Homework 2

CS5460: Operating Systems, Spring 2018 **Jake Pitkin**
University of Utah

6.13 **Chapter 5 of OSC discusses possible race conditions on various kernel data structures. Most scheduling algorithms maintain a run queue, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. Briefly give at least one advantage and one disadvantage of each of these approaches.**

Multiple queue pros: The semaphore operations wait() and signal() must be executed atomically. It's difficult to disable interrupts on every processor (to prevent interleaved operations). But, when each processor has it's own queue interrupts only the processor executing the wait() or signal() operation needs to be disabled. Much more scalable than single queue systems and multiple queues make exploiting cache affinity easier.

Multiple queue cons: Load balancing is required to attempt to keep an even distribution across all of the processors. Moving processes around during the load balancing process will invalidate the cache and lose cache affinity.

Single queue pros: Simple to implement. Can use the same scheduling algorithms as you would use in a single processor setup.

Single queue cons: Locks in code can reduce performance of a single queue system. Additionally, since a processor pulls a process off the front of the queue processes end up running on different processors often. This makes for poor cache affinity unless a mechanism is in place to promote not moving processes around as much.

5.20a **Identify the race condition(s).**

5.20b **Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).**

5.20c **Could we replace the integer variable "int number_of_processes = 0" with the atomic integer "atomic_t number_of_processes = 0" to prevent the race condition(s)? (Assume here this atomic_t has safe, lock-free, atomic loads and stores and an atomic fetch_and_add/fetch_and_subtract like operation.)**

10.10 **Briefly explain why the OS often uses an FCFS disk-scheduling algorithm when the underlying device is an SSD.**

10.14 **Describe one advantage and two disadvantages of using SSDs as a caching tier compared with using only magnetic disks.**

12.16 **Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This filesystem has 12 direct disk blocks, as well as one single, one double, and one triple indirect disk**

blocks entry in its inode. What is the maximum size of a file that can be stored in this file system?

First we will consider how much can be stored by the 12 direct disk blocks. Each disk block is 8 KB and there are 12 so they can store 96 KB in total.

Next we consider the single indirect block entry. A pointer to a disk block requires 4 bytes so we can fit 8 KB / 4 B = 64 disk block pointers in a disk block. This single indirect block entry can map $64 * 8$ KB = 512 KB in total.

The double indirect block entry can store 64 (using the math from the previous step) disk block pointers. Each pointer points to a single indirect block entry which we know can store 512 KB. So the double indirect block entry can store $64 * 512 = 32,768$ KB in total.

Finally the triple indirect block entry has 64 disk block pointers to double indirect block entries. Giving $64 * 32,768 = 2,097,152$ KB in total.

To get the maximum size of a file that can be stored, we sum the above four totals:

$$96\ KB + 512\ KB + 32,768\ KB + 2,097,152\ KB = \boxed{2,130,528\ KB}$$