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

实验设计报告

开课学期：2024 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：11 月 1 日 地点：T2210

学生班级：22 级 4 班

学生学号：220110430

学生姓名：吴梓滔

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2024 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？

内存分配器是内核代码中的 kalloc.c 所定义的内容，为用户程序分配内存的。它的作用是管理空闲物理内存空间，根据上层的申请和释放调用，进行相应的分配和释放内存块。

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？

```
struct {
        struct spinlock lock;
        struct run *freelist;
} kmem;
struct run {
        struct run *next;
};
```

数据结构是结构体 kmem，由一个自旋锁 lock 和 run 类型链表构成。结构体 run 类型是链表的单位。它实际上会用来存储内存块。

kinit()函数：初始化 kmem 结构。包括初始化锁和初始化存储页链表。只在 main()函数调用，且只由 cpu0 执行。

freerange()函数：初始化 kmem 中的空闲页链表。他会对起始地址 pa_start 到末尾地址 pa_end 中每一个空闲内存页初始化成 run 类型的链表项，添加到链表中。只在 kinit 中被调用。

kfree()函数：读入指定地址，将指定地址的这个页清空，插入空闲页链表中。即调用处释放某个内存页，释放后成为空闲页。

kalloc()函数：将空闲页链表中的一个项从链表中取出，并返回 void 型指针，即将这个空闲页分配给调用者去使用。

c. 为什么指导书提及的优化方法可以提升性能？

因为原本性能低的原因是多核心实际上同时只有一个在进行内存分配，其他等待，优化后多个核心基本可以同时进行内存分配。原本情况因为空闲内存页是一条链表，由所有核心共享，在操作时必须互斥，所以只有一个核心在做分配，其他核心等待；优化方法本质上就是拆散空闲内存页链表，给每个核心分配一个空闲页链

表，每个核心一般可以用对应的空闲页链表进行分配，而不需要考虑互斥操作，从而实现真正的并行。

2、 磁盘缓存

a. 什么是磁盘缓存？它的作用是？

磁盘缓存是在内存中做的一个缓冲区，负责在文件系统和外存之间做一个数据块的缓存，由 bio.c 实现的内容。作用是利用计算机程序局部性的原理，加速文件访问的速度，提高性能。

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）。

这么做是为了做一个双向链表，从而实现 LRU 替换策略。缓存会涉及到缓冲区替换的问题，LRU 是好的替换策略，为了实现简易的 LRU 策略，使用双向链表存储缓冲区数据块，越靠近链表头的一般越新，越靠近链表尾的一般越旧。在遍历寻找一个数据块时，从链表头向表尾的顺序遍历，如果前不久使用过该块，可能靠近表头，很快找到；如果没有找到，那么需要找出一个已经没有使用的块替换。已经没有使用的块一般是比较旧的，从链表尾向表头的反向遍历，可能比较快地找到目标数据块。如果没有做双向链表，就不能够实现这个功能。

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。

原本性能低的原因是本来并行的几个进程共享缓冲区管理链表，每次只能有一个核心在操作缓冲区，其他核心等待。使用哈希表，可以将缓冲区链表拆解成若干个链表，把数据块散列映射到各个链表，使得各核心操作缓冲区占用同一个链表的可能性小，实现并行而提高性能。

不能够使用内存分配器的优化方法。对于每个核心而言，内存彼此是独立的，空闲内存里面没有共享的数据，所以内存分配器只需要和核心一一对应。但是对于数据块缓冲区而言，外存的数据是每个核心共享的，任何一个核心都有可能要访问任

何一个数据块。内存分配器的优化思路类似于隔离，把每个核心操作的内存隔离开来，就互斥了；磁盘缓存的优化思路是分散，把集中在一个链的数据块散到多个链表，每个数据块固定地映射在某个链表，从而减少两个核心同时操作一个链表的概率。

二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写

1.kalloc.c 的修改方案

根据 xv6 定义的 cpu 数量 NCPU，创建相应的 kmem 结构数组，命名为 kmems，与 cpu 一一对应。此后运行时，cpu 应调用自己的 id，只操作自己 id 对应的空闲页链表。

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

在改变数据结构后，对初始化操作进行相应的修改。原本的 kinit 是初始化锁，并调用 freerange 来初始化链表。并且 kinit 是只由 cpu0 在 main 函数里调用一次的。现在的数据结构，kmems 数组每个元素里都有一个锁，应该对每一个锁在这里一次性初始化。通过一个循环语句，初始化所有的锁，都命名为“kmem”。

```
for (int i = 0; i < NCPU; i++) {
    initlock(&kmems[i].lock, "kmem");
  }
```

此后，仍调用 freerange 函数来初始化链表，不做修改。对于 freerange 的内容，从简单起见，可以不做修改。根据设计，一个核心在调用自己对应的空闲页链表时，如果没有空闲页，会去其他核心的链表窃取空闲页。如果不改 freerange，那么 cpu0 运行 freerange，会调用 kfree 将所有的内存页初始化在 cpu0 对应的空闲页链表里。之后多个核心并行调度，例如 cpu[i]在没有空闲页时，可以到 cpu0 的链表里面窃取空闲页，之后释放的时候，会释放到 cpu[i]的链表里，这样也能构建出多个链表。因此，freerange 可以不修改，让 cpu0 在初始化时把所有空闲页分配在 kmems[0]的链表里面。

对于 kfree 和 kalloc 两个操作，要根据当前运行的 cpuid 来决定操作。原本的 kmem 结构变成了 kmems[NCPU]数组，思路是获取运行时的 cpuid 为 cid，以 cid 作为索引，将同样的操作改成对 kmems[cid]结构的链表来进行。

在 kfree()函数中，核心步骤修改为：

```
r->next = kmems[cid].freelist;
kmems[cid].freelist = r;
```

在 kalloc 中，需要考虑当前的链表中找不到空闲页的情况，此时是需要到其他有

空闲页的链表中窃取。思路是在做完常规的获取后，检查获得到的 r 是否为空。如果不为空，说明正常获取到了，可以去返回；如果为空，说明没有获取到。此时就需要窃取。窃取的对象简单地从 0 号开始递增寻找。如果在其他的链表里找到了空闲页，便直接将其分配出去。

```
int cid = cpuid(); //  获取 cpuid
  acquire(&kmems[cid].lock);
  r = kmems[cid].freelist;                    // 第一次，在本链表内获取
  if(r)
    kmems[cid].freelist = r->next;
  release(&kmems[cid].lock);

if(!r)
    for (int i = 0; i < NCPU; i++)
      if (cid != i) {
        acquire(&kmems[i].lock);
        r = kmems[i].freelist;
        if(r)
          kmems[i].freelist = r->next;   //从 i 号的空闲页链中尝试 pop 一个
        release(&kmems[i].lock);
        //下面再检查一下 r 有了没有，如果有就 break 出去。没有才循环
        if(r)
          break;
      }
```

此外，在获取 cpuid 以及使用 cpuid 的时候，调用 push_off()和 pop_off()关闭和开启中断。

2.bio.c 修改方案

取哈希函数的模值 13，设计 13 个哈希桶。将 bcache 的结构做响应修改，做 13 个哈希桶，哈希桶中存放链表，同时设置 13 个链表的锁，为每个哈希桶及链表配置一个。此外，要为 bcache 配置一个全局大锁，在替换策略中，如果需要到其他桶取寻找缓冲区块替换时，需要持有大锁。

```
struct {
  // struct spinlock lock;
  struct buf buf[NBUF];       // NBUF=30
   struct spinlock dasuo;        //  全局大锁
  // struct buf head;             NBUCKETS=13
  struct spinlock locks[NBUCKETS];   // 13 个锁,每个哈希桶配一个
  struct buf hashbuckets[NBUCKETS];   //  相当于 13 个 head
} bcache;
```

对于 30 个缓冲区块，在初始化的时候也按照模 13 的哈希函数，映射到对应哈希桶内。对于外存里的数据块，假设数据块的设备号是 dev，区块号是 blockno，那么使用哈希函数(dev+blockno) % 13 来确定对应的哈希桶号。

bcache 的初始化 binit 函数，要构建好每一个哈希桶内的双向链表。双向链表的构建采用头插法，初始先设置每一个链表头的 prev 和 next 都是表头本身；之后遍历每一个缓冲区块（下面称 buf 块），将每一个 buf 块依次用头插法，插入到映射的链表里。

```
for (int i = 0; i < NBUCKETS; i++) {
    bcache.hashbuckets[i].prev = &bcache.hashbuckets[i];
    bcache.hashbuckets[i].next = &bcache.hashbuckets[i];
  }
  for (int i = 0; i < NBUF; i++) {
    b = bcache.buf + i;
    int hashno = hashfn(i);   // 计算哈希值,这意味着 b 应该到哪个哈希桶里
    b->next = bcache.hashbuckets[hashno].next;
    b->prev = &bcache.hashbuckets[hashno];
    initsleeplock(&b->lock, "buffer");      // 初始化 buf 块的锁
    bcache.hashbuckets[hashno].next->prev = b;
    bcache.hashbuckets[hashno].next = b;
  }
```

bget 函数是 bcache 的一个核心操作。它先检查链表中是否有装载指定数据块的 buf 块，若有则返回，若没有则需要准备一个缓冲区块返回，供上一层即 bread 从外存读入。在改进后，bget 不但需要有以上两个步骤，还需要增加在当前链表缺少 buf 块时，到其他的哈希桶链表中挪用 buf 块。

bget 并不是供用户程序调用的函数，用户程序调用 bread 和 bwrite 等，而 bget 是 bread 中调用的一个中间函数，bget 负责取回 buf 块，不论这个 buf 块是否有数据。在功能上，bread 不需要做任何修改。bwrite 将指定的数据块写回外存，也不需要做任何修改。

brelse 将一个 buf 块释放，释放后，这个 buf 块的使用者便减少 1，同时会检查 buf 块的使用者为 0 时，将其插入链表头，等待被替换掉，符合 LRU 策略。

bget 和 brelse 中涉及到一些对应的加锁和解锁，因此要综合考虑 bget 和 brelse 的实现。

对 bget，设计 3 段的遍历查找，第一次是找目标数据块对应的哈希桶，查找是否已经装载在 buf 块中，如果有，将直接返回该 buf 块。如果没有，再进入第二段查找。其中，引用次数 refcnt 的变化为自增 1。因为如果某个数据块已经装载在 buf 块中，说明此前已经有被调用过。无论此时它是有被调用，还是没有被调用，该次的 bget 调用，都意味着这个 buf 块多了一个引用者，因此将 refcnt 自增 1。第一段查找的代码如下所示。

```
acquire(&bcache.locks[hashno]);
  for(b = bcache.hashbuckets[hashno].next; b != &bcache.hashbuckets[hashno]; b
= b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;          // 有用户要用这个缓冲区块，所以引用计数加 1。
      release(&bcache.locks[hashno]);
      acquiresleep(&b->lock);
      return b;
```

```
        }
      }
```

第一段查找如果没有返回，则会开始第二段查找。第二段查找是查找该哈希桶内是否有可以使用的 buf 块。如果有，就可以进行替换，将该 buf 块设置为供目标数据块使用。在这一过程中，采取从链表尾向链表头的遍历方法，找第一个引用次数为 0 的 buf 块，符合 LRU 策略。对于新块在 refcnt 的设置，由于此前哈希桶内没有这个数据块，所以这次 bget 调用是该块的第一次引用，refcnt 设置为 1。由于 bget 是被 bread 调用的，bread 在缓存不命中时，是先用 bget 把 buf 块设置好，再通过系统调用在 bread 里完成的数据读入。因此在这里 valid 设置为 0，直到外层 bread 完成读取后，再设置为 1。第二段查找的代码如下所示。

```
    for(b = bcache.hashbuckets[hashno].prev; b != &bcache.hashbuckets[hashno]; b = b->prev){
      if(b->refcnt == 0) {
        b->dev = dev;
        b->blockno = blockno;
        b->valid = 0;
        b->refcnt = 1;
        release(&bcache.locks[hashno]);
        acquiresleep(&b->lock);
        return b;
      }
    }
```

若这两段查找都没有返回，说明目标数据块映射的哈希桶内既找不到目标块，也找不到可以用来读入目标数据的 buf 块，所以必须从其他哈希桶挪用 buf 块。因此，要离开当前哈希桶，依次在其他哈希桶寻找。首先要注意锁的切换。

此前获取的锁是&bcache.locks[hashno]，在第一段查找之前获取，一直持有到第二段查找结束。在第三段查找开始前，应该先释放，随后获取其他桶的锁。

```
    release(&bcache.locks[hashno]); //  释放锁
```

在第三段查找中，必须注意锁的管理。第二段查找结束释放锁后，当前线程并不持有目标桶的锁，别的线程可以会修改目标桶 hashno 里的内容，有可能其他的并发线程已经完成了相同数据块替换与读入。为了保证跨桶取块时的互斥，在跨桶取块的时候必须持有全局大锁，任何时刻只能有一个进程进行跨桶取块。

在当前进程获取全局大锁后，再获取目标桶的锁，还需先对目标桶进行一次检查，防止在当前进程从释放目标桶锁到再次获取锁的这个期间内，其他进程完成了目标数据块的替换。如果经过检查，目标桶内仍然没有目标数据块，那么就可以到其他哈希桶寻找可替换的 buf 块。

在第三段查找中，从 0 号哈希桶开始依次遍历每个哈希桶，先获取当前哈希桶的锁，再开始在该哈希桶中查找可用 buf 块。此时刻，当前进程按顺序持有三个锁：全局大锁、目标桶 hashno 的锁，当前要查找可替换 buf 块的桶的锁。查找条件和第二段查找类似，在找到后，便将这个可用 buf 块 b 从所在链表移除，回到目标桶 hashno 中，将 b 头插法插入链表中，再返回这个 buf 块。在返回前，需要按顺序释放持有的三个互斥锁。锁管理的代码如下。

```
    acquire(&bcache.dasuo);
    acquire(&bcache.locks[hashno]);
```

```
……
acquire(&bcache.locks[i]);     // 正在跨到第 i 号桶查找可用 buf 块
……
release(&bcache.locks[i]);
release(&bcache.locks[hashno]);
release(&bcache.dasuo);
```

在查找后，进行挪用的代码如下。

```
    // 从其他桶移动到目标桶
        b->next->prev = b->prev;
        b->prev->next = b->next;
        release(&bcache.locks[i]);
        acquire(&bcache.locks[hashno]);   // 获取目标桶锁
        // 更新目标桶链表
        b->next = bcache.hashbuckets[hashno].next;
        b->prev = &bcache.hashbuckets[hashno];
        bcache.hashbuckets[hashno].next->prev = b;
        bcache.hashbuckets[hashno].next = b;
        release(&bcache.locks[hashno]);
        acquiresleep(&b->lock);
        return b;
```

brelse 中，将指定 buf 块的引用次数减少 1。同时检查引用次数是否为 0，若为 0，则要调整 buf 块在链表中的位置。之所以移动到头部，是因为这个 buf 块作为引用次数为 0 的块，是最近期被释放的，所以要远离尾端。核心代码如下所示。

```
releasesleep(&b->lock);
  acquire(&bcache.locks[hashno]);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.hashbuckets[hashno].next;
    b->prev = &bcache.hashbuckets[hashno];
    bcache.hashbuckets[hashno].next->prev = b;
    bcache.hashbuckets[hashno].next = b;
  }
  release(&bcache.locks[hashno]);
```

bpin 和 bunpin 也要做相应更改，对于指定的 buf 块 b，先计算其哈希映射的值，用于获取其所在哈希桶的锁。在获取锁的情况下，对 b 的 refcnt 做相应更改即可。

三、 实验结果截图

请给出 kalloctest、bcachetest 、usertests 及 make grade 的测试结果。

kalloctest:

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 51002
lock: kmem: #fetch-and-add 0 #acquire() 185911
lock: kmem: #fetch-and-add 0 #acquire() 196159
lock: bcache: #fetch-and-add 0 #acquire() 10
lock: bcache: #fetch-and-add 0 #acquire() 2
lock: bcache: #fetch-and-add 0 #acquire() 4
lock: bcache: #fetch-and-add 0 #acquire() 6
lock: bcache: #fetch-and-add 0 #acquire() 6
lock: bcache: #fetch-and-add 0 #acquire() 16
lock: bcache: #fetch-and-add 0 #acquire() 284
lock: bcache: #fetch-and-add 0 #acquire() 4
lock: bcache: #fetch-and-add 0 #acquire() 4
lock: bcache: #fetch-and-add 0 #acquire() 4
--- top 5 contended locks:
lock: proc: #fetch-and-add 25712 #acquire() 136625
lock: virtio_disk: #fetch-and-add 13324 #acquire() 57
lock: proc: #fetch-and-add 7419 #acquire() 136704
lock: proc: #fetch-and-add 4773 #acquire() 136704
lock: proc: #fetch-and-add 1489 #acquire() 136629
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
.....
test2 OK
```

bcachetest:

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32939
lock: kmem: #fetch-and-add 0 #acquire() 18
lock: kmem: #fetch-and-add 0 #acquire() 128
lock: bcache: #fetch-and-add 0 #acquire() 4168
lock: bcache: #fetch-and-add 0 #acquire() 6182
lock: bcache: #fetch-and-add 0 #acquire() 6178
lock: bcache: #fetch-and-add 0 #acquire() 6324
lock: bcache: #fetch-and-add 0 #acquire() 6320
lock: bcache: #fetch-and-add 0 #acquire() 6464
lock: bcache: #fetch-and-add 0 #acquire() 6306
lock: bcache: #fetch-and-add 0 #acquire() 4530
lock: bcache: #fetch-and-add 0 #acquire() 4228
lock: bcache: #fetch-and-add 0 #acquire() 3642
lock: bcache: #fetch-and-add 0 #acquire() 4216
lock: bcache: #fetch-and-add 0 #acquire() 2520
lock: bcache: #fetch-and-add 0 #acquire() 4120
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 145496 #acquire() 1119
lock: proc: #fetch-and-add 36703 #acquire() 76625
lock: proc: #fetch-and-add 28020 #acquire() 76310
lock: proc: #fetch-and-add 21976 #acquire() 76274
lock: proc: #fetch-and-add 11833 #acquire() 76294
tot= 0
test0: OK
start test1
test1 OK
```

usertests:

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3240
            sepc=0x000000000000573e stval=0x000000000000573e
usertrap(): unexpected scause 0x000000000000000c pid=3241
            sepc=0x000000000000573e stval=0x000000000000573e
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate:
```

```
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test dirtest: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6215
            sepc=0x0000000000002198 stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6216
            sepc=0x0000000000002198 stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6217
            sepc=0x0000000000002198 stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6218
            sepc=0x0000000000002198 stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6219
            sepc=0x0000000000002198 stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6220
            sepc=0x0000000000002198 stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6221
            sepc=0x0000000000002198 stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6222
            sepc=0x0000000000002198 stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6223
            sepc=0x0000000000002198 stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6224
```

```
            sepc=0x0000000000002198 stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x0000000000002198 stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x0000000000002198 stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6266
            sepc=0x0000000000004256 stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6270
            sepc=0x0000000000002308 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 forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

make grade:

```
make[1]: Leaving directory '/home/students/220110430/xv6-oslab24-hitsz'
== Test running kalloctest ==
$ make qemu-gdb
(122.2s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (12.3s)
== Test running bcachetest ==
$ make qemu-gdb
(9.9s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (158.7s)
== Test time ==
time: OK
Score: 70/70
220110430@comp4:~/xv6-oslab24-hitsz$
```