**实验部分一：适应多核的内存分配器（multi-core memory allocater）**

1、具体要求

1) 本实验必须使用 多核 的环境，否则数据将没有意义；

2) 请使用initlock()初始化锁，并要求锁名字以kmem开头；

3) 运行kalloctest查看实现的代码是否减少了锁争用（#test-and-set 没有获取到此锁的次数小于10则为通过）；

4) 运行 usertests sbrkmuch 以测试修改代码后系统是否仍可以分配所有的内存；

5) 运行usertests，确保其能能够全部通过；

6) kalloctest和usertests的输出如下图（锁争用的次数大大减少），具体的数据会有所差别。

2、思路

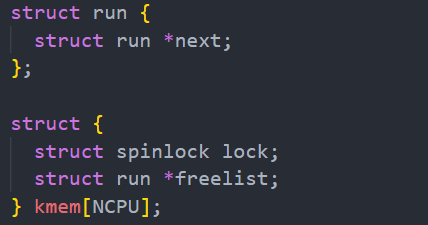
代码之前的实现是多进程抢占单个cpu的kmem内存。现在我们改为多核环境下的实现，为每一个cpu分配一个kmem，让进程在多个kmem之间竞争，提升了利用率。

注意到，我们优先使用当前cpu的kmem，通过调用cpuid获取当前cpuid，注意在此前后需要开关中断，也就是调用push\_off和pop\_off。

另一点是lock的使用。每一个kmem对象都具备lock成员，在操作kmem.freelist前后需要对kmem.lock进行acquire和release。

kalloc.c下面有kinit，kfree，freerange，kalloc四个函数，由于freerange是复用kfree的，所以我们需要修改其中kinit，kfree，kalloc三个函数。

首先观察结构



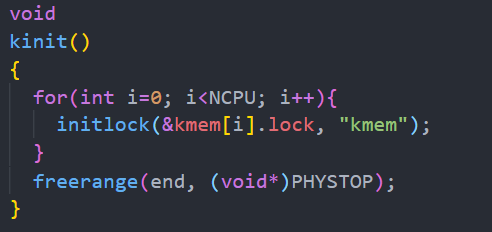
显然是描述链表节点的，所以基本确定了kmem是链式存储的，而kmem[NCPU]数组存放的是链表头结点。

3、代码

思考：代码变了什么，没变什么？你是如何做到在增加功能的同时没破坏原有功能的？

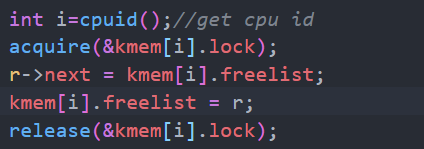
本次修改的补丁性很强，只是将原本的kmem结构扩展为kmem结构数组，针对数组的元素也就是一个kmem对象，我们并没有减少其逻辑。

1.在kinit函数中，逻辑基本没变化：



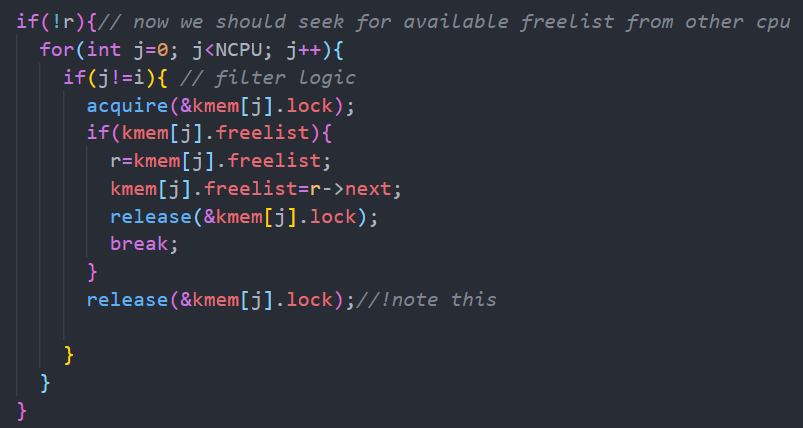
由初始化变量修改为初始化数组。

2.在kfree函数中，逻辑基本没变化：



仍然是链表的析构逻辑。

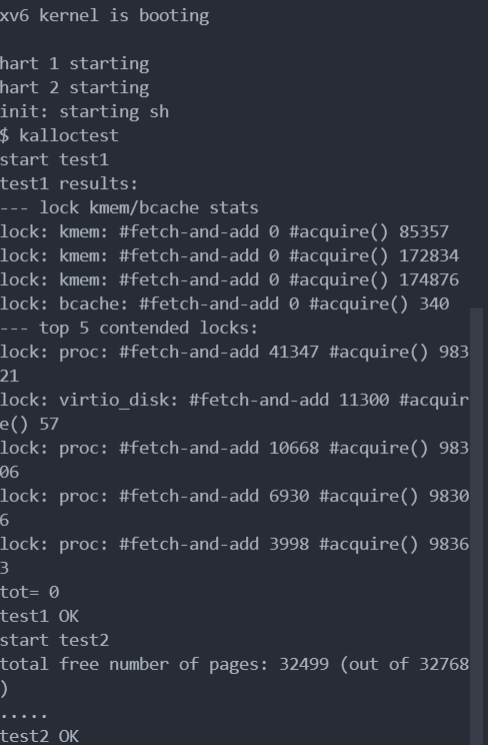
3.在kalloc函数中，针对kmem对象我们增加的逻辑如下：



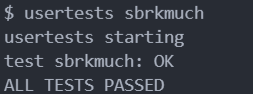
这是一段filter-logic，用于在当前cpu的kmem被占用的时候，寻找其他cpu的可用kmem。**是资源竞争的实现。**最内层if中，仍然是仍然是链表的析构逻辑。

