Homework 07
Allie Duncan and Kyle McConnaughay

Experiment 1:

Summary:

We set out to discover how often synchronization errors occur and how to fix them using locks. First, we ran two concurrent threads that both updated the same variable. We then implemented an incorrect locking system and a correct locking system. During all three parts of the experiment we recorded the frequency of concurrent reads. For the initial section, with no variable protection, we were surprised at how frequent concurrent reads occurred. We expected a much lower result. From this, we can infer that synchronization is an issue that must be addressed even in simple programs. Also unexpected was how ineffective the broken lock implementation was. There was a difference, but the difference was not as distinct as we expected.

Details:

In example.c, we added two for loops: one in entry( ) and one in main( ). In each loop, the env counter was incremented and then the thread slept for one second. The sleep( ) function call was added to help us see and understand how the threads interleaved., We then had the child and parent loop for one hundred times each, which enabled us to characterize how often synchronization errors occurred. From five data samplings, concurrent reading occurred an average of 12.4% of the time, with a standard deviation of ±5.4%.

When using the broken 'lock' implementation, the number of concurrent reads decreased, but did not achieve complete mutual exclusion. In periods when both threads were waiting to acquire the lock it, sometimes they simultaneously acquired the lock. This allowed for concurrent reading and incrementing of the counter. On average, concurrent readings with the broken lock occurred 5.86%, with a standard deviation of ±2.2%

When using the assembly code implementation, we executed:

```
> gcc -m32 -Wall oldexample.c lock.x86.s -o example -lpthread
```

where oldexample.c is the broken lock implementation (without make_lock, acquire, and release – which is provided by lock.x86.s). However, the script deadlocks. We assume that this implementation would work with a queue system, without a while loop.

Experiment 2:

Summary:

We set out to discover how to implement mutexes and how effective they are. By editing lock.c so that is uses mutexes, we can still use example.c to run the threads. Therefore, only minor edits were required. We expect mutexes to work completely (they are designed for the task, after all). And they did! We incremented to 1000 and there were no synchronization errors. From this, we can infer that mutexes are an easy and reliable method to protect shared data.

Details:

The frequency of synchronization errors using the pthread lock implementation is 0%! We cannot compare the efficiency of lock implementation with pthread mutexes because lock.x.86.s is deadlocking. However, we posit that mutexes would be faster (as most build-in functions are faster). The extra time is spent in user code which wastes the operating system's recourses; when the acquire lock is spinning, mutexes just have the thread wait.