        totalWT += p[i].WT;

        // finding total turn around time
        totalTAT += p[i].TAT;
}

cout << "After execution of all processes ... \n";

// after all process executes
cout << "PNo\tAT\tBT\tCT\tTAT\tWT\n";

for (i = 0; i < 4; i++) {
        cout << p[i].processno << "\t";
        cout << p[i].AT << "\t";
        cout << p[i].BTbackup << "\t";
        cout << p[i].CT << "\t";
        cout << p[i].TAT << "\t";
        cout << p[i].WT << "\t";
        cout << endl;
}

cout << endl;
```

```
    cout << "Total TAT = " << totalTAT << endl;
    cout << "Average TAT = " << totalTAT / 4.0 << endl;
    cout << "Total WT = " << totalWT << endl;
    cout << "Average WT = " << totalWT / 4.0 << endl;
    return 0;
}
```

SJF

```
#include<stdio.h>

void 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("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;           //contains process number
    }

    //sorting burst time in ascending order using selection sort
    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;            //waiting time for first process will be zero

    //calculate waiting time
    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;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        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;     //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

PRIORITY SCHEDULING PREEMPTIVE

```
#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(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;           //contains process number
    }

    //sorting burst time, priority and process number in ascending order using selection sort
    for(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;            //waiting time for first process is zero

    //calculate waiting time
    for(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;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        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;
}
```

FCFS

```
 #include<stdio.h>


int main()
{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    printf("Enter total number of processes(maximum 20):");
    scanf("%d",&n);

    printf("\nEnter Process Burst Time\n");
    for(i=0;i<n;i++)
    {
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
    }

    wt[0]=0;    //waiting time for first process is 0

    //calculating waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }

    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

    //calculating turnaround time
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;
    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d",avtat);

    return 0;
}
```

FF

```
#include<stdio.h>

void main()
{
	int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

	for(i = 0; i < 10; i++)
	{
		flags[i] = 0;
		allocation[i] = -1;
	}

	printf("Enter no. of blocks: ");
	scanf("%d", &bno);

	printf("\nEnter size of each block: ");
	for(i = 0; i < bno; i++)
		scanf("%d", &bsize[i]);

	printf("\nEnter no. of processes: ");
	scanf("%d", &pno);

	printf("\nEnter size of each process: ");
	for(i = 0; i < pno; i++)
		scanf("%d", &psize[i]);
	for(i = 0; i < pno; i++)	//allocation as per first fit
		for(j = 0; j < bno; j++)
			if(flags[j] == 0 && bsize[j] >= psize[i])
			{
				allocation[j] = i;
				flags[j] = 1;
				break;
			}

	//display allocation details
	printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
	for(i = 0; i < bno; i++)
	{
		printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
		if(flags[i] == 1)
			printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);
		else
			printf("Not allocated");
	}
}
```

BF

```
#include<stdio.h>

void main()
{
	int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
	static int barray[20],parray[20];

	printf("\n\t\t\tMemory Management Scheme - Best Fit");
	printf("\nEnter the number of blocks:");
	scanf("%d",&nb);
	printf("Enter the number of processes:");
	scanf("%d",&np);
```

```
        printf("\nEnter the size of the blocks:-\n");
        for(i=1;i<=nb;i++)
  {
                printf("Block no.%d:",i);
    scanf("%d",&b[i]);
  }

        printf("\nEnter the size of the processes :-\n");
        for(i=1;i<=np;i++)
  {
    printf("Process no.%d:",i);
    scanf("%d",&p[i]);
  }

        for(i=1;i<=np;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(barray[j]!=1)
                        {
                                temp=b[j]-p[i];
                                if(temp>=0)
                                        if(lowest>temp)
                                        {
                                                parray[i]=j;
                                                lowest=temp;
                                        }
                        }
                }

                fragment[i]=lowest;
                barray[parray[i]]=1;
                lowest=10000;
        }

        printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
        for(i=1;i<=np && parray[i]!=0;i++)
                printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

WF

```
#include<stdio.h>

int main()
{
    int fragments[10], blocks[10], files[10];
    int m, n, number_of_blocks, number_of_files, temp, top = 0;
    static int block_arr[10], file_arr[10];
    printf("\nEnter the Total Number of Blocks:\t");
    scanf("%d",&number_of_blocks);
    printf("Enter the Total Number of Files:\t");
    scanf("%d",&number_of_files);
    printf("\nEnter the Size of the Blocks:\n");
    for(m = 0; m < number_of_blocks; m++)
    {
        printf("Block No.[%d]:\t", m + 1);
        scanf("%d", &blocks[m]);
    }
    printf("Enter the Size of the Files:\n");
```

```
    for(m = 0; m < number_of_files; m++)
    {
        printf("File No.[%d]:\t", m + 1);
        scanf("%d", &files[m]);
    }
    for(m = 0; m < number_of_files; m++)
    {
        for(n = 0; n < number_of_blocks; n++)
        {
            if(block_arr[n] != 1)
            {
                temp = blocks[n] - files[m];
                if(temp >= 0)
                {
                    if(top < temp)
                    {
                        file_arr[m] = n;
                        top = temp;
                    }
                }
            }
            fragments[m] = top;
            block_arr[file_arr[m]] = 1;
            top = 0;
        }
    }
    printf("\nFile Number\tFile Size\tBlock Number\tBlock Size\tFragment");
    for(m = 0; m < number_of_files; m++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", m, files[m], file_arr[m],
blocks[file_arr[m]], fragments[m]);
    }
    printf("\n");
    return 0;
}
```

# ACTIVITY 10

NAME: SHREYA MAHESHWARI

REGISTRATION NUMBER: 18BCE0167

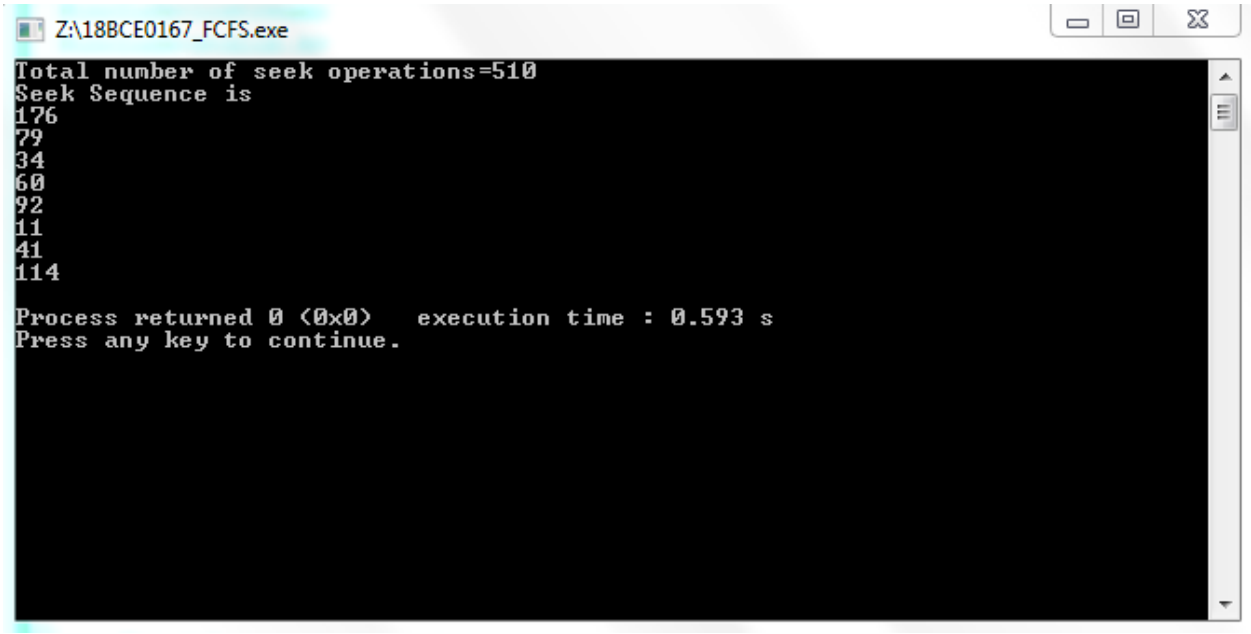
1. **FCFS Disk Scheduling Algorithm**

**CODE:**

```
//FCFS Disk Scheduling Algorithm
#include<iostream>
#include<stdlib.h>
using namespace std;
int size=8;
void FCFS(int arr[], int head)
{
   int seek_count=0;
   int distance, cur_track;
   for(int i=0; i<size; i++)
   {
      cur_track = arr[i];
      distance=abs(cur_track - head);
      seek_count += distance;
      head=cur_track;
   }
   cout<<"Total number of seek operations="
      <<seek_count<<endl;
```

