Ty Abbott

README

Program Correctness

One way for a concurrent program to be incorrect, is if individual threads are not run in the correct sequential order. In both the push and pop methods, there is no return from the method until the pop or push has been executed. For the thread to get the next instruction (pop or push) it must finish what it is doing. This means each thread will run sequentially, and will never get out of order.

Atomic Variables

To solve the issue of threads popping the wrong value from the stack, because in the middle of execution a new value was pushed I made the stack atomic. The entire stack is atomic, meaning the values cannot be updated in multiple threads at the same time. Using compare and set, the thread will check to see if the stack has changed, if no changes to the stack have occurred the thread will perform the push or pop. If a change has occurred, the head of the stack will get updated in the method and rechecked.

Similarly with the stack, the counter must also be atomic. If the counter is not atomic it can updated to the wrong value. The reason this happens is because incrementing is actually multiple instructions. The value must be read, incremented, and then stored again. If the counter is at 5. A scenario with a non-atomic counter is one where Thread 1 calls increment and shortly after Thread 2 calls increment. Thread 1 and 2 simultaneously read the value of 5. Both threads simultaneously increment it to 6, and Thread 1 writes it as 6 first, then Thread 2 writes it as 6 again. This is incorrect as the counter should read as 7, not 6. The method of using an atomic counter is the same as the stack. I use a compare and set to check if the variable has been incremented. If it has, then the new value is retrieved and it is rechecked.

In Java Atomic Variables and Volatile variables are different. A volatile variable will not be optimized by the compiler and so it will not be cached. Instead the variable is always accessed from hardware memory. A volatile variable can change at any time. The volatile keyword lets threads know the variable can be changed at any time, but it does not have access to atomic instructions. It can only perform 1 instruction at a time, for instance just read or just write. If you want to read in the value and modify it before it possibly changes you need an atomic variable with atomic instructions. An atomic variable has the same memory properties as a volatile variable.

Locks

Using an atomic stack, and utilizing the compare and set (CAS) method was key to getting rid of locks. Using CAS there is no need to lock. When multiple threads are accessing the stack at the same time, the thread that calls CAS first with the original stack “wins” and gets to update and return from the method. The other threads are then sent back to retrieve the updated stack using the get method and trying to execute again.

Example scenario: The stack is currently [5, 32, 90, 7], with 5 being the top of the stack. There are three threads running, and they all simultaneously call different instructions. Thread 1 calls pop. Thread 2 calls push 13. Thread 3 calls push 4. They all get the same copy of the stack at the beginning of their execution, [5, 32, 90, 7]. Thread 1 pops and has [32, 90, 7]. Thread 2 pushes and has [13, 5, 32, 90, 7]. Thread 3 pushes and has [4, 5, 32, 90, 7]. They all call compare and set, but thread 3 calls it slightly faster and has its push executed. The stack is now [4, 5, 32, 90, 7]. So when Thread 1 and 2 execute CAS, they return false because for thread 1: 4 does not equal 5. For thread 2: 4 does not equal 13. So they must now redo the execution with the newly updated Stack.

Similarly to locks, an atomic variable will not update with the wrong value. The main difference here is that threads are never locked out. They keep running without being suspended. The trick is to handle this scenario where the CAS is false, and redo the operation.