HW4 Report
RTOS Analysis

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Student ID: 081358

***Abstract*—Time quantum set-up in an OS system would affect how many context switching should take to execute a program. Different time quantum is tested and the default 10ms works pretty well in our experiment. A set of RISC-V Atomic instruction built mutex get and give functions are also tested along side Peterson's algorithm. They work better than the default FreeRTOS synchronization method at the cost of not handling priority inversion.**

***Index Terms*—Real Time OS, context switching, mutex, RISC-V, Aquila**

I. Tracing thread management in RTOS

FreeRTOS supports multiple thread execution through a series of API. Each thread in FreeRTOS is called a Task. Each task has its own Task control block(TCB) recording the task's run time, including its own stack and register values. Tasks are labeled by four states: run, ready, suspend, and block. Fig.1. shows the relation between states and events to trigger or the API supported to move between states.

The user would write a handler function to describe the behavior of the task. The function must locate in an infinite loop without breaking out. The only way of terminating the tasks is through the $xTaskDelete$ function. RTOS would clean up it's resources and recycle the data structures. To register the task for execution, we would call $xTaskCreate$, passing in the function pointer of the handler, the task priority and other task related information. RTOS creates a new TCB entry, fill in the information and push the TCB into the stack. When the task ends, $vTaskDelete$ would remove the TCB entry from the stack.

After setting up the handler function for all tasks and place them in the stack. We would call $xTaskStartScheduler$ to start the task scheduler. The function sets up an IDLE thread with priority 0 (the lowest priority). Then select the most prioritized task. Tasks with even priority would share execution time by round robin. The system clock now starts ticking and would interrupt the execution by the clock. Context switching between two tasks is storing the task state in its TCB and put the task into ready state. Select the next ready task and start execution. We would further discuss context switching in the next section.

II. Context-switching in RTOS

Context-switching is RTOS is initiated by hardware timer interrupts. Hardware CLINT module would send out timer interrupt signals regularly by monitoring the mtime and mtimecmp registers. When mtime meets mtimecmp, a machine timer interrupt signal is sent to the CSR module to set up control registers representing a timer interrupt. If the interrupt is confirmed, the CSR module would contact the program counter module to jump to the interrupt handler function, $freertos\_risc\_v\_trap\_handler$.

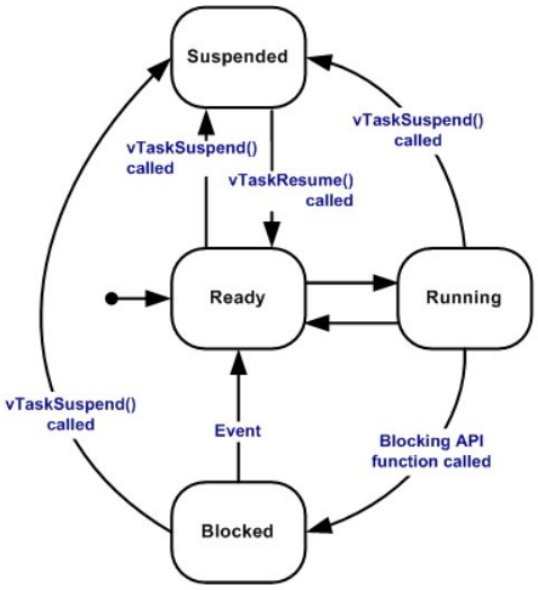

Fig. 1. FreeRTOS valid task state transitions

In the trap handling function, RTOS stores the current register file and all other registers. Then reads the interrupt cause from mcause register from the CSR module and test if the interrupt is synchronous. In the handle of asynchronous interrupt, we rewind the timer for next timer interrupt and jump to $xTaskIncrementTick$. The function increments the counter for each task, unblock any blocking task if their waiting time is up. After updating the time of the task queue, it determines if there's any available task of equal or higher priority than the current running task. If yes, a context-switch shall be performed by calling another function. A hook function, $vApplicationTickHook$, would be call by $xTaskIncrementTick$ too, since its defined as empty

