**SYSU 18364012 陈鑫锐**

**OS ASSIGNMENT3**

1. Xv6 lab: Multithreading/Uthread: switching between threads

第一步：在uthread\_switch.S中实现thread\_switch函数，参考kernel/switch.S，保存当前线程的寄存器，恢复即将要切换到线程的寄存器

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)

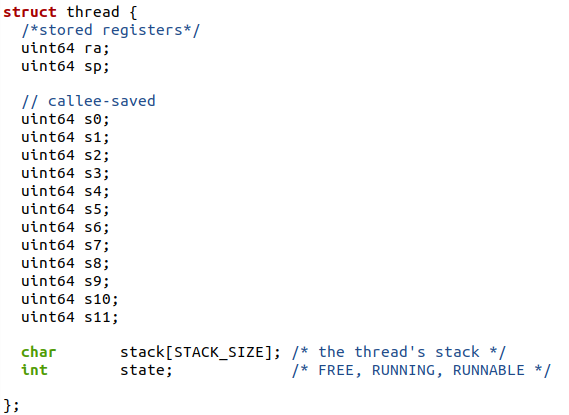
ld s9, 88(a1)

ld s10, 96(a1)

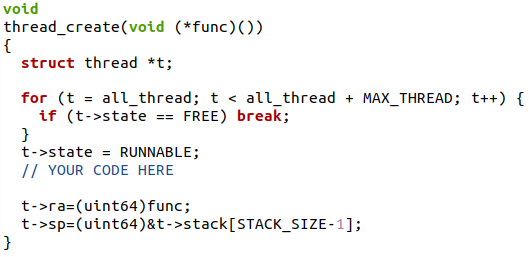
ld s11, 104(a1)

ret

第二步：在uthread.c的struct thread添加寄存器字段



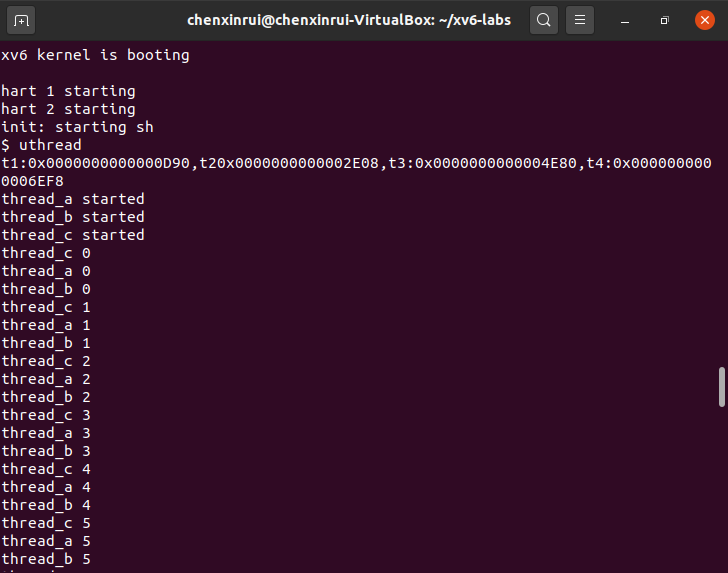
第三步：在thread\_create()中添加保存新线程返回地址和栈指针，返回地址就是输入的函数指针func，栈指针指向struct thread->stack的最后一个元素的地址，因为栈指针是从高地址向低地址增长的

