Quickly Reacquirable Locks[*]

Dave Dice Mark Moir
Sun Microsystem Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

William Scherer III
Computer Science Department
University of Rochester
Rochester, NY 14620

Abstract

We propose a new class of *quickly reacquirable* mutual exclusion locks (QRLs) that are optimized for the case in which a single process repeatedly acquires and releases the lock and in which no other process attempts to acquire the same lock. This case is very common in an important class of applications that includes such systems as Java^{TM} Virtual Machines.

When the first holder of a QRL first acquires the lock, it *biases* the lock to itself. This requires the process to execute a one-time `compare-and-swap` instruction. Thereafter, this *bias holder* process can reacquire and release the lock via a highly optimized *ultra fast path*. If a second process attempts to acquire a QRL, then the lock reverts to a "default" lock. We demonstrate generalized techniques that enable use of any standard mutual exclusion lock as the default lock. Finally, we discuss sample techniques to allow reinitialization of a QRL lock so that it can be rebiased. Such techniques are particularly valuable in the case of migratory data access patterns.

Although relaxed consistency models would typically require the use of expensive synchronization mechanisms such as memory barriers or read-modify-write instructions, we demonstrate a novel *collocation* technique that constrains reordering of loads and stores in the TSO (Total Store Order) memory model without their use. In our strongest result, we use this technique to create a highly optimized class of QRLs.

1 Introduction

1.1 Mutual Exclusion

The `mutual exclusion` problem has a long history. Beginning with Dijkstra's publication of Dekker's first correct mutual exclusion lock in 1965 [4], and continuing through to the present time, mutual exclusion locks have been the focus of hundreds if not thousands of research papers and other technical writings. Because the problem is so pervasive, it ranks among the most well-known in all of computer science.

The mutual exclusion problem arises in a domain wherein each participating process executes, in strict cyclic order, program regions labeled *remainder*, *acquire*, *critical section*, and *release*. A solution to the mutual exclusion problem consists of code for the `acquire()` and `release()` operations. These operations are required to guarantee that once a process successfully completes an `acquire()` operation, no other process will complete an `acquire()` operation before the first process invokes a `release()` operation. Solutions to the mutual exclusion problem are often referred to as locks.

1.2 Performance Characterization of Locks

An important advance in the study of lock performance is due to Lamport [6]. In his fast mutual exclusion algorithm, a *fast path* guarantees that the `lock` and `unlock` operations complete in time independent of the number of *potential* contending processes in the important case where a single process contends for the lock in isolation. Locks that share this property are called *fast locks*, and most important locks used in production software are fast locks.

[*]Contact email: `Dave.Dice@sun.com` or `Mark.Moir@sun.com`

Within the category of fast locks, we have focused on identifying particularly important sub-cases as well as on how to optimize for them. One sub case that we believe to be particularly important in systems such as Java^{TM} Virtual Machines is the case wherein a single process repeatedly acquires and releases a lock, but other processes rarely if ever access this lock. Such cases arise because the overall historical expense due to the overhead of locking has been such that system designers have gone to extensive lengths to avoid synchronization wherever possible. (Locks in these cases are present in order to provide safety in the case where another process needs to share access to the data; however, these cases are very rare by design. Such locks are also useful for minimizing synchronization overhead in code that needs to run in both single-threaded and multi-threaded configurations.) It is for this very case that our proposed class of locks are optimized. We shall refer to locks optimized for this case henceforth as *quickly reacquirable locks* (QRLs). We shall also refer to the optimized code path in which a process reacquires and subsequently releases a lock that it has previously held, and that no other process has ever held, as an *ultra fast path*.

Having identified QRLs as a sub-case of interest, we now proceed to develop techniques for optimizing them. In order to do this, however, we must first decide upon what constitutes an optimized reacquirable lock. Generally speaking, QRLs are optimized by reducing the type and number of instructions that are required in the ultra fast path. In particular, we focus on the use of atomic read-modify-write instructions in the ultra fast path, because these instructions are typically much more expensive than other instructions.

2 Overview

2.1 Atomic Read-Modify-Write Instructions

In this section we describe several atomic read-modify-write instructions that are commonly used in locks, and that we wish to avoid in the ultra fast path of QRLs. In the remainder of this paper, we will adopt the industry convention of referring to these particular instructions as *atomic* instructions.

The first instruction, `compare-and-swap` (hereafter, `CAS`), atomically checks to see whether the value in the memory location referenced by a pointer is some given value and if so, replaces it with another given value. Pseudocode for the `CAS` instruction is as follows:

```
Boolean CAS(int *ptr, int old, int new) {
    atomically {
        if (*ptr == old) {
            *ptr = new;
            return true;
        } else {
            return false;
        }
    }
}
```

Another common instruction, `SWAP`, atomically substitutes a value for the contents of the memory location referenced by a pointer and returns the old contents of that location. Pseudocode follows:

```
int SWAP(int *ptr, int new) {
    atomically {
        int result = *ptr;
        *ptr = new;
        return result;
    }
}
```

One final instruction bears mention. The memory barrier (hereafter, `membar`) constrains the processor from performing certain types of instruction reorderings. In particular, `membar`s are typically used in the Total Store Order (TSO) memory model [8] to prevent read operations from being reordered before write operations. These instructions are expensive because their implementation typically consists of draining a processor's reorder buffer, thereby limiting the pipelining of instructions that the processor can perform. For this reason, we wish to avoid `membar` instructions in the ultra fast paths of QRLs.

