HW4 FreeRTOS Analysis

楊永琪, 109654020

***Abstract*—This is the report of FreeRTOS Analysis. This report analyzes the multithread behavior of FreeRTOS and counts the overhead of multithreading using ILA.**

I. Introduction

In the following report, first I will describe the algorithm of the task's creation and how the scheduler starts running a task. In the third section, I will describe the context switch algorithm in FreeRTOS. The fourth section in this report describes the differences between entering critical sections and suspending taskscheduler. In the fifth part, I describe the implementation of mutex take and give in FreeRTOS. The final section provides the analysis of multithreading overhead.

II. Task Scheduler

Before starting running the tasks by calling the *vTaskStartScheduler* function, we can create some tasks first by calling *xTaskCreate.* From task creation to start running a task contains many steps, and will be described below.

A. Data Structure of a TaskList and the Predefined Variables

Before creating the task, inside FreeRTOS will contain an array of List_t with size configMAX_PRIORIY named pxReadyTaskLists which will store all the readied tasks according to their priorities. The list in FreeRTOS is a double linked list. Each item in a list is called ListItem_t. Besides two pointers, ListItem_t contains xItemValue which most of the time it stores the value of configMAX_PRIORIY - uxPriority (its own priority value), however when need to add the task to the delay list, it will store the value of xTimetoWake instead. Both are sorted in the ascending order when using the *vListInsert* function. ListItem_t also contains a pointer to the TCB_t and a pointer to List_t that owns ListItem_t itself.

Other than pxReadyTaskLists, there are few predefined variables in the program: 1. pxDelayedTaskList, which points to the tasks that are delayed by the programmer, or are blocked because they are waiting for some events to happen within a time limit set by programmer. 2. pxOverflowDelayedTaskList, which points to the task list that the tick count of the wake up time is overflowed (xTicksToWait + xTickCount < xTickCount, where xTickCount is the current tick count). When xTickCount overflow happens, the pointers' value of pxDelayedTaskList and pxOverflowDelayedTaskList will be exchanged. 3. xSuspendedTaskList, which contains the tasks that are suspended. 4. xPendingReadyList, contains the tasks that are readied when the task scheduler is suspended. 5. pxCurrentTCB, a pointer to the current task's TCB_t.

B. Task Create

TCB_t is a basic structure of a task. It contains two ListItem_t xStateListItem and xEventListItem. xStateListItem is used for referencing the state of the task, and will be put into a ready, delayed or suspended list addressed above. xEventListItem is used to be referenced by the event list, which can be put into WaitingToSend or WaitingToReceive lists in a queue or xPendingReadyList. TCB_t also contains two stack pointers, one is called pxStack which points to the end of the stack of the task(low address). Another is called pxTopOfStack, which points to the top of the stack that is being used and pxTopOfStack will always be the first element in TCB_t and on the top of the task's stack, it will always store the pc address of that task that will be returned to when the task is running again.

When calling the xTaskCreate function, it will allocate space for stack and TCB_t and initialize the items in it. After that, it will call the *pxPortInitialiseStack* function to initialize the task's stack as followed:

First, the program will read the mstatus register's value (make sure the MIE bit is disabled and MPP is set to machine mode), other registers' value and put them on the stack, then it put pxCode (the starting address of the task's function) into a0 register as the return value and will be passed in to (*pxTopOfStack), so the value that sp is pointing to will be the pointer to the start address of the task's function. This arrangement is needed when performing context switches and will be discussed later.

Next, the scheduler will insert the newly created task to the pxReadyTaskLists.

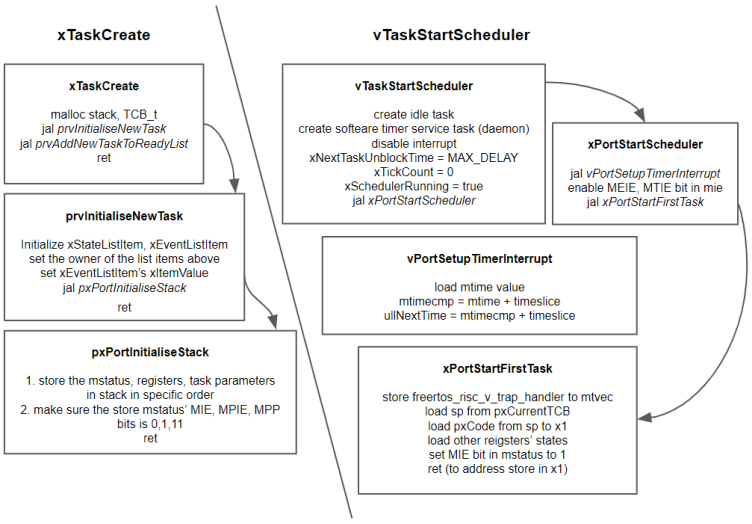

Fig I. Flow Chart of *xTaskCreate* and *vTaskStartScheduler*

C. Details on How to Start a Task

When calling the *vTaskStartScheduler* function, first it will automatically create an idle task *prvIdleTask* with priority tskIDLE_PRIORITY. What it does is just keep checking if there's another task that has the same priority as the idle task.

