HW4_FreeRTOS analysis

黃柏竣, 110550123

***Abstract*—此作業對 FreeRTOS 這個 Real-time OS(operating system)做運行及測試分析。先 trace code 了解背後各個函式如何合作運行，並在有興趣的函式對應的硬體或 PC(program counter)做判斷、紀錄，以此來計算並做分析。而分析的內容主要是 threads 之間 context switch 或是使用 mutex 做同步的 overhead。**

I. INTRODUCTION

Real-time OS 是一種設計用於即時應用的作業系統，希望在特定時間範圍內完成任務，以滿足實時性的需求。

而 FreeRTOS 是一個開源的 real-time OS，此作業主要在探討 FreeRTOS 在 Aquila(單核處理器)上運行的狀況，關注在 real-time OS 執行 multi-thread 程式，thread 之間在互動、執行時，了解 interrupt、synchronization 的處理及 context switch 背後 source code 如何實作；並透過 Vivado ILA 增加 counter，來紀錄這幾個操作的 overhead，再做後續的討論。

II. FREERTOS

A. Trigger timer interrupt

Aquila 內的 clint module，除了有資料要寫入，否則 mtime 會在每個 clock cycle 都加 1，可以算是當前執行的時間；而 module 內的 mtimecmp 則是會在 xTaskIncrementTick 時，將原先 mtimecmp 的值增加一個 time quantum 的時間，timer interrupt 就是發生在 mtime 大於等於 mtimecmp 的時候。

B. Time quantum setting

FreeRTOS 對每個程式輪到時，要執行的 time quantum 設定在 FreeRTOSConfig.h，configTICK_RATE_HZ 會決定實際是幾秒鐘，就是 1 除以這個的設定值；而在對 mtimecmp 做增加時，還會看另一個設定的參數 configCPU_CLOCK_HZ，會將 TICK_RATE 乘上 CPU_CLOCK 加到原本的值後再更新進去。

C. Timer interrupt code trace

在 mtime 等於 mtimecmp 時，FreeRTOS 的 PC 會跳到定義在 portASM.S 裡的 freertos_risc_v_trap_handler，藉由此函式來做 interrupt 的處理，看 code 看起來是將各 register 以及 TCB(task control block) 的資訊都存進 stack(memory)中，然後因為 timer interrupt 是外部的計數數到觸發 interrupt，不是執行程式時自己出錯，所以是 asynchornous，接下來就會進到 handle_asynchronous 函式。

而此 handle_asynchronous 先做的事情是把 mtimecmp 讀進來之後，做 increment tick，再增加一個 time quantum 的時間做更新，之後會進 vTaskContextSwitch 函式，會將 ready task list 中 priority 最高的 task 資訊設定為 pxCurrentTCB，最後是 processed_source，把記憶體及 pxCurrentTCB 的資訊都讀到 register 內，做最後切到另一個 task 的動作。

D. Mutex take/give code trace

在 FreeRTOS 中，是用 queue 來實作 mutex 跟 semaphore，而此次作業中關注在 mutex 部分，FreeRTOS 在 take mutex 時是呼叫 xQueueSemaphoreTake 這個函式，give mutex 則是呼叫 xQueueGenericSend。

xQueueSemaphoreTake 會先做一些 assert 檢查，然後看有沒有人已經拿走 mutex(對 queue 做讀取)，若有，就會在一個無窮迴圈裡等待，而如果還沒被拿走，就確認一下 priority，如果拿走 mutex，就要更新持有 mutex 者，讓程式開始跑。

xQueueGenericSend 也同樣做一些 assert 檢查，會送訊號到 queue 裡，如果 queue 不是滿的，就會成功送入 queue 中，之後會看有沒有 task 在等 mutex，如果有，就切換成 priority 符合的。

E. Enter/Exit critical section code trace

從 objdump 檔中看到程式執行到 enter/exit critical section 時，會分別跳入 vTaskEnterCritical、vTaskExitCritical 這兩個函式。

而這兩個函式會對 TCB 中的 critical nesting 值做增減，enter 會做加，exit 會做減；vTaskEnterCritical 會將 mstatus 的 MIE bit 設為 0，代表不接受 interrupt 請求；vTaskExitCritical 是在 critical nesting 減少為 0 時，會將此 bit 設回 1。這樣的機制應是透過不讓其他程式 interrupt，來達到 critical section 內部不會同時有其他程式執行，因為只要有一程式 enter critical section，就不會被 interrupt，直到 exit。