2.2 Categorization of Previous Locks

Most existing well-known locks contain atomic instructions in both the `acquire()` and `release()` fast path operations. The MCS lock [7] is a typical example in that, even absent contention, it uses a `SWAP` instruction in the acquire path and a `CAS` in the release path. Another well-known lock, TATAS (the so-called test and test and set lock) uses a test-and-set or `CAS` atomic instruction to acquire the lock, though releasing the lock is a simple write. Other locks optimized specifically for use inside Java^{TM} Virtual Machines include Metalock [1] (`CAS` in both acquire and release), Thin locks [2] (`CAS` in acquire), and Relaxed-locks [3] (`CAS` in acquire). For QRLs, our target is to avoid all such instructions in both the acquire and release portions of the ultra fast code path.

3 Construction of Atomic-Free QRLs

In this section, we first present an overview and detailed description of our preferred QRL embodiment. Later, we discuss several potential design alternatives. This latter set is not meant to be an exclusive list of all possible QRL implementations; rather, it provides an indication of the wide applicability of this idea.

3.1 Generic QRL Implementation

Our atomic-free QRLs consist of two additional fields added to some default lock (which can be any standard lock). The first field is a status word that takes on one of the values: {`NEUTRAL, BIASED, REVOKING, DEFAULT`}. Initially the lock is `NEUTRAL` and the first time the lock is acquired, the process acquiring the lock changes the status to `BIASED`. If a second process attempts to acquire the lock, it eventually changes the lock status to `DEFAULT`, but may set the status to `REVOKING` as an intermediary state in the revocation protocol. When the lock state is `BIASED`, the first field additionally contains a process identifier for the current bias owner. In a non-rebiasable QRL, the state can only move forward, in order, through these values. We later discuss QRLs that can be rebiased.

The second field is a Boolean bit that indicates whether the bias holder currently holds the lock. Hence, when the lock has been acquired, and not subsequently revoked, the bias holder can acquire or release the lock by simply toggling this bit. We adopt the notation that a process that has set this bit has *acquired the QRL via the quicklock*. A process can acquire the QRL via either the quicklock or via the default lock; our constructions ensure that both "sub-locks" cannot be concurrently held.

Switching a lock from `NEUTRAL` to being biased to a particular process can be done for the one-time cost of a single `CAS` operation. The `CAS` (or some equivalent mechanism) is necessary in order to prevent a race condition in which multiple processes simultaneously attempt to acquire a previously `NEUTRAL` lock.

The main difficulty in constructing an atomic-free QRL lies in coordinating the revocation of a lock between a revoking process and the bias holder process (that might be anywhere in the acquire-release cycle). Race conditions that must typically be addressed include a revocation that occurs simultaneous to a biased acquire as well as a revocation that occurs simultaneous to a biased release. Yet another race condition occurs when multiple processes attempt to revoke the lock simultaneously; however, this last race can be avoided by using `CAS` to change the lock status word to the intermediate `REVOKING` state: the process for which the `CAS` succeeds is the "true revoker" and any other processes revert to acquiring the default lock.

3.2 Atomic-Free QRL Implementation Overview

Our goal is to avoid the use of expensive synchronization instructions such as `membar`s and atomic instructions. In a sequentially consistent multiprocessor [5], it is straightforward to design a technique in which a bias holder reacquires the lock by first writing its quicklock bit and then verifying that the lock has not been revoked; and revoking processes first indicate revocation and then check whether the bias holder has acquired the quicklock. This is attractive because the ultra fast path consists only of load and store instructions. Unfortunately, however, such techniques are not typically correct in multiprocessors with memory models that are weaker than sequential consistency, and therefore expensive `membar` instructions are needed to ensure correctness.

For our purposes, we do not require the full effects of a `membar` instruction in the biased acquire and release code paths. Rather, it is sufficient to preclude a single read (of the lock status) from being reordered before a single write (of the quicklock). This in turn allows us to consider techniques that exploit subtleties of the TSO memory model. In particular, TSO requires that instructions in the same thread appear to be executed in program order. Hence, if we can introduce an artificial dependency between the write and read instructions mentioned above, we can ensure that they are not adversely reordered.

