OPERATING SYSTEM LAB

(KCS451)

SUBMITTED BY
ROHIT KUMAR
1900320100131

SUBMITTED TO: NIDHI SINGH

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
ABES ENGINEERING COLLEGE
GHAZIABAD

INDEX

Expt./ Job No.	Experiment / Job Details	Pre-Lab Work	Implementation	Output	Viva	Total
1	Program to Implement FCFS					
2	Program to Implement SJF					
3	Program to Implement Priority Scheduling					
4	Program to Implement Round Robin					
5	Program to Implement SRTF					
6	Program to Implement Bankers Algorithms					
7	Program to Implement Memory Allocation Scheme					
8	Program to Implement Page Replacement Algorithms					
9	Program to Implement Disk Scheduling Algorithms					

Dated: 14/04/2021

Experiment No. – 01 (FCFS)

Objective:

Calculate average waiting time and average turnaround time using FCFS scheduling algorithm using following input.

Design generic code that will run both input

Program:

```
// Program to Implement First Come First Serve Algorithm
// Rohit Kumar | 1900320100131
#include<stdio.h>
void main()
{
      // STDIN
    int n;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int AT[n],BT[n],CT[n],WT[n],TAT[n];
    printf("Enter the AT & BT of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d",&AT[i],&BT[i]);
    }
    // PROCESSING
    float avgWT = 0.0;
    float avgTAT = 0.0;
    // Calculation for 1st Process
    CT[0] = AT[0] + BT[0];
    TAT[0] = CT[0] - AT[0];
    WT[0] = TAT[0] - BT[0];
    // Loop for Calculation
    for(int i=1; i<n; i++)
    {
        if(CT[i-1]>AT[i])// if CT of 1st Process is more then than Arrival of
Second Process
            CT[i] = CT[i-1] + BT[i];
        else // if CT of 1st Process is less then than Arrival of Second Process
            CT[i] = AT[i] + BT[i];
        TAT[i] = CT[i] - AT[i];
        WT[i] = TAT[i] - BT[i];
        avgTAT += TAT[i];
        avgWT += WT[i];
    }
    avgWT = avgWT/n;
    avgTAT = avgTAT/n;
    // STDOUT
    printf("\n\tAT\tBT\tWT\tTAT\n");
    for(int i=0;i<n;i++)
        printf("P[%d]\t%d\t%d\t%d\t%d\n",i,AT[i],BT[i],WT[i],TAT[i]);
    printf("\nAverage WT : %f Units\n",avgWT);
    printf("Average TAT : %f Units",avgTAT);
}
```

Input/Output:

Q1:

Process	Arrival Time	Burst Time	Waiting Time	TAT
P1	0	4	0	4
P2	2	3	2	5
P3	6	5	1	6
P4	8	2	4	6
P5	9	1	5	6
P6	10	3	5	8

```
      AT    ST    WT    TAT
P1    0     4     0     4
P2    2     3     2     5
P3    6     5     1     6
P4    8     2     4     6
P5    9     1     5     6
P6    10    3     5     8
Average Wait time =2.83 unit
Average Turn Around time =5.83 unit
```

Average waiting time = **2.8 Units.**
Average turnaround time = **5.83 Units.**

Q2

Process	Arrival Time	Burst Time	Waiting Time	TAT
P1	6	1	0	1
P2	4	3	0	3
P3	2	2	1	3
P4	8	1	1	2
P5	15	3	0	3

```
      AT    ST    WT    TAT
P1    2     1     0     1
P2    4     3     0     3
P3    6     2     1     3
P4    8     1     1     2
P5    15    3     0     3
Average Wait time =0.4 unit
Average Turn Around time =2.4 unit
```

Average waiting time = **0.4 Units.**
Average turnaround time **= 2.4 Units.**

Result:

We have Successfully implemented FCFS.

