**ANALYSIS OF FREERTOS SOURCE CODE:**

**IMPLEMENTATION OF THE READY LIST IN THE SCHEDULER**

**TEAM MEMBERS:**

|  |  |
| --- | --- |
| **Name** | **Roll number** |
| Akkina Aatesh | CB.EN.U4CSE21404 |
| B V V Satyanarayana | CB.EN.U4CSE21409 |
| Bhavana N | CB.EN.U4CSE21411 |
| Nadimpalli Ujwal Srimanth Varma | CB.EN.U4CSE21440 |
| Potu Tejaswi | CB.EN.U4CSE21445 |

**1. Introduction: |**

The popular open-source operating system FreeRTOS (Real-Time Operating System) was created especially for real-time embedded devices. It gives these systems a simple and effective way to manage tasks, schedules, and resource allocation. Numerous sectors, including automotive, aerospace, medical technology, and industrial automation, employ FreeRTOS because of its many capabilities.

The report's goal is to investigate how the Ready List is implemented in the FreeRTOS scheduler. The scheduler uses the Ready List, an essential data structure, to keep track of tasks that are prepared to run but are awaiting the processor's execution. Gaining insight into the FreeRTOS task scheduling process requires understanding the Ready List's implementation specifics.

The report's primary emphasis will be on the Ready List's implementation. It will go through the list's data structure and the techniques used for task addition, deletion, and prioritization. The report will also go through any trade-offs or optimizations made during the Ready List implementation, explaining the rationale for each decision.

**2. Overview of FreeRTOS Scheduler:                                                                                               |**

**TASK MANAGEMENT AND SCHEDULING**

The FreeRTOS scheduler is a fundamental component of task management in the FreeRTOS. Its primary role is to manage and schedule the execution of tasks in an efficient and deterministic manner. The following components are some of the works in task management where a scheduler works:

1. Task Creation

2. Task Initialization

3. Task Activation

4. Task Execution

5. Preemption

6. Context Switching

7. Task Synchronization and Communication

8.Task Suspension and Resumption

9. Idle Task

FreeRTOS scheduler is priority-based preemptive scheduling, where higher-priority tasks always take precedence over lower-priority tasks.

**IMPORTANCE OF READY LIST**

FreeRTOS uses a "ready list" to keep track of all tasks that are currently ready to run. The use of a Ready List as the data structure for managing tasks in the Ready state is a design choice made in FreeRTOS due to its efficiency and simplicity. While other data structures could potentially be used for similar purposes, the Ready List offers several advantages: such as:

1.  **Efficiency:** The Ready List provides efficient operations for task insertion, removal, and retrieval. The Ready List allows for fast access and manipulation of tasks in the Ready state as maintains tasks in a sorted order based on priority

2.  **Priority Ordering:** The Ready List allows tasks to be ordered based on priority-based preemption and selects the highest-priority task for execution. ensuring that critical tasks receive immediate attention.

3.  **Simplicity:** The use of a Ready List provides a simple approach to task management. The simplicity of the Ready List contributes to the overall efficiency and maintainability of the FreeRTOS scheduler.

4.  **Less consumption of memory**: Real-time embedded systems often have limited memory resources, and using a lightweight data structure like the Ready List helps optimize memory usage.

**3. Understanding the Ready List: |**

The concept of the Ready List and its purpose in task scheduling.

It is a data structure that helps in efficient task scheduling in any OS not only in freeRTOS.It maintains an organized collection of ready tasks which helps the scheduler identify which task is going to be executed. In freeRTOS, the scheduler selects the highest priority tasks for execution first. All the tasks in the ready list are in waiting mode. Waiting means all the processes have fulfilled all the prerequisites and are ready to be executed by the CPU/Processor.

