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

实验设计报告

开课学期：2021 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：2021/10/28 地点：T2210

学生班级：计科 2 班

学生学号：1190303311

学生姓名：王志军

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？

内存分配器就是系统底层的物理内存管理器，包括物理内存相关数据结构和操作；它的作用就是管理物理内存，将用户和物理内存隔离开，使用户只需要关注虚拟内存空间，其维护一个空闲内存块链表，为用户分配一页可用的内存，或释放一块内存，所有进程都可以访问这个链表，缓存分配和释放时有锁机制

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？

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

Kmem 是内存分配器的数据结构，有两个属性，一个是空闲内存块链表的自旋锁，一个是空闲块链表；
操作：kinit 初始化链表和锁，为链表分配 end 到 PHYSTOP 之间的所有内存块，通过调用 freearrange 实现；
Kfree 释放一页大小的内存；
Kalloc 分配一页大小的内存，大小为 4K，

c. 为什么指导书提及的优化方法可以提升性能？

未优化时所有 cpu 共用同一个内存空闲链表，这个空闲链表只有一个锁，为所有 cpu 分配和回收内存页，当多个进程同时请求空闲链表时就会出现锁争用，在只有一个链表的情况下这种争用出现的次数会很多，导致性能下降；
指导书提及的优化算法是为每个 CPU 分配一个空闲链表，并为他们都初始化一个锁，当自己的空闲链表为空时，去其他 cpu 的空闲链表寻找可用的内存块并加入到自己的链表中，也就是 steal，这样多个 cpu 争用同一块链表的情况就会大大减少，只有当自己没有空闲链表可用时才会出现这种情况，出现的机率不大，因此可以提升性能。

2. 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存是内存和磁盘之间的一种结构，由于内存读取速度很快，磁盘很慢，因此将磁盘中常用的文件放入内存可以大大加快运行速度，提升性能，这块内存就是磁盘缓存，其作用就是存储常用数据块，提升性能。

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）。

首先 brelse 释放一个缓存块时是将其按照头插法插入的，因此链表中 head.next 是最近使用的，head.prev 是最少使用的；
当在 cache 中寻找对应的块时是从 head 按照 next 方式遍历，这样就是在最近使用过的一些 cache 块中寻找，命中的概率较大，效率也高；
在 cache 中寻找缓存块未命中，需要找到一个空闲块载入数据时，从 head.prev 开始，按照 prev 方式遍历，就是从最少使用的那些块中寻找，更合适，减少了常用块的频繁换入换出。

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。

哈希表将磁盘块 block 分配到了不同的桶中，不同的桶都有各自的锁，因此可以实现多个进程同时访问不同桶内 block 的情况，减少了锁争用；
不能，内存分配器的优化算法是为每个 cpu 分配一个数据结构，缓存在进程和 CPU 之间是共享的，磁盘缓存产生锁争用的一个原因是多个 CPU 读取不同的 block 时会产生锁争用，哈希表减少的就是这部分争用，产生争用的对象不是 cpu 而是 block，内存分配器的优化算法并不能减少这类争用。

二、 实验详细设计

1、 内存分配器

根据指导书的提示，大致思路就是为每个 CPU 分配一个空闲链表和锁，分配内存块时先去当前 cpu 下的空闲链表寻找空闲块，找到了就分配，否则去其他 cpu 的空闲链表中寻找，释放内存块时将内存块放在自己的空闲链表中；

首先修改空闲块结构，为每一个 cpu 分配一个空闲链表：

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

struct kmem kmems[NCPU]; // 为每个CPU维护一个空闲链表
```

初始化时需要给每个 cpu 的空闲链表初始化锁，同意命名为 kmem：

```
void
kinit()
{
  for(int i=0; i<NCPU; i++)
    initlock(&kmems[i].lock, "kmem");//初始化所有cpu空闲链表的锁，都命名为kmem，否则测试结果
  freerange(end, (void *)PHYSTOP);
}
```

释放内存块时先获取当前 cpu 号，然后给其对应的空闲链表加锁，接着释放内存块再释放：

```
kfree(void *pa)
{
  push_off();//获取当前cpu号，push_off禁止中断，pop_off启用
  uint64 cpu = cpuid();
  pop_off();
  struct run *r;

