**Lab 9 file system**

1. **Large files (moderate)**
   1. **实验目的**

通过使用二级索引的方式扩充xv6文件系统支持的文件大小。

xv6的文件inode结构体中，包括了12个直接块与1个间接块，共能支持12+1\*256=268块（xv6中BSIZE=1024），即文件大小限制为268\*1024bytes.本实验要通过减少1个直接块，增加1个双重间接块来实现支持11+1\*256+256\*256=65803块的文件大小实现。

* 1. **实验步骤**
     1. 更改部分系统结构体定义

kernel/fs.h:修改直接块数量宏定义与dinode(磁盘中文件inode结构)

......

//#define NDIRECT 12

#define NDIRECT 11 //减少一个直接块,增加一个双间接块(lab9-1)

#define NINDIRECT (BSIZE / sizeof(uint)) //单间接块存储指针数量 256

#define NDINDIRECT (NINDIRECT \* NINDIRECT) //双间接块存储指针数量 256\*256(lab9-1)

#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT) //最大文件上限(lab9-1)

// On-disk inode structure

//NDIRECT+1->NDIRECT+2(lab9-1)

struct dinode {

short type; // 文件类型 (T\_FILE, T\_DIR, T\_DEV)

short major; // 设备类型的 major 设备号

short minor; // 设备类型的 minor 设备号

short nlink; // 链接计数器

uint size; // 文件大小（字节数）

uint addrs[NDIRECT+2]; // 直接块和间接块的块号

};

......

kernel/file.h:

对应修改内存中inode结构

......

// 修改了NDIRECT+1->NDIRECT+2

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

};

* + 1. 更改bmap函数以实现双重间接块索引

kernel/fs.c/ static uint bmap(struct inode \*ip, uint bn)

首先将 bn 减去 NINDIRECT，将逻辑块号转换为相对于双间接块的块号。

//根据逻辑块号所处区间不同决定哪种块及读取方式

//下级块通过balloc分配时,需要用log\_write更新上级磁盘块

static uint

bmap(struct inode \*ip, uint bn)

{

uint addr, \*a;

struct buf \*bp;

if(bn < NDIRECT){...} //逻辑块号为直接块

bn -= NDIRECT;//转换为相对于单间接块的块号(11~267->0~255)

if(bn < NINDIRECT){...}//逻辑块号为单间接块

//新增部分

bn -= NINDIRECT;//转换为相对于双间接块的块号(268~65803->0~65535(上面减过NDIRECT了))

如果逻辑块号小于 NDINDIRECT，说明它在双间接块范围内，继续执行以下操作，否则抛出 panic。

if(bn < NDINDIRECT){//逻辑块号为双间接块

先检查 inode 结构中的双间接块地址（ip->addrs[NDIRECT + 1]），如果该地址为 0，表示双间接块还没有分配，需要调用 balloc 分配一个块，并更新 inode 结构中的地址。

if((addr = ip->addrs[NDIRECT + 1]) == 0)//获取双间接块地址

ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

读取双间接块内容，获得一个指向单间接块的数组指针 a。

在单间接块中查找逻辑块号对应的单间接块的地址。由于单间接块内可以存放多个直接块的地址，这里需要将 bn / NINDIRECT 来定位单间接块的索引。如果地址为 0，表示该单间接块还没有分配，需要调用 balloc 分配一个块，并在 a 数组中记录这个块的地址。然后，使用 log\_write 函数记录这次写操作。

释放之前读取的双间接块缓存。

再次读取刚刚分配的单间接块，获得一个指向直接块的数组指针 a。

在直接块数组中查找逻辑块号对应的直接块的地址。由于直接块数组内存放直接块的地址，这里只需要将 bn 对 NINDIRECT 取模即可。如果地址为 0，表示该直接块还没有分配，需要调用 balloc 分配一个块，并在 a 数组中记录这个块的地址。然后，再次使用 log\_write 函数记录这次写操作。

释放刚刚读取的直接块缓存。

返回最终找到的直接块的地址。

if(bn < NDINDIRECT){//逻辑块号为双间接块

if((addr = ip->addrs[NDIRECT + 1]) == 0)//获取双间接块地址

ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if((addr = a[bn / NINDIRECT]) == 0) {//获取单间接块地址

a[bn / NINDIRECT] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

bn %= NINDIRECT;

if((addr = a[bn]) == 0) {//获取直接块地址

a[bn] = addr = balloc(ip->dev);

log\_write(bp);

}

brelse(bp);

return addr;

}

panic("bmap: out of range");

};

......

