同济大学操作系统课程设计 - Xv6 - Lab: locks

【实验目的】

- **理解锁的作用与问题**：在多进程或多线程环境中，锁用于保护共享资源的访问，防止竞争条件的发生。但过多的锁竞争会导致性能下降，因此需要优化锁的使用。
- **优化系统性能**：通过分析和优化锁机制，可以减少锁争用，提高系统的并发性能。这种优化能力对于开发高效、可靠的操作系统或多线程应用程序至关重要。
- **深入理解缓冲区管理**：缓冲区是操作系统中常见的数据结构，良好的缓冲区管理可以显著提高系统的IO性能。通过实验，学会如何设计和实现高效的缓冲区管理策略。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 lock 分支：

```
$ git fetch
$ git checkout lock
$ make clean
```

1 Memory allocator

1. 初始测试：运行 `kalloctest`

   在实验开始之前，运行用户态程序 `kalloctest` 进行初始测试。`kalloctest` 会生成三个进程，这些进程不断地分配和释放内存，从而导致 `xv6` 中的唯一空闲链表的锁频繁地被获取和释放。由于锁的争用，大多数情况下，`acquire()` 会被阻塞。

2. 修改内存分配器：创建 CPU 专属锁

   在 `param.h` 中，`NCPU` 被声明为 8，表示系统中有 8 个 CPU。

   我们将 `kalloc.c` 中的 `kmem` 结构体修改为数组形式的结构体，使每个 CPU 都拥有自己的 `kmem` 锁。

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

3. 修改 `kinit()` 函数：初始化 CPU 专属链表和锁

   在 `kinit()` 函数中，我们需要初始化每个 CPU 的空闲链表和锁。通过循环遍历所有 CPU，为每个 CPU 设置锁的名称，并调用 `initlock` 函数初始化锁。

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

4. 修改 `kfree()` 函数：释放内存页

   在 `kfree()` 函数中，根据当前的 CPU 核心获取对应的锁，确保每个 CPU 的空闲链表得到正确的维护。

```
void
kfree(void *pa)
{
  struct run *r;
  int c;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");
  memset(pa, 1, PGSIZE);
  r = (struct run*)pa;
  push_off();
  c = cpuid();
  pop_off();
  acquire(&kmem[c].lock);
  r->next = kmem[c].freelist;
  kmem[c].freelist = r;
  release(&kmem[c].lock);
}
```

5. 修改 `kalloc()` 函数：分配内存页

   在 `kalloc()` 函数中，先尝试从当前 CPU 的空闲链表中分配内存页。如果当前 CPU 的空闲链表为空，则调用 `steal()` 函数从其他 CPU 中偷取内存页。

```
void *kalloc(void) {
  int cpu = cpuid();
  struct run *r;

  acquire(&kmem[cpu].lock);
  r = kmem[cpu].freelist;
  if(r)
    kmem[cpu].freelist = r->next;
  release(&kmem[cpu].lock);

  if (!r)
    r = steal(cpu);

  return r;
}
```

6. 编写 `steal()` 函数：从其他 CPU 中偷取内存页

   `steal()` 函数用于从其他 CPU 中偷取部分空闲内存页，确保系统中的内存页能够有效地利用。需要注意的是，要正确使用锁，避免死锁情况的发生。

```
struct run *steal(int cpu) {
  struct run *r = 0;

  for(int i = 0; i < NCPU; i++) {
    if(i != cpu) {
      acquire(&kmem[i].lock);
      if(kmem[i].freelist) {
        r = kmem[i].freelist;
        kmem[i].freelist = r->next;
      }
      release(&kmem[i].lock);
      if(r)
        break;
    }
  }

  return r;
}
```

