/\*Develop a menu driven C program to implement the CPU Scheduling Algorithms

Priootity (Non-Preemptive and Preemptive) and Round Robin \*/

#include<stdio.h>

#include <stdlib.h>

#include<string.h>

#include<ctype.h>

//Struture representing each process

struct Job{

char \*PID;

double arrivalTime;

double burstTime;

double dummy; //Copy of burst time

double waitTime;

double turnTime;

double responseTime;

int priority;

int nope; //Number of pre-emptions

int chance; //Keep track of chance of process in Round Robin scheduling

};

typedef struct Job Process;

void initialise(Process \*p);

void acceptProcess(Process \*p);

void sortOnArrivalTime(Process p[],int start\_index,int end\_index);

void sortOnPriority(Process p[],int number\_of\_processes);

void displayGanttChart(char \*Gantt\_Chart[],int number\_of\_interval,double start\_times[],double end\_times[]);

void printWaitTime(Process P[],int number\_of\_processes);

void printTurnTime(Process P[],int number\_of\_processes);

void printRespTime(Process P[],int number\_of\_processes);

void NonPriority(Process P[],int number\_of\_processes);

void Priority(Process P[],int number\_of\_processes);

void RoundRobin(Process P[],int number\_of\_processes,int tq);

typedef struct{

int front,rear;

Process data[100];

int capacity,size;

}Queue;

void initialiseQueue(Queue \*q);

int isFull(Queue \*q);

int isEmpty(Queue \*q);

void enqueue(Queue \*q,Process x);

Process dequeue(Queue \*q);

void display(Queue \*q);

void initialiseQueue(Queue \*q){

q->front=q->rear=-1;

q->capacity=100;

q->size=0;

}

int isFull(Queue \*q){

if((q->rear==q->capacity-1&&q->front==0)||(q->rear==q->front-1))

return 1;

else

return 0;

}

int isEmpty(Queue \*q){

if(q->front==-1)

return 1;

else

return 0;

}

void enqueue(Queue \*q,Process x){

if(isFull(q))

printf("Queue is full ");

else{

if(q->front==-1)

q->front++;

if(q->rear==q->capacity-1)

q->rear=0;

else

q->rear++;

q->size++;

q->data[q->rear]=x;

}

}

Process dequeue(Queue \*q){

Process x;

initialise(&x);

if(isEmpty(q))

printf("Queue is empty");

else{

x=q->data[q->front];

q->size--;

if(q->front==q->rear)

q->front=q->rear=-1;

else if(q->front==q->capacity-1)

q->front=0;

else

q->front++;

}

return x;

}

void display(Queue \*q){

if(isEmpty(q))

printf("\nQueue is empty\n");

else{

int i=q->front;

while(i!=q->rear){

printf("%s ",q->data[i].PID);

if(i==q->capacity-1)

i=0;

else

i++;

}

printf("%s ",q->data[i].PID);

}

}

int checkQueue(Process p,Queue \*q){

if(isEmpty(q))

printf("\nQueue is empty\n");

else{

int i=q->front;

while(i!=q->rear){

if(strcmp(p.PID,q->data[i].PID)==0)

return 1;

if(i==q->capacity-1)

i=0;

else

i++;

}

if(strcmp(p.PID,q->data[i].PID)==0)

return 1;

return 0;

}

}

#include "Scheduling2Proto.h"

//Initialising the data members of each process

void initialise(Process \*p){

p->PID=(char\*)malloc(100\*sizeof(char));

p->arrivalTime=0.0;

p->burstTime=0.0;

p->dummy=0.0;

p->waitTime=0.0;

p->turnTime=0.0;

p->responseTime=-1.0;

p->priority=0;

p->nope=0;

p->chance=0;

}

//Accepting data of each process

void acceptProcess(Process \*p){

printf("\n Enter Process ID: ");scanf(" %s",p->PID);

printf("\n Enter arrival time: ");scanf("%lf",&p->arrivalTime);

printf("\n Enter burst time: ");scanf("%lf",&p->burstTime);

printf("\n Enter priority: ");scanf("%d",&p->priority);

p->dummy=p->burstTime;

}

//Sorting using Insertion sort

void sortOnArrivalTime(Process p[],int start\_index,int end\_index){

for(int i=start\_index;i<end\_index;i++){

Process key=p[i];

int j=i-1;

for(;j>=start\_index&&key.arrivalTime<p[j].arrivalTime;j--)

p[j+1]=p[j];

p[j+1]=key;

