操作系统期末大作业

18364125 庄鑫康

Lab 1.Multithreading

1.1 为了实现所需结果,文本保存思路可以参考 kernel 中线程切换的操作，在 uthread.c 文件中加入保存文本信息的 s0~s11,(报告中英文注释均为 xv6 源码注释）

```
struct thread {
    /*寄存器*/
    uint64 ra;
    uint64 sp;
    // 保存文本
    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 */


} « end thread » ;
```

仿照 kernel/swtich.S 对 user/uthread_switch.S 进行如下编写，即保存当前 contex，将新的 context 加载。

```
.globl swtch
swtch:
        sd ra, 0(a0)    #uint64 通过八字节保存，将old_context返回地址指针ra，栈顶指针sp放入寄存器a0中
        sd sp, 8(a0)
        sd s0, 16(a0)   #将old_context文本内容s0~s11依次保存寄存器a0中
        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)   #将a1中new_context寄存器内容加载
        ld sp, 8(a1)   #new_context内容包括ra，sp与s0~s11依次加载
        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 #将执行switch前保存的ra设置为pc，新进程从上一条执行switch函数的下一条指令开始运行
```

在进程创建时将新线程寄存器指向栈顶，由于题目要求 thread_schedule()在运行刚才加入的切换代码后，自动开始运行对应函数。在 thread_create 中，将 ra 寄存器的值赋成对应函数的入口地址，同时让每一个线程都将自己的 sp 寄存器指向栈底，由于栈地址从高向低增长，所以 sp 寄存器赋为栈数组的最高地址：

```
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-1];
}
```

在 thread_schedule 中添加对 swtich 函数的调用

```
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
```

最后得到如下图所示输出结果：

```
hart 2 starting
hart 1 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

2. Xv6 lab: Lock/Memory allocator

根据题意，由于最开始的 xv6 中在 kalloctest 中，三个进程不断通过调用 kalloc 和 kfree 来增长或释放它们的内存空间，然而在 kalloc 和 kfree 中只有一个锁 kmem.lock 可以保证这三个进程的访问不冲突，这样一来，导致了大量的无效锁请求。

我们需要为每个 CPU 维护一个空闲列表，每个列表有自己的锁。不同 CPU 上的分配和释放可以并行运行，因为每个 CPU 将在不同的列表上操作。主要的挑战是处理这样的情况:一个 CPU 的空闲列表是空的，而另一个 CPU 的列表有空闲内存;在这种情况下，一个 CPU 必须“窃取”另一个 CPU 的空闲列表的一部分。窃取可能会引入锁争用，但希望这种情况不会经常发生。

而根据题目提示，首先提示了我们使用 NCPU，即：

```
#define NCPU          8  // maximum number of CPUs
```

定义了 CPU 的数量。

同时提示使用 cpuid 获取当前运行 CPU 号，调用 freerange()初始化所有的 free list。

其中 kalloc.C 文件中 freerange()的源码为：

```
void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

freerange()函数通过调用 kfree(),使用头插法将 end~PHYSTOP 之间的内存一页页地加入到 free list 中进行管理，该部分代码如下：

```
  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;  //采用头插法
  kmem.freelist = r;
  release(&kmem.lock);
```

2.1 为每个 CPU 维护一个 free list

加入结构体数组：

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
struct kmem kmems[NCPU];
```

2.2 修改 kfree（）函数

```
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();// 关闭中断
  int i=cpuid();//CPUのID
  acquire(&kmem[i].lock);
  r->next = kmem[i].freelist;
  kmem[i].freelist = r;  //使用头插法
  release(&kmem[i].lock);

  pop_off();//开中断
} « end kfree »
```

2.3 修改 kalloc（）函数,添加判断当前 CPU 的 freelist 不够时“借用”其他的 freelist

