《操作系统》Assignment3

年冬冬

18364073

2021 年 1 月 23 日

目录

1.xv6 Lab: Multithreading/Uthread: switching between threads...... 3

1.1 题目要求 3

1.2 实验代码 3

1.3 实验结果 5

2.Xv6 lab: Lock/Memory allocator................................ 6

2.1 题目要求 6

2.2 实验代码 6

2.3 实验结果 8

3.Xv6 lab: Lock/Buffer cache 10

3.1 题目要求 10

3.2 实验代码 10

3.3 实验结果 14

4.Xv6 lab: File System/Large files............................ 15

4.1 题目要求 15

4.2 实验代码 15

4.3 实验结果 18

1.xv6 Lab: Multithreading/Uthread: switching between threads

1.1 题目要求

在本练习中，你将为用户级线程系统设计上下文切换机制，然后实现它。首先，你的 xv6 有两个文件 user/uthread.c 和 user/uthread_switch.S，以及 Makefile 中的一个用于构建 uthread 程序的规则。uthread.c 包含大多数用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和在线程之间切换的代码。你的工作是想出一个创建线程的计划，并保存/恢复寄存器以在线程之间切换，并实施该计划。

1.2 实验代码

1. 在 struct thread 添加寄存器字段，能够保存进程内容。其中 ra 为返回地址，sp 为栈顶指针地址。

```
struct thread {
  /*stored registers*/
  uint64 ra;
  uint64 sp;
  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;

  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */

};
```

2. 在 thread_create()中添加线程进程入口，设置栈顶指针 sp 进行用户内存分配。

```
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->ra=(uint64)func;
  t->sp=(uint64)&t->stack[STACK_SIZE-1];
}
```

