Research Paper

Jack Stillwell

Multi-threaded data structures

This paper represents a written record of various aspects of an investigation into data structures native to concurrent environments. In reflecting on this topic, various sidelights and paths both parabolic and divergent appear in the following text. The ideological underpinnings of this endeavor follow the apocryphal Hegelian triad, tied up with the meta-ontological moving-into-becoming of the possibility of new horizons at the epistemological limit of the always-already unknowable. Recall, now, that all structures are built upon a base. Any base has as its substance only a dialectical tension realized in material. Any superstructure relies upon the same dialectical contraction within its own composition, but the tension between it and the base allow for the superstructure to take on multitudinous forms. Yet, always the base defines the limit of the morphological potential.

All of this means that when you get new stuff to use to make things, you can probably make new things. Things that were impossible before. But people will always want to keep building the same things. So, frequently, we just keep building the same thing. So, I ended up building the same thing, but always thinking towards something new.

I produced a C++ implementation of the non-blocking queue from Michael and Scott’s Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms[[1]](#footnote-1).

The algorithm is simple. It is basically an atomic triple checking loop using compare and swap operations.

To Enqueue:

1. Atomically load the contents of the tail and retain a copy on the stack.
2. Compare the value saved to another atomic load of the tail.
3. Check to see if the tail you have is a “clean” tail, or if it is an errant tail with a “next.”
4. Use compare and swap to either
   1. If you have the actual tail, try to store the new value as the next of the tail
      1. If you did it, break the loop.
      2. Else continue
   2. Otherwise, try to put the real tail in the tail using CAS, then loop either way.
5. Once out of the loop, try to update the tail to point at the new tail.

We can see right away that we never lock or block or wait while we enqueue. We can also imagine a world in which a thread trying to enqueue an item would fail to ever enqueue successfully.

The reason this algorithm is not wait-free is that it is impossible to know how long it will take a given thread to enqueue an item, or even if the enqueue will necessarily terminate for any particular thread. However, whenever used to insert a finite number of items, even with an infinite number of concurrent operations, we can guarantee that the delayed thread will complete the operation. Infinite queues are not supported in this implementation, as we do not have a lazy dequeue.

The dequeue operation uses the same comparisons:

1. Atomically load the contents of the head and the tail and retain copies on the stack.
2. Compare the value saved to another atomic load of the head.
3. Check to see if the head is the tail.
   1. If so, and they are both empty, don’t return anything.
   2. Otherwise, use CAS to try to set the tail to be its own next, then loop
4. Copy the data to return, then
   1. use compare and swap to Delete the head, and return
   2. fail compare and swap and start over

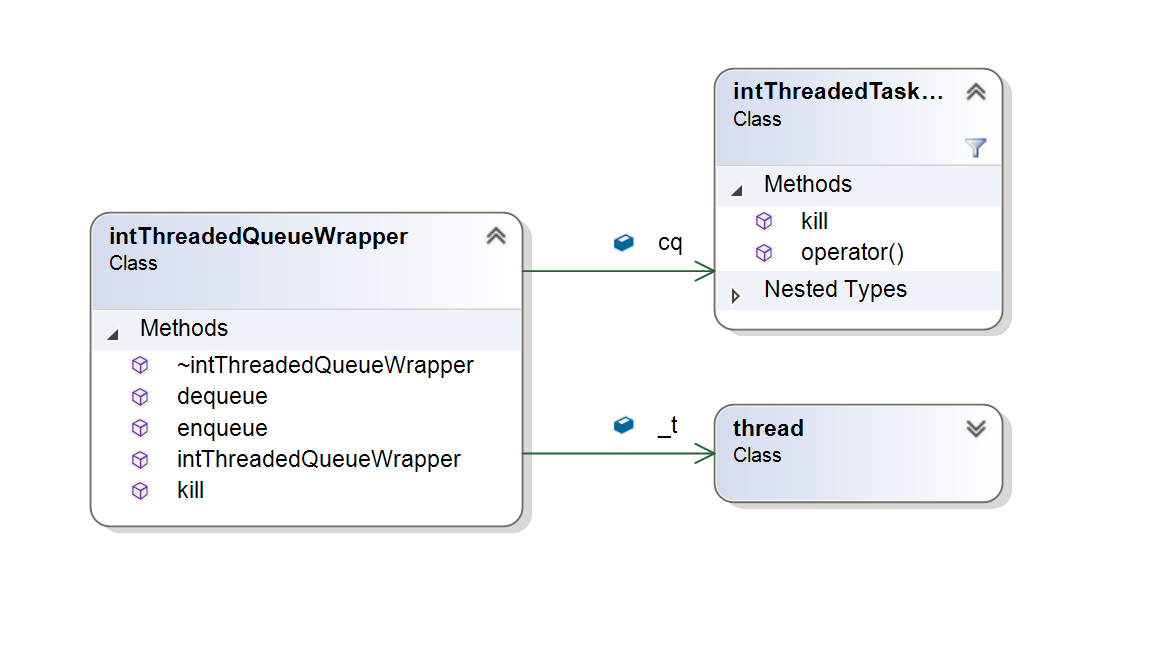
For additional implementation details, see the paper Scott’s Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms[[2]](#footnote-2).

