**FINAL REPORT**

***OS Project***

As the Field Work for Course

**OPERATING SYSTEM**

**(CSE 316)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **SR.** | **Registration** | **Name of Students** | **Roll no.** | **Total** | **Marks** | **Signature** |  |
| **No** | **No.** | **Marks** | **Obtained** |  |
|  |  |  |  |
|  |  |  |  |  |  |  |  |
| 1 | 11806793 | K HARSHA VARDHAN REDDY | RK18KPA13 | **30** |  |  |  |
|  |  |  |  |  |  |  |  |

**Email Id:** hvreddy777@gmail.com

**Github link:**

**Submitted To Samreen Fayaz mam, Lovely Professional University Jalandhar, Punjab, India.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



**Description:**

Definition of Non-Preemptive Scheduling

Non-preemptive Scheduling is one which can be applied in the circumstances when a process **terminates**, or a process switches from **running** to **waiting state**. In Non-Preemptive Scheduling, once the resources (CPU) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state.

Unlike preemptive scheduling, non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits for the process to complete its CPU burst time and then it can allocate the CPU to another process.

**Example of non Preemptive scheduling are**: shortest remaining time first(SRTF), priority, round robin

**Explanation**

Suppose there are four processes P1, P2, P3 and P4 whose arrival time and burst time are given in the table below:

**Process****Arrival Time****CPU Burst Time**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 |  | 0 |  | 4 |
|  |  | |  | |
| P2 |  | 1 |  | 1 |
|  |  |  |  |  |
| P3 |  | 2 |  | 2 |
|  |  |  |  |  |
| P4 |  | 3 |  | 1 |

When the process starts execution (i.e. CPU assigned), priority for that process changes at the rate of m=1.When the process waits for CPU in the ready queue (but not yet started execution), its priority changes at a rate n=2. All the processes are initially assigned priority value of 0 when they enter ready queue for the first time . The time slice for each process is q = 1. When two processes want to join ready queue simultaneously, the process which has not executed recently is given priority.

**Complexity:**

We discuss the problem of scheduling af set of independent tasks T, each ti ϵ T of lenght ℓi ϵ *Z*+, on m identical processors. We allow preemption but assume a communication delay of time k ϵ *N*. Whenever a task is preempted from one processor to another, there must be a delay of at least k time units. We show that if *k* = 1, an optimal schedule can be found in polynomial time but if k ⩾ 2, the corresponding decision problem is NP-complete.

**WHY Scheduling:**

To make your day more logical and efficient, you work on a schedule. An operating system operates in a similar manner: by scheduling tasks, improving efficiency, reducing delays and wait times (response times to the system), and managing CPU resources better.

This activity is called **process scheduling**. A process is like a job in computer systems that can be executed. Some processes are input/output (I/O) like graphics display process, others are CPU-focused and can be transparent to users. If your computer freezes, sometimes the underlying issue could be that a system process is trying to acquire CPU resources, but those resources are already occupied by other processes. Through process scheduling, the operating system tries to avoid these kind of deadlocks and lockups.

**Criteria for Scheduling:**

There are several criteria for invoking the best scheduling policies for a system:

|  |  |  |  |
| --- | --- | --- | --- |
| **Criteria** |  | **Description** |  |
|  |  |  |  |
|  |  |  |  |
| CPU Utilization |  | Reduces the strain on the CPU and manages the percentage of time the CPU is |  |
|  | busy |  |
|  |  |  |
|  |  |  |  |
| Throughput |  | Increases the number of processes completed in a given time frame |  |
|  |  |  |  |
|  |  |  |  |
| Wait Time |  | Reduces the waiting time of a process |  |
|  |  |  |  |
| Response Time |  | Minimizes the time a user has to wait for a process to run |  |
|  |  |  |  |
|  |  |  |  |
| Turnaround |  | Total time a process takes to run, from start to finish (includes all waiting time) |  |
| Time |  |  |
|  |  |  |
|  |  |  |  |

**Scheduling Policies:**

To fulfill those criteria, a scheduler has to use various policies or strategies:

**1.Fairness**

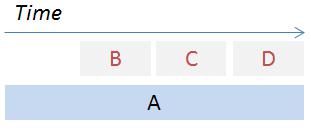
Just as it isn't fair for someone to bring a loaded shopping cart to the 10-items-or-less checkout, the operating system shouldn't give an unfair advantage to a process that will interfere with the criteria we listed (CPU utilization, wait time, throughput). It's important to balance long-running jobs and ensure that the lighter jobs can be run quickly.