Dated: 14/04/2021

Experiment No. – 02 (SJF)

Objective:

Calculate the average waiting time and average turnaround time using SJF (Non-Primitive Scheduling)

Program:

```
// Program to Implement Shortest Job First (SJF)
// Rohit Kumar | 1900320100131
#include<stdio.h>

void main()
{
      // STDIN
    int n;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int AT[n],BT[n],CT[n],WT[n],TAT[n];
    printf("Enter the AT & BT of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d",&AT[i],&BT[i]);
    }
    // PROCESSING
    float AvgWT = 0.0;
    float AvgTAT = 0.0;
    int flag[n];
    //Initialize Flag of Every process to Zero
    for(int i=0;i<n;i++)
        flag[i]=0;

    int time=AT[0],min,k;
    for(int j=0;j<n;j++)
    {
        min=100;
        k=0;
        //Loop to Find Less Burst Time Process by Checking Flag
        for(int i=0;i<n;i++)
        {
            if(flag[i]==0)
            {
                if(AT[i]<=time)
                {
                    if(BT[i]<min)
                    {
                        min=BT[i];
                        k=i;
                    }
                }
            }
        }
        CT[k]=min+time;
        time=time+min;
        flag[k]=1;
```

```
        WT[k]=CT[k]-(BT[k]+AT[k]);
        TAT[k]=CT[k]-AT[k];
        AvgTAT += TAT[k];
        AvgWT += WT[k];
    }

    AvgWT = AvgWT/n;
    AvgTAT = AvgTAT/n;
      // STDOUT
    printf("\n\tAT\tBT\tWT\tTAT\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t%d\t%d\t%d\t%d\n",i,AT[i],BT[i],WT[i],TAT[i]);

    }
    printf("\nAverage WT : %f Units\n",AvgWT);
    printf("Average TAT : %f Units",AvgTAT);
}
```

Input/Output:

```
No of Process  :  5
Enter the AT & BT of Process
P[0]    0       4
P[1]    4       1
P[2]    1       2
P[3]    6       4
P[4]    8       2

        AT      BT      WT      TAT
P[0]    0       4       0       4
P[1]    4       1       0       1
P[2]    1       2       4       6
P[3]    6       4       1       5
P[4]    8       2       3       5

Average WT : 1.600000 Units
Average TAT : 4.200000 Units
```

Result:

We have Successfully implemented SJF.

Dated: 04/05/2021

Experiment No. – 03 (Priority)

Experiment No. 3(a)

Objective:

Calculate avgWT and avgTAT using Priority Scheduling (Preemptive).

Program:

```
#include<stdio.h>
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void main()
{
    int n;
    float avgWT = 0.0;
    float avgTAT = 0.0;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int AT[n],BT[n],PR[n],CT[n],WT[n],TAT[n],RBT[n];
    printf("Enter the AT,BT and Priority of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d %d ",&AT[i],&BT[i],&PR[i]);
    }
    // Sort the Input
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n-1-i;j++)
        {
            if(*(AT+j)>*(AT+j+1))
            {
                swap((AT+j),(AT+j+1));
                swap((BT+j),(BT+j+1));
                swap((PR+j),(PR+j+1));
            }
        }
    }
    int sum=0;
    for(int i=0;i<n;i++)
    {
        RBT[i]=BT[i];
        sum+=BT[i];
    }
    int time=AT[0],min,k;
    for(int i=0;i<sum;i++)
    {
        min=100;
        k=0;
        for(int j=0;j<n;j++)
        {
            if(RBT[j]!=0)
            {
                if(AT[j]<=time)
```

```
                {
                    if(PR[j]<min)
                    {
                        min=PR[j];
                        k=j;
                    }
                }
            }
        }
        RBT[k]=RBT[k]-1;
        time=time+1;
        CT[k]=time;
    }
    for(int i=0;i<n;i++)
    {
        WT[i]=CT[i]-AT[i]-BT[i];
        TAT[i]=CT[i]-AT[i];
        avgTAT += TAT[i];
        avgWT += WT[i];
    }
    avgWT = avgWT/n;
    avgTAT = avgTAT/n;
    printf("\n   \tAT\tBT\tPR\tWT\tTAT\n");
    for(int i=0;i<n;i++)
        printf("P[%d]\t%d\t%d\t%d\t%d\t%d\n",i,AT[i],BT[i],PR[i],WT[i],TAT[i]);
    printf("\nAverage Waiting Time is %f Units\n",avgWT);
    printf("Average Turn Around Time is %f Units",avgTAT);
}
```

