DSA proj

Description:

1. Introduction:

Briefly introduce FreeRTOS and its significance in real-time embedded systems.
Explain the purpose of the report and the specific focus on the implementation of the Ready List in the FreeRTOS scheduler.
Provide an overview of the structure of the report.

1. Overview of FreeRTOS Scheduler:

Provide a high-level explanation of the FreeRTOS scheduler's role in task management and scheduling.
Discuss the importance of the Ready List in the scheduler's functionality.

1. Understanding the Ready List:

Explain the concept of the Ready List and its purpose in task scheduling.
Discuss the significance of the Ready List in managing tasks' states (i.e., ready, blocked, running) efficiently.

4.Analyzing the Implementation of the Ready List:

Describe the data structure used to implement the Ready List in FreeRTOS.
Explain the rationale behind the choice of data structure and its advantages in terms of performance and memory usage.

1. Detailed Examination of Ready List Operations:

Discuss the operations performed on the Ready List, such as task insertion, removal, and prioritization.
Provide code snippets or pseudocode to illustrate how these operations are implemented in the FreeRTOS source code.
Analyze the time complexity of these operations and their impact on the scheduler's efficiency.

1. Integration of the Ready List with the Scheduler:

Explain how the Ready List interacts with other components of the scheduler, such as context switching and task preemption.
Discuss any synchronization mechanisms or algorithms used to ensure thread safety and prevent race conditions.

Task Control Block- Structure

```
* Task control block.  A task control block (TCB) is allocated for each task,
 * and stores task state information, including a pointer to the task's context
 * (the task's run time environment, including register values)
 */
```

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t  *pxTopOfStack;  /*< Points to the location of the last item placed on the tasks stack.  THIS MUST BE THE FIRST MEM

#if ( portUSING_MPU_WRAPPERS == 1 )
    xMPU_SETTINGS xMPUSettings;   /*< The MPU settings are defined as part of the port layer.  THIS MUST BE THE SECOND MEMBER OF THE TCB ST
    BaseType_t    xUsingStaticallyAllocatedStack; /* Set to pdTRUE if the stack is a statically allocated array, and pdFALSE if the stack i
#endif

    ListItem_t      xGenericListItem; /*< The list that the state list item of a task is reference from denotes the state of that task (Rea
    ListItem_t      xEventListItem;   /*< Used to reference a task from an event list. */
    UBaseType_t     uxPriority;     /*< The priority of the task.  0 is the lowest priority. */
    StackType_t     *pxStack;     /*< Points to the start of the stack. */
    char        pcTaskName[ configMAX_TASK_NAME_LEN ];/*< Descriptive name given to the task when created.  Facilitates debugging only. */

#if ( portSTACK_GROWTH > 0 )
    StackType_t   *pxEndOfStack;    /*< Points to the end of the stack on architectures where the stack grows up from low memory. */
#endif

#if ( portCRITICAL_NESTING_IN_TCB == 1 )
    UBaseType_t   uxCriticalNesting;  /*< Holds the critical section nesting depth for ports that do not maintain their own count in the po
#endif

#if ( configUSE_TRACE_FACILITY == 1 )
    UBaseType_t   uxTCBNumber;    /*< Stores a number that increments each time a TCB is created.  It allows debuggers to determine when a
    UBaseType_t   uxTaskNumber;   /*< Stores a number specifically for use by third party trace code. */
#endif

#if ( configUSE_MUTEXES == 1 )
    UBaseType_t   uxBasePriority;   /*< The priority last assigned to the task - used by the priority inheritance mechanism. */
    UBaseType_t   uxMutexesHeld;
#endif

#if ( configUSE_APPLICATION_TASK_TAG == 1 )
    TaskHookFunction_t pxTaskTag;
#endif

#if ( configGENERATE_RUN_TIME_STATS == 1 )
    uint32_t    ulRunTimeCounter; /*< Stores the amount of time the task has spent in the Running state. */
#endif

