Homework Assignment 2: Ping Pong Game

Michael Edelman

October 8, 2021

1 The Problem

The game needs a strictly alternating sequence of PINGS and PONGS. When we instantiate a PING thread and a PONG thread each thread will be executing concurrently and will take its "turn" whenever the Operating Systems Scheduler has scheduled those actions. However, each thread is executing and taking its turn *completely independent* of the other thread. This makes no sense because each thread can only take its turn after the other thread took its turn. When thread one does PING only then can thread two do PONG and once Thread two does PONG thread one can go again and do a PING. In order to achieve this the threads need to communicate to each other when the other can go.

2 The Solution

In order to solve this problem the threads need to communicate to each other when the other can go. This can be achieved in the implementation of the aquire() and release(). aquire() will aquire permission (from the other thread) and then the thread will take its turn. After, release() will give permission to the other thread to take its turn. The implementation centers around a static variable *whosTurn*. when a thread takes its turn it changes this variable to the the other type. So for example if the PING thread took its turn it would then (by calling the release()) set *whosTurn* to Type.PONG. On the flip side, a Thread will not take its turn until *whosTurn* equals its own Type. So for example a PING thread will wait (by calling the aquire() which uses a busy loop) until *whosTurn* is set to Type.PING by the PONG thread.

Thus this mechanism accomplishes exactly what we need by creating communication between the threads and particularly dependency between them where a PING thread can't take its turn until the PONG thread did or vice versa. However there is actually an additional issue that will make this implementation still fail. That is that often times data is copied to caches to achieve quicker access. However this means two threads may be reading and updating the copies of the data and not the datum itself and therefore data inconsistency can arise. Imagine if the PING and PONG thread had a copy of *whosTurn*. If one of the threads changes its copy to notify the other thread to take its turn - the other thread will never know because that change is not reflected on its own copy. Thus enters the **volatile** keyword. By adding this modifier on *whosTurn* we ensure that *whosTurn* is stored in main memory and thus both threads are refering to the exact same piece of data and can effectively communicate. **volatile** also ensures read and writes are atomic and the compiler doesn't reorder the code to optimize it.

Note: You might have noticed a bootstrap issue - if the threads need the other to take its turn - then how do we start at all? This is easily dealt with by setting *whosTurn* to Type.PING from the start. Therefore the Type.PING thread executes first.