Lab 2: ChCore Report

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。

`CNTPCT_EL0`寄存器报告当前的计数值。`CNTKCTL_EL1`控制EL0是否可以访问系统定时器[1]。

`_start` 函数的开头通过以下步骤来控制多核处理器的初始化：

1. `mrs x8, mpidr_el1`：这个指令读取**多处理器亲和性寄存器**（`mpidr_el1`）的值，并将结果存储在寄存器 `x8` 中。该寄存器中存储了包含 CPU ID的信息。在多核系统中，每个核心都有一个唯一的 CPU ID。
2. `and x8, x8, #0xFF`：代码对存储在 `x8` 中的值与 `0xFF`（十进制255）进行按位与操作，得到最低的8位bit，也即 CPU ID。
3. `cbz x8, primary`：这个指令检查 `x8` 中的结果（CPU ID）是否为零。如果 CPU ID 为零，则当前是第一个核心，则跳转到标签 `primary`。

对于第一个核心（CPU ID 0）：

- 代码执行主分支（primary），继续进行主要核心初始化。

对于其他所有核心（CPU ID 大于0）：

- `cbz` 不会跳转到 `primary`。
- 这些核心将继续执行 `cbz` 指令后面的代码，具体来说是标有 `wait_for_bss_clear` 的部分。

这种方法让第一个核心（CPU ID 0）先进入主要初始化流程，其他核心则持续等待直到看见 `secondary_boot_flag` 变为 非 0 值.

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。

填写代码为：

```
mrs x9, CurrentEL
```

此代码通过 `CurrentEL` 系统寄存器可获得当前异常级别，然后写入x9寄存器。经验证，此时x9的值为12，表示处在EL3.

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。具体地，我们需要在跳转到 EL1 时暂时屏蔽所有中断、并使用内核栈（`sp_el1` 寄存器指定的栈指针）。

填入的代码为：

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

此代码将跳转地址的链接写入x9寄存器，然后通过 `msr` 写入异常链接寄存器。然后在借x9寄存器来设置保存的程序状态寄存器 `spsr_el3`，以达到暂时屏蔽所有中断，并使用内核栈 `sp_el1` 寄存器指定的栈指针。

```
#define SPSR_ELX_DAIF           (0b1111 << 6)
#define SPSR_ELX_EL1H           (0b0101)
```

这里两个数是用来设置SPSR寄存器的 `DAIF` 和 `M[4:0]` 的，分别对应屏蔽所有中断和切换栈指针[2]。

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？

- 因为 C 语言的函数调用机制依赖于栈。当一个函数被调用时，它的参数、返回地址以及局部变量都会被压入栈中。
- 如果没有设置栈，那么这些数据就无处存放，函数无法正确执行，可能会导致程序崩溃。

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` 段用于存储程序中未初始化的全局变量和静态变量。如果不清零 `.bss` 段，那么这些变量的初始值将是不确定的，可能包含任意的非法数据。由于程序通常会在使用这些变量之前对它们进行初始化，所以可能不影响内核的执行。然而，如果程序使用了未初始化的全局变量和静态变量，那么不清零 `.bss` 段可能会得到错误的值，从而导致内核无法正常工作。

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。

```
early_uart_init();
for(int i = 0; str[i] != '\0'; i++){
  early_uart_send(str[i]);
}
```

首先 `early_uart_init` 初始化 UART ，然后遍历字符串用 `early_uart_send` 函数发送单个字符（也就是输出字符），从而实现发送字符串。

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
```

设置 `SCTLR_EL1` 的 `M` 位，使能 `MMU`[3]。

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）。

- 优势：可以节省大量空间；在物理地址存在大量空洞的情况下，非连续的多级页表可以减少空间不必要的开支。
- 劣势：多级页表增加了访问时间，带来更多的时间开销。
- 4KB：4KB / 64 = 4 * 1024 * 8 / 64 = 512. 最底层的物理页数为 4GB / 4KB = 2^{20} . 所需三级页表数 $2^{20}/512 = 2^{11} = 2048$, 所需二级页表数 2048/512 = 4, 加上一个一级页表和一个零级页表，总共需要 1 + 1 + 4 + 2048 = 2054张页表页数量。所需物理内存为 2054 * 4KB = 8216KB
- 2MB：最底层的物理页数为 4GB / 2MB = 2^{11} . 所需二级页表数 $\lceil 2^{11}/2^{9} \rceil = 4$, 一个一级页表和一个零级页表，总共需要 1 + 1 + 4 = 6 张页表页数量。所需物理内存为 6 * 4KB = 24KB

练习题 9：请在 `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) /* high mem, va - KERNEL_VADDR  = 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 + KERNEL_VADDR; vaddr < KERNEL_VADDR + PHYSMEM_END; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr - KERNEL_VADDR) /* high mem, va - KERNEL_VADDR = 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 */
```

