OS Assignment

Student: Caio Marteli

Id: 19598552

This document contains all source code as requested.

PP program:

header.h

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: header.h \*/

/\* Part of OS Assignment Program \*/

/\* ================================== \*/

#ifndef HEADER\_H

#define HEADER\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

/\*Boolean\*/

#define FALSE 0

#define TRUE !FALSE

/\*constants\*/

#define MAX\_SIZE 100 /\*Max number of processes\*/

/\*

\* Structs

\*/

struct Process

{

int arrivalTime, burstTime, priority; /\*all integers\*/

char letter;

};

typedef struct

{

int totalProcesses; /\*all integers\*/

struct Process processes[MAX\_SIZE]; /\*Array of processes\*/

} Input;

// Node

typedef struct node {

int burst;

char letter;

// Lower values indicate higher priority

int priority;

struct node\* next;

} Node;

//Methods

//main

void enter(char\* filename);

void printInfo(Input\* inp);

void menu();

//schedulers.c

void priority(Input\* inp);

int compareArrival(const void \*s1, const void \*s2);

void addToReadyQ(Input\* inp, Node\*\* rq, int\* ct);

double getAvgBurst(Input\* inp);

//fileIO.c

void loadInput(char\* filename, Input\* inp);

Input processInput(char\* filename);

int countLines(char\* filename);

//header priorityQueue.c

Node\* createNode(int d, int p, char l);

void dequeue(Node\*\* head);

void queue(Node\*\* head, int b, int p, char l);

int isEmpty(Node\*\* head);

#endif

Main.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: main.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

int main()

{

menu();

return 0;

}

/\*

\* Simple menu, allows for fileName entering or command 'QUIT' by user

\*/

void menu()

{

char userStr[11];

int pass = FALSE;

do

{

printf("PP simulation:\n");

scanf("%s", userStr);

if(strcmp(userStr, "QUIT") == 0)

{

printf("\*\*\*\*Goodbye!\*\*\*\*\n");

pass = TRUE;

}

else

{

enter(userStr);

//enter("burst\_list2"); //debug HARDCODED

}

}while(pass != TRUE);

}

/\*

\* Method for loading input file from filename parameter

\* IMPORT: filename(char)

\*/

void enter(char\* filename)

{

Input input;

loadInput(filename, &input);

//printInfo(&input); //DEBUG

priority(&input);

}

/\*

\* Utility/debug method prints relevant information from struct made from input file

\* IMPORT: inp(struct pointer)

\*/

void printInfo(Input\* inp)

{

int i;

printf("Total Processes:%d\n",inp->totalProcesses);

for(i = 0; i < inp->totalProcesses; i++)

{

printf("Symbol:%c\nA.T:%d\nB.T:%d\nPRIO:%d\n\n",

inp->processes[i].letter, inp->processes[i].arrivalTime, inp->processes[i].burstTime, inp->processes[i].priority);

}

}

fileIO.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: fileio.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\*IMPORT: String filenameB (filename)

\*EXPORT: Board struct

\*checks then loads command line files, returns true if load is successful

\*/

void loadInput(char\* filename, Input\* inp)

{

FILE\* fp = fopen(filename, "r");

if(fp == NULL) /\*checks if file exists\*/

{

perror(filename);

}

else

{

\*inp = processInput(filename); /\*processes any file and returns it as a struct\*/

printf("Loaded File: %s\n",filename);

}

fclose(fp);

}

/\*NEED TO ADD ALL METHODS\*/

Input processInput(char\* filename)

{

Input inp;

FILE\* fp = fopen(filename, "r");

int i, lineNum;

char symbol = 'A';

lineNum = countLines(filename);

inp.totalProcesses = lineNum; /\*gets number of processes on file by counting the lines\*/

for(i = 0; i < lineNum; i++) /\*loads rest of file into struct\*/

{

fscanf(fp,"%d %d %d", \

&inp.processes[i].arrivalTime, &inp.processes[i].burstTime, &inp.processes[i].priority);

/\*assigns process a letter for gant\*/

inp.processes[i].letter = symbol;

symbol++;

}

fclose(fp);

return inp;

}

/\*

\* Import: filename (string)

\* Export: Num of lines in file (int)

\* Gets number of lines in file

\* PS. filename must be parsed here because scanf changes location of file pointer

\*/

int countLines(char\* filename)

{

int count = 1;

char chr;

