操作系统实验指导

山东大学网络空间安全学院
王伟嘉

第一次实验

1. 实验环境介绍

1.1 μC/OS-II 简介

μC/OS-II，是一个专门为嵌入式应用设计的实时操作系统。除了一些和处理器相关的代码用汇编编写，其他绝大部分代码使用 C 语言，所以在学习了 C 语言程序设计和计算机组成原理，阅读这个操作系统的代码并不困难。该操作系统具有很好的可扩展性，人们可以在它具有很好的可扩展性，人们可以在这个系统的基础上二次开发很多功能。同时这个系统是开放源码的（不开放源码也很难有机会去学习），但是并不代表这个操作系统是免费的（当然对于学习目的是免费），如果使用这个操作系统去完成某个产品，那么就需要购买昂贵的版权许可证。当然这也从侧面说明了这个操作系统的价值：既然用作商用，至少说明它是一个相当“靠谱”的系统。另外，该系统也被广泛使用在从照相机到航空电子产品的各种应用中。

1.2 ARM Cortex-M3 编程模型

本实验选择 ARM Cortex-M3 的嵌入式微处理器作为硬件环境的处理器，主要是出于以下几个目的：

1. 使用广泛。据 Arm 公布的数据显示，2019 年第四季度（10 到 12 月，3 个月的时间），全球共卖出了 42 亿颗 Cortex-M 系列的芯片，可以证明这个芯片使用的广泛性。
2. 足够简单。一个足够简单的处理器有助于学生专注与操作系统本身。
3. 方便的仿真环境。操作系统是运行在一个裸机上的，所以实验中或者 1)使用一个没有安装任何操作系统的硬件设备，或者 2）用仿真。显然后者使用仿真的方法往往比较方便，毕竟我们更希望能专注与操作系统本身。ARM Cortex-M3 有个非常好的集编辑、编译、仿真的工具链：Keil 软件，本课程的实验基于这个工具链的硬件上仿真器。

1.2.1 ARM Cortex-M3 的寄存器

一个处理器最基本的就是它的寄存器组，这里通过描述寄存器来介绍 ARM Cortex-M3 的

操作方法。ARM Cortex-M3 的寄存器组一共包括统 13 个通用寄存器--R0 到 R12、栈指针寄存器、链接寄存器、程序寄存器、5 个特殊功能寄存器。其中特殊功能寄存器只能由专用的 MSR/MRS 指令访问。如图 1 所示。

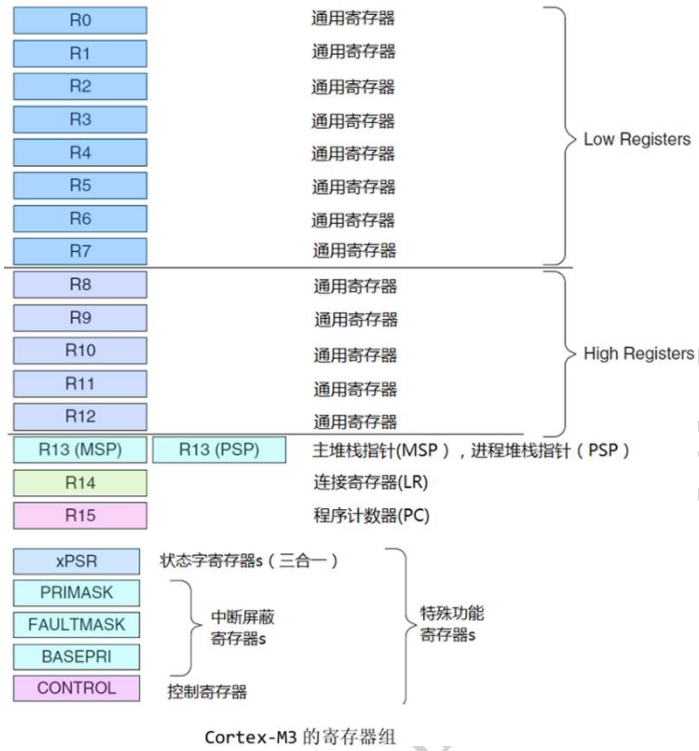


图 2 ARM Cortex-M3 的寄存器组

通用寄存器。这些是正常使用的寄存器。有个要注意的是 R0 到 R7 是支持所有的指令，但是 R8 到 R12 这几个高位寄存器只支持 32 位的指令，过多使用会增大程序大小。正常建议尽可能使用 R0 到 R7。

```
例:
ADD r0, r0, r1
ADD r0, r1
MOV r0, r1
MOV r0, #0x01
LDR   r0, [r1]
（大小写不敏感）
```

