CS232L Operating Systems
Assignment 1 : Simulate a Scheduler

Name: Muhammad Munawwar Anwar
ID: ma04289

September 20, 2020

1 Main

1.1 Main.c

```c
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include "Scheduler.h"

int main (int argc, char* argv[]){
    char fifo[5] = "FIFO";
    char sjf[4] = "SJF";
    char stcf[5] = "STCF";
    char rr[3] = "RR";
    if ((argc-1)<2)
    {
        fprintf(stderr,"Error. Usage: ./mysched filename POLICY \nwhere POLICY can be one of the following strings:\nFIFO\nSJF\nSTCF\nRR\n");
        exit(1);
    }
    else
    {
        if(strcmp(argv[2],fifo)!=0 && strcmp(argv[2],sjf)!=0 && strcmp(argv[2],stcf)!=0 && strcmp(argv[2],rr)!=0  )
        {
            fprintf(stderr,"Error. Usage: ./mysched filename POLICY \nwhere POLICY can be one of the following strings:\nFIFO\nSJF\nSTCF\nRR\n");
            exit(1);
        }
    }


    FILE * stream = fopen(argv[1],"r");
    int NumProcesses = 0;
    int x = 0;

    struct Process * ProcessList = NULL;
    char line[1024];
    char pname[10];
    int  pid;
    int  duration;
    int  arrivaltime;
    while (fscanf(stream,"%s",line)!=EOF){

        if (ProcessList == NULL){
            NumProcesses = NumProcesses+1;
            ProcessList = (struct Process*) malloc(NumProcesses * sizeof(struct Process));
        }
        else if (ProcessList != NULL)
        {
            NumProcesses = NumProcesses+1;
            ProcessList = realloc(ProcessList,NumProcesses * sizeof(struct Process));
        }
        char * token = strtok(line,":");
        strcpy((ProcessList+x)->pname,token);
        token = strtok(NULL,":");
        pid = atoi(token);
        (ProcessList+x)->pid = pid;
        token = strtok(NULL,":");
        duration = atoi(token);
        (ProcessList+x)->duration = duration;
        token = strtok(NULL,":");
        arrivaltime = atoi(token);
        (ProcessList+x)->arrivaltime = arrivaltime;
        (ProcessList+x)->time_spent_running = 0;
        x++;
    }

    bubbleSort(ProcessList,NumProcesses);


    struct node * head = NULL;
    int timer = 1;
    int index = 0;
    struct Process* Running = NULL;
    int removal_time = 0;
    int exec_time = 0;
    int pc = 0;
    if (strcmp(argv[2],fifo)==0)
    {

        while (1){
            printf("%d:",timer);
            if (Running != NULL) // A process is currently executing
            {
                printf("%s:",Running->pname);
                print(head);
                if (removal_time == timer) // Execution of the Current Process has completed
                {
                    Running = dequeue(&head);   // Dequeue the process from the Ready Queue
                    if (Running->pid == -1)
                        break;
                    removal_time = timer+ Running->duration;
                }

                for (int index = pc; index<NumProcesses;index++)
                {
                        if( (ProcessList+index)->arrivaltime == timer)
                        {
                            enqueue(&head,(ProcessList+index),(ProcessList+index)->arrivaltime);   // Add the Next Process to the Ready Queue
                            pc ++;
                            break;
                        }
                }
            }
            else
            {
                printf("idle:");
                print(head);
                for (int index = pc; index<NumProcesses;index++)
                    {
                        if( (ProcessList+index)->arrivaltime == timer)
                        {
                            enqueue(&head,(ProcessList+index),(ProcessList+index)->arrivaltime);   // Add the Next Process to the Ready Queue
                            pc++;
                            Running = dequeue(&head);
                            removal_time = timer+ Running->duration;
                            break;
                        }
                }
            }
            timer++;
        }

    }
    if (strcmp(argv[2],sjf)==0)
    {

        while (1){
            printf("%d:",timer);
            if (Running != NULL)
            {
                printf("%s:",Running->pname);
                print(head);
                if (removal_time == timer)  // Execution of the Running Process has completed
                {

                    Running = dequeue(&head);   // Dequeue the process
                    if (Running->pid == -1)
                        break;
                    removal_time = timer+ Running->duration;
                }

                for (int index = pc; index<NumProcesses;index++)
                    {
                        if( (ProcessList+index)->arrivaltime == timer)
                        {
                            enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);   // Add the Next Process to the Ready Queue
                            pc ++;
                            break;
                        }
                }

