中山大學

操作系统作业

姓名：晋军

学号：18364037

学院：智能工程学院

第一题：Multithreading/Uthread: switching between threads

作业要求：实现用户的线程切换

实现思路：像内核线程切换一样，给线程添加上下文信息用于保存状态，然后在 thread_schedule 实现交换上下文的功能。

实验过程：

1 修改在 user/uthread.c 中修改上下文结构，使其能够保存上下文内容，与 kernel 进程切换的结构类似。

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context  context;      /* 线程上下文 */
};
```

2 将交换函数添加到 thread_schedule 中，这里与 kernel 中的 scheduler（）类似，旧线程为 t，新线程为 next_thread

```
if (current_thread != next_thread) {
    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->context, (uint64)&current_thread->context);
  } else
    next_thread = 0;
```

3 在 uthread_switch.S 中具体实现 thread_switch,这里与 kernel 中的 swithc.S 类似

```
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 */
```

4 创建 thread_create 函数。thread_schedule()在运行切换代码后，希望自动开始运行对应函数，所以在初始化线程的时候，将 ra 寄存器的值赋成对应函数的入口地址，这样在切换结束后运行汇编代码中的 ret 就自动跳转到 ra 指向的位置了；另一个需要初始化的是 sp 寄存器，因为每个线程都各自有一个栈，所以要让自己的 sp 寄存器指向自己的栈底，

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

运行结果

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

第二题：Lock/Memory allocator

题目要求：要求给物理内存分配程序重新设计锁，使得等待锁时的阻塞尽量少。

实现思路：xv6 上只有一个内存链表供应多个 cpu 使用，因此想要实现目标，就希望让每个 cpu 拥有自己的内存链表

实验过程:

1 为每个 cpu 分配内存链表

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

2 修改 kinit()和 kfree()，获取和释放对应 cpu 的锁，在获取 cpu id 的时候关闭中断，防止线程切换

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

```c
void
kfree(void *pa){
////
        push_off();
          int id = cpuid();
          pop_off();

          acquire(&kmem[id].lock);
          r->next = kmem[id].freelist;
          kmem[id].freelist = r;
          release(&kmem[id].lock);push_off();
          int id = cpuid();
          pop_off();

          acquire(&kmem[id].lock);
          r->next = kmem[id].freelist;
          kmem[id].freelist = r;
          release(&kmem[id].lock);

}
```

3 最后在添加偷取其他 cpu 空闲内存链表的功能

```c
void *
kalloc(void)
{
  struct run *r;

  push_off();
  int id = cpuid();
  pop_off();

  acquire(&kmem[id].lock);
  r = kmem[id].freelist;
  if(r)
    kmem[id].freelist = r->next;
  release(&kmem[id].lock);

  // steal memory
  if (!r)
  {
    for (int i = 0; i < NCPU; i++)
    {
      if (i == id) 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);
    }
  }

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

运行结果

```
lock: proc: #fetch-and-add 24315 #acquire() 165760
lock: proc: #fetch-and-add 8715 #acquire() 165830
lock: virtio_disk: #fetch-and-add 8508 #acquire() 75
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
$
```

第三题：Lock/Buffer cache

题目要求：缓解磁盘缓冲区锁竞争情况

实现思路：使用 LRU+hash table 的方法

实验过程：

1 使用 13 作为 hash table 的大小，将 buf 分段映射

```c
struct {
  struct spinlock lock;
  struct buf buf[NBUF];              // block[30]
  /** 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;
```

2 修改和 head 有关的 binit，bget，brelse，bpin，bunpin 函数

```
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;
  }
}
```

```
void
bpin(struct buf *b) {
  int id = hash(b->blockno);
  acquire(&bcache.lock[id]);
  b->refcnt++;
  release(&bcache.lock[id]);
}

void
bunpin(struct buf *b) {
  int id = hash(b->blockno);
  acquire(&bcache.lock[id]);
  b->refcnt--;
  release(&bcache.lock[id]);
}
```

3 bget 函数里，如果找到相应缓冲区的话，就返回，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

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

  int id = hash(blockno);

  acquire(&bcache.lock[id]);

  // Is the block already cached?
  for(b = bcache.head[id].next; b != &bcache.head[id]; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock[id]);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  for (int j = hash(blockno+1); j != id; j = (j + 1) % NBUCKETS)
  {
    acquire(&bcache.lock[j]);
    for(b = bcache.head[j].prev; b != &bcache.head[j]; b = b->prev){
      if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        // 将别的hash桶里缓冲区放入当前id的缓冲区中
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.lock[j]);
        b->next = bcache.head[id].next;
        b->prev = &bcache.head[id];
        bcache.head[id].next->prev = b;
        bcache.head[id].next = b;
        release(&bcache.lock[id]);
        acquiresleep(&b->lock);
        return b;
      }
    }
    release(&bcache.lock[j]);
  }
  panic("bget: no buffers");
}
```

4 实验结果

```
lock: bcache: #fetch-and-add 0 #acquire() 6749
lock: bcache: #fetch-and-add 0 #acquire() 6711
lock: bcache: #fetch-and-add 0 #acquire() 7911
lock: bcache: #fetch-and-add 0 #acquire() 6213
lock: bcache: #fetch-and-add 0 #acquire() 6212
lock: bcache: #fetch-and-add 0 #acquire() 4158
lock: bcache: #fetch-and-add 0 #acquire() 4155
--- top 5 contended locks:
lock: proc: #fetch-and-add 18128630 #acquire() 4217754
lock: proc: #fetch-and-add 2751180 #acquire() 4210304
lock: proc: #fetch-and-add 562367 #acquire() 4210499
lock: proc: #fetch-and-add 353086 #acquire() 4210305
lock: virtio_disk: #fetch-and-add 286571 #acquire() 1319
tot= 0
test0: OK
start test1
test1 OK
$
```

第四题：Lock/Buffer cache

题目要求：添加大文件系统

实验思路：更改索引，即增加一级索引，由原来的一级索引增加为二级索引。

实验流程：

1 将 FSSIZE 更改为 20000

```
#define FSSIZE       20000  // size of file system in blocks
#define MAXPATH      128   // maximum file path name
```

2 修改宏定义以及 dinode 结构体，使其适配二级索引

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT+NDOUBLE_INDIRECT)

// On-disk inode structure
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
};
```

3 inode 也需要修改

```
// in-memory copy of an inode
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 修改 bmap 函数，主要是将第一次索引的值作为第二次索引的目标，即重复一次索引。

```c
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 < NDOUBLE_INDIRECT){
        uint bn_level_1=bn/NINDIRECT;
        uint bn_level_2=bn%NINDIRECT;
    // Load level-1 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;
        //load level-2 indirect block
    if((addr = a[bn_level_1]) == 0){
      a[bn_level_1] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

    bp2 = bread(ip->dev, addr);
    a2 = (uint*)bp->data;
        //load data block
    if((addr = a[bn_level_2]) == 0){
      a[bn_level_2] = addr = balloc(ip->dev);
      log_write(bp2);
    }
    brelse(bp2);

    return addr;
  }

  panic("bmap: out of range");
}
```

5 实验结果

```
$ bigfile
...................................
...................................
...................................
...................................
...................................
...................................
...................................
...................................
...........................
wrote 65803 blocks
bigfile done; ok
$
```