Lab 8: locks

1950787 杨鑫

1. 实验目的：

- 熟悉进程锁的概念及使用
- 修改相关数据结构和加锁策略以减少锁的竞争
- 体会锁的必要性以及所带来的一些问题

2. 实验步骤：

首先切换分支至 lock branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout lock
make clean
```

a. Memory allocator

实验目的：

在本实验中，您将获得重新设计代码以提高并行性的经验。多核机器上并行性差的一个常见症状是高锁争用。提高并行性通常涉及更改数据结构和锁定策略以减少争用。您将为 xv6 内存分配器和块缓存执行此操作。

程序 user/kalloctest 强调 xv6 的内存分配器：三个进程扩大和缩小它们的地址空间，导致对 kalloc 和 kfree 的多次调用。 kalloc 和 kfree 获得 kmem.lock。 kalloctest 打印（作为“#fetch-and-add”）由于尝试获取另一个内核已经持有的锁（对于 kmem 锁和其他一些锁）而导致的循环迭代次数。获取中的循环迭代次数是锁定争用的粗略度量。

对于每个锁，acquire 维护对该锁的调用计数，以及获取中的循环尝试但未能设置锁的次数。 kalloctest 调用一个系统调用，使内核打印 kmem 和 bcache 锁（这是本实验的重点）和 5 个最争用锁的计数。如果存在锁争用，获取循环迭代的次数将会很大。系统调用返回 kmem 和 bcache 锁的循环迭代次数的总和。

kalloctest 中锁争用的根本原因是 kalloc() 有一个空闲列表，由一个锁保护。要消除锁争用，您必须重新设计内存分配器以避免单个锁和列表。基本思想是为每个 CPU 维护一个空闲列表，每个列表都有自己的锁。不同 CPU 上的分配和释放可以并行运行，因为每个 CPU 将在不同的列表上运行。主要挑战将是处理一个 CPU 的空闲列表为空，但另一个 CPU 的列表有空闲内存的情况；在这种情况下，一个 CPU 必须“窃取”另一个 CPU 的空闲列表的一部分。窃取可能会引入锁争用，但希望这种情况很少见。

你的工作是实现每个 CPU 的空闲列表，并在 CPU 的空闲列表为空时进行窃取。您必须为所有以“kmem”开头的锁命名。也就是说，您应该为每个锁调用 initlock，并传递一个以“kmem”开头的名称。

实验分析：

首先需要修改 kmem 数据结构，以前是所有CPU共享它，所以进行 kalloc 和 kfree 时会产生比较多的锁竞争，所以考虑为每个CPU单独设置，即将 kmem 设置为数组。然后相应的需要修改初始化方法，以及 kalloc 和 kfree 方法。

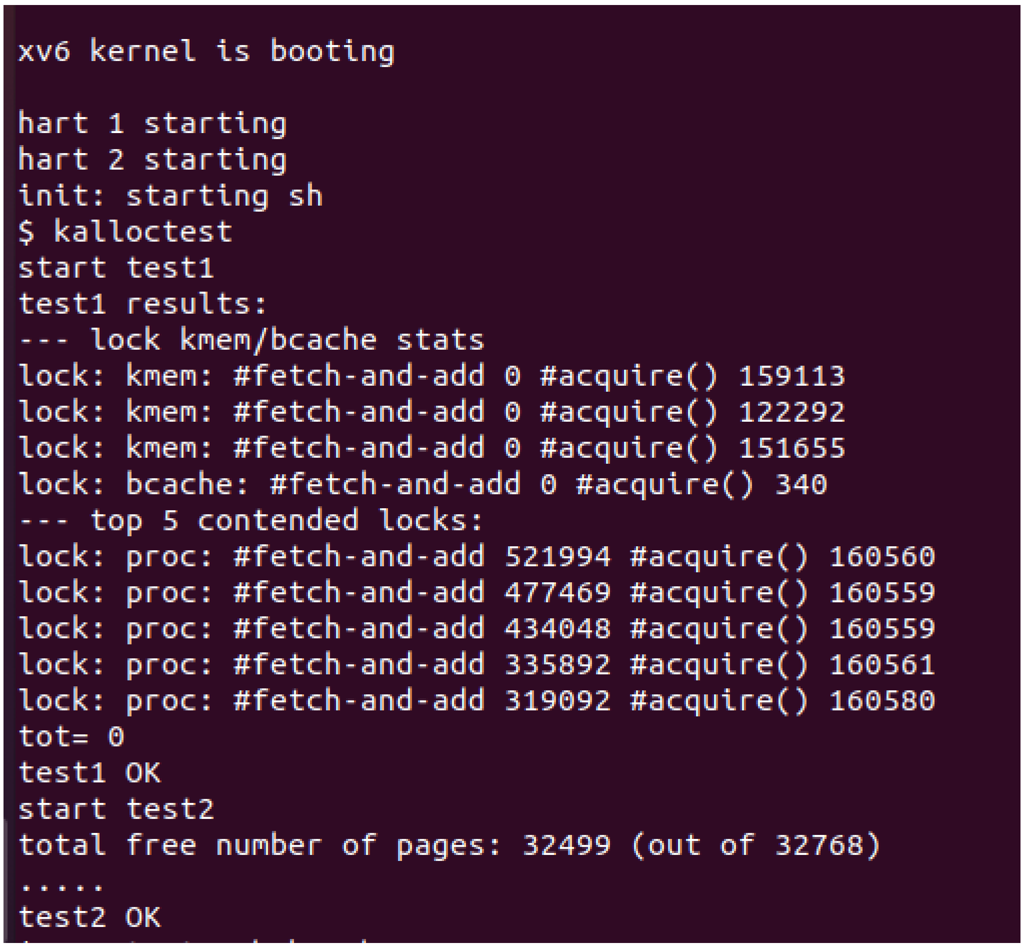
实验核心代码：

kalloc.c

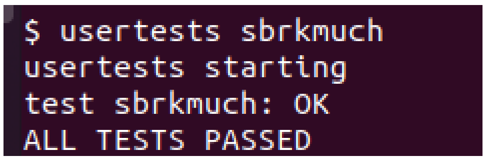
```
// Physical memory allocator, for user processes,
// kernel stacks, page-table pages,
// and pipe buffers. Allocates whole 4096-byte pages.

