Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Allahham	Student ID:	19170251
Other name(s):	Ahmad		
Unit name:	Operating Systems Assignment	Unit ID:	~~19170251~~ COMP2006
Lecturer / unit coordinator:	Sie Teng Soh	Tutor:	Arlen
Date of submission:	10/05/2021	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Ahmad Allahham Date of signature: 10/05/2021

(By submitting this form, you indicate that you agree with all the above text.)

README

HOW TO COMPILE (IGNORE WARNINGS):

TASK1:

gcc Task1.c Functions.c -o Task1

TASK2:

gcc Task2.c Functions.c -o Task2

TASK3:

gcc Task3.c Functions.c -pthread -o Task3

HOW TO RUN:

TASK1:

./Task1

TASK2:

./Task2

TASK3:

./Task3

Assumptions

ASSUMPTION 1:

the processes in the input file are sorted by 'arrival time'

ASSUMPTION 2:

The priority of each process is UNIQUE (meaning no two processes

can have the same priority)

ASSUMPTION 3:

The user MUST include the file extension for the input file name

(i.e: "sim_input.txt", not "sim_input")

Mutual Exclusion Discussion

Only one of the two thread ‘threadA’ and ‘threadB’ can update ‘buffer2’ at a time. This is done by the first thread that is given access to the function blocking the second thread using a mutex. Once it finishes its execution, it increments the number of programs run and waits on the condition that the number of programs run is equal to 2, releasing its lock. Then the second thread executes its program and updates buffer2. The new value in buffer2 is stored and now that the number of programs run equals 2, the condition is met, signalling the first thread to finish its operation. Once both threads have updated buffer2, the parent thread is signalled to resume operation. The main program then waits for the parent thread to complete its execution then prints what buffer2 was updated to using the two child threads.

Testing

The program works correctly, except for when the file name provided by the user is invalid, the output for the average turnaround time and average waiting time is unexpected (may print nothing, may print strange symbols). Otherwise, if the user follows the assumptions stated in the “Assumptions” section, the program will work perfectly.

Sample Input & Output

Input

0 24 10

1 23 7

3 1 5

4 20 4

4 100 2

6 10 3

8 1 9

8 2 6

32 1 8

56 5 1

Output

PP Program

------ 0

| P1 |

------ 1

| P2 |

------ 3

| P3 |

------ 4

| P5 |

------ 56

| P10 |

------ 61

| P5 |

------ 109

| P6 |

------ 119

| P4 |

------ 139

| P8 |

------ 141

| P2 |

------ 162

| P9 |

------ 163

| P7 |

------ 164

| P1 |

------ 187

the average turnaround time = 112.700000, the average waiting time: 94.000000

SRTF Program

------ 0

| P1 |

------ 3

| P3 |

------ 4

| P4 |

------ 6

| P6 |

------ 8

| P7 |

------ 9

| P8 |

------ 11

| P6 |

------ 19

| P4 |

------ 32

| P9 |

------ 33

| P4 |

------ 38

| P1 |

------ 59

| P10 |

------ 64

| P2 |

------ 87

| P5 |

------ 187

the average turnaround time = 38.900000, the average waiting time: 20.200000

ParentThread

---------- SRTF Gantt Chart ----------

------ 0

| P1 |

------ 3

| P3 |

------ 4

| P4 |

------ 6

| P6 |

------ 8

| P7 |

------ 9

| P8 |

------ 11

| P6 |

------ 19

| P4 |

------ 32

| P9 |

------ 33

| P4 |

------ 38

| P1 |

------ 59

| P10 |

------ 64

| P2 |

------ 87

| P5 |

------ 187

---------- PP Gantt Chart ----------

------ 0

| P1 |

------ 1

| P2 |

------ 3

| P3 |

------ 4

| P5 |

------ 56

| P10 |

------ 61

| P5 |

------ 109

| P6 |

------ 119

| P4 |

------ 139

| P8 |

------ 141

| P2 |

------ 162

| P9 |

------ 163

| P7 |

------ 164

| P1 |

------ 187

---------- Average Turn Around Time And Average Waiting Time For PP and SRTF Programs ----------
-----

PP: the average turnaround time = 112.700000, the average waiting time: 94.000000

