



实验三：锁机制的应用

《操作系统》课程实验

● 目录



● 实验目的

- 了解锁的原理，掌握锁竞争对程序并行性的影响；
- 理解xv6的内存分配器memory allocator以及磁盘缓存buffer cache的工作原理；
- 掌握Lock在xv6中内存分配器/磁盘缓存中的作用；
- 学习利用减少锁竞争优化xv6系统。

● 实验任务

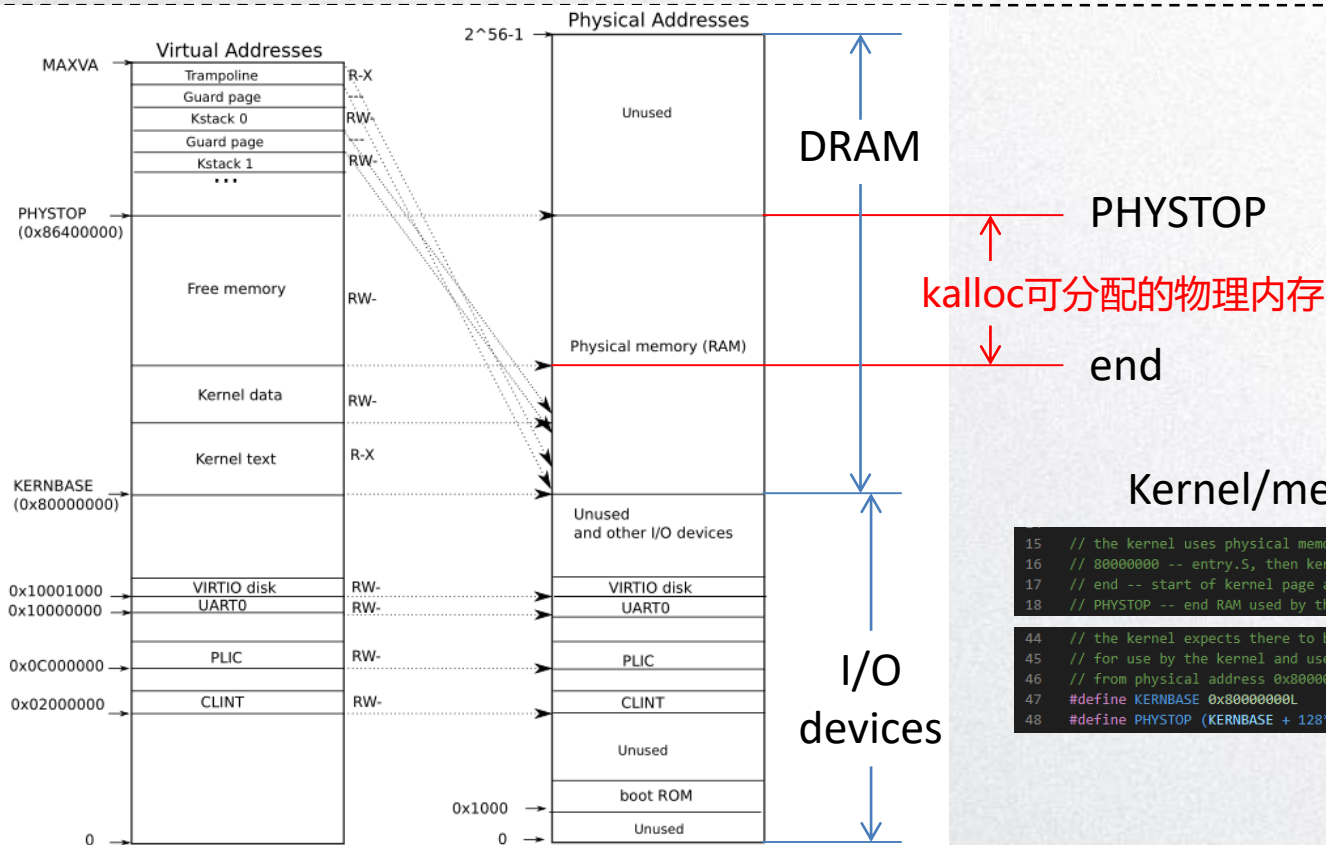
- 实验开始前，请切换到lock分支：
 - 同步上游仓库：<https://hitsz-lab.gitee.io/os-labs-2021/tools/#31>
- 多核计算机上并行性差的一个常见症状是高锁争用。为了减少锁争用、提高并行性，通常需要同时改变数据结构和锁的策略，本实验要求重新设计xv6以下两个模块：
 - 内存分配器Memory Allocator
 - 磁盘缓存Buffer Cache

● 实验任务一

重新设计内存分配器 (Memory Allocator)

- 修改空闲内存链表的管理机制，使每个CPU使用独立的链表；
- 支持在当前CPU的空闲链表 (freelist) 为空，其他链表不为空的情况下，可以从其他CPU的空闲链表中窃取 (steal) 空闲内存；
- 实验前后运行测评程序kallotest，查看是否减少锁争用，并检查它是否仍然可用于分配所有内存。

实验原理 | Xv6内核地址空间



Kernel/memlayout.h

```
15 // the kernel uses physical memory thus:
16 // 80000000 -- entry.S, then kernel text and data
17 // end -- start of kernel page allocation area
18 // PHYSTOP -- end RAM used by the kernel
```

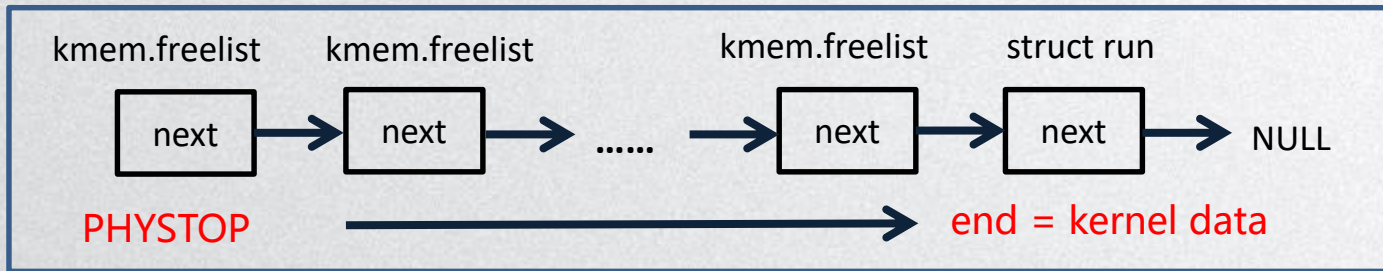
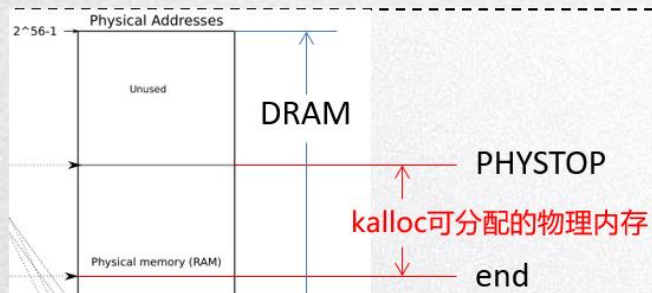
```
44 // the kernel expects there to be RAM
45 // for use by the kernel and user pages
46 // from physical address 0x80000000 to PHYSTOP.
47 #define KERNBASE 0x80000000L
48 #define PHYSTOP (KERNBASE + 128*1024*1024)
```

Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

来自《xv6 book》

● 实验原理 | 内存分配器——初始化

- 通过保存所有空闲页（1页4KB）来初始化链表。



空闲链表

● 实验原理 | 内存分配器

➤ XV6内存分配器的执行流程

```
26 void
27 kinit()
28 {
29     initlock(&kmem.lock, "kmem");
30     freerange(end, (void*)PHYSTOP);
31 }
```

entry.S
Xv6从entry.S开始启动，给当前CPU开一个4KB的栈空间

kernel/start.c: start()
跳转到C语言的入口处，设置中断，跳转到main函数

kernel/main.c: main()
consoleinit()首先设置console，就是printf打印位置

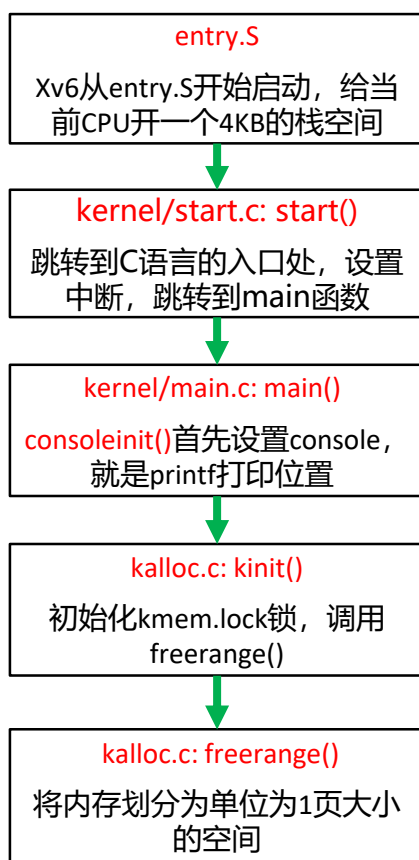

kalloc.c: kinit()
初始化kmem.lock锁，调用freerange()



● 实验原理 | 内存分配器

➤ XV6内存分配器的执行流程

```
33 void
34 freerange(void *pa_start, void *pa_end)
35 {
36     char *p;
37     p = (char*)PGROUNDUP((uint64)pa_start);
38     for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
39         kfree(p);
40 }
```



实验原理 | 内存分配器

➤ XV6内存分配器的执行流程

```
42 // Free the page of physical memory pointed at by v,
43 // which normally should have been returned by a
44 // call to kalloc(). (The exception is when
45 // initializing the allocator; see kinit above.)
46 void
47 kfree(void *pa)
48 {
49     struct run *r;
50
51     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
52         panic("kfree");
53
54     // Fill with junk to catch dangling refs.
55     memset(pa, 1, PGSIZE);
56
57     r = (struct run*)pa;
58
59     acquire(&kmem.lock);
60     r->next = kmem.freelist;
61     kmem.freelist = r;
62     release(&kmem.lock);
63 }
```

entry.S
Xv6从entry.S开始启动，给当前CPU开一个4KB的栈空间

kernel/start.c: start()
跳转到C语言的入口处，设置中断，跳转到main函数

kernel/main.c: main()
consoleinit()首先设置console，就是printf打印位置

kalloc.c: kinit()
初始化kmem.lock锁，调用freerange()

