OS lab4 : multithreading&locks 报告

吕佳鸿 10235501436

multithreading

Uthread: switching between threads

这个实验的要求是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。

因为是用户级线程，不需要设计用户栈和内核栈，用户页表和内核页表等等切换，所以本实验中只需要一个类似于`context`的结构，而不需要费尽心机的维护`trapframe`

所以首先定义一下context

```
// 用户线程的上下文结构体
struct thread_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;
};
```

修改`thread`结构体，添加`context`字段

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

参考`switch.S`在`unthread_switc.S`里添加：

```
    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_scheduler`，添加线程切换语句。thread_schedule() 函数负责进行用户多线程间的调度. 此处是通过函数的主动调用进行的线程切换. 其主要工作就是从当前线程在线程数组的位置开始寻找一个 RUNNABLE 状态的线程进行运行. 所以要将线程进行切换

```
  thread_switch((uint64)&t->context, (uint64)&current_thread->context);
```

在`thread_create`中对`thread`结构体做一些初始化设定，修改`ra`返回地址和`sp`栈指针， 在线程初始化进行设置，这样在后续调度切换时便能保持其正确性

```
t->context.ra = (uint64)func;                  // 设定函数返回地址
t->context.sp = (uint64)t->stack + STACK_SIZE;  // 设定栈指针
```

Using threads

来看一下程序的运行过程：设定了五个散列桶，根据键除以5的余数决定插入到哪一个散列桶中，插入方法是头插法

首先为什么会造成数据丢失：

假设现在有两个线程T1和T2，两个线程都走到put函数，且假设两个线程中key%NBUCKET相等，即要插入同一个散列桶中。两个线程同时调用insert(key, value, &table[i], table[i])，insert是通过头插法实现的。如果先insert的线程还未返回另一个线程就开始insert，那么前面的数据会被覆盖

因此引入锁变量pthread_mutex_t，在尝试写入的时候上锁，通过避免并发put()在哈希表中读取或写入的内存重叠来提高并行速度, 可以考虑哈希表的每个bucket加锁。

为每个散列桶定义一个锁，将五个锁放在一个数组中，并进行初始化

```
pthread_mutex_t lock[NBUCKET] = { PTHREAD_MUTEX_INITIALIZER }; // 每个散列桶一把锁
```

在`put`函数中对`insert`上锁

```
if(e){
    // update the existing key.
    e->value = value;
} else {
    pthread_mutex_lock(&lock[i]);
    // the new is new.
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&lock[i]);
}
```

Barrier

利用pthread提供的条件变量方法，实现同步屏障机制

此处主要涉及互斥锁和条件变量配合达到线程同步.首先条件变量的操作需要在互斥锁锁定的临界区内.然后进行条件判断, 此处即判断是否所有的线程都进入了 barrier() 函数, 若不满足则使用 pthread_cond_wait() 将当前线程休眠, 等待唤醒; 若全部线程都已进入 barrier() 函数, 则最后进入的线程会调用 pthread_cond_broadcast() 唤醒其他由条件变量休眠的线程继续运行. 需要注意的是, 对于 pthread_cond_wait() 涉及三个操作: 原子的释放拥有的锁并阻塞当前线程, 这两个操作是原子的; 第三个操作是由条件变量唤醒后会再次获取锁

线程进入同步屏障barrier时，将已进入屏障的线程数量加1，然后再判断是否已经达到总线程数：如果未达到，则进入睡眠，等待其他线程；如果已经达到，则唤醒所有在barrier中等待的线程，所有线程继续执行；屏障轮数+1；

```
static void
barrier()
{
  // 申请持有锁
  pthread_mutex_lock(&bstate.barrier_mutex);

  bstate.nthread++;
  if(bstate.nthread == nthread) {
```

```
    // 所有线程已到达
    bstate.round++;
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  } else {
    // 等待其他线程
    // 调用pthread_cond_wait时，mutex必须已经持有
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  // 释放锁
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

Result

```
$ make qemu-gdb
uthread: OK (4.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/parallels/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/parallels/xv6-labs-2020'
ph_safe: OK (5.0s)
== Test ph_fast == make[1]: Entering directory '/home/parallels/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/parallels/xv6-labs-2020'
ph_fast: OK (12.3s)
== Test barrier == make[1]: Entering directory '/home/parallels/xv6-labs-2020'
make[1]: 'barrier' is up to date.
make[1]: Leaving directory '/home/parallels/xv6-labs-2020'
barrier: OK (12.3s)
== Test time ==
time: OK
Score: 60/60
```