SRTF: the average turnaround time = 38.900000, the average waiting time: 20.200000

Source Code

Task1.c

```
/************************************************
* Author: Ahmad Allahham                        *
* Filename: Task1.c                             *
* Date: 10/05/2021                              *
************************************************/
#include "Functions.h"
int main(void)
{
   char fileName[14];
   char* programOutput;
   do {
      printf("PP simulation: ");
      scanf("%s", fileName);
      if (strcmp(fileName, "QUIT") != 0) {
         programOutput = runPPProgram(fileName);
         printf("%s", programOutput);
         printf("-----------------------------------------------------------------\n");
      }
   } while (strcmp(fileName, "QUIT") != 0);
   return 0;
}
```

Task2.c

```
/************************************************
* Author: Ahmad Allahham                        *
* Filename: Task2.c                             *
* Date: 10/05/2021                              *
************************************************/
#include "Functions.h"
int main(void)
{
   char fileName[14];
   char* programOutput;
   do {
      printf("SRTF simulation: ");
      scanf("%s", fileName);
      if (strcmp(fileName, "QUIT") != 0) {
         programOutput = runSRTFProgram(fileName);
         printf("%s", programOutput);
         printf("-----------------------------------------------------------------\n");
      }
   } while (strcmp(fileName, "QUIT") != 0);
   return 0;
}
```

Task3.c

```
/************************************************
* Author: Ahmad Allahham                        *
* Filename: Task3.c                             *
* Date: 10/05/2021                              *
************************************************/
#include "Functions.h"
// Declaration of thread condition variable filenameRead
pthread_cond_t filenameRead = PTHREAD_COND_INITIALIZER;
// Declaration of thread condition variable programsExecuted
pthread_cond_t programsExecuted = PTHREAD_COND_INITIALIZER;
// Declaration of thread condition variable programsExecuted2
pthread_cond_t programsExecuted2 = PTHREAD_COND_INITIALIZER;
// declaring mutex
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
char buffer1[14];
char* buffer2;
int numOfProgramsRan = 0;
char programOutputs[2][1000];
int parentThreadBlocking = FALSE;
int childrenBlocking = FALSE;
int main(void) {
   pthread_t parentThread;
   pthread_t threadA;
   pthread_t threadB;
   // Create threads A and B.
   // If isPPExecuting = TRUE, then PP program runs
   // If isPPExecuting = FALSE, then STRF program runs
   Container* AContainer = (Container*) malloc(sizeof(Container));
   AContainer->isPPExecuting = TRUE;

   Container* BContainer = (Container*) malloc(sizeof(Container));
   BContainer->isPPExecuting = FALSE;
   do {
      parentThreadBlocking = FALSE;
      pthread_create(&threadA, NULL, &threadExecution, (void*) AContainer);
      pthread_create(&threadB, NULL, &threadExecution, (void*) BContainer);
      // sleep for 1 second giving threadA and threadB chance to run first
```

```
        sleep(1);
        parentThreadBlocking = TRUE;
        pthread_create(&parentThread, NULL, &threadExecution, NULL);
        // wait for the completion of parentThread
        pthread_join(parentThread, NULL);
        if (strcmp(buffer1, "QUIT") != 0) {
            printf("---------- Average Turn Around Time And Average Waiting Time For PP and SRTF Programs ----------\n\n");
            printf("PP: %s\nSRTF: %s\n", programOutputs[0], programOutputs[1]);
            printf("-------------------------------------------------------------------------------------------------------\n");
        }
    } while (strcmp(buffer1, "QUIT") != 0);
    return 0;
}
void* threadExecution(void* container) {
    int isPPExecuting;
    if (container != NULL) {
        isPPExecuting = ((Container*) container)->isPPExecuting;
    }
    // acquire a lock
    pthread_mutex_lock(&lock);
    if (!parentThreadBlocking) {
        // let's wait on conition variable filenameRead
        pthread_cond_wait(&filenameRead, &lock);
        if (strcmp(buffer1, "QUIT") != 0) {
            if (isPPExecuting) {
                printf("---------- PP Gantt Chart ----------\n");
                buffer2 = runPPProgram(buffer1);
                strcpy(programOutputs[0], buffer2);
                numOfProgramsRan++;
            } else {
                printf("---------- SRTF Gantt Chart ----------\n");
                buffer2 = runSRTFProgram(buffer1);
                strcpy(programOutputs[1], buffer2);
                numOfProgramsRan++;
            }
            if (numOfProgramsRan == 2) {
                numOfProgramsRan = 0;
                pthread_cond_signal(&programsExecuted);
```

```
            pthread_cond_signal(&programsExecuted2);
        } else {
            pthread_cond_wait(&programsExecuted2, &lock);
        }
    } else {
        if (isPPExecuting) {
            numOfProgramsRan++;
            printf("PP: terminate.\n");
        } else {
            numOfProgramsRan++;
            printf("SRTF: terminate.\n");
        }
        if (numOfProgramsRan == 2) {
            numOfProgramsRan = 0;
            pthread_cond_signal(&programsExecuted);
        }
    }
} else {
    // Let's signal condition variable filenameRead
    printf("Filename: ");
    scanf("%s", buffer1);
    pthread_cond_broadcast(&filenameRead);
    // make the parentThread wait for both programs to complete
    pthread_cond_wait(&programsExecuted, &lock);
    if (strcmp(buffer1, "QUIT") == 0) {
        printf("Parent Thread: terminate.\n");
    }
}
// release locks
pthread_mutex_unlock(&lock);
return NULL;
}
```

