HW5 RTOS Analysis

學號:0711282 邱頎霖

INTRODUCTION

這次作業以 HW3 的 Aquila 的環境(無 SD 卡)搭配 FreeRTOS 進行, 利用老師一開始給予的 *rtos_test.c* 做各種更改與 FreeRTOS 原始碼為主, ILA 為輔進行 trace, 首先會說明關於 FreeRTOS 內 context switch 每一個步驟是如何運作, 以及找出 context switch 的 overhead, 最後 trace 關於 mutex, semaphore, critical section 是如何在 FreeRTOS 內部實現, 雖然有些步驟過於繁瑣且需要呼叫相當多其他 function, 但搭配流程圖進行說明可以更清楚了解 mutex, semaphore 等等是如何被實現的, 最後以測量出他們的 overhead 做結束。

ANALYZE

一開始拿到原始程式碼後根據spec 調整後執行時發生不少問題， 例如無法跑出正確內容, 直到在 task2 內多輸出其他內容或是也執行 pi 程式後才順利可以跑出正常結果, 但後續仍然發生不少有關 stackoverflow 的問題, 包括我想使用原先定義在 time.h 定義好的 clock() 進行時間統計, 或是改原先定義好的 time slice 都發生類似的問題, 其中對於 clock() function 有幾次成功沒有跑出 stackoverflow 的警訊, 但是卻無法跑出結果(類似無窮迴圈的情況), 甚至重複跑出stack overflow 多達 20 次後, 結果再跑一次後就突然又正常可以顯示結果, 因此我從了解與 trace FreeRTOS 程式碼下手, 一邊 trace spec 要求的內容與分析一邊更了解 FreeRTOS 一些內部運作原理, 一開始選修這門課也是看中 Aquila 與 FreeRTOS 做結合, 因此希望可以多花時間多看這方面的內容, 也將了解到的內容寫在報告中, 之後也可以隨時回來做 review。

Task

在 FreeRTOS 中 task 等同於平常對於 process 的認知, 但更精確的對比的話應該對應為 thread, prcoess 需要 PCB 來保存狀態, task 也需要 TCB 保存當前 task 的狀態, 且後續在 context switch 時, 就是在做任務之間的切換, 這時也需要保存與載入 task 的資訊, 通通都是由 TCB 這個資料結構所負責。 透過 *tskTCB* 查看 TCB 內包含 task name, task 的 priority, 範圍由0 到 *configMAX_PRIORITIES*-1, 此值定義在 *FreeRTOSConfig.h* 在排程上 priority 為相當重要的數值, 且在 FreeRTOS 內 priority 是越小表示優先度越低, 而在排程的一開始 FreeRTOS 會先創見一個優先度為 0 的 task 再開始進行排程, 後續講述排程時還會提及相關細節, 在 TCB 內還會看到紀錄這個 task 本身的 stack 的起始位置, 最後較重要的是還分別有兩個型態為 *ListItem_t* 的 *xStateListItem* 與 *xEventListItem*, 這兩個成員與排程也有很大的關係, 如其型態名我們可以知道這兩個 TCB 成員分別為某 list 之中的內容物, 前者表示當前 task 的狀態, 根據此狀態插入該狀態的 list 中 (Ready, Blocked, Suspended), 處於非 running 的 task 屬於上述三狀態之一, 這邊可以提及 Blocked 與 Suspended 的差異, 兩者的差異我認為 blocked 屬於被動式的因為當前資源無法存取導致被設定為 blocked, Suspended 則是主動地讓自己被暫停; 另一者則表示插入 event list 中, 我們之後便可由 list 直接找到該 TCB 了。

比較細節的部份可以找到在 *tskTCB* 內會有判斷在 *FreeRTOSConfig.h* 是否有*configUSE_MUTEXES == 1* 之定義, 裡面的內容實做保存當前的 priority, 會特別需要保存是因為 Mutex 中會有 priority 繼承的機制, 後續在講解 Mutex 都還會提及相關細節。