#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "riscv.h"
#include "defs.h"

void freerange(void *pa_start, void *pa_end);

extern char end[]; // first address after kernel.
                   // defined by kernel.ld.

struct run {
  struct run *next;
};

struct kmem {
  struct spinlock lock;
  struct run *freelist;
};

struct kmem kmemArray[NCPU];

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

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

```c
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
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_current = cpuid();
  acquire(&(kmemArray[cpu_current].lock));
  r->next = kmemArray[cpu_current].freelist;
  kmemArray[cpu_current].freelist = r;
  release(&(kmemArray[cpu_current].lock));
  pop_off();
}

// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  push_off();
  int cpu_current = cpuid();
  acquire(&(kmemArray[cpu_current].lock));
  r = kmemArray[cpu_current].freelist;
  if(r)
    kmemArray[cpu_current].freelist = r->next;
  else {
    for (int i = 0; i < NCPU; i++) {
      if (kmemArray[i].freelist) {
        acquire(&(kmemArray[i].lock));
        r = kmemArray[i].freelist;
        kmemArray[i].freelist = r->next;
        release(&(kmemArray[i].lock));
        break;
      }
    }
  }
  release(&(kmemArray[cpu_current].lock));
  pop_off();

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

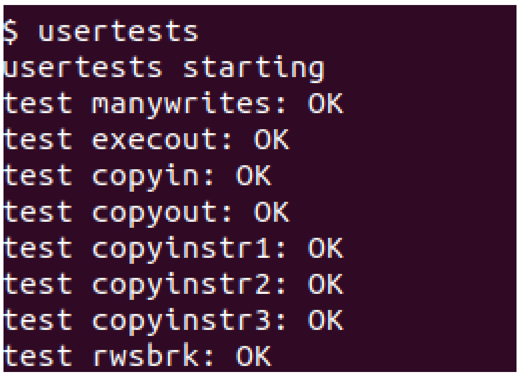
测试:



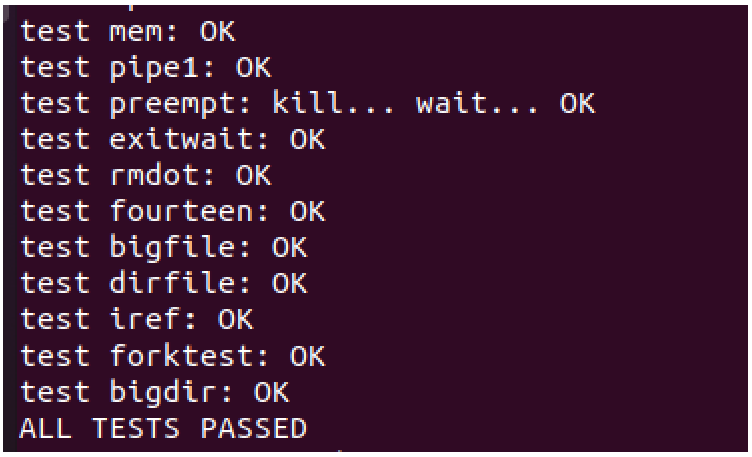
```
xv6 kernel is booting

