Report

Author: Karthik Karavatt

StudentID: 20619965

This report contains the software solution of the assignment. It used showcase and explain the coding decisions I have made.

It will be split into the following parts

- Code section
- Synchronization Discussion
- Tests and inconsistencies
- Sample input and output

Code

main.c

```c
//Name: Karthik Karavatt
//StudentID: 20619965
#include "assignmentMethods.h"
#include "linkedList.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  //global variable from assignmentMethods.c
  extern pthread_mutex_t writeToLog;
  extern pthread_mutex_t listLock;
  extern pthread_mutex_t fileLock;
  extern pthread_cond_t cond;
  extern pthread_cond_t queueFull;
  extern pthread_cond_t continueOperation;
  //command line arguments
  int m = atoi(argv[1]);
  int t_c = atoi(argv[2]);
  int t_w = atoi(argv[3]);
  int t_d = atoi(argv[4]);
  int t_i = atoi(argv[5]);
  // M has to be greater than 0 for the program to work
  // Otherwise it will not make sense
  if(m <= 0){
```

```
    exit(0);
}
pthread_t id, t1, t2, t3, t4;
CustomerArgs args;
Teller teller1, teller2, teller3, teller4;
//using linked list made in USP to act as a queueFull
//The only methods that will be used is insert last and remove first
//Therefore it is equivilent to a queue
LinkedList *c_queue = createList();
//assigning tellers variables
teller1.id = "1";
teller1.t_i = t_i;
teller1.m = m;
teller1.t_d = t_d;
teller1.t_w = t_w;
teller1.list = c_queue;
teller1.served = 0;
teller2.id = "2";
teller2.m = m;
teller2.t_i = t_i;
teller2.t_d = t_d;
teller2.t_w = t_w;
teller2.list = c_queue;
teller2.served = 0;
teller3.id = "3";
teller3.m = m;
teller3.t_i = t_i;
teller3.t_d = t_d;
teller3.t_w = t_w;
teller3.list = c_queue;
teller3.served = 0;
teller4.id = "4";
teller4.m = m;
teller4.t_i = t_i;
teller4.t_d = t_d;
teller4.t_w = t_w;
teller4.list = c_queue;
teller4.served = 0;
//arguments for the customer thread
args.list = c_queue;
args.t_c = t_c;
args.m = m;
// thread initalization
pthread_mutex_init(&listLock, NULL);
pthread_mutex_init(&writeToLog, NULL);
pthread_mutex_init(&fileLock, NULL);
```

```
  pthread_cond_init(&cond, NULL);
  pthread_cond_init(&queueFull, NULL);
  //thread creation
  pthread_create(&id, NULL, customer, (void *)&args);
  pthread_create(&t1, NULL, teller, (void *)&teller1);
  pthread_create(&t2, NULL, teller, (void *)&teller2);
  pthread_create(&t3, NULL, teller, (void *)&teller3);
  pthread_create(&t4, NULL, teller, (void *)&teller4);
  //thread join
  pthread_join(id, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  pthread_join(t4, NULL);
  // freeing resources
  freeList(c_queue);
  pthread_mutex_destroy(&listLock);
  pthread_mutex_destroy(&writeToLog);
  pthread_cond_destroy(&cond);
  pthread_cond_destroy(&queueFull);
  return EXIT_SUCCESS;
}
```