程序计数器 PC（R15）。记录当前指令的地址+2 或+4。

```
例:
LOOP
    MOV    PC,   =LOOP
-----------------------
LOOP
    B          LOOP
-----------------------
LOOP
    MOV   R0,   =LOOP
    BX         R0
```

连接寄存器 LR（R14）。用于在调用子程序（函数）时存储返回地址。例如，当你在使用 BL(分支并连接，Branch and Link)指令时，就自动填充 LR 的值。注意 LR 只能存上一次跳转前的地址，如果存在多次嵌套子程序调用，就要利用 PUSH 把 LR 保存到栈中，并在子程序返回之前 POP 出来。

```
例:
main      ;主程序
          BL      function1      ; 使用"分支并连接"指令呼 function1
                                 ;PC=function1，并且 LR 装入下一条指令地址
          …
function1
          …                      ; function1 的代码
          BX      LR             ; 函数返回
```

堆栈指针寄存器 SP（R13）。记录当前堆栈指针，PUSH/POP 指令参考。该寄存器是以下两个内部寄存器复用的，由 CONTROL 的第 1 位决定。

1. 主堆栈指针（MSP）：默认的堆栈指针，它由 OS 内核、中断处理以及所有需要特权访问的应用程序代码来使用的。
2. 进程堆栈指针（PSP），用于常规的应用程序代码（不处于中断处理中时）。

控制寄存器 CONTROL。控制处理器的状态与使用的堆栈。如图 3 所示。

位	功能
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后的缺省值） 1=选择进程堆栈指针 PSP 在线程或基础级（没有在响应异常——译注），可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 **Handler 模式永远都是特权级的。**

图 3 控制寄存器

状态寄存器 xPSR。记录当前处理器的状态。

中断屏蔽寄存器。三个寄存器共同使用屏蔽一定优先级的中断。

1.2.2 ARM Cortex-M3 的工作模式、堆栈指针选择

ARM Cortex-M3 提供两种工作模式：处理模式(handler mode)、线程模式(thread mode)。其中处理模式是中断下的模式，而线程模式是除了中断之外的模式。两种特权(由 CONTROL[0]控制）为特权级（privileged）和非特权级（unprivileged)。其中在特权级下无论是线程

模式还是处理模式，都是核心态。而在非特权级下，处理器处于线程模式时为用户态，而处理器处于处理模式下核心态。可以看出，只有在非特权级下，处理器才有可能提供保护机制。如图 4 所示。