FILE\* ptr = fopen(filename, "r");

chr = getc(ptr);

while (chr != EOF) /\* counts newlines until end of file\*/

{

if (chr == '\n')

{

count++;

}

chr = getc(ptr);

}

fclose(ptr);

return count;

}

priorityQueue.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: priorityQueue.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\* Creates new readyQeue by creating head node also holds letter for PID

\* IMPORT: [b]urst(integer), [p]riority(integer), [l]etter(char)

\*/

Node\* createNode(int d, int p, char l)

{

Node\* head = (Node\*)malloc(sizeof(Node));

head->burst = d;

head->priority = p;

head->letter = l;

head->next = NULL;

return head;

}

/\*

\* Removes element at head of list

\* IMPORT: head(struct pointer)

\*/

void dequeue(Node\*\* head)

{

Node\* temp = \*head;

(\*head) = (\*head)->next;

free(temp);

}

/\*

\* Queue according to priority

\* IMPORT: [b]urst(integer), [p]riority(integer), [l]etter(char)

\*/

void queue(Node\*\* head, int b, int p, char l)

{

Node\* start = (\*head);

Node\* temp = createNode(b, p, l);

// If new head of has lesser priority we insert new node before head.

if ((\*head)->priority > p)

{

temp->next = \*head;

(\*head) = temp;

}

else

{

// Goes through list and finds position to insert node

while (start->next != NULL && start->next->priority < p)

{

start = start->next;

}

temp->next = start->next;

start->next = temp;

}

}

/\*

\* Checks if list is empty

\* IMPORT: head(struct pointer)

\* EXPORT: boolean(int)

\*/

int isEmpty(Node\*\* head)

{

return (\*head) == NULL;

}

Schedulers.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: schedulers.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\* Starts PP scheduler draws gant and ticks clock forward

\* IMPORT: inp(struct pointer)

\*/

void priority(Input\* inp)

{

int currentTime = 0;

double avgWaitTime = 0.0, avgTurnTime = 0.0;

char currentProccess = ' ';

int index = 0;

int \*waitingTime = (void\*)malloc(sizeof(int)\*inp->totalProcesses);

//sorts input array by arrival time

qsort(inp->processes, inp->totalProcesses, sizeof(struct Process), compareArrival);

//initalise queue by loading first from array

Node\* rq = createNode(inp->processes[0].burstTime, inp->processes[0].priority, inp->processes[0].letter);

currentProccess = inp->processes[0].letter;

printf("| %c ", currentProccess);

//while queue has items proccess

while (!isEmpty(&rq))

{

addToReadyQ(inp, &rq, &currentTime); //adds to ready queue

if(currentProccess != rq->letter) //context switch

{

currentProccess = rq->letter;

printf("| %c ", currentProccess);

waitingTime[index] = currentTime;

index++;

}

currentTime++; //clock tick

rq->burst--; //decreases processing time by a clock tick

if(rq->burst == 0) //if finished processing dequeue

{

dequeue(&rq);

}

}//END OF WHILE LOOP

waitingTime[index] = currentTime;

double temp = 0.0;

printf("|\n0 ");

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

{

printf("%d ", waitingTime[i]);

temp += (double)waitingTime[i]; //calculates average

}

temp = temp - waitingTime[index];//deletes last added value for avg

avgWaitTime = temp /(index+1);

avgTurnTime = getAvgBurst(inp) + avgWaitTime;

printf("\nAverage waiting time: %0.2f\n", avgWaitTime);

printf("Average turnaround time: %0.2f\n", avgTurnTime);

free(waitingTime);

}

/\*

\* Adds any processes at current A.T to readyQueue

\* IMPORT: inp(address of input array), [r]eady[q]ueue(struct pointer), [c]urrent[t]ime(int pointer)

\*/

void addToReadyQ(Input\* inp, Node\*\* rq, int\* ct)

{

for(int i = 0; i < inp->totalProcesses; i++)

{

if(inp->processes[i].arrivalTime == \*ct)

{

queue(rq, inp->processes[i].burstTime, inp->processes[i].priority, inp->processes[i].letter);

}

}

}

/\*

\* EXPORT: averageBurstTime(double)

\* IMPORT: inp(address of input array), [r]eady[q]ueue(struct pointer), [c]urrent[t]ime(int pointer)

\*/

double getAvgBurst(Input\* inp)

