

lab1 机器启动

黄培正 521021910454

思考题1

_start 函数的开头首先通过读取 mpidr_el1 寄存器获取当前 CPU 的 ID，并将其存储在寄存器 x8 中。然后通过与操作和比较，检查当前 CPU 的 ID 是否为 0。如果当前 CPU 的 ID 不为 0，则会跳转到 primary 标签处执行初始化流程。

在 primary 标签处，ChCore 会继续将当前 CPU 切换到 EL1 级别，并设置好栈指针，然后跳转到 init_c 函数执行初始化流程。

而对于其他 CPU，当其进入 _start 函数后，会先读取 secondary_boot_flag 中对应 CPU ID 的标志位，如果标志位为 0，表示其他 CPU 应该暂停执行，因此会通过一个循环等待其他核心的标志位被设置为非零值，然后设置 CPU ID 并跳转到 secondary_init_c 函数执行初始化流程。

因此，ChCore 通过在 _start 函数中检测当前 CPU 的 ID，然后根据不同的 CPU ID 执行不同的初始化流程，从而实现了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功能。

练习题2

添加的代码如下：

```
mrs x9, CurrentEL
```

CurrentEL 系统寄存器可获得当前异常级别，将其值移动到 x9，下面使用 x9 的值检查当前的异常级别。

练习题3

添加的代码如下：

```
adr x9, .Ltarget
msr elr_el3, x9
mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
msr spsr_el3, x9
```

这段代码的主要工作是设置 EL3 的 exception link register，将 ret 位置的标签设置到这里，为 eret 做准备，并且设置 EL3 的状态寄存器 SPSR，包括 debug, error, interrupt 和 fast interrupt。

思考题4

C 语言在函数调用的过程中，涉及函数参数的压栈、返回地址压栈等行为，如果不设置栈的话，C 函数无法传递多个参数或者无法返回调用者等情况。

思考题5

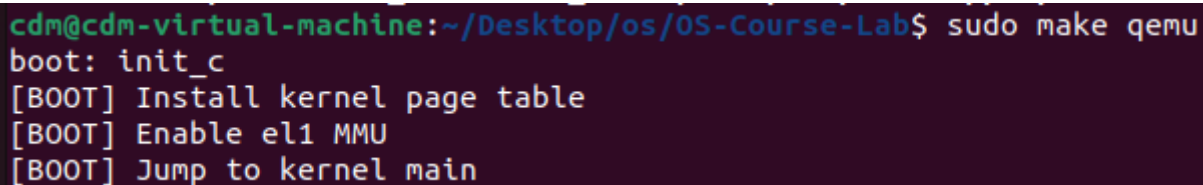
如果C函数使用了这些没有初始化的全局变量或是静态变量，或导致未知错误，影响内核运行。

练习题6

添加的代码如下：

```
void uart_send_string(char *str)
{
    /* LAB 1 TODO 3 BEGIN */
    /* BLANK BEGIN */
    int i = 0;
    for (i = 0; str[i] != '\0'; i++) {
        early_uart_send(str[i]);
    }
    /* BLANK END */
    /* LAB 1 TODO 3 END */
}
```

之后成功显示字符：

A terminal window with a dark background and light-colored text. The prompt is 'cdm@cdm-virtual-machine:~/Desktop/os/OS-Course-Lab\$'. The command 'sudo make qemu' has been executed. The output shows the boot process: 'boot: init_c', '[BOOT] Install kernel page table', '[BOOT] Enable el1 MMU', and '[BOOT] Jump to kernel main'.

```
cdm@cdm-virtual-machine:~/Desktop/os/OS-Course-Lab$ sudo make qemu
boot: init_c
[BOOT] Install kernel page table
[BOOT] Enable el1 MMU
[BOOT] Jump to kernel main
```

