

《操作系统》

期末作业设计报告

院（系）：智能工程学院

专 业：智能科学与技术

学生姓名：江汉康

学 号：18364036

二〇二一年一月

目 录

第 1 章 Lab: Multithread/Uthread

- 1.1 题目分析
- 1.2 实现过程
- 1.3 实现结果

第 2 章 Lab: Lock/Memory allocator

- 2.1 题目分析
- 2.2 实现过程
- 2.3 实现结果

第 3 章 Lab: Lock/Buffer cache

- 3.1 题目分析
- 3.2 实现过程
- 3.3 实现结果

第 4 章 Lab: File System/Large files

- 4.1 题目分析
- 4.2 实现过程
- 4.3 实现结果

第 1 章 lab: Multithreading/Uthread

1.1 题目分析

题目希望我们实现用户线程的上下文切换，我们可以仿照进程切换的过程实现

1.2 实现过程

为了实现线程切换，首先我们要对线程进行定义

线程定义：线程需要栈空间 state context

```
1.  struct context{
2.      //线程的上下文
3.      uint64 ra;
4.      uint64 sp;
5.
6.      // callee-saved
7.      uint64 s0;
8.      uint64 s1;
9.      uint64 s2;
10.     uint64 s3;
11.     uint64 s4;
12.     uint64 s5;
13.     uint64 s6;
14.     uint64 s7;
15.     uint64 s8;
16.     uint64 s9;
17.     uint64 s10;
18.     uint64 s11;
19. };
20.
21. struct thread{
22.     char stack[STACK_SIZE];
23.     int state;
24.     struct context context;
25. };
```

线程创建： 创建一个新线程

```
1.  void
2.  thread_create(void (*func)()){
3.      struct thread *t;
```

```

4.     for(t = all_thread; t < all_thread + MAX_THREAD; t++){//将 t 添加入
        all_thread
5.         if (t->state == FREE) break;//当遇到空 thread 就退出，并指向空
6.     }
7.     t->state = RUNNABLE;//可执行
8.     t->context.ra = (uint64) func;//将函数地址添加到返回地址寄存器
9.     t->context.sp = (uint64) &t->stack + STACK_SIZE;//栈顶指针设置
10. }

```

线程切换选择：通过遍历现存线程，执行第一个找到的 RUNNABLE 线程

```

1. void
2. thread_schedule(void){
3.     struct thread *t,*next_thread;
4.
5.     next_thread = 0;
6.     t = current_thread + 1;//指向当前 thread 的下一个 thread
7.
8.     for(int i = 0; i < MAX_THREAD; i++){//从 current_thread 下一个开始，遍历
        all_thread，找到第一个 RUNNABLE thread
9.         if(t >= all_thread + MAX_THREAD)
10.            t = all_thread;//超过范围，从头开始
11.         if(t->state == RUNNABLE) {
12.             next_thread = t;//确定 next_thread
13.             break;
14.         }
15.         t = t + 1;
16.     }
17.     if(next_thread == 0){//未找到 next_thread
18.         printf("thread_schedule: no runnable threads\n");
19.         exit(-1);
20.     }
21.
22.     if (current_thread != next_thread) {//current_thread = next_thread
23.         next_thread->state = RUNNING;
24.         t = current_thread;
25.         current_thread = next_thread;
26.         thread_switch((uint64) t, (uint64) current_thread);//线程切换
27.     }
28.     else
29.         next_thread = 0;
30. }

