# Operating Systems Report

A picture containing text, electronics, screenshot, font

Description automatically generated

Contents

[Operating Systems Report 1](#_Toc134452160)

[Design Implementation: 3](#_Toc134452161)

[customerQueue 3](#_Toc134452162)

[Services.c 3](#_Toc134452163)

[FileIO.c 3](#_Toc134452164)

[Synchronisation: 3](#_Toc134452165)

[Cases: 3](#_Toc134452166)

[SAMPLE INPUTS: 4](#_Toc134452167)

[SOURCE CODE: 4](#_Toc134452168)

[customerQueue.c 4](#_Toc134452169)

[customerQueue.h 8](#_Toc134452170)

[customerStruct.h 9](#_Toc134452171)

[fileIO.c 9](#_Toc134452172)

[fileIO.h 15](#_Toc134452173)

[main.c 16](#_Toc134452174)

[services.c 20](#_Toc134452175)

[services.h 23](#_Toc134452176)

[tellerStruct.h 23](#_Toc134452177)

[timers.c 24](#_Toc134452178)

[timers.h 24](#_Toc134452179)

# Design Implementation:

## customerQueue

Queue()-> Queue customers

Dequeue()-> Remove customer from line

isEmpty()-> Checks if queue is empty

freeQueue()-> Frees memory allocation for queue

peekFirst()-> get first customer in queue

peekLast()-> get last customer in queue

getTotal()-> returns total customers that entered the queue

getCurrentSize()-> returns current size of the queue

## Services.c

Teller(): Creates 4 teller threads and removes 1 customer from the queue to serve. After which will log all the information in 3 different logs

Customer(): Reads from file and creates customer objects and stores in a customer queue.

## FileIO.c

Open()-> reads the given file

Teller and Customer logs-> These will log all the information to the r\_log.txt file by appending to the end

# Synchronisation:

Synchronisation is achieved through the use of mutual exclusion functions. In particular the use of pthread\_mutex\_lock and mutex\_unlock are used to lock shared variables and resources. In particular they are used when queueing, dequeuing, opening and writing to files. As 5 threads are accessing the queue, the threads must wait for the mutex to unlock so that there is no race condition. The 4 teller threads and customer thread also access the r\_log file, and therefore cannot all access it at the same time, and therefore needs a mutex lock and unlock.

# Cases:

The program will wait for customer thread to finish queueing customers, due to using pthread\_join, and teller threads when implemented the pthread\_join also start 1 after the other. Therefore thread 1 will always start first and then 2, 3, 4 and wait for eachother to finish before grabbing another customer from the queue. However when pthread\_join is not used, the threads will start instantly and grab customers from the queue whenever available. When customer thread does not use pthread\_join, teller threads can grab customers when the customer thread is still queueing customers, and therefore does not need to wait for all customers to be queued. However with pthread\_join, tellers will wait for customer thread to finish queueing customers.

Beside this flaw, the functionality of the program does what is needed, and works correctly.

## SAMPLE INPUTS:

|  |  |  |
| --- | --- | --- |
| TEST | INPUT | OUTPUT |
| Invalid Input | ./cq 5 1 1 1 | ERROR MESSAGE -> Check pic 1 |
| Valid input | ./cq 5 1 1 1 1 | Program continues |

Picture 1 enters an invalid amount of arguments. The output will say an error and what the format of the arguments should be.

PIC 1:

A screenshot of a computer

Description automatically generated with medium confidence

Picture 2 enters the current amount of arguments. The program will run as usual and exit when finished.

PIC 2:



## SOURCE CODE:

# customerQueue.c

#include "customerQueue.h"

/\*

Jaron Rose 18823978

Queue c file to queue and dequeue customers in a linked list

\*/

/\*Create queue allocation\*/

*c\_queue*\* head = NULL;

*c\_queue*\* tail = NULL;

int total;

int currentSize;

/\*

Queue(ADD) customers into linked list

\*/

*Customer*\* enqueue(*Customer*\* *customerInfo*)

{

    int val;

*c\_queue*\* newCustomer = (*c\_queue*\*)malloc(sizeof(*c\_queue*));

    /\*Assign data to new node\*/

    newCustomer->customer = *customerInfo*;

    val = isEmpty();

    if(val == 1)

    {

        head = newCustomer;

        head->previous = NULL;

        tail = newCustomer;

        tail->next = NULL;

    }

    else

    {

        /\*If linked list is not empty, go to end\*/

        while(tail->next != NULL)

        {

            /\*last node is temp\*/

            tail = tail->next;

        }

        tail->next = newCustomer;

        newCustomer->previous = tail;

        tail = newCustomer;

        tail->next = NULL;

    }

    total++;

    currentSize++;

    return tail->customer;

}

/\*

Dequeue(REMOVE) customers from linked list

\*/

*Customer*\* dequeue()

{

*Customer*\* nodeValue = NULL;

    int val;

    val = isEmpty();

    if(val == 1)

    {

        printf("Queue is empty");

    }

    else

    {

        nodeValue = head->customer;

        head = head->next;

    }

    currentSize--;

    return nodeValue;

}

/\*

Check if the queue is empty

\*/

int isEmpty()

{

    int empty = FALSE;

    if(head == NULL)

    {

        empty = TRUE;

    }

    return empty;

}

/\*

Retrieve first in queue customer

\*/

*Customer*\* peekFirst()

{

*Customer*\* nodeValue = NULL;

    int val;

    val = isEmpty();

    if(val == 1)

    {

        printf("First node is empty");

    }

    else

    {

        nodeValue = head->customer;

    }

    return nodeValue;

}

/\*

Retrieve customer that is last in queue

\*/

*Customer*\* peekLast()

{

*Customer*\* nodeValue = NULL;

    int val;

    val = isEmpty();

    if(val == 1)

    {

        printf("First node is empty");

    }

    else

    {

        nodeValue = tail->customer;

    }

    return nodeValue;

}

/\*

Free the queue array

\*/

void freeQueue()

{

*c\_queue*\* temp;

    while(head != NULL)

    {

        temp = head;

        head = head->next;

        free(temp);

    }

}

/\*

Display all customers in the queue

\*/

void displayList()

{

    while (head->next != NULL)

    {

        printf("Customer ID: %d\nCustomer Service Type: %c \n", head->customer->customerID, head->customer->serviceType);

        head = head->next;

    }

    if (head == NULL)

    {

        printf("list is empty\n");

    }

}

/\*

Retrieve total customers that have been in queue

\*/

int getTotal()

{

    return total;

}

/\*

Retrieve current number of customers in queue

\*/

int getCurrentSize()

{

    return currentSize;

}

# customerQueue.h

#ifndef CUSTOMERQUEUE\_H

#define CUSTOMERQUEUE\_H

#include "customerQueue.h"

#include <stdio.h>

#include <stdlib.h>

#include "customerStruct.h"

/\*

Jaron Rose 18823978

Customer Queue to add customers or remove customers from the queue

\*/

#include "customerStruct.h"

#include "pthread.h"

*pthread\_mutex\_t* queueMutex;

*pthread\_mutex\_t* customerMutex;

*pthread\_mutex\_t* logMutex;

*pthread\_mutex\_t* sleepMutex;

*pthread\_cond\_t* tellerCondition;

*pthread\_cond\_t* tellerFinishCondition;

typedef struct *c\_queue*

{

*Customer*\* customer;

    struct *c\_queue* \*next;

    struct *c\_queue* \*previous;

}*c\_queue*;

*Customer*\* enqueue(*Customer*\* *customerInfo*);

*Customer*\* dequeue();

int isEmpty();

*Customer*\* peekFirst();

*Customer*\* peekLast();

void displayList();

void freeQueue();

int getTotal();

int getCurrentSize();

#define TRUE 1

#define FALSE !TRUE

#endif

# customerStruct.h

#ifndef CUSTOMERSTRUCT\_H

#define CUSTOMERSTRUCT\_H

/\*

Jaron Rose 18823978

Customer Struct to store customer information

\*/

typedef struct *Customer*

{

    int customerID;

    char serviceType;

    int tellerID;

    struct *tm*\* arrivalTime;

    struct *tm*\* responseTime;

    struct *tm*\* completionTime;

}*Customer*;

#endif

# fileIO.c

#include "fileIO.h"

/\*

\* Jaron Rose 18823978

\* File input, reads given file and writes (APPENDS) to given file

\*/

int count = 0;

void\* open(void\* *fileName*)

{

    /\*Variable declarations\*/

*FILE*\* file;

*Customer*\* customer;

    int done;

    int currentLine;

    int keepReading;

    char\* line;

    char\* temp;

    /\*Current line in the file\*/

    currentLine = 0;

    done = FALSE;

    keepReading = TRUE;

    /\*Memory allocation\*/

    line = (char\*)malloc(sizeof(char\*));

    /\*Open the file\*/

    file = fopen((char\*)*fileName*, "r");

    customer = (*Customer*\*)malloc(sizeof(*Customer*));

    do

    {

        /\*File is empty or doesnt exist\*/

        if(file == NULL)

        {

            perror("Error opening file: ");

            done = TRUE;

        }

        else

        {

            if(ferror(file))

            {

                perror("Error reading file: ");

                done = TRUE;

            }

            else

            {

                currentLine = 0;

                do

                {

                    /\*Get first line\*/

                    fgets(line, sizeof(line), file);

                    if(feof(file))

                    {

                        keepReading = FALSE;

                    }

                    /\*Continue last place in the file\*/

                    else if(currentLine == count)

                    {

                        /\*Tokenise to assign values to customer\*/

                        temp = strtok(line, " ");

                        customer->customerID = atoi(temp);

                        temp = strtok(NULL, " ");

                        customer->serviceType = \*(char\*)temp;

                        customer->arrivalTime = systemTime();

                        keepReading = FALSE;

                    }

                    currentLine++;

                }

                while (keepReading);

                count++;

                done = TRUE;

            }

        }

    }

    while (!done);

    free(line);

    if(file != NULL)

    {

        /\*Close file\*/

        fclose(file);

    }

    return (void\*)customer;

}

/\*Log customer details\*/

void customerLog(*Customer*\* *customer*)

{

*FILE*\* file;

    char\* fileName;

    fileName = "r\_log.txt";

    /\*Append to file\*/

    file = fopen(fileName, "a");

    if(file == NULL)

    {

        perror("Error opening file: ");

        free(*customer*);

    }

    else

    {

        if(ferror(file))

        {

            perror("Error reading file: ");

            free(*customer*);

        }

        else

        {

            fprintf(file, "-------------------------------------------------\n");

            fprintf(file, "Customer %d: %c\n", *customer*->customerID, *customer*->serviceType);

            fprintf(file, "Arrival Time: %s", asctime(*customer*->arrivalTime));

            fprintf(file, "-------------------------------------------------\n");

        }

    }

    /\*Close file\*/

    fclose(file);

}

/\*Teller response logging\*/

void tellerResponseLog(*Customer*\* *customer*)

{

*FILE*\* file;

    char\* fileName;

    fileName = "r\_log.txt";

    /\*Append to file\*/

    file = fopen(fileName, "a");

    if(file == NULL)

    {

        perror("Error opening file: ");

        free(*customer*);

    }

    else

    {

        if(ferror(file))

        {

            perror("Error reading file: ");

            free(*customer*);

        }

        else

        {

            fprintf(file, "-------------------------------------------------\n");

            fprintf(file, "Teller: %d\n", *customer*->tellerID);

            fprintf(file, "Customer: %d\n", *customer*->customerID);

            fprintf(file, "Arrival Time: %s", asctime(*customer*->arrivalTime));

            fprintf(file, "Response Time: %s", asctime(*customer*->responseTime));

            fprintf(file, "-------------------------------------------------\n");

        }

    }

    /\*Close file\*/

    fclose(file);

}

/\*Teller will log when finished serving customer\*/

void tellerCompletionLog(*Customer*\* *customer*)

{

*FILE*\* file;

    char\* fileName;

    fileName = "r\_log.txt";

    /\*Append to file\*/

    file = fopen(fileName, "a");

    if(file == NULL)

    {

        perror("Error opening file: ");

        free(*customer*);

    }

    else

    {

        if(ferror(file))

        {

            perror("Error reading file: ");

            free(*customer*);

        }

        else

        {

            fprintf(file, "-------------------------------------------------\n");

            fprintf(file, "Teller: %d\n", *customer*->tellerID);

            fprintf(file, "Customer: %d\n", *customer*->customerID);

            fprintf(file, "Arrival Time: %s", asctime(*customer*->arrivalTime));

            fprintf(file, "Completion Time: %s", asctime(*customer*->completionTime));

            fprintf(file, "-------------------------------------------------\n");

        }

    }

    /\*Close file\*/

    fclose(file);

}

/\*When teller terminates, teller will log information\*/

void tellerTerminationLog(*Customer*\* *customer*, *Teller*\* *teller*)

{

*FILE*\* file;

    char\* fileName;

    fileName = "r\_log.txt";

    /\*Append to file\*/

    file = fopen(fileName, "a");

    if(file == NULL)

    {

        perror("Error opening file: ");

        free(*customer*);

    }

    else

    {

        if(ferror(file))

        {

            perror("Error reading file: ");

            free(*customer*);

        }

        else

        {

            fprintf(file, "-------------------------------------------------\n");

            fprintf(file, "Termination: %d\n", *customer*->tellerID);

            fprintf(file, "#Served Customers: %d\n", *teller*->servedCustomers);

            fprintf(file, "Start Time: %s", asctime(*teller*->startTime));

            fprintf(file, "Termination Time: %s", asctime(*teller*->terminationTime));

            fprintf(file, "-------------------------------------------------\n");

        }

    }

    /\*Close file\*/

    fclose(file);

}

/\*When no more customers are in queue, last teller will log all teller statistics\*/

void\* tellerStatisticLog(void\* *teller*)

{

*FILE*\* file;

    char\* fileName;

    int total;

*Teller*\* tellers;

    tellers = (*Teller*\*)*teller*;

    fileName = "r\_log.txt";

    /\*Append to file\*/

    file = fopen(fileName, "a");

    if(file == NULL)

    {

        perror("Error opening file: ");

    }

    else

    {

        if(ferror(file))

        {

            perror("Error reading file: ");

        }

        else

        {

            total = getTotal();

            fprintf(file, "-------------------------------------------------\n");

            fprintf(file, "Teller Statistics\n");

            fprintf(file, "Teller 1 served: %d\n", tellers[0].servedCustomers);

            fprintf(file, "Teller 2 served: %d\n", tellers[1].servedCustomers);

            fprintf(file, "Teller 3 served: %d\n", tellers[2].servedCustomers);

            fprintf(file, "Teller 4 served: %d\n", tellers[3].servedCustomers);

            fprintf(file, "Total number of customers: %d\n", total);

            fprintf(file, "-------------------------------------------------\n");

        }

    }

    /\*Close file\*/

    fclose(file);

    return 0;

}

# fileIO.h

#ifndef FILEIO\_H

#define FILEIO\_H

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

#include "fileIO.h"

#include "customerStruct.h"

#include "customerQueue.h"

#include "timers.h"

#include "tellerStruct.h"

/\*

Jaron Rose 18823978

FILE IO to read and write to files

\*/

#include "customerQueue.h"

#include "tellerStruct.h"

void\* open(void\* *fileName*);

void customerLog(*Customer*\* *customer*);

void tellerResponseLog(*Customer*\* *customer*);

void tellerCompletionLog(*Customer*\* *customer*);

void tellerTerminationLog(*Customer*\* *customer*, *Teller*\* *teller*);

void\* tellerStatisticLog(void\* *teller*);

#define TRUE 1

#define FALSE !TRUE

#endif

# main.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include "fileIO.h"

#include "customerQueue.h"

#include "customerStruct.h"

#include "tellerStruct.h"

#include "services.h"

#include "timers.h"

#include <unistd.h>

/\*

\* Jaron Rose 18823978

\* Main function

\*/

/\*Number of teller threads can be changed to increase number of threads\*/

#define TELLER\_THREADS 4

/\*Mutual Exclusion declarations\*/

*pthread\_mutex\_t* queueMutex;

*pthread\_mutex\_t* customerMutex;

*pthread\_mutex\_t* logMutex;

/\*Forward declarations\*/

void customerThread(char\* *fileName*);

void tellerThread();

void initialiseMutualExclusion();

void destroyMutualExclusion();

int main(int *argc*, char\* *argv*[])

{

    /\*

    Variable initialisation

    \*/

    int queueSize;

    int customerArrival;

    int withdrawal;

    int deposit;

    int information;

    char\* fileName;

    fileName = "c\_file.txt";

    /\*

    If not enough commands given in the command line

    \*/

    if(*argc* <= 5 || *argc* > 6)

    {

        printf("ERROR: invalid number of args entered.\n");

        printf("FORMAT: ./cq m(size of queue) t(time for customer arrival) t1 t2 t3(time for W D I service type).\n");

