

# Lab8 locks

---

## Lab8 locks

前置知识

实验内容

### Memory allocator (moderate)

任务

步骤（只涉及 `kernel/kalloc.c`）

1. `kernel/kalloc.c` 结构体修改
2. 初始化逻辑
3. 内存释放逻辑
4. 内存分配逻辑

测试成功

### Buffer cache (hard)

任务

设计方案：

步骤

1. 缓存结构的变化
2. 初始化函数的变化
3. 缓冲区获取函数 `bget` 的变化
4. 缓冲区释放函数 `brelease` 的变化

测试成功

实验得分

## 前置知识

## 实验内容

### Memory allocator (moderate)

#### 任务

**问题描述：** 当前的xv6内存分配器使用了单一的自由列表和单一的锁（`kmem.lock`），在多核机器上会造成严重的锁竞争。实验程序 `user/kalloc_test` 会测试xv6的内存分配器，测量在获取`kmem`锁时的循环次数（即 `acquire` 调用的循环次数）作为锁竞争的粗略衡量指标。

你需要通过重构内存分配器，减少kmem锁的竞争次数。

**解决方案：** 你的任务是为每个CPU维护一个自由列表（freelist），每个列表有各自的锁。这样，不同CPU上的分配和释放操作可以并行进行，因为每个CPU操作的是不同的列表。

- 当一个CPU的自由列表为空，而另一个CPU的列表中有空闲内存时，CPU需要从另一个CPU的自由列表中“偷取”内存。虽然这种“偷取”可能会引入锁竞争，但通常会比较少见。
- 所有的锁名称都应以“kmem”开头，你应该调用 `initlock` 并传递一个以“kmem”开头的名称。

## 步骤（只涉及kernel/kalloc.c）

### 1. kernel/kalloc.c 结构体修改

- 修改前：

```
1 struct {
2     struct spinlock lock;
3     struct run *freelist;
4 } kmem;
```

- 原始代码中，`kmem` 结构体包含一个 `spinlock` 和一个 `freelist`。这意味着所有的 CPU 都共用一个自由列表和一个锁。

- 修改后：

```
1 struct {
2     struct spinlock lock;
3     struct run *freelist;
4 } kmems[NCPU];
```

- 修改后的代码将 `kmem` 改为 `kmems[NCPU]`，为每个 CPU 分配了一个独立的 `freelist` 和 `spinlock`。这样每个 CPU 都有自己的内存分配和锁机制，减少了多个 CPU 同时访问同一锁时的竞争。

**目的：** 通过为每个 CPU 配置独立的 `freelist` 和锁，减少多核系统对单一锁的竞争，从而提高系统的并行处理能力。

## 2. 初始化逻辑

- 修改前:

```
1 void
2 kinit()
3 {
4     initlock(&kmem.lock, "kmem");
5     freerange(end, (void*)PHYSTOP);
6 }
```

- 原始代码中, `kinit()` 只为 `kmem.lock` 初始化, 并没有考虑多核情况下的锁竞争问题。

- 修改后:

```
1 void
2 kinit()
3 {
4     for(int id = 0; id < NCPU; id++)
5     {
6         char name_buf[16] = {0};
7         snprintf(name_buf, sizeof(name_buf), "kmem_lock_%d",
8 id);
9         printf("kmem lock name: %s\n", name_buf);
10        initlock(&kmems[id].lock, name_buf);
11    }
12    printf("kinit(): initlock() complete. \n");
13    freerange(end, (void*)PHYSTOP);
14 }
```

- 修改后的代码中, `kinit()` 会为每个 CPU 初始化自己的锁, 并命名为 `kmem_lock_<CPU_ID>`, 这使得每个 CPU 都有自己的锁, 可以独立操作自己的 `freelist`。

目的: 通过独立初始化每个 CPU 的锁, 进一步减少锁的竞争, 提高系统的并发性。

## 3. 内存释放逻辑

- 修改前:

```

1 void
2 kfree(void *pa)
3 {
4     struct run *r;
5     ...
6     acquire(&kmem.lock);
7     r->next = kmem.freelist;
8     kmem.freelist = r;
9     release(&kmem.lock);
10 }

