哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2023 年秋季

课程名称：操作系统

实验名称：锁机制的应用

实验性质：课内实验

实验时间：2023.10.24 地点：T2507

学生班级：智能强基-计算机

学生学号：210010101

学生姓名：房煊梓

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 年 9 月

一、 回答问题

1、 内存分配器

a. 什么是内存分配器？它的作用是？

内存分配器是对内存进行分配和回收的物理内存管理器。
它的作用是对上层提供 kalloc()和 kfree()接口来管理剩余的空闲物理内存，在需要使用新内存空间的时候调用 kalloc()，在需要释放内存空间的时候调用 kfree()。

b. 内存分配器的数据结构是什么？它有哪些操作（函数），分别完成了什么功能？

(1)内存分配器的数据结构包含自旋锁 lock 和由空闲物理页组成的链表 freelist。

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

(2)kinit 函数：初始化分配器。
freerange 函数：把空闲内存页加到链表里。
kfree 函数：释放内存。首先将 void *pa 开始的物理页的内容全部置为 1，然后将这空闲页物理内存加到链表头。
kalloc 函数：申请内存。移除并返回空闲链表头的第一个元素，即给调用者分配 1 页物理内存。

c. 为什么指导书提及的优化方法可以提升性能？

多个 CPU 申请和释放页面的要求很频繁。若采取原方案，每次操作 kmem.freelist 都需先申请 kmem.lock，造成多个 CPU 同时申请一个锁，锁争用较多。若采取优化方案，每个 CPU 核使用独立的链表，每个链表都有一个锁。申请页面时先申请当前 CPU 的锁，若当前 CPU 的 freelist 中有空闲内存块则直接分配，此时不会申请其他 CPU 的锁，减少了锁争用；若无空闲块则尝试获取其他 CPU 的锁，且只需取得一个 CPU 的锁即可窃取其内存块，不会再申请其他 CPU 的锁，减少了锁争用。优化方法减少了锁争用，所以可提升性能。

2、 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存是磁盘与文件系统交互的中间层。为提升性能，将最近经常访问的磁盘块缓存在内存里，相应的内存空间即为磁盘缓存。
它的作用有：
(1)同步访问磁盘块以确保内存里每个块只有一份复制，且每次只有一个内核进程可以使用那份复制。
(2)缓存常用块，使得不必每次都从硬盘上读取它们。
(3)修改缓存块的内容后，确保磁盘中对应内容的更新。

b. buf 结构体为什么有 prev 和 next 两个成员，而不是只保留其中一个？请从这样做的优点分析（提示：结合通过这两种指针遍历链表的具体场景进行思考）。

brelse()中，获取锁之后将引用计数减去 1，若引用计数为 0，则将 block cache 脱离出链表，再将它插入到链表头部，表示这是最近使用过的 block cache，因此 next 指向最近使用过的 block cache，而 prev 则指向最不常使用的 block cache。
bget()中，查找对应块，使用 next 遍历链表，即先从最近使用过的 block cache 开始遍历，效率较高；若未命中，需要找空闲块，使用 prev 遍历链表，即先从最不常用的 block cache 开始遍历，效率较高。

c. 为什么哈希表可以提升磁盘缓存的性能？可以使用内存分配器的优化方法优化磁盘缓存吗？请说明原因。

(1)使用哈希桶，将各块块号 blockno 的某种散列值作为 key 对块分组，并为每个哈希桶分配一个专用的锁。由哈希桶来代替链表，当要获取和释放缓存块时，只需对某个哈希桶进行加锁，桶之间的操作就可并行进行，提升磁盘缓存性能。
(2)不能使用内存分配器的优化方法优化磁盘缓存。因为对于磁盘缓存，每个 buf 缓存的磁盘块是不一样的，并且每个 CPU 都可能使用任何一个块。若采用内存分配器的优化方法，即每个 CPU 使用独立的链表和锁，则当 CPU 需要使用一个块时，它必须也只能去获取该块所在链表的锁，而无法像内存分配器的情况来进行分配或窃取的操作，无法减少锁争用。

二、 实验详细设计

1.内存分配器详细设计

(1)修改内存分配器数据结构，将 kalloc 的共享 freelist 改为每个 CPU 独立的 freelist，并且建立 kmems 数组，大小为 NCPU。

```
//将kalloc的共享freelist改为每个CPU独立的freelist
struct kmem{
  struct spinlock lock;
  struct run *freelist;
};
struct kmem kmems[NCPU];
```

(2)修改 kinit()

