lab4_report

思考题 1：阅读汇编代码 kernel/arch/aarch64/boot/raspi3/init/start.S 。说明 ChCore是如何选定主CPU，并阻塞其他CPU的执行的。

选定主CPU：

```
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    and x8, x8, #0xFF
    cbz x8, primary
```

可以看出，ChCore先读取寄存器`mpidr_el1`的值，并取其后8位进行判断：如果为0，则说明是主CPU，那么跳转到`primary`处执行代码；如果不是0，则说明为其他CPU。

阻塞其他CPU：

```
wait_for_bss_clear:
    adr x0, clear_bss_flag      he11oliu, 23小时前 •
    ldr x1, [x0]
    cmp     x1, #0
    bne wait_for_bss_clear

    /* Turn to el1 from other exception levels. */
    bl  arm64_elX_to_el1

    /* Prepare stack pointer and jump to C. */
    mov x1, #INIT_STACK_SIZE
    mul x1, x8, x1
    ldr     x0, =boot_cpu_stack
    add x0, x0, x1
    add x0, x0, #INIT_STACK_SIZE
    mov sp, x0
```

其他CPU会通过检查`clear_bss_flag`的值是否为0来判断`bss`段有没有被清空：若不为0则说明未清空，跳转回`wait_for_bss_clear`处继续循环检查；若已经清空，则将异常级别转为`el1`并着手准备栈指针。

```
wait_until_smp_enabled:
    /* CPU ID should be stored in x8 from the first line */
    mov x1, #8
    mul x2, x8, x1
    ldr x1, =secondary_boot_flag
    add x1, x1, x2
    ldr x3, [x1]
    cbz x3, wait_until_smp_enabled

    /* Set CPU id */
    mov x0, x8
    bl  secondary_init_c
```

之后，其他CPU会继续循环等待，并通过检查`secondary_boot_flag`的值来判断是否跳出循环终止等待。在终止等待之后设置CPU Id并运行`secondary_init_c`从而激活该CPU。

这就是阻塞的全部过程。

思考题2：阅读汇编代码 kernel/arch/aarch64/boot/raspi3/init/start.S, init_c.c 以及 kernel/arch/aarch64/main.c ，解释用于阻塞其他CPU核心的 secondary_boot_flag 是物理地址还是虚拟地址？是如何传入函数 enable_smp_cores 中，又该如何赋值的（考虑虚拟地址/物理地址）？

答：`secondary_boot_flag`是物理地址，因为此时`secondary CPU`的`MMU`还没有初始化，仍然在使用物理地址。

`secondary_boot_flag`数组在`init_c`函数中初次出现，被传递给`start_kernel`函数，然后在其中继续被放入寄存器`X0`中传递给`main`函数，并在`main`函数内调用`enable_smp_cores`时被传递过去。

在`init_c.c`中，`secondary_boot_flag`被初始化为`{NOT_BSS, 0, 0, ...}`。在`enable_smp_cores`中，先通过`phys_to_virt`将`secondary_boot_flag`的物理地址转换为虚拟地址，然后`secondary_boot_flag[i] = 0xBEEFUL`进行相关的赋值操作，最后调用`flush_dcache_area`将修改的内容flush掉。

思考题5：在 el0_syscall 调用 lock_kernel 时，在栈上保存了寄存器的值。这是为了避免调用 lock_kernel 时修改这些寄存器。在 unlock_kernel 时，是否需要将寄存器的值保存到栈中，试分析其原因。

答：因为在`el0_syscall`执行`unlock_kernel`后，将继续执行`exception_exit`，会从内核栈中恢复用户态的寄存器并回归到用户态。此后，就不再需要使用`caller-saved register`了，因此不需要再存入栈中保护。

思考题6：为何 idle_threads 不会加入到等待队列中？请分析其原因?

答：因为idle_threads 的作用是在CPU核心没有要调度的线程时运行，防止内核忙等而导致大内核锁锁住整个内核。其本质并非是想要交由内核调度运行的可执行线程，不应被分配时间片，也就不应该加入到等待队列中接受FIFO原则的调度。

思考题8：如果异常是从内核态捕获的，CPU核心不会在 kernel/arch/aarch64/irq/irq_entry.c 的 handle_irq 中获得大内核锁。但是，有一种特殊情况，即如果空闲线程（以内核态运行）中捕获了错误，则CPU核心还应该获取大内核锁。否则，内核可能会被永远阻塞。请思考一下原因。

答：若运行在内核态的空闲线程不持有大内核锁，在`handle_irq`中，其仍然会因调用`eret_to_thread(switch_context())`而导致放锁，使得有`lock->owner++`。

当正常状态下，`lock->owner`和`lock->next`均为`n`时，若有两个内核态的空闲线程中均捕获了错误，则会导致`lock->owner`变为`n+2`。那么下一个线程想要拿锁时，`lock->next`将会变为`n+1`，而此时有`lock->owner > lock->next`，所以该线程将无法放锁，导致内核永远被阻塞。