# Lab8 locks

### 实验提要：

In this lab you'll gain experience in re-designing code to increase parallelism. A common symptom of poor parallelism on multi-core machines is high lock contention. Improving parallelism often involves changing both data structures and locking strategies in order to reduce contention. You'll do this for the xv6 memory allocator and block cache.

在本实验中，您将获得重新设计代码以提高并行度的经验。 多核计算机上并行性差的常见症状是锁争用较高。 改善并行性通常涉及更改数据结构和锁定策略，以减少争用。 您将为xv6内存分配器和块缓存执行此操作。

## Memory allocator

### 实验目的：

Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call initlock for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run usertests sbrkmuch. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in usertests pass. make grade should say that the kalloctests pass.

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。 您必须提供所有以`kmem`开头的锁名称。 也就是说，您应该为每个锁调用`initlock`，并传递以` kmem`开头的名称。 运行`kalloctest`以查看您的实现是否减少了锁争用。 要检查它是否仍可以分配所有内存，请运行`usertests sbrkmuch`。 您的输出将类似于以下所示，尽管`kmem`锁的总争用数量有所减少，但总争用减少了很多。 确保`usertests`中的所有测试均通过。` make grade`应该说`kalloctests`通过了。

### 实验步骤：

1. 在kernel/kalloc.c中将kmem修改为数组，为每个CPU分配一个kmem

struct {

  struct spinlock lock;

  struct run \*freelist;

} kmem[NCPU];

1. 修改kinit函数，为当前CPU的’freelist’分配所有的空闲内存空间

  for (int i = 0; i < NCPU; i++)

    initlock(&kmem[i].lock, "kmem");

1. 修改kfree函数，让他在获取cpuid前需要关闭中断

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;

  //获取cpuid前关闭中断

  push\_off();

  int id = cpuid();

  pop\_off();

  acquire(&kmem[id].lock);

  r->next = kmem[id].freelist;

  kmem[id].freelist = r;

  release(&kmem[id].lock);

}

1. 修改kalloc函数，用于查找当前CPU中是否有空闲块，没有则需要从其他CPU中获取。

void \*

kalloc(void)

{

  struct run \*r;

  push\_off();

  int id = cpuid();

  pop\_off();

  acquire(&kmem[id].lock);

  r = kmem[id].freelist;

  if(r)

    kmem[id].freelist = r->next;

  else {

    for (int i = 0; i < NCPU; i++) {

      if (i == id)

          continue;

      acquire(&kmem[i].lock);

      r = kmem[i].freelist;

      if(r)

        kmem[i].freelist = r->next;

      release(&kmem[i].lock);

      if(r)

          break;

    }

  }

  release(&kmem[id].lock);

  if(r)

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

  return (void\*)r;

}

实验中遇到的问题和解决办法：

在freelist中cpu1不够去借cpu2的页，cpu2也不够也去借cpu3的页，但cpu3也不够去借cpu1的页时可能会产生死锁，一种解决方式是先释放自己的锁再去借页，把借的页头指针先保存起来，再申请本cpu的锁来更新空闲链表。

实验小结：

该 lab 的实验目标，即是为每个CPU分配独立的 freelist，这样多个 CPU并发分配物理页就不再会互相排斥了，提高了并行性，这样的思路非常精妙，既可以避免资源的竞争，又可以充分利用资源，这种思想其实在内存分页等方面都有体现。

## Buffer cache

### 实验目的

Modify the block cache so that the number of acquire loop iterations for all locks in the bcache is close to zero when running bcachetest. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify bget and brelse so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for bcache.lock).

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。目的是通过重新设计文件系统的 buffer & cache 的数据结构来降低锁的竞争

### 实验步骤

1. 根据实验指导，修改kernel/bio.c中的bcache结构体

struct {

  struct spinlock biglock;

  struct spinlock lock[NBUCKET];//每个桶的锁

  struct buf buf[NBUF];

  // 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[NBUCKET];//

} bcache;

将buf修改为大小为13的数组，获取trap.c中的ticks,并且还要所有桶中的锁

1. 在buf.h头文件中添加lastuse声明

  uint lastuse;

1. 修改binit函数，将双向链表进行初始化操作，每个结点前驱结点和后继结点都是源节点

void

binit(void)

{

  struct buf \*b;

  initlock(&bcache.biglock, "bcache\_biglock");

  for (int i = 0; i < NBUCKET; i++)

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

  // Create linked list of buffers

  for (int i = 0; i < NBUCKET; i++) {

    bcache.head[i].next = &bcache.head[i];

    bcache.head[i].prev = &bcache.head[i];

  }

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

    b->next = bcache.head[0].next;

    b->prev = &bcache.head[0];

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

    bcache.head[0].next->prev = b;

    bcache.head[0].next = b;

  }

}

1. 修改bio.c下的brelse,bunpin,bpin函数，并增加一个散列函数hash实现加锁和散列。

int

hash(int blockno)

{

  return blockno % NBUCKET;

}

void

brelse(struct buf \*b)

{

  if(!holdingsleep(&b->lock))

    panic("brelse");

  releasesleep(&b->lock);

  int i = hash(b->blockno);

  acquire(&bcache.lock[i]);

  b->refcnt--;

  if (b->refcnt == 0) {

    // no one is waiting for it.