If there exists, it will call the *taskYIELD* function which is implemented as assembly code *"ecall"* and will raise an exception then jump to ISR and perform context switch.

After creating the idle task, it will set the mtimecmp register value by reading the mtime value and call the function *xPortStartFirstTask*. This function will first set up the mtvec register value to be the starting address of the interrupt handler routine *freertos_risc_v_trap_handler*. After that it will load the sp of pxCurrentTCB and from the value of sp, load pxCode to x1 registers, after loading other registers from the stack of the task and enable the interrupt, it will call "*ret*" to the destination address stores in x1. Because of this, a task can never call "return", it can be ended by calling the *vTaskDelete* function at the end of a task.

III. Context Switch

Context switches occur when the time slice is used and there are other tasks in the ready list with equal priority or when another higher-priority task is readied. Former is triggered by the timer interrupt, while the latter is implemented by "ecall". If some event happens, such as a message is sent to queue or received from queue, and there is another higher-priority task waiting for this event to happen, a context switch will be triggered by ecall. Either way, they will both jump to *freertos_risc_v_trap_handler* to start a routine described below.

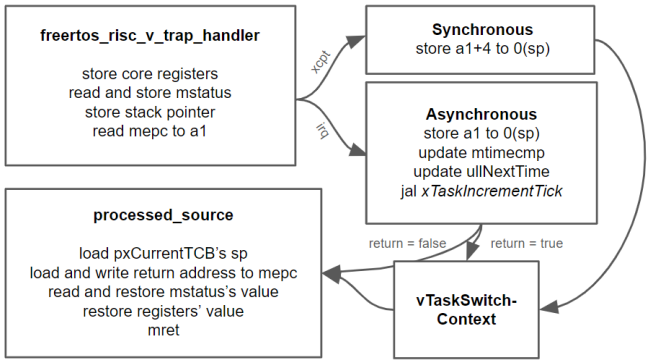

Fig II. Interrupt Service Routine in FreeRTOS

As shown in Fig.I, when an asynchronous interrupt happens, such as timer interrupt, CSR will store the oldest valid pc that is being processed in pipeline stages to mepc register. *freertos_risc_v_trap_handler* will then store all the necessary registers' states including mstatus, mepc, core registers, additional registers to the stack of current task according to a specific order. Next, it will update mtimecmp value by ullNextTime and increment ullNextTime by adding time slice.

Then, the handling process will go to *xTaskIncrementTick* to check if there are tasks in the delay list that need to be unblocked. If there are some higher or equal priority tasks that need unblocking or there are other tasks with the same priority in the ready list, it will return true, otherwise, it will return false.

If the return value is true, the system will proceed to *vTaskSwitchContext* to update the pxCurrentTCB to the suitable task, which corresponds to the next item in the highest non-empty priority ready list.

After that, the system will update the core registers value, additional registers, mstatus and mepc and call "mret" to mepc.

When the system triggers an interrupt through "ecall" instruction, it raises an exception and signals the CSR during the write-back stage. CSR then will store wbk_pc2csr+4 to mepc to avoid an infinite loop.

IV. Critical Sections and Suspend Scheduler

The pseudo code for entering the critical section is "csrs mstatus, 8" , and to exit the critical section is implemented as "csrc mstatus, 8". What these codes do is to set the MIE bit in mstatus to 0 to ignore interrupt or to 1 to enable interrupt.

While tracing FreeRTOS, I noticed that there are some scenarios where in the critical section, the system will call "ecall" to perform context switch, given that preemption is enabled. In these situations, the system will still perform context switch since exceptions are not considered as part of the external, timer and software timer interrupts. And after context switch, the mstatus will be updated by the new task's original mstatus state, so the interrupt can be still disabled or enabled depending on the task's states.

To avoid context switches, another method is to use *vTaskSuspendAll* and *xTaskResumeAll* functions to respectively suspend and resume the scheduler. When *vTaskSuspendAll* is called, timer interrupts can still occur if the interrupt is not disabled. However, the system will not perform context switch and will continue to execute the original task. Consequently, xPendedTicks will be incremented by 1 every time the timer interrupt happens.

