OS Lab Report
Assignment 2

Name : ADNAN KHURSHID
Roll No : 002010501025
Group : A1
Class : BCSE III
Semester : 5

1. Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and appropriate scheduling algorithm to be chosen by the scheduler.
Format of the profile:
<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) .... >-1
(Each information is separated by blank space and each job profile ends with -1. Lesser priority
number denotes higher priority process with priority number 1 being the process with highest
priority.)
Example: 2 3 0 100 2 200 3 25 -1 1 1 4 60 10 ..... -1 etc.
Testing:
a. Create job profiles for 20 jobs and use three different scheduling algorithms (FCFS, preemptive Priority and Round Robin (time slice:20)).
b. Compare the average waiting time, turnaround time of each process for the different scheduling algorithms.

CODE :

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <unordered_map>
#include <utility>
#include <fstream>
```

```
#include<limits.h>
#include <algorithm>
using namespace std;

// class to store details of a job, including their execution profiles
class Job {
private:
    int jobId;                  // job id
    int priority;               // priority (lesser value, higher
priority)
    int arrivalTime;            // arrival time
    vector<int> cpuBursts;      // patches of cpu burst times, i.e. cpu
burst(1), cpu burst(2), ...
    vector<int> ioBursts;       // patches of i/o burst times, i.e. i/o
burst(1), i/o burst(2), ...
    int cntCPU;                 // total count of cpu bursts to occur
    int cntIO;                  // total count of i/o bursts to occur
    int nextCPU;                // index of next cpu burst to be
performed
    int nextIO;                 // index of next i/o burst to be
performed
    int nextArrivalTime;        // next arrival time for ready queue
    int totalTimeReqd;          // sum of all bursts
    bool preempt;               // to track flag for preemptive
algorithms
    // the execution profile of the job is like: cpu burst(1), i/o
burst(1), cpu burst(2), i/o burst(2), ...
public:
    Job() {                     // default constructor
        jobId = priority = arrivalTime = cntCPU = cntIO = nextCPU =
nextIO = nextArrivalTime -1;
    }
    Job(vector<int> execProfile) {  // paramaterized constructor with
execution profile as the argument
        int sz = execProfile.size();
        // for (int i = 0; i < sz; ++i)
        //     cout << execProfile[i] << " ";
        // cout << "\n";
        jobId = execProfile[0];
        priority = execProfile[1];
        nextArrivalTime = arrivalTime = execProfile[2];
        cntCPU = cntIO = 0;
        totalTimeReqd = 0;
```

```
        preempt = false;
        for (int i = 3; i < sz; ++i) {
            if (i & 1) {        // for cpu bursts
                cpuBursts.push_back(execProfile[i]);
                ++cntCPU;
            } else {            // for i/o bursts
                ioBursts.push_back(execProfile[i]);
                ++cntIO;
            }
            totalTimeReqd += execProfile[i];
        }
        nextCPU = nextIO = 0;
        cout << "Job extracted with ID: " << jobId << "\n"; // debug
statement
    }
    // getter functions
    int getJobId()            {   return jobId;           }
    int getPriority()         {   return priority;        }
    int getArrivalTime()      {   return arrivalTime;     }
    int getCntCPU()           {   return cntCPU;          }
    int getCntIO()            {   return cntIO;           }
    int getNextCPU()          {   return nextCPU;         }
    int getNextIO()           {   return nextIO;          }
    int getCurrCPUTime()      {   return cpuBursts[nextCPU];}
    int getCurrIOTime()       {   return ioBursts[nextIO];}
    int getNextArrivalTime(){   return nextArrivalTime; }
    int getTotalTime()        {   return totalTimeReqd;   }
    int getPreempt()          {   return preempt;         }
    // setter functions
    void incNextCPU()         {   if (nextCPU < cntCPU)   ++nextCPU;  }
    void incNextIO()          {   if (nextIO < cntIO)     ++nextIO;   }
    void setPreempt()         {   preempt = true;                     }
    void unsetPreempt()       {   preempt = false;                    }
    void updateCPUTime(int dur) {
        cpuBursts[nextCPU] -= dur;
    }
    void updateArrival(int dur) {
        if (cpuBursts[nextCPU] == dur)
            nextArrivalTime = (cpuBursts[nextIO] + ioBursts[nextIO]);
        cpuBursts[nextCPU] -= dur;
    }
    // checker functions
    bool cpuLeft()            {   return nextCPU < cntCPU;    }
```

```
    bool ioLeft()           {   return nextIO < cntIO;      }

};

// comparator class for ordering jobs on the basis of arrival time
class JobComparatorFCFS {
public:
    bool operator()(Job& a, Job& b) {
        return a.getArrivalTime() < b.getArrivalTime();
    }
};

// comparator class for ordering jobs on the basis of priority
class JobComparatorPriority {
public:
    bool operator()(Job& a, Job& b) {
        return a.getPriority() > b.getPriority();
    }
};

// abstract class for Job Scheduling algorithms
class JobScheduler {
protected:
    vector<Job> jobs;
    int jobsLeft;
    int totalWaitingTime;
    float avgWaitingTime;
    int totalTurnaroundTime;
    float avgTurnaroundTime;
public:
    // to parse the file and create vector of job profiles
    JobScheduler(string filename) {
        ifstream fin;
        fin.open(filename, ios::in);
        int num;
        totalTurnaroundTime = totalWaitingTime = jobsLeft = 0;
        while (fin >> num) {
            if (num == -1) {
                break;
            } else {    // new job starting
                vector<int> v (1, num);
                fin >> num;
                if (num != -1)
```

```
                    v.push_back(num);
                while (fin >> num) {
                    if (num == -1) {
                        break;
                    } else {
                        v.push_back(num);
                    }
                }
                Job J(v);
                jobs.push_back(J);
                ++jobsLeft;
            }
        }
    }
    // pure virtual function to schedule processes following the
scheduling algorithms
    virtual void schedule() = 0;
    // to show results like average waiting and turnaround time
    void showAnalysis() {
        avgWaitingTime = totalWaitingTime * 1.0 / jobs.size();
        avgTurnaroundTime = totalTurnaroundTime * 1.0 / jobs.size();
        cout << "\nAverage Turnaround Time = " << avgTurnaroundTime <<
"\n";
        cout << "Average Waiting Time = " << avgWaitingTime << "\n";
    }
};

// class for fcfs scheduling, inherited from JobScheduler class
class FCFS_Scheduler: public JobScheduler {
    queue<Job> ready_queue; // ready queue for CPU
    unordered_map<int, vector<Job> > block_queue; // block queue for
i/o operations
    vector<pair<int, int> > ganttChart;  // to store the schedule
public:
    // sort the jobs based on arrival time
    FCFS_Scheduler(string filename): JobScheduler(filename) {
        sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
    }

    virtual void schedule() {
        int sz = jobs.size(), index = 0;
        for (int i = 0; i < sz; ++i) {  // push all the processes
initially into the block queue
```

```
            block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
            totalTurnaroundTime += jobs[i].getTotalTime();
            // cout << jobs[i].getJobId() << " -> " <<
jobs[i].getArrivalTime() << "\n";
        }
        int timeline = 0;
        bool cpuEmpty = true;
        Job currentJob; int nextTerminate = INT_MAX;
        while (jobsLeft > 0) {
            // process coming from block queue
            for (Job j : block_queue[timeline]) {
                ready_queue.push(j);
            }
            // if the current time is the end of cpu burst of a job
            if (timeline == nextTerminate) {
                // free the cpu
                cpuEmpty = true;
                // if the job terminates, reduce remaining job count
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
            }

            // if the cpu is free
            if (cpuEmpty) {
                // and there are jobs on the ready queue
                if (!ready_queue.empty()) {
                    // pick the first job from the ready queue
                    currentJob = ready_queue.front();
                    ready_queue.pop();
                    // compute the time of end  of its cpu burst
                    nextTerminate = timeline +
currentJob.getCurrCPUTime();
                    // store the profile for the current job in gantt
chart

ganttChart.push_back(make_pair(currentJob.getJobId(), timeline));
                    // occupy the cpu
                    cpuEmpty = false;
                    // update its next arrival time

currentJob.updateArrival(currentJob.getCurrCPUTime());
                    // update its cpu index
                    currentJob.incNextCPU();
```

```
                    // if io is left
                    if (currentJob.ioLeft()) {
                        // resume io
                        currentJob.incNextIO();
                        // add the job to the appropriate index of the
waiting queue
                        if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(current
Job);
                        }
                    }
                }
            }
            // cout << timeline << " " << ready_queue.size() << "\n";
            totalTurnaroundTime += ready_queue.size();
            totalWaitingTime += ready_queue.size();
            ++timeline;
        }
        cout << "\nFCFS Gantt Chart --> \n";
        for (pair<int, int> t : ganttChart) {
            cout << "{Job " << t.first << " @ Time " << t.second << "}
";
        }
    }
    void showAnalysis() {
        cout << "\n\nFCFS Scheduling Algorithm Statistics: \n";
        JobScheduler::showAnalysis();
    }
};