The ready list allows fast insertion and removal of tasks as their states change(we need to select a data structure where insertion and deletion take minimum time). When a task become ready or blocked or preempted then it can easily be added or taken back from the ready list. The smoothness or this agility ensures that the scheduler can respond promptly to changes in task availability and make optimal scheduling decisions. Efficiency is very important for task scheduling in OS as it allows OS to make the best use of its resources. The ready list plays a major role in achieving this efficiency.

The significance of the Ready List in managing tasks' states (i.e., ready, blocked, running) efficiently.

So ready list stores all the **waiting/ready state** tasks based on their priority and it makes the scheduler’s job easier to select which process to be executed next.

So if a Task is **blocked** for a particular reason due to some I/O wait after its waiting is completed it goes into a ready state and is added back into the ready list.

**Running** tasks are the Tasks that are executed in the CPU, when these tasks are preempted for some higher priority task then they are added back to the ready list again. So the ready list acts as a repository for these kinds of tasks until they start their execution again. By providing a mechanism for quick insertion, removal, and prioritization of tasks, the Ready List contributes to the overall efficiency and responsiveness of the task scheduling process in real-time operating systems like FreeRTOS.

4.  **Analyzing the Implementation of the Ready List: |**

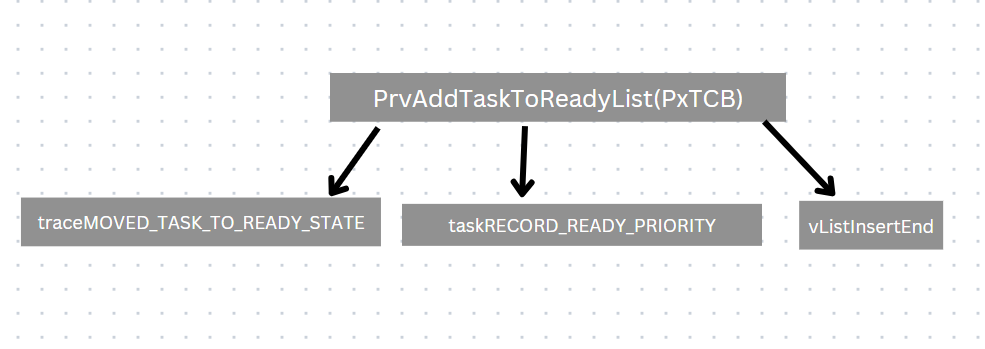
The ready list is implemented using an array of circular doubly linked lists. In other words, there exists an independent circular doubly linked list at every index of the array. Generally, a circular doubly linked list is a combination of both doubly linked lists and circular lists. The worst-case time complexity is O(n) and Auxiliary space O(1) since constant extra space is used in implementing this data structure. Inserting in a basic array with its index known has a time complexity O(1). Deletion operations have a time complexity of O(1) in the case of doubly linked lists.

The added advantage of choosing this data structure is that in a doubly linked list, traversal happens in both directions, and since it’s circular, jumping from head to tail and vice versa is done in a constant time O(1). Due to these properties, circular doubly linked lists are used to implement the Fibonacci heap, in media players, etc.

