**实验九 file system**

**9.1 实验目的**

1. 了解UNIX系统以及xv6系统组织文件的方式

2. 理解直接块号和间接块号

3. 学习硬链接和软链接的实现方式

**9.2 实验内容**

9.2.1 Large files（moderate）

修改bmap（），以便除了直接块和单间接块之外，它还实现双间接块。你只需要有11个直接块，而不是12个，为你的新的双间接块腾出空间；不允许更改磁盘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（与当前块一样）；13号应该是你的新双间接块。当bigfile写入65803个块并成功运行usertests时，完成此练习：

$ **bigfile**

..................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................

wrote 65803 blocks

done; ok

$ usertests

...

ALL TESTS PASSED

$

9.2.2 Symbolic links（moderate）

您将实现symlink（char\*target，char\*path）系统调用，该调用在引用由target命名的文件的路径处创建一个新的符号链接。有关更多信息，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。当测试产生以下输出（包括usertests）时，您的解决方案就完成了。

$ **symlinktest**

Start: test symlinks

test symlinks: ok

Start: test concurrent symlinks

test concurrent symlinks: ok

$ usertests

...

ALL TESTS PASSED

$

9.2.3 submit

提交实验。

**9.3 实验过程**

9.3.1 large files

（1）阅读xv6-book文件系统相关部分并阅读fs.c中的bmap()。

（2）修改fs.h的宏定义和结构体

#define NDIRECT 11

#define NINDIRECT (BSIZE / sizeof(uint))

#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT \* NINDIRECT)

struct dinode {

 uint addrs[NDIRECT+2]; // Data block addresses

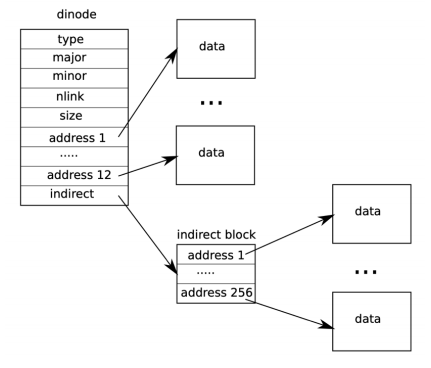
};

struct inode {

 uint addrs[NDIRECT+2]; // Data block addresses

};

（3）原本bmap()支持以及块表，如下图所示



现修改bmap()使其支持二级块表

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;

  }

  panic("bmap: out of range");

}

// Truncate inode (discard contents).

// Caller must hold ip->lock.

void

itrunc(struct inode \*ip)

{

  int i, j;

  struct buf \*bp;

  uint \*a;

  for(i = 0; i < NDIRECT; i++){

    if(ip->addrs[i]){

      bfree(ip->dev, ip->addrs[i]);

      ip->addrs[i] = 0;

    }

  }

  if(ip->addrs[NDIRECT]){

    bp = bread(ip->dev, ip->addrs[NDIRECT]);

    a = (uint\*)bp->data;

    for(j = 0; j < NINDIRECT; j++){

      if(a[j])

        bfree(ip->dev, a[j]);

    }

    brelse(bp);

    bfree(ip->dev, ip->addrs[NDIRECT]);

    ip->addrs[NDIRECT] = 0;

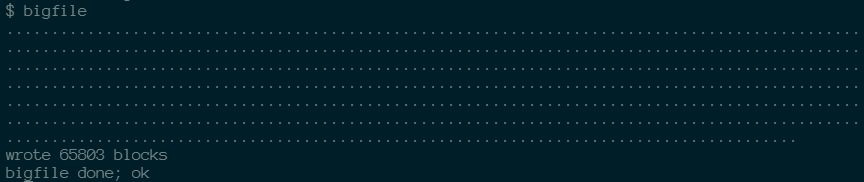
  }

  ip->size = 0;

  iupdate(ip);

}

（4）运行largefile



9.3.2 symbolic links

（1）硬连接指通过索引节点来进行连接。在Linux的文件系统中，保存在磁盘分区中的文件不管是什么类型都给它分配一个编号，称为索引节点号(Inode Index)。在Linux中，多个文件名指向同一索引节点是存在的。一般这种连接就是硬连接。硬连接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接到重要文件，以防止“误删”的功能。其原因如上所述，因为对应该目录的索引节点有一个以上的连接。只删除一个连接并不影响索引节点本身和其它的连接，只有当最后一个连接被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除。