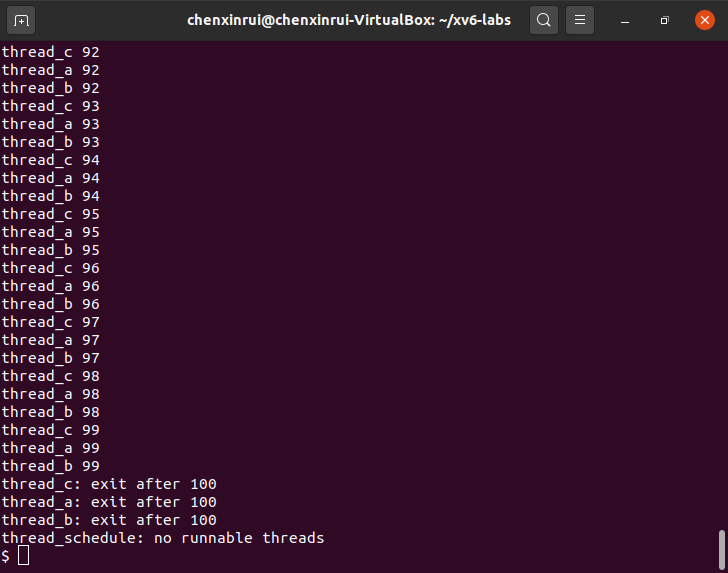


第四步：最后在thread\_schedule()中添加调用

4

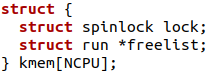
运行结果：



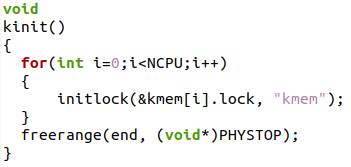


2. Xv6 lab: Lock/Memory allocator

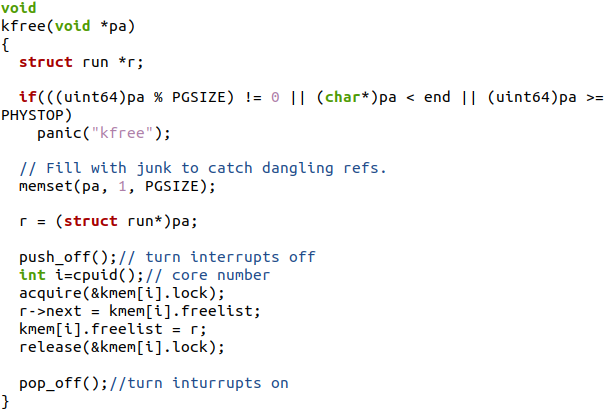
第一步：在kalloc.c中首先将kmem修改为数组，这样每个cpu对应一份freelist和lock



第二步：初始化kmem时将每个cpu对应kmem[i]都初始化



第三步：修改kfree代码



第四步：修改kalloc代码

**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){*//current cpu->freelist is empty*

**for**(**int** j**=**0;j**<**NCPU;j**++**){*//borrow from other cpu->freelist*

**if**(j**!=**i){

acquire(**&**kmem[j].lock);

**if**(kmem[j].freelist){

r**=**kmem[j].freelist;

kmem[j].freelist**=**r**->**next;

release(**&**kmem[j].lock);

**break**;

}

release(**&**kmem[j].lock);

}

}

}

pop\_off();*//turn on inturrupt*

**if**(r)

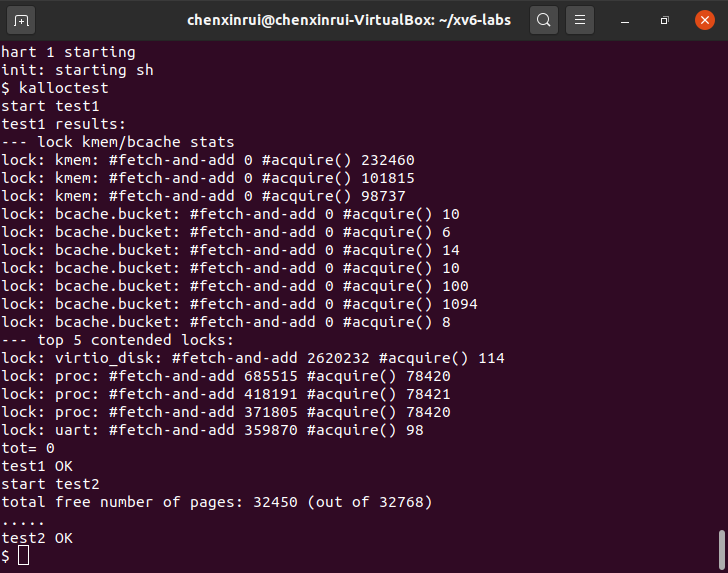
memset((**char\***)r, 5, PGSIZE); *// fill with junk*

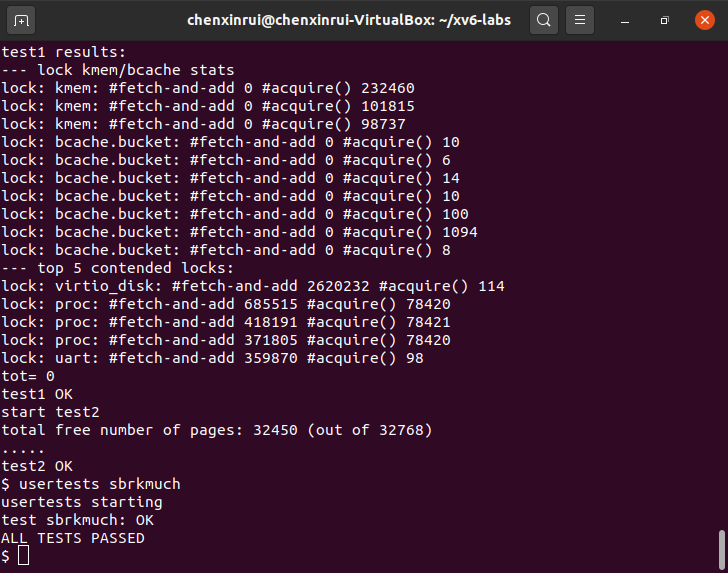
**return** (**void\***)r;

