Critical Concurrency

Entire Assignment due **2019-02-27 23:59**
Graded files:

- barrier.c
- queue.c
- semamore.c

Content

Learning Objectives

The learning objectives for Critical Concurrency are:

- Synchronization Primitives
- Common Patterns in Multi-Threaded Programs
- Thread-Safe Datastructures and Their Design

Overview

There are four main components to this lab, three of which are graded. These are Rendezvous (not graded), Semamore, Barrier, and Thread-safe Queue. Each of these represent very common synchronization problem (or slight twists on them) that will do you well to become familiar with.

Good luck!

Rendezvous (UNGRADED)

This is a problem for you to think about. We have provided a worked solution to this problem but PLEASE try to solve this problem before looking at the solution!

Problem description:

Given two threads, `a` and `b`, and the fact that both have two tasks (`a1` , `a2` , `b1` , `b2`), how do you get both `a1` and b1 to run before either `a2` and `b2` ? In `rendezvous.c` , you need to modify the two functions (`modifyB_printA` & `modifyA_printB`) using semaphores so that both quotes `A` and `B` are modified before being printed.

semamore.c

NOTE: A semamore is **NOT** a real thing! It is simply a made up clever name! This means you can't use them in future assignments (unless you re-implement it).

A normal semaphore blocks when the value within the semaphore reaches 0. A semamore blocks when it reaches 0, but also blocks when it reaches some maximum value. You can think of a semamore as a top-bounded semaphore. In semamore.c, you are given four functions to work on, `semm_init` , `semm_wait` , `semm_post` , `semm_destroy` . `semm_post` is the important difference. When `semm_post` reaches `max_val` (as defined in the semamore struct in semamore.h), it blocks.

There are four functions in total you will be writing:

- `void semm_init(Semamore *s, int value, int max_val);`
- `void semm_wait(Semamore *s);`
- `void semm_post(Semamore *s);`
- `void semm_destroy(Semamore *s);`

barrier.c

In rendezvous you saw an example of an one-time-use barrier. Now, you get to build code to support a reusable barrier. At the cost of being redundant, a reusable barrier is one that can get used more than once. Say you have threads executing code in a for loop and you want them to stay in sync. That is, each thread should be on the i'th iteration of the loop when every other thread is on the i'th iteration. With a reusable barrier, you can stop threads from going to the i+1'th iteration until all of them have finished the i'th.

Note that most barrier implementations (including the pthread library barrier) are "reusable", but never say so. This is because it simply does not make sense to have a "not-reusable" barrier. Thus, we are only iterating to you that the barrier your build should be reusable so that you understand what it means.

You can find more info in the barriers coursebook entry (http://cs241.cs.illinois.edu/coursebook/Synchronization#barriers)

Your goal is to implement the functions

- `int barrier_destroy(barrier_t *barrier);`
- `int barrier_init(barrier_t *barrier, unsigned num_threads);`
- `int barrier_wait(barrier_t *barrier);`

so that a `barrier_t` using these functions is a working reusable barrier.

queue.c

NOTE: Do not use semaphores or your semamore here.

Your task is to build a thread safe queue, that also may or may not be bounded, by implementing the functions in queue.c. The maxSize of the queue can be set to either a positive number or a non-positive number. If positive, your queue will block if the user tries to push when the queue is full. If not positive, your queue should never block upon a push (the queue does not have a max size). If your queue is empty then you should block on a pull. You should make use of the node struct to store and retrieve information. In the end, your queue implementation should be able to handle concurrent calls from multiple threads. `queue_create` and `queue_destroy` will not be called by multiple threads.

The queue is completely independent of the data that the user feeds it. The queue should make use of the constructors and destructors provided by the user to handle data.

Your goal is to implement the functions

- `queue* queue_create (ssize_t max_size);`
- `void queue_destroy (queue* this);`
- `void queue_push (queue* this, void* element);`
- `void* queue_pull (queue* this);`

Testing

Testing is ungraded, but highly recommended

Since the implementation of your semamore is quite close to an actual semaphore, please test this on your own in a variety of ways. Be careful of race conditions! They can be hard to find! We've given you a `semamore_tests.c` file to write tests in.

For `barrier_test.c` we have provided you with a simple test case. Feel free to expand on it, as it is not exhaustive/perfect. Learning how to use the barrier is just as important as writing it, since you will be using barriers on the Password Cracker MP :)

For `queue_test.c` we would like you to write tests yourself. Learning to write tests for multi-threaded code is very important. You will also be using this queue in the Password Cracker MP :) (we will give you a working version; you will not be penalized on the MP for not successfully completing the lab)

Thread Sanitizer

We have another target executed by typing `make tsan`. This compiles your code with Thread Sanitizer.

ThreadSantizer is a race condition detection tool. See this page (http://cs241.cs.illinois.edu/coursebook/Background#tsan) for more information.

We will be using ThreadSanitizer to grade your code! If the autograder detects a data race, you won't automatically get 0 points, but a few points will be deducted.

Helpful Hints and Notes

- Make sure you thoroughly test your code! Race conditions can be hard to spot!
- Attempting to visualize your code or diagram it in certain cases can sometimes be a huge aid and is highly recommended!

** In any of `semamore.c`, `barrier.c`, or `queue.c` you may not use semaphores or pthread_barriers **

ANYTHING not specified in these docs is considered undefined behavior and we will not test it For example, calling queue_push(NULL, NULL) can do whatever you want it to. We will not test it.