HW#4 Report
RTOS Analysis

邱振源, 111550100

***Abstract*—這次的做作業主要是針對 real-time OS (FreeRTOS) 進行 multithreading 的分析，分析 FreeRTOS 如何處理 thread 的 task management、context-switching 以及 synchronization 的行為。同時透過 profile 電路分析這些行為的 overhead 以及改變 time quantum 對這些行為或 data cache 的影響。**

Keywords—RISC-V; Aquila; RTOS; FreeRTOS; profile circuit; multithreading; mutex; context-switching; synchronization

I. INTRODUCTION

這次的作業主要可以針對分析 RTOS 分成三個部分：分析 FreeRTOS 的 thread management、context-switching、synchronization。我的報告也會大致分成這三個部份來分析相關的行為。同時，我也討論了改變 time quantum 對 RTOS 造成的影響。

II. THREAD MANAGEMENT

A. Concept of Tasks

FreeRTOS 的 API 有支援 multithreading 的 execution，而每一個 thread 在 FreeRTOS 裡稱為一個 task，每個 task 有屬於它的 Task Control Block (TCB)，FreeRTOS 也藉由 TCB 來完成 task management。pxCurrentTCB 會記錄正在執行的 task 的 TCB，而 TCB 也會記錄一些重要的變數：

a) pxTopOfStack and pxStack

用來記錄 stack 的起始位置和 top 的位置，在讀取和刪除 task 相關資訊的時候可以協助管控。

b) uxPriority

記錄 task 的 priority，另外可以注意 0 為 priority 最低的，與在一般 OS 中 nice value 的概念不同。這是因為 task 的 list 是根據 ListItem_t.xItemValue 進行升序排列，而 xItemValue 則是透過 configMAX_PRIORITIES - uxPriority 所得。

c) xStateListItem

記錄 task 的 state，我將在下一部分進行討論。

B. States of Tasks

根據 FreeRTOS 的官方網站，task 會在四種 state 進行轉換，FreeRTOS 便藉由這些 state 轉換來 manage threads，而 Fig 1.也說明不同 state 之間的轉換關係[1]：

a) Running: task 正在被執行(正在使用處理器)。

b) Ready: task 能夠被執行，未執行的原因是已有更高 priority 的 task 正在運行。

c) Blocked: task 正在等待臨時或外部事件，呼叫 blocking 的 API 會讓 task 到這個 state，可以設立一個 timeout 時間，自動解除 blocked state。

d) Suspende: 和 blocked 相同但不能設定 timeout，只能透過 vTaskSuspend() 和 xTaskResume()進入退出。

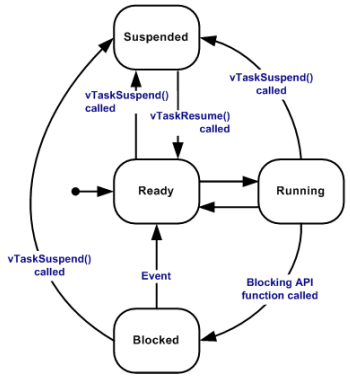

Fig. 1. Valid Task State Transitions

C. Task Creation And Scheduler

在寫完 task handler function 來描述 task 的行為之後，使用者可以呼叫 xTaskCreate 創建一個新的 TCB entry，xTaskCreate 會將 function pointer、task priority 等資訊寫入，並藉由 prvInitialiseNewTask 初始化 TCB 中的變數，透過 prvAddNewTaskToReadyList 將新建立的 TCB 加入 ready list。

接著會呼叫 vTaskStartScheduler，在這個 function 會先建立一個 idle thread，接著會設定一些重要的變數，像是將 xSchedulerRunning 設為 true，同時初始化 xTickCount，並透過 xTimerCreateTimerTask 設定相關的 timer，在最後呼叫 xPortStartScheduler 開始進行 schedule 的相關事項。值得注意的是，除非呼叫 xTaskEndScheduler，否則 vTaskStartScheduler 將不會再執行。

III. CONTEXT SWITCHING

分析 context switching 的部分主要接續前半部分，繼續分析 xPortStartScheduler 以及其呼叫的相關函式，並在分析完函式之後，討論整個 context switching 的 overall algorithm。

A. xPortStartScheduler

在這個函式會設定 mtvec 這個暫存器的狀態，同時透過 vPortSetupTimerInterrupt 來設定 timer 的資訊，並透過 xPortStartFirstTask 在 stack 上設定 context 的相關訊息。

B. vPortSetupTimerInterrupt

這個函式主要在 manage timer interrupt，他初始化像 ullNextTime 和 pullMachineTimerCompareRegister 這類可以設定下一次 timer interrupt 的變數。當 interrupt 發生時，這些變數會被載入 freertos_risc_v_trap_handler。

