

1 思考题

选择主 cpu 的逻辑和 lab1 时候是一致的。通过 `mpidr_el1` 中的 `cpuid` 信息，如果当前是 0 号核，则直接跳转到 `primary` 执行并进行初始化，否则就顺序执行。在非 0 号核顺序执行的过程中，首先会被阻塞在 `wait_for_bss_clear` 区段，直到 `clear_bss` 完成后，才会继续执行。然后又会被阻塞在 `wait_until_smp_enabled` 区段，直到相应的 `secondary_boot_flag` 被设置好，才会继续执行。最后这些其他的 cpu 核心都会通过 `secondary_init_c` 完成初始化。

2 思考题

`main.c` 函数在调用 `enable_smp_cores` 时传入了参数 `boot_flag`，这是一个物理地址。在 `smp.c` 的 `enable_smp_cores` 函数中，会使用 `phys_to_virt` 将其转化为 `secondary_boot_flag`，得到一个虚拟地址。

3 练习题

由于我们是要按顺序激活各个其他 cpu，在 `enable_smp_cores` 的循环逻辑中，对于每个 `cpuid`，我们先设置其对应的 `secondary_boot_flag` 为 `cpu_run`，之后对于所有 `cpu_status` 为 `cpu_hang` 的核心，通过循环将其挂起即可。

在 `main.c` 的 `secondary_start` 中，则需要把相应的 `cpu_status` 设置为 `cpu_run`。

4 练习题

由于我们实现的是排号锁，主要是通过 `lock` 结构体中的 `owner` 和 `next` 属性来维护放锁和拿锁的逻辑。

`unlock` 函数实现的是放锁逻辑，放锁时候只需要将对应的 `lock` 的 `owner` 自增即可。`is_locked` 判断则需要比较 `lock` 的 `owner` 和 `next` 大小。余下的三个函数则是 `kernel` 的 `lock`、`unlock` 和 `lock_init` 的实现，使用代码中给我们提供的 `big_kernel_lock`，在三个函数中分别调用相应的 `lock`、`unlock` 和 `lock_init` 即可。

我们需要在 `main.c` 的 `main` 函数创建根线程之前，`secondary_start` 函数其他进程开始调度之前加锁。在 `irq_entry.c` 的中断处理和异常处理时，最开始就要判断是否拿锁。在 `irq_entry.S` 的 `sync_el0_64` 的异常返回前，要放锁；在 `el0_syscall` 使用 `ldp` 汇编指令保存寄存器值之前，需要拿锁，异常返回之前再放锁；在 `__eret_to_thread` 异常返回前，要放锁。

5 思考题

不需要。在我们 `unlock_kernel` 的时候，会将 `big_kernel_lock` 的 `owner` 字段进行修改，这个修改是不涉及寄存器的，所以不需要保存到栈上。

6 思考题

实验文档告诉我们，当 CPU 核心没有要调度的线程时，它不应在内核态忙等，否则它持有的大内核锁将锁住整个内核。idle_threads 的引入就是为了让各个 cpu 在没有线程时候去执行他，防止内核被锁住。由于我们现在使用的是 RoundRobin 的调度策略，所有在等待队列中的进程具有一致的优先级，如果 idle_threads 进入等待队列的话，会造成性能损耗。

7 练习题

enqueue 时先对 thread 的属性进行一些判断，如果满足条件，则设置该 thread 的 state 和 cpuid，然后通过 list_append 宏将其加入到等待队列的尾部，并修改对应 cpuid 的就绪队列的长度参数。deque 时候的操作和 enqueue 是对偶关系，按照 enqueue 时候进行的修改倒序进行即可。choose_thread 的时候，我们尝试将等待队列的队首线程出队，如果队列为空，那么就返回对应 cpu 的空闲线程即可。shed 时候，我们首先对当前线程的参数进行一些必要的检查，然后将其挂起并选取下一个线程，最后进行线程切换。

8 思考题

当线程捕获错误的时候，如果当前线程是一个用户态线程，cpu 核心会获得大内核锁，并转换至内核态来解决相应的错误。但是空闲线程本身并不是对任何用户态线程的抽象，他只是我们为了防止 cpu 没有线程可以调度而人为创造出来的线程，它本身就是执行在内核态的（而不是用户态）。如果在空闲线程捕获了错误的时候 cpu 没有拿大内核锁，可能 cpu 没有办法去解决空闲线程上出现的错误，这样就导致这个 cpu 的空闲线程被 kill 掉，这样当这个 cpu 再次调度的时候，就没有相应的能调度的空闲线程了，这可能会导致永远阻塞。

9 练习题

sys_yield 中直接将线程的 budget 设置为 0，然后进行切换即可；sys_get_cpu_id 系统调用里面直接内嵌 smp_get_cpu_id 即可。

10 练习题

主 CPU 以及其他 CPU 的初始化流程分别在 main 和 secondary_start 中，我们在预留的位置填入 timer_init 即可。

11 练习题

先对当前线程进行检查，符合条件的情况下，直接对 budget 进行减一即可。

12 练习题

`thread.c` 中的 `sys_set_affinity` 和 `sys_get_affinity` 函数中已经帮我们实现了绝大部分代码，我们只需要将上下文变量中的 `aff` 和线程内置属性中的 `affinity` 互相赋值即可。在 `policy_rr.c` 中，我们扩展之前实现的代码，之前是通过 `smp_get_cpu_id` 来获取当前线程所在的 `cpu`，现在我们可以支持通过当前线程的 `affinity` 属性选取其亲和的 `cpu`。

