OS - Experiment 1

Name: Kartik Jolapara Sapid: 60004200107

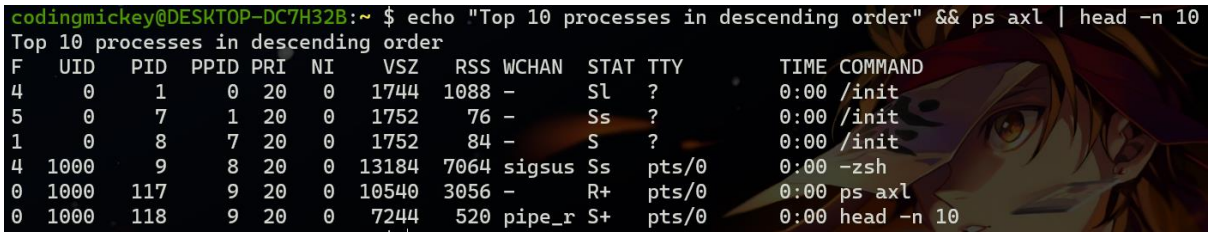
Div./Batch: B/B1 Branch: Computer Engineering

AIM:

Explore the internal commands of Linux and Write shell scripts to do the following.

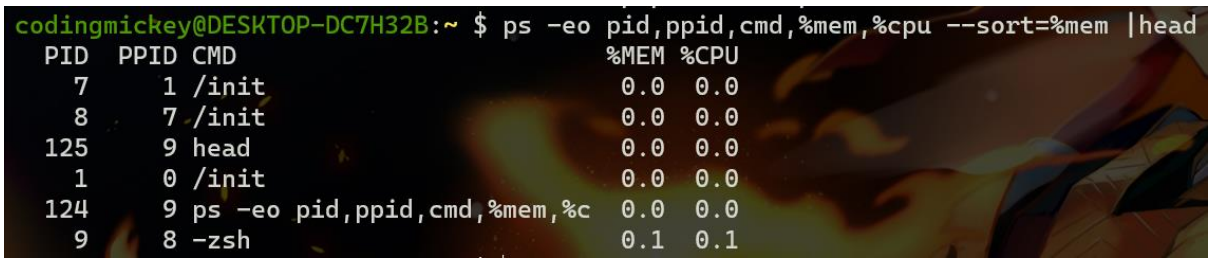
1. Display top 10 processes in descending order

```
codingmickey@DESKTOP-DC7H32B:~ $ echo "Top 10 processes in descending order" && ps axl | head -n 10
```



2. Display processes with highest memory usage.

```
codingmickey@DESKTOP-DC7H32B:~ $ ps -eo pid,ppid,cmd,%mem,%cpu --sort=%mem |head
```

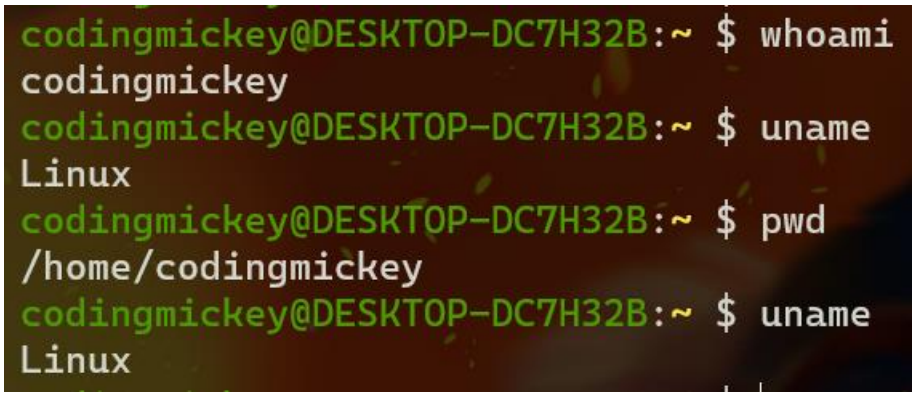


3. Display current logged in user and no. of users.

```
codingmickey@DESKTOP-DC7H32B:~ $ who -u
codingmickey@DESKTOP-DC7H32B:~ $ who -u | wc -l
```

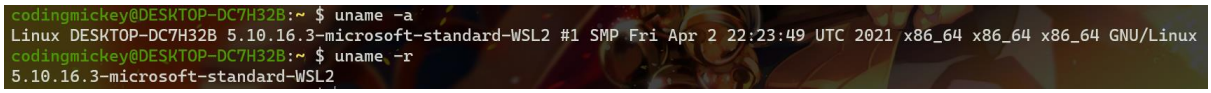
4. Display current shell, home directory, operating system type, current working directory.

```
codingmickey@DESKTOP-DC7H32B:~ $ whoami
codingmickey@DESKTOP-DC7H32B:~ $ uname
codingmickey@DESKTOP-DC7H32B:~ $ pwd
codingmickey@DESKTOP-DC7H32B:~ $ uname
```



5. Display OS version, release number.

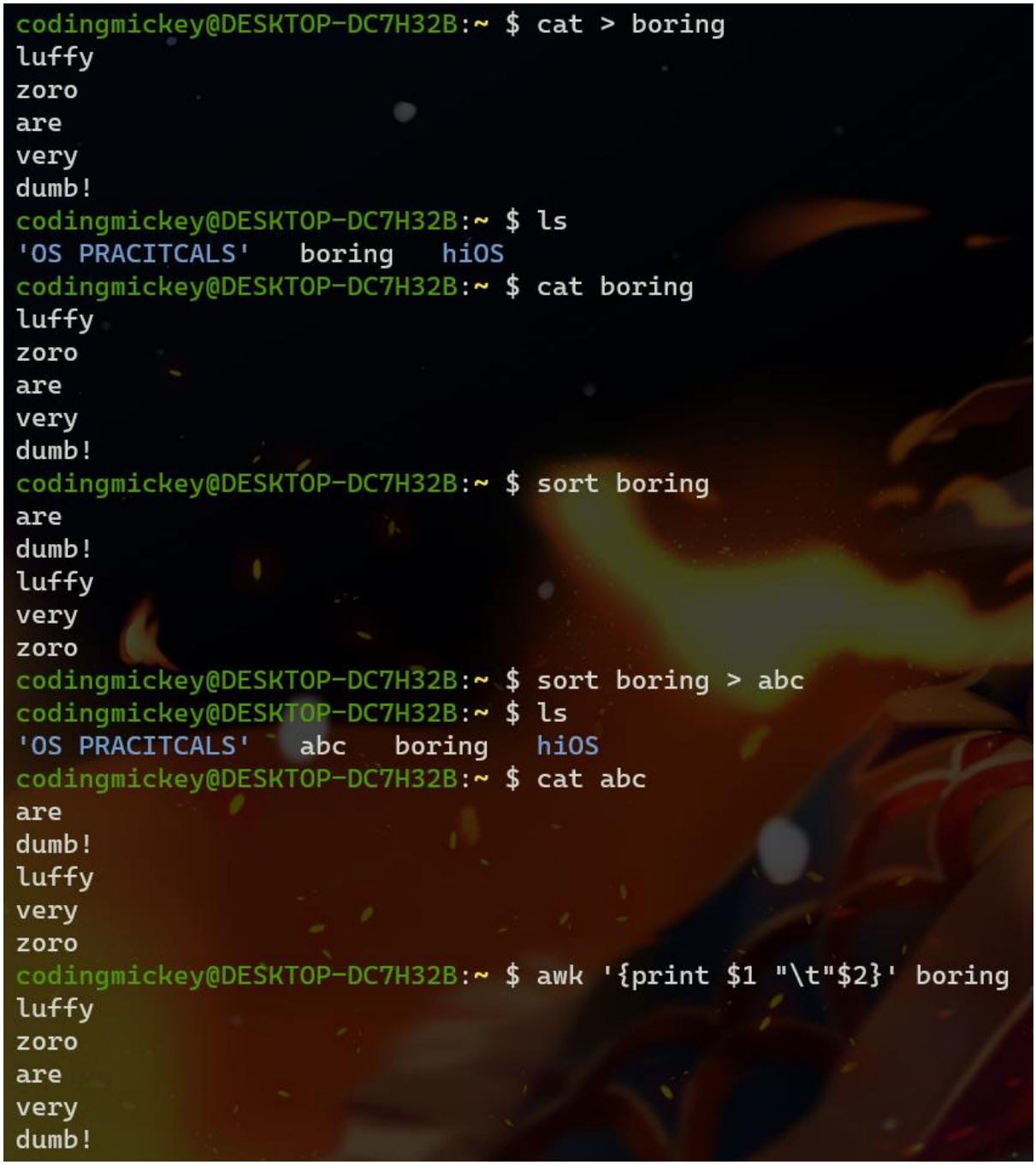
```
codingmickey@DESKTOP-DC7H32B:~ $ uname -a
codingmickey@DESKTOP-DC7H32B:~ $ uname -r
```



6. Illustrate the use of sort, grep, awk, etc.

```
codingmickey@DESKTOP-DC7H32B:~ $ cat > boring
codingmickey@DESKTOP-DC7H32B:~ $ ls
codingmickey@DESKTOP-DC7H32B:~ $ cat boring
codingmickey@DESKTOP-DC7H32B:~ $ sort boring
```

```
codingmickey@DESKTOP-DC7H32B:~ $ sort boring > abc
codingmickey@DESKTOP-DC7H32B:~ $ ls
codingmickey@DESKTOP-DC7H32B:~ $ cat abc
codingmickey@DESKTOP-DC7H32B:~ $ awk '{print $1 "\t"$2}'
boring
```



Conclusion:

The Linux based operating systems have a wide variety of commands or system calls that can be invoked through the command line or shell, like bash, to perform system and functions. Linux provides a more open approach to its system calls compared to other operating systems based on the UNIX philosophy. Linux based systems are developed to be able to be used only from the command line using the the terminals without the need of a GUI. Hence the commands present cater to every aspect of the system from daily use to system diagnostics.

OS - Experiment 2

Name: Kartik Jolapara Sapid: 60004200107

Div./Batch: B/B1 Branch: Computer Engineering

AIM:

System calls for file manipulation.

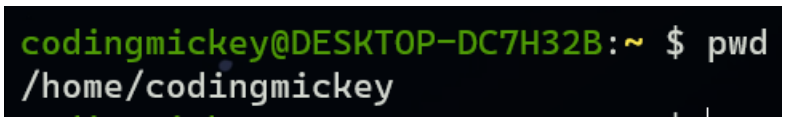
Problem Statement:

Try different file manipulation operations provided by linux

1. pwd command

pwd, short for the print working directory, is a command that prints out the current working directory in a hierarchical order, beginning with the topmost root directory (/). To check your current working directory, simply invoke the pwd command as shown.

```
codingmickey@DESKTOP-DC7H32B:~ $ pwd
```



2. mkdir command

You might have wondered how we created the tutorials directory. Well, it's pretty simple. To create a new directory use the mkdir (make directory) command as follows:

```
codingmickey@DESKTOP-DC7H32B:~ $ mkdir hiOS
```

```
codingmickey@DESKTOP-DC7H32B:~ $ ls
'OS PRACITCALS'   abc   boring
codingmickey@DESKTOP-DC7H32B:~ $ mkdir hiOS
codingmickey@DESKTOP-DC7H32B:~ $ ls
'OS PRACITCALS'   abc   boring   hiOS
```

3. ls command

The ls command is a command used for listing existing files or folders in a directory. For example, to list all the contents in the home directory, we will run the command.

```
codingmickey@DESKTOP-DC7H32B:~ $ ls
```

```
codingmickey@DESKTOP-DC7H32B:~ $ ls
'OS PRACITCALS'   abc   boring   hiOS
```

4. cd command

To change or navigate directories, use the cd command which is short for change directory.

For instance, to navigate to particular directory run the command:

```
$ cd hiOS
```

To go a directory up append two dots or periods in the end.

```
$ cd ..
```

To go back to the home directory run the cd command without any arguments.

```
$ cd
```

```
codingmickey@DESKTOP-DC7H32B:~ $ cd hiOS
codingmickey@DESKTOP-DC7H32B:~/hiOS $ cd ..
codingmickey@DESKTOP-DC7H32B:~ $
```

5. rmdir command

The rmdir command deletes an empty directory. For example, to delete or remove the tutorials directory, run the command:

```
codingmickey@DESKTOP-DC7H32B:~ $ rmdir hiOS
```

6. touch command

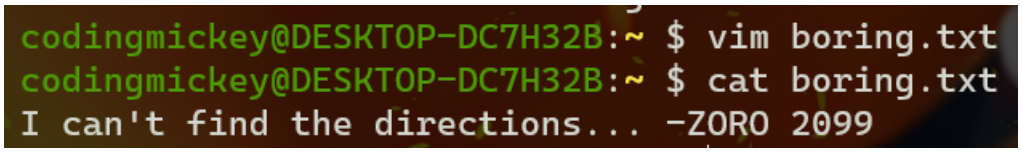
The touch command is used for creating simple files on a Linux system. To create a file, use the syntax: $ touch filename For example, to create a file1.txt file, run the command:

```
codingmickey@DESKTOP-DC7H32B:~ $ touch boring.txt
```

7. cat command

To view the contents of a file, use the cat command as follows:

```
codingmickey@DESKTOP-DC7H32B:~ $ cat fileName
```



8. mv command

The mv command is quite a versatile command. Depending on how it is used, it can rename a file or move it from one location to another. To move the file, use the syntax below:

```
$ mv filename /path/to/destination/
```

9. cp command

The cp command, short for copy, copies a file from one file location to another. Unlike the move command, the cp command retains the original file in its current location and makes a duplicate copy in a different directory. The syntax for copying a file is shown below.

```
$ cp /file/path /destination/path
```

10. Deleting a file

Deleting a File rm command could be used to delete a file. It will remove the filename file from the directory.

```
$rm filename
```

Conclusion:

Thus, we studied various Linux commands.

Operating Systems

Experiment 3

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

Building multi-threaded and multi-process applications

Problem Statement -

1) Build an instance of bus ticket reservation system using multithreading for the following scenario.

   ABC Bus service has only two seats left for reservations. Two users are trying to book the ticket at the same time.
2) Create separate thread per user. Show how this code is leading to inconsistency
3) Improve the code by applying synchronization.
4) Conclude the experiment by stating the importance of synchronization in multiprocess and multithread application

Theory -

Multithreading

The concept of Multithreading consists of two terms – a process and a thread. A process is a program being executed. A process can be further divided into independent units known as threads. A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

Therefore, by the above information, it is understood that Multithreading means that you have multiple threads of execution inside the same application. A thread is like a separate CPU executing your application. Thus, a multithreaded application is like an application that has multiple CPUs executing different parts of the code at the same time

Runnable Interface

Java runnable is an interface used to execute code on a concurrent thread. It is an interface which is implemented by any class if we want that the instances of that class should be executed by a thread.

The runnable interface has an undefined method run() with void as return type, and it takes in no arguments. The runnable interface provides a standard set of rules for the instances of classes which wish to execute code when they are active. The most common use case of the Runnable interface is when we want only to override the run method. When a thread is started by the object of any class which is implementing Runnable, then it invokes the run method in the separately executing thread.