```
    cout<<"Seek Sequence is"<<endl;


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

        cout<<arr[i]<<endl;

    }


}

int main()

{

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

    int head = 50;

    FCFS(arr,head);

    return 0;

}
```

**OUTPUT:**



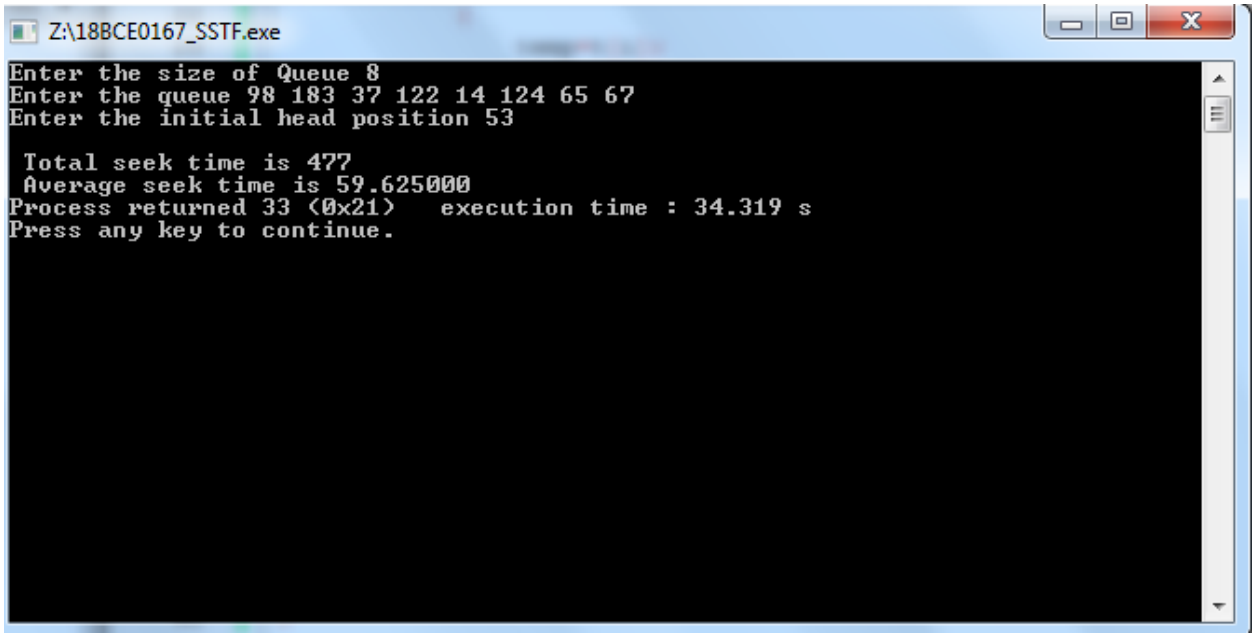
**2. SSTF Disk Scheduling Algorithm**

**CODE:**

```
#include<stdio.h>

#include<stdlib.h>

#include<math.h>

void main()

{

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

   float avg;

   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++)

   {

      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("\n Total seek time is %d\t",seek);
  avg=seek/(float)n;
  printf("\n Average seek time is %f\t",avg);
}
```

**OUTPUT**:




**3. SCAN Disk Scheduling Algorithm**

**CODE:**

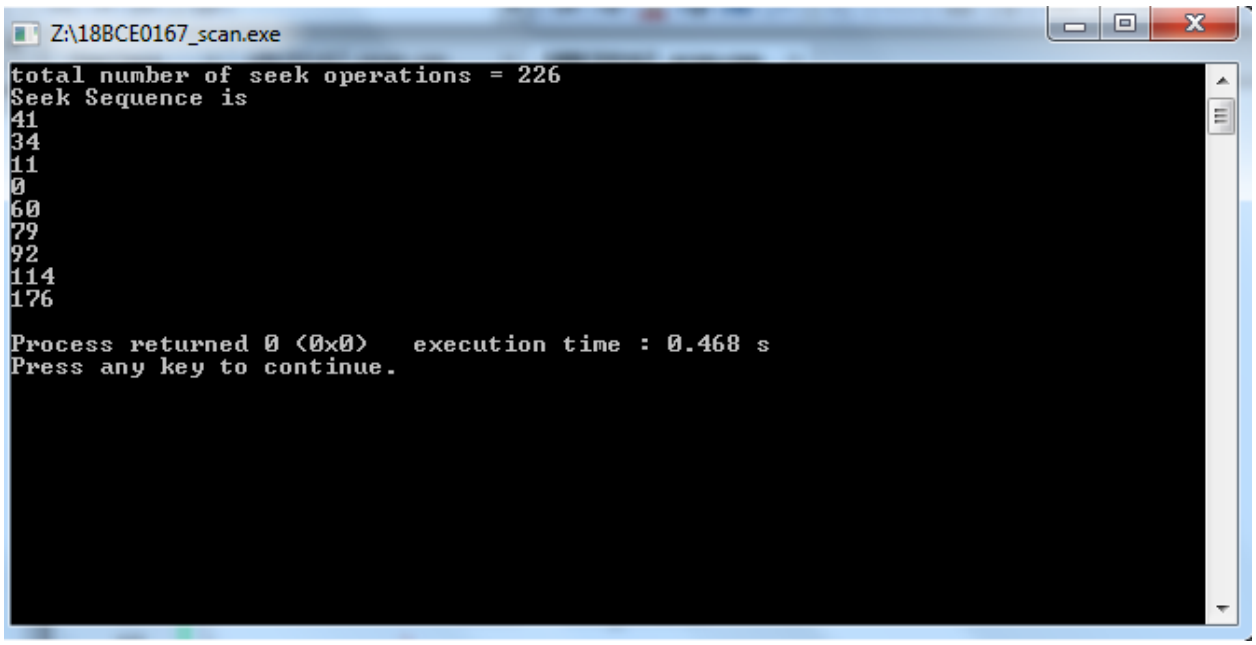
```
#include<iostream>

#include<stdlib.h>

#include<vector>

#include<algorithm>

using namespace std;

int size=8;

int disk_size=200;

void SCAN(int arr[], int head, string direction)

{

   int seek_count = 0;

   int distance, cur_track;

   vector<int> left, right;

   vector<int> seek_sequence;
```

```
if(direction == "left")
    left.push_back(0);
else if(direction == "right")
    right.push_back(disk_size - 1);

for(int i=0; i<size; i++){
    if(arr[i]<head)
        left.push_back(arr[i]);
    if(arr[i]>head)
        right.push_back(arr[i]);
}
sort(left.begin(), left.end());
sort(right.begin(), right.end());

int run=2;
while(run--){
    if(direction == "left"){
        for(int i=left.size()-1; i>=0; i--){
            cur_track = left[i];
            seek_sequence.push_back(cur_track);
            distance=abs(cur_track-head);
            seek_count += distance;
            head = cur_track;
        }
```

```
            direction = "right";
        }
        else if(direction == "right"){
            for(int i=0; i<right.size(); i++){
                cur_track = right[i];
                seek_sequence.push_back(cur_track);
                distance= abs(cur_track-head);
                seek_count += distance;
                head = cur_track;
            }
            direction = "left";
        }
    }
    cout<<"total number of seek operations = "
        <<seek_count<<endl;


    cout<<"Seek Sequence is" << endl;
    for(int i=0; i<seek_sequence.size(); i++){
        cout<<seek_sequence[i]<<endl;
    }
}
int main()
{
    int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
    int head=50;
```

```
    string direction = "left";

    SCAN(arr, head, direction);

    return 0;