Our scheme for introducing this dependency is to collocate the status field and the quicklock field into the two halves of a 32-bit word. Then, we perform the following steps in order to reacquire the lock on the ultra fast path:

1. perform a half-word store on the quicklock field
2. load the ENTIRE 32 bit word
3. shift out the status field from what we just read

Now, because the data read in step 2 includes the data that was written in step 1, the read must be ordered after the write. Locks based on this technique, including both a generalized implementation and examples that use MCS and TATAS as the default lock may be found in Appendix A. (The lines marked with triple asterisks in the generic QRL lock are replaced with whatever code is appropriate to the specific default lock in question.) This approach provides portability to any memory model in which apparent program order is guaranteed for instructions executed on the same processor; further, it requires no operating system intervention. We note that some care must be used to ensure that a compiler does not optimize steps S2 and S3 into a half-word read, as this would once again allow the read to be reordered.

This collocation technique, for precluding the reordering of a single read, is likely to be useful in many more situations than just this lock. In general, schemes of this form can potentially be used in any situation where a memory barrier is used to prevent one or more reads from being reordered ahead of a particular write.

3.3 Detailed Description of Lock

In this section, we provide a detailed description of our generic QRL lock, the source code for which may be found in Appendix B.

When a thread attempts to acquire the QRL, there are four distinct cases to consider, based on the value of the status word.

Case 1: The first case occurs when the lock has never been held (so the lock's status word is `NEUTRAL`). In this case, the thread verifies the lock state (lines 1, 13, 16) and attempts to install itself as the lock's bias holder via a `CAS` instruction (lines 18-20). A `CAS` is necessary here to prevent two or more threads from simultaneously biasing the lock to themselves. (Threads that fail this `CAS` fall through to the revocation code, described below as the third acquisition case.) We note that some thread is guaranteed to complete the `CAS` operation successfully. When a thread successfully installs itself as the bias holder, this step simultaneously changes the lock state from `NEUTRAL` to `BIASED` and implicitly grants the lock to the new bias holder (hence the "1" in the first part of the DWORD in line 20). Finally, the return of "1" (line 22) indicates that the lock was acquired via (and should thus be released via) the ultra fast path.

Case 2: The expected common case for QRL locks occurs when a bias holder thread attempts to reacquire the lock. This case is detected immediately by checking the lock status word (line 4). In this case, the process simply resets the quicklock flag (line 6), verifies that the lock hasn't been revoked (line 7), and signals acquisition of the lock via the ultra fast path (line 8). In the event that the lock has been revoked, the bias holder clears the quicklock flag and falls through to the default acquisition case (lines 9 and 70-73). Note that the collocation technique described in the overview ensures that the resampling of the lock's status word (line 7) cannot be reordered above the write to the quicklock flag (line 6). This in turns ensures that if a bias holder attempts to reacquire the lock simultaneously with some other process attempting to revoke the lock, then either the revoker detects that the bias holder has reacquired the lock or the bias holder detects that the lock has been revoked. Thus, they do not both proceed to their critical sections.

Case 3: The next acquisition case occurs the first time that a thread other than the bias holder attempts to acquire the lock, detected by the lock status field being biased to another thread. Like the first acquisition case, this can happen at most once in the lifetime of the lock. This case is detected in line 30, when the thread determines that the lock is currently biased, but not to itself. Lines 34-37 set up and attempt a `CAS` operation to convert the lock from `BIASED` status to `REVOKING`. (`REVOKING` means that a thread is currently in the middle of revoking the lock.) This `CAS` is in a loop (lines 32, 60) because it could fail if the bias holder toggles the quicklock flag between lines 34 and 35. In any event, a fresh read of the status word is needed in this case; line 58 accomplishes this. Assuming that such a toggle eventually does not happen, the `CAS` ensures the existence of a unique revoking thread. Any other threads that were concurrently attempting to revoke the lock will note that the lock is no longer biased (line 60), and spin until the default lock is established by the revoker (lines 66-67). Finally, non-revokers fall through to the default acquisition path (lines 70-73). Meanwhile, the revoker thread initializes the default lock to an acquired state (line 40) and changes the lock status to `DEFAULT`. As demonstrated in the QRL-MCS and QRL-TATAS implementation examples in Appendix A, initializing the default lock is uncontended, so may be done without using atomic instructions that might otherwise be needed. Next, the revoker waits until the former bias holder is not holding the quicklock (lines 47-48). Finally, because successfully revoking the bias holder implicitly grants the first non-quick acquisition of the lock, the revoker signals success via the default path (line 49).

Case 4: The final case occurs when the lock status is `DEFAULT`. From this point forward, lock acquisition consists of three steps. First, a thread fails the two tests for non-`DEFAULT` status on lines 13 and 30. Next, it acquires the default lock using whatever default protocol is desired (implied here by line 72; this should actually be replaced with appropriate code for the default lock). Finally, it signals success via the default path (line 73) by returning 0.

Releasing a QRL is very simple. If the thread acquired the QRL via the fast path (line 74), it just resets the quicklock (line 75). Otherwise, it executes the default lock's release protocol (lines 76-79).

One other implementation note bears mention. Specifically, the `MAKEDWORD`, `LOWWORD`, and `HIGHWORD` macros are all configured for machines with a 32-bit big-endian memory hardware access scheme. Anyone skilled in the art can trivially create appropriate equivalent macros for other platforms as needed.

