Lab8：Locks

该实验由Memory allocator、Buffer cache两部分组成。

1. Memory allocator

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。

实验步骤

1. 在kernel/kalloc.c中将kmem修改为数组，为每个CPU分配一个kmem

struct {

  struct spinlock lock;

  struct run \*freelist;

} kmem[NCPU];

1. 修改kinit函数，为当前CPU的’freelist’分配所有的空闲内存空间

  for (int i = 0; i < NCPU; i++)

    initlock(&kmem[i].lock, "kmem");

1. 修改kfree函数，让他在获取cpuid前需要关闭中断

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;

  //获取cpuid前关闭中断

  push\_off();

  int id = cpuid();

  pop\_off();

  acquire(&kmem[id].lock);

  r->next = kmem[id].freelist;

  kmem[id].freelist = r;

  release(&kmem[id].lock);

}

1. 修改kalloc函数，用于查找当前CPU中是否有空闲块，没有则需要从其他CPU中获取。

void \*

kalloc(void)

{

  struct run \*r;

  push\_off();

  int id = cpuid();

  pop\_off();

  acquire(&kmem[id].lock);

  r = kmem[id].freelist;

  if(r)

    kmem[id].freelist = r->next;

  else {

    for (int i = 0; i < NCPU; i++) {

      if (i == id)

          continue;

      acquire(&kmem[i].lock);

      r = kmem[i].freelist;

      if(r)

        kmem[i].freelist = r->next;

      release(&kmem[i].lock);

      if(r)

          break;

    }

  }

  release(&kmem[id].lock);

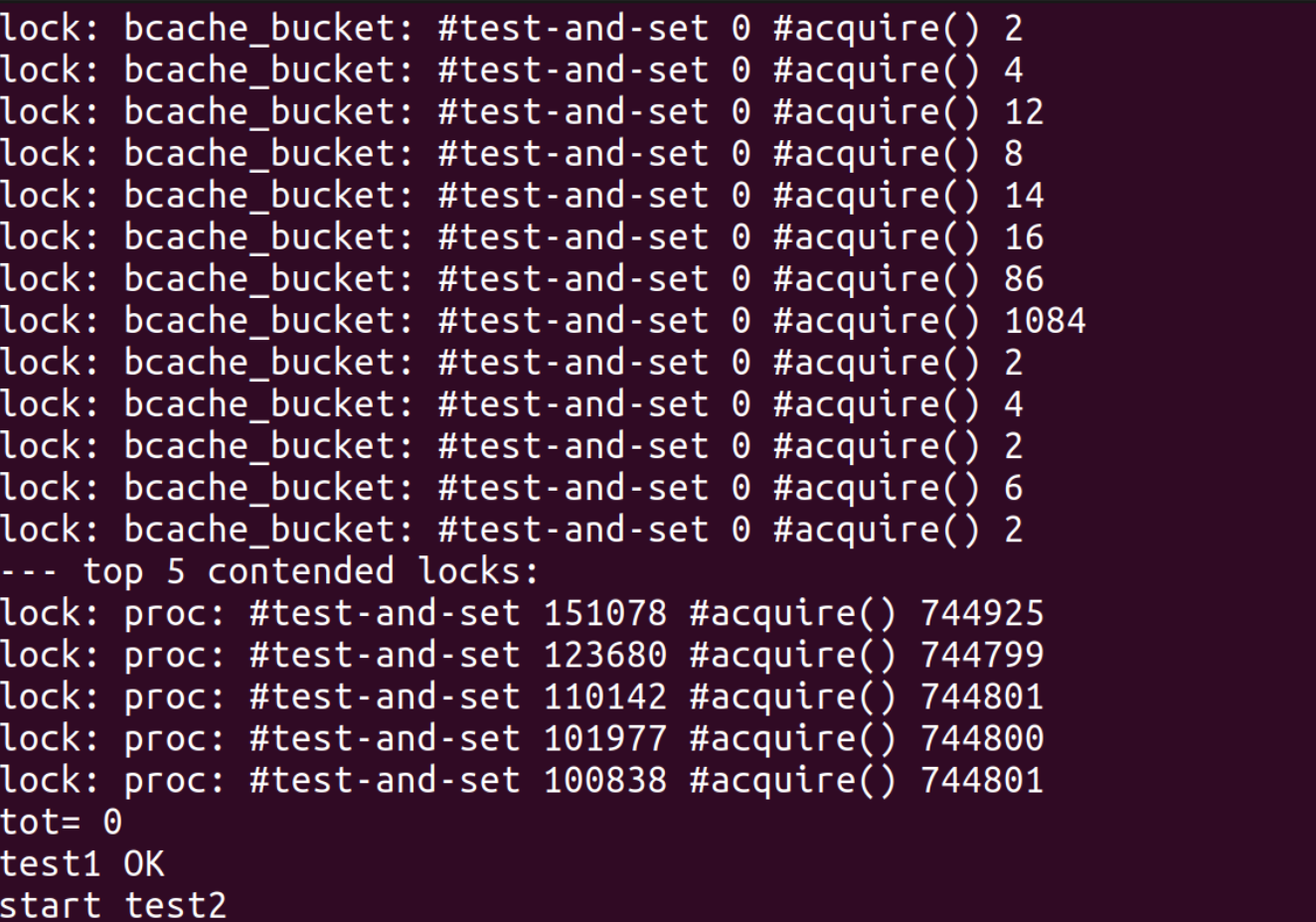
  if(r)

