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

实验设计报告

开课学期：2022 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：10.21 地点：T2507

学生班级：5 班

学生学号：200110513

学生姓名：宗晴

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2022 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（allocator）定义在 kernel/kalloc.c 中，在 xv6 内核刚启动时，从 kernel 结束地址（end）一直到 PHYSTOP 的空间都是空闲的，内存分配器是对上层提供 kalloc()和 kfree()接口来管理这段剩余的空闲物理内存的模块。

它的作用是将从 kernel 结束地址（end）一直到 PHYSTOP 的物理内存资源划分为 4096 字节大小的页，保存所有空闲页，释放闲置的内存页，为内存不足的进程分配空闲页，进而减少内部碎片。

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？

内存分配器的数据结构如下：

```
struct run {
  struct run *next;
};

struct kmem{
  struct spinlock lock;
  struct run *freelist;
};
struct kmem kmems[NCPU];// 使每个CPU核使用独立的链表
```

主要包括用于分配物理内存页的链表以及用来保护该链表的自旋锁。

它有 kinit()、freerange(void *pa_start, void *pa_end)、kfree(void *pa)、kalloc(void) 操作（函数）。

kinit()：用于初始化内存分配器，包括初始化自旋锁，以及利用 freerange(end, (void*)PHYSTOP)给所有运行 freerange 的 CPU 分配空闲的内存，即将从 kernel 结束地址（end）一直到 PHYSTOP 的内存页链添加到对应 CPU 的空闲页链表 freelist 的表头。

freerange(void *pa_start, void *pa_end)：释放从 pa_start 直到 pa_end 的内存页，通过调用 kfree(p)将这些物理页的内容全部置为 1，然后将这些空闲页加到对应 CPU 的空闲页链表 freelist 的表头。

kfree(void *pa)：释放 pa 所指向的内存页，将该内存页的内容全部置 1，然后将它添加到当前 CPU 的空闲页链表 freelist 的表头。

kalloc(void)：用于申请内存，给内存不足的进程分配一页内存，返回指向空闲页的指针，同时将该页从空闲页链表 freelist 中摘下。

c. 为什么指导书提及的优化方法可以提升性能？

在原始的方案中，kalloc 在 kalloc()和 kfree()中对空闲页链表 freelist 的操作进行了上锁，从而保护了空闲页链表 freelist，避免重复操作以及内存页丢失。但在多 CPU 并行时，会频繁出现多个 CPU 争抢同一把锁的情况，因此降低了并发性，从而降低了性能。而指导书提及的优化方法，使每个 CPU 核使用独立的链表 ，而不是原先的共享链表，这样就降低了多个 CPU 争抢同一把锁的频率，从而提升了进程的并发性，也提升了性能。

2、 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存（Buffer Cache）定义在 kernel/bio.c 中，它是最近经常访问的磁盘块在内存里的复制，是磁盘与文件系统交互的中间层。

作用：由于 xv6 的文件系统是以磁盘数据块为单位从磁盘读写数据的，并且对磁盘的读取非常慢，而内存的速度要快得多，所以磁盘缓存即将最近经常访问的磁盘块缓存在内存里可以避免 CPU 频繁访问磁盘，提高读写效率，从而提升性能（此时内存起到 cache 的作用）。

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）。

这样做便于对磁盘缓存链表的各种操作。prev 和 next 两个成员意味着该磁盘缓存链表为双向链表，相比于只保留其中一个，双向链表可以非常方便的正向和逆向查找链表中的缓存块。由于原始方案中，磁盘缓存链表按照最后一次被访问时间的早晚，以及缓存块是否空闲来排序，因此在查找空闲磁盘块时需要从后向前查找，并且在需要移动或从链表中移出缓存块时，需要指向其前后缓存块的指针，所以双向链表更为便捷，且很大程度上降低了时间开销。

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。

在原始方案中，所有磁盘缓存都被组织在同一链表中，并且由一把大锁加以保护，因此多个进程无法并发地修改磁盘缓存，它们的请求只能被顺序地处理，因此大大降低了性能。而哈希表将各块块号 blockno 的某种散列值作为 key 对块进行分组，并为每个哈希桶分配一个专用的锁。当要获取和释放缓存块时，只需要对对应的哈希桶进行加锁，桶之间的操作可以并行，提升了磁盘缓存的性能。

