OS Lab 5

Locks

王世炟

PB20151796

December 3, 2022

Part 1: 实验要求

在本实验中，我们要学会设计锁，主要目的是降低多线程情况下对锁的竞争，即降低锁的粒度，将一个大锁更换为一些粒度小的锁，这样可以大幅度降低锁的竞争。

Part 2: Memory allocator

2.1 问题简述

第一个任务是解决内存块的竞争问题。

2.2 知识基础

在 kalloctest 中，三个进程不断通过调用 kalloc 和 kfree 来增长或释放它们的内存空间，然而在 kalloc 和 kfree 中只有一个锁 kmem.lock 可以保证这三个进程的访问不冲突，这样一来，导致了大量的无效锁请求，这就导致了 lock contention。

出现 lock contention 的根本原因是 kalloc() 中只有一个 free list，该 free list 只被一个 lock 保护。为了消除 lock contention，我们必须想办法避免单 free list、单 lock 问题。最基本的思想就是为每一个 CPU 都维护一个 free list，并且每一个 free list 都有一个独自的 lock。这样一来，不同的 CPU 就能并行地利用 kalloc() 与 kfree() 分配或释放内存了，因为每个 CPU 都只会操作不同的 free list。在实现这个方案的过程中，最具挑战的就是: 当一个 CPU 的 free list 为空，但是另一个 CPU 的 free list 还有空闲块时，我们就应该从有空闲 free list 的 CPU 处“偷”一个空闲内存块。

2.3 实验步骤

(1) 更改内存块结构

首先，为每一个 CPU 维护一个 freelist 和自旋锁 (利用 kernel/param.h 中的 NCPU):

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

(2) **更改** kinit() **函数**

在初始化时，为每一个 CPU 初始化一个自旋锁：

```
void
kinit()
{
  for (int i = 0; i < NCPU; i++)
  {
    initlock(&kmem[i].lock, "kmem");
  }

  freerange(end, (void*)PHYSTOP);
}
```

(3) **更改** kfree() **函数**

只需获取 CPU 的 ID，并对于该 CPU 的 freelist 进行更改即可 (注意根据提示，使用 cpuid() 时要关闭中断：

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP
     )
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  push_off();
  int cpu_id = cpuid();
  pop_off();
```

```
17
18
19    acquire(&kmem[cpu_id].lock);
20    r->next = kmem[cpu_id].freelist;
21    kmem[cpu_id].freelist = r;
22    release(&kmem[cpu_id].lock);
23
24  }
```

(4) **更改** kalloc() **函数**

如果当前 CPU 有空闲块，就直接返回空闲块，freelist 相应改变；若没有，则查找取其他 CPU 的 freelist，如果有空闲块，则“偷”过来。

```
1   void *
2   kalloc(void)
3   {
4     struct run *r;
5
6     push_off();
7     int cpu_id = cpuid();
8     pop_off();
9
10    acquire(&kmem[cpu_id].lock);
11    r = kmem[cpu_id].freelist;
12    if(r)
13      kmem[cpu_id].freelist = r->next;
14    else {
15      for (int i = 0; i < NCPU; i++)
16      {
17        if (i == cpu_id)
18          continue;
19        acquire(&kmem[i].lock);
20        r = kmem[i].freelist;
21        if(r)
22          kmem[i].freelist = r->next;
23        release(&kmem[i].lock);
24        if(r)
25          break;
26      }
27    }
28    release(&kmem[cpu_id].lock);
29
```

```
30    if(r)
31      memset((char*)r, 5, PGSIZE); // fill with junk
32    return (void*)r;
33  }
```

(5) **测试结果**

上述操作完成后可以得到以下结果：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 165751
lock: kmem: #fetch-and-add 0 #acquire() 118237
lock: kmem: #fetch-and-add 0 #acquire() 149080
lock: bcache: #fetch-and-add 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #fetch-and-add 1125370 #acquire() 189564
lock: virtio_disk: #fetch-and-add 761482 #acquire() 114
lock: proc: #fetch-and-add 686111 #acquire() 189563
lock: proc: #fetch-and-add 656922 #acquire() 189564
lock: proc: #fetch-and-add 590382 #acquire() 189565
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

图 1: Memory allocator passed.