3. 在 thread_schedule()中添加调用

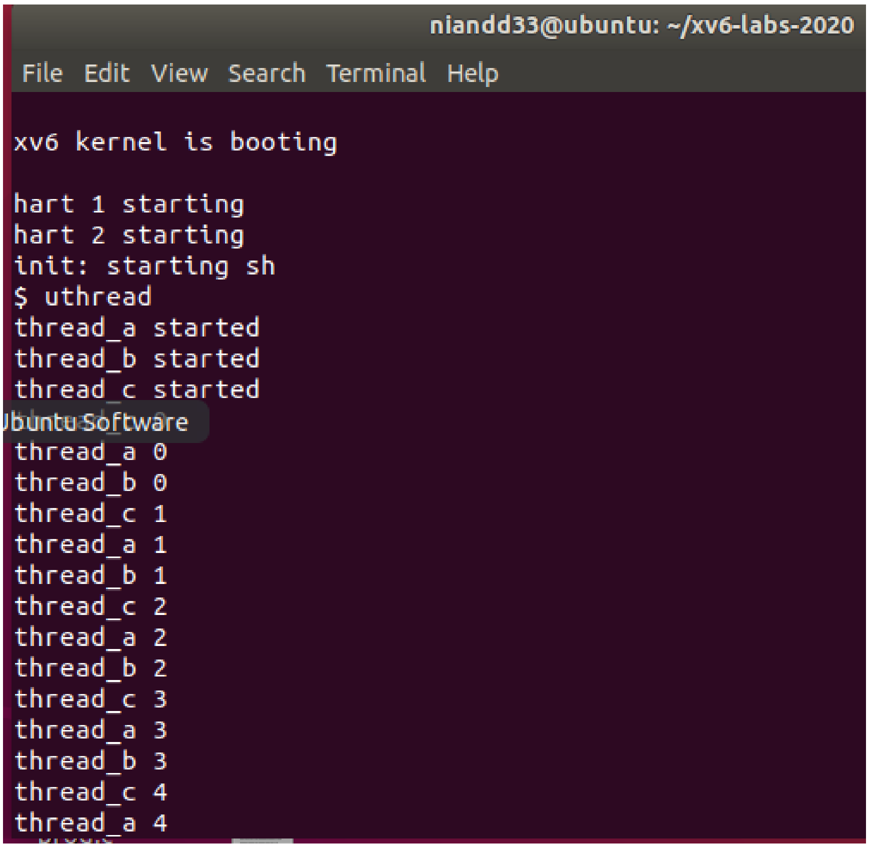
Open ▾ *uthread.c ~/xv6-labs-2020/user

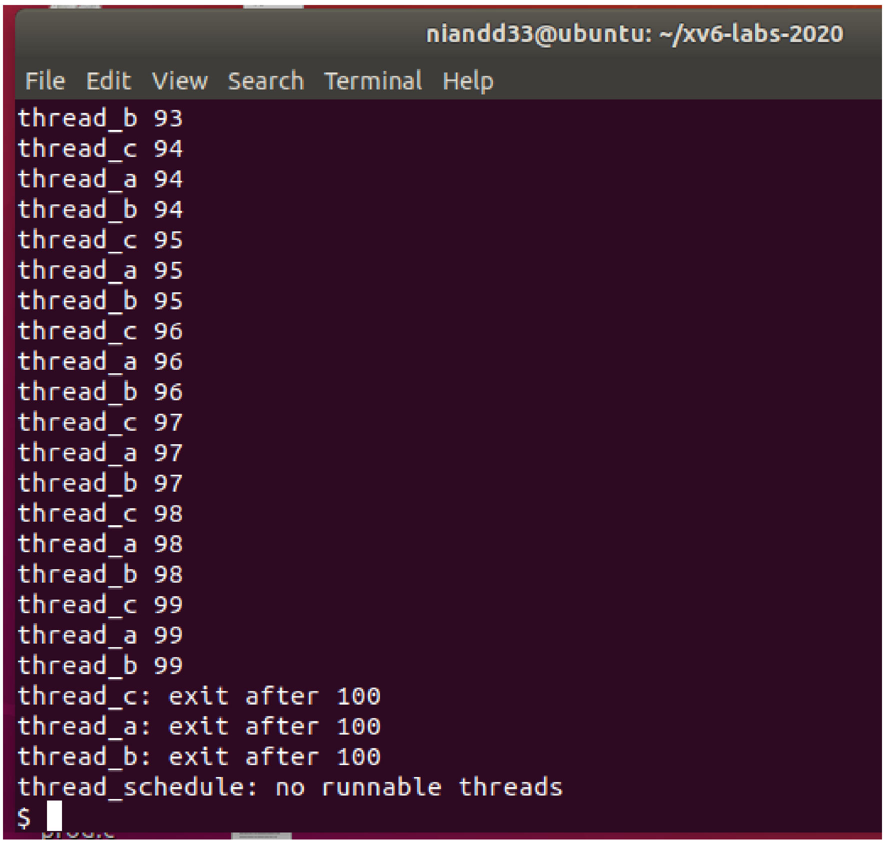
```c
  next_thread = 0;
  t = current_thread + 1;
  for(int i = 0; i < MAX_THREAD; i++){
    if(t >= all_thread + MAX_THREAD)
      t = all_thread;
    if(t->state == RUNNABLE) {
      next_thread = t;
      break;
    }
    t = t + 1;
  }

  if (next_thread == 0) {
    printf("thread_schedule: no runnable threads\n");
    exit(-1);
  }

  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:*/
     thread_switch((uint64)t,(uint64)next_thread);

  } else
    next_thread = 0;
}
```

4. 在 uthread_switch.S 中添加寄存器

```
thread_switch:
        /* YOUR CODE HERE */
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)
        ret    /* return to ra */
```

1.3 实验结果


nianddd33@ubuntu: ~/xv6-labs-2020
File Edit View Search Terminal Help
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
Ubuntu Software
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4


nianddd33@ubuntu: ~/xv6-labs-2020
File Edit View Search Terminal Help
thread_b 93
thread_c 94
thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$

