

《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](#_Toc62328110)

[1.1 题目要求 3](#_Toc62328111)

[1.2 实验代码 3](#_Toc62328112)

[1.3 实验结果 5](#_Toc62328113)

[2.Xv6 lab: Lock/Memory allocator 6](#_Toc62328114)

[2.1 题目要求 6](#_Toc62328115)

[2.2 实验代码 6](#_Toc62328116)

[2.3 实验结果 8](#_Toc62328117)

[3.Xv6 lab: Lock/Buffer cache 10](#_Toc62328119)

[3.1 题目要求 10](#_Toc62328120)

[3.2 实验代码 10](#_Toc62328121)

[3.3 实验结果 14](#_Toc62328122)

[4.Xv6 lab: File System/Large files 15](#_Toc62328123)

[4.1 题目要求 15](#_Toc62328124)

[4.2 实验代码 15](#_Toc62328125)

[4.3 实验结果 18](#_Toc62328126)

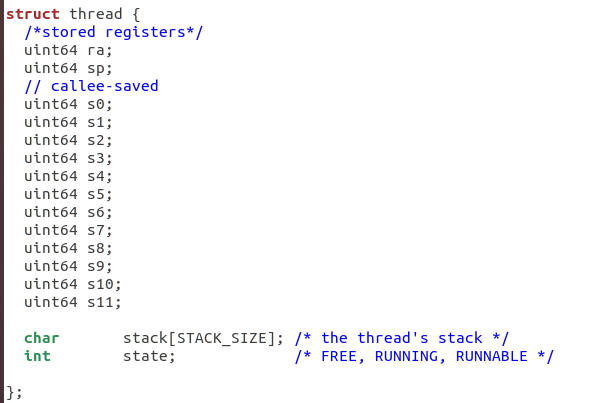
# 1.xv6 Lab: Multithreading/Uthread: switching between threads

## 1.1 题目要求

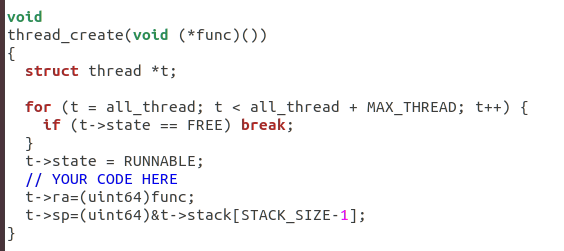
在本练习中，你将为用户级线程系统设计上下文切换机制，然后实现它。首先，你的xv6有两个文件user/uthread.c和user/uthread\_switch.S，以及Makefile中的一个用于构建uthread程序的规则。uthread.c包含大多数用户级线程包，以及三个简单测试线程的代码。 线程包缺少一些用于创建线程和在线程之间切换的代码。你的工作是想出一个创建线程的计划，并保存/恢复寄存器以在线程之间切换，并实施该计划。

## 1.2 实验代码

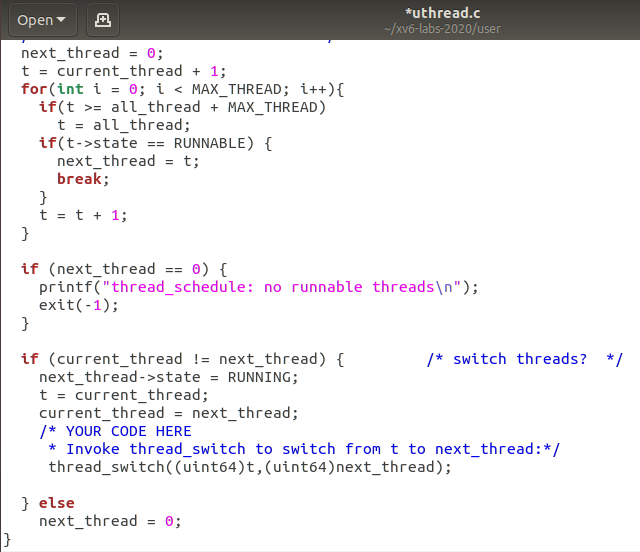
1. 在struct thread添加寄存器字段,能够保存进程内容。其中ra为返回地址，sp为栈顶指针地址。



