Curtin University

Department of Electrical Engineering, Computing, and Mathematical Sciences.

Operating System Assignment

Dicky Larson Gultom

19487537

Source Code

Main Driver File

```
#include "main.h"

#define NUM_TELLER 4
pthread_cond_t conditionArrayFULL = PTHREAD_COND_INITIALIZER;
pthread_cond_t conditionEmpty = PTHREAD_COND_INITIALIZER;
pthread_mutex_t fileMutex;

FILE *r_log;

int main(int argc, char *argv[])
{
  r_log = fopen("r_log.txt","w");
  //create and initialise mutex;
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex,NULL);
  pthread_mutex_init(&fileMutex,NULL);
  //this customerThread will be responsible for enqueue in customer on a fifo queue in a set of
interval.
  pthread_t customerThread;
  //this is the teller tread
  pthread_t teller[NUM_TELLER];

  customerQueue *queue = malloc(sizeof(customerQueue));

  queue->size=0;

  informationCustomerThread *infoCustThread = malloc(sizeof(informationCustomerThread));
  tellerStats *tellerObject = malloc(sizeof(tellerStats)*NUM_TELLER);

  //time keeping purpose
  time_t timeEpoch;
  struct tm *startTime, *finishTime;
  startTime = (struct tm*)malloc(sizeof(struct tm));
  finishTime = (struct tm*)malloc(sizeof(struct tm));

  int lengthQueue, periodicTime, tw, td, ti;
  if (argc == 6)
  {
    lengthQueue = atoi(argv[1]);
    periodicTime = atoi(argv[2]);
    tw = atoi(argv[3]);
    td = atoi(argv[4]);
    ti = atoi(argv[5]);

    queue->mutex = mutex;

    customerNode *data = malloc(sizeof(customerNode)*lengthQueue);
    infoCustThread->periodicTime = periodicTime;
    infoCustThread->queue = queue;
    infoCustThread->space = lengthQueue;

    queue->data = data;
    queue->MAX_SIZE = lengthQueue;

    for(int i = 0; i<NUM_TELLER; i++)
    {
      time(&timeEpoch);
      localtime_r(&timeEpoch,startTime);
      tellerObject[i].sHr = startTime->tm_hour;
      tellerObject[i].sMin = startTime->tm_min;
      tellerObject[i].sSec = startTime->tm_sec;
      tellerObject[i].tellerNames = i+1;
      tellerObject[i].mutex = mutex;
      tellerObject[i].ti = ti;
      tellerObject[i].tw = tw;
      tellerObject[i].td = td;
      tellerObject[i].queue = queue;
      pthread_create(&teller[i],NULL,Bank,&tellerObject[i]);
    }

    pthread_create(&customerThread, NULL, customer,(void*)infoCustThread);

    for(int i = 0; i<NUM_TELLER; i++)
    {
      pthread_join(teller[i],NULL);
      //when the thread has joined then we can call time again
      time(&timeEpoch);
      localtime_r(&timeEpoch,finishTime);
      tellerObject[i].fHr = finishTime->tm_hour;
      tellerObject[i].fMin = finishTime->tm_min;
      tellerObject[i].fSec = finishTime->tm_sec;

      //write it into the file
      fprintf(r_log,"--------------------------\n");
      fprintf(r_log,"Termination: teller-%d\n",tellerObject[i].tellerNames);
      fprintf(r_log,"#served customers: %d\n",tellerObject[i].nCustomer);
      fprintf(r_log,"Start
time: %02d:%02d:%02d\n",tellerObject[i].sHr,tellerObject[i].sMin,tellerObject[i].sSec);
      fprintf(r_log,"Termination
time: %02d:%02d:%02d\n",tellerObject[i].fHr,tellerObject[i].fMin,tellerObject[i].fSec);
      fprintf(r_log,"--------------------------\n");

    }
    printf("Teller Statistic\n");
    printf("Teller-1 serves %d customers\n",tellerObject[0].nCustomer);
    printf("Teller-2 serves %d customers\n",tellerObject[1].nCustomer);
    printf("Teller-3 serves %d customers\n",tellerObject[2].nCustomer);
    printf("Teller-4 serves %d customers\n",tellerObject[3].nCustomer);
    int total = tellerObject[0].nCustomer +
tellerObject[1].nCustomer+tellerObject[2].nCustomer+tellerObject[3].nCustomer;
    printf("Total Customers Served: %d\n",total);


    pthread_join(customerThread,NULL);

    free(data);
  }
  else
  {
    printf("Insufficient Arguments\n");
  }

  //destroy the mutex
  pthread_mutex_destroy(&mutex);
  pthread_mutex_destroy(&fileMutex);

  //free the remaining memory allocated
  free(queue);
  free(tellerObject);
  free(infoCustThread);
  //free the time keeping struct
  free(startTime);
  free(finishTime);
  return 0;
}

void *customer(void* arg)
{
  informationCustomerThread *infoCustThread = (informationCustomerThread*)arg;
  int periodicTime = (int)infoCustThread->periodicTime;
  int queueSpace = (int)infoCustThread->space;
  customerQueue *Queue = infoCustThread->queue;

  time_t rawtime;
  struct tm *accurateTime = (struct tm*)malloc(sizeof(struct tm));
  char mode;
  int position;

  FILE *inputFile = fopen("c_file","r");
  if (inputFile != NULL)
  {
    while(!feof(inputFile))
    {
      if(fscanf(inputFile,"%d %c", &position, &mode) == 2)