            }
            else
            {
                printf("idle:");
                print(head);
                for (int index = pc; index<NumProcesses;index++)
                    {
                        if( (ProcessList+index)->arrivaltime == timer)
                        {
                            enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);   // Add the Next Process to the Ready Queue
                            pc++;
                            Running = dequeue(&head);
                            removal_time = timer+ Running->duration;
                            break;
                        }
                }
            }
            timer ++;
        }

    }
    if (strcmp(argv[2],stcf)==0)
    {
        int count1 =0;

        while (1)
        {
            printf("%d:",timer);
            if (Running==NULL)
            {
                printf("idle:");
                print(head);
                for (int index = 0; index<NumProcesses;index++)
                {
                    if( (ProcessList+index)->arrivaltime == timer)
                    {
                        enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);
                        Running = dequeue(&head);
                        removal_time = timer+ Running->duration;
                        count1 = count1 +1;
                    }
                }
            }
            else if (Running!=NULL)
            {
                printf("%s:",Running->pname);
                print(head);
                if (removal_time == timer)  // Execution of the Running Process has completed
                {

                    Running = dequeue(&head);   // Dequeue the process
                    if (Running->pid == -1)
                    {
                        break;
                    }
                    else
                    {
                        removal_time = timer+ Running->duration;
                    }
                }
                for (int index = 0; index<NumProcesses;index++)
                {
                    if( (ProcessList+index)->arrivaltime == timer)
                    {
                        enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);
                        count1 =count1 +1;
                        if (removal_time-timer >(ProcessList+index)->duration)
                        {
                            Running->duration = removal_time - timer;
                            enqueue(&head,Running,Running->duration);
                            Running = dequeue(&head);
                            removal_time = timer+ Running->duration;
                        }

                    }
                }

            }
            timer++;

        }
    }
    if (strcmp(argv[2],rr)==0)
    {
        int count = 0;
        while (count !=NumProcesses || head!=NULL || Running!=NULL)
        {
            printf("%d:",timer);
            if (Running == NULL)
            {
                printf("idle:");
                print(head);
                for (int index = 0; index<NumProcesses;index++)
```

```c
                {
                    if((ProcessList+index)->arrivaltime == timer)
                    {
                        enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);
                        count = count + 1;
                    }

                }
                if (head!=NULL)
                {
                    Running = dequeue(&head);
                    if (Running->pid == -1)
                    break;
                    removal_time = timer + 1;

                    Running->duration = Running->duration - 1;
                }
            }
            else
            {
                printf("%s:",Running->pname);
                print(head);
                if (removal_time==timer)
                {
                    if (Running->duration!=0)
                    {
                        //Running->duration = Running->duration - 1;
                        add_last(&head,Running);
                    }
                    else
                    {
                        Running = NULL;
                    }

                }
                for (int index = 0; index<NumProcesses;index++)
                {
                    if( (ProcessList+index)->arrivaltime == timer)
                    {
                        enqueue(&head,(ProcessList+index),(ProcessList+index)->duration);
                        count = count + 1;



                    }
                }
                if (head!=NULL)
                {
                Running = dequeue(&head);
                if (Running->pid == -1)
                    break;
                removal_time = timer + 1;
                Running->duration = Running->duration - 1;
                }
            }
            timer++;
        }


    }
    free(ProcessList);
    fclose(stream);
    return 0;

}
```