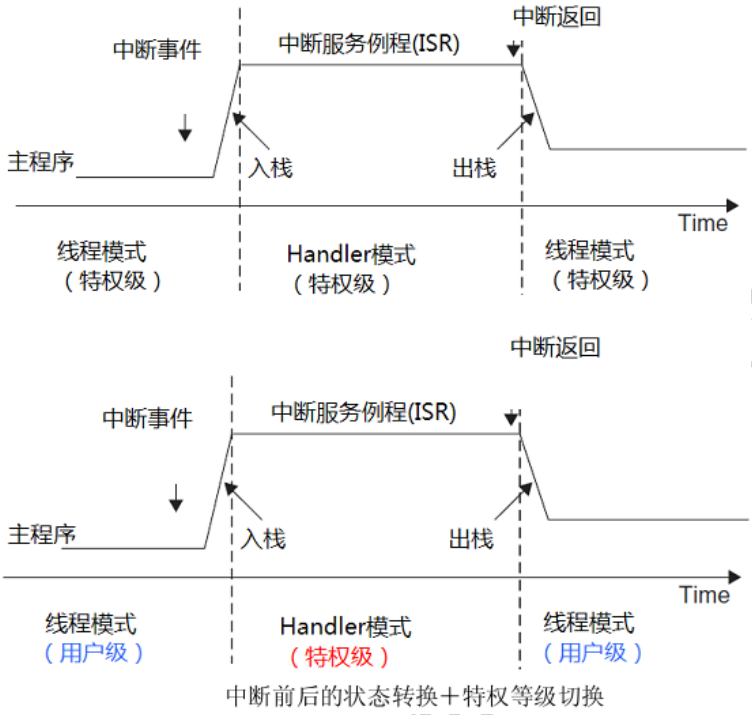


图 4 中断前后的状态与特权等级切换

堆栈指针选择类似。如图 5 所示。

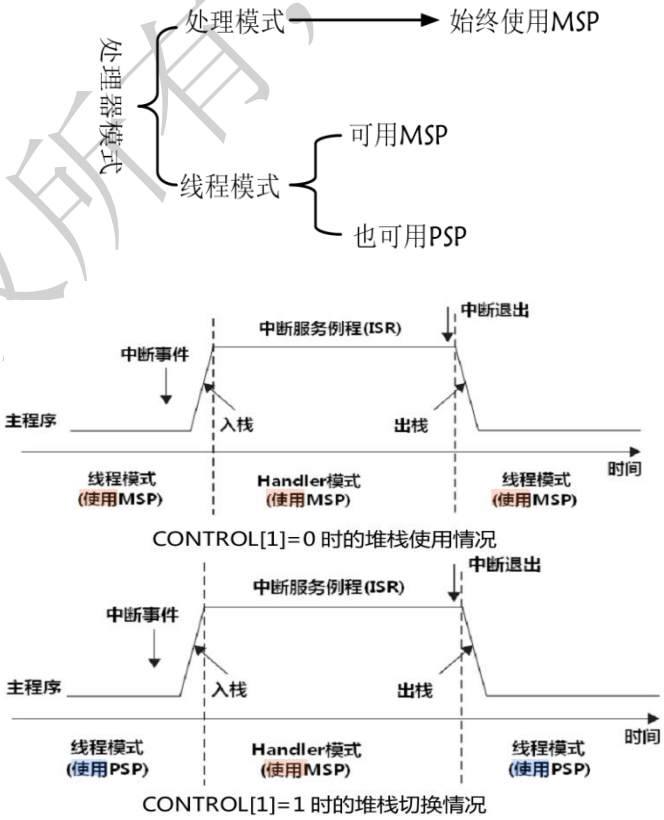


图 5 堆栈指针选择

1.2.3 ARM Cortex-M3 的中断响应

当 Cortex M3 开始响应一个中断时，会自动进行以下三部操作:

1. 入栈：把 8 个寄存器的值压入栈。入栈顺序如图 6 所示。
2. 取向量：从向量表中找出对应的服务程序入口地址。
3. 选择主堆栈指针 MSP，更新连接寄存器 LR，更新程序计数器 PC：其中 L 据中断之前的处理器模式，LR 更新为三个“EXC_RETURN”值，中断返回的时候会根据 LR 的值返回到不同的处理器模式，如图 7 所示。

