A Wait-Free Queue with Wait-Free Memory Reclamation

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Abstract

Queues are a widely deployed data structure. They are used extensively in many multi-threaded applications, or as a communication mechanism between threads or processes. We propose a new linearizable multi-producer-multi-consumer queue we named Turn queue, with wait-free progress bounded by the number of threads, and with wait-free bounded memory reclamation. Its main characteristics are: a simple algorithm that does no memory allocation apart from creating the node that is placed in the queue, a new wait-free consensus algorithm using only the atomic instruction compare-and-swap (CAS), and is easy to plugin with other algorithms for either enqueue or dequeue methods.

Keywords wait-free non-blocking queue, low latency

1. Wait-Free Queue

When it comes to fairness and low latency, the most desirable of the progress conditions is wait-freedom. Wait-free queue algorithms are rare when both enqueue and dequeue operations can be executed by multiple threads (MPMC).

An essential part of any singly-list based queue is the nodes that compose it. The `Node` of Turn queue contains the pointer to the item, the pointer to the next node that follows in the list, and two extra variables, `enqTid` indicates which thread is requesting the enqueue, and `deqTid` indicates which thread will have the node for its dequeue request.

Two arrays are also required, `enqueuers` and `dequeuers`, with size `maxThreads`. In `enqueuers`, the position assigned to a specific thread, will be the index placed in the node's `enqTid`. `enqueuers` is an array of atomic pointers to nodes to be inserted in the queue, where `nullptr` means no active request. The `dequeuers` is in reality two arrays of nodes named `deqself` and `deqhelp`, when equal it represents an active dequeue request.

PPoPP '17 February 04-08, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-4493-7/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3018743.3019022

Algorithm 1 enqueue(T* item, int tid)

```
1  Node* myNode = new Node(item,tid);
2  enqueuers[tid].store(myNode);
3  for (int i = 0; i < maxThreads; i++) {
4    if (enqueuers[tid].load() == nullptr) return;
5    Node* ltail = hp.protectPtr(kHpTail, tail.load(), tid);
6    if (ltail != tail.load()) continue;
7    if (enqueuers[ltail->enqTid].load() == ltail) {
8      Node* tmp = ltail;
9      enqueuers[ltail->enqTid].compare_exchange_strong(tmp, nullptr);
10   }
11   for (int j = 1; j < maxThreads+1; j++) {
12     Node* nToHelp = enqueuers[(j + ltail->enqTid)%maxThreads].load();
13     if (nToHelp == nullptr) continue;
14     Node* nullnode = nullptr;
15     ltail->next.compare_exchange_strong(nullnode, nToHelp);
16     break;
17   }
18   Node* lnext = ltail->next.load();
19   if (lnext != nullptr) tail.compare_exchange_strong(ltail, lnext);
20 }
21 enqueuers[tid].store(nullptr, std::memory_order_release);
```

At the basis of the Turn queue is a singly-linked list, where enqueing consists of one CAS on the tail's `next` variable and another CAS to advance `tail`, just like on the MS queue (Michael and Scott 1996). Our consensus algorithm is based on the consensus protocol of the Turn starvation-free mutual exclusion lock (Correia and Ramalhete 2015), where each thread publishes its intent, in this case, an intent to enqueue a node, and the decision of who is the next thread is based on who is the next request *to the right* of the current `turn`. The current turn is the `enqTid` of the queue's tail.

The intent to enqueue is signaled with a non-null store of the node's pointer to be enqueued in `enqueuers`. The first non-null entry in the `enqueuers` array that is *to the right* of the current `turn` is the one that all threads will attempt to help next to enqueue, executing a CAS on `Node.next`, similarly to the MS lock-free enqueue algorithm. Each thread calling `enqueue()` will continue helping the next enqueue request until its own request has been completed.

The mechanism that insures a wait-free bounded enqueue relies on the `enqTid` variable of the node referenced by `tail`. This indicates to all threads that are executing an enqueue, that the last request that was satisfied, belonged to the thread that published its request in position `enqTid` of the `enqueuers` array. All threads realize that the thread's

