Process synchronization using semaphores: The "Cavemen" problem A pre-historic cave with magnificent wall paintings has a very narrow entrance that can only let one visitor in/out at a time. Complete the following pseudo-code to ensure that there will never be more than fifteen (15) people in the cave. The code is executed by each cave visitor process. Filling in the boxes with either semaphore initial values or semaphore calls (`wait()` or `signal()`).

semaphore s1 = ☐

semaphore s2 = ☐

```
Cave_exploration ( ) {

        ☐   // this box may contain one or two calls

        Enter_the_cave ( );

        ☐   // this box may contain one or two calls

        Look_at_the_paintings ( );

        ☐   // this box may contain one or two calls

        Exit_the_cave ( );

        ☐   // this box may contain one or two calls

}
```

Thread synchronization using semaphores: The Reader-Writer Problem Consider the following *one-writer many-readers* problem:

"There is one writer thread and multiple reader threads accessing the same file. When a reader is reading, other readers are allowed to proceed directly while the writer must wait. When the writer is writing, no reader can access the file"

An implementation using semaphores is given below:

```
int readcount;          // shared and initialized to 0
sema_t mutex, wrt;      // both initialized to 1

// For the writer              // For each reader
wait(wrt);                  wait(mutex);
write the file;                  readcount ++;
signal(wrt);                if (readcount == 1)
                                      wait(wrt);
                                 signal(mutex);
                                 read the file;
                                 wait(mutex);
                                 readcount --;
                                 if (readcount == 0)
                                      signal(wrt);
                                 signal(mutex);
```

(1) Does the above implementation realize mutual exclusion between reader and writer?

(2) There is a possibility that the above implementation leads to starvation (i.e., thread waiting indefinitely within semaphore)? Describe a *specific* situation in which starvation will occur.

Thread synchronization using semaphores: The "H_2O" problem: You've been hired by Mother Nature to help with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom thread executes a procedure `hReady()` when it is ready to react; and each O atom thread invokes a procedure `oReady()` when it is ready.

Your job is to write the code for `hReady()` and `oReady()`. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the procedures must call the procedure `makeWater()`. After the `makeWater()` call, two instances of `hReady()` and one instance of `oReady()` should return.

Your solution should avoid *starvation* and *busy-waiting*. You may assume that the semaphore implementation enforces the FIFO policy for thread wake-ups – the thread waiting the longest in `wait()` always grabs the semaphore after a `signal()`.

(1) Consider the following implementation. Does it work? Briefly explain your answer. If it doesn't work, show how to fix it.

```
int numHydrogen = 0;
sema_t pairOfHydrogen = 0;
sema_t oxygen = 0;

void hReady() {
     numHydrogen ++;
     if ((numHydrogen % 2) == 0) {
          signal(pairOfHydrogen);
     }
     wait(oxygen);
}

void oReady() {
     wait(pairOfHydrogen);
     makeWater();
     signal(oxygen);
     signal(oxygen);
}
```

(2) Now consider a different implementation below. Does it work? Briefly explain.

```
sema_t hPresent = 0;
sema_t waitForWater = 0;

void hReady() {
     signal(hPresent);
     wait(waitForWater);
}

void oReady() {
     wait(hPresent);
     wait(hPresent);
     makeWater();
     signal(waitForWater);
     signal(waitForWater);
}
```

(3) Suppose the OS wakes up the thread that stays the *shortest* time in the semaphore queue (namely last-in-first-out or LIFO) upon `signal()`, will the implementation in **(2)** still work?