```
riscv64-linux-gnu-objdump -t kernel/kernel
make[1]: 离开目录“/home/ubuntu/桌面/xv6-la
== Test running kalloctest ==
$ make qemu-gdb
(362.7s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (37.7s)
== Test running bcachetest ==
```

图 2: usertests passed.

Part 3: Buffer cache

3.1 问题简述

Buffer Cache 出现的问题和 Memory Allocator 中的问题其实类似，只不过这里是管理磁盘缓存。试想这样一个情况：三个进程大量读写磁盘，而磁盘缓存只有一个 lock，这就导致三个进程的竞争异常激烈。本任务就是要解决这个问题。

3.2 问题分析

xv6 文件系统的 buffer cache 采用了一个全局的锁 bcache.lock 来负责对 buffer cache 进行读写保护，当 xv6 执行读写文件强度较大的任务时会产生较大的锁竞争压力，因此需要一个哈希表，将 buf entry 以 buf.blockno 为键哈希映射到这个哈希表的不同的 BUCKET 中，给每个 BUCKET 一个锁，NBUCKET 最好选择素数。注意：这个实验

不能像上一个一样给每个 CPU 一个 bcache，因为文件系统在多个 CPU 之间是真正实现共享的，否则将会造成一个 CPU 只能访问某些文件的问题。

3.3 实验步骤

(1) 修改结构

修改 bache 的结构，将 buf 分为 13 份（hint 提示我们选择 13），使其成成为哈希表：

```
#define NBUCKTE 13

struct {
  struct spinlock lock[NBUCKTE];
  struct buf buf[NBUF];
  struct buf hashbucket[NBUCKTE];
  // 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;
} bcache;
```

(2) 获取哈希值

```
int
hash(int n)
{
  return n % NBUCKTE;
}
```

(3) 改写 binit() 函数

对于 binit() 函数，同第一部分，先为所有的桶初始化锁，再将所有的 buffer 块放入一个桶里面，建立双向循环链表。

```
void
binit(void)
{
  struct buf *b;
  for (int i = 0; i < NBUCKTE; i++)
  {
    initlock(&bcache.lock[i], "bcache_hash"); // 初始化锁
    // 将哈希表的头结点指向自己
```

```
    bcache.hashbucket[i].prev = &bcache.hashbucket[i];
    bcache.hashbucket[i].next = &bcache.hashbucket[i];
  }

  // Create linked list of buffers
  // bcache.head.prev = &bcache.head;
  // bcache.head.next = &bcache.head;

  // 头插法建立双向链表，将所有buffer块加入到第一个桶中
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.hashbucket[0].next;
    b->prev = &bcache.hashbucket[0];
    initsleeplock(&b->lock, "buffer");
    bcache.hashbucket[0].next->prev = b;
    bcache.hashbucket[0].next = b;
  }
}
```

(4) **改写** bget() **函数**

在 bget() 函数中，如果缓存中有块，则直接获得该块。如果缓存中没有块，但是对应的哈希桶存在空闲的最久未使用的缓存块，则直接设置该缓存块。否则从其他的哈希桶中找出空闲的缓存块。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  int id = hash(blockno);
  acquire(&bcache.lock[id]);

  // Is the block already cached?
  for(b = bcache.hashbucket[id].next; b != &bcache.hashbucket[id]; b = b->
      next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      // 释放对应桶的锁
      release(&bcache.lock[id]);
      acquiresleep(&b->lock);
      return b;
    }
  }

