Lab - Locks 目录

1. Memory allocator (moderate)

1.1 实验目的

- 对xv6的内存分配器进行优化

1.2 实验步骤

1. 在未修改前，所有内存块由一个锁管理，若有多个进程并发地获取内存，则会造成非常多的等待，从而降低并行性造成较大的性能浪费。要进行相应的优化，考虑对每个CPU核心单独维护一个链表存储内核内存空间，并在空闲列表为空（或满）时"借用"其他CPU的内存空间中，从而避免对单一锁和链表的竞争。
2. 参考 `kernel/param.h` 中的 `NCPU` 宏定义，在 `kernel/kalloc.c` 中修改 `kmem` 为大小为 `NCPU` 的数组，每个元素含有一个锁和一个指针头（用于访问链表）：

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU]; // 每个CPU都有维护独有的kmem
```

3. 修改 `kinit()` 函数，初始化所有CPU上的列表：

```
void kinit()
{
    // 每个CPU列表初始化
    char kmem_name[32];
    for (int i = 0; i < NCPU; i++) {
        snprintf(kmem_name, 32, "kmem_%d", i);
        initlock(&kmem[i].lock, kmem_name);
    }
    freerange(end, (void *)PHYSTOP);
}
```

4. 观察 `freerange()` 函数：

```
void freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char *)PGROUNDUP((uint64)pa_start);
    for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE)
        kfree(p);
}
```

其实现逻辑是调用 `kfree()` 释放范围内的每一个页表，因此修改 `kfree()` 函数，在xv6系统中，当CPU获取任何锁时，xv6总是禁用该CPU上的中断，而当一个CPU未持有自旋锁时，便可以重新启用中断，因此有如下代码：

```
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;

    // 当CPU获取任何锁时，xv6总是禁用该CPU上的中断。
    push_off();

    int CPUID = cpuid();
    acquire(&kmem[CPUID].lock);
    // 从链表头插入空闲页
    r->next = kmem[CPUID].freelist;
    kmem[CPUID].freelist = r;
    release(&kmem[CPUID].lock);

    // 当CPU未持有自旋锁时，xv6重新启用中断
    pop_off();
}
```

5. 修改`kalloc()`函数，在关闭中断、获取锁之后，首先查找当前CPU上的空闲页，若不存在则到其他CPU上去查找，要注意的是在访问其他CPU时也要管理锁的获取和释放：

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

    push_off();
    int CPUID = cpuid();
    acquire(&kmem[CPUID].lock);

    r = kmem[CPUID].freelist;

    // 在当前CPU上查找空闲页
    if (r)
        kmem[CPUID].freelist = r->next;

    if (r == 0) { // 若当前CPU上没有空闲页
        // 在其他CPU上查找空闲页
        for (int i = 0; i < NCPU; i++) {
            if (i == CPUID)
                continue;
            // 要获取其他CPU的锁
            acquire(&kmem[i].lock);
            r = kmem[i].freelist;
            if (r)
                kmem[i].freelist = r->next;
            release(&kmem[i].lock);
            if (r) // 已找到，脱出循环
                break;
        }
    }
```

```
    release(&kmem[CPUID].lock);
    pop_off();

    if (r)
        memset((char *)r, 5, PGSIZE); // fill with junk
    return (void *)r;
}
```

6. 运行`kalloctest`测试，结果如下：

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 1270
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 52047 #acquire() 141
lock: proc: #test-and-set 30117 #acquire() 704003
lock: proc: #test-and-set 29108 #acquire() 704009
lock: proc: #test-and-set 8059 #acquire() 303884
lock: proc: #test-and-set 4741 #acquire() 303883
tot= 0
test1 OK
start test2
total free number of pages: 32497 (out of 32768)
.....
test2 OK
start test3
child done 1
child done 100000
test3 OK
```

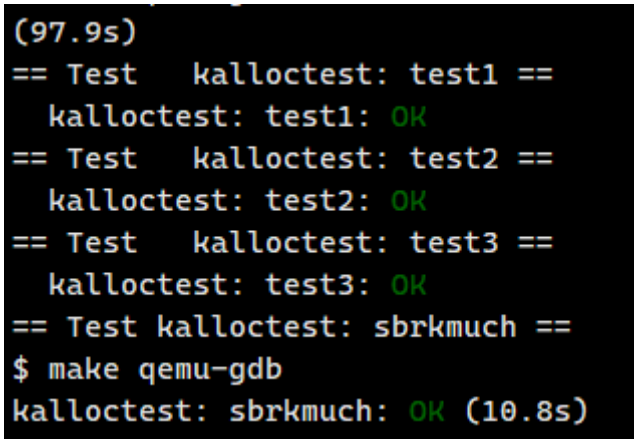
7. 运行`usertests sbrk`测试，结果如下：

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

8. 运行`usertests`测试，最终结果如下（仅展示部分）：