3.4 Other QRL Implementations

In this section we briefly describe other techniques that can be used to implement QRLs.

Our first alternative QRL uses signals during the revocation cycle. Specifically, a signal handler executed by the bias owner at the request of the revoking process can inspect the state of the QRL and switch it over to the default lock if the bias holder is not currently holding the lock. If the bias holder *is* currently holding the lock, the signal handler can set state such that the revoking process must wait until the bias holder releases its lock. Once this latter state is set, the bias holder cannot reacquire the QRL except via the default lock path. Source code for our signal-based lock may be found in Appendix B.

Although the signal-based approach works well in many cases, many large pre-existent code bases are not signal safe; adding signals to such code bases can introduce bugs. For example, the basic file I/O functions `read()` and `write()` report what looks like an I/O error if they are interrupted during an operation: additional checking is required to determine that the operation need only be retried. Finally, signals are not supported in all operating systems, so a signal-based lock is inherently limited to platforms that do support them.

A close relative to the signal-based alternative applies in a garbage-collected environment. Here, at stop-the-world garbage-collection time, all threads other than the garbage collector are paused. By adding a special revocation handler to the garbage collection process, it becomes possible to effect the revocation directly without worrying about concurrency. There are two major drawbacks to this approach. First, tying the revocation process to garbage collection requires that a revoking thread wait for the next garbage collection, which can be an arbitrarily long wait. Second, this approach is intimately tied to particular garbage collection implementations and implicitly requires the ability to stop threads.

For completeness, we mention that another alternative QRL can be constructed in a manner similar to our preferred embodiment, but using `membar` instructions instead of the collocation trick. Such an approach could conceivably be preferable if the practical cost of a membar were to become sufficiently small (because it would be portable to more memory models); however, this is not the case in current hardware. Source code for a `membar`-based QRL may be found in Appendix C. We note also that this implementation can be restructured to make the revoking process spin until the quicklock flag is cleared. An improvement of this form has already been incorporated into our preferred embodiment and may be seen in Appendix A.

As yet another example alternative QRL, Solaris cross-calls could be used to generate memory barriers in a remote processor when a revoking process needs to coordinate with it. In this approach, instead of executing a `membar` for every acquire and release by the bias holder, only a single `membar` would be needed at revocation time. This is sufficient to achieve an atomic-free lock; however, it would be highly operating system specific and thus difficult to port to other platforms.

4 Construction of Re-biasable Locks

QRL locks are optimized for a single-process repeated-acquisition data access pattern. However, another common pattern, migratory data access, would not get nearly as much benefit from a QRL lock. Nonetheless, a straightforward extension of the QRL concept can be used in order to support such access. Specifically, if we add the ability to rebias a QRL lock once contention dies down, this *reversion* will be sufficient.

Unlike the basic QRL scheme, rebiasing cannot easily be generalized to all underlying default lock types. This is because the absence of contention is more readily detected at release time for some locks, but at acquire time for others. For example, the best indication that contention has concluded in the MCS lock is if a lock holder has no successor in the lock queue, but the only place where contention can be detected in the TATAS lock is by counting the number of attempts required to CAS the lock to "held".

In the following subsections, we detail sample schemes for rebiasing QRL locks that use two different default locks. One skilled in the art could readily identify other options for creating rebiasable QRL locks; we provide these schemes to illustrate the concept.

4.1 Rebiasable QRL-MCS Lock Construction

To create a rebiasable QRL-MCS lock, the ideal place to perform the rebiasing is at release time. Here, the following code can be executed (where `I` is the current lock holder's MCS qnode and `L` is the QRL-MCS lock pointer) at the beginning of the default release path:

```
if (NULL == I->next && !I->flag) /* !I->flag: uncontended acquire */
{
    L->lockword = MAKEDWORD(REBIASING, 0);
    membar();
    if (NULL == I->next) /* resample */
    {
        L->lockword = MAKEDWORD(BIASED(id), 0);
        return;
    }
    L->lockword = MAKEDWORD(DEFAULT, 0);
}
...remainder of release code follows here...
```

This change adds a state `REBIASING` which is used to mark the reversion from `DEFAULT` to `BIASED` in the same way that `REVOKING` marks the transition from `BIASED` to `DEFAULT`. The code itself uses the same general resampling technique to verify that the reversion of the lock completed before any other process got in line in the default lock; this prevents a race condition that might otherwise occur.

In addition to this modification, updates to the QRL-MCS acquisition code are required. These are very similar to the code that pauses processes during the window while the lock is being revoked; therefore, we do not present them here.

