EC440: Project 3 – Thread Synchronization

Project Goals:

- To make it possible for threads to safely share data
- To enable synchronizing threads around specific points

Collaboration Policy:

You are encouraged to discuss this project with your classmates and instructors. You **must** develop your own solution. While you **must not** share your threading code with other students, you **are encouraged** to share test case code that you developed to test your solution for this assignment.

Deadline:

Project 3 is due March 22 at 4:30 PM EDT.

Project Description:

In the previous project, we implemented a part of pthreads that enables creation and execution of multiple threads. Now, we are going to add some more features to that library to support interaction between threads through synchronization.

The main features we need to add are:

1. Thread locking and unlocking, to prevent your scheduler from running at specific times.
2. An implementation for pthread barriers, to ensure one thread can wait for another thread to finish, and to collect a completed thread's exit status.
3. Support for mutexes in your threads, which will enable mutual exclusion from critical regions of multithreaded code.

Required New Functions to Implement

You will implement functions to support thread synchronization through mutexes and barriers. Each of these features includes init/destroy functions and one or more functions that control the synchronization state.

Mutex Functions

```
int pthread_mutex_init(
      pthread_mutex_t *restrict mutex,
      const pthread_mutexattr_t *restrict attr);
```

The `pthread_mutex_init()` function initializes a given `mutex`. The `attr` argument is unused in this assignment (we will always test it with NULL). Behavior is undefined when an already-initialized mutex is re-initialized. Always return 0.

```
int pthread_mutex_destroy(
      pthread_mutex_t *mutex);
```

The `pthread_mutex_destroy()` function destroys the referenced `mutex`. Behavior is undefined when a mutex is destroyed while a thread is currently blocked on, or when destroying a mutex that has not been initialized. Behavior is undefined when locking or unlocking a destroyed mutex, unless it has been re-initialized by `pthread_mutex_init`. Return 0 on success.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

The `pthread_mutex_lock()` function locks a referenced `mutex`. If the mutex is not already locked, the current thread acquires the lock and proceeds. If the mutex is already locked, the thread blocks until the mutex is available. If multiple threads are waiting on a mutex, the order that they are awoken is undefined. Return 0 on success, or an error code otherwise.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The `pthread_mutex_unlock()` function unlocks a referenced `mutex`. If another thread is waiting on this mutex, it will be woken up so that it can continue running. Note that when that happens, the woken thread will finish acquiring the lock. Return 0 on success, or an error code otherwise.

Barrier Functions

```
int pthread_barrier_init(

      pthread_barrier_t *restrict barrier,

      const pthread_barrierattr_t *restrict attr,

      unsigned count);
```

The `pthread_barrier_init()` function initializes a given `barrier`. The `attr` argument is unused in this assignment (we will always test it with NULL). The `count` argument specifies how many threads must enter the barrier before any threads can exit the barrier. Return 0 on success. It is an error if count is equal to zero (return EINVAL). Behavior is undefined when an already-initialized barrier is re-initialized.

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

The `pthread_barrier_destroy()` function destroys the referenced `barrier`. Behavior is undefined when a barrier is destroyed while a thread is waiting in the barrier or when destroying a barrier that has not been initialized. Behavior is undefined when attempting to wait in a destroyed barrier, unless it has been re-initialized by `pthread_barrier_init`. Return 0 on success.

`int pthread_barrier_wait(pthread_barrier_t *`*`barrier`*`);`

The `pthread_barrier_wait()` function enters the referenced `barrier`. The calling thread shall not proceed until the required number of threads (from `count` in `pthread_barrier_init`) have already entered the barrier. Other threads shall be allowed to proceed while this thread is in a barrier (unless they are also blocked for other reasons). Upon exiting a barrier, the order that the threads are awoken is undefined. Exactly one of the returned threads shall return PTHREAD_BARRIER_SERIAL_THREAD (it does not matter which one). The rest of the threads shall return 0.

Recommended New Functions to Implement

We also recommend that you create new `static void lock()` and `static void unlock()` functions. Your lock function should disable the timer that calls your schedule routine, and unlock should re-enable the timer. You can use the `sigprocmask` function to this end (one function using SIG_BLOCK, the other using SIG_UNBLOCK, with a mask on your alarm signal). Use these functions to prevent your scheduler from running when your threading library is *internally* in a critical section (users of your library will use barriers and mutexes for critical sections that are external to your library).

Given Code:

You are expected to extend the threading implementation and tests you started in homework 2.

No new code is provided with this assignment.

Student Environment Update

Many students have requested that man pages be installed in ec440-student-env so that you don’t need to enter and exit the environment to switch between reading manuals and testing code in the environment.

The student environment has been updated to include man pages! To get the update, run:

```
docker pull docker.io/dannosliwcd/ec440-student
```

Hints:

1) **Test early and test often.** “The program does not crash” does not necessarily imply “the program works.” Determine expected behavior ahead of time, then verify that it actually occurs.
2) Consider how you will track the internal state of your mutexes and barriers. You have access to pthread_barrier_t or pthread_mutex_t data for each of those interfaces. You might notice that pthread_barrier_t doesn’t make it easy to see how its internal structure looks (it is just a union of size and align). There are probably a lot of ways you can use this structure, but two relatively simple ones come to mind:
    a) Store the address of your own malloc’d data structure inside that union.
    Pro: This works with arbitrary data structures of any size beyond a pointer’s size.
    Con: You have to deal with malloc. If you choose to cast and assign your pointer to the `long __align` member, watch out for issues with signed type conversion.
    b) Create your own separate wrapper union of both pthread_barrier_t and your own data structure.
    Pro: No additional memory allocation needed.
    Con: Only works if your data structure is the same size or smaller than pthread_barrier_t. May be confusing to read these nested unions.
3) Consider *what* kind of internal state to track for mutexes and barriers. You’ll likely need to track something that indicates the current state of the synchronization object. You’ll likely also need some kind of list of which threads are waiting to be allowed to run.
4) You may be aware of a value that can be used for static initialization of pthread mutexes. You **do not** need to support use of that initializer (which may restrict your implementation options). We will not use that feature in our grading system.
5) You will likely need to add an additional BLOCKED state to your threads. A blocked thread should not be selected for execution by your scheduler. Both barriers and mutexes can result in a blocked state.

Submission Guidelines:

Your threading library must be written in C and run on the Linux Gradescope environment. Submit your ***source code and makefile*** to Gradescope. When we run `make` in your submission, a threads.o object file must be produced, containing the compiled definitions of the required functions listed in this homework prompt. This file must not define a main function.

Your threading library **must not** depend on libpthread. You are implementing an alternative to libpthread in this assignment.

Your final submission must also include a `README` file (README.md is also okay). If you relied on any **external resources** to help complete this assignment, note the resource and the challenge it helped resolve. **If not**, say so. Use this file to provide any **additional notes** you would like to share with instructors about your submission. Aside from that, it just needs a **short description** of the purpose of the project. But you are permitted to include additional details that *you* want in your README.

You can extend the same README you started in homework 2.

Oral Exams

When the submission deadline is reached, we will start oral exam sessions over Zoom. In these sessions, we will ask you to explain how some parts of your program work. Details about how to schedule an oral exam time will be posted in the week leading up to the exams.