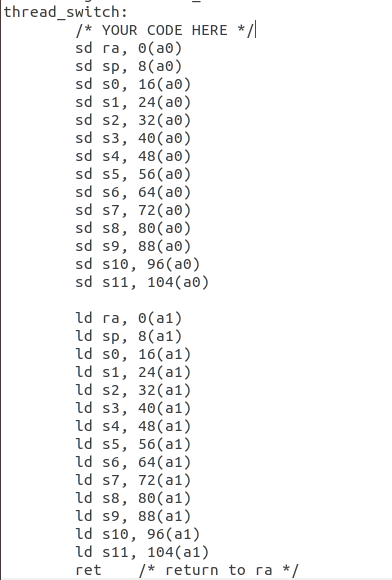
2. 在thread\_create()中添加线程进程入口，设置栈顶指针sp进行用户内存分配。



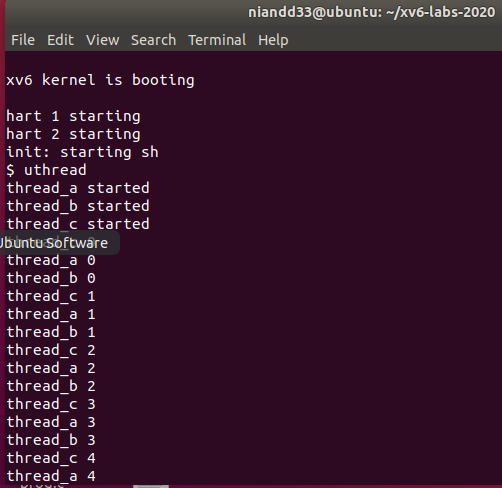
3.在thread\_schedule()中添加调用

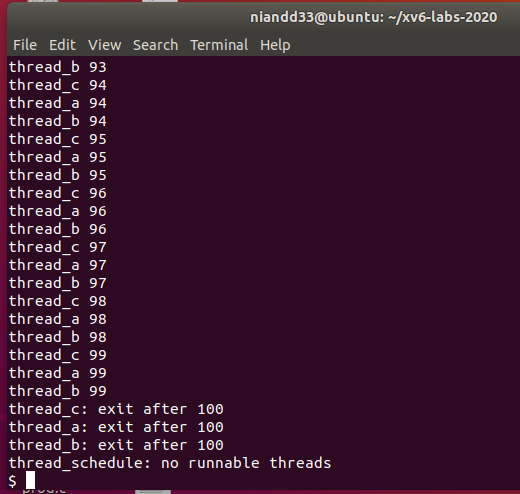


4. 在uthread\_switch.S中添加寄存器



## 1.3 实验结果





# 2.Xv6 lab: Lock/Memory allocator

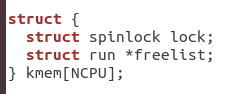
## 2.1 题目要求

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。

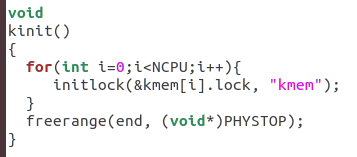
kalloctest中锁争用的根本原因是kalloc（）具有单个空闲列表，并受单个锁保护。要删除锁争用，需要重新设计内存分配器以避免单个锁和列表。基本思想是为每个CPU维护一个空闲列表，每个列表都有自己的锁。不同CPU上的分配和释放可以并行运行，因为每个CPU将在不同的列表上运行。主要的挑战将是处理一个CPU的空闲列表为空，而另一个CPU的列表具有空闲内存的情况。在这种情况下，一个CPU必须“窃取”另一CPU空闲列表的一部分，窃取可能会引入锁争用。

## 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。



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.



