Data Race
Andy Tran

1. Build the given source code with two threads accessing the shared vector. Is the code executing correctly? Use thread sanitizer to analyze the code for data races.

Building the given source code with two threads accessing the shared vector does not execute correctly and runs into a data race issue. It gives the warning from thread sanitizer of data race.

```
WARNING: ThreadSanitizer: data race (pid=3106897)
```

Then thread sanitizer would give the summaries like these below.

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA+0x2a6c) in std::vector<int, std::allocator<int> >::push_back(int const&)
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA+0x35d2) in void __gnu_cxx::new_allocator<int>::construct<int, int const&>(int*, int const&)
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA+0x2c7a) in std::vector<int, std::allocator<int> >::size() const
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA+0x442a) in std::enable_if<std::__is_bitwise_relocatable<int, void>::value, int*>::type std::__relocate_a_1<int, int>(int*, int*, int*, std::allocator<int>&)
```

2. Fix data race using mutex. Demonstrate that your code has no data races (via results and sanitizer output)

To fix the data race issue, I added a mutex to help lock and unlock the thread when they are accessing the shared vector.

```
void increment() {
    for (int i = 0; i < 1000000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);

        vec.push_back(i);
    }
}
```

After implementing the mutex, the resulting print statement was able to print the vector size being 2000000 and the thread sanitizer did not display any information because there were no longer any issues.

```
andyqt1@circinus-37 17:52:26 ~/131
$ g++ PartA_mutex.cpp -o PartA_mutex -Wall -std=c++11 -fsanitize=thread -fPIE -pthread -pie
andyqt1@circinus-37 17:52:46 ~/131
$ ./PartA_mutex
Vector size: 2000000
```

3. Now use compare_exchange primitive that was shown during the discussion. What are the benefits of using such synchronization primitives?

In place of the mutex, I implement the compare and exchange primitive that would lock the current thread if another thread was running.

```cpp
void increment() {
    bool expected;
    for (int i = 0; i < 1000000; ++i) {
        expected = false;
        while (!lock.compare_exchange_strong(expected, true)) {
            expected = false;
        }
        vec.push_back(i);
        lock = false;
    }
}
```

And the result of the compare and exchange primitive working are the same as the mutex working results.

```
andyqt1@circinus-37 17:52:53 ~/131
$ g++ PartA_comp_exc.cpp -o PartA_comp_exc -Wall -std=c++11 -fsanitize=thread -fPIE -pthread -pie
andyqt1@circinus-37 17:53:25 ~/131
$ ./PartA_comp_exc
Vector size: 2000000
```

The potential benefit of using the compare and exchange synchronization primitives is that it could be faster than using the mutex synchronization primitive because the overhead could be less. Another benefit is that it helps prevent deadlock of resources and fixes the data race issue that was present in the original source code.

4. (Extra 5 points) Use sleep_for function so that the main thread will sleep for 100 milliseconds between two thread creations. Does the code work? Does this method fix the data race problem? Why or why not?

```cpp
int main() {
    std::thread t1(increment);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Vector size: " << vec.size() << std::endl;
    return 0;
}
```

The code does not work and still has the data race issue. The warning is the same and the summaries are the same as it was before implementing the mutex or compare and exchange synchronization primitives.

```
WARNING: ThreadSanitizer: data race (pid=3104105)
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA_comp_extra_credit+0x2c1c) in std::vector<int, std::allocator<int> >::push_back(int const&)
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA_comp_extra_credit+0x321a) in std::vector<int, std::allocator<int> >::size() const
```

```
SUMMARY: ThreadSanitizer: data race (/home/andyqt1/131/PartA_comp_extra_credit+0x4d8a) in std::enable_if<std::__is_bitwise_relocatable<int, void>::value, int*>::type std::__relocate_a_1<int, int>(int*, int*, int*, std::allocator<int>&)
```

The data race issue is still present because the time that the thread is sleeping is not long enough for the other thread to finish, so they are both still trying to access the same vector at the same time. If the time was greater than it could work as a potential solution to eliminate the data race in this example.