

**Analysis of FreeRTOS Source Code: Implementation of the Ready List in the Scheduler**

**GROUP NAME: ALGOHOLICS**

**TEAM MEMBERS:**

|  |  |
| --- | --- |
| **Name** | **Roll Number** |
| 1)Abhisheakraj O | CB.EN.U4CSE21402 |
| 2)Nithin G | CB.EN.U4CSE21443 |
| 3)Amruthiyu S | CB.EN.U4CSE21449 |
| 4)Srutija B | CB.EN.U4CSE21460 |
| 5)T. Bala Bhargavi | CB.EN.U4CSE21462 |

**CONTENT**

|  |  |
| --- | --- |
| **1)Introduction** | **3** |
| **2) Overview of FreeRTOS Scheduler** | **3-4** |
| **3) Understanding the Ready List** | **4** |
| **4) Analyzing the Implementation of the Ready List** | **5-6** |
| **5) Detailed Examination of Ready List Operations** | **6-8** |
| **6) Integration of the Ready List with the Scheduler** | **8-9** |
| **7)Conclusion** | **9-10** |
| **8)References** | **10** |

**Introduction:**

FreeRTOS (Real-Time Operating System) is an open-source, real-time operating system designed for embedded systems. It provides a lightweight and scalable solution for managing tasks, scheduling, inter-task communication, and synchronization in real-time applications. FreeRTOS is widely used in various industries, including automotive, aerospace, medical devices, and consumer electronics, due to its reliability, efficiency, and portability.

The purpose of the report is to analyze and present the implementation of the Ready List in the FreeRTOS scheduler. The Ready List is a critical component of the scheduler that keeps track of all the tasks that are ready to run. It plays a crucial role in determining the task scheduling order and ensuring the timely execution of tasks in a real-time system.

The structure of the report will generally follow the following sections:

1. Introduction: Provides an overview of the report, introducing FreeRTOS and its significance in real-time embedded systems.
2. FreeRTOS Overview: Provides a detailed overview of the FreeRTOS operating system. It also highlights the key components of FreeRTOS relevant to the Ready List implementation.
3. Understanding Ready List: Explores the Ready List data structure, algorithms, and associated functions used in the FreeRTOS scheduler. Discusses how tasks are added, removed, and prioritized in the Ready List.
4. Analyzing Ready List implementation: Discusses the design considerations involved in the Ready List implementation.
5. Detailed Examination of Ready List Operations: Evaluates the performance implications of the Ready List implementation. Discusses the impact on task scheduling, context switching, and system responsiveness.
6. Integration of the Ready List with the Scheduler: how the Ready List interacts with other components of the scheduler and synchronization mechanisms or algorithms used to ensure thread safety and prevent race conditions.
7. Conclusion: Summarizes the findings, highlights the significance of the Ready List implementation, and discusses potential future developments or improvements.

It's important to note that the actual structure and content of the report may vary based on specific requirements, guidelines, or preferences set by the organization or individual requesting the report.

**Overview of FreeRTOS Scheduler:**

The FreeRTOS scheduler is a key component in task management and scheduling within the FreeRTOS real-time operating system. It is responsible for allocating and prioritizing processor time among the tasks (also known as threads or processes) running on the system

The primary goal of the FreeRTOS scheduler is to ensure that each task is given the opportunity to be executed in a fair and timely manner, based on their relative priorities and the availability of system resources. It achieves this by using a preemptive priority-based scheduling algorithm.

When tasks are created in FreeRTOS, they are assigned a priority level that determines their relative importance. The scheduler maintains a data structure known as the Ready List, which holds all the tasks that are ready to execute but are waiting for the processor to become available. The Ready List is organized based on the priority of tasks, with the highest priority tasks being placed at the front of the list.

The importance of the Ready List lies in its role in determining which task should be executed next. When the scheduler is invoked, it examines the Ready List and selects the task with the highest priority that is ready to run. The selected task is then given control of the processor, allowing it to execute its code until it voluntarily yields control or is preempted by a higher-priority task.

