RTOS Analysis

Zih-Bo Yu(于子博), 111550016

***Abstract* —In this report, we will first explore the task management mechanism in FreeRTOS and analyze statistical data under different scenarios. Then, we will discuss synchronization issues such as mutexes.**

I. Task Management in FreeRTOS

A. *A high level view of tasks*

In this report, the provided rtos_run.c is used to test task management in FreeRTOS. In the initial version of rtos_run.c, two tasks are created, and they share a common shared_counter. Each time one of the tasks executes, it increments both the shared_counter and its own individual counter. Additionally, task1 is responsible for printing the values of all three counters at the end. In the main() function, the two tasks are created using xTaskCreate(), followed by a call to vTaskStartScheduler() to start the execution of these tasks.

In the task's handler function, the actions that the task should perform are defined. In the initial code, the tasks perform very similar operations. Both run a while loop, where at the start of each iteration, their counter is incremented. They then enter a busy computation phase to simulate heavy calculations. The while loop terminates when shared_counter reaches the limit. Since task1 is responsible for printing the final output after execution, it uses vTaskDelay() to wait for task2 to complete its execution.

B. *Tasks related algorithm*

The various task-related functions in FreeRTOS are implemented in task.c. This file contains the core functionality that manages tasks, including their creation, scheduling, and state transitions. Each task is represented by a Task Control Block (TCB), which is a structure that holds all the necessary information about the task, including its state, priority, stack pointer, and other metadata. When a task is created using xTaskCreate, an entry is added to the TCB, and the task's status and associated information are initialized.

In the TCB, when a task is created, the task's state is initially set to "ready," and it is placed into a ready list, awaiting execution by the scheduler. If xTaskStartScheduler is called, the scheduler starts running and determines which task should execute based on priority. Additionally, a special lowest priority IDLE task is created to ensure that if no other tasks are ready to run, the system doesn't remain idle. This IDLE task will execute when there are no other tasks in the ready list.

When a context switch is needed, the status and information stored in the TCB are required to save and restore the task's state. A context switch occurs when the scheduler decides to switch from the current running task to another task. During the context switch, the current task's CPU registers, including the program counter (PC), are saved in its TCB. The TCB provides a mechanism to restore the task's state, including the stack pointer and other relevant data, so that the task can resume from where it left off when it is scheduled again.

In FreeRTOS, tasks can exist in several states, including running, blocked, ready, deleted, and suspended. A running task is currently executing, a ready task is prepared to run but is waiting for the CPU, a blocked task is waiting for an event or timeout, and a deleted task is removed from the scheduler. The suspended state is unique in that a task in this state will not be scheduled for execution and will not respond to events or timeouts. Tasks in the suspended state can only be resumed by explicit calls like vTaskResume. The vTaskSuspendAll function is commonly used in system operations to prevent task switching temporarily. This function suspends the scheduler, which prevents context switches from occurring. It is typically used in critical sections of code where atomicity is required to ensure that the system's state remains consistent during complex operations.

In the hardware module CLINT, the timer interrupt mechanism is triggered by a comparison of mtime and mtimecmp. If mtime exceeds mtimecmp, a timer interrupt is generated. When timer interrupt, the interrupt service routine (ISR) checks the irq_taken flag, and if it is set, the program counter (PC) is redirected to the interrupt handler function. The interrupt handler saves the current state of the CPU, including all registers and the register file. After determining that the interrupt was triggered by the timer, the handler will jump to the function xTaskIncrementTick. This function is responsible for incrementing the global tick count and determining if any tasks should be unblocked due to expired timeouts.

The function xTaskIncrementTick also checks if a context switch is necessary. If configUSE_TIME_SLICING is set to 1, and all tasks have the same priority, a context switch is guaranteed to occur after every timer interrupt. This is because, in this configuration, the tasks will share the CPU time equally, and once a time slice expires, the scheduler will perform a context switch to ensure fair task execution. The system checks if the current task should continue running or if another task with the same priority needs to be scheduled.