2.Xv6 lab: Lock/Memory allocator

2.1 题目要求

对于每个锁，acquire 都会维护获取该锁的调用次数，以及获取中的循环尝试但未设置锁的次数。如果存在锁争用，则获取循环的迭代次数将很大。kalloctest 调用一个系统调用，系统调用返回 kmem 和 bcache 锁的循环迭代总数。

kalloctest 中锁争用的根本原因是 kalloc（）具有单个空闲列表，并受单个锁保护。要删除锁争用，需要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。主要的挑战将是处理一个 CPU 的空闲列表为空，而另一个 CPU 的列表具有空闲内存的情况。在这种情况下，一个 CPU 必须“窃取”另一 CPU 空闲列表的一部分，窃取可能会引入锁争用。

2.2 实验代码

1. 首先将 kmem 修改为数组，这样每个 cpu 对应一个 freelist 和 lock。

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

2. 修改 kernel/kalloc.c 中的 kinit 函数,初始化 kmem 时将每个 cpu 对应 kmem[i]都初始化.

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

3. 修改 kfree 函数

```
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;

```

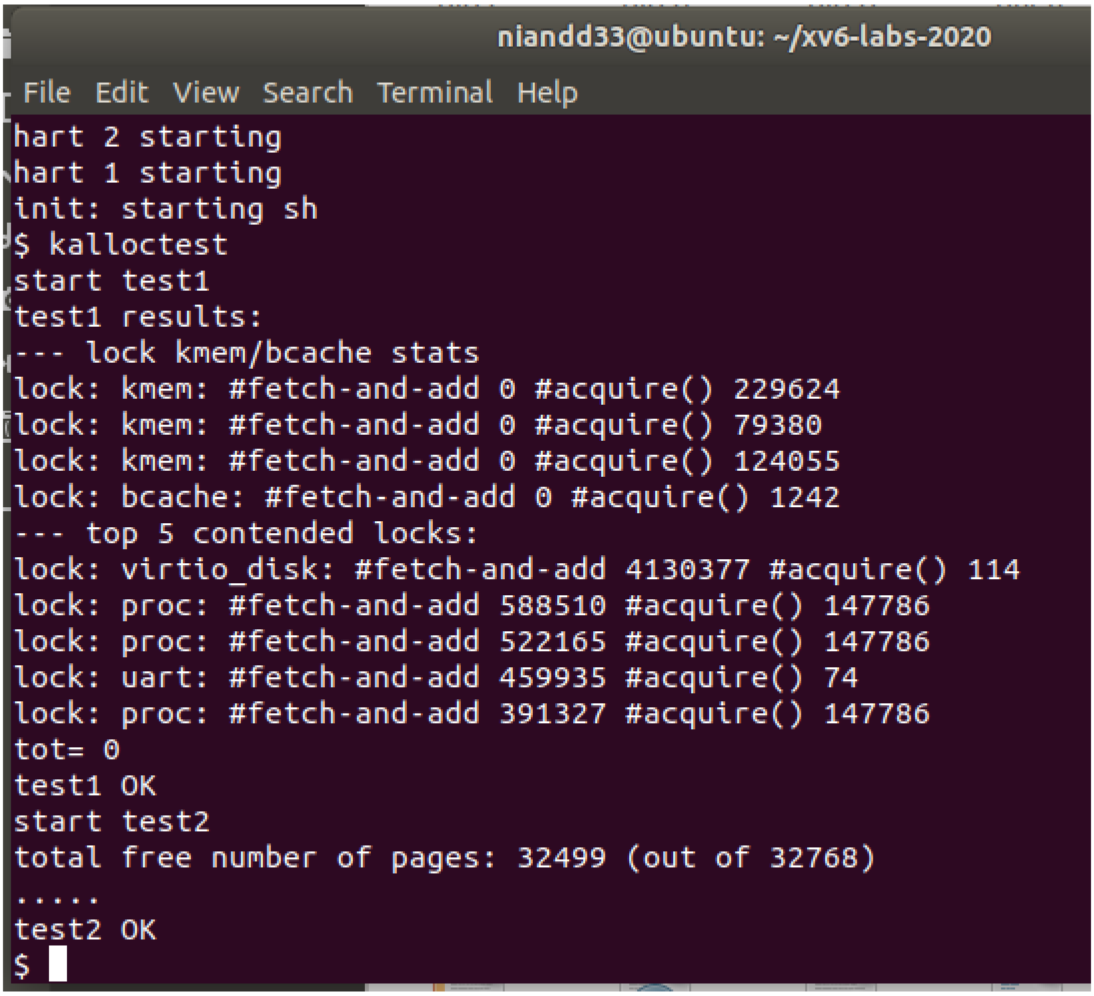
```
  push_off();//  关中断
  int i=cpuid();// CPU id
  acquire(&kmem[i].lock);
  r->next = kmem[i].freelist;
  kmem[i].freelist = r;
  release(&kmem[i].lock);

  pop_off();//开中断
}
```

