华东师范大学计算机科学与技术学院上机实践报告

课程名称：操作系统	年级：2022级	基础实验成绩：
指导教师：石亮	姓名：田亦海	扩展思考成绩：
上机实践名称：实验1	学号：10225101529	上机实践日期：2023年9月25日

Ⅰ 练习题

Q2

在 **kernel/arch/aarch64/boot/raspi3/init/tools.S** 中 arm64_elX_to_el1 函数的 LAB 1 TODO 1 处填写一行汇编代码，获取 CPU 当前异常级别。

A2

查看汇编代码，发现后面根据寄存器x9的值，比较其是否为各个异常等级，继而设置为ex1.可知应该是在todo内将当前异常等级的值赋给了x9.补充代码如下：

```
    /* LAB 1 TODO 1 BEGIN */
    mrs x9, CurrentEL
    /* LAB 1 TODO 1 END */
```

```
BEGIN_FUNC(arm64_elX_to_el1)
    /* LAB 1 TODO 1 BEGIN */
    mrs x9, CurrentEL
    /* LAB 1 TODO 1 END */

    // Check the current exception level.
    cmp x9, CURRENTEL_EL1
    beq .Ltarget
    cmp x9, CURRENTEL_EL2
    beq .Lin_el2
    // Otherwise, we are in EL3.

    // Set EL2 to 64bit and enable the HVC instruction.
    mrs x9, scr_el3
    mov x10, SCR_EL3_NS | SCR_EL3_HCE | SCR_EL3_RW
    orr x9, x9, x10
    msr scr_el3, x9

    // Set the return address and exception level.
    adr x9, .Ltarget
    msr elr_el3, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el3, x9
    eret

.Lin_el2:
```

Q3

在 arm64_elX_to_el1 函数的 LAB 1 TODO 2 处填写大约 4 行汇编代码，设置从 EL3 跳转到 EL1 所需的 elr_el3 和 spsr_el3 寄存器值

A3

切换异常状态时，我们需要修改`ELR_ELn`以及`SPSR_ERn`的值。具体地，将`ELR_ELn`设置为异常跳转恢复后地目标地址，`SPSR_ERn`设置一些处理器的状态。我们需要从EL3改变到EL1，因此设置`ELR_EL3`为.Ltarget的地址(即ret指令的位置)，设置`SPSR_EL3`为`SPSR_ELX_DAIF | SPSR_ELX_EL1H`，分别对应暂时屏蔽异常事件以及改变为EL1等级。代码：

```
/* LAB 1 TODO 2 BEGIN */
    adr x9, .Ltarget
    msr elr_el3, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el3, x9
/* LAB 1 TODO 2 END */
```

```
    // Set the return address and exception level.
    /* LAB 1 TODO 2 BEGIN */
    adr x9, .Ltarget
    msr elr_el3, x9
    mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
    msr spsr_el3, x9
    /* LAB 1 TODO 2 END */
    eret
```

Q6

在 **kernel/arch/aarch64/boot/raspi3/peripherals/uart.c** 中 LAB 1 TODO 3 处实现通过 UART 输出字符串的逻辑。

A6

这个就不用解释了

```
76
77  void uart_send_string(char *str)
78  {
79          /* LAB 1 TODO 3 BEGIN */
80          for (int i = 0; str[i] != '\0'; i++)
81                  early_uart_send(str[i]);
82          /* LAB 1 TODO 3 END */
83  }
84
```

Q7

在 kernel/arch/aarch64/boot/raspi3/init/tools.S 中 LAB 1 TODO 4 处填写一行汇编代码，以启用 MMU

A7

```
254     /* Enable MMU */
255     /* LAB 1 TODO 4 BEGIN */
256     orr     x8, x8, #1
257     /* LAB 1 TODO 4 END */
```