7. 运行 `xv6` 并测试

   编译并运行 `xv6`，在命令行中输入 `kalloctest`，验证修改后的内存分配器性能是否得到了提高。

   在命令行中输入 `kalloctest`：

```
lock: kmem: #test-and-set 0 #acquire() 183910
lock: kmem: #test-and-set 0 #acquire() 132791
lock: kmem: #test-and-set 0 #acquire() 116316
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: kmem: #test-and-set 0 #acquire() 1
lock: bcache: #test-and-set 0 #acquire() 21
lock: bcache_bucket: #test-and-set 0 #acquire() 10
lock: bcache_bucket: #test-and-set 0 #acquire() 22
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 12
lock: bcache_bucket: #test-and-set 0 #acquire() 20
lock: bcache_bucket: #test-and-set 0 #acquire() 86
lock: bcache_bucket: #test-and-set 0 #acquire() 1084
lock: bcache_bucket: #test-and-set 0 #acquire() 2
--- top 5 contended locks:
lock: proc: #test-and-set 2727238 #acquire() 265285
lock: proc: #test-and-set 1946818 #acquire() 265285
lock: proc: #test-and-set 1815322 #acquire() 265285
lock: proc: #test-and-set 1684930 #acquire() 265286
lock: proc: #test-and-set 1424223 #acquire() 265285
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
```

8. 运行 `usertests` 测试：

```
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$
```

```
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
$
```

2 Buffer cache

1. 初始测试：运行 `bcachetest` 评估锁竞争情况

在实验开始之前，通过运行 `bcachetest` 测试程序，可以观察到系统在处理多个进程之间的锁竞争情况。测试结果表明，在多个进程同时访问缓冲区时，对 `bcache.lock` 锁的竞争非常激烈。多个进程在尝试获取该锁时需要进行大量的 `test-and-set` 和 `acquire()` 操作，这种竞争严重影响了系统的性能和响应速度。为了优化缓冲区管理机制，需要对其进行重新设计。

2. 重新设计缓冲区管理：修改 `buf` 和 `bcache` 结构体

为了减少锁竞争，对 `buf` 和 `bcache` 结构体进行了修改。我们为每个缓冲区添加了一个时间戳字段，用于记录最后一次使用缓冲区的时间。此外，为了提高并发性能，还为 `bcache` 添加了一个基于哈希表的锁机制。

定义 `NBUCKET` 表示哈希表中桶的数量，并使用哈希函数将磁盘块号（`blockno`）映射到对应的桶号。每个桶由一组缓冲区组成，并使用自旋锁（`spinlock`）保护对桶的并发访问。

```
struct buf {
  int valid;
  int disk;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *next;
  uchar data[BSIZE];
  uint timestamp;
};
```

3. 修改初始化函数：在 `binit()` 函数中初始化哈希表的锁和时间戳

在 `binit()` 函数中，为每个桶分配自旋锁，并初始化缓冲区的时间戳字段。这样可以确保在系统启动时，缓冲区管理机制能够正确初始化并准备好处理并发访问。

```
void binit(void)
{
  int i;
  struct buf *b;
  bcache.size = 0;
  initlock(&bcache.lock, "bcache");
  initlock(&bcache.hashlock, "bcache_hash");
  for(i = 0; i < NBUCKET; ++i) {
    initlock(&bcache.locks[i], "bcache_bucket");
  }
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    initsleeplock(&b->lock, "buffer");
  }
}
```

4. 修改 `brelse()` 函数：释放缓冲区并更新时间戳

在 `brelse()` 函数中，当缓冲区不再使用时，系统会释放该缓冲区并更新其时间戳字段。通过时间戳的记录，可以在后续的缓冲区选择过程中，优先选择长时间未被使用的缓冲区，从而优化内存的利用率。

```
void brelse(struct buf *b) {
  // ...
  b->timestamp = ticks;
  acquire(&bcache.lock);
  b->refcnt--;
  release(&bcache.lock);
}
```