练习题7

添加的代码如下：

```
orr    x8, x8, #SCTLR_EL1_M
```

思考题8

1. 优势：由于多级页表允许页表中出现空洞，若某级页表对应的某条目为空，那么该条目对应的下一级页表都无需存在，因此在应用程序的虚拟地址空间大部分都没有分配的情况下，多级页表可以有效压缩页表的大小。
2. 劣势：由于需要在多级页表中进行查找，相比与单级页表速度有所下降。
3. 4KB粒度：4GB的地址范围对应的页的数量是 $4\text{GB}/4\text{KB} = 2^{32}/2^{12} = 2^{20}$ ，在每一级页表中，所对应的下一级 PTE的数量最多为 $4\text{KB}/64\text{bit} = 2^{12}/2^3 = 2^9$ ，所以所需要的 L3级页表的数量为 $2^{20}/2^9 = 2^{11}$ ，需要的 L2级页表的数量为 $2^{11}/2^9 = 2^2$ ，需要的 L1和 L0级页表的数量为1，所以需要的页表总数为 $2^{11} + 2^2 + 1 + 1 = 2054$ ，所占用的物理内存大小为 8216KB

4. 2MB粒度：4GB的地址范围对应的页的数量为 $4\text{GB}/4\text{KB} = 2^{32}/2^{21} = 2^{11}$ ，所以所需要的 L2级页表的数量为 $2^{11}/2^9 = 2^2$ ，需要的 L1和 L0级页表的数量为1，所以需要的页表总数为 $2^2 + 1 + 1 = 6$ ，所占用的物理内存大小为 24KB

思考题9

参考练习题10的代码，L0和L1页表都只有一项，L2页表有 $((\text{PERIPHERAL_BASE} - \text{PHYSMEM_START}) + (\text{PHYSMEM_END} - \text{PERIPHERAL_BASE})) / \text{SIZE_2M} = 512$ 项，总共占用 $(1+1+512) * 2\text{MB}$ 物理内存。

练习题10

添加的代码如下：

```
/* TTBR1_EL1 0-1G */
/* LAB 1 TODO 5 BEGIN */
/* Step 1: set L0 and L1 page table entry */
/* BLANK BEGIN */
vaddr = KERNEL_VADDR + PHYSMEM_START;
boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE
                                     | IS_VALID;
boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) | IS_TABLE
                                     | IS_VALID;

/* BLANK END */

/* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
/* BLANK BEGIN */
for (; vaddr < KERNEL_VADDR + PERIPHERAL_BASE; vaddr += SIZE_2M) {
    boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) | UXN | ACCESSED | NG
        | INNER_SHARABLE | NORMAL_MEMORY | IS_VALID;
}
/* BLANK END */

/* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
/* BLANK BEGIN */
for (vaddr = KERNEL_VADDR + PERIPHERAL_BASE;
     vaddr < KERNEL_VADDR + PHYSMEM_END;
     vaddr += SIZE_2M) {
    boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
        (vaddr - KERNEL_VADDR) | UXN | ACCESSED | NG
        | INNER_SHARABLE | DEVICE_MEMORY | IS_VALID;
}
/* BLANK END */
/* LAB 1 TODO 5 END */
```

思考题11

在启用mmu之后，下一条指令还是在低地址运行，不配置低地址页表就会出现地址翻译出错。进而尝试跳转到异常处理函数（Exception Handler），该异常处理函数的地址为异常向量表基地址（vbar_el1 寄存器）加上 0x200。此时我们没有设置异常向量表（vbar_el1 寄存器的值是0），因此执行流会来到 0x200 地址，此处的代码为非法指令，会再次触发异常并跳转到 0x200 地址。使用 GDB 调试，在 GDB 中输入 continue 后，待内核输出停止后，按 Ctrl-C，可以观察到内核在 0x200 处无限循环。

思考题12

在 _start 函数的 primary 标签处，只有当当前 CPU 的 ID 为 0 时才会执行初始化流程。这意味着只有 0 号核心会在这一阶段执行初始化流程，而其他核心会暂停执行。

其他核心会在 primary 标签处被暂停执行，直到等待 _start 函数中的 init_c 函数执行完毕后，才会由 init_c 函数中的代码唤醒其他核心。在 init_c 函数中 `start_kernel(secondary_boot_flag);` 唤醒其他核心，以便其他核心能够继续执行后续的操作。