Input/Output:

```
No of Process  :  4
Enter the AT,BT and Priority of Process
P[0]    0       5       1
P[1]    3       5       1
P[2]    4       5       3
P[3]    5       6       2

        AT      BT      PR      WT      TAT
P[0]    0       5       1       0       5
P[1]    3       5       1       2       7
P[2]    4       5       3       12      17
P[3]    5       6       2       5       11

Average Waiting Time is 4.750000 Units
Average Turn Around Time is 10.000000 Units
```

Result:

We have verified **Priority Scheduling (Preemptive)** Successfully.

Experiment No. 3(b)

Objective:

Calculate avgWT and avgTAT using Priority Scheduling (Non - Preemptive).

Program:

```
#include<stdio.h>
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void main()
{
    int n;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int AT[n],BT[n],PR[n],CT[n],WT[n],TAT[n],RBT[n];
    printf("Enter the AT,BT and Priority of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d%d",&AT[i],&BT[i],&PR[i]);
    }
    float avgWT = 0.0;
    float avgTAT = 0.0;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n-1-i;j++)
        {
            if(*(AT+j)>*(AT+j+1))
            {
                swap((AT+j),(AT+j+1));
                swap((BT+j),(BT+j+1));
                swap((PR+j),(PR+j+1));
            }
        }
    }
    int flag[n];
    for(int i=0;i<n;i++)
    {
        flag[i]=0;
    }
    int time=AT[0],min,k;
    for(int j=0;j<n;j++)
    {
        min=100;
        k=0;
        for(int i=0;i<n;i++)
        {
            if(flag[i]==0)
            {
                if(AT[i]<=time)
                {
                    if(PR[i]<min)
                    {
                        min=PR[i];
                        k=i;
                    }
```

```
                }
            }
        }
        CT[k]=BT[k]+time;
        time=time+BT[k];
        flag[k]=1;
        WT[k]=CT[k]-(BT[k]+AT[k]);
        TAT[k]=CT[k]-AT[k];
        avgTAT += TAT[k];
        avgWT += WT[k];
    }

    avgWT = avgWT/n;
    avgTAT = avgTAT/n;

    printf("\n  \tAT\tBT\tPR\tWT\tTAT\n");
    for(int i=0;i<n;i++)
        printf("P[%d]\t%d\t%d\t%d\t%d\t%d\n",I,AT[i],BT[i],PR[i],WT[i],TAT[i]);
    printf("\nAverage Waiting Time is %f Units\n",avgWT);
    printf("Average Turn Around Time is %f Units",avgTAT);
}
```

Input/Output:

```
No of Process  :  4
Enter the AT,BT and Priority of Process
P[0]    0       4       1
P[1]    2       5       1
P[2]    3       4       2
P[3]    3       4       1

        AT      BT      PR      WT      TAT
P[0]    0       4       1       0       4
P[1]    2       5       1       2       7
P[2]    3       4       2       10      14
P[3]    3       4       1       6       10

Average Waiting Time is 4.500000 Units
Average Turn Around Time is 8.750000 Units
```

Result:

We have verified **Priority Scheduling (Non - Preemptive)** Successfully.

Dated: 04/05/2021

Experiment No. – 04 (Round Robin)

