

MIT_Lab9 File System

20307130350 陈丹纯 信息安全 2022/12/19

Large files

实验目的

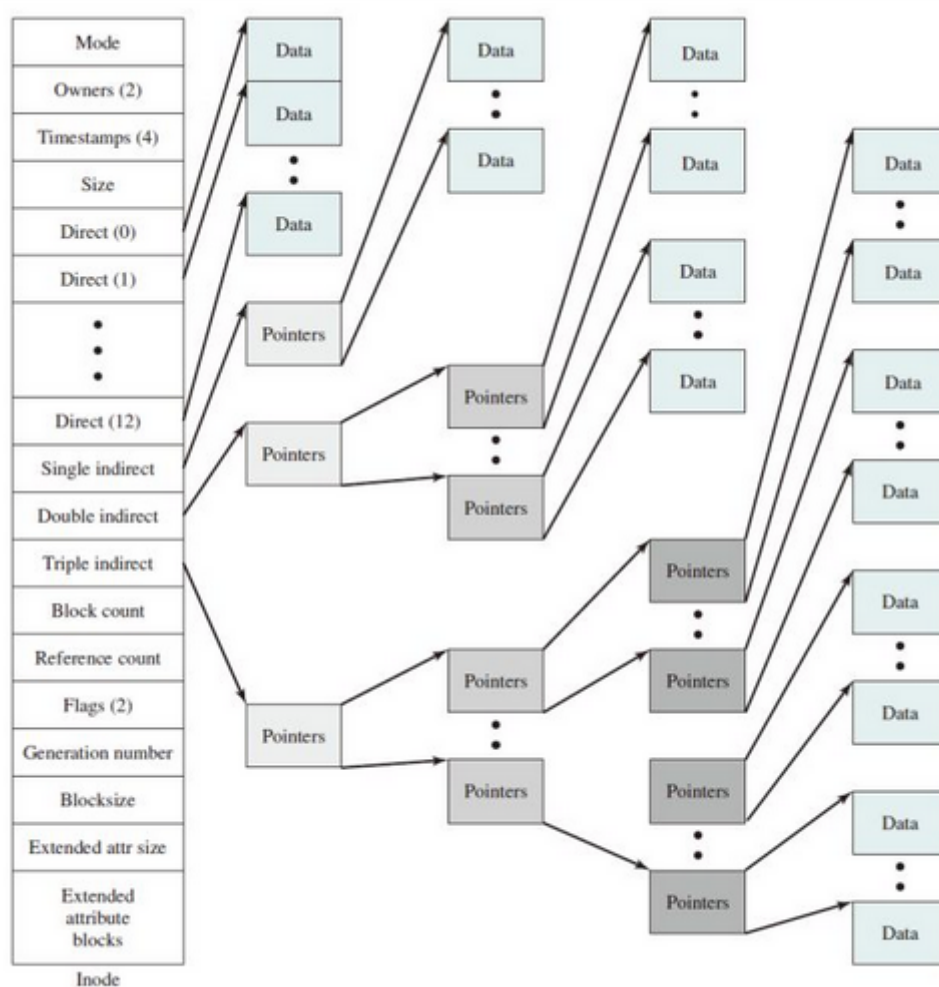
实现二级间接索引的inode结构。

实验思路

1. 用原来的一个直接索引替换为二级间接索引。

即 `ip->addrs[]` 的前11个元素是direct block，第12个元素是一个singly-indirect block，第13个元素所一个doubly-indirect block。

2. 修改 `bmap()` 和 `itrunc()` 来适配doubly-indirect block



实验过程

1. 修改 kernel/fs.h 中的直接块数目的宏定义 `NDIRECT` 为11，并且增加一个doubly-indirect block 数量的宏定义，修改最大的block数。

```
!6
!7 #define NDIRECT 11
!8 #define NINDIRECT (BSIZE / sizeof(uint)) //一级indirect所指向的data块数量 — 1k/4 (每个blocknumber是
uint4字节, 每个blocksize=1)
!9 #define NDINDIRECT (NDIRECT * NINDIRECT)
!0 #define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
!1 #define NSYMLINK 10
```

