Lab: Allocator for xv6

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

1. 利用bd\_malloc 实现文件动态分配

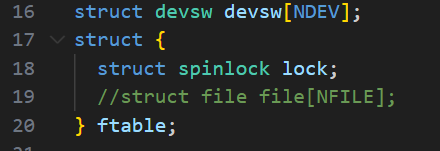
**实验目的：**

在xv6的file.c中定义了一个ftable，如下所示。我们可以很清晰的发现，ftable是通过声明静态file数组来实现文件的分配，这样一来，就导致一个进程只能打开固定数量NFILE的文件，通过阅读源码可知NFILE为100。本任务要求利用Buddy Allocator动态分配文件，这样一来可打开的文件数就能够大于NFILE了。

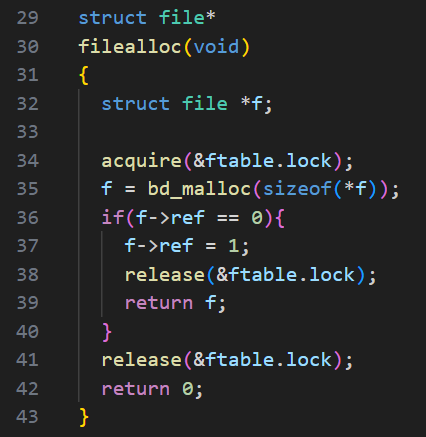
**实验步骤：**

修改file.c，使得文件分配方式变为利用Buddy Allocator动态分配，然后重新make qemu，通过运行alloctest来检查修改是否成功。

修改代码过程如下：



注释掉本来用于分配文件的静态数组



修改filealloc函数

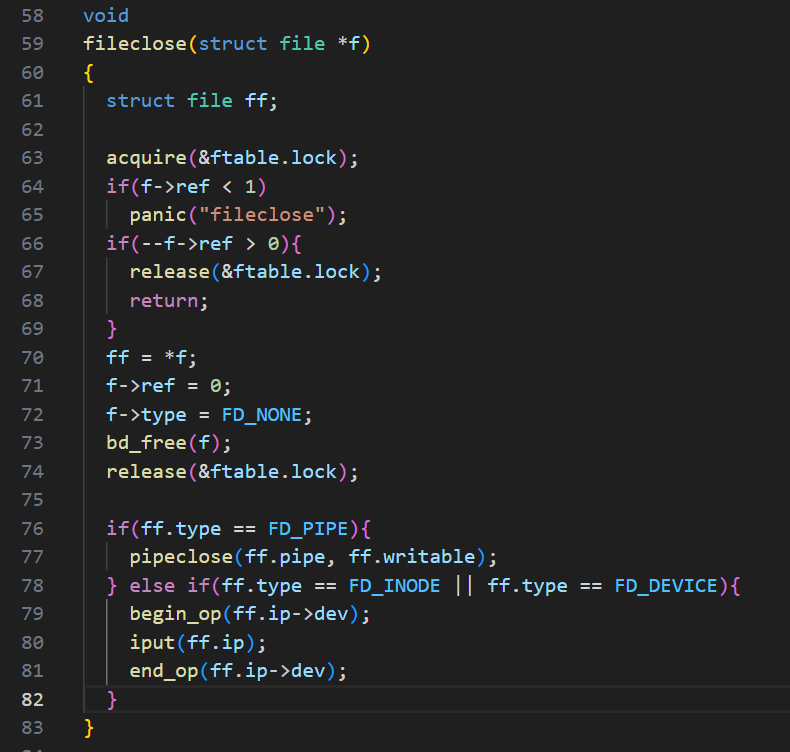
filealloc() 的主要职责是：

分配内存：从 Buddy 分配器（bd\_malloc）中申请一个 struct file 的内存空间。

初始化字段：将文件结构的关键字段设置为初始状态。

线程安全：通过自旋锁 ftable.lock 保护全局文件表 (ftable) 的并发访问。

首先使用 Buddy 分配器动态分配内存。如果分配失败（返回 0），打印错误并返回空指针。然后显式初始化所有字段，避免未定义行为。



修改fileclose函数

fileclose() 的主要职责是：

减少引用计数：管理文件的引用计数，决定是否真正释放资源。

资源清理：根据文件类型（管道、inode 或设备）执行特定的关闭操作。

内存释放：最终通过 Buddy 分配器 (bd\_free) 回收文件结构的内存。

通过 ftable.lock 保护对文件表 (ftable) 的并发访问，检查 ref < 1 防止非法释放（触发内核恐慌 panic），减少引用计数后，若 ref > 0，表示仍有其他引用存在，直接返回。然后进行资源清理，管道文件：调用 pipeclose 关闭管道，释放缓冲区。inode/设备文件：begin\_op/end\_op：包裹磁盘操作的事务逻辑（确保原子性）。iput：减少 inode 的引用计数，必要时释放 inode 和磁盘块。最后将 struct file 的内存归还给 Buddy 系统。

1. 优化Buddy Allocator

**实验目的：**

xv6的Buddy Allocator在buddy.c中实现。通过阅读源码，可以发现Buddy Allocator中的每种规格大小的内存块都保留一个比特位，用于标识该块是被占有还是空闲，见下面的代码。这样一来Buddy Allocator的就会使用大量的内存用于保存这些比特位，这导致了Buddy Allocator的空间利用效率变得低下。本任务要求利用如下策略对现有的Buddy Allocator进行优化：只用一个比特位标识一对buddy内存块（两块）的使用情况。例如，对于buddy块B1和B2，这个比特记录了“B1 is free XOR B2 is free”（即，B1空闲异或B2空闲，进一步的，B1、B2其中一个空闲，则该比特为1；B1和B2都空闲或者都被占用，则该比特为0)。这样一来，一旦buddy块中的一个块被分配或者释放，都需要调整这个比特的值。

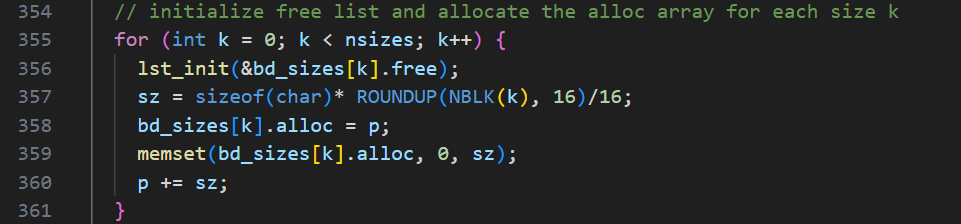
为了便于理解，我们给出下表以说明这个过程：



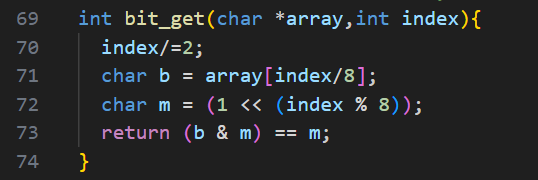
我们仅分析“B1空闲、B2被占用”的状态，其他场景类似。在该状态下，此比特值为1，这时如果我们释放B2，那么我们就知道B1和B2都处于空闲状态，可以进行合并。按照xv6官方指导书的说法，运用这个优化策略，每一对buddy块就能节省1比特，当xv6利用优化后的Buddy Allocator管理大约128MB的空闲内存时，该方案就可以节省大概1MB的内存。

**实验步骤：**

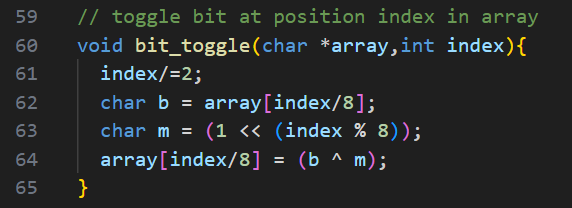
修改buddy.c，使得可以使用一位比特来记录两个，然后重新make qemu，通过运行alloctest来检查修改是否成功。

代码修改过程如下：  


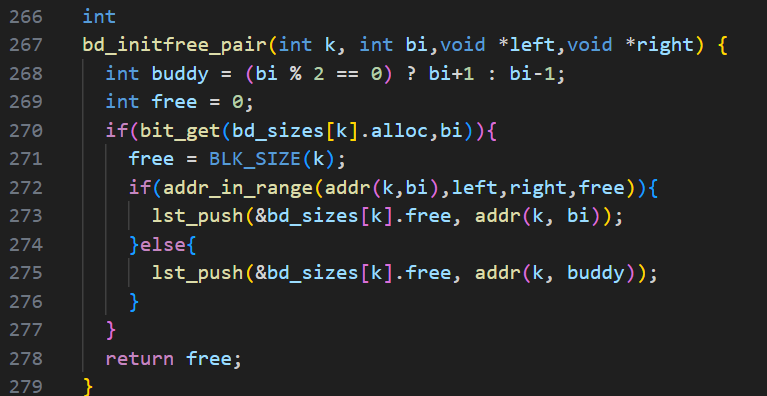
首先，调整bd\_init()中分配的alloc数组的大小。此时我们仅需要原来一半大小的数组，这里需要注意的是，为了保证分子是16的整数倍ROUNDUP(NBLK(k), 16)是必要的。



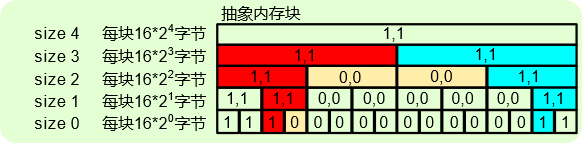
接着编写bit\_get()函数，以获取Buddy的公用比特位。



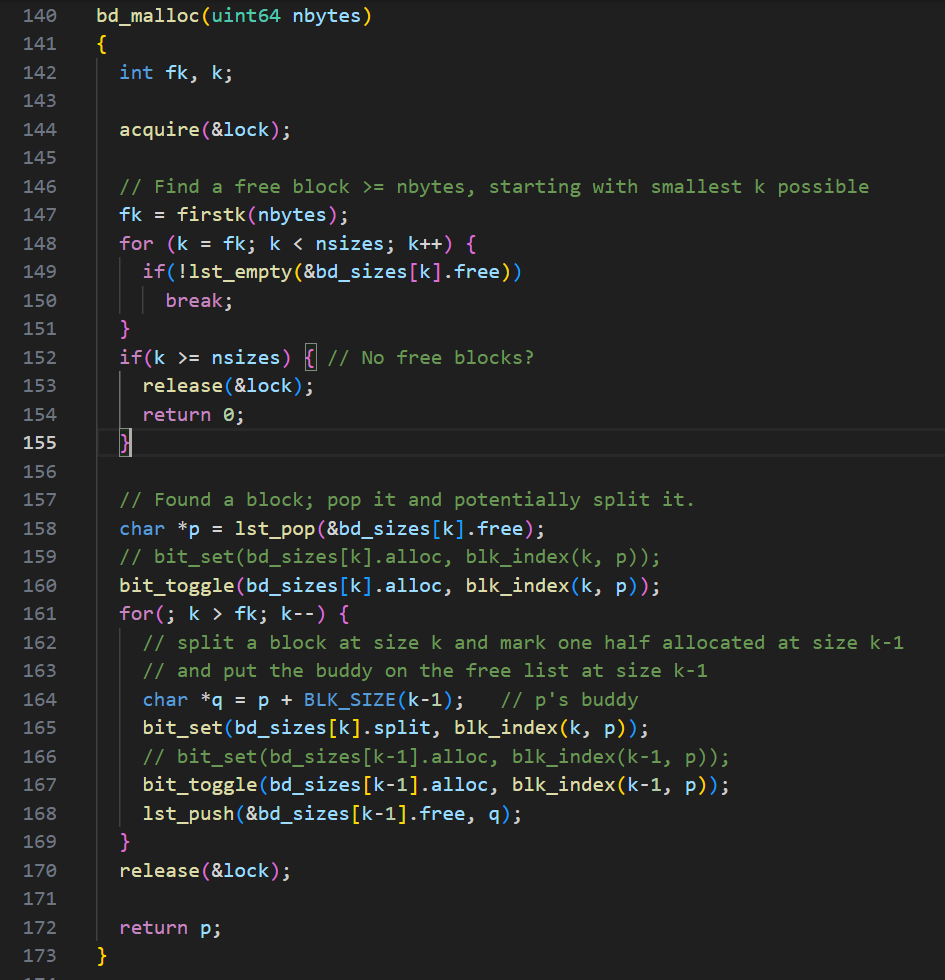
编写 bit\_toggle()函数，以实现一对Buddy公用一个比特位的操作



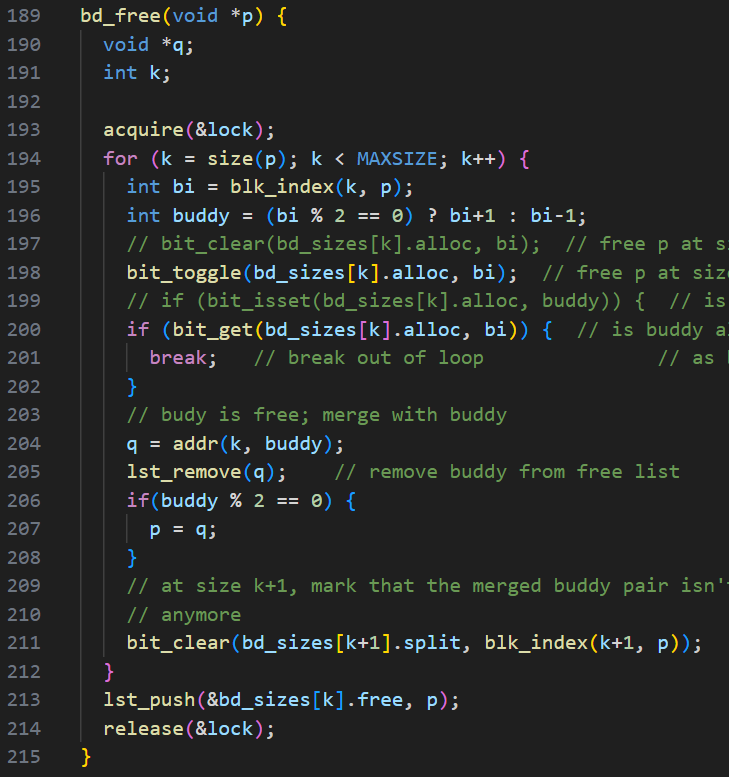
修改bd\_initfree\_pair函数，使之通过bit\_get()来判断是否应该将某个内存块加入空闲列表。这里用到了一个技巧，通过判断if(bi == left)即可决定究竟是将buddy块加入空闲列表还是将bi内存块加入空闲列表。为什么可以这样呢？回顾我们在Code Thru中观察到的现象一：应该加入到free中的内存块只出现在每一个size的两端。再者，我们观察传入的bd\_initfree()函数的参数：p与bd\_end，它们分别表示meta段末尾地址、无效内存的起始地址，接下来，我们继续借用我工的图，重绘以标记p与bd\_end的位置，如下图所示（红色代表p，蓝色代表bd\_end）：



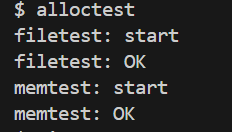
接着，我们观察传入bd\_initfree\_pair的参数，通过阅读源码可知，为：left = blk\_index\_next(k, p)与right = blk\_index(k, bd\_end)。其中left代表的是在相应size k对应的内存块p的后一块，right代表的就是相应size k的bd\_end对应的内存块。值得注意的是，在left对应的块不是bd\_end对应的块的情况下（如size 3），其应该是空闲的，而right对应的块永远都是已被分配的。先来考察size 3的情况，由于size 3中仅有的一对buddy都被分配了，因此它们谁也不应该加入到free中；再来考察非size 3的情况，由于left对应的块永远为空闲，因此其buddy一定被占用（因为mutual\_bit\_get(bd\_sizes[k].alloc, bi)），我们应该将left块加入到free中，而right对应的块永远被分配，因此其buddy一定为空闲（同样因mutual\_bit\_get(bd\_sizes[k].alloc, bi)），我们应该将其buddy加入free。



