同济大学操作系统课程设计——Lab8: Locks

2151422武芷朵 Tongji University, 2024 Summer

Lab8: Locks：锁的实验

项目地址：[wzd232604/TJOS-xv6-2024-labs: 同济大学操作系统课程设计-xv6实验 (github.com)](https://github.com/wzd232604/TJOS-xv6-2024-labs)

综述

- 在并发编程中我们经常用到锁来解决同步互斥问题，但是一个多核机器上对锁的使用不当会带来很多的所谓 “lock contention” 问题。本次实验的目标就是对涉及到锁的数据结构进行修改，从而降低对锁的竞争。
- 资源重复设置：在 kalloc 中，通过设置多份资源以减少进程的等待概率；

  细化加锁粒度：在 bcache 中，通过精细化的加锁管理，减少资源加锁冲突的概率。

切换到 `lock` 分支：

```
git fetch
git checkout lock
make clean
```

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout lock -f
Branch 'lock' set up to track remote branch 'lock' from 'origin'.
Switched to a new branch 'lock'
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie  user/_stats user/_kalloctest user/_bcachetest \
ph barrier
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

1. Memory allocator (moderate)

1.1 实验目的

通过修改内存分配器的设计，以减少锁竞争，从而提高多核系统中的性能。具体来说，需要实现每个CPU都有一个独立的自由列表（free list），每个列表都有自己的锁。这样可以让不同CPU上的内存分配和释放操作可以并行进行，从而减少锁的争用。还需要实现当一个CPU的自由列表为空时，能够从其他CPU的自由列表中获取部分内存。

1.2 实验步骤

原先只有一个freelist，一个lock给八个cpu竞争使用，需要重新设计为八个freelist、八个lock给八个cpu使用。按照要求，一个 CPU 的空闲链表为空时，需要向别的 CPU“借”一些空间以保证所有空间都能正常分配。

1. 在实验开始之前，运行 `kalloctest` 测试：

```
hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 68099 #acquire() 433016
lock: bcache: #test-and-set 0 #acquire() 1248
--- top 5 contended locks:
lock: kmem: #test-and-set 68099 #acquire() 433016
lock: virtio_disk: #test-and-set 48324 #acquire() 114
lock: proc: #test-and-set 35555 #acquire() 210053
lock: proc: #test-and-set 24013 #acquire() 210053
lock: proc: #test-and-set 16566 #acquire() 209838
tot= 68099
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

用户态程序 kalloctest 产生三个进程，这些进程会不断分配并释放内存，使得原始的 xv6 中的只有一个空闲链表的数据结构的锁被不断获取和释放，且大多数时候对锁的 acquire() 会被阻塞。

test1 的结果：

- 锁 `kmem` 的测试中，有大量的 `#test-and-set` 和 `#acquire()` 操作，这意味着在尝试获取这个锁时，很多次需要重新尝试（test-and-set 是一种获取锁的方式，acquire 是另一种方式）。
- 锁 `proc` 也有大量的 `#test-and-set` 和 `#acquire()` 操作，这也可能是导致问题的原因之一。

从输出来看，问题主要集中在 `kmem` 和 `proc` 这两个锁上，出现了大量的锁竞争和争用情况。因此这是实验中所要解决的问题：减少锁争用，提高内存分配器的性能。

2. 在实验代码中找到内存分配器相关的部分，NCPU 在 `param.h` 中声明，值为 8。

```
C param.h
kernel > C param.h > ...
1   #define NPROC        64  // maximum number of processes
2   #define NCPU          8  // maximum number of CPUs
3   #define NOFILE       16  // open files per process
```

将`kalloc.c` 中`kmem`修改成数组形式结构体，每个CPU分配一个keme锁：

```
kernel > C kalloc.c > freerange(void *, void *)
19    };
20
21    struct {
22      struct spinlock lock;
23      struct run *freelist;
24    } kmem[NCPU];
25
```

3. 修改 `kinit()` 函数： 初始化每个 `CPU` 的空闲链表和锁，使用循环设置每个锁的名称，以及调用 initlock 初始化锁。

```
kernel > C kalloc.c > freerange(void *, void *)
25
26    void
27    kinit()
28    {
29      int i;
30      for (i = 0; i < NCPU; i++) {
31        initlock(&kmem[i].lock, "kmem");
32      }
33    //  initlock(&kmem.lock, "kmem");    // lab8-1
34      freerange(end, (void*)PHYSTOP);
35    }
36
```

4. 修改 `kfree()` 函数： 释放内存页时，根据当前 `CPU` 核心获取对应的锁，确保每个 `CPU` 的空闲链表正确维护。

```
kernel > C kalloc.c > kfree(void *)
50  void
51  kfree(void *pa)
52  {
53    struct run *r;
54    int c;
55
56    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
57      panic("kfree");
58
59    // Fill with junk to catch dangling refs.
60    memset(pa, 1, PGSIZE);
61
62    r = (struct run*)pa;
63    push_off();
64    c = cpuid();
65    pop_off();
66    acquire(&kmem[c].lock);
67    r->next = kmem[c].freelist;
68    kmem[c].freelist = r;
69    release(&kmem[c].lock);
70  }
```

5. 修改 `kalloc()` 函数： 在分配内存页时，先尝试从本地 `CPU` 的空闲链表获取，如果为空则调用 `steal()` 函数偷取其他 CPU 的一半空闲内存页。

```
kernel > C kalloc.c > ...
118  kalloc(void)
122    int c;
123    push_off();
124    c = cpuid();
125    pop_off();
126
127    acquire(&kmem[c].lock);
128    r = kmem[c].freelist;
129    if (r)
130     kmem[c].freelist = r->next;
131     release(&kmem[c].lock);
132
133    if (!r && (r = steal(c))) {
134        acquire(&kmem[c].lock);
135        kmem[c].freelist = r->next;
136        release(&kmem[c].lock);
137      }
138    if(r)
139      memset((char*)r, 5, PGSIZE); // fill with junk
140    return (void*)r;
141  }
```

6. 编写 `steal()` 函数： 该函数用于从其他 `CPU` 偷取部分空闲内存页，保证正确的锁使用和避免死锁。

kernel > C kalloc.c > ...

```c

// steal half page from other cpu's freelist - lab8-1
struct run *steal(int cpu_id) {
    int i;
    int c = cpu_id;
    struct run *fast, *slow, *head;
    // 若传递的cpuid和实际运行的cpuid出现不一致,则引发panic
    // 加入该判断以检查在kalloc()调用steal时CPU不会被切换
    if(cpu_id != cpuid()) {
      panic("steal");
    }
    // 遍历其他NCPU-1个CPU的空闲物理页链表
    for (i = 1; i < NCPU; ++i) {
        if (++c == NCPU) {
            c = 0;
        }
        acquire(&kmem[c].lock);
        // 若链表不为空
        if (kmem[c].freelist) {
            // 快慢双指针算法将链表一分为二
            slow = head = kmem[c].freelist;
            fast = slow->next;
            while (fast) {
                fast = fast->next;
                if (fast) {
                    slow = slow->next;
                    fast = fast->next;
                }
            }
            // 后半部分作为当前CPU的空闲链表
            kmem[c].freelist = slow->next;
            release(&kmem[c].lock);
            // 前半部分的链表结尾清空,由于该部分链表与其他链表不再关联,因此无需加锁
            slow->next = 0;
            // 返回前半部分的链表头
            return head;
        }
        release(&kmem[c].lock);
    }
    // 若其他CPU物理页均为空则返回空指针
    return 0;
}
```

7. 利用make qemu指令运行xv6：

8. 在命令行中输入 `kalloctest`：

```
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 92711
lock: kmem: #test-and-set 0 #acquire() 162391
lock: kmem: #test-and-set 0 #acquire() 177918
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: bcache: #test-and-set 0 #acquire() 340
--- top 5 contended locks:
lock: proc: #test-and-set 133611 #acquire() 777301
lock: proc: #test-and-set 124582 #acquire() 777302
lock: proc: #test-and-set 101859 #acquire() 777303
lock: proc: #test-and-set 86892 #acquire() 777302
lock: proc: #test-and-set 80690 #acquire() 777462
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