```
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)//当前cpu对应freelist为空
  {
    for(int j=0;j<NCPU;j++)//从其他cpu的freelist中借用
    {
      if(j!=i)
      {
        acquire(&kmem[j].lock);
        if(kmem[j].freelist)
      {
        r=kmem[j].freelist;
            kmem[j].freelist=r->next;//获得锁后将其他CPU的freelist接上
        release(&kmem[j].lock);
        break;
      }
        release(&kmem[j].lock);
      }
    }
  }

  pop_off();//turn on inturrupt

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

结果如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 222350
lock: kmem: #fetch-and-add 0 #acquire() 197834
lock: kmem: #fetch-and-add 0 #acquire() 12887
lock: bcache: #fetch-and-add 0 #acquire() 334
--- top 5 contended locks:
lock: proc: #fetch-and-add 39833 #acquire() 148043
lock: proc: #fetch-and-add 9289 #acquire() 148102
lock: proc: #fetch-and-add 5285 #acquire() 148104
lock: virtio_disk: #fetch-and-add 5142 #acquire() 57
lock: pr: #fetch-and-add 1795 #acquire() 5
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$
```

3. Lock/Buffer cache

3.1 参考 bio.c 中 bcache.head 的操作，修改 bcache 结构实现要求的 13 位哈希表

```
struct {
  struct spinlock lock[NBUCKET];
  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.

  //buf结构表头
  struct buf *table[NBUCKET];

} bcache;
```

在 buf.h 中修改 buf 结构,每个 buf 共有 NBUCKET 个哈希表，所有哈希表共享 NBUF 个 buf。然后添加字段 time_stamp，用以标记 buf 的时间戳，如下图所示：

```
struct buf {
  int valid;   // 标记数据是否已经被硬盘读取
  int disk;    // 标记硬盘是否有自己的buf
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU缓存链表
  struct buf *next;
  uchar data[BSIZE];
  uint time_stamp;   //添加记录时间戳

};
void //初始化bache
binit(void)
{
  struct buf *b;

  //上锁
  for(int i=0;i<NBUCKET;i++)
  {
        for(b = bcache[i].buf; b < bcache[i].buf+NBUF; b++){
            initsleeplock(&b->lock, "buffer");
        }
      initlock(&bcache[i].lock, "bcache.bucket");
  }

  if(show) printf("binit\n");
```

为了获取 buf 的 bget 函数中核心代码如下：

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

  //初始化buf、时间戳、块号
  struct buf *b=0;

  int i= idx(blockno);
  uint min_time_stamp=-1;
  struct buf *min_b=0;//最小时间对应buf

  acquire(&bcache[i].lock);//只持有一个锁

  // 该block是否已经有缓存
  for(b = bcache[i].buf; b < bcache[i].buf+NBUF; b++){
        if(b->dev==dev && b->blockno == blockno) //命中
        {
            b->refcnt++;
            release(&bcache[i].lock);
            acquiresleep(&b->lock);
            return b;
        }//找到最小时间对应的buf
        if(b->refcnt==0 && b->time_stamp<min_time_stamp)
        {
       min_time_stamp=b->time_stamp;
       min_b=b;
        }
```

同时将 bio.c 中在源代码基础上对其他获取锁的操作进行修改后，以 bpin 和 bunpin 为例，如下图所示：

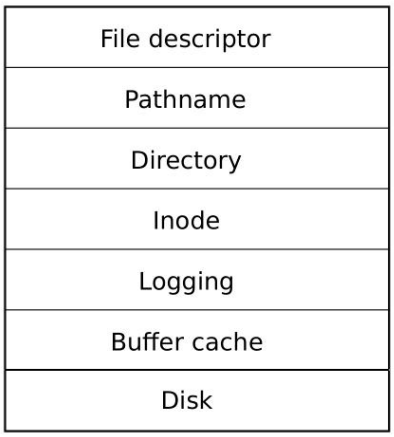
```
void
bpin(struct buf *b) {
  /*
  acquire(&bcache.lock);
  b->refcnt++;
  release(&bcache.lock);
  */

  int i=idx(b->blockno);
  acquire(&bcache[i].lock);
  b->refcnt++;
  release(&bcache[i].lock);


}

void
bunpin(struct buf *b) {
  /*
  acquire(&bcache.lock);
  b->refcnt--;
  release(&bcache.lock);*/

  int i=idx(b->blockno);
  acquire(&bcache[i].lock);
  b->refcnt--;
  release(&bcache[i].lock);
```

