Lab8-Locks

Memory allocator

实验目的

- 改进xv6操作系统的内存分配器，减少锁争用问题
  - 为了减少锁争用，需要重新设计内存分配器，使其避免单一锁和单一自由列表。具体方法是为每个CPU核心维护一个自由列表，每个列表都有自己的锁，从而允许不同CPU上的分配和释放操作并行运行。
- 分析锁争用问题，实现窃取机制
  - 窃取机制：解决一个CPU的自由列表为空而另一个CPU的自由列表有空闲内存的情况，实现一个CPU可以"窃取"另一个CPU自由列表的一部分内存。这种窃取可能会引入锁争用，但希望这种情况不频繁发生。

实验步骤

1. 为每个CPU核心维护一个kmem,包含自由列表和锁，`NCPU`是指CPU核心的数量

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU]; // 每个CPU核心都维护一个kmem
```

2. 由于kmem现在是一个数组，需要修改`kinit()`函数，以初始化所有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);
}
```

3. 修改`kalloc`函数，实现从当前CPU的自由列表中分配一个空闲页，如果当前CPU没有空闲页，就会从其他CPU的自由列表中窃取一个空闲页。

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

    // 关闭中断，防止中断过程中修改共享数据
    push_off();

    // 获取当前CPU的ID
    int CPUID = cpuid();

    // 获取当前CPU的kmem锁，保护共享数据
    acquire(&kmem[CPUID].lock);

    // 在当前CPU的自由列表中查找空闲页
    r = kmem[CPUID].freelist;

    // 如果在当前CPU的自由列表中找到空闲页
    if (r)
        kmem[CPUID].freelist = r->next;

    // 如果当前CPU的自由列表中没有空闲页
    if (r == 0) {
        // 在其他CPU的自由列表中查找空闲页
        for (int i = 0; i < NCPU; i++) {
            if (i == CPUID)
                continue; // 跳过当前CPU

            // 获取其他CPU的kmem锁
            acquire(&kmem[i].lock);

            // 在其他CPU的自由列表中查找空闲页
            r = kmem[i].freelist;
            if (r)
                kmem[i].freelist = r->next;

            // 释放其他CPU的kmem锁
            release(&kmem[i].lock);

            // 如果找到空闲页，跳出循环
            if (r)
                break;
        }
    }

    // 释放当前CPU的kmem锁
    release(&kmem[CPUID].lock);

    // 恢复中断
    pop_off();
```

4. 修改kfree函数，首先检查传入的地址是否有效，然后用垃圾数据填充该页。接着关闭中断，获取当前CPU的ID和kmem锁，将空闲页插入当前CPU的自由列表头部，释放锁并恢复中断

```
void kfree(void *pa)
{
    struct run *r;

    // 检查给定的地址是否有效
    if (((uint64)pa % PGSIZE) != 0 || (char *)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // 用垃圾数据填充页，以捕获悬空引用
    memset(pa, 1, PGSIZE);

    r = (struct run *)pa;

    // 关闭中断，防止中断过程中修改共享数据
    push_off();

    // 获取当前CPU的ID
    int CPUID = cpuid();

    // 获取当前CPU的kmem锁，保护共享数据
    acquire(&kmem[CPUID].lock);

    // 从链表头插入空闲页
    r->next = kmem[CPUID].freelist;
    kmem[CPUID].freelist = r;

    // 释放当前CPU的kmem锁
    release(&kmem[CPUID].lock);

    // 恢复中断
    pop_off();
}
```

5. 测试

运行 `make qemu` 后执行 `kalloctest`：

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 356
--- top 5 contended locks:
lock: proc: #test-and-set 130291 #acquire() 585004
lock: proc: #test-and-set 128844 #acquire() 584967
lock: proc: #test-and-set 110929 #acquire() 985136
lock: proc: #test-and-set 110430 #acquire() 985129
lock: proc: #test-and-set 98731 #acquire() 585006
tot= 0
test1 OK
start test2
total free number of pages: 32497 (out of 32768)
.....
test2 OK
start test3
usertrap(): unexpected scause 0x000000000000000f pid=6
            sepc=0x000000000000039e stval=0x0000000000000003
child done 1
test3 OK
```

