Lab4 实验报告

习题一

第一个核执行`_start`，向$0xd8$开始的位置写入其余三个核初始启动程序的入口`mp_start`。写入后这些核并行独立运行，各自开始执行`mp_start`。

系统寄存器按如下顺序更改：

1. `scr_el3`：管理哪些异常会被切换到 el3 处理
   - RW, bit[10] = 1: 指定接下去更低等级在 AArch64 下执行。el2（存在 el2 ）或是 el1（不存在 el2 ）。
   - HCE, bit[8] = 1: 允许 HVC 指令在 el1 及以上的等级被处理
   - SMD, bit[7] = 1: 允许 SMC 指令在 el1 及以上的等级被处理
   - RESERVED, bit[5:4] = 11: RES1
   - NS, bit[0] = 1: el0 和 el1 处于安全模式，这样这些等级的访存可以访问被保护的内存区域
2. `spsr_el3`：保存发生异常切换到 el3 时的进程状态
   - A I F, bit[8:6] = 111: SError、IRQ、FIQ 在异常出现时被遮挡
   - M, bit[3:0] = 1001: 查表无结果
3. `elr_el3`：保存 el3 处理完成后需要跳回的位置，这里直接把地址设为 el2 函数入口
4. `hcr_el2`：类似`scr_el3`，管理哪些异常会被切换到 el2 处理
   - RW, bit[31] = 1: 指定 el1 等级在 AArch64 下执行。el0 根据 PSTATE 寄存器决定
5. `sctlr_el1`提供对系统的顶级控制，包括内存系统
   - RES1 保存, [28], [29], [22], [23], [20], [11], [8], [7] = 1
   - EE, bit [25] = 0: el1 下 tlb 上的路径搜索中数据翻译用小端法
   - E0E, bit[24] = 0: el0 下的数据访问使用小端法
   - I, bit[12] = 1: 在 el0、el1 上的访存中的指令读取是否经过 cache 无限制。
   - C, bit[2] = 1: 在 el0、el1 上的数据访问、tlb 访问是否经过 cache 无限制。
   - M, bit[0] = 0: 在 el0、el1 上关于高地址空间 (stage 1) 的访存不经过地址翻译
6. `spsr_el2`：保存发生异常切换到 el2 时的进程状态
   - A I F, bit[8:6] = 111: *SError*、*IRQ*、*FIQ* 在异常出现时被遮挡
   - M, bit[3:0] = 1001: 该异常由 supervisor 发出
7. `elr_el2`：保存 el2 处理完成后需要跳回的位置，这里直接把地址设为 el1 函数入口
8. `ttbr0_el1`和`ttbr1_el1`：el1 模式下高地址空间和低地址空间 0 级页表的入口地址，这里初始值都设为 kpgdir 这个手动初始化页表的地址。
9. `tcr_el1`：控制 el0 和 el1 的地址翻译
   - T0SZ, T1SZ = 64 - 48: 设定高地址空间和低地址低地址大小为 48 位共 2^{48} Bytes
   - TG0 = 00, TG1 = 10: 低地址空间页面大小 4KB，高地址空间页面大小 16KB
   - SH0, SH1 = 3: 高低地址的 cache 共享，只有 inner cache（一级缓存等）是共享的，inner cache 的修改要进行广播
   - (IRGN0, ORGN0), (IRGN1, ORGN1) = 5: inner cache 和 outer cache 使用写回模式，别修改了才写回
10. `mair_el1`：内存属性编码
11. `sctlr_el1`：提供对系统的顶级控制，包括内存系统
    - M, bit[0] = 1: 在 el0、el1 上关于高地址空间 (stage 1) 的访存经过地址翻译
    - 其余不变

12. `sp`：当前使用的栈指针，这里选的是 `spsr_el1`。令第 i 个核的 `sp` 值为 $\_start - i * PGSIZE$，即每个核有 2048 Bytes 大小的栈空间，从 $\_start$ 开始并紧挨着上一个核的栈空间。
13. `spsel`：选择哪个异常等级的栈指针。设为 1 表示使用 el1 的栈指针。

在`mp_start`中，读取当前异常等级并判断，以当前在 el3 为例，会先进入 `el3` 函数，初始化完成 el3 等级有关寄存器之后，修改 `elr_el3` 寄存器的值为 `el2` 函数入口，通过 `eret` 指令让处理器退出 el3 等级进入 el2 等级并跳转到 `el2` 函数，在 el2 等级下完成 `el2` 函数中相关初始化后进入 `el1` 函数。

