Lab 8 Locks

目录

[1. Memory allocator (moderate) 1](#_Toc174491483)

[1.1 实验目的 1](#_Toc174491484)

[1.2 实验步骤 1](#_Toc174491485)

[1.3 实验结果 4](#_Toc174491486)

[1.4 遇到的问题 6](#_Toc174491487)

[1.5 实验心得 6](#_Toc174491488)

[2. Buffer cache (hard) 7](#_Toc174491489)

[2.1 实验目的 7](#_Toc174491490)

[2.2 实验步骤 7](#_Toc174491491)

[2.3 实验结果 13](#_Toc174491492)

[2.4 实验中遇到的问题和解决方法 13](#_Toc174491493)

[2.5 实验心得 14](#_Toc174491494)

[3. 测试结果 15](#_Toc174491495)

Memory allocator (moderate)

* 1. 实验目的
* 重构内存分配器的代码,实现每个CPU都有自己的空闲内存列表,并各自用锁保护,以消除单个全局锁和空闲列表带来的争用
* 当一个CPU的空闲列表为空时,实现从其他CPU的列表中“偷取”部分空闲内存的机制。这可能会引入一些锁争用,但希望很少发生
* 熟悉并运用xv6中与并发和锁相关的接口,如锁、关闭中断等
  1. 实验步骤
* 为每个CPU核心维护一个kmem,包含自由列表和锁， NCPU 是指CPU核心的数量

struct {

struct spinlock lock;

struct run \*freelist;

} kmem[NCPU]; // 每个CPU都有维护独有的kmem

* 由于kmem现在是一个数组，需要修改 kinit() 函数，以初始化所有CPU上的列表：

void kinit()

{

// 每个CPU列表初始化

char kmem\_name[32];

for (int i = 0; i < NCPU; i++) {

snprintf(kmem\_name, 32, "kmem\_%d", i);

initlock(&kmem[i].lock, kmem\_name);

}

freerange(end, (void \*)PHYSTOP);

}

* 修改 kalloc 函数，实现从当前CPU的自由列表中分配一个空闲页，如果当前CPU没有空闲页，就会从其他CPU的自由列表中窃取一个空闲页

// Allocate one 4096-byte page of physical memory.

// Returns a pointer that the kernel can use.

// Returns 0 if the memory cannot be allocated.

void \*kalloc(void)

{

struct run \*r;

// 关闭中断，防止中断过程中修改共享数据

push\_off();

// 获取当前CPU的ID

int CPUID = cpuid();

// 获取当前CPU的kmem锁，保护共享数据

acquire(&kmem[CPUID].lock);

// 在当前CPU的自由列表中查找空闲页

r = kmem[CPUID].freelist;

// 如果在当前CPU的自由列表中找到空闲页

if (r)

kmem[CPUID].freelist = r->next;

// 如果当前CPU的自由列表中没有空闲页

if (r == 0) {

// 在其他CPU的自由列表中查找空闲页

for (int i = 0; i < NCPU; i++) {

if (i == CPUID)

continue; // 跳过当前CPU

// 获取其他CPU的kmem锁

acquire(&kmem[i].lock);

// 在其他CPU的自由列表中查找空闲页

r = kmem[i].freelist;

if (r)

kmem[i].freelist = r->next;

// 释放其他CPU的kmem锁

release(&kmem[i].lock);

// 如果找到空闲页，跳出循环

if (r)

break;

}

}

// 释放当前CPU的kmem锁

release(&kmem[CPUID].lock);

// 恢复中断

pop\_off();

// 如果找到了空闲页，用垃圾数据填充该页

if (r)

memset((char \*)r, 5, PGSIZE);

// 返回找到的空闲页的地址

return (void \*)r;

}

* 修改kfree函数，首先检查传入的地址是否有效，然后用垃圾数据填充该页。接着关闭中断，获取当前CPU的ID和kmem锁，将空闲页插入当前CPU的自由列表头部，释放锁并恢复中断

