FreeRTOS: Rate Monotonic Scheduling

Computer Architectures and Operating Systems Project (Track 1.2: HacIOSsim)

Group 26: Merico Michele Marino Alberto Maniero Edoardo Seidita Nicola

July 15, 2024

Politecnico di Torino

Contents

Rate Monotonic Scheduling

RM Task creation

New functions

Implementation of RM

Starting the Scheduler & Switching Context

Simulation of RM Task

Performance comparison

Rate Monotonic Scheduling

Rate Monotonic (RM) Scheduling

Rate Monotonic (RM) Scheduling is a **fixed-priority algorithm** used in real-time systems. In RM, each task is assigned a static priority at compilation time, which remains constant throughout its execution.

The scheduler is **preemptive**, so it can interrupt the execution of a currently running task to start another one that has a higher priority.

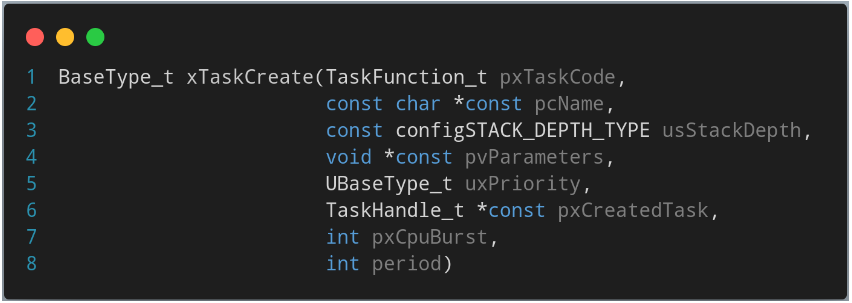
In RM:

- Shorter period → higher priority
- Longer period → lower priority

This ensures that tasks with more frequent execution requirements are prioritized over those with longer periods.

RM Task creation

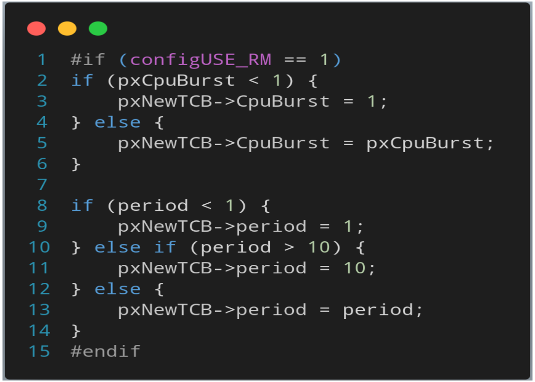
RM Task definintion



```
BaseType_t xTaskCreate(TaskFunction_t pxTaskCode,
                       const char *const pcName,
                       const configSTACK_DEPTH_TYPE usStackDepth,
                       void *const pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *const pxCreatedTask,
                       int pxCpuBurst,
                       int period)
```

RM Task creation

Define new fields in the task to store the estimated Cpu Burst of the task and its Period. The period is expressed in tenth of second and limited to ten only for debug purposes.

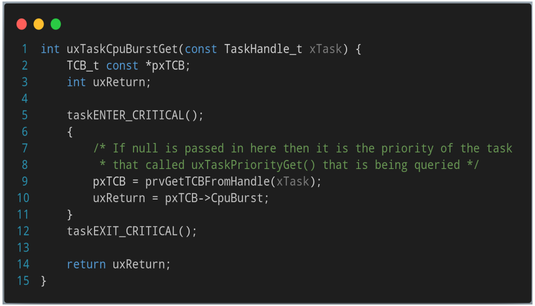


```
#if (configUSE_RM == 1)
if (pxCpuBurst < 1) {
    pxNewTCB->CpuBurst = 1;
} else {
    pxNewTCB->CpuBurst = pxCpuBurst;
}

if (period < 1) {
    pxNewTCB->period = 1;
} else if (period > 10) {
    pxNewTCB->period = 10;
} else {
    pxNewTCB->period = period;
}
#endif
```