```

其中

```
thread_switch((uint64) t, (uint64) current_thread); //线程切换
```

调用了thread_switch.S

上下文切换: thread_switch.S, 基本与kernel/swtch.S相同

```
1.      .text
2.
3.      /*
4.         * save the old thread's registers,
5.         * restore the new thread's registers.
6.         */
7.
8.      .globl thread_switch
9. thread_switch:
10.     sd ra, 0(a0)
11.     sd sp, 8(a0)
12.     sd s0, 16(a0)
13.     sd s1, 24(a0)
14.     sd s2, 32(a0)
15.     sd s3, 40(a0)
16.     sd s4, 48(a0)
17.     sd s5, 56(a0)
18.     sd s6, 64(a0)
19.     sd s7, 72(a0)
20.     sd s8, 80(a0)
21.     sd s9, 88(a0)
22.     sd s10, 96(a0)
23.     sd s11, 104(a0)
24.
25.     ld ra, 0(a1)
26.     ld sp, 8(a1)
27.     ld s0, 16(a1)
28.     ld s1, 24(a1)
29.     ld s2, 32(a1)
30.     ld s3, 40(a1)
31.     ld s4, 48(a1)
32.     ld s5, 56(a1)
33.     ld s6, 64(a1)
34.     ld s7, 72(a1)
35.     ld s8, 80(a1)
36.     ld s9, 88(a1)
37.     ld s10, 96(a1)
38.     ld s11, 104(a1)
```

```
39.  
40.         ret
```

主函数实现：为了达到输出效果，需要创建三个新线程 *thread_a thread_b thread_c*，以 *cab* 的顺序循环执行线程

```
1. int  
2. main(int argc, char *argv[])  
3. {  
4.     a_started = b_started = c_started = 0;  
5.     a_n = b_n = c_n = 0;  
6.     thread_init();//将 main 设为当前线程  
7.     //创建线程  
8.     thread_create(thread_a);  
9.     thread_create(thread_b);  
10.    thread_create(thread_c);  
11.    //schedule  
12.    thread_schedule();  
13.    exit(0);  
14. }
```

在 *main* 函数中，我们要创建三个线程，三个线程所做的事都相同，这里我们以 *thread a* 为例

在 *thread* 中，每执行一次，*a_n* 就加一，然后立刻放弃 CPU，进入 *thread_schedule*

```
1. void  
2. thread_a(){  
3.     int i;  
4.     printf("thread_a started\n");  
5.     a_started = 1;//a 已经 start  
6.     while(b_started == 0 || c_started == 0)  
7.         thread_yield();//thread_a 放弃 CPU，进入 thread_schedule  
8.  
9.     for (i = 0; i < 100; i++) {  
10.        printf("thread_a %d\n", i);  
11.        a_n += 1;  
12.        thread_yield();  
13.    }  
14.    printf("thread_a: exit after %d\n", a_n);  
15.  
16.    current_thread->state = FREE;  
17.    thread_schedule();  
18. }
```

在`thread_a` 中调用了`thread_yield` 函数，该函数用于线程放弃 CPU，进入`thread_schedule`

`thread_schedule`:线程放弃 CPU

```
1. void
2. thread_yield(void)
3. {
4.     current_thread->state = RUNNABLE;
5.     thread_schedule();
6. }
```

1.3 实现结果

```
hart 2 starting
hart 1 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
```

```
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
$
```

第 2 章 lab: Lock/Memory allocator

2.1 题目分析

在题目中，用`kalloctest`说明了 xv6 代码的不足之处：

即多个 CPU 在共用内存的同时，共用了内存链表，当 CPU 申请内存时，调用`kalloc()`，从`kalloc()`我们可以发现，它们访问的都是同一个内存链表，而为了实现同步，在访问链表前需要获取链表的 lock，当其他 CPU 占用时，当前 CPU 需要等待 lock 释放

```
1. void *
2. kalloc(void)
3. {
4.     struct run *r;
5.
6.     acquire(&kmem.lock); // 获取链表锁
7.     r = kmem.freelist;
8.     if(r)
9.         kmem.freelist = r->next; // 加入链表，并前移表头，这说明越晚分配的 page，越早调用
10.    release(&kmem.lock); // 释放
11.
12.    if(r)
13.        memset((char*)r, 5, PGSIZE); // fill with junk
14.    return (void*)r;
15. }
```