assignmentMethods.c

```
//Name: Karthik Karavatt
//StudentID: 20619965
#include "assignmentMethods.h"
#include "linkedList.h"
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

// Globals
pthread_mutex_t writeToLog;
pthread_mutex_t listLock;
pthread_mutex_t fileLock;
pthread_cond_t cond;
pthread_cond_t queueFull;
int fileread = 0; // Indicates if all customers have been dealt with
int served[4]; // array holding info about how many cusomers a teller has served
int tellersLeft = 4; // how many tellers are still running (not terminated)
```

```c
// logs a string to the log file
void logTofile(char *message) {
  FILE *file;
  file = fopen("r_log", "a");
  fprintf(file, "%s", message);
  fflush(file);
  fclose(file);
}
// log some common infor about the customer
void logCustomer(char *customerString, char *serviceString, char *onlyTime) {
  logTofile("--------------------------------------\n");
  logTofile(customerString);
  logTofile(": ");
  logTofile(serviceString);
  logTofile("\n");
  logTofile("Arival Time: ");
  logTofile(onlyTime);
  logTofile("--------------------------------------\n");
}
// add customer from the file to the queue
void addCustomer(LinkedList *list, char line[], int t_c) {
  time_t curTime;
  struct tm *timeString;
  char *customerString;
  char *serviceString;
  char *splitString;
  Customer *customer = malloc(sizeof(Customer));
  int index = 0;
  // split a line from the file into sub strings diliter is the " "
  splitString = strtok(line, " \n\0");
  // itterate through split string
  while (splitString != NULL) {
    if (index == 0) {
      customerString = splitString;
      customer->number = (char *)malloc(strlen(customerString) + 1);
      strcpy(customer->number, customerString);
    } else if (index == 1) {
      serviceString = splitString;
      serviceString[strcspn(serviceString, "\n")] = 0;
      customer->service = *splitString;
    }
    index++;
    splitString = strtok(NULL, " ");
  }
  time(&curTime);
  timeString = localtime(&curTime);
```

```
  char *onlyTime = (char *)malloc((101) * sizeof(char));
  sprintf(onlyTime, "%d:%d:%d\n", timeString->tm_hour, timeString->tm_min,
          timeString->tm_sec);
  customer->arivalTime = onlyTime;
  insertLast(list, (void *)customer);
  // lock the opertion to write to the file
  pthread_mutex_lock(&writeToLog);
  logCustomer(customerString, serviceString, onlyTime);
  // unlock the opertion to write to the file
  pthread_mutex_unlock(&writeToLog);
}

// customer function
void *customer(void *data) {
  CustomerArgs *args = (CustomerArgs *)data;
  LinkedList *list = args->list;
  int t_c = args->t_c;
  int m = args->m;
  FILE *fptr;
  char line[50];
  fptr = fopen("c_file", "r");
  // itterates through the whole file
  while (fgets(line, sizeof(line), fptr)) {
    // lock the access to the linked list
    pthread_mutex_lock(&listLock);
    if (list->size == m) {
      // wiatis until the queue is empty before adding more customers
      pthread_cond_wait(&queueFull, &listLock);
    }
    addCustomer(list, line, t_c);
    //signal that customer has been added
    pthread_cond_signal(&cond);
    // unlock the access to the queue
    pthread_mutex_unlock(&listLock);
    // sleeps
    sleep(t_c);
    // locks the list again
  }
  fclose(fptr);
  pthread_mutex_lock(&listLock);
  // if the whole file has been read, customer will signal the teller until
  // queue is empty this is so there is no dead lock between the tellers Because
  // they are waiting for the signal from the customer
  while (list->size != 0) {
    pthread_cond_signal(&cond);
    pthread_cond_wait(&queueFull, &listLock);
```

```
  }
  pthread_mutex_unlock(&listLock);
