5.6 Building H_2O

This problem has been a staple of the Operating Systems class at U.C. Berkeley for at least a decade. It seems to be based on an exercise in Andrews's *Concurrent Programming* [1].

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

As each thread passes the barrier, it should invoke `bond`. You must guarantee that all the threads from one molecule invoke `bond` before any of the threads from the next molecule do.

In other words:

- If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
- If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

We don't have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets; thus, if we examine the sequence of threads that invoke `bond` and divide them into groups of three, each group should contain one oxygen and two hydrogen threads.

Puzzle: Write synchronization code for oxygen and hydrogen molecules that enforces these constraints.

5.6.1 H_2O hint

Here are the variables I used in my solution:

Water building hint

```
mutex = Semaphore(1)
oxygen = 0
hydrogen = 0
barrier = Barrier(3)
oxyQueue = Semaphore(0)
hydroQueue = Semaphore(0)
```

`oxygen` and `hydrogen` are counters, protected by `mutex`. `barrier` is where each set of three threads meets after invoking `bond` and before allowing the next set of threads to proceed.

`oxyQueue` is the semaphore oxygen threads wait on; `hydroQueue` is the semaphore hydrogen threads wait on. I am using the naming convention for queues, so `oxyQueue.wait()` means "join the oxygen queue" and `oxyQueue.signal()` means "release an oxygen thread from the queue."

5.6.2 H_2O solution

Initially `hydroQueue` and `oxyQueue` are locked. When an oxygen thread arrives it signals `hydroQueue` twice, allowing two hydrogens to proceed. Then the oxygen thread waits for the hydrogen threads to arrive.

Oxygen code

```
mutex.wait()
oxygen += 1
if hydrogen >= 2:
    hydroQueue.signal(2)
    hydrogen -= 2
    oxyQueue.signal()
    oxygen -= 1
else:
    mutex.signal()

oxyQueue.wait()
bond()

barrier.wait()
mutex.signal()
```

As each oxygen thread enters, it gets the mutex and checks the scoreboard. If there are at least two hydrogen threads waiting, it signals two of them and itself and then bonds. If not, it releases the mutex and waits.

After bonding, threads wait at the barrier until all three threads have bonded, and then the oxygen thread releases the mutex. Since there is only one oxygen thread in each set of three, we are guaranteed to signal `mutex` once.

The code for hydrogen is similar:

Hydrogen code

```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    hydroQueue.signal(2)
    hydrogen -= 2
    oxyQueue.signal()
    oxygen -= 1
else:
    mutex.signal()

hydroQueue.wait()
bond()

barrier.wait()
```

An unusual feature of this solution is that the exit point of the mutex is ambiguous. In some cases, threads enter the mutex, update the counter, and exit the mutex. But when a thread arrives that forms a complete set, it has to keep the mutex in order to bar subsequent threads until the current set have invoked `bond`.

After invoking `bond`, the three threads wait at a barrier. When the barrier opens, we know that all three threads have invoked `bond` and that one of them holds the mutex. We don't know *which* thread holds the mutex, but it doesn't matter as long as only one of them releases it. Since we know there is only one oxygen thread, we make it do the work.

This might seem wrong, because until now it has generally been true that a thread has to hold a lock in order to release it. But there is no rule that says that has to be true. This is one of those cases where it can be misleading to think of a mutex as a token that threads acquire and release.

5.7 River crossing problem

This is from a problem set written by Anthony Joseph at U.C. Berkeley, but I don't know if he is the original author. It is similar to the H_2O problem in the sense that it is a peculiar sort of barrier that only allows threads to pass in certain combinations.

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

As each thread boards the boat it should invoke a function called `board`. You must guarantee that all four threads from each boatload invoke `board` before any of the threads from the next boatload do.

After all four threads have invoked `board`, exactly one of them should call a function named `rowBoat`, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.