Lecture #7: Implementing Mutual Exclusion

****
Review -- 1 min
****

Solution #3 to too much milk works, but it is really unsatisfactory:
1) Really complicated – even for this simple example, hard to convince yourself that it really works
2) A's code different than B's – what if lots of threads. Code would have to be slightly different for each thread.
3) While A is waiting, it is consuming CPU time (busy-waiting)

Aside: computer science lore: N-process version of this exists and works (problem #2). "Bakery algorithm" (Lamport). But still complex (I was at a conference in 2001 that had a 1 hour invited lecture on the Bakery algorithm…), still have busy waiting, and other performance problems

****
Outline - 1 min
****

- Hardware support for synchronization
- Building higher-level synchronization programming
- Abstractions on top of hardware support

****
Preview - 1 min
****

Too much milk showed – implementing concurrent program directly w/ loads and stores is tricky and error-prone. Instead, a programmer is going to want to use higher level operations such as locks.

Today – how do we implement these higher-level operations

Next lecture – what higher-level primitives make it easier to write correct concurrent programs?

****
Lecture - 30 min
****

1. Implementing mutual exclusion

1.1 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

1.2 Ways of implementing locks

All require some level of hardware support

1.2.1 Atomic memory load and store

see too much milk lecture

NB: I used to regard the discussion of how to build locks from load and store as (a) important for understanding the issues and (b) historically important. But, I assumed that no one would actually build a lock this way these days because modern architectures have better hardware support. Well, as recently as summer 2001 I had to build a lock using load and store. What was the "primitive" architecture I was using? Javascript! I used the Bakery algorithm (see text!)

1.2.2 Disable interrupts (uniprocessor only)

Two ways for dispatcher to get control
1) **internal event** – thread does something to relinquish the CPU – system call or exception
2) **external events** – interrupt 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, tell the hardware to delay handling of external events until after we're done with the atomic operation

1.2.2.1 A flawed, but very simple solution

Why not do the following:

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

QUESTION: on a uniprocessor, does this solve too much milk?
QUESTION: what is wrong with this solution?

1. Need to support synchronization operations in user-level code; Kernel can't allow user code to get control when interrupts disabled (might never get CPU back)

   Problem for user-level code running kernel threads
   This one not so much of a problem for kernel-level code running kernel threads or user-level code running user-level threads (but others are…)

2. Real-time systems need to guarantee how long it takes to respond to interrupts, but critical sections can be arbitrarily long. Thus, 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

1.2.2.2 Implementing locks by disabling interrupts


```
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;
}
```

QUESTION: 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 they have the lock.

When disabling interrupts, the check and set operations occur w/o any other thread having the chance to execute in the middle.

QUESTION: Why do we need to enable interrupts inside the loop in Acquire?
Otherwise, since interrupts are off, the lock holder will never get a chance to run, to release the lock.

****
Admin - 3 min
****

****
Lecture - 33 min
****

1.2.3 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 other CPUs from entering the critical section

Instead, every modern processor architecture provides some sort 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

1.2.3.1 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 store conditional (R4000, Alpha) – designed to fit better with load/store architectures

- read value in one instruction
- do some operations
- when store occurs, check if value has been modified in the mean time. If not, OK. If it has changed, abort, and jump back to start

1.2.3.2 Implementing locks with test&set

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

```
Inially, 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

1.3 Busy-waiting

busy-waiting: thread consumes CPU cycles while it is waiting

solutions above use busy-waiting

- Inefficient
- problems if threads have different priorities
  - 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

- for semaphores and monitors, waiting thread may wait for an arbitrary length of time; thus, even if busy waiting was OK for locks, could be very inefficient for implementing other primitives

1.3.1 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 TCB on queue of threads waiting
        for lock
```

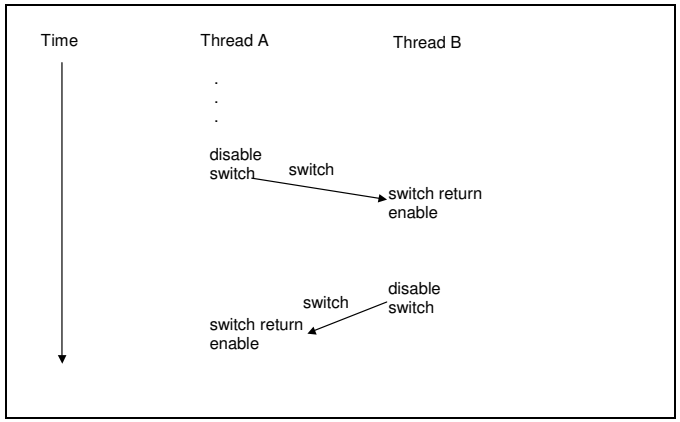
```
        switch // copy state to TCB; copy TCB'
               // to CPU
    } else{
        value = BUSY
    }
    enable interrupts;

Lock::Release()
    disable interrupts
    if anyone on wait queue{
        take a waiting thread off
        put it on ready queue
    } else{
        value = FREE;
    }
    enable interrupts
```

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

Option 1: before putting thread on the wait queue?
Then release can check queue, and not wake the thread up
Option 2: after putting thread on wait queue, but before calling switch?
Then Release puts thread on ready queue, but thread is already on ready queue! When thread wakes up, it will call switch (go to sleep), missing the wakeup from Release.

solution – maintain invariant: interrupts disabled in core switch() routine → each thread disabled interrupts at some particular point before calling switch() → each thread responsible for enabling interrupts at corresponding point on way out.

e.g. call Lock(); // enabled
disable interrupts
call switch();
return switch();
enable interrupts
return Lock(); // enabled



Interrupt disable and enable pattern across context switches

QUESTION: is it OK for thread to check the while() condition when it wakes up?

When the sleeping thread wakes up, it returns from Sleep back to Acquire. Interrupts are still disabled, so it is OK to check lock value, and if it is free, grab the lock and turn on interrupts

Aside:

- the above works for case when Lock() is implemented as a procedure call (e.g., for user-level code using user-level threads or for kernel-level code using kernel level threads)
- a bit different in the case when user-level code uses kernel-level threads (since system call may often disable interrupts/save state to TCB), but same basic idea
- also think about case where *timer interrupt* happens (rather than call to Lock()) – how do Lock() –intiated switches and timer-interrupt-initiated switches interleave? Turns out it works – just

maintain the invariant that interrupts turned off before calling switch() and turned back on when switch() returns

- e.g. call foo()
  timer interrupt // disables interrupts
  call switch();
  return switch();
  enable interrupts
  rti // enabled
  keep processing foo() // enabled

Notice that if you interleave these things, the right thing still happens.

1.3.2 Locks using test&set, with minimal busy waiting

How would 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

```
Lock::Acquire()
     while (test&set(guard))
       ;
     if(value != free)
          put on queue of threads waiting for
          lock
          call switch and 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
```

1.4 Summary

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

****
Summary - 1 min
****