# National Institute 0f Technology, Hamirpur

## Department Of Computer Science and Engineering
## July-Dec 2020


| | |
|---|---|
| **Subject Name:**<br>Advanced O.S Lab | **Subject Code:**<br>CSD – 416 |
| **Course:**<br>Advanced O.S | **Semester:**<br>4th Year, 7th Semester |
| **Submitted By:**<br><br>Student Name: Rishabh Katna<br>Roll No: 17MI552 | **Submitted To:**<br><br>Dr. Siddhartha Chauhan |
| **Faculty Signature:** | |

# Experiment 1

## Aim : To simulate various job scheduling algorithms.

## Theory :

## Job Scheduling algorithms:

These are the algorithms used by the CPU to schedule various job in efficient ways to achieve maximum performance and CPU utilization.

### The Purpose of a Scheduling algorithm

1.Maximum CPU utilization

2.Fare allocation of CPU

3.Maximum throughput

4.Minimum turnaround time

5.Minimum waiting time

6.Minimum response time

Various job scheduling algorithms are used to achieve this purpose. Some of them are :

a. fcfs scheduling ( First-Come-First-Serve Algorithm ).

b. sjf scheduling ( Shortest Job First Scheduling )

c. round robin scheduling

d. priority scheduling

### The main initializer code used to setup the program is given below.

### Initializer Code:

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

vector<string> namesOfProcesses = {"P1", "P2", "P3", "P4", "P5"};
vector<int> arrivalTimes = {};
vector<int> burstTimes = {};
vector<int> priorities = {2, 3, 4, 5, 5};


const int INFINITY = 100;
const int TIME_QUANTUM = 2;
```

```
class Job {
public:
    string name;
    bool isServed = false, isPresentInReadyQueue = false;
    int arrivalTime, burstTime, waitingTime, completionTime, turnAroundTime,
originalBurstTime, priority;

    Job(string name, int arrivalTime, int burstTime, int priority){
        this->name = name;
        this->arrivalTime = arrivalTime;
        this->burstTime = burstTime;
        this->waitingTime  = 0;
        this->completionTime = 0;
        this->turnAroundTime = 0;
        this->isServed = false;
        this->originalBurstTime = this->burstTime;
        this->priority = priority;
    }

    void printStats(){
        cout
        << setw(20) << this->name
        << setw(5) << this->arrivalTime
        << setw(5) << this->originalBurstTime
        << setw(5) << this->waitingTime
        << setw(5) << this->completionTime
        << setw(5) << this->turnAroundTime
        << endl;
    }
};

void insertJobsInJobQueue(vector<Job> &jobQueue)
{
    int numberOfJobs = namesOfProcesses.size();

    generateRandomArrivalTimes();

    generateRandomBurstTimes();

    for(int i=0; i<numberOfJobs; i++){
        string nameOfCurJob = namesOfProcesses[i];
        int arrivalTimeOfCurJob = arrivalTimes[i];
        int burstTimeOfCurJob = burstTimes[i];
        int priorityOfCurJob = priorities[i];

        Job newJob(nameOfCurJob, arrivalTimeOfCurJob, burstTimeOfCurJob,
priorityOfCurJob);

        jobQueue.push_back(newJob);
    }


}

void insertJobsInReadyQueue(queue<Job*> &readyQueue, vector<Job> &jobQueue, int
timer){

    // check for all the jobs which are arrived before timer
    // passed in original jobQueue
    for(auto &job : jobQueue){
        // check if job has arrived and has not been isisServed yed
        if(job.arrivalTime <= timer && job.isServed == false &&
```

```
job.isPresentInReadyQueue == false){
            Job* jobptr = &job;
            // push address of job in the ready queue
            jobptr->isPresentInReadyQueue = true;
            readyQueue.push(jobptr);
        }
    }