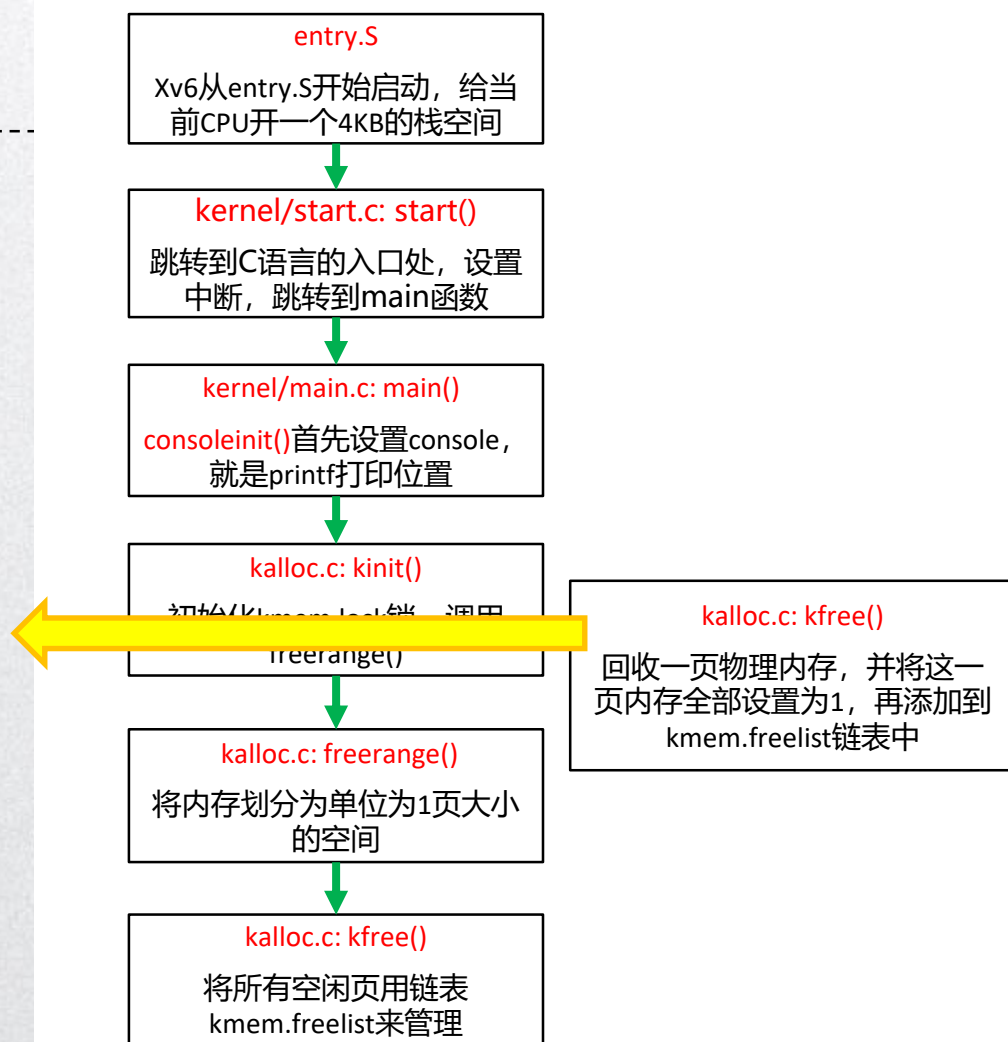
kalloc.c: freerange()
将内存划分为单位为1页大小的空间

kalloc.c: kfree()
将所有空闲页用链表kmem.freelist来管理

实验原理 | 内存分配器

➤ 释放内存

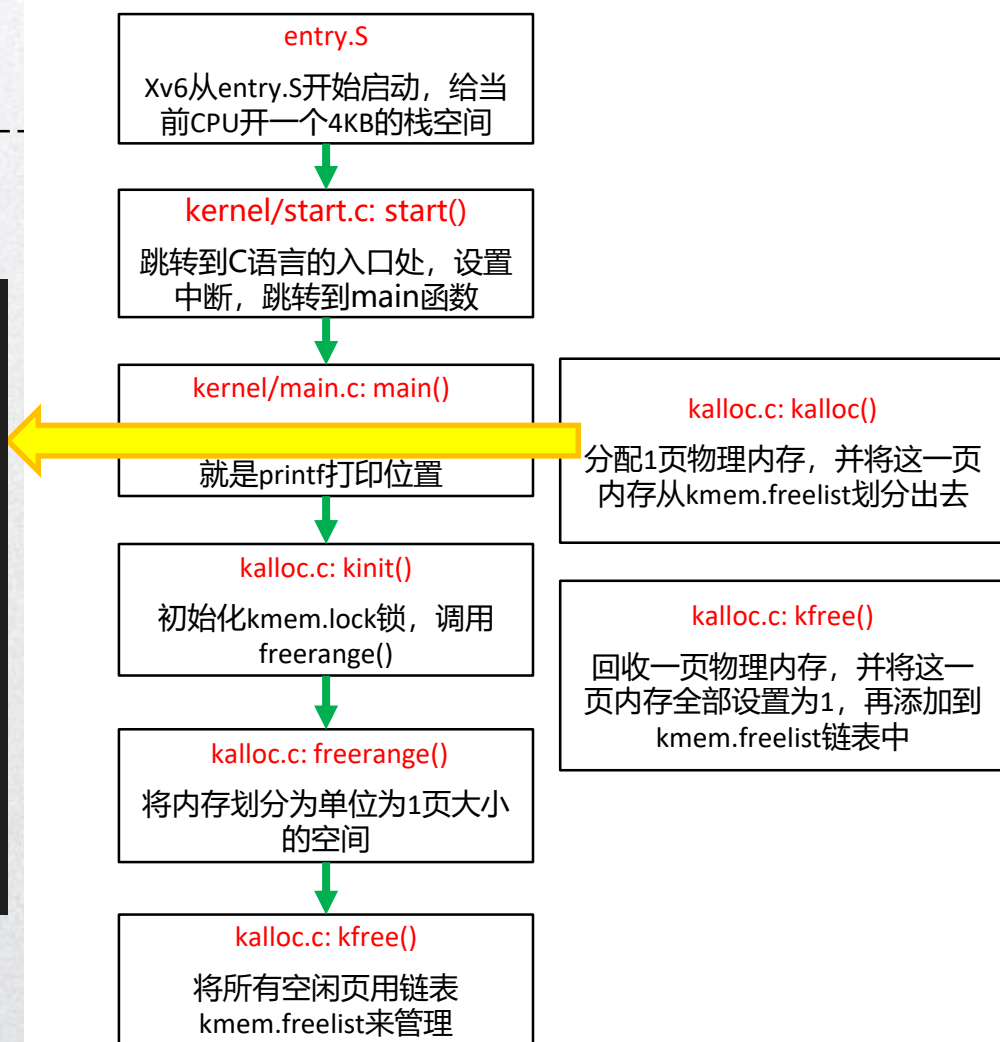
```
42 // Free the page of physical memory pointed at by v,  
43 // which normally should have been returned by a  
44 // call to kalloc(). (The exception is when  
45 // initializing the allocator; see kinit above.)  
46 void  
47 kfree(void *pa)  
48 {  
49     struct run *r;  
50  
51     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)  
52         panic("kfree");  
53  
54     // Fill with junk to catch dangling refs.  
55     memset(pa, 1, PGSIZE);  
56  
57     r = (struct run*)pa;  
58  
59     acquire(&kmem.lock);  
60     r->next = kmem.freelist;  
61     kmem.freelist = r;  
62     release(&kmem.lock);  
63 }
```



实验原理 | 内存分配器

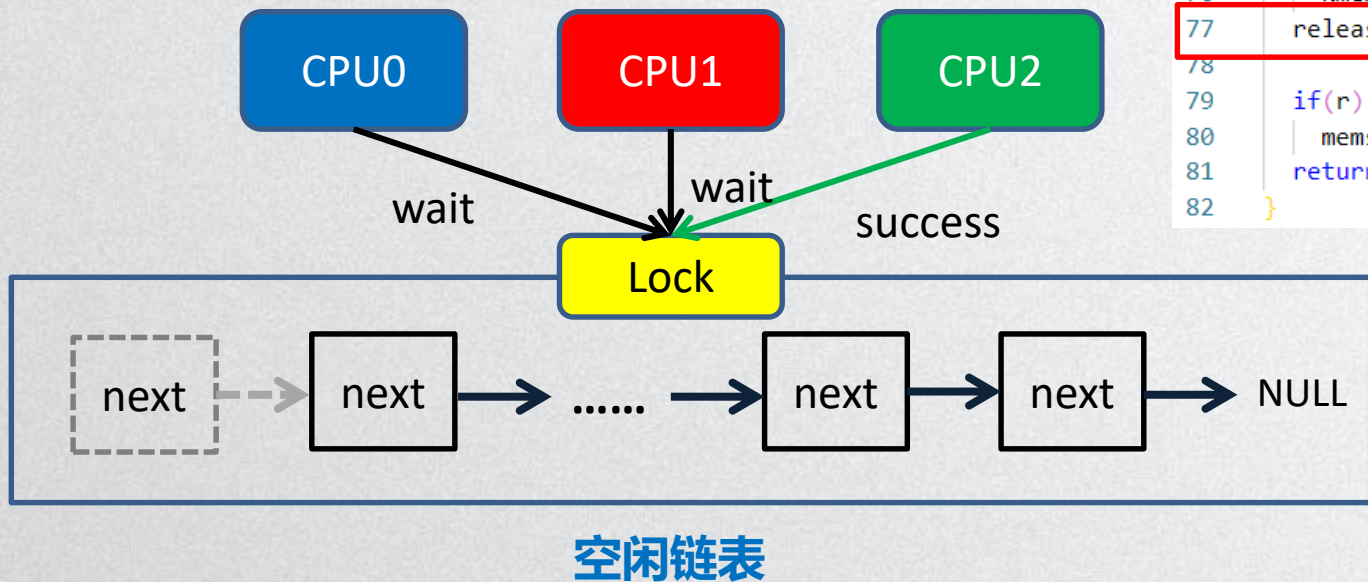
➤ 申请内存

```
65 // Allocate one 4096-byte page of physical memory.
66 // Returns a pointer that the kernel can use.
67 // Returns 0 if the memory cannot be allocated.
68 void *
69 kalloc(void)
70 {
71     struct run *r;
72
73     acquire(&kmem.lock);
74     r = kmem.freelist;
75     if(r)
76         kmem.freelist = r->next;
77     release(&kmem.lock);
78
79     if(r)
80         memset((char*)r, 5, PGSIZE); // fill with junk
81     return (void*)r;
82 }
```



● 实验原理 | 内存分配器—申请内存 kalloc()