// class for round robin scheduling, inherited from JobScheduler class
class RoundRobin_Scheduler: public JobScheduler {
    queue<Job> ready_queue; // ready queue for CPU
    unordered_map<int, vector<Job> > block_queue; // block queue for
i/o operations
    vector<pair<int, int> > ganttChart;  // to store the schedule
    int timeSlice;
public:
    // sort the jobs based on arrival time
    RoundRobin_Scheduler(string filename): JobScheduler(filename) {
        sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
        timeSlice = 25;
```

```
    }
    virtual void schedule() {
        int sz = jobs.size(), index = 0;
        for (int i = 0; i < sz; ++i) {  // push all the processes
initially into the block queue
            block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
            totalTurnaroundTime += jobs[i].getTotalTime();
            // cout << jobs[i].getJobId() << " -> " <<
jobs[i].getArrivalTime() << "\n";
        }
        int timeline = 0;
        bool cpuEmpty = true;
        Job currentJob; int nextTerminate = INT_MAX;
        while (jobsLeft > 0) {
            // process coming from block queue for the given time
            for (Job j : block_queue[timeline]) {
                ready_queue.push(j);
            }
            // if the current time is the end of cpu burst of a job
            if (timeline == nextTerminate) {
                // free the cpu
                cpuEmpty = true;
                // if the job terminates, reduce remaining job count
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
                // if it is end of timeslice, remove it from running
state and add to ready queue
                else if (currentJob.getPreempt() == true &&
currentJob.getCurrCPUTime() > 0) {
                    currentJob.unsetPreempt();
                    ready_queue.push(currentJob);
                }
            }
            // if the cpu is free
            if (cpuEmpty) {
                // and there are jobs on the ready queue
                if (!ready_queue.empty()) {
                    // pick the first job from the ready queue
                    currentJob = ready_queue.front();
                    ready_queue.pop();
                    // compute the time of end  of its cpu burst

ganttChart.push_back(make_pair(currentJob.getJobId(), timeline));
```

```
                    // occupy the cpu
                    cpuEmpty = false;
                    // get the next terminating point
                    int val = min(currentJob.getCurrCPUTime(),
timeSlice);
                    nextTerminate = timeline + val;
                    // update its next arrival time
                    currentJob.updateArrival(val);
                    // if the current cpu burst is 0, increment the cpu
index
                    if (currentJob.getCurrCPUTime() == 0) {
                        currentJob.incNextCPU();
                        // if io is left
                        if (currentJob.ioLeft()) {
                            // resume io
                            currentJob.incNextIO();
                            // add the job to the appropriate index of
the waiting queue
                            if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(current
Job);
                            }
                        }
                    } else {
                        currentJob.setPreempt();
                    }
                }
            }
            // cout << timeline << " " << ready_queue.size() << "\n";
            totalTurnaroundTime += ready_queue.size();
            totalWaitingTime += ready_queue.size();
            ++timeline;
        }
        cout << "\nRound Robin Gantt Chart --> \n";
        for (pair<int, int> t : ganttChart) {
            cout << "{Job " << t.first << " @ Time " << t.second << "}
";
        }

    }
    void showAnalysis() {
        cout << "\n\nRound Robin Scheduling Algorithm Statistics: \n";
```

```
        JobScheduler::showAnalysis();
    }
};

// class for priority based scheduling, inherited from JobScheduler
class
class Priority_Scheduler: public JobScheduler {
    priority_queue<Job, vector<Job>, JobComparatorPriority>
ready_queue; // ready queue for CPU
    unordered_map<int, vector<Job> > block_queue; // block queue for
i/o operations
    vector<pair<int, int> > ganttChart;  // to store the schedule
    int timeSlice;
public:
    // sort the jobs based on arrival time
    Priority_Scheduler(string filename): JobScheduler(filename) {
        sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
        timeSlice = 25;
    }
    virtual void schedule() {
        int sz = jobs.size(), index = 0;
        for (int i = 0; i < sz; ++i) {  // push all the processes
initially into the block queue
            block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
            totalTurnaroundTime += jobs[i].getTotalTime();
            // cout << jobs[i].getJobId() << " -> " <<
jobs[i].getArrivalTime() << "\n";
        }
        int timeline = 0;
        bool cpuEmpty = true;
        Job currentJob; int nextTerminate = INT_MAX;
        while (jobsLeft > 0) {
            // job coming from wait queue for the current timeline
            for (Job j : block_queue[timeline]) {
                ready_queue.push(j);
            }
            // if the current time is the end of cpu burst of a job
            if (timeline == nextTerminate) {
                // free the cpu
                cpuEmpty = true;
                // if the job terminates, reduce remaining job count
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
```

```
                // if it is end of timeslice, remove it from running
state and add to ready queue
                else if (currentJob.getPreempt() == true &&
currentJob.getCurrCPUTime() > 0) {
                    currentJob.unsetPreempt();
                    ready_queue.push(currentJob);
                }
            }
            // if the cpu is free
            if (cpuEmpty) {
                // and there are jobs on the ready queue
                if (!ready_queue.empty()) {
                    // pick the first job from the ready queue
                    currentJob = ready_queue.top();
                    ready_queue.pop();
                    // compute the time of end of its cpu burst

ganttChart.push_back(make_pair(currentJob.getJobId(), timeline));
                    // occupy the cpu
                    cpuEmpty = false;
                    // get the next terminating point
                    int val = min(currentJob.getCurrCPUTime(),
timeSlice);
                    nextTerminate = timeline + val;
                    currentJob.updateArrival(val);
                    // update its next arrival time
                    if (currentJob.getCurrCPUTime() == 0) {
                        currentJob.incNextCPU();
                        // if io is left
                        if (currentJob.ioLeft()) {
                            // resume io
                            currentJob.incNextIO();
                            // add the job to the appropriate index of
the waiting queue
                            if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(current
Job);
                            }
                        }
                    } else {
                        currentJob.setPreempt();
                    }
```

```
                }
            }
            // cout << timeline << " " << ready_queue.size() << "\n";
            totalTurnaroundTime += ready_queue.size();
            totalWaitingTime += ready_queue.size();
            ++timeline;
        }
        cout << "\nPriority Gantt Chart --> \n";
        for (pair<int, int> t : ganttChart) {
            cout << "{Job " << t.first << " @ Time " << t.second << "}
";
        }

    }
    void showAnalysis() {
        cout << "\n\nPriority Based Scheduling Algorithm Statistics:
\n";
        JobScheduler::showAnalysis();
    }
};

// class to run the simulation
class Runner {
    string filename;
public:
    Runner(string f) {
        filename = f;
    }
    void filegenerator() {
        int sz = 30;
        string filename = "jobprofiles_random.txt";
        ofstream fout;
        fout.open(filename, ios::out);
        for (int i = 1; i <= sz; ++i) {
            fout << i << " ";
            int arrival = rand() % 100;
            fout << arrival << " ";
            int priority = rand() % 17;
            fout << priority << " ";
            int burstsz = 1 + rand() % 13;
            for (int j = 0; j < burstsz; ++j) {
                // cpu
                int exp = rand() % 7;
```

```
                fout << (1 << exp) << " ";
                // i/o
                exp = rand() % 7;
                fout << (1 << exp) << " ";
            }
            int last = rand() % 2;
            if (last) {
                int exp = rand() % 7;
                fout << (1 << exp) << " ";
            }
            fout << "-1 ";
        }
    }
    void run() {
        cout << "Generate random file: (y/n) ";
        char choice;
        cin >> choice;
        if (choice == 'y' || choice == 'Y') {
            filename = "jobprofiles_random.txt";
            filegenerator();
        }
        FCFS_Scheduler F(filename);
        RoundRobin_Scheduler R(filename);
        Priority_Scheduler P(filename);
        F.schedule();
        R.schedule();
        P.schedule();
        F.showAnalysis();
        R.showAnalysis();
        P.showAnalysis();
    }
};


signed main() {
    Runner R("jobprofiles.txt");
    R.run();
    return 0;
}
```

2. Create child processes: X and Y.
a. Each child process performs 10 iterations. The child process displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have different outputs (i.e. another interleaving of processes' traces).

CODE :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10


void displayData(char* processName, int iteration){

    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName,
getpid(), iteration+1);
    int x = rand() %5;
    sleep(x);
}

int main()
{

    int pidX = -1;
    int pidY = -1;

    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
          srand(time(0)+getpid());
        for (int i = 0; i < ITERATIONS; i++){

            displayData("X", i);
        }
    }
    else
    {
```

```
        pidY = fork();
        if (pidY == 0){
            // Inside Child Y
            srand(time(0)+getpid());
            for (int i = 0; i < ITERATIONS; i++)
            {
                displayData("Y", i);
            }
        }
    }

    for (int i = 1; i <= 2; i++){
        wait(NULL);
    }

    return 0;
}
```