在 test1 中，可以看到 `acquire` 的循环迭代次数明显减少，表明锁的争用较之前有所减少，每个 CPU 现在有了自己的 `freelist`，减少了 CPU 之间在访问内存分配器时的竞争。

9. 运行 `usertests sbrkmuch` 测试，确保内存分配器仍然能够正确分配所有内存。

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

10. 运行 `usertests` 测试，确保所有的用户测试都通过。

```
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

1.3 实验中遇到的问题和解决办法

1. 问题：原始的内存分配器在多核系统上存在锁的争用问题，导致性能下降。

- 解决办法： 分离锁： 通过为每个 `CPU` 核心创建独立的空闲内存页链表和锁，避免了多核之间的锁争用。这样，每个 `CPU` 核心可以独立地进行内存分配和释放操作，提高了性能和效率。

2. 问题：读取CPU的ID时没有禁用中断，使得程序出错。

- 解决办法：在实现过程中，需要确保每次获取锁和释放锁的顺序是一致的，避免出现循环等待的情况。除此之外，这很有可能是多个核心同时访问共享资源而没有正确的同步机制，也许会导致竞争、数据损坏、不一致性或不可预测的结果。

1.4 实验心得

- 通过完成这个实验，我逐渐理解了操作系统内核的基本结构以及如何解决多核心环境下的锁竞争问题。在多核心环境下，锁的使用是解决并发访问问题的关键。
- 通过优化锁的使用，不仅提高了内存分配的性能，还加深了我对操作系统中并发控制的理解。

2. Buffer cache (hard)

2.1 实验目的

优化xv6操作系统中的缓冲区缓存（buffer cache），减少多个进程之间对缓冲区缓存锁的争用，从而提高系统的性能和并发能力。通过设计和实现一种更加高效的缓冲区管理机制，使得不同进程可以更有效地使用和管理缓冲区，减少锁竞争和性能瓶颈。

2.2 实验步骤

原始的 xv6 中，对于缓存的读写是由单一的锁 `bcache.lock` 来保护的，导致了如果系统中有多个进程在进行IO 操作，则等待获取 `bcache.lock` 的开销就会较大。为了减少加锁的开销，可以将缓存分为几个桶，为每个桶单独设置一个锁，这样如果两个进程访问的缓存块在不同的桶中，则可以同时获得两个锁进行操作，而无需等待加锁。目标是将`bcachetest` 中统计值tot 降到规定值以下。

1. 运行 `bcachetest` 测试程序，观察锁竞争情况和测试结果

```
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 921720
lock: kmem: #test-and-set 0 #acquire() 2094420
lock: kmem: #test-and-set 0 #acquire() 2845943
lock: kmem: #test-and-set 0 #acquire() 1402
lock: kmem: #test-and-set 0 #acquire() 1402
lock: kmem: #test-and-set 0 #acquire() 1402
lock: kmem: #test-and-set 0 #acquire() 1402
lock: kmem: #test-and-set 0 #acquire() 1402
lock: bcache: #test-and-set 50672 #acquire() 1762390
--- top 5 contended locks:
lock: proc: #test-and-set 63801258 #acquire() 110114852
lock: proc: #test-and-set 61820618 #acquire() 110970061
lock: proc: #test-and-set 59670381 #acquire() 110962311
lock: virtio_disk: #test-and-set 26216350 #acquire() 111580
lock: log: #test-and-set 16413592 #acquire() 74210
tot= 50672
0124
test0: FAIL
start test1
test1 OK
$
```

   测试结果表明，在多个进程之间，对bcache.lock锁的竞争比较激烈，导致多个进程在试图获取该锁时需要进行较多次的test-and-set操作和acquire()操作。这说明了缓冲区管理中存在较大的竞争问题，可能影响了系统的性能和响应速度。

2. 重新设计和实现缓冲区管理机制，修改`buf`和 `bcache` 结构体，为每个缓冲区添加时间戳字段，为 `bcache` 添加哈希表的锁。

bio.c 6, M ● buf.h M ×

kernel > buf.h > ...

```c
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *next;      // hash list
  uchar data[BSIZE];
  uint timestamp;   // the buf last using time - lab8-2
};
```

定义`NBUCKET`表示管理的bucket个数，并且定义hash映射（磁盘块号blockno到bucket号）

`bcache`结构体中加入`buckets`（buffer cache的管理单位，将一堆buf数组组织在一起成为bucket）以及 locks 作为每个bucket的互斥锁

kernel > bio.c > binit(void)

```c

#define NBUCKET 13      // the count of hash table's buckets
#define HASH(blockno) (blockno % NBUCKET)

struct {
  struct spinlock lock;   // used for the buf alloc and size
  struct buf buf[NBUF];
  int size;     // record the count of used buf - lab8-2
  struct buf buckets[NBUCKET];  // lab8-2
  struct spinlock locks[NBUCKET];   // buckets' locks - lab8-2
  struct spinlock hashlock;     // the hash table's lock - lab8-2
  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
//  struct buf head;    // lab8-2
} bcache;

```

- `struct spinlock lock`是自旋锁，用于保护对该分桶的并发访问。当有线程要操作这个分桶时，需要先获取这个锁。
- `struct buf buf[NBUF]`数组的每个元素代表一个缓冲区。缓冲区结构体`struct buf`存储了实际的数据内容、元数据和与缓冲区相关的控制信息。

3. 修改初始化函数： 在`binit()`函数中初始化哈希表的锁和缓冲区的时间戳字段。

kernel > C bio.c > bget(uint, uint)

```
44  void
45  binit(void)
46  {
47    int i;
48    struct buf *b;
49
50    bcache.size = 0;  // lab8-2
51    initlock(&bcache.lock, "bcache");
52    initlock(&bcache.hashlock, "bcache_hash");    // init hash lock - lab8-2
53    // init all buckets' locks  - lab8-2
54    for(i = 0; i < NBUCKET; ++i) {
55      initlock(&bcache.locks[i], "bcache_bucket");
56    }
57
58    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
59
60      initsleeplock(&b->lock, "buffer");
61
62    }
63  }
64
```

4. 修改 `brelse()` 函数： 释放缓冲区时，更新时间戳字段，根据时间戳来选择缓冲区进行重用。

kernel > C bio.c > bget(uint, uint)

