实验报告

18364067 罗淦元

1.switching between threads

第一部分要求我们实现线程系统的上下文切换机制并通过uthread测试。那么我们首先联想到进程切换体系中的上下文机制。为了对调度过程中寄存器值的保存和还原，我们只需要将callee-save寄存器保存起来，原线程调用thread_switch时先保存自己的上下文，将栈指针指向新线程的上下文，同时弹出恢复寄存器。所以线程调度总是调用thread_switch来进行，且只有线程再次被调度时才能恢复到调用thread_switch前的状态。参考这一做法，我们首先需要为线程创建一个上下文结构。

```
struct thread {
  uint64     ra;
  uint64     sp;

  //非易失性存储器
  uint64     s0;
  uint64     s1;
  uint64     s2;
  uint64     s3;
  uint64     s4;
  uint64     s5;
  uint64     s6;
  uint64     s7;
  uint64     s8;
  uint64     s9;
  uint64     s10;
  uint64     s11;

  char     stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

然后仿造swtch.S来进行上下文的保存。

```
    .globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
```

```
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)
    ret    /* return to ra */
```

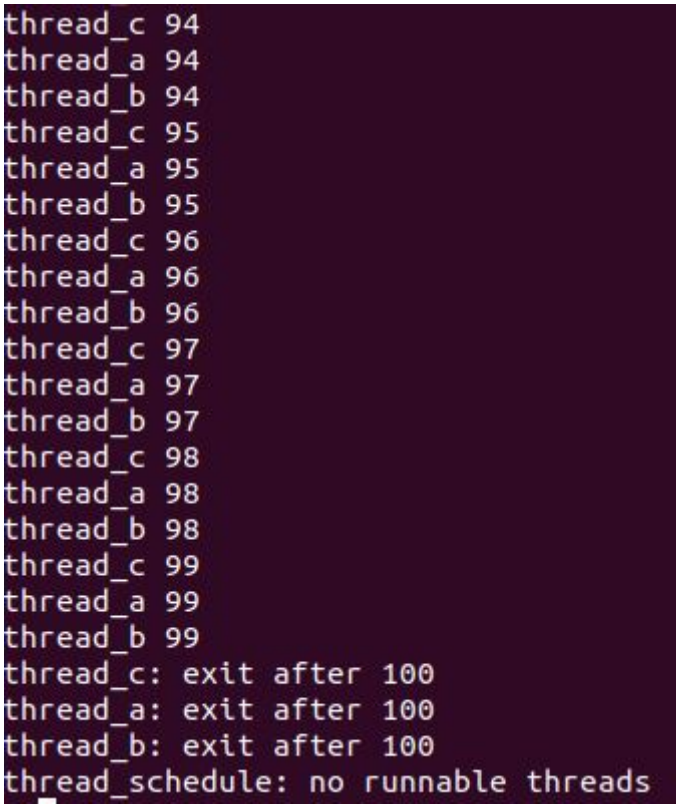
接着，在线程创建thread_create时，我们需要将进程的代码段存入到上下文的ra中，并将栈指针存入到sp中。

```
  t->ra = (uint64)func;
  t->sp = (uint64)(t->stack + STACK_SIZE);
```

这样前置工作就准备完成，每个线程在创建时将会准备自己的ra和sp，并在上下文切换时正确地存储起来。最后，在线程调度时调用thread_switch，使得线程的状态被存储起来。而当线程被调用，将会通过ra和sp恢复到调度前的状态。

```
thread_switch((uint64)t, (uint64)next_thread);
```

运行结果如下。



2.Memory allocator

根据题目要求，我们主要的目的是实现每个CPU单独的freelist，并且在CPU空闲列表为空时，尝试"窃取"其他CPU的freelist，使得进程能够减少争用次数并通过测试。我们首先分析kalloctest在系统中执行的流程。

对于test1，首先调用了ntas函数。ntas函数会调用statistics从statistics文件中读取数据到buffer，并在需要时打印出buffer内容。然后ntas在创建NCHILD个子进程并多次申请扩大4096个字节的空间，在新分配空间中其中第4*sizeof(int)的位置进行一次数据写入，再缩小4096个字节大小的进程空间。在这个过程中，多次调用了kalloc和kfree，导致了产生了对锁的频繁争抢。

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 119571 #acquire() 433016
lock: bcache: #fetch-and-add 0 #acquire() 334
--- top 5 contended locks:
lock: proc: #fetch-and-add 155090 #acquire() 525073
lock: kmem: #fetch-and-add 119571 #acquire() 433016
lock: proc: #fetch-and-add 92435 #acquire() 527530
lock: proc: #fetch-and-add 75414 #acquire() 526729
lock: proc: #fetch-and-add 32044 #acquire() 525057
tot= 119571
test1 FAIL
```

