同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 8

2151767 薛树建

日期：2023 年 8 月 17 日

目录

Lab8：locks

1. Memory allocator (moderate)

1) 实验目的

本实验提示此前已经实现的内存分配器由于采用一个 freelist 来管理空闲内存的，但是该空闲内存只有一个锁，在单个 cpu 运行下还可以，但是多个 cpu 并行执行时，就会出现很严重的排队现象，cpu 在尝试 acquire 获取锁的循环迭代次数将会很大。

实验网站给了我们一个解决方法，即每一块 cpu 都有一个自己的列表与一把锁，这样便可以解决上面的问题，需要注意的是要处理当一个 cpu 列表无空闲块时，需要去窃取其他 cpu 的空闲块。

2) 实验步骤

a) 首先修改 kmem，将其设置为一个与 cpu 个数一样的数组

```
struct {
  struct spinlock lock;
  struct run *freelist;
  char  name[16];
} kmem[NCPU];
```

b) 在 kinit()函数中初始化对应的锁，这里按照实验要求为每一个锁初始化了不同的名字(实际上也可以不添加)，并调用 freerange 函数，释放内存

```
void
kinit()
{
  for (int i = 0; i < NCPU; ++i) {
    snprintf(kmem[i].name, 16, "%s%d", "kmem", i);//为每
    initlock(&kmem[i].lock, kmem[i].name);
  }

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

c) 我们接下来去修改 kfree 函数，将要释放的物理内存快地址采用头插法插入到 freelist 的头部，同时根据实验提示，cpuid()函数只有在关闭中断时其结果才是可靠的。调用 push_off 与 pop_off 来开关中断。同时也要注意获取对应的锁，避免多个线程同时访问出错。

```
r = (struct run*)pa;

push_off();
int cpu=cpuid();
pop_off();

acquire(&kmem[cpu].lock);
r->next = kmem[cpu].freelist;
kmem[cpu].freelist = r;
release(&kmem[cpu].lock);
```

d) 最后我们需要去实现 kalloc 函数，注意要添加从其他 cpu 中窃取空闲块的操作。先获取 cpuid，接着判断是否有空闲块，

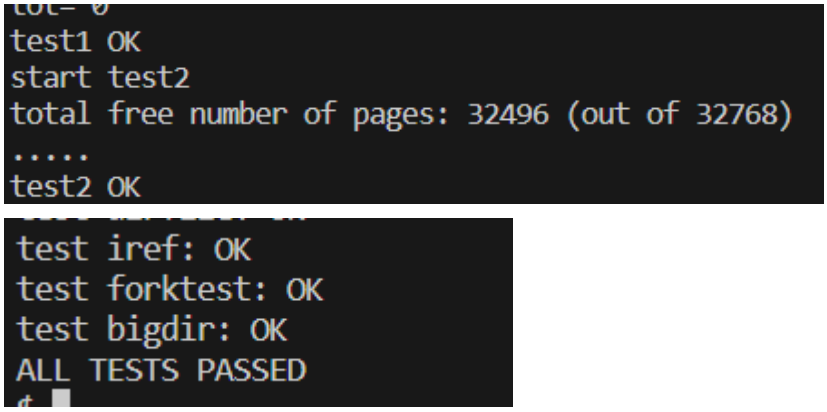
```
push_off();
int cpu=cpuid();
pop_off();


acquire(&kmem[cpu].lock);
r = kmem[cpu].freelist;
if(r)
  kmem[cpu].freelist = r->next;
```

如果有空闲块，则就去除并修改 freelist，否则直接就遍历 cpu 寻找具体空闲块的 cpu.

