Lab4, RTOS Analyis

JUN-KAI CHEN, 111550001

***Abstract*—In this paper, I analyze the concept of context switching and mutex behavior as performed by the RTOS running on the Aquila SoC. In particular, it explains how these two tasks are executed and measures the associated overhead by adding a simple counter.**

***Index Terms*—Aquila, FreeRTOS, Context Switch, Mutex**

I. Context Switch

To understand how a context switch occurs, we must separate it into two parts: the hardware side and the software side.

A. Hardware

The context switch perfromed by the FreeRTOS is trigger by an interrupt signal. For Aquila, the CLINT module creates interrupt signals regularly. Once the counter meets the threshold, the signal is asserted and then flow to the CSR file. After handling the interrupt, the program counter should be set to the address based on the `mtvec` and `mcause` regsiters. The original address should be saved to the `mepc` register and used by the `mret` instruction.

On the hardware side, the timer interrupt is handled similarly to a function call. Thus, its behavior is simpler than that of the software side. The only thing to consider might be the impact on the cache, as switching contexts may require flushing the caches.

B. Software

Based on the concept of hardware, we know that if the machine hopes to perform the context switch task, it must modify the `mtvec` register before it enable the time interruption.

From the program of the main function, we can know that it will simply create two tasks and call `vTaskStartScheduler` to invoke these two tasks. The scheduler function creates an idle task in case the CPU has no jobs. Once there is no task to execute, the scheduler will switch to the idle task. Finally, it jumps to the `xPortStartScheduler` function, which writes the address of the `freertos_risc_v_trap_handler` function to the `mtvec` register by invoking the `xPortStartFirstTask` which is written in assembly code. In this way, whenever an interrupt happens, the processor can handle it properly.

It can be seen that when a timer interrupt occurs, the processor's work will be directed to the trap handler function. The working flow of the trap handler function is diagrammed in Fig. 1, since this function contains some code that handles other types of interrupts, I've shown the parts not related to the context switch in gray. This function invokes the `xTaskIncrementTask` function to know whether the context switch should be perform by the `vTaskSwitchContext` function.

The flow chart of the last two functions are shown in Fig. 2 and Fig. 3 respectively. Some of the macro function used by the functions are empty. My guess is that these functions may not be needed in the application in this homework. These macro functions are not shown in the flow chart because of simplicity. I have also simplified the flow chart slightly for the same reason.

As can be seen from the charts, the timer interrupt will update the system information, including the blocked task and the timer, by the `xTaskIncrementTick` function. The result of this function will determine whether the context switch is performed by the `vTaskSwitchContext` function.

The `vTaskSwitchContext` invokes the `taskSELECT_HIGHEST_PRIORITY_TASK` macro function to get the highest priority level among the tasks that can currently be executed. Through the `listGET_OWNER_OF_NEXT_ENTRY` function, the next task to be executed is obtained according to the priority and set as the current task (stored in `pxCurrentTCB` variable). Finally, the priority record is updated to current level.

When these two functions end, the remaining program of the `freertos_risc_v_trap_handler` function will restore the machine status according to the content of the `pxCurrentTCB` variable to complete a context switch.

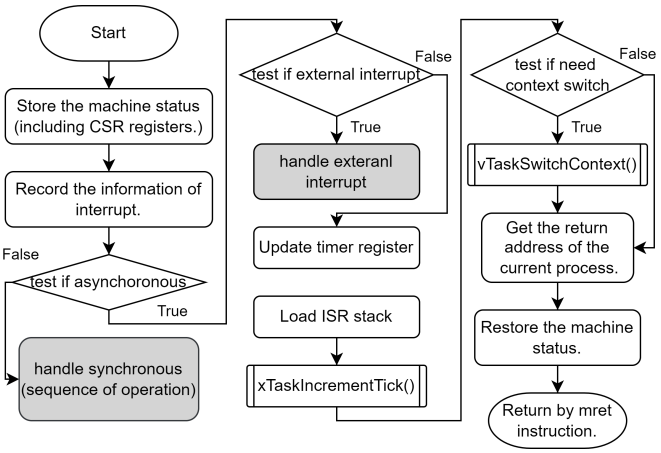

Fig. 1. The flow chart of `freertos_risc_v_trap_handler` function.

