Lecture #7: Shared objects and locks

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

Independent v. cooperating threads
-- can't reason about all possible interleavings

Too much milk:
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)
4) Even if you get it right, compiler or processor reordering of instructions may break your code. There are particular directives you have to use to tell the compiler/HW that you need the load/store order to be honored exactly…
   To get a flavor of these issues, see http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

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

[see next page]

```
// i is thread ID; n is max thread ID
bakeryLock:Lock(int i){
while(true){
        int j;
        choosing[i] = true;
        number[i] = max(number[0], number[1], …, number[n-1]) + 1;
        choosing[i] = false;
        for(j = 0; j < n; j++){
                while(choosing[j]){
                        ; // no-op
                }
                while(number[j] != 0
                        && (number[j] < number[i]
                                || (number[j] == number[i] && j<i))){
                        ; // No-op
                }
}

BakeryLock:Unlock(int i){
        number[i] = 0;
}
```

[aside: Compare above code with 5th edition Silbershatz on which it is based; text casts this as "The structure of process P_i in the bakery algorithm". Better – object oriented approach]

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

(1) Shared objects
(2) Locks and mutual exclusion
    a. Safety and liveness
    b. Critical section problem
    c. Implementing locks -- hardware support for synchronization
-

****
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. Shared objects

1.1 The Big Picture

Coding standards	shared objects, 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 Object oriented programming model

PICTURE

- methods that access shared state are **critical sections**
- associate a **lock** with each shared object
- acquire/release the lock when entering/exiting a method that is a critical section

Warning: Many of the "classic" synchronization problems were formulated before OO programming. Many of the textbooks still present the “classic” (non-OO) answers. **Much** better to think of answers in OO format.

Programming model is restricted -- each shared variable is only accessed within its object's critical sections → only one thread can read/write a shared variable at a time

1) *Entry section* “Lock” before entering critical section, before accessing shared data
   wait if locked
   Key idea – all synchronization involves waiting.

2) *Exit section* “unlock” when leaving, after done accessing shared data

2. Critical section problem

Shared state – state read or written by more than one thread

Synchronization: using atomic operations to ensure cooperation among threads accessing shared state

Lesson from last time – using “load” and “store” as our atomic operation is not tractable
→ critical section problem

Consider a collection of shared state and all code that reads or writes that shared state

Critical section – a set of code that accesses shared state
Critical section problem – ensure that all critical sections on a collection of shared state appear to execute atomically

> *i.e.,* thread A can never observe a state where thread B has partially executed a critical section

Rather than reasoning about atomic load/stores, reason about atomic critical sections
→ *fewer, coarser-grained interleavings*

→ *high-level invariants*

Solution to critical section problem must satisfy 3 rules:

1) **Mutual Exclusion:**
   roughly:"don't let more than one in at a time"
   precisely: never more than one thread is executing in a critical section. One thread in CS *excludes* others.
2) **Progress:**
   roughly: "let someone in" (avoid trivial solution of let no one in)
   precisely: if no threads are executing in a critical section, and a thread wishes to enter a critical section, a thread must eventually be guaranteed to enter the critical section
3) **Bounded waiting:**
   roughly: "be fair" (fairness; avoid trivial solution of "let thread 1 in")
   precisely: if thread T wishes to enter a critical section, then there exists a bound on the number of other threads that may enter the critical section before T enters

Assumption: all threads are operating at non-zero speed (*over infinite time, each ready thread is scheduled an infinite number of times*), but you cannot make any assumption about the relative speed of the threads

The rules sound a bit strange. Basic idea is simple ("roughly" above). Specific wording of precise version is to couple the progress and bounded waiting in a way that works without making assumptions about thread behaviors, clocks, schedulers, etc.

Power tool:

safety -- the program never does anything bad
liveness -- the program eventually does something good

QUESTION: which properties above are safety and which are liveness?

Safety and liveness are key properties for reasoning about programs. Any definition of a correct program can be composed of a set of safety properties and a set of liveness properties.

3. Implementing locks

3.1 Ways of implementing locks

All require some level of hardware support

3.1.1 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!)

