Synchronization Problem Solutions

1. Building H_2O
   a. Correctness constraints
      i. Each hydrogen thread waits to be grouped with one other hydrogen and oxygen before returning
      ii. Each oxygen thread waits for two other hydrogens before returning
      iii. Only one thread access shared state at a time
   b. There is only one condition any thread will wait for, i.e. a water molecule being formed. However, it will be necessary to signal hydrogen and oxygen threads independently, so we will use two condition variables, `waitingH` and `waitingO`.
   c. It will be necessary to know the number of hydrogen and oxygen threads in the monitor. But it would be more useful to know how many hydrogen and oxygen threads have been assigned and have not been assigned to water molecules; let these be `int wH` (number of waiting hydrogens), `wO` (number of waiting oxygens), `aH` (number of assigned hydrogens), and `aO` (number of assigned oxygens). These are all initialized to 0.
   d.
```
Hydrogen() {
  lock.acquire();
  wH++;

  // while not allowed to leave
  while (aH == 0) {
    // try to make a water molecule
    if (wH >= 2 && wO >= 1) {
      wH-=2; aH+=2;
      wO-=1; aO+=1;
      waitingH.signal();
      waitingO.signal();
    }
    // else wait for somebody else to
    else {
      waitingH.wait();
    }
  }
  aH--;
  lock.release();
}

Oxygen() {
  lock.acquire();
  wO++;

  // while not allowed to leave
  while (aO == 0) {
```

```
      // try to make a water molecule
      if (wH >= 2 && wO >= 1) {
        wH-=2; aH+=2;
        wO-=1; aO+=1;
        waitingH.signal();
        waitingH.signal();
      }
      // else wait for somebody else to
      else {
        waitingO.wait();
      }
    }
    aO--;
    lock.release();
  }
```

2. FIFO Semaphores
    a. Correctness constraints
        i. The counter is always non-negative
        ii. Every call to V increments the counter or wakes up a thread in P
        iii. Every call to P decrements the counter or puts the current thread to sleep
        iv. If other threads are waiting to be signalled when P is called, the current thread is put to sleep
        v. Only one thread accesses shared state at a time
    b. Threads will wait for only one thing, namely a call to V (call the condition variable `waitingForV`).
    c. It will of course be necessary to track the value of the semaphore (`unsigned counter`). It will also be necessary to track the number of threads waiting on `waitingForV` (`int waiters = 0`).
    d.
```
P() {
  lock.acquire();
  if (counter == 0 || waiters > 0) {
    waiters++;
    waitingForV.wait();
  }
  else {
    counter--;
  }
  lock.release();
}

V() {
  lock.acquire();
  if (waiters > 0) {
    waiters--;
    waitingForV.signal();
  }
```

```
    else {
      counter++;
    }
    lock.release();
  }
```

3. River Crossing
    a. Correctness constraints
        i. People board the boat using `BoardBoat()` such that situations with 3 employees and 1 hacker, or vice versa, do not occur
        ii. When the boat is full, one of the people in it calls `RowBoat()`
        iii. Only one person can access the boat's state at a time
    b. Hackers and employees outside the boat will wait to get in using condition variables `waitingToBoardH` and `waitingToBoardE`. People inside the boat will wait for the boat to leave using condition variable `waitingToRow`.
    c. First, it will be necessary to know the number of hackers waiting to be assigned to a boat (`int wH`) and the number of employees waiting to be assigned to a boat (`int wE`). It will also be necessary to know the number of hackers that haven't yet boarded but have been assigned to a boat (`int aH`) and the number of such employees (`int aE`). Finally, the last person in the boat needs to row it, so it's necessary to know the number of people in the boat (`int inBoat`). All these variables are initialized to 0.
    d.
```
HackerArrives() {
  lock.acquire();

  wH++;

  // while not allowed to board
  while (aH == 0) {
    // if there's another hacker, and room for 2, allow
2 to board
    if (inBoat+aH+aE < 4 && wH >= 2) {
      wH-=2; aH+=2;
      waitingToBoardH.signal();
    }
    // else wait for somebody else to find a pair
    else {
      waitingToBoardH.wait();
    }
  }

  // board
  aH--;
  BoardBoat();
  inBoat++;

  // if the last person, wake everybody else up and
```

```
leave
  if (inBoat == 4) {
    waitingToRow.broadcast();
    RowBoat();
    // new boat is empty, let everybody run for it
    inBoat = 0;
    waitingToBoardH.signal();
    waitingToBoardH.signal();
    waitingToBoardE.signal();
    waitingToBoardE.signal();
  }
  // else wait for last person
  else {
    waitingToRow.wait();
  }

  lock.release();
}
```

The code for `EmployeeArrives()` is completely symmetric.