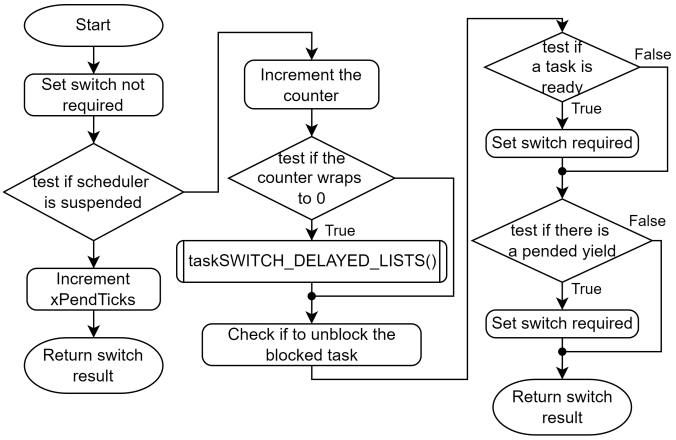

Fig. 2. The flow chart of `xTaskIncrementTick` function.

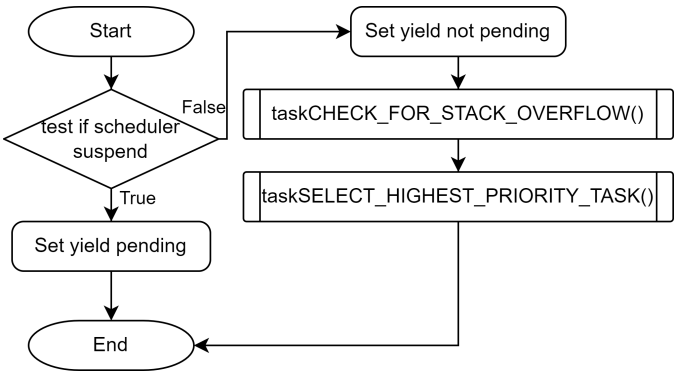

Fig. 3. The flow chart of `vTaskSwitchContext` function.

C. Statistic

Since the overhead of context switching is a concern, I designed a module to count the cycles from when a timer interrupt arrives until the `mret` instruction is reached. In addition to timer interrupts, I also calculated the number of calls and execution time of the three functions drawn as flow charts.

The average time spent by each function at different time quantum is recorded in Figure 4. For reference, I recorded the number of timer interrupt under different time quantums in Table 1. The number of triggers of other functions is basically the same, but the trap handler function seems to be used in other places, and the number of triggers is slightly higher.

Judging from the chart, the processing time of the timer interrupt does not seem to have a clear relationship with the time quantum. Other functions take more time at low trigger frequencies, with the exception of the xt function at 10000 Hz. His time spent increased a lot. I guess the reason is that the scheduler has to spend more time processing blocked tasks.

II. Synchronization

To approach data synchronization, mutex and semaphore are the method to be used. Semaphore is usually used when a task wakes up another task. In contrast, mutex is more suitable

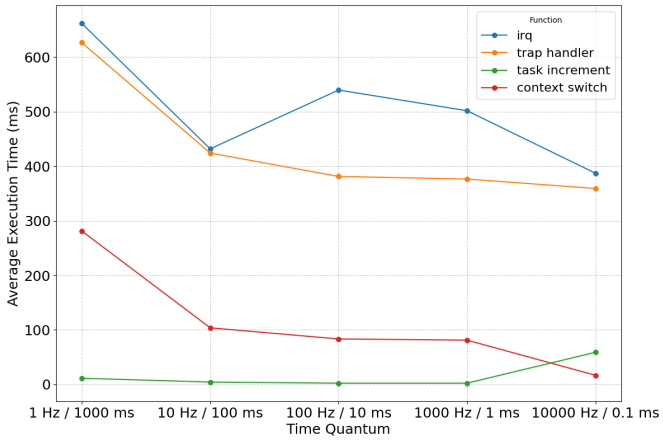

Fig. 4. Average Execution Time vs Time Quantum.

TABLE I
THE NUMBER OF TIMER INTERRUPT UNDER DIFFERENT TIME QUANTUMS.

time quantum (Hz)	number of interrupts
1	6
10	48
100	482
1,00	4,870
10,000	53,107

for this homework since we need to ensure that only one task could accesses the shared resources at the same time.

Another interesting thing is that, though the term ‘semaphore’ is used here, the implementation of it is based on queue, which is the method used for multiple tasks to communicate in FreeRTOS.

A. Mutex Initialization

The `xSemaphoreCreateMutex` function will be used to create and initialize the mutex. Since the program supports dynamic allocation, this function will call another function called `xQueueCreateMutex` and pass an argument. This queue function creates a queue with a length of 1 and a size of 0 and then initializes it. Initialization clears the owner of the queue and sets the type of the queue to mutex.