4. 修改 kalloc 函数

```
void *
kalloc(void)
{
  struct run *r;

  push_off();//  关中断
  int i=cpuid();// CPU id
  acquire(&kmem[i].lock);
  r = kmem[i].freelist;
  if(r)
    kmem[i].freelist = r->next;
  release(&kmem[i].lock);

  if(!r)//如果当前 cpu 对应 freelist 为空
  {
     for(int j=0;j<NCPU;j++)//从其他 cpu 的 freelist 中借用,遍历所有 CPU
     {
       if(j!=i)
       {
          acquire(&kmem[j].lock);//获取该 CPU 的锁
          if(kmem[j].freelist)
      {
          r=kmem[j].freelist;//获取该 CPU 的空闲列表
              kmem[j].freelist=r->next;  //该 CPU 空闲列表-1，给窃取者
          release(&kmem[j].lock);
          break;
      }
          release(&kmem[j].lock);
       }
     }
  }
```

```c

  pop_off();//开中断

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

2.3 实验结果



```
nianddd33@ubuntu: ~/xv6-labs-2020
File  Edit  View  Search  Terminal  Help
.....
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 truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3247
            sepc=0x00000000000056a0 stval=0x00000000000056a0
usertrap(): unexpected scause 0x000000000000000c pid=3248
```

```
nianddd33@ubuntu: ~/xv6-labs-2020
File  Edit  View  Search  Terminal  Help
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6283
            sepc=0x00000000000022cc stval=0x000000000000fb90
OK
test opentest: OK
test writetest: OK
test writebig: 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
$
```

3.Xv6 lab: Lock/Buffer cache

3.1 题目要求

测试命令 bcachetest 创建多个进程，这些进程重复读取不同的文件，如果多个进程密集使用文件系统，则它们可能会争用 bcache.lock。所以输出 bcache 锁的获取循环迭代次数将很高。题目要求修改块缓存，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。题目建议使用一个哈希表在缓存中查找块号，该哈希表每个哈希存储桶均具有锁定状态。

3.2 实验代码

1.选取题目建议的 bucket 大小 13，使用固定数量的存储桶，不动态调整哈希表的大小。

```
#define NBUCKET 13
struct {
  struct spinlock lock[NBUCKET];//每个 bucket 都有一个锁
  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 hashbucket[NBUCKET];//哈希表，数据结构为链表
} bcache;


uint idx(uint blockno)//散列函数
{
   return blockno % NBUCKET;
}
```

2. 在 kernel/buf.h 的 struct 中添加字段 time_stamp，用以标记 buf 的时间戳。通过此更改，brelse 不需要获取 bcache 锁，并且 bget 可以根据时间戳选择最近最少使用的块。

```
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 //最少使用的
  struct buf *next; // 最近使用的
  uchar data[BSIZE];
  uint time_stamp;//时间戳
};
```

3. 修改 kernel/bio.c 中的 binit()函数。

