华东师范大学数据学院上机实践报告

课程名称： 操作系统　　**年级：**2022 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**李芳

上机实践名称： Locks　　**学号：**10214602404　　**上机实践日期：**2024.05.09

一、实验目的

- 优化block cache争用
- 改进 xv6 操作系统的缓冲区缓存（buffer cache），以减少在多进程频繁使用文件系统时对 `bcache.lock` 的争用。

二、实验内容

- 原始版本的buffer cache由一个大锁bcache.lock保护，限制了并行运行的效率，且使用的是双链表的管理方式。
- 修改 `kernel/bio.c` 文件：放弃双链表的管理方式，利用**hash bucket**思想修改缓冲区缓存，以减少对 `bcache.lock` 的争用。**使用哈希表在缓存中查找块号block number，哈希表的每个哈希桶都有一个锁。**
- 删除所有缓冲区的列表（如 `bcache.head` 等），改为使用缓冲区上次使用的时间戳（即使用 `kernel/trap.c` 中的 `ticks`）。通过这种改变，`brelse` 不需要获取 `bcache` 锁，`bget` 可以基于 `ticks` 选择最近最少使用的块。
- 运行 `bachetest` 时，bcache中所有锁的acquire循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数之和应该为0，本实验允许总和小于500。
- 保持每个块最多只有一个缓存副本的不变性。
- 最终运行 `bcachetest` 和 `usertests` 时保持所有测试通过。实验结果通过如下图

三、使用环境

Vscode & xv6

四、实验过程及结果

- **实验过程**
  - buf.h

要想在 bget 函数中找到最近最少用的缓存块，就需要对 buf 结构增加一个计时参数，

即 uint lastuse 存放 ticks：

```
kernel > C buf.h > ...
1   struct buf {
2     int valid;   // has data been read from disk?
3     int disk;    // does disk "own" buf?
4     uint dev;
5     uint blockno;
6     struct sleeplock lock;
7     uint refcnt;
8     struct buf *prev; // LRU cache list
9     struct buf *next;
10    uchar data[BSIZE];
11
12    uint lastuse; //最近使用时间片
13
14  };
```

- bio.c
  - 提前声明变量、函数、结构体：因为要提前设定好装缓存块的哈希桶的组数和每组数量，并且引入外部变量时间片，同时为了方便找到缓存块号映射到的哈希桶号，提前写好映射函数 hash()。最后编写整个缓存管理代码的基础----哈希桶结构体。

```
24
25  #define BUCKETSIZE 13   // 桶数量
26  #define BUFFERSIZE 5    // 桶大小
27  extern uint ticks;      // 引入时间片
28
29  // 哈希映射函数
30  int hash(uint blockno)
31  {
32    return blockno % BUCKETSIZE;
33  }
34
35  struct
36  {
37    struct spinlock lock;        // 每个桶一个自旋锁
38    struct buf buf[BUFFERSIZE]; // 每个桶放5个缓冲块
39  } bcachebucket[BUCKETSIZE];   // 13个桶结构
40
```

  - binit()函数：对新的哈希桶缓存区进行初始化，每组桶初始化一个自旋锁，桶中每个块初始化一个睡眠锁。

```
72  void binit(void)
73  {
74    for (int i = 0; i < BUCKETSIZE; i++)
75    {
76      initlock(&bcachebucket[i].lock, "bcachebucket"); // 一个桶初始一个自旋锁
77      for (int j = 0; i < BUFFERSIZE; i++)
78      {
79        initsleeplock(&bcachebucket[i].buf[j].lock, "buffer"); // 一个块初始一个睡眠锁
80      }
81    }
82  }
```

◆ bget()函数：

先根据块号，获取对应桶号，并且获取该桶的自旋锁。然后分为三种基本情况：第一，遍历该桶中的每个块，之后去检查块设备号和块号：如果对应，则增加使用进程数，更新使用时间片，先释放桶的自旋锁，再获取该正确块的睡眠锁，就可以直接返回该块地址；如果没找到，就进入第二种情况。

```
23  static struct buf *
24  bget(uint dev, uint blockno)
25  {
26    struct buf *b;
27
28    // 得到哈希桶号，自旋锁上对应桶
29    int bucketno = hash(blockno);
30    acquire(&bcachebucket[bucketno].lock);
31
32    // 找到对应设备号和块号的缓存块
33    for (int i = 0; i < BUFFERSIZE; i++)
34    {
35      b = &bcachebucket[bucketno].buf[i];
36      if (b->dev == dev && b->blockno == blockno)
37      {
38        b->refcnt++;         // 增加使用进程数
39        b->lastuse = ticks; // 更新时间片
40        release(&bcachebucket[bucketno].lock);
41        acquiresleep(&b->lock);
42        return b;
43      }
44    }
```