5. **Detailed Examination of Ready List Operations: |**

* **configMAX\_PRIORITIES-**This is a variable that helps us to know about the range of priority values assigned to a Task. This variable is assigned to a value of 10 in the library freeRTOSconfig.h. So a task can have a minimum priority of 0 and a maximum priority configMAX\_PRIORITIES-1 which is 9.
* Here a list or an array kind of thing PxReadyTasksLists[configMAX\_PRIORITIES] is defined. So it is defined as **list\_t PxReadyTasksLists[configMAX\_PRIORITIES].** So here list\_t refers to a structure variable for the structure xLIST. It has the structure of members.
* **uxNumberOfItems -** This member represents the current number of items in the list. It stores the count of items present in the list at any given time.
* **pxIndex:** This member is used as an index to walk through the list. It points to the last item returned by a call to the listGET\_OWNER\_OF\_NEXT\_ENTRY()[this gives the task which executes next in the CPU] macro, which is typically used to iterate over the items in the list.
* **xListEnd:** This member represents a list item that acts as a marker indicating the end of the list. It contains the maximum possible item value, ensuring that it always appears at the end of the list. The xListEnd item is used to determine when the list traversal reaches the end.
* **PxReadyTasksLists[10]:** The array index corresponds to the priority of the task.
* **UxPriority-** It is the priority assigned to a task.
* **UxTopReadyPriority-** highest priority among ready tasks.
* **UxCurrentBasePriority-**Priority when the task was created
* **UxPriorityUsedOnEntry-**Priority when the task enters a critical region
* **UxNewPriority-**So sometimes when a higher priority task is waiting for a lower priority task then the lower priority task’s priority is temporarily increased (Also known as priority inheritance)
* **pxTCB**-is set to the address of the owner of the next list item.
* **pxCurrentTCB**- it is set to the address of the currently executing task
* The list is the linked list of all tasks with the same priority. A list Item is an element of a list. These two words will be used a lot in the explanation of the functions.
* A **TCB(**Task control block) stores all the information related to a task in it. It is a structure defined with the name tskTaskControlBlock. The structure variable is tskTCB and it is typdefed as TCB\_t.

**PrvAddTaskToReadyList(PxTCB)**



It is calling 3 different functions:

1. traceMOVED\_TASK\_TO\_READY\_STATE- parameter is PxTCB(which is a pointer to a list item). It traces if the task has been moved from a blocked/suspended state to a ready state so that according to its priority it can be added to the linked list.
2. taskRECORD\_READY\_PRIORITY - so parameter is PxTCB->UxPriority. This function assigns the UxTopReadyPriority. If the entering task has a priority greater than the present UxTopReadyPriority then UxTopReadyPrioirty is assigned to the value of the priority of the currently entering task.
3. vListInsertEnd( List\_t \* const pxList, ListItem\_t \* const pxNewListItem ) - This function is used to add the Tasks TCB or the newListItem to the ready list. The 2 parameters are the **newListItem** to be added and in which linked list it needs to be added. In this function, we take a pxIndex value. The new Item is inserted at the last of that Linked list as this task will be selected at the last by listGET\_OWNER\_OF\_NEXT\_ENTRY(). The number of values in the pxList is updated.

The time complexity of this function would mainly depend on the complexity of the operations performed within the function, such as logging or tracing mechanisms, data storage, and any additional processing that might be involved. If the function performs simple and constant-time operations, such as updating a data structure or storing information, the time complexity would be considered O(1) or constant time.

However, if the function involves more complex operations, such as searching or manipulating data structures, the time complexity could be higher. For example, if the function needs to search through a large data structure to find the appropriate location to log the task transition, the time complexity could be O(n), where n represents the size of the data structure.

* taskRECORD\_READY\_PRIORITY - It is made of trivial operations (if and assignment statements) that takes O(1).
* vListInsertEnd- it just adds an element at the last of the linked list where a pointer to the last of the list is already given. Thus, takes O(1) time.
* Hence, it depends on taskRECORD\_READY\_PRIORITY. If it has complex operations then it is O(n)+O(1)+O(1)=O(N)
* If it has trivial operations then it is O(1)+O(1)+O(1)=O(1)
* The implementation of taskRECORD\_READY\_PRIORITY is not there in the code it is just defined as a macro so we cannot judge the actual time complexity.
* But there is a small macro defined which just does a trivial operation left shift so it can be considered as O(1)

**vTaskSwitchContext**

This Function is already defined in os\_task.h. The body of the function is written in os\_tasks.c.

Context switching happens, if there is a change in the process which is being executed. This function first checks whether the scheduler is suspended, and then decides on performing context switching.

In the else part a function traceTASK\_SWITCHED\_OUT() Function is called it says that the current process which is executing has been switched out.