```
void
binit(void)
{
  struct buf *b;
  for (int i=0; i<NBUCKETS;i++){
      initlock(&bcache.lock[i], "bcache");// 将 bucket 的头节点初始化为自己
      bcache.hashbucket[i].prev = &bcache.hashbucket[i];
      bcache.hashbucket[i].next = &bcache.hashbucket[i];
  }

  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
     b->time_stamp  = ticks; // 记录一下 b 的时间戳
     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. 修改 bget 函数，在哈希表中搜索缓冲区，并在找不到缓冲区时为该缓冲区分配一个条目，这必须是原子的。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  int i= idx(blockno);//获取当前缓冲区块号的哈希码
  acquire(&bcache[i].lock);

  // Is the block already cached?
  for(b = bcache.hashbucket[i].next; b != &bcache.hashbucket[i]; b = b->next
){
          if(b->dev==dev && b->blockno == blockno)//hit
          {
              b->time_stamp  = ticks; // 记录一下 b 的时间戳
              b->refcnt++;
              release(&bcache.lock[i]);
              acquiresleep(&b->lock);
              return b;
          }// 找到最小的时间戳对应 buf
```

```

  }

  // Not cached.如果找不到，就需要从其它 bucket 中寻找
  // Recycle the least recently used (LRU) unused buffer.
  for (int newnum = idx(blockno+1); newnum != i; newnum = (newnum+1)%NBUCKET){
      acquire(&bcache.lock[newnum]);
      // 遍历该 bucket 的链表
      for(b = bcache.hashbucket[newnum].prev; b != &bcache.hashbucket[newnum]; b = b->prev){
         if(b->refcnt  == 0) {// 找到了，修改该缓冲区块相应属性，并移到对应的 bucket 中
            b->time_stamp  = ticks; // 记录一下 b 的时间戳
            b->dev = dev;
            b->blockno = blockno;
            b->valid = 0;
            b->refcnt = 1;

            // 先移出原来 bucket
            b->next->prev = b->prev;
            b->prev->next = b->next;
            release(&bcache.lock[newnum]);

            //放到需要该缓冲区块的 bucket
            b->next = bcache.hashbucket[i].next;
            b->prev = &bcache.hashbucket[i];

            //将该区块连接到当前 bucket 中
            bcache.hashbucket[i].next->prev = b;
            bcache.hashbucket[i].next = b;
            release(&bcache.lock[i]);
            acquiresleep(&b->lock);
            return b;
         }
      }
      release(&bcache.lock[newnum]);  // 当前 bucket 中没找到，仍要释放锁
  }
  panic("bget: no buffers");
}
```

5. 修改 brelse，bpin，bunpin 函数。

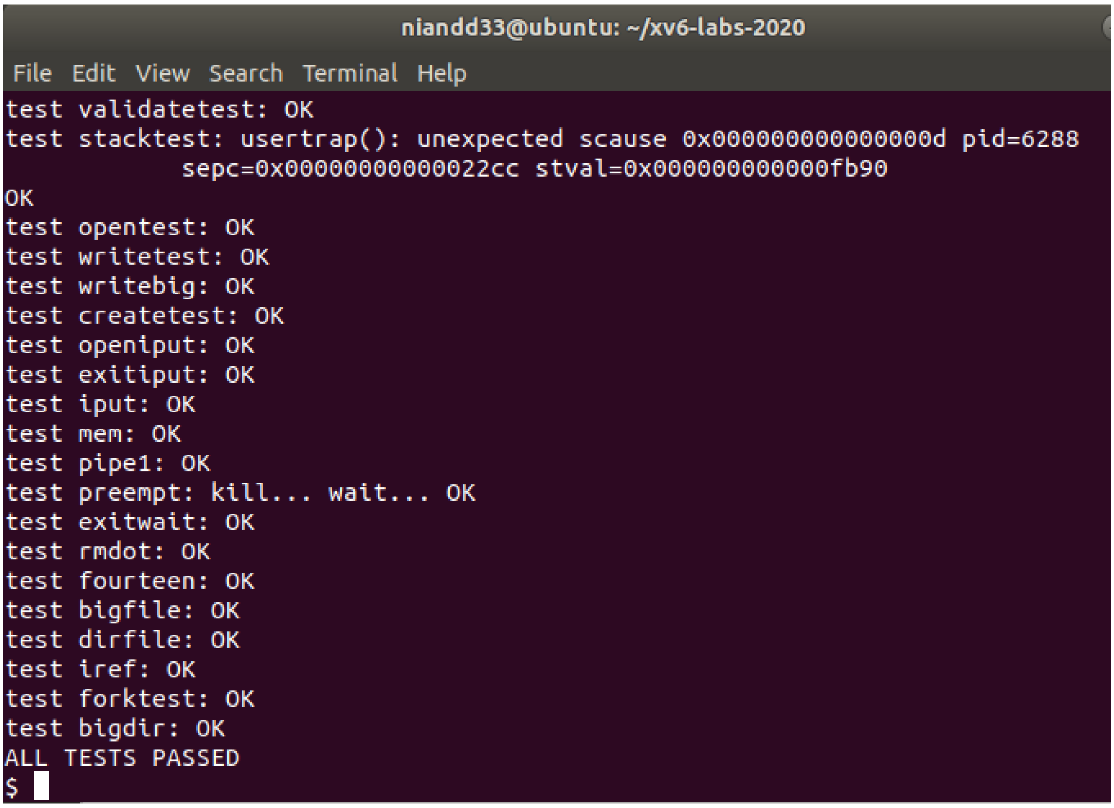
```c
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");
  releasesleep(&b->lock);

