Lab4:多核调度与IPC

陈景韬 523030910028

思考题1

```
BEGIN_FUNC(_start)
          mrs       x8, mpidr_el1 //mpidr_el1：Multi-Processor ID Register，存储当前 CPU 的
          and       x8, x8,  #0xFF
          cbz       x8, primary

          /* Wait for bss clear */
```

在 `start.S` 中，我们首先取出 `mpidr_el1` 中存储的当前cpu的唯一标识到通用寄存器 `x8` 中，然后将其与0比较，如果为0则跳转到primary函数部分继续进行初始化，否则进入 `wait_for_bss_clear` 部分，后续将会在 `wait_until_smp_enabled` 函数中，等待主核完成初始化，再进入 `secondary_init_c` ，开始从核的初始化流程。

思考题2

`start_kernel()` 被定义在 `head.S` 中，反汇编得到的代码：

```
head.S.dbg.obj:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000000000 <start_kernel>:
   0:    58000302          ldr      x2, 60 <secondary_cpu_boot+0x38>
   4:    91400442          add      x2, x2, #0x1, lsl #12
   8:    9100005f          mov      sp, x2
   c:    a9bf07e0          stp      x0, x1, [sp, #-16]!
  10:    90000002          adrp     x2, 0 <empty_page>
  14:    d5182002          msr      ttbr0_el1, x2
  18:    d5033fdf          isb
  1c:    94000000          bl       0 <flush_tlb_all>
  20:    a8c107e0          ldp      x0, x1, [sp], #16
  24:    94000000          bl       0 <main>

0000000000000028 <secondary_cpu_boot>:
  28:    aa0003f3          mov      x19, x0
  2c:    d2820001          mov      x1, #0x1000                // #4096
  30:    9b017c02          mul      x2, x0, x1
  34:    58000163          ldr      x3, 60 <secondary_cpu_boot+0x38>
  38:    8b030042          add      x2, x2, x3
  3c:    91400442          add      x2, x2, #0x1, lsl #12
  40:    9100005f          mov      sp, x2
  44:    90000003          adrp     x3, 0 <empty_page>
  48:    d5182003          msr      ttbr0_el1, x3
  4c:    d5033fdf          isb
  50:    94000000          bl       0 <flush_tlb_all>
  54:    aa1303e0          mov      x0, x19
  58:    94000000          bl       0 <secondary_start>
         ...
```

可以看到在 start_kernel() 中实现了基页表寄存器的配置， flush 掉 tlb 之后进入 main() 函数继续执行。

```
0000000000000000 <main>:
  ...
  d4:    f9400000          ldr      x0, [x0]
  d8:    94000000          bl       0 <printk>
  dc:    94000000          bl       0 <init_fpu_owner_locks>
  e0:    aa1303e0          mov      x0, x19
  e4:    94000000          bl       0 <enable_smp_cores>
  e8:    90000000          adrp     x0, 0 <main>
  ec:    f9400000          ldr      x0, [x0]
  f0:    94000000          bl       0 <printk>
  f4:    94000000          bl       0 <disable_fpu_usage>
  ...
```

main() 函数被定义在 main.c 中，后续会继续进入 enable_smp_cores 执行。

```
void enable_smp_cores(paddr_t boot_flag)
{
          int i = 0;
          long *secondary_boot_flag;

          /* Set current cpu status */
          cpu_status[smp_get_cpu_id()] = cpu_run;
          secondary_boot_flag = (long *)phys_to_virt(boot_flag);
          for (i = 0; i < PLAT_CPU_NUM; i++) {
                    secondary_boot_flag[i] = 1;
                    flush_dcache_area((u64) secondary_boot_flag,
                                        (u64) sizeof(u64) * PLAT_CPU_NUM);
                    asm volatile ("dsb sy");
                    while (cpu_status[i] == cpu_hang)
                    ;
                    kinfo("CPU %d is active\n", i);
          }
          /* wait all cpu to boot */
          kinfo("All %d CPUs are active\n", PLAT_CPU_NUM);
          init_ipi_data();
}
```

enable_smp_cores 被定义在 /Lab4/kernel/arch/aarch64/machine/smp.c 中，通过函数定义可以看到 secondary_boot_flag 是 boot_flag 从物理地址转换为虚拟地址的结果，所以我们用于阻塞其他cpu的 secondary_boot_flag 其实是虚拟地址。