    memset((char\*)r, 5, PGSIZE); // fill with junk

  return (void\*)r;

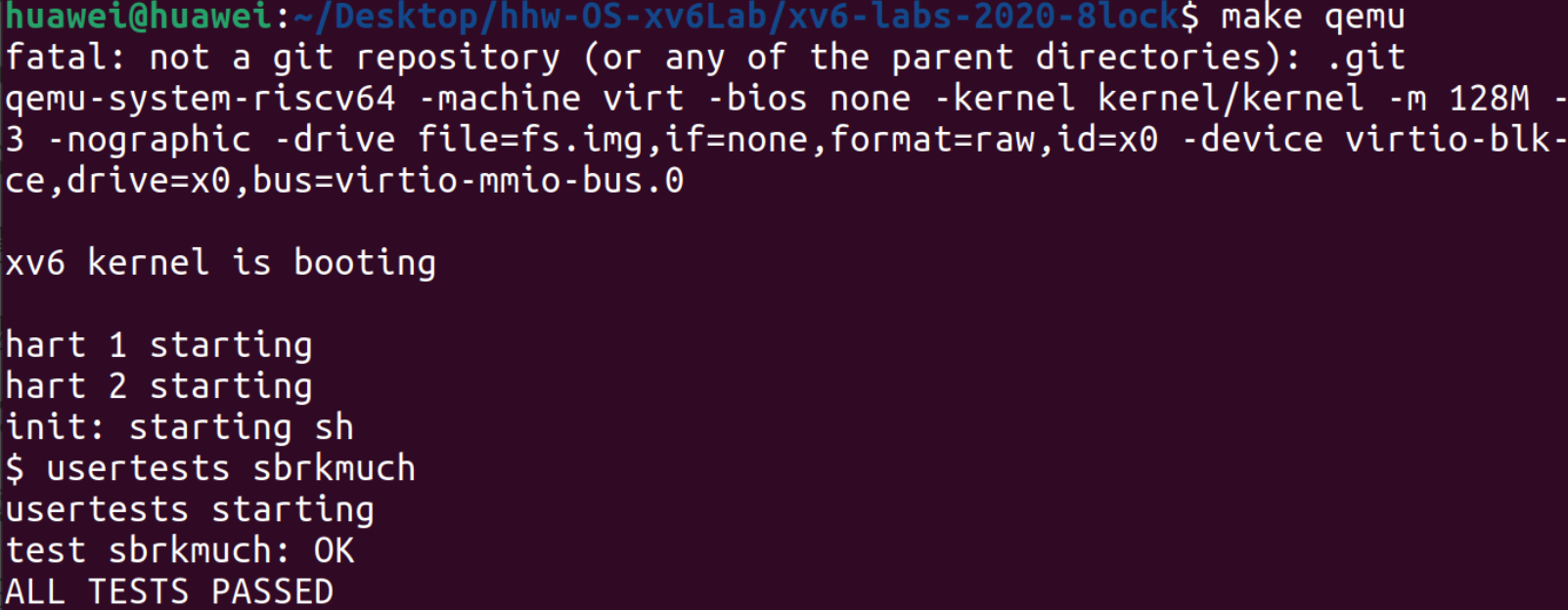
}

5.利用make qemu指令运行xv6，在命令行中输入kalloctest:

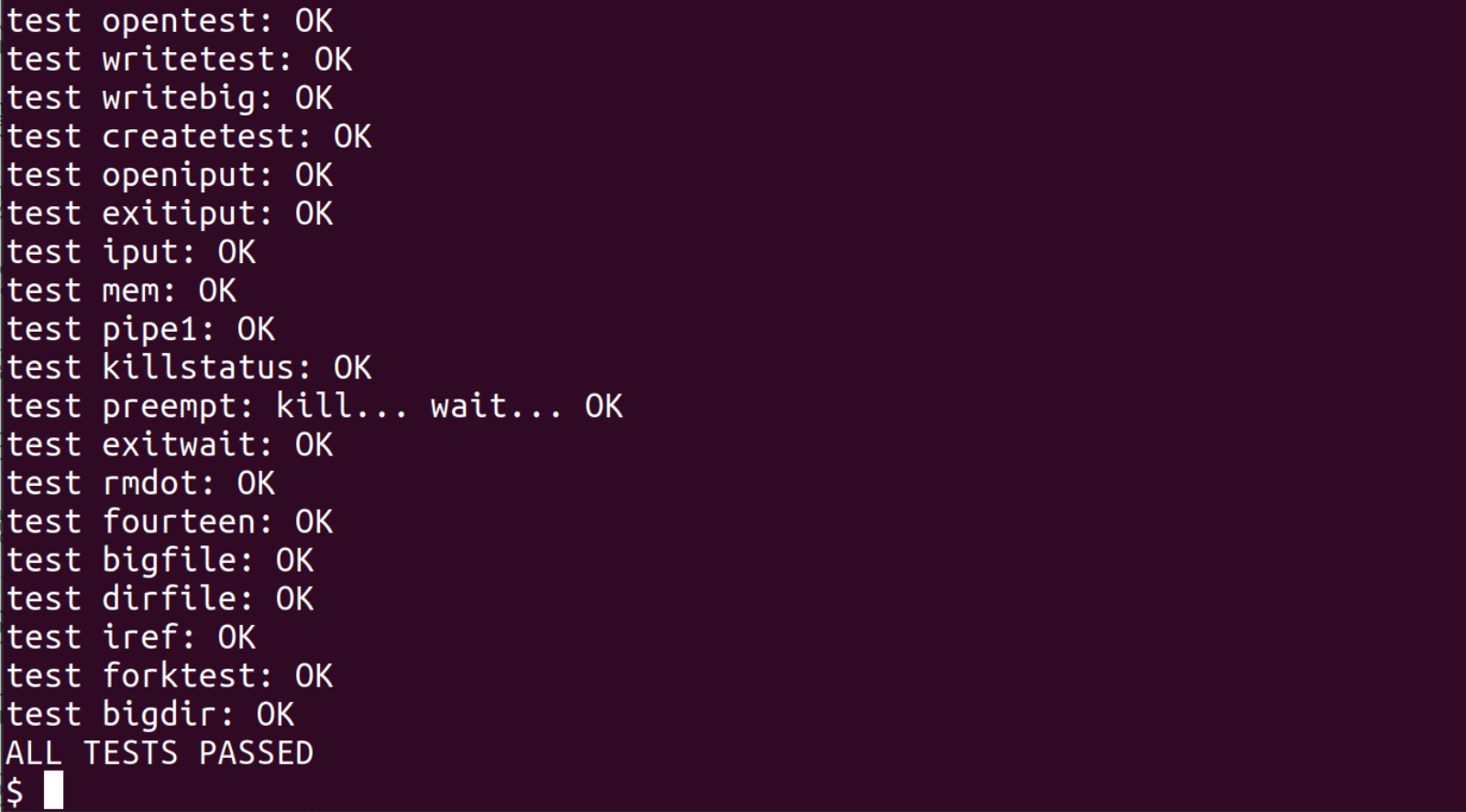


在 test1 中，可以看到 acquire 的循环迭代次数明显减少，表明锁的争用较之前有所减少，每个 CPU 现在有了自己的 freelist，减少了 CPU 之间在访问内存分配器时的竞争。

6. 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。



7.运行 usertests 测试，确保所有的用户测试都通过。



遇到的问题与心得

遇到的问题：

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。

实验心得

本实验我理解锁了竞争，包括锁竞争产生的原因以及解决方法。理解通过增加锁的数量来降低锁竞争的原理，并应用到程序中。

2. Buffer cache

实验目的

修改块缓存，以便在运行`bcachetest`时，`bcache`中所有锁的获取循环迭代次数接近于零。目的是通过重新设计文件系统的 buffer & cache 的数据结构来降低锁的竞争

