Politecnico di Torino

Department of Control and Computer Engineering

Master's degree in Cybersecurity

Lottery Scheduling Implementation for FreeRTOS

Pietro Armenante - 329425
Christian Coduri - 333235
Giovanni Lombardi - 329015
Luca Serafini - 329442

Academic year 2023/2024

1 Introduction to FreeRTOS

FreeRTOS is a real-time operating system kernel for embedded devices, designed to be small and simple. It is mostly written in the C programming language to make it easy to port and maintain.

1.1 Explore Fundamentals APIs

To better understand FreeRTOS functionalities and its API, a practical example has been developed, integrating the essential components of the kernel for task execution.

1.1.1 UART Configuration

The UART (Universal Asynchronous Receiver-Transmitter) stands as one of the most widely employed device-to-device communication protocols. In this context, the UART peripheral facilitates the transmission and reception of serial data between the microcontroller (*Cortex-MPS2*) and external devices (*QEMU*). The initialization of the UART peripheral is carried out within the function `prvUARTInit`, enabling functionalities such as printing messages to the console using `printf`.

1.1.2 Task Creation API

To create tasks, in the main the function `xTaskCreate` is used. The required parameters include: a pointer to the task function (code executed by the task), a descriptive name for the task, the number of words allocated for the task's stack, the value passed as the parameter to the created task, the priority at which the task will execute (low priority numbers denote low priority tasks), and a handle to the created task.

```
#define TASK_1_PRIORITY 6
#define TASK_2_PRIORITY 1
#define TASK_3_PRIORITY 6
#define TASK_4_PRIORITY (configMAX_PRIORITIES - 1)

#define STK_SIZE 200

// ( ... )

void main(void)
{
    // (...)

    xRet_1 = xTaskCreate(vTask1, "vTask1", STK_SIZE, (void *)pcTextForTask1, TASK_1_PRIORITY, &xHandle_1);
    xRet_2 = xTaskCreate(vTask2, "vTask2", STK_SIZE, (void *)pcTextForTask2, TASK_2_PRIORITY, &xHandle_2);
    xRet_3 = xTaskCreate(vTask3, "vTask3", STK_SIZE, (void *)pcTextForTask3, TASK_3_PRIORITY, &xHandle_3);
    xRet_4 = xTaskCreate(vTask4, "vTask4", STK_SIZE, (void *)pcTextForTask4, TASK_4_PRIORITY, &xHandle_4);

    // (...)
}
```

1.1.3 Mutex API for Synchronization

It is possible to create a mutex simply by using the FreeRTOS Semaphore API.

```
xMutex = xSemaphoreCreateMutex();
```

This semaphore will be used to ensure mutual exclusion while accessing the UART peripheral from multiple tasks, preventing data corruption or conflicts during simultaneous access.

```
void vTask1(void *pvParameters)
{
    char *pcTaskName = (char *)pvParameters;

    for (;;)
    {
        // Attempt to take the mutex, blocking indefinitely if necessary
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE)
        {
            // ( ... )

```

```
            // Release the mutex
            xSemaphoreGive(xMutex);

            // Introduce a time-slice delay
            vTaskDelay(pdMS_TO_TICKS(ROUND_ROBIN_TIME_SLICE));
        }
        else
        {
            printf("Error taking the mutex\r\n");
        }
    }
}
```

In the example, the first three tasks are executed using this simple structure, alternating among themselves via semaphore synchronization.

1.1.4 Timer and Task Notifications API

In contrast, the execution of the fourth task relies on a timer mechanism. Initially, the timer must be created. The function API requires arguments such as a timer name, a timer period specified in ticks, a parameter indicating autoreload to determine whether it should be a one-shot timer or a repeated one, a Timer ID, and finally a callback function.

```
xTimer = xTimerCreate("Timer", pdMS_TO_TICKS(10000), pdTRUE, 0, vTimerCallback);
```

Concerning the timer period, the `pdMS_TO_TICKS` macro converts milliseconds to ticks, which represent the fundamental time unit in FreeRTOS. In this scenario, the timer period is configured for 10 seconds.