      {
        if(validateInput(mode)==1)
        {
          pthread_mutex_lock(&Queue->mutex);
          //check if the full
          while(Queue->size==queueSpace)
          {
            //wait until a customer is served
            pthread_cond_wait(&conditionArrayFULL,&Queue->mutex);
          }

          pthread_mutex_lock(&fileMutex);

          time(&rawtime);
          localtime_r(&rawtime,accurateTime);

          fprintf(r_log,"--------------------------\n");
          fprintf(r_log, "Customer: %d -> %c\n",position,mode);
          fprintf(r_log, "Arrival Time: %02d:%02d:%02d\n",accurateTime->tm_hour, accurateTime-
>tm_min, accurateTime->tm_sec);
          fprintf(r_log,"--------------------------\n");
          fprintf(r_log,"\n");
          pthread_mutex_unlock(&fileMutex);
          (Queue->data[Queue->size]).customerNumber = position;
          (Queue->data[Queue->size]).serviceType = mode;
          (Queue->data[Queue->size]).aHr= accurateTime->tm_hour;
          (Queue->data[Queue->size]).aMin= accurateTime->tm_min;
          (Queue->data[Queue->size]).aSec= accurateTime->tm_sec;
          Queue->size++;

          pthread_cond_signal(&conditionEmpty);
          pthread_mutex_unlock(&Queue->mutex);
        }
        else
        {
          printf("Invalid Transaction\n");
        }
      }
      else
      {
        printf("Invalid Input\n");
      }
    }
    sleep(periodicTime);
  }
  else
  {
    printf("Cannot Find Any File. Exiting\n");
  }

  return NULL;
}


void *Bank(void *arg)
{
```

```
time_t rawtime;
static struct tm *accurateTime,*accurateFinish;
accurateTime = (struct tm*)malloc(sizeof(struct tm));
accurateFinish = (struct tm*)malloc(sizeof(struct tm));
tellerStats *teller = (tellerStats*)arg;
customerQueue *queue = teller->queue;
int numberofCustomerServed = 0;
//pthread_t ID = pthread_self();

//the prefix "a" will represent arrival time;
int aHr,aMin,aSec;
//the prefix "r" will represent response time
int rHr,rMin,rSec;
//the prefix "c" will represent completion time
int cHr,cMin,cSec;

customerNode customer;
pthread_mutex_lock(&teller->mutex);
while(queue->size == 0)
{
  pthread_cond_wait(&conditionEmpty,&teller->mutex);
}
pthread_mutex_unlock(&teller->mutex);
while(queue->size > 0)
{

  pthread_mutex_lock(&teller->mutex);
  customer = queue->data[0];

  aHr = customer.aHr;
  aMin = customer.aMin;
  aSec = customer.aSec;

  //budget-style dequeue
  for(int i=0; i<queue->size-1;i++)
  {
    queue->data[i] = queue->data[i+1];
  }
  queue->size--;
  pthread_cond_signal(&conditionArrayFULL);
  pthread_mutex_unlock(&teller->mutex);


  //lock the file
  pthread_mutex_lock(&fileMutex);
  time(&rawtime);
  localtime_r(&rawtime,accurateTime);
  rHr = accurateTime->tm_hour;
  rMin = accurateTime->tm_min;
  rSec = accurateTime->tm_sec;
```

```
   //writing into the file
   fprintf(r_log,"--------------------------\n");
   fprintf(r_log,"Teller: %d\n",teller->tellerNames);
   fprintf(r_log,"Customer: %d\n",customer.customerNumber);
   fprintf(r_log,"Arrival Time: %02d:%02d:%02d\n",aHr,aMin,aSec);
   fprintf(r_log,"Response Time: %02d:%02d:%02d\n",rHr,rMin,rSec);
   fprintf(r_log,"--------------------------\n");
   fprintf(r_log,"\n");

   pthread_mutex_unlock(&fileMutex);

   //serving simulation
   switch(tolower(customer.serviceType))
   {
     case 'd':
       sleep(teller->td);
       break;
     case 'i':
       sleep(teller->ti);
       break;
     case 'w':
       sleep(teller->tw);
       break;
     default:
       sleep(1);
       break;
   }

   //after simulation, the thread will once again write it on the file
   pthread_mutex_lock(&fileMutex);
   time(&rawtime);
   localtime_r(&rawtime,accurateFinish);

   cHr =  accurateFinish->tm_hour;
   cMin = accurateFinish->tm_min;
   cSec = accurateFinish->tm_sec;

   fprintf(r_log,"--------------------------\n");
   fprintf(r_log,"Teller: %d\n",teller->tellerNames);
   fprintf(r_log,"Customer: %d\n",customer.customerNumber);
   fprintf(r_log,"Arrival Time: %02d:%02d:%02d\n",aHr,aMin,aSec);
   fprintf(r_log,"Response Time: %02d:%02d:%02d\n",rHr,rMin,rSec);
   fprintf(r_log,"Completion Time: %02d:%02d:%02d\n",cHr,cMin,cSec);
   fprintf(r_log,"--------------------------\n");
   fprintf(r_log,"\n");
   pthread_mutex_unlock(&fileMutex);
   numberofCustomerServed++;

 }
 teller->nCustomer=numberofCustomerServed;
```

```
  return NULL;
}

int validateInput(char mode)
{
  int result;
  mode = tolower(mode);
  if(mode == 'w' || mode == 'd' || mode == 'i')
  {
    result = 1;
  }
  else
  {
    result = 0;
  }

  return result;
}
```