Extends Thread

One way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run() method, which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

Code -

Using Multithreading (NO Synchronization)

```
class Passenger extends Thread {
    int seats_req;
    String name;

    Passenger(int s, String n, Reservation r) {
        super(r);
        seats_req = s;
        name = n;
```

```
    }
}

class Reservation implements Runnable {
    int seats_aval = 2;

    public void run() {
        Passenger p = (Passenger) Thread.currentThread();
        book(p.seats_req, p.name);
    }

    void book(int req, String n) {
        System.out.println(n);
        if (req <= seats_aval) {
            System.out.println("seats avaliable: " + seats_aval);
            seats_aval -= req;
            System.out.println(req + " tickets booked for " + n);
        } else {
            System.out.println("seats avaliable: " + seats_aval);
            System.out.println("Tickets not avaliable!!");
        }
    }
}

public class MultiThreading {
    public static void main(String[] args) {
        Reservation r = new Reservation();
        Passenger p1 = new Passenger(2, "Kartik", r);
        Passenger p2 = new Passenger(2, "Meet", r);
        p2.start();
        p1.start();
    }
}
```

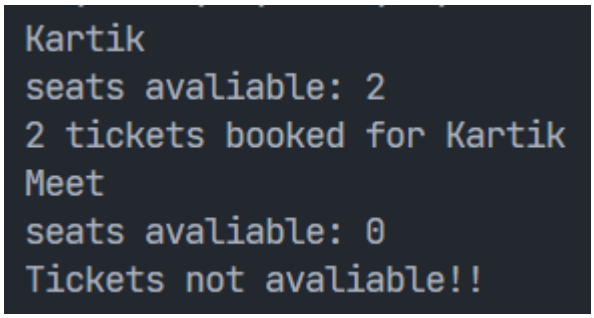
Output (NO Synchronization)

```
Kartik
Meet
seats avaliable: 2
seats avaliable: 2
2 tickets booked for Meet
2 tickets booked for Kartik
```

Using Multithreading (With Synchronization)

```
class Passenger extends Thread {
    int seats_req;
    String name;

    Passenger(int s, String n, Reservation r) {
        super(r);
        seats_req = s;
        name = n;
    }
}

class Reservation implements Runnable {
    int seats_aval = 2;

    public void run() {
        Passenger p = (Passenger) Thread.currentThread();
        book(p.seats_req, p.name);
    }

    // for synchronization change below function to synchronized
    synchronized void book(int req, String n) {
        System.out.println(n);
        if (req <= seats_aval) {
            System.out.println("seats avaliable: " + seats_aval);
            seats_aval -= req;
            System.out.println(req + " tickets booked for " + n);
        } else {
            System.out.println("seats avaliable: " + seats_aval);
            System.out.println("Tickets not avaliable!!");
        }
    }
}

public class MultiThreading {
    public static void main(String[] args) {
        Reservation r = new Reservation();
        Passenger p1 = new Passenger(2, "Kartik", r);
        Passenger p2 = new Passenger(2, "Meet", r);
        p1.start();
        p2.start();
    }
}
```

Output (With Synchronization)



Conclusion –

The concept of multithreading was executed, using it in the application of bus reservation system.

Operating Systems

Experiment 4

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

CPU scheduling algorithms like FCFS, SJF, Round Robin etc.

Problem Statement -

1. Perform comparative assessment of various Scheduling Policies like FCFS, SJF (preemptive and non-preemptive), Priority (preemptive and non-preemptive) and Round Robin.
2. Take the input processes, their arrival time, burst time, priority, quantum from user.

Theory -

Scheduling algorithms are used when more than one process is executable and the OS has to decide which one to run first.

Terms used

1. Submit time: The process at which the process is given to CPU
2. Burst time: The amount of time each process takes for execution
3. Response time: The difference between the time when the process starts execution and the submit time.
4. Turnaround time: The difference between the time when the process completes execution and the submit time.

First Come First Serve (FCFS)

First come first serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

Advantages of FCFS

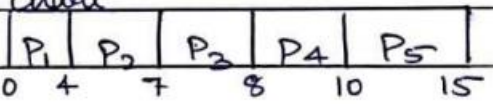
- Simple to understand
- Easy to implement

Disadvantages of FCFS

1. The scheduling method is non preemptive, the process will run to the completion.
2. Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.
3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
1	0	4	4	4	0
2	1	3	7	6	3
3	2	1	8	6	5
4	3	2	10	7	5
5	4	5	15	11	6

Gantt Chart



Average Turnaround Time = 6.8

Average Waiting Time = 3.8

Code –

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    vector<pair<float, float>> vect;
    vector<float> TT;
    vector<float> WT;
    int burst = 0, Total = 0;
    int p;
    cout << "Enter no processess: ";
    cin >> p;
    float AT[p], BT[p];
    cout << "Now enter the times in order of->\nAT BT\n";
    for (int i = 0; i < p; i++)
    {
        cin >> AT[i];
        cin >> BT[i];
    }
    int n = sizeof(AT) / sizeof(AT[0]);
    for (int i = 0; i < n; i++)
    {
        vect.push_back(make_pair(AT[i], BT[i]));
    }
    sort(vect.begin(), vect.end());
    int cmpl_T = 0;
    Total = 0;
    for (int i = 0; i < n; i++)
    {
        int tt = 0;
        cmpl_T += vect[i].second;
        tt = cmpl_T - vect[i].first;
        TT.push_back(tt);
        Total += TT[i];
    }
    float Avg_TT = Total / (float)n;
    float total_wt = 0;
    for (int i = 0; i < n; i++)
    {
        int wt = 0;
        if (i == 0)
        {
            wt = TT[i] - vect[i].second;
            WT.push_back(wt);
        }
        else
        {
```

```
            wt = TT[i] - vect[i].second;
            WT.push_back(wt);
            total_wt += WT[i];
        }
    }
    float Avg_WT = total_wt / n;
    printf("Process\tWT \tTAT\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t%.2f\t%.2f\n", i + 1, WT[i], TT[i]);
    }
    printf("Average turn around time is : %.2f\n", Avg_TT);
    printf("Average waiting time is : %.2f\n", Avg_WT);
    return 0;
}
```

Output -

```
Enter no processess: 5
Now enter the times in order of->
AT BT
0 4
1 3
2 1
3 2
4 5
Process WT      TAT
1       0.00    4.00
2       3.00    6.00
3       5.00    6.00
4       5.00    7.00
5       6.00    11.00
Average turn around time is : 6.80
Average waiting time is : 3.80
```

Shortest Job First (SJF)

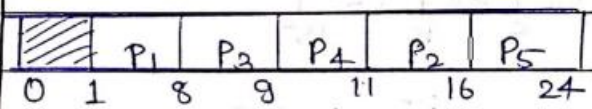
Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Example.

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
1	1	7	8	7	0
2	2	5	16	14	9
3	3	1	9	6	5
4	4	2	11	7	5
5	5	8	24	19	11

Gantt chart



Average Turnaround Time = 10.6

Average Waiting Time = 6.

Code –

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    vector<pair<float, float>> vect;
    vector<pair<float, float>> vect1;
    vector<pair<float, float>>::iterator it;
    vector<float> CT;
    vector<float> TT;
    vector<float> WT;
    int burst = 0, Total = 0;
    int p;
    cout << "Enter no processess: ";
    cin >> p;
    float AT[p], BT[p];
    cout << "AT BT\n";
    for (int i = 0; i < p; i++)
    {
        cin >> AT[i];
        cin >> BT[i];
    }
    int n = sizeof(AT) / sizeof(AT[0]);
    for (int i = 0; i < n; i++)
    {
        vect.push_back(make_pair(AT[i], BT[i]));
    }
    sort(vect.begin(), vect.end());
    int cmpl_T = vect[0].first + vect[0].second;
    vect1.push_back(make_pair(vect[0].first, vect[0].second));
    CT.push_back(cmpl_T);
    vect.erase(vect.begin());
    int min = 999;
    int index;
    int Total1 = 0;
    Total = 0;
    while (vect.size() > 0)
    {
        it = vect.begin();
        min = 999;
        for (int i = 0; i < vect.size(); i++)
        {
            if (vect[i].first <= cmpl_T && min > vect[i].second)
            {
                min = vect[i].second;
```

```
                index = i;
            }
        }
        if (min == 999)
        {
            for (int i = 0; i < vect.size(); i++)
            {
                if (min > vect[i].second)
                {
                    min = vect[i].second;
                    index = i;
                }
            }
            cmpl_T = vect[index].first + vect[index].second;
        }
        else
        {
            cmpl_T += vect[index].second;
        }
        CT.push_back(cmpl_T);
        int at = vect[index].first;
        int bt = vect[index].second;
        vect1.push_back(make_pair(at, bt));
        vect.erase(it + index);
    }
    for (int i = 0; i < n; i++)
    {
        int tt = CT[i] - vect1[i].first;
        int wt = tt - vect1[i].second;
        TT.push_back(tt);
        WT.push_back(wt);
        Total += TT[i];
        Total1 += WT[i];
    }
    float Avg_TT = Total / (float)n;
    float Avg_WT = Total1 / (float)n;
    printf("Process\t\tAT\tBT\tCT\tTT\tWT\n");
    for (int i = 0; i < vect1.size(); i++)
    {
        printf("%d\t\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", i + 1,
vect1[i].first, vect1[i].second,
               CT[i], TT[i], WT[i]);
    }
    printf("Average waiting time is : %.2f\n", Avg_WT);
    printf("Average turn around time is : %.2f\n", Avg_TT);
    return 0;
}
```

Output -

```
Enter no processess: 5
AT BT
1 7
2 5
3 1
4 2
5 8
Process         AT      BT      CT      TT      WT
1               1.00    7.00    8.00    7.00    0.00
2               3.00    1.00    9.00    6.00    5.00
3               4.00    2.00    11.00   7.00    5.00
4               2.00    5.00    16.00   14.00   9.00
5               5.00    8.00    24.00   19.00   11.00
Average waiting time is : 6.00
Average turn around time is : 10.60
```

Conclusion –

We have completed the execution of the scheduling algorithms FCFS and SJF(Non-preemptive)

Operating Systems

Preemptive scheduling assignment

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

CPU scheduling algorithms like Preemptive Priority, Round Robin etc.

Problem Statement -

1. Perform comparative assessment of various Scheduling Policies like
2. Priority preemptive and Round Robin.

Theory -

Scheduling algorithms are used when more than one process is executable and the OS has to decide which one to run first.

Terms used

1. Submit time: The process at which the process is given to CPU
2. Burst time: The amount of time each process takes for execution
3. Response time: The difference between the time when the process starts execution and the submit time.
4. Turnaround time: The difference between the time when the process completes execution and the submit time.

Priority Scheduling

Each process is assigned a priority and executable process with highest priority is allowed to run

Code –

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long int
void pro_sort(vector<pair<pair<ll, ll>, pair<ll, ll>>> &p)
{
    for (int i = 0; i < p.size() - 1; i++)
    {
        for (int j = 0; j < p.size() - i - 1; j++)
        {
            if (p[j].second.first > p[j + 1].second.first)
            {
                swap(p[j], p[j + 1]);
            }
        }
    }
}
int main()
{
    ll n, calc = 0;
    cout << "Number of processes: ";
    cin >> n;
    cout << "Process"
         << " "
         << "Priority"
         << " "
         << "AT"
         << " "
         << "BT" << endl;
    vector<pair<pair<ll, ll>, pair<ll, ll>>> process, copy;
    for (int i = 0; i < n; i++)
    {
        int pro, pri, at, bt;
        cin >> pro >> pri >> at >> bt;
        process.push_back({{pro, pri}, {at, bt}});
        calc += bt;
    }
    copy = process;
    pro_sort(copy);
    pro_sort(process);
```

```
    unordered_map<ll, ll> ma;
    int flag = 1;
    int t = min(process[1].second.first, process[0].second.first +
process[0].second.second);
    process[0].second.second -= (t - process[0].second.first);
    int start = process[0].second.first;
    calc += start;
    vector<pair<ll, ll>> gc;
    gc.push_back({process[0].first.first, t});
    while (t < calc)
    {
        int maxp = INT_MIN, idx;
        for (int i = 0; i < n; i++)
        {
            int pro = process[i].first.first;
            int pri = process[i].first.second;
            int at = process[i].second.first;
            if (ma[pro] != 1 && at <= t)
            {
                if (pri > maxp)
                {
                    maxp = pri;
                    idx = i;
                }
            }
        }
        int pro = process[idx].first.first;
        int pri = process[idx].first.second;
        int at = process[idx].second.first;
        int prev = gc[gc.size() - 1].second;
        gc.push_back({process[idx].first.first, prev + 1});
        process[idx].second.second -= 1;
        t += 1;
        if (process[idx].second.second == 0)
        {
            ma[process[idx].first.first] = 1;
        }
    }
    unordered_map<ll, ll> m;
    vector<ll> ct(n + 1), tat(n + 1), wt(n + 1);
    for (int i = gc.size() - 1; i >= 0; i--)
    {
        if (m[gc[i].first] == 0)
        {
            ct[gc[i].first] = gc[i].second;
            m[gc[i].first] = 1;
        }
    }
    for (int i = 0; i < n; i++)
```

```
    {
        int pro = process[i].first.first;
        tat[pro] = ct[pro] - process[i].second.first;
        wt[pro] = tat[pro] - copy[i].second.second;
    }
    float avgt = 0, avgw = 0;
    cout << "Process\t\t"
         << "Priority\t"
         << "AT\t"
         << "BT\t"
         << "CT\t"
         << "TAT\t"
         << "WT\t" << endl;
    for (int i = 0; i < n; i++)
    {
        int pro = copy[i].first.first;
        cout << copy[i].first.first << "\t\t" << copy[i].first.second <<
"\t\t" << copy[i].second.first << "\t" << copy[i].second.second << "\t" <<
ct[pro] << "\t" << tat[pro] << "\t" << wt[pro] << endl;
        avgt += tat[pro];
        avgw += wt[pro];
    }
    cout << "Average tat: " << avgt * 1.0 / n << endl;
    cout << "Average wt: " << avgw * 1.0 / n << endl;
}
```

Output -

```
Kartik:OS Exp 6 - Preemptive Scheduling(SJF, RR)/ (master) $ c++ priorityScheduling.cpp -o priorityScheduling
Kartik:OS Exp 6 - Preemptive Scheduling(SJF, RR)/ (master) $ ./priorityScheduling.exe
Number of processes: 4
Process Priority AT BT
1       12       0  4
2       2        2  8
3       5        6  3
4       7        4  9
Process         Priority        AT      BT      CT      TAT     WT
1               12              0       4       4       4       0
2               2               2       8       24      22      14
4               7               4       9       13      9       0
3               5               6       3       16      10      7
Average tat: 11.25
Average wt: 5.25
```

Round Robin (RR)

- Each process is assigned a time interval called its quantum (time slice)
- If the process is still running at the end of the quantum the CPU is preempted and given to another process, and this continues in circular fashion, till all the processes are completely executed

Code –

```
#include <iostream>
#include <cstdlib>
#include <queue>
#include <cstdio>
using namespace std;
/* C++ Program to Round Robin*/
typedef struct process
{
    int id, at, bt, st, ft, pr;
    float wt, tat;
} process;
process p[10], p1[10], temp;
queue<int> q1;
int accept(int ch);
void turnwait(int n);
void display(int n);
void ganttrr(int n);
int main()
{
    int i, n, ts, ch, j, x;
    p[0].tat = 0;
    p[0].wt = 0;
    n = accept(ch);
    ganttrr(n);
    turnwait(n);
    display(n);
    return 0;
}
int accept(int ch)
{
    int i, n;
    printf("Enter the Total Number of Process: ");
    scanf("%d", &n);
    if (n == 0)
    {
        printf("Invalid");
        exit(1);
```

```
    }
    cout << endl;
    for (i = 1; i <= n; i++)
    {
        printf("Enter an Arrival Time of the Process P%d: ", i);
        scanf("%d", &p[i].at);
        p[i].id = i;
    }
    cout << endl;
    for (i = 1; i <= n; i++)
    {
        printf("Enter a Burst Time of the Process P%d: ", i);
        scanf("%d", &p[i].bt);
    }
    for (i = 1; i <= n; i++)
    {
        p1[i] = p[i];
    }
    return n;
}
void ganttrr(int n)
{
    int i, ts, m, nextval, nextarr;
    nextval = p1[1].at;
    i = 1;
    cout << "\nEnter the Time Slice or Quantum: ";
    cin >> ts;
    for (i = 1; i <= n && p1[i].at <= nextval; i++)
    {
        q1.push(p1[i].id);
    }
    while (!q1.empty())
    {
        m = q1.front();
        q1.pop();
        if (p1[m].bt >= ts)
        {
            nextval = nextval + ts;
        }
        else
        {
            nextval = nextval + p1[m].bt;
        }
        if (p1[m].bt >= ts)
        {
            p1[m].bt = p1[m].bt - ts;
        }
        else
        {
```

