

# Lab6 实验报告

## 一、实验内容

### 1. 任务一：大文件（Large files）

#### 1.1 实验目标

修改 `bmap()`，使其除了实现 direct 块和 singly-indirect 块外，还能实现 doubly-indirect 块。

#### 1.2 实验步骤

##### 1. 修改相关宏与Inode结构体

在 `fs.h` 中修改：

把一个直接块作为二级间址块，故 `NDIRECT` 减为11

增添表示二级间址块数的宏 `NDOUBLEINDIRECT`

修改文件最大大小的宏

```
1 #define NDIRECT 11
2 #define NINDIRECT (BSIZE / sizeof(uint))
3 #define NDOUBLEINDIRECT (NINDIRECT * NINDIRECT)
4 #define MAXFILE (NDIRECT + NINDIRECT + NDOUBLEINDIRECT)
```

同时修改 `file.h/inode` 与 `fs.h/dinode` 结构体中的 `addrs` 数组：

```
1 struct inode {
2     // ...
3     uint addrs[NDIRECT+1+1];
4 };
```

```
1 struct dinode {
2     // ...
3     uint addrs[NDIRECT+1+1];    // Data block addresses
4 };
```

## 2. 修改 fs.c/bmap()

`bmap` 根据inode指针与逻辑块号（`addrs`数组中的偏移量），返回所指向的**磁盘块号**。函数中使用了`uint`类型的`addr`，它所指的就是磁盘块的块号。

增加二级索引的方式很简单，`bmap` 已经给出了一级间址的架构，仿照着索引两次即可。

```
1 // 二级间址
2 if(bn < NDOUBLEINDIRECT){
3     // Load double-indirect block, allocating if necessary.
4
5     // 一级索引
6     if((addr = ip->addrs[NDIRECT + 1]) == 0)
7         ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
8     bp = bread(ip->dev, addr);
9     a = (uint*)bp->data;
10    if ((addr = a[bn / NINDIRECT]) == 0){
11        a[bn / NINDIRECT] = addr = balloc(ip->dev);
12        log_write(bp);
13    }
14    brelse(bp);
15
16    // 二级索引
17    bp = bread(ip->dev, addr);
18    a = (uint*)bp->data;
19    if((addr = a[bn % NINDIRECT]) == 0){
20        a[bn % NINDIRECT] = addr = balloc(ip->dev);
21        log_write(bp);
22    }
23    brelse(bp);
24    return addr;
25 }
```

## 3. 修改 fs.c/itrunc()

该函数用于释放一个inode所指向所有数据块，因此对于二级间址块也要释放掉。

同样，基于一级间址的块的释放逻辑基础上，可以推出二级间址块释放的代码，只需两重循环，先释放二级块，再释放一级块。

```
1 // 释放二级间址块
2 if(ip->addrs[NDIRECT + 1]){
3     bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
4     a = (uint*)bp->data;
```

```

5      for (i = 0; i < NINDIRECT; ++i){
6          if(a[i]){
7              double_bp = bread(ip->dev, a[i]);
8              b = (uint*)double_bp->data;
9              for (j = 0; j < NINDIRECT; ++j){
10                 if (b[j])
11                     bfree(ip->dev, b[j]);
12             }
13             brelse(double_bp);
14             bfree(ip->dev, a[i]);
15         }
16     }
17     brelse(bp);
18     bfree(ip->dev, ip->addrs[NDIRECT + 1]);
19     ip->addrs[NDIRECT + 1] = 0;
20 }
21

```

### 1.3 实验结果

bigfile

```
$ bigfile
.....
.....
.....
.....
.....
.....
.....
wrote 65803 blocks
bigfile done; ok
```

usertests

```
test exitinput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

## 2. 任务二：符号链接 (Symbolic links)

## 2.1 实验目标

实现符号链接（软链接）的系统调用 `symlink(char *target, char *path)`，该调用在引用由 `target` 命名的文件的路径处创建一个新的符号链接。

在 `path` 路径下添加一个“symlink”类型的文件，文件内容为 `target`（`target` 是目标文件的路径）。

## 2.2 实验步骤

### 1. 添加系统调用相关声明

