Turning Krieger's MCS Lock into a Send Queue
or, a Case for Reusing Clever, Mostly Lock-Free Code in a Different Area

Benjamin Engel and **Marcus Völp**

Technische Universität Dresden
Operating-Systems Group

Nöthnitzer Strasse 46, Dresden, Germany
{voelp, engel}@os.inf.tu-dresden.de

Abstract

Lock- and wait-free data structures can be constructed in a generic way. However, when complex operations are involved, their practical use is rather limited due to high performance overheads and, in some settings, difficult to fulfil object lifecycles.

While working on a synchronous inter-processor communication (IPC) path for multicore systems, we stumbled over a clever piece of code that did fulfil most of the properties that this path requires for its send queue. Unfortunately, this piece of code was by no means a data-structure publication or somehow related to send queues. Reporting on our experience in translating Krieger's MCS-style reader-writer lock into a send queue for cross-processor IPC, we would like to make the point that sometimes, searching for code could end up in a valuable treasure chest even for largely different areas.

1 Introduction

Predictability, security and the ease to support application-tailored OS functionalities all speak for microkernels and microhypervisors as host operating systems for todays and future manycore systems. In particular, augmented virtualization enviroments as we find them in desktop, server and embedded systems benefit from the ability to co-host large legacy software stacks next to more sensitive code such as real-time subsystems [2, 3] and secure applications [4].

Synchronous inter-process communication (IPC) is one of the central mechanisms many of these kernels implement. Reasons favoring synchronous (i.e., unbuffered and blocking) IPC are:

1. asynchronous communication and coordination primitives can easily be built on top of synchronous IPC, however the reverse is not possible in this generality;
2. synchronous IPC is able to exploit preallocated memory buffers, thereby eliminating the need for memory allocation during the message transfer; and,
3. processor local synchronous IPC implementations can be as fast as a few hundred cycles [1], which makes them suitable for higher-level synchronization primitives.

While processor-local IPC not necessarily requires a send queue, it turns out that in cross-processor IPC paths senders have to be blocked at the receiver and processed in some arrival-dependent order. In other words, multiprocessor synchronous IPC needs a send queue to avoid unbounded starvation of senders waiting to rendevous with their receiver.

This paper reports on our experience implementing such a send queue and the surprising result we found: basically a wait-free MCS-style reader-writer lock by Krieger et al. [7] provides all the essential

[1] 288 cycles on an Intel Core i7 920 2.67 GHz [5].

functionality that we required. In the following, we summarize the key requirements of send queues, motivate our choice of a non-blocking implementation and highlight the difficulties of using non-blocking code in the kernel. Then we briefly outline the original reader-writer lock by Krieger and present our modifications to turn it into a send queue. We evaluate our results and relate them to others, before introducing our idea on a treasure chest of non-blocking building blocks.

2 Send Queue Requirements

The primary purpose of a send queue is to order incoming requests to prevent sender starvation. In processor-local IPC paths, the order of send operations is governed by the sequence threads are picked by the scheduler. This completely eliminating the need of a send queue. For example, facilitating time and priority inheritance, servers may receive time from the currently highest prioritized thread blocked on them. Using this time, they can therefore complete potentially pending requests to process the request of the time provider [6]. The crutial feature ensuring this ordering is the ability to *donate* time to other threads, an operation which is easily done locally but very hard to apply across processor boundaries. Therefore, for cross-processor IPC, the ordering of incoming requests needs another mechanism: a send queue.

When designing IPC primitives, multiple, partially contradicting targets need close attention, like performance and flexibility. Individual send and receive operations allow for a high degree of freedom, but add complexity. Threads may invoke multiple servers by sending multiple messages before entering a receive state or they may receive from another client before replying to a previous one. On the other hand, allowing only calls (i.e., atomic send and receive operations) to invoke servers and restricting servers to reply only to the last caller, possibly after calling other servers in the course of handling the caller's request, limits the flexibility of IPC but simplifies synchronization and is sufficient for most use cases. Processor-local versions of such a call-reply style IPC path can be very fast because they do not need any form of synchronization [5].

The operations required of a send queue are *enqueuing* to the tail of the list and *dequeuing* the head when replying. Depending on whether these operations are used to block threads only in the case of a contended server or in every IPC, enqueuing into an empty list and dequeuing the head must be fast. In addition, send queues should also support a dequeue operation from the middle of the list, allowing threads to cancel their not-yet-started IPC. Situations where callers have to abort an IPC include the deletion of a thread and timeouts (e.g., set to react on certain error situations).

