**DSA REVIEW -1**

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

**TEAM MEMBERS:**

Ankita Venu CB.EN.U4CSE21405

Jayanthi Srivani CB.EN.U4CSE21424

Mattaparti Anjali CB.EN.U4CSE21437

Yella Satya Sai Veer CB.EN.U4CSE21469

Yedla Nrusimha Praneeth CB.EN.U4CSE21470

**Introduction:**

FreeRTOS, which stands for "Free Real-Time Operating System," is an open-source, real-time operating system designed specifically for embedded systems. The significance of FreeRTOS lies in its ability to provide a reliable and deterministic execution environment for real-time applications. FreeRTOS offers a preemptive scheduling algorithm that allows tasks to be prioritized and executed in a deterministic manner, ensuring critical tasks are given priority and completed on time. The purpose of this report is to understand how the ready list is implemented in FreeRTOS. The report is structured into several key sections. It begins with an introduction, where the FreeRTOS scheduler is explained using readylist and its data structure used to implement, its advantages in terms of performance and memory usage, followed by the importance of the Ready List in the scheduler's functionality. It also discusses the operations performed on the Ready List, the time complexity of these operations and their impact on the scheduler's efficiency, interaction with other components of the scheduler, such as context switching and task preemption.

**Overview of FreeRTOS Scheduler:**

The FreeRTOS scheduler is a critical component of the FreeRTOS real-time operating system that manages the execution of tasks in an embedded system. The scheduler's primary role is to determine which task should be executed next from the pool of ready tasks. It operates based on priority-based preemptive scheduling, where tasks with higher priorities are given precedence over lower-priority tasks. The Ready List allows the scheduler to quickly select the highest-priority task for execution by maintaining the ready tasks in a structured manner using the circular doubly linked list as its data structure. It also facilitates the insertion and removal of tasks as they transition between different states. When a task becomes ready, it is inserted based on its priority and when a task is blocked, completes its execution it is removed from the Ready List. The Ready List plays a vital role in meeting real-time requirements of the embedded system, by minimizing response times and ensuring timely task execution.

**Understanding the Ready List:**

A ready list, in the context of task scheduling, is a data structure used by an operating system or scheduler to keep track of tasks that are ready to run. Priority ordering allows the scheduler to quickly identify and select the highest-priority task for execution, ensuring that the most critical tasks are given precedence. Three primary states of ready list are ready state, blocked state and running state. Ready state is when a task becomes ready, such as when it completes its initialization or when it unblocks after waiting for a resource, it is added to the ready list. Blocked state is when a task is no longer ready, such as when it becomes blocked or completes its execution, it is removed from the ready list. The running state is not directly associated with the ready list itself. When a task is selected from the ready list by the scheduler for execution, it transitions from the ready state to the running state. The running state indicates that the task is currently being executed by the CPU. Once a task is in the running state, it continues to execute until it either completes its execution, gets preempted by a higher-priority task, or voluntarily yields the CPU.

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

The ready list is implemented as a circular doubly linked list in FreeRTOS. This means that each node in the list has a pointer to the next node in the list, and the last node in the list has a pointer back to the first node in the list. This allows the list to be traversed in either direction, and it also allows tasks to be quickly added and removed from the list. The choice of a circular doubly linked list for the Ready List was made for several reasons. First, they are very efficient for inserting and deleting nodes. Second, they are very flexible. They can be used to implement a variety of different data structures, including the Ready List. Third, they are very scalable. They can be used to implement a Ready List with many tasks. The ready list is a critical data structure in FreeRTOS. The linked list implementation of the ready list allows the scheduler to access the ready list quickly and efficiently. Performance deals with time complexity and memory wise its terms to the space complexity which are very important for an embedded system. Let us discuss about them.Using a circular doubly linked list for the ready list in a scheduler can provide several advantages:

1. Efficient Insertion and Deletion

2. Bidirectional Traversal

3. Circular Structure

4. Flexibility in Task Ordering

5. Memory Efficiency

The advantages of using a circular doubly linked list for the Ready List in terms of performance and memory usage are as follows:

* Performance: Linked lists are very efficient for inserting and deleting nodes. This is important for the Ready List, as tasks are frequently added and removed from the list as they are scheduled and unscheduled.
* Memory usage: Linked lists are very memory efficient. Each node in a linked list only requires a small amount of memory. This is important for embedded systems, which often have limited memory resources.

**Time and Space Complexity:**

The time and space complexity of various operations on a circular doubly linked list are as follows:

1. Accessing an element:

- Time complexity: O(n)

In the worst case, you may need to traverse the entire list to find the desired element. Thus, the time complexity is linear, where n is the number of elements in the list.

- Space complexity: O(1)

Accessing an element does not require any additional space. It only requires a reference to the node being accessed.

2. Insertion at the beginning/end:

- Time complexity: O(1)

Inserting an element at the beginning/end of a circular doubly linked list can be done in constant time. It involves updating the necessary links without traversing the entire list.

