实验 1：机器启动

薛翔元 (521030910387)

> 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。

`_start` 的前三行代码如下所示。

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

该段代码读取 `mpidr_el1` 寄存器的低 8 位并存入 `x8` 寄存器中，查阅文档可知该字段为 `Aff0`，表示多核处理器中的核心编号，因此仅有 0 号核心满足跳转条件，进入 `primary` 函数进行初始化，其余核心则依次进入 `wait_for_bss_clear` 和 `wait_until_smp_enabled` 被暂时挂起，等待基本初始化完成后继续执行。

> 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。

填写代码如下所示。

```
mrs x9, CurrentEL
```

该行代码使用 `mrs` 指令将 `CurrentEL` 寄存器的值存入 `x9` 寄存器中，与下文中 `cmp` 指令保持一致。

使用 GDB 单步调试，可以看到当前异常级别已经被正确获取。

```
(gdb) break arm64_elx_to_el1
Breakpoint 1 at 0x88000
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, 0x0000000000088000 in arm64_elx_to_el1 ()
(gdb) ni
0x0000000000088004 in arm64_elx_to_el1 ()
(gdb) print $x9
$1 = 12
```

> 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）。

填写代码如下所示。

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

该段代码首先将 `elr_el3` 寄存器的值设置为 `x9` 寄存器的值设置为 `.Ltarget` 符号的地址，表示执行 `eret` 指令后跳转到 `.Ltarget` 符号处，从 `arm64_elX_to_el1` 函数中返回，然后设置 `spsr_el3` 寄存器的 `DAIF` 和 `EL1H` 字段，前者表示屏蔽所有中断，后者表示使用内核栈。

使用 GDB 单步调试，可以看到已经正确返回到 `_start` 函数。

```
(gdb) ni
0x0000000000088098 in arm64_elX_to_el1 ()
0x0000000000080060 in _start ()
0x0000000000080064 in _start ()
0x0000000000080068 in _start ()
```

> 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？

在 C 语言中，局部变量和函数调用都是通过栈来实现的，因此栈的存在是 C 程序能够正确运行的基本条件。

如果不设置栈，则 C 函数调用时将无法正确保存上下文，也无法正确传参和返回，此外，所有局部变量将不可用。

> 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。

C 语言规定，未初始化的全局变量和静态变量会被清零，因此 `clear_bss` 保证了 C 程序的正确性。

1. 进入 `init_c` 函数之前，`.bss` 段相应内存区域已被修改为非零值。
2. 此后，C 程序使用了未初始化的全局变量或静态变量。
3. C 程序的正确性依赖于这些变量的零初始值。

当以上条件均满足时，不清理 `.bss` 段将导致内核无法正确工作。

> 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。

实现代码如下所示。

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

首先调用 `early_uart_init` 函数初始化 UART，然后顺序遍历字符串，调用 `early_uart_send` 函数输出每个字符。

至此，ChCore 内核已经可以正确输出字符串。

```
$ make qemu
boot: init_c
[BOOT] Install kernel page table
[BOOT] Enable el1 MMU
[BOOT] Jump to kernel main
```

> 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。

填写代码如下所示。

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

该行代码使用 `orr` 指令修改 `x8` 寄存器的值，从而设置 `sctlr_el1` 寄存器的 `M` 字段，表示启用 MMU，在 GDB 中可以观察到内核在 `0x200` 处无限循环。

```
(gdb) continue
Continuing.

Thread 1 received signal SIGINT, Interrupt.
0x0000000000000200 in ?? ()
```

> 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）。

多级页表被切分在若干等长的页表空间，并以链表形式索引，这让页表在内存中可以离散存储。多级页表允许页表中出现空洞，有效利用了虚拟地址空间的稀疏性，节省了页表的内存占用。然而，多级页表增加了访存次数，提高了地址翻译的时间开销。

以 $4\mathrm{KB}$ 粒度映射 $4\mathrm{GB}$ 地址空间，需要 $2^{32}/2^{12}=2^{20}$ 个物理页，对应 L3 页表 $2^{20}/2^{9}=2^{11}$ 个，L2 页表 $2^{11}/2^{9}=4$ 个，以及 L1 和 L0 页表各 1 个，共计 $2^{11}+4+1+1=2054$ 个页表页，占用物理内存 $2054\times 4\mathrm{KB}=8216\mathrm{KB}\approx 8\mathrm{MB}$。

以 $2\mathrm{MB}$ 粒度映射 $4\mathrm{GB}$ 地址空间，需要 $2^{32}/2^{21}=2^{11}$ 个物理页，对应 L2 页表 $2^{11}/2^{9}=4$ 个，以及 L1 和 L0 页表各 1 个，共计 $4+1+1=6$ 个页表页，占用物理内存 $6\times 4\mathrm{KB}=24\mathrm{KB}$。