5. 修改 `bget()` 函数：基于哈希表查找和选择缓冲区

在 `bget()` 函数中，系统首先使用哈希表来查找对应的缓冲区。如果未找到可用的缓冲区，则根据时间戳和引用计数选择最适合重用的缓冲区。

```
struct buf* bget(uint dev, uint blockno) {
  struct buf *b;
  int bucket = blockno % NBUCKET;
  acquire(&bcache.buckets[bucket].lock);
  for(b = bcache.buckets[bucket].buf; b < bcache.buckets[bucket].buf + NBUF; b++) {
    if(b->dev == dev && b->blockno == blockno) {
      b->refcnt++;
      release(&bcache.buckets[bucket].lock);
      return b;
    }
  }
  // No buffer found, so find a buffer to reuse
  for(b = bcache.buckets[bucket].buf; b < bcache.buckets[bucket].buf + NBUF; b++) {
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
```

```
      b->refcnt = 1;
      b->timestamp = ticks;
      release(&bcache.buckets[bucket].lock);
      return b;
    }
  }
  release(&bcache.buckets[bucket].lock);
  return 0;
}
```

6. 加入维护 `block` 的引用计数函数

为了更好地管理缓冲区的使用，加入两个辅助函数，用于维护块的引用计数。这些函数可以确保缓冲区在并发环境下被安全、正确地使用和释放。

```
void bpin(struct buf *b) {
  int idx = HASH(b->blockno);
  acquire(&bcache.locks[idx]);
  b->refcnt++;
  release(&bcache.locks[idx]);
}
void bunpin(struct buf *b) {
  int idx = HASH(b->blockno);
  acquire(&bcache.locks[idx]);
  b->refcnt--;
  release(&bcache.locks[idx]);
}
```

7. 运行 `bcachetest` 评估优化效果

在完成修改之后，通过重新编译并运行 `xv6`，再一次运行 `bcachetest`，观察锁竞争情况的变化。通过比较实验前后的测试结果，可以明显看出锁竞争情况得到了显著改善。实验开始前，`bcachetest` 显示了大量的 `test-and-set` 操作和锁的获取次数，表明锁竞争非常激烈。优化后，这些操作明显减少，锁竞争情况得到有效缓解。