```

- 原始代码中，`kfree()` 操作会获取全局的 `kmem.lock`，将释放的内存块添加到全局的 `freelist` 中。
- 修改后：

```

1 void
2 kfree(void *pa)
3 {
4     struct run *r;
5     ...
6     int cpu_id = cpuid(); // 获取当前 CPU ID
7     acquire(&kmems[cpu_id].lock);
8     r->next = kmems[cpu_id].freelist;
9     kmems[cpu_id].freelist = r;
10    release(&kmems[cpu_id].lock);
11 }

```

- 修改后的代码中，`kfree()` 通过 `cpuid()` 获取当前 CPU 的 ID，然后在对应的 `kmems[cpu_id]` 上进行操作，这样每个 CPU 都能独立管理自己的内存释放。

目的: 让每个 CPU 独立管理自己的内存释放，减少多个 CPU 对同一锁的竞争。

#### 4. 内存分配逻辑

- 修改前：

```

1 void *
2 kalloc(void)
3 {
4     struct run *r;
5     ...
6     acquire(&kmem.lock);
7     r = kmem.freelist;
8     if(r)
9         kmem.freelist = r->next;
10    release(&kmem.lock);
11    ...
12    return (void*)r;
13 }

```

- 原始代码中，`kalloc()` 直接从全局的 `kmem.freelist` 中分配内存，这意味着所有 CPU 都必须竞争同一把锁。
- 修改后:

```

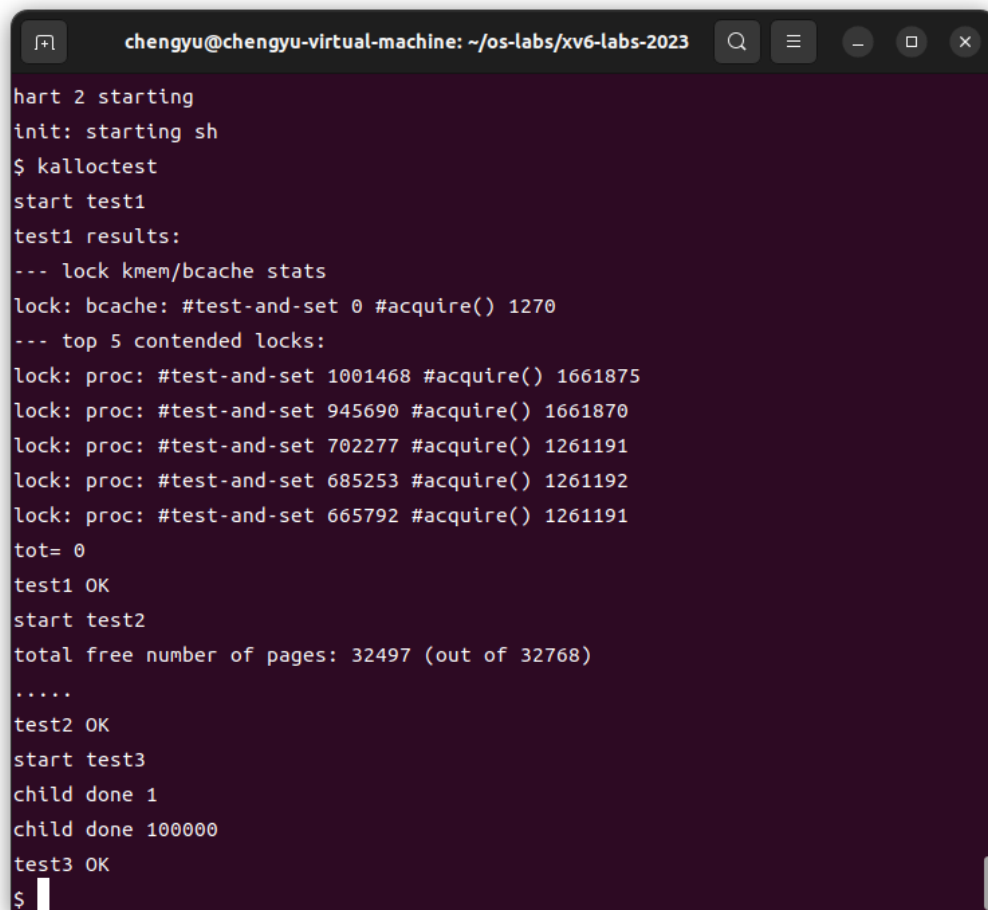
1 void *
2 kalloc(void)
3 {
4     struct run *r;
5     int cpu_id = cpuid(); // 获取当前 CPU ID
6     acquire(&kmems[cpu_id].lock);
7     r = kmems[cpu_id].freelist;
8     if(r)
9         kmems[cpu_id].freelist = r->next;
10    release(&kmems[cpu_id].lock);
11    for(int id = 0; !r && id < NCPU; id++)
12    {
13        if(id == cpu_id) continue;
14        acquire(&kmems[id].lock);
15        r = kmems[id].freelist;
16        if(r)
17            kmems[id].freelist = r->next;
18        release(&kmems[id].lock);
19    }
20    ...
21    return (void*)r;
22 }

