

实验五

一、实验目的

实验要求：本实验要求实现一个简单的文件系统

1.1. 格式化程序

编写一个格式化程序，按照文件系统的数据结构，构建磁盘文件（即 os.img）

1.2. 内核

内核初始化时，按照文件系统的数据结构，读取磁盘，加载用户程序，并对 OPEN、READ、WRITE、LSEEK、CLOSE、REMOVE 这些系统调用提供相应接口

1.3. 用户程序

基于 OPEN、READ 等系统调用实现 LS、CAT 这两个函数，并使用以下用户程序测试

二、实验过程

1.1

首先格式化程序，我们发现生成 os.img 是在 Makefile 中实现：

```
QEMU = qemu-system-i386

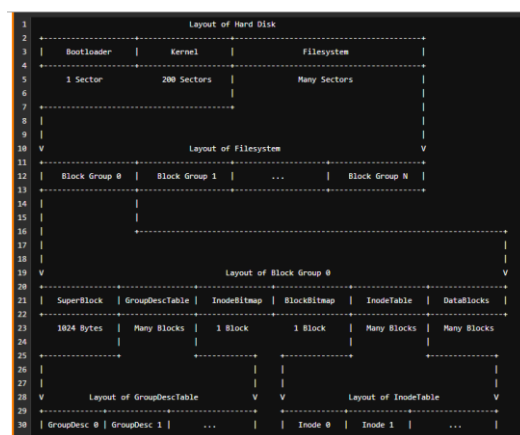
os.img:
    @cd bootloader; make
    @cd kernel; make
    @cd app; make
    cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
```

那么我们要对磁盘进行格式化，生成达到目标的 os.img 就必须在 Makefile 中调用格式化程序，这里我们通过在 lab5/utils 目录下编写 Disk.c 程序实现格式化生成 os.img 的功能，所以 Makefile 文件应修改为以下内容：

```
os.img:
    @cd bootloader; make
    @cd kernel; make
    @cd app; make
    @gcc utils/Disk.c -o utils/Disk
    @utils/Disk bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf os.
img
    @#cat bootloader/bootloader.bin kernel/kMain.elf app/uMain.elf > os.img
```

然后我们来看编写的 Disk.c 文件；

根据要求，我们实现的磁盘结构应如以下：



而根据我的设计，简化实现，我将磁盘设为 1024 个扇区大小，Group 是设置为一个，所以 superblock 占一个块（两扇区），GroupDescTable 为一个扇区，InodeBitmap 、

BlockBitmap 均占一个块（两扇区），InodeTable 因为一个 Inode 节点我定义为 128 字节，所以共 256 个 Inode 节点，共 64 个扇区，余下均为数据块区。

具体数据结构参数如下：（格式化程序与内核中相同）结构已有教程中所给

```
#define SECTOR_SIZE 512
#define BLOCKSIZE 1024
#define SUPER_BLOCK_SIZE 1024
#define GROUP_DESC_SIZE 512
#define INODE_SIZE 128
#define POINTER_NUM 26
#define DIRENTRY_SIZE 64
#define DIRENT_NAME_LENGTH 60
```

而每个目录项为 64 字节。

那么在此基础上，我们开始根据输入的 bootloaderfile, kernelfile 和 appfile 进行处理，使用 c 标准库中的文件读写函数，得到每个文件的内容和长度，将 BootLoader 和 kernel 分别复制到第 0 号扇区和 1~200 号扇区。

随后对磁盘进行格式化处理。使用 format 函数。对超级块、inodeBitmap、blockBitmap、GroupDesc、索引节点表进行初始化。而且注意此时创建了主目录文件/，占一个文件索引节点，底下有两个默认目录‘.’和‘.’，作为每个目录创建时必备，该文件指向数据块区第一个块序号 272 号扇区。

```
void format() {
    superbuffer.sectorNum = 1024;
    superbuffer.inodeNum = 256;
    superbuffer.blockNum = 512;
    superbuffer.availInodeNum = 256;
    superbuffer.availBlockNum = 512;
    superbuffer.blockSize = 1024;
    superbuffer.inodesPerGroup = 256;
    superbuffer.blocksPerGroup = 1024;

    groupbuffer.inodeBitmap = 204;
    groupbuffer.blockBitmap = 206;
    groupbuffer.inodeTable = 208;
    groupbuffer.blockTable = 272;
    groupbuffer.availInodeNum = 256;
    groupbuffer.availBlockNum = 512;

    memset(inodeBitmap, 0, BLOCKSIZE);
    memset(blockBitmap, 0, BLOCKSIZE);
}
```