实验步骤

1. 根据实验指导，修改kernel/bio.c中的bcache结构体

struct {

  struct spinlock biglock;

  struct spinlock lock[NBUCKET];//每个桶的锁

  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 head[NBUCKET];//

} bcache;

将buf修改为大小为13的数组，获取trap.c中的ticks,并且还要所有桶中的锁

1. 在buf.h头文件中添加lastuse声明

  uint lastuse;

1. 修改binit函数，将双向链表进行初始化操作，每个结点前驱结点和后继结点都是源节点

void

binit(void)

{

  struct buf \*b;

  initlock(&bcache.biglock, "bcache\_biglock");

  for (int i = 0; i < NBUCKET; i++)

    initlock(&bcache.lock[i], "bcache");

  // Create linked list of buffers

  for (int i = 0; i < NBUCKET; i++) {

    bcache.head[i].next = &bcache.head[i];

    bcache.head[i].prev = &bcache.head[i];

  }

  for(b = bcache.buf; b < bcache.buf+NBUF; b++){

    b->next = bcache.head[0].next;

    b->prev = &bcache.head[0];

    initsleeplock(&b->lock, "buffer");

    bcache.head[0].next->prev = b;

    bcache.head[0].next = b;

  }

}

1. 修改bio.c下的brelse,bunpin,bpin函数，并增加一个散列函数hash实现加锁和散列。

int

hash(int blockno)

{

  return blockno % NBUCKET;

}

void

brelse(struct buf \*b)

{

  if(!holdingsleep(&b->lock))

    panic("brelse");

  releasesleep(&b->lock);

  int i = hash(b->blockno);

  acquire(&bcache.lock[i]);

  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.head[i].next;

    //b->prev = &bcache.head[i];

    //bcache.head[i].next->prev = b;

    //bcache.head[i].next = b;

    b->lastuse = ticks;

  }

  release(&bcache.lock[i]);

}

void

bpin(struct buf \*b) {

  int i = hash(b->blockno);

  acquire(&bcache.lock[i]);

  b->refcnt++;

  release(&bcache.lock[i]);

}

void

bunpin(struct buf \*b) {

  int i = hash(b->blockno);

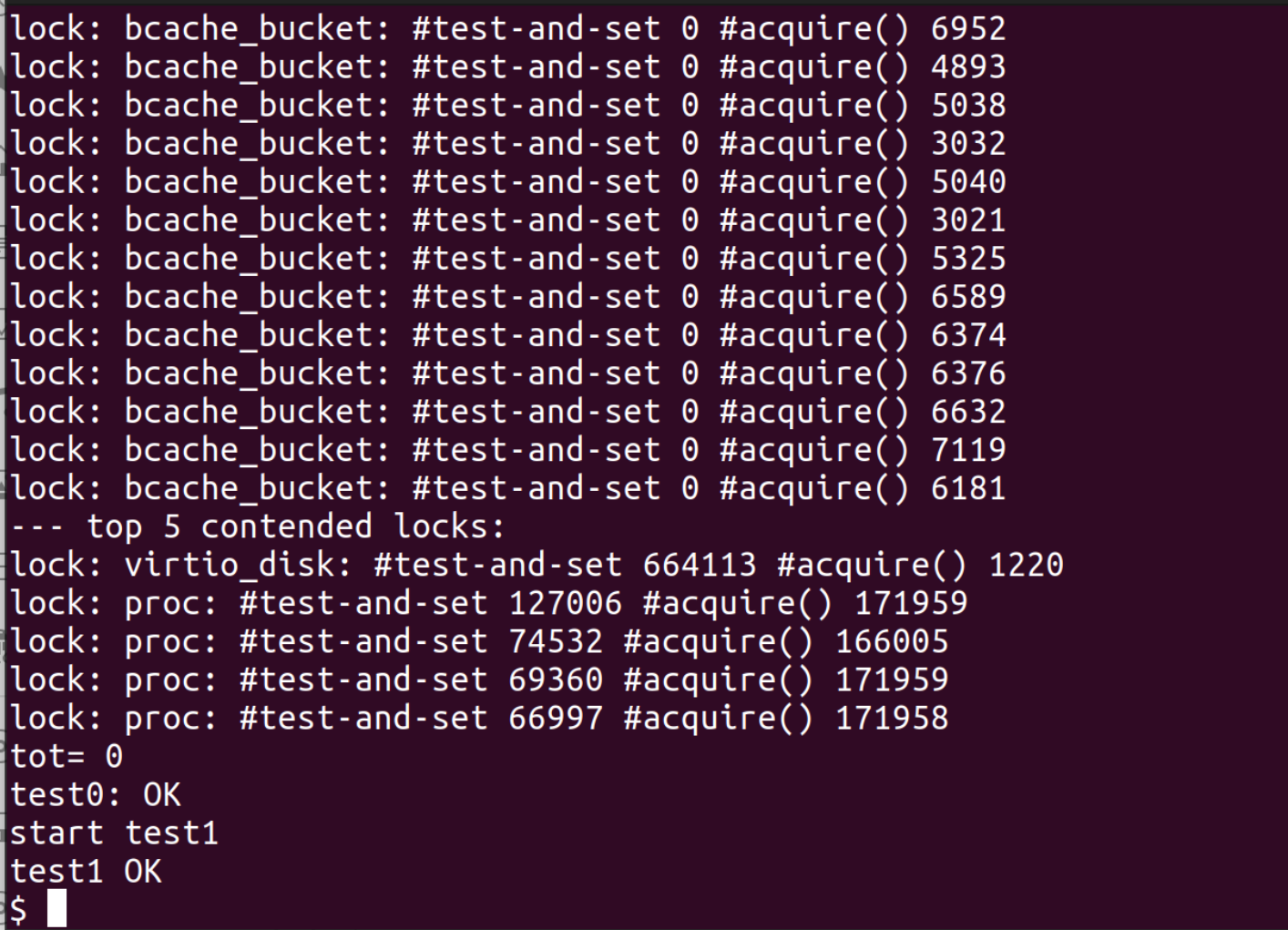
  acquire(&bcache.lock[i]);

