操作系统大作业 3 实验报告

李煜 18364053

本次大作业参考了 xv6 实验手册的实验指引完成。

https://pdos.csail.mit.edu/6.828/2020/labs/thread.html
https://pdos.csail.mit.edu/6.828/2020/labs/lock.html
https://pdos.csail.mit.edu/6.828/2020/labs/fs.html

1.Xv6 lab: Multithreading/Uthread: switching between threads, 30 分

> Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

实验 1 的任务是让我们修改 uthread.c 的代码，使得它能够支持进程之间的切换。首先我们要完成 thread_create.c 来正确创建线程，然后要在 uthread_Switch.S 中实现进程的切换。

第一题我们可以通过模仿上下文切换方式来完成线程的上下文切换。

实验开始前先切换到 uthread 分支。

(1)首先修改 uthread.c 中 thread 的定义:

```
14 struct thread {
15
16
17   uint64     ra;
18   uint64     sp;
19   // callee registers
20   uint64     s0;
21   uint64     s1;
22   uint64     s2;
23   uint64     s3;
24   uint64     s4;
25   uint64     s5;
26   uint64     s6;
27   uint64     s7;
28   uint64     s8;
29   uint64     s9;
30   uint64     s10;
31   uint64     s11;
32
33   char       stack[STACK_SIZE]; /* the thread's stack */
34   int        state;             /* FREE, RUNNING, RUNNABLE */
35
36 };
```

(2)然后模仿上下文切换，在 uthread_Switch.S 中补充线程上下文的切换:

```
        .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 */
```

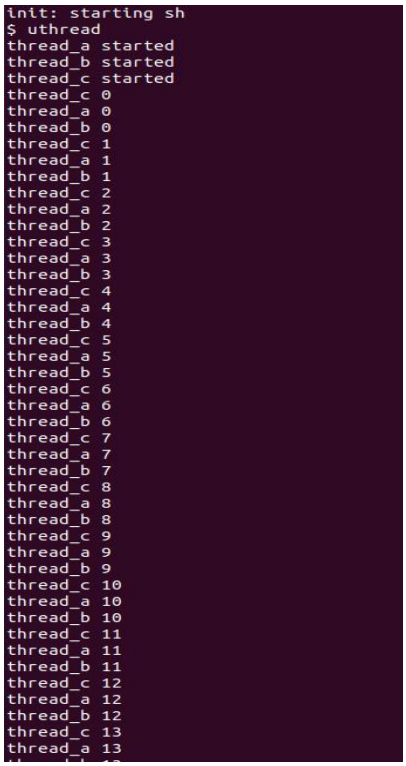
(3)接着修改 thread_create 让它能够记录线程的返回地址 ra 和栈地址 sp:

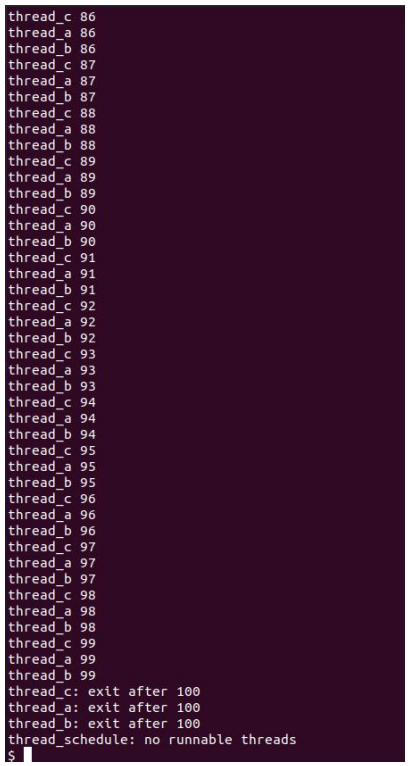
```
void
thread_create(void (*func)())
{
  struct thread *t;

  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);
}

```

(4)最后在 thread_schedule 中添加上 thread_switch 的调用:

```
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;
}
```

运行结果如下：





2.Xv6 lab: Lock/Memory allocator，20 分

> Your job is to implement per-CPU freelists, and stealing when a CPU's free list is empty. You must give all of your locks names that start with "kmem". That is, you should call `initlock` for each of your locks, and pass a name that starts with "kmem". Run kalloctest to see if your implementation has reduced lock contention. To check that it can still allocate all of memory, run `usertests sbrkmuch`. Your output will look similar to that shown below, with much-reduced contention in total on kmem locks, although the specific numbers will differ. Make sure all tests in `usertests` pass. `make grade` should say that the kalloctests pass.