Following the initialization and creation, it is necessary to start the timer:

```
xTimerStart(xTimer, 0);
```

At this point, the timer starts counting. When it expires a timer callback function is called. This function's purpose is to notify Task 4 that the event has occurred. Subsequently, Task 4, which was awaiting this notification, can proceed to execute its code: attempting to acquire the semaphore and printing its message.

```
static void vTimerCallback(TimerHandle_t xTimer)
{
    // Notify the fourth task that 10 seconds have passed
    vTaskNotifyGiveFromISR(xHandle_4, NULL);
}

static void vTask4(void *pvParameters)
{
    char *pcTaskName = (char *)pvParameters;

    for (;;)
    {
        // Wait for the timer callback to signal that 10 seconds have passed
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        // Acquire semaphore and print the message as done for other tasks
        // ( ... )
    }
}
```

1.2 Task Priority & Scheduling

FreeRTOS implements a scheduling policy characterized by the following features:

- **Fixed-Priority**: Each task is assigned a priority from 0 to (`configMAX_PRIORITIES - 1`) and this remain constant, although they may receive temporary boosts.
- **Preemptive**: Ensures that the highest priority task capable of running takes precedence, even if it interrupts a lower priority task.
- **Round-Robin time-slicing**: allowing tasks of equal priority to take turns running.

One consequence of always running the highest priority task capable of running is that a high priority task that never enters the Blocked or Suspended state will permanently **starve all lower priority** tasks of any execution time. This is the reason why it is generally preferable to create tasks that are **event-driven**. This characteristic explains why Task 4 in the preceding example, which had the highest priority, did not starve the other tasks.

2 Purpose of the Report

The purpose of this report is to implement a lottery scheduling algorithm in FreeRTOS, providing an alternative to the default configuration discussed previously.

A lottery scheduler is a probabilistic scheduling algorithm structured around the following principles:

1. **Ticket Allocation**: Each process is assigned a specific number of lottery tickets according to its priority. Higher-priority processes typically receive more tickets, increasing their likelihood of being selected by the scheduler.

2. **Random Ticket Selection**: When it's time to choose the next process for execution, the scheduler randomly selects a ticket from the total pool of tickets.

3. **Winner Determination**: The process associated with the winning ticket is designated for execution. While all processes have the opportunity to be chosen, those with a greater number of tickets have higher odds of winning the lottery and being scheduled for execution.

4. **Execution**: The selected process is then given control of the CPU and allowed to execute for a certain time slice or until it terminates or yields the CPU.

5. **Repeat**: The lottery ticket scheduler continues to repeat this process, periodically selecting a new process to run based on the outcome of the lottery drawing.

On average, CPU time is proportional to the number of tickets given to each tasks. **Giving each process at least one lottery ticket guarantees that it has a non-zero probability of being selected at each scheduling operation**. For this reason the lottery ticket scheduler provides a fair method for allocating CPU time among processes, **mitigating the risk of starvation**.

2.1 Pseudo Code

Here's a pseudo code representation of the lottery ticket scheduler:

```
function lottery_scheduler(processes):
    total_tickets = calculate_total_tickets(processes)
    while True:
        winning_ticket = generate_random_number(1, total_tickets)
        selected_process = select_process_by_ticket(processes, winning_ticket)
        execute(selected_process)

def calculate_total_tickets(processes):
    total = 0
    for process in processes:
        total += process.tickets
    return total

def select_process_by_ticket(processes, winning_ticket):
    current_ticket = 0
    for process in processes:
        current_ticket += process.tickets
        if current_ticket >= winning_ticket:
            return process
```

2.2 Example

Let's consider a simplified scenario with two processes:

- Process A: 5 lottery tickets
- Process B: 3 lottery tickets

In total, there are 8 lottery tickets distributed among the two processes. Now, let's simulate the lottery drawing process. The scheduler generates a random number between 1 and 8 to select the winning ticket.

Suppose the generated random number is 6. To determine the winning process, we traverse the list of processes, keeping a running count of the tickets. Since the winning ticket falls within the range of Process B's tickets, Process B is selected as the winner and for this reason is scheduled to run next.

3 Implementation of Lottery Scheduling in FreeRTOS

To provide flexibility in selecting between priority-based and lottery scheduling algorithms, a similar configuration approach as other FreeRTOS settings is adopted.

3.1 Configuration Costant

In the file `FreeRTOSConfig.h`, there is a set of constants, which can be modified to alter the default behavior of the scheduler. This customization empowers users to use a scheduler that aligns better with their specific requirements

Within this file, a constant named `configUSE_TICKETS` has been defined. Setting it to 1 indicates the adoption of the lottery scheduling algorithm, while setting it to 0 implies the utilization of the default scheduler.