便可得到如下实验结果：

```
hart 2 starting
hart 1 starting
init: starting sh
$ bachetest
exec bachetest failed
$ bachetest
exec bachetest failed
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33011
lock: kmem: #fetch-and-add 0 #acquire() 108
lock: kmem: #fetch-and-add 0 #acquire() 52
lock: bcache.bucket: #fetch-and-add 54 #acquire() 8296
lock: bcache.bucket: #fetch-and-add 4 #acquire() 10298
lock: bcache.bucket: #fetch-and-add 9 #acquire() 10442
lock: bcache.bucket: #fetch-and-add 190 #acquire() 8748
lock: bcache.bucket: #fetch-and-add 16 #acquire() 8476
lock: bcache.bucket: #fetch-and-add 60 #acquire() 9794
lock: bcache.bucket: #fetch-and-add 88 #acquire() 8288
--- top 5 contended locks:
lock: proc: #fetch-and-add 196871 #acquire() 72459
lock: proc: #fetch-and-add 82194 #acquire() 72776
lock: virtio_disk: #fetch-and-add 61045 #acquire() 1014
lock: proc: #fetch-and-add 15244 #acquire() 72431
lock: proc: #fetch-and-add 13548 #acquire() 72431
tot= 421
test0: OK
start test1
test1 OK
$
```

4. lab File System/Large files

xv6 中的文件系统结构如下图所示：



disk 在 virtio 硬盘上读写 buffer cache 提供缓存

logging 将来自高层对几个块的更新包装成一个事务（transaction），保证发生崩溃时，块的更新具有原子性（要么都不执行，要么全部执行）

inode 提供单个的文件，表现为 inode，有一个独一无二的 inum，一些块保存该文件的数据

directory 将目录实现为一种特殊的 inode，他的内容是一系列 dirent（directory entry），每个 dirent 包含一个文件名和一个 inum

pathname 提供有层次地路径名，并使用递归地查找来解析他们

file descriptor 将 unix 的各种资源抽象化，使其都可以通过文件描述符访问

而在这个任务中，需要增加 xv6 文件的最大大小。而本来的 xv6 文件被限制为 268 个块，即 268*BSIZE（1024）字节。这个限制来自于这样一个限定：一个 xv6 索引节点包含 12 个直接索引和一个一级间接索引（最多可容纳 256 个块号的块），总共有 12+256=268 个块。我们需要更改 xv6 文件系统代码，来支持每个 inode 中的二级间接索引，其中包含 256 个一级间接索引块的地址，每个块最多可以包含 256 个数据块地址，使一个文件最多可以包含 256256+256+11 个块（11 个而不是 12 个，因为我们将为二级间接索引块牺牲一个直接索引块号）。

4.1 根据题目要求，需要牺牲了一个直接索引，将其变为二级间接索引，我们修改 fs.h 中的宏定义和 dinode 结构体，如下图所示：

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLEINDIRECT (NINDIRECT*NINDIRECT) //二级索引
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLEINDIRECT)
```

将直接索引的数量从原来的 12 改为 11，将其改为二级索引，同时改变 dinode 结构体：

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses NDIRECT+1改为NDIRECT+2
};
```

4.2 根据题目提示，我们同时也将 file.h 中的 date 数组范围改为 NDIRECT+2,

```
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];
};
```

4.3 修改 fs.c 文件中的 bmap()函数。

我们希望增加的二级间接索引，即通过原本的 256 个地址再去寻找一个中间块，再找到最终的数据块，借间接索引的方式来达到存储更大文件的目的。而本来的 bmap()函数中传入的是 bn（logical block number），所以我们需要三个 if 来判断它究竟属于那个类型的索引类型。

可通过 bn 先除再整除的方式求得文件具体位置，然后先去找中间层的间接索引，读出来地址后，再去找第二层间接索引，写下刚读到的索引地址。然后根据这个索引再去找数据块写下数据。

一个 Indirect 元素：前 N 个元素代表 N 个索引，指向 N 个 block 的位置，最后 1 个元素指向 1 个 block，这个 block 本身就是索引，即 1k byte/4 byte=256 个索引，指向另外 256 个 block。

