National University of Computer and Emerging Sciences

Project REPORT
for
Operating Systems

|PROJECT : Real-time Scheduling with RPI|

SUBMITTED BY:

Arsalan Zubair	20K-0215
Zubair Ahmed	19K-0258
Chander Parkash	20K-1091

Section: CS

INTRODUCTION:

Real-time CPU scheduling is an Operating Systems project based on C language, designed in a way to work with Linux OS and Raspberry PI.

It is composed of a scheduler, clock and process. The scheduler is supposed to be the important part of this project as priority has important use cases, the one with the highest priority and ones with the lowest priorities. Depending on the characteristics of the scheduling algorithms, it accepts tasks based on the deadline and with the proper use of the algorithms we can prioritize and schedule it.

Features:

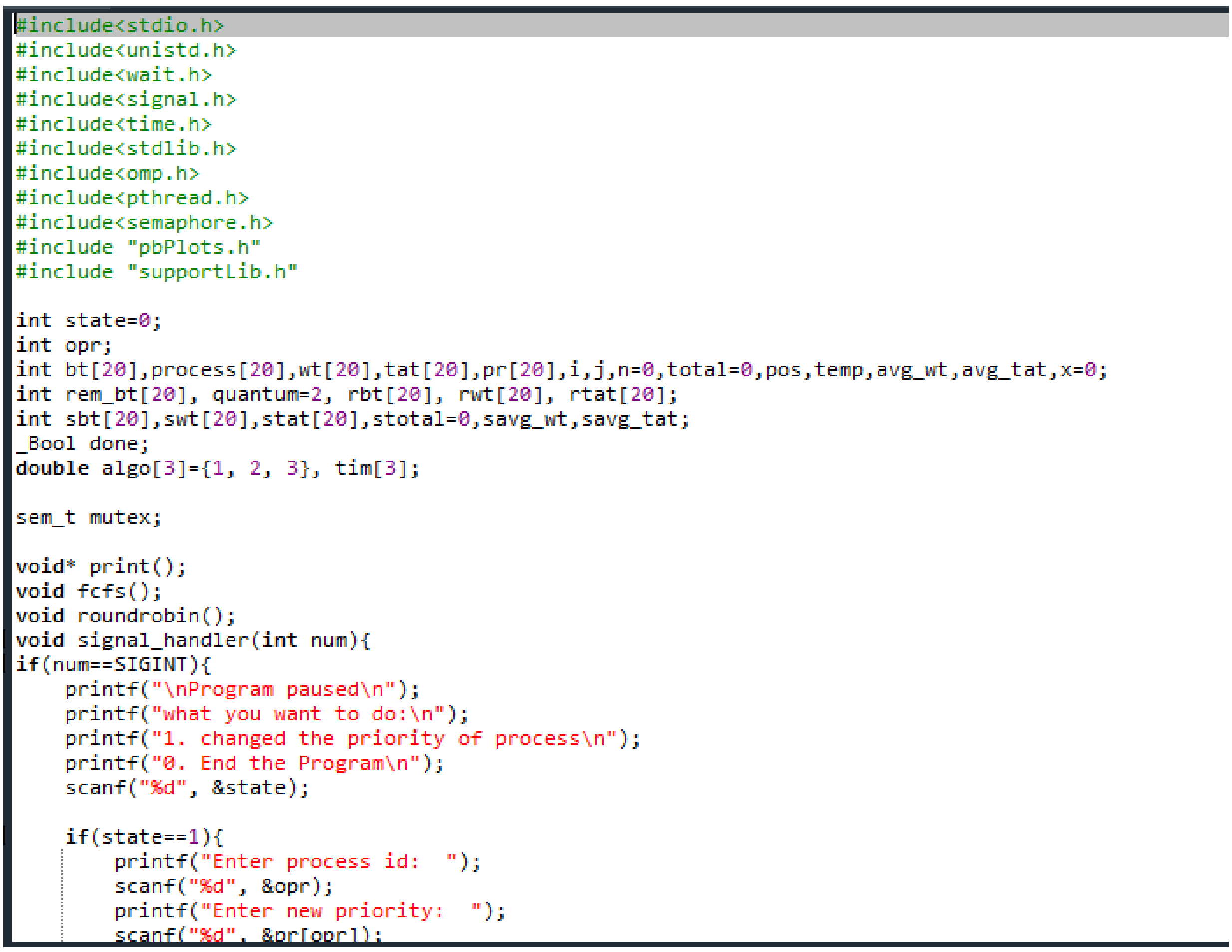
The main function/features deals with the processing id's, along with the waiting and the burst time, once provided with the sufficient information, the scheduling algorithms like FCPS, SJF, Round Robin works and gets the job done.