3.2 New TCB

To integrate this modification in an elegant way, the Task Control Block (TCB) structure has been revised to include the effective number of tickets assigned to each task.

```
// Task control block structure (one for each task)
typedef struct tskTaskControlBlock
{
    volatile StackType_t * pxTopOfStack;

    // Other parameters
    // (...)

    #if ( configUSE_TICKETS == 1)
        int nTickets;
    #endif

} tskTCB;
```

3.3 Ticket Allocation

In the `task.h` file, the prototype of the function `xTaskCreate`, which creates tasks in a dynamic way, has been modified to allow specifying the number of tickets directly when creating a new task.

```
#if (configSUPPORT_DYNAMIC_ALLOCATION == 1)
BaseType_t xTaskCreate(TaskFunction_t pxTaskCode,                   // Pointer to the task entry function
                       const char * const pcName,                   // Name for the task
                       const configSTACK_DEPTH_TYPE usStackDepth,   // Number of words to allocate - stack
                       void * const pvParameters,                   // Parameter passed to the created task
                       UBaseType_t uxPriority,                      // Priority of the created task
                       TaskHandle_t * const pxCreatedTask,          // To pass out the task's handler
                       int pxTicketNumber                           // Number of tickets that the task has
                       ) PRIVILEGED_FUNCTION;
#endif
```

The function `xTaskCreate`'s definition, contained in the file `task.c`, has been changed in the passed parameters. Moreover, an additional parameter has been included in the function call to `prvInitialiseNewTask`.

```c
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    BaseType_t xTaskCreate(...)
    {
        TCB_t * pxNewTCB;
        BaseType_t xReturn;

        // Allocating space for the Stack
        // (...)

        // Allocating space for the TCB
        // (...)

        if( pxNewTCB != NULL )
        {
            // (...)

            prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, pvParameters, uxPriority,
                                  pxCreatedTask, pxTicketNumber, pxNewTCB, NULL );
            prvAddNewTaskToReadyList( pxNewTCB );
            xReturn = pdPASS;
        }
        else
        {
            xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
        }

        return xReturn;
    }
#endif
```

The `prvInitialiseNewTask` function has been updated to store also the number of ticket with all the necessary information into the already allocated TCB. On the other hand, the functionality of the `prvAddNewTaskToReadyList` function, responsible for adding the TCB of the created task to the ready list, remains unaltered.

The added part in the `prvInitialiseNewTask` function is the following one:

```c
static void prvInitialiseNewTask(...)
{
    // (...)

    /* Store the number of tickets in the TCB. */
    #if (configUSE_TICKETS == 1)
        if(pxTicketNumber > 100)    // Task with pxTicketNumber setted at NULL (e.g IDLE task)
            pxNewTCB->nTickets = 0;
        else
            pxNewTCB->nTickets = pxTicketNumber;

        printf("\t-> The number of tickets set for the task %s is: %d\n", pcName, pxNewTCB->nTickets);
    #endif

    // (...)
}
```

Thanks to this process, the function to create task dynamically can be called in the following way:

```c
xTaskCreate(vTask, "vTask", STACK_SIZE, (void *)param, TASK_PRIORITY, &xHandle, 55);
```

Due to the addition of a new parameter, all occurrences of the `xTaskCreate` function throughout the Operating System have been modified. This change involves appending the new parameter, set to `null`, ensuring compatibility with the default scheduler algorithm.

3.4 Random Ticket Selection

With the structure in place, the next step is to implement the scheduling logic.

In the file `task.c` there is a function called `vTaskSwitchContext`, which is responsible for switching from one task to another when a context switch is required. This function achieves its functionalities by calling other two functions: `taskSELECT_HIGHEST_PRIORITY_TASK()` and `traceTASK_SWITCHED_IN()`. The first one is where the decision is made about which task should be selected to run next. After this call, the variable `pxCurrentTCB` holds a pointer to the TCB of the task to run. Then, the `traceTASK_SWITCHED_IN()` does the effective switch.

In order to implement our customized logic, an alternative to the `taskSELECT_HIGHEST_PRIORITY_TASK` function needed to be introduced. For this purpose, the `vTaskSwitchContext` function has been modified as follows :

```
void vTaskSwitchContext( void )
{
    // (...)

    # if ( configUSE_TICKETS == 1)
        int randomTicket = rand() % 100 +1;             // Selected ticket between 1 and 100
        taskSELECT_LOTTERY_WINNER_TASK(randomTicket);
    #else
        taskSELECT_HIGHEST_PRIORITY_TASK();
    #endif

    traceTASK_SWITCHED_IN();

    // (...)
}
```

The same modification has been applied to the `vTaskStartScheduler` function, which is invoked by the main to start the scheduler and choose the initial task to run. The only difference between the code added in this two functions is that in the `vTaskStartScheduler` there is a call to `srand()`, which initializes the seed for all subsequent calls to `rand()`.

