lab4 实验报告 18307130104 赵文轩

赵文轩 18307130104

多核的启动流程（习题一）

由于 armstub8.S 会将 cpu id 为 0 的 cpu 直接跳转到 0x80000 的位置，让其他 cpu 进行死循环，因此我们首先考虑 id 为 0 的 cpu。

id 为 0 的 cpu

从 entry.S 的 _start 开始执行。首先获得 mp_start 的地址，再将其他三个 cpu 入口值（0xd8 + (cpuid() << 3) ）设置成 mp_start 的地址，唤醒其他 cpu。所有 cpu 一同进入 mp_start 开始启动。

启动流程

id 为 0 的 cpu 再完成上述步骤之后进入该步骤。而其他 cpu 在被 id 为 0 的 cpu 唤醒之后直接进入这一步骤。

和单核的启动流程相同，首先需要将 Exception Level 调整到 EL1。调整到 EL1 之后，需要进行对于物理内存和虚拟内存的映射，对于不同的 cpu ，这一步的操作是完全相同的，也就是说，所有 cpu 对虚拟内存到物理内存的映射具有一直的方式。但是不同于虚拟内存，不同 cpu 具有不同的栈空间和栈指针 sp。具体来说，_start 节代码段之后分出了 4 个页，每个页都分配给不同的 cpu 作为栈空间，页开头的地址就是 sp。完成虚拟内存的映射和栈空间的分配后，跳转到 main 进行初始化。

未关闭中断状态下的锁（习题二）

如果操作系统在内核模式下没有关闭中断，可能导致当前代码获得锁之后，被中断处理程序打断而去竞争同一个锁，从而导致死锁。

解决这个问题可以采用递归锁，或者进入中断程序之前释放当前持有的所有锁。考虑到进入中断程序之前释放持有的锁不仅和锁机制的初衷相违背，而且还要考虑复原问题，我认为递归锁是一个比较合适的解决方案。

xv6 中实现了睡眠锁（sleep lock），主要应对需要持锁进入睡眠的情况（IO 操作）。睡眠锁是不需要关闭中断的。

初始化函数加锁（习题三）

加锁函数的选择和原则

初始化函数需要考虑的问题主要针对内存访问的方面。显然对于仅仅初始化 cpu 内部指针的函数是不需要加锁的，因为树莓派是 SMP 架构的，每个 cpu 都会有自己单独的一套寄存器，对寄存器的修改和访问不涉及共享区域。根据初始化函数是否涉及寄存器的读取和修改可以知道 lvbar 函数必须每个 cpu 执行一次。

对于涉及内存访问和修改的函数，我选择能少运行的就少运行。诸如 memset，console_int，check_free_list 函数，虽然重复运行对系统正确性不会造成影响，但是我仍然选择只运行一次。irq_init 函数中包含对于时钟的初始化，应该是必须只运行一次。alloc_init 函数包含对于页表的清空，多次调用会导致同一个页表在 free_list 上出现多次，导致后面程序的页表重复分配。

timer_init() 函数中，虽然出现了对于内存的访问和修改。但是不仅包含了对于寄存器的修改，对与内存的修改也是根据 cpuid 选择不同的位置，因此能够保证不会产生冲突，也是必须每个 cpu 执行一次。

总的来说，一个基本的原则就是：涉及寄存器修改的函数每个 cpu 执行一次，涉及共享内存的函数总共只能执行一次。

加锁的实现

如何通过加锁保证当前函数在所有 cpu 上只会执行一次，方法就是另外设置一个变量表示当前函数是否被执行过，然后对这个变量加锁，第一个得到锁的 cpu 将该变量设置成 cpuid，执行只能执行一遍的函数时判断该变量是否等于 cpuid。实际上这个方法和指定一个 cpu 来执行这些函数并没有什么本质上的区别，甚至还不那么高效。将变量赋值成 cpuid 的好处是只需要修改完变量就可以直接释放锁，而不需要等待所有函数执行完再进行释放，勉强算是一个小优化。

遇到的问题

一开始发现锁会同时被 4 个 cpu 获取，进而导致释放锁的时候产生“未上锁”的错误。在~~徐大爷~~徐逸培同学的帮助下，发现了如果定义的锁不进行初始化会被放置在 .bss 段，在 memset 中被清零，因此在定义锁的时候需要先初始化一下。

```
acquire(&lk);
if(ex == -1){
    ex = cpuid();
}
release(&lk);
if(ex == cpuid()){
    // excute once
}
// other
```

其他类型的锁

稍微试了一下 mcs 和 读写锁两种锁

读写锁

想写一下读写锁的初衷是觉得控制函数执行一次的锁特别适合读写锁。但是实际上实现了之后发现并不是这样的。

读写锁需要考虑的一个问题是是否提供保证不释放锁的情况下，进行写锁和读锁的互相转换的转换。

读锁向写锁的转换。由于持有读锁的时候肯定是持有写锁的，所以需要等待其他读锁释放，然后释放当前读锁。如果同时存在两个读锁希望换成写锁，那么就会造成死锁。

写锁向读锁的转换。显然持有写锁的时候读锁是不会被持有的，所以只需要请求读锁即可。所以这个功能是原本设计就自带的。

所以在我的设计中读写锁不能存在读锁在不释放的情况下变成写锁，这就导致了我之前的设计无法愉快地分开使用读写功能。

mcs 锁

mcs 锁遇到的最大问题就是 __atomic_test_and_set 函数的第二个参数要求是 int 而不能是用户指定类型，以及 __atomic_compare_and_swap 这种函数是不存在的。所以我找到了__sync_ 系列操作，虽然官方更加推荐使用 __atomic_ 系列指令。

只需要为每个 cpu 多定义一个 lk_i 即可。不管是定义成全局变量（需要每个 cpu 定义一个），还是局部变量，只要不定义在 bss 段就行。

```
struct mcslock lk_i;
mcs_acquire(&lk, &lk_i);
if(ex == -1) ex = cpuid();
mcs_release(&lk, &lk_i);
if(ex == cpuid()){
    // excute once
}
// other
```

为 kalloc.c 和 vm.c 增加锁

加锁的原则是：访问可能引起冲突的公共内存时，需要加锁。

因此 kalloc.c 中需要在访问 kmem 之前，请求锁 kmemlk，完成后释放。值得一提的是，这里对 kmemlk 加锁非常适合读写锁，check_free_list 中请求读锁，kalloc 中申请写锁，kfree 中申请写锁。

vm.c 中的 pgdir 是不需要加锁的。