  // changing the file read value indicates to tellers they shoud terminate
  pthread_mutex_lock(&fileLock);
  fileread = 1;
  pthread_mutex_unlock(&fileLock);
  // broadcast signals all tellers that they should terminate
  pthread_cond_broadcast(&cond);
  return EXIT_SUCCESS;
}

// free all customers left in queue
void *teller(void *data) {
  pthread_mutex_lock(&fileLock);
  time_t completeionT, responseT;
  struct tm *completeionString, *responseString;
  Teller *teller = (Teller *)data;
  LinkedList *list = teller->list;
  int m = teller->m;
  while (fileread == 0) {
    pthread_mutex_unlock(&fileLock);
    pthread_mutex_lock(&listLock);
    if (list->size == 0) {
      // wait for a signal from the customer when the queue is full
      pthread_cond_wait(&cond, &listLock);
    } else {
      Customer *customer = (Customer *)removeFirst(list);
      // signal to the customer function that a customer has been removed
      pthread_cond_signal(&queueFull);
      pthread_mutex_unlock(&listLock);
      teller->served += 1;
      served[atoi(teller->id) - 1] = teller->served;
      responseT = time(&responseT);
      responseString = localtime(&responseT);
      char responseTime[100];
      sprintf(responseTime, "%d:%d:%d\n", responseString->tm_hour,
              responseString->tm_min, responseString->tm_sec);
      pthread_mutex_lock(&writeToLog);
      logTofile("Teller: ");
      logTofile(teller->id);
      logTofile("\n");
      logTofile("Customer: ");
      logTofile(customer->number);
      logTofile("\n");
      logTofile("Arrival time: ");
      logTofile(customer->arivalTime);
```

```
  logTofile("Response time: ");
  logTofile(responseTime);
  // unlocks when customer is being servieced
  pthread_mutex_unlock(&writeToLog);
  switch (customer->service) {
  case 'W':
    sleep(teller->t_w);
    break;
  case 'D':
    sleep(teller->t_d);
    break;
  case 'I':
    sleep(teller->t_i);
    break;
  }
  char completeionTime[100];
  completeionT = time(&responseT);
  completeionString = localtime(&responseT);
  sprintf(completeionTime, "%d:%d:%d\n", completeionString->tm_hour,
          completeionString->tm_min, completeionString->tm_sec);
  pthread_mutex_lock(&writeToLog);
  logTofile("Teller: ");
  logTofile(teller->id);
  logTofile("\n");
  logTofile("Customer: ");
  logTofile(customer->number);
  logTofile("\n");
  logTofile("Arrival time: ");
  logTofile(customer->arivalTime);
  logTofile("Completion time: ");
  logTofile(completeionTime);
  pthread_mutex_unlock(&writeToLog);
  // Acessing fileread vairable is locked now
  pthread_mutex_lock(&fileLock);
  // locks it again
  pthread_mutex_lock(&listLock);
  free(customer->arivalTime);
  free(customer->number);
  free(customer);
}
// check if the teller is the last one left
// list lock is still enabled here so checking tellers left is okay
if (tellersLeft == 1) {
  char served1[100];
  char served2[100];
  char served3[100];
```

```
      char served4[100];
      pthread_mutex_lock(&writeToLog);
      logTofile("Teller 1 serverd: ");
      sprintf(served1, "%d\n", served[0]);
      logTofile(served1);
      logTofile("Teller 2 serverd: ");
      sprintf(served2, "%d\n", served[1]);
      logTofile(served2);
      logTofile("Teller 3 serverd: ");
      sprintf(served3, "%d\n", served[2]);
      logTofile(served3);
      logTofile("Teller 4 serverd: ");
      sprintf(served4, "%d\n", served[3]);
      logTofile(served4);
      pthread_mutex_unlock(&writeToLog);
    }
    pthread_mutex_unlock(&listLock);
  }
  // decremetns this value when a teller terminates
  tellersLeft--;
  pthread_mutex_unlock(&listLock);
  pthread_mutex_unlock(&fileLock);
  return EXIT_SUCCESS;
}
```

