CS3601-Lab1

Q1

Q：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。

首先取出当前CPU的标识到x8

```
mrs x8, mpidr_el1
and x8, x8, #0xFF
cbz x8, primary
```

当x8为0时代表为主核，直接跳转到primary段进入到arm64_elX_to_el1等后续流程中，

而对于其他副核，回继续进入wait_until_smp_enabled段：

```
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
```

反复检查secondary_root_flag中的对应标志，一开始都是0，直到非0才命令副核进入init阶段

因此一开始_start时只有0号主核会直接进入init，其他核暂停执行

Q2

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。

填写代码：

```
mrs x9, CurrentEL
and x9, x9, #0xf
```

(想不出来怎么用一行写完…)

先取CurrentEL到x9，结合前面对于CURRENTEL_EL1的四位二进制0b0100定义，可以知道我们应当取x9的末四位进行进一步比较，于是用#0xf作掩码即可。

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。

填写代码：

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

elr_el3应指向返回ret位置，对应到段落号就是.Ltarget

spsr_el3应设置为EL1h，所以将其设置为SPSR_ELX_EL1H

Q4

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?

C函数处处都需要使用到栈，包括函数参数的保存，临时变量存储。而且本身对其的调用也需要栈来存储返回的地址。

如果不设置栈，sp无法正常返回，指向错误的内存地址，无法正常运行程序。

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。

在实际应用的时候，当内核重启的时候，bss段中可能保存有之前运行残留的数据。这时如果不清理bss段，一些全局变量或者本应该为0的初始变量可能不为0，导致运行出错。

还有一些运行状态有关的变量不为0，可能会导致pc的顺序紊乱，破坏程序逻辑

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。

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

调用前面定义的用于输出单个字符的early_uart_send，使用循环依次输出str中的每个字符即可，内存中的str实际上是指针，这里直接用str迭代就行。

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。

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

其他几位都已经设置好，仿照下面的语句将M为设置为启用即可

Q8

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）。

- 优势：在地址区域稀疏时可以有效节省空间，并且内存利用效率高，都是按需分配
- 劣势：一方面是访问多极页表会增大地址转换的开销，并且在最坏情形下空间开销大大增加

4KB粒度：

总条目数 = 4GB/4KB = 1M L3页表数 = 1M / 512 = 2K L2页表数 = 2K / 512 = 4 L1页表数 = 1 L0页表数 = 1

所以一共需要的页表页数量 = 2048 + 4 + 1 + 1 = 2054

2MB粒度:

无L3

总条目数 = 4GB/2MB = 2K L2页表数 = 2K / 512 = 4 L1页表数 = 1 L0页表数 = 1

所以一共需要的页表页数量 = 4 + 1 + 1 = 6

Q9

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?

空间一共2GB，以2MB粒度映射:

总条目数 = 2GB/2MB = 1K L2页表数 = 1K / 512 = 2 L1页表数 = 1 L0页表数 = 1

占用物理内存 = (2 + 1 + 1) * 4KB = 16KB

Q10

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 */
vaddr = KERNEL_VADDR + PHYSMEM_START;
boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE | IS_VALID | NG;
boot_ttbr1_l1[GET_L1_INDEX(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 < KERNEL_VADDR + PERIPHERAL_BASE; vaddr += SIZE_2M){
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr - KERNEL_VADDR)
                | UXN
                | ACCESSED
                | NG
                | 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
                | DEVICE_MEMORY
                | IS_VALID;
}
/* BLANK END */
/* LAB 1 TODO 5 END */
```

仿照前后，将高位虚拟地址映射到对应的低位地址，并设置各位参数。

Q11

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。

当然是要的，因为我们前面已经打开了MMU，那么后续对于寻址操作来说，就会把地址当作虚拟地址去解析，如果这里不配置页表就会出现错误。

（经过试验，如果不配置低地址页表，在返回_start的时候就会直接错误

Q12

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_kernel(secondary_boot_flag)中会主核会设置secondary_boot_flag[1][2][3]的值让他们恢复执行。

之所以一开始只让0号核心执行初始化流程，应该是为了保证初始化的顺序，如果并行启动的话那么很有可能会出现前面的依赖还未初始化，而后面已经在尝试调用的情况。而且0号核心初始化过的内存管理和中断设置等等都无需重复初始化，后面的核心只需要把自己的特定数据初始化后直接加进来就可以，提高了效率。

陈景韬 523030910028