第二，采用最近最少用策略，对空闲块进行查找和覆盖：遍历该组哈希桶中的每一块，找到时间片最小的一个空闲块（证明很久都没用、没更新了）；如果没找到，就进入第三种情况。

```
46  // 没找到已存在的对应缓存块，最近最少用策略找空闲块覆盖
47  uint flag = 0xffffffff; // 获取最近时间片
48  int index = -1;         // 获取块索引
49  for (int i = 0; i < BUFFERSIZE; i++)
50  {
51    b = &bcachebucket[bucketno].buf[i];
52    if (b->refcnt == 0 && b->lastuse < flag)
53    { // 更新空闲并且时间片小的块
54      flag = b->lastuse;
55      index = i;
56    }
57  }
58
```

第三，去邻居桶中找合适空闲块。先遍历非映射桶的邻居哈希桶，获取桶的自旋锁，遍历其中所有的块，查找是否有最早没用过的空闲块，有的话，就更新索引。邻居桶中找到符合空闲块的情况下，然后对一系列块变量进行初始化，完成后释放桶自旋锁，获取块睡眠锁，直接返回块地址。这种情况下最后要释放邻居桶的自旋锁；

最后，假设所有情况都不符合，就把映射桶的自旋锁释放，然后抛出 panic 异常。

```
// 如果在当前桶中没有找到合适的空闲缓冲区,尝试从邻居桶中寻找空闲缓冲区
if (least_idx == -1)
{
  for (int neighbor_bucket = (bucket + 1) % BUCKETSIZE;neighbor_bucket != bucket;neighbor_bucket = (neighbor_bucket + 1) % BUCKETSIZE)
  {
    acquire(&bcachebucket[neighbor_bucket].lock);
    for (int i = 0; i < BUFFERSIZE; i++)
    {
      b = &bcachebucket[neighbor_bucket].buf[i];
      if (b->refcnt == 0 && b->lastuse < flag)
      {
        flag = b->lastuse;
        least_idx = i;
        break;
      }
    }
    // 如果在邻居桶中找到空闲缓冲区，进行相应的初始化操作
    if (least_idx != -1)
    {
      b = &bcachebucket[neighbor_bucket].buf[least_idx];
      b->dev = dev;
      b->blockno = blockno;
      b->lastuse = ticks;
      b->valid = 0;
      b->refcnt = 1;
      release(&bcachebucket[neighbor_bucket].lock);
      acquiresleep(&b->lock);
      return b;
    }
    release(&bcachebucket[neighbor_bucket].lock);
  }

  // 如果遍历所有桶后依然没有找到空闲缓冲区,放锁，抛出panic
  release(&bcachebucket[bucket].lock);
  panic("bget: no used buffer for recycle");
}
```

- brelse()函数：

首先要判断是否获取到该块的睡眠锁：如果无法获取锁，也就是锁被占用，证明这一块还在使用中，就不能被释放，此时抛出一个 panic；

获取到睡眠锁了，就证明已经未使用了，就可以进行设置。首先释放掉睡眠锁，那么这块就能再重新被启用。然后，根据块号获取哈希桶号，获取桶自旋锁。之后，使用进程数减一，没有使用的进程时，更新时间片，然后释放桶的自旋锁。

```
28  void brelse(struct buf *b)
29  {
30    if (!holdingsleep(&b->lock))
31      panic("brelse");
32
33    releasesleep(&b->lock);
34
35    int bucketno = hash(b->blockno);
36    acquire(&bcachebucket[bucketno].lock);
37    b->refcnt--;
38    if (b->refcnt == 0)
39    { //没有正在使用的进程时，要更新时间片
40      b->lastuse = ticks;
41    }
42    release(&bcachebucket[bucketno].lock);
43  }
```

- bpin()函数 & bunpin()函数：在原始函数的基础上只需要在对锁进行操作前，哈希对应桶号，并且获取、释放桶的自旋锁就可以。

```
52  void bpin(struct buf *b)
53  {
54    int bucketno = hash(b->blockno);
55
56    acquire(&bcachebucket[bucketno].lock);
57    b->refcnt++;
58    release(&bcachebucket[bucketno].lock);
59  }
```

```
61  void bunpin(struct buf *b)
62  {
63    int bucketno = hash(b->blockno);
64
65    acquire(&bcachebucket[bucketno].lock);
66    b->refcnt--;
67    release(&bcachebucket[bucketno].lock);
68  }
```