Functions.h

```
/************************************************
* Author: Ahmad Allahham                        *
* Filename: Functions.h                         *
* Date: 10/05/2021                              *
************************************************/
#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#define HEAP_SIZE 20
#define FALSE 0
#define TRUE !FALSE
/* structure for a process */
typedef struct {
   int arrivalTime;
   int burstTime;
   int burstTimeConst;
   int priority;
   int turnAroundTime;
   int waitingTime;
   int id;
} Process;
typedef struct {
   Process **arr;
   int count;
   int capacity;
   int heap_type; // for min heap , 1 for max heap
} Heap;
typedef struct {
   int isPPExecuting;
} Container;
Heap *CreateHeap(int capacity,int heap_type);
void insert_PP(Heap *h, Process* process);
void heapify_bottom_top_PP(Heap *h,int index);
void heapify_top_bottom_PP(Heap *h, int parent_node);
```

```
Process* PopMin_PP(Heap *h);

void insert_SRTF(Heap *h, Process* process);

void heapify_bottom_top_SRTF(Heap *h,int index);

void heapify_top_bottom_SRTF(Heap *h, int parent_node);

Process* PopMin_SRTF(Heap *h);

void print(Heap *h);

int isFileEmpty(FILE* in);

void parseProcess(Process* processExecuting, Process* newProcess);

char* runPPProgram(char* fileName);

char* runSRTFProgram(char* fileName);

void* threadExecution(void* PPExecuting);
```

Functions.c

```
/************************************************
* Author: Ahmad Allahham                        *
* Filename: Functions.c                         *
* Date: 10/05/2021                              *
************************************************/
#include "Functions.h"
/************************************************
* Author: Sudhanshu Patel                       *
* Function: CreateHeap, insert (modified to     *
* 'insert_PP' && 'insert_SRTF'),                *
* heapify_bottom_top (modified to               *
* 'heapify_bottom_top_PP' and                   *
* 'heapify_top_bottom_SRTF'),                   *
* PopMin (modified to 'PopMin_PP' and           *
* 'PopMin_SRTF'),                               *
* Email: sudhanshuptl13@gmail.com               *
* Date: 10/05/2021                              *
************************************************/
/* Sudhanshu Patel  */
/*
Min Heap implementation in c
*/
Heap *CreateHeap(int capacity,int heap_type){
   Heap *h = (Heap * ) malloc(sizeof(Heap)); //one is number of heap
   //check if memory allocation is fails
   if(h == NULL){
      printf("Memory Error!");
      return;
   }
   h->heap_type = heap_type;
   h->count=0;
   h->capacity = capacity;
   h->arr = (Process **) malloc(capacity*sizeof(Process*)); //size in bytes
   //check if allocation succeed
   if ( h->arr == NULL){
      printf("Memory Error!");
      return;
```

```
    }
    return h;
}
/* ----------------------PP Functions---------------------- */
void insert_PP(Heap *h, Process* process){
    if( h->count < h->capacity){
        h->arr[h->count] = (Process*) malloc(sizeof(Process));
        parseProcess(h->arr[h->count], process);
        heapify_bottom_top_PP(h, h->count);
        h->count++;
    }
}
void heapify_bottom_top_PP(Heap *h,int index){
    Process* temp;
    int parent_node = (index-1)/2;
    if(h->arr[parent_node]->priority > h->arr[index]->priority){
        //swap and recursive call
        temp = h->arr[parent_node];
        h->arr[parent_node] = h->arr[index];
        h->arr[index] = temp;
        heapify_bottom_top_PP(h,parent_node);
    }
}
void heapify_top_bottom_PP(Heap *h, int parent_node){
    int left = parent_node*2+1;
    int right = parent_node*2+2;
    int min;
    Process* temp;
    if(left >= h->count || left <0)
        left = -1;
    if(right >= h->count || right <0)
        right = -1;
    if(left != -1 && h->arr[left]->priority < h->arr[parent_node]->priority)
        min=left;
    else
        min =parent_node;
    if(right != -1 && h->arr[right]->priority < h->arr[min]->priority)
        min = right;
```

```
    if(min != parent_node){
        temp = h->arr[min];
        h->arr[min] = h->arr[parent_node];
        h->arr[parent_node] = temp;
        // recursive  call
        heapify_top_bottom_PP(h, min);
    }
}
Process* PopMin_PP(Heap *h){
    Process* pop;
    if(h->count==0){
        printf("\n__Heap is Empty__\n");
        return NULL;
    }
    // replace first node by last and delete last
    pop = h->arr[0];
    h->arr[0] = h->arr[h->count-1];
    h->count--;
    heapify_top_bottom_PP(h, 0);
    return pop;
}
/* ----------------------SRTF Functions---------------------- */
void insert_SRTF(Heap *h, Process* process){
    if( h->count < h->capacity){
        h->arr[h->count] = (Process*) malloc(sizeof(Process));
        parseProcess(h->arr[h->count], process);
        heapify_bottom_top_SRTF(h, h->count);
        h->count++;
    }
}
void heapify_bottom_top_SRTF(Heap *h,int index){
    Process* temp;
    int parent_node = (index-1)/2;
    if(h->arr[parent_node]->burstTime > h->arr[index]->burstTime){
        //swap and recursive call
        temp = h->arr[parent_node];
        h->arr[parent_node] = h->arr[index];
        h->arr[index] = temp;
```

```
        heapify_bottom_top_SRTF(h,parent_node);
    }
}
void heapify_top_bottom_SRTF(Heap *h, int parent_node){
    int left = parent_node*2+1;
    int right = parent_node*2+2;
    int min;
    Process* temp;
    if(left >= h->count || left <0)
        left = -1;
    if(right >= h->count || right <0)
        right = -1;
    if(left != -1 && h->arr[left]->burstTime < h->arr[parent_node]->burstTime)
        min=left;
    else
        min =parent_node;
    if(right != -1 && h->arr[right]->burstTime < h->arr[min]->burstTime)
        min = right;
    if(min != parent_node){
        temp = h->arr[min];
        h->arr[min] = h->arr[parent_node];
        h->arr[parent_node] = temp;
        // recursive  call
        heapify_top_bottom_SRTF(h, min);
    }
}
Process* PopMin_SRTF(Heap *h){
    Process* pop;
    if(h->count==0){
        printf("\n__Heap is Empty__\n");
        return NULL;
    }
    // replace first node by last and delete last
    pop = h->arr[0];
    h->arr[0] = h->arr[h->count-1];
    h->count--;
    heapify_top_bottom_SRTF(h, 0);
    return pop;
