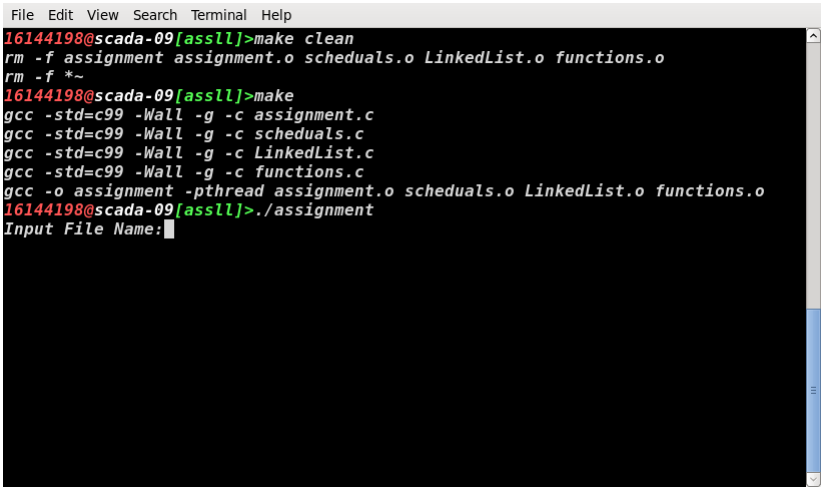
Operating Systems 200 Assignment Report

Bradley Jay Schoone - Student ID: 16144198

May 2014

1 Usage

- Firstly, run "make clean" to remove the old object files and compiles C.
- Build the program by running "make".
- Run the program by entering "./assignment" in the terminal.



2 Mutual Exclusion

Parent Thread

The parent thread is able to write to the input buffer and read from the output buffer. The parent puts a lock on the mutex on the input buffer and lets the user input a name (with relevant validation), once this is correct, it writes to the buffer and sets a flag for the children threads, broadcasts a signal and lets go of the lock.
When it gets to output buffer, it creates a lock on the output buffer and waits in a while loop until the children threads have written to the buffer and set the

relevant flag. Once it passes the wait, it reads from the buffer and outputs the data to the user. Sets the flag back for the other child thread to write to the buffer and then loops this functionality for a second time.

Children Threads

The children threads are able to read from the input buffer, and write to the output buffer. When it gets to input buffer, it creates a lock on the input buffer and waits in a while loop until the parent thread has written to the buffer and set the relevant flag, the child thread then reads from the buffer and stores the data for use in its functionality.
When the child thread gets to the output buffer, the child thread locks the mutex. If the thread is already locked, it will wait until the parent sets the relevant flag back to the original value to let it know it is ready for the second input. Once it gets the lock, it writes to the buffer, sets the relevant flag for the parent thread and then signals the waiting threads that it has completed and unlocks the mutex.

3 Testing

There are multiple test cases i have constructed for my program.

Functionality

- The program does close with "QUIT"
- If the name is invalid (ie the file does not exist) the program will ask for the user to input another name
- If the file is correct, but the data contained in the file isnt of the format specified, the program will not crash/segfault, it will simple print "NAN' for the output of the averages.

