HW4 Report

謝翔丞, 109550025

***Abstract*—以 FreeRTOS 搭配原先已掛載 cache 版本之 Aquila，同時以 ILA 為輔，嘗試對於 RTOS 執行過程之 context-switch 以及 synchronization 進行行為分析與計算，發現 Context-Switch 次數隨 Time Slice 上升而下降，同時超過 99.9%的比例在處理 asynchronous 的 case，並且利用 atomic instruction 嘗試撰寫 mutex 並進行測試功能正常。. (*Abstract*)**

Keywords— Aquila, FreeRTOS, mutex, atomic instruction (key words)

I. Introduction

本次作業是建立在 HW3 的基礎上，也就是在掛載 cache 之後的 aquila 上執行 FreeRTOS，透過分析 RTOS 的 thread management 並且藉由 ILA 輔助計算出 context-switch 以及 synchronization 的 overhead， 同時以 atomic instruction 來實作出 mutex。.

II. Analysis Of FreeRTOS

A. *Task*

本次旨在分析 FreeRTOS 對於 thread 的管理.，因此從 Task.c 著手我認為是再適合不過的選擇。在此處，task 可以被視為一個 thread，而這個 thread 的排程則是藉由 TCB 來負責掌控，我們可以先從幾個變數與函式看起。

1. xStateListItem, 這個變數記錄了 task 的三個相關 state 資訊，包含 ready, blocked 或是 Suspended。
2. uxPriority,用來記錄每個 task 的 priority，值得注意的是 FreeRTOS 裏頭將 0 視為最低的 priority 逐而遞增，恰好與我們熟知的 Nice Value 判斷順序顛倒。
3. pxReadyTaskList，這個 array 給定 ConfigMAX_PRIORITIES 的長度，目的是儲存特定 priority 並且已經 ready 的 task。
4. xDelayList1&2,其一如其名是為了計算 Delay task，而另一個的用意則是為了應付前者 overflow 時的情況而存在。
5. xTickCount,這個變數在此次分析中頗為重要，是一個初始為 0，用來計算程式執行過程發生幾次 tick interrupt 的角色。

B. *Main*

接下來會從 Main function 逐步 trace 幾個重要的流程，在 Main 中依序進到兩個主要函式，而第二項也是本篇著墨的重點處:

1. xTaskCreate
   - 甲、 這個函式判斷一旦 putSTACK_GROWTH >= 0 則要先使用 pvPortmalloc 去 allocate TCB 再 allocate stack，反之則是與前述相反的操作。
   - 乙、 接著再依序進入 prvInitialiseNewTask 以及 prvAddTaskList，於此同時 pxCurrentTCB 也會根據自身使否為 Null 來選擇要更新成 pxTCB 或是選擇 “優先度較高或是較為新者”，之所以這邊會補充"較為新者”是因為，在 FreeRTOS 的分級裡，相同優先度者以較新加入者為優先。
2. vTaskSchedular
   - 甲、 這個函式在一開始會建立一個優先度為 0 的 IdleTask，並且進行一些基本的設置，例如將 xSchedularReady 設為 True、xTickCount 設為 0，設置 xSchedularRunning,用來表示 Schedular 已經開始運作。
   - 乙、 呼叫 xStartSchedular 函式，內部會再尋訪兩個 xPortStartFirstTask 以及 vPortSetupTimeInterrupt，後者如其名負

責設置 riscV csr 的 mtime 與 mtimecmp，這裡我們可以一併看到 aquila 中 CLINT.v 122 行的 " assign tmr_irq_o = (mtime >= mtimecmp) & (| mtimecmp); " 一旦 mtime>=mtimecmp 時會觸發 interrupt。

丙、 我們再進一步向下深入，當發生 interrupt 要執行 context-switch 時，xPortFirstTask 會先跳到 freertos_risc_v_trap_handler 進行多次的 sw，接著更新 currentTCB，隨後將 csr mvtvecv 寫入；此處可一併參照 Aquila 的 programcounter.v 和 csr_file.v，programcounter 拿到的 PC_handler_i 就是 csr_file.v 中透過讀取 csr mtvec 而提供的 Output，如此完成兩邊的 r / w。

