HW 4 – RTOS Analysis

沈昱宏, 110705013

***Abstract*—In this homework assignment, I analyze the performance of FreeRTOS for multithreading. The software code I am executing is a simple code with two tasks executing in different threads. I traced the OS kernel code and analyzed how thread management and synchronization are done. I measured the overhead of context-switching and synchronization of the application.**

***Keywords*—Aquila, FreeRTOS, Multithreading**

I. Context Swtiching Behavior

A. Task Creation

In rtos_run.c, the main function calls xTaskCreate to create two new tasks. The function is defined in tasks.c line 728. It will create a pxNewTCB (task control block) with a stack, initialize it with some information (priority, pxTaskCode, pcName…), and add it to the ready list.

B. Start Scheduler

After the creation of the two tasks, vTaskStartScheduler is called. In this function, an IDLE task is created. After that, timer is intialized and xPortStartScheduler (in port.c) is called to start the scheduler, enable interrupts, start the first task, and transfer control to FreeRTOS by calling xPortStartFirstTask in portASM.s.

C. Scheduling

The logic of task scheduling can be found in the function xTaskIncrementTick(). This function is called when system tick interrupt occurs and handles task sceduling decisions (returns whether xSwitchRequired). The context switch will happen when there is a task having a priority greater than or equal to the current executing task, which means that when there is a task with highest priority, the task will never be preempted, and if there are n tasks with the same priority, each of the tasks can use 1/n of the time. In the source code, the two tasks have the same priority, so the two tasks will take turns to execute (assume there is no semaphore).

D. Context Switching

When interrupt is triggered, vTaskSwitchContext in port.c will be called to select a task with taskSELECT_HIGHEST_PRIOIRTY_TASK(). The commands of loading and restoring registers can be seen in portASM.S.

II. Synchronization

A. Critical Section

The function taskENTER_CRITICAL is used to enter critical sections. It is implemented in the function vTaskEnterCritical in tasks.c. It calls the function portDISABLE_INTERRUPTS() in portmacro.h to disable the iterrupts. portDISABLE_INTERRUPTS calls "csrc mstatus, 8" to set the 4th bit of the mstatus register in the csr in order to disable machine-level interrupts to ensure critical sections of code are executed atomically. As for taskEXIT_CRITICAL, the iterrupt is enabled with "csrs mstatus, 8" to set the register in csr.

B. Mutex Lock

The implementation of xSemaphoreTake is in queue.c. The function is named xQueueSemaphoreTake. In the function, several validations are done initially. When a semaphore is taken, the count will be decreased (this part is done in critical section). en the signal rst is 0. If the mutex is locked and the should be released by another task, the function will call queueYILD_IF_USING_PREEMPTION() to allow other task to execute.

C. Mutex Unlock

The mutex unlock function xSemaphoreGive is implemented in xQueueGenericSend in queue.c. In the function, it checks if the queue has space for the new item. If there is space, it calls prvCopyDataToQueue to insert the item, adjust states and unblock any tasks waiting for data (done in critical sections).

III. Measure Context Switch Overhead

A. Profiling

I add a counter compute the total number of time context switch occurs and the total cycle is used for context switching. When the program counter equals to the first address of the free_rtos_risc_v_trap_handler, context switch starts. I observed the program execution and found out that the last command be executed in context switch is the last few lines in the function processed_source (lots of nop). I used these two points as the starting point and the ending point of the context switch.

B. Result

The following table is the data I stored for profiling and some statistics. The context switch ratio is computed by dividing "context switch cycle count" with "total cycle count", and it shows that the context switch only accounts for 0.17% of the cycle, which is only a small number of cycles. In addition, the context switch overhead is computed by dividing "context switch cycle count" with "context switch count". This shows that the context switch will use 695 cycles on average to finish.

TABLE I. RESULT UNDER DEFAULT SETTING

Column	*Value*
Total Cycle Count	201893593
Context Switch Cycle Count	342402
Context Switch Ratio	0.17 %
Context Switch Count	493
Context Swich Overhead (Cycles)	695

IV. SYCHRONIZATION OVERHEAD

A. Mutex Lock & Unlock

I use the address in the two handlers directly as the starting point and ending point of the locking and unlocking the mutex. I count the total number of the mutex locked and the total number of cycles being executed during the mutex locked.

B. Result

The following table shows the result of using a mutex (I did not count the sections in taskENTER_CRITICAL and taskEXIT_CRITICAL). The synchronization overhead here is computed by dividing the “total cycle used for mutex” (including lock and unlock) by the “mutex use count”. Note that the “Mutex Use Count here is not exactly 10000 because the variable “done” is only updated in atomic sections (when mutex is used).

TABLE II. RESULT UNDER DEFAULT SETTING

Column	*Value*
Total Cycle Count	201893593
Total Cycle used for Mutex Lock	1531008
Total Cycle used for Mutex Unlock	2066482
Total Cycle used for Mutex	3597490
Mutex Cycle Ratio	1.78 %
Mutex Use Count	10002
Synchronization Overhead (Cycles)	360

V. ADJUSTING TICK RATE

I adjusted the value of configTICK_RATE_HZ in FreeRTOSConfig.h to see what will happen. The result shows that it will affect the rate of context switch performed. When I double the tick rate, the context switching time will reduce because the tick will increase in the speed two times than the orignal speed. Therefore, when I double the tick rate, the context switch amount also doubles.

VI. CONCLUSION

In this analysis, I examined how FreeRTOS manages tasks and synchronization, focusing on context switching and sycronization. The observations show that FreeRTOS handles these operations efficiently, with small time spent on switching tasks or managing shared resources. Overall, FreeRTOS proves to be an effective solution for real-time task management.