OUTPUT :

adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/Q2$ ./a
Process : X | PID : 13551 | Iteration : 1
Process : Y | PID : 13552 | Iteration : 1
Process : X | PID : 13551 | Iteration : 2
Process : X | PID : 13551 | Iteration : 3
Process : Y | PID : 13552 | Iteration : 2
Process : Y | PID : 13552 | Iteration : 3
Process : X | PID : 13551 | Iteration : 4
Process : Y | PID : 13552 | Iteration : 4
Process : X | PID : 13551 | Iteration : 5
Process : X | PID : 13551 | Iteration : 6
Process : X | PID : 13551 | Iteration : 7
Process : Y | PID : 13552 | Iteration : 5
Process : X | PID : 13551 | Iteration : 8
Process : X | PID : 13551 | Iteration : 9
Process : X | PID : 13551 | Iteration : 10
Process : Y | PID : 13552 | Iteration : 6
Process : Y | PID : 13552 | Iteration : **7**
Process : Y | PID : 13552 | Iteration : 8
Process : Y | PID : 13552 | Iteration : 9
Process : Y | PID : 13552 | Iteration : 10

b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronisation.

CODE :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10
sem_t *sem1;

void displayData(char *processName, int iteration)
{

    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName,
getpid(), iteration + 1);
    int x = rand() % 3;
    sleep(x);
}

int main()
{

    sem_unlink("sem1");
    sem1 = sem_open("sem1", O_CREAT, 0777, 0);

    int pidX = -1;
    int pidY = -1;

    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        srand(time(0) + getpid());
        for (int i = 0; i < ITERATIONS; i++)
        {
```

```
            sem_wait(sem1);
            displayData("X", i);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0)
        {
            // Inside Child Y
            srand(time(0) + getpid());
            for (int i = 0; i < ITERATIONS; i++)
            {

                displayData("Y", i);
                sem_post(sem1);
            }
        }
    }

    for (int i = 1; i <= 2; i++)
    {
        wait(NULL);
    }

    return 0;
}
```

OUTPUT :

```
adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/Q2$ ./b
Process  : Y  | PID : 14799 | Iteration : 1
Process  : Y  | PID : 14799 | Iteration : 2
Process  : X  | PID : 14798 | Iteration : 1
Process  : X  | PID : 14798 | Iteration : 2
Process  : Y  | PID : 14799 | Iteration : 3
Process  : Y  | PID : 14799 | Iteration : 4
Process  : X  | PID : 14798 | Iteration : 3
Process  : Y  | PID : 14799 | Iteration : 5
Process  : Y  | PID : 14799 | Iteration : 6
Process  : X  | PID : 14798 | Iteration : 4
Process  : X  | PID : 14798 | Iteration : 5
Process  : Y  | PID : 14799 | Iteration : 7
```

Process : X | PID : 14798 | Iteration : 6
Process : Y | PID : 14799 | Iteration : 8
Process : X | PID : 14798 | Iteration : 7
Process : Y | PID : 14799 | Iteration : 9
Process : X | PID : 14798 | Iteration : 8
Process : Y | PID : 14799 | Iteration : 10
Process : X | PID : 14798 | Iteration : 9
Process : X | PID : 14798 | Iteration : 10

c. Modify the program so that X and Y now perform in lockstep [both perform iteration I, then iteration i+1, and so on] with the condition mentioned in Q (2b) above.

CODE :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10

void displayData(char *processName, int iteration)
{

    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName,
getpid(), iteration + 1);
    int x = rand() % 5;
    sleep(x);
}

int main()
{

    int pidX = -1;
    int pidY = -1;

    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
```

```
        srand(time(0) + getpid());
        for (int i = 0; i < ITERATIONS; i++)
        {

            displayData("X", i);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0)
        {
            // Inside Child Y
            srand(time(0) + getpid());
            for (int i = 0; i < ITERATIONS; i++)
            {
                displayData("Y", i);
            }
        }
    }

    for (int i = 1; i <= 2; i++)
    {
        wait(NULL);
    }

    return 0;
}
```

OUTPUT :

adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/Q2$ ./c
Process : X | PID : 15039 | Iteration : 1
Process : Y | PID : 15040 | Iteration : 1
Process : X | PID : 15039 | Iteration : 2
Process : Y | PID : 15040 | Iteration : 2
Process : X | PID : 15039 | Iteration : 3
Process : Y | PID : 15040 | Iteration : 3
Process : X | PID : 15039 | Iteration : 4
Process : Y | PID : 15040 | Iteration : 4

Process : X | PID : 15039 | Iteration : 5
Process : Y | PID : 15040 | Iteration : 5
Process : X | PID : 15039 | Iteration : 6
Process : Y | PID : 15040 | Iteration : 6
Process : X | PID : 15039 | Iteration : 7
Process : Y | PID : 15040 | Iteration : 7
Process : X | PID : 15039 | Iteration : 8
Process : Y | PID : 15040 | Iteration : 8
Process : X | PID : 15039 | Iteration : 9
Process : Y | PID : 15040 | Iteration : 9
Process : X | PID : 15039 | Iteration : 10
Process : Y | PID : 15040 | Iteration : 10

d. Add another child process Z.
Perform the operations as mentioned in Q (2a) for all three children.
Then perform the operations as mentioned in Q (2c) [that is, 3 children in lockstep].

CODE :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10
sem_t *semx;
sem_t *semy;
sem_t *semz;

void displayData(char *processName, int iteration)
{

    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName,
getpid(), iteration + 1);
    int x = rand() % 5;
    sleep(x);
}
```

```
int main()
{

    sem_unlink("semx");
    semx = sem_open("semx", O_CREAT, 0777, 1);
    sem_unlink("semy");
    semy = sem_open("semy", O_CREAT, 0777, 0);
    sem_unlink("semz");
    semz = sem_open("semz", O_CREAT, 0777, 0);

    int pidX;
    int pidY;
    int pidZ;

    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        srand(time(0) + getpid());
        for (int i = 0; i < ITERATIONS; i++)
        {

            sem_wait(semx);
            displayData("X", i);
            sem_post(semy);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0)
        {
            // Inside Child Y
            srand(time(0) + getpid());
            for (int i = 0; i < ITERATIONS; i++)
            {
                sem_wait(semy);
                displayData("Y", i);
                sem_post(semz);
            }
        }
        else
        {
```

```
            pidZ = fork();
            if (pidZ == 0)
            {
                srand(time(0) + getpid());
                for (int i = 0; i < ITERATIONS; i++)
                {
                    sem_wait(semz);
                    displayData("Z", i);
                    sem_post(semx);
                }
            }
        }
    }

