

Lab5 文件系统

vigorweijia@foxmail.com

一、代码框架及数据结构

```
wja@ubuntu:~/lab5-171220144XX/lab5-171220144赵四/lab$ tree
.
├── boot
│   ├── main.c
│   └── Makefile
├── boot_loader
│   ├── boot.c
│   ├── boot.h
│   ├── Makefile
│   └── start.S
├── kernel
│   ├── include
│   │   ├── common
│   │   │   ├── assert.h
│   │   │   ├── const.h
│   │   │   ├── types.h
│   │   │   └── utils.h
│   │   ├── common.h
│   │   ├── device
│   │   │   ├── disk.h
│   │   │   ├── keyboard.h
│   │   │   ├── serial.h
│   │   │   ├── timer.h
│   │   │   └── vga.h
│   │   ├── device.h
│   │   ├── fs
│   │   │   ├── ext.h
│   │   │   └── fs.h
│   │   ├── io
│   │   │   ├── cpu.h
│   │   │   ├── io.h
│   │   │   ├── irq.h
│   │   │   └── memory.h
│   │   └── x86.h
│   ├── kernel
│   │   ├── disk.c
│   │   ├── doIrq.S
│   │   ├── fs.c
│   │   ├── i8259.c
│   │   ├── idt.c
│   │   ├── irqHandle.c
│   │   ├── keyboard.c
│   │   ├── kvm.c
│   │   ├── serial.c
│   │   ├── timer.c
│   │   └── vga.c
│   ├── util
│   │   ├── abort.c
│   │   └── utils.c
│   ├── main.c
│   └── Makefile
├── lib
│   ├── lib.h
│   ├── syscall.c
│   ├── types.h
│   ├── utils.c
│   └── utils.h
├── Makefile
└── tools
    ├── genBoot.pl
    ├── genK
    │   ├── data.h
    │   ├── func.c
    │   ├── func.h
    │   ├── main.c
    │   ├── Makefile
    │   ├── types.h
    │   ├── utils.c
    │   └── utils.h
    └── genKernel.pl
```

相较于前面的实验，这个实验多了文件系统的相关操作。

在utils文件夹下面有一个genFS子文件夹，表示生成文件系统。更重要的功能是把原来基本没有区分结构的用户文件、系统文件加以区分结构（通过文件系统）。由实验导读可知，在这部分中需要把uMain.elf文件加载到initrd中。

查询资料得知，对initrd的解读如下：查询资料中对initrd的解读如下：initrd 的英文含义是 boot loader initialized RAM disk，就是由 boot loader 初始化的内存盘。在linux内核启动前，boot loader 会将存储介质中的 initrd 文件加载到内存，内核启动时会在访问真正的根文件系统前先访问该内存中的 initrd 文件系统。

具体执行用户程序的流程还是和从前一样：genFS->bootloader->kMain->uMain。genFS是在编译后生成文件系统。

文件系统中，涉及到一系列繁琐的数据结构：

每个block由两个扇区组成，每个扇区为512B。

超级块（Super Block）占1024字节，记录了文件系统的基本信息：

```
union SuperBlock {
    uint8_t byte[SUPER_BLOCK_SIZE];
    struct {
        int32_t sectorNum; //总扇区数量
        int32_t inodeNum; //总inode数量
        int32_t blockNum; //总数据block数量
        int32_t availInodeNum; //总可用（空闲）inode数量
        int32_t availBlockNum; //总可用数据block数量
        int32_t blockSize; //每个block的大小(1024B)
        int32_t inodesPerGroup; //每组中inode数量
        int32_t blocksPerGroup; //每组中block数量
    };
};
```