```
else{//此时说明freelist已经没有空闲，需要偷取其他cpu的空闲列表
  for(int i=0;i<NCPU;i++){
    if(i==cpu)//此时为自身
      continue;


    acquire(&kmem[i].lock);

    if(!kmem[i].freelist) {//其他cpu空闲表也为空
      release(&kmem[i].lock);
      continue;
    }
    r = kmem[i].freelist;
    kmem[i].freelist = r->next;
    release(&kmem[i].lock);
    break;
  }

}
release(&kmem[cpu].lock);
```

在遍历过程中需要注意获取锁来保证正确性，若遍历到的 cpu 号是自身则直接忽略；若不是则获取对应锁并判断是否有空闲块，如果有，则去除并作对应修改，若果没有则继续遍历。

e) 测试结果如下：



3) 实验中遇到的问题和解决方法

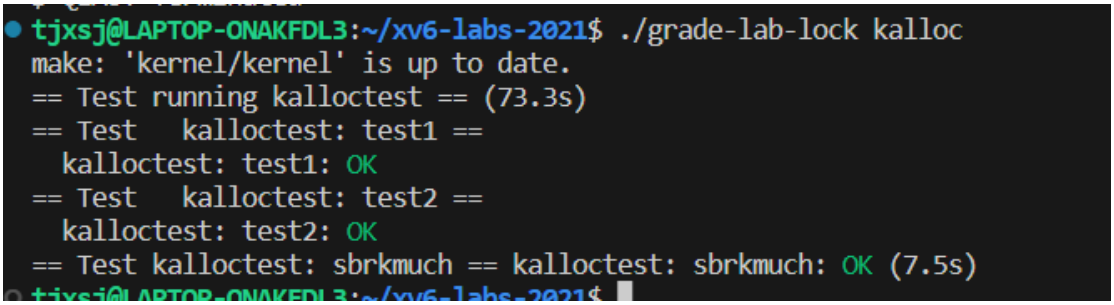
本实验内容主要涉及对内存分配器的一个优化，相应的结构体信息与作用等都可以通过阅读实验指导书来了解。实际实验时并没有遇到太大的问题。

4) 实验心得

在完成本实验后，我对于 xv6 操作系统的内存分配器有了一定的了解，同时本实验的思路通过设置多个 cpu 来达到快速分配内存，减小每个进程的等待时间，一定程度上可以看作时通过重复设置锁的资源，来增加每个进程分配内存时获取到锁而无需等待的概率。

在本次实验中，我加深了对内存管理和操作系统的理解，也提升了我的编程能力和实践经验。

本实验评分如下：



2. Buffer cache (hard)

1) 实验目的

缓存块可以提高整个系统的性能。在本实验中，xv6 原本的缓存块是通过一个单一的 lock 来保护的，这就会导致如果多个进程同时访问会导致等待获取 lock 的开销增大。

我们本次实验需要通过某种方式来减少这种开销，要求 acquire 获取锁的循环迭代次数接近于 0（实际上小于 500 就行）。本实验给我们提供了一个思路，我们可以将缓存划分为若干个桶（例如 13），为每个桶单独设置一个锁，内存块通过哈希运算找到自己所对应的桶，每一个桶都储存着自己列表的 head 结点。如果某物理块还没有映射到 buf 中，则再其对应的桶中找一个最近没用过的进行替换，否则则需要向其他桶中去借取。

2) 实验步骤

a) 我们首先需要去修改 bcache 中的成员，在其中加入桶对应的锁与 buf 的数组。

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // 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 buckets[NBUCKET];
  struct spinlock locks[NBUCKET];
} bcache;
```

b) 修改 binit 函数，完成对 bcache 的初始化。

先初始化整体的锁，接着遍历桶，初始化各个桶的锁与 buf 数组，buf 数组中储存的便是各个桶各自的 buf 列表的 head 元素，以此来达到划分的目的。

```
initlock(&bcache.lock,"bcache");
// Create linked list of buffers
//bcache.head.prev = &bcache.head;
//bcache.head.next = &bcache.head;
for(int i=0; i<NBUCKET; i++) {//每个哈希桶的元素
                              //同时每一个哈希桶
  initlock(&bcache.locks[i], "locks");

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

最后初始化整体的 buf 数组中的元素。

```
for(b = bcache.buf; b < bcache.buf+NBUF; b++){
  initsleeplock(&b->lock, "buflock");
  b->lasttick=0;
  b->dev=-1;
  b->blockno=-1;
  b->refcnt=0;
  b->next = bcache.buckets[0].next;
  b->prev = &bcache.buckets[0];
  bcache.buckets[0].next->prev = b;
  bcache.buckets[0].next = b;
}
```

c) 接下来是 bget 函数的修改，bget 函数接受两个参数，分别是设备号与块号，以此来返回该块对应的 buf。如果不能找到，则需要为该块分配新的 buf。
在此之前，由于会出现替换块的情况。我们需要在 buf 结构体中添加一个变量 lasttick 来记录上一次该 buf 引用的 ticks，如下：

