IN4343 Real-Time System Lab 0 - uCOS III Survey

Group 24

Yuhang Tian
5219728

Mingyu Gao
5216281

March 22, 2021

1 Introduction to μC/OS-III

μC/OS-III is a real-time kernel featured with scalability, ROMable and preemptive. As a third-generation kernel, it can manage unlimited tasks and offer resource management, synchronization, inter-task communication, and more. Our survey on μC/OS-III is based on the micro-controller, STM32.
This section will first show the kernel architecture. After that, it will give an instance of how to create a task. Then, it will present the types of tasks that μC/OS-III supports with its mechanism. Finally, it will demonstrate the creation of a new driver.

1.1 Kernel Architecture

Normally, a single CPU can only execute a single task. The kernel of a real-time system is designed for the scheduling of tasks, which is named "multitasking". μC/OS-III is a preemptive kernel, using Interrupt Service Routines (ISR) to temporally stop a task. During the interruption, it can insert another task, and the original context will be restored after the inserted task is completed. In other words, a real-time kernel is responsible for scheduling and switching the CPU between various tasks. The preemptive kernel architecture has been visualized in figure 1.

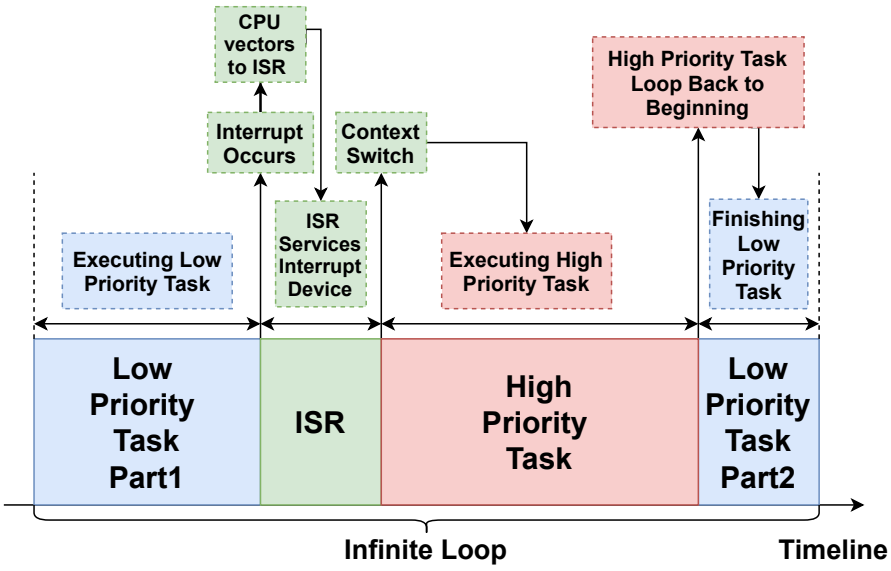


Figure 1: μC/OS-III Preemptive Kernel

Figure 1 shows that, 1) a low-priority task is currently being executed. 2) At the end of an interruption period, ISR will check whether there are high-priority task waiting to be processed, and observe whether there are any tasks. 3) In this situation, it will i) remember the current state of the running task - in order to resume the context - and ii) insert the task with a higher priority. 4) Because of this interruption and incursion, the CPU gives priority to the inserted task. 5) After the CPU completes the higher-priority task, it will resume the original low-priority task according to its memory.

1.2 Task Creation

The task can be created by *OSTaskCreate()* which has been implemented by default. The function takes **13** augments which meanings have been shown in table 1.

Argument with an example	Description
(OS_TCB*) &AppTaskStartTCB	the control block of a task
(CPU_CHAR*) "App Task Start"	a pointer to the task name inside the OS_TCB of the task
(OS_TASK_PTR) AppTaskStart	the address of the task code
(void*) 0	the actual argument that the task receives when it first begins
(OS_PRIO) APP_TASK_START_PRIO	the priority of the task
(CPU_STK*) &AppTaskStartStk[0]	the base address of the stack
(CPU_STK_SIZE) skt_limit	the task's stack that can be used to determine the allowable stack growth of the task
(CPU_STK_SIZE) skt_size	the size of the task's stack in number of CPU_STK elements
(OS_MSG_QTY) 0	the maximum number of messages it can receive
(OS_TICK) 0	the task starts by waiting for one tick to expire before it does anything useful
(void*) 0	extra storage area set by users
(OS_OPT) OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR	the contents of the stack will be cleared when the task is created
(OS_ERR*) &err	a pointer to a variable that will receive an error code