* + 1. 更改itrunc(inode truncate)以实现二级索引的释放操作

kernel/fs.c

首先，代码遍历 inode 中的直接块（前 NDIRECT 个块），并释放它们所占用的磁盘块。具体做法是调用 bfree 函数释放磁盘块，然后将 inode 结构中的块地址置为 0。

如果 inode 中的第 NDIRECT 块（单间接块）有分配，说明文件之前的大小超过了 NDIRECT 个块，需要继续处理间接块。进入新增部分的代码。

从磁盘读取双间接块的内容，并获得单间接块的数组指针 a。

遍历单间接块中的每个单块（指向直接块的块），如果单块有分配，就进入下一层循环。

从磁盘读取该直接块的内容，并获得直接块的数组指针 a2。遍历直接块中的每个块，如果块有分配，则调用 bfree 函数释放该块，并将 a2 数组中的对应项置为 0。

释放当前直接块的缓存（bp2），调用 bfree 函数释放当前单块，并将单间接块数组 a 中的对应项置为 0，释放双间接块的缓存（bp）。

调用 bfree 函数释放 inode 中的双间接块，同时将 inode 结构中的相应块地址置为 0。

将 inode 的大小（文件大小）置为 0，然后调用 iupdate 函数更新 inode 结构信息。

......

#void

itrunc(struct inode \*ip)

{

int i,j;

struct buf \*bp, \*bp2;

uint \*a, \*a2;

for(i = 0; i < NDIRECT; i++){...}

if(ip->addrs[NDIRECT]){...}

//新增部分

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] = 0;

}

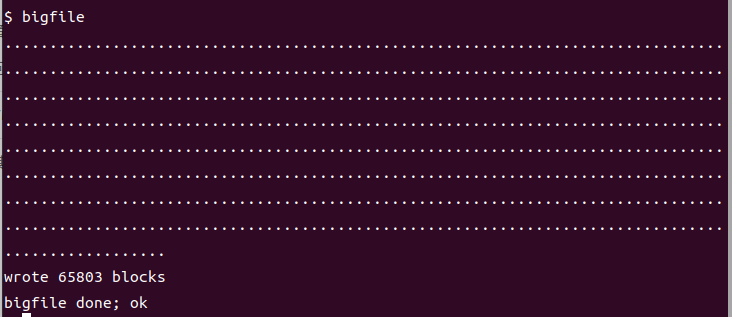
ip->size = 0;

iupdate(ip);

}

......

* + 1. make测试



* 1. **实验中遇到的问题和解决方法**

问题：make qemu中运行bigfile（以及后一个实验symlinktest出现同样问题）正常运行，但make grade中出错。

解决方法：修改grade-lab-fs中的超时限制至足够大(180->1800)

@test(40, "running bigfile")

def test\_bigfile():

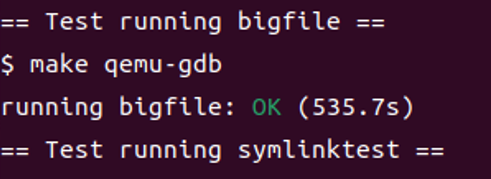
r.run\_qemu(shell\_script([

'bigfile'

]), timeout=1800)

r.match('^wrote 65803 blocks$')

r.match('^bigfile done; ok$')



问题：进程不正常退出，无法完成。

解决方式：发现原因：编写sys\_open函数时手滑，将块下标写错了。文件内下标小于NDIRECT为直接块，等于NDIRECT的块为单间接块，NDIRECT+1为双间接块。

* 1. **实验心得**

通过修改 xv6 文件系统来支持更大文件和添加符号链接，我深入理解了文件系统的数据结构如何影响文件的大小、组织和存储。我学会了如何操作不同类型的块来管理文件的数据，并且了解了在文件系统中的不同层次之间如何传递控制和数据。

1. **Symbolic links (moderate)**
   1. **实验目的**

实现符号链接(类似于软链接)这一种特殊类型文件，来用于创建文件路径的引用即指向其他文件或目录的链接，类似于“快捷方式”或“引用”，而不是实际的数据块。

xv6的文件inode结构体中，包括了12个直接块与1个间接块，共能支持12+1\*256=268块（xv6中BSIZE=1024），即文件大小限制为268\*1024bytes.本实验要通过减少1个直接块，增加1个双重间接块来实现支持11+1\*256+256\*256=65803块的文件大小实现。

* 1. **实验步骤**
     1. 增加定义及系统调用等

user/usys.pl

......

entry("symlink");

user/user.h

......

int symlink(const char\*, const char\*);

kernel/syscall.h

......

