Locking

Required reading: spinlock.c

Why coordinate?

Mutual-exclusion coordination is an important topic in operating systems, because many operating systems run on multiprocessors. Coordination techniques protect variables that are shared among multiple threads and updated concurrently. These techniques allow programmers to implement atomic sections so that one thread can safely update the shared variables without having to worry that another thread intervening. For example, processes in xv6 may run concurrently on different processors and in kernel-mode share kernel data structures. We must ensure that these updates happen correctly.

List and insert example:

```
struct List {
  int data;
  struct List *next;
};

List *list = 0;

insert(int data) {
  List *l = new List;
  l->data = data;
  l->next = list;  // A
  list = l;        // B
}
```

What needs to be atomic? The two statements labeled A and B should always be executed together, as an indivisible fragment of code. If two processors execute A and B interleaved, then we end up with an incorrect list. To see that this is the case, draw out the list after the sequence A1 (statement executed A by processor 1), A2 (statement A executed by processor 2), B2, and B1.

How could this erroneous sequence happen? The varilable *list* lives in physical memory shared among multiple processors, connected by a bus. The accesses to the shared memory will be ordered in some total order by the bus/memory system. If the programmer doesn't coordinate the execution of the statements A and B, any order can happen, including the erroneous one.

The erroneous case is called a race condition. The problem with races is that they are difficult to reproduce. For example, if you put print statements in to debug the incorrect behavior, you might change the time and the race might not happen anymore.

Atomic instructions

The programmer must be able express that A and B should be executed as single atomic instruction. We generally use a concept like locks to mark an atomic region, acquiring the lock at the beginning of the section and releasing it at the end:

```
void acquire(int *lock) {
   while (TSL(lock) != 0) ;
}

void release (int *lock) {
  *lock = 0;
}
```

Acquire and release, of course, need to be atomic too, which can, for example, be done with a hardware atomic TSL (try-set-lock) instruction:

The semantics of TSL are:

```
   R <- [mem]   // load content of mem into register R
   [mem] <- 1   // store 1 in mem.
```

In a harware implementation, the bus arbiter guarantees that both the load and store are executed without any other load/stores coming in between.

We can use locks to implement an atomic insert, or we can use TSL directly:

```
int insert_lock = 0;

insert(int data) {

  /* acquire the lock: */
  while(TSL(&insert_lock) != 0)
    ;

  /* critical section: */
  List *l = new List;
  l->data = data;
  l->next = list;
  list = l;

  /* release the lock: */
  insert_lock = 0;
}
```

It is the programmer's job to make sure that locks are respected. If a programmer writes another function that manipulates the list, the programmer must must make sure that the new functions acquires and releases the appropriate locks. If the programmer doesn't, race conditions occur.

This code assumes that stores commit to memory in program order and that all stores by other processors started before insert got the lock are observable by this processor. That is, after the other processor released a lock, all the previous stores are committed to memory. If a processor executes instructions out of order, this assumption won't hold and we must, for example, a barrier instruction that makes the assumption true.

Example: Locking on x86

Here is one way we can implement acquire and release using the x86 xchgl instruction:

```
struct Lock {
  unsigned int locked;
};

acquire(Lock *lck) {
  while(TSL(&(lck->locked)) != 0)
    ;
}

release(Lock *lck) {
  lck->locked = 0;
}

int
TSL(int *addr)
{
  register int content = 1;
  // xchgl content, *addr
  // xchgl exchanges the values of its two operands, while
  // locking the memory bus to exclude other operations.
  asm volatile ("xchgl %0,%1" :
                "=r" (content),
                "=m" (*addr) :
                "0" (content),
                "m" (*addr));
  return(content);
}
```

the instruction "XCHG %eax, (content)" works as follows:

1. freeze other CPUs' memory activity
2. temp := content
3. content := %eax
4. %eax := temp
5. un-freeze other CPUs

steps 1 and 5 make XCHG special: it is "locked" special signal lines on the inter-CPU bus, bus arbitration

This implementation doesn't scale to a large number of processors; in a later lecture we will see how we could do better.

Lock granularity

Release/acquire is ideal for short atomic sections: increment a counter, search in i-node cache, allocate a free buffer.

What are spin locks not so great for? Long atomic sections may waste waiters' CPU time and it is to sleep while holding locks. In xv6 we try to avoid long atomic sections by carefully coding (can you find an example?). xv6 doesn't release the processor when holding a lock, but has an additional set of coordination primitives (sleep and wakeup), which we will study later.

