Lab 8 locks

Memory allocator (moderate)

实验目的

通过重新设计内存分配器，优化锁的争用情况，提高内存分配的性能和效率。

实验步骤

1. 为每个 `CPU` 核心创建独立的空闲内存页链表和锁。 当本地 `CPU` 的空闲链表为空时，尝试偷取其他 `CPU` 的空闲链表，需要关闭中断以保证正确性。
2. 使用 `kmems` 数组存储每个 `CPU` 的空闲内存页链表和锁，添加 `lockname` 字段记录锁的名称。
3. 修改 `kinit()` 函数： 初始化每个 `CPU` 的空闲链表和锁，使用循环设置每个锁的名称，以及调用 initlock 初始化锁。

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

4. 修改 `kfree()` 函数： 释放内存页时，根据当前 `CPU` 核心获取对应的锁，确保每个 `CPU` 的空闲链表正确维护。

```
void kfree(void *pa) {
    // ...
    acquire(&kmems[c].lock);
    r->next = kmems[c].freelist;
    kmems[c].freelist = r;
    release(&kmems[c].lock);
}
```

5. 修改 `kalloc()` 函数： 在分配内存页时，先尝试从本地 `CPU` 的空闲链表获取，如果为空则调用 `steal()` 函数偷取其他 CPU 的一半空闲内存页。

```
void *kalloc() {
    // ...
    acquire(&kmems[c].lock);
    r = kmems[c].freelist;
    if (r)
```

```
        kmems[c].freelist = r->next;
    release(&kmems[c].lock);

    if (!r && (r = steal(c))) {
        acquire(&kmems[c].lock);
        kmems[c].freelist = r->next;
        release(&kmems[c].lock);
    }
    // ...
}
```

6. 编写 `steal()` 函数： 该函数用于从其他 CPU 偷取部分空闲内存页，保证正确的锁使用和避免死锁。

```
struct run *steal(int cpu_id) {
    int i;
    int c = cpu_id;
    struct run *fast, *slow, *head;
    // ...
    for (i = 1; i < NCPU; ++i) {
        if (++c == NCPU) {
            c = 0;
        }
        acquire(&kmems[c].lock);
        if (kmems[c].freelist) {
            slow = head = kmems[c].freelist;
            fast = slow->next;
            // ...
            kmems[c].freelist = slow->next;
            release(&kmems[c].lock);
            slow->next = 0;
            return head;
        }
        release(&kmems[c].lock);
    }
    return 0;
}
```

实验中遇到的问题和解决办法

1. 问题：原始的内存分配器在多核系统上存在锁的争用问题，导致性能下降。

- 解决办法： 分离锁： 通过为每个 CPU 核心创建独立的空闲内存页链表和锁，避免了多核之间的锁争用。这样，每个 CPU 核心可以独立地进行内存分配和释放操作，提高了性能和效率。

实验心得

通过重新设计内存分配器，深入理解了锁的机制在多核系统中的重要性。通过优化锁的使用，不仅提高了内存分配的性能，还加深了我对操作系统中并发控制的理解。

Buffer cache (hard)

实验目的

通过优化缓冲区管理，减少锁的争用，提高系统文件 `I/O` 操作的性能和效率。

实验步骤

1. 使用线程安全的哈希表来管理缓存中的块号。 移除缓冲区的链表结构，使用基于时间戳的 `LRU` 算法来管理缓冲区的重用。
2. 修改数据结构： 修改 `buf` 和 `bcache` 结构体，为每个缓冲区添加时间戳字段，为 `bcache` 添加哈希表的锁。

```
struct buf {
    // ...
    uint timestamp; // 添加时间戳字段
    // ...
};

struct bcache {
    // ...
    struct spinlock lock; // 添加哈希表的锁
    // ...
};
```

3. 修改初始化函数： 在 `binit()` 函数中初始化哈希表的锁和缓冲区的时间戳字段。

```
void binit(void) {
    // ...
    initlock(&bcache.lock, "bcache");
    // ...
    for (i = 0; i < NBUF; ++i) {
        b->timestamp = 0; // 初始化时间戳
    }
    // ...
}
```

4. 修改 `brelse()` 函数： 释放缓冲区时，更新时间戳字段，根据时间戳来选择缓冲区进行重用。

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

5. 修改分配函数： 在 `bget()` 函数中，使用哈希表来寻找对应的缓冲区，若未找到则根据时间戳和引用计数来选择缓冲区进行覆盖重用。

```
struct buf *bget(uint dev, uint blockno) {
    // ...
    acquire(&bcache.lock);
    b = hashget(dev, blockno); // 从哈希表获取缓冲区
    if (!b) {
        b = chooseandevictabuf(dev); // 根据时间戳选择缓冲区进行覆盖重用
    }
    // ...
    release(&bcache.lock);
    // ...
}
```

实验中遇到的问题和解决办法

1. 问题：缓冲区管理效率低下： 原始的缓冲区管理方式使用链表结构，重用缓冲区时效率低下。

- 解决办法： 使用哈希表和时间戳： 引入线程安全的哈希表来管理缓冲区，使用时间戳字段来实现基于时间的 `LRU` 算法。这样可以更高效地管理缓冲区，提高文件 `I/O` 操作的性能。

实验心得

通过优化缓冲区管理方式，我学会了如何在实际系统中使用更高效的数据结构和算法来提升性能。这个实验不仅加深了我对操作系统中缓存管理的理解，还让我深刻体会到性能优化的重要性。

评分

```
== Test running bcachetest ==
$ make qemu-gdb
(8.1s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (85.9s)
== Test time ==
time: OK
Score: 70/70
```