OS Lab Assignment 4

Submitted By:

Manroop Parmar

101906134 3EC6

1. Priority Scheduling Algorithm

Non-Preemptive

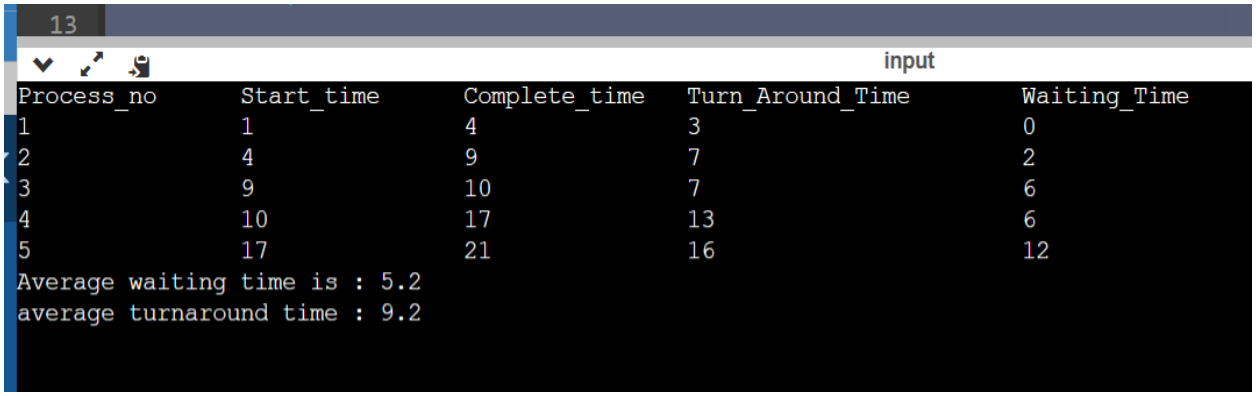
```
#include <bits/stdc++.h>
using namespace std;
#define totalprocess 5

struct process
{
int at,bt,pr,pno;
};

process proc[50];
bool comp(process a,process b)
{
if(a.at == b.at)
{
return a.pr<b.pr;
}
else
```

```
{
    return a.at<b.at;
}
}
void get_wt_time(int wt[])
{
int service[50];
service[0] = proc[0].at;
wt[0]=0;


for(int i=1;i<totalprocess;i++)
{
service[i]=proc[i-1].bt+service[i-1];

wt[i]=service[i]-proc[i].at;
    if(wt[i]<0)
    {
    wt[i]=0;
    }
}

}

void get_tat_time(int tat[],int wt[])
{
// Filling turnaroundtime array
```

```
for(int i=0;i<totalprocess;i++)
{
    tat[i]=proc[i].bt+wt[i];
}

}

void findgc()
{
//Declare waiting time and turnaround time array
int wt[50],tat[50];

double wavg=0,tavg=0;

// Function call to find waiting time array
get_wt_time(wt);
//Function call to find turnaround time
get_tat_time(tat,wt);

int stime[50],ctime[50];

stime[0] = proc[0].at;
ctime[0]=stime[0]+tat[0];

// calculating starting and ending time
for(int i=1;i<totalprocess;i++)
```

```
    {
            stime[i]=ctime[i-1];
            ctime[i]=stime[i]+tat[i]-wt[i];
    }

cout<<"Process_no\tStart_time\tComplete_time\tTurn_Around_Time\tWaiting_Time"<<endl;

    // display the process details

for(int i=0;i<totalprocess;i++)
    {
            wavg += wt[i];
            tavg += tat[i];

            cout<<proc[i].pno<<"\t\t"<<
                    stime[i]<<"\t\t"<<ctime[i]<<"\t\t"<<
                    tat[i]<<"\t\t\t"<<wt[i]<<endl;
    }

            // display the average waiting time
            //and average turn around time

    cout<<"Average waiting time is : ";
    cout<<wavg/(float)totalprocess<<endl;
    cout<<"average turnaround time : ";
    cout<<tavg/(float)totalprocess<<endl;
```

```
}

int main()
{
int arrivaltime[] = { 1, 2, 3, 4, 5 };
int bursttime[] = { 3, 5, 1, 7, 4 };
int priority[] = { 3, 4, 1, 7, 8 };

for(int i=0;i<totalprocess;i++)
{
    proc[i].at=arrivaltime[i];
    proc[i].bt=bursttime[i];
    proc[i].pr=priority[i];
    proc[i].pno=i+1;
    }

    //Using inbuilt sort function

    sort(proc,proc+totalprocess,comp);

    //Calling function findgc for finding Gantt Chart

    findgc();

    return 0;
}
```



