HW#4 Report

賴御安, 110550168

***Abstract*—本文針對FreeRTOS處理multithreading進行分析。前面的部分先對FreeRTOS的原始碼進行分析, 提出FreeRTOS對於thread的管理及同步。後半部份則加上電路去計算context switching中會有的overhead (*Abstract*)**

Keywords*— multithreading; context switching;*Mutex; Aquila; FreeRTOS; verilog; (key words)

I. Introduction

在這次作業的前半部分, 我會分析FreeRTOS對於thread management和synchronization的處理方式, 並提出其中重要的函數和一些與aquila相關的部分。

接著會解釋這次計算overhead的counter實作時抓取的訊號, 並解釋其中的理由。同時也對得到的數據進行分析與討論。

II. FreeRTOS分析

由於這次需要trace的程式碼較多, 我選擇從rtos_run.c開始分析, 對於其中main function中有使用到的函數去做觀察, 再延伸進行分析。底下會解釋這些function的作用與重點。若遇到呼叫較大的function會在獨自寫成一個分支。但在開始分析前, 我會先帶出幾個對於thread manage較為重要的變數或是型別, 以方便後面的分析。

A. 重要變數及型別

a) struct TCB_t

這是一個struct, 用來表示一個task control block。對於task的操作就是靠這個型別的變數, pxCurrentTCB便是記錄現在正在執行的task的control block。在這個struct中有幾個變數是較常使用到的。

首先是uxPriority , 這個會記錄這個task的優先級, 其中0為優先級最大, 並非像是平常認知中的數字大的會比較大。

再來是xStateListItem, 這個變數會記錄task的state, 有會是Ready、Blocked、Suspended其中一種。

另外, 為了確保在讀取TCB時能同時得到stack pointer, 且在刪除TCB時能同時刪除task, 因此在這個struct中也有存放pxStack以記錄Stack的起始位置。

在這個struct中還有存放最後一個放進Stack的item的位置, 也就是pxTopofStack。這可以讓我們在讀取回TCB後不需要重新走過Stack就能知道新的item要放在何處。

b) pxReadyTasksLists

這是一個array type的變數, array的大小是指priority, 而這個變數是紀錄不同priority中ready的tasks。當time quantum到了之後, 會從這個array中選擇要切換的task。

B. xSemaphoreCreateMutex

xSemaphoreCreateMutex是由semphr.h的MARCO定義的, 在support dynamic allocation的時候會等同於xQueueCreateMutex, 也就是利用queue加上標記作為mutex使用。在創建mutex時, 中間會有許多檢查點, 用來確認創建的變數或指標是否正常運作, 這個部分在後面的函數中也很常看見。其中, configASSERT會中斷程式, 表示在這個點若出錯會導致整個程式錯誤執行, 而也有一些前綴為trace的函數, 根據看到的性質, 我認為是方便trace的函數。最後進行queue的初始化, prvInitialiseMutex。

C. xTaskCreate

這個函數會根據portSTACK_GROWTH的MARCO決定stack的成長方向, 為了確保stack不要長入TCB裡, 因此如果stack往下長, 則需要先放置stack再放TCB, 反之則需要先放置TCB再放stack。在FreeRTOS中, 是定義為前者。將TCB和stack創建好後, 就會進到下面兩個函數。

c) prvInitialiseNewTask

在這個函數中會去初始化這個新創造的TCB中的變數, 例如pcTaskName、pxEndOfStack等等。

d) prvAddNewTaskToReadyList

在這個函數中, 會需要先取得mutex, 因為在這個函數內會對task list進行修改, 所以需要確保不會有其他函數同時在做修改或讀取。當如果是第一個task則還會需要初始化task list。若不是第一個task則會在scheduler沒有執行時去選擇最高的priority作為currentTCB。最後會由MARCO定義的幾個函數來將新的TCB放到list的最後面。

D. vTaskStartScheduler

當從main呼叫這個函數時, 會先在這邊創造IDLE的thread, 根據是否支援static allocation來決定創造的方式。在我們的設定上是用xTaskCreateStatic來創造IDLE的thread。而在創造時, 會使用給定的user provided RAM, 並且會將priority設為0, 也就是最高優先級。

接著則是xTimerCreateTimerTask , 負責設置timer, 方法則是與創造IDLE thread類似, 取得對應位置之後用static的方式創造。

再來是對於xTickCount的設定, 這是做為紀錄發生tick interupt的變數, 由於需要確保在設定時不會同時被作修改和中斷, 因此會需要利用portDISABLE_INTERRUPT來關閉中斷的功能。這個功能會在第一個task執行時被重新啟動。在關閉中斷的功能時, 會將xTickCount初始化為0, 另外將xSchedulerRunning設為true, 表示啟動scheduler, 另外會給予next task一個延遲的時間, 以用來記錄unblock的時間。

最後則會開始xPortStartScheduler, 也就是進行schedule的函數。這邊要注意的是vTaskStartScheduler到這邊之後就不會繼續執行了, 除非呼叫到xPortEndScheduler。

