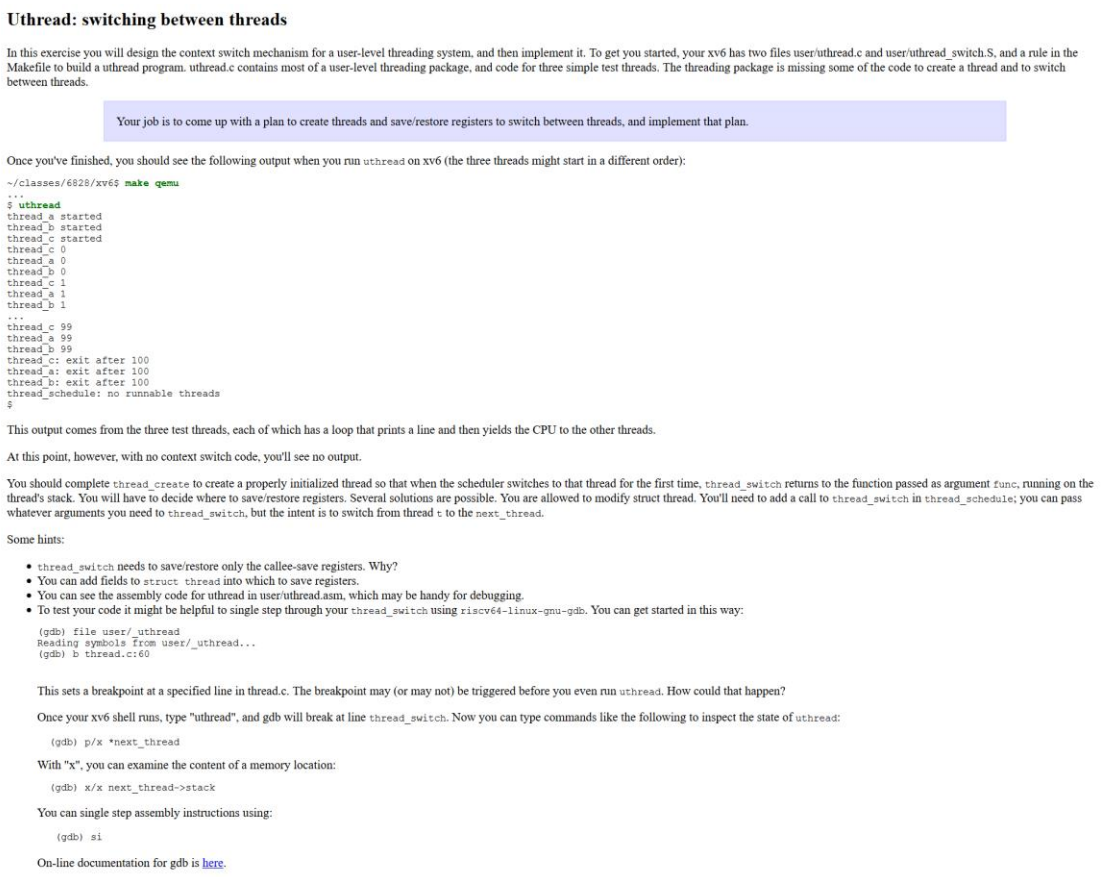
# 操作系统第三次大作业

张灿晖 18364114 智科 2 班

## 1 Uthread: switching between threads

### 1.1 题目截图



**Uthread: switching between threads**

In this exercise you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files user/uthread.c and user/uthread_switch.S, and a rule in the Makefile to build a uthread program. uthread.c contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.

Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan.

Once you've finished, you should see the following output when you run `uthread` on xv6 (the three threads might start in a different order):

```
~/classes/6828/xv6$ make qemu
...
$ 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 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
$
```

This output comes from the three test threads, each of which has a loop that prints a line and then yields the CPU to the other threads.

At this point, however, with no context switch code, you'll see no output.

You should complete `thread_create` to create a properly initialized thread so that when the scheduler switches to that thread for the first time, `thread_switch` returns to the function passed as argument `func`, running on the thread's stack. You will have to decide where to save/restore registers. Several solutions are possible. You are allowed to modify struct thread. You'll need to add a call to `thread_switch` in `thread_schedule`; you can pass whatever arguments you need to `thread_switch`, but the intent is to switch from thread `t` to the `next_thread`.

Some hints:

- `thread_switch` needs to save/restore only the callee-save registers. Why?
- You can add fields to `struct thread` into which to save registers.
- You can see the assembly code for uthread in user/uthread.asm, which may be handy for debugging.
- To test your code it might be helpful to single step through your `thread_switch` using `riscv64-linux-gnu-gdb`. You can get started in this way:

  ```
  (gdb) file user/_uthread
  Reading symbols from user/_uthread...
  (gdb) b thread.c:60
  ```

  This sets a breakpoint at a specified line in thread.c. The breakpoint may (or may not) be triggered before you even run `uthread`. How could that happen?

  Once your xv6 shell runs, type "uthread", and gdb will break at line `thread_switch`. Now you can type commands like the following to inspect the state of `uthread`:

  ```
    (gdb) p/x *next_thread
  ```

  With "x", you can examine the content of a memory location:

  ```
    (gdb) x/x next_thread->stack
  ```

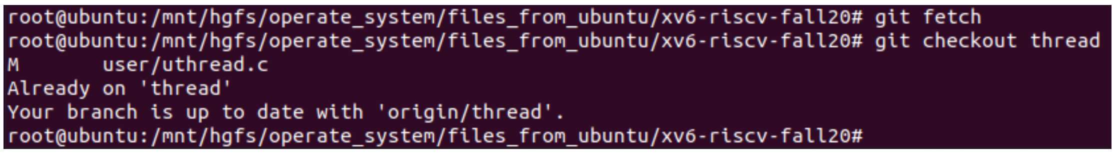
  You can single step assembly instructions using:

  ```
     (gdb) si
  ```

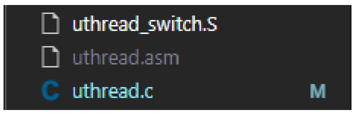
  On-line documentation for gdb is here.

### 1.2 实验步骤

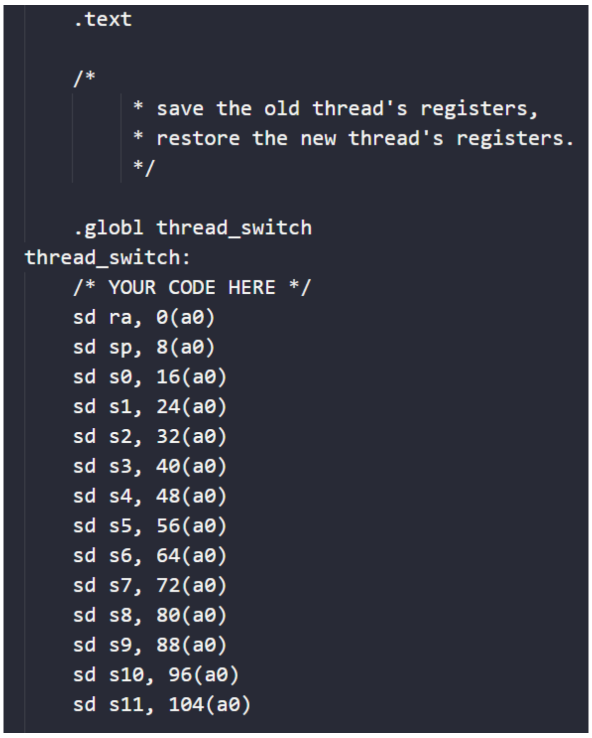
先依次输入代码 git fetch 和 git checkout thread 切换到 thread 分支，随后会发现 xv6 文件夹中多出两个文件，即 user/uthread.c 和 user/uthread_switch.S，我们直接在这两个文件中对代码进行修改即可。



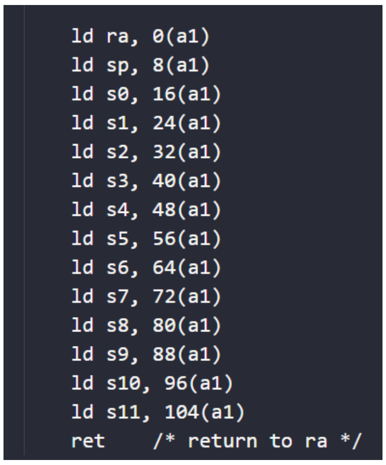
```
root@ubuntu:/mnt/hgfs/operate_system/files_from_ubuntu/xv6-riscv-fall20# git fetch
root@ubuntu:/mnt/hgfs/operate_system/files_from_ubuntu/xv6-riscv-fall20# git checkout thread
M       user/uthread.c
Already on 'thread'
Your branch is up to date with 'origin/thread'.
root@ubuntu:/mnt/hgfs/operate_system/files_from_ubuntu/xv6-riscv-fall20#
```



先打开文件 uthread_switch.S，观察其内容并将其与文件 kernel/switch.S 对比，发现两者内容基本一致，同时对其内容（已在下两图中展示）进行分析，发现其主要完成了一系列寄存器的换出与换入，于是我们可以判断该文件实现的主要功能为协程的切换。



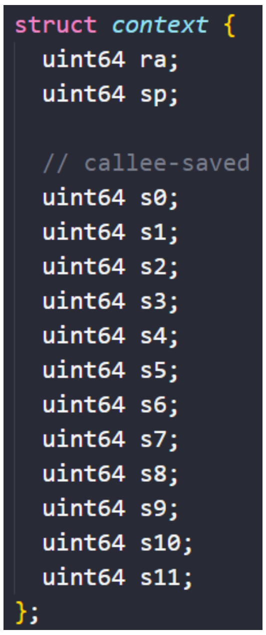
```
    .text

    /*
         * save the old thread's registers,
         * restore the new thread's registers.
         */

    .globl thread_switch
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)
```



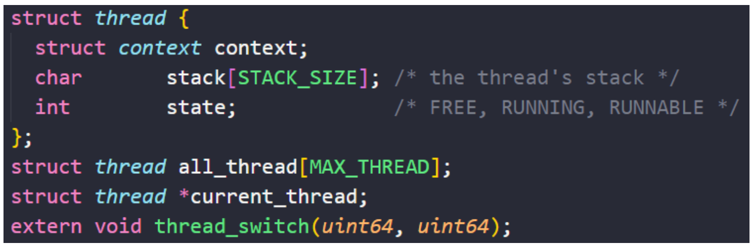
```
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)
    ret     /* return to ra */
```

随后，我们对文件 uthread.c 进行分析。我们看到，在文件的开头定义了结构体 context，该结构体表示进程的上下文。紧接着定义了结构体 thread，该结构体即表示单个线程，含有三个数据域，分别为表征线程上下文的结构体 context，存储内容的字符数组 stack，表示线程当前状态的整形变量 state。

定义了这两个结构体后，文件定义了结构体数组 all_thread 存储所有线程；定义了结构体指针 current_thread 指向当前线程；声明了外部函数 thread_switch，该函数完成线程的切换。



