Justin Van Dort Homework 6

ENEE447 4/14/19

When running the kernel on a single core, the code works as expected. But, upon enabling multicore, it is apparent that the multiple cores are messing with each other. When sharing a single stack, the output of the kernel is relatively normal. The noticeable differences include some duplication of output characters, and the do\_blinker not working. Furthermore, the code appears to choose a “random” core to run on. In some cases, all output was from c0, but when restarting, I noticed the output would also come from c1 and c2.

Giving each core their own thread made everything much worse. Granted, the heart of the problem with multicore isn’t sharing a stack, but instead the problem is sharing resources. But, when giving each core its own stack, each core is completely independent except for the devices they are interfacing with. The output of 1\_boot.stacks was extremely convoluted and appeared almost as gibberish.

Of all the possible solutions to this problem, I believe the simplest is to limit each thread to accessing a single device at any given time. In the case of our kernel and 1\_boot.stacks, each thread had its own state but were trying to access the UART at the same time. By only allowing a single thread to access the UART, it would be impossible for the threads to conflict since they can’t access the same device at the same time.

I was thinking it would be possible to simply protect the devices with locks, so that each thread could not use the device in the same instant, but this would not work for our use case. Imagine each thread is running the same code. Each thread is waiting for the user to press enter. If the UART was locked so that only one thread can read the UART at a single instant, the first thread would read the RETURN press, but then the following threads would be waiting for that same RETURN. Since the first thread consumed the RETURN press, all other threads would wait as if the RETURN key was never pressed.

Perhaps different devices can be handled differently when it comes to multicore. For example, using a single UART simultaneously for multiple tasks doesn’t really make sense unless the device on the other side of the UART is set-up for multiplexing. But, for something like a disk drive, it may be possible to use something like locks so that multiple threads can access the disk concurrently.

Imagine there are several independent threads on multiple cores which want to access the disk. Say the first thread acquires the lock on the disk. Then each other thread which wants to access the disk will be blocked by the lock. But, after the first thread reads/writes the disk and completes it’s transaction, it will release the lock. Then, the next thread will be able to acquire the lock and access the disk.

I think the main point here is that some devices are inherently single-use while others can be multiplexed. Reiterating, the UART fundamentally should only be accessed by a single device. Only one thread should be able to read and write the UART. Once the thread terminates, another thread may then use the UART, but while one thread is running and using the UART – even if it is sleeping – another thread should not be able to access the UART. But, in the case of a disk, accesses are transactional. As long as two threads are not using the disk at the same instant, locks will ensure that the disk will function properly.