从`kalloctest`打印信息也可以发现，CPU 之间对内存链表的竞争非常激烈

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: bcache: #fetch-and-add 0 #acquire() 1260
--- top 5 contended locks:
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: proc: #fetch-and-add 23737 #acquire() 130718
lock: virtio_disk: #fetch-and-add 11159 #acquire() 114
lock: proc: #fetch-and-add 5937 #acquire() 130786
lock: proc: #fetch-and-add 4080 #acquire() 130786
tot= 83375
test1 FAIL
```

解决这个问题的核心想法是：题目提示，我们可以为每一个 CPU 都分配一个独立内存链表，当链表又空闲内存时就直接访问自己的链表，无需共用，当某个 CPU 内存链表内存不足时，从另外一个 CPU 的空闲内存窃取一部分内存，而窃取内存这一部分才是这个题目的难点所在

2.2 实现过程

重新定义内存链表`kmem`：每个链表属于每个 CPU

```
1.  struct kmem_t {
2.      struct spinlock lock;
3.      struct run *freelist;
4.  } kmem[NCPU];
```

初始化 `lock` 和链表：

这里因为只有 0 号 CPU 才会调用`kinit()`，所以初始时是将所有内存都分配给了 0 号 CPU，而其他 CPU 则通过窃取来获得空闲内存，但这并不意味着多个内存链表就失去了意义，内存的使用是一个动态的过程，当其他 CPU 申请多个 PAGE 后，只要这个 PAGE 不被窃取，这个链表就不会被频繁的占用 `lock`，导致 CPU 要为链表锁等待，而当 CPU 释放一个 PAGE 后，其他 CPU 也有可能利用到这一 PAGE，不会导致内存的浪费。

```
1.  void
2.  kinit()
3.  {
4.      //只有 CPU=0 才调用
5.      for (int i = 0; i < NCPU; i++)
6.          initlock(&kmem[i].lock, "kmem");//初始化 lock
7.      freerange(end, (void*)PHYSTOP);//将所有内存都分配个第一个调用它的 CPU0
8.  }
```

`freerange`函数与原来相同，需要改变的是`kfree`函数：我们要做的只是为 `kmem[0]`分配全部内存

```
1.      void
2.  kfree(void *pa)
3.  {
4.      struct run *r;
5.
6.      if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
7.          panic("kfree");
8.
9.
10.     // Fill with junk to catch dangling refs.
11.     memset(pa, 1, PGSIZE);
12.
13.     r = (struct run*)pa;
14.
15.     push_off();
```

```

16. int hart = cpuid();//安全获取 CPUid
17. pop_off();
18. struct kmem_t* k = &kmem[hart];
19. acquire(&k->lock);
20. r->next = k->freelist;
21. k->freelist = r;
22. release(&k->lock);
23. }

```

*kalloc()*实现当 CPU 链表不为空时，分配页，为空时，遍历每一个 CPU 的链表，这里需要先获取其他 CPU 链表的 lock 才能访问，然后再“窃取”空闲页，这里本来应该再处理无内存窃取的情况，但想一想这情况又涉及到内存管理，前面的 Lab 也做过类似的题目，也就没有进行优化

```

1. void *
2. kalloc(void)
3. {
4.     struct run *r;
5.
6.     push_off();
7.     int hart = cpuid();
8.     pop_off();
9.     struct kmem_t* k = &kmem[hart];
10.
11.     acquire(&k->lock);
12.     r = k->freelist;
13.     if(r)//当前 CPU 链表有空闲内存
14.         k->freelist = r->next;//更新
15.     release(&k->lock);
16.
17.     if (!r) { //当前 CPU 链表无空闲内存
18.         for (int i = 0; i < NCPU; i++) { //遍历 kmem[i]
19.             struct kmem_t* k = &kmem[i];
20.             acquire(&k->lock); //首先获得 lock
21.             r = k->freelist;
22.             if(r) { //查看是否有空闲内存
23.                 k->freelist = r->next; //更新
24.                 release(&k->lock);
25.                 break;
26.             }
27.             release(&k->lock);
28.         }
29.     }
30.
31.     if(r)

```

```
32.     memset((char*)r, 5, PGSIZE); // fill with junk
33.
34.     return (void*)r};
```

2.3 实现结果