```

- 修改后的代码中，`kalloc()` 首先尝试从当前 CPU 的 `freelist` 中分配内存。如果失败，会尝试从其他 CPU 的 `freelist` 中“偷取”内存。

目的: 优先从本地 CPU 的 `freelist` 分配内存，减少跨 CPU 的内存操作，降低锁竞争的概率，并在必要时通过“偷取”机制保持灵活性。

## 测试成功



```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 1270
--- top 5 contended locks:
lock: proc: #test-and-set 1001468 #acquire() 1661875
lock: proc: #test-and-set 945690 #acquire() 1661870
lock: proc: #test-and-set 702277 #acquire() 1261191
lock: proc: #test-and-set 685253 #acquire() 1261192
lock: proc: #test-and-set 665792 #acquire() 1261191
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
$
```

## Buffer cache ([hard](#))

### 任务

- 这个实验的目标是减少xv6操作系统中的块缓存（block cache）的锁争用，从而提高系统在多核环境下的并行性能。具体来说，你需要重构`kernel/bio.c`中的块缓存管理代码，降低多个进程同时访问块缓存时的锁竞争；

- 当多个进程密集使用文件系统时，可能会争用 `bcache.lock`，导致性能下降。实验程序 `bcachetest` 会创建多个进程，反复读取不同的文件，以生成对 `bcache.lock` 的争用。你需要修改块缓存的实现，使得 `bcache.lock` 的争用次数大幅降低。

## 设计方案：

- 使用哈希表：使用哈希表来查找缓存块，并为每个哈希桶（`bucket`）分配一个锁。这样可以减少全局锁 `bcache.lock` 的争用，因为不同的进程可以并行访问不同的哈希桶。
- 哈希表设计：
  - 选择一个合适的哈希函数，根据块号（`block number`）将块映射到不同的哈希桶中。
  - 使用固定数量的哈希桶，建议使用一个素数（如13）作为桶的数量，以减少哈希冲突的可能性。
- 移除全局缓存块列表：移除 `bcache.head` 等全局缓存块列表，并且不再实现LRU（最近最少使用）算法。这样可以避免在 `bre1se` 函数中获取 `bcache.lock`。
- 选择缓存块：
  - 在 `bget` 中，你可以选择任何引用计数为0的块，而不需要选择最近最少使用的块。
  - 如果查找缓存块失败，需要找到一个未使用的块来替换，这个过程可能需要放弃当前所有锁，并重新开始。
- 处理死锁：在某些情况下，可能需要同时持有两个锁（例如在进行块替换时，可能需要同时持有 `bcache.lock` 和哈希桶的锁）。你需要确保在这些情况下不会发生死锁。

## 步骤

### 1. 缓存结构的变化

- 改动前：

```

1 //kernel/bio.c
2 struct {
3     struct spinlock lock;
4     struct buf buf[NBUF];
5
6     // 链表，用于维护所有缓存块的顺序。
7     struct buf head;
8 } bcache;

```

- 原始代码中，整个缓存系统使用一个全局的 `bcache` 结构体，其中包括一个全局锁 `lock` 和一个包含所有缓存块的链表 `head`。所有对缓存的操作都需要获取这个全局锁，这在多核环境下容易产生锁竞争。
- 改动后：

```

1 struct {
2     struct spinlock lock;
3     struct buf buf[NBUF];
4
5     // 链表，用于维护每个哈希桶中的缓存块顺序。
6     struct buf head;
7 } bcaches[BCACHE_NUM]; // 哈希表

```

- 改动后的代码将缓存系统分成多个小的缓存池 `bcaches[BCACHE_NUM]`，每个缓存池都有自己的锁 `lock` 和缓存块链表 `head`。这些缓存池通过哈希表进行管理，不同的块号会被映射到不同的缓存池中，这样可以显著减少锁的争用。

