SHANDONG UNIVERSITY

操作系统第 1 次实验报告

谢子洋 202100460116

2023 年 3 月 20 日

目录

1 实验内容

1.1 题目 1

当 CPU 处于内核态时可以执行全部指令, 操作所有寄存器. 而当 CPU 处于用户态时仅能执行非特权指令, 操作部分寄存器.

本题中要求调用 ASM_Switch_To_Unprivileged() 函数, 使 CPU 进入用户态. 用户态下能进行的操作有限, 欲正常执行程序需要使用系统调用触发中断, 此时 CPU 为内核态, 可以完成用户态下无法进行的操作.

1.1.1 初始化时钟

在原代码中, 程序通过函数传参的方式将值 (rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC) 传入函数 __STATIC_INLINE uint32_t SysTick_Config(). 而当 CPU 切换到用户态时, 将参数直接传入函数会造成错误.

部分原代码如下:

```
static void systick_init(void)
{
    RCC_ClocksTypeDef rcc_clocks;
    RCC_GetClocksFreq(&rcc_clocks);
    SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);
}
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
  if ((ticks - 1) > SysTick_LOAD_RELOAD_Msk)  return (1);
  SysTick->LOAD  = ticks - 1;
  NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
  SysTick->VAL   = 0;
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk   |
                   SysTick_CTRL_ENABLE_Msk;
  return (0);
}
```

可以先用 buffer 暂存参数值, 随后触发软中断, 在进行中断处理时再将该参数传入函数. 这是因为进行中断处理时 CPU 处于核心态, 所以可以在此时传入参数.

修改后代码:

```
//main.c文件syscall_systick_init函数增加语句 1.保存参数到buffer 2.触发软中断
static void syscall_systick_init(void)
{
  RCC_ClocksTypeDef rcc_clocks;
  RCC_GetClocksFreq(&rcc_clocks);
  //参数保存到buffer中
    *((uint32_t *)buffer)=rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC;
    __asm{SWI 0x02} //触发软中断
}
```

```c
//stm32g10x_it.c文件SVC_Handler_Main函数中switch语句增加case如下:
void SVC_Handler_Main(int flag)
{
    switch (flag)
    {
        case 0x02: {
            uint32_t data=*((uint32_t *)buffer);//读出buffer中保存的参数
            SysTick_Config(data);               //传入参数
            break;
        }
    }
}
```

1.1.2 显示输出

显示输出为 I/O 操作, 切换到用户态下直接执行会导致硬件异常, 将 print() 函数修改为系统调用, 在函数中内联汇编代码, 使用 SWI 指令执行软中断, 并进行中断处理, 以输出字符串.

系统调用中收首先将字符串数据保存到缓冲区 buffer 中, 代码将 buffer 定义为全局变量, 便于中断处理函数进行访问. 系统调用再触发软中断, 进入中断处理程序, 此时 CPU 处于核心态, 可以将 buffer 指向的字符串进行输出.

修改后代码如下:

```c
//main.c文件syscall_print_str函数添加功能:1.将字符串保存到buffer 2.触发软中断
  void syscall_print_str(char *str)
  {
    int i=0;
    while(1)                //循环,保存字符串到buffer中
    {
      if (*(str+i) == '\0')
      {
        ((char *)buffer)[i] = str[i];
        break;
      }
      ((char *)buffer)[i] = str[i];
      i++;
    }
    __ASM{SWI 0x01}        //触发软中断
  }
```

```c
  //stm32g10x_it.c文件SVC_Handler_Main函数中switch语句增加case如下:
  void SVC_Handler_Main(int flag)
  {
    switch (flag)
    {
      case 0x01:
      {
        char*poi=buffer;//读取字符串首地址
        uint32_t i;
        i = 0;
```

```
11          while(1)          //输出打印字符串
12          {
13            if ( *(poi+i) == '\0'  ) {
14              break;
15            }
16            if(fputcp(*(poi+i))==0){
17              break;
18            }
19            i++;
20          }
21          break;
22        }
23      }
24    }
```

1.2 题目 2

1.2.1 异常 1

main() 中 OSSTART() 触发异常, 单步调试查找出错位置:

main() -> OSSTART() -> OSStartHighRdy() -> STRB R1, [R0]

进行调试, 发现执行 STRB 指令时发生硬件错误, 此指令用以写寄存器至存储 (内存). 用户态下执行该指令会发生异常, 因此要在此处触发软中断并进行中断处理.

原代码如下:

```
void  OSStart (void)
{
    if (OSRunning == OS_FALSE) {
        OS_SchedNew();
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();
    }
}
```

修改为系统调用后代码如下:

```
//main.c文件增加syscall_print_str函数,定义如下:
  void  syscall_OSSTART(void)
  {
      if (OSRunning == OS_FALSE) {
          OS_SchedNew();
          OSPrioCur     = OSPrioHighRdy;
          OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
          OSTCBCur      = OSTCBHighRdy;
          //added
          __ASM{SWI 0x03}
      }
  }
```

```
//stm32g10x_it.c文件SVC_Handler_Main函数中switch语句增加case如下:
  void SVC_Handler_Main(int flag)
  {
    switch (flag)
    {
      case 0x03:
      {
        OSStartHighRdy();
        break;
      }
    }
  }
```

1.2.2 异常 2

task 中 OSTimeDly(100) 函数触发异常, 单步调试查找出错位置, 并根据宏定义

$$\#define\ \ OS\_TASK\_SW()\ \ OSCtxSw()$$

可知程序错误来源:

task1() -> OSTimeDly() -> OS_Sched() -> OS_TASK_SW() <=> OSCtxSw() -> STR R1, [R0]

汇编指令中 STR 指令写存储 (内存) 触发异常. 解决方式与上文基本相同, 通过内联汇编手动触发软中断, 并进行中断处理.

