CSC 464 – Assignment Two

Part Three

Michael Windels
November 26, 2018
V00854091

For Dr. Yvonne Coady's CSC 464 (Fall 2018) class.
Repository: https://github.com/MWindels/uvic-csc464-a2

3 Lock-Free Double-Counting Reference Counter

This section describes my implementation of a lock-free reference counter that double-counts all accesses to an object. This section also includes a description of the tests used to evaluate the correctness of the counter's implementation, and an analysis of some testing results. This lock-free double-counting reference counter is (an important) part of my lock-free hash table, which I will implement as my project for the end of the term.

3.1 Description of the Implementation

A double-counting reference counter is a reference counter which counts references twice. In this lock-free implementation, the counter has two parts. The first part is an external counter comprised of a pointer to the inner counter, and a count variable. This external structure is an atomic object. The second part is an internal counter allocated on the heap, which contains the data being protected, and an atomic count variable. When the pointer field in the external counter is modified, or the reference counter is destroyed, the internal counter's count field is decremented by the value of the external counter's count field. The internal counter will persist on the heap until its count variable reaches zero (i.e. when no more threads can observe it).

The count field in the external counter counts the number of threads which have accessed the data. Conversely, the count field in the internal counter counts the number of threads which have given up access to the data. Eventually, both values should be equal. Hence when the pointer field in the external counter changes (thus decrementing the count field in the internal counter by the value of the external counter's count field), we should expect the value of the internal counter's count field to be zero once the last thread observing the data gives up access to the data. Therefore, this scheme prevents the internal counter (and hence the data) from being deleted until all threads have stopped observing the data, and when no new threads can gain access to the data.

The `double_ref_counter` class which I have implemented in C++ serves as exactly the kind of lock-free double-counting reference counter described above. The `double_ref_counter` supports three operations, `obtain`, `replace`, and `try_replace`.

To `obtain` a reference, the external counter is atomically read, then its count field is incremented, then it is swapped back in using the atomic Compare-And-Swap (CAS) operation. This procedure loops until CAS succeeds. Once the operation succeeds, the pointer to the internal counter is returned in an RAII (Resource Allocation Is Initialization) object to the user (called a `counted_ptr`). This RAII object acts as an interface to the data in the internal counter. When the RAII object is destroyed, the internal counter's count field is incremented, and any necessary clean-up is performed.

To `replace` a reference, a new internal counter is created on the heap with the parameters passed to `replace`. Then, the external counter is atomically swapped (with CAS) for a new external counter which points to the newly allocated internal counter. The old internal counter's count field is decremented by the value of the old external counter's count field, and the old internal counter is deleted if necessary.

The `try_replace` operation behaves much like a `replace` operation, but it requires an additional parameter that specifies the expected value of the data (as a `counted_ptr`). If the pointer field in the external counter differs from the pointer field in the expected value parameter, then `try_replace` will fail, return false, and not modify the `double_ref_counter`. Otherwise, it will return true, and behave like a `replace` operation.

My implementation can be found at the first link below. My implementation was inspired by Shlomi Steinberg's double-counting reference counters. Steinberg also used their double-counting reference counters to help implement their own lock-free (and wait-free) hash table. Though my term project is also a lock-free hash table, I plan to take a different approach. Their implementation can be found at the second link below.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/lib/p3/double_ref_counter.hpp

https://shlomisteinberg.com/2015/09/28/designing-a-lock-free-wait-free-hash-map/

3.2 Description of the Tests

To test my implementation, I have put together a simple program which spins up three different categories of threads. Each thread performs one of the three `obtain`, `replace`, or `try_replace` operations. When launching the testing program from the command line, we can specify the number of threads in each category to spin off. The threads are started in a random order, which allows us to test the three operations arbitrarily interleaved with each other. By specifying a large number of threads, we can also test the correctness of the `double_ref_counter` under stress.

In the tests, `loud_object` objects are stored in the `double_ref_counter`. A `loud_object` prints some output every time an instance is constructed, copied, moved, or destroyed. This allows us to see the lifetime of the `loud_object`, and watch where it begins and ends in relation to operations being performed on the `double_ref_counter`. The testing code is (like the implementation) written in C++, and can be found at the following link.

https://github.com/MWindels/uvic-csc464-a2/blob/master/src/tst/p3/ref_tester.cpp

To run the tests for yourself, you will need to compile them first. I have included a Makefile in the repository to help compile the test program. This Makefile assumes you have the GNU C++ compiler. To compile the test program, simply type `make` into the command line while in the same directory as the Makefile. Then, simply type `ref_tester obtainers replacers try_replacers` into the command line, where `obtainers`, `replacers`, and `try_replacers` are the number of threads which will call `obtain`, `replace`, and `try_replace` respectively.

3.3 Testing Results and Correctness

Figure 3.1 shows an example of the test program run with thirty `obtain` threads, one `replace` thread, and no `try_replace` threads. The output clearly demonstrates how the `loud_objects` are created before any thread observes them, and deleted after every thread is finished observing them.