4、测试

kalloctest测试



usertests sbrkmuch测试



另外，在boot和exec sh的时候，明显感觉使用multi-core memory allocater的系统变快了。

**实验部分二：锁管理的磁盘缓存（Buffer Cache With Lock）**

1、具体要求

1) 理想状态下，bcachetest中数据块缓存相关的所有锁test-and-set的总和应该为0，但本实验中总和不超过500即可；

2) 请修改bget()和brelse()，使得缓存区并发的查询和释放不容易发生锁争用，比如，不是所有流程都得等bcache.lock；

3) 同样要求usertests中的用例全部通过，最后的输出如下（具体数据有所出入）：

2、思路

这种磁盘缓存技术在linux里面好像是叫slab机制？

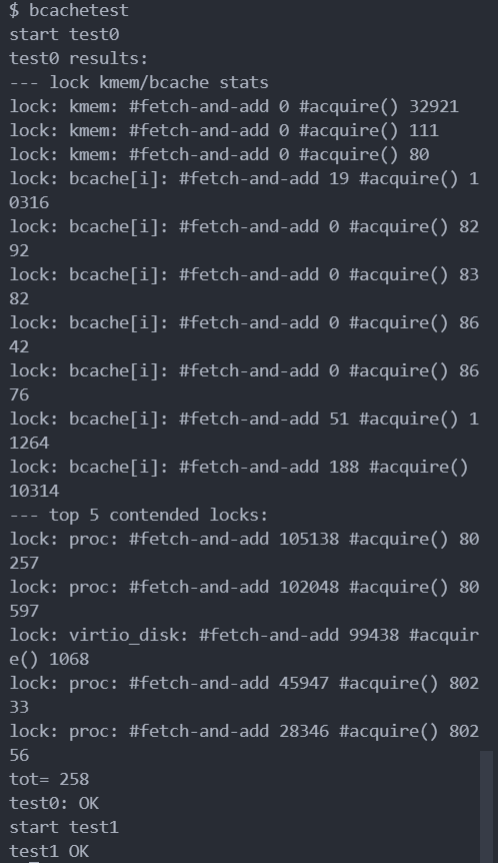
主要思路是使用哈希表维护

另外一点就是，我们需要使用一个spinlock数组struct spinlock bucket\_locks[NBUCKETS];来维护对bucket的互斥访问。

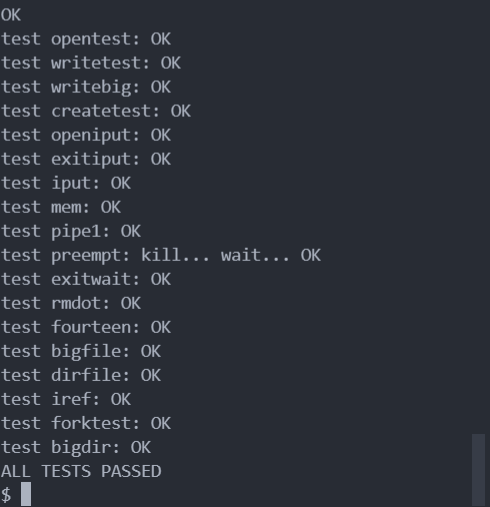
3、代码

4、测试

bcachetest测试



usertest测试



grade脚本评分的就不测了，太烧cpu了，应该就是重复一下上面两个测试，都可以通过的。

5、实验体会

本质上还是数据结构的综合运用， struct buf buf[NBUF]; 这个数组是顺序表和链表的复合体，是LRU操作的核心。

实际上，这个LRU系统在LeetCode上出现过： https://leetcode-cn.com/problems/lru-cache/solution/

**实验部分三：回答问题**

1. 什么是内存分配器？它的作用是？

内存分配器是分配内存的。

1. 内存分配器的数据结构是什么？它有哪些操作（函数），分别完成了什么功能？

核心数据结构是链表，操作原语有init，free，freerange，alloc。作用分别是：init调用freerange初始化整个链表；free释放单个节点使其变得可用；freerange释放多个节点使其可用；alloc分配链表中的可用节点。

1. 为什么指导书提及的优化方法可以提升性能？

单核变多核，提升了多核CPU的利用率。

1. 什么是磁盘缓存？它的作用是？

磁盘缓存缓存了磁盘中的数据，当内存需要的数据命中的时候（当然，基于局部性原理，热数据是很容易命中的），直接取缓存中的数据，提升了效率。

1. buf结构体为什么有prev和next两个成员，而不是只保留其中一个？请从这样做的优点分析。

说明这是一个双向循环链表，可以逆向迭代和循环迭代，可以快速找到头结点和尾节点，是空间换时间的结构。

具体场景比如，在双向链表中位于中间的节点，可以通过prev指针访问到头结点，这一点是单向链表无法完成的。再比如，bcache.head直接提供next找到most recent node

1. 为什么哈希表可以提升磁盘缓存的性能？可以使用内存分配器的优化方法优化磁盘缓存吗？请说明原因。

使用哈希表，将各块块号blockno的某种散列值作为key对块进行分组，并为每个哈希桶分配一个专用的锁。显然增加了锁的数量，侧面反映了并发度的增加。

“内存分配器的优化方法”应该是指数组的方式，哈希表和数组还是存在差别的。哈希表能散列，其插入和删除操作的时间复杂度是O(1)的，而顺序表数组的插入和删除的时间复杂度是O(n)的，所以哈希表更适用于磁盘缓存的场景。从原子操作的角度，哈希表O(1)的时间也是自旋锁能接受的，毕竟自旋锁被长时间持有，会降低CPU利用率。