OPERATING SYSTEM ASSIGNMENT

NAME: **DORA PRAVEEN KUMAR**
ROLLNO: **09**
GITHUBLINK: **https://github.com/DoraPraveenKumar/OS-Project.git**
REGNO: **11810758**
SECTION: **K18ZV**

QUES 1:

Write a program in C which reads input CPU bursts from the first line of a text file named as CPU_BURST. txt. Validate the input numbers whether the numbers are positive integers or not. Consider the numbers as CPU burst. If there are 5 positive integers in the first line of the text file then the program treat those arg ument as required CPU bust for P1, P2, P3, P4, and P5 process and calculate average waiting time and averag e Turnaround time. Consider used scheduling algorithm as SJF and same arrival time for all the processes.

CONCEPTS USED:

CPU SCHEDULING:
This algorithm uses for allowing one program to use CPU temporarily regardless of their priority, FILE REA DING.
SHORTEST JOB FIRST ALGORITHMS:
This reduces the waiting time of programs that have lesser time for execution. This doesn't follow the hig h priority or low priority it just follows the shortest job first.

CODE:

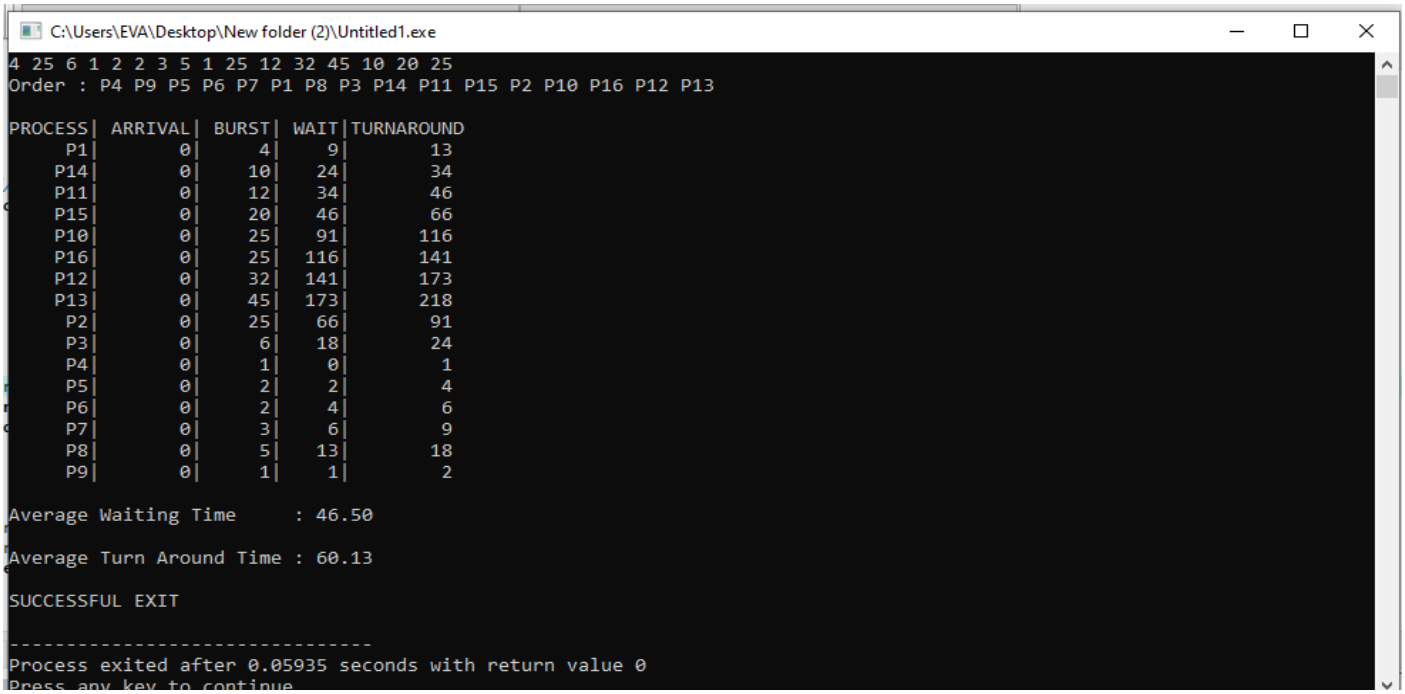
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#define FILE_NAME "CPU_BURST.txt"
struct Process{
        int at,bt,wt,tat;
        char name[4];
};
struct Process initialize(int at,int bt,int name){
        struct Process X;
        X.bt = bt;
        X.at = at;
        sprintf(X.name,"P%d",name+1);
        return X;
}