不可以使用内存分配器的优化方法优化磁盘缓存，因为某一内存页是被其对应

CPU 单独使用的，而磁盘缓存对于所有 CPU 则是共享的，若为每个 CPU 单独分配磁盘缓存则会造成极大的浪费。

二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写

1、内存分配器（Memory Allocator）

对于每个 CPU 维护一个独立的空闲页面链表，以及其对应的锁。若某 CPU 的空闲页面已经用完，则向其它 CPU 的空闲页面链表中窃取空闲页。

为此需修改 kernel/kalloc.c 中的对应代码。

首先创建 kmems 数组，用于存储每个 CPU 独立的 kmem：

```
struct run {
  struct run *next;
};

struct kmem{
  struct spinlock lock;
  struct run *freelist;
};
struct kmem kmems[NCPU];// 使每个CPU核使用独立的链表
```

同时对这些 CPU 对应的锁命名，由于最多有 8 个 CPU，因此定义 8 个锁名称即可：

```
// NCPU大小为8，因此定义8个以kmem开头的锁名称
char *lock_names[8] = {"kmem0", "kmem1", "kmem2", "kmem3", "kmem4", "kmem5", "kmem6", "kmem7"};
```

修改 kinit()函数，使其初始化每个 CPU 对应的锁：

```
for(int i = 0; i < NCPU; i++)
  initlock(&(kmems[i].lock), lock_names[i]);
```

在释放某一内存页时，需要将该内存页添加到对应 CPU 的空闲页链表中：

```
// 获取当前CPU序号，在调用cpuid时需要保证当前进程不可被中断
push_off();
int cpu_id = cpuid();
pop_off();

// 将对应的空闲页放入空闲列表中
acquire(&kmems[cpu_id].lock);
r->next = kmems[cpu_id].freelist;
kmems[cpu_id].freelist = r;
release(&kmems[cpu_id].lock);
```

如上图，首先获取当前的 CPU id，在调用 cpuid()时需要使用 push_off()和 pop_off()保证当前进程不可被中断，然后获取该 CPU 对应的锁，将待释放的内存页放入该 CPU 对应的空闲页链表 freelist 中，最后释放锁。

最后修改 kalloc(void)函数，首先获取当前的 CPU id，同样该过程不可被中断：

```
push_off();
int cpu_id = cpuid();
pop_off();
```

然后，获取当前 CPU 对应的锁，查看其空闲页链表中是否还有空闲页，若有则将该空闲页 r 从空闲页链表中移出，然后释放该锁：

```
acquire(&kmems[cpu_id].lock);
r = kmems[cpu_id].freelist;
if(r){
  kmems[cpu_id].freelist = r->next;
  release(&kmems[cpu_id].lock);
}
```

若当前 CPU 的空闲页链表已空，则需要从其他 CPU 的空闲页链表中窃取空闲页：

```
else{  // 此时需要从其他CPU的freelist中窃取内存块
  // 先释放当前CPU对应的锁，从而避免死锁
  release(&kmems[cpu_id].lock);
  for(int i = 0; i < NCPU; i++){
    if(i == cpu_id) continue;  // 如果是当前CPU则跳过
    acquire(&kmems[i].lock);
    r = kmems[i].freelist;  // 访问该CPU的freelist
    if(r){  // 窃取1页内存
      kmems[i].freelist = r->next;
      release(&kmems[i].lock);
      break;  // 窃取成功则停止
    } else release(&kmems[i].lock);
  }
}
```

如上图，首先应该释放当前 CPU 的锁，从而避免两个 CPU 持有各自的锁，同时想要获取对方的锁的情况，避免死锁。然后遍历所有 CPU_id，若是当前 CPU 则跳过，否则获取该 CPU 的锁，并且访问该 CPU 的空闲页链表，查看是否有空闲页。若有空闲页，则窃取一页内存，将该空闲页从链表中移出，然后释放该 CPU 的锁，并且跳出循环。若没有空闲页，则释放该 CPU 的锁，然后继续访问下一个 CPU 是空闲页链表。