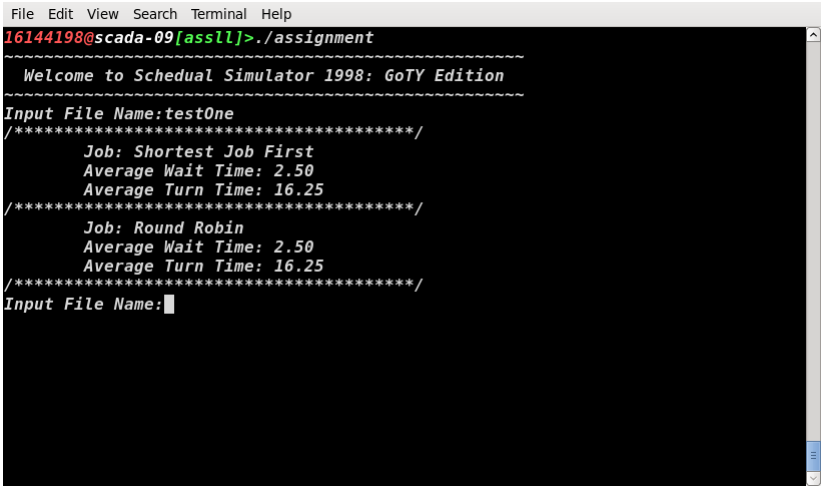
Validity

- The testOne file is a vanilla test file where the first job starts at time 0, there are no gaps of wait time between jobs and the file is in perfect syntax.
- The testTwo file is to make sure the scheduals work correctly where incoming data comes in at the same time as another.
- The testThree file is to make sure that the schedual will run correctly where there are large gaps in the file where the processes will have to wait.
- The testFour file is the data that was given to us in the Assignment Specification

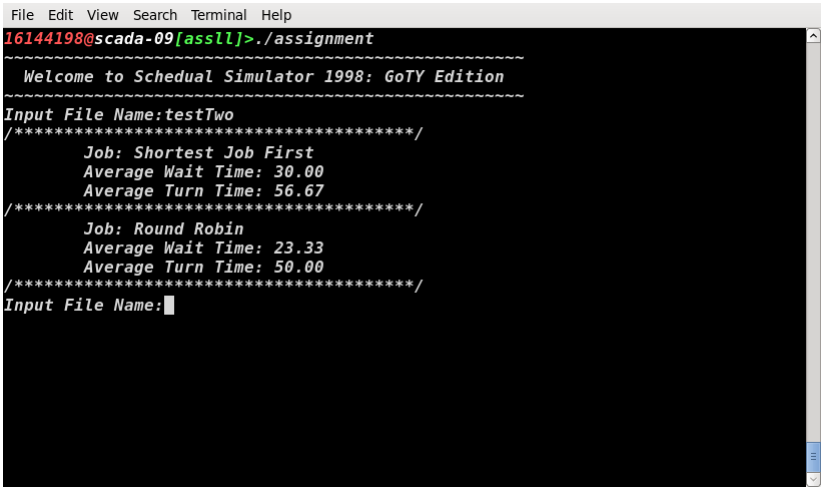
Having run these tests and compared them to fully worked out answers done in either the pracs/the duration of the assignment, i am fairly certain this answers are correct.