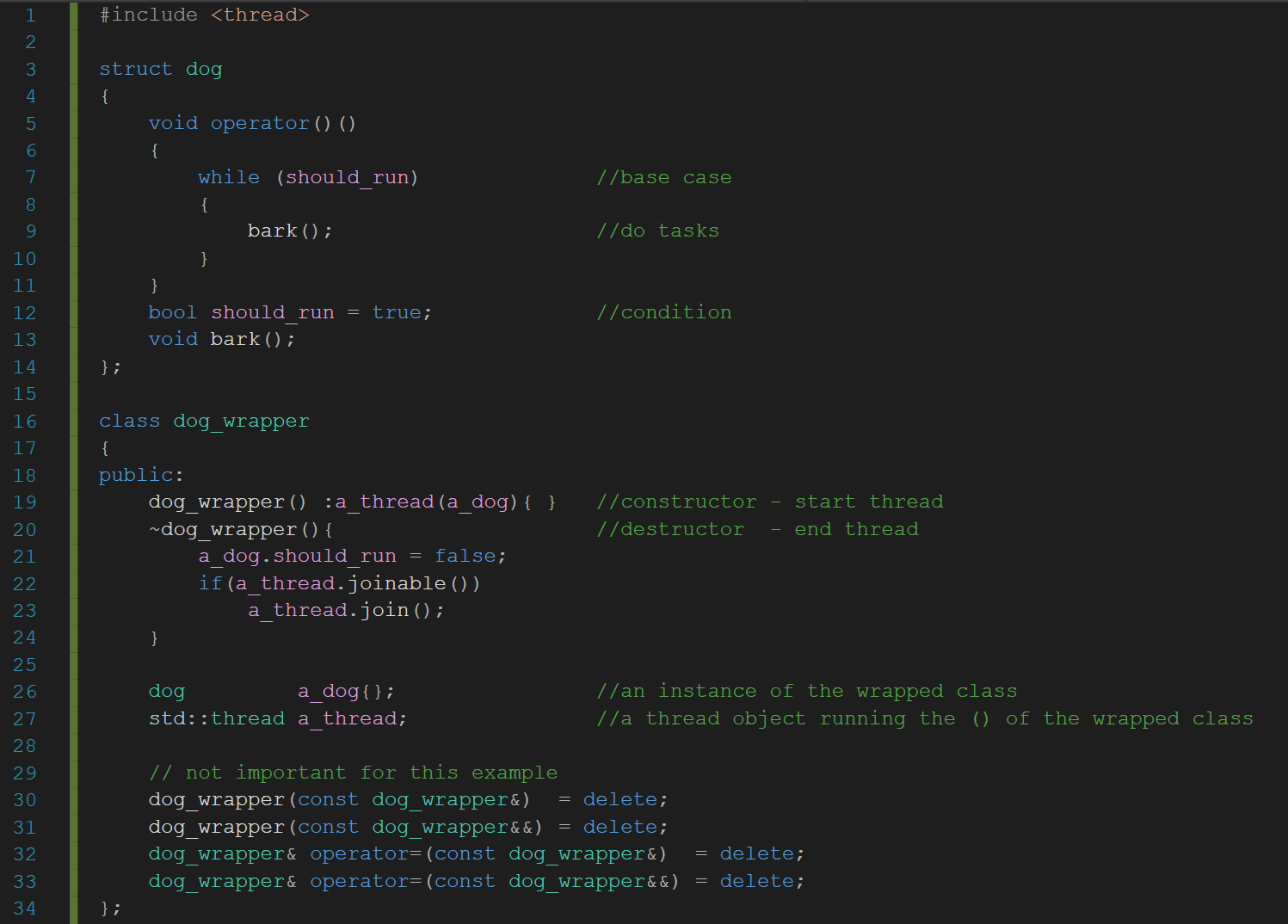
This method works because it constantly checks and atomically rechecks that the data is consistent, or else it starts over. Memory ordering and atomic checks prevent live-locking.

