操作系统第一次实验

学号：202200460104

班级：网安一班

姓名：密语

2024 年 3 月 23 日

目录

1 摘要

本次实验环境基于 μC/OS-II，设计系统调用，完成用户态下的时钟初始化和显示输出和创建任务交替输出以及在不使用指针的情况下在用户态下实现显示输出的功能。

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 任务实现过程

2.2.1 首先分析任务一的 main 函数

```
#ifdef FIRST_TASK
int main(void)
{
    buffer = malloc(BufferLen);
    ASM_Switch_To_Unprivileged();
    print_str("Hello world!\\n");
    systick_init();
    while(1){}
}
#endif
```

- `buffer = malloc(BufferLen);` 为 buffer 指针申请一块大小为 BufferLen 的地址空间
- `ASM_Switch_To_Unprivileged();`将 CPU 模式从内核态转换为用户态
- `print_str("Hello world!\\n");`打印字符串 Hello world!
- `systick_init();`时钟初始化

2.2.2 在用户态下进行调试

发现当程序运行到这个函数的时候发生了硬中断（Hard Fault exception occurs!）

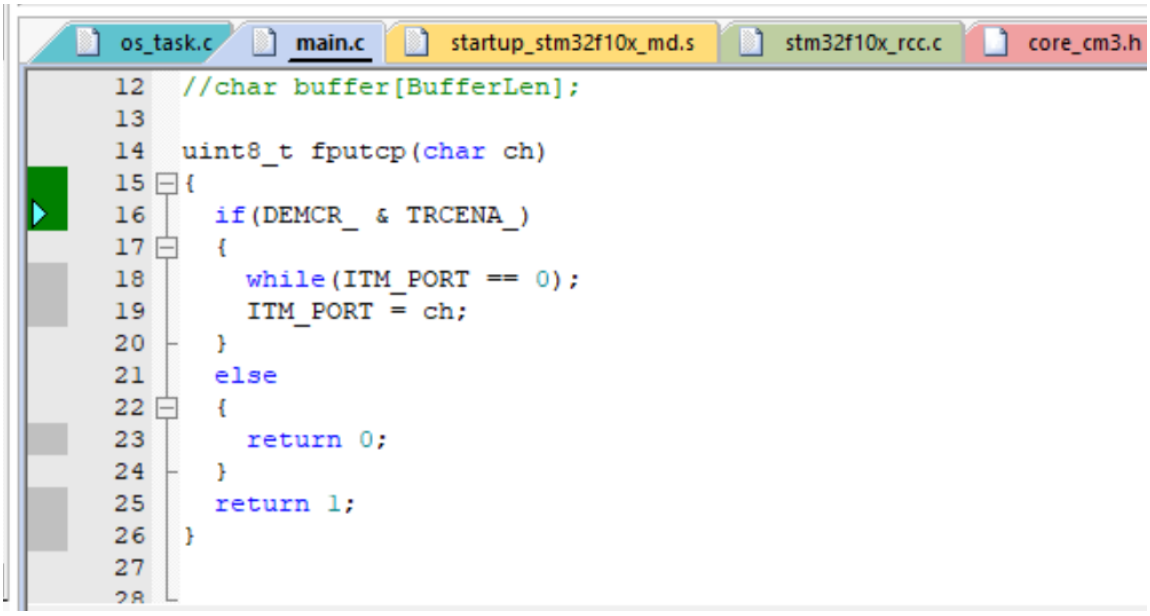

图 1: 调试界面

回溯发现这个函数在`print_str()` 中被调用（第 39 行）

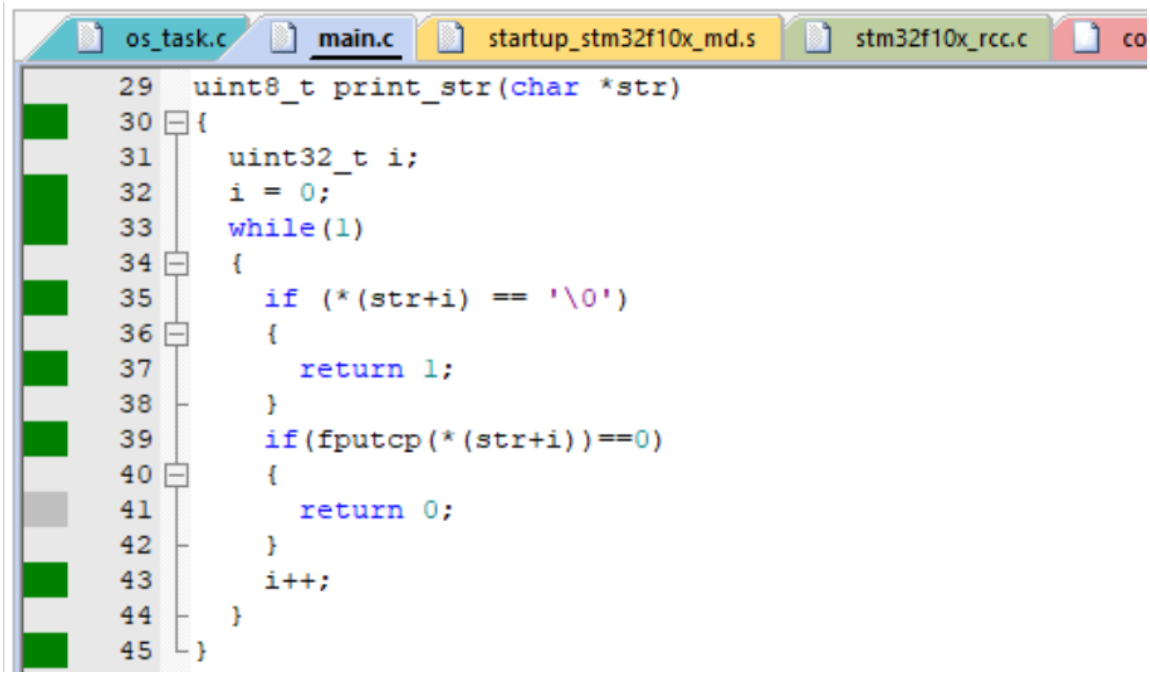

图 2: 调试界面

为了解决这个问题，我们可以把字符串赋值给 buffer 指针，然后通过中断处理程序进行输出的显示。为此，我们定义一个函数`syscall_print_str()`。

```
1 void syscall_print_str(char *str)
2 {
3     int i=0;
4     while(1)
```

```
        {
                if (*(str+i) == '\0')
                {
                        ((char *)buffer)[i] = str[i];
                        break;
                }
                ((char *)buffer)[i] = str[i];
                i++;
        }
        __ASM{
                SWI 0x01
        }
}
```

这个函数将 str 指针中储存的字符串赋值给 buffer 指针，然后跳转到中断处理函数中进行输出。中断处理函数改为：

```
void SVC_Handler_Main(int flag)
{
    switch (flag)
        {
                case 0x01:
                {
                        print_str(buffer);
                        break;
                }
        }
}
```

因此 main 函数改为：

```
#ifdef FIRST_TASK
int main(void)
{
        buffer = malloc(BufferLen);
        ASM_Switch_To_Unprivileged();
        syscall_print_str("Hello world!\n");
        systick_init();
        while(1){}
}
#endif
```

继续调试，发现成功打印出 Hello world!