```
init: starting sh
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 173462
lock: kmem: #fetch-and-add 0 #acquire() 132041
lock: kmem: #fetch-and-add 0 #acquire() 127577
lock: bcache: #fetch-and-add 0 #acquire() 334
--- top 5 contended locks:
lock: proc: #fetch-and-add 941641 #acquire() 104311
lock: proc: #fetch-and-add 859931 #acquire() 104330
lock: proc: #fetch-and-add 611012 #acquire() 104312
lock: proc: #fetch-and-add 533846 #acquire() 104311
lock: proc: #fetch-and-add 487601 #acquire() 104311
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 bigdir: OK
ALL TESTS PASSED
```

第 3 章 Lock/Buffer cache

3.1 题目分析

在题目中，用 `bcachetest` 程序展现了原操作系统在 `bcache.lock` 使用时的不足，当多个 CPU 密集使用 `file system` 时，需要先申请 `bcache.lock`，这会造成 lock 的激烈竞争，多个 CPU 处于 `acquire loop` 中，降低 CPU 的效率。

为解决这个问题，题目提到我们可以参考前面的 `kalloc` 的实现，但我们无法像前面一样为每一个 CPU 都分配自己的磁盘缓冲区链表，因为缓冲区本身较大，并且多个 CPU 可能会同时访问一块缓冲区

题目提示我们可以用 `hash table` 实现查找，同时为了解决 `hash table` 的冲突问题，采用哈希桶，将缓冲区分为多个 13 哈希桶，并且为每一个哈希桶都分配 lock，这样只有当 CPU 访问同一个桶时才会发生冲突，以减少 `acquire loop` 的可能性

该题目较难实现的是哈希桶找不到缓冲区时，如何分配一个条目，并且当缓冲区满时如何利用 LRU 策略，在 13 个哈希桶中找到换出的 block。

3.2 实现过程

定义哈希表和哈希桶

```
1. struct {
2.     struct spinlock lock;
3.     struct buf buf[NBUF];           // block[30]
4.     //双向链表
5.     struct buf buckets[NBUKETS];
6.     struct spinlock bucketslock[NBUKETS];
7.
8. } bcache;
```

初始化

对每一个哈希桶的 lock 进行初始化，并且创建和初始化哈希表

```
1. void
2. binit(void)
3. {
4.     struct buf *b;
5.     initlock(&bcache.lock, "bcache");
6.     for (int i = 0; i < NBUKETS; i++) //为每个桶的 lock 初始化
7.     {
8.         initlock(&bcache.bucketslock[i], "bcache.bucket");
9.         bcache.buckets[i].prev = &bcache.buckets[i];
10.        bcache.buckets[i].next = &bcache.buckets[i];

```

```

11. }
12.
13. for (b = bcache.buf; b < bcache.buf + NBUF; b++)
14. {
15.     int hash = getHb(b); //获取 hash key (取模)
16.     b->time_stamp = ticks; //为 buf 添加时间戳
17.     b->next = bcache.buckets[hash].next;
18.     b->prev = &bcache.buckets[hash];
19.     initsleeplock(&b->lock, "buffer");
20.     bcache.buckets[hash].next->prev = b;
21.     bcache.buckets[hash].next = b;
22. }
23. }

```

哈希表查找

为了实现 LRU 策略，找出换出的 block，由于 *bcache* 是一个 LRU list，所以我们找到最近的 $r \rightarrow refcnt == 0$ ，就能确定该 block 是 least recently used

```

1. static struct buf*
2. bget(uint dev, uint blockno)
3. {
4.     int hash = getH(blockno); //获取 hash key
5.     struct buf *b;
6.
7.     acquire(&bcache.bucketslock[hash]); //获取对应桶的 lock
8.     //查找 hash table
9.     for(b = bcache.buckets[hash].next; b != &bcache.buckets[hash]; b = b->next)
10.    ){
11.        if(b->dev == dev && b->blockno == blockno){
12.            b->time_stamp = ticks; //更新时间戳，在 brelease 中使用
13.            b->refcnt++;
14.            release(&bcache.bucketslock[hash]);
15.            acquiresleep(&b->lock);
16.            return b;
17.        }
18.
19.        //当前缓冲区无目的 block，根据 LRU 策略，选择换出 block
20.        for (int i = 0; i < NBUKETS; i++)
21.        {
22.            if(i != hash){
23.                acquire(&bcache.bucketslock[i]);
24.                //遍历每一个哈希桶

```

