Lab 12 - Concurrency 2

Due: December 10, 2017

1 Introduction **1**

2 Assignment **1**

2.1 Functions 2

3 Concurrent Programming Concepts **3**

3.1 Barriers 3

3.2 Mutexes 4

3.3 Condition Variables 4

4 Testing **5**

4.1 Debugging 5

5 Getting Checked Off **6**

1 Introduction

One of the most anticipated events at this year's Scare Games is the race across the Monsters University campus. With their honor and the Golden Mutex on the line, Mike and Sulley have entered the race with the rest of Oozma Kappa. Their primary competition is Monsters University's most prestigious fraternity, Roar Omega Roar.

On the day of the Scare Games, the race is revealed to be a relay race! The Scare Games staff has never hosted a relay race before, so they need your help to prepare the infrastructure for the race.

2 Assignment

In this week's assignment, you will use the concurrent programming skills you acquired in last week's lab, plus a few new concepts, to implement a simulated "relay race". The stencil code includes structures and functions that provide a simple race framework. The race should obey the following guidelines:

- The relay race is between two teams, with two racers (racer A and racer B) per team.

- Each racer is its own thread, which does some calculation that takes several seconds to complete (use the **`calculate()`** function in the provided stencil code. It calculates the 40th Fibonacci number using a naive recursive solution).
- All racer threads must begin execution at the same time (hint: use a barrier!).
- When the race begins, each team's racer A starts its computation. Racer B must wait for racer A to finish before it may proceed.
- When a racer B finishes, it attempts to lock a mutex associated with the race struct, and prints a victory or defeat message once it has acquired the lock.
- The team of the first racer B to lock the mutex and print its message is considered the "winner".

To complete this task you will need to use *mutexes*, *barriers*, and *condition variables*. See section 3 and/or the lecture slides for explanations of these concurrent programming structures.

2.1 Functions

Your job is to fill in the missing portions of the following functions:

- **`race_init()`** All you have to do in this function is initialize the barrier and mutex associated with the race struct, using **`pthread_barrier_init()`** and **`pthread_mutex_init()`**. Default barrier attributes are fine here.
- **`create_racer()`** This function should initialize the mutex and condition variable associated with a racer. This can be accomplished using **`pthread_mutex_init()`** and **`pthread_cond_init()`**. See the man pages for more detailed explanations of these functions.
- **`run_racer_a()`** This is the function that represents a racer A. Three things should happen in this function:
    1. Wait at the barrier associated with the race struct to ensure that it does not begin executing until all four racer threads are ready.
    2. Perform its racer's computation by calling **`calculate()`**.
    3. Set its racer's **`finished`** value to 1 and signal its condition variable. This informs the racer B thread that it is all set to proceed.
- **`run_racer_b()`** This is the function that represents a racer B. Five things should happen in this function:

1. Wait at the barrier associated with the race struct to ensure that the race does not begin until all four racer threads are ready.
2. Wait for the corresponding racer A thread to finish.
3. Inform the user that racer A has finished by calling **`handoff()`**.
4. Perform the racer's own computation by calling **`calculate()`**.
5. Lock the race mutex, call **`announce()`**, and then unlock the race mutex.

- **`start_race()`** In this function you should create the four racer threads using **`pthread_create()`**, and then wait for them all to complete by calling **`pthread_join()`** on each thread. As in last week's lab, the functions you will be running in separate threads require more than one argument. As such, you will once again need to wrap the arguments for each function in a struct before you pass them to **`pthread_create()`**. The struct you should use is **`args_t`**, which is provided in the stencil code. For more information, see the man pages for **`pthread_create()`** and **`pthread_join()`**.
- **`destroy_race()`** All you have to do here is destroy the race barrier, race mutex, and all condition variables/mutexes associated with the racers.

3 Concurrent Programming Concepts

3.1 Barriers

A barrier is useful structure in concurrent programming that allows for synchronization of an arbitrary number of threads.

- `int pthread_barrier_init(pthread_barrier_t *barrier,`

  `const pthread_barrierattr_t *attr, unsigned count)`
- `int pthread_barrier_destroy(pthread_barrier_t *barrier)`
- `int pthread_barrier_wait(pthread_barrier_t *barrier)`

When creating a barrier, the user specifies some number (**`count`**) of threads that must wait at the barrier before any of the waiting threads may continue. Once **`count`** threads are waiting at the barrier, all of those threads resume execution.