One of the key benefits of a synchronous IPC path is the possibility to preallocate all message buffers to avoid memory allocation during the IPC. The immediate consequence for the send queue is that all operations have to operate on preallocated memory as well. In particular, if a call is finished, the caller's message buffer and all meta data used during the IPC must be ready for use in subsequent IPC calls. In particular for generic constructions of lock-free data structures, these properties are difficult to fulfil. Still non-blocking implementations have their benefits: they typically scale to higher CPU numbers than lock-based variants and, in our case, the effort and overhead required for a fair lock protecting the send queue is in the same order as the effort and overhead of a mostly lock-free send queue.

3 Krieger's MCS-style Reader Writer Lock

By swinging a single pointer to the tail of a list, MCS locks implicitly arrange lock acquiring threads in FIFO order. Either if a thread enqueues into an empty queue (i.e., $\mathsf{tail} = 0$ prior to the enqueue operation) or if the previous lock holder releases the lock by clearing a field the next thread in the list spins on, the thread at the head of the list becomes lock holder.

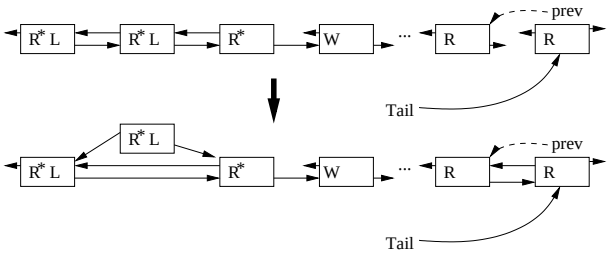

FIGURE 1: *Data structure and dequeue operation from Krieger's MCS lock*

Eliminating contention on the active reader counter of Mellor-Crummey and Scott's original reader-writer lock [8], Krieger et al. [7] introduce a second pointer to each queue element to enqueue readers into a lazily updated double-linked list. Finished readers dequeue from the middle of the list using spin locks that are only taken for the purpose of

protecting neighboring list nodes during a dequeue. The last active reader releases a subsequent writer if present. Figure 1 illustrates this algorithm and the used data structures.

The key features of Krieger's MCS lock, makeing it a perfect starting point for send queues, are:

1. an implicit FIFO ordering by threads atomically swinging the tail pointer to their list element and lazily enqueuing afterwards;
2. an extremely fast enqueue operation for the uncontended case (essentially just an atomic swap of the tail pointer);
3. in most situations, an extremely fast dequeue operation of the head element (essentially only the release of the next thread and an atomic compare-and-swap if the queue is empty afterwards); and,
4. the possibility to dequeue from the middle of the queue.

4 Turning Krieger's Lock into a Send Queue

Our goal is to translate Krieger's MCS lock into a mostly lock-free send queue for a call-reply style IPC path. In general, there are two principle ways to use such a send queue: enqueue callers only if the callee is contended, or, enqueue all callers that are sending or waiting to send to the callee. In the first case, exclusive control over the callee must be taken to determine whether it is ready to receive new requests. If not, the caller would block in the send queue while waiting for the callee to be ready to receive its request. Otherwise, the caller, in case of a caller-driven implementation, or the callee, in case of a callee-driven implementation, starts transferring the message. However, because locking the callee to obtain exclusive control ideally involves a fair lock that facilitates local spinning, we expect comparable costs for locking the callee and for enqueuing into an MCS-style send queue.

To avoid the additional overhead of locking the callee in the uncontended case (i.e., when the callee is already receiving), we maintain the following invariant:

Invariant I: *A callee with an empty send queue is always receiving and implicitly locked by the first thread entering the send queue.*

The first part of Invariant I is ensured automatically by the call-reply style IPC path as long as ongoing requests are not aborted (Section 4.1 below discusses how the spirit of this invariant can be maintained in case of aborts). The second part is ensured by leaving the callee locked in situations where the last thread dequeues itself from the send queue.

```
type SQ_Item = record
  next : ^sq_item
  pred : ^sq_item
  lock : precedence_spinlock
type Status = enum {EMPTY, NOT_EMPTY, OTHER}
type send_queue = class
  head : ^sq_item
  tail : ^sq_item

method enqueue(I : ^SQ_Item) : Status
    pred : ^SQ_Item := swap(tail, I)
    if pred = nil
      head := I
      release_precedence(I->lock)
      return EMPTY
    I->pred := pred
    // store fence
    pred->next := I
    release_clear_precedence(I->lock)
    return NOT_EMPTY
```

