Lab8 locks

在这个实验中，您将获得重新设计代码以提高并行性的经验。多核机器上并行性差的一个常见症状是高锁争用。提高并行性通常需要更改数据结构和锁定策略，以减少争用。您将为xv6内存分配器和块缓存执行此操作。

1. Memory allocator (moderate)

1）实验目的

> Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation

> has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the kalloctests pass.
> 您的工作是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。 您必须提供所有以 `kmem` 开头的锁名称。 也就是说，您应该为每个锁调用 `initlock` ，并传递以 `kmem` 开头的名称。 运行 `kalloctest` 以查看您的实现是否减少了锁争用。 要检查它是否仍可以分配所有内存，请运行 `usertests sbrkmuch` 。 您的输出将类似于以下所示，尽管 `kmem` 锁的总争用数量有所减少，但总争用减少了很多。 确保 `usertests` 中的所有测试均通过。 `make grade` 应该说 `kalloctests` 通过了。

2）实验步骤

编写代码

在 `kernel/kalloc.c` 文件中修改 `kmem` 的定义，变为一个 `kmem` 数组，使得每个CPU都有对应的 `freelist` 和 `lcok`

```
struct {
  struct spinlock lock;
  struct run *freelist;
// 修改为数组 每个CPU对应一个kmem lab8-1
} kmem[NCPU];
```

在 `kernel/kalloc.c` 文件中修改 `kmem` 的初始化函数 `kinit()` ，为 `kmem` 数组中的每个元素进行初始化

```
void
kinit()
{
  // 给每个kmem初始化 lab8-1
  for (int i = 0; i < NCPU; ++i)
        initlock(&kmem[i].lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}
```

在 `kernel/kalloc.c` 文件中修改 `kfree()` 函数，获取当前CPUID时先关闭中断，然后再对该CPU执行相应的锁操作

```
void
kfree(void *pa)
{
  ...
  r = (struct run*)pa;
  // 关闭中断 获取那个CPU lab8-1
  push_off();
  int index = cpuid();
  pop_off();

  acquire(&kmem[index].lock);
  r->next = kmem[index].freelist;
  kmem[index].freelist = r;
  release(&kmem[index].lock);
}
```

在 `kernel/kalloc.c` 文件中修改 `kalloc()` 函数，获取当前CPUID时先关闭中断，查找当前CPU有无空闲块，有就返回，没有就找CPU中的空闲块返回

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

  // 关闭中断 获取那个CPU
  push_off();
  int index = cpuid();
  pop_off();

  acquire(&kmem[index].lock);
  r = kmem[index].freelist;
  // 有空闲块
  if(r)
    kmem[index].freelist = r->next;
  else {
        for (int i = 0; i < NCPU; ++i) {
                if (i == index)
                        continue;
                acquire(&kmem[i].lock);
                r = kmem[i].freelist;
                if (r)
                        kmem[i].freelist = r->next;
                release(&kmem[i].lock);
                if (r)
                        break;
        }
  }
  release(&kmem[index].lock);
  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `kalloctest` 进行测试

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 55284
lock: kmem: #fetch-and-add 0 #acquire() 190180
lock: kmem: #fetch-and-add 0 #acquire() 187565
lock: bcache: #fetch-and-add 0 #acquire() 1248
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 48217 #acquire() 114
lock: proc: #fetch-and-add 47090 #acquire() 174952
lock: proc: #fetch-and-add 15350 #acquire() 174993
lock: pr: #fetch-and-add 10046 #acquire() 5
lock: proc: #fetch-and-add 8166 #acquire() 174993
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

`kalloctest` 测试通过

键入指令 `usertests sbrkmuch` 进行测试

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

`usertests sbrkmuch` 测试通过

3）实验中遇到的问题和解决方法

根据实验提示，本实验的逻辑比较简单，基本上都是对 `kmem` 数组根据当前使用的CPU进行定位，找到CPU的内存块

4）实验心得

这段提示看得不是很明白： `cpuid` 函数返回当前内核号，但是只有在中断关闭时调用它并使用其结果才是安全的。 您应该使用 `push_off()` 和 `pop_off()` 来打开和关闭中断

于是我们上网搜索相关资料

在操作系统中，中断是一种机制，允许外部事件（如硬件设备的输入/输出请求）打断正在执行的程序，并立即处理这些事件。当中断发生时，处理器会暂停当前正在执行的指令，保存当前的上下文，并跳转到相应的中断处理程序

在多任务操作系统中，同时可能有多个任务在运行，这些任务共享计算机的资源。为了确保任务之间的正确协作和资源的正确使用，操作系统需要对中断进行管理和控制

当调用 `cpuid` 函数时，它会读取当前内核号并返回结果。然而，在多任务环境中，其他任务可能会在任何时候被调度并运行，包括在 `cpuid` 函数执行期间。如果在执行 `cpuid` 函数期间发生中断，可能会**导致内核号的值变化**，从而导致不正确的结果

