Lab - File System 目录

1. Large files (moderate)

1.1 实验目的

- 增加 xv6 系统中文件的最大大小限制

1.2 实验步骤

1. 了解xv6系统的文件结构，理解文件大小上限的限制原因。

   在xv6系统中的`inode`中，使用了12个直接块和1个一级索引块，一级索引块指向了256个直接块，因此单个文件的最大大小为12 + 256 = 268个块。

   要想使上限增加，可以考虑移除一个直接块，使用二级索引，二级索引指向256个一级索引，每个一级索引又指向256个直接块，因此总大小可以达到：256 * 256 + 256 + 11 = 65803 个。

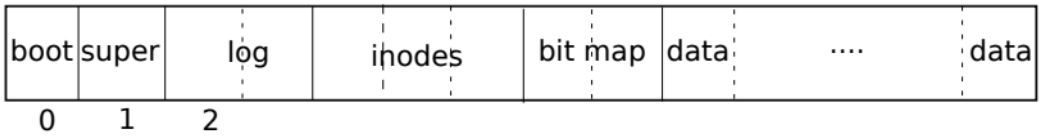

Figure 8.2: Structure of the xv6 file system.

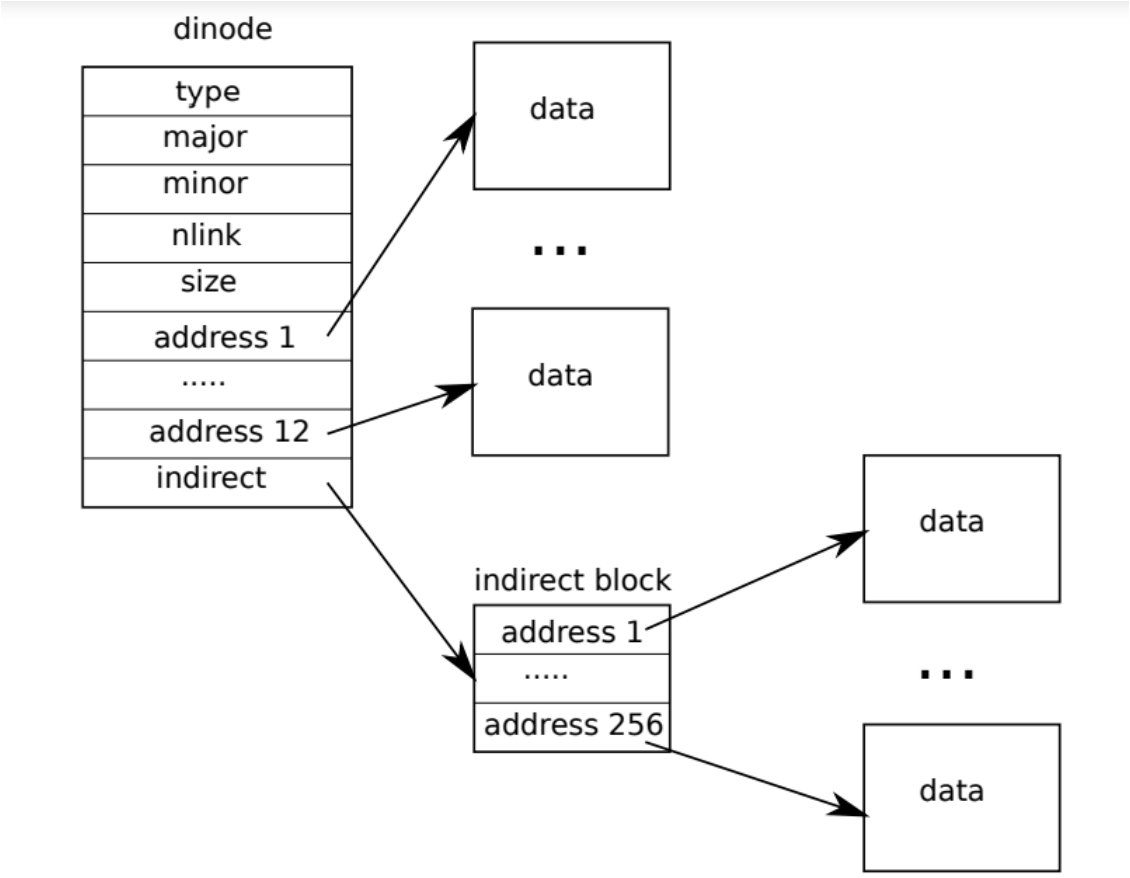

Figure 8.3: The representation of a file on disk.

