Lab 4: Multicore and Locking

习题解答

1. 多核启动流程

> 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。

初始状态时，所有 CPU 的 PC 从地址 `0x0` 开始。启动流程中，CPU0 (BSP, Bootstrap Processor) 的 PC 先跳转到 `_start`；其他 CPU (APs, Application Processors) 执行死循环（`wfe`，未收到外部 event 信号时约等于 `nop`）等待，直到收到 BSP 发出的 event 信号（`sev`）后唤醒，随后跳转到 `mp_start`。

启动时，CPU 首先根据 CurrentEL (Current Exception Level) 决定当前是以 EL2 或者 EL3 启动。

- 如果当前是 EL3，CPU 将初始化 SCR (Secure Configuration Register), SPSR (Saved Program Status Register), ELR (Exception Link Register)，并切换 exception level 到 EL2
- 如果当前是 EL2，CPU 将初始化 HCR (Hyp Configuration Register), SCTLR (System Control Register), SPSR, ELR，并切换 exception level 到 EL1
- 切换到 EL1 后，CPU 将初始化 TCR (Translation Control Register), MAIR (Memory Attribute Indirection Register), MMU (Memory Management Unit), SP (Stack Pointer)，其中栈指针 SP 的地址设置为 `_start - cpuid() * PGSIZE`（需 16 bytes 对齐），并将 `ttbr0` 和 `ttbr1` 页表均映射到地址为 `kpgdir` 的物理内存空间，最后跳转到主循环 `main`，启动 kernel

2. 自旋锁时为什么要关中断

> 请阅读 `kern/spinlock.c` 并思考一下，如果我们在内核中没有关中断的话， `kern/spinlock.c` 是否有问题？如果有的话，应该如何修改呢？

如果没有在当前内核中关中断，那么当本地的中断处理程序需要访问某个被自旋锁的资源时，会开始死循环以等待被锁的资源释放。但是中断处理程序的优先级高于其他系统调用或者用户程序，这意味着目前占用这个资源的程序将永远得不到机会释放被锁的资源，于是就形成了一个死锁。

因此如果内核没有关中断，我们就需要在自旋锁的实现（`kern/spinlock.c`）里关中断。具体来说，就是在加锁前关中断，解锁后开中断。可参考 xv6-riscv 中的实现 [1]。

3. 给主循环加锁

> 注意到所有 CPU 都会并行进入 `kern/main.c:main`，而其中有些初始化函数是只能被调用一次的，请简单描述一下你的判断和理由，并在 `kern/main.c` 中加锁来保证这一点。

以下初始化函数只能被调用一次：

- `console_init`：初始化控制台，只需调用一次即可
- `alloc_init`：初始化内存和页表，只需调用一次即可

其余初始化函数则需要每个内核都调用一次。

这里我们通过一个自旋锁 `started_lock` 来确保以上初始化函数只会在 CPU0 (BSP) 启动过程中被调用一次。当 CPU0 启动完成后，解锁 `started_lock`，唤醒其他 CPU (APs)。具体代码如下：

```
// kern/main.c

struct spinlock started_lock = {1};

void
main()
{
    extern char edata[], end[], vectors[];

    if (cpuid() == 0) {
        memset(edata, 0, end - edata);
        console_init();
        cprintf("CPU 0: Init started.\n");
        alloc_init();
        cprintf("Allocator: Init success.\n");
        check_map_region();
        check_free_list();
        irq_init();
        lvbar(vectors);
        timer_init();
        started_lock.locked = 0;  // allow APs to run
    } else {
        while (started_lock.locked) {}
        cprintf("CPU %d: Init started.\n", cpuid());
        irq_init();
        lvbar(vectors);
        timer_init();
    }
    cprintf("CPU %d: Init success.\n", cpuid());

    while (1) {}
}
```

测试环境

- OS: Ubuntu 18.04.5 LTS (WSL2 4.4.0-19041-Microsoft)
- Compiler: gcc version 8.4.0 (Ubuntu/Linaro 8.4.0-1ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Debugger: GNU gdb 8.2 (Ubuntu 8.2-0ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Emulator: QEMU emulator version 5.0.50
- Using GNU Make 4.1

---

1. mit-pdos/xv6-riscv: Xv6 for RISC-V - GitHub ↩