If configGENERATE\_RUN\_TIME\_STATS==1 or it is enabled then the runtime statistics are collected and stored in the ulRuntimeCounter variable(for how much time the process is in running state is being calculated here)

The formula for calculating runtime is

**ulRuntimeCounter+= ulTotalRunTime - ulTaskSwitchedInTime**

So where ulTaskSwitchedInTime is given marking the start of the process. The ulTotalRuntTime is the total run time of the code. Subtracting both of them gives the total running time of the task.

After this calculation stack overflow conditions are checked for security reasons and the

Function taskSELECT\_HIGHEST\_PRIORITY\_TASK() is called which selects the highest priority task from the linked list and sends it for execution.

**taskSELECT\_HIGHEST\_PRIORITY\_TASK()**.

1. The function iterates through the priority queues in descending order until it finds a non-empty queue. This is done using the listLIST\_IS\_EMPTY() macro, which checks if a given list is empty.

2. If the current priority queue is empty, the uxTopReadyPriority variable is decremented to move to the next lower priority queue.

3. Once a non-empty queue is found, the listGET\_OWNER\_OF\_NEXT\_ENTRY() macro is used to select the next task from the list. This macro typically returns a pointer to the control block (pxCurrentTCB) of the selected task.

4. The selected task is considered the highest priority task and can be scheduled for execution.

Example: Let's consider a simplified example where there are three priority queues (0, 1, and 2), and the variable uxTopReadyPriority is initially set to 2. Suppose priority queue 1 is empty, and priority queues 2 and 0 have ready tasks.

1. The function starts iterating from priority 2. Since queue 2 has ready tasks, the loop is exited.

2. The listGET\_OWNER\_OF\_NEXT\_ENTRY() macro selects the next task from priority queue 2.

3. The selected task, let's call it TaskX, is considered the highest priority task and can be scheduled for execution.

traceTASK\_SWITCHED\_IN() is called to say that the context switch happened.

Time complexity is O(1)+O(1)+O(1)+O(n)+O(1) is the time complexity so it is O(n)

That O(n) is to find out the highest priority task in the ready list. The time complexity of taskSELECT\_HIGHEST\_PRIORITY\_TASK() mainly depends on the number of tasks in each index value(priority). In the worst-case scenario, where all ready lists(which means there are no ready tasks in all the priority values) are empty, the time complexity is O(N), where N is the number of priority values (for example if there are priorities ranging from 0 to 8 then the number of priority values are 9).

 O(1)+O(1)+O(1)+O(n)+O(1) [first O(1) is to check whether the scheduler is suspended or not, the second O(1) is to calculate the run time, third O(1) is to check stack overflow, O(n) is already mentioned and last O(1) is for Switch Newlib's \_impure\_ptr variable to point to the \_reent structure specific to this task. ]

**3rd function:-  uxListRemove**

The function is defined in the os\_list.c it has the parameter pxItemToRemove as a listitem, an element in the linked list of the same priority. The address of the linked lists is stored in the pxList. The pxItemToRemove->pvContainer returns the address of the container in which the linked list is stored. pxItemToRemove is an item or node in the pxList. As pxItemToRemove is an address so directly to delete no need to traverse the list due to which it just takes O(1) time.

After removing that node, observing the node is still associated with the list, thus disconnecting it with an assignment operation, pxItemToRemove->pvContainer = NULL, and then decrement the number of items in pxList.

So the time complexity is O(1).

**pxCurrentTCB**- it is a pointer that points to the TCB of the currently running tasks. So if it is NULL then there is no task currently executing which means if a task comes then it can be directly executed. When no higher Priority tasks are ready to run the pxCurrentTCB points to the TCB of the idle task.

This idle task has the least priority and runs only at the time when there are no processes ready to execute or the processor is idle. It prevents the processor from executing random instructions or entering an infinite loop and makes the processor remain responsive. This variable is mainly updated or assigned in the function named xTaskGenericCreate() where pxcurrentTCB is updated during preemption or context switching.

