Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

Pseudocode from article of the above name, *ACM TOCS*, February 1991. John M. Mellor-Crummey and Michael L. Scott, with later additions due to (a) Craig, Landin, and Hagersten, and (b) Auslander, Edelsohn, Krieger, Rosenburg, and Wisniewski. All of these algorithms (except for the non-scalable centralized barrier) perform well in tests on machines with scores of processors.

Spinlocks

- Simple test_and_set lock with exponential backoff. Similar to code developed by Tom Anderson (*IEEE TPDS*, January 1990). Grants requests in unpredictable order; starvation is theoretically possible, but highly unlikely in practice. Spins (with backoff) on remote locations. Requires test_and_set.
- Ticket lock with proportional backoff. Grants requests in FIFO order. Spins (with backoff) on remote locations. Requires fetch_and_increment.
- Anderson's array-based queue lock (*IEEE TPDS*, January 1990). Grants requests in FIFO order. Requires $O(pn)$ space for p processes and n locks. Spins only on local locations on a cache-coherent machine. Requires fetch_and_increment and atomic_add.
- Graunke and Thakkar's array-based queue lock (*IEEE Computer*, June 1990). Grants requests in FIFO order. Requires $O(pn)$ space for p processes and n locks. Spins only on local locations on a cache-coherent machine. Requires fetch_and_store.
- The MCS list-based queue lock. Grants requests in FIFO order. Requires only $O(p+n)$ space for p processes and n locks. Requires a local "queue node" to be passed in as a parameter (alternatively, additional code can allocate these dynamically in acquire_lock, and look them up in a table in release_lock). Spins only on local locations on both cache-coherent and non-cache-coherent machines. Requires fetch_and_store and (ideally) compare_and_swap.
- The CLH list-based queue lock. Discovered independently by Travis Craig at the University of Washington (UW TR 93-02-02, February 1993), and by Anders Landin and Eric Hagersten of the Swedish Institute of Computer Science (*IPPS*, 1994). Requires $2p + 3n$ words of space for p processes and n locks (Cf. $2p + n$ for the MCS lock). Requires a local "queue node" to be passed in as a parameter. Spins only on local locations on a cache-coherent machine. Can be modified, with an extra level of indirection, to spin only on local locations on a non-cache-coherent machine as well (variant not shown here). Requires fetch_and_store.
- The K42 MCS variant. Alternative version of MCS that avoids the need to pass a queue node as argument. Due to Marc Auslander, David Edelsohn, Orran Krieger, Bryan Rosenburg, and Robert W. Wisniewski of the K42 group at IBM's T. J. Watson Research Center. Has a spin in release_lock that is local only on a machine that caches remote locations coherently. Also, as originally designed, has a loop in acquire_lock that can, in principle, perform an unbounded number of remote references, but can reasonably be expected not to do so in practice. Can be modified to perform only a constant number of remote references, but at the cost of an extra atomic operation on the uncontended path. As of summer 2006 this lock has not been published in any technical forum, but IBM has applied for a US patent.

Barriers

- A sense-reversing centralized (non-scalable) barrier. Similar to code employed by Hensgen, Finkel, and Manber (*IJPP*, 1988), and to a technique attributed to Dimitrovsky (*Highly Parallel Computing*, Almasi and Gottlieb, Benjamin/Cummings, 1989). *Omega(p)* operations on critical path; unbounded total number of remote operations. Constant space. Requires fetch_and_decrement.
- A software combining tree barrier with optimized wakeup. Similar to code developed by Yew, Tzeng, and Lawrie (*IEEE TC*, April 1987). *Omega(log p)* operations on critical path; $O(p)$ total remote operations on cache-coherent machine, unbounded on non-cache-coherent. $O(p)$ space. Requires fetch_and_decrement.
- Hensgen, Finkel, and Manber's dissemination barrier (*IJPP*, 1988). Improves on the earlier "butterfly" barrier of Brooks (*IJPP*, 1986). *Theta(log p)* operations on critical path; *Theta(p log p)* total remote operations. $O(p)$ space. Requires no atomic operations other than load and store.
- Tournament barrier with tree-based wakeup. Arrival phase due to Hensgen, Finkel, and Manber (*IJPP*, 1988). Also similar to a barrier of Lubachevsky (*ICPP '89*). Modifications also developed independently by Craig Lee (*SPDP*

'90). *Theta(log n)* operations on critical path (larger constant than dissemination barrier); *Theta(p)* total remote operations. *O(p)* space. Requires no atomic operations other than load and store.

- A simple scalable tree-based barrier. *Theta(log p)* operations on critical path; *2p-2* total remote operations (minimum possible without broadcast). *O(p)* space. Requires no atomic operations other than load and store.

Simple test_and_set lock with exponential backoff

```
type lock = (unlocked, locked)

procedure acquire_lock (L : ^lock)
    delay : integer := 1
    while test_and_set (L) = locked     // returns old value
        pause (delay)                   // consume this many units of time
        delay := delay * 2

procedure release_lock (L : ^lock)
    lock^ := unlocked
```

Ticket lock with proportional backoff

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    my_ticket : unsigned integer := fetch_and_increment (&L->next_ticket)
        // returns old value; arithmetic overflow is harmless
    loop
        pause (my_ticket - L->now_serving)
            // consume this many units of time
            // on most machines, subtraction works correctly despite overflow
        if L->now_serving = my_ticket
            return

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Anderson's array-based queue lock

```
type lock = record
    slots : array [0..numprocs -1] of (has_lock, must_wait)
        := (has_lock, must_wait, must_wait, ..., must_wait)
        // each element of slots should lie in a different memory module
        // or cache line
    next_slot : integer := 0

// parameter my_place, below, points to a private variable
// in an enclosing scope

procedure acquire_lock (L : ^lock, my_place : ^integer)
    my_place^ := fetch_and_increment (&L->next_slot)
        // returns old value
    if my_place^ mod numprocs = 0
        atomic_add (&L->next_slot, -numprocs)
        // avoid problems with overflow; return value ignored
    my_place^ := my_place^ mod numprocs
    repeat while L->slots[my_place^] = must_wait      // spin
    L->slots[my_place^] := must_wait                  // init for next time
```

```
procedure release_lock (L : ^lock, my_place : ^integer)
    L->slots[(my_place^ + 1) mod numprocs] := has_lock
```

Graunke and Thakkar's array-based queue lock

```
type lock = record
    slots : array [0..numprocs -1] of Boolean := true
        // each element of slots should lie in a different memory module
        // or cache line
    tail : record
        who_was_last : ^Boolean := 0
        this_means_locked : Boolean := false
        // this_means_locked is a one-bit quantity.
        // who_was_last points to an element of slots.
        // if all elements lie at even addresses, this tail "record"
        // can be made to fit in one word
processor private vpid : integer // a unique virtual processor index

procedure acquire_lock (L : ^lock)
    (who_is_ahead_of_me : ^Boolean, what_is_locked : Boolean)
        := fetch_and_store (&L->tail, (&slots[vpid], slots[vpid]))
    repeat while who_is_ahead_of_me^ = what_is_locked

procedure release_lock (L : ^lock)
    L->slots[vpid] := not L->slots[vpid]
```

The MCS list-based queue lock

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode      // initialized to nil

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

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil      // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked              // spin

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil        // no known successor
        if compare_and_store (L, I, nil)
            return
            // compare_and_store returns true iff it stored
        repeat while I->next = nil          // spin
    I->next->locked := false
```

Alternative version of release_lock, without compare_and_store:

```
procedure release_lock (L : ^lock, I : ^qnode)
    if I->next = nil        // no known successor
        old_tail : ^qnode := fetch_and_store (L, nil)
        if old_tail = I     // I really had no successor
            return
        // we have accidentally removed some processor(s) from the queue;
```

