1. Example to illustrate interleavings: say that thread A executes f() and thread B executes g(). (Here, we are using the term "thread" abstractly, to refer to execution contexts that share memory.)

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 = x + 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 = 1;
thread 1: head = 1;

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
reg1 <−− reg1 + 1   # 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
}
4. 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);
    }

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

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.

5a. Test-and-set spinlock

    void acquire(Lock *lock) {
        pushcli();    /* what does this do? */
        while (1) {
            if (xchg_val(&lock->locked, 1) == 1) {
                while (lock->locked);
            }
        }
    }

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

5b. Test-and-test-and-set lock

    void acquire(Lock *lock) {
        pushcli();
        while (xchg_val(&lock->locked, 1) == 1) {
            while (lock->locked);
        }
    }
The spinlocks presented above have fairness issues on NUMA machines (cores closer to the memory containing the ‘locked’ variable are more likely to succeed in acquiring the lock).

Ticket locks address that issue. They rely on an atomic primitive known as "fetch and increment."

On the x86, we implement fetch and increment with the XADD instruction, but note that this instruction is not atomic by default, so we need the LOCK prefix.

Here’s pseudocode:

```c
int fetch_and_increment (int* addr) {
    LOCK: // remember, this is pseudocode
    int was = *addr;
    *addr = was + 1;
    return was;
}
```

Here’s inline assembly:

```c
inline int fetch_and_increment(int *addr) {
    int was = 1;
    asm volatile("lock xaddl %1, %0" : "+m" (*addr), "+r" (was)  // Output ":1" (was), "+m" (*addr) // Input
    );
    return was;
}
```

Ticket locks are fair, as noted above, but they (and baseline spinlocks) have performance issues when there is a lot of contention. These issues fundamentally result from cross-talk among CPUs (which undermines caching and generates traffic on the memory bus). This phenomenon is investigated in depth in the "Scalable Locks are Dangerous" paper.

The locks presented below address that issue. These are known as MCS locks.


A. CAS / CMPXCHG

Useful operation: compare-and-swap, known as CAS. Says: "atomically check whether a given memory cell contains a given value, and if it does, then replace the contents of the memory cell with this other value; in either case, return the original value in the memory location".

On the X86, we implement CAS with the CMPXCHG instruction, but note that this instruction is not atomic by default, so we need the LOCK prefix.

Here’s pseudocode:

```c
int cmpxchg_val(int* addr, int oldval, int newval) {
    LOCK: // remember, this is pseudocode
    int was = *addr;
    if (*addr == oldval)
        *addr = newval;
    return was;
}
```

Here’s inline assembly:

```c
uint32_t cmpxchg_val(uint32_t* addr, uint32_t oldval, uint32_t newval) {
    uint32_t was;
    asm volatile("lock cmpxchg %3, %0" : "+m" (*addr), "+a" (was) ":a" (oldval), "+m" (*addr) : ":cc");
    return was;
}
```

B. The MCS lock

Each CPU has a qnode structure in *local* memory. Here, local can mean local memory in NUMA machine or its own cache line that other CPUs are not allowed to cache (i.e., the cache line is in exclusive mode):

```c
typedef struct qnode {
    struct qnode* next;
    bool someoneelse_locked;
} qnode;
```

typedef qnode* lock; // a lock is a pointer to a qnode

---The lock itself is literally the *tail* of the list of CPUs holding or waiting for the lock.

---While waiting, a CPU spins on its local "locked" flag.
mutexes

Here's the code for acquire:

```c
void acquire(lock* lockp, qnode* I) {
    I->next = NULL;
    qnode* predecessor = xchg_val(lockp, I);
    if (predecessor != NULL) { // queue was non-empty
        predecessor->next = I;
        while (I->someoneelse_locked);    // spin
    } // we hold the lock!
    I->someoneelse_locked = true;
    while (I->someoneelse_locked);    // spin
    // next line makes lockp point to I (that is, it sets *lockp <- I)
    // and returns the old value of *lockp. Uses atomic operation
    // XCHG. see earlier in handout (or earlier handouts)
    // for implementation of xchg_val.
    predecessor = xchg_val(lockp, I);    // "A"
    if (predecessor != NULL) { // queue was non-empty
        predecessor->next = I;    // "B"
        I->next = NULL;          // "C"
        while (!I->next);   // no known successor
        // swap successful: lockp was pointing to I, so now
        // *lockp == NULL, and the lock is unlocked. we can
        // go home now.
        return;
    } // if we get here, then there was a timing issue: we had
    // no known successor when we first checked, but now we
    // have a successor: some CPU executed the line "A"
    // above. Wait for that CPU to execute line "B" above.
    while (!I->next);    // just setting that waiter's "someoneelse_locked" flag to false
    I->next->someoneelse_locked = false;
    // we hold the lock!
    I->someoneelse_locked = true;
    while (I->someoneelse_locked);    // spin
    // next line makes lockp point to I (that is, it sets *lockp <- I)
    // and returns the old value of *lockp. Uses atomic operation
    // XCHG. see earlier in handout (or earlier handouts)
    // for implementation of xchg_val.
    predecessor = xchg_val(lockp, I);    // "A"
    if (predecessor != NULL) { // queue was non-empty
        predecessor->next = I;    // "B"
        I->next = NULL;          // "C"
        while (!I->next);   // no known successor
        // swap successful: lockp was pointing to I, so now
        // *lockp == NULL, and the lock is unlocked. we can
        // go home now.
        return;
    } // if we get here, then there was a timing issue: we had
    // no known successor when we first checked, but now we
    // have a successor: some CPU executed the line "A"
    // above. Wait for that CPU to execute line "B" above.
    while (!I->next);    // just setting that waiter's "someoneelse_locked" flag to false
```

Here's the code for release:

```c
void release(lock* lockp, qnode* I) {
    if (!I->next)   { // no known successor
        if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
            // swap successful: lockp was pointing to I, so now
            // *lockp == NULL, and the lock is unlocked. we can
            // go home now.
            return;
        } // if we get here, then there was a timing issue: we had
        // no known successor when we first checked, but now we
        // have a successor: some CPU executed the line "A"
        // above. Wait for that CPU to execute line "B" above.
        while (!I->next);    // just setting that waiter's "someoneelse_locked" flag to false
    I->next->someoneelse_locked = false;
```

What's going on?

---If the lock is unlocked, then *lockp == NULL.

---If the lock is locked, and there are no waiters, then *lockp
points to the qnode of the owner

---If the lock is locked, and there are waiters, then *lockp
points to the qnode at the tail of the waiter list.

---Here's the code for release:

```c
void release(lock* lockp, qnode* I) {
    if (!I->next)   { // no known successor
        if (cmpxchg_val(lockp, I, NULL) == I) {     // "C"
            // swap successful: lockp was pointing to I, so now
            // *lockp == NULL, and the lock is unlocked. we can
            // go home now.
            return;
        } // if we get here, then there was a timing issue: we had
        // no known successor when we first checked, but now we
        // have a successor: some CPU executed the line "A"
        // above. Wait for that CPU to execute line "B" above.
        while (!I->next);    // just setting that waiter's "someoneelse_locked" flag to false
```

What's going on?

---If I->next == NULL and *lockp == I, then no one else is
waiting for the lock. So we set *lockp == NULL.

---If I->next == NULL and *lockp != I, then another CPU is in
acquire (specifically, it executed its atomic operation, namely
line "A", before we executed ours, namely line "C"). So we wait for
the other CPU to put the list in a sane state, and then drop
down to the next case:

---If I->next != NULL, then we know that there is a spinning
waiter (the oldest one). Hand it the lock by setting its flag to
false.