Project 3:
Multi-Threaded Kernel

By: Jesse Millwood and Emily Theis
Class: EGR 424
Instructor: Professor Parikh
Date: July 30, 2015

1. Introduction

This project was to develop a preemptive multi-threaded kernel on the LM3S6965 microcontroller. This was done using four threads controlled by the kernel, that run one at a time but could be interrupted at any time to give control to another thread. In order for the thread to retain all its information when returned to, the kernel used the stack to save and restore the state of the thread. Each thread has its own stack so that they could each be interrupted and restored with no lost information.

2. Modified or Created Functions

a. Main

This function was set up so that the peripherals, such as the UART, OLED, LED, and Systick Timer and Interrupt would be set up and initialized first. Next, the function to create the threads was called (initialize_threads), and the locking mechanism was initialized. Finally, the main function ends with a while loop that will continually run the kernel.

periphs_init

This function is called by main to initialize the peripherals: OLED, LED0, and Systick Timer. It then calls the master interrupt enable.

b. Threads

initialize_threads

This function was called during the setup in main. The four threads were set up in a structure that stored its buffer array, active state, and pointer to the top of the stack. This function then allocated memory on the stack for the thread table. Finally, it calls the createThread function that called the assembly file, “create.s”.

create.s

This function was the most important part of the project to make the kernel work properly. The parameter passed to it was the thread stack pointer. Next, the current register values for r4 through r11 were saved in the array, since they are not automatically stored on the stack. It then branches to the function ThreadStarter().

c. Scheduler

scheduler_handler

The scheduler used was similar to the template given. However, it was set as the exception handler for both the SysTick Interrupt and SVC Call Interrupt. The SysTick Timer provided pre-emption, where the scheduler would be called every 1 ms. The SVC

Call allowed for the scheduler to be called manually, but using yield(). In the scheduler, it first saved the current status of the active thread by called the save registers function. Next, it searched for the next active thread in the thread table. It then restored this thread's most recent state, so that it could continue operation where it had left off.

d. Yield

This function contained the assembly code to call an SVC Call. This would then trigger a return to the scheduler, where it could switch threads. It was called to begin the kernel in main, and in the threads.

e. Save Registers

save_registers

This function was used to replace setjump(). It saves the current status of the thread by first saving the process stack pointer into r1. It then saves the values that are not automatically saved, which includes r4 through r12, and r1 which hold the process stack pointer. The function then returns 0.

f. Restore Registers

restore_registers

This function was used to replace longjmp(). It restores the state of the given thread from its specific memory location on stack. It does so by first loading the process stack pointer from r1. The values saved from the registers from save_reg_state were then loaded back into r4-r12. Finally, the function executes a fake exception return. This was necessary because the entire scheduler, save, and load process was located inside of an exception handler. While inside the handler, the program is running in privileged mode using the main stack pointer. While this was necessary for the scheduler, it needed to return to using the process stack pointer in order for the threads to operate properly. Without this, the interrupts in the threads would be ignored until the threads were complete. Therefore, the fake return was used so that when "bx lr" was called, it would return to the value 0xFFFFFFFD, which was found in a table of potential values for this purpose.

g. Lock

The lock was implemented so that two threads could have access to the same peripherals but still have exclusive access. This was used for the UART peripheral. It was an extra credit feature of the project.

lock_init

This function was used to initialize the locks in the structure. There are three parts of the structure: the state (to determine if the lock is acquired or released), the count (to count how many locks have been acquired), and the owner (to determine which thread owns the lock).

lock_acquire

This function was used when the thread needed to acquire a lock. First, it incremented the lock count in case more than one lock was used on the same thread. Next, it uses assembly to gain exclusive access to the peripheral. If successful, it changes the state to locked, and changes the lock owner to the current thread. If the peripheral is already locked, it branches to return 0 (unsuccessful).

lock_release

This function was used when the thread needed to give up control of the peripheral. It first decrements the lock count, then changes the state and owner to indicate that the thread is unlocked.

