# Lab 4 实验报告

潘文峥（学号：520030910232）

## 思考题1：

在 `start.S` 中，首先将CPU的ID从系统寄存器 `mpidr_el1` 存到 `x8` 中，并通过与 `0xFF` 相与判断是否为0号CPU（主核），如果是则跳转到 `primary` 执行主核的初始化；而其他CPU将继续执行并被阻塞在 `wait_until_smp_enabled`，直到主核初始化完毕并设置 `secondary_boot_flag` 后才执行 `secondary_init_c` 进行其他CPU的初始化。

## 思考题2：

是虚拟地址，因为MMU已启动；是通过main函数的参数 `boot_flag` 传入 `enable_smp_cores` 的。

## 练习题3：

在 `secondary_start` 函数中， `cpu_status[cpuid] = cpu_run` 设置当前cpu状态；

在 `enable_smp_cores` 函数中，通过 `secondary_boot_flag[i] = 1` 设置flag，并通过 `while(cpu_status[i] == cpu_hang);` 进行等待下一个cpu.

## 练习题4：

1. unlock时将 `lock->owner` 加一，将当前排号置为下一个表示放锁；并通过检查当前号是否等于下一个号判断是否上锁： `ret = lock->owner < lock->next` .
2. 通过封装好的 `lock_init` , `lock` , `unlock` 函数完成锁的获取、释放和初始化。
3. 在 `irq_entry.c` 中异常处理和中断处理函数中拿锁。
4. 在 `irq_entry.S` 中同步异常处理、系统调用结束以及线程切换时放锁。

## 思考题5：

不需要。因为后续将回到用户态并恢复用户态寄存器上下文，寄存器原值不再被使用。

## 思考题6：

因为等待队列中只应该放入有意义的用户态线程，而 `idle_threads` 表示没有需要执行的线程时调用，若放入等待队列随时间片轮转，则会造成时间片浪费。

## 练习题7：

通过补充`rr_sched_enqueue`，`rr_sched_dequeue`，`rr_sched_choose_thread`与`rr_sched`函数，完善了`policy_rr.c`中的调度功能。

## 思考题8：

原因是大内核锁是ticket lock，在空闲线程执行过程中，遇到待执行的用户态线程将切回用户态线程，此时要做unlock，若不获取锁，则排号被跳过，永远无法获取锁。

## 练习题9：

完成`sys_yield`函数：

```
struct thread *thread = current_thread;
        BUG_ON(!thread);
        thread->thread_ctx->sc->budget = 0;
sched();
eret_to_thread(switch_context());
```

## 练习题10：

在main函数中主CPU以及其他CPU的初始化流程中加入`timer_init`函数的调用。

## 练习题11：

完成`sched_handle_timer_irq`函数：

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

## 练习题12：

在`thread.c`中完成亲和性设置：

```
thread->thread_ctx->affinity = aff;
```

## 练习题13：

完善`launch_process`函数，完成对进程的创建。

## 练习题14：

在`libchcore/src/ipc/ipc.c`与`kernel/ipc/connection.c`中实现了IPC相关的代码，运行lab4.bin之后输出如下：

```
Hello from ChCore Process Manager!
Hello from user.bin!
Hello from ipc_client.bin!
IPC test: connect to server 2
IPC no data test .... Passed!
IPC transfer data test .... Passed!
IPC transfer cap test .... Passed!
IPC transfer large data test .... Passed!
20 threads concurrent IPC test .... Passed!
```

## 练习题15：

完成`wait_sem`和`signal_sem`函数：

```
s32 wait_sem(struct semaphore *sem, bool is_block)
{
        s32 ret = 0;
        /* LAB 4 TODO BEGIN */
        if (sem->sem_count == 0) {
                if (is_block) {
                        current_thread->thread_ctx->state = TS_WAITING;
                        list_append(&(current_thread->sem_queue_node), &(sem->waiting_threads));
                        ++sem->waiting_threads_count;
                        obj_put(sem);
                        sched();
                        eret_to_thread(switch_context());
                } else ret = -EAGAIN;
        }
        else --sem->sem_count;
        /* LAB 4 TODO END */
        return ret;
}

s32 signal_sem(struct semaphore *sem)
{
        /* LAB 4 TODO BEGIN */
        if (sem->waiting_threads_count > 0) {
                struct thread *awake = list_entry(sem->waiting_threads.next, struct thread, sem_queue_node);
                list_del(&awake->sem_queue_node);
                --sem->waiting_threads_count;
                sched_enqueue(awake);
        }
        else ++sem->sem_count;
```

```
        /* LAB 4 TODO END */
        return 0;
}
```

## 练习题16：

实现producer和consumer：

```
void *producer(void *arg)
{
        int new_msg;
        int i = 0;

        while (i < PROD_ITEM_CNT) {
                /* LAB 4 TODO BEGIN */
                __chcore_sys_wait_sem(empty_slot, 1);
                /* LAB 4 TODO END */
                new_msg = produce_new();
                buffer_add_safe(new_msg);
                /* LAB 4 TODO BEGIN */
                __chcore_sys_signal_sem(filled_slot);
                /* LAB 4 TODO END */
                i++;
        }
        __sync_fetch_and_add(&global_exit, 1);
        return 0;
}

void *consumer(void *arg)
{
        int cur_msg;
        int i = 0;

        while (i < COSM_ITEM_CNT) {
                /* LAB 4 TODO BEGIN */
                __chcore_sys_wait_sem(filled_slot, 1);
                /* LAB 4 TODO END */
                cur_msg = buffer_remove_safe();

                /* LAB 4 TODO BEGIN */
                __chcore_sys_signal_sem(empty_slot);
                /* LAB 4 TODO END */
                consume_msg(cur_msg);
                i++;
        }
        __sync_fetch_and_add(&global_exit, 1);
        return 0;
}
```

## 练习题17：

```c
void lock_init(struct lock *lock)
{
        /* LAB 4 TODO BEGIN */
        lock->lock_sem = __chcore_sys_create_sem();
        __chcore_sys_signal_sem(lock->lock_sem);
        /* LAB 4 TODO END */
}

void lock(struct lock *lock)
{
        /* LAB 4 TODO BEGIN */
        __chcore_sys_wait_sem(lock->lock_sem, 1);
        /* LAB 4 TODO END */
}

void unlock(struct lock *lock)
{
        /* LAB 4 TODO BEGIN */
        __chcore_sys_signal_sem(lock->lock_sem);
        /* LAB 4 TODO END */
}
```

如上，使用内核信号量实现了阻塞互斥锁。