FIGURE 2: *Types and enqueue operation*

Figure 2 shows the pseudocode for enqueuing into the send queue[2]. Like in MCS locks, the function `enqueue` starts by atomically swinging the tail pointer of the send queue to the list element of the invoking caller and remembering the old value of tail. If this old value is `nil`, the queue was empty and the function may return after updating the head pointer. Otherwise, `enqueue` first sets the predecessor pointer of its element before completing the enqueue operation by updating the predecessor's next pointer. As pointed out by Krieger, this write sequence prevents a race with a concurrent dequeue operation from the middle of the list and may require an additional fence. The role of the precedence spinlock and the validity of head will be discussed after we have introduced the two dequeue functions.

Krieger's MCS lock does not distinguish between dequeuing from the middle of the list and dequeuing the head element, since finished readers have to retract from the queue lock no matter where they are. For a send queue however, the former operation is only invoked when a thread cancels its IPC, whereas dequeuing the head is used in every reply to a caller, thereby completing the IPC.

[2]For a better comparison, we adopt the pseudocode introduced by Mellor-Crummey and Scott [8], which is also used in Krieger et al. [7].

```
method dequeue_head(I : ^SQ_Item) : Status
  if acquire_precedence(I->lock) = false
    return OTHER
  if I->next = nil
    if !compare_swap(tail, I, nil)
      repeat while I->next = nil
  next : ^SQ_Item := I->next
  if next != nil
    next->pred := nil
    head := next
    I->next := nil
    return NOT_EMPTY
  I->next := nil
  return EMPTY
```

FIGURE 3: *Dequeue head operation*

Figure 3 shows the pseudocode for dequeueing the head element. Again we defer the discussion of the precedence spin lock to Section 4.1. Like in all MCS-style locks, a nil next pointer can indicate one of two situations: either the dequeue operation is about to remove the last thread from the list (in this case, tail = I holds) or, a thread is about to enqueue into the list but did not yet manage to update the predecessor's next pointer. The atomic compare-and-swap checks in which of the two states the list is in and clears tail in the first case. Otherwise, the dequeuing thread spins until the next pointer is set. This spinning is bounded because enqueue executes in the kernel with interrupts disabled. After returning from this loop, either the list is empty or the next pointer is set and the dequeuing thread can update the head pointer to the corresponding thread, now being the new head of the list.

Notice that the head pointer is not always current. In particular, dequeue_head does not update head in case the list gets empty. The invariant that maintains correctness of this implementation is the following:

Invariant II: *Head is valid only for the thread that is under exclusive control of the callee.*

An immediate consequence of this invariant is, that threads may not poll head until they reach the front of the send queue. The following sequence illustrates this race:

1. Thread A enqueues and dequeues from the list. Assuming the list was empty, head now refers to A because dequeue_head will not clear the head pointer, as doing so would race with with B's concurrent enqueue operation.

2. Thread B starts enqueuing itself but is delayed after updating the tail pointer.

3. Thread A returns, enqueues itself to the list and finds itself to be the head because B did not yet update this pointer after Step 1. If now a thread C enqueues itself and both A and B dequeue themselves using dequeue_head, A will set C's predecessor to nil but B's later dequeue will set the already left A as a new head.

Although the list structure itself is maintained, any head dependent action by C will now block forever or work on the wrong head A. For call-reply style IPC paths, the restriction Invariant II imposed on the use of the head pointer is no problem, because head is used only by the thread having exclusive control of the callee and in one of the following two situations: (1) to pull in the next message after a reply; and (2) to identify the thread to reply to. In a callee-driven implementation, the callee is active anyway when these situations occur. In a caller-driven implementation, the exclusive owner of the callee takes over control of the thread at the head of the send queue to push its message to the receiver.

