

Lab5: locks

PB20050987 梁兆懿

本实验需要更改xv6 内存分配器和块缓存的数据结构和锁定策略，以减少争用，提高多核计算机并行性。首先我们需要切换内存分支到lock：

```
$ git fetch
$ git checkout lock
$ make clean
```

Memory allocator

该实验需要对XV6内核的内存页面分配器进行改进，实现增加各个CPU分配物理内存的效率。要消除xv6系统的锁争用，需要重新设计内存分配器以避免单个锁和列表。基本思想是为每个CPU维护一个空闲列表，每个列表都有自己的锁。不同CPU上的分配和释放可以并行运行，因为每个CPU将在不同的列表上运行。

实验代码

根据提示，我们定义NCPU个kmem结构体的数组代替原有的kmem结构体，在kalloc.c文件中：

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

struct {
    struct spinlock lock;
    struct run *freelist;
    char lock_name[7];
} kmem[NCPU];
```

在kinit函数中，我们对每个锁都进行初始化：

```
void
kinit()
{
    for (int i = 0; i < NCPU; i++) {
        snprintf(kmem[i].lock_name, sizeof(kmem[i].lock_name), "kmem_%d", i);
        initlock(&kmem[i].lock, kmem[i].lock_name);
    }
    freerange(end, (void*)PHYSTOP);
}
```

我们为每个CPU核心分配一个空闲链表，kalloc和kfree都在本核心的链表上进行，只有在当前核心的链表为空时才去访问其他核心的链表。对于kfree函数，需要将释放的页面插入到当前核心对应链表上，借助提示，我们用push_off()和pop_off()来关闭和打开中断，在中断时使用函数cupid获取当前内核编号：

```
void
```

```

kfree(void *pa)
{
    ...

    r = (struct run*)pa;
    push_off();
    int id = cpuid();
    acquire(&kmem[id].lock);
    r->next = kmem[id].freelist;
    kmem[id].freelist = r;
    release(&kmem[id].lock);
    pop_off();
}

```

而对于 `kalloc` 函数，我们首先在当前核心是申请页面；前若申请失败时，就尝试从其他核心上获取页面。

```

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

    push_off();
    int id = cpuid();
    acquire(&kmem[id].lock);
    r = kmem[id].freelist;
    if(r) kmem[id].freelist = r->next;
    else {
        int temp = 0, i = 0;
        for(i = 0; i < NCPU; i++) {
            if (i == id) continue;
            acquire(&kmem[i].lock);
            struct run *p = kmem[i].freelist;
            if(p) {
                struct run *fp = p;
                struct run *pre = p;
                while (fp && fp->next) {
                    fp = fp->next->next;
                    pre = p;
                    p = p->next;
                }
                kmem[id].freelist = kmem[i].freelist;
                if (p == kmem[i].freelist) {
                    kmem[i].freelist = 0;
                }
                else {
                    kmem[i].freelist = p;
                    pre->next = 0;
                }
                temp = 1;
            }
            release(&kmem[i].lock);
        }
        if (temp) {
            r = kmem[id].freelist;
            kmem[id].freelist = r->next;
            break;
        }
    }
}

```

```

    }
}
}
release(&kmem[id].lock);
pop_off();

if(r)memset((char*)r, 5, PGSIZE);
return (void*)r;
}

```

实验结果

```

ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
lock: bcache41: #fetch-and-add 0 #acquire() 2
lock: bcache47: #fetch-and-add 0 #acquire() 1280
lock: bcache87: #fetch-and-add 0 #acquire() 8
lock: bcache88: #fetch-and-add 0 #acquire() 2
lock: bcache89: #fetch-and-add 0 #acquire() 2
lock: bcache90: #fetch-and-add 0 #acquire() 2
lock: bcache91: #fetch-and-add 0 #acquire() 4
lock: bcache92: #fetch-and-add 0 #acquire() 2
lock: bcache105: #fetch-and-add 0 #acquire() 8
lock: bcache106: #fetch-and-add 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #fetch-and-add 1051014 #acquire() 260530
lock: uart: #fetch-and-add 595286 #acquire() 300
lock: proc: #fetch-and-add 572690 #acquire() 260530
lock: proc: #fetch-and-add 562002 #acquire() 260530
lock: proc: #fetch-and-add 552595 #acquire() 260530
tot= 0
test1 OK
start test2
total free number of pages: 32447 (out of 32768)
.
....
test2 OK
$ $

```

Buffer cache

该实验需要对XV6的磁盘缓冲区进行优化，以便使得缓冲区中所有锁的获取循环迭代次数接近于零。在初始的XV6磁盘缓冲区中是使用一个LRU链表来维护的，而这就导致了每次获取、释放缓冲区时就要对整个链表加锁。

实验代码

根据提示，我们使用哈希表来代替链表，提高并行性能。使用固定数量的桶，每个哈希桶在具有锁的哈希表在缓存中查找块号。在文件 kalloc.c 中：