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() 159113
lock: kmem: #fetch-and-add 0 #acquire() 122292
lock: kmem: #fetch-and-add 0 #acquire() 151655
lock: bcache: #fetch-and-add 0 #acquire() 340
--- top 5 contended locks:
lock: proc: #fetch-and-add 521994 #acquire() 160560
lock: proc: #fetch-and-add 477469 #acquire() 160559
lock: proc: #fetch-and-add 434048 #acquire() 160559
lock: proc: #fetch-and-add 335892 #acquire() 160561
lock: proc: #fetch-and-add 319092 #acquire() 160580
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```



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



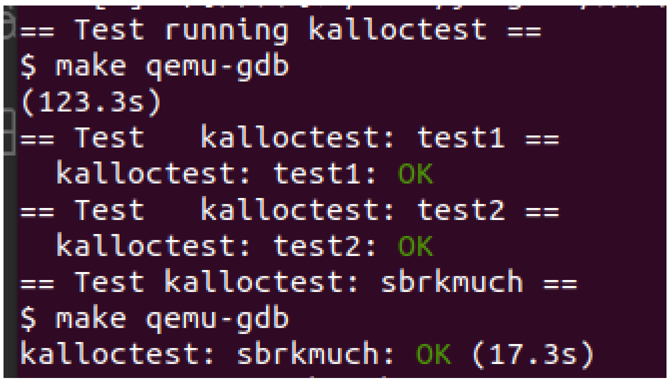
```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
```



```
test mem: OK
test pipe1: 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
```

可以通过 kalloctest，usertests sbrkmuch，usertests 测试

评分：



```
== Test running kalloctest ==
$ make qemu-gdb
(123.3s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (17.3s)
```

可以通过所有测试

b. Buffer cache

实验目的：

本实验是要对 xv6 的磁盘缓冲区进行优化。在初始的 xv6 磁盘缓冲区中是使用一个LRU链表来维护的，而这就导致了每次获取、释放缓冲区时就要对整个链表加锁，也就是说缓冲区的操作是完全串行进行的。

为了提高并行性能，我们可以用哈希表来代替链表，这样每次获取和释放的时候，都只需要对哈希表的一个桶进行加锁，桶之间的操作就可以并行进行。只有当需要对缓冲区进行驱逐替换时，才需要对整个哈希表加锁来查找要替换的块。

本次实验就是要修改数据结构和锁来实现哈希表代替链表，从而提高缓冲区操作的并行度。

实验分析：

由于要将原来的LRU链表改为哈希表，所以需要对数据结构做一定的修改，并添加一个哈希数组用于存放相应的内容。然后在初始化的时候也需要修改，将原来链表的元素均匀的分到哈希数组中并编号，然后对哈希桶中的锁进行初始化等；然后要修改 brelse, bpin, bunpin 让他们操作的锁不再是原来的整个链表，而是对应的桶的锁；最后还要重写 bget 函数，首先在对应的桶当中查找当前块是否被缓存，如果被缓存就直接返回；如果没有被缓存的话，就需要查找一个块并将其逐出替换。这里使用的策略是先在当前桶当中查找，当前桶没有查找到再去全局数组中查找，这样的话如果当前桶中有空闲块就可以避免全局锁，以尽可能减少锁竞争。