Only the `obtain` threads (and the `try_replace` threads) are considered to observe data in the `double_ref_counter`. The `replace` threads do not return any information about the data in the `double_ref_counter`, so they are not considered to observe the data.

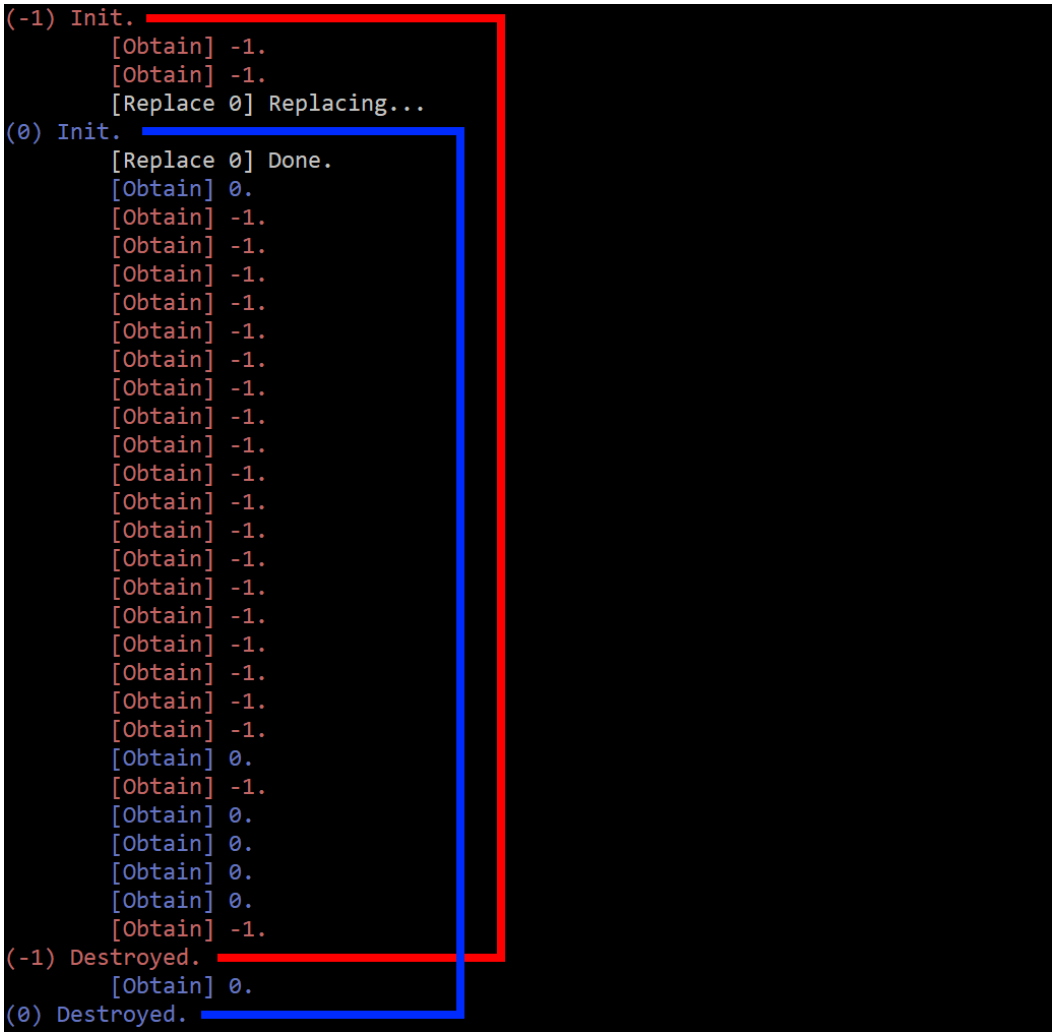

Figure 3.1 – The testing program run with thirty `obtain` threads, and one `replace` thread.

Notice how every `obtain` happens after a corresponding construction ("(*x*) Init.") and before a corresponding destruction ("(*x*) Destroyed."). This indicates that the data is (correctly) persisting until all observing threads are finished observing. However, some of the `obtain` threads appear to observe the value "-1" even after the `replace` thread has changed the value to "0". It is likely that these obtain threads did in fact observe the value "-1". This is possible (and correct, even after the value has been replaced) because those threads (atomically) obtained the pointer to the value "-1", and simply did not print their value until after the `replace` had fully executed.

The fact that the "-1" value can still be observed even after it was replaced also indicates that the internal counters (which store the data) are correctly persisting on the heap. Furthermore, it indicates that the RAII class (`counted_ptr`) that protects access to the data in the internal counter is doing its job properly. Once an object is destroyed, we see no further attempts to access them.

Additional testing also reveals that the `try_replace` operation works as intended. Figure 3.2 demonstrates a test with thirty `obtain` threads, and two `try_replace` threads. One of the threads successfully replaces the value, while the other thread fails, and does not replace the value.

