Locks and modularity

When designing a system we desire clean abstractions and good modularity. We'd like a caller not to have to know how a callee implements a particular function. We'd like to hide locking behavior in this way -- for example, it would be nice if the lock on each file-system inode was the private business of methods on inodes.

It turns out it's hard to use locks in a modular way. Two problems come up that require non-modular global knowledge: multi-module invariants and deadlock.

First, multi-module invariants. You could imagine the code implementing directories exporting various operations like `dir_lookup(d, name)`, `dir_add(d, name, inumber)`, and `dir_del(d, name)`. These directory operations would each acquire the lock on `d`, do their work, and release the lock. Then higher-level code could implement operations like moving a file from one directory to another (one case of the UNIX `rename()` system call):

```
move(olddir, oldname, newdir, newname){
  inumber = dir_lookup(olddir, oldname)
  dir_del(olddir, oldname)
  dir_add(newdir, newname, inumber)
}
```

This code is pleasingly simple, but has a number of problems. One of them is that there's a period of time when the file is visible in neither directory. Fixing that, sadly, seems to require that the directory locks *not* be hidden inside the `dir_` operations, thus:

```
move(olddir, oldname, newdir, newname){
  acquire(olddir.lock)
  acquire(newdir.lock)
  inumber = dir_lookup(olddir, oldname)
  dir_del(olddir, oldname)
  dir_add(newdir, newname, inumber)
  release(newdir.lock)
  release(olddir.lock)
}
```

The above code is ugly in that it exposes the implementation of directories to `move()`, but (if all you have is locks) you have to do it this way.

The second big problem is deadlock. Notice that `move()` holds two locks. What would happen if one process called `move(dirA, ..., dir B, ...)` while another process called `move(dirB, ..., dir A, ...)`?

In an operating system the usual solution is to avoid deadlocks by establishing an order on all locks and making sure that every thread acquires its locks in that order. For the file system the rule might be to lock the directory with the lowest inode first. Picking and obeying the order on *all* locks requires that modules make public their locking behavior, and requires them to know about other module's locking. This can be painful and error-prone.

Fine-grained locks usually make both of these problems worse.

Some terms

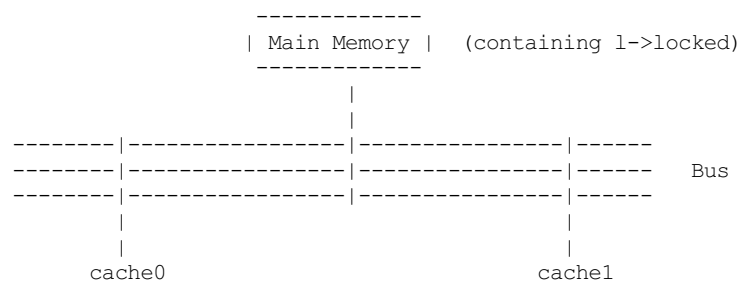
Locks are a way to implement "mutual exclusion", which is the property that only one CPU at a time can be executing a certain piece of code. This is also called "isolation atomicity".

The code between an acquire() and a release() is sometimes called a "critical section."

The more general idea of ensuring one thread waits for another when neccessary for correctness is called "coordination." People have invented a huge number of coordination primitives; locks are just one example, which we use because they are simple. We'll see some more coordination primitives next.

Spin Lock Subtleties

The `xchg R,M` instruction is expensive because it usually involves a bus transaction, to ensure that the read and write both happen atomically. Here is a computer system diagram to explain this better:



Each CPU usually has a private L1 cache. Consistency between the caches is maintained using a "cache coherence protocol", whereby the system ensures that each memory location has only one valid value at any time.

Thus if CPU0 access location X, a 100 times (without any intervening access to X by CPU1), the first access will result in a bus transaction (whereby X will be fetched either from the main memory, or from CPU1's cache through the cache coherence protocol). All the other 99 accesses will be served locally from cache0, without incurring a bus transaction. In a multiprocessor system, the bus often becomes a performance bottleneck (especially if the number of CPUs is high), and so reducing the number of bus transactions is an important optimization.

In our implementation of `acquire()` and `release()`, we use the atomic `xchg` instruction that needs to perform two memory accesses (one read and one write) in an atomic fashion. This can only be achieved by a bus transaction, because all other processors need to be informed that the two memory accesses need to be done atomically. Hence, if a CPU is spin-waiting inside a lock acquire function, it is constantly making bus transactions, which can severely affect performance.

