Lock 实验报告

专　业：软　件　工　程

指　导　老　师：王　冬　青

年　级：大　学　二　年　级

学　号：2153061

姓　名：谢　嘉　麒

1. Memory allocator (moderate)
    1.1.实验目的
    1.2.实验步骤
        1.2.1.分析实验要点
        1.2.2.构造kmem数组
        1.2.3.修改kinit()
        1.2.4.修改kfree()
        1.2.5.修改kalloc()
        1.2.6.编译与检测
    1.3.实验中遇到的问题和解决方法
        1.3.1.设置每个锁的名称
    1.4.实验心得
2.Buffer cache (hard)
    2.1.实验目的
    2.2.实验步骤
        2.2.1.分析实验要点
        2.2.2.修改结构体
        2.2.3.修改binit()
        2.2.4.修改brelse()
        2.2.5.修改bpin()和bunpin()
        2.2.6.修改bget()函数
        2.2.7.编译与检测
    2.3.实验中遇到的问题和解决方法
        2.3.1.bget()函数中各个模块的加锁
    2.4.实验心得

1. Memory allocator (moderate)

1.1.实验目的

本次实验提供程序user/kalloctest对xv6的内存分配器进行压力测试：三个进程增长和缩小他们的地址空间，导致对kalloc和kfree的多次调用。kalloc和kfree获取kmem.lock，kalloctest打印出由于尝试获取另一个核心已经持有的锁而导致acquire函数中循环迭代的次数，从而对锁竞争有了粗略衡量。

实验要求使用一个专用的未加载的多核及其，任务是实现每个CPU的自由列表，并在每一个CPU的空闲列表为空时进行窃取。需要给每个锁以kmem为开头命名，即调用initlock，并传递一个以kmem开头的名称。运行kalloctest以查看实现是否减少了锁征用。

1.2.实验步骤

1.2.1.分析实验要点

在xv6中直接运行kalloctest，可以看到acquire被调用了433070次，循环迭代尝试获取锁7419次，kmem也是争用最多的五个锁之一。

因此实验要重新设计内存分配器，避免使用单个锁和单个空闲内存页链表，减少争用；

设计每个CPU都有自己的空闲内存页链表和对应的锁；

当前CPU的空闲内存页链表为空时能偷取其他CPU的空闲内存页链表；

cpuid()函数返回当前的CPU核序号，但前后需要关闭中断。

1.2.2.构造kmem数组

每个CPU需要有一个空闲内存页链表和相应的锁，因此在kernel/kalloc.c中定义kmem结构体替换为kmem数组，规定数组大小为CPU核心数NCPU，并且在结构体中添加lockname字段记载每个锁的名称。

1.2.3.修改kinit()

将kinit()函数中对kmem锁的初始化修改为对kmem数组的锁的循环初始化。

使用snprintf()函数设置每个锁的名称，并将其存储在lockname字段，保证每个锁的名称以kmem开头。

具体修改部分如下：

```
for (i = 0; i < NCPU; ++i) {
   snprintf(kmem[i].lockname, 8, "kmem_%d", i);
   initlock(&kmem[i].lock, kmems[i].lockname);
 }
```

1.2.4.修改kfree()

根据实验材料的具体代码可以推断，最初freearrange()将空闲内存分配给当前运行CPU的freelist，而其内部即调用kfree()进行内存回收。

每次都需要调用kfree()释放的物理页由当前运行的CPU的freelist回收，因此可以使用cpuid()函数获取当前CPU核心的序号，使用kmem中对应的锁和freelist进行回收。具体方法如下：

（1）定义strcut run类型指针r，用于将释放的页添加到空闲列表；

（2）定义变量c，用于保存CPU编号；

（3）首先检查函数物理地址是否页对齐，若否则触发panic，即这表示可能释放了无效的页；

（4）利用memset检查函数地址是否在合法的内核地址范围内；

（5）在释放页面前将页面内容填充为1，确保释放的页重新分配前，若任何代码尝试访问该页，将得到非法或不一致的数据；

（6）将传递给kfree函数的地址转换为struct run的指针，用acquire获取kmem对应的锁，避免多个CPU同时修改该CPU空闲链表；

（7）将释放的页r添加到当前CPU的空闲列表开头，表示该页可用于下一次分配；

（8）释放kmem对应的锁，即当前CPU空闲列表更新完成。

1.2.5.修改kalloc()

完成以上两个函数后，还需要修改最主要的kalloc()函数，具体方法如下：

（1）定义strcut run类型的指针r，并且用push_off和pop_off禁用中断，获取当前CPU编号；

（2）获取kmem中的锁;

