This assignment is designed to work with threads and global shared data. That also means synchronization of access and modification of the global data. To make this a bit more interesting, we will be simulating a hypothetical problem at Lakewood, WWU’s waterfront facilities on Lake Whatcom.

Lakewood rents kayaks, canoes and sailboats for use on the lake. (In reality, they have more than just these three, but we will limit it to just these 3.) When they are rented, the renters need to have life jackets on while on the water. We are going to assume that the life jackets are the critical resource. When a renter uses a kayak, one life jacket will be needed for the one person kayak. When a canoe is rented, two life jacket will be needed. When a sailboat is rented, four life jackets will be needed. (Our rental groups are very uniform!) The problem is that only 10 life jackets are available. So, two sailboats and one canoe will utilize all life jackets and future rentals must wait for previous rentals to return the life jackets after they are done using them. (Yes, I know, Lakewood must have a lot more than 10 life jackets.)

So, when all life jackets are in use, the renters must get in line and wait for enough life jackets to be returned. Also, we want the renters to be serviced in the order they arrived. So, for example, if a sailboat is to be rented and there is less than four life jackets available, they must wait until enough life jackets are returned to allow the sailboat to take four. Even if the renter behind the sailboat wants a kayak and needs only one life jacket, they can’t get it until after the sailboat gets theirs, even if there are available life jackets, just not the four needed for the sailboat.

Finally, the maximum in line should be 5 renters. If the line has 5 renters waiting, a new renter will just look at the line and decide to go away.

You are to simulate this situation. Here are the details:

- Maximum of 10 life jackets.
- Three water craft, a kayak, a canoe and a sailboat.
- Life jackets needed are one for the kayak, two for the canoe and four for the sailboat.
- Your main() function must spawn a thread for each renter.
Each thread represents a group of renters.

Each thread randomly selects one of the three water crafts.

Each thread randomly selects how long to use it, a value from 0 to 7 time units. You will use the sleep(3) library call to cause a thread to “use the craft” for that length of time.

Each thread should do the following:

1. Print out the group number, the water craft requested and the number of life jackets needed.
2. Check to see if there are enough life jackets available.
3. If no life jackets are available then the group should wait. If less than 5 groups are waiting, add the group to the waiting queue and report they are waiting. (This should block the thread also.) If 5 or more groups are waiting, report that the group will not wait and exit the thread.
4. When enough life jackets area available, report that the group will be using them and the remaining available life jackets.
5. Call sleep with the “use time” for this group.
6. Report the return of the life jackets and the number of life jackets available.
7. If there is a queue, unblock as many as possible without running out of life jackets. This could be anywhere from zero to the entire queue.

The main function should process argv as follows:

1. argv[1] is the number of groups to generate.
2. argv[2] (optional) is a number, which is the rate for new groups to arrive. The default number is 10. Take this number and divide by 2 and then use a random number between zero and this number as the value for sleep(3). (You may use nanosleep(3) to get partial seconds of wait time.) Your main thread should sleep this random amount of time between creating threads. (You must allow having only two argv. Having an argv[2] does not require an argv[3].)
3. argv[3] (optional and must have argv[2] to get argv[3]) Initialize the random number generator with “time(NULL)”. The default value should be 0. (srandom(0); or srandom(time(NULL));)
• The loop that creates new threads should join threads that have finished.

• After finishing creating all threads, the main function should join all remaining threads so that all threads run to termination.

Notes:

• You need to create a directory “lakewood” in your repository on the same level as the “ush” directory, NOT in your ush directory. You MUST have a single file named “lakewood.c” in the lakewood directory. When you have it completed to your satisfaction, create the branch “a5” and push it as normal. Do not create a5 until you are ready to turn in your program.

• You may start with the program “pth-mutex.c” as shown in class videos.

• Test your program incrementally. Don’t write the entire program and then try to debug it. Get the thread creation and termination working before you worry about shared data and synchronization.

• If you haven’t done linked lists in C, I have created a simple queue program using dynamic memory. /home/phil/public/csci347/queue.c is the name of the program on the Linux machines. Use it as an example. DO NOT USE the code directly in your work.

• Using a condition variable per thread could be useful.

• Using at least one condition variable is required.

• You may run my solution: /home/phil/.bin/lakewood Running it with the arguments of 20 and 5 can show you acceptable output. Arguments of 20 and 3 can show you when customers arrive too fast. Arguments of 20 and 8 still shows minimal queuing.

What to turn in:

• Your source code.

• A run as follows: “./lakewood 20 5 r”

• Don’t forget your cover sheet and turn in all of this as a .pdf on canvas.