OS Lab4 实验报告

王梓萌 521030910015

1 思考题一

启动多核

在实验1中我们已经介绍，在QEMU模拟的树莓派中，所有CPU核心在开机时会被同时启动。在引导时这些核心会被分为两种类型。一个指定的CPU核心会引导整个操作系统和初始化自身，被称为主CPU（primary CPU）。其他的CPU核心只初始化自身即可，被称为其他CPU（backup CPU）。CPU核心仅在系统引导时有所区分，在其他阶段，每个CPU核心都是被相同对待的。

> 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，并阻塞其他其他CPU的执行的。

在树莓派真机中，还需要主CPU手动指定每一个CPU核心的的启动地址。这些CPU核心会读取固定地址的上填写的启动地址，并跳转到该地址启动。在kernel/arch/aarch64/boot/raspi3/init/init_c.c中，我们提供了wakeup_other_cores函数用于实现该功能，并让所有的CPU核心同在QEMU一样开始执行_start函数。

与之前的实验一样，主CPU在第一次返回用户态之前会在kernel/arch/aarch64/main.c中执行main函数，进行操作系统的初始化任务。在本小节中，ChCore将执行enable_smp_cores函数激活各个其他CPU。

> 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，又是如何赋值的（考虑虚拟地址/物理地址）？

这段代码通过检查CPU ID来确定主CPU，并利用同步机制（如等待BSS清除和等待SMP启用的信号）来控制其他CPU的执行流程，直到它们被允许继续执行。

确定主CPU的方法为：

- `mrs x8, mpidr_el1`：这条指令从`mpidr_el1`系统寄存器中读取当前CPU的唯一ID，并存储到寄存器`x8`中。
- `and x8, x8, #0xFF`：这条指令通过与操作`AND`将`x8`的值限制在255（即`0xFF`）以内，这通常用于提取多核CPU中的核心编号。
- `cbz x8, primary`：如果`x8`（即CPU ID）为0，跳转到标签`primary`。这意味着只有CPU ID为0的处理器（主CPU）会执行标签`primary`下的代码。在多核处理器系统中，通常CPU ID为0的核心作为主CPU来引导操作系统。

阻塞其他CPU的方法为：

- 通过判断代码，使非主CPU等待BSS段（未初始化的全局静态数据）被清除（`wait_for_bss_clear`循环。
- 通过`wait_until_smp_enabled`循环等待一个信号（`secondary_boot_flag`），这个信号表明它可以继续初始化。如果信号为非，将保持阻塞状态。

这样以后，主CPU会通过调用`bl arm64_elX_to_el1`切换到较低的异常级别（EL1），设置堆栈指针，并跳转到C语言环境中的初始化函数`init_c`。

直到主CPU完成对多核的支持之后，`clear_bss_flag`地址处的值 和 `secondary_boot_flag`地址指向的数组中对应flag变量相继被设为1，则非主CPU会停止阻塞，开始进行类似的异常级别切换和堆栈设置并最终会跳转到另一个C语言环境中的函数`secondary_init_c`以完成它们的初始化。

2 思考题二

`secondary_boot_flag` **是一个存储物理地址的指针**，所指向的物理地址上存储的是一个长整型类型的数组，数组的长度使CPU的数量，每个元素存储的是一个值为1或者0的flag变量。每个非主CPU在 `start.S` 文件中的 `wait_until_smp_enabled` 循环中正是不断读取 `secondary_boot_flag[#]` 位置存储的flag，确定是否停止等待，开始进入初始化流程（`#` 为CPUID）。由于非主CPU运行 `start.S` 代码是，mmu 尚未开启，所以显然 `secondary_boot_flag` 是一个物理地址。

`secondary_boot_flag` 被传入 `enable_smp_cores` 函数的过程是：

首先在 `init_c.c` 文件中，`secondary_boot_flag` 被定义和初始化（此时主CPU的mmu也没有被打开，因此也能证实它是物理地址），然后在调用 `start_kernel()` 的时候，`secondary_boot_flag` 被作为参数传入，随后猜测 start_kernel 会调用 `main.c` 中的 `main()` 函数，`secondary_boot_flag` 同样被作为参数传入。

然后在main函数中，程序调用了 `enable_smp_cores()` 函数，`secondary_boot_flag` 同样被作为参数传入。以上为参数传入的过程。

赋值的过程是：

根据以下代码可以看出

```
long *secondary_boot_flag;

/* Set current cpu status */
cpu_status[smp_get_cpu_id()] = cpu_run;
secondary_boot_flag = (long *)phys_to_virt(boot_flag);
for (i = 0; i < PLAT_CPU_NUM; i++) {
    secondary_boot_flag[i] = 1;
    flush_dcache_area((u64) secondary_boot_flag,
              (u64) sizeof(u64) * PLAT_CPU_NUM);
    asm volatile ("dsb sy");
    while (cpu_status[i] == cpu_hang)
    ;
    kinfo("CPU %d is active\n", i);
}
```

传入的物理地址首先通过 `phys_to_virt()` 函数转化为mmu开启后可以寻址的虚拟地址，然后再使用循环，对 `secondary_boot_flag` 指向位置的flag 数组进行赋值，每个元素赋值为1，即对应的非主CPU读取flag值之后可以退出 `wait_until_smp_enabled` 循环，开始初始化。

3 练习1

`rr_sched_init` 初始化了round_robin算法的等待队列，具体来说我们需要完成等待队列结构体中的队列初始化，元素数量成员变量初始化，以及锁的初始化，这些都能够调用lab为我们准备好的初始化函数完成。

具体实现在源代码中

4 练习2

`__rr_sched_enqueue` 函数具体实现了将一个线程插入到等待队列的任务。具体地我们需要使用列表相关操作的辅助函数完成添加队列元素到队尾，同时更新队列元素数量。

具体实现在源代码中。

5 练习3

`find_runnable_thread`函数实现了找到等待队列中具备被调度资格的第一个任务（线程）的任务。具体地，据被调度资格的定义包括线程处于 `ready` 状态，线程具有余额时间片等等，具体的检查方式在源代码的注释中已经给出。

接着，我们要将find函数返回的任务从等待队列中出队，实现`__rr_sched_dequeue`以完成这一任务。该函数与 `__rr_sched_enqueue` 函数相似。

具体实现在源代码中。

6 练习4

`sys_yield`是一个syscall函数，使用户态程序可以主动让出CPU核心触发线程调度。作为一个接口函数，它的实现是通过调用我们先前实现的 `rr_sched` 总函数完成的，因此在函数体中秩序调用`rr_sched()`函数即可

7 练习5

物理时钟的初始化需要完成:

- 读取 CNTFRQ_EL0 寄存器，为全局变量 cntp_freq 赋值。
- 根据 TICK_MS（由ChCore决定的时钟中断的时间间隔，以ms为单位，ChCore默认每10ms触发一次时钟中断）和cntfrq_el0 （即物理时钟的频率）计算每两次时钟中断之间 system count 的增长量，将其赋值给 cntp_tval 全局变量，并将 cntp_tval 写入 CNTP_TVAL_EL0 寄存器!
- 根据上述说明配置控制寄存器CNTP_CTL_EL0的值为1

具体实现在源代码中

8 练习6

`handle_timer_irq` 函数在每次时钟中断时刻被选择并调用。它需要递减当前正在运行的current_thread 的时间片额度，并在其归0的时候调用调度dequeue函数进行处理，选择新的任务开始运行。

具体实现在源代码中

9 练习7

具体实现在源代码中