1. Implementing threads

Per-thread state in thread control block:

```c
typedef struct tcb {
  unsigned long esp;    /* Stack pointer of thread */
  char *t_stack;        /* Bottom of thread's stack */
  /* ... */
};
```

Machine-dependent thread-switch function:

```c
void swtch(tcb *current, tcb *next);
```

Machine-dependent thread initialization function:

```c
void thread_init(tcb *t, void (*fn) (void *), void *arg);
```

Implementation of `swtch(current, next)`:

```c
pushl %ebp; movl %esp, %ebp # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi # Save callee-saved regs
movl (%ebp),%edx # %edx = current
movl 12(%ebp),%eax # %eax = next
movl %esp,(%edx) # %edx->esp = %esp
movl (%eax),%esp # %esp = %eax->esp
popl %edi; popl %esi; popl %ebx # Restore callee saved regs
popl %ebp # Restore frame pointer
ret # Resume execution
```

[thanks to David Mazieres]

2. How can we implement locks, `acquire()`, and `release()`?

2a. Here is a BADLY BROKEN implementation:

```c
struct Lock {
  int locked;
};
```

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

```c
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 we were trying to eliminate to begin with. 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.
2b. Here's a way that is correct but that is appropriate only in some circumstances:

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;
    int is_held;
    int owner;
    thread_list waiters;    /* queue of thread TCBs */
    Lock wait_lock;     /* as in 2b */
}

(a) freeze all CPUs’ memory activity for address addr

(b) do something

(c) un-freeze memory activity

(d) freeze all CPUs’ memory activity for address addr

(e) do something

(f) un-freeze memory activity

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).

2c. Here's an object that does not involve busy waiting. 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 2b */
}
```

The implementation of acquire() and release() would be something like:

```
void mutex_acquire(Mutex *m) {
    acquire(&m->wait_lock);   /* we spin to acquire wait_lock */
    if (xchg_val(&m->locked, 1) == 0)
        break;
    while (1) {
        if (xchg_val(&m->locked, 0) == 0)
            break;
    }
}
```

```
void mutex_release(Mutex *m) {
    acquire(&m->wait_lock);    /* we spin to acquire wait_lock */
    m->is_held = true;     /* we now hold the mutex */
    m->owner = self;
    release(&m->wait_lock);
}
```

4. 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 2b

--A "mutex" is a lock that works by having a "waiting" queue and then protecting that waiting queue with atomic hardware instructions, as in 2c. 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.