Locks

Memory allocator

实验目的

实验目的：

> 这个任务要求给物理内存分配程序重新设计锁，使得等待锁时的阻塞尽量少。可以按CPU的数量将空闲内存分组，分配内存的时候优先从当前所用CPU所管理的空闲内存中分配，如果没有则从其他CPU的空闲内存中获取，这样就可以把原来的锁拆开，每个CPU各自处理自己的空闲内存时只要锁上自己的锁就行了

实验提示：

> **提示：**
>
> - 您可以使用**kernel/param.h**中的常量`NCPU`
> - 让`freerange`将所有可用内存分配给运行`freerange`的CPU。
> - 函数`cpuid`返回当前的核心编号，但只有在中断关闭时调用它并使用其结果才是安全的。您应该使用`push_off()`和`pop_off()`来关闭和打开中断。
> - 看看**kernel/sprintf.c**中的`snprintf`函数，了解字符串如何进行格式化。尽管可以将所有锁命名为“`kmem`”。

大致解题思路：

> - 以 *NCPU* 为基准，给每一个CPU创建一个内存池
> - 使用锁管理每一个内存池
> - 当某个内存池为空时，从其他内存池偷取内存块

实验步骤

1. 首先根据提示1定义利用常量定义NCPU个`kmem`结构体，并在`kinit`函数中对锁进行初始化。

```
//将kmem定义为一个数组
struct
{
  struct spinlock lock;
  struct run *freelist;
} kmem[NCPU];
```

2. 然后修改`kinit`，为所有锁初始化以`kmem`开头的名称，该函数只会被一个CPU调用，`freerange`调用`kfree`将所有空闲内存挂在该CPU的空闲列表上

```
void kinit()
{
  char lockname[8];
  //初始化所有锁
  for (int i = 0; i < NCPU; i++)
  {
    snprintf(lockname, sizeof(lockname), "kmem_%d", i);
    initlock(&kmem[i].lock, lockname);
  }
  //这里不用修改，默认将内存分配给运行这一函数的cpu
  freerange(end, (void *)PHYSTOP);
}
```

3. 修改`kfree`，使用`cpuid()`返回的结果时必须关闭中断

```
void kfree(void *pa)
{
  struct run *r;

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

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

  r = (struct run *)pa;

  push_off(); // 关中断
  int id = cpuid();
  //获取锁以保证内存池的使用安全
  acquire(&kmem[id].lock);
  r->next = kmem[id].freelist;
  kmem[id].freelist = r;
  release(&kmem[id].lock);
  pop_off(); //开中断
}
```

4. 最后修改`kalloc`，使得在当前CPU的空闲列表没有可分配内存时窃取其他内存的

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

  push_off(); // 关中断
  int id = cpuid();
  acquire(&kmem[id].lock);
  r = kmem[id].freelist;
  //先从自己的内存池中寻找可用的内存块
  if (r)
  {
    kmem[id].freelist = r->next;
  }
  else //没有空闲块则从其他内存池中寻找
  {
    int antid; // another id
    // 遍历所有CPU的空闲列表
    for (antid = 0; antid < NCPU; ++antid)
    {
      if (antid == id)
        continue;
      //寻找前记得上锁
      acquire(&kmem[antid].lock);
      r = kmem[antid].freelist;
      if (r)
      {
        kmem[antid].freelist = r->next;
        release(&kmem[antid].lock);
        break;
      }
      release(&kmem[antid].lock);
    }
  }
  release(&kmem[id].lock);
  pop_off(); //开中断