**xTaskGenericCreate()**is a function that is used to create tasks in freeRTOS.

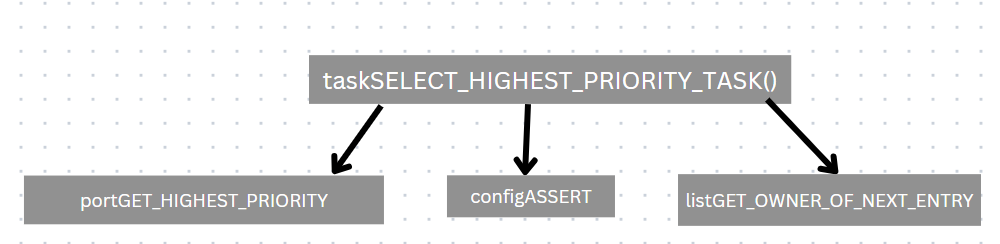
Creating task, in detail:

1. Allocating memory: It allocates memory for the task's control block (TCB) and stack space, based on the specified stack size.
2. Initializing the task's control block: It sets up the necessary data structures and initializes the task's control block, including its state, priority, stack pointer, and other relevant information.
3. Creating the task's stack: It sets up the initial stack frame for the task, including any necessary context information, such as the task function's entry point and the task parameters.
4. Adding the task to the task list: It adds the newly created task to the system's task list, making it eligible for scheduling and execution.
5. uxTopReadyPriority-It stores the highest priority among ready tasks. It is an unsigned integer.

It is changed in the function TaskRecordReadyPriority when a new task is added to the ready list and has a higher priority than the value of uxTopReadyPriority. This variable is also used to find the highest priority linked list which contains ready tasks. So this variable is used as an index to access that linked list. This variable can only be accessed in privileged mode. It is a volatile variable that may unexpectedly change outside the control of the code

**4th Function:-**

**taskSELECT\_HIGHEST\_PRIORITY\_TASK()**



It has function calls and none of these functions call another function.

1. **configAssert():** configASSERT( listCURRENT\_LIST\_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) ) > 0 ); verifies that the length of the priority list for the top priority (uxTopPriority) of ready tasks is greater than zero. If the condition is false (the list is empty), the program execution will be halted or an appropriate error handling mechanism will be triggered, depending on the configASSERT implementation.
2. **portGET\_HIGHEST-PRIORITY:** In FreeRTOS, this function is used to determine the highest priority task in the system. It calculates the highest priority by analyzing a bitmap of ready priorities.
3. **listGET\_OWNER\_OF\_NEXT\_ENTRY():** This first line creates a constant pointer pxConstList that points to the input pxList. The purpose of this constant pointer is to ensure that the pxList is not modified within the macro function. The next line updates the pxIndex pointer of the list, moving it to the next item in the list. It points to the list item that follows the current item. If the if condition is true, it means the index pointer has reached the end of the list. In that case, the following action is performed:

( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;

updates the index pointer to the next item in the list. It effectively wraps around the beginning of the list since the end marker is not considered a valid list item.

This step ensures that the index pointer always points to a valid list item and allows for continuous traversal of the list, even when the end of the list is reached. It creates a circular behavior where the index pointer moves back to the beginning of the list after reaching the end marker. This function is for updating the pxTCB value to the owner of the next item which is going to be executed.

Summary of the entire function

So the function taskSELECT\_HIGHEST\_PRIORITY\_TASK() first gets the highest priority among all the tasks using the first function call and checks whether there are actually any ready tasks available of the priority returned by the first function and then points the pxTCB to the highest priority ready task available which is nothing but the next task to be executed.