执行usertests sbrk:

```
$ usertests sbrk
usertests starting
ALL TESTS PASSED
```

执行usertest:

```
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
usertests slow tests starting
test bigdir: OK
test manywrites: OK
test badwrite: OK
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
```

运行 make grade

```
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test   kalloctest: test3 ==
  kalloctest: test3: OK
```

实验中遇到的问题和解决办法

在第一次修改kalloc和kfree时因为没有关闭中断导致无法通过测试，后查询资料后才意识到需要关闭中断，在多核系统中，关闭中断可以防止当前CPU在处理临界区代码时被中断，从而避免在中断过程中修改共享数据，确保临界区代码的原子性。如果在获取锁之前发生中断，可能会导致死锁或不一致的状态。因此需要在获取锁之前关闭中断，在释放锁之后恢复中断。

实验心得

在这次实验中，我深刻体会到了多核系统中内存管理的复杂性和挑战性。刚开始时，我对内存分配器的实现有些信心满满，认为只要按照步骤来进行修改和优化，就能顺利完成任务。然而，现

实远比想象中的复杂。

在修改`kalloc`和`kfree`函数时，我一度忽略了关闭中断的重要性。最初的代码虽然能够编译通过，但在实际运行时却频繁出现不可预见的错误和锁争用问题。经过多次调试，我终于意识到，在多核环境中，确保数据操作的原子性是多么关键。关闭中断不仅仅是为了防止数据竞争，更是为了保证系统的稳定性和一致性。通过这次实验，我不仅掌握了多核内存分配的技术要点，更深刻理解了系统设计中的一些重要原则。

Buffer cache

实验目的

- 改进xv6操作系统的块缓存管理机制，减少在多进程使用文件系统时产生的锁争用问题
- 实现并维护缓存不变性，确保缓存中每个块只有一个副本

实验步骤

1. 改进方案说明：当文件系统被多个进程频繁访问时，它们可能会争夺`bcache.lock`，导致性能下降。为了优化这一问题，我们需要减少锁冲突的概率。直接将锁粒度缩小到每个块是不切实际的，因为这样会增加管理复杂度和锁开销。于是使用哈希表维护一组固定数量的哈希桶，每个桶有一个独立的锁。选择一个素数（如 13）作为哈希桶的数量，降低哈希冲突的概率。每个哈希桶的锁只负责保护该桶中的缓存块，限制锁争用范围，减少冲突。

2. 修改`kernel/buf.h`中的`buf`结构体，增加`LUtime`（最新使用时间）和`curBucket`（当前所属哈希桶）字段，方便查找和管理：

```
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;      // 当前所属哈希桶
};
```

3. 修改`bcache`结构体，增加`NBUCKET`个桶，每个桶指向一个链表，每个桶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;
```

4. 现在需要维护`NBUCKET`个哈希桶，因此需要跟着修改`binit()`，完成对桶的初始化

```
void binit(void)
{
    // 初始化全局缓存锁
    initlock(&bcache.lock, "bcache_lock");

    // 初始化每个bucket的锁
    char name[32];
    for (int i = 0; i < NBUCKET; i++) {
        snprintf(name, 32, "bucket_lock_%d", i);  // 生成锁的名称
        initlock(&bcache.bucket_locks[i], name);  // 初始化bucket锁
        bcache.bucket[i].next = 0;  // 初始化bucket链表头指针为0
    }

