```
// GROUP D
// By Corey Green, Robby Hallock, and Kyle McCullough
// decoreyon.green@okstate.edu
// robert.hallock@okstate.edu
// kymccul@okstate.edu
// CS 4323
// finalGroupProject
// 4-26-22

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>
#include <sys/types.h>
#include <time.h>
//#include "Robby.h"
//#include "Corey.h"
//#include "Kyle.h"

#define TOTALMUTEX 12
#define TOTALCOUNT 2

/*
INPUTS:
# of medical pros.
# of total patients
# patient capaity
# sofa space
```

```
# max enter time interval
# patient checkup time
*/

struct threadStruct
{
   char *occupation;
   int id;
   int threadID;
   int bondId;
   clock_t waitTime;
};

struct summary
{
   int successfulCheckups;
   long medicalProAvgWaitTime;
   int patientsThatLeft;
   long patientsAvgWaitTime;
};
int successfulCheckups;
int avgStaffWaitTime;
int patientsLeft;
int avgPatientsWaitTime;
pthread_mutex_t mutex[TOTALMUTEX];
sem_t count[TOTALCOUNT];
int totalRoomCapacity;
int totalSofaCapacity;
int maxSofaCapacity;
```

```
int checkupTime;
int left;
int buffer;
int maxPatients;
struct summary summary;



void patientThreadFunc(struct threadStruct*);
void staffThreadFunc(struct threadStruct*);
void enterWaitingRoom();
void sitOnSofa();

void getMedicalCheckup();
void makePayment();
void leaveClinic();

void waitForPatients();
void performMedicalCheckup();
void acceptPayment();

//void patientThreadFunc(struct threadStruct*);--
//void staffThreadFunc(struct threadStruct*);
//void enterWaitingRoom();
//void sitOnSofa();
//void getMedicalCheckup();
//void makePayment();
//void leaveClinic();
//void waitForPatients();
//void performMedicalCheckup();
```

```
//void acceptPayment();



//------------------------------------------------------------------------


typedef struct Task {
   //void (*taskFunction)(struct threadStruct*);
   int selector;
   struct threadStruct* args;
   //int stop;
} Task;

Task queue[256];
int remainingTasks = 0;

//pthread_mutex_t mutexQueue;

void queueTask(Task task){
   pthread_mutex_lock(&mutex[11]);
   queue[remainingTasks++] = task;
   pthread_mutex_unlock(&mutex[11]);
}

Task dequeue(){
   Task task = queue[0];
   for (int i = 0; i < remainingTasks-1; i++){
      queue[i] = queue[i+1];
```

```
    }
    remainingTasks--;
    return task;
}

void* mainThreadLoop(void* args){
    Task task;
    int flag = 1;
    while(flag){
        pthread_mutex_lock(&mutex[11]);
        if (remainingTasks < 1){
            pthread_mutex_unlock(&mutex[11]);
            continue;
        }
        task = dequeue();

        pthread_mutex_unlock(&mutex[11]);
        switch(task.selector){
            case 0:
                staffThreadFunc(task.args);
                break;
            case 1:
                patientThreadFunc(task.args);
                break;
            case 2:
                flag = 0;
                break;
        }
    }
```

```
}

//------------------------------------------------------------------------

// allow command line args
int main(int argc, char *argv[])
{
   left = 0;
   buffer = -1;
   //pthread_mutex_init(&mutexQueue, NULL);
   srand(time(NULL));
   int medicalStaff, totalPatients, roomCapacity, sofaSpace, maxTimeInterval;
   // assigns the arguments to ints
   medicalStaff = atoi(argv[1]);
   totalPatients = atoi(argv[2]);
   roomCapacity = atoi(argv[3]);
   sofaSpace = atoi(argv[4]);
   maxTimeInterval = atoi(argv[5]);
   checkupTime = atoi(argv[6]);

   totalRoomCapacity = roomCapacity;
   totalSofaCapacity = sofaSpace;
   maxSofaCapacity = sofaSpace;
   maxPatients = totalPatients;
```

```
   // Initialze Semaphores
   for (int i = 0; i < TOTALMUTEX; i++)
      pthread_mutex_init(&mutex[i], NULL);
   pthread_mutex_lock(&mutex[2]);
   pthread_mutex_lock(&mutex[4]);
   pthread_mutex_lock(&mutex[5]);