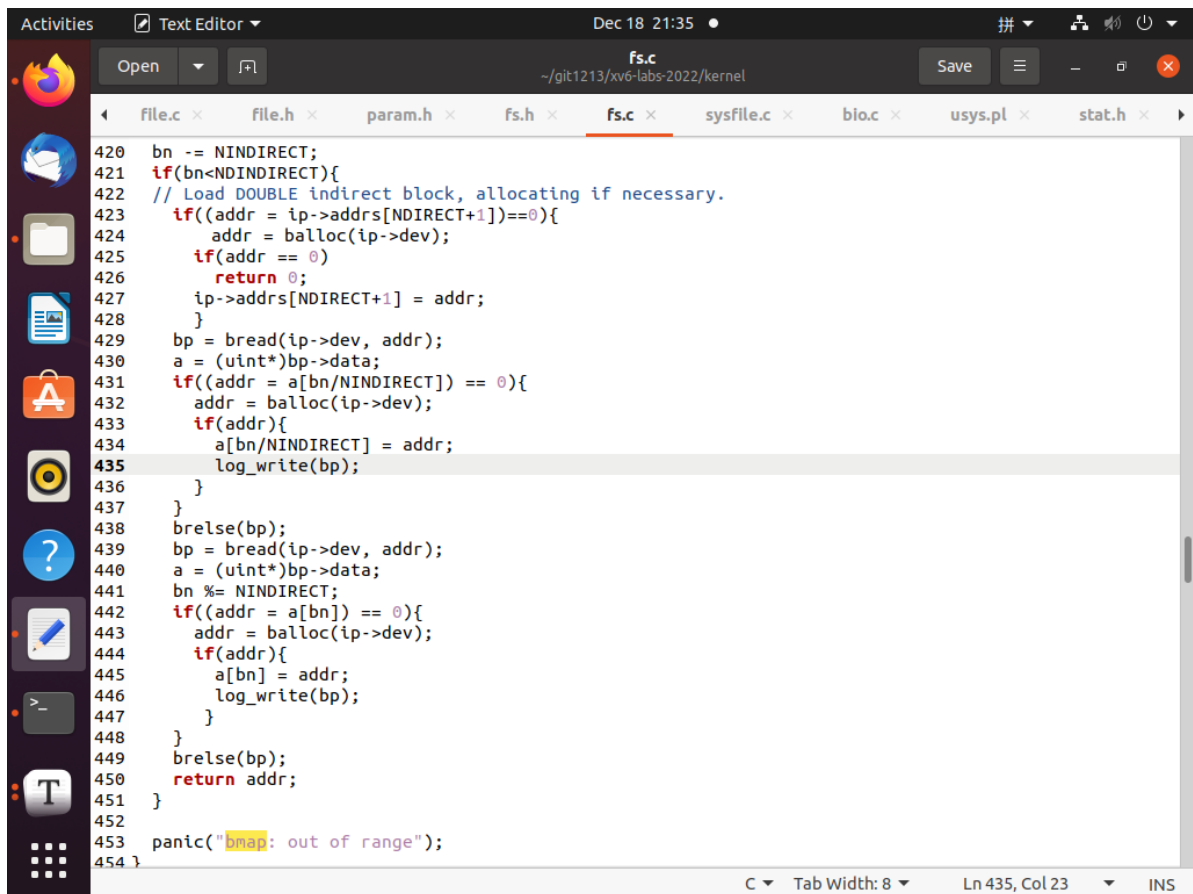
2. 修改inode相关结构体的块号数组。kernel/fs.h 中的磁盘inode结构体 `dinode` 的 `addrs` 字段；
kernel/file.h 中的内存结构体 `inode` 的 `addrs` 字段。分别改成 `NDIRECT+2`, 保证inode总块数不变。

```
33 // On-disk inode structure
34 struct dinode {
35     short type;           // File type
36     short major;          // Major device number (T_DEVICE only)
37     short minor;          // Minor device number (T_DEVICE only)
38     short nlink;          // Number of links to inode in file system
39     uint size;            // Size of file (bytes)
40     uint addrs[NDIRECT+2]; // Data block addresses
41 };
42
43
44 // in-memory copy of an inode
45 struct inode {
46     uint dev;             // Device number
47     uint inum;            // Inode number
48     int ref;              // Reference count
49     struct sleeplock lock; // protects everything below here
50     int valid;            // inode has been read from disk?
51
52     short type;           // copy of disk inode
53     short major;
54     short minor;
55     short nlink;
56     uint size;
57     uint addrs[NDIRECT+2];
58 };
```