  if (r)
    memset((char *)r, 5, PGSIZE); // fill with junk
  return (void *)r;
}
```

实验结果

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem_cpu_0: #fetch-and-add 0 #acquire() 162653
lock: kmem_cpu_1: #fetch-and-add 0 #acquire() 106892
lock: kmem_cpu_2: #fetch-and-add 0 #acquire() 163470
lock: bcache_eviction: #fetch-and-add 0 #acquire() 4
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 35
lock: bcache_eviction: #fetch-and-add 0 #acquire() 3
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 112
lock: bcache_eviction: #fetch-and-add 0 #acquire() 2
```

```
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 83
lock: bcache_eviction: #fetch-and-add 0 #acquire() 1
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 1046
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 21
lock: bcache_eviction: #fetch-and-add 0 #acquire() 1
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 24
lock: bcache_eviction: #fetch-and-add 0 #acquire() 2
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 43
lock: bcache_eviction: #fetch-and-add 0 #acquire() 4
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 51
lock: bcache_eviction: #fetch-and-add 0 #acquire() 4
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 39
--- top 5 contended locks:
lock: proc: #fetch-and-add 513115 #acquire() 157365
lock: proc: #fetch-and-add 447346 #acquire() 157365
lock: proc: #fetch-and-add 285332 #acquire() 157365
lock: proc: #fetch-and-add 252229 #acquire() 157365
lock: proc: #fetch-and-add 250138 #acquire() 157365
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests
usertests starting
test manywrites: 3
OK
test execout: 3

OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3247
            sepc=0x00000000000056a4 stval=0x00000000000056a4
usertrap(): unexpected scause 0x000000000000000c pid=3248
            sepc=0x00000000000056a4 stval=0x00000000000056a4
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test dirtest: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000215c stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000215c stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000215c stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000215c stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000215c stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000215c stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000215c stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6235
            sepc=0x000000000000215c stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6236
            sepc=0x000000000000215c stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6237
            sepc=0x000000000000215c stval=0x000000008006ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6238
            sepc=0x000000000000215c stval=0x000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6239
            sepc=0x000000000000215c stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6240
            sepc=0x000000000000215c stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x000000000000215c stval=0x000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6242
            sepc=0x000000000000215c stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6243
            sepc=0x000000000000215c stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6244
            sepc=0x000000000000215c stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6245
            sepc=0x000000000000215c stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x000000000000215c stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x000000000000215c stval=0x00000000800e7ef0
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6248
            sepc=0x000000000000215c stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6249
            sepc=0x000000000000215c stval=0x0000000080100590
usertrap(): unexpected scause 0x000000000000000d pid=6250
            sepc=0x000000000000215c stval=0x000000008010c8e0
usertrap(): unexpected scause 0x000000000000000d pid=6251
            sepc=0x000000000000215c stval=0x0000000080118c30
usertrap(): unexpected scause 0x000000000000000d pid=6252
            sepc=0x000000000000215c stval=0x0000000080124f80
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x000000000000215c stval=0x00000000801312d0
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000215c stval=0x000000008013d620
usertrap(): unexpected scause 0x000000000000000d pid=6255
            sepc=0x000000000000215c stval=0x0000000080149970
usertrap(): unexpected scause 0x000000000000000d pid=6256
            sepc=0x000000000000215c stval=0x0000000080155cc0
usertrap(): unexpected scause 0x000000000000000d pid=6257
            sepc=0x000000000000215c stval=0x0000000080162010
usertrap(): unexpected scause 0x000000000000000d pid=6258
            sepc=0x000000000000215c stval=0x000000008016e360
usertrap(): unexpected scause 0x000000000000000d pid=6259
            sepc=0x000000000000215c stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6260
            sepc=0x000000000000215c stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6261
            sepc=0x000000000000215c stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6262
            sepc=0x000000000000215c stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6263
            sepc=0x000000000000215c stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6264
            sepc=0x000000000000215c stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6265
            sepc=0x000000000000215c stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6266
            sepc=0x000000000000215c stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6267
            sepc=0x000000000000215c stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6279
            sepc=0x00000000000041fc stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6283
            sepc=0x00000000000022cc stval=0x000000000000fb90
OK
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 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
```

Buffer cache

实验目的

实验内容：

> 这个任务要求给硬盘缓存分配程序重新设计锁，使得等待锁时的阻塞尽量少。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，也就是说缓冲区的操作是完全串行进行的。为了提高并行性能，我们可以用哈希表来代替链表，这样每次获取和释放的时候，都只需要对哈希表的一个桶进行加锁，桶之间的操作就可以并行进行。只有当需要对缓冲区进行驱逐替换时，才需要对整个哈希表加锁来查找要替换的块。这样节能大幅减少了阻塞。

实验提示：

> - 请阅读xv6手册中对块缓存的描述（第8.1-8.3节）。
> - 可以使用固定数量的散列桶，而不动态调整哈希表的大小。使用素数个存储桶（例如13）来降低散列冲突的可能性。
> - 在哈希表中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的。
> - 删除保存了所有缓冲区的列表（`bcache.head`等），改为标记上次使用时间的时间戳缓冲区（即使用**kernel/trap.c**中的`ticks`）。通过此更改，`brelse`不需要获取bcache锁，并且`bget`可以根据时间戳选择最近使用最少的块。
> - 可以在`bget`中串行化回收（即`bget`中的一部分：当缓存中的查找未命中时，它选择要复用的缓冲区）。
> - 在某些情况下，您的解决方案可能需要持有两个锁；例如，在回收过程中，您可能需要持有bcache锁和每个bucket（散列桶）一个锁。确保避免死锁。
> - 替换块时，您可能会将`struct buf`从一个bucket移动到另一个bucket，因为新块散列到不同的bucket。您可能会遇到一个棘手的情况：新块可能会散列到与旧块相同的bucket中。在这种情况下，请确保避免死锁。
> - 一些调试技巧：实现bucket锁，但将全局`bcache.lock`的`acquire`/`release`保留在`bget`的开头/结尾，以串行化代码。一旦您确定它在没有竞争条件的情况下是正确的，请移除全局锁并处理并发性问题。您还可以运行`make CPUS=1 qemu`以使用一个内核进行测试。