进一步地，我们可以看到 enable_smp_cores() 通过循环遍历各个核心的方式，
将 secondary_boot_flag 用 flush_dcache_area() 传递给不同的核心。

```
0000000000000000 <flush_dcache_area>:
   0:    d53b0023          mrs     x3, ctr_el0
   4:    d503201f          nop
   8:    d3504c63          ubfx    x3, x3, #16, #4
   c:    d2800082          mov     x2, #0x4                   // #4
  10:    9ac32042          lsl     x2, x2, x3
  14:    8b010001          add     x1, x0, x1
  18:    d1000443          sub     x3, x2, #0x1
  1c:    8a230000          bic     x0, x0, x3
  20:    d50b7e20          dc      civac, x0
  24:    8b020000          add     x0, x0, x2
  28:    eb01001f          cmp     x0, x1
  2c:    54ffffa3          b.cc    20 <flush_dcache_area+0x20>  // b.lo, b.ul, b.last
  30:    d5033f9f          dsb     sy
  34:    d65f03c0          ret
```

flush_dcache_area() 被定义在 tools.S 中，可以看到其在这里的作用是通过缓存类型寄存器来不断将 secondary_boot_flag 刷新到内存中，以此来将其传递给其他的的cpu核心。这便是其赋值的方式。

练习题1

循环枚举每一个cpu核对应的列表，分别将其初始化。具体的操作需要对queue_meta类型下的每一个字段进行初始化，找到 queue_meta 的定义：

```
struct queue_meta {
        struct list_head queue_head;
        unsigned int queue_len;
        struct lock queue_lock;
        char pad[pad_to_cache_line(sizeof(unsigned int)
                                   + sizeof(struct list_head)
                                   + sizeof(struct lock))];
};
```

queue_head 为 struct list_head 类型，使用 init_list_head() 将其初始化； queue_len 为 int ，这里赋值成0就可以； queue_lock 为 struct lock 类型，使用 lock_init() 初始化。

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

for(int i = 0; i < PLAT_CPU_NUM; i++)
{
        init_list_head(&(rr_ready_queue_meta[i].queue_head));

        lock_init(&(rr_ready_queue_meta[i].queue_lock));

        rr_ready_queue_meta[i].queue_len = 0;
}

/* LAB 4 TODO END (exercise 1) */
```

练习题2

执行入队操作，调用 `list_append()` 将 `thread->head` 插入到当前cpu对应的调度队列中。另外别忘了给meta的列表长度加一。

```
/* 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

类似，调用 `list_del()` 执行出队，然后把列表长度减一。

```
/* 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

调用一次 `rr_sched_enqueue()` 将 `old` 重新入队。

```
/* LAB 4 TODO BEGIN (exercise 4) */
/* Refill budget for current running thread (old) and enqueue the current thread.*/

rr_sched_enqueue(old);

/* LAB 4 TODO END (exercise 4) */
```

练习题5

- 第一步：读取寄存器 `cntfrq_el0` 的值，赋值给cntp_freq，可以参照原函数上面的语句模仿实现
- 第二步：计算 `cntp_tval` ，查手册可以看到时钟频率的单位是Hz，也就是1/s，而我们的TICK_MS是10/ms，所以进行一个单位的统一之后相乘
- 第三步：运行msr指令，将cntp_tval写入寄存器

```
/* 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) */
```

- 第四步：计算 `timer_crl` ，根据tutorial中的说明，控制寄存器英高使得时钟开启，且时钟中断不屏蔽，因此令 `timer_crl=1` 即可。
- 第五步：写入寄存器。

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

timer_ctl = 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) */
```

练习题6

在 `irq.c` 中，补全switch语句的case，按照注释要求调用 `handle_timer_irq()` 然后返回。

```
  /* 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) */
```

在 timer.c 的 handle_time_irq() 中，判断当前线程所拥有的时间片是否充足，充足则直接将其减一即可。

```
  /* LAB 4 TODO BEGIN (exercise 6) */
  /* Decrease the budget of current thread by 1 if current thread is not NULL */
  /* We will call the sched_periodic in the caller handle_irq so no need to call sched()

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

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

根据注释提示，找到 policy_rr.c 中定义的 rr_sched_refill_budget() ，在原来的代码基础上补全，使得在线程old重新入队之前，恢复其调度时间片 DEFAULT_BUDGET 。

```
  /* LAB 4 TODO BEGIN (exercise 4) */
  /* Refill budget for current running thread (old) and enqueue the current thread.*/

  rr_sched_refill_budget(old, DEFAULT_BUDGET);
  rr_sched_enqueue(old);

  /* LAB 4 TODO END (exercise 4) */
```

练习题7

在 register_server() 中配置好 config 的 routine 和 cb_thread 字段。

```
  /* 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) */
```

在 `create_connection()` 中，实现所建立的连接的相关参数和共享内存参数的对齐：

```
  /* 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) */
```

在 `ipc_thread_migrate_to_server()` 中，前面两行为新分配的IPC回调线程分配栈空间，并设置下一指令地址。后面四行参考 `kernel/user-include/uapi/ipc.h` 中的定义可以分别填入四个参数。

```
  /* 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) */
```

接下来在 sys_register_client 中简上分配栈和next_ip，然后最后一行传入server_config中已经填写好的服务端入口 declared_ipc_routine_entry 。

```
/* 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) */
```

最后一部分，在 sys_ipc_register_cb_return 中，将服务器的共享内存地址设置好：

```
/* 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) */
```