Camilo Lozano
9/28/20

Programming Assignment 2: Minotaur

Problem 1:

The key to solving this problem is having a designated primary guest, or captain per se. The captain will keep tally of how many people have eaten a cupcake. The captain oversees replenishing the cupcake whenever he comes into the room. The other guests just must eat the cupcake the first time they are in a room when there is a cupcake available, they must not request a new one, and only do this once. This allows for when the captain comes back into the room for them to be able to keep tally of the amount of people that have been in the room. Once the captains internal count reaches N-1 (where N is a positive integer representing the number of guests). Even though this algorithm is not the most efficient since it does not guarantee success when all guests have at least visited once but it does work in a concurrent system.

The implementation in Rust relies uses a Guest Enum which represents what time of guest they are either the primary (which is the captain) or secondary which is everyone else. Both of those implement the method visit which is an iteration of going into the cupcake room and deciding if they are to eat the cupcake, get a new one or pass. The main thread represents the minotaur where they select a random guest to visit the cupcake room. The cupcake is represented by a mutable Boolean variable that is shared between the threads. Only to be accessed by the current guest in the room only. Once their visit is done the thread returns ownership of the variables back to the minotaur and the next round starts. The program spits of the guest list at the end, which is an array of Guest Enums, the last one is the captain and it will show its internal counter that everyone has been accounted for.

Problem 2:

For this problem I concluded which the last strategy for managing the guests visiting the vase, but each option has its pro's and con's

1) The problem with this implementation is two-fold. Every thread is going to be waiting for the door to unlock, meaning that there is going to be many wasted cycles of each thread just waiting and checking on one variable to see if it is open or not, then once it unlocks you'll have every thread trying to jump in at once which leads to the second problem. There is no guarantee that any of the guests will be able to visit the room. Depending on the implementation if thread 1 finishes and decides to line up again it could get priority over other queues and it would be unfair to the other guests. The only advantage to this method is that it would be the easiest to programmatically implement requiring almost no coordination between

the threads to communicate with each other. This method is most closely related to the Test-And-Set-Locks from chapter seven.

2) This method suffers similar downfalls to the first method when it comes to guarantees and wasted cycles. The biggest difference I see between the implementation for these two methods is that in this one the threads will be reading from a variable to see if it is busy and then set some sort of sleep between interval checks, ideally a random sleep so each thread can have more of chance to go in the room. This would improve on the TASLock method as you do not have all threads in trying to get in at the same time. In this case the thread oversees changing back the lock to available. This method would be like the TAS-Based Spin Locks Revisited.
3) The main benefit with this method is that there is an established queue and defined ordering of threads. Doing this prevents randomness from the last two algorithms giving every guest a potential fair chance in getting access. The idea is for each thread once they are done to be able to directly tell the other thread that it is their turn next in line to go ahead. This is a much better method as this can prevent wasted cycles on every thread checking one specific variable between all of them. Rather it would just check on its on flag, and if implemented properly (with something like cache line buffers so cache doesn't get invalidated on each thread changing the flags) we should be able to prevent cache invalidations which would cause each thread going down further the cache hierarchy causing even more delays. This method would be like an Exponential Backoff.

I implemented the third algorithm before the lecture on this so my implementation is different on how ideally it should be, but it was surprisingly similar, and I get close to how this one algorithm works. This is implemented with Rust channels, channels are the best way in Rust to send information between threads, depending on the individual implementation it can be single producer multiple consumers, multiple producer single consumer, etc. Each thread gets its own channel (which is a pair of sender/receiver), and the thread waits on its receiver channel. When it receives the signal it then acts goes in the room and calls the next one. Since there is an ordered queue, the current thread in the room knows who is next, once it is done visiting the thread has a list of every other threads sender channel and then it sends the rest of the queue order to the next thread.

What's nice about this method is that since it uses channels (specifically the crossbeam channel package) it implements the most efficient CPU cycle waiting method for every thread without allowing the possibility of multiple threads being in the room at once. While the queue is generated sequentially at the beginning of the program it technically should be able to expand mid-way through although that is not implemented. Additionally with this method it makes it easy to have repeat visits for each thread, and since it is known when the queue is empty when the last thread receives the queue it is easy to know when the algorithm is done