循环结束后，判断是否成功获取到一页空闲页，即判断 r 是否为空。若非空，代表成功获取到一页内存页，将该内存页清空后，返回指针 r，否则 kalloc 失败，已无空闲内存页。

2、磁盘缓存（Buffer Cache）

同时使用哈希表以及时间戳进行优化。修改 kernel/bio.c 中的对应代码。

取哈希桶的数量为质数 13，以减少哈希征用，将各块块号 blockno 模 13 作为 key 对块进行分组，并为每个哈希桶分配一个专用的锁；在 buf 数据结构中增加 `uint time;  // 时间戳` 用于表示时间戳，同时在 kernel/bio.c 中通过 kernel/trap.c 中的 ticks 函数获得时间戳：

```
extern uint ticks;  // 时间戳，在kernel/trap.c中被不断更新
```

修改 bcache 数据结构如下：

```
struct {
  // 全局锁，只在窃取其它桶的内存块时使用，并不会影响并发效率
  struct spinlock lock;
  struct spinlock bucketlocks[NBUCKETS];  // 每个哈希桶的锁
  struct buf buf[NBUF];
  struct buf hashbucket[NBUCKETS]; //每个哈希桶对应一个链表
} bcache;
```

保留全局锁用于保证窃取缓存块操作的原子性，同时建立 NBUCKETS=13 个哈希桶，每个哈希桶有一个独立的链表和对应的锁。

定义哈希函数如下：

```
uint
hash(uint n)
{
  return n % NBUCKETS;
}
```

首先修改 binit(void)函数，初始化全局锁，初始化每个哈希桶的锁，以及构建每个哈希桶的硬盘缓存双向链表：

```
// 初始化全局锁
initlock(&bcache.lock, "bcache");

for(int i = 0; i < NBUCKETS; i++){
  // 初始化每个哈希桶的锁
  initlock(&bcache.bucketlocks[i], "bcache.bucket");

  // Create linked list of buffers
  bcache.hashbucket[i].prev = &bcache.hashbucket[i];
  bcache.hashbucket[i].next = &bcache.hashbucket[i];
}
```

然后将 NBUF 个缓存块通过哈希函数，按照顺序均匀地分配到每个桶里，利用头插法将各缓存块插入链表的头部，并且设置各缓存块的时间戳为 0。

```
// 初始化哈希链表，将缓存块均匀分配到每个桶里
for(int i = 0; i < NBUF; i++){
    uint key = hash(i);
    b = &bcache.buf[i];
    b->next = bcache.hashbucket[key].next;
    b->prev = &bcache.hashbucket[key];
    initsleeplock(&b->lock, "buffer");
    bcache.hashbucket[key].next->prev = b;
    bcache.hashbucket[key].next = b;
    b->time = 0;
}
```

然后修改 bget(uint dev, uint blockno)函数，由于涉及多种死锁以及未命中

时重复加载磁盘块进入缓存的情况，所以该函数的设计较为复杂。整体分为 3 个部分：首先在自己的哈希桶中查找对应磁盘块是否命中；若命中则返回指向该缓存块的指针，若未命中则在自己的桶中找到最后一次被访问时间最早的空闲缓存块；若存在，则修改该缓存块的相关参数，同时返回指向它的指针，若不存在则去别的哈希桶中找到最后一次被访问时间最早的空闲缓存块，将其移到当前的哈希桶中，同时修改该缓存块的相关参数，并返回指向它的指针，若仍不存在，则失败。

首先在自己的哈希桶中查找对应磁盘块是否命中。通过哈希函数获取该磁盘块所在哈希桶的 key，然后获取该哈希桶的锁，从头开始遍历磁盘缓存块链表，若命中，即 dev 和 blockno 均正确，则将其的引用率加一，然后释放哈希桶的锁，将该缓存块锁住之后返回指向该缓存块的指针：

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

  // 获取对应哈希桶的锁
  uint key = hash(blockno);
  acquire(&bcache.bucketlocks[key]);