}

}

//Sorting based on Priority

void sortOnPriority(Process p[],int number\_of\_processes){

for(int i=0;i<number\_of\_processes;i++){

Process key=p[i];

int j=i-1;

for(;j>=0&&key.priority<p[j].priority;j--)

p[j+1]=p[j];

p[j+1]=key;

}

}

//Display Gantt Chart

void displayGanttChart(char \*Gantt\_Chart[],int number\_of\_interval,double start\_times[],double end\_times[]){

//Display top line

printf("\n Gantt\_Chart:\n");

for(int i=0;i<number\_of\_interval;i++){

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

}

//Display order of processes

printf("\n|");

for(int i=0;i<number\_of\_interval;i++)

printf("\_\_\_\_\_%4s\_\_\_\_\_|",Gantt\_Chart[i]);

printf("\n");

//Display time line

int i=0;

for(i=0;i<number\_of\_interval;i++)

printf("%-15.0lf",start\_times[i]);

printf("%-15.0lf",end\_times[i-1]);

printf("\n\n");

}

//Print Wait Time

void printWaitTime(Process P[],int number\_of\_processes){

int i=0;

double sum=0.0;

printf("\n Wait Time:\n");

for(i=0;i<number\_of\_processes;i++){

printf(" %-5.2lf",P[i].waitTime);

sum+=P[i].waitTime;

}

printf("\nAverage: %-5.2lf",sum/number\_of\_processes);

printf("\n");

}

//Print Turnaround Time

void printTurnTime(Process P[],int number\_of\_processes){

int i=0;

double sum=0.0;

printf("\n Turnaround Time:\n");

for(i=0;i<number\_of\_processes;i++){

printf(" %-5.2lf",P[i].turnTime);

sum+=P[i].turnTime;

}

printf("\nAverage: %-5.2lf",sum/number\_of\_processes);

printf("\n");

}

//Print Response Time

void printRespTime(Process P[],int number\_of\_processes){

int i=0;

double sum=0.0;

printf("\n Response Time:\n");

for(i=0;i<number\_of\_processes;i++){

if(P[i].responseTime<0)

P[i].responseTime=0.0;

printf(" %-5.2lf",P[i].responseTime);

sum+=P[i].responseTime;

}

printf("\nAverage: %-5.2lf",sum/number\_of\_processes);

printf("\n");

}

//Non-Preemptive Priority Scheduling

void NonPriority(Process P[],int number\_of\_processes){

//Total time of execution

double sum=0;

for(int i=0;i<number\_of\_processes;i++)

sum+=P[i].burstTime;

//Gantt chart

char \*Gantt\_Chart[100];

for(int i=0;i<100;i++)

Gantt\_Chart[i]=(char\*)malloc(10\*sizeof(char));

//Start and end times of processes

int interval=0;

double start\_times[100];

double end\_times[100];

for(int time=0;time<sum;){

Process tmp[100];

for(int i=0;i<100;i++)

initialise(&tmp[i]);

int tctr=0;

for(int i=0;i<number\_of\_processes;i++)

if(P[i].arrivalTime<=time&&P[i].priority){

tmp[tctr++]=P[i];

}

sortOnPriority(tmp,tctr);

if(tctr==0){

strcpy(Gantt\_Chart[interval],"////");

if(interval==0){

start\_times[interval]=0;

}

else{

start\_times[interval]=end\_times[interval-1];

}

}

else{

for(int i=0;i<tctr;i++){

strcpy(Gantt\_Chart[interval],tmp[i].PID);

if(interval==0){

start\_times[interval]=0;

}

else{

start\_times[interval]=end\_times[interval-1];

}

end\_times[interval]=start\_times[interval]+tmp[i].burstTime;

int j=0;

for(j=0;j<number\_of\_processes;j++){

if(strcmp(tmp[i].PID,P[j].PID)==0){

P[j].priority=0;

P[j].waitTime=start\_times[interval]-P[j].arrivalTime;

P[j].turnTime=P[j].waitTime+P[j].burstTime;

P[j].responseTime=P[j].waitTime;

}

}

interval++;

}

}

time=end\_times[interval-1];