If multiple tasks have the same priority, FreeRTOS uses a round-robin scheduling approach within that priority level, ensuring fair execution among tasks of equal importance. This allows for time-slicing, where each task in the same priority level is given an equal share of the processor's time.

The Ready List is dynamic and can be modified by the scheduler as tasks become ready or blocked. For example, when a task completes its execution or enters a blocking state (such as waiting for a resource or an event), it is removed from the Ready List until it becomes ready to run again.

Overall, the FreeRTOS scheduler and the Ready List play a crucial role in managing the execution of tasks in a real-time system. By efficiently prioritizing and allocating processor time, the scheduler ensures that tasks are executed in a timely manner, meeting the requirements of the application and providing a responsive and deterministic system behavior.

**Understanding the Ready List:**

The Ready List is a data structure used by the scheduler in a real-time operating system (RTOS) to manage the states of tasks and facilitate efficient task scheduling. Its purpose is to hold all the tasks that are ready to run, meaning they are eligible for execution but are waiting for the processor to become available.

The Ready List is typically implemented using a suitable data structure that allows for efficient insertion and removal of tasks. The tasks in the Ready List are organized based on their priority levels, with the highest priority tasks placed at the front of the list.

The significance of the Ready List lies in its role in managing the states of tasks efficiently. It allows the scheduler to quickly identify which tasks are ready to run, enabling it to make informed decisions about task scheduling. The Ready List provides the scheduler with a snapshot of the system's current state, indicating which tasks are available for execution.

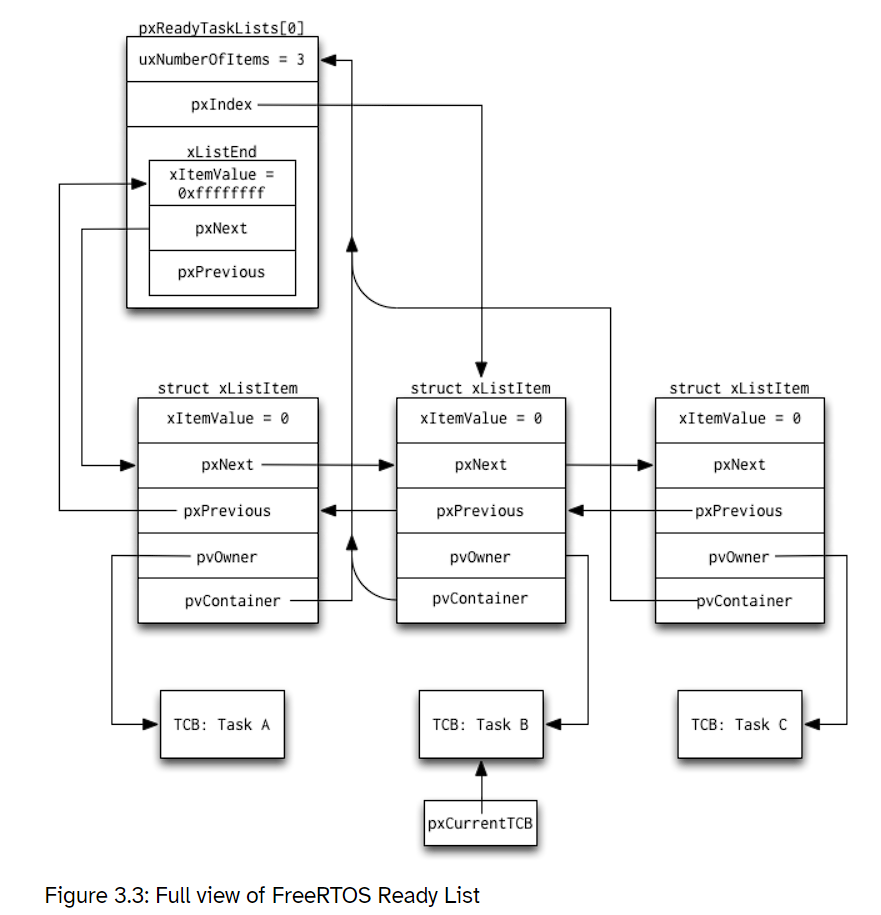
When the scheduler is invoked, it examines the Ready List to determine the highest priority task that is ready to run. This allows the scheduler to allocate the processor's time to the most critical task based on its priority. By organizing tasks based on their priority levels, the Ready List ensures that higher priority tasks are given precedence over lower priority ones, enabling efficient utilization of system resources.