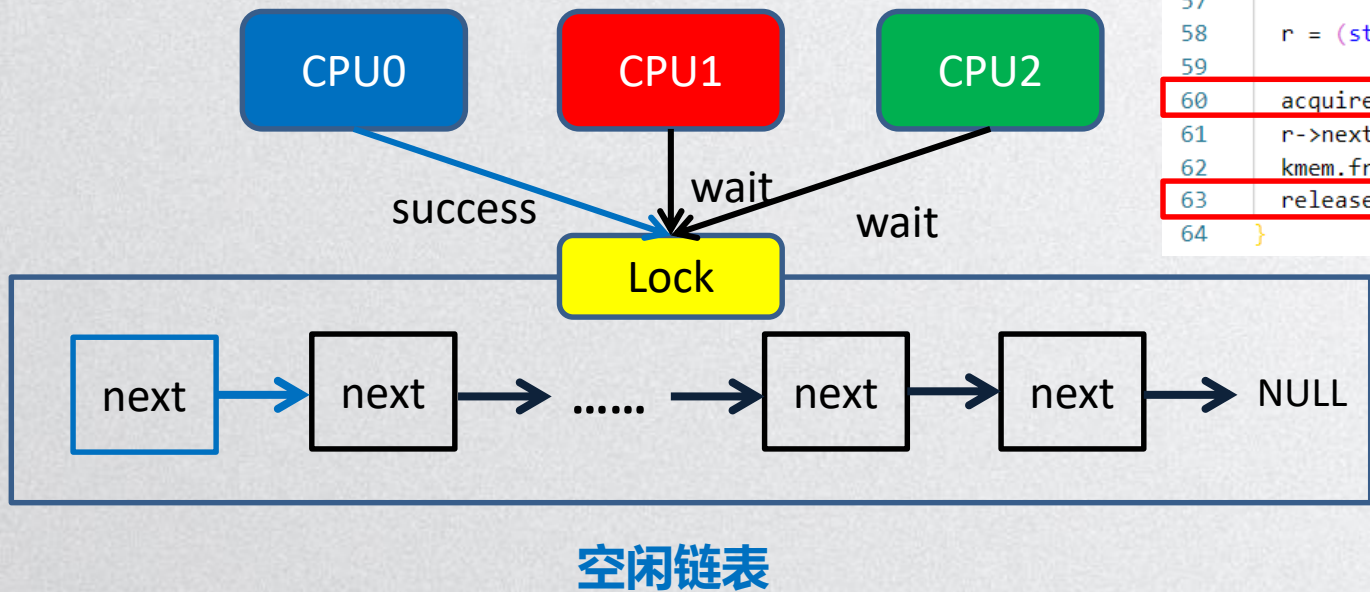
- Xv6支持运行在多核的系统上
- 当CPU2调用kalloc并获取到kmem.lock时，它可以分配1页物理内存



```
68 void *
69 kalloc(void)
70 {
71     struct run *r;
72
73     acquire(&kmem.lock);
74     r = kmem.freelist;
75     if(r)
76         kmem.freelist = r->next;
77     release(&kmem.lock);
78
79     if(r)
80         memset((char*)r, 5, PGSIZE);
81     return (void*)r;
82 }
```

● 实验原理 | 内存分配器—释放内存 kfree()

- 当CPU0调用kfree并获取到kmem.lock时，
它将1页物理内存加到空闲链表头

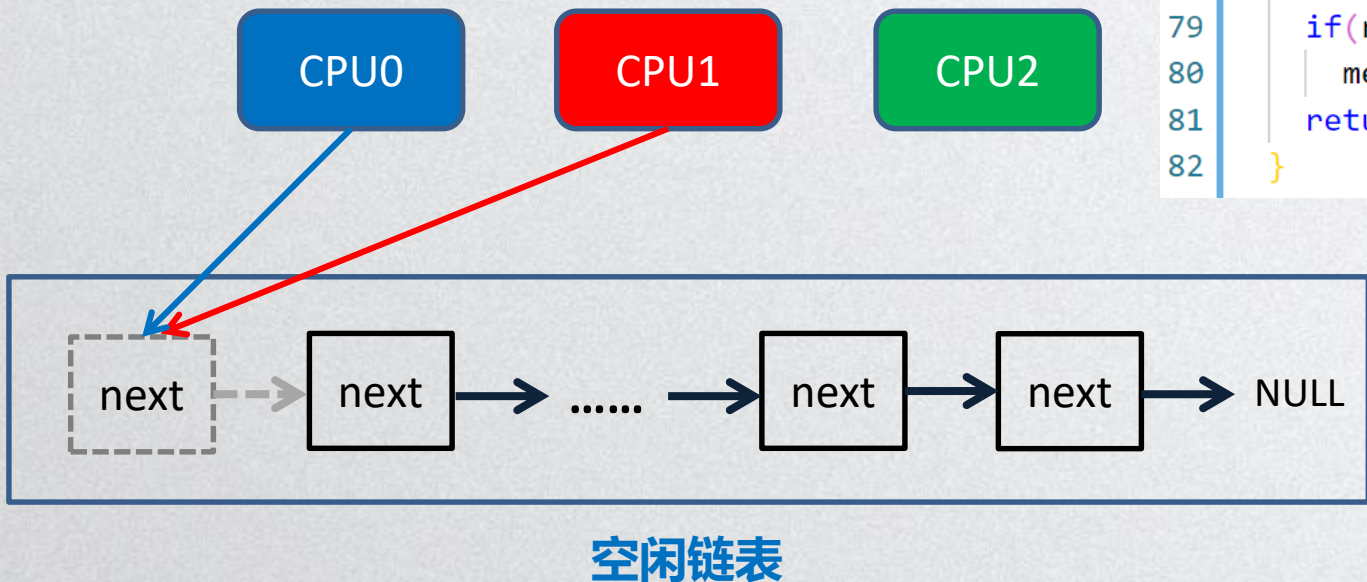


```
48  ✓ kfree(void *pa)
49  {
50      struct run *r;
51
52      if(((uint64)pa % PGSIZE) != 0
53          panic("kfree");
54
55      // Fill with junk to catch dan
56      memset(pa, 1, PGSIZE);
57
58      r = (struct run*)pa;
59
60      acquire(&kmem.lock);
61      r->next = kmem.freelist;
62      kmem.freelist = r;
63      release(&kmem.lock);
64  }
```

● 实验原理 | 为什么要上锁?

- 假设第73和77行的锁操作被注释掉了，此刻有CPU0和CPU1同时执行到第74行，两个CPU就会从freelist中拿出同一个内存块，这就会导致两个进程共用一块内存

```
69 kalloc(void)
70 {
71     struct run *r;
72
73     //acquire(&kmem.lock);
74     r = kmem.freelist;
75     if(r)
76         kmem.freelist = r->next;
77     //release(&kmem.lock);
78
79     if(r)
80         memset((char*)r, 5, PGSIZE);
81     return (void*)r;
82 }
```



● 实验原理 | 自旋锁的原理

➤ __sync_lock_test_and_set()

利用硬件提供的原子操作(amoswap)

当lk->lock=0 (当前锁空闲)

__sync_lock_test_and_set返回0,
往lk->lock写入1, 即获取到锁。

当lk->lock=1 (当前锁被锁)

__sync_lock_test_and_set返回1,
while自旋循环, 直至获取到锁。

➤ 两个重要的计数:

- #acquire: 获取锁的次数 (lk->n)
- #test-and-sets: 没有获取到锁的次数 (lk->nts)

```
34  acquire(struct spinlock *lk)
35  {
36      push_off(); // disable interrupts to avoid deadlock.
37      if(holding(lk))
38          panic("acquire");
39
40      __sync_fetch_and_add(&(lk->n), 1);
41
42  // On RISC-V, sync_lock_test_and_set turns into an atom.
43  //  a5 = 1
44  //  s1 = &lk->locked
45  //  amoswap.w.aq a5, a5, (s1)
46  while(__sync_lock_test_and_set(&lk->locked, 1) != 0) {
47      __sync_fetch_and_add(&lk->nts, 1);
48  }
```

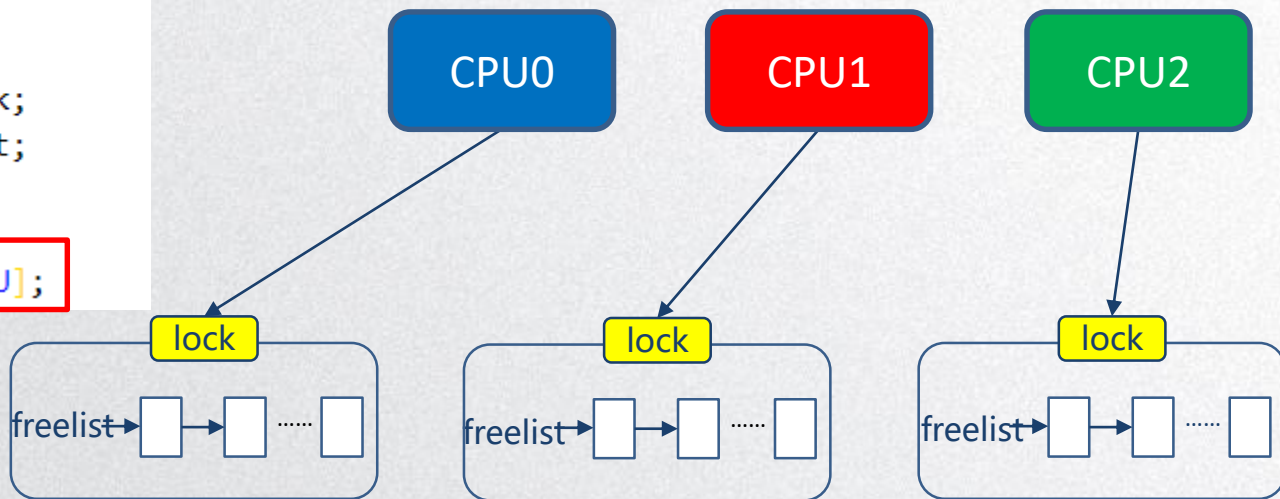

● 实验步骤 | 优化策略

```
264  ifndef CPUS
265  CPUS := 3
266  endif
```

给每个CPU构建一个空闲页面链表:

#define NCPU 8 // maximum number of CPUs (默认只有3个CPU (makefile))

```
102  struct run {
103      struct run *next;
104  };
105
106  struct kmem{
107      struct spinlock lock;
108      struct run *freelist;
109  };
110
111  struct kmem kmems[NCPU];
```



● 实验步骤 | 优化策略

➤ kalloc分配内存时，假设CPU1和CPU2找不到空闲页表怎么办？

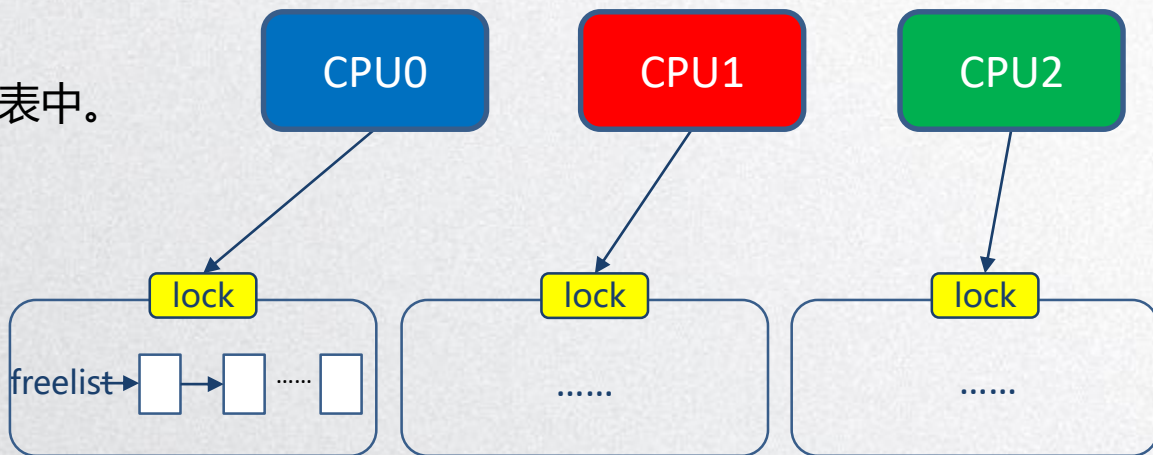
- 其他CPU的空闲链表中窃取 (steal) 空闲内存

窃取成功后，返回1页物理内存给调用者

➤ kfree释放内存时

获取当前CPU编号 (`cpuid()`)

将该页放入当前CPU的空闲页表中。



● 实验要求 | 优化效果

获取锁的次数 (lk->n)

- 请使用initlock()初始化锁，并要求锁名字以 **kmem** 开头；
- 运行kalloctest查看实现的代码是否减少了锁争用（**#test-and-set** 没有获取到此锁的次数小于10则为通过），并确认它可以分配所有的内存；
- 实现的代码不得影响xv6的基本功能，所以请确保**usertests**能够全部通过；
- 图中具体的数据会有所差别。