4.2 Rebiasable QRL-TATAS Lock Construction

To create a rebiasable QRL-TATAS lock, two modifications to the basic scheme suffice. First, if a process is able to acquire the TATAS default lock on its first attempt, then this is an indication that either the process "got lucky" or that contention is low. (Conversely, if the process is not able to acquire the TATAS lock on its first CAS, this definitely means that contention remains.) So, a counter can be added to the lock that is only modified when a process holds the TATAS default lock. Initialized to some constant tuned to the application for which the lock is used, this counter is reset each time a process acquires the TATAS default lock under contention, and decremented each time a process acquires the TATAS default lock on its first attempt. If the counter ever hits zero, the current default lock holder simply needs to overwrite the QRL lock fields with a tuple indicating that the lock is biased to and currently acquired by the current lock holder. Effectively, then, the lock holder is switching to holding the QRL lock from the default TATAS lock.

A second needed modification is to add a check to the default lock acquisition path to test for reversion of the lock. When this is detected, a process simply retries the entire lock acquisition from the beginning of the acquire function.

5 Summary

We have presented QRL locks, a novel class of mutual exclusion algorithms that are heavily optimized for a very common data access pattern in which a single process repeatedly and solely acquires a lock. Our QRL locks represent the first true atomic-free locks for this ultra fast path. Because they can be generalized to use any mutual exclusion algorithm with a standard interface, as well as many algorithms that do not use a standard interface, QRL locks can obtain the benefits of any properties of such locks for the uncontended case at the expense of a mere handful of non-atomic instructions in their critical path. QRL locks are optimized for a single-process repeated-acquisition data access pattern; however, we have also demonstrated rebiasable QRLs that can be used with migratory data access patterns.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[2] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[3] D. Dice. Implementing fast java[tm] monitors with relaxed-locks. In *Proceedings of the USENIX JVM'01 Conference*, 2001.

[4] E. Dijkstra. Solution to a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[6] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[7] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[8] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9.* PTR Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1994.

A Collocation-based QRL implementations

A.1 Definitions Common to All Implementations

```
/* statuses for qrl locks */
#define BIASED(id)  ((int)(id) << 2)
#define NEUTRAL 1
#define DEFAULT    2
#define REVOKED 3
#define ISBIASED(status) (0 == ((status) & 3))

/* word manipulation (big-endian versions shown here) */
#define MAKEDWORD(low, high) (((unsigned int)(low) << 16) | (high))
#define HIGHWORD(dword) ((unsigned short)dword)
#define LOWWORD(dword) ((unsigned short)(((unsigned int)(dword)) >> 16))
```

A.2 Generalized Lock

```
typedef volatile struct tag_qrlgeneric_lock
{
  volatile union
  {
    volatile struct
    {
      volatile short quicklock;
      volatile short status;
    }
    h;
    volatile int data;
  }
  lockword;
  /* *** PLUS WHATEVER FIELDS ARE NEEDED FOR THE DEFAULT LOCK *** */
}
qrlgeneric_lock;

int qrlgeneric_acquire(qrlgeneric_lock *L, int id)
{
  int status = L->lockword.h.status;

  /* If the lock's mine, I can reenter by just setting a flag */
  if (BIASED(id) == status)
  {
    L->lockword.h.quicklock = 1;
    if (BIASED(id) == HIGHWORD(L->lockword.data))
      return 1;
    L->lockword.h.quicklock = 0; /* I didn't get the lock, so be sure */
                                 /* not to block the process that did */
  }

  if (DEFAULT != status)
  {
    /* If the lock is unowned, try to claim it */
    if (NEUTRAL == status)
    {
      if (CAS(&L->lockword,      /* By definition, if we saw */
        MAKEDWORD(0, NEUTRAL), /* neutral, the lock is unheld */
        MAKEDWORD(1, BIASED(id))))
      {
       return 1;
      }
      /* If I didn't bias the lock to me, someone else just grabbed
         it. Fall through to the revocation code */
      status = L->lockword.h.status; /* resample */
     }

     /* If someone else owns the lock, revoke them */
     if (ISBIASED(status))
     {
      do
      {
       unsigned short biaslock = L->lockword.h.quicklock;
       if (CAS(&L->lockword,
       MAKEDWORD(biaslock, status),
       MAKEDWORD(biaslock, REVOKED)))
       {
        /* I'm the revoker. Set up the default lock. */
        /* *** INITIALIZE AND ACQUIRE THE DEFAULT LOCK HERE *** */
        /* Note: this is an uncontended acquire, so it */
        /* can be done without use of atomics if this is */
        /* desirable. */
        L->lockword.h.status = DEFAULT;

        /* Wait until quicklock is free */
        while (LOWWORD(L->lockword.data))
         ;
        return 0; /* And then it's mine */
       }

       /* The CAS could have failed and we got here for either of
          two reasons. First, another process could have done the
          revoking; in this case we need to fall through to the
          default path once the other process is finished revoking.
          Secondly, the bias process could have acquired or released
          the biaslock field; in this case we need merely retry. */
       status = L->lockword.h.status;
      }
      while (ISBIASED(L->lockword.h.status));
     }

     /* If I get here, the lock has been revoked by someone other
        than me. Wait until they're done revoking, then fall through
        to the default code. */
     while (DEFAULT != L->lockword.h.status)
      ;
    }

    /* Regular default lock from here on */
    assert(DEFAULT == L->lockword.h.status);
    /* *** DO NORMAL (CONTENDED) DEFAULT LOCK ACQUIRE FUNCTION HERE *** */
```

```
73  return 0;
}

void qrlgeneric_release(qrlgeneric_lock *L, int acquiredquickly)
{
74  if (acquiredquickly)
75  L->lockword.h.quicklock = 0;
76  else
77  {
78    /* *** DO NORMAL DEFAULT LOCK RELEASE FUNCTION HERE *** */
79  }
}
```

