Real-Time Task Scheduling and Management in FreeRTOS: An Analysis of the Ready List

Team Members

Jagadeeshwar	CB.EN.U4CSE21423
Manoj Kumaran S	CB.EN.U4CSE21436
Sanjay Balamurugan	CB.EN.U4CSE21453
Sri Nandhini R	CB.EN.U4CSE21459

AMRITA VISHWA VIDYAPEETHAM

DEEMED TO BE UNIVERSITY

Introduction:

It becomes more challenging to guarantee that embedded systems meet real-time performance requirements as they get more complicated. Implementing kernel objects and RTOS functionality into hardware using high-level synthesis is one method for reducing response time. This work adapts a previous technique for doing this to the well-liked RTOS FreeRTOS.

Tasks can be established in FreeRTOS either statically or dynamically. Linked lists are used to maintain the control data for tasks. The authors limit task creation until just before the scheduler starts in order to increase performance. This makes it possible to store the task control data in an array. Software timers are used as independent timer-equipped processes. The authors also offer approaches for building a data queue for asynchronous data exchange and dispatch disabling for mutual exclusion. A condensed version of a sample program included with FreeRTOS was used by the authors to construct a hardware module. Task control and data queue operations could be completed by the module in under 300 ns and 700 ns, respectively. The outcomes of this work demonstrate that high-level synthesis can be used to implement RTOS functions in hardware. This can speed up RTOS-based systems' response times, which is beneficial for many applications.

Objective of the Report:

- To Discuss:
    1. Introduction to the FreeRTOS Scheduler
    2. Recognizing the Ready List
    3. Examining How the Ready List Was Implemented
    4. Extensive Review of Ready List Operations
    5. Integration of the Ready List with the Scheduler
- Conclusion
- References

Introduction to the FreeRTOS Scheduler

Within the FreeRTOS real-time operating system, the FreeRTOS scheduler is essential for task management and scheduling. Its main job is to assign CPU time to various tasks or threads in accordance with their priorities and predetermined scheduling policies.

The scheduler keeps a ready list of all the tasks that are prepared for execution. The scheduling algorithm it uses is preemptive priority-based, meaning that tasks with higher priorities take precedence over those with lower priorities. The highest-priority task is chosen from the ready list whenever the scheduler is called, and the CPU is then assigned to that task in order to execute it.

When a task with a higher priority is ready or when a running task voluntarily releases the CPU, the scheduler also manages task switching. It ensures a smooth transition between tasks by saving the context of the running task and restoring the context of the chosen task to be executed. Overall, by managing task priorities and scheduling them effectively, the FreeRTOS scheduler ensures fair and efficient CPU utilisation, enabling real-time systems to meet their timing requirements.

the importance of the Ready List in the scheduler's functionality

An essential part of a scheduler's functionality is the Ready List. All the tasks that are prepared to be executed are stored in it, acting as a central data structure. Based on the tasks' priorities, the scheduler can quickly decide which task should have CPU control using the Ready List. By offering a ready pool of tasks, it facilitates effective task selection by removing the need to comb through the entire task set in search of the following eligible task. The Ready List also ensures fairness and prioritization, enabling the prompt completion of tasks with higher priorities. The Ready List improves the responsiveness and real-time capabilities of the scheduler in managing task execution by keeping a structured and accessible list of ready tasks.

II. Recognizing the Ready List

The scheduler in FreeRTOS uses a data structure called the Ready List to control task scheduling. All of the tasks in the system that are prepared to be executed are contained in it, serving as a list or queue. Each task on the Ready List has a priority value assigned to it.

The Ready List's function is to make efficient task selection based on priorities possible. The scheduler searches the Ready List for the task with the highest priority when deciding which task should be carried out next. This enables the scheduler to quickly determine which task deserves to be given CPU priority.

Fairness and task prioritisation are ensured by the Ready List. The Ready List is organised so that tasks with a higher priority are at the top, where they can immediately access the CPU when they are prepared to run. When there are no higher-priority tasks left, lower-priority tasks are moved to the bottom of the list and only receive CPU time then.

The Ready List in FreeRTOS makes task scheduling quick and easy by keeping a list of prepared tasks that is well-organized and using a priority-based selection algorithm.

The importance of the Ready List in the scheduler's functionality

A crucial part of FreeRTOS is the Ready List, which keeps track of all tasks that are prepared to be executed. Depending on the task's priority, the scheduler uses the Ready List to decide which task needs to run next. The states of tasks, such as whether they are blocked or in progress, are also managed by the Ready List. FreeRTOS can make sure that tasks are scheduled fairly and effectively by effectively managing the Ready List.

