AliOS 任务上下文切换原理分析

前言

一般来讲，MCU 相关的操作系统都具有最基本的多任务管理功能。多任务管理功能包含任务的上下文切换，因其涉及到特定寄存器的操作，都是使用汇编代码开发，相应代码由软件厂商提供。

在 STM32 技术人员的实际支持工作中，例如 IDE 移植，可能需要读懂和修改这些汇编代码。本文就是从这一需求出发，描述 AliOS 操作系统里的任务上下文切换的基本原理。读者在明白了上下文切换原理后，去阅读和修改任何 MCU 操作系统的关于任务上下文切换的汇编代码就不会是个难题。本文包含的代码引用基于 STM32F4（ARM Cortex-M4 内核）芯片上的实现。

上下文切换的源动力

带操作系统的应用程序，上下文切换的逻辑在 **PendSV** 中断处理函数中完成。 为什么是 **PendSV** 中断，而不是其它的中断? ARM 技术文档里提到，**PendSV** 设计为上下文切换服务的中断请求，优先级最低。任务切换的策略可以根据时间片和优先级来制定。切换时机可由系统滴答时钟（**SysTick**）触发，或者由操作系统根据已有策略来触发 **PendSV** 中断。

以 AliOS 的系统时钟中断处理函数为例，则可以看到有如下代码：

```
void SysTick_Handler(void)
{
  HAL_IncTick();
  if (0 == HAL_GetTick() % 10) {
    krhino_intrpt_enter();
    krhino_tick_proc();
    krhino_intrpt_exit();
  }
  //HAL_SYSTICK_IRQHandler();
}
```

由系统时钟触发

其中 krhino_intrpt_exit 则会通过 cpu_intrpt_switch 触发 **PendSV** 中断，然后 **PendSV** 中断处理函数负责完成上下文切换。cpu_intrpt_switch:

```
  LDR     R0, =SCB_ICSR
  LDR     R1, =ICSR_PENDSVSET
  STR     R1, [R0]
  BX      LR
```

在 AliOS 中有很多地方，根据系统的需要，还可直接调用 cpu_task_switch 来触发 **PendSV** 中断。这一部分是纯软件的设计，不再列举。

```
cpu_task_switch:
  LDR     R0, =SCB_ICSR
  LDR     R1, =ICSR_PENDSVSET
  STR     R1, [R0]
  BX      LR
```

由操作系统自己的策略触发

上下文切换

上下文切换的汇编代码实现的功能就是把前一个与当前任务相关的寄存器信息进行压栈保存，比如当前栈指针、TCB 信息等，把将要运行的新任务的栈内信息恢复到对应寄存器中为新任务的执行做好准备。这里的上下文就是指各个寄存器信息。

寄存器分类

- **通用寄存器**

R0-R12 是 32 位通用寄存器。

- **堆栈指针 SP**

R13 是堆栈指针寄存器。堆栈分为复位后缺省使用的主堆栈（由 **MSP 指向**）以及用户应用程序使用的任务堆栈（由 **PSP 指向**）。可以通过控制寄存器控制当前正在使用的堆栈。

- **链接寄存器 LR**

R14 是链接寄存器，用在函数或者异常/中断返回。

- **程序计数器 PC**

R15 是程序计数器。它指向当前运行的指令地址。

- **程序状态寄存器 xPSR**

程序状态寄存器包含三个寄存器：应用程序状态寄存器 **APSR**, 中断程序状态寄存器 **IPSR** 和执行程序状态寄存器 **EPSR**。**APSR** 包含上一条指令执行后的条件标志，**IPSR** 包含有中断号，**EPSR** 包含有被中断指令的待执行寄存器信息，诸如 **LDM**、**STM**、**PUSH**、**POP**，或 **IF-THEN** 条件指令的信息以及当前是否运行在 **Thumb** 状态。

- **优先级掩码寄存器 PRIMASK（1 位）**

PRIMASK 置 1 后，系统阻止除 NMI 和 hard fault 外的所有异常/中断。

- **故障掩码寄存器 FAULTMASK（1 位）**

FAULTMASK 置 1 后，系统阻止除 **NMI** 外的 所有异常/中断。

- **基线掩码寄存器 BASEPRI（9 位）**

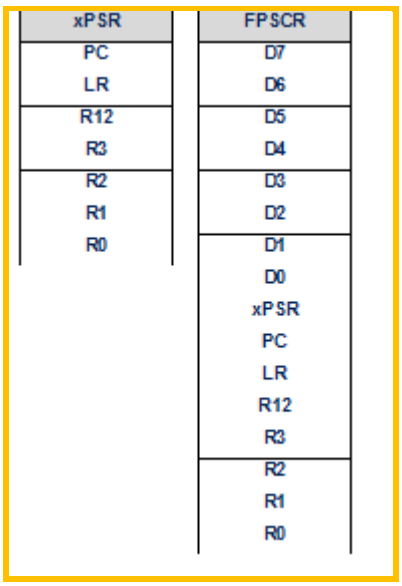
根据在 **BASEPRI** 中的优先级设置，系统阻止优先级小于等于它的所有异常/中断。

- **控制寄存器 CONTROL**

CONTROL 寄存器控制在线程（**THREAD**）模式下所处的特权级别（特权级还是用户级），以及当前系统使用的堆栈。中断处理（**Handler**）模式只能处于特权级，总是使用主堆栈（由 **MSP 指向**）。可以通过读出 **SPSEL** 位来进行验证。线程（**THREAD**）模式可以通过 CONTROL 寄存器切换是使用主堆栈（由 **MSP 指向**），还是任务堆栈（由 **PSP 指向**）。

硬件管理的寄存器

在上下文切换的中断处理中，一部分寄存器由硬件管理。也就是说，在进入 PendSV 中断处理函数时一部分寄存器信息被自动压栈保存，在退出 PendSV 中断时这些寄存器自动被弹出。
根据 ARM 文档，在上下文切换中，不考虑浮点和考虑 32 个单精度或者 16 个双精度浮点寄存器两种情况下，由硬件管理的寄存器列表分别如下：



我们要特别注意主堆栈 **MSP** 以及任务堆栈 **PSP** 在这里的用法。

进入 **PendSV** 中断处理函数，与任务有关的硬件管理的寄存器信息被自动压栈进 **PSP** 中（因为任务是用户应用程序的一部分，因此使用的是任务堆栈）；在退出 **PendSV** 时，是 **PSP** 里的相应内容被自动弹出。读者可以在中断之外查看 **CONTROL** 寄存器的 **SPSEL** 位，值为 1 表示的是当前任务使用任务堆栈（由 **PSP 所指向**）。读者可在 **PendSV** 的汇编代码里看到以下代码，目的是将 **PSP** 里的任务堆栈恢复：

```
/* return stack = PSP */
MSR     PSP, R0
ORR     LR, LR, #0x04
```

但是，**PendSV 本身**和所有其他的中断处理函数一样，使用的是主堆栈（由 **MSP 指向**）。这使我们不用担心中断处理函数是否弄脏了任务的堆栈，比如给任务堆栈添加了不必要的内容。这里值得一提的是，从逻辑上来说，使用一个堆栈指针就已经足够了。

软件管理的寄存器

不考虑浮点，则 R4~R11 和 LR 寄存器需要软件在中断处理函数里进行处理。软件管理的寄存器在压栈时，紧随在硬件管理的寄存器之后。栈是向地址减小的方向生长的。从这个角度来看，软件管理的寄存器在栈里处于更低的地址。若考虑浮点，则需要额外压栈浮点寄存器：D8~D15

在 **PendSV** 相应的压栈代码和弹栈代码如下：

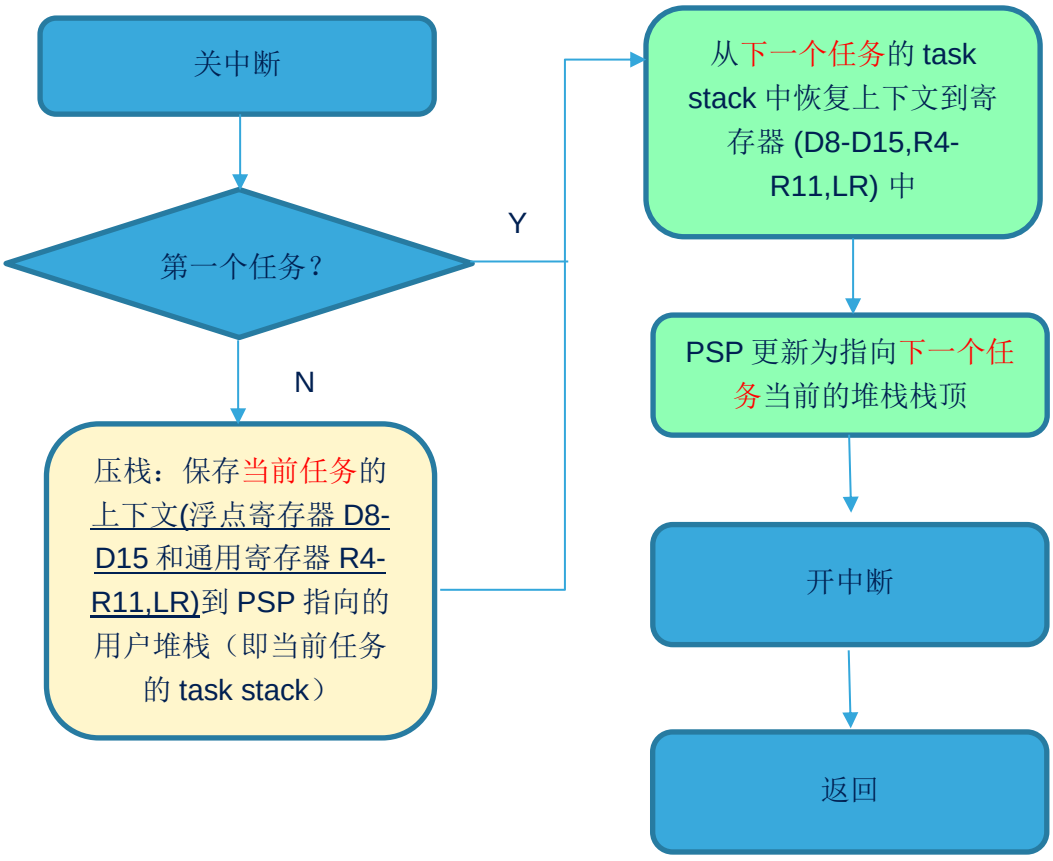
```
;save context
IF      {FPU} != "SoftVFP"
VSTMFD  R0!, {D8 - D15}
ENDIF
SUBS    R0, R0, #0x24
STM     R0, {R4-R11, LR}

;restore context
LDM     R0, {R4-R11, LR}
ADDS    R0, R0, #0x24
IF        {FPU} != "SoftVFP"
```

```
VLDMFD  R0!, {D8 - D15}
ENDIF
```

锐利的读者可能已经注意到，软件和硬件都同时对 LR 进行了压栈。事实上，软件压栈的 LR 包含的是异常返回值 EXC_RETURN，该值可指定异常返回时的堆栈（主堆栈还是用户堆栈）和处理器工作模式（用户模式还是特权模式）。

PendSV 的处理流程

硬件管理的寄存器，不需要在 PendSV 函数里手动进行处理。另外，切换到第一个任务（没有更早的任务），也不需要保存当前任务的的寄存器信息到堆栈。已经保存有寄存器信息的堆栈栈顶指针，一般存在任务的控制结构里；以 AliOS 为例，在进入 PendSV 中断处理前已经将下一个任务的控制结构准备就绪，可以通过 g_preferred_ready_task 进行访问。



小结一下：对于 AliOS 操作系统，每个 task 在创建时用户都会为它指定一段存储区域作为物理上该任务的 task_stack；PSP 永远指向当前任务的 task_stack；在任务切换时，执行 PendSV：切换 PSP，并且把内核的当前状态（寄存器组的值）保存到当前这个马上要被切走的老任务的 task_stack → 保存上下文@入栈，再把马上要被调度的新任务上下文恢复到内核的寄存器中 → 恢复上下文 @ 出栈。

总结

本文简单介绍了 AliOS 的任务上下文切换原理，包括哪些是硬件负责入栈和出栈的寄存器，哪些是软件负责入栈和出栈的寄存器，以及上下文切换的流程。理解 MCU 操作系统的上下文切换原理，有助于阅读操作系统厂商的汇编源代码，进行移植或者其它高级应用。

附录

```
PendSV_Handler
    CPSID   I
    MRS     R0, PSP  // R0 指向当前的任务堆栈栈顶
    ;branch if cpu_first_task_start
    CBZ     R0, _pendsv_handler_nosave

    ;hardware saved R0~R3,R12,LR,PC,xPSR

    ;save context
    IF      {FPU} != "SoftVFP"
    VSTMFD  R0!, {D8 - D15}
    ENDIF

    SUBS    R0, R0, #0x24
    STM     R0, {R4-R11, LR}  // 把当前任务的上下文保存到任务堆栈（栈顶也随之调整）

    ;g_active_task->task_stack = context region
    LDR     R1, =g_active_task
    LDR     R1, [R1]
    STR     R0, [R1]

    bl      krhino_stack_ovf_check

_pendsv_handler_nosave
    LDR     R0, =g_active_task
    LDR     R1, =g_preferred_ready_task
    LDR     R2, [R1]
    STR     R2, [R0]
    ;R0 = g_active_task->task_stack = context region
    LDR     R0, [R2]    // L0 指向新任务控制块中设置的任务堆栈

    ;restore context
    LDM     R0, {R4-R11, LR}
    ADDS    R0, R0, #0x24   // 恢复新任务的上下文（从其任务堆栈中读到内核寄存器组）

    IF     {FPU} != "SoftVFP"
    VLDMFD  R0!, {D8 - D15}
    ENDIF

    ;return stack = PSP
    MSR     PSP, R0       // PSP 指向新任务的堆栈

    ;after exception return: stack = PSP
    ORR     LR, LR, #0x04

    CPSIE   I
    ;hardware restore R0~R3,R12,LR,PC,xPSR
    BX      LR
```

```
    ALIGN
    END
```

LDR R0, [R1]：把以 R1 为地址的存储区内容读到 R0 中　　【Memory → R】右到左
STR R0, [R1]：把 R0 存储到以 R1 为地址的存储区　　【R → Memory】左到右

LDM R0, {R4-R11, LR}：把以 R0 所指的存储区内容读到寄存器组中　　【Memory → Ri】左到右
STM R0, {R4-R11, LR}：把寄存器组的内容保存到 R0 所指的存储区　　【Ri → Memory】右到左

MRS R0, PSP：把状态寄存器（PSP）的值读到 R0　　右到左
MSR PSP, R0：把 R0 写到状态寄存器（PSP）中　　右到左