Table 1: Task Creation Function - input arguments

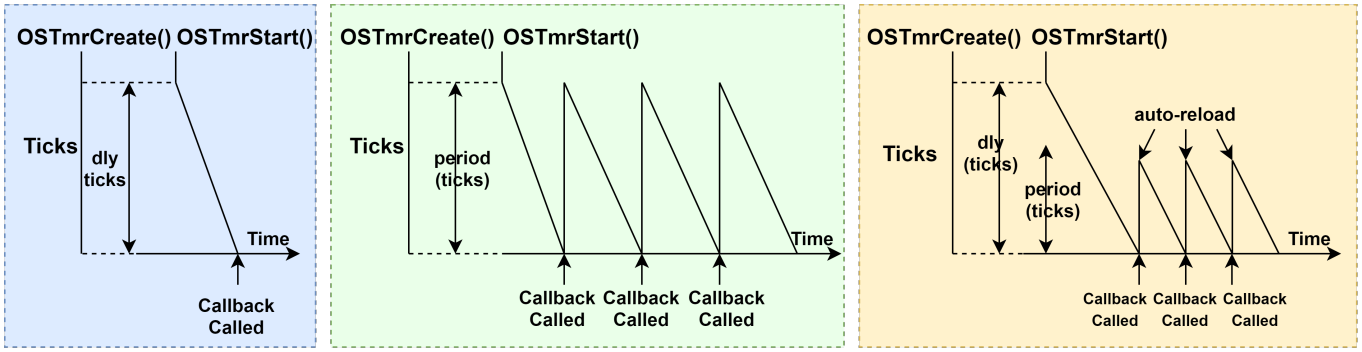
This function has four returned values as shown in table 2.

Return Values	Description
OS_NO_ERROR	whether function succeeds
OS_PRIO_EXIST	whether tasks with this priority already exists
OS_PRIO_INVALID	whether the priority is larger than OS_LOWEST_PRIO
OS_NO_MORE_TCB	whether OS_TCB remains

Table 2: Task Creation Function - Return Values

1.3 Task Type and Mechanism

μC/OS-III can support three types of tasks, **one-shot**, **periodic without delay** and **periodic with initial delay**. All tasks are configured by the timer.Sub-figure 2(a) shows **one-shot timer**, which countdowns from the initial value to zero and calls the callback function until it stops. **A periodic timer** can either have initial or no initial delay which is configured by the keyword *dly*. If *dly* is 0, the timer will have no delay. The period is set by the keyword *period*. In terms of **no initial delay periodic timer**, as shown in sub-figure 2(b), it will also countdown from the initial value to zero in one period. However, when it reaches zero, it will call the callback function to reload the value again and then repeat the identical things for the following periods. Turning to the **initial delay periodic timer**, shown in sub-figure2(c), in the first period, it has a larger value to countdown compared to the following period. This variance causes the delay in executing callback merely for the first period.



(a) One-shot Timer (b) Periodic Timer Without Initial Delay (c) Periodic Timer With Initial Delay

Figure 2: Different Timers For Different Types of Tasks

1.4 Driver Creation

In order to create and manage a new driver, users should allocate its address to the address stack. The address stack can either grow from high memory to low memory or vice versa. In order to control the deriving module, the users utilize *TCB*, which records the information of a task, to allocate the memory for a task. Users use *OS_TCB StartTaskTCB* to generate a module, where *OS_TCB* is a *struct* and cannot be modified. The users use a heap where the dynamic memory can be assigned. The program is running on BBS, Data segment and Code segment. The task structure and code structure diagrams have been shown in figure 3.