2. 在`kernel/fs.h`中修改相关定义，使直接块的个数为11，要注意的是，在`dinode`中也要修改`addrs`数组，因`NDIRECT`从12改为了11，因此数据块的地址要改为`NDIRECT+2`个以保持大小一致：

```
#define NDIRECT 11 // 把个直接块改为11个直接块
```

```
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT * NINDIRECT) // 最大文件大小

// On-disk inode structure
struct dinode {
    short type;  // File type
    short major; // Major device number (T_DEVICE only)
    short minor; // Minor device number (T_DEVICE only)
    short nlink; // Number of links to inode in file system
    uint size;   // Size of file (bytes)

    uint addrs[NDIRECT + 2]; // NDIRECT从12改为了11，因此数据块的地址要改为
NDIRECT+2个以保持大小一致
};
```

3. 同理，修改`kernel/file.h`中修改`inode`定义：

```
// in-memory copy of an inode
struct inode {
    uint dev;              // Device number
    uint inum;             // Inode number
    int ref;               // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;             // inode has been read from disk?

    short type; // copy of disk inode
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT + 2]; // 因为NDIRECT改为11，所以这里也应该保持11+2个地址
};
```

4. 修改`kernel/fs.c`中的`bmap()`函数，从三个层级对块进行遍历和分配，细节见注释：

```
static uint bmap(struct inode *ip, uint bn)
{
    // 将逻辑块号映射为物理块号

    uint addr, *a;
    struct buf *bp;

    // 直接块
    if (bn < NDIRECT) {
        if ((addr = ip->addrs[bn]) == 0) {
            // 若对应的物理块号不存在，则分配一个新的物理块
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[bn] = addr;
        }
        return addr;
    }

    bn -= NDIRECT; // 复位
```

```
    // 一级索引范围内：0 ~ 256-1
    if (bn < NINDIRECT) {
        if ((addr = ip->addrs[NDIRECT]) == 0) {
            // 若索引目录不存在，则创建
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[NDIRECT] = addr;
        }

        bp = bread(ip->dev, addr); // 读取块
        a = (uint *)bp->data;
        if ((addr = a[bn]) == 0) {
            // 找到对应地址，若不存在分配磁盘块
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[bn] = addr;
            log_write(bp);
        }
        brelse(bp); // 释放块
        return addr;
    }

    bn -= NINDIRECT;
    if (bn < NINDIRECT * NINDIRECT) {

        uint iL1 = bn / NINDIRECT; // 一级索引
        uint iL2 = bn % NINDIRECT; // 二级索引

        if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
            // 若第一级索引目录不存在，则创建
            addr = balloc(ip->dev);
            if (addr == 0) {
                return 0;
            }
            ip->addrs[NDIRECT + 1] = addr;
        }
        bp = bread(ip->dev, addr); // 读取第一级索引目录
        a = (uint *)bp->data;
        if ((addr = a[iL1]) == 0) {
            // 如果对应的二级索引目录不存在，分配一个新的物理块，并将其记录在一级索引目录
中
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[iL1] = addr;
            log_write(bp);
        }
        brelse(bp); // 使用了bp来读取，此时释放

        bp = bread(ip->dev, addr); // 读取二级索引目录
        a = (uint *)bp->data;
```

```
        if ((addr = a[iL2]) == 0) {
            // 如果对应的物理块号不存在，分配一个新的物理块，并将其记录在二级索引目录中
            addr = balloc(ip->dev);
            if (addr == 0) {
                brelse(bp);
                return 0;
            }
            a[iL2] = addr;
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    panic("bmap: out of range");
}
```

