CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture 7: Implementing Mutual Exclusion

7.0 Main Point:

Hardware support for synchronization

Building higher-level synchronization programming abstractions on top of the hardware support.

7.1 The Big Picture

The abstraction of threads is good, but concurrent threads sharing state is still too complicated (think of the "too much milk" example). Implementing a concurrent program directly with loads and stores would be tricky and error-prone.

So we'd like to provide a synchronization abstraction that hides/manages most of the complexity and puts the burden of coordinating multiple activities on the OS instead of the programmer – Give the programmer higher level operations, such as locks.

In this lecture, we'll explore how one might implement higher-level operations on top of the atomic operations provided by hardware.

In the next lecture, we'll explore what higher-level primitives make it easiest to write correct concurrent programs.

	concurrent programs			
High level atomic operations (API)	locks	semaphores	monitors	send&receive
Low level atomic operations (hardware)	load/store	interrupt disable		test&set

Relationship among synchronization abstractions

7.2 Ways of implementing locks

All require some level of hardware support.

7.2.1 Atomic memory load and store

See too much milk lecture!

7.2.2 Directly implement locks and context switches in hardware

Makes hardware slow! One has to be careful not to slow down the common case in order to speed up a special case.

7.2.3 Disable interrupts (uniprocessor only)

Two ways for dispatcher to get control:

- internal events – thread does something to relinquish the CPU
- external events – interrupts cause dispatcher to take CPU away

On a uniprocessor, an operation will be atomic as long as a context switch does not occur in the middle of the operation. Need to prevent both internal and external events. Preventing internal events is easy (although virtual memory makes it a bit tricky).

Prevent external events by disabling interrupts, in effect, telling the hardware to delay handling of external events until after we're done with the atomic operation.

7.2.3.1 A flawed, but very simple solution

Why not do the following:

```
Lock::Acquire() { disable interrupts;}
Lock::Release() { enable interrupts;}
```

1. Need to support synchronization operations in user-level code. Kernel can't allow user code to get control with interrupts disabled (might never give CPU back!).
2. Real-time systems need to guarantee how long it takes to respond to interrupts, but critical sections can be arbitrarily long. Thus, one should leave interrupts off for shortest time possible.
3. Simple solution might work for locks, but wouldn't work for more complex primitives, such as semaphores or condition variables.
   •

7.2.3.2 Implementing locks by disabling interrupts

Key idea: maintain a lock variable and impose mutual exclusion only on the operations of testing and setting that variable.

```
class Lock {
      int value = FREE;
}

Lock::Acquire() {
      Disable interrupts;
      if (value == BUSY) {
```

```
            Put on queue of threads waiting for lock
            Go to sleep
            // Enable interrupts? See comments below
        } else {
            value = BUSY;
        }
        Enable interrupts;
    }

    Lock::Release()
        Disable interrupts;
        If anyone on wait queue {
            Take a waiting thread off wait queue
            Put it at the front of the ready queue
        } else {
            value = FREE;
        }
        Enable interrupts;
    }
```

Why do we need to disable interrupts at all? Otherwise, one thread could be trying to acquire the lock, and could get interrupted between checking and setting the lock value, so two threads could think that they both have the lock.

By disabling interrupts, the check and set operations occur without any other thread having the chance to execute in the middle.

When does Acquire re-enable interrupts in going to sleep?

Before putting the thread on the wait queue?

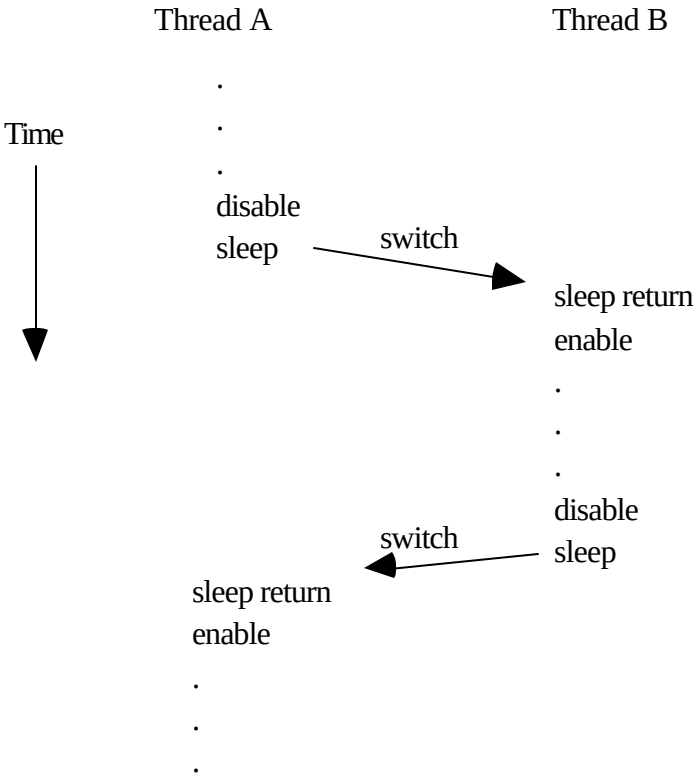
Then Release can check the queue, and not wake the thread up.

