Lab 9 File System

目录

[1. Large files (moderate) 1](#_Toc174491598)

[1.1 实验目的 1](#_Toc174491599)

[1.2 实验步骤 1](#_Toc174491600)

[1.3 实验结果 6](#_Toc174491601)

[1.4 遇到的问题 8](#_Toc174491602)

[1.5 实验心得 8](#_Toc174491603)

[2. Symbolic links (moderate) 9](#_Toc174491604)

[2.1 实验目的 9](#_Toc174491605)

[2.2 实验步骤 9](#_Toc174491606)

[2.3 实验结果 11](#_Toc174491607)

[2.4 实验中遇到的问题和解决方法 12](#_Toc174491608)

[2.5 实验心得 12](#_Toc174491609)

[3. 测试结果 13](#_Toc174491610)

Large files (moderate)

* 1. 实验目的
* 文件系统索引结构设计：通过实现双重间接块,学习如何设计多级索引结构,扩展文件系统功能
* 映射关系抽象：抽象文件逻辑块号和磁盘块号之间的映射关系,理解不同级别的间接寻址
* 系统扩展性思考：学习考虑文件系统的扩展性,比如索引块的组织方式如何支持更大的文件
* 按需分配：实现按需分配双重间接块及其间接块,减少空间浪费
  1. 实验步骤
* 将直接块的数量-1，改成11，于此同时要调整 dinode 中的 addrs 数据块的数量,以及最大文件数量 MAXFILE

#define NDIRECT 11 // 改为11个直接块

#define NINDIRECT (BSIZE / sizeof(uint))

#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT \* NINDIRECT) // 最大文件大小

// 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];

};

// 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];

};

* 理解和修改 bmap() 函数，将文件的逻辑块号（相对于文件起始位置的块号）映射到实际的物理块号，并在需要时分配新的物理块,并且支持直接块、一级间接块和二级间接块，以实现对更大文件的支持。

// Return the disk block address of the nth block in inode ip.

// If there is no such block, bmap allocates one.

// returns 0 if out of disk space.

static uint bmap(struct inode \*ip, uint bn)

{

// 将逻辑块号映射为物理块号

uint addr, \*a;

struct buf \*bp;

// 直接块处理

if (bn < NDIRECT) {

// 若对应的物理块号不存在，则分配一个新的物理块

if ((addr = ip->addrs[bn]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

return 0;

}

ip->addrs[bn] = addr;

}

return addr;

}

bn -= NDIRECT; // 调整逻辑块号以适应间接块范围

// 一级间接块处理

if (bn < NINDIRECT) {

// 若索引目录不存在，则创建

if ((addr = ip->addrs[NDIRECT]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

return 0;

}

ip->addrs[NDIRECT] = addr;

}

bp = bread(ip->dev, addr); // 读取一级间接块

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

// 找到对应地址，若不存在分配磁盘块

if ((addr = a[bn]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

brelse(bp);

return 0;

}

a[bn] = addr;

log\_write(bp);

}

brelse(bp); // 释放块

return addr;

}

bn -= NINDIRECT; // 调整逻辑块号以适应二级间接块范围

// 二级间接块处理

if (bn < NINDIRECT \* NINDIRECT) {

uint iL1 = bn / NINDIRECT; // 一级间接块索引

uint iL2 = bn % NINDIRECT; // 二级间接块索引

// 若第一级索引目录不存在，则创建

if ((addr = ip->addrs[NDIRECT + 1]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

return 0;

}

ip->addrs[NDIRECT + 1] = addr;

}

bp = bread(ip->dev, addr); // 读取第一级索引目录

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

// 如果对应的二级索引目录不存在，分配一个新的物理块，并将其记录在一级索引目录中

if ((addr = a[iL1]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

brelse(bp);

return 0;

}

a[iL1] = addr;

log\_write(bp);

}

brelse(bp); // 释放一级索引块

bp = bread(ip->dev, addr); // 读取二级索引目录

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

// 如果对应的物理块号不存在，分配一个新的物理块，并将其记录在二级索引目录中

if ((addr = a[iL2]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

brelse(bp);

return 0;

}

a[iL2] = addr;

log\_write(bp);

}

brelse(bp); // 释放二级索引块

return addr;

}

// 超出支持的最大文件大小范围

panic("bmap: out of range");

}

* 修改 itrunc 函数，使其能够结合新的二级索引结构，截断一个 inode 的内容，即释放该inode 所占用的所有磁盘块

// Truncate inode (discard contents).