- **实验结果**
  - 运行 bcachetest:
    - bcache 中所有锁的 acquire 循环迭代次数接近于零。
    - 块缓存中涉及的所有锁的计数之和为 0。

```
hart 1 starting
hart 2 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 33027
lock: bcachebucket: #test-and-set 0 #acquire() 4120
lock: bcachebucket: #test-and-set 0 #acquire() 4734
lock: bcachebucket: #test-and-set 0 #acquire() 9004
lock: bcachebucket: #test-and-set 0 #acquire() 6174
lock: bcachebucket: #test-and-set 0 #acquire() 6174
lock: bcachebucket: #test-and-set 0 #acquire() 6174
lock: bcachebucket: #test-and-set 0 #acquire() 6174
lock: bcachebucket: #test-and-set 0 #acquire() 6194
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 11492507 #acquire() 1119
lock: proc: #test-and-set 3470119 #acquire() 197981
lock: proc: #test-and-set 2589458 #acquire() 197980
lock: proc: #test-and-set 2437579 #acquire() 197980
lock: proc: #test-and-set 2414157 #acquire() 197980
tot= 0
test0: OK
start test1
test1 OK
```

  - 运行 usertests: 所有测试通过

```
问题    输出    调试控制台    终端    端口

            sepc=0x0000000000002158 stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6292
            sepc=0x0000000000002158 stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6293
            sepc=0x0000000000002158 stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6305
            sepc=0x00000000000044fc stval=0x0000000000012000
OK
test sbrkarg: OK
test sbrklast: OK
test sbrk8000: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6311
            sepc=0x0000000000002376 stval=0x000000000000fb50
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 killstatus: 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
```

五、总结

- **实验复盘**

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

- ✓ 竞争条件：多个进程并发地修改某个共享数据结构，并因此产生不确定的结果。
- ✓ 临界区：进程并发地修改共享数据结构的代码段
- ✓ 互斥锁：这次实验中的锁是一种互斥原语，一次只有一个 CPU 能够获得锁，给一个临界区上锁，意味着一次只能有一个 CPU 在临界区中执行。
  - ■ 锁的作用：避免更新丢失、使多步操作变为原子性的操作、维持一些规则或特性的不变性等等。
  - ■ 互斥锁解决临界区问题要求：互斥、进步以及有限等待/无饥饿
  - ■ 死锁：

    引起死锁，需要四个条件同时成立：
    - ◆ 互斥：至少有一个资源处于非共享模式，即进程对于需要的资源进行互斥的访问。如果另一进程申请该资源，则申请进程应等到该资源释放为止。
    - ◆ 占有并等待：进程至少占有一个资源，并等待其它资源，而该资源被其它进程所占有。
    - ◆ 非抢占：进程获得的资源不能被抢占，亦即锁、信号量等不能被抢占。
    - ◆ 循环等待：进程之间存在一个环路，环路上每个进程都额外持有一个资源，而这个资源又是下一个进程要申请的。

      预防死锁，破除四个必须条件中的一个即可：
    - ◆ 循环等待->全序和偏序的上锁顺序、根据锁的地址作为获取锁的顺序；
    - ◆ 占有并等待->原子抢锁（每个进程执行前，先申请并成功获得所需所有资源）；
    - ◆ 非抢占->trylock 接口（如果一个进程持有资源并申请另一个不能立即分配的资源，并且即将进入等待状态，那么它现在持有的资源都可被抢占）；
    - ◆ 互斥->利用硬件指令来构造无等待 wait-free 的数据结构。

同步工具除了锁，还有条件变量、信号量，都是基于软件的解决临界区问题的方式，还有硬件方法：

- ✓ 硬件的原子指令：Test and Set（测试并设置）、Compare and Swap（比较并交换）、Fetch and add（获取并增加）、Load-Linked and Store-Conditional（链接加载和条件式存储）等，可以支持上面提到的同步工具实现。下面用 C 语言演示逻辑：
  - ■ Test and Set：返回 old_ptr 指向的旧值 old，同时更新成新值 new。

```
int TestAndSet(int *old_ptr, int new){
  int old = *old_ptr;  // 获取旧的值
  *old_ptr = new;      // 存储新的值
  return old;          // 返回旧的值
}
```

- Compare and Swap：检测 ptr 指向的值是否和 expected 相等，返回该内存地址的实际值 actual。如果是，就更新 ptr 的值为 new，否则什么也不更新。

```
int CompareAndSwap(int *ptr, int expected, int new){
  int actual = *ptr;
  if(actual == expected)
    *ptr = new;
  return actual;
}
```

- Fetch and add：特定地址的值自增 1，并且返回旧值

```
int FetchAndAdd(int *ptr){
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

- Load-Linked and Store-Conditional：Load-Linked 从内存中取出值存入一个寄存器；Store-Conditional 是只有上一次加载的地址 ptr 在这期间都没有被更新时，返回 1 表示成功，同时更新经过 Load-Linked 得到的 ptr 中的值；如果失败，返回 0 且不会更新 ptr 中的值。

