Lab: locks

Git链接[Jeery1/xv6-6.S081 at lock](https://github.com/Jeery1/xv6-6.S081/tree/lock)

**实验目的：**理解锁的基本原理

临界区保护：学习如何使用锁保护共享资源，避免数据竞争（Data Race）。

锁的实现：分析自旋锁（Spinlock）和睡眠锁（Sleep-lock）的底层机制，理解原子操作（如 xchg、test-and-set）的作用。

死锁与性能：研究锁的粒度、锁竞争（Contention）对系统性能的影响，以及如何避免死锁（Deadlock）。

2. 实现与优化锁

自旋锁优化：在xv6中改进自旋锁的实现（如减少锁冲突、降低CPU空转开销）。

睡眠锁应用：针对长时间持有的锁（如文件系统操作），用睡眠锁替代自旋锁，减少CPU资源浪费。

锁的调试：通过断言和日志验证锁的正确性，定位死锁或竞争条件问题。

3. 实际场景中的应用

内存分配器：为 kalloc/kfree 设计高效锁机制，解决多线程竞争问题。

文件系统：保护文件系统元数据（如inode、block cache）的并发访问。

进程调度：理解xv6调度器（Scheduler）中锁的作用，避免进程状态不一致。

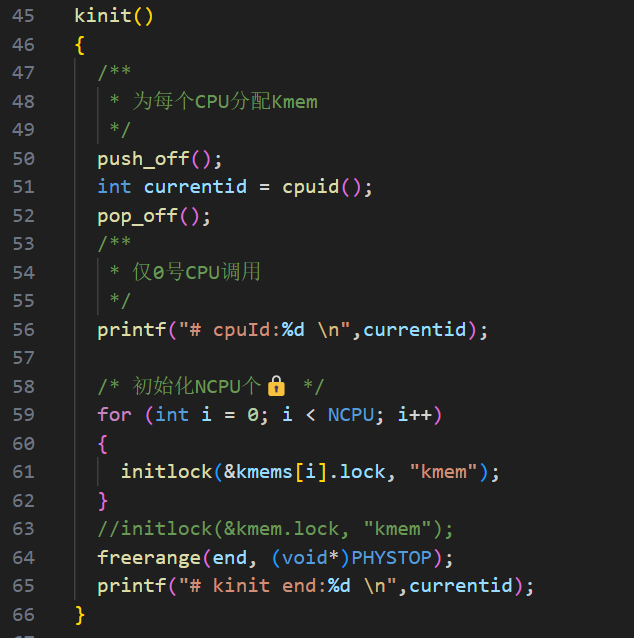
**实验步骤：**

**Memory Allocator**

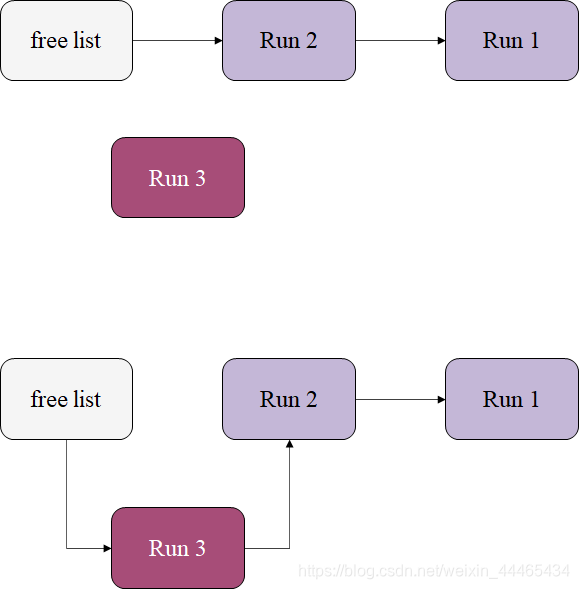
在kalloctest中，三个进程不断通过调用kalloc和kfree来增长或释放它们的内存空间，然而在kalloc和kfree中只有一个锁kmem.lock可以保证这三个进程的访问不冲突，这样一来，导致了大量的无效锁请求