    for (int i = 1; i <= 2; i++)
    {
        wait(NULL);
    }

    return 0;
}
```

OUTPUT :

adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/Q2$ ./d
Process : X | PID : 15421 | Iteration : 1
Process : Y | PID : 15422 | Iteration : 1
Process : Z | PID : 15423 | Iteration : 1
Process : X | PID : 15421 | Iteration : 2
Process : Y | PID : 15422 | Iteration : 2
Process : Z | PID : 15423 | Iteration : 2
Process : X | PID : 15421 | Iteration : 3
Process : Y | PID : 15422 | Iteration : 3
Process : Z | PID : 15423 | Iteration : 3
Process : X | PID : 15421 | Iteration : 4
Process : Y | PID : 15422 | Iteration : 4
Process : Z | PID : 15423 | Iteration : 4
Process : X | PID : 15421 | Iteration : 5
Process : Y | PID : 15422 | Iteration : 5
Process : Z | PID : 15423 | Iteration : 5
Process : X | PID : 15421 | Iteration : 6
Process : Y | PID : 15422 | Iteration : 6
Process : Z | PID : 15423 | Iteration : 6

Process : X | PID : 15421 | Iteration : 7
Process : Y | PID : 15422 | Iteration : 7
Process : Z | PID : 15423 | Iteration : 7
Process : X | PID : 15421 | Iteration : 8
Process : Y | PID : 15422 | Iteration : 8
Process : Z | PID : 15423 | Iteration : 8
Process : X | PID : 15421 | Iteration : 9
Process : Y | PID : 15422 | Iteration : 9
Process : Z | PID : 15423 | Iteration : 9
Process : X | PID : 15421 | Iteration : 10
Process : Y | PID : 15422 | Iteration : 10
Process : Z | PID : 15423 | Iteration : 10

3. Implement the following applications using different IPC mechanisms. Your choice is restricted to Pipe,
FIFO:
a. Broadcasting weather information (one broadcasting process and more than one listeners)

CODE :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <time.h>
#include <sys/wait.h>
#include <string.h>

#define BUFFER_LEN 200

int main()
{
    int lc;
    printf("Enter number of listeners : ");
    scanf("%d", &lc);
```

```
    if (lc == 0)
    {
        printf("Listeners cannot be zero\n");
        exit(0);
    }

    // dynamic array of file descriptors
    int **fdarr = (int **)malloc(lc * sizeof(int *));
    for (int i = 0; i < lc; i++)
        fdarr[i] = (int *)malloc(2 * sizeof(int));

    for (int i = 0; i < lc; i++)
    {
        int res = pipe(fdarr[i]);
        if (res == -1)
        {
            printf("Pipe creation failed for listener %d", i + 1);
            return 1;
        }
    }

    // forking a listener creator process
    int listener_creator = fork();
    if (listener_creator == -1)
    {
        printf("Fork failed");
        exit(0);
    }

    if (listener_creator == 0)
    {
        for (int i = 0; i < lc; i++)
        {
            int listener = fork();
            if (listener < 0)
            {
                perror("A listener creation failed\n");
                exit(1);
            }
            int listener_id = i + 1;
            if (listener == 0)
            {
                char data[BUFFER_LEN];
```

```
                while (1)
                {
                    int res = read(fdarr[listener_id - 1][0], data,
BUFFER_LEN);
                    if (res > 0)
                    {
                        if (strcmp(data, "EOF") == 0)
                        {
                            printf("\nListener %d finished\n",
listener_id);
                            break;
                        }
                        printf("Listener %d: %s\n", listener_id, data);
                        sleep(0.5);
                    }
                }

                close(fdarr[listener_id - 1][0]);
            }
        }
    }
    else
    {
        // Close all read ends of pipes
        for (int i = 0; i < lc; i++)
        {
            close(fdarr[i][0]);
        }
        // Open file
        FILE *fp = fopen("weather.txt", "r");
        if (fp == NULL)
        {
            printf("File open failed");
            return 1;
        }
        // Read file line by line
        char buffer[BUFFER_LEN];
        while (fgets(buffer, BUFFER_LEN, fp) != NULL)
        {
            // Write to all pipes
            for (int i = 0; i < lc; i++)
            {
                write(fdarr[i][1], buffer, BUFFER_LEN);
```

```
            }
        }

        // Send something to all pipes to indicate end of file
        for (int i = 0; i < lc; i++)
        {
            write(fdarr[i][1], "EOF", BUFFER_LEN);
        }

        // Close all write ends of pipes
        for (int i = 0; i < lc; i++)
        {
            close(fdarr[i][1]);
        }

        // Wait for child process to finish
        wait(NULL);
    }

    return 0;
}
```

OUTPUT :
adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/Q3/weather_broadcast$ ./weather
Enter number of listeners : 5
Listener 1: Forecast For Friday 05/18/20XX

Listener 2: Forecast For Friday 05/18/20XX

Listener 4: Forecast For Friday 05/18/20XX

Listener 3: Forecast For Friday 05/18/20XX

Listener 5: Forecast For Friday 05/18/20XX

Listener 1: Maximum temperature today near 86 degrees.

Listener 2: Maximum temperature today near 86 degrees.

Listener 4: Maximum temperature today near 86 degrees.

Listener 3: Maximum temperature today near 86 degrees.

Listener 5: Maximum temperature today near 86 degrees.

Listener 1: A partly cloudy and warm day is expected.

Listener 2: A partly cloudy and warm day is expected.

Listener 4: A partly cloudy and warm day is expected.

Listener 5: A partly cloudy and warm day is expected.

Listener 1: Lowest relative humidity near 33 percent.

Listener 3: A partly cloudy and warm day is expected.

Listener 2: Lowest relative humidity near 33 percent.

Listener 4: Lowest relative humidity near 33 percent.

Listener 5: Lowest relative humidity near 33 percent.

Listener 1: Expect 13 hours of sunshine which is 87 percent of possible sunshine.
Listener 3: Lowest relative humidity near 33 percent.

Listener 2: Expect 13 hours of sunshine which is 87 percent of possible sunshine.
Listener 4: Expect 13 hours of sunshine which is 87 percent of possible sunshine.
Listener 5: Expect 13 hours of sunshine which is 87 percent of possible sunshine.

Listener 1 finished

Listener 2 finished
Listener 3: Expect 13 hours of sunshine which is 87 percent of possible sunshine.

Listener 4 finished

Listener 3 finished

Listener 5 finished

b. Telephonic conversation (between a caller and a receiver)

CODE :

Caller.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <stdbool.h>

int main(int argc, char *argv[])
{
    int canwrite = 1;
    char msg[200];
    mkfifo("./myfifo", 0666);
    while (true)
    {
        if (canwrite)
        {

            printf("Enter reply: ");
            fgets(msg, 200, stdin);
            msg[strcspn(msg, "\n")] = 0;
            fflush(stdin);
            int fd = open("myfifo", O_WRONLY);
            if (fd == -1)
            {
                printf("Error opening fifo\n");
                exit(1);
            }

            write(fd, msg, sizeof(msg));
            close(fd);
            canwrite = 0;
        }
        else
        {
            int fd = open("myfifo", O_RDONLY);
            if (fd == -1)
            {
                printf("Error opening fifo\n");
```

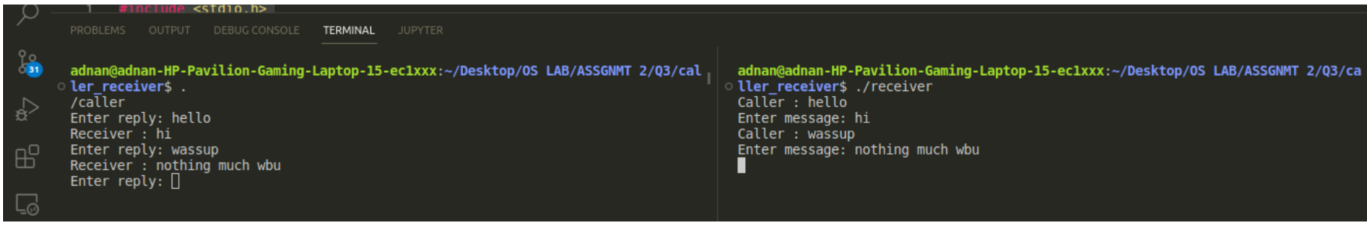
```
                exit(1);
            }
            read(fd, msg, sizeof(msg));
            close(fd);
            printf("Receiver : %s\n", msg);
            canwrite = 1;
        }

        fflush(stdin);
        fflush(stdout);
    }

    return 0;
}
```

Receiver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <stdbool.h>

int main(int argc, char *argv[])
{
    int canRead = 1;
    char msg[200];
    while (true)
    {
        if (canRead)
        {
            int fd = open("myfifo", O_RDONLY);
            if (fd == -1)
            {
                printf("Error opening fifo\n");
                exit(1);
            }
            read(fd, msg, sizeof(msg));
            close(fd);
```

```
            printf("Caller : %s\n", msg);
            canRead = 0;
        }
        else
        {
            printf("Enter reply: ");
            fgets(msg, 200, stdin);
            msg[strcspn(msg, "\n")] = 0;
            fflush(stdin);
            int fd = open("myfifo", O_WRONLY);
            if (fd == -1)
            {
                printf("Error opening fifo\n");
                exit(1);
            }

            write(fd, msg, sizeof(msg));
            close(fd);
            canRead = 1;
        }

        fflush(stdin);
        fflush(stdout);
    }

    return 0;
}
```

OUTPUT :



4. Write a program for p-producer c-consumer problem, p, c >= 1. A shared circular buffer that can hold 25
items is to be used. Each producer process stores any numbers between 1 to 80 (along with the producer id)
in the buffer one by one and then exits. Each consumer process reads the numbers from the buffer and adds

them to a shared variable TOTAL (initialised to 0). Though any consumer process can read any of the
numbers in the buffer, the only constraint being that every number written by some producer should be read
exactly once by exactly one of the consumers.
(a) The program reads in the value of p and c from the user, and forks p producers and c consumers.
(b) After all the producers and consumers have finished (the consumers exit after all the data produced by
all producers have been read), the parent process prints the value of TOTAL.
Test the program with different values of p and c.

