Lab 实验报告 VIII

Lab: Lock

丁睿

dromniscience@gmail.com

更新：2020 年 12 月 9 日

目录

1 任务完成清单

Subtask	Done?	Time
Memory allocator	Y	1h
Buffer cache	Y	2h

Grade: 70/70

我的作业已开放在这个网页上。

2 详细情况 & 困难和收获

本次 lab 可以看作 multithread 的拓展。它围绕空闲页分配器和文件系统的缓存展开，核心的要求是利用功能需求修改已有数据结构使得互斥锁的设计粒度更加精细 (fine-grained)。

特殊的一个任务场景是 Buffer Cache 这个子任务下，它放宽了对各种竞争条件的要求。因此，在鲁棒性和性能之间取得平衡成为了一个重要的考量因素。由于我开始片面最求最严格的正确性，导致评测时有一定几率超时 (并非死锁)。后来我根据它提供的假设注释掉一层加锁要求，运行时间便降低为原来的 20% 并顺利通过测试。

我提交的代码 (kernel/bio.c) 中以注释的方式保留了我的最初版本。我将简单分析二者性能差距的原因。

2.1 Memory Allocator

为每一个 CPU 维护空闲页的链表将极大地细化临界区的保护要求。通过一个简单的数据结构的改变，我们有效地减少了锁的冲突。

现在介绍一些细节。当一个 CPU 的链表为空时，应当从其他的 CPU 中抢占。而如果所有的 CPU 链表均空，则说明没有空闲物理页。注意，这种数据结构的改变必须对其上的一切函数和设计透明 (transparent)。这意味着:

- `kfree`和`kalloc`的语义不变，包括出错标记；
- `ksteal`这一表示从其他 CPU 的链表中“窃取”空闲页的过程只在`kalloc`内部被调用。

为此，先将标量链表改为向量链表:

```
/* kernel/kalloc.c */
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem; // old version
```

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU]; // new version
```

其次，需要在kfree和kalloc中调用cpuid获取核的编号。由于中断引发的调度决定了这一属性需要被保护，因此要在其周围套上push_off和pop_off。一旦获得了cpuid，我们就可以开放中断，因为此时即使调度导致进程真正的 CPU 和它不符，那么我们只是简单地从旧的 CPU 的空闲链表中取一个空闲页。这不会导致死锁。

最后提及ksteal的实现。当我们调用此过程时即说明当前核无空闲页，而kalloc只是简单地从别的核上借过来一页，并不需要再用到本核的链表，从而应当在调用ksteal前释放锁以减少加锁冲突。为了避免ksteal检查 CPU 的顺序同化，我设计成从当前 CPU 的序号加 1 开始，循环地检查一轮。

以下是实验结果。从fetch-and-add的数据中我们看出锁的冲突被完全避免。

```
--- lock kmem/bcache stats
lock: kmem 0: #fetch-and-add 0 #acquire() 75929
lock: kmem 1: #fetch-and-add 0 #acquire() 178661
lock: kmem 2: #fetch-and-add 0 #acquire() 178462
lock: kmem 3: #fetch-and-add 0 #acquire() 37
lock: kmem 4: #fetch-and-add 0 #acquire() 37
lock: kmem 5: #fetch-and-add 0 #acquire() 37
lock: kmem 6: #fetch-and-add 0 #acquire() 37
lock: kmem 7: #fetch-and-add 0 #acquire() 37
```

我们可能会好奇为何 hart 2 和 3 触发了这么多次acquire，事实上，这是由于测试数据本身造成的。每个核的负载不完全一样。除了直接阅读源码，我们还可以如此验证：初始化时即在freerange内循环地把空白页挂到各个核的空闲链表下，以此控制初始局面各个核基本相同。我们得到如下结果:

```
lock: kmem 0: #fetch-and-add 0 #acquire() 47151
lock: kmem 1: #fetch-and-add 0 #acquire() 166936
lock: kmem 2: #fetch-and-add 0 #acquire() 198475
lock: kmem 3: #fetch-and-add 0 #acquire() 4091
lock: kmem 4: #fetch-and-add 0 #acquire() 4090
lock: kmem 5: #fetch-and-add 0 #acquire() 4090
lock: kmem 6: #fetch-and-add 0 #acquire() 4090
lock: kmem 7: #fetch-and-add 0 #acquire() 4090
```

可见它不是均匀分配的。实际上，参与测试的一共只有 3 个。

2.2 Buffer Cache

这一次我们在 Memory Allocator 使用的技巧不再适用，因为任意核必须有能力访问所有的硬盘缓存。上节中提到的设计透明的要求此处可以再重复一遍。但是注意，测试样例已保证

- 不会出现两个进程并发地使用同一个块；
- 不会出现两个进程并发地驱逐一个块并做替换；

首先不考虑此约束，一个合理的设计方案如下图所示：

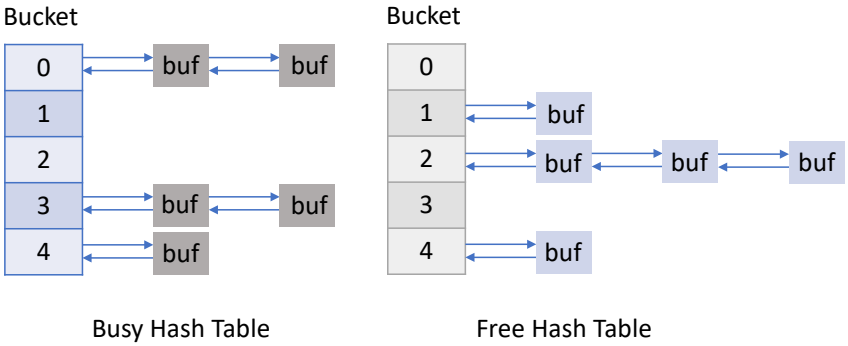

图 1: 为空闲块 (引用计数为 0) 和有效块 (引用计数不为 0) 分别维护一个开散列表，其中每个开散列以循环链表组织。图中没有画出最后一个`struct buf`的节点块中的`next`域和头节点中的`prev`域的指针。

每个桶的操作都需要一个单独的自旋锁保护，所以最初的设计里应当这样定义`bcache`变量:

```
struct {
  struct buf buf[NBUF]; // blocks
  struct spinlock headlock[NBUCKS]; // mutex
  struct spinlock freelock[NBUCKS]; // mutex
  struct buf head[NBUCKS]; // Head nodes of busy hash table
  struct buf free[NBUCKS]; // Head nodes of free hash table
} bcache;
```

为了实现高效的代码复用和封装，我们可以将哈希函数以及向双向链表中指定位置插入元素或者删除特定元素的函数单独包装。初始时由于`blockno`域都为零，故可加入`hash(0)`的那个空闲哈希表的桶中。

对于`bget`，它总是计算哈希值获取相应有效链表的锁，以更新引用计数。如果没有找到，这时它还应当轮流地查找空闲哈希表的每个桶。我们总是将最新释放的块插入相应桶的开头。因此，只需查找`free[index]->prev`就可以在 $O(1)$ 时间内完成单桶的 LRU 替换。在轮询的过程中它还需要持有相应空闲桶的链表 (尽管我们已经知道即使不持有这把锁在这个 lab 的测试数据下不会有问题)。

类似地，`brelse`更新引用计数时需要持有有效块所在桶的锁，并移除节点。一旦完成，它就可以释放这把锁。在将它插入空闲桶的时候，同样需要持有对应的锁。

注意到题目中的约束，我们实际上可以放心地合并两个列表，如下图所示:

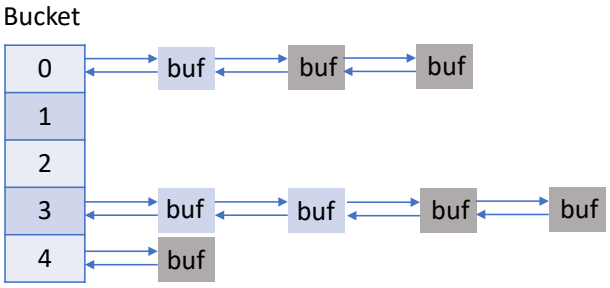

图 2: 空闲块 (引用计数为 0) 和有效块 (引用计数不为 0) 共同使用一个散列表，新近释放的块总是回到当前链表的表首以实现 LRU。图中没有画出最后一个`struct buf`的节点块中的`next`域和头节点中的`prev`域的指针。

当我们发生 miss 的时候，可以同时持有两把锁——一把是待分配的节点所在的桶的锁，另一把则是搜索的桶的锁。注意到若无假定，则可能产生死锁。另一个需要注意的情况则是搜索的桶和待分配的桶是同一个的情况，这时特判一下不要试图再一次获取锁即可。

我们以`bcachetest`的结果说明它们的效率:

旧方案的结果

```
--- top 5 contended locks:
lock: proc: #fetch-and-add 158486 #acquire() 285236
lock: proc: #fetch-and-add 142022 #acquire() 296476
lock: proc: #fetch-and-add 131066 #acquire() 296530
lock: proc: #fetch-and-add 121289 #acquire() 296541
lock: proc: #fetch-and-add 90260 #acquire() 285237
```

新方案的结果

```
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 88838 #acquire() 1197
lock: proc: #fetch-and-add 61778 #acquire() 71892
lock: proc: #fetch-and-add 9607 #acquire() 71552
lock: proc: #fetch-and-add 8615 #acquire() 71546
lock: proc: #fetch-and-add 7590 #acquire() 71525
```

明显观察到，双层加锁的结构将严重地增加文件操作的时间。即使 Buffer Cache 的加锁冲突被完全避免，但是这却极大地增加了`proc`的锁的自旋周期，从而引发系统效率下降，进而超时。有时我们必须在性能和功能之间做某种权衡。当然，这样的新方案不适合一个实用的系统，因为它牺牲了必要的安全性来换取性能。

3 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. *The RISC-V Reader: An Open Architecture Atlas*
   David Patterson, Andrew Waterman *1st Edition*
3. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95