New functions

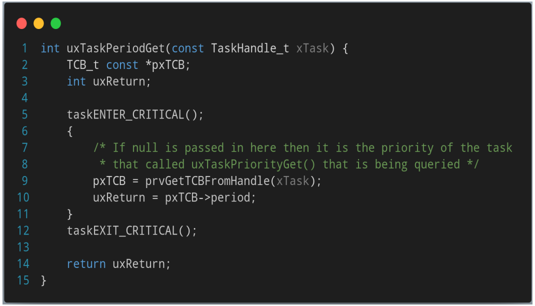
New functions

Retrieve the Cpu Burst of a specified task.



```c
int uxTaskCpuBurstGet(const TaskHandle_t xTask) {
    TCB_t const *pxTCB;
    int uxReturn;

    taskENTER_CRITICAL();
    {
        /* If null is passed in here then it is the priority of the task
         * that called uxTaskPriorityGet() that is being queried */
        pxTCB = prvGetTCBFromHandle(xTask);
        uxReturn = pxTCB->CpuBurst;
    }
    taskEXIT_CRITICAL();

    return uxReturn;
}
```

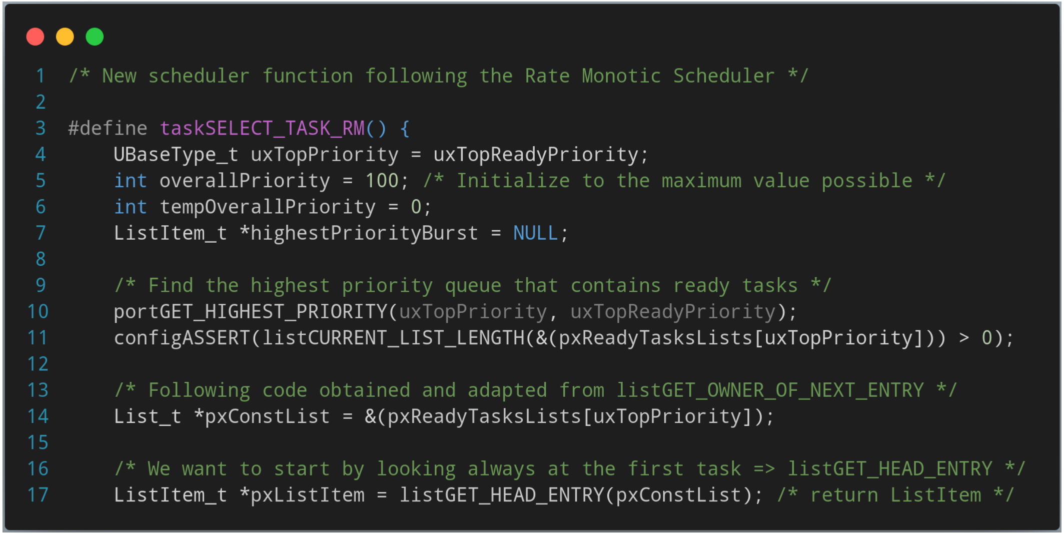
Retrieve the Period of a specified task.



```c
int uxTaskPeriodGet(const TaskHandle_t xTask) {
    TCB_t const *pxTCB;
    int uxReturn;

    taskENTER_CRITICAL();
    {
        /* If null is passed in here then it is the priority of the task
         * that called uxTaskPriorityGet() that is being queried */
        pxTCB = prvGetTCBFromHandle(xTask);
        uxReturn = pxTCB->period;
    }
    taskEXIT_CRITICAL();

    return uxReturn;
}
```