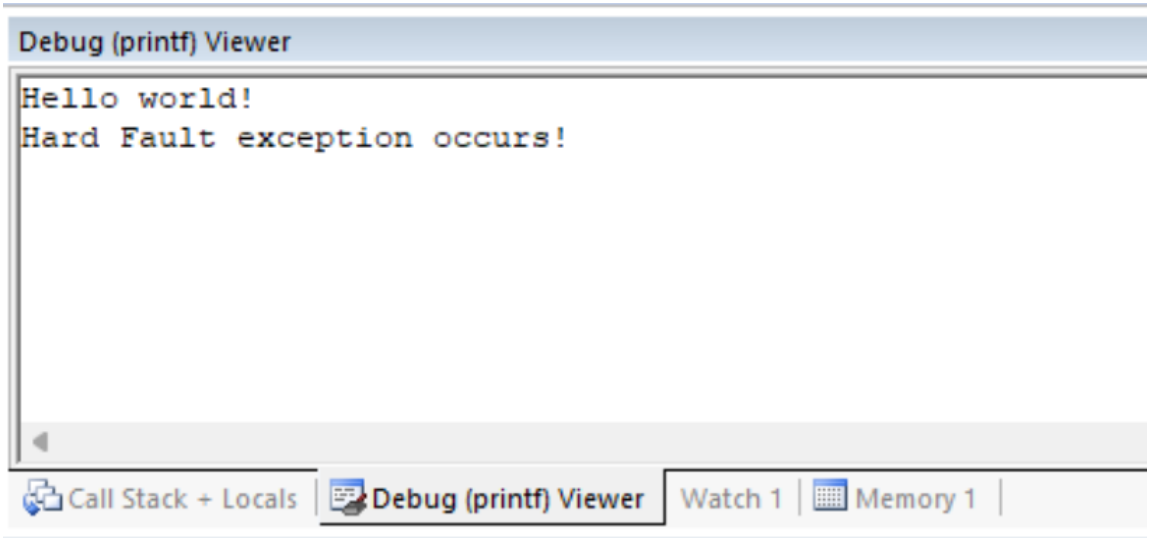

图 3: 输出界面

但是后面又发生了硬中断，说明systick_init()函数在不能在用户态下运行
继续单步调试，发现在此处发生了硬中断：

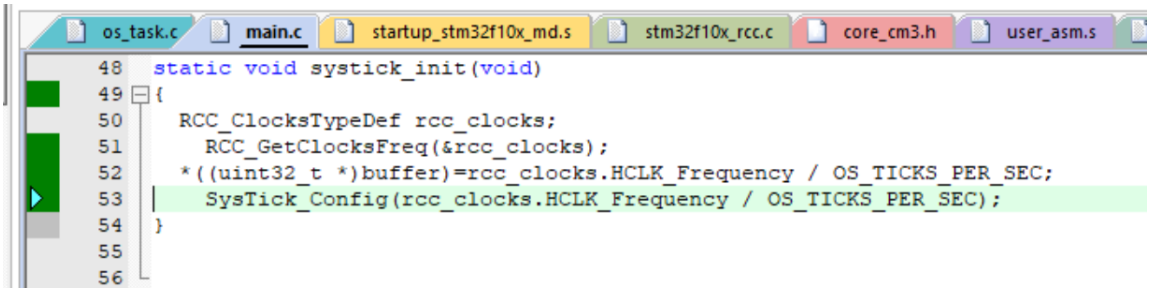

图 4: 调试界面

于是我们还是采用上面的思路，在此处跳转到中断处理程序里执行
SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);
为此我们定义一个syscall_systick_init() 函数，利用 buffer 指针进行传参。

```
1  static void syscall_systick_init(void)
2  {
3      RCC_ClocksTypeDef rcc_clocks;
4      RCC_GetClocksFreq(&rcc_clocks);
5      *((uint32_t *)buffer)=
6          rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC;
7      __asm
8      {
9          SWI 0x02
10     }
11
12 //SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);
13 }
```

中断处理函数改为：

```
void SVC_Handler_Main(int flag)
{
    switch (flag)
    {
        case 0x01:
        {
                print_str(buffer);
                break;
        }
        case 0x02:
        {
                SysTick_Config(*((uint32_t *)buffer));
                break;
        }
    }
}
```

main 函数修改为：

```
#ifdef FIRST_TASK
int main(void)
{
    buffer = malloc(BufferLen);
    ASM_Switch_To_Unprivileged();
    syscall_print_str("Hello world!\n");
    syscall_systick_init();
    while(1){}
}
#endif
```

