Operating System Simulation Based Assignment

Name: Vikartan Sood

Registration No.:11606546

Email Address: vikartansood@gmail.com

GitHub Link: https://github.com/vikartansood/operating_system.git

Question Assigned-

Consider a scheduling approach which is non-pre-emptive similar to shortest job next in nature. The priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement. The jobs that have spent a long time waiting compete against those estimated to have short run times.

Solution:

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

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

CODE:

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
int process_id;
float arr_tim, wt_tim, brst_tim, tnarnd_tim, sch_tbl;
bool isComplete;
}process;


void project(int i, process p[])
{
printf("PROCESS ID: ");
scanf("%d", &p[i].process_id);
printf("ARRIVAL TIME: ");
scanf("%f", &p[i].arr_tim);
printf("BURST TIME: ");
scanf("%f", &p[i].brst_tim);
p[i].isComplete = false;
}


void sort(process p[], int i, int start)
{
int k = 0, j;
process temp;
```

```
for (k = start; k<i; k++)                                   //O(1)
{
for (j = k+1; j<i; j++)                                     //O(1)
{
if(p[k].brst_tim < p[j].brst_tim)
continue;
else
{
temp = p[k];
p[k] = p[j];
p[j] = temp;
}
}
}
}
int main()
{
int n, i, k = 0, j = 0;
float avgwt_tim = 0.0, avgtnarnd_tim = 0.0, tst = 0.0;
printf("Enter the number of processes: ");
scanf("%d",&n);
process p[n];
for (i = 0; i<n; i++)                                             //O(1)
{
printf("\nEnter process number %d's details: ",i);
project(i,p);
}
for (i = 0; i<n; i++)                                               //O(1)
```

```
{
if (p[i].isComplete == true)
continue;
else
{
k = i;
while (p[i].arr_tim<=tst && i<n)
i++;
sort (p,i,k);
i = k;
if(p[i].arr_tim<=tst)
p[i].sch_tbl = tst;
else

p[i].sch_tbl = p[i].arr_tim;
p[i].sch_tbl = tst;
p[i].isComplete = true;
tst += p[i].brst_tim;

p[i].wt_tim = p[i].sch_tbl - p[i].arr_tim;
p[i].tnarnd_tim = p[i].brst_tim + p[i].wt_tim;
avgwt_tim += p[i].wt_tim;
avgtnarnd_tim += p[i].tnarnd_tim;
}
}
```

```
avgwt_tim /= n;

avgtnarnd_tim /= n;

printf("PROCESS SCHEDULE TABLE: \n");

printf("\tPROCESS ID \t ARRIVAL TIME \t BURST TIME \t WAIT TIME \t TURNAROUND TIME\n");
for (i = 0; i<n; i++)                                        //O(1)
printf("\t%d\t\t%f\t%f\t%f\t%f\n", p[i].process_id,p[i].arr_tim, p[i].brst_tim, p[i].wt_tim, p[i].tnarnd_tim);

printf("\nAVERAGE WAIT TIME: %f", avgwt_tim);

printf("\nAVERAGE TURNAROUND TIME: %f\n", avgtnarnd_tim);
}
```

Time Complexity:

Total Time complexity in this type of code will be equal to O (n log n)

<u>Advantage</u>: Shorter jobs will get the CPU on priority basis.

<u>Disadvantage</u>: The jobs with larger burst time might not get turn for a long time even when they arrived in the system at an early time period. This problem is called as starvation.

Test case 1:

Process	**Arrival Time**	**Burst Time**
P1	0	20
P2	5	36
P3	13	19
P4	17	42

Test case 2:

Process Id	**Wait Time**	**Turnaround Time**
1	0	20
2	7	26
3	34	70
4	58	100
Average Time:	24.75	54.0

Gantt Chart:

P1	P3	P2	P4	
0	**20**	**39**	**75**	**119**

<u>Algorithm of SJF:</u>

Input (I, Id, AT, CT, TAT, BT, WT)

1) Sort all the processes in increasing order

according to their burst time.

2)Then apply the following algorithm

3)Input the processes along with their burst time (bt).

4)Find waiting time (wt) for all processes.

5)As first process that comes need not to wait so

6)waiting time for process 1 will be 0 i.e. wt [0] = 0

7)Find **waiting time** for all other processes i.e. for

8)process i ->

    wt[i] = bt[i-1] + wt[i-1]

9)Find **turnaround time** = waiting_time + burst_time

for all processes.

10)Find **average waiting time** =

total_waiting_time / no_of_processes

11)Similarly, find **average turnaround time** =

total_turn_around_time / no_of_processes.