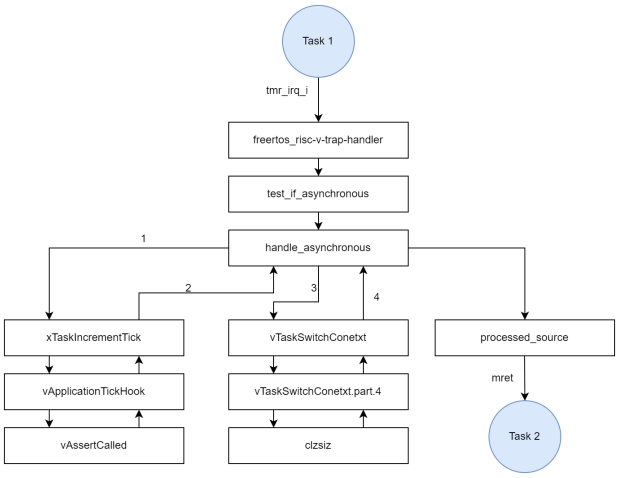

Fig. 2. Function calls during context-switching

function, nothing would happen.

Function $vTaskSwitchContext$ would be called if context-switching is about to be performed. It selects the highest priority task currently in the stack and use $traceTASK\_SWITCHED\_IN$ to link the TCB of the chosen task. The function returns back to $handle\_asynchronous$. $Processed\_source$ is then called, restoring all register files and state registers. The CPU is now in the same state as when the process was last interrupted, a key mret instruction brings the PC back to the scope of the selected task. The task would then execute for another time quantum, waiting for another time interrupt. The function called in a context-switch is showed in Fig.2.

III. Analysis of Context-switching statistics

To measure the context-switching overhead, counters were set up in Aquila. We would like measure two key indicators, the clock cycles it spent and the instructions executed. The counters would starts counting when the CSR sends interrupt signals to Program counter and changes the program counter to the interrupt handler. The counter stops counting when we step into the mret instruction in $Processed\_source$. That is the overhead of a single context switching. Table I shows the overhead of context switching under different time quantum setups.

From Table I, we know when time quantum equals 10ms, the performance reaches its peak. 10 ms is actually the default time quantum set-up. A large time quantum would allow a task to execute till its content. However, if the task is busy waiting for a mutex or entering a critical section, the long time period is a complete waste. On the other hand, a smaller time quantum is also harmful to the system due to the task would keep being interrupted during the execution. 10ms quantum size seems to strike a perfect balance.

A longer time quantum would have less context switching but would in average wait longer for each time when context switches. Nevertheless, the minimum and maximum of context switching cycles are not strongly affected by the time quantum setup. Context switching itself is a piece of code full of memory access, that is the reason why it takes on average 3 clock cycles to execute each instruction. I used to think by setting a smaller time quantum would deteriorate the system performance by introducing loads to context switching. The statistics shows however, in table 1, context switching overhead is not the primary factor of effecting the overall performance. There must be other factors compromising the performance.

IV. Synchronization in RTOS

Under a multi-thread runtime environment, we shall make no assumption of which thread be selected to run. Instead, the execution of threads is controlled by the task scheduler. If the programmer has a specific sequence of execution in mind, synchronization is a way to manipulate the thread's execution. The simplest form of a synchronization tool is a mutex. A mutex is a global object visible to both threads and only one thread could obtain the mutex at once. We could design a system where only the thread with a mutex gets to access the protected resource to protect it from corruption.

A. Critical Section

A critical section is a section of code requires to be executed without being interrupt. FreeRTOS provides $taskENTER\_CRITICAL$ and $taskEXIT\_CRITICAL$ functions to accomplish the goal. FreeRTOS promises it by brutally turning off the task scheduler, allowing the task to execute until it exits the critical section. Although such method is not applicable to multi-core systems, a critical section is the most fundamental way of synchronization in FreeRTOS.

B. Semaphores

FreeRTOS provides series of API to create and operate with a synchronization tool, a semaphore. A semaphore is implemented as a queue, any task holding the Semaphore would be temporarily promoted to a higher priority than all other waiting tasks to prevent priority inversion. A priority inversion is when a lower priority task holding a mutex was preempted, a higher priority task waiting for the mutex has to wait, seemingly become an even lower priority task. Semaphore implementation in FreeRTOS introduces multiple critical sections to protect certain critical section. We may conclude using a Semaphore as a synchronizing tool is more expensive than a critical section in FreeRTOS.

