Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors

Pseudocode from article of the above name, *PPoPP '91*. John M. Mellor-Crummey and Michael L. Scott. The queue-based locks below perform well in tests on machines with scores of processors.

- Simple, non-scalable reader-preference lock. For maximum throughput on small machines. Starves writers under continuous reader load. Starvation of individual writers is theoretically possible even under non-continuous load, though this is unlikely in practice. Will not starve readers. Employs atomic_add and compare_and_store.
- Simple, non-scalable writer-preference lock. (Not in the *PPoPP* paper.) Grants FIFO access to writers. Starves readers under continuous writer load. Employs atomic_add and compare_and_store.
- Simple, non-scalable, fair reader-writer lock Starvation-free. Employs fetch_and_clear_then_add.
- Reader-preference queue-based lock with local-only spinning. For maximum throughput on large machines. Starves writers under continuous reader load; grants them FIFO access otherwise. Will not starve readers. Employs compare_and_store, fetch_and_add, fetch_and_and, fetch_and_or, and fetch_and_store.
- Writer-preference queue-based lock with local-only spinning. Grants FIFO access to writers. Starves readers under continuous writer load. Employs compare_and_store, fetch_and_add, fetch_and_or, and fetch_and_store.
- Fair queue-based reader-writer lock with local-only spinning. Starvation-free. Employs atomic_increment, compare_and_store, fetch_and_decrement, and fetch_and_store. Incorporates a bug fix due to Keir Fraser.

Simple, non-scalable reader-preference lock

```
type lock = unsigned integer
  // layout of lock
  //  31          ...           1        0
  // +-------------------------+----------------+
  // | interested reader count | active writer? |
  // +-------------------------+----------------+

const WAFLAG = 0x1  // writer active flag
const RC_INCR = 0x2 // to adjust reader count

procedure start_write (L : ^lock)
    repeat until compare_and_store (L, 0, WAFLAG)

procedure start_read (L : ^lock)
    atomic_add (L, RC_INCR)
    repeat until (L^ & WAFLAG) = 0

procedure end_write (L : ^lock)
    atomic_add (L, -WAFLAG)

procedure end_read (L : ^lock)
    atomic_add (L, -RC_INCR)
```

Simple, non-scalable writer-preference lock

```
type lock = record
    write_requests, write_completions : unsigned integer := 0, 0
    rdr_cnt_and_flag : unsigned integer
        // layout
        //  31          ...           1        0
        // +-------------------------+----------------+
        // | interested reader count | active writer? |
```

```
        // +-------------------------+----------------+

    const WFLAG = 1     // writer active flag
    const RC_INCR = 2   // constant used to adjust the reader count

procedure start_read (L : ^lock)
    repeat until L->write_requests = L->write_completions
    atomic_add (&L->rdr_cnt_and_flag, RC_INCR)
    repeat until (L->rdr_cnt_and_flag & WFLAG) = 0

procedure start_write (L : ^lock)
    previous_writers : unsigned integer
        := fetch_and_increment (&L->write_requests)
    repeat until L->write_completions = previous_writers
    repeat until compare_and_store (&L->rdr_cnt_and_flag, 0, WFLAG)

procedure end_read (L : ^lock)
    atomic_add (&L->rdr_cnt_and_flag, -RC_INCR)

procedure end_write (L: ^lock)
    atomic_add (&L->rdr_cnt_and_flag, -WFLAG)
    atomic_increment (&L->write_completions)
```

Simple, non-scalable, fair reader-writer lock

```
type counter = unsigned integer
   // layout of counter
   //  31    ...   16 15    ...     0
   // +-------------------------------+
   // | reader count | writer count  |
   // +-------------------------------+

const RC_INCR = 0x10000 // to adjust reader count
const WC_INCR = 0x1     // to adjust writer count
const W_MASK  = 0xffff  // to extract writer count

// mask bit for top of each count
const WC_TOPMSK = 0x8000
const RC_TOPMSK = 0x80000000

type lock = record
    requests : counter := 0
    completions : counter := 0

procedure start_write (L : ^lock)
    counter prev_processes :=
        fetch_and_clear_then_add (&L->requests, WC_TOPMSK, WC_INCR)
    repeat until completions = prev_processes

procedure start_read (L : ^lock)
    counter prev_writers :=
        fetch_and_clear_then_add (&L->requests, RC_TOPMSK, RC_INCR) & W_MASK
    repeat until (completions & W_MASK) = prev_writers

procedure end_write (L: ^lock)
    clear_then_add (&L->completions, WC_TOPMSK, WC_INCR)

procedure end_read (L : ^lock)
    clear_then_add (&L->completions, RC_TOPMSK, RC_INCR)
```