assignmentMethods.h

```
#pragma once
#include "linkedList.h"

typedef struct Customer {
  char* number;
  char service;
  char* arivalTime;
} Customer;

typedef struct CustomerArgs {
  LinkedList *list;
  int t_c;
  int m;
} CustomerArgs;

typedef struct Teller{
    LinkedList* list;
    char* id;
    int m;
    int t_w;
```

```
    int t_d;
    int t_i;
    int served;
}Teller;

void logTofile(char *message);
void printCustomer(void *data);
void *customer(void *data);
void *teller(void* data);
```

linkedList.c

```
/*author Karthik Karavatt
 * StudentID 20619965 */
/* This linked list will act like a queue*/
// Created when doing a USP practical
#include "linkedList.h"
#include <stdio.h>
#include <stdlib.h>

LinkedList *createList() {
  LinkedList *list = (LinkedList *)malloc(sizeof(LinkedList));
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;
  return list;
}

void insertFirst(LinkedList *list, void *data) {
  Node *node = (Node *)malloc(sizeof(Node));
  node->data = data;
  node->next = NULL;
  /* If list is empty */
  if (list->head == NULL) {
    node->next = list->tail;
    list->head = node;
  }
  /* if the list has one item*/
  else if (list->head->next == NULL) {
    Node *temp = list->head;
    temp->before = node;
    list->head = node;
    node->next = temp;
    list->tail = temp;

  } else {
    Node *temp = list->head;
```

```
    temp->before = node;
    list->head = node;
    node->next = temp;
  }
  list->size++;
}

void insertLast(LinkedList *list, void *data) {
  Node *currentTail;
  Node *node = (Node *)malloc(sizeof(Node));
  node->data = data;
  node->next = NULL;
  /*if list is empty */
  if (list->head == NULL) {
    list->head = node;
    list->head->next = list->tail;
  } else {
    currentTail = list->tail;
    /*if there is only one node */
    if (currentTail == NULL) {
      list->head->next = node;
      node->before = list->head;
      list->tail = node;
    } else {
      currentTail->next = node;
      node->before = currentTail;
      list->tail = node;
    }
  }
  list->size++;
}
void *removeFirst(LinkedList *list) {
  void *data;
  Node *node;
  /*check if list is empty*/
  if (list->head != NULL) {
    node = list->head;
    data = node->data;
    /*check if there is only one node*/
    if (list->tail == NULL) {
      list->head = NULL;
    }
    /*check if there is only two node*/
    else if (list->head->next == list->tail) {
      list->head = list->tail;
      list->head->before = NULL;
```

```
      list->tail = NULL;
    } else {
      data = list->head->data;
      list->head = list->head->next;
      list->head->before = NULL;
    }
    free(node);
    list->size--;
  }
  return data;
}
void *removeLast(LinkedList *list) {
  void *data;
  Node *node;
  /* check if list is empty */
  if (list->head != NULL) {
    /* check if there is more than one node */
    if (list->tail != NULL) {
      node = list->tail;
      data = node->data;
      node->before->next = NULL;
      /* check if there are only two node */
      if (node->before == list->head) {
        list->tail = NULL;
      } else {
        list->tail = node->before;
      }
      node = NULL;
    }
    /* If there is only one node */
    else {
      node = list->head;
      data = node->data;
      list->head = NULL;
    }
    free(node);
    list->size--;
  }
  return data;
}

void printList(LinkedList *list, listFunc func) {
  Node *node = list->head;
  if (node != NULL) {
    while (node != NULL) {
      func(node->data);
```

```
      node = node->next;
    }
    node = NULL;
  }
}

void freeNode(LinkedList *list) {
  Node *curNode = list->head;
  Node *nextNode;
  while (curNode != NULL) {
    nextNode = curNode->next;
    free(curNode);
    curNode = nextNode;
    list->size--;
  }
}

void freeList(LinkedList *list) {
  freeNode(list);
  free(list);
}
```

linkedList.h

```
#pragma once
typedef struct Node
{
   void* data;
   struct Node* before;
   struct Node* next;
}Node;
typedef struct LinkedList
{
   Node* head;
   Node* tail;
   int size;
}LinkedList;
typedef void(*listFunc)(void* data);
LinkedList* createList(void);
void insertFirst(LinkedList* list, void* data);
void insertLast(LinkedList* list, void* data);
void* removeFirst(LinkedList* list);
void* removeLast(LinkedList* list);
void printList(LinkedList* list, listFunc func);
void freeList(LinkedList* list);
```

