**操作系统期末大作业**

学 院：智能工程学院

姓 名：刘德鹏

学 号：18364059

邮 箱：liudp5@mail2.sysu.edu.cn

提交日期：2021/01/24

**Xv6 lab: Multithreading/Uthread: switching between threads**

仿照上下文切换的方式完成进程切换

在trampoline.S中的形式为

首先修改uthread.c中的thread定义如下

struct thread {

  /\*\* My Implementation  \*/

  uint64     ra;

  uint64     sp;

  // callee registers

  /\*\* thread\_switch needs to save/restore only the callee-save registers.   \*/

  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 \*/

};

模仿proc.h中进程上下文的切换，在uthread\_Switch.S中补充线程上下文的切换：

修改uthread\_Switch.S为

.text

/\* Switch from current\_thread to next thread\_thread, and make

\* next\_thread the current\_thread. Use t0 as a temporary register,

\* which should be caller saved. \*/

.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 \*/

修改thread\_create，使之能够记录线程的返回地址ra与栈地址sp。其中，返回地址意味着当切换线程时，线程应该返回到什么哪个地址。这里应该是传入的函数的入口func

void

thread\_create(void (\*func)())

{

  struct thread \*t;

  for (t = all\_thread; t < all\_thread + MAX\_THREAD; t++) {

    if (t->state == FREE) break;

  }

  t->state = RUNNABLE;

  // YOUR CODE HERE

  t->ra = (uint64)func;

  t->sp = (uint64)(t->stack + STACK\_SIZE);

}

最后，在thread\_schedule中添加thread\_switch调用：

void

thread\_schedule(void)

{

  struct thread \*t, \*next\_thread;

  /\* Find another runnable thread. \*/

  next\_thread = 0;

  t = current\_thread + 1;

  for(int i = 0; i < MAX\_THREAD; i++){

    if(t >= all\_thread + MAX\_THREAD)

      t = all\_thread;

    if(t->state == RUNNABLE) {

      next\_thread = t;

      break;

    }

    t = t + 1;

  }

  if (next\_thread == 0) {

    printf("thread\_schedule: no runnable threads\n");

    exit(-1);

  }

  if (current\_thread != next\_thread) {         /\* switch threads?  \*/

    next\_thread->state = RUNNING;

    t = current\_thread;

    current\_thread = next\_thread;

    /\* YOUR CODE HERE

     \* Invoke thread\_switch to switch from t to next\_thread:

     \* thread\_switch(??, ??);

     \*/

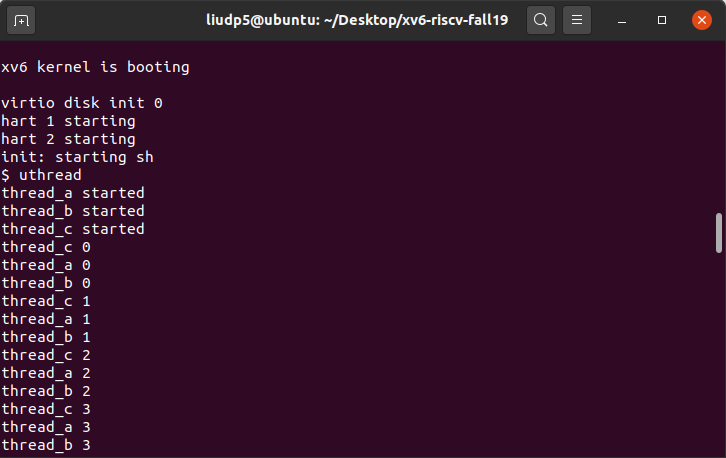
     thread\_switch((uint64)t, (uint64)next\_thread);

  } else

    next\_thread = 0;

}

测试结果





**Xv6 lab: Lock/Memory allocator**

由于有多个CPU

实现一个多free list的初始化

struct kmem{

  struct spinlock lock;

  struct run \*freelist;

