Project 2: Thread Synchronization

You will write 2 programs in C for this assignment, working with your same partner from project 1. You should submit a zip file containing the 2 C programs. Make sure both partner names are in comments at the top of each file, and that you use good coding standard in general!

Part 1 - Building H_2O_4S Molecules

You will use only **semaphores** as the synchronization methods for this problem. See the example program semtest.c for a demonstration of the 3 semaphore methods you can use (sem_open, sem_wait, and sem_post). You should not use any other methods in semaphore.h, they are not even supported on the mac (e.g. sem_init or sem_getvalue). Remember you can use a semaphore as a basic lock for mutual exclusion as well as counting semaphores.

A molecule of sulfuric acid is composed of two hydrogen atoms, one sulfur atom, and four oxygen atoms. In this project, you will be writing the synchronization needed for three different types of threads to interact in a manner similar to how these atoms interact to form a molecule of sulfuric acid.

A particular system contains threads of three different types, which we call hydrogen, sulfur, and oxygen. At some point these threads must interact with one another in a specific way: two hydrogen threads, one sulfur thread, and four oxygen threads must be present for the interaction to occur. If any are missing, the remaining ones **must wait** until the missing threads arrive at some later time.

You will create a program in H2S04.c to implement this synchronization. The program will contain at least the following three functions:

```
        void hydrogen();
        void sulfur();
        void oxygen();
```

When a thread reaches the point where it wants to interact with other threads it calls the function corresponding to its type. As each new thread arrives, your class should check to see whether there are now enough threads to form an H2SO4 group. If so, all threads in the group are allowed to depart; otherwise the newly arriving thread is blocked. There is no limit to the number of threads that might be waiting.

There is one additional requirement: when a group of threads departs, the hydrogen threads must leave first, followed by the sulfur thread, followed by the oxygen threads. However, if there are more than the required number of threads of the same type waiting, it makes no difference which of the waiting ones is unblocked.

Note that the sample program semtest.c would not meet the requirements for this project even to form just a molecule of H20, as the hydrogen atoms do not wait for a whole molecule to be completed, only the oxygen atoms wait until at least 2 hydrogens have been produced.

You should test your program in a main() function of your own, but when graded my own tests will be substituted for yours, so make sure you have clear comments as to where any test code that creates & joins threads could be placed (e.g. you should open all semaphores for use first thing in the main program).

Part 2 - Boating Oahu to Molokai

You will use **condition variables** and locks as the synchronization methods for this problem. The example program conditiontest.c demonstrates how to use these with pthreads.

A number of Hawaiian adults and children are trying to get from Oahu to Molokai. Unfortunately, they have only one boat which can carry maximally two children or one adult (but *not* one child and one adult). The boat can be rowed back to Oahu, but it requires a pilot to do so. Arrange a solution to transfer everyone from Oahu to Molokai. You may assume that there are at least two children.

The method beginTransfer(int nChildren, int nAdults) should create a thread for each child or adult. Your synchronization mechanism cannot rely on knowing how many children or adults are present beforehand, although you are free to attempt to determine this among the threads (i.e. you can't pass the nChildren or nAdults values to your threads or set them as globals, but you are free to have each thread increment a different shared variable, if you wish).

Your solution must have no busy waiting, and it must eventually end. Note that it is not necessary to terminate all the threads -- you can leave them blocked waiting for a condition variable. The threads representing the adults and children cannot have access to the numbers of threads that were created, but you may use these number in begin() in order to determine when all the adults and children are across and you can return.

The idea behind this task is to use independent threads to solve a problem. You are to program the logic that a child or an adult would follow if that person were in this situation. For example, it is reasonable to allow a person to see how many children or adults are on the same island they are on. A person could see whether the boat is at their island. A person can know which island they are on. All of this information may be stored with each individual thread or in shared variables. So a counter that holds the number of children on Oahu would be allowed, so long as only threads that represent people on Oahu could access it.

What is not allowed is a thread which executes a "top-down" strategy for the simulation. For example, you may not create threads for children and adults, then have a controller thread simply send commands to them through communicators. The threads must act as if they were individuals.

Information which is not possible in the real world is also not allowed. For example, a child on Molokai cannot magically see all of the people on Oahu. That child may remember the number of people that he or she has seen leaving, but the child may not view people on Oahu as if it were there. (Assume that the people do not have any technology other than a boat!)

You will reach a point in your simulation where the adult and child threads believe that everyone is across on Molokai. At this point, you are allowed to do one-way communication from the threads to begin() in order to inform it that the simulation may be over (e.g. change the value of a global variable).