Group Descriptor Table用于存放组描述符，每个描述符记录该组中的基本信息：

```

union GroupDesc {
    uint8_t byte[GROUP_DESC_SIZE];
    struct {
        int32_t inodeBitmap; //inode位图 (用于记录空闲inode) ,以扇区为
单位
        int32_t blockBitmap; //block位图
        int32_t inodeTable; //inode区的位置, 以扇区为单位
        int32_t availInodeNum; //可用inode数量
        int32_t availBlockNum; //可用数据block数量
    };
};

```

Inode的结构，本次实验只考虑了文件的基本类型（普通、目录、管道文件等）而没有考虑具体的访存权限：

```

union Inode {
    uint8_t byte[INODE_SIZE]; //用于对齐
    struct {
        int16_t type; //文件类型
        int16_t linkCount; //链接数
        int32_t blockCount; //占用block数量
        int32_t size; //文件实际大小
        int32_t pointer[POINTER_NUM]; //存储位置, 指向某些扇区
        int32_t singlyPointer; //一级索引
        int32_t doublyPointer; //二级索引
        int32_t triplyPointer; //三级索引
    };
};

```

目录文件的结构（每个目录项），目录文件的inode中的pointer记录了目录文件的数据的存储位置（也就是目录文件的表项）：

```
union DirEntry {
    uint8_t byte[DIRENTRY_SIZE];
    struct {
        int32_t inode;//inode号
        char name[NAME_LENGTH]; //文件名
    };
};
```

以上是静态的信息，在OS启动的时候还会添加活动表项。

设备表项：

```
struct Device {
    int state;//设备是否被占用
    int value;
    int inodeOffset;//设备的inode号，因为设备被抽象为了文件
    struct ListHead pcb;//阻塞在设备上的进程
};
```

文件表项，每次实验中没有区分进程打开文件表、系统打开文件表和活动inode表，而是把三个表项合并在一起了：

```
struct File {
    int state;//文件是否被打开
    int inodeOffset;//文件的inode号
    int offset;//文件指针位置
    int flags;//文件读写权限
};
```

二、函数接口解读以及思路

生成文件系统genFS

这部分代码会在编译后，打开qemu之前执行。

stringCpy("fs.bin", driver, NAME_LENGTH - 1);这句代码会把driver的值赋为fs.bin，也就是之后对driver的操作，等同于在对fs.bin进行操作。在经过后续代码的处理后，文件fs.bin可以被视作为完整的文件系统，作为磁盘的一部分，但是其内部数据是空的，还需要mkdir、touch、cp等指令创建或拷贝内容。

format函数的作用是格式化，我的理解就是初始化，填写基本的表项以及清空某些内容。如format的源代码所示，对fs.bin进行文件的写入操作，初始化了超级块、组描述符表（initGroupHeader）以及根目录（initRootDir）等。

所以在对磁盘（fs.bin）进行操作之前，要先格式化。

然后就可以创建一系列文件夹、文件了。

mkdir、touch、cp、ls等操作在执行的时候均需要指定磁盘（driver），后接绝对路径。

系统调用中可用的接口函数

为了方便操作，框架代码中已经实现好了一部分文件系统的相关操作。

读取Inode，需要给定文件路径，成功找到时返回0，失败返回-1

```
int readInode (SuperBlock *superBlock, GroupDesc *groupDesc,
               Inode *destInode, int *inodeOffset, const char
               *destFilePath) {
    //...
    *inodeOffset = groupDesc->inodeTable * SECTOR_SIZE; //
inodeOffset of '/'
    diskRead((void*)destInode, sizeof(Inode), 1, *inodeOffset);
    //从inode表的起始位置开始寻找，根目录一级级开始寻找
    while(destFilePath[count] != 0) { // not empty
        //考虑各种不规范的路径
        for (i = 0; i < blockCount; i++) {
            //读取各级目录文件，比较字符串，一级级寻找
        }
        if (i < blockCount) {
            if (cond == 0)
                count += size + 1;
            else
```

```

        return 0; //found file
    }
    else
        return -1;
}
return 0;
}

```

读写块的操作readBlock/writeBlock，成功读写返回0，失败时返回-1，需要给出文件的inode以及读写它的哪一块

```

int readBlock (SuperBlock *superBlock, Inode *inode, int
blockIndex, uint8_t *buffer) {
    int divider0 = superBlock->blockSize / 4;
    int divider1 = divider0 * divider0;
    int divider2 = divider1 * divider0;
    int bound0 = POINTER_NUM; //直接索引到存储的扇区
    int bound1 = bound0 + divider0; //一级索引界限
    int bound2 = bound1 + divider1; //二级索引界限
    int bound3 = bound2 + divider2; //三级索引界限
    if (blockIndex < bound0) {
        //...
    }
    else if (blockIndex < bound1) {
        //...分级读磁盘
    }
    else if (blockIndex < bound2) {
        //...
    }
    else if (blockIndex < bound3) {
        //...
    }
    else //过大
        return -1;
}

```

分配inode，成功时返回0，失败时返回-1，需要给出父目录的路径，创建类型

```

int allocBlock (SuperBlock *superBlock, GroupDesc *groupDesc,
Inode *inode, int inodeOffset) {
    /*是否够分配*/
    ret = calNeededPointerBlocks(superBlock, inode->blockCount);
    if (superBlock->availBlockNum < ret + 1) return -1; //不够分配
    /*寻找空闲块*/
    getAvailBlock(superBlock, groupDesc, &blockOffset);
    /*分配块*/
    allocLastBlock(superBlock, groupDesc, inode, inodeOffset,
blockOffset);
}

```

释放inode，成功时返回0，失败时返回-1，需要给出父目录的路径

```

int freeBlock (SuperBlock *superBlock, GroupDesc *groupDesc,
Inode *inode, int inodeOffset) {
    int ret = 0;
    while (inode->blockCount != 0) { //是否释放完该文件占用的所有block
        ret = freeLastBlock(superBlock, groupDesc, inode,
inodeOffset); //继续释放
        if (ret == -1) return -1;
    }
}

```

读写磁盘diskRead/diskWrite（在disk.c中）。当知道某个inode的inode号时，也就知道了它在磁盘上的位置，这时直接调用diskRead比调用readInode要方便，因为readInode是在只知道文件路径的情况下访问其inode，涉及到多次磁盘操作。

```

void diskwrite (void *destBuffer, int size, int num, int offset)
{
    int quotient = offset / SECTOR_SIZE; //读写的扇区号
    int remainder = offset % SECTOR_SIZE;
    readSect((void*)buffer, 201 + quotient + j); //由于读写的基本单位
是扇区，所以要先把整个扇区取出来
    while (i < size * num) {
        buffer[(remainder + i) % SECTOR_SIZE] =
((uint8_t*)destBuffer)[i];
        i ++;
    }
}

```

```
if ((remainder + i) % SECTOR_SIZE == 0) { //缓冲区满, 开始写
    磁盘
        writeSect((void*)buffer, 201 + quotient + j);
        j ++;
        readSect((void*)buffer, 201 + quotient + j);
    }
}
writeSect((void*)buffer, 201 + quotient + j);
}
```

文件操作的系统调用

需要实现的系统调用有：打开文件`syscallOpen`、写文件`syscallWriteFile`、读文件`syscallReadFile`、随机访问`syscallLseek`、关闭文件`syscallClose`以及删除文件`syscallRemove`。

`syscallOpen`的思路：需要考虑到文件是否存在。如果存在，要判断访问权限、是否有重复打开等情况。若不存在，要判断是否有创建权限，以及已打开表项是否已满等情况。

`syscallClose`的思路：和打开操作相反，需要考虑到文件是否存在以及文件是否已经被关闭。

`syscallWriteFile`的思路：要考虑到每次写文件的操作是以块为单位的，所以先要取出一个块。要考虑到输入后的大小，若跨越了一个block，需要重新为该文件分配存储空间。还需要考虑缓存的问题，如输入的内容超过一个block，既要重新分配存储空间，又要缓存输入内容，每次写入一个block。