3.1.2 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

3.1.2.1 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. *Non-modular.*

3. Simple solution might work for locks, but wouldn't work for more complex primitives, such as semaphores or condition variables

3.1.2.2 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

****

3.1.3 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

3.1.3.1 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

3.1.3.2 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

3.2 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

3.2.1 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
```

Why is it OK to context switch to a different thread with interrupts turned off?
When does Acquire re-enable interrupts in going to sleep?

Option 1: before putting thread on the wait queue?

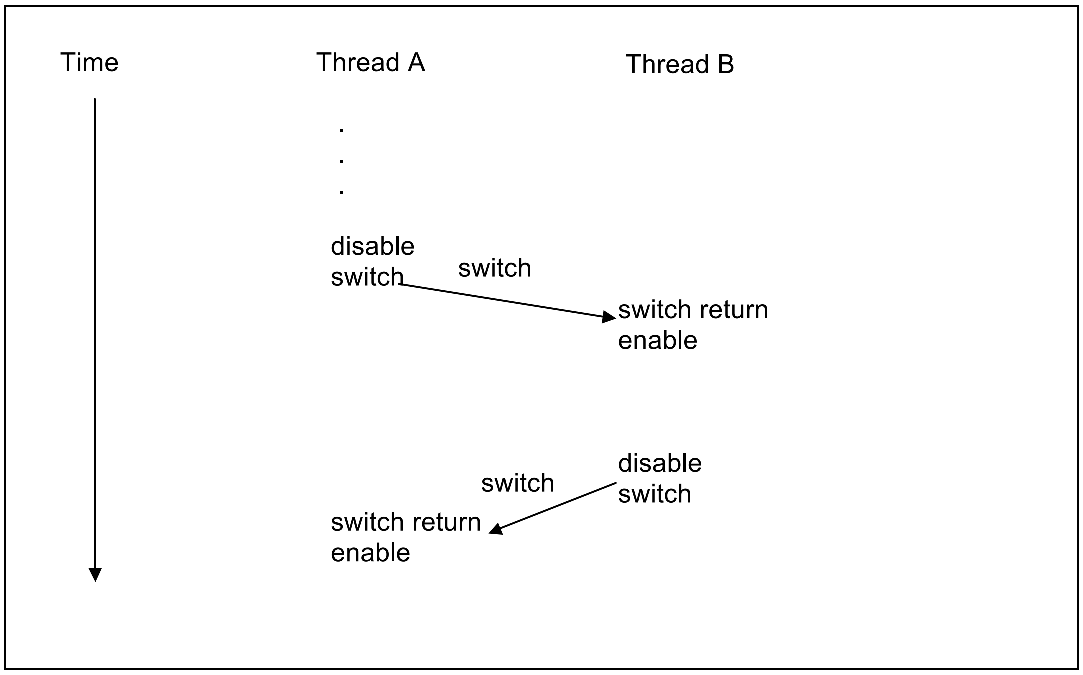
> Then release can check queue, and not wake the thread up

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() –initiated 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.

3.2.2 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

THIS CODE IS NOT QUITE RIGHT (also need interrupt disable before taking self off ready list and/or when touching ready list). See textbook draft.

```
Lock::Acquire()
     while (test&set(guard))
      ;
     if(value != free)
          disable interrupts; // Must finish
          put on queue of threads waiting for lock
          set guard to 0
          call switch
          enable interrupts;
     } else{
          value = BUSY
          guard = 0
    }



Lock::Release()
```

```
while(test&set(guard))
     ;
if(anyone on wait queue)
     take a waiting thread off
     disable interrupts; // Must finish
     put it on ready queue
     enable interrupts;
} else {
     value = free
}
guard = 0
```

Why disable interrupts?

NOT mutual exclusion (safety)

instead: liveness: Make sure I always finish my work

(1) After I take myself off ready list but before I put myself on waiting list, if I am interrupted I would be on no list and never rescheduled

(2) Ready list is (likely) protected by spinlock; once grab ready list spinlock, cannot be interrupted or system grinds to a halt

~~[older approaches/discussions before I talked about disabling interrupts here...]~~

~~[tricky bit: When do I take thread off ready queue and mark myself not runnable? Before release guard → if I am descheduled, no one will be able to grab guard. After relase guard → other thread could move me off waiting queue before I mark my self not runnable, and I'll never be seen again.~~

~~One solution is to accept a bit of busy waiting – switch() just yields and I still occasionally get scheduled for a moment to check state [need to fix code above to deal with "spurious rescheduling" – recheck value].~~

~~Another solution is to make switch mark me "not runnable" but only if I am still on queue. Switch needs to be called while still holding guard, and it needs a pointer to the wait queue. Inside of switch, on return path (after someone else switches to me), I am still holding "guard" from prior thread and if queue pointer non-null, I look to see~~

~~if prior thread is on waiting queue; if so, I make prior thread not runnable; then I release prior thread guard.~~

~~Details depend on exact semantics; test&set v. turn off interrupts; library v. kernel; … This bit of code tends to be tricky…~~

~~We do something of this flavor in lab 6.~~

3.3 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

****

4. Case study: Linux 2.6 kernel/mutex.c, kernel/mutex.h, include/linux/mutex.h, include/asm_x86/mutex_32.h

