Lab4 实验报告

- 于凡奇 18307130182

习题一

为了确保你完全掌握了多核的启动流程，请简要描述一下 `kern/entry.S` 中各个 CPU 的初始状态如何、经历了哪些变化？至少包括对 PC、栈指针、页表的描述。

所有CPU从 armstub8.S 的代码开始执行，但只有 BSP 跳转到了0x80000处执行 entry.S 的代码，其余 AP 停留在 `wfe` 指令处。

BSP

1. BSP 从 `_start` 开始执行，分别向三处内存地址 `0xd8 + cpuid()` 写入 `mp_start` 的地址。随后执行 `dsb` 和 `isb` 指令进行同步，保证在内存写入完成后再执行后面的指令 `sev`（否则其它 AP 可能在内存写入之前就进行入口值是否为零的判断，导致失败）。执行 `sev` 后，其它3个 AP 从 `wfe` 中恢复运行，4个CPU开始并行执行。
2. 在 BSP 上，继续进行之前和 Lab1 Booting 中类似的操作：
    - 逐步降低 BSP 上的异常等级；
    - 配置页表，使得 `ttbr0` 和 `ttbr1` 两块页表映射到同一片物理内存上；
    - 开启 MMU 。
3. 之后获取 cpuid ，并将栈指针 `sp` 设为 `_start - cpuid() * PGSIZE`。至此，准备工作已经完成，CPU 跳转至 `main` 函数入口。

AP

AP 与 BSP 相比，没有第1步，但第2、3步与之完全相同。3个 AP 都会跳转至 `mp_start` 处开始执行，所以它们的 PC 值都是一样的。由于执行的页表配置的代码相同，AP 与 BSP 的页表也是一样的。不同之处在于，每个 AP 拥有各不相同的栈指针。

习题二

如果开启内核中断，如果某一个 CPU 在获取锁后遇到中断，那么中断处理程序和其它的 CPU 在此期间都将无法获得锁，直到该中断返回。解决方案可以是在中断处理程序的开头释放锁。

习题三

```
if (cpuid() == 0) {
    /* TODO: Use `memset` to clear the BSS section of our program. */
    memset(edata, 0, end - edata);
    /* TODO: Use `cprintf` to print "hello, world\n" */
    console_init();
    alloc_init();
    cprintf("Allocator: Init success.\n");
```

```
    check_free_list();

    irq_init();

    started = 1;
}

while (started == 0)
    ;

lvbar(vectors);
timer_init();

cprintf("CPU %d: Init success.\n", cpuid());
```

- `memset()` 清理内存，在页表相同的情况下让哪个 CPU 做都一样，故只用调用一次。
- `console_init()` 对 UART 外设进行初始化，需向外设写入相关参数，外设属于共享资源，初始化仅需进行一次，且不应同时访问。加锁后由几个 CPU 多次调用这个函数应该也是无害的，只是这样做没有必要。
- `alloc_init()` 对内存进行初始化，该函数只能被调用一次（无论是否加锁），因为多次调用 `free_range()` 会使 `kmem.freelist` 中产生重复的“空闲页”。
- `irq_init()` 配置全局中断，由于全局中断都被路由到 CPU0 ，这个函数也只需要在 CPU0 上执行一次。
- `timer_init()` 函数需要分别对4个 CPU timer 进行初始化，故需要在4个 CPU 中分别调用。
- `lvbar(vectors)` 需要初始化每个 CPU 的 `vbar_el1` 寄存器，故也需要在4个 CPU 中分别调用。