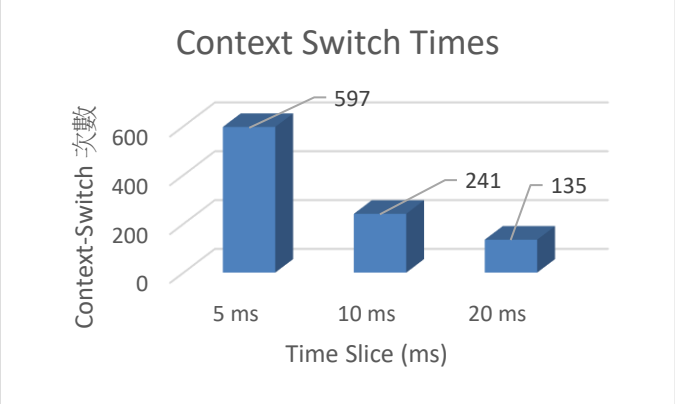

Figure 1.Context Switch times to Time Slice

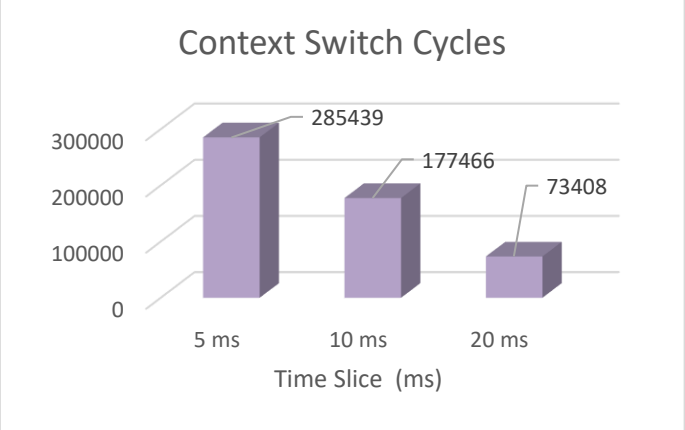

Figure 2.Context Switch Cycles to Time Slice

III. CONTEXT-SWITCH

經過前面的步驟，我們接著開始探討 task management 的重點，Context-Switch。

Context-Switch 產生的流程可以用 Figure 5.Context-Switch Workflow 來表示，首先經由 freertos_risc_v_trap_handler，不管任何狀態都會先進入這個函示，接著才會到 test_if_asynchronous 進行分流，以下分成兩條路:

A. Asynchronous

Asynchronous 這條路 trace 下去會是一條相對較深較長的行程，判斷為 Asynchronous 後會走到 handle_asynchronous 並判斷 test_if_external_interrupt，接著會分為 yes / no 兩支

1. Yes :

   甲、 首先進入 as_yet_unhandled，直到 a0 與 t1 register value 相同時，便繼續 load xTSRStackTop 並往後走到 vExternalISR，經過一個指令便進行 processed_source。

2. No :

   甲、 第一步透過 Load_from_pullmachine 取得 mtime 並且計算出 mtimecmp，經過 load_pull_next_time 、uxTimerIncrementsForOneTick 並 load xISRStack 以及 xTaskIncrementTick，在這裡會更新一些關於 tick 的變數，如 xTickCount 等等，同時此函式的回傳直式另個韓式，也就是 xSwitchRequired，會在此處判斷成式是否需要進行 Context-Switch。

   乙、 承接上述，若需要進行 Context-Switch 會先到 vTaskSwitchContext，這裡會先將 Schedular 暫停、透過 taskSELECT_HIGHEST_PRIORITY_TASK 找到下一個具有較高 prioroty 的 task 並 TCB 存到 pxCurrentTCB。

   丙、 之後歸入 processed_source，將 pxCurrentTCB load 進入 register 內就完成一次的 interrupt 了。

B. Synchronous

如果是 Synchronous 會先進入 handle_synchronous，接著 test_if_enviroment_call 判斷是否是 enviroment_call，下一步就直接進行 processed_source。

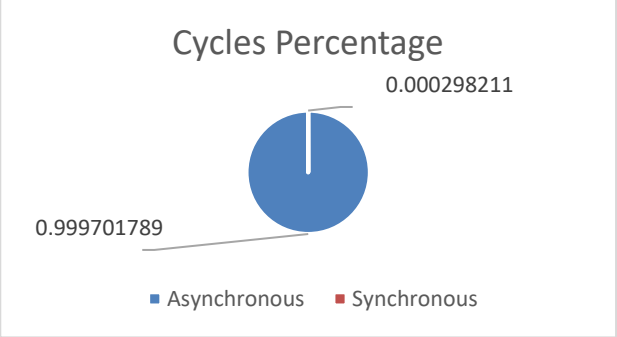

Figure 3.Cycles between Asynchronous / Synchronous

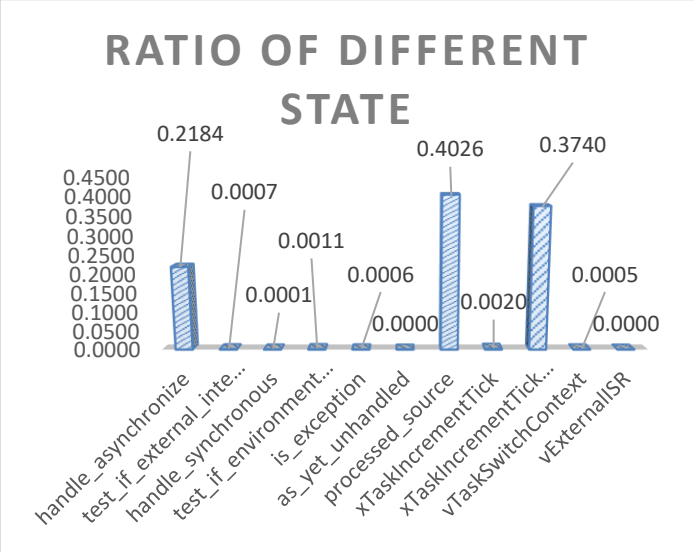

Figure 4.Ratio of Different State

IV. MUTEX

這次關於 Mutex 會分為兩個部份來進行，首先是 trace FreeRTOS 內部建立 mutex 的流程，其次是利用 Atomic Instructions 實作 Mutex:

A. Mutex in FreeRTOS

FreeRTOS 中建立 mutex 的流程相當遠長，先調用 xSemaphore 函示，而此函式背後會呼叫 xQueueGenericCreate 利用建立 Queue 的方式先宣告一個基本的 mutex 樣貌，接著使用 prcInitialiseMutex 將前步驟的 queue 調整為針對 mutex 的基本設置，例如將 pcHead 設為 Null，因為 mutex 本身不需要儲存資料不同於一般的 queue，並且將 mutext 的 task 也設為 Null，再呼叫 xQueueGenericSend，至此就完成一個 FreeRTOS 內 Mutex 的建構。

B. Mutex Implementing Using Atomic Instructions

本次在利用 Atomic Instruction 實作 Mutex 也花費我許多時間，首先是尋找要在何處撰寫程式碼，其次是在撰寫完程式碼以及成功編譯出 elf 檔案後，放到板子上執行竟然卡死在幾個 lw、amoswap.w.aq 與 bnez 指令中陷入無限循環，為此又花了好些時間處理這個問題，很可惜最後並未如願解決。

a) 關於我實作手法，此次我是以老師講義提供的範例 assembly code 取代原本 rtos_test.c 中，task1_handler 與 task2_handler 的 *xSemaphoreTake* 以及 *xSemaphoreGive*，並將早前透過 *xSemaphoreCreateMutex* 函式宣告取得的 xMutex 作為 lock_addr 傳入組語進行訊算。

b) 實作程式碼

```
#if USE_MUTEX
        // xSemaphoreTake(xMutex, portMAX_DELAY);
        asm volatile ("lui t0,%hi(xMutex)");
        asm volatile ("lw t3,%lo(xMutex)(t0)");
        asm volatile ("li t0,1");
        asm volatile ("0:");
        asm volatile ("lw t1, (t3)");
        asm volatile ("bnez t1,0b");
        asm volatile ("amoswap.w.aq t1, t0, (t3)");
        asm volatile ("bnez t1,0b");
#endif
#if USE_MUTEX
        // xSemaphoreGive(xMutex);
        asm volatile ("lui t0,%hi(xMutex)");
        asm volatile ("lw t3,%lo(xMutex)(t0)");
        asm volatile ("amoswap.w.aq t1, t0, (t3)");

#endif
```

c) 前述提及本次陷入 deadlock 的指令起自 "*lw t1,(t3)*"，因為 t1 為 0 因此可以往後進行，卻在遇到 amoswap 之後，導致更新過的 t1 不為 0，表示此時的 lock 已經被占用住因而跳回 *lw* 重新等待 lock 被釋放出來。這部份很可惜的是雖然我已經使用 ILA 搭配 omjdump 逐步分析我的程式碼執行過程，卻未能在時間內完成足夠的 bug 分析進而修正我的 atomic instruction mutex。

V. 心得

這次的作業對我個人而言算是本學期難度最高的，花上許多時間在 trace FreeRTOS 整份 code 並做整理，另外此次作業讓我收穫最多的是在一步步尋訪 code 的過程，如丙、，真正體會到軟體和硬體是如何整合在一起的，這樣

的過程對我而言十分有趣，也很感謝老師規劃這樣的課程內容!

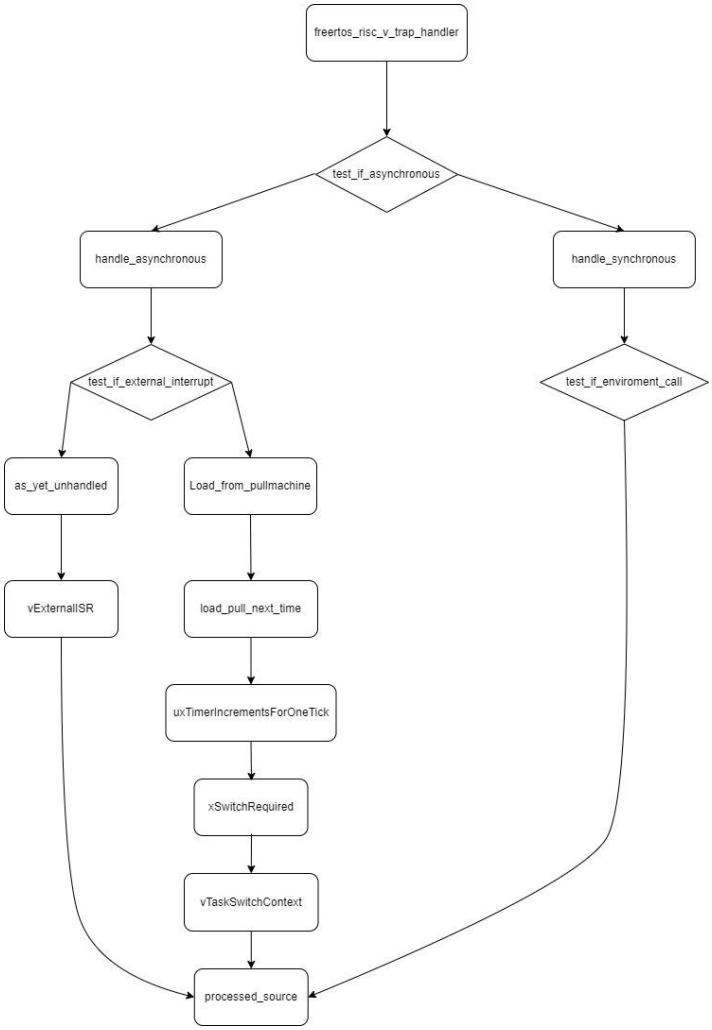

Figure 5.Context-Switch Workflow