3.5 Winner Determination

Finally, the function responsible for determining the winner in the lottery scheduling is defined as a macro named `taskSELECT_LOTTERY_WINNER_TASK`. This function takes the previously extracted `randomTicket` as input and use it to determine the winning process.

```
/* The assumption is that all the task we want to run with the lottery scheduling have the same priority. */
#define taskSELECT_LOTTERY_WINNER_TASK(randomTicket)
{
    UBaseType_t uxTopPriority = uxTopReadyPriority;
    int accumulatedTickets = 0;

    /* Find the highest priority queue that contains ready tasks (from taskSELECT_HIGHEST_PRIORITY_TASK) */
    portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );
    configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) ) > 0 );

    /* Following code obtained and adapted from listGET_OWNER_OF_NEXT_ENTRY */
    List_t* pxConstList = &(pxReadyTasksLists[uxTopPriority]);

    /* We want to start by looking always at the first task => listGET_HEAD_ENTRY */
        /* pxIndex = Used to walk through the List_t, point to a xLIST_ITEM */
        /* pvOwner = Pointer to the TCB. It is contained in the xLIST_ITEM */
    ListItem_t* pxListItem = listGET_HEAD_ENTRY(pxConstList); /*return ListItem */

    while(1){
                ( pxCurrentTCB ) = ( pxListItem )->pvOwner;
                accumulatedTickets = accumulatedTickets + ( pxCurrentTCB )-> nTickets;

                if(accumulatedTickets >= randomTicket){
                        printf("Extracted ticket: %d\tThe winner is %s which has %d tickets\n",
                                randomTicket, ( pxCurrentTCB )->pcTaskName, ( pxCurrentTCB )-> nTickets);
                        break;
                }

                pxListItem = ( pxListItem )->pxNext;
    }
} /* taskSELECT_LOTTERY_WINNER_TASK */
```

4 Evaluation of the new scheduler

As mentioned before, if each process receives at least one lottery ticket it is ensured a non-zero probability of selection during scheduling operations.

The implemented scheduling algorithm has changed FreeRTOS from a priority-based scheduler, which utilized Round-Robin (RR) scheduling in cases of equal priority, to a priority-based scheduler employing a

lottery algorithm under such circumstances. This is attributed to the fact that in the previously presented code, the first action undertaken is to find the higher priority queue of tasks (line 8, Section 3.5).

This enables us to establish a form of double-layered priority scheduler, where the first layer represents the actual priority, and the second layer is based on tickets.

4.1 Equal Priority Tasks: Analysis RR vs TS

The selected analysis aims to compare the two scheduling algorithms in the case of multiple tasks with equal priority. To accomplish this objective, a structure was implemented to manage time and counter variables. Furthermore, adjustments were made to the body of individual tasks to measure the distribution of task execution and the time used for context switching.

```
1  // Number of execution before print the statistics
2  #define TOT_EXECUTIONS 10000
3
4  // Structure for computing Tasks' statistics
5  typedef struct timeStats {
6      int numberOfExecution[NUMBER_OF_TASKS];
7      int totalNumberOfExecution;
8
9      uint32_t previousTaskEndTime;
10     uint32_t startTaskTime;
11     uint32_t cumulativeContextSwitchTime;
12 } timeStats;
13
14 // Used structure and initialization
15 timeStats stats;
16
17 // (...)
18
19 void main(void)
20 {
21     stats.totalNumberOfExecution = 0;
22     stats.previousTaskEndTime = NULL;
23     stats.startTaskTime = NULL;
24     stats.cumulativeContextSwitchTime = 0;
25
26     // (...)
27 }
28
29 void vTask1(void *paramTask1)
30 {
31     for (;;)
32     {
33         vTaskSuspendAll();
34         //Record the current system time as the start time in milliseconds
35         stats.startTaskTime = xTaskGetTickCount();
36
37         // If is not the first task => compute the time of context switch
38         if(stats.previousTaskEndTime != NULL)
39             stats.cumulativeContextSwitchTime += stats.startTaskTime - stats.previousTaskEndTime;
40
41         stats.numberOfExecution[0]++;
42         stats.totalNumberOfExecution++;
43
44         // Print statistics after the threshold TOT_EXECUTIONS has been reached
45         if(stats.totalNumberOfExecution % TOT_EXECUTIONS == 0)
46         {
47             for(int i = 0; i < NUMBER_OF_TASKS; i++)
48                 printf("Task %d usage:  %d out of %d\n", i+1, stats.numberOfExecution[i],
                    ↪  stats.totalNumberOfExecution);
49
50             printf("Total running time: %u\n",  xTaskGetTickCount());
51             printf("Total Context Switch Time: %u ms\n\n", stats.cumulativeContextSwitchTime);
52         }
53
54         // Set the previousTaskEndTime to compute the next context switch time
55         stats.previousTaskEndTime = xTaskGetTickCount();
56         xTaskResumeAll();
57     }
58 }
```

```
59
60 // (...)
```

The frequency of the RTOS tick interrupt is crucial for the time measurement within the analysis. It is defined in the `FreeRTOSConfig.h` file under the name `configTICK_RATE_HZ`. By setting this parameter to 1000 (1KHz), a tick interrupt occurs every 1ms.

