Lab4：多线程和锁机制

实验1：多线程（Multithreading）

实验内容：

实现一个用户态的线程库;尝试使用线程来为程序提速；并且尝试实现一个同步屏障

任务1：Uthread: switching between threads (moderate)

补全 uthread.c，完成用户态线程功能的实现。

这里的线程相比现代操作系统中的线程而言，更接近一些语言中的“协程”（coroutine）。原因是这里的“线程”是完全用户态实现的，多个线程也只能运行在一个CPU上，并且没有时钟中断来强制执行调度，需要线程函数本身在合适的时候主动yield释放CPU。这样实现起来的线程并不对线程函数透明，所以比起操作系统的线程而言更接近coroutine。

这个实验其实相当于在用户态重新实现一遍 xv6 kernel 中的 scheduler() 和 swtch() 的功能，所以大多数代码都是可以借鉴的。

要点

- 添加代码到 user/uthread.c 的 thread_create(), thread_schedule() 函数以及 user/uthread_switch.S 的 thread_swtich 函数.
- 确保当 thread_schedule() 第一个运行给定线程时, 该线程执行传递给 thread_create() 的函数在该线程的堆栈中.
- 确保 thread_switch 保存被切换出去的线程的寄存器, 恢复被切换到的线程的寄存器, 并返回到切换到的线程指令中最后停止的位置.

实验思路

该部分实验是在用户模式模拟一个进程有多个用户线程。 通过 thread_create() 创建线程, thread_schedule() 进行线程调度。 这与 xv6 的线程调度是大同小异的。对于此处用户多线程之间的切换, 也需要保存寄存器信息, 实际上就可以直接参考内核线程切换的 struct context 结构体, 需要保存的寄存器信息是一致的。而每个线程会有独立的需要执行的函数和线程栈, 则需要在创建线程进行设置。

实验步骤

- **设置线程上下文结构体**

由于此时用户多线程切换需要保存的寄存器信息和 xv6 内核线程切换需要保存的寄存器信息是一致的, 因此可以直接使用 kernel/proc.h 中定义的 struct context 结构体。 此处为了代码的清晰独立, 单独设置了 struct ctx 结构体, 其成员是和前者一致的。

```
// Saved registers for thread context switches.
struct ctx {
    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;
};
```

- **在线程结构体 struct thread 中添加线程上下文字段 context**

很显然, 上文定义的线程上下文结构体 struct ctx 是和线程一一对应的, 应作为线程结构体的一个成员变量.

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct ctx context;       // thread's context - lab7-1
};
```

- **添加代码到 thread_create() 函数**

thread_create() 函数主要进行线程的初始化操作: 其先在线程数组中找到一个状态为 FREE 即未初始化的线程, 然后设置其状态为 RUNNABLE 等进行初始化. 这里要注意到, 传递的 thread_create() 参数 func 需要记录, 这样在线程运行时才能运行该函数, 此外线程的栈结构是独立的, 在运行函数时要在线程自己的栈上, 因此也要初始化线程的栈指针. 而在线程进行调度切换时, 同样需要保存和恢复寄存器状态, 而上述二者实际上分别对应着 ra 和 sp 寄存器, 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.

```
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
```

```
  // set thread's function address and thread's stack pointer - lab7-1
  t->context.ra = (uint64) func;
  t->context.sp = (uint64) t->stack + STACK_SIZE;
}
```

- **添加代码到 thread_schedule() 函数**

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. 实际上与 kernel/proc.c 中的 scheduler() 函数是很相似的. 而很明显在找到线程后就需要进行线程的切换, 调用函数 thread_switch() 根据其在 user/thread.c 中的外部声明以及指导书的要求可以推断出, 该函数应该是定义在 user/uthread_switch.S, 用汇编代码实现. 因此其功能应该与 kernel/swtch.S 中的 swtch() 函数一致, 进行线程切换时的寄存器代码的保存与恢复。

```
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(&t->context, &current_thread->context);   // switch thread -
lab7-1
  } else
    next_thread = 0;
}
```

- **最后在 user/uthread_switch.S 中添加 thread_switch 的代码.**

正如上文所述, 该函数实际上功能与 kernel/swtch.S 中的 swtch 函数一致, 而由于此处 struct ctx 与内核的 struct context 结构体的成员是相同的, 因此该函数可以直接复用 kernel/swtch.S 中的 swtch 代码.

