ass: Barriers

In this assignment we will explore how to implement a barrier using condition variables provided by the pthread library. A barrier is a point in an application at which all threads must wait until all other threads reach that point too. Condition variables are a sequence coordination technique similar to xv6's sleep and wakeup. Submit your solutions before the beginning of the next lecture to the submission web site.

Please feel free to collaborate with others on these exercises.

Download barrier.c and compile it on your laptop or Athena machine:

```
$ gcc -g -O2 -pthread barrier.c
$ ./a.out 2
Assertion failed: (i == t), function thread, file barrier.c, line 55.
```

The 2 specifies the number of threads that synchronize on the barrier (`nthread` in `barrier.c`). Each thread sits in a tight loop. In each loop iteration a thread calls `barrier()` and then sleeps for some random number of microseconds. The assert triggers, because one thread leaves the barrier before the other thread has reached the barrier. The desired behavior is that all threads should block until `nthread`s have called `barrier`.

Your goal is to achieve the desired behavior. In addition to the lock primitives that you have seen before, you will need the following new pthread primitives (see man pthread for more detail):

```
pthread_cond_wait(&cond, µtex);  // go to sleep on cond, releasing lock mutex
pthread_cond_broadcast(&cond);     // wake up every thread sleeping on cond
```

`pthread_cond_wait` releases the `mutex` when called, and re-acquires the `mutex` before returning.

We have given you `barrier_init()`. Your job is to implement `barrier()` so that the panic won't occur. We've defined `struct barrier` for you; its fields are for your use.

There are two issues that complicate your task:

- You have to deal with a succession of barrier calls, each of which we'll call a round. `bstate.round` records the current round. You should increase `bstate.round` when each round starts.
- You have to handle the case in which one thread races around the loop before the others have exited the barrier. In particular, you are re-using `bstate.nthread` from one round to the next. Make sure that a thread that leaves the barrier and races around the loop doesn't increase `bstate.nthread` while a previous round is still using it.

Test your code with one, two, and more than two threads.

Submit: your modified barrier.c