Lecture 6: Implementing Mutual Exclusion

6.0 Main Point:

Hardware support for synchronization
How to build higher-level synchronization abstractions on top of the hardware support.

6.1 The Big Picture

	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

Too much milk example showed that implementing a concurrent program directly with loads and stores would be tricky and error-prone. Instead, a programmer is going to want to use higher level operations, such as locks.

First, how do we implement these higher level operations?

Next, what higher-level primitives make it easiest to write correct concurrent programs?

6.2 Ways of implementing locks

6.2.1 Atomic memory load and store

See too much milk lecture!

6.2.2 Directly implement locks and context switches in hardware

Implemented in the Intel 432. Makes hardware slow!

6.2.3 Disable interrupts (uniprocessor only)

Two ways for dispatcher to get control:
internal events -- thread does something to relinquish the CPU
external events -- interrrupts 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.

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.

6.2.3.1 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. I/O devices often need real-time response (within some time bound). But critical sections can be arbitrarily long. Thus, leave interrupts off for shortest time possible.

6.2.3.2 Another flawed solution

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

```
Lock::Acquire() {
   Disable interrupts;
   while (value != FREE) {
      Enable interrupts; // allow interrupts
      Disable interrupts;
      }
   value = BUSY;
   Enable interrupts;

Lock::Release()
   Disable interrupts;
   value = FREE;
   Enable interrupts;
```

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

6.2.3.3 Locks using interrupt disable, without busy-waiting

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

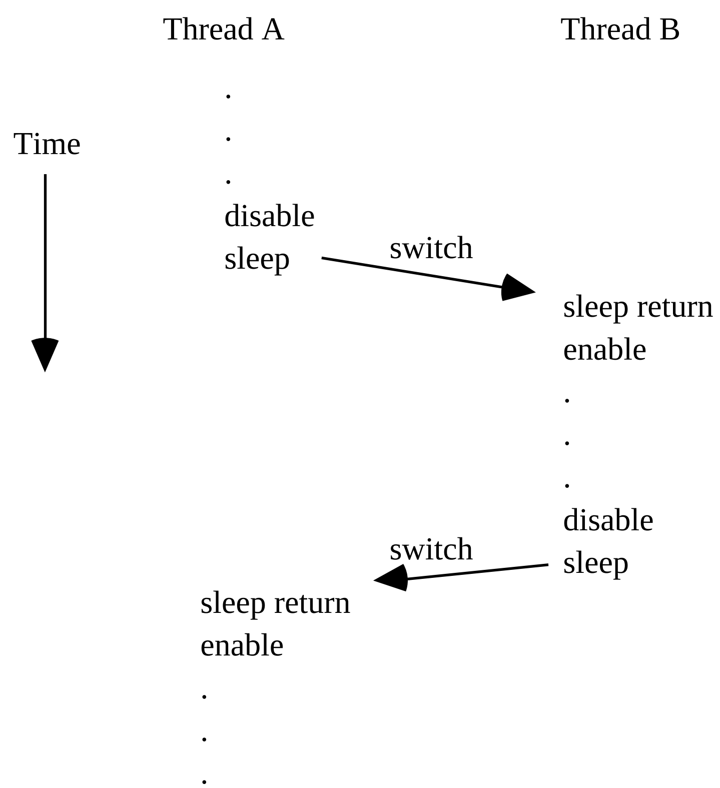
```
Lock::Acquire()
   Disable interrupts;
   if (value == BUSY) {
      put on queue of threads waiting for lock
      go to sleep
   } else {
      value = BUSY;
   }
   Enable interrupts;

 Lock::Release()
    Disable interrupts;
    if anyone on wait queue {
       take a waiting thread off
       put it on ready queue
```

```
} else {
   value = FREE;
}
Enable interrupts;
```

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 Sleep back to Acquire. Interrupts are still disabled, so it's OK to check lock value, and if it's free, grab the lock, and then turn on interrupts.



Interrupt disable and enable pattern across context switches

6.2.4 Atomic read-modify-write instructions

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

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.

6.2.4.1 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. 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.

6.2.4.2 Implementing locks with test&set

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

```
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.

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

How would we do 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.

```
Lock::Acquire()
   while (test&set(guard))
     ;
   if (value != FREE) {
       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 on ready queue
    } else {
       value = FREE;
    }
    guard = 0;
```

6.4 Summary

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