**5th Function:-  taskRESET\_READY\_PRIORITY**

The taskRESET\_READY\_PRIORITY function is used to reset the ready priority of a task to its original base priority. It is typically called when a task transitions from the running state to the ready state or when a task's priority is changed dynamically during runtime. Functions:

1. Determines the task's base priority: The function first determines the base priority of the task. The base priority is the priority assigned to the task during its creation or initialization.
2. Resets the ready priority: It sets the ready priority of the task to its base priority. This means that the task's priority is reverted to its original value, disregarding any temporary priority changes that may have occurred.
3. Updates the task's position in the ready list: The function ensures that the task is correctly positioned in the ready list based on its new ready priority. This involves removing the task from its current position in the list and inserting it at the appropriate position according to its updated priority.
4. Allows the scheduler to consider the task for execution: By resetting the ready priority, the task becomes eligible for scheduling based on its updated priority. The scheduler can then select the most suitable task for execution based on the priority-based scheduling algorithm. It takes a single parameter, uxPriority, which represents the priority of the task being reset. The macro checks if the ready list corresponding to the given priority is empty. It does this by using the listCURRENT\_LIST\_LENGTH macro, which returns the number of items currently in a list. If the ready list is empty (i.e., its length is zero), it calls a port-specific macro (e.g., portRESET\_READY\_PRIORITY) to update the ready priority information. This port-specific macro adjusts the uxTopReadyPriority if necessary based on the empty list.
5. **Integration of the Ready List with the Scheduler: |**

FreeRtos uses HigherPriorityTaskWoken() function to perform the context-switching operation. It ensures that the scheduler returns to the highest priority task after the interrupt is dealt with. It is always set to pdFALSE before passing into any function, to ensure that the context switch and task pre-emption operations are performed correctly. Overall, the ready list acts as a central data structure that allows the scheduler to efficiently manage task scheduling, prioritize tasks, handle preemption, and determine the next task to run based on their priority levels.

Binary semaphores do not have a priority inheritance mechanism which makes them the better choice for implementing synchronization. Task synchronization is implemented by having one task or interrupt give and another task take the semaphore. It is referenced using the variable SemaphoreHandle\_t type. A function **xEventGroupSync()** is used to synchronize multiple tasks where each process has to wait for other processes to reach a synchronization point before proceeding further.

1. **Conclusion: |**

In conclusion, The Ready list is used to store tasks that are ready to run, and it is used by the scheduler to determine which task should run next. The Ready List is implemented as a linked list of task control blocks (TCBs). Each TCB contains information about the task, such as its priority, its state, and its stack pointer.

The Ready List is significant for two reasons.

1)It allows the scheduler to quickly and efficiently find the highest-priority task that is ready to run. This is important because it ensures that the highest-priority tasks are always allowed to run, even if there are a large number of tasks in the Ready List.

2)the Ready List allows the scheduler to maintain a consistent view of the state of all tasks in the system. This is important because it allows the scheduler to make informed decisions about which task should run next, and it allows the scheduler to detect and handle deadlocks and other problems.

3) The ready list can be manipulated by the task itself, or by other tasks. For example, a task can use the vTaskSuspend() function to remove itself from the ready list, or the vTaskResume() function to add itself back to the ready list. Other tasks can also manipulate the ready list using the vTaskDelete() function to delete a task from the ready list, or the vTaskSuspendAll() function to suspend all tasks in the ready list.

Limitations include Memory overhead (which might arise when the number of tasks potentially increases: memory allocation should be taken care of) and the priority inversion issue (the ready list in FreeRtos doesn’t address this directly; Task lists is used instead.)

The implementation of the Ready List in FreeRTOS is well-designed and efficient. The use of a linked list allows for easy operations (insertion, removal) of tasks from the Ready List.

FreeRTOS is a very well-written and well-documented piece of software. It is a very flexible and extensible piece of software. It can be used to implement a wide variety of real-time systems, from simple embedded systems to complex industrial control systems.

1. **References:- |**

* https://www.freertos.org/
* <https://stackoverflow.com/questions/60002263/how-to-simulate-stack-overflow-on-freertos>