} kmem;

struct kmem kmems[NCPU];//多个kmem

void

kinit()//实现多free list的初始化

{

  push\_off();

  int currentid = cpuid();

  pop\_off();

  printf("# cpuId:%d \n",currentid);

  /\* 初始化NCPU个锁\*/

  for (int i = 0; i < NCPU; i++)

  {

    initlock(&kmems[i].lock, "kmem");

  }

  //initlock(&kmem.lock, "kmem");

  freerange(end, (void\*)PHYSTOP);

}

实现插入和弹出，将free list的插入push()和弹出pop()操作封装为函数

struct run\* trypopr(int id){

  struct run \*r;

  r = kmems[id].freelist;

  if(r)

    kmems[id].freelist = r->next;

  return r;

}

void trypushr(int id, struct run\* r){

  if(r){

    r->next = kmems[id].freelist;

    kmems[id].freelist = r;

  }

  else

  {

    panic("cannot push null run");

  }

}

修改kfree()函数，查看空余CPU，用cpuid()获取相应的id，然后将被释放的块插入相应的free list

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 currentid = cpuid();

  pop\_off();

  acquire(&kmems[currentid].lock);

  trypushr(currentid, r);

  release(&kmems[currentid].lock);

}

当一个CPU的free list为空，但是另一个CPU的free list还有空闲块时，我们就应该从有空闲free list的CPU处“偷”一个空闲内存块。

在kalloc()函数中，首先检查当前CPU的free list，如果当前CPU的free list不为空，那就直接pop一个run出来，将之初始化后，返回给调用者；如果当前CPU对应的free list为空，就需要去查询其它CPU对应的free list，找到空闲的块r后，按照如下步骤操作：

1.将r插入自己的free list中；

2.将r从自己的free list中弹出，将之初始化，返回给调用者；

void \*

kalloc(void)

{

  struct run \*r;

  int issteal = 0;/\*\* 标识是否为偷盗 \*/

  push\_off();

  int currentid = cpuid();

  pop\_off();

  acquire(&kmems[currentid].lock);

  r = trypopr(currentid);

  /\*\*

   \* 将id的一块free page卸下，然后给currentid

   \* 这个过程中经历了：

   \*

   \* 1.卸下id的free page

   \* 2.为current id的freelist添加该page

   \* 3.将current id的freelist中的该page卸载掉

   \* 4.返回该page

   \*

   \* 整个过程完成了current id偷盗id的free page的行为

   \*/

  if(!r){

    //printf("oops out of memory\n");

    for (int id = 0; id < NCPU; id++)

    {

      /\* steal first run \*/

      if(id != currentid){

        /\*\* 锁住id的freelist，此时不让其他cpu访问  \*/

        if(kmems[id].freelist){

          acquire(&kmems[id].lock);

          /\*\* 卸下id的free page \*/

          r = trypopr(id);

          /\*\* 为currentid的freelist添加一个run \*/

          trypushr(currentid, r);

          issteal = 1;

          release(&kmems[id].lock);

          break;

        }

      }

      //printf("\n");

    }

  }

  /\*\* 如果是偷盗的，则把currentid的freelist释放出来  \*/

  if(issteal)

    r = trypopr(currentid);

  release(&kmems[currentid].lock);

  if(r){

    //printf("currentid: %d, r: %p\n", currentid, r);

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

  }

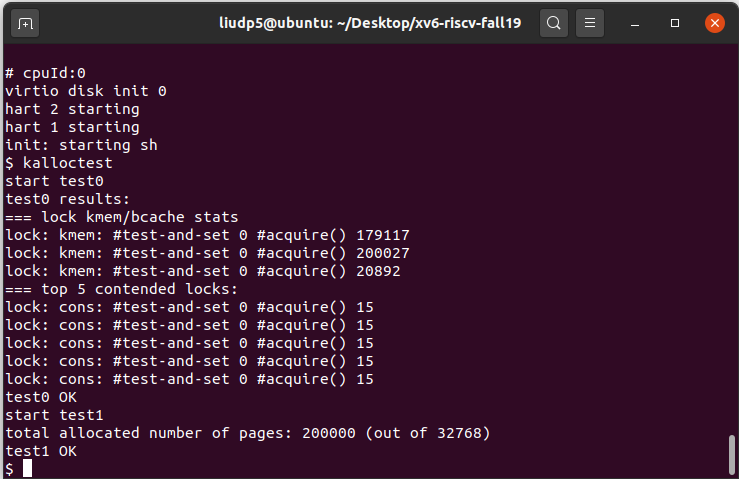
  /\*\* 返回该page  \*/

  //printf("issteal: %d \n", issteal);