```
	.text

	/*
	 * save the old thread's registers,
	 * restore the new thread's registers.
	 */

	.globl thread_switch
thread_switch:
	/* YOUR CODE HERE */
	# same as swtch in swtch.S - lab7
	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 */
```

测试

在 xv6 中运行 uthread:

任务2：Using threads (moderate)

分析并解决一个哈希表操作的例子内，由于 race-condition 导致的数据丢失的问题。

```
[假设键 k1、k2 属于同个 bucket]

thread 1: 尝试设置 k1
thread 1: 发现 k1 不存在，尝试在 bucket 末尾插入 k1
--- scheduler 切换到 thread 2
thread 2: 尝试设置 k2
thread 2: 发现 k2 不存在，尝试在 bucket 末尾插入 k2
thread 2: 分配 entry，在桶末尾插入 k2
--- scheduler 切换回 thread 1
thread 1: 分配 entry，没有意识到 k2 的存在，在其认为的 “桶末尾”（实际为 k2 所处位置）插
入 k1

[k1 被插入，但是由于被 k1 覆盖，k2 从桶中消失了，引发了键值丢失]
```

哈希表的线程安全问题是: 多个线程同时调用 put() 对同一个 bucket 进行数据插入时, 可能会使得先插入的 entry 丢失. 具体来讲, 假设有 A 和 B 两个线程同时 put(), 而恰好 put() 的参数 key 对应到了哈希表的同一 bucket. 同时假设 A 和 B 都运行到 put() 函数的 insert() 处, 还未进入该函数内部, 这就会导致两个线程 insert() 的后两个参数是相同的, 都是当前 bucket 的链表头, 如若线程 A 调用 insert() 插入完 entry 后, 切换到线程 B 再调用 insert() 插入 entry, 则会导致线程 A 刚刚插入的 entry 丢失.

要点

- 使用互斥锁 pthread_mutex_t 来进行数据一致性的保护.
- 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

步骤

- **定义互斥锁数组**

根据指导书可知, 此处主要通过加互斥锁来解决线程不安全的问题. 此处没有选择使用一个互斥锁, 这样会导致访问整个哈希表都是串行的. 而考虑到对该哈希表, 实际上只有对同一 bucket操作时才可能造成数据的丢失, 不同bucket之间是互不影响的, 因此此处是构建了一个互斥锁数组, 每个 bucket对应一个互斥锁.

```
pthread_mutex_t locks[NBUCKET]; // lab7-2
```

- **在 main() 函数中对所有互斥锁进行初始化**

```
int
main(int argc, char *argv[])
{
  pthread_t *tha;
  void *value;
  double t1, t0;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s nthreads\n", argv[0]);
    exit(-1);
  }
  nthread = atoi(argv[1]);
  tha = malloc(sizeof(pthread_t) * nthread);
  srandom(0);
  assert(NKEYS % nthread == 0);
  for (int i = 0; i < NKEYS; i++) {
    keys[i] = random();
  }
  // initialize locks - lab7-2
  for(int i = 0; i < NBUCKET; ++i) {
      pthread_mutex_init(&locks[i], NULL);
  }

  //
  // first the puts
  // ...
}
```

- **在 put() 中加锁**

由于线程安全问题是由于对 bucket 中的链表操作时产生的, 因此要在对链表操作的前后加锁. 但实际上, 对于加锁的临界区可以缩小至 insert() 函数. 原因是 insert() 函数采取头插法插入 entry, 在函数的最后才使用 *p=e 修改 bucket 链表头 table[i] 的值, 也就是说, 在前面操作的同时, 并不会对 bucket 链表进行修改, 因此可以缩小临界区的方法 (根据 “Roy__Mustang” 的指正, 这里不能将加锁范围缩小至 insert() 函数内部的 *p=e, 因为进入 insert() 函数后, 参数 p 和 n 都已确定, 否则多线程对同一 bucket 链表插入时只会有一个新建的 entry 能成功插入, 其他的会被覆盖, 从而导致错误.)

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;

  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
```

```
      e->value = value;
    } else {
      pthread_mutex_lock(&locks[i]);    // lock - lab7-2
      // the new is new.
      insert(key, value, &table[i], table[i]);
      pthread_mutex_unlock(&locks[i]);  // unlock - lab7-2
    }
  }
```

- **不需要在 get() 中加锁**

get() 函数主要是遍历 bucket 链表找寻对应的 entry, 并不会对 bucket 链表进行修改, 实际上只是读操作, 因此无需加锁.

- **增大 NBUCKET** 增大 NBUCKET 即增加哈希表的 bucket数, 从而一定程度上会减少两个同时运行的线程 put() 时对同一个 bucket 进行操作的概率, 自然也就减少了锁的争用, 能够一定程度上挺高并发性能. 此处选择 NBUCKET=7.

```
#define NBUCKET 7   // lab7
```

测试

./grade-lab-thread ph_fast 单项测试

任务3：Barrier(moderate)

利用 pthread 提供的条件变量方法，实现同步屏障机制。

要点

- 实现满足线程同步的 barrier.
- 使用条件变量 pthread_cond_t 配合互斥锁完成多线程同步.

实验思路

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.

实验步骤

根据上述思路, 在 barrier() 函数中添加如下代码. 注意对于变量 bstate.round 和 bstate.nthread 的设置需要在 pthread_cond_broadcast() 唤醒其它线程之前, 否则其他线程进入下一轮循环时可能这两个字段的值还未得到修改.

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  // lab7-3
  pthread_mutex_lock(&bstate.barrier_mutex);
  // judge whether all threads reach the barrier
  if(++bstate.nthread != nthread)  {    // not all threads reach
    pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex);  // wait other
threads
  } else {  // all threads reach
    bstate.nthread = 0; // reset nthread
    ++bstate.round; // increase round
    pthread_cond_broadcast(&bstate.barrier_cond);   // wake up all sleeping
threads
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

测试 ./barrier

实验2：锁机制（Locks）

实验内容

重新设计 xv6 的内存分配和磁盘块缓存机制以提高它们的并行性。并行性的性能可以由锁争用的次数反应出来——差的并行代码会导致高锁争用。 而提高并行效率的根本方法，就是减小锁的颗粒度。

任务1：Memory allocator (moderate)

实验思路 原本的 kalloc.c 导致高锁争用的原因是 xv6 只维护了一个空闲页面链表，该链表有一个锁。为了减少锁争用，我们可以给每一个 CPU 维护一个空闲页面链表，这样不同 CPU 就可以并行地内存分配和释放，因为它们之间相互独立。但是，当一个 CPU 的空闲页面被分配完之后，它需要从其他的 CPU 的空闲页面链表中窃取一部分空闲页面。窃取过程可能导致锁争用，但是不会很频繁。

我们的任务就是实现每一个 CPU 一个空闲链表，且在链表为空时窃取页面。所有锁的名字必须以 kmem 开头。kalloctest 检测是否减少了锁争用，usertests sbrkmuch 检测是否仍然能够分配所有内存

实验步骤

- 定义与初始化

这里自定义了一个初始化函数 kfree_specific 将指定 pa 分配给指定 cpu 的 freelist

```
struct mem{
  struct spinlock lock;
  struct run *freelist;
};

struct mem kmem[NCPU];

void
kfree_specific(void *pa, int cpuid)
{
  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  struct run* r = (struct run*)pa;
  struct mem* cpu_mem = &kmem[cpuid];
  acquire(&cpu_mem->lock);
  r->next = cpu_mem->freelist;
  cpu_mem->freelist = r;
  release(&cpu_mem->lock);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  push_off();
  int hart = cpuid();
  pop_off();
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){
    kfree_specific(p, hart);
  }
}
```

- 分配内存

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;
  push_off();
  int hart = cpuid();
  pop_off();
  struct mem* cpu_mem = &kmem[hart];
  acquire(&cpu_mem->lock);
  r = cpu_mem->freelist;
  if(r){
    cpu_mem->freelist = r->next;
```

```
    release(&cpu_mem->lock);
  }
  else // "steal" free memory from other cpu mem list
  {
    release(&cpu_mem->lock);
    for (int i = 0; i < NCPU ; i++) {
      cpu_mem = &kmem[i];
      acquire(&cpu_mem->lock);
      r = cpu_mem->freelist;
      if (r){
        cpu_mem->freelist = r->next;
        release(&cpu_mem->lock);
        break;
      }
      release(&cpu_mem->lock);
    }
  }

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

- 释放内存

```
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);

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

  r = (struct run*)pa;
  //int hart = get_pa_cpu_id((uint64)r);
  struct mem* cpu_mem = &kmem[hart];
  acquire(&cpu_mem->lock);
  r->next = cpu_mem->freelist;
  cpu_mem->freelist = r;
  release(&cpu_mem->lock);
}
```

测试

make qemu

$ kalloctest

$ usertests sbrkmuch

$ usertests

任务2：Buffer cache

实验内容

通过重新设计文件系统的 buffer & cache 的数据结构来降低锁的竞争

实验思路

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

实验步骤

- 定义初始化

binit 将原来的 bcache.buf 所有 buf 平摊到每个桶中。

```
#define BUCKET_CNT 13
#define NBUF (BUCKET_CNT * 3)

struct bcache_bucket{
  struct buf head;
  struct spinlock lock;
};
struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  struct bcache_bucket bucket[BUCKET_CNT];
} bcache;

int hash_key(int blockno){
  return blockno % BUCKET_CNT;
}

void binit(void)
{
  struct buf *b;
  char buf[32];
```

```
  int sz = 32;

  initlock(&bcache.lock, "bcache");

  for (int i = 0; i < BUCKET_CNT; i++){
    snprintf(buf, sz, "bcache.bucket_%d", i);
    initlock(&bcache.bucket[i].lock, buf);
  }

  // Create linked list of buffers
  int blockcnt = 0;
  struct bcache_bucket* bucket;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    initsleeplock(&b->lock, "buffer");
    b->access_time = ticks;
    b->blockno = blockcnt++;
    bucket = &bcache.bucket[hash_key(b->blockno)];
    b->next = bucket->head.next;
    bucket->head.next = b;
  }
}
```

- **获取 buf**
  - 哈希获取对应的桶
  - 获取该桶的锁 **bcache_bucket.lock**
  - 检查该块是否已缓存
    - 未缓存
      - 首先遍历该桶的列表 **bcache_bucket.head**
      - 检查是否有 **buf.refcnt == 0** 的 **buf**，取其中距离上次访问时间最久的**buf**
        - 没有，则需要寻找一块新的 **buf**
          - 遍历全局 **bcache.buf** ，此时需要获取全局锁 **bcache.lock**
          - 找到 **buf.refcnt == 0** 的 **buf**，取其中距离上次访问时间最久的 **buf**
          - 哈希 **buf.blockno** 获取该 **buf** 原来的桶
          - 遍历该 **buf** 原来的桶，从 **bcache_bucket.head** 中移除该 **buf** ，此时应持有该桶的锁 **bcache_bucket.lock** ，释放 **buf** 完毕时释放锁
          - 遍历时需要注意，需要再次检查 **buf.refcnt == 0**，因为前面遍历全局 **bcache.buf** 的时候，没有持有该桶的锁，在此期间 **buf** 可能被引用了
          - 将该 **buf** 移动到新桶
          - 更新 **buf** 结构体信息
        - 有，则不需要移动该 **buf**
          - 更新 **buf**结构体信息
    - 已缓存
      - 更新 **buf** 结构体信息

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

```
  struct buf *b, *lrub;
  struct bcache_bucket* bucket = &bcache.bucket[hash_key(blockno)];

  acquire(&bucket->lock);
  // Is the block already cached?
  for(b = &bucket->head; b; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      b->access_time = ticks;
      release(&bucket->lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  /**
   * 检查没有发现cached的 buf，那么就根据 ticks LRU 策略
   * 选择第 id 号哈希桶中 ticks 最小的淘汰，ticks 最小意味着
   * 该 buf 在众多未被使用到（ b->refcnt==0 ）的 buf 中是距今最远的
   * */
  // find bucket lru buffer
  lrub = 0;
  uint min_time = 0x8fffffff;
  for(b = &bucket->head; b; b = b->next){
    if (b->refcnt == 0 && b->access_time < min_time){
      min_time = b->access_time;
      lrub = b;
    }
  }
   /**
   直接覆盖待淘汰的buf，无需再将其中的旧内容写回至disk中
  标记位valid置0，为了保证能够读取到最新的数据
    */
  if (lrub) {
    goto setup;
  }

  // Not cached.
  // find in the global array
  acquire(&bcache.lock);

findbucket:
  lrub = 0;

  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    if(b->refcnt == 0 && b->access_time < min_time) {
      lrub = b;
    }
  }

  if (lrub) {
    // step 1 : release from the old bucket
    // need to hold the old bucket lock
    struct bcache_bucket* old_bucket = &bcache.bucket[hash_key(lrub->blockno)];
    acquire(&old_bucket->lock);
```

```
    if (lrub->refcnt != 0){
      release(&old_bucket->lock);
      goto findbucket;
    }

    b = &old_bucket->head;
    struct buf* bnext = b->next;
    while (bnext != lrub) {
      b = bnext;
      bnext = bnext->next;
    }
    b->next = bnext->next;

    // we don't need to modify bcache.bucket , so we release the lock
    release(&old_bucket->lock);
    // step 2 : add to target bucket
    lrub->next = bucket->head.next;
    bucket->head.next = lrub;
    release(&bcache.lock);

setup:
    lrub->dev = dev;
    lrub->blockno = blockno;
    lrub->valid = 0;
    lrub->refcnt = 1;
    lrub->access_time = ticks;
    release(&bucket->lock);
    acquiresleep(&lrub->lock);
    return lrub;
  }
panic("bget: no buffers");
}
```

- **释放 buf**

该函数用于释放缓存块. 在原本的实现中, 若其引用计数为 0, 则将其移至双向链表表头, 这样双向链表表头是最近使用的, 表尾是最近未使用的, 构成一个 LRU 序列, 方便 bget() 函数寻找缓存块。 而此处要使用基于时间戳的 LRU 实现, 此外，由于是通过哈希表管理, 加锁也由原本的全局锁改为缓存块所在的 bucket 的锁。

```
// Release a locked buffer.
// Move to the head of the most-recently-used list.
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

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

```
  struct bcache_bucket* bucket = &bcache.bucket[hash_key(b->blockno)];
  acquire(&bucket->lock);
  b->refcnt--;
  if(b->refcnt == 0)
   b->lastuse = ticks;
  release(&bucket->lock);
}
```

- **bpin 和 bunpin**

这块内容较为简单，将锁从全局锁替换为桶锁即可

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

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

测试

手动进入 qemu

make qemu

$bcachetest

$usertests