

International Islamic University Chittagong

OPERATING SYSTEMS LAB MANUAL

Course Name: Operating Systems Lab
Course Code: CSE-3632



Submitted To
Mohammad Zainal Abedin
Assistant Professor
Dept. of Computer Science and Engineering
IIUC

Submitted By

Name: Tehsim Fariha
ID: C193207
Semester: 6th
Section: 6AF

PROGRAM OUTCOMES

PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.
PROGRAM EDUCATIONAL OUTCOMES	
PEO1	To induce strong foundation in mathematical and core concepts, which enable them to participate in research , in the field of computer science.
PEO2	To be able to become the part of application development and problem solving by learning the computer programming methods, of the industry and related domains.
PEO3	To Gain the multidisciplinary knowledge by understanding the scope of association of computer science engineering discipline with other engineering disciplines.
PEO4	To improve the communication skills, soft skills, organizing skills which build the professional qualities, there by understanding the social responsibilities and ethical attitude.

OPERATING SYSTEMS LAB SYLLABUS

S. No.	List of Experiments
1	Implementation of Producer-Consumer Problem
2	Simulate the following CPU scheduling algorithms: a) FCFS b) SJF c) Round Robin d) Priority
3	Simulate Bankers Algorithm for Deadlock Avoidance
4	Simulate all page replacement Algorithms: a) FIFO b) Optimal c) LRU
5	Write a C program to simulate the following contiguous memory allocation techniques: a) Worst-fit b) Best fit c) First fit

COURSE OBJECTIVES:

This lab complements the operating systems course. With this course Students are able to:

COB 1: Implement practical experience with designing and implementing concepts of operating systems.

COB 2: write the code to implement and modify various concepts in Operating Systems using Linux environment.

COB 3: Implement Various CPU Scheduling Algorithms.

COB 4: Implement Various Page replacement Algorithms.

COB 5: Understand and Implement Banker's Algorithm

COURSE OUTCOMES:

CO1: Understand and implement basic services and functionalities of the operating system using system calls and able to **Understand** the benefits of thread over process and implement synchronized programs using multithreading concepts.

CO2: Use modern operating system calls and synchronization libraries in software/ hardware interfaces.

CO3: Analyse and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.

CO4: Implement memory management schemes and page replacement schemes.

CO5: Simulate file allocation and organization techniques.

CO6: Understand the concepts of deadlock in operating systems and implement them in multiprogramming system.

EXPERIMENT-1

OBJECTIVE:

Write a program to implementation of Producer-Consumer Problem

DESCRIPTION:

There is one Producer and one Consumer in the producer-consumer problem.

Producer –

The producer process executes a set of statements int produce to create a data element and stores it in the buffer.

Consumer –

If the buffer has items, a consumer process executes a statement consume with the data element as a parameter.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C/ Code blocks.

ALGORITHM:

Producer code:

```
item nextProduced;
while( true ) {
/* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( . . . );

    /* Wait for space to become available */
    while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
        ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;
}
```

Consumer code:

```
item nextConsumed;
while( true ) {
/* Wait for an item to become available */
    while( in == out )
        ; /* Do nothing */
/* Get the next available item */
    nextConsumed = buffer[ out ];
    out = ( out + 1 ) % BUFFER_SIZE;

    /* Consume the item in nextConsumed
       ( Do something with it ) */
}
```

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/XFBKdyyDRV/>

PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;
int buffer[5],in=0,out=0,n,item,con;
void output();
void produce();
void consumer();

void produce()
{
    while(true)
    {
        if(((in+1)%5)==out)
        {
            cout<<"Buffer is full"<<endl;
            output();
            break;
        }
        cout<<"Enter number: ";
        cin>>item;
        buffer[in]=item;
        in=(in+1)%5;
        output();
    }
}

void consumer()
{
```

```
while(true)
{
    if(in==out)
    {
        cout<<"Buffer is empty"<<endl;
        output();
        break;
    }
    con=buffer[out];
    cout<<"Consume number is "<<con<<endl;
    out=(out+1)%5;
    output();
}
}
void output()
{
    cout<<"1.Producer 2.Consumer 3.Exit"<<endl;
    cin>>n;
    if(n==1)
    {
        produce();
    }
    if(n==2)
    {
        consumer();
    }
}
int main()
{
    output();
}
```

Input/Output:

```
"E:\Operating System Lab\prc" X + v
1.Producer 2.Consumer 3.Exit
1
Enter number: 40
1.Producer 2.Consumer 3.Exit
1
Enter number: 30
1.Producer 2.Consumer 3.Exit
1
Enter number: 60
1.Producer 2.Consumer 3.Exit
2
Consume number is 40
1.Producer 2.Consumer 3.Exit
2
Consume number is 30
1.Producer 2.Consumer 3.Exit
2
Consume number is 60
1.Producer 2.Consumer 3.Exit
```

EXPERIMENT-2

OBJECTIVE:

Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for the above problem.

a) FCFS b) SJF c) Round Robin d) Priority

DESCRIPTION:

Assume all the processes arrive at the same time.

FCFS CPU SCHEDULING ALGORITHM:

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

SJF CPU SCHEDULING ALGORITHM:

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

ROUND ROBIN CPU SCHEDULING ALGORITHM:

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

PRIORITY CPU SCHEDULING ALGORITHM:

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

CPU SCHEDULING

Maximum CPU utilization obtained with multiprogramming

CPU–I/O Burst Cycle – Process execution consists of a *cycle* of

CPU execution and I/O wait

CPU burst distribution

EXPERIMENT : 2a

a) First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
----------------	-------------------

P1	24
----	----

P2	3
----	---

P3	3
----	---

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:



Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: $(0 + 24 + 27)/3 = 17$

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Burst times of processes
5. calculate the waiting time of each process
 $wt[i+1]=bt[i]+wt[i]$
6. calculate the turnaround time of each process
 $tt[i+1]=tt[i]+bt[i+1]$
7. Calculate the average waiting time and average turnaround time.
8. Display the values
9. Stop

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/jWdF774qzX/>

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int pid[15];
    int bt[15];
    int n;
    cout<<"Enter the number of processes: ";
    cin>>n;

    cout<<"Enter process id of all the processes: ";
    for(int i=0;i<n;i++)
    {
        cin>>pid[i];
    }

    cout<<"Enter burst time of all the processes: ";
    for(int i=0;i<n;i++)
    {
        cin>>bt[i];
    }

    int i, wt[n];
    wt[0]=0;

    //for calculating waiting time of each process
    for(i=1; i<n; i++)
    {
        wt[i]= bt[i-1]+ wt[i-1];
    }

    cout<<"Process ID    Burst Time    Waiting Time    TurnAround Time\n";
    float twt=0.0;
    float tat= 0.0;
    for(i=0; i<n; i++)
    {
        cout<<pid[i]<<"\t\t";
        cout<<bt[i]<<"\t\t";
        cout<<wt[i]<<"\t\t";

        //calculating and printing turnaround time of each process
        cout<<bt[i]+wt[i]<<"\t\t";
```

```

    cout<<"\n";

    //for calculating total waiting time
    twt += wt[i];

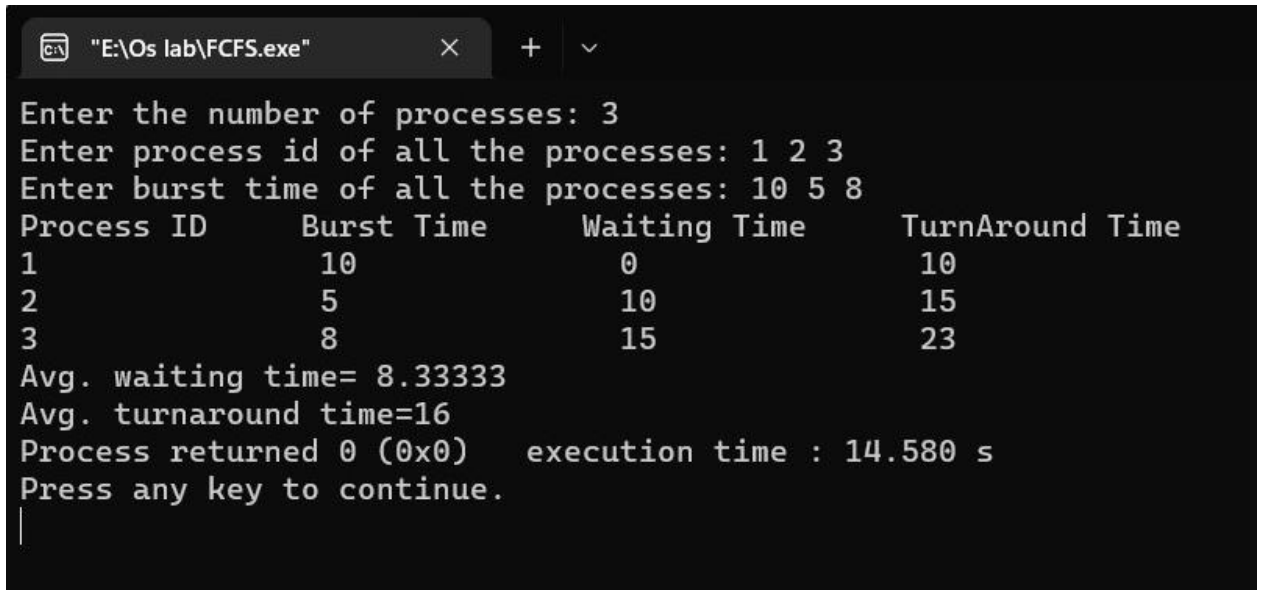
    //for calculating total turnaround time
    tat += (wt[i]+bt[i]);
}
float att,awt;

//for calculating average waiting time
awt = twt/n;

//for calculating average turnaround time
att = tat/n;
cout<<"Avg. waiting time= "<<awt<<"\n";
cout<<"Avg. turnaround time="<< att;
}

```

Input/Output:



```

E:\Os lab\FCFS.exe
Enter the number of processes: 3
Enter process id of all the processes: 1 2 3
Enter burst time of all the processes: 10 5 8
Process ID      Burst Time      Waiting Time      TurnAround Time
1                10                0                10
2                5                 10               15
3                8                 15               23
Avg. waiting time= 8.33333
Avg. turnaround time=16
Process returned 0 (0x0)    execution time : 14.580 s
Press any key to continue.
|

```

EXPERIMENT : 2b

NAME OF THE EXPERIMENT: Simulate the following CPU Scheduling Algorithms

b) SJF (non-preemptive):

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

Example of Preemptive SJF:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	3.0	4

P1	P2	P3	P2	P4	P1
----	----	----	----	----	----

Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/MCxNYzKtHq/>

PROGRAM:

```
#include <stdio.h>

int main()
{
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, limit;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;
    printf("\nEnter the Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Details of %d Processesn", limit);
    for(i = 0; i < limit; i++)
    {
        printf("\nEnter Arrival Time:\t");
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time:\t");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    burst_time[9] = 9999;
    for(time = 0; count != limit; time++)
    {
        smallest = 9;
        for(i = 0; i < limit; i++)
        {
            if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest] && burst_time[i] > 0)
            {
                smallest = i;
            }
        }
        burst_time[smallest]--;
        if(burst_time[smallest] == 0)
        {
            count++;
            end = time + 1;
            wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
            turnaround_time = turnaround_time + end - arrival_time[smallest];
        }
    }
    average_waiting_time = wait_time / limit;
    average_turnaround_time = turnaround_time / limit;
    printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
    printf("Average Turnaround Time:\t%lf\n", average_turnaround_time);
    return 0;
}
```


Input/Output:

```

enter the no of processes : 4
the arrival time for process P1:0
the burst time for process P1:7
the arrival time for process P2:4
the burst time for process P2:1
the arrival time for process P3:2
the burst time for process P3:4
the arrival time for process P4:5
the burst time for process P4:4

Process           Turnaround Time           Waiting Time
P[1] |            7 |            0
P[2] |            4 |            3
P[3] |           10 |            6
P[4] |           11 |            7

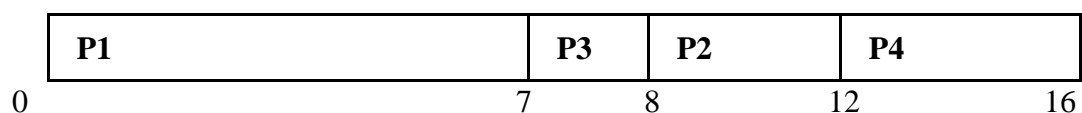
average waiting time = 4

average turnaround time = 8
Process returned 0 (0x0)   execution time : 13.700 s
Press any key to continue.
|