    //b->next->prev = b->prev;

    //b->prev->next = b->next;

    //b->next = bcache.head[i].next;

    //b->prev = &bcache.head[i];

    //bcache.head[i].next->prev = b;

    //bcache.head[i].next = b;

    b->lastuse = ticks;

  }

  release(&bcache.lock[i]);

}

void

bpin(struct buf \*b) {

  int i = hash(b->blockno);

  acquire(&bcache.lock[i]);

  b->refcnt++;

  release(&bcache.lock[i]);

}

void

bunpin(struct buf \*b) {

  int i = hash(b->blockno);

  acquire(&bcache.lock[i]);

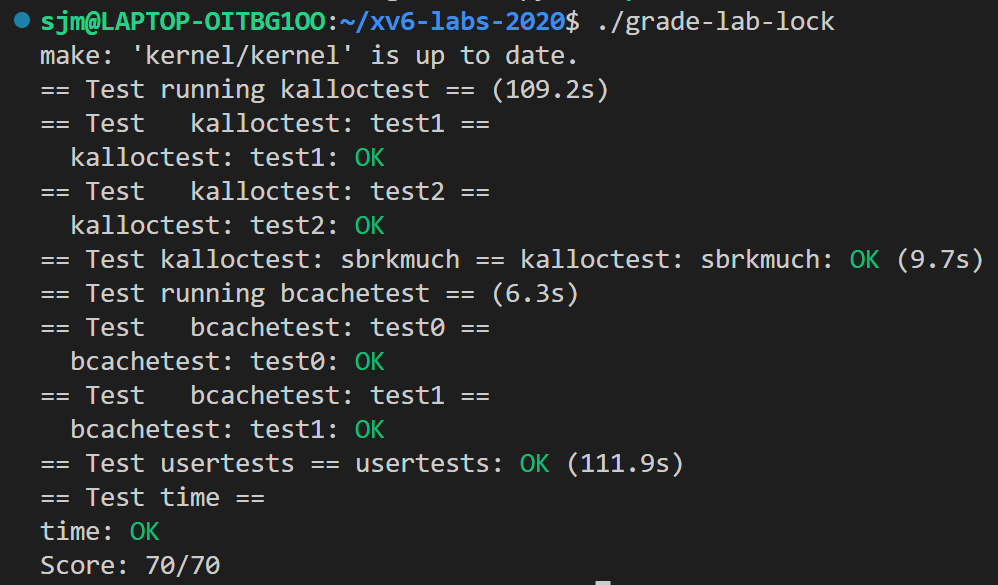
  b->refcnt--;

  release(&bcache.lock[i]);

}

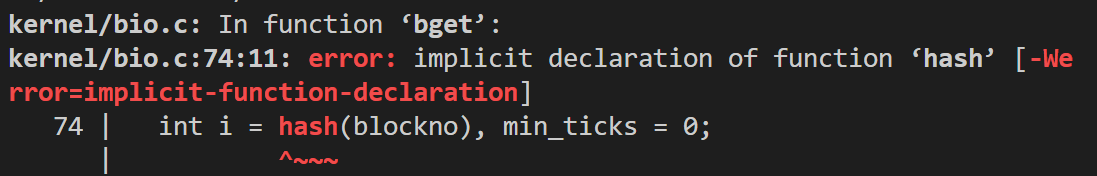
1. 最后修改bget函数，先遍历判断缓冲是否命中，若命中则直接返回，若没有命中则要释放锁。因为释放锁后可能会有新的缓存，所以需要挨次遍历检查是否命中，若还是没有命中则获取这个桶的空闲块，若该桶没有空闲块则去别的桶寻找。

在终端运行./grade-lab-lock，实验结果如下：



### 实验中遇到的问题和解决办法

1.找不到相应函数



解决办法是在使用某个函数前需要先声明（头文件中或者在该.c文件头）

1. 如果在评分的时候有类似panic:freeing free block的报错，俺么原因可能是在测试前没有make clean清理.  
   实验心得

锁是管理多核并行必须要使用的保护机制，但并不是随意在任何时机任何粒度下使用锁，如果使用锁不当，可能会使程序串行执行，大大降低性能。