**Lab 6**

**Part A:** **Large files**

首先在fs.h中按照题目要求修改相关宏与结构体定义，将direct inode数量更改为11并加入一个doubly-indirect inode。同时，添加宏定义NININDIRECT:

#define **NININDIRECT** (**BSIZE** / sizeof(**uint**)) \* (**BSIZE** / sizeof(**uint**))

其表示的是doubly-indirect inode所占据的数据块总数，值为NINDIRECT的平方。之后，在fs.c中修改bmap映射部分内容，添加二级block映射。模仿一级block，先对块序号减去NINDIRECT，随后判断bn是否超出二级block数，若超出则陷入panic。定义变量index、offset，分别对应bn所表示的数据块对应的一级block序号以及在该block下的偏移量。随后访问相应block中数据，首先通过NINDIRECT+1为下标访问inode中的数据内容，随后通过index访问一级block内容，并最终以offset得到数据块对应的地址（若为零则为其新分配一块block地址空间）。修改部分相关代码如下。

bn -= **NINDIRECT**;

if(bn < **NININDIRECT**){

int index = bn / **NINDIRECT**;

int offset = bn % **NINDIRECT**;

if((addr = ip->addrs[**NDIRECT** + 1]) == 0)

ip->addrs[**NDIRECT** + 1] = addr = **balloc**(ip->dev);

bp = **bread**(ip->dev, addr);

a = (**uint**\*)bp->data;

if((addr = a[index]) == 0){

a[index] = addr = **balloc**(ip->dev);

**log\_write**(bp);

}

**brelse**(bp);

bp = **bread**(ip->dev, addr);

a = (**uint**\*)bp->data;

if((addr = a[offset]) == 0){

a[offset] = addr = **balloc**(ip->dev);

**log\_write**(bp);

}

**brelse**(bp);

return addr;

}

随后在itrunc函数中为doubly-indirect inode添加回收部分内容。首先与先前类似，以NINDIRECT+1为下标访问doubly-direct inode，之后通过一个二层循环遍历该inode下对应的所有block，若地址非零则释放先前分配的地址空间。代码如下。

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]){

bp\_next = **bread**(ip->dev, a[j]);

a\_sec = (**uint**\*)bp\_next->data;

for(k = 0; k< **NINDIRECT**; k++){

if(a\_sec[k]){

**bfree**(ip->dev, a\_sec[k]);

}

}

**brelse**(bp\_next);

**bfree**(ip->dev, a[j]);

}

}

**brelse**(bp);

**bfree**(ip->dev, ip->addrs[**NDIRECT** + 1]);

ip->addrs[**NDIRECT** + 1] = 0;

}

运行结果如下：



**Part B:** **Symbolic links**

首先添加系统调用symlink相关定义（user/usys.pl中添加entry，user/user.h中添加定义，kernel/syscall.h中添加宏定义，kernel/syscall.c中添加掩码及句柄定义），并根据提示在fcntl.c中添加宏定义O\_NOFOLLOW。之后，在sysfile.c中添加sys\_symlink系统调用函数。首先在create函数中添加T\_SYMLINK类型的创建方式（直接返回结构体ip即可），之后在sys\_symlink中可直接调用create创建象征链接inode，之后使用writei函数向ip中写入被链接的路径即可。

**uint64**

**sys\_symlink**(void)

{

char target[**MAXPATH**], path[**MAXPATH**];

struct **inode** \*ip;

if(**argstr**(0, target, **MAXPATH**) < 0 || **argstr**(1, path, **MAXPATH**) < 0) return -1;

**begin\_op**();

ip = **create**(path, **T\_SYMLINK**, 0, 0);

if(ip == 0){

**end\_op**();

return -1;

}

if(**writei**(ip, 0, (**uint64**)target, 0, **MAXPATH**) != **MAXPATH**) return -1;

**iunlockput**(ip);

**end\_op**();

return 0;

}

随后修改sys\_open部分内容，在其中添加象征链接的访问方式，进入象征链接访问的条件为当前模式并非O\_NOFOLLOW并且ip的类型为SYMLINK。定义变量depth表示symlink深度，target表示当前象征链接下的目标路径，并通过namei函数让ip指向下一层，即当前inode的象征链接目标路径对应inode，以此对ip循环更新，并注意查看ip类型：若某次循环时ip类型不再是symlink，说明访问到真实数据块，因此可以跳出循环。每次循环对depth进行判断，若深度大于十则直接返回-1表示访问失败。

if(!(omode & **O\_NOFOLLOW**) && ip->type == **T\_SYMLINK**){

int depth = 0;

char target[**MAXPATH**];

while(ip->type == **T\_SYMLINK**){

if(depth == 10){

**iunlockput**(ip);

**end\_op**();

return -1;

}

depth++;

**memset**(target, 0, sizeof(target));

**readi**(ip, 0, (**uint64**)target, 0, **MAXPATH**);

**iunlockput**(ip);

if((ip = **namei**(target)) == 0){

**end\_op**();

return -1;

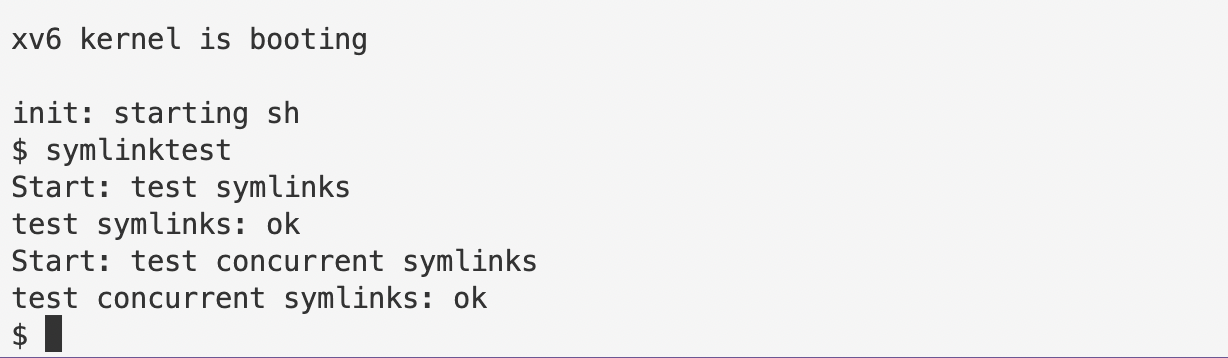
}

**ilock**(ip);

}

}

运行结果如下：



**Thoughts**

本次实验实现了二级indirect inode以及文件的象征性链接，加深了对xv6文件系统的理解。