Technology:

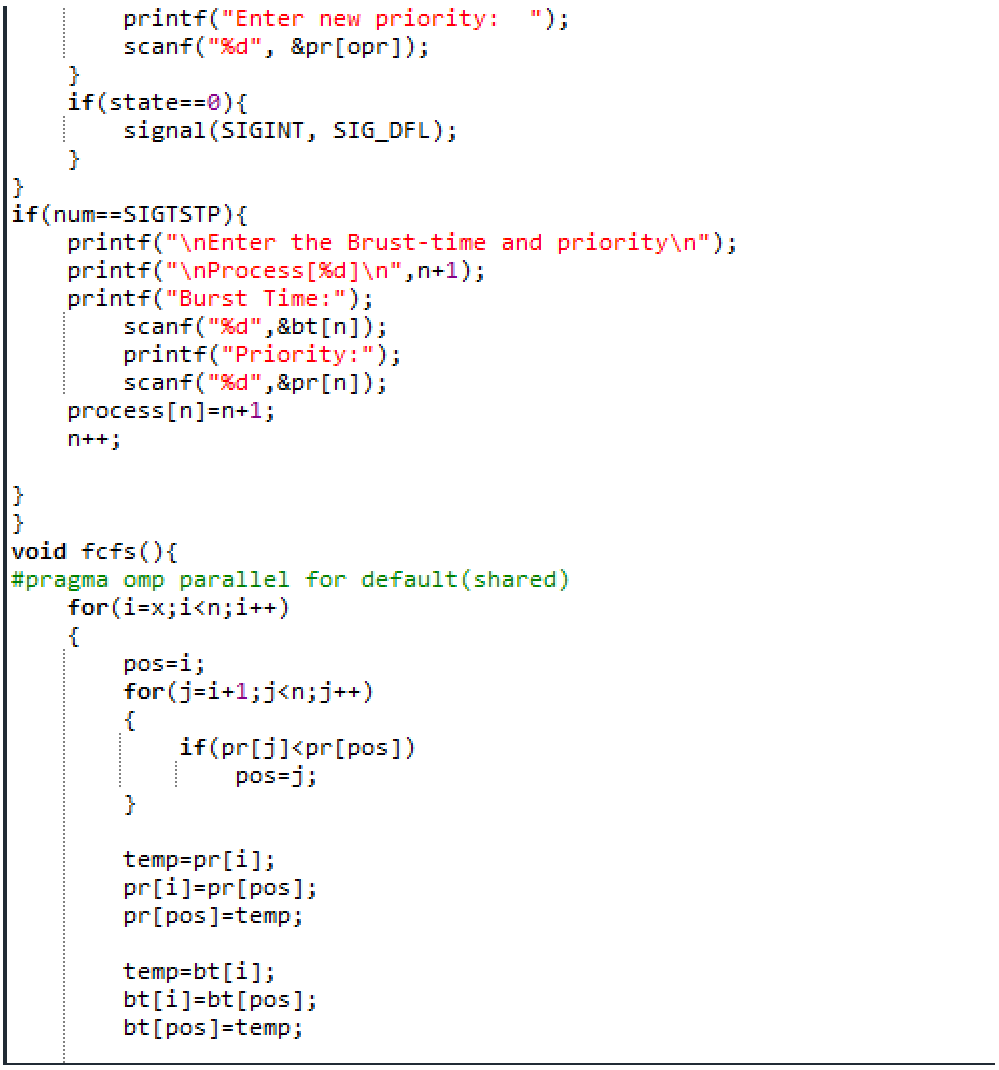
Linux(Ubuntu/Raspbian OS)