```
188   brelse(struct buf *b)
190     int idx;
191     if(!holdingsleep(&b->lock))
192       panic("brelse");
193
194     releasesleep(&b->lock);
195
196     // change the lock - lab8-2
197     idx = HASH(b->blockno);
198     acquire(&bcache.locks[idx]);
199     b->refcnt--;
200     if (b->refcnt == 0) {
201       b->timestamp = ticks;
202     }
203
204     release(&bcache.locks[idx]);
205   }
```

5. 修改分配函数： 在 `bget()` 函数中，使用哈希表来寻找对应的缓冲区，若未找到则根据时间戳和引用计数来选择缓冲区进行覆盖重用。

kernel > C bio.c > bget(uint, uint)

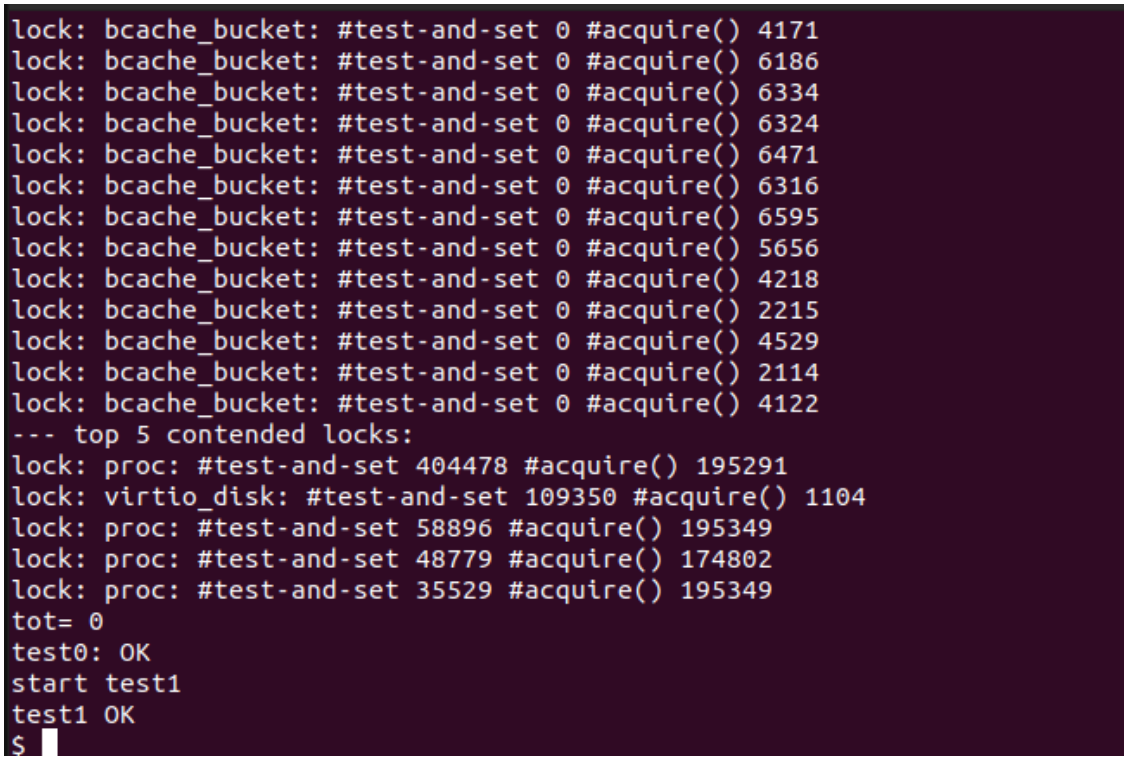
```
68  // In either case, return locked buffer.
69  static struct buf*
70  bget(uint dev, uint blockno)
71  {
72    struct buf *b;
73    // lab8-2
74    int idx = HASH(blockno);
75    struct buf *pre, *minb = 0, *minpre;
76    uint mintimestamp;
77    int i;
78
79    // loop up the buf in the buckets[idx]
80    acquire(&bcache.locks[idx]);  // lab8-2
81    for(b = bcache.buckets[idx].next; b; b = b->next){
82      if(b->dev == dev && b->blockno == blockno){
83        b->refcnt++;
84        release(&bcache.locks[idx]);  // lab8-2
85        acquiresleep(&b->lock);
86        return b;
87      }
88    }
89
90    // Not cached.
91    // check if there is a buf not used -lab8-2
92    acquire(&bcache.lock);
93    if(bcache.size < NBUF) {
94      b = &bcache.buf[bcache.size++];
95      release(&bcache.lock);
96      b->dev = dev;
97      b->blockno = blockno;
98      b->valid = 0;
99      b->refcnt = 1;
100     b->next = bcache.buckets[idx].next;
101     bcache.buckets[idx].next = b;
102     release(&bcache.locks[idx]);
103     acquiresleep(&b->lock);
104     return b;
105   }
106   release(&bcache.lock);
107   release(&bcache.locks[idx]);
108
```

```
kernel > C bio.c > bget(uint, uint)
70  bget(uint dev, uint blockno)
111   acquire(&bcache.hashlock);
112   for(i = 0; i < NBUCKET; ++i) {
113       mintimestamp = -1;
114       acquire(&bcache.locks[idx]);
115       for(pre = &bcache.buckets[idx], b = pre->next; b; pre = b, b = b->next) {
116           // research the block
117           if(idx == HASH(blockno) && b->dev == dev && b->blockno == blockno){
118               b->refcnt++;
119               release(&bcache.locks[idx]);
120               release(&bcache.hashlock);
121               acquiresleep(&b->lock);
122               return b;
123           }
124           if(b->refcnt == 0 && b->timestamp < mintimestamp) {
125               minb = b;
126               minpre = pre;
127               mintimestamp = b->timestamp;
128           }
129       }
130       // find an unused block
131       if(minb) {
132           minb->dev = dev;
133           minb->blockno = blockno;
134           minb->valid = 0;
135           minb->refcnt = 1;
136           // if block in another bucket, we should move it to correct bucket
137           if(idx != HASH(blockno)) {
138               minpre->next = minb->next;    // remove block
139               release(&bcache.locks[idx]);
140               idx = HASH(blockno);  // the correct bucket index
141               acquire(&bcache.locks[idx]);
142               minb->next = bcache.buckets[idx].next;    // move block to correct bucket
143               bcache.buckets[idx].next = minb;
144           }
145           release(&bcache.locks[idx]);
146           release(&bcache.hashlock);
147           acquiresleep(&minb->lock);
148           return minb;
149       }
150       release(&bcache.locks[idx]);
151       if(++idx == NBUCKET) {
152           idx = 0;
153       }
154   }
155
156   panic("bget: no buffers");
157 }
```

6. 加入两个维护block的refcnt的函数：

