

Chcore Lab-1 机器启动

姓名：李佳鑫 学号：521030910096

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。

解答:

实现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过程:

- 在24-25行代码处，提取出`mpidr_el1` 系统寄存器的[7:0] 位， 获得对应ID
 - `mpidr_el1` 系统寄存器的[7:0] (`aff0`)：表示一个core中的第几个thread
 - 其中值越小，对应的重要性越高
- 在26行，判断是否`aff0`为0，如果为零，则进入初始化过程，即`primary`函数
- 如果不为零，往下继续执行，进入`wait_for_bss_clear`函数
 - 直到进入`primary`函数的core通过`clear_bss()` 去除bass area完成后，继续往下执行
- 在42行，将特征级转为`el1`
- 在45-50行，准备好栈空间和对应的SP指针
- 然后进入`wait_until_smp_enabled`
 - 不断`secondary_boot_flag` 对应位是否为0，为零则等待
 - 由于`secondary_boot_flag` 被初始化为`{NOT_BSS, 0, 0, ...}`
因此在 ID = 0的core 完成一些基本的初始化之前会暂停往下执行程序
- 直到ID = 0的core 完成一些基本的初始化后，对应的`secondary_boot_flag` 位被置为非零，`non-primary core` 对应的程序跳出循环，往下执行，将CPU iD 存入`x0`后，调用`secondary_init_c` 进入他们对应的初始化

`_start` 函数如下:

```
1  /*
2   * Copyright (c) 2023 Institute of Parallel And Distributed Systems (IPADS),
   Shanghai Jiao Tong University (SJTU)
3   * Licensed under the Mulan PSL v2.
4   * You can use this software according to the terms and conditions of the
   Mulan PSL v2.
5   * You may obtain a copy of Mulan PSL v2 at:
6   *   http://license.coscl.org.cn/MulanPSL2
7   * THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF ANY
   KIND, EITHER EXPRESS OR
8   * IMPLIED, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT, MERCHANTABILITY
   OR FIT FOR A PARTICULAR
```

```

9  * PURPOSE.
10 * See the Mulan PSL v2 for more details.
11 */
12
13 #include "consts.h"
14 #include <common/asm.h>
15
16 .extern arm64_elx_to_el1
17 .extern boot_cpu_stack
18 .extern secondary_boot_flag
19 .extern secondary_init_c
20 .extern clear_bss_flag
21 .extern init_c
22
23 BEGIN_FUNC(_start)
24     mrs x8, mpidr_el1
25     and x8, x8, #0xFF
26     cbz x8, primary
27
28     /* Wait for bss clear */
29 wait_for_bss_clear:
30     adr x0, clear_bss_flag
31     ldr x1, [x0]
32     cmp     x1, #0
33     bne wait_for_bss_clear
34
35     /* Set cntkctl_el1 to enable cntvct_el0.
36        * Enable it when you need to get current tick
37        * at EL0, e.g. Running aarch64 ROS2 demos
38     */
39     mov x10, 0b11
40     msr cntkctl_el1, x10 */
41
42     /* Turn to el1 from other exception levels. */
43     bl arm64_elx_to_el1
44
45     /* Prepare stack pointer and jump to C. */
46     mov x1, #INIT_STACK_SIZE
47     mul x1, x8, x1
48     adr     x0, boot_cpu_stack
49     add x0, x0, x1
50     add x0, x0, #INIT_STACK_SIZE
51     mov sp, x0
52
53 wait_until_smp_enabled:
54     /* CPU ID should be stored in x8 from the first line */
55     mov x1, #8
56     mul x2, x8, x1
57     ldr x1, =secondary_boot_flag
58     add x1, x1, x2
59     ldr x3, [x1]
60     cbz x3, wait_until_smp_enabled
61
62     /* Set CPU id */
63     mov x0, x8
64     b     secondary_init_c

```

```

64
65     /* Should never be here */
66     b    .
67
68 primary:
69
70     /* Turn to el1 from other exception levels. */
71     bl   arm64_elx_to_el1
72
73     /* Prepare stack pointer and jump to c. */
74     adr   x0, boot_cpu_stack
75     add   x0, x0, #INIT_STACK_SIZE
76     mov   sp, x0
77
78     b     init_c
79
80     /* Should never be here */
81     b     .
82     END_FUNC(_start)

```

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。

提示：通过 `CurrentEL` 系统寄存器可获得当前异常级别。通过 GDB 在指令级别单步调试可验证实现是否正确。

代码补全:

```

1     /* LAB 1 TODO 1 BEGIN */
2     /* BLANK BEGIN */
3     mrs x9, CurrentEL
4     /* BLANK END */
5     /* LAB 1 TODO 1 END */

```

实现过程:

- 按照提示，`CurrentEL` 系统寄存器可获得当前异常级别
- 利用 `mrs` 指令 将 `CurrentEL` 系统寄存器的值存入到 `x9` 即可

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）。

代码补全:

```

1  adr x9, .Ltarget
2  msr elr_el3, x9
3  mov x9, SPSR_ELX_DAIF | SPSR_ELX_EL1H
4  msr spsr_el3, x9

```

实现过程:

- `elr_el3` 作为异常连接寄存器，控制 `eret` 执行后的PC，因此只需：
 - 将.Ltarget 对应的地址值读出，放入x9
 - 再利用 `msr` 指令，设置 `elr_el3` 寄存器值
- `spsr_el3` 作为保存的程序状态寄存器，控制 `eret` 执行后的程序状态和异常返回后的异常级别
 - 其中：

条件判断相关的域

分类	字段	描述
条件标志位	N	负数标志位。 在结果是有符号的二进制补码情况下，如果结果为负数，则N=1；如果结果为非负数，则N=0。
	Z	零标志位。 如果结果为0，则Z=1;如果结果为非零，则Z=0。
	C	无符号数溢出标志位。 > 对于加法运算，当发生无符号数溢出时，C=1。 > 对于其他指令，C通常不变。
	V	有符号数溢出标志位。 > 对于加减法指令，在操作数和结果是有符号的整数时，如果发生溢出，则V=1;如果无溢出发生，则V=0。 > 对于其他指令，V通常不发生变化。

如cmp指令，tst指令，条件判断时用到这些标志位；还有像sbc等运算指令，会将标志位一并加入运算。

异常掩码标志位

异常掩码标志位	D	调试比特位。使能该比特位可以在异常处理过程中打开调试断点和软件单步等功能。
	A	用来屏蔽系统错误（SError）。
	I	用来屏蔽IRQ中断。
	F	用来屏蔽FIQ中断。

- 1:表示屏蔽了这个异常
- 0: 表示没有屏蔽这个异常

执行状态控制位	SS	软件单步。该比特位为1，说明在异常处理时使能了软件单步功能。
	IL	不合法的异常状态。
	nRW	当前执行模式。 0：处于AArch64执行模式 1：处于AArch32执行模式
	EL	当前异常等级。 0：表示EL0 1：表示EL1 2：表示EL2 3：表示EL3
	SP	选择栈指针寄存器。当运行在EL0时，处理器选择EL0的栈指针，SP_EL0；当处理器运行在其他异常等级时，处理器可以选择使用SP_EL0或者对应的SP_ELn寄存器。

部分字段有专门对应的寄存器，如NZCV寄存器、CurrentEL寄存器。注意：EL0级别只能访问PSTATE.{N, Z, C, V}和PSTATE.{D, A, I, F}字段。

- 根据提示：在跳转到 EL1 时暂时屏蔽所有中断、并使用内核栈和给定的SPSR_ELX_DAIF | SPSR_ELX_EL1H，只需：
- 将该寄存器的值设置为 SPSR_ELX_DAIF | SPSR_ELX_EL1H

