

# Lab8 locks

---

## Lab8 locks

前置知识

实验内容

### Memory allocator (moderate)

任务

添加 `lock`

修改 `freerange` 函数(`kernel/kalloc.c`)

修改 `kalloc` 和 `kfree` 函数

测试成功

### Buffer cache (hard)

任务

设计方案:

修改 `buf.h`

实现哈希表和桶锁:

添加参数 `kernel/param.h`

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/kalloctest` 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。

## 添加 `lock`

- 第一部分涉及到内存分配的代码，xv6 将空闲的物理内存 `kmem` 组织成一个空闲链表 `kmem.freelist`，同时用一个锁 `kmem.lock` 保护 `freelist`，所有对 `kmem.freelist` 的访问都需要先取得锁，所以会产生很多竞争。解决方案也很直观，给每个 CPU 单独开一个 `freelist` 和对应的 `lock`，这样只有同一个 CPU 上的进程同时获取对应锁才会产生竞争。

```
1 //kernel/kalloc.c
2 struct {
3     struct spinlock lock;
4     struct run *freelist;
5 } kmem[NCPU];
```

## 修改 `freerange` 函数(`kernel/kalloc.c`)

- 改前：

```
1 void
2 freerange(void *pa_start, void *pa_end)
3 {
4     char *p;
5     p = (char*)PGROUNDUP((uint64)pa_start);
6     for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
7         kfree(p);
8 }
```

- 改后：

```

1 void
2 freerange(void *pa_start, void *pa_end)
3 {
4     char *p;
5     p = (char*)PGROUNDUP((uint64)pa_start);
6     for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
7         kfree(p);
8 }

```

- 哈哈没什么区别！

## 修改kalloc和kfree函数

- 修改kalloc和kfree函数，使它们操作当前CPU的自由列表：

```

1 // Free the page of physical memory pointed at by v,
2 // which normally should have been returned by a
3 // call to kalloc(). (The exception is when
4 // initializing the allocator; see kinit above.)
5 void
6 kfree(void *pa)
7 {
8     struct run *r;
9
10    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa
11    >= PHYSTOP)
12        panic("kfree");
13
14    // Fill with junk to catch dangling refs.
15    memset(pa, 1, PGSIZE);
16
17    r = (struct run*)pa;
18
19    // Ensure not interrupted while getting the CPU ID
20    push_off();
21    // Get the ID of the current CPU
22    int cpu = cpuid();
23    pop_off();
24
25    acquire(&kmem[cpu].lock);

```

```

25     r->next = kmem[cpu].freelist;
26     kmem[cpu].freelist = r;
27     release(&kmem[cpu].lock);
28 }
29
30 // Allocate one 4096-byte page of physical memory.
31 // Returns a pointer that the kernel can use.
32 // Returns 0 if the memory cannot be allocated.
33 void *
34 kalloc(void)
35 {
36     struct run *r;
37
38     push_off();
39     int cpu = cpuid();
40     pop_off();
41
42     acquire(&kmem[cpu].lock);
43     r = kmem[cpu].freelist;
44     if(r)
45         kmem[cpu].freelist = r->next;
46     else // add: steal page from other CPU
47     {
48         struct run* tmp;
49
50         // Loop over all other CPUs in NCPU range
51         for (int i = 0; i < NCPU; ++i)
52         {
53             if (i == cpu) // can't be itself
54                 continue;
55
56             // Acquire a lock on its freelist to prevent contention.
57             acquire(&kmem[i].lock);
58             tmp = kmem[i].freelist;
59             // no page to steal
60             if (tmp == 0) {
61                 release(&kmem[i].lock);
62                 continue;
63             } else {
64                 for (int j = 0; j < 1024; j++) {
65                     // steal 1024 pages
66                     if (tmp->next)

```

```

67         tmp = tmp->next;
68     else
69         break;
70 }
71
72     // change freelist
73     kmem[cpu].freelist = kmem[i].freelist;
74     kmem[i].freelist = tmp->next;
75     tmp->next = 0;
76
77     release(&kmem[i].lock);
78     break;
79 }
80 }
81 r = kmem[cpu].freelist;
82 if (r)
83     kmem[cpu].freelist = r->next;
84 } // end steal page from other CPU
85 release(&kmem[cpu].lock);
86
87 if(r)
88     memset((char*)r, 5, PGSIZE); // fill with junk
89 return (void*)r;
90 }

```

测试成功

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
hart 2 starting
init: starting sh
$ kalloc_test
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。

## 修改 `buf.h`

```
1 struct buf {
2     char used;
3     int valid;    // has data been read from disk?
4     int disk;    // does disk "own" buf?
5     uint dev;
6     uint blockno;
7     struct sleeplock lock;
8     uint refcnt;
9     struct buf *prev; // LRU cache list
10    struct buf *next;
11    uchar data[BSIZE];
12 };
```