Header File

```
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <stdint.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>

typedef struct customerNode
{
   int customerNumber;
   char serviceType;
   int aHr;
   int aMin;
   int aSec;
}customerNode;

typedef struct customerQueue
{
   customerNode *data;
   int size;
   int MAX_SIZE;
   pthread_mutex_t mutex;
}customerQueue;

typedef struct tellerStats
{
   int tellerNames;
```

```
    int ti;
    int tw;
    int td;
    //time keeping purpose .. start time
    int sHr;
    int sMin;
    int sSec;
    //time keeping purpose ... finish time
    int fHr;
    int fMin;
    int fSec;
    customerQueue *queue;
    int nCustomer;
    pthread_mutex_t mutex;

}tellerStats;

typedef struct informationCustomerThread{
    int periodicTime;
    int space;
    customerQueue *queue;
}informationCustomerThread;

void *customer(void *arg);
void *Bank(void *arg);
customerNode dequeue(customerQueue *queue);
int validateInput(char mode);
```

Thread Synchronisation

In this program, the four main thread synchronisation functions used are *pthread_mutex_lock()*, *pthread_mutex_unlock()*, *pthread_cond_wait(),* and *pthread_cond_signal()*. The customer queue and the file pointer are the resources shared between all four teller threads, and the customer thread is responsible for putting the customer into the queue. To ensure the integrity of the data, only one thread must exclusively access the resource. This can be achieved by calling the mutex lock function to acquire the mutex lock and release the lock once the thread is finished accessing the shared resource.