在全局数组中查找时，要先加上表级锁，当找到一个块之后，就可以根据块的信息查找到对应的桶，之后再对该桶加锁，将块从桶的链表上取下来，释放锁，最后再加到当前桶的链表上去。

这里有个困难就是全局数组中找到一个块之后，到对该桶加上锁之间有一段其他操作，可能在这个期间这个块就被其他进程夺取了。因此，需要在加上锁之后判断是否被用了，如果被用了就要重新查找。

实验核心代码：

buf.h

```
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 *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
  uint tick;
};
```

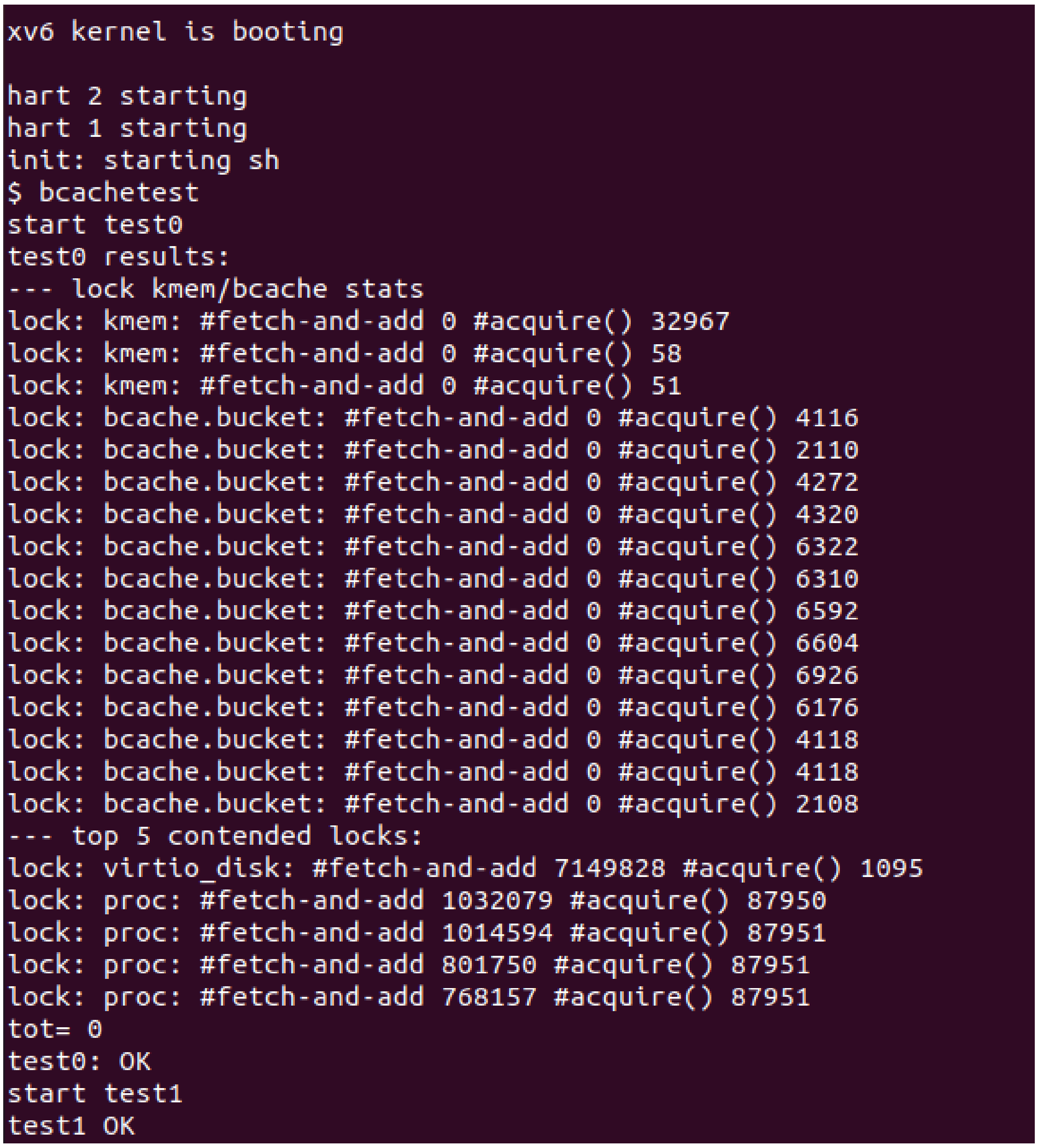
bio.c

```
// Buffer cache.
//
// The buffer cache is a linked list of buf structures holding
// cached copies of disk block contents.  Caching disk blocks
// in memory reduces the number of disk reads and also provides
// a synchronization point for disk blocks used by multiple processes.
//
// Interface:
// * To get a buffer for a particular disk block, call bread.
// * After changing buffer data, call bwrite to write it to disk.
// * When done with the buffer, call brelse.
// * Do not use the buffer after calling brelse.
// * Only one process at a time can use a buffer,
//     so do not keep them longer than necessary.


#include "types.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "riscv.h"
#include "defs.h"
#include "fs.h"
#include "buf.h"
#define HASH_NUM 13

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

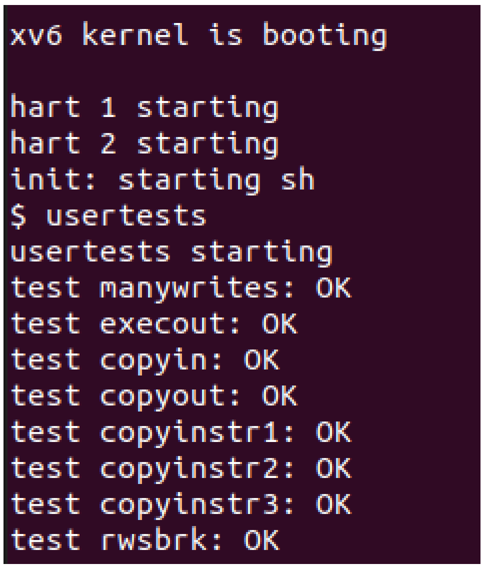
```
} bcache;