{

double temp = 0.0, avgBurst = 0.0;

for(int i = 0; i < inp->totalProcesses; i++)

{

temp += inp->processes[i].burstTime;

}

avgBurst = temp/inp->totalProcesses;

return avgBurst;

}

/\*

\* Takes two processes, returns the one with earliest arrival time

\* if the same returns first parameter, used for sorting input array

\*/

int compareArrival(const void \*s1, const void \*s2)

{

struct Process \*p1 = (struct Process \*)s1;

struct Process \*p2 = (struct Process \*)s2;

int compare = p1->arrivalTime - p2->arrivalTime;

if(compare == 0)

{

return(p1->priority - p2->priority);

}

else

{

return(p1->arrivalTime - p2->arrivalTime);

}

}

SRTF Program

header.h

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: header.h \*/

/\* Part of OS Assignment Program \*/

/\* ================================== \*/

#ifndef HEADER\_H

#define HEADER\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

/\*Boolean\*/

#define FALSE 0

#define TRUE !FALSE

/\*constants\*/

#define MAX\_SIZE 100 /\*Max number of processes\*/

/\*

\* Structs

\*/

struct Process

{

int arrivalTime, burstTime, priority; /\*all integers\*/

char letter;

};

typedef struct

{

int totalProcesses; /\*all integers\*/

struct Process processes[MAX\_SIZE]; /\*Array of processes\*/

} Input;

// Node

typedef struct node {

int burst;

char letter;

struct node\* next;

} Node;

//Methods

//main

void enter(char\* filename);

void printInfo(Input\* inp);

void menu();

//schedulers.c

void shortestFirst(Input\* inp);

int compareArrival(const void \*s1, const void \*s2);

void addToReadyQ(Input\* inp, Node\*\* rq, int\* ct);

double getAvgBurst(Input\* inp);

//fileIO.c

void loadInput(char\* filename, Input\* inp);

Input processInput(char\* filename);

int countLines(char\* filename);

//header shortestFirst.c

Node\* createNode(int b, char l);

void dequeue(Node\*\* head);

void queue(Node\*\* head, int b, char l);

int isEmpty(Node\*\* head);

#endif

main.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: main.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

int main()

{

menu();

return 0;

}

/\*

\* Simple menu, allows for fileName entering or command 'QUIT' by user

\*/

void menu()

{

char userStr[11];

int pass = FALSE;

do

{

printf("SRTF simulation:\n");

scanf("%s", userStr);

if(strcmp(userStr, "QUIT") == 0)

{

printf("\*\*\*\*Goodbye!\*\*\*\*\n");

pass = TRUE;

}

else

{

enter(userStr);

//enter("burst\_list2"); //debug HARDCODED

}

}while(pass != TRUE);

}

/\*

\* Method for loading input file from filename parameter

\* IMPORT: filename(char)

\*/

void enter(char\* filename)

{

Input input;

loadInput(filename, &input);

//printInfo(&input); //DEBUG

shortestFirst(&input);

}

/\*

\* Utility/debug method prints relevant information from struct made from input file

\* IMPORT: inp(struct pointer)

\*/

void printInfo(Input\* inp)

{

int i;

printf("Total Processes:%d\n",inp->totalProcesses);

for(i = 0; i < inp->totalProcesses; i++)

{

printf("Symbol:%c\nA.T:%d\nB.T:%d\n\n",

inp->processes[i].letter, inp->processes[i].arrivalTime, inp->processes[i].burstTime);

}

}

fileIO.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: fileio.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\*IMPORT: String filenameB (filename)

\*EXPORT: Board struct

\*checks then loads command line files, returns true if load is successful

\*/

void loadInput(char\* filename, Input\* inp)

{

FILE\* fp = fopen(filename, "r");

if(fp == NULL) /\*checks if file exists\*/

{

perror(filename);

}

else

{

\*inp = processInput(filename); /\*processes any file and returns it as a struct\*/

printf("Loaded File: %s\n",filename);

}

fclose(fp);

}

/\*NEED TO ADD ALL METHODS\*/

Input processInput(char\* filename)

{

Input inp;

FILE\* fp = fopen(filename, "r");

int i, lineNum;

char symbol = 'A';

lineNum = countLines(filename);

inp.totalProcesses = lineNum; /\*gets number of processes on file by counting the lines\*/

for(i = 0; i < lineNum; i++) /\*loads rest of file into struct\*/

{

fscanf(fp,"%d %d %d", \

&inp.processes[i].arrivalTime, &inp.processes[i].burstTime, &inp.processes[i].priority);

