实验 4：多核、多进程、调度与IPC

薛翔元 (521030910387)

> 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。

汇编函数 `_start` 的前三行代码如下所示。

```
mrs x8, mpidr_el1
and x8, x8, #0xFF
cbz x8, primary
```

该段代码读取 `mpidr_el1` 寄存器的低 8 位并存入 `x8` 寄存器中，查阅文档可知该字段为 `Aff0`，表示多核处理器中的核心编号，因此仅有 0 号核心作为 primary CPU，进入 `primary` 函数进行初始化，其余核心作为 backup CPU，依次进入 `wait_for_bss_clear` 和 `wait_until_smp_enabled` 被暂时挂起，等待基本初始化完成后继续执行。

> 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）。

由于进入 `_start` 函数时 MMU 尚未初始化，因此 `secondary_boot_flag` 是物理地址，当 `init_c` 函数调用 `start_kernel` 函数时，会将 `secondary_boot_flag` 作为参数传入，进而由 `main` 函数将其传递给 `enable_smp_cores` 函数，此时 `secondary_boot_flag` 仍然是物理地址，因此 `enable_smp_cores` 函数使用 `phys_to_virt` 将其转换为虚拟地址，由于此时地址翻译已经开启，C 语言可以对其正常赋值。

> 练习题 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。

利用循环遍历每个 CPU 核心，调用 `init_list_head` 初始化队列，同时初始化队列长度信息即可。

```
/* LAB 4 TODO BEGIN (exercise 1) */
/* Initial the ready queues (rr_ready_queue_meta) for each CPU core */
unsigned int cpuid;

for (cpuid = 0; cpuid < PLAT_CPU_NUM; cpuid++) {
    init_list_head(&(rr_ready_queue_meta[cpuid].queue_head));
    rr_ready_queue_meta[cpuid].queue_len = 0;
}
/* LAB 4 TODO END (exercise 1) */
```

> 练习题 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。

调用 `list_append` 将线程加入到相应队列中，同时更新队列长度信息即可。

```
/* LAB 4 TODO BEGIN (exercise 2) */
/* Insert thread into the ready queue of cpuid and update queue length */
/* Note: you should add two lines of code. */
list_append(&(thread->ready_queue_node), &(rr_ready_queue_meta[cpuid].queue_head));
rr_ready_queue_meta[cpuid].queue_len++;
/* LAB 4 TODO END (exercise 2) */
```

> 练习题 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 Dequeue 测试点。

对于 `find_runnable_thread` 函数，调用 `for_each_in_list` 遍历队列，找到第一个满足条件的线程即可。

```
/* 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)) {
        break;
    }
}
/* LAB 4 TODO END (exercise 3) */
```

对于 `__rr_sched_dequeue` 函数，调用 `list_del` 将线程从队列中移除，同时更新队列长度信息即可。

```
/* 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) */
```

> 练习题 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。

对于 `sys_yield` 函数，调用 `sched` 函数触发调度即可。

```
/* LAB 4 TODO BEGIN (exercise 4) */
/* Trigger sched */
/* Note: you should just add a function call (one line of code) */
sched();
/* LAB 4 TODO END (exercise 4) */
```

对于 `rr_sched` 函数，调用 `rr_sched_enqueue` 将线程重新加入队列即可。

```
/* LAB 4 TODO BEGIN (exercise 4) */
/* Enqueue current running thread */
/* Note: you should just add a function call (one line of code) */
rr_sched_enqueue(old);
/* LAB 4 TODO END (exercise 4) */
```

> 练习题 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 "Physical Timer was successfully initialized!" 输出来判断是否正确对物理时钟进行初始化。

调用 `asm` 函数，利用汇编指令读写寄存器，正确填写变量 `cntp_tval` 和 `timer_ctl` 的值即可。

```
/* 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 * TICK_MS / 1000;
/* 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) */
```

```
/* LAB 4 TODO BEGIN (exercise 5) */
/* Note: you should add two lines of code. */
/* Calculate the value of timer_ctl */
timer_ctl = 0b1;
/* 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) */
```

> 练习题 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 "Preemptive Schedluing Test Done!" 并通过 Preemptive Scheduling 测试点。

对于 `plat_handle_irq` 函数，当中断号为 `INT_SRC_TIMER1` 时调用 `handle_timer_irq` 函数并返回即可。

```
/* 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();
    return;
/* LAB 4 TODO END (exercise 6) */
```

对于 `handle_timer_irq` 函数，递减当前线程的时间片并调用 `sched` 函数触发调度即可。

```
/* LAB 4 TODO BEGIN (exercise 6) */
/* Decrease the budget of current thread by 1 if current thread is not NULL */
if (current_thread != NULL) {
    current_thread->thread_ctx->sc->budget--;
}
/* Then call sched to trigger scheduling */
sched();
/* LAB 4 TODO END (exercise 6) */
```

对于 `rr_sched` 函数，将线程时间片重置为 `DEFAULT_BUDGET` 即可。

```
/* 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: 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。

根据注释依次填写变量，完成参数传递即可，其中 `arch_set_thread_stack` 和 `arch_set_thread_next_ip` 函数的参数应从对应的 `config` 结构体中获取。

```
/* 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) */
```

```
/* 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) */
```

至此，运行 `make qemu` 可以正常进入 `shell`，运行 `make grade` 可以通过所有测试。

```
 ______     __  __     ______     __  __     ______     __         __
/\  ___\   /\ \_\ \   /\  ___\   /\ \_\ \   /\  ___\   /\ \       /\ \
\ \ \____  \ \  __ \  \ \___  \  \ \  __ \  \ \ \__ \   \ \ \____  \ \ \____
 \ \_____\  \ \_\ \_\  \/\_____\  \ \_\ \_\  \ \_____\  \ \_____\  \ \_____\
  \/_____/   \/_/\/_/   \/_____/   \/_/\/_/   \/_____/   \/_____/   \/_____/

Welcome to ChCore shell!
```

```
===============
Grading lab 4...(may take 10 seconds)
GRADE: Scheduler metadata initialization: 10
GRADE: Schedule Enqueue: 10
GRADE: Schedule Dequeue: 10
GRADE: Cooperative Schedluing: 20
GRADE: Preemptive Schedluing: 20
GRADE: Test IPC: 30
===============
Score: 100/100
```