实验报告

Lab 6

file system

姓名：张明瑞

班级：2020 级信息安全

学号：20307130247

一、 实验步骤及结果

1.Large files

Part I 实验步骤

(1)在准备阶段中检查 kernel/param.h 中，FSSIZE 的值在该实验环境下是否为 200000：

```
#define FSSIZE       200000  // size of file system in blocks
```

(2)在 kernel/fs.h 中修改部分参数的值：

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


// 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];   // Data block addresses
};
```

① 将 NDIRECT 的值从 12 修改为 11，将一个直接索引更改为二级间接索引；

② 新增变量 NDOUBLE_INDIRECT 为二级间接索引可增加的 Block 引用数量；

③ 将 MAXFILE 的值加上 NDOUBLE_INDIRECT；

④ 将结构 dinode 的成员数组 addrs[]内部的大小由 NDIRECT+1 更改为 NDIRECT+2，以保持 inode 大小不变。

(3)与(2)同理，在 kernel/file.h 中修改结构 inode 的成员 addrs[]的大小，将其大小加 1，以留待二级间接索引使用：

```
// 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];
};
```

(4)在 kernel/fs.c 中修改 bmap()函数和 itrunc()函数：

①在 bmap()函数中添加如下代码段：

```
bn -= NINDIRECT;

if(bn < NDOUBLE_INDIRECT){
  uint bn_l1 = bn / NINDIRECT;
  uint bn_l2 = 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[bn_l1]) == 0){
    a[bn_l1] = addr = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp2 = bread(ip->dev, addr);
  a2 = (uint*)bp2->data;
  if((addr = a2[bn_l2]) == 0){
    a2[bn_l2] = addr = balloc(ip->dev);
    log_write(bp2);
  }
  brelse(bp2);

  return addr;
}
```

在 fs.h 的宏定义内容中可以发现一个一级间接索引可以指示 256 个地址，而添加二级间接索引正是为了借一级间接索引的内容再度指向索引，以增大文件存储大小上限。

在 bmap()的前两个 if 条件判断语句通过 bn 与 NDIRECT 和 NIDIRECT 的大小来判断 bn 的索引类型，于是模仿前面 if 条件语句的写法，编写二级间接索引的部分。

值得注意的是 NDOUBLE_INDIRECT 的意义为块的个数而非直接的序号，所以要通过除以和取余的方式来确定文件的位置，先找第一层的索引，再找第二层的索引。

②在 itrunc()函数中添加如下代码段：

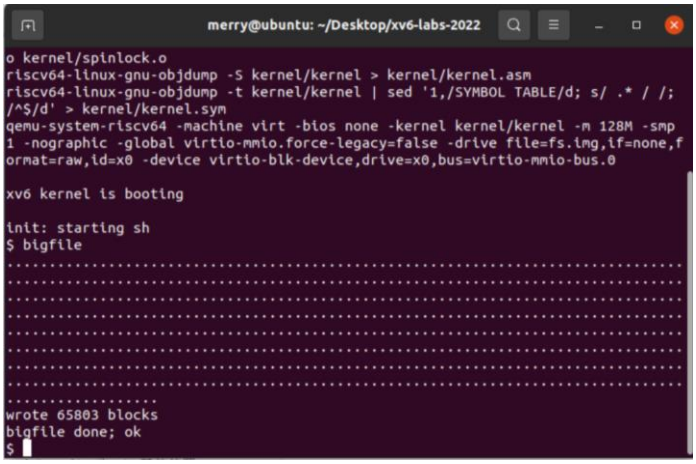
```
if(ip->addrs[NDIRECT + 1]){
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
  a = (uint*)bp->data;
  for(j = 0; j < NINDIRECT + 1; j++){
    if(a[j]){
      bp2 = bread(ip->dev, a[j]);
      a2 = (uint*)bp2->data;
      for(k = 0; k < NINDIRECT; k++){
        if(a2[k]){
          bfree(ip->dev, a2[k]);
        }
      }
      brelse(bp2);
      bfree(ip->dev, a[j]);
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 1]);
  ip->addrs[NDIRECT + 1] = 0;
}
```

模仿先前已有的对直接索引和一级间接索引的处理，但额外加入一个内嵌的 for 循环以处理二级间接索引内容。

其中 for(j = 0; j < NINDIRECT + 1; j++) 用于遍历一级索引，对于每一个一级间接索引内容，都使用一次 for(k = 0; k < INDIRECT; k++)来遍历二级索引。

Part II 实验结果

运行 bigfile 指令，成功通过测试：



2.Symbolic links

Part I 实验步骤

(1)为 symlink 添加系统调用：

①在 user/user.h 中添加一个用户态下调用 symlink 的接口

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

②在 user/usys.pl 中添加 symlink 的入口

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

③在 kernel/syscall.h 中添加系统调用号