Reader-preference queue-based lock with local-only spinning

```
type qnode = record
    next : ^qnode
    blocked : Boolean

type RPQlock = record
    reader_head : ^qnode := nil
    writer_tail : ^qnode := nil
```

```
    writer_head : ^qnode := nil
    rdr_cnt_and_flags : unsigned integer := 0

// layout of rdr_cnt_and_flags:
//  31     ...      2       1              0
// +-----------------+-------------+-----------------+
// | interested rdrs | active wtr? | interested wtr? |
// +-----------------+-------------+-----------------+

const WIFLAG = 0x1  // writer interested flag
const WAFLAG = 0x2  // writer active flag
const RC_INCR = 0x4 // to adjust reader count

// I points to a qnode record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure start_write (L : ^RPQlock, I : ^qnode)
    with I^, L^
        blocked := true
        next := nil
        pred: ^qnode := fetch_and_store (&writer_tail,I)
        if pred = nil
            writer_head := I
            if fetch_and_or (&rdr_cnt_and_flag, WIFLAG) = 0
                if compare_and_store (&rdr_cnt_and_flag, WIFLAG, WAFLAG)
                    return
            // else readers will wake up the writer
        else
            pred->next := I
        repeat while blocked

procedure end_write (L: ^RPQlock, I : ^qnode)
    with I^, L^
        writer_head := nil
        // clear wtr flag and test for waiting readers
        if fetch_and_and (&rdr_cnt_and_flag, ~WAFLAG) != 0
            // waiting readers exist
            head : ^qnode := fetch_and_store (&reader_head, nil)
            if head != nil
                head->blocked := false
        // testing next is strictly an optimization
        if next != nil or not compare_and_store (&writer_tail, I, nil)
            repeat while next = nil  // resolve successor
            writer_head := next
            if fetch_and_or (&rdr_cnt_and_flag,WIFLAG) = 0
                if compare_and_store (&rdr_cnt_and_flag, WIFLAG, WAFLAG)
                    writer_head->blocked := false
            // else readers will wake up the writer

procedure start_read (L : ^RPQlock, I : ^qnode)
    with I^, L^
        // incr reader count, test if writer active
        if fetch_and_add (&rdr_cnt_and_flag, RC_INCR) & WAFLAG
            blocked := true
            next := fetch_and_store (&reader_head, I)
            if (rdr_cnt_and_flag & WAFLAG) = 0
                // writer no longer active; wake any waiting readers
                head : ^qnode := fetch_and_store (&reader_head, nil)
                if head != nil
                    head->blocked := false
            repeat while blocked            // spin
            if next != nil
                next->blocked := false

procedure end_read (L : ^RPQlock, I : ^qnode)
    with I^, L^
        // if I am the last reader, resume the first
        // waiting writer (if any)
        if fetch_and_add (&rdr_cnt_and_flag, -RC_INCR) = (RC_INCR + WIFLAG)
            if compare_and_store (&rdr_cnt_and_flag, WIFLAG, WAFLAG)
                writer_head->blocked := false
```

