# Lab8: Locks

2351882 王小萌

Tongji University,2025 Summer

代码仓库链接：https://github.com/wxmxmw/Tongji-University-xv6-labs-2021.git



## 1 Memory allocator

### 1.1 实验目的

为了减少多核系统中的锁竞争并提升性能，可以重构内存分配器的设计。具体做法是为每个 CPU 分配一个 独立的自由列表（free list），并且每个自由列表都有专属的锁。这样，不同 CPU 上的内存分配和释放 操作可以并行执行，显著减少锁争用。同时，当某个 CPU 的自由列表耗尽时，它应能够从其他 CPU 的自由 列表中获取部分内存。

### 1.2 实验步骤

1. 在 xv6 中运行 kalloctest，输出如下：

```
hart 2 starting
hart 1 starting
init: starting sh
$ kallocatest
exec kallocatest failed
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 98472 #acquire() 433070
lock: bcache: #test-and-set 0 #acquire() 1376
--- top 5 contended locks:
lock: proc: #test-and-set 163489 #acquire() 4215604
lock: proc: #test-and-set 162860 #acquire() 4215373
lock: proc: #test-and-set 160260 #acquire() 4215518
lock: proc: #test-and-set 151686 #acquire() 4215574
lock: proc: #test-and-set 143763 #acquire() 4215605
tot= 98472
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

可以看出，test1 测试未通过。并且针对 kmem 锁的争用情况：acquire() 函数被调用了 433070 次，自旋尝试获取锁的次数为 98472 次。锁 proc 也有大量的 #test-and-set 和 #acquire() 操作。

2. 观察到 NCPU 在 param.h 中声明，值为 8。

```
kernel > C param.h
#define NPROC        64  // maximum number of processes
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
```

3. 将 kalloc.c 中 kmem 修改成数组形式结构体，每个 CPU 分配一个 keme 锁：

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

4. 修改 kalloc.c 中的 kinit() 函数： 初始化每个 CPU 的空闲链表和锁

```c
void
kinit()
{
  int i;
  for (i = 0; i < NCPU; ++i) {
  //snprintf(kmem[i].lockname, 8, "kmem_%d", i); //the name of the lo
  initlock(&kmem[i].lock,"kmem");
  }
  //  initlock(&kmem.lock, "kmem");   // lab8-1
  freerange(end, (void*)PHYSTOP);
}
```

5. 修改 kalloc.c 中的 kfree() 函数，确保每个 CPU 的空闲链表正确维护。

```c
void
kfree(void *pa)
{
  struct run *r;
  int c;    // cpuid - lab8-1

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PH
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;
  // get the current core number - lab8-1
  push_off();
  c = cpuid();
  pop_off();
  // free the page to the current cpu's freelist - lab8-1
  acquire(&kmem[c].lock);
  r->next = kmem[c].freelist;
  kmem[c].freelist = r;
  release(&kmem[c].lock);
}
```

6. 修改 kalloc.c 中的 kalloc() 函数,分配内存页

```c
void *
kalloc(void)
{
  struct run *r;
  int c;
  push_off();
  c = cpuid();
  pop_off();

  acquire(&kmem.lock);
  r = kmem.freelist;

  if(r)
    kmem[c].freelist = r->next;
  release(&kmem[c].lock);
  if(!r&(r=steal(c))){
    acquire(&kmem[c].lock);
    kmem[c].freelist = r->next;
    release(&kmem[c].lock);
  }
  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

7. 编写 kalloc.c 中的 steal() 函数,从其他 CPU 偷取部分空闲内存页,并添加声明

```c
struct run *steal(int cpu_id){
  int i;
  int c =cpu_id;
  struct run *fast,*slow,*head;
  if(cpu_id!=cpuid()){
    panic("steal: cpu_id err");
  }
  for(int i=1;i<NCPU;++i){
    if(++c == NCPU){
      c=0;
    }
    acquire(&kmem[c].lock);
    if(kmem[c].freelist){
      slow = head = kmem[c].freelist;
      fast = slow->next;
      while(fast){
        fast=fast->next;
        if(fast){
          slow=slow->next;
          fast=fast->next;
        }
      }
      //the latter half change to current cpu freelist
      kmem[c].freelist =slow->next;
      release(&kmem[c].lock);
      //clear the first half
      slow->next=0;
      return head;
    }
    release(&kmem[c].lock);
  }
  return 0;
```

kernel > C kalloc.c

```c
#include "riscv.h"
#include "defs.h"
struct run *steal(int cpu_id);
void freerange(void *pa_start, void *pa_end);
```