由于原本的一个锁变为每个 CPU 独立的锁，故需要对每个 CPU 的锁都进行初始化，利用循环进行锁的初始化。而 freerange 为所有运行 freerange 的 CPU 分配空闲的内存，故不需要循环。

```
void
kinit()
{
  //initlock(&kmem.lock, "kmem");
  //循环初始化
  int i=0;
  for(i=0;i<NCPU;i++){
    initlock(&kmems[i].lock,"kmem_lock");
  }
  //freerange为所有运行freerange的CPU分配空闲的内存,不用放在循环内
  freerange(end, (void*)PHYSTOP);
}
```

(3)修改 kfree()

需要将内存块放入当前 CPU 的 freelist 中。因此，需要先关中断，再调用 cpuid()来获取当前 CPU 的 id 号。然后根据 id 号获取相应 CPU 的锁，释放内存块，然后释放相应 CPU 的锁，再开中断。

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

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  //关中断
  push_off();
  //获取当前CPU的id号
  int cpu_id = cpuid();
  //先获得锁，然后释放内存块(将内存块放入当前CPU的freelist中)，再释放锁
  acquire(&kmems[cpu_id].lock);
  r->next = kmems[cpu_id].freelist;
  kmems[cpu_id].freelist = r;
  release(&kmems[cpu_id].lock);
  //开中断
  pop_off();

}
```

(4)修改 kalloc()

根据分配内存块的机制，优先分配当前 CPU 的 freelist 中的内存块，因此需要关中断，然后调用 cpuid()来获取当前 CPU 的 id 号。然后获取当前 CPU 的锁，将其 freelist 赋给 r。检查 r，若 r 不为空，说明当前 CPU 的 freelist 有空闲块，进行分配。然后释放当前 CPU 的锁，再开中断。

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

  //关中断
  push_off();
  //优先分配当前CPU的freelist内存块，首先要获取当前CPU的id
  int now_cpu_id = cpuid();
  //先获取当前cpu的锁，然后将其freelist赋给r
  acquire(&kmems[now_cpu_id].lock);
  r = kmems[now_cpu_id].freelist;
  //如果r不为空，说明当前CPU的freelist有空闲块，分配
  if(r){
    kmems[now_cpu_id].freelist = r->next;
    //释放锁且开中断
    release(&kmems[now_cpu_id].lock);
    pop_off();
  }else{
```

若当前 CPU 没有空闲内存块，则从其他 CPU 的 freelist 中窃取内存块。做法是循环查找其他 CPU，获取其锁，检查是否有空闲内存块，若有则将空闲块窃取给当前 CPU，然后释放其他 CPU 的锁，出循环。然后释放当前 CPU 的锁，开中断。

```
}else{
  //检查其他cpu的freelist进行窃取
  //flag表示是否找到了满足条件的cpu
  int i=0,flag=0;
  for(i=0;i<NCPU;i++){
    //flag为1则跳出循环
    if(flag==1){
      break;
    }
    //检查的是非当前cpu
    if(i!=now_cpu_id){
      //获取锁，将freelist赋给r
      acquire(&kmems[i].lock);
      r=kmems[i].freelist;
      if(r){
        //r不为空，就是找到了，分配
        flag=1;
        kmems[i].freelist=r->next;
      }
      //释放锁
      release(&kmems[i].lock);
    }
  }
  //释放锁且开中断
  release(&kmems[now_cpu_id].lock);
  pop_off();
}
```

而当所有 CPU 都没有空闲块时，返回 0。

总体截图如下：

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

  //关中断
  push_off();
  //优先分配当前CPU的freelist内存块，首先要获取当前CPU的id
  int now_cpu_id = cpuid();
  //先获取当前cpu的锁，然后将其freelist赋给r
  acquire(&kmems[now_cpu_id].lock);
  r = kmems[now_cpu_id].freelist;
  //如果r不为空，说明当前CPU的freelist有空闲块，分配
  if(r){
    kmems[now_cpu_id].freelist = r->next;
    //释放锁且开中断
    release(&kmems[now_cpu_id].lock);
    pop_off();
  }else{
    //检查其他cpu的freelist进行窃取
    //flag表示是否找到了满足条件的cpu
    int i=0,flag=0;
    for(i=0;i<NCPU;i++){
      //flag为1则跳出循环
      if(flag==1){
        break;
      }
      //检查的是非当前cpu
      if(i!=now_cpu_id){
        //获取锁，将freelist赋给r
        acquire(&kmems[i].lock);
        r=kmems[i].freelist;
        if(r){
          //r不为空，就是找到了，分配
          flag=1;
          kmems[i].freelist=r->next;
        }
        //释放锁
        release(&kmems[i].lock);
      }
    }
    //释放锁且开中断
    release(&kmems[now_cpu_id].lock);
    pop_off();
  }


