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: More scalable than single queue systems and multiple queues make exploiting cache affinity easier. 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, only interrupts on the processor executing the wait() or signal() operation needs to be disabled.

Multiple queue cons: Load balancing is required in an 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 more dramatically than a multiple 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).**

On the second line of `allocate_process()` the variable `number_of_processes` is checked to see if the maximum number of processes have been allocated. Multiple threads could not satisfy this if-condition and enter the `else` block. Now each thread in the else block could allocate a new process and it would be possible for the number of processes to surpass `MAX_PROCCESSES`.

The race condition is on the variable `number_of_processes` in general. A race condition can occur when releasing a processes resources and decrementing `number_of_processes`. Another thread could be making the if-condition in `allocate_process()` and there could be available resources that `number_of_processes` doesn't reflect yet.

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).**

The race conditions are caused by `number_of_processes`. Anytime this variable is accessed a lock must be acquired. This means called `acquire()` at the top of `allocate_process()` and `release_process()`. As well as `release()` before they each return.

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.)**

As the code is currently written, this wouldn't help with the first race condition described in part a. Consider `number_of_processes` = 254 and two threads check the if-condition (atomically) and still move into the else block. Now both allocate resources and we have more than 255 processes.

We do have an atomic fetch_and_add to work with now. We could modify the code to increment `number_of_processes` and check the if-condition in one atomic operation. This would resolve our race condition centered around checking a shared counter and incrementing said counter.

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

An SSD is a random access device. Any location on the disk can be read quickly regardless of where the data is located on the device and independent of the previous read. This makes FCFS a nice algorithm of choice as there are no disk seeks with an SSD. Other algorithms such as SSTF, Scan/Look, and C-Scan/C-Look are better for magnetic disks as seek time must be taken into account.

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

Advantages: SSDs are 10 times cheaper than DRAM per bit. They also have 10 times lower-latency compared to a magnetic disk. This makes them a good candidate to operate as a caching tier, of reasonable size, for the magnetic disk. SSD's perform very well at random I/O as they are a random access device.

Disadvantages: Caching data that a process reads from the magnetic disk will cause many writes to the SSD. Cache eviction polices could be tricky as well. When you want to re-write any page (evicting old data) the entire block the page resides in must be erased. If too many writes are performed to an SSD it will wear out its flash blocks. Finally, when the disk is more than 50% utilized the performance begins to fall quickly. They will result in a lot of wasted disk space as you can't use the entire disk as a cache.

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 many block addresses can be stored in a block. Disk blocks are 8 KB and have 4 byte addresses. So they can store $8,192/4 = 2,048$ addresses.

There are 12 disc blocks addressed by the 12 direct disk blocks.

The single indirect disk block can address $2,048$ disk blocks.

The double indirect block can address $2,048$ single indirect blocks that each address $2,048$ disc blocks themselves giving $2,048^2$ addressed disc blocks.

Finally the triple indirect block addresses $2,048$ double indirect blocks or $2,048^3$ direct blocks.

To get the maximum size of a file that can be stored, we sum together the number of addressed disc blocks and multiply by the storage of a disc block:

$$(12 + 2,048 + 2,048^2 + 2,048^3) * 8,192\ bytes$$

$$(12 + 2,048 + 4,194,304 + 8,589,934,592) * 8,192\ bytes$$

$$\boxed{70,403,120,791,552\ bytes}$$