8. 运行 xv6，并在命令行中输入 kalloctest ，可以看到 acquire 的循环迭代次数明显减少，表明锁的争用较之前有所减少。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 77629
lock: kmem: #test-and-set 0 #acquire() 182911
lock: kmem: #test-and-set 0 #acquire() 172481
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: bcache: #test-and-set 0 #acquire() 340
--- top 5 contended locks:
lock: proc: #test-and-set 296689 #acquire() 5668557
lock: proc: #test-and-set 289378 #acquire() 5668529
lock: proc: #test-and-set 235224 #acquire() 5668529
lock: proc: #test-and-set 227207 #acquire() 5668528
lock: proc: #test-and-set 224774 #acquire() 5668529
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

9. 运行 usertests 测试，确保所有的用户测试都通过

```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

10. 单项检测得分：

```
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-lock kalloctest
make: 'kernel/kernel' is up to date.
== Test running kalloctest == (37.6s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (5.1s)
```

## 1.3 实验中遇到的问题和解决办法

**问题 ：** 读取 CPU 的 ID 时没有禁用中断，使得程序出错。。

**解决办法 ：** 在实现过程中，需要确保每次获取锁和释放锁的顺序是一致的，避免出现循环等待的情况。除此之外，这很有可能是多个核心同时访问共享资源而没有正确的同步机制，也许会导致竞 争、数据损坏、不一致性或不可预测的结果。

## 1.4 实验心得

本次实验通过优化锁的使用，不仅提高了内存分配的性能，还加深了我对操作系统中并发控制的理解。

# 2 Buffer cache

## 2.1 实验目的

1. 优化 xv6 操作系统中的缓冲区缓存（buffer cache），减少多个进程之间对缓冲区缓存锁的争用，从而 提高系统的性能和并发能力。通过设计和实现一种更加高效的缓冲区管理机制，使得不同进程可以更有 效地使用和管理缓冲区，减少锁竞争和性能瓶颈
2. 原始的 xv6 中，对于缓存的读写是由单一的锁 bcache.lock 来保护的，导致了如果系统中有多个进程 在进行 IO 操作，则等待获取 bcache.lock 的开销就会较大。为了减少加锁的开销，可以将缓存分为几 个桶，为每个桶单独设置一个锁，这样如果两个进程访问的缓存块在不同的桶中，则可以同时获得两个 锁进行操作，而无需等待加锁。目标是将 bcachetest 中统计值 tot 降到规定值以下。。

## 2.2 实验步骤

1. 运行 bcachetest 测试程序，观察锁竞争情况和测试结果

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32872
lock: kmem: #test-and-set 0 #acquire() 98
lock: kmem: #test-and-set 0 #acquire() 71
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: kmem: #test-and-set 0 #acquire() 2
lock: bcache: #test-and-set 29754 #acquire() 72492
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 126050 #acquire() 1236
lock: proc: #test-and-set 126014 #acquire() 3075793
lock: proc: #test-and-set 109598 #acquire() 3096368
lock: proc: #test-and-set 107231 #acquire() 3096370
lock: proc: #test-and-set 105411 #acquire() 3096311
tot= 29754
test0: FAIL
start test1
test1 OK
$
```

2. 设计和实现缓冲区管理机制，修改 buf 和 bcache 结构体，为每个缓冲区添加时间戳字段。为 bcache 添加哈希表的锁。

kernel > C buf.h

```c
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; //hash list
  uchar data[BSIZE];
  uint timestamp//the buf last using time -lab8-2
};
```

```c
#define NBUCKET 13
#define HASH(blockno)(blockno & NBUCKET)

struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  int size; //the count of used buf -lab8-2
  struct buf buckets [NBUCKET];
  struct spinlock locks[NBUCKET];//buckets' locks
  struct spinlock hashlck; //the hash table's lock

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  //struct buf head;
} bcache;
```

3. 修改初始化函数。初始化哈希表的锁和缓冲区的时间戳字段

```c
void
binit(void)
{
  int i;
  struct buf *b;
  bcache.size=0;

  initlock(&bcache.lock, "bcache");
  initlock(&bcache.hashlock, "bcache_hash"); //init hashlock

  //init all buckets' lock
  for(i=0;i<NBUCKET;++i){
    initlock(&bcache.lock[i], "bcache_bucket");
  }

  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    initsleeplock(&b->lock, "buffer");

  }
}
```

4. 修改 brelse() 函数： 释放缓冲区时，更新时间戳字段，根据时间戳来选择缓冲区进行重用。

```c
void
brelse(struct buf *b)
{
  int idx;
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  idx =HASH(b->blockno)
  acquire(&bcache.lock[idx]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
   b->timestamp = ticks;
  }

  release(&bcache.lock);
}
```