```
            p1[m].bt = 0;
        }
        while (i <= n && p1[i].at <= nextval)
        {
            q1.push(p1[i].id);
            i++;
        }
        if (p1[m].bt > 0)
        {
            q1.push(m);
        }
        if (p1[m].bt <= 0)
        {
            p[m].ft = nextval;
        }
    }
}
void turnwait(int n)
{
    int i;
    for (i = 1; i <= n; i++)
    {
        p[i].tat = p[i].ft - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        p[0].tat = p[0].tat + p[i].tat;
        p[0].wt = p[0].wt + p[i].wt;
    }
    p[0].tat = p[0].tat / n;
    p[0].wt = p[0].wt / n;
}
void display(int n)
{
    int i;
    /*
    Here
    at = Arrival time,
    bt = Burst time,
    time_quantum= Quantum time
    tat = Turn around time,
    wt = Waiting time
    */
    cout << "\n=========================================================\n";
    cout << "\n\nHere AT = Arrival Time\nBT = Burst Time\nTAT = Turn Around
Time\nWT = Waiting Time\n ";

    cout << "\n====================TABLE=============================\n";
    printf("\nProcess\tAT\tBT\tFT\tTAT\t\tWT");
    for (i = 1; i <= n; i++)
    {
```

```
        printf("\nP%d\t%d\t%d\t%d\t%f\t%f", p[i].id, p[i].at, p[i].bt,
p[i].ft, p[i].tat, p[i].wt);
    }
    cout << "\n=================================================\
n";
    printf("\nAverage Turn Around Time: %f", p[0].tat);
    printf("\nAverage Waiting Time: %f\n", p[0].wt);
}
```

Output –

```
Kartik:OS Exp 6 - Preemptive Scheduling(SJF, RR)/ (master) $ ./RR
Enter the Total Number of Process: 5

Enter an Arrival Time of the Process P2: 3
Enter an Arrival Time of the Process P3: 6
Enter an Arrival Time of the Process P4: 4
Enter an Arrival Time of the Process P5: 1

Enter a Burst Time of the Process P1: 5
Enter a Burst Time of the Process P2: 3
Enter a Burst Time of the Process P3: 78
Enter a Burst Time of the Process P4: 3
Enter a Burst Time of the Process P5: 8

Enter the Time Slice or Quantum: 3

=================================================


Here AT = Arrival Time
BT = Burst Time
TAT = Turn Around Time
WT = Waiting Time

==================TABLE=============================

Process AT      BT      FT      TAT             WT
P1      2       5       10      8.000000        3.000000
P2      3       3       8       5.000000        2.000000
P3      6       78      99      93.000000       15.000000
P4      4       3       16      12.000000       9.000000
P5      1       8       30      29.000000       21.000000
=================================================n
Average Turn Around Time: 29.400000
Average Waiting Time: 10.000000
```

OS - Experiment 5

Name: Kartik Jolapara Sapid: 60004200107

Div./Batch: B/B1 Branch: Computer Engineering

AIM:

To implement various memory allocation techniques like first fit, best fit and worst fit.

THEORY:

- **First fit**

This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory space more than or equal to its size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Code:

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

int main()
{
    int noOfPartitions;
    cout << "Enter the number of patitions: ";
    cin >> noOfPartitions;
    int partitionMemory[noOfPartitions], tempPMemory[noOfPartitions];
    for (int i = 0; i < noOfPartitions; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> partitionMemory[i];
```

```
        tempPMemory[i] = partitionMemory[i];
    }
    int noOfProcesses;
    cout << "Enter the number of processes: ";
    cin >> noOfProcesses;
    int processesMemory[noOfProcesses];
    for (int i = 0; i < noOfProcesses; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> processesMemory[i];
    }
    string firstFit[noOfPartitions], notAllocatedProcesses;
    for (int i = 0; i < noOfPartitions; i++)
    {
        firstFit[i] = "X";
    }
    for (int i = 0; i < noOfProcesses; i++)
    {
        bool isAlloc = false;
        for (int j = 0; j < noOfPartitions; j++)
        {
            if (processesMemory[i] <= partitionMemory[j])
            {
                partitionMemory[j] -= processesMemory[i];
                if (firstFit[j] == "X")
                {

                    firstFit[j] = "P" + to_string(i + 1);
                }
                else
                {
                    firstFit[j] += ", P" + to_string(i + 1);
                }
                isAlloc = true;
                break;
            }
        }
        if (!isAlloc && notAllocatedProcesses.empty())
        {
            notAllocatedProcesses = "P" + to_string(i + 1);
        }
        else if (!isAlloc)
        {
            notAllocatedProcesses += ", P" + to_string(i + 1);
        }
    }
    cout << "\nPartitions\t\tFirst Fit\n";
    for (int i = 0; i < noOfPartitions; i++)
```

```
    {
        cout << tempPMemory[i] << "\t\t\t" << firstFit[i] << "\n";
    }
    if (notAllocatedProcesses.empty())
    {
        cout << "There are no unallocated processes!\n";
    }
    else
    {
        cout << "The unallocated processes are: " <<
notAllocatedProcesses << "\n";
    }

    return 0;
}
```

Output:

```
Enter the number of patitions: 4
Partition 1: 10
Partition 2: 50
Partition 3: 30
Partition 4: 20
Enter the number of processes: 3
Partition 1: 20
Partition 2: 60
Partition 3: 10

Partitions              First Fit
10                      P3
50                      P1
30                      X
20                      X
The unallocated processes are: P2
```

- **Best fit**

This method keeps the free/busy list in order by size-smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closestfitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.

Code:

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

int main()
{
    int noOfPartitions;
    cout << "Enter the number of patitions: ";
    cin >> noOfPartitions;
    int partitionMemory[noOfPartitions], tempPMemory[noOfPartitions];
    for (int i = 0; i < noOfPartitions; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> partitionMemory[i];
        tempPMemory[i] = partitionMemory[i];
    }
    int noOfProcesses;
    cout << "Enter the number of processes: ";
    cin >> noOfProcesses;
    int processesMemory[noOfProcesses];
    for (int i = 0; i < noOfProcesses; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> processesMemory[i];
    }
    string bestFit[noOfPartitions], notAllocatedProcesses;
    for (int i = 0; i < noOfPartitions; i++)
    {
        bestFit[i] = "X";
    }
    for (int i = 0; i < noOfProcesses; i++)
    {
        bool isAlloc = false;
        int minSize;
        for (int j = 0; j < noOfPartitions; j++)
        {
            if (processesMemory[i] <= partitionMemory[j] && !isAlloc)
            {
```

```
                minSize = j;
                isAlloc = true;
            }
            else if (processesMemory[i] <= partitionMemory[j])
            {
                if (partitionMemory[j] < partitionMemory[minSize])
                {
                    minSize = j;
                }
            }
        }
        if (isAlloc)
        {
            partitionMemory[minSize] -= processesMemory[i];
            if (bestFit[minSize] == "X")
            {

                bestFit[minSize] = "P" + to_string(i + 1);
            }
            else
            {
                bestFit[minSize] += ", P" + to_string(i + 1);
            }
            isAlloc = true;
        }
        else if (!isAlloc && notAllocatedProcesses.empty())
        {
            notAllocatedProcesses = "P" + to_string(i + 1);
        }
        else if (!isAlloc)
        {
            notAllocatedProcesses += ", P" + to_string(i + 1);
        }
    }
    cout << "\nPartitions\t\tBest Fit\n";
    for (int i = 0; i < noOfPartitions; i++)
    {
        cout << tempPMemory[i] << "\t\t\t" << bestFit[i] << "\n";
    }
    if (notAllocatedProcesses.empty())
    {
        cout << "There are no unallocated processes!\n";
    }
    else
    {
        cout << "The unallocated processes are: " <<
notAllocatedProcesses << "\n";
    }
```

```
    return 0;
}
```

Output:

```
Enter the number of patitions: 4
Partition 1: 10
Partition 2: 50
Partition 3: 30
Partition 4: 20
Enter the number of processes: 3
Partition 1: 20
Partition 2: 60
Partition 3: 10

Partitions              Best Fit
10                      P3
50                      X
30                      X
20                      P1
The unallocated processes are: P2
```

- **Worst fit**

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole.

Code:

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

int main()
{
    int noOfPartitions;
    cout << "Enter the number of patitions: ";
    cin >> noOfPartitions;
    int partitionMemory[noOfPartitions], tempPMemory[noOfPartitions];
    for (int i = 0; i < noOfPartitions; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> partitionMemory[i];
        tempPMemory[i] = partitionMemory[i];
    }
    int noOfProcesses;
    cout << "Enter the number of processes: ";
    cin >> noOfProcesses;
    int processesMemory[noOfProcesses];
    for (int i = 0; i < noOfProcesses; i++)
    {
        cout << "Partition " << i + 1 << ": ";
        cin >> processesMemory[i];
    }
    string worstFit[noOfPartitions], notAllocatedProcesses;
    for (int i = 0; i < noOfPartitions; i++)
    {
        worstFit[i] = "X";
    }
    for (int i = 0; i < noOfProcesses; i++)
    {
        bool isAlloc = false;
        int maxSize;
        for (int j = 0; j < noOfPartitions; j++)
        {
            if (processesMemory[i] <= partitionMemory[j] && !isAlloc)
            {
                maxSize = j;
```

```
                isAlloc = true;
            }
            else if (processesMemory[i] <= partitionMemory[j])
            {
                if (partitionMemory[j] > partitionMemory[maxSize])
                {
                    maxSize = j;
                }
            }
        }
        if (isAlloc)
        {
            partitionMemory[maxSize] -= processesMemory[i];
            if (worstFit[maxSize] == "X")
            {

                worstFit[maxSize] = "P" + to_string(i + 1);
            }
            else
            {
                worstFit[maxSize] += ", P" + to_string(i + 1);
            }
            isAlloc = true;
        }
        else if (!isAlloc && notAllocatedProcesses.empty())
        {
            notAllocatedProcesses = "P" + to_string(i + 1);
        }
        else if (!isAlloc)
        {
            notAllocatedProcesses += ", P" + to_string(i + 1);
        }
    }
    cout << "\nPartitions\t\tWorst Fit\n";
    for (int i = 0; i < noOfPartitions; i++)
    {
        cout << tempPMemory[i] << "\t\t\t" << worstFit[i] << "\n";
    }
    if (notAllocatedProcesses.empty())
    {
        cout << "There are no unallocated processes!\n";
    }
    else
    {
        cout << "The unallocated processes are: " <<
notAllocatedProcesses << "\n";
    }
```

```
    return 0;
}
```

Output:

```
Enter the number of patitions: 4
Partition 1: 10
Partition 2: 50
Partition 3: 30
Partition 4: 20
Enter the number of processes: 3
Partition 1: 20
Partition 2: 60
Partition 3: 10

Partitions              Worst Fit
10                      X
50                      P1, P3
30                      X
20                      X
The unallocated processes are: P2
```

Operating Systems

Experiment 6 – Producer Consumer

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

To study the concept of semaphore and solve producer consumer problem using semaphore.

Theory -

Process Synchronization

Process Synchronization is a way to coordinate processes that use shared data. It occurs in an operating system among cooperating processes. Cooperating processes are processes that share resources. While executing many concurrent processes, process synchronization helps to maintain shared data consistency and cooperating process execution. Processes have to be scheduled to ensure that concurrent access to shared data does not create inconsistencies. Data inconsistency can result in what is called a race condition. A race condition occurs when two or more operations are executed at the same time, not scheduled in the proper sequence, and not exited in the critical section correctly

Semaphore

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization. The definitions of wait and signal are as follows –

- Wait

  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
  while (S<=0);

  S--;
}
```

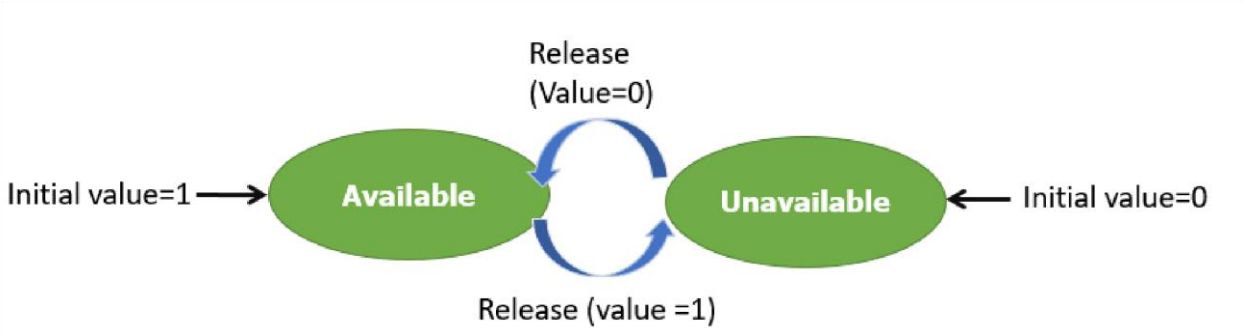
- Signal

  The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

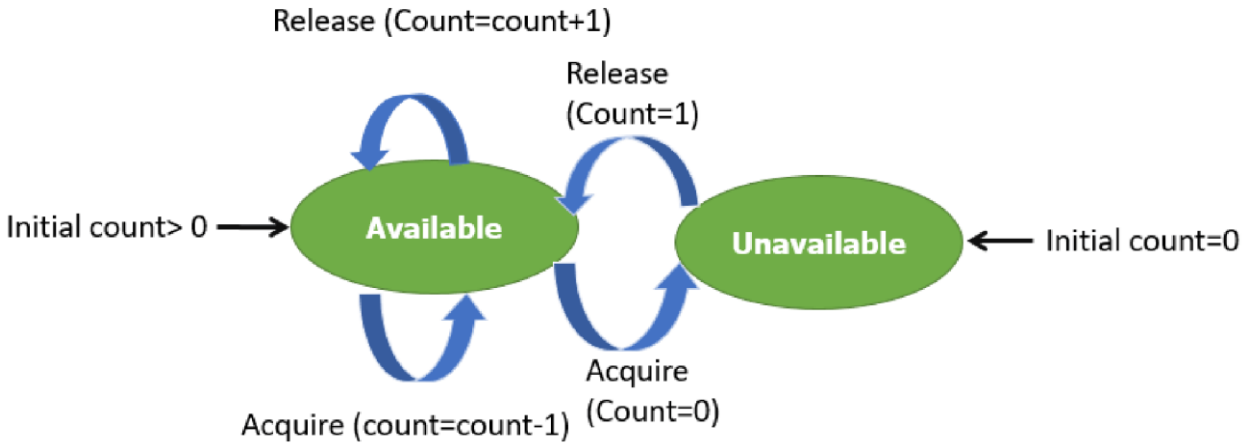
Semaphores are of two types:

1. **Binary Semaphore –**
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.



2. **Counting Semaphore –**

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.



Advantages of Semaphores:

Some of the advantages of semaphores are as follows –

1. Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
2. There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
3. Semaphores are implemented in the machine independent code of the microkernel. So, they are machine independent.

Disadvantages of Semaphores:

Some of the disadvantages of semaphores are as follows –

1. Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
2. Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
3. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Counting Semaphore vs. Binary Semaphore

Counting Semaphore	Binary Semaphore
No mutual exclusion	Mutual exclusion
Any integer value	Value only 0 and 1
More than one slot	Only one slot
Provide a set of Processes	It has a mutual exclusion mechanism.

Code –