在 *task.c* 內還有一些屬於 global 的重要資訊, 比如 *pxCurrentTCB* 表示當前 task 的 TCB, 在之後排程非常常見, 還有透過 *pxReadyTasksLists[configMAX_PRIORITIES]* 可以知道在 FreeRTOS 內是以multi-level queue 進行 schedule, 每一個 prioirty 都有一條 list, 同時也看得出 priority 最多就是到 *configMAX_PRIORITIES-1*, 另外一開始查看覺得非常奇怪的是 *xDelayedTaskList1* 與 *xDelayedTaskList2* 兩個內容, 怎麼會突然出現用數字加在變數名字後面這種感覺不是很好的命名方式的變數, 後來 trace 好一段時間才發現原來這兩者是要用來解決 tick 發生overflow , 後續也會在詳細提及。 *xTickCount* 則是用來紀錄當前發生多少次 tick interrupt, *xSchedulerRunning* 可以讓使用者知道當前是否有開啓排程, 可以看到目前為止 TCB 與 task 很大部份都是為了排程做準備。

在了解 TCB 內容後便可以回頭看 FreeRTOS 中 *xTaskCreate* 如何實現創立一個 task。 大致上分為三個步驟, 首先判定 stack 的增長方向後使用 *pvPortMalloc* 分配空間給 TCB 內的stack, 接著以 *prvInitialiseNewTask* 初始化 TCB, 包含 task name 等等上述在 TCB 內容敘述過的成員, 最後 *prvAddNewTaskToReadyList* 將此 task 加入 ready list 中。 雖然大致上步驟如上述, 但這邊有些小細節需要注意, 在加入 ready list 時會需要考慮排程器是否正在運作(開始排程), 如果在排程器尚未啓用的情況, *pxCurrentTCB* 會不斷更新為更高優先度的 task 或是與當前最高優先度但比較晚加入的 task, 也就是說最一開始執行的第一個 task 一定是被建立出來 task 中優先度最高, 如果是相同優先度則是越晚被建立出來的 task 會優先被執行; 反之如果已經開始排程則是判斷當前被建立出來的 task 優先度是否更高, 是的話則發生 context switch, 因此這時候我們就可以理解為什麼老師給的原始碼中 task2 需要先 delay, 是因為 task2 比 task1 還晚建立且兩者優先度相同, 因此 task2 會優先被執行。 此外, task 建立過程中是直接拿 Heap 空間來作為 TCB 本身一些成員的儲存空間與 stack 的空間, 因此在執行過程中如果我們發生 stack overflow 並不一定指的是 task 本身的 stack 不夠用, 也很有可能是 Heap 不夠用導致發生。

Scheduler

探討 context switch 之前我認為至少還需要看過 FreeRTOS 內有關 Scheduler 的實做細節, 而從 *vTaskStartScheduler()* 切入是個不錯的進入點。 首先在 *vTaskStartScheduler()* 裡頭會以靜態或是動態方式建立 idle task, 所謂的 idle task 其實就只是個 priority 為 0 為了排程而建立的 task, 接著關閉 interrupt 避免干擾, 初始化一些與排程有關的變數, 例如前面都有提到的 *xSchedulerRunning* 設定為 *pdTRUE* 表示現在排程器有啓動, 以及初始化 *xTickCount* 為 0 , 負責紀錄當前 tick interrup 發生的次數等等, 最後呼叫 *xPortStartScheduler* 才真正開始做排程。 而在 *xPortStartScheduler* 裡又分別呼叫兩個主要 function: *vPortSetupTimerInterrupt* 與 *xPortStartFirstTask*, 前者負責設定 mtime 與 mtimecmp, 在 RISC-V 內定義 mtime 與 mtimecmp 兩個 register, 當 mtime > mtimecmp 時發生 interrupt, 因此在 *xPortStartScheduler* 內會先拿我們在 *FreeRTOSConfig.h* 定義好的 *configCPU_CLOCK_HZ* 與 *configTICK_RATE_HZ* 先算出下一次 interrupt 的 mtimecmp 值, 一旦 mtime 大於 mtimecmp 便發生 interrupt。

