OS-Course-Lab1 机器启动

> **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。

ChCore 首先从 `mpidr` 寄存器读取当前处理器核 id，如果为 0 跳转到 primary，否则往下执行，进入等待循环。

> **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。

`CurrentEL` 寄存器保存了当前 CPU 的异常级别，可以通过 `mrs` 指令读取该寄存器的值。根据后续代码，我们将其读取到寄存器 `x9` 中。

> **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。

`elr_el3` 寄存器保存了从 EL3 跳转到下一个异常级别时的返回地址，阅读后续代码可知，异常级别设置完成后，通过 `eret` 指令跳转到该地址。因此，我们需要将目标地址（即标签 `.Ltarget` 的地址）加载到 `elr_el3` 寄存器中。

`spsr_el3` 寄存器保存了从 EL3 跳转到下一个异常级别时的程序状态寄存器值。我们需要将 `SPSR_ELX_DAIF`（表示禁用中断）和 `SPSR_ELX_EL1H`（表示进入 EL1 ）进行异或运算，然后将结果写入 `spsr_el3` 寄存器中。

> **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？

不设置启动栈，会导致发生异常时，CPU 无法正确保存返回地址和局部变量，从而引发不可预期的行为或系统崩溃。

> **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。

如果不清理 `.bss` 段，未初始化的全局变量和静态变量可能包含随机值，这可能导致内核在访问这些地址时出现不可预期的行为，进而影响系统的稳定。

> **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。

用 `early_uart_init` 函数初始化 UART 硬件，然后通过循环遍历字符串的每个字符，调用 `early_uart_send` 函数将字符发送到 UART 进行输出。

> **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。

使用 `orr` 指令将 `SCTLR_EL1_M` 标志位设置到 `SCTLR_EL1` 寄存器中，从而启用 MMU。

> **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）。

在 AArch64 中，使用多级页表的主要优势在于可以有效地管理大内存空间，同时减少页表的内存占用。多级页表通过将虚拟地址空间分成多个层级，可以在需要时动态分配页表项，从而避免了单级页表在映射大内存时需要占用大量连续物理内存的缺点。
以 4KB 粒度映射 0 ~ 4GB 地址范围：

- 每个页表项映射 4KB 内存。
- $4\text{GB} \times 4\text{KB} = 1,048,576$ 个页表项，共需要 $1048576 \times 512 = 2048$ 个 L3 页表页（每个页表页包含 512 个页表项）。
- 需要 4 个 L2 页表页，1 个 L1 页表页，1 个 L0 页表页，总共需要 2054 个页表页。
  以 2MB 粒度映射 0 ~ 4GB 地址范围：
- 每个页表项映射 2MB 内存。
- $4\text{GB} \times 2\text{MB} = 2048$ 个块，共需要 $2048 \times 512 = 4$ 个 L2 页表页。
- 需要 1 个 L1 页表页，1 个 L0 页表页，总共需要 6 个页表页。

> **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?

对于 `0x00000000 - 0x3FFFFFFF` 范围：

- L2 页表条目数：$1\text{GB}/2\text{MB} = 512$ 个条目，需要 1 个 L2 页表页。
- L1 页表条目数：1 个条目。
- L0 页表条目数：1 个条目。

对于 `0x40000000 - 0x7FFFFFFF` 范围：

- L1 页表条目数：1 个条目。
- L0 页表条目数：1 个条目。

综上，总共需要 1 个 L2 页表页，2 个 L1 页表页，2 个 L0 页表页。每个页表页大小为 4KB，因此总共需要占用 20KB 的物理内存。

> **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。

仿照低地址进行设置，注意偏移量 `KERNEL_VADDR`。

> **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。

在后续 `el1_mmu_activate` 函数中，需要读取 `ttbr0_el1` 寄存器来启用 MMU，而 `ttbr0_el1` 指向的是低地址页表。因此，必须为低地址配置页表以确保 MMU 能够正确工作。

> **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?

当 0 号核心完成执行完时钟，调度器，锁的初始化，重置 `secondary_boot_flag` 后恢复执行。
这是因为 0 号核心负责初始化系统的关键组件，为其他核心的启动准备好资源后，其他核心才能安全地加入执行，避免内核混乱。