```

```
19    // Not cached.
20    // Recycle the least recently used (LRU) unused buffer.
21    for(b = bcache.hashbucket[id].prev; b != &bcache.hashbucket[id]; b = b->
          prev){
22      if(b->refcnt == 0) {
23        b->dev = dev;
24        b->blockno = blockno;
25        b->valid = 0;
26        b->refcnt = 1;
27        release(&bcache.lock[id]);
28        acquiresleep(&b->lock);
29        return b;
30      }
31    }
32
33    // 当前桶中没有空闲块
34    release(&bcache.lock[id]);
35    for (int i = 0; i < NBUCKTE; i++)
36    {
37      if (i == id)
38      {
39        continue;
40      }
41      acquire(&bcache.lock[i]);
42
43      for (b = bcache.hashbucket[i].prev; b != &bcache.hashbucket[i]; b=b->
            prev)
44      {
45        acquire(&bcache.lock[id]);
46        if (b->refcnt == 0)
47        {
48          b->dev = dev;
49          b->blockno = blockno;
50          b->valid = 0;
51          b->refcnt = 1;
52
53          // 将b从当前桶中取出来
54          b->next->prev = b->prev;
55          b->prev->next = b->next;
56
57
58          // 将b插到等待buffer块的桶的开头
59          b->prev = &bcache.hashbucket[id];
```

```
60          b->next = bcache.hashbucket[id].next;
61          bcache.hashbucket[id].next->prev = b;
62          bcache.hashbucket[id].next = b;
63
64
65          release(&bcache.lock[id]);
66          release(&bcache.lock[i]);
67          acquiresleep(&b->lock);
68          return b;
69        }
70        release(&bcache.lock[id]);
71      }
72      release(&bcache.lock[i]);
73
74    }
75    panic("bget: no buffers");
76  }
```

(5) **改写** brelse() **函数**

释放某个块时，只要获取对应哈希桶的锁即可。

```
1  void
2  brelse(struct buf *b)
3  {
4    if(!holdingsleep(&b->lock))
5      panic("brelse");
6
7    releasesleep(&b->lock);
8    int id = hash(b->blockno);
9    acquire(&bcache.lock[id]);
10   b->refcnt--;
11   if (b->refcnt == 0) {
12     // no one is waiting for it.
13     b->next->prev = b->prev;
14     b->prev->next = b->next;
15     b->next = bcache.hashbucket[id].next;
16     b->prev = &bcache.hashbucket[id];
17     bcache.hashbucket[id].next->prev = b;
18     bcache.hashbucket[id].next = b;
19   }
20
21   release(&bcache.lock[id]);
```

```
22 }
```

(6) **改写** bpin(),bunpin() **函数**

同 brelse()，只要获取对应哈希桶的锁即可。

```
1  void
2  bpin(struct buf *b) {
3    int id = hash(b->blockno);
4    acquire(&bcache.lock[id]);
5    b->refcnt++;
6    release(&bcache.lock[id]);
7  }
8
9  void
10 bunpin(struct buf *b) {
11   int id = hash(b->blockno);
12   acquire(&bcache.lock[id]);
13   b->refcnt--;
14   release(&bcache.lock[id]);
15 }
```

可通过所有测试:

```
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/S
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
== Test running kalloctest ==
$ make qemu-gdb
(345.9s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (38.5s)
== Test running bcachetest ==
$ make qemu-gdb
(58.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (503.7s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 70/70
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$
```

图 3: alltests passed

Part 4: 实验结果

运行 make grade，检验结果：

```
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/S
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
== Test running kalloctest ==
$ make qemu-gdb
(345.9s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (38.5s)
== Test running bcachetest ==
$ make qemu-gdb
(58.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (503.7s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 70/70
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$
```

图 4: Result

本次实验完成了 Locks 有关功能的实现和实际应用。在实验过程中得到了以下收获：

- 对于自旋锁和睡眠锁有了更深刻的理解
- 复习了死锁的解锁方法。
- 认识到了操作系统锁设计合理性以及正确性