Objective:
Calculate avgWT and avgTAT using Round Robin

Program:

```
#include<stdio.h>
void main()
{
    int n,q;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int at[n],bt[n],ct[n],wt[n],tat[n],rbt[n];
    printf("Enter the AT & BT of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d",&at[i],&bt[i]);
    }
    printf("\nEnter Quantum  :  ");
    scanf("%d",&q);
    float AvgWT = 0.0;
    float AvgTAT = 0.0;
    int sum=0;
    for(int i=0;i<n;i++)
    {
        rbt[i]=bt[i];
        sum=sum+bt[i];
    }
    int time=0;
    while(time<sum)
    {
        for(int i=0;i<n;i++)
        {
            if(rbt[i]!=0)
            {
                if(rbt[i]>=q)
                {
                    rbt[i]=rbt[i]-q;
                    time=time+q;
                    if(rbt[i]==0)
                        ct[i]=time;
                }
                else
                {
                    time=time+rbt[i];
                    rbt[i]=0;
                    ct[i]=time;
                }
            }
        }
    }
    for(int i=0;i<n;i++)
    {
        wt[i]=ct[i]-bt[i];
        tat[i]=ct[i];
```

```
        AvgTAT += tat[i];
        AvgWT += wt[i];
    }
    AvgWT = AvgWT/n;
    AvgTAT = AvgTAT/n;
    printf("\n   \tAT\tBT\tWT\tTAT\n");
    for(int i=0;i<n;i++)
        printf("P[%d]\t%d\t%d\t%d\t%d\n",i,at[i],bt[i],wt[i],tat[i]);
    printf("\nAverage Waiting Time is %f Units\n",AvgWT);
    printf("Average Turn Around Time is %f Units",AvgTAT);
}
```

Input/Output :

```
No of Process  :  5
Enter the AT & BT of Process
P[0]    0       4
P[1]    3       6
P[2]    6       7
P[3]    0       8
P[4]    2       9

Enter Quantum  :  2

        AT      BT      WT      TAT
P[0]    0       4       8       12
P[1]    3       6       16      22
P[2]    6       7       22      29
P[3]    0       8       23      31
P[4]    2       9       25      34

Average Waiting Time is 18.799999 Units
Average Turn Around Time is 25.600000 Units
```

Result:

We have verified **Round Robin** Successfully.

Experiment No. – 05 (SRTF)

Objective:

Calculate avgWT and avgTAT using SRTF

Program:

```
#include<stdio.h>
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void main()
{
    int n,q;
    printf("No of Process  :  ");
    scanf("%d",&n);
    int at[n],bt[n],ct[n],wt[n],tat[n],rbt[n];
    printf("Enter the AT & BT of Process\n");
    for(int i=0;i<n;i++)
    {
        printf("P[%d]\t",i);
        scanf("%d%d",&at[i],&bt[i]);
    }
    float AvgWT = 0.0;
    float AvgTAT = 0.0;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n-1-i;j++)
        {
            if(*(at+j)>*(at+j+1))
            {
                swap((at+j),(at+j+1));
                swap((bt+j),(bt+j+1));
            }
        }
    }
    int sum=0;
    for(int i=0;i<n;i++)
    {
        rbt[i]=bt[i];
        sum+=bt[i];
    }
    int time=at[0],min,k;
    for(int i=0;i<sum;i++)
    {
        min=100;
        k=0;
        for(int j=0;j<n;j++)
        {
            if(rbt[j]!=0)
            {
                if(at[j]<=time)
                {
                    if(rbt[j]<min)
```

```
                    {
                        min=rbt[j];
                        k=j;
                    }
                }
            }
        }
        rbt[k]=rbt[k]-1;
        time=time+1;
        ct[k]=time;
    }
    for(int i=0;i<n;i++)
    {
        wt[i]=ct[i]-at[i]-bt[i];
        tat[i]=ct[i]-at[i];
        AvgTAT += tat[i];
        AvgWT += wt[i];
    }
    printf("\n   \tAT\tBT\tWT\tTAT\n");
    for(int i=0;i<n;i++)
        printf("P[%d]\t%d\t%d\t%d\t%d\n",i,at[i],bt[i],wt[i],tat[i]);
    printf("\nAverage Waiting Time is %f Units \n",AvgWT/n);
    printf("Average Turn Around Time is %f Units",AvgTAT/n);
}
```

Input/Output:

```
No of Process  :  5
Enter the AT & BT of Process
P[0]    2       7
P[1]    0       2
P[2]    1       9
P[3]    2       8
P[4]    7       9

        AT      BT      WT      TAT
P[0]    0       2       0       2
P[1]    1       9       16      25
P[2]    2       7       0       7
P[3]    2       8       7       15
P[4]    7       9       19      28

Average Waiting Time is 8.400000 Units
Average Turn Around Time is 15.400000 Units
```

Result:

We have verified **SRTF** Successfully.

Dated: XX/XX/2021

Experiment No. 06 (Bankers Algorithms)

Objective: Implementation of Bankers Algorithm

Program:

```
# Program to Implement Bankers Algorithms
# Rohit Kumar | 1900320100131
def calNeed(max, allocation):
    need = []
    for i in range(5):
        l = []
        for j in range(3):
            l.append(max[i][j]-allocation[i][j])
        need.append(l)
    return need


def safe_seq(availabe, max, allocatin, need):
    n = len(allocatin)
    work = availabe.copy()
    finish = [False]*n
    ans = []
    k = 0
    while k < len(allocatin):
        for i in range(len(allocatin)):
            if finish[i] == False:
                if need[i][0] <= work[0] and need[i][1] <= work[1] and need[i][2] <= wor
k[2]:
                    ans.append(f"P{i}")
                    for j in range(3):
                        allocatin[i][j] += need[i][j]
                        work[j] -= need[i][j]
                        need[i][j] = max[i][j]-allocatin[i][j]
                    for j in range(3):
                        work[j] += allocatin[i][j]
                    finish[i] = True
        k += 1
    return ans


if __name__ == "__main__":
    availabe = [3, 3, 2]
    max = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
    allocation = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
    need = calNeed(max, allocation)
    safeSeq=[]
    print("SAFE SEQUENCE", *safe_seq(availabe, max, allocation, need))

    while True:
        p = int(input("ENTER PROCESS ID  :: "))
        req = list(map(int, input("ENTER REQUEST  :: ").split()))
        flag = False
        if req[0] <= need[p][0] and req[1] <= need[p][1] and req[2] <= need[p][2
```

```
        if req[0] <= availabe[0] and req[1] <= availabe[1] and req[2] <= availabe[2]]:
            for i in range(len(req)):
                availabe[i] -= req[i]

                allocation[p][i] += req[i]
                need[p][i] -= req[i]
            safeSeq = safe_seq(availabe, max, allocation, need)
            flag = True
    if len(safeSeq) == 5:
        print("SAFE SEQUENCE ", *safeSeq)
    else:
        for i in range(len(req)):
            availabe[i] += req[i]
            allocation[p][i] -= req[i]
            need[p][i] += req[i]
    if flag == False:
        print("SAFE SEQUENCE NOT OBTAINED")

    con = input("WANT TO CONTINUE TYPE Y/N")
    if con == "n" or con == "N":
        break
```

Input/Output:

```
SAFE SEQUENCE P1 P3 P4 P0 P2
```

```
ENTER PROCESS ID  :: 4
ENTER REQUEST  :: 3 3 0
SAFE SEQUENCE NOT OBTAINED
WANT TO CONTINUE TYPE Y/Ny
ENTER PROCESS ID  :: 0
ENTER REQUEST  :: 0 2 0
SAFE SEQUENCE  P3 P1 P2 P0 P4
WANT TO CONTINUE TYPE Y/Nn
```

Dated: 15/06/2021

Experiment No. – 07 (Memory Allocation)

Objective:

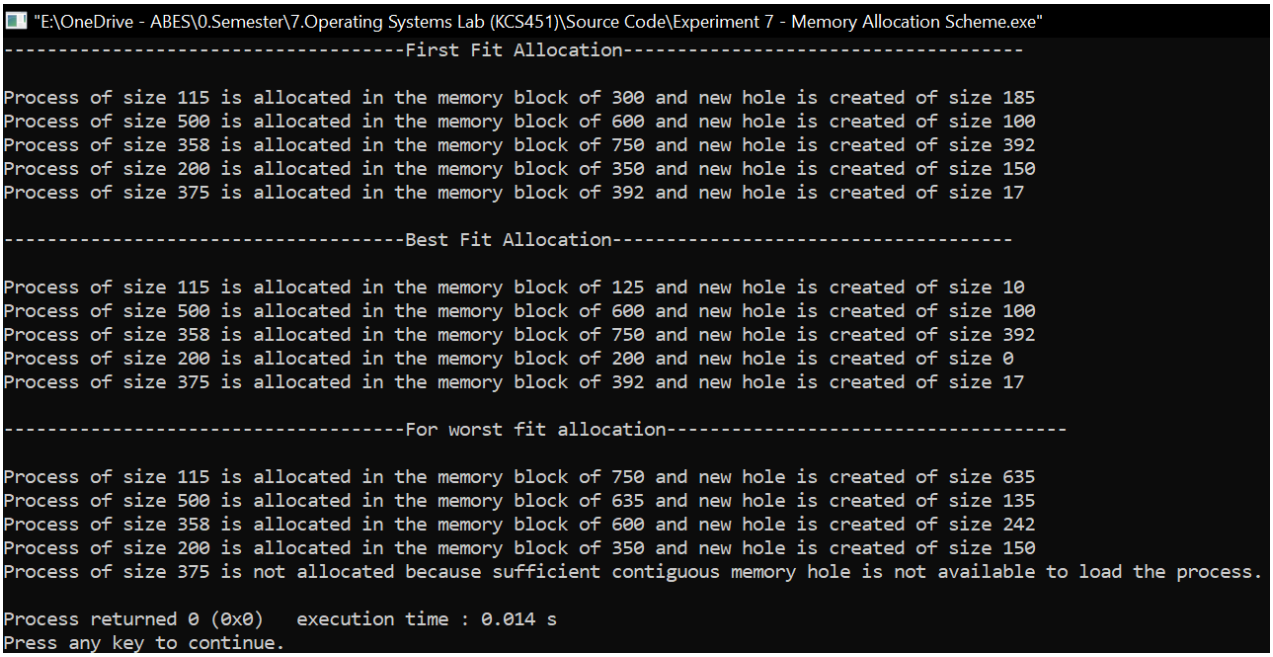
Implement memory allocation for six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?
And print the output in following manner for each process under every scheme:

Program:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i, j, flag, x, y, min, alloc[100], k = 0, index, bestIdx, size1[100],
size2[100];
    int size[6] = {300, 600, 350, 200, 750, 125};
    int process[5] = {115, 500, 358, 200, 375};
    for (i = 0; i < 6; i++)
    {
        size1[i] = size[i];
        size2[i] = size[i];
    }
    printf("-------------------------------------First Fit Allocation-----------
-------------------------\n\n");
    for (j = 0; j < 5; j++)
    {
        flag = 0;
        for (i = 0; i < 6; i++)
        {
            if (size[i] >= process[j])
            {
                flag = 1;
                x = size[i];
                size[i] = size[i] - process[j];
                y = size[i];
                break;
            }
        }
        if (flag == 1)
        {
            printf("Process of size %d is allocated in the memory block of %d
and new hole is created of size %d\n", process[j], x, y);
        }
        else
            printf("Process of size %d is not allocated because sufficient
contiguous memory hole is not available to load the process.\n", process[j]);
    }
    printf("\n-------------------------------------Best Fit Allocation----------
-------------------------\n\n");
    for (j = 0; j < 5; j++)
    {
        index = -1;
        for (i = 0; i < 6; i++)
        {
            if (size1[i] >= process[j])
            {
                if (index == -1)
                    index = i;
```

```
                else if (size1[index] > size1[i])
                    index = i;
            }
        }
        if (index != -1)
        {
            x = size1[index];
            size1[index] -= process[j];
            y = size1[index];
            printf("Process of size %d is allocated in the memory block of %d
and new hole is created of size %d\n", process[j], x, y);
        }
        else
            printf("Process of size %d is not allocated because sufficient
contiguous memory hole is not available to load the process.\n", process[j]);
    }
    printf("\n-------------------------------------For worst fit allocation-----
--------------------------------\n\n");
    for (j = 0; j < 5; j++)
    {
        index = -1;
        for (i = 0; i < 6; i++)
        {
            if (size2[i] >= process[j])
            {
                if (index == -1)
                    index = i;
                else if (size2[index] < size2[i])
                    index = i;
            }
        }
        if (index != -1)
        {
            x = size2[index];
            size2[index] -= process[j];
            y = size2[index];
            printf("Process of size %d is allocated in the memory block of %d
and new hole is created of size %d\n", process[j], x, y);
        }
        else
            printf("Process of size %d is not allocated because sufficient
contiguous memory hole is not available to load the process.\n", process[j]);
    }
}
```

Input/Output:



Dated: 06/07/2021

Experiment No. – 08 (Page Replacement Algorithms)

Objective:

Consider the following page reference string:
7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

Programs:

```
#include<stdio.h>

int minimum(int arr[], int n)
{
    int i, m = 0;
    for(i=0; i<n; i++)
    {
        if(arr[i] < arr[m])
            m = i;
    }
    return m;
}

int maximum(int arr[], int n)
{
    int i, m = 0;
    for(i=0; i<n; i++)
    {
        if(arr[i] > arr[m])
            m = i;
    }
    return m;
}

void FIFO(int str[], int frames_no, int total_pages)
{
    int i, j, flag, last_in = -1, page_fault = 0, frames[frames_no];
    for(i=0; i<total_pages; i++)
    {
        flag = 0;
        for(j=0; j<frames_no; j++)
        {
            if(str[i]==frames[j])
                flag = 1;
        }
        if(flag == 0)
        {
            last_in = (last_in+1)%frames_no;
            frames[last_in] = str[i];
            page_fault++;
        }
```

```
    }

printf("****************************(FIFO)*****************************\n");
    printf("Final Frames :\n");
    for(i=0; i<frames_no; i++)
        printf("%d\t",frames[i]);

    printf("No of Page faults : %d\n",page_fault);
}

void LRU(int str[], int frames_no, int total_pages)
{
    int i, frames[frames_no], page_faults = 0, full = 0, age[frames_no], j,
flag, leastRecent;
    for(i=0; i<total_pages; i++)
    {
        if(full < frames_no)
        {
            frames[full] = str[i];
            age[full] = i;
            full++;
            page_faults++;
        }
        else
        {
            flag = 0;
            for(j=0; j<frames_no; j++)
            {
                if(str[i]==frames[j])
                    {
                        flag = 1;
                        age[j] = i;
                    }
            }
            if(flag == 0)
            {
                leastRecent = minimum(age, frames_no);
                frames[leastRecent] = str[i];
                age[leastRecent] = i;
                page_faults++;
            }
        }
    }

printf("\n****************************(LRU)*****************************\n");
    printf("Final Frames :\n");
    for(i=0; i<frames_no; i++)
        printf("%d\t",frames[i]);
    printf("No of Page faults : %d\n",page_faults);
}

void Optimal(int str[], int frames_no, int total_pages)
{
    int i, frames[frames_no], page_faults = 0, full = 0, j, flag,
freq[frames_no], optimal, k;
    for(i=0; i<total_pages; i++)
    {
        if(full < frames_no)
        {
            frames[full] = str[i];
            full++;
```

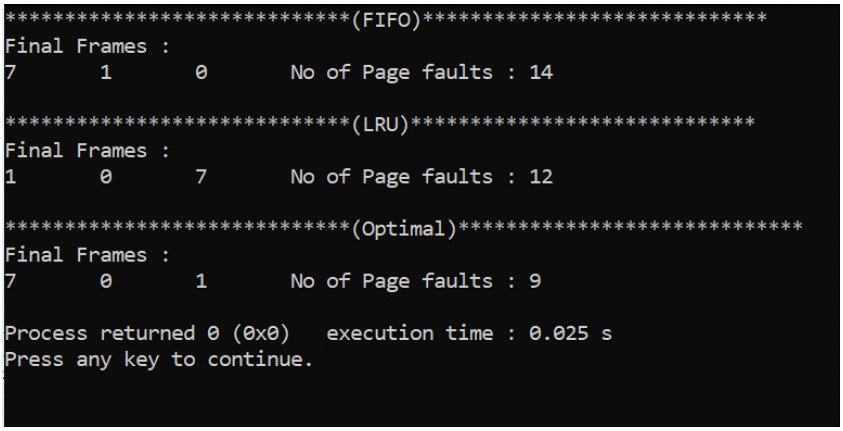
```
            page_faults++;
        }
        else
        {
            flag = 0;
            for(j=0; j<frames_no; j++)
            {
                if(str[i]==frames[j])
                    {
                        flag = 1;
                    }
            }
            if(flag == 0)
            {
                for(j=0; j<frames_no; j++)
                {
                    int f = 0;
                    for(k=i; k<total_pages; k++)
                    {
                        if(str[k] == frames[j])
                            {
                                freq[j] = k;
                                f = 1;
                                break;
                            }
                    }
                    if(f==0)
                        freq[j] = 100;
                }
                optimal = maximum(freq, frames_no);
                frames[optimal] = str[i];
                page_faults++;
            }
        }
    }