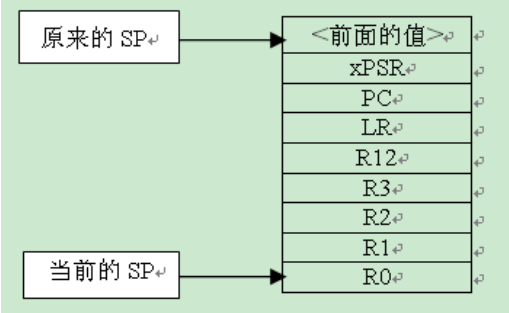


图 6 中断入栈顺序

合法的EXC_RETURN值及其功能

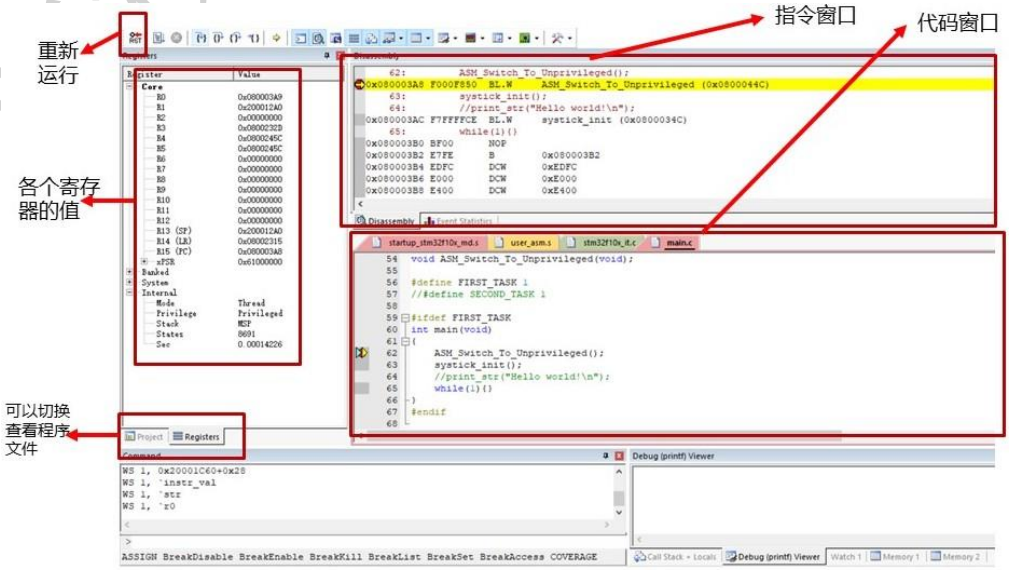
EXC_RETURN 数值	功能
0xFFFF_FFF1	返回handler模式
0xFFFF_FFF9	返回线程模式，并使用主堆栈(SP=MSP)
0xFFFF_FFFD	返回线程模式，并使用线程堆栈(SP=PSP)

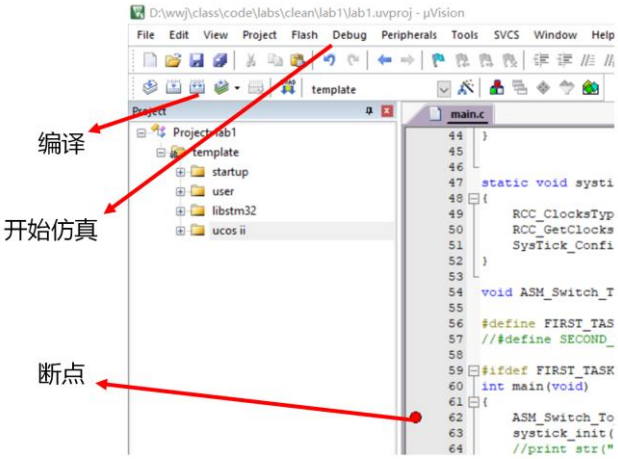
图 7 EXC_RETURN 值以及功能

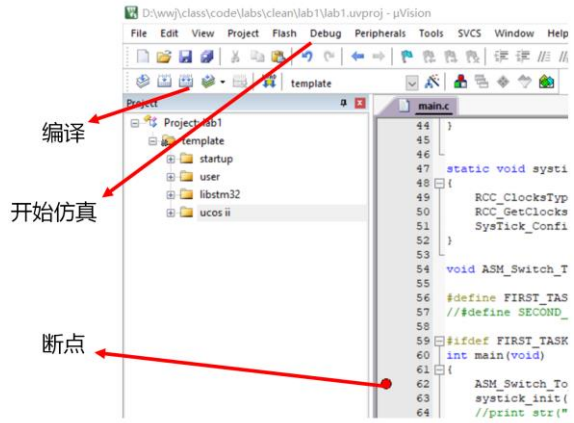
中断返回：首先出栈 8 个寄存器：PC，xPSR，r0，r1，r2，r3，r12，LR，从所选的堆栈中出栈（堆栈指针由 EXC_RETURN 选择），并调整 SP。