C. Peterson's Algorithm

Peterson's algorithm is a software mutex capable of protecting shared resources among two threads. It works by defining a protocol of obtaining and releasing the mutex as follow: The thread willing to get the mutex first broadcasts it's need, then giving other threads the priority to obtain the mutex and put itself into a spinning lock, waiting for the other thread to release the mutex. The beauty of the

TABLE I
CONTEXT-SWITCHING STATISTICS ABOUT DIFFERENT TIME QUANTUM

Time quantum	Context-switching						Total	
	Counts	INST	Cycles	avg. Cycles	min Cycles	max Cycles	INST	Cycles
1 ms	393	35,269	95,068	242	275	1,664	316,872,411	532,288,175
2 ms	191	16,382	48,085	252	275	1,660	278,292,220	466,883,288
5 ms	78	6,718	21,408	274	325	1,666	299,328,611	502,291,288
10 ms	39	3,467	12,353	317	348	1,676	237,932,972	401,564,123
20 ms	28	2,583	10,104	361	351	1,678	325,110,947	547,191,844
20 ms	28	2,583	10,104	361	351	1,678	325,110,947	547,191,844
100 ms	15	1,316	6,774	452	348	1,656	414,208,705	698,471,311

algorithm is how altruism could help allocate a scare resource between two hungry threads. However, Peterson's algorithm includes a spinning lock, a mechanic harmful for efficiency. The generalized version of Peterson's algorithm is called the Filter algorithm.

D. RISC-V Atomic Instruction

RISC-V support atomic instructions in "A" extension. We could utilize the swap instructions In Atomic Memory Operation (AMO) as cornerstones to build our mutex functions. $amoswap.w.aq$ acquires the mutex while $amoswap.w.rl$ releases the mutex. Both instruction takes 3 parameters, 2 registers (rd, rs2) and an address (rs1). It writes the value in address rs1 into rd then writes rs2's value into rs1 in an atomic action.

The implementation is easy. A global object, an integer, is initialized to 0 as the mutex. The mutex-taking function loads the lock's address into a register and test if it's occupied(equal to 1). If not, it writes 1 into the address with $amoswap.w.aq$. The thread successfully obtains the mutex if the atomic operation is completed. Releasing the lock is just writing zero to the global mutex object using $amoswap.w.rl$ instruction.

V. SYNCHRONIZATION OVERHEAD

Mutex give and take functions shall be carefully labeled with $\#pragma\ no-inline$ to avoid compiler merging their assembly code into task handler functions. Critical sections are used to protect the printing functions so they are referenced less the other two. Both Peterson's algorithm and the RISC-V atomic instruction synchronization algorithm are tested. Table II shows the result.

TABLE II
OVERHEAD OF DIFFERENT SYNCHRONIZATION METHODS

	Critical Section	**Peterson's Algo.**	**Atomic INST.**
Acquire times	14	10,000	10,000
clock cycles	491	250,086	310.274
avg. cycles	35.071	25.009	31.027
max cycles	114	111	176
min cycles	11	25	31
Release times	14	10,000	10,000
clock cycles	722	119.898	210,054
avg. cycles	51.571	11.990	21.005
max cycles	103	40	77
min cycles	10	11	20

The experiment shows that the FreeRTOS critical section is not a very efficient design, which is not what I expected. It takes around 50 cycles to release the mutex. Atomic instruction implementation do protect the mutex well, but it's not doing as well as Peterson's Algorithm. For some strange reason, maybe by my bad design, Peterson's Algorithm seems to take the least cycle of taking and releasing a mutex. It's obvious that our self-written Peterson's mutex get&give functions would outperform the semaphore provided by FreeRTOS, because priority inversion isn't handled in our function. That may cause a problem if two tasks are of different priority.

VI. CONCLUSION AND DISCUSSION

I used to think embedded system could only run a single program like Arduino does. By introducing the operating system layer, embedded systems now have multiple options to multi-task. Although multi-tasking is a feature most modern operating system would offer, it comes with a cost. Context switching overheads are inversely related to the processor performance thus an appropriate time quantum must be set. Multi-thread execution brings synchronization a problem too. Semaphores are OS provided objects to build critical section protection. Using such objects could ensure the correctness of the execution, trading off the usage rate of the CPU. We could also build the synchronization tool by atomic instructions, which is theoretically be the most efficient way to go.