Lab9: File System

2351882 王小萌

Tongji University,2025 Summer

代码仓库链接：https://github.com/wxmxmw/Tongji-University-xv6-labs-2021.git

1 Large Files

1.1 实验目的

扩展 xv6 文件系统，实现双层映射的机制，使其支持更大的文件大小。使用三级索引，共有 11 个直接索引，1 个间接索引块和 1 个二级间接索引块，故总共支持 文件大小为 11 + 256 + 256 × 256 = 65803 块。

1.2 实验步骤

1. 打开 kernel/fs.h 中，查找 NDIRECT 和 NINDIRECT 的定义。这些常量表示直接块和单间接块的数量。

```
25  #define FSMAGIC 0x10203040
26
27  #define NDIRECT 12
28  #define NINDIRECT (BSIZE / sizeof(uint))
29  #define MAXFILE (NDIRECT + NINDIRECT)
30
```

2. 在 kernel/file.h 中更新文件的 struct inode 数据结构；
   在 kernel/fs.h 中更新文件的 struct dinode 数据结构。

```
#define FSMAGIC 0x10203040

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

```
// 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]; //-lab9-1
};
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE on
  short minor;          // Minor device number (T_DEVICE on
  short nlink;          // Number of links to inode in file
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
};
```

3. 修改 kernel/fs.c 的 bmap 函数。添加双层间接映射的逻辑。

```
bmap(struct inode *ip, uint bn)
{
  uint addr, *a, *b;
  struct buf *inbp, *ininbp;
```

4. 在 kernel/fs.c 的 itrunc 函数中，添加对双层间接映射的清除逻辑，确保释放双层映射的数据块。

```
461
462     if(ip->addrs[NDIRECT+1]) {
463       bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
464       a = (uint*)bp->data;
465       for (j = 0; j < NINDIRECT; j++) {
466         if (a[j]) {
467           inbp = bread(ip->dev, a[j]);
468           b = (uint*)inbp->data;
469           for (k = 0; k < NINDIRECT; k++) {
470             if (b[k])
471               bfree(ip->dev, b[k]);
472           }
473           brelse(inbp);
474           bfree(ip->dev, a[j]);
475         }
476       }
477       brelse(bp);
478       bfree(ip->dev, ip->addrs[NDIRECT+1]);
479       ip->addrs[NDIRECT+1] = 0;
480     }
481
482     ip->size = 0;
483     iupdate(ip);
484   }
```

5. 运行 xv6，并在命令行中输入 bigfile，结果如下：

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

6. 单项得分：

```
.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/
 .* / /; /^$/d' > kernel/kernel.sym
== Test running bigfile == running bigfile: OK (88.1s)
vale@Puppyyoo: /xv6-labs-2021$
```

1.3 实验中遇到的问题和解决办法

问题 ： 如何理清楚各种数据块号之间的关系。

解决办法 ： xv6 文件系统使用 inode 来管理文件。修改 得其在丢弃 bmap 函数，还需要同步修改 itrunc() ，使 inode 的时候回收所有的数据块。由于添加了二级间接块的结构，因此也需要添加对 该部分的块的释放的代码。释放的方式同一级间接块号的结构，只需要两重循环去分别遍历二级间 接块以及其中的一级间接块。

1.4 实验心得

本次实验中，我成功地修改了 xv6 操作系统的文件系统，实现了双层映射的机制，从而使文件 可以占据更大的大小。这个实验让我更深入地理解了文件系统底层的数据管理和映射逻辑。

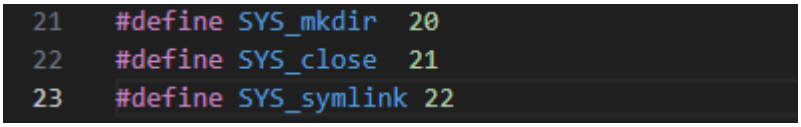
2 Symbolic Links

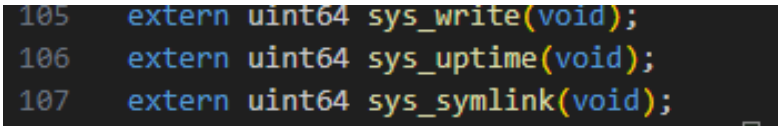
2.1 实验目的

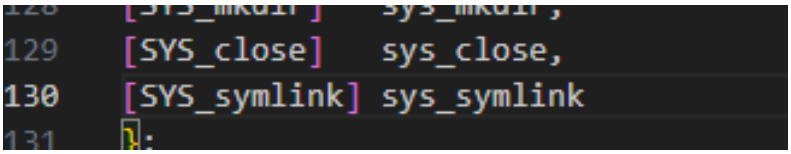
1. 在 xv6 操作系统中实现符号链接（软链接）的功能。

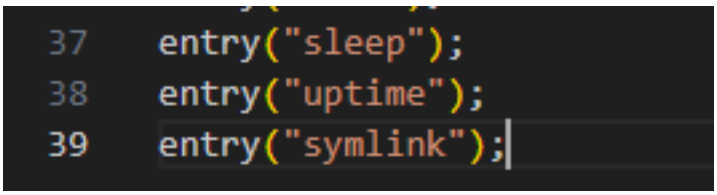
2.2 实验步骤

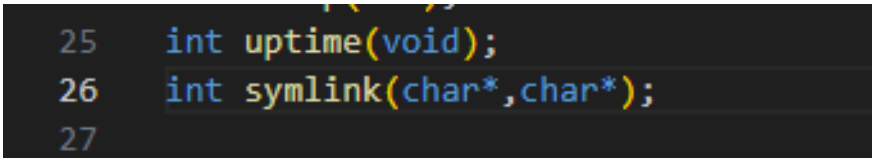
1. 创建系统调用：kernel/syscall.h、kernel/syscall.c 、user/user.h ： 以及 user/usys.pl 中添加有关 symlink 系统调用的定义声明。











2. 在 kernel/stat.h 中添加一个新的文件类型 T_SYMLINK ， 用于表示符号链接。

kernel > C stat.h

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

3. 在 kernel/fcntl.h 中添加一个新的打开标志 O_NOFOLLOW ， 该标志可以与 open 系统调用一起使用。

kernel > C fcntl.h