Language: C

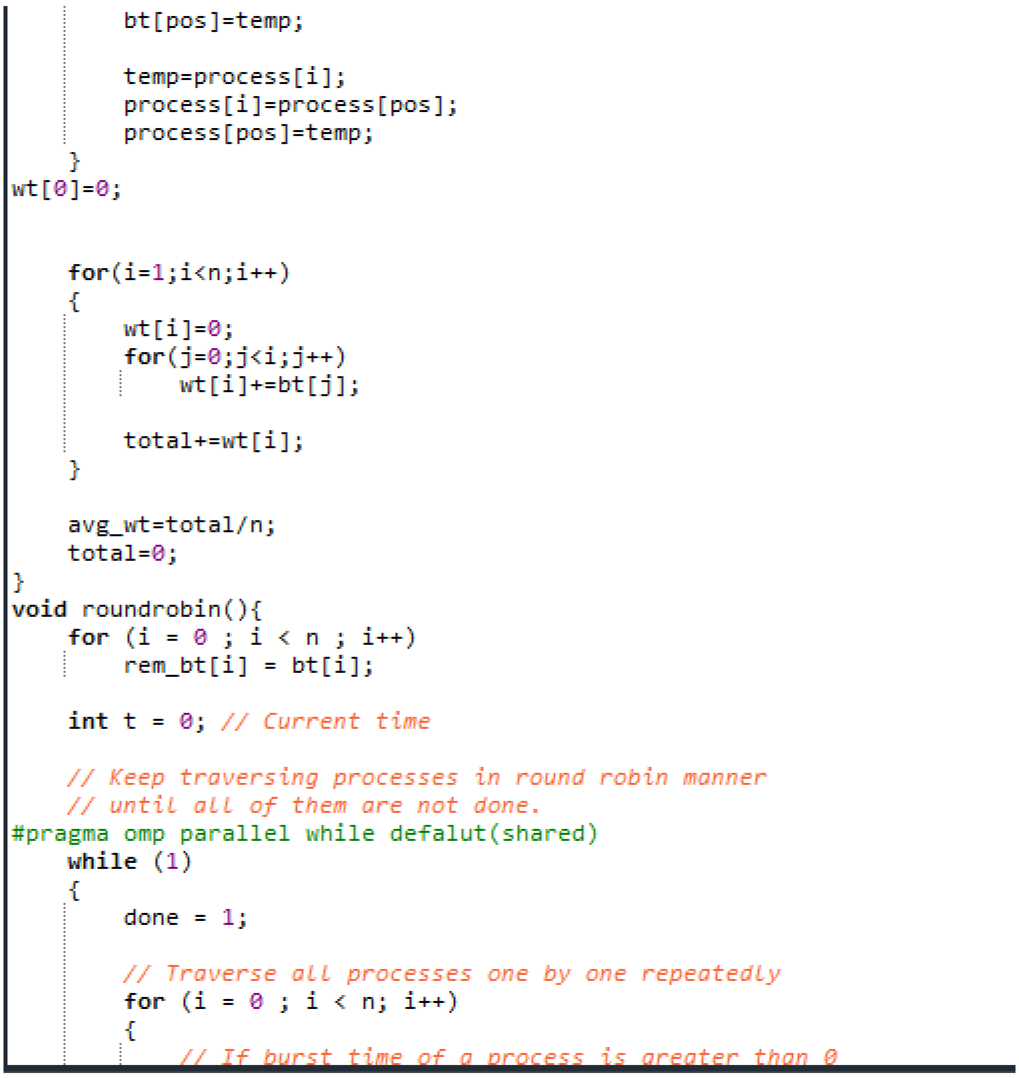
Code Snippets



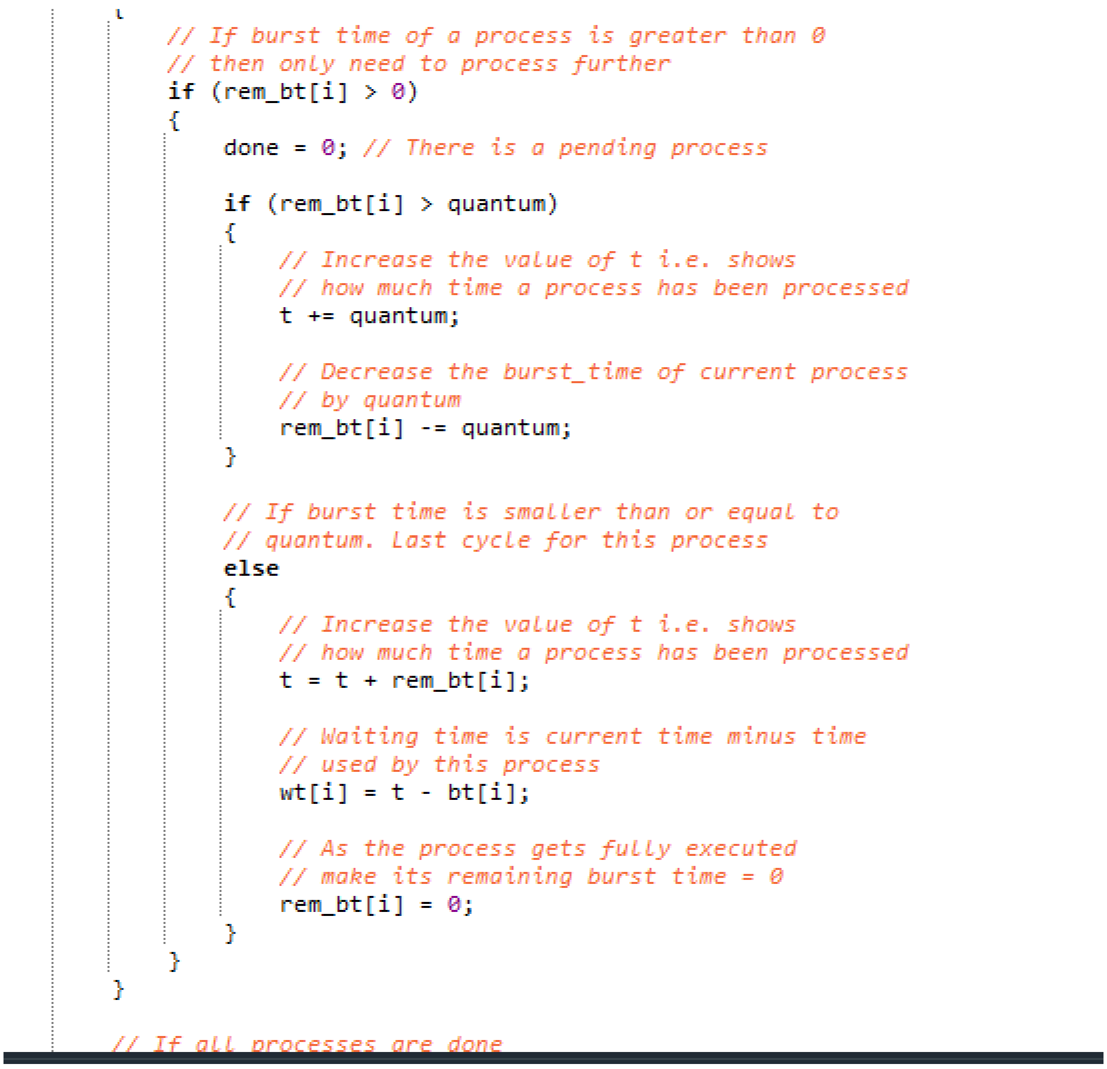
```c
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
#include<signal.h>
#include<time.h>
#include<stdlib.h>
#include<omp.h>
#include<pthread.h>
#include<semaphore.h>
#include "pbPlots.h"
#include "supportLib.h"

int state=0;
int opr;
int bt[20],process[20],wt[20],tat[20],pr[20],i,j,n=0,total=0,pos,temp,avg_wt,avg_tat,x=0;
int rem_bt[20], quantum=2, rbt[20], rwt[20], rtat[20];
int sbt[20],swt[20],stat[20],stotal=0,savg_wt,savg_tat;
_Bool done;
double algo[3]={1, 2, 3}, tim[3];

sem_t mutex;

void* print();
void fcfs();
void roundrobin();
void signal_handler(int num){
if(num==SIGINT){
    printf("\nProgram paused\n");
    printf("what you want to do:\n");
    printf("1. changed the priority of process\n");
    printf("0. End the Program\n");
    scanf("%d", &state);

    if(state==1){
        printf("Enter process id:  ");
        scanf("%d", &opr);
        printf("Enter new priority:  ");
        scanf("%d", &pr[opr]);
```



