HW4 RTOS Analysis

郭宗信, 111550105

***Abstract*—This paper presents a comprehensive analysis of thread management and synchronization mechanisms in FreeRTOS v202111.00, focusing on its implementation on the Aquila RISC-V platform. We examine the context-switching overhead through cycle-accurate measurements under different time quantum configurations and analyze two distinct synchronization approaches: semaphore-based (xSemaphoreTake/xSemaphoreGive) and critical section-based (taskENTER_CRITICAL/taskEXIT_CRITICAL) protection mechanisms. The investigation includes measurements of synchronization overhead and cache impact during context switches.**

Keywords—Real-time operating systems, FreeRTOS, thread synchronization, context switching, performance analysis, RISC-V

I. Introduction

The task management mechanism in *FreeRTOS* is similar to thread management in traditional operating systems, where each thread is referred to as a task. Every task has an associated ***Task Control Block (TCB)*** that stores its state information, which is crucial for saving and restoring states during context switches, figure is the brief flow chart of the state phasing.

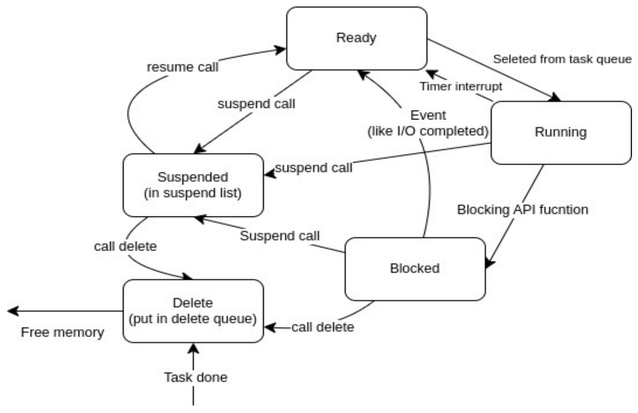


Fig. 1. working status of tasks.

The *TCB* contains several key pieces of information: the task's name, its priority (ranging from 0 to *configMAX_PRIORITIES-1*, with higher values indicating higher priority), the starting address of the task's stack, and two members of type *ListItem_*t, namely *xStateListItem* and *xEventListItem*. The *xStateListItem* is used to indicate the current state of the task, which could be Running (currently executing), Ready (ready to execute), Blocked (waiting for a resource), Suspended (explicitly paused), or Deleted. The main difference between Blocked and Suspended states lies in how they are entered: Blocked is a passive state entered when a task is unable to acquire a resource, while Suspended is an active state entered via API calls. The *xEventListItem*, on the other hand, is used for event management. When config *USE_MUTEXES* is enabled, the *TCB* also stores additional priority information to support priority inheritance for mutexes.

A task enters the Blocked state when it is waiting for a queue, mutex, or after calling *vTaskDelay()*, and this state has a timeout. In contrast, the Suspended state can only be entered or exited through the *vTaskSuspend()* and *vTaskResume() APIs*, typically used to pause tasks that do not need to execute, thus conserving system resources. Separate lists are maintained for Blocked and Suspended tasks.

For task deletion, the process involves placing the task in the list of tasks waiting to be deleted. If a task is marked for deletion, it is cleaned up, and its *TC*B is deleted, including releasing any memory it occupied. A task cannot delete itself completely because a context switch is required for the cleanup process. Instead, it is added to a termination list, and the idle task will later check this list to free up memory allocated by the scheduler for the *TCB* and stack of the deleted task.

Besides, *FreeRTOS* maintains several critical global variables. The *pxCurrentTCB* points to the *TC*B of the currently executing task. The *pxReadyTasksLists* array implements a multi-level queue scheduling mechanism, where each priority level corresponds to a separate list. Delayed tasks are managed using xDelayedTaskList1 and *xDelayedTaskList*2, which handle tick overflow situations. The *xTickCount* variable records the number of tick interrupts, and *xSchedulerRunning* indicates whether the scheduler is active.