`syscallReadFile`的思路：类似写的操作，要考虑到每次读文件是以块为单位的。还需考虑读入长度过大的问题，即添加读入缓存，每次最多读取一个block。还需要判断实际读入的长度，这一点对于判断是否抵达文件末尾极为重要。

`syscallLseek`的思路：需要考虑到whence的类型，以及文件的大小。

`syscallRemove`的思路：需要考虑文件是否存在。释放inode的时候需要知道被删除文件父目录的inode。

用户层面的函数

用户层面需要实现两个函数，ls和cat，思路较为简单。

ls函数可以直接读取指定的目录文件，然后打印目录项中的文件名或子目录名。

cat函数可以直接读取文件内容。

三、代码编写

genFS较为简单，可以仿照被注释掉的内容进行操作。创建需要的文件夹以及文件，然后把uMain.elf复制到initrd。

syscallOpen:

```
void syscallOpen(struct StackFrame *sf) {
    //...
    // STEP 2
    // TODO: try to complete file open
    //取出父目录路径，需要打开的文件名
    if(ret == 0) { //若文件存在
        if(/*访问权限出错*/) { /*...*/ }
        //检查设备表
        //检查（已打开）文件表项
        //若没问题，分配表项，初始化文件控制表项
    }
    else { //不存在，尝试创建文件
        if(/*若没有创建权限*/) { /*...*/ }
        ret = readInode(/*父目录inode*/);
        if(ret == -1) { /*父目录打开失败*/ }
        if(/*创建普通文件*/) {
            ret = allocInode(/*父目录路径，父目录inode，新文件名，普通文件*/);
        }
        else /*创建目录文件*/ {
            ret = allocInode(/*父目录路径，父目录inode，新文件名，目录文件*/);
        }
        if(ret == -1) { /*创建失败*/ }
        else {
            /*分配相应的表项*/
        }
    }
}
```

```

        /*分配失败*/
    }
}
}

```

syscallClose:

```

// STEP 6
// TODO: try to complete file close
if(/*超过界限*/) {ret = -1;}
else {/*检查表项并清空*/}
if(ret == -1) {/*关闭失败*/}
else {/*关闭文件*/}

```

syscallWriteFile:

```

void syscallWriteFile(struct StackFrame *sf) {
    //...
    // STEP 3
    // TODO: try to complete file write
    if(/*跨越block*/) {/*重新分配*/}
    readBlock(&sBlock, &inode, blockIndex, buffer);/*读入对应的块*/
    for(i = 0; i < size; i++) {
        /*写入缓冲*/
        if(/*跨越block*/) {/*重新分配*/}
        if(/*缓冲区满*/) {/*写入磁盘*/}
    }
    if(/*剩余未写入磁盘内容*/) {
        if(/*跨越block*/) {/*重新分配*/}
        /*写入磁盘*/
    }
    if(ret == -1) {/*写入失败*/}
    else {/*写入成功, 修改offset等*/}
}

```

syscallReadFile:

```

void syscallReadFile(struct StackFrame *sf) {
    //...
    // STEP 4
    // TODO: try to complete file read
    if(/*读入内容超越文件大小*/) { /*...*/}
    ret = readBlock(&sBlock, &inode, blockIndex, buffer); /*读入对应的块*/
    for(i = 0; i < size; i++) {
        if(/*读入内容超越文件大小*/) { /*...*/}
        /*写入缓冲*/
        if(/*缓冲区满*/) { /*写入字符串并读取下一个block*/}
    }
    if(ret == -1) { /*读入失败*/}
    else { /*读入成功*/}
}

```

syscallLseek:

```

void syscallLseek(struct StackFrame *sf) {
    // STEP 5
    // TODO: try to complete seek
    //...
    if(/*SEEK_SET*/) {}
    else if(/*SEEK_CUR*/) {}
    else if(/*SEEK_END*/) {}
    else { /*失败*/}
}