When resuming the task scheduler by calling *xTaskResumeAll,* scheduler first checks if there are tasks in the xPendingReadyList and puts them in the ready list. Subsequently, based on the interrupt count stored in xPendedTicks, scheduler will call *xTaskIncrementTick,* to update the current xTickCount and perform context switch if needed.

V. Mutexes

Queues in FreeRTOS are used for inter-task communication. A task can send a message to a queue if the queue isn't full, and also can receive a message from a queue if the queue is not empty. If the queue is full or empty when trying to send or receive, the task will be blocked within a time limit of the value xTicksToWait set by the programmer or will be released when the queue is not full or empty again.

Semaphores and mutexes are implemented utilizing the queue structure. The differences between Semaphores and Mutexes is that mutexes have priority inheritance mechanism, and only a task that has acquired the mutex can release it. A

binary semaphore and a regular mutex use the *xSemaphoreTake* and *xSemaphoreGive* function to acquire and release the mutex. A mutex is created with an initial value of 1, after creation, a task can first acquire the mutex and then release it. Below will describe the algorithm of mutex take and mutex give.

A. *xSemaphoreTake*

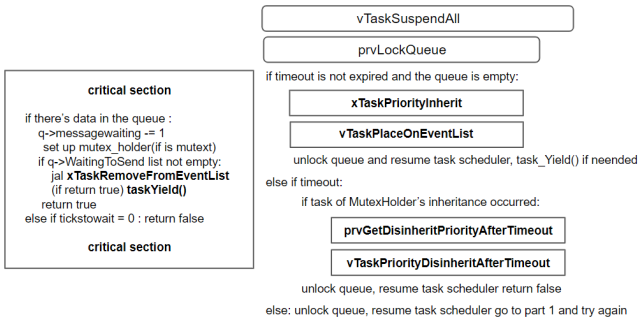

Fig III. Flow Chart of *SemaphoreTake*

When calling *xSemaphoreTake* function, it consists of two parts, and those two parts are wrapped in an infinite loop. The first part is to check if there is data available in the mutex, if there is, then decrement the semaphore count. After taking the semaphore, check if there are tasks waiting to send(give) the semaphore and unblock them if there is. If the queue type is mutex, the system will set the MutexHolder to pxCurrentTCB. These operations are performed in a critical section to protect the shared variable. If there is no resource available in the queue, the program proceeds to the second part.

In the second part, if the task has not timed out and the queue is still empty, the scheduler will put the task on queue->xTasksWaitingToReceive list, then perform a context switch. If mutex is being used, the scheduler will first make sure that the task of MutexHolder inherits the priority of the highest priority task that's waiting on the mutex. If the task of MutexHolder is in the ready list and the new task waiting on the mutex has higher priority than the current priority of the mutex holder, scheduler will remove it from the ready list and insert it to the higher priority ready list.

If timeout occurs, and task of MutexHolder did inherit other tasks' priority, then scheduler will make sure the MutexHolder inherit the highest priority among the tasks in the list of queue->xTasksWaitingToReceive (The current task that has been waiting on the mutex will be removed from queue->xTasksWaitingToReceive during *xTaskIncrementTick*). Scheduler will reset the priority of the task of MutexHolder and place it in the right priority ready list if the task of MutexHolder is already in the ready list.

If a timeout doesn't occur and semaphore is now available, the task will go to part one and try to acquire the semaphore again.

B. *xSemaphoreGive*

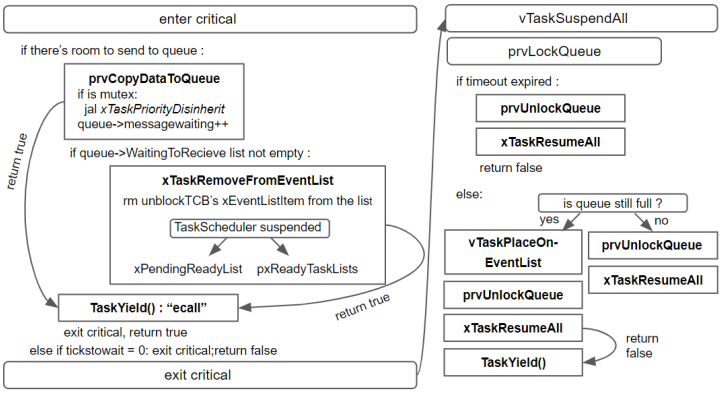

Fig IV. Flow Chart of *SemaphoreGive*

The *xSemaphoreGive* is essentially *xQueueGenericSend* with specific parameters. It also consists of two parts. First, the scheduler will check if there is room for a task to send a message to the queue(mutex). If there is, the scheduler increments semaphore's count by 1. If the semaphore type is mutex, the scheduler will call *xTaskPriorityDisinherit* to first check if the current task is the task of the MutexHolder. Second, if the holder of mutex has inherited other tasks' priority and the holder is no longer holding mutex (semaphore count is 0), the scheduler resets the task's priority and places the task into the right priority ready list. After that, the scheduler checks if there are other tasks in the WaitingToReceive list, if there are, it proceeds to *xTaskRemoveFromEventList*. If it returns true or the priority of the task is disinherited, the scheduler will perform context switch.