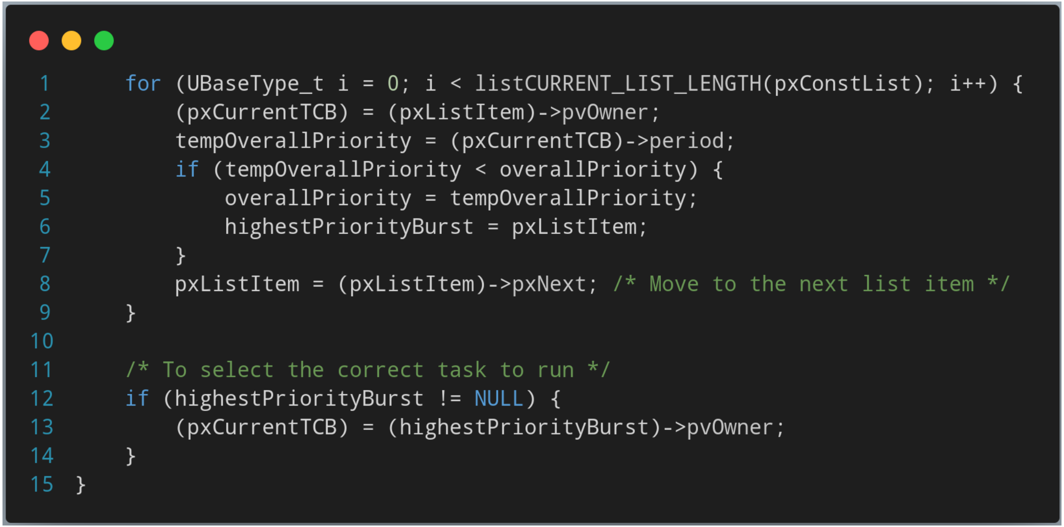
Implementation of RM

RM Select Function - Part 1



```c
/* New scheduler function following the Rate Monotic Scheduler */

#define taskSELECT_TASK_RM() {
    UBaseType_t uxTopPriority = uxTopReadyPriority;
    int overallPriority = 100; /* Initialize to the maximum value possible */
    int tempOverallPriority = 0;
    ListItem_t *highestPriorityBurst = NULL;

    /* Find the highest priority queue that contains ready tasks */
    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 */
    ListItem_t *pxListItem = listGET_HEAD_ENTRY(pxConstList); /* return ListItem */
```

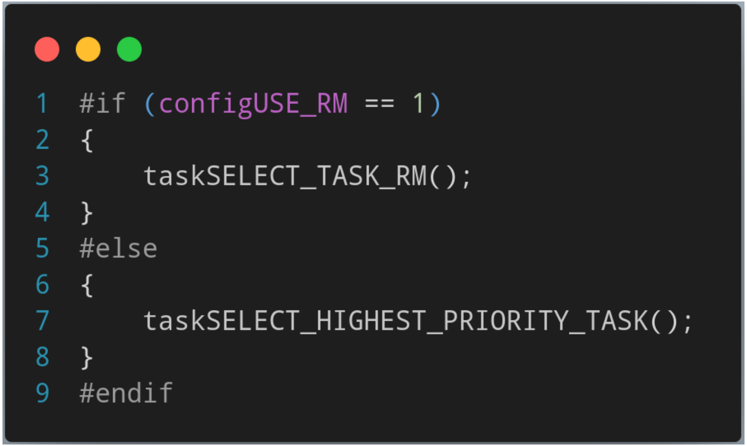
RM Select Function - Part 2



```c
    for (UBaseType_t i = 0; i < listCURRENT_LIST_LENGTH(pxConstList); i++) {
        (pxCurrentTCB) = (pxListItem)->pvOwner;
        tempOverallPriority = (pxCurrentTCB)->period;
        if (tempOverallPriority < overallPriority) {
            overallPriority = tempOverallPriority;
            highestPriorityBurst = pxListItem;
        }
        pxListItem = (pxListItem)->pxNext; /* Move to the next list item */
    }

    /* To select the correct task to run */
    if (highestPriorityBurst != NULL) {
        (pxCurrentTCB) = (highestPriorityBurst)->pvOwner;
    }
}
```

Starting the Scheduler & Switching Context

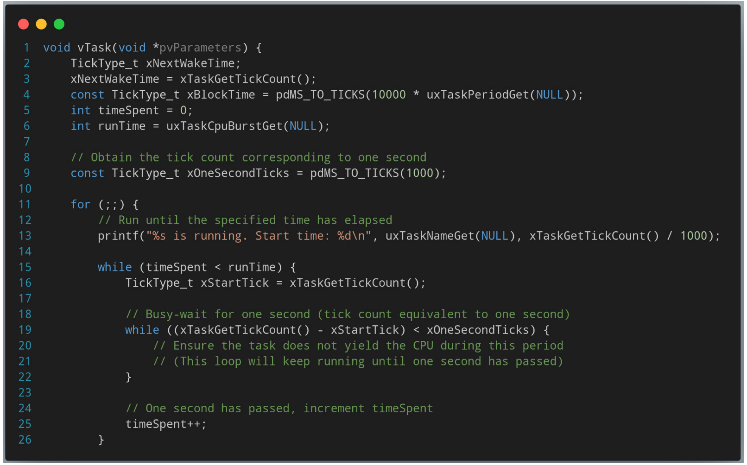
Starting the Scheduler & Switching Context



```
#if (configUSE_RM == 1)
{
    taskSELECT_TASK_RM();
}
#else
{
    taskSELECT_HIGHEST_PRIORITY_TASK();
}
#endif
```