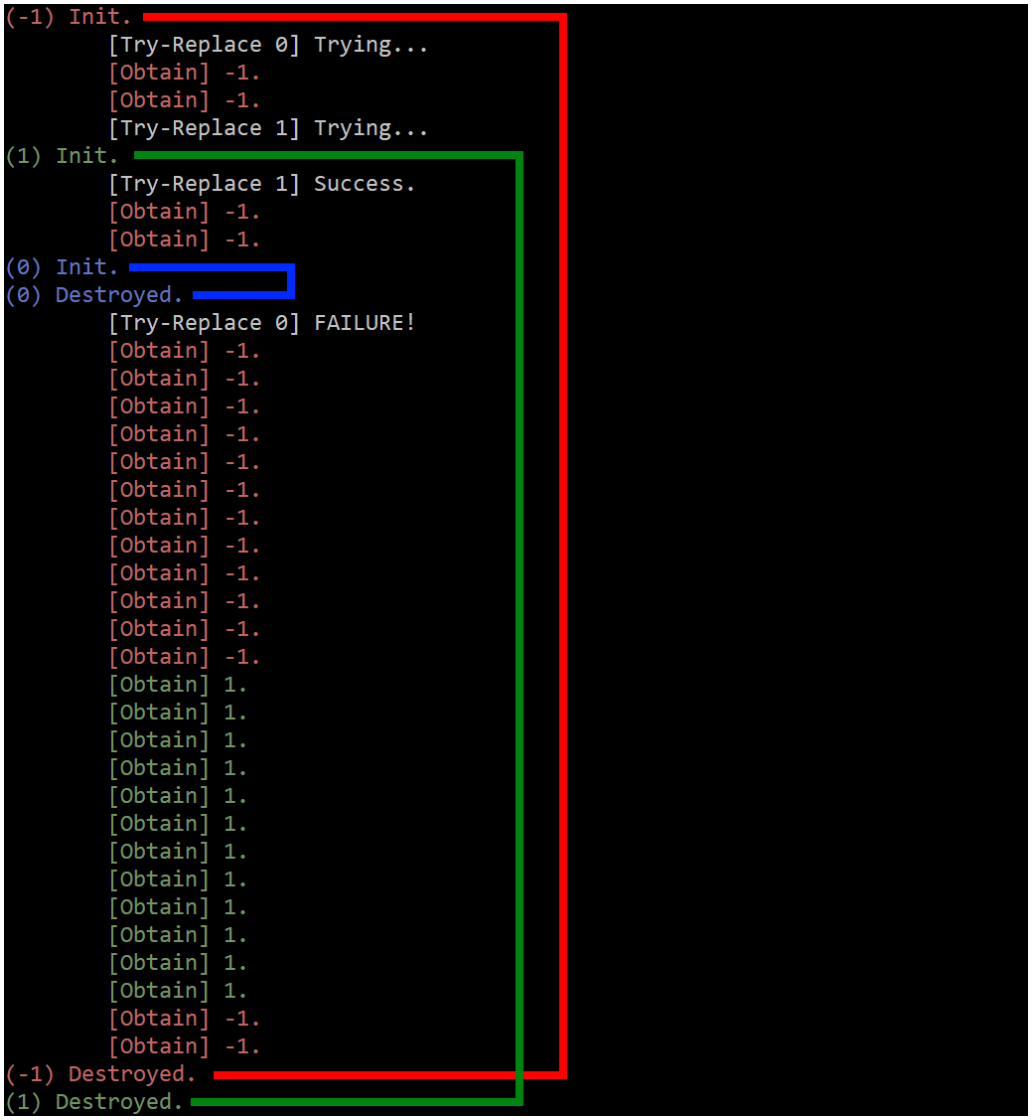

Figure 3.2 – The testing program run with thirty `obtain` threads, and two `try-replace` threads.

The "Try-Replace 0" thread in Figure 3.2 fails because it likely observed the value "-1", then while attempting to swap value, "Try-Replace 1" intervened and successfully replaced the value with "1". When "Try-Replace 0" resumed, it observed the value "1" and immediately returned false without modifying the `double_ref_counter` (though it had created a new `loud_object` before failing). This indicates that `try_replace` is working correctly.

3.4 Future Considerations

Though I have not had time to do so now, at some point in the future I would like to impose less restrictive memory orderings on the atomic operations in `double_ref_counter`. All atomic operations in `double_ref_counter` are currently sequentially consistent. In C++, sequentially consistent atomics impose a single total order of atomic operations upon which every thread agrees. This incurs a performance penalty, though it is less severe on some architectures like x86 and x86-64. Most importantly though, it makes the code much easier to reason about. Without sequentially consistent atomics, happens-before relationships in a single thread which we might take for granted may not be preserved when shared data is observed in multiple threads. Therefore, for simplicity's sake, I have kept all of the atomic variables sequentially consistent.

Despite this, I do believe that some of my atomic operations could be relaxed somewhat. In some situations, my atomics may be able to use acquire-release semantics. In the future, I may revise the memory ordering constraints imposed by my current implementation. For more information on atomics which are not sequentially consistent, I recommend Herb Sutter's 2012 "*atomic<> Weapons*" talk. A link to both parts of the talk can be found on Sutter's website at the link below.

https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/