Time quantum	1	5	10	50	100
Context switch counts	4123	809	402	81	40
Average overhead	815.3255	840.3498	842.5547	892.1111	894.625
Dcache write hit rate	0.998714	0.999255	0.999340	0.999397	0.999403
Dcache read hit rate	0.996199	0.999069	0.999517	0.999883	0.999924
Context switch write hit rate	97.25%	97.56%	97.52%	97.43%	97.37%
Context switch read hit rate	7.74%	7.5%	7.42%	7.17%	7.02%

When a context switch occurs, the task that was running is saved in its TCB, and the CPU registers, including the program counter, are restored from the TCB of the task being switched to. This process is essential to ensure that tasks can resume their execution from the point where they were last interrupted. The program counter is restored using an instruction like mret, which returns control to the task that is now being scheduled.

II. Analysis of Context-switching

By comparing the information in the table above, we can observe that the time quantum is inversely proportional to the number of context switches. This is because, in this configuration, a timer interrupt directly leads to a context switch. Meanwhile, the average overhead increases as the time quantum rises. I believe this is related to the data cache (dcache). Therefore, I measured the read/write hit rates of the dcache during context switches. It was found that while the overall read/write hit rates improve with an increasing time quantum, the hit rate during context switches actually decreases. Additionally, there are significantly more read operations than write operations during context switches, so the decline in the read hit rate can explain the increase in average overhead.

The increase in time quantum can improve the overall read/write hit rate because, during a context switch, the data in the dcache still belongs to the previous task. Frequent task switching leads to a higher miss rate.

III. Synchronization In FreeRTOS

In FreeRTOS, a mutex (mutual exclusion semaphore) is implemented using a queue with a length of 1. This design choice allows the mutex to represent a binary state: "taken" or "available." The queue serves as a synchronization mechanism without actually storing any data. Specifically, FreeRTOS uses the xQueueGenericCreate function to create this queue. It sets the length to 1, meaning the queue can hold only one item at any given time, and the item size to 0, as no actual data is stored. The purpose of this setup is purely to signal whether the mutex is currently held by a task or available for acquisition.

1) Mutex Acquisition and Blocking

The xSemaphoreTake function is used to acquire the mutex. It works by waiting for the queue to become available. If the queue is empty (indicating the mutex is "taken"), the calling task is blocked and placed in the mutex's waiting queue until either:

1. The mutex is released by the task holding it, allowing the blocked task to proceed, or
2. The specified timeout period expires, in which case xSemaphoreTake returns an error.

2) Mutex Release and Task Unblocking

The xSemaphoreGive function releases the mutex by adding an item back to the queue, marking it as "available." When a mutex is released, FreeRTOS examines the tasks in the waiting queue to identify the highest-priority task. This task is unblocked and given ownership of the mutex. The priority-based task selection ensures that higher-priority tasks waiting for the mutex are not starved by lower-priority tasks, maintaining system responsiveness.

3) Priority Inheritance for Deadlock Prevention

FreeRTOS implements a priority inheritance mechanism as part of the mutex's behavior. If a high-priority task is waiting for a mutex that is held by a lower-priority task, the system temporarily elevates the priority of the lower-priority task to match the waiting task. This ensures that the lower-priority task can complete its critical section as quickly as possible and release the mutex, thereby reducing the risk of priority inversion. Once the mutex is released, the lower-priority task's original priority is restored.

sync_cnt	20004
sync_overhead_cnt	4008531
average_sync_overhead	200.3865

IV. Analysis of Synchronization

In the table above, we can observe that sync_cnt, which represents the number of xSemaphoreTake or xSemaphoreGive calls, is approximately twice the value of COUNTER_LIMIT, which is entirely reasonable. Furthermore, regardless of changes in the time quantum, the statistical data remains relatively consistent. Comparing the cycle count consumed by context switches, we find that synchronization requires significantly more cycles. Therefore, optimizing synchronization-related functions becomes highly important.

V. WHAT I LEARNED

In this assignment, I learned how to systematically trace the functionality of various functions, starting from the high-level main function and delving deeper into the implementation details. This process not only enhanced my understanding of the overall program structure but also provided me with valuable insights into how multi-threading is achieved on a single-core processor. I gained a better grasp of how tasks are scheduled, how critical sections are protected using synchronization mechanisms like mutexes, and how FreeRTOS ensures data consistency in a multi-threaded environment.