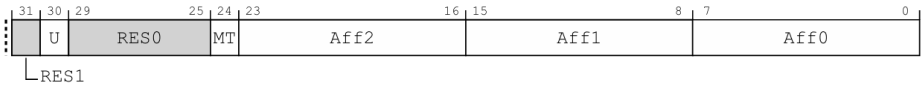
思考题 1：_start 函数开头三行如下：

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

该段代码首先将 mpidr_el1 寄存器放入 x8 寄存器，随后读取其低 8 位。查阅资料，其低 8 位存放的是 Aff0，也就是 0 号 CPU 核心的编号，0 号核因满足跳转条件进入 primary 函数进入初始化流程，而其它核继续往下运行，进入 wait_for_bss_clear 函数和 wait_until_smp_enabled 函数循环，等待 0 号核初始化流程基本结束再进行其它核的初始化。



练习题 2：填写代码如下：

```
mrs x9, CurrentEL
```

该代码将 CurrentEL 寄存器的值存入 x9 寄存器中，以便后续使用 x9 寄存器判断所处的异常级别。通过 GDB 调试，在代码执行后读取 x9 的值为 12，查阅手册，发现 x9 寄存器已经正确地存储了当前的异常级别 EL3。

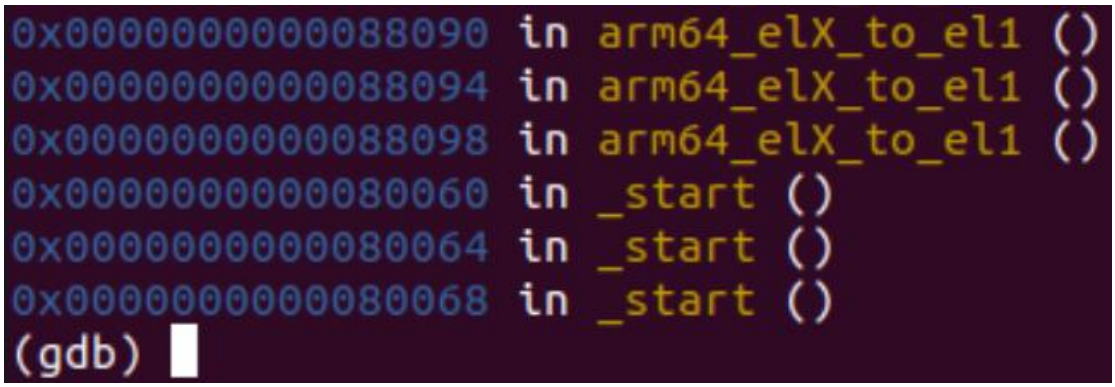
0b00	EL0.
0b01	EL1.
0b10	EL2.
0b11	EL3.

```
Thread 1 hit Breakpoint 1, 0x0000000000088000 in arm64_elX_to_el1 ()
(gdb) ni
0x0000000000088004 in arm64_elX_to_el1 ()
(gdb) print $x9
$1 = 12
(gdb)
```

练习题 3：填写代码如下：

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

该段代码将.Ltarget 的地址存入 x9 寄存器，表示执行 eret 后跳转的地址，从 arm64_elX_to_el1 函数返回到_start，SPSR_ELX_DAIF 屏蔽了 DAIF 的 4 类中断，将 EL1h 的值通过 x9 存入 spsr_el3 寄存器中。通过 GDB 调试，发现可以从函数回到_start 函数。



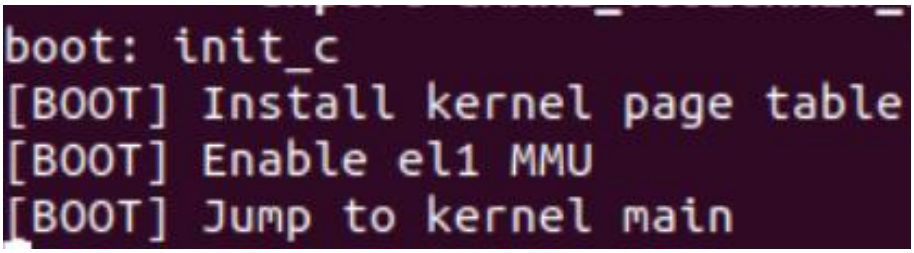
思考题 4：由于 C 函数的运行需要栈来存储返回地址和保存临时变量等，不设置栈则会导致如 sp 从 0 开始往下减等错误，导致 C 函数无法运行报错。

思考题 5：如果不清零.bss 段，则可能出现部分全局或静态变量未被初始化为 0，而当 C 函数期望这些变量初始值为 0 时，实际的值并非为 0，这可能会导致内核无法工作。