Let's take a look at the policies and then do a final comparison that addresses the fairness of each item.

**2. FCFS - First Come First Served**

Also called FIFO (first-in-first-out), **first-come-first-served (FCFS)** processes jobs in the order in which they are received. This is not a very fair policy, because a long-running job could be running, and other processes have to wait for it to finish. To the end-user, this could look like a system freeze or lock-up.

Consider the following example of a long-running process A that now holds other processes B, C and D as hostage.



**3. Round Robin**

This policy works like musical chairs but more methodical. A timer is used to determine when to move the current running process to the back of the line. For example, you set the timer to 10ms; if the printer process isn't done within that time frame, move it to the end of the line and move up the next process.

If the time (10ms) is too long, processes still have to wait longer, which basically puts you back into FCFS territory. However, if the time is too small, you spend more time context switching and hence, the throughput suffers.

## What is Shortest Job First Scheduling?

**4. Non preemptive Shortest Job First (SJF)**

**it** is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

**There are basically two types of SJF methods:**

* Non-preemptive SJF
* Preemptive SJF

**What are the advantages of non preemptive scheduling?**

* It does not have overheads
* Can give good response time
* Can produce very fair usage
* Works well with realtime and priority scheduling

Question

|  |  |  |
| --- | --- | --- |
| process | Arrival time | Burst time |
| P1 | 0 | 20 |
| P2 | 5 | 36 |
| P3 | 13 | 19 |
| P4 | 7 | 42 |

SOLUTION

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **process** | **Arrival time** | **Burst time** | **Completion time** | **Turn around time** | **Waiting time** |
| **P1** | **0** | **20** | **20** | **20** | **0** |
| **P2** | **5** | **36** | **75** | **70** | **34** |
| **P3** | **13** | **19** | **39** | **26** | **7** |
| **P4** | **17** | **42** | **117** | **100** | **58** |

**Gantt chart**

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P3 | P2 | P4 |

**P1=20**

**P3=39**

**P2=75**

**P4=117**

**Average turn around time**

**(20+70+26+100)/4 =54**

**Average waiting time**

**(0+34+7+58)/4 =24.75**

**CODE**

include<stdio.h>

int main()

{

int i,n,p[10]={1,2,3,4,5,6,7,8,9,10},min,k=1,btime=0;

int bt[10],temp,j,at[10],wt[10],tt[10],ta=0,sum=0;

float wavg=0,tavg=0,tsum=0,wsum=0;

printf(" Shortest Job First Scheduling ( NP )\n");

printf("\nEnter the No. of processes :");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("\tEnter the burst time of %d process :",i+1);

scanf(" %d",&bt[i]);

printf("\tEnter the arrival time of %d process :",i+1);

scanf(" %d",&at[i]);

}

/\*Sorting According to Arrival Time\*/

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

if(at[i]<at[j])

{

temp=p[j];

p[j]=p[i];

p[i]=temp;

temp=at[j];

at[j]=at[i];

at[i]=temp;

temp=bt[j];

bt[j]=bt[i];

bt[i]=temp;

}

}

}

/\*Arranging the table according to Burst time,

Execution time and Arrival Time

Arrival time <= Execution time

\*/

for(j=0;j<n;j++)

{

btime=btime+bt[j];

min=bt[k];

for(i=k;i<n;i++)

{

if (btime>=at[i] && bt[i]<min)

{

temp=p[k];

p[k]=p[i];

p[i]=temp;

temp=at[k];

at[k]=at[i];

at[i]=temp;

temp=bt[k];

bt[k]=bt[i];

bt[i]=temp;

}

}

k++;

}

wt[0]=0;

for(i=1;i<n;i++)

{

sum=sum+bt[i-1];

wt[i]=sum-at[i];

wsum=wsum+wt[i];

}

wavg=(wsum/n);

for(i=0;i<n;i++)

{

ta=ta+bt[i];

tt[i]=ta-at[i];

tsum=tsum+tt[i];

}

tavg=(tsum/n);

printf("");

printf("\n RESULT:-");

printf("\nProcess\t Burst\t Arrival\t Waiting\t Turn-around" );

for(i=0;i<n;i++)