2.2.3 运行效果

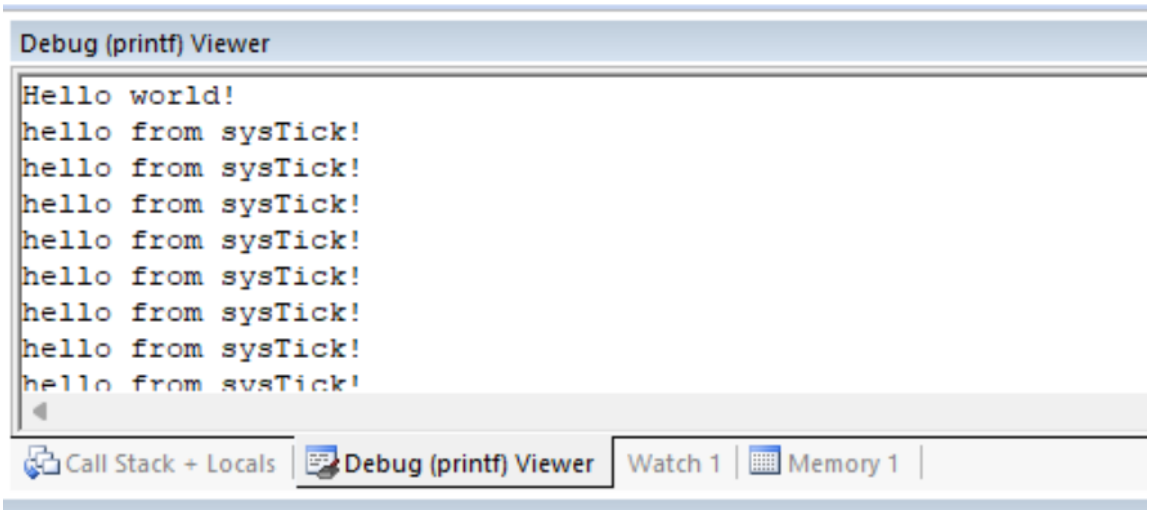

图 5: 输出界面

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
- 使用`OSTaskCreate()`建立两个任务，并交替输出
- 要求：CPU 为 unprivileged 模式

3.2 任务实现过程

3.2.1 首先分析任务二的 main 函数

```
1  int main(void)
2  {
3      buffer = malloc(BufferLen);
4      ASM_Switch_To_Unprivileged();
5      syscall_print_str("Hello world!\n");
6      OSInit();
7      syscall_systick_init();
8      OSTaskCreate(task1, (void *)0,
9          &task1_stk[TASK1_STK_SIZE-1], TASK1_PRIO);
10     OSTaskCreate(task2, (void *)0,
11         &task2_stk[TASK2_STK_SIZE-1], TASK2_PRIO);
12     OSStart();
13     return 0;
14 }
```

- `buffer = malloc(BufferLen);` 为 buffer 指针申请一块内存空间
- `ASM_Switch_To_Unprivileged();` 将 CPU 模式从内核态转为用户态
- `syscall_print_str("Hello world!\n");` 用户态下输出字符串
- `OSInit();` 在 OS 应用中的 main() 函数中首先被调用，是 OS 运行的第一个函数, 它完成各初始变量的初始化。
- `syscall_systick_init();` 时钟初始化
- `OSTaskCreate();` 创建任务
- `OSStart();`运行优先级最高的就绪任务

3.2.2 分析任务函数

```
1  static void task1(void *p_arg)
2  {
3          for (;;)
4          {
5                  OSTimeDly(100);
6                  syscall_print_str("Hello from Task 1!\n");
7                  OSTimeDly(100);
8          }
9  }
10 static void task2(void *p_arg)
11 {
12         for (;;)
13         {
14                 OSTimeDly(100);
15                 syscall_print_str("Hello from Task 2!\n");
16                 OSTimeDly(100);
17         }
18 }
```