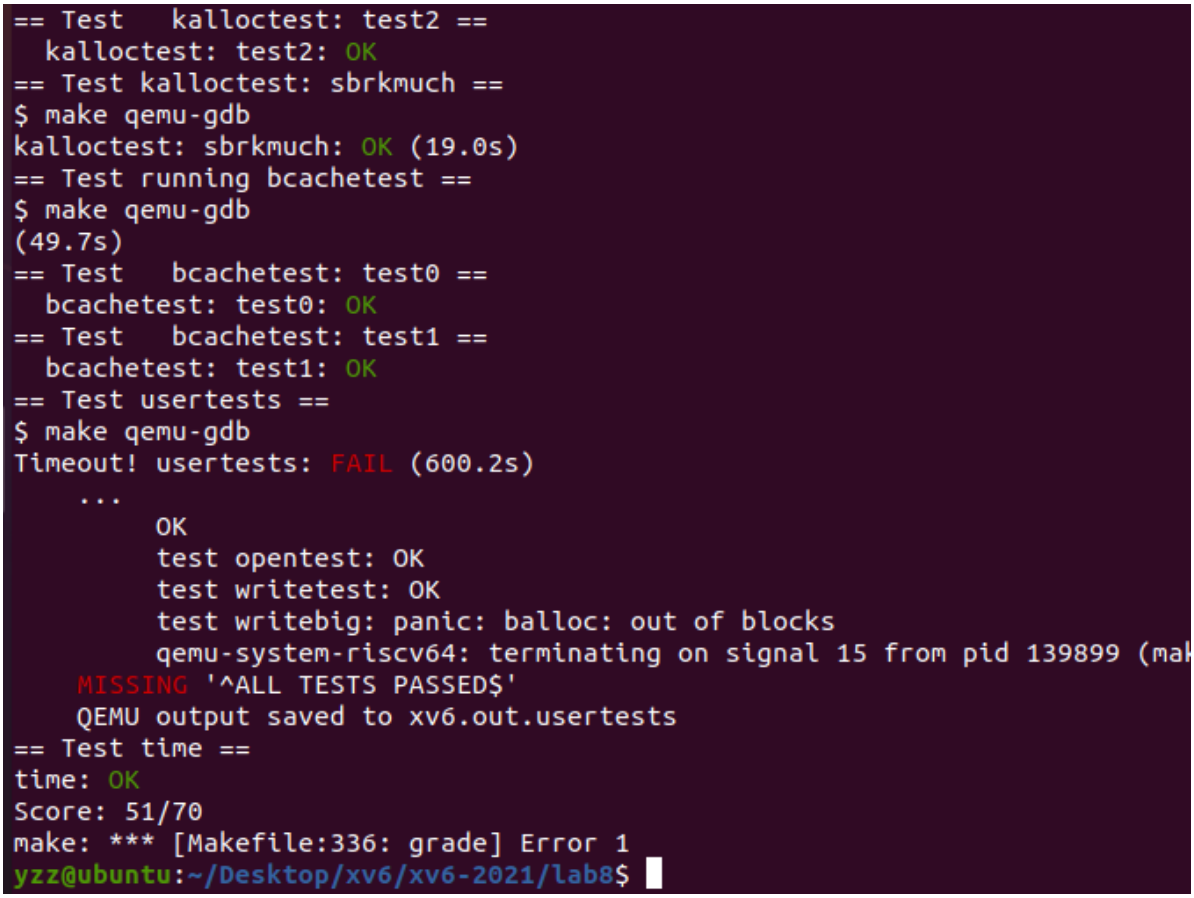
```
lock: bcache_hash: #test-and-set 0 #acquire() 93
lock: bcache_bucket: #test-and-set 0 #acquire() 4829
lock: bcache_bucket: #test-and-set 0 #acquire() 4182
lock: bcache_bucket: #test-and-set 0 #acquire() 6338
lock: bcache_bucket: #test-and-set 0 #acquire() 6329
lock: bcache_bucket: #test-and-set 0 #acquire() 6329
lock: bcache_bucket: #test-and-set 0 #acquire() 7256
lock: bcache_bucket: #test-and-set 0 #acquire() 6676
lock: bcache_bucket: #test-and-set 0 #acquire() 9368
lock: bcache_bucket: #test-and-set 0 #acquire() 5084
lock: bcache_bucket: #test-and-set 0 #acquire() 5087
lock: bcache_bucket: #test-and-set 0 #acquire() 3337
lock: bcache_bucket: #test-and-set 0 #acquire() 4889
lock: bcache_bucket: #test-and-set 0 #acquire() 2881
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 11910344 #acquire() 1254
lock: proc: #test-and-set 3804525 #acquire() 603302
lock: proc: #test-and-set 3378714 #acquire() 603301
lock: proc: #test-and-set 3269024 #acquire() 603303
lock: proc: #test-and-set 3173844 #acquire() 603303
tot= 0
test0: OK
start test1
```

【分析讨论】

通过本次实验，我深入了解了操作系统内存管理的关键机制，特别是在多核系统中的锁争用问题。通过给每个 CPU 分配专属的锁和空闲链表，显著减少了锁争用，提高了系统的性能。这次实验不仅强化了我对操作系统原理的理解，还提升了我在并发编程中的实战能力。另外，我理解了操作系统中缓冲区管理的重要性，以及如何通过优化锁机制来减少并发访问时的锁竞争。通过引入基于哈希表的缓冲区管理机制，并为每个桶分配独立的锁，大大提高了系统在多进程环境下的性能。这次实验不仅让我掌握了更高级的并发编程技巧，还增强了我在优化系统性能方面的能力。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：


== Test   kalloctest: test2 ==
kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (19.0s)
== Test running bcachetest ==
$ make qemu-gdb
(49.7s)
== Test   bcachetest: test0 ==
bcachetest: test0: OK
== Test   bcachetest: test1 ==
bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (600.2s)
...
OK
test opentest: OK
test writetest: OK
test writebig: panic: balloc: out of blocks
qemu-system-riscv64: terminating on signal 15 from pid 139899 (mak
MISSING '^ALL TESTS PASSED$'
QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 51/70
make: *** [Makefile:336: grade] Error 1
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab8$