Pre-emptive

```
// CPP program to implement preemptive priority scheduling
#include <bits/stdc++.h>
using namespace std;

struct Process {
    int processID;
    int burstTime;
    int tempburstTime;
    int responsetime;
    int arrivalTime;
    int priority;
    int outtime;
    int intime;
};
void insert(Process Heap[], Process value, int* heapsize,
                        int* currentTime)
{
```

```
    int start = *heapsize, i;
    Heap[*heapsize] = value;
    if (Heap[*heapsize].intime == -1)
            Heap[*heapsize].intime = *currentTime;
    ++(*heapsize);

    // Ordering the Heap
    while (start != 0 && Heap[(start - 1) / 2].priority >Heap[start].priority) {
            Process temp = Heap[(start - 1) / 2];
            Heap[(start - 1) / 2] = Heap[start];
            Heap[start] = temp;
            start = (start - 1) / 2;
    }
}
void order(Process Heap[], int* heapsize, int start)
{
    int smallest = start;
    int left = 2 * start + 1;
    int right = 2 * start + 2;
    if (left < *heapsize && Heap[left].priority <Heap[smallest].priority)
            smallest = left;
    if (right < *heapsize && Heap[right].priority <Heap[smallest].priority)
            smallest = right;

    // Ordering the Heap
    if (smallest != start) {
            Process temp = Heap[smallest];
```

```
            Heap[smallest] = Heap[start];
            Heap[start] = temp;
            order(Heap, heapsize, smallest);
    }
}
Process extractminimum(Process Heap[], int* heapsize,
                                        int* currentTime)
{
    Process min = Heap[0];
    if (min.responsetime == -1)
            min.responsetime = *currentTime - min.arrivalTime;
    --(*heapsize);
    if (*heapsize >= 1) {
            Heap[0] = Heap[*heapsize];
            order(Heap, heapsize, 0);
    }
    return min;
}

// Compares two intervals according to starting times.
bool compare(Process p1, Process p2)
{
    return (p1.arrivalTime < p2.arrivalTime);
}
void scheduling(Process Heap[], Process array[], int n, int* heapsize, int* currentTime)
{
    if (heapsize == 0)
```

```
            return;

        Process min = extractminimum(Heap, heapsize, currentTime);
        min.outtime = *currentTime + 1;
        --min.burstTime;
        printf("process id = %d current time = %d\n",
            min.processID, *currentTime);
        if (min.burstTime > 0) {
            insert(Heap, min, heapsize, currentTime);
            return;
        }

        for (int i = 0; i < n; i++)
            if (array[i].processID == min.processID) {
                array[i] = min;
                break;
            }
    }
void priority(Process array[], int n)
    {
        sort(array, array + n, compare);

        int totalwaitingtime = 0, totalbursttime = 0,
            totalturnaroundtime = 0, i, insertedprocess = 0,
            heapsize = 0, currentTime = array[0].arrivalTime,
            totalresponsetime = 0;
```

```
Process Heap[4 * n];

// Calculating the total burst time
// of the processes
for (int i = 0; i < n; i++) {
        totalbursttime += array[i].burstTime;
        array[i].tempburstTime = array[i].burstTime;
}

do {
        if (insertedprocess != n) {
                for (i = 0; i < n; i++) {
                        if (array[i].arrivalTime == currentTime) {
                                ++insertedprocess;
                                array[i].intime = -1;
                                array[i].responsetime = -1;
                                insert(Heap, array[i], &heapsize, ¤tTime);
                        }
                }
        }
        scheduling(Heap, array, n, &heapsize, ¤tTime);
        ++currentTime;
        if (heapsize == 0 && insertedprocess == n)
                break;
} while (1);

for (int i = 0; i < n; i++) {
```

```
            totalresponsetime += array[i].responsetime;
            totalwaitingtime += (array[i].outtime - array[i].intime -
    array[i].tempburstTime);
            totalbursttime += array[i].burstTime;
    }
    printf("Average waiting time = %f\n",
            ((float)totalwaitingtime / (float)n));
    printf("Average response time =%f\n",
            ((float)totalresponsetime / (float)n));
    printf("Average turn around time = %f\n",
            ((float)(totalwaitingtime + totalbursttime) / (float)n));
}

// Driver code
int main()
{
    int n, i;
    Process a[5];
    a[0].processID = 1;
    a[0].arrivalTime = 4;
    a[0].priority = 2;
    a[0].burstTime = 6;
    a[1].processID = 4;
    a[1].arrivalTime = 5;
    a[1].priority = 1;
    a[1].burstTime = 3;
```

```
    a[2].processID = 2;
    a[2].arrivalTime = 5;
    a[2].priority = 3;
    a[2].burstTime = 1;
    a[3].processID = 3;
    a[3].arrivalTime = 1;
    a[3].priority = 4;
    a[3].burstTime = 2;
    a[4].processID = 5;
    a[4].arrivalTime = 3;
    a[4].priority = 5;
    a[4].burstTime = 4;
    priority(a, 5);
    return 0;
}
```

```
process id = 3 current time = 1
process id = 3 current time = 2
process id = 5 current time = 3
process id = 1 current time = 4
process id = 4 current time = 5
process id = 4 current time = 6
process id = 4 current time = 7
process id = 1 current time = 8
process id = 1 current time = 9
process id = 1 current time = 10
process id = 1 current time = 11
process id = 1 current time = 12
process id = 2 current time = 13
process id = 5 current time = 14
process id = 5 current time = 15
process id = 5 current time = 16
Average waiting time = 4.400000
Average response time =1.600000
Average turn around time = 7.200000
```