/\*assigns process a letter for gant\*/

inp.processes[i].letter = symbol;

symbol++;

}

fclose(fp);

return inp;

}

/\*

\* Import: filename (string)

\* Export: Num of lines in file (int)

\* Gets number of lines in file

\* PS. filename must be parsed here because scanf changes location of file pointer

\*/

int countLines(char\* filename)

{

int count = 1;

char chr;

FILE\* ptr = fopen(filename, "r");

chr = getc(ptr);

while (chr != EOF) /\* counts newlines until end of file\*/

{

if (chr == '\n')

{

count++;

}

chr = getc(ptr);

}

fclose(ptr);

return count;

}

schedulers.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: schedulers.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\* Starts PP scheduler draws gant and ticks clock forward

\* IMPORT: inp(struct pointer)

\*/

void shortestFirst(Input\* inp)

{

int currentTime = 0;

double avgWaitTime = 0.0, avgTurnTime = 0.0;

char currentProccess = ' ';

int index = 0;

int \*waitingTime = (void\*)malloc(sizeof(int)\*inp->totalProcesses);

//sorts input array by arrival time

qsort(inp->processes, inp->totalProcesses, sizeof(struct Process), compareArrival);

//initalise queue by loading first from array

Node\* rq = createNode(inp->processes[0].burstTime, inp->processes[0].letter);

currentProccess = inp->processes[0].letter;

printf("| %c ", currentProccess);

//while queue has items proccess

while (!isEmpty(&rq))

{

addToReadyQ(inp, &rq, &currentTime); //adds to ready queue

if(currentProccess != rq->letter) //context switch

{

currentProccess = rq->letter;

printf("| %c ", currentProccess);

waitingTime[index] = currentTime;

index++;

}

currentTime++; //clock tick

rq->burst--; //decreases processing time by a clock tick

if(rq->burst == 0) //if finished processing dequeue

{

dequeue(&rq);

}

}//END OF WHILE LOOP

waitingTime[index] = currentTime;

double temp = 0.0;

printf("|\n0 ");

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

{

printf("%d ", waitingTime[i]);

temp += (double)waitingTime[i]; //calculates average

}

temp = temp - waitingTime[index];//deletes last added value for avg

avgWaitTime = temp /(index+1);

avgTurnTime = getAvgBurst(inp) + avgWaitTime;

printf("\nAverage waiting time: %0.2f\n", avgWaitTime);

printf("Average turnaround time: %0.2f\n", avgTurnTime);

free(waitingTime);

}

/\*

\* Adds any processes at current A.T to readyQueue

\* IMPORT: inp(address of input array), [r]eady[q]ueue(struct pointer), [c]urrent[t]ime(int pointer)

\*/

void addToReadyQ(Input\* inp, Node\*\* rq, int\* ct)

{

for(int i = 0; i < inp->totalProcesses; i++)

{

if(inp->processes[i].arrivalTime == \*ct)

{

queue(rq, inp->processes[i].burstTime, inp->processes[i].letter);

}

}

}

/\*

\* EXPORT: averageBurstTime(double)

\* IMPORT: inp(address of input array), [r]eady[q]ueue(struct pointer), [c]urrent[t]ime(int pointer)

\*/

double getAvgBurst(Input\* inp)

{

double temp = 0.0, avgBurst = 0.0;

for(int i = 0; i < inp->totalProcesses; i++)

{

temp += inp->processes[i].burstTime;

}

avgBurst = temp/inp->totalProcesses;

return avgBurst;

}

/\*

\* Takes two processes, returns the one with earliest arrival time

\* if the same returns first parameter, used for sorting input array

\*/

int compareArrival(const void \*s1, const void \*s2)

{

struct Process \*p1 = (struct Process \*)s1;

struct Process \*p2 = (struct Process \*)s2;

int compare = p1->arrivalTime - p2->arrivalTime;

if(compare == 0)

{

return(p1->burstTime - p2->burstTime);

}

else

{

return(p1->arrivalTime - p2->arrivalTime);

}

}

shortestFirst.c

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: shortestFirst.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include "header.h"

/\*

\* Creates new readyQeue by creating head node also holds letter for PID

\* IMPORT: [b]urst(integer), [p]riority(integer), [l]etter(char)

\*/

Node\* createNode(int b, char l)

{

Node\* head = (Node\*)malloc(sizeof(Node));

head->burst = b;

head->letter = l;

head->next = NULL;

return head;

}

/\*

\* Removes element at head of list

\* IMPORT: head(struct pointer)

\*/

void dequeue(Node\*\* head)

{

Node\* temp = \*head;

(\*head) = (\*head)->next;

free(temp);

}

/\*

\* Queue according to priority in this case priority is shortest burst

\* IMPORT: [b]urst(integer), [l]etter(char)

\*/

void queue(Node\*\* head, int b, char l)

{

Node\* start = (\*head);

Node\* temp = createNode(b, l);

// If new head of has lesser burst we insert new node before head.

if ((\*head)->burst > b)

{

temp->next = \*head;

(\*head) = temp;

}

else

{

// Goes through list and finds position to insert node

while (start->next != NULL && start->next->burst < b)

{

start = start->next;

}

temp->next = start->next;

start->next = temp;

}

}

/\*

\* Checks if list is empty

\* IMPORT: head(struct pointer)

\* EXPORT: boolean(int)

\*/

int isEmpty(Node\*\* head)

{

return (\*head) == NULL;

}

Thread Program(unfinished)

/\* ================================== \*/

/\* Author: Caio Marteli \*/

/\* Student ID: 19598552 \*/

/\* Name: main.c \*/

/\* OS Assignment \*/

/\* ================================== \*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

/\*Boolean\*/

#define FALSE 0

#define TRUE !FALSE

//Declaring thread condition variable

pthread\_cond\_t cond1 = PTHREAD\_COND\_INITIALIZER;

//Declaring mutex

pthread\_mutex\_t lock1 = PTHREAD\_MUTEX\_INITIALIZER;

/\*

\* Struct serves as storage since we can only parse one pointer to threads

\*/

typedef struct

{

char buffer1[40];

char buffer2[40];

} Buffers;

void \*threadA(void \*arg)

{

pthread\_cond\_init(&cond1, NULL);

//Buffers \*bf = (Buffers\*) arg;

pthread\_mutex\_lock(&lock1);

printf("Thread A init...\n");

execl("../PP/PP", "PP", (char\*) NULL); //CANNOT RUN AS THREAD AS IT REPLACES PROCESS

pthread\_cond\_wait(&cond1, &lock1);

// Start critical section

//pthread\_mutex\_lock(&lock1);

// End critical section

pthread\_mutex\_unlock(&lock1);

pthread\_exit(NULL);

}

/\*

\* Thread B runs PP program

\*/

void \*threadB(void \*arg)

{

printf("Thread B init...\n");

execl("../SRTF/SRTF", "SRTF", (char\*) NULL); //runs SRTF sim

pthread\_cond\_wait(&cond1, &lock1);

pthread\_exit(NULL);

}

int main(void)

{

Buffers bf; //Struct used to store buffers

pthread\_cond\_init(&cond1, NULL);

pthread\_t idA; //Thread ID's

//pthread\_t idB;

//creates new threads(thread ID, thread attributes, function to run, arguments)

pthread\_create(&idA, NULL, threadA, &bf); //runs thread A

//pthread\_create(&idB, NULL, threadB, &bf); ////runs thread B

printf("Enter a valid filename or QUIT\n");

scanf("%s", bf.buffer1);

//pthread\_cond\_signal(&cond1);

//pthread\_join(idA, NULL);

//wait until threads work is done (id, parameter to return)

//pthread\_join(idB, NULL);

return 0;

}

/\*

//pthread\_mutex\_lock(&runB);

void suspendMe()

{ // tell the thread to suspend

pthread\_mutex\_lock(&runB);

//m\_SuspendFlag = 1;

pthread\_mutex\_unlock(&runB);

}

void resumeMe()

{ // tell the thread to resume

pthread\_mutex\_lock(&runB);

//m\_SuspendFlag = 0;

//phtread\_cond\_broadcast(&m\_ResumeCond);

pthread\_mutex\_unlock(&runB);

}

\*/

/\*

void checkSuspend()

{ // if suspended, suspend until resumed

pthread\_mutex\_lock(&m\_SuspendMutex);

while (m\_SuspendFlag != 0) pthread\_cond\_wait(&m\_ResumeCond, &m\_SuspendMutex);

pthread\_mutex\_unlock(&m\_SuspendMutex);

}

\*/