出现lock contention的根本原因是kalloc()中只有一个free list，该free list只被一个lock保护。为了消除lock contention，我们必须想办法避免单free list、单lock问题。最基本的思想就是为每一个CPU都维护一个free list，并且每一个free list都有一个独自的lock。这样一来，不同的CPU就能并行地利用kalloc()与kfree()分配或释放内存了，因为每个CPU都只会操作不同的free list。当一个CPU的free list为空，但是另一个CPU的free list还有空闲块时，我们就应该从有空闲free list的CPU处“偷”一个空闲内存块。

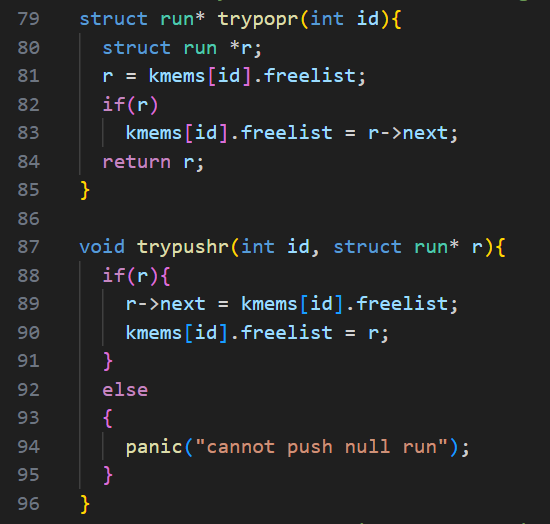
那么，首先我们需要实现一个多free list的初始化，相关代码如下：



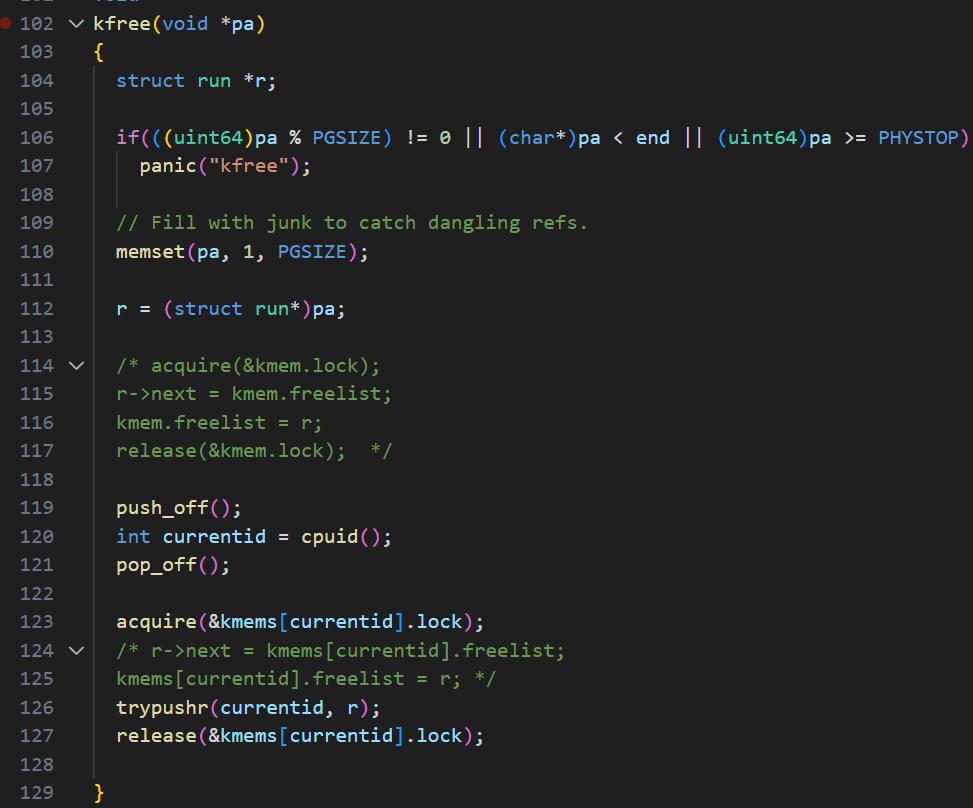
接下来，通过阅读源码，我们可以知道kalloc.c中对free list的操作是**头插法**，原理见下图：



为了方便，我们可以将free list的插入push()和弹出pop()操作先封装为函数，方便调用。注意，由于这里我们有多个kmem，因此要用kmems[id].freelist获取相应的free list