- `OSTimeDly(100);` 任务延时函数，延时 100ms
- `syscall_print_str();` 分别打印两个语句

3.2.3 在用户态下进行调试

发现在执行 OSStartHighRdy() 函数时触发了硬中断。

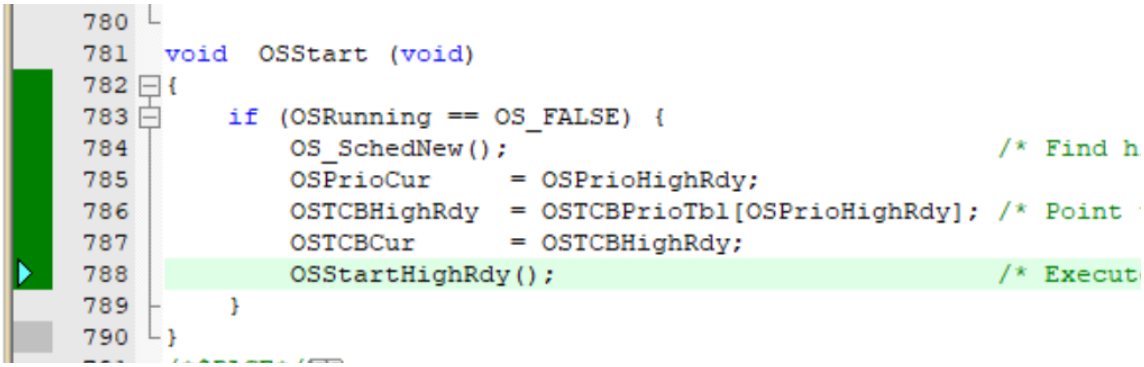

图 6: 输出界面

于是我们还是采用跳转＋中断处理函数的思路，将其修改为系统调用。修改后的 OSStart() 函数：

```
void  OSStart (void)
{
    if (OSRunning == OS_FALSE) {
        OS_SchedNew();
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur      = OSTCBHighRdy;
                                __asm{
                                        SWI 0x03
                                }
        //OSStartHighRdy();
    }
}
```

并在中断处理函数中增加相应的分支：

```
void SVC_Handler_Main(int flag)
{
        switch (flag)
        {
                case 0x01:
                {
                        print_str(buffer);
                        break;
                }
                case 0x02:
                {
                        SysTick_Config(*((uint32_t *)buffer));
                        break;
                }
                case 0x03:
```

```
                {
                        OSStartHighRdy();
                        break;
                }
        }
}
```

继续进行单步调试，发现执行到 OS_TASK_SW() 函数时触发了硬中断。

还是采用跳转＋中断处理函数的思路，此处改为：

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3
#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++;
                __asm{
                      SWI 0x04
                }
                //OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

同时中断处理函数改为：

```
void SVC_Handler_Main(int flag)
{
        switch (flag)
        {
                case 0x01:
                {
                        print_str(buffer);
                        break;
                }
                case 0x02:
```

```
11                  {
12                          SysTick_Config(*((uint32_t *)buffer));
13                          break;
14                  }
15                  case 0x03:
16                  {
17                          OSStartHighRdy();
18                          break;
19                  }
20                  case 0x04:
21                  {
22                          OS_TASK_SW();
23                          break;
24                  }
25          }
26  }
```