B. Mutex Take

The mutex take function `xSemaphoreTake` is directed to the `xQueueSemaphoreTake` function. Fig. 5 is a flowchart inllustrating this function. Simply put, this function checks whether the mutex is available and accesses it if so. Otherwise it will try to update other work that is waiting and try to wait for the mutex to be available until it times out.

C. Mutex Give

The mutex give function `xSemaphoreGive` is directed to the `xQueueGenericSend` function, with Fig. 6 be a flowchart illustrating this function. If the mutex is available, it releases the lock and checks whether there are other tasks waiting for the mutex to ensure the efficiency.

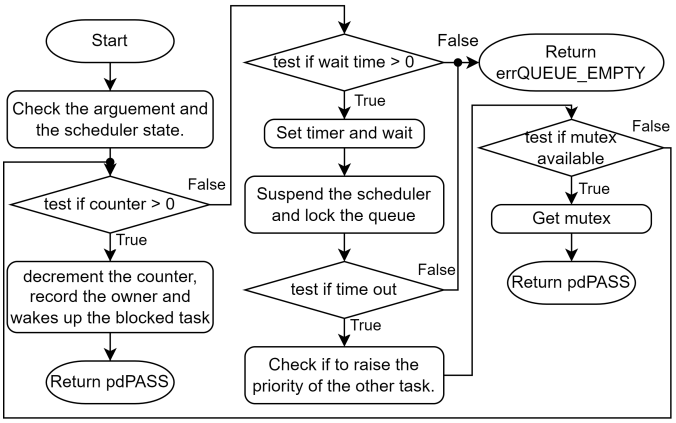

Fig. 5. The flowchart of mutex take function.

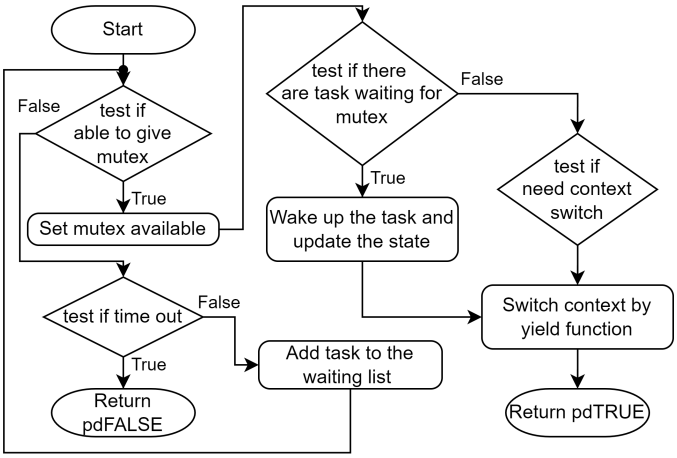

Fig. 6. The flowchart of mutex give function.

D. Statistic

Unlike context switch, the cost of using mutex does not change significantly at different time quantum. For reference only, I recorded the data in Table 2 with the time quantum be 100 Hz.

From the experimantal data, although the cost of establishing a mutex is higher, it only needs to be executed once. In comparison, firstly, the complete operation of using and releasing a mutex requires about 340 cycles. If a mutex is used to protect a frequently accessed data, the impact on time will be greater.

TABLE II
THE COST OF USING MUTEX.

function	number	average cost (cycles)
mutex create	1	3,275
mutex take	9,999	148.75
mutex give	9,996	191.95

TABLE III
THE MISS NUMBER OF INSTRUCTION CACHE AT DIFFERENT TIME QUANTUM.

time quantum (Hz)	miss number
1	508
10	642
100	1,188
1,000	6,941
10,000	16,286

III. CACHE

For time sharing OS such as RTOS, a context switch may cause the cache to flush and re-fetch data since different task is accessinf CPU. Therefore, it can be assumed that if context switch occurs more frequently, the number of cache accesses to main memory will also increase.

I collected the number of times the two caches read and wrote to the main memory under different time quantums. Since the instruction cache cannot be written to, this section is not listed.

Considering that the tasks executed in the job program do not frequently access data, the number of data cache errors will not change with the time quantum. Therefore, I only include the results of instruction cache in Table 2.

As shown in the table data, the number of instruction cache misses is positively correlated with the switching frequency, but the two are not directly proportional. My idea is that if context switch happens frequently, the time allocated to a thread will be shorter, and only a small part of the instruction cache will be updated. When switching back to the original thread, it shall encounter fewer cache miss. Therefore, the miss rate is not proportional to frequency.