Lab4: locks

- 姓名：吴欣怡
- 学号：PB21051111
- 虚拟机用户名：OS-PB21051111

Memory allocator

实验分析

分析实验文档中的要求:
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。所
有锁的命名必须以“kmem”开头，尽力减少锁争用。

实验过程

定义一个kmem元素，长NCPU的数组结构体

```
struct kmem{
  struct spinlock lock;
  struct run *freelist;
};

struct kmem kemArray[NCPU];
```

修改kinit函数，初始化数组的锁

```
void
kinit()
{
  for(int i=0;i<NCPU;i++){
    initlock(&(kemArray[i].lock), "kmem");
  }
  freerange(end, (void*)PHYSTOP);
}
```

修改kfree函数，kfree 函数的作用是将一个已分配的物理内存页标记为可用，以便后续的内存分配可以重新使用这个物理页。
push_off 函数关闭中断，避免并发。
获取当前 CPU 的 ID，即 cpuid() 返回值。
获取当前 CPU 对应的内存池的锁，即 kemArray[cpuId].lock。
将释放的物理页（r）插入到当前 CPU 对应的空闲页链表的头部。
释放完毕后，使用 pop_off 函数打开中断，解锁。

```
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();
  int cpuId=cpuid();
  acquire(&(kemArray[cpuId].lock));
  r->next = kemArray[cpuId].freelist;
  kemArray[cpuId].freelist = r;
  release(&(kemArray[cpuId].lock));
  pop_off();
}
```

修改kalloc,关闭中断：

使用 push_off 函数关闭中断，目的是为了避免并发问题。
获取当前 CPU ID：

获取当前 CPU 对应的内存池的锁，即 kemArray[cpuId].lock。
尝试从当前 CPU 的空闲页链表中分配内存：

将当前 CPU 空闲页链表的头部（即 kemArray[cpuId].freelist）赋值给指针变量 r。
如果 r 不为空（表示有空闲页），将当前 CPU 的空闲页链表的头指针指向下一个节点。
如果当前 CPU 没有空闲页，则尝试从其他 CPU 的空闲页链表中偷取：

使用 for 循环遍历其他 CPU（除了当前 CPU）。
在找到有空闲页的 CPU 后，获取该 CPU 的锁，并从其空闲页链表头部取出一个空闲页（r）。在偷取的过程中，先获取目标 CPU 的锁，然后释放该锁。再获取当前 CPU 的锁，以避免死锁。
释放当前 CPU 对应的内存池的锁。
使用 pop_off 函数打开中断。

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

  push_off();
  int cpuId=cpuid();
  acquire(&(kemArray[cpuId].lock));
  r = kemArray[cpuId].freelist;
  if(r){
    kemArray[cpuId].freelist = r->next;}
  else{
    for(int i =(cpuId+1)%NCPU, j=0;j< NCPU-1; i= (i+1)%NCPU,j++){
```

```
      if (kemArray[i].freelist){
        acquire(&(kemArray[i].lock));
        r= kemArray[i].freelist;
        kemArray[i].freelist= r->next;
        release(&(kemArray[i].lock));
        break;
      }
     }
  }
  release(&(kemArray[cpuId].lock));

  pop_off();

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

Buffer　cache

实验分析

1.修改块缓存，以便在运行bcachetest 时，bcache（buffer cache的缩写）中所有锁的acquire 循环迭代次数接近于零。
2.处理两个进程同时使用相同的块号、
当两个进程同时在cache中未命中时、两个进程同时使用冲突的块时等情况。

实验过程

第一步

定义总的链表数：13

```
#define NBUCKETS 13
```

修改bcache的定义

```
struct {
  struct spinlock lock[NBUF];
  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[NBUF];
} bcache;
```

修改binit,循环遍历NBUCKETS个哈希桶。
对于每个哈希桶，初始化相应的锁（bcache.lock[i]）。
初始化链表表头（bcache.head[i]）为一个循环链表，即 prev 和 next 都指向自己。这样，每个哈希桶维护了一个独立的缓冲区链表。

```
void
binit(void)
{
  struct buf *b;

  for(int i=0;i<NBUCKETS;i++)
  {
    //初始化所有锁
    initlock(&bcache.lock[i], "bcache");
    //初始化所有链表表头
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
  }

  //把所有块插到0号链表里
  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;
  }
}
```

解释一下参数
dev（Device）：
表示缓冲区块所属的设备。
blockno（Block Number）：表示缓冲区块在设备上的块号。文件系统通常将磁盘分为多个块，每个块都有一个唯一的块号。blockno 存储了缓冲区块的块号。
valid：用于标识缓冲区块的内容是否有效。如果 valid 为 1，表示缓冲区块中的数据有效；如果为 0，表示数据无效。通常，当从磁盘读取数据到缓冲区块时，会将 valid 置为 1。
refcnt（Reference Count）：表示缓冲区块的引用计数。引用计数用于跟踪有多少个指针指向该缓冲区块。当 bget 函数找到一个已经缓存的缓冲区块时，会增加其引用计数。在释放缓冲区块时，会减少引用计数。当引用计数为 0 时，说明没有指针引用该缓冲区块，可以被重新分配使用。

