# Locking

## The lock abstraction

We'd like a general tool to help programmers force serialization. There are a number of things we'd like to be able to express for lists:

1. It's not enough to serialize just calls to insert(); if there's also delete(), we want to prevent a delete() from running at the same time as an insert().
2. Serializing *all* calls to insert() is too much: it's enough to serialize just calls that refer to the same list.
3. We don't need to serialize all of insert(), just lines A and B.

We're looking for an abstraction that gives the programmer this kind of control. The reason to allow concurrency when possible (items 2 and 3) is to increase performance.

While there are many coordination abstractions, locking is among the most common. A lock is an object with two operations: acquire(L) and release(L). The lock has state: it is either locked or not locked. A call to acquire(L) waits until L is not locked, then changes its state to locked and returns. A call to release(L) marks L as not locked. So when you have a data structure on which operations should proceed one at a time (i.e. not be interleaved), associate a lock object L with the data structure and bracket the operations with acquire(L) and release(L).

```
Lock list_lock;

insert(int data){
  List *l = new List;
  l->data = data;

  acquire(&list_lock);

  l->next = list ; // A
  list = l;        // B

  release(&list_lock);
}
```

How can we implement acquire() and release()? Here is a version that **DOES NOT WORK**:

```
struct Lock {
  int locked;
}

void broken_acquire(Lock *lock) {
  while(1){
    if(lock->locked == 0){ // C
      lock->locked = 1;    // D
      break;
    }
  }
}

void release (Lock *lock) {
  lock->locked = 0;
}
```

What's the problem? Two acquire()s on the same lock on different CPUs might both execute line C, and then both execute D. Then both will think they have acquired the lock. This is the same kind of race we were trying to eliminate in insert(). But we have made a little progress: now we only need a way to prevent interleaving in one place (acquire()), not for many arbitrary complex sequences of code (e.g. A and B in insert()).

## Atomic instructions

We're in luck: most CPUs provide "atomic instructions" that are equivalent to the combination of lines C and D and prevent interleaving from other CPUs.

For example, the x86 `xchg %eax, addr` instruction does the following:

1. freeze other CPUs' memory activity for address addr
2. temp := *addr
3. *addr := %eax
4. %eax = temp
5. un-freeze other memory activity

The CPUs and memory system cooperate to ensure that no other operations on this memory location occur between when `xchg` reads and when it writes.

We can use `xchg` to make a correct acquire() and release() (you can find the actual syntax in the xv6 source):

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(Lock *lock){
  while(1){
    if(xchg(&lock->locked, 1) == 0)
      break;
  }
}

void release(Lock *lock){
  lock->locked = 0;
}
```

Why does this work? If two CPUs execute `xchg` at the same time, the hardware ensures that one `xchg` does both its steps before the other one starts. So we can say "the first `xchg`" and "the second `xchg`". The first `xchg` will read a 0 and set lock->locked to 1 and the acquire() will return. The second `xchg` will then see lock->locked as 1, and the acquire will loop.

This style of lock is called a spin-lock, since acquire() stays in a loop while waiting for the lock.

## Spinning vs blocking

Spin-locks are good when protecting short operations: increment a counter, search in i-node cache, allocate a free buffer. In these cases acquire() won't waste much CPU time spinning even if another CPU holds the lock.

Spin locks are not so great for code that has to wait for events that might take a long time. For example, it would not be good for the disk reading code to get a spin-lock on the disk hardware, start a read, wait for the disk to finish reading the data, and then release the lock. The disk often takes 10s of milliseconds (millions of instructions) to complete an operation. Spinning wastes the CPU; if another process is waiting to execute, it would be better to run that process until the disk signals it is finished by causing an interrupt.

xv6 has sleep() and wakeup() primitives that are better for processes that need to wait for I/O.