Assignment Objectives

The objective of this assignment is to give you hands-on experience with parallel programming and various synchronization methods. You are going to implement a synchronization primitive (Barrier) in the first part and in the second part you are required to implement a thread-safe list using some synchronization primitives.

Part I - Barrier

Introduction

An N-process barrier is a synchronization mechanism that allows N threads, where N is a fixed number, to wait until all of them have reached a certain point. Once all threads have reached this certain point (the barrier), they may all continue. For example, we can use barriers to synchronize N people who watch a movie, if each person runs the following thread:

```
buy_popcorn();
buy_something_to_drink();
// wait for everyone else before we start the movie
barrier.wait();
watch_the_movie();
// wait until everyone leaves the cinema
barrier.wait();
close_the_door();
```

In the above example, each thread that calls `barrier.wait()` blocks until the last thread enters the synchronization barrier, and only then all threads are released. Note that the same barrier is used twice to synchronize all threads, so once a thread unblocks, it can re-enter the barrier and block again.

Implementation Requirements

Implement the following Barrier API, as defined in the provided `Barrier.h` header file:

1. **`Barrier(unsigned int num_of_threads);`**
    - This is the constructor of the Barrier. it receives one parameter `num_of_threads` - which is the number of threads that this barrier is supposed to handle.
    - Return value: A Barrier Object.

2. **`void Barrier::Wait();`**
    - Block and wait until N threads reach this point.
      If you're the N^{th} thread to call this method wake up the rest of the threads else, block.
    - Parameters: none.
    - Return value: none.

3. **`unsigned int Barrier::WaitingThreads();`**
    - Returns the number of threads which are currently waiting on this barrier.
    - Parameters: none.
    - Return value: number of threads waiting on the barrier.

4. **`~Barrier();`**
    - This is the deconstructor of the Barrier. destroys or releases any resources that it used (like semaphores and mutexes)
    - Parameters: none.
    - Return value: none.

Important Notes and Tips

- **Don't modify the signatures of the methods defined in the Barrier.h header file.** We will test your code according to this interface.
- You may add any class members you need (integers, Boolean, data structures, …), but **your implementation should use only the following synchronization primitives: semaphores and mutexes.**
  To put it another way, you are not allowed to use any of the following types and their associated functions: `pthread_cond_t`, and obviously not `pthread_barrier_t`.
- You can use as many semaphores as you need.
- In your implementation, you can ignore any errors that the semaphore and mutex functions may return. For instance, according to the man pages, `sem_init()` returns the value EINVAL when the initial value of the semaphore exceeds SEM_VALUE_MAX; we will not check such cases in our tests.
- Use only standard POSIX threads and synchronization functions. You are not allowed, of course, to use the C++11 thread support or the C++11 atomic operations libraries.
- You should try to make your implementation as concurrent and as efficient as possible. The performance of your solution can affect your grade. However, efficiency must not come at the cost of correctness!
- **Make sure your code is running on your virtual machine or Azure as was explained and done in previous homework.**