Watch this video in-depth:
https://www.youtube.com/watch?v=gQdflOUZQvA&list=PLbtzT1TYeoMhTPzyTZboW_j7TPAnjv9XB&index=4

Spinlocks and Mutexes {
    Put the code in user/{filename}.h
    Just copy the spinlock code from the kernel.
    Remember to change the type of the locked variable to uint8. It will be important later.
    Comment out push_off() and pop_off() since they are no longer relevant in the user space.
    Since mycpu() is not available to the user, use getpid() instead to check who is holding the lock.

    While acquiring mutex lock, I think it is better to use sleep(1) instead of writing a yield() system call,
        because the kernel will run the scheduler immedieately after the process has yielded, and might
        allocate another time slice to the same process that called yield() in the first place.
        It is guaranteed to happen if #runnable <= #cpu.
        This leads to busy waiting.
}

Conditionals {
    You'll need to use a queue inside the conditional variable struct. Just copy the one in producer_consumer.c
        There is an edge case where popping from an empty queue messes up the queue.
        Remember to handle it.

    Follow this link:
https://www.andrew.cmu.edu/course/15-440-sp11/applications/ln/lecture7.html
        Use the queue to store the pid of waiting threads.

    Note that the mutex-release and going to sleep MUST be atomic, but I couldn't find a way to handle it
        from the user-space without implementing a modified version of sleep(). Suggestions are welcome.

    The sleep() function in proc.c acquires a lock on the process before releasing the spinlock. This is "atomic"
        because no other process can wake up the process without acquiring p->lock, so no other process can wake it up
        before the calling process has released the lock and gone to sleep. This circumvents the lost wakeup issue.

We'll need to make our own copy of sleep(). Let's call it new_sleep(). Drop all references to p->chan, we don't need it.
comment out the acquire(lk) line at the end, the lock re-acquiring will happen in the user space.

Let's assume our version of sleep system call is release_sleep. In the pseudocode given in the link, they called

```
release_mutex (mx));
thr_suspend (self);
```

We need to replace these two lines with a call to release_sleep, like:
release_sleep(mx->locked)

We could take a pointer to the lock as an input to the system call calling new_sleep() (to use as lk),
but the problem is that we can't directly modify user-space variables from the kernel space,
so we can't release the lock with __sync_lock_release() directly. We must use copyout(), or something like that.
But the problem with copyout is that it is not atomic, and might lead to race conditions if another thread
tries to modify the lock while copyout() is running.

Instad, we could try taking the address of the 1-byte locked variable inside the lock structure as a pointer,
like this: release_sleep(&mutex->lock). Then, we'll call a modified version of copyout() on this pointer
from inside new_sleep(), where release(lk) used to be.

Make the following modifications to your version of copyout():

1. Drop the parameters src and len, since len is assumed to be 1 (that's why locked was turned into uint8).
2. Remove the while loop. Only one iteration is needed.
3. Drop the variable n, and delete every line that references this variable.
4. Write these two lines before return 0:

```
__sync_synchronize();
__sync_lock_release((uint8*)(pa0 + (dstva - va0)));
```

pa0 + (dstva - va0) is the physical address of the locked varaible. By calling release on this address,
we are essentially doing the same thing done in the release() function in spinlock.c,
just in a more roundabout way.

If you are keeping track of pid, remember to set mutex->pid to 0 in the userspace, before calling release_sleep(),
just like we did while releasing spinlocks and mutexes.

It is the user's responsibility to handle spurious wakeups.

Waking up a sleeping process is extremely easy, just use pid instead of chan in a copy of wakeup() in proc.c
}

For semaphore, see page 405 of OSTEP.