在 Aquila 內部的 *CLENT.v* 內我們可以看到 mtime 與 mtimecmp 的運用, 當 *CLENT.v* 內的 counter 數到41666 的時候就會 mtime 就會 +1, 同時表示經過 1 milisecond, 與上頁描述結合在一起我們可以在拿到原始程式碼不改動的情況下使用 ILA 觀察到 mtimecmp 為 mtime + 4, mtimecmp 的推算由 *port.c* 的 *uxTimerIncrementsForOneTick* 而來, 因此我認為老師在 spec 中的說明可能並非 time quantum 為 10ms 而是 4ms 左右, 而 time slice 與 context switch 也有相當大的關係, 如果 time slice 給的越長代表相對應的 context switch 次數也隨之減少, 反之如果 time slice 過短, 將會把時間大部分花在 context switch 上而降低效能。 FIG1. 為 time slice 與 context switch 的結果表格, 可以看到的確隨著 time slice 的提高而降低了 context switch 的次數。

time slice (ms)	5ms	10ms	20ms
context switch 次數	2080	1030	512

FIG. 1. time slice and context switch

xPortStartFirstTask 顧名思義就是要開始第一個 task, 如果單純要c code 下去 trace 這段會相當不容易懂, 但如果以 objdump 觀察相對好懂許多, 可以看到 *xPortStartFirstTask* 這個function 做的事情是把 *freertos_risc_v_trap_handler* 的address 給存到 *mvtec* 內, 在一開始 trace code 的時候其實是由 Aquila 下手, 那時候就非常疑惑當發生 interrupt 的時後是怎麼進到 *freertos_risc_v_trap_handler*, 一直到後來把 FreeRTOS 大概看過後才能理解整個運作流程是發生 synchronous interrupt 或是 asynchronous interrupt, 又或是 exception 時, 都會統一先進到 *freertos_risc_v_trap_handler* 後再根據種類做處理, 而在 Aquila 內處理的方式是發生interrupt 後, 可以在 *program_counter.v* 內看到將*PC_handler_i* assign 到*pc_r*, 而*PC_handler_i* 取得方式為透過 *xPortStartFirstTask* 內 *freertos_risc_v_trap_handler* 的 address 給存到 *mvtec* 內, 發生 interrupt 或 exception 後進入 *freertos_risc_v_trap_handler*。除此之外, *xPortStartFirstTask* 的其餘工作就是將 pxCurrentTCB 的內容給 load 到 register

Context Switch overhead

接著便能切入重點: context switch 是如何產生的, 在 trace 的過程中必經 *freertos_risc_v_trap_handler*, 無論發生 interrupt 或 exception 都會先進入此 function。 因此 function 內容也相當龐大, 且附上我畫的流程圖(FIG. 2.)方便說明與清楚了解每個步驟與相對應的原始碼。 在進行任何操作前首先必須得先備份所有當前 task 的register, 接著判斷所以 interrupt 還是 exception, 一開始沒有拉 ILA 訊號計算 cycle 還算到了 exception 的 cycle, 要特別小心。 因為我們這邊探討的是 context switch, 因此著重於 asynchronous interrupt 的部份, asynchronous 與 synchronous 的差異是前者是外部來打斷執行, 後者屬於自己執行階段時主動放棄。 在判定是屬於 asynchronous interrupt 之後, 會在額外判定是否是由週邊裝置發出 external interrupt, 在 Aquila 原始碼中我們可以看到對於是不處理 external interrupt 的, 所以在流程圖便不討論發生 external interrupt 的處理方式。 在確定非 external interrupt 後會取得 mtime 的數值, 並且計算出 mtimecmp 的數值。 接著跳到 *xTaskIncrementTick*, 如同字面上的意思因為現在發生一次 tick interrupt, 因此我們需要更新一些與 tick 有關的變數如 *xTickCount*, 並且這個 function 的回傳值 *xSwitchRequired* 將會決定是否需要做 context switch, 要做 context switch 的前提有兩個, 必須要在*FreeRTOSConfig.h* 內有定義為 preeptive 非 cooperative, 且當前有 task 大於等於當前 task 的 priority 才需要 context switch, priority 等於也需要做 context switch 是因為在 preeptive scheduling 中相同優先度必須共享 cpu running。