// Caller must hold ip->lock.

void itrunc(struct inode \*ip)

{

int i, j, k;

struct buf \*bp, \*bpL2;

uint \*a, \*b;

// 释放直接块

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

if (ip->addrs[i]) {

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

ip->addrs[i] = 0;

}

}

// 释放一级索引目录对应的物理块

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;

}

// 释放二级索引目录对应的物理块

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]) {

bpL2 = bread(ip->dev, a[j]);

b = (uint \*)bpL2->data;

for (k = 0; k < NINDIRECT; k++) {

if (b[k])

bfree(ip->dev, b[k]);

}

brelse(bpL2);

bfree(ip->dev, a[j]);

}

}

brelse(bp);

bfree(ip->dev, ip->addrs[NDIRECT + 1]);

ip->addrs[NDIRECT + 1] = 0;

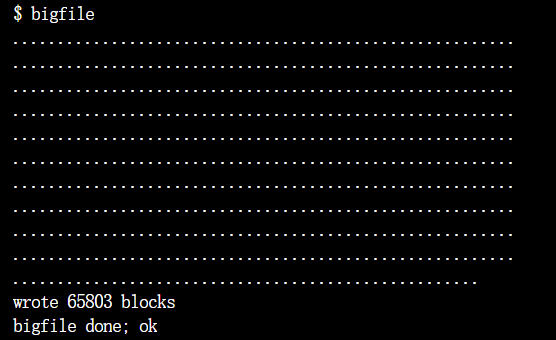
}

ip->size = 0;

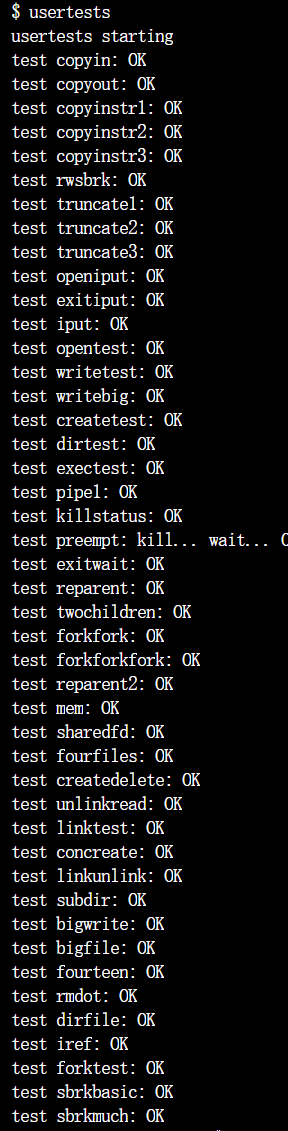
iupdate(ip); // 更新 inode 到磁盘

}

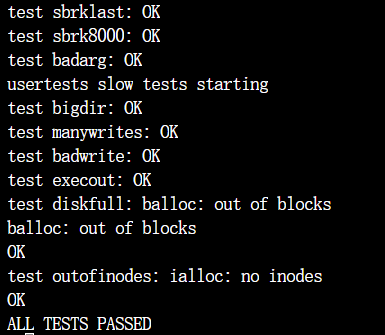
* 1. 实验结果
* 运行 make qemu 后执行 bigfile ：



* 执行usertests ：



·······



* 1. 遇到的问题
* 问题：

在实现新的指针布局时，我出现了指针计算错误、越界访问这些问题，导致文件系统无法正确处理文件块。

* 解决方法：

仔细检查指针计算的逻辑，确保计算正确。对于数组、指针的偏移等操作，要特别注意边界条件。在访问数据块之前，始终进行边界检查，确保不会越界。

* 1. 实验心得

通过实现双重间接块，我深入理解了如何设计多级索引结构以支持更大的文件。直接块、一级间接块和二级间接块的结合，使得文件系统能够灵活地扩展文件的存储能力，同时保持了对存储空间的有效管理。

在实现按需分配双重间接块时，我学会了如何通过减少预分配的方式来减少空间浪费。在 bmap() 函数中，只有在需要时才分配新的物理块，这种按需分配策略有效降低了存储空间的占用，提高了系统资源的利用率。

Symbolic links (moderate)

* 1. 实验目的
* 实现符号链接(symbolic link)功能,学习其机制
* 掌握路径查找(pathname lookup)的实现
* 学习处理符号链接打开(open)的逻辑
* 实现符号链接跟踪(follow)和防止循环依赖
  1. 实验步骤
* 在 kernel/syscall.h 中定义 SYS\_symlink 的系统调用号

#define SYS\_symlink 22

* 在 user/usys.pl 中添加 symlink 的用户接口

entry("symlink");

* 在 user/user.h 中声明 symlink 函数

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

* 在 Makefile 中添加编译. $U/\_symlinktest\
* 在 kernel/syscall.c 中添加 sys\_symlink 的系统调用入口

extern uint64 sys\_symlink(void);

static int (\*syscalls[])(void) = {

// 其他系统调用

[SYS\_symlink] sys\_symlink,

};

