思考题1：

> 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 _start 函数中，首先通过 mrs 指令从 mpidr_el1 寄存器中读取当前处理器的 ID，并将其存储在 x8 寄存器中。然后，使用 and 指令将 x8 寄存器的值与 0xFF 进行位与操作，结果仍存储在 x8 中。接着，cbz 指令检查 x8 寄存器的值是否为零，如果为零，则跳转到 primary 标签处的代码，这意味着当前处理器被选为主 CPU。

对于非主 CPU（即 x8 的值不为零的 CPU），它们会进入一个等待循环，直到 clear_bss_flag 变量为零。这是一个同步控制，确保所有 CPU 在 BSS 区域被清零后再继续执行。

然后，非主 CPU 进入另一个等待循环，直到 secondary_boot_flag 变量为非零。这是另一个同步控制，确保所有 CPU 在 SMP（对称多处理）启用后再继续执行。

这样，ChCore 内核就通过 mpidr_el1 寄存器的值来选定主 CPU，并通过 clear_bss_flag 和 secondary_boot_flag 变量来阻塞其他 CPU 的执行，直到满足特定的条件。

思考题2

> 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）？

是虚拟地址，因为使用secondary_boot_flag时MMU已经启动

secondary_boot_flag通过main函数的参数boot_flag传入enable_smp_cores

在kernel/arch/aarch64/main.c文件中的main函数中可知，boot_flag是smp的boot flag地址，是物理地址，main函数调用了smp.c中的enable_smp_cores函数，在这个函数中调用了对boot_flag 调用phys_to_virt将其转化为虚拟地址以获得secondary_boot_flag

练习题1

> 练习 1：在 kernel/sched/policy_rr.c 中完善 `rr_sched_init` 函数，对 `rr_ready_queue_meta` 进行初始化。在完成填写之后，你可以看到输出“Scheduler metadata is successfully initialized!”并通过 Scheduler metadata initialization 测试点。
>
> 提示：sched_init 只会在主 CPU 初始化时调用，因此 rr_sched_init 需要对每个 CPU 核心的就绪队列都进行初始化。

补全代码如下：

```
int rr_sched_init(void)
{
        /* LAB 4 TODO BEGIN (exercise 1) */
        /* Initial the ready queues (rr_ready_queue_meta) for each CPU core */
        for (unsigned int cpuid = 0; cpuid < PLAT_CPU_NUM; cpuid++) {
                rr_ready_queue_meta[cpuid].queue_len = 0;
                init_list_head(&(rr_ready_queue_meta[cpuid].queue_head));
                lock_init(&(rr_ready_queue_meta[cpuid].queue_lock));
        }
        /* LAB 4 TODO END (exercise 1) */
```

```
        test_scheduler_meta();
        return 0;
}
```

对`rr_ready_queue_meta`中每个CPU核心进行初始化，同时进行上锁

练习题2

> 练习 2：在 kernel/sched/policy_rr.c 中完善 `__rr_sched_enqueue` 函数，将`thread`插入到`cpuid`对应的就绪队列中。在完成填写之后，你可以看到输出“Successfully enqueue root thread”并通过 Schedule Enqueue 测试点。

补全代码如下：

```
list_append(&thread->ready_queue_node, &rr_ready_queue_meta[cpuid].queue_head);
rr_ready_queue_meta[cpuid].queue_len += 1;
```

将`thread`插入到`cpuid`对应的就绪队列中。

练习题3

> 练习 3：在 kernel/sched/sched.c 中完善 `find_runnable_thread` 函数，在就绪队列中找到第一个满足运行条件的线程并返回。 在 kernel/sched/policy_rr.c 中完善 `__rr_sched_dequeue` 函数，将被选中的线程从就绪队列中移除。在完成填写之后，运行 ChCore 将可以成功进入用户态，你可以看到输出“Enter Procmgr Root thread (userspace)”并通过 Schedule Enqueue 测试点。

补全代码如下：

`find_runnable_thread()`：

```
        /* LAB 4 TODO BEGIN (exercise 3) */
        /* Tip 1: use for_each_in_list to iterate the thread list */
        /*
         * Tip 2: Find the first thread in the ready queue that
         * satisfies (!thread->thread_ctx->is_suspended &&
         * (thread->thread_ctx->kernel_stack_state == KS_FREE
         * || thread == current_thread))
         */
        for_each_in_list (
                thread, struct thread, ready_queue_node, thread_list) {
                if (!thread->thread_ctx->is_suspended
                    && (thread->thread_ctx->kernel_stack_state == KS_FREE
                        || thread == current_thread)) {
                        return thread;
                }
        }
        /* LAB 4 TODO END (exercise 3) *
```