#if ( configUSE_NEWLIB_REENTRANT == 1 )
    /* Allocate a Newlib reent structure that is specific to this task.
    Note Newlib support has been included by popular demand, but is not
    used by the FreeRTOS maintainers themselves.  FreeRTOS is not
    responsible for resulting newlib operation.  User must be familiar with
    newlib and must provide system-wide implementations of the necessary
    stubs. Be warned that (at the time of writing) the current newlib design
    implements a system-wide malloc() that must be provided with locks. */
    struct  _reent xNewLib_reent;
#endif

#if ( configUSE_TASK_NOTIFICATIONS == 1 )
    volatile uint32_t ulNotifiedValue;
    volatile eNotifyValue eNotifyState;
#endif
    //#if ( configUSE_EDFVD_SCHEDULER == 1 )
    /*UBaseType_t task_NormalBudget;
    UBaseType_t task_SafeBudget;
    UBaseType_t task_DegradedBudget1;
    UBaseType_t task_DegradedBudget2;
    UBaseType_t task_Behaviour1;
    UBaseType_t task_Behaviour2;
    UBaseType_t task_Criticality;
    UBaseType_t task_current_executionLimit;
    UBaseType_t task_current_Behaviour;
    UBaseType_t uxOriginalPriority;
    UBaseType_t uxPseudoPriority;*/

    //uint32_t taskProperty[3][4] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    UBaseType_t task_Criticality;
    UBaseType_t task_current_executionLimit;
    UBaseType_t task_current_Behaviour;
    UBaseType_t taskID;
    UBaseType_t task_SafeBudget;


    //   #endif
} tskTCB;
```

The provided code defines a structure named `tskTaskControlBlock`, which represents the control block or control structure used by an operating system to manage a task in an RTOS environment. Let's go through the different members of this structure:

1. `pxTopOfStack`: A pointer to the location of the last item placed on the task's stack. It is typically used during context switching to restore the task's execution state.
2. `xMPUSettings` and `xUsingStaticallyAllocatedStack`: These members are conditionally compiled based on the `portUSING_MPU_WRAPPERS` configuration. If enabled, they hold settings related to Memory Protection Unit (MPU) usage and whether the stack is statically allocated or dynamically allocated.
3. `xGenericListItem`: This member represents a list item used to link the task control block to a list that denotes the state of the task (e.g., Ready, Blocked, or Suspended).

4. `xEventListItem`: This member is used to reference the task from an event list. Event lists are typically used for synchronization between tasks.
5. `uxPriority`: The priority of the task. It represents the scheduling priority assigned to the task, with 0 being the lowest priority.
6. `pxStack`: Points to the start of the stack associated with the task. It allows the RTOS to access the task's stack for stack management and context switching.
7. `pcTaskName`: A character array that holds a descriptive name given to the task when it was created. This name is mainly used for debugging purposes.
8. `pxEndOfStack`: This member is conditionally compiled based on the `portSTACK_GROWTH` configuration. It points to the end of the stack, useful for architectures where the stack grows up from low memory.
9. `uxCriticalNesting`: If `portCRITICAL_NESTING_IN_TCB` is enabled, this member holds the critical section nesting depth. It is used in ports that do not maintain their own count in the port layer.
10. `uxTCBNumber` and `uxTaskNumber`: These members are conditionally compiled based on the `configUSE_TRACE_FACILITY` configuration. They store numbers used for debugging and tracing purposes, allowing debuggers or third-party trace code to identify tasks uniquely.
11. `uxBasePriority` and `uxMutexesHeld`: These members are conditionally compiled based on the `configUSE_MUTEXES` configuration. They store information related to priority inheritance mechanism and the number of mutexes held by the task.
12. `pxTaskTag`: This member is conditionally compiled based on the `configUSE_APPLICATION_TASK_TAG` configuration. It represents a hook function associated with the task.
13. `ulRunTimeCounter`: Conditionally compiled based on the `configGENERATE_RUN_TIME_STATS` configuration, this member stores the amount of time the task has spent in the Running state. It is used for runtime statistics generation.
14. `xNewLib_reent`: If `configUSE_NEWLIB_REENTRANT` is enabled, this member allocates a Newlib reent structure specific to the task. Newlib is a C library and this structure allows reentrancy support.
15. `ulNotifiedValue` and `eNotifyState`: These members are conditionally compiled based on the `configUSE_TASK_NOTIFICATIONS` configuration. They are used for task notification mechanism, storing the notified value and the state of the task's notification.
16. `task_Criticality`, `task_current_executionLimit`, `task_current_Behaviour`, `taskID`, and `task_SafeBudget`: These members are additional task-specific variables added to the structure. They store information related to task critical
17. `uxSchedulerSuspended`: This is a volatile variable that indicates whether the scheduler is currently suspended or not. When the scheduler is suspended, context switches are held pending, and interrupts must not manipulate the data structures related to task scheduling. If an interrupt needs to unblock a task while the scheduler is suspended, it moves the task's event list item into the `xPendingReadyList`, which is a list of tasks that have been readied while the scheduler was suspended. The tasks in this list will be moved to the real ready list when the scheduler is unsuspended. Access to the `xPendingReadyList` is restricted to critical sections.
18. `ulTaskSwitchedInTime` and `ulTotalRunTime`: These variables are used for generating run-time statistics if the configuration option `configGENERATE_RUN_TIME_STATS` is enabled. They are used to measure and store the execution time of tasks. `ulTaskSwitchedInTime` holds the value of a timer/counter the last time a task was switched in, and `ulTotalRunTime` stores the total amount of execution time defined by the run-time counter clock.