实验 2 的任务是让我们解决多锁 contention 问题，官方给出的解决方案如下：

The root cause of lock contention in kalloctest is that kalloc() has a single free list, protected by a single lock. To remove lock contention, you will have to redesign the memory allocator to avoid a single lock and list. The basic idea is to maintain a free list per CPU, each list with its own lock. Allocations and frees on different CPUs can run in parallel, because each CPU will operate on a different list. The main challenge will be to deal with the case in which one CPU's free list is empty, but another CPU's list has free memory; in that case, the one CPU must "steal" part of the other CPU's free list. Stealing may introduce lock contention, but that will hopefully be infrequent.

实验开始前先切换到 lock 分支

根据实验指导给出的 hints:

Some hints:

- You can use the constant `NCPU` from kernel/param.h
- Let `freerange` give all free memory to the CPU running `freerange`.
- The function `cpuid` returns the current core number, but it's only safe to call it and use its result when interrupts are turned off. You should use `push_off()` and `pop_off()` to turn interrupts off and on.
- Have a look at the `snprintf` function in kernel/sprintf.c for string formatting ideas. It is OK to just name all locks "kmem" though.

(1)首先实现一个多 free list 的初始化:

```
26 struct kmem kmems[NCPU];
27
28 void
29 kinit()
30 {
31   /* 为每个CPU分配Kmem*/
32   push_off();
33   int currentid = cpuid();
34   pop_off();
35   printf("# cpuId:%d \n",currentid);
36   /* 初始化NCPU个lock */
37    for (int i = 0; i < NCPU; i++)
38   {
39     initlock(&kmems[i].lock, "kmem");
40   }
41   freerange(end, (void*)PHYSTOP);
42   printf("# kinit end:%d \n",currentid);
43 }
44
```

(2)由于 kalloc.c 中对于 free list 的操作是头插法，因此我们可以将 free list 的 push()和 pop()封装成函数以便调用，同时我们留意到，由于我们有多个 kmem，因此我们要用 kmems[id].freelist 获取对应的 free list:

```
59 struct run* trypopr(int id){
60   struct run *r;
61   r = kmems[id].freelist;
62   if(r)
63     kmems[id].freelist = r->next;
64   return r;
65 }
66
67 void trypushr(int id, struct run* r){
68   if(r){
69     r->next = kmems[id].freelist;
70     kmems[id].freelist = r;
71   }
72   else
73   {
74     panic("cannot push null run");
75   }
76 }
```

(3)接着我们完成 kfree(), 调用 cpuid()获取对应的 id, 然后将被释放的块插入相应 free list

```
78 void
79 kfree(void *pa)
80 {
81   struct run *r;
82
83   if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
84     panic("kfree");
85
86   // Fill with junk to catch dangling refs.
87   memset(pa, 1, PGSIZE);
88
89   r = (struct run*)pa;
90
91   push_off();
92   int currentid = cpuid();
93   pop_off();
94
95   acquire(&kmems[currentid].lock);
96   trypushr(currentid, r);
97   release(&kmems[currentid].lock);
98 }
99
```

(4)最后，我们要完成 kalloc()。首先，当前 cpu 的 free list 不为空时，直接 pop 一个 run，初始化后返还给调用者；如果当前 cpu 对应的 free list 为空，直接去查询其他 cpu 对应的 free list，找到空闲块 r 后将 r 插入自己的 free list 中，将 r 从自己的 free list 中弹出，初始化后返回给调用者:

```
103 void *
104 kalloc(void)
105 {
106   struct run *r;
107   int issteal = 0;/*标记是否为偷盗 */
108   push_off();
109   int currentid = cpuid();
110   pop_off();
111
112   acquire(&kmems[currentid].lock);
113
114   r = trypopr(currentid);
115
116   if(!r){
117     for (int id = 0; id < NCPU; id++)
118     {
119       if(id != currentid){
120         /*锁住id的freelist，不让其他cpu访问*/
121         if(kmems[id].freelist){
122           acquire(&kmems[id].lock);
123           /*卸下id的free page*/
124           r = trypopr(id);
125           /*为currentid的freelist添加一个run*/
126           trypushr(currentid, r);
127           issteal = 1;
128           release(&kmems[id].lock);
129           break;
130         }
131       }
132     }
133   }
134   /*如果是偷盗的，释放currentid的freelist*/
135   if(issteal)
136     r = trypopr(currentid);
137
138   release(&kmems[currentid].lock);
139   if(r){
140     memset((char*)r, 5, PGSIZE); // fill with junk
141   }
142   /* 返回该page  */
143   return (void*)r;
144 }
```