printf("\n*************************(Optimal)*************************\n"
);
    printf("Final Frames :\n");
    for(i=0; i<frames_no; i++)
        printf("%d\t",frames[i]);

    printf("No of Page faults : %d\n",page_faults);
}

int main(void)
{
    int page[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};
    int total_pages = 20;
    int frames_no = 3;
    FIFO(page, frames_no, total_pages);
    LRU(page, frames_no, total_pages);
    Optimal(page, frames_no, total_pages);
}
```

Output:


**********(FIFO)**********
Final Frames :
7 1 0 No of Page faults : 14
**********(LRU)**********
Final Frames :
1 0 7 No of Page faults : 12
**********(Optimal)**********
Final Frames :
7 0 1 No of Page faults : 9
Process returned 0 (0x0) execution time : 0.025 s
Press any key to continue.

Dated: 06/07/2021

Experiment No. – 09 (Disk Scheduling Algorithms)

Objective:

Consider total number of cylinders as 200. Head pointers processing the cylinder 53 currently after processing the request of cylinder no 5.
New request for processing are : 98, 183, 37, 122, 14, 124, 65, 67. Find the number of cylinder crossed while processing this request using following scheduling

- **FIFO**
- **SSTF**
- **SCAN**
- **C-SCAN**
- **LOOK**
- **C-LOOK**

Note: Implement any three of them and upload the code along with output snapshot.

Program & Output:

1.FCFS

```
# FCFS Disk Scheduling
process=[98, 183, 37, 122, 14, 124, 65, 67]
n=abs(53-process[0])
for i in range(len(process)-1):
    n+=abs(process[i]-process[i+1])
print("Total Cylinders Crossed : ",n)
```

```
Total Cylinders Crossed :  640
```

2.SSTF

```
# SCAN Disk Scheduling
process=[98, 183, 37, 122, 14, 124, 65, 67]
cylinder_size=200
starting=53
starting_index=4
cycle=starting-process[starting_index]
for i in range(starting_index,len(process)-1):
    cycle += abs(process[i] - process[i + 1])
cycle+=cylinder_size-process[-1]
cycle+=cylinder_size-process[starting_index-1]
for i in range(starting_index-1,0,-1):
    cycle+=abs(process[i]-process[i-1])
print("Total Cylinders Crossed : ",cycle)
```

```
Total Cylinders Crossed :  737
```

3.SCAN

```
# SSTF Disk Scheduling
process=[98, 183, 37, 122, 14, 124, 65, 67]
check=[0]*len(process)
starting_index=4
cycle=53-min(process[starting_index],process[starting_index-1])
if process[starting_index]>process[starting_index-1]:
    i=starting_index-1
else:
    i=starting_index
check[i]=1
def returnL(index):
    for i in range(index,-1,-1):
        if check[i]==0:
            return i
    return 10000
def returnR(index):
    for i in range(index,len(check)):
        if check[i]==0:
            return i
        return returnL(index)
while(check.count(1)!=len(process)):
    l=returnL(i)
    r=returnR(i)
    cycle+= abs(process[i] - min(process[l],process[r]))
    check[i]=1
    if process[l] > process[r]:
        i = r
    else:
        i = l
print("Total Cylinders Crossed : ",cycle)
```

```
Total Cylinders Crossed :  581
```