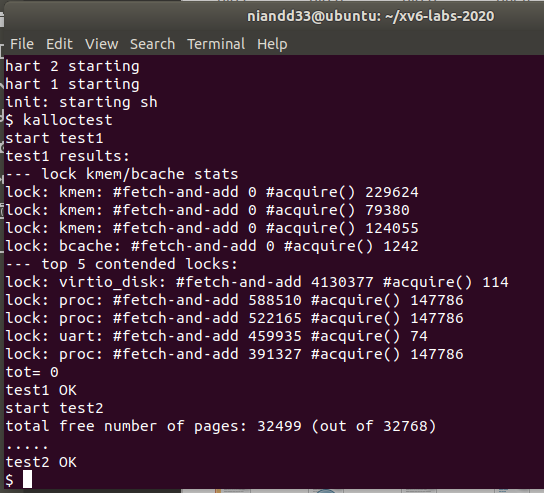
3. 修改kfree函数

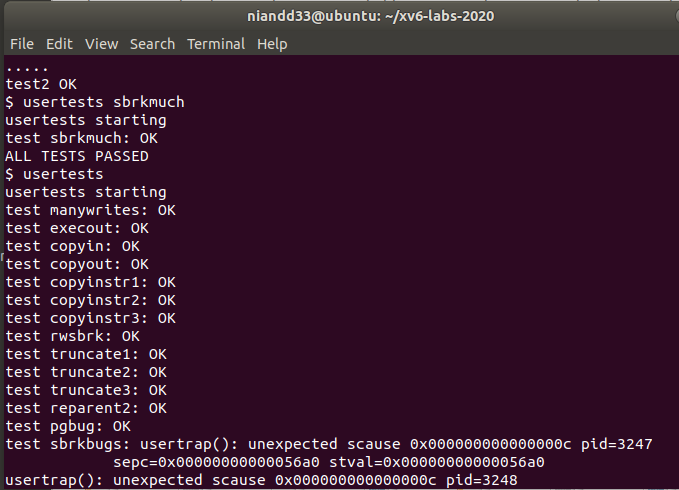
1. **void**
2. kfree(**void** \*pa)
3. {
4. **struct** run \*r;
6. **if**(((uint64)pa % PGSIZE) != 0 || (**char**\*)pa < end || (uint64)pa >= PHYSTOP)
7. panic("kfree");
8. // Fill with junk to catch dangling refs.
9. memset(pa, 1, PGSIZE);
11. r = (**struct** run\*)pa;
13. push\_off();// 关中断
14. **int** i=cpuid();// CPU id
15. acquire(&kmem[i].lock);
16. r->next = kmem[i].freelist;
17. kmem[i].freelist = r;
18. release(&kmem[i].lock);
20. pop\_off();//开中断
21. }

4. 修改kalloc函数

1. **void** \*
2. kalloc(**void**)
3. {
4. **struct** run \*r;
6. push\_off();// 关中断
7. **int** i=cpuid();// CPU id
8. acquire(&kmem[i].lock);
9. r = kmem[i].freelist;
10. **if**(r)
11. kmem[i].freelist = r->next;
12. release(&kmem[i].lock);
14. **if**(!r)//如果当前cpu对应freelist为空
15. {
16. **for**(**int** j=0;j<NCPU;j++)//从其他cpu的freelist中借用,遍历所有CPU
17. {
18. **if**(j!=i)
19. {
20. acquire(&kmem[j].lock);//获取该CPU的锁
21. **if**(kmem[j].freelist)
22. {
23. r=kmem[j].freelist;//获取该CPU的空闲列表
24. kmem[j].freelist=r->next; //该CPU空闲列表-1，给窃取者
25. release(&kmem[j].lock);
26. **break**;
27. }
28. release(&kmem[j].lock);
29. }
30. }
31. }
33. pop\_off();//开中断
35. **if**(r)
36. memset((**char**\*)r, 5, PGSIZE); // fill with junk
37. **return** (**void**\*)r;
38. }

## 2.3 实验结果





## 

# 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。

1. #define NBUCKET 13
2. **struct** {
3. **struct** spinlock lock[NBUCKET];//每个bucket都有一个锁
4. **struct** buf buf[NBUF];
6. // Linked list of all buffers, through prev/next.
7. // Sorted by how recently the buffer was used.
8. // head.next is most recent, head.prev is least.
9. **struct** buf hashbucket[NBUCKET];//哈希表，数据结构为链表
10. } bcache;

13. uint idx(uint blockno)//散列函数
14. {
15. **return** blockno % NBUCKET;
16. }

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。

1. **struct** buf {
2. **int** valid;   // has data been read from disk?
3. **int** disk;    // does disk "own" buf?
4. uint dev;
5. uint blockno;
6. **struct** sleeplock lock;
7. uint refcnt;
8. **struct** buf \*prev; // LRU cache list //最少使用的
9. **struct** buf \*next; // 最近使用的
10. uchar data[BSIZE];
11. uint time\_stamp;//时间戳
12. };

3.修改kernel/bio.c中的binit()函数。

1. **void**
2. binit(**void**)
3. {
4. **struct** buf \*b;
5. **for** (**int** i=0; i<NBUCKETS;i++){
6. initlock(&bcache.lock[i], "bcache");// 将bucket的头节点初始化为自己
7. bcache.hashbucket[i].prev = &bcache.hashbucket[i];
8. bcache.hashbucket[i].next = &bcache.hashbucket[i];
9. }
11. **for**(b = bcache.buf; b < bcache.buf+NBUF; b++){
12. b->time\_stamp = ticks; // 记录一下b的时间戳
13. b->next = bcache.hashbucket[0].next;
14. b->prev = &bcache.hashbucket[0];
15. initsleeplock(&b->lock, "buffer");
16. bcache.hashbucket[0].next->prev = b;
17. bcache.hashbucket[0].next = b;
18. }
19. }