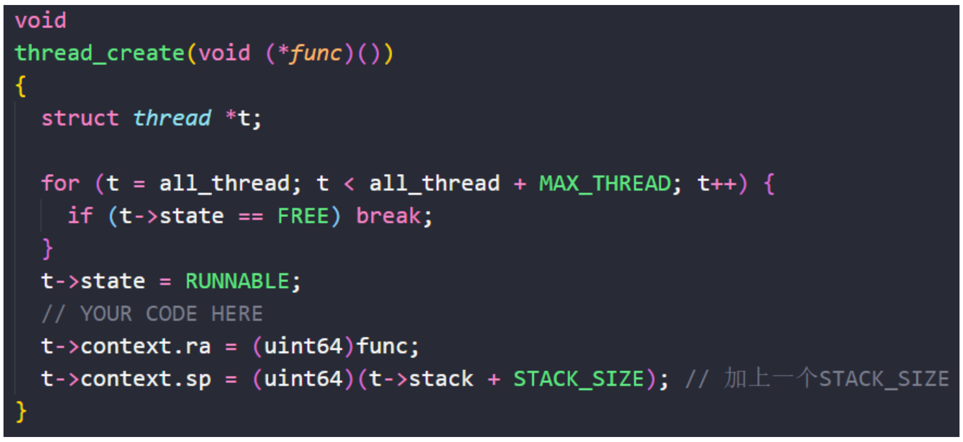
```
struct context {
  uint64 ra;
  uint64 sp;

  // 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;
};
```



```
struct thread {
  struct context context;
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
extern void thread_switch(uint64, uint64);
```

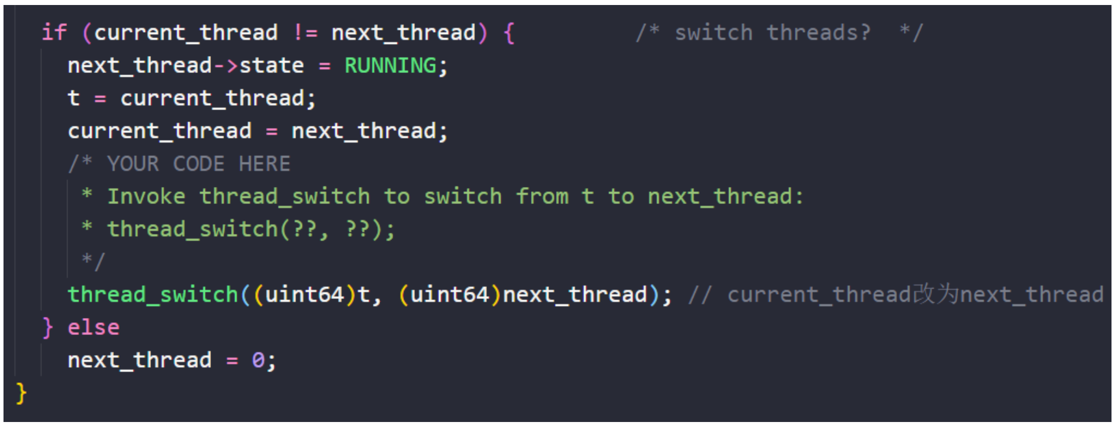
接下来，我们对代码进行修改，以完成实验要求的功能。先分析函数 thread_create，该函数首先通过 for 循环在线程池中找到一个状态为 FREE，即空闲线程后跳出循环，将该线程的状态设为 RUNNABLE，即可执行状态。随后，由于需要记录当线程结束时需要返回哪个地址，我们将 t->context.ra 返回地址项设为函数 thread_create 传入的参数，即指向某个特定函数入口的函数指针 func；将 t->context.sp 栈顶指针项设为 t->stack+STACK_SIZE，

即结构 thread 中合法的最高地址，也就是栈底。



```c
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->context.ra = (uint64)func;
  t->context.sp = (uint64)(t->stack + STACK_SIZE); // 加上一个STACK_SIZE
}
```

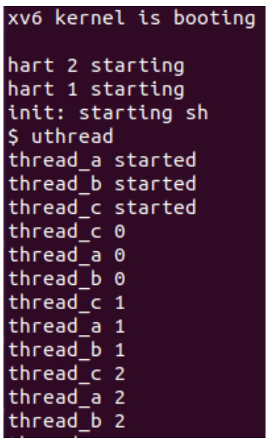
随后，我们在函数 thread_schedule 中更改函数 thread_switch 的参数，以完成协程切换，即下图中的 thread_switch((uint64)t, (uint64)next_thread)，由于 t 表示当前线程，next_thread 线程。因此我们很自然地将 next_thread 作为该函数的第二个参数传入（其实这里不改直接传 current_thread 也可以，毕竟之前已将 next_thread 的值赋与 current_thread)。



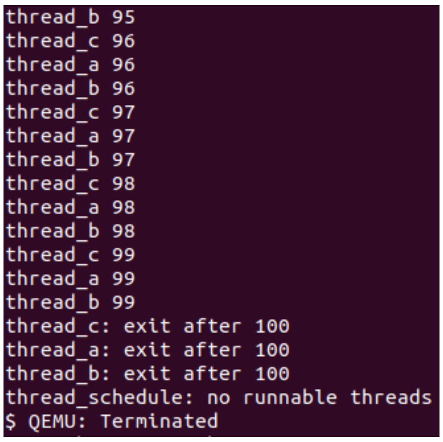
```c
  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)t, (uint64)next_thread); // current_thread改为next_thread
  } else
    next_thread = 0;
}
```

## 1.3 运行结果

输入 uthread 进行测试，结果如下。



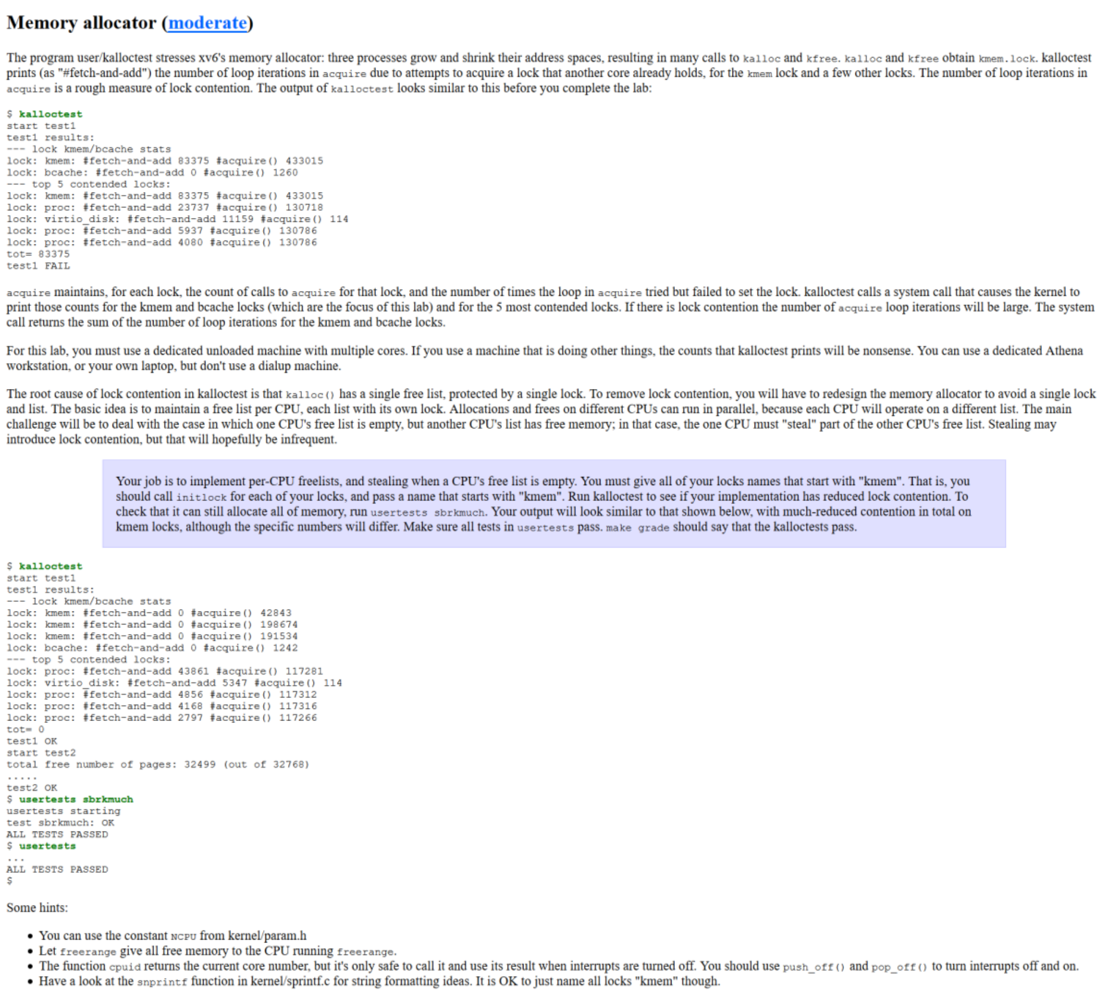
```
xv6 kernel is booting

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 2
thread_a 2
thread_b 2
```



```
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
$ QEMU: Terminated
```

# 2 Lock: Memory allocator

## 2.1 题目截图



### Memory allocator (moderate)

The program user/kalloctest stresses xv6's memory allocator: three processes grow and shrink their address spaces, resulting in many calls to `kalloc` and `kfree`. `kalloc` and `kfree` obtain `kmem.lock`. kalloctest prints (as "#fetch-and-add") the number of loop iterations in `acquire` due to attempts to acquire a lock that another core already holds, for the `kmem` lock and a few other locks. The number of loop iterations in `acquire` is a rough measure of lock contention. The output of `kalloctest` looks similar to this before you complete the lab:

```
$ 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
```

`acquire` maintains, for each lock, the count of calls to `acquire` for that lock, and the number of times the loop in `acquire` tried but failed to set the lock. kalloctest calls a system call that causes the kernel to print those counts for the kmem and bcache locks (which are the focus of this lab) and for the 5 most contended locks. If there is lock contention the number of `acquire` loop iterations will be large. The system call returns the sum of the number of loop iterations for the kmem and bcache locks.

For this lab, you must use a dedicated unloaded machine with multiple cores. If you use a machine that is doing other things, the counts that kalloctest prints will be nonsense. You can use a dedicated Athena workstation, or your own laptop, but don't use a dialup machine.

The root cause of lock contention in kalloctest is that `kalloc()` has a single free list, protected by a single lock. To remove lock contention, you will have to redesign the memory allocator to avoid a single lock and list. The basic idea is to maintain a free list per CPU, each list with its own lock. Allocations and frees on different CPUs can run in parallel, because each CPU will operate on a different list. The main challenge will be to deal with the case in which one CPU's free list is empty, but another CPU's list has free memory; in that case, the one CPU must "steal" part of the other CPU's free list. Stealing may introduce lock contention, but that will hopefully be infrequent.

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the kalloctests pass.