```
kernel > C bio.c > bpin(buf *)
207   void
208   bpin(struct buf *b) {
209     int idx = HASH(b->blockno);
210     acquire(&bcache.locks[idx]);
211     b->refcnt++;
212     release(&bcache.locks[idx]);
213   }
214
215   void
216   bunpin(struct buf *b) {
217     int idx = HASH(b->blockno);
218     acquire(&bcache.locks[idx]);
219     b->refcnt--;
220     release(&bcache.locks[idx]);
221   }
222
```

7. `make qemu`，`bcachetest`



比较实验开始前和实验进行后的测试结果，可以看出锁竞争明显减少，在实验进行之前的测试中，锁竞争非常严重。`bcachetest` 测试中显示了大量的 `test-and-set` 操作和锁的获取次数，这表明在并发访问缓冲区池时存在大量的竞争。而最新测试结果中，锁竞争明显减少，`test-and-set` 操作和锁的获取次数较少。

2.3 实验中遇到的问题和解决办法

1. 问题：缓冲区管理效率低下： 原始的缓冲区管理方式使用链表结构，重用缓冲区时效率低下。

- 解决办法： 使用哈希表和时间戳： 引入线程安全的哈希表来管理缓冲区，使用时间戳字段来实现基于时间的 `LRU` 算法。这样可以更高效地管理缓冲区，提高文件 `I/O` 操作的性能。

2. 问题： 在修改缓冲区管理代码时，可能会遇到并发情况下的数据一致性问题，如资源分配冲突、竞争条件。

- 解决办法： 增加了锁，根据缓冲区的块号计算哈希索引 `v`，获取对应的分桶结构体指针 `bucket` 和分桶的自旋锁，以确保在操作缓冲区引用计数时不会被其他线程干扰。

2.4 实验心得

- 通过优化缓冲区管理方式，我学会了如何在实际系统中使用更高效的数据结构和算法来提升性能。
- 在Buffer Cache中，LRU算法用于确定哪个缓存块应该被释放，以便为新的数据块腾出空间。LRU算法的核心思想是，最近最少使用的缓存块应该被优先释放，因为它们更有可能在未来不会被再次访问。而在结构体的设计上，`struct bucket` 结构体包含一个锁 `lock` 和一个链表头 `head`。链表头 `head` 表示了一个双向循环链表，用于存储缓存块。这个链表是按照缓存块被使用的时间顺序来排序的，即最近被使用的缓存块位于链表头部，最少使用的位于链表尾部。

3 实验检验得分

1. 在实验目录下创建`time.txt`，填写完成实验时间数

2. 在终端中执行`make grade`

   (性能原因导致超时)

```
== Test running kalloctest ==
$ make qemu-gdb
(596.1s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (56.2s)
== Test running bcachetest ==
$ make qemu-gdb
(63.3s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (600.8s)
    ...
         OK
         test opentest: OK
         test writetest: OK
```

```
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (600.8s)
    ...
         OK
         test opentest: OK
         test writetest: OK
         test writebig: panic: balloc: out of blocks
         qemu-system-riscv64: terminating on signal 15 from pid 23128 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 51/70
make: *** [Makefile:336: grade] Error 1
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```