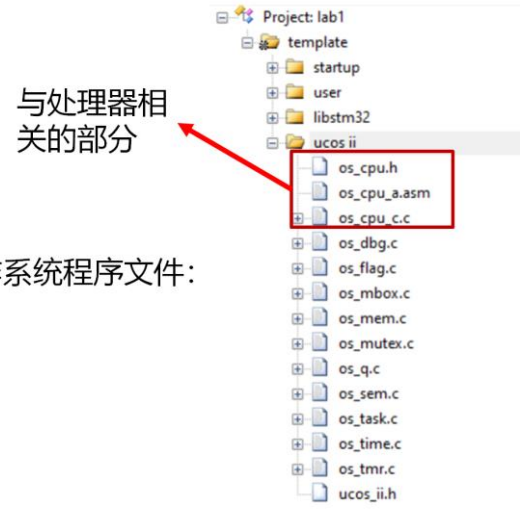
1.3 实验平台介绍

本小节以图片的形式简单介绍 Keil 实验平台与μC/OS-II 操作系统代码结构。

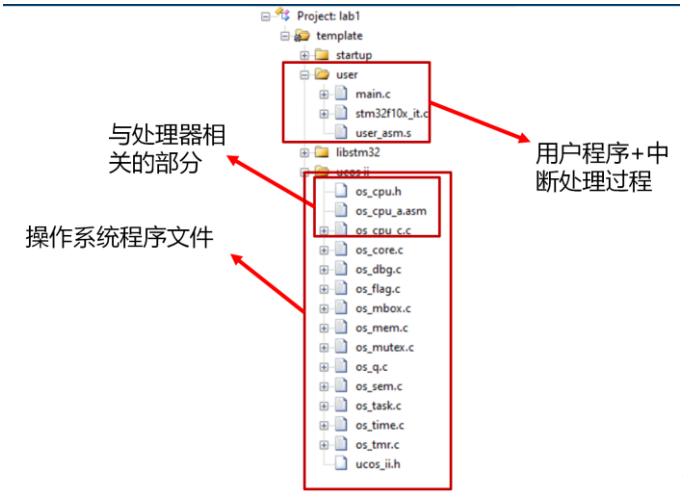


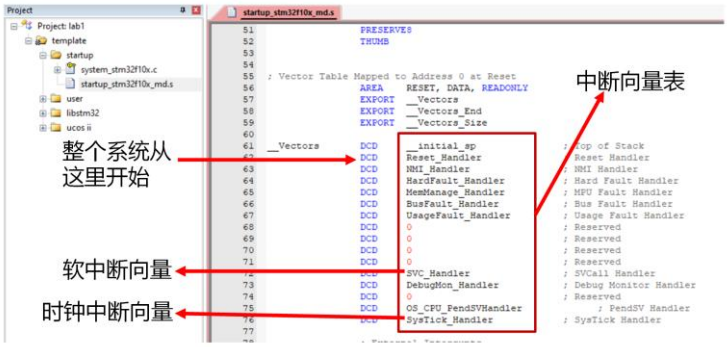


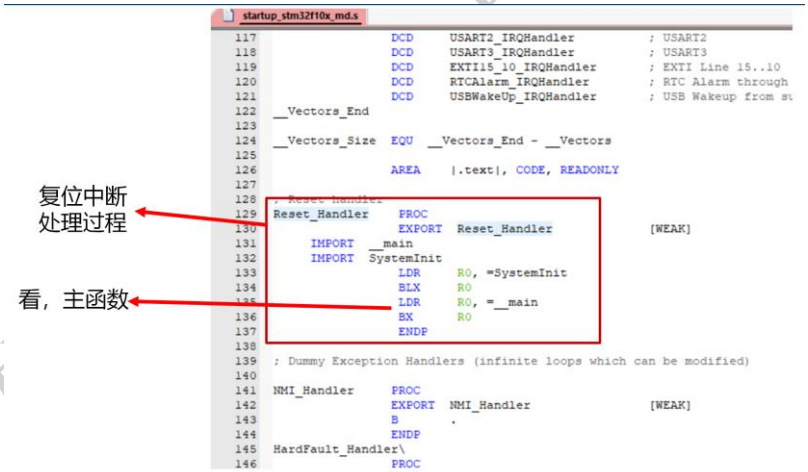


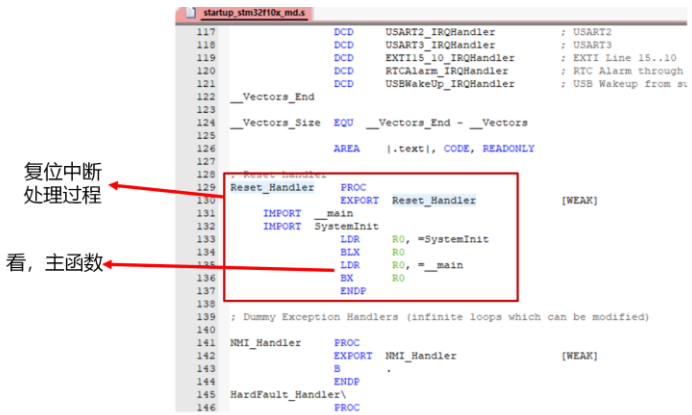


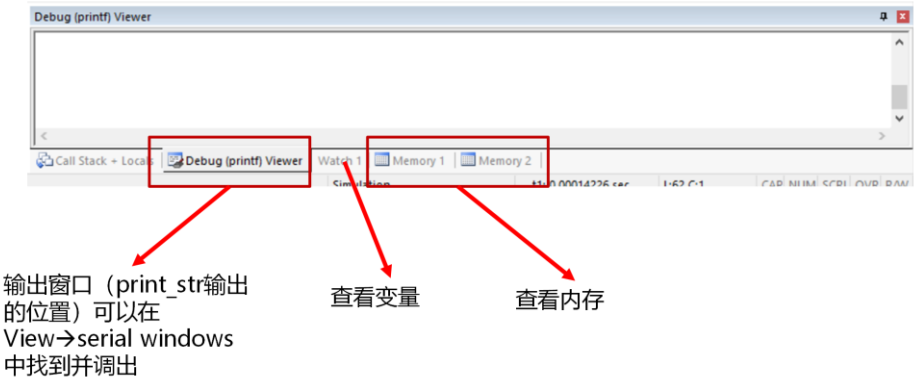
操作系统程序文件：


Project: lab1
template
startup
user
main.c
stm32f10x_it.c
user_asm.s
libstm32
ucos_ii
os_cpu.h
os_cpu_a.asm
os_cpu_c.c
os_core.c
os_dbg.c
os_flag.c
os_mbox.c
os_mem.c
os_mutex.c
os_q.c
os_sem.c
os_task.c
os_time.c
os_tmr.c
ucos_ii.h
与处理器相关的部分
用户程序+中断处理过程
操作系统程序文件


startup_stm32f10x_md.s
中断向量表
整个系统从这里开始
软中断向量
时钟中断向量


startup_stm32f10x_md.s
复位中断处理过程
看，主函数


startup_stm32f10x_md.s
复位中断处理过程
看，主函数



Keil 的安装：先安装 MDK529.EXE，再安装 MDKCM525.EXE，默认安装即可。

2. 任务一

2.1 任务内容

设计以下系统调用，以完成用户态下的时钟初始化和显示输出：

1. 初始化时钟
2. 显示输出

要求：

1. 以上两个系统调用同时存在
2. 中断处理过程尽可能短

2.2 提示

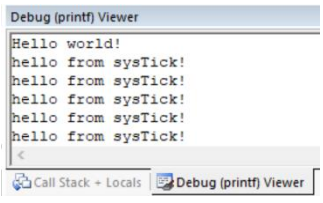