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

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

  r = (struct run*)pa;

  acquire(&kmems[cpu].lock);//内存块回收到当前cpu的空闲链表中
  r->next = kmems[cpu].freelist;
  kmems[cpu].freelist = r;
  release(&kmems[cpu].lock);
}
```

分配内存块时，先获取当前 cpu 的 id，对其空闲链表加锁然后先获取其空闲链表，如果没有可用的就释放锁，遍历所有 cpu 的空闲链表，对每一个链表先加锁再找空闲块，没有就释放锁，到下一个 cpu 的空闲链表寻找，直到找打一个空闲块。

```
acquire(&kmems[cpu].lock);//先对当前cpu空闲链表加锁
r = kmems[cpu].freelist;
if(r)//如果非空，就释放锁返回空闲块
{
  kmems[cpu].freelist = r->next;
  release(&kmems[cpu].lock);
}
else//否则代表自己没有可分配的空闲块，去其他cpu steal
{
  release(&kmems[cpu].lock);//steal之前释放当前cpu的锁
  for(int i=0; i<NCPU; i++)
  {
    acquire(&kmems[i].lock);//循环取出每个cpu的空闲链表，查询是否非空
    r = kmems[i].freelist;
    if(r)//非空就跳出循环，返回一个空闲块
    {
      kmems[i].freelist = r->next;
      release(&kmems[i].lock);
      break;
    }
    release(&kmems[i].lock);//释放cpu i的锁，检查下一个cpu的空闲链表
  }
```

2、 磁盘缓存

采取哈希桶的方式减少锁争用，分成 13 个桶；

数据结构，分成 13 个哈希桶，每个桶一个锁：

```
#define NBUCKETS 13

struct {
  struct spinlock lock[NBUCKETS];//分成13个桶，每个桶一个锁
  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 hashbucket[NBUCKETS];//13个桶的head
} bcache;
```

初始化，初始化每一个桶的锁，并将所有 cache 块分配给第一个桶，其他桶获取缓存块时发现自己的桶内不命中也没有可用的块，就去其他桶 steal：

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

  for(int i=0; i<NBUCKETS; i++)//为每一个桶初始化锁
  {
    initlock(&bcache.lock[i], "bcache");
  }

  // Create linked list of buffers
  for(int i=0; i<NBUCKETS; i++)
  {
    bcache.hashbucket[i].prev = &bcache.hashbucket[i];
    bcache.hashbucket[i].next = &bcache.hashbucket[i];
  }
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.hashbucket[0].next;//把所有的缓存cache都分配给第一个桶，后续的桶steal
    b->prev = &bcache.hashbucket[0];
    initsleeplock(&b->lock, "buffer");
    bcache.hashbucket[0].next->prev = b;
    bcache.hashbucket[0].next = b;
  }
}
```

Bget 查找 cache 块，先获取 block 的哈希值，在对应的哈希桶内从 head 开始，next 向前寻找最近使用过的内存块，命中则返回：

```
 for(b = bcache.hashbucket[bucket].next; b != &bcache.hashbucket[bucket]; b = b->next){
   if(b->dev == dev && b->blockno == blockno){
     b->refcnt++;
     release(&bcache.lock[bucket]);
     acquiresleep(&b->lock);
     return b;
   }
 }
```

不命中时需要找到一个空闲块，先在当前桶内从后往前找，因为后面的块是最不常使用的，找到了返回：

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

找不到时需要去其他桶 steal，遍历其他桶，每处理一个桶都需要对其加锁，没有可用的 cache 块就解锁然后处理下一个，否则使用该块，同时要直接把这个块加入到自己的桶中，这样其他访问这个 block 的请求就可以立刻在相应的哈希桶中找到，避免触发重复缺页。

Brelse 回收时直接将块会受到对应的哈希桶中即可，最后还需要修改 bin bunpin，把锁改成对应哈希桶的锁。

三、 实验结果截图

请填写

```
== Test running kalloctest ==
$ make qemu-gdb
(145.5s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (23.6s)
== Test running bcachetest ==
$ make qemu-gdb
(27.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (173.7s)
== Test time ==
time: OK
Score: 70/70
1190303311@OSLabExecNode0:~/xv6-labs-2020$
```