```
#include <stdio.h>
int s = 0, n = 0, e = 4, b[4];
int Wait(int *s)
{
    return (--*s);
}
int Signal(int *s)
{
    return (++*s);
}
void producer()
{
    int a;
    printf("Enter value to Produce\n");
    scanf("%d", &a);
    Wait(&e);
    Wait(&s);
    b[n] = a;
    Signal(&s);
    Signal(&n);
    for (int i = 0; i < n; i++)
    {
        printf("%d ", b[i]);
    }
    printf("\n");
}
void consumer()
{
    Wait(&n);
```

```
    Wait(&s);
    for (int i = 1; i < n; i++)
    {
        b[i - 1] = b[i];
    }
    Signal(&s);
    Signal(&e);
    for (int i = 0; i < n; ++i)
    {
        printf("%d ", b[i]);
    }
    printf("\n");
}
int main()
{
    int c;
    printf("Hello\n");
    do
    {
        printf("Enter Choice\n");
        printf("1. Produce \n2. Consume \n3. Exit\n");
        scanf("%d", &c);
        switch (c)
        {
        case 1:
            if (e == 0)
            {
                printf("Buffer is full\n");
            }
            else
            {
                producer();
            }
            break;
        case 2:
            if (e == 4)
            {
                printf("Buffer is empty\n");
            }
            else
            {
                consumer();
            }
            break;
        case 3:
            break;
        default:
            printf("Invalid Choice\n");
        }
```

```
    } while (c != 3);
    return 0;
}
```

Output -

```
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
1
1
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
2
1 2
Enter Choice
1. Produce
2. Consume
3. Exit
1
```

```
Enter value to Produce
3
1 2 3
Enter Choice
1. Produce
2. Consume
3. Exit
2
1 2
Enter Choice
1. Produce
2. Consume
3. Exit
2
1
Enter Choice
1. Produce
2. Consume
3. Exit
```

```
2
Enter Choice
1. Produce
2. Consume
3. Exit
2
Buffer is empty
Enter Choice
1. Produce
2. Consume
3. Exit
```

```
Enter Choice
1. Produce
2. Consume
3. Exit
1
Enter value to Produce
4
1 2 3 4
Enter Choice
1. Produce
2. Consume
3. Exit
1
Buffer is full
Enter Choice
1. Produce
2. Consume
3. Exit
```

Conclusion -

Learnt and understood the concept of semaphore and solved producer consumer problem using semaphore.

Operating Systems

Experiment – Banker's Algorithm

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

To apply bankers algorithm and check whether the system is in deadlock or not.

Theory –

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes.

The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the Banker's Algorithm in detail. Also, we will solve problems based on the Banker's Algorithm. To understand the Banker's Algorithm first we will see a real word example of it. Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'.

If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system. Similarly, it works in an operating system.

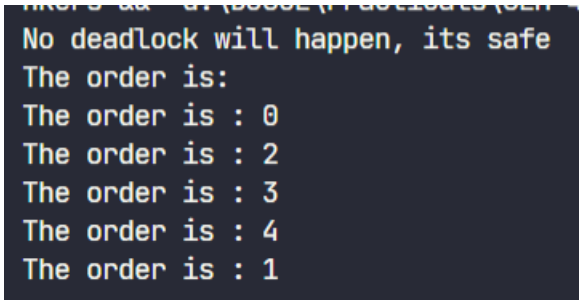
When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.

Code –

```
#include <stdio.h>
#include <stdbool.h>
#define process 5
#define resource 4

void isSafe(int processes[], int available[], int
max_req[process][resource], int allocated[process][resource])
{
    int remaining_need[process][resource];
    int order[process], index = 0;
    for (int i = 0; i < process; i++)
    {
        for (int j = 0; j < resource; j++)
        {
            remaining_need[i][j] = max_req[i][j] - allocated[i][j];
        }
    }
    bool check[process] = {false, false, false, false, false};
    for (int i = 0; i < process; i++)
    {
        for (int j = 0; j < process; j++)
        {
            int count = 0;
            if (check[j] == false)
            {
                for (int k = 0; k < resource; k++)
                {
                    if (available[k] >= remaining_need[j][k])
                    {
                        count++;
                    }
                    else
                    {
                        break;
                    }
                }
                if (count == resource)
                {
```

```
                    check[j] = true;
                    for (int l = 0; l < resource; l++)
                    {
                        available[l] += allocated[j][l];
                    }
                    order[index] = j;
                    index++;
                }
            }
            else
            {
                continue;
            }
        }
    }
    if (index == process)
    {
        printf("No deadlock will happen, its safe\nThe order is: \n");
        for (int p = 0; p < process; p++)
        {
            printf("The order is : %d \n", order[p]);
        }
    }
    else
    {
        printf("deadlock will happen, its unsafe\n");
    }
}
int main()
{
    int processes[process] = {0, 1, 2, 3, 4};
    int available[process] = {1, 5, 2, 0};
    int allocated[process][resource] = {{0, 0, 1, 2},
                                        {1, 0, 0, 0},
                                        {1, 3, 5, 4},
                                        {0, 6, 3, 2},
                                        {0, 0, 1, 4}};
    int max_req[process][resource] = {{0, 0, 1, 2},
                                      {1, 7, 5, 0},
                                      {2, 3, 5, 6},
                                      {0, 6, 5, 2},
                                      {0, 6, 5, 6}};
    isSafe(processes, available, max_req, allocated);
}
```

Output –



Conclusion -

Banker's Algorithm has been successfully implemented. Banker's Algo helps the operating system manage and process control request for each type of resource in the computer system.

OS Experiment - 8

Name: Kartik Jolapara **SAP ID:60004200107** **Batch: B1**

Aim- Using the CPU-OS simulator to analyze and synthesize the following:

a. Process Scheduling algorithms.
b. Thread creation and synchronization.
c. Deadlock prevention and avoidance.

Problem Statement:

1) Install CPU-OS simulator
2) Perform the following steps

a) Process Scheduling algorithms

Loading and Compiling Program

You need to create some executable code so that it can be run by the CPU simulator under the control of the OS simulator. In order to create this code, you need to use the compiler which is part of the system simulator. This compiler is able to compile simple high-level source statements similar to Visual Basic. To do this, open the compiler window by selecting the **COMPILER…** button in the current window. You should now be looking at the compiler window. In the compiler window, enter the following source code in the compiler's source editor window (under **PROGRAM SOURCE** frame title):