If the first attempt to give semaphore doesn't succeed and has not timed out, the scheduler will recheck if the queue is still full after exiting critical section. If it is not, the scheduler will attempt to give the semaphore again. If timed out, it will return false. If the queue is full, the task will be removed from the ready list and placed on the delay list, also, the scheduler will insert task->xEventListItem to queue->WaitingToSend list according to its priority. After that, the scheduler will proceed with *xTaskResumeAll,* the return value of this function indicates whether a context switch occurred during *xTaskResumeAll,* if the return is false, scheduler will yield to a context switch.

The priority inheritance mechanism in FreeRTOS is described as "simplified" and won't solve every priority inversion problem. I noticed that the scheduler won't rearrange the order of the tasks that are blocked on an event. Although the scheduler will set the xEventListItem->xItemValue to be the inheritance priority, it won't reorder the list. It will only affect the tasks that are already in the ready lists.

In the below scenario, when taskC and taskA are waiting for mutex2, since taskA has a higher priority, in the WaitingToReceive list, taskA is the next item to be unblocked when mutex1 is released. Meanwhile, taskB wants to take

mutex2 which is held by taskC, thus taskC's will inherit taskB's priority. However, when taskD releases mutex1, taskA will first acquire mutex1 since it is placed in the first item to be accessed in the queue->WaitingToReceive list.

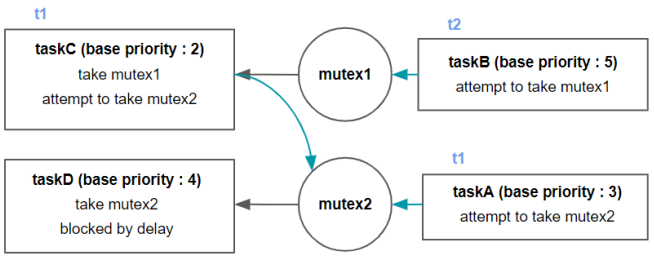

Fig V.

VI. Analysis

For calculating the overhead of multithreading, I count the average cycles of ISR, mutex take, mutex give and enter and exit critical from the moment when the processor enters task1 and before deleting task1. As the results shown in Table I, when increasing timeslice, total cycles to perform the tasks will reduce due to the decrement count of context switches.

TABLE I. AVERAGE CYCLES OF RTOS_RUN

Time Slice	**total count**	***ISR (context switch)***		***mutex take***		***mutex give***	
		cnt	***avg***	***cnt***	***avg***	***cnt***	***avg***
5ms	201,959,368	976	371.07	10002	167.88	10002	234.08
10ms	201,880,018	489	373.04	10002	156.01	10002	219.94
20ms	201,619,356	244	377.09	10002	157.72	10002	220.29
40ms	201,337,478	124	385.14	10002	157.18	10002	229.38

TABLE II. ENTER & EXIT CRITICAL SECTIONS (TIMESLICE=10MS)

Enter count	***Enter Avg Cycles***	***Exit Counte***	***Exit Avg Cycles***
20031	17	20038	27.01

TABLE III. I/D CACHE MISS & HIT RATIO

Time Slice	***I cache***			***D cache***		
	total count	***miss count***	***multi-thread***	***total count***	***miss ratio***	***multi-thread***
5ms	202,986,869	547	77.70%	13,644,355	0.291%	48.50%
10ms	202,876,188	4,080	50.29%	13,599,685	0.186%	46.32%
20ms	202,022,429	5,550	50.63%	13,508,304	0.139%	43.93$
40ms	203,401,671	14,100	49.63%	13,527,899	0.117%	42.26%

The range of testing I/D cache hit and miss rates is the same as the testing in Table I, but uses pcu_pc, exe_pc2mem to calculate instead of wbk_pc2csr. The multithread column indicates the percentage of multithread functions (mutex, enter and exit critical section and ISR) in miss count. The data presented in Table III may vary every time I run the test. After testing for a few more times, I found out that the miss ratio of D cache will decrease when the timeslice increases. However, the miss count in I cache can be quite different every time I run the test. The miss count of timeslice = 40ms is about 9000 to 14100, the miss count of timeslice = 5ms is about 470 to 500. The miss count of timeslice = 10ms most of the time is around 3900 to 4100, sometimes can be greater than 6900, and the miss count of timeslice = 20ms varies from 2700 to 5600.