习题二

有问题，如果进程 A 在对资源 x 加上锁后，又出现了一个外部中断 B，B 也需要访问资源 x 并加锁，这样就会和 A 的锁发生冲突。
需要在加锁之前关闭中断，解锁后打开中断。

习题三

对 bss 段的赋初始值 0 操作`memset(edata, 0, end - edata)`：只需要最开始赋初始值即可，所有核共用同一个 bss 段内存。若每个核执行一次会让其它核对 bss 段数据的修改清空。

`irq_init()`：函数只需要对特定内存地址写入数据，不需要每个核都写，只要一个核写一次即可。

加锁方式

在 `main.c` 中的 `memset` 函数上加锁，同时添加全局计数器变量 cnt，执行一次 `memset` 后 cnt 置 1，其它核即使得到锁也不会执行 `memset`。需要注意的是这里的锁和 cnt 都需要在定义是设初始值，这样可以存放在 data 段而不是 bss 段，否则在 `memset` 中会把放在 bss 段的全局变量锁置为 0。

在 `trap.c` 中的 `irq_init` 函数上加锁，同时类似以上保证只有一个核会执行。

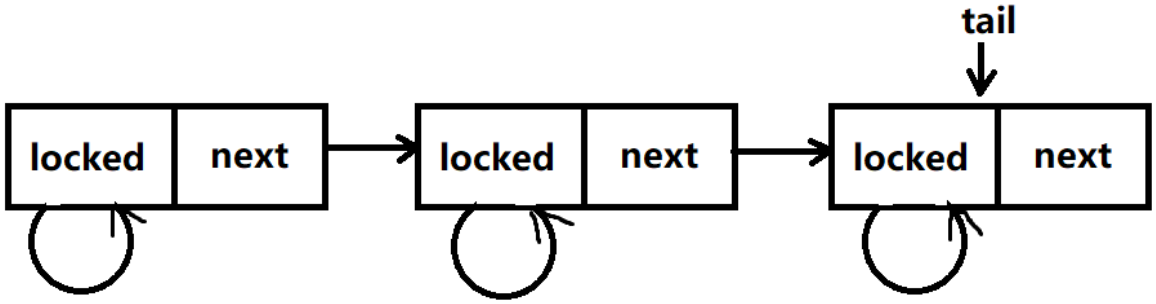
在 `kalloc.c` 中的 `alloc_init` 函数上加锁，保证只有一个核会执行。

在 `kalloc.c` 中的 `alloc_init`、`check_free_list`、`kalloc` 函数上加锁，三个函数公用一个锁，保证只有一个函数会修改或访问全局变量 $kmem$。

锁的实现

在多核或众核机器中，由于每个核心的部分 cache 不共享，而且对内存亲和性也不同（NUMA），如果所有核心都在同一全局变量（同一内存地址）上自旋，某一核心对锁的修改需要和其余核同步，涉及到芯片上长距离通信或 cache 和内存的同步，造成很大的性能损失。

在 `spinlock.c` 中实现了 mcs 锁，改善这类损失。



如图时 mcs 锁的结构示意图，当需要使用锁时需要新建一个节点，所有的节点按申请时间形成链表结构。每个节点有一个 flag，询问是否可以得到该资源不必访问全局锁，只需要询问该节点的 flag 值是否为 0。另外，flag 值仅会在前一个节点锁被解锁的时候被修改，这样核心可以一直在本地 cache 上对锁进行自旋访问，每个锁只有在资源可用和解锁时需要和全局内存进行一次同步，提高访存的效率。

另外还有 clh 锁，与 mcs 锁类似，也是维护了一个节点链表结构，没有在全局锁上进行自旋访问。clh 锁是在前一个节点的 flag 上进行自旋访问，解锁时释放前一个锁的节点，并且把自己节点的 flag 设为 0 。

在实现上，由于 AArch64 中不支持使用 `malloc` 和 `new` 函数，无法在堆空间中手动分配、释放内存资源，在实现 mcs 锁的时是在函数的栈空间中自动分配、释放锁变量，在实现上有些麻烦。同时由于这个原因，没有实现 clh 锁。另外，在 gcc 中对锁队列全局变量 tail 的读写要使用 `__atomic_` 的 gcc 函数使得这些操作原子化。

在 `vm.c` 中还对 `kalloc` 函数进行测试，测试内容是对每个核都建立 $0x000000 + (cpuid << 12)$ 向 $0x006000 + (cpuif << 12)$ 的页表映射，在 `map_region` 和 `pgdir_walk` 函数中均不加锁，直接使用 `kalloc` 的细粒度锁，看结果是否正确。