4 Sample Input/Output of Running Program

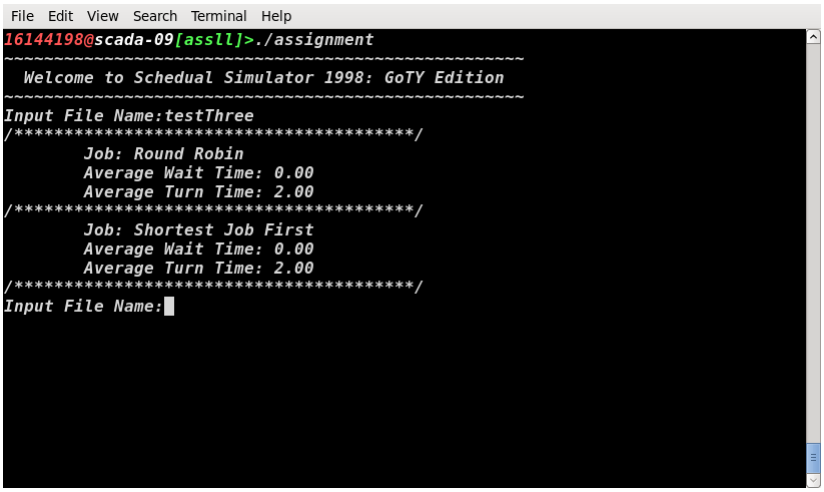
Example: testOne



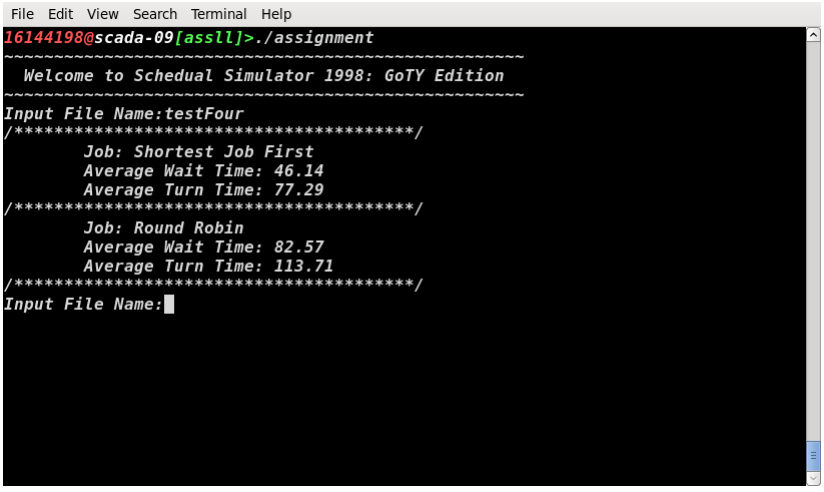
Example: testTwo



Example: testThree



Example: testFour



5 Assumptions Made

- That we were not meant to print out a Gantt Chart
- Assume when sorting, that the sort wanted was a stable sort (ie If sorting by arrival time and there were multiple with said arrival time, sort them so they mirror the files input)
- The user will not try to read a file called "QUIT" or want to read from a file longer than 10 characters (The program will handle this gracefully either way)

6 Source Code

```
#ifndef ASSIGNMENT_H
#define ASSIGNMENT_H

#define MAX_INT 99999;

#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include "scheduals.h"

void waitForThreads();
void inputName();

#endif
```

Listing 1: assignment Header File

```c
#include "assignment.h"
int main(int argc, char const *argv[])
{
  pthread_t thread1, thread2;
  int rrThread, sjfThread;

  inputFlag = 0;threadReadyFlag = 0;outputFlag = 0, quit=0;

  pthread_cond_init(&inputCond,NULL);
  pthread_cond_init(&outputCond, NULL);
  pthread_cond_init(&countCond, NULL);
  pthread_mutex_init(&inputMutex, NULL);
  pthread_mutex_init(&outputMutex, NULL);
  pthread_mutex_init(&countMutex, NULL);

  rrThread = pthread_create( &thread1, NULL, roundRobin, NULL);
    sjfThread = pthread_create( &thread2, NULL, shortetJobFirst,
    NULL);


  /**************************************
  WAIT UNTIL THREADS HAVE BEEN CREATED AND
  STARTED UP
  **************************************/
  pthread_mutex_lock(&countMutex);
  while(threadReadyFlag != 2)
  {
    pthread_cond_wait(&countCond, &countMutex);
  }
  threadReadyFlag = 0;
  pthread_mutex_unlock(&countMutex);

  /**************************************/
  printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
  printf("  Welcome to Schedual Simulator 1998: GoTY Edition\n");
  printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
  while(quit != 1)
  {
    /**************************************
    LOCK THE INPUT AND HAVE THE USER INPUT IT
    **************************************/
    pthread_mutex_lock(&inputMutex);
    do{
    printf("Input File Name:");
    scanf("%10s", inputBuffer);
    }while(!file_exist(inputBuffer) && !(strcmp(inputBuffer, "QUIT"
    ) == 0));

    inputFlag = 1;
    pthread_cond_broadcast(&inputCond);
    pthread_mutex_unlock(&inputMutex);
    /**************************************/
    if( !(strcmp(inputBuffer, "QUIT") == 0) )
    {
      /**************************************
      WAIT FOR OUTPUT TO BE FILLED, THEN LOCK
      AND REMOVE THE DATA
```

```
56        ****************************************/
57        printf("/****************************************/\n");
58        for(int ii=0; ii < 2; ii++)
59        {
60          //printf("For %d\n", ii);
61          pthread_mutex_lock(&outputMutex);
62          while(outputFlag != 1)
63          {
64            pthread_cond_wait(&outputCond, &outputMutex);
65          }
66
67          if(outputBuffer.j == 'S')
68            printf("\tJob: Shortest Job First\n");
69          else
70            printf("\tJob: Round Robin\n");
71          printf("\tAverage Wait Time: %.2f\n", outputBuffer.waitTime
    );
72          printf("\tAverage Turn Time: %.2f\n", outputBuffer.turnTime
    );
73          printf("/****************************************/\n");
74          outputFlag = 0;
75          pthread_cond_broadcast(&outputCond);
76          pthread_mutex_unlock(&outputMutex);
77        }
78        /****************************************/
79       }
80      else
81      {
82        quit = 1;
83      }
84    }
85    pthread_join(thread1, NULL);
86    pthread_join(thread2,NULL);
87    return 0;
88  }
```

