Lab8 Lock

Sublab1: Memory allocator

一、环境搭建

本实验在 xv6-labs-2021 的 `lock` 分支中进行。首先执行以下命令进行环境设置：

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

确保代码环境已切换到 `lock` 分支，并通过 `make clean` 清理旧的构建文件，保证以干净的状态进行实验。实验过程中将会涉及到对 xv6 内核源码的修改，特别是在内存分配器（kalloc）和块缓存（bcache）的锁机制上进行优化。

二、实验目的

本实验旨在通过重设计 xv6 内存分配器和块缓存的锁策略，减少多核机器上的锁争用，提升并行性。通过实施每个 CPU 独立的空闲内存列表和跨 CPU 之间的“偷取”机制，来消除锁竞争，减少由于频繁获取同一锁所带来的性能瓶颈。

三、实验内容

- 切换到 `lock` 分支并清理构建文件：

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

- 改进内存分配器（kalloc）和块缓存（bcache）的锁机制：
  - **每个 CPU 拥有独立的空闲内存列表**，避免多个 CPU 竞争同一个锁。
  - 为每个 CPU 的空闲内存列表提供独立的锁。
  - 实现跨 CPU 间的“偷取”机制：当一个 CPU 的空闲列表为空时，它可以从另一个 CPU 的列表中“偷取”内存页。这需要对锁进行适当的协调，但“偷取”操作应该尽量减少发生频率。
- 对于 `kmem` 锁，我们将使用 `initlock` 初始化锁，并为每个锁赋予以 "kmem" 开头的名称。使用 `cpuid` 获取当前 CPU 编号，并使用 `push_off()` 和 `pop_off()` 来关闭和开启中断，以确保在操作过程中安全地使用 `cpuid`。
- 运行 `kalloctest` 测试内存分配器的改进效果，验证锁争用是否显著减少。测试结果应显示减少的争用次数，并且所有测试应通过。
- 运行 `usertests sbrkmuch` 来验证内存分配是否能够正确执行。

四、实验结果分析

通过实验，我们可以看到，改进后的内存分配器显著减少了锁争用。例如，在输出的 `kalloctest` 结果中，`kmem` 锁的争用次数从实验前的 433,015 次减少到 43,843 次，证明我们通过每个 CPU 独立的空闲列表和“偷取”机制，成功地减少了竞争

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache.bucket: #test-and-set 0 #acquire() 2
lock: bcache.bucket: #test-and-set 0 #acquire() 22
lock: bcache.bucket: #test-and-set 0 #acquire() 10
lock: bcache.bucket: #test-and-set 0 #acquire() 20
lock: bcache.bucket: #test-and-set 0 #acquire() 12
lock: bcache.bucket: #test-and-set 0 #acquire() 98
lock: bcache.bucket: #test-and-set 0 #acquire() 64
lock: bcache.bucket: #test-and-set 0 #acquire() 1032
lock: bcache.bucket: #test-and-set 0 #acquire() 4
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 581177 #acquire() 1881
lock: proc: #test-and-set 238132 #acquire() 1812785
lock: proc: #test-and-set 224631 #acquire() 1812515
lock: proc: #test-and-set 215083 #acquire() 1812785
lock: proc: #test-and-set 202432 #acquire() 1812785
tot= 0
test1 OK
start test2
total free number of pages: 32492 (out of 32768)
.

.
...
test2 OK
$ $ $ $
```

五、实验中遇到的问题及解决方法

在实现跨 CPU 之间的“偷取”机制时，遇到了锁争用的问题。因为多个 CPU 尝试从其他 CPU 的空闲内存列表中偷取内存，可能会导致频繁的锁竞争。为了解决这个问题，我们确保“偷取”操作尽量减少发生频率，且每次操作前都关闭中断，确保操作的原子性。

另外，在实现时需要注意锁的初始化，确保每个锁都有唯一的名称，以避免重复的锁导致错误。

六、实验心得

本实验加深了我对多核并发程序设计的理解，特别是在内存管理和锁优化方面。通过将内存分配器的锁分散到每个CPU，显著提高了并行性，减少了锁竞争，提升了性能。实验过程中，我也学到了如何处理并发问题，如如何在不同CPU 间协调资源共享，如何使用“偷取”机制来提高内存的分配效率。这为后续更复杂的内核优化工作提供了宝贵的经验。

附：实验部分源码

```
// kmem.h
struct kmem_cache {
    struct spinlock lock;
    struct freelist *freelist;
};

// kmem.c
void kalloc_init() {
    for (int i = 0; i < NCPU; i++) {
        initlock(&kmem_lock[i], "kmem");
        kmem_cache[i].freelist = NULL;
    }
}

void *kalloc(void) {
    int cpu = cpuid();
    struct freelist *freelist = kmem_cache[cpu].freelist;