```

Example of Non- Pre-emptive SJF:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	3.0	4



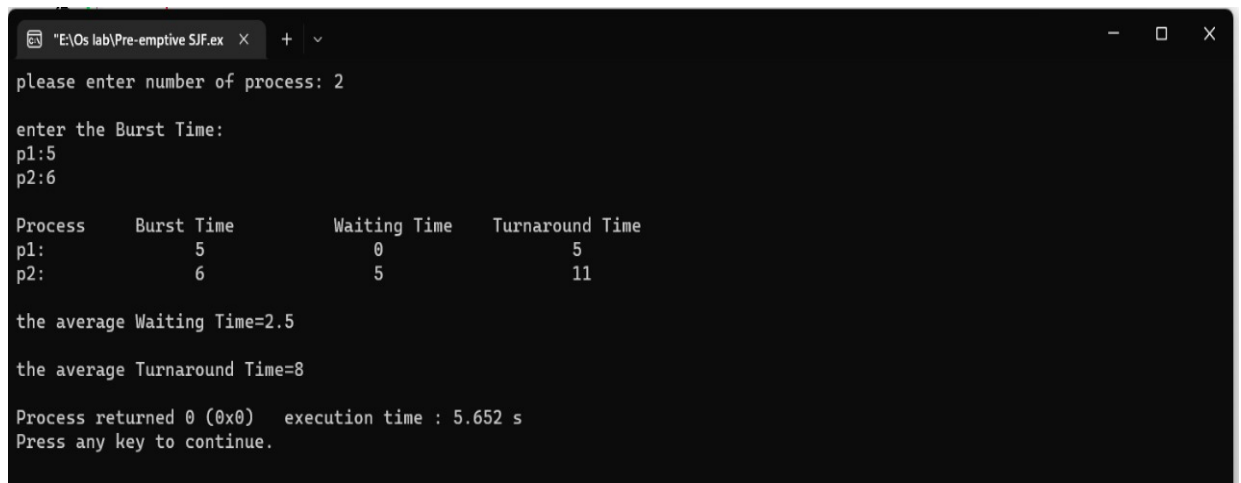
PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int burst_time[20],p[20],waiting_time[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_waiting_time,avg_tat;
    cout<<"please enter number of process: ";
    cin>>n;
    cout<<"\nenter the Burst Time:\n";
    for(i=0; i<n; i++)
    {
        cout<<"p"<<i+1<<":";
        cin>>burst_time[i];
        p[i]=i+1;
    }
    // from here, burst times sorted
    for(i=0; i<n; i++)
    {
        pos=i;
        for(j=i+1; j<n; j++)
        {
            if(burst_time[j]<burst_time[pos])
            {
                pos=j;
            }
        }
        swap(burst_time[i],burst_time[pos] );
        swap(p[i],p[pos] );
    }

    waiting_time[0]=0;
    //for calculating waiting time of each process
    for(i=1; i<n; i++)
    {
        waiting_time[i]= burst_time[i-1]+ waiting_time[i-1];
        total+=waiting_time[i];
    }
    avg_waiting_time=(float)total/n;
    total=0;
    cout<<"\nProcess\t Burst Time \tWaiting Time\tTurnaround Time";
    for(i=0; i<n; i++)
    {
        tat[i]=burst_time[i]+waiting_time[i];
        total+=tat[i];
    }
```

```
        cout<<"\n" << p[i] << ": \t\t " << burst_time[i] << " \t\t " << waiting_time[i] << " \t\t " << tat[i];
    }
    avg_tat = (float)total/n;
    cout<<"\n\nthe average Waiting Time=" << avg_waiting_time << endl;
    cout<<"\nthe average Turnaround Time=" << avg_tat << endl;
}
```

Input/Output:



```
"E:\Os lab\Pre-emptive SJF.exe" x + v
please enter number of process: 2

enter the Burst Time:
p1:5
p2:6

Process    Burst Time    Waiting Time    Turnaround Time
p1:         5         0              5
p2:         6         5              11

the average Waiting Time=2.5

the average Turnaround Time=8

Process returned 0 (0x0)   execution time : 5.652 s
Press any key to continue.
```

EXPERIMENT : 2c

a) Round Robin Scheduling

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

Round Robin:

Example of RR with time quantum=3

Process	Burst time
1	4
2	3
3	2
4	5
5	1

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the burst times of the processes
5. Read the Time Quantum
6. if the burst time of a process is greater than time Quantum then subtract time quantum from the burst time Else
Assign the burst time to time quantum.
7. calculate the average waiting time and turn around time of the processes.
8. Display the values
9. Stop

SOURCE CODE LINK: <https://paste.ubuntu.com/p/DgsTnz5MyS/>

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int st[10], bt[10], wt[10], tat[10], n, tq;
    int i, count = 0, swt = 0, stat = 0, temp, sq = 0;
    float awt = 0.0, atat = 0.0;
    printf("Enter number of processes:");
    scanf("%d", &n);
    printf("Enter burst time for sequences:");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &bt[i]);
        st[i] = bt[i];
    }
    printf("Enter time quantum:");
    scanf("%d", &tq);
    while (1)
    {
        for (i = 0, count = 0; i < n; i++)
        {
            temp = tq;
            if (st[i] == 0)
            {
                count++;
                continue;
            }
            if (st[i] > tq)
                st[i] = st[i] - tq;
            else if (st[i] >= 0)
            {
                temp = st[i];
                st[i] = 0;
            }
            sq = sq + temp;
            tat[i] = sq;
        }
        if (n == count)
            break;
    }
    for (i = 0; i < n; i++)
    {
```

```

        wt[i] = tat[i] - bt[i];
        swt = swt + wt[i];
        stat = stat + tat[i];
    }
    awt = (float)swt / n;
    atat = (float)stat / n;
    printf("Process_no Burst time Wait time Turn around time");
    for (i = 0; i < n; i++)
        printf("\n%d\t %d\t %d\t %d", i + 1, bt[i], wt[i], tat[i]);
    printf("\nAvg wait time is %f Avg turn around time is %f", awt, atat);
    getch();
}