5. 修改`itrunc()`函数，使得其配合三级块结构进行空间的销毁：

```
// Truncate inode (discard contents).
// Caller must hold ip->lock.
void itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp;
    uint *a;

    for (i = 0; i < NDIRECT; i++) {
        if (ip->addrs[i]) {
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    // 释放一级索引目录对应的物理块
    if (ip->addrs[NDIRECT]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
            if (a[j])
                bfree(ip->dev, a[j]);
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

    struct buf *bpL2;
    uint *b;
    int k;
    // 释放二级索引目录对应的物理块
    if (ip->addrs[NDIRECT + 1]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint *)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
            if (a[j]) {
                bpL2 = bread(ip->dev, a[j]);
```

```
                b = (uint *)bpL2->data;
                for (k = 0; k < NINDIRECT; k++) {
                    if (b[k])
                        bfree(ip->dev, b[k]);
                }
                brelse(bpL2);
                bfree(ip->dev, a[j]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }
    ip->size = 0;
    iupdate(ip);
}
```

6. 运行`bigfile`，测试大文件生成结果：

```
init: starting sh
$ bigfile
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..........................
wrote 65803 blocks
bigfile done; ok
```

7. 运行`usertests`，测试相关调用：

```
test manywrites: OK
test badwrite: OK
test execout: OK
test diskfull: balloc: out of blocks
balloc: out of blocks
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

8. 运行`grade`，结果如下：

```
== Test running bigfile == running bigfile: OK (60.9s)
```

1.3 实验中遇到的问题和解决办法

- **问题：** 触发崩溃：`panic: bmap: out of range`

  解决方法：该错误是由于越界的访问引起的。应注意到在访问完直接块后有`bn-=NDIRECT`操作用于复位，如果在第二次需要复位时忘了复位操作，则会导致这样的越界，注意减回相应大小即可。

1.4 实验心得

在完成`Large Files`实验过程中，我对xv6操作系统的文件系统机制特别是文件的映射和数据块的管理方面有了更深入的理解，通过修改`bmap()`函数实现文件的双重间接块，使我对文件的逻辑块和物理块之间的映射关系有了更清晰的认识，并学会了如何通过逻辑块号计算出对应的物理块号，从而实现文件的存储和读写。理解了文件系统的底层逻辑，明白了诸如日志、中断等机制的实现。充分利用空间，将一个`inode`中的 12 个直接块和 1 个一级间接块减少为 11 个直接块，以腾出空间来存储二级间接块。这大大启发了我，在今后工作学习中都有很多可以从中学习的地方。

2. Symbolic links (moderate)

2.1 实验目的

- 优化xv6中的缓冲区缓存（`Buffer Cache`）

2.2 实验步骤

1. 在`Makefile`中添加编译：

```
$U/_symlinktest\
```

2. 在`user/user.h`中添加函数定义，接收两个字符串：

```
int symlink(char *, char *);
```

3. 在`user/usys.pl`中添加入口：

```
entry("symlink");
```

4. 在`kernel/syscall.h`中添加系统调用号：

```
#define SYS_symlink 22
```

5. 在`kernel/syscall.c`中完成添加相应调用：

```
extern uint64 sys_symlink(void);
static uint64 (*syscalls[])(void) = {
    ...
    [SYS_symlink] sys_symlink
};
```

6. 在`kernel/stat.h`中添加宏定义，用于标识一个文件是否为`symlink`：

```
#define T_SYMLINK 4 // 软链接
```

7. 在`kernel/fcntl.h`中添加宏定义，该标志用于`open()`系统调用。传递给`open()`的标志使用按位或运算符组合：

```
#define O_NOFOLLOW 0x800 // 该标志用于open系统调用。传递给open的标志使用按位或运算符组合
```

8. 在`kernel/fs.h`中定义软链接的递归最大次数：