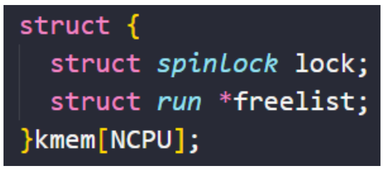
```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 42843
lock: kmem: #fetch-and-add 0 #acquire() 198674
lock: kmem: #fetch-and-add 0 #acquire() 191534
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #fetch-and-add 43861 #acquire() 117281
lock: virtio_disk: #fetch-and-add 5347 #acquire() 114
lock: proc: #fetch-and-add 4856 #acquire() 117312
lock: proc: #fetch-and-add 4168 #acquire() 117316
lock: proc: #fetch-and-add 2797 #acquire() 117266
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
...
ALL TESTS PASSED
$
```

Some hints:

- You can use the constant `NCPU` from kernel/param.h
- Let `freerange` give all free memory to the CPU running `freerange`.
- The function `cpuid` returns the current core number, but it's only safe to call it and use its result when interrupts are turned off. You should use `push_off()` and `pop_off()` to turn interrupts off and on.
- Have a look at the `snprintf` function in kernel/sprintf.c for string formatting ideas. It is OK to just name all locks "kmem" though.

## 2.2 实验步骤

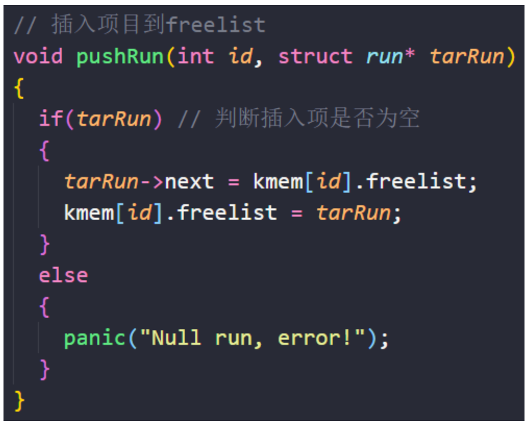
如下所示的结构体数组 kmem 用于表示机器的各 CPU，共有 NCPU 个，一个变量中包含有一个 spinlock 锁变量 lock 和一个自由链表指针变量 freelist。



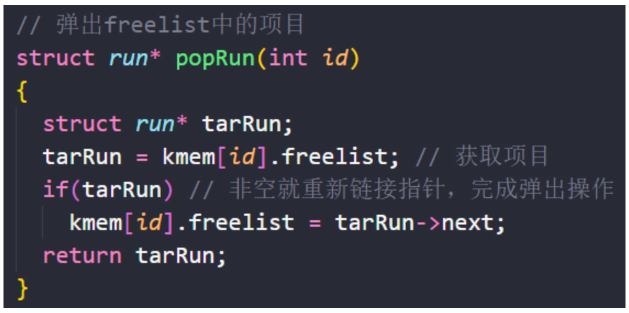
```
struct {
  struct spinlock lock;
  struct run *freelist;
}kmem[NCPU];
```

根据题设条件我们可以得知，出现锁争抢的根本原因在于函数 kalloc 中只有一个自由链表，该自由链表只被一个锁保护。为了避免锁争抢，我们必须改进单自由链表、单锁的情况。最基本的思想就是为每一个 CPU 都维护一个自由链表，同时每一个自由链表都有一个独自的锁。这样一来，不同的 CPU 就能并行地利用函数 kalloc/kfree 分配与释放内存了，因为每个 CPU 都会操作不同的自由链表。在实现这个方案的过程中，最具挑战的就是：当一个 CPU 的自由链表无项目，但是另一个 CPU 的自由链表还有空闲块时，我们就应该从有空闲自由链表的 CPU 处“窃取”一个空闲内存块。

根据 xv6 对自由链表的定义，可知链表的插入使用的是头插法，于是我们可以封装两个函数 pushRun 和 popRun，分别完成自由链表的插入操作和弹出操作，函数定义如下。

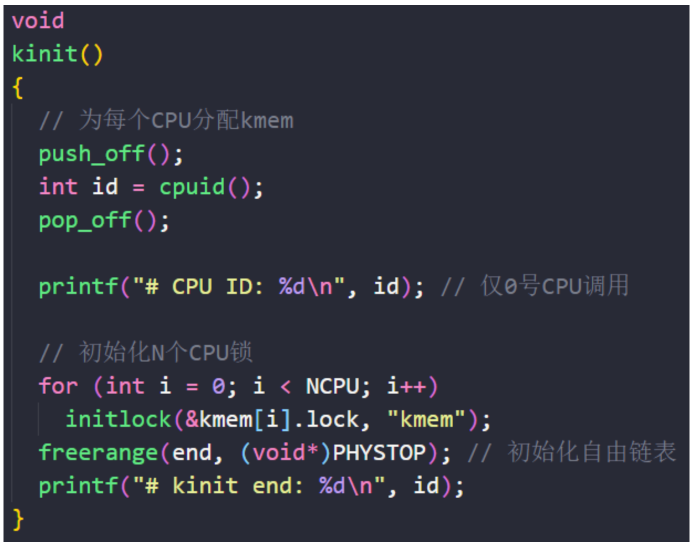


```c
// 插入项目到freelist
void pushRun(int id, struct run* tarRun)
{
  if(tarRun) // 判断插入项是否为空
  {
    tarRun->next = kmem[id].freelist;
    kmem[id].freelist = tarRun;
  }
  else
  {
    panic("Null run, error!");
  }
}
```



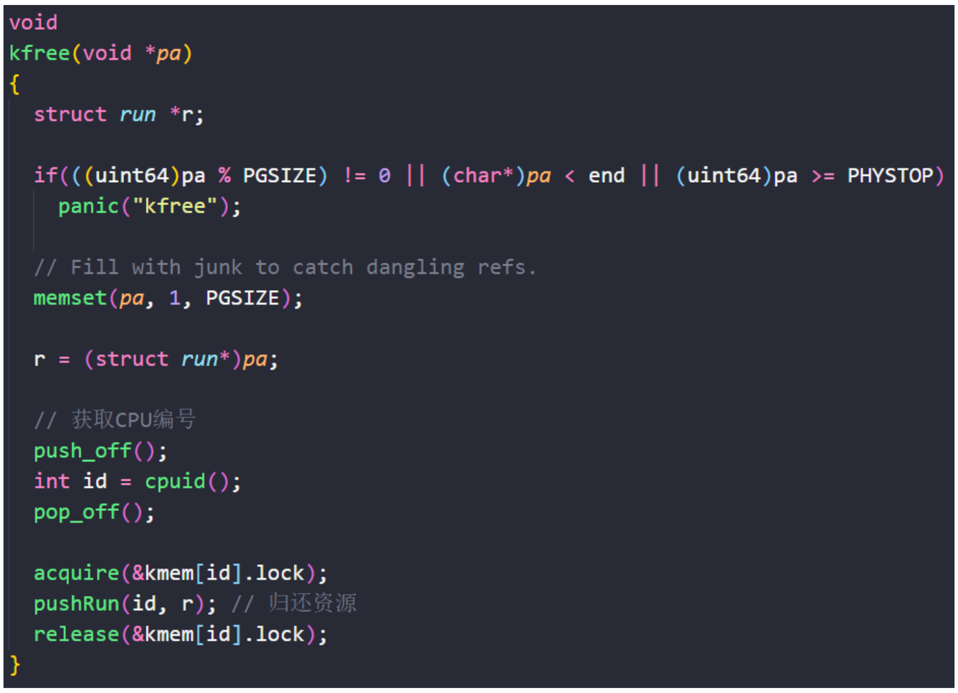
```c
// 弹出freelist中的项目
struct run* popRun(int id)
{
  struct run* tarRun;
  tarRun = kmem[id].freelist; // 获取项目
  if(tarRun) // 非空就重新链接指针，完成弹出操作
    kmem[id].freelist = tarRun->next;
  return tarRun;
}
```

接下来，对函数 kinit 进行分析，先根据题目中给出的方式，获取当前 CPU 的 id，通过调用函数 cpuid 完成，随后打印输出 id 值。接下来，利用 for 循环初始化 N 个 CPU 锁，通过在 for 循环中对结构体数组 kmem 的每个成员调用 initlock 实现。退出 for 循环后，调用函数 freerange 初始化自由链表。



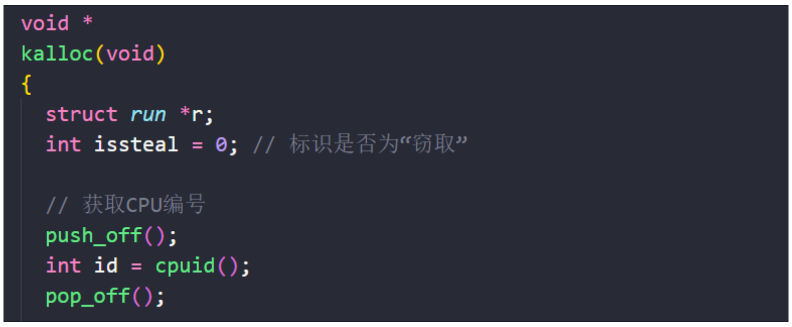
```c
void
kinit()
{
  // 为每个CPU分配kmem
  push_off();
  int id = cpuid();
  pop_off();

  printf("# CPU ID: %d\n", id); // 仅0号CPU调用

  // 初始化N个CPU锁
  for (int i = 0; i < NCPU; i++)
    initlock(&kmem[i].lock, "kmem");
  freerange(end, (void*)PHYSTOP); // 初始化自由链表
  printf("# kinit end: %d\n", id);
}
```

接下来，对函数 kfree 进行分析，接收了传入的指针 pa 后，先利用 if 语句判定是否要打印说明文字，再调用函数 memset 对 pa 指向的块进行处理，随后将 void 类型的指针 pa 强制转型为一个 struct run 型的指针 r。随后，先调用函数 cpuid 获取当前 CPU 标识号 id，再调用函数 popRun，将块 r 插入相应的自由链表，以归还资源。



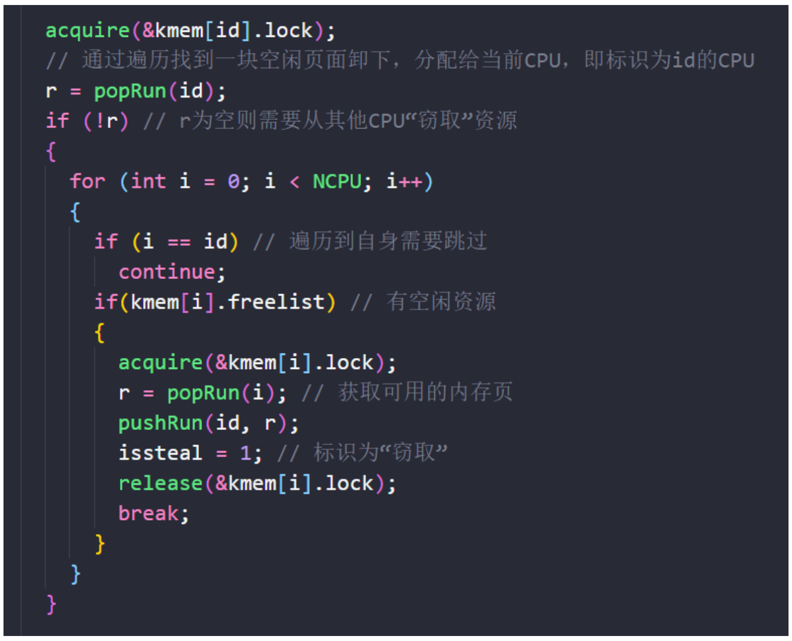
```c
void
kfree(void *pa)
{
  struct run *r;

  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CPU编号
  push_off();
  int id = cpuid();
  pop_off();

  acquire(&kmem[id].lock);
  pushRun(id, r); // 归还资源
  release(&kmem[id].lock);
}
```