int main(){
```

```
	FILE *fp = fopen(FILE_NAME,"r");
	if(!fp)
		return -1*printf("FILE OPEN ERROR!\n");
int d,i,j,count=0;
int *queue = (int*)malloc(sizeof(int));
//inputs are space separated integers on a single line of a txt file located in the same directory
while(EOF != fscanf(fp,"%d ",&d )){
	printf("%d ",d);
	queue = (int*)realloc(queue,(count+1)*sizeof(int));
	queue[count++] = d;
}
fclose(fp);
	//int queue[] = {3,1,3,2,4,5};
	struct Process P[count];
	for(i=0; i<count; i++)
		P[i] = initialize(0,queue[i],i);
	//sort
	for(i=1; i<count; i++){
		for(j=0; j<count-i; j++){
			if(P[j].bt>P[j+1].bt){
				struct Process temp = P[j];
				P[j] = P[j+1];
				P[j+1] = temp;
			}
		}
	}
	//FCFS non-preemptive [same arrival time]
	//after sorting we can apply FCFS which will result in SJF]
	printf("\nOrder : ");
	int elapsed_time=0;
	for(i=0; i<count; i++){
		P[i].wt = elapsed_time;
		P[i].tat= P[i].wt+P[i].bt;
		elapsed_time += P[i].bt;
		printf("%s ",P[i].name);
	}
	//sort again
	for(i=1; i<count; i++){
		for(j=0; j<count-i; j++){
			if(P[j].name[1]>P[j+1].name[1]){
				struct Process temp = P[j];
				P[j] = P[j+1];
				P[j+1] = temp;
```

```
                        }
                    }
                }
                printf("\n\n%7s|%8s|%6s|%5s|%s\n","PROCESS","ARRIVAL","BURST","WAIT","TURNAROUND")
;
                int total_wt=0,total_tt=0;
                for(i=0; i<count; i++){
                    total_wt+=P[i].wt;
                    total_tt+=P[i].tat;
                    printf("%7s|%8d|%6d|%5d|%9d\n",P[i].name,P[i].at,P[i].bt,P[i].wt,P[i].tat);
                }
                printf("\nAverage Waiting Time    : %.2f\n",total_wt*1.0/count);
                printf("\nAverage Turn Around Time : %.2f\n",total_tt*1.0/count);
                return 0*printf("\nSUCCESSFUL EXIT\n");
}
```

OUTPUT:



ALGORITHM:

Declare a text file with name CPU_BURST.txt file which contains all the process burst times in it.

Step1- We check if the file exists or not if not we throw a error message.

Step2- We add all the CPU burst time to a queue to process on it.

Step3- Now we sort the data using bubble sort .

Step4- For the processes with same burst file they will be executed by first come first serve.

Step5- Now we apply shortest job first to the sorted data.

Step6- And then we sort the data again.

Step7- We display the data of process, arrival, burst, wait and turn around

Step8-

Now we do the calculations of average waiting time, average turn around time and display the answers

TEST CASES:

- Check the turn around time by entering values and check wether turn around time is correct or no t.
- Check the output and compare the file that you have mentioned in "File " that read with name "cp u_burst".
- Check the data wether following sjf or not.

QUES 2:

A uniprocessor system has n number of CPU intensive processes, each process has its own requirement of CPU burst.The process with lowest CPU burst is given the highest priority. A late arriving higher priority process can preempt a currently running process with lower priority. Simulate a scheduler that is scheduling the processes in such a way that higher priority process is never starved due to the execution of lower priority process. What should be its average waiting time and average turnaround time if no two processes are arriving at same time

CONCEPTS USED:

CPU SCHEDULING,
UNIPROCESSING OF CPU INTENSIVE PROCESSES.
These has been used in this question.

CODE:

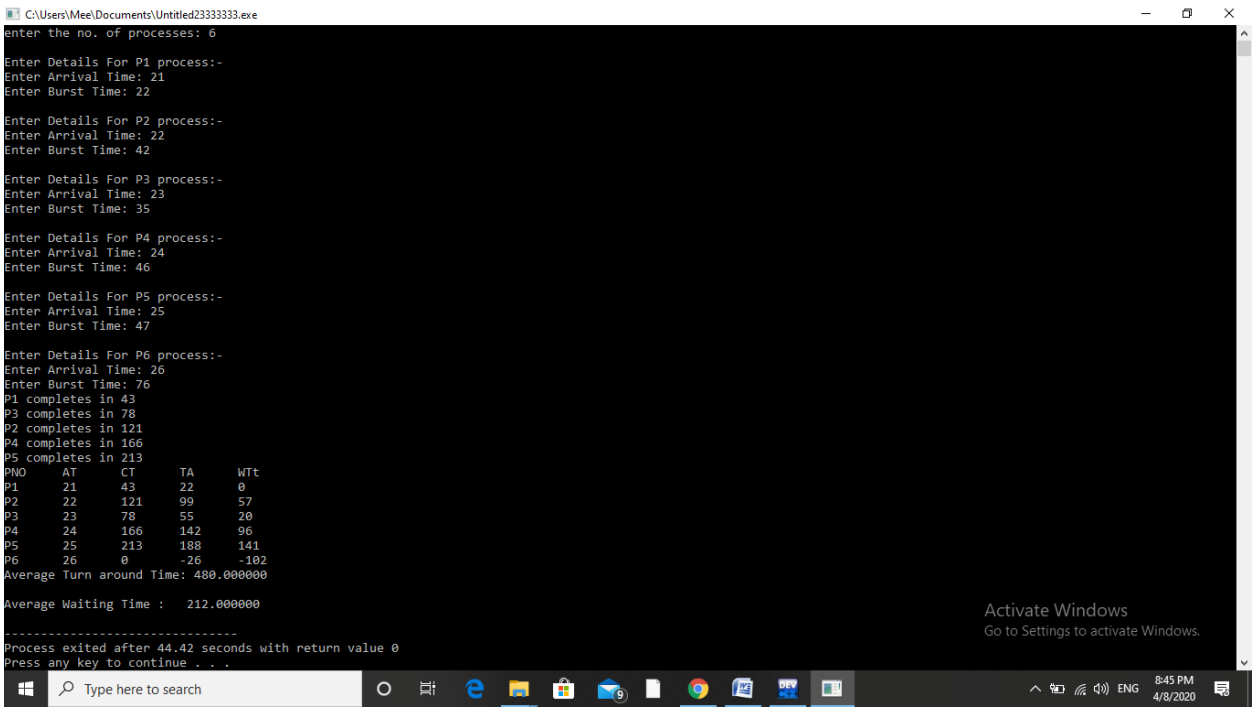
```
#include<stdio.h>
int n;
struct process
{
	int p_no;
	int arrival_t,burst_t,ct,wait_t,taround_time,p;
	int flag;
}
p_list[100];
void Sorting()
{
	struct process p;
	int i, j;
	for(i=0;i<n-1;i++)
	{
		for(j=i+1;j<n;j++)
		{
			if(p_list[i].arrival_t > p_list[j].arrival_t)
			{
				p = p_list[i];
				p_list[i] = p_list[j];
				p_list[j] = p;
			}
		}
	}
}
int main()
```