目的：通过引入哈希表结构，减少多个进程对同一个全局锁的争用，允许多个进程同时对不同的缓存池进行操作，从而提高系统在多核环境下的并行性能。

## 2. 初始化函数的变化

- 改动前：

```

1 //kernel/bio.c
2 void
3 binit(void)
4 {
5     struct buf *b;
6
7     initlock(&bcache.lock, "bcache");
8

```



```

9      // 创建缓存块的链表
10     bcache.head.prev = &bcache.head;
11     bcache.head.next = &bcache.head;
12     for(b = bcache.buf; b < bcache.buf+NBUF; b++){
13         b->next = bcache.head.next;
14         b->prev = &bcache.head;
15         initsleeplock(&b->lock, "buffer");
16         bcache.head.next->prev = b;
17         bcache.head.next = b;
18     }
19 }

```

- 原始代码中，只初始化了一个全局锁 `bcache.lock` 和一个包含所有缓存块的链表 `bcache.head`。
- 改动后：

```

1  //kernel/bio.c
2  void
3  binit(void)
4  {
5      struct buf *b;
6
7      for(int id = 0; id < BCACHE_NUM; id++) {
8          char name_buf[16] = {0};
9          snprintf(name_buf, sizeof(name_buf),
10 "bcache_lock_%d", id);
11          printf("kmem lock name: %s\n", name_buf);
12          initlock(&bcaches[id].lock, name_buf);
13
14          // 创建每个哈希桶的缓存块链表
15          bcaches[id].head.prev = &bcaches[id].head;
16          bcaches[id].head.next = &bcaches[id].head;
17          for(b = bcaches[id].buf; b < bcaches[id].buf+NBUF;
18 b++){
19              b->next = bcaches[id].head.next;
20              b->prev = &bcaches[id].head;
21              initsleeplock(&b->lock, "buffer");
22              bcaches[id].head.next->prev = b;
23              bcaches[id].head.next = b;
24          }
25      }
26      printf("binit(): complete. \n");

```

- 改动后的代码为每个哈希桶（`bcaches` 数组中的每个元素）初始化自己的锁和缓存块链表。这允许多个进程在不同的哈希桶中并行操作缓存块。

目的: 通过初始化多个缓存池并为每个缓存池分配独立的锁, 进一步减少全局锁的争用, 从而提高并行性能。

### 3. 缓冲区获取函数 `bget` 的变化

- 改动前:

```

1 //kernel/bio.c
2 static struct buf*
3 bget(uint dev, uint blockno)
4 {
5     struct buf *b;
6
7     acquire(&bcache.lock);
8
9     // 查找缓存中是否存在目标块
10    for(b = bcache.head.next; b != &bcache.head; b = b->next){
11        if(b->dev == dev && b->blockno == blockno){
12            b->refcnt++;
13            release(&bcache.lock);
14            acquiresleep(&b->lock);
15            return b;
16        }
17    }
18
19    // 未缓存, 需要回收最近最少使用的缓存块
20    for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
21        if(b->refcnt == 0) {
22            b->dev = dev;
23            b->blockno = blockno;
24            b->valid = 0;
25            b->refcnt = 1;
26            release(&bcache.lock);
27            acquiresleep(&b->lock);
28            return b;

```

```

29     }
30 }
31 panic("bget: no buffers");
32 }

```

- 原始代码使用一个全局锁 `bcache.lock` 来保护对整个缓存链表的访问。
- 改动后:

```

1  static struct buf*
2  bget(uint dev, uint blockno)
3  {
4      struct buf *b;
5      int id = blockno % BCACHE_NUM;
6
7      acquire(&bcaches[id].lock);
8
9      // 查找缓存中是否存在目标块
10     for(b = bcaches[id].head.next; b != &bcaches[id].head;
11         b = b->next){
12         if(b->dev == dev && b->blockno == blockno){
13             b->refcnt++;
14             release(&bcaches[id].lock);
15             acquiresleep(&b->lock);
16             return b;
17         }
18     }
19
20     // 未缓存, 需要回收最近最少使用的缓存块
21     for(b = bcaches[id].head.prev; b != &bcaches[id].head;
22         b = b->prev){
23         if(b->refcnt == 0) {
24             b->dev = dev;
25             b->blockno = blockno;
26             b->valid = 0;
27             b->refcnt = 1;
28             release(&bcaches[id].lock);
29             acquiresleep(&b->lock);
30             return b;
31         }
32     }
33     panic("bget: no buffers");
34 }