```

25.     for(b = bcache.buckets[i].prev; b != &bcache.buckets[i]; b = b->prev){
26.         if(b->refcnt == 0){
27.             b->time_stamp = ticks;
28.             b->dev = dev;
29.             b->blockno = blockno;
30.             b->valid = 0;    //表示不在 memory 中，需要从 disk 调出
31.             b->refcnt = 1;
32.
33.             /** 将 b 脱出 */
34.             b->next->prev = b->prev;
35.             b->prev->next = b->next;
36.
37.             /** 将 b 接入 */
38.             b->next = bcache.buckets[hash].next;
39.             b->prev = &bcache.buckets[hash];
40.             bcache.buckets[hash].next->prev = b;
41.             bcache.buckets[hash].next = b;
42.             release(&bcache.bucketslock[i]);
43.             release(&bcache.bucketslock[hash]);
44.             acquiresleep(&b->lock);
45.             return b;
46.         }
47.     }
48.     release(&bcache.bucketslock[i]);
49. }
50. }
51. panic("bget: no buffers");
52. }

```

释放 block

在**brelease**中，为了避免出现 CPU 同时调用**brelease**，对同一个 block 释放两次，根据题目提示，运用时间轴进行判断哪个 CPU 获得 block 控制权

```

1. void
2. brelease(struct buf *b)
3. {
4.
5.     if(!holdingsleep(&b->lock))
6.         panic("brelease");
7.
8.     releasesleep(&b->lock);
9.
10.    int blockno = getHb(b); //获取 key
11.    //防止两个 CPU 同时想要释放同一个 block

```

```

12. //通过时间戳能够有效避免两次释放同一个 block
13. //所以这里没有使用到 lock
14. b->time_stamp = ticks;
15. if(b->time_stamp == ticks){
16.     b->refcnt--;
17.     if(b->refcnt == 0){
18.         /** 将 b 脱出 */
19.         b->next->prev = b->prev;
20.         b->prev->next = b->next;
21.
22.         /** 将 b 接入 */
23.         b->next = bcache.buckets[blockno].next;
24.         b->prev = &bcache.buckets[blockno];
25.         bcache.buckets[blockno].next->prev = b;
26.         bcache.buckets[blockno].next = b;
27.     }
28. }
29. }

```

3.3 实现结果

bcachetest

```

$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32982
lock: kmem: #fetch-and-add 0 #acquire() 59
lock: kmem: #fetch-and-add 0 #acquire() 39
lock: bcache: #fetch-and-add 0 #acquire() 2481
lock: bcache: #fetch-and-add 0 #acquire() 1448
lock: bcache: #fetch-and-add 0 #acquire() 2496
lock: bcache: #fetch-and-add 0 #acquire() 2170
lock: bcache: #fetch-and-add 0 #acquire() 3179
lock: bcache: #fetch-and-add 0 #acquire() 3176
lock: bcache: #fetch-and-add 0 #acquire() 3392
lock: bcache: #fetch-and-add 0 #acquire() 3219
lock: bcache: #fetch-and-add 0 #acquire() 4781
lock: bcache: #fetch-and-add 0 #acquire() 3601
lock: bcache: #fetch-and-add 0 #acquire() 2736
lock: bcache: #fetch-and-add 0 #acquire() 2475
lock: bcache: #fetch-and-add 0 #acquire() 1474
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 191397 #acquire() 1378
lock: proc: #fetch-and-add 113049 #acquire() 87151
lock: proc: #fetch-and-add 56438 #acquire() 86747
lock: proc: #fetch-and-add 54043 #acquire() 86726
lock: proc: #fetch-and-add 48809 #acquire() 86726
tot= 0
test0: OK
start test1
test1 OK

```

usertests

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
```

```
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$ █
```