A.3 MCS-based Lock

```
typedef struct tag_qrlmcs_node
{
  volatile struct tag_qrlmcs_node *next;
  volatile int flag;
}
qrlmcs_node;

typedef volatile struct tag_qrlmcs_lock
{
  volatile union
  {
  volatile struct
  {
    volatile short quicklock;
    volatile short status;
  }
  h;
  volatile int data;
  }
  lockword;
  volatile qrlmcs_node *defaultlock;
}
qrlmcs_lock;

void qrlmcs_initialize(qrlmcs_lock *L)
{
  L->lockword.data = MAKEDWORD(0, NEUTRAL);
  L->defaultlock = NULL;
}

int qrlmcs_acquire(qrlmcs_lock *L, qrlmcs_node *I, int id)
{
  int status = L->lockword.h.status;

  /* If the lock's mine, I can reenter by just setting a flag */
  if (BIASED(id) == status)
  {
    L->lockword.h.quicklock = 1;
    if (BIASED(id) == HIGHWORD(L->lockword.data))
      return 1;
    L->lockword.h.quicklock = 0; /* I didn't get the lock, so be sure */
  }                              /* not to block the process that did */

  if (DEFAULT != status)
  {
    /* If the lock is unowned, try to claim it */
    if (NEUTRAL == status)
    {
      if (CAS(&L->lockword,      /* By definition, if we saw */
          MAKEDWORD(0, NEUTRAL), /* neutral, the lock is unheld */
          MAKEDWORD(1, BIASED(id))))
      {
        return 1;
      }
      /* If I didn't bias the lock to me, someone else just grabbed
         it. Fall through to the revocation code */
      status = L->lockword.h.status; /* resample */
    }

    /* If someone else owns the lock, revoke them */
    if (ISBIASED(status))
    {
      do
      {
        unsigned short biaslock = L->lockword.h.quicklock;
        if (CAS(&L->lockword,
          MAKEDWORD(biaslock, status),
          MAKEDWORD(biaslock, REVOKED)))
        {
          /* I'm the revoker. Claim the head of the queue. */
          I->next = NULL;
          L->defaultlock = I;
          L->lockword.h.status = DEFAULT;

          /* Wait until lock is free */
          while (LOWWORD(L->lockword.data))
            ;
          return 0; /* And then it's mine */
        }

        /* The CAS could have failed and we got here for either of
           two reasons. First, another process could have done the
           revoking; in this case we need to fall through to the
           default path once the other process is finished revoking.
           Secondly, the bias process could have acquired or released
           the biaslock field; in this case we need merely retry. */
        status = L->lockword.h.status;
      }
      while (ISBIASED(L->lockword.h.status));
    }

    /* If I get here, the lock has been revoked by someone other
       than me. Wait until they're done revoking, then fall through
```

```
       to the default code. */
    while (DEFAULT != L->lockword.h.status)
      ;
  }

  /* Regular MCS from here on */
  assert(DEFAULT == L->lockword.h.status);
  I->next = NULL;
  qrlmcs_node *pred = (qrlmcs_node *)SWAP(&L->defaultlock, I);
  if (NULL != pred)
  {
    I->flag = 1;
    pred->next = I;
    while (I->flag)
      ;
  }
  return 0;
}

void qrlmcs_release(qrlmcs_lock *L, qrlmcs_node *I, int acquiredquickly)
{
  /* Releasing a quickly acquired lock is very easy */
  if (acquiredquickly)
  {
    L->lockword.h.quicklock = 0;
    return;
  }

  /* Otherwise, go through the MCS release procedure */
  if (NULL == I->next)
  {
    if (CAS(&L->defaultlock, I, NULL))
      return;
    while (NULL == I->next)
      ;
  }
  I->next->flag = 0;
}
```