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

2.磁盘缓存详细设计

(1)构建磁盘缓存数据结构。构建 13 个哈希组，每个哈希组有一个 linked list 及一个 lock。并且具有一个全局大锁。

```
//构建磁盘缓存数据结构，构建13个哈希组
#define NBUCKETS 13
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;
  struct buf hashbucket[NBUCKETS];  //每个哈希队列一个linked list及一个lock
  struct spinlock total_lock;       //全局大锁
} bcache;
```

(2)修改 binit()

由于每个哈希队列有一个 linked list 及一个 lock，此外还有一个全局大锁，故需要初始化全局大锁，利用循环初始化每个哈希队列的 lock 和创建每个哈希队列的 linked list。

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

  //initlock(&bcache.lock, "bcache");
  //初始化全局大锁
  initlock(&bcache.total_lock, "bcache_total_lock");
  //循环初始化锁
  int i=0;
  for(i=0;i<NBUCKETS;i++){
    initlock(&bcache.lock[i],"bcache_lock");
  }

  // Create linked list of buffers
  //bcache.head.prev = &bcache.head;
  //bcache.head.next = &bcache.head;

  //循环创建linkedlist
  for(i=0;i<NBUCKETS;i++){
    bcache.hashbucket[i].prev = &bcache.hashbucket[i];
    bcache.hashbucket[i].next = &bcache.hashbucket[i];
  }
```

而对于块，则需要使用哈希桶，将各块块号 blockno 的某种散列值作为 key 对块进行分组，这里选择的散列方式是取余。

```
for(b = bcache.buf; b < bcache.buf+NBUF; b++){
  //使用哈希桶，将各块块号blockno的某种散列值作为key对块进行分组,这里选择取余
  int b_hash_value = b->blockno % NBUCKETS;
  b->next = bcache.hashbucket[b_hash_value].next;
  b->prev = &bcache.hashbucket[b_hash_value];
  initsleeplock(&b->lock, "buffer");
  bcache.hashbucket[b_hash_value].next->prev = b;
  bcache.hashbucket[b_hash_value].next = b;
}
```

(3)修改 bget()

首先通过取余获得当前 blockno 对应的哈希值，然后获取相应桶的锁，利用 next 指针查看相应桶中的数据块，若命中，则释放相应桶的锁，获取块的睡眠锁，并且返回结果。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  //获取当前blockno对应的哈希值
  int hash_value = blockno % NBUCKETS;

  //acquire(&bcache.lock);
  //获取相应桶的锁
  acquire(&bcache.lock[hash_value]);

  // Is the block already cached?
  //查看相应桶里的数据块
  for(b = bcache.hashbucket[hash_value].next; b != &bcache.hashbucket[hash_value]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock[hash_value]);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

若未命中，则利用 prev 指针从链表尾部开始找到最不常使用的 block cache，腾出空间以用来存放新的 block cache，并且释放相应桶的锁，获取块的睡眠锁，返回结果。

```
// Not cached.
// Recycle the least recently used (LRU) unused buffer.
//找相应桶的未使用的缓存块
for(b = bcache.hashbucket[hash_value].prev; b != &bcache.hashbucket[hash_value]; b = b->prev){
    if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        release(&bcache.lock[hash_value]);
        acquiresleep(&b->lock);
        return b;
    }
}
```

若还未找到，则需要从其他哈希桶找一个未被使用的缓存块，移入到相应的哈希桶链表中使用。首先释放原有桶的锁，再获取全局大锁，之后获取原有桶的锁，再循环检查其他桶。检查其他桶时，先获取其他桶的锁，然后利用 prev 搜索其未使用的缓存块，若找到了，则需要将其移入相应哈希桶链表中使用，并按顺序释放锁。