修改bget：具体步骤：用锁确保同一哈希桶内的操作是互斥的，从而提高并发性。
已缓存情况：当请求的块已经在缓冲区中时，直接找到对应的缓冲区块，增加引用计数，释放哈希桶锁，然后获取缓冲区块的睡眠锁，最后返回该缓冲区块。
未缓存情况：在当前哈希桶的链表中寻找一个未被引用的块。如果找到，设置相应的属性（dev、blockno、valid、refcnt）。
如果在当前哈希桶未找到未被引用的块，那么遍历其他哈希桶的链表，寻找未被引用的块，并将其移动到当前哈希桶的链表头，以便下次更容易找到。
死锁避免：在从其他哈希桶偷取未被引用的块时，确保先释放被偷取哈希桶的锁，再获取当前哈希桶的锁，以避免死锁。

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

  int id=blockno%NBUCKETS;

  acquire(&bcache.lock[id]);

  //已缓存
  for(b = bcache.head[id].next; b != &bcache.head[id]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock[id]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  //未缓存
  //在当前链表里找到一个未使用块，缓存在这里
  for(b = bcache.head[id].prev; b != &bcache.head[id]; b = b->prev){
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;

      release(&bcache.lock[id]);


      acquiresleep(&b->lock);
      return b;
    }
  }

  //在当前链表里未找到未使用块,去其他链表里偷
  for(uint i=1;i<NBUCKETS;i++)
  {
    uint steal_id=(id+i)%NBUCKETS;
    acquire(&bcache.lock[steal_id]);
    for(b = bcache.head[steal_id].prev; b != &bcache.head[steal_id]; b = b->prev)
    {
      if(b->refcnt == 0)
      {
      //
          release(&bcache.lock[steal_id]);
          b->dev = dev;
          b->blockno = blockno;
          b->valid = 0;
          b->refcnt = 1;
```

```
          //从原来链表剥离，查找自己链表的head.next处
          b->next->prev = b->prev;
          b->prev->next = b->next;

          b->next = bcache.head[id].next;
          b->prev = &bcache.head[id];
          bcache.head[id].next->prev = b;
          bcache.head[id].next = b;

          release(&bcache.lock[id]);

          //release(&bcache.lock[steal_id]);
          //这里是之前写错的地方，导致了死锁

          acquiresleep(&b->lock);
          return b;
      }
    }
    release(&bcache.lock[steal_id]);
  }
  panic("bget: no buffers");
}
```

修改brelse，其作用是释放一个已经被获取（锁定）的缓冲区块，并将其放回缓冲区。
获取哈希桶索引 id。，获取哈希桶的锁（bcache.lock[id]），确保对当前哈希桶的操作是互斥的。
检查是否持有缓冲区块的睡眠锁：使用 holdingsleep 函数检查当前线程是否已经持有了缓冲区块的睡眠锁。
如果没有持有，说明存在错误，产生 panic。
释放缓冲区块的睡眠锁：使用 releasesleep 函数释放缓冲区块的睡眠锁，允许其他线程访问该缓冲区块。
操作引用计数和链表：减少缓冲区块的引用计数（b->refcnt--）。
如果引用计数减少到 0，表示没有其他线程在使用该缓冲区块，可以放回缓冲区。
将缓冲区块移动到哈希桶对应的链表头部，即最近使用的位置。这有助于提高缓冲区的效率。
释放哈希桶锁

```
void
brelse(struct buf *b)
{
  uint id=(b->blockno)%NBUCKETS;
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock[id]);
  b->refcnt--;
  if (b->refcnt == 0) {
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head[id].next;
    b->prev = &bcache.head[id];
    bcache.head[id].next->prev = b;
```

```
    bcache.head[id].next = b;
  }

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

修改bpin和bunpin，只需要把前面的模式改成对数组操作的就行。

```
void
bpin(struct buf *b) {
  uint id=(b->blockno)%NBUCKETS;
  acquire(&bcache.lock[id]);
  b->refcnt++;
  release(&bcache.lock[id]);
}

void
bunpin(struct buf *b) {
  uint id=(b->blockno)%NBUCKETS;
  acquire(&bcache.lock[id]);
  b->refcnt--;
  release(&bcache.lock[id]);
}
```

实验评分

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ python3 grade-lab-lock
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_LOCK -DLAB_LOCK -MD -mcmodel=medan
y -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/bio.
o kernel/bio.c
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o
 kernel/console.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o
 kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o k
ernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfil
e.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/stats.o kernel/sprintf.o
riscv64-unknown-elf-ld: warning: cannot find entry symbol _entry; defaulting to 0000000080000000
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
== Test running kalloctest == (180.7s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch == kalloctest: sbrkmuch: OK (28.1s)
== Test running bcachetest == (49.6s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests == usertests: OK (468.3s)
== Test time ==
time: OK
Score: 70/70
```

实验总结

经历了自己写出死锁占用内存页导致连qemu都无法运行的窘态，在舍弃修改从头来过的过程中参考了网上的资料，不过因为代码中还是存在潜在的死锁问题导致不能完全运行正确。自己检查出死锁的问题还是很有成就感的，可惜因为前面忘记保存错误版本而不能在自己原来写的上面debug，略有遗憾。