// Free the page of physical memory pointed at by pa,

// which normally should have been returned by a

// call to kalloc(). (The exception is when

// initializing the allocator; see kinit above.)

void kfree(void \*pa)

{

struct run \*r;

// 检查给定的地址是否有效

if (((uint64)pa % PGSIZE) != 0 || (char \*)pa < end || (uint64)pa >= PHYSTOP)

panic("kfree");

// 用垃圾数据填充页，以捕获悬空引用

memset(pa, 1, PGSIZE);

r = (struct run \*)pa;

// 关闭中断，防止中断过程中修改共享数据

push\_off();

// 获取当前CPU的ID

int CPUID = cpuid();

// 获取当前CPU的kmem锁，保护共享数据

acquire(&kmem[CPUID].lock);

// 从链表头插入空闲页

r->next = kmem[CPUID].freelist;

kmem[CPUID].freelist = r;

// 释放当前CPU的kmem锁

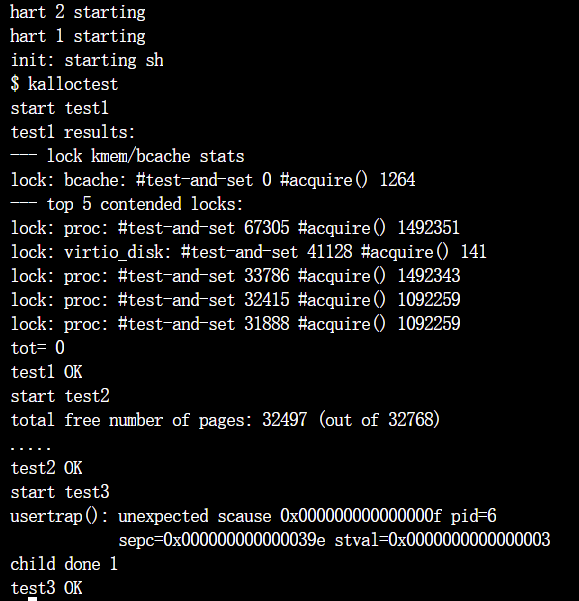
release(&kmem[CPUID].lock);

// 恢复中断

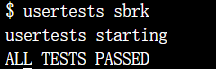
pop\_off();

}

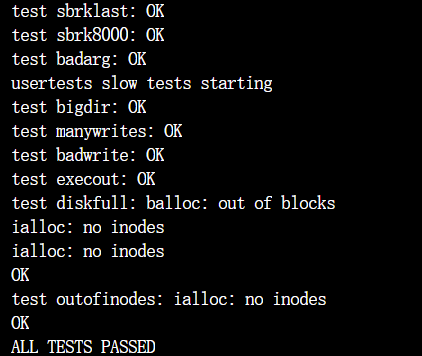
* 1. 实验结果
* 运行 make qemu 后执行 kalloctest ：



执行usertests sbrk ：



执行usertests ：



* 1. 遇到的问题
* 问题：

在为每个CPU分配独立的freelist和锁时，会出现线程竞争和死锁的问题。这是因为多个线程可能会同时尝试获取锁，导致死锁情况。同时，在“偷取”内存页的逻辑中，不同CPU之间可能会相互竞争。在这学期的操作系统理论课程上，大多数讲的都是单CPU，很少提到多核处理器，所以我对多个CPU并发编程没有一个很具体的概念。

* 解决方法：

使用正确的同步机制，确保在关键代码段使用适当的同步机制，比如说自旋锁。避免死锁，确保在获取多个锁时按照相同的顺序获取，以避免死锁。锁是一种用于保护共享数据的工具，它可以保证在任何时刻只有一个线程可以访问被保护的数据。如果一个线程持有一个锁，那么其他尝试获取该锁的线程将会被阻塞，直到锁的持有者释放了锁。