  // Is the block already cached?
  for(b = bcache.hashbucket[key].next; b != &bcache.hashbucket[key]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.bucketlocks[key]);  // 释放对应哈希桶的锁
      acquiresleep(&b->lock);
      return b;
    }
  }
```

若未命中，则在自己的桶中找到最后一次被访问时间最早的空闲缓存块。设定一个 first_unused_flag 标志，用于判断是否已经找到第一个空闲缓存块，初始值为 1。用 LRU_b 存储最后一次被访问时间最早的空闲缓存块，LRU_time 存储该空闲缓存块的被访问时间。由于此时有时间戳，因此顺序查找即可。遍历桶中的缓存块，查看被引用量 refcnt 是否为 0，为 0 代表是空闲缓存块，若找到一个这样的缓存块并且 first_unused_flag 标志为 1，则说明这是第一个找到的空闲缓存块，将 first_unused_flag 标志置 0，并且用它的时间戳初始化 LRU_time。继续循环，若又找到一个空闲缓存块，由于 first_unused_flag 标志为 0，说明之前已经找到过空闲缓存块，那么将其最后一次被访问时间与 LRU_time 比较，若更早，则将 LRU_b 和 LRU_time 替换为当前的空闲缓存块对应的指针和时间。

遍历结束后，查看 first_unused_flag 标志，若为 0，说明自己的桶中存在空闲缓存块，修改找到的最后一次被访问时间最早的空闲缓存块的 dev、blockno、valid、refcnt、time 等参数，然后释放当前哈希桶的锁，将该缓存块锁住之后返回指向该缓存块的指针：

```
// Not cached.
// Recycle the least recently used (LRU) unused buffer.
struct buf *LRU_b = b;  // the least recently used (LRU) unused buffer
uint LRU_time;  // the least recently used (LRU) time
int first_unused_flag = 1;  // 判断是否查找到第一个unused buffer，用于给LRU_time赋初值
// 先查找当前哈希桶内的。
// 此时有时间戳，因此无需从后向前查找
for(b = bcache.hashbucket[key].next; b != &bcache.hashbucket[key]; b = b->next){
  if(b->refcnt == 0) {
    if(first_unused_flag){
      first_unused_flag = 0;
      LRU_time = b->time;
      LRU_b = b;
    } else{
      if(b->time < LRU_time){
        LRU_time = b->time;
        LRU_b = b;
      }
    }
  }
}
if(!first_unused_flag){  // flag已被修改，说明已找到一个unused buffer
  LRU_b->dev = dev;
  LRU_b->blockno = blockno;
  LRU_b->valid = 0;
  LRU_b->refcnt = 1;
  LRU_b->time = ticks;
  release(&bcache.bucketlocks[key]);  // 释放对应哈希桶的锁
  acquiresleep(&LRU_b->lock);
  return LRU_b;
}
```

若 first_unused_flag 标志为 1，说明自己的桶中不存在空闲缓存块，此时需要去别的哈希桶中找到最后一次被访问时间最早的空闲缓存块，将其移到当前的哈希桶中。

此处我们需要锁上对于全体哈希桶的全局锁，下面阐述为何需要该锁：若不使用该锁，则首先由于我们需要获取其它哈希桶的锁，因此我们要先释放掉当前哈希桶的锁，否则当两个进程持有各自的锁，同时要获取对方的锁时会发生死锁；但若当前进程 P1 先释放掉当前哈希桶的锁之后发生进程切换，另一进程 P2 也要读取同样的磁盘块时，P2 也会需要从其它哈希桶里窃取缓存块到当前哈希桶中，然而当切换回前一进程 P1 继续执行时，P1 会继续窃取内存到同一哈希桶中，因此会使得该哈希桶中出现两块缓存存储了相同的磁盘块内容，从而产生错误。因此为保证不出现重复操作，我们需要让窃取其它哈希桶中的缓存块这一操作保持原子性，所以需要锁上对于全体哈希桶的全局锁。

但若在持有当前哈希桶的锁的基础上获取全局锁，那么其它进程在持有它们对应哈希桶的锁的基础上，会被阻塞在获取全局锁这一步，因此当前进程即使获取了全局锁，也会被阻塞在获取其它桶的锁这一步，这样便产生了死锁。所以需要先释放当前哈希桶的锁，然后再获取全局锁。由于全局锁只在窃取其它桶中的缓存块时使用，因此并不会影响并发效率。

获取全局锁之后，我们还需要再次检查所需磁盘块在当前哈希桶中是否命中，

下面阐述原因：若当前进程 P1 在释放掉当前哈希桶的锁之后发生进程切换，而进程 P2 要访问同一磁盘块，则有可能在进程 P2 中从别的桶里窃取到缓存块到当前桶里，且将磁盘块的内容复制到了缓存块中，此时再切换回进程 P1，若不再次检查该磁盘块是否命中，则有可能再一次将保存了该磁盘块内容的缓存块放到当前桶中，造成重复。所以为避免这种情况，我们需要在获取全局锁之后，再次检查所需磁盘块在当前哈希桶中是否命中：

