Sam Detor

Lab 5

Part 1

The first part of this lab required creating a contract between the kernel and the task dictating what kernel methods were available to the task, and how the task should be run. I choose to structure my tasks as static libraries. This seemed to be the simplest way to compile crates that lacked a main method and create a binary that could be relatively easily linked to the kernel one. At first, I tried to access the peripherals directly from the stm32f4 crate in the flash_blue task, but soon realized that this would not work, as that crate has a lot of requirements that couldn't be met by a static library. After receiving some guidance from office hours, I then realized that it would be much smarter to expose certain methods to the task that were defined in the kernel source code that the task could call I then created set_led and sleep methods, which allowed tasks to access peripherals and remove themselves from the scheduling rotation for a certain amount of system ticks. Another aspect of the kernel/task agreement was that the task being loaded, flash_blue, had a method "start", which runs the task and never returns, as well as a non-zero static variable "task_stack_size", which determines the size of the task stack. These were defined using "extern "C"" in the kernel source code files, so the kernel knew the variables/functions were going to be defined in another file and would get these values at link time. This was the same way that the set_led and sleep methods were defined in the task source code.

After figuring out the kernel-task agreement, I then had to link the task and kernel binaries together. I first tried compiling the kernel into .o files and then manually linking the kernel binary and task using the arm-none-eabi-gcc and a custom linker script. However, I was having issues linking the kernel binary to the various crates it imported, specifically cortex-m and the cortex-m-rt crates. After struggling with that for a while, I decided to switch to using a build.rs script, which turned out to be a lot simpler of a process. The build script executed right before the compiling/linking process of "cargo build". Therefore, in my build.rs file, I told the linker to link the kernel binary to a static library named "flash_blue" and where to look for it. After this, the cortex-m-rt build script ran when that crate was built, and created the necessary custom linker script. So the linker linked libflash_blue with the kernel using that linker script.

Then, I still had to debug some of my methods exposed to the tasks, namely "sleep", in order to get the whole system working. In order to better organize my information stored about each task and create scaffolding for my part 2 code, I created a TaskInfo struct array, which stored information like whether the task was loaded or awake. This TaskInfo struct array also originally contained the task stack pointer. Since in my design the context switch method has to be called outside of the "free" blocks where data in mutexes is safely accessed, and the context switch method requires the stack pointers, I decided to make the TaskInfo struct array unsafe. However, whenever the sleep method was called, it would get stuck in the loop at the end of the method that checks whether the task had been reset to awake by the scheduler after the timer expired.

This loop was originally put in because the sleep method could execute in less than 10ms (the allotted time amount given to each task when it is being run). Therefore, before the scheduler had a chance to switch to a new task, the task that was supposedly asleep would be executing lines of code. In this case, it would prematurely toggle the led. I didn't want to trigger the context switch early or it would have messed up the timing for theoretical other tasks that could have been sleeping at that point (looking ahead to part 2).

After some debugging and comparison with my part 4 code, I realized that when the "awake" value in the TaskInfo struct the loop depended on was made a safe variable accessible by a mutex, then the execution no longer got stuck in the sleep method. I am still not sure why that happened, but my best guess is that some sort of race condition occurred. However, that is why I choose to move the task stack pointers into a seperate unsafe static mutable array and make the TaskInfo struct array mutex protected.

In order to generally prevent the use of unsafe rust code, I tried to use as many mutex protected or local static variables instead of global mutable static variables as possible. One notable example of this was the TASK_RUNNING and

WHOOSE_RUNNING variables. Instead of having one global mutable static, I choose to use a mutex protected global variable and a local mutable static to store the same value. This is because information about which task was running needed to be accessed in all parts of the interrupt handler, and many other kernel methods. This way, this value could be safely accessed outside of the mutex data access zones in the interrupt handler and in other kernel methods.

Source Code for Part 1

The only source code I used for Part 1 was my Lab 4 part 4 code. I used the memory.x file, the makefile, the Cargo.toml, and .config files practically unchanged. I used a similar kernel design as before, however, I adapted it so it could load tasks from static libraries, run an idle task when all other tasks were sleeping, support different stack sizes for different tasks, and store more information on each task among other improvements. Please see Source code for Part 4 in my lab 4 report for more details.