Concurrency Final Project: Lock-free Queues in Rust

Arvind Raghavan and Matthew Pabst

May 9th, 2020

All of the code we've written for this assignment is available at https://www.github.com/PabstMatthew/rust-lockfree.

1 Background

As we've discussed at length in class, locks have plenty of issues. Beyond the arbitrariness of mutual exclusion as a synchronization primitive, locks are the source of many problems such as deadlock, priority inversion, preemption tolerance, signal safety, and poor performance in general. While other blocking synchronization primitives can improve programmability, they often have the same functional and performance issues of locks.

Lock-free algorithms are an alternative solution that provide concurrent access to shared data without blocking or exclusivity. Unlike blocking algorithms, which can allow a slow or delayed process to prevent faster processes from completing work indefinitely, lock-free algorithms guarantee that if there are one or more threads trying to perform operations, at least one will finish in a finite number of steps. Thus, non-blocking algorithms are much more robust to sudden, non-deterministic slowdown, such as from thread preemption or cache misses. However, because blocking mutual exclusion primitives are not used, it becomes much more difficult to reason about correctness. Thus, lock-free algorithms have been limited to data structure libraries, in which the extra time and attention pays dividends in terms of performance.

We present two lock-free queue implementations, one that leverages Crossbeam's epoch-based garbage collection library and one that forgoes garbage collection altogether. While the latter leaks memory, we believe it is valuable to compare to other implementations to highlight the cost of garbage collection. The source code for our implementations can be found at https://www.github.com/PabstMatthew/rust-lockfree. We compare the run-time and peak memory usage of both of our implementations with `Crossbeam`'s SegQueue (a mostly lock-free implementation with some bounded spinning and segmented, amortized memory allocation), the Rust `lockfree` library's lock-free queue, which uses reference counting, and standard blocking queue implementations on a diverse set of workloads and present the results.

2 Methodology

Results were collected by running single trials on the UTCS host `aries.cs.utexas.edu`, which has an x86-64 Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor with 4 physical cores and 8 hyper-threaded cores. The host's OS version is 4.15.0-88-generic 88-Ubuntu. For each configuration, 6 samples were taken individually and averaged, and two outliers were discarded to prevent OS-level caching from affecting the results. Because there is no serial baseline to compare against, we present results as runtime in milliseconds.

3 Implementation

Initially, we implemented a Michael-Scott queue without any memory reclamation, which uses a linked list as the underlying data structure. The `push` algorithm involves attempting to load the tail, check its `next` pointer, and CAS the tail's next pointer with the new node. At each step of the algorithm, the data structure must be checked to ensure that no thread has changed its state during the method. While this might seem

to imply that threads could get stuck looping forever, the key insight is that the actual state change happens in a single observable CAS, meaning that the only way a thread aborts is if another thread makes progress. The `pop` algorithm is similar, attempting to load the head and its `next` pointer, then CAS'ing the head's `next` pointer, retrying any time the head changes underneath the current thread.

Using pseudocode from online resources (linked in Section 6), implementing the queue without memory management was relatively simple. However, implementing memory reclamation proved to be the most challenging aspect of implementing the lock-free queue.

3.1 Challenges

The first challenges that we ran into with our implementation was how to communicate the safety of our algorithm to the Rust compiler. Unsurprisingly, the Rust compiler does not appreciate dereferencing raw pointers, or moving data out from under raw pointers. Looking at different lock-free data structure implementations online helped us understand how to perform unsafe actions on the queue internals. Specifically, we discovered that the use of the `UnsafeCell` wrapper allowed us to use potentially unsafe operations on the data in the queue, while the `MaybeUninit` wrapper allowed us to create uninitialized fields in Rust structures.

The second challenge that we encountered was how to add memory reclamation to our implementation. Initially, we thought using reference counts would be the simplest solution, however since double-word CAS is not available or in any way idiomatic in Rust, we realized that this approach would be impossible. If a thread cannot atomically load a pointer and update its reference count, then race conditions will inevitably crop up in which one thread frees some data while another is holding the pointer, allowing for use-after-free bugs. After coming to this conclusion, we looked to `Crossbeam`'s queue implementation for inspiration on how to solve the memory reclamation problem. To our surprise, we ended up realizing that this library queue was essentially using a to protect a small section of code, enabling them to dodge this issue. After doing some digging, we found a blog post by the library's main contributor explaining that the hybrid of lock-free and spinning code (combined with exponential backoff) improved performance in many benchmarks. Well, what other options did we have?

The next solution we looked into was hazard pointers, an idea published by Alexandrescu and Michael to solve the memory reclamation issue associated with lock-free programming. Many C++ implementations online used this approach so we thought it might be a feasible approach. However, hazard pointers are implemented by having variables tied to each thread, which are readable by other threads, so that threads only free data when no other thread currently holds the pointer. This approach would require significantly modifying Rust internals, so we decided against it.