In my implementation, there are two threads conditions that will be used in the wait function and there are also two mutex. The two conditions are *conditionArrayFull* and *conditionArrayEmpty* which are responsible for telling other threads to wait whenever the customer queue is full or empty. The *conditionArrayFull* condition will be called by the customer thread whereas the *conditionArrayEmpty* will be called by the teller thread. Furthermore, the two mutexes are file mutex and customer mutex. The file mutex is responsible whenever the threads need to write into the *r_log file* whereas the customer mutex is used whenever the thread needs to access the customer queue.

The customer thread will only run the *customer* function. This function is only responsible for inserting the customer into the predetermined-sized queue from the file in a set interval of time. The main thread will pass a struct containing a pointer to the queue, the size of the queue, and the queue interval. There is no need to mutex lock the input file as the only this thread is the only thread that is using it. However, when the thread inserts the customer into the queue, the customer mutex lock is called and once it is finished, the thread called the customer mutex unlock function to allow other threads to access the queue.

Furthermore, when the customer queue is full i.e., when the current count is equal to the size of the queue, the thread will pause infinitely by calling the wait function. This thread will wait until one of the teller threads calls the signal function. Similarly, when the queue size is zero, the teller threads will wait infinitely by calling the wait function using different conditions and will proceed to process the customers who are in the queue when the customer thread calls the signal function with the same condition.

The teller threads contain many mutex lock and unlock compared to the customer thread. When the queue size is greater than zero, the teller thread will try to acquire the customer mutex lock to dequeue the customer that is positioned on the "head" of the queue. Once it is dequeued, the mutex lock is then released. Then the thread will call the file mutex lock to write the response time to the *r_log* file and release it afterward. Similarly, once the serving simulation has been completed, the teller threads will once again call the file mutex lock to write it into the *r_log* file.

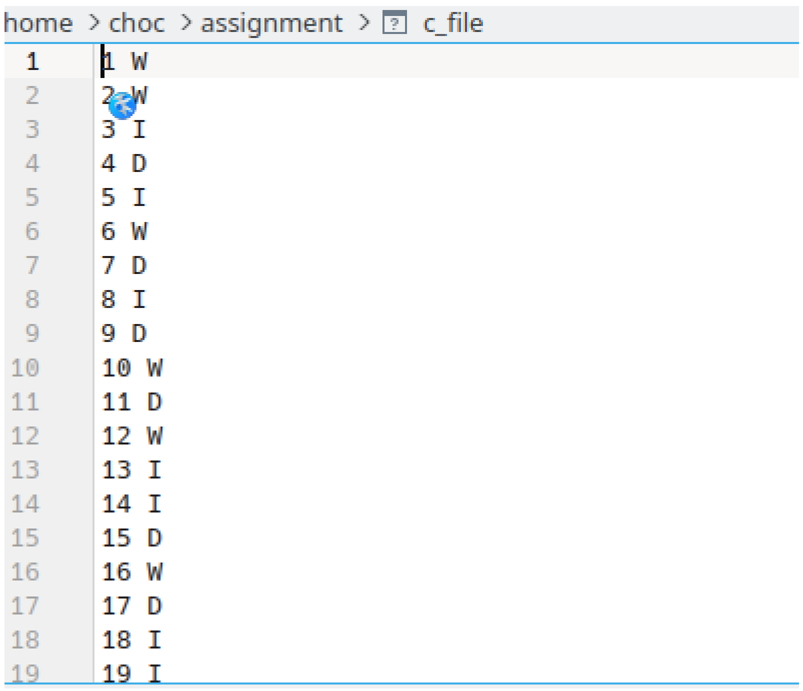
All the teller threads will have their own struct that is used to store parameters such as the time the tellers initiated, the number of customers served, and their termination time. The teller struct is passed using a pointer from the main thread to the teller threads. As a result of this, there is no need to use mutex lock and unlock whenever the teller threads access the struct as each teller thread has its own copy of the teller struct.