```
        // we need to put them back
        usurper := fetch_and_store (L, old_tail)
        repeat while I->next = nil          // wait for pointer to victim list
        if usurper != nil
            // somebody got into the queue ahead of our victims
            usurper->next := I->next        // link victims after the last usurper
        else
            I->next->locked := false
    else
        I->next->locked := false
```

The CLH list-based queue lock

```
type qnode = record
    prev : ^qnode
    succ_must_wait : Boolean
type lock = ^qnode  // initialized to point to an unowned qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->succ_must_wait := true
    pred : ^qnode := I->prev := fetch_and_store (L, I)
    repeat while pred->succ_must_wait

procedure release_lock (ref I : ^qnode)
    pred : ^qnode := I->prev
    I->succ_must_wait := false
    I := pred       // take pred's qnode
```

The K42 MCS variant

```
// Locks and queue nodes use the same data structure:
type lnode = record
    next : ^lnode
    union
        locked : Boolean    // for queue nodes
        tail : ^lnode       // for locks
type lock = lnode

// If threads are waiting for a held lock, next points to the queue node
// of the first of them, and tail to the queue node of the last.
// A held lock with no waiting threads has value <&head, nil>.
// A free lock with no waiting threads has value <nil, nil>.

procedure acquire_lock (L : ^lnode)
    I : lnode
    loop
        predecessor : ^lnode := L->tail
        if predecessor = nil
            // lock appears not to be held
            if compare_and_store (&L->tail, nil, &L->next)
                // I have the lock
                return
        else
            // lock appears to be held
            I.next := nil
            if compare_and_store (&L->tail, predecessor, &I)
                // I'm in line
                I.locked := true
                predecessor->next := &I
                repeat while I.locked             // wait for lock

                // I now have the lock
                successor : ^lnode := I.next
```

```
            if successor = nil
                L->next := nil
                if ! compare_and_store (&L->tail, &I, &L->next)
                    // somebody got into the timing window
                    repeat
                        successor := I.next
                    while successor = nil     // wait for successor
                    L->next := successor
                return
            else
                L->next := successor
                return

procedure release_lock (L : ^lnode)
    successor : ^lnode := L->next
    if successor = nil        // no known successor
        if compare_and_store (&L->tail, &L->next, nil)
            return
        repeat
            successor := L->next
        while successor = nil             // wait for successor
    successor->locked := false
```

Alternative version of acquire_lock, without remote spin:

```
procedure acquire_lock (L : ^lnode)
    I : lnode
    I.next := nil

    predecessor : ^lnode := fetch_and_store (&L->tail, &I)
    if predecessor != nil      // queue was non-empty
        I.locked := true
        predecessor->next := &I
        repeat while I.locked               // wait for lock

    // I now have the lock
    successor : ^lnode := I.next
    if successor = nil
        L->next := nil
        if ! compare_and_store (&L->tail, &I, &L->next)
            // somebody got into the timing window
            repeat
                successor := I.next
            while successor = nil           // wait for successor
            L->next := successor
    else
        L->next := successor
```

A sense-reversing centralized (non-scalable) barrier

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense   // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense         // last processor toggles global sense
    else
        repeat until sense = local_sense
```

A software combining tree barrier with optimized wakeup

```
type node = record
    k : integer             // fan-in of this node
    count : integer         // initialized to k
    locksense : Boolean     // initially false
    parent : ^node          // pointer to parent node; nil if root

shared nodes : array [0..P-1] of node
    // each element of nodes allocated in a different memory module or cache line
processor private sense : Boolean  := true
processor private mynode : ^node    // my group's leaf in the combining tree

procedure combining_barrier
    combining_barrier_aux (mynode)      // join the barrier
    sense := not sense                  // for next barrier

procedure combining_barrier_aux (nodepointer : ^node)
    with nodepointer^ do
        if fetch_and_decrement (&count) = 1     // last one to reach this node
            if parent != nil
                combining_barrier_aux (parent)
            count := k                          // prepare for next barrier
            locksense := not locksense          // release waiting processors
        repeat until locksense = sense
```