第 4 章 lab: File System/Large files

4.1 题目分析

题目主要让我们增加 xv6 的文件最大大小，在原 xv6 系统中，因为 *inode* 结构只有两种映射（12 个直接映射编号和 1 个块存储的间接编号），两种映射关系使得文件大小限制在 $(12+256=)$ 268 个 block (Block size = 1024byte)，当 *bigfile* 命令运行时，希望创建大于限制的最大大小文件时就产生了错误。

顺着这个思路，为了增大文件最大大小，我们只需要为 *inode* 增加一个双层间接映射便能增大限制大小，其中双层间接映射每层包括 256 个编号地址，另外取直接映射中的一个编号作为加入的第二层间接映射地址存储的块的编号。

这样一个 *inode* 就有直接映射，单层间接映射，双层间接映射三种映射关系，最终限制大小为 $(256*256+256+11=)$ 65803 个 block。

我们要做的是添加双层映射：

4.2 实现过程

修改 *file system* 的宏定义

修改 *fs.h*

提示我们在 *fs.h* 文件定义了 *struct dinode*，因此我们第一步应该修改 *dinode*，在结构种添加双重间接指针并修改直接映射指针编号个数为 11，同时更改最大文件大小。最终根据题目提示，*addrs[0:10]* 对应直接映射，*addrs[11]* 对应间接指针，*addrs[12]* 对应双重映射指针。

```
1. //修改 define
2. #define NDIRECT 11 //直接映射减一
3. #define SINGLEDIRECT (BSIZE / sizeof(uint))//间接映射指针
4. #define NINDIRECT (SINGLEDIRECT + SINGLEDIRECT * BSIZE / sizeof(uint))//双重
   间接指针
5. #define MAXFILE (NDIRECT + NINDIRECT)//限制的最大文件大小
6.
7.
8. // On-disk inode structure
9. struct dinode {
10.     short type;           // File type
11.     short major;          // Major device number (T_DEVICE only)
12.     short minor;          // Minor device number (T_DEVICE only)
13.     short nlink;          // Number of links to inode in file system
14.     uint size;            // Size of file (bytes)
15.     //增加双重间接指针后，地址对应的编号也要修改
16.     //addrs[0:10]对应直接映射，addrs[11]对应间接指针，addrs[12]对应双重映射指针
17.     uint addrs[NDIRECT+1]; // Data block addresses
18. };
```

修改param.h

修改最大文件大小

```
1. #define FSSIZE      200000 // size of file system in blocks
```

修改 block 与编号的地址的映射关系:

修改bmap()函数:

bmap()的功能是将文件的逻辑数据好映射到磁盘数据号中。

其中，直接映射和单层间接映射原 xv6 系统已经帮我们实现，我们要做的是当 block number 处于双重映射地址范围时，提供映射功能

需要注意的是，题目提示我们，每一次调用bread()后都要释放 block:brelse()

```
1. static uint
2. bmap(struct inode *ip, uint bn)//bn 表示 block number, 表示文件中 block 的序号
3. {
4.     uint addr, *a, *a2;
5.     struct buf *bp, *bp2;
6.     if(bn < NDIRECT){
7.         //当 bn<NDIRECT(11)时, 分配直接映射地址
8.         if((addr = ip->addrs[bn]) == 0)
9.             ip->addrs[bn] = addr = balloc(ip->dev);
10.        return addr;
11.    }
12.    bn -= NDIRECT;    //若不是直接映射, 减去 NDIRECT, 进入是否间接映射判断
13.    if(bn < NINDIRECT){
14.        //属于单层间接映射范围
15.        if(bn < SINGLEDIRECT){
16.            if((addr = ip->addrs[NDIRECT]) == 0)
17.                ip->addrs[NDIRECT] = addr = balloc(ip->dev);
18.            bp = bread(ip->dev, addr);
19.            a = (uint*)bp->data;
20.            if((addr = a[bn]) == 0){
21.                a[bn] = addr = balloc(ip->dev);
22.                log_write(bp);
23.            }
24.            brelse(bp);
25.        }
26.        else
27.        {
28.            //属于双层映射范围
29.            uint bn_level_1 = bn/NINDIRECT; // 分别表示两层映射指针
30.            uint bn_level_2 = bn%NINDIRECT;
31.            if((addr=ip->addrs[NDIRECT+1])==0) // 首先访问双层间接指针地址, 查看是否是已
                创建的空闲地址, 若不是则分配一个
32.                ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
```

```

33.     bp = bread(ip->dev, addr);
34.     a = (uint*)bp->data;
35.     if((addr = a[bn_level_1])==0){ // 访问一级指针
36.         a[bn_level_1] = addr = balloc(ip->dev);
37.         log_write(bp);
38.     }
39.     brelse(bp); // 每次 bread 都要 brelse
40.     bp2 = bread(ip->dev, addr); // 访问二级指针
41.     a2 = (uint*)bp2->data;
42.     if ((addr = a2[bn_level_2])==0){
43.         a2[bn_level_2] = addr = balloc(ip->dev);
44.         log_write(bp2);
45.     }
46.     brelse(bp2);
47.     return addr;
48. }
49.
50. panic("bmap: out of range");
51. }