Listing 2: assignment .c File

```c
#ifndef SCHEDUALS_H
#define SCHEDUALS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <pthread.h>
#include "functions.h"
#include "LinkedList.h"

typedef struct OutputBuffer{char j; double waitTime; double
    turnTime;} OutputBuffer;

pthread_cond_t inputCond;
pthread_cond_t outputCond;
pthread_cond_t countCond;
pthread_mutex_t inputMutex;
pthread_mutex_t outputMutex;
pthread_mutex_t countMutex;

int inputFlag, outputFlag,threadReadyFlag, quit;
char inputBuffer[11];
OutputBuffer outputBuffer;

void *roundRobin();
void *shortetJobFirst();

#endif
```

Listing 3: scheduals Header File

```c
#include "scheduals.h"

void *roundRobin(){
  int sumOfWait=0,sumOfTurnaround=0,earlyStart = INT_MAX;;
  int ii,time,remain,a,b,r,f, timeQuantum, numOfProcess;
  double avgWaitTime, avgTurnTime;
  char fileName[11];
  LinkedList* jobQueue, *processing, *finished;
  LinkedListNode* currJob, *checkedJob;

  jobQueue = createLinkedList();
  processing = createLinkedList();
  finished = createLinkedList();


  /*************************************
  TELL THE PARENT THAT IT IS READY BY
  INCREMENTING THE READYCOUNT
  *************************************/
  pthread_mutex_lock(&countMutex);
  threadReadyFlag = threadReadyFlag + 1;
  pthread_cond_broadcast(&countCond);
  pthread_mutex_unlock(&countMutex);
  /*************************************/


  while(quit!=1)
  {
    /*************************************
    WAIT UNTIL THE PARENT HAS INPUT
    *************************************/
    pthread_mutex_lock(&inputMutex);
    while(inputFlag != 1)
    {
      pthread_cond_wait(&inputCond, &inputMutex);
    }
    snprintf(fileName, 11, "%s", inputBuffer);
    inputFlag=2;
    pthread_cond_broadcast(&inputCond);
    pthread_mutex_unlock(&inputMutex);
    /*************************************/

    if(!(strcmp(fileName, "QUIT") == 0) )
    {
      calcProcesses(fileName,&numOfProcess);
      remain=numOfProcess;
      loadLinkedList(fileName, jobQueue, numOfProcess, &timeQuantum
    , &earlyStart);
      time=earlyStart;

      /*Check at the beginging to see if the finished queue contais
     all jobs*/
      while( getLength(finished) != numOfProcess)
      {
        /*remove a job from processing, if it has no jobs, check
    jobQueue
          and remove it from there instead*/
```

```
55          if(getLength(processing)!=0)
56            currJob = removeFirstElement(processing);
57          else if(getLength(jobQueue) != 0){
58            currJob = removeFirstElement(jobQueue);
59            time = currJob->a;
60          }
61
62          /* Decrement the burst time by timeQuantum, if it is now 0
63            put it in the finished queue */
64          if(currJob->r <= timeQuantum && currJob->r >= 0){
65            time += currJob->r;
66            currJob->r = 0;
67            a = currJob->a;
68            b = currJob->b;
69            r = currJob->r;
70            f = time;
71            insertNodeEnd(finished, a,b,r,f,currJob->pid);
72          }
73          else if(currJob->r > timeQuantum){
74            currJob->r -= timeQuantum;
75            time += timeQuantum;
76          }
77          /*  else check the rest of
78            the jobs in the job queue to see if any have come whilst
79            it was running and put them onto the processing queue */
80
81          for(int jj=1;jj<=getLength(jobQueue); jj++){
82            checkedJob = removeFirstElement(jobQueue);
83            if(checkedJob->a <= time){
84              a = checkedJob->a;
85              b = checkedJob->b;
86              r = checkedJob->r;
87              f = checkedJob->f;
88              insertNodeEnd(processing, a,b,r,f,checkedJob->pid);
89            }
90            else{
91              a = checkedJob->a;
92              b = checkedJob->b;
93              r = checkedJob->r;
94              f = checkedJob->f;
95              insertNodeEnd(jobQueue, a,b,r,f,checkedJob->pid);
96            }
97          }
98          /*then put the job that was being processed onto the
99            back of the processing queue */
100         if(currJob->r != 0){
101           a = currJob->a;
102           b = currJob->b;
103           r = currJob->r;
104           f = currJob->f;
105           insertNodeEnd(processing, a,b,r,f,currJob->pid);
106         }
107       }
108
109       /* go through the list and calculate the sum of wait
110         and turnaround time */
111       for(ii=0; ii<numOfProcess; ii++)
```

```
112         {
113           currJob = removeFirstElement(finished);
114           sumOfWait += currJob->f - currJob->b - currJob->a;
115           sumOfTurnaround += currJob->f - currJob->a;
116         }
117         avgWaitTime = ((double)sumOfWait)/((double)numOfProcess);
118         avgTurnTime = ((double)sumOfTurnaround)/((double)numOfProcess