* 在 kernel/sysfile.c 中添加 sys\_symlink 的实现

uint64 sys\_symlink(void)

{

char target[MAXPATH]; // 存储符号链接目标路径

char path[MAXPATH]; // 存储符号链接路径

// 从系统调用参数中获取目标路径和符号链接路径

argstr(0, target, MAXPATH);

argstr(1, path, MAXPATH);

// 等待日志系统可用，并确保有足够的日志空间来保存此调用的写入

begin\_op();

// 创建一个新的 inode，并将其类型设为 T\_SYMLINK（符号链接）

struct inode\* ip = create(path, T\_SYMLINK, 0, 0);

// 如果 inode 创建失败，结束日志操作并返回错误码

if (ip == 0) {

end\_op();

return -1;

}

// 将目标路径写入 inode 的数据块中，以存储符号链接指向的目标

if (writei(ip, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {

// 如果写入失败，释放 inode 并结束日志操作，返回错误码

iunlockput(ip);

end\_op();

return -1;

}

// 释放 inode 并解锁

iunlockput(ip);

// 结束日志操作

end\_op();

return 0; // 成功返回 0

}

* 在 kernel/sysfile.c 中修改 open 函数，处理符号链接的解析

uint64 sys\_open(void)

{

// 省略的部分代码...

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

// 如果没有指定 O\_NOFOLLOW 标志，并且文件类型是符号链接 (T\_SYMLINK)

// 那么我们需要递归地解析符号链接，直到找到一个非符号链接的文件或者超出递归深

度。

char path[MAXPATH]; // 用于存储符号链接指向的目标路径

for (int i = 0; i < SYMLINK\_REC\_MAX; i++) {

// 设置递归深度限制为 SYMLINK\_REC\_MAX（定义于 fs.h）

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

// 从 inode 中读取符号链接存储的目标路径，如果读取失败，返回错误

iunlockput(ip);

end\_op();

return -1;

}

iunlockput(ip); // 释放当前的 inode

// 使用 namei 解析路径并获取目标路径的 inode

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

end\_op();

return -1;

}

ilock(ip); // 锁定新的 inode

if (ip->type != T\_SYMLINK) {

// 如果找到的 inode 不是符号链接类型，结束循环，继续正常打开文件

break;

}

}

if (ip->type == T\_SYMLINK) {

// 如果超出递归深度，仍然没有找到非符号链接文件，返回错误

iunlockput(ip);

end\_op();

return -1;

}

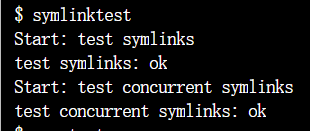
}

// 省略的部分代码...

return fd; // 返回文件描述符

}

* 1. 实验结果
* 运行 make qemu 后执行 symlinktest ：



* 1. 实验中遇到的问题和解决方法
* 问题：

实现符号链接跟踪时，我遇到循环依赖的情况，也就是符号链接链条形成环路，导致无限递归或死锁。

* 解决方法：

实现一个循环检测机制，记录已经访问的inode，当发现重复访问时，停止跟踪并报错。设置递归深度限制，当达到一定深度时停止跟踪，以防止无限递归。在每次递归查询时，注意释放之前持有的inode锁，避免死锁问题。

* 1. 实验心得

通过在 xv6 操作系统中实现符号链接功能，我理解了符号链接作为一种特殊的文件类型，可以将一个文件路径链接到另一个目标文件路径。这个机制在现代操作系统中非常常见，能够为文件系统提供更灵活的路径管理。实验中，我通过定义 SYS\_symlink 系统调用并在内核中实现了 sys\_symlink 函数，成功实现了符号链接的基本功能。

在符号链接的实现过程中，路径查找是关键步骤之一。在 sys\_open 函数中，我学习并实现了如何处理符号链接的解析，特别是如何递归地解析符号链接直到找到实际的目标文件或达到递归深度限制。通过这一过程，我掌握了路径查找的核心思想，并能够处理符号链接路径解析中可能遇到的各种问题。

符号链接解析时，可能会遇到循环依赖的情况，即符号链接形成了一个环。在实验中，我通过设置递归深度限制（SYMLINK\_REC\_MAX），有效地防止了这种循环依赖的发生。这让我理解了在设计文件系统时，如何通过合理的限制来避免死锁和无限循环等问题。

在修改 sys\_open 函数时，我学习了如何处理符号链接文件的打开逻辑。在正常情况下，符号链接应解析到其目标文件并打开目标文件，而不是直接打开符号链接本身。

测试结果