```

```
}
/* ----------------------Utility Functions---------------------- */
int isFileEmpty(FILE* in) {
   int ch;
   int result;
   ch = fgetc(in);
   if (ch == EOF) {
                result = TRUE;
   } else {
                fseek(in, 0, SEEK_SET);
                result = FALSE;
   }
   return  result;
}
void parseProcess(Process* processToBeUpdated, Process* newProcess) {
   processToBeUpdated->arrivalTime = newProcess->arrivalTime;
   processToBeUpdated->burstTime = newProcess->burstTime;
   processToBeUpdated->burstTimeConst = newProcess->burstTimeConst;
   processToBeUpdated->priority = newProcess->priority;
   processToBeUpdated->id = newProcess->id;
   processToBeUpdated->turnAroundTime = newProcess->turnAroundTime;
   processToBeUpdated->waitingTime = newProcess->waitingTime;
}
/* ----------------------PP Program---------------------- */
char* runPPProgram(char* fileName) {
   char* output = (char*) malloc(1024);
   // open the input file for reading
   FILE* in = fopen(fileName, "r");
   if (in == NULL) {
      perror("Error reading input file");
   } else {
      if (!(isFileEmpty(in))) {
         Heap *readyQueue = CreateHeap(HEAP_SIZE, 0);
         Heap *completedProcesses = CreateHeap(HEAP_SIZE, 0);
         int currentTime = 0;
         Process* newProcess = (Process*) malloc(sizeof(Process));
         Process* processExecuting = (Process*) malloc(sizeof(Process));
         int processCounter = 0;
```

```
while (fscanf(in, "%d %d %d", &newProcess->arrivalTime, &newProcess->burstTime, &newProcess->priority) == 3) {
    newProcess->burstTimeConst = newProcess->burstTime;
    processCounter++;
    newProcess->id = processCounter;
    if (processCounter == 1) {
        currentTime = newProcess->arrivalTime;
        parseProcess(processExecuting, newProcess);
        printf("------ %d\n", currentTime);
    } else {
        while ((newProcess->arrivalTime - currentTime) >= processExecuting->burstTime && processExecuting->burstTime > 0) {
            printf("| P%d |\n", processExecuting->id);
            currentTime += processExecuting->burstTime;
            printf("------ %d\n", currentTime);
            processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
            processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
            processExecuting->burstTime = 0;
            insert_PP(completedProcesses, processExecuting);
            if (readyQueue->count > 0) {
                processExecuting = PopMin_PP(readyQueue);
            }
        }
        if (processExecuting->burstTime != 0) {
            processExecuting->burstTime -= newProcess->arrivalTime - currentTime;
        }
        if (processExecuting->priority > newProcess->priority) {
            if (currentTime < newProcess->arrivalTime) {
                printf("| P%d |\n", processExecuting->id);
                printf("------ %d\n", newProcess->arrivalTime);
            }
            if (processExecuting->burstTime != 0) {
                insert_PP(readyQueue, processExecuting);
            } else {
                processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
                processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
                insert_PP(completedProcesses, processExecuting);
            }
            parseProcess(processExecuting, newProcess);
        } else {
```

```
                insert_PP(readyQueue, newProcess);
            }
            currentTime = newProcess->arrivalTime;
        }
    }
    /* print remaining processes in readyQueue */
    if (processExecuting->burstTime > 0) {
        currentTime += processExecuting->burstTime;
        printf("| P%d |\n", processExecuting->id);
        printf("------ %d\n", currentTime);
        processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
        processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
        insert_PP(completedProcesses, processExecuting);
    }
    while (readyQueue->count > 0) {
        processExecuting = PopMin_PP(readyQueue);
        printf("| P%d |\n", processExecuting->id);
        currentTime += processExecuting->burstTime;
        printf("------ %d\n", currentTime);
        processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
        processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
        insert_PP(completedProcesses, processExecuting);
    }
    // print average turnAroundTime and average waitingTime
    double averageTurnAroundTime = 0;
    double averageWaitingTime = 0;
    while (completedProcesses->count > 0) {
        newProcess = PopMin_PP(completedProcesses);
        averageTurnAroundTime += (double) newProcess->turnAroundTime;
        averageWaitingTime += (double) newProcess->waitingTime;
    }

    averageTurnAroundTime /= processCounter;
    averageWaitingTime /= processCounter;