    }

    /\*Commands must not be negative and queue size must be greater than 0 but smaller than 100\*/

    else if(atoi(*argv*[1]) < 0 || atoi(*argv*[1]) > 100 || atoi(*argv*[2]) < 0 || atoi(*argv*[3]) < 0 || atoi(*argv*[4]) < 0 || atoi(*argv*[5]) < 0)

    {

        printf("ERROR: POSITIVE NUMBERS ONLY.\n");

        printf("FORMAT: ./cq m(0 < size of queue < 100) t(time for customer arrival) t1 t2 t3(time for W D I service type).\n");

    }

    else

    {

        /\*Initialise mutual exclusions\*/

        initialiseMutualExclusion();

        /\*Take arguments and give to variables\*/

        queueSize = atoi(*argv*[1]);

        customerArrival = atoi(*argv*[2]);

        withdrawal = atoi(*argv*[3]);

        deposit = atoi(*argv*[4]);

        information = atoi(*argv*[5]);

        /\*Set sleep timers and queue size\*/

        services(queueSize, customerArrival, withdrawal, deposit, information);

        sleep(customerArrival);

        /\*Start and create threads\*/

        customerThread(fileName);

        tellerThread();

        sleep(20);

        /\*Free memory allocations\*/

        destroyMutualExclusion();

        freeQueue();

        pthread\_exit(0);

    }

    return 0;

}

/\*This method will create a customer thread to run customer()\*/

void customerThread(char\* *fileName*)

{

*pthread\_t* id;

    pthread\_create(&id, NULL, &customer, *fileName*);

    pthread\_join(id, NULL);

}

/\*This method will create 4 teller threads and 4 teller objects to run teller()\*/

void tellerThread()

{

*pthread\_t*\* id;

*Teller*\* tellers;

    int val;

    int i;

    /\*Memory allocations\*/

    id = (*pthread\_t*\*)malloc(sizeof(*pthread\_t*) \* TELLER\_THREADS);

    tellers = (*Teller*\*)malloc(sizeof(*Teller*) \* TELLER\_THREADS);

    for(i = 0; i < TELLER\_THREADS; i++)

    {

        /\*Default values\*/

        tellers[i].tellerID = i + 1;

        tellers[i].servedCustomers = 0;

        tellers[i].startTime = systemTime();

    }

    val = isEmpty();

    while(val != 1)

    {

        /\*If queue is empty while running, pause and wait for customers\*/

        val = isEmpty();

        if(val == 1)

        {

            sleep(2);

        }

        /\*4 Teller threads\*/

        pthread\_create(&id[0], NULL, &teller, &tellers[0]);

        pthread\_join(id[0], NULL);

        pthread\_create(&id[1], NULL, &teller, &tellers[1]);

        pthread\_join(id[1], NULL);

        pthread\_create(&id[2], NULL, &teller, &tellers[2]);

        pthread\_join(id[2], NULL);

        pthread\_create(&id[3], NULL, &teller, &tellers[3]);

        pthread\_join(id[3], NULL);

    }

    pthread\_create(&id[3], NULL, &tellerStatisticLog, tellers);

    pthread\_join(id[0], NULL);

    pthread\_join(id[1], NULL);

    pthread\_join(id[2], NULL);

    pthread\_join(id[3], NULL);

    /\*Free memory allocations\*/

    free(tellers);

    free(id);

    pthread\_exit(0);

}

/\*Initialise mutexes and conditions\*/

void initialiseMutualExclusion()

{

    pthread\_mutex\_init(&queueMutex, NULL);

    pthread\_mutex\_init(&logMutex, NULL);

    pthread\_mutex\_init(&sleepMutex, NULL);

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

}

/\*Destroy mutexes and conditions\*/

void destroyMutualExclusion()

{

    pthread\_mutex\_destroy(&queueMutex);

    pthread\_mutex\_destroy(&logMutex);

    pthread\_mutex\_destroy(&sleepMutex);

    pthread\_cond\_destroy(&tellerCondition);

}

# services.c

#include "services.h"

/\*

\* Jaron Rose 18823978

\* Functions for teller and customer to queue and dequeue

\*/

int queueSize;

int customerArrival;

int withdrawalTime;

int depositTime;

int informationTime;

/\*Initialise variables for sleep times\*/

void services(int *queueNum*, int *customerTimer*, int *w*, int *d*, int *i*)

{

    queueSize = *queueNum*;

    customerArrival = *customerTimer*;

    withdrawalTime = *w*;

    depositTime = *d*;

    informationTime = *i*;