```

- 改动后的代码根据 `bblockno` 计算哈希值，并将缓存块映射到对应的哈希桶中。然后，`bget` 函数只需要获取对应哈希桶的锁，而不是全局锁。

目的: 通过使用哈希表，减少对单一全局锁的依赖，允许不同的进程并行操作不同的哈希桶，从而提高并发性能。

#### 4. 缓冲区释放函数 `brelease` 的变化

- 改动前:

```
1 //kernel/bio.c
2 void
3 brelease(struct buf *b)
4 {
5     if(!holdingsleep(&b->lock))
6         panic("brelease");
7
8     releasesleep(&b->lock);
9
10    acquire(&bcache.lock);
11    b->refcnt--;
12    if (b->refcnt == 0) {
13        // 没有人在等待它
14        b->next->prev = b->prev;
15        b->prev->next = b->next;
16        b->next = bcache.head.next;
17        b->prev = &bcache.head;
18        bcache.head.next->prev = b;
19        bcache.head.next = b;
20    }
21
22    release(&bcache.lock);
23 }
```

- 原始代码在释放缓存块时，需要获取全局锁 `bcache.lock`，以更新全局缓存链表。

- 改动后:

```
1 void
2 brelease(struct buf *b)
3 {
4     if(!holdingsleep(&b->lock))
```

```

5     panic("brelse");
6
7     releasesleep(&b->lock);
8
9     int id = b->blockno % BCACHE_NUM;
10    acquire(&bcaches[id].lock);
11    b->refcnt--;
12    if (b->refcnt == 0) {
13        // 没有人在等待它
14        b->next->prev = b->prev;
15        b->prev->next = b->next;
16        b->next = bcaches[id].head.next;
17        b->prev = &bcaches[id].head;
18        bcaches[id].head.next->prev = b;
19        bcaches[id].head.next = b;
20    }
21
22    release(&bcaches[id].lock);
23 }

```

- 改动后的代码通过使用哈希表中的锁 `bcaches[id].lock`，只在对应的哈希桶内操作缓存块链表，而不再需要获取全局锁。

目的: 减少对全局锁的依赖，通过哈希桶的锁机制，实现更细粒度的并发控制。

测试成功

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
lock: bcache.bucket: #test-and-set 0 #acquire() 6178
lock: bcache.bucket: #test-and-set 0 #acquire() 6180
lock: bcache.bucket: #test-and-set 0 #acquire() 4276
lock: bcache.bucket: #test-and-set 0 #acquire() 4270
lock: bcache.bucket: #test-and-set 0 #acquire() 2262
lock: bcache.bucket: #test-and-set 0 #acquire() 4268
lock: bcache.bucket: #test-and-set 0 #acquire() 2678
lock: bcache.bucket: #test-and-set 0 #acquire() 4682
lock: bcache.bucket: #test-and-set 0 #acquire() 6452
lock: bcache.bucket: #test-and-set 0 #acquire() 6176
lock: bcache.bucket: #test-and-set 0 #acquire() 6176
lock: bcache.bucket: #test-and-set 0 #acquire() 6180
lock: bcache.bucket: #test-and-set 0 #acquire() 6180
--- top 5 contended locks:
lock: proc: #test-and-set 2405911 #acquire() 1444321
lock: proc: #test-and-set 2340309 #acquire() 1444101
lock: proc: #test-and-set 2143476 #acquire() 1444558
lock: proc: #test-and-set 1972978 #acquire() 1468423
lock: proc: #test-and-set 1934631 #acquire() 1468423
tot= 0
test0: OK
start test1
test1 OK
$
```

## 实验得分

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
(153.4s)
== Test   kallocetest: test1 ==
    kallocetest: test1: OK
== Test   kallocetest: test2 ==
    kallocetest: test2: OK
== Test   kallocetest: test3 ==
    kallocetest: test3: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (19.5s)
== Test running bcachetest ==
$ make qemu-gdb
(22.7s)
== Test   bcachetest: test0 ==
    bcachetest: test0: OK
== Test   bcachetest: test1 ==
    bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (106.7s)
== Test time ==
time: OK
Score: 80/80
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$
```