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1);
}
void release(struct lock *l) {
  l->locked = 0;
}
```

Instead, it may be better to write the code like this:

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  l->locked = 0;
}
```

In this case, while waiting, we use a regular memory read instruction (in the inner loop) to test the current value of `l->locked`. For the time that `l->locked` is not changed by the other CPU, `l->locked` will get cached into cache0, and the accesses by a CPU will be only to its local cache (thus avoiding bus transactions). If the local cache reports that the value has changed (which means another CPU must have changed its value), this code again tries the `xchg` instruction, which may succeed this time. If it fails again, we again go into the inner waiting loop. This is a more efficient method of waiting as it reduces bus transactions significantly. Modern processors provide special instructions (e.g., `pause` on x86) to make this waiting operation even more efficient.

Effect of Compiler/Architecture Optimizations

Compiler optimizations typically do not work well with code that can be executed concurrently. Using locks, we have ensured that the only code that can execute concurrently are the `acquire()` and `release()` functions. Let's see what could happen if the compiler tries to optimize our `acquire()` and `release()` implementation.

Register allocation of variables

Register allocation of a variable involves reading the variable from memory into a register, and then performing the operations on the register locally, before writing the register back to memory. For example, in the following code, the variable `a` gets register allocated to register `R`.

```
a = 1;
b = a + 2;
a = a + 1;
```

may get compiled to

```
load  a, Reg     //load 'a' from memory to register
b = Reg + 2
Reg = Reg + 1
store Reg, a     //store current value of 'a' from register to memory
```

Intelligent register allocation reduces the number of memory accesses, and thus reduces memory accesses. But what would happen if the `l->locked` variable in our `acquire()` function gets register allocated. It will result in an infinite loop inside acquire, causing a deadlock.

Because compiler optimizations assume sequential code, they do not play well with code that could be executed concurrently. Languages typically provide ways to tell the compiler to not optimize accesses to certain variables (for example, the `volatile` keyword in C). In this case, we should either write the waiting loop in assembly, or use the volatile keyword appropriately to ensure that the `l->locked` variable does not get register allocated.

Reordering of Memory Accesses

Usually, compilers are free to reorder accesses to "independent" memory locations. For a compiler, two locations are independent, if they are different. For example, it is perfectly valid for a compiler to invert the order of accesses to variables `a` and `b` like this:

```
a = 2;
b = 3;
```

becomes

```
b = 3;
a = 2;
```

For a compiler which only looks at sequential semantics, this is a valid optimization. Similarly, even if the compiler does not reorder the instructions, the hardware is free to reorder the memory accesses at runtime. For example, if the access to variable `"a"` is a cache miss, while the access to variable `"b"` is a cache hit, the access to `"b"` will complete first. This is true for most modern processors, and this behaviour is called "out-of-order" execution.

In our lock implementation, if the access to a shared variable gets reordered with the access to the lock variable (`l->locked`), our semantics get violated. For example,

```
acquire(l);
hits++;
release(l);
```

may become equivalent to (either at compile-time or at runtime)

```
acquire(l);
release(l);
hits++;
```

To avoid this, it is possible to tell the compiler and the hardware to not reorder instructions. Many architectures that support out-of-order execution, also provide "`fence`" instructions, which can be used to instruct the hardware to not reorder memory accesses across the fence instruction. On the x86 architecture, the locked `xchg` instruction implicitly also acts as a fence, so our `acquire` implementation is correct even in the presence of memory access reordering. However, we need to explicitly insert a `fence` instruction in our `release()` implementation like this:

```
void acquire(struct lock *l) {
  Reg = 1;
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  fence;   //assembly instruction
  l->locked = 0;
}
```

Spinlock implementations inside the kernel

For a uniprocessor kernel, spin locks are not needed. Instead, mutual exclusion can be ensured by clearing (and re-enabling) insterrupts around before (and after) the critical section.

For a multiprocessor kernel, spin locks as implemented above suffice for ensuring mutual exclusion among different CPUs. However, this still does not protect a CPU from its own interrupt handler. For example, if an interrupt can occur within a critical section, and the interrupt handler could access the shared data (which was also being accessed within the critical section), mutual exclusion would get violated. On xv6, this is true for the process table `ptable`, for example, where the timer handler may need to inspect the `ptable` and may find it in an inconsistent state, if the timer interrupt occurred in the middle of an access to the `ptable`. Worse, if the timer handler also tries to acquire the `ptable.lock`, it would result in a deadlock.

To deal with this, xv6's spinlock also disables the interrupts on a call to `acquire`, and reenables them on a call to `release`. To allow a thread to acquire multiple locks simultaneously, the interrupts are enabled and disabled in a recursive manner, as follows:

```
void acquire(struct lock *l) {
  Reg = 1;
  pushcli();
  while (xchg(Reg, &l->locked) == 1) {
    while (l->locked == 1);
  }
}
void release(struct lock *l) {
  fence;   //assembly instruction
  l->locked = 0;
  popcli();
}
```

The `pushcli()` function uses the `cli` instruction internally to disable interrupts. The `popcli()` function uses the `sti` instruction internally to re-enable interrupts. They also maintain a per-CPU counter `cpu->ncli` to correctly handle nested calls to `pushcli()` and `popcli()`.

```
void pushcli(void) {
  if (cpu->ncli == 0) {
    cli;
  }
  cpu->ncli++;
}
void popcli(void) {
  cpu->ncli--;
  if (cpu->ncli == 0) {
    sti;
  }
}
```

Spinlock implementations in the user-mode

In the usermode, spinlocks work exactly like they work in kernel-mode, except that they do not need to disable and enable interrupts. Kernel's interrupt handlers are not expected to touch user data, so disabling interrupts is not required. Notice that the `xchg` instruction is an unprivileged instruction, and has identical semantics in both user and kernel modes.