makefile

```
CC = gcc
LD = gcc
CFLAGS = -g -pthread
LFLAGS = -lm -s
OBJ = main.o linkedList.o assignmentMethods.o
EXEC = main
$(EXEC): $(OBJ)
    $(LD) $(CFLAGS) $(OBJ) -o $(EXEC)

main.o: main.c linkedList.h assignmentMethods.h
    $(CC) -c main.c $(CFLAGS)

linkedList.o: linkedList.c
    $(CC) -c linkedList.c $(CFLAGS)

assignmentMethods.o: assignmentMethods.c linkedList.h
    $(CC) -c assignmentMethods.c $(CFLAGS)

clean:
    $(RM) $(EXEC) $(OBJ)

val:
    valgrind --leak-check=full -s ./$(EXEC) 3 1 1 1 1
run:
    ./main 3 1 1 1 1
gdb:
    gdb --args main 2 1 1 1 1

# DO NOT DELETE
```

Synchronization Discussion

To understand how the synchronization works, lets see what variables are being shared.

The c_queue is the main shared variable in the program. It is shared by the following functions:

- customer
- teller

The customer runs on one thread, whereas, the teller runs on 4 threads. So 5 threads need access to the c_queue.

The r_log file is also a shared variable in the program. It is represented as the file variable in the logTofile function.

This is also shared by the customer and teller functions. Therefore 5 threads

require access to this variable.

The fileread variable located in assignmentMethods.c is also shared among all 5 threads. Although, only the customer thread can modify it, teller threads can only read it. This variable indicated when the c_file has been fully read.

The tellersLeft integer , is another shared variable only accessed and modified by teller threads. It indicates how many tellers are still executing.

The server array, is shared by all teller threads. This indicates the number of customers served by each teller thread.

To achieve synchronization for the c_queue, the mutex lock listLock was used. If a thread needed to access or modify the c_queue, the lock must be enabled first. Then after the use it must be unlocked. For example in the customer function

assignmentMethods.c, customer

```
pthread_mutex_lock(&listLock);
if (list->size == m) {
  // wiatis until the queue is empty before adding more customers
  pthread_cond_wait(&queueFull, &listLock);
}
addCustomer(list, line, t_c);
//signal that a customer has been added
pthread_cond_signal(&cond);
// unlock the access to the queue
pthread_mutex_unlock(&listLock);
```

When the customer thread wants to check the size of the c_queue (called list in this function), it enables the listLock by calling `pthread_mutex_lock(listLock)`. This lock is also enabled when a customer is added to the c_queue, using the `addCustomer()` function. After which it is unlocked using the `pthread_mutex_unlock(&listLock)` method.

This example also highlights another way synchronization is achieved, through the use of `pthread_cond_signal()` and `pthread_cond_wait()`. When `list->size == m` it indicates that the c_queue is full. When this situation occurs, the customer thread must wait until it gets the queueFull signal.

The queueFull signal is emitted by the teller thread. When a customer is removed from the list

assignmentMethods.c, teller

```
Customer *customer = (Customer *)removeFirst(list);
// signal to the customer function that a customer has been removed
pthread_cond_signal(&queueFull);
```

When a customer is removed from the c_queue, the signal is emitted, this stops the customer thread from waiting and it continues execution.

Similarly, the teller thread also will be put into a waiting state when the c_queue is full.

assignmentMethods.c, teller

```
if (list->size == 0) {
  // wait for a signal from the customer when the queue is full
  pthread_cond_wait(&cond, &listLock);
}
```

The teller thread can break out of the wait when the customer tread adds a customer to the list. It is important to note that the listLock is still locked before the `pthread_cond_wait()` is called. But after it is called, the mutex lock is unlocked until the signal is received from the customer thread.

assignmentMethods.c, customer

```
addCustomer(list, line, t_c);
//signal that customer has been added
pthread_cond_signal(&cond);
```

When a customer is added to the c_queue, the cond signal will be emitted, which breaks the wait condition of the teller thread, as the queue is not empty.

This prevents deadlocks from occurring as there can never be situation where the c_queue is both empty and full at the same time. And the size value read by each tread is always accurate as a mutex lock is enabled before accessing them. ## Tests and inconsistencies

Sample input and output