3. 修改 kernel/fs.c 中的 `bmap()`。

该函数用于返回inode的相对块号对应的磁盘中的块号。

只需要对inode的第13个addr即二级间接索引处理即可，仿照一级间接索引块的方法，只需要索引两次。



```
420 bn -= NINDIRECT;
421 if(bn<NDINDIRECT){
422     // Load DOUBLE indirect block, allocating if necessary.
423     if((addr = ip->addrs[NDIRECT+1])!=0){
424         addr = balloc(ip->dev);
425         if(addr == 0)
426             return 0;
427         ip->addrs[NDIRECT+1] = addr;
428     }
429     bp = bread(ip->dev, addr);
430     a = (uint*)bp->data;
431     if((addr = a[bn/NINDIRECT]) == 0){
432         addr = balloc(ip->dev);
433         if(addr){
434             a[bn/NINDIRECT] = addr;
435             log_write(bp);
436         }
437     }
438     brelse(bp);
439     bp = bread(ip->dev, addr);
440     a = (uint*)bp->data;
441     bn %= NINDIRECT;
442     if((addr = a[bn]) == 0){
443         addr = balloc(ip->dev);
444         if(addr){
445             a[bn] = addr;
446             log_write(bp);
447         }
448     }
449     brelse(bp);
450     return addr;
451 }
452
453 panic("bmap: out of range");
454 }
```

4. 修改 kernel/fs.c 的 itrunc()

该函数用于释放inode的数据块。

需要对二级间接块释放，仿照一级间接块即可，需要两重循环去遍历二级间接块及其指向的一级间接块。


```
utegan@ubuntu: ~/git1213/xv6-labs-2022
sepc=0x00000000000002492 stval=0x0000000000000000
OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6456
sepc=0x00000000000005c62 stval=0x00000000000005c62
usertrap(): unexpected scause 0x000000000000000c pid=6457
sepc=0x00000000000005c62 stval=0x00000000000005c62
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
usertests slow tests starting
test bigdir: OK
test manywrites: OK
test badwrite: OK
test execout: OK
test diskfull: balloc: out of blocks
ialloc: no inodes
ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
$
```

Symbolic Links

实验目的

1. 创建符号链接（软链接）的系统调用symlink。
2. 为 `sys_open()` 提供软链接的打开支持（通过软链接打开文件）

symlink vs link

符号链接是用专门的特殊文件类型“符号链接文件”来实现的，文件中只有一个表明链接对象路径的字符串。一旦建立了符号链接，删除操作删除的所符号链接文件，其他所有操作都访问符号链接引用的文件。

硬链接link()的代码如下：

link接受两个参数，把待链接的文件路径放在old里，把新的链接路径放在new里。`link()` 使用 `namei` 查询old的inode，返回指针ip，增加ip的nlink数；使用 `nameiparent()` 查询new的父目录的inode，然后使用 `dirlink()` 把new加入它的父目录中。

```
uint64
sys_link(void)
{
    char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
    struct inode *dp, *ip;

    if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
        return -1;

    begin_op();
    if((ip = namei(old)) == 0){
        end_op();
```

```

        return -1;
    }

    ilock(ip);
    if(ip->type == T_DIR){
        iunlockput(ip);
        end_op();
        return -1;
    }

    ip->nlink++;
    iupdate(ip);
    iunlock(ip);

    if((dp = nameiparent(new, name)) == 0)
        goto bad;
    ilock(dp);
    if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
        iunlockput(dp);
        goto bad;
    }
    iunlockput(dp);
    iput(ip);

    end_op();

    return 0;

bad:
    ilock(ip);
    ip->nlink--;
    iupdate(ip);
    iunlockput(ip);
    end_op();
    return -1;
}

```

实验思路

1. 添加符号链接（软链接）的系统调用symlink。
2. 修改 `open()` 系统调用处理符号链接的情况，且符号链接的目标文件仍为符号链接文件时需要递归查找目标文件。
3. 以 `O_NOFOLLOW` 打开符号链接不会跟踪到链接的文件。
4. 其他系统调用不会跟踪符号链接，之后处理符号链接文件本身。