2. Round Robin

```
// C++ program for implementation of RR scheduling
#include<iostream>
using namespace std;

// Function to find the waiting time for all
// processes
```

```
void findWaitingTime(int processes[], int n,
                        int bt[], int wt[], int quantum)
{
        // Make a copy of burst times bt[] to store remaining
        // burst times.
        int rem_bt[n];
        for (int i = 0 ; i < n ; i++)
                rem_bt[i] = bt[i];

        int t = 0; // Current time

        // Keep traversing processes in round robin manner
        // until all of them are not done.
        while (1)
        {
                bool done = true;

                // Traverse all processes one by one repeatedly
                for (int i = 0 ; i < n; i++)
                {
                        // If burst time of a process is greater than 0
                        // then only need to process further
                        if (rem_bt[i] > 0)
                        {
                                done = false; // There is a pending process

                                if (rem_bt[i] > quantum)
```

```
        {
            // Increase the value of t i.e. shows
            // how much time a process has been processed
            t += quantum;

            // Decrease the burst_time of current process
            // by quantum
            rem_bt[i] -= quantum;
        }

        // If burst time is smaller than or equal to
        // quantum. Last cycle for this process
        else
        {
            // Increase the value of t i.e. shows
            // how much time a process has been processed
            t = t + rem_bt[i];

            // Waiting time is current time minus time
            // used by this process
            wt[i] = t - bt[i];

            // As the process gets fully executed
            // make its remaining burst time = 0
            rem_bt[i] = 0;
        }
    }
```

```
                }

                // If all processes are done
                if (done == true)
                break;
        }
}

// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,  int bt[], int wt[], int tat[])
{
        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n ; i++)
                tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
void findavgTime(int processes[], int n, int bt[], int quantum)
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;

        // Function to find waiting time of all processes
        findWaitingTime(processes, n, bt, wt, quantum);

        // Function to find turn around time for all processes
        findTurnAroundTime(processes, n, bt, wt, tat);
```

```
	// Display processes along with all details
	cout << "Processes "<< " Burst time "
		<< " Waiting time " << " Turn around time\n";

	// Calculate total waiting time and total turn
	// around time
	for (int i=0; i<n; i++)
	{
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
			<< wt[i] <<"\t\t " << tat[i] <<endl;
	}

	cout << "Average waiting time = "
		<< (float)total_wt / (float)n;
	cout << "\nAverage turn around time = "
		<< (float)total_tat / (float)n;
}

// Driver code
int main()
{
	// process id's
	int processes[] = { 1, 2, 3};
	int n = sizeof processes / sizeof processes[0];
```

```
    // Burst time of all processes
    int burst_time[] = {10, 5, 8};

    // Time quantum
    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```