另外 uxTimerIncrementsForOneTick 用來計算 time slice 的長度，會決定下一次的 timer interrupt，他會被存起來用來 enable hardware CLINT。clint.v 中紀錄了 CLINT 使用 hardware clock 來計算 interrupt threshold，並根據結果去處發 timer interrupt 的發生。

C. *xPortStartFirstTask*

這個函式是透過 assembly instruction 編寫，在這個函式中，像是 mtvec 的暫存器會指向 freertos_risc_v_trap_handler，來確保系統正確的執行 context switching 與處理 interrupt，此外，這個函式也會讀取 pxCurrentTCB 的 stack 裡面存放的資料，並讀取 csr_file.v 中 mstatus 的值。

D. *freertos_risc_v_trap_handler*

透過這個函式，FreeRTOS 可以透過 stack pointer 儲存 register 的內容，並讀取 CSR 裡的 mcause。透過 mcause 的 MSB 來判斷當前的 interrupt 是 synchronous 或是 asynchronous，進而跳轉到不同的 handler function。

E. *handle_synchronous*

將 exception 接在產生 exception 的 pc 後面，再透過 test_if_environment_call 執行 vTaskSwitchContext 和 processed_source，這部分會在 asynchronous 後一並討論。

F. *handle_asynchronous*

重設 timer 至下次 interrupt 的時間點，並透過 test_if_external_interrupt 檢查是否為外部的 interrupt，如果是的話，會再次確認 mcause 是不是 machine external interrupt，不是話會跳到 as_yet_unhandled，否則便會讀取 ISR 的 stack 資料，進入外部的 external interrupt，最後再跳到 processed_source。

如果不是 external interrupt 的話便會根據 instruction bit 的不同，增加 ullNextTime 的 timer tick，同時將 ISR 的 stack 讀出並進入 xTaskIncrementTick。

G. *xTaskIncrementTick*

這個函式會設定 xTickCount，依據 task 的狀態增加不同的 tick，並藉由判斷是否有 task 比當前的 task 有更高的 priority，來決定 xSwitchRequired 的值。

H. *vTaskSwitchContext*

這個函式會先確定 scheduler 的 state 不是 suspended，接著尋找 pxReadyTaskLists 中 priority 最高的 task，並透過 traceTASK_SWITCHED_IN 來 link 要執行的 TCB，再跳回 handle_asynchronous 並接續執行 processed_source。

I. *processed_source*

這個函式會先讀出 pxcurrentTCB 中的 stack 位置，並根據 sp 設定 mstatus、mepc 和相關變數，如此將現在的變數設定成切換前的狀態，mepc 也能得到下個準備要跑的指令。

J. *Overall Algorithm*

FreeRTOS 會透過 hardware 的 timer interrupt 形式來完成 context switching，根據 time quantum 在 CLINT module 設定 mtimecmp 的暫存器，同時計算 mtime，如果 mtime 增加到跟 mtimecmp 一致，便會將 interrupt 的訊號傳到 CSR，接著便會將 pc 跳到 interrupt handler，也就是 freertos_risc_v_trap_handler。

接續的步驟細節大致與前面 A.到 I.相同，透過 mtvec 處理 handler function，用 stack pointer 分配確保 task 的 context 能被妥善保存，再來透過 mcause 來判斷 interrupt 的類型，之後用 xTaskIncrementTick 增加系統的 tick count，再透過 vTaskSwitchContext 來找到最高 prority 的 task 進行 context switch，最後 handler function 會 restore context，呼叫 mret 退出 function 等待新 task。Fig. 2 劃出了整個 algorithm 的流程，但簡化了一些不是主要的程序來讓圖表更易讀。

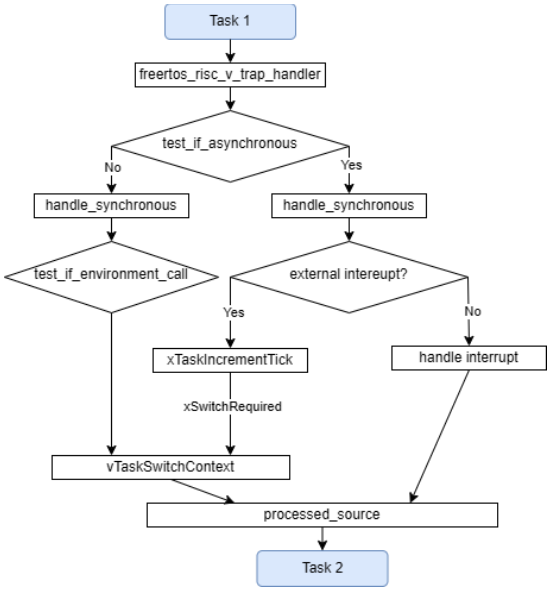

Fig. 2. Workflow of Context Switching

K. *Overhead vs. Time Quantum Size*

TABLE I. CONTEXT SWITCHING OVERHEAD VS. TIME QUANTUM SIZE

Time Quantum	**1ms**	**5ms**	**10ms**	**20ms**
Context Switch #	5065	980	492	245
Avg. Overhead	727	765	788	780
Total Cycle	207831567	202396683	201665109	201657316