```
#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/sysfile.c 中实现 sys_symlink 系统调用，将目标路径写入新创建的符号链接文件 的数据块中。

```
int sys_symlink(char *target, char *path) {
  char kpath[MAXPATH], ktarget[MAXPATH];
  memset(kpath, 0, MAXPATH);
  memset(ktarget, 0, MAXPATH);
  struct inode *ip;
  int n, r;

  if((n = argstr(0, ktarget, MAXPATH)) < 0)
    return -1;

  if ((n = argstr(1, kpath, MAXPATH)) < 0)
    return -1;

  int ret = 0;
  begin_op();

  if((ip = namei(kpath)) != 0){
    // symlink already exists
    ret = -1;
    goto final;
  }
  // create an inode block for the symlink
  ip = create(kpath, T_SYMLINK, 0, 0);
  if(ip == 0){
    ret = -1;
    goto final;
  }
  // write the target path into inode's data block
  if ((r = writei(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0)
    ret = -1;
  iunlockput(ip);

final:
  end_op();
  return ret;
}
```

5. 修改 kernel/sysfile.c 中的 open 系统调用，以处理路径引用到符号链接的情况。

```
// deal with a symbolic link if it is and no_follow flag is not set
int depth = 0;
while (ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
  char ktarget[MAXPATH];
  memset(ktarget, 0, MAXPATH);
  if ((r = readi(ip, 0, (uint64)ktarget, 0, MAXPATH)) < 0) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlockput(ip);
  if((ip = namei(ktarget)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  depth++;
  if (depth > 10) {
    // maybe form a cycle
    iunlockput(ip);
    end_op();
    return -1;
  }
}
```

6. 在 Makefile 中添加对测试文件 symlinktest.c 的编译

```
Makefile
174    UPROGS=\
191        $U/_symlinktest\
192
```

7. 执行 make qemu，运行 symlinktest。结果如下：

```
xv6 kernel is booting

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

8. 单项测试得分：

```
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-fs symlink
make: 'kernel/kernel' is up to date.
== Test running symlinktest == (1.1s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
vale@Puppyyoo:~/xv6-labs-2021$
```

2.3 实验中遇到的问题和解决办法

问题：如何正确处理软链接的创建和打开逻辑。

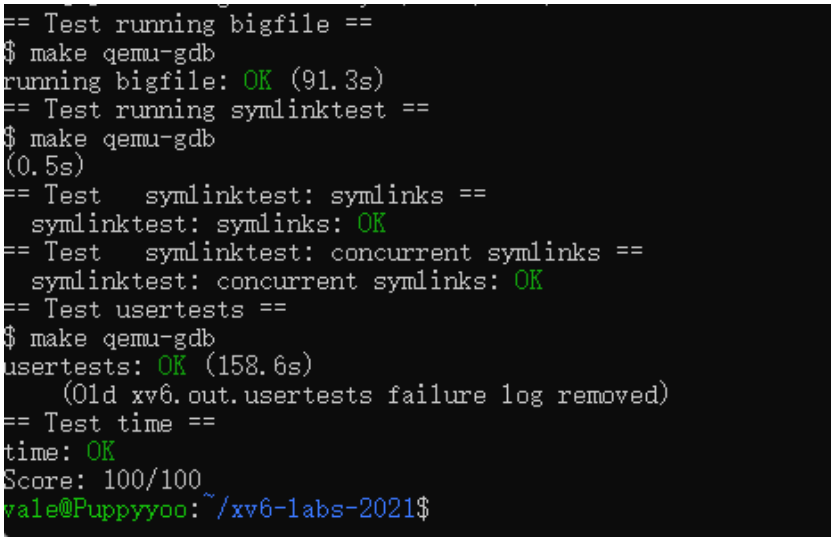
解决办法：在 sys_symlink 中，将目标路径写入符号链接的数据块。在 sys_open 中，对打开 的文件进行判断，如果是符号链接则递归解析，直至找到实际文件或检测到循环。

2.4 实验心得

在本次实验中，我成功地向 xv6 操作系统添加了符号链接（软链接）的支持。符号链接是一种特殊类型的文件，可以跨越磁盘设备引用其他文件。

3 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。
2. 在终端中执行 make grade,得到 lab9 总分：



3. lab9 代码提交