CODE :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <time.h>
#include <sys/wait.h>

/* name of the semaphore */
#define SEMOBJ_NAME "/mutex"

/* maximum number of seconds to sleep between each loop operation */
#define MAX_SLEEP_SECS 3

/* maximum buffer size */
#define BUFFER_SIZE 25

/* defining the structure for buffer */
typedef struct
{
    int in, out;
    int items[BUFFER_SIZE]; // shared circular queue
    int TOTAL;
    int produce_complete, consume_complete;
} BUFFER;
```

```
/* initialize the buffer */
void init_buffer(BUFFER *buf)
{
    buf->in = 0;
    buf->out = 0;
    buf->TOTAL = 0;
    buf->produce_complete = 0;
    buf->consume_complete = 0;
    for (int i = 0; i < BUFFER_SIZE; i++)
        buf->items[i] = 0;
}

int main()
{
    int p, c;
    int flag1, flag2;
    printf("Enter no of producers: ");
    scanf("%d", &p);
    printf("Enter no of consumers: ");
    scanf("%d", &c);
    if (p & c == 0)
    {
        perror("No of consumers or producers cannot be zero");
        exit(1);
    }

    /* getting a new semaphore for the shared segment */
    sem_unlink(SEMOBJ_NAME);
    sem_t *bufmutex = sem_open(SEMOBJ_NAME, O_CREAT, 0777, 1);
    if (bufmutex == SEM_FAILED)
    {
        perror("In sem_open()");
        exit(1);
    }
    /* requesting the semaphore not to be held when completely
unreferenced */
    sem_unlink(SEMOBJ_NAME);

    /* requesting the shared segment */
    BUFFER *buf = (BUFFER *)mmap(NULL, sizeof(BUFFER), PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
    if (buf == MAP_FAILED)
    {
```

```
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d
bytes) at %p.\n", (int)sizeof(BUFFER), buf);

    /* initialize the buffer */
    init_buffer(buf);

    /* seeding the random number generator */
    // srand (time(NULL));

    /* create the producer manager process */
    int producer_manager = fork();

    if (producer_manager < 0)
    {
        perror("In producer manager process");
        exit(1);
    }

    else if (producer_manager == 0)
    {
        for (int i = 0; i < p; i++)
        {
            // sleep(1);
            // srand(time(NULL));

            // create a new producer
            int producer = fork();
            if (producer < 0)
            {
                perror("In producer process");
                exit(1);
            }
            else if (producer == 0)
            {
                int item_produced = 0;
                srand(time(0) + getpid());
                int x = rand() % 10 + 1;
                // loop until all item produced
                while (item_produced < x)
                {
```

```
                    sem_wait(bufmutex);
                    // produce item
                    int produce = random() % 80 + 1;
                    buf->items[buf->in] = produce;
                    printf("Producer %d produced %d at %d\n", getpid(),
produce, buf->in);
                    buf->in = ((buf->in + 1) % BUFFER_SIZE);
                    sem_post(bufmutex);
                    item_produced++;
                }
                exit(0);
            }
        }
        wait(NULL);
        buf->produce_complete = 1;
        exit(0);
    }

    /* create the consumer manager process */
    int consumer_manager = fork();

    if (consumer_manager < 0)
    {
        perror("In consumer manager process");
        exit(1);
    }
    else if (consumer_manager == 0)
    {
        for (int i = 0; i < c; i++)
        {
            // create a new consumer
            int consumer = fork();

            // wait for the semaphore for mutual exclusive access to
the buffer
            if (consumer < 0)
            {
                perror("In consumer process");
                exit(1);
            }

            // wait if the buffer is empty
```

```
            else if (consumer == 0)
            {
                srand(time(0) + getpid());
                int item_consumed = 0;
                int y = random() % 10 + 1;
                while (item_consumed < y)
                {
                    // consume item
                    sem_wait(bufmutex);
                    if (buf->in == buf->out)
                    {
                        sem_post(bufmutex);
                        continue;
                    }

                    int consume = buf->items[buf->out];
                    printf("Consumer %d consumed %d at %d\n", getpid(),
consume, buf->out);
                    buf->out = (buf->out + 1) % BUFFER_SIZE;
                    buf->TOTAL += consume;

                    // release the semaphore
                    sem_post(bufmutex);
                    item_consumed++;
                }
                exit(0);
            }
        }
        wait(NULL);
        buf->consume_complete = 1;
        exit(0);
    }

    while (wait(NULL) > 0)
        ;

    /* freeing the reference to the semaphore */
    sem_close(bufmutex);
    printf("TOTAL -> %d\n", buf->TOTAL);

    /* release the shared memory space */
    munmap(buf, sizeof(BUFFER));
```

```
    return 0;
}
```

OUTPUT :
adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS
LAB/ASSGNMT 2$ ./a.out
Enter no of producers: 5
Enter no of consumers: 8
Shared memory segment allocated correctly (120 bytes) at 0x7fdc60282000.
Producer 18190 produced 4 at 0
Producer 18190 produced 54 at 1
Producer 18190 produced 56 at 2
Consumer 18200 consumed 4 at 0
Consumer 18200 consumed 54 at 1
Consumer 18201 consumed 56 at 2
Producer 18192 produced 50 at 3
Producer 18192 produced 61 at 4
Producer 18194 produced 60 at 5
Producer 18194 produced 73 at 6
Producer 18194 produced 63 at 7
Producer 18194 produced 53 at 8
Producer 18194 produced 25 at 9
Producer 18194 produced 6 at 10
Producer 18194 produced 43 at 11
Producer 18194 produced 54 at 12
Producer 18194 produced 65 at 13
Consumer 18199 consumed 50 at 3
Consumer 18193 consumed 61 at 4
Consumer 18193 consumed 60 at 5
Consumer 18193 consumed 73 at 6
Consumer 18193 consumed 63 at 7
Consumer 18193 consumed 53 at 8
Consumer 18193 consumed 25 at 9
Producer 18198 produced 45 at 14
Producer 18198 produced 38 at 15
Producer 18198 produced 48 at 16
Producer 18198 produced 31 at 17
Consumer 18202 consumed 6 at 10
Consumer 18202 consumed 43 at 11
Consumer 18202 consumed 54 at 12
Producer 18196 produced 49 at 18
Consumer 18197 consumed 65 at 13
Consumer 18197 consumed 45 at 14

Consumer 18197 consumed 38 at 15
Consumer 18197 consumed 48 at 16
Consumer 18197 consumed 31 at 17
Consumer 18197 consumed 49 at 18
TOTAL -> 878

5. Write a program for the Reader-Writer process for the following situations:
a) Multiple readers and one writer: writer gets to write whenever it is ready (reader/s wait)

Approach :
First of all I have created a shared memory variable which will be read and updated by readers and writers respectively. Then I have created a semaphore which will help in achieving exclusion during writing. I have taken the number of readers and the number of times the writer will do write operations from the user. Then I have forked a reader creator process which will fork N readers and I have written the code for the writer in main itself.
In the writer I have used wait() on the semaphore so when the writer wants to write it takes control of the shared variable and I have used wait() in the beginning of the reader so it will wait until the writer has completed its operation and has incremented the semaphore again. Now when the reader gets the signal it gets out of wait and immediately signals the semaphore again so multiple readers can read simultaneously.

Code :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <time.h>
#include <sys/wait.h>

int main()
{

    int *var = (int *)mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANON, -1, 0);
```

```
    int *writing = (int *)mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);

    if (var == MAP_FAILED)
    {
        perror("In mmap()");
        exit(1);
    }

    int readers;
    printf("Enter number of readers : ");
    scanf("%d", &readers);

    int wcount;

    printf("Enter how many times you want to write : ");
    scanf("%d", &wcount);

    // creating readers

    int reader_creator = fork();

    if (reader_creator == 0)
    {
        for (int i = 0; i < readers; i++)
        {
            int reader = fork();

            if (reader < 0)
            {
                perror("In fork()");
                exit(1);
            }

            int r = i + 1;

            if (reader == 0)
            {

                srand(time(0));
                int rcount = (rand() % (8 - 4 + 1)) + 4;

                while (rcount--)
```

```
                {
                    while (*writing)
                        ;
                    printf("Reader %d read %d\n", r, *var);
                    int number = (rand() % (12 - 2 + 1)) + 1;
                    sleep(number);
                }

                return 01;
            }
        }

        while (wait(NULL) > 0)
            ;
        return 0;
    }

    while (wcount--)
    {
        (*writing) = 1;
        srand(0);
        int x = rand() % 100;
        *var += x;
        printf("Writer writing %d\n", *var);
        sleep(2);
        (*writing) = 0;
        int number = (rand() % (8 - 1 + 1)) + 2;
        sleep(number);
    }

    wait(NULL);

    return 0;
}
```

5.

b) Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other writer is currently writing (reader/s wait)