My list_lock protects all lists; inserts to different lists are blocked. A lock per list would waste less time spinning so you might want "fine-grained" locks, one for every object BUT acquire/release are expensive (500 cycles on my 3 ghz machine) because they need to talk off-chip.

Also, "correctness" is not that simple with fine-grained locks if need to maintain global invariants; e.g., "every buffer must be on exactly one of free list and device list". Per-list locks are irrelevant for this invariant. So you might want "large-grained", which reduces overhead but reduces concurrency.

This tension is hard to get right. One often starts out with "large-grained locks" and measures the performance of the system on some workloads. When more concurrency is desired (to get better performance), an implementor may switch to a more fine-grained scheme. Operating system designers fiddle with this all the time.

Recursive locks and modularity

When designing a system we desire clean abstractions and good modularity. We like a caller not have to know about how a callee implements a particul functions. Locks make achieving modularity more complicated. For example, what to do when the caller holds a lock, then calls a function, which also needs to the lock to perform its job.

There are no transparent solutions that allow the caller and callee to be unaware of which lokcs they use. One transparent, but unsatisfactory option is recursive locks: If a callee asks for a lock that its caller has, then we allow the callee to proceed. Unfortunately, this solution is not ideal either.

Consider the following. If lock x protects the internals of some struct foo, then if the caller acquires lock x, it know that the internals of foo are in a sane state and it can fiddle with them. And then the caller must restore them to a sane state before release lock x, but until then anything goes.

This assumption doesn't hold with recursive locking. After acquiring lock x, the acquirer knows that either it is the first to get this lock, in which case the internals are in a sane state, or maybe some caller holds the lock and has messed up the internals and didn't realize when calling the callee that it was going to try to look at them too. So the fact that a function acquired the lock x doesn't guarantee anything at all. In short, locks protect against callers and callees just as much as they protect against other threads.

Since transparent solutions aren't ideal, it is better to consider locks part of the function specification. The programmer must arrange that a caller doesn't invoke another function while holding a lock that the callee also needs.

Locking in xv6

xv6 runs on a multiprocessor and is programmed to allow multiple threads of computation to run concurrently. In xv6 an interrupt might run on one processor and a process in kernel mode may run on another processor, sharing a kernel data structure with the interrupt routing. xv6 uses locks, implemented using an atomic instruction, to coordinate concurrent activities.

Let's check out why xv6 needs locks by following what happens when we start a second processor:

- 1516: mp_init (called from main0)
- 1606: mp_startthem (called from main0)
- 1302: mpmain
- 2208: scheduler.
  Now we have several processors invoking the scheduler function. xv6 better ensure that multiple processors don't run the same process! does it?
  Yes, if multiple schedulers run concurrently, only one will acquire proc_table_lock, and proceed looking for a runnable process. if it finds a process, it will mark it running, longjmps to it, and the process will release proc_table_lock. the next instance of scheduler will skip this entry, because it is marked running, and look for another runnable process.

Why hold proc_table_lock during a context switch? It protects p->state; the process has to hold some lock to avoid a race with wakeup() and yield(), as we will see in the next lectures.

Why not a lock per proc entry? It might be expensive in in whole table scans (in wait, wakeup, scheduler). proc_table_lock also protects some larger invariants, for example it might be hard to get proc_wait() right with just per entry locks. Right now the check to see if there are any exited children and the sleep are atomic -- but that would be hard with per entry locks. One could have both, but that would probably be neither clean nor fast.

Of course, there is only processor searching the proc table if acquire is implemented correctly. Let's check out acquire in spinlock.c:

- 1807: no recursive locks!

- 1811: why disable interrupts on the current processor? (if interrupt code itself tries to take a held lock, xv6 will deadlock; the panic will fire on 1808.)
    - can a process on a processor hold multiple locks?
- 1814: the (hopefully) atomic instruction.
    - see sheet 4, line 0468.
- 1819: make sure that stores issued on other processors before we got the lock are observed by this processor. these may be stores to the shared data structure that is protected by the lock.

Locking in JOS

JOS is meant to run on single-CPU machines, and the plan can be simple. The simple plan is disabling/enabling interrupts in the kernel (IF flags in the EFLAGS register). Thus, in the kernel, threads release the processors only when they want to and can ensure that they don't release the processor during a critical section.

In user mode, JOS runs with interrupts enabled, but Unix user applications don't share data structures. The data structures that must be protected, however, are the ones shared in the library operating system (e.g., pipes). In JOS we will use special-case solutions, as you will find out in lab 6. For example, to implement pipe we will assume there is one reader and one writer. The reader and writer never update each other's variables; they only read each other's variables. Carefully programming using this rule we can avoid races.