Tasks and Task Management

INTRODUCTION

A task is one of the main building blocks of an RTOS application. It is an independent functional unit with its own context, essentially a small independent program. An RTOS application usually consists of several tasks, which don't know about the existence of the other tasks. It's up to the application designer to let them share data or wait for each other through other RTOS components, like queues.

TASKS

The data structure, containing the data defining the task, is called the Task Control Block (TCB). The TCB may contain information like the stack pointer, stack size, MPU settings, priority of the task, state of the task etc. It helps the kernel manage the tasks while, for example, switching context.

The task function is what defines its functionality. It has an entry point where everything that must be done once can be put, like initializing local variables. It must not return, that means that after the entry point there will be a kind of infinite loop inside of the task function. To stop the execution of the task it can be deleted via API calls. It is also possible for several tasks to use the same task function implementation. The task function may also take parameters which are defined at the time of the creation of the task.

As each task is an independent functional unit, each needs a separate execution environment. Therefore, each task has its own stack, where local data like variables, function parameters or function return addresses can be saved. If a task is switched out by the scheduler, its current context, like processor registers etc., will be saved on the task's stack and the context of the swapped in task will in turn be restored from the new task's stack.

A task can be assigned a priority value, which is used by the scheduler to determine which task should be executed if several tasks are ready to run.

As mentioned above, each task has a state. The task state can be divided into two groups:

1. Running: The tasks function is currently being processed.

2. Not running:

- Ready: The task is ready to be executed but another task with a higher or the same priority is running.

- Blocked: The task is waiting for an event, for example a certain time has passed or data in a queue is available. This state is used so that no processing time is used (not polling the events) while waiting. If the event does not happen within a certain time frame, a timeout may be generated and the task can go to the ready state but might want to handle the timeout condition.

- Suspended: The task was suspended by the application via an API call (vTaskSuspend) and is therefore not using any processing time. It stays in this state until it is resumed via another API call (vTaskResume).

TASK MANAGEMENT

An RTOS application usually consists of several tasks. In a processor with a single core, only one task can run at any time and therefore all the tasks must share the available processing time. The scheduler is used to decide which task should be in the running state. The way it decides which task to set to the running state is by calling the scheduling policy. It also defines when a context switch should be performed.

PRIORITY-BASED PREEMPTIVE SCHEDULING

The scheduler ensures that the task in the ready or running state with the highest priority is running. Therefore, as soon as a higher priority task than the running task is entering the ready state, the lower priority running task is preempted and swapped out.

TIME-SLICING

If several tasks, in the ready and running state, share the same priority and are at the highest priority in these states, time-slicing is used to share the processing time between these tasks. A time-slice is defined as the time between two tick interrupts of the RTOS. At each tick interrupt, the scheduler selects another one of those tasks from the ready list to be swapped in.

EXAMPLES

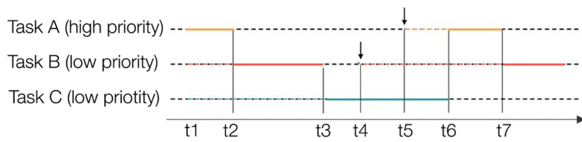

Figure 1: Cooperative scheduling

Figure 1 shows an example of cooperative scheduling. Assume the application consists of three tasks. Task A has the highest priority and Task B and C have the same lower priority. At t1, Task A is in the running state until t2 where it is blocked because it is waiting for an external event. Both Task B and Task C are in the ready state, so at this point the scheduler can either choose Task B or Task C to go into running state (here Task B is chosen). Task B is now running until t3, where it yields by calling the API function. As Task A is still blocked and Task C is the only task in the ready state, the scheduler chooses Task C to switch to the running state. At t4, Task B enters the ready state again. At t5 the event for Task A to return to the ready state occurs but even if Task A is of the highest priority, it must wait for Task C to actively yield or change its state. That is happening at t6 where the scheduler selects Task A to switch from the ready state to running, as it is the highest priority task in the ready state. At t7, Task A again blocks and the scheduler chooses Task B to go to the running state.

When using this scheduling policy tasks must include a point where a task switch will be performed, or else the other tasks will not get enough processing time (task starvation). An advantage is that the time when the task is switched is determined by the task itself. That means it is easier to ensure that the task is done accessing shared resources that must not be accessed by different tasks at the same time e.g., serial ports.

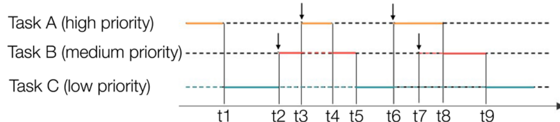

Figure 2: Priority-based preemptive scheduling

Figure 2 shows an example of priority-based preemptive scheduling. The application consists of three tasks with Task A having the highest priority, Task B having a medium priority and Task C at the lowest priority. At t1, Task A enters the blocked state and the scheduler selects Task C to go to the running state as it is the only task in the ready state. At t2, Task B enters the ready state and Task C is immediately preempted by the scheduler due to the higher priority of Task B. Now Task B is running and Task C enters the ready state. At t3 Task A enters the ready state, due to an event it was waiting for, and the scheduler in turn preempts Task B as Task A has the highest priority. Task A is running while Task B and Task C are both in the ready state until t4, where Task A again blocks to wait on an external event. This leads to a context switch to task B, which is the highest priority task in the ready state. At t5 Task B also enters the blocked state and Task C, which is the only task in the ready state, is selected by the scheduler. At t6, Task A unblocks and so the scheduler preempts Task C to select Task A for running. At t7 Task B enters the ready state but as Task A, which is currently running, has a higher priority it remains in the ready state. At t8, Task A blocks and Task B can now enter the running state. At t9, Task B blocks and therefore Task C can now run.

The example shows that in priority-based preemptive scheduling, high priority tasks immediately get processing time when they need it; this makes the application very responsive.

TASKS & SAFE**RTOS**®

Figure 3 shows an example where priority-based preemptive scheduling is combined with time-slicing. This scheduling policy is typically used in SAFE**RTOS**. The application has three tasks, Task A at the highest priority and Task B and C at the same lower priority. At t1 Task A blocks and waits for an event to happen, therefore Task B or C can go to the running state as both are in the ready state and at the same priority. In this example the scheduler chooses Task B. At the next tick interrupt, the scheduler swaps in Task C as time-slicing is active and Task C is at the same priority as Task B. This switching between both tasks continues at each tick interrupt until t2, where the event which Task A is waiting for happens. Task B is preempted due to a lower priority and Task A runs until t3 where it blocks again. The scheduler now chooses Task C from the ready state list and it runs until the next tick interrupt.

This scheduling policy keeps the advantage of high responsiveness of high priority tasks and shares processing time between tasks of equal priority.

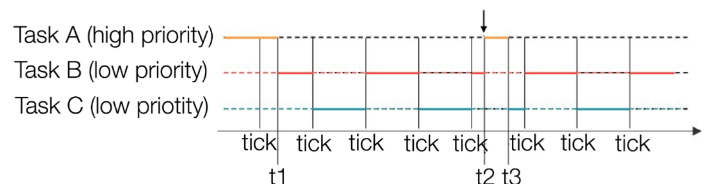

Figure 3: Priority-based preemptive scheduling with time-slicing

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

Copyright 2022 WITTENSTEIN high integrity systems Ltd. V1.0. Correct at time of publishing.