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.

<u>Source Code for Part 1</u>

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.

Part 2

The challenge for part 2 was being able to support a varying number of tasks. I choose to limit my user tasks to 4, as the lab spec required 4 tasks to be run at once. After deciding on that, I had to figure out how the kernel was going to distinguish between tasks at link time, because all tasks had to have a "start" method and a "task_stack_size" variable, according to the kernel agreement. With some guidance from office hours, I used the arm-none-eabi-objcopy in my build makefile script to change each task's "start" and "task_stack_size" symbols to "startx" and "taskx_stack_size", where x was a number from 0-3. Since the kernel set the requirement that there were a maximum of 4 tasks, I created 4 start function declarations (start0, start1, start2, and start3) and 4 task_stack_size declarations using "extern "C"" in the kernel source code. These methods would then be linked to their corresponding startx or taskx_stack_size and link time.

Next, I modified the linker-script that the cortex-m-rt outputted on a "normal build" of the kernel to provide a default value for all the startx and taskx_stack_size symbols that could be overwritten at link time. Therefore, even if only two libraries were linked, the linker wouldn't complain about start2 and start3, for example, being undefined symbols. But, for the libraries that are linked, there wouldn't be any issues with multiple definitions of a symbol. This is because the default value would be automatically replaced by the "strong" definitions of the symbols found in the task. This was achieved in the same way the linker script used to provide default methods for the interrupt handlers (using the PROVIDE function). I then changed the rust flags so that my custom linker script described above was used instead of the one cortex-m-rt creates. I also had to edit the build.rs script to link more than one static library. Then, in the main function, to figure out if a task was loaded in each of the four start functions, the value of the taskx_stack_size variable was checked. If it was zero (the default value), no task had been loaded. As mentioned previously, the kernel/task agreement stated that the taskx_stack_size value had to be non-zero. Otherwise, a task was initialized using that stack size and function address in taskx_stack_size and startx, respectively.

After that, my logic for my initialize task and TIM3 interrupt handler had to be changed to include support for multiple tasks. One issue I ran into when re-working the scheduler logic, the question of how to decide what task to run after the idle task had been run (i.e. after all tasks were asleep). To make this process simple, I introduced a PAUSED_TASK local mutable static variable. If the task identified as "next_task" was asleep, it would be stored in this variable. If no other task were awake, the idle task ran. The next context switch (from the idle task), the task stored in PAUSED_TASK would be the first to be checked if it was awake. That way, the round robin pattern could be preserved as much as possible even with tasks being put to sleep/woken up.

Lastly, for my new initialize_task function, I used the stack minimum of the task that had been loaded before the previous task minus 4 bytes as the new stack start of the next task. This allowed me to support variable stack sizes simply. However, this meant that all stack sizes were required to be 4 byte aligned, so that they could be used in the calculation of the next stack pointer without creating unaligned pointers. I also enforced a

4 KB maximum stack size to make sure that all stacks could be supported without using all available RAM.

My methods for minimizing the use of unsafe rust stayed constant from Part 1 to Part 2, so please see the Part 1 section of this lab report for more details.

Source code for Part 2

I used a modified version of the cortex-m-rt linker script in my code. The main change I made was providing default values for the startx and taskx_stack_size symbols described above using the PROVIDE method.

I also used my Part 1 code as source code. I changed my Part 1 code to support multiple tasks, which required changing the main.rs, build.rs, makefile, and .config file from Part 1. These improvements are detailed above. Please see my Part 1 source code section for more details.

Part 3

To start, I first slightly changed my kernel/task agreement from Part 2. Since the memory from unloaded tasks is reclaimed by the kernel in Part 3, it was simpler to have every task have a set stack size (I chose 3KB). This way, blocks of memory returned from unloaded tasks could be easily reused to house new tasks.

I chose to manage the memory the kernel was giving or reclaiming from the tasks using a "free_blocks" array. It was initialized in the main method to have four, the maximum number of user tasks my kernel allows, memory addresses that were 3KB apart. They could then be used for task stack pointers. Therefore, instead of calculating a task's initial stack pointer using the previously initialized task's stack pointer like my part 2 kernel, a starting stack pointer was chosen from the free_blocks array. When an address was chosen, it was removed from the free_blocks array and replaced with a zero. Then, when that task was unloaded, the starting stack pointer could be returned to the free_blocks array for later use.

For the loading/unloading of tasks, I chose to use a circular strategy. Tasks were loaded into lower indexed spots in the task_info_array first. Once the last available task was loaded, in order to load more tasks, one would have to be removed. Since tasks are removed from the oldest task, an empty spot would be at the beginning of the task array. So the next time a task is loaded after the last loaded task, it will be loaded in the first available spot in the task_info_array, even if there are other spots available. This allows for "age-based" unloading to go smoothly. Since tasks are always loaded from lowest to highest index, then unloading tasks from lowest to highest index makes sure that the oldest task is always unloaded first.

The loading and unloading methods—the EXTI0 and EXTI9_5 interrupt handlers, respectively—each had local static variables to keep track of task_ids to be loaded or unloaded next. This provided a safe and easy way to implement the circular loading and unloading. For the loading, I choose to create an array of the linked task function pointers so that the local static variable (NEXT_LOAD) could also be used to find the function pointer of the next task to be loaded as well.

The actual process of loading a task was the same as initializing a task in my part 2 kernel. A starting stack pointer was chosen, the task stack was initialized, the task_pointers and task_info arrays were updated, and the tasks_loaded variable was incremented. Unloading consisted of decrementing the tasks_loaded variable, returning the starting stack pointer for the task—saved in the TaskInfo struct—to the free_blocks array, and marking the task as unloaded. As the spec mentions, the user button was used for loading and PC7 was used for unloading.

The mechanics of the TIM3 handler, which was responsible for the context switching, didn't change much from my part 2 kernel. Since dealing with sleeping tasks was similar to dealing with unloaded ones, I just added to the logic of finding an awake task to ensure that the task being run was loaded. If the task set to run next wasn't loaded, then the rest of the tasks were searched through to find a loaded task to run instead.

The biggest issue I ran into during this assignment was probably trying to find the smoothest design for loading and unloading. I tried a few different things, including trying to track the first and last loaded task and load and unload such that all the loaded tasks were in a continuous region of the task_info array. I hoped to do that so that the scheduler didn't need to worry about tasks being unloaded because it only operated within the bounds of the first and last loaded task. I decided to change strategies because it was difficult to load new tasks and unload the oldest task such that all the loaded tasks stayed in a continuous region of the task_info_array. In the end, the simplicity of my current design seemed to outweigh the sligh efficiency benefit I could have gotten from the other design.

In order to reduce my use of unsafe rust during this part of the lab, all new global variables added during this part were put into mutexes. Additionally, since I gave every task a set stack size instead of a variable one, I no longer had to use externally defined static variables, making the code I used to initialize each task safer.

Source code for part 3

I used my Part 2 code as source code. I changed my Part 2 code to support dynamic loading, which required changing main.rs and the makefile from Part 2. I also removed the static task_stack_size variable from the source code of each task and removed the renaming of the task_stack_size symbol in the "prep_tasks" section of the makefile. The improvements to main.rs have been detailed above. Please see my Part 2 source code section for more details.