Programming Assignment 2

Michael Ortiz

**Problem 1**

My solution to this problem relies on the fact that it is very similar conceptually to the prisoner problem we discussed in class. At a high level it can be solved as follows: One guest will be the one to let minotaur know that everyone has had cake. Each guest will eat cake once, and only once. For each guest, if there is cake and that guest has yet to eat some, he/she will eat it and move on. In all other circumstances the guest will ignore the cake. When the counter comes in, he/she will refill the cake and take note of it, but he/she will not eat it. Otherwise, the counter will ignore the cake. Once the counter has refilled the cake N-1 times they can be certain that everyone has had something to eat and will announce to the minotaur as such.

In order to simulate this puzzle, I wrote a class “labyrinth.cpp” that contains an array of Booleans where the ith element represents the whether the ith guest has had cake. In main, within a loop that runs until every guest has had some cake, on each of iteration a guest is randomly selected and an available thread is summoned from my naïve implementation of a basic thread pool that calls the “traverseLabyrinth()” function from my class. The logic described above is run within this function that is protected by a mutex.

In the long run this solution is guaranteed to work, though it gets much slower as N increases. For N = 100 it runs in about half a second on my machine, for N = 200 it runs in about 2.5 seconds and the performance continues to grow worse. I have included a graph of runtimes in the submission folder.

**Problem 2**

For this problem I chose to simulate the third solution. One main advantage of this approach is that it avoids the potential massive performance issues of the first solution in which many threads could be doing nothing for a long time while they wait to access the vase (which they may never get), this is analogous to the issues of a TAS lock. Another advantage is that it also has superior performance to solution 2 which is analogous to a backoff lock which has scaling issues for large numbers of guests (threads). It is for these reasons that I believe solution 3, which is most analogous to a queue based lock, is superior. However, the main disadvantage of this approach is that it uses a lot of space for a large number of guests.

In order to simulate this solution, I made use of thread suspension and ticks. The user enters a set number of ticks and for each tick the following is done: A guest is randomly chosen to enter the vase room on a separate thread and that thread sleeps for 2 seconds to represent the time taken to view the vase. Meanwhile, on the main thread, while the room is occupied, guests are randomly placed into a queue to view the vase next. For each guest queued the main thread also calls sleep for a short and random amount of time in order to introduce variation. By the end of the tick, the guest who was viewing the vase is returned to the list of potential guests to view the vase. Repeat for designated number of ticks.

**How to run**

Compile in a linux environment with the following command: g++ main.cpp vase.cpp labyrinth.cpp -pthread and run with ./a.out. Once the program is running it provide directions at the command line to run each of the individual problems.