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 42843
lock: kmem: #fetch-and-add 0 #acquire() 198674
lock: kmem: #fetch-and-add 0 #acquire() 191534
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #fetch-and-add 361 #acquire() 117281
lock: virtio_disk: #fetch-and-add 5347 #acquire() 114
lock: proc: #fetch-and-add 56 #acquire() 117312
lock: proc: #fetch-and-add 68 #acquire() 117316
lock: proc: #fetch-and-add 97 #acquire() 117266
tot= 0
te
st
total free number of pages: 82199 (out of 82199)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests
ALL TESTS PASSED
$
```

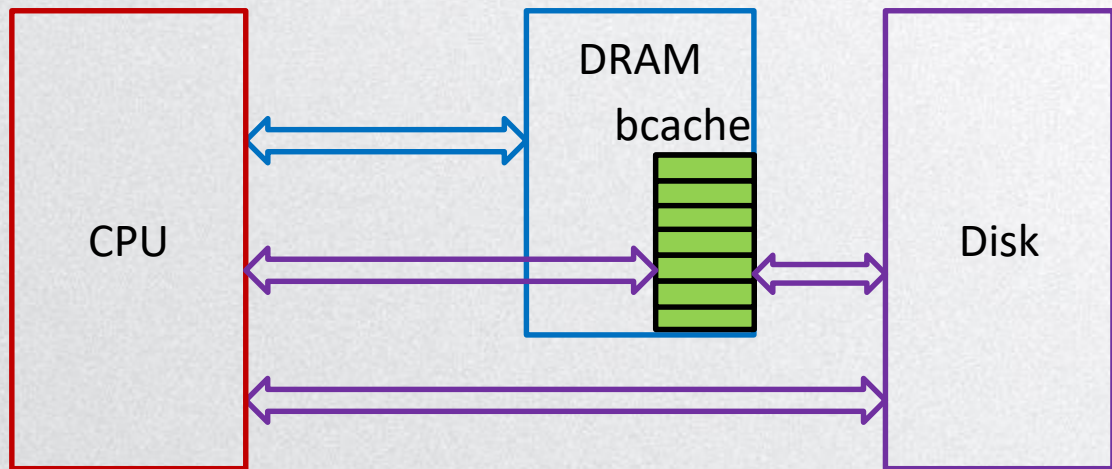

● 实验任务二

重新设计磁盘缓存 (Buffer cache)

- 修改磁盘缓存块列表的管理机制，建立缓存块和锁的**哈希映射**；
- 可用**多个锁管理磁盘缓存**，从而减少缓存块管理的锁争用；
- **实验前后**运行测评程序**bcachetest**，查看是否减少锁争用；
- 同样要求usertests中的用例全部通过。

● 实验原理 | Buffer cache简介

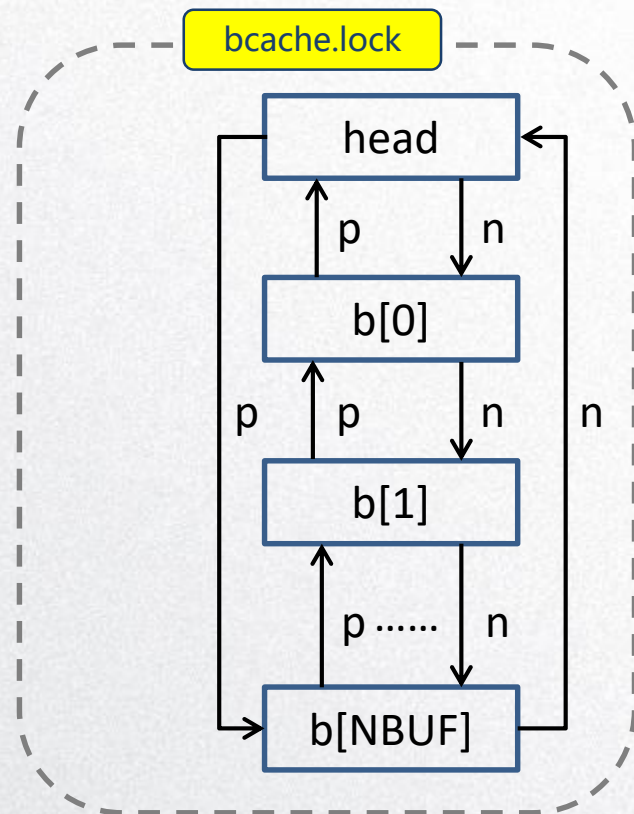
- 在xv6文件系统中，Buffer Cache（也称为bcache）为一个磁盘与内存文件系统交互的中间层，在kernel/bio.c中实现；



● 实验原理 | Buffer cache简介(bio.c)

- binit()构建了循环双向链表
- 数据结构bcache维护了一个由静态数组**struct buf buf[NBUF]**组成的双向链表，它以块为单位，每次读入或写出一个磁盘块，放到一个内存缓存块中 (bcache.buf)，同时自旋锁 bcache.lock用于用户互斥访问。

```
26 struct {
27     struct spinlock lock;
28     struct buf buf[NBUF];
29
30     // Linked list of all buffers, through prev/next.
31     // Sorted by how recently the buffer was used.
32     // head.next is most recent, head.prev is least.
33     struct buf head;
34 } bcache;
35
36 void
37 binit(void)
38 {
39     struct buf *b;
40
41     initlock(&bcache.lock, "bcache");
42
43     // Create linked list of buffers
44     bcache.head.prev = &bcache.head;
45     bcache.head.next = &bcache.head;
46     for(b = bcache.buf; b < bcache.buf+NBUF; b++){
47         b->next = bcache.head.next;
48         b->prev = &bcache.head;
49         initsleeplock(&b->lock, "buffer");
50         bcache.head.next->prev = b;
51         bcache.head.next = b;
52     }
53 }
```



● 实验原理 | Buffer cache简介(bio.c)

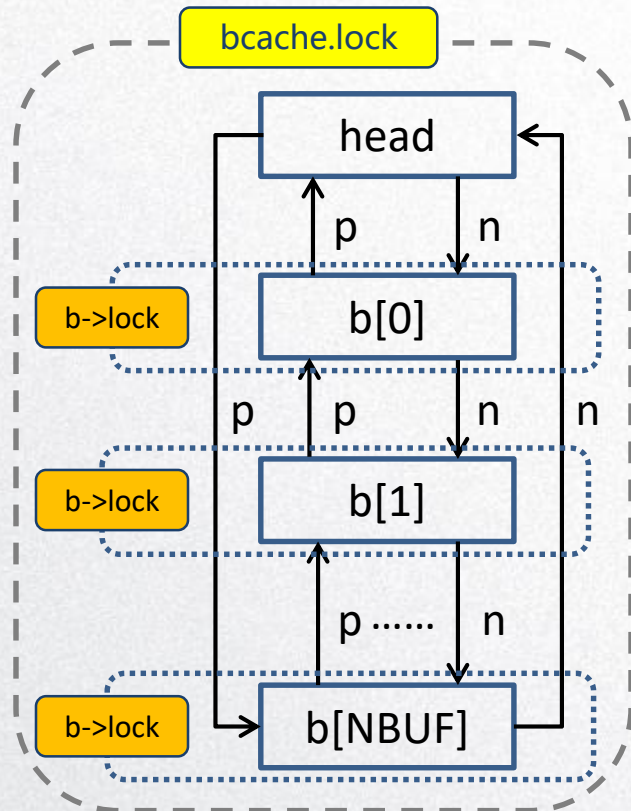
- kernel/buf.h
- 静态数组struct buf buf[NBUF]定义