4.1 Cancel

```
method
dequeue_middle(I : ^SQ_Item) : Status
    pred : ^SQ_Item = I->pred
    // chase and lock pred
    repeat while pred != nil
      if try_acquire(pred->lock)
        if pred = I->pred
          break
      release(pred->lock)
      pred := I->pred
    if pred
      acquire_precedence(I->lock)
      prev->next := nil
      if I->next = nil
        if !compare_swap(tail, I, prev)
          repeat while I->next = nil
      next : ^SQ_Item := I->next
      next->pred := pred
      if next != nil
        pred->next := next
      release(prev->lock)
      I->pred := nil
      I->next := nil
    return dequeue_head(I)
```

FIGURE 4: *Dequeue middle operation*

Figure 4 shows the pseudocode for dequeuing threads from the middle of the send queue, which is required for canceling waiting threads. Except for the precedence spinlocks, the above code directly resembles Krieger's reader unlock operation. The first while loop chases the predecessor pointer of the dequeuing caller's list element to lock it for the subsequent dequeue operation. In our code base, this loop is preemption reactive in the sense that it will abort

the dequeue operation if a preemption is pending. We have omitted this test for reasons of simplicity.

Having locked the predecessor, the own lock of the caller's list element is acquired to prevent concurrent dequeues from modifying the prev and next pointers. Like in all MCS-style locks, compare-and-swap is used to update the tail pointer in situations where the tail element of the list is dequeued. Otherwise, the dequeue operation waits for a pending enqueue operation to update the next pointer of the to-be-dequeued element. Together with enqueue and dequeue_head, dequeue_middle maintains the invariant that dequeued list elements are always precedence locked. In the following we describe the role of these precedence locks in greater detail.

The primary purpose of the spin lock is to protect the prev and next pointers from concurrent dequeues. In our version, we grant dequeue_head precedence over concurrent dequeue operations from the middle of the list, which are invoked as part of an IPC cancel operation and, unlike dequeue_head, are not performance critical. To grant precedence, threads dequeuing from the middle can obtain the lock only if it is free and the precedence bit is clear. Therefore, by setting the precedence bit in case the lock could not be acquired immediately, the callee replying to the caller will obtain the lock after at most the duration of one dequeue operation.

In a callee-driven implementation of synchronous IPC, there is one situation where two threads concurrently attempt to acquire the lock with precedence: when the reply of a callee to its caller collides with an IPC cancel operation to the head of the send queue. In this situation and because dequeuing from the send queue is the last operation of the IPC, it does not matter which one of the threads completes its dequeue operation. Therefore, a thread will bail out from the dequeue_head operation with the status code Other if it finds the lock with the precedence bit set. If these two operations happen in any sequence one after the other, it is important to abort the second operation because otherwise, after the compare-and-swap operation in dequeue_head, the second dequeue_head would wait for a thread to update its next pointer, which will never happen. In Krieger's MCS lock and likewise in our send queue, there is no means to detect just from the link information whether or not an element is enqueued. The invariant that dequeued elements are precedence locked introduces precisely this information to avoid the above lifelock in subsequent dequeue_head operations.

Because threads dequeuing from the middle may refer to elements that are no longer enqueued, threads and their corresponding send queue items may not be deallocated immediately upon their destruction. Instead, we reuse a read-copy update (RCU) like deferred destruction scheme [9], that was already available in Nova [5].

Although a callee is not necessarily receiving, the spirit of Invariant I holds trivially in a callee-driven implementation because a caller enqueuing into an empty list will simply activate the callee no matter in what state it is. After a cancel, this may result in the callee completing its prior operation or performing some cleanup before entering the receive state during which it will pull the caller's message.

4.2 Evaluation

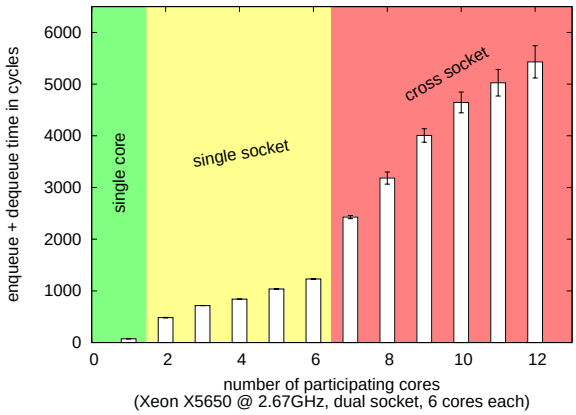

FIGURE 5: *Send queue overhead*

To evaluate the performance overhead of our send queue we have measured roundtrip times of the operations involved in a call-reply pair — enqueue followed by dequeue_head if the caller is at the head of the send queue. Measurements were done on an Intel Xeon X5650 @ 2.67 GHz by increasing the number of cores participating in the roundtrip. As seen in Figure 5, the call/reply to an uncontended server on a single core is quite fast, having multiple cores on the same socket competing on a queue adds a overhead of up to 1200 cycles, but having to access the cross-socket interconnect is really painful.