}



void printFinalInformation(vector<Job> &finishedJobs, int totalWaitingTime, int
totalTurnAroundTime){

    int totalNumberOfJobs = namesOfProcesses.size();
    double averageWaitingTime = (1.0 * totalWaitingTime) / (totalNumberOfJobs);
    double averageTurnAroundTime = (1.0 * totalTurnAroundTime) /
(totalNumberOfJobs);
    cout
    << setw(20) << "Name"
    << setw(5) << "A.T"
    << setw(5) << "B.T"
    << setw(5) << "W.T"
    << setw(5) << "C.T"
    << setw(5) << "T.A.T"
    << endl;



    for(auto job : finishedJobs) job.printStats();
    printf("Average turn around time in the system = %.2lf\n",
averageTurnAroundTime);
    printf("Average waiting time in the system = %.2lf\n", averageWaitingTime);
    cout <<
"......................................................................." <<
endl;

}


////////////////// generate random arrival times ////////////////////
void generateRandomArrivalTimes()
{
    // change the arrival times
    int numberOfProcesses = namesOfProcesses.size();

    arrivalTimes.resize(numberOfProcesses);


    for(int i=0; i < numberOfProcesses; i++){
        int randomArrivalTime = 0;

        // generate random arrival time
        randomArrivalTime = (rand() % 19);

        arrivalTimes[i] = randomArrivalTime;
    }

}


///////////////// generate random burst times ////////////////////////
```

```
void generateRandomBurstTimes()
{
    // change the arrival times
    int numberOfProcesses = namesOfProcesses.size();

    burstTimes.resize(numberOfProcesses);


    for(int i=0; i < numberOfProcesses; i++){
        int randomBurstTime = 0;

        // generate random burst time
        randomBurstTime = (rand() % 10);

        burstTimes[i] = randomBurstTime;
    }

}
```

## a. F.C.F.S algorithm :

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

## Code :

```
class FcfsScheduler{
    void serveJob(Job &jobToServe, int timer){

        jobToServe.waitingTime = (jobToServe.arrivalTime > timer) ? 0 : (timer
- jobToServe.arrivalTime);

        jobToServe.completionTime = jobToServe.arrivalTime +
jobToServe.waitingTime + jobToServe.burstTime;

        jobToServe.turnAroundTime = jobToServe.completionTime -
jobToServe.arrivalTime;




    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
            int timer = 0;

    // stores the finished jobs
            vector<Job> finishedJobs;

            // calculate the total waiting time of the processes
            int totalWaitingTime = 0, totalTurnAroundTime = 0;

            // process jobs while there are jobs available in ready queue
            // delay of counter time
            while(!jobQueue.empty()){
                // get the first job from the readyQueue
```

```
                Job jobToServe = jobQueue[0]; jobQueue.erase(jobQueue.begin());

                // get the time at which job will leave the system
                serveJob(jobToServe, timer);

                totalTurnAroundTime += jobToServe.turnAroundTime;


                totalWaitingTime += jobToServe.waitingTime;

                // book a new time slot for the new job
                timer = jobToServe.completionTime;

                // since the job is completed, hence we can add the job in the
finishedJobs array
                finishedJobs.push_back(jobToServe);
            }

            printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }



};
```

## b. SJF scheduling

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

```
class SjfScheduler {
    protected:
        const int inf = 100;
        vector<Job> finishedJobs = {};
        int totalWaitingTime = 0, totalTurnAroundTime = 0;
        void insertJobsInReadyQueue(vector<Job*> &readyQueue, vector<Job>
&jobQueue, int timer){
            for(auto &job : jobQueue){
                // check if job has arrived and has not been served yed
                if(job.arrivalTime <= timer && job.isServed == false){
                    Job* jobptr = &job;
                    // push address of job in the ready queue
                    readyQueue.push_back(jobptr);
                }
            }

        }

        static bool onBurstTime(Job* j1, Job* j2){
            return j1->burstTime <= j2->burstTime;
        }
};
```

## i. SJF Preemptive scheduling

It is the preemptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution.

