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

实验设计报告

开课学期：2022 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：_____地点：_____

学生班级：5

学生学号：200110529

学生姓名：梁爽

评阅教师：_____

报告成绩：_____

实验与创新实践教育中心印制

2022 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？

内存分配器（Memory Allocator）负责内存分配与管理，使用内存分配器就是为了更加高效地更加安全地利用内存

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？

分配器的核心数据结构是由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 大小的页帧来管理，并使用 **自旋锁** （spinlock）进行保护。每个空闲页在链表里都是 `struct run next` 指向下一个空闲物理页。

kinit()：初始化分配器
Freerange():把空闲内存页加到链表里，借用 kfree，范围是 pa_start 至 pa_end
kfree()：函数用于释放指定的物理内存页，将其添加至 freelist 中
kalloc()：用来分配内存物理页，移除并返回空闲链表头的第一个元素，即给调用者分配 1 页物理内存。

c．为什么指导书提及的优化方法可以提升性能？

修改空闲内存链表，可以减少锁的争抢， 使每个 CPU 核使用独立的链表 ，而不

是共享链表。这样等分，就不会让所有的 CPU 争抢一个空闲区域。所以锁的被争抢次数会大大减少，性能也会得到提高。

2、 磁盘缓存

a．什么是磁盘缓存？它的作用是？

将下载到的数据先保存于系统为软件分配的内存空间中，当保存到内存池中的数据达到一个程度时，便将数据保存到硬盘中
作用：磁盘缓存是为了减少 CPU 透过 I/O 读取磁盘机的次数，提升磁盘 I/O 的效率，用一块内存来储存存取较频繁的磁盘内容；因为内存的存取是电子动作，而磁盘的存取是 I/O 动作，感觉上磁盘 I/O 变得较为快速，提高了性能。

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）。

在释放缓存块和从其他散列桶窃取缓存块时，为了维持原有链表的相对有序性，需要同时有前面块和后面块的指针，否则移除这个缓存块之后，别的块顺序和位置不能确认。

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。

将各块块号 blockno 的某种散列值作为 key 对块进行分组，并为每个哈希桶分配一个专用的锁。通过哈希桶来代替链表，当要获取和释放缓存块时，只需要对某个哈希桶进行加锁，桶之间的操作就可以并行进行，提供并行性能。
不可以用内存分配器的优化方式，因为他的优化方案是使每个 CPU 核使用独立的链表，每个空闲物理页只能存在于一个 freelist 中，而磁盘缓存中的数据，每

一个 CPU 都有可能会访问，如果将数据限定在单独一个 CPU 上，没有意义，且磁盘缓存是提高进程并发性，减少锁争用的策略，和进程有关，一个 CPU 也可能有多个进程。

二、 实验详细设计

1.对于 kalloc.c 文件

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

将空闲链表独立分配给每一个 CPU 核

```
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);

}
```

kinit 初始化，根据最大 CPU 数量来设定好每一个 lockname 并初始化对应的锁

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

在 kfree 中，被释放的内存页直接插入到其对应的 cpu 所对应的链表的头部就可以

```
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);
```

在 kalloc 中，当前 CPU 自己有空闲内存块，则直接用。当前 CPU 没有空闲内存块，则从其他 CPU 的 freelist 中 窃取 内存块；所有 CPU 都没有空闲块时，返回 0。每一次访问都需要上锁，以进行互斥访问。

2. 在磁盘缓存实验中，先在 buf.h 中添加 timestamp 时间戳，

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

定义一个 hashbuf 数据结构作为散列桶，每一个 hashbuf 桶都有自己的缓存块（知道头结点就知道了缓存块），同时还有一个自选锁，在 bcache 中，初始化 NBUCKETS 个散列桶

```
struct hashbuf hashbucket[NBUCKETS];
```

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

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

初始化散列通自旋锁和头节点

```
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.hashbucket[0].head.next;
    b->prev = &bcache.hashbucket[0].head;
    initsleeplock(&b->lock, "buffer");
    bcache.hashbucket[0].head.next->prev = b;
    bcache.hashbucket[0].head.next = b;
  }
```

利用头插法初始化缓冲区列表,全部放到散列桶 0 上

在 bget 函数中

```
int cur=HASH(blockno);//当前块的 blocknumber
  acquire(&bcache.hashbucket[cur].lock);
```

利用当前块号获取哈希值，得到他处于哪一个散列桶中。

```
for(b = bcache.hashbucket[cur].head.next; b != &bcache.hashbucket[cur].head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;

      //使用时间戳
      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);
      release(&bcache.hashbucket[cur].lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
```

查看我们想要的这个块是否已经在缓存中，如果是就更新时间戳

```
int cnt=0,i=cur;
  struct buf* tmp;
  b=0;
```

cnt 代表我们已经访问的散列桶个数，访问所有散列桶，以此来找到可用的空闲块，变量 i 自增，代表当前访问的散列桶。tmp 作为临时变量，起到了遍历链表的作用，b 用来记录满足要求的这一块

```
    if(holding(&bcache.hashbucket[i].lock))
    {
      continue;
    }
    acquire(&bcache.hashbucket[i].lock);
    for(tmp = bcache.hashbucket[i].head.prev; tmp != &bcache.hashbucket[i].head; tmp=
tmp->prev){
      if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp)) {
        b=tmp;
      }
    }
```

遍历当前散列桶查找可用块。且时间戳越早越好。

```
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 != cur) {
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.hashbucket[i].lock);

        b->next = bcache.hashbucket[cur].head.next;
        b->prev = &bcache.hashbucket[cur].head;
        bcache.hashbucket[cur].head.next->prev = b;
        bcache.hashbucket[cur].head.next = b;
      }
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;
      acquire(&tickslock);
      b->timestamp = ticks;
      release(&tickslock);
      release(&bcache.hashbucket[cur].lock);
      acquiresleep(&b->lock);
      return b;
    }
    else
    {
      // 在当前散列桶中未找到，则直接释放自旋锁
```

```
    if(i != cur)
      release(&bcache.hashbucket[i].lock);
  }
```

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。在当前散列桶中未找到，则直接释放锁

```
if(!holdingsleep(&b->lock))
    panic("brelse");

  int cur=HASH(b->blockno);
  releasesleep(&b->lock);
  acquire(&bcache.hashbucket[cur].lock);
  b->refcnt--;
  acquire(&tickslock);
  b->timestamp = ticks;
  release(&tickslock);

  release(&bcache.hashbucket[cur].lock);
```

在 brelse 函数中，我们更新时间戳和缓存块的引用次数 refcnt

先调用 holdingsleep()查询是否已经获取到该睡眠锁，确保有带锁后，才调用 releasesleep()释放该锁

同时在每一步操作中，考虑好锁的使用以及互斥，在想访问某个散列桶时，要通过 holding 返回的此桶的自旋锁状态来访问，在 brelse 时，也要等到缓存块的睡眠锁被释放才能用，用 holdingsleep，每次操作时也要先获取锁再释放锁。

三、 实验结果截图

```
make[1]: Leaving directory '/home/students/200110529/xv6-labs-2020'
== Test running kalloctest ==
$ make qemu-gdb
(121.1s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (12.6s)
== Test running bcachetest ==
$ make qemu-gdb
(10.8s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (165.8s)
== Test time ==
time: OK
Score: 70/70
```