ChCore 实验 4

班级：F2103003 姓名：黄伊新 学号：521030910070

1. **思考 1：根据代码说明 ChCore 是如何选定主 CPU，并阻塞其他 CPU 的执行的**

   ChCore 在进入 _start 函数以后，首先从系统寄存器 mpidr_el1 中读取了值放入 x8，并用 and 掩膜得到了值，该值即为目前的 cpuid。若 cpuid 为 0 则跳转到 primary 去执行进一步的启动，即 cpu 0 被设定为了主 cpu。而其他的 cpu 则继续往下执行。当执行到 wait_until_smp_enabled 时就会发生阻塞。只要等到 secondary_boot_flag[cpuid] 非零时才会被允许继续执行，进入 secondary_init_c 去进行 cpu 的初始化。而具体主 cpu 是如何控制的是在 init_c.c 这个函数中，他表明 seconday_boot_flag 会在 enable_smp_cores 被置为非零。这个函数是在 smp.c 这个文件当中，他按顺序将副 cpu 的 secondary_boot_flag 置为 1，并且需要等到对应的 cpu_status 不等于 cpu_hang 后才接着往下进行其他副 cpu 的启动，保证了其顺序执行。而 cpu_status 的设置是在 secondary_init_c 的执行中，当该副 cpu 初始化完成以后就会将其置为 cpu_run。

2. **思考 2：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）**

   secondary_boot_flag 在 start.S 中是物理地址，此时还没有启动 MMU，所以只可能为物理地址。后续它是通过 main.c 中的函数传参传入了 boot_flag，然后在 enable_smp_cores 中，boot_flag 通过 phys_to_virt 转为虚拟地址存为一个局部变量 secondary_boot_flag，然后主 cpu 通过用虚拟地址访问然后对他的值进行修改，从而去更新值来达到让其他 cpu 启动的这么个目的。

3. **练习 1：完善 rr_sched_init 函数**

   本练习的主要思路是调用各类函数进行初始化。考察 rr_ready_queue_meta 数据结构，需要初始化的为 queue head，queue len 和 queue lock，queue head 和 queue lock 都有对应的函数直接进行初始化，而 queue len 是通过直接设为 0 进行初始化。

```
for (int i = 0; i < PLAT_CPU_NUM; i++){
  init_list_head(&(rr_ready_queue_meta[i].queue_head));
  rr_ready_queue_meta[i].queue_len = 0;
  lock_init(&(rr_ready_queue_meta[i].queue_lock));
// rr_ready_queue_meta[i].pad = '\0';
}

```

4. **练习 2：完善 ___rr_sched_enqueue，插入 thread**

   可以看到 queue 使用一个 list 来进行维护的，那么就可以通过调用函数 list_append 来进行插入。同时，由于进行了插入，也需要更改元数据，queue len 需要加 1。

5. **练习 3：完善 find_runnable_thread 函数和 ___rr_sched_dequeue 函数**

   根据提示，可以使用 for_each_in_list 这个函数来遍历 thread list 寻找第一个符合条件的 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;
  }
}

```

   在 dequeue 函数中那么就是要把这个 thread 移出这个 list，可以直接调用 list_del 函数来进行，同时和前面类似，也要修改 meta data。

6. **练习 4：完善系统调用 sys_yield**

   只需要简单第哦啊用 shced() 函数即可，目的就是让用户态程序可以触发线程调度。

```
sched();
```

7. **练习 5：完善 plat_timer_init 函数**

   首先，根据指示，通过 mrs 指令读取 CNTFRQ_EL0 寄存器，为全局变量 cntp_freq 赋值。接着，cntp_tval 根据定义应当为 cntp_freq * TICK_MS / 1000，根据公式为其进行赋值。最后，将该值通过 msr 函数来写入系统寄存器 cntp_tval_el0 即可。

```
asm volatile ("mrs %0, cntfrq_el0": "=r" (cntp_freq));

cntp_tval = (cntp_freq * TICK_MS) / 1000;

asm volatile ("msr cntp_tval_el0, %0" ::"r"(cntp_tval));
```

8. **练习 6:完善 plat_handle_irq 函数,handle_time_irq 函数和 rr_ssched 函数**

   在 irq.c 中需要完善的是对于 irq 的讨论，当其为 INT_SRC_TIMER1 中，调用 handle_timer_irq 并返回。

```
case INT_SRC_TIMER1:
  handle_timer_irq();
  return;
```

在 timer.c 中，我们需要完成的是 budget 的递减。主要就是需要对 current thread 的情况进行判断以后递减即可。

```
if (current_thread && current_thread->thread_ctx && current_thread
    ->thread_ctx->sc && current_thread->thread_ctx->sc->budget &&
     current_thread->thread_ctx->sc->budget> 0){
  current_thread->thread_ctx->sc->budget--;
}
```

在 policy_rr.c 中的操作也很简单，将目前的 thread 的 budge 置为 DEFAULT_BUDGET 即可。

```
old->thread_ctx->sc->budget = DEFAULT_BUDGET;

```

9. **练习 7：完善 connection.c 中关于 IPC 连接的代码**

首先是在 register_server 中，主要是对 ipc_server_config 进行设置，具体设置的值如下：

```
config->declared_ipc_routine_entry = ipc_routine;

config->register_cb_thread = register_cb_thread;

```

接着在 create_connection 函数中，主要需要完善的是 IPC 连接二者公用的 shared memory 的一些性质。client_shm_uaddr 是 client 端的这个 shared memory 的地址，这个是函数调用的时候直接传入的。shm_size 指的是这块共享的 shared memory 的大小，他是在前面通过 get_pmo_size 得到的，同名变量直接赋值即可。shm_cap_in_client 指的是在客户端的 cap_t 的值，也是通过函数直接传入的；最后 shm_cap_in_server 也已经在前面通过 cap_copy 获得了，由于 shared memory 是在 client 这边负责的，所以没有传入而是在里面获得的，具体的赋值代码如下：

```
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;

```

第三处需要完成的代码在 ipc_thread_migrate_to_server 中。这边主要是对 thread 的各种性质进行设置。stack 的设置的值采用的是之前在 sys_ipc_register_cb_return 中在 handler_config 设置的 ipc_routine_stack。同理，ip 是同样方式设置的 ipc_routine_entry。而对于参数，在 ipc.h 中指出，这四个参数分别为 shm_ptr，max_data_len，send_cap_num 和 client_badge，这几项都在函数中有对应的变量，赋值即可。具体代码如下：

```
arch_set_thread_stack(target, handler_config->ipc_routine_stack);
arch_set_thread_next_ip(target, handler_config->ipc_routine_entry)
    ;

```

```
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);

```

下一段需要填充的是 sys_register_client 这个函数，同样也是需要设置 thread 在返回以后的一个状态。这边需要设置的是 register_cb_thread 的 context，这个也是提前有设置，存在 register_cb_config 中，直接读取即可。而对于这边需要传入的 arg0，参考 chcore-port/ipc.c 中传入的是 server 这个信息，arg0 设置为 server_config 中所存的 entry 值。最终的赋值结果如下：

```
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);

arch_set_thread_arg0(register_cb_thread, server_config->
    declared_ipc_routine_entry);

```

最后一处需要填写的地方在 sys_ipc_register_cb_return 中，比较的简单，主要是设置这个 connection 的 shared memory 的 server 方面的 add 的值，这个也是在调用函数的时候有进行传入的直接赋值即可。

```
conn->shm.server_shm_uaddr = server_shm_addr;

```