```
class SjfPreemptiveScheduler : public SjfScheduler{
    void serveJob(Job* jobToServe, int timer){
        jobToServe->isServed = true;
        jobToServe->completionTime = timer + 1;
        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe-
>arrivalTime;
        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe-
>originalBurstTime;
        // jobToServe->responseTime = jobToServe->waitingTime;
    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
           int timer = 0;
           while(timer < inf){
                // address of all job objects
                vector<Job*> readyQueue;

                // this code is wrong
                insertJobsInReadyQueue(readyQueue, jobQueue, timer);

                if(readyQueue.empty()){
                    timer++;
                    continue;
                }

                sort(readyQueue.begin(), readyQueue.end(), onBurstTime);

                Job* jobToServe = readyQueue[0]; // address of the job to serve

                jobToServe->burstTime -= 1;

                if(jobToServe->burstTime == 0) {
                    serveJob(jobToServe, timer);
                    totalWaitingTime += jobToServe->waitingTime;
                    totalTurnAroundTime += jobToServe->turnAroundTime;


                    finishedJobs.push_back(*jobToServe);
                }

                timer ++;
            }
            printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }
 };
```

## ii. SJF Non-Preemptive scheduling

he job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

```
class SjfNonPreemptiveScheduler : public SjfScheduler{
    void serveJob(Job* jobToServe, int timer)
    {
        jobToServe->isServed = true;

        jobToServe->waitingTime = timer - jobToServe->arrivalTime;

        jobToServe->completionTime = timer + jobToServe->burstTime;

        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe-
>arrivalTime;
    }
    public:
        void scheduleJobs(vector<Job> jobQueue){
            int timer = 0;
            while(timer < inf){
                // address of all job objects
                vector<Job*> readyQueue;

                insertJobsInReadyQueue(readyQueue, jobQueue, timer);

                // if no job has arrived, then don't do anything
                if(readyQueue.empty()){
                    timer++;
                    continue;
                }

                sort(readyQueue.begin(), readyQueue.end(), onBurstTime);

                Job* jobToServe = readyQueue[0]; // address of the job to serve

                serveJob(jobToServe, timer);

                timer = jobToServe->completionTime;


                totalWaitingTime += jobToServe->waitingTime;

                totalTurnAroundTime += jobToServe->turnAroundTime;

                finishedJobs.push_back(*jobToServe);

            }

            printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }



};
```

## c. Prioriy Scheduler

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is same then they will be scheduled according to their arrival time.

```
class ComparyPriorities
{
    /* data */
    public:
        bool operator () (Job* j1, Job* j2){
            return j1->priority < j2->priority;
        }
};



class PriorityScheduler {
    vector<Job> finishedJobs = {};

    void insertJobsInReadyQueue(priority_queue<Job*, vector<Job*>,
ComparyPriorities> &readyQueue, vector<Job> &jobQueue, int timer){

    // check for all the jobs which are arrived before timer
    // passed in original jobQueue
        for(auto &job : jobQueue){
            // check if job has arrived and has not been isisServed yed
            if(job.arrivalTime <= timer && job.isServed == false &&
job.isPresentInReadyQueue == false){
                Job* jobptr = &job;
                // push address of job in the ready queue
                jobptr->isPresentInReadyQueue = true;
                readyQueue.push(jobptr);
            }
        }

    }

    void serveJob(Job* jobToServe, int timer){

        jobToServe->isServed = true;

        jobToServe->completionTime = timer;

        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe-
>arrivalTime;

        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe-
>originalBurstTime;
    }


    public:

        void scheduleJobs(vector<Job> jobQueue){

            int timer = 0;

            priority_queue<Job*, vector<Job*>, ComparyPriorities> readyQueue;
// when something is entered, it gets to its appr location

            int totalWaitingTime = 0, totalTurnAroundTime = 0;
```

```
            while(timer < INFINITY){

                // insert jobs in ready queue
                insertJobsInReadyQueue(readyQueue, jobQueue, timer);


                if(readyQueue.empty()) {
                    timer++;
                    continue;
                }




                // get the best job
                Job* jobToServe = readyQueue.top();


                jobToServe->burstTime--;


                timer++;



                if(jobToServe->burstTime == 0){
                    // done processing .... no need for more cpu allocation
                    // out of our readyQueue
                    readyQueue.pop();

                    serveJob(jobToServe, timer);

                    totalWaitingTime += jobToServe->waitingTime;

                    totalTurnAroundTime += jobToServe->turnAroundTime;

                    finishedJobs.push_back(*jobToServe);
                }


            }



            printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);
        }


};
```

## d. Round-Robin Scheduler :

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will get executed in the cyclic way. Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a preemptive type of scheduling.