    // 初始化每个buffer
    for (int i = 0; i < NBUF; i++) {
        struct buf *b = &bcache.buf[i];  // 获取第i个buffer
        initsleeplock(&b->lock, "buffer");  // 初始化buffer的sleeplock锁

        b->LUtime = 0;  // 初始化最近使用时间
        b->refcnt = 0;  // 初始化引用计数
        b->curBucket = 0;  // 初始化当前所属bucket为0

        // 将buffer加入到bucket[0]的链表中
        b->next = bcache.bucket[0].next;  // 将buffer的next指向当前bucket[0]的链表
头
        bcache.bucket[0].next = b;  // 更新bucket[0]的链表头指针，使其指向新的buffer
    }
}
```

5. 修改`bget`函数，实现了通过哈希表在缓存中查找指定的块，如果找不到则根据LRU算法在其他哈希桶中寻找或替换缓存块并更新其信息。

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

    // 首先尝试在当前哈希桶中查找
    acquire(&bcache.bucket_locks[index]);
    for (b = bcache.bucket[index].next; b; b = b->next) {
        if (b->dev == dev && b->blockno == blockno) {
            b->refcnt++;
            release(&bcache.bucket_locks[index]);
            acquiresleep(&b->lock);
            return b;
        }
    }
    release(&bcache.bucket_locks[index]);

    // 检查是否在其他桶中
    acquire(&bcache.lock);
    for (int i = 0; i < NBUCKET; i++) {
        if (i != index) {
            acquire(&bcache.bucket_locks[i]);
            for (b = bcache.bucket[i].next; b; b = b->next) {
                if (b->dev == dev && b->blockno == blockno) {
                    b->refcnt++;
                    release(&bcache.bucket_locks[i]);
                    release(&bcache.lock);
                    acquiresleep(&b->lock);
                    return b;
                }
            }
            release(&bcache.bucket_locks[i]);
        }
    }

    // 进行LRU替换
    struct buf *LRUb = 0;
    uint LUtime = UINT_MAX;
    int curBucket = -1;

    for (int i = 0; i < NBUCKET; i++) {
        acquire(&bcache.bucket_locks[i]);
        for (b = &bcache.bucket[i]; b->next; b = b->next) {
            if (b->next->refcnt == 0 && b->next->LUtime < LUtime) {
                LRUb = b;
                LUtime = b->next->LUtime;
                curBucket = i;
            }
        }
        if (curBucket != i) {
            release(&bcache.bucket_locks[i]);
```

```
            }
        }

        if (LRUb == 0) {
            release(&bcache.lock);
            panic("bget: No buffer.");
        }

        // 从LRU缓存中删除并插入到当前桶
        struct buf *p = LRUb->next;
        LRUb->next = p->next;

        if (curBucket != index) {
            release(&bcache.bucket_locks[curBucket]);
            acquire(&bcache.bucket_locks[index]);
            p->next = bcache.bucket[index].next;
            bcache.bucket[index].next = p;
        }

        p->dev = dev;
        p->blockno = blockno;
        p->refcnt = 1;
        p->valid = 0;
        p->curBucket = index;

        release(&bcache.bucket_locks[index]);
        release(&bcache.lock);
        acquiresleep(&p->lock);
        return p;
    }
```

6. 修改`brelse()`函数，当一个块引用数为0时将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]);
}
```

7. 修改 `bpin()` 和 `bunpin()` 函数,使其适应引入哈希桶之后的情况

```
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]);
}
```

8. 测试

运行 `make qemu` 后执行 `bcahetest`

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache_lock: #test-and-set 0 #acquire() 89
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 737580 #acquire() 1116
lock: proc: #test-and-set 435128 #acquire() 1043098
lock: proc: #test-and-set 348079 #acquire() 1043098
lock: proc: #test-and-set 328430 #acquire() 1043098
lock: proc: #test-and-set 202892 #acquire() 1043098
tot= 0
test0: OK
start test1
test1 OK
```