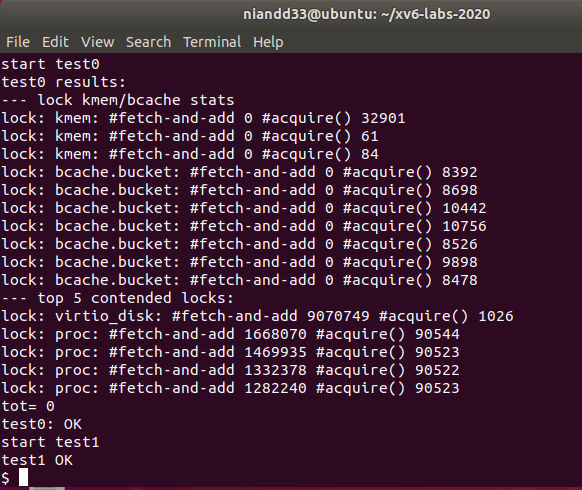
4. 修改bget函数，在哈希表中搜索缓冲区，并在找不到缓冲区时为该缓冲区分配一个条目，这必须是原子的。

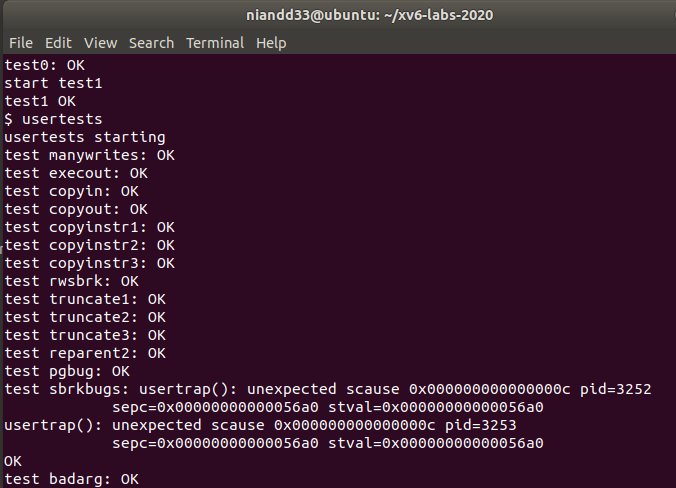
1. **static** **struct** buf\*
2. bget(uint dev, uint blockno)
3. {
4. **struct** buf \*b;
5. **int** i= idx(blockno);//获取当前缓冲区块号的哈希码
6. acquire(&bcache[i].lock);
8. // Is the block already cached?
9. **for**(b = bcache.hashbucket[i].next; b != &bcache.hashbucket[i]; b = b->next){
10. **if**(b->dev==dev && b->blockno == blockno)//hit
11. {
12. b->time\_stamp = ticks; // 记录一下b的时间戳
13. b->refcnt++;
14. release(&bcache.lock[i]);
15. acquiresleep(&b->lock);
16. **return** b;
17. }// 找到最小的时间戳对应buf
19. }
21. // Not cached.如果找不到，就需要从其它bucket中寻找
22. // Recycle the least recently used (LRU) unused buffer.
23. **for** (**int** newnum = idx(blockno+1); newnum != i; newnum = (newnum+1)%NBUCKET){
24. acquire(&bcache.lock[newnum]);
25. // 遍历该bucket的链表
26. **for**(b = bcache.hashbucket[newnum].prev; b != &bcache.hashbucket[newnum]; b = b->prev){
27. **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
28. b->time\_stamp = ticks; // 记录一下b的时间戳
29. b->dev = dev;
30. b->blockno = blockno;
31. b->valid = 0;
32. b->refcnt = 1;
34. // 先移出原来bucket
35. b->next->prev = b->prev;
36. b->prev->next = b->next;
37. release(&bcache.lock[newnum]);
39. //放到需要该缓冲区块的bucket
40. b->next = bcache.hashbucket[i].next;
41. b->prev = &bcache.hashbucket[i];
43. //将该区块连接到当前bucket中
44. bcache.hashbucket[i].next->prev = b;
45. bcache.hashbucket[i].next = b;
46. release(&bcache.lock[i]);
47. acquiresleep(&b->lock);
48. **return** b;
49. }
50. }
51. release(&bcache.lock[newnum]); // 当前bucket中没找到，仍要释放锁
52. }
53. panic("bget: no buffers");
54. }