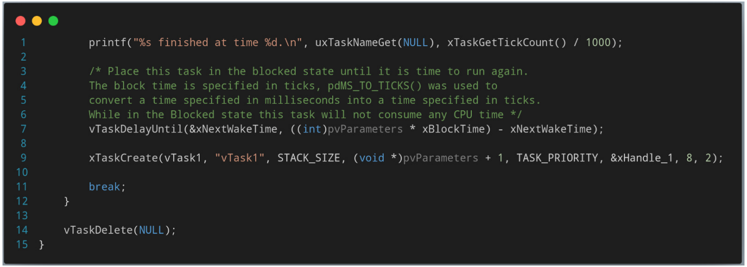
Simulation of RM Task

How RM Tasks are simulated - Part 1



```c
void vTask(void *pvParameters) {
    TickType_t xNextWakeTime;
    xNextWakeTime = xTaskGetTickCount();
    const TickType_t xBlockTime = pdMS_TO_TICKS(10000 * uxTaskPeriodGet(NULL));
    int timeSpent = 0;
    int runTime = uxTaskCpuBurstGet(NULL);

    // Obtain the tick count corresponding to one second
    const TickType_t xOneSecondTicks = pdMS_TO_TICKS(1000);

    for (;;) {
        // Run until the specified time has elapsed
        printf("%s is running. Start time: %d\n", uxTaskNameGet(NULL), xTaskGetTickCount() / 1000);

        while (timeSpent < runTime) {
            TickType_t xStartTick = xTaskGetTickCount();

            // Busy-wait for one second (tick count equivalent to one second)
            while ((xTaskGetTickCount() - xStartTick) < xOneSecondTicks) {
                // Ensure the task does not yield the CPU during this period
                // (This loop will keep running until one second has passed)
            }

            // One second has passed, increment timeSpent
            timeSpent++;
        }
```

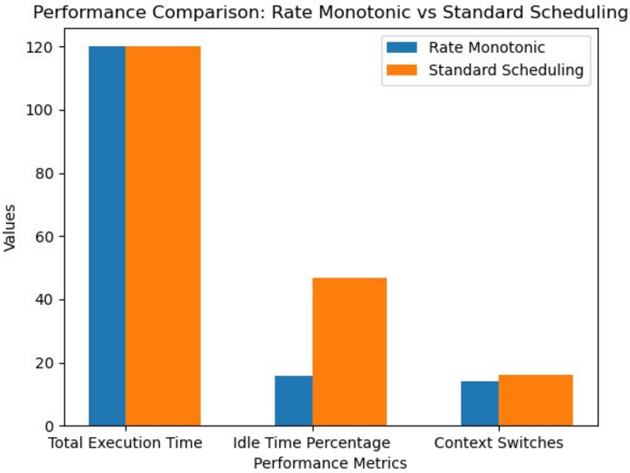
How RM Tasks are simulated - Part 2



```c
        printf("%s finished at time %d.\n", uxTaskNameGet(NULL), xTaskGetTickCount() / 1000);

        /* Place this task in the blocked state until it is time to run again.
        The block time is specified in ticks, pdMS_TO_TICKS() was used to
        convert a time specified in milliseconds into a time specified in ticks.
        While in the Blocked state this task will not consume any CPU time */
        vTaskDelayUntil(&xNextWakeTime, ((int)pvParameters * xBlockTime) - xNextWakeTime);

        xTaskCreate(vTask1, "vTask1", STACK_SIZE, (void *)pvParameters + 1, TASK_PRIORITY, &xHandle_1, 8, 2);

        break;
    }

    vTaskDelete(NULL);
}
```

Performance comparison

RM scheduling VS Standard FreeRTOS Scheduling



Thanks for your attention!