最后，我们对本次实验的核心部分，即函数 kalloc 进行分析。函数体中先定义了一个 run 结构体 r 和一个标识 CPU 中的自由链表是否为“窃取”的整型变量 issteal。接下来调用函数 cpuid 获取当前 CPU 编号。



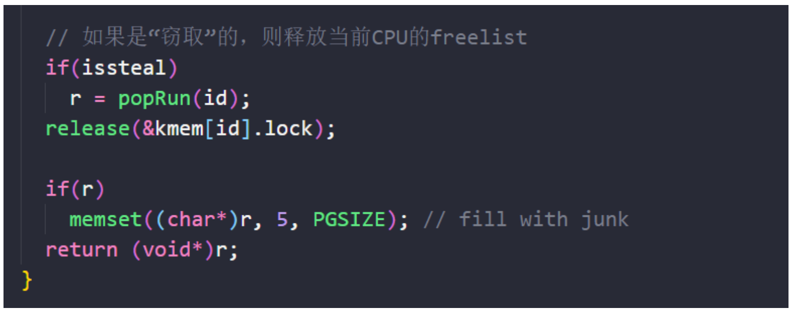
```c
void *
kalloc(void)
{
  struct run *r;
  int issteal = 0; // 标识是否为“窃取”

  // 获取CPU编号
  push_off();
  int id = cpuid();
  pop_off();
```

随后，调用函数 acquire 获取当前 CPU 的锁，防止被其他线程访问，并调用函数 popRun 尝试将当前 CPU 的自由链表卸下，函数返回结果存储在变量 r 中。接下来，对变量 r 的值进行判断，如果 r 值为空则说明需要从其他 CPU 窃取资源，则进入 for 循环遍历 CPU 结构体数组 kmem，在循环中对非标识号非 id 的 CPU 均进行判断，如果其 freelist 数据域非空，则说明有空闲资源，先调用函数 acquire 获取锁，再调用函数 popRun 获取可用的内存页，再调用函数 pushRun 将获取的资源加入当前 CPU 中，并将变量 issteal 的值改为 1，以标识窃取行为，最后调用 break 跳出 for 循环。



```
    acquire(&kmem[id].lock);
    // 通过遍历找到一块空闲页面卸下，分配给当前CPU，即标识为id的CPU
    r = popRun(id);
    if (!r) // r为空则需要从其他CPU“窃取”资源
    {
        for (int i = 0; i < NCPU; i++)
        {
            if (i == id) // 遍历到自身需要跳过
                continue;
            if(kmem[i].freelist) // 有空闲资源
            {
                acquire(&kmem[i].lock);
                r = popRun(i); // 获取可用的内存页
                pushRun(id, r);
                issteal = 1; // 标识为“窃取”
                release(&kmem[i].lock);
                break;
            }
        }
    }
```

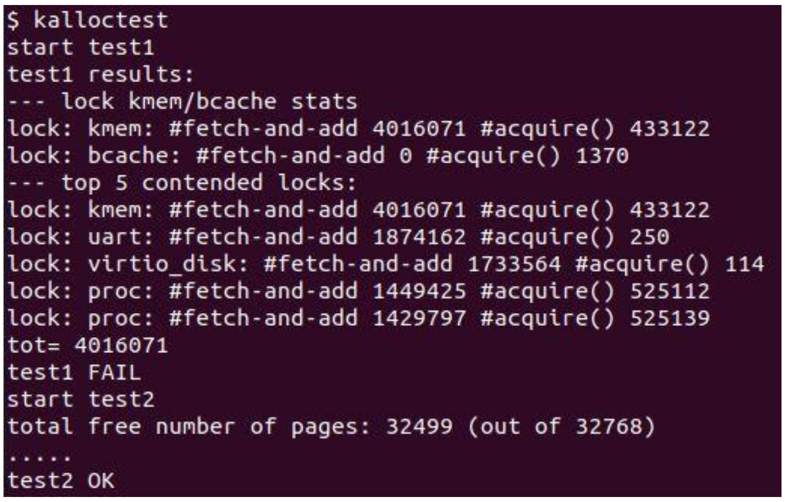
接下来，对变量 issteal 的值进行判断，如果该变量值非 0，则说明为窃取的，要将其 freelist 释放，调用函数 popRun 弹出，结果储存在变量 r 中，再对变量 r 的值进行判断，如非空则调用函数 memset 进行释放。最后对变量 r 进行强制转型，返回 void 类型的指针 r。



```
    // 如果是“窃取”的，则释放当前CPU的freelist
    if(issteal)
        r = popRun(id);
    release(&kmem[id].lock);

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

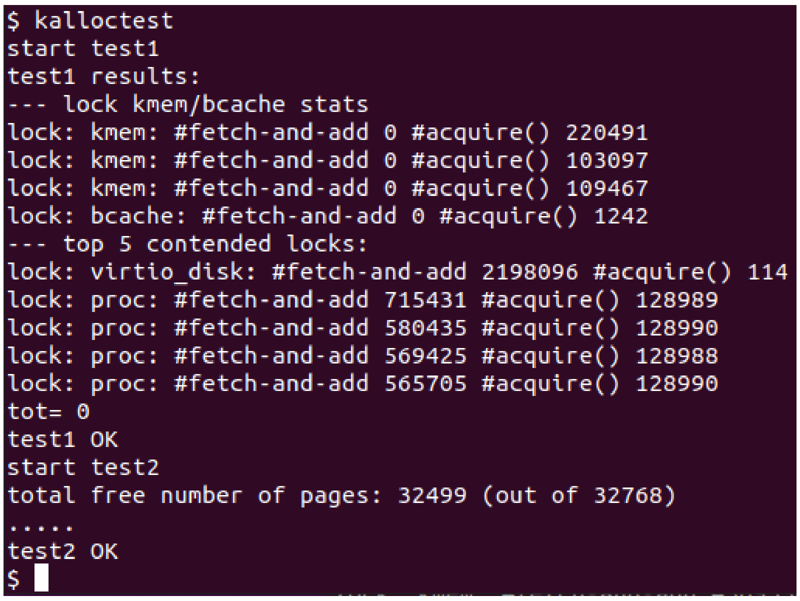
## 2.3 运行结果

更改 kernel/kalloc.c 前，输入 kalloctest 的运行结果如下。



```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 4016071 #acquire() 433122
lock: bcache: #fetch-and-add 0 #acquire() 1370
--- top 5 contended locks:
lock: kmem: #fetch-and-add 4016071 #acquire() 433122
lock: uart: #fetch-and-add 1874162 #acquire() 250
lock: virtio_disk: #fetch-and-add 1733564 #acquire() 114
lock: proc: #fetch-and-add 1449425 #acquire() 525112
lock: proc: #fetch-and-add 1429797 #acquire() 525139
tot= 4016071
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

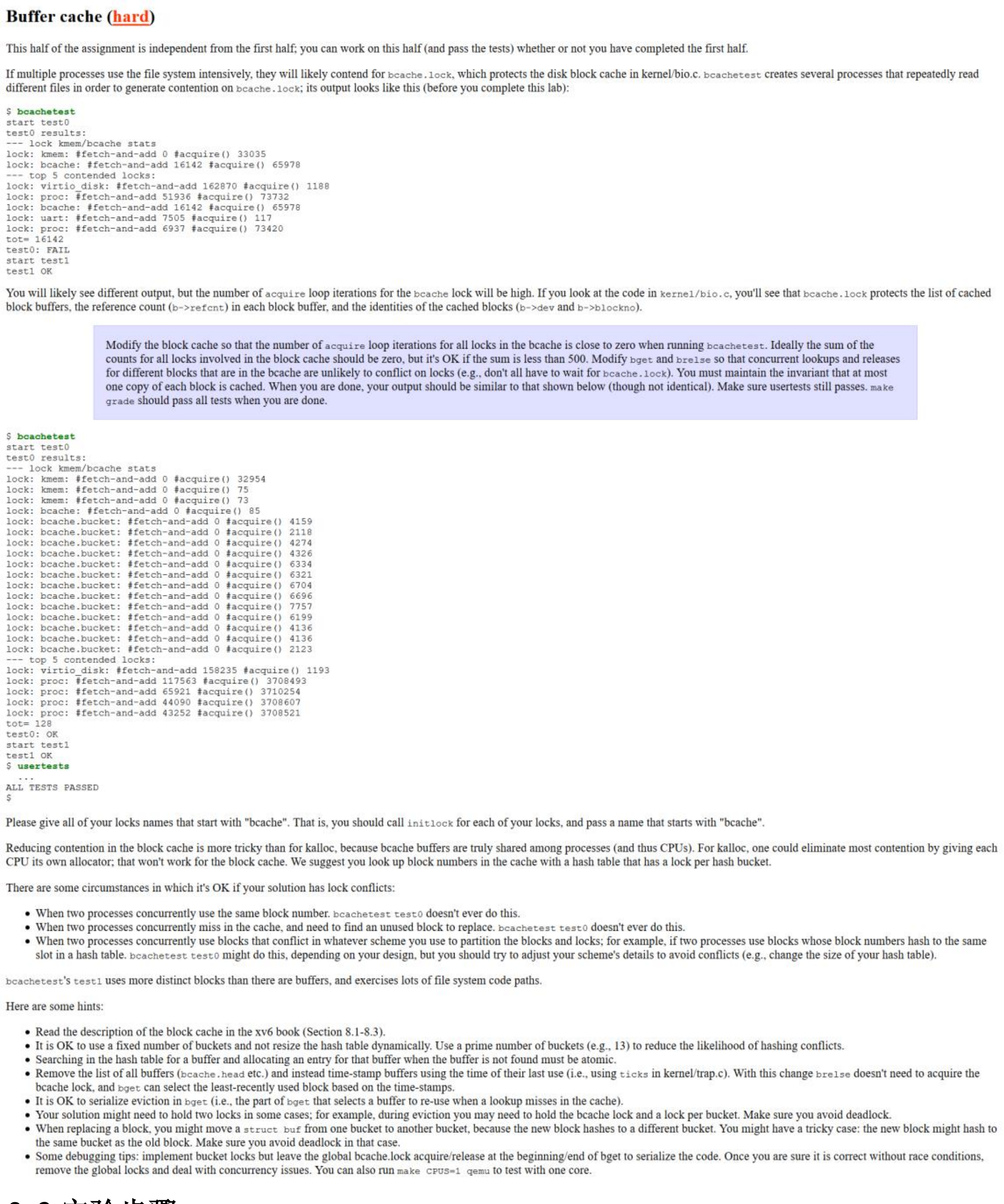
更改 kernel/kalloc.c 后，输入 kalloctest 的运行结果如下。



```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 220491
lock: kmem: #fetch-and-add 0 #acquire() 103097
lock: kmem: #fetch-and-add 0 #acquire() 109467
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 2198096 #acquire() 114
lock: proc: #fetch-and-add 715431 #acquire() 128989
lock: proc: #fetch-and-add 580435 #acquire() 128990
lock: proc: #fetch-and-add 569425 #acquire() 128988
lock: proc: #fetch-and-add 565705 #acquire() 128990
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

# 3 Lock: Buffer cache

## 3.1 题目截图



### Buffer cache (hard)

This half of the assignment is independent from the first half; you can work on this half (and pass the tests) whether or not you have completed the first half.

If multiple processes use the file system intensively, they will likely contend for `bcache.lock`, which protects the disk block cache in kernel/bio.c. `bcachetest` creates several processes that repeatedly read different files in order to generate contention on `bcache.lock`; its output looks like this (before you complete this lab):

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33035
lock: bcache: #fetch-and-add 16142 #acquire() 65978
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 162870 #acquire() 1188
lock: proc: #fetch-and-add 51936 #acquire() 73732
lock: bcache: #fetch-and-add 16142 #acquire() 65978
lock: uart: #fetch-and-add 7505 #acquire() 117
lock: proc: #fetch-and-add 6937 #acquire() 73420
tot= 16142
test0: FAIL
start test1
test1 OK
```

You will likely see different output, but the number of `acquire` loop iterations for the `bcache` lock will be high. If you look at the code in `kernel/bio.c`, you'll see that `bcache.lock` protects the list of cached block buffers, the reference count (`b->refcnt`) in each block buffer, and the identities of the cached blocks (`b->dev` and `b->blockno`).

Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure usertests still passes. `make grade` should pass all tests when you are done.

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32954
lock: kmem: #fetch-and-add 0 #acquire() 75
lock: kmem: #fetch-and-add 0 #acquire() 73
lock: bcache: #fetch-and-add 0 #acquire() 85
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4159
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2118
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4274
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4326
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6334
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6321
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6704
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6696
lock: bcache.bucket: #fetch-and-add 0 #acquire() 7757
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2123
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 158235 #acquire() 1193
lock: proc: #fetch-and-add 117563 #acquire() 3708493
lock: proc: #fetch-and-add 65921 #acquire() 3710254
lock: proc: #fetch-and-add 44090 #acquire() 3708607
lock: proc: #fetch-and-add 43252 #acquire() 3708521
tot= 128
test0: OK
start test1
test1 OK
$ usertests
  ...
ALL TESTS PASSED
$
```

Please give all of your locks names that start with "bcache". That is, you should call `initlock` for each of your locks, and pass a name that starts with "bcache".

Reducing contention in the block cache is more tricky than for kalloc, because bcache buffers are truly shared among processes (and thus CPUs). For kalloc, one could eliminate most contention by giving each CPU its own allocator; that won't work for the block cache. We suggest you look up block numbers in the cache with a hash table that has a lock per hash bucket.

There are some circumstances in which it's OK if your solution has lock conflicts:

- When two processes concurrently use the same block number. `bcachetest test0` doesn't ever do this.
- When two processes concurrently miss in the cache, and need to find an unused block to replace. `bcachetest test0` doesn't ever do this.
- When two processes concurrently use blocks that conflict in whatever scheme you use to partition the blocks and locks; for example, if two processes use blocks whose block numbers hash to the same slot in a hash table. `bcachetest test0` might do this, depending on your design, but you should try to adjust your scheme's details to avoid conflicts (e.g., change the size of your hash table).

`bcachetest`'s `test1` uses more distinct blocks than there are buffers, and exercises lots of file system code paths.

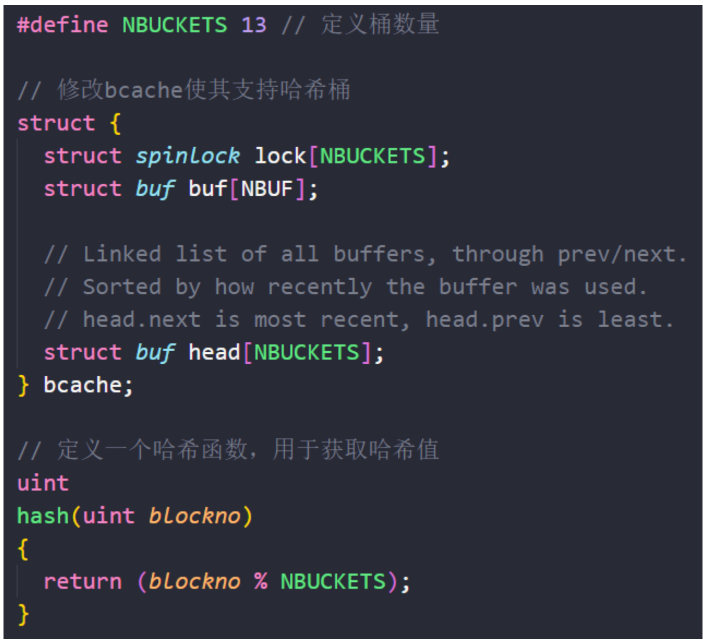
Here are some hints:

- Read the description of the block cache in the xv6 book (Section 8.1-8.3).
- It is OK to use a fixed number of buckets and not resize the hash table dynamically. Use a prime number of buckets (e.g., 13) to reduce the likelihood of hashing conflicts.
- Searching in the hash table for a buffer and allocating an entry for that buffer when the buffer is not found must be atomic.
- Remove the list of all buffers (`bcache.head` etc.) and instead time-stamp buffers using the time of their last use (i.e., using `ticks` in kernel/trap.c). With this change `brelse` doesn't need to acquire the bcache lock, and `bget` can select the least-recently used block based on the time-stamps.
- It is OK to serialize eviction in `bget` (i.e., the part of `bget` that selects a buffer to re-use when a lookup misses in the cache).
- Your solution might need to hold two locks in some cases; for example, during eviction you may need to hold the bcache lock and a lock per bucket. Make sure you avoid deadlock.
- When replacing a block, you might move a `struct buf` from one bucket to another bucket, because the new block hashes to a different bucket. You might have a tricky case: the new block might hash to the same bucket as the old block. Make sure you avoid deadlock in that case.
- Some debugging tips: implement bucket locks but leave the global bcache.lock acquire/release at the beginning/end of bget to serialize the code. Once you are sure it is correct without race conditions, remove the global locks and deal with concurrency issues. You can also run `make CPUS=1 qemu` to test with one core.

## 3.2 实验步骤

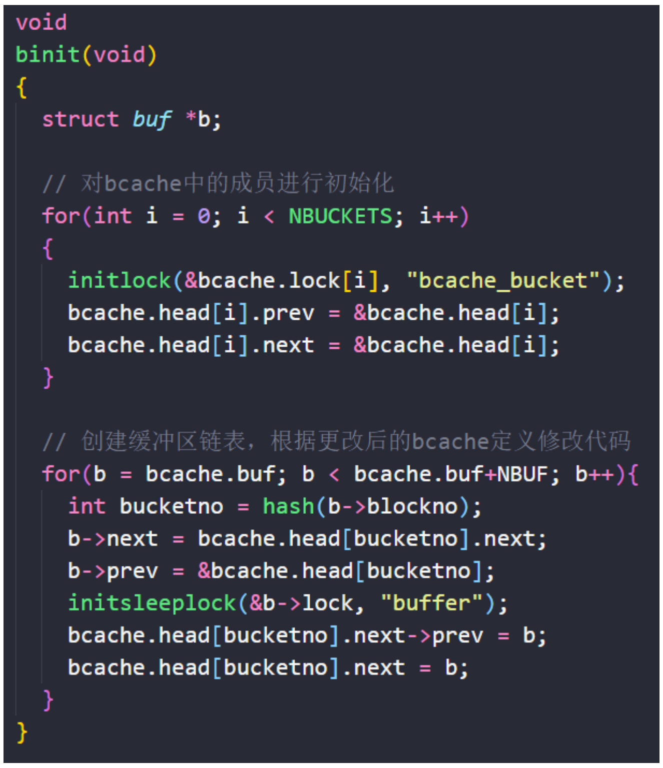
该实验的大意为在多线程环境下，在未修改磁盘缓冲区结构 buffer cache 时，因为该结构只有一个全局锁，会导致获取缓冲区时锁竞争严重，我们需要解决这个问题。这里我们可以为每个 CPU 分配属于自己的内存链表，但是不能为 CPU 分配属于自己的磁盘缓冲区，我们在设计上应该让多个 CPU 访问磁盘同一块区域时访问同一块缓冲区，减少空间浪费，并且提升系统性能。在该实验中，我们主要需要修改的文件为 kernel/bio.c

在文件开始处，我们先定义一个宏常量 NBUCKETS，该常量表示哈希桶的数量，这里设置为 13，因为题干中有指出设置为 13 这样的质数有利于减少冲突。随后，我们修改结构体 bcache，让变量 lock 和 head 都成为数组类型的变量，以使 bcache 在结构上支持哈希桶。接下来，我们再定义一个哈希函数 hash，该函数对传入的键进行一个取模操作以获取哈希值。



```c
#define NBUCKETS 13 // 定义桶数量

// 修改bcache使其支持哈希桶
struct {
  struct spinlock lock[NBUCKETS];
  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[NBUCKETS];
} bcache;

// 定义一个哈希函数，用于获取哈希值
uint
hash(uint blockno)
{
  return (blockno % NBUCKETS);
}
```

接下来，我们对函数 kinit 进行修改。先用一个 for 循环对 bcache 的中的数据域进行初始化。随后进入下一个 for 循环，创建缓冲区链表，共计创建了 NBUF 个缓冲区链表，其中我们要根据缓冲区 b 的 blockno 值对其进行哈希，再对 bcache 中对应下标的成员进行初始化。



```c
void
binit(void)
{
  struct buf *b;

  // 对bcache中的成员进行初始化
  for(int i = 0; i < NBUCKETS; i++)
  {
    initlock(&bcache.lock[i], "bcache_bucket");
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
  }

  // 创建缓冲区链表，根据更改后的bcache定义修改代码
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    int bucketno = hash(b->blockno);
    b->next = bcache.head[bucketno].next;
    b->prev = &bcache.head[bucketno];
    initsleeplock(&b->lock, "buffer");
    bcache.head[bucketno].next->prev = b;
    bcache.head[bucketno].next = b;
  }
}
```

接下来，对函数 bget 进行分析，该函数用于当在自己的哈希桶中没有找到相应的扇区时，从别的哈希桶处“窃取”一个扇区。函数体中先调用函数 hash 获取传入参数 blockno 的哈希值 bucketno，并调用函数 acquire 获取 bcache[bucketno]的锁，开始进行操作。接下来进入 for 循环，遍历链表 bcache.head[bucketno]，在 for 循环体内有一个 if 条件判定，用于判定当前循环变量 b 的 dev 值是否与传入的 dev 参量相等，同时判定 b 的 blockno 值是否与传入的 blockno 值相等，如果这两个条件都满足，说明找到了目标扇区，可以让 b 的 refcnt 值自增，然后释放锁，并且将当前 b 作为返回值返回。

```c
// 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 bucketno = hash(blockno);

  acquire(&bcache.lock[bucketno]);

  // Is the block already cached?
  for(b = bcache.head[bucketno].next; b != &bcache.head[bucketno]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock[bucketno]);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