Algorithm 2 dequeue() and helper functions

```
1  T* dequeue(const int tid) {
2    Node* prReq = deqself[tid].load(); Node* myReq = deqhelp[tid].load();
3    deqself[tid].store(myReq);
4    for (int i=0; i < maxThreads; i++) {
5      if (deqhelp[tid].load() != myReq) break; // No need for HP
6      Node* lhead = hp.protectPtr(kHpHead, head.load(), tid);
7      if (lhead != head.load()) continue;
8      if (lhead == tail.load()) {
9        deqself[tid].store(prReq); // Rollback request to dequeue
10       giveUp(myReq, tid);
11       if (deqhelp[tid].load() != myReq) { deqself[tid].store(); break; }
12       return nullptr;
13     }
14     Node* lnext = hp.protectPtr(kHpNext, lhead->next.load(), tid);
15     if (lhead != head.load()) continue;
16     if (searchNext(lhead, lnext) != NOIDX) casDeqAndHead(lhead, lnext, tid);
17   }
18   Node* myNode = deqhelp[tid].load();
19   Node* lhead = hp.protectPtr(kHpHead, head.load(), tid);
20   if (lhead == head.load() && myNode == lhead->next.load())
21     head.compare_exchange_strong(lhead, myNode);
22   hp.retire(prReq, tid); return myNode->item;
23 }
24
25 void giveUp(Node* myReq, const int tid) {
26   Node* lhead = head.load();
27   if (deqhelp[tid].load() != myReq || lhead == tail.load()) return;
28   hp.protectPtr(kHpHead, lhead);
29   if (lhead != head.load()) return;
30   Node* lnext = hp.protectPtr(kHpNext, lhead->next.load());
31   if (lhead != head.load()) return;
32   if (searchNext(lhead, lnext) == NOIDX) lnext->casDeqTid(NOIDX, tid);
33   casDeqAndHead(lhead, lnext, tid);
34 }
```

request on that position was the last to be executed so it will be the next request *to the right* to have its turn and get its request executed. When it reaches the end of the array, it will start processing requests at the start of the array. It can happen that not all threads actively calling `enqueue()` will agree on which is the node immediately to the right of the current turn. Regardless of which node gets enqueued, the new `turn` will be based on the node that was successfully inserted at the tail of the list, from then on, all previously published requests will be seen by current or new threads calling `enqueue()`. *At most* there will be `maxThreads`-1 other nodes inserted in front of a request to enqueue, making the enqueue wait-free bounded by the number of threads.

The `dequeue()` algorithm uses the same Turn consensus algorithm. The dequeue operation can have two possible outcomes, it can return an item from the queue, or a null pointer when the queue is empty. When the queue is not empty, the algorithm follows the same approach used in the enqueue, it uses the Turn consensus to determine which request is to be satisfied next. A thread i calling `dequeue()` will *open* a request by setting `deqself[i]` to point to the same node as `deqhelp[i]`. Following the Turn consensus protocol, when a thread j decides that it is thread's i turn, it will assign the next node's `deqTid` to i using a CAS, and if successful, from that moment on, the node belongs to thread i. All threads that see `node.deqTid` different from `NOIDX` will be able to help by doing CAS in `deqhelp[i]`, this will guarantee that the result is visible to thread i and at the same time, it will make `deqhelp[i]` different from `deqself[i]` which closes the request. Representing a logical dequeue request with nodes of the queue, avoids the overhead of allocating a dequeue request object, and subsequently tracking its lifetime.

In case of an empty queue, due to `head` pointing to the same node as `tail`, the dequeue request will have to be rollbacked. This procedure is executed by `giveUp()`.

To rollback the dequeue request, thread i will set `deqself[i]` to its previous value, which was stored in a local variable `prReq`. It can happen that during the rollback, another thread that saw the request of thread i opened, assigned it a node, and that node was present on the list between the read of `head` in line 6 and the read of `head` in line 26. In that case, the result is published in `deqhelp[i]`, and thread i can return the item, thus aborting the giveup. Unfortunately, it is still possible that some thread saw the request opened but has not yet published in `deqhelp[i]`. In this case it is necessary to guarantee that the first node of the queue is assigned to some thread. If there isn't any open dequeue request, thread i will assign the node to it self. Due to happens-before guarantees, once the head advances, all threads seeing the new head will subsequently see the rollback. From this moment on, if there wasn't already a node assigned to thread i, then it is guaranteed that there will not be a node assigned to thread i for this request. The `giveUp()` method will guarantee that it will only return a null pointer if and only if no other helping thread has assigned a node to `deqhelp[i]` and that the head has advanced after the request has been rollbacked. Therefore, all threads will see a closed request for thread i once they realize the head has advanced.

To avoid losing wait-free progress, we used Hazard Pointers (HP) for memory reclamation (Michael 2004). In HP, the protect operation can be wait-free if instead of a load-store-load loop on the pointer that is being read, we do a single load-store-load sequence where the second load validates that the `head` of the queue has not advanced. Otherwise, it will jump to the next iteration in the algorithm's loop.

We have presented the first linearizable memory-unbounded MPMC wait-free queue with wait-free memory reclamation.

References

A. Correia and P. Ramalhete. Mutual exclusion - two linear wait software solutions. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/cralgorithm-2015.pdf`, 2015.

M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.