实验过程

1. 在 `kernel/syscall.h`, `kernel/syscall.c`, `user/usys.pl`, `user/user.h` 中添加系统调用 `symlink` 的定义声明。

```
16 [SYS_unlink] sys_unlink,
17 [SYS_link] sys_link,
18 [SYS_mkdir] sys_mkdir,
19 [SYS_close] sys_close,
20 [SYS_symlink] sys_symlink, //lab fs -2
21 };

extern uint64 sys_unlink(void);
2 extern uint64 sys_mkdir(void);
3 extern uint64 sys_close(void);
4 extern uint64 sys_symlink(void); //lab fs-2

37 entry("sleep");
38 entry("uptime");
39 entry("symlink");

22 char* sbrk(int);
23 int sleep(int);
24 int uptime(void);
25 int symlink(char *target, char *path);
26
-- .. --
```

2. 在 `kernel/stat.h` 中添加新的文件类型 `T_SYMLINK`.

```
3 #define T_DEVICE 3 // Device
4 #define T_SYMLINK 4 // symbolic link
5
```

3. 在 `kernel/fcntl.h` 中添加文件标志位为 `O_NOFOLLOW`, 注意 `open()` 中是要与文件标志位进行or操作, 所以不能与现有的flags重复。

```
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 0x800 //add for lab9-2
```

4. 在 `kernel/sysfile.c` 中实现 `sys_symlink()` 函数。

该函数用来生成符号链接。符号链接相当于一个特殊的独立的文件。其中存储的数据即目标文件的路径。

因此在该函数中, 首先通过 `create()` 创建符号链接路径对应的inode结构, 注意使用 `T_SYMLINK` 来标志符号链接文件类型。然后通过 `writei()` 将链接的目标文件路径写入inode中。

```

14 uint64
15 sys_symlink(void)
16 {
17     char target[MAXPATH], path[MAXPATH];
18     struct inode *ip, *dp;
19     if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0){
20         return -1;
21     }
22     begin_op();
23     if((ip = namei(target)) != 0){
24         if(ip->type == T_DIR){
25             end_op();
26             return -1;
27         }
28     }
29     if((dp = create(path, T_SYMLINK, 0, 0)) == 0){
30         end_op();
31         return -1;
32     }
33     //write the path of the file to the inode
34     if(writei(dp, 0, (uint64)target, 0, MAXPATH) != MAXPATH){
35         panic("symlink: writei");
36     }
37
38     iunlockput(dp);
39     end_op();
40     return 0;
41 }

```

5. 修改 `kernel/sysfile.h` 中的 `sys_open()`。

打开文件时，如果遇到符号链接，直接打开对应的文件。避免符号链接彼此之间互相链接导致死循环，设置访问深度10。每次先读取对应的inode，然后根据其中的文件名称找到对应的inode，然后继续判断该inode是否为符号链接。


```

6     }
7     if(!(omode & O_NOFOLLOW)){
8         int i=0;
9         //深度为10
10        for(; i<10 && ip->type == T_SYMLINK; i++){
11            //读取对应的inode
12            if(readi(ip,0,(uint64)path,0,MAXPATH)==0){
13                iunlockput(ip);
14                end_op();
15                return -1;
16            }
17            //根据文件路径找到对应的inode
18            if((dp = namei(path)) ==0 ){
19                iunlockput(ip);
20                end_op();
21                return -1;
22            }
23            iunlockput(ip);
24            ip = dp;
25            ilock(ip);
26        }
27        if(i==10){
28            iunlockput(ip);
29            end_op();
30            return -1;
31        }
32    }
33 }
34

```

实验结果

```

test diskfull: balloc: out of blocks
ialloc: no inodes
$ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok

```

```
utegan@ubuntu: ~/git1213/xv6-l
sepc=0x00000000000002492 stval=0x0000000000000000
OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x0000000000000000
sepc=0x00000000000005c62 stval=0x0000000000000000
usertrap(): unexpected scause 0x0000000000000000c
sepc=0x00000000000005c62 stval=0x0000000000000000
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
usertests slow tests starting
test bigdir: OK
test manywrites: OK
test badwrite: OK
test execout: OK
test diskfull: balloc: out of blocks
ialloc: no inodes
ialloc: no inodes
OK
test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

实验反思

这次lab不算难，但前期花费了大量时间阅读源码和xv6 book来理解file system。