```
PRIVILEGED_DATA TCB_t * volatile pxCurrentTCB = NULL;
```

This line declares a volatile pointer variable named `pxCurrentTCB` of type `TCB_t`. The `volatile` qualifier indicates that the variable may be modified asynchronously by an interrupt or another task. In this case, it is initialized to `NULL`.

```
PRIVILEGED_DATA static List_t pxReadyTasksLists[configMAX_PRIORITIES]; /*< Prioritized ready tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList1; /*< Delayed tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList2; /*< Delayed tasks (two lists are used - one for delays that have overflowed the current ti
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList; /*< Points to the delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList; /*< Points to the delayed task list currently being used to hold tasks
PRIVILEGED_DATA static List_t xPendingReadyList; /*< Tasks that have been readied while the scheduler was suspended. They will be moved to
```

These lines declare various static variables with different types related to task scheduling and management. The `PRIVILEGED_DATA` macro likely denotes that these variables are accessed in privileged mode. Here's a breakdown of each variable:

- `pxReadyTasksLists` is an array of `List_t` structures, representing prioritized ready tasks. The size of the array is determined by the `configMAX_PRIORITIES` configuration parameter.
- `xDelayedTaskList1` and `xDelayedTaskList2` are `List_t` structures used to hold delayed tasks. Two lists are used to handle delays that have overflowed the current tick count.
- `pxDelayedTaskList` and `pxOverflowDelayedTaskList` are volatile pointers to `List_t`. They point to the delayed task list currently being used and the list holding tasks that have overflowed the current tick count, respectively. The `volatile` qualifier indicates that these pointers may be modified asynchronously.
- `xPendingReadyList` is a `List_t` structure used to hold tasks that have been readied while the scheduler was suspended. These tasks will be moved to the ready list when the scheduler is resumed.