Blocking Locks

Blocking locks involve changing the status of the current thread from RUNNING to BLOCKED in the process table (or the list of PCBs, depending on how the processes are maintained). For xv6, the global `ptable` array is the process table structure. A kernel thread that needs to wait on a lock `acquire`, will set its status to "BLOCKED on lock l", and call the scheduler. Similarly, when a thread calls `release(l)`, it scans the `ptable` and changes the status of one of the blocked processes (if any), blocked on `l` from BLOCKED to READY.

Notice that the accesses to the `ptable` also need to be mutually exclusive. This is done using a spinlock (`ptable.lock` in the case of xv6). Hence, a blocking lock typically uses a spinlock internally. Of course, the spinlock is released, as soon as the ptable is updated.

Blocking locks at user-level

At the user-level, blocking locks need to suspend a thread. If the threads are kernel-level threads, this involves making a system call to tell the kernel to block on a lock acquire. Similarly, release involves making a system call to unblock one of the threads blocked on that lock.

If the threads are user-level threads however, all this can be done entirely in userspace, without any kernel involvement. For user-level threads, the `ptable` itself will be maintained (and manipulated) at the user-level.

Locking Variations

Recursive locks

Our current lock abstraction does not allow a code path which acquires the same lock twice --- execution of this code path will cause a deadlock. For example, as we discussed previously, if an interrupt occurs in the middle of a critical section, and the interrupt handler tries to acquire an already-held lock, a deadlock would result.

One proposal to solve this issue is to allow the same thread to acquire a lock multiple times, but still disallow different threads to acquire the lock at the same time. Here is an example implementation of recursive locks, which uses regular locks internally.

```
void recursive_acquire(struct rlock *rl) {
  if (rl->owner == cur_thread) {
    rl->count++;
  } else {
    acquire(&rl->lock);
    rl->count = 0;
    rl->owner = cur_thread;
  }
}
void recursive_release(struct rlock *rl) {
  rl->count--;
  if (rl->count == 0) {
    release(&rl->lock);
    rl->owner = -1;
  }
}
```

Usually, recursive locks are a bad idea. Most programmers maintain the invariant that a critical section always begins in a consistent state, i.e., immediately after successfully returning from a lock acquire, the state of the system is consistent. Similarly, they maintain the invariant that the state of the system is left in a consistent state on a lock release, i.e., the state is consistent just before a call to `release()`. (By consistency, we mean that certain invariants hold about the program state. For example, in our bank account program, consistency may mean that the total money in the

bank is constant).

Recursive locks encourage the programmer to allow a critical section to begin in an inconsistent state. Consider a function `foo()` which acquires a lock `l`, and then calls another function `bar()`. The call to the function `bar()` may be synchronous (i.e., the programmer has a call chain in her program that eventually leads to `bar` from `foo`), or asynchronous (e.g., if `bar()` is called within an interrupt handler, and the interrupt is received within `foo()`). In either case, `bar()` may be assuming that the state is consistent when it begins its critical section. However, because the function was called in the middle of the critical section of `foo()`, `bar` may unexpectedly see inconsistent state.

```
foo() {
  acquire();
  ....
  bar();   //at this point, state is inconsistent
  ....
  release();
}

bar() {
  acquire();
  ....      // assumes that if acquire succeeds, the state here is consistent.
  ....
  release();
}
```

Because recursive locks encourage such bugs, they are not considered very useful.

Try locks

Another locking variation is a try lock, where the `acquire()` function returns a SUCCESS or a FAILURE value, depending on whether the acquisition was successful or not. Here is a sample implementation of a trylock.

```
bool try_acquire(struct trylock *tl) {
  Reg = 1;
  xchg(Reg, &tl->locked);
  if (Reg == 1) {
    return false;  //FAILURE
  } else {
    return true;   //SUCCESS
  }
}
void try_release(struct trylock *tl) {
  fence;
  tl->locked = 0;
}
```

The caller of `try_acquire` may decide its action depending on the return value. A regular acquire can be implemented trivially by calling `try_acquire` in a loop till it succeeds. However, `try_acquire` gives the flexibility to the caller to do something else, if the lock is not currently available.