大致解题思路

> - 创建一个大小为13的哈希表，表中元素为cache池。每个池拥有一个锁，保证操作的原子性。
> - cache块根据 *blockno* 哈希到不同池中。
> - 读取新的磁盘块时，先到对应的cache池中寻找是否已缓存；若无，则先在自己的池中寻找空闲块；若无，则到其他池子寻找空闲块，找到后将该块转移到本池。
> - 为cache块打上时间戳，每次使用时更新，寻找空闲块时，优先选择最久未被使用的空闲块。

> - 块被释放时并不做另外的操作，留在原地。

实验内容

1. 为了提高并行性能，我们可以用哈希表来代替链表，这样每次获取和释放的时候，都只需要对哈希表的一个桶进行加锁，桶之间的操作就可以并行进行。只有当需要对缓冲区进行驱逐替换时，才需要对整个哈希表加锁来查找要替换的块。但是使用哈希表就不能使用链表来维护`LRU`信息，因此我们需要在`buf`结构体中添加`timestamp`域来记录释放的事件。

```
/*kernel/buf.h*/
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;
  uchar data[BSIZE];
  //增加时间戳
  uint timestamp;
};
```

2. 然后我们根据提示1定义哈希桶结构，并在`bcache`中删除全局缓冲区链表，改为使用素数个散列桶.

```
//bio.c
/*
  创建哈希表，锁，和hash函数
*/
#define NBUCKET 13
#define HASH(id) (id % NBUCKET)

struct hashbuf
{
  struct buf head;      // 头节点
  struct spinlock lock; // 锁
};

struct
{
  struct buf buf[NBUF];
  struct hashbuf buckets[NBUCKET]; // 散列桶
} bcache;

```

3. 然后在`binit`中
   1. 初始化散列桶的锁
   2. 将所有散列桶的`head->prev`、`head->next`都指向自身表示为空
   3. 将所有的缓冲区挂载到`bucket[0]`桶上

```c
//跟上一实验一样的做法，初始化锁并将所有块挂在第一个池子里
void binit(void)
{
  struct buf *b;
  char lockname[16];

  for (int i = 0; i < NBUCKET; ++i)
  {
    // 初始化散列桶的自旋锁
    snprintf(lockname, sizeof(lockname), "bcache_%d", i);
    initlock(&bcache.buckets[i].lock, lockname);

    // 初始化散列桶的头节点
    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }

  // Create linked list of buffers
  for (b = bcache.buf; b < bcache.buf + NBUF; b++)
  {
    // 利用头插法初始化缓冲区列表,全部放到散列桶0上
    b->next = bcache.buckets[0].head.next;
    b->prev = &bcache.buckets[0].head;
    initsleeplock(&b->lock, "buffer");
    bcache.buckets[0].head.next->prev = b;
    bcache.buckets[0].head.next = b;
  }
}
```

4. 然后我们需要更改`brelse`，使其不再获取全局锁

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

  int bid = HASH(b->blockno);

  releasesleep(&b->lock);

  acquire(&bcache.buckets[bid].lock);
  b->refcnt--;

  // 更新时间戳
  // 由于LRU改为使用时间戳判定，不再需要头插法
  acquire(&tickslock);
  b->timestamp = ticks;
  release(&tickslock);