struct bmem {
  struct spinlock lock;
  struct buf head;
};

static struct bmem hash_table[HASH_NUM];

void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");

  b = bcache.buf;
  for (int i = 0; i < HASH_NUM; i++) {
    initlock(&(hash_table[i].lock), "bcache.bucket");
    int count = NBUF / HASH_NUM;
    if (i < NBUF % HASH_NUM) count++;

    for (int j = 0; j < count; j++, b++) {
      b->blockno = i;
      b->next = hash_table[i].head.next;
      hash_table[i].head.next = b;
    }
  }

  for (b = bcache.buf; b < bcache.buf + NBUF; b++)
    initsleeplock(&b->lock, "buffer");
}

// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  int num = blockno % HASH_NUM;
  acquire(&(hash_table[num].lock));

  for (b = hash_table[num].head.next; b != 0; b = b->next) {
    if (b->dev == dev && b->blockno == blockno) {
      b->refcnt++;
      release(&(hash_table[num].lock));
      acquiresleep(&b->lock);
      return b;
    }
  }

  int minist = __INT_MAX__;
  struct buf* replace = 0;

  for (b = hash_table[num].head.next; b != 0; b = b->next) {
    if (b->refcnt == 0 && b->tick < minist) {
      replace = b;
      minist = b->tick;
    }
  }
  if (replace) {
    replace->dev = dev;
    replace->blockno = blockno;
    replace->valid = 0;
    replace->refcnt = 1;
    release(&(hash_table[num].lock));
    acquiresleep(&replace->lock);
    return replace;
  }

  acquire(&bcache.lock);

