Lab9 file system

前置知识

实验内容

Large files (moderate)

任务

- 目前，xv6文件系统中的每个inode包含12个“直接”块号和一个“单一间接”块号，后者引用一个块，该块最多可以包含256个块号，总共可以引用12+256=268个块。因此，xv6文件的最大大小限制为268个块。

  你需要修改xv6文件系统代码，支持每个inode中的“双重间接”块，这样文件的最大大小可以增加到65803个块，即256*256+256+11个块（因为我们将牺牲一个直接块号以用于双重间接块）。

理解文件系统结构

- `struct dinode`（定义在`fs.h`中）描述了磁盘上的inode结构。你需要关注`NDIRECT`、`NINDIRECT`、`MAXFILE`以及`struct dinode`中的`addrs[]`元素。
- `bmap()`函数位于`fs.c`中，用于查找文件在磁盘上的数据块。`bmap()`会根据需要分配新块以容纳文件内容，并在需要时分配间接块来存储块地址。

修改`bmap`及`itrunc`函数：

- 你需要在`bmap()`中实现双重间接块（doubly-indirect block），同时保持对直接块和单一间接块的支持。
- 将`ip->addrs[]`的前11个元素保留为直接块，第12个元素用作单一间接块，第13个元素用作新的双重间接块。
- 修改后的`bmap()`应该能够映射文件的逻辑块号（bn）到磁盘块号，并在需要时分配新的块。

```
// 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;
  // 去除已经由直接块和单间接块映射的块数，以得到在双间接块中的相对块号

  if (bn < NDBL_INDIRECT) {
    // 如果文件的双间接块不存在，则分配一个
    if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
      addr = balloc(ip->dev);
      if (addr == 0)
        return 0;
      ip->addrs[NDIRECT + 1] = addr;
    }

    // 读取双间接块
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    // 计算在单间接块数组中的索引，即第几个单间接块
    uint index1 = bn / NINDIRECT;

    // 如果这个单间接块不存在，则分配一个
    if ((addr = a[index1]) == 0) {
      addr = balloc(ip->dev);
      if (addr == 0)
        return 0;
      a[bn / NINDIRECT] = addr;
      log_write(bp);  // Record changes in the log
    }
    brelse(bp);

    // 读取相应的单间接块
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    // 计算在单间接块中的索引，即单间接块中的第几个数据块
    uint index2 = bn % NINDIRECT;

```

```
    // 如果这个数据块不存在，则分配一个
    if ((addr = a[index2]) == 0) {
      addr = balloc(ip->dev);
      if (addr == 0)
        return 0;
      a[bn % NINDIRECT] = addr;
      log_write(bp);  // Record changes in the log
    }
    brelse(bp);
    return addr;  // Returns the actual data block
  }

  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;
```

```
  }

  if (ip->addrs[NDIRECT + 1]) {
    // 读取双间接块
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;

    for (i = 0; i < NINDIRECT; ++i) {
      if (a[i] == 0) continue;

      // 读取单间接块
      struct buf* bp2 = bread(ip->dev, a[i]);
      uint* b = (uint*)bp2->data;
      for (j = 0; j < NINDIRECT; ++j) {
        if (b[j])
          bfree(ip->dev, b[j]); // 释放数据块
      }
      brelse(bp2);

      bfree(ip->dev, a[i]); // 释放单间接块
      a[i] = 0;
    }
    brelse(bp);

    bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放双间接块
    ip->addrs[NDIRECT + 1] = 0;
  }

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

修改`struct inode`

- `kernel/file.h`:

```
//添加到结构体中
uint addrs[NDIRECT+2];
```

Symbolic links (moderate)

任务

- 这个实验的目标是为xv6操作系统增加对符号链接（symbolic links）的支持。符号链接是一种特殊的文件类型，它通过路径名引用另一个文件。当打开符号链接时，内核会跟随链接指向的文件。符号链接类似于硬链接，但硬链接只能指向同一磁盘上的文件，而符号链接可以跨磁盘设备引用文件。虽然xv6不支持多个设备，但实现这个系统调用是理解路径名查找工作原理的好练习。
- 实现`symlink`系统调用：

  任务描述： 你将实现`symlink(char *target, char *path)`系统调用，它在`path`位置创建一个新的符号链接，指向`target`文件。你需要确保在实现过程中符号链接能够正确地处理，尤其是处理路径名查找的递归和循环检测。

步骤

创建新系统调用：

- 为`symlink`创建一个新的系统调用编号，并在`user/usys.pl`和`user/user.h`中添加对应的条目。
- 在`kernel/sysfile.c`中实现一个空的`sys_symlink`函数。

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
```

```
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
//here
int symlink(char*, char*);
```