```
test execout: OK
test diskfull: balloc: out of blocks
ialloc: no inodes
ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

9. 运行`grade`测试，结果如下：



1.3 实验中遇到的问题和解决办法

- **问题：** 关闭中断和启用中断的时机如何判断?

  解决方法：参考xv6手册，“Xv6 is more conservative: when a CPU acquires any lock, xv6 always disables interrupts on that CPU. Interrupts may still occur on other CPUs, so an interrupt’s acquire can wait for a thread to release a spinlock; just not on the same CPU.”“Xv6 re-enables interrupts when a CPU holds no spinlocks;”，因此一旦要获取锁，则首先要关闭中断；要想启用中断，则先要释放锁。

- **问题：** `snprintf()`函数的用法

  解决方法：其原型为：`int snprintf(char *str, size_t size, const char *format, ...);`

  - `str`：指向用于存储结果字符串的字符数组的指针。
  - `size`：`str` 的大小，即最多允许输出的字符个数，包括结尾的空字符 '\0'。这是为了防止缓冲区溢出。
  - `format`：格式控制字符串，包含了指定输出格式的占位符和修饰符。
  - `...`：可变数量的参数，用于填充 `format` 中的占位符。

1.4 实验心得

通过完成内存分配器的优化实验，我理解了操作系统内核中是如何实现高效的并发数据结构和锁机制的。在设计优化方案时，考虑到多核处理器的特点，决定为每个CPU维护一个独立的空闲列表。这样一来，不同CPU上的内存分配和释放可以并行进行，从而减少了锁争用的可能性。然而，在实现"stealing"机制时，我遇到了一些困难，因为需要确保在多个CPU之间正确地共享和更新空闲列表。在尝试优化的过程中，我明白了操作系统内核优化的重要性，合理的锁设计和并发控制对于提高系统性能和稳定性至关重要。