A.4 TATAS-based Lock

```
#define QRL_BASE 50   /* Initial backoff value */
#define QRL_CAP  800  /* Maximum backoff value */

typedef struct tag_qrltas_lock
{
  volatile union
  {
    struct
    {
      short quicklock;
      short status;
    }
    h;
    int data;
  }
  lockword;
  volatile long defaultlock;
}
qrltas_lock;

void qrltas_initialize(qrltas_lock *L)
{
  L->lockword.data = MAKEDWORD(0, NEUTRAL);
  L->defaultlock = 0;
}

int qrltas_acquire(qrltas_lock *L, int id)
{
  int status = L->lockword.h.status;

  /* If the lock's mine, I can reenter by just setting a flag */
  if (BIASED(id) == status)
  {
    L->lockword.h.quicklock = 1;
    if (BIASED(id) == HIGHWORD(L->lockword.data))
      return 1;
    L->lockword.h.quicklock = 0; /* I didn't get the lock, so make sure I
                                    don't block up the process that did */
  }

  if (DEFAULT != status)
  {
    /* If the lock is unowned, try to claim it */
    if (NEUTRAL == status)
    {
      if (CAS(&L->lockword,      /* By definition, if we saw */
          MAKEDWORD(0, NEUTRAL), /* neutral, the lock is unheld */
          MAKEDWORD(1, BIASED(id))))
      {
        /* Biasing the lock counts as an acquisition */
        return 1;
      }
      /* If I didn't bias the lock to me, someone else just grabbed
         it. Fall through to the revocation code */
      status = L->lockword.h.status; /* resample */
    }

    /* If someone else owns the lock, revoke them */
    if (ISBIASED(status))
    {
      do
      {
        unsigned short biaslock = L->lockword.h.quicklock;
        if (CAS(&L->lockword,
          MAKEDWORD(biaslock, status),
          MAKEDWORD(biaslock, REVOKED)))
        {
          /* I'm the revoker. Claim my lock. */
          L->defaultlock = 1;
          L->lockword.h.status = DEFAULT;

          /* Wait until lock is free */
          while (LOWWORD(L->lockword.data))
            ;
          return 0; /* And then it's mine */
        }

      /* The CAS could have failed and we got here for either of
         two reasons. First, another process could have done the
         revoking; in this case we need to fall through to the
         default path once the other process is finished revoking.
         Secondly, the bias process could have acquired or released
         the biaslock field; in this case we need merely retry. */
      status = L->lockword.h.status;
      }
      while (ISBIASED(L->lockword.h.status));
    }

    /* If I get here, the lock has been revoked by someone other
       than me. Wait until they're done revoking, then fall through
       to the default code. */
    while (DEFAULT != L->lockword.h.status)
      ;
  }

  /* Regular Tatas from here on */
  assert(DEFAULT == L->lockword.h.status);
  while (!CAS(&L->defaultlock, 0, 1))
  while (L->defaultlock)
    ;
  return 0;
}

void qrltas_release(qrltas_lock *L, int acquiredquickly)
{
  if (acquiredquickly)
    L->lockword.h.quicklock = 0;
  else
    L->defaultlock = 0;
}
```

B Appendix II: Signal-based QRL implementation

```
// UMux.Bias encoding:
// 0|00 : Neutral == NULL
// T|00 : Biased - unlocked
// T|01 : Biased - locked
// 0|10 : Default
//
// T is of type "UThread *" or a direct "raw" %g7 value.

#define BNEUTRAL 0
#define BBIAS 1
#define BDEFAULT 2

// Support alternate "thread" encodings:
// a. raw = G7
// b. raw = UThread *

#define RAWID TLSSELF // UThread *
#define RAW2T(r) (r)

#define RAWID _RawSelf // g7 -> ulwp_t or uthread_t
#define RAW2T(r) EELookup(r)

static
int revTrap ( int signo, siginfo_t * si, ucontext_t * ctx)
{
    uintptr_t * regv ;
    uintptr_t ip, rs, b ;
    UThread * Self ;
    int isp, err ;
    UMux * m ;

    MEMBAR(StoreLoad) ;
    regv  = ctx->uc_mcontext.gregs ;
    printf ("Signal %d: code=%X addr=%X trp=%X pc=%X USP=%X ISP=%X\n",
        signo,
        si->si_code,
        si->__data.__fault.__addr,
        si->__data.__fault.__trapno,
        regv[REG_PC],
        regv[REG_O6],
        &isp) ;

    // If interrupt IP is within a critical section, restart it.
    ip = regv[REG_PC] ;
    rs = csRestart (ip) ;
    if (rs == NULL) rs = QRestart(ip) ;
    if (rs != 0) {
        if (Verbose) printf ("Restart %X\n", rs) ;
        regv[REG_PC]  = rs ;
        regv[REG_nPC] = rs + 4 ;
    }
    Self = TLSSELF ;
    m = Self->revMux ;
    ASSERT (m && m->Bias == (RAWID|BBIAS)) ;

    // Promote/Inflate: Convert to heavy-weight lock
    b = m->Bias ;
    if (b & BBIAS) {
        err = _lwp_mutex_trylock (m->Inflated) ;
        ASSERT (err == 0) ;
    }
    m->Bias = BDEFAULT ;
    Self->revMux = NULL ;

    return 0 ;
}

static
int    Revoke (UThread * t, UMux * m)             // Cancel reservation
{

    if (Verbose) printf ("REVOKE (%X %X)\n", t, m) ;

    _lwp_mutex_lock (&t->revLock) ;
    if (m->Bias == BDEFAULT) {
        _lwp_mutex_unlock (&t->revLock) ;
        return 0 ;
    }

    // Promote to heavy-weight inflated state
    // Should _this thread lock m->Inflated, or the revokee?
    ASSERT (m->Inflated == NULL) ;
    m->Inflated = (lwp_mutex_t *) malloc (sizeof(lwp_mutex_t)) ;
    memset (m->Inflated, 0, sizeof(lwp_mutex_t)) ;

    // Make a synthetic synchronous RPC call : revTrap (m)
    t->revMux = m ;
    thr_kill (t->ThreadID, SIGUSR1) ;         // X-call
    while (t->revMux != NULL) ;             // spin

    _lwp_mutex_unlock (&t->revLock) ;
    return 0 ;
}

int    ILock (UMux * m)
{
    uintptr_t b ;
    UThread * Self ;

    Self = RAWID ;

    // Optimisitic agro form ... avoids RTS->RTO upgrade on MP systems.
    if (Agro && csCAS (&m->Bias, Self, INT(Self)|BBIAS) == Self) return 0 ;

  Retry:
    b = m->Bias ;
    if (b == Self) {
        if (csCAS (&m->Bias, b, b|BBIAS) == b) { return 0 ; }
        goto Retry ;
    }
    if (b == BDEFAULT)     { goto DefaultPath ; }
    if (b == BNEUTRAL)     { CAS (&m->Bias, b, Self) ; goto Retry ; }
    ASSERT (b != (INT(Self)|BBIAS)) ;
    Revoke (RAW2T(INT(b) & ~BBIAS), m) ;
  DefaultPath:
    return _lwp_mutex_lock (m->Inflated) ;
}

int    IUnlock (UMux * m)
{
    UThread * Self ;
    uintptr_t b ;
    uintptr_t bs ;

    Self = RAWID ;
    bs = INT(Self)|BBIAS ;
    if (Agro && csCAS (&m->Bias, bs, Self) == bs) return 0 ;

  Retry:
    b = m->Bias ;
    if (b == (INT(Self)|BBIAS)) {
        if (csCAS (&m->Bias, b, Self) == b) {
            return 0 ;
        }
        goto Retry ;
    }
    ASSERT (b == BDEFAULT) ; // Caveat: Proper error checking needed here
    return _lwp_mutex_unlock (m->Inflated) ;
}

int    ITry (UMux * m)
{
    return 0 ;
}
```