3. Context Switch

The context switch is the time it takes for the kernel to go from one thread and switch to another. Basically, it is the amount of time it spends in the scheduler function. This includes saving the state of the current thread, finding the next active thread, and restoring the state of the next active thread. This determines how quickly the kernel can operate.

In order to measure the context switch time, an LED pin on the board was turned on upon entry into the scheduler, and turned off when exiting. This pulse was then captured and measured using an oscilloscope. As shown below, the context switch time was determined to be 8.2 μs. This is much faster than the 1ms Systick Timer.

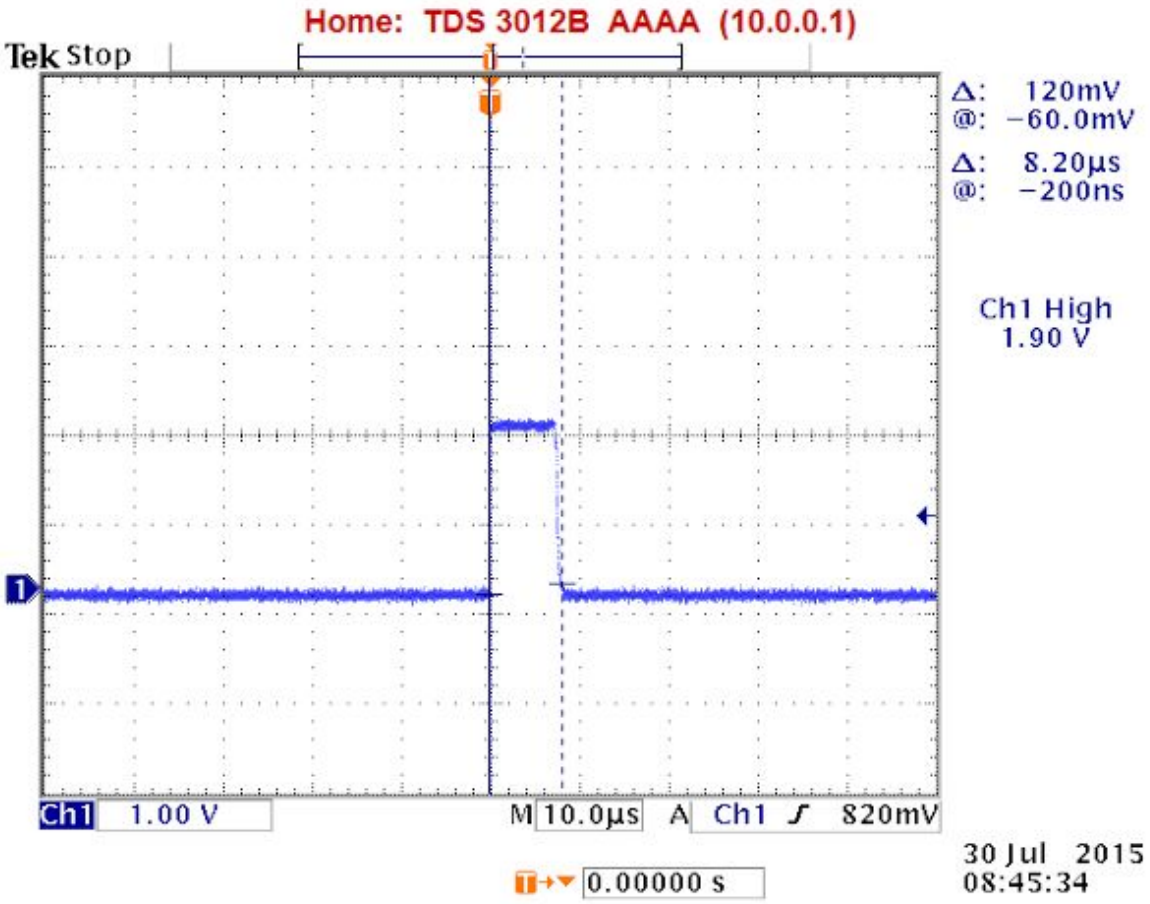

Figure 1: Context Switch Time Measurement