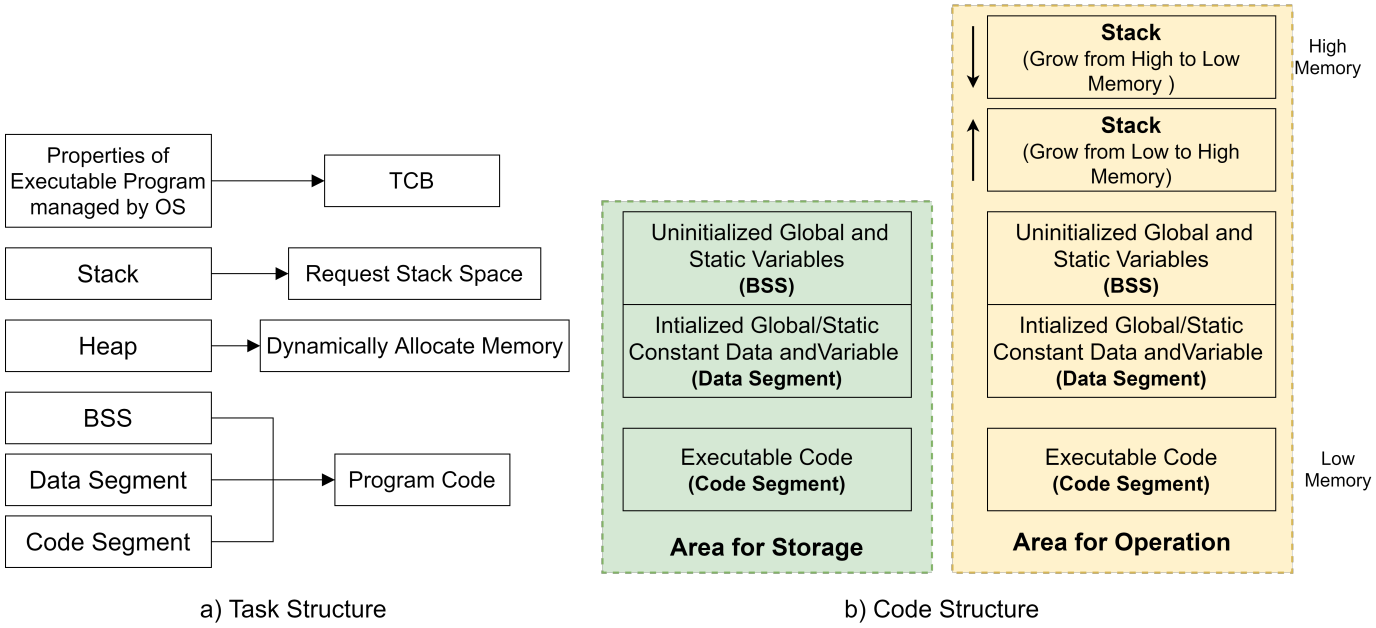


Figure 3: Task and Code Structures

2 Scheduling Policies

μC/OS-III is a preemptive, priority-based kernel. It can support the following scheduling methods:

Table 3: Scheduling Methods

Scheduling Method	Main Feature	Figure
Preemptive Scheduling	the highest priority ready task will receive the CPU	figure 4
Round-Robin	one task is allowed to run for a predetermined amount of time	figure 5