III. IMPLEMENTATION

A. Check task1, task2

從 rtos_run.c 及 rtos_run.objdump 下手，兩個 task 都是做 vTaskDelete 後，代表該 task 結束，所以我個別有做一個 PC 的判斷，看它們有沒有執行到 vTaskDelete 結束，若兩個 tasks 都結束，接下來的 idle task 就不是我們所在意的，就不會再去對各值(像 context switch cycles 等等)做更新。

因為觀察 objdump，看到 task1 主要會跳出去的函式是做 mutex 的 take/give 及 critical section 的 enter/exit，所以就判斷是否執行 task1 的方法就是看 PC 是不是在 task1 的開

頭到 vTaskDelete 之間；而 task2 也是用同樣的方式，不過因為 task2 主要做的動作是跳入 rand 這個函式，且觀察 objdump 的結果發現沒有其他函式會呼叫到 rand，所以也將執行 rand 函式的時段，也判斷為正在執行 task2。

B. *Check context switch*

用 Vivado ILA 及 rtos_run 的 objdump 檔案做觀察，發現到當 mtime 和 mtimecmp 相同時，會將 context switch 的次數加 1，而 PC 也會跳到 freertos_risc_v_trap_handler 的開頭，因此用一個 register 存是否在做 context switch，當 PC 到上述函式的開頭，此 register 內的值就變為 1，之後一直到再次執行 task1 或 task2，才會變為 0，而變為 1 的這段時間會在每個 clock cycle 都對記錄 context switch cycles 數的 register 加 1，用這樣的方法判斷並取得數據。

C. *Check take/give mutex*

觀察 rtos_run.objdump，如同前面所說，是用 xQueueSemaphoreTake 及 xQueueGenericSend 這兩個函式，來做 mutex 的 take/give。

但因為有看到這兩個函式蠻常跳到其他函式執行，且那些函式也不只被它們呼叫，因此我認為不能用一段 PC 的區間來判斷是否正在做 mutex 的 take/give，不然會將其他也跳到同函式的執行 cycles 數也算進去，所以這邊是用兩個 register 看函式開頭的 PC 來判斷是否開始，並分別在 PC 到這兩個函式做 return 的地方時，代表 take/give mutex 的結束。

當函式開始時就會在紀錄個別次數的 register 加 1，而一直到函式結束，會一直在每個 clock cycle 對對應 register 做增加，以算出 take/give mutex 的次數及 clock cycle 數。

D. *Check enter/exit critical section*

是否 enter/exit critical section 是看 vTaskEnterCritical、vTaskExitCritical 這兩個函式，而根據 objdump 提供的資訊，這兩個相較其他算是比較短的函式，而且也沒有跳到其他函式做執行，因此這邊實作的判斷方式就是直接看是否在該函式的 PC 區段內，若是，就代表正在執行，會在剛進入函式執行時，對紀錄個別次數的 register 加 1，並只要在那個區間內，就會每個 clock cycle 增加 register 內的值。

IV. RESULTS

A. *Overhead of context switch caused by timer interrupt*

在 FreeRTOSConfig.h 修改 tick rate 後，進行測試，執行 FreeRTOS，從 Vivado ILA 得到各結果整理出的資訊如下表 TABLE I.。.

TABLE I. CONTEXT SWITCH TIMES, CYCLES AND AVERAGE WITH DIFFERENT TICK RATE RUNNING FREERTOS

TICK_RATE_HZ	*Context Switch times*	*Context switch cycles*	*Average context switch cycles*
10	48	22226	463.04
20	96	40096	417.67
50	242	95592	395.01
100	480	186737	389.04
200	963	366033	380.10
500	2423	918278	378.98
1000	4866	1846865	379.54

B. *Take/Give mutex overhead*

這部分皆是在 tick rate 設定為 100 的情況下測試，執行 FreeRTOS，從 Vivado ILA 得到結果整理出的資訊如表 TABLE II.。.

TABLE II. AVERAGE TIMES, CYCLES AND AVERAGE OF TAKE/GIVE MUTEX RUNNING FREERTOS

Take mutex times	*Take mutex cycles*	*Average take mutex cycles*	*Give mutex times*	*Give mutex cycles*	*Average give mutex cycles*
19550	1623655	83.05	12849	1971393	153.43

C. *Enter/Exit critical section*

此部分同樣是在 tick rate 設定為 100 的情況下測試，執行 FreeRTOS，從 Vivado ILA 得到結果整理出的資訊如表 TABLE III.。.