III. Context Switching及其前置

接下來會著重分析xPortStartScheduler, 也就是處理multithreading的函數。底下會將xPortStartScheduler中有關Context Switching的函數一一分析, 並將分別去敘述如何去出裡synchronous跟asynchronous的情況。

E. xPortStartScheduler

這個函數是接續前部分敘述, 開始進行schedule。其中, 對於timer的設定則是依靠vPortSetupTimerInterrupt, 這個在main裡定義的函數。由這個函數可以設定mtime, 若timer已

經啟動了的話則會關閉它並重新設定。時間設定完畢就會到下一個函數。

F. xPortStartFirstTask

從這邊開始進入Timer interrupt handling routine, 也就是在portASM.S裡。因為有CLINT, 因此可以直接branch到FreeRTOS trap handler裡, 所以便會利用(G.)的函數進行處理。

而這個函數則負責讀取pxCurrentTCB裡面的資料, 也就是data in stack。同時這邊會讀取csr_file.v中mstatus的值, 並對其修改, 進而造成interrupt的發生。這邊也是先前提到disable interrupt中提到會重新啟動的位置, 具體實作方式則是將MIE bit設成1。

G. xPortStartFirstTask

在這邊會將現在變數的狀態存進新配置的空間裡。並將新的stack pointer位置給pxCurrentTCB。同時也會讀取一些csr裡的變數, mstatus,、mcause、mepc, 這些變數就是對應於csr_file.v裡的狀態。

接下來就到判斷synchronous跟asynchronous的部分。在這邊透過test_if_asynchronous作簡單的判斷。如果mepc的最高bit是1時, 就表示是synchronous, 若是0則會是asynchronous, 在後面便能以這個條件去判斷。

IV. SYNCHRONOUS & ASYNCHRONOUS

H. Synchronous (handle_synchronous)

當判斷為Synchronous時, 便會將expection放在產生expection的PC後面。接著則是執行test_if_environment_call中的vTaskSwitchContext和processed_source。這部分會等解釋完Asynchronous後再一起提到, 因為這兩種都會進入這個部分。

I. Asynchronous (handle_asynchronous)

當判斷為Asynchronous時, 便會分成兩個部分。根據mcause的值來檢查是否是外部的interrupt。底下就會分成這兩個部份去討論。

e) is external interrupt

這邊會再次確認mcause是不是符合Machine external interrupt定義的值, 若不是就會進到as_yet_unhandled而持續卡在這邊。而當mcause符合時, 就會讀取ISR的stack並進入外部的external interrupt, 最後再回到這邊並跳到processed_source。

f) not external interrupt

若進入這個部分, 則會將compare register跟ullNextTime的位置先讀出來, 並依據risc-v的instruction長度決定要執行哪種指令。當32-bit時則會將ullNextTime的low跟high word分別讀出來, 並將timer增加一個tick, 接著做完overflow的檢查後存入ullNextTime。64-bit的處理額較簡單, 將timer增加一個tick後直接存回ullNextTime。執行完後則是將ISR的stack讀出並進入xTaskIncrementTick。

xTaskIncrementTick這個函數主要的功能是要增加tick。其中會根據scheduler的狀態, 若是在suspend時則會增加xPendedTicks, 而不在suspend時則會在增加xTickCount後進行許多判斷, 確保不會有overflow、造成task timeout expired, 同時需要設定是否需要進行context switch。這些設定會在後面回到handle_asynchronous時作為判斷branch的部分設定。

回到handle_asynchronous, 這邊會根據是否造成task timeout expired來決定是要先跳到processed_source或是需要先執行vTaskSwitchContext。

J. vTaskSwitchContext

在這個函數中, 我們一樣需要先看scheduler的狀態, 若是在suspend則也會進行pending, 若不是才會開始context switching。

context switching的第一步是讀取total run time, 將其與switch in time比較並更新currentTCB的run time counter。這部分是在記錄這個task的running time。在更新時做的減法並未有overflow check是由於這個地方需要overflow來觸發更新。接下來則是將錯誤碼紀錄於currentTCB中。

當上述執行完後, 就開始選擇新的task執行, 也就是在task.c中define的taskSELECT_HIGHEST_PRIORITY_TASK, 透過一個while迴圈去找整個pxReadyTasksLists中最高優先級的task, 並將currentTCB控制的task變成最高優先級的task。這部分就是靠指標的賦值, 因為這讓我們不需要去定義每個struct的assign, 而是靠對address的操控。

K. processed_source

這個函數會做最後的處理, 讀出pxcurrentTCB中的stack的位置, 並將裡面的值拿出來更新mstatus、mepc和其他變數。這樣就會將現在的變數設定成被切換前的狀態, mepc也能得到下個準備要跑的指令。

V. 數據統計

這次總共需要計算兩種overhead, 分別是context-switch和synchronization。我將counter放在csr_file.v中, 因為這個檔案是與multithreading相關的檔案, 且裡面也有mstatus、mepc等等與其相關的變數。