```

Input/Output:

```

E:\Operating System Lab\RO x + v
Enter Details of Process[1]
Arrival Time: 0
Burst Time: 10
Enter Details of Process[2]
Arrival Time: 3
Burst Time: 5
Enter Details of Process[3]
Arrival Time: 5
Burst Time: 2
Enter Details of Process[4]
Arrival Time: 6
Burst Time: 6
Enter Details of Process[5]
Arrival Time: 8
Burst Time: 4

Enter Time Quantum: 4

Process ID      Burst Timet Turnaround Timet Waiting Timen
Process[3]      2           5           3
Process[5]      4           10          6
Process[2]      5           20          15
Process[4]      6           19          13
Process[1]      10          27          17

Average Waiting Time: 10.8
Avg Turnaround Time: 16.2
Process returned 0 (0x0) execution time : 282.397 s
Press any key to continue.

```

EXPERIMENT : 2d

NAME OF THE EXPERIMENT: Simulate the following CPU Scheduling Algorithms

d) Priority

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

In Priority Scheduling, each process is given a priority, and higher priority methods are executed first, while equal priorities are executed [First Come First Served](#) or [Round Robin](#).

There are several ways that priorities can be assigned:

Internal priorities are assigned by technical quantities such as memory usage, and file/IO operations. External priorities are assigned by politics, commerce, or user preference, such as importance and amount being paid for process access (the latter usually being for mainframes).

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Priorities of processes
5. sort the priorities and Burst times in ascending order
5. calculate the waiting time of each process
 $wt[i+1]=bt[i]+wt[i]$
6. calculate the turnaround time of each process
 $tt[i+1]=tt[i]+bt[i+1]$
6. Calculate the average waiting time and average turnaround time.
7. Display the values

SOURCE CODE LINK: <https://paste.ubuntu.com/p/QSMTrTNSJT/>

PROGRAM:

```
#include<iostream>

using namespace std;

int main()
{
    int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
    cout<<"Enter Total Number of Process:";
    cin>>n;

    cout<<"\nEnter Burst Time and Priority\n";
    for(i=0;i<n;i++)
    {
        cout<<"\nP["<<i+1<<"]\n";
        cout<<"Burst Time:";
        cin>>bt[i];
        cout<<"Priority:";
        cin>>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;

    cout<<"\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];
        cout<<"\nP["<<p[i]<<"]\t\t "<<bt[i]<<"\t\t "<<wt[i]<<"\t\t"<<tat[i];
    }

    avg_tat=total/n;    //average turnaround time
    cout<<"\n\nAverage Waiting Time="<<avg_wt;
    cout<<"\nAverage Turnaround Time="<<avg_tat;

    return 0;
}

```

Input/Output:

```
"E:\Operating System Lab\Prii  X  +  v

Enter Burst Time and Priority

P[1]
Burst Time:10
Priority:4

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:4
Priority:1

P[4]
Burst Time:7
Priority:3

Process      Burst Time      Waiting Time      Turnaround Time
P[3]          4              0                4
P[2]          2              4                6
P[4]          7              6               13
P[1]         10             13              23

Average Waiting Time=5
Average Turnaround Time=11
Process returned 0 (0x0)  execution time : 34.327 s
Press any key to continue.
```

EXPERIMENT : 3

OBJECTIVE:

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

DESCRIPTION:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

NAME OF EXPERIMENT: Simulate Banker's Algorithm for Deadlock Avoidance.

AIM: Simulate Banker's Algorithm for Deadlock Avoidance to find whether the system is in safe state or not.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

DEAD LOCK AVOIDANCE

To implement deadlock avoidance & Prevention by using Banker's Algorithm.

Banker's Algorithm:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

- n - Number of process, m - number of resource types.
- Available: Available[j]=k, k – instance of resource type R_j is available.
- Max: If max[i, j]=k, P_i may request at most k instances resource R_j .
- Allocation: If Allocation [i, j]=k, P_i allocated to k instances of resource R_j
- Need: If Need[I, j]=k, P_i may need k more instances of resource type R_j ,
- Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm:

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
 - Finish[i] =False
 - Need<=WorkIf no such I exists go to step 4.
3. work=work+Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm:

Let Request i be request vector for the process P_i , If request i=[j]=k, then process P_i wants k instances of resource type R_j .

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise P_i must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows;

Available=Available-Request I;

Allocation I =Allocation+Request I;

Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process P_i is allocated its resources. However if the state is unsafe, the P_i must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.

-
8. If the new request comes then check that the system is in safety.
 9. or not if we allow the request.
 10. stop the program

Source code link: <https://paste.ubuntu.com/p/rxq9FDtTCX/>

Program:

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

int maxi[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;

void input();

void show();

void cal();

void request();

int main()
{
    int i, j, f;
    cout<<"***** Banker's Algo *****"<<endl;
    input();
    cal();
    show();
    request();

    return 0;
}

void input()
{
    int i, j;
    cout<<"Enter the no of Processes: ";
    cin >> n;
    cout<<"Enter the no of resources instances: ";
    cin >> r;
    cout<<"Enter the Max Matrix:"<<endl;
    for (i = 0; i < n; i++)
```

```

    {
        for (j = 0; j < r; j++)
        {
            cin >> maxi[i][j];
        }
    }
    cout<<"Enter the Allocation Matrix"<<endl;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < r; j++)
        {
            cin >> alloc[i][j];
        }
    }
    cout<<"Enter the available Resources: "<<endl;
    for (j = 0; j < r; j++)
    {
        cin >> avail[j];
    }
}

void show()
{
    int i, j;
    cout<<"Process\t Allocated\t Need\t Max\t Available"<<endl;
    for (i = 0; i < n; i++)
    {
        cout<<"\nP"<<i<<"\t";
        for (j = 0; j < r; j++)
        {
            cout<<alloc[i][j]<<" ";
        }
        cout<<"\t";
        for (j = 0; j < r; j++)
        {
            cout<<maxi[i][j]<<" ";
        }
        cout<<"\t";
        for (j = 0; j < r; j++)
        {
            cout<<need[i][j]<<" ";
        }
        cout<<"\t";
        if (i == 0)
        {
            for (j = 0; j < r; j++)
                cout<<avail[j]<<" ";
        }
    }
}

```