   pthread_mutex_lock(&mutex[6]);

   sem_init(&count[0], 0, 0);
   sem_init(&count[1], 0, 0);



   pthread_t threads[medicalStaff+totalPatients];

   struct threadStruct contents[totalPatients];
   struct threadStruct contentsM[medicalStaff];



   // printf("medicalstaff = %d, totalPatients = %d, roomCapacity = %d, sofaSpace = %d, maxTimeInterval
= %d, checkupTime = %d", medicalStaff, totalPatients, roomCapacity, sofaSpace, maxTimeInterval,
checkupTime);
   for (int i = 0; i < medicalStaff+totalPatients; i++){
      if (pthread_create(&threads[i], NULL, &mainThreadLoop, NULL)){
         printf("Failed to create thread\n");
      }
   }

   for (int i = 0; i < medicalStaff; i++){
      contentsM[i].id = i;
```

```
    contentsM[i].occupation = "Staff";
    Task t = {
        .selector = 0,
        .args = &contentsM[i],
    };
    queueTask(t);
}

for (int i = 0; i < totalPatients; i++){
    int ms = (rand() % maxTimeInterval) + 1;
    contents[i].id = i;
    contents[i].occupation = "Patient";
    Task t = {
        .selector = 1,
        .args = &contents[i],
    };
    queueTask(t);
    usleep(ms * 1000);
}

while(1){
    pthread_mutex_lock(&mutex[9]);
    if (left >= totalPatients){
        pthread_mutex_unlock(&mutex[9]);
        break;
    }
    pthread_mutex_unlock(&mutex[9]);
}
```

```
    for (int i = 0; i < medicalStaff; i++){
        sem_post(&count[0]);
    }


    for (int i = 0; i < totalPatients+medicalStaff; i++){
        Task t = {
            .selector = 2
        };
        queueTask(t);
    }


    for (int i = 0; i < medicalStaff+totalPatients; i++){
        if (pthread_join(threads[i], NULL)){
            perror("Failed to join thread\n");
        }
    }


        summary.patientsAvgWaitTime = summary.patientsAvgWaitTime/summary.successfulCheckups;
        summary.medicalProAvgWaitTime = summary.medicalProAvgWaitTime/medicalStaff;
        printf("Statistical Summary:\n");
        printf("----------------------------------------------------------------------------\n");
        printf("Number of successful checkups: %d \n", summary.successfulCheckups);
        printf("Average waiting time for Medical Professionals: %ldms \n",
summary.medicalProAvgWaitTime);
        printf("Number of Patients that left: %d \n", summary.patientsThatLeft);
        printf("Average wait time for patients: %ldms \n", summary.patientsAvgWaitTime);
        //TODO: change and label time values for Average wait time
```

```
    for (int i = 0; i < TOTALMUTEX; i++){
        pthread_mutex_destroy(&mutex[i]);
    }
    for (int i = 0; i < TOTALCOUNT; i++){
        sem_destroy(&count[i]);
    }

    return 0;
}

void patientThreadFunc(struct threadStruct *contents)
{
    contents->threadID = (int)gettid();

    printf("Patient %d (Thread ID: %d) Arrived to clinic\n", contents->id, contents->threadID);


    pthread_mutex_lock(&mutex[0]);
    if (totalRoomCapacity < 1) {
        leaveClinic(contents, 0);
        pthread_mutex_unlock(&mutex[0]);
        return;
    }
    contents->waitTime = clock();
    enterWaitingRoom();
    pthread_mutex_unlock(&mutex[0]);
```

```
    pthread_mutex_lock(&mutex[1]);

    if (totalSofaCapacity <= 0) {
        printf("Patient %d (ThreadID: %d): Standing in the waiting room\n", contents->id, contents->threadID);
    }
    pthread_mutex_unlock(&mutex[1]);

    while(1){
        pthread_mutex_lock(&mutex[1]);
        if(totalSofaCapacity > 0) {
            sitOnSofa(contents);
            pthread_mutex_unlock(&mutex[1]);
            break;
        }
        pthread_mutex_unlock(&mutex[1]);
    }

    sem_post(&count[0]);
    sem_wait(&count[1]);
    getMedicalCheckup(contents);

    makePayment(contents);
    leaveClinic(contents, 1);
}

void staffThreadFunc(struct threadStruct *contents)