Listing 1: hello.c

2 Scheduler

2.1 Scheduler.c

```c
#ifndef SCHEDULER_H
#define SCHEDULER_H

struct Process{

    char pname[10];
    int  pid;
    int  duration;
    int  arrivaltime;
    int  time_spent_running;
};



struct node{
    struct Process* ProcessNode;
    struct node * next;
    int priority;
};


void bubbleSort(struct Process * ProcessList, int n);
void enqueue (struct node ** headaddr, struct Process* ProcessNode, int priority);
void add_last (struct node ** headaddr,struct Process* ProcessNode);
struct Process* dequeue(struct node ** headaddr);
int print (struct node * head);

#endif
```

Listing 2: Scheduler.h

2.2 Scheduler.h

```c
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include "Scheduler.h"

void bubbleSort(struct Process* ProcessList, int n)
{
   int i, j;
    char T_pname[10];
    int  T_pid;
    int  T_duration;
    int  T_arrivaltime;
    for (i = 0; i < n-1; i++)
    {
        for (j = 0; j < n-i-1; j++)
        {
            if ((ProcessList+j)->arrivaltime > (ProcessList+j+1)->arrivaltime)
            {
                strcpy(T_pname,(ProcessList+j)->pname);
                strcpy((ProcessList+j)->pname,(ProcessList+j+1)->pname);
                strcpy((ProcessList+j+1)->pname,T_pname);

                T_duration = (ProcessList+j)->duration;
                (ProcessList+j)->duration = (ProcessList+j+1)->duration;
                (ProcessList+j+1)->duration = T_duration;

                T_pid = (ProcessList+j)->pid;
                (ProcessList+j)->pid = (ProcessList+j+1)->pid;
                (ProcessList+j+1)->pid = T_pid;
                T_arrivaltime = (ProcessList+j)->arrivaltime;
                (ProcessList+j)->arrivaltime = (ProcessList+j+1)->arrivaltime;
                (ProcessList+j+1)->arrivaltime = T_arrivaltime;
            }
        }
    }
}

void enqueue (struct node ** headaddr,struct Process* ProcessNode, int priority){

    if (headaddr==NULL){
        fprintf(stderr, "NULL ptr passed\n"); exit(1);
    }

    struct node * n = malloc(sizeof(struct node));


    if (n==NULL){
        fprintf(stderr,"memory allocation failed\n");exit(1);
    }
    n->ProcessNode = ProcessNode;
    n->next = NULL;
    n->priority = priority;


    if (*headaddr == NULL){
        *headaddr = n;
    }
    else {


        if (priority <(*headaddr)->priority)
        {
            n->next= *headaddr;
            *headaddr = n;
        }
        else
        {

            struct node* tmp = * headaddr;
            while (tmp->next != NULL && tmp->next->priority<= priority)
            {
                tmp = tmp -> next;
            }

            n->next = tmp->next;
            tmp-> next = n;
        }
    }
}

void add_last(struct node ** headaddr,struct Process* ProcessNode){

    if (headaddr==NULL){
        fprintf(stderr, "NULL ptr passed\n"); exit(1);
    }

    struct node * n = malloc(sizeof(struct Process));


    if (n==NULL){
        fprintf(stderr,"memory allocation failed\n");exit(1);
    }
    n->ProcessNode = ProcessNode;
    n->next = NULL;


    if (*headaddr == NULL){ // empty list
        *headaddr = n;
        n->priority = n->ProcessNode->arrivaltime;
    }
    else {
        // get to tail
        struct node* tmp = * headaddr;
        while (tmp->next != NULL)
        {
            tmp = tmp -> next;
        }
        tmp -> next = n;
        n->priority = tmp->next->priority + 1;
    }
}

struct  Process* dequeue (struct node ** headaddr){


    if (headaddr==NULL){
        fprintf(stderr, "NULL ptr passed\n"); exit(1);
    }

    if (*headaddr == NULL){ // empty list
        printf("empty");
        struct Process * ProcessNode = malloc(sizeof(struct Process));
        ProcessNode->pid = -1;
        return ProcessNode;
    }

    else
    {
        struct Process * ProcessNode = malloc(sizeof(struct Process));
        struct node *n = *headaddr;
        *headaddr = (*headaddr)->next;
        ProcessNode = n-> ProcessNode;
        free(n);
        return ProcessNode;
    }

}

int print (struct node * head)
{
    if (head == NULL)
    {
        fprintf(stdout,"empty:\n");
        return 0;
    }
    else
    {
        while(head!=NULL)
        {
            fprintf(stdout,"%s(%d),",head->ProcessNode->pname,head->ProcessNode->duration);

            head = head ->next;
        }
        fprintf(stdout,":\n");

    }
    return 1;

}
```

Listing 3: main.c

3 MakeFile

A MakeFile Interface to run all the other MakeFiles

```
build: main.o Scheduler.o
gcc -o main.out main.o Scheduler.o
```

```
main.out : main.o Scheduler.o
gcc -o main.out main.o Scheduler.o

main.o : main.c
gcc -c main.c

Scheduler.o : Scheduler.c
gcc -c Scheduler.c




rebuild:
rm main.o Scheduler.o
gcc -c main.c
gcc -c Scheduler.c
gcc -o main.out Scheduler.o main.o

clean:
rm main.o Scheduler.o

run:
./main.out processes.dat FIFO
./main.out processes.dat SJF
./main.out processes.dat STCF
./main.out processes.dat RR
```