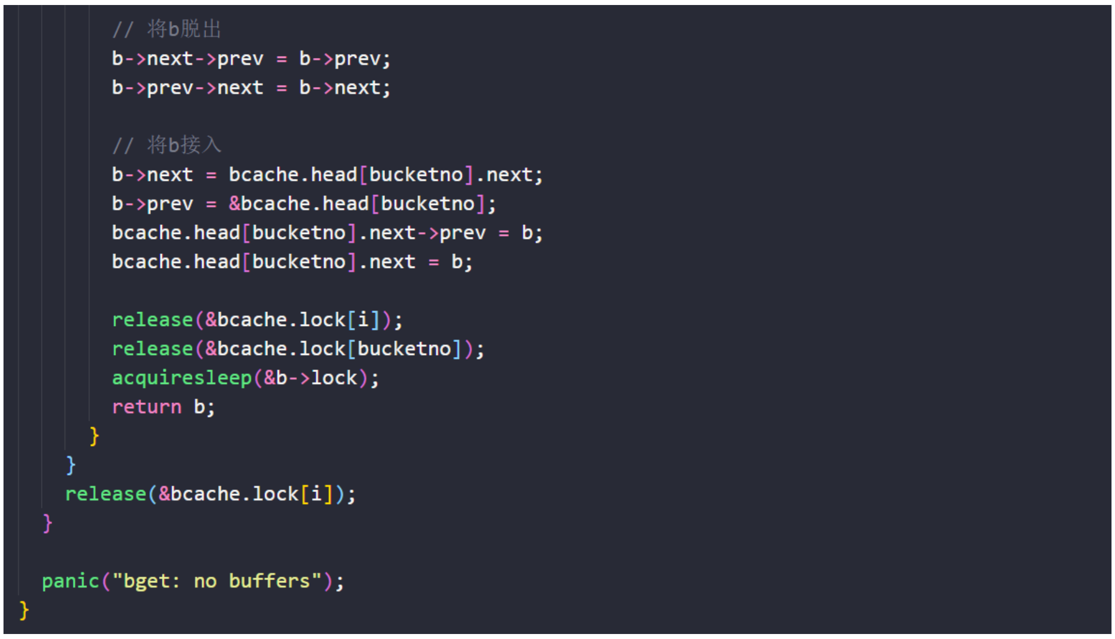
然而，如果遍历完链表 bcache[bucketno]都没有找到目标扇区，则说明我们需要执行“窃取”操作。于是进入下面的 for 循环，对其他哈希值所对应的桶进行遍历。在循环体中，先获取锁 bcache.lock[i]，然后进入内层 for 循环，对链表 bcache.head[i]进行遍历，其内部的 if 变量表示如果找到一个块 b 的 refcnt 值为 0，则说明我们找到了一个可以使用的扇区，故将其 dev 数据域设为传入参数 dev，blockno 值设为传入参数 blockno，valid 值设为 0，refcnt 值设为 1，表示该扇区已被我们占用。

```c
  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  for(int i = hash(blockno + 1); i != bucketno; i = (i + 1)%NBUCKETS) // 添加一个外层for循环
  {
    acquire(&bcache.lock[i]);
    for(b = bcache.head[i].prev; b != &bcache.head[i]; b = b->prev){
      if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
```

随后，我们改变两个指针 b->next->prev 和 b->prev->next 的指向，将 b 从原来的位置脱出，再重新链接一系列指针，将 b 接入 bcache.head[bucketno]，即完成了扇区的“窃取”。最后，我们释放一系列锁，将 b 作为返回值返回。

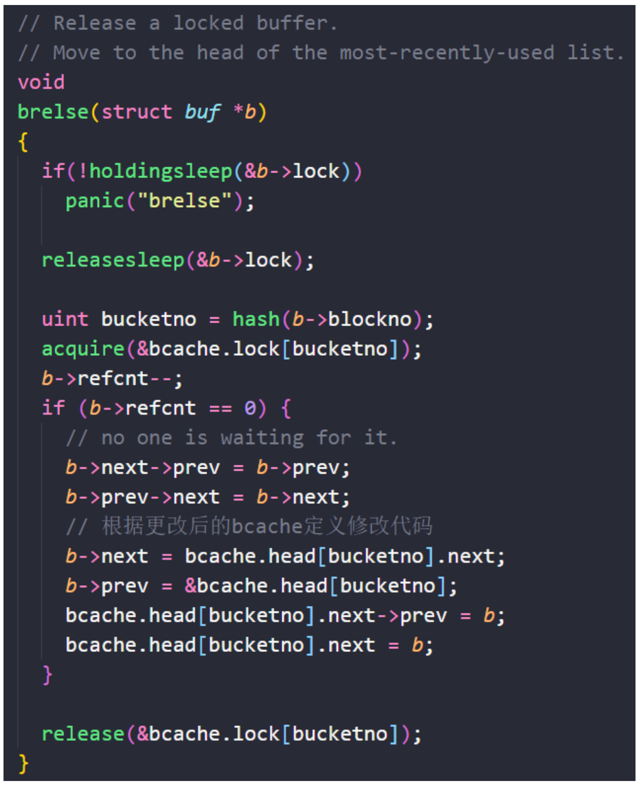
然而，如果 if 条件（判定 b 是否可用）未满足时，则调用函数 release 释放锁 bcache.lock[i]，进入 for 循环的下一轮。

最后，如果 for 循环结束都没有找到可用的块，则调用函数 panic 报错，表示找不到可用的缓冲区。

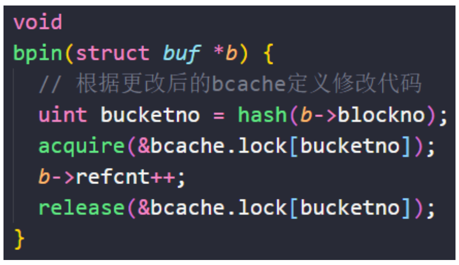


```
        // 将b脱出
        b->next->prev = b->prev;
        b->prev->next = b->next;

        // 将b接入
        b->next = bcache.head[bucketno].next;
        b->prev = &bcache.head[bucketno];
        bcache.head[bucketno].next->prev = b;
        bcache.head[bucketno].next = b;

        release(&bcache.lock[i]);
        release(&bcache.lock[bucketno]);
        acquiresleep(&b->lock);
        return b;
      }
    }
    release(&bcache.lock[i]);
  }

  panic("bget: no buffers");
}
```

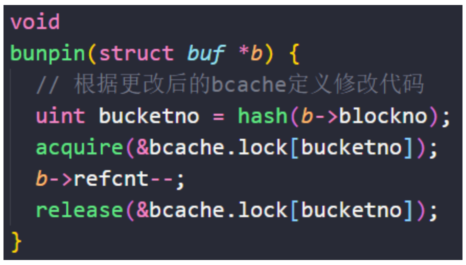
此外，我们还需要根据修改后的 bcache 结构对后续函数中的部分语句进行调整，主要表现在在 bcache 的成员变量后增加“[bucketno]”，列出如下。



```
// 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);

  uint bucketno = hash(b->blockno);
  acquire(&bcache.lock[bucketno]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    // 根据更改后的bcache定义修改代码
    b->next = bcache.head[bucketno].next;
    b->prev = &bcache.head[bucketno];
    bcache.head[bucketno].next->prev = b;
    bcache.head[bucketno].next = b;
  }

  release(&bcache.lock[bucketno]);
}
```



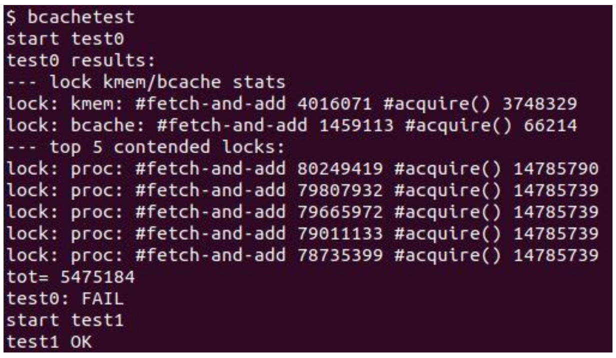
```
void
bpin(struct buf *b) {
  // 根据更改后的bcache定义修改代码
  uint bucketno = hash(b->blockno);
  acquire(&bcache.lock[bucketno]);
  b->refcnt++;
  release(&bcache.lock[bucketno]);
}
```



```
void
bunpin(struct buf *b) {
  // 根据更改后的bcache定义修改代码
  uint bucketno = hash(b->blockno);
  acquire(&bcache.lock[bucketno]);
  b->refcnt--;
  release(&bcache.lock[bucketno]);
}
```

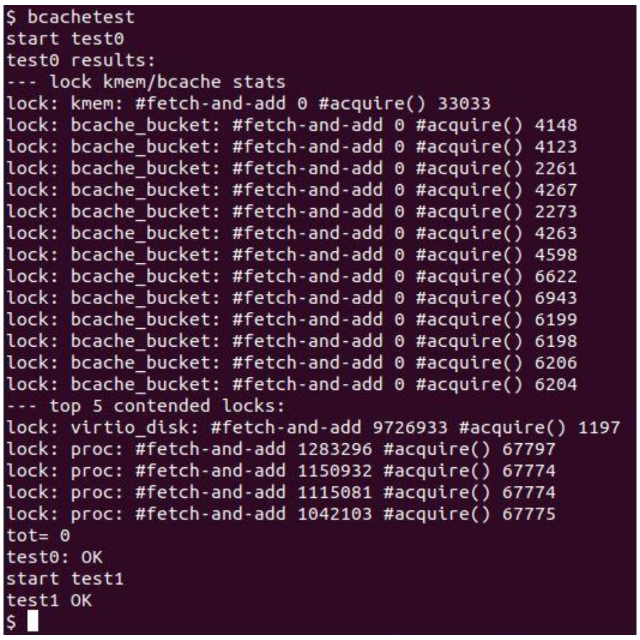
## 3. 3 运行结果

修改代码前的 bcachetest 运行截图。



```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 4016071 #acquire() 3748329
lock: bcache: #fetch-and-add 1459113 #acquire() 66214
--- top 5 contended locks:
lock: proc: #fetch-and-add 80249419 #acquire() 14785790
lock: proc: #fetch-and-add 79807932 #acquire() 14785739
lock: proc: #fetch-and-add 79665972 #acquire() 14785739
lock: proc: #fetch-and-add 79011133 #acquire() 14785739
lock: proc: #fetch-and-add 78735399 #acquire() 14785739
tot= 5475184
test0: FAIL
start test1
test1 OK
```