Locks

Memory allocator

程序`user/kalloctest.c`强调了xv6的内存分配器：三个进程增长和缩小地址空间，导致对`kalloc`和`kfree`的多次调用。`kalloc`和`kfree`获得`kmem.lock`。`kalloctest`打印（作为“#fetch-and-add”）在`acquire`中由于尝试获取另一个内核已经持有的锁而进行的循环迭代次数，如`kmem`锁和一些其他锁。`acquire`中的循环迭代次数是锁争用的粗略度量。

`acquire`为每个锁维护要获取该锁的`acquire`调用计数，以及`acquire`中循环尝试但未能设置锁的次数。`kalloctest`调用一个系统调用，使内核打印`kmem`和`bcache`锁（这是本实验的重点）以及5个最有具竞争的锁的计数。如果存在锁争用，则`acquire`循环迭代的次数将很大。

本实验完成的任务是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。每个CPU都维护一个空闲列表，初始时将所有的空闲内存分配到某个CPU，此后各个CPU需要内存时，如果当前CPU的空闲列表上没有，则窃取其他CPU的。例如，所有的空闲内存初始分配到CPU0，当CPU1需要内存时就会窃取CPU0的，而使用完成后就挂在CPU1的空闲列表，此后CPU1再次需要内存时就可以从自己的空闲列表中取。

首先，将`kmem`定义为一个数组，包含`NCPU`个元素，即每个CPU对应一个

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

修改`kinit`，为所有锁初始化以“kmem”开头的名称，该函数只会被一个CPU调用，`freerange`调用`kfree`将所有空闲内存挂在该CPU的空闲列表上

```
void
kinit()
{
  char lockname[8];
  for(int i = 0;i < NCPU; i++) {
    snprintf(lockname, sizeof(lockname), "kmem_%d", i);
    initlock(&kmem[i].lock, lockname);
  }
  freerange(end, (void*)PHYSTOP);
}
```

修改`kfree`，使用`cpuid()`和它返回的结果时必须关中断

```
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();  // 关中断
  int id = cpuid();
  acquire(&kmem[id].lock);
  r->next = kmem[id].freelist;
  kmem[id].freelist = r;
```

```
  release(&kmem[id].lock);
  pop_off();  //开中断
}
```

修改`kalloc`，使得在当前CPU的空闲列表没有可分配内存时窃取其他内存的

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

  push_off();// 关中断
  int id = cpuid();
  acquire(&kmem[id].lock);
  r = kmem[id].freelist;
  if(r)
    kmem[id].freelist = r->next;
  else {
    int antid;  // another id
    // 遍历所有CPU的空闲列表
    for(antid = 0; antid < NCPU; ++antid) {
      if(antid == id)
        continue;
      acquire(&kmem[antid].lock);
      r = kmem[antid].freelist;
      if(r) {
        kmem[antid].freelist = r->next;
        release(&kmem[antid].lock);
        break;
      }
      release(&kmem[antid].lock);
    }
  }
  release(&kmem[id].lock);
  pop_off();  //开中断

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

Buffer cache

缓冲区缓存是一块很大的内存空间，为了确保多进程之间有序的读写，xv6为缓冲区缓存配备了一个大锁。这已经可以实现有序操作了，但还是存在问题:比如进程A只想修改disk block0的那一小块内容，但却要对整块缓冲区缓存上锁，相当于进程A不需要访问的其他大部分区域也被上了锁不能做任何操作，效率非常低下

实验思路则是将原来的 bcache.head 调整成 hash table，每个桶都有自己的锁。从而减少对全局锁 bcache.lock 的竞争。 给struct buf 结构增加时间戳，用于寻找最少使用的 buffer 当 目标桶 找不到合适的元素时，需要去全局的 bcache.buf 中寻找 buf.refcnt == 0 且最不经常使用的块，然后将该 buf 挪到 目标桶 。需要注意如下： 先将该buf 从其所属的 原来的桶 移除，然后再挪到桶 a，需要注意 原来的桶 和 目标桶 不能是同一个桶（同一个则不需要操作移动了），否则可能导致死锁。 遍历全局的 bcache.buf 时，需要加锁 bcache.lock 遍历找到 目标元素时，需要注意，目标元素 原来的桶 是没加锁的，因此从找到 目标元素 再到准备给 原来的桶 加锁移除该元素时，这中间存在空档期，有可能在这期间该块被引用了，导致 buf.refcnt != 0 ，此时不能在用这块 buf。因此在操作从 原来的桶 删除时，需要再检查一下 buf.refcnt ，如果不为 0 ，则需要重新遍历全局的 bcache.buf，找寻新的 目标元素

新的改进方案，可以建立一个从 blockno 到 buf 的哈希表，并为每个桶单独加锁。这样，仅有在两个进程同时访问的区块同时哈希到同一个桶的时候，才会发生锁竞争。当桶中的空闲 buf 不足的时候，从其他的桶中获取 buf。