运行`make grade`：

```
== Test running kalloctest ==
$ make qemu-gdb
(175.1s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test   kalloctest: test3 ==
  kalloctest: test3: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (14.7s)
== Test running bcachetest ==
$ make qemu-gdb
(17.4s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (86.6s)
== Test time ==
time: OK
Score: 80/80
```

实验中遇到的问题和解决办法

这个实验中涉及到了三种锁比较容易混淆，下面是对三种锁的梳理，分别是**bcache.lock**（Spinlock，自旋锁），保护整个缓存系统，确保在访问和修改全局缓存状态时不会发生数据竞争，主要用于全局性的操作，例如在其他桶中查找缓存块或进行LRU替换时，防止多个进程同时修改全局状态。；**bcache.bucket_locks[]**，(Spinlock，自旋锁)每个哈希桶拥有一个自旋锁，用于保护桶中的缓存块。这些锁的粒度较小，只保护特定哈希桶中的数据，减少了锁争用，允许多个进程并发地访问不同的哈希桶。；**buf.lock**，（Sleeplock，睡眠锁）每个缓存块都有一个睡眠

锁，用于保护单个缓存块的数据一致性。睡眠锁允许进程在访问或修改缓存块的数据时进入睡眠状态，直到锁可用为止。它用于长时间持有锁的操作，例如读取或写入磁盘数据时，确保对该缓存块的独占访问。

实验心得

这次实验让我深刻体会到了操作系统中锁机制的重要性，以及在实际应用中平衡性能和复杂性的挑战。最初，我对锁的理解仅停留在理论层面，认为只要添加锁就能保证数据的一致性。然而，在实际操作中，我发现简单的锁机制往往会带来性能上的瓶颈，特别是在多进程并发访问时。

在修改 `bcache` 的过程中，最开始采用最简单的全局锁来保护整个缓存系统。虽然这样可以保证数据的一致性，但随之而来的锁争用问题导致系统性能大幅下降。在运行 `bcachetest` 时，我看到锁争用次数远超预期，系统效率低下，这让我意识到必须优化锁的粒度。

于是，我引入了哈希表和细粒度的桶锁。每个桶独立维护一个锁，这样在多进程访问不同桶时，可以并行操作而不产生冲突。这种改进大大减少了锁争用次数，系统性能显著提升。在此过程中，我还学会了如何平衡锁的粒度和管理复杂性，避免直接为每个缓存块加锁带来的高开销。通过这次实验，我不仅掌握了锁的实际应用技巧，也深刻体会到理论与实践结合的重要性。锁机制在保证数据一致性的同时，如何设计和优化锁策略以提高系统性能，是一个需要不断探索和实践的问题。

总结

本实验旨在改进Xv6操作系统的内存分配器和块缓存管理机制，以减少锁争用问题，提高多核系统的并行性能。具体目标包括为每个CPU核心维护一个自由列表，实现窃取机制，改进块缓存管理，维护缓存一致性。

- Memory allocator（1种锁）
    - **目的**:
        - 改进内存分配器，减少锁争用问题。
        - 通过为每个CPU核心维护一个自由列表和实现窃取机制，减少锁争用并提高并行性能。
    - **心得**： 通过为每个CPU核心维护独立的自由列表和锁，减少了锁争用问题，提高了多核系统的内存分配效率。实现窃取机制，确保在某个CPU自由列表为空时仍能高效分配内存。通过关闭中断，防止中断过程中修改共享数据，确保临界区代码的原子性。
- Buffer cache(3种锁)
    - **目的**:
        - 改进块缓存管理机制，减少在多进程使用文件系统时的锁争用问题。
        - 实现并维护缓存不变性，确保缓存中每个块只有一个副本。

- **心得**： 通过引入哈希桶和细粒度锁，减少了块缓存管理中的锁争用问题，提高了系统性能。哈希表和LRU替换策略相结合，确保缓存中每个块只有一个副本，同时减少了锁冲突。理解了不同锁的作用和使用场景，确保数据一致性的同时提高并行性能。