

文件系统

目录

171860588

史文泰

171860588@smail.nju.edu.cn

一	实验目的	----- 2
二	实验分析	
1.	Step1	----- 2
2.	Step2	----- 2
3.	Step3	----- 5
4.	Step4	----- 6
5.	Step5	----- 7
6.	Step6	----- 7
7.	Step7	----- 8
8.	Step8	----- 8
9.	Step9	----- 9
三	实验结果展示	----- 9

一 实验目的

①深入了解 UNIX 文件系统的组成，了解 Inode，SuperBlock，Directory 等概念，复习 open，close 等各类文件操作。

②实现简单的文件系统，学习文件操作的实现方式，深入理解 UNIX 系统对于文件的管理。

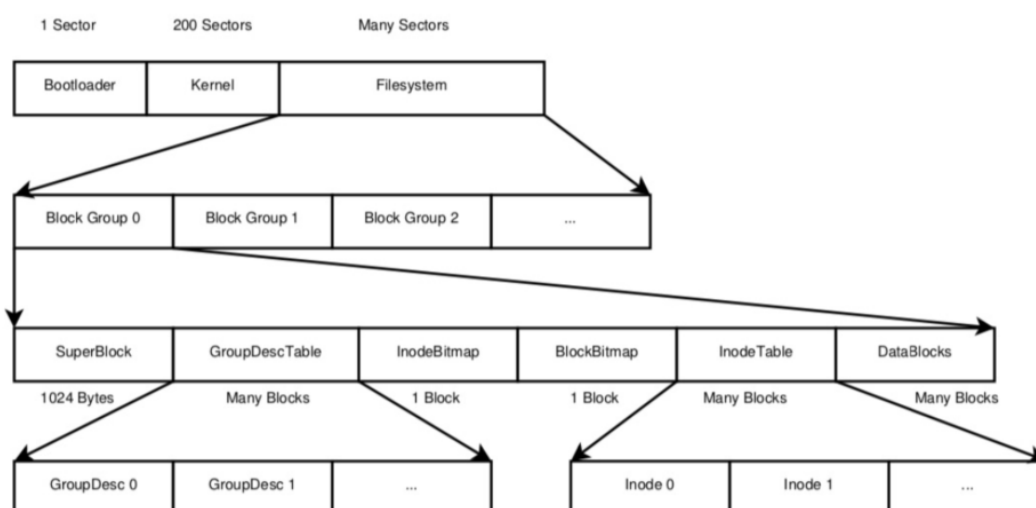
二 实验分析

①Step1

实验第一步，要求按照文件系统的数据结构，构建磁盘文件，即 os.img。

由提示可知所有需要用到的函数均已在 func.c 中给出，其给出了在宿主机上操作镜像文件 fs.bin 的必要接口。同时在 utils.c 中给出了常用的字符串操作函数。

1) 下图为磁盘文件布局：



由上图及相关宏可知，磁盘以扇区为最小单位，每个 Block = 2 个扇区。故首先须以 Block 为单位对磁盘进行初始化，对于 GroupHeader 和 RootDirectory 进行初始化。此为 format() 函数。

2) 下图为我们实现的文件系统目录结构：

```
+/  
|---+boot  
|   |---initrd      #用户态初始化程序  
|   |---...  
|---+dev  
|   |---stdin       #标准输入设备文件  
|   |---stdout      #标准输出设备文件  
|   |---...  
|---+usr  
|   |---...         #用户文件  
|---...
```

接着，根据该目录结构创建 boot，dev，usr 目录和 initrd，stdin，stdout 文件。目录创建调用 mkdir()函数，文件创建调用 touch()函数。同时，需将用户初始化程序 uMain.elf 拷贝到 initrd 中，这调用 cp()函数。实际上创建目录和文件的操作分配了 inode，并在相应父目录中记录了目录项，这些函数的实现为后续的系统调用实现提供了参考。

构建好基本的文件目录结构后，通过 ls 查看目录，结果与样例仅有 uMain.elf 大小不同，如下图：

```
ls /  
Name: boot, Type: 2, LinkCount: 1, BlockCount: 1, Size: 1024.  
Name: dev, Type: 2, LinkCount: 1, BlockCount: 1, Size: 1024.  
Name: usr, Type: 2, LinkCount: 1, BlockCount: 0, Size: 0.  
LS success.  
8185 inodes and 3051 data blocks available.  
ls /boot/  
Name: initrd, Type: 1, LinkCount: 1, BlockCount: 15, Size: 14448.  
LS success.  
8185 inodes and 3051 data blocks available.  
ls /dev/  
Name: stdin, Type: 1, LinkCount: 1, BlockCount: 0, Size: 0.  
Name: stdout, Type: 1, LinkCount: 1, BlockCount: 0, Size: 0.  
LS success.  
8185 inodes and 3051 data blocks available.  
ls /usr/  
LS success.  
8185 inodes and 3051 data blocks available.
```