{

printf("\n p%d\t %d\t %d\t\t %d\t\t\t%d",p[i],bt[i],at[i],wt[i],tt[i]);

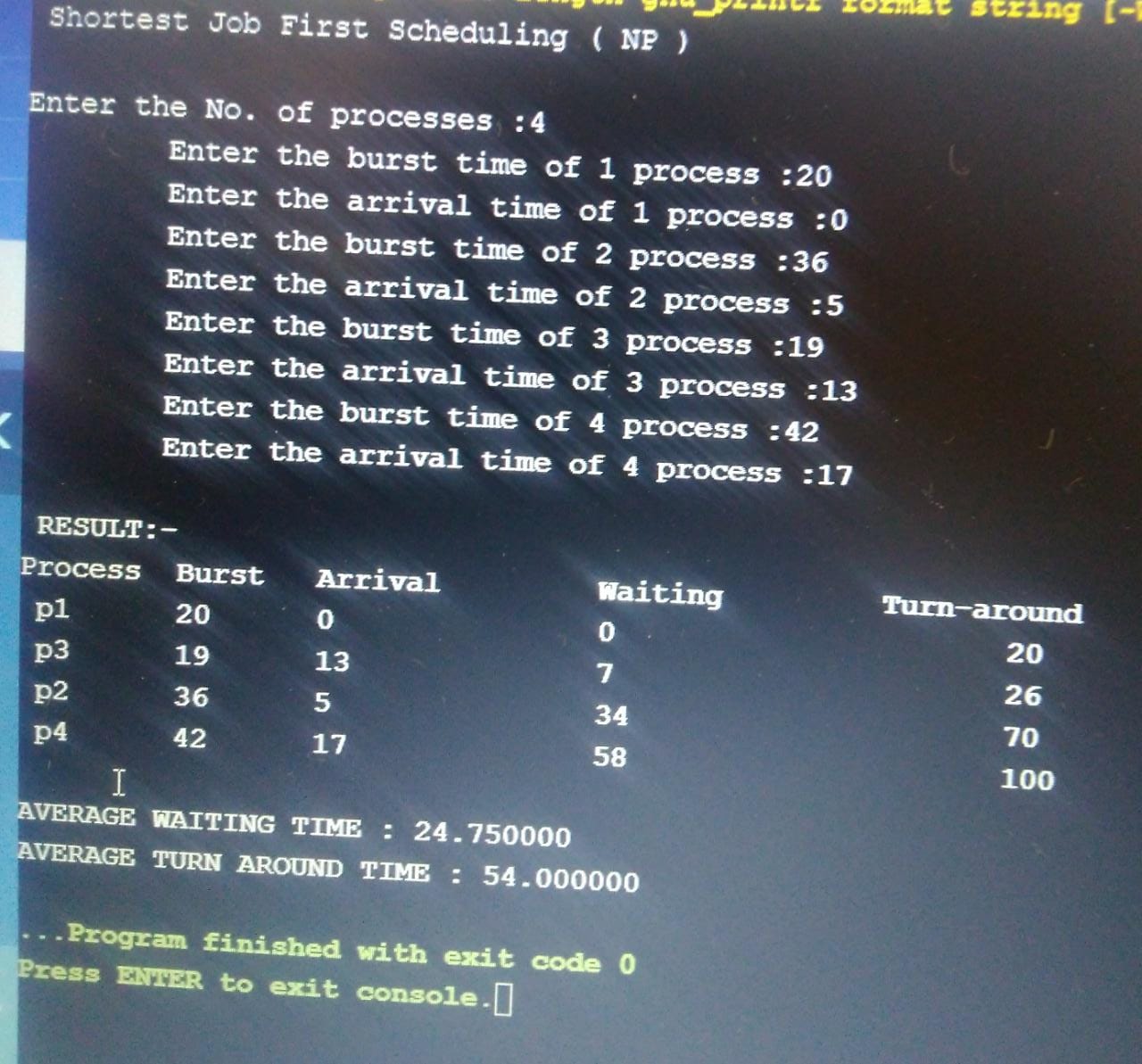
}

printf("\n\nAVERAGE WAITING TIME : %f",wavg);

printf("\nAVERAGE TURN AROUND TIME : %f",tavg);

return 0;

}

outputgithub link: