操作系统课程设计

Xv6实验报告

Lab8 锁

|  |  |
| --- | --- |
| Name | 朱从周 |
| ID | 2351893 |
| Number | 42028701 |
| major | 软件工程 |



同济大学

Tongji University

1. **环境搭建**
2. 下载VMware虚拟机，并且下载 Ubuntu 镜像并安装 Ubuntu 系统。
3. 启动Ubuntu，安装本项目所需的所有软件，运行：

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

1. 下载xv6内核源码

$ git clone git://github.com/mit-pdos/xv6-riscv.git

1. 更新镜像源

$ sudo nano /etc/apt/sources.list

$ sudo apt-get update

1. 获取源码

$ git clone git://g.csail.mit.edu/xv6-labs-2021

$ cd xv6-labs-2021

$ git checkout lock

**1. 实验目的**

通过修改内存分配器的设计，以减少锁竞争，从而提高多核系统中的性能。具体来说，需要实现每个CPU都有一个独立的自由列表（free list），每个列表都有自己的锁。这样可以让不同CPU上的内存分配和释放操作可以并行进行，从而减少锁的争用。还需要实现当一个CPU的自由列表为空时，能够从其他CPU的自由列表中获取部分内存。

优化xv6操作系统中的缓冲区缓存（buffer cache），减少多个进程之间对缓冲区缓存锁的争用，从而提高系统的性能和并发能力。通过设计和实现一种更加高效的缓冲区管理机制，使得不同进程可以更有效地使用和管理缓冲区，减少锁竞争和性能瓶颈。

**2. 实验内容**

**Memory allocator**

在kalloc.c直接把 kmem 改成数组：

**struct** {

**struct** spinlock lock;

**struct** run **\***freelist;

} kmem[NCPU];

然后同时修改初始化函数 kinit ，该函数只在xv6启动的时候被一个CPU调用：

**void**

**kinit**()

{

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

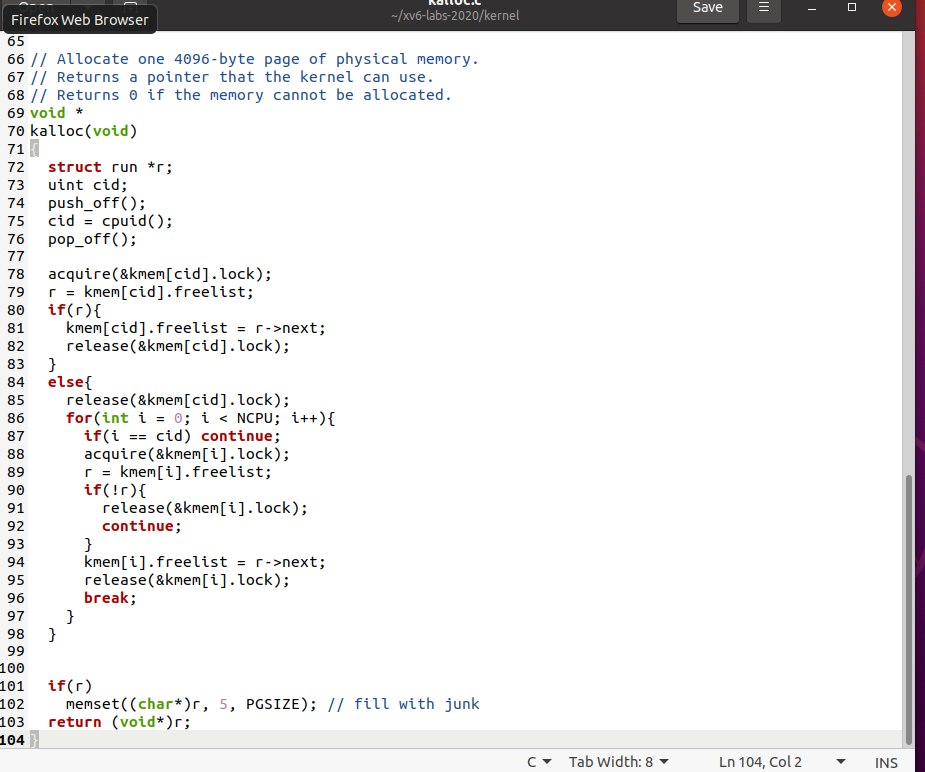
initlock(**&**kmem[i].lock, "kmem");