```
struct buf *next;
uchar data[BSIZE];
uint lasttick;
};
```

我们先寻找该块是否已经映射到了 buf 上，若是直接返回。大致思路便是先哈希运算得到该块对应的桶，遍历桶对应的所有 buf 来寻找。找到之后，增加其引用计数，并返回一个获取了睡眠锁的 buf 块。

```
struct buf *b;
int index = blockno % NBUCKET;
acquire(&bcache.locks[index]);

// Is the block already cached?
for(b = bcache.buckets[index].next; b != &bcache.buckets[index]; b = b->next){
  if(b->dev == dev && b->blockno == blockno){
    b->refcnt++;
    release(&bcache.locks[index]);
    acquiresleep(&b->lock);
    return b;
  }
}
release(&bcache.locks[index]);
```

接着我们需要去从其他的桶中去借一个 buf 块。因而此时我们需要先获取全局的锁（bcache 的锁），主要是为了防止多个进程同时操作桶而导致错误。
随后我们再运行一次和上面一样的操作。原因是可能本进程在等全局锁的，可能有一个相同的进程已经在该代码区域为该块申请了新的 buf，这样就会出现一个物理块对应两个 buf 的错误情况。

```
struct buf *tarbuf=0;
uint minticks=ticks;
acquire(&bcache.lock);
acquire(&bcache.locks[index]);
for(b = bcache.buckets[index].next; b != &bcache.buckets[index]; b = b->next){
      if(b->dev == dev && b->blockno == blockno){
          // found in hash table
          // increase ref count
          b->refcnt++;
          release(&bcache.locks[index]);
          release(&bcache.lock);
          // acquire sleep lock
          acquiresleep(&b->lock);
          return b;
      }
  }
```

接着去从本桶中利用 LRU 替换策略选一个最近没有用的块。

```
 for(b = bcache.buckets[index].next; b != &bcache.buckets[index]; b = b->next){
   if(b->refcnt==0&&b->lasttick<minticks){//LRU
     minticks=b->lasttick;
     tarbuf=b;
   }
 }
```

能找到的话，就直接替换并返回相应的 buf 块指针。

```
//此时tarbuf不为空，不需要借
//初始化
tarbuf->dev = dev;
tarbuf->blockno = blockno;
tarbuf->valid = 0;
tarbuf->refcnt = 1;
release(&bcache.locks[index]);
release(&bcache.lock);
acquiresleep(&tarbuf->lock);
return tarbuf;
```

如果不存在的话，则需要向其他桶去借，本质上还是将其 buf 块插入到自己桶的列表中。思路基本一致，采用 LRU 算法寻找一个 buf 块进行替换。

```
for(int i=0; i<NBUCKET; i++) {
  if(i == index)
    continue;

  acquire(&bcache.locks[i]);
  minticks = ticks;
  for(b = bcache.buckets[i].next; b != &bcache.buckets[i]; b = b->next){
    if(b->refcnt==0 && b->lasttick<=minticks) {
      minticks = b->lasttick;
      tarbuf = b;
    }
  }
  if(tarbuf){
```

```
if(tarbuf){
  tarbuf->dev = dev;
  tarbuf->blockno = blockno;
  tarbuf->valid = 0;
  tarbuf->refcnt = 1;

  //将其从i号桶中取出，双向列表的删除操作
  tarbuf->next->prev = tarbuf->prev;
  tarbuf->prev->next = tarbuf->next;


  //将其放入index对应的桶,双向列表的插入操作
  tarbuf->next = bcache.buckets[index].next;
  bcache.buckets[index].next->prev = tarbuf;
  bcache.buckets[index].next = tarbuf;
  tarbuf->prev = &bcache.buckets[index];
  release(&bcache.locks[i]);
  release(&bcache.locks[index]);//释放锁
  release(&bcache.lock);
  acquiresleep(&tarbuf->lock);
  return tarbuf;
}
release(&bcache.locks[i]);//释放该锁，继续循环
```

如果再其他块中也找不到，则释放对应的锁之后，运行 panic 函数。