在读写硬盘的时候，需要通过 `bread()` 函数得到相应的缓存（缓存中已经存放了硬盘对应块中的数据）

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!b->valid) {
    virtio_disk_rw(b, 0);
    b->valid = 1;
  }
  return b;
}
```

注意这里先调用了 `bget()` 函数。这个 `bget()` 会首先判断是否之前已经缓存过了硬盘中的这个块。如果有，那就直接返回对应的缓存，如果没有，会去找到一个最长时间没有使用的缓存，并且把那个缓存分配给当前块。所有的缓存被串到了一个双向链表里。链表的第一个元素是最近使用的，最后一个元素是很久没有使用的。

每次 `bget()` 的时候会先遍历一遍链表，检查当前块是否已经被存到缓存里了。如果没有，那就会从后到前遍历链表（意味着是从最久没有使用的开始找），找到第一个引用计数为 0 （代表没有程序正在使用这个块）的缓存作为当前块的缓存。

实验描述中给我们的提示是实现一个散列表。散列表会把块号映射到块缓存的桶，那么只有两个进程试图操作同一个桶中的块缓存，才会造成竞争。而且在查找所需块缓存时页不需要遍历所有的缓存，只需要遍历对应的桶。

当然，在对应桶中没有足够缓存时，我们可以像在 `kalloc()` 中一样，从别的桶中偷缓存。

这个实验中的散列表还是比较容易理解的。不过散列表中也有涉及页表分配实验中“偷”的过程，这样会陷入一种两难的境地。

在“偷”的过程中，我们会需要同时获得当前桶的锁，也需要检查别的桶，所以需要拿到别的桶的锁。这样就不可避免的同时持有了两把锁。这就造成了，在任何时候想要分配缓存，都需要竞争这个链表的锁。

```
假设块号 b1 的哈希值是 2，块号 b2 的哈希值是 5
并且两个块在运行前都没有被缓存
-----------------------------------------
CPU1                    CPU2
-----------------------------------------
bget(dev, b1)           bget(dev,b2)
    |                       |
    V                       V
获取桶 2 的锁             获取桶 5 的锁
    |                       |
    V                       V
缓存不存在，遍历所有桶     缓存不存在，遍历所有桶
    |                       |
    V                       V
  ......                  遍历到桶 2
    |                  尝试获取桶 2 的锁
    |                       |
    V                       V
  遍历到桶 5           桶 2 的锁由 CPU1 持有，等待释放
尝试获取桶 5 的锁
    |
    V
桶 5 的锁由 CPU2 持有，等待释放
```

除了锁相关的问题，我们还需要考虑如何找出最长时间没用过的缓存（LRU, least recent used）。因为 LRU 缓存通常在短时间之内不会再用到，所以在缓存不够的时候一般会回收这些缓存。

在原来的设计中，我们维护了一个双向链表，如果有新释放的缓存就加到链表的前面。所以链表尾部的缓存是最久没使用的，反之亦然。

但是在新设计中，我们维护了好几条链表（桶）没有办法在这些链表之间做比较。那么我们可以给 `buf` 结构体新加一个 `timestamp` 属性，表示最后一次使用的时间。而这个最后使用的时间可以从 `ticks` 全局变量获得，这个变量是由计时器中断维护的。代码如下：

定义哈希桶结构，并在`bcache`中删除全局缓冲区链表，改为使用素数个散列桶

```
#define NBUCKET 13
#define HASH(id) (id % NBUCKET)

struct hashbuf {
  struct buf head;       // 头节点
  struct spinlock lock;  // 锁
};

struct {
  struct buf buf[NBUF];
  struct hashbuf buckets[NBUCKET];  // 散列桶
} bcache;
```

(2). 在`binit`中，（1）初始化散列桶的锁，（2）将所有散列桶的`head->prev`、`head->next`都指向自身表示为空，（3）将所有的缓冲区挂载到`bucket[0]`桶上，代码如下

```
void
binit(void) {
  struct buf* b;
  char lockname[16];

  for(int i = 0; i < NBUCKET; ++i) {
    // 初始化散列桶的自旋锁
    snprintf(lockname, sizeof(lockname), "bcache_%d", i);
    initlock(&bcache.buckets[i].lock, lockname);

    // 初始化散列桶的头节点
    bcache.buckets[i].head.prev = &bcache.buckets[i].head;
    bcache.buckets[i].head.next = &bcache.buckets[i].head;
  }


