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

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：4 学时　地点：T2608

学生班级：19 级计科 10 班

学生学号：190111005

学生姓名：倪洪达

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？

在 xv6 中内存分配器是用来管理物理内存的一项，它提供了 kalloc 和 kfree 接口来管理物理内存。通过 kalloc 和 kfree，屏蔽了对物理内存的管理，使得调用者只需要关心虚拟地址空间，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？

内存分配器的数据结构是**由空闲物理页组成的链表 freelist 和保护这个链表的自旋锁**组成的。每个空闲页在链表里都是 struct run，next 指向下一个空闲物理页。

它的操作即 kalloc 中的几个函数，其中：

kinit()用于初始化分配器，通过保存所有空闲页来初始化链表。

freerange()来把空闲内存页加到链表里。

kfree()函数用于释放指定的物理内存页，将其添加至 freelist 中。

kalloc()函数用于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？

指导书所提出的优化方法即给每一个 CPU 都分配一个对应的内存分配器（即空闲页面链表）来控制，当通过 cpuid()函数获得 CPU 的编号后，先在对应的内存分配器还有剩余物理页的情况下分配内存。如果对应的内存分配器没有足够的内存，再从别的 CPU 的内存分配器窃取。

这个方法可以极大程度减少锁的争用，多个锁管理各自的空闲页面链表，就只有当对应的页面没有剩余而需要窃取的时候，才会发生争用的情况。通过 kalloctest.c 通过的结果也能知道，CPU 没有获取到锁的次数大大减小了。

2．磁盘缓存

a． 什么是磁盘缓存？它的作用是？

xv6 的文件系统是以磁盘数据块为单位从磁盘读写数据的。由于对磁盘的读取非常慢，因此将最近经常访问的磁盘块缓存在内存里可以大大提升性能，也就是说磁盘缓存是磁盘与文件系统交互的中间层。

在 xv6 中，磁盘缓存主要有三个作用：

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线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）。

buf 的结构体表示的是一个双向链表节点，具有 prev 和 next 两个成员。虽然在大部分情况下，这个链表只需要从前往后遍历。一旦查找数据块未命中，从后往前遍历，去找之前被遍历过但没被使用过的数据块来使用则是最好的选择。因为如果仍然往后遍历，前面没被使用过的数据块永远不会被使用，会造成资源浪费。因此设计成双向链表是必须的。

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。

使用哈希表其实是对缓存块的一种合理划分，可以通过给每个哈希桶上锁，来减少锁的争用。在 bget()未命中的情况下，从其他哈希桶中来选择未被使用的缓存块效率比直接往前遍历更高。

不可以，由于磁盘缓存独特的命中与未命中的机制，如果单纯地按照 CPU 的数目来分割磁盘，则后续管理磁盘的时候就无法使用块号的取模方法，在检测命中与否的过程中依然需要遍历整个磁盘，减少了锁争用而无法减少锁的使用。虽然避免了多进程访问同一个数据块的可能，但是总体并没有得到优化。

二、 实验详细

1、 内存分配器

参照指导书方案改写了空闲物理页链表，使每个 CPU 有对应分配器和锁。内存初始化代码略。

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

acquire(&kmems[id].lock);
r->next = kmems[id].freelist;
kmems[id].freelist = r;
release(&kmems[id].lock);
```

以上是释放内存的代码，可以看到只需要对当前 CPU 对应的物理页进行分配。

```
if (r)
{
  kmems[id].freelist = r->next;
  release(&kmems[id].lock);
}
else
{
  release(&kmems[id].lock);
  for (int i = 0; i < NCPU; i++)
  {
    if (i != id) //在非对应的主存中寻找可用空间（窃取）
    {
      acquire(&kmems[i].lock);
      r = kmems[i].freelist;
      if (r)
      {
        kmems[i].freelist = r->next;
        release(&kmems[i].lock);
        break;
      }
      else
      {
        release(&kmems[i].lock);
        continue;
      }
    }
  }
}
```

以上申请内存的代码，还有剩余空间的情况不阐述。在当前 CPU 对应的物理

页中没有可用空间时需要窃取，此时只需要遍历各个 CPU，只要有一个 CPU 对应的物理页有剩余空间即可 break.

2、 磁盘缓存

参照指导书设计了哈希表，没有使用时间戳。桶的值选择了 19 个（可以通过测试）

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

  for (int i = 0; i < NBUCKETS; i++)
  {
    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;
    b->prev = &bcache.hashbucket[0];
    initsleeplock(&b->lock, "buffer");
    bcache.hashbucket[0].next->prev = b;
    bcache.hashbucket[0].next = b;
  }
}
```

初始化代码即对每个锁都作初始化，每个链表的双指针都指向头节点。而一开始缓存块号是 0，故放在 0 号哈希桶。

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

  int hash = blockno % NBUCKETS; //获得对应锁和哈希桶的下标
  acquire(&bcache.lock[hash]);
  // Is the block already cached?
  for (b = bcache.hashbucket[hash].next; b != &bcache.hashbucket[hash]; b =
b->next)
  {
    if (b->dev == dev && b->blockno == blockno)
    {
```

```
      b->refcnt++;
      release(&bcache.lock[hash]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  for (int i = (hash + 1) % NBUCKETS; i != hash; i = (i + 1) % NBUCKETS)
  {
    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;

        b->next->prev = b->prev;
        b->prev->next = b->next;//从旧缓存块中取出

        b->next = bcache.hashbucket[hash].next;
        b->prev = &bcache.hashbucket[hash];
        bcache.hashbucket[hash].next->prev = b;
        bcache.hashbucket[hash].next = b;//将缓存块添加到头部

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

bget()的代码如上，通过定义变量 hash 来获得对应下标，代表待判断的块号根据取模的规则只可能出现在 hash 号桶里面，否则就是未命中。未命中的情况下，我选择了从当前桶的下一个桶开始遍历全部的桶，以防止遍历一样的桶从而出现死锁。如果在当前桶中有未被使用的块，就把这个块取出，放到我们下标为 hash 的桶的头部即可。

```
void brelse(struct buf *b)
{
  if (!holdingsleep(&b->lock))
    panic("brelse"); //如果睡眠锁没被锁住，直接返回

  releasesleep(&b->lock); //否则就是被锁住，此时把睡眠锁锁释放

  int hash = b->blockno % NBUCKETS; //获得对应锁和哈希桶的下标
  acquire(&bcache.lock[hash]);
  b->refcnt--;
  if (b->refcnt == 0)
  {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    // 从旧缓存块中取出
    b->next = bcache.hashbucket[hash].next;
    b->prev = &bcache.hashbucket[hash];
    bcache.hashbucket[hash].next->prev = b;
    bcache.hashbucket[hash].next = b;
    // 将缓存块添加到头部
  }
  release(&bcache.lock[hash]);
}
```

由于去掉了全局的锁，brelse 改为使用当前块号对应的哈希桶的锁。

三、 实验结果截图

```
== Test running kalloctest ==
$ make qemu-gdb
(121.8s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (12.0s)
== Test running bcachetest ==
$ make qemu-gdb
(11.1s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (152.1s)
== Test time ==
time: OK
Score: 70/70
190111005@OSLabExecNode0:~/xv6-labs-2020$
```