测试结果如下：

```
# cpuId:0
# kinit end:0
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() 176137
lock: kmem: #fetch-and-add 0 #acquire() 139867
lock: kmem: #fetch-and-add 0 #acquire() 117036
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 5594850 #acquire() 114
lock: proc: #fetch-and-add 1470820 #acquire() 136458
lock: proc: #fetch-and-add 835496 #acquire() 136458
lock: proc: #fetch-and-add 798453 #acquire() 136458
lock: proc: #fetch-and-add 784503 #acquire() 136458
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
```

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

```
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6283
            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
$
```

3.Xv6 lab: Lock/Buffer cache，30 分

> Modify the block cache so that the number of `acquire` loop iterations for all locks in the bcache is close to zero when running `bcachetest`. Ideally the sum of the counts for all locks involved in the block cache should be zero, but it's OK if the sum is less than 500. Modify `bget` and `brelse` so that concurrent lookups and releases for different blocks that are in the bcache are unlikely to conflict on locks (e.g., don't all have to wait for `bcache.lock`). You must maintain the invariant that at most one copy of each block is cached. When you are done, your output should be similar to that shown below (though not identical). Make sure usertests still passes. `make grade` should pass all tests when you are done.

第三个问题与第二个问题有一些类似，当多个进程大量读写磁盘，而磁盘只有一个 lock，这就导致多个进程竞争激烈，实验 3 就是为了解决这个问题。根据提示，我们要修改 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。我们要实现一个大小为 13 的哈希桶。

(1)首先修改 bcache.head，以使其支持哈希桶结构:

```
25
26 struct {
27   struct spinlock lock;
28   struct buf buf[NBUF];
29   /*循环双向链表  */
30   // Linked list of all buffers, through prev/next.
31   struct buf buckets[NBUKETS];
32   struct spinlock bucketslock[NBUKETS];
33
34 } bcache;
```

(2)然后初始化 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;
  }
}
```

(3)然后完成 bget()，使其支持哈希桶

```
static struct buf*
bget(uint dev, uint blockno)
{
  int hash = getH(blockno);
  struct buf *b;
  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;
    }
  }

  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;

          /*脱出b*/
          b->next->prev = b->prev;
          b->prev->next = b->next;

          /*接入b*/
          b->next = bcache.buckets[hash].next;
          b->prev = &bcache.buckets[hash];
          bcache.buckets[hash].next->prev = b;
          bcache.buckets[hash].next = b;

          release(&bcache.bucketslock[i]);
          release(&bcache.bucketslock[hash]);
          acquiresleep(&b->lock);
          return b;
        }
      }
    }
```

(4)接下来我们要实现 brelse()，我们利用时间戳机制来解决这个问题

```
141 void
142 brelse(struct buf *b)
143 {
144   if(!holdingsleep(&b->lock))
145     panic("brelse");
146
147   releasesleep(&b->lock);
148
149   int blockno = getHb(b);
150   b->time_stamp = ticks;
151   if(b->time_stamp == ticks){
152     b->refcnt--;
153     if(b->refcnt == 0){
154       /*脱出b*/
155       b->next->prev = b->prev;
156       b->prev->next = b->next;
157
158       /*接入b*/
159       b->next = bcache.buckets[blockno].next;
160       b->prev = &bcache.buckets[blockno];
161       bcache.buckets[blockno].next->prev = b;
162       bcache.buckets[blockno].next = b;
163     }
164   }
165 }
```

运行结果如下：

```
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33020
lock: kmem: #fetch-and-add 0 #acquire() 73
lock: kmem: #fetch-and-add 0 #acquire() 66
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6209
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6183
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6337
lock: bcache_bucket: #fetch-and-add 0 #acquire() 6337
lock: bcache_bucket: #fetch-and-add 0 #acquire() 7398
lock: bcache_bucket: #fetch-and-add 0 #acquire() 7220
lock: bcache_bucket: #fetch-and-add 0 #acquire() 5525
lock: bcache_bucket: #fetch-and-add 0 #acquire() 5246
lock: bcache_bucket: #fetch-and-add 0 #acquire() 5543
lock: bcache_bucket: #fetch-and-add 0 #acquire() 5236
lock: bcache_bucket: #fetch-and-add 0 #acquire() 3485
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4975
lock: bcache_bucket: #fetch-and-add 0 #acquire() 4201
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 26380341 #acquire() 1368
lock: proc: #fetch-and-add 2283514 #acquire() 91596
lock: proc: #fetch-and-add 2202288 #acquire() 91595
lock: proc: #fetch-and-add 2198544 #acquire() 91596
lock: proc: #fetch-and-add 2144763 #acquire() 91594
tot= 0
test0: OK
start test1
test1
$
```

Show Applications

4.Xv6 lab: File System/Large files，20 分

In this assignment you'll increase the maximum size of an xv6 file. Currently xv6 files are limited to 268 blocks, or 268*BSIZE bytes (BSIZE is 1024 in xv6). This limit comes from the fact that an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks.