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

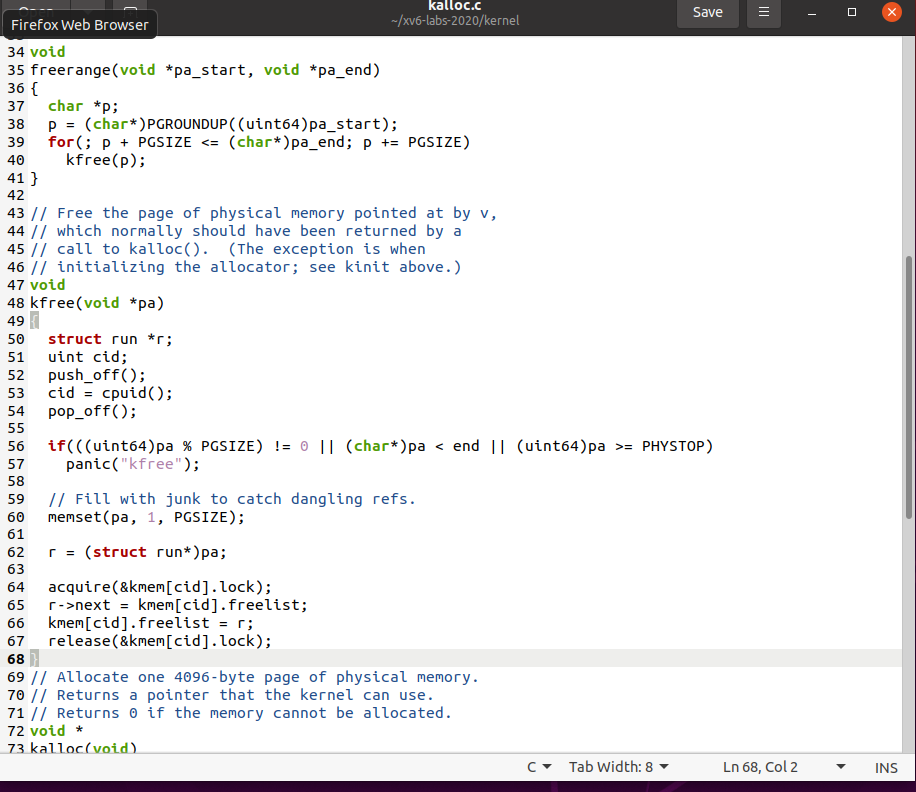
}

分配内存的过程。对于每个CPU对应的链表，如果链表为空，则需要从其它CPU的链表中偷取。

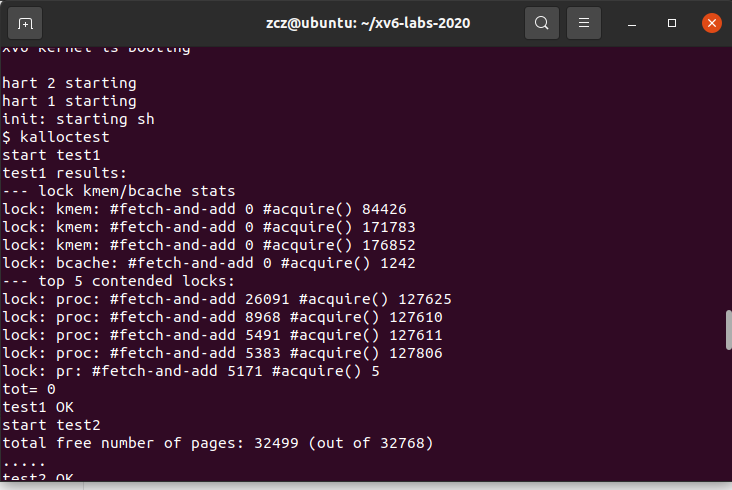
具体实现如下：

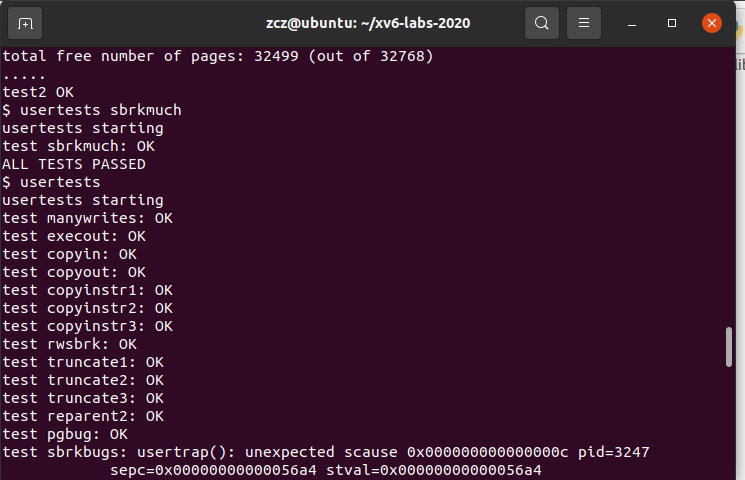