}

/\*Enqueue customers to the queue\*/

void\* customer(void\* *fileName*)

{

*Customer*\* customer;

    int i;

    for(i = 0; i < queueSize; i++)

    {

        /\*Retrieve customer from file\*/

        customer = open(*fileName*);

        /\*Queue customer\*/

        pthread\_mutex\_lock(&customerMutex);

        enqueue(customer);

        pthread\_mutex\_unlock(&customerMutex);

        /\*Log customer information\*/

        pthread\_mutex\_lock(&logMutex);

        customerLog(customer);

        pthread\_mutex\_unlock(&logMutex);

        /\*Wait for next customer to arrive\*/

        sleep(customerArrival);

    }

    return 0;

}

/\*Determines which service type to how long teller will take to serve customer\*/

void getSleep(*Customer*\* *customer*)

{

    if(*customer*->serviceType == 'w')

    {

        sleep(withdrawalTime);

    }

    else if(*customer*->serviceType == 'd')

    {

        sleep(depositTime);

    }

    else if(*customer*->serviceType == 'i')

    {

        sleep(informationTime);

    }

    else

    {

        /\*Default\*/

        sleep(2);

    }

}

/\*Teller will retrieve customer from queue, and serve customer based on

sleep time, and then log all information\*/

void\* teller(void\* *teller*)

{

*Customer*\* customer;

*Teller*\* custTeller;

    int val;

    /\*Cast void\* to teller\*/

    custTeller = (*Teller*\*) *teller*;

    val = isEmpty();

    if(val != 1)

    {

        /\*Grab customer from the queue\*/

        pthread\_mutex\_lock(&queueMutex);

        customer = dequeue();

        pthread\_mutex\_unlock(&queueMutex);

        /\*Teller is serving the customer\*/

        getSleep(customer);

        /\*Get system time\*/

        customer->responseTime = systemTime();

        customer->tellerID = custTeller->tellerID;

        /\*Log information\*/

        pthread\_mutex\_lock(&logMutex);

        tellerResponseLog(customer);

        pthread\_mutex\_unlock(&logMutex);

        getSleep(customer);

        /\*Get system time\*/

        customer->completionTime = systemTime();

        /\*Log information\*/

        pthread\_mutex\_lock(&logMutex);

        tellerCompletionLog(customer);

        pthread\_mutex\_unlock(&logMutex);

        /\*Keep track of how many customers teller has served\*/

        custTeller->servedCustomers += 1;

        getSleep(customer);

        /\*Get system time\*/

        custTeller->terminationTime = systemTime();

        /\*Log information\*/

        pthread\_mutex\_lock(&logMutex);

        tellerTerminationLog(customer, custTeller);

        pthread\_mutex\_unlock(&logMutex);

    }

    return 0;

}

# services.h

#ifndef SERVICES\_H

#define SERVICES\_H

/\*

Jaron Rose 18823978

Teller and customer methods, teller will grab customers from queue,

customers will join the queue

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#include <string.h>

#include "fileIO.h"

#include "customerQueue.h"

#include "timers.h"

#include "tellerStruct.h"

#include<unistd.h>

#include "customerStruct.h"

void services(int *queueNum*, int *customerTimer*, int *w*, int *d*, int *i*);

void getSleep(*Customer*\* *customer*);

void\* customer(void\* *fileName*);

void\* teller(void\* *tellerNum*);

#define TRUE 1

#define FALSE !TRUE

#endif

# tellerStruct.h

#ifndef TELLERSTRUCT\_H

#define TELLERSTRUCT\_H

/\*

Jaron Rose 18823978

Struct to store teller information

\*/

typedef struct *Teller*

{

    int tellerID;

    int servedCustomers;

    struct *tm*\* startTime;

    struct *tm*\* terminationTime;

}*Teller*;

#endif

# timers.c

#include "timers.h"

/\*

Jaron Rose 18823978

Method to retrieve system local time and date

\*/

struct *tm*\* systemTime()

{

*time\_t* rawTime;

    struct *tm*\* timeInfo;

    time(&rawTime);

    timeInfo = localtime(&rawTime);

    return timeInfo;

}

# timers.h

#ifndef TIMERS\_H

#define TIMERS\_H

/\*

Jaron Rose 18823978

Get system local time

\*/

#include <time.h>

#include <stdio.h>

struct *tm*\* systemTime();

#define TRUE 1

#define FALSE !TRUE

#endif