`Vader` 要跳转到高位地址，所以要加上 `KERNEL_VADDR`，然后通过循环时候 step 的大小来控制粒度。需要注意的是

`(vaddr - KERNEL_VADDR) /* low mem, va = pa */`，这里要设置为物理地址（低位地址）。

思考题 10：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。

因为在启动mmu的 `el1_mmu_activate` 函数中设置sctlr_el1后，chcore将使用虚拟地址，然而下一条指令仍位于低地址空间，使得chcore无法继续初始化。

验证：删除掉低地址配置代码后，chcore停止在 `[BOOT] Install kernel page table`, 同时gdb显示进入 `invalidate_cache_all`.

```
os@ubuntu:~/OS-Course-Lab$ make qemu-gdb
boot: init_c
[BOOT] Install kernel page table
```

```
   0x880e8 <invalidate_cache_all+76>       lsl     w12, w8, w5
   0x880ec <invalidate_cache_all+80>       ubfx    w7, w1, #13, #15
   0x880f0 <invalidate_cache_all+84>       lsl     w7, w7, w2
   0x880f4 <invalidate_cache_all+88>       lsl     w13, w8, w2
   0x880f8 <invalidate_cache_all+92>       orr     w11, w10, w9
   0x880fc <invalidate_cache_all+96>       orr     w11, w11, w7
   0x88100 <invalidate_cache_all+100>      dc      isw, x11
   0x88104 <invalidate_cache_all+104>      subs    w7, w7, w13
  >0x88108 <invalidate_cache_all+108>      b.ge    0x880f8 <invalidate_cache_all+92>  // b.tcont
   0x8810c <invalidate_cache_all+112>      subs    x9, x9, x12
   0x88110 <invalidate_cache_all+116>      b.ge    0x880ec <invalidate_cache_all+80>  // b.tcont
   0x88114 <invalidate_cache_all+120>      add     w10, w10, #0x2
   0x88118 <invalidate_cache_all+124>      cmp     w3, w10
   0x8811c <invalidate_cache_all+128>      dsb     sy
   0x88120 <invalidate_cache_all+132>      b.gt    0x880b4 <invalidate_cache_all+24>
   0x88124 <invalidate_cache_all+136>      ic      iallu
   0x88128 <invalidate_cache_all+140>      isb
   0x8812c <invalidate_cache_all+144>      ret
   0x88130 <el1_mmu_activate>              stp     x29, x30, [sp, #-16]!
remote Thread 1.1 In: invalidate_cache_all                                 L??   PC: 0x88108
0x00000000000880a4 in invalidate_cache_all ()
(gdb) si
0x00000000000880a8 in invalidate_cache_all ()
(gdb) si
0x00000000000880ac in invalidate_cache_all ()
(gdb) si 10
0x00000000000880d4 in invalidate_cache_all ()
(gdb) si
0x00000000000880d8 in invalidate_cache_all ()
(gdb) si
0x00000000000880dc in invalidate_cache_all ()
(gdb) si
0x00000000000880e0 in invalidate_cache_all ()
(gdb) si 10
0x0000000000088108 in invalidate_cache_all ()
(gdb) si 10
0x0000000000088108 in invalidate_cache_all ()
(gdb) si 10
0x0000000000088108 in invalidate_cache_all ()
(gdb) a
```

1. https://zhuanlan.zhihu.com/p/453687153 ↩
2. https://blog.csdn.net/weixin_44073864/article/details/111192476 ↩
3. https://blog.csdn.net/jielunqiu/article/details/84939858 ↩