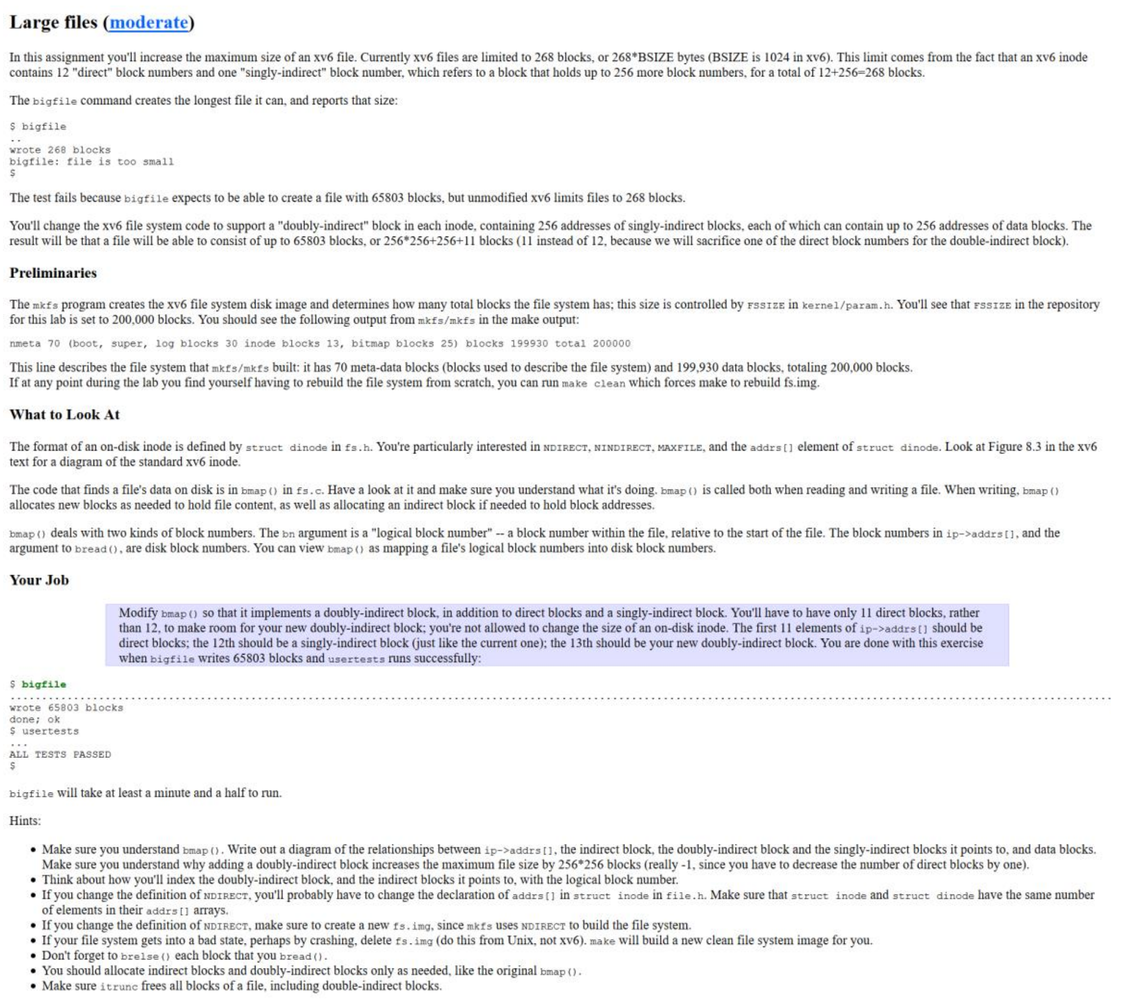
修改代码后的 bcachetest 运行截图。可以看出，对代码的修改大幅减少了冲突。



```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33033
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4148
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4123
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2261
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4267
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2273
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4263
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4598
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6622
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6943
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6198
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6206
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6204
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 9726933 #acquire() 1197
lock: proc: #fetch-and-add 1283296 #acquire() 67797
lock: proc: #fetch-and-add 1150932 #acquire() 67774
lock: proc: #fetch-and-add 1115081 #acquire() 67774
lock: proc: #fetch-and-add 1042103 #acquire() 67775
tot= 0
test0: OK
start test1
test1 OK
$ 
```

# 4 File System: Large files

## 4.1 题目截图



### Large files (moderate)

In this assignment you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 268 blocks, or 268*BSIZE bytes (BSIZE is 1024 in xv6). This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks.

The `bigfile` command creates the longest file it can, and reports that size:

```
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
$
```

The test fails because `bigfile` expects to be able to create a file with 65803 blocks, but unmodified xv6 limits files to 268 blocks.

You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks. The result will be that a file will be able to consist of up to 65803 blocks, or 256*256+256+11 blocks (11 instead of 12, because we will sacrifice one of the direct block numbers for the double-indirect block).

#### Preliminaries

The `mkfs` program creates the xv6 file system disk image and determines how many total blocks the file system has; this size is controlled by `FSSIZE` in `kernel/param.h`. You'll see that `FSSIZE` in the repository for this lab is set to 200,000 blocks. You should see the following output from `mkfs/mkfs` in the make output:

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks 199930 total 200000
```

This line describes the file system that `mkfs/mkfs` built: it has 70 meta-data blocks (blocks used to describe the file system) and 199,930 data blocks, totaling 200,000 blocks.
If at any point during the lab you find yourself having to rebuild the file system from scratch, you can run `make clean` which forces make to rebuild fs.img.

#### What to Look At

The format of an on-disk inode is defined by `struct dinode` in `fs.h`. You're particularly interested in `NDIRECT`, `NINDIRECT`, `MAXFILE`, and the `addrs[]` element of `struct dinode`. Look at Figure 8.3 in the xv6 text for a diagram of the standard xv6 inode.

The code that finds a file's data on disk is in `bmap()` in `fs.c`. Have a look at it and make sure you understand what it's doing. `bmap()` is called both when reading and writing a file. When writing, `bmap()` allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses.

`bmap()` deals with two kinds of block numbers. The `bn` argument is a "logical block number" -- a block number within the file, relative to the start of the file. The block numbers in `ip->addrs[]`, and the argument to `bread()`, are disk block numbers. You can view `bmap()` as mapping a file's logical block numbers into disk block numbers.

#### Your Job

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests` runs successfully:

```
$ bigfile
..................................................................................................................................
wrote 65803 blocks
done; ok
$ usertests
...
ALL TESTS PASSED
$
```

`bigfile` will take at least a minute and a half to run.

Hints:

- Make sure you understand `bmap()`. Write out a diagram of the relationships between `ip->addrs[]`, the indirect block, the doubly-indirect block and the singly-indirect blocks it points to, and data blocks. Make sure you understand why adding a doubly-indirect block increases the maximum file size by 256*256 blocks (really -1, since you have to decrease the number of direct blocks by one).
- Think about how you'll index the doubly-indirect block, and the indirect blocks it points to, with the logical block number.
- If you change the definition of `NDIRECT`, you'll probably have to change the declaration of `addrs[]` in `struct inode` in `file.h`. Make sure that `struct inode` and `struct dinode` have the same number of elements in their `addrs[]` arrays.
- If you change the definition of `NDIRECT`, make sure to create a new `fs.img`, since `mkfs` uses `NDIRECT` to build the file system.
- If your file system gets into a bad state, perhaps by crashing, delete `fs.img` (do this from Unix, not xv6). `make` will build a new clean file system image for you.
- Don't forget to `brelse()` each block that you `bread()`.
- You should allocate indirect blocks and doubly-indirect blocks only as needed, like the original `bmap()`.
- Make sure `itrunc` frees all blocks of a file, including double-indirect blocks.

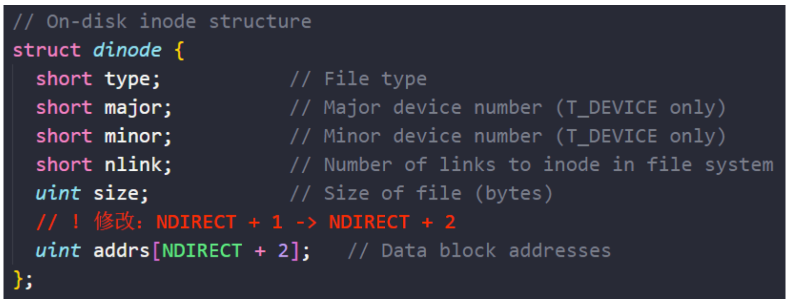
## 4.2 实验步骤

修改 bmap()，使其实现双重间接块，以及直接块和单间接块。我们必须拥有 11 个直接块，而不是 12 个，以便为新的双重间接块腾出空间;不允许更改磁盘上节点的大小。ip->addrs[]的前 11 个元素应该是直接块；第 12 个区块应该是单一间接的区块(就像当前的区块一样);第十三号应该是你新的双重间接块。

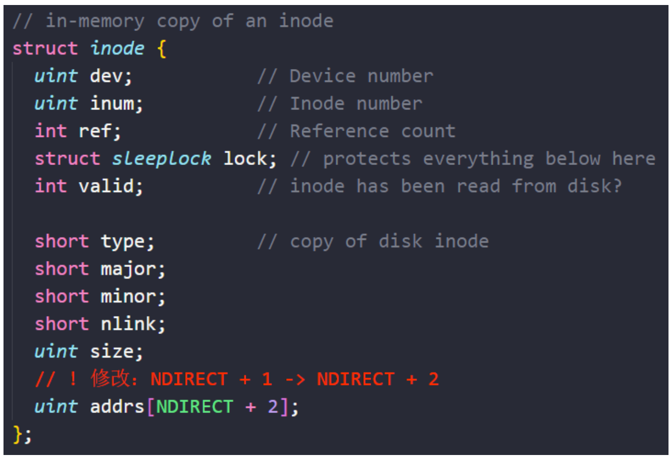
首先，我们对文件 kernel/fs.h、kernel/file.h 中的常量进行修改。对 kernel/fs.h 的修改如下，先将表示直接块大小的常量 NDIRECT 值由 12 改为 11，便为新的双重间接块腾出空间；再新增一个常量 NSECONDINDIRECT 用于表示第二层间接块。

```
#define NDIRECT 11 // ! 修改: 12 -> 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NSECONDINDIRECT (NINDIRECT * NINDIRECT) // ! 新增项
// ! 修改: NDIRECT + NINDIRECT -> NDIRECT + NINDIRECT + NSECONDINDIRECT
#define MAXFILE (NDIRECT + NINDIRECT + NSECONDINDIRECT)
```

我们还需要对结构体 dinode 进行修改。将其中数组 addrs 的规模由 NDIRECT + 1 改为 NDIRECT + 2。同理，我们也对文件 kernel/file.h 中的 inode 作类似修改。

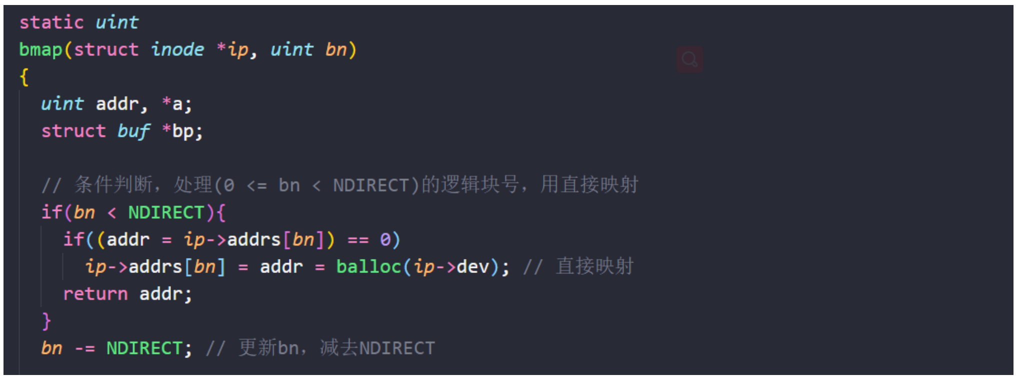


