Project #3
Pre-emptive Multi-threaded Kernel

EGR 424

7/31/18

Mike Ford, Brandon Baars, Jason Hunter

<u>Introduction:</u>

This project had us develop a pre-emptive multi-threaded kernel on the LM3S6965 microcontroller. The end result of your efforts will be a kernel that provides an abstracted view of the hardware, such that the application developer can be concerned only with writing independent threads/processes to perform system functions. Thread creation, thread switching, scheduling, etc. will all be handled by your kernel.

<u>Proj3.c</u>

void SystickInit()

This function initializes the SysTick to generate a 1ms interrupt. This interrupt is what will be used as a pre-emptive kernel. After the interrupt fires, the scheduler will take over and revoke access to the thread, thus allowing another thread to do work.

void InitializeLED()

This function initializes the LED on Port F as output.

void InitializePeripherals()

This function initializes the System Clock to run directly from the crystal, initializes the OLED display, sets the UART pins to be used on GPIO Port A by Pins 0 and 1 as well as configuring UART for 115, 200 baud and 8-N-1 operation.

void threadStarter()

This function is the starting point for all the threads. It runs all the user level threads, by context switching between them from the createThread() in the LR of the jump buffer. It calls the current thread using the thread table. When the thread is returned, it returns back and updates the active thread. It then returns control to the scheduler.

void createThread(int *state, char *stack) A.K.A *Create.S*

This function is called in the main function after memory is allocated for the thread. It will save the current values of register r4-r11, r13, and LR in the saved register array.

void scheduler_handler()

This function saves the state of the running thread by saving the registers. It then determines which thread to run next based on the current thread

void yield()

This function will execute the SVC instruction. This would trigger the scheduler_handler function, meaning that it returns back to the scheduler.

int save_registers(unsigned* buffer)

This function saves the PSP into R1, as well as stores R1, R4-R12.

void restore_registers(unsigned* buffer)

This function restores the register by loaded the value of R0 into R1, Updates the Special Register to use the value it stored from save register. And then reloads the R4-R12. The value 0xfffffffd is then loaded into LR to fake a return.

void lock_release(lock_t* lock)

This function checks to see if the lock owner is the current thread and if the lock count is 0. If it is, the owner is set to nobody and the lock state is set to unlocked.

unsigned lock_acquire(lock_t* lock)

This function checks to see if the lock is set to locked. If it is, the function returns an unsuccessful. If it is unlocked, the count is incremented by one, the state is set to lock, and the lock owner is set to the current thread.

void lock_init(lock_t* lock)

This function initializes our UART lock at the beginning of the program. This sets the lock owner to nobody and the lock to unlocked with a count of 0.

<u>Threads.c</u>

void UART_Thread1() & void UART_Thread2()

This function tries to acquire the uart lock. If it is successful, it starts printing via UART. If it is not successful it yields control back to the scheduler. Otherwise, a systick interrupt will give control back to the scheduler.

void OLED_Thread()

This function has arbitrary for loop to act as a delay so that the scheduler revokes access and each write to the OLED display. It should write “OLED STRING THREAD”, once that completed it yields

void LED_Thread()

The LED Thread has a for loop which acts as a delay to visually indicate when the LED is on vs. when it is off. After the delay it toggles the LED pin and yields control back to the scheduler.

void idle_thread()

The idle thread tries to enter sleep mode by calling an assembly function ‘WFI’ which is *Wait for Interrupt*. This puts the processor in sleep mode and waits for the systick interrupt. Upon this interrupt, the scheduler should take over and give access to the next thread.

<u>Context Switch:</u>

The context switch is the switching from one process or thread to another. In our program, we had a interrupt every 1ms that would switch from one thread to another. To ensure that the actual time it takes to switch between threads is at around 1ms, we measured the time using an oscilloscope. The GPIO Port E Pin 3 was toggled HIGH vs. LOW to measure the context switch. The actual time to switch between thread was found to be 7.2 microseconds. This can be seen below with the vertical cursors at the start and end of the switch.