哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2023 年秋季

课程名称：操作系统

实验名称：锁机制的应用

实验性质：课内实验

实验时间：10.19 地点：T2507

学生班级：8

学生学号：210110812

学生姓名：李春阳

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 年 9 月

一、 回答问题

1、 内存分配器

a. 什么是内存分配器？它的作用是？

物理内存分配器（allocator）定义在 kernel/kalloc.c 中，主要功能是为用户进程和内核分配和释放物理内存页面。

具体来说，xv6 中的内存分配器将系统物理内存划分成固定大小的块，并维护空闲块列表。内核和用户进程可以使用 kalloc() 和 kfree() 等函数来申请和释放内存块。当需要内存块时，内核会从空闲块列表中查找可用的块并分配，当释放内存块时，该块会被添加到空闲块列表中以便下次分配使用。

b. 内存分配器的数据结构是什么？它有哪些操作（函数），分别完成了什么功能？

数据结构如下：

```
// run结构体就是一个指向自身的指针，用于指向下一个空闲页表开始位置
struct run {
  struct run *next;
};

// 管理物理内存的结构,有一把锁lock保证访问时的互斥性,以及一个指向
struct kmem{
  struct spinlock lock;
  struct run *freelist;
};

// 为每个CPU独立的freelist
struct kmem kmems[NCPU];
```

内存分配器的核心数据结构是空闲物理页组成的链表 freelist，这个空闲页链表将物理内存划分成 4KB 大小的页来管理，并使用自旋锁进行保护。每个空闲页在链表里都由 struct run next 指向下一个空闲物理页。

它有 kinit()、freerange(void *pa_start, void *pa_end)、kfree(void *pa)、kalloc(void)操作（函数）。

- kinit()：通过保存所有空闲页初始化分配器，并初始化自旋锁。kinit()调用 freerange()来把空闲内存页加到链表里，而 freerange()调用 kfree()把每个空闲页（地址范围从 pa_start 至 pa_end）逐一加到链表的表头。
- freerange(void *pa_start, void *pa_end)：释放内存页（地址范围从 pa_start 至 pa_end），然后将这些空闲页逐一加到链表的表头。
- kfree(void *pa)：用于释放指定的物理内存页，将其添加至 freelist 的表头中，参数 pa 为需要释放的物理页页号，即物理页的首地址。
- kalloc(void)：用来分配内存物理页，返回指向空闲页的指针，同时将该页从空闲页链表 freelist 中摘下。

c. 为什么指导书提及的优化方法可以提升性能？

在未优化的方案中，kalloc()和 kfree()函数对 freelist 的操作进行了上锁操作。当多个 CPU 并行时，则会出现多个 CPU 同时争抢一把锁的情况，这减少了并发性，降低了性能。而优化方法使每个 CPU 核使用独立的链表，而非原来的共享链表。这样等分，就不会出现所有的 CPU 争抢一个空闲区域的情况，从而提升了性能。

2、 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存是经常访问的磁盘在内存中的复制，是磁盘与文件系统交互的中间层，定义在 kernel/bio.c 中。由于 xv6 的文件系统是以磁盘数据块为单位从磁盘读写数据的，对磁盘的读取非常慢，而内存的速度要快得多，因此将最近经常访问的磁盘块缓存在内存里可以大大提升性能（此时内存起到 cache 的作用）。

b. buf 结构体为什么有 prev 和 next 两个成员，而不是只保留其中一个？请从这样做的优点分析（提示：结合通过这两种指针遍历链表的具体场景进行思考）。

buf 结构体有 prev 和 next 两个成员，从而形成了双向链表，这可以加速查找链表中的缓存块，大大提升了性能。在查找空闲块时，双向链表可以从正向和逆向查找链表中的缓存块。由于原始方案中，磁盘缓存链表按照缓存块是否空闲来排序，因此在查找空闲磁盘块时需要从后向前查找，在这种情况下，双向链表效率更高。

c. 为什么哈希表可以提升磁盘缓存的性能？可以使用内存分配器的优化方法优化磁盘缓存吗？请说明原因。

未使用哈希表时，所有的缓存块均处于一个链表中，且被一把自旋锁 bache.lock 锁住。在这种情况下，如果多个进程同时访问磁盘缓存，则仅有一个进程可以访问到，其余的进程均需要等待，无法并发执行，这大大的降低了性能。而使用哈希桶将各缓存块进行分组、并为每个哈希桶分配一个专用的锁后，当需要访问某缓存块时，仅需要对所在的哈希桶进行加锁，而桶和桶之间可以并行操作，这提升了磁盘缓存的性能。

二、 实验详细设计

1. 内存分配器：

由于优化方案中需对每个 CPU 核使用独立的链表，因此我们首先创建数组 kmems 数组，用于储存每个 CPU 独立的 kmem。

```
// run结构体就是一个指向自身的指针，用于指向下一个空闲页表开始位置
struct run {
  struct run *next;
};

// 管理物理内存的结构,有一把锁lock保证访问时的互斥性,以及一个指向
struct kmem{
  struct spinlock lock;
  struct run *freelist;
};

// 为每个CPU独立的freelist
struct kmem kmems[NCPU];
```

修改 kinit()，使 kmems 数组中对应的锁均被初始化，且在初始化的时候利用 snprintf 函数为 kmems 数组中的锁进行命名，锁名字以 kmem 开头。

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

修改 kfree()，当释放某一内存页时，通过 cpuid()函数获取当前的 CPU 的 id 号，并使用 push_off()和 pop_off()保证当前进程不可被中断。

```
// 获取当前CPU编号
push_off();
int cpu_id = cpuid();
pop_off();
```

接着，将对应的空闲页放入空闲列表中。我们首先获取当前 CPU 的锁，然后将待释放的内存页放入该 CPU 对应的空闲页链表 freelist 中，最后释放锁。

```
// 将对应的空闲页放入空闲列表中
acquire(&kmems[cpu_id].lock);
r->next = kmems[cpu_id].freelist;
kmems[cpu_id].freelist = r;
release(&kmems[cpu_id].lock);
```

最后修改 kalloc()，同 kfree()一样，需要首先获取当前的 CPU 的 id 号，在此不

再赘述。然后获取当前 CPU 对应的锁，查看其空闲页链表中是否还有空闲页（r 是否为 null），若有则将该空闲页 r 从空闲页链表中移出，然后释放该锁。

```
104   acquire(&kmems[cpu_id].lock);
105   r = kmems[cpu_id].freelist;
106
107   //如果当前CPU有空闲内存块
108   if(r) {
109     kmems[cpu_id].freelist = r->next;
110     release(&kmems[cpu_id].lock);
111   }
```

如果当前 CPU 没有空闲内存块，则需要从其他 CPU 的 freelist 中窃取内存块。在搜索其他 CPU 的 freelist 中是否有空闲块之前，先释放当前 CPU 对应的锁，已避免死锁。然后遍历除了当前 CPU 的 id 外的所有 CPU 的 id，并获取该 CPU 的锁，同时访问该 CPU 的空闲页链表，查看是否有空闲页。若有空闲页，则窃取一页内存，将该空闲页从链表中移出，然后释放该 CPU 的锁；否则释放该 CPU 的锁，继续访问下一个 CPU。如果遍历所有 CPU 都没有空闲页，则返回 0。

```
112   //如果当前CPU没有空闲内存块
113   else {
114     release(&kmems[cpu_id].lock);
115     for(int i=0; i<NCPU; i++) {
116       if (i == cpu_id)  continue;
117
118       acquire(&kmems[i].lock);
119       r = kmems[i].freelist;
120       if (r) {
121         kmems[i].freelist = r->next;
122         release(&kmems[i].lock);
123         break;
124       } else {
125         release(&kmems[i].lock);
126       }
127
128       // 所有CPU都没有空闲块时，返回0
129       if(i == NCPU-1)  return 0;
130     }
131   }
```

2. 磁盘缓存

在该任务中，我采用哈希桶进行优化。优化后的结构体如下。

```
struct {
  struct spinlock overall_lock;   //全局锁
  struct spinlock lock[NBUCKETS]; //每个哈希桶的锁
  struct buf buf[NBUF];
  struct buf hashbucket[NBUCKETS]; //每个哈希队列一个linked list及一个lock
} bcache;
```

取哈希桶的数量为 13，并为每个哈希桶分配一个专用的锁，同时增加一个全局锁。全局锁仅在挪用不同哈希桶之间的内存块时使用，因此并不影响其余进程的并行。

定义哈希函数如下，以将哈希桶与锁对应起来。

```
uint
hash (uint n) {
  return n % NBUCKETS;
}
```

首先修改 binit(void)函数，初始化全局锁及每个哈希桶的锁，并将每个哈希桶的双向链表初始化。

```
initlock(&bcache.overall_lock, "bcache");

//初始化哈希锁
for(int i=0; i<NBUCKETS; i++){
  snprintf(bcache_name[i], sizeof(bcache_name[i]), "bcache%d", i);
  initlock(&bcache.lock[i], bcache_name[i]);

  //双向链表初始化
  bcache.hashbucket[i].prev = &bcache.hashbucket[i];
  bcache.hashbucket[i].next = &bcache.hashbucket[i];
}
```

然后将每个缓存块通过哈希函数均匀的分配在每个哈希桶内，采用头插法。

```
for(int i=0; i<NBUF; i++){
  uint h = hash(i);
  b = &bcache.buf[i];
  b->next = bcache.hashbucket[h].next;
  b->prev = &bcache.hashbucket[h];
  initsleeplock(&b->lock, "buffer");
  bcache.hashbucket[h].next->prev = b;
  bcache.hashbucket[h].next = b;
}
```

接着修改 brelse(struct buf *b)函数。对当前缓存块所在的哈希桶上锁，然后将该缓存块的 refcnt 减一。若 refcnt 减为了 0，代表此时没有进程在使用该缓存块，那么该缓存块可以被释放，则将该块从链表中删去，并添加到链表头。

```
uint key = hash(b->blockno);
acquire(&bcache.lock[key]);
b->refcnt--;
if (b->refcnt == 0) {
  // 没有地方再引用这个块，将块链接到缓存区链头
  // 将b从链表中删去
  b->next->prev = b->prev;
  b->prev->next = b->next;
  // 将b添加到链表头
  b->next = bcache.hashbucket[key].next;
  b->prev = &bcache.hashbucket[key];
  bcache.hashbucket[key].next->prev = b;
  bcache.hashbucket[key].next = b;
```

然后修改 bpin(struct buf *b)和 bunpin(struct buf *b)函数，同理对当前缓存块所在的哈希桶上锁，而后对缓存块的 refcnt 参数进行相应操作，最后释放哈希桶的锁。

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

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

最难修改的是 bget(uint dev, uint blockno)函数。在这个函数中我们主要实现 3 个逻辑：首先获取所查询的缓存块对应的哈希桶的锁，如果所查询的缓存块在自己的哈希桶中命中，则释放锁并返回指向该缓存块的指针。

```
uint key = hash(blockno);
acquire(&bcache.lock[key]);

// 如果在自己的哈希桶中命中
for(b = bcache.hashbucket[key].next; b != &bcache.hashbucket[key]; b = b->next){
  if(b->dev == dev && b->blockno == blockno){
    b->refcnt++;
    release(&bcache.lock[key]);
    acquiresleep(&b->lock);
    return b;
  }
}
```

若未命中，则在自己的桶中从后往前寻找空闲缓存块，如果存在空闲块，则替换后释放锁，并返回指向该缓存块的指针。

```
// 如果没命中，首先在自己的哈希桶中寻找空闲块，若有则替换后返回
for(b = bcache.hashbucket[key].prev; b != &bcache.hashbucket[key]; b = b->prev){
  if(b->refcnt == 0) {
    b->dev = dev;
    b->blockno = blockno;
    b->valid = 0;
    b->refcnt = 1;
    release(&bcache.lock[key]);
    acquiresleep(&b->lock);
    return b;
  }
}
```

若自己的哈希桶内不存在空闲块，则需要去别的桶中窃取一个内存块。但需要先释放当前哈希桶的锁，同时获取全局锁。

```
// 本桶没有refcnt == 0的桶，就要找别的桶，但是需要先释放本桶的锁
release(&bcache.lock[key]);

// 获取全局锁。仅在挪用不同哈希桶之间的内存块时使用到了全局锁，因此并不影响其余进程的并行
acquire(&bcache.overall_lock);
```

遍历其余的所有哈希桶，寻找可以窃取的块。在窃取之前，获取当前遍历到的哈希桶的锁，然后判断在这个桶内是否可以寻找到一个空闲块。如果可以找到，则进行替换，把该块从此哈希桶中删除并添加到当前的哈希桶中。在将空闲块添加到当前哈希桶中的过程中，需要对当前哈希桶进行上锁，并在替换后解锁，否则容易产生死锁。替换和添加工作完成之后，释放当前遍历到的哈希桶的锁及全局锁，然

后返回指向该缓存块的指针。

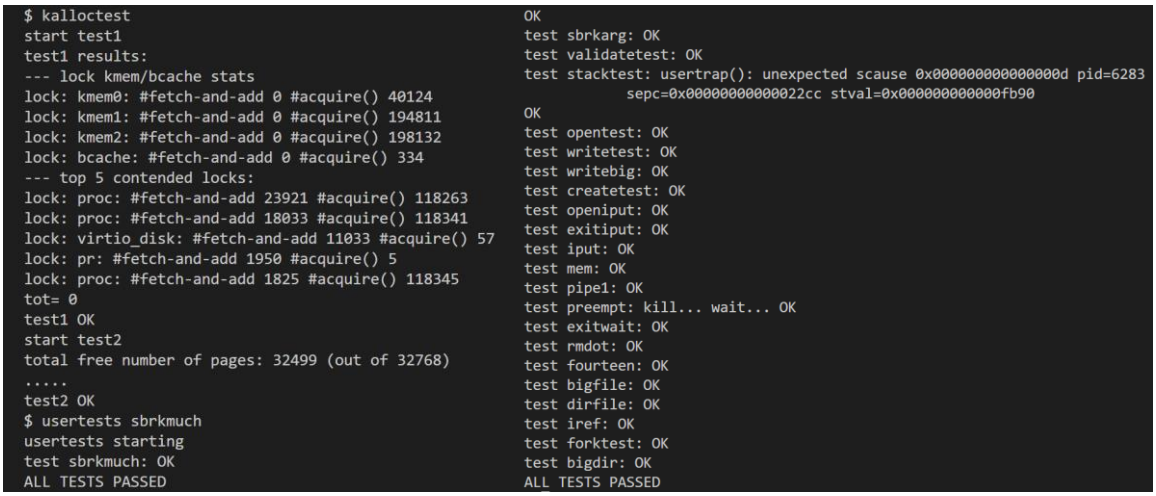
```
// 遍历所有哈希桶，寻找可以窃取的块
for (int i=0; i<NBUCKETS; i++){
  if (i == key) continue;
  acquire(&bcache.lock[i]);   //获取该哈希桶的锁

  // 判断在其他哈希桶中是否寻找到一个空闲块
  for(b = bcache.hashbucket[i].prev; b != &bcache.hashbucket[i]; b = b->prev){
    if(b->refcnt == 0) {
      // 可以找到
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;

      // 把该块从哈希桶i中删除
      b->next->prev = b->prev;
      b->prev->next = b->next;

      // 将该块添加到当前哈希桶中
      acquire(&bcache.lock[key]);
      b->next = &bcache.hashbucket[key];
      b->prev = bcache.hashbucket[key].prev;
      bcache.hashbucket[key].prev->next = b;
      bcache.hashbucket[key].prev = b;
      release(&bcache.lock[key]);

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

如果仍未寻找到空闲块，则释放全局锁并执行 panic。

```
  release(&bcache.overall_lock);
  panic("bget: no buffers");
}
```

三、 实验结果截图

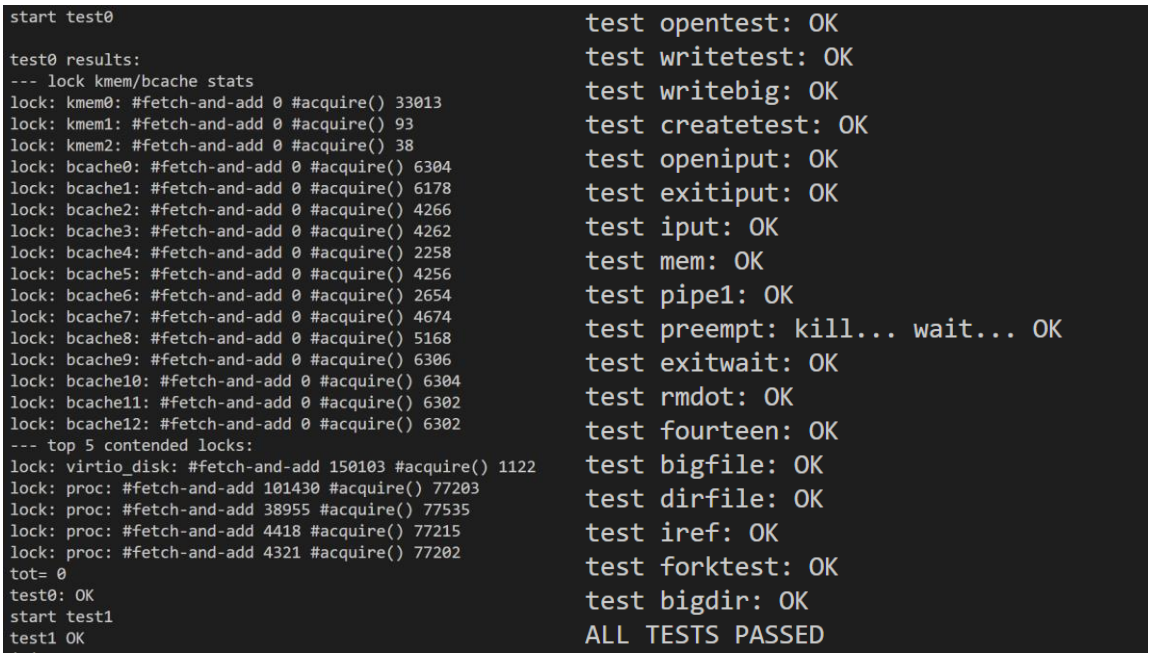
kalloctest：（右侧 usertests 为部分截图）



```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem0: #fetch-and-add 0 #acquire() 40124
lock: kmem1: #fetch-and-add 0 #acquire() 194811
lock: kmem2: #fetch-and-add 0 #acquire() 198132
lock: bcache: #fetch-and-add 0 #acquire() 334
--- top 5 contended locks:
lock: proc: #fetch-and-add 23921 #acquire() 118263
lock: proc: #fetch-and-add 18033 #acquire() 118341
lock: virtio_disk: #fetch-and-add 11033 #acquire() 57
lock: pr: #fetch-and-add 1950 #acquire() 5
lock: proc: #fetch-and-add 1825 #acquire() 118345
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
```

```
OK
test sbrkarg: OK
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
```

bcachetest：（右侧 usertests 为部分截图）



```
start test0

test0 results:
--- lock kmem/bcache stats
lock: kmem0: #fetch-and-add 0 #acquire() 33013
lock: kmem1: #fetch-and-add 0 #acquire() 93
lock: kmem2: #fetch-and-add 0 #acquire() 38
lock: bcache0: #fetch-and-add 0 #acquire() 6304
lock: bcache1: #fetch-and-add 0 #acquire() 6178
lock: bcache2: #fetch-and-add 0 #acquire() 4266
lock: bcache3: #fetch-and-add 0 #acquire() 4262
lock: bcache4: #fetch-and-add 0 #acquire() 2258
lock: bcache5: #fetch-and-add 0 #acquire() 4256
lock: bcache6: #fetch-and-add 0 #acquire() 2654
lock: bcache7: #fetch-and-add 0 #acquire() 4674
lock: bcache8: #fetch-and-add 0 #acquire() 5168
lock: bcache9: #fetch-and-add 0 #acquire() 6306
lock: bcache10: #fetch-and-add 0 #acquire() 6304
lock: bcache11: #fetch-and-add 0 #acquire() 6302
lock: bcache12: #fetch-and-add 0 #acquire() 6302
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 150103 #acquire() 1122
lock: proc: #fetch-and-add 101430 #acquire() 77203
lock: proc: #fetch-and-add 38955 #acquire() 77535
lock: proc: #fetch-and-add 4418 #acquire() 77215
lock: proc: #fetch-and-add 4321 #acquire() 77202
tot= 0
test0: OK
start test1
test1 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
```

makegrade：

```
$ make qemu-gdb
(125.2s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (12.2s)
== Test running bcachetest ==
$ make qemu-gdb
(10.1s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (159.9s)
== Test time ==
time: OK
Score: 70/70
```