（3）从当前CPU空闲列表kmem[c].freelist获取一个空闲的页，如果有可用的页，则从当前CPU空闲列表中取一个；

（4）若当前CPU空闲列表为空，即没有可用页，则依次遍历其他CPU，找到第一个有可用页的CPU，将页从该CPU空闲列表中取出；

（5）获取到一个空闲页后，释放相应的锁。

（6）获取空闲页r后，使用memset将该页内容填充为5，以便之后使用中检测是否有未初始化的数据；

（7）最后返回分配的页的地址，或若没有可用的空闲页即返回空指针。

1.2.6.编译与检测

完成以上步骤后进行编译，执行kalloctest命令进行检测，得到预期结果：

```
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 142276
lock: kmem: #test-and-set 0 #acquire() 129785
lock: kmem: #test-and-set 0 #acquire() 161009
lock: bcache: #test-and-set 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #test-and-set 2358041 #acquire() 541319
lock: proc: #test-and-set 2326359 #acquire() 541890
lock: proc: #test-and-set 2235875 #acquire() 541891
lock: proc: #test-and-set 2132210 #acquire() 541832
lock: proc: #test-and-set 2064677 #acquire() 541891
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

1.3.实验中遇到的问题和解决方法

1.3.1.设置每个锁的名称

最初对于如何设置每个锁的名称存在疑问，查找资料后了解到锁名称的记录是指针的浅拷贝lk->name=name，因此对于每个锁的名称需要使用全局的内存进行记录，而不是函数的局部变量，才能防止内存的丢失。

因此使用sprintf()函数来设置每个锁的名称。

1.4.实验心得

通过本次实验，一方面警示笔者锁的争用问题，另一方面笔者也在具体的代码实现中收获颇丰。通过具体的资料和代码，笔者了解到锁的命名方法，锁之间争用的解决方法和这种新的解决空闲页调用的方法。

2.Buffer cache (hard)

2.1.实验目的

本实验与前半部分无关，可以独立进行并通过测试。如果多个进程对文件系统进行大量使用，则很可能争夺bcache.lock，该锁保护了内核中的磁盘块缓存。

实验要求修改块缓存，使在运行bcachetest时，所有与块缓存相关的锁acquire循环迭代次数接近零。理想情况下所有涉及到块缓存的锁的循环迭代次数之和为零，但这个和小于500也可以接受。同时要求修改bget和brelse，使对于位于块缓存中的不同块的并发查找和释放不太可能发生在锁上的冲突。必须保持至多有一个副本的块被缓存的不变性。

2.2.实验步骤

2.2.1.分析实验要点

根据bcachetest测试结果可以看出，acquire()调用65930次，循环尝试获取锁38272次，可以看出bcache锁争用情况很明显。

修改bget()和brelse()来尽可能减少锁的争用。

使用线程安全的哈希表来寻找cache中的块号。

移除缓冲区的链表结构，使用最近使用时间的时间戳的缓冲区来代替。

2.2.2.修改结构体

（1）需要修改kernel/buf.h中的buf结构体。

不再使用双向链表而是使用哈希表，对于bucket使用单向链表，不再需要prev字段。

同时由于算法基于时间戳比较，因此添加timestamp用于记录最后使用缓冲块的时间。

（2）需要修改kernel/bio.c中的bcache结构体。

添加size字段用于记录已经分配到哈希表的缓存块struct buf的数量。

添加bucktes[NBUCKET]数组作为哈希表的bucket数组，其中NBUCKET为bucket数目。

添加locks[NBUCKET]作为每个bucket对应的锁。

添加hashlock作为哈希表的全局锁。

2.2.3.修改binit()

这一函数不再使用双向链表，而是新增size，bucket数组的锁locks以及哈希表全局锁hashlock进行初始化。

2.2.4.修改brelse()

这一函数修改为基于时间戳的LRU视线，不再使用双向链表，只需要将当前时间戳记录的缓存快的timestamp。加锁也由原本的全局锁改为缓存块所在的bucket的锁。

2.2.5.修改bpin()和bunpin()

将原本的全局锁替换为缓存块对应的bucket的锁即可。

2.2.6.修改bget()函数

修改bget()函数使得哈希表中寻找对应的缓存块，未找到则进行分配或者根据LRU和引用计数选择缓存块进行覆盖重用。具体修改方式如下：

（1）根据blockno在哈希表相应的bucket链表中寻找对应缓存块，找到则直接返回；、

（2）未找到则进行缓存块分配。根据bcache.size获取当前已分配的缓存块，若还有缓存块未分配，则进行分配。将缓存块初始化后，再插入到对应的哈希表bucket中；

（3）全部缓存块分配后，根据时间戳进行重用。先从目标的bucket，找到引用计数为0且时间戳最小的缓存块进行重用。未找到则依次向后续的bucket链表中查找，此时由于bucket链表和目标不一致，因此需要将当前缓存块移到目标的bucket；

2.2.7.编译与检测

完成以上步骤后，编译通过，执行bcachetest，得到预期结果：

```
tot= 0
test0: OK
start test1
test1 OK
$
```

2.3.实验中遇到的问题和解决方法

2.3.1.bget()函数中各个模块的加锁

整个函数调用的字段较多，因此加锁也很复杂。

（1）首先需要对遍历哈希表时的bucket加锁；

（2）对size字段进行读取和更新时加锁；

（3）寻找可用缓存块时加锁，即哈希表全局锁。

2.4.实验心得

通过本次实验，笔者体会了减少加锁开销的方法，可以与实验1总结为两种：

（1）资源重复：在kalloc中设置多份资源减少进程的等待；

（2）细化加锁：在bcache中将加锁复杂化精密化，减少资源加锁的冲突。