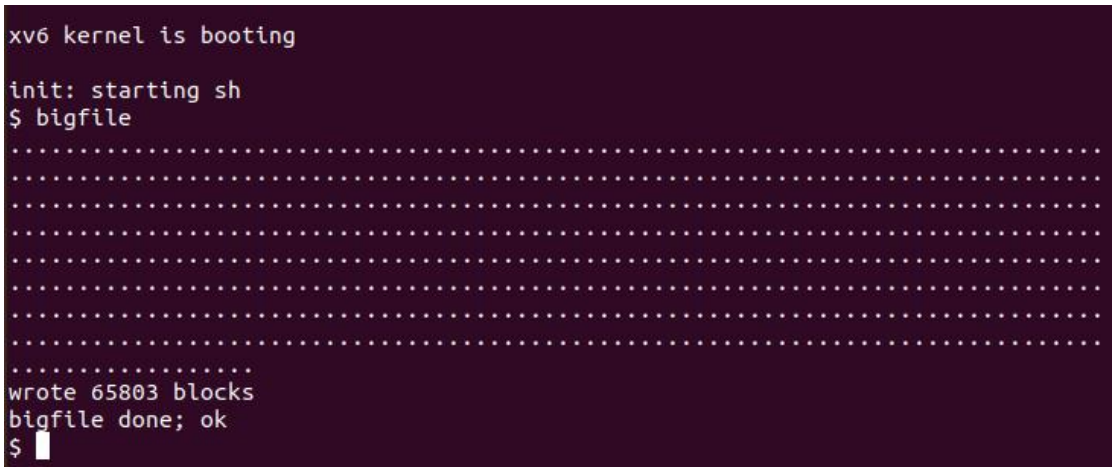
而两个 Indirect 元素：前 N 个元素代表 N 个索引，第 N+1 个同上，第 N+2 个指向一个索引 block，包含 256 个索引，每个索引又指向一个索引 block，包含 256 个 block。总共可以包含 256*256=65536 个 block

部分核心代码如下：

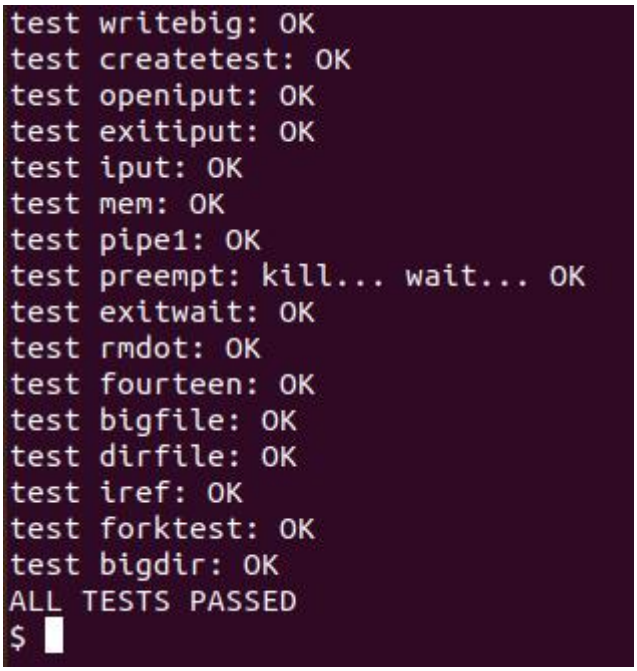
```
if(bn < NINDIRECT_1){
  // 需要调用中间层索引
  if((addr = ip->addrs[NDIRECT]) == 0)
    ip->addrs[NDIRECT] = addr = balloc(ip->dev);//获取数据地址
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  if((addr = a[bn]) == 0){
    a[bn] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp); //释放锁
  return addr;
}

bn -= NINDIRECT_1; //bn=bn-一级索引

if(bn < NINDIRECT_2){
  // 需加载下一级索引
  if((addr = ip->addrs[NDIRECT+1]) == 0)
    ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
  uint bn_1= ( bn & 0xff00)>>8 ;//level1 索引
  uint bn_2= bn & 0xff;//level2 索引
  if((addr = a[bn_1]) == 0){
    a[bn_1] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);
```

同时每次 bread()后都要 bresle()。

在实际实验过程中遇到了很多诡异的问题，包括在 bigfile 测试最后的阶段弹出 bmap out of range 的错误
最终结果如下图所示：



运行 usertest 结果如图：



总结：本次 xv6 实验对 xv6 操作系统的多线程，锁，和文件管理系统都有了更加深刻的了解，但是由于时间的关系很多代码和具体实现相信都有着一定的改良方案，希望借本学期对于 xv6 系统的多次实验探索，对整体计算操作系统的不同功能结构有更深刻的了解与体会，同时为以后的相关学习工作打下一定的基础。