```

    }
}

void cal()
{
    int finish[100], flag = 1, k, c1 = 0;
    int i, j;
    for (i = 0; i < n; i++)
    {
        finish[i] = 0;
    }

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < r; j++)
        {
            need[i][j] = maxi[i][j] - alloc[i][j];
        }
    }
    cout<<"\n";
    while (flag)
    {
        flag = 0;
        for (i = 0; i < n; i++)
        {
            int c = 0;
            for (j = 0; j < r; j++)
            {
                if ((finish[i] == 0) && (need[i][j] <= avail[j]))
                {
                    c++;
                    if (c == r)
                    {
                        for (k = 0; k < r; k++)
                        {
                            avail[k] += alloc[i][j];
                            finish[i] = 1;
                            flag = 1;
                        }
                        cout<<"P"<<i<<"->";
                        if (finish[i] == 1)
                        {
                            i = n;
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
}
for (i = 0; i < n; i++)
{
    if (finish[i] == 1)
    {
        c1++;
    }
    else
    {
        cout<<"P"<<i<<"->";
    }
}
if (c1 == n)
{
    cout<<"\n The system is in safe state\n";
}
else
{
    cout<<"\n Process are in dead lock\n";
    cout<<"\n System is in unsafe state\n";
}
}

```

```

void request()
{
    int c, pid, request[100][100], i;
    cout<<"\n Do you want make an additional request for any of the process ? (1=Yes|0=No)";
    cin>>c;
    if (c == 1)
    {
        cout<<"\n Enter process number : ";
        cin>>pid;
        cout<<"\n Enter additional request : \n";
        for (i = 0; i < r; i++)
        {
            cout<<" Request for resource "<<i+1<<": ";
            cin>>request[0][i];
        }
        if (request[0][i] <= need[pid][i])
        {
            if (request[0][i] <= avail[i])
            {
                for (i = 0; i < r; i++)
                {
                    avail[i] -= request[0][i];
                    alloc[pid][i] += request[0][i];
                    need[pid][i] -= request[0][i];
                }
            }
        }
    }
}

```

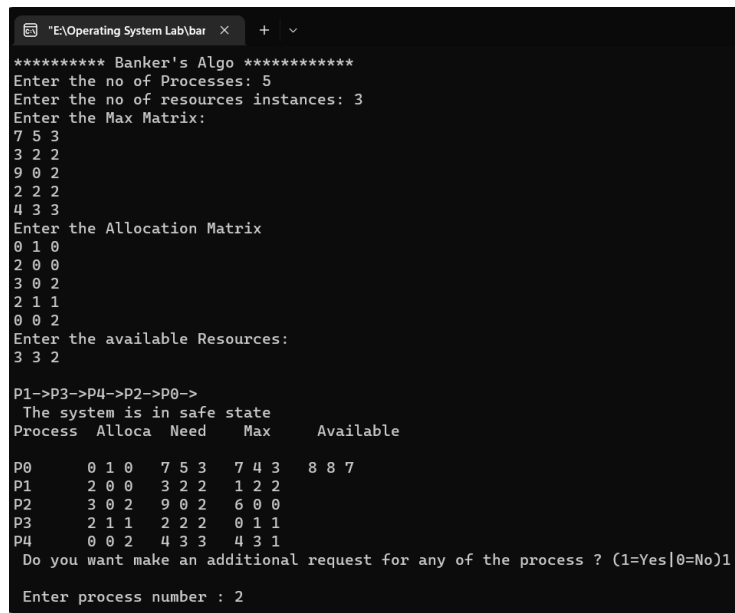
```

    }
    cal();
    show();
}
else
{
    cout << "Resource not available" << endl;
    exit(0);
}
}

else
{
    cout<<"\nError encountered\n";
    exit(0);
}
}
else
{
    exit(0);
}
}

```

Input/Output:



```

E:\Operating System Lab\bar  x  +  v
***** Banker's Algo *****
Enter the no of Processes: 5
Enter the no of resources instances: 3
Enter the Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available Resources:
3 3 2

P1->P3->P4->P2->P0->
The system is in safe state
Process  Alloca  Need    Max    Available
P0      0 1 0    7 5 3    7 4 3    8 8 7
P1      2 0 0    3 2 2    1 2 2
P2      3 0 2    9 0 2    6 0 0
P3      2 1 1    2 2 2    0 1 1
P4      0 0 2    4 3 3    4 3 1
Do you want make an additional request for any of the process ? (1=Yes|0=No)1
Enter process number : 2

```

Do you want make an additional request for any of the process ? (1=Yes|0=No)1

Enter process number : 2

Enter additional request :

Request for resource 1: 1 0 2

Request for resource 2: Request for resource 3:

P0->P1->P2->P3->P4->

The system is in safe state

Process	Alloca	Need	Max	Available
---------	--------	------	-----	-----------

P0	0 1 0	7 5 3	7 4 3	14 15 12
----	-------	-------	-------	----------

P1	2 0 0	3 2 2	1 2 2	
----	-------	-------	-------	--

P2	4 0 4	9 0 2	5 0 -2	
----	-------	-------	--------	--

P3	2 1 1	2 2 2	0 1 1	
----	-------	-------	-------	--

P4	0 0 2	4 3 3	4 3 1	
----	-------	-------	-------	--

Process returned 0 (0x0) execution time : 202.252 s

Press any key to continue.

EXPERIMENT-4(a)

OBJECTIVE: Simulate all page replacement Algorithms

a)FIFO b) Optimal c)LRU

DESCRIPTION:

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page- replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count

NAME OF EXPERIMENT: Simulate page replacement algorithms:

a) FIFO

AIM: Simulate FIFO page replacement algorithms.

HARDWARE REQUIREMENTS:

AMD Ryzen 5 3500U based Desktop PC RAM of 8GB

SOFTWARE REQUIREMENTS: Code Blocks

THEORY:

FIFO algorithm:

The simpler page replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replace, the oldest page is chosen.

We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		4	0	0	0	3	3			3	2			2	2	1

ALGORITHM:

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Allocate the pages in to frames in First in first out order.
7. Display the number of page faults.
8. stop

SOURCE CODE LINK : <https://pastebin.ubuntu.com/p/wXJ8Zz5zSQ/>

PROGRAM:

```
#include<stdio.h>
int main()
{
    int incomingStream[] = {4 , 1 , 2 , 4 , 5};
    int pageFaults = 0;
    int frames = 3;
    int m, n, s, pages;
    pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
    printf(" Incoming \t Frame 1 \t Frame 2 \t Frame 3 ");
    int temp[ frames ];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
    for(m = 0; m < pages; m++)
    {
        s = 0;
        for(n = 0; n < frames; n++)
        {
            if(incomingStream[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = incomingStream[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
```

```

    }
    printf("\n");
    printf("%d\t\t",incomingStream[m]);
    for(n = 0; n < frames; n++)
    {
        if(temp[n] != -1)
            printf(" %d\t\t", temp[n]);
        else
            printf(" - \t\t");
    }
}
printf("\nTotal Page Faults:\t%d\n", pageFaults);
return 0;
}

```

Input/Output:

```

E:\Os lab\Fifo.exe
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3
The Page Replacement Process is --
7      -1      -1      PF No.1
7      0      -1      PF No.2
7      0      1       PF No.3
2      0      1       PF No.4
2      0      1       PF No.5
2      3      1       PF No.6
2      3      0       PF No.7
4      3      0       PF No.8
4      2      0       PF No.9
4      2      3       PF No.10
0      2      3       PF No.11
0      2      3       PF No.12
0      1      3       PF No.13
0      1      2       PF No.14
0      1      2       PF No.15
7      1      2       PF No.16
7      0      2       PF No.17
7      0      1       PF No.18

The number of Page Faults using FIFO are 15
Process returned 0 (0x0)   execution time : 39.415 s
Press any key to continue.

```

EXPERIMENT-4(b)

NAME OF EXPERIMENT: Simulate page replacement algorithms
b)Optimal

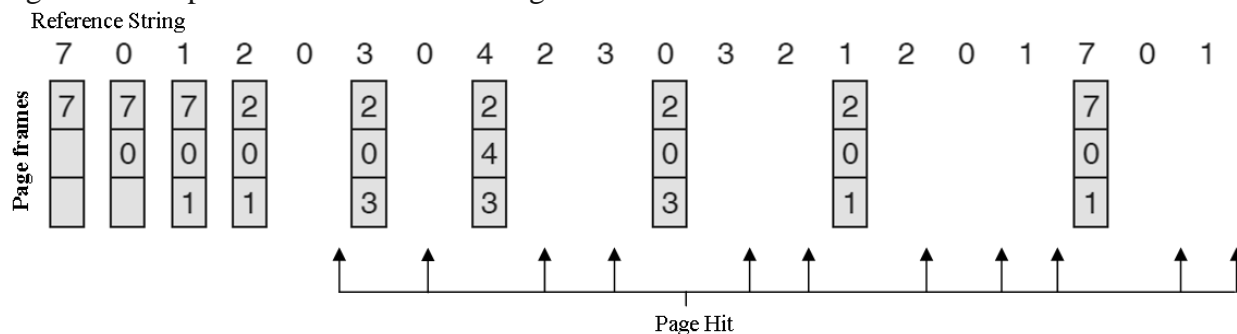
AIM: Simulate Optimal page replacement algorithms.

HARDWARE REQUIREMENTS:
AMD Ryzen 5 3500U based Desktop PC RAM of 8GB

SOFTWARE REQUIREMENTS: Code Blocks

ALGORITHM :

By utilizing optimal page-replacement algorithm ensure the most minimal conceivable page fault rate for a fixed number of frames. For instance, on our example reference string, the optimal page-replacement algorithm would yield nine-page faults, as to represent in Figure 1. Initially the first three references cause (pre-defined number of page frames) faults that fill the three frames that are empty. The reference to page 2 replaces page 7 because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With just nine-page faults, optimal page replacement is obviously superior to a FIFO algorithm, which results in fifteen faults. (In the event that we ignore the first three, which all algorithms must endure, at that point optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.



Tragically, the optimal page-replacement algorithm is hard to execute, in light of the fact that it requires future knowledge of the reference string. Therefore, the optimal page replacement algorithm is utilized mainly for comparison studies. For example, it may be valuable to realize that, although a new algorithm is not optimal, it is around 12.3 (%) percent of optimal at worst and within 4.7 (%) percent on average.

SOURCE CODE LINK : <https://pastebin.ubuntu.com/p/rd8ZK2Q3sf/>

PROGRAM:

```
#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;
}

```

Input/Output:

```
"E:\Operating System Lab\opt" x + v
Enter number of frames: 3
Enter number of pages: 20
Enter page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7      -1      -1
7      0      -1
7      0      1
2      0      1
2      0      1
2      0      3
2      0      3
2      4      3
2      4      3
2      4      3
2      0      3
2      0      3
2      0      3
2      0      1
2      0      1
2      0      1
2      0      1
7      0      1
7      0      1
7      0      1