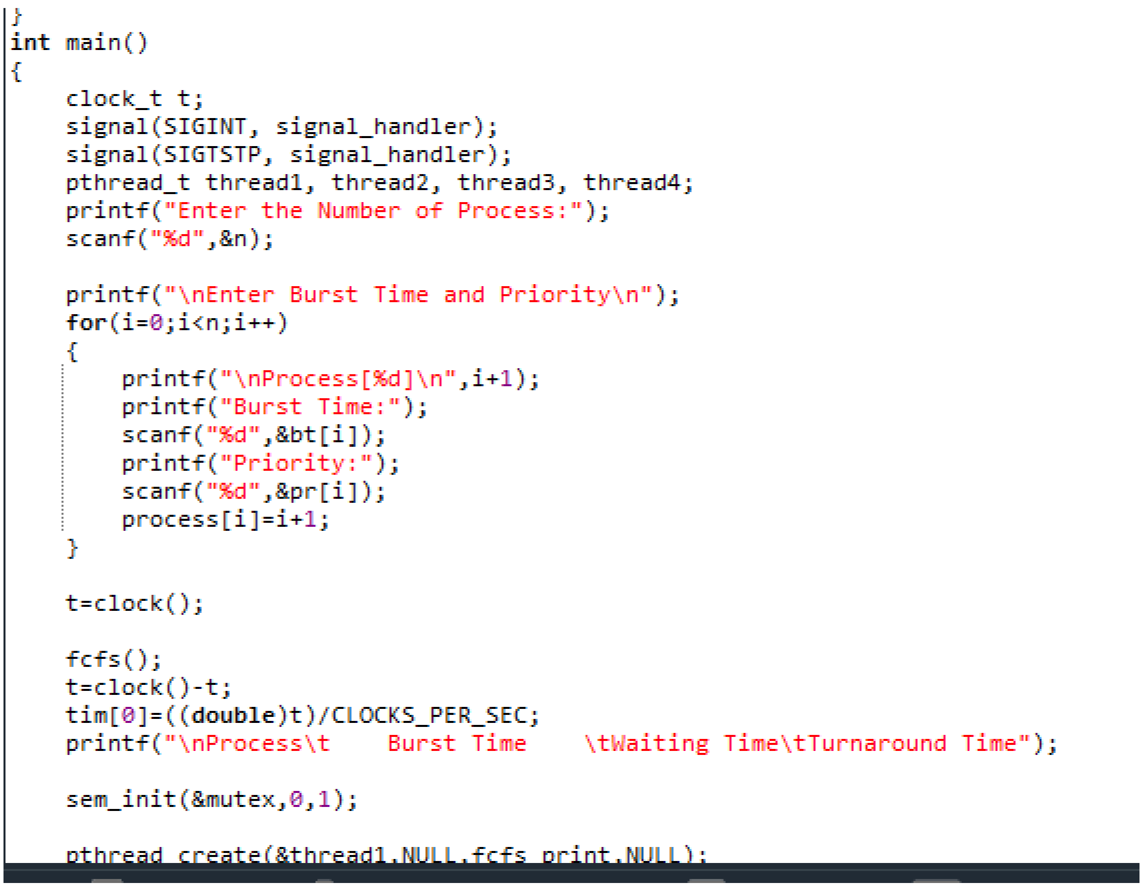
```c
        printf("Enter new priority:  ");
        scanf("%d", &pr[opr]);
    }
    if(state==0){
        signal(SIGINT, SIG_DFL);
    }
}
if(num==SIGTSTP){
    printf("\nEnter the Brust-time and priority\n");
    printf("\nProcess[%d]\n",n+1);
    printf("Burst Time:");
        scanf("%d",&bt[n]);
        printf("Priority:");
        scanf("%d",&pr[n]);
    process[n]=n+1;
    n++;

}
}
void fcfs(){
#pragma omp parallel for default(shared)
    for(i=x;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
        }

        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;
```



```
        bt[pos]=temp;

        temp=process[i];
        process[i]=process[pos];
        process[pos]=temp;
    }
wt[0]=0;


    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=total/n;
    total=0;
}
void roundrobin(){
    for (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.
#pragma omp parallel while defalut(shared)
    while (1)
    {
        done = 1;

        // Traverse all processes one by one repeatedly
        for (i = 0 ; i < n; i++)
        {
            // If burst time of a process is greater than 0
```



```
        // If burst time of a process is greater than 0
        // then only need to process further
        if (rem_bt[i] > 0)
        {
            done = 0; // 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
```



```
}
int main()
{
    clock_t t;
    signal(SIGINT, signal_handler);
    signal(SIGTSTP, signal_handler);
    pthread_t thread1, thread2, thread3, thread4;
    printf("Enter the Number of Process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nProcess[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        process[i]=i+1;
    }

    t=clock();

    fcfs();
    t=clock()-t;
    tim[0]=((double)t)/CLOCKS_PER_SEC;
    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");

    sem_init(&mutex,0,1);

    pthread_create(&thread1,NULL,fcfs_print,NULL);
```