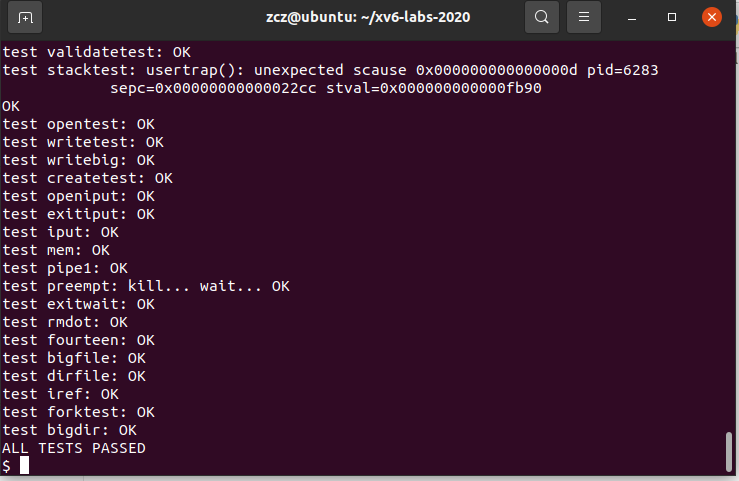
然后考虑释放内存，只需要在对应CPU的链表加入元素即可：



测试结果：







**测试通过！**

**Buffer cache**

首先，需要初始化：

**struct** bucket{

**struct** spinlock lock;

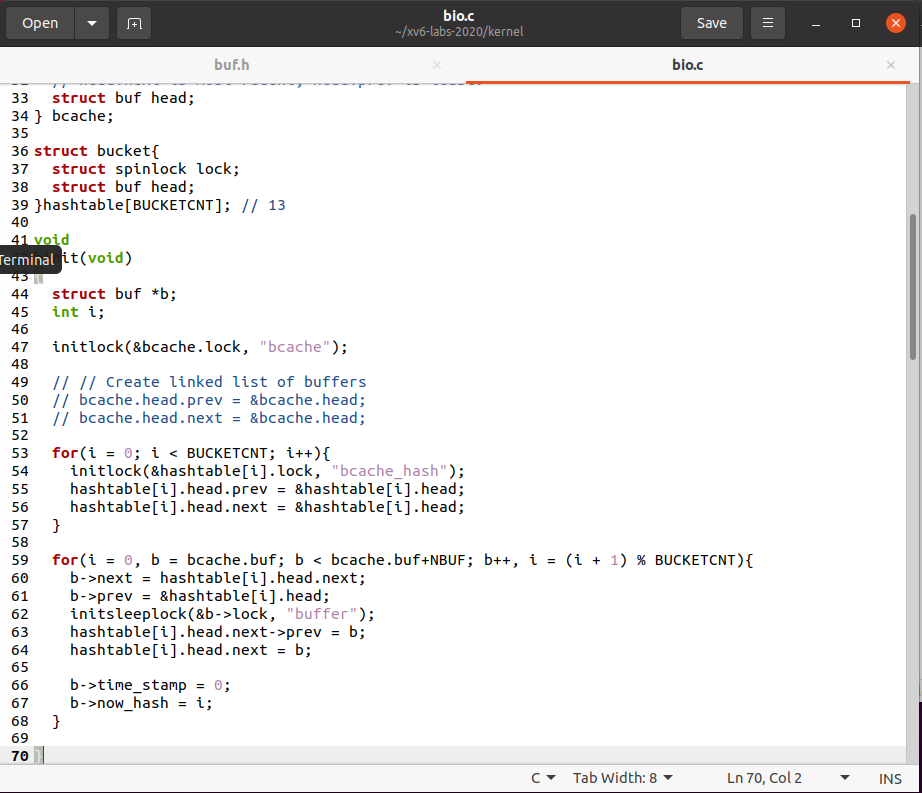
**struct** buf head;