TABLE III. AVERAGE TIMES, CYCLES AND AVERAGE OF ENTER/EXIT CRITICAL SECTION RUNNING FREERTOS

Enter critical section times	*Enter critical section cycles*	*Average enter critical section cycles*	*Exit critical section times*	*Exit critical section cycles*	*Average exit critical section cycles*
20153	341087	16.92	40204	642414	15.98

V. DISCUSSION

A. *Context switch overhead of different tick rate*

調整 FreeRTOS 的 tick rate 並做測試後，發現把 tick rate 設成比 200 大後，也就是 time quantum 設為小於等於 5 ms 後，平均花在 context switch 的 cycle 數就差不多在 380 左右，應該是因為 time quantum 的變動幅度也跟著變小，僅有個位數毫秒的差別，而 tick rate 小時，都是幾十毫秒在變化，所以在 tick rate 大的部分會比較不容易看出差距。

而 time quantum 設比較大時，average overhead 較高，代表表現是比較差的，原本想可能是因為在更新 mtimecmp 時因為要加的數字較大，使得需要的 cycle 數變多，但後來想想也不太可能，畢竟也沒大到超過 32bit 的範圍，可是感覺我有注意到的其他部分在不同 time quantum 都是差不多的流程，應該不會有多大變化，也許需要再更仔細 trace code，才能看出潛在原因。

此外，雖然 tick rate 在越設越大時，有明顯看見 context switch cycles 往下掉的趨勢，但在 500 變到 1000 時，就開始有上升一些，我想這可能是因為 context switch 開始變得越來越頻繁，程式被分到的執行時間不夠長，就會有更多比例花在 context switch 上。

B. *The way I chose for implementation*

雖然在此作業中我大多使用 PC 去判斷，當前是否正在做某個函式裡的動作，並對函式對應的 register 做處理，但其實這是我想不到其他辦法才這麼實作，我認為用 PC 去判斷，主要的缺點就是，如果把 code 更動的話，objdump 也會變，也就代表 PC 跟著變，判定的範圍或是進入點等等都要再做處置，而不是像 timer interrupt 那樣比較 mtime 跟 mtimecmp 來判斷，不論軟體怎麼變都還是那樣，不用多做變動。但我不確定這部分是否有辦法達成，因為像是 enter critical section 有將 mstatus 的 MIE bit 做變化，但我也有看到其他的函式也做同樣動作，這會使就算在硬體部分看 register 內的值也無法確認是哪個函式在執行。

C. *Take/Give mutex overhead*

從測試結果看來，take/give mutex 的 overhead 比其他的還多，原因應該是簡報有提到的 priority 處理，因為也有在 objdump 中看到 priority 相關的函式，不過沒有把 priority inversion disable 後做測試，因為改 code 重編譯後，PC 會跟著變動，感覺實作起來不太好處理，除了軟體部分，硬體部分也要改。

除此之外， 有關 average 的執行 cycle 數，give mutex 高於 take mutex 的多很多，兩者都有無窮迴圈，而我所想到可能的原因，就是剛好在 give 時，有被 block 住，在無窮迴圈中繞更久，而 take 相較之下較沒有問題，才使得這樣的情況發生，不過為何會剛好在 give 發生較多，以及如何做改善和優化，可以再做探討。

D. *Enter/Exit critical section overhead*

Enter/Exit critical section 的 average clock cycle 數是差不多的，認為是蠻合理的，畢竟它們兩個函式都比較簡短，也沒有無窮迴圈之類的東西會卡在那邊，所以我認為是蠻合理的，讓我比較不解的是兩者的呼叫次數竟然是不同的，而且還差距蠻大，因為就我原本的理解，一個 enter 應該要搭配一個 exit，雖然看 objdump 內函式的出現頻繁程度比例確實和測試出來的差不多，個人目前猜想是覺得說不定跟 critical nesting 有關係，但具體是為何，還需要進一步研究。

E. *Overhead percentage of rtos_run*

因為 take/give mutex 的 overhead 相較於另外兩者還多蠻多的，所以就想要再確認一下，這所謂較多的 overhead 對整個 rtos_run 有多大的影響。

因此，我有再去測試並確認一下，想知道 rtos_run 整個跑完 task1 及 task2 的總 cycle 數，就用 mtime 這個從一開始就在數的值當作代表，雖然 clint module 在 write enable 時會停止增加，表示會少計數到一些，但根據 tick rate 設定為 100 時，Vivado ILA 的結果，mtime 仍有到五億多的值，我認為在這樣的數量級下，那部分少算的影響是小的。而在此情況下去看 take/give mutex 的 overhead，其實影響是不到 1%的，那 context switch 及 enter/exit critical section 的比例也就更小了，雖然這些部分用這種方式判斷看起來沒有太大影響，但也同樣代表有其他 loading 很重的部分存在，之後可能可以嘗試 homework 1 的 gprof 及實作的 profiler 等工具做分析。

APPENDIXES

Fig. 1. Line graph of average context switch cycles of different tick rate