    snprintf(output, 1024, "the average turnaround time = %f, the average waiting time: %f\n", averageTurnAroundTime,
averageWaitingTime);
  }
  if (ferror(in)) {
```

```
            perror("Error reading from file");
        }
        fclose(in);
    }
    return output;
}
/* ----------------------SRTF Program---------------------- */
char* runSRTFProgram(char* fileName) {
    char* output = (char*) malloc(1024);
    // open the input file for reading
    FILE* in = fopen(fileName, "r");
    if (in == NULL) {
        perror("Error reading input file");
    } else {
        if (!(isFileEmpty(in))) {
            Heap *readyQueue = CreateHeap(HEAP_SIZE, 0);
            Heap *completedProcesses = CreateHeap(HEAP_SIZE, 0);
            int currentTime = 0;
            Process* newProcess = (Process*) malloc(sizeof(Process));
            Process* processExecuting = (Process*) malloc(sizeof(Process));
            int processCounter = 0;
            while (fscanf(in, "%d %d %d", &newProcess->arrivalTime, &newProcess->burstTime, &newProcess->priority) == 3) {
                newProcess->burstTimeConst = newProcess->burstTime;
                processCounter++;
                newProcess->id = processCounter;
                if (processCounter == 1) {
                    currentTime = newProcess->arrivalTime;
                    parseProcess(processExecuting, newProcess);
                    printf("------ %d\n", currentTime);
                } else {
                    while ((newProcess->arrivalTime - currentTime) >= processExecuting->burstTime && processExecuting->burstTime > 0) {
                        printf("| P%d |\n", processExecuting->id);
                        currentTime += processExecuting->burstTime;
                        printf("------ %d\n", currentTime);
                        processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
                        processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
                        processExecuting->burstTime = 0;
                        insert_SRTF(completedProcesses, processExecuting);
```

```
            if (readyQueue->count > 0) {
                processExecuting = PopMin_SRTF(readyQueue);
            }
        }
        if (processExecuting->burstTime != 0) {
            processExecuting->burstTime -= newProcess->arrivalTime - currentTime;
        }
        if (processExecuting->burstTime > newProcess->burstTime) {
            if (currentTime < newProcess->arrivalTime) {
                printf("| P%d |\n", processExecuting->id);
                printf("------ %d\n", newProcess->arrivalTime);
            }
            if (processExecuting->burstTime != 0) {
                insert_SRTF(readyQueue, processExecuting);
            } else {
                processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
                processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
                insert_SRTF(completedProcesses, processExecuting);
            }
            parseProcess(processExecuting, newProcess);
        } else {
            insert_SRTF(readyQueue, newProcess);
        }
        currentTime = newProcess->arrivalTime;
    }
}
/* print remaining processes in readyQueue */
if (processExecuting->burstTime > 0) {
    currentTime += processExecuting->burstTime;
    printf("| P%d |\n", processExecuting->id);
    printf("------ %d\n", currentTime);
    processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
    processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
    insert_SRTF(completedProcesses, processExecuting);
}
while (readyQueue->count > 0) {
    processExecuting = PopMin_SRTF(readyQueue);
    printf("| P%d |\n", processExecuting->id);
```

```
            currentTime += processExecuting->burstTime;
            printf("------ %d\n", currentTime);
            processExecuting->turnAroundTime = currentTime - processExecuting->arrivalTime;
            processExecuting->waitingTime = processExecuting->turnAroundTime - processExecuting->burstTimeConst;
            insert_SRTF(completedProcesses, processExecuting);
          }
          // print average turnAroundTime and average waitingTime
          double averageTurnAroundTime = 0;
          double averageWaitingTime = 0;
          while (completedProcesses->count > 0) {
            newProcess = PopMin_SRTF(completedProcesses);
            averageTurnAroundTime += (double) newProcess->turnAroundTime;
            averageWaitingTime += (double) newProcess->waitingTime;
          }

          averageTurnAroundTime /= processCounter;
          averageWaitingTime /= processCounter;
          snprintf(output, 1024, "the average turnaround time = %f, the average waiting time: %f\n", averageTurnAroundTime,
averageWaitingTime);
        }
        if (ferror(in)) {
          perror("Error reading from file");
        }
        fclose(in);
    }
    return output;
}
```