2. Buffer cache (hard)

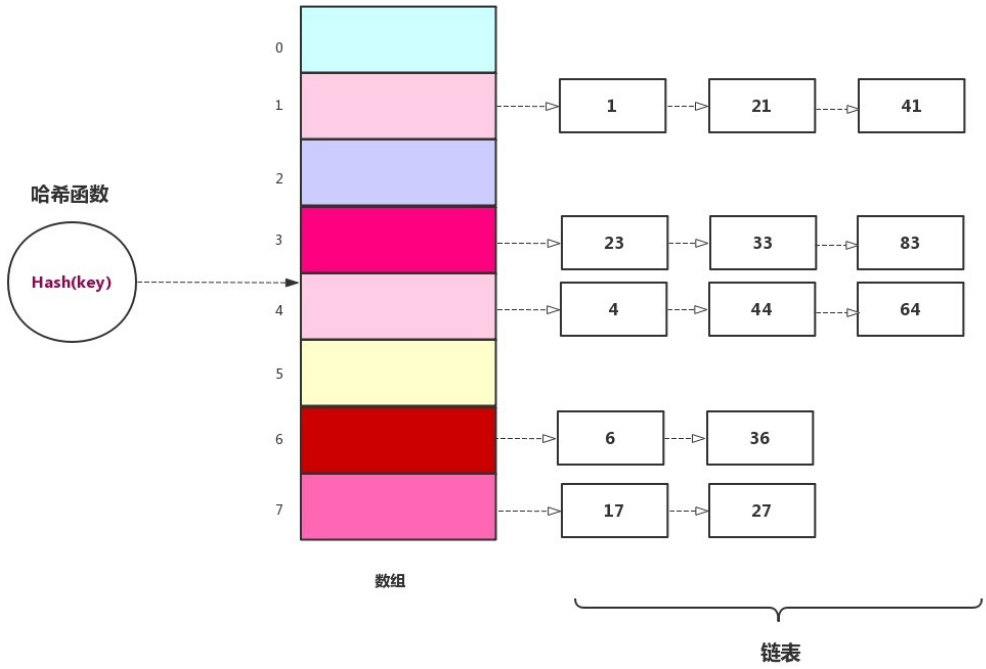
2.1 实验目的

- 优化xv6中的缓冲区缓存（`Buffer Cache`）

2.2 实验步骤

1. 如果多个进程高频率地使用文件系统，它们可能会争夺`bcache.lock`从而导致性能下降。要对此进行优化，考虑每个进程在使用的时候竞争的是bcache的锁，该锁的粒度较大，因此发生冲突的概率较高；为解决该问题，采用细粒度的锁来减少冲突，但是粒度小到每个块的锁是不合适的，因此建立哈希表，维护数量为13的哈希桶，每个桶中有一个锁，从而达到减少冲突的目的。

在该实验中，实质上使用了一个哈希链表如下图所示，通过访问数组形式的哈希桶（每个桶作为链表头）来遍历所有的块，下图是一个类似的参考：



2. 修改`kernel/buf.h`中对于`buf`结构的定义，使其记录下该块最近的使用时间`LUtime`（Latest Used time），和当前属于哪个哈希桶的标识`curBucket`（用于遍历时方便释放和获取锁）：

```
struct buf {
    int valid; // has data been read from disk?
    int disk;  // does disk "own" buf?
    uint dev;
    uint blockno;
    struct sleeplock lock;
    uint refcnt;
    struct buf *prev; // LRU cache list
    struct buf *next;
    uchar data[BSIZE];

    uint LUtime;   // 最近使用时间
    int curBucket; // 当前属于哪个哈希桶
};
```

3. 在`kernel/bio.c`中加入一些宏定义和哈希函数，用于完成从块号到哈希索引的转换：

```
#define NBUCKET 13 // 哈希桶上限，质数个桶能减少聚集概率
#define INTMAX 0x7fffffff

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

4. 修改`bcache`结构体，维护`NBUCKET`个桶，每个桶指向一个链表，每个桶也有一个属于自己的锁存储在`bucket_locks`数组中：

```
struct {
    struct spinlock lock;
    struct buf buf[NBUF];

    // 维护NBUCKET个桶，每个桶维护一个链表
    // 每个桶都有一个自己的锁，用于保护自己的链表
    // 而每个桶中存储的元素buf作为缓冲区存在自己的锁
    struct buf bucket[NBUCKET];
    struct spinlock bucket_locks[NBUCKET];
} bcache;
```

5. 修改初始化函数`binit()`，使其能够完成桶初始化和块初始化，其中对于桶和`buffer cache`的命名应当依据要求：

```
void binit(void)
{
    initlock(&bcache.lock, "bcache_lock");

    // 初始化bucket
    char name[32];
    for (int i = 0; i < NBUCKET; i++) {
        snprintf(name, 32, "bucket_lock_%d", i);
        initlock(&bcache.bucket_locks[i], name);
        bcache.bucket[i].next = 0;
    }

    // 初始化buffer
    for (int i = 0; i < NBUF; i++) {
        struct buf *b = &bcache.buf[i]; // 链表头指针
        initsleeplock(&b->lock, "buffer");

        b->LUtime = 0;
        b->refcnt = 0;
        b->curBucket = 0;