在之后即可使用 fs_mkdir 创建目录，在这里有三个目录创建，首先调用 mkdir 函数，然后在其中调用 findInodenum 找到文件的路径对应的 inode 节点，如果没有该文件的路径，就进行创建，inode 即为创建的对应的节点。在主目录文件的 inode 节点下对其目录文件进行改动，增加一个目录项。然后找到创建的 inode，调用 allocinode 函数，创建节点，然后找到对应索引节点，调用 allocblock 函数，找到文件对应的数据块区，最后进行创建目录文件，创建时有两个（默认）目录项‘.’和‘.’。对磁盘进行写后，即完成目录创建。下为部分代码截图：

```

int fs_mkdir(const char *path) {
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t DirentBuf[BLOCKSIZE];
    int32_t num = 0;
    union Inode *inode0 = NULL;
    union DirEntry *dirent0 = NULL;
    int direntnum;
    int32_t newinode;

    char file_name[256];
    num = findinodenum(path, file_name);

    readSect(InodeBuf, 208 + num / 4, 2);
    inode0 = ((union Inode *)InodeBuf) + num % 4;
    direntnum = inode0->size;

    readSect(DirentBuf, inode0->pointer[0], 2);
    dirent0 = (union DirEntry *)DirentBuf;
    dirent0 += direntnum;
}

```

随后创建文件，调用 fs_create_file 函数，使用 str 函数库函数，对 path 进行处理，找到对应的上层目录，然后通过其 inode 节点找到其目录文件，然后找其下一层目录，直到下一层目录中没有目标文件，那么就进行创建文件。此处用多层循环实现。

```

int fs_create_file(const char *path) {
    char aimpath[256];
    strcpy(aimpath, path + 1);
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t DirentBuf[BLOCKSIZE];
    int32_t num = 0;
    union Inode *inode0 = NULL;
    union DirEntry *dirent0 = NULL;
    int i = 0, j = 0;
    int direntnum;
    int flag = 0;
    char *s = strtok(aimpath, "/");
    int newinode;
    while (s != NULL)
    {
        readSect(InodeBuf, 208 + num / 4, 2);
        inode0 = ((union Inode *)InodeBuf) + num % 4;
        direntnum = inode0->size;
    }
}

```

然后使用 fs_write 函数进行对新创建的文件写操作。同样调用 findinodenum 函数找到目标文件的 inode 节点，然后比较其原有 offset 与要写的长度之和，防止超过文件最大长度，然后对文件进行磁盘写操作，返回写的长度。

```

size_t fs_write(const char *path, uint8_t *buf, size_t count, int offset)
{
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t BlockBuf[BLOCKSIZE];
    int32_t num = 0;
    union Inode *inode0 = NULL;
    union DirEntry *dirent0 = NULL;
    int index;
    int blocknum0, blocknum1;
    uint8_t *buf1 = buf;

    char file_name[256];
    num = findinodenum(path, file_name);
    readSect(InodeBuf, 208 + num / 4, 2);
    inode0 = ((union Inode *)InodeBuf) + num % 4;

    blocknum0 = (inode0->size - 1) / SECTOR_SIZE + 1;
    blocknum1 = (offset + count - 1) / SECTOR_SIZE + 1;
    int i;

```

最后调用 save 函数进行保存。

```

void save(char *path) {
    FILE *fd = fopen(path, "wb");
    fwrite(disk, sizeof(disk), 1, fd);
    fclose(fd);
}

```

即为以上所有操作。

```

//printf("app = %d\n", appsize);
memcpy(disk, bootloaderBuf, SECTOR_SIZE);
memcpy(disk + SECTOR_SIZE, kernelBuf, 200 * SECTOR_SIZE);
format();
fs_mkdir("/sbin");
fs_mkdir("/dev");
fs_mkdir("/usr");
fs_create_file("/usr/app");
fs_write("/usr/app", appBuf, appsize, 0);
save(aimfile);