main.c 文件中可以发现以下用来选择当前的任务的定义。

```
89
90  #define FIRST_TASK 1
91  //#define SECOND_TASK 1
92
```

主函数 main()函数如下：

```
int main(void)
{
  //ASM_Switch_To_Unprivileged();
  print_str("Hello world!\n");
  systick_init();
  while(1){}
}
```

点击 rebuild，然后点击 Debug→Start/Stop Debug Session 开始调试。可以看到输出窗口中出现：



其中Hello world来自主函数的print_str("Hello world!\n")。而hello from sysTick!则来自时钟中断。中断处理函数在 stm32f10x_it.c 中：

```
void SysTick_Handler(void)
{
  print_str("hello from sysTick!\n");
  OSIntEnter();
  OSTimeTick();
  OSIntExit();
}
```

可以看到每次时钟中断出现，总会输出 hello from sysTick!。另外，时钟中断的配置函数在主函数中调用（systick_init()）。

现在在主函数中通过反注释掉 ASM_Switch_To_Unprivileged()来启动用户模式。此时再次运行会发现无法成功输出。ASM_Switch_To_Unprivileged()指向一段汇编代码，在 user_asm.s 中，这段代码通过设置 control 寄存器来修改处理器模式。

```
28  ASM_Switch_To_Unprivileged
29      MRS     R0, control
30      ORR     R0, #1
31      MSR     control, R0
32      BX      LR
```