In this implementation, the time is stored differently. Firstly, I call the *time()* function and call *localtime_r*() function afterward. According to the manual guide, the *localtime_r()* function is thread safe compared to its counterpart *localtime()*. Instead of saving the whole time struct parameter to the customer node and teller struct, I only save the three most important parameters which are the hour, the minute, and the seconds. I have tried saving the whole time struct, but this results in changing the content of the time struct i.e., the termination and initiation time of teller threads will be equal.

Inputs and Outputs

This implementation is written, compiled and tested using the Kubuntu operating system which is simulated using the Oracle VM Virtual Box. The gcc version used in this program is 12.2.0

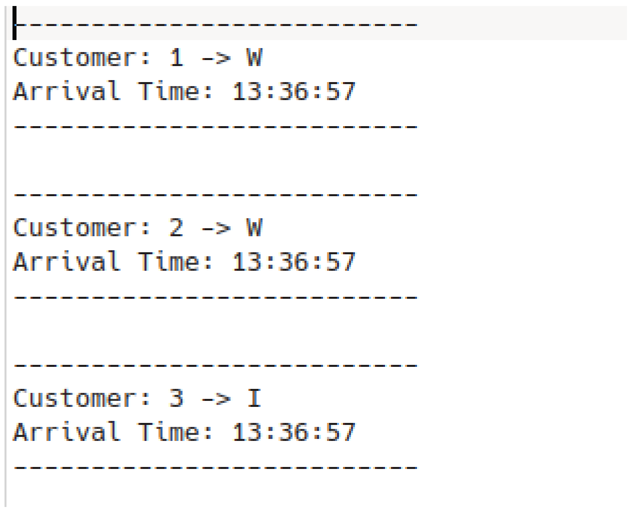
The valid sample input is as follows:



home > choc > assignment > c_file

```
1 W
2 W
3 I
4 D
5 I
6 W
7 D
8 I
9 D
10 W
11 D
12 W
13 I
14 I
15 D
16 W
17 D
18 I
19 I
```

When the program has successfully inserted the customer into the queue, the following text will be written in the *r_log* file.

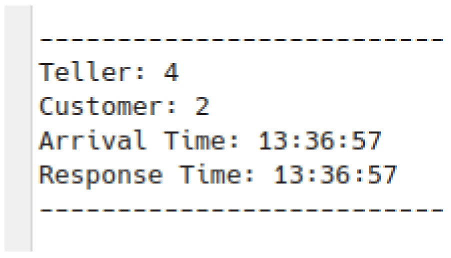


```
--------------------------
Customer: 1 -> W
Arrival Time: 13:36:57
--------------------------

--------------------------
Customer: 2 -> W
Arrival Time: 13:36:57
--------------------------

--------------------------
Customer: 3 -> I
Arrival Time: 13:36:57
--------------------------
```

When the teller has successfully dequeue a customer into the queue, the following text will be written in the *r_log* file:



```
--------------------------
Teller: 4
Customer: 2
Arrival Time: 13:36:57
Response Time: 13:36:57
--------------------------
```

When the teller has successfully served the customer, the following text will be written in the *r_log* file:

```
--------------------------
Teller: 1
Customer: 25
Arrival Time: 13:37:07
Response Time: 13:37:17
Completion Time: 13:37:20
--------------------------
```

When there are no more customers, the teller threads will write the following text in the *r_log* file.

```
--------------------------
Termination: teller-1
#served customers: 10
Start time: 13:36:57
Termination time: 13:37:25
--------------------------
--------------------------
Termination: teller-2
#served customers: 7
Start time: 13:36:57
Termination time: 13:37:25
--------------------------
--------------------------
Termination: teller-3
#served customers: 7
Start time: 13:36:57
Termination time: 13:37:25
--------------------------
```