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

实验设计报告

开课学期：大三上

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：10.29 地点：T2210

学生班级：1901105

学生学号：190110509

学生姓名：王铭

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2018 年 12 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？

内存分配器就是对物理内存进行管理的一个结构，它提供了接口给调用者完成申请或者释放内存的需求，调用者不需要知道如何维护内存，就可以利用接口完成对内存的使用。

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为：

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

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

其中 freelist 为空闲页链表的头节点，lock 为保护这个空闲页链表的自旋锁。

内存分配器主要有释放内存和申请内存两个操作。

①释放内存即 kfree(void *pa)函数，传入的参数为要释放的物理页号，即物理页的首地址，完成把传入的物理页加入到内存分配器的 freelist 链表的功能。

②申请内存即 void* kalloc(void)函数，其功能为将当前内存分配器中的空闲页链表 freelist 的第一个空闲内存页移除并返回给调用者。

c. 为什么指导书提及的优化方法可以提升性能？

通过察看源码，可以发现只有一个 freelist，在多 CPU 环境下，每个 CPU 核可以独立运行，但在进行内存操作时，由于要访问的 freelist 只有一个，为临界资源，故需要通过申请保护 freelist 的锁来完成对内存的操作，没有获取到锁的 CPU 就会等待，直至其获取锁才能继续工作，这样在很大程度上降低了多核 CPU 的并发工作效率。

通过为每一个 CPU 建立一个独立的空闲内存页链表，可以减少多个 CPU 对同一个锁的争抢，即正常情况下，每个 CPU 只需要获取当前其独享的空闲内存页链表的保护锁即可。只有在当前 CPU 的 freelist 中没有空闲页链表时，才会与别的 CPU 争抢锁，以获得内存页的分配，从而提升了性能。

2. 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存就是一段缓存区，是磁盘与文件系统交互的中间层。由于对磁盘的读取速度远远慢于内存，可以在内存中分配一段缓存区域用于磁盘缓存，将那些经常访问到的磁盘块存入该缓存区域，在需要访问这些磁盘块的数据时，就可以从内存缓存区中获取数据，不用再去读写磁盘块，从而达到提升性能的作用。

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

由于每一次释放掉一个块时，若该块没有在被使用了，则将这个块放到头节点之后。因而在查看一个磁盘块是否被映射到磁盘缓存时，用 next 这个指针遍历，最先访问的到的就是距离目前最近被使用过的块，根据最近使用过的块很有可能再次访问，可以更快找到要访问磁盘块对应的缓存块。

同理，在需要找到一个空闲块时，用 prev 遍历，最先访问到的就是距离目前最远的块，该块由于距离上一次被使用的时间较长，所以优先利用该块去映射别的磁盘块。

综上，设置 prev 和 next 两个成员能够提高效率。

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。

由于使用链表访问磁盘缓存，在查询时，只能按照顺序遍历的方法查找，且只有一个锁，会导致效率较低，通过哈希映射可以加快查找速度，同时由于每一个哈希桶都拥有一个锁，可以在不同进程查询不同哈希桶中是否有其要访问的磁盘块的缓存数据时，减少对锁的争夺，从而提高磁盘缓存的性能。

我认为不能用内存分配器的优化方法优化磁盘缓存，因为对于每一个 CPU 来说，可以为其分配专属的空闲页链表，但磁盘缓存是将磁盘上的数据映射到了内存的缓存区，这个缓存区是被多个进程所共享的，若要为每一个 CPU 均建立一个磁盘缓存显然会造成资源的浪费。

二、 实验详细设计

1.内存分配器

首先，为每一个 CPU 都维护一个内存分配器，用 kmems 数组表示，如下图。

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

struct kmem{
  struct spinlock lock;
  struct run *freelist;
};
struct kmem kmems[NCPU]; // 最多8个cpu
```

在初始化时，为每一个 CPU 的内存分配器初始化锁，调用 freerange 函数，将当前所有内存页都加入到当前 CPU 的 freelist 中（初始化时，为 0 号 CPU），其中 end 为进入内核的首个物理页，PHYSTOP 为最后一个物理页。如下图。

```
void
kinit()
{
  // 获取cpuid前关中断
  for(int i = 0;i < NCPU;i++)
    initlock(&kmems[i].lock,names[i]);
  freerange(end, (void*)PHYSTOP);
}
```

修改 kalloc 函数，申请内存时，首先关中断获取当前 CPU 号，根据 CPU 号作为 kmems 的数组下标可获取当前 CPU 保护 freelist 的锁，判断当前空闲链表中是否有空闲页，若有空闲页，取该空闲页所链接的下一个空闲页作为 freelist 的头节点，释放锁并返回该空闲链表即可。若当前内存页中没有空闲页，则需要从别的 CPU 的 freelist 中抢一块内存页返回。依

次获取每个 CPU 保护 freelist 的锁，重复上述操作，若有空闲内存页，则释放访问 CPU 的锁，返回该空闲页即可。若当前访问的 CPU 的 freelist 也为空，则释放锁，访问下一个 CPU。如下图。