```
release(&bcache.bucketlocks[key]);  // 释放对应哈希桶的锁
// Still not cached.
// 获取全局锁，保证为未命中的缓存分配一个新的条目的操作是原子性的
// 仅在挪用不同哈希桶之间的内存块时使用到了全局锁，因此并不影响其余进程的并行
acquire(&bcache.lock);
// 再次判断是否命中，两个进程并行时，防止为同一未命中的缓存分配新的条目的操作重复
// 从而避免在一个桶里加入两个相同的缓存块
acquire(&bcache.bucketlocks[key]);
// Is the block already cached?
for(b = bcache.hashbucket[key].next; b != &bcache.hashbucket[key]; b = b->next){
  if(b->dev == dev && b->blockno == blockno){
    b->refcnt++;
    release(&bcache.bucketlocks[key]);  // 释放对应哈希桶的锁
    acquiresleep(&b->lock);
    return b;
  }
}
release(&bcache.bucketlocks[key]);  // 释放对应哈希桶的锁
```

然后我们再从其它哈希桶中窃取内存。用 first_unused_flag 标志判断是否已经找到第一个空闲缓存块，用 LRU_b 存储最后一次被访问时间最早的空闲缓存块，LRU_time 存储该空闲缓存块的被访问时间。再增加一个 LRU_hash 用于存储找到的空闲缓存块在哪一个哈希桶里。与找自己桶里的空闲缓存块类似，跳过当前哈希桶，遍历其它哈希桶里的所有内存块，找到最后一次被访问时间最早的空闲缓存块。过程类似便不再赘述：

```
// 再查找其余哈希桶内的。
int LRU_hash = key;  // LRU unused buffer所在的哈希桶
for(int i = 0; i < NBUCKETS; i++){
  if(i == key) continue;
  acquire(&bcache.bucketlocks[i]);  // 获取该哈希桶的锁
  for(b = bcache.hashbucket[i].next; b != &bcache.hashbucket[i]; b = b->next){
    if(b->refcnt == 0) {
      if(first_unused_flag){
        first_unused_flag = 0;
        LRU_time = b->time;
        LRU_b = b;
        LRU_hash = i;
      } else{
        if(b->time < LRU_time){
          LRU_time = b->time;
          LRU_b = b;
          LRU_hash = i;
        }
      }
    }
  }
  release(&bcache.bucketlocks[i]);    // 释放该哈希桶的锁
}
```

遍历完后，判断是否找到了空闲缓存块。若找到，则需要将该缓存块从其对应的哈希桶中移出，获取编号为 LRU_hash 的哈希桶的锁，给该缓存块赋上对应的值，然后将其从当前的双向循环队列中移出，释放该哈希桶的锁。然后我们获取原始哈希桶的锁，将其添加到原双向循环队列的队头，释放原哈希桶的锁。释放全局锁，然后将该缓存块的锁置 1 后，返回指向它的指针。

若未找到，则意味着已经没有空闲缓存块了，释放全局锁，然后进入 panic：