```

syscallRemove:

```

void syscallRemove(struct StackFrame *sf) {
    //...
    // STEP 7
    // TODO: try to complete file remove
    //取出父目录路径, 被删除的文件名
    ret2 = readInode(/*读入父目录inode*/);
    if(ret2 == -1) { /*读入父目录失败*/}
    if(ret == 0) { //文件存在

```

```

        /*检查设备表*/
        /*检查（已打开）文件表*/
        ret2 = freeInode(/*父目录inode, 被删除文件inode, 被删除文件名
*/);

        if(ret2 == 0) { /*成功删除*/}
        else { /*删除失败*/}
    }
    else {} //文件不存在
}

```

关于文件的访问权限，O_READ是读权限，O_WRITE是写权限，O_CREATE是创建权限，O_DIRECTORY表示该文件是目录文件

ls:

```

int ls(char *destFilePath) {
    // STEP 8
    // TODO: ls
    int fd = open(destFilePath, /*访问权限*/);
    while((ret = read(fd, buf, BUF_SIZE)) != 0) { /*打印*/}
    close(fd);
}

```

cat:

```

int cat(char *destFilePath) {
    // STEP 9
    // TODO: cat
    int fd = open(destFilePath, /*访问权限*/);
    while((ret = read(fd, buf, BUF_SIZE)) != 0) { /*打印*/}
    close(fd);
}