3.2 Epoch-based Memory Reclamation

Luckily, the `Crossbeam` library provides a utility called epoch-based reclamation first described by Keir Fraser in his PhD thesis. The general idea is to have three epochs and record discarded but not yet freed objects in a list for each epoch. Threads can be running in the current or previous epoch, so the previous previous epoch contains objects that are safe to free. `Crossbeam` implements this idea succinctly using wrappers to describe the ownership properties of pointers in the data structure. The library essentially requires explicit declaration of transferring ownership into the data structure, explicit marking of shared pointers, and explicit declaration of when an object is ready to be discarded. In Section 4, we compare our memory-leaking implementation with our epoch-based reclamation implementation to illustrate the overheads of memory reclamation in lock-free code.

4 Results

We built a set of four diverse benchmarks in order to measure different aspects of each implementation. Below is a description of each benchmark and an analysis of the results.

1. **Pop Heavy**

   The Pop Heavy benchmark starts by serially initializing each queue with integers from 0 to 2^{20}. It then spawns n worker threads which, while there is data left in the queue, read an integer i from the

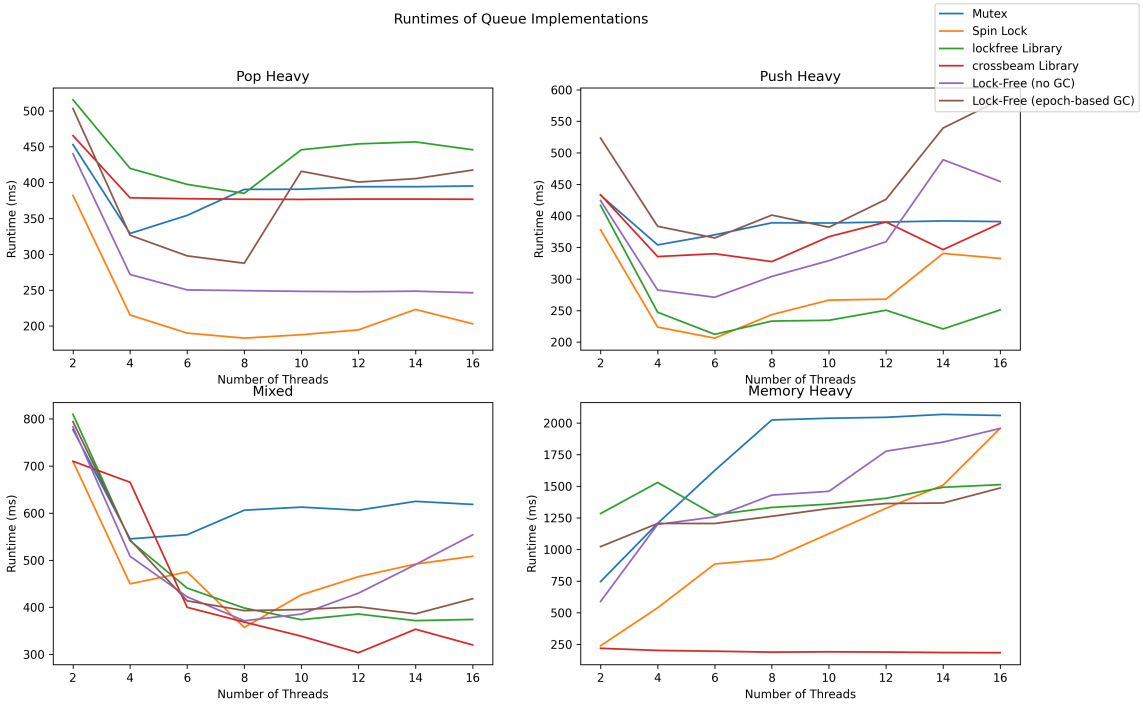

Figure 1: Runtime of queue implementations on multiple workloads

queue and calculate whether it is a prime by iterating through all numbers up to $\sqrt{i}$. The run-time for each algorithm can be seen in the "Pop Heavy" quadrant of Figure 1.

Surprisingly, the spinlock outperforms all other implementations, even with 16 cores. We believe this is because there is low contention for this workload. Even with 16 cores, the majority of time is spent doing the CPU-intensive primes calculation. In addition, the only concurrent calls appear in the form of calls to pop, which are short and have a small critical section. Thus, the spinlock is able to take advantage of these conditions to minimize the time spent waiting for the lock.

Our implementation without garbage collection is the second fastest. This makes sense because the workload doesn't allocate any new memory after the serial portion. Thus, the cost of memory management that the other implementations pay does not give any performance benefits.

