# RTOS Analysis

Chih-Hsuan Chen, 110550029

***Keywords*—FreeRTOS; RTOS; Operating System; Context Switch; Multithread; Synchronization; Race; Mutex**

## I. Introduction

To fully utilize the CPU resource, achieving multi-tasking is dispensable goal for the operating system, especially in a real-time operating system. There are serveral critical issues of multi-tasking operating systems, the two of them are context switch overhead and synchronization.

Synchronization is a vital part to guarantee the correctness of share data between tasks. However, it needs more CPU cycles to using mutex to protect the critical section. Also, a large amount of time will be wasted if the context switchs happen too frequently, but the operating system isn't able to achieve real-time if context switchs are not frequent enough.

In this assignment, we will analyze the context switch and synchronization mechanism of FreeRTOS, and observe the effect from modifing some parameter settings.

## II. Context Switch

### *A. Algorithm*

FreeRTOS uses multilevel queue scheduling. Each priority has its own ready queue (list), which contains all ready tasks having that priority, and each queue has a index pointing to the task that was chosen to run. The tasks in the same queue will share the time slice of that queue.

The main procedure of context switch is defined in vTaskSwitchContext() in task.c:

1. Check that the scheduler is not suspended.
2. Switch out the current running task.
3. If configGENERATE_RUN_TIME_STAS is true (the system should generate runtime statistics), compute the total running time of the current running task.
4. Check for stack overflow.
5. Store the system errno to the TCB of the current running task.
6. Choose the ready task with highest priority using taskSELECT_HIGHEST_PRIORITY_TASK() and switch it in.
7. Load the errno from the TCB of the new task as the system errno.

taskSELECT_HIGHEST_PRIORITY_TASK(), which is mentioned in 5., is a function also defined in task.c. And the following is the procedure:

1. Find the non-empty ready queue with the hightest priority.
2. Move the index of the queue to the next task. If it doesn't point to the end marker of the queue, the task pointed will be chosen as the new task. Set its TCB as the TCB of current running task.

Since the index of a queue indexes through that queue, all the tasks in the same queue get an equal share of the time slice.

As for the conditions that trigger a context switch. vTaskSwitchContext() will be called when:

- There is a timer interrupt and some conditions are satisfied. (Refer to test_if_timer in handle_asynchronous in portASM.S)
- There is an environment call from M-mode. (Refer to test_if_environment_call in portASM.S)
- The current running task is suspended and there are some ready tasks. (Refer to vTaskSuspend() in task.c)

The function xTaskIncrementTick() will be called before vTaskSwitchContext() in the first situation that a timer interrupt happens, and vTaskSwitchContext() will be called if return value of xTaskIncrementTick() is true. xTaskIncrementTick() checks if there are some tasks should be unblock and if they have higher priority then the current running task, or there are some ready tasks having the same priority as the current running task. If one of these conditions is satisfied, xTaskIncrementTick() returns true, which means that it requires a context switch. The following is the procedure of xTaskIncrementTick():

1. Check that the scheduler is not suspended.
2. Increase the RTOS tick and see if the wake time of some blocked tasks is arrived. If so, remove them from the blocked (delayed) queue and put them to the ready queues.
3. If there are unblocked tasks having higher priority than the current running task and the system is set preemtive. Return true.
4. If the ready queue that has the same priority as the current running task is not empty and the system is set preemtive. Return true.

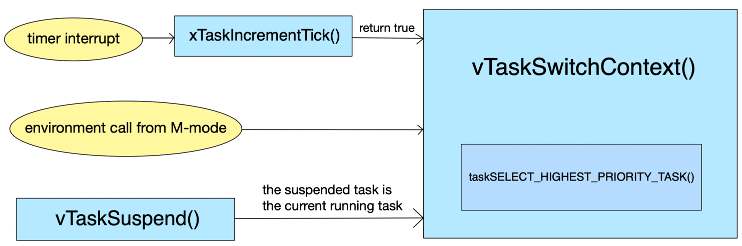
timer interrupt
xTaskIncrementTick()
return true
vTaskSwitchContext()
environment call from M-mode
taskSELECT_HIGHEST_PRIORITY_TASK()
the suspended task is the current running task
vTaskSuspend()

Fig. 1. The trigger conditions of context switch

### B. *Overhead*

*1) Implementation:* Two counters ”total” and ”cxtsw” and one flag ”counting” are added in aquila_top.v. The counter ”total” counts the total cycles from the process start to end. If there is a timer interrupt, the flag ”counting” will change to true, and the counter ”cxtsw” starts counting. When the PC falls into ”processed_source”, ”counting” changes to false and ”cxtsw” stops counting.

