## 练习 1

使用 git diff lab3 命令可以发现实验三和本实验之间_start 部分代码的差异如下：

```
-    /* hang all secondary processors before we intorduce multi-processors */
-secondary_hang:
-            bl secondary_hang
+            /* Wait for bss clear */
+wait_for_bss_clear:
+            adr         x0, clear_bss_flag
+            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, #0x1000
+            mul          x1, x8, x1
+            adr          x0, boot_cpu_stack
+            add          x0, x0, x1
+            add          x0, x0, #0x1000
+             mov         sp, x0
+
+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
```

与之前粗暴地把所有副处理器挂起不同，在本实验中，主 CPU 与原来的行为一致，而副 CPU 首先等待 BSS 段清零，然后开始启动，初始化 CPU 的栈，然后被阻塞，接下来副 CPU 还要等待主 CPU 显式地激活它（通过设置 secondary_boot_flag 的值为 1），当副 CPU 被激活时，它就执行初始化的过程。

## 练习 2

在 enable_smp_cores()函数中把每个 CPU 核心的 secondary_boot_flag 的值设置为 1，在 secondary_start()函数中 cpu_status 的值设置为 1 即可。

## 练习 3

会导致并发问题，虽然不同 CPU 核心使用不同的内核堆栈，但不同核心可以同时运行代码，导致数据竞争的问题的出现。

## 练习 4

排号锁类似于医院里的叫号机。当多个病人在排一个医生时，叫号机从小到大给每个排队的

病人分配一个号，只有得到的号等于显示屏上显示的号的病人才能进去看病。所以 unlock 函数的做法是把显示屏上的号（owner）加一，is_locked 函数的做法是判断病人拿到的号（next）是否等于显示屏上的号。

## 练习 5

首先要按照文档的描述在 6 个位置调用 lock_kernel()或 unlock_kernel()接口，然后实现 kernel_lock_init()、lock_kernel()和 unlock_kernel()。这三个函数的实现只要通过调用三个已经实现好的函数即可完成。

## 练习 6

在 el0_syscall 调用 lock_kernel()完成后，还要执行之后的代码，所以需要在栈上保存寄存器的值；在 exception_return 中调用 unlock_kernel()完成后，就从内核态返回到用户态，此前寄存器的状态将不再被需要，不需要保存在栈上。

## 练习 7

该练习需要完善五个函数：
rr_sched_enqueue：把一个线程添加到 rr_ready_queue 中，同时要把该线程的状态设为 TS_READY，把该线程对应的 cpu 核心设为当前 CPU 核心。这里还要注意边界条件的检查、判断被添加的线程是否已经在队列中以及被添加的线程是否为空闲线程。
rr_sched_dequeue：把一个线程移出队列，并把线程状态设为 TS_INTER。条件检查同样是必要的。
rr_sched_choose_thread：选择要调度的线程，这里选择的是队列的头节点，注意这里还要调用 rr_sched_dequeue 使之出队。
rr_sched_refill_budget：设置线程 budget 的值。
rr_sched：调度函数。如果当前有正在运行的线程，则调用函数 rr_sched_enqueue 把它添加到队列里。然后调用 rr_sched_choose_thread 函数选择要调度的线程，将该线程的 budget 设为 DEFAULT_BUDGET，最后调用 switch_to_thread 函数即可。

## 练习 8

空闲线程不能运行，因为它无法释放大内核锁，导致内核被永远阻塞。由于 CPU 核心已经拿到了大内核锁，再一次获取大内核锁将无法成功，这样就成功防止了空闲线程的运行。所以 handle_irq()函数要做一点变动。

## 练习 9

sys_get_cpu_id()只要调用 smp_get_cpu_id()即可。
sys_yield()需要调用 sched()函数实现调度。

## 练习 10

取消注释……

## 练习 11

rr_sched()函数需要判断当前线程的 budget 是否为 0，只有 budget 为 0 才能进行调度，rr_sched_handle_timer_irq()只需要将当前线程的预算减一即可。此外，还需要在 sys_yield()加上下面的代码来重置预算：

```
current_thread->thread_ctx->sc->budget = 0;
```

## 练习 12

增加了线程的亲和性支持，首先判断线程的 affinity 是否为 NO_AFF，如果是，则把线程加入当前 CPU 核心的 rr_ready_queue，反之加入它指定的 CPU 核心的 rr_ready_queue。
做完这个练习后运行 make grade，发现 yield spin 并没有成功，仔细研究后发现 handle_irq() 遗漏了一些代码。文档中要求处理时钟中断时,将当前线程的预算减少一，然后根据预算是否等于零判断是否进行调度，但这里并没有实现这个要求，完善这个函数再运行 make grade，发现 yield spin 成功通过！

## 练习 13

此练习要求完善 sys_set_affinity 和 sys_get_affinity 系统调用，用于设置或获取线程的亲和性。在这里要注意一些条件的判断。

## 练习 14

第一处和第二处：文档中已经要求 PMO 的大小为 MAIN_THREAD_STACK_SIZE 且被立即分配，所以两处应分别填 MAIN_THREAD_STACK_SIZE、PMO_DATA。
第三处：栈顶的位置应该为栈的基地址加上栈的尺寸，所以此处应该填 MAIN_THREAD_STACK_BASE + MAIN_THREAD_STACK_SIZE。
第四处：注释给出提示：stack_offset 是栈的基地址与 sp 寄存器的值之差。由于这一步是在父进程的本地内存中构造一个初始页面，所以 sp 寄存器指向的位置应该距离栈顶一个页面的大小。故此处应该填 MAIN_THREAD_STACK_SIZE - PAGE_SIZE。
第五、六、七处：这一步是将 main_stack_pmo 映射到具有 VM_READ 和 VM_WRITE 权限的子进程的地址 MAIN_THREAD_STACK_BASE。所以这三处分别应该填 main_stack_cap、MAIN_THREAD_STACK_BASE 和 VM_READ | VM_WRITE。
第八处：文档中给出提示从 sp 开始其线程的栈，所以此处应该填 stack_top - PAGE_SIZE。

## 练习 15

A、B 两部分已经给出，只要实现 C 部分即可。C 部分就是把 child_process_cap 和 child_main_thread_cap 指向的值分别设为 new_process_cap 和 main_thread_cap。

## 练习 16

ipc_call.c
第一处：要求填写 sp 寄存器的值，很显然要填的是 conn->server_stack_top。
第二处：服务器要执行的过程是回调函数，所以此处应该填 conn->target->server_ipc_config->callback。
第三处：显然要填 arg。
第四处：此处要填一个指向回调函数的参数的指针，共享内存中的 server_user_addr 应当存有参数。所以此处填 conn->buf.server_user_addr。
ipc_return.c
第一处：显然要填 ret_value。
此外，sys_ipc_call()函数中还要加上一行对 ipc_send_cap()函数的调用。

## 练习 17

简化版的 sys_ipc_call()，模仿它即可。