```
class RoundRobinScheduler {
    static bool onArrivalTime(Job j1, Job j2){
        return j1.arrivalTime < j2.arrivalTime;
    }



    void serveJob(Job* jobToServe, int timer){
        jobToServe->isServed = true;
            // mark isisServed as true
        // these are calculated accurately
        jobToServe->completionTime = timer;
    // calculate waiting time, execution time and tat
        jobToServe->turnAroundTime = jobToServe->completionTime - jobToServe-
>arrivalTime;

        jobToServe->waitingTime = jobToServe->turnAroundTime - jobToServe-
>originalBurstTime;
    }
public:



    void scheduleJobs(vector<Job> jobQueue) {

        sort(jobQueue.begin(), jobQueue.end(), onArrivalTime);


        vector<Job> finishedJobs = {};
        int timer = 0;

        // stores the addresses of the jobs

        queue<Job*> readyQueue;

        // store the job processes in previous step
        Job* prevJobProcessed = NULL;


        int totalWaitingTime = 0;
        int totalTurnAroundTime = 0;

        while(timer < INFINITY){

            insertJobsInReadyQueue(readyQueue, jobQueue, timer);

            // insert the prevJobProcesses
            if(prevJobProcessed != NULL) readyQueue.push(prevJobProcessed);

            prevJobProcessed = NULL;

            if(readyQueue.empty()){
                timer += 1;
                continue;
            }
```

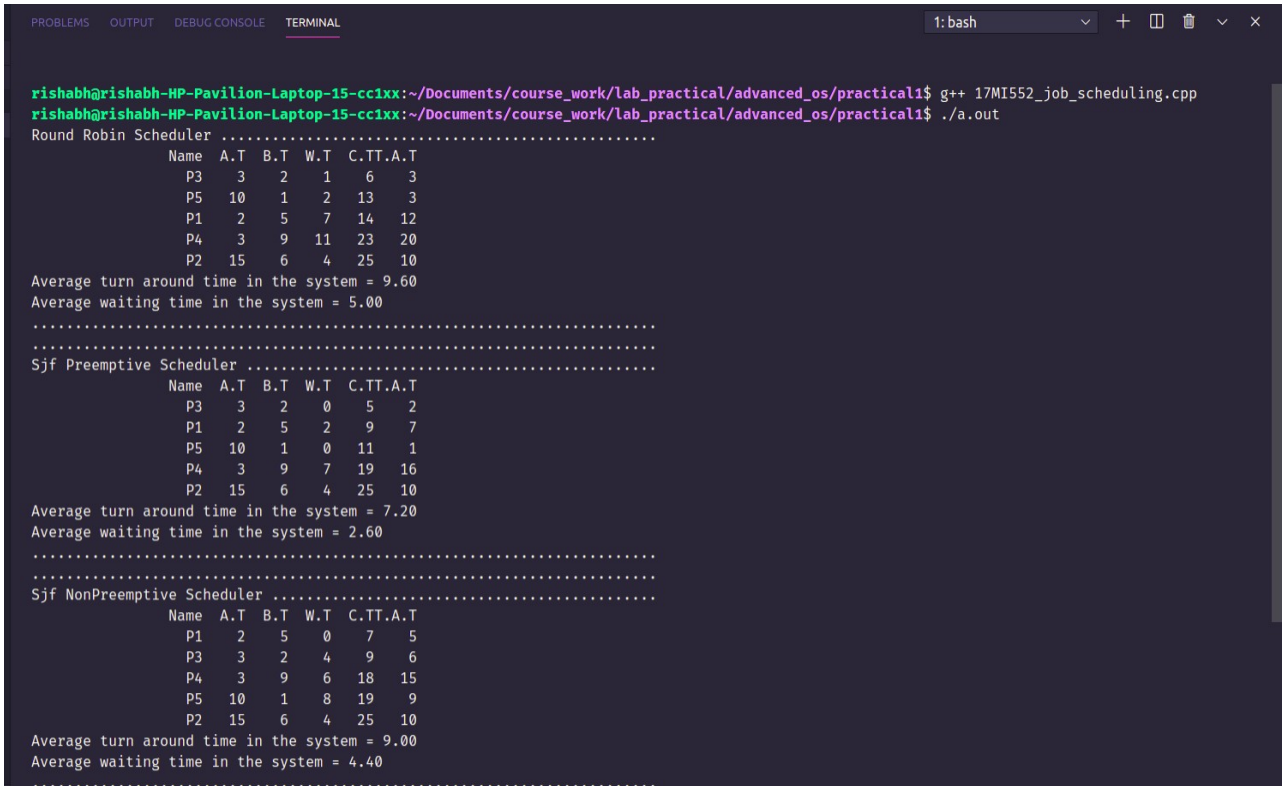
```
            /// if some jobs have arrived