```

完成格式化后，我们开始要将用户文件装载至内核，与之前不同，我们要将其从文件系统中读取，使用 fs_size 首先读取其长度，然后使用 fs_read 函数进行读取，最后与之前相同，使用 elf 来装载。

```

}"/
int appsize = 0;
appsize = fs_size("/usr/app");
//Log("appsize = %d\n", appsize);
int inode = findinodenum("/usr/app", NULL);
//Log("appinode = %d\n", inode);
fs_read(inode, tmp, appsize, 0);
//Log("readsize = %d\n", readsize);

struct ELFHeader * ELFHDR = ((struct ELFHeader *)tmp);
struct ProgramHeader *ph, *eph;

```

最后实现系统调用，此次系统调用很多。


```
void sys_cat(struct TrapFrame *tf) {
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    fs_cat((const char *)(tf->ecx));
}
```

fs_cat 函数，就是找到对应文件节点，然后，fs_read 函数读取，然后 Log 输出。

```
void fs_cat(const char *path) {
    uint8_t InodeBuf[BLOCKSIZE];
    int32_t num = 0;
    union Inode *inode0 = NULL;
    int size;
    int i;
    char file_name[128];
    num = findinodenum(path, file_name);
    readSect(InodeBuf, 208 + num / 4);
    readSect(InodeBuf + 512, 208 + num / 4 + 1);
    inode0 = ((union Inode *)InodeBuf) + num % 4;
    size = inode0->size;
    fs_read(findinodenum(path, NULL), InodeBuf, inode0->size, 0);
    for (i = 0; i < size; i++) {
        Log("%c", InodeBuf[i]);
    }
}
```

Open 即调用 fs_create 进行文件创建

```
void sys_open(struct TrapFrame *tf) {
    int num;
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    num = createFCB();
    //Log("FCB num = %d", num);
    FCBlist[num].offset = 0;
    FCBlist[num].inode = fs_create((const char *)(tf->ecx));
    //Log("FCB inode = %d", FCBlist[num].inode);
    tf->eax = num;
}
```

Open 调用 fs_create 进行文件创建，与格式化程序中 create_file 相类似。

```
int fs_create(const char *path) {
    char aimpath[128];
    strcpy(aimpath, path + 1);
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t DirentBuf[BLOCKSIZE];
    int32_t num = 0;
    union Inode *inode0 = NULL;
    union DirEntry *dirent0 = NULL;
    int i = 0, j = 0;
    int direntnum;
    int flag = 0;
    char *s = strtok(aimpath, "/");
    int newinode;
```

Write 即调用 fs_write 进行操作，与格式化程序中 write 相类似


```

void sys_fswrite(struct TrapFrame *tf) {
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    uint8_t *buf = (uint8_t *) (tf->ecx);
    int size = tf->edx;
    fs_write(FCBlist[tf->ebx].inode, buf, size, FCBlist[tf->ebx].offset);
    FCBlist[tf->ebx].offset += size;
    //Log("offset = %d\n", FCBlist[tf->ebx].offset);
    tf->eax = size;
}

```

以下为 write 和 read 内容，较简单相似

```

int fs_read(int32_t inode, uint8_t *buf, size_t count, int offset) {
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t BlockBuf[BLOCKSIZE];
    int32_t num = inode;
    union Inode *inode0 = NULL;
    int index;
    uint8_t *buf1 = buf;

    readSect(InodeBuf, 208 + num / 4);
    readSect(InodeBuf + 512, 208 + num / 4 + 1);
    inode0 = ((union Inode *) InodeBuf) + num % 4;

    index = offset / SECTOR_SIZE;
    offset = offset % SECTOR_SIZE;
    while (count > 0)

```

```

size_t fs_write(int32_t inode, uint8_t *buf, size_t count, int offset)
{
    uint8_t InodeBuf[BLOCKSIZE];
    uint8_t BlockBuf[BLOCKSIZE];
    int32_t num = inode;
    union Inode *inode0 = NULL;
    int index;
    int blocknum0, blocknum1;
    uint8_t *buf1 = buf;

    readSect(InodeBuf, 208 + num / 4);
    readSect(InodeBuf + 512, 208 + num / 4 + 1);
    inode0 = ((union Inode *) InodeBuf) + num % 4;

    blocknum0 = (inode0->size - 1) / SECTOR_SIZE + 1;
    blocknum1 = (offset + count - 1) / SECTOR_SIZE + 1;
    int i;

```

完成以上所有内容，我们对于输出方式，写了个 Log 函数进行输出。实际与 printf 相同。

