1. 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 recording the task’s run time, including its own stack and register values. Tasks are labeled by four states: run, ready, suspend, and block. Fig0 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 task 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.

2. 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 to represent a timer interrupt. If the timer interrupt is confirmed, CSR would contact the program counter module to jump to the interrupt handler function, freertos\_risc\_v\_trap\_handler( ).

In the trap handling function, RTOS stores the current register file and all other registers, read the interrupt cause from mcause register from the CSR module and test if the interrupt is synchronous. In the handle of asynchronous interrupt, we would have to set up the mtimer for next timer interrupt and jump to xTaskIncrementTick( ). The function increments the timer 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 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 handle of the chosen task to TCB. The function returns back to handle\_asynchrnous( ). 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.

3. 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 1 shows the overhead of context switching under different time quantum setups.

From Table 1, 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, but 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. The chose of 10ms 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 isn’t 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.

Synchronization in RTOS

Under a multithread runtime environment, we shall make no assumption of which thread would be select 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.

1. 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 but brutally turning off the task scheduler, allowing the task to execute until it exits the critical section. Although such method is not applicable to multicore systems, a critical section is the most fundamental way of synchronization in FreeRTOS.

1. Semaphore

FreeRTOS provides series of API to create an operate with a synchronization tool, a semaphore. A semaphore is implemented as a queue, any task holding the Semaphore would be temporarily promote 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 also uses critical sections to protect certain critical section. We may conclude using a Semaphore as a synchronizing tool is much more expensive than by a critical section in FreeRTOS.

1. 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 of the mutex. 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 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.

1. Riscv atomic instructions

RISC-V support atomic instructions in A extension. We could the swap instruction In Atomic Memory Operation (AMO) 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 and writes rs2’s value into rs1 in an atomic action.

The implementation is easy. A global lock object, an integer, is initialized to 0. The mutex-taking function loads the lock’s address into a register and test if it’s occupied(equal to 1). If not, it uses amoswap.w.rl tries to write 1 into the address. If the writing is success, the thread obtains the mutex. Releasing the lock is just writing zero to the global mutex object using amoswap.w.rl instruction.