同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 9

2151767 薛树建

日期：2023 年 8 月 18 日

目录

Lab9: file system

1. Large files (moderate)

1) 实验目的

本实验目的是让我们去增大 xv6 文件的大小，已知 xv6 文件 inode 采用的是二级索引。含有 12 个直接块和一个间接块，一个块最多可容纳地址数量为 256（1024/4），因而此时文件大小为 12+256=258。

我们本实验需要将其中的一个直接块改成三级间接块，这样便可以增大文件大小至 11+256+256*256=65803。

2) 实验步骤

实验思路较为简单，就是对源码进行修改将一个直接块改成二级间接块。

a) 首先需要修改 fs.h 下的一些参数，并增添新的参数。

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT ((BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint)))
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
#define NADDR_PER_BLOCK (BSIZE / sizeof(uint))//每个块中的地址的数量，256
```

由于我们修改了 NDIRECT 的值，其为直接块的数量，由此需要对 dinode 于 inode 两个结构体中的变量做修改。由原先的+1 变为+2,因为索引项的数量仍然为 13 个保持不变。

```
  uint size;
  uint addrs[NDIRECT+2];
};
```

b) 接着我们修改 bmap 函数，在其中增添符合二级间接块的映射。

bn 记录的为该块在该文件中的次序。若其减去 NINDIRECT 仍未正数，表明其属于二级间接块。

```
bn -= NINDIRECT;
// 二级间接块的情况
if(bn < NDINDIRECT) {
  int index2 = bn / NADDR_PER_BLOCK; //二级间接块的位置
  int index1 = bn % NADDR_PER_BLOCK; //一级间接块的块号
  // 读出二级间接块
  if((addr = ip->addrs[NDIRECT + 1]) == 0)
    ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
  bp = bread(ip->dev, addr);
  a = (uint*)bp->data;
```

令此时的 bn 整除 256 得到其位于一级间接块的位置，接着在对 256 取余获得其位于直接块上的位置，随后先读出 inode 内部二级间接块的物理地址，若为 0，则调

用 balloc 对其进行分配，获取一个新的块号，然后我们调用 bread 读取该文件结点对应的 buf 缓冲块，获取该缓冲块的数据。
该缓冲块的数据就是一级间接块的地址。

```
if((addr = a[index2]) == 0) {
  a[index2] = addr = balloc(ip->dev);
  log_write(bp);// 更改了当前块的内容，标记以供后续写回磁盘
}
brelse(bp);

bp = bread(ip->dev, addr);//读取二级间接块
a = (uint*)bp->data;
if((addr = a[index1]) == 0) {
  a[index1] = addr = balloc(ip->dev);
  log_write(bp);
}
brelse(bp);
return addr;
```

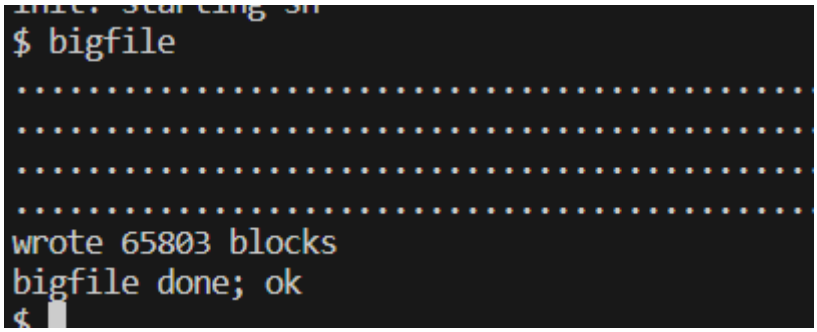
根据之前获取的 index2 偏移量来读取对应的块，如果当前块不存在，则继续申请新的内存块，不过需要注意此时申请新的内存块的话是对缓冲块 buf 中的内容进行修改的，需要调用 log_write 写入日志。结束之后调用 brelse 来释放 buf 块。
之后的直接块操作类似也是先读取 buf，接着在读取数据，没有的话就分配并将该修改写入磁盘，最后释放 buf 块。

c) 最后我们需要修改 itrunc 函数来释放对应的二级间接块。
基本思路还是类似，借鉴已经实现的二级索引于直接索引的释放。

```
struct buf* bp1;
uint* a1;
if(ip->addrs[NDIRECT + 1]) {//二级索引是否有效
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);//读取二级间接块
  a = (uint*)bp->data;
  for(i = 0; i < NADDR_PER_BLOCK; i++) {
    if(a[i]) {
      bp1 = bread(ip->dev, a[i]);
      a1 = (uint*)bp1->data;
      for(j = 0; j < NADDR_PER_BLOCK; j++) {
        if(a1[j])
          bfree(ip->dev, a1[j]);
      }
      brelse(bp1);
      bfree(ip->dev, a[i]);
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 1]);//将addrs上最后一块释放
  ip->addrs[NDIRECT + 1] = 0;
}
ip->size = 0;
iupdate(ip);
```

首先查看该二级间接块是否有效，接着读取其中的内容，也就是一级索引块的地址。对其中的 256 项遍历，对存在的块，读取其内容，并遍历释放其指向的直接块中的物理块，释放完之后在释放 buf 块于其自身，采用这种方式将二级索引块全部释放。

d) 测试如下：



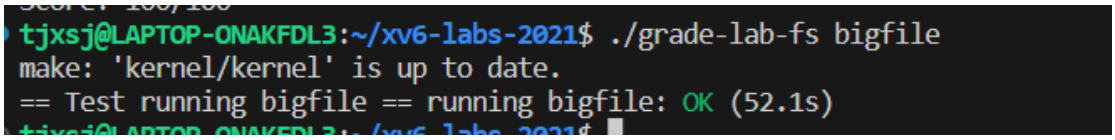
3) 实验中遇到的问题和解决方法

本实验的内容思路比较简单，但是实际实现可能有些复杂。具体写代码的时候可以参考函数中已经实现的部分来写，整体还是较为容易的。

4) 实验心得

实际上本实验还是比较简单，答题思路很清晰，就是去将一个直接块修改为间接块从而来增大 xv6 的文件大小。我通过对于课堂上的老师讲的这部分内容其实比较困惑，但是在做了该实验之后便逐渐清晰了起来。

评分如下：



2. Symbolic links (moderate)

1) 实验目的

本实验让我们去实现 xv6 文件系统的符号链接功能。

符号链接又称为软链接，是指通过路径名链接的文件；当一个符号链接被打开时，内核会跟随链接指向被引用的文件。它于硬链接类似，但是硬链接是创建了一个文件的副本，二者之间互相影响，仅限于指向同一磁盘或同一文件系统上的文件；而符号链接只是记录了文件的路径，二者之间的独立的，并且其可以跨磁盘设备。

我们在本实验中需要去实现了一个 sys_syslink(target, path)系统调用，在路径 path 处创建一个新的指向 target 的符号链接。

2) 实验步骤

a) 符号链接属于一种新的文件类型，我们应当在 stat.h 添加其相关的宏定义

```
#define T_DEVICE  3   // Device
#define T_SYMLINK 4  //表明此文件类型为符号链接
```

在提示中，向 open 传递 O_NOFOLLOW 标志时，open 应打开符号链接（而不是跟随符号链接），我们还应在 fcntl.h 增添关于 O_NOFOLLOW 的标志。当其为 1 时，表示单纯打开符号链接这个文件，而非其路径指向的文件

```
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x004
```

b) 接着是系统调用的常规操作，在 user/user.h 中添加对应的声明，在 usys.pl 文件中添加系统调用的入口。在 kernel/syscall.h 中添加系统调用号，在 kernel/syscall.c 中添加映射。同时不要忘了在 makefile 中加入测试程序。

```
int symlink(char* ,char*);
entry("symlink");
[SYS_symlink]   sys_symlink,
#define SYS_symlink  22
$U/_symlinktest\
```

c) 我们在 sysfile.c 文件中去实现 sys_symlink 系统调用

```
uint64
sys_symlink(void){
  char target[MAXPATH], path[MAXPATH];
  struct inode* ip_path;

  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {/
    return -1;
  }

  begin_op();
  // 分配一个inode结点，create返回锁定的inode
  ip_path = create(path, T_SYMLINK, 0, 0);//创建该符号链接文件的inode
  if(ip_path == 0) {
    end_op();
    return -1;
  }
  //写入target路径
  if(writei(ip_path, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip_path);//写入失败释放
    end_op();
    return -1;
  }

  iunlockput(ip_path);
  end_op();
  return 0;
}
```

大体逻辑就是先获取参数的值，接着调用 begin_op(对内存中的 log 区域进行加锁)，接着调用 create 创建该符号链接文件的 inode，向其中写入 target，之后将创建的 inode 的锁解除并释放其内存，最后调用 end_op()提交文件系统的 log，并释放对应的锁

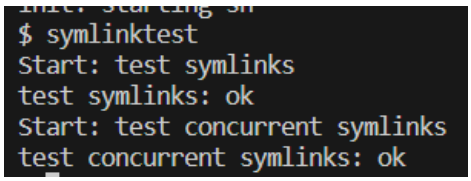
d) 最后我们需要修改 open 的系统调用，使其能够打开符号链接文件。

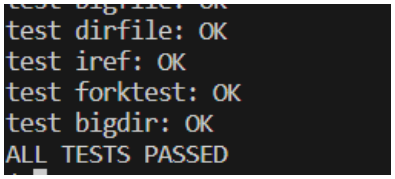
```
sysfile.c
if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  // 若符号链接指向的仍然是符号链接，则递归的跟随它
  // 直到找到真正指向的文件
  // 但深度不能超过MAX_SYMLINK_DEPTH
  for(int i = 0; i < MAXDEPTH; i++) {
    // 读出符号链接指向的路径
    if(readi(ip, 0, (uint64)path, 0, MAXPATH) != MAXPATH) {
      iunlockput(ip);
      end_op();
      return -1;
    }
    iunlockput(ip);
    ip = namei(path);//获取当前路径结点
    if(ip == 0) {//不存在释放
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type != T_SYMLINK)//一旦不是符号链接文件则退出
      break;
  }

  if(ip->type == T_SYMLINK) {// 超过最大允许深度后仍然为符号链接，则返
    iunlockput(ip);
    end_op();
    return -1;
  }
}
```

首先判断文件类型是不是符号链接同时 O_NOFOLLOW 是否为 0，若满足，则调用 readi 函数读取路径，同时调用 namei 函数获取对应 inode，如果此时 inode 不是符号链接文件则退出，否则继续递归直到到达最大递归深度 MAXDEPTH。超过最大深度后仍然为符号链接文件则返回-1。

e) 测试如下：





3) 实验中遇到的问题和解决方法

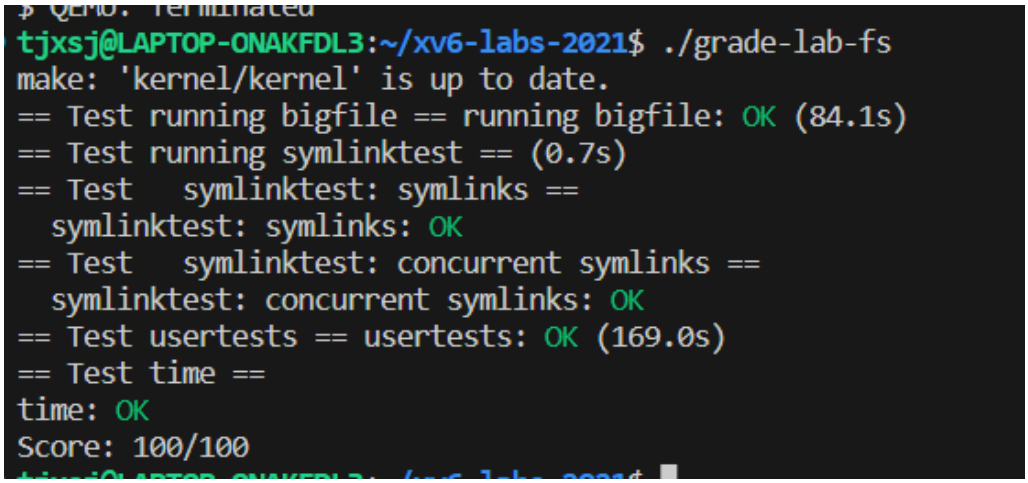
本实验难度较低，需要了解对应的一些处理函数，在阅读了相关代码之后再去编写相应的代码还是挺容易的。

4) 实验心得

完成 xv6 实验 lab9 的 symlink 实验后，我对符号链接的概念和实现有了更深入的了解，并且学到了如何在 xv6 中实现符号链接。

符号链接其实就是一个特殊类型的文件，它的内容为其他文件的路径，与硬链接创建一个文件副本不同，它就单纯提供文件的路径，这种方式使得符号链接可以不局限于一个磁盘或是文件系统中。我个人觉得它和 c++指针有些相似。

评分如下：



总结

通过完成本次实验，我对于 xv6 文件的多级索引机制以及符号链接等相关知识有了一定的了解。

多级索引机制就是将一个直接块指向一个物理块，该物理块在指向对应的物理块。这样对于二级索引可以增加 256-1 个块，三级索引则可以增加 256*256-1 个物理块。

符号链接则是一种特殊的文件类型，它其中储存的内容为另外一个文件的路径。

本实验总体评分如下：