Additionally, the Ready List plays a crucial role in managing the transitions between different task states, such as ready, blocked, and running. This dynamic nature of the Ready List allows the scheduler to efficiently handle task state changes, ensuring that tasks are properly managed and scheduled based on their readiness and priority.

**Analyzing the Implementation of the Ready List:**

The FreeRTOS list is a standard circular doubly linked list with a couple of interesting additions. Each list element holds a number, xItemValue, that is the usually the priority of the task being tracked or a timer value for event scheduling. Lists are kept in high-to-low priority order, meaning that the highest-priority xItemValue (the largest number) is at the front of the list and the lowest priority xItemValue (the smallest number) is at the end of the list.

The pxNext and pxPrevious pointers are standard linked list pointers. pvOwner is a pointer to the owner of the list element. This is usually a pointer to a task's TCB object. pvOwner is used to make task switching fast in vTaskSwitchContext(): once the highest-priority task's list element is found in pxReadyTasksLists[], that list element's pvOwner pointer leads us directly to the TCB needed to schedule the task.



The ready list in FreeRTOS uses a circular doubly linked list data structure for several reasons, including performance and memory usage advantages. Let's explore the rationale behind this choice and its benefits:

Efficient Task Scheduling: The ready list represents the set of tasks that are ready to run. By using a circular doubly linked list, FreeRTOS can efficiently manage and schedule tasks. The circular nature of the list allows for seamless looping through the list, ensuring that no task is left behind or missed during the scheduling process.

Constant Time Insertion and Removal: Circular doubly linked lists provide constant time insertion and removal operations at both ends (head and tail) of the list. When a task becomes ready, it can be quickly inserted at the appropriate position based on its priority without traversing the entire list.

Priority-Based Sorting: The circular doubly linked list allows tasks to be sorted based on their priority. This sorting mechanism ensures that higher priority tasks are scheduled before lower priority ones, improving the responsiveness and real-time capabilities of the system.

Memory Efficiency: Doubly linked lists are memory-efficient compared to other data structures like arrays or singly linked lists. Each task control block (TCB) in the ready list only needs to store the necessary pointers to the previous and next tasks, along with the task's data, this saves memory.

Dynamic Task Priorities: FreeRTOS allows dynamic priority adjustments, where tasks can change their priorities during runtime. The circular doubly linked list structure enables quick repositioning of tasks in the ready list based on their updated priorities.

**Detailed Examination of Ready List Operations:**

**VTaskSwitchContext:**

In FreeRTOS, the vTaskSwitchContext function is a key component of the task scheduling mechanism. It is responsible for performing a context switch between tasks, allowing the scheduler to switch execution from one task to another based on their priorities and scheduling policies. Its primary role is to save the context of the currently running task and restore the context of the next task to be executed. This process involves preserving the task's program counter, stack pointer, and other relevant registers.

**vTaskSwitchContext is implemented in FreeRTOS:**

**Saving the Current Task's Context**: The current task's context, including the program counter, stack pointer, and other relevant registers, is saved. This is usually done by storing the values in the task's TCB or in a dedicated data structure associated with the task.

**Selecting the Next Task**: The next task to be executed is selected based on its priority and scheduling policies.

**Restoring the Next Task's Context**: The saved context of the next task, previously stored in its TCB or context data structure, is restored.

