**实验八 locks**

**8.1 实验目的**

1. 提高对于并行性的认识

2. 理解多核机器上并行性差的常见症状为high lock contention

3. 学习如何通过更改数据结构和锁定策略减少争用

**8.2 实验内容**

8.2.1 Memory allocator（moderate）

您的工作是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。您必须给出所有以“kmem”开头的锁的名称。也就是说，您应该为每个锁调用initlock，并传递一个以“kmem”开头的名称。运行kalloctest以查看您的实现是否减少了锁争用。要检查它是否仍然可以分配所有内存，请运行usertests sbrkmuch。您的输出将与下面所示的类似，在kmem锁上的争用总数将大大减少，尽管具体的数字会有所不同。确保usertests中的所有测试都通过。

8.2.2 Buffer cache（hard）

修改块缓存，以便在运行bcachetest时，bcache中所有锁的获取循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但如果总和小于500则可以。修改bget和brelse，以便bcache中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待bcache.lock）。必须保持不变，即每个块最多缓存一个副本。完成后，您的输出应该与下面显示的类似（尽管不完全相同）。

8.2.3 submit

提交实验。

**8.3 实验过程**

8.3.1 memory allocator

（1）本次实验目的为对多CPU为每个CPU拆分一个空闲空间链表并采用单独的锁管理来降低锁冲突带来的额外开销，增加并行性。

（2）在param.h中定义了xv6支持最大并行的CPU数量NCPU，因此将kalloc.c中kmem定义为NCPU个。

struct {

  struct spinlock lock;

  struct run \*freelist;

} kmem[NCPU];

（2）对kmem初始化进行修改

void

kinit()

{

  for(int i=0;i<NCPU;i++)

  {

      initlock(&kmem[i].lock, "kmem");

  }

  freerange(end, (void\*)PHYSTOP);

}

（3）修改kfree，对于释放页面放入freelist的机制进行修改。

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();// turn interrupts off

  int i=cpuid();// core number

  acquire(&kmem[i].lock);

  r->next = kmem[i].freelist;

  kmem[i].freelist = r;

  release(&kmem[i].lock);

  pop\_off();//turn inturrupts on

}

（4）修改kalloc函数，当在当前核心上申请失败时尝试从其他核心上获取页面。

void

void \*

kalloc(void)

{

  struct run \*r;

  push\_off();// turn interrupts off

  int i=cpuid();// core number

  acquire(&kmem[i].lock);

  r = kmem[i].freelist;

  if(r)

  {

    kmem[i].freelist = r->next;

  }

  release(&kmem[i].lock);

  if(!r)

  {

     for(int j=0;j<NCPU;j++)

     {

       if(j!=i)

       {

          acquire(&kmem[j].lock);

          if(kmem[j].freelist)

    {

        r=kmem[j].freelist;

              kmem[j].freelist=r->next;

        release(&kmem[j].lock);

        break;

    }

          release(&kmem[j].lock);

       }

     }

  }

  pop\_off();

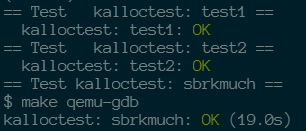
  if(r)

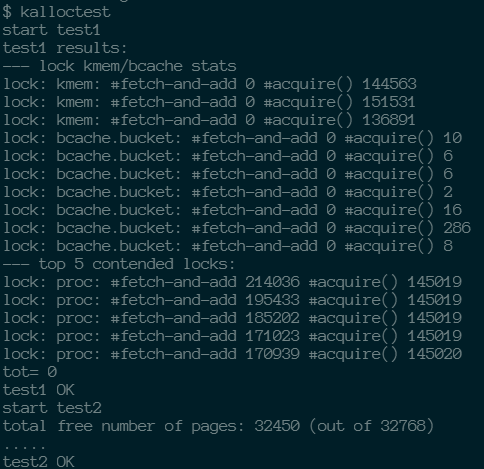
    memset((char\*)r, 5, PGSIZE); // fill with junk

  return (void\*)r;

}

（5）运行memory allocator





8.3.2 buffer cache

（1）在多个进程密集的使用文件系统时，他们可能会争用bcache.lock，所以针对磁盘空间中的块号进行散列处理，并用哈希表维护缓冲区。

（2）根据提示我们决定哈希桶的大小为13

#define NBUCKET 13

并在buf.h中加入timestamp用于替代原有链表LRU功能

  uint time\_stamp;

（3）在boi.c中的brelse中维护timestamp。

  if (b->refcnt == 0) {

    // no one is waiting for it.

    b->time\_stamp=ticks;

  }

（4）为每个哈希桶加上自旋锁并提供映射规则。

struct bucket {

 struct spinlock lock;

 struct buf head;

}hashtable[NBUCKET];

int

hash(uint dev, uint blockno)

{

 return blockno % NBUCKET;

}

（5）在binit中加入对哈希表的初始化

void

binit(void)

{

  struct buf \*b;

  initlock(&bcache.lock, "bcache");

  for(b = bcache.buf; b < bcache.buf+NBUF; b++){

    initsleeplock(&b->lock, "buffer");