When creating a new task, the *xTaskCreate()* function performs three main steps: allocating stack memory, initializing the *TCB*, and adding the task to the ready list. The method for adding a task to the ready list depends on whether the scheduler is running. If the scheduler has not yet started, the *pxCurrentTCB* is updated to point to the highest-priority task or the most recently created task. If the scheduler is already running, the system checks the priority of the new task to decide whether to perform a context switch. This explains why, at the same priority level, the most recently created task is executed first.

II. Context Switching Behavior Analysis

A. Context Switching Algorithm

The context-switching mechanism in *FreeRTOS* is implemented using a hardware timer interrupt. The system sets the *mtimecmp* register in the hardware CLINT module based on the configured time quantum. When the *mtim*e counter in the CLINT module increments to match the value in the *mtimecmp* register, an interrupt signal is sent to the *CSR* module. Upon receiving this signal, the *CSR* module notifies the *Program Counter Unit (PCU*) to jump the PC to the interrupt handler, *freertos_risc_v_trap_handler*. Below is the flow chart of *freertos_risc_v_trap_handler*.

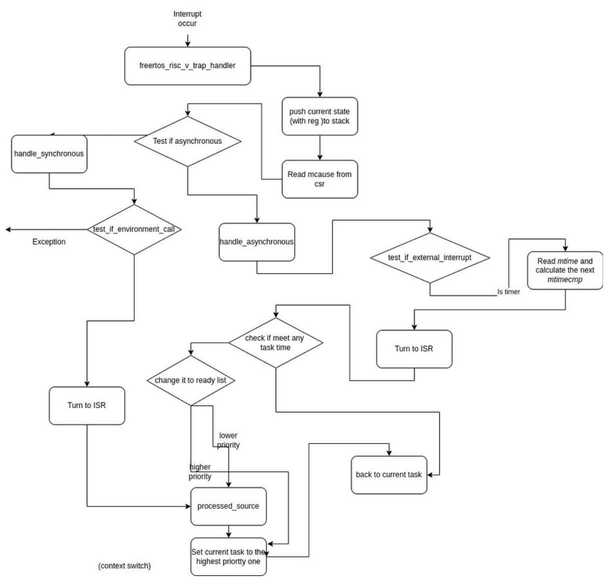


Fig. 2. Flow chart of freertos_risc_v_trap_handler.

Upon entering the *freertos_risc_v_trap_handler*, the system first backs up all the registers of the current task and reads the *mcause* register in the *CSR* module to determine the type of interrupt. According to the *RISC-V* encoding standard, the highest bit of *mcause* indicates the interrupt type: a value of 0 represents a synchronous interrupt, typically triggered by the program itself (e.g., an environment call), while a value of 1 indicates an asynchronous interrupt caused by an external event. Context switching primarily deals with asynchronous interrupts.

After confirming an asynchronous interrupt, the system checks whether it originated from an external device. In the Aquila source code, external interrupt handling is omitted. The system then reads the *mtim*e value and calculates a new *mtimecmp* value to prepare for the next interrupt. Subsequently, it calls the *xTaskIncrementTic*k function to update relevant variables, such as *xTickCount*, and determine if a context switch is necessary. A context switch occurs only if the system is configured in preemptive mode (as defined in *FreeRTOSConfig.h*) and there exists another task with a priority equal to or higher than the current task. Even for tasks with the same priority, switching ensures fair CPU time allocation.

If a context switch is required, the system invokes the *vTaskSwitchContext* function. This function suspends the scheduler, identifies the highest-priority task using t*askSELECT_HIGHEST_PRIORITY_TASK*, and updates the *pxCurrentTCB* pointer to the task's *TCB*. During this process, the system must handle potential tick count overflow. When *xTickCount* increments and wraps back to 0, it triggers an overflow, prompting a call to *taskSWITCH_DELAYED_LISTS* for handling. *FreeRTOS* uses two lists, xDelayedTaskList1 and xDelayedTaskList2, to manage delayed tasks. The pointer *pxDelayedTaskList* refers to xDelayedTaskList1, while *pxOverflowDelayedTaskList* points to xDelayedTaskList2.

Normally, all delayed tasks are stored in *pxDelayedTaskLis*t. When a task calls *vTaskDelay*, the system calculates the task's wake-up time by adding the delay value to the current *xTickCoun*t and stores this result in the task's *TCB's xItemValu*e. The system compares *xTickCount* with *xItemValue* to determine if the task should be woken. When an overflow occurs, new delayed tasks are added to *pxOverflowDelayedTaskLis*t, and *taskSWITCH_DELAYED_LISTS* swaps the two lists to ensure proper task wake-up.

Finally, whether a context switch occurs or not, the system executes the *processed_source* function. This function restores all register values, including general-purpose and status registers, to return the core to its state before the interrupt. It reads the new task's stack pointer and values from *pxCurrentTCB*, writes them to the *mepc* and *mstatus* registers, and then uses the *mret* instruction to resume execution. The *mret* instruction sets the PC to the address of the next task using the values in *mepc* and *mstatu*s. The system also recalculates the next time quantum and configures the corresponding interrupt time, entering an idle state to await the next interrupt.

B. *Context Switching Overhead vs. Time Quantum Size*

Add the *profiler* to *csr_file.v* and pipelined the flush data from the *WriteBack* stage to the csr_file, then use the *ILA* tool in *Vivado* to mark debug signals, seen the PC values from *rtos_run.objdump* as a reference. Begin recording when the PC enters main, and stop after both task 1 and task 2 have completed. The context switch measurement starts at *freertos_risc_v_trap_handle*r and ends at enter processed_source, capturing a full context switch cycle.

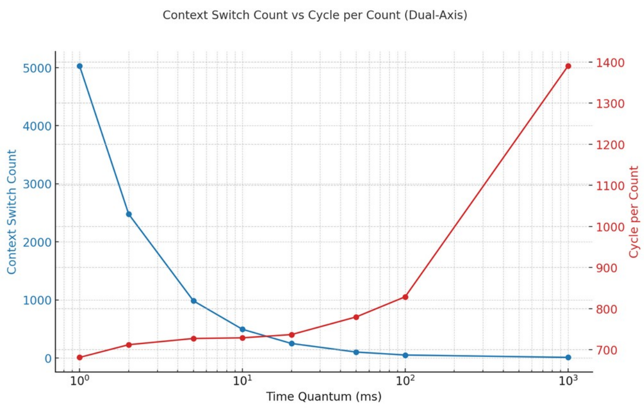


Fig. 3. Context Switching Overhead vs. Time Quantum Size

C. *Impact of Context Switching on Data Cache*

Below is the chart demonstrate the impact on d-cache. This chart, based on **read/write hit/miss data** and **total read/write requests** collected via the HW3 profiler, illustrates the **impact of context switching on data cache performance.** Despite a sample code is too simple leading to hit rates being consistently close to 1, the overall trend still can reveals :Read/Write Hit Rates (green and blue lines): Both hit rates increase as the time quantum grows, approaching 1. This suggests better data cache utilization with fewer context switches.

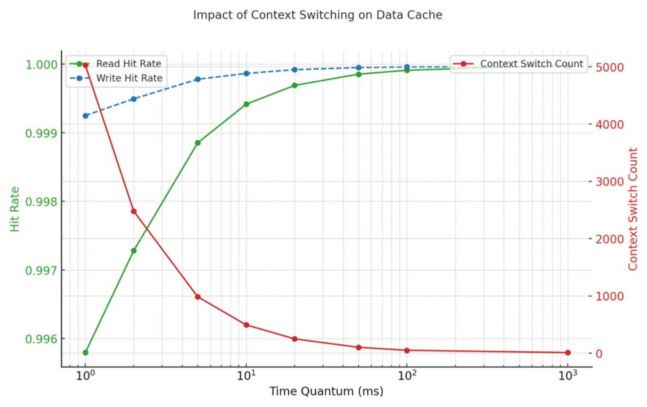


Fig. 4. Impact of Context Switching on Data Cache

III. Synchronization Behavior Analysis

In multitasking systems, proper management of shared resources is crucial. FreeRTOS provides two basic synchronization mechanisms to prevent race conditions: critical sections and mutex (implemented through semaphores). While critical sections are simple but block system scheduling, mutexes offer better priority management despite higher overhead.

A. *Critical Section*

A critical section refers to a short segment of code that cannot be interrupted or preempted. In FreeRTOS, taskENTER_CRITICAL() and taskEXIT_CRITICAL() are provided to suspend the entire task scheduler while executing that portion of the code. This ensures that operations on shared variables within the critical section are not interrupted by other tasks, effectively preventing race conditions. However, this approach also halts the execution of all other tasks, temporarily reducing the system's concurrency.

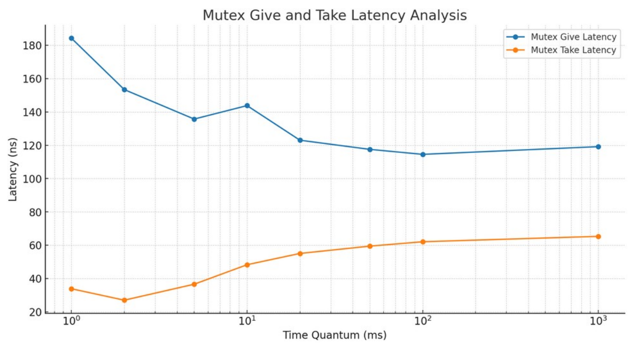
B. *Mutex Implementation and Semaphore*

TA mutex is a synchronization mechanism built upon the concept of semaphores, using the functions xSemaphoreTake() and xSemaphoreGive() to lock and unlock resources. In FreeRTOS, creating a mutex is essentially done by calling xQueueGenericCreate() to instantiate a special queue with a length of one and an item size of zero, followed by prvInitialiseMutex() to initialize it as a structure suitable for synchronizing access to a single resource.

Operationally, when a task needs to access a protected resource, it must first acquire the mutex. If the mutex is already held by another task, the requesting task is blocked until the mutex becomes available. Under the hood, this coordination relies on queue-based mechanisms—such as xQueueGenericReceive() and xQueueGenericSend() to handle task blocking and unblocking, ensuring that resource accesses occur in a safe and orderly manner.

C. *Overhead Analysis of Semaphore and Critical section*

The analysis method uses *Vivado's ILA*. Mutex give/take cycles were measured between entry and return PC values of xQueueSemaphoreTake and xQueueGenericSend functions. Critical section overhead was measured between PC values of vTaskEnterCritical() and vTaskExitCritical().



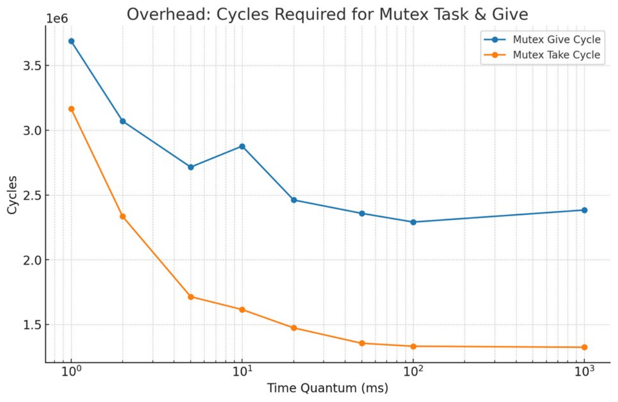


Fig. 5. and 6. cycles and latency required for Mutex Take and Give

TABLE I. Enter and Exit Critical Section Latency

time quantum	*Enter latency*	*Exit latency*
1ms	***18.27***	***17.38***
2ms	***17.16***	***25.67***
5ms	***17.50***	***36.17***
10ms	***19.35***	***44.59***
20ms	***16.99***	***48.24***
50ms	***16.99***	***50.80***
100ms	***16.98***	***51.05***
1000ms	***16.98***	53.75

First chart show that both cycles decrease with increasing time quantum, stabilizing at higher values. This reduction may be due to fewer context switches and better resource availability with larger time slices. Next figure she the give latency decreases as time quantum increases, indicating improved efficiency. However, take latency increases slightly, likely due to contention or delays in accessing shared resources over longer periods. The final table demonstrate that Entering critical sections remains stable, but exiting shows a significant reduction with larger time quanta, implying fewer synchronization overheads as contention decreases.