```
//还没找到，需要从其他哈希桶找一个未被使用的缓存块，移入到相应的哈希桶链表中使用
//先释放这个锁
release(&bcache.lock[hash_value]);
//再获取全局大锁
acquire(&bcache.total_lock);
//获取原有桶的锁
acquire(&bcache.lock[hash_value]);
int i=0;
for(i=0;i<NBUCKETS;i++){
  //检查的是其他桶
  if(i!=hash_value){
    //获取新桶的锁
    acquire(&bcache.lock[i]);
    //搜索缓存块且为未命中的缓存分配一个新条目，是原子操作
    for(b = bcache.hashbucket[i].prev; b != &bcache.hashbucket[i]; b = b->prev){
      //找到一个未使用的缓存块
      if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        //注意如果b移走，链表的指针需要改动
        b->prev->next = b->next;
        b->next->prev = b->prev;
        //释放其他桶的锁
        release(&bcache.lock[i]);
        //需要把未使用的缓存块移入相应哈希桶链表中使用
        b->next = bcache.hashbucket[hash_value].next;
        b->prev = &bcache.hashbucket[hash_value];
        bcache.hashbucket[hash_value].next->prev = b;
        bcache.hashbucket[hash_value].next = b;
        //释放原有桶的锁
        release(&bcache.lock[hash_value]);
        acquiresleep(&b->lock);
        //释放全局大锁
        release(&bcache.total_lock);
        return b;
      }
    }
    //释放新桶的锁
    release(&bcache.lock[i]);
  }
}
//释放全局大锁
release(&bcache.total_lock);
panic("bget: no buffers");
```

(4)修改 brelse()

通过取余获得当前 blockno 对应的哈希值，然后获取相应桶的锁，释放相应桶的块即可，最后释放相应桶的锁。

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

  releasesleep(&b->lock);

  //相应的桶
  int hash_value = b->blockno % NBUCKETS;
  acquire(&bcache.lock[hash_value]);
  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[hash_value].next;
    b->prev = &bcache.hashbucket[hash_value];
    bcache.hashbucket[hash_value].next->prev = b;
    bcache.hashbucket[hash_value].next = b;
  }

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

(5)修改 bpin()

只需要修改为对相应桶的操作即可。

```
void
bpin(struct buf *b) {
  //相应的桶
  int hash_value = b->blockno % NBUCKETS;
  acquire(&bcache.lock[hash_value]);
  b->refcnt++;
  release(&bcache.lock[hash_value]);
}
```

(6)修改 bunpin()

只需要修改为对相应桶的操作即可。

```
void
bunpin(struct buf *b) {
  //相应的桶
  int hash_value = b->blockno % NBUCKETS;
  acquire(&bcache.lock[hash_value]);
  b->refcnt--;
  release(&bcache.lock[hash_value]);
}
```

三、 实验结果截图

1.kalloctest 截图

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem_lock: #fetch-and-add 0 #acquire() 47144
lock: kmem_lock: #fetch-and-add 0 #acquire() 196845
lock: kmem_lock: #fetch-and-add 0 #acquire() 189050
lock: bcache_total_lock: #fetch-and-add 0 #acquire() 9
lock: bcache_lock: #fetch-and-add 0 #acquire() 13
lock: bcache_lock: #fetch-and-add 0 #acquire() 5
lock: bcache_lock: #fetch-and-add 0 #acquire() 7
lock: bcache_lock: #fetch-and-add 0 #acquire() 13
lock: bcache_lock: #fetch-and-add 0 #acquire() 11
lock: bcache_lock: #fetch-and-add 0 #acquire() 283
lock: bcache_lock: #fetch-and-add 0 #acquire() 3
lock: bcache_lock: #fetch-and-add 0 #acquire() 3
lock: bcache_lock: #fetch-and-add 0 #acquire() 3
lock: bcache_lock: #fetch-and-add 0 #acquire() 11
--- top 5 contended locks:
lock: proc: #fetch-and-add 23249 #acquire() 161060
lock: virtio_disk: #fetch-and-add 9709 #acquire() 66
lock: proc: #fetch-and-add 6073 #acquire() 161140
lock: proc: #fetch-and-add 2468 #acquire() 161145
lock: proc: #fetch-and-add 1141 #acquire() 161045
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$
```