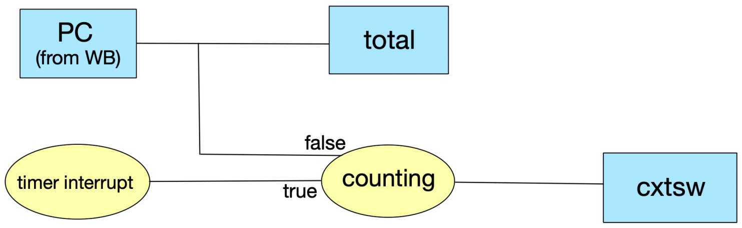

Fig. 2. The logical design of context switch analyze circuit

*2) Result:* Figure 3. shows the analyze result of context-switching overhead with different time quantum. The overhead is computed as the ratio between total cycles and cycles during context switch. Frequency 25Hz, 50Hz, 100Hz, 200Hz, 400Hz, 800Hz were tested, and we can find that the line is almost linear, indicates that the context-swiching overhead is proportional to the frequency, inversely proportional to the time quantum, as the slide of this assignment mentioned.

However, the test case has only two tasks (except the IDLE task), while there are often much more tasks run simoutaneously. Thus further testing is needed to see whether the context-swiching overhead is still proportional to the frequency under such complex condition.

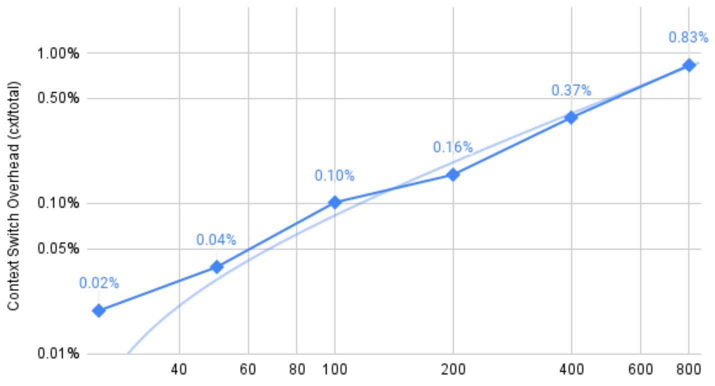

Fig. 3. Context-switching overhead with different time quantum

## III. Synchronization

### A. *Algorithm*

Except for directly using the APIs for entering/exiting a critical section , FreeRTOS uses mutex and semaphore to implement synchronization, and semaphore can be distinguished into binary semaphore and counting semaphore, whose value can be set to a maximum of one or more than one, respectively. Mutex and semaphore are quite similar, but they still have some difference and different usages:

- Mutex:
  - Priority inheritance mechanism: If a task gets a mutex, its priority will be increased to the maximum priority among the tasks waiting for the mutex, until it unlocks the mutex. This mechanism can reduce (but doesn’t cure) the effect of priority inversion.
  - Cannot be operated by interrupts
  - Better choice for implementing simple mutual exclusion
- Semaphore:
  - No priority inheritance mechanism
  - Can be operated by interrupts
  - Binary Semaphore: Better choice for implementing synchronization (between tasks or between tasks and an interrupt)
  - Counting Semaphore: Typically used for counting events and resource arrangement

Both mutex and semaphore are implemented with queue in FreeRTOS. There are some critical attributes of queue that will be mentioned later:

- uxMessagesWaiting: The number of items in the queue
- uxLength: The maximum number of items can be in the queue
- uxItemSize: The size of each item in the queue
- xTasksWaitingToSend: The tasks blocked waiting to give the mutex/semaphore. Ordered by their priority.
- xTasksWaitingToReceive: The tasks blocked waiting to take the mutex/semaphore. Ordered by their priority.
- uxRecursiveCallCount (only be used in recursive mutexes): The number of times that the holder of the mutex takes the mutex