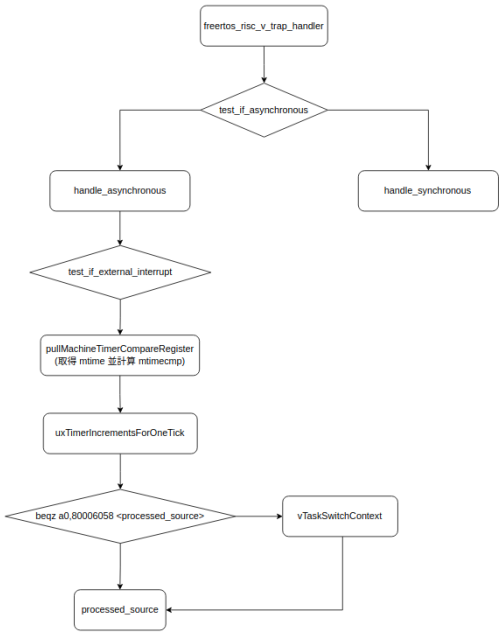

FIG. 2. freertos_risc_v_trap_handler workflow

如果判定需要做 context switch 則先到*vTaskSwitchContext*, *vTaskSwitchContext* 會先將排程器給暫停, 接著透過 *taskSELECT_HIGHEST_PRIORITY_TASK* 找到下一者具有更高優先權的 task, 存放其 TCB 到 *pxCurrentTCB*。 這邊有個細節可以注意, 一個是承上述我們需要在這個 function 內對 *xtickcount* 做加一的動作, 那一定會有 overflow 的情況發生, 因此研究稍微理解 FreeRTOS 如何針對 overflow 做處理, 當 *xTickCount+1* 後變成 0 的時候就是發生 overflow, 此時呼叫 *taskSWITCH_DELAYED_LISTS* 處理 overflow, 回顧一開始提到我在 FreeRTOS 內發現的兩個命名看起來很怪的 function: *xDelayedTaskList1* 與 *xDelayedTaskList2* 兩者就是此時派上用場, 首先 *pxDelayedTaskList* 會指向 *xDelayedTaskList1*; 而 *pxOverflowDelayedTaskList* 指向 *xDelayedTaskList2*, 先考慮 overflow 不發生的情況任何 delay 的 task 都是放到 *pxDelayedTaskList*, 這邊配合 *rtos_test.c* 內有使用到 *vTaskDelay* 進行說明 delay 如何進行, delay 的運作就是利用 *xTickCount* 再加上 *vTaskDelay* 算出來就是下一次應該 wake up 的時間點, 因此 *vTaskDelay* 的單位是 tick 並非是秒數等等, 算出下次 wake up 的時間點後存在 TCB 內的 *xItemValue*, 之後比較 *xTickCount* 與 *xItemValue* 就可以知道現在該不該 wake up 此 task, 因此 overflow 的情況才需要特別處理, 而 delay 的 task 在沒有發生 overflow 的情況下會被加入 *pxDelayedTaskList* 等待*xTickCount* 的到來, 如果是發生 overflow 的情況則是將 task 加入 *pxOverflowDelayedTaskList*, 之後透過 *taskSWITCH_DELAYED_LISTS* 將兩個 list 的內容做對換, 也就是將 *pxDelayedTaskList* 與 *pxOverflowDelayedTaskList* 的內容做對調, overflow 情況下 *pxDelayedTaskList* 應該要是 empty list 因此對調也可以說是 將 *pxOverflowDelayedTaskList* 內容複製到 *pxDelayedTaskList* 後, reset *pxOverflowDelayedTaskList* 的概念。 回到 context switch 的步驟, 最後一個步驟無論有無進行 context switch 最後都會到 *processed_source* 內, 將 *pxCurrentTCB* load 到 register 內, 整個 tick interrupt 便完成了。

分析整個 context switch 的過程中各個步驟所需要的 cycle 數量, 分別以 ILA 統計各個步驟所需要的 cycle 數量, 如 TABLE. 1. 所示, 因為每次測量的誤差大到可能多達上百 cycle, 因此這邊取的是約十次平均值, 可以看到影響需要 cycle 數量最高的仍然為 *vTaskSwitchContext*, 而且在這個作業當中我們只有兩個 task, 如果想像今天我們多達上百的 task 時在做 *vTaskSwitchContext* 時想必會更花時間。

type	cycles
handle_asynchronous	80
xTaskIncrementTick	280
vTaskSwitchContext	700
processed_source	450