CODE :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <time.h>
#include <sys/wait.h>

sem_t *wrtMutex, *waitMutex;

int main()
{

    int *var = (int *)mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANON, -1, 0);
    int *w_wrtcount = (int *)mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
    int *w_waitcount = (int *)mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);

    if (var == MAP_FAILED)
    {
        perror("In mmap()");
        exit(1);
    }

    int readers, writers;
    sem_unlink("/wrtMutex");
    wrtMutex = sem_open("/wrtMutex", O_CREAT, 0777, 1);
    sem_unlink("/waitMutex");
    waitMutex = sem_open("/waitMutex", O_CREAT, 0777, 1);
    printf("Enter number of readers : ");
    scanf("%d", &readers);
    printf("Enter number of writers : ");
    scanf("%d", &writers);

    int writer_creator = fork();
```

```
    if (writer_creator == 0)
    {
        for (int i = 0; i < writers; i++)
        {
            int writer = fork();

            int w = i + 1;

            if (writer == 0)
            {
                srand(time(0));
                int wcount = (rand() % (4 - 2 + 1)) + 2;
                while (wcount--)
                {
                    sem_wait(waitMutex);
                    (*w_waitcount)++;
                    sem_post(waitMutex);

                    sem_wait(wrtMutex);

                    (*w_wrtcount)++;

                    sem_wait(waitMutex);
                    (*w_waitcount)--;
                    sem_post(waitMutex);

                    srand(time(0));
                    int x = rand() % 100;
                    *var += x;
                    printf("Writer %d writing %d\n", w, *var);
                    sleep(2);
                    printf("Writer %d finished writing %d\n", w, *var);
                    (*w_wrtcount)--;
                    sem_post(wrtMutex);

                    if (wcount)
                    {
                        srand(time(0));
                        int number = (rand() % (12 - 6 + 1)) + 6;
                        sleep(number);
                        // printf("Writer %d wants to write again\n",
w);
                    }
```

```
            }
            return 0;
        }
    }
    while (wait(NULL) > 0)
        ;

    return 0;
}

int reader_creator = fork();

if (reader_creator == 0)
{
    for (int i = 0; i < readers; i++)
    {
        int reader = fork();

        int r = i + 1;

        if (reader == 0)
        {

            srand(time(0));
            int rcount = (rand() % (8 - 2 + 1)) + 2;

            while (rcount--)
            {
                while (*w_waitcount > 0 || *w_wrtcount > 0)
                    ;
                printf("Reader %d read %d\n", r, *var);
                if (rcount)
                {
                    srand(time(0));
                    int number = (rand() % (8 - 1 + 1)) + 1;
                    sleep(number);
                }
            }

            return 0;
        }
    }
```

```
        while (wait(NULL) > 0)
            ;
        return 0;
    }

    while (wait(NULL) > 0)
        ;
    sem_unlink("/wrtMutex");
    sem_unlink("/waitMutex");
    sem_destroy(wrtMutex);
    sem_destroy(waitMutex);

    return 0;
}
```

OUTPUT :

```
Enter number of readers : 8
Enter number of writers : 4
Writer 1 writing 7
Writer 1 finished writing 7
Writer 2 writing 70
Writer 2 finished writing 70
Writer 3 writing 149
Writer 3 finished writing 149
Writer 4 writing 237
Writer 4 finished writing 237
Reader 4 read 237
Reader 7 read 237
Reader 2 read 237
Reader 6 read 237
Reader 1 read 237
Reader 3 read 237
Reader 5 read 237
Reader 8 read 237
Writer 2 writing 324
Writer 2 finished writing 324
Reader 4 read 324
Reader 8 read 324
Reader 7 read 324
Reader 3 read 324
Reader 2 read 324
Reader 6 read 324
Reader 5 read 324
Reader 1 read 324
Writer 3 writing 392
Writer 3 finished writing 392
Writer 1 writing 449
Writer 1 finished writing 449
Reader 4 read 449
```

```
Reader 8 read 237
Writer 2 writing 324
Writer 2 finished writing 324
Reader 4 read 324
Reader 8 read 324
Reader 7 read 324
Reader 3 read 324
Reader 2 read 324
Reader 6 read 324
Reader 5 read 324
Reader 1 read 324
Writer 3 writing 392
Writer 3 finished writing 392
Writer 1 writing 449
Writer 1 finished writing 449
Reader 4 read 449
Reader 2 read 449
Reader 8 read 449
Reader 5 read 449
Reader 1 read 449
Reader 7 read 449
Reader 6 read 449
Reader 3 read 449
Writer 4 writing 478
Writer 4 finished writing 478
Reader 4 read 478
Reader 3 read 478
Reader 8 read 478
Reader 7 read 478
Reader 1 read 478
Reader 5 read 478
Reader 2 read 478
Reader 6 read 478
```

Ln 79, Col 19 Spaces: 4 UTF-8 LF C Go Live Linux

6. Implement Dining Philosophers' problem using Monitor. Test the program with (a) 5 philosophers
and 5 chopsticks, (b) 6 philosophers and 6 chopsticks, and (c) 7 philosophers and 7 chopsticks

APPROACH :

We use an enum state with values **THINKING –** When a philosopher doesn't want to gain access to either fork. **HUNGRY –** When a philosopher wants to enter the critical section. **EATING –** When the philosopher has got both the forks, i.eHe has entered the section.
Philosopher i can set the variable state[i] = EATING only if his two neighbours are not eating (state[(i+N-1) % N] != EATING) and (state[(i+1) % N] != EATING).

We also need to declare `condition self[5];`

This allows philosopher i to delay himself when he is hungry but is unable to obtain the chopsticks he needs.

CODE :

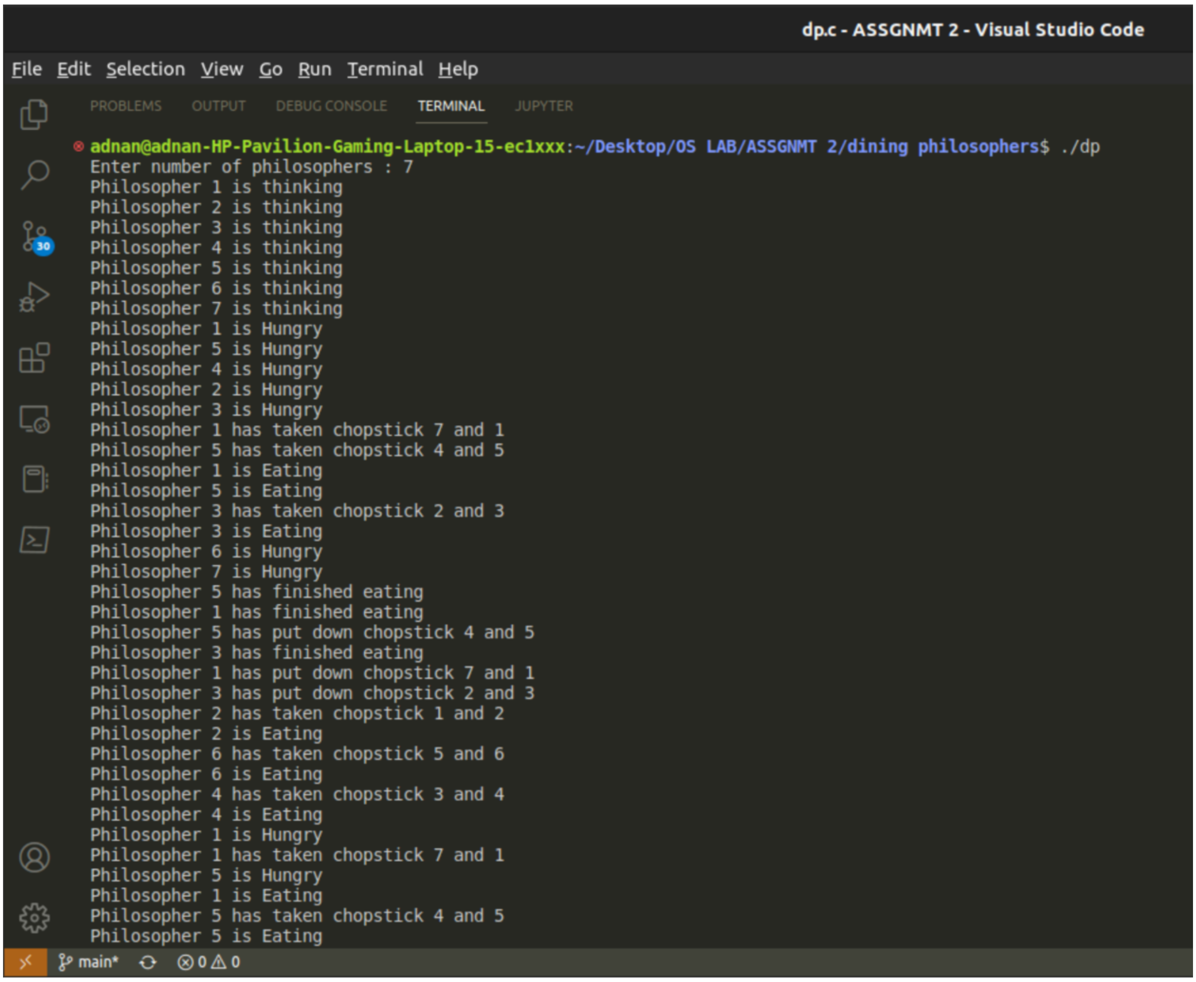
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
```