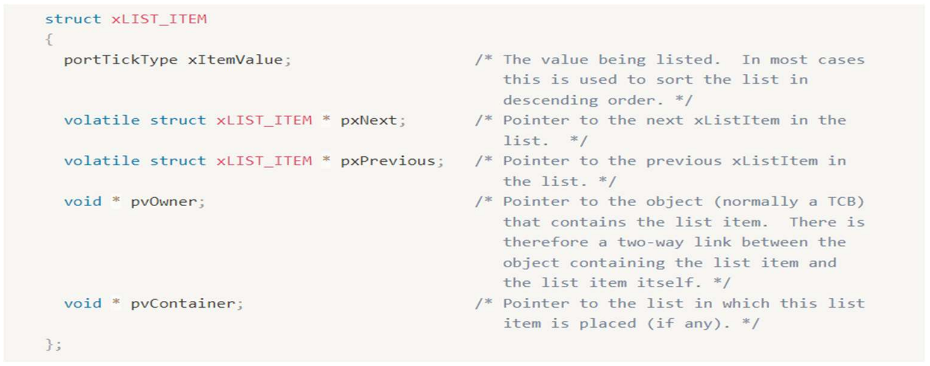
Advantages of using a Ready List:

- Efficiency: The Ready List can be quickly searched to find the next task to run, which can improve the overall performance of the system.
- Fairness: The Ready List ensures that all tasks are given a fair chance to run, regardless of their priority.
- Flexibility: The Ready List can be used to manage a variety of task states, such as when a task is blocked or running.

The Ready List is a critical component of FreeRTOS that plays an important role in managing tasks' states and ensuring that tasks are scheduled in a fair and efficient manner.

III. Examining How the Ready List Was Implemented

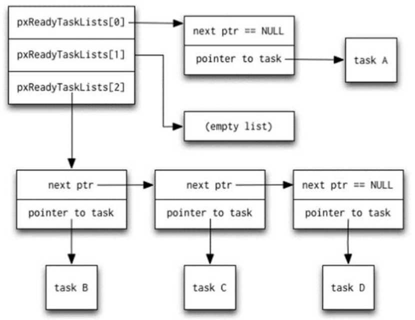
The Ready List in FreeRTOS is implemented as an array of doubly linked list. The array is indexed using the priority of the task(0 being the highest priority). The structure of the Ready list is as follows:



```
struct xLIST_ITEM
{
  portTickType xItemValue;                 /* The value being listed.  In most cases
                                              this is used to sort the list in
                                              descending order. */
  volatile struct xLIST_ITEM * pxNext;     /* Pointer to the next xListItem in the
                                              list.  */
  volatile struct xLIST_ITEM * pxPrevious; /* Pointer to the previous xListItem in
                                              the list. */
  void * pvOwner;                          /* Pointer to the object (normally a TCB)
                                              that contains the list item.  There is
                                              therefore a two-way link between the
                                              object containing the list item and
                                              the list item itself. */
  void * pvContainer;                      /* Pointer to the list in which this list
                                              item is placed (if any). */
};
```

The choice of a linked list data structure, specifically a task list, for implementing the Ready List in FreeRTOS offers several advantages in terms of performance and memory usage.

- Dynamic Size: Linked lists allow for the dynamic addition and removal of elements. This makes it simple to add or remove tasks from the Ready List as their status during execution changes. The list can grow or shrink dynamically depending on the number of ready tasks, ensuring effective memory usage.
- Priority-based selection: Task prioritisation is made simple with linked lists. The scheduler can quickly determine which task has the highest priority by arranging the list items according to task priorities. The task selection process is made easier by this prioritisation, which also guarantees that the CPU is swiftly turned over to higher priority tasks.
- Linked lists offer a comparatively minimal memory overhead when compared to other data structures like arrays. Only the extra memory needed to store pointers and information is needed for each list item. They are thus especially well suited to resource-constrained systems where memory utilisation is important.
- Flexibility in Traversal: The scheduling techniques are flexible since linked lists may be traversed both forward and backward. Circular doubly linked lists, for instance, allow for simple movement in both directions, making it easier to pick jobs based on various scheduling principles or context-switching needs.



Basic view of FreeRTOS Ready List

IV. Operations Performed on the ready list:

The prvAddTaskToReadyList() macro in FreeRTOS is used to add a task to the ready list. The macro takes a single parameter, which is a pointer to the task control block (TCB) of the task to be added. The macro works as follows:

1. It calls the traceMOVED_TASK_TO_READY_STATE() function to record that the task has been moved to the ready state.
2. It calls the taskRECORD_READY_PRIORITY() function to record the task's priority.
3. It calls the vListInsertEnd() function to insert the task's TCB into the ready list at the end of the list for the task's priority.

vListInsert And uxListRemove are functions used to manipulate linked lists, which are fundamental data structures used in the kernel for managing tasks and other kernel objects. These operations are mostly employed for task synchronisation and scheduling.