{
    contents->threadID = (int)gettid();
```

```
    while(1){
        contents->waitTime = clock(); //start----
        pthread_mutex_lock(&mutex[1]);
        if (totalSofaCapacity == maxSofaCapacity) waitForPatients(contents);
        pthread_mutex_unlock(&mutex[1]);
        sem_post(&count[1]);
        sem_wait(&count[0]);
        if (left >= maxPatients) return;
        contents->waitTime = clock() - contents->waitTime; //end----
        pthread_mutex_lock(&mutex[10]);
        summary.medicalProAvgWaitTime += contents->waitTime;
        pthread_mutex_unlock(&mutex[10]);
        performMedicalCheckup(contents);
        acceptPayment(contents);
    }
}



// MARK: funcs used by patients
void enterWaitingRoom()
{
    totalRoomCapacity--;

}

void sitOnSofa(struct threadStruct *contents)
{
    totalSofaCapacity--;
```

```
   printf("Patient %d (Thread ID: %d): Sitting on a sofa in the waiting room\n", contents->id, contents->threadID);
}
void getMedicalCheckup(struct threadStruct *contents)
{
   int staffId;

   pthread_mutex_lock(&mutex[1]);
   totalSofaCapacity++;
   pthread_mutex_unlock(&mutex[1]);
   pthread_mutex_lock(&mutex[0]);
   totalRoomCapacity++;
   printf("Patient %d (ThreadID: %d): Getting checkup\n", contents->id, contents->threadID);

   pthread_mutex_unlock(&mutex[0]);

   pthread_mutex_lock(&mutex[3]);

   buffer = contents->id;
   pthread_mutex_unlock(&mutex[4]);
   pthread_mutex_lock(&mutex[5]);

   contents->bondId = buffer;

   pthread_mutex_unlock(&mutex[3]);

   usleep(checkupTime * 1000);

}
```

```
void makePayment(struct threadStruct *contents)
{
   pthread_mutex_lock(&mutex[8]);
   printf("Patient %d (ThreadID: %d): Making payment to Medical Staff %d\n", contents->id, contents->threadID, contents->bondId);
   pthread_mutex_unlock(&mutex[2]);
   pthread_mutex_lock(&mutex[6]);
   pthread_mutex_unlock(&mutex[8]);
}
void leaveClinic(struct threadStruct *contents, int isSuccessful)
{
   pthread_mutex_lock(&mutex[9]);
   left++;
   pthread_mutex_unlock(&mutex[9]);
   if (isSuccessful){
      printf("Patient %d (ThreadID: %d): Leaving the clinic after receiving checkup\n", contents->id, contents->threadID);
      contents->waitTime = clock() - contents->waitTime;
      pthread_mutex_lock(&mutex[10]);
      summary.patientsAvgWaitTime += contents->waitTime;
      pthread_mutex_unlock(&mutex[10]);
      pthread_mutex_lock(&mutex[10]);
      summary.successfulCheckups++;
      pthread_mutex_unlock(&mutex[10]);
   } else {
      printf("Patient %d (Thread ID: %d): Leaving without checkup.\n", contents->id, contents->threadID);
      pthread_mutex_lock(&mutex[10]);
      summary.patientsThatLeft++;
      pthread_mutex_unlock(&mutex[10]);
```

```
    }
}

// MARK: funcs used by staff
void waitForPatients(struct threadStruct *contents)
{
    printf("Medical Professional %d (Thread ID: %d): Waiting for patient \n", contents->id, contents->threadID);
}
void performMedicalCheckup(struct threadStruct *contents)
{
    pthread_mutex_lock(&mutex[4]);

    contents->bondId = buffer;
    buffer = contents->id;

    pthread_mutex_unlock(&mutex[5]);

    printf("Medical Professional %d (Thread ID: %d): Checking patient %d\n", contents->id, contents->threadID, contents->bondId);

}
void acceptPayment(struct threadStruct *contents)
{
    pthread_mutex_lock(&mutex[7]);
    pthread_mutex_lock(&mutex[2]);
    printf("Medical Professional %d (Thread ID: %d): Accepted payment from Patient %d\n", contents->id, contents->threadID, contents->bondId);
    pthread_mutex_unlock(&mutex[6]);
    pthread_mutex_unlock(&mutex[7]);
```

```
}

/*
Reference:

Title: Thread Pools with function pointers in C
Author: codeVault
Source Code: https://code-vault.net/lesson/w1h356t5vg:1610029047572

*/
```