```
// 查看是否找到空闲缓存块
if(!first_unused_flag){  // flag已被修改，说明已找到一个unused buffer
  acquire(&bcache.bucketlocks[LRU_hash]);  // 获取LRU unused buffer所在的哈希桶的锁
  LRU_b->dev = dev;
  LRU_b->blockno = blockno;
  LRU_b->valid = 0;
  LRU_b->refcnt = 1;
  LRU_b->time = ticks;
  LRU_b->prev->next = LRU_b->next;  // 将该缓存块从当前哈希桶中移出
  LRU_b->next->prev = LRU_b->prev;
  release(&bcache.bucketlocks[LRU_hash]);  // 释放该哈希桶的锁

  acquire(&bcache.bucketlocks[key]);  // 获取key所在的哈希桶的锁
  bcache.hashbucket[key].next->prev = LRU_b;  // 将该缓存块移到key所在的哈希桶内
  LRU_b->next = bcache.hashbucket[key].next;
  LRU_b->prev = &bcache.hashbucket[key];
  bcache.hashbucket[key].next = LRU_b;
  release(&bcache.bucketlocks[key]);  // 释放key所在的哈希桶的锁

  release(&bcache.lock);  // 释放全局锁
  acquiresleep(&LRU_b->lock);
  return LRU_b;
}

release(&bcache.lock);  // 释放全局锁
panic("bget: no buffers");
```

接着 brelse(struct buf *b)函数，由于使用了时间戳，因此便不需要将释放的缓存块移动到队头。通过哈希函数获取该缓存块对应的哈希桶编号，获取该桶的锁，然后将该缓存块的 refcnt 减一，若 refcnt 减为了 0，代表此时没有进程在使用该缓存块，那么该缓存块可以被释放，此时只需修改缓存块的时间为当前时间戳即可，最后释放该哈希桶的锁：

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

  releasesleep(&b->lock);

  uint key = hash(b->blockno);
  acquire(&bcache.bucketlocks[key]);

  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.

    // 有时间戳后便不需要将释放的缓存块移动到队头
    // b->next->prev = b->prev;
    // b->prev->next = b->next;
    // b->next = bcache.head.next;
    // b->prev = &bcache.head;
    // bcache.head.next->prev = b;
    // bcache.head.next = b;

    // 修改时间戳
    b->time = ticks;
  }

  release(&bcache.bucketlocks[key]);
}
```

对于 bpin(struct buf *b)和 bunpin(struct buf *b)函数，需要通过哈希函数获取缓存块对应的哈希桶编号，在获取所在哈希桶的锁的基础上对缓存块的 refcnt 参数进行相应操作，然后释放哈希桶的锁：

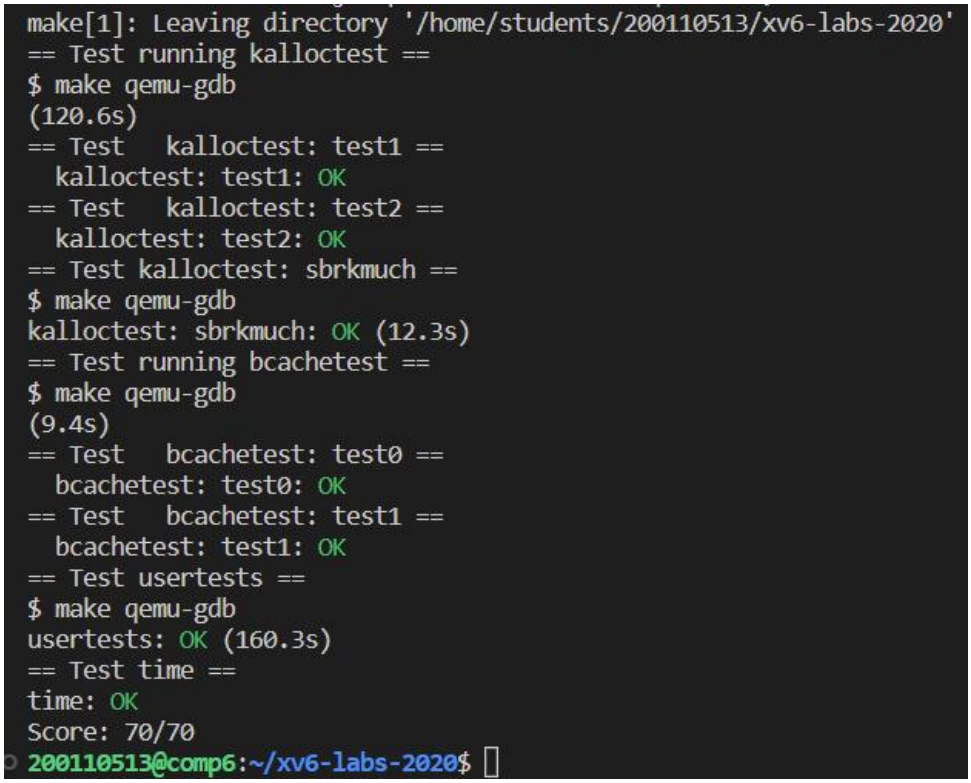
```
void
bpin(struct buf *b) {
  uint key = hash(b->blockno);
  acquire(&bcache.bucketlocks[key]);
  b->refcnt++;
  release(&bcache.bucketlocks[key]);
}

void
bunpin(struct buf *b) {
  uint key = hash(b->blockno);
  acquire(&bcache.bucketlocks[key]);
  b->refcnt--;
  release(&bcache.bucketlocks[key]);
}
```

