```
Handout for CS 372H
Class 7
7 February 2012

1. Example to illustrate interleavings: say that thread A executes f()
and thread B executes g(). (Here, we are using the term "thread"
abstractly. This example applies to any of the approaches that fall
under the word "thread".)

    a.

        int x;

        f() { x = 1; }

        g() { x = 2; }

        What are possible values of x after A has executed f() and B has
        executed g()?

    b.
        int y = 12;

        f() { x = y + 1; }
        g() { y = y * 2; }

        What are the possible values of x?

    c.
        int x = 0;
        f() { x = x + 1; }
        g() { x = x + 2; }

        What are the possible values of x?

2. Linked list example

    struct List_elem {
        int data;
        struct List_elem* next;
    };

    List_elem* head = 0;

    insert(int data) {
        List_elem* l = new List_elem;
        l->data = data;
        l->next = head;
        head = l;
    }

    What happens if two threads execute insert() at once and we get the
    following interleaving?

    thread 1: l->next = head
    thread 2: l->next = head
    thread 2: head = l;
    thread 1: head = l;
```



```
3. Producer/consumer example:

    /*
    "buffer" stores BUFFER_SIZE items
    "count" is number of used slots. a variable that lives in memory
    "out" is next empty buffer slot to fill (if any)
    "in" is oldest filled slot to consume (if any)
    */

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();
             while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
         }
     }

     void consumer (void *ignored) {
         for (;;) {
             while (count == 0)
                ; // do nothing
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count--;
             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    /*
       what count++ probably compiles to:
        reg1 <-- count      # load
        reg1 <-- reg1 + 1   # increment register
        count <-- reg1      # store

       what count-- could compile to:
        reg2 <-- count      # load
        reg2 <-- reg2 - 1   # decrement register
        count <-- reg2      # store
    */

    What happens if we get the following interleaving?

        reg1 <-- count
        reg1 <-- reg1 + 1
        reg2 <-- count
        reg2 <-- reg2 - 1
        count <-- reg1
        count <-- reg2
```

```

4. Some other examples. What is the point of these?

    [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
    66−76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]

    a. Can both "critical sections" run?

        int flag1 = 0, flag2 = 0;

        int main () {
            tid id = thread_create (p1, NULL);
            p2 (); thread_join (id);
        }

        void p1 (void *ignored) {
            flag1 = 1;
            if (!flag2) {
                critical_section_1 ();
            }
        }

        void p2 (void *ignored) {
            flag2 = 1;
            if (!flag1) {
                critical_section_2 ();
            }
        }

    b. Can use() be called with value 0, if p2 and p1 run concurrently?

        int data = 0, ready = 0;

        void p1 () {
            data = 2000;
            ready = 1;
        }
        int p2 () {
            while (!ready) {}
            use(data);
        }

    c. Can use() be called with value 0?

        int a = 0, b = 0;

        void p1 (void *ignored) { a = 1; }

        void p2 (void *ignored) {
          if (a == 1)
            b = 1;
        }

        void p3 (void *ignored) {
          if (b == 1)
            use (a);
        }

    d.
        /* keyword "register" tells compiler to place the variable in a
          register, not on the stack. So f, g are local to each thread. */

        int flag1 = 0, flag2 = 0;

        int p1 (void *ignored)       int p2 (void *ignored)
        {                            {
          register int f, g;           register int f, g;
          flag1 = 1;                   flag2 = 1;
          f = flag1;                   f = flag2;
          g = flag2;                   g = flag1;
          return 2*f + g;              return 2*f + g;
        }                            }

```



```
5. Protecting the linked list......

        Lock list_lock;

        insert(int data) {
            List_elem* l = new List_elem;
            l−>data = data;

            acquire(&list_lock);

            l−>next = head;         // A
            head = l;               // B

            release(&list_lock);
        }

6. How can we implement list_lock, acquire(), and release()?

    6a. Here is A BADLY BROKEN implementation:

        struct Lock {
          int locked;
        }

        void [BROKEN] acquire(Lock *lock) {
          while (1) {
            if (lock−>locked == 0) { // C
              lock−>locked = 1;    // D
              break;
            }
          }
        }

        void release (Lock *lock) {
          lock−>locked = 0;
        }

        What’s the problem? Two acquire()s on the same lock on different
        CPUs might both execute line C, and then both execute D. Then
        both will think they have acquired the lock. This is the same
        kind of race that we were trying to eliminate in insert().  But
        we have made a little progress: now we only need a way to
        prevent interleaving in one place (acquire()), not for many
        arbitrary complex sequences of code.