Using this queue, I created an infrastructure to guarantee[[3]](#footnote-3) the asynchronous, atomic processing of operations on given sets of data and in the queue. While the implementation of the queue does not leak, and the implementation of the taskqueue doesn’t leak, my tests leak all memory.

The Thread Wrapper

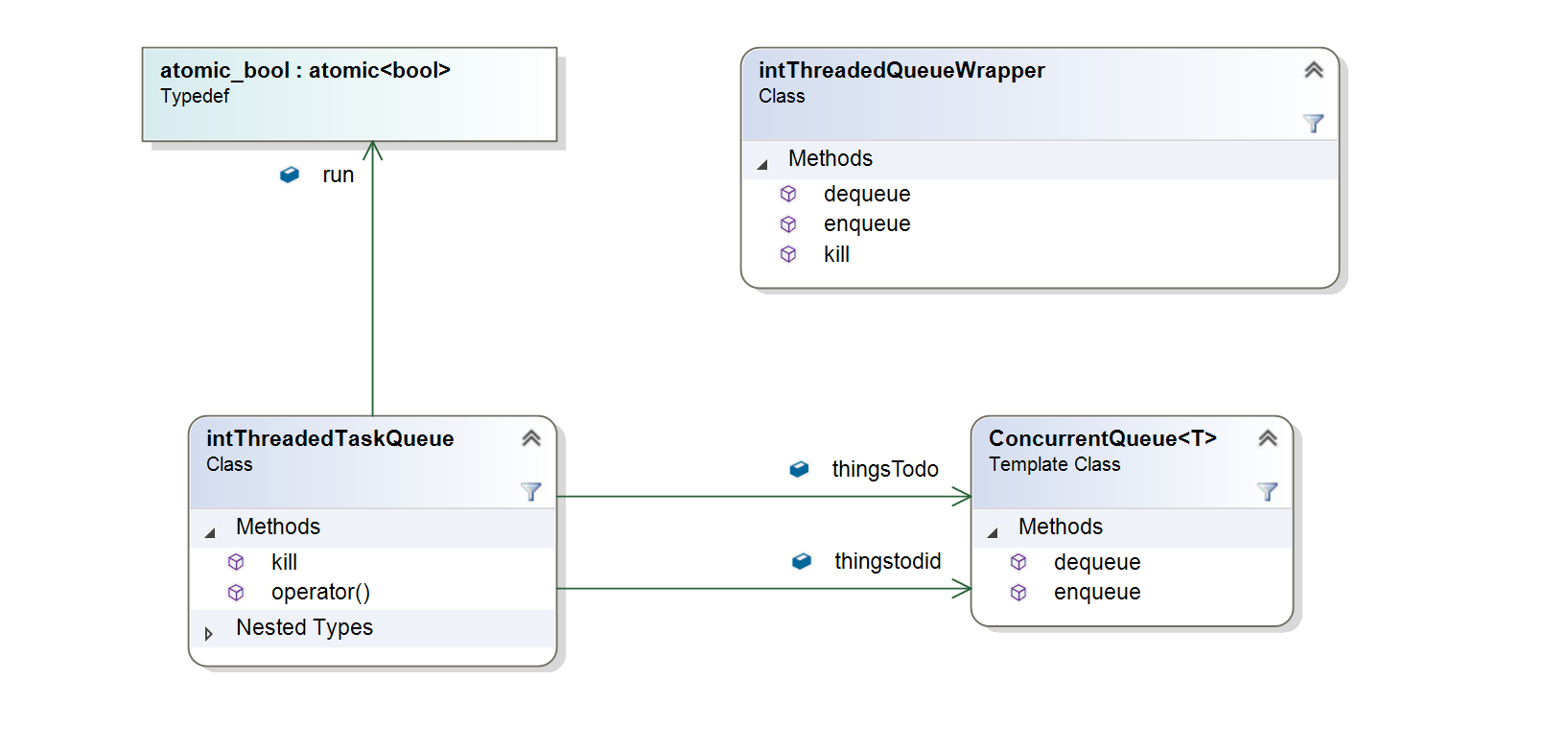
A basic way to manage threads is to create a thread wrapper class. The class will contain a thread object and an object of the type which we wish to thread. Using either the operator()() overload, or any other function, the function should loop on a terminal condition that can be signaled by the wrapper.