Both mutex and semaphore are special types of queue that have item size zero, because it doesn’t need to really read or write data to that queue but only considers and operates the number of items in the queue (uxMessagesWaiting). Also, set the item size to zero can help the APIs for queues to specify whether the queue is actually a mutex/semaphore without checking the queue type twice or more. The number of items in the queue means the value of the mutex/semaphore. Note that since such queue will be accessed by multiple tasks and some of the operations cannot be disrupted, the operations that will modify the queue (such as take/give the semaphore) will use APIs for entering/exiting a critical section to protect the queue.

The basic and important operations to a mutex/semaphore are create, take and give. These operations have serveral variations for static, interrupts (only for semaphores) and recursive mutex (mutex that can be take more than once by the same task, and should be given the same times to unlock in FreeRTOS. Here we focus on the basic version.

*1) Create:*

- Mutex:

    1. Create a queue whose length is one, item size is zero and type is mutex.
    2. Use prvInitialiseMutex() to initialize the queue to a mutex.
        a. Set the holder of the mutex as NULL.
        b. Set the queue type as mutex.
        c. Set uxRecursiveCallCount as zero.
        d. Give the mutex. The value of a mutex/semaphore should be initialize to one so that it can be taken by a task.

- Binary Semaphore:
    1. Create a queue whose length is one, item size is zero and type is binary semaphore.
    2. Give the semaphore.
- Counting Semaphore:
    1. Given the max value and initial value of the counting semaphore, create a queue whose length is the given max value, item size is zero and type is counting semaphore.
    2. Set uxMessagesWaiting as the given initial value.

*2) Take:*

1. Check whether the item size of the queue is zero to determine whether the queue is a mutex/semaphore.
2. Check that the scheduler isn't suspended.
3. Enter the critical section.
    4. If the queue is not empty, which means that the value of the mutex/semaphore is larger than zero and it can be taken. Decrement uxMessagesWaiting.
    5. If it is a mutex, set its holder as the current running task, and increase uxMutexesHeld of that task.
    6. If there are tasks that are blocked waiting to give the mutex/semaphore, and unblock the one that has the highest priority. And this task will preempt the current running task if it has higher priority.
    7. Exit the critical section and return.

   (If this function hasn't returned, it means that the queue is empty.)
8. Lock the queue.
9. Check whether it is a mutex. If so, enter the critical section.
    10. Do priority inheritance. Inherit the priority from the current running task if its priority is higher than the mutex holder's.
11. Exit the critical section.
12. Push the current running task into xTasksWaitingToReceive.
13. Unlock the queue.

*3) Give:*

1. Check whether the item size of the queue is zero to determine whether the queue is a mutex/semaphore.
2. Check that the scheduler isn't suspended.
3. Enter the critical section.
    4. Check if the queue is not full, which means that the value of the mutex/semaphore is less than the maximum value and it can be given.
    5. Do function prvDataCopyToQueue:
        6. Check whether the item size is zero (and the item size of a mutex/semaphore is exactly zero). If so, it doesn't really need a data copy.
        7. If it is a mutex, disinherit the priority (the original priority is stored in its TCB), and set the mutex holder to NULL.
        8. Increase uxMessagesWaiting of the queue.
    9. If there are tasks that are blocked waiting to take the mutex/semaphore, and unblock the one that has the highest priority. And this task will preempt the current running task if it has higher priority.
    10. Exit the critical section and return.

   (If this function hasn't returned, it means that the queue is full.)
11. Lock the queue.
12. Push the current running task into xTasksWaitingToSend.
13. Unlock the queue.

*4) Critical Section:* The APIs for entering/exiting a critical section are implemented by disabling interrupts, and calls for these APIs and be nested

- Enter:
    1. Disable interrupts.
    2. Increase the critical nesting depth variable of the current task.
    3. Check that it is not used in an interrupt.
- Exit:
    1. Check that the scheduler isn't suspended.
    2. Decrement the critical nesting depth variable of the current task if it is larger than zero.
    3. If the critical nesting depth becomes zero, enable interrupts to exit the critical section.