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？

解答：

因为C函数是由C语言编写的，里面涉及很多的函数调用,需要栈对返回地址、寄存器状态进行存储。

如果不设置栈，函数调用前的状态不能得到保存，x30等寄存器的值可能会被覆盖，导致不能回到函数调用前的位置。

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。

由于.bss段用于存储未初始化的全局变量和静态变量，

如果不清零，那么未初始化的全局变量和静态变量的值将是不确定的，不再是0，在内核使用未初始化的全局变量和静态变量时，将无法正常工作。

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。

代码补全:

```
1 |     long i = 0;
2 |     while(str[i] != '\0')
3 |     {
4 |         early_uart_send(str[i]);
5 |         i ++;
6 |     }
```

实现过程:

- 已经有输出一个字符的函数 `early_uart_send(char);`
- 利用循环判断 字符串达到结尾 `\0` , 不断调用 上述函数即可

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。

代码补全:

```
1 |     orr     x8, x8, 1
```

实现过程:

- 根据提示: 在 EL1 异常级别启用 MMU 是通过配置系统寄存器 `sctlr_el1` 实现的, 其中:
 - 是否启用 MMU (M 字段)
 - 是否启用对齐检查 (A SA0 SA nAA 字段)
 - 是否启用指令和数据缓存 (C I 字段)
 - 且查阅资料直到: M 字段是在第0位
- 因此只需将 `x8` 寄存器的第0位置为1, 在后续代码中写入到 `sctlr_el1` 即可

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) 。

解答:

多级页表相对于单级页表:

- 优势:
 - 压缩了页表的大小, 当虚拟地址范围较大时, 大大节省页表所占空间
 - 能够支持大页, 在用TLB进行缓存时进一步提高效率
- 劣势:
 - 增加了访存次数
 - 当页表存满时, 存储相同地址数目的映射, 多级页表所占空间更大

(1) 以4KB粒度来映射 0 ~ 4GB地址:

总共有: $2^{32}/2^{12} = 2^{20}$ 个页

一个页有: $2^{12}/2^3 = 2^9$ 个页表项

3级页表数: $2^{20}/2^9 = 2^{11}$ 个

2级页表数: $2^{11}/2^9 = 4$ 个

1级页表: 1个

0级页表: 1个

总共页表数量: $1 + 1 + 4 + 2^{11}$ 个

物理内存: $(2^{11} + 6) * 2^{12} B$

(2) 以2MB粒度来映射 0 ~ 4GB地址:

总共有: $2^{32}/2^{21} = 2^{11}$ 个页

2级页表: $2^{11}/2^9 = 4$ 个

1级页表、0级页表: 1个

总共页表数量: $1 + 1 + 4 = 6$ 个

物理内存: $6 * 4KB = 24KB$

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。

补全代码:

```
1  /* Step 1: set L0 and L1 page table entry */
2  /* BLANK BEGIN */
3  vaddr = 0xffffffff0000000000;
4
5  boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE
6                                     | IS_VALID | NG;
7  boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) | IS_TABLE
```

```

8                                     | IS_VALID | NG;
9     /* BLANK END */
10
11     /* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
12     /* BLANK BEGIN */
13     for (; vaddr < PERIPHERAL_BASE + 0xffffffff0000000000; vaddr += SIZE_2M) {
14         boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
15             (vaddr - 0xffffffff0000000000) /* high mem */
16             | UXN /* Unprivileged execute never */
17             | ACCESSED /* Set access flag */
18             | NG /* Mark as not global */
19             | INNER_SHARABLE /* Sharebility */
20             | NORMAL_MEMORY /* Normal memory */
21             | IS_VALID;
22     }
23
24     /* BLANK END */
25
26     /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
27     /* BLANK BEGIN */
28     for (vaddr = PERIPHERAL_BASE + 0xffffffff0000000000; vaddr < PHYSMEM_END +
29         0xffffffff0000000000; vaddr += SIZE_2M) {
30         boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
31             (vaddr - 0xffffffff0000000000) /* high mem */
32             | UXN /* Unprivileged execute never */
33             | ACCESSED /* Set access flag */
34             | NG /* Mark as not global */
35             | DEVICE_MEMORY /* Device memory */
36             | IS_VALID;
37     }