```

四、运行结果

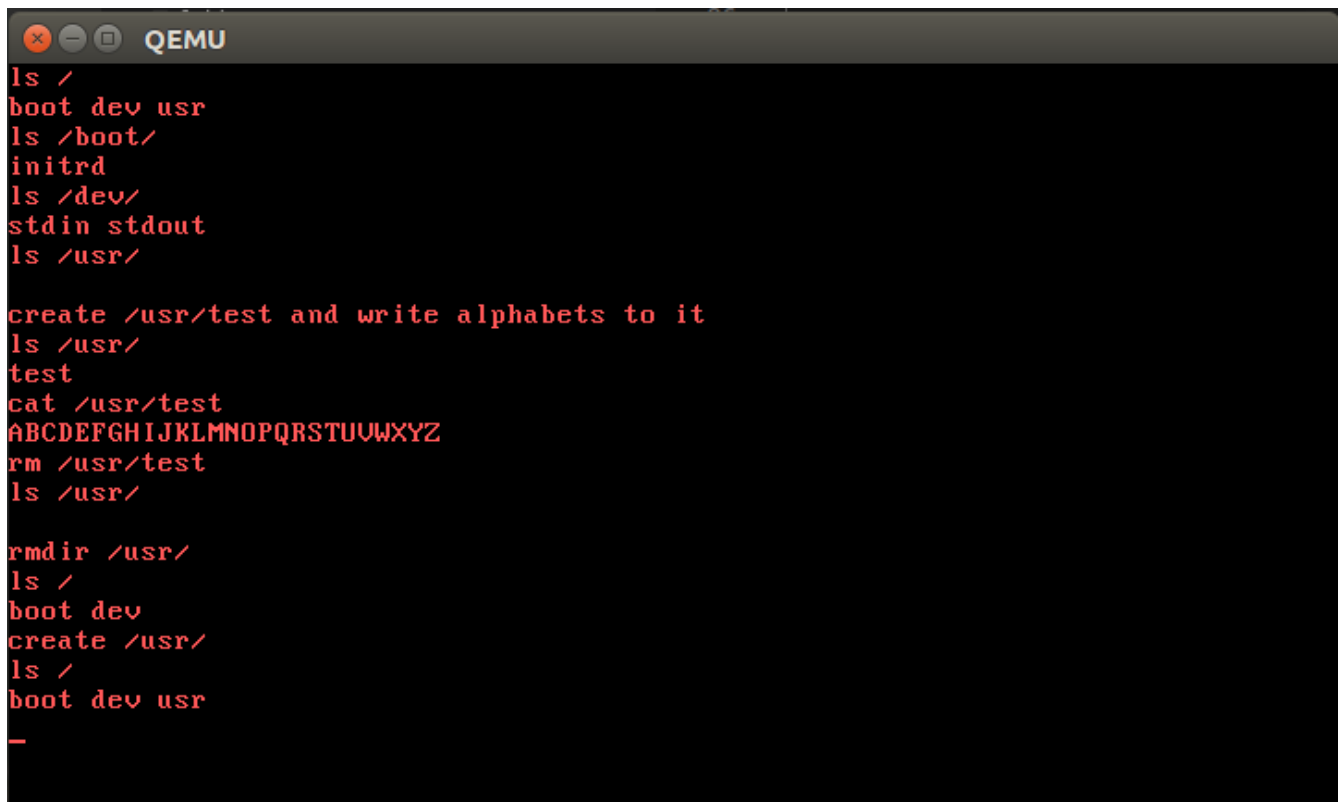
```

gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-pointer -Wall -Werror -O2 -I../lib -c -o ../lib/syscall.o ../lib/sys
gcc -m32 -march=i386 -static -fno-builtin -fno-stack-protector -fno-omit-frame-pointer -Wall -Werror -O2 -I../lib -c -o ../lib/utls.o ../lib/utl
ld -m elf_i386 -e uEntry -Ttext 0x00000000 -o uMain.elf ./main.o ../lib/syscall.o ../lib/utls.o
format fs.bin -s 8196 -b 2
FORMAT success.
8191 inodes and 3070 data blocks available.
Incorrect destination file path.
mkdir /boot
MKDIR success.
8190 inodes and 3069 data blocks available.
mkdir /dev
MKDIR success.
8189 inodes and 3069 data blocks available.
mkdir /usr
MKDIR success.
8188 inodes and 3069 data blocks available.
touch /dev/stdin
TOUCH success.
8187 inodes and 3068 data blocks available.
touch /dev/stdout
TOUCH success.
8186 inodes and 3068 data blocks available.
cp uMain.elf /boot/initrd
CP success.
8185 inodes and 3052 data blocks available.
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 3052 data blocks available.
ls /boot/
Name: initrd, Type: 1, LinkCount: 1, BlockCount: 14, Size: 13356.
LS success.
8185 inodes and 3052 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 3052 data blocks available.
ls /usr/
LS success.
8185 inodes and 3052 data blocks available.
make[1]: Leaving directory '/home/wja/lab5-171220144XX/lab5-171220144赵四/lab/app'
cat bootloader/bootloader.bin kernel/kMain.elf app/fs.bin > os.img

```

在genFS的时候，initrd的大小不是期望的13400，是因为这个文件的大小会随着用户程序而改变（即uMain.elf的代码）。

正常输出：



```

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
create /usr/
ls /
boot dev usr
-

```

加入后续测试：

由于写文件系统的时间比较紧迫，所以还是有许多没有想清楚的地方，好在在考试之前复习了文件系统，对文件系统操作的具体流程有一个比较清晰的认识。

在step1的时候，创建的字符串必须赋值到一个变量上。把uMain.elf复制到initrd的时候，不能先创建initrd文件，因为检查过原代码后发现cp函数并没有覆盖掉文件的功能。

在执行创建文件和删除文件的时候，一定要考虑文件类型，和父目录是否为根目录，不然字符串解析容易出错。才分配、释放inode的时候，需要注意文件名不能为绝对路径。

读写文件的时候，要仔细考虑边界情况，什么时候写入、读入，什么时候重新分配block。

有些部分读取inode可以使用readDisk代替，因为已经知道了inode号。在重新分配完存储空间后，需要用diskWirte写回inode。