Barriers are critical to ensure that the relay race we are trying to implement is fair - the order in which racer threads are created should not impact the race's outcome!

You should create a barrier that prevents any racer thread from beginning execution until all the racers have been created.

To initialize a barrier, you can use the **`pthread_barrier_init()`** function. Threads which must wait at the barrier (i.e. racer functions) should call **`pthread_barrier_wait()`** on the barrier. As always, see the man pages for more information on these functions. Be sure to clean up your barrier using **`pthread_barrier_destroy()`** once it is no longer needed.

3.2 Mutexes

A mutex (short for **Mut**ual **Ex**clusion) is a “lock” that grants a thread exclusive access to a particular region of code or data. Only one thread can lock a mutex at any given time. You should already be familiar with using mutexes from last week's lab.

For this lab, you will use a mutex to ensure that only one team can win the race. Additionally, you will need to use mutexes in conjunction with condition variables, as outlined below.

3.3 Condition Variables

Like barriers, condition variables are an important concurrent programming structure that allow for the synchronization of multiple threads. The mechanism for synchronization is quite different, however: as the name suggests, condition variables synchronize threads based on the state of some condition.

Sometimes a thread must not proceed until a particular condition is satisfied. In situations like this, the thread *waits* for the condition, putting itself to sleep until it has been *signaled* that the condition has become true. If another thread updates this condition, it should signal the sleeping threads, causing them to resume execution. This process allows threads to avoid continuously polling the condition.

- `int pthread_cond_init(pthread_cond_t *cond,`

  `pthread_condattr_t *cond_attr)`
- `int pthread_cond_broadcast(pthread_cond_t *cond)`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `int pthread_cond_destroy(pthread_cond_t *cond)`

The concepts behind condition variables have a nice mapping to C programming, with the addition of a mutex. This mutex must be used to lock data or code associated with the condition of interest. The mutex must be locked when **`pthread_cond_wait()`** is called. When a thread calls this function, the thread is put to sleep and the mutex is unlocked. When a thread wakes up from waiting, it re-locks the mutex, so be sure to unlock that mutex once it is no longer needed. Signaling is a bit easier - all a thread needs to do to signal a condition variable is call **`pthread_cond_broadcast()`**, which signals all threads waiting on the indicated condition variable.

In this lab, you will use condition variables to ensure that the “relay” part of your race is correctly implemented. A given team's racer B thread must wait for its team's racer A thread to finish calculating before it may begin its own calculation.

For a more in-depth explanation of condition variables, see slide XXXI-10 in the lecture slides on the course website.

4 Testing

To test your program, run *./race* (built by the provided Makefile) several times. You should observe the following behavior:

- After a few seconds, the string “Racer A of team _____ has finished!” will be printed twice.
- After a few more seconds, the strings “Team _____ has locked the Golden Mutex and won the race!” and “Team _____ has come in second place, securing the Silver Mutex!” will be printed.

Note that the order in which teams hand off and finish may (and should!) vary from run to run.

This is not an error, but an intentionally produced race condition.

4.1 Debugging

Concurrent programming occasionally leads to a common class of bugs called deadlocks. To solve deadlocks, it is often helpful to look at the state of threads while they are stuck to identify what they are waiting for. **gdb** offers a set of helpful debugging tools for multithreaded programs that allows just that. To start **gdb** with your program, type

```
gdb ./race
```

Once **gdb** is started up, use the **run** command followed by the program arguments to run your code.

While your program is running, it is possible to interrupt and pause it so you may inspect the state of each thread. You may interrupt **gdb** using **CTRL-Z**.

From here, you may print a list of threads by typing **info threads**. This command enumerates the threads. You will see a ***** next to the current thread you are inspecting. To inspect a different thread, type **thread NUMBER** where **NUMBER** corresponds to the numbers in the list you printed.

You will find it is most helpful to print a backtrace with the **bt** command. This will print the functions the current thread is executing.

If you want to run a command for all threads, use **`thread apply all <command>`**. For example, if you would like to print a backtrace for all threads, run **`thread apply all bt`**.

Use this information to discern where any deadlocks are coming from and how to fix your code.

5 Getting Checked Off

When you are done, run **`33lab_checkoff lab12`** to submit and check off your lab. You may run this command as many times as you want; we will use the most recent grade and handin time recorded by this program.