- Space complexity: O(1)

Inserting at the beginning/end does not require any additional space beyond the new node being inserted.

3. Deletion at the beginning/end:

- Time complexity: O(1)

Deleting an element from the beginning/end of a circular doubly linked list can be done in constant time. It involves updating the necessary links without traversing the entire list.

- Space complexity: O(1)

Deleting from the beginning/end does not require any additional space.

4. Traversal/Iteration:

- Time complexity: O(n)

Traversing or iterating over all the elements in a circular doubly linked list requires visiting each node once. Thus, the time complexity is linear, where n is the number of elements in the list.

- Space complexity: O(1)

Traversal or iteration does not require any additional space beyond the references used to store the current node during the process.

**Detailed Examination of Ready List Operations:**

**vTaskSwitchContext():**

The vTaskSwitchContext() function in FreeRTOS is responsible for switching the context of the currently running task to the next highest priority task that is ready to run. The function is called by a number of other FreeRTOS functions, including the vTaskDelay() function, the vTaskResume() function, and the taskYIELD() macro. The vTaskSwitchContext() function has a time complexity of O(1). This is because the function does not need to iterate through any lists or perform any other time-consuming operations. The function simply saves the context of the current task, selects the next highest priority task, and then restores the context of the selected task. The vTaskSwitchContext() function has a significant impact on the efficiency of the FreeRTOS scheduler. The function is called very frequently, so any performance improvements that can be made to the function can have a big impact on the overall performance of the system.

**prvAddTaskToReadyList:**

The prvAddTaskToReadyList function is a macro in the FreeRTOS source code that is used to add a task to the ready list. The function takes a pointer to the task control block (TCB) of the task to be added as its only argument. The function first calls traceMOVED\_TASK\_TO\_READY\_STATE function to record that the task has been moved to the ready state. The function then calls the taskRECORD\_READY\_PRIORITY function to record the task's priority. Finally, the function calls the vListInsertEnd function to insert the task into the ready list at the end of the list for the task's priority. The time complexity of the prvAddTaskToReadyList function is O(1), which means that the function takes a constant amount of time to execute, regardless of the number of tasks in the ready list. This is because the vListInsertEnd function uses a linked list data structure to store the tasks in the ready list as the time complexity of these operations is O(1). The impact of the prvAddTaskToReadyList function on the scheduler's efficiency is that it can slightly increase the time it takes for the scheduler to find the next task to run. This is because the scheduler must search through the entire ready list to find the task with the highest priority. However, the impact of this increase in time is usually very small, and the prvAddTaskToReadyList function is a very efficient way to add a task to the ready list.

**uxListRemove:**

The `uxListRemove` function is a utility function used in FreeRTOS to remove a task control block (TCB) from a list, such as the Ready List. Its primary application is within the `vTaskDeleteOverrun` function, which handles tasks that have exceeded their execution time and need to be deleted from the system. The function takes a pointer to the item to be removed (`ListItem\_t \* const pxItemToRemove`) and updates the neighboring items' pointers in the list to unlink the specified item effectively. The time complexity of the `uxListRemove` function is O(1), indicating that its execution time remains constant regardless of the list's size. This is because the function only involves updating a few pointers, and the operation does not depend on the number of items in the list. Consequently, the function executes quickly, making it an efficient means of removing a task from the Ready List. When a task is deleted or no longer ready to run, it must be promptly removed from the Ready List to prevent unnecessary scheduling delays. By utilizing `uxListRemove` within the `vTaskDeleteOverrun` function, the scheduler can efficiently eliminate overrun tasks from the Ready List. This ensures that the scheduler does not waste time attempting to execute tasks that have exceeded their allotted execution time, thereby improving the system's overall efficiency. Efficient removal of tasks from the Ready List helps maintain a responsive and well-organized task scheduling mechanism, enabling the scheduler to focus on selecting the highest-priority tasks effectively.

**uxTopReadyPriority:**

The uxTopReadyPriority variable in FreeRTOS code is a global variable that stores the highest priority task that is ready to run. The variable is declared in the FreeRTOS header file FreeRTOS.h.The uxTopReadyPriority variable is initialized to the highest possible priority when FreeRTOS is started. The variable is updated whenever a task is added to the ready list or removed from the ready list. The time complexity of accessing the uxTopReadyPriority variable is constant. The variable is a global variable, so it can be accessed directly without any searching or sorting. The impact of the uxTopReadyPriority variable on scheduler efficiency is minimal. The variable is only updated when a task is added to the ready list or removed from the ready list. These events are relatively rare, so the impact on scheduler efficiency is negligible.

**PxCurrentTCB:**

The pxCurrentTCB variable in FreeRTOS code is a global variable that points to the TCB (Task Control Block) of the currently running task. The TCB is a data structure that contains all of the information that the scheduler needs to manage a task, such as the task's priority, state, stack pointer, and so on. The pxCurrentTCB variable is initialized to NULL when FreeRTOS is started. The variable is updated whenever a task is switched to.The pxCurrentTCB variable can be accessed by any function in FreeRTOS. The time complexity of accessing the pxCurrentTCB variable is constant. The variable is a global variable, so it can be accessed directly without any searching or sorting. The impact of the pxCurrentTCB variable on scheduler efficiency is minimal. The variable is only updated when a task is switched to. This event is relatively rare, so the impact on scheduler efficiency is negligible,which improves the overall performance of system.