注意在 user_asm.s 开始出需要声明 ASM_Switch_To_Unprivileged()为全局属性，这样其他文件才能访问。

```
4
5     EXPORT ASM_Switch_To_Unprivileged
6
```

同时，在调用 ASM_Switch_To_Unprivileged() 的文件中也要声明 ASM_Switch_To_Unprivileged()的属性。如在 main.c 中可以找到：

```
68
69  void ASM_Switch_To_Unprivileged(void);
70
```

现在判断程序无法正确执行的原因，单步跟踪 print_str()，该函数调用 fputcp 函数，而 fputcp 函数出现 IO 操作，由于在用户模式下不支持 IO 操作，处理器自动产生硬故障中断。所以，需要把 print_str()修改为系统调用。见下图。

```
11  #define BufferLen 100
12  void *buffer;
```

```
72  void syscall_print_str(char *str)
73  {
74    int i=0;
75    while(1)
76    {
77      if (*(str+i) == '\0')
78      {
79        ((char *)buffer)[i] = str[i];
80        break;
81      }
82      ((char *)buffer)[i] = str[i];
83      i++;
84    }
85    __ASM{
86      SWI 0x01
87    }
88  }
```

首先该函数把 str 中的内容复制到 buffer 中（该 buffer 为一个全局指针，在 main 函数中使用 malloc 自动分配内存），然后利用内联汇编执行软中断指令，而软中断指令自动触发中断，并跳转到 user_asm.s 中的中断处理程序中：

```
; @brief  This function handles SVCall exception.
; @param  None
; @retval None
SVC_Handler
    TST   LR, #4
    MRSEQ R1, MSP
    MRSNE R1, PSP
    ;r1 <- sp
    LDR   R0, [R1,#24]
    ;r0 <- pc
    SUB   R0, 2
    ;r0 <- instruction pointer
    LDR   R1, [R0]
    ; r1 <- instruction
    AND   R0, R1, 0xFF
    B     SVC_Handler_Main
```

可以跳转到这里是因为这个程序覆盖了启动文件 startup_stm32f10x_md.s 的 SVC_Handler。这个程序负责把 SWI 0x01 中的 0x01 放入 R0，然后调用 SVC_Handler_Main 函数。具体地，首先利用 LR 中的值来判断中断触发时候入栈的栈指针，然后利用栈指针找到中断触发时候的 PC，而 SWI 0x01 指令是放在 PC-2 处的，故此处把 PC 减 2，然后时候 LDR 指令读取 SWI 0x01 指令，最后通过异或 0xFF 获得立即数 0x01。

根据 C 编译器的惯例，在汇编中调用函数时，第一个参数在 R0 寄存器中，所以在 SVC_Handler_Main 中（stm32f10x_it.c 文件中）flag 的值即为 R0 的值，也是 SWI 0x01 的立即数。

该函数通过判断立即数，执行相应的真正中断处理。**这里的代码留给同学们填入。**

```
extern void * buffer;
void SVC_Handler_Main(int flag)
{
  switch (flag)
  {
    case 0x01:
    {
      // Your code here
      break;
    }
    case 0x02:
    {
      // Your code here
      break;
    }
  }
}
```

同样地，systick_init()也存在用户态下无法访问 IO 的问题，可以利用类似的方法单步调试并编写系统调用。注意此时使用的 SWI 后面的立即数不能是 0x01 了，需要定义一个其他的数，并且在 SVC_Handler_Main 中编写相应处理过程。

3. 任务二

3.1 任务内容

修改操作系统与 CPU 有关的源代码，使任务运行在用户态：

1. 使用 OSTaskCreate ()建立两个任务，并交替输出

要求：

1. CPU 为 unprivileged 模式

注意：实验开始之前，注释掉 SysTick_Handle 函数中的“print_str("hello from sysTick!\n");”以去掉不必要的输出

3.2 提示

单步调式（重点注意 OSInit()、OSTaskCreate()、OSStart()和 OSIntExit()函数，但

不是每个函数都要修改)，找到可能 unprivileged 模式下不能运行的部分，并修改为系统调用。

4. 任务三（选做）

4.1 任务内容

前面的任务中打印函数 syscall_print_str 需要事先准备一个 buffer，请你试图给出一个不需要 buffer 的版本。