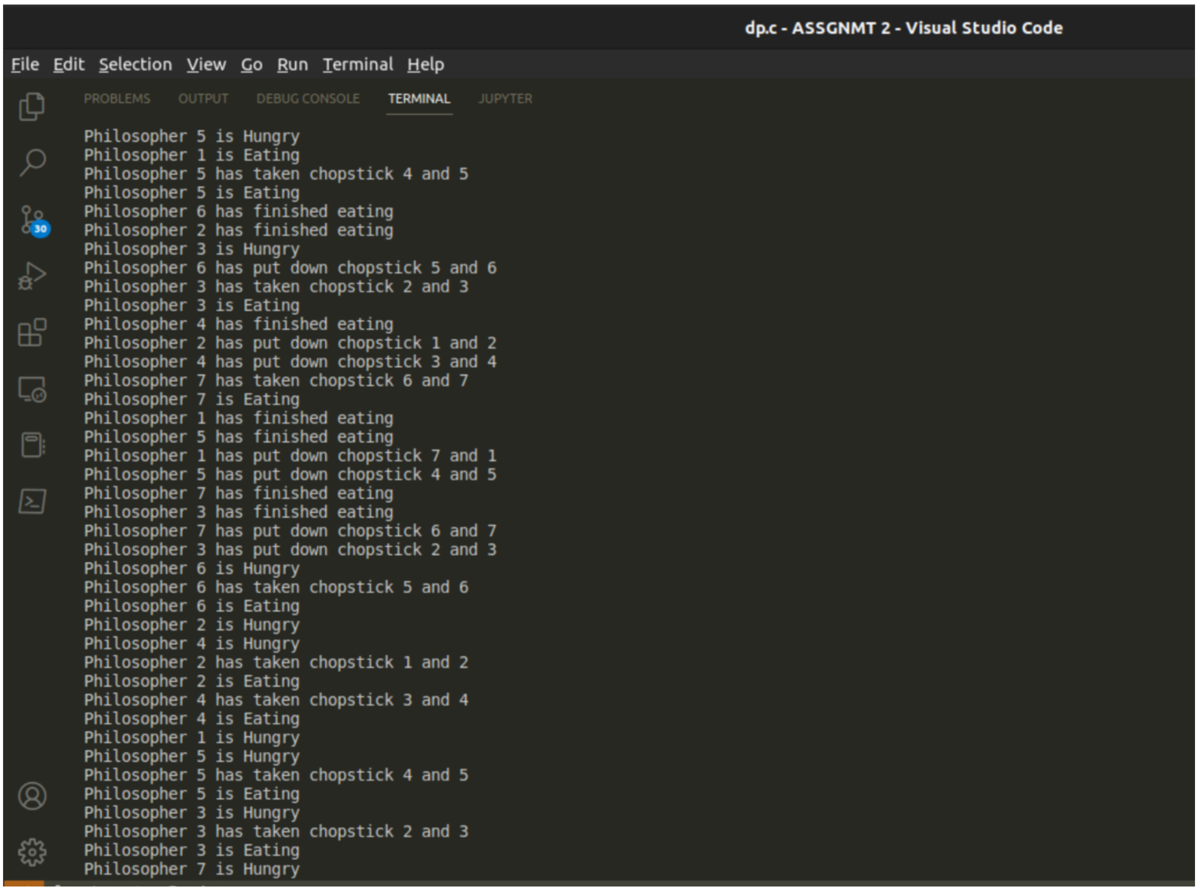
```
#include <sys/stat.h>
#include <semaphore.h>
#include <time.h>
#include <sys/wait.h>
#include <string.h>

char *create_sem_name(int i)
{
      char *buffer;
      buffer = (char *)malloc(sizeof(char) * 12);
      sprintf(buffer, "/self%d", i);
      return buffer;
}

int N;

enum State
{
      THINKING,
      HUNGRY,
      EATING
};

enum State *state;

sem_t *mutex;

sem_t **self;

void eat(int i)
{
      // eating
      printf("Philosopher %d has taken chopstick %d and %d\n", i + 1,
(i + N - 1) % N + 1, i + 1);
      printf("Philosopher %d is Eating\n", i + 1);
      sleep(3);
      printf("Philosopher %d has finished eating\n", i + 1);
      printf("Philosopher %d has put down chopstick %d and %d\n", i +
1, (i + N - 1) % N + 1, i + 1);
}

void test(int i)
{
```

```
	if ((state[(i + N - 1) % N] != EATING) && (state[i] == HUNGRY) &&
(state[(i + 1) % N] != EATING))
	{
		state[i] = EATING;
		sem_post(self[i]);
	}
}

void pickup(int i)
{
	state[i] = HUNGRY;
	printf("Philosopher %d is Hungry\n", i + 1);
	test(i);
	if (state[i] != EATING)
	{
		sem_wait(self[i]);
	}
}

void putdown(int i)
{
	state[i] = THINKING;
	test((i + N - 1) % N);
	test((i + 1) % N);
}

int main()
{

	printf("Enter number of philosophers : ");
	scanf("%d", &N);

	state = (enum State *)mmap(NULL, sizeof(enum State) * N,
PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);

	for (int i = 0; i < N; i++)
	{
		state[i] = THINKING;
		printf("Philosopher %d is thinking\n", i + 1);
	}

	mutex = sem_open("/mutex", O_CREAT, 0777, 1);
```

```
    self = (sem_t **)mmap(NULL, sizeof(sem_t *) * N, PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);

    for (int i = 0; i < N; i++)
    {
        char *name = create_sem_name(i + 1);
        sem_unlink(name);
        self[i] = sem_open(name, O_CREAT, 0777, 0);
    }

    int philosopher_creator = fork();

    if (philosopher_creator == 0)
    {
        for (int i = 0; i < N; i++)
        {
            int philosopher = fork();
            if (philosopher == 0)
            {
                while (1)
                {
                    srand(time(0));
                    int t = rand() % 5 + 1;
                    sleep(t);
                    pickup(i);
                    eat(i);
                    putdown(i);
                }
                return 0;
            }
        }
        while (wait(NULL) > 0)
            ;
        return 0;
    }

    wait(NULL);
    return 0;
}
  }
```

OUTPUT :


dp.c - ASSGNMT 2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
adnan@adnan-HP-Pavilion-Gaming-Laptop-15-ec1xxx:~/Desktop/OS LAB/ASSGNMT 2/dining philosophers$ ./dp
Enter number of philosophers : 7
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 6 is thinking
Philosopher 7 is thinking
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 1 has taken chopstick 7 and 1
Philosopher 5 has taken chopstick 4 and 5
Philosopher 1 is Eating
Philosopher 5 is Eating
Philosopher 3 has taken chopstick 2 and 3
Philosopher 3 is Eating
Philosopher 6 is Hungry
Philosopher 7 is Hungry
Philosopher 5 has finished eating
Philosopher 1 has finished eating
Philosopher 5 has put down chopstick 4 and 5
Philosopher 3 has finished eating
Philosopher 1 has put down chopstick 7 and 1
Philosopher 3 has put down chopstick 2 and 3
Philosopher 2 has taken chopstick 1 and 2
Philosopher 2 is Eating
Philosopher 6 has taken chopstick 5 and 6
Philosopher 6 is Eating
Philosopher 4 has taken chopstick 3 and 4
Philosopher 4 is Eating
Philosopher 1 is Hungry
Philosopher 1 has taken chopstick 7 and 1
Philosopher 5 is Hungry
Philosopher 1 is Eating
Philosopher 5 has taken chopstick 4 and 5
Philosopher 5 is Eating
main*
0 0



7. Write a program that will find out whether a system is in safe state or not with following
specifications:
Command line input: name of a file - The file contains the initial state of the system as given
below:
#no of resources 4 #no of instances of each resource 2 4 5 3
#no of processes 3 #no of instances of each resource that each process needs in its lifetime 1 1 1 1,
2 3 1 2, 2 2 1 3
The program waits to accept a resource allocation request to be supplied by the user or read from
another file:
For example: 0 1 0 1 1 indicates that p0 has requested allocation of 1 instance of R0, R2 and R3
each.
Your program should declare the result:
(1) should this request be granted?

(2) if your answer is yes, print the safe sequence in which all remaining needs can be granted one
by one and also grant the request. If the requesting process's need is NIL, the program internally
releases all its resources. Go back to accept another request till all processes finish with all their
needs.
Testing:
a. Generate possible request sequences of each process.
b. Each such sequence must satisfy the maximum requirements of the process.