  b->refcnt--;

  release(&bcache.lock[i]);

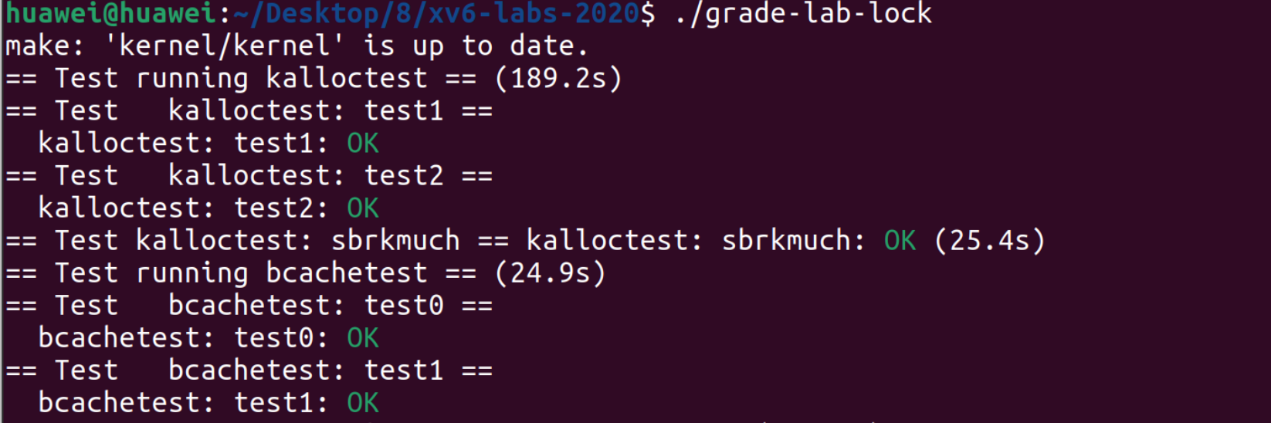
}

1. 最后修改bget函数，先遍历判断缓冲是否命中，若命中则直接返回，若没有命中则要释放锁。因为释放锁后可能会有新的缓存，所以需要挨次遍历检查是否命中，若还是没有命中则获取这个桶的空闲块，若该桶没有空闲块则去别的桶寻找。
2. 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。

7. ./grade-lab-lock:



遇到的问题与心得

遇到的问题：

真正实现之前，一定要修改 kernel/param.h 里的宏 FSSIZE，改大成 10000,由于实验要求没有指出这一点，导致我在 usertests 的 writebig 测试点卡在 PANIC: balloc: out of blocks 很久

实验心得

本实验是关于并行编程中问题的性能优化。在处理多线程或多进程中资源抢占而导致的锁竞争现象，常用的做法是通过分割资源并且分别加锁。锁的数量越多，单个锁上的冲突就越少（但是也要考虑锁的数量增多的负面影响）。我在实验中体会了分割资源来使锁竞争得到缓解、提高资源并行利用率。