

# 一、实验内容

## 1.1. 大文件

### 1.1.1 实验步骤

a. 首先修改fs.h中的相关定义，将直接指向数据块的指针改为11个，多出一个用于记录二级间接指针。用NDOUBLEINDIRECT表示二级间接指针指向的块中记录的一级间接指针的数量，NSINGLEINDIRECT表示每个一级间接指针指向的块中记录的指向数据块的指针数量。此时MAXFILE需要加上二级间接指针所增加的数据块。

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLEINDIRECT (BSIZE / sizeof(uint))
#define NSINGLEINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + NSINGLEINDIRECT * NDOUBLEINDIRECT)
```

b. 修改dinode和inode结构体的定义。

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

struct inode {
    .....
    uint addrs[NDIRECT+2];
};
```

c. 修改bmap函数实现，加入二级间接指针的部分。首先减去直接指针和间接指针的数据块数量，得到需寻找的块在二级间接指针所记录的块中的序号，通过该序号整除NSINGLEINDIRECT得到对应的一级间接指针，读取该指针指向的块。而后再根据序号模NSINGLEINDIRECT得到对应的指向数据块的指针索引，并返回该地址即可(过程中若碰到对应的指针为空的情况则需要调用balloc分配块)。

```
bn -= NINDIRECT;

if(bn < NDOUBLEINDIRECT * NSINGLEINDIRECT){
```

```

if((addr = ip->addrs[NDIRECT+1]) == 0){
    addr = balloc(ip->dev);
    if(addr == 0)
        return 0;
    ip->addrs[NDIRECT+1] = addr;
}
bp = bread(ip->dev, addr);
a = (uint*)bp->data;
int idx0 = bn / NSINGLEINDIRECT;
if((addr = a[idx0]) == 0){
    addr = balloc(ip->dev);
    if(addr){
        a[idx0] = addr;
        log_write(bp);
    }
}
brelse(bp);

bp = bread(ip->dev, addr);
a = (uint*)bp->data;
int idx1 = bn % NSINGLEINDIRECT;
if((addr = a[idx1]) == 0){
    addr = balloc(ip->dev);
    if(addr){
        a[idx1] = addr;
        log_write(bp);
    }
}
brelse(bp);
return addr;
}

```

d. 修改itrunc函数实现，添加释放二级间接指针存储的数据块功能。首先判断二级间接指针是否为空，若是则无需处理。否则访问二级间接指针指向的块，遍历其中的一级间接指针，若不为空则访问其指向的块并遍历其中指向数据块的指针。若指向数据块的指针不为空则调用bfree释放对应的数据块，同时在每一级遍历结束后也要释放对应的块。

```

if(ip->addrs[NDIRECT+1]){
    bp0 = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a0 = (uint*)bp0->data;
    for(j = 0; j < NDOUBLEINDIRECT; j++){
        if(a0[j]){
            bp1 = bread(ip->dev, a0[j]);
            a1 = (uint*)bp1->data;
            for(k = 0; k < NSINGLEINDIRECT; k++){
                if(a1[k])
                    bfree(ip->dev, a1[k]);
            }
            brelse(bp1);
            bfree(ip->dev, a0[j]);
            a1[k] = 0;
        }
    }
}

```

```
brelse(bp0);  
bfree(ip->dev, ip->addrs[NDIRECT+1]);  
ip->addrs[NDIRECT+1] = 0;  
}
```

## 1.1.2 实验测试结果

- bigfile

```
$ bigfile
```

```
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

```
wrote 65803 blocks
```

```
bigfile done; 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
```

- usertests

## 1.2 符号链接

### 1.2.1 实验步骤

- a. 在user/usys.pl、user/user.h以及syscall.c中添加对应的条目。

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

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

```
extern uint64 sys_symlink(void);

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

b. 添加T\_SYMLINK文件类型，添加O\_NOFOLLOW表示打开符号链接时是否要直接返回符号链接本身而不是其指向的文件。

```
#define T_SYMLINK 4
```

```
#define O_NOFOLLOW 0x800
```

c. 实现sys\_symlink系统调用。首先根据路径创建一个T\_SYMLINK类型的文件，而后将目标路径写入该文件中。参考sys\_link系统调用的实现，在关于文件系统的开始需要调用begin\_op，在结束时需要调用end\_op，以支持文件系统的日志机制。同时由于create时已经获得了对应的inode的锁，所以在结束时还需要调用iunlockput以释放锁并减少对内存中的inode的引用数量。

```
int
sys_symlink(void){
    char target[MAXPATH], path[MAXPATH];
    int target_size;
    struct inode* ip;
    if((target_size = argstr(0, target, MAXPATH)) < 0)
        return -1;
    if(argstr(1, path, MAXPATH) < 0)
        return -1;

    begin_op();
    if((ip = create(path, T_SYMLINK, 0, 0)) == 0){
        end_op();
        return -1;
    }

    if(writei(ip, 0, (uint64)target, 0, target_size) < target_size){
        iunlockput(ip);
        end_op();
        return -1;
    }
}
```

```

}

iunlockput(ip);
end_op();
return 0;
}

```

- 修改sys\_open系统调用以支持路径指向符号链接的情况。当inode类型为T\_SYMLINK，且文件打开模式中O\_NOFOLLOW为假时，便需要寻找符号链接指向的真实文件。首先读取inode指向的文件内容获得其指向的目标，而后读取该目标对应的inode并判断其类型。若不为T\_SYMLINK，则说明已经查找到最终目标，否则继续进行迭代查找。当查找次数超过10次则认为符号链接产生了循环，返回错误代码。

```

if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
int find = 0;
for(int i = 0; i < 10; i++){
    char target[MAXPATH];
    if(readi(ip, 0, (uint64)target, 0, MAXPATH) <= 0){
        iunlockput(ip);
        end_op();
        return -1;
    }
    iunlockput(ip);

    if((ip=namei(target)) == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    if(ip->type != T_SYMLINK){
        find = 1;
        break;
    }
}
if(!find){
    end_op();
    return -1;
}
}

```

## 1.2.2 实验测试结果

- symlinktest

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

- usertests

```
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

---

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (81.4s)
== Test running symlinktest ==
$ make qemu-gdb
(0.5s)
== Test   symlinktest: symlinks ==
    symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
    symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (172.0s)
== Test time ==
time: OK
Score: 100/100
```