Lab6 File System

21307110014

1 大文件

本任务要求增大xv6支持的最大文件大小，通过把一个direct块变成doubly-direct块来实现。

首先修改fs.h中的宏定义`NDIRECT`，将其值从12改为11，再加入宏`NININDIRECT`并修改`MAXFILE`的定义：

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

`NININDIRECT`即一个doubly-direct块实际能放多少个块。这样文件大小限制`MAXFILE`增大了`NININDIRECT-1`。同时还需要修改fs.h及file.h中`addrs`成员变量的大小为`NDIRECT+1+1`。

然后需要修改fs.c中的`bmap`，增加`bn`大于`NINDIRECT`时的处理。

```
  // ...
  bn -= NINDIRECT;
  if(bn < NININDIRECT){
    // Load doubly-indirect block, allocating if necessary.
    if ((addr = ip->addrs[NDIRECT+1]) == 0)
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    // 第一级
    a = (uint*)bp->data;
    i = bn/NINDIRECT;
    if((addr = a[i]) == 0){
      a[i] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    // 第二级
    a = (uint*)bp->data;
    i = bn%NINDIRECT;
    if((addr = a[i]) == 0){
      a[i] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }
```

其中第一级和第二级的索引分别为`bn/NINDIRECT`和`bn%NINDIRECT`可以用类似二维数组的方式来理解。

在fs.c中的`itrunc`中也要释放掉doubly-indirect块中每一级的块。

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

2 符号链接

本任务要求在xv6中支持符号连接。

首先实现创建符号链接的系统调用`symlink(char *target, char *path)`，在`path`处创建一个指向`target`的符号链接。在syscall.h等文件中注册新系统调用后，在sysfile.c中实现`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();
  if((ip = namei(path)) == 0){ // the path inode doesn't exist yet
    ip = create(path, T_SYMLINK, 0, 0);
    iunlock(ip);
  }
  ilock(ip);
  // write the target into the path inode
  if(writei(ip, 0, (uint64)target, ip->size, MAXPATH) != MAXPATH){
    panic("symlink");
  }
  iunlockput(ip);
  end_op();
  return 0;
}
```

接下来修改`sys_open`，加入对symlink文件类型的处理。如果打开一个符号链接，除非指定`O_NOFOLLOW`，`open`需要顺着链接追踪到非symlink文件。

```
  if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
    for(int cnt = 0; ip->type == T_SYMLINK; cnt++){
      if(cnt >= 10){ // cycle
        iunlockput(ip);
        end_op();
        return -1;
      }
      if(readi(ip, 0, (uint64)path, ip->size - MAXPATH, MAXPATH) != MAXPATH){
        panic("open symlink");
      }
      iunlockput(ip);
      if((ip=namei(path)) == 0){ // target file not found
        end_op();
        return -1;
      }
      ilock(ip);
    }
  }
```

最后在fcntl.h和stat.h中分别加入上面用到的宏定义`O_NOFOLLOW`和`T_SYMLINK`。

运行结果

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (131.6s)
== Test running symlinktest ==
$ make qemu-gdb
(0.9s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (213.2s)
== Test time ==
time: OK
Score: 100/100
```