（2）另外一种连接称之为符号连接（Symbolic Link），也叫软连接。软链接文件有类似于Windows的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

（3）sys\_link的作用就是 假设原本有目录 /a/b 现在调用path( /a/b , /c/d ) 那么目录a指向b的同时还指向c目录下的d ，但是二者必须在同一磁盘之上，软连接sys\_symlink就是取消了二者必须在同一磁盘上的限制实现sys\_symlink,在dp->data添加MAXPATH个字节 保存symlink指向的target path。sys\_symlink实现如下。

uint64 sys\_symlink(void)

{

   char name[DIRSIZ],target[MAXPATH],path[MAXPATH];

   struct inode \*dp,\*ip ;

   if(argstr(0,target,MAXPATH) <0 || argstr(1,path,MAXPATH)<0)

   {

      return -1;

   }

   begin\_op();

   //target does not need to exist for the system call

   // just store it in path inode's data block

   if((ip=namei(target)) != 0 && ip->type!=T\_DIR  ) // it's ok if target doesn's exist

   {

      //if ip exist,increase nlink

      ilock(ip);

      ip->nlink++;

      iupdate(ip);

      iunlockput(ip);

   }

   if((dp = namei(path)) ==0 )//path对应inode节点不存在

   {

      if((dp=nameiparent(path,name))==0)//父目录也不存在

      {

         printf("NO inode corresponding to path's parent\n");

         goto bad;

      }

      else

      {

         //利用create函数新建一个inode

         if((dp=create(path,T\_SYMLINK,0,0))==0)// last two arguments are for T\_DEVICE only

     {

           printf("create symlink for path fail!\n");

     goto bad;

     }

      else

      {

     iunlock(dp);

       }

      }

   }

   ilock(dp);

   //store target in the end of directory dp's data block

   writei(dp,0,(uint64)target,dp->size,MAXPATH);

   dp->type=T\_SYMLINK;

   iunlockput(dp);

   end\_op();

   return 0;

   bad:

     end\_op();

     return -1;

}

（4）修改sys\_open()，将namei(path) 替换成自定义函数getip()

struct inode\* getip(char \*path,uint depth ,int omode)

{

    // reached the max symbolic recursive depth

    if(depth > SYMLINK\_MAX\_DEPTH)

    {

       return 0;

    }

    struct inode \*ip;

    // file not exist

    if((ip = namei(path)) == 0){

      return 0;

    }

    ilock(ip);

    if(! (omode & O\_NOFOLLOW ) && ip->type==T\_SYMLINK )

    {

       //path reference to a symbolic link and O\_NOFOLLOW flag is not set

       //recursively follow the symbolic link until a non-link file is reached

       char next[MAXPATH];

       if(readi(ip,0,(uint64)next,ip->size-MAXPATH ,MAXPATH) == 0 ) // 读ip->data的最后MAXPATH个字节作为ip的symlink指向的target path

       {

         iunlock(ip);

         return 0;

       }

       iunlock(ip);

       return  getip(next, depth+1 ,omode);

     }

     iunlock(ip);

     return ip;

}

uint64

sys\_open(void)

{

  char path[MAXPATH];

  int fd, omode;

  struct file \*f;

  struct inode \*ip;

  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;

    }\*/

    if((ip=getip(path , 0 , omode )) == 0)

    {

       end\_op();

       return -1;

    }

    ilock(ip);

    if(ip->type == T\_DIR && omode != O\_RDONLY){

      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);

  }

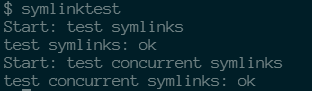
  iunlock(ip);

  end\_op();

  return fd;

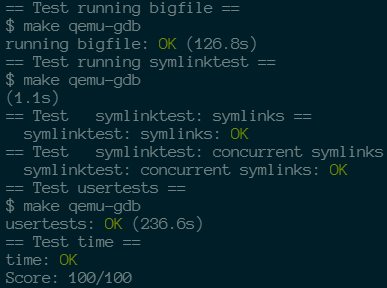
}

（5）运行sysinfo



**9.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**9.5 实验小结**

本次实验考察了unix系统的文件组织结构，并且让我加深了对一级间接文件结构和二级间接文件结构的理解和认识。同时，在编写软链接的时候我进一步区分了硬链接和软链接的区别，也理解了为什么说“软链接也是一种文件”的概念。这种思路将复杂的系统简化为了几种类型让整个逻辑结构更加直观了。

。