}
```

**OUTPUT:**




**4. CSCAN Disk Scheduling Algorithm**

**CODE:**

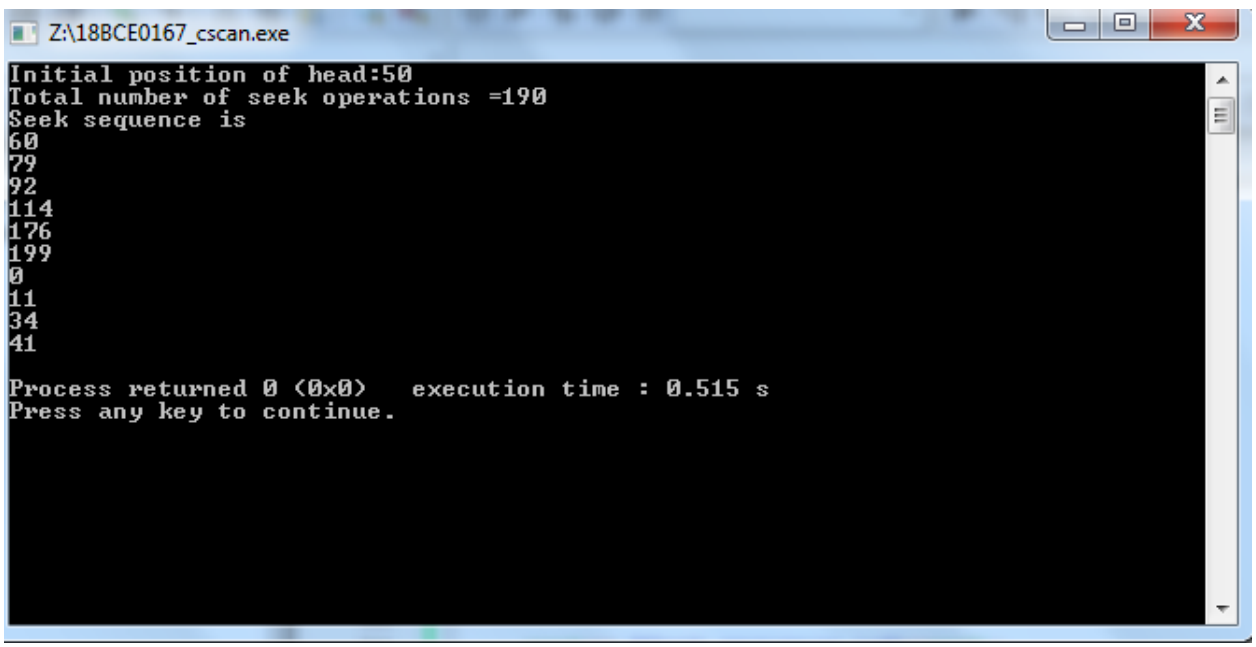
```
#include<iostream>

#include<vector>

#include<stdlib.h>

#include<algorithm>

using namespace std;


int size = 8;

int disk_size = 200;
```

```
void CSCAN(int arr[], int head)
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;

    left.push_back(0);
    right.push_back(disk_size-1);

    for(int i=0; i<size; i++){
        if(arr[i]<head)
            left.push_back(arr[i]);
        if(arr[i]>head)
            right.push_back(arr[i]);
    }

    sort(left.begin(), left.end());
    sort(right.begin(), right.end());

    for(int i=0; i<right.size(); i++){
        cur_track = right[i];
        seek_sequence.push_back(cur_track);

        distance =abs(cur_track-head);
```

```
        seek_count += distance;

        head = cur_track;
    }
    head = 0;

    for(int i=0; i<left.size(); i++){
        cur_track = left[i];

        seek_sequence.push_back(cur_track);

        distance =abs(cur_track-head);
        seek_count += distance;
        head = cur_track;
    }
    cout<< "Total number of seek operations ="
        << seek_count<<endl;

    cout<< "Seek sequence is"<<endl;

    for(int i=0; i<seek_sequence.size(); i++){
        cout<< seek_sequence[i] << endl;
        }
}
int main()
```

```
{
   int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
   int head=50;

   cout<< "Initial position of head:"<< head<<endl;
   CSCAN(arr, head);

   return 0;
}
```

**OUTPUT:**




5. **LOOK Disk Scheduling Algorithm**

**CODE:**

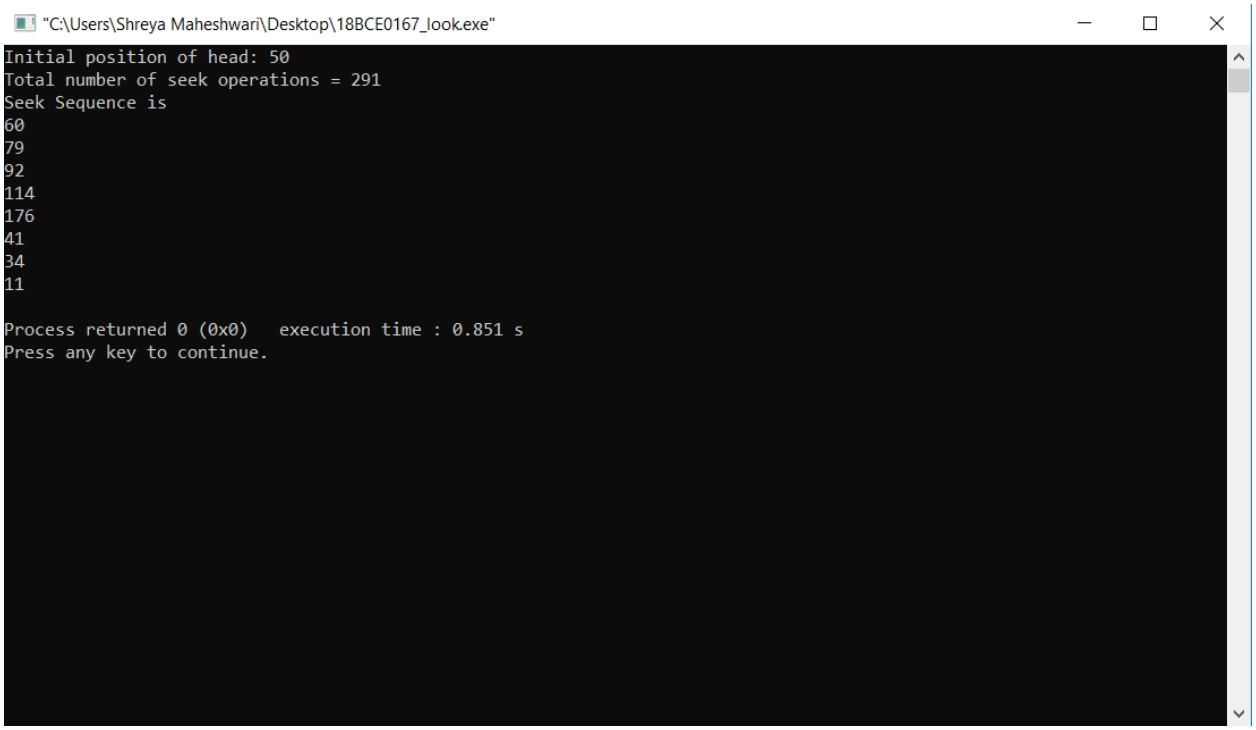
```
#include<iostream>
#include<vector>
#include<stdlib.h>
```

```
#include<algorithm>

using namespace std;


int size = 8;

int disk_size = 200;


void LOOK(int arr[], int head, string direction)

{

   int seek_count = 0;

   int distance, cur_track;

   vector<int> left, right;

   vector<int> seek_sequence;


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

      if (arr[i] < head)

         left.push_back(arr[i]);

      if (arr[i] > head)

         right.push_back(arr[i]);

   }


   sort(left.begin(), left.end());

   sort(right.begin(), right.end());


   int run = 2;

   while (run--) {
```

```
    if (direction == "left") {
        for (int i = left.size() - 1; i >= 0; i--) {
            cur_track = left[i];
            seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        direction = "right";
    }
    else if (direction == "right") {
        for (int i = 0; i < right.size(); i++) {
            cur_track = right[i];
            seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        direction = "left";
    }
}

cout << "Total number of seek operations = "
     << seek_count << endl;
```

```
    cout << "Seek Sequence is" << endl;


    for (int i = 0; i < seek_sequence.size(); i++) {
        cout << seek_sequence[i] << endl;
    }
}
int main()
{
    int arr[size] = { 176, 79, 34, 60,
                    92, 11, 41, 114 };
    int head = 50;
    string direction = "right";


    cout << "Initial position of head: "
         << head << endl;