        );
119         sumOfTurnaround = 0;
120         sumOfWait = 0;
121 
122         /*************************************
123         LOCK THE OUTPUT AND INSERT THAT INTO THE
124         BUFFER
125         *************************************/
126         pthread_mutex_lock(&outputMutex);
127 
128         while(outputFlag != 0)
129         {
130           pthread_cond_wait(&outputCond, &outputMutex);
131         }
132         outputBuffer.turnTime = avgTurnTime;
133         outputBuffer.waitTime = avgWaitTime;
134         outputBuffer.j = 'R';
135         outputFlag = 1;
136         pthread_cond_broadcast(&outputCond);
137         pthread_mutex_unlock(&outputMutex);
138         /*************************************/
139       }
140     }
141     return NULL;
142 }
143 
144 void *shortetJobFirst()
145 {
146     char fileName[11];
147     int ii, sum_burstTime=0, sumOfTurnaround = 0, sumOfWait = 0,
          earlyStart = INT_MAX;
148     int  time, smallest,remain, timeQuantum, numOfProcess;
149     double avgWaitTime, avgTurnTime;
150 
151     /*************************************
152     TELL THE PARENT THAT IT IS READY BY
153     INCREMENTING THE READYCOUNT
154     *************************************/
155     pthread_mutex_lock(&countMutex);
156     threadReadyFlag = threadReadyFlag + 1;
157     pthread_cond_broadcast(&countCond);
158     pthread_mutex_unlock(&countMutex);
159     /*************************************/
160     while(quit!=1)
161     {
162       /*************************************
163       WAIT UNTIL THE PARENT HAS INPUT
164       *************************************/
165       pthread_mutex_lock(&inputMutex);
166       while(inputFlag != 2)
```

```
167     {
168       pthread_cond_wait(&inputCond, &inputMutex);
169     }
170
171     snprintf(fileName, 11, "%s", inputBuffer);
172     inputFlag = 0;
173     pthread_cond_broadcast(&inputCond);
174     pthread_mutex_unlock(&inputMutex);
175     /*************************************/
176
177     if( !(strcmp(inputBuffer, "QUIT") == 0) )
178     {
179       earlyStart = INT_MAX;
180
181       calcProcesses(fileName,&numOfProcess);
182       int arriveTime[numOfProcess+1], burstTime[numOfProcess+1],
    burstCopy[numOfProcess+1];
183
184
185       loadFile(fileName, arriveTime, burstTime, &timeQuantum, &
    earlyStart);
186       bubbleSort(numOfProcess, arriveTime, burstTime);
187
188       memcpy(burstCopy, burstTime, (numOfProcess+1)*sizeof(int));
189       /* Set the last array value (not an actual job) to INT_MAX
190         so that the loops can find the smallest value */
191       burstCopy[numOfProcess]=INT_MAX;
192       arriveTime[numOfProcess]=INT_MAX;
193
194       time=earlyStart;
195       remain = numOfProcess;
196
197         while(remain > 0)
198         {
199         smallest=numOfProcess;
200         /* Find the next job with the smallest remaining burst time
     */
201         for(ii=0;ii<numOfProcess;ii++)
202         {
203             if(arriveTime[ii]<=time && burstCopy[ii]>0 && burstCopy
    [ii]<burstCopy[smallest])
204             smallest=ii;
205         }
206
207         /*If no jobs came in whilst the other was running, pick the
208           next one with the shortest arrival time */
209         if(smallest == numOfProcess)
210         {
211           for(ii=0;ii<numOfProcess;ii++)
212           {
213               if(burstCopy[ii]>0 && arriveTime[ii]<arriveTime[
    smallest])
214               smallest = ii;
215           }
216           time = arriveTime[smallest];
217         }
218         remain--;
```

```
219         /* re-calculate the sum of WT and TT and make the burst
    time
220            of the current job =0 */
221         sumOfTurnaround+=time+burstCopy[smallest]-arriveTime[
    smallest];
222         sumOfWait+=time-arriveTime[smallest];
223         time+=burstCopy[smallest];
224         burstCopy[smallest]=0;
225         }
226
227         avgWaitTime = sumOfWait*1.0/numOfProcess;
228       avgTurnTime = sumOfTurnaround*1.0/numOfProcess;
229       sumOfTurnaround = 0;
230       sumOfWait = 0;
231       sum_burstTime = 0;
232       numOfProcess = 0;
233       /*************************************
234       LOCK THE OUTPUT AND INSERT THAT INTO THE
235       BUFFER
236       *************************************/
237       pthread_mutex_lock(&outputMutex);
238
239       while(outputFlag != 0)
240       {
241         pthread_cond_wait(&outputCond, &outputMutex);
242       }
243
244       outputBuffer.turnTime = avgTurnTime;
245       outputBuffer.waitTime = avgWaitTime;
246       outputBuffer.j = 'S';
247
248       outputFlag = 1;
249       pthread_cond_broadcast(&outputCond);
250       pthread_mutex_unlock(&outputMutex);
251       /*************************************/
252     }
253   }
254   return NULL;
255 }
```