```
(Debugging info, signal handling, and compiler hints omitted…)

struct mutex {
        /* 1: unlocked, 0: locked, negative: locked, possible waiters */
        atomic_t            count;
        spinlock_t          wait_lock;
        struct list_head    wait_list;
};

void inline fastcall __sched mutex_lock(struct mutex *lock)
{
        …
        _mutex_fastpath_lock(&lock->count, _mutex_lock_slowpath);
}

/**
 * Change the count from 1 to a value lower than 1, and call <fn> if it
 * wasn't 1 originally. This function MUST leave the value lower than 1
 * even when the "1" assertion wasn't true.
 */
#define __mutex_fastpath_lock(count, fail_fn)                      \
do {
        …                                       \
        __asm__ __volatile__(                                      \
                LOCK_PREFIX "   decl (%%eax)      \n"              \
                        "   jns 1f          \n"              \
                        "   call "#fail_fn" \n"              \
                        "1:                 \n"              \
                                                             \
                :"=a" (dummy)                                \
                : "a" (count)                                \
                : "memory", "ecx", "edx");                   \
} while (0)
```

```
/*
 * Lock a mutex (possibly interruptible), slowpath:
 */
static inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                    unsigned long ip)
{
       struct task_struct *task = current;
       struct mutex_waiter waiter;
       unsigned int old_val;
       unsigned long flags;

       preempt_disable();

       spin_lock_mutex(&lock->wait_lock, flags);

       …

       /* add waiting tasks to the end of the waitqueue (FIFO): */
       list_add_tail(&waiter.list, &lock->wait_list);
       waiter.task = task;

       old_val = atomic_xchg(&lock->count, -1);
       if (old_val == 1)
              goto done;

       …

       for (;;) {
              /*
               * Lets try to take the lock again - this is needed even if
               * we get here for the first time (shortly after failing to
               * acquire the lock), to make sure that we get a wakeup once
               * it's unlocked. Later on, if we sleep, this is the
               * operation that gives us the lock. We xchg it to -1, so
               * that when we release the lock, we properly wake up the
               * other waiters:
               */
              old_val = atomic_xchg(&lock->count, -1);
              if (old_val == 1)
                     break;

              …

              /* didnt get the lock, go to sleep: */
              spin_unlock_mutex(&lock->wait_lock, flags);
              preempt_enable_no_resched();
              schedule();
              preempt_disable();
              spin_lock_mutex(&lock->wait_lock, flags);
       }

done:
       …
       /* got the lock - rejoice! */
       mutex_remove_waiter(lock, &waiter, task_thread_info(task));
       …

       /* set it to 0 if there are no waiters left: */
       if (likely(list_empty(&lock->wait_list)))
              atomic_set(&lock->count, 0);

       spin_unlock_mutex(&lock->wait_lock, flags);
```

```
        …

        return 0;
}
```

4.1 Unlock

```
void fastcall __sched mutex_unlock(struct mutex *lock)
{
        /*
         * The unlocking fastpath is the 0->1 transition from 'locked'
         * into 'unlocked' state:
         */
        __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}

/**
 * try to promote the mutex from 0 to 1. if it wasn't 0, call <fail_fn>.
 * In the failure case, this function is allowed to either set the value
 * to 1, or to set it to a value lower than 1.
*/
#define __mutex_fastpath_unlock(count, fail_fn)                         \
do {                                                            \
        unsigned int dummy;                                     \
                                                                \
        …
                                                                \
        __asm__ __volatile__(                                           \
                LOCK_PREFIX "   incl (%%eax)     \n"                    \
                        "   jg 1f           \n"                 \
                        "   call "#fail_fn" \n"                 \
                        "1:                 \n"                 \
                                                                \
                :"=a" (dummy)                                   \
                : "a" (count)                                   \
                : "memory", "ecx", "edx");                      \
} while (0)


/*
 * Release the lock, slowpath:
 */
static fastcall inline void
__mutex_unlock_common_slowpath(atomic_t *lock_count, int nested)
{
        struct mutex *lock = container_of(lock_count, struct mutex, count);
        unsigned long flags;

        spin_lock_mutex(&lock->wait_lock, flags);
        …

        /*
         * some architectures leave the lock unlocked in the fastpath failure
         * case, others need to leave it locked. In the later case we have to
         * unlock it here
         */
        if (__mutex_slowpath_needs_to_unlock())
                atomic_set(&lock->count, 1);

        if (!list_empty(&lock->wait_list)) {
                /* get the first entry from the wait-list: */
                struct mutex_waiter *waiter =
                                list_entry(lock->wait_list.next,
                                           struct mutex_waiter, list);

                …

                wake_up_process(waiter->task);
        }

        …

        spin_unlock_mutex(&lock->wait_lock, flags);
}
```