```
PRIVILEGED_DATA static List_t pxReadyTasksLists[configMAX_PRIORITIES]; /*< Prioritized ready tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList1; /*< Delayed tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList2; /*< Delayed tasks (two lists are used - one for delays that have overflowed the current ti
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList; /*< Points to the delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList; /*< Points to the delayed task list currently being used to hold tasks
PRIVILEGED_DATA static List_t xPendingReadyList; /*< Tasks that have been readied while the scheduler was suspended. They will be moved to
```

These lines declare several variables with different types and qualifiers. Here's a breakdown of each variable:

- `pxReadyTasksLists` is an array of `List_t` structures used to represent prioritized ready tasks. The size of the array is determined by the `configMAX_PRIORITIES` configuration value.
- `xDelayedTaskList1` and `xDelayedTaskList2` are `List_t` structures representing delayed tasks. The comment indicates that two lists are used, likely for handling delays that have overflowed the current tick count.
- `pxDelayedTaskList` and `pxOverflowDelayedTaskList` are volatile pointers to `List_t`. They serve as references to the delayed task lists currently being used. The `volatile` qualifier indicates that these pointers may be modified asynchronously.
- `xPendingReadyList` is a `List_t` structure used to hold tasks that have been readied while the scheduler was suspended. The comment explains that these tasks will be moved to the ready list when the scheduler is resumed.

note:

The `configMAX_PRIORITIES` is a configuration macro or constant defined in the system. It represents the maximum number of priorities supported by the system's task scheduler or scheduler algorithm.

In a real-time operating system (RTOS) or task scheduling framework, tasks are often assigned priorities to determine their relative importance and order of execution. The `configMAX_PRIORITIES` macro defines the total number of priority levels available in the system. Each task is assigned a priority value within this range, typically ranging from 0 (highest priority) to `configMAX_PRIORITIES - 1` (lowest priority).

The specific value of `configMAX_PRIORITIES` depends on the system or application requirements and can vary. It is typically defined in a configuration header file or as a compile-time macro, allowing the system designer to adjust the number of priority levels based on the application's needs. The value is chosen to strike a balance between the granularity of priority levels and the system's resource limitations.

1. `#if (INCLUDE_vTaskDelete == 1)`
   This conditional block checks if the macro `INCLUDE_vTaskDelete` is defined and has a value of 1. If so, the code within this block is included for compilation.
   - `PRIVILEGED_DATA static List_t xTasksWaitingTermination;`
     This line declares a static variable `xTasksWaitingTermination` of type `List_t`. It represents a list of tasks that have been deleted but their memory has not yet been freed.

- `PRIVILEGED_DATA static volatile UBaseType_t uxTasksDeleted = (UBaseType_t) 0U;`
  This line declares a static volatile variable `uxTasksDeleted` of type `UBaseType_t`. It represents the number of tasks that have been deleted. The `volatile` keyword indicates that the variable may be modified asynchronously by interrupt handlers or other threads.

2. `#if (INCLUDE_vTaskSuspend == 1)`
   This conditional block checks if the macro `INCLUDE_vTaskSuspend` is defined and has a value of 1. If so, the code within this block is included for compilation.

   - `PRIVILEGED_DATA static List_t xSuspendedTaskList;`
     This line declares a static variable `xSuspendedTaskList` of type `List_t`. It represents a list of tasks that are currently suspended.

3. `#if (INCLUDE_vTaskDelete == 1)`
   This conditional block checks if the macro `INCLUDE_vTaskDelete` is defined and has a value of 1. If so, the code within this block is included for compilation.

   - `PRIVILEGED_DATA static List_t xTasksWaitingTermination;`
     This line declares a static variable `xTasksWaitingTermination` of type `List_t`. It represents a list of tasks that have been deleted but their memory has not yet been freed.

   - `PRIVILEGED_DATA static volatile UBaseType_t uxTasksDeleted = (UBaseType_t) 0U;`
     This line declares a static volatile variable `uxTasksDeleted` of type `UBaseType_t`. It represents the number of tasks that have been deleted. The `volatile` keyword indicates that the variable may be modified asynchronously by interrupt handlers or other threads.

line 233:Definition of ready list