部分原代码如下:

```
void  OS_Sched (void){
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif

    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) {
        if (OSLockNesting == 0) {
            OS_SchedNew();
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];

#if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++;
#endif
                OSCtxSwCtr++;
                OS_TASK_SW();
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```

```
OSCtxSw
    CPSID   I
    LDR     R0, =NVIC_INT_CTRL
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]
    CPSIE   I
    BX      LR
```

修改为系统调用后代码如下:

```
//os_core.c文件OS_Sched函数 1.去除对OS_TASK_SW()的调用 2. 增加__ASM{SWI 0x04}
  void  OS_Sched (void){
  #if OS_CRITICAL_METHOD == 3
      OS_CPU_SR  cpu_sr = 0;
  #endif
```

```
6
7       OS_ENTER_CRITICAL();
8       if (OSIntNesting == 0) {
9           if (OSLockNesting == 0) {
10              OS_SchedNew();
11              if (OSPrioHighRdy != OSPrioCur) {
12                  OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
13
14  #if OS_TASK_PROFILE_EN > 0
15                  OSTCBHighRdy->OSTCBCtxSwCtr++;
16  #endif
17                  OSCtxSwCtr++;
18                  //added
19                  __ASM{SWI 0x04}
20              }
21          }
22      }
23      OS_EXIT_CRITICAL();
24  }
```

```
1   //stm32g10x_it.c文件SVC_Handler_Main函数
2   //1.switch语句增加case 2.case中调用OS_TASK_SW():
3   void SVC_Handler_Main(int flag)
4   {
5     switch (flag)
6     {
7       case 0x04:
8       {
9         OS_TASK_SW();
10        break;
11      }
12    }
13  }
```

1.2.3 异常 3

原代码中输出打印字符串使用的是 print_str() 函数, 切换到用户态时则应使用系统调用 syscall_print_str().

1.3 题目 3

1.3.1 思路

使用寄存器保存字符数组首地址, 在中断处理时读取寄存器保存值, 以找到字符串地址并进行输出.

1.3.2 执行过程及修改内容

(1) 调用新的系统调用 syscall_print_str_2():

```
//main.c文件中添加新函数定义如下:
void syscall_print_str_2(char *str){
    __ASM{SWI 0x05}
}
```

C 语言编译成汇编语言时, 函数接收的第一个参数会保存在寄存器 R0 中, 因此该系统调用 syscall_print_str_2() 的参数 char *str 会被暂存到寄存器 R0 中.

其后执行 SWI 指令进行跳转, 跳转并不会改变 R0 保存的值.

(2) 随后进入一段汇编代码:

```
//user_asm.s文件SVC_Handler函数添加如下指令(对原函数执行无影响)
SVC_Handler
    MOV   R7,R0               ;added code
    TST   LR, #4
    MRSEQ R1, MSP
    MRSNE R1, PSP
    LDR   R0, [R1,#24]
    SUB   R0, 2
    LDR   R1, [R0]
    AND   R0, R1, 0xFF
    MOV   R1,R7                 ;added code
    B     SVC_Handler_Main
```

首先将 R0 中的值暂存到 R7 中, 之后按原代码执行, 最后在执行跳转前再将 R7 中的值保存到 R1. 因为该过程原来并未使用到寄存器 R7, 所以写入 R7 是安全的. 并且 R0 寄存器是在跳转前才被写入新值, 对原过程计算立即数不造成影响.

汇编执行完成后 R0 保存 SWI 指令中的立即数; 而 R1 则保存待输出字符串的首地址.

(3) 接着跳转到 SVC_Handler_Main() 函数:

```
//stm32g10x_it.c文件SVC_Handler_Main()函数switch语句中添加case如下:
void SVC_Handler_Main(int flag,void* str)
{
    switch (flag)
    {
        case 0x05:
        {
            char*poi=(char*)str;
            print_str(poi);
            break;
        }
    }
}
```

此时第一个参数 flag 的值保存在寄存器 R0 中, 而增加的参数 str 的值保存在寄存器 R1 中. 所以参数 str 即为待输出字符串首地址, 此时调用函数输出即可.

2 结果分析与实验心得

当 CPU 处于内核态时可进行任何操作, 具有最高权限, 但内核态不能开放给任何程序, 因为可能造成未知异常或安全问题.

因此对于一般的程序,CPU 处于用户态下, 欲想进行一些特别操作需调用提前写好的系统调用. 系统调用能触发软中断令 CPU 切换到内核态, 在中断处理中进行一般进程无法执行的操作.

对于用户态下会触发硬中断的操作, 可以将指令改为系统调用, 在其中手动触发软中断并执行中断处理程序.

软中断是唯一可以主动进入特权模式的方法.

参考文献

[1] Andrew S.Tanenbaum,Herbert Bos. 现代操作系统. 北京: 机械工业出版社,2017.