我將檢測 context switching 的 profile 電路加在 csr_file.v 中，在觀察.objdump 檔的 pc 後，讓 profile 電路在進入 main 之後開始執行，兩個 task 都進入 vTaskDelete 之後結束，在這個區間中，遇到 freertos_risc_v_handler 的時候會開始判定 context switching，而在 processed_source 執行到 mret 的時候結束一次 context switching 的判定，並計算這中間經過的 cycle 數，進而判定 context switching 的 overhead。

TABLE I 記錄了在不同 time quantum 下，context switching 的次數與 overhead，可以發現隨著 time quantum 變大，需要做 context switching 的次數會越少，這樣的原因是因為 time quantum 的大小會影響多久進行一次 interrupt，另外可以發現 average overhead 的差距並不明顯，因此，降低 context switch 的次數可以有效的提升效能，然而，提升 time quantum 也有可能讓 CPU 花很多時間在做完 task 之後進行等待，依此要在 time quantum 和 context switching 的次數之間做好平衡。

IV. SYNCHRONIZATION

當使用者在 multithreading 的環境中設定了不同 task 需要有共用的變數時，要有好的同步機制來避免錯誤的修改共用變數或 race 的相關問題，最常見的是使用 mutex 這類的 semaphore 去 lock 需要修改的 critical section，我在這裡會討論 FreeRTOS 提供的兩個 synchronization APIs。

A. *taskENTER_CRITICAL() and taskEXIT_CRITICAL()*

critical section 是被保護住的一個不能被中斷的區域，而 taskENTER_CRITICAL() 和 taskEXIT_CRITICAL() 是 FreeRTOS 提供的進出 critical section 的方式，當某個 task 進入 critical section 時，FreeRTOS 會停止 task scheduler 調度任務的運行直到 exit critical section。這種方式可以很簡單的完成共享資源的保護，在臨界區域短時很有效，然而他會導致系統的其他任務無法執行，進而影響整體效能。

B. xSemaphoreTake() and xSemaphoreGive()

Semaphore 相較 critical seciton 更為靈活，semaphore 會用一個 queue 來管理對共享資源的擁有權，進而控制 task 進入共享資源的權限。擁有 semaphore 會被視為擁有最高的 priority，這樣可以避免低 priority 的 task 因為進入 critical section，擁有 mutex 而導致的 priority inversion 的問題。

FreeRTOS 提供 xSemaphoreTake() 和 xSemaphoreGive() 來實現 semaphore，需要執行的 critical section 會被夾在這兩個函式之間，當 task 執行到 xSemaphoreTake()的時候，FreeRTOS 會檢查 semaphore 是不是已經被其他 task 所擁有，如果其他 task 佔有 semaphore，FreeRTOS 會掛起這個 task 直到 semaphore 可被使用；如果 semaphore 可被使用，則 task 會擁有 semaphore。當 critical section 結束之後，task 會呼叫 xSemaphoreGive() 進而釋出 semaphore。如此讓高 priority 的 task 不會被低 priority 的 task block，更加靈活。

C. Overhead of Different Synchronization Approach

TABLE II. SYNCHRONIZATION OVERHEAD

Way	EnteCritical	ExitCrititcal	SemaphoreTake	SemaphoreGive
latency	16.25	25.98	211.75	216.78

TABLE III. CYCLE PERCENTAGE OF USING CRITICAL VS. SEMAPHORE

Time Quantum	1ms	5ms	10ms	20ms
Context Switch #	95.04%	97.35%	97.36%	97.96%
enter critical	67.52%	49.77%	49.68%	49.84%
exit critical	67.56%	49.93%	50.01%	50.00%
Total Cycle	96.49%	97.97%	98.25%	98.22%

在計算兩種不同的 synchronization 的部分，我將 enter/exit critical section 計算成 pc 進入 vTaskEnterCritical 和 vTaskExitCritical 之間的 cycle 數，而 semaphore take/give 則是計算進入 xQueueSemaphoreTake 和 xQueueGenericSend 之間的 cycle 數。同時我也嘗試將 rtos_run.c 使用 semaphore 的部分換成 enter/exit critical section 的方式，也同樣成功維護了 shared counter。