  return (void\*)r;

}

测试结果



**Xv6 lab: Lock/Buffer cache**

首先修改bcache，以使其支持哈希桶结构：

struct {

  struct spinlock lock;

  struct buf buf[NBUF];             // block[30]

  /\*\* 实则循环双向链表  \*/

  // Linked list of all buffers, through prev/next.

  // head.next is most recently used.

  //struct buf head;

  struct buf buckets[NBUKETS];

  struct spinlock bucketslock[NBUKETS];

} bcache;

初始化bcache：

void

binit(void)

{

  struct buf \*b;

  /\*\* 在head头插入b  \*/

  initlock(&bcache.lock, "bcache");

  for (int i = 0; i < NBUKETS; i++)

  {

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

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

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

  }

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

  {

    int hash = getHb(b);

    b->time\_stamp = ticks;

    b->next = bcache.buckets[hash].next;

    b->prev = &bcache.buckets[hash];

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

    bcache.buckets[hash].next->prev = b;

    bcache.buckets[hash].next = b;

  }

}

bget()，其实完成这一部分完全就是将原代码改写为支持哈希桶即可。需要注意的是：

在bget函数里，如果找到相应缓冲区的话，就简单返回就行，但如果没找到，就去其他hash桶里偷个来放自己所属的缓冲区里，如果别的hash桶里没有的话就报错

static struct buf\*

bget(uint dev, uint blockno)

{

  int hash = getH(blockno);

  struct buf \*b;

  /\*\*

   \*

   \* My modification

   \*/

  acquire(&bcache.bucketslock[hash]);

  for(b = bcache.buckets[hash].next; b != &bcache.buckets[hash]; b = b->next){

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

      b->time\_stamp = ticks;

      b->refcnt++;

      //printf("## end has \n");

      release(&bcache.bucketslock[hash]);

      acquiresleep(&b->lock);

      return b;

    }

  }

  // If there is no cached buffer for the given sector, bget must make one, possibly reusing a buffer

  // that held a different sector. It scans the buffer list a second time, looking for a buffer that

  // is not in use

  // (b->refcnt = 0); any such buffer can be used. Bget edits the buffer metadata to record the new

  // device and sector number and acquires its sleep-lock. Note that the assignment b->valid = 0

  // ensures that bread will read the block data from disk rather than incorrectly using the buffer’s

  // previous contents.

  // Not cached; recycle an unused buffer.

  /\*\*

   \*

   \* My modification

   \*/

  for (int i = 0; i < NBUKETS; i++)

  {

    if(i != hash){

      acquire(&bcache.bucketslock[i]);

      for(b = bcache.buckets[i].prev; b != &bcache.buckets[i]; b = b->prev){

        if(b->refcnt == 0){

          b->time\_stamp = ticks;

          b->dev = dev;

          b->blockno = blockno;

          b->valid = 0;     //important

          b->refcnt = 1;

          /\*\* 将b脱出  \*/

          b->next->prev = b->prev;

          b->prev->next = b->next;

          /\*\* 将b接入  \*/

          b->next = bcache.buckets[hash].next;

          b->prev = &bcache.buckets[hash];

          bcache.buckets[hash].next->prev = b;

          bcache.buckets[hash].next = b;

          //printf("## end alloc: hash: %d, has: %d\n", hash,i);

          release(&bcache.bucketslock[i]);

          release(&bcache.bucketslock[hash]);

          acquiresleep(&b->lock);

          return b;