**Performing the Context Switch**: The context switch is performed by updating the processor's registers with the new task's context. This allows the processor to start executing instructions from the newly selected task.

**Task Execution Resumes**: After the context switch, the execution of the new task resumes as if it had been running continuously. The task continues executing until it yields the CPU, enters a blocked state, or gets preempted by a higher-priority task.

In general, the time complexity of the context switch operation itself in FreeRTOS is relatively low and typically considered constant time or O(1). This is achieved by carefully managing task control blocks (TCBs) and using efficient data structures and algorithms to select the next task to run.

**PrvAddTaskToReadyList:**

The function prvAddTaskToReadyList in FreeRTOS is responsible for adding a task to the ready list when it becomes ready to run. The ready list is a data structure that keeps track of all the tasks that are ready to be scheduled and executed by the FreeRTOS kernel.

**Here is an overview of how prvAddTaskToReadyList works in FreeRTOS:**

**Checking the Task Priority**: The function starts by checking the priority of the task being added to the ready list. The priority determines the position of the task within the ready list, with higher-priority tasks typically placed towards the front of the list.

**Updating the Task's State**: If the task is not already in the ready state, the function updates the task's state to indicate that it is now ready to run.

**Inserting the Task into the Ready List**: The function inserts the task into the ready list based on its priority. FreeRTOS uses a priority-based data structure, often a list or array of task control blocks (TCBs), to represent the ready list. The task is inserted in the appropriate position within the list to maintain the order of priorities.

**Updating the Scheduler's Ready List Data**: After adding the task to the ready list, the function updates any relevant data structures or variables used by the scheduler to keep track of the ready tasks.

the time complexity of prvAddTaskToReadyList in FreeRTOS is typically O(1) or constant time, with the caveat that the array implementation of the ready list may have an O(N) insertion time complexity if the number of tasks exceeds the predefined maximum. However, in practice, the number of tasks in FreeRTOS is usually small and bounded, making the time complexity effectively constant.

**PrvRemoveTaskFromReadyList:**

In FreeRTOS, prvRemoveTaskFromReadyList is a function that is responsible for removing a task from the ready list when it is no longer ready to run. The ready list is a data structure that keeps track of all the tasks that are ready to be scheduled and executed by the FreeRTOS kernel.

**Here is an overview of how prvRemoveTaskFromReadyList works in FreeRTOS:**

**Finding the Task in the Ready List**: The function starts by searching for the task to be removed within the ready list. It looks for the task based on its priority or any other identifying criteria.

**Removing the Task from the Ready List**: Once the task is found, the function removes it from the ready list. This involves updating the pointers or reorganizing the data structure to exclude the task.

**Updating the Scheduler's Ready List Data**: After removing the task from the ready list, the function updates any relevant data structures or variables used by the scheduler to reflect the change.

**Optional Task State Update**: Depending on the specific implementation, prvRemoveTaskFromReadyList may also update the state of the removed task.

the time complexity of prvRemoveTaskFromReadyList in FreeRTOS is typically O(N) in the worst case, where N is the number of tasks in the ready list. However, in practice, the number of tasks is usually small and bounded, making the time complexity effectively constant (O(1)).

**UxTopReadyPriority:**

In FreeRTOS, uxTopReadyPriority is a variable that represents the highest priority level among the ready tasks. It is used by the task scheduler to determine the priority of the next task to be scheduled for execution.

The value of uxTopReadyPriority is dynamically updated as tasks become ready or blocked. It reflects the highest priority level among all the tasks that are currently in the ready state. When a task becomes ready, its priority is compared to the value of uxTopReadyPriority, and if it is higher, the value of uxTopReadyPriority is updated accordingly.

**PxCurrentTCB:**

In FreeRTOS, pxCurrentTCB is a pointer to the Task Control Block (TCB) of the currently executing task. The TCB is a data structure that holds information about a task, such as its stack pointer, state, priority, and other relevant task-specific data.

