```c
//Andrew Ingle 04/08/2021 - Team I - Final Project
//Has to be built but not run.
//This is the primary server program, will be executed when server_driver
 calls execpl
//This program receives server name from server_driver parent process, assigns
 itself a port based on name
//manages thread pool and listens for next client, and assigns thread to run
 trainTicketMaster() for each customer
//when customer exists program via menu selection, thread returns to pool

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <pthread.h>

#include "max_trainSeating.h"
#include "andrew_trainTicketMaster.h"
#include "andrew_serverFuncs.h"

#include "caleb_server.h"

#include <semaphore.h>

#define THREAD_NUM 5

//multi thread/Thread pool code
//job that will be added to cue
typedef struct Job {
    int (*functionToExecute)(int, int, availableSeats*,int,sem_t*,sem_t*);
     //this will become int (*trainTicketMaster());
    int arg1, arg2, arg4; //will be for clients socket and server_name, shm
     ptr and shm fd
    availableSeats* arg3;
    sem_t *arg5;
    sem_t *arg6;
} Job;

//a queue of jobs that can be assigned to the next thread
Job jobQueue[200]; //don't really need 200 here
int jobCount = 0;
pthread_mutex_t mutexQueue;
pthread_cond_t condQueue;
```

```
//called to execute actual job in this case trainTicketMaster
void executeJob(Job* job) {
    job->functionToExecute(job->arg1, job->arg2,
     job->arg3,job->arg4,job->arg5,job->arg6);
}

//submitting job to be execetued with mutex to make sure more than one thread
 doesn't update
void submitJobForExecution(Job job) {
    pthread_mutex_lock(&mutexQueue);
    jobQueue[jobCount] = job;
    jobCount++;
    pthread_mutex_unlock(&mutexQueue);
    pthread_cond_signal(&condQueue);
}

//thread pool thread initiated
void* startThread(void* args) {

    while (1){
        Job job;
        //mutex locked so job count incremented correctly
        pthread_mutex_lock(&mutexQueue);
        //when no jobs thead will wait
        while (jobCount == 0) {
            pthread_cond_wait(&condQueue, &mutexQueue);
        }

        job = jobQueue[0];

        for (int i = 0; i < jobCount - 1; i++) {
            jobQueue[i] = jobQueue[i + 1];
        }
        jobCount--;
        pthread_mutex_unlock(&mutexQueue);

        executeJob(&job); //thread will Run TrainTickeMaster
    }

}


int main() {


    //semaphores to make sure not more than one thread tries to run
     trainTicketMaster at same instant
    sem_t mutexSem;

    sem_init(&mutexSem,0,1);
```

```
  int server_socket; //socket for tcp

  //Shared Memory Code (shared mem code by Max, integrated with Andrew)
  availableSeats *ptr; //pointer to shared memory object

  // size (in bytes) of shared memory object
  const int SIZE = sizeof(availableSeats) * 2; //Size of our struct * 2 so
   we can hold two days of entries
  // name of the shared memory object
  const char *name = "CS4323";
  // shared memory file descriptor
  int shm_fd;

  // Create or open a shared memory object
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
  // Set the shared memory object size
  ftruncate(shm_fd, SIZE);
  // Map the shared memory object into the current address space
  ptr = mmap(0, SIZE,  PROT_WRITE, MAP_SHARED, shm_fd, 0);


  availableSeats day1; //Struct for the first day
  availableSeats day2; //Struct for the second day


  day1.dateInt = 1; //Create the integer date for the first struct
  day2.dateInt = 2; //Create the integer date for the second struct

  day1.ticketNumber = 101; //Assign the ticketNumber for the first struct
  day2.ticketNumber = 201; //Assign the ticketNumber for the second struct

  const int sizeOfSeatsArray = sizeof(day1.seats) / sizeof(int);

  //Use for loop to initialize all array values to zero as every seat starts
   as being open
  for(int i = 0; i < sizeOfSeatsArray; i++) {
      day1.seats[i] = 0;
      day2.seats[i] = 0;
  }

//store each newProduct into shared memory object
  *(ptr) = day1;
  *(ptr + 1) = day2;

  // memory map the shared memory object
  ptr = (availableSeats *) mmap(0, SIZE, O_RDWR, MAP_SHARED, shm_fd, 0);
```

```
 //FIFO WITH SERVER_DRIVER

int fd = open("myfifo1", O_RDONLY); //fifo between server_driver and
 server_main

char client_message[50]; //buffer for socket receive for testing

int server_name;
//reading from pipe connected to parent process server_driver
read(fd,&server_name,sizeof(int)); //server now named either 1,2 or 3



sleep(2);
read(fd,&server_socket,sizeof(int)); //server now named either 1,2 or 3


printf("\nServer %d is alive and named!\n",server_name);
close(fd);

sem_t *reader_sem = sem_open(SEM_READER_NAME, O_RDWR);
sem_t *writer_sem = sem_open(SEM_WRITER_NAME, O_RDWR);
if (reader_sem == SEM_FAILED || writer_sem == SEM_FAILED) {
    perror("sem_open(3) failed");
    exit(EXIT_FAILURE);
}

//THREAD CREATION FOR THREAD POOL
pthread_t th[THREAD_NUM];
pthread_mutex_init(&mutexQueue, NULL);
pthread_cond_init(&condQueue, NULL);
int i;
//creating the threads
 for (i = 0; i < THREAD_NUM; i++) {
     if (pthread_create(&th[i], NULL, &startThread, NULL) != 0) {
         perror("Failed to create the thread");
     }
 }



int client_socket; //socket for client

 //live server code
while( (client_socket = accept(server_socket, NULL, NULL)) ){
     printf("\nConnection accepted by server %d,",server_name);
     //printf("\nserver %d about to call
      trainTicketMaster()\n",server_name); //for debugging
   //assign thread to call run trainTicketMaster, protected from concurrecy
    issues with sem mutex
```

```
        sem_wait(&mutexSem); //sem decremented
        Job t = {.functionToExecute = &trainTicketMaster, .arg1 =
         client_socket, .arg2 = server_name, .arg3 = ptr, .arg4 = shm_fd,
         .arg5 = reader_sem,.arg6 = writer_sem }; //ptr = shared mem ptr

        submitJobForExecution(t);
        sem_post(&mutexSem); //sem incremented
    }

    if (client_socket<0){
        perror("\nserver accept failed after accept loops\n");
     }

    //thread will return to pool when client exits program from menu
     for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join the thread");
        }
     }

     sem_destroy(&mutexSem); //destroy my mutexSem
     pthread_mutex_destroy(&mutexQueue); //destroy threads
     pthread_cond_destroy(&condQueue);

     //unlink from semaphore file
     if (  sem_unlink(SEM_READER_NAME) < 0 || sem_unlink(SEM_WRITER_NAME) < 0){
        perror("sem_unlink(3) failed");
     }
     close(server_socket);

     printf("\nServer Exited from main\n"); //for debugging

     return 0;
}
```