```
program LoopTest
      i = 0  for n = 0
      to 40  i = i + 1
      next  end
```

Now you need to compile this in order to generate the executable code. To do this, click on the **COMPILE…** button. You should see the code created on the right in **PROGRAM CODE** view. Make a habit of saving your source code.

Click on the button **SHOW…** in **BINARY CODE** view. You should now see the **Binary Code for LOOPTEST** window. Study the program code displayed in hexadecimal format.

Now, this code needs to be loaded in memory so that the CPU can execute it. To do this, first we need to specify a base address (in **ASSEMBLY CODE** view): uncheck the box next to the edit box with label **Base Address**, and then enter 100 in the edit box. Now, click on the **LOAD IN MEMORY…** button in the current window. You should now see the code loaded in memory ready to be executed. You are also back in the CPU simulator at this stage. This action is

equivalent to loading the program code normally stored on a disc drive into RAM on the real computer systems.

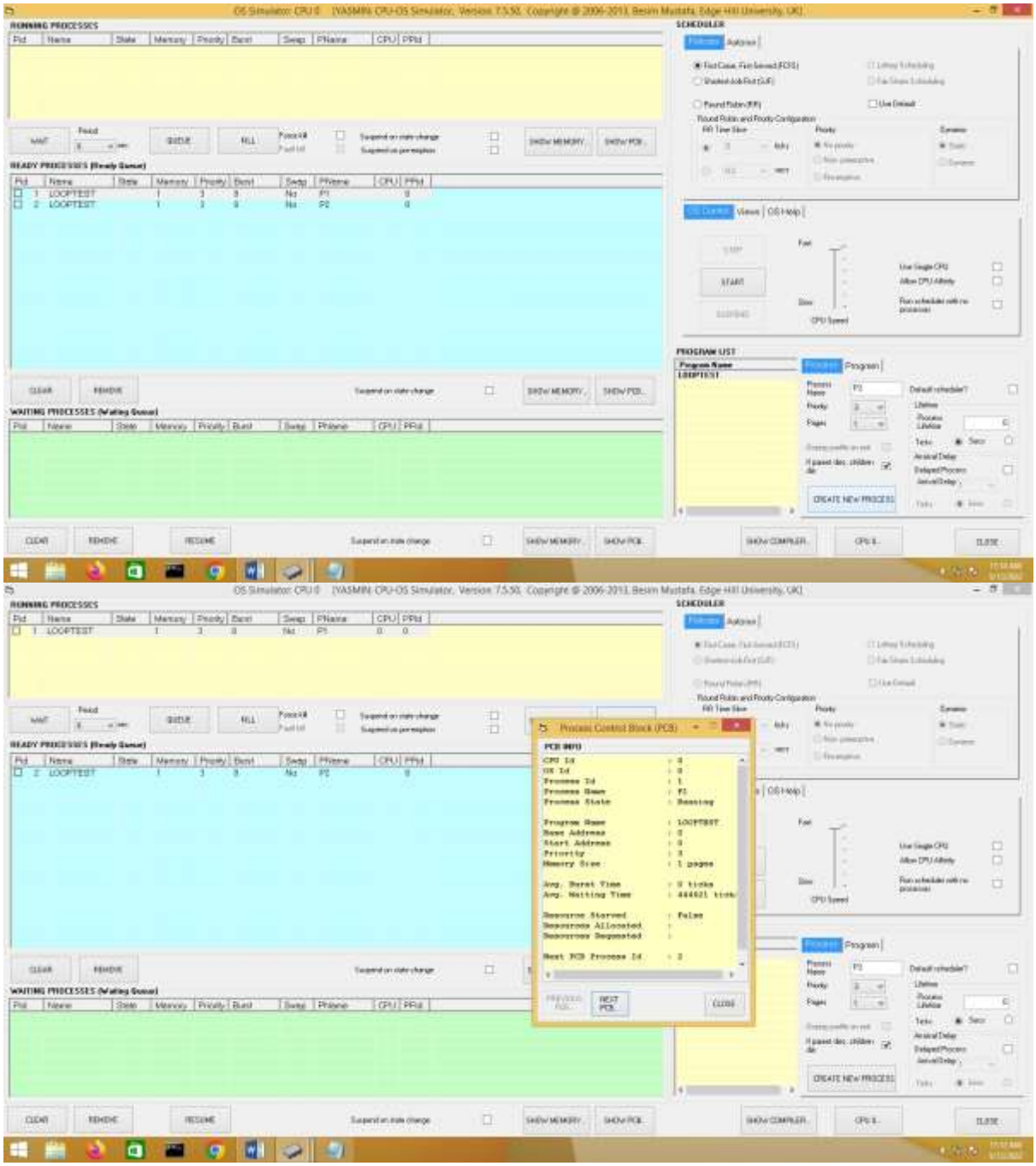
Creating processes from programs in the OS simulator.

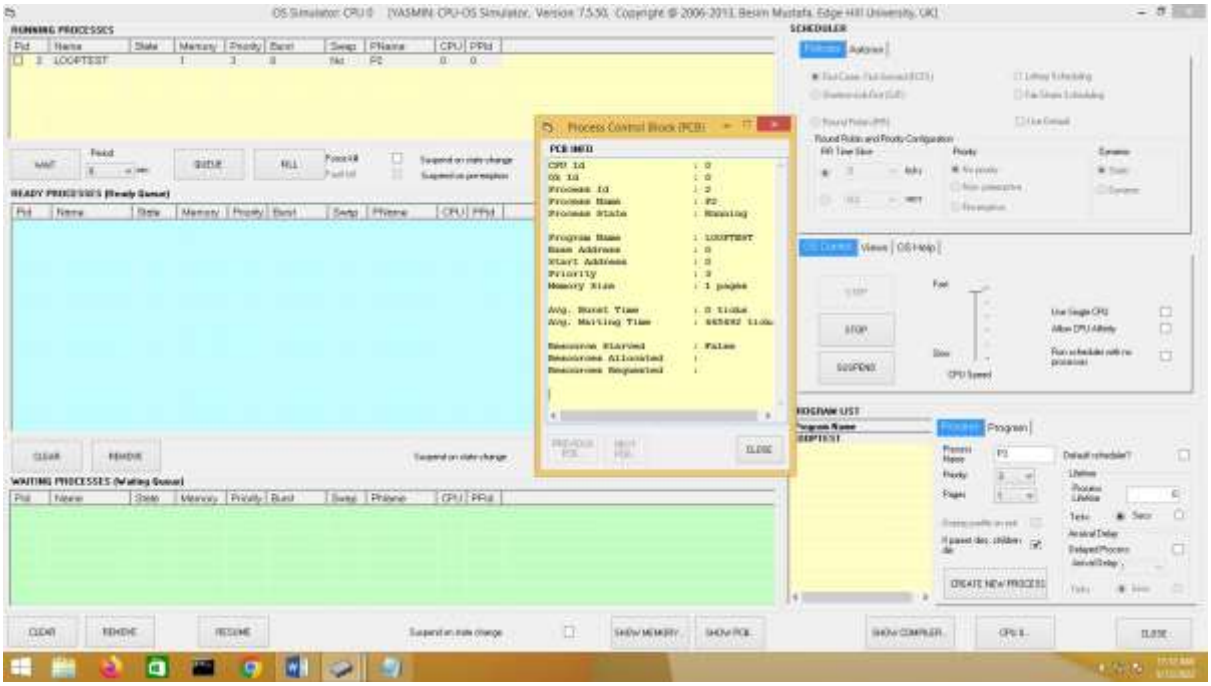
We are now going to use the OS simulator to run this code. To enter the OS simulator, click on the **OS 0…** button in the current window. The OS window opens. You should see an entry, titled **LoopTest**, in the **PROGRAM LIST** view. Now that this program is available to the OS simulator, we can create as many instances, i.e. processes, of it as we like. You do this by clicking on the **CREATE NEW PROCESS** button. Repeat this four times. Observe the four instances of the program being queued in the ready queue which is represented by the **READY PROCESSES** view.

NOTE: it is very important that you follow the instructions below without any deviation. If you do, then you must re-do the exercise from the beginning as any follow-up action(s) may give the wrong results.

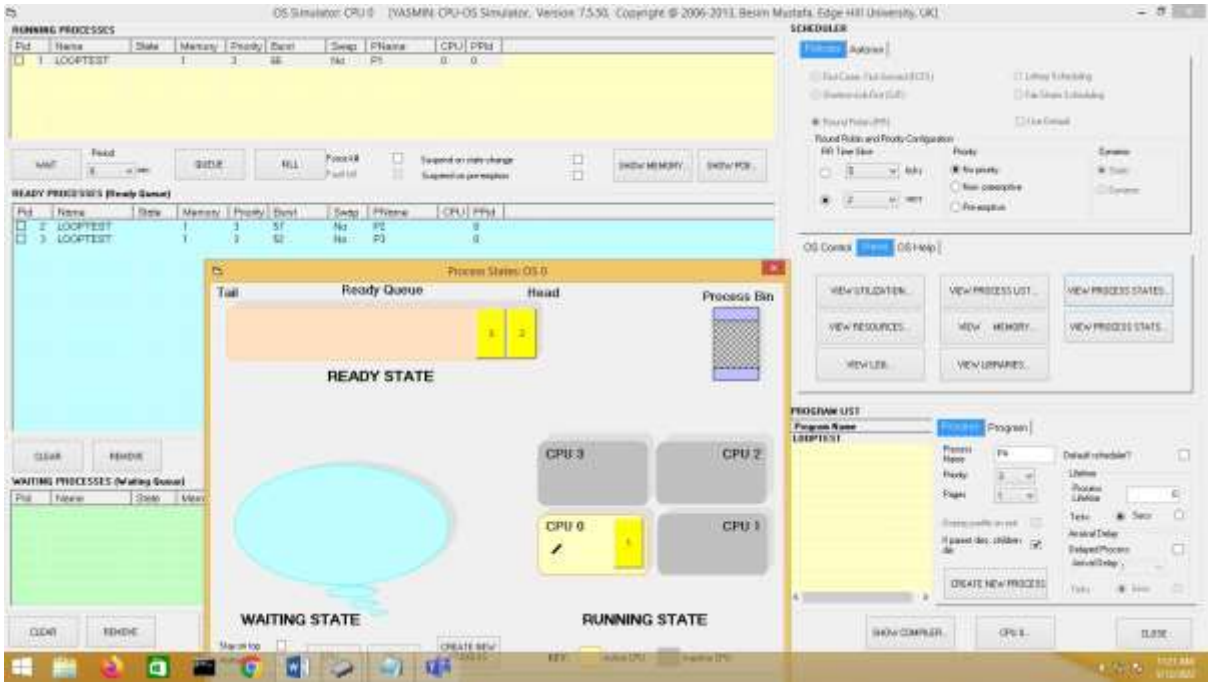
Selecting different scheduling policies and run the processes in the OS simulator

Make sure the **First-Come-First-Served (FCFS)** option is selected in the **SCHEDULER/Policies** view. At this point the OS is inactive. To activate, first move the **Speed** slider to the fastest position, then click on the **START** button. This should start the OS simulator running the processes. Observe the instructions executing in the CPU simulator window. Make a note of what you observe in the box below as the processes are run (you need to concentrate on the two views: **RUNNING PROCESSES** and the **READY PROCESSES** during this period).

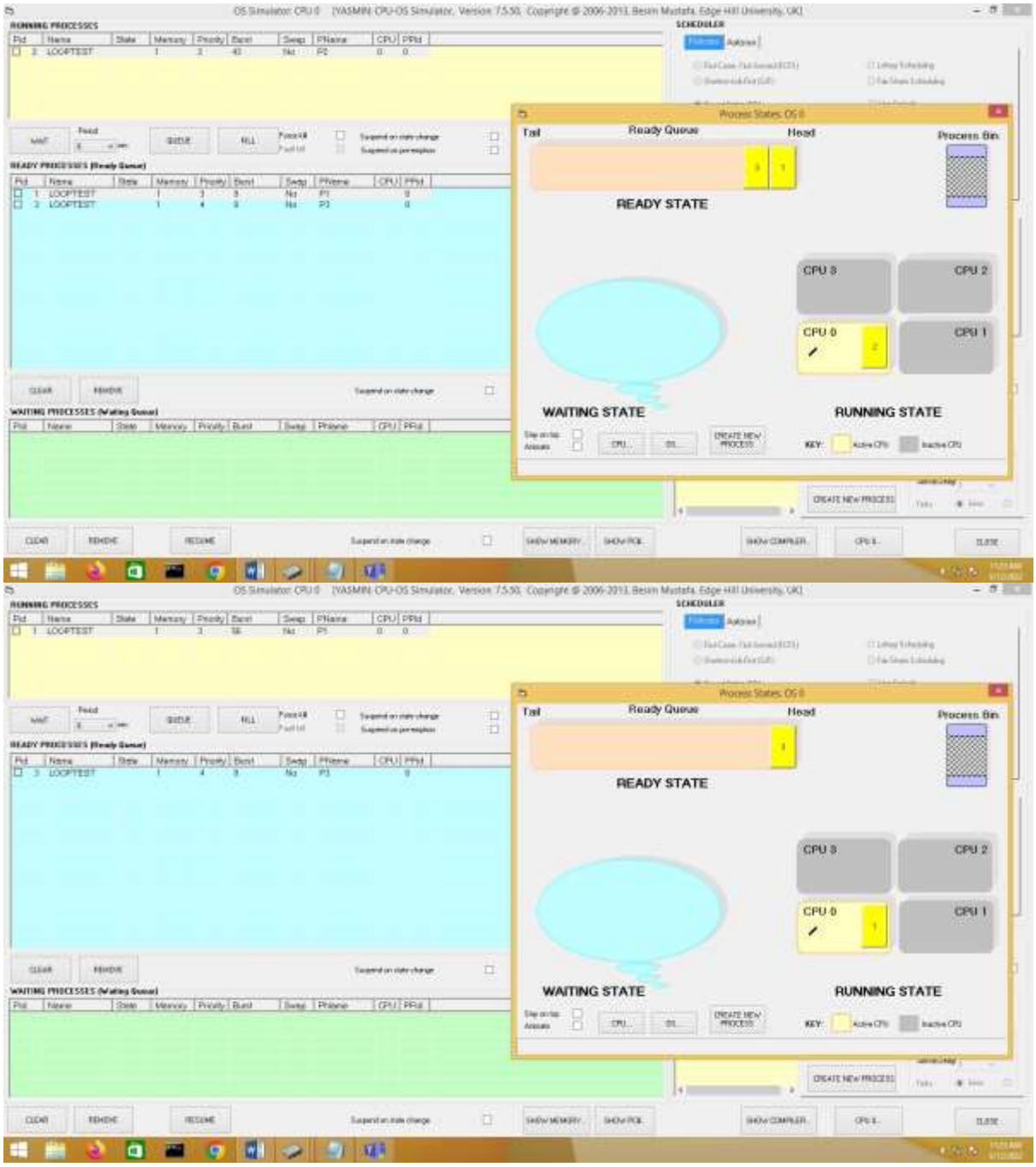

OS Simulator: CPU 0 [YASMIN: CPU-OS Simulator, Version 7.5.50, Copyright @ 2006-2013, Besim Mustafa, Edge Hill University, UK]
RUNNING PROCESSES
Pid
Name
State
Memory
Priority
Burst
Swap
PName
CPU
PPid
READY PROCESSES (Ready Queue)
1 LOOPTEST
2 LOOPTEST
WAITING PROCESSES (Waiting Queue)
SCHEDULER
Round Robin and Priority Configuration
OS Control
Views
OS Help
CPU Speed
PROGRAM LIST
Program Name
LOOPTEST
CREATE NEW PROCESS
CLEAR
REMOVE
RESUME
SHOW MEMORY
SHOW PCB
SHOW COMPILER
CLOSE
OS Simulator: CPU 0 [YASMIN: CPU-OS Simulator, Version 7.5.50, Copyright @ 2006-2013, Besim Mustafa, Edge Hill University, UK]
Process Control Block (PCB)
PCB INFO
CPU Id : 0
OS Id : 0
Process Id : 1
Process Name : P1
Process State : Running
Program Name : LOOPTEST
Base Address : 0
Start Address : 0
Priority : 3
Memory Size : 1 pages
Avg. Burst Time : 0 ticks
Avg. Waiting Time : 444621 ticks
Resources Starved : False
Resources Allocated :
Resources Requested :
Next PCB Process Id : 2
PREVIOUS PCB
NEXT PCB
CLOSE



When all the processes finish, do the following. Select **Round Robin (RR)** option in the **SCHEDULER/Policies** view. Then select the **No priority** option in the **SCHEDULER/Policies/Priority** frame. Create three processes. Click on the **START** button and observe the behaviors of the processes until they all complete. You may wish to use speed slider to slow down the processes to better see what is happening. Make a note of what you observed in the box below and compare this with the observation in step 1 above.

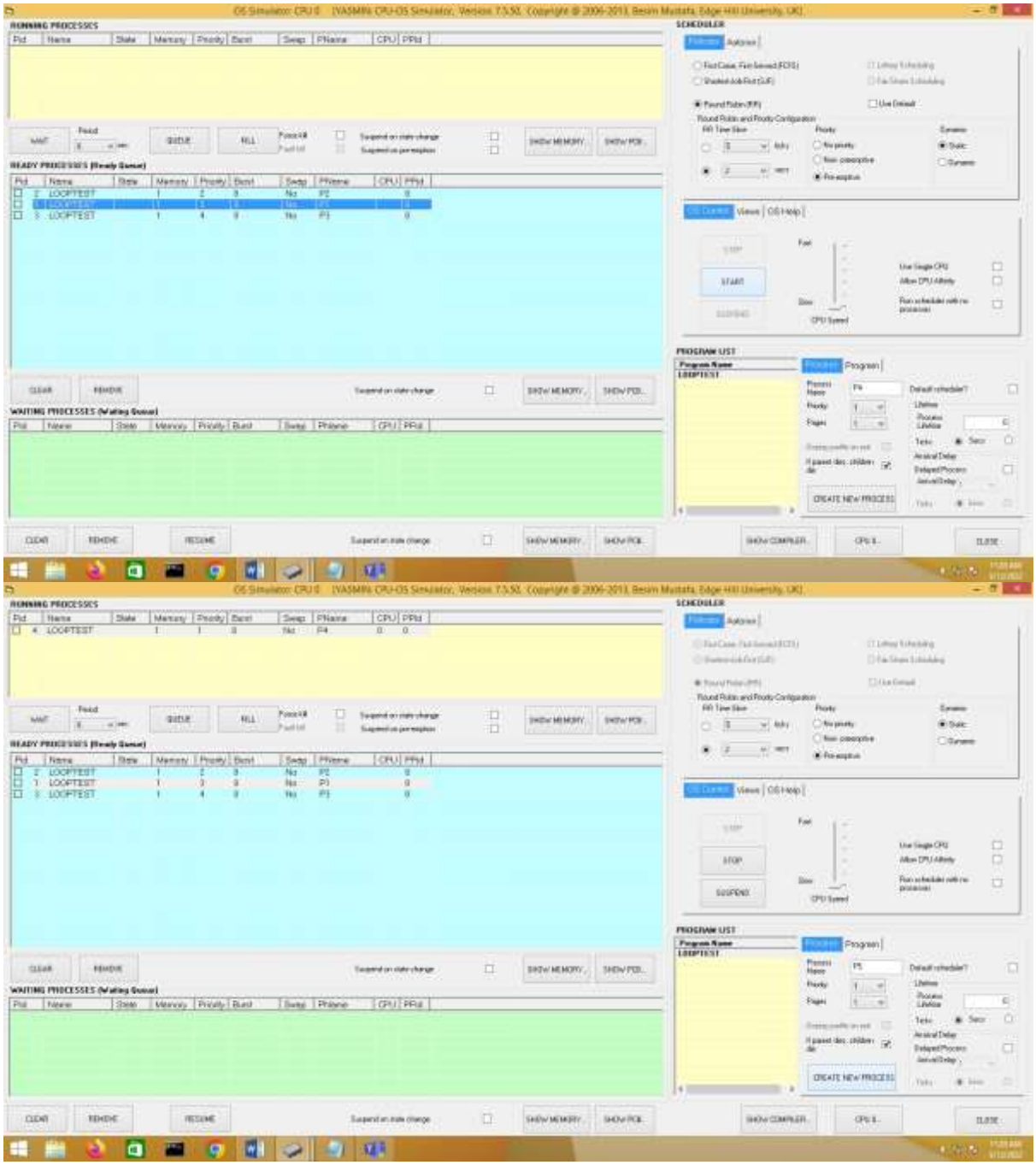


Then select the **Non-preemptive** priority option in the **SCHEDULER/Policies/Priority** frame. Create three processes with the following priorities: 3, 2 and 4. Use the **Priority** drop down list to select priorities. Observe the order in which the three processes are queued in the ready queue represented by the **READY PROCESSES** view and make a note of this in the box below (note that the lower the number the higher the priority is).



Slide the **Speed** selector to the slowest position and then hit the **START** button. While the first process is being run do the following. Create a fourth process with priority 1. Make a note of what you observe (pay attention to the **READY PROCESSES** view) in the box below.

Now kill all four processes one by one as they start running. Next, select the **Pre-emptive** option in the **SCHEDULER/Policies/Priority** frame. Create the same three processes as in step 3 and then hit the **START** button. While the first process is being run do the following. Create a fourth process with priority 1. Make a note of what you observe (pay attention to the **RUNNING PROCESSES** view). How is this behavior different than that in step 4 above?



Thread creation and synchronization.

Loading and Compiling a Program

In the compiler window enter the following source code:

```
program ThreadTest1  sub
     thread1 as thread
     writeln("In thread1")
     while true  wend
     end sub  sub thread2
     as thread  call
     thread1 writeln("In
```

```
        thread2")  while true
        wend
        end sub
        call thread2
        writeln("In main")  do
        loop
end
```

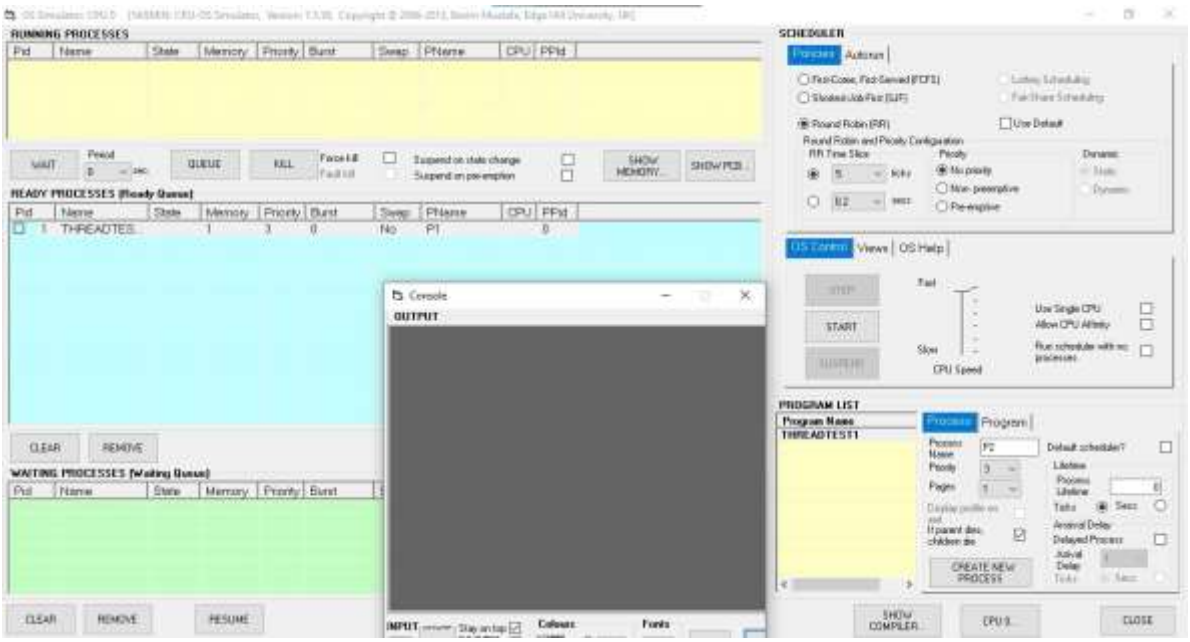
Compile the above source and load the generated code in memory.

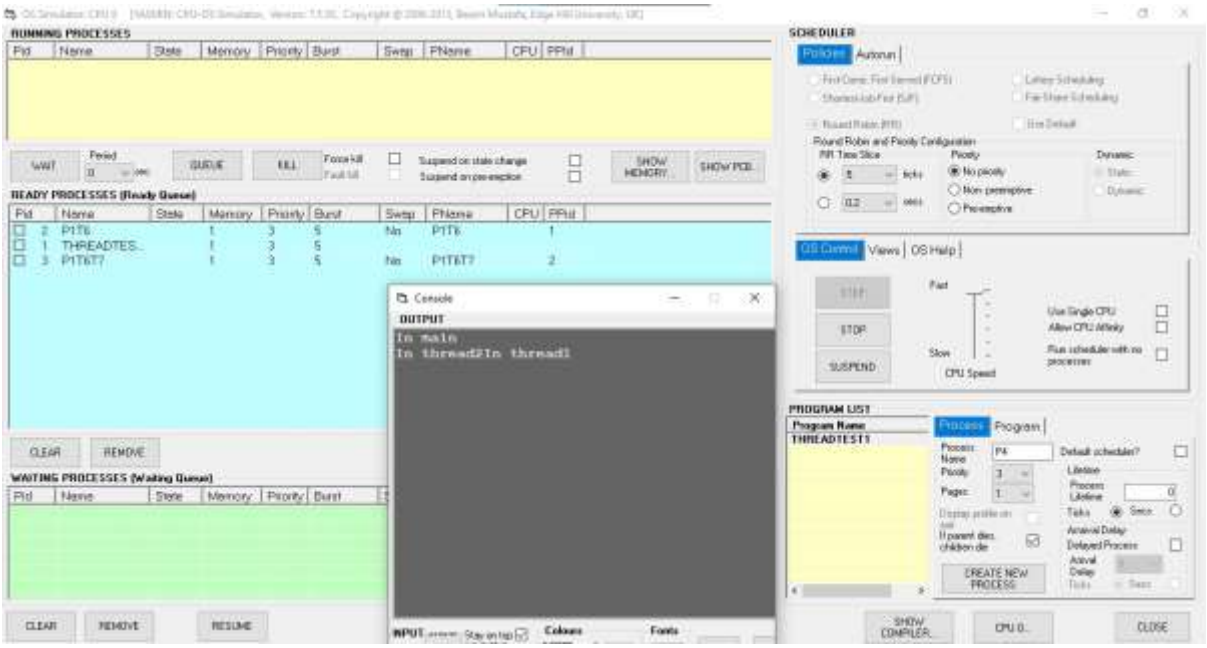
Make the console window visible by clicking on the **INPUT/OUTPUT…** button. Also make sure the console window stays on top by checking the **Stay on top** check box.

Now, go to the OS simulator window (use the **OS…** button in the CPU simulator window) and create a single process of program *ThreadTest1* in the program list view. For this use the **CREATE NEW PROCESS** button.

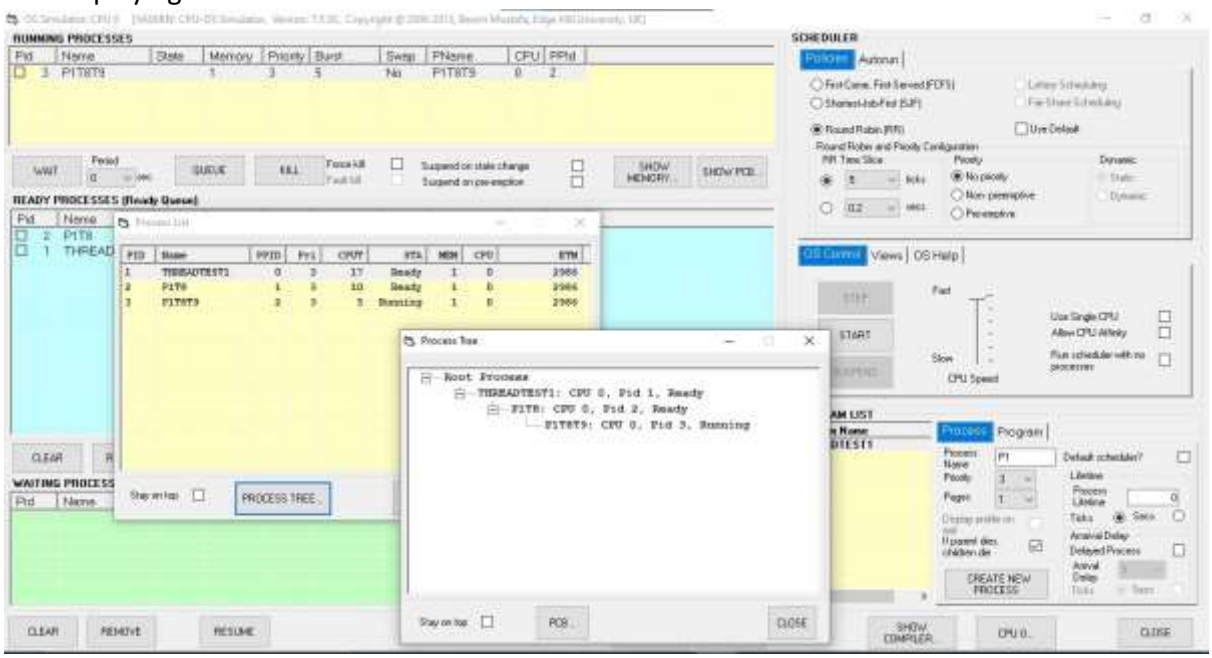
Make sure the scheduling policy selected is **Round Robin** and that the simulation speed is set at maximum.

Hit the **START** button and at the same time observe the displays on the console window. Briefly explain your observations and the no. of processes created in the box below.





Now, click on the **Views** tab and click on the **VIEW PROCESS LIST…** button. Observe the contents of the window now displaying.
In the Process List window hit the **PROCESS TREE…** button. Observe the contents of the window now displaying.



Stop the running processes by repeatedly using the **KILL** button in the OS simulator window.

Synchronization

Loading and Compiling a Program

In the compiler window, enter the following source code in the compiler source editor area (under **PROGRAM SOURCE** frame title). Make sure your program is exactly the same as the one below (best to use copy and paste for this).

```
program CriticalRegion1  var g
     integer  sub thread1 as
     thread  writeln("In
     thread1")  g = 0  for n =
     1 to 20
g = g + 1
          next
          writeln("thread1 g = ", g)
          writeln("Exiting thread1")
     end sub  sub thread2 as
     thread  writeln("In
     thread2")  g = 0
     for n = 1 to 12
     g = g + 1
          next
          writeln("thread2 g = ", g)
          writeln("Exiting thread2")
     end sub
     writeln("In main")
      call
     thread1
     call
     thread2

     wait
     writeln("Exiting main")
end
```

The above code creates a main program called *CriticalRegion1*. This program creates two threads thread1 and thread2. Each thread increments the value of the global variable **g** in two separate loops.

- Compile the above code using the **COMPILE…** button.
- Load the CPU instructions in memory using the **LOAD IN MEMORY** button**.**
- Display the console using the **INPUT/OUTPUT…** button in CPU simulator.
- On the console window check the **Stay on top** check box.

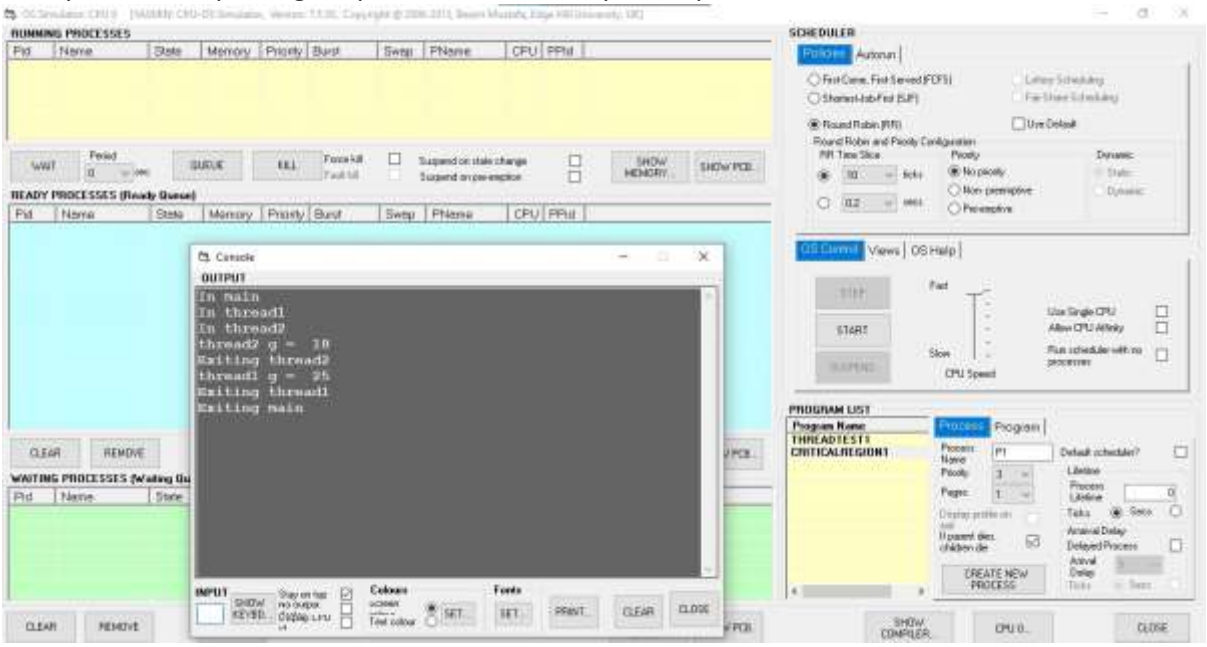
Running the above code

- Enter the OS simulator using the **OS 0…** button in CPU simulator.
- You should see an entry, titled *CriticalRegion1*, in the **PROGRAM LIST** view.
- Create an instance of this program using the **NEW PROCESS** button.
- Select **Round Robin** option in the **SCHEDULER/Policies** view.
- Select **10 ticks** from the drop-down list in **RR Time Slice** frame.

- Make sure the console window is displaying (see above).
- Move the **Speed** slider to the fastest position.
- Start the scheduler using the **START** button.

Now, follow the instructions below without any deviations:

When the program stops running, make a note of the two displayed values of **g**. Are these values what you were expecting? Explain if there are any discrepancies.

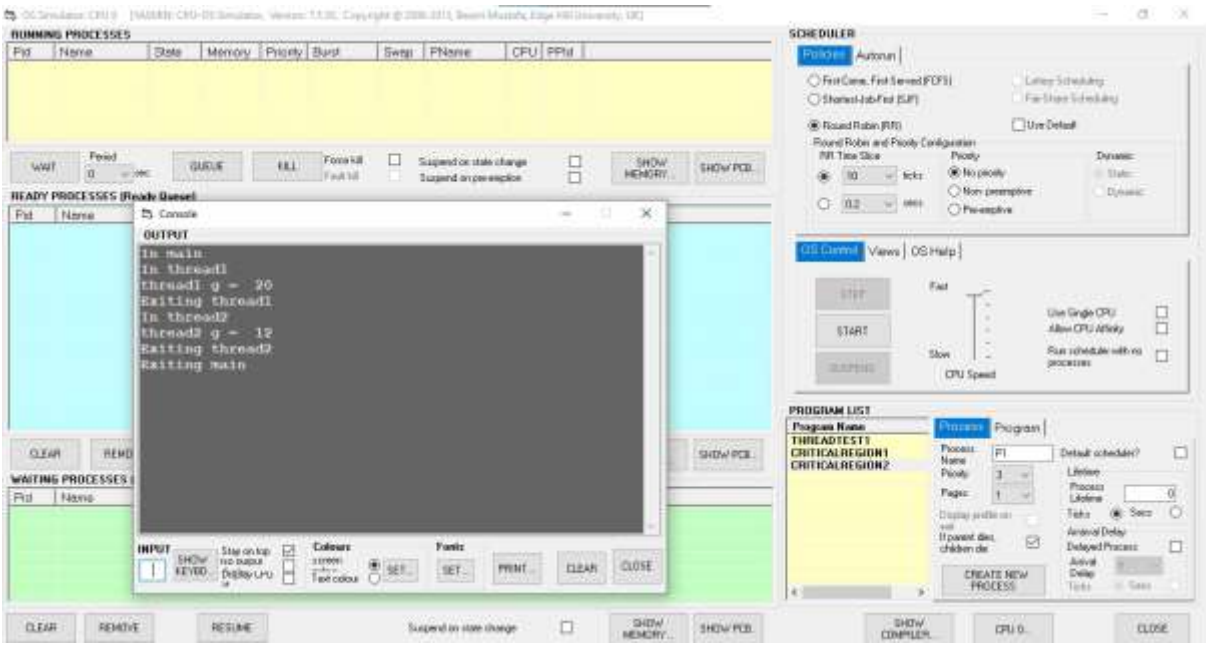


Modify this program as shown below. The changes are in bold and underlined. Rename the program *CriticalRegion2*.

```
program CriticalRegion2  var
     g integer
sub thread1 as thread synchronise
      writeln("In thread1")  g
          = 0  for n = 1 to 20
          g = g + 1
          next
          writeln("thread1 g = ", g)
          writeln("Exiting thread1")
     end sub
     sub thread2 as thread synchronise
          writeln("In thread2")  g =
          0
     for n = 1 to 12
     g = g + 1
          next
          writeln("thread2 g = ", g)
          writeln("Exiting thread2")
     end sub
```

```
writeln("In main")
 call
thread1
call
thread2

wait
writeln("Exiting main")  end
```

Compile the above program and load in memory as before. Next, run it and carefully observe how the threads behave. Make a note of the two values of variable **g**.



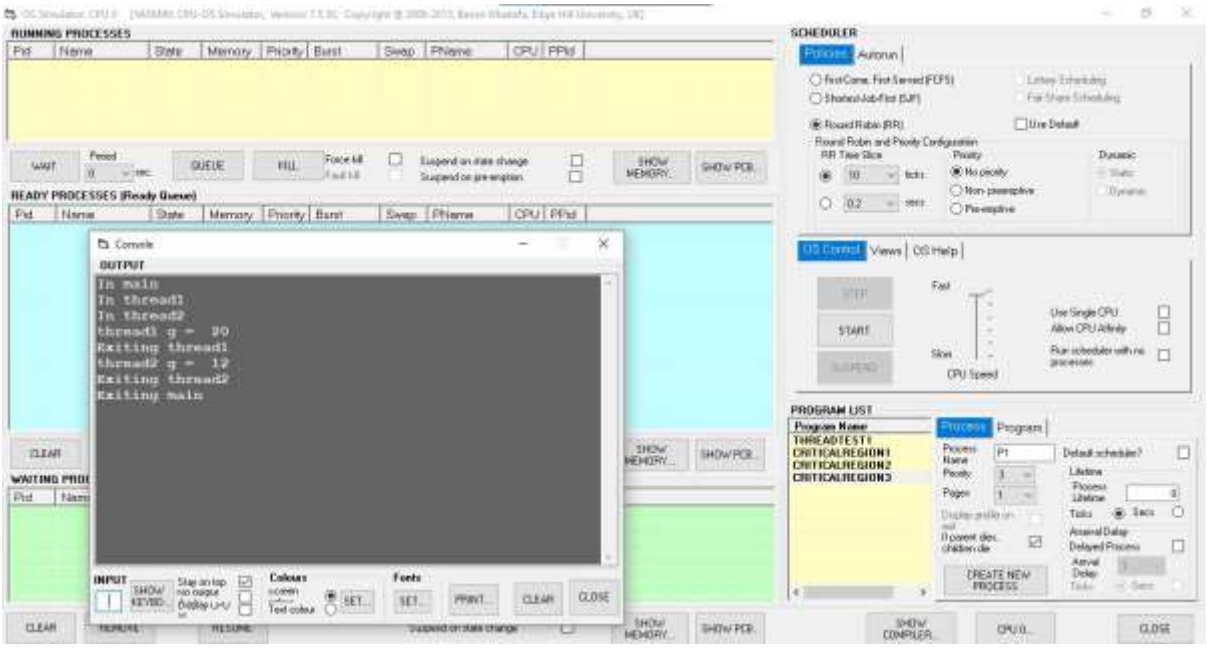
Modify this program for the second time. The new additions are in bold and underlined. Remove the two **synchronise** keywords. Rename it *CriticalRegion3*.

```
program CriticalRegion2  var g
     integer  sub thread1 as
     thread  writeln("In
     thread1")
          enter
          g = 0  for n =
          1 to 20  g = g
          + 1
          next
          writeln("thread1 g = ", g)
          leave
          writeln("Exiting thread1")
```

```
end sub  sub thread2 as
thread writeln("In
thread2")
      enter
      g = 0  for n =
      1 to 12
      g = g + 1
      next
      writeln("thread2 g = ", g)
      leave
      writeln("Exiting thread2")
end sub
writeln("In main")
 call
thread1
call
thread2

wait
writeln("Exiting main")
                                                                    end
```

NOTE: The **enter** and **leave** keyword pair protect the program code between them. This makes sure the protected code executes exclusively without sharing the CPU with any other thread.

Locate the CPU assembly instructions generated for the **enter** and **leave** keywords in the compiler's **PROGRAM CODE** view. You can do this by clicking in the source editor on any of the above keywords. Corresponding CPU instruction will be highlighted.:
Compile the above program and load in memory as before. Next, run it. Make a note of the two values of variable **g**.



Deadlock prevention and avoidance

Four processes are running. They are called **P1** to **P4**.There are also four resources available (only one instance of each). They are named **R0** to **R3**. At some point of their existence each process allocates a different resource for use and holds it for itself forever. Later each of the processes request another one of the four resources.

Use the Scenario P1 holding R0 and waiting for R1. P2 Holding R1 and waiting for R2. P3 holding R2 and waiting for R3. P4 holding R3 and waiting for R0.
Draw the resource allocation graph for a four process deadlock condition.

In the compiler window, enter the following source code in the compiler source editor area (under **PROGRAM SOURCE** frame title).

```
program DeadlockP N
resource(X, allocate)
wait(3)  resource(Y,
allocate)  for n = 1 to
20  next  end
```

2)

a. Copy the above code and paste it in three more edit windows so that you have a total of four pieces of source code. (Click on New under compiler to create new edit windows)
b. In each case change **N** in the program name to 1 to 4, e.g. DeadlockP**1,** DeadlockP**2,** etc.
c. Look at your graph you constructed in (1) above and using that information fill in the values for each of the **X**s and **Y**s in the four pieces of source code. (X is resource the process is holding and Y is the resource process is waiting for. Eg. For P1, X=0 and Y=1)
d. Compile each one of the four source code.
e. Load in memory the four pieces of code generated.
f. Now switch to the OS simulator.
g. Create a single instance of each of the programs. You can do this by double-clicking on each of the program names in the **PROGRAM LIST** frame under the **Program Name** column.
h. In the **SCHEDULER** frame select **Round Robin (RR)** scheduling policy in the **Policies** tab.
i. In OS Control tab, push the speed slider up to the fastest speed.
j. Select the **Views** tab and click on the **VIEW RESOURCES...** button.

k. Select **Stay on top** check box in the displayed window.
l. Back in the **OS Control** tab use the **START** button to start the OS scheduler and observe the changing process states for few seconds.
m. Have you got a deadlock condition same as you constructed in (1) above? If you haven't then check and if necessary re-do above. Do not proceed to (n) or (3) below until you get a deadlock condition.
n. If you have a deadlock condition then click on the **SHOW DEADLOCKED PROCESSES…** button in the **System Resources** window. Does the highlighted resource allocation graph look like yours?

Now that you created a deadlock condition let us try two methods of getting out of this condition:

a. In the **System Resources** window, there should be four resource shapes that are in red colour indicating they are both allocated to one process and requested by another.
b. Select one of these resources and click on the **Release** button next to it.
c. Observe what is happening to the processes in the OS Simulator window.
d. Is the deadlock situation resolved? Explain briefly why this helped resolve the deadlock.

e. Re-create the same deadlock condition (steps in 2 above should help).

f. Once the deadlock condition is obtained again do the following: In the OS Simulator window, select a process in the waiting queue in the **WAITING PROCESSES** frame.

g. Click on the REMOVE button and observe the processes.

h. Has this managed to resolve the deadlock? Explain briefly why this helped resolve the deadlock.

This part of the exercises was about two methods of **recovering** from a deadlock condition **after** it happens.

We now look at two methods of **preventing** a deadlock condition **before** it happens.

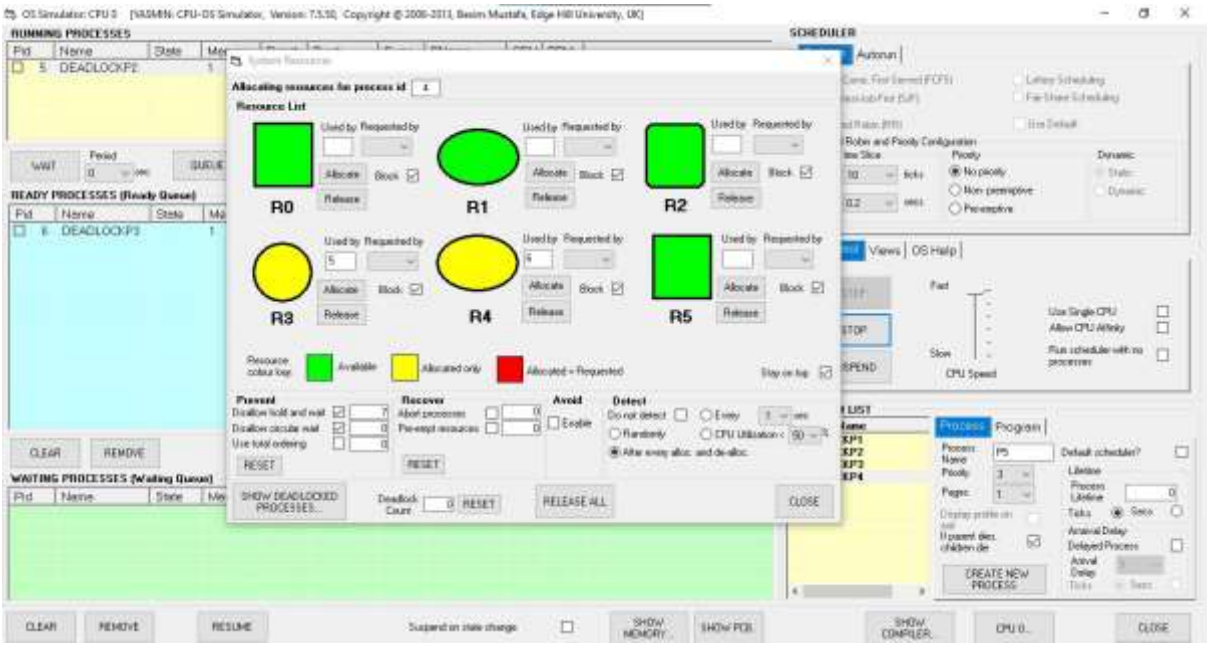
a. In the **System Resources** window select the **Disallow hold and wait** check box in the **Prevent** frame.

b. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues.

c. Next, uncheck the **Disallow hold and wait** check box and check the **Disallow circular wait** check box.

d. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues.

We are now going to try a third method of preventing deadlocking before it happens. It is called "total ordering" method. Here the resources are allocated in increasing resource id numbers only. So, for example, resource R3 must be allocated after resources R0 to R2 and resource R1 cannot be allocated after resource R2 is allocated. Looking at your resource allocation graph can you see how this ordering can prevent a deadlock? Comment.



a. In the **System Resources** window select the **Use total ordering** check box in the **Prevent** frame. The other options should be unchecked.

b. Try to re-create the same deadlock condition as before. Have you been successful? What happened? Click on the **SHOW DEADLOCKED PROCESSES...** button and observe the displayed information in the text window for potential clues. What happened? Comment.


RUNNING PROCESSES
READY PROCESSES (Ready Queue)
WAITING PROCESSES (Waiting Queue)
SCHEDULER
R0
R1
R2
R3
R4
R5
SHOW DEADLOCKED PROCESSES...
RELEASE ALL
CLOSE

Operating Systems

Experiment 9

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

To implement page replacement algorithms.

Theory -

Page Replacement Algorithms

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).

Page Replacement Algorithms:

1. First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

2. Least Recently Used (LRU)

In this algorithm, page will be replaced which is least recently used.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory.

Code –

```
#include <iostream>

using namespace std;
int fSize, n;

void answer(int *data, int **ans)
{
    int hit = 0;
    cout << "Answer:" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << data[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "_"
             << " ";
    }
    cout << endl;
    for (int j = 0; j < fSize; j++)
    {
        for (int i = 0; i < n; i++)
        {
            cout << ans[i][j] << " ";
        }
        cout << endl;
    }
    for (int i = 0; i < n; i++)
    {
        if (ans[i][fSize] == -1)
        {
            cout << "F ";
        }
        else
        {
            hit++;
            cout << "H ";
        }
    }
    cout << endl
         << "Number of Page Hits: " << hit;
    cout << endl
         << "Number of Page Faults: " << (n - hit);
}
void fifo(int *data, int **ans)
```

```
{
    int in = 0, temp = 0, counter = fSize;
    for (int i = 0; i < n; i++)
    {
        if (i < counter)
        {
            for (int j = 0; j < fSize; j++)
            {
                if (j < (i + fSize - counter))
                {
                    ans[i][j] = ans[i - 1][j];
                    if (data[i] == ans[i - 1][j])
                    {
                        temp = 1;
                    }
                }
                else
                    ans[i][j] = 0;
            }
            if (temp == 1)
            {
                ans[i][fSize] = 0;
                counter++;
            }
            else
            {
                ans[i][(i + fSize - counter)] = data[i];
                ans[i][fSize] = -1;
            }
            temp = 0;
        }
        else
        {
            for (int j = 0; j < fSize; j++)
            {
                ans[i][j] = ans[i - 1][j];
                if (data[i] == ans[i - 1][j])
                {
                    temp = 1;
                }
            }
            if (temp == 1)
            {
                ans[i][fSize] = 0;
            }
            else
            {
                ans[i][in] = data[i];
                in = (in + 1) % fSize;
```

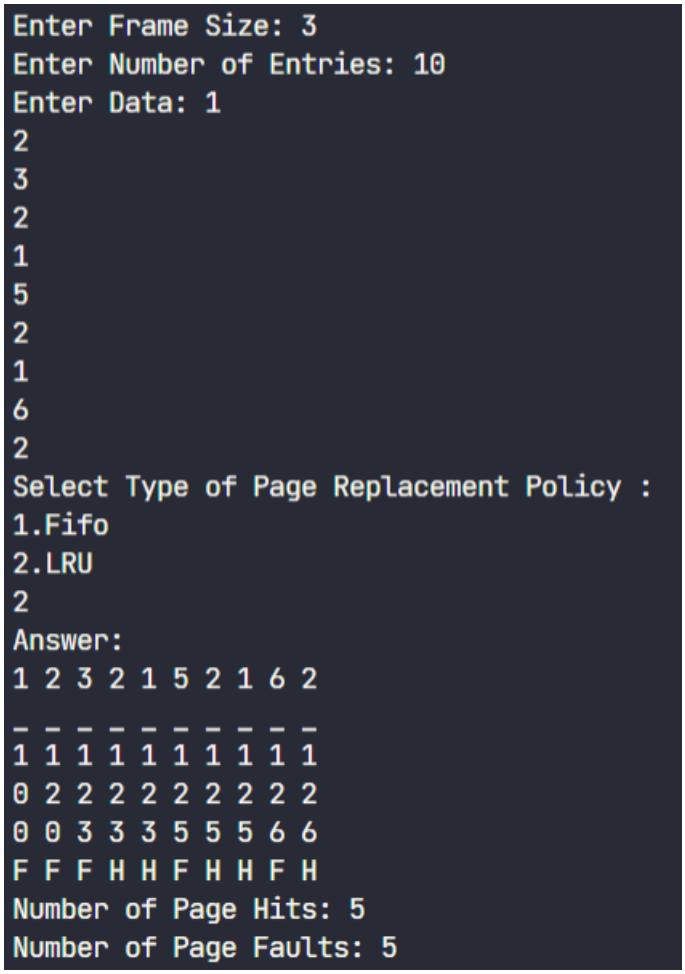
```
                ans[i][fSize] = -1;
            }
        }
        temp = 0;
    }
    answer(data, ans);
}
void lru(int *data, int **ans)
{
    int in = 0, temp = 0, counter = fSize, key = 0;
    int *q = new int[fSize];
    for (int i = 0; i < n; i++)
    {
        if (i < counter)
        {
            for (int j = 0; j < fSize; j++)
            {
                if (j < (i + fSize - counter))
                {
                    ans[i][j] = ans[i - 1][j];
                    if (data[i] == ans[i - 1][j])
                    {
                        key = j;
                        temp = 1;
                    }
                }
                else
                    ans[i][j] = 0;
            }
            if (temp == 1)
            {
                ans[i][fSize] = 0;
                q[key] = i;
                counter++;
            }
            else
            {
                ans[i][(i + fSize - counter)] = data[i];
                ans[i][fSize] = -1;
                q[(i + fSize - counter)] = i;
            }
            temp = 0;
        }
        else
        {
            for (int j = 0; j < fSize; j++)
            {
                ans[i][j] = ans[i - 1][j];
                if (data[i] == ans[i - 1][j])
```

```
                {
                    temp = 1;
                    q[j] = i;
                }
            }
            if (temp == 1)
            {
                ans[i][fSize] = 0;
            }
            else
            {
                in = 0;
                for (int j = 0; j < fSize; j++)
                {
                    if (q[in] > q[j])
                    {
                        in = j;
                    }
                }
                ans[i][in] = data[i];
                q[in] = i;
                ans[i][fSize] = -1;
            }
        }
        temp = 0;
    }
    answer(data, ans);
}
int main()
{
    int choice;
    cout << "Enter Frame Size: ";
    cin >> fSize;
    cout << "Enter Number of Entries: ";
    cin >> n;
    int *data = new int[n];
    cout << "Enter Data: ";
    for (int i = 0; i < n; i++)
    {
        cin >> data[i];
    }
    int **ans = new int *[n];
    for (int i = 0; i < n; i++)
    {
        ans[i] = new int[fSize + 1];
    }
    cout << "Select Type of Page Replacement Policy :" << endl;
    cout << "1.Fifo\n2.LRU\n";
    cin >> choice;
```

```
    switch (choice)
    {
    case 1:
        fifo(data, ans);
        break;
    case 2:
        lru(data, ans);
        break;
    default:
        break;
    }
    return 0;
}
```

Output -

```
Enter Frame Size: 3
Enter Number of Entries: 10
Enter Data: 1
2
3
2
1
5
2
1
6
2
Select Type of Page Replacement Policy :
1.Fifo
2.LRU
1
Answer:
1 2 3 2 1 5 2 1 6 2