The pxCurrentTCB pointer is typically used by the kernel to access and manipulate the properties of the currently executing task. It allows the kernel to perform various operations on the task, such as updating its state, priority, or other attributes.

**Integration of the Ready List with the Scheduler:**

The Ready List in FreeRTOS interacts with other components of the scheduler, such as context switching and task preemption, to ensure proper task execution and responsiveness in real-time embedded systems.

Context switching is the process of saving the current context (state) of a task and restoring the context of another task. When a task becomes ready to run, it is added to the Ready List. The scheduler selects the highest-priority task from the Ready List and performs a context switch, saving the current task's context and restoring the selected task's context. This allows the selected task to resume execution from where it left off.

Task preemption occurs when a higher-priority task becomes ready to run while a lower-priority task is currently executing. The Ready List plays a crucial role in determining task preemption. If a higher-priority task is added to the Ready List while a lower-priority task is running, the scheduler will preempt the lower-priority task and switch to the higher-priority task, ensuring that the most important tasks receive the necessary resources and execution time.

To ensure thread safety and prevent race conditions in accessing and modifying the Ready List, FreeRTOS incorporates various synchronization mechanisms and algorithms. Some of these include:

Mutexes: Mutexes (mutual exclusion) are used to protect critical sections of code that manipulate the Ready List. They ensure that only one task can access the Ready List at a time, preventing race conditions where multiple tasks may try to modify it simultaneously. Mutexes are typically used when adding or removing tasks from the Ready List.

Semaphores: Semaphores can be employed to synchronize access to the Ready List. They allow tasks to wait until a specific condition is met before accessing the Ready List. For example, a counting semaphore can be used to limit the number of tasks that can access the Ready List concurrently.

Atomic operations: FreeRTOS may utilize atomic operations when performing specific operations on the Ready List. Atomic operations guarantee that a specific operation will be completed without interruption, preventing race conditions. These operations are typically used for critical tasks, such as updating task priorities or manipulating the Ready List pointers.

By employing these synchronization mechanisms and algorithms, FreeRTOS ensures that the Ready List is accessed and modified in a thread-safe manner, preventing data corruption and maintaining the integrity of the scheduler's operation. This allows for reliable multitasking and proper task prioritization in real-time embedded systems.

**CONCLUSION:**

After analyzing the implementation of the Ready List in FreeRTOS, several findings have emerged:

* The Ready List is a fundamental component of the FreeRTOS scheduler, responsible for managing the execution of tasks. It plays a crucial role in determining task priorities, preemption, and context switching.
* The choice of data structure for the Ready List implementation (linked list, array, etc.) has a significant impact on performance and memory usage.
* Algorithms for task insertion, removal, and prioritization in the Ready List vary based on the data structure used. These algorithms are designed to efficiently handle task scheduling and minimize overhead, ensuring deterministic and predictable execution in real-time systems.
* Synchronization mechanisms, such as mutexes, semaphores, and atomic operations, are employed to maintain thread safety and prevent race conditions when accessing and modifying the Ready List. These mechanisms ensure the integrity of the scheduler's operation.

The significance of the Ready List in achieving efficient task scheduling and management cannot be overstated. It enables the scheduler to make informed decisions about task priorities, preemption, and context switching. By maintaining a well-organized Ready List, FreeRTOS can efficiently manage the execution of tasks, ensuring that critical and time-sensitive tasks receive appropriate attention.

In conclusion, the Ready List implementation in FreeRTOS serves as a cornerstone for efficient task scheduling and management in real-time embedded systems. Understanding its design, algorithms, and synchronization mechanisms contributes to building reliable and responsive applications in resource-constrained environments. The insights gained from studying the FreeRTOS source code deepen the understanding of real-time operating systems and provide a foundation for developing high-performance embedded systems.

**REFERENCES:**

<https://aosabook.org/en/v2/freertos.html>

<https://qiriro.com/ecs6264/static_files/references/understanding-freertos.pdf>

<https://stackoverflow.com/>