练习 07

在课程主页上,您会找到程序 critical.c 的源代码。这个程序表现出竞争条件
当多个线程尝试同时访问共享数据结构时。您将通过添加测试来修复它
and-set spinlock 来保护对共享数据结构的访问。

该程序旨在展示操作系统内部可能发生的各种事情。它维护一个
节点的链表,使用多个线程重复地从列表的前面删除节点并将它们排到列表的后面。这有点像操作系统在使用循环法来获取 CPU 时可能会做的事情

调度。在多个内核中,它可能会从就绪队列中删除进程,让它们运行一段时间,然后将它们放在队列的后面(如果它们尚未完成)。

该程序制作了一个包含 100 个节点的链表。它使用户指定数量的工作线程重复出列
列表头部的一个节点,增加节点中的计数器,然后将该节点排入列表尾部。
跨多个线程共享这样的数据结构可能是个问题。如果多个线程试图修改数据
同时,他们可能会使其处于不一致的状态。

该程序采用单个命令行参数,即要创建的工作线程数。如果您不指定任何内容,
该程序只是创建了一个新线程,因此它不应该表现出任何竞争条件。当你运行程序时,你
可能不会得到与我在下面得到的相同的总数,但你应该仍然有 100 个节点和相当大的总数。
然而,如果你用多线程运行它(尤其是在多处理器上),事情可能会出错。为了
例子:

$ ./关键
节点:100 总计数:623111250
$ ./关键 2
分段错误(核心已转储)

您将通过使用 gcc 支持的测试和添加基于自旋锁的互斥解决方案来修复该程序
设置操作。编译器提供了一组原子操作,包括称为 __sync_lock_test_and_set() 的操作
和 __sync_lock_release()。我将在下面总结如何使用这些,但你也可以找到一些相当简洁的
这些文档位于:

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

您将使用这两个操作在 dequeueNode() 和
入队节点()函数。每当一个线程需要访问链表时,它会确保没有其他线程是

同时访问它。

- 创建一个全局整型变量作为您的锁。当没有线程访问时,它的值为零
  链接列表,当线程正在访问列表时值为 1。
- 入口代码,写一个循环,反复调用__sync_lock_test_and_set(),尝试设置内存
  您的锁存储位置（第一个参数)为值 1（第二个参数）。 __sync_lock_test_and_set() 函数返回您尝
  试设置的内存的先前内容,因此您的
  loop 可以一直调用它直到它返回零。然后,您知道您已经成功地将锁从 0 更改为 1。如果该函数返回 1,则您知道其他某个线程拥
  有锁,因此您必须继续尝试。
- 对于您的退出代码,您只需将您的锁所在的内存地址传递给__sync_lock_release()
  存储。它会自动将其设置回零。

添加此代码后,它应该会更正竞争条件。您可能想在真正的多处理器上尝试一下
在你上交之前,只是为了给它一个更具挑战性的测试。添加的同步代码可能会减慢您的速度
程序减少了一点,所以所有节点的总数可能会低很多。

当您的 critical.c 程序运行时,将其提交给名为 EX07 的 Gradescope 作业。