因此，我们需要为每个CPU维护一个freelist，实现对锁抢占情况的减少。首先，我们在kernel / param.h中设置NCPU为8，然后考虑kinit函数。

```
void kinit(){
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}
```

因为我们需要为每一个CPU维护一个freelist，因此在初始化阶段，需要给他们分别初始化一个独立的锁。

```
struct {
  struct spinlock lock;
  struct run *freelist;
}kmem[NCPU];                //对于每个CPU维护一个freelist结构

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

然后在初始阶段freerange时，通过修改kfree函数，初始化多个freelist到其对应cpu。

```
  //获取CPUID，需要为原子操作
  push_off();
  int hart = cpuid();
  pop_off();

  //记录内存块到本CPU的freelist
  acquire(&kmem[hart].lock);
  r->next = kmem[hart].freelist;
  kmem[hart].freelist = r;
  release(&kmem[hart].lock);
```

然后，在调用kalloc时，我们首先需要获取当前cpuid，并请求其对应的锁。如果此时freelist不为空，我们就能够直接将这一块内存从freelist分配出去。

```
   //获取CPUID
  push_off();
  int hart = cpuid();
  pop_off();

  acquire(&kmem[hart].lock);    //请求对应kmen的锁
  r = kmem[hart].freelist;
  if(r)
    kmem[hart].freelist = r->next;  //从freelist取出一块空余内存
  release(&kmem[hart].lock);
```

但是如果产生了freelist为空，且其他cpu中还有空余内存的情况，我们就需要从其他CPU中窃取一块内存。即抢占其他CPU的kmen锁并取出一块内存

```
  if (!r)   //如果没有空余内存
  {
    for (int i = 0; i < NCPU; i++)
    {
```

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

修改后的test1如图，可以看到这是有效的。单独维护的freelist减少了kmen和bcache锁的平均抢占。

```
kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 51363
lock: kmem: #fetch-and-add 0 #acquire() 193625
lock: kmem: #fetch-and-add 0 #acquire() 188081
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 178115 #acquire() 114
lock: proc: #fetch-and-add 126421 #acquire() 92242
lock: proc: #fetch-and-add 51412 #acquire() 92258
lock: proc: #fetch-and-add 21075 #acquire() 92463
lock: proc: #fetch-and-add 7267 #acquire() 92281
tot= 0
test1 OK
```

同样能够正常通过test2和usertests sbrkmuch

```
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

3.Buffer cache

这一部分需要解决的主要是磁盘块缓存在多个进程重复读取不同文件时，产生的bcache.lock抢占问题。根据要求，我们需要修改bget和brelse以实现并发查找和释放不会在锁上起冲突的块的释放。

我们首先需要改变buffer cache结构本身来实现哈希映射。原本的bcache结构包含一个自旋锁、一个大小为NBUF的buf块和一个buf头。根据提示，我们应该需要使其包含一个大小为BUCKETS_NUM的哈希表，且每个bucket有一个单独的锁。一开始，我们将BUCKETS_NUM预设为13。

```
struct {
  struct buf buf[NBUF];
  struct buf head[BUCKETS_NUM];
  struct spinlock lock[BUCKETS_NUM];
} bcache;
```

在初始化时，首先需要单独初始化每个bucket的lock和对应的双向链表，并将buf中所有的buffer放入到第一个bucket。后续其他bucket需要缓存块再从bucket#0"窃取"。

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

  //初始化每个bucket的lock
  for (int i = 0; i < BUCKETS_NUM; i++)
    initlock(&bcache.lock[i], "bucket");
  //初始化双向链表
  for (int i = 0; i < BUCKETS_NUM; i++)
  {
    bcache.head[i].prev = &bcache.head[i];
    bcache.head[i].next = &bcache.head[i];
  }
  //将所有的buffer加到第一个buckets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;
  }
}
```

在获取buffer时，不通过遍历全部buffer的方式，而是通过哈希函数获取桶，然后在桶内查找。为了简便起见，这里使用求余函数作为块号到桶号的映射。在桶中我们遍历查找有无对应的块，如果找到了就使其在锁上睡眠等待调用。