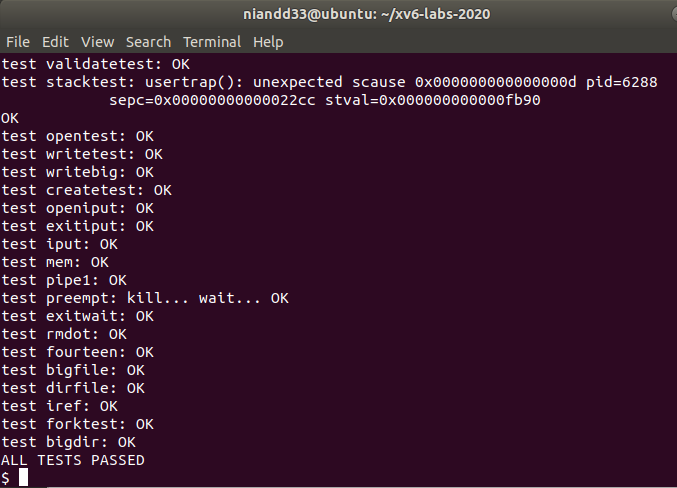
5. 修改brelse，bpin，bunpin 函数。

1. **void**
2. brelse(**struct** buf \*b)
3. {
4. **if**(!holdingsleep(&b->lock))
5. panic("brelse");
6. releasesleep(&b->lock);
8. **int** i=idx(b->blockno);
10. b->time\_stamp = ticks;//用时间戳代替原先的上锁
11. **if**(b->time\_stamp == ticks){
12. b->refcnt--;
13. **if** (b->refcnt == 0) {
14. // no one is waiting for it.
15. b->next->prev = b->prev;
16. b->prev->next = b->next;
17. b->next = bcache.hashbucket[i].next;
18. b->prev = &bcache.hashbucket[i];
19. bcache.hashbucket[i].next->prev = b;
20. bcache.hashbucket[i].next = b;
21. }
22. }
23. }
24. **void**
25. bpin(**struct** buf \*b) {
26. **int** i=idx(b->blockno);
27. acquire(&bcache.lock[i]);
28. b->refcnt++;
29. release(&bcache.lock[i]);
30. }
32. **void**
33. bunpin(**struct** buf \*b) {
34. **int** i=idx(b->blockno);
35. acquire(&bcache.lock[i]);
36. b->refcnt--;
37. release(&bcache.lock[i]);
38. }