```
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_symlink 22
```

④在 kernel/syscall.c 中进行添加 symlink 指令

```
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_symlink(void);
105
106 // An array mapping syscall numbers from syscall.h
107 // to the function that handles the system call.
108 static uint64 (*syscalls[])(void) = {
109 [SYS_fork]    sys_fork,
110 [SYS_exit]    sys_exit,
111 [SYS_wait]    sys_wait,
112 [SYS_pipe]    sys_pipe,
113 [SYS_read]    sys_read,
114 [SYS_kill]    sys_kill,
115 [SYS_exec]    sys_exec,
116 [SYS_fstat]   sys_fstat,
117 [SYS_chdir]   sys_chdir,
118 [SYS_dup]     sys_dup,
119 [SYS_getpid]  sys_getpid,
120 [SYS_sbrk]    sys_sbrk,
121 [SYS_sleep]   sys_sleep,
122 [SYS_uptime]  sys_uptime,
123 [SYS_open]    sys_open,
124 [SYS_write]   sys_write,
125 [SYS_mknod]   sys_mknod,
126 [SYS_unlink]  sys_unlink,
127 [SYS_link]    sys_link,
128 [SYS_mkdir]   sys_mkdir,
129 [SYS_close]   sys_close,
130 [SYS_symlink] sys_symlink,
```

(2)修改 Makefile 文件中的 UPROGS，以便于进行测试：

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_symlinktest\
```

添加测试文件“symlinktest”。

(3)在 kernel/fcntl.h 中添加宏定义 O_NOFOLLOW，设置其值为 0x800，

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x800
```

(4)在 kernel/stat.h 中增加文件类型 T_SYMLINK 以代表软链接：

```
#define T_SYMLINK 4   // symbolic link
```

(5)在 kernel/sysfile.c 中增加函数 sys_symlink：

```
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;
}
```

其中 target 是原本硬链接指向的目标文件，path 是存放软链接文件的路径，ip 指向软链接。

首先使用函数 argstr()读入参数，然后创建一个 inode 赋给 ip，接着使用 writei 函数将 target 的内容写到软链接文件中。

(6)在 kernel/sysfile.c 的函数 sys_open()中添加如下代码段：

```
if (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  int i = 0;
  while (ip->type == T_SYMLINK) {
    if (readi(ip, 0, (uint64)&path, 0, MAXPATH) == -1) {
      iunlockput(ip);
      end_op();
      return -1;
    }
    iunlockput(ip);
    if ((ip = namei(path)) == 0) {
      end_op();
      return -1;
    }
    i++;
    if (i == 10) {
      end_op();
      return -1;
    }
    ilock(ip);
  }
}
```

open()函数的作用是打开一个文件，即在系统文件表里找到空的文件结构，并向其中写入需要的数据，再返回指向该文件结构地址的指针并给出文件描述符。

而这段代码的目的是根据文件的 symbolic link 找到其 inode。

首先通过 O_NOFOLLOW 判断是否为软链接，如果有此标志则为软链接，否则为普通文件。

使用 while 循环，当发现非软链接时跳出循环，循环内部读取文件的路径，并使用变量 path 保存该路径，之后使用 path 去查找文件的 inode。

最后使用 ilock()给 inode 加锁。

Part II 实验结果

运行指令 symlinktest，输出如下：

```
merry@ubuntu:~/Desktop/xv6-labs-2022$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
1 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$
```

二、 实验感想

其实实验过程遇到了很多不顺利的地方，尤其是第二个实验中，因为第一个实验是要处理二级间接索引，新添加的代码大部分可以模仿已有的直接索引、一级索引代码的风格，再加以测试。

跟着实验页面的提示走，除了参考 xv6 手册外，遇到困难时我也上网查找了一些资料，尝试理解别人的代码逻辑，最终也成功解决了问题。理论课上学刚完文件系统部分不久，这次实验正好加深了我对操作系统中的文件系统的理解，让我将理论与实践结合起来，阅读代码的能力也得到了提升。

额外参考的博客链接：

[1] MIT 6.S081 2021: Lab file system, https://zhuanlan.zhihu.com/p/435280998

[2] MIT 6.S081 Lab8 File System,https://www.cnblogs.com/KatyuMarisaBlog/p/14361622.html