- - - - - - - - - -
1 1 1 1 1 5 5 5 5 2
0 2 2 2 2 2 2 1 1 1
0 0 3 3 3 3 3 3 6 6
F F F H H F H F F F
Number of Page Hits: 3
Number of Page Faults: 7
```



Conclusion

Page replacement algorithms 1.) FIFO and 2.) LRU are successfully executed.

Operating Systems

Experiment 10

Name: Kartik Jolapara **SAP ID: 60004200107**

Div.: B1 **Branch: Computer Engineering**

Aim -

To implement Disk scheduling algorithm SSTF, SCAN.

Theory -

Disk Scheduling Algorithms are needed because a process can make multiple I/O requests and multiple processes run at the same time. The requests made by a process may be located at different sectors on different tracks. Due to this, the seek time may increase more. These algorithms help in minimizing the seek time by ordering the requests made by the processes.

Shortest Seek Time First (SSTF): In this algorithm, the shortest seek time is checked from the current position and those requests which have the shortest seek time is served first. In simple words, the closest request from the disk arm is served first.

SCAN: In this algorithm, the disk arm moves in a particular direction till the end and serves all the requests in its path, then it returns to the opposite direction and moves till the last request is found in that direction and serves all of them.

Code –

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

void sort(int disk[], int n)
{
```

```
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            int temp;
            if (disk[j] > disk[j + 1])
            {
                temp = disk[j];
                disk[j] = disk[j + 1];
                disk[j + 1] = temp;
            }
        }
    }
}
void sstf(int disk[], int head, int n)
{
    int sequence[n];
    for (int k = 0; k < n; k++)
    {
        sequence[k] = 0;
    }
    int track = 0;
    int new_postion = head;
    for (int j = 0; j < n; j++)
    {
        int min = 999;
        int idx = 0;
        for (int i = 0; i < n; i++)
        {
            int temp = 0;
            if (disk[i] > new_postion)
            {
                temp = disk[i] - new_postion;
            }
            else
            {
                temp = new_postion - disk[i];
            }
            if (temp < min && !sequence[i])
            {
                min = temp;
                idx = i;
            }
        }
        cout << "Request procced " << disk[idx] << endl;
        if (disk[idx] > new_postion)
        {
            track += disk[idx] - new_postion;
        }
```

```
        else
        {
            track += new_postion - disk[idx];
        }
        new_postion = disk[idx];
        sequence[idx] = 1;
    }
    cout << "Track movement is " << track << endl;
}
void scan(int disk[], int head, int n)
{
    int sequence[n];
    int pos = head;
    for (int i = 0; i < n; i++)
    {
        sequence[i] = 0;
    }
    int track = 0;
    sort(disk, n);
    for (int j = 0; j < n; j++)
    {
        if (disk[j] > 50 && !sequence[j] && pos != 199)
        {
            cout << "Request processed" << disk[j] << endl;
            sequence[j] = 1;
        }
    }
    if (pos != 199)
    {
        track = 199 - 50;
        pos = 199;
    }
    int new_postion = 199;
    for (int k = n - 1; k >= 0; k--)
    {
        int sum = 0;
        if (disk[k] < 199 && !sequence[k])
        {
            cout << "Request processeds" << disk[k] << endl;
            sequence[k] = 1;
            sum = new_postion - disk[k];
            track += sum;
            new_postion = disk[k];
        }
    }
    cout << "Track moment " << track << endl;
}
int main()
{
```

```
    int n;
    cout << "No of request" << endl;
    cin >> n;
    int disk[n];
    cout << "Enter postion of head" << endl;
    int head;
    cin >> head;
    cout << "Enter request" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> disk[i];
    }
    cout << "Enter 1 for SSTF and 2 for SCAN" << endl;
    int o;
    cin >> o;
    if (o == 1)
        sstf(disk, head, n);
    if (o == 2)
        scan(disk, head, n);
}
```

Output –

```
No of request
10
Enter postion of head
50
Enter request
23 105 188 198 110 89 23 45 89 90
Enter 1 for SSTF and 2 for SCAN
1
Request procced 45
Request procced 23
Request procced 23
Request procced 89
Request procced 89
Request procced 90
Request procced 105
Request procced 110
Request procced 188
Request procced 198
Track movement is 202
```

```
No of request
10
Enter postion of head
50
Enter request
23 105 188 198 110 89 23 45 89 90
Enter 1 for SSTF and 2 for SCAN

2
Request processed89
Request processed89
Request processed90
Request processed105
Request processed110
Request processed188
Request processed198
Request processeds45
Request processeds23
Request processeds23
Track moment 325
```

Conclusion

Disk Scheduling Algorithms like SSTF and SCAN are successfully executed.

Name: Kaustik Jolapalla
SAP ID: 60004200107
Div: B

DATE
PAGE NO.

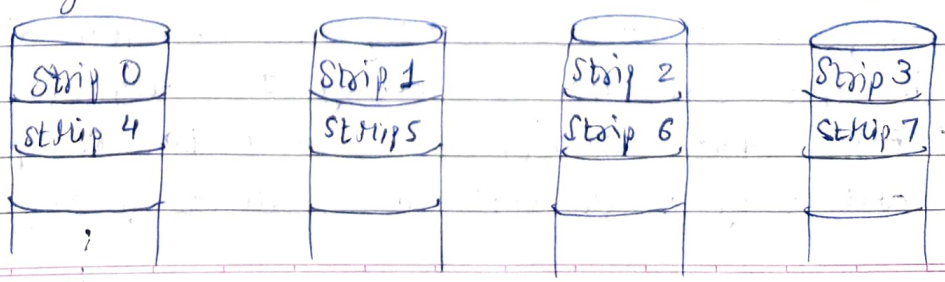
OS
Assignment 1

Q. 1) Describe RAID disk management. Provide detailed desc of the all the levels of RAID.

→
- RAID is a set of physical disk drivers viewed by the operating system as a signal logical drive
- RAID is the acryonm for Redundant Array of Independent Disks or Redudant Array of Inexpensive Disks.
- Here the data are distributed across the physical drivers of an array in a scheme known as stripping
- Redudent disk capacity is used to store parity information which guarantees data recoverability in the case of an disk failure
- RAID consists of seven levels.

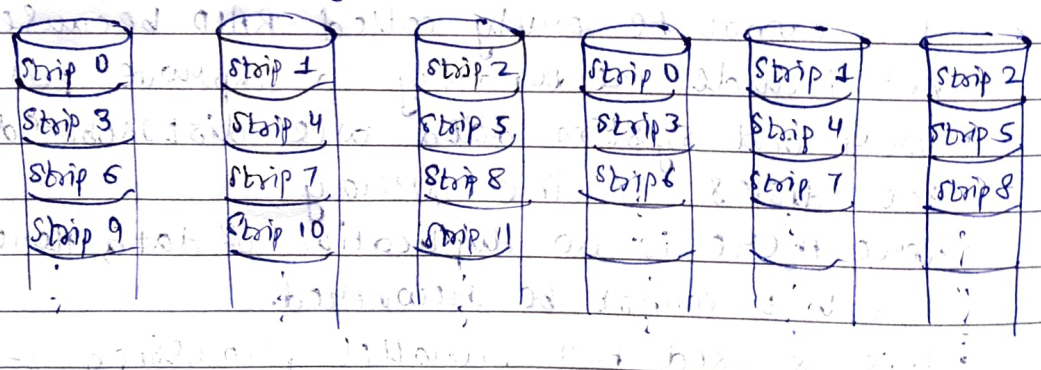
RAD level 0

- This cannot be truly called RAID because it does not include redundancy to improve performance
- User and system data are distributed across all the disks in the array
- Here there is no duplication of data here a block once lost cannot be recovered.
- This is used for parallel processing of data which increases the speed of operation.
- The logical disk is divided into strips.



⊛RAID level 1

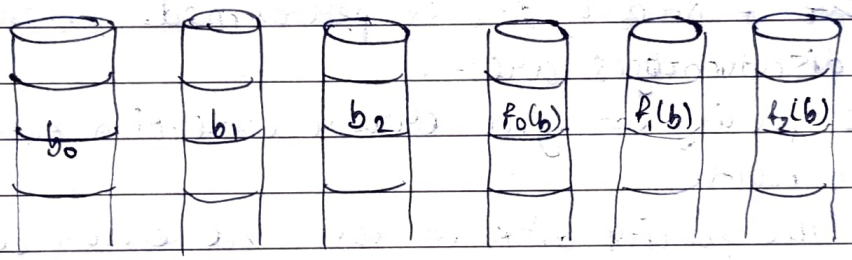
- Also known as mirrored disks as redundancy is achieved by the simple expedient of duplicating all the data
- Here if one drive fails the data may still be accessed from the second drive.
- Here there is no 'write penalty' ie. a write request requires that both corresponding strips be updated but this can be done in parallel
- The principal disadvantage is cost as it requires twice the disk space of the logical device it supports.
- RAID 1 configuration is limited to device that stores system software and data and other highly critical files. In these cases, RAID 1 provides real time backup of all data. So that is the event of a disk failure, all of the critical data is still immediately available



⊛RAID Level 2

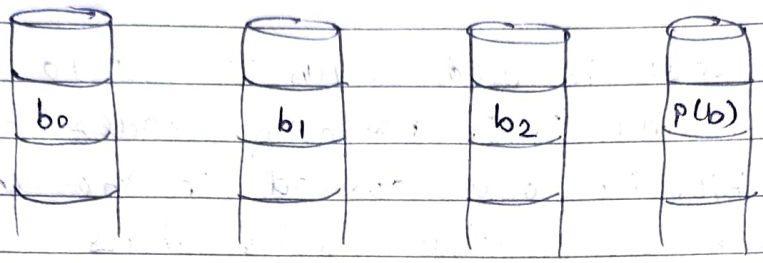
- It makes use of parallel ~~to~~ access technique. In a parallel access array, all number disks participate in the execution of every I/O request.
- Here, data stripping is used to strip. The strips are very small often of small as a single byte/word.

- A hamming code is used, which is able to correct single bit error. These hamming codes are calculated bitwise and stored in the corresponding bit positions on multiple parity disks
- RAID level 2 requires less disks than RAID level 1.
- ~~RAID level~~ If there is a single bit error, the controller can recognize and correct the error instantly so that read access time is not slow. But multiple errors cannot be checked and corrected.
- RAID level 2 will only be effective choice in an environment in which many disk error occour.



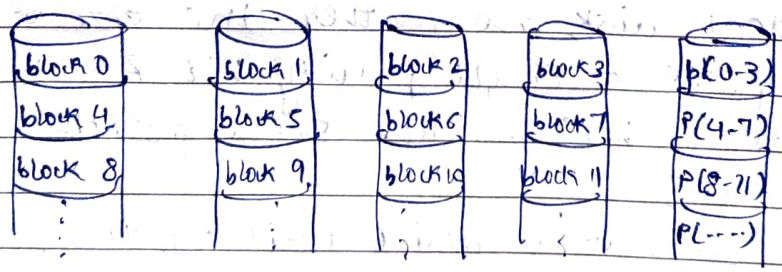
⊛ RAID level 3

- It is similar to RAID level 2
- Here, the difference is that it requires only a single redundant disk, no matter the size of disk array
- It computes a single parity bit for the set of individual bits in the same position on all the data disks.
- In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced the missing data can be restored and operations are resumed
- It can achieve very high data transfer rates.



⊛ RAID level 4

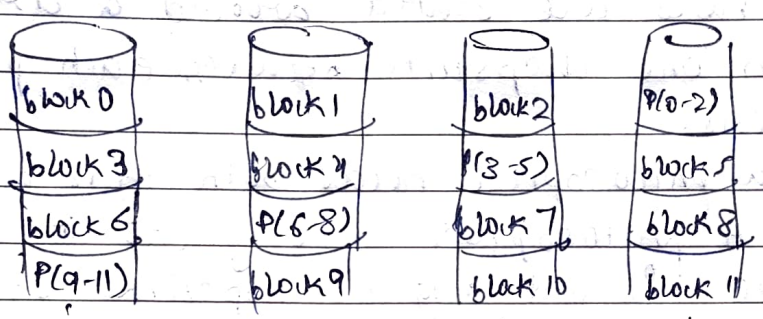
- Uses independent access technique. In independent access array each number disk operates seperately.
- Here the data is striped in blocks
- A parity strip is calculated and this parity is stored in the corresponding strip on the parity disk.
- It involves a write penalty when an I/O write request of small size is performed.
- The disadvantages are:-
  1. The parity can give error correction for only one block
  2. If the parity disk is lost the error correction cannot be done
  3. Since parity is given to one disk only, this disk is repeatedly used. This results in an overload and it may slow down.



⊛ RAID Level 5

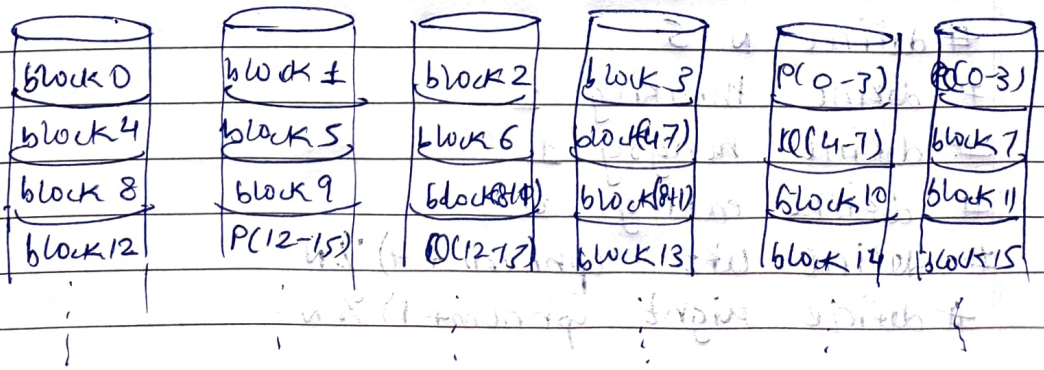
- Similar to RAID level 4
- Here the parity strips are distributed all across the disks.

- The allocation is done by round-robin scheme
- This helps in avoiding the potential I/O bottleneck of single parity disk as seen in RAID level 4.
- It has the characteristics that the loss of any one disk does not result in data loss.



RAID level 6

- Here two different parity calculations are carried out and stored in seperate blocks on different disks.
- Thus, if user data requires N disks RAID level 6 will contain N+2 disks
- The main advantage is that it provides extremely high data availability
- It includes a write penalty because each write affects two parity blocks.
- Presence of two parity strips helps in calculating for two errors.



2) Explain dining philosopher's problem and its solution using semaphores.

→ DINING PHILOSOPHER'S PROBLEM

- The dining philosophers problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers.
- To eat a philosopher needs both their and left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available
- In case if both immediate left & right chopstick of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.
- The dining philosophers problem is a classical problem of synchronization and demonstrates a large class of concurrency control problems.

Solution using semaphores

```
#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[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]);
    }
}
void take_forks(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 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 *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_creat (&thread_id[i], NULL,
                   philosophers, &phil[i]);
    printf("Philosopher %d is thinking\n", i+1);
}
for (i=0; i<N; i++)
    pthread_join(thread_id[i], NULL);
}
```

Name: Kartik Jolapara
SAP ID: 60004200107
Div: B

OS Assignment 2

Q.1) a) XV6

→ ① XV6 is a teaching operating system developed for MIT course

② XV6 provides the basic interfaces introduced by Ken Thompson and Dennis Richie's Unix OS, also mimicking its internal design.

③ Because it is based on UNIX it's also has a Kernel where services can be accessed by the means of a system call It provides system calls the ones provided by the UNIX.

④ It uses X86 instruction set. The CPU is time shared among processes; it transparently switches the available CPUs among the set of ready processes.

⑤ The XV6 kernel uses file description an integer representing a Kernal managed by object that a process may read or write from as an index in per process table

⑥ In XV6 file system provides data files which contain uninterpreted byte arrays, which contain named references to data files and other directions

⑦ It also supports special node files to that act as interface to I/O device

⑧ XV6 implements a monolithic kernel

Q.1) b) RTOS

→ ① Real time OS are used in environments where a large number of events, mostly external to the computer systems must be prepared, accepted and processed in a short time or within certain deadlines.

② Such application are individual control, telephone switching equipment, flight control & real time simulations
③ This system is extremely time bound and deadline must be met
④ The scheduler employs some combination of priority based scheduler to ensure deadlines of important interrupts are met.
⑤ It is very lightweight to improve the performance by cutting down on overheads.
⑥ It is extremely error resilence and employs a watch dog.

Q.1) c) Mobile operating system

→ ① A mobile operating system runs on smart/dumb phones tablets like devices.
② while it performs jobs some or atleast similar to that of desktop OS. It incorporates various other elements and different requirements.
③ At basic level it incorporates a specialized UI, meant for smaller screens and touch interfaces.
④ Because these devices have battery they are very efficient and as such avoid too much load on CPUs whenever possible
⑤ Today majority of smartphones use ARM chips and as such many mobile OSs specifically specifically make use of optimizations made possible by ARM architecture

⑥ A mobile also contains radio receiver & transmitter module to facilitate Radio network connection. The network stack is optimized to correctly make use of this module while ensuring both high performance & optimal battery usage

⑦ The security on these OSs is often very tight and do not follow for admin previleges in most OS unless the device is loaded with a different bootloader and in turn a new OS.

Eg

(i) Android

It is the most common as well as well known mobile based OS based on Linux 2.6 to provide services such as security, memory management, process management, device model etc

(ii) iOS

It is a closed source OS by Apple developed for iPhones & iPods. It is based Darwin, which itself is an open source UNIX like operating system

(iii) KaiOS

It is developed by ~~for~~ Mozilla for dumb keyed phones.