Lab 9: file system

1950787 杨鑫

实验目的：

- 加深对文件系统的理解
- 理解文件存储方式和 inode 的组织架构，并修改相应结构来扩充文件的最大大小
- 链接文件符号链接的含义和作用，以及实现方式
- 实现文件符号链接，并实现打开符号链接文件的功能

实验步骤：

首先切换分支至 fs branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout fs
make clean
```

a. Large files

实验目的：

在此作业中，您将增加 xv6 文件的最大大小。目前 xv6 文件限制为 268 个块，或 268 * BSIZE 字节（在 xv6 中 BSIZE 为 1024）。这个限制来自这样一个事实，一个 xv6 inode 包含 12 个“直接”块号和一个“一级间接”块号，这是指一个块最多可以容纳 256 个块号，总共 12 + 256=268块。
您将更改 xv6 文件系统代码以支持每个 inode 中的“二级间接”块，其中包含 256 个单间接块地址，每个块最多可包含 256 个数据块地址。结果将是一个文件将能够包含多达 65803 个块，或 256 * 256 + 256 + 11 个块（11 个而不是 12 个，因为我们将为双间接块牺牲一个直接块号）。
修改 bmap()，使其除了直接块和单间接块外，还实现双重间接块。你只需要 11 个直接块，而不是 12 个，就可以为新的双重间接块腾出空间；您不能更改磁盘 inode 的大小。 ip->addrs[] 的前 11 个元素应该是直接块；第 12 个应该是一个单独的间接块（就像现在的块一样）；第 13 个应该是你新的双重间接块。

实验分析：

首先查阅文档查看文件的 inode 的组织结构：

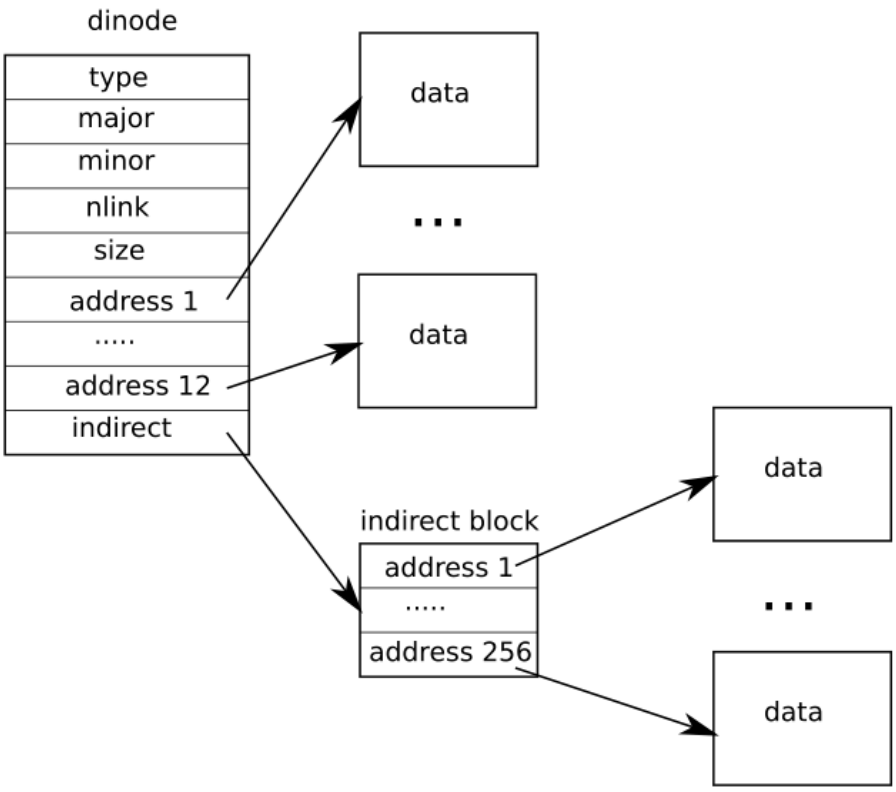


Figure 8.3: The representation of a file on disk.

可以看到初始为 12 个直接块和 1 个一级索引块，文件最多含 12 + 256 = 268 个块；而扩张后为 11 个直接块和 1 个一级索引和 1 个二级索引块，大小变为 11 + 256 + 256 * 256 个块。于是要修改相应的常量以适应现在的结构；同时要修改 bmap 以确保每次从文件中可以获取指定块的内容。

实验核心代码：

fs.h

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

// 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
};
```

fs.c

```
// Inode content
//
```

```
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  bn -= NINDIRECT;
  if (bn < NDOUBLEINDIRECT) {
    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 / NINDIRECT]) == 0) {
      a[bn / NINDIRECT] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    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");
}
```

测试：

```
xv6 kernel is booting

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

```
xv6 kernel is booting

uinit: starting sh
$ usertests
exec uusertests failed
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: 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
```

可以通过 bigfile 和 usertests 测试

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-fs bigfile
make: “kernel/kernel”已是最新。
== Test running bigfile == running bigfile: OK (164.3s)
```

可以通过所有测试

b. Symbolic links

实验目的：

在本练习中，您将向 xv6 添加符号链接。符号链接（或软链接）是指通过路径名链接的文件；当一个符号链接被打开时，内核会跟随链接指向被引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管 xv6 不支持多个设备，但实现这个系统调用是一个很好的练习，可以帮助您了解路径名查找的工作原理。