依据注释找到满足条件的`thread`。

`__rr_sched_dequeue()`：

```
        /* LAB 4 TODO BEGIN (exercise 3) */
        /* Delete thread from the ready queue and upate the queue length */
        /* Note: you should add two lines of code. */
        list_del(&thread->ready_queue_node);
        rr_ready_queue_meta[thread->thread_ctx->cpuid].queue_len--;
        /* LAB 4 TODO END (exercise 3) */
```

`__rr_sched_enqueue` 函数的逆过程。

练习题4

> 练习 4：在kernel/sched/sched.c中完善系统调用`sys_yield`，使用户态程序可以主动让出CPU核心触发线程调度。 此外，请在kernel/sched/policy_rr.c 中完善`rr_sched`函数，将当前运行的线程重新加入调度队列中。在完成填写之后，运行 ChCore 将可以成功进入用户态并创建两个线程交替执行，你可以看到输出“Cooperative Schedluing Test Done!”并通过 Cooperative Schedluing 测试点。

- `sys_yield`，调用`sched()`；
- 通过`rr_sched_enqueue(old)`将当前正在运行的线程重新加入调度队列中。

练习题5

> 练习 5：请根据代码中的注释在kernel/arch/aarch64/plat/raspi3/irq/timer.c中完善`plat_timer_init`函数，初始化物理时钟。需要完成的步骤有：
>
> - 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。
>
> 由于启用了时钟中断，但目前还没有对中断进行处理，所以会影响评分脚本的评分，你可以通过运行ChCore观察是否有"[TEST] Physical Timer was successfully initialized!: OK"输出来判断是否正确对物理时钟进行初始化。

根据注释，补全代码如下：

```
void plat_timer_init(void)
{
        u64 timer_ctl = 0;
        u32 cpuid = smp_get_cpu_id();

        /* Since QEMU only emulate the generic timer, we use the generic timer
         * here */
        asm volatile("mrs %0, cntpct_el0" : "=r"(cntp_init));
        kdebug("timer init cntpct_el0 = %lu\n", cntp_init);

        /* LAB 4 TODO BEGIN (exercise 5) */
        /* Note: you should add three lines of code. */
        /* Read system register cntfrq_el0 to cntp_freq*/
        asm volatile("mrs %0, cntfrq_el0" : "=r"(cntp_freq));
        /* Calculate the cntp_tval based on TICK_MS and cntp_freq */
        cntp_tval = (cntp_freq / 1000 * TICK_MS);
```

```
        /* Write cntp_tval to the system register cntp_tval_el0 */
        asm volatile("msr cntp_tval_el0, %0" ::"r"(cntp_tval));
        /* LAB 4 TODO END (exercise 5) */

        tick_per_us = cntp_freq / 1000 / 1000;
        /* Enable CNTPNSIRQ and CNTVIRQ */
        put32(core_timer_irqcntl[cpuid], INT_SRC_TIMER1 | INT_SRC_TIMER3);

        /* LAB 4 TODO BEGIN (exercise 5) */
        /* Note: you should add two lines of code. */
        /* Calculate the value of timer_ctl */
        timer_ctl = 0 << 1 | 1;
        /* Write timer_ctl to the control register (cntp_ctl_el0) */
        asm volatile("msr cntp_ctl_el0, %0" ::"r"(timer_ctl));
        /* LAB 4 TODO END (exercise 5) */

        test_timer_init();
        return;
}
```

练习题6

> 练习 6：请在kernel/arch/aarch64/plat/raspi3/irq/irq.c中完善`plat_handle_irq`函数，当中断号irq为INT_SRC_TIMER1（代表中断源为物理时钟）时调用`handle_timer_irq`并返回。 请在kernel/irq/irq.c中完善`handle_timer_irq`函数，递减当前运行线程的时间片budget，并调用sched函数触发调度。 请在kernel/sched/policy_rr.c中完善`rr_sched`函数，在将当前运行线程重新加入就绪队列之前，恢复其调度时间片budget为DEFAULT_BUDGET。 在完成填写之后，运行ChCore 将可以成功进入用户态并打断创建的“自旋线程”让内核和主线程可以拿回CPU核心的控制权，你可以看到输出"Hello, I am thread 3. I'm spinning."和“Thread 1 successfully regains the control!”并通过 Preemptive Scheduling 测试点。