Writer-preference queue-based lock with local-only spinning

```
type qnode : record
    next : ^qnode
    blocked : Boolean

type WPQlock = record
    reader_head : ^qnode := nil
    writer_tail : ^qnode := nil
    writer_head : ^qnode := nil
    rdr_cnt_and_flags : unsigned integer := 0

// layout of rdr_cnt_and_flags:
//  31   ...    3         2          1          0
// +-------------+----------+--------------+------+
// | active rdrs | int. rdr? | wtr & no rdr? | wtr? |
// +-------------+----------+--------------+------+

const WFLAG1 = 0x1  // writer interested or active
const WFLAG2 = 0x2  // writer, no entering rdr
const RFLAG = 0x4   // rdr int. but not active
const RC_INCR = 0x8 // to adjust reader count

// I points to a qnode record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure start_write (L : ^WPQlock, I : ^qnode)
    with I^, L^
        blocked := true
        next := nil
        pred : ^qnode := fetch_and_store (&writer_tail, I)
        if pred = nil
            set_next_writer (L, I)
        else
            pred->next := I
        repeat while blocked

procedure set_next_writer (L : ^WPQlock, W : ^qnode)
    with L^
        writer_head := W
        if not (fetch_and_or (&rdr_cnt_and_flags,WFLAG1) & RFLAG)
            // no reader in timing window
            if not (fetch_and_or (&rdr_cnt_and_flags, WFLAG2) >= RC_INCR)
                // no readers are active
                W->blocked := false

procedure start_read (L : ^WPQlock, I : ^qnode)
    with I^, L^
        blocked := true
        next := fetch_and_store (&reader_head, I)
        if next = nil
            // first arriving reader in my group
            // set rdr interest flag, test writer flag
            if not (fetch_and_or (&rdr_cnt_and_flags, RFLAG)
                    & (WFLAG1 + WFLAG2))
                // no active or interested writers
                unblock_readers (L)
        repeat while blocked
        if next != nil
            atomic_add (&rdr_cnt_and_flags, RC_INCR)
            next->blocked := false // wake successor

procedure unblock_readers (L : ^WPQlock)
    with L^
        // clear rdr interest flag, increment rdr count
        atomic_add (&rdr_cnt_and_flags, RC_INCR - RFLAG)
        // indicate clear of window
        if (rdr_cnt_and_flags & WFLAG1) and not (rdr_cnt_and_flags & WFLAG2)
            atomic_or (&rdr_cnt_and_flags, WFLAG2)
        // unblock self and any other waiting rdrs
        head : ^qnode := fetch_and_store (&reader_head, nil)
        head->blocked := false

procedure end_write (L : ^WPQlock, I : ^qnode)
```

```
    with I^, L^
        if next != nil
            next->blocked := false
        else
            // clear writer flag, test reader interest flag
            if fetch_and_and (&rdr_cnt_and_flags, ~(WFLAG1 + WFLAG2)) & RFLAG
                unblock_readers (L)
            if compare_and_store (&writer_tail, I, nil)
                return
            else
                repeat while next = nil
                set_next_writer (L, next)

procedure end_read (L : ^WPQlock, I : ^qnode)
    with I^, L^
        if (fetch_and_add (&rdr_cnt_and_flags, -RC_INCR) & ~RFLAG)
                = (RC_INCR + WFLAG1 + WFLAG2)
            // last active rdr must wake waiting writer
            writer_head->blocked := false
        // if only WFLAG1 is set and not WFLAG2, then
        // the writer that set it will take care of itself
```

Fair queue-based reader-writer lock with local-only spinning

```
type qnode = record
    class : (reading, writing)
    next : ^qnode
    state : record
        blocked : Boolean    // need to spin
        successor_class : (none, reader, writer)
type lock = record
    tail : ^qnode := nil
    reader_count : integer := 0
    next_writer : ^qnode := nil

// I points to a qnode record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor

procedure start_write (L : ^lock; I : ^qnode)
    with I^, L^
        class := writing; next := nil
        state := [true, none]
        pred : ^qnode := fetch_and_store (&tail, I)
        if pred = nil
            next_writer := I
            if reader_count = 0 and fetch_and_store (&next_writer,nil) = I
                // no reader who will resume me
                blocked := false
        else
            // must update successor_class before updating next
            pred->successor_class := writer
            pred->next := I
        repeat while blocked

procedure end_write (L: ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not compare_and_store (&tail, I, nil)
            // wait until succ inspects my state
            repeat while next = nil
            if next->class = reading
                atomic_increment (&reader_count)
            next->blocked := false

procedure start_read (L : ^lock; I : ^qnode)
    with I^, L^
        class := reading; next := nil
        state := [true, none]
        pred : ^qnode := fetch_and_store (&tail, I)
        if pred = nil
            atomic_increment (&reader_count)
            blocked := false  // for successor
        else
```

```
            if pred->class = writing or compare_and_store (&pred->state,
                        [true, none], [true, reader])
                // pred is a writer, or a waiting reader
                // pred will increment reader_count and release me
                pred->next := I
                repeat while blocked
            else
                // increment reader_count and go
                atomic_increment (&reader_count)
                pred->next := I
                blocked := false
        if successor_class = reader
            repeat while next = nil
            atomic_increment (&reader_count)
            next->blocked := false

procedure end_read (L : ^lock; I : ^qnode)
    with I^, L^
        if next != nil or not compare_and_store (&tail, I, nil)
            // wait until successor inspects my state
            repeat while next = nil
            if successor_class = writer
                next_writer := next
        if fetch_and_decrement (&reader_count) = 1
                and (w := next_writer) != nil
                and reader_count = 0
                and compare_and_store (&next_writer, w, nil)
            // I'm the last active reader and there exists a waiting
            // writer and no readers *after* identifying the writer
            w->blocked := false
```

NB: This code for end_read differs from that published in the *PPoPP* paper. The version here reflects a bug fix submitted by Keir Fraser of Cambridge University in January 2003.

Last Change: 8 December 2003 / scott@cs.rochester.edu