3.2.4 运行效果

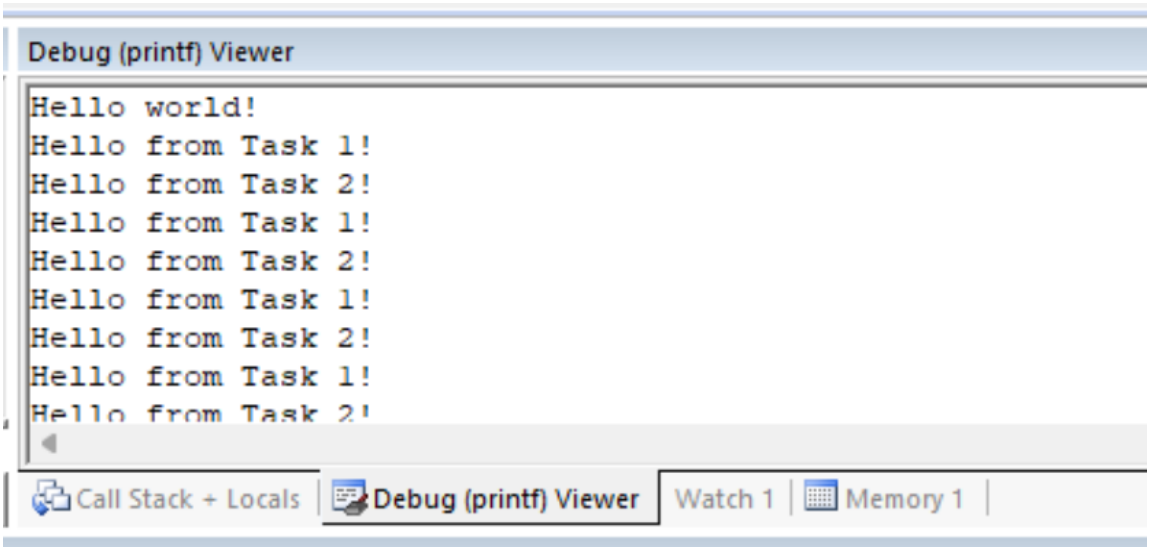

图 7: 输出界面

4 任务三

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。

4.2 任务实现过程

4.2.1 首先分析 syscall_print_str 函数

```
1   void syscall_print_str(char *str)
2   {
3       int i=0;
4       while(1)
5       {
6           if (*(str+i) == '\0')
7           {
8               ((char *)buffer)[i] = str[i];
9               break;
10          }
11          ((char *)buffer)[i] = str[i];
12          i++;
13      }
14      __ASM{
15          SWI 0x01
16      }
17  }
```

由于 str 指针是第一个参数，它被保存在 R0 寄存器中，因此我们可以直接在中断处理函数中使用 R0 寄存器的值。

我们再分析一下中断处理函数（SVC_Handler）：

```
1   SVC_Handler
2       TST   LR, #4
3       MRSEQ R1, MSP
4       MRSNE R1, PSP
5       ;r1 <- sp
6       LDR   R0, [R1,#24]
7       ;r0 <- pc
8       SUB   R0, 2
9       ;r0 <- instruction pointer
10      LDR   R1, [R0]
11      ; r1 <- instruction
12      AND   R0, R1, 0xFF
13      B     SVC_Handler_Main
```

我们可以发现在第 6 行，R0 寄存器中存储的是 PC 寄存器的值，即指令指针，之前 R0 存储的 str 指针的值被覆盖掉了，因此我们考虑先把 R0 寄存器的值保存在 R6 寄存器，然后再使用。

于是我们加上了这一行代码：MOV R6, R0

最终 SVC_Handler 返回 SVC_Handler_Main 函数,R0 存储的是 SVC_Handler_Main 函数的第一个参数的值，即 flag 的值。

我们考虑给 SVC_Handler_Main 函数添加一个参数 char *str，然后将 R6 的值传给 R1，即 str 的值传给了 char *str。

于是我们在最后加上了这一行代码：MOV R1, R6

修改后的 SVC_Handler 函数：

```
SVC_Handler
    MOV  R0, R6
    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
    MOV   R6, R0
    B     SVC_Handler_Main
```