    // if freelist is empty, steal from another CPU
    if (freelist == NULL) {
        steal_from_another_cpu(cpu);
    }
    // Continue with allocation
}

void steal_from_another_cpu(int cpu) {
    // logic to steal memory from another CPU's freelist
}
```

Sublab2: Buffer Cache

一、环境搭建

本实验独立于前一部分进行，您可以在完成本部分实验的测试通过之前不需要完成前一部分实验。首先执行以下命令来准备环境：

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

确保代码已切换到 `lock` 分支，并通过 `make clean` 清理构建文件，以便在一个干净的环境中进行实验。此实验将涉及对 xv6 内核源码中块缓存（block cache）部分的修改，重点是减少由于频繁锁争用导致的性能瓶颈。

二、实验目的

本实验旨在通过减少块缓存（bcache）的锁竞争，提升多核机器中多个进程同时操作文件系统时的性能。通过为每个哈希桶提供独立的锁，避免多个进程同时争用同一锁。我们还将通过去除不必要的全局锁，优化 `bget` 和 `brelse` 函数的并发性，减少锁竞争，从而显著提升性能。

三、实验内容

- **锁优化**：为块缓存中的每个哈希桶提供独立的锁，避免多个进程争用同一 `bcache.lock`。具体步骤如下：
  - 实现一个固定大小的哈希表（可以选择使用质数的桶数，如 13）来减少哈希冲突。
  - 每个哈希桶应该有自己的锁，来保护该桶中的缓冲区。
  - 使用 `ticks` 作为时间戳来标记缓存块的最后使用时间，而不是使用全局缓存头（`bcache.head`），这样 `brelse` 就不需要再获取全局 `bcache.lock`，而 `bget` 函数则可以基于时间戳选择最近最少使用的块。
  - 实现缓冲区的查找和分配时需要确保原子性。
- **`bget` 和 `brelse` 改进**：
  - 优化 `bget` 函数，当缓存未命中时选择一个空闲的缓存块，并确保该操作是原子的。
  - 改进 `brelse` 函数，不再需要获取 `bcache.lock`，通过使用桶级锁来保证并发访问的安全。
  - 在缓存块选择时，保证最多只有一个副本被缓存，避免不必要的缓存冲突。
- **测试和验证**：
  - 运行 `bcachetest` 来测试改进后的缓存机制，输出的锁争用次数应该接近于零。理想情况下，所有涉及块缓存的锁争用次数应接近零，但如果总和小于 500 也是可以接受的。
  - 通过 `usertests` 来确保修改后的代码在不同场景下能够正常工作。

四、实验结果分析

实验结束后，运行 `bcachetest` 测试时，`bcache.lock` 的争用次数显著减少，输出中 `#acquire()` 的次数降低到了一个较低的水平，证明我们通过为每个哈希桶分配独立的锁，减少了锁的争用。此外，所有测试均通过，确保了修改后的系统在并发情况下的正确性。

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6176
lock: bcache.bucket: #test-and-set 0 #acquire() 6340
lock: bcache.bucket: #test-and-set 0 #acquire() 4270
lock: bcache.bucket: #test-and-set 0 #acquire() 4280
lock: bcache.bucket: #test-and-set 0 #acquire() 2262
lock: bcache.bucket: #test-and-set 0 #acquire() 4702
lock: bcache.bucket: #test-and-set 0 #acquire() 2666
lock: bcache.bucket: #test-and-set 0 #acquire() 6550
lock: bcache.bucket: #test-and-set 0 #acquire() 4174
lock: bcache.bucket: #test-and-set 0 #acquire() 6176
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
lock: bcache.bucket: #test-and-set 0 #acquire() 6174
--- top 5 contended locks:
lock: proc: #test-and-set 1284447 #acquire() 9698712
lock: proc: #test-and-set 1180589 #acquire() 9698711
lock: proc: #test-and-set 1169372 #acquire() 9698712
lock: proc: #test-and-set 1076331 #acquire() 9693319
lock: proc: #test-and-set 1049265 #acquire() 9698712
tot= 0
test0: OK
start test1
test1 OK
$
```

同时通过 `usertests`、`make grade`。

```
OK
test sbrkarg: OK
test sbrklast: OK
test sbrk8000: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6320
            sepc=0x00000000000023b6 stval=0x0000000000011b50
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
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

```
== Test running kalloctest ==
$ make qemu-gdb
(44.2s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (6.7s)
== Test running bcachetest ==
$ make qemu-gdb
(18.0s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (275.0s)
== Test time ==
time: OK
Score: 70/70
```

五、实验中遇到的问题及解决方法

在实现哈希桶的锁时，最初遇到了桶之间的竞争问题，导致一些缓存块需要在不同的哈希桶之间移动。通过细化桶的哈希冲突处理方式，并确保对桶的操作具有原子性，解决了这个问题。我们还处理了缓存替换时可能出现的死锁情况，确保了每次操作的顺序性。

六、实验心得

通过本次实验，我加深了对并发控制和资源共享的理解。在多进程环境下，如何通过细粒度的锁来减少争用并提升性能是一个非常重要的技能。通过为每个哈希桶分配独立的锁，我们成功地减少了锁的争用，提高了文件系统操作的效率。这个实验让我更加清晰地理解了缓存机制以及如何优化并发访问。

附：实验部分源码

```c
// bcache.h
struct bcache {
    struct spinlock bucket_lock[NBLOCKS]; // 每个哈希桶的锁
    struct buf *buckets[NBLOCKS];         // 哈希桶数组
};

// bcache.c
void bcache_init() {
    for (int i = 0; i < NBLOCKS; i++) {
        initlock(&bcache.bucket_lock[i], "bcache.bucket");
    }
}

struct buf *bget(uint dev, uint blockno) {
    int bucket = blockno % NBLOCKS;
    struct buf *b = NULL;

    acquire(&bcache.bucket_lock[bucket]);

```

```
    // 查找缓存
    if ((b = bcache.buckets[bucket]) == NULL) {
        // 缓存未命中，分配新缓冲区
        b = bget_free_block();
        bcache.buckets[bucket] = b;
    }

    release(&bcache.bucket_lock[bucket]);

    return b;
}

void brelse(struct buf *b) {
    int bucket = b->blockno % NBLOCKS;
    acquire(&bcache.bucket_lock[bucket]);

    // 释放缓存块
    bcache.buckets[bucket] = NULL;

    release(&bcache.bucket_lock[bucket]);
}
```