```
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];/*< Prioritised ready tasks. */
```

The line `PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];` declares a static array `pxReadyTasksLists` of type `List_t`. Each element of the array represents a list for tasks that are ready to be executed at a specific priority level. The array size is determined by the macro `configMAX_PRIORITIES`, which defines the maximum number of priority levels in the system.

In FreeRTOS, tasks are organized based on their priorities, where higher-priority tasks preempt lower-priority tasks. The `pxReadyTasksLists` array provides a data structure to hold these ready tasks in a prioritized manner.

Here's how the ready list works:

- Each element of the `pxReadyTasksLists` array represents a priority level. For example, `pxReadyTasksLists[0]` represents the list of tasks at the lowest priority, `pxReadyTasksLists[1]` represents the list of tasks at the next higher priority, and so on.
- The `List_t` type is a linked list structure that includes pointers to the first and last items in the list, as well as the number of items in the list. It provides operations to manipulate the list, such as inserting, removing, or iterating over the elements.
- By maintaining a separate list for each priority level, tasks can be efficiently organized and retrieved based on their priority. When a task becomes ready, it is added to the corresponding list according to its priority.
- The `pxReadyTasksLists` array is defined as `static`, which means it retains its value between function calls and has a global scope within the file where it is defined. This allows the ready lists to be accessed and modified by various functions in that file.

Overall, the `pxReadyTasksLists` array provides a data structure to manage ready tasks at different priority levels, allowing the FreeRTOS scheduler to prioritize and execute tasks based on their assigned priorities.

line 413:prvaddtasktoreadylist

Function Definition:

```
#define prvAddTaskToReadyList( pxTCB )                                    \
        traceMOVED_TASK_TO_READY_STATE( pxTCB )                           \
        taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );               \
        vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xGenericListItem ) )
```

The macro `prvAddTaskToReadyList` is used in FreeRTOS to add a task to the ready list, indicating that the task is now ready for execution. Let's break down the usage of this macro:

1. `traceMOVED_TASK_TO_READY_STATE( pxTCB )`: This is a trace macro used for debugging or monitoring purposes. It records that a task has transitioned to the ready state. The `pxTCB` parameter represents a pointer to the Task Control Block (TCB) of the task being added to the ready list.
2. `taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority )`: This macro updates the readiness status of the task at its specific priority level. It records that a task with a certain priority (`( pxTCB )->uxPriority`) is ready for execution. The `taskRECORD_READY_PRIORITY` macro internally calls `portRECORD_READY_PRIORITY` to set the corresponding bit in the readiness status.
3. `vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xGenericListItem ) )`: This function call inserts the task's list item (`( pxTCB )->xGenericListItem`) at the end of the list corresponding to its priority level (`( pxTCB )->uxPriority`). The `pxReadyTasksLists` is an array of lists, where each index represents a priority level, and the list contains the tasks ready at that priority level.

By invoking `prvAddTaskToReadyList`, the task's readiness is recorded, its priority is updated, and it is inserted into the appropriate ready list. This process ensures that tasks are appropriately prioritized and ready for execution based on their priority levels.

another definiton:

The function `prvAddTaskToReadyList` is responsible for adding a task to the appropriate ready list in FreeRTOS. It is used in the context of task scheduling when a task is ready to run.

In the provided code, `prvAddTaskToReadyList` is called to add a task (`pxUnblockedTCB`) to the ready list after it has been removed from an event list or a delayed list.

Here is an overview of the purpose of `prvAddTaskToReadyList`:

1. The function first determines the index of the ready list based on the priority of the task (`pxUnblockedTCB->uxPriority`).
2. It then checks if the ready list at the determined index is empty. If it is empty, the task is added directly to the head of the list.
3. If the ready list is not empty, the task is inserted into the ready list in priority order. The task is placed after any existing tasks with the same priority and before tasks with lower priorities.
4. After the task is added to the ready list, the `uxCurrentNumberOfTasks` counter is incremented to reflect the increase in the number of tasks that are ready to run.

The purpose of `prvAddTaskToReadyList` is to maintain the ready lists in a way that allows tasks to be selected for execution based on their priorities. By adding tasks to the appropriate ready list, FreeRTOS ensures that tasks with higher priorities are given preference in the scheduling algorithm.

Note that the specific implementation of `prvAddTaskToReadyList` may vary depending on the version and configuration of FreeRTOS being used. The provided explanation is a general overview of the purpose and behavior of this function.

Other Places where this function is used:

(lines 719-735):

In this block of code after creating a new task and checking if the trace facility is enabled the addTasktoReadyList function is called and the task is inserted at the right posititon in the ready List based upon its priority.This ensures that the task is scheduled and ready to be executed by the RTOS scheduler.

(lines 1312-1387):

In this block of code first the priority of the task is updated . Then we check if the task is present in the ready list if yes it is first removed from the the list using uxListRemove and added back using the Add task function . If the priority of the task has changed it may need to be moved to a new position in the ready list, this is taken care by the prvAddTaskToReadyList function.After this it is checked if preemption is required using the taskYield_if_using_preemption.

(lines 1525-1575):

In this part of the code we first check if the task is suspended .If yes we remove the task from the suspended list using the uxListRemove() function with the task's generic list item as the argument and then we add it back to the ready list for execution.

(lines 1610-1645):

In summary, this code snippet handles the resumption of a task from an ISR (Interrupt Service Routine). If the task is suspended, it checks the scheduler's state, moves the task to the ready list if the scheduler is not suspended, sets the yield flag if necessary, or holds the task in the pending ready list if the scheduler is suspended.

(lines 1780-1865):

In FreeRTOS, the scheduler can be suspended temporarily to perform certain operations without task switching. Suspending the scheduler is done by entering a critical section using `taskENTER_CRITICAL()` or similar functions.

The purpose of this code is to resume all tasks that were pending while the scheduler was suspended. It moves these tasks from the pending list to their respective ready lists, processes any pending ticks, and potentially performs a context switch if a higher-priority task becomes ready.A critical section is a block of code where the scheduler is temporarily suspended to ensure the atomicity of certain operations.

(lines 2761-2810):

This code represents the function `xTaskRemoveFromEventList()`, which is responsible for removing a task from an event list and adding it to the ready list for execution.The process is as follows:

- The task is removed from the event list using `uxListRemove()`. At this point, the task is considered unblocked and ready to run.
- If the scheduler is not suspended (`uxSchedulerSuspended == pdFALSE`), the task is removed from the generic list (if present) and added to the ready list using `prvAddTaskToReadyList()`. This allows the task to be scheduled for execution.

(lines 2829-2870):

This function removes a task from an unordered event list, adds it to the ready list, and provides information about the need for a context switch based on task priorities. It is specifically designed for use in the event flags implementation and requires the scheduler to be suspended during its execution.

(lines 3671-3729):

It is regarding the inheritance of priority to ensure that a high priority task is not starved.The priority of the low priority task is increased

Time complexity:

The time complexity of `prvAddTaskToReadyList` depends on the time complexity of `vListInsertEnd`, the function called within it.

In `vListInsertEnd`, a new list item is inserted at the end of the specified list. The function performs the following operations:

1. Assign the `pxNext` and `pxPrevious` pointers of the new list item.
2. Update the `pxNext` and `pxPrevious` pointers of neighboring list items to maintain the linked list structure.
3. Update the `pvContainer` pointer of the new list item to point to the list it is inserted into.
4. Increment the `uxNumberOfItems` counter of the list.

The time complexity of `vListInsertEnd` depends on the number of items in the list. In the worst case scenario, where the new list item is inserted at the end of a list with `n` items, the time complexity is O(n). This is because the function needs to traverse the list to reach the last item and perform the necessary pointer updates.

Therefore, the time complexity of `prvAddTaskToReadyList` is also O(n), where `n` represents the number of tasks already present in the corresponding ready list. The time complexity increases linearly with the size of the ready list.

The function takes the pointer to the TCB as its parameter , checks the readiness of the task based on a certain priority and invokes the “vListInsertEnd” function to add the task to end of the ready list based upon the priority of the task. By inserting the task at the end it ensures that the task will be considered for execution after other tasks with same priority in the ready list.The free RTOS scheduler will select the task