锁的使用需要小心，过度的锁定会导致性能下降，，不正确的使用锁则可能导致死锁或者数据不一致性。另外，可以采用超时机制或者尝试次数限制来避免永久性的锁竞争。还可以仔细设计“偷取”逻辑：在“偷取”内存页时，确保避免两个CPU互相“偷取”的情况。可以使用类似于试探性获取锁的方法来判断是否应该继续“偷取”。

* 1. 实验心得

通过为每个CPU核心维护独立的空闲内存列表，并为每个列表设置锁，可以有效减少全局锁和单一空闲列表带来的争用问题。这种设计提升了并发处理的效率，尤其是在多核处理器环境中，使内存分配操作更加高效。

实验中通过锁机制来保护共享数据，确保在多线程环境下的数据一致性。特别是在kalloc和kfree函数中，锁的使用避免了多个CPU同时访问相同内存区域时可能出现的竞争条件。

Buffer cache (hard)

* 1. 实验目的
* 修改块缓存的代码，使得在运行 bcachetest 时，所有涉及到的锁的获取循环迭代次数接近零。尽可能减少不同锁之间的竞争，以提高性能
* 通过对 bget 和 brelse 函数的修改，使得在块缓存中对不同块的并发查找和释放操作不太可能在锁上发生冲突。这不仅能减少锁的竞争，还要保持每个块只有一个副本被缓存的不变性
  1. 实验步骤
* 改进方案说明：当文件系统被多个进程频繁访问时，它们可能会争夺 bcache.lock ，导致性能下降。为了优化这一问题，我们需要减少锁冲突的概率。直接将锁粒度缩小到每个块是不切实际的，因为这样会增加管理复杂度和锁开销。于是使用哈希表维护一组固定数量的哈希桶，每个桶有一个独立的锁。选择一个素数（如 13）作为哈希桶的数量，降低哈希冲突的概率。每个哈希桶的锁只负责保护该桶中的缓存块，限制锁争用范围，减少冲突
* 修改 kernel/buf.h 中的 buf 结构体，增加 LUtime （最新使用时间）和 curBucket （当前所属哈希桶）字段，方便查找和管理：

struct buf {

int valid; // has data been read from disk?

int disk; // does disk "own" buf?

uint dev;

uint blockno;

struct sleeplock lock;

uint refcnt;

struct buf \*prev; // LRU cache list

struct buf \*next;

uchar data[BSIZE];

uint LUtime; // 最近使用时间

int curBucket; // 当前所属哈希桶

};

* 修改 bcache 结构体，增加 NBUCKET 个桶，每个桶指向一个链表，每个桶有一个属于自己的锁存储在 bucket\_locks 数组中：

struct {

struct spinlock lock;

struct buf buf[NBUF];

// 维护NBUCKET个桶，每个桶维护一个链表

// 每个桶都有一个自己的锁，用于保护自己的链表

// 而每个桶中存储的元素buf作为缓冲区存在自己的锁

struct buf bucket[NBUCKET];

struct spinlock bucket\_locks[NBUCKET];

} bcache;

* 现在需要维护 NBUCKET 个哈希桶，因此需要跟着修改 binit() ，完成对桶的初始化

void binit(void)

{

// 初始化全局缓存锁

initlock(&bcache.lock, "bcache\_lock");

// 初始化每个bucket的锁

char name[32];

for (int i = 0; i < NBUCKET; i++) {

snprintf(name, 32, "bucket\_lock\_%d", i); // 生成锁的名称

initlock(&bcache.bucket\_locks[i], name); // 初始化bucket锁

bcache.bucket[i].next = 0; // 初始化bucket链表头指针为0

}

// 初始化每个buffer

for (int i = 0; i < NBUF; i++) {

struct buf \*b = &bcache.buf[i]; // 获取第i个buffer

initsleeplock(&b->lock, "buffer"); // 初始化buffer的sleeplock锁

b->LUtime = 0; // 初始化最近使用时间

b->refcnt = 0; // 初始化引用计数

b->curBucket = 0; // 初始化当前所属bucket为0

// 将buffer加入到bucket[0]的链表中

b->next = bcache.bucket[0].next; // 将buffer的next指向当前bucket[0]的链表头

bcache.bucket[0].next = b; // 更新bucket[0]的链表头指针，使其指向新的buffer

}

}

* 修改 bget 函数，实现了通过哈希表在缓存中查找指定的块，如果找不到则根据LRU算法在其他哈希桶中寻找或替换缓存块并更新其信息

// Look through buffer cache for block on device dev.