        // 将buffer加入到bucket[0]中
        b->next = bcache.bucket[0].next;
        bcache.bucket[0].next = b;
    }
}
```

6. 修改`bget()`函数，有如下逻辑：
    - 通过索引获取锁，在当前桶中寻找空闲块
    - 若找到则返回，若没找到，去其他的桶中寻找空闲块，要先释放掉当前索引的桶的锁，再次获取其他索引的桶的锁，因为占有越多的锁，发生冲突的概率就越高
    - 依照LRU原则找到符合条件的块，具体代码如下，详情可见注释：

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

    uint index = hash(dev, blockno);
```

```
    acquire(&bcache.bucket_locks[index]);
    struct buf *b = bcache.bucket[index].next;

    // 在当前bucket[index]中查找
    while (b) {
        if (b->dev == dev && b->blockno == blockno) {
            // 已找到
            b->refcnt++;
            release(&bcache.bucket_locks[index]);
            acquiresleep(&b->lock);
            return b;
        }
        b = b->next;
    }

    // 未找到，需要从其他bucket中查找
    // 占有bucket_lock时，再获取其他bucket锁是不安全的，同时占有多个bucket锁
    // 容易导致循环等待造成死锁，故先释放当前bucket锁
    release(&bcache.bucket_locks[index]);

    // 因其他进程可能使用使用该块，因此要检查一遍该block是否已经在其他bucket中
    // 如果已经在其他bucket中，则等待获取sleeplock再返回即可
    acquire(&bcache.lock);
    b = bcache.bucket[index].next;
    while (b) {
        if (b->dev == dev && b->blockno == blockno) {
            // 已被其他进程放入其他bucket中
            acquire(&bcache.bucket_locks[index]);
            b->refcnt++;
            release(&bcache.bucket_locks[index]);
            release(&bcache.lock); // 释放bcache锁

            // 重新获取该block的锁，睡眠等待即可，等待完毕便可返回
            acquiresleep(&b->lock);
            return b;
        }
        b = b->next;
    }

    // 若还未找到，需要依据LRU从其他bucket中查找
    // 在当前bucket[index]中查找空闲缓冲区或者最近最少使用的缓冲区
    struct buf *LRUb = 0;
    uint curBucket = -1;
    uint LUtime = INTMAX;

    for (int i = 0; i < NBUCKET; i++) {
        acquire(&bcache.bucket_locks[i]);

        b = &bcache.bucket[i];
        int found = 0;

        while (b->next) {
            if (b->next->refcnt == 0 && LRUb == 0) {
                // 如果找到空闲缓冲区且之前还没有找到过空闲缓冲区
                LRUb = b;
                LUtime = b->next->LUtime;
                found = 1;
            }
```

```
            else if (b->next->refcnt == 0 && b->next->LUtime < LUtime) {
                // 如果找到空闲缓冲区，且该缓冲区上次使用时间更早
                LRUb = b;
                LUtime = b->next->LUtime;
                found = 1;
            }
            b = b->next;
        }
        if (found) {
            // 更新了LRUb，要释放这个桶之前的bucket锁
            if (curBucket != -1) {
                // 释放之前的bucket锁
                release(&bcache.bucket_locks[curBucket]);
            }
            curBucket = i;
        }
        else {
            // 没找到，释放访问的桶
            release(&bcache.bucket_locks[i]);
        }
    }
    if (LRUb == 0) {
        panic("bget: No buffer.");
    }
    else {
        struct buf *p = LRUb->next;

        if (curBucket != index) {
            // 删除LRUb节点
            LRUb->next = p->next;
            release(&bcache.bucket_locks[curBucket]);

            // 将LRUb节点放入当前bucket[index]中
            acquire(&bcache.bucket_locks[index]);
            p->next = bcache.bucket[index].next;
            bcache.bucket[index].next = p;
        }

        // 更新LRUb的信息
        p->dev = dev;
        p->blockno = blockno;
        p->refcnt = 1;
        p->valid = 0;
        p->curBucket = index;

        release(&bcache.bucket_locks[index]); // 释放bucket[index]锁
        release(&bcache.lock);                // 释放bcache锁
        acquiresleep(&p->lock);               // 获取LRUb的锁
        return p;
    }
}
```

7. 修改 `brelse()` 函数，若一个块引用数为0，即意味着完全释放，此时要把当前时间记录为最近使用时间：

```
void brelse(struct buf *b)
{
```

```
    if (!holdingsleep(&b->lock))
        panic("brelse");

    releasesleep(&b->lock);

    uint index = hash(b->dev, b->blockno);
    acquire(&bcache.bucket_locks[index]);
    b->refcnt--;
    if (b->refcnt == 0) {
        //没有进程引用这块buffer, 则为空闲状态释放, 记录最近使用时间
        b->LUtime = ticks;
    }
    release(&bcache.bucket_locks[index]);
}
```