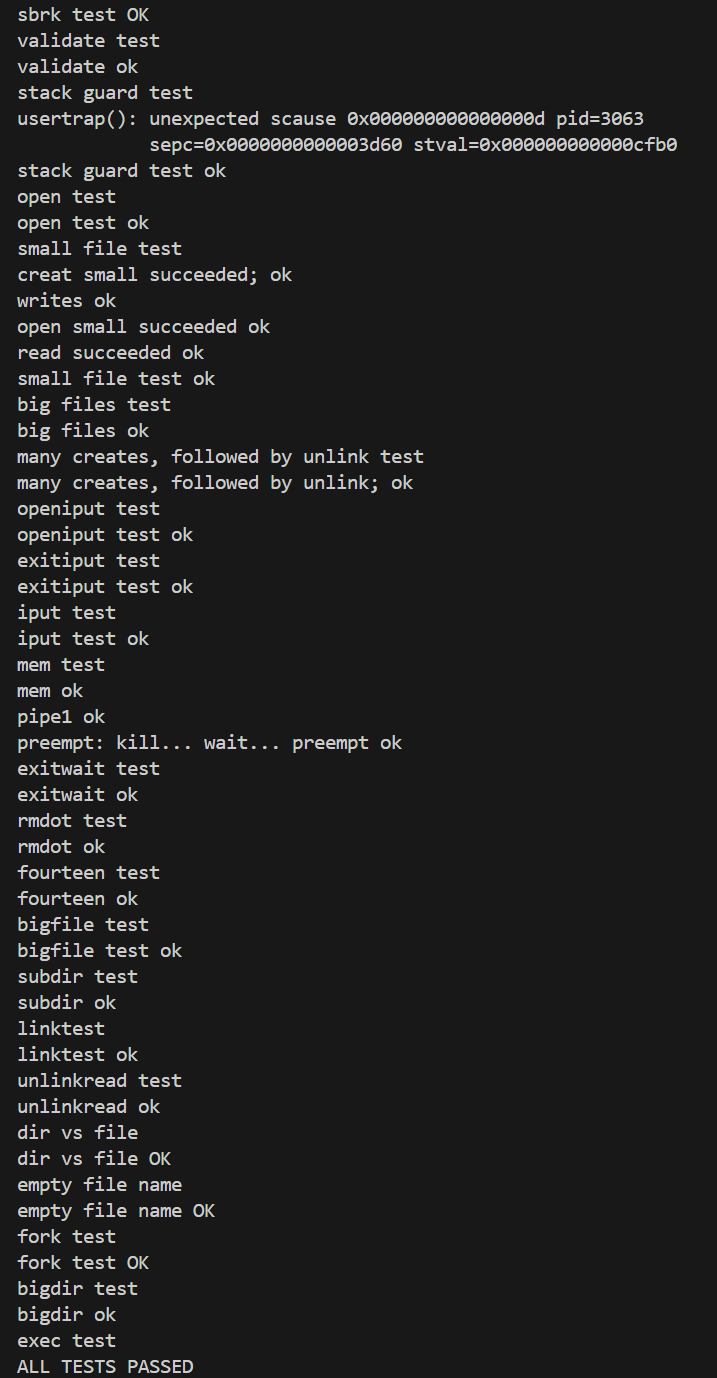
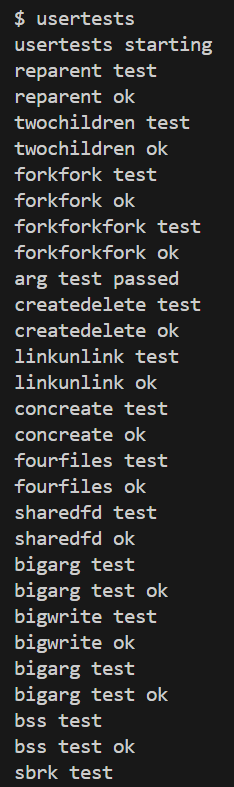
接下来，我们修改bd\_malloc()。修改方式较为简单，只需要将所有的bit\_set改为bit\_toggle()即可。



接着，我们对bd\_free如法炮制

**实验现象：  
**

执行alloctest，通过



执行usertests，全部通过。

**实验中遇到的问题和解决办法：**

问题描述：

在 free 操作后，发现相邻的空闲内存块没有被合并，导致内存碎片化严重。

调试过程：

打印空闲链表，发现释放后的块仍然独立存在，未与相邻块合并。

检查 free 函数逻辑，发现未正确处理前后块的合并情况。

原因分析：

释放内存时，仅简单插入空闲链表，未检查物理地址相邻的空闲块。

未更新合并后块的头信息（header），导致后续分配错误。

解决办法：

在 free 时检查前后块是否空闲，并进行合并：

问题描述：

在多进程测试时，偶尔出现内存管理数据结构（如 free\_list）损坏，导致系统崩溃。

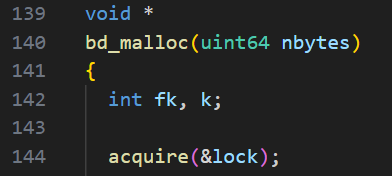
调试过程：

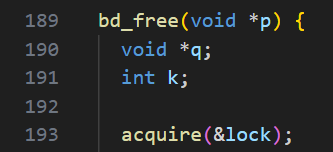
使用 printf 调试，发现 malloc 和 free 并发执行时可能同时修改链表。

未加锁保护共享数据（如 free\_list）。

解决办法：

在 malloc 和 free 中加锁，确保线程安全：





**实验心得：**

通过本次实验，我深刻理解了动态内存分配器的核心挑战与实现细节。内存碎片的优化管理是关键，合理的合并策略能显著提升内存利用率；在多线程环境下，锁机制对数据安全的保护必不可少，这让我对操作系统的并发控制有了更直观的认识。调试过程中，打印空闲链表和内存块信息的方法极大提升了问题定位效率。此外，内存对齐问题和分配算法的选择（如最佳适应策略）让我意识到底层系统开发中对硬件特性的考量有多么重要。这次实验不仅让我掌握了内存管理的基本原理，更启发我思考如何进一步优化分配策略。