4.1.1 Results of the analysis

From the tables provided below, it is observable the behavior of the two distinct scheduling approaches. Round Robin aims to allocate CPU utilization equally among all involved tasks, theoretically achieving a percentage of 33 for each task.

Given that in this implementation the total number of tickets is 100, the ticket scheduler achieves a distribution that respects the number of tickets assigned to each task. For instance, the `TS (33, 33, 34)` tends to achieve similar results to those of `RR`.

Table 1: Comparison of Round Robin and Ticket Scheduling after 10,000 task executions

Task	**RR**	**TS (33, 33, 34)**	**TS (60, 30, 10)**	**TS (89, 10, 1)**
Task 1	3.007 (30%)	2.938 (29%)	8.755 (88%)	10.000 (100%)
Task 2	2.456 (25%)	3.909 (39%)	1.245 (12%)	0 (0%)
Task 3	4.537 (45%)	3.153 (32%)	0 (0%)	0 (0%)
Running Time:	15ms	13ms	16ms	14ms
Context Switch Time:	13ms	12ms	15ms	12ms

Table 2: Comparison of Round Robin and Ticket Scheduling after 100,000 task executions

Task	**RR**	**TS (33, 33, 34)**	**TS (60, 30, 10)**	**TS (89, 10, 1)**
Task 1	36.866 (37%)	37.877 (38%)	68.737 (69%)	89.299 (89%)
Task 2	29.494 (29%)	31.202 (31%)	19.045 (19%)	10.131 (10%)
Task 3	33.640 (34%)	30.921 (31%)	12.218 (12%)	570 (1%)
Running Time:	194ms	201ms	221ms	204ms
Context Switch Time:	193ms	200ms	219ms	203ms

Table 3: Comparison of Round Robin and Ticket Scheduling after 1,000,000 task executions

Task	**RR**	**TS (33, 33, 34)**	**TS (60, 30, 10)**	**TS (89, 10, 1)**
Task 1	358.745 (36%)	360.297 (36%)	649.925 (65%)	915.966 (91.6%)
Task 2	278.440 (28%)	277.808 (28%)	242.865 (24%)	76.623 (7.7%)
Task 3	362.815 (36%)	361.895 (36%)	107.210 (11%)	7.411 (0.7%)
Running Time:	2.444ms	2.348ms	2.485ms	2.587ms
Context Switch Time:	2.443ms	2.347ms	2.484ms	2.585ms

Regarding the running and context switch times, it is notable that the running time is predominantly composed by the context switch time. The similarity between the two arises because the tasks do not execute any operations.

4.1.2 Context Switch Time

Another observation is that the context switch time can vary within the same implementation of ticket scheduling based on the position of tasks within the process list.

If the task with the highest number of tickets is positioned at the beginning of the process list, this implies that the winner of the majority of ticket extractions can be obtained with the first access to the list. In contrast, if it was positioned at the end of the list, this would mean having to traverse the entire list to obtain the winner.

From the table below, it is shown that in the case of executing 3 million tasks, when the first task possesses 98% of the tickets, the total context switch time amounts to approximately 7 seconds. Conversely, in the opposite scenario where it is necessary to traverse the entire list in 98% of cases, two seconds more will be required.

Table 4: Comparison of Round Robin and Ticket Scheduling after 3,000,000 task executions

Task	**RR**	**TS (98, 1, 1)**	**TS (1, 1, 98)**
Task 1	1.065.460 (35.5%)	2.949.580 (98.3%)	37.969 (1.3%)
Task 2	860.132 (28.7%)	19.299 (0.6%)	24.133 (0.8%)
Task 3	1.074.408 (35.8%)	31.121 (1.0%)	2.937.898 (97.9%)
Running Time:	7.115ms	7.070ms	9.223ms
Context Switch Time:	7.114ms	7.068ms	9.221ms

To enhance this process's efficiency, it is generally advisable to organize the list in sorted order, arranging tasks from the highest number of tickets to the lowest. This sorting doesn't impact the algorithm's correctness, but it typically minimizes the number of list iterations, particularly when a few processes hold the majority of the tickets.