            // take first job of the queue
            Job* jobToServe = readyQueue.front(); readyQueue.pop();
            // process the job for interval

            int processingInterval = min(TIME_QUANTUM, jobToServe->burstTime);

            jobToServe->burstTime -= processingInterval;

            timer += processingInterval;

            // check if job has done its processing

            // job will not be pushed in ready queue again
            if(jobToServe->burstTime != 0){
                prevJobProcessed = jobToServe;
                continue;
            }



            // burstTime == 0
            serveJob(jobToServe, timer);

            // cout << jobToServe->name << ' ' << jobToServe->burstTime <<
endl;
            totalWaitingTime += jobToServe->waitingTime;

            totalTurnAroundTime += jobToServe->turnAroundTime;

            finishedJobs.push_back(*jobToServe);

        }

        printFinalInformation(finishedJobs, totalWaitingTime,
totalTurnAroundTime);

    }




};
```

## Output :

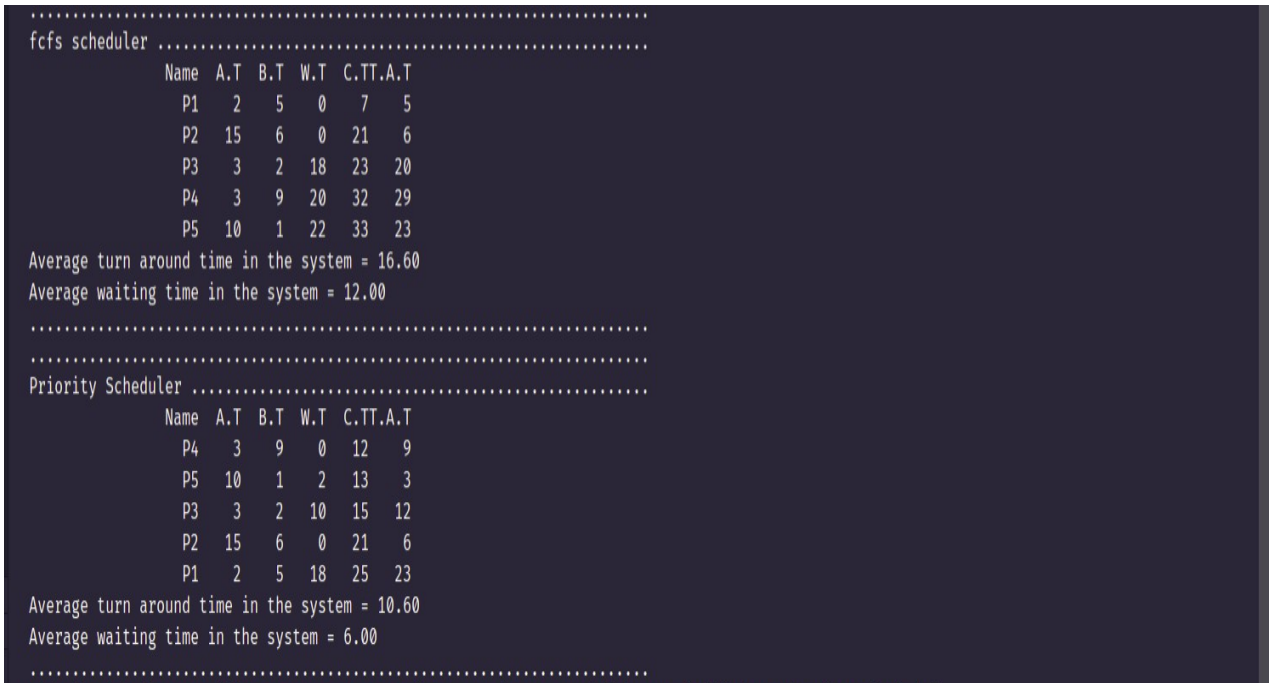
## driver Code :