先完成kfree()，调用cpuid()获取相应的id，然后将被释放的块插入相应的free list就好：

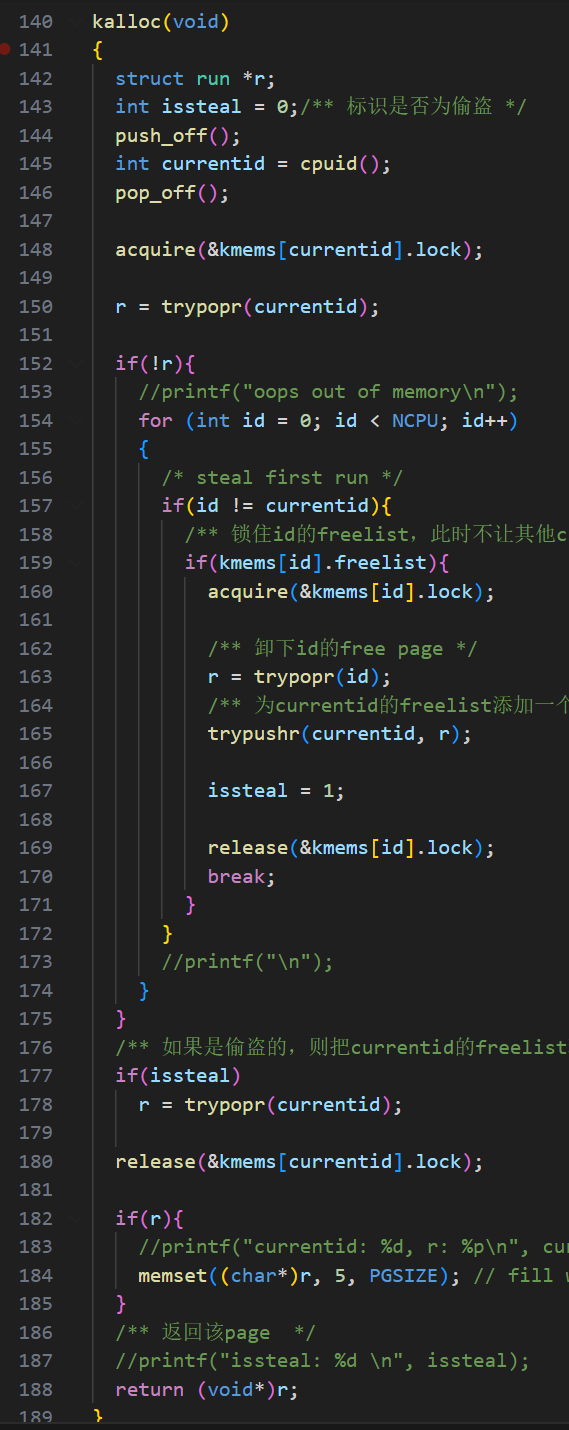


最后，完成kalloc()。首先，如果当前CPU的free list不为空，那就直接pop一个run出来，将之初始化后，返回给调用者；如果当前CPU对应的free list为空，就需要去查询其它CPU对应的free list，找到空闲的块r后，按照如下步骤操作：