## B. Overhead

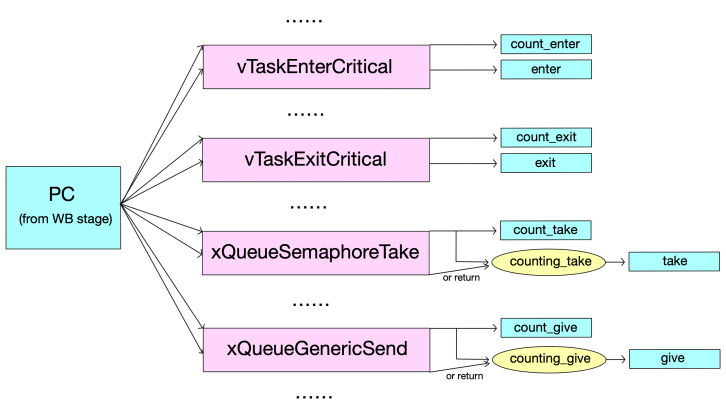

Fig. 4. The logical design of synchronization analyze circuit

*1) Implementation:* Eight counters ”enter”, ”leave”, ”take”, ”give”, ”count_enter/leave/take/give” and two flags ”counting_take/give” are added in aquila_top.v. They count the total cycles and the times when the PC (from WB stage) falls into function of enter a critical section, exit a critical section, take a mutex/semaphore and give a mutex/semaphore, respectively. Since the take and give functions will call other functions, it cannot just simply detect whether the PC is in the functions to determine whether take/give operations are processing. It needs flags to record whether it should count.

Once the PC falls on the functions beginning, increase the counter ”count_enter/leave/take/give”, and change the flags ”counting_take/give” to 1; if the PC falls into the enter/leave critical sections functions, increase the corresponding counter ”enter”, ”leave” until the PC leaves the functions. If the PC falls at the end of take/give functions or ret in the middle, change the corresponding flag to 0. The counters ”take” and ”give” will be increased in each cycle if their flags are 1.

*2) Result:* Table I shows the analyze result of synchronization overhead of different operations. We can observe that the times of leaving a CS $>$ entering a CS; also, the times of giving a mutex $>$ taking a mutex and they are almost equal. It is reasonable because if every critical section has exited correctly, the times of entering a critical section/taking a mutex will be no more than the times of leaving a critical section/giving a mutex.

We can also find that the overhead of directly entering/leaving a critical section is significantly lower than using a mutex. This is because what the former does is just disabling interrupts, while the opertions of the later are much more complex as mentioned, and the APIs for directly entering/leaving a critical section are used in the take and give functions originally.

TABLE I
OVERHEAD OF SYNCHRONIZATION

| | *enter a CS* | *leave a CS* | *take a mutex* | *give a mutex* |
|---|---|---|---|---|
| **total cycles** | 348,282 | 623,003 | 1,810,951 | 2,131,358 |
| **times** | 19,382 | 58,299 | 19,100 | 19,102 |
| ***avg cycles*** | **17.9694** | **10.6863** | **94.8142** | **111.5777** |

## IV. DISCUSSION

We have seen that the overhead of context-switch indeed closely depends on the time quantum. However, a vital goal of a RTOS is to respond in real-time, and it needs a small enough time quantum, thus we cannot expand or shrink the time quantum too much to reduce the overhead or increase real-time responsiveness. We must find the balance between them according to the usage and goals of the OS.

On the other hand, to enter a critical section, directly disabling interrupts is significantly faster than using a mutex or a semaphore, but if the critical section is too long, the task that enters the critical section will occupy the whole CPU time and lower the real-time responsiveness of other tasks since no interrupt can disrupt it. Therefore, we have to decide which method to use according to the length of critical sections.

## REFERENCES

[1] http://www.openrtos.net/a00106.html
[2] https://www.freertos.org/features.html
[3] https://stackoverflow.com/questions/35011322/why-not-to-use-mutex-inside-an-interrupt
[4] https://www.geeksforgeeks.org/difference-between-priority-inversion-and-priority-inheritance/