5. 修改分配函数 bget() 函数，使用哈希表来寻找对应的缓冲区，

```c
bget(uint dev, uint blockno)
{
  struct buf *b;
  //lab8-2
  int idx=HASH(blockno);
  struct buf *pre,*minb = 0,*minpre;
  uint mintimestamp;
  int i;

  //loop up
  acquire(&bcache.locks[idx]);
  for(b = bcache.buckets[idx].next; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.locks[idx]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  // check if there is a buf not used
  acquire(&bcache.lock);
  if(bcache.size<NBUF){
    b=&bcache.buf[bcache.size++];
    release(&bcache.lock);
    b->dev = dev;
    b->blockno = blockno;
    b->valid = 0;
    b->refcnt = 1;
    b->next = bcache.buckets[idx].next;
    bcache.buckets[idx].next=b;
```

6. 运行 xv6，并运行 bcachetest

结果如下：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32934
lock: kmem: #test-and-set 0 #acquire() 49
lock: kmem: #test-and-set 0 #acquire() 54
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: bcache: #test-and-set 0 #acquire() 84
lock: bcache_hash: #test-and-set 0 #acquire() 54
lock: bcache_bucket: #test-and-set 0 #acquire() 2112
lock: bcache_bucket: #test-and-set 0 #acquire() 4130
lock: bcache_bucket: #test-and-set 0 #acquire() 2268
lock: bcache_bucket: #test-and-set 0 #acquire() 4266
lock: bcache_bucket: #test-and-set 0 #acquire() 4316
lock: bcache_bucket: #test-and-set 0 #acquire() 6315
lock: bcache_bucket: #test-and-set 0 #acquire() 6648
lock: bcache_bucket: #test-and-set 0 #acquire() 7931
lock: bcache_bucket: #test-and-set 0 #acquire() 6179
lock: bcache_bucket: #test-and-set 0 #acquire() 6187
lock: bcache_bucket: #test-and-set 0 #acquire() 6181
lock: bcache_bucket: #test-and-set 0 #acquire() 4121
lock: bcache_bucket: #test-and-set 0 #acquire() 4122
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 99556 #acquire() 1125
lock: proc: #test-and-set 39215 #acquire() 977840
lock: proc: #test-and-set 37494 #acquire() 977840
lock: proc: #test-and-set 36899 #acquire() 977839
lock: proc: #test-and-set 36860 #acquire() 998365
tot= 0
test0: OK
start test1
test1 OK
$
```

7. 单项测试得分：

```
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-lock bcachetest
make: 'kernel/kernel' is up to date.
== Test running bcachetest == (5.6s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
vale@Puppyyoo:~/xv6-labs-2021$
```

## 2.3 实验中遇到的问题和解决办法

**问题**：在修改缓冲区管理代码时，可能会遇到并发情况下的数据一致性问题，如资源分配冲突、 竞争条件。

**解决办法**：： 增加了锁，根据缓冲区的块号计算哈希索引 v，获取对应的分桶结构体指针 bucket 和分桶的自旋锁，以确保在操作缓冲区引用计数时不会被其他线程干扰。

## 2.4 实验心得

通过本次实验，我通过优化缓冲区管理方式，学会了如何在实际系统中使用更高效的数据结构和算法来提升性能。

# 3 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。

| | | | |
|---|---|---|---|
| time.txt | 2025/8/26 19:47 | 文本文档 | 1 KB |

2. 创建 answers-lock 文件将程序运行结果填入。
3. 在终端中执行 make grade,得到 lab8 总分：

```
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
== Test running kalloctest ==
$ make qemu-gdb
(37.3s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (4.8s)
== Test running bcachetest ==
$ make qemu-gdb
(4.7s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (71.8s)
== Test time ==
time: OK
Score: 70/70
vale@Puppyyoo:~/xv6-labs-2021$
```

4. 代码提交

```
vale@Puppyyoo:~/xv6-labs-2021$ git add .
vale@Puppyyoo:~/xv6-labs-2021$ git commit -m"lab8"
[lock 9f2aa23] lab8
 9 files changed, 384 insertions(+), 56 deletions(-)
 create mode 100644 answers-lock.txt
 create mode 100644 kernel/bio.c:Zone.Identifier
 create mode 100644 kernel/buf.h:Zone.Identifier
 create mode 100644 kernel/kalloc.c:Zone.Identifier
 create mode 100644 kernel/param.h:Zone.Identifier
 create mode 100644 time.txt
vale@Puppyyoo:~/xv6-labs-2021$ git status
On branch lock
Your branch is ahead of 'origin/lock' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```