将r插入自己的free list中；

将r从自己的free list中弹出，将之初始化，返回给调用者；

这就完成了“偷”的过程。当然在操作free list的过程中要注意lock的使用，代码实现如下：

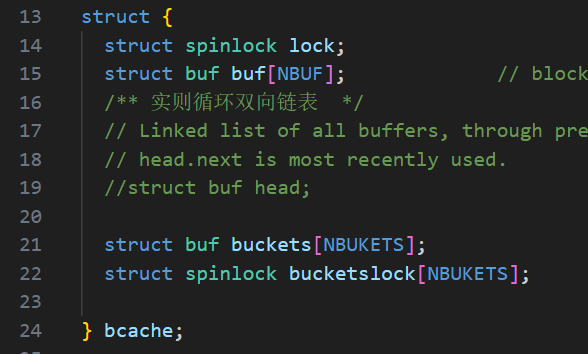


至此，我们就完成了Memory Allocator

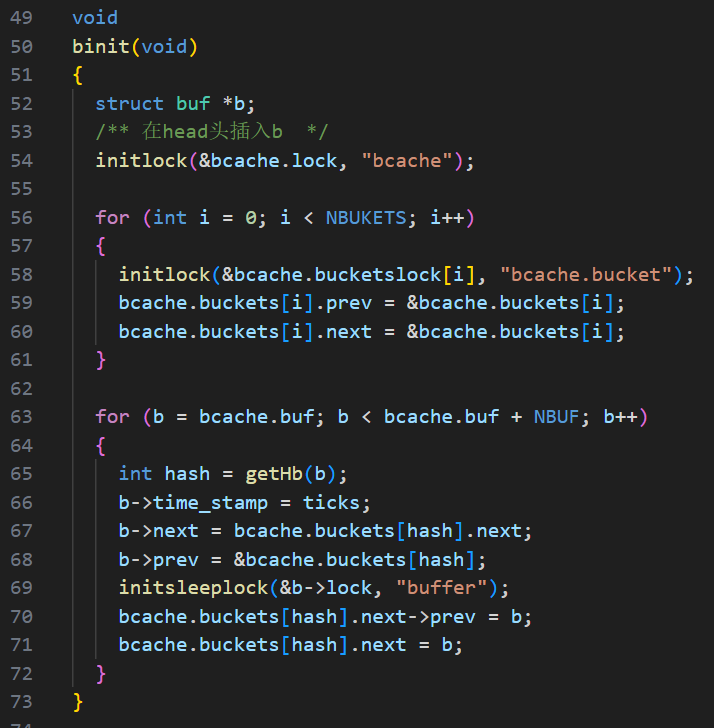
**Buffer Cache**

Buffer Cache出现的问题和Memory Allocator中的问题其实类似，只不过这里是管理磁盘缓存。试想这样一个情况：三个进程大量读写磁盘，而磁盘缓存只有一个lock，这就导致三个进程的竞争异常激烈。本任务就是要解决这个问题。

