## Exercise 07

On the course homepage, you will find source code for a program, critical.c. This program exhibits a race condition when multiple threads try to access a shared data structure at the same time. You're going to fix it by adding a test-and-set spinlock to protect access to the shared data structure.

The program is intended to show the kinds of things that could happen inside the operating system. It maintains a linked list of nodes, using multiple threads to repeatedly remove nodes from the front of the list and enqueue them at the back of the list. This is kind of like what the operating system might be doing if it's using round robin to for CPU scheduling. Across multiple cores, it may be removing processes from the ready queue, letting them run for a time quantum, then putting them at the back of the queue if they're not done yet.

This program makes a linked list of 100 nodes. It makes a user-specified number of worker threads repeatedly dequeue a node from the head of the list, increment a counter in the node and then enqueue the node at the tail of the list. Sharing a data structure like this across multiple threads could be a problem. If multiple threads try to modify the data structure at the same time, they could leave it in an inconsistent state.

The program takes a single command-line argument, the number of worker threads to create. If you don't specify any, the program just creates one new thread, so it shouldn't exhibit any race conditions. When you run the program, you may not get the same total count I get below, but you should still have 100 nodes and a fairly large total count. However, if you run it with multiple threads (especially on a multiprocessor), things will probably go wrong. For example:

```
$ ./critical
nodes: 100  total count: 623111250
$ ./critical 2
Segmentation fault (core dumped)
```

You're going to fix the program by adding a spinlock-based mutual-exclusion solution using a gcc-supported test-and-set operation. The compiler offers a collection of atomic operations, including ones called __sync_lock_test_and_set() and __sync_lock_release(). I'll summarize how to use these below, but you can also find some fairly terse documentation for these at:

http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Atomic-Builtins.html

You're going to use these two operations to implement your entry and exit code in both the dequeueNode() and enqueueNode() function. Whenever a thread needs to access the linked list, it will make sure no other thread is

accessing it at the same time.

- Make a global int variable to serve as your lock. It will have a value of zero when no thread is accessing the linked list and a value of one when a thread is accessing the list.
- For the entry code, write a loop that repeatedly calls __sync_lock_test_and_set(), trying to set the memory where your lock is stored (the first parameter) to the value 1 (the second parameter). The __sync_lock_test_and_set() function returns the previous contents of the memory you're trying to set, so your loop can keep calling it until it returns zero. Then, you know you've successfully changed the lock from zero to 1. If the function returns a 1, then you know some other thread has the lock, so you'll have to keep trying.
- For your exit code, you just need to pass to __sync_lock_release() the memory address where your lock is stored. It will automatically set it back to zero.

Once you've added this code it should correct the race condition. You may want to try it out on a real multiprocessor before you turn it in, just to give it a more challenging test. The added synchronization code will probably slow your program down a good bit, so the total count across all the nodes may be a lot lower.

When your critical.c program is working, submit it to the Gradescope assignment named EX07.