```

实现过程：

- 第一步：设置L1、L0页表对应的入口
 - 初始化 vaddr 为 高位虚拟地址的起始 0xffff_0000_0000_0000 + 0x00000000
 - 分别使用 GET_L0_INDEX(vaddr)、GET_L1_INDEX(vaddr) 来提取虚拟地址的L0、L1页表的对应页索引，将对应的页表项设置为下一个页表项的地址，并设置好对应表示的 IS_TABLE、IS_VAILD、NG 字段
- 第二步：设置L2 页表对应的每个页表项的值
 - 利用for循环一次将对应的虚拟地址对应的页表项设置为对应虚拟地址 - 0xffff_0000_0000_0000，并设置好对应的属性字段
 - 这里需要分成两次循环，分别对物理内存（SDRAM）、共享外设内存进行映射，**因为对应的属性字段不同**
 - 具体的属性字段可以参考低地址映射的属性字段
- 对应最后的本地（每个 CPU 核独立）外设内存，代码框架已经给出

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 `el1_mmu_activate` 后，CPU 会从使用物理内存地址转化到使用虚拟内存地址，但是在 `start_kernel` 之前，依旧使用的低地址，如果没有配置低地址的虚拟内存映射，会导致这一阶段的虚拟内存地址没有物理地址对应。并且在 `start_kernel` 过程中的一段时间内，许多寄存器（比如 `x30`）存的地址依旧是低地址，依旧需要低地址的虚拟内存映射。

验证如下：

- 运行代码，观察指令的地址变化，发现确实在 `start_kernel` 之前，依旧使用的是低地址
- 同时在 `start_kernel` 过程中的一段时间内，许多寄存器（比如 `x30`）存的地址依旧是低地址

```
os@ubuntu: ~/OS-Course-Lab
[ No Source Available ]

0x88454 <secondary_init_c+12> mov     w19, w0
0x88458 <secondary_init_c+16> bl     0x88130 <el1_mmu_activate>
0x8845c <secondary_init_c+20> mov     w0, w19
>0x88460 <secondary_init_c+24> ldr     x19, [sp, #16]
0x88464 <secondary_init_c+28> ldp     x29, x30, [sp], #32
0x88468 <secondary_init_c+32> b       0x80088 <__secondary_cpu_boot>

remote Thread 1.2 In: secondary_init_c L?? PC: 0x88460
0x0000000000008845c in secondary_init_c ()
1: $x30 = 558172
2: /x $x30 = 0x8845c
0x00000000000088460 in secondary_init_c ()
1: $x30 = 558172
2: /x $x30 = 0x8845c
(gdb)

os@ubuntu: ~/OS-Course-Lab
[ No Source Available ]

0xfffff000008e018 <start_kernel+24> isb
0xfffff000008e01c <start_kernel+28> bl     0xfffff00000
>0xfffff000008e020 <start_kernel+32> ldp     x0, x1, [sp],
0xfffff000008e024 <start_kernel+36> bl     0xfffff00000
0xfffff000008e028 <secondary_cpu_boot> mov     x19, x0
0xfffff000008e02c <secondary_cpu_boot+4> mov     x1, #0x1000

remote Thread 1.1 In: start_kernel L?? PC: 0xfffff000008e020
0xfffff000008e01c in start_kernel ()
1: $x30 = 524384
2: /x $x30 = 0x80060
0xfffff000008e020 in start_kernel ()
1: $x30 = -1099511046112
2: /x $x30 = 0xfffff000008e020
(gdb) S
```

- 当去掉低地址配置页表，程序运行失败