C Appendix III: membar-based QRL implementation

```
int qrl_membar_lock (UMux * m)
{
    Thread * Self ;
    Thread * Bias ;
    Self = GetReflexiveSelf ();

  Top:
    Bias = m->Bias ;
    if (Bias == Self) {                 // ultra fast-path locking
        ASSERT (Self->InCrit == NULL) ;
        Self->InCrit = m ;              // Enter inner critical section
        MEMBAR(StoreLoad) ;             // store incrit, load bias
        if (m->Bias == Self) {          // resample
            ASSERT (m->Owner == NULL);  // error check / diagnostics
            m->Owner = Self ;           // Take ownership of the mutex
            Self->InCrit = NULL ;       // Exit inner critical section
            return 0 ;                  // Success
        }
        Self->InCrit = NULL ;           // Exit inner critical section
        MEMBAR(StoreLoad) ;
        goto Top ;
    }
    if (Bias != DEFAULT) {
        if (Bias == NULL) {             // Neutral ?
            CAS (&m->Bias, NULL, Self); // transition neutral->biased
            goto Top ;
        } else {
            if (Bias == REVOKING)
                goto Top ;              // spin
            // Contention: demote and revoke the oplock
            // 1st revoker performs the revocation
            // subsequent revokers spin until the 1st revoker completes.
            // Store bias, Load incrit
            if (CAS(&m->Bias, Bias, REVOKING) != Bias) {
                goto Top ;
            }
            MEMBAR(StoreLoad) ;
            while (Bias->InCrit == m) ; // spin while CS is occupied
            ASSERT (m->Bias == REVOKING) ;
            m->Bias = DEFAULT ;            // transition biased->default
            goto Top ;
        }
    }
    ASSERT (Bias == DEFAULT) ;

    ... default path ...
    // The ultra-fast path and the default path share UMux.Owner.
    Acquire lock via CAS (&m->Owner, NULL, Self)
}


int qrl_membar_unlock (UMux * m)
```

```
{
    UThread * Self = GetReflexiveSelf ();

  Top:
    if (m->Bias == Self) {
        ASSERT (Self->InCrit == NULL) ;
        Self->InCrit = m ;          // Enter inner critical section
        MEMBAR(StoreLoad) ;         // Store incrit, load bias
        if (m->Bias == Self) {      // resample
            m->Owner = NULL ;       // Drop ownership of the lock
            Self->InCrit = NULL ;   // Exit inner critical section
            return 0 ;
        }
        Self->InCrit = NULL ;       // Exit inner critical section
        MEMBAR(StoreLoad) ;
        goto Top ;
    }

    ... default path ...
    m->Owner = NULL ; wakeup successors, etc.
}
```