练习题 6：输入代码如下：

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

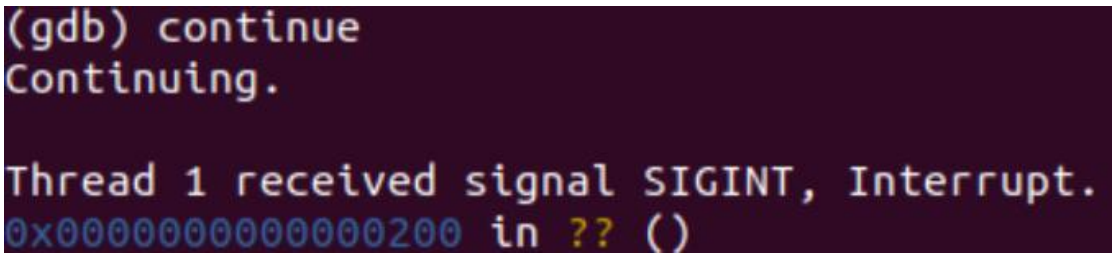
该函数遍历字符串中的所有字符，调用 early_uart_send 函数进行输出。



练习题 7：填写代码如下：

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

该端代码通过 x8 的 orr 操作开启了 MMU，通过 GDB 调试发现在 0x200 处进行循环。



思考题 8：多级页表相比于单极页表，多级页表在稀疏的情况下所需的空间较小，但用到了多级翻译导致查找所需的时间较长。

4KB 进行映射，需要$\frac{2^{32}}{2^{12}}=2^{20}$个物理页，需要$\frac{2^{20}}{2^{9}}=2^{11}$个 L3 页表，$\frac{2^{11}}{2^{9}}=2^{2}$个 L2 页表，1 个

L1 和 1 个 L0 页表，共需要$2^{11}+4+1+1=2054$个页表页。

同理，用 2MB 进行映射，需要$\frac{2^{32}}{2^{21}}=2^{11}$个物理页，4 个 L2 页表，1 个 L1 和 1 个 L0 页表，用需要$4+1+1=6$个页表页。

思考题 9：在练习题 10 中，需要以 2MB 块颗粒度映射 0x0 到 0x40000000 的地址，需要512个物理页，1 个 L2 页表，1 个 L1 页表，1 个 L0 页表。一共需要 3 个页表页，占用物理内存12KB。

练习题 10：填写代码如下：

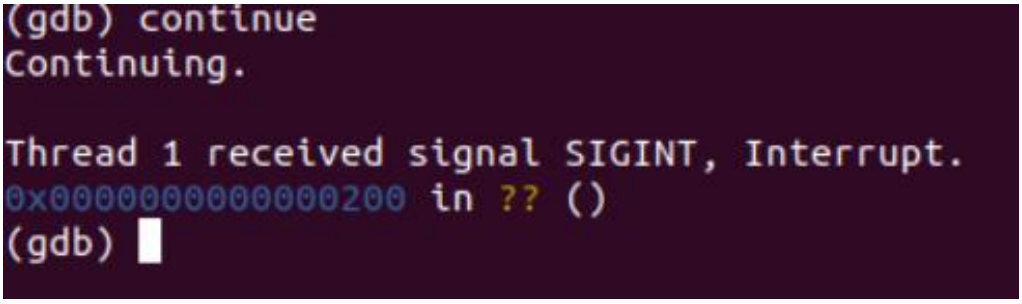
```
/* TTBR1_EL1 0-1G */
/* LAB 1 TODO 5 BEGIN */
/* Step 1: set L0 and L1 page table entry */
/* BLANK BEGIN */
vaddr = KERNEL_VADDR + PHYSMEM_START;
boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE | IS_VALID;
boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) | IS_TABLE | IS_VALID;
/* 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)
                | UXN
                | ACCESSED
                | NG
                | INNER_SHARABLE
                | NORMAL_MEMORY
                | IS_VALID;
}
/* BLANK END */

/* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
/* BLANK BEGIN */
for (vaddr = KERNEL_VADDR + PERIPHERAL_BASE; vaddr < KERNEL_VADDR + PHYSMEM_END; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr - KERNEL_VADDR)
                | UXN
                | ACCESSED
                | NG
                | DEVICE_MEMORY
                | IS_VALID;
}
/* BLANK END */
/* LAB 1 TODO 5 END */
```

通过分别对 0x00000000-0x3f000000 和 0x3f000000-0x40000000 设置不同的字段并以 2MB 块颗粒度进行映射完成对高地址页表的配置。完成页表配置后再次运行即可进入 main 函数

思考题 11：因为在启用 MMU 的时刻，它的下一条指令还是在低位运行，如果不对 MMU 进行配置，则会跳到 0x200 进行循环导致后续指令无法执行。尝试删除配置低地址的代码，然后进行调试，结果如下：



思考题 12：只有在 BSS 段初始化完成且 main 函数执行完时钟，调度器，锁的初始化后设置了 secondary_boot_flag 状态时才会恢复执行。通过优先让 0 号核初始化了关键的系统资源，确保基本环境就绪以避免多核心同时修改资源导致的不一致性冲突造成核之间的相互干扰。