```
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;                    //增加查询次数记录
      release(&bcache.lock);          //释放bcache的锁
      acquiresleep(&b->lock);         //获取对应缓存块的睡眠锁
      return b;
    }
  }
```

反之，如果没有，就使用LRU算法将最久未使用的buffer扔出缓存区。具体来说，从当前块映射的bucket的下一个开始，查找其他桶内最少被使用的buffer块，如果其现在b->refcnt == 0不被使用，则将其回收并利用。

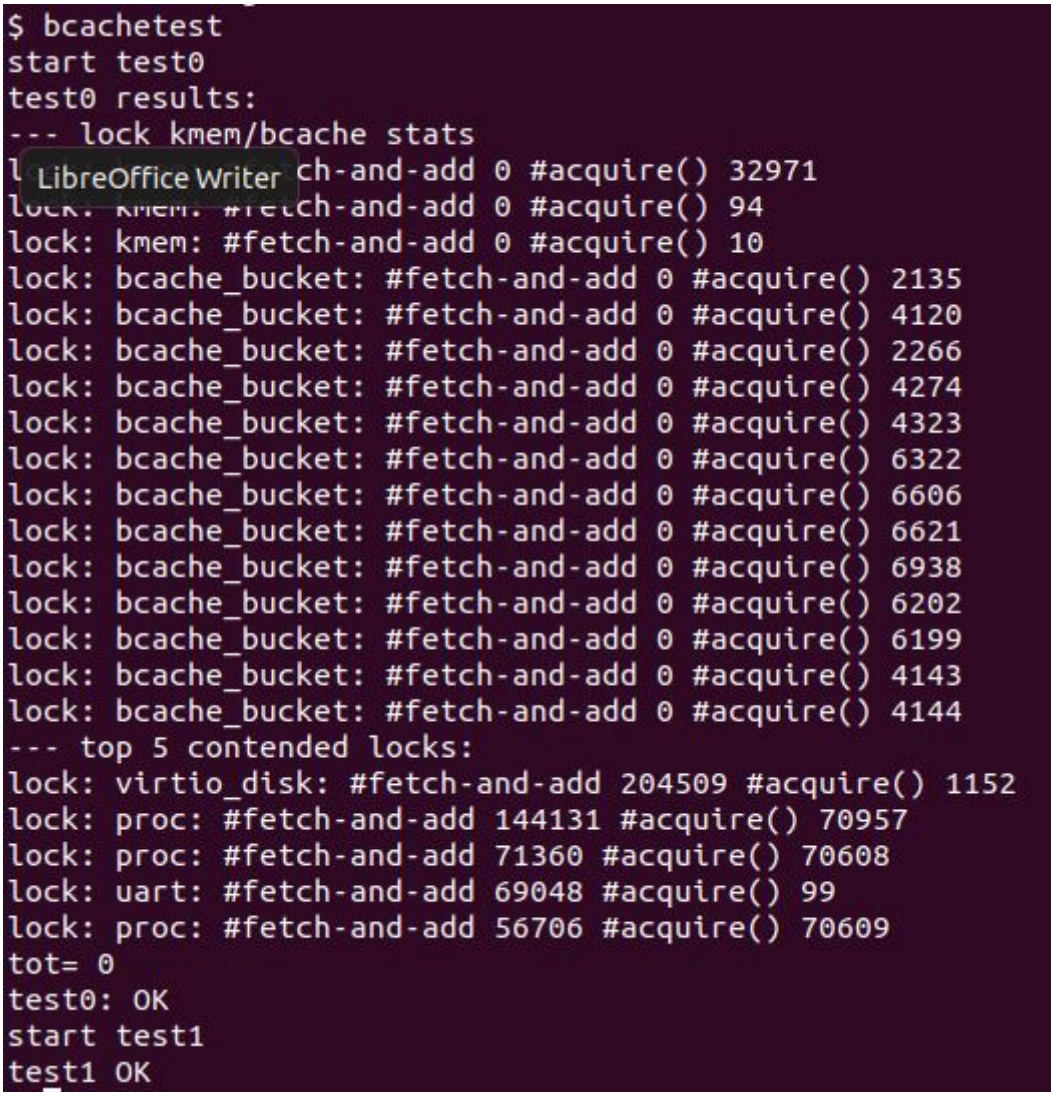
```
  int i = bucketno;
  while(1)
  {
    i = (i + 1) % NBUCKETS;
```

