**Kailey Cozart**

**Distributed, Scalable Computing**

**LM2**

**Run Instructions/Note:** On a couple of these exercises, I couldn’t get the code to work with Ubuntu/VS Code, but I mainly used XCode. In order to run, I just copy pasted the cpp file in to an empty XCode project and ran it.

**Part 1:** Explain your findings! Walk me through your reasoning behind your code structure and why it proves one way or another the thread safeness of vector.

For push\_back() and pop\_back(), I used one piece of code. I had a large loop where the threads would first push\_back() a value and then pop\_back() the last value in the vector. If these vector functions were thread safe, the values would be pushed and popped back, and the program would complete successfully, with an vector size of 0. On the other hand, if the vector functions were not thread safe, too many things would be popped or pushed back, either resulting in an error or a vector larger than 0. Because my code consistently ended at 0, I know that these functions are not thread safe.

\*\*For operator[], I created a vector initialized with a 1 for every element. Then, I used the operator[] to re-assign the value (tid \* 33) to all vector elements. If the operator[] was not thread safe, I would expect that some of the same values to be changed multiple times, so instead of getting 1\*33, 2\*33, 3\*33, etc. for my answers, I would get 1\*33, 2\*33\*3, etc. Since the calculations worked out, I think that the operator[] is thread safe.

\*\*For insert(), I created an uninitialized vector, and then inserted items. I would think that, if multiple things were going to be inserted into the same spot, it would crash. Since it crashed a lot, I think it isn’t thread safe.

\*\*For swap(), I created a vector where all values were 33, then I changed each value by swapping each element with the value of 66 that was stored in a separate vector. If it wasn’t thread safe, I would think that it would crash when two elements were grabbed. Because it didn’t crash after many runs, I think that it’s thread safe.

**Part 2:** What was your mutex strategy? What are your critical sections?

My mutex strategy was to put the mutex in the bank class. My critical sections were when the balance variable was being changed, so the withdrawing and depositing were the critical sections in this exercise. Therefore, when withdrawing, the mutex would lock so that the conditions could be checked. If there was money to withdraw, the money would be withdrawn and the mutex would unlock. Otherwise, the mutex would unlock and wait before locking and checking the condition again. The deposit function also had the mutex lock when it started depositing and had the mutex unlock when the deposit was completed.

**Part 3:** Are you sure your code will not dead-lock? Why? How many mutex structures are you using?

I have a mutex that is locked and unlocked by three different functions: PriorityWithdrawal(), Withdrawal(), and Deposit(). I am sure that my code will not dead-lock because, if the condition for a function to occur is not met, the mutex is unlocked and a wait period is given before the mutex is locked and the conditions for that function are set again.

**Part 4:** (No Question)

**Bonus:** (No Question)