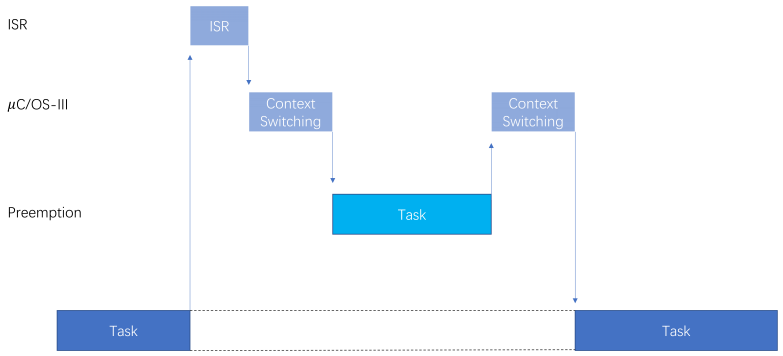


Figure 4: Preemptive Scheduling

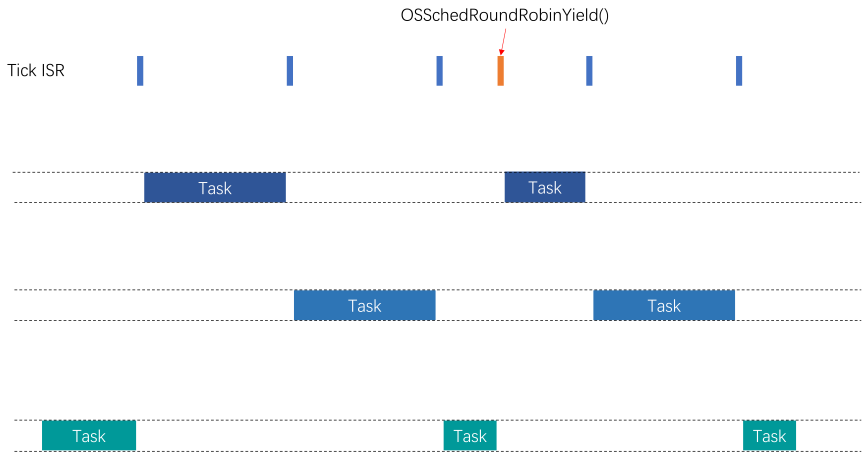


Figure 5: Round-Robin Scheduling

OSSched(), which is called by task-level code, and OSIntExit() called at the end of each ISR.

```
void OSSched (void) {
    Disable interrupts;
    if (OSIntNestingCtr > 0) {
    return;
    }
    if (OSSchedLockNestingCtr > 0) {
    return;
    }
    Get highest priority ready;
    Get pointer to OS_TCB of next highest priority task;
    if (OSTCBNHighRdyPtr != OSTCBCurPtr) {
    Perform task level context switch;
    }
    Enable interrupts;
}
```

```
void OSSched (void) {
    void OSIntExit (void) {
        if (OSIntNestingCtr == 0) {
        return;
        }
        OSIntNestingCtr--;
        if (OSIntNestingCtr > 0) {
        return;
        }
        if (OSSchedLockNestingCtr > 0) {
        return;
        }
        Get highest priority ready;
        Get pointer to OS_TCB of next highest priority task;
        if (OSTCBHighRdyPtr != OSTCBCurPtr) {
        Perform ISR level context switch;
        }
    }
}
```

3 Tracing and Debugging

A μC/OS-III application can enable a built-in statistics task, which collects information about the processor usage of all tasks in the system. Micrium also provides three powerful monitoring tool, μ**C/Probe**,**Tracealyzer** and **SEGGER**, allowing to inspect the state of any variable, memory location, and I/O port in a μC/OS-III enabled application during run time. We also introduce a tool **Grasp** from paper [2].

3.1 Trace Steps

The kernel trace can be simplified as three steps: *trace*, *trace visualization* and *trace measurement*.

3.1.1 Trace

Two main tracing approaches are instrumentation and sampling. The former inserts code at certain positions of the system(such as the top of particular method calls), and records key events at run time. The latter one analyzes a system by a pro-filer during run time rather than change the code. Current mainstream is instrumentation tracing where a recorder component generates a trace file that later serves as input for the visualization application.

3.1.2 Trace Visualization

Tracking contains a large amounts of data, which requires filtering mechanisms to extract specific information. So that users can display events they are interested in. A trace can be visualized in different ways, showing task execution on timeline or the contribution of one task to current processor load. All existing tracing tools can visualize only single level scheduling.

3.1.3 Trace Measurement

Measurements are the metrics to analyze the kernel scheduling, always contains execution time, response time, the best, the worst and average case of jobs.

3.2 Tools Introduction

Here we compare 4 trace tools for μ COS-III

Tool	Tracealyzer	SEGGER	μC/Probe	Grasp
Features	Horizontal trace view; CPU load graph; User event signal plot; Kernel object utilization	Minimally intrusive; Live analysis of captured data; Displays tasks in priority order; records 1 million events	display at runtime; graphical indicators and controls; built-in kernel awareness; oscilloscope at high sampling rates	visualize hierarchical systems; provide insight into buffer contents; Easily extend the trace with arbitrary custom events and visualizations

Table 4: Trace Tools for μCOS-III

4 Mutual Exclusion

For μCOS-III, mutual exclusion are merely allowed to be called by tasks not by ISRs. The APIs of mutex are shown in table 5 and their relationships are shown in 7.