②Step2

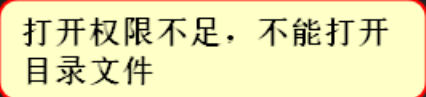
第二步，完成 open 系统调用。框架代码中给出 readInode()函数调用，经过分析可知该函数由参数中给出的路径名读取到需要打开文件对应的 Inode。获得

Inode 后，逻辑如下：

1. 需要打开的文件已经存在，即 `readInode()` 函数返回值为 0。

对于创建文件时，文件的种类可分为普通文件，目录文件和设备文件。若需要打开的文件为目录文件，则在 `open` 时提供的访问控制权限须拥有 `O_DIRECTORY` 权限，否则打开失败，代码如下：

```
if(destInode.type == DIRECTORY_TYPE){ //no directory permission
    if(sflag < 8){
        pcb[current].regs.eax = -1;
        return;
    }
}
```



若为非目录文件，首先判断该文件是否已经被打开。对于设备文件，仅可以打开一次，故若该设备已打开，则结束调用；若未打开，则将其 `state` 置为 1 后结束。

对于普通文件，可多次打开。内核中有 `File` 数据结构，类似于 UNIX 中的系统已打开文件表。每次 `open`，在 `File` 表中寻找一个空闲表项记录本次打开的 Inode 偏移量，指针偏移量和访问控制权限。需要注意的是，由后续框架代码可知，为将设备文件和其他文件区分开来，前 `MAX_DEV_NUM` 个文件为设备文件，后 `MAX_FILE_NUM` 个文件为其他文件，故须返回 `index + MAX_DEV_NUM`。代码如下：

```
//----- create a new file -----
file[i].state = 1;
file[i].inodeOffset = destInodeOffset;
file[i].offset = 0;
file[i].flags = sflag;
//put(file[i].offset);
//-----
pcb[current].regs.eax = i + MAX_DEV_NUM; //plus MAX_DEV_NUM
```



2. 需打开的文件还未存在。此时，首先判断是否具有 `O_CREATE` 权限，若无，则不可以新建文件，打开失败。否则，从空闲 `File` 表项中找出一项分配。若

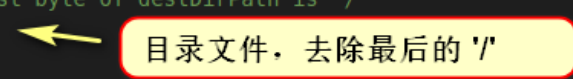
找不到，也失败。

若可以正确分配，则需要新建 Inode 并将 Inode 和名称构成目录项写到父目录中，其中 Inode 为文件 Inode 的偏移量。这部分功能与 touch()函数相似，可对照完成。首先找到父目录的地址，进而找到父目录的 Inode，接着调用 allocInode()函数将目录项写入父目录数据区，并为新的文件分配，初始化 Inode，修改对应的 SuperBlock 等参量。

需要注意的是，touch 仅仅完成了普通文件的创建，而我们仍需要创造目录文件。对于目录文件，需要将路径名最后的 '/' 去除后再查找父目录。部分代码如下：

```
//----- directory -----
length = strlen(path);
if (path[length - 1] == '/') { // last byte of destDirPath is '/'
    *((char*)path + length - 1) = 0;
}
//-----

ret = allocInode(&sBlock, gDesc, // safe operation, none of the pa
&fatherInode, fatherInodeOffset,
destInode, destInodeOffset, path + size + 1, create_type);
```



③Step3

第三步，完成 write()系统调用。对于写入操作，需要考虑写入字符个数超出当前文件长度的情况。逻辑如下：

1. 将当前偏移量所在的一个 Block 优先读出，便于后续以整块的方式进行写入。需要注意，若此时恰好处于文件末尾且文件长度恰好为整数倍个 Block，则须先调用 allocBlock()函数分配一个 Block，并将此空 Block 读出，否则再 readBlock()时会因为 Block 不足而产生未定义结果，表现在可能在编译后随机死循环于某个 readBlock()中。代码如下：

```

if(quotient == inode.blockCount){ //!!!!!!!!!!!!!!!!!!!!!!
    ret = allocBlock(&sBlock, gDesc, &inode, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
    inode.blockCount++;
    if (ret == -1){ // no enough block to allocate
        pcb[current].regs.eax = -1;
        return;
    }
}
ret = readBlock(&sBlock, &inode, quotient, buffer);