```
1 // usys.pl
2 entry("symlink");
```

```
1 // user.h
2 int symlink(char* target, char* path);
3
```

```
1 // syscall.h
2 #define SYS_symlink 22
3
```

```
1 // syscall.c
2 extern uint64 sys_symlink(void);
3
4 [SYS_symlink] sys_symlink,
5
```

在 Makefile 的 `UPROGS` 添加：

```
1 # Makefile
2 $U/_symlinktest\
3
```

### 2. 添加相关宏

在 `stat.h` 中

```
1 #define T_DIR      1    // Directory
2 #define T_FILE     2    // File
3 #define T_DEVICE   3    // Device
4 #define T_SYMLINK  4    // symbolic link
```

在 `fcntl.h` 中

```
1 #define O_RDONLY  0x000
2 #define O_WRONLY  0x001
3 #define O_RDWR    0x002
4 #define O_CREATE  0x200
5 #define O_TRUNC    0x400
6 #define O_NOFOLLOW 0x004 //不与其它宏冲突即可
```

### 3. 实现系统调用 `sysfile.c/sys_symlink`

该函数流程如下：

- 调用 `create`，在path上**创建**一个新的inode，文件类型为T\_SYMLINK
- 调用 `writei`，将target写入inode的数据块中

```
1 uint64 sys_symlink(void) {
2     char target[MAXPATH];
3     char path[MAXPATH];
4     struct inode *ip;
5
6     if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
7         return -1;
8
9     begin_op();
10
11     if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
12         end_op();
13         return -1;
14     }
15
16     if(writei(ip, 0, (uint64)target, 0, MAXPATH) != MAXPATH)
17         panic("sys_symlink: writei");
18
19     iunlockput(ip);
20     end_op();
21 }
```

```

21
22     return 0;
23 }
24

```

#### 4. 修改函数 `sysfile.c/sys_open`

修改`sys_open`调用，以便能够打开软连接类型的文件

- 由于存在软连接指向软连接的情况，因此要不断循环检测打开的文件是否是软连接，直到打开的文件不是软连接为止。
- 如果链接形成循环，则必须返回错误代码。可以通过以下方式估算存在循环：通过在链接深度达到某个阈值（例如10）时返回错误代码。根据hint，通过维护一个`depth`变量，记录软连接迭代的深度，超过`MAXFOLLOWDEPTH`（10）则返回错误。
- 当含有`O_NOFOLLOW`标志位时，不进行软连接索引，而是直接返回软连接原文
- 主要的函数有两个，`readi()` 用于读取inode中的软连接，`namei()` 用于获取path上文件的inode

```

1  ...
2  else {
3      if((ip = namei(path)) == 0){
4          end_op();
5          return -1;
6      }
7      ilock(ip);
8
9      //修改sys_open调用，以便能够打开软连接类型的文件
10     while(ip->type==T_SYMLINK && !(omode&O_NOFOLLOW) && depth<MAXFOLLOWDEPTH) {
11         readi(ip, 0, (uint64)path, 0, MAXPATH);
12         iunlockput(ip);
13         if((ip = namei(path)) == 0) {
14             end_op();
15             return -1;
16         }
17         ilock(ip);
18         ++depth;
19     }
20
21     if(depth == MAXFOLLOWDEPTH) {
22         iunlock(ip);
23         end_op();
24         return -1;
25     }
26

```

```
27     if(ip->type == T_DIR && omode != O_RDONLY){
28         iunlockput(ip);
29         end_op();
30         return -1;
31     }
32 }
33 ...
```

## 2.3 实验结果

symlinktest

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

usertests

```
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

## 二、测试结果

make grade

```
== Test running bigfile ==  
$ make qemu-gdb  
running bigfile: OK (89.2s)  
== Test running symlinktest ==  
$ make qemu-gdb  
(1.4s)  
== Test  symlinktest: symlinks ==  
symlinktest: symlinks: OK  
== Test  symlinktest: concurrent symlinks ==  
symlinktest: concurrent symlinks: OK  
== Test usertests ==  
$ make qemu-gdb  
usertests: OK (177.6s)  
== Test time ==  
time: OK  
Score: 100/100
```