  // Create linked list of buffers
```

```
  // Create linked list of buffers
  for(b = bcache.buf; b < bcache.buf + NBUF; b++) {
    // 利用头插法初始化缓冲区列表,全部放到散列桶0上
    b->next = bcache.buckets[0].head.next;
    b->prev = &bcache.buckets[0].head;
    initsleeplock(&b->lock, "buffer");
    bcache.buckets[0].head.next->prev = b;
    bcache.buckets[0].head.next = b;
  }
}
```

(3). 在***buf.h***中增加新字段`timestamp`，这里来理解一下这个字段的用途：在原始方案中，每次`brelse`都将被释放的缓冲区挂载到链表头，禀明这个缓冲区最近刚刚被使用过，在`bget`中分配时从链表尾向前查找，这样符合条件的第一个就是最久未使用的。而在提示中建议使用时间戳作为LRU判定的法则，这样我们就无需在`brelse`中进行头插法更改结点位置

```
struct buf {
  ...
  ...
  uint timestamp;  // 时间戳
};
```

(4). 更改`brelse`，不再获取全局锁

```
void
brelse(struct buf* b) {
  if(!holdingsleep(&b->lock))
    panic("brelse");

  int bid = HASH(b->blockno);

  releasesleep(&b->lock);

  acquire(&bcache.buckets[bid].lock);
  b->refcnt--;

  // 更新时间戳
  // 由于LRU改为使用时间戳判定，不再需要头插法
  acquire(&tickslock);
  b->timestamp = ticks;
  release(&tickslock);

  release(&bcache.buckets[bid].lock);
}
```

(5). 更改`bget`，当没有找到指定的缓冲区时进行分配，分配方式是优先从当前列表遍历，找到一个没有引用且

(3). 更改`bget`，当没有找到指定的缓冲区的进行分配，分配方式是优先从当前列表遍历，找到一个没有引用且`timestamp`最小的缓冲区，如果没有就申请下一个桶的锁，并遍历该桶，找到后将该缓冲区从原来的桶移动到当前桶中，最多将所有桶都遍历完。在代码中要注意锁的释放

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

  int bid = HASH(blockno);
  acquire(&bcache.buckets[bid].lock);

  // Is the block already cached?
  for(b = bcache.buckets[bid].head.next; b != &bcache.buckets[bid].head; b = b->next) {
    if(b->dev == dev && b->blockno == blockno) {
      b->refcnt++;

      // 记录使用时间戳
      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  b = 0;
  struct buf* tmp;

  // Recycle the least recently used (LRU) unused buffer.
  // 从当前散列桶开始查找
  for(int i = bid, cycle = 0; cycle != NBUCKET; i = (i + 1) % NBUCKET) {
    ++cycle;
    // 如果遍历到当前散列桶，则不重新获取锁
    if(i != bid) {
      if(!holding(&bcache.buckets[i].lock))
        acquire(&bcache.buckets[i].lock);
      else
        continue;
    }

    for(tmp = bcache.buckets[i].head.next; tmp != &bcache.buckets[i].head; tmp = tmp->next)
      // 使用时间戳进行LRU算法，而不是根据结点在链表中的位置
      if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp))
        b = tmp;

    if(b) {
```

```
    if(b) {
      // 如果是从其他散列桶窃取的，则将其以头插法插入到当前桶
      if(i != bid) {
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.buckets[i].lock);

        b->next = bcache.buckets[bid].head.next;
        b->prev = &bcache.buckets[bid].head;
        bcache.buckets[bid].head.next->prev = b;
        bcache.buckets[bid].head.next = b;
      }

      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;

      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);

      release(&bcache.buckets[bid].lock);
      acquiresleep(&b->lock);
      return b;
    } else {
      // 在当前散列桶中未找到，则直接释放锁
      if(i != bid)
        release(&bcache.buckets[i].lock);
    }
  }

  panic("bget: no buffers");
}
```

(6). 最后将末尾的两个小函数也改一下

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

void
bunpin(struct buf* b) {
  int bid = HASH(b->blockno);
  acquire(&bcache.buckets[bid].lock);
```

```
  b->refcnt--;
  release(&bcache.buckets[bid].lock);
}
```

踩过的坑：

1. bget中重新分配可能要持有两个锁，如果桶a持有自己的锁，再申请桶b的锁，与此同时如果桶b持有自己的锁，再申请桶a的锁就会造成死锁！因此代码中使用了`if(!holding(&bcache.bucket[i].lock))`来进行检查。此外，代码优先从自己的桶中获取缓冲区，如果自身没有依次向后查找这样的方式也尽可能地避免了前面的情况。
2. 在`bget`中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的！在提示中说`bget`如果未找到而进行分配的操作可以是串行化的，也就是说多个CPU中未找到，应当串行的执行分配，同时还应当避免死锁。

result

```
$ make qemu-gdb
(174.9s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (19.4s)
== Test running bcachetest ==
$ make qemu-gdb
(38.6s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (278.2s)
== Test time ==
time: OK
Score: 70/70
```