Hensgen, Finkel, and Manber's dissemination barrier

```
type flags = record
    myflags : array [0..1] of array [0..LogP-1] of Boolean
    partnerflags : array [0..1] of array [0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags
    // allnodes[i] is allocated in shared memory
    // locally accessible to processor i

// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//    allnodes[i].partnerflags[r][k] points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for instance : integer := 0 to LogP-1
        localflags^.partnerflags[parity][instance]^ := sense
        repeat until localflags^.myflags[parity][instance] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Tournament barrier with tree-based wakeup

```
type round_t = record
    role : (winner, loser, bye, champion, dropout)
    opponent : ^Boolean
    flag : Boolean
shared rounds : array [0..P-1][0..LogP] of round_t
    // row vpid of rounds is allocated in shared memory
    // locally accessible to processor vpid
processor private sense : Boolean := true
```

```
processor private vpid : integer  // a unique virtual processor index

// initially
//    rounds[i][k].flag = false for all i,k
// rounds[i][k].role =
//    winner if k > 0, i mod 2^k = 0, i + 2^(k-1) < P, and 2^k < P
//    bye if k > 0, i mod 2^k = 0, and i + 2^(k-1) >= P
//    loser if k > 0 and i mod 2^k = 2^(k-1)
//    champion if k > 0, i = 0, and 2^k >= P
//    dropout if k = 0
//    unused otherwise; value immaterial
// rounds[i][k].opponent points to
//    rounds[i-2^(k-1)][k].flag if rounds[i][k].role = loser
//    rounds[i+2^(k-1)][k].flag if rounds[i][k].role = winner or champion
//    unused otherwise; value immaterial

procedure tournament_barrier
    round : integer := 1
    loop                // arrival
        case rounds[vpid][round].role of
            loser:
                rounds[vpid][round].opponent^ := sense
                repeat until rounds[vpid][round].flag = sense
                exit loop
            winner:
                repeat until rounds[vpid][round].flag = sense
            bye:        // do nothing
            champion:
                repeat until rounds[vpid][round].flag = sense
                rounds[vpid][round].opponent^ := sense
                exit loop
            dropout:   // impossible
        round := round + 1
    loop                // wakeup
        round := round - 1
        case rounds[vpid][round].role of
            loser:      // impossible
            winner:
                rounds[vpid][round].opponent^ := sense
            bye:        // do nothing
            champion:   // impossible
            dropout:
                exit loop
    sense := not sense
```

A simple scalable tree-based barrier

```
type treenode = record
    parentsense : Boolean
    parentpointer : ^Boolean
    childpointers : array [0..1] of ^Boolean
    havechild : array [0..3] of Boolean
    childnotready : array [0..3] of Boolean
    dummy : Boolean     // pseudo-data

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in shared memory
    // locally accessible to processor vpid
processor private vpid : integer     // a unique virtual processor index
processor private sense : Boolean

// on processor i, sense is initially true
// in nodes[i]:
//    havechild[j] = true if 4*i+j+1 < P; otherwise false
//        NB: there's an off-by-one error in the previous line in the
//        pseudocode in the paper.  Thanks to Kishore Ramachandran for
```

```
//        catching this.
//    parentpointer = &nodes[floor((i-1)/4)].childnotready[(i-1) mod 4],
//       or &dummy if i = 0
//    childpointers[0] = &nodes[2*i+1].parentsense, or &dummy if 2*i+1 >= P
//    childpointers[1] = &nodes[2*i+2].parentsense, or &dummy if 2*i+2 >= P
//    initially childnotready = havechild and parentsense = false

procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready = {false, false, false, false}
        childnotready := havechild    // prepare for next barrier
        parentpointer^ := false       // let parent know I'm ready
        // if not root, wait until my parent signals wakeup
        if vpid != 0
            repeat until parentsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```