5 Related Work and Treasure Chest

In the literature, one finds several lock-free implementations of common data structures such as lists, stacks and trees [11, 12]. However, typically these

data-structures where designed as a reference implementation to illustrate some high-level concepts such as linearizability [1], wait-freeness [14] or obstruction freeness [13]. Others demonstrate the use of memory management schemes such as hazard pointers [10] to establish the type safe memory these data structures require. However, little work is published on the building blocks leading to lock-free data structures (Valois' work [11] forming an exception) and on non-pure algorithms that, like Krieger's MCS lock, are lock-free in the important operations but use potentially unfair locks when this unfairness does no harm. The consequences are that it is difficult to find an implementation that perfectly suits a given problem, it is hard to identify the building blocks used in such an implementation and even harder to perform the necessary adjustments.

More raising the problem than providing a definite solution, we propose to collect searchable building blocks for lock-free data structures. By building blocks we mean essential ways to introduce a certain functionality and the prerequisites and properties they imply. For example, dequeuing from the middle of a doule-linked list may be implemented using Valois' helper nodes [11], but with the limitation of not being able to reuse these nodes immediately or, in a mostly lock-free fashion, with Krieger's spin locks that are just used for the dequeue operation. RCU [9] is an excellent building block for read-most data structures and deferred object destruction, however send queues are write dominated. In our case, relaxing the validity of the head pointer allowed for a very simple implementation of the queue and the queue-state dependent implicit locking of the callee improves performance for the uncontended case.

6 Conclusions

This paper describes the modifications necessary to turn Krieger's MCS-style queue lock into a send queue for call-reply style synchronous IPC paths. To our surprise, Krieger's lock already provides all the essential functionality. Our send queue extends Kriegers lock in two invariant driven ways: by introducing a head pointer and by introducing an implicit locking scheme where threads enqueuing into an empty queue immediately get hold of the callee lock. This resulted in a fast send queue usable in call-reply style synchronous cross-processor IPC paths.

References

[1] *M. Herlihy and J. Wing*, "Linearizability: a correctness condition for concurrent objects," ACM Trans. on Programming Languages and Systems, vol. 12, no. 3, pp. 463492, 1990.

[2] *M. Roitzsch and H. Härtig*, "Ten Years of Research on L4-Based Real-Time" Proc. of the Eighth Real-Time Linux Workshop, Lanzhou, China, 2006

[3] *Guanghui Cheng, Nicholas Mc Guire, Qingguo Zhou and Lian Li*, "L4eRTL: Port of eRTL(PaRTiKle) to L4/Fiasco microkernel" 11th RT Linux Workshop, 2009

[4] *H. Härtig* "Security Architectures Revisited" Proc. of the 10th ACM SIGOPS European Workshop, France, Sept. 2002

[5] *U. Steinberg and B. Kauer* "NOVA: A Microhypervisor-Based Secure Virtualization Architecture" Proc. of EuroSys, April 2010

[6] *U. Steinberg, J. Wolter and H. Härtig* "Fast Component Interaction for Real-Time Systems" Proc. of the 17th Euromicro Conference on Real-Time Systems, July 2005

[7] *O. Krieger, M. Stumm, R. Unrau and J. Hanna* "A Fair Fast Scalable Reader-Writer Lock" Proc. of the IEEE International Conference on Parallel Processing, 1993

[8] *J. Mellor-Crummey and M. Scott* "Scalable reader-writer synchronization for shared-memory multiprocessors" 3rd ACM Symp. on Principles and Practice of Parallel Programming, 106-113, April 1991

[9] *P. McKenney* "Read-Copy Update" *http://www.rdrop.com/users/paulmck/RCU/*

[10] *M. Michael* "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects" IEEE Trans. on Parallel and Distributed Systems 15 (6): 491504, 2004

[11] *J. Valois* "Lock-free linked lists using compare-and-swap" 14th Annual ACM Symp. on Principles of Distributed Computing, 214-222, Aug 1995

[12] *K. Fraser* "Practical lock-freedom" PhD Thesis, University of Cambridge, Feb 2004

[13] *M. Herlihy, V. Luchangco and M. Moir* "Obstruction-Free Synchronization: Double-Ended Queues as an Example" 23rd Int. Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003

[14] *M. Herily* "Wait-free synchronization" ACM Trans. on Programming Languages and Systems, 13(1):124-149, Jan 1991