Total Page Faults = 9
Process returned 0 (0x0)   execution time : 54.463 s
Press any key to continue.
```

EXPERIMENT-4(c)

NAME OF EXPERIMENT: Simulate page replacement algorithms:

c) LRU

AIM: Simulate LRU page replacement algorithms.

HARDWARE REQUIREMENTS:

AMD Ryzen 5 3500U based Desktop PC RAM of 8GB

SOFTWARE REQUIREMENTS: Code Blocks

ALGORITHM:

- 1.Start
- 2.Read the number of frames
- 3.Read the number of pages
- 4.Read the page numbers
- 5.Initialize the values in frames to -1
- 6.Allocate the pages in to frames by selecting the page that has not been used for the longest period of time.
- 7.Display the number of page faults.
- 8.stop

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/vq6DVMN3fx/>

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
int fsize;
int frm[15];
void display();
void main()
{
    int pg[100], change[15], nPage, i, j, k, l, index, pf = 0, temp, flag = 0, flag1 = 0, found, max;
    printf("\n Enter frame size:");
    scanf("%d", &fsize);
    printf("\n Enter number of pages:");
    scanf("%d", &nPage);

    for (i = 0; i < nPage; i++)
    {
        printf("\n Enter page[%d]:", i + 1);
        scanf("%d", &pg[i]);
    }
}
```

```

for (i = 0; i < fsize; i++)
    frm[i] = -1;

printf("\n page | \t Frame content ");
printf("\n-----");
for (j = 0; j < nPage; j++)
{
    flag = 0;
    flag1 = 0;
    for (i = 0; i < fsize; i++)
    {
        if (frm[i] == pg[j])
        {
            flag = 1;
            flag1 = 1;
            break;
        }
    }
    if (flag == 0)
    {

        for (i = 0; i < fsize; i++)
        {
            if (frm[i] == -1)
            {
                frm[i] = pg[j];
                pf++;
                flag1 = 1;
                break;
            }
        }
    }

    if (flag1 == 0)
    {
        for (i = 0; i < fsize; i++)
            change[i] = 0;

        for (i = 0; i < fsize; i++)
        {
            for (k = j + 1; k <= nPage; k++)
            {
                if (frm[i] == pg[k])
                {
                    change[i] = k - j;
                    break;
                }
            }
        }
    }
}

```

```

    }
    found = 0;

    for (i = 0; i < fsize; i++)
    {
        if (change[i] == 0)
        {
            index = i;
            found = 0;
            break;
        }
    }
    if (found == 0)
    {
        max = change[0];
        index = 0;

        for (i = 1; i < fsize; i++)
        {
            if (max < change[i])
            {
                max = change[i];
                index = i;
            }
        }
        frm[index] = pg[j];
        pf++;
    }
    printf("\n %d   |", pg[j]);
    display();
}
printf("\n-----");
printf("\n total page fault:%d", pf);
getch();
}
void display()
{
    int i;
    for (i = 0; i < fsize; i++)
        printf("\t %d", frm[i]);
}

```

Input/Output:

```
"E:\Operating System Lab\lru.  × + ▾ "E:\Operating System Lab\lru.  × + ▾
Enter frame size:3
Enter number of pages:12
Enter page[1]:0
Enter page[2]:1
Enter page[3]:2
Enter page[4]:3
Enter page[5]:0
Enter page[6]:1
Enter page[7]:4
Enter page[8]:0
Enter page[9]:1
Enter page[10]:2
Enter page[11]:3
Enter page[12]:4
page | Frame content
-----
0 | 0 -1 -1
1 | 0 1 -1
2 | 0 1 2
3 | 0 1 3
0 | 0 1 3
1 | 0 1 3
4 | 0 1 4
0 | 0 1 4
1 | 0 1 4
2 | 2 1 4
3 | 2 1 3
4 | 4 1 3
-----
page | Frame content total page fault:8
```

EXPERIMENT-5

OBJECTIVE: Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION:

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.

a) Worst-fit

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/92P672sRfX/>

PROGRAM:

```
#include<iostream>
#include<algorithm>
using namespace std;

struct node{
    int memsize;
    int allocp=-1;
    int pos;
    int allocSize;
}m[200];

bool posSort(node a,node b){
    return a.pos < b.pos;
}

bool memSort(node a,node b){
    return a.memsize > b.memsize;
}

int main(){
```

```

int nm,np,choice, i, j, p[200];
cout<<"Enter number of blocks\n";
cin>>nm;
cout<<"Enter block size\n";
for(i=0;i<nm;i++){
    cin>>m[i].memsize;
    m[i].pos=i;
}
cout<<"Enter number of processes\n";
cin>>np;
cout<<"Enter process size\n";
for(i=0;i<np;i++){
    cin>>p[i];
}
cout<<"\n\n";
sort(m,m+nm,memSort);
int globalFlag=0;
for(i=0;i<np;i++){
    int flag=0;
    for(j=0;j<nm;j++){
        if(p[i]<=m[j].memsize && m[j].allocp==-1){
            m[j].allocp=i;
            m[j].allocSize=p[i];
            flag=1;
            break;
        }
    }
    if(flag==0){
        cout<<"Unallocated Process P"<<i+1<<"\n";
        globalFlag=1;
    }
}
sort(m,m+nm,posSort);
cout<<"\n";
int intFrag=0,extFrag=0;
cout<<"Memory\t\t";
for(i=0;i<nm;i++){
    cout<<m[i].memsize<<"\t";
}
cout<<"\n";
cout<<"P. Alloc.\t";
for(i=0;i<nm;i++){
    if(m[i].allocp!=-1){
        cout<<"P"<<m[i].allocp+1<<"\t";
    }
    else{
        cout<<"Empty\t";
    }
}

```

```

    }
    cout<<"\n";
    cout<<"Int. Frag.\t";
    for(i=0;i<nm;i++){
        if(m[i].allocp!=-1){
            cout<<m[i].memsize-m[i].allocSize<<"\t";
            intFrag+=m[i].memsize-m[i].allocSize;
        }
        else{
            extFrag+=m[i].memsize;
            cout<<"Empty\t";
        }
    }
    cout<<"\n";
    cout<<"\n";

    if(globalFlag==1)
        cout<<"Total External Fragmentation: "<<extFrag<<"\n";
    else{
        cout<<"Available Memory: "<<extFrag<<"\n";
    }
    cout<<"Total Internal Fragmentation: "<<intFrag<<"\n";
    return 0;
}

```

Input/Output:

```

E:\Operating System Lab\wo  x  +  v
Enter number of blocks
5
Enter block size
100 500 200 300 600
Enter number of processes
4
Enter process size
212 417 112 426

Unallocated Process P4

Memory      100    500    200    300    600
P. Alloc.   Empty P2    Empty P3    Empty P1
Int. Frag.   Empty 83     Empty 188   388

Total External Fragmentation: 300
Total Internal Fragmentation: 659

Process returned 0 (0x0)   execution time : 43.649 s
Press any key to continue.