}

运行结果：

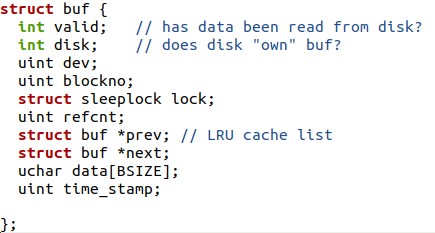
运行kalloctest以查看您的实现是否减少了锁争用，并运行usertests sbrkmuch来检查它是否仍可以分配所有内存。



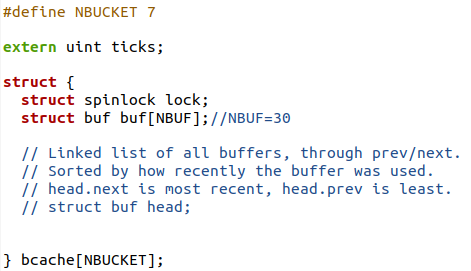


3. Xv6 lab: Lock/Buffer cache

第一步：在kernel/buf.h的struct中添加字段time\_stamp，用以标记buf的时间戳



第二步：bget()中可能出现两种情况：hit(命中)，eviction（驱逐之前的替换新的）。在这里将bcache设计成如下形式，其中NBUCKET需要小一点，因为锁的数目有限，而binit至少需要NBUF\*NBUCKET+NBUCKET个锁



第三步：修改后的kernel/bio.c如下

uint **idx**(uint blockno){

**return** blockno **%** NBUCKET;

}

**void binit**(**void**){

**struct** buf **\***b;

**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");

}

}

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

**struct** buf **\***b**=**0;

**int** i**=** idx(blockno);

uint min\_time\_stamp**=-**1;

**struct** buf **\***min\_b**=**0;

acquire(**&**bcache[i].lock);*//only need to hold one lock to avoid competition*

*// Is the block already cached?*

**for**(b **=** bcache[i].buf; b **<** bcache[i].buf**+**NBUF; b**++**){

**if**(b**->**dev**==**dev **&&** b**->**blockno **==** blockno){*//hit*

b**->**refcnt**++**;

release(**&**bcache[i].lock);

acquiresleep(**&**b**->**lock);

**return** b;

}*//find the buf according to the smallest time\_stamp*

**if**(b**->**refcnt**==**0 **&&** b**->**time\_stamp**<**min\_time\_stamp){

min\_time\_stamp**=**b**->**time\_stamp;

min\_b**=**b;

}

}

b**=**min\_b;

**if**(b**!=**0){

b**->**dev **=** dev;

b**->**blockno **=** blockno;

b**->**valid **=** 0;

b**->**refcnt **=** 1;

release(**&**bcache[i].lock);

acquiresleep(**&**b**->**lock);

**return** b;

}

panic("bget: no buffers");}

**void brelse**(**struct** buf **\***b){

**if**(**!**holdingsleep(**&**b**->**lock))

panic("brelse");

releasesleep(**&**b**->**lock);

**int** i**=**idx(b**->**blockno);

acquire(**&**bcache[i].lock);

b**->**refcnt**--**;

**if**(b**->**refcnt **==** 0){*// no one is waiting for it.*

b**->**time\_stamp**=**ticks;

}

release(**&**bcache[i].lock);}

**void bpin**(**struct** buf **\***b){

**int** i**=**idx(b**->**blockno);

acquire(**&**bcache[i].lock);

b**->**refcnt**++**;

release(**&**bcache[i].lock);