```

```
    6b. Here’s a way that is correct but only sometimes appropriate:
        Use an atomic instruction on the CPU. For example, on the x86,
        doing
                "xchg addr, %eax"
        does the following:

        (i)   freeze all CPUs’ memory activity for address addr
        (ii)  temp = *addr
        (iii) *addr = %eax
        (iv)  %eax = temp
        (v)   un−freeze memory activity

        /* pseudocode */
        int xchg_val(addr, value) {
            %eax = value;
            xchg (*addr), %eax
        }

        struct Lock {
          int locked;
        }

        /* bare−bones version of acquire */
        void acquire (Lock *lock) {
          pushcli();    /* what does this do? */
          while (1) {
            if (xchg_val(&lock−>locked, 1) == 0)
              break;
          }
        }

        /* optimization in acquire; call xchg_val() less frequently */
        void acquire(Lock* lock) {
            pushcli();
            while (xchg_val(&lock−>locked, 1) == 1) {
                while (lock−>locked) ;
            }
        }

        void release(Lock *lock){
           xchg_val(&lock−>locked, 0);
           popcli();    /* what does this do? */
        }

        The above is called a *spinlock* because acquire() spins.

        The spinlock above is great for some things, not so great for
        others. The main problem is that it *busy waits*: it spins,
        chewing up CPU cycles. Sometimes this is what we want (e.g., if
        the cost of going to sleep is greater than the cost of spinning
        for a few cycles waiting for another thread or process to
        relinquish the spinlock). But sometimes this is not at all what we
        want (e.g., if the lock would be held for a while: in those
        cases, the CPU waiting for the lock would waste cycles spinning
        instead of running some other thread or process).


    6c. Here’s an object that does not involve busy waiting; it can work
    as the list_lock mentioned above. Note: the "threads" here
    can be user−level threads, kernel threads, or threads−inside−kernel.
    The concept is the same in all cases.

        struct Mutex {
            bool is_held;           /* true if mutex held */
            thread_id owner;        /* thread holding mutex, if locked */
            thread_list waiters;    /* queue of thread TCBs */
            Lock wait_lock;         /* as in 6b */
        }

        Now, instead of acquire(&list_lock) and release(&list_lock) as
        abve, we’d write, mutex_acquire(&list_mutex) and
        mutex_release(&list_mutex). The implementation of the latter two
        would be something like this:
```



```

        void mutex_acquire(Mutex *m) {

            acquire(&m−>wait_lock);   /* we spin to acquire wait_lock */
            while (m−>is_held) {     /* someone else has the mutex */
                m−>waiters.insert(current_thread)
                release(&m−>wait_lock);
                schedule();   /* run a thread that is on the ready list */
                acquire(&m−>wait_lock);   /* we spin again */
            }
            m−>is_held = true;      /* we now hold the mutex */
            m−>owner = self;
            release(&m−>wait_lock);
        }

        void mutex_release(Mutex *m) {

            acquire(&m−>wait_lock);    /* we spin to acquire wait_lock */
            m−>is_held = false;
            m−>owner = 0;
            wake_up_a_waiter(m−>waiters); /* select and run a waiter */
            release(&m−>wait_lock);

        }

        [Please let me (MW) know if you see bugs in the above.]

7. NOTE: Unfortunately, insert() with these locks is correct only if
there are some constraints on the order in which the CPU carries out
memory reads and writes. For example, if insert() were executed so that
the read at A appeared to another processor (and to memory) to be
executed before the acquire(), then insert() would be incorrect even
with locks.

How do we get the required guarantee? Answer: by ensuring that neither
the programmer nor the processor reorders instructions with respect to
the acquire().

8. Terminology

    To avoid confusion, we will use the following terminology in this
    course (you will hear other terminology elsewhere):

    −−A "lock" is an abstract object that provides mutual exclusion

    −−A "spinlock" is a lock that works by busy waiting, as in 6b

    −−A "mutex" is a lock that works by having a "waiting" queue and
    then protecting that waiting queue with atomic hardware
    instructions, as in 6c. The most natural way to "use the hardware"
    is with a spinlock, but there are others, such as turning off
    interrupts, which works if we’re on a single CPU machine.