  }

  b = bcache.buf;

  for (int i = 0; i < NBUCKET; i++) {

    initlock(&hashtable[i].lock, "bcache\_bucket");

    for (int j = 0; j < NBUF / NBUCKET; j++) {

      b->blockno = i;

      b->next = hashtable[i].head.next;

      hashtable[i].head.next = b;

      b++;

    }

  }

}

（6）当需要查找一个块并将其替换的时候找到timestamp最小的空闲块，如果没有这个空闲块就需要在哈希表中寻找，修改bget函数。

static struct buf\*

bget(uint dev, uint blockno)

{

  // printf("dev: %d blockno: %d Status: ", dev, blockno);

  struct buf \*b;

  int idx = hash(dev, blockno);

  struct bucket\* bucket = hashtable + idx;

  acquire(&bucket->lock);

  // Is the block already cached?

  for(b = bucket->head.next; b != 0; b = b->next){

    if(b->dev == dev && b->blockno == blockno){

      b->refcnt++;

      release(&bucket->lock);

      acquiresleep(&b->lock);

      // printf("Cached %p\n", b);

      return b;

    }

  }

  // Not cached.

  // First try to find in current bucket.

  int min\_time = 0x8fffffff;

  struct buf\* replace\_buf = 0;

  for(b = bucket->head.next; b != 0; b = b->next){

    if(b->refcnt == 0 && b->timestamp < min\_time) {

      replace\_buf = b;

      min\_time = b->timestamp;

    }

  }

  if(replace\_buf) {

    // printf("Local %d %p\n", idx, replace\_buf);

    goto find;

  }

  // Try to find in other bucket.

  acquire(&bcache.lock);

  refind:

  for(b = bcache.buf; b < bcache.buf + NBUF; b++) {

    if(b->refcnt == 0 && b->timestamp < min\_time) {

      replace\_buf = b;

      min\_time = b->timestamp;

    }

  }

  if (replace\_buf) {

    // remove from old bucket

    int ridx = hash(replace\_buf->dev, replace\_buf->blockno);

    acquire(&hashtable[ridx].lock);

    if(replace\_buf->refcnt != 0)  // be used in another bucket's local find between finded and acquire

    {

      release(&hashtable[ridx].lock);

      goto refind;

    }

    struct buf \*pre = &hashtable[ridx].head;

    struct buf \*p = hashtable[ridx].head.next;

    while (p != replace\_buf) {

      pre = pre->next;

      p = p->next;

    }

    pre->next = p->next;

    release(&hashtable[ridx].lock);

    // add to current bucket

    replace\_buf->next = hashtable[idx].head.next;

    hashtable[idx].head.next = replace\_buf;

    release(&bcache.lock);

    // printf("Global %d -> %d %p\n", ridx, idx, replace\_buf);

    goto find;

  }

  else {

    panic("bget: no buffers");

  }

  find:

  replace\_buf->dev = dev;

  replace\_buf->blockno = blockno;

  replace\_buf->valid = 0;

  replace\_buf->refcnt = 1;

  release(&bucket->lock);

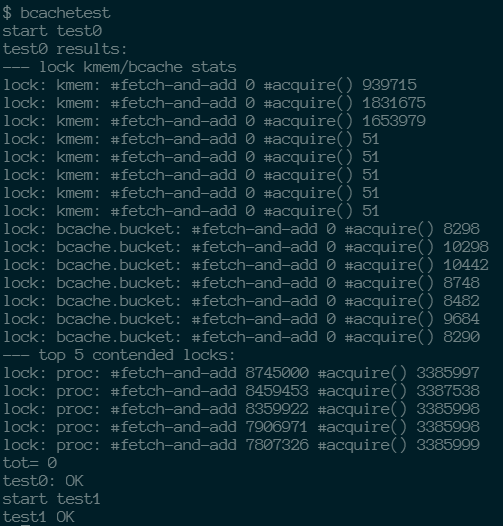
  acquiresleep(&replace\_buf->lock);

  return replace\_buf;

}

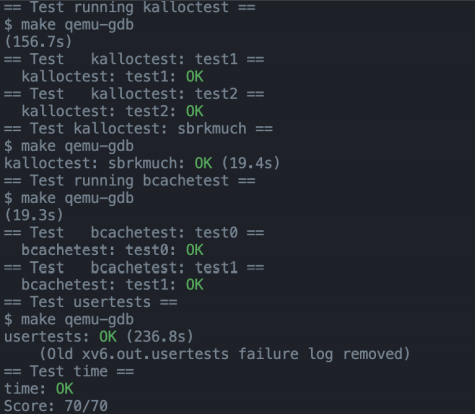
将brelse、bpin、bunpin中的锁替换为哈希桶的锁hashtable[i].lock即完成。

（7）运行bcachetest



**8.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**8.5 实验小结**

本次实验考察的是对于并行系统的优化，通过分割资源来使高锁竞争得到缓解，提高其并行利用率，在文件较大或者其他情况下这对于操作系统的优化较大，可以提高资源的利用率。

在处理多线程或多进程中资源抢占导致的锁竞争现象可以通过分割资源并分别加锁，锁的数量越多单个锁上冲突就越少。而这种并行性正是操作系统提高资源利用率所要追求的。