Listing 4: scheduals .c File

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "LinkedList.h"

void calcProcesses(char*, int*);
void bubbleSort(int, int[], int[]);
int file_exist (char*);
void loadLinkedList(char*, LinkedList*, int, int *, int*);
void loadFile(char*, int[], int[], int*, int*);

#endif
```

Listing 5: functions Header File

```c
#include "functions.h"

void calcProcesses(char* name, int* numOfProcess){
  FILE *read = fopen(name, "r");
  int ii =0, x;

  fscanf(read, "%d", &x);
  while (fscanf(read, "%d", &x) == 1){
    ii++;
  }
  fclose(read);
  *numOfProcess = (ii/2);
}
/*******************************************/
/*    A stable sort algorithm used to sort
    the data before putting it into
    a linkedlist                 */
/*******************************************/
void bubbleSort(int numOfProcess, int arriveTime[], int burstTime
    [])
{
  int length = numOfProcess - 1, sorted = 0, pass = 0, temp;

  while(sorted != 1){
    sorted = 1;
    for(int ii=0;ii<(length-pass);ii++)
    {
      if( arriveTime[ii] > arriveTime[ii+1])
      {
        temp = arriveTime[ii];
        arriveTime[ii] = arriveTime[ii+1];
        arriveTime[ii+1]=temp;

        temp = burstTime[ii];
        burstTime[ii] = burstTime[ii+1];
        burstTime[ii+1]=temp;
        sorted = 0;
      }
    }
    pass++;
  }
}

/*******************************************/
/*  Takes the sorted data and load it into
  the a linkedlist                */
/*******************************************/

void loadLinkedList(char* name, LinkedList* jobQueue, int
    numOfProcess, int *timeQuantum, int* earlyStart)
{

  int arriveTime[numOfProcess], burstTime[numOfProcess];
  int a,b,r,f, ii=0;
  char pid;