#define SYS\_symlink 22

kernel/syscall.c

......

extern uint64 sys\_symlink(void);

......

[SYS\_symlink] sys\_symlink,

}

* + 1. 添加新文件标志位

kernel/stat.h

#define T\_SYMLINK 4 // Symbolic Links

kernel/fcntl.h

取值可以自由定义，但注意需要避免与其他标志位出现位运算冲突。

#define O\_NOFOLLOW 0x004 //取值需要避免位运算冲突(lab9-2)

* + 1. 实现sys\_symlink函数

kernel/sysfile.c

首先从寄存器中获取用户传递的两个字符串参数：target 和 path。argstr 函数用于从用户空间复制字符串到内核空间中，并返回字符串的长度。如果参数获取失败，函数返回 -1。

uint64

sys\_symlink(void) {

struct inode \*ip;

char target[MAXPATH], path[MAXPATH];

int n;

if((n = argstr(0, target, MAXPATH)) < 0)

return -1;

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

return -1;

// 参数获取失败

通过 namei 函数，尝试获取 path 对应的目录项（inode）。如果 namei 返回非空指针，则表示要创建的链接已经存在，返回 -1。

if((ip = namei(path)) != 0)

return -1;

// 要创建的链接已存在

在创建链接之前，使用 begin\_op 函数开始一个文件系统操作。然后，调用 create 函数来创建一个新的 inode，类型为 T\_SYMLINK，表示符号链接。如果创建失败，则释放文件系统操作并返回 -1。

begin\_op();

if((ip = create(path, T\_SYMLINK, 0, 0)) == 0) {// 分配并返回inode指针

end\_op();

return -1;

}// 创建失败

接下来，使用 writei 函数将 target 字符串写入创建的 inode。writei 函数将字符串从内存中的 target 写入到 inode 的数据块中。如果写入失败，则释放 inode 并结束文件系统操作。

if(writei(ip, 0, (uint64)target, 0, n) != n) {// target已经在内存中,而不是通过缓冲区传递

iunlockput(ip);//解锁inode并释放对其的引用

end\_op();

return -1;

}

最后，解锁并释放创建的 inode，并使用 end\_op 函数结束文件系统操作。如果整个过程成功完成，函数返回 0，表示符号链接创建成功。

iunlockput(ip);

end\_op();

return 0;

}

* + 1. 修改sys\_open函数以适应新的符号链接文件类型

char path[MAXPATH];

int fd, omode;

struct file \*f;

struct inode \*ip;

int n;

int symlinkDepth = 0;

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

return -1;

begin\_op();

此处为原有代码：如果omode & O\_CREATE为真，表示打开文件时需要创建文件。函数会调用create创建文件，并获取相应的inode。如果创建失败，则会结束文件系统操作并返回-1；若为假，表示打开的是已存在的文件。那么函数会调用namei查找文件名对应的inode。如果文件不存在，则会结束文件系统操作并返回 -1。若存在，函数会锁定该inode。

if(omode & O\_CREATE){... // 创建新文件

} else {// 打开已有文件

if((ip = namei(path)) == 0){

end\_op();

return -1;

}

ilock(ip);

实验新增部分：进入循环，递归向下并记录链接深度：

如果inode的类型是T\_SYMLINK，表示这是一个符号链接。此时，函数会读取符号链接的目标路径，然后通过namei继续查找目标路径对应的inode。如果查找成功，那么函数会再次锁定该 `inode`，然后继续进行后续操作；如果目标路径不存在，或者在跟随符号链接过程中遇到循环引用（递归深度超过了最大深度），则会结束文件系统操作并返回 -1。