```
int LoadLinked(int *ptr){
  return *ptr;
}

int StoreConditional(int *ptr, int value){
  if(在上一次加载ptr之后，期间没有对ptr的更新){
    *ptr = value;
    return 1;  // 成功
  }else{
    return 0;  // 失败
  }
}
```

✓ 内存屏障：强制内存的更新对所有 CPU 可见。

- 当执行内存屏障指令时，系统确保，在内存屏障之前所有的 load 和 store，都会在内存屏障之后的 load 或 store 执行前完成。

✓ 原子变量：提供互斥机制，保证在更新该变量时不会出现竞争条件。

本次实验基本只涉及到了两种锁的操作：自旋锁和睡眠锁，基本操作大概为 init，acquire，release，hold 等。

- 自旋锁（Spining Lock）：
  - 实现在 kernel/spinlock.c & spinlock.h 文件中，里面用到了 RISC-V 的 test_and_set 指令，还利用了 RISC-V 的内存屏障指令以及中断的控制（push_off & pop_off）来实现自旋锁。
  - 自旋锁不满足有限等待的要求，可能会导致某些线程饥饿。忙等会持续消耗 CPU 资源，因此单 CPU 的情况，性能开销相当大；但是，在多 CPU 上，尤其是当线程数大于 CPU 数时，或者临界区较短时，自旋锁相对合适。
- 睡眠锁（Sleep Lock）：
  - 实现在 kernel/sleeplock.c & sleeplock.h 文件中，采用了自旋锁和真正的睡眠锁相结合的方式。睡眠锁允许中断开放，中断处理程序不能使用睡眠锁（会导致死锁）；另外，因为睡眠锁会让出 CPU，所以不能在自旋锁保护的临界区中使用睡眠锁，但可以在睡眠锁保护的临界区中使用自旋锁。
  - 睡眠锁可以避免大量 CPU 周期的浪费，但缺点是来回切换进程会有较大的上下文切换开销。

Xv6 操作系统中还有更多的锁操作，类似下图：

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

由于并发任务和并发问题，会产生锁的争夺，所以设计代码时，减少争夺现象可以提高运行处理效率，也

就是这次实验的一个考察点：先采用粗粒度（Coarse-grained）的上锁方式，对要保护的临界区上一把大锁，再在保证并发正确性的情况下，逐步地拆解这些数据结构使用细粒度（Fine-grained）的上锁方式。

涉及其他代码文件解析：

Kernel/bio.c 和 Kernel/buf.h 是 xv6 操作系统中负责管理缓冲区缓存的代码文件。缓存的缓冲区用于存储从磁盘读取的数据块，从而减少磁盘 I/O 操作的频率。

- Kernel/bio.c

  定义缓冲区缓存结构体 buf 以及相关数据：

  int valid;缓冲区是否包含有效的数据、int disk;是否被磁盘"拥有"、uint dev;设备号、uint blockno;块号、struct sleeplock lock;睡眠锁，用于同步、uint refcnt;引用计数、struct buf *prev;LRU 缓存列表的前一节点、struct buf *next; LRU 缓存列表的后一节点、uchar data[BSIZE];缓冲区的数据。添加的 uint lastuse;最近使用的时间片

- Kernel/buf.h

  实现缓存的初始化、获取、释放和同步等功能：

  - 修改前：使用双向链表管理缓存，实现 LRU 缓存管理策略。
    - 先声明双向链表结构：lock（保护缓存区的自旋锁）、buf（固定大小的缓冲区数组）、head（双向链表的头结点）
    - Binit()函数：初始化缓存系统，包括缓存数组和双向链表。然后将所有缓冲区插入到链表中，并初始化每个缓冲区的睡眠锁。
    - Bget()函数：查找缓存区，如果已经缓存则返回该缓冲区；如果未缓存，按照 LRU 策略从双向链表尾部查找一个空闲缓冲区进行复用；获取到缓冲区后，锁定该缓冲区并返回。
    - Brelse()函数：释放缓冲区，更新引用计数；如果没有进程引用该缓冲区，将其移动到双向链表的头部，表示最近使用。
    - Bpin()和 bunpin()函数：增加或减少缓冲区的引用计数，用于管理缓冲区的生命周期。

- **实验总结**

本次实验主要对 xv6 操作系统中有关缓冲区管理代码相关结构进行优化，将双向链表改为哈希桶结构，根据实验要求来对应操作完成所有代码修改后，通过了最后的 test。这节课不仅让我熟悉了 xv6 中有关锁的一些操作原理以及知识，同时还帮我复习了课堂上学习的实现互斥的理论知识，收获很大。