Threads and Synchronization

Project Overview

The Tonight Show with Jimmy Fallon is a wonderfully funny show that is on 5 days a week. It turns out that the tickets to go to a live recording of the show are free, but the problem is that tickets are limited and you need to be quick with getting a ticket before anyone else does. This assignment is a simulation of getting tickets to the tonight show by multiple people at the same time. This is a tricky assignment that illustrates threads, synchronization, and critical sections. The code is not long, but the details are subtle. You will write a simulation to order tickets from the Tonight Show phone ordering system.

Let us imagine that to get a ticket each person must make a phone call to the ticket system operator as soon as the tickets go on sale for the same day's show. Here we will assume that there are five call lines, but only three operators. When a phone call arrives, we see if there is a free line. If so, the phone rings; otherwise it is busy. Once the phone rings, we wait until there is a free operator, and when we get one, we pretend to get service. Each phone call is represented by a thread, and you will have more threads running than phone lines. There are several components to the project.

Semaphores

A semaphore provides a mechanism to avoid race conditions. In general, it provides mutually exclusive access to a resource. We can easily simulate a lock using a binary semaphore:

```
sem_t lock;

int main() {
  sem_init(&lock, 0, 1);
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, phonecall, NULL);
  pthread_create(&tid2, NULL, phonecall, NULL);
  ...
}

void* phonecall(void* vargp) {
  sem_wait(&lock);
  // critical section
  sem_post(&lock);
```

```
}
```

In this case, we initialize the semaphore (lock) to 1. This indicates that only 1 thread may be allowed to enter a critical section. This is typically referred to as a binary semaphore. What if we want to allow more than 1 thread into a section of code? For the problem we are trying to solve we have a limited resource (operators), but we have many phone calls that are being made to the ticket system concurrently. We can easily make a change to the initialization of the semaphore such that it allows more than 1 thread into a section of code at a time. Assuming we want to model three operators as a resource, we would create and initialize like so:

```
sem_t operators;
sem_init(&operators, 0, 3);
```

We would then have something like the following code in each phone call thread:

```
sem_wait(&operators);
// Proceed with ticket ordering process
sem_post(&operators);
```

So, the binary semaphore provides exclusive access for a single thread to a critical section of code, whereas an n-ary semaphore, typically referred to as a counting semaphore, provides a limit of **n** threads to **n** resources. For this simulation you will need to use both concepts to restrict access for 1 thread to shared state and to allow multiple threads to gain access to a limited number of operators.

Phone Calls

A phone call is represented by a thread function that will have a caller's **id** as a function local variable. The **id** variable is initialized by incrementing a global variable called **next_id** and assigning it to the **id** variable inside the thread function. Do you need synchronization here? The idea is that each time a new thread is created it will get a unique id that can be used to identify its output. To make things simple initially, write a thread function called **phonecall** that prints a message "this is a phone call" before the thread function ends. Later you will do the meaty part of the simulation.

Your main thread function will declare an array of thread id's (**pthread_t**) that will represent each phone call. Then write a loop calling **pthread_create** given the **phonecall** thread function and each corresponding entry of the array of thread id's. After we create each **phonecall** thread, we want to block the main thread until all the other **phonecall** threads complete. To do that, you need to write another loop that uses **pthread_join** on each thread.

Synchronization

Once you have the threads going, you'll want to do the real part of the simulation. Your **phonecall** thread function should declare the following static data and initialize the semaphores properly:

```
static sem_t connected_lock;
static sem_t operators;
static int NUM_OPERATORS = 3;
static int NUM_LINES = 5;
static int connected = 0;    // Callers that are connected
```

```
void* phonecall(void* vargp) {
      ...
}
```

The variable **connected** tells how many callers are currently connected (a call attempt is declared busy if connected==NUM_LINES). Access to **connected** must be controlled by a critical section, and is done by using the **connected_lock** binary semaphore. The **operators** counting semaphore is used once a call connects. Here is a sketch of the **phonecall** thread function algorithm:

1. Print that an attempt to connect has been made.
2. Check if the connection can be made:
    a. You'll need to test connected in a critical section
    b. If the line is busy, exit the critical section, print a message, and try again
    c. If the line is not busy, update connected, exit the critical section, and print a message, and continue to the next step.
3. Wait for an operator to be available (use a counting semaphore)
4. Print a message that the order is being taken by an operator
5. Simulate a ticket order by sleeping for a few seconds (sleep(3))
6. Print a message that the order is complete (and update the semaphore)
7. Update connected (using a binary semaphore)
8. Print a message that the call is over

Make sure that your output has the caller id to distinguish the output from different callers. Your program should print out something like the following:

```
...
Thread [THREAD_ID] is calling line, busy signal
Thread [THREAD_ID] has available line, call ringing
Thread [THREAD_ID] is speaking to operator
Thread [THREAD_ID] has bought a ticket!
Thread [THREAD_ID] has hung up!
...
```

Write all your code in a single **jimmy_fallon.c** file.

Testing

1. Test your program by running with 1 phone call.
2. Test your program by running with 10 phone calls.
3. Test your program by running with 20 phone calls.
4. Test your program by running with 50 phone calls.
5. Test your program by running with 100 phone calls.
6. Test your program by running with 240 phone calls.

240 is the maximum seating capacity of the tonight show.

Video Demonstration

You must provide a link to a 2-minute video demonstration. Your video should be short and get to the point, showing your program being **compiled** and **executed**. You must use your voice to guide the watcher and demonstrate all required outputs.

Grading Breakdown

1. Project Requirements (40 points)
    a. Binary semaphores are used properly to protect critical regions of code.
    b. A counting semaphore is used properly to restrict the use of resources (operators).
    c. All semaphores are correctly initialized and destroyed.
    d. A thread function exists and is implemented properly.
    e. A global integer **next_id** exists and is properly updated in the thread function.
    f. The phonecall thread properly updates the shared state for the number of callers in a critical section.
    g. The program properly prints output with the caller's id.
    h. The static modifier is used properly for both the thread local variables as well as any globals.
2. Design and Implementation (30 points)
    a. Functions are declared and used properly.
    b. Data structures and data types are declared and used properly.
    c. Variable and function naming is clear and help with understanding the purpose of the program.
    d. Global variables are minimized, declared, and used properly.
    e. Control flow (e.g., if statements, looping constructs, function calls) are used properly.
    f. Algorithms are clearly constructed and are efficient. For example, there is no extra looping, unreachable code, confusing or misleading constructions, and missing or incomplete cases.
3. Coding Style (5 points)
    a. Code is written in a consistent style. For example, curly brace placement is the same across all if/then/else and looping constructors.
    b. Proper and consistent indenting is adhered to across the entire implementation.
    c. Proper spacing is used making the code understandable and readable.
4. Comments (5 points)
    a. Comments in the code are used to document algorithms.
    b. Variables are documented such that they aid the reader in understanding your code.
    c. Functions are documented to indicate the purpose of the parameters and return values.
5. Video (14 points)
    a. The video shows the code being compiled from the command line.
    b. The video shows the code being executed and satisfies the output requirements.
    c. The video shows your program running to completion with a single phone call.
    d. The video shows your program running to completion with 10 phone calls.
    e. The video shows your program running to completion with 240 phone calls.
6. README.txt (5 points)
    a. The README.txt file is well written.
    b. The README.txt file provides an overview of your implementation.
    c. The README.txt file explains how your code satisfies each of the requirements.
7. GradeScope Submission (1 point)
    a. You automatically get a point for submitting to gradescope.

Submission

You must submit the following to Gradescope by the assigned due date:

- **`Makefile`** - the build file to use with the **`make`** program.
- **Source Files** - source and header files that provide the solution to this project.
- **`README.txt`** - this is a text file containing an overview/description of your submission highlighting the important parts of your implementation. You should also explain where in your implementation your code satisfies each of the requirements of the project or any requirements that you did not satisfy. The goal should make it easy and obvious for the person grading your submission to find the important rubric items. This text file should also clearly include a URL to your video for us to review. The video should not exceed 2 minutes in length and must provide a recorded voice to guide the viewer.

You must use the VM environment to write your code. Make sure you submit this project to the correct assignment on Gradescope!