您将实现 symlink(char *target, char *path) 系统调用，它会在 path 处创建一个新的符号链接，该链接引用由 target 命名的文件。有关详细信息，请参阅手册页符号链接。要进行测试，请将 symlinktest 添加到 Makefile 并运行它。

实现 symlink(target, path) 系统调用以在指向目标的路径上创建一个新的符号链接。请注意，系统调用成功时不需要存在目标。您将需要选择某个位置来存储符号链接的目标路径，例如，在 inode 的数据块中。 symlink 应该返回一个表示成功 (0) 或失败 (-1) 的整数，类似于链接和取消链接。

修改 open 系统调用以处理路径引用符号链接的情况。如果文件不存在，则打开必须失败。当进程在要打开的标志中指定 O_NOFOLLOW 时， open 应该打开符号链接（而不是跟随符号链接）。

如果链接文件也是符号链接，则必须递归地跟随它，直到到达非链接文件。如果链接形成循环，则必须返回错误代码。如果链接的深度达到某个阈值（例如，10），您可以通过返回错误代码来近似此值。

其他系统调用（例如，链接和取消链接）不得遵循符号链接；这些系统调用对符号链接本身进行操作。对于本实验，您不必处理指向目录的符号链接。

实验分析：

本次实验要求实现符号链接。首先按照以前添加系统调用的方式修改若干文件并添加 sys_symlink 的声明，然后自定义一些常量以方便后续使用。然后实现 sys_symlink：即打开 path 指定的文件，然后将需要链接的路径名写入其 inode 的第一个块中即可。

然后要修改 sys_open 以实现打开这类符号链接文件：若打开模式是 NOFOLLOW 的，则会循环的打开该文件，直到文件格式不是链接类型或者循环次数超过10（近似看做循环链接），否则打开失败。

实验核心代码：

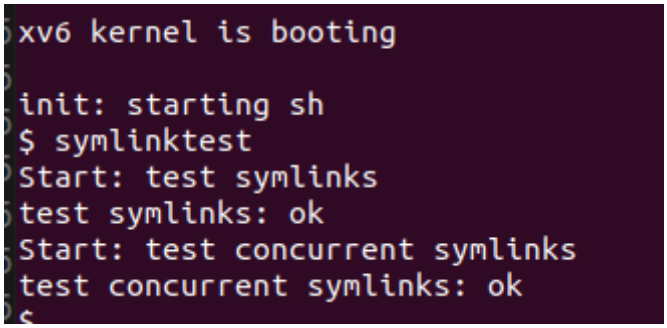
sysfile.c

```
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip, *dp;
  int n;

  if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
```

```
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

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

  if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
    iunlockput(ip);
    end_op();
    return -1;
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }

  if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
```

```
    f->major = ip->major;
  } else {
    f->type = FD_INODE;
    f->off = 0;
  }
  f->ip = ip;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

  if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
  }

  iunlock(ip);
  end_op();

  return fd;
}


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, strlen(target)) != strlen(target)) {
    end_op();
    return -1;
  }
  iunlockput(ip);
  end_op();
  return 0;
}
```

测试：



```
xv6 kernel is booting

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: 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
```

可以通过 symlinktest 和 usertests

实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：

```
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (159.1s)
== Test running symlinktest ==
$ make qemu-gdb
(0.7s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (284.3s)
== Test time ==
time: OK
Score: 100/100
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

通过了所有测试

问题以及解决办法：

a.文件的最大大小

在 xv6 操作系统中，文件的最大大小是有限制的，这取决于 inode 的组织结构，初始为 12 个直接块和 1 个一级索引块，文件最多含 12 + 256 = 268 个块；而想要扩充后的文件最大大小为 11 个直接块和 1 个一级索引和 1 个二级索引块，大小变为 11 + 256 + 256 * 256 个块。

这就需要修改文件 inode 的结构，并且在 bmap 中增加二级索引节点数据块的读取方法。具体来说就是先找到一级索引 i / NINDIRECT 的位置，然后在在一级索引指出的表中找到直接存储数据的块（第 i % NDIRECT 个）。

b.文件系统调用的使用

查看源码可知，使用文件系统调用前，必须使用 begin_op()，使用完毕后，必须使用 end_op()。而且 inode 指针在使用完毕正常退出时，需要使用 iunlockput() 将其解锁并放回。

c. 符号链接的递归打开

由于符号链接可能指向另一个符号链接文件，所以我们必须追踪至真正的文件。但是有可能符号链接文件是一个循环，这会导致打开出现死循环，从而使程序无法正常运行；所以，我们这里设置循环打开次数不得大于 10 次，用于近似死循环的情况（当然，这样做并不完美，最好还是使用一种真正可以判断死循环的方法，这里做了一些简化）

实验心得：

1. 本次实验熟悉了文件结构，以及文件的 inode 和文件的内容组织形式
2. 文件最大大小和 inode 中一些结构的组织架构有关，可以修改这些结构以改变文件最大大小
3. 了解了符号链接文件，以及相关操作
4. 实现符号链接文件的创建和打开功能，体会更加深刻