// If not found, allocate a buffer.

// In either case, return locked buffer.

static struct buf \*bget(uint dev, uint blockno)

{

uint index = hash(dev, blockno);

acquire(&bcache.bucket\_locks[index]);

struct buf \*b = bcache.bucket[index].next;

// 在当前bucket[index]中查找

while (b) {

if (b->dev == dev && b->blockno == blockno) {

// 已找到

b->refcnt++;

release(&bcache.bucket\_locks[index]);

acquiresleep(&b->lock);

return b;

}

b = b->next;

}

// 未找到，需要从其他bucket中查找

// 占有bucket\_lock时, 再获取其他bucket锁是不安全的, 同时占有多个bucket锁

// 容易导致循环等待造成死锁，故先释放当前bucket锁

release(&bcache.bucket\_locks[index]);

// 因其他进程可能使用使用该块，因此要检查一遍该block是否已经在其他bucket中

// 如果已经在其他bucket中，则等待获取sleeplock再返回即可

acquire(&bcache.lock);

b = bcache.bucket[index].next;

while (b) {

if (b->dev == dev && b->blockno == blockno) {

// 已被其他进程放入其他bucket中

acquire(&bcache.bucket\_locks[index]);

b->refcnt++;

release(&bcache.bucket\_locks[index]);

release(&bcache.lock); // 释放bcache锁

// 重新获取该block的锁，睡眠等待即可，等待完毕便可返回

acquiresleep(&b->lock);

return b;

}

b = b->next;

}

// 若还未找到，需要依据LRU从其他bucket中查找

// 在当前bucket[index]中查找空闲缓冲区或者最近最少使用的缓冲区

struct buf \*LRUb = 0;

uint curBucket = -1;

uint LUtime = INTMAX;

for (int i = 0; i < NBUCKET; i++) {

acquire(&bcache.bucket\_locks[i]);

b = &bcache.bucket[i];

int found = 0;

while (b->next) {

if (b->next->refcnt == 0 && LRUb == 0) {

// 如果找到空闲缓冲区且之前还没有找到过空闲缓冲区

LRUb = b;

LUtime = b->next->LUtime;

found = 1;

}

else if (b->next->refcnt == 0 && b->next->LUtime < LUtime) {

// 如果找到空闲缓冲区，且该缓冲区上次使用时间更早

LRUb = b;

LUtime = b->next->LUtime;

found = 1;

}

b = b->next;

}

if (found) {

// 更新了LRUb，要释放这个桶之前的bucket锁

if (curBucket != -1) {

// 释放之前的bucket锁

release(&bcache.bucket\_locks[curBucket]);

}

curBucket = i;

}

else {

// 没找到，释放访问的桶

release(&bcache.bucket\_locks[i]);

}

}

if (LRUb == 0) {

panic("bget: No buffer.");

}

else {

struct buf \*p = LRUb->next;

if (curBucket != index) {

// 删除LRUb节点

LRUb->next = p->next;

release(&bcache.bucket\_locks[curBucket]);

// 将LRUb节点放入当前bucket[index]中

acquire(&bcache.bucket\_locks[index]);

p->next = bcache.bucket[index].next;

bcache.bucket[index].next = p;

}

// 更新LRUb的信息

p->dev = dev;

p->blockno = blockno;

p->refcnt = 1;

p->valid = 0;

p->curBucket = index;

release(&bcache.bucket\_locks[index]); // 释放bucket[index]锁

release(&bcache.lock); // 释放bcache锁

acquiresleep(&p->lock); // 获取LRUb的锁

return p;

}

}

* 修改 brelse( ) 函数，当一个块引用数为0时将把当前时间记录为最近使用时间：

// Release a locked buffer.