为了避免这种情况，可以使用 `push_off()` 和 `pop_off()` 函数来关闭和打开中断。通过调用 `push_off()` 函数，可以禁用中断，确保在执行 `cpuid` 函数期间不会发生中断。然后，在获取到 `cpuid` 函数的结果后，可以调用 `pop_off()` 函数来恢复中断状态，使得其他任务能够继续正常运行

通过使用 `push_off()` 和 `pop_off()` 函数来关闭和打开中断，可以确保在执行cpuid函数期间不会发生中断，从而保证获取到的内核号是准确和安全的。这样可以避免由于中断引起的竞态条件和不一致性问题

2. Buffer cache (hard)

1）实验目的

> Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure usertests still passes. `make grade` should pass all tests when you are done.
> 修改块缓存，使bcache中所有锁的 `acquire` 循环迭代次数在运行 `bcachetest` 时接近于零。理想情况下，块缓存中涉及的所有锁的计数之和应该为零，但如果总和小于500也可以。修改 `bget` 和 `brelse`，以便bcache中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待 `bcache.lock`。您必须保持不变，即每个块最多缓存一个副本。完成后，您的输出应该与下面显示的类似（尽管不完全相同）。确保用户测试仍然通过，当你完成所有的测试时，你应该通过所有的测试。

2）实验步骤

编写代码

修改 kernel/buf.h 文件中的 buf 结构体定义，添加数据成员 tick 记录上次使用时间，用于LRU方案

```
struct buf {
  ...
  uchar data[BSIZE];
  // 用于LRU记录上次使用时间 lab8-2
  uint tick;
};
```

修改 kernel/bio.c 文件中 bcache 的定义
定义素数个数的桶，从而尽可能地减少桶冲突
为每个桶 bcache 中的每个桶都加上锁

```
struct {
  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.

  // lab8-2
  struct spinlock big_lock;
  struct spinlock lock[NBUCKET];
  struct buf head[NBUCKET];
} bcache;
```

在 kernel/bio.c 文件中添加哈希函数，从而能通过块号映射到对应的桶

```
// 哈希函数
int hash(int blockno) {
  return blockno % NBUCKET;
}
```

修改 kernel/bio.c 文件中 bcache 的初始化函数 binit()

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

  initlock(&bcache.big_lock, "bcache_big_lock");
  // Create linked list of buffers
  // bcache.head.prev = &bcache.head;
  // bcache.head.next = &bcache.head;
  // 初始化桶链表 lab8-2
  for (int i = 0; i < NBUCKET; ++i) {
    initlock(&bcache.lock[i], "bcache_bucket");
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
  }
  // 初始化buf lab8-2
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head[0].next;
    b->prev = &bcache.head[0];
    initsleeplock(&b->lock, "buffer");
    bcache.head[0].next->prev = b;
    bcache.head[0].next = b;
  }
}
```

在 `kernel/bio.c` 文件中修改 `bget()` 函数，判断是否命中，若命中就返回；若不命中，按*从大到小*的顺序解锁，此时可能有新缓存，仍要判断是否命中；若仍不命中，就按LRU的方式查找获取空闲块；若没有就去别的桶获取空闲块

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;
  // lab8-2
  int index = hash(blockno);
  int min_tick = __UINT32_MAX__;
  struct buf* target_buf = 0;

  acquire(&bcache.lock[index]);

  // Is the block already cached?
  // 命中
  for(b = bcache.head[index].next; b != &bcache.head[index]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      // 记录使用时间lab8-2
      // b->tick = ticks;
      release(&bcache.lock[index]);
      acquiresleep(&b->lock);
      return b;
    }
  }
  release(&bcache.lock[index]);

  // 不命中
  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  acquire(&bcache.big_lock);
  acquire(&bcache.lock[index]);
  // 当前桶中找空闲
  for (b = bcache.head[index].next; b != &bcache.head[index]; b = b->next) {
    if (b->dev == dev && b->blockno == blockno) {
      b->refcnt++;
      release(&bcache.lock[index]);
      release(&bcache.big_lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  for (b = bcache.head[index].next; b != &bcache.head[index]; b = b->next) {
    if (b->refcnt == 0 && (target_buf == 0 || b->tick < min_tick)) {
      min_tick = b->tick;
      target_buf = b;
    }
  }

  if (target_buf) {
    target_buf->dev = dev;
    target_buf->blockno = blockno;
```

```
      target_buf->refcnt++;
      target_buf->valid = 0;
      release(&bcache.lock[index]);
      release(&bcache.big_lock);
      acquiresleep(&target_buf->lock);
      return target_buf;
    }

    // 从其他桶找空闲块
    for (int i = hash(index + 1); i != index; i = hash(i + 1)) {
      acquire(&bcache.lock[i]);
      for (b = bcache.head[i].next; b != &bcache.head[i]; b = b->next) {
        if (b->refcnt == 0 && (target_buf == 0 || b->tick < min_tick)) {
          min_tick = b->tick;
          target_buf = b;
        }
      }
      if (target_buf) {
        target_buf->dev = dev;
        target_buf->refcnt++;
        target_buf->valid = 0;
        target_buf->blockno = blockno;
        // 从原桶中删除
        target_buf->next->prev = target_buf->prev;
        target_buf->prev->next = target_buf->next;
        release(&bcache.lock[i]);
        //加锁
        target_buf->next = bcache.head[index].next;
        target_buf->prev = &bcache.head[index];
        bcache.head[index].next->prev = target_buf;
        bcache.head[index].next = target_buf;
        release(&bcache.lock[index]);
        release(&bcache.big_lock);
        acquiresleep(&target_buf->lock);
        return target_buf;
      }
      release(&bcache.lock[i]);
    }

    release(&bcache.lock[index]);
    release(&bcache.big_lock);
    panic("bget: no buffers");
  }
```