After putting the thread on the wait queue, but before going to sleep?

Then Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep! It will go to sleep, missing the wakeup from Release.

To fix this, in Nachos, interrupts are disabled when you call Thread::Sleep; it is the responsibility of the next thread to run to re-enable interrupts.

When the sleeping thread wakes up, it returns from Thread::Sleep back to Acquire. Interrupts are still disabled, so turn on interrupts.



Interrupt disable and enable pattern across context switches

An important point about structuring code: If you look at the Nachos code you will see lots of comments about the assumptions made concerning when interrupts are disabled.

This is an example of where modifications to and assumptions about program state can't be localized within a small body of code. When that's the case you have a very good chance that eventually your program will "acquire" bugs: as people modify the code they may forget or ignore the assumptions being made and end up invalidating the assumptions.

Can you think of other examples where this will be a concern?

> What about acquiring and releasing locks in the presence of C++ exception exits out of a procedure?

7.2.4 Atomic read-modify-write instructions

On a multiprocessor, interrupt disable doesn't provide atomicity. It stops context switches from occurring on that CPU, but it doesn't stop the other CPUs from entering the critical section.

One could provide support to disable interrupts on all CPUs, but that would be expensive: stopping everyone else, regardless of what each CPU is doing.

Instead, every modern processor architecture provides some kind of atomic read-modify-write instruction. These instructions **atomically** read a value from memory into a register, and write a new value. The hardware is responsible for implementing this correctly on both uniprocessors (not too hard) and multiprocessors (requires special hooks in the multiprocessor cache coherence strategy).

Unlike disabling interrupts, this can be used on both uniprocessors and multiprocessors.

7.2.4.1 Examples of read-modify-write instructions:

- **test&set** (most architectures) – read value, write 1 back to memory
- **exchange** (x86) – swaps value between register and memory
- **compare&swap** (68000) – read value, if value matches register, do exchange
- **load linked and conditional store** (R4000, Alpha) – designed to fit better with load/store architecture (speculative computation). Read value in one instruction, do some operations, when store occurs, check if value has been modified in the meantime. If not, ok. If it has changed, abort, and jump back to start.

7.2.4.2 Implementing locks with test&set

Test&set reads location, sets it to 1, and returns old value.

Another flawed, but simple solution:

Initially, `lock value = 0;`

```
Lock::Acquire
      while (test&set(value) == 1) //while BUSY
            ;

Lock::Release
      value = 0;
```

If lock is free, test&set reads 0 and sets value to 1, so lock is now busy. It returns 0, so Acquire completes. If lock is busy, test&set reads 1 and sets value to 1 (no change), so lock stays busy, and Acquire will loop.

Busy-waiting: thread consumes CPU cycles while it is waiting.

Although the machine can at least field interrupts, this is inefficient and could cause problems if threads can have different priorities. It's inefficient because the busy-waiting thread will consume its time slice doing nothing useful (waiting threads slow the thread holding the lock!). If the busy-waiting thread has higher priority than the thread holding the lock, the timer will go off, but (depending on the scheduling policy), the lower priority thread might never run!

7.2.4.3 Locks using test&set, with minimal busy-waiting

How might we implement locks with test&set, without busy waiting? Turns out you can't, but you can minimize busy waiting. Idea: only busy-wait to atomically check lock value; if lock is busy, give up CPU.

Use a guard on the lock itself (multiple layers of critical sections!)

Waiter gives up the processor so that Release can go forward more quickly:

```
Lock::Acquire()
      while (test&set(guard)) // Short busy-wait time
            ;
      if (value == BUSY) {
            Put on queue of threads waiting for lock
            Go to sleep & set guard to 0
      } else {
            value = BUSY;
            guard = 0;
      }
}

Lock::Release()
      while (test&set(guard))
            ;
      if anyone on wait queue {
            take a waiting thread off
            put it at the front of the ready queue
      } else {
            value = FREE;
      }
      guard = 0;
}
```

Notice that sleep has to be sure to reset the guard variable. Why can't we do it just before or just after the sleep?

7.3 Summary

Load/store, disabling and enabling interrupts, and atomic read-modify-write instructions, are all ways that we can implement higher level atomic operations.