ChCore 实验 1

班级：F2103003 姓名：黄伊新 学号：521030910070

1. **思考题：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 start 函数中，首先 cpu 将存在 mpidr_el1 中的含有 cpu id 信息的数读到 x8 寄存器中，并通过 and 指令滤掉其他的多余信息，这样存在 x8 中的就是 cpu id 这一项内容了。然后通过 cbz 指令，只允许 id 为 0 的 cpu 核跳转到 primary 处继续执行初始化流程，其他核则往下执行 wait_for_bss_clear 函数，该函数对某个地址进行读取，直到被设为 0 后才会接着往下执行，这样子的效果就是其他核暂停执行了。

2. **练习题：写一行汇编代码，获取 CPU 当前异常级别**

   CurrentEL 寄存器中存储了现在的异常级别，通过 mrs 读取到通用寄存器 x9 中即可。

3. **练习题：设置从 EL3 跳转到 EL1 所需的 elr_el3 和 spsr_el3 寄存器值**

   首先设置 elr_el3 的值，他指向的是 eret 后返回的地址。由于后续希望该函数执行完成后能返回到 start 函数，故设置 elr_el3 的值为.Ltarget 的地址值，.Ltarget 后直接 ret 可以返回回 start。.Ltarget 的地址值通过 adr 指令读取到 x9 寄存器中。

   接着设置 spsr_el3 寄存器的值。本处的要求是“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）”。

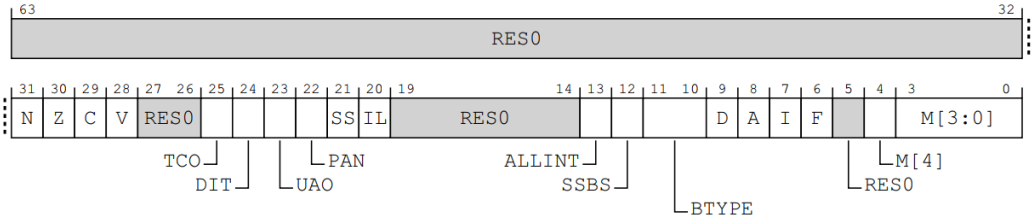

图 1: SPSR 寄存器

图 1 展示了 SPSR 寄存器的每一位对应的内容。其中 DAIF 四位主要控制的是是否允许中断，那么根据要求本处四位全部设置为 1，这样就可以屏蔽所有的中断。M[4] 设置为 0 保证是在 AArch64 execution state。最后四位指示的是 eret 后的特权级，本处根据目标选择 EL1，故前两位为 01，第三位无用设置为 0，且跳转后使用 sp_el1 寄存器，故最后一位设置为 1。本处通过使用预先设置好的 SPSR_ELX_DAIF 核 SLSR_ELX_EL1H 两个宏来进行设置。

4. **思考题：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？**

   可以看到在 C 函数的 19 行，C 代码声明了一个栈，这也就意味着 C 函数是默认已经有栈分配给他了。操作系统只能自己给自己分配栈，为了程序正确运行，所以需要在进入 C 函数以前就启动栈，否则 C 函数在代码中涉及到栈的有关部分就会出现问题，比如访问未定义的区域、所存地址未被保护被其他程序修改等。

5. **思考题：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 段存储的是未初始化的全局变量和静态变量，而 C 语言默认是他们被初始化为 0 了。如果我们没有执行 clear_bss 且 C 语言由于默认已初始化的缘故直接对某些变量进行处理，那么就会出现问题，和预期结果不符，内核的一些工作可能也就无法顺利进行了。

6. **练习题：实现通过 UART 输出字符串的逻辑**

   该函数输入为需要输出的字符串，而 early_uart_send 函数则是输出一个字符，故我们只需要遍历传入的字符串并依次调用 early_uart_send 函数即可。

7. **练习题：填写一行汇编代码，以启用 MMU**

   由于我们是需要启动 MMU，所以使用的是 orr 代码，让 M 位置 1，#SCTLR_EL1_M 控制的就是具体操作的位置。

8. **思考题：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）**

   多级页表相比单极页表的优点在于它允许虚拟页表中有“空洞”存在，在大概率的事件下他的内存占用会比单极页表小得多。但是多级页表由于是逐级查询的，增加了访存次数；同时他需要同时维护好几个页表；在虚拟内存占用比较满的时候，多级页表会多出查询的几个页表，此时多级页表是比单极页表占的内存多。

   首先计算 4KB 粒度。共需要最终映射 2^{20} 个页，一个三级页表可以映射 2^9 个页，那么就至少需要 2^{11} 个三级页表，另外还需要再配备 4 个二级页表，一个一级页表和一个零级页表，总共最终需要 2054 个页表。一个页表大小为 4K，那么最终就需要 8216K 的物理内存。

   接着是 2MB 粒度。共需要最终映射 2^{11} 个页，那么就需要 4 个二级页表来完成映射，还需要再配备一个一级页表和一个零级页表，共 6 个页表，24K 物理内存。

9. **练习题：配置内核高地址页表**

本处操作仿照在低地址处的操作，但是相比低地址，需要给所有的虚拟地址加上 offset 0xffffff0000000000UL，同时写入的页表设为 EL1 级别的页表。同时，考虑到是将虚拟地址 addr + 0xffffff0000000000UL 的映射到 addr，在页表项中填写对应的物理地址时需要减去 0xffffff0000000000UL。

10. **思考题：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**

    在启用 MMU 以后所有的地址都会被当作虚拟地址来对待，但是开启后由于本身物理地址是在低地址区的，读取下一条代码时会直接调用低地址对应的页表来翻译，若不配置低地址页表会使得在开启 MMU 后无法正常执行下一条指令从而报错，直到设置转为高地址后才会停止使用低地址的页表。

    测试将低地址的页表配置删除后，gdb 显示 Cannot access memory at address 0x200，符合上面关于为何要配置低地址的阐述。此处就是首先由于没有配置页表，无法访问下一条指令对应的地址位置，调用异常处理函数，但是由于我们异常处理函数没有设置，默认指向 0x200，那么这个时候由于 0x200 也是属于低地址范畴的，也无法访问，所以是报这个错。