```
struct buf {  
    int valid; // 该buffer包含对应磁盘块的数据  
    int disk; // 缓存区的内容是否已经被提交到了磁盘  
    uint dev; // 设备号  
    uint blockno; // 缓存的磁盘块号  
    struct sleeplock lock; // 睡眠锁  
    uint refcnt; // 该块被引用次数 (即被多少个进程拥有)  
    struct buf *prev; // LRU cache list  
    struct buf *next;  
    uchar data[BSIZE];  
};
```

数值字段

指针字段

data字段



● 实验原理 | Buffer cache简介(bio.c)

➤ bcache.lock 自旋锁

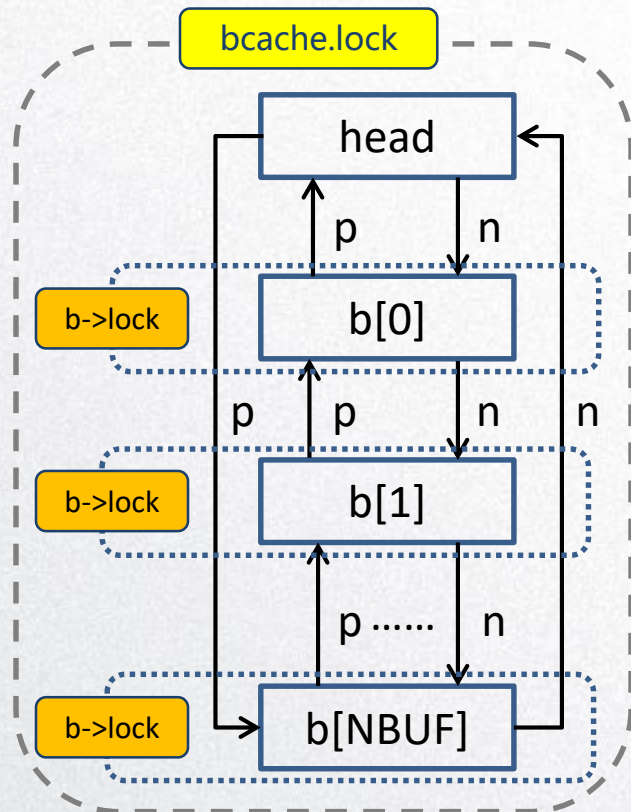
(用于表示bcache链表是否被锁住)

- bget()查找缓存块
- brelse()将缓存块挂载置head.prev
- bpin()引用缓存块
- bunpin()去除引用缓存块

➤ b->lock 睡眠锁

(用于表示缓存数据块buf是否被锁住)

- bget()获取缓存块
- bwrite()将缓存块写入磁盘
- brelse()释放缓存块



● 实验原理 | 自旋锁与睡眠锁

- 睡眠锁其实也是调用自旋锁acquire()保证原子性操作：
 - `acquiresleep()`: 查询b->lock是否被锁, 如果被锁了(==1), 就睡眠, **让出CPU**, 直到wakeup()唤醒后, 获取到锁, 并将b->lock置1
 - `releasesleep()`: 释放锁, 并调用wakeup()
 - `holdingsleep()`: 返回锁的状态 (1: 锁住或0: 未锁)

```
21 void
22 acquiresleep(struct sleeplock *lk)
23 {
24     acquire(&lk->lk);
25     while (lk->locked) {
26         sleep(lk, &lk->lk);
27     }
28     lk->locked = 1;
29     lk->pid = myproc()->pid;
30     release(&lk->lk);
31 }
32
33 void
34 releasesleep(struct sleeplock *lk)
35 {
36     acquire(&lk->lk);
37     lk->locked = 0;
38     lk->pid = 0;
39     wakeup(lk);
40     release(&lk->lk);
41 }
42
43 int
44 holdingsleep(struct sleeplock *lk)
45 {
46     int r;
47
48     acquire(&lk->lk);
49     r = lk->locked && (lk->pid == myproc()->pid);
50     release(&lk->lk);
51     return r;
52 }
```

● 实验原理 | 自旋锁与睡眠锁

自旋锁 Spinlock

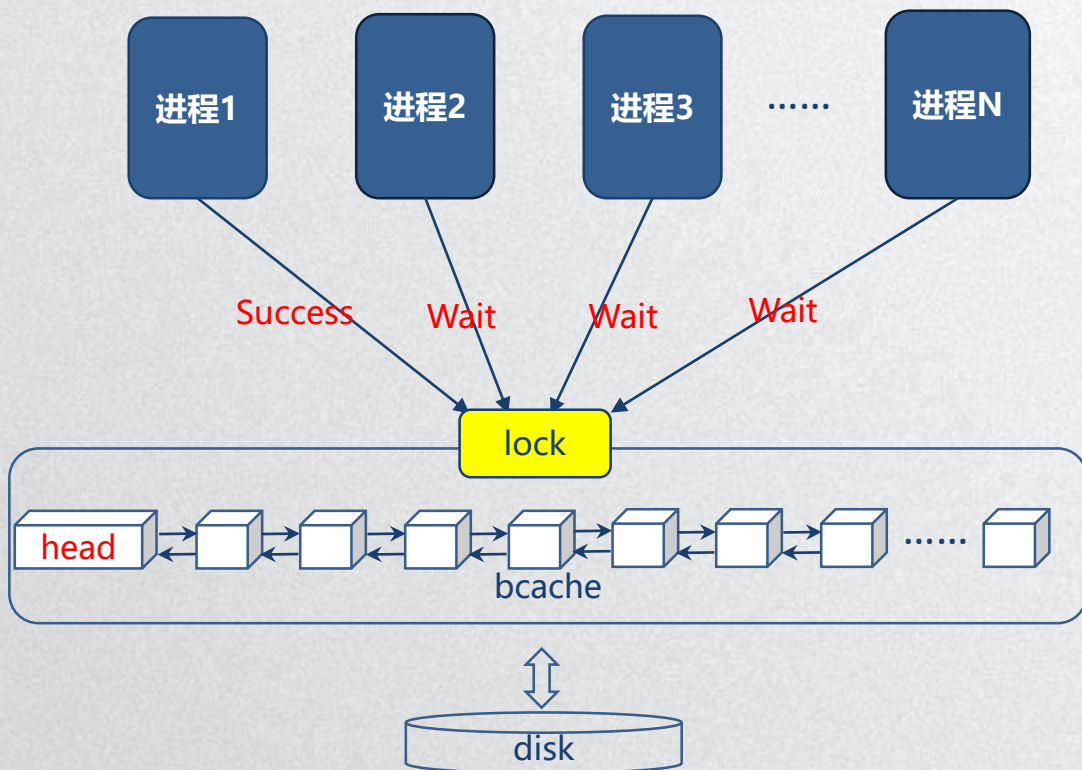
- 在短时间进行轻量级加锁；
- 获取过程一直进行忙循环—旋转—等待锁重新可用（占用CPU时间长）；
- xv6要求在持有spinlock期间，中断不允许发生。

睡眠锁 Sleep-lock

- 适合长时间持有；
- 获取锁期间可以让出CPU；
- 持有Sleep-lock期间允许中断发生，但不允许在中断程序中使用。

请参阅《[xv6 book](#)》 6.7节

● 实验原理 | bcache存在问题



