# 2024/3/14~2023/3/16

一旦涉及到内存和文件系统，在编码的时候就要格外小心。对于这个lab，xv6能够顺利启动已经成功一大半了；实验侧重考察的锁的使用反而不那么困难。

## XV6第八章

讨论的都是文件系统，放在lab fs里细说。

这个实验需要详细地了解了spinlock的生效机制，便编码

## Lab8 Lock

### Memory allocator

多个cpu使用一个freelist以及保护freelist的锁来管理内存，会导致竞争。

于是，为每个cpu都建立一个freelist以及对应的锁，初始化时比较省事地把内存都分给了cpu0，当每个cpu的内存不够的时候，就遍历获取每个freelist的锁，获得某一freelist访问权后从freelist移动一块内存到当前CPU。

代码不贴了，在gitee仓库里的kalloc.c下。

PS:写这条时突然想起如下的代码极有可能引发cid和i之间的死锁，但在测试中均正常通过了。。。

void \*

kalloc(void)

{

  struct run \*r;

...

  int cid=cpuid();

  acquire(&kmem[cid].lock);

  ...

    acquire(&kmem[i].lock);

    r = kmem[i].freelist;

    if(r){

      kmem[i].freelist= r->next;

      release(&kmem[i].lock);

      break;

    }

    release(&kmem[i].lock);

  }

  release(&kmem[cid].lock);

...

}

### Bcache

第一个的升级版，管理文件读写的缓冲区。

将磁盘块号映射到哈希桶中，磁盘块只能映射到对应桶中的buffer，如果桶中buffer未命中，则在全局buffer中找到最久未使用的buffer，将该buffer移动带当前桶中，并与当前硬盘块建立映射。对每个桶单独上锁，而不是之前的一次性对整个磁盘上锁。

别的不多说了，说一下锁的思路：

对于函数bget，首先要acquire目标哈希桶（内装buffer）的锁，如果在目标桶内buffer命中，就直接release掉这个锁并且返回。但是如果没找到，就需要根据LRU定位最久未使用的buffer以及其存在的源哈希桶，显然要把这个buffer从源哈希桶移动到目标哈希桶中。

这时候需要获取源哈系统的锁，显然是会发生死锁的：

* + - *进程1：目标桶已获取，想要源桶，发生进程切换；*
    - *进程2：源桶已获取，想要目标桶。死锁*

显然，桶间并行在需要buffer换桶的情况下是不现实的，如果需要移动buffer，需要对整个桶上锁。

于是，思路为：

* + - *当buffer正常命中时，所有桶操作并行执行。*
    - *当buffer miss时，所有桶操作停止，等待miss的替换操作。*

第一想法即在找到目标buffer前全程持有目标桶锁，在桶miss后持有大锁，再获取源桶锁，buffer移动完成后释放源桶锁，目标buffer获取后释放大锁和目标桶锁。

但是，这种方法显然违背了锁的序列化。存在如下两种获取锁顺序：

* + - *获取（目标）桶锁，获取大锁*
    - *获取大锁，获取（源）桶锁*

死锁将会发生。

一种解决方法是，在桶miss，获取大锁之前，释放掉目标桶锁。但这种方案存在潜在的风险，即：

* + - *进程1：桶miss，释放掉目标桶锁，进程切换。*
    - *进程2：桶miss，迅速地完成了miss后的替换。*

此时，进程1恢复执行的时候，想要替换的buffer已经存在在目标桶、并且已经和硬盘中的某一块建立了映射；但进程1毫不知情！它将会重复进程2进行的操作，这会导致同一硬盘块被映射了两次，会出现bcache不一致的灾难性后果。

解决方案为：进程1在恢复执行之后，获取大锁->获取目标桶锁->再次执行buffer查找->获取源桶锁->buffer替换->……

这样就避免了进程1对于miss 的buffer已经被加载这一事件毫不知情。

代码不贴了，都在gitee。

### 总结

第二个bcache实验其实和mem allocator大同小异。只是其中死锁和竞争现象会更激烈地表现在xv6中（比如无法正常启动），导致debug异常困难……最开始我一直怀疑是自己的编码有疏忽，重新整理编码后才考虑到锁的问题上来。同为内核级编程，同是有bug就炸机，困难程度不亚于pgtbl