2.bcachetest 截图

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem_lock: #fetch-and-add 0 #acquire() 243452
lock: kmem_lock: #fetch-and-add 0 #acquire() 1899424
lock: kmem_lock: #fetch-and-add 0 #acquire() 1913831
lock: kmem_lock: #fetch-and-add 0 #acquire() 51
lock: kmem_lock: #fetch-and-add 0 #acquire() 51
lock: kmem_lock: #fetch-and-add 0 #acquire() 51
lock: kmem_lock: #fetch-and-add 0 #acquire() 51
lock: kmem_lock: #fetch-and-add 0 #acquire() 51
lock: bcache_total_lock: #fetch-and-add 0 #acquire() 18
lock: bcache_lock: #fetch-and-add 0 #acquire() 6196
lock: bcache_lock: #fetch-and-add 0 #acquire() 6179
lock: bcache_lock: #fetch-and-add 0 #acquire() 6326
lock: bcache_lock: #fetch-and-add 0 #acquire() 6320
lock: bcache_lock: #fetch-and-add 0 #acquire() 6326
lock: bcache_lock: #fetch-and-add 0 #acquire() 6310
lock: bcache_lock: #fetch-and-add 0 #acquire() 4540
lock: bcache_lock: #fetch-and-add 0 #acquire() 4556
lock: bcache_lock: #fetch-and-add 0 #acquire() 2999
lock: bcache_lock: #fetch-and-add 0 #acquire() 4119
lock: bcache_lock: #fetch-and-add 0 #acquire() 2111
lock: bcache_lock: #fetch-and-add 0 #acquire() 4119
lock: bcache_lock: #fetch-and-add 0 #acquire() 4177
--- top 5 contended locks:
lock: proc: #fetch-and-add 220471 #acquire() 3841759
lock: proc: #fetch-and-add 186875 #acquire() 3841759
lock: proc: #fetch-and-add 185941 #acquire() 3841758
lock: proc: #fetch-and-add 170385 #acquire() 3841758
lock: proc: #fetch-and-add 169009 #acquire() 3841759
tot= 0
test0: OK
start test1
test1 OK
$
```

3.usertests 截图

```
$ 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=3249
            sepc=0x00000000000056b0 stval=0x00000000000056b0
usertrap(): unexpected scause 0x000000000000000c pid=3250
            sepc=0x00000000000056b0 stval=0x00000000000056b0
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test dirtest: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000215c stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000215c stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000215c stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000215c stval=0x0000000080024 9f0
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000215c stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6235
            sepc=0x000000000000215c stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6236
            sepc=0x000000000000215c stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6237
            sepc=0x000000000000215c stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6238
            sepc=0x000000000000215c stval=0x000000008061a80
usertrap(): unexpected scause 0x000000000000000d pid=6239
            sepc=0x000000000000215c stval=0x0000000080066ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6240
            sepc=0x000000000000215c stval=0x0000000080007a120
usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x000000000000215c stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6242
            sepc=0x000000000000215c stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6243
            sepc=0x000000000000215c stval=0x000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6244
            sepc=0x000000000000215c stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6245
            sepc=0x000000000000215c stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x000000000000215c stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x000000000000215c stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6248
            sepc=0x000000000000215c stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6249
            sepc=0x000000000000215c stval=0x00000000800e7ef0
usertrap(): unexpected scause 0x000000000000000d pid=6250
            sepc=0x000000000000215c stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6251
            sepc=0x000000000000215c stval=0x0000000080100590
usertrap(): unexpected scause 0x000000000000000d pid=6252
            sepc=0x000000000000215c stval=0x000000008010c8e0
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x000000000000215c stval=0x0000000080118c30
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000215c stval=0x0000000080124f80
usertrap(): unexpected scause 0x000000000000000d pid=6255
            sepc=0x000000000000215c stval=0x0000000081312d0
usertrap(): unexpected scause 0x000000000000000d pid=6256
            sepc=0x000000000000215c stval=0x000000008013d620
usertrap(): unexpected scause 0x000000000000000d pid=6257
            sepc=0x000000000000215c stval=0x0000000080149970
usertrap(): unexpected scause 0x000000000000000d pid=6258
            sepc=0x000000000000215c stval=0x00000000801 55cc0
usertrap(): unexpected scause 0x000000000000000d pid=6259
            sepc=0x000000000000215c stval=0x0000000080162010
usertrap(): unexpected scause 0x000000000000000d pid=6260
            sepc=0x000000000000215c stval=0x000000008016e360
usertrap(): unexpected scause 0x000000000000000d pid=6261
            sepc=0x000000000000215c stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6262
            sepc=0x000000000000215c stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6263
            sepc=0x000000000000215c stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6264
            sepc=0x000000000000215c stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6265
            sepc=0x000000000000215c stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6266
            sepc=0x000000000000215c stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6267
            sepc=0x000000000000215c stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6268
            sepc=0x000000000000215c stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6269
            sepc=0x000000000000215c stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6281
            sepc=0x00000000000041f8 stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6285
            sepc=0x00000000000022cc stval=0x00000000000fb90
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
$
```

4.make grade 截图

```
== Test running kalloctest ==
$ make qemu-gdb
(119.5s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (11.6s)
== Test running bcachetest ==
$ make qemu-gdb
(9.4s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (148.9s)
== Test time ==
time: OK
Score: 70/70
210010101@comp1:~/xv6-oslab23-hitsz$
```