```

若恰巧处于文件末尾和Block末尾，则先分配一个新的Block

2. 接着按字节依次将 str 中的内容赋值到 buffer 中，每次赋值一个 Block，并调用 writeBlock()将赋值好的 buffer 写回文件。与此同时改变文件指针偏移量，并在超出文件当前大小后增加文件的大小。当超出文件已分配的 Block 后，须为文件分配新的 Block。循环直到 str 被完全读入。代码如下：

```

for(m = remainder; m < SECTOR_SIZE * SECTORS_PER_BLOCK; ++m){
    buffer[m] = str[index];
    index = index + 1;
    file[sf->ecx - MAX_DEV_NUM].offset++; //change offset
    if(file[sf->ecx - MAX_DEV_NUM].offset > inode.size){ //file bigger
        inode.size = file[sf->ecx - MAX_DEV_NUM].offset; //change filesize
    }
    if(index == size){ //over
        break;
    }
}
ret = writeBlock(&sBlock, &inode, quotient, buffer);

```

3. 最终将修改后的 Inode 信息协会 Inode 文件，并返回成功写入的字节数。

代码如下：

```

diskWrite(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
pcb[current].regs.eax = size;

```

④Step4

第四步，完成 read()系统调用。read 与 write 大体类似，不同之处在于当超出文件当前大小后，应返回最大能够读到的字节数。简要逻辑如下：

1. 将偏移量所在 Block 优先读出，同样考虑恰巧处于 Block 交界处的情况，此时能够读到的最大字节数为 0，应直接返回。

2. 选取文件剩余字节和需要读取字节数中较小的作为实际读取字节数,按字节依次每个 Block 中的内容读到 str 中,并在超出 Block 数或读取完毕后在末尾添加 '\0' 后返回。

⑤Step5

第五步,完成 lseek()系统调用。逻辑如下:

1. 对于普通文件,需要考虑 whence 为 SEEK_SET, SEEK_CUR 和 SEEK_END 的情况。同时,需要注意指针偏移量不可越界,当越界时,将指针指向对应边界。逻辑较为简单,不再赘述。

2. 对于设备文件,并没有文件偏移量,也不支持 lseek()调用,故返回错误。

代码如下:

```
if(index < MAX_DEV_NUM){           //device, no offset
    pcb[current].regs.eax = -1;
    return;
}
```

⑥Step6

第六步,完成 close()系统调用。逻辑如下:

1. 对于设备文件和其他文件一起考虑。查找 Dev 表和 File 表,若当前文件被打开,则将 state 置为 0,将其关闭。若未打开,则关闭失败。代码如下:

```
index = index - MAX_DEV_NUM;
if(file[index].state == 1){
    file[index].state = 0;
    pcb[current].regs.eax = 0;
    return;
}
else{
    pcb[current].regs.eax = -1;
    return;
}
```

文件已打开, 关闭

文件未打开, 失败

⑦Step7

第七步，完成 `remove()` 系统调用。`remove` 调用负责删除给定文件。逻辑如下：

1. 根据 Inode 偏移量查找 Dev 表和 File 表，若此文件正在被使用，则不可删除，返回 -1。代码如下：

```
if(index != -1){           //in use, cannot remove
    pcb[current].regs.eax = -1;
    return;
}
```

2. 若为普通文件或设备文件，需按 `rm()` 函数中找到父目录，删除对应目录项后删除该 Inode，主体函数为 `freelnode()`。

3. 若为目录文件，则需按 `rmdir()` 函数，找到父目录，删除父目录中目录项，并删除本目录下的所有目录项后再删除 Inode，主体函数也为 `freelnode()`，参数为 `DIRECTORY_TYPE`。

⑧Step8

第八步，完成 `ls()` 库函数。在库函数的实现中，需要调用我们之前实现的系统调用。`ls` 函数完成对于目录中目录项名称的显示，仅需按 `O_DIRECTORY | READ` 权限打开目录文件，每次读取一个目录项后显示即可。需要注意的是，读取到的目录项可能为空，则不应显示。代码如下：


```

if(fd < 4){
    return -1;
}
while((ret = read(fd, (uint8_t*)&dirEntry, sizeof(DirEntry))) > 0){
    //printf("hahahah\n");
    if(dirEntry.inode == 0){
        continue;
    }
    printf("%s ", dirEntry.name);
}

```

无用Inode, 不显示

⑨Step9

第九步, 完成 cat()库函数。同样仅需以 O_READ 权限打开文件, 按每个 Block 读取至 buffer 中后输出即可。需要注意, 这两个库函数在返回前均需将文件关闭。代码如下:

```

fd = open(destFilePath, O_READ);
if(fd < 4){
    return -1;
}
while((ret = read(fd, (uint8_t*)(buffer), 1024)) > 0){
    for(int i = 0; i < ret; i++){
        printf("%c", buffer[i]);
    }
}
close(fd);

```

三 实验结果展示

```

ls /
boot dev usr
ls /boot/
initrd
ls /dev/
stdin stdout
ls /usr/

create /usr/test and write alphabets to it
ls /usr/
test
cat /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
rm /usr/test
ls /usr/

rmdir /usr/
ls /
boot dev usr
create /usr/
ls /
boot dev usr

```