在 `kernel/bio.c` 文件中 `brelse()` 中调用哈希函数，决定对应那个桶

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

  releasesleep(&b->lock);

  // lab8-2
  int index = hash(b->blockno);

  acquire(&bcache.lock[index]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // LRU记录辅助
    b->tick = ticks;
  }

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

在 `kernel/bio.c` 文件中修改 `bpin()` 函数和 `bunpin()` 函数

```
void
bpin(struct buf *b) {
  // lab8-2
  int index = hash(b->blockno);
  acquire(&bcache.lock[index]);
  b->refcnt++;
  release(&bcache.lock[index]);
}

void
bunpin(struct buf *b) {
  // lab8-2
  int index = hash(b->blockno);
  acquire(&bcache.lock[index]);
  b->refcnt--;
  release(&bcache.lock[index]);
}
```

测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统QEMU模拟器

键入指令 `bcachetest` 进行测试

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32938
lock: kmem: #fetch-and-add 0 #acquire() 27
lock: kmem: #fetch-and-add 0 #acquire() 76
lock: bcache_big_lock: #fetch-and-add 0 #acquire() 123
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2129
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4123
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4333
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6343
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6340
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6331
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6696
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6707
lock: bcache_bucket: #fetch-and-add 0 #acquire() 7743
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4138
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4139
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2128
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4137
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 411429 #acquire() 1254
lock: proc: #fetch-and-add 49131 #acquire() 73865
lock: proc: #fetch-and-add 23939 #acquire() 73503
lock: proc: #fetch-and-add 10043 #acquire() 73509
lock: proc: #fetch-and-add 7906 #acquire() 73509
tot= 0
test0: OK
start test1
test1 OK
```

结果符合预期

键入 `Ctrl+a` ， 松开，然后键入 `x` ， 退出xv6系统

退出xv6进行单元测试

```
root@LAPTOP-UER420HO:~/lab8_lock# ./grade-lab-lock
make: 'kernel/kernel' is up to date.
== Test running kalloctest == (73.9s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (6.9s)
== Test running bcachetest == (4.5s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests == usertests: OK (78.2s)
== Test time ==
time: OK
Score: 70/70
```

3）实验中遇到的问题和解决方法

链表的连接

主要是对于 `bget()` 函数的修改，设计大量的链表操作，写了好几次都没对
还是得在纸上画一画

freeing free block

`./make-grade-lock` 的时候会有 `panic:freeing free block` 错误。

解决：在测试之前先清理，使用以下命令

```
make clean
```

4）实验心得

在本次实验中，我深感数据结构的重要性，对于链表的操作一不小心就会造成程序的错误，最后还是得在纸张上画出示意图才能更加清晰地理解每一条语句在干什么
本次实验的任务是利用哈希映射来完成不同不同数据块到不同的桶的映射，这里为了简单，直接只用了取余构造哈希函数，同时利用了素数个桶保证尽可能地减少哈希冲突；最后这里实际上使用了开散列的方式，也就是说每一个桶可以存储多个元素，以链表相互连接，这样的方法简单易实现，并且确实可以有效处理大量的哈希冲突；但它有可能导致较长的搜索时间，特别是当链表长度变得很长的时候
并且，这里我们根据程序的局部性原理，使用LRU算法，选择最近最少访问的块进行替换。这里很好地为我展现了在计算机组成原理课程上的理论的实现，更加加深了理解