修改后的 SVC_Handler_Main 函数：

```

void SVC_Handler_Main(int flag, char *str)
{
    switch (flag)
    {
        case 0x01:
        {
            print_str(buffer);
            break;
        }
        case 0x02:
        {
            SysTick_Config(*((uint32_t *)buffer));
            break;
        }
        case 0x03:
        {
            OSStartHighRdy();
            break;
        }
        case 0x04:
        {
```

```
23                  OS_TASK_SW();
24                  break;
25              }
26              case 0x05:
27              {
28                  print_str(str);
29              }
30          }
31      }
```

我们定义一个新的函数 syscall_print_str_without_buffer，它不需要 buffer 指针，直接使用 str 指针。syscall_print_str_without_buffer 函数如下：

```
1       \begin{lstlisting}
2       void syscall_print_str_without_buffer(char *str)
3       {
4           __ASM{
5               SWI 0x05
6           }
7       }
```

这个函数将直接跳转到中断处理函数中，然后调用 print_str 函数。char *str 作为第二个参数传给了 SVC_Handler_Main 函数。这样就可以不使用 buffer 指针打印字符串了。

main 函数也要做相应的修改：

```
1   int main(void)
2   {
3           //buffer = malloc(BufferLen);
4           ASM_Switch_To_Unprivileged();
5           syscall_print_str_without_buffer("Hello world!\n");
6           //syscall_print_str_without_buffer("Hello world!\n");
7           syscall_systick_init();
8           while(1){}
9   }
```

4.2.2 运行效果

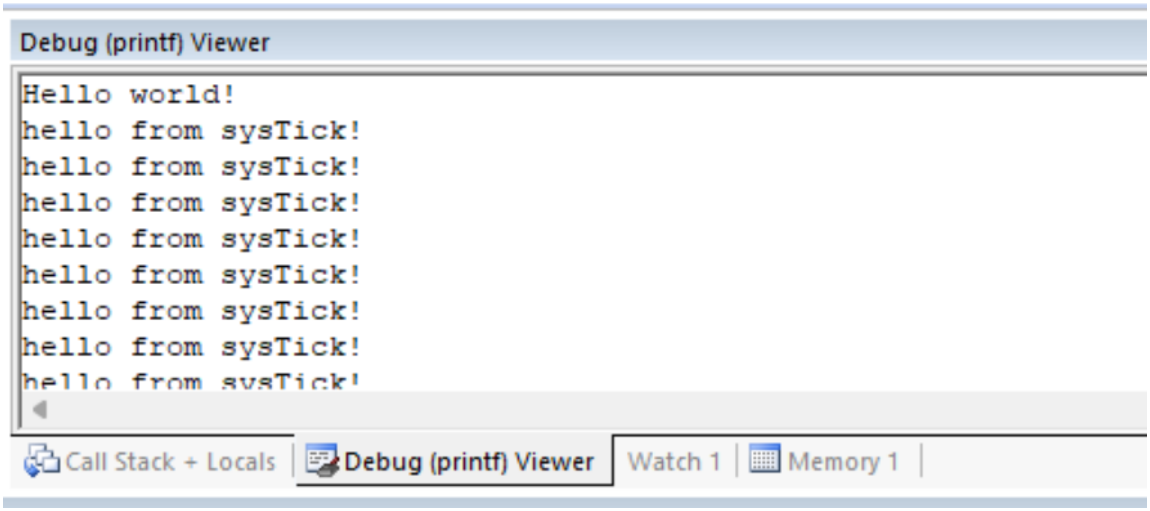

图 8: 输出界面

5 总结

通过本次实验，我学会了如何在用户态下实现系统调用，如何在用户态下实现时钟初始化和显示输出，以及如何创建任务并交替输出。我还了解了函数的参数在寄存器中的存储方式，以及如何在不使用 buffer 指针的情况下实现显示输出的功能。这次实验让我对操作系统的内核有了更深入的了解，对操作系统的实现原理有了更深刻的认识。