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

The scheduler interacts with the Ready List in a number of ways, including:

Context switching: Context switching is the process of saving the state of the current task and then switching to another task. This is done when the scheduler decides that it is time to switch to another task. The task control block of the currently running task is saved, and the task control block of the selected task is retrieved from the Ready List to restore its context during the context switch.

Task preemption: Task preemption is the process of forcibly removing a task from the CPU and switching to another task. This is done when a task with a higher priority is ready to run.The Ready List enables task preemption by providing a mechanism to prioritize tasks based on their priority levels.

Synchronization mechanisms and algorithms to ensure thread safety and prevent race conditions are:

Mutexes: Mutexes are synchronization objects that protect shared data by allowing exclusive access to one task at a time. Tasks that acquire a mutex gain exclusive access to the shared data, while other tasks are blocked until the mutex is released.

Semaphores: Semaphores are synchronization objects used to control access to limited resources. Tasks that acquire a semaphore gain permission to access the resource, while other tasks wait until a permit becomes available.

Recursive mutexes: Recursive mutexes allow a task to acquire a mutex multiple times, which can be useful when a task needs to access shared data repeatedly without releasing the mutex each time.

Spin locks: Spin locks are locks that don't block a task if the lock is held by another task. Instead, the task repeatedly checks if the lock becomes available. Spin locks are suitable when tasks need frequent access to shared data for a short duration.

**Performance Evaluation:**

Performance characteristics of the Ready List implementation offer several advantages for efficient task management:

Efficient Task Insertion and Removal: A circular doubly linked list allows for efficient task insertion and removal operations, similar to a regular doubly linked list. The constant time complexity of O(1) ensures quick updates to the Ready List without compromising performance.

Traversal Efficiency: With the circular doubly linked list structure, traversal in both forward and backward directions remains efficient. This enables fast searching for specific tasks or identifying the highest-priority task within the Ready List. Traversal operations can be performed swiftly by navigating through the circular list, facilitating rapid task identification.

Preemptive Scheduling: The circular doubly linked list implementation still supports preemptive scheduling, allowing higher-priority tasks to interrupt and preempt lower-priority tasks. This ensures prompt execution of critical tasks and enhances system responsiveness and prioritization.

Scalability: Similar to the regular doubly linked list implementation, a circular doubly linked list can handle a growing number of tasks while maintaining efficient performance. Operations such as traversal and finding the highest-priority task still have a constant time complexity (O(1)), making it scalable for applications with numerous tasks. The circular structure ensures consistent performance regardless of the list's size.

Despite these advantages, there are potential limitations to be aware of:

Additional Complexity: The circular doubly linked list introduces additional complexity compared to a regular doubly linked list due to the circular nature of the list. Managing wraparound conditions during traversal or operations that involve moving across the boundary of the circular list requires careful handling to ensure correct behavior.

Memory Overhead: Using a circular doubly linked list incurs the same memory overhead as a regular doubly linked list. Each task node still requires space for two pointers. This additional memory usage may impact resource-constrained systems or applications with a large number of tasks. Employing memory optimization techniques may be necessary to mitigate this limitation.

Traversal Overhead: As the number of tasks in the Ready List increases, traversal operations may still impact performance. While the time complexity remains constant, the time spent traversing the circular list can become a limiting factor, especially if frequent traversal operations are required.

Scheduling Policy Limitations: Like the regular doubly linked list implementation, a circular doubly linked list may have limitations when it comes to supporting complex scheduling policies beyond priority-based scheduling. Additional customization or modifications may be required to accommodate policies such as round-robin or deadline-based scheduling.

**Conclusion:**

In summary, the Ready List is vital for achieving efficient task scheduling and management. It enables prioritization of tasks, efficient resource allocation, implementation of scheduling algorithms, quick task selection, and dynamic task management. By effectively managing the Ready List, the scheduler can optimize system performance, responsiveness, and resource utilization, ensuring the efficient execution of tasks in an operating system environment. To conclude, it is been implemented by circular doubly linked list which provides us with a better performance and memory. As we say the best time complexity plays a major role in the efficiency of the data structure. Circular doubly linked list provides us with the time complexity of O(1) for both insertion and deletion and hence chosen as an appropriate data structure for the Ready list implementation.

**References:**

<https://www.freertos.org/FreeRTOS_Support_Forum_Archive/February_2015/freertos_FreeRTOS_List_01873575j.html>

<https://www.codingninjas.com/codestudio/library/freertos>

<https://www.sanfoundry.com/c-program-circular-doubly-linked-list/#:~:text=Time%20Complexity%3A%20O(n),the%20size%20of%20the%20list>.

<https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure>