Lab2

PartA

1. *Compile and run the atomic counter test using the nosync counter implementation:* `./test_atomic_counter nosync`*.* ***Do not modify the nosync implementation.*** *Does it work? Why or why not?*
    - No, it doesn't work. It passed single-threaded tests, but failed on multithreaded tests. Because we expected the result to be 0, there's inconsistency in the actual value due to lack of synchronization.
2. *Implement the* ***lock-based*** *atomic counter in* `atomic_counter_lock.cpp`*. You will find* `TODO` *marks in the functions that you need to change. You should also read the comments in the base class* `atomic_counter` *in the* `atomic_counters.hpp` *file and also take a look at the* `atomic_counter_lock` *class definition as well. Compile (using the makefile) and run the test:* `./test_atomic_counter lock`*. Does it work? Why or why not?*
    - Before the implementation, it still failed due to a data race issue. The issue was solved after adding the `std::lock_guard` class and **constructing** an object named `guard`. We pass `m_lock` to the **constructor** as an argument. This successfully addresses the data race problem.
3. *Implement the* ***atomic inc/dec-based*** *atomic counter in* `atomic_counter_atomic_incdec.cpp`*. You will find* `TODO` *marks in the functions that you need to change. Use the* `fetch_add` *and* `fetch_sub` *functions available in C++. For this, you will also need to modify the header file, to make sure that the* `m_value` *variable if of the correct type. Compile (using the makefile) and run the test:* `./test_atomic_counter atomic_incdec`*. Does it work? Why or why not?*
    - Failed before we modified the code. (Show code)
4. *Implement the* ***atomic CAS-based*** *atomic counter in* `atomic_counter_atomic_cas.cpp`*. Use the* `compare_exchange_strong/weak` *functions in C++. Compile (using the makefile) and run the test:* `./test_atomic_counter atomic_cas`*. Does it work? Why or why not? How do the* `compare_exchange` *functions work?*

- Failed before we modified the code. (Show code). `compare_exchange` this is a common atomic operation provided by C++ atomic types. It involves three main steps:
    1. **Compare**: The value of the atomic object is compared with a non-atomic value.
    2. **Exchange**: If the comparison is true, then a new value is atomically exchanged with the value of the atomic object.
    3. **Load**: If the comparison is false, then the current value of the atomic object is atomically loaded and stored.

5. *Observe the performance (in operations per second) of the different tests and rank them from fastest to slowest. Why do they end up in this order?*

    - `lock` > `atomic_incdec` > `atomic_cas`
      Using lock and unlock creates a very small critical section to protect a value, and locks actually suspend thread execution, freeing up CPU resources for other tasks. However, they incur obvious context-switching overhead when stopping or restarting the thread. On the contrary, using atomic operations means it doesn't wait and keeps trying until success (e.g., busy-waiting), so it doesn't incur context-switching overhead, but it also doesn't free up CPU resources.

      In this case, we can infer that context-switching is less costly than busy-waiting. This is likely because when doing context-switching, there isn't much data that needs to be moved back and forth (only the program counter and a value, perhaps?).

      For `atomic_cas`, it will store a value regardless of whether the comparison is true or false, and it has one more instruction (`compare`) every single time. So it's reasonable to expect that `atomic_cas` is slower than `atomic_incdec`.

PartB

1. *Compile and run the `std::mutex` based lock: `./test_user_lock mutex`. Does it work? Why or why not.*

    - It works because value `v` has a data racing issue while two threads are running the program. However, the program uses `lock` and `unlock` to create a critical section, guaranteeing that no two processes can exist in the critical section at any given point in time.

2. *Implement mutual exclusion using Dekker's algorithm in `user_lock_dekker.cpp`. Do not use any locks, instead use atomic operations where necessary. Test your implementation: `./test_user_lock dekker`. Does it work? Why or why not.*

    - We can only use atomic value with Dekker's algo to implement mutual exclusion, it's because if one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.

3. *What is the memory model of the language? What about the memory model of the underlying hardware? Which memory model are we programming for here?*

    - For the language, we use the C++ memory model. The program was running on `arm64-apple-darwin23.3.0`, which defined the memory model under this architecture. We were using the C++ memory model while programming, and there's no need to concern ourselves with the memory model of the underlying hardware, as the compiler will handle that.

4. *What if we had a different programming model?*

    - I don't think that will be a problem because the compiler will still convert the code into validated instructions for the memory model of the underlying hardware.

5. *Observe the performance of the two different lock implementations. Which one is faster? Why?*

    - Using `lock` and `unlock` is approximately twice as fast as using an `atomic` value with `Dekker's algorithm`. For `./test_user_lock mutex` creates a very small critical section to protect a value, and locks actually suspend thread execution, freeing up CPU resources for other tasks, but incurring obvious context-switching overhead when stopping/restarting the thread. On the contrary, `./test_user_lock dekker` uses atomic operations, which means it doesn't wait and keeps trying until success (e.g., busy-waiting), so it doesn't incur context-switching overhead, but it also doesn't free up CPU resources.

      In this case, we can infer that context-switching is less costly than busy-waiting. I reckon it's because when doing context-switching, there isn't much data that needs to be moved back and forth (only the program counter and a value maybe?)