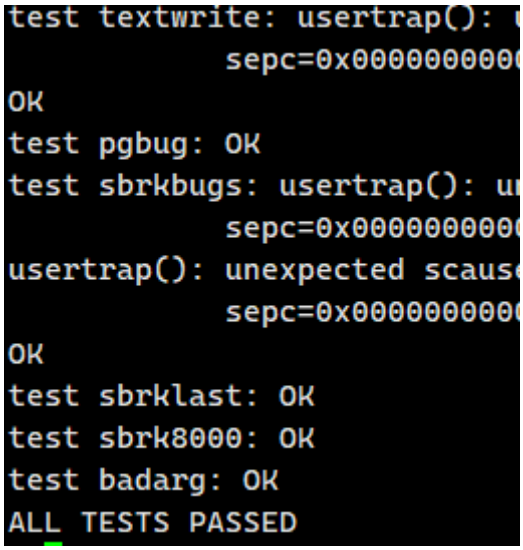
8. 修改`bpin()`和`bunpin()`函数，使其能正确增减引用数：

```
void bpin(struct buf *b)
{

    uint index = hash(b->dev, b->blockno);
    acquire(&bcache.bucket_locks[index]);
    b->refcnt++;
    release(&bcache.bucket_locks[index]);
}

void bunpin(struct buf *b)
{

    uint index = hash(b->dev, b->blockno);
    acquire(&bcache.bucket_locks[index]);
    b->refcnt--;
    release(&bcache.bucket_locks[index]);
}
```

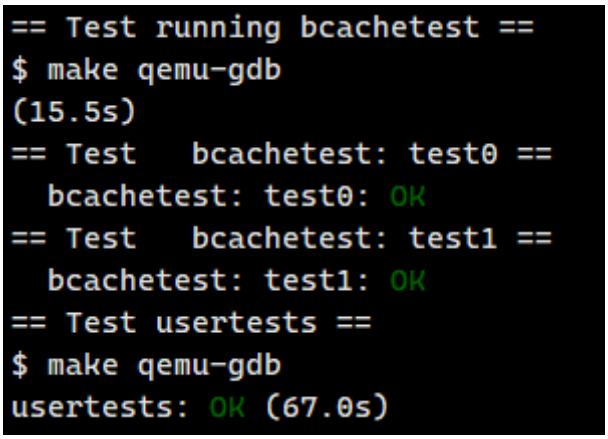
9. 运行`bcachetest`，得到结果如下：

```
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33053
lock: bcache_lock: #test-and-set 0 #acquire() 85
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 409756 #acquire() 1104
lock: proc: #test-and-set 24919 #acquire() 256506
lock: proc: #test-and-set 24824 #acquire() 256510
lock: proc: #test-and-set 23479 #acquire() 256501
lock: proc: #test-and-set 8331 #acquire() 256492
tot= 0
test0: OK
start test1
test1 OK
```

10. 运行`usertests`，得到结果如下：



11. 运行`grade`程序，结果如下：



2.3 实验中遇到的问题和解决方法

- **问题：** 为什么要使用哈希桶，而且大小要规定为质数13?

  解决方法： 使用哈希桶的目的是为了细化粒度（相较于`bcache`过大，`buf`过小），使用质数13是为了减少哈希冲突的概率。若使用到了相同的索引，则称为产生了哈希冲突，从而可能导致哈希聚集，使遍历效率下降，可考虑二次哈希来降低聚集。但如果使用的是非质数大小的桶，可能会导致不良后果，例如，假设哈希表的容量为15，某个key经过双哈希函数后得到的数组下标为0，步长为5。那么这个探测过程的序列是0,5,10,0,5,10，一直只会尝试这三个位置，永远找不到空白位置来存放，最终会导致崩溃。但在本题当中使用的是哈希链表，不以质数作为桶的大小尽管不会崩溃，但会增加聚集而导致浪费。

- **问题：** 多个结构体中锁的关系混乱

  解决方法： 多个缓冲区可能映射到同一个哈希桶，所以在访问这些哈希桶时，需要使用独立的`spinlock`来保护它们，以避免并发访问时的冲突。`buf`中的`sleeplock`用于保护单个缓冲区的读写操作，而`struct spinlock bucket_locks[NBUCKET]`数组用于保护哈希桶的访问。`bcache`中还有一个`lock`用于控制自身的访问。

2.4 实验心得

这次实验是对操作系统内核中缓冲区缓存的优化，通过并发控制和数据结构的调整，减少锁争用以提高系统性能。实验中，我学会了面对共享资源上的竞争情况时如何处理多个进程之间的并发问题。在优化缓冲区缓存的过程中，我深入理解了xv6操作系统的内核设计和数据结构。哈希表和哈希桶的设计让我对并发数据结构有了更深刻的认识，同时也学会了如何使用锁来确保共享资源的安全访问。同时为了避免死锁，需要小心处理可能导致死锁的情况，在移动缓冲区块时，可能需要同时持有多个锁，因此需要谨

慎处理。在使用哈希表的过程中，我又对哈希表的设计、使用、原理有了更加深入的认识，总而言之，此次实验让我受益匪浅。