        }

      }

      release(&bcache.bucketslock[i]);

    }

  }

  panic("bget: no buffers");

}

实现brelse()，不需要采用任何lock，利用时间戳机制就能解决问题

void

brelse(struct buf \*b)

{

  //printf("#---------------------------------------- brelse! ----------------------------------------\n");

  if(!holdingsleep(&b->lock))

    panic("brelse");

  releasesleep(&b->lock);

  int blockno = getHb(b);

  b->time\_stamp = ticks;

  if(b->time\_stamp == ticks){

    b->refcnt--;

    if(b->refcnt == 0){

      /\*\* 将b脱出  \*/

      b->next->prev = b->prev;

      b->prev->next = b->next;

      /\*\* 将b接入  \*/

      b->next = bcache.buckets[blockno].next;

      b->prev = &bcache.buckets[blockno];

      bcache.buckets[blockno].next->prev = b;

      bcache.buckets[blockno].next = b;

    }

  }

}

测试结果



**Xv6 lab: File System/Large files**

1.为inode添加一个二级索引，使之能够支持分配更大的文件；

2.学会如何创建symbol link软链接

为inode添加一个Doubel Indirect索引块，这样每一个inode就能够支持11 + 256 + 256 \* 256 = 65803个数据块的文件了。修改宏定义如下

/\*\* direct blocks \*/

#define NDIRECT 11

/\*\* indirect blocks \*/

#define SINGLEDIRECT (BSIZE / sizeof(uint))

#define NINDIRECT (SINGLEDIRECT + SINGLEDIRECT \* BSIZE / sizeof(uint))

#define MAXFILE (NDIRECT + NINDIRECT)

其中，NDIRECT代表直接索引指向块的数量；SINGLEDIRECT表示一级索引能够指向的块的数量；NINDIRECT索引代表一级和二级索引一共可指向的块的数量。

修改kernel/param.h中的FSSIZE宏定义为200000

#define FSSIZE       200000  // size of file system in blocks

模仿bmap原有分配块的操作，实现二级索引列表

static uint

bmap(struct inode \*ip, uint bn)

{

  /\*\*

   \* bmap() deals with two kinds of block numbers.

   \* The bn argument is a "logical block number" --

   \* a block number within the file, relative to the start of the file.

   \* The block numbers in ip->addrs[], and the argument to bread(),

   \* are disk block numbers.

   \* You can view bmap() as

   \*

   \*

   \* mapping a file's logical block numbers into disk block numbers.

   \*/

  // printf("-----------------------------------------------\n");

  // printf("bn1: %d\n", bn);

  // NINDIRECT 256

  // printf("NINDIRECT: %d\n", NINDIRECT); //256

  uint addr, \*a, \*a2;

  struct buf \*bp, \*bp2;

  /\*\* bn是相对file（inode）的虚拟编号  \*/

  /\*\* 直接返回块ip->addrs[bn]  \*/

  if(bn < NDIRECT){

    if((addr = ip->addrs[bn]) == 0)

      ip->addrs[bn] = addr = balloc(ip->dev);

    return addr;

  }

  /\*\* 否则减去NDIRENT  \*/

  bn -= NDIRECT;

  //11 + 256 + 256 \* 256

  if(bn < NINDIRECT){

    if(bn < SINGLEDIRECT){

      // Load indirect block, allocating if necessary.

      if((addr = ip->addrs[NDIRECT]) == 0)

        ip->addrs[NDIRECT] = addr = balloc(ip->dev);

      /\*\* Return a locked buf with the contents of the indicated block.  \*/

      bp = bread(ip->dev, addr);

      /\*\*

       \* 为何 #define NINDIRECT (BSIZE / sizeof(uint)) ？

       \* 因为一个buf有BSIZE个data，一个block地址为64位，故刚好存储256个block指针，NINDIRECT = 256

       \*/

      /\*\* 一个文件描述符 \*/

      a = (uint\*)bp->data;

      if((addr = a[bn]) == 0){

        a[bn] = addr = balloc(ip->dev);

        log\_write(bp);