L. context-switch overhead

我計算context switching overhead的方式是透過將特定reg當作flag的方式, 通過特定pc時將其值賦予為1。這時counter便會不斷增加, 直到context switching結束。這些特定pc的由來則是從rtos_run.objdump中取得。透過前一部分trace code時可以發現freertos_risc_v_trap_handler是在處理context switching時會先進入的地方。因此我將啟動flag的pc設定為freertos_risc_v_trap_handler的第一個指令位置。這邊會延伸出兩種變數, 一個是算context switching的次數, 另一個是算context switching總共花費的cycle數。第一種只會在第一個指令位置加1而已, 可以參考Fig.1。而第二種是直到最後執行的函數processed_source結束後才停止增加, 數值在Fig.2。因為當兩個task都執行完後, 仍然會持續有timer在執行, 也就仍然會進入context-switch的部分, 因此我特別去找兩個task的最後一個指令的位置, 分別用register去紀錄pc有沒有跑到最後一個指令的位置。當兩個task的最後一個指令都跑過後, 就不會觸發counter增加。這個功能在計算Synchronization overhead時也有用到。

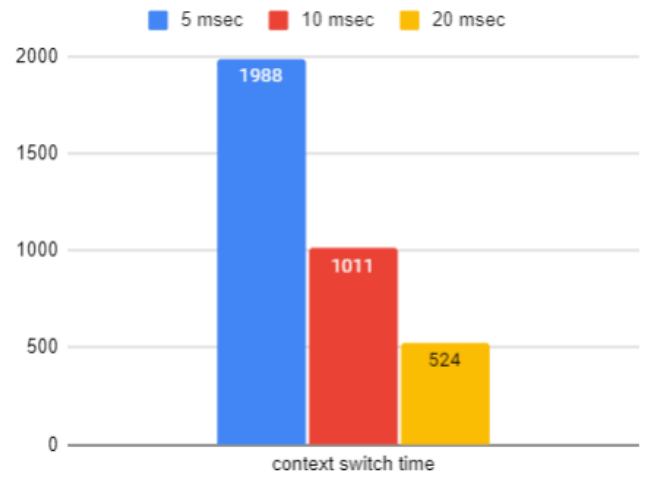

Fig. 1. context switching times in different time quantum

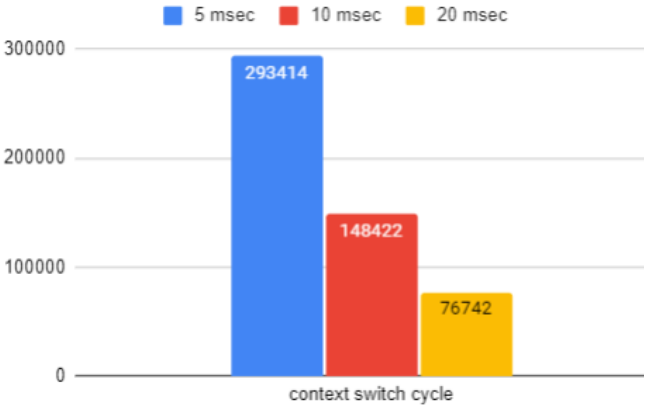

Fig. 2. context switching overhead in different time quantum

透過上面兩張圖，我們可以發現當time quantum越大時，需要做context-switching的次數就越少。這相當合理，因為我們執行interrupt的條件是當時間跑超過一個數值後。觀察context-switching的次數和cycle數也可以發現它們呈現的關係非常接近正比，也就是當context-switching發生越多次，總共所需要花費的cycle也會越多。因此當我們要調整time quantum時需要考慮這樣的花費是不是值得的。

M. Synchronization overhead

對於Synchronization overhead的計算方式則跟前面計算context-switch overhead類似，透過context_switch_flag配合pc到指定位置(handle_synchronous或handle_asynchronous的第一個指令)來觸發另一種flag，進而去計算synchronous或asynchronous的cycle。當執行完processed_source時會重置flag。我並未將freertos_risc_v_trap_handler計入cycle的原因是因為在那個函數時並不知道是哪一種，且無論哪種都需要執行它。Fig.3則是計算後的cycle數。

	5 msec	10 msec	20 msec
sync cycle	2714	1913	189
async cycle	219807	148422	5665

Fig. 3. Synchronous & Asynchronous cycles

Fig.4則是根據在Synchronization時會使用到的函數的running cycle數。不同函數的pc的區分則是依照objdump中的不同pc位置。藉由統計這些也可看出花費較多cycle數的函數，也可去檢查自己分析RTOS的行為是否合理。例如對於test_if_external_interrupt的分析中，若是使用external interrupt的話，vExternalISR就應該要有cycle，因此可以判斷在我們沒有使用這種interrupt。或是對於全部函數中，花費最多cycle數的函數vTaskSwitchContext這件事情也是合理的，因為不管synchronous還是asynchronous都會需要進入並透過while迴圈去找當下已經ready的函數。

cycle_test_if_asynchronous	2012
cycle_handle_asynchronous	19681
cycle_test_if_external	1446
cycle_xTaskIncrementTisk	3983
cycle_vTaskSwitchContext	41585
cycle_as_yet_unhandled	0
cycle_vExternalISR	0
cycle_handle_synchronous	50
cycle_test_if_environment	202
cycle_is_expection	107

Fig. 4. the total run cycle of each function in context-switching