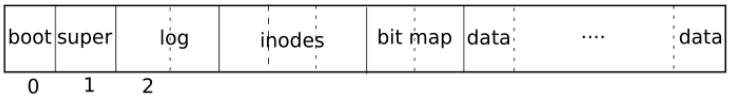
Lab9: file system 实验报告

做过课内操作系统实验“基于FUSE架构的青春版EXT2文件系统”后，实现本次实验较为简单

xv6 的文件系统和EXT2文件系统有诸多相似之处。xv6文件系统中，物理磁盘读写扇区大小为**512B**，以block形式存储文件，一个block为两个扇区，即一个block为**1024B**

文件系统磁盘布局如下：



- 第0块为启动区
- 第1块为超级块
- **log区域用于故障恢复**
- inodes存储索引节点，文件信息存储在索引节点inode中，每个文件对应一个inode，inode中还含有存放文件内容的磁盘块的索引信息
- **bit map记录数据区使用情况**
- data为数据区

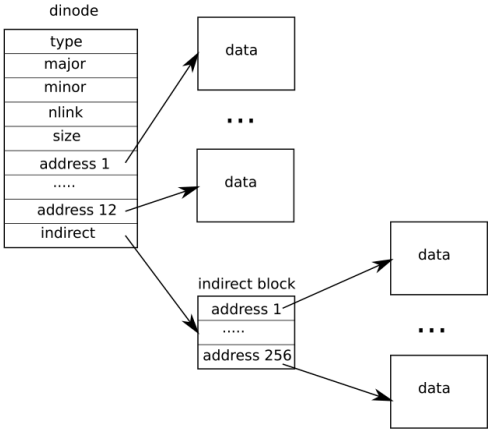
---

1. 首先切换分支：

```
$ git fetch
$ git checkout fs
$ make clean
```

Large files(moderate)

目前xv6系统中的 inode 有 12个直接索引（直接对应了 data 区域的磁盘块），1个一级索引（存放另一个指向 data 区域的索引）。因此，**最多支持 12 + 256 = 268 个数据块。**



`bigfile`希望能够创建一个包含65803个块的文件，但未修改的xv6将文件限制为268个块。因此我们需要修改索引节点结构，使其包含11个直接索引节点、1个一级索引节点和1个二级索引节点，此时文件最大可以包含**11+256+256*256=65803 个数据块**

1. 修改 *kernel/fs.h* 中的直接索引节点个数、单个文件支持的最大磁盘块数以及 struct dinode（磁盘上的 inode 结构体）

```c
#define NDIRECT 11  // 直接索引节点个数
#define NINDIRECT (BSIZE / sizeof(uint))  // 一级索引节点指向磁盘数
#define NDINDIRECT  (NINDIRECT * NINDIRECT) // 二级索引节点指向磁盘数
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)  //单个文件最多可以占据的磁盘块数

// 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
};
```

2. 修改 *kernel/file.h* 中的 struct inode（内存中的 inode 结构体）

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

3. 修改 *kernel/fs.c* 中的bmap函数，仿照一级索引，写下二级索引：

```
    return addr;
  }
  bn -= NINDIRECT;

  if(bn < NDINDIRECT){
    if((addr = ip->addrs[NDIRECT + 1]) == 0)  // 如果二级索引块还未分配
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
    // 读取二级索引块数据到缓冲区中
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    // 判断次级目录项是否存在
    if((addr = a[bn/NINDIRECT]) == 0){
      a[bn/NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);

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

    // 判断bn对应的数据块是否存在
    if((addr = a[bn%NINDIRECT]) == 0){
      a[bn%NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```

4. 修改 *kernel/fs.c* 中的itrunc函数，模仿一级索引的释放，添加第二级索引的释放操作：

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

  ip->size = 0;
  iupdate(ip);
}
```

5. 执行测试命令：`bigfile`

```
220110512@comp4:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive fil
e=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ bigfile
..........................................................................................................
..........................................................................................................
..........................................................................................................
..........................................................................................................
..........................................................................................................
..........................................................................................................
....................................
wrote 65803 blocks
bigfile done; ok
$
```