1. **vListInsert:** A linked list can have items inserted into it in a prioritised order using the vListInsert function. It requires a pointer to the item that needs to be inserted as well as the list that it should go into.
2. **uxListRemove**: An item can be eliminated from a linked list using the uxListRemove function. It requires a pointer to the item that needs to be deleted and then returns the number of items that are still in the list after the deletion. Prior to calling this function, the item must already be in the list.

The vTaskSwitchContext() function in FreeRTOS facilitates task switching. It updates flags and performs tracing operations, such as recording task switches and runtime statistics. It selects the highest priority ready task, schedules it for execution, and restores its context, ensuring smooth task transitions in the system.

pxCurrentTCB is a global variable in FreeRTOS that points to the task control block (TCB) of the currently running task. The TCB is a data structure that stores all of the information about a task, such as its stack pointer, its priority, its state, and its handle.

Time Complexity Analysis:

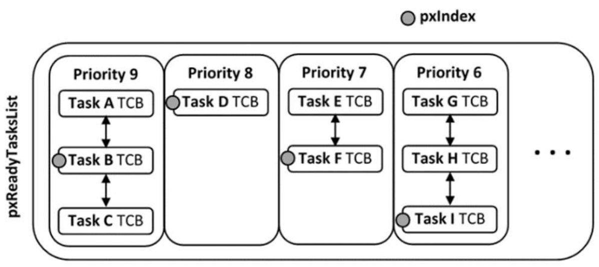
The time complexity of the prvAddTaskToReadyList() macro is O(n), where n is the number of tasks already in the ready list for the task's priority. This is because the vListInsertEnd() function has to iterate through the list to find the end, which takes O(n) time.However, the vListInsertEnd() function can be optimized to have a constant time complexity if the list is implemented as a doubly linked list. In this case, the vListInsertEnd() function can simply add the task to the end of the list without having to iterate through the list.

The time complexity of the prvAddTaskToReadyList() macro can also be reduced by using a hash table to store the ready lists. In this case, the task can be inserted into the hash table in O(1) time.

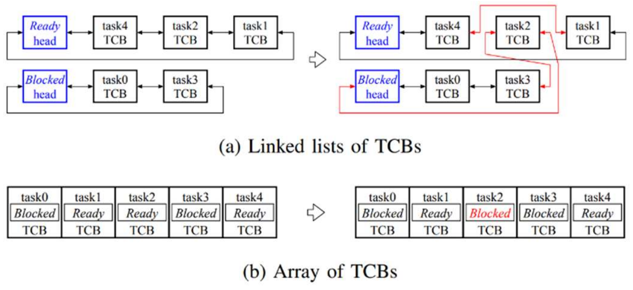
The time complexity of vTaskSwitchContext() from FreeRTOS code is O(n), where n is the number of tasks in the system. This is because vTaskSwitchContext() must save the state of all of the running tasks before it can switch to a new task. The state of a task includes its stack pointer, registers, and priority. Saving the state of all of the tasks can be a relatively expensive operation, especially if there are a large number of tasks in the system.

V. Integration of the Ready List with the Scheduler:

The Ready List is a crucial component of the scheduler in FreeRTOS and plays a key role in task scheduling, context switching, and task preemption. It represents the list of tasks that are ready to run, meaning they have met their scheduling criteria and are waiting for the CPU to execute them.



When a task becomes ready, it is added to the Ready List based on its priority. The Ready List is organized in a priority-based order, with tasks of higher priority at the front of the list. This allows the scheduler to efficiently select the highest priority task for execution.The vTaskSwitchContext() function in the FreeRTOS kernel plays a crucial role in task switching within the system. Before performing any actions, it first checks if the scheduler is suspended. If it is indeed suspended, the function does nothing and returns. However, if the scheduler is active, the function proceeds with the following steps.Firstly, it updates the xYieldPending variable, which is a flag indicating whether a task has requested a context switch. This flag is set to pdFALSE, indicating that no context switch is currently pending.Next, the function invokes the traceTASK_SWITCHED_OUT() function. This function is responsible for recording the fact that the currently running task is being switched out of execution. The FreeRTOS trace facility utilizes this information to keep track of task switching events.In cases where the configGENERATE_RUN_TIME_STATS configuration option is enabled, the function may additionally update the run time counter of the current task. This counter keeps track of the amount of time the task has been running, aiding in runtime statistics generation.Similarly, if the configUSE_NEWLIB_REENTRANT configuration option is enabled, the function updates the _impure_ptr variable to point to the _reent structure specific to the current task. This is necessary for ensuring thread safety in applications that utilize the Newlib C library with reentrancy support.Following these preliminary steps, the function calls the taskSELECT_HIGHEST_PRIORITY_TASK() function.