当中断号irq为INT_SRC_TIMER1（代表中断源为物理时钟）时调用`handle_timer_irq`并返回。

```
        switch (irq) {
                /* LAB 4 TODO BEGIN (exercise 6) */
                /* Call handle_timer_irq and return if irq equals INT_SRC_TIMER1
                 * (physical timer) */
        case INT_SRC_TIMER1:
                handle_timer_irq();
                break;
        /* LAB 4 TODO END (exercise 6) */
        default:
                // kinfo("Unsupported IRQ %d\n", irq);
                break;
        }
```

在将当前运行线程重新加入就绪队列之前，恢复其调度时间片budget为DEFAULT_BUDGET

```
/* LAB 4 TODO BEGIN (exercise 6) */
/* Refill budget for current running thread (old) */
old->thread_ctx->sc->budget = DEFAULT_BUDGET;
/* LAB 4 TODO END (exercise 6) */
```

练习题7

> 练习 7：在user/chcore-libc/musl-libc/src/chcore-port/ipc.c与kernel/ipc/connection.c中实现了大多数IPC相关的代码，请根据注释补全kernel/ipc/connection.c中的代码。之后运行ChCore可以看到 “[TEST] Test IPC finished!” 输出，你可以通过 Test IPC 测试点。

依据相关注释补全

```
        /* LAB 4 TODO BEGIN (exercise 7) */
        /* Complete the config structure, replace xxx with actual values */
        /* Record the ipc_routine_entry  */
        config->declared_ipc_routine_entry = ipc_routine;

        /* Record the registration cb thread */
        config->register_cb_thread = register_cb_thread;
        /* LAB 4 TODO END (exercise 7) */
```

```
        /* LAB 4 TODO BEGIN (exercise 7) */
        /* Complete the following fields of shm, replace xxx with actual values
         */
        conn->shm.client_shm_uaddr = shm_addr_client;
        conn->shm.shm_size = shm_size;
        conn->shm.shm_cap_in_client = shm_cap_client;
        conn->shm.shm_cap_in_server = shm_cap_server;
        /* LAB 4 TODO END (exercise 7) */
```

```
        /* Set the target thread SP/IP/arguments */
        /* LAB 4 TODO BEGIN (exercise 7) */
        /*
         * Complete the arguments in the following function calls,
         * replace xxx with actual arguments.
         */

        /* Note: see how stack address and ip are get in
         * sys_ipc_register_cb_return */
        arch_set_thread_stack(target, handler_config->ipc_routine_stack);
        arch_set_thread_next_ip(target, handler_config->ipc_routine_entry);

        /* see server_handler type in uapi/ipc.h */
        arch_set_thread_arg0(target, shm_addr);
        arch_set_thread_arg1(target, shm_size);
        arch_set_thread_arg2(target, cap_num);
        arch_set_thread_arg3(target, conn->client_badge);
        /* LAB 4 TODO END (exercise 7) */
```

```
        /* LAB 4 TODO BEGIN (exercise 7) */
        /* Set target thread SP/IP/arg, replace xxx with actual arguments */
        /* Note: see how stack address and ip are get in sys_register_server */
        arch_set_thread_stack(register_cb_thread,
                              register_cb_config->register_cb_stack);
        arch_set_thread_next_ip(register_cb_thread,
                                register_cb_config->register_cb_entry);
```

```
        /*
         * Note: see the parameter of register_cb function defined
         * in user/chcore-libc/musl-libc/src/chcore-port/ipc.c
         */
        arch_set_thread_arg0(register_cb_thread,
                             server_config->declared_ipc_routine_entry);
        /* LAB 4 TODO END (exercise 7) */
```

```
        /* LAB 4 TODO BEGIN (exercise 7) */
        /* Complete the server_shm_uaddr field of shm, replace xxx with the
         * actual value */
        conn->shm.server_shm_uaddr = server_shm_addr;
        /* LAB 4 TODO END (exercise 7) */
```