```
  release(&bcache.locks[i]);//释放该锁，继续
}
release(&bcache.locks[index]);//释放该锁
release(&bcache.lock);
panic("bget: no buffers");
```

d) 最后是修改一个 brelse 函数，没有释放时减少对应的引用数，当引用数为 0 是修改 lasttick

```
  acquire(&bcache.locks[index]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    //b->next->prev = b->prev;
    //b->prev->next = b->next;
    //b->next = bcache.buckets[index].nex
    //b->prev = &bcache.buckets[index];
    //bcache.buckets[index].next->prev =
    //bcache.buckets[index].next = b;
    b->lasttick=ticks;
  }

  release(&bcache.locks[index]);
}
```

e) 最后修改以下末尾的函数，将对应的全局锁更换为对应块的局部锁即可

```
void
bpin(struct buf *b) {
  int index=b->blockno%NBUCKET;
  acquire(&bcache.locks[index]);
  b->refcnt++;
  release(&bcache.locks[index]);
}

void
bunpin(struct buf *b) {
  int index=b->blockno%NBUCKET;
  acquire(&bcache.locks[index]);
  b->refcnt--;
  release(&bcache.locks[index]);
}
```

f) 测试如下：

```
lock: virtio_disk: #test-and-set 303111 #acquire() 117
lock: proc: #test-and-set 61061 #acquire() 497759
lock: proc: #test-and-set 50510 #acquire() 497779
lock: proc: #test-and-set 49086 #acquire() 497767
lock: proc: #test-and-set 48313 #acquire() 518290
tot= 0
test0: OK
start test1
test1 OK
```

```
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

3) 实验中遇到的问题和解决方法

在本实验中之前写代码的时候考虑不周，在 bget 函数中出现了死锁。

```
}
release(&bcache.locks[index]);
// Not cached.
// Recycle the least recently used (LRU) unused


struct buf *tarbuf=0;
uint minticks=ticks;
acquire(&bcache.lock);
acquire(&bcache.locks[index]);
for(b = bcache.buckets[index].next; b != &bcache
    if(b->dev == dev && b->blockno == blockno)
      // found in hash table
```

在最初写的时候检查是否已有 buf 之后并没有释放锁，这就会导致若此时有另外一个进程正在寻找其他桶的 buf 时会尝试获取该锁（占用了全局锁），而此时本进程又占用了该桶的局部所尝试获取全局锁，进而造成互相占用的死锁发生。通过释放锁在上锁解决。

另外在 usertests 会出现 test writebig: out of block 报错提示，通过查询相关内容之后，需要修改相应的宏定义。

```
#define FSSIZE       10000  // size of file system in blocks
#define MAXPATH      128   // maximum file path name
#define NBUCKET      13    // 哈希表桶数
```

将 fssize 的值增大即可解决。

4) 实验心得

通过完成本实验，我对于 xv6 的缓存机制有了一定的了解，在初始化阶段，每个 buf 的 lasttick 都被初始化为 0，这样的话我们后续就不需要先找空闲的 buf，若没有在使用 LRU 替换；这样做可以直接使用 LRU 替换，更为简便。

同时也了解了自旋锁与睡眠锁二者之间的区别，前者更加适用于临界区较短的情况，可以提高并发性；后者则主要适用于临界区较长的情况，可以减少自旋锁带来的循环迭代，避免浪费 cpu 资源。

评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-lock bcache
make: 'kernel/kernel' is up to date.
== Test running bcachetest == (7.0s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
```

总结

通过完成本次实验，我对于 xv6 操作系统内部的内存分配以及缓冲分配有了更加深入的理解，同时通过本次实验我对于之前学过的一些知识也有了复习，第二个实验中的编码也一定程度上提高了我的编码能力。

本实验总体评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kalloctest == (73.4s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (7.2s)
== Test running bcachetest == (6.3s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests == usertests: OK (132.9s)
== Test time ==
time: OK
Score: 70/70
```