  release(&bcache.buckets[bid].lock);
}
```

5. 然后我们根据提示5，更改`bget`，使得其进行串行化回收。首先我们应该在对应的桶当中查找当前块是否被缓存，如果被缓存就直接返回。当没有找到指定的缓冲区时进行分配，分配方式是优先从当前列表遍历，找到一个没有引用且`timestamp`最小的缓冲区，如果没有就申请下一个桶的锁，并遍历该桶，找到后将该缓冲区从原来的桶移动到当前桶中，最多将所有桶都遍历完。

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

  int bid = HASH(blockno);
  //注意获取锁
  acquire(&bcache.buckets[bid].lock);

  // Is the block already cached?
  for(b = bcache.buckets[bid].head.next; b != &bcache.buckets[bid].head; b = b-
>next) {
    if(b->dev == dev && b->blockno == blockno) {
      b->refcnt++;

      // 记录使用时间戳
      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  b = 0;
  struct buf* tmp;

  // Recycle the least recently used (LRU) unused buffer.
  // 从当前散列桶开始查找
  for(int i = bid, cycle = 0; cycle != NBUCKET; i = (i + 1) % NBUCKET) {
    ++cycle;
    // 如果遍历到当前散列桶，则不重新获取锁
    if(i != bid) {
      if(!holding(&bcache.buckets[i].lock))
        acquire(&bcache.buckets[i].lock);
      else
        continue;
    }

    for(tmp = bcache.buckets[i].head.next; tmp != &bcache.buckets[i].head; tmp =
tmp->next)
      // 使用时间戳进行LRU算法，而不是根据结点在链表中的位置
      if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp))
        b = tmp;

    if(b) {
      // 如果是从其他散列桶窃取的，则将其以头插法插入到当前桶
      if(i != bid) {
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.buckets[i].lock);

        b->next = bcache.buckets[bid].head.next;
        b->prev = &bcache.buckets[bid].head;
        bcache.buckets[bid].head.next->prev = b;
        bcache.buckets[bid].head.next = b;
      }

      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;

      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
    } else {
      // 在当前散列桶中未找到，则直接释放锁
      if(i != bid)
        release(&bcache.buckets[i].lock);
    }
  }

  panic("bget: no buffers");
}
```

6. 最后将`bpin`和`bunpin`的锁替换为桶级锁

```
void
bpin(struct buf *b)
{
  int bid = HASH(b->blockno);
  acquire(&bcache.buckets[bid].lock);
  b->refcnt++;
  release(&bcache.buckets[bid].lock);
}

void
bunpin(struct buf *b) {
  int bid = HASH(b->blockno);
  acquire(&bcache.buckets[bid].lock);
  b->refcnt--;
  release(&bcache.buckets[bid].lock);
}
```

实验结果

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem_cpu_0: #fetch-and-add 0 #acquire() 32906
lock: kmem_cpu_1: #fetch-and-add 0 #acquire() 46
lock: kmem_cpu_2: #fetch-and-add 0 #acquire() 137
```

```
lock: bcache_eviction: #fetch-and-add 0 #acquire() 9
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6405
lock: bcache_eviction: #fetch-and-add 0 #acquire() 10
lock: bcache_bufmap: #fetch-and-add 218 #acquire() 6839
lock: bcache_eviction: #fetch-and-add 0 #acquire() 11
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6821
lock: bcache_eviction: #fetch-and-add 0 #acquire() 9
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 8681
lock: bcache_eviction: #fetch-and-add 0 #acquire() 5
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4206
lock: bcache_eviction: #fetch-and-add 0 #acquire() 5
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4206
lock: bcache_eviction: #fetch-and-add 0 #acquire() 3
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 2194
lock: bcache_eviction: #fetch-and-add 0 #acquire() 3
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4202
lock: bcache_eviction: #fetch-and-add 0 #acquire() 2
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 2193
lock: bcache_eviction: #fetch-and-add 0 #acquire() 4
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4205
lock: bcache_eviction: #fetch-and-add 0 #acquire() 7
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 4419
lock: bcache_eviction: #fetch-and-add 0 #acquire() 7
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6425
lock: bcache_eviction: #fetch-and-add 0 #acquire() 8
lock: bcache_bufmap: #fetch-and-add 0 #acquire() 6420
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 3798974 #acquire() 1182
lock: proc: #fetch-and-add 412238 #acquire() 81331
lock: proc: #fetch-and-add 393675 #acquire() 81289
lock: proc: #fetch-and-add 339530 #acquire() 81289
lock: proc: #fetch-and-add 332930 #acquire() 81290
tot= 218
test0: OK
start test1
test1 OK
$ usertests
usertests starting
test manywrites: 1

OK
test execout: 1
OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3247
            sepc=0x00000000000056a4 stval=0x00000000000056a4
usertrap(): unexpected scause 0x000000000000000c pid=3248
            sepc=0x00000000000056a4 stval=0x00000000000056a4
```

```
OK
test badarg:
OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test dirtest: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000215c stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000215c stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000215c stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000215c stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000215c stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000215c stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000215c stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6235
            sepc=0x000000000000215c stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6236
            sepc=0x000000000000215c stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6237
            sepc=0x000000000000215c stval=0x000000008006ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6238
            sepc=0x000000000000215c stval=0x000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6239
            sepc=0x000000000000215c stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6240
            sepc=0x000000000000215c stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x000000000000215c stval=0x000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6242
            sepc=0x000000000000215c stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6243
            sepc=0x000000000000215c stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6244
            sepc=0x000000000000215c stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6245
```