  int i=idx(b->blockno);

  b->time_stamp  = ticks;//用时间戳代替原先的上锁
  if(b->time_stamp == ticks){
     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.hashbucket[i].next;
        b->prev = &bcache.hashbucket[i];
        bcache.hashbucket[i].next->prev = b;
        bcache.hashbucket[i].next = b;
      }
   }
}
```

```c
void
bpin(struct buf *b) {
  int i=idx(b->blockno);
  acquire(&bcache.lock[i]);
  b->refcnt++;
  release(&bcache.lock[i]);
}

void
bunpin(struct buf *b) {
  int i=idx(b->blockno);
  acquire(&bcache.lock[i]);
  b->refcnt--;
  release(&bcache.lock[i]);
}
```

3.3 实验结果

```
niandd33@ubuntu: ~/xv6-labs-2020
File Edit View Search Terminal Help
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32901
lock: kmem: #fetch-and-add 0 #acquire() 61
lock: kmem: #fetch-and-add 0 #acquire() 84
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8392
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8698
lock: bcache.bucket: #fetch-and-add 0 #acquire() 10442
lock: bcache.bucket: #fetch-and-add 0 #acquire() 10756
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8526
lock: bcache.bucket: #fetch-and-add 0 #acquire() 9898
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8478
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 9070749 #acquire() 1026
lock: proc: #fetch-and-add 1668070 #acquire() 90544
lock: proc: #fetch-and-add 1469935 #acquire() 90523
lock: proc: #fetch-and-add 1332378 #acquire() 90522
lock: proc: #fetch-and-add 1282240 #acquire() 90523
tot= 0
test0: OK
start test1
test1 OK
$
```

```
niandd33@ubuntu: ~/xv6-labs-2020
File Edit View Search Terminal Help
test0: OK
start test1
test1 OK
$ 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 truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3252
            sepc=0x00000000000056a0 stval=0x00000000000056a0
usertrap(): unexpected scause 0x000000000000000c pid=3253
            sepc=0x00000000000056a0 stval=0x00000000000056a0
OK
test badarg: OK
```



4.Xv6 lab: File System/Large files

4.1 题目要求

测试命令 bigfile 希望能够创建具有 65803 个块的文件，但是未经修改的 xv6 会将文件限制为 268 个块。这是因为：xv6 索引节点包含 12 个“直接”块编号和一个“单间接”块编号，这是指最多容纳 256 个以上块编号的块，总共 12 + 256 = 268 块。

所以需要更改 xv6 文件系统代码，以在每个 inode 中支持“双重间接”块，其中包含 256 个单间接块地址，每个间接块最多可以包含 256 个数据块地址。 结果将是一个文件最多可以包含 65803 个块或 256 * 256 + 256 + 11 个块（11 个代替 12 个，因为我们将为双间接块牺牲一个直接块号）。

4.2 实验代码

1.修改 fs.h 中的宏定义以及 dinode 结构，为其添加一个双重间接指针，同时将直接指针-1，addrs[0-10]对应 11 个直接指针，addrs[11]对应间接指针，addrs[12]对应双重间接指针。