经过查阅文档，我们知道`SCTLR_EL1_M`也就是`sctlr_el1`的第0位表示MMU的启动与否,因此我们用orr指令将x8第0位置为1即可

第256行的意思是`x8=x8|1`

在后面会把`x8`赋值给`sctlr_el1`的，这样就启动MMU了。

PS：全局搜索找到文件，可以找到宏定义`SCTLR_EL1_M`相当于`1<<0`

所以也可以填写为`orr     x8,x8,#SCTLR_EL1_M`

```
141 #define SCTLR_EL1_C    BIT(2) /* Cacheability control for
142 #define SCTLR_EL1_A    BIT(1) /* Alignment check enable *
143 #define  #define BIT(x) (1ULL << (x))
144         BIT(0) /* MMU enable for EL1 and EL0 stage 1 add
145
```

II 思考题

Q1

阅读 _start 函数的开头，尝试说明 ChCore 是如何让其中一个核首先进入初始化流程，并让其他核暂停执行的

A1

_start函数是ChCore启动时先执行的一段代码，QEMU会先启动4个CPU核心，都执行_start。需要让其中一个核先开始初始化。

```
20  BEGIN_FUNC(_start)
21      mrs x8, mpidr_el1   //mpidr_el1中是当前PE的cpuid
22      and x8, x8, #0xFF   //保留低八位
23      cbz x8, primary     //如果为0，则跳转到primary(0为cpu编号)
24
25      /* hang all secondary processors before we introduce smp */
26      b   . //死循环
27
28  primary:
29      /* Turn to el1 from other exception levels. */
30      bl  arm64_elX_to_el1  //函数调用，设置异常级别为el1
31
32      /* Prepare stack pointer and jump to C. */
33      ldr     x0, =boot_cpu_stack   //以下为继续做的一些初始化内容
34      add     x0, x0, #INIT_STACK_SIZE
35      mov     sp, x0
36
37      bl  init_c
38
39      /* Should never be here */
40      b   .
41  END_FUNC(_start)
42  //这样虽然四个核都执行_start,但只有cpu0继续进行primary中的初始化，其他均被死循环挂起。
```

Q4

结合此前 ICS 课的知识，并参考 kernel.img 的反汇编（通过 aarch64-linux-gnu-objdump -S 可获得），说明为什么要在进入 C 函数之前设置启动栈。如果不设置，会发生什么？

A4

参考此段代码：

（左侧为kernel.img的反汇编，右侧为start.S）

```
000000080010 <primary>:
0010:   94001ffc    bl 88000 <arm64_elX_to_el1>
0014:   580000a0    ldr   x0, 80028 <primary+0x18>
0018:   91400400    add   x0, x0, #0x1, lsl #12
001c:   9100001f    mov   sp, x0
0020:   940020e2    bl 883a8 <init_c>
0024:   14000000    b  80024 <primary+0x14>
0028:   00088980    .word 0x00088980
002c:   00000000    .word 0x00000000
0030:   14000400    b  81030 <__start_kernel_veneer+0xff8>
0034:   d503201f    nop

000000080038 <__start_kernel_veneer>:
0038:   58000090    ldr   x16, 80048 <__start_kernel_venee
003c:   10000011    adr   x17, 8003c <__start_kernel_venee
0040:   8b110210    add   x16, x16, x17
0044:   d61f0200    br x16
0048:   0000ffc4    .word 0x0000ffc4
004c:   ffffff00    .word 0xffffff00
..
1030:   0000006b    .word 0x0000006b
1034:   00490003    .word 0x00490003
```

```
18 .extern init_c
19
20 BEGIN_FUNC(_start)
21         mrs     x8, mpidr_el1 //
22         and     x8, x8, #0xFF
23         cbz     x8, primary
24
25         /* hang all secondary processors before we introduce smp */
26         b       .
27
28 primary:
29         /* Turn to el1 from other exception levels. */
30         bl      arm64_elX_to_el1
31
32         /* Prepare stack pointer and jump to C. */
33         ldr     x0, =boot_cpu_stack
34         add     x0, x0, #INIT_STACK_SIZE
35         mov     sp, x0
36
37         bl      init_c
38
39         /* Should never be here */
40         b       .
41 END_FUNC(_start)
```