## 实现哈希表和桶锁：

- 在 `kernel/bio.c` 中定义一个哈希表结构，包含哈希桶和对应的锁。
- 修改 `bget` 和 `brelse` 函数以使用哈希表，而不是全局缓存块列表。

```
1 struct bucket {
2     struct spinlock lock;
3     // Linked list of all buffers, through prev/next.
4     // Sorted by how recently the buffer was used.
5     // head.next is most recent, head.prev is least.
6     struct buf head;
7 };
8
9 struct {
10     struct buf buf[NBUF];
11     struct bucket bucket[NBUCKET];
12 } bcache;
13
14 static uint hash_v(uint key) {
15     return key % NBUCKET;
16 }
```

```
1 // Look through buffer cache for block on device dev.
2 // If not found, allocate a buffer.
3 // In either case, return locked buffer.
4 static struct buf*
5 bget(uint dev, uint blockno)
6 {
7     uint v = hash_v(blockno);
8     struct bucket* bucket = &bcache.bucket[v];
9     acquire(&bucket->lock);
10
11     // Is the block already cached?
12     for (struct buf *buf = bucket->head.next; buf != &bucket-
13         >head;
14         buf = buf->next) {
15         if(buf->dev == dev && buf->blockno == blockno){
16             buf->refcnt++;
17             release(&bucket->lock);
18             acquiresleep(&buf->lock);
19             return buf;
20         }
21     }
```



```

19     }
20 }
21
22 // Not cached.
23 // Recycle the least recently used (LRU) unused buffer.
24 for (int i = 0; i < NBUF; ++i) {
25     if (!bcache.buf[i].used &&
26         !__atomic_test_and_set(&bcache.buf[i].used,
27     __ATOMIC_ACQUIRE)) {
28         struct buf *buf = &bcache.buf[i];
29         buf->dev = dev;
30         buf->blockno = blockno;
31         buf->valid = 0;
32         buf->refcnt = 1;
33
34         buf->next = bucket->head.next;
35         buf->prev = &bucket->head;
36         bucket->head.next->prev = buf;
37         bucket->head.next = buf;
38         release(&bucket->lock);
39         acquiresleep(&buf->lock);
40         return buf;
41     }
42 }
43 panic("bget: no buffers");
44 }

```

```

1 // Release a locked buffer.
2 // Move to the head of the most-recently-used list.
3 void
4 brelse(struct buf *b)
5 {
6     if(!holdingsleep(&b->lock))
7         panic("brelse");
8
9     releasesleep(&b->lock);
10
11     uint v = hash_v(b->blockno);
12     struct bucket* bucket = &bcache.bucket[v];
13     acquire(&bucket->lock);
14

```

```

15     b->refcnt--;
16     if (b->refcnt == 0) {
17         // no one is waiting for it.
18         b->next->prev = b->prev;
19         b->prev->next = b->next;
20         __atomic_clear(&b->used, __ATOMIC_RELEASE);
21     }
22
23     release(&bucket->lock);
24 }

```

```

1 void
2 binit(void)
3 {
4     for (int i = 0; i < NBUF; ++i) {
5         initsleeplock(&bcache.buf[i].lock, "buffer");
6     }
7     for (int i = 0; i < NBUCKET; ++i) {
8         initbucket(&bcache.bucket[i]);
9     }
10 }

```

添加参数 `kernel/param.h`

```

1 #define NPROC          64 // maximum number of processes
2 #define NCPU           8 // maximum number of CPUs
3 #define NOFILE         16 // open files per process
4 #define NFILE          100 // open files per system
5 #define NINODE          50 // maximum number of active i-nodes
6 #define NDEV           10 // maximum major device number
7 #define ROOTDEV        1 // device number of file system root
   disk
8 #define MAXARG          32 // max exec arguments
9 #define MAXOPBLOCKS    10 // max # of blocks any FS op writes
10 #define LOGSIZE         (MAXOPBLOCKS*3) // max data blocks in on-
   disk log
11 #define NBUF            (MAXOPBLOCKS*3) // size of disk block
   cache
12
13 // #define FSSIZE          1000 // size of file system in blocks

```

```

14 #ifdef LAB_FS
15 #define FSSIZE      200000 // size of file system in blocks
16 #else
17 #ifdef LAB_LOCK
18 #define FSSIZE      10000 // size of file system in blocks
19 #else
20 #define FSSIZE      2000 // size of file system in blocks
21 #endif
22 #endif
23
24 #define MAXPATH      128 // maximum file path name
25 #define NBUCKET      13 // a prime number of buckets

```

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