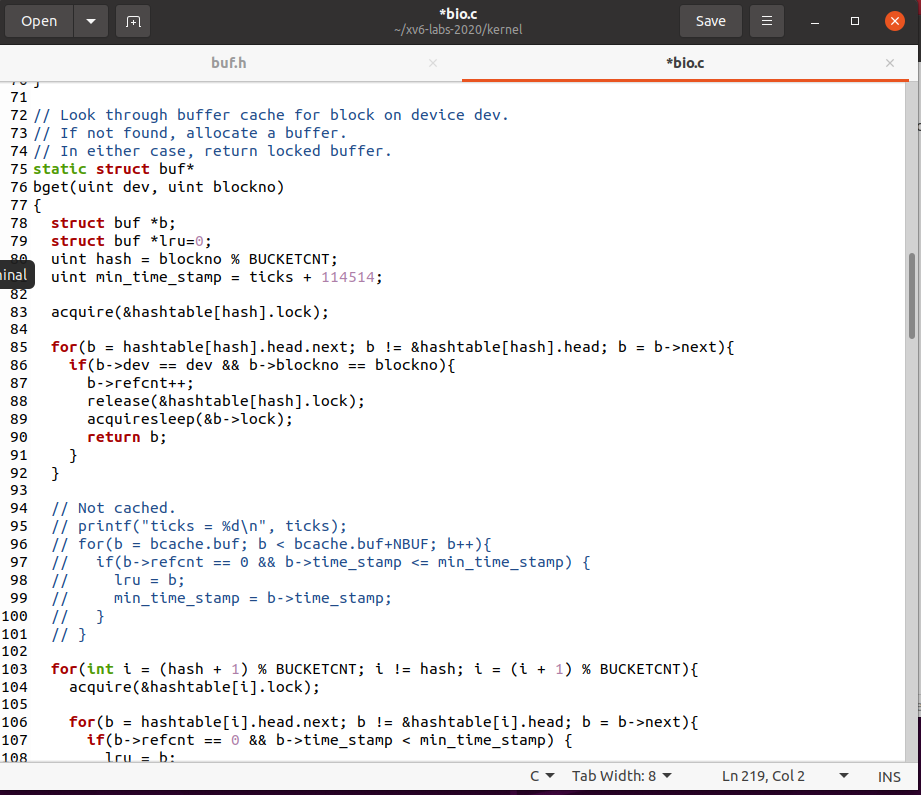
}hashtable[BUCKETCNT]; *// 13*

然后考虑如何获得buffer cache。首先在自己哈希值上的链表上找，如果没有则去其它小链表上轮换查找。

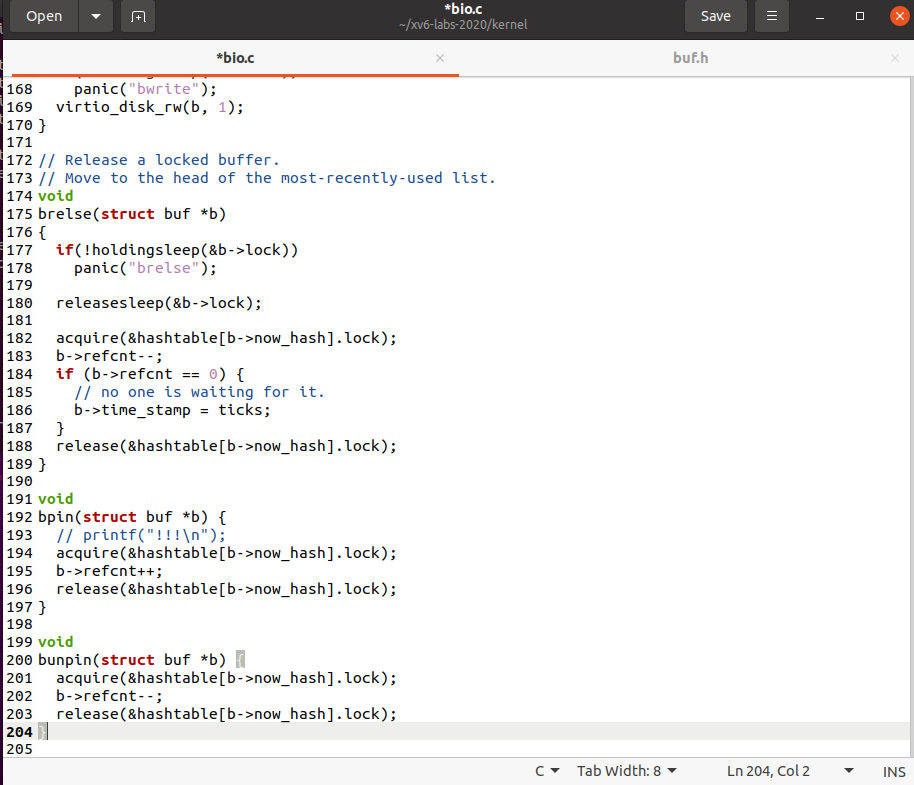
根据lab提示，我们采用lru策略。因此可以看到我上面给出的 buf 结构体相比于原有的结构体，多了时间戳 time\_stamp 字段。

代码截图如下：





最后考虑一下释放的操作：



同时，我们需要修改一下其余函数：

