# Scalable coordination

Required reading: Mellor-Crummey and Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, TOCS, Feb 1991.

## Overview

Shared memory machines are bunch of CPUs, sharing physical memory. Typically each processor also mantains a cache (for performance), which introduces the problem of keep caches coherent. If processor 1 writes a memory location whose value processor 2 has cached, then processor 2's cache must be updated in some way. How?

- Bus-based schemes. Any CPU can access "dance with" any memory equally ("dance hall arch"). Use "Snoopy" protocols: Each CPU's cache listens to the memory bus. With write-through architecture, invalidate copy when see a write. Or can have "ownership" scheme with write-back cache (E.g., Pentium cache have MESI bits---modified, exclusive, shared, invalid). If E bit set, CPU caches exclusively and can do write back. But bus places limits on scalability.
- More scalability w. NUMA schemes (non-uniform memory access). Each CPU comes with fast "close" memory. Slower to access memory that is stored with another processor. Use a directory to keep track of who is caching what. For example, processor 0 is responsible for all memory starting with address "000", processor 1 is responsible for all memory starting with "001", etc.
- COMA - cache-only memory architecture. Each CPU has local RAM, treated as cache. Cache lines migrate around to different nodes based on access pattern. Data only lives in cache, no permanent memory location. (These machines aren't too popular any more.)

## Scalable locks

This paper is about cost and scalability of locking; what if you have 10 CPUs waiting for the same lock? For example, what would happen if xv6 runs on an SMP with many processors?

What's the cost of a simple spinning acquire/release? Algorithm 1 *without* the delays, which is like xv6's implementation of acquire and release (xv6 uses XCHG instead of test_and_set):

```
each of the 10 CPUs gets the lock in turn
meanwhile, remaining CPUs in XCHG on lock
lock must be X in cache to run XCHG
  otherwise all might read, then all might write
so bus is busy all the time with XCHGs!
can we avoid constant XCHGs while lock is held?
```

test-and-test-and-set

```
only run expensive TSL if not locked
spin on ordinary load instruction, so cache line is S
acquire(l)
  while(1){
    while(l->locked != 0) { }
    if(TSL(&l->locked) == 0)
      return;
  }
```

suppose 10 CPUs are waiting, let's count cost in total bus transactions

```
CPU1 gets lock in one cycle
  sets lock's cache line to I in other CPUs
9 CPUs each use bus once in XCHG
  then everyone has the line S, so they spin locally
CPU1 release the lock
CPU2 gets the lock in one cycle
8 CPUs each use bus once...
So 10 + 9 + 8 + ... = 50 transactions, O(n^2) in # of CPUs!
Look at "test-and-test-and-set" in Figure 6
```

Can we have *n* CPUs acquire a lock in O(*n*) time?

What is the point of the exponential backoff in Algorithm 1?

```
Does it buy us O(n) time for n acquires?
Is there anything wrong with it?
may not be fair
exponential backoff may increase delay after release
```

What's the point of the ticket locks, Algorithm 2?

```
one interlocked instruction to get my ticket number
then I spin on now_serving with ordinary load
release() just increments now_serving
```

why is that good?

```
+ fair
+ no exponential backoff overshoot
+ no spinning on
```

but what's the cost, in bus transactions?

```
while lock is held, now_serving is S in all caches
release makes it I in all caches
then each waiters uses a bus transaction to get new value
so still O(n^2)
```

What's the point of the array-based queuing locks, Algorithm 3?

```
    a lock has an array of "slots"
    waiter allocates a slot, spins on that slot
    release wakes up just next slot
  so O(n) bus transactions to get through n waiters: good!
  anderson lines in Figure 4 and 6 are flat-ish
    they only go up because lock data structures protected by simpler
lock
  but O(n) space *per lock*!
```

Algorithm 5 (MCS), the new algorithm of the paper, uses compare_and_swap:

```
int compare_and_swap(addr, v1, v2) {
  int ret = 0;
  // stop all memory activity and ignore interrupts
  if (*addr == v1) {
    *addr = v2;
    ret = 1;
  }
  // resume other memory activity and take interrupts
  return ret;
}
```

What's the point of the MCS lock, Algorithm 5?

```
  constant space per lock, rather than O(n)
  one "qnode" per thread, used for whatever lock it's waiting for
  lock holder's qnode points to start of list
  lock variable points to end of list
  acquire adds your qnode to end of list
    then you spin on your own qnode
  release wakes up next qnode
```

## Wait-free or non-blocking data structures

The previous implementations all block threads when there is contention for a lock. Other atomic hardware operations allows one to build implementation wait-free data structures. For example, one can make an insert of an element in a shared list that don't block a thread. Such versions are called wait free.

A linked list with locks is as follows:

```
Lock list_lock;

insert(int x) {
  element *n = new Element;
  n->x = x;

  acquire(&list_lock);
  n->next = list;
  list = n;
  release(&list_lock);
}
```

A wait-free implementation is as follows:

```
insert (int x) {
  element *n = new Element;
  n->x = x;
  do {
     n->next = list;
  } while (compare_and_swap (&list, n->next, n) == 0);
}
```

How many bus transactions with 10 CPUs inserting one element in the list? Could you do better?

The paper by Fraser and Harris compares lock-based implementations versus corresponding non-blocking implementations of a number of data structures.

It is not possible to make every operation wait-free, and there are times we will need an implementation of acquire and release. research on non-blocking data structures is active; the last word isn't said on this topic yet.