}

displayGanttChart(Gantt\_Chart,interval,start\_times,end\_times);

printWaitTime(P,number\_of\_processes);

printTurnTime(P,number\_of\_processes);

printRespTime(P,number\_of\_processes);

}

//Preemptive Priority Scheduling

void Priority(Process P[],int number\_of\_processes){

//Total time of execution

double sum=0;

for(int i=0;i<number\_of\_processes;i++)

sum+=P[i].burstTime;

//Gantt chart

char \*Gantt\_Chart[100];

for(int i=0;i<100;i++)

Gantt\_Chart[i]=(char\*)malloc(10\*sizeof(char));

//Start and end times of processes

int interval=0;

double start\_times[100];

double end\_times[100];

for(int time=0;time<sum;time++){

int flag=0;

Process tmp[100];

for(int i=0;i<100;i++)

initialise(&tmp[i]);

int tctr=0;

for(int i=0;i<number\_of\_processes;i++)

if(P[i].arrivalTime<=time&&P[i].burstTime){

tmp[tctr++]=P[i];

}

sortOnPriority(tmp,tctr);

for(int i=0;i<number\_of\_processes;i++){

if(strcmp(tmp[0].PID,P[i].PID)==0)

P[i].burstTime--;

}

if(interval==0){

strcpy(Gantt\_Chart[interval],tmp[0].PID);

start\_times[interval]=0;

flag=1;

interval++;

}

else{

if(strcmp(Gantt\_Chart[interval-1],tmp[0].PID)!=0){

end\_times[interval-1]=time;

strcpy(Gantt\_Chart[interval],tmp[0].PID);

start\_times[interval]=end\_times[interval-1];

flag=1;

interval++;

}

}

int j=0;

for(j=0;j<number\_of\_processes;j++){

if(flag&&strcmp(tmp[0].PID,P[j].PID)==0){

P[j].waitTime+=start\_times[interval-1]-P[j].arrivalTime;

if(P[j].waitTime>0.0){

P[j].nope++;

P[j].waitTime-=(P[j].dummy-P[j].burstTime-P[j].nope);

printf("\n%s %d",P[j].PID,P[j].nope);

if(P[j].nope>1){

P[j].waitTime-=P[j].nope;

}

}

P[j].turnTime=P[j].waitTime+P[j].dummy;

if(P[j].responseTime<0.0)

P[j].responseTime=start\_times[interval-1]-P[j].arrivalTime;

}

}

}

end\_times[interval-1]=sum;

displayGanttChart(Gantt\_Chart,interval,start\_times,end\_times);

printWaitTime(P,number\_of\_processes);

printTurnTime(P,number\_of\_processes);

printRespTime(P,number\_of\_processes);

}

//Round Robin Scheduling

void RoundRobin(Process P[],int number\_of\_processes,int tq){

//Total time of execution

double sum=0;

for(int i=0;i<number\_of\_processes;i++)

sum+=P[i].burstTime;

//Gantt chart

char \*Gantt\_Chart[100];

for(int i=0;i<100;i++)

Gantt\_Chart[i]=(char\*)malloc(10\*sizeof(char));

//Start and end times of processes

int interval=0;

double start\_times[100];

double end\_times[100];

Queue RQ;

initialiseQueue(&RQ);

for(int time=0;time<sum;){

for(int i=0;i<number\_of\_processes;i++){

if(isEmpty(&RQ)){

if(P[i].arrivalTime<=time&&P[i].burstTime>0&&P[i].chance==0){

enqueue(&RQ,P[i]);

P[i].chance=1;

}

}

else{

if(P[i].arrivalTime<=time&&P[i].burstTime>0&&P[i].chance==0&&!checkQueue(P[i],&RQ)){

enqueue(&RQ,P[i]);

P[i].chance=1;

}

}

}

Process DQ;

initialise(&DQ);

if(isEmpty(&RQ))

break;

while(isEmpty(&RQ)==0){

DQ=dequeue(&RQ);

strcpy(Gantt\_Chart[interval],DQ.PID);

if(interval==0){

start\_times[interval]=0;

}

else{

start\_times[interval]=end\_times[interval-1];

}

for(int i=0;i<number\_of\_processes;i++){

if(strcmp(P[i].PID,Gantt\_Chart[interval])==0){

if(P[i].burstTime<tq)

end\_times[interval]=start\_times[interval]+P[i].burstTime;

else

end\_times[interval]=start\_times[interval]+tq;