如果inode的类型不是 `T\_SYMLINK`，表示打开的是一个普通文件或目录。此时，函数会继续进行后续操作。

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

if (symlinkDepth++ > MAX\_SYMLINK\_DEPTH) {// 记录深度避免死循环

iunlockput(ip);

end\_op();

return -1;

}

if (!readi(ip, 0, (uint64)path, 0, MAXPATH)) {

iunlockput(ip);

end\_op();

return -1;

}

iunlockput(ip);

if((ip = namei(path)) == 0){ // target path 不存在

end\_op();

return -1;

}

ilock(ip);

}

后续原有函数部分：

如果类型为目录且只读，则退出操作并报错。

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

iunlockput(ip);

end\_op();

return -1;

}

}

如果类型为设备文件但设备号小于零或大于等于设备数量，同样退出报错。

if(ip->type == T\_DEVICE && (ip->major < 0 || ip->major >= NDEV)){

iunlockput(ip);

end\_op();

return -1;

}

分配新file结构体与文件描述符。

if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){

if(f)

fileclose(f);

iunlockput(ip);

end\_op();

return -1;

}

如果inode是设备文件，file结构体的type字段会被设置为FD\_DEVICE，并保存设备号major。

如果inode是普通文件，file结构体的type字段会被设置为FD\_INODE，并初始化文件的偏移量off为 0。

然后，函数会设置file结构体的ip字段为找到的inode，并根据omode设置 readable和writable标志位。

如果omode & O\_TRUNC为真，表示打开文件时需要清空文件内容。如果打开的是普通文件，那么函数会调用 `itrunc` 清空文件内容。

最后释放inode的锁，结束文件系统操作，然后返回文件描述符 fd。

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);

}

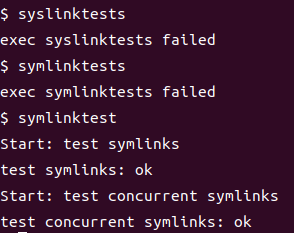
iunlock(ip);

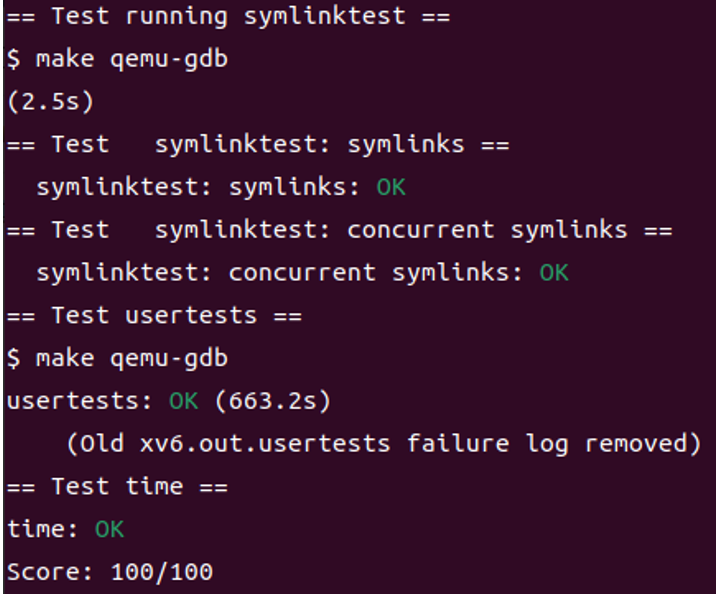
end\_op();

return fd;

};

* + 1. make构建测试及提交





* 1. **实验中遇到的问题和解决方法**

问题：涉及到事务处理(begin\_op()等函数使用时)出错。

解决方法：分析报错信息，查阅他人实验笔记记录的错误寻找相近报错并修改。

问题：同上一个bigfile实验，make qemu中symlinktest正常运行，但make grade中出错。

解决方法：修改grade-lab-fs中的超时限制至足够大(360->3600)

@test(19, "usertests")

def test\_usertests():

r.run\_qemu(shell\_script([

'usertests'

]), timeout=3600)

r.match('^ALL TESTS PASSED$')

* 1. **实验心得**

通过设计符号链接和路径名解析，我深入了解符号链接是如何与路径名解析和文件打开操作交互的，如何处理符号链接，以及如何递归地跟随符号链接链，对文件系统内核的理解更一层。