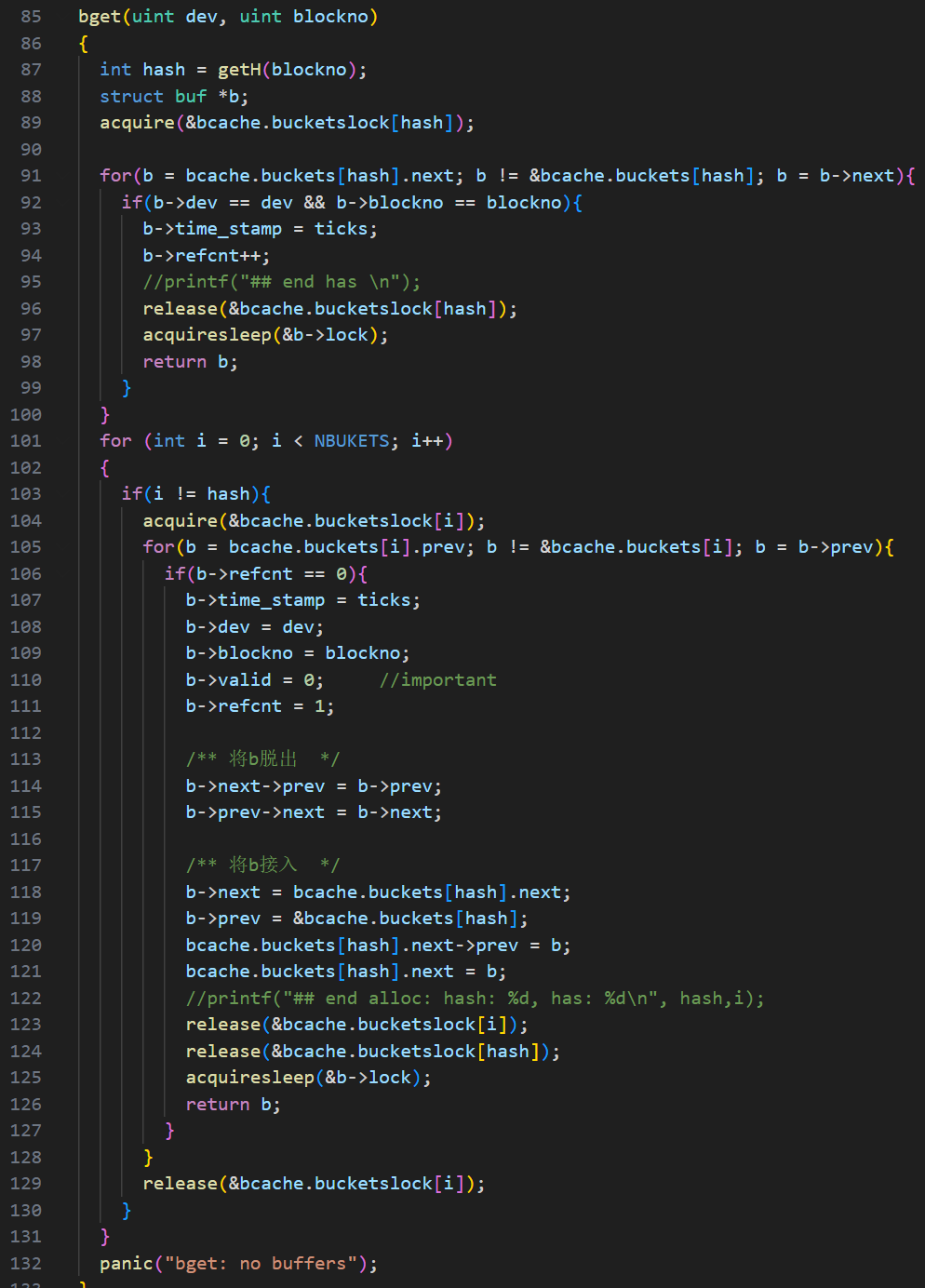
首先修改bcache，以使其支持哈希桶结构：



其次初始化bcache：



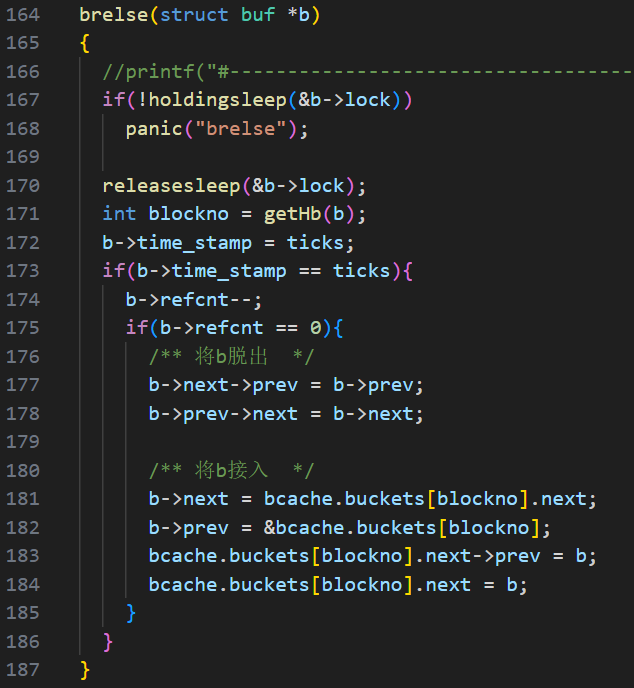
然后完成bget()，其实完成这一部分完全就是将原代码改写为支持哈希桶即可。需要注意的是，当在自己的哈希桶中没有找到相应的扇区时，应该从别的哈希桶处“偷”，这就和Memroy Allocator做的事情一样了。

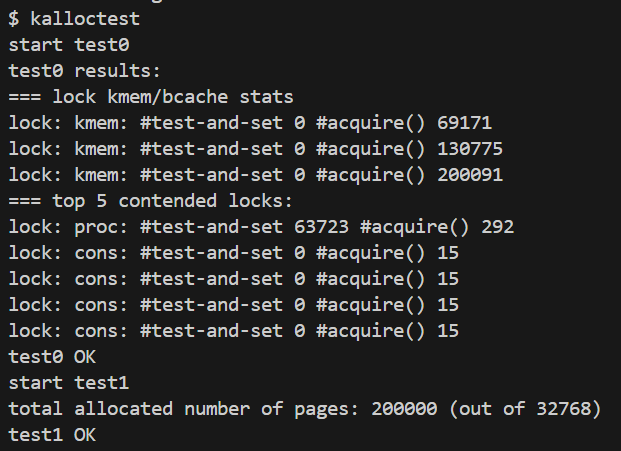


接下来，我们实现brelse()。

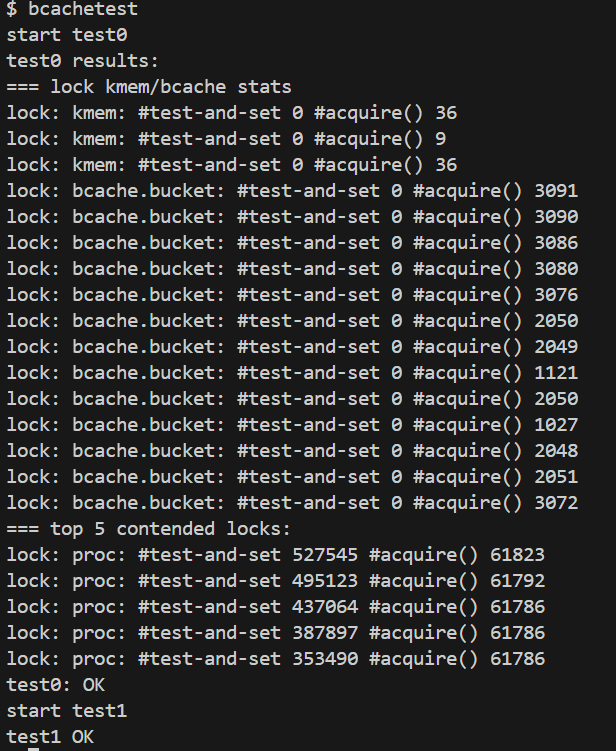
首先检查缓冲区是否持有睡眠锁（holdingsleep()），如果没有则 panic。

然后释放睡眠锁（releasesleep()）。获取缓冲区的块号（getHb(b)）。将当前系统时钟 ticks 赋值给缓冲区的 time\_stamp。减少引用计数 refcnt，如果引用计数减到 0，则：将缓冲区从当前哈希桶链表中移除，再将缓冲区插入到对应哈希桶链表的头部（MRU 位置）。



**实验现象：**

运行kalloctest成功



运行bcachtest成功

**实验中遇到的问题和解决办法：**

kalloc 锁竞争引发死锁

问题描述：

在 kalloc() 中，若两个 CPU 同时申请内存并触发 steal（从其他 CPU 的空闲列表偷页），可能因锁获取顺序不一致导致死锁。

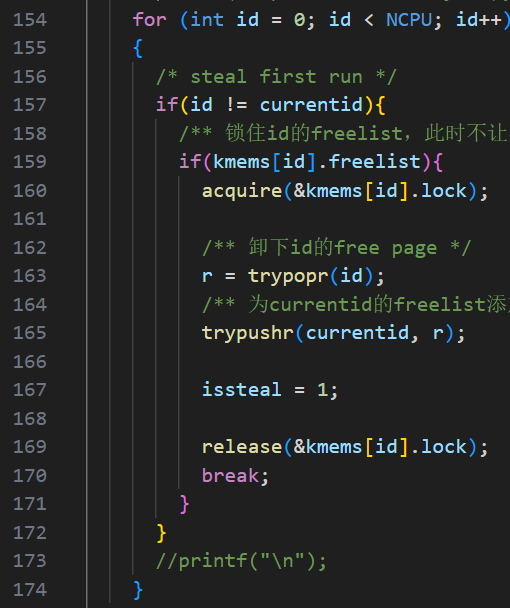
调试过程：

运行 usertests 时偶发系统挂起，backtrace 显示多个 CPU 卡在 acquire(&kmem.lock)。

发现 steal 操作会遍历所有 CPU 的 kmem，可能形成循环等待。

解决办法：

固定锁获取顺序：为每个 CPU 的 kmem 分配唯一 ID，steal 时按 ID 从小到大加锁，若未偷到页，则释放所有锁后重试，避免长时间持锁。



**实验心得：**

通过本次锁机制实验，我对操作系统并发控制有了更深刻的认识。在优化bcache锁的过程中，我体会到锁粒度设计的重要性。死锁问题的调试经历尤其珍贵，通过分析backtrace和固定锁获取顺序，我掌握了预防死锁的实用技巧。这次实验不仅让我熟悉了自旋锁、睡眠锁等基础同步原语，更培养了我分析锁竞争、平衡性能与正确性的能力。这些经验让我对操作系统的并发设计有了更立体的理解，也为后续学习更高级的同步机制打下了坚实基础。