```
    if (i == bucketno)
      continue;

    acquire(&bcache.lock[i]);

    for (b = bcache.head[i].prev; b != &bcache.head[i]; b = b->prev)
    {
      if (b->refcnt == 0)
      {
        // 将buffer块从buffer#i移动到bucket#bucketno
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        b->prev->next = b->next;
        b->next->prev = b->prev;
        //释放i号桶锁
        release(&bcache.lock[i]);
        //将缓存块放入bucket#bucketno
        b->prev = &bcache.head[bucketno];
        b->next = bcache.head[bucketno].next;
        b->next->prev = b;
        b->prev->next = b;
        release(&bcache.lock[bucketno]);
        acquiresleep(&b->lock);
        return b;
      }
    }
```

测试结果如图。


$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
LibreOffice Writer
ch-and-add 0 #acquire() 32971
ch-and-add 0 #acquire() 94
lock: kmem: #fetch-and-add 0 #acquire() 10
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2135
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4120
lock: bcache_bucket: #fetch-and-add 0 #acquire() 2266
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4274
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4323
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6322
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6606
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6621
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6938
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6202
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4143
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4144
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 204509 #acquire() 1152
lock: proc: #fetch-and-add 144131 #acquire() 70957
lock: proc: #fetch-and-add 71360 #acquire() 70608
lock: uart: #fetch-and-add 69048 #acquire() 99
lock: proc: #fetch-and-add 56706 #acquire() 70609
tot= 0
test0: OK
start test1
test1 OK

usertests:

```
usertrap(): unexpected scause 0x000000000000000d pid=6227
            sepc=0x000000000000215c stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000215c stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000215c stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000215c stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000215c stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000215c stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000215c stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000215c stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x00000000000041fa stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6250
            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
```

4.Large File

由题目得，本题需要增加xv6文件的最大大小为65803个块。具体实现方法是更改索引节点，使其由直接块编号、单间接块编号和新增的双重间接块编号组成。由于双间接块需要牺牲一个直接块号，我们首先在fs.h中将NDIRECT值改为11，并修改MAXFILE为(NDIRECT + NINDIRECT + NINDIRECT*NINIDECT),需要注意的是由于NDIRECT改变，架构数据块地址索引改为

```
uint addrs[NDIRECT+2];   // Data block addresses
```

然后在bmap中增加映射。对于bn在(NDIRECT+NINDIRECT)以上，小于MAXFILE的情况，首先加载最后一个indirect block，如果为空说明这部分的块没有被分配。分配一块用来存储一级索引，然后通过除法获得一级索引中的下标，并在需要时分配一块，然后通过一级索引使用求余访问二级索引，来访问最终对应的块地址。

最终bmap函数如图所示。

```
static uint
```

```
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  bn -= NINDIRECT;
  uint *indirect, *dindirect,indirect_idx, dindirect_idx;
  struct buf *bp2;
  if (bn < NINDIRECT*NINDIRECT) {
    // 如果第NDIRECT+1位为0，说明双间接块号部分的数据还未被映射，分配一个块给第NDIRECT+1位
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT + 1] = addr =balloc(ip->dev);
    //读取对应的块，并在里面存储双间接块号
    bp = bread(ip->dev, addr);
    indirect = (uint *) bp->data;
    //获取块在双间接块时第一次映射的下标，如果该块未被分配则分配
    indirect_idx = bn / NINDIRECT;
    if ((addr = indirect[indirect_idx]) == 0) {
      addr = indirect[indirect_idx] =balloc(ip->dev);
      //写入日志
      log_write(bp);
    }
    //读取块到buffer
    bp2 = bread(ip->dev, addr);
    //获取双间接块第二次映射的下标
    dindirect = (uint *) bp2->data;
    dindirect_idx = bn % NINDIRECT;
    //如果块为空，分配一块。
    if((addr = dindirect[dindirect_idx]) == 0) {
      addr =dindirect[dindirect_idx] = balloc(ip->dev);
      log_write(bp2);
    }
    brelse(bp2);
    brelse(bp);
    return addr;
  }
  panic("bmap: out of range");
```

```
}
```

测试结果如图