```
$ bcachetest
start test0
test0 results:
=== lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33026
lock: kmem: #test-and-set 0 #acquire() 50
lock: kmem: #test-and-set 0 #acquire() 73
lock: bcache: #test-and-set 186438 #acquire() 65650
=== top 5 contended locks:
lock: bcache: #test-and-set 186438 #acquire() 65650
lock: proc: #test-and-set 52912 #acquire() 66921
lock: proc: #test-and-set 14693 #acquire() 66568
lock: proc: #test-and-set 13379 #acquire() 66568
lock: proc: #test-and-set 12117 #acquire() 66568
test0: FAIL
start test1
test1 OK
```

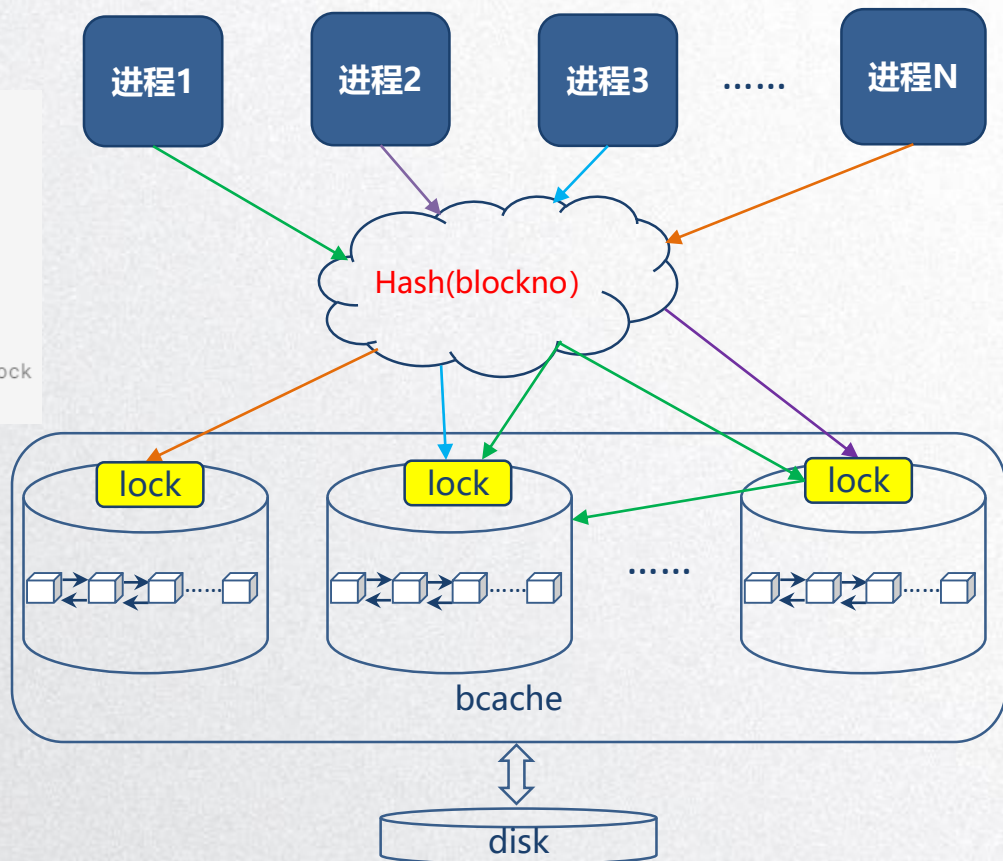
● 实验原理 | 优化策略（一）构建哈希表

以blockno为key构建哈希表和哈希桶：

```
#define NBUCKETS 13
```

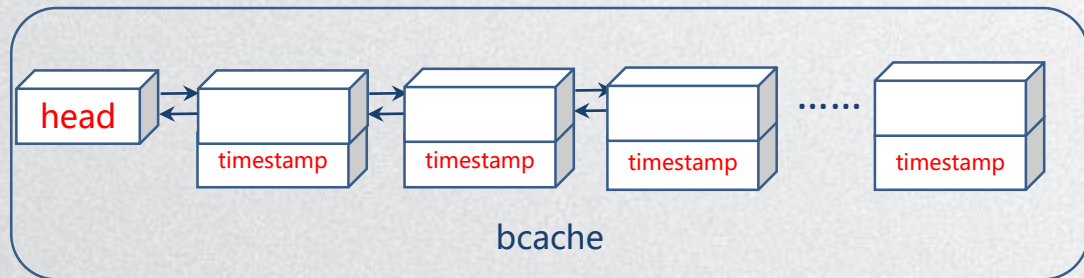
```
struct {  
    struct spinlock lock[NBUCKETS];  
    struct buf buf[NBUF];  
  
    // Linked list of all buffers, through prev/next.  
    // head.next is most recently used.  
    // struct buf head;  
    struct buf hashbucket[NBUCKETS]; //每个哈希队列一个linked list及一个lock  
} bcache;
```

当bget()查找数据块未命中时，bget可从其他哈希桶选择一个未被使用的缓存块，移入到当前的哈希桶链表中使用



● 实验原理 | 优化策略（二）增加时间戳

- 使用时间戳作为判断缓存块最后一次被访问的顺序的依据。
- 此项改动可使brelse不再需要锁上bcache lock。
 - brelse不需要将缓存块添加到头部，只需改写时间戳即可。
- bget也可通过时间戳得知最后一次被访问时间最早的空闲缓存块。
- 时间戳可通过kernel/trap.c中的ticks获得。



● 实验要求 | 优化效果

- 请将本实验涉及的锁以“**bcache**”开头来命名，可以调用initlock()来初始化bcache锁；
- 理想状态下，bcachetest中数据块缓存相关的所有锁test-and-set的总和应该为0，但本实验中**总和不超过500**即可；
- 每个数据块**最多只能维护一份缓存**；
- 同样要求**usertests**中的用例全部通过；
- 图中具体的数据会有所差别。

```
$ bcachetest
start test0
test0 results:
=== lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 32968
lock: kmem: #test-and-set 0 #acquire() 53
lock: kmem: #test-and-set 0 #acquire() 53
lock: bcache: #test-and-set 0 #acquire() 598
lock: bcache.bucket: #test-and-set 0 #acquire() 4139
lock: bcache.bucket: #test-and-set 0 #acquire() 4131
lock: bcache.bucket: #test-and-set 0 #acquire() 2360
lock: bcache.bucket: #test-and-set 0 #acquire() 4307
lock: bcache.bucket: #test-and-set 0 #acquire() 2419
lock: bcache.bucket: #test-and-set 0 #acquire() 4420
lock: bcache.bucket: #test-and-set 0 #acquire() 4934
lock: bcache.bucket: #test-and-set 18 #acquire() 8692
lock: bcache.bucket: #test-and-set 0 #acquire() 6457
lock: bcache.bucket: #test-and-set 0 #acquire() 6197
lock: bcache.bucket: #test-and-set 0 #acquire() 6191
lock: bcache.bucket: #test-and-set 0 #acquire() 6210
lock: bcache.bucket: #test-and-set 0 #acquire() 6198
=== top 5 contended locks:
lock: proc: #test-and-set 1113301 #acquire() 68753
lock: proc: #test-and-set 845107 #acquire() 68685
lock: proc: #test-and-set 822143 #acquire() 68685
lock: proc: #test-and-set 808826 #acquire() 68685
lock: proc: #test-and-set 664514 #acquire() 68727
test0: OK
start test1
test1 OK
$ usertests
...
ALL TESTS PASSED
$
```

● 实验要点 | 注意事项

- 逻辑设计上，有的流程可能需要同时持有多个相同或者不同类型的锁，请合理规划获取锁的顺序，避免死锁问题；
- 代码实现时，可调用函数cpuid()获取进程运行的核ID，但该函数只有在中断关闭时才能安全使用，请在使用前后使用push_off()和pop_off()关闭和打开中断；
- 请认真阅读实验指导书 <http://hitsz-lab.gitee.io/os-labs-2021/lab3/part1/>

● 实验要点 | 作业提交

- 请务必提前自测`grade-lab-lock`;
- 截止下一次实验课提交, 请参考实验二的[提交方式](#);
- **注意: 我们下一次实验课 (第10周) 有现场验收, 要求如下:**
 - **请同学们在下一次实验课准备好前三次实验 (util、syscall、lock) 的代码, 助教会一对一问实验内容及代码相关的问题, 请大家认真对待。**

The background is a light gray with a subtle grid pattern. It features several circles of different sizes and colors (dark blue and white) scattered across the frame. A large white circle with a dark blue border is positioned in the upper left, containing the word 'THANKS' in red. Other circles of various sizes are placed around it, some solid and some with borders.

THANKS

同学们，
请开始实验吧！