  int tag, rnum;
  do {
    tag = 1;
    for (b = bcache.buf; b < bcache.buf + NBUF; b++) {
      if (b->refcnt == 0 && b->tick < minist) {
        replace = b;
        minist = b->tick;
      }
    }
    if (replace) {
      rnum = replace->blockno % HASH_NUM;
      acquire(&(hash_table[rnum].lock));
      if (replace->refcnt != 0) {       // To avoid locked by others between
find and acquire.
        release(&(hash_table[rnum].lock));
        tag = 0;
      }
    } else {
      panic("bget: no buffers");
    }
  } while (!tag);
  struct buf *pre = &hash_table[rnum].head;
  struct buf *post = hash_table[rnum].head.next;
  while (post != replace) {
    pre = pre->next;
    post = post->next;
  }
  pre->next = post->next;
  release(&(hash_table[rnum].lock));
  replace->next = hash_table[num].head.next;
  hash_table[num].head.next = replace;
  release(&bcache.lock);
  replace->dev = dev;
  replace->blockno = blockno;
  replace->valid = 0;
  replace->refcnt = 1;
  release(&(hash_table[num].lock));
  acquiresleep(&replace->lock);
  return replace;
}


// Return a locked buf with the contents of the indicated block.
```

```c
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!b->valid) {
    virtio_disk_rw(b, 0);
    b->valid = 1;
  }
  return b;
}

// Write b's contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  virtio_disk_rw(b, 1);
}

// Release a locked buffer.
// Move to the head of the most-recently-used list.
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  uint64 num = b->blockno % HASH_NUM;
  acquire(&(hash_table[num].lock));
  b->refcnt--;
  if (b->refcnt == 0) b->tick = ticks;
  release(&(hash_table[num].lock));
}

void
bpin(struct buf *b) {
  uint64 num = b->blockno % HASH_NUM;
  acquire(&(hash_table[num].lock));
  b->refcnt++;
  release(&(hash_table[num].lock));
}

void
bunpin(struct buf *b) {
  uint64 num = b->blockno % HASH_NUM;
  acquire(&(hash_table[num].lock));
  b->refcnt--;
  release(&(hash_table[num].lock));
}
```

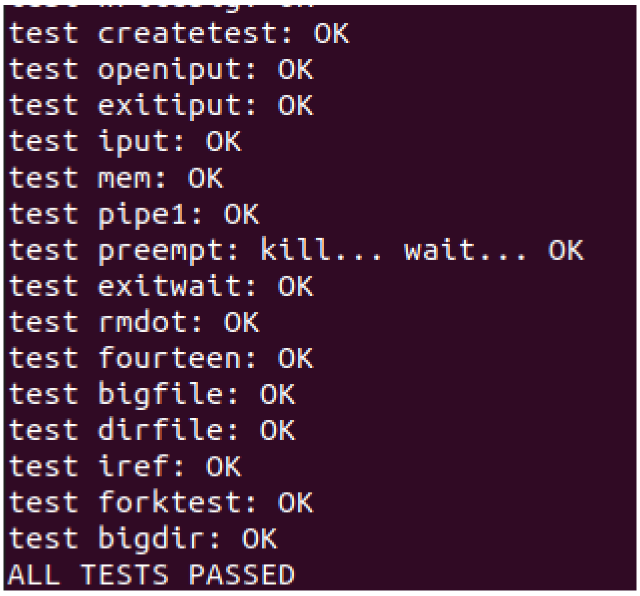
测试：



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32967
lock: kmem: #fetch-and-add 0 #acquire() 58
lock: kmem: #fetch-and-add 0 #acquire() 51
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4116
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2110
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4272
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4320
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6322
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6310
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6592
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6604
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6926
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6176
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4118
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4118
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2108
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 7149828 #acquire() 1095
lock: proc: #fetch-and-add 1032079 #acquire() 87950
lock: proc: #fetch-and-add 1014594 #acquire() 87951
lock: proc: #fetch-and-add 801750 #acquire() 87951
lock: proc: #fetch-and-add 768157 #acquire() 87951
tot= 0
test0: OK
start test1
test1 OK
```



