Question 4 – Pthreads

1. C code using the pthreads library can be used to implement a monitor, but you have to do everything by hand. In particular, you do the following transformation for a singleton monitor like the one we are creating here:

```c
monitor {
    int field1;
    <type> field2;
    condition C1, C2;

    method1(args) {
        ...
        wait(C1)
        signal(C2)
        ...
    }
}

int field1;
<type> field2;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t C1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t C2 = PTHREAD_COND_INITIALIZER;

method1(args) {
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_wait(&C1, &mutex);
    pthread_cond_signal(&C2);
    ...
    pthread_mutex_unlock(&mutex);
}
```

The changes are as follows:

- each method becomes a function, and all the fields become global variables. (this works because in our case the monitor is a singleton class – i.e. there is only ever a single instance)
- a single explicit mutex ('mutex' in this case) replaces the implicit mutex in the monitor.
- each method locks the explicit mutex on entry, and unlocks it on exit. No other code ever touches this mutex, and no other mutexes are used.
- each wait() call [and in our case, sleep_safely()] must pass the explicit mutex so that it can be unlocked while waiting.

2. You might want to visualize your code as a game board – each thread corresponds to a token moving along a path. Only one token (thread) moves at once, and it advances until it reaches a spot corresponding to a wait() or sleep() call. At that point it stops and another token can start moving.

3. Note that signal() and broadcast() do not leave the monitor. This means that if thread A calls signal(C), thus waking thread B which was blocked in wait(C), thread B will not return from wait() until after A leaves the monitor, by wait(), sleep_safely(), or return.

In some cases this may result in “overlap” of events, where e.g. two students sit down with the professor a millisecond or two before the previous two get up. This is OK.
4. Note that the professor() function is called in a tight loop:

```c
while (1)
    professor();
```

If the professor function returns without waiting, then it will loop consuming 100% of the CPU, which may cause your threads to run a bit strangely. In other words, **don't** do something like this:

```c
professor(void)
{
    if (bool_variable) {
        pthread_mutex_lock(&mutex);
        do something...
        pthread_mutex_unlock(&mutex);
    }
}
```

Besides, there's an error in this code – the entire method should be guarded by 'mutex', not just the body of the 'if'. The access to 'bool_variable' is entirely unprotected, and subject to race conditions.