```

```
9. Producer/consumer example [also known as bounded buffer]

   9a. buggy implementation

    /*
    "buffer" stores BUFFER_SIZE items
    "count" is number of used slots. a variable that lives in memory
    "out" is next empty buffer slot to fill (if any)
    "in" is oldest filled slot to consume (if any)
    */

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();
             while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
         }
     }

     void consumer (void *ignored) {
         for (;;) {
             while (count == 0)
                ; // do nothing
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count−−;
             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    −−Review: what’s the problem?
    −−Answer: count++ and count−− might compile to, respectively:

        reg1 <−− count      # load
        reg1 <−− reg1 + 1   # increment register
        count <−− reg1      # store

        reg2 <−− count      # load
        reg2 <−− reg2 − 1   # decrement register
        count <−− reg2      # store

    −−Review: why not use instructions like "addl $0x1, _count"?
    −−Answer: not atomic if there are multiple CPUs.

    −−Review: so why not use "LOCK addl $0x1, _count"?
    −−Answer: we could do that here, but LOCK won’t save us every time

    −−Review: so use general−purpose approach to protecting
    critical sections: locks (or mutexes).
```



```
   9b. Producer/consumer [bounded buffer] using mutexes

     Mutex mutex;

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();

             acquire(&mutex);
             while (count == BUFFER_SIZE) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
             release(&mutex);
         }
     }

     void consumer (void *ignored) {
         for (;;) {

             acquire(&mutex);
             while (count == 0) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count−−;
             release(&mutex);

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }
```

```

    9c. Producer/consumer [bounded buffer] using mutexes and condition
    variables

         Mutex mutex;
         Cond nonempty;
         Cond nonfull;

         void producer (void *ignored) {
             for (;;) {
                 /* next line produces an item and puts it in nextProduced */
                 nextProduced = means_of_production();

                 acquire(&mutex);
                 while (count == BUFFER_SIZE)
                    cond_wait(&nonfull, &mutex);

                 buffer [in] = nextProduced;
                 in = (in + 1) % BUFFER_SIZE;
                 count++;
                 cond_signal(&nonempty, &mutex);
                 release(&mutex);
             }
         }

         void consumer (void *ignored) {
             for (;;) {

                 acquire(&mutex);
                 while (count == 0)
                    cond_wait(&nonempty, &mutex);

                 nextConsumed = buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count−−;
                 cond_signal(&nonfull, &mutex);
                 release(&mutex);

                 /* next line abstractly consumes the item */
                 consume_item(nextConsumed);
             }
         }


        Question: why does cond_wait need to both release the mutex and
        sleep? Why not:

            while (count == BUFFER_SIZE)  {
                release(&mutex);
                cond_wait(&nonfull);
                acquire(&mutex);
            }

```



```
    9d.  Producer/consumer [bounded buffer] with semaphores

        Semaphore mutex(1);             /* mutex initialized to 1 */
        Semaphore empty(BUFFER_SIZE);  /* start with BUFFER_SIZE empty slots */
        Semaphore full(0);             /* 0 full slots */

        void producer (void *ignored) {
             for (;;) {
                 /* next line produces an item and puts it in nextProduced */
                 nextProduced = means_of_production();

                /*
                 * next line diminishes the count of empty slots and
                 * waits if there are no empty slots
                 */
                 sem_down(&empty);
                 sem_down(&mutex);  /* get exclusive access */

                 buffer [in] = nextProduced;
                 in = (in + 1) % BUFFER_SIZE;

                 sem_up(&mutex);
                 sem_up(&full);   /* we just increased the # of full slots */
             }
         }

         void consumer (void *ignored) {
             for (;;) {

                 /*
                  * next line diminishes the count of full slots and
                  * waits if there are no full slots
                  */
                 sem_down(&full);
                 sem_down(&mutex);

                 nextConsumed = buffer[out];
                 out = (out + 1) % BUFFER_SIZE;

                 sem_up(&mutex);
                 sem_up(&empty);   /* one further empty slot */

                 /* next line abstractly consumes the item */
                 consume_item(nextConsumed);
             }
         }

        Semaphores *can* (not always) lead to elegant solutions (notice
        that the code above is fewer lines than 1c) but they are much
        harder to use.

        The fundamental issue is that semaphores make implicit (counts,
        conditions, etc.) what is probably best left explicit. Moreover,
        they *also* implement mutual exclusion.

        For this reason, you should not use semaphores. This example is
        here mainly for completeness and so you know what a semaphore
        is. But do not code with them. Solutions that use semaphores in
        this course will receive no credit.

```