```
            sepc=0x000000000000215c stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x000000000000215c stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x000000000000215c stval=0x00000000800e7ef0
usertrap(): unexpected scause 0x000000000000000d pid=6248
            sepc=0x000000000000215c stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6249
            sepc=0x000000000000215c stval=0x0000000080100590
usertrap(): unexpected scause 0x000000000000000d pid=6250
            sepc=0x000000000000215c stval=0x000000008010c8e0
usertrap(): unexpected scause 0x000000000000000d pid=6251
            sepc=0x000000000000215c stval=0x0000000080118c30
usertrap(): unexpected scause 0x000000000000000d pid=6252
            sepc=0x000000000000215c stval=0x0000000080124f80
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x000000000000215c stval=0x00000000801312d0
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000215c stval=0x000000008013d620
usertrap(): unexpected scause 0x000000000000000d pid=6255
            sepc=0x000000000000215c stval=0x0000000080149970
usertrap(): unexpected scause 0x000000000000000d pid=6256
            sepc=0x000000000000215c stval=0x0000000080155cc0
usertrap(): unexpected scause 0x000000000000000d pid=6257
            sepc=0x000000000000215c stval=0x0000000080162010
usertrap(): unexpected scause 0x000000000000000d pid=6258
            sepc=0x000000000000215c stval=0x000000008016e360
usertrap(): unexpected scause 0x000000000000000d pid=6259
            sepc=0x000000000000215c stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6260
            sepc=0x000000000000215c stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6261
            sepc=0x000000000000215c stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6262
            sepc=0x000000000000215c stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6263
            sepc=0x000000000000215c stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6264
            sepc=0x000000000000215c stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6265
            sepc=0x000000000000215c stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6266
            sepc=0x000000000000215c stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6267
            sepc=0x000000000000215c stval=0x00000000801dc130
OK
test sbrkfail:
usertrap(): unexpected scause 0x000000000000000d pid=6279
            sepc=0x00000000000041fc stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6283
            sepc=0x00000000000022cc stval=0x000000000000fb90
OK
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 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
```

最终结果

```
== Test running kalloctest ==
$ make qemu-gdb
(155.4s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (18.3s)
== Test running bcachetest ==
$ make qemu-gdb
(31.1s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (249.2s)
== Test time ==
time: OK
Score: 70/70
```

实验感悟

经过这次实验，让我更加深刻的了解的锁的机制，以及更加清晰的了解死锁的形成，并看到了如何避免死锁的方法。Xv6 实现了两种类型的锁：自旋锁和睡眠锁。

（1）Xv6 将自旋锁表示为一个 结构体 `spinlock`（kernel/spinlock.h:2）。该结构中 包含locked字段，当锁可获得时， locked 为零，当锁被持有时，locked 为非零。但有时 xv6 需要长时间保持一个锁。例如，文件系统在磁盘上读写文件内容 时，会保持一个文件的锁定，这些磁盘操作可能需要几十毫秒。如果另一个进程想获取一个 50 自旋锁，那么保持那么长的时间会导致浪费，因为获取进程在自旋的同时会浪费 CPU 很长 时间。自旋锁的另一个缺点是，一个进程在保留自旋锁的同时不能让出 CPU；我们希望做到 这一点，这样其他进程可以在拥有锁的进程等待磁盘的时候使用 CPU。在保留自旋锁的同时 让出是非法的，因为如果第二个线程试图获取自旋锁，可能会导致死锁；因为获取自旋锁不 会让出 CPU，第二个线程的自旋可能会阻止第一个线程运行并释放锁。在持有锁的同时让出 也会违反在持有自旋锁时中断必须关闭的要求。因此，我们希望有一种锁，在等待获取的过 程中产生 CPU，并在锁被持有时允许让出 cpu（和中断）。

（2）因此Xv6也提供了睡眠锁(sleep-locks)。acquiresleep(kernel/sleeplock.c:22) 在等待的过程中让出 CPU。在高层次上，sleep-lock 有一个由 spinlock 保护的锁定字段，而 acquiresleep 对 sleep 的调用会原子性地让出 CPU 并释放 spinlock。因此在 acquiresleep 等待时，其他线程便可以继续执行。

最后通过实验，学会了利用哈希函数来实现效率的提升这一个很有用的技巧。得到了很大的收获。