TABLE. 1. cycles of each step in freertos_risc_v_trap_handler

Mutex Semaphore and Critical section

資源控管與同步任務毫無疑問是作業系統中重要的議題, 也算是此次作業重點之一, 在說明 FreeRTOS 內如何實現各個議題前, 進行 mutex 與 semaphore 之間的比較, mutex 應使用於對某項資源的控管, 且同一任務負責上鎖與解鎖, Semophore 使用於同步任務, 一個任務給出訊號後另一個任務接收信號, 最大的差異是 mutex 一定由相同任務負責上鎖與解鎖且具有優先度繼承的機制, 所謂的優先度繼承為短暫提昇某優先度較低的任務的優先度, 結束時再還原為原先的優先度, 考慮沒有優先度繼承的情況, 假如某優先度較低的 task t1 先獲得資源, 此時另一個優先度較高的 task t2 出現且也需要 t1 此時獲得的資源, 將會變成 t2 無法使用此資源而 t1 握有資源卻無法進行類似 deadlock 的情況, 如果有優先度繼承便可以先暫時提昇 t1 的優先度, 讓他順利執行任務後將資源釋放。

在將 FreeRTOS 關於 mutex, semophore 等原始程式碼看過一遍後會發現其實兩者實現時還蠻相似的, 兩者與 FreeRTOS 內定義的的 Queue 有相當大的關聯性, 在 FreeRTOS 內可以看到 list 以及 queue 等資料結構是如何被廣泛應用甚至可以說是整個 FreeRTOS 的基礎。 因為在 *rtos_test.c* 使用的是 mutex 因此從 mutex 下手, 但 semaphore 其實與 mutex 大同小異, 後續會再多說明。在 *rtos_test.c* 中呼叫 *xSemaphoreCreateMutex* 來建立 mutex, 實際上是在背後透過 *xQueueGenericCreate* 建立一個 queue。 queue 的結構可以看 *QueueDefinition* 大概有以下這些比較重要成員: *pcHead* 用來指向整個 queue 空間內起始位置, *pcWriteTo* 指向 queue 裡面下一次 push 資料進來的位置, *uxMessagesWaiting* 紀錄當前 queue 裡面有多少 item, *uxLength* 表示 queue 的最大容量, *uxItemSize* 表示 queue 內 item 大小, *xTasksWaitingToSend* 與 *xTasksWaitingToReceive* 分別紀錄為了從 queue 要寫入與讀取導致 block 的 task。 而 *xQueueGenericCreate* 的工作主要有兩個, 一者是分配空間給此新的 queue, 另一者是透過 *prvInitialiseNewQueue* 初始化 queue, 可以看到初始化過程大致上就是把 *pcHead* 分配到的空間, 並將初始化 queue 的 *uxLength* 與 *uxItemSize*, 這兩個數值都是透過 *xQueueGenericCreate* 傳入的數值。這邊有個細節需要注意, 仔細看會發現 *xQueueCreateMutex* 在呼叫 *xQueueGenericCreate* 時指定 *uxLength* 為 1 與 *uxItemSize* 為 0, 一開始看覺得奇怪為什麼 *uxItemSize* 為 0？後來了解是因為 mutex 並不像一般的 queue 是在放資料, 因此並不需要 queue 中 item 的儲存空間, 只是單純利用 queue 本身的資料結構就足夠。後續再透過 *prvInitialiseMutex* 會覆蓋調原本使用 *xQueueGenericCreate* 對 queue 的通用設定改成專門替 mutex 的設定, 而 *prvInitialiseMutex*在初始化時使用的變數大多數為 define 定義好的, 因此如果遇到非上面說明的成員基本上就是去define 找看看是不是已經先定義過, 初始化過程可以看到針對 mutex 會將 pcHead 給設定為 null, 呼應前面提到的 mutex 只需要 queue 結構本身就足以實現, 以及將持有 mutex 的 task 設定為 null, 最後呼叫 *xQueueGenericSend* 釋放 mutex, 後續也會說明到這個 function。 到此為止已經說明完如何建立 mutex,

承接建立 mutex 後我想先說明 *xSemaphoreGive*, 這個 function 主要是釋放 mutex, 而實際上負責執行的 function 為 *xQueueGenericSend*, 又是與 queue 有關的操作, 且有了上面建立 mutex 的經驗我們大概可以猜測他一定也是先呼叫一些通用的功能, 最後針對 mutex 做特別處理。