## 3.3 实验结果







# 4.Xv6 lab: File System/Large files

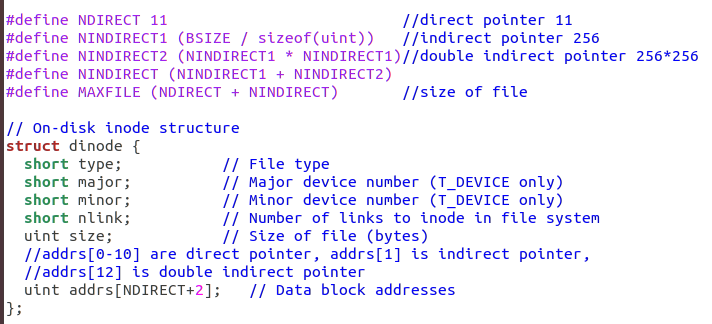
## 4.1 题目要求

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。



2.然后修改fs.c中的bmap()函数。

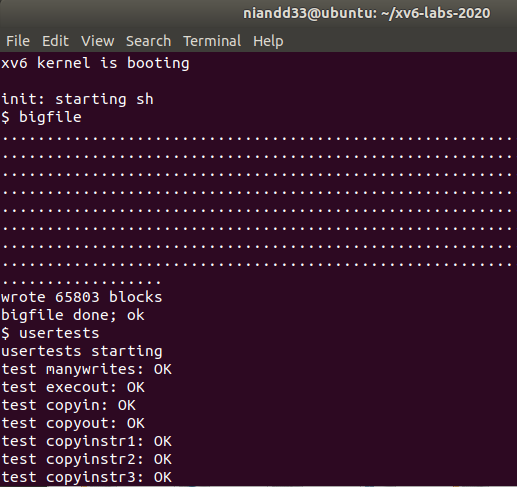
1. **static** uint
2. bmap(**struct** inode \*ip, uint bn)
3. {
4. uint addr, \*a;
5. **struct** buf \*bp;
7. **if**(bn < NDIRECT){ //bn仍在直接指针范围内，可以直接访问
8. **if**((addr = ip->addrs[bn]) == 0)
9. ip->addrs[bn] = addr = balloc(ip->dev);
10. **return** addr;
11. }
13. bn -= NDIRECT; //减去直接指针，判断在不在间接指针范围内
14. **if**(bn < NINDIRECT1){
15. // Load indirect block, allocating if necessary.
16. **if**((addr = ip->addrs[NDIRECT]) == 0)//访问间接指针，没有就分配一个
17. ip->addrs[NDIRECT] = addr = balloc(ip->dev);
18. bp = bread(ip->dev, addr);//访问间接指针指向的指针
19. a = (uint\*)bp->data;//访问间接指针指向的指针所指的内容
20. **if**((addr = a[bn]) == 0){//为空就分配一个
21. a[bn] = addr = balloc(ip->dev);
22. log\_write(bp);
23. }
24. brelse(bp);
25. **return** addr;
26. }
28. bn -= NINDIRECT1;//减去间接指针，判断在不在双重间接指针范围内
29. **if**(bn < NINDIRECT2){
30. uint bn\_1= ( bn & 0xff00)>>8 ;//双重间接指针中的一级指针
31. uint bn\_2= bn & 0xff;//双重间接指针中的二级指针
33. // Load double indirect block, allocating if necessary.
34. **if**((addr = ip->addrs[NDIRECT+1]) == 0)//访问双重间接指针，没有就分配一个
35. ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
36. bp = bread(ip->dev, addr);
37. a = (uint\*)bp->data;
38. **if**((addr = a[bn\_1]) == 0){//访问一级指针
39. a[bn\_1] = addr = balloc(ip->dev);
40. log\_write(bp);
41. }
42. brelse(bp);
44. bp = bread(ip->dev, addr);
45. a = (uint\*)bp->data;
46. **if**((addr = a[bn\_2]) == 0){//访问二级指针
47. a[bn\_2] = addr = balloc(ip->dev);
48. log\_write(bp);
49. }
50. brelse(bp);
52. **return** addr;
53. }
55. panic("bmap: out of range");
56. }

3. 修改 itrunc 函数，主要功能是释放所有的指针。

1. **void**
2. itrunc(**struct** inode \*ip)
3. {
4. **int** i, j, k;
5. **struct** buf \*bp;
6. **struct** buf \*bp2;
7. uint \*a;
8. uint \*a2;
10. **for**(i = 0; i < NDIRECT; i++){//释放直接指针
11. **if**(ip->addrs[i]){
12. bfree(ip->dev, ip->addrs[i]);
13. ip->addrs[i] = 0;
14. }
15. }
17. **if**(ip->addrs[NDIRECT]){//释放间接指针
18. bp = bread(ip->dev, ip->addrs[NDIRECT]);
19. a = (uint\*)bp->data;
20. **for**(j = 0; j < NINDIRECT1; j++){
21. **if**(a[j])
22. bfree(ip->dev, a[j]);
23. }
24. brelse(bp);
25. bfree(ip->dev, ip->addrs[NDIRECT]);
26. ip->addrs[NDIRECT] = 0;
27. }
29. **if**(ip->addrs[NDIRECT+1]){//释放双重间接指针
30. bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
31. a = (uint\*)bp->data;
32. **for**(j = 0; j < NINDIRECT1; j++){//释放一级指针
33. **if**(a[j])
34. {
35. bp2 = bread(ip->dev, a[j]);
36. a2 = (uint\*)bp2->data;
37. **for**(k=0 ;k < NINDIRECT1 ;k++)//释放二级指针
38. {
39. **if**(a2[k])
40. bfree(ip->dev,a2[k]);
41. }
42. brelse(bp2);
43. bfree(ip->dev, a[j]);
44. a[j]=0;
45. }
46. }
47. brelse(bp);
48. bfree(ip->dev, ip->addrs[NDIRECT+1]);
49. ip->addrs[NDIRECT+1] = 0;
50. }
52. ip->size = 0;
53. iupdate(ip);
54. }

## 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。



……

……