```
{
        int i,t=0,b_t=0,peak;
        int a[10];
        float wait_time = 0, taround_time = 0, avg_w_t=0, avg_taround_time=0;
        printf("enter the no. of processes: ");
        scanf("%d",&n);
        for(i = 0; i < n; i++)
        {
                p_list[i].p_no = i+1;
                printf("\nEnter Details For P%d process:-\n", p_list[i].p_no);
                printf("Enter Arrival Time: ");
                scanf("%d", &p_list[i].arrival_t );
                printf("Enter Burst Time: ");
                scanf("%d", &p_list[i].burst_t);
                p_list[i].flag = 0;
                b_t = b_t + p_list[i].burst_t;
        }
        Sorting();
        for(int i=0;i<n;i++)
        {
                a[i]=p_list[i].burst_t;
        }
        p_list[9].burst_t = 9999;
        for(t = p_list[0].arrival_t; t <= b_t+1;)
        {
                peak = 9;
                for(i=0;i<n;i++)
                {

        if(p_list[i].arrival_t <= t && p_list[i].burst_t < p_list[peak].burst_t && p_list[i].flag != 1)
                {
                peak = i;
        }
        if(p_list[peak].burst_t==0 && p_list[i].flag != 1)
        {
                p_list[i].flag = 1;
                p_list[peak].ct=t;p_list[peak].burst_t=9999;
                printf("P%d completes in %d\n",p_list[i].p_no,p_list[peak].ct);
        }
}
t++;
(p_list[peak].burst_t)--;
}
```

```
for(i=0;i<n;i++)
{
        p_list[i].taround_time=(p_list[i].ct)-(p_list[i].arrival_t);
        avg_taround_time=avg_taround_time+p_list[i].taround_time;
        p_list[i].wait_t=((p_list[i].taround_time)-a[i]);
        avg_w_t=avg_w_t+p_list[i].wait_t;
}
printf("PNO\tAT\tCT\tTA\tWTt\n");
for(i=0;i<n;i++)
{
        printf("P%d\t%d\t%d\t%d\t%d\n",p_list[i].p_no,p_list[i].arrival_t,p_list[i].ct,p_list[i].taround_ti
me,p_list[i].wait_t);
}
printf("Average Turn around Time: %f\t\n\n",avg_taround_time);
printf("Average Waiting Time :\t %f\t\n",avg_w_t);
}
```

OUTPUT:



ALGORITHM:

Step1- We accept the no of processes from the user.
Step2- WE accept the details of arrival time and burst time of n processes entered by the user.
Step3- We sort the data in ascending order to give priority to the task with lower time.
Step4-
We compute the data print which task is first to be executed and finished and how much time it takes.
Step5- We display the data AT,CT,TA,WT in a tabular form.
Step 6- We calculate and display the average turnaround time and average waiting time.

TEST CASES:

- Input the no processors and check the ouput wether it returns the same or not.
- Input the values of arrival time and burst time. Then check the Turn around time and Total waiting time.
- Check the output manually then check wether it returning same or not.