    LOOK(arr, head, direction);


    return 0;
}
```

**OUTPUT**:




6. **C-LOOK Disk Scheduling Algorithm**

**CODE:**

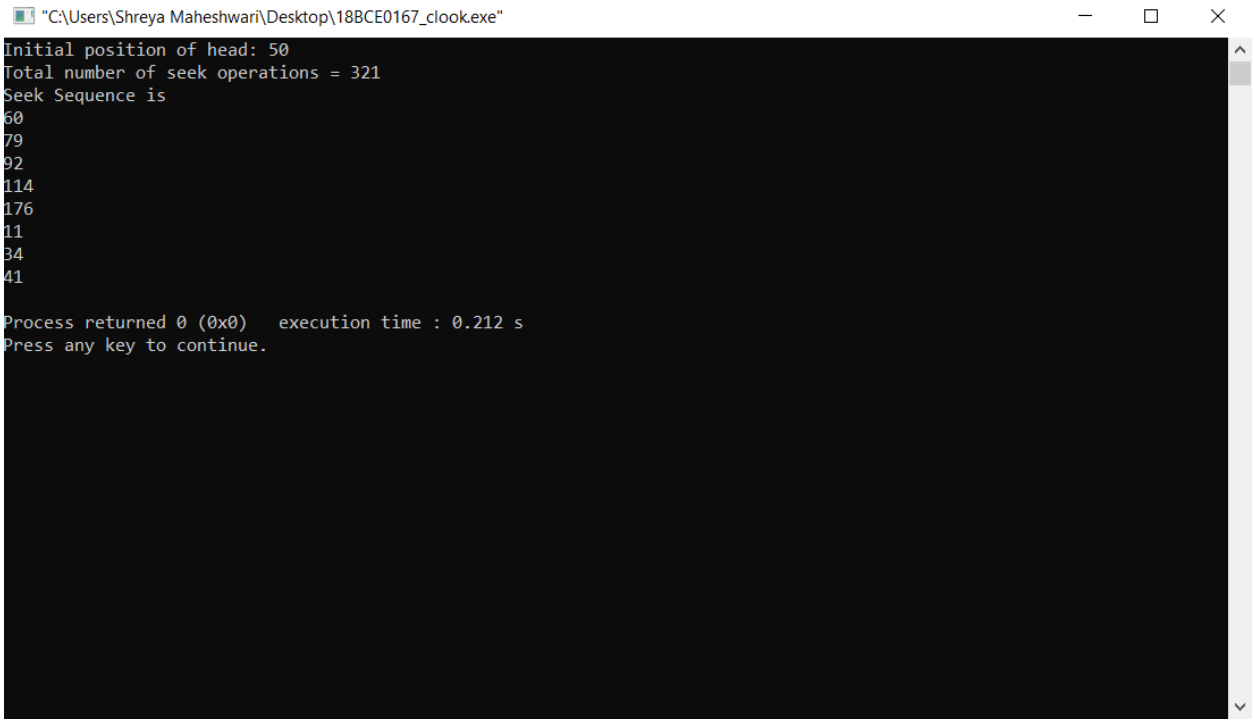
```
#include<iostream>

#include<vector>

#include<stdlib.h>

#include<algorithm>

using namespace std;


int size = 8;

int disk_size = 200;


void CLOOK(int arr[], int head)

{
```

```
int seek_count = 0;
int distance, cur_track;
vector<int> left, right;
vector<int> seek_sequence;

for (int i = 0; i < size; i++) {
    if (arr[i] < head)
        left.push_back(arr[i]);
    if (arr[i] > head)
        right.push_back(arr[i]);
}

sort(left.begin(), left.end());
sort(right.begin(), right.end());

for (int i = 0; i < right.size(); i++) {
    cur_track = right[i];
    seek_sequence.push_back(cur_track);
    distance = abs(cur_track - head);
    seek_count += distance;
    head = cur_track;
}
seek_count += abs(head - left[0]);
head = left[0];
for (int i = 0; i < left.size(); i++) {
```

```
        cur_track = left[i];
        seek_sequence.push_back(cur_track);
        distance = abs(cur_track - head);
        seek_count += distance;
        head = cur_track;
    }

    cout << "Total number of seek operations = "
         << seek_count << endl;

    cout << "Seek Sequence is" << endl;

    for (int i = 0; i < seek_sequence.size(); i++) {
        cout << seek_sequence[i] << endl;
    }
}
int main()
{
    int arr[size] = { 176, 79, 34, 60,
                      92, 11, 41, 114 };
    int head = 50;

    cout << "Initial position of head: " << head << endl;

    CLOOK(arr, head);
```

```
    return 0;

}
```

**OUTPUT**:




# WRITTEN NOTES:

**OS LAB ASSIGNMENT**

**ACTIVITY 9**

Name: Shreya Maheshwari

Registration number: 18BCE0167

**CIGARETTE SMOKER**

**CODE:**

```
#include <pthread.h>
#include <semaphore.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

sem_t smoker_waits[3];
sem_t agent;

typedef void *(*smokers_t)(void *);

void* smoker_0(void* arg)
{
  while (true)
  {
    nanosleep((struct timespec[]){{0, rand() % 200000000}}, NULL);
    printf("Smoker zero waiting to smoke (has paper)\n");

    sem_wait(&smoker_waits[0]);
    printf("Smoker zero received matchges and tobacco. Now making cigarette.\n");
    sem_post(&agent);
  }

  return NULL;
}

void* smoker_1(void* arg)
{
  while (true)
  {
    nanosleep((struct timespec[]){{0, rand() % 200000000}}, NULL);
    printf("Smoker one waiting to smoke (has matches)\n");
```

```
    sem_wait(&smoker_waits[1]);
    printf("Smoker zero received matches and tobacco. Now making cigarette.\n");
    sem_post(&agent);
  }

  return NULL;
}

void* smoker_2(void* arg)
{
  while (true)
  {
    nanosleep((struct timespec[]){{0, rand() % 200000000}}, NULL);
    printf("Smoker two waiting to smoke (has tobacco)\n");

    sem_wait(&smoker_waits[2]);
    printf("Smoker zero received matches and tobacco. Now making cigarette.\n");
    sem_post(&agent);
  }

  return NULL;
}

int main(int argc, char* arvg[])
{
  srand(time(NULL));
  sem_init(&agent, 0, 1);

  pthread_t smoker_threads[3];
  smokers_t smoker_functions[3] = { smoker_0, smoker_1, smoker_2 };

  for (int i = 0; i < 3; ++i)
  {
    sem_init(&smoker_waits[i], 0, 0);

    if(pthread_create(&smoker_threads[i], NULL, smoker_functions[i], NULL) == EAGAIN)
    {
      perror("Insufficent resources to create thread\n");
      return 0;
    }
  }

  while (true)
  {
    sem_wait(&agent);
    sem_post(&smoker_waits[rand() % 2]);
```

```
  }
  for (int i = 0; i < 3; ++i)
  {
    pthread_join(smoker_threads[i], NULL);
  }
}
```

OUTPUT:

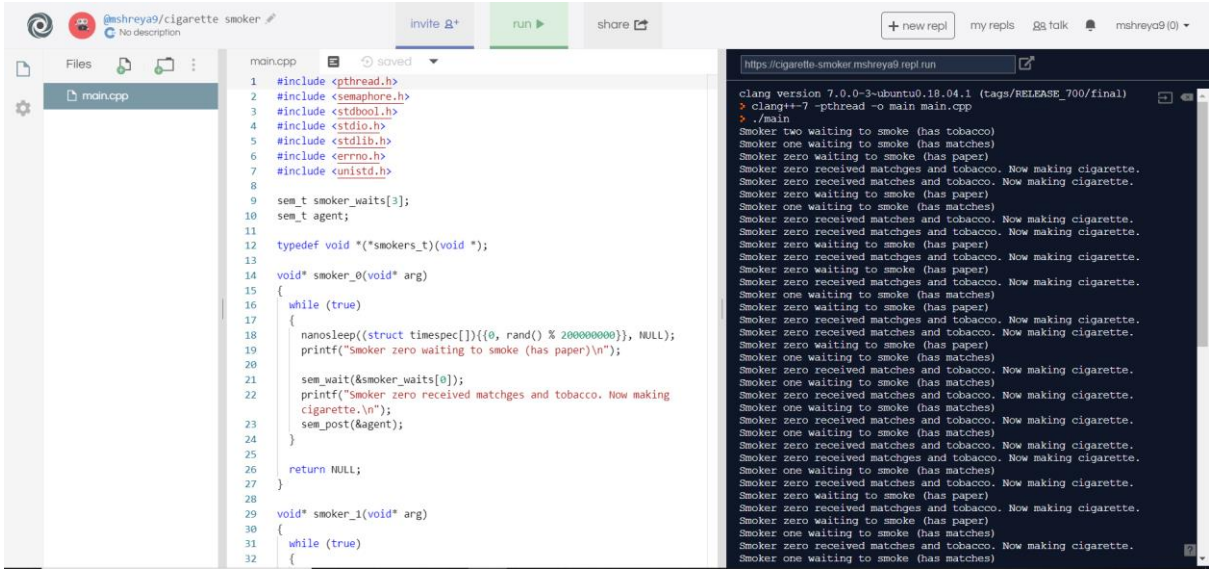
Online Compiler: The code is run on **repl.it**

**(Lab's telnet was not working; hence, online compiler is used)**

**<u>Account:</u>**

Name: Shreya Maheshwari

Username: @mshreya9




**<u>SLEEPING BARBER PROBLEM:</u>**

CODE:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_CUSTOMERS 25

void *customer(void *num);
```

```
void *barber(void *);
void randwait(int secs);

// waitingRoom Limits the # of customers allowed  to enter the waiting room at one time.
sem_t waitingRoom;
// barberChair ensures mutually exclusive access to the barber chair.
sem_t barberChair;
// barberPillow is used to allow the barber to sleep until a customer arrives.
sem_t barberPillow;
// seatBelt is used to make the customer to wait until the barber is done cutting his/her hair.
sem_t seatBelt;
// Flag to stop the barber thread when all customers have been serviced.
int allDone = 0;

int main(int argc, char *argv[])
{
   pthread_t btid;
   pthread_t tid[MAX_CUSTOMERS];
   int i, x, numCustomers, numChairs; int Number[MAX_CUSTOMERS];
   printf("Maximum number of customers can only be 25. Enter number of customers and
chairs.\n");
   scanf("%d",&x);
   numCustomers = x;
   scanf("%d",&x);
   numChairs = x;
   if (numCustomers > MAX_CUSTOMERS) {
     printf("The maximum number of Customers is %d.\n", MAX_CUSTOMERS);
     system("PAUSE");
     return 0;
   }
   printf("A solution to the sleeping barber problem using semaphores.\n");
   for (i = 0; i < MAX_CUSTOMERS; i++) {
      Number[i] = i;
   }
   // Initialize the semaphores with initial values...
   sem_init(&waitingRoom, 0, numChairs);
   sem_init(&barberChair, 0, 1);
   sem_init(&barberPillow, 0, 0);
   sem_init(&seatBelt, 0, 0);

   // Create the barber.
   pthread_create(&btid, NULL, barber, NULL);

   // Create the customers.
   for (i = 0; i < numCustomers; i++) {
      pthread_create(&tid[i], NULL, customer, (void *)&Number[i]);
   }
```

```
   // Join each of the threads to wait for them to finish.
   for (i = 0; i < numCustomers; i++) {
      pthread_join(tid[i],NULL);
   }
   // When all of the customers are finished, kill the barber thread.
   allDone = 1;
   sem_post(&barberPillow); // Wake the barber so he will exit.
   pthread_join(btid,NULL);
   system("PAUSE");
   return 0;
}

void *customer(void *number) {
    int num = *(int *)number; // Leave for the shop and take some random amount of  time
to arrive.
    printf("Customer %d leaving for barber shop.\n", num);
    randwait(5);
    printf("Customer %d arrived at barber shop.\n", num); // Wait for space to open up in
the waiting room...
    sem_wait(&waitingRoom);
    printf("Customer %d entering waiting room.\n", num); // Wait for the barber chair to
become free.
    sem_wait(&barberChair); // The chair is free so give up your spot in the  waiting room.
    sem_post(&waitingRoom); // Wake up the barber...
    printf("Customer %d waking the barber.\n", num);
    sem_post(&barberPillow); // Wait for the barber to finish cutting your hair.
    sem_wait(&seatBelt); // Give up the chair.
    sem_post(&barberChair);
    printf("Customer %d leaving barber shop.\n", num);
}

void *barber(void *junk)
{
// While there are still customers to be serviced... Our barber is omnicient and can tell if
there are  customers still on the way to his shop.

  while (!allDone) { // Sleep until someone arrives and wakes you..
   printf("The barber is sleeping\n");
   sem_wait(&barberPillow); // Skip this stuff at the end...
   if (!allDone)
   { // Take a random amount of time to cut the customer's hair.
    printf("The barber is cutting hair\n");
    randwait(3);
    printf("The barber has finished cutting hair.\n"); // Release the customer when done
cutting...
    sem_post(&seatBelt);
   }
```

```
    else {
        printf("The barber is going home for the day.\n");
    }
  }
}

void randwait(int secs) {
    int len = 1;
    sleep(len);
}
```

OUTPUT:

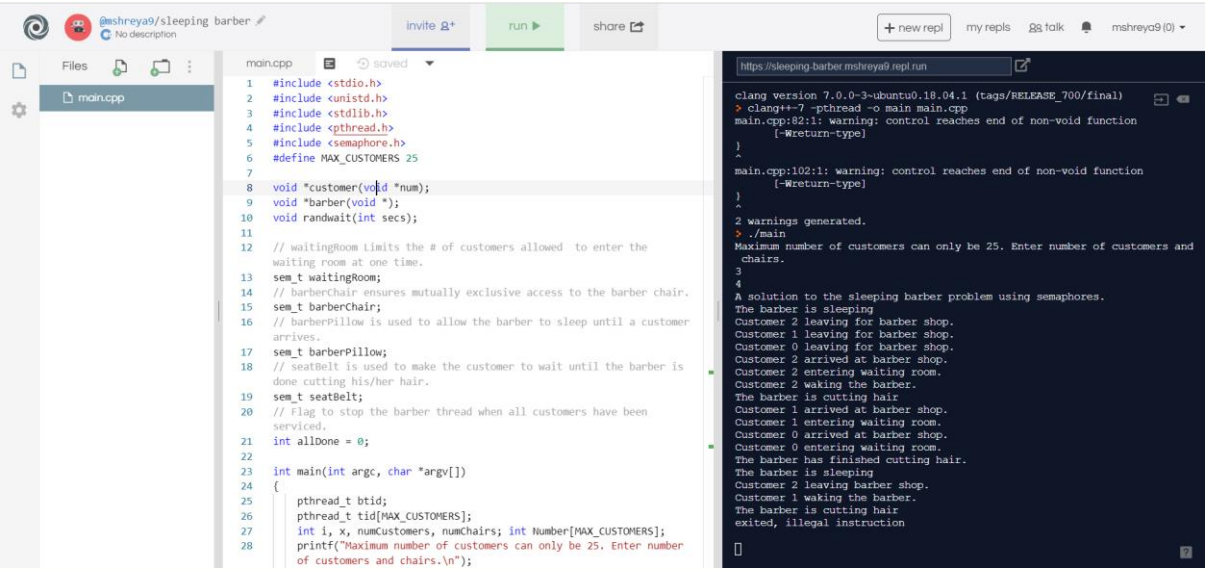
Online Compiler: The code is run on **repl.it**

**(Lab's telnet was not working; hence, online compiler is used)**

**<u>Account:</u>**

Name: Shreya Maheshwari

Username: @mshreya9




**<u>BARRIER IMLPEMENTATION</u>**

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

static int nthread = 1;
```

```
static int round = 0;

struct barrier {
  pthread_mutex_t barrier_mutex;
  pthread_cond_t barrier_cond;
  int nthread;      // Number of threads that have reached this round of the
barrier
  int round;     // Barrier round
} bstate;

static void
barrier_init(void)
{
  assert(pthread_mutex_init(&bstate.barrier_mutex, NULL) == 0);
  assert(pthread_cond_init(&bstate.barrier_cond, NULL) == 0);
  bstate.nthread = 0;
}

static void
barrier()
{
  pthread_mutex_lock(&bstate.barrier_mutex);
  bstate.nthread++;
  if (bstate.nthread == nthread) {
    pthread_cond_broadcast(&bstate.barrier_cond);
    bstate.nthread = 0;
    bstate.round++;
  }
  else {
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}

static void *
thread(void *xa)
{
  long n = (long) xa;
  long delay;
  int i;

  for (i = 0; i < 20000; i++) {
    int t = bstate.round;
    assert (i == t);
    barrier();
    usleep(random() % 100);
  }
}
```

```
int
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  long i;
  double t1, t0;

  if (argc < 2) {
    fprintf(stderr, "%s: %s nthread\n", argv[0], argv[0]);
    exit(-1);
  }
  nthread = atoi(argv[1]);
  tha = malloc(sizeof(pthread_t) * nthread);
  srandom(0);

  barrier_init();

  for(i = 0; i < nthread; i++) {
    assert(pthread_create(&tha[i], NULL, thread, (void *) i) == 0);
  }
  for(i = 0; i < nthread; i++) {
    assert(pthread_join(tha[i], &value) == 0);
  }
  printf("OK; passed\n");
}
```

Online Compiler: The code is run on **repl.it**

**(Lab's telnet was not working; hence, online compiler is used)**

**<u>Account:</u>**

Name: Shreya Maheshwari

Username: @mshreya9

**The implementation of the above code was not done on the online compiler, as shown in the output screen**

@mshreya9/barrier implementation
No description

invite | stop | share | + new repl | my repls | talk | mshreya9 (0)

Files

main.cpp

main.cpp saved