```c
#define NDIRECT 11                         //direct pointer 11
#define NINDIRECT1 (BSIZE / sizeof(uint))   //indirect pointer 256
#define NINDIRECT2 (NINDIRECT1 * NINDIRECT1)//double indirect pointer 256*256
#define NINDIRECT (NINDIRECT1 + NINDIRECT2)
#define MAXFILE (NDIRECT + NINDIRECT)       //size of file

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  //addrs[0-10] are direct pointer, addrs[1] is indirect pointer,
  //addrs[12] is double indirect pointer
  uint addrs[NDIRECT+2];   // Data block addresses
};
```

2. 然后修改 fs.c 中的 bmap() 函数。

```c
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){ //bn 仍在直接指针范围内，可以直接访问
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }

  bn -= NDIRECT; //减去直接指针，判断在不在间接指针范围内
  if(bn < NINDIRECT1){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT])  == 0)//访问间接指针，没有就分配一个
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);//访问间接指针指向的指针
    a = (uint*)bp->data;//访问间接指针指向的指针所指的内容
    if((addr = a[bn]) == 0){//为空就分配一个
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  bn -= NINDIRECT1;//减去间接指针，判断在不在双重间接指针范围内
  if(bn < NINDIRECT2){
```

```
    uint bn_1= ( bn & 0xff00)>>8 ;//双重间接指针中的一级指针
    uint bn_2= bn & 0xff;//双重间接指针中的二级指针

    // Load double indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT+1]) == 0)//访问双重间接指针，没有就分配一个
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn_1]) == 0){//访问一级指针
      a[bn_1] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn_2]) == 0){//访问二级指针
      a[bn_2] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    return addr;
  }

  panic("bmap: out of range");
}
```

3. 修改 itrunc 函数，主要功能是释放所有的指针。

```
void
itrunc(struct inode *ip)
{
  int i, j, k;
  struct buf *bp;
  struct buf *bp2;
  uint *a;
  uint *a2;

  for(i = 0; i < NDIRECT; i++){//释放直接指针
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }
```

```

  if(ip->addrs[NDIRECT]){//释放间接指针
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT1; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }

  if(ip->addrs[NDIRECT+1]){//释放双重间接指针
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT1; j++){//释放一级指针
      if(a[j])
      {
    bp2 = bread(ip->dev, a[j]);
        a2 = (uint*)bp2->data;
        for(k=0 ;k < NINDIRECT1 ;k++)//释放二级指针
    {
       if(a2[k])
              bfree(ip->dev,a2[k]);
    }
    brelse(bp2);
        bfree(ip->dev, a[j]);
        a[j]=0;
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
  }

  ip->size = 0;
  iupdate(ip);
}
```

4.3 实验结果

实验结果如图，可以看出已正确输出预期结果。

niandd33@ubuntu: ~/xv6-labs-2020

File Edit View Search Terminal Help

```
xv6 kernel is booting

init: starting sh
$ bigfile
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
......................
wrote 65803 blocks
bigfile done; ok
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
```

……

……

niandd33@ubuntu: ~/xv6-labs-2020

File Edit View Search Terminal Help

```
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6219
            sepc=0x00000000000022ce stval=0x000000000000fb90
OK
test opentest: OK
test writetest: OK
test writebig: 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
$
```