```
xv6 kernel is booting

# cpuId:0
# kinit end:0
init: starting sh
$ big file
exec big failed
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
$
```

The test fails because `bigfile` expects to be able to create a file with 65803 blocks, but unmodified xv6 limits files to 268 blocks.

You'll change the xv6 file system code to support a "doubly-indirect" block in each inode, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks. The result will be that a file will be able to consist of up to 65803 blocks, or 256*256+256+11 blocks (11 instead of 12, because we will sacrifice one of the direct block numbers for the double-indirect block).

Modify `bmap()` so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block. You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode. The first 11 elements of `ip->addrs[]` should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block. You are done with this exercise when `bigfile` writes 65803 blocks and `usertests` runs successfully:

首先切换分支到 filesystem

实验 4 要求我们增加 xv6 文件的最大大小。

(1) 修改 fs.c 文件中的 dinode 结构，去掉一个直接索引将其改为二级间接索引

```
25 #define FSMAGIC 0x10203040
26
27 #define NDIRECT 11
28 #define NINDIRECT (BSIZE / sizeof(uint))
29 #define NDOUBLE_INDIRECT(NINDIRECT * NINDIRECT)
30 #define MAXFILE (NDIRECT + NINDIRECT + NDOUBLE_INDIRECT)
31
32 // On-disk inode structure
33 struct dinode {
34   short type;           // File type
35   short major;          // Major device number (T_DEVICE only)
36   short minor;          // Minor device number (T_DEVICE only)
37   short nlink;          // Number of links to inode in file system
38   uint size;            // Size of file (bytes)
39   uint addrs[NDIRECT+2];   // Data block addresses
40 };
```

(2)同时 fs.h 文件中的定义也要修改

```
16 // in-memory copy of an inode
17 struct inode {
18   uint dev;           // Device number
19   uint inum;          // Inode number
20   int ref;            // Reference count
21   struct sleeplock lock; // protects everything below here
22   int valid;          // inode has been read from disk?
23
24   short type;         // copy of disk inode
25   short major;
26   short minor;
27   short nlink;
28   uint size;
29   uint addrs[NDIRECT+2];
30 };
```

(3)修改 bmap()，使其不仅能够直接和单间结块还能和双间接块。分析并模仿直接索引和一级间接索引的程序，写下二级间接索引的程序。

```
401
402   bn -= NINDIRECT;
403   if (bn < NDOUBLE_INDIRECT) {
404     uint bn_level_1 = bn / NINDIRECT;
405     uint bn_level_2 = bn % NINDIRECT;
406
407     // Load first indirect block, allocating if necessary.
408     if((addr = ip->addrs[NDIRECT + 1]) == 0)
409       ip->addrs[NDIRECT + 1] = addr =balloc(ip->dev);
410
411     bp = bread(ip->dev, addr);
412     a = (uint *) bp->data;
413     //load second indirect block, allocating if necessary.
414     if ((addr = a[bn_level_1]) == 0) {
415       a[bn_level_1] = addr = balloc(ip->dev);
416       log_write(bp);
417     }
418     brelse(bp);
419
420     bp = bread(ip->dev, addr);
421     a = (uint *) bp->data;
422     //load data block, allocating if necessary
423     if ((addr = a[bn_level_2]) == 0) {
424       a[bn_level_2] = addr = balloc(ip->dev);
425       log_write(bp);
426     }
427     brelse(bp);
428     return addr;
429   }
430   panic("bmap: out of range");
431 }
```

结果如下：

```
xv6 kernel is booting

virtio disk init 0
init: starting sh
$ bigfile
..........................................................
..........................................................
..........................................................
..........................................................
..........................................................
..........................................................
..........................................................
..........................................................
..................
wrote 65803 blocks
bigfile done; ok
```

```
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

最后，感谢谭老师与助教师兄师姐们一学期的辛苦付出，提前祝老师和师兄师姐们新春快乐！