从汇编代码进入C函数时，设置了sp栈指针为boot_cpu_stack的最高位（很明显栈从高到低增长，所以初始为最高位）。调用C函数时，被调函数内部会把当前SP复制到FP（x29）中，并保存返回地址、调用函数的寄存器、参数等。如果不初始化启动栈，就无法利用栈保存现场以及其他数据，无法通过栈里的地址返回。

```
595   0000000883a8 <init_c>:
596   83a8:   a9bf7bfd    stp   x29, x30, [sp, #-16]!
597   83ac:   910003fd    mov   x29, sp
598   83b0:   97ffffe5    bl 88344 <clear_bss>
599   83b4:   94001173    bl 8c980 <early_uart_init>
600   83b8:   90000000    adrp  x0, 88000 <arm64_elX_to_el1>
601   83bc:   9110c000    add   x0, x0, #0x430
602   83c0:   940011d0    bl 8cb00 <uart_send_string>
603   83c4:   97ffe36c    bl 81174 <init_boot_pt>
```

Q5

在实验 1 中，其实不调用 clear_bss 也不影响内核的执行，请思考不清理 .bss 段在之后的何种情况下会导致内核无法工作。

```
void init_c(void)
{
        /* Clear the bss area for the kernel image */
        clear_bss();

        /* Initialize UART before enabling MMU. */
        early_uart_init();
        uart_send_string("boot: init_c\r\n");
```

A5

全局变量与静态变量没有初始化或初始化值为0时，都会放在.bss段。如果不清零.bss段的话，会出现一个问题：如果我们定义了一个全局变量`int a=0;`,那么会放在.bss段，我们认为a的值已经是0了。经过使用，a的值可能会变成其他值。那么下次运行时，如果不清空.bss段，a的初始值不是0，但我们以为a的初始值已经是0了，这样可能就会出现未知的问题。

也就是说，不清理.bss段时，若上一次运行时.bss中的值被修改为非零，这次运行时可能会出现未知问题。

附上clear_bss

```
0000000000088344 <clear_bss>:
   88344:   d10083ff    sub   sp, sp, #0x20
   88348:   900000c0    adrp  x0, a0000 <init_end+0x10000>
   8834c:   f9400400    ldr   x0, [x0, #8]
   88350:   f9000be0    str   x0, [sp, #16]
   88354:   900000c0    adrp  x0, a0000 <init_end+0x10000>
   88358:   f9400800    ldr   x0, [x0, #16]
   8835c:   f90007e0    str   x0, [sp, #8]
   88360:   f9400be0    ldr   x0, [sp, #16]
   88364:   f9000fe0    str   x0, [sp, #24]
   88368:   14000006    b  88380 <clear_bss+0x3c>
   8836c:   f9400fe0    ldr   x0, [sp, #24]
   88370:   3900001f    strb  wzr, [x0]
   88374:   f9400fe0    ldr   x0, [sp, #24]
   88378:   91000400    add   x0, x0, #0x1
   8837c:   f9000fe0    str   x0, [sp, #24]
   88380:   f9400fe1    ldr   x1, [sp, #24]
   88384:   f94007e0    ldr   x0, [sp, #8]
   88388:   eb00003f    cmp   x1, x0
   8838c:   54ffff03    b.cc  8836c <clear_bss+0x28>  // b.lo, b.ul, b.last
   88390:   900000c0    adrp  x0, a0000 <init_end+0x10000>
   88394:   f9400c00    ldr   x0, [x0, #24]
   88398:   f900001f    str   xzr, [x0]
   8839c:   d503201f    nop
   883a0:   910083ff    add   sp, sp, #0x20
   883a4:   d65f03c0    ret
```

III 拓展思考题

（本次实验无拓展思考题）

IV 总结

请在下面附上执行make grade命令的分数截图，作为实验成绩的参考。