Project 2: Thread Synchronization - part (b)

Condition Variables - Island Crossings

You may work with a partner of your choice on this project, or work alone. You will use **condition variables** (and the associated locks) as the synchronization tools for this problem. The example program condvar.c demonstrates how to use these with pthreads. You may NOT use semaphores.

deliverables: boat.c with your modifications - will be compiled and run using the provided boatMain.c and boat.h. Make sure your solution works with the provided files, and use good coding standard in general.

A number of Hawaiian adults and children are trying to get from Oahu to Molokai (how I wish I was one of them). Unfortunately, they have only one small boat that holds at most two children or one adult (but *not* one child and one adult). The boat requires a pilot to row it in either direction between islands, you can't send people to Molokai and then magic the boat back to Oahu. Arrange a solution to transfer everyone from Oahu to Molokai.

You may assume that there are at least two children to start on Oahu, and that the boat is initially docked there.

You must use the files I have provided. You may NOT modify boatMain.c or boat.h (you could temporarily for testing, but you won't even submit these so your boat.c must work with the originals). You should keep the starting code in boat.c that initializes the shared variables with the main program. You can do whatever you want in the thread functions, as long as your solution works and correctly signals to main when there are no people left on Oahu that it can exit (your program should not hang unless you start it with <2 children). However, the initial code given in each thread function works well to ensure that the correct starting counts of children and adults on Oahu are calculated before anyone decides how to act, which will highly likely be important to your solution, so keep that there unless you are sure you want to change it. If you do change it you still have to somehow signal main that you are ready to move on...

You may add as many global variables as you wish to boat.c, including shared data, more locks, and more condition variables. You should initialize anything that needs it in the init() function. You may NOT use any semaphores, and remember you may not add to or modify boat.h.

You must use the provided functions boardBoat, boatCross, and leaveBoat with the correct arguments whenever a thread takes one of those actions. A thread MUST call one of those functions when it takes the corresponding action, and MAY ONLY make a call when it is actually taking that action itself (e.g. a child can't call boardBoat with "ADULT" as the argument, or a person on Molokai can't call boardBoat with "OAHU" as the argument). You should not have any other print statements in your final solution, though you may use lots while testing!

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 as long as you exit the main process.

Included in the materials is a sample output file with a couple of possible correct print outputs. Your solution doesn't have to be as absolutely efficient as possible, as long as no one does anything illegal (e.g. exits the boat without first boarding the boat, or an adult boarding the boat when a child is already on it), and everyone makes it eventually.

You can test what the provided code does right now, by compiling with "gcc boatMain.c boat.c" and running with "./a.out nAdults nChildren" where nAdults is an integer between 0 and 9, and nChildren is an integer between 2 and 9. If you want to include optional output you can put it in an "if (verbose)" statement and it will only be printed if you include an extra command line argument of 1 (e.g. "a.out 2 5 1").
The program currently creates all of the threads, makes sure all of them have run through their initial code so that the number of children and adults on Oahu is initially correct, then lets them all procede. Each simply boards the boat, crosses to Molokai, and exits the boat, as if there were infinite boats waiting on Oahu and each person can just take their own. Once the count of people on Oahu is correctly decremented to 0, the main program finishes, whether or not the threads themselves have finished (this way you can leave threads sleeping in queues on Molokai without hanging the program).

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, a person could see whether the boat is at their island and whether anyone is in the boat, but not who is getting in/out of the boat at the other island. What is not allowed is a thread which executes a "top-down" strategy for the simulation. For example, you may not have a controller thread simply send commands to the child/adult threads through whatever communicators. The threads must act as if they were individuals. I added the unchangeable starting code to make these constraints hopefully more clear, but if you are at all unsure if what you're doing is valid, just ask!!!