```
    barrier();
    usleep(random() % 100);
  }
}

int
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  long i;
  double t1, t0;

  if (argc < 2) {
    fprintf(stderr, "%s: %s nthread\n", argv[0], argv[0]);
    exit(-1);
  }
  nthread = atoi(argv[1]);
  tha = malloc(sizeof(pthread_t) * nthread);
  srandom(0);

  barrier_init();

  for(i = 0; i < nthread; i++) {
    assert(pthread_create(&tha[i], NULL, thread, (void *) i) == 0);
  }
  for(i = 0; i < nthread; i++) {
    assert(pthread_join(tha[i], &value) == 0);
  }
  printf("OK; passed\n");
}
```

https://barrier-implementation.mshreya9.repl.run

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang++-7 -pthread -o main main.cpp
main.cpp:54:1: warning: control may reach end of non-void function
      [-Wreturn-type]
}
^
main.cpp:69:9: error: assigning to 'pthread_t *' (aka 'unsigned long *')
      from incompatible type 'void *'
  tha = malloc(sizeof(pthread_t) * nthread);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning and 1 error generated.
compiler exit status 1
```

# WRITTEN NOTES

25/9/19

# ACTIVITY 9

## 1) CIGARETTE - SMOKER PROBLEM

```
# include < stdio . h >
# include < sys/types . h >
# include < semaphore . h >
# include < pthread . h >
int table_used = 1, generated_item [2], generated = 0;
char* item [ ] = { "tobacco", "paper", "matches" };
sem_t table;
void * agent ( void * arg )
{
    int i, j, k=0;
    while (1)
    {
        sleep (1);
        sem_wait (&table);
        if (table_used == 1)
        {
            i = k;
            j = i+1;
            if (j == 3)
                j = 0;
            k = j;
            generated_item [0] = i;
            generated_item [1] = j;
            printf ("agent is produced %s, %s \n", item [i],
                item [j]);
            generated = 1;
            table_used = 0
    }
```

Scanned by CamScanner

```
        sem_post (&table);
    }
}

void * smoker i (int i)
{
    while (1)
    {
        sleep (1);
        sem_wait (&table);
        if (table_used == 0)
        {
            if (generated && generated_item [0] != 1 &&
generated_item [1] != i )
            {
                printf("smoker %d completed his smoking",
                                                    i);
                table_used = 1;
                generated = 0;
            }
        }
        sem_post (&table);
    }
}

main ()
{   pthread_t smker 0, smker 1, smker 2, agnt;
    sem_init (&table, 0, 1);
    pthread_create (&agnt, 0, agent, 0);
    pthread_create (&smker 0, 0, smoker i, 0);
    pthread_create (&smker 1, 0, smoker i, 1);
    pthread_create (&smker 2, 0, smoker i, 2);
    while (1);
}
```

Scanned by CamScanner

Output :

Smoker 3 >> waiting for matches & tobacco
" 2 waiting for tobacco & paper
" 1 " " matches & paper
" 4 " " " " "
" 3 " " tobacco & paper
" 0 " " matches & tobacco
Agent 0 giving out tobacco & paper
Smoker 2 now making a cigarette
Smoker 2 waiting for tobacco & paper
Agent 1 giving out matches & tobacco

(contd.)

Scanned by CamScanner

DATE / /

# 2. SLEEPING BARBER PROBLEM.

```
# include < stdio.h >
# include < sys/types.h >
# include < pthread.h >
# define seats 6

void * customerMaker ();
void * barberShop ();
void * waitingRoom ();
void CheckQueue ();

pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
int returnTime = 5, current = 0;
int main ()
{
    pthread_t barber, customerM;
    pthread_create (&customerM, 0, customerMaker, NULL);
    pthread_create (&barber, 0, barberShop, NULL);
    while (1);
}
void * customerMaker ()
{
    int i = 0;
    pthread_t customer (seats);
    while ( i < (seats) )
    {
        i++;
    sleep (1);
    pthread_create (&customer [seats], 0, waitingRoom, NULL);
    }
}
```

TNFL

Scanned by CamScanner

```
void * waitingRoom()
{
    pthread_mutex_lock(&queue_mutex);
    checkQueue();
    sleep(returnTime);
    waitingRoom();
}
void * barberShop()
{
    ~~pthread_mutex_lock(&queue_mutex);~~
    ~~checkQueue();~~
    ~~sleep(returnTime);~~
    printf("The barber has opened the store.\n");
    while(1)
    {
        if(current == 0)
        {
            printf("\t The shop is empty, barber is
                    sleeping.\n");
            printf("\t Customer has arrived, customer
                    wakes him.\n");

            printf("\t\t\t\t Barber wakes up.\n");
            current++;
        }
        else
        {
            printf("\t\t\t Barber begins cutting hair.\n");
            sleep(1);
            current--;
            printf("\t\t\t\t Hair cut complete, customer
                    leaving store.\n");
        }
    }
}
```

TNFL

Scanned by CamScanner

```
void checkQueue ()
{
    current++;
    printf("\t Customer has arrived in the waiting room
            \t\t\t\t\t\t\t %d Customers in store \n", current);

    printf("\t\t Customer checking chairs. \n");
    if (current < seats)
    {
        printf("\t\t Customer takes a seat. \n");
        pthread_mutex_unlock (&queue_mutex);
        return;
    }
    if (current >= seats)
    {
        printf("\t\t All chairs full, leaving store. \n");
        current --;
        pthread_mutex_unlock (&queue_mutex);
        return;
    }
}
```

## 3 BARRIER IMPLEMENTATION

```
#include <pthread.h>
#include "errors.h"
#include "barrier.h"

int barrier_init (barrier_t *barrier, int count)
{ int status;
```

Input

Enter the no. of customers - 2
Enter the no. of chairs = 1

Output

Customer 0 leaving for barber shop
The barber is sleepy
Customer 1 leaving for barber shop
Customer 0 arrived at barber shop
" " entering waiting room
" " walking the barber
The barber is cutting hair
Customer 1 arrived at barber shop
" " entering waiting room
The barber finished cutting hair
The barber is sleepy
Customer 0 leaving barber shop
" 1 walking to barber

Scanned by CamScanner

```
barrier -> threshold = barrier -> counter = count;
barrier -> cycle = 0;
status = pthread_ ~~cond~~ mutex_init (&barrier -> mutex, NULL)
if (status != 0)
    return status
status = pthread_cond_init (&barrier -> cv, NULL);
if (status != 0) {
        pthread_mutex_destroy (&barrier -> mutex);
        return status;
}
barrier -> valid = BARRIER_VALID;
return 0;
}

int barrier_destroy (barrier_t * barrier)
{
    int status, status2;
    if (barrier -> valid != BARRIER_VALID)
        return EINVAL;
    status = pthread_mutex_lock (&barrier -> mutex);
    if (status != 0)
        return status;
    if (barrier -> counter != barrier -> threshold) {
        pthread_mutex_unlock (&barrier -> mutex);
        return EBusy;
    }
    barrier -> valid = 0;
    status = pthread_mutex_unlock (&barrier -> mutex);
    if (status != 0)
        return status;
    status = pthread_mutex_destroy (&barrier -> mutex);
    status2 = pthread_cond_destroy (&barrier -> cv);
    return (status == 0 ? status : status2);
}
```

Scanned by CamScanner

```
int barrier_wait(barrier_t * barrier)
{
    int status, cancel, tmp, cycle;
    if (barrier -> valid = BARRIER_VALID)
        return EINVAL;
    status = pthread_mutex_lock(&barrier -> mutex);
    if (status != 0)
        return status;
    cycle = barrier -> cycle;
    if (--barrier -> counter == 0) {
        barrier -> cycle = !barrier -> cycle;
        barrier -> counter = barrier -> threshold;
        status = pthread_cond_broadcast(&barrier -> cv);
        if (status == 0)
            status = -1;
    } else {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &cancel);
        while (cycle == barrier -> cycle) {
            status = pthread_cond_wait {
                &barrier -> cv, &barrier -> mutex);
            if (status != 0) break;
        }
        pthread_setcancelstate(cancel, &tmp);
    }
    pthread_mutex_unlock(&barrier -> mutex);
    return status;
}
```

barrier.h ⇒

```
#include <pthread.h>
typedef struct barrier_tag {
    pthread_mutex_t    mutex;
    pthread_cond_t     cv;
    int                valid;
    int threshold;
    int counter;
    int cycle;
} barrier_t;
```

Scanned by CamScanner

```
#define BARRIER_VALID
#define BARRIER_INITIALIZER (cnt) \
  {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
    BARRIER_VALID, cnt, cnt, 0}
extern int barrier_init (barrier_t *barrier, int count);
extern int barrier_destroy (barrier_t *barrier);
extern int barrier_wait (barrier_t *barrier);
```

errors.h

```
#include <limits.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef DEBUG
#define DPRINTF(arg) printf arg
#else
#define DPRINTF(arg)
#endif
#define err_abort (code, text) do { \
        fprintf (stderr, "%s at \"%s\":%d: %s\n", \
            text, __FILE__, __LINE__, strerror (code)); \
    abort () ; \
  } while (0)
#define errno_abort (text) do { \
        fprintf (stderr, "%s at \"%s\":%d: %s\n", \
            text, __FILE__, __LINE__, strerror (errno)); \
    abort (); \
  } while (0)
```

DRIVER CODE :-

DRIVER CODE: -

```
int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];
  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT
                                          +1);

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

      short_ids[i] = i;

      pthread_create(&ids[i], NULL, threadfn, &short_ids[i]);
  }

  printf("main() is ready.\n");
  pthread_barrier_wait(&mybarrier);
  printf("main() is going!\n");
  for(i=0; i< THREAD_COUNT; i++) {
      pthread_join(ids[i], NULL);
  }
  pthread_barrier_destroy(&mybarrier);
  return 0;
}
```

8.5

Scanned by CamScanner

# ASSIGNMENT 8

NAME: SHREYA MAHESHWARI
REGISTRATION NUMBER: 18BCE0167

DINING-PHILOSOPHER PROBLEM

CODE:

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
	if (state[phnum] == HUNGRY
		&& state[LEFT] != EATING
		&& state[RIGHT] != EATING) {
		// state that eating
		state[phnum] = EATING;

		sleep(2);

		printf("Philosopher %d takes fork %d and %d\n",
				phnum + 1, LEFT + 1, phnum + 1);

		printf("Philosopher %d is Eating\n", phnum + 1);
```

```
                // sem_post(&S[phnum]) has no effect
                // during takefork
                // used to wake up hungry philosophers
                // during putfork
                sem_post(&S[phnum]);
        }
}

// take up chopsticks
void take_fork(int phnum)
{

        sem_wait(&mutex);

        // state that hungry
        state[phnum] = HUNGRY;

        printf("Philosopher %d is Hungry\n", phnum + 1);

        // eat if neighbours are not eating
        test(phnum);

        sem_post(&mutex);

        // if unable to eat wait to be signalled
        sem_wait(&S[phnum]);


        sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

        sem_wait(&mutex);

        // state that thinking
        state[phnum] = THINKING;

        printf("Philosopher %d putting fork %d and %d down\n",
                phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);

        test(LEFT);
        test(RIGHT);

        sem_post(&mutex);
```

```
}

void* philospher(void* num)
{

        while (1) {

                int* i = num;

                sleep(1);

                take_fork(*i);

                sleep(0);

                put_fork(*i);
        }
}

int main()
{

        int i;
        pthread_t thread_id[N];

        // initialize the semaphores
        sem_init(&mutex, 0, 1);

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

                sem_init(&S[i], 0, 0);

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

                // create philosopher processes
                pthread_create(&thread_id[i], NULL,
                                        philospher, &phil[i]);


                printf("Philosopher %d is thinking\n", i + 1);
        }

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

                pthread_join(thread_id[i], NULL);
}
```

OUTPUT:

```
Philosopher 1 is thinking                                  4,1          4%
Philosopher 2 is thinking                                  5,1          6%
Philosopher 3 is thinking                                  6,1          8%
Philosopher 4 is thinking                                  7,1          9%
Philosopher 5 is thinking                                  8,1         11%
Philosopher 1 is Hungry                                    9,1         12%
Philosopher 2 is Hungry                                    10,1        14%
Philosopher 3 is Hungry                                    11,1        16%
Philosopher 4 is Hungry                                    12,1        17%
Philosopher 5 is Hungry                                    13,1        19%
Philosopher 5 takes fork 4 and 5                           14,1        20%
Philosopher 5 is Eating                                    15,1        22%
Philosopher 5 putting fork 4 and 5 down                    16,1        24%
Philosopher 5 is thinking                                  17,1        25%
Philosopher 4 takes fork 3 and 4                           18,1        27%
Philosopher 4 is Eating                                    19,1        29%
Philosopher 1 takes fork 5 and 1                           20,1        30%
Philosopher 1 is Eating                                    21,1        32%
Philosopher 4 putting fork 3 and 4 down                    22,1        33%
Philosopher 4 is thinking                                  23,1        35%
-- INSERT --                                               24,1        37%
```

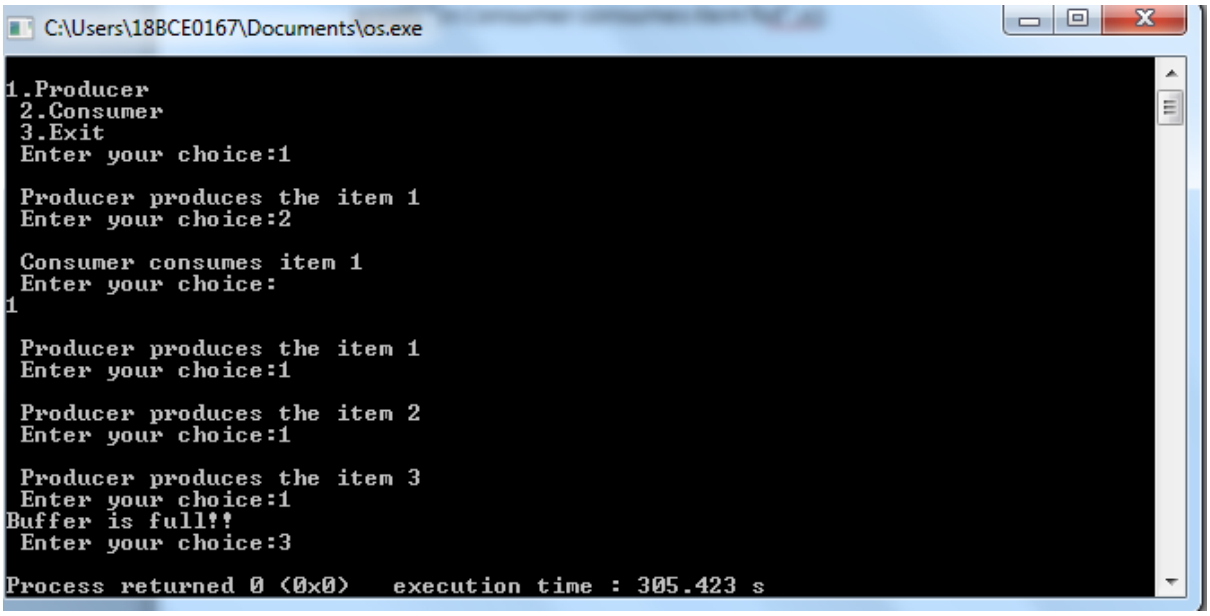
## PRODUCER CONSUMER PROBLEM

## CODE:

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

int mutex=1, full=0, empty=3, x=0;
int main()
{
   int n;
   void producer();
   void consumer();
   int wait(int);
   int signal(int);
   printf("\n1.Producer \n 2.Consumer\n 3.Exit");
   while(1)
   {
      printf("\n Enter your choice:");
      scanf("%d",&n);
      switch(n)
      {
         case 1: if((mutex==1) && (empty!=0))
            producer();
         else
            printf("Buffer is full!!");
         break;

         case 2: if((mutex==1) && (full !=0))
            consumer();
            else
               printf("Buffer is empty!!");
            break;
         case 3:
            exit(0);
            break;
      }
   }
   return 0;
}
int wait(int s)
{
```

```
    return(--s);
}
int signal(int s)
{
    return(++s);
}
void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\n Producer produces the item %d",x);
    mutex=signal(mutex);
}
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\n Consumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}
```

OUTPUT:




## READER-WRITER PROBLEM:

### CODE-

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

void *reader(void *);
void *writer(void *);
```

```
int readcount=0,writecount=0,sh_var=5,bsize[5];
sem_t x,y,z,rsem,wsem;
pthread_t r[3],w[2];

void *reader(void *i)
{
        cout << "\n------------------------ ";


cout << "\n\n reader-" << i << " is reading";

        sem_wait(&z);
        sem_wait(&rsem);
        sem_wait(&x);
        readcount++;
        if(readcount==1)
           sem_wait(&wsem);
        sem_post(&x);
        sem_post(&rsem);
        sem_post(&z);
        cout << "\nupdated value :" << sh_var;
        sem_wait(&x);
        readcount--;
        if(readcount==0)
           sem_post(&wsem);
        sem_post(&x);
}

void *writer(void *i)
{
        cout << "\n\n writer-" << i << "is writing";
        sem_wait(&y);
        writecount++;
        if(writecount==1)
        sem_wait(&rsem);
        sem_post(&y);
        sem_wait(&wsem);

        sh_var=sh_var+5;
        sem_post(&wsem);
        sem_wait(&y);
        writecount--;
        if(writecount==0)
        sem_post(&rsem);
        sem_post(&y);
}

int main()
{
        sem_init(&x,0,1);
```

```
    sem_init(&wsem,0,1);
    sem_init(&y,0,1);
    sem_init(&z,0,1);
    sem_init(&rsem,0,1);

    pthread_create(&r[0],NULL,(void *)reader,(void *)0);
    pthread_create(&w[0],NULL,(void *)writer,(void *)0);
    pthread_create(&r[1],NULL,(void *)reader,(void *)1);
    pthread_create(&r[2],NULL,(void *)reader,(void *)2);
    pthread_create(&r[3],NULL,(void *)reader,(void *)3);
    pthread_create(&w[1],NULL,(void *)writer,(void *)3);
    pthread_create(&r[4],NULL,(void *)reader,(void *)4);

    pthread_join(r[0],NULL);
    pthread_join(w[0],NULL);
    pthread_join(r[1],NULL);
    pthread_join(r[2],NULL);
    pthread_join(r[3],NULL);
    pthread_join(w[1],NULL);
    pthread_join(r[4],NULL);

    return(0);
}
```

OUTPUT-

```
-------------------------
 reader-0 is reading
updated value : 5

 writer-0 is writing
-------------------------
 reader-1 is reading
updated value : 10
-------------------------
 reader-2 is reading
updated value : 10
-------------------------
 reader-3 is reading
updated value : 10

 writer-3 is writing
-------------------------
 reader-4 is reading
```

# WRITTEN NOTES:

18/9/18

# ACTIVITY 8

Q1. PRODUCER CONSUMER PROBLEM USING SEMAPHORES

A semaphore S is an integer variable that can be accessed only through two standard operations: wait() and signal().

The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

To solve this problem, we need two counting semaphores Full and Empty. "Full keeps track of no. of items in the buffer at any given time and "Empty" keeps track of no. of unoccupied slots

Initialization of semaphores –

```
mutex = 1
Full = 0        // Initially, all slots are empty. Thus full slots are 0
Empty = n       // All slots are empty initially
```

Solution for Producer –

```
do {

    wait(empty);
    wait(mutex);

    signal(mutex);
    signal(full);
} while(true)
```

TNPL

Scanned by CamScanner

## Solution for Consumer -

```
do {

    wait ( full ) ;
    wait ( mutex ) ;

    signal ( mutex ) ;
    signal ( empty ) ;

} while ( true )
```

## CODE :-

```
# include < sys/types.h >
# include < stdio.h >

int mutex = 1, full = 0, empty = 3, x = 0 ;
int main ( )
{
    int n ;
    void producer ( ) ;
    void consumer ( ) ;
    int wait ( int ) ;
    int signal ( int ) ;
    printf ( "\n 1. Produce \n 2. Consumer \n 3. Exit" ) ;
    while ( 1 )
    {
        printf ( "\n Enter your choice : " ) ;
        scanf ( "%d" , &n ) ;
        switch ( n )
        {
```

Scanned by CamScanner

```
        case 1 : if ((mutex == 1) && (empty != 0))
                    produce();
                 else
                    printf("Buffer is full!!");
                 break;

        case 2 : if ((mutex == 1) && (full != 0))
                    consume();
                 else
                    printf("Buffer is empty!!");
                 break;
        case 3 :
                 exit(0);
                 break;
            }
    }
    return 0;
}
int wait (int s)
{
    return (--s);
}
int signal (int s)
{
    return (++s);
}
void produce ()
{
    mutex = wait(mutex);
    full = signal(full);
    empty = wait(empty);
    x++;
    printf("\n Producer produces the item %d", x);
    mutex = signal(mutex);
}
```

Scanned by CamScanner

O/P

1. Produce
2. Consume
3. Exit

Enter your choice : 1

Produce produces the item 1
Enter your choice : 2

Consumer consumes item 1
Enter your choice : 2
Buffer is empty !!

Enter your choice : 1

Produce produces the item 1
Enter your choice : 1

Produce produces the item 2
Enter your choice : 1

Produce produces the item 3
Enter your choice : 1
Buffer is full !!
Enter your choice : 3

Scanned by CamScanner

```
void consume ()
{
    mutex = wait (mutex);
    full = wait (full);
    empty = signal (empty);
    printf ("\n Consume consumes item %d", x);
    x - 1;
    mutex = signal (mutex);
}
```

2. <u>DINING PHILOSOPHER PROBLEM</u>

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

# define N 5
# define THINKING 2
# define HUNGRY 1
# define EATING 0
# define LEFT (phnum + 4) % N
# define RIGHT (phnum + 1) % N

int state [N];
int phil [N];

void test (int phnum)
{
    if (state [phnum] == HUNGRY &&
        state [LEFT] != EATING
        && state [RIGHT] != EATING) {
            state [phnum] = EATING;
```

TNPL

Scanned by CamScanner

```
            sleep(2);
            printf("Philosopher %d takes fork %d
                and %d\n", phnum+1, LEFT+1,
                phnum+1);
printf("Philosopher %d is Eating\n", phnum+1);
    sem_post(&S[phnum]);
  }
}

void take_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum+1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d & %d
        down\n", phnum+1, LEFT+1, phnum+1);
    printf("Philosopher %d is thinking\n", phnum+1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
```

O/P :

Philosopher 1 is thinking
" 2 is thinking
" 3 is thinking
" 4 is thinking
" 5 is thinking
" 1 is hungry
" 3 is hungry
" 4 is hungry
" 5 is hungry
" 5 takes fork 4 & 5
" 5 is eating
" 2 is hungry
" 2 takes fork 1 & 2
" 2 is eating
" 5 putting fork 4 & 5 down
" 5 is thinking
" 4 takes fork 3 & 4
" 4 is eating
" 2 is putting fork 1 & 2 down
" 2 is thinking
" 1 takes fork 5 & 1
" 1 is eating
" 5 is hungry
" 4 putting fork 3 & 4 down

Scanned by CamScanner

DATE / /

```
void * philosopher (void * num)
{
    while (1) {
            int * i = num ;
            sleep (1) ;
            take_fork (* i) ;
            sleep (0) ;
            put_fork (* i) ;
        }
}
int main ()
{
    int i ;
    pthread_t thread_id [N] ;

    sem_init (& mutex , 0 , 1) ;
    for ( i=0 ; i<N ; i++ )
        sem_init (& s[i] , 0 , 0) ;

    for (i=0 ; i<N ; i++ ) {
        pthread_create (& thread_id [i] , NULL ,
                philosopher , & phil [i] ) ;

    printf (" Philosopher %d is thinking \n" , i+1) ;
}

for (i=0 ; i<N ; i++ )
    pthread_join (thread_id [i] , NULL) ;

}
```

TNFL

Scanned by CamScanner

3. READER WRITER PROBLEM

```
# include <sys/types.h>
# include <pthread.h>
# include <semaphore.h>

sem_t mutex, writeblock;
int data = 0, rcount = 0;

void * reader (void * arg)
{
    int f;
    f = ((int) arg);
    sem_wait (& mutex);
    rcount = rcount + 1;
    if (rcount == 1)
    sem_wait (& writeblock);
    sem_post (& mutex);
    printf ("Data read by the reader %d is %d\n", f, data);
    sleep (1);
    sem_wait (& mutex);
    rcount = rcount - 1;
    if (rcount == 0)
        sem_post (& writeblock);
    sem_post (& mutex);
}
void * writer (void * arg)
{
    int f;
    f = ((int) arg);
    sem_wait (& writeblock);
    data ++;
```

Scanned by CamScanner

O/P

reader -0 is ready
updated value : 5

writer -0 is writing
- - - - - - - - -

reader -1 is ready
updated value : 10
- - - - - - - - -

reader -2 is ready
updated value : 10
- - - - - - - - -

reader -3 is ready
updated value : 10

writer -3 is writing
- - - - - - - - -

reader -4 is ready.

Scanned by CamScanner

```
    printf("Data written by the writer %d is %d\n",
                    i, data);
    sleep(1);
    sem_post(&writerblock);
}
int main()
{
    int i, b;
    pthread_t rtid[5], wtid[5];
    sem_init(&mutex, 0, 1);
    sem_init(&writeblock, 0, 1);
    for (i=0; i<=2; i++)
    {
        pthread_create(&wtid[i], NULL, writer, (void*)i);
        pthread_create(&rtid[i], NULL, reader, (void*)i);
    }
    for (i=0; i<=2; i++)
    {
        pthread_join(wtid[i], NULL);
        pthread_join(rtid[i], NULL);
    }
    return 0;
}
```

19/9/19

9