Our implementation with epoch-based garbage collection outperforms the `Crossbeam` library for a smaller number of threads, but then becomes worse as the number of threads increases. It makes sense that the compare and swap operations can become pathologically worse for our implementation as contention increases, but we are not sure how the `Crossbeam` implementation is able to avoid this behavior.

Finally, the `lockfree` library seems to perform the worst. A brief look at the source code shows that the while the push implementation is wait-free, the pop implementation has much more overhead. We believe that the `lockfree` implementation is optimizing for the push operation and is thus slow for this pop heavy workload.

2. **Push Heavy**

   The Push Heavy benchmark initializes an empty queue, then spawns n worker threads which each iterate through $\frac{2^{20}}{n}$ integers, calculate whether the integers are primes, but then push the numbers to the list regardless. The primes calculation is done to ensure that the workload is not entirely memory latency bound and thus has some potential benefits from concurrency. This workload tests how different implementations handle contention between pushes.

Surprisingly, the `lockfree` implementation performs the best here! As noted previously, the `lockfree` implementation is able to implement `push` in a wait-free manner by moving more complexity into the pop operation. Thus, it excels in this push-heavy workload. Other than that, the trends tend to follow that of the pop-heavy workload. The spinlock performs the best for low numbers of threads, and then becomes pathologically worse. Our implementation without garbage collection is faster than `Crossbeam`, Mutex, and our implementation with epoch-based garbage collection. Interestingly, while in the pop heavy workload our implementation with garbage collection was able to outperform `Crossbeam` for lower numbers of threads, in this workload it is outperformed by `crossbeam` unilaterally.

3. **Mixed Workload** The mixed workload spawns $n/2$ threads which push 2^{20} integers into the queue and $n/2$ threads which read integers from the queue and calculate whether or not they are prime. This workload measures how contention between pushes and pops affects performance.

   It appears that `Crossbeam` is able to perform the best, with the rest coming close behind and the Mutex performing the worse by far. It is interesting to note the the `lockfree` implementation performs well in this mixed workload, though this may be because the contention is biased more towards pushes, as numbers get immediately pushed by $n/2$ threads while the other $n/2$ have to do some prime calculation before they can read. It's also interesting to note that our version with garbage collection starts to outperform the version without garbage collection for high numbers of threads. We believe this may occur because, with higher numbers of threads, there may be more overlap between pushes and pops, which might allow the garbage collector to reuse allocations (explained in more detail in the following section).