  FILE *read = fopen(name, "r");
```

```
  int x, y, jj=0;
  *earlyStart = 999999;
  fscanf(read, "%d", timeQuantum);
  while (fscanf(read, "%d %d", &x, &y) == 2){
    arriveTime[jj]=x;
    burstTime[jj]=y;

    if(x <= *earlyStart)
      *earlyStart=x;
    jj++;
  }
  fclose(read);

  bubbleSort(numOfProcess, arriveTime, burstTime);


  for(ii=0;ii<numOfProcess;ii++)
  {
    a = arriveTime[ii];
    b = burstTime[ii];
    r = b;
    f = 0;
    pid = 'A'+ ii;
    insertNodeEnd(jobQueue, a, b, r, f, pid);
  }
}

void loadFile(char* name, int arriveTime[], int burstTime[], int *
    timeQuantum, int* earlyStart)
{
  FILE *read = fopen(name, "r");
  int x=999999, y, jj=0;

  fscanf(read, "%d", timeQuantum);
  while (fscanf(read, "%d %d", &x, &y) == 2){
    arriveTime[jj]=x;
    burstTime[jj]=y;

    if(x <= *earlyStart)
      *earlyStart=x;
    jj++;
  }
  fclose(read);
}

/*******************************************/
/*  Returns 1 or 0 depending on if the
  file exists or not                */
/*******************************************/
int file_exist (char *filename)
{
  struct stat buffer;
  return (stat(filename, &buffer) == 0);
}
```

Listing 6: functions .c File

```c
#ifndef _LINKEDLIST_H_
#define _LINKEDLIST_H_


/*****************************
* LinkedListNode Struct
* Contains a pointer to another linked list node and a void pointer
     to some sort of data
*****************************/
typedef struct LinkedListNode { int a; int b; int r; int f; char
    pid; struct LinkedListNode *next;} LinkedListNode;

/*****************************
* LinkedList Struct
* Contains a pointer to a linked list node
*****************************/

typedef struct LinkedList { LinkedListNode *head; } LinkedList;

LinkedList* createLinkedList();
void insertNodeEnd(LinkedList *, int,int,int,int, char );
LinkedListNode* removeFirstElement(LinkedList *);
int getLength(LinkedList *);
void freeLinkedList(LinkedList*);

#endif
```

Listing 7: LinkedList header File

```c
#include <stdio.h>
#include <stdlib.h>
#include "LinkedList.h"

LinkedList* createLinkedList()
{
  LinkedList *list;

  list = (LinkedList*)malloc(sizeof(LinkedList));
  list->head = NULL;
  return list;
}

void insertNodeEnd(LinkedList *list, int a,int b,int r,int f, char
    pid)
{
  LinkedListNode *newNode = NULL, *tempNode = NULL;

  newNode = (LinkedListNode*)malloc(sizeof(LinkedListNode));
  newNode->next = NULL;
  newNode->a = a;
  newNode->b = b;
  newNode->r = r;
  newNode->f = f;
  newNode->pid = pid;

  tempNode = list->head;
  /*Check if the list is empty*/
  if(list->head == NULL)
  {
    list->head = newNode;
  }
  /*If it is not, then itterate over the list*/
  else{
  while( (tempNode->next) != NULL)
  {
    tempNode = tempNode->next;
  }

  tempNode->next = newNode;
  }
}

LinkedListNode* removeFirstElement(LinkedList *list)
{
  LinkedListNode *temp = NULL;

  temp = list->head;
  list->head = temp->next;
  return temp;
}

int getLength(LinkedList *list)
{

  int length = 0;
  LinkedListNode *current;
```

```
57
58    current = list->head;
59    while(current != NULL) {
60      length++;
61      current = (*current).next;
62    }
63    return length;
64  }
65
66
67  void freeLinkedList(LinkedList *list)
68  {
69    LinkedListNode *node, *nextNode;
70    node = list->head;
71    while(node != NULL) {
72      nextNode = (node->next);
73      free(node);
74      node = nextNode;
75    }
76    list->head=NULL;
77  }
```

Listing 8: LinkedList .c File