```
int main()
{
    vector<Job> jobQueue = {};

    insertJobsInJobQueue(jobQueue);

    // start the processing

    cout << "Round Robin
Scheduler ....................................................." << endl;
    RoundRobinScheduler roundRobinScheduler;
    roundRobinScheduler.scheduleJobs(jobQueue);
    cout <<
"....................................................................." <<
endl;

    cout << "Sjf Preemptive
Scheduler ..............................................." << endl;
    SjfPreemptiveScheduler sjfPreemptiveScheduler;
    sjfPreemptiveScheduler.scheduleJobs(jobQueue);
    cout <<
"....................................................................." <<
endl;

    cout << "Sjf NonPreemptive
Scheduler ..............................................." << endl;
    SjfNonPreemptiveScheduler sjfNonPreemptiveScheduler;
    sjfNonPreemptiveScheduler.scheduleJobs(jobQueue);

    cout << "fcfs
scheduler ....................................................." << endl;
    FcfsScheduler fcfsScheduler;
    fcfsScheduler.scheduleJobs(jobQueue);
    cout <<
"....................................................................." <<
endl;

    cout << "Priority
Scheduler ....................................................." << endl;
    PriorityScheduler priorityScheduler;
    priorityScheduler.scheduleJobs(jobQueue);


}
```

## Output :



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                  1: bash

rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/advanced_os/practical1$ g++ 17MI552_job_scheduling.cpp
rishabh@rishabh-HP-Pavilion-Laptop-15-cc1xx:~/Documents/course_work/lab_practical/advanced_os/practical1$ ./a.out
Round Robin Scheduler ..................................................
                Name  A.T  B.T  W.T  C.TT.A.T
                  P3    3    2    1    6    3
                  P5   10    1    2   13    3
                  P1    2    5    7   14   12
                  P4    3    9   11   23   20
                  P2   15    6    4   25   10
Average turn around time in the system = 9.60
Average waiting time in the system = 5.00
.....................................................................
.....................................................................
Sjf Preemptive Scheduler ..............................................
                Name  A.T  B.T  W.T  C.TT.A.T
                  P3    3    2    0    5    2
                  P1    2    5    2    9    7
                  P5   10    1    0   11    1
                  P4    3    9    7   19   16
                  P2   15    6    4   25   10
Average turn around time in the system = 7.20
Average waiting time in the system = 2.60
.....................................................................
.....................................................................
Sjf NonPreemptive Scheduler ...........................................
                Name  A.T  B.T  W.T  C.TT.A.T
                  P1    2    5    0    7    5
                  P3    3    2    4    9    6
                  P4    3    9    6   18   15
                  P5   10    1    8   19    9
                  P2   15    6    4   25   10
Average turn around time in the system = 9.00
Average waiting time in the system = 4.40
.....................................................................
```



```
.....................................................................
fcfs scheduler ........................................................
                Name  A.T  B.T  W.T  C.TT.A.T
                  P1    2    5    0    7    5
                  P2   15    6    0   21    6
                  P3    3    2   18   23   20
                  P4    3    9   20   32   29
                  P5   10    1   22   33   23
Average turn around time in the system = 16.60
Average waiting time in the system = 12.00
.....................................................................
.....................................................................
Priority Scheduler ....................................................
                Name  A.T  B.T  W.T  C.TT.A.T
                  P4    3    9    0   12    9
                  P5   10    1    2   13    3
                  P3    3    2   10   15   12
                  P2   15    6    0   21    6
                  P1    2    5   18   25   23
Average turn around time in the system = 10.60
Average waiting time in the system = 6.00
.....................................................................
```