Lab1 实验报告

Author: *刘佳隆*

Student ID: *518010910009*

Email: *liujl01@sjtu.edu.cn*

思考题 1

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。

在初始化阶段：

```
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1 // 读取当前核的 ID，并保存到 x8 中
    and x8, x8, #0xFF // 取出核 ID 的低 8 位
    cbz x8, primary // 如果核 ID 为 0，则跳转到 primary 标签
```

`mpidr_el1` 是处理器亲和力寄存器，该寄存器值为该处理单元的另一种标识，表示对应处理器单元的亲和力，该值用于系统调度。此处先读取当前核的 ID，如果核 ID 为 0，则跳转到 `primary` 标签。因此 Chcore 会让核 ID 为 0 的核首先进入初始化流程。

练习题 2

在 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。

```
mrs x9, CurrentEL
```

练习题 3

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。

```
adr x9, .Ltarget
msr elr_el3, x9  // elr: exception link register
mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
msr spsr_el3, x9 // spsr: saved program status register
```

思考题 4

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?

在进入 C 函数之前设置启动栈是为了保证 C 函数能够正常运行。C 函数需要栈保存返回地址、局部变量等。如果不设置启动栈，会导致栈指针指向一个未知的地址，这样 C 函数就无法正常运行。

思考题 5

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。

`.bss` 段用于存储未初始化的全局变量和静态变量。如果不清理 `.bss` ，当使用一些假定初始值为 0 的全局变量和静态变量时，会导致错误的结果。

练习题 6

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。

```
while (*str) {
        early_uart_send(*str);
        str++;
}
```

练习题 7

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。

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

思考题 8

请思考多级页表相比单级页表带来的优势和劣势（如果有的话），并计算在 AArch64 页表中分别以 4KB 粒度和 2MB 粒度映射 0～4GB 地址范围所需的物理内存大小（或页表页数量）。

多级页表相比单级页表的优势：

1. 多级页表可以将内存划分为更小的块，进而在内存分配中减少内存碎片。
2. 页表空间的分配方式比较灵活，可以实现按需分配而不是预先全部分配，从而节约内存空间。

多级页表相比单级页表的劣势：

1. 多级页表的访问时间比单级页表长，因为多级页表需要多次访问内存。
2. 多级页表的页表项数目更多，占用更多的内存空间。
3. 多级页表中的页大小更小，更容易导致 TLB Miss。

4KB：

- 页表页数：$2^{32}/2^{12} = 2^{20}$
- L3 页表页数：$2^{20}/2^{9} = 2^{11}$
- L2 页表页数：$2^{11}/2^{9} = 2^{2}$
- 页表页占用内存：$(2^{11} + 2^{2}) \times 4KB = 8MB$

2MB：

- 页表页数：$2^{32}/2^{21} = 2^{11}$
- L2 页表页数：$2^{11}/2^{9} = 2^{2}$
- 页表页占用内存：$(2^{2}) \times 4KB = 16KB$

思考题 9

请结合上述地址翻译规则，计算在练习题 10 中，你需要映射几个 L2 页表条目，几个 L1 页表条目，几个 L0 页表条目。页表页需要占用多少物理内存?

- L2 页表条目：$2^{31}/2^{12} = 2^{19}$
- L1 页表条目：$2^{19}/2^{9} = 2^{10}$
- L0 页表条目：$2^{10}/2^{9} = 2^{1}$
- 页表页占用内存：$(2^{10} + 2^{1} + 2^{1}) \times 4KB = 4MB$

练习题 10

在 init_kernel_pt 函数的 LAB 1 TODO 5 处配置内核高地址页表（boot_ttbr1_l0、boot_ttbr1_l1 和 boot_ttbr1_l2），以 2MB 粒度映射。

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

/* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
/* BLANK BEGIN */
for (vaddr = PHYSMEM_START; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr) /* low mem, va = pa */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | INNER_SHARABLE /* Sharebility */
                | NORMAL_MEMORY /* Normal memory */
                | IS_VALID;
}
/* BLANK END */

/* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
/* BLANK BEGIN */
for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr) /* low mem, va = pa */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | DEVICE_MEMORY /* Device memory */
                | IS_VALID;
}
/* BLANK END */
/* LAB 1 TODO 5 END */
```

思考题 11

请思考在 init_kernel_pt 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

因为在开启翻译的 `el1_mmu_activate` 函数中，仍需要低地址的虚拟地址与物理地址相同的特性来开启翻译。低地址在这个过程中起到过渡作用，在启动之后这段地址空间会留给用户态。

当将配置低地址页表的代码注释之后， `el1_mmu_activate` 在执行到 `msr sctlr_el1, x8` 时会触发异常，导致内核无法正常启动。

思考题 12

在一开始我们暂停了三个其他核心的执行，根据现有代码简要说明它们什么时候会恢复执行。思考为什么一开始只让 0 号核心执行初始化流程?

在 `_start` 函数中，当核 ID 不为 0 时，会一直循环等待。

```
/* Wait for bss clear */
wait_for_bss_clear:
        adr     x0, clear_bss_flag
        ldr     x1, [x0]
        cmp     x1, #0
        bne     wait_for_bss_clear

...

wait_until_smp_enabled:
        /* CPU ID should be stored in x8 from the first line */
        mov     x1, #8
        mul     x2, x8, x1
        ldr     x1, =secondary_boot_flag
        add     x1, x1, x2
        ldr     x3, [x1]
        cbz     x3, wait_until_smp_enabled
```

根据以上循环判断的代码，当 Core 0 完成 bss 清理和 SMP 启用后，其他核心才会恢复执行。

一开始只让 0 号核心执行初始化流程是为了保证系统的全局状态在所有处理器间是一致的。CPU 0 作为所谓的“bootstrap”处理器或“BSP”（Bootstrap Processor），负责创建一个稳定的、可预测的环境，例如设置内存控制器、初始化中断控制器、构建基本的调度环境等。在其它CPU能够启动之前，需要这些环境已经准备就绪。此外，分步进行初始化可以避免多个CPU同时尝试配置相同资源带来的竞争条件和其他并发问题。