```

修改释放所有文件块代码: *itrunc()*

题目提示我们只需要修改*itrunc()*函数就能够实现
同样我们只需要增加双层间接映射的块行

```

1. static void
2. itrunc(struct inode *ip)
3. {
4.     int i, j;
5.     struct buf *bp, *bp2;
6.     uint *a, *a2;
7.     for(i = 0; i < NDIRECT; i++){
8.         //释放直接映射的 block
9.         if(ip->addrs[i]){
10.            bfree(ip->dev, ip->addrs[i]);
11.            ip->addrs[i] = 0; //表示指针指向空
12.        }
13.    }
14.    if(ip->addrs[NDIRECT]){
15.        //释放单层间接映射的 block
16.        bp = bread(ip->dev, ip->addrs[NDIRECT]);
17.        a = (uint*)bp->data;
18.        for (j = 0; j < SINGLEDIRECT; j++)
19.        {
20.            if(a[j])

```

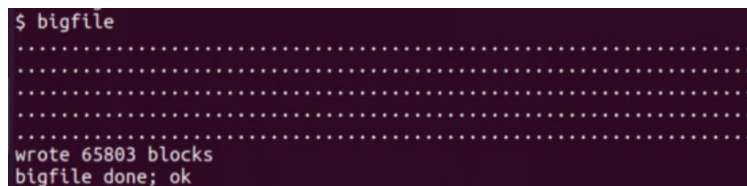
```

21.         bfree(ip->dev, a[j]);
22.     }
23.
24.     brelse(bp);
25.     bfree(ip->dev, ip->addrs[NDIRECT]);
26.     ip->addrs[NDIRECT] = 0;
27. }
28. if(ip->addrs[NDIRECT + 1]){
29.     //释放双层映射的 block
30.     int pos = NDIRECT + 1;
31.     bp = bread(ip->dev, ip->addrs[pos]); //获得指针
32.     a = (uint*)bp->data;
33.     for (i = 0; i < NINDIRECT; i++)
34.     { //第一层
35.         if(a[i]){
36.             bp2 = bread(ip->dev, a[i]);
37.             a2 = (uint *)bp2->data;
38.             for (j = 0; j < SINGLEDIRECT; j++)
39.             { //第二层
40.                 if(a2[j])
41.                     bfree(ip->dev, a2[j]);
42.             }
43.             brelse(bp2);
44.             bfree(ip->dev, a[i]);
45.             a[i] = 0;
46.         }
47.     }
48.     brelse(bp);
49.     bfree(ip->dev, ip->addrs[pos]);
50.     ip->addrs[pos] = 0;
51. }
52. ip->size = 0;
53. iupdate(ip);
54. }

```

4.3 实现结果

bigfile



```

$ bigfile
.....
.....
.....
.....
.....
wrote 65803 blocks
bigfile done; ok

```

usertests

```
bigfile done; ok  
$ usertests  
usertests starting  
test manywrites: OK  
test execout: OK  
test copyin: OK  
test copyout: OK
```

```
test bigdir: OK  
ALL TESTS PASSED  
$ █
```