Locks

8 Nov 2010 (Part 2)

Peter Sewell    Jaroslav Ševčík    Tim Harris
Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

Hierarchical locks

Parallel performance
bool testAndSet(bool *b) {
    bool result;
    atomic {
        result = *b;
        *b = TRUE;
    }
}
Test and set

- Suppose two threads use it at once

Thread 1: \(\text{testAndSet}(b)\rightarrow\text{true}\)

Thread 2: \(\text{testAndSet}(b)\rightarrow\text{false}\)
void acquireLock(bool *lock) {
    while (testAndSet(lock)) {
        /* Nothing */
    }
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}
Test and set lock

lock: TRUE

void acquireLock(bool *lock) {
    while (testAndSet(lock)) {
        /* Nothing */
    }
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}
What are the problems here?

- testAndSet implementation causes contention
- Only supports mutual exclusion: not reader-writer locking
- No control over locking policy
- Spinning may waste resources while waiting
Contestion from testAndSet
Multi-core h/w – separate L2

testAndSet(k)

Single-threaded core

L1 cache

L2 cache

Main memory

Single-threaded core

L1 cache

L2 cache
Multi-core h/w – separate L2
Multi-core h/w – separate L2

testAndSet(k)

Single-threaded core

Single-threaded core

L1 cache

L1 cache

L2 cache

L2 cache

Main memory
General problem

- No *logical conflict* between two failed lock acquires
- Cache protocol introduces a *physical conflict*
- For a good algorithm: only introduce physical conflicts if a logical conflict occurs
Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

Hierarchical locks

Parallel performance
Test and test and set lock

void acquireLock(bool *lock) {
    do {
        while (*lock) {
        }
    } while (!testAndSet(lock));
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}

FALSE => lock available
TRUE => lock held

Spin while the lock is held... only do testAndSet when it is clear
Based on Fig 7.4, Herlihy & Shavit, “The Art of Multiprocessor Programming”
void acquireLock(bool *lock) {
    do {
        while (*lock) {
        }
    } while (testAndSet(lock));
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}
Back-off algorithms

1. Start by spinning, watching the lock
2. After an interval \( c \) spin locally for \( s \) (without watching the lock)

What should “\( c \)” be?
What should “\( s \)” be?
Time spent waiting “c”

• Lower values:
  – Less time to build up a set of threads that will stampede

• Higher values:
  – Less likelihood of a delay between a lock being released and a waiting thread noticing
Spinning time “s”

• Lower values:
  – More responsive to the lock becoming available

• Higher values:
  – If the lock doesn’t become available then the thread makes fewer accesses to the shared variable
Methodical approach

• For a given workload and performance model:
  – What is the best that an oracle could do (e.g. given perfect knowledge of lock demands)?
  – How does a practical algorithm compare with this?

• Look for an algorithm with a bound between its performance and that of the oracle

• “Competitive spinning”
Rule of thumb

• Spin for a duration that’s comparable with the shortest back-off interval
• Exponentially increase the per-thread back-off interval (resetting it when the lock is acquired)
• Use a maximum back-off interval that is large enough that waiting threads don’t interfere with the system’s performance
Test-and-set (TAS) locks
TATAS locks & backoff
Queue-based locks
Hierarchical locks
Parallel performance
Queue-based locks

• Lock holders queue up: immediately provides FCFS behavior
• Each spins *locally* on a flag in their queue entry: no remote memory accesses while waiting
• A lock release wakes the next thread directly: no stampede
MCS locks

lock: FALSE

Local flag

QNode 1: FALSE
QNode 2: FALSE
QNode 3: FALSE

Lock identifies tail

Head

Tail
void acquireMCS(mcs *lock, QNode *qn) {
    QNode *prev;
    qn->flag = false;
    qn->next = NULL;
    while (true) {
        prev = lock->tail;
        /*Label 1*/
        if (CAS(&lock->tail, prev, qn)) break;
    }
    if (prev != NULL) {
        prev->next = qn; /*Label 2*/
        while (!qn->flag) { } // Spin
    }
}
```c
void releaseMCS(mcs *lock, QNode *qn) {
    if (lock->tail = qn) {
        if (CAS(&lock->tail, qn, NULL)) return;
    }
    while (qn->next != NULL) {} 
    qn->next->flag = TRUE; 
}
```

If we were at the tail then remove us

Wait for next lock holder to announce themselves; signal them
Test-and-set (TAS) locks
TATAS locks & backoff
Queue-based locks
Hierarchical locks
Parallel performance
Hierarchical locks

Memory bus

Memory

Shared L2 cache

Core 1

Core 2

Core 3

Core 4

Core 5

Core 6

Core 7

Core 8
Hierarchical locks
Hierarchical locks

Pass lock “nearby” if possible

Core 1 -> Core 2
Core 3 -> Core 4

Shared L2 cache

Core 5 -> Core 6
Core 7 -> Core 8

Shared L2 cache

Memory bus

Call this a “cluster” of cores

Memory
Hierarchical TATAS with backoff

```c
void acquireLock(bool *lock) {
  do {
    holder = *lock;
    if (holder != -1) {
      if (holder == MY_CLUSTER) {
        BackOff(SHORT);
      } else {
        BackOff(LONG);
      }
    }
  } while (!CAS(lock, -1, MY_CLUSTER));
}
```

-1 => lock available
n => lock held by cluster n
Hierarchical locks

Avoid this cycle repeating, starving 5 & 7...

Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8

Shared L2 cache
Shared L2 cache

Memory bus

Memory
Hierarchical CLH queue lock

Local queue:

- Lock identifies *tail*

- Flag => successor must wait

- TRUE

- myNode

- myPred

Thread private variables represent links implicitly

Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit
Hierarchical CLH queue lock

Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit
Hierarchical CLH queue lock

Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit
Hierarchical CLH queue lock

Local queue:

myNode  myPred

Global queue:

myNode  myPred

Splice whole list to tail of global queue

Set “Tail When Spliced” flag: next local queue entry will be a new cluster master

Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit
Test-and-set (TAS) locks
TATAS locks & backoff
Queue-based locks
Hierarchical locks
Parallel performance
An aside: is this a better algorithm?

• How fast does it run without contention?
  – Each thread acquires and releases different locks
  – Threads acquire and release the same lock... but not at the same time

• How fast does it run with contention?
  – \( n \) threads trying to acquire the same lock at the same time
  – How does performance scale as \( n \) varies?
An aside: is this a better algorithm?

- Wall-clock time?
- Memory accesses?
- Instruction count?
- Cache lines read/written?
An aside: is this a better algorithm?

Execution time on Machine M1

- Algorithm A
- Algorithm B

Low contention: B is best

High contention: A is best
An aside: is this a better algorithm?

...and on this machine B is best everywhere!