Table 5: Mutex API

Function	Operation
OSMutexCreate()	Create a mutex
OSMutexDel()	Delete a mutex
OSMutexPend()	Wait on a mutex
OSMutexPendAbort()	Abort the wait on a mutex
OSMutexPost()	Release a mutex

In order to demonstrate the mechanisms, we use an example shown in figure 6 below.

In figure 6, step (2) represents that task L requires a mutex so as to access the shared resource. However, task H, owning a higher priority, preempts task L, so task L is suspended by the kernel. In this situation, μC/OS-III increase the priority of task L to the same priority as task H to allow task L to finish with the resource and prevent task L from being preempted by medium-priority tasks. Hence, In step (7),

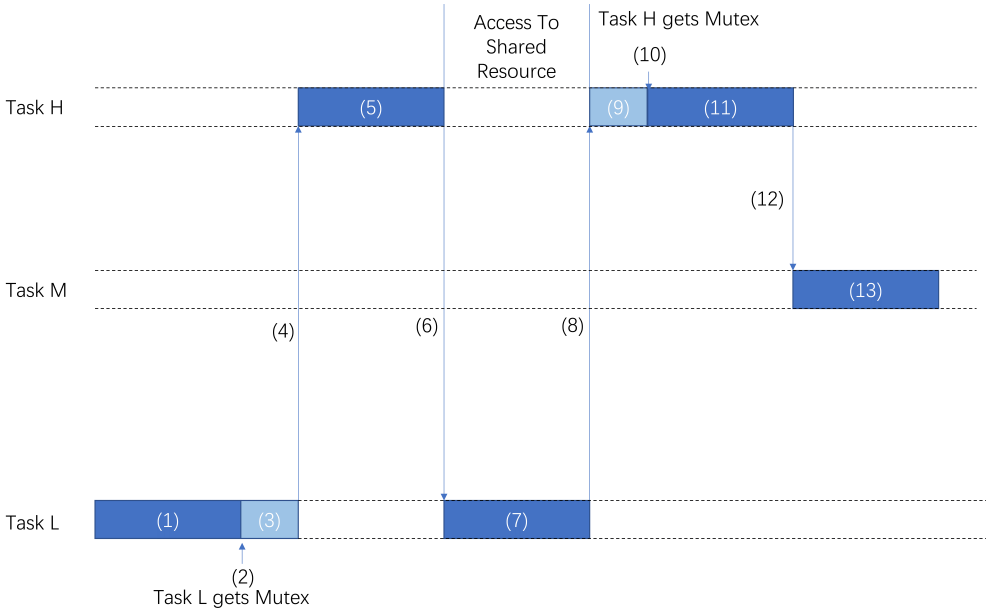


Figure 6: Mutex Mechanism

task L continues working with the resource and releases the mutex after finishing. After that, in step (8) μC/OS-III notices that task L was raised in priority and thus lowers task L to its original priority. In step (9), task H now has the mutex and can access the shared resource.

A semaphore can be used instead of a mutex if none of the tasks competing for the shared resource have deadlines to be satisfied. However, if there are deadlines to meet, you should use a mutex prior to accessing shared resources. Semaphores are subject to unbounded priority inversions, while mutex are not.

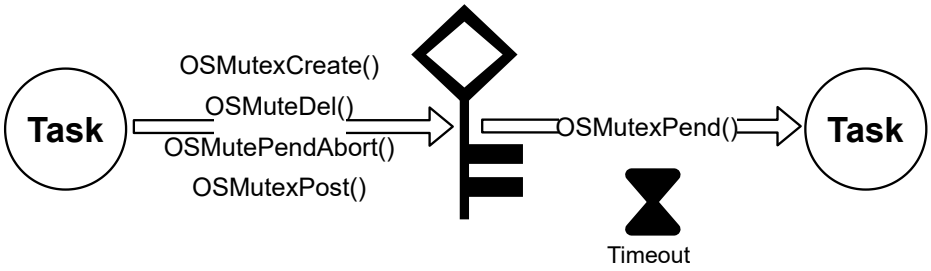


Figure 7: Mutex Mechanism

References

[1] Jean J. Labrosse, 2011, *μC/OS-III: The Real-Time Kernel For the STM32 ARM Cortex-M3*, Micrim Press.

[2] Mike Holenderski, Martijn M.H.P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien, 2010, *Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems*.