xQueueGenericSend function 應該是目前 trace 下來最複雜的內容, 搭配我畫的流程圖 FIG. 3. 應該可以更好理解與說明。 首先得說明 *xQueueGenericSend* 的重要參數, *xTicksToWait* 表示如果當前 queue 滿了願意等待多久, 因此如果後續看到 queue 滿了但是 *xTicksToWait* 又是 0 的話將會馬上 return。 第一步是先進入 critical section, 接著判斷 queue 內是否還有空間或是插入 queue 的方式為 *xQueueOverwrite* 方式, 還有另外兩種插入 queue 的方式分別為 *xQueueSendToBack* 與 *xQueueSendToFront*, 差異分別為覆蓋內容的 push 到 queue 或是從 queue 前端或是尾端插入, 如果 queue 內還有空間或是以覆蓋式插入的話多判斷有無在 *FreeRTOSConfig.h* 內設定 queue set, 使用 queue set 在訊息交換上可以提高更多效率。 使用 queue set 的話呼叫 *prvCopyDataToQueue*將要插入 queue 的內容給複製到 queue 內, 這邊的複製指的是 deep copy, 將內容完全複製一份, 然後判斷此 queue 是否屬於某 queue set, 是的話再判斷當前是否以覆蓋式插入且在複製前 queue 內的 item 數量是否不為 0, 是的話就 NOP 因為不必通知 queue set 因為內容已經被覆蓋掉, 如果當前非覆蓋式插入或是 queue item 為 0, 則以 *prvNotifyQueueSetContainer* 通知 queue set 這個 queue 內發生變動, *prvNotifyQueueSetContainer* 的回傳值也會決定需不需要做 context switch, 因為向 queue set 通知有可能導致更高 priority 的 task unblock, 如果 *prvNotifyQueueSetContainer* 的回傳值不需要 context switch 則 NOP。 反之如果該 queue 沒有屬於任一 queue set 或是在 *FreeRTOSConfig.h* 內沒有設定 queue set, 則判斷有無 task 等到了queue 裡面的 data, 如果有則 unblock, 並且判斷此 task 的優先度是否比當前 task 還高, 是的話就 context switch, 否的話就 NOP。

如果 queue 已經滿了且不能以覆蓋式插入 queue, 就得判斷 *xTicksToWait* 願意等待多久時間, 如果為 0 則離開 critical section 且馬上 return, 如果非 0 則以 *vTaskInternalSetTimeOutState* 與 *xTicksToWait* 設定 timeout 時間, 如果進到此狀態的意思就是當前的 task 因為 queue 已經滿了而處於 block 狀態, 最後離開 critical section。 一旦離開 critical section 後 task 便可以對 queue 進行讀寫, 接著呼叫 *vTaskSuspendAll()* 與 *prvLockQueue*, 前者會暫停排程使 task 不會被切換, 後者負責將 queue 給上鎖, 因為暫停排程時仍可以被 interrupt 影響進而發生 context switch, 因此最保險的方法就是替 queue 給上鎖避免被其他因素給更動。 最後判斷當前是否已經超過 timeout 時間, 如果超過的話則則將 queue 解鎖並且恢復排程後 ret, 如果還沒超過 timeout 的話判斷當前 queue 內是否還有空間, 如果已經沒有空間了呼叫 *vTaskPlaceOnEventList* 將當前 task 的 event list 加到 delay list 後解鎖 queue, 如果還有空間的話解鎖 queue 並切恢復排程再試一次。

花費這麼多篇幅後對於 mutex 還有兩個重要性質沒有提到, 一者是如何獲得 mutex 另一者就是優先度繼承, 而上面沒有提到優先度繼承是因為上面說明的是釋放 mutex *xSemaphoreGive*, 還沒說明的是相反部份的獲得 mutex 的 *xSemaphoreTake*, 但其實仔細看會發現兩者之間超級相似, 只有差在 *xSemaphoreTake* 內需要實做優先度繼承, 但也只多了約一小段程式碼, 在 *xSemaphoreTake* 裡頭會判斷當前是否是 mutex, 是的話呼叫 *pvTaskIncrementMutexHeldCount* 設定 mutex 持有者, 設定方式其實是對當前 *pxCurrentTCB* 的 *uxMutexesHeld* 做加一, 表示目前 mutex 有人正在使用而已, 最後將 *pxCurrentTCB* assign 到 queue 的 *xMutexHolder*。 而優先度繼承則是承上述, 當 mutex 已經有人使用後最後會呼叫到, *xTaskPriorityInherit*, 這個 function 就是在做優先度繼承, 他會判斷當前 xMutexHolder 內的 TCB 的 priority 是否小於當前的 task, 如果是的話則發生優先度繼承！最後使用完需要將該 task 的 priority 給還原,