P[i].burstTime-=tq;

P[i].burstTime=(P[i].burstTime<0)?0:P[i].burstTime;

}

}

time=end\_times[interval];

interval++;

int j=0;

for(j=0;j<number\_of\_processes;j++){

if(strcmp(DQ.PID,P[j].PID)==0){

P[j].waitTime+=start\_times[interval-1]-P[j].arrivalTime;

if(P[j].burstTime>0)

P[j].nope++;

if(P[j].waitTime>0.0){

P[j].waitTime-=(P[j].dummy-P[j].burstTime-P[j].nope);

if(P[j].waitTime<0.0)

P[j].waitTime+=(P[j].dummy-P[j].burstTime);

if(P[j].nope>=1){

P[j].waitTime-=P[j].nope;

}

}

P[j].turnTime=P[j].waitTime+P[j].dummy;

if(P[j].responseTime<0.0)

P[j].responseTime=start\_times[interval-1]-P[j].arrivalTime;

}

}

for(int i=0;i<number\_of\_processes;i++){

if(isEmpty(&RQ)){

if(P[i].arrivalTime<=time&&P[i].burstTime>0&&P[i].chance==0){

enqueue(&RQ,P[i]);

P[i].chance=1;

}

}

else{

if(P[i].arrivalTime<=time&&P[i].burstTime>0&&P[i].chance==0&&!checkQueue(P[i],&RQ)){

enqueue(&RQ,P[i]);

P[i].chance=1;

}

}

}

for(int i=0;i<number\_of\_processes;i++){

if(strcmp(DQ.PID,P[i].PID)==0)

if(P[i].burstTime>0)

enqueue(&RQ,P[i]);

}

}

}

end\_times[interval-1]=sum;

displayGanttChart(Gantt\_Chart,interval,start\_times,end\_times);

printWaitTime(P,number\_of\_processes);

printTurnTime(P,number\_of\_processes);

printRespTime(P,number\_of\_processes);

}

#include "Scheduling2Impl.h"

int main(){

printf("\n\t\tCPU SCHEDULING ALGORITHMS\n");

Process p[100];

int number\_of\_processes;

int algo\_option;

do{

printf("\nChoose your scheduling algorithm ");

printf("\n1. Round Robin\n2. Priority\n0. Exit\n Your Choice: ");

scanf("%d",&algo\_option);

//RoundRobin Scheduling

if(algo\_option==1){

printf("\nEnter the number\_of\_processes:");scanf("%d",&number\_of\_processes);

printf("\nEnter the details of the processes:");

int i;

for(i=0;i<number\_of\_processes;i++){

initialise(&p[i]);

acceptProcess(&p[i]);

}

int tq;

printf("\nEnter the time quantum: ");scanf("%d",&tq);

Process RRp[100];

for(i=0;i<number\_of\_processes;i++){

initialise(&RRp[i]);

RRp[i]=p[i];

}

printf("\n Round Robin Scheduling Output:\n ");

RoundRobin(RRp,number\_of\_processes,tq);

}

//Priority Scheduling

else if(algo\_option==2){

printf("\nEnter the number\_of\_processes:");scanf("%d",&number\_of\_processes);

printf("\nEnter the details of the processes:");

int i;

for(i=0;i<number\_of\_processes;i++){

initialise(&p[i]);

acceptProcess(&p[i]);

}

char preemp\_option;

printf("\n Use Pre-emption? y/n ");scanf(" %c",&preemp\_option);

//Non preemptive Priority Scheduling

if(preemp\_option=='n'||preemp\_option=='N'){

Process NPrip[100];

for(i=0;i<number\_of\_processes;i++){

initialise(&NPrip[i]);

NPrip[i]=p[i];

}

printf("\n Non-preemptive SJF Scheduling Output:\n ");

NonPriority(NPrip,number\_of\_processes);

}

//Preemptive Priority Scheduling

else if(preemp\_option=='y'||preemp\_option=='Y'){

Process PPrip[100];

for(i=0;i<number\_of\_processes;i++){

initialise(&PPrip[i]);

PPrip[i]=p[i];

}

printf("\n Preemptive Priority Scheduling Output:\n ");

Priority(p,number\_of\_processes);

}

else{

printf("\n Invalid choice\n");

}

}

else if(algo\_option!=0){

printf("\n Invalid option\n");

}

else;

}while(algo\_option);

}