```
void *
kalloc(void)
{
  struct run *r;
  push_off();
  int id = cpuid();
  pop_off();
  acquire(&kmems[id].lock);
  r = kmems[id].freelist;
  if(r)
    kmems[id].freelist = r->next;
  release(&kmems[id].lock);
  if(!r){
    for(int i = 0;i < NCPU;i++){
      acquire(&kmems[i].lock);
      struct run* p = kmems[i].freelist;
      if(p)
        {
          kmems[i].freelist = p->next;
          r = p;
          release(&kmems[i].lock);
          break;
        }
      release(&kmems[i].lock);
    }
  }
  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

修改 kfree 函数，首先关中断，获取当前的 CPU 号，根据 CPU 号作为 kmems 数组的下标，获取当前 CPU 保护 freelist 的锁，并将传入的内存页加入到对应 freelist 链表头，释放锁即可。

```
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;
  push_off();
  int id = cpuid();
  pop_off();
  acquire(&kmems[id].lock);
  r->next = kmems[id].freelist;
  kmems[id].freelist = r;
  release(&kmems[id].lock);
}
```

2.磁盘缓存

首先修改 bcache 的数据结构，将原先的单个链表拆分成 17 个哈希桶，每个桶都有一个锁进行保护。

```
#define NBUCKETS 17
struct {
  struct spinlock lock[NBUCKETS];
  struct buf buf[NBUF];
  struct buf hashbucket[NBUCKETS];

  // 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;
} bcache;
```

修改初始化函数，首先为每个桶的锁和头节点初始化。然后为当前缓存中的每一个缓存初始化其锁，通过散列函数计算得到桶号，加入到该桶中即可。

```
void
binit(void)
{
  struct buf *b;
  // 初始化锁和各个哈希桶的头结点
  for(int i = 0;i < NBUCKETS;i++){
    initlock(&bcache.lock[i],name[i]);
    bcache.hashbucket[i].next = &bcache.hashbucket[i];
    bcache.hashbucket[i].prev = &bcache.hashbucket[i];
  }
  // 将各个缓存映射到对应的桶中
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    // 取散列函数，得到对应的桶号
    int key = b->blockno % NBUCKETS;
    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;
  }
}
```

修改 bget 函数，通过传入的磁盘块号散列得到可能缓存它的桶号，获取该桶的锁，通过 next 指针遍历该桶，若该磁盘块已经被缓存，增加其被引用的次数，释放该桶的锁并返回。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  int key = blockno % NBUCKETS;
  acquire(&bcache.lock[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.lock[key]);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

若未找到，则说明缓冲区中并未缓存该磁盘块，通过 prev 指针遍历该桶，寻找引用次数为 0 的缓存块（由于通过 prev 指针遍历，其上一次的使用时间应该是最久的），根据传入的参数修改其 dev 和 blockno 的值，设置其有效位为 0 表示需要读取磁盘块。释放该桶的锁并返回该缓存块。

```
  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  for(b = bcache.hashbucket[key].prev; b != &bcache.hashbucket[key]; b = b->prev){
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;
      release(&bcache.lock[key]);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

若当前桶内没有没有被使用的缓存块，则需要从别的哈希桶中寻找。首先释放当前哈希桶的锁，遍历每一个哈希桶，获取其锁，仍然通过 prev 指针遍历，若找到了一个未被使用的缓存块，则根据传入的参数修改其 dev 和 blockno 的值，设置其有效位为 0 表示需要读取磁盘块，同时需要获取它需要被加入的桶的锁，插入到对应桶中，插入后释放该桶的锁，返回该缓存块指针即可。若当前遍历的桶中没有这样的缓存块，也需要释放锁，避免死锁。

```
  for(int i = 0;i < NBUCKETS;i++){
    if(i == key) continue;
    acquire(&bcache.lock[i]);
    for(b = bcache.hashbucket[i].prev;b != &bcache.hashbucket[i];b = b->prev){
      if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        // 将b原来所属的链表结构改变
        b->next->prev = b->prev;
        b->prev->next = b->next;
        // 将b加入到key所指的哈希桶中
        acquire(&bcache.lock[key]);
        b->next = bcache.hashbucket[key].next;
        b->prev = &bcache.hashbucket[key];
        bcache.hashbucket[key].next->prev = b;
        bcache.hashbucket[key].next = b;
        release(&bcache.lock[key]);

        release(&bcache.lock[i]);
        acquiresleep(&b->lock);
        return b;
      }
    }
    // 避免死锁
    release(&bcache.lock[i]);
  }
  panic("bget: no buffers");
```

修改 brelse 函数，先通过散列找到对应的桶，获取该桶的锁，b 的使用次数减一，若此时 b 的使用次数为 0，则将其插入到当前桶的头节点下一个位置（因为目前该块为最近被使用的）。释放锁结束。

```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");
  int key = b->blockno % NBUCKETS;
  releasesleep(&b->lock);
  acquire(&bcache.lock[key]);
  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[key].next;
    b->prev = &bcache.hashbucket[key];
    bcache.hashbucket[key].next->prev = b;
    bcache.hashbucket[key].next = b;
  }

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

修改 bpin 和 bunpin 函数，散列计算并获取锁，增加或者减少使用次数后释放锁，结束。

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

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

三、 实验结果截图

截图如下，全部通过。

```
== Test running kalloctest ==
$ make qemu-gdb
(97.6s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (15.4s)
== Test running bcachetest ==
$ make qemu-gdb
(13.1s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (193.5s)
== Test time ==
time: OK
Score: 70/70
[ming@localhost xv6-labs-2020]$
```