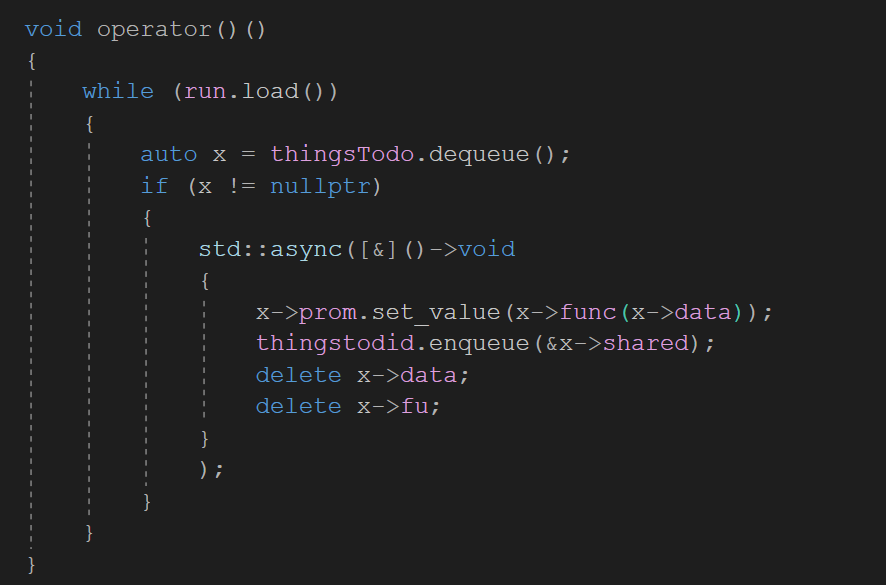
As we can see in the code example, the constructor of the wrapper starts the thread running with an object of the type of the wrapped class. A new thread now runs the dog object’s loop, continuously barking. If we want the dog to stop barking, we need to talk to it’s owner, the dog\_wrapper, in order to get the message to the dog.

In the instance of the intThreadWrapperQueue (wrapper), we have a wrapper for a less trivial class. The intThreadedTaskQueue(task queue) looks like this:



When the wrapper is told to enqueue something, it enqueues that item in the task queue’s queue of tasks to do. Dequeue retrieve completed work from the queue of std::shared\_futures representing completed or deferred tasks.

The task queue’s operator method looks like this:



So we have a spin lock on an atomic bool. That is the desired case because we only want this single thread to be able to direct the execution on the tasks that are passed into it from the wrapper. The std::async function’s default launch status is std::launch::async | std::launch::deferred. The binary or means that the decision on whether to do this task asynchronously or just be lazy about the execution is left up to the compiler.

1. Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96). ACM, New York, NY, USA, 267-275. DOI=http://dx.doi.org/10.1145/248052.248106 [↑](#footnote-ref-1)
2. Michael and Scott. *ibid*. [↑](#footnote-ref-2)
3. Barring the usual exceptions about casting and pointer math and the like. [↑](#footnote-ref-3)