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

For this question I have provided you with a .h file that you may NOT modify and you MUST include in your solution program, using the same syntax as the include in my H2SO4Test.c file. You can think of this .h file as similar to an interface in Java - you must implement all of the functions in the .h file in your solution program, using the function names, parameters, and return types exactly as given. You should name your solution program H2SO4.c. You can first simply add a function stub to your .c file for each of the 5 headers in the .h file, where all you do is print what function you are in and return something of the correct type. This will allow you to compile and run so that you can test as you go (highly recommended). To compile the overall program you simply list both .c files after gcc, e.g. "gcc H2SO4.c H2SO4Test.c –o molecules", and make sure both .c files and the .h are all in the current directory. You don't need to modify the test program, and in fact you will only turn in your solution .c file, but you should check out the code given in the tests, as there's lots of useful stuff, like command line arguments, and creating pthreads, and more!

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 from semaphore.h, as they may or may not be supported depending on your OS (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. Each thread type corresponds to one of the 3 atoms and the program for each atom type should go in the function with the same name. The very first thing you should do in each of the 3 functions is to output that this atom has been produced. After that the algorithm for each atom thread may differ, but the general idea will be to use semaphores to synch the atoms into forming molecules as described below. You may create as many semaphores as you wish (I use several in my solution), and should declare these as global variables so all threads can access them. You should open all used semaphores in openSems and close/unlink them all in closeSems.

In order to form a molecule of H2SO4 two hydrogen threads, one sulfur thread, and four oxygen threads must be present. If any are missing, the remaining ones **must wait** (efficiently, no busy waiting) until the missing threads arrive at some later time.

As each new thread arrives (is created), your program should check to see whether there are now enough threads to form an H2SO4 group. If some required atoms are still missing, the newly arriving thread should be blocked on some semaphore's wait queue. There is no limit to the number of threads that might be waiting (e.g. you might produce 100 Hydrogen atoms, all of which will have wait, before you produce any Oxygen or Sulfur atoms).

If all required atoms are present, you should PRINT that a molecule has been formed, and allow all threads in the molecule to depart, again PRINTING when each one leaves. The "molecule formed" print must come before all of the "leaving" prints, and there is an 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 new threads may arrive and print their "produced" statements interleaved with the "leaving" statements of the just-formed molecule, however you should not allow the creation of the next molecule until all current atoms have left, even if all necessary atoms are present.

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.

Run my example solution executable with several different arguments to see what the output should look like. The simplest example is to create exactly enough of each atom to form a single molecule, so try that first. A C executable is compiled for a specific platform (i.e. version of an OS) and is rarely compatible with any other, so I will compile my solution on a lab computer and you will likely have to run it on a lab computer.

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. You may NOT use semaphores except for the specific start and end circumstances described below.

A number of Hawaiian adults and children are trying to get from Oahu to Molokai (oh how I wish I was one of them). Unfortunately, they have only one 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, sorry =) 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 docked there.

You can write your solution in a single .c file, but your main() function definition should be the first defined function in your file, which means you will have to declare your other functions above main, but then define (implement) them after main. I will replace at least most of your main with mine as part of the testing when grading so make sure you follow the instructions given here precisely for what should go in main and what the rest of your functions should look like (name, arguments, return, etc.).

You should have 2 thread methods, named child and adult, with the usual void* type for both the argument and return. These methods must include appropriate print statements that describe in detail anytime a person takes an action. The statement should state whether the person is a child or adult and the action being taken. Actions should include

- "showing up" on Oahu (first thing in each thread, only occurs once upon thread creation)
- getting into boat (should include which island on)
- rowing boat (should include which island rowing from and which rowing to)
- getting out of boat (should include which island on)

Your main method should

- read the number of children and the number of adults as command line arguments
- call the initSynch() method – initSynch() should open/initialize all of your locks/condition-vars/semaphores
- create a thread for each person, so a total of (num children + num adults) threads. Each thread should have NULL as the 2nd and 4th arguments, a unique address as the 1st argument, and the appropriate function (child or adult) as the 3rd argument.
- somehow wait until all threads have started and then notify threads that they may begin crossing the ocean (see below for more detail) – you may use a semaphore for this
- somehow wait until all people have arrived on Molokai before exiting – you may use a semaphore for this as well

Your main method should NOT access any global variables except synch types (mutex, cond, sem). This means you cannot for example have main (or any non-thread method it calls) set as global the total number of children/adults. Your threads will have to work this out for themselves.

You may have as many global variables as you wish shared among threads, but they should only be accessed by threads, which means only in the child or adult methods.

You may have as many locks and condition variables as you wish, and these can be accessed from anywhere. You may not use semaphores, except for the 2 special cases of

- making sure no threads start to cross the ocean until all threads have arrived on Oahu
- making sure the main process does not exit until all threads have arrived on Molokai

You will have to figure out how exactly to use semaphores for these 2 purposes, but the general idea is that each thread when it first runs will do any init stuff needed and then somehow sleep/block in a wait queue. Then the main() function will somehow realize that all threads have started running and wake all of those sleeping up (note that a thread does not immediately run when created, it simply gets put in the ready queue, so just because main() finishes creating all threads does not mean they have all had a chance to run). This is necessary for any solution to work (see *footnote at end for more on why). After main() wakes all of the threads on Oahu, it should go to sleep/block in a wait queue until the last thread arrives on Molokai, at which point that last thread must somehow realize it's last and wake main() so that the process can exit.

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.

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 know which island they are on and see how many children or adults are on their same island, but they shouldn't be able to see the other island. A person could see whether the boat is at their island and whether anyone is in the boat. Any/all of this information may be stored with each individual thread or in shared variables. So a global variable 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 have a controller thread simply send commands to the child/adult threads through whatever communicators. The threads must act as if they were individuals.

*Why must wait for all threads to at least start running before let any cross ocean – the minimum guarantee for this problem is that there will be at least 2 children and the boat on Oahu. This means somewhere in your solution you must account for the case that there are exactly 2 children (and no one else) with the boat on Oahu, at which point obviously the 2 children should row the boat to Molokai and signal that everyone is across. As stressed many times in the problem description above, each thread can only have local information, so if a child thread on Oahu looks around and sees no one except one other child thread, and sees the boat, they should get in the boat and wait for the other child who also sees the same thing to get in as well, then they should both row to Molokai and signal everyone is across. This means that if main() planned to create 10 children and 10 adults, but happened to be interrupted after creating the first 2 children, and those 2 child threads each got to run before main() ran again creating more threads, the 2 children would do exactly what's described above, as they have no way to know that more people will be popping onto Oahu later, and once they've rowed away they will never see those people as they will assume they are done. So – we must guarantee that all threads have actually arrived on Oahu before any threads are allowed to leave!