}

**void bunpin**(**struct** buf **\***b) {

**int** i**=**idx(b**->**blockno);

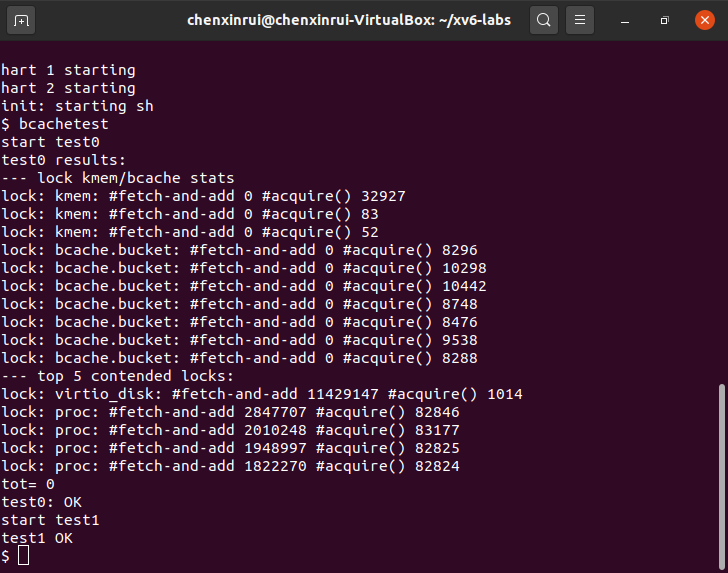
acquire(**&**bcache[i].lock);

b**->**refcnt**--**;

release(**&**bcache[i].lock);

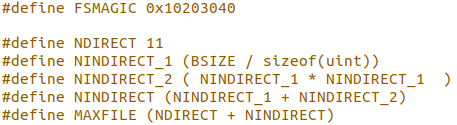
}

运行结果：



4. Xv6 lab: File System/Large files

第一步：首先修改fs.h中的宏定义，NINDIRECT\_1是第一级间接访问的block数目，NINDIRECT\_2是第二级间接访问的block数目



第二步：修改bmap（），使其除了直接块和单间接块之外还实现双间接块。ip->addrs []的前11个元素应该是直接块；第十二个应该是一个间接块（就像当前的块一样）；第13个应该是一个新的双间接块。

**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\_1){

*// Load first-level 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\_1;

**if**(bn **<** NINDIRECT\_2){

*// Load second-level indirect block, allocating if necessary.*

**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*

uint bn\_2**=** bn **&** 0xff;*//level2*

**if**((addr **=** a[bn\_1]) **==** 0){

a[bn\_1] **=** addr **=** balloc(ip**->**dev);

log\_write(bp);

}

brelse(bp);

bp **=** bread(ip**->**dev, addr);

a **=** (uint**\***)bp**->**data;

**if**((addr **=** a[bn\_2]) **==** 0){

a[bn\_2] **=** addr **=** balloc(ip**->**dev);

log\_write(bp);

}

brelse(bp);

**return** addr;

}

panic("bmap: out of range");

}

第三步：确保itrunc释放文件的所有块，包括双间接块。

**void itrunc**(**struct** inode **\***ip){

**int** i, j, k;

**struct** buf **\***bp;

**struct** buf **\***bp2;

uint **\***a;

uint **\***a2;

**for**(i **=** 0; i **<** NDIRECT; i**++**){

**if**(ip**->**addrs[i]){

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

ip**->**addrs[i] **=** 0;

}

}

**if**(ip**->**addrs[NDIRECT]){

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

a **=** (uint**\***)bp**->**data;

**for**(j **=** 0; j **<** NINDIRECT\_1; j**++**){

**if**(a[j])

bfree(ip**->**dev, a[j]);

}

brelse(bp);

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

ip**->**addrs[NDIRECT] **=** 0;

}

**if**(ip**->**addrs[NDIRECT**+**1]){

bp **=** bread(ip**->**dev, ip**->**addrs[NDIRECT**+**1]);

a **=** (uint**\***)bp**->**data;

**for**(j **=** 0; j **<** NINDIRECT\_1; j**++**){

**if**(a[j]){

bp2 **=** bread(ip**->**dev, a[j]);

a2 **=** (uint**\***)bp2**->**data;

**for**(k**=**0 ;k **<** NINDIRECT\_1 ;k**++**){

**if**(a2[k]) bfree(ip**->**dev,a2[k]);

}

brelse(bp2);

bfree(ip**->**dev, a[j]);

}

}

brelse(bp);

bfree(ip**->**dev, ip**->**addrs[NDIRECT**+**1]);

ip**->**addrs[NDIRECT**+**1] **=** 0;

}

ip**->**size **=** 0;

iupdate(ip);

}

运行结果：