      }

      brelse(bp);

    }

    else

    {

      /\*\*

       \* 实现double-indirect

       \* 结构：

       \* addr -> 256个single-indirect -> 256 \* 256个block

       \* 256 + 11 = 267 0 ~ 266为前面的，267开始后面为后面的

       \*  \*/

      bn -= SINGLEDIRECT;

      /\*\* 宏在做乘除运算时记得加括号5555555555555555555  \*/

      /\*\* single-indirect索引\*/

      int single\_indirect\_index = bn / SINGLEDIRECT;

      /\*\* single-indirect内部相对索引  \*/

      int relative\_offset\_bn = bn % SINGLEDIRECT;

      /\*\* 下标12是double-indirect  \*/

      int pos = NDIRECT + 1;

      if((addr = ip->addrs[pos]) == 0)

        ip->addrs[pos] = addr = balloc(ip->dev);

      bp = bread(ip->dev, addr);

      a = (uint\*)bp->data;

      if((addr = a[single\_indirect\_index]) == 0){

        printf("bn: %d, addr: %p\n",bn, addr);

        a[single\_indirect\_index] = addr = balloc(ip->dev);

        log\_write(bp);

      }

      brelse(bp);

      bp2 = bread(ip->dev, addr);

      a2 = (uint\*)bp2->data;

      if((addr = a2[relative\_offset\_bn]) == 0){

        a2[relative\_offset\_bn] = addr = balloc(ip->dev);

        log\_write(bp2);

      }

      brelse(bp2);

    }

    printf("-----------------------------------------------\n");

    return addr;

  }

  panic("bmap: out of range");

}

将itrunc补充完整，以释放inode下的所有数据块，同样，做法参考原本的itrunc即可

static void

itrunc(struct inode \*ip)

{

  int i, j;

  struct buf \*bp, \*bp2;

  uint \*a, \*a2;

  /\*\* Free掉所有Direct \*/

  for(i = 0; i < NDIRECT; i++){

    if(ip->addrs[i]){

      bfree(ip->dev, ip->addrs[i]);

      ip->addrs[i] = 0;

    }

  }

  /\*\* Free掉single-direct  \*/

  if(ip->addrs[NDIRECT]){

    bp = bread(ip->dev, ip->addrs[NDIRECT]);

    a = (uint\*)bp->data;

    /\*\*

     \* 修改NINDIRECT为SINGLEDIRECT

     \* for(j = 0; j < NINDIRECT; j++){

     \*   if(a[j])

     \*     bfree(ip->dev, a[j]);

     \* }

     \*/

    for (j = 0; j < SINGLEDIRECT; j++)

    {

      /\* code \*/

      if(a[j])

        bfree(ip->dev, a[j]);

    }

    brelse(bp);

    bfree(ip->dev, ip->addrs[NDIRECT]);

    ip->addrs[NDIRECT] = 0;

  }

  /\*\* Free掉double-direct  \*/

  if(ip->addrs[NDIRECT + 1]){

    printf("free double\n");

    int pos = NDIRECT + 1;

    bp = bread(ip->dev, ip->addrs[pos]);

    a = (uint\*)bp->data;

    int number\_of\_single\_direct = BSIZE / sizeof(uint);

    for (i = 0; i < number\_of\_single\_direct; i++)

    {

      if(a[i]){

        bp2 = bread(ip->dev, a[i]);

        a2 = (uint \*)bp2->data;

        for (j = 0; j < SINGLEDIRECT; j++)

        {

          if(a2[j])

            bfree(ip->dev, a2[j]);

        }

        brelse(bp2);

        bfree(ip->dev, a[i]);

        a[i] = 0;

      }

    }

    brelse(bp);

    bfree(ip->dev, ip->addrs[pos]);

    ip->addrs[pos] = 0;

  }

  ip->size = 0;

  iupdate(ip);

}

测试结果