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("mknod");
entry("unlink");
entry("fstat");
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
//here
entry("symlink");
```

定义符号链接文件类型：

- 在`kernel/stat.h`中添加一个新的文件类型`T_SYMLINK`，用于表示符号链接。

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
#define T_SYMLINK 4   // Soft symbolic link - lab 9.2

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

添加新标志：

- 在`kernel/fcntl.h`中添加一个新的标志`O_NOFOLLOW`，用于`open`系统调用。当指定该标志时，`open`应该打开符号链接本身，而不是跟随符号链接指向的文件。确保该标志不会与现有标志冲突。

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x004    // lab 9.2
```

实现`symlink`系统调用：

- 在`path`位置创建一个新的符号链接，链接到`target`。符号链接的目标路径可以存储在inode的数据块中。
- `symlink`应返回一个整数，表示成功（0）或失败（-1），类似于`link`和`unlink`系统调用。

```
// Create the path new as a link to the same inode as old.
uint64
sys_link(void)
{
  char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
  struct inode *dp, *ip;

```

```
  if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }

  ip->nlink++;
  iupdate(ip);
  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  end_op();

  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return -1;
}
```

修改`open`系统调用：

- 修改`open`系统调用，以处理路径指向符号链接的情况。如果文件不存在，`open`应当失败。
- 如果`open`指定了`O_NOFOLLOW`标志，应当打开符号链接本身，而不是跟随符号链接。
- 如果符号链接指向的文件也是符号链接，必须递归地跟随直到达到非链接文件。如果符号链接形成了循环，应当返回错误代码。你可以通过设置一个递归深度的阈值（例如10）来近似处理这个问题。

```
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;
    }
    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;
  }

  // handle the symlink - lab 9.2
  if(ip->type == T_SYMLINK && (omode & O_NOFOLLOW) == 0) {
    if((ip = follow_symlink(ip)) == 0) {
      // 此处不用调用iunlockput()释放锁
      // follow_symlinktest()返回失败时,锁在函数内已经被释放
      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);
  }
```

```
72
73    iunlock(ip);
74    end_op();
75
76    return fd;
77  }
78
79  // Generating symbolic links
80  uint64
81  sys_symlink(void) {
82    char target[MAXPATH], path[MAXPATH];
83    struct inode *ip;
84    int n;
85
86    if ((n = argstr(0, target, MAXPATH)) < 0
87      || argstr(1, path, MAXPATH) < 0) {
88      return -1;
89    }
90
91    begin_op();
92    // create the symlink's inode
93    if((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
94      end_op();
95      return -1;
96    }
97    // write the target path to the inode
98    if(writei(ip, 0, (uint64)target, 0, n) != n) {
99      iunlockput(ip);
100     end_op();
101     return -1;
102   }
103
104   iunlockput(ip);
105   end_op();
106   return 0;
107 }
```

确保其他系统调用的正确性：

- 修改其他相关的系统调用（如`link`和`unlink`），使它们操作符号链接本身，而不是跟随符号链接指向的文件。

```
// recursively follow the symlinks - lab9-2
// Caller must hold ip->lock
// and when function returned, it holds ip->lock of returned ip
static struct inode*
follow_symlink(struct inode* ip) {
  uint inums[NSYMLINK];
  int i, j;
  char target[MAXPATH];

  for(i = 0; i < NSYMLINK; ++i) {
    inums[i] = ip->inum;
    // read the target path from symlink file
    if(readi(ip, 0, (uint64)target, 0, MAXPATH) <= 0) {
      iunlockput(ip);
      printf("open_symlink: open symlink failed\n");
      return 0;
    }
    iunlockput(ip);

    // get the inode of target path
    if((ip = namei(target)) == 0) {
      printf("open_symlink: path \"%s\" is not exist\n",
target);
      return 0;
    }
    for(j = 0; j <= i; ++j) {
      if(ip->inum == inums[j]) {
        printf("open_symlink: links form a cycle\n");
        return 0;
      }
    }
    ilock(ip);
    if(ip->type != T_SYMLINK) {
      return ip;
    }
  }

  iunlockput(ip);
  printf("open_symlink: the depth of links reaches the
limit\n");
  return 0;
}
```

实验得分