执行测试命令：`usertests`

```
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

Symbolic links(moderate)

关于**硬链接**和**软链接**：

- 硬链接是指多个文件名指向同一个inode号码。有以下特点：
  - 可以用不同的文件名访问同样的内容；
  - 对文件内容进行修改，会影响到所有文件名；
  - 删除一个文件名，不影响另一个文件名的访问。
- 软链接也是一个文件，但是文件内容指向另一个文件的 inode。打开这个文件时，会自动打开它指向的文件，类似于 windows 系统的快捷方式。

xv6 中没有符号链接（软链接），这个任务需要我们实现一个符号链接

1. 在*kernel/stat.h*中添加新文件类型 `T_SYMLINK`，表示符号链接：

```
4  #define T_SYMLINK 4   // Symlink
```

2. 在*kernel/fcntl.h*中补齐 `O_NOFOLLOW` 的定义，不应与任何现有标志重叠：

```
6  #define O_NOFOLLOW   0x800
```

3. 增加symlink的系统调用：

   - *user/usys.pl*：

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

   - *user/user.h*：

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

   - *kernel/syscall.h*：

```
23  #define SYS_symlink 22
```

   - *kernel/syscall.c*：

```
extern uint64 sys_symlink(void);

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_symlink] sys_symlink,
};
```

- 在 *kernel/sysfile.c*中实现系统调用：

```
uint64
sys_symlink(void)
{
  char path[MAXPATH], target[MAXPATH];
  struct inode *ip;
  // 读取参数
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;

  // 开启文件系统操作
  begin_op();

  // 创建类型为T_SYMLINK的新inode
  if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
    end_op();
    return -1;
  }
  // 在符号链接的文件中写入被链接的文件target
  if(writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
    iunlockput(ip);
    end_op();
    return -1;
  }
  // 提交事务
  iunlockput(ip);
  end_op(); //结束文件系统操作
  return 0;
}
```

4. 在 *kernel/sysfile.c*修改sys_open函数，添加对符号链接的处理，如果遇到符号链接，直接打开对应的文件。这里为了避免符号链接彼此之间互相链接，导致死循环，设置了一个访问深度（我设成了30），如果到达该访问次数，则说明打开文件失败

```
300     if(omode & O_CREATE){
302       if(ip == 0){
303         end_op();
304         return -1;
305       }
306     } else {
307       int depth;
308       for(depth = 0; depth < 10; depth++) {
309         // 查找path对应的inode
310         if((ip = namei(path)) == 0){
311           end_op();
312           return -1;
313         }
314         ilock(ip);
315         if(ip->type == T_DIR && omode != O_RDONLY){
316           iunlockput(ip);
317           end_op();
318           return -1;
319         }
320         // 如果是符号链接，则循环处理，直到找到真正的文件为止
321         if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
322           if(readi(ip, 0, (uint64)path, 0, MAXPATH) < 0) {
323             iunlockput(ip);
324             end_op();
325             return -1;
326           }
327         } else {
328           break;
329         }
330         iunlockput(ip);
331       }
332       // 循环超过了一定的次数，可能发生了循环链接，返回-1
333       if(depth == 10) {
334         end_op();
335         return -1;
336       }
337     }
```

5. 将 `symlinktest` 添加到*Makefile*:

```
178     $U/_symlinktest\
```

6. 执行测试命令： `symlinktest`

```
220110512@comp4:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive fil
e=fs.img,if=none,format=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
$
```

执行测试命令：`usertests`

```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

结果截图

由于大文件需要运行的时间较长，所以需要延长 *grade-lab-fs* 中 timeout 的时间：

```
@test(40, "running bigfile")
def test_bigfile():
    r.run_qemu(shell_script([
        'bigfile'
    ]), timeout=600)
    r.match('^wrote 65803 blocks$')
    r.match('^bigfile done; ok$')
```

```
@test(19, "usertests")
def test_usertests():
    r.run_qemu(shell_script([
        'usertests'
    ]), timeout=800)
    r.match('^ALL TESTS PASSED$')
```

执行命令`make grade`

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (350.8s)
== Test running symlinktest ==
$ make qemu-gdb
(1.4s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (603.4s)
== Test time ==
time: OK
Score: 100/100
220110512@comp4:~/xv6-labs-2020$
```