```

b) Best-fit

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/WB962R2xVj/>

PROGRAM:

```
#include<iostream>
#include<algorithm>
using namespace std;
struct node{
    int memsize;
    int allocp=-1;
    int pos;
    int allocSize;
}m[200];
bool posSort(node a,node b){
    return a.pos < b.pos;
}
bool memSort(node a,node b){
    return a.memsize < b.memsize;
}
int main(){
    int nm,np,choice, i, j, p[200];
    cout<<"Enter number of blocks\n";
    cin>>nm;
    cout<<"Enter block size\n";
    for(i=0;i<nm;i++){
        cin>>m[i].memsize;
        m[i].pos=i;
    }
    cout<<"Enter number of processes\n";
    cin>>np;
    cout<<"Enter process size\n";
    for(i=0;i<np;i++){
        cin>>p[i];
    }
    cout<<"\n\n";
    sort(m,m+nm,memSort);
    int globalFlag=0;
    for(i=0;i<np;i++){
        int flag=0;
        for(j=0;j<nm;j++){
            if(p[i]<=m[j].memsize && m[j].allocp===-1){
                m[j].allocp=i;
                m[j].allocSize=p[i];
                flag=1;
                break;
            }
        }
    }
}
```

```

    }
}
if(flag==0){
    cout<<"Unallocated Process P"<<i+1<<"\n";
    globalFlag=1;
}
}
sort(m,m+nm,posSort);
cout<<"\n";
int intFrag=0,extFrag=0;
cout<<"Memory\t\t";
for(i=0;i<nm;i++){
    cout<<m[i].memsize<<"\t";
}
cout<<"\n";
cout<<"P. Alloc.\t";
for(i=0;i<nm;i++){
    if(m[i].allocp!=-1){
        cout<<"P"<<m[i].allocp+1<<"\t";
    }
    else{
        cout<<"Empty\t";
    }
}
cout<<"\n";
cout<<"Int. Frag.\t";
for(i=0;i<nm;i++){
    if(m[i].allocp!=-1){
        cout<<m[i].memsize-m[i].allocSize<<"\t";
        intFrag+=m[i].memsize-m[i].allocSize;
    }
    else{
        extFrag+=m[i].memsize;
        cout<<"Empty\t";
    }
}
cout<<"\n";
cout<<"\n";
if(globalFlag==1)
    cout<<"Total External Fragmentation: "<<extFrag<<"\n";
else
{
    cout<<"Available Memory: "<<extFrag<<"\n";
}

cout<<"Total Internal Fragmentation: "<<intFrag<<"\n";
return 0;
}

```

Input/Output:

```
"E:\Operating System Lab\bes" X + v

Memory Management Scheme - Best Fit
Enter the number of blocks:5
Enter the number of processes:4

Enter the size of the blocks:-
Block no.1:10
Block no.2:15
Block no.3:5
Block no.4:9
Block no.5:3

Enter the size of the processes :-
Process no. 1:1
Process no. 2:4
Process no. 3:7
Process no. 4:12

Process_no    Process_size    Block_no    Block_size    Fragment
1             1               5           3             2
2             4               3           5             1
3             7               4           9             2
4             12              2          15            3

Process returned 0 (0x0)   execution time : 41.978 s
Press any key to continue.
```

C) FIRST-FIT:

SOURCE CODE LINK: <https://pastebin.ubuntu.com/p/ph8DXNMF8S/>

PROGRAM:

```
#include<iostream>
#include<algorithm>
using namespace std;
struct node{
    int memsize;
    int allocp=-1;
    int pos;
    int allocSize;
}m[200];
bool posSort(node a,node b){
    return a.pos < b.pos;
}
bool memSort(node a,node b){
    return a.memsize < b.memsize;
}
```

```

int main(){
    int nm,np,choice, i, j, p[200];
    cout<<"Enter number of blocks\n";
    cin>>nm;
    cout<<"Enter block size\n";
    for(i=0;i<nm;i++){
        cin>>m[i].memsize;
        m[i].pos=i;
    }
    cout<<"Enter number of processes\n";
    cin>>np;
    cout<<"Enter process size\n";
    for(i=0;i<np;i++){
        cin>>p[i];
    }
    cout<<"\n\n";
    //sort(m,m+nm,memSort);
    int globalFlag=0;
    for(i=0;i<np;i++){
        int flag=0;
        for(j=0;j<nm;j++){
            if(p[i]<=m[j].memsize && m[j].allocp== -1){
                m[j].allocp=i;
                m[j].allocSize=p[i];
                flag=1;
                break;
            }
        }
        if(flag==0){
            cout<<"Unallocated Process P"<<i+1<<"\n";
            globalFlag=1;
        }
    }
    sort(m,m+nm,posSort);
    cout<<"\n";
    int intFrag=0,extFrag=0;
    cout<<"Memory\t\t";
    for(i=0;i<nm;i++){
        cout<<m[i].memsize<<"\t";
    }
    cout<<"\n";
    cout<<"P. Alloc.\t";
    for(i=0;i<nm;i++){
        if(m[i].allocp!= -1){
            cout<<"P"<<m[i].allocp+1<<"\t";
        }
        else{
            cout<<"Empty\t";
        }
    }
}

```

```
    }
}
cout<<"\n";
cout<<"Int. Frag.\t";
for(i=0;i<nm;i++){
    if(m[i].allocp!=-1){
        cout<<m[i].memsize-m[i].allocSize<<"\t";
        intFrag+=m[i].memsize-m[i].allocSize;
    }
    else{
        extFrag+=m[i].memsize;
        cout<<"Empty\t";
    }
}
cout<<"\n";
cout<<"\n";
if(globalFlag==1)
    cout<<"Total External Fragmentation: "<<extFrag<<"\n";
else
{
    cout<<"Available Memory: "<<extFrag<<"\n";
}
    cout<<"Total Internal Fragmentation: "<<intFrag<<"\n";
return 0;
}
```

Input/Output:

```
"E:\Operating System Lab\Firs  X + v
Enter number of blocks
3
Enter block size
30 5 10
Enter number of processes
3
Enter process size
10 6 9

Unallocated Process P3

Memory          30      5      10
P. Alloc.       P1      Empty  P2
Int. Frag.      20      Empty  4

Total External Fragmentation: 5
Total Internal Fragmentation: 24

Process returned 0 (0x0)   execution time : 27.771 s
Press any key to continue.
```