**void**

**bpin**(**struct** buf **\***b) {

*// printf("!!!\n");*

acquire(**&**hashtable[b**->**now\_hash].lock);

b**->**refcnt**++**;

release(**&**hashtable[b**->**now\_hash].lock);

}

**void**

**bunpin**(**struct** buf **\***b) {

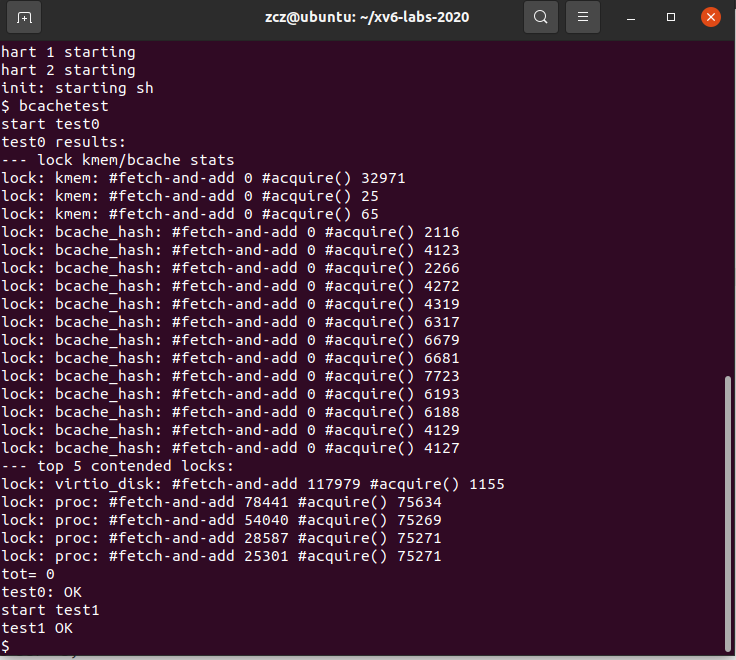
acquire(**&**hashtable[b**->**now\_hash].lock);