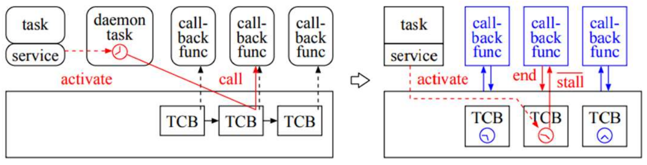
(a) Linked lists of TCBs

(b) Array of TCBs

In other words, The highest priority task is granted CPU time. If multiple tasks have equal priority, it uses round-robin scheduling among them. Lower priority tasks must wait.

It is important that high priority tasks don't execute 100% of the time, because lower priority tasks would never get CPU time. It's a fundamental problem of real-time programming.

Usually, you want to assign a high priority to a task that must react fast to some important event, perform quick action, and go to sleep, letting less important stuff to work in the meantime.



Implementation of software timer

A generic example of such a system may be:

1. highest priority - device drivers tasks (valve control, ADC, DAC, etc)
2. medium priority - administrative subsystem (console task, telnet task)
3. lower priority - several application tasks (www server, data processing, etc)

Lowest priority is given to general applications, that are scheduled using round-robin, which gives a more or less equal number of CPU time.

vTaskSwitchContext() selects the highest-priority ready task and puts it in the pxCurrentTCB variable like this:

```
/* Find the highest-priority queue that contains ready tasks. */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT( uxTopReadyPriority );
    --uxTopReadyPriority;
}

/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of the same
priority get an equal share of the processor time. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopReadyPriority ] ) );
```

Medium priority - console tasks. The system operator cannot be cut off by a malfunctioning www server that gets stuck in an infinite loop. Those tasks are not running 100% of the time. For example, it may execute command-line commands from the administrator.

Highest priority - device drivers, handling critical events, such as machinery control. You may be interested in opening a safety valve if boiler pressure gets too high and you really don't want to wait until some stupid HTML rendering is finished in the webserver thread. Such tasks are run for a limited amount of time only.

Methods to prevent race Conditions and Synchronisation method:

Mutexes (short for mutual exclusion) are used to protect shared resources from simultaneous access by multiple tasks. They ensure that only one task can access a critical section of code or a shared resource at a time. FreeRTOS provides the **xSemaphoreTake()** and **xSemaphoreGive()** functions to acquire and release mutexes, respectively. Semaphores are used for signaling and synchronization between tasks. They can be used to control access to a shared resource or to coordinate task execution. FreeRTOS provides binary semaphores (**xSemaphoreCreateBinary()**) and counting semaphores (**xSemaphoreCreateCounting()**) for different synchronization needs. Queues are used for inter-task communication and synchronization. They allow tasks to send and receive messages or data items in a thread-safe manner. FreeRTOS provides **xQueueSend()** and **xQueueReceive()** functions to send and receive data through queues. Software timers allow tasks to schedule and synchronize actions based on time intervals. They can be used for periodic tasks, timeouts, or delaying operations. FreeRTOS provides the **xTimerCreate()** function to create and manage software timers.

These synchronization mechanisms help prevent race conditions and ensure thread safety by providing controlled access to shared resources and orderly execution of tasks in a concurrent system. Proper usage of these mechanisms in critical sections of code can help avoid conflicts and maintain data integrity.

Conclusion:

The analysis of the implementation of the Ready List in FreeRTOS highlights its significance in achieving efficient task scheduling and management. The Ready List plays a crucial role in determining the order in which tasks are executed based on their priorities. It has been implanted through the data structure Array of Doubly linked lists. It allows the scheduler to quickly select the highest priority task for execution, facilitating effective utilization of system resources. By organizing tasks in a priority-based order, the Ready List enables efficient context switching and task preemption. Context switching is triggered by events such as task yield, delay, or the arrival of higher-priority tasks. The scheduler utilizes the Ready List to determine the next task to run during a context switch, ensuring smooth transitions between tasks.

Studying the FreeRTOS source code provides insights into the inner workings of a real-time operating system and the complexities involved in managing concurrent tasks. In order to ensure thread safety and avoid race conditions, the analysis emphasises the significance of synchronisation mechanisms like mutexes, semaphores, and queues that work with the Ready List.The Ready List serves as a crucial building block for efficient task scheduling and coordination in FreeRTOS, enabling the creation of reliable and responsive real-time applications.

References:

W. Nakano, Y. Shinohara and N. Ishiura, "Full Hardware Implementation of FreeRTOS-Based Real-Time Systems," TENCON 2021 - 2021 IEEE Region 10 Conference (TENCON), Auckland, New Zealand, 2021, pp. 435-440, doi: 10.1109/TENCON54134.2021.9707328.

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