三、 实验结果截图

请填写

make grade:



kalloctest:

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem0: #fetch-and-add 0 #acquire() 45192
lock: kmem1: #fetch-and-add 0 #acquire() 193602
lock: kmem2: #fetch-and-add 0 #acquire() 194302
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8
lock: bcache.bucket: #fetch-and-add 0 #acquire() 10
lock: bcache.bucket: #fetch-and-add 0 #acquire() 10
lock: bcache.bucket: #fetch-and-add 0 #acquire() 282
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #fetch-and-add 23301 #acquire() 127089
lock: virtio_disk: #fetch-and-add 11340 #acquire() 57
lock: proc: #fetch-and-add 6588 #acquire() 127171
lock: proc: #fetch-and-add 3642 #acquire() 127171
lock: pr: #fetch-and-add 2384 #acquire() 5
tot= 0
test1 OK
start test2
total free number of pages: 32495 (out of 32768)
.....
test2 OK
$ bcachetest
```

bcachetest:

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem0: #fetch-and-add 0 #acquire() 261750
lock: kmem1: #fetch-and-add 0 #acquire() 1972128
lock: kmem2: #fetch-and-add 0 #acquire() 2028000
lock: kmem3: #fetch-and-add 0 #acquire() 51
lock: kmem4: #fetch-and-add 0 #acquire() 51
lock: kmem5: #fetch-and-add 0 #acquire() 51
lock: kmem6: #fetch-and-add 0 #acquire() 51
lock: kmem7: #fetch-and-add 0 #acquire() 51
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6180
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6178
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4266
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4262
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2258
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4256
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2526
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4552
lock: bcache.bucket: #fetch-and-add 0 #acquire() 5056
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6180
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6178
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6176
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6174
--- top 5 contended locks:
lock: proc: #fetch-and-add 252793 #acquire() 3185640
lock: proc: #fetch-and-add 210192 #acquire() 3185640
lock: proc: #fetch-and-add 196867 #acquire() 3185640
lock: proc: #fetch-and-add 140194 #acquire() 3185641
lock: virtio_disk: #fetch-and-add 129529 #acquire() 1110
tot= 0
test0: OK
start test1
test1 OK
$ usertests
```

usertests:

```
test1 OK
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3249
            sepc=0x000000000000569a stval=0x000000000000569a
usertrap(): unexpected scause 0x000000000000000c pid=3250
            sepc=0x000000000000569a stval=0x000000000000569a
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 exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6216
            sepc=0x000000000000215c stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6217
            sepc=0x000000000000215c stval=0x0000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6218
            sepc=0x000000000000215c stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6219
            sepc=0x000000000000215c stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6220
            sepc=0x000000000000215c stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6221
            sepc=0x000000000000215c stval=0x0000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6222
            sepc=0x000000000000215c stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6223
            sepc=0x000000000000215c stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6224
            sepc=0x000000000000215c stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6225
            sepc=0x000000000000215c stval=0x0000000008006ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6226
            sepc=0x000000000000215c stval=0x0000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6227
            sepc=0x000000000000215c stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000215c stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000215c stval=0x0000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000215c stval=0x000000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000215c stval=0x000000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6232
```

……

```
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000215c stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6255
            sepc=0x000000000000215c stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6267
            sepc=0x00000000000041f8 stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6271
            sepc=0x00000000000022cc stval=0x000000000000fb90
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```