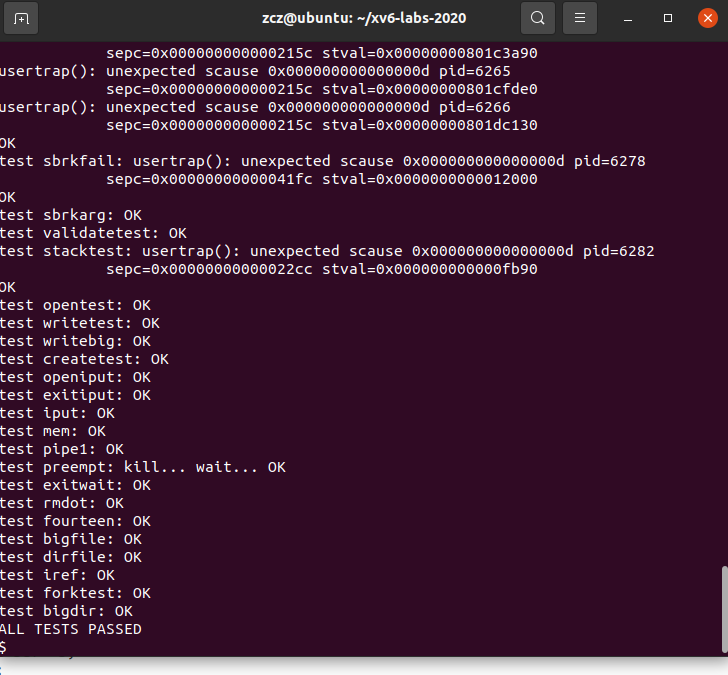
b**->**refcnt**--**;

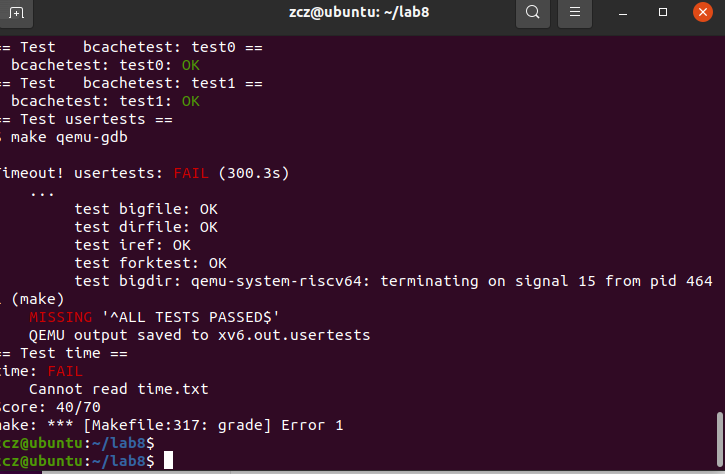
release(**&**hashtable[b**->**now\_hash].lock);

}

**测试结果：**







**3. 实验心得**

在开始实验之前，我深入理解了 Buffer Cache 的结构，包括缓存的大小、缓存块的管理方式以及数据结构等。这为我后续的编码和设计提供了重要的指导。

实验要求的功能涵盖了从缓存的获取、写入到缓存的释放等多个方面。我将实验任务分成不同的阶段，逐步实现和测试每个功能。这样有助于确保每个功能都能够独立正常运行。

在多线程环境下，缓存的并发访问可能导致竞态条件等问题。我使用了适当的同步机制，如自旋锁和信号量，以确保缓存的安全性和一致性。

此外，我注意到了这次设计的内核还包含了LRU（最近最少使用）算法，在Buffer Cache中，LRU算法用于确定哪个缓存块应该被释放，以便为新的数据块腾出空间。LRU算法的核心思想是，最近最少使用的缓存块应该被优先释放，因为它们更有可能在未来不会被再次访问。而在结构体的设计上，struct bucket 结构体包含一个锁 lock 和一个链表头 head。链表头 head 表示了一个双向循环链表，用于存储缓存块。这个链表是按照缓存块被使用的时间顺序来排序的，即最近被使用的缓存块位于链表头部，最少使用的位于链表尾部。这正是LRU算法的核心思想。