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
```

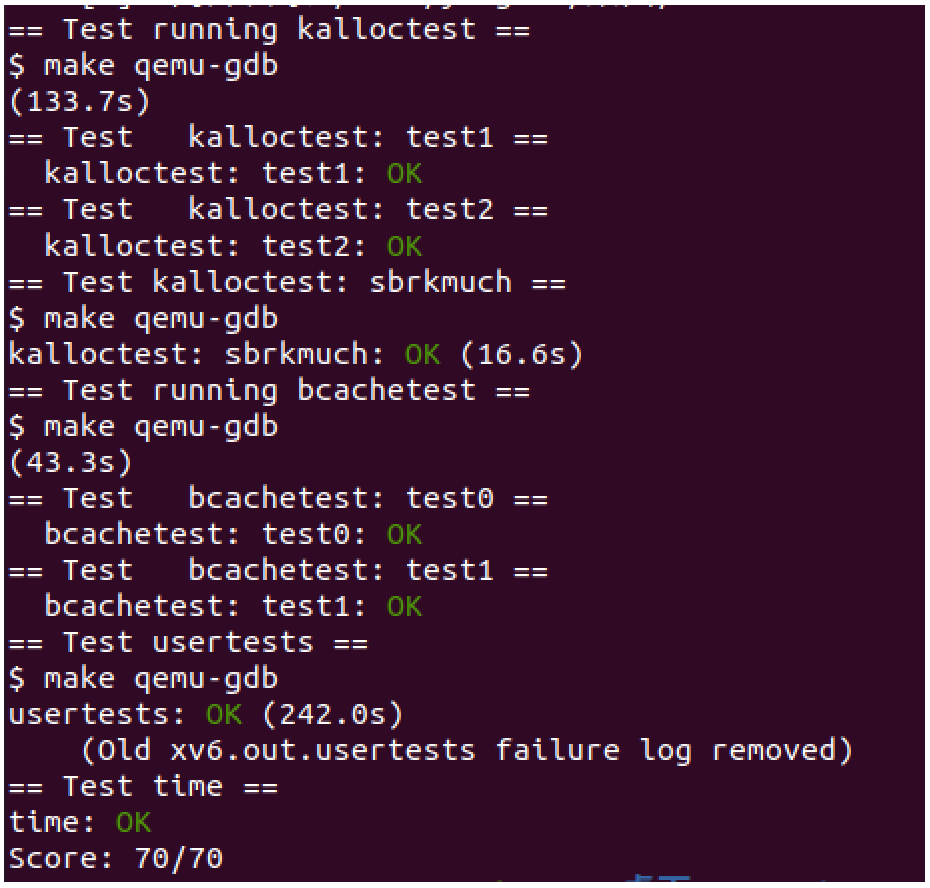


```
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

可以通过 bcachetest 和 usertests 测试

3. 实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：



```
== Test running kalloctest ==
$ make qemu-gdb
(133.7s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (16.6s)
== Test running bcachetest ==
$ make qemu-gdb
(43.3s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (242.0s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 70/70
```

通过了所有测试

4. 问题以及解决办法：

a.没有考虑周全并发执行时可能的错误而导致错误

上面提到了一点，在全局数组中找到一个块之后，到对该桶加上锁之间有一段其他操作，可能在这个期间这个块就被其他进程夺取了。因此，需要在加上锁之后判断是否被用了，如果被用了就要重新查找。我在开始的时候没有考虑这一点，导致测试一直不通过，后来仔细排查才发现这样一个漏洞。加锁就是为了解决并发执行程序时可能存在的错误，但是往往很多时候我们没有考虑周全，如上述这个例子，它比较不容易被发现，但是会导致比较严重的错误，所以这就要求在操作系统级编程的时候要准确的处理逻辑关系，仔细排查任何一个可能存在的错误，否则会为后面的编程带来很大的麻烦。

b.没有将所有块放入哈希桶而导致异常

在初始化的时候，一开始只是为每个桶分配 NBUF / HASH_NUM 个块，然后进行一系列初始化操作。但是没有注意到 NBUF 不一定能被 HASH_NUM 整除，这样就会导致部分快没有被放入哈希桶并且初始化。然后会出现错误，导致测试不通过，后来稍微修改了一下，保证每一块都能放入桶中并初始化，程序才能得以正常执行并通过测试。

5. 实验心得：

1. 此次实验熟悉了进程锁的使用，体会了进程的同步互斥关系
2. 通过对内存分配和磁盘缓冲区分配的加锁等操作进行优化和改进以减少锁竞争和一些其他的问题，从而提高程序并发执行的效率
3. 在有关锁的编程中容易遇到很多“坑”，编程时要格外小心并注意 debug