// Move to the head of the most-recently-used list.

void brelse(struct buf \*b)

{

if (!holdingsleep(&b->lock))

panic("brelse");

releasesleep(&b->lock);

uint index = hash(b->dev, b->blockno);

acquire(&bcache.bucket\_locks[index]);

b->refcnt--;

if (b->refcnt == 0) {

//没有进程引用这块buffer, 则为空闲状态释放，记录最近使用时间

b->LUtime = ticks;

}

release(&bcache.bucket\_locks[index]);

}

* 修改 bpin( ) 和 bunpin( ) 函数,使其适应引入哈希桶之后的情况

void bpin(struct buf \*b)

{

uint index = hash(b->dev, b->blockno);

acquire(&bcache.bucket\_locks[index]);

b->refcnt++;

release(&bcache.bucket\_locks[index]);

}

void bunpin(struct buf \*b)

{

uint index = hash(b->dev, b->blockno);

acquire(&bcache.bucket\_locks[index]);

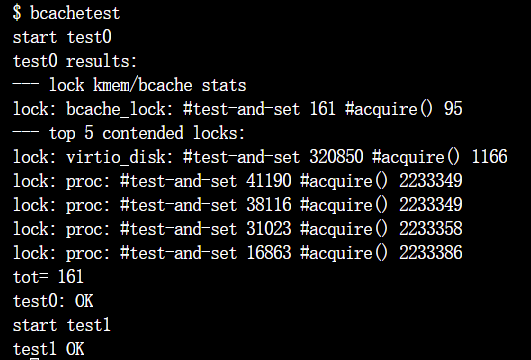
b->refcnt--;

release(&bcache.bucket\_locks[index]);

}

* 1. 实验结果

运行 make qemu 后执行 bcachetest ：



* 1. 实验中遇到的问题和解决方法
* 问题：

为什么要使用哈希桶，而且大小要规定为质数13

* 解决方法：

使用哈希桶的目的是为了细化粒度（相较于 bcache 过大， buf 过小），使用质数13是为了减少哈希冲突的概率。若使用到了相同的索引，则称为产生了哈希冲突，从而可能导致哈希聚集，使遍历效率下降，可考虑二次哈希来降低聚集。但如果使用的是非质数大小的桶，可能会导致不良后果，例如，假设哈希表的容量为15，某个key经过双哈希函数后得到的数组下标为0，步长为5。那么这个探测过程的序列是0,5,10,0,5,10，一直只会尝试这三个位置，永远找不到空白位置来存放，最终会导致崩溃。但在本题当中使用的是哈希链表，不以质数作为桶的大小尽管不会崩溃，但会增加聚集而导致浪费。

* 1. 实验心得

在实验中，我使用了 pthread\_mutex\_lock 和 pthread\_mutex\_unlock 对哈希表的桶进行加锁和解锁操作。这让我理解了互斥锁的作用，即通过确保只有一个线程能够进入临界区来解决竞态条件问题。互斥锁的使用大大提高了程序的安全性和可靠性，保证了并发通过将锁的粒度缩小到哈希桶，而不是每个缓存块，显著减少了不同进程之间的锁竞争。这个方法既保持了锁管理的简洁性，又在性能上取得了平衡，避免了直接为每个缓存块设置锁带来的复杂性和开销。

引入哈希表维护多个哈希桶，每个桶有独立的锁，这种设计有效降低了哈希冲突的概率，减少了锁的争用，提高了并发环境下的缓存查找效率。通过选择合适的哈希桶数量（如素数），进一步优化了缓存的性能。

通过在不同的哈希桶中应用LRU（最近最少使用）算法，实现了缓存块的动态替换。这不仅保证了缓存块的一致性，还优化了内存的使用效率。在处理缓存块时，合理应用LRU策略，确保高效的缓存管理。操作的正确性。

测试结果