則會發生在 *xSemaphoreGive* 中呼叫 *prvCopyDataToQueue* 會特別判斷當前是否為 mutex type 或是一般的 queue, 如果是 mutex 的話會額外呼叫 *xTaskPriorityDisinherit*, 利用 TCB 內部的兩個成員: *uxBasePriority* 與 *uxPriority* 判斷是否曾經發生過優先度繼承。 是的話便還原到原本的 priority。而 mutex 還有一個性質是要由同個 task 進行上鎖與解鎖, 但我在 trace 的過程中並沒有看到 FreeRTOS 原始碼有特別判定是否由同個 task 進行上鎖與解鎖, 因此自己做了幾次測試發現如果由非上鎖的 task 解鎖仍然會失敗, 必須要得由上鎖的 task 解鎖才能成功, 但卻沒有看到像是 recursive mutex 有特別判斷 task 的程式碼, 後來仍然沒有找到原因有點可惜。 可以看到原先 queue 的操作包山包海, 連 mutex 的實現也是 queue 所有操作內的一小部份。 最後附上 TABLE. 2. 為 ILA 觀察到 *xSemaphoreTake* 與 *xSemaphoreGive* 所需要的 cycle 數量。

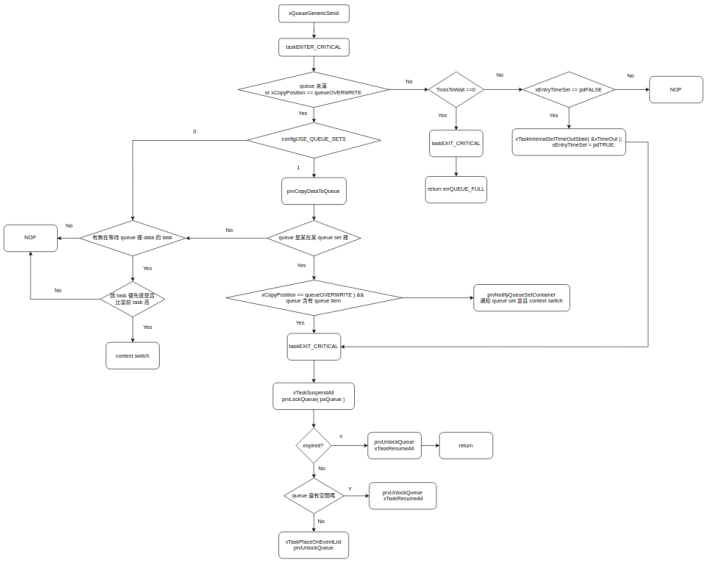

FIG. 2. *xSemaphoreGive* workflow

type	cycles
xSemaphoreTake	233
xSemaphoreGive	293

TABLE. 2. cycles of mutex

最後說明 critical section, critical section 與 mutex 和 semaphore 又有點不同, 進入 critical section 表示希望現在要進行一段不會被打斷的操作, 在 FreeRTOS 上的實現與 mutex 與 Semaphore 利用 queue 相比來的簡單許多, 進入 critical section 追朔到最後會是由 *vTaskEnterCritical* 完成, 可以看到就是將 interrupt 給關閉, 且會紀錄層數, 層數的用途在 exit function: *vTaskExitCritical* 就可以看到用途, 只有在回到最外層時才可以開啓 interrupt。 最後 TABLE. 3. 為進入與離開 ciritical section 的 cost

type	cycles
vTaskEnterCritical	35
vTaskExitCritical	50

TABLE. 3. cycles of critical section

最後附上這學期的心得, 這次作業其實是我一開始修課的目的, 因為一直對軟硬體如何整合在一起很好奇, 在前幾次作業補足了之前計算機組織沒有真的實現的功能, 雖然每次都花超級多時間, 但是對於這些內容不僅有複習到且有更加深入的理解, 還有嚴格要求報告格式讓我對 Latex 更加熟習不少, 比較可惜的是這學期末事情比較多(甚至考到 19 周QQ)不能把時間全都投進這個最有興趣的作業, 只能把重要的 FreeRTOS 與 Aquila 內的 code trace 過且拉 ILA 出來看, 卡最久的部份應該是在 trap handler 與 interrupt 是如何與 Aquila 內結合在一起, 花上幾天的時間才終於搞懂, 以及搞懂 queue 的操作也是很花時間也是比較難以理解的部份, 最後感謝老師每份作業都花時間親自批改！