```
#define SYMLINK_REC_MAX 10 // 软链接的最大递归次数
```

9. 在`kernel/sysfile.c`中添加系统调用的实现`sys_symlink()`，为实现软链接，把目标路径`target`存储在一个`inode`中，而后再`open()`的过程中作处理：

```
uint64 sys_symlink(void)
{
    char target[MAXPATH];
    char path[MAXPATH];
    argstr(0, target, MAXPATH);
    argstr(1, path, MAXPATH);

    // begin_op一直保持等待直到日志系统当前未处于提交中
    // 并且直到有足够的未被占用的日志空间来保存此调用的写入
    begin_op();

    struct inode *ip = create(path, T_SYMLINK, 0, 0); // 根据给定的符号虚拟地址
创建一个inode

    if (ip == 0) {
        end_op();
        return -1;
    }

    if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
        // 将目标地址存放在inode中，以实现软链接
        iunlockput(ip);
        end_op();
        return -1;
    }

    iunlockput(ip); // 释放inode
    end_op();       // 结束日志系统
    return 0;
}
```

10. 修改`open()`系统调用，使其能够通过软链接定位到目标路径，详情见注释：

```
uint64 sys_open(void)
{

    ...

    if (!(omode & O_NOFOLLOW) && ip->type == T_SYMLINK) {
        // 软链接的判断条件为：
        // 1. 文件类型是SYMLINK
        // 2. 非O_NOFOLLOW
        // 而且最终定位到的文件不应是软链接类型，要设置递归深度，若超过该深度则返回错误
        char path[MAXPATH];
        for (int i = 0; i < SYMLINK_REC_MAX; i++) {
            // 递归深度为SYMLINK_REC_MAX（定义于fs.h）

            if (readi(ip, 0, (uint64)path, 0, MAXPATH) < MAXPATH) {
                // 读取软链接所存储的目标路径
                iunlockput(ip);
                end_op();
```

```
                return -1;
            }

            iunlockput(ip);
            // namei计算path并返回相应的inode
            if ((ip = namei(path)) == 0) {
                end_op();
                return -1;
            }
            ilock(ip);

            if (ip->type != T_SYMLINK)
                // 找到不是软链接类型的文件inode，继续正常打开
                break;
        }

        if (ip->type >= T_SYMLINK) {
            // 若超出递归深度，返回-1
            iunlockput(ip);
            end_op();
            return -1;
        }
    }

    ...

}
```

11. 运行`symlinktest`，结果如下：

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
```

12. 运行usertests，结果如下：

```
ialloc: no inodes
ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

13. 运行grade测试，结果如下：

```
== Test running symlinktest ==
$ make qemu-gdb
(0.7s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
```

2.3 实验中遇到的问题和解决方法

- **问题：** 如果出现软链接的循环使用怎么办，比如说路径A链接到了路径B，而路径B又链接到了路径A

  解决方法： 考虑设置递归的深度最大值，每一次链接的进入都会更加深入栈中，当超过规定的阈值便自动退出并返回错误，这样做避免了陷入死循环。

- **问题：** 软链接链接文件后应该怎么处理去实现定向？

  解决方法： 对于`symlink()`函数，仅仅把它当作是符号上的链接，真正的路径定向需要在`open()`调用中修改。

2.4 实验心得

本实验主要实现了xv6文件系统中的软链接功能。通过实现符号链接系统调用，我了解了如何处理路径引用软链接的情况、如何跟随多个软链接直到到达非链接文件、如何修改系统调用和文件系统数据结构以支持新的功能，实验使我在inode和数据块的管理方面有了更深入的理解。实验中对于可能出现的循环引用，考虑设置递归的深度最大值，每一次链接的进入都会更加深入栈中，当超过规定的阈值便自动退出并返回错误，这样做避免了陷入死循环。并通过合理的函数功能分布较好实现了软链接的功能，这次实验让我受益良多。