TABLE II 紀錄了 rtos_run.c 用 taskENTER_CRITICAL、taskEXIT_CRITICAL、xSemaphoreTak 和 xSemaphoreGive 的 overhead，可以發現直接用 enter/exit critical section 的 overhead 較小。TABLE III 則是記錄了使用 enter/exit critical section 的方式維護 shared counter 相較 semaphore 的結果，可以發現 context switching 的次數沒有太大的差距，然而調度 vTaskEnterCritical 和 vTaskExitCritical 的次數大幅下降，total cycle 也較少，從 TABLE II 可以發現在執行 xSemaphoreTake()和 xSemaphoreGive()的 overhead 較大，可能也是因為這樣對整體效能造成影響。

根據上表，我推測 xSemaphoreTake 和 xSemaphoreGive 可以更好的維護有複雜 priority 的 multithreading 問題，然而，像這次沒有 priority 問題的測試程式可能直接簡單的用 enter/exit critical section 的方式 block 對於 task 的調度，就可以在不用花太多 overhead 的情況下完成對 share memory 的保護，也是之後再設計 mutex 的時候應該要考慮的取捨。

V. DISSCUSS IMPACT ON DATA CACHE

A. Test Concept

由於老師上課說原本的 source code 規模太小，無法看出對 cache 的影響，因此我用矩陣乘法的運算來測試改變 time quantum 對 data cache 的影響，在 single thread 的時候，先隨機建立兩個矩陣，並計算正確答案，接著將矩陣拆成兩個部分分別讓兩個 task handler 執行，在兩個 task 都算完之後，再比較 result 與原先計算的結果來確保 mutex 正確運行。

另外我也設計了另外一部分測試成 task1 做矩陣乘法，task2 先對分開的 4 個 array 做 bubble sort，再對這四個 array 做 merge sort，想測試如果兩個 task 不是做需要共同維護的任務，context switching 會對 cache 造成的影響。

而 profile 電路的部分大致跟 Lab3 一樣，將在 csr_file 算出需要計算的部分傳入 dcache，再用 Lab3 的電路檢測 time quantum 與 context switching 對 dcache 的影響。

B. Result And Analysis

TABLE IV. CACHE MISS RATE WITH MATRIX MULTIPLICATION

Time Quantum	1ms	5ms	10ms	20ms
Context Switch #	1893	342	142	79
Total Cycle	76606103	70571002	69779683	69552062
Read Miss Rate	12.73%	12.45%	12.43%	12.40%
Write Miss Rate	5.00%	4.65%	4.65%	4.59%
Cache Miss Rate	12.06%	11.90%	11.90%	11.88%

TABLE V. MISS RATE WITH MATRIX MULTIPLICATION AND SORT

Time Quantum	1ms	5ms	10ms	20ms
Context Switch #	143	29	24	9
Total Cycle	6249029	6124789	6171514	6113166
Cache Miss Rate	7.95%	7.84%	7.83%	7.82%

TABLE IV 是用 semaphore 維護 critical section 讓兩個 thread 憶起做矩陣乘法，可以發現隨著 time quantum 的提升，無論是 read 或是 write，miss rate 都有下降的趨勢，這是因為隨著 time quantum 提升，同一個 cache line 較有可能被重複訪問到，進而提升 hit 的次數，降低 miss rate。

TABLE V 則是記錄如果兩個 task 一個做矩陣乘法一個做排序的結果，可以發現 cache miss rate 隨 context switching 的次數減少而降低，這是因為每次進行 context switching，cache 便需要讀入新的 task 的資料，原本存在 cache 的資料便有可能被替換，當頻繁切換，miss rate 自然上升。

VI. CONCLUSION

這次的 Lab 做的分析大致分成三部分：time quantum 對 context switching 的影響，synchronization 的 overhead，以及 context switching 對 data cache 的影響。再第一部分可以發現，time quantum 對 context switching 的平均 overhead 影響不明顯，但提昇 time quantum 會大幅降低 context switching 的次數，對整體的 overhead 來說影響巨大。

在第二部分可以發現，對複雜優先級的多執行緒問題，可以用 semaphore 維護 critical section，但其成本與 overhead 較大，對於簡單的測試程式，直接用 block 的方式 enter/exit critical section，就可以有效的完成對 shared resource 的保護。

最後是對 data cache 影響的部分，可以發現提升 time quantum 會降低再 multithreading 狀態下的 cache miss rate，但過高的 time quantum 可能會造成 CPU 閒置，減低效率。統合上述所有，針對不同任務選擇合適的方法是提高效能的關鍵。

REFERENCES

[1] FreeRTOS 官方網站, https://www.freertos.org/