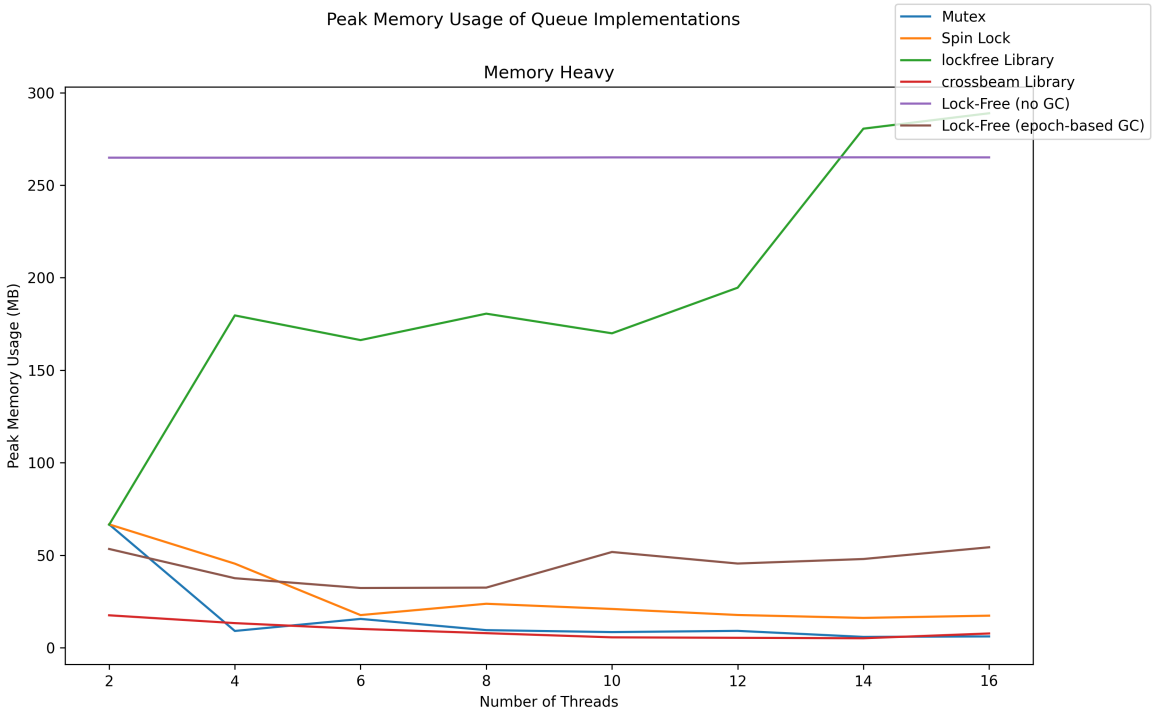

Figure 2: Peak memory usage of queue implementations on Memory Heavy workload

4. **Memory Heavy**

   The Memory Heavy benchmark aims to stress the garbage collection mechanisms of each queue implementation. It does so by spawning $n/2$ "push" threads and $n/2$ "pop" threads, similar to the mixed workload, but instead of computing primes, the "push" and "pop" threads just repeatedly push and

pop integers until 2^{22} have passed through the queue. We hand-tuned this number of integers to have a reasonable workload run-time (around 30 seconds for each run).

The run-time results for the memory heavy workload are shown in Figure 1. The first thing to note is that the the run-times actually increase as we increase the number of threads. Because we are not doing any CPU-intensive work (like the prime computation we do in the other workloads), this workload is memory latency bound. Thus, adding more threads actually slows down performance, as contention increases and there are minimal concurrency gains to begin with.

The fastest implementation is `Crossbeam`'s SegQueue, which makes sense because the SeqQueue uses amortized, segmented allocation aimed specifically at decreasing the overhead of memory allocation. Our epoch-based lock-free implementation is able to outperform the `lockfree` Rust library, which uses reference counting. This also makes sense for workloads that stress memory, as an epoch-based approach should be able to amortize the cost of memory management, instead of having to deal with reference counters for each push and pop. Finally, the spinlock implementation, while performing well with small numbers of threads, starts to perform pathologically worse as the number of threads increases, as expected.

Interestingly, our epoch-based lock-free queue actually outperforms our lock-free queue without garbage collection. We're not exactly sure why this is, as it seems that an implementation that doesn't have to bother with garbage collection should be faster as long as we don't overflow the maximum amount of memory (which we don't come close to doing). We believe that repeatedly allocating small, unaligned chunks of memory may be triggering some pathological heap behavior. Because the other implementations free their linked list nodes, the heap allocator is able to reuse those perfectly sized memory blocks. Instead, our queue without garbage collection is allocating pages and pages of these nodes, which when combined with other allocations could fracture the heap and slow down operations.

We also measured the peak memory usage of each implementation for the memory heavy workload. The results can be seen in Figure 2. Our implementation without garbage collection used the most memory and was constant across all numbers of threads. Our epoch-based garbage collector also used a constant amount of memory, though it used more memory than the Mutex and spinlock implementations. This makes sense because our implementation allows memory to grow for each epoch before freeing it, whereas the other two free memory right away. In addition, the `crossbeam` implementation with segmented allocation had the lowest peak memory usage. Very surprisingly, the `lockfree` library implementation had pathological memory usage as the number of cores increased, and even used more memory than our memory-leaking implementation when run with 16 cores! Looking at the `lockfree` documentation, it seems that the queue is implemented with per-thread "garbage lists". The lists are deleted **only** when there is no contention on the list, and there is no way to force a flush of the list if the memory overhead is too high. Thus, when there are 16 cores and lots of contention, the `lockfree` implementation is not able to free any of its garbage lists, and the peak memory usage skyrockets.

5 Conclusion

We compared our lock-free queue implementations with library-based lock-free queues and some standard blocking queues. While it is clear that there are performance gains available from building lock-free data structures, it is also clear that realizing those performance gains, especially at scale, can come at a cost. The Rust `lockfree` implementation is a great example of this; it achieves significant performance gains for some workloads, but also exhibits pathological peak memory usage for memory intensive workloads. We believe that the `crossbeam` SegQueue, while not entirely lock-free, shows that trade-offs, including segmented memory allocation and some bounded spinning, may be necessary to achieve performance gains for real-world applications.

6 References

Here are some helpful websites and resources that we referenced during the project:

- The Rust documentation contains plenty of helpful examples that I used as a guide throughout the project.
- This blog post by Christian Hergert provides an overview and example C++ implementation of a Michael-Scott lockfree queue.
- This blog post by Stjepan Glavina gives an overview of lockfree programming in the crossbeam library.
- The Lockfree library which provides a counter-based lock-free queue implementation.
- The `crossbeam` library which provides several tools for concurrent programming, including a library that facilitates epoch-based garbage collection.
- This blog post by Aaron Turon provides a tutorial on crossbeam's epoch-based memory reclamation, although it is slightly outdated.