APPROACH :

First we read the number of resources , maximum instances of each resource , number of processes and the maximum need of each process then we create an available vector which stores current available resources , an allocation matrix that holds the currently allocated resources to process , a need matrix that holds the need of the processes and a max matrix that stores the maximum requirement of a process in its lifetime. Then we take a request from user and store it in a request vector and check whether the request is <= need , if not then we discard the request else we check if resources are available then we proceed further else we discard the request. Now proceeding further we create duplicates of the vectors and matrices to pretend that we have granted the request and then run the safety algorithm to check if the system will remain in safe state or not after the allocation. If it is in fact in safe state then we update the original matrices but if not in safe state then we do not update the original matrices. We repeat this until need for all processes becomes zero.

CODE :

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

void printdata(vector<int> &available, vector<vector<int>> &max,
vector<vector<int>> &allocation, vector<vector<int>> &need)
{
    int p = max.size();
    int r = max[0].size();

    cout << "Process\t\t"
         << "Allocation\t"
```

```
         << "Max\t\t"
         << "Need\t\t"
         << "Available\t\t" << endl;

    for (int i = 0; i < p; i++)
    {
        cout << "P" << i << "\t\t";

        for (int j = 0; j < r; j++)
        {
            cout << allocation[i][j] << " ";
        }
        cout << "\t";

        for (int j = 0; j < r; j++)
        {
            cout << max[i][j] << " ";
        }

        cout << "\t";

        for (int j = 0; j < r; j++)
        {
            cout << need[i][j] << " ";
        }

        cout << "\t";

        if (i == 0)
        {
            for (int j = 0; j < r; j++)
            {
                cout << available[j] << " ";
            }
        }
        cout << endl;
    }
}

int main()
{

    // read from a file
```

```
ifstream fin("input.txt");

// number of resources is in the first line
int r;
fin >> r;

vector<int> available(r);

// available resources are in the second line
for (int i = 0; i < r; i++)
{
    fin >> available[i];
}

// number of processes is in the third line
int p;
fin >> p;

vector<vector<int>> max(p, vector<int>(r));

// max resources are in the fourth line
for (int i = 0; i < p; i++)
{
    for (int j = 0; j < r; j++)
    {
        fin >> max[i][j];
    }
}

// initializing an allocation matrix with all zeros

vector<vector<int>> allocation(p, vector<int>(r, 0));

// initializing a need matrix with all the needs of the processes

vector<vector<int>> need(p, vector<int>(r));

for (int i = 0; i < p; i++)
{
    for (int j = 0; j < r; j++)
    {
        need[i][j] = max[i][j];
```

```
        }
    }

    printdata(available, max, allocation, need);

    int count = 0;
    while (count < p)
    {
    again:
        int p_num;
        vector<int> req(r);

        cout << "Enter allocation request for process: ";
        cin >> p_num;
        for (int i = 0; i < r; i++)
        {
            cin >> req[i];
        }

        for (int i = 0; i < r; i++)
        {
            if (req[i] > need[p_num][i])
            {
                cout << "Error! Request is greater than need" << endl;
                goto again;
            }
            else if (req[i] > available[i])
            {
                cout << "Error! Request <= need but greater than
available resources" << endl;
                goto again;
            }
        }

        printf("Request is safe\n");

        printf("Now checking if the system is in safe state or not
after granting the request\n");

        // declaring duplicate vectors to check if the system is in
safe state or not

        vector<int> dup_available(available.begin(), available.end());
```

```
        vector<vector<int>> dup_allocation(allocation.begin(),
allocation.end());

        vector<vector<int>> dup_need(need.begin(), need.end());

        // granting the request

        for (int i = 0; i < r; i++)
        {
            dup_available[i] -= req[i];
            dup_allocation[p_num][i] += req[i];
            dup_need[p_num][i] -= req[i];
        }

        vector<int> work(r);

        work = dup_available;

        vector<bool> finish(p, false);
        vector<int> safe_seq(p);
        int ind = 0;
        bool flag = false;
        while (true)
        {

            for (int i = 0; i < p; i++)
            {
                if (finish[i] == false)
                {
                    bool canfinish = true;
                    for (int j = 0; j < r; j++)
                    {
                        if (dup_need[i][j] > work[j])
                        {
                            canfinish = false;
                            break;
                        }
                    }

                    if (canfinish == false)
                        continue;
```

```
                for (int j = 0; j < r; j++)
                {
                    work[j] += dup_allocation[i][j];
                }

                finish[i] = true;
                safe_seq[ind++] = i;
                flag = true;
            }
        }

        if (flag == false)
            break;

        flag = false;
    }

    for (int i = 0; i < p; i++)
    {
        if (finish[i] == false)
        {
            cout << "System is not in safe state" << endl;
            cout << "Request cannot be granted" << endl;
            goto again;
        }
    }

    cout << "System is in safe state" << endl;

    cout << "Safe sequence is: ";
    for (int i = 0; i < p - 1; i++)
    {
        cout << safe_seq[i] << " -> ";
    }
    cout << safe_seq[p - 1] << endl;

    cout << "Request granted" << endl;

    // updating the original vectors

    for (int i = 0; i < r; i++)
    {
        available[i] -= req[i];
```

```
            allocation[p_num][i] += req[i];
            need[p_num][i] -= req[i];
        }

        // checking if need of a process is zero

        int sum = 0;

        for (int i = 0; i < r; i++)
        {
            sum += need[p_num][i];
        }

        if (sum == 0)
        {
            cout << "Process " << p_num << " has finished" << endl;

            // freeing allocated resources

            for (int i = 0; i < r; i++)
            {
                available[i] += allocation[p_num][i];
                allocation[p_num][i] = 0;
            }
            count++;
        }

        printdata(available, max, allocation, need);
    }

    cout << "All Processes have finished" << endl;

    return 0;
}
```

OUTPUT :

Process	Allocation	Max	Need	Available
P0	0 0 0 0	1 1 1 1	1 1 1 1	2 4 5 3
P1	0 0 0 0	2 3 1 2	2 3 1 2	
P2	0 0 0 0	2 2 1 3	2 2 1 3	

```
Enter allocation request for process: 0 1 0 1 1
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 1 -> 2
Request granted
Process      Allocation    Max        Need       Available
P0           1 0 1 1       1 1 1 1    0 1 0 0    1 4 4 2
P1           0 0 0 0       2 3 1 2    2 3 1 2
P2           0 0 0 0       2 2 1 3    2 2 1 3
Enter allocation request for process: 2 1 2 1 2
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 2 -> 1
Request granted
Process      Allocation    Max        Need       Available
P0           1 0 1 1       1 1 1 1    0 1 0 0    0 2 3 0
P1           0 0 0 0       2 3 1 2    2 3 1 2
P2           1 2 1 2       2 2 1 3    1 0 0 1
Enter allocation request for process: 1 0 1 0 1
Error! Request <= need but greater than available resources
Enter allocation request for process: 0 0 1 0 0
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 2 -> 1
Request granted
Process 0 has finished
Process      Allocation    Max        Need       Available
P0           0 0 0 0       1 1 1 1    0 0 0 0    1 2 4 1
P1           0 0 0 0       2 3 1 2    2 3 1 2
P2           1 2 1 2       2 2 1 3    1 0 0 1
Enter allocation request for process: 1 0 1 0 1
Now checking if the system is in safe state or not after granting the request
System is not in safe state
Request cannot be granted
Enter allocation request for process: 1 0 1 0 0
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 2 -> 1
Request granted
Process      Allocation    Max        Need       Available
```

P0	0 0 0 0	1 1 1 1	0 0 0 0	1 1 4 1
P1	0 1 0 0	2 3 1 2	2 2 1 2	
P2	1 2 1 2	2 2 1 3	1 0 0 1	

Enter allocation request for process: 2 1 0 0 1
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 2 -> 1
Request granted
Process 2 has finished

Process	Allocation	Max	Need	Available
P0	0 0 0 0	1 1 1 1	0 0 0 0	2 3 5 3
P1	0 1 0 0	2 3 1 2	2 2 1 2	
P2	0 0 0 0	2 2 1 3	0 0 0 0	

Enter allocation request for process: 1 2 2 1 2
Now checking if the system is in safe state or not after granting the request
System is in safe state
Safe sequence is: 0 -> 1 -> 2
Request granted
Process 1 has finished

Process	Allocation	Max	Need	Available
P0	0 0 0 0	1 1 1 1	0 0 0 0	2 4 5 3
P1	0 0 0 0	2 3 1 2	0 0 0 0	
P2	0 0 0 0	2 2 1 3	0 0 0 0	

All Processes have finished