> 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。

实现代码如下所示。

```
/* Step 1: set L0 and L1 page table entry */
vaddr = PHYSMEM_START + KERNEL_VADDR;
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;

/* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
for (; vaddr < PERIPHERAL_BASE + KERNEL_VADDR; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr - KERNEL_VADDR) /* high mem, va = pa + offset */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | INNER_SHARABLE /* Sharebility */
                | NORMAL_MEMORY /* Normal memory */
                | IS_VALID;
}

/* Step 3: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
for (vaddr = PERIPHERAL_BASE + KERNEL_VADDR; vaddr < PHYSMEM_END + KERNEL_VADDR;
vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr - KERNEL_VADDR) /* high mem, va = pa + offset */
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | DEVICE_MEMORY /* Device memory */
                | IS_VALID;
}
```

由于此处对高地址进行映射，虚拟地址需要在物理地址上增加 `KERNEL_VADDR` 的偏移量，然后按照 `boot_ttbr1_l0`、`boot_ttbr1_l1` 和 `boot_ttbr1_l2` 的顺序配置页表，其中 `0x00000000` 到 `0x3f000000` 设置 `NORMAL_MEMORY` 字段，`0x3f000000` 到 `0x40000000` 设置 `DEVICE_MEMORY` 字段。

至此，ChCore 内核已经可以正确进入 `main` 函数完成启动流程。

```
[INFO] [ChCore] uart init finished
[INFO] [ChCore] per-CPU info init finished
[INFO] [ChCore] vc_mem: 0x3c000000, size: 0x4000000
[INFO] [ChCore] physmem_map: [0x5f5000, 0x3c000000)
[INFO] [ChCore] mm init finished
[INFO] CPU freq 62500000, set timer 625000
[INFO] [ChCore] interrupt init finished
[INFO] [ChCore] pmu init finished
[INFO] [ChCore] sched init finished
[INFO] CPU 0 is active
[INFO] CPU 1 is active
[INFO] CPU freq 62500000, set timer 625000
[INFO] CPU freq 62500000, set timer 625000
[INFO] CPU 2 is active
[INFO] CPU 3 is active
[INFO] All 4 CPUs are active
[INFO] CPU freq 62500000, set timer 625000
[INFO] [ChCore] boot multicore finished
```

```
[INFO] [ChCore] create initial thread done
[WARN] SYS_rt_sigprocmask is not implemented.
User Init: booting fs server (FSMGR and real FS)
[procmgr] register server value = 0
[procmgr] Launching tmpfs...
[WARN] SYS_rt_sigprocmask is not implemented.
[procmgr] Launching fsm...
User Init: booting network server
[fsm] Mounting fs from local binary: /tmpfs.srv...
[fsm] TMPFS is up, with cap = 11
[tmpfs] register server value = 0
[WARN] SYS_rt_sigprocmask is not implemented.
[fsm] [FSM] register server value = 0
[procmgr] Launching lwip...
[procmgr] Launching chcore-shell...
load library name:/lwip.srv
load library name:chcore_shell.bin
map library base:0x737d40822000
[procmgr] Launching usb_devmgr...
[procmgr] Launching network-cp.service...
load library name:/network-cp.service
map library base:0x713999992000
setup_peripheral_mappings done.
load library complete
No USB support.
[WARN] SYS_rt_sigprocmask is not implemented.
load library complete
[WARN] SYS_rt_sigprocmask is not implemented.
[WARN] SYS_membarrier is not implmeneted.
map library base:0x779446832000
load library complete

[lwip] Host at 192.168.0.3 mask 255.255.255.0 gateway 192.168.0.1
 ______     __  __     ______     __  __     ______     __         __
/\  ___\   /\ \_\ \   /\  ___\   /\ \_\ \   /\  ___\   /\ \       /\ \
\ \ \____  \ \  __ \  \ \___  \  \ \  __ \  \ \ \__ \  \ \ \____  \ \ \____
 \ \_____\  \ \_\ \_\  \/\_____\  \ \_\ \_\  \ \_____\  \ \_____\  \ \_____\
  \/_____/   \/_/\/_/   \/_____/   \/_/\/_/   \/_____/   \/_____/   \/_____/


Welcome to ChCore shell!
$ [WARN] SYS_rt_sigprocmask is not implemented.
[WARN] SYS_membarrier is not implmeneted.
[lwip] TCP/IP initialized.
[lwip] Add netif 0x58c7effbeed0
[lwip] register server value = 0
Network-CP-Daemon: running at localhost:4096
```

> 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。

开启 MMU 之前，内核直接运行在低地址，设置 `sctlr_el1` 寄存器开启地址翻译后，`el1_mmu_activate` 函数仍留在低地址空间，因此在 `init_kernel_pt` 函数中必须为低地址配置页表，否则后续指令无法正确寻址。

暂时删去 `init_kernel_pt` 函数中配置低地址页表的相关代码，然后使用 GDB 单步调试 `el1_mmu_activate` 函数，观察其执行过程。

```
(gdb) break el1_mmu_activate
Breakpoint 1 at 0x88138
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, 0x0000000000088138 in el1_mmu_activate ()
(gdb) ni 20
0x0000000000088188 in el1_mmu_activate ()
(gdb) ni
0x000000000008818c in el1_mmu_activate ()
0x0000000000088190 in el1_mmu_activate ()
0x0000000000088194 in el1_mmu_activate ()
Cannot access memory at address 0x88130
Cannot access memory at address 0x88194
Cannot access memory at address 0x88194
```

可以看到，在 MMU 开启以后立即发生寻址错误，这一现象验证了上述解释。