13 练习题

根据上下文代码实现即可。

14 练习题

需要实现比较多的函数调用，根据上下文注释，设置对应的参数并进行函数调用即可。

15 练习题

在 `wait_sem` 中，需要首先检查对应信号量的 `sem_count` 是否为零，如果不为零，直接减减即可。如果为零，则需要判断是否将当前线程加入等待队列，如果需要加入等待队列，则需要将当前线程加入等待队列，然后进行调度。

在 `signal_sem` 中，需要首先检查是否有线程正在等待资源，如果有，则直接将信号量的 `sem_count` 加一即可。否则，就通过 `list_entry` 宏来唤醒一个等待线程，并使用上面实现过的 `shed_enqueue` 函数将其入队。

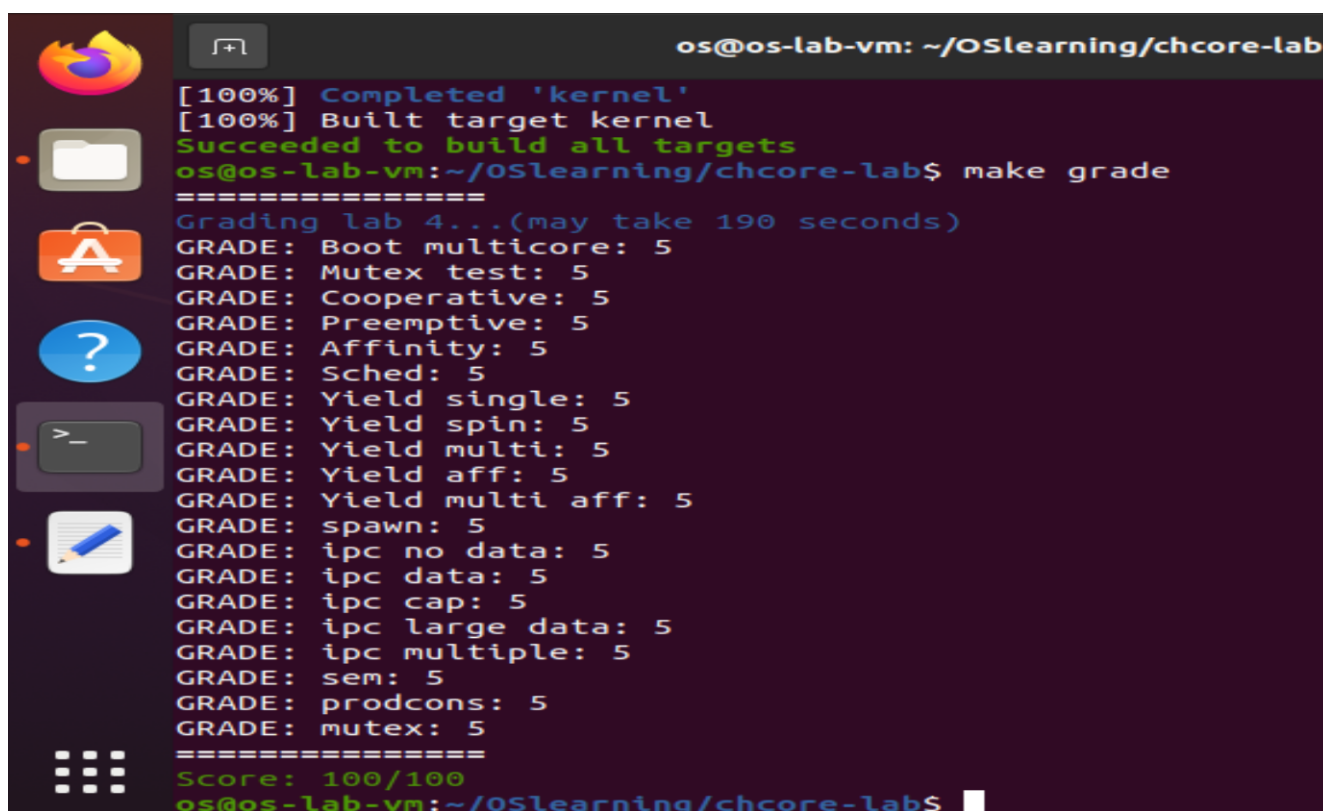
16 练习题

这里主要使用了 `__chcore_sys_wait_sem` 和 `__chcore_sys_signal_sem` 两个系统调用，正确地将上文代码提供的 `emptyt_slot` 和 `filled_slot` 传入即可。

17 练习题

在 `lock_init`、`lock` 和 `unlock` 中，我们分别需要使用 `__chcore_sys_create_sem`、`__chcore_sys_wait_sem` 和 `__chcore_sys_signal_sem` 三个系统调用，将 `lock` 中的 `lock_sem` 传入系统调用即可。

18 运行结果



```
os@os-lab-vm: ~/OSlearning/chcore-lab
[100%] Completed 'kernel'
[100%] Built target kernel
Succeeded to build all targets
os@os-lab-vm:~/OSlearning/chcore-lab$ make grade
=====
Grading lab 4...(may take 190 seconds)
GRADE: Boot multicore: 5
GRADE: Mutex test: 5
GRADE: Cooperative: 5
GRADE: Preemptive: 5
GRADE: Affinity: 5
GRADE: Sched: 5
GRADE: Yield single: 5
GRADE: Yield spin: 5
GRADE: Yield multi: 5
GRADE: Yield aff: 5
GRADE: Yield multi aff: 5
GRADE: spawn: 5
GRADE: ipc no data: 5
GRADE: ipc data: 5
GRADE: ipc cap: 5
GRADE: ipc large data: 5
GRADE: ipc multiple: 5
GRADE: sem: 5
GRADE: prodcons: 5
GRADE: mutex: 5
=====
Score: 100/100
os@os-lab-vm:~/OSlearning/chcore-lab$
```

图 1: make grade 结果