```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  // ! 修改: NDIRECT + 1 -> NDIRECT + 2
  uint addrs[NDIRECT + 2];   // Data block addresses
};
```



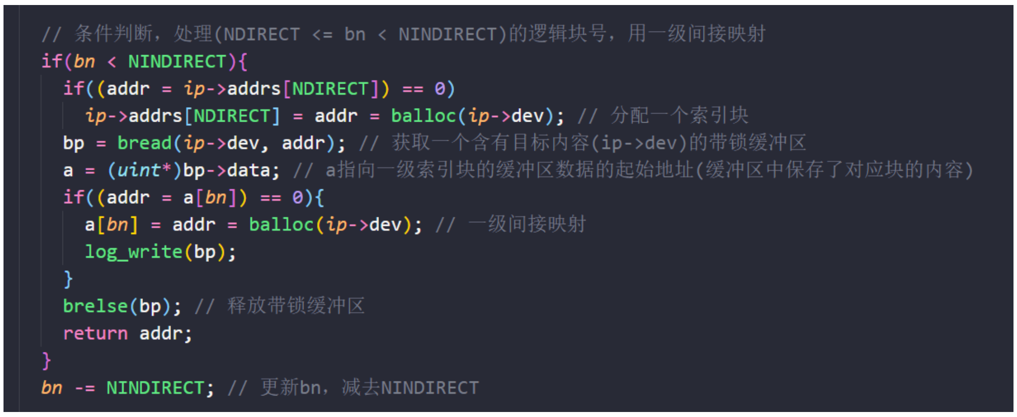
```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  // ! 修改: NDIRECT + 1 -> NDIRECT + 2
  uint addrs[NDIRECT + 2];
};
```

接下来，我们对函数 bmap 进行分析，这也是我们要修改的核心部分。函数体中先定义了一个无符号整形变量 addr、一个无符号整形指针 a 和一个缓冲区结构体 bp，addr 用于存储需要返回的地址，a 和 bp 用于保存临时变量。随后，外层 if 语句先对传入参数 bn 小于 NDIRECT 的情况进行判断，即 0 <= bn < NDIRECT 时的情况，随后内层 if 语句先将 ip->addrs[bn]的值赋予 addr，再判定其值是否为 0，如果为 0 则说明地址对应的块为空，可以直接进行映射，通过语句 ip->addrs[bn] = addr = balloc(ip->dev);完成映射，最后将 addr 作为返回值返回。若 if 块中的条件不满足，则不进入 if 块，并将 bn 的值减去 NDIRECT。



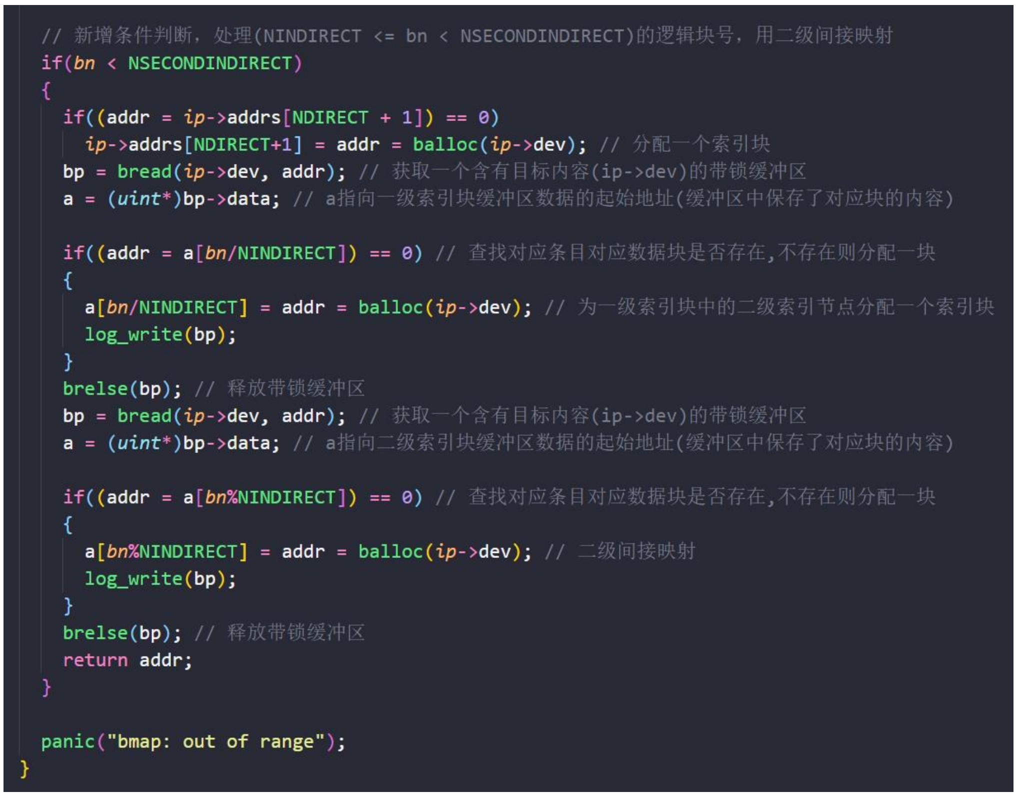
```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  // 条件判断，处理(0 <= bn < NDIRECT)的逻辑块号，用直接映射
  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev); // 直接映射
    return addr;
  }
  bn -= NDIRECT; // 更新bn，减去NDIRECT
```

随后，再通过 if 语句对 bn 的值进行判断，以处理 NDIRECT <= bn < NINDIRECT 的情况。在 if 块中，先分配一个索引块，随后调用函数 bread 获取一个含有目标内容，即 ip->dev 的带锁缓冲区，再令变量 a 指向一级索引块的缓冲区数据的起始地址(缓冲区中保存了对应块的内容)。随后，对 a[bn]的可用性进行判断，如果其值为 0，则进行一级间接映射，并调用函数 log_write(bp)记录，随后调用函数 brelse 释放带锁缓冲区，再返回 addr。如果 bn 不满足条件，则减去值 NINDIRECT。



```c
// 条件判断，处理(NDIRECT <= bn < NINDIRECT)的逻辑块号，用一级间接映射
if(bn < NINDIRECT){
  if((addr = ip->addrs[NDIRECT]) == 0)
    ip->addrs[NDIRECT] = addr = balloc(ip->dev); // 分配一个索引块
  bp = bread(ip->dev, addr); // 获取一个含有目标内容(ip->dev)的带锁缓冲区
  a = (uint*)bp->data; // a指向一级索引块的缓冲区数据的起始地址(缓冲区中保存了对应块的内容)
  if((addr = a[bn]) == 0){
    a[bn] = addr = balloc(ip->dev); // 一级间接映射
    log_write(bp);
  }
  brelse(bp); // 释放带锁缓冲区
  return addr;
}
bn -= NINDIRECT; // 更新bn，减去NINDIRECT
```

随后，通过 if 语句再对 bn 的值进行判断，以处理 NDIRECT <= bn < NINDIRECT 的逻辑块号，使用二级间接映射。大致方法与一级间接映射类似，先分配一个一级索引块，获取带锁缓冲区，再令变量 a 指向一级索引块缓冲区数据的起始地址。随后，通过语句(addr = a[bn/NINDIRECT]) == 0 查找对应条目对应数据块是否存在，如果不存在则分配一块并通过函数 log_write 记录。随后释放带锁缓冲区，并更新变量 bp 和 a 的值。随后，再通过语句(addr = a[bn/NINDIRECT]) == 0 查找对应条目对应数据块是否存在，不存在则继续分配，完成二级间接映射并通过函数 log_write 记录，随后释放带锁缓冲区并返回 addr。最后，如果 bn 大于等于 NSECONDINDIRECT，则调用函数 panic 发送报错信息。

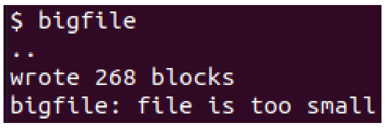


```c
  // 新增条件判断，处理(NINDIRECT <= bn < NSECONDINDIRECT)的逻辑块号，用二级间接映射
  if(bn < NSECONDINDIRECT)
  {
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev); // 分配一个索引块
    bp = bread(ip->dev, addr); // 获取一个含有目标内容(ip->dev)的带锁缓冲区
    a = (uint*)bp->data; // a指向一级索引块缓冲区数据的起始地址(缓冲区中保存了对应块的内容)

    if((addr = a[bn/NINDIRECT]) == 0) // 查找对应条目对应数据块是否存在,不存在则分配一块
    {
      a[bn/NINDIRECT] = addr = balloc(ip->dev); // 为一级索引块中的二级索引节点分配一个索引块
      log_write(bp);
    }
    brelse(bp); // 释放带锁缓冲区
    bp = bread(ip->dev, addr); // 获取一个含有目标内容(ip->dev)的带锁缓冲区
    a = (uint*)bp->data; // a指向二级索引块缓冲区数据的起始地址(缓冲区中保存了对应块的内容)

    if((addr = a[bn%NINDIRECT]) == 0) // 查找对应条目对应数据块是否存在,不存在则分配一块
    {
      a[bn%NINDIRECT] = addr = balloc(ip->dev); // 二级间接映射
      log_write(bp);
    }
    brelse(bp); // 释放带锁缓冲区
    return addr;
  }

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

接下来，根据题目给出的提示，由于操作系统的文件结构已经更改，我们需要先删除文件 fs.img，再重新输入 make 运行，以对 xv6 项目进行编译，编译完成后再输入 bigfile 进行测试即可。

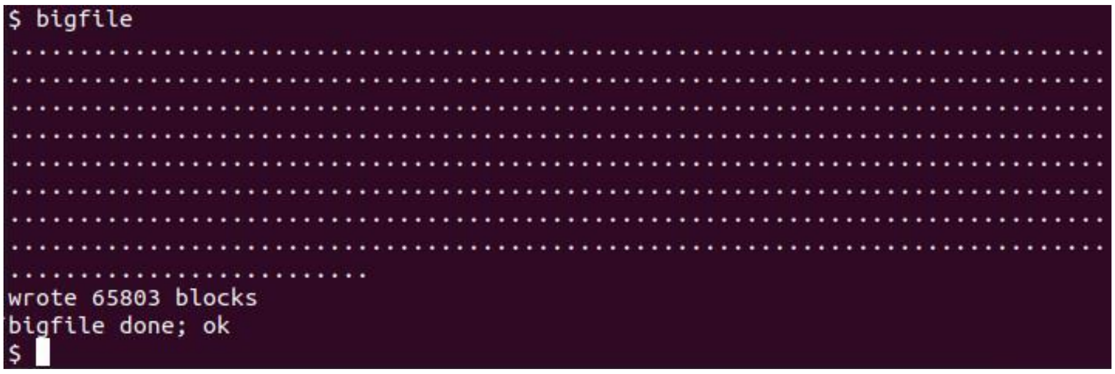
## 4.3 运行结果

修改 kernel/fs.h 和 kernel/fs.c 之前，输入 bigfile 进行测试。



```
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
```

修改 kernel/fs.h 和 kernel/fs.c 之后，输入 bigfile 进行测试。



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