```

struct {
    struct spinlock lock;
    char name[10];
    struct buf buf[NSIZE];
} bcache[BUCKETNUM];

int hash(uint dev, uint blockno) {
    return blockno % BUCKETNUM;
}

```

与此同时，我们在文件 `param.h`，增加桶长度和链表长度的宏定义：

```
#define BUCKETNUM    108
#define NSIZE        2
```

在文件 `buf.h` 中，对结构体增加时间标识 `timestamp`，从而可以标识最近最少使用，并且删去 `prev` 域：

```
struct buf {
    int valid;
    int disk;
    uint dev;
    uint blockno;
    struct sleeplock lock;
    uint refcnt;
    // struct buf *prev;
    struct buf *next;
    uchar data[BSIZE];
    uint timestamp;
};
```

在 `binit` 函数中，我们对哈希表进行初始化：

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

    for (int i = 0; i < BUCKETNUM; i++) {
        // 为每个桶设置锁并初始化
        snprintf(bcache[i].name, 10, "bcache%d", i);
        initlock(&bcache[i].lock, bcache[i].name);
        for (int j = 0; j < NSIZE; j++) {
            b = &bcache[i].buf[j];
            b->refcnt = 0;
            initsleeplock(&b->lock, "buffer");
            b->timestamp = ticks;
        }
    }
}
```

在 `brelease` 函数中对 `timestamp` 标识进行更新，并且需要将链表的锁替换掉：

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

    releasesleep(&b->lock);

    int id = hash(b->dev, b->blockno);
    acquire(&bcache[id].lock);
    b->refcnt--;
    if (b->refcnt == 0) {
        // no one is waiting for it.
```

```

        b->timestamp = ticks;
    }

    release(&bcache[id].lock);
}

```

而后我们需要修改bget函数，对应的桶当中查找当前块是否被缓存，若当前块被缓存就直接返回；否则需要查找一个块并将其替换。我们在全局数组中查找时，要先加上锁，在找到对应的块后，就可以根据块的信息查找到对应的桶并且对该桶加锁，将块从桶的链表中删去，然后释放锁，最后再添加到当前桶的链表上去。

```

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

    int id = hash(dev, blockno);
    acquire(&bcache[id].lock);

    for(int i = 0; i < BUCKETSIZ; i++){
        b = &bcache[id].buf[i];
        if(b->dev == dev && b->blockno == blockno){
            b->refcnt++;
            release(&bcache[id].lock);
            acquiresleep(&b->lock);
            return b;
        }
    }
    uint min = -1;
    struct buf* tmp = 0;
    for (int i = 0; i < BUCKETSIZ; i++) {
        b = &bcache[id].buf[i];
        if (b->refcnt == 0 && b->timestamp < min) {
            min = b->timestamp;
            tmp = b;
        }
    }
    if (min != -1) {
        tmp->dev = dev;
        tmp->blockno = blockno;
        tmp->valid = 0;
        tmp->refcnt = 1;
        release(&bcache[id].lock);
        acquiresleep(&tmp->lock);
        return tmp;
    }

    panic("bget: no buffers");
}

```

最后，我们还需要将bpin和bunpin函数的单个锁替换为桶数组的锁：

```

void
bpin(struct buf *b) {
    int id = hash(b->dev, b->blockno);

```

```

    acquire(&bcache[id].lock);
    b->refcnt++;
    release(&bcache[id].lock);
}

void
bunpin(struct buf *b) {
    int id = hash(b->dev, b->blockno);
    acquire(&bcache[id].lock);
    b->refcnt--;
    release(&bcache[id].lock);
}

```

实验结果

```

ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
== Test running kalloc test ==
$ make qemu-gdb
(287.7s)
== Test    kalloc test: test1 ==
    kalloc test: test1: OK
== Test    kalloc test: test2 ==
    kalloc test: test2: OK
== Test kalloc test: sbrkmuch ==
$ make qemu-gdb
kalloc test: sbrkmuch: OK (29.0s)
== Test running bcachetest ==
$ make qemu-gdb
(69.0s)
== Test    bcachetest: test0 ==
    bcachetest: test0: OK
== Test    bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (432.6s)
== Test time ==
time: OK
Score: 70/70
ubuntu@VM5878-LZY: /home/ubuntu/文档/xv6-labs-2020$

```

实验感受与收获

这次实验虽然难度更上一层楼，但是相较于上次实验顺利得多，可能由于课本的相关知识掌握的比较好，以及提示比较清晰的缘故。和上一个实验相比，这次实验更考验并行思维，主要体现在锁的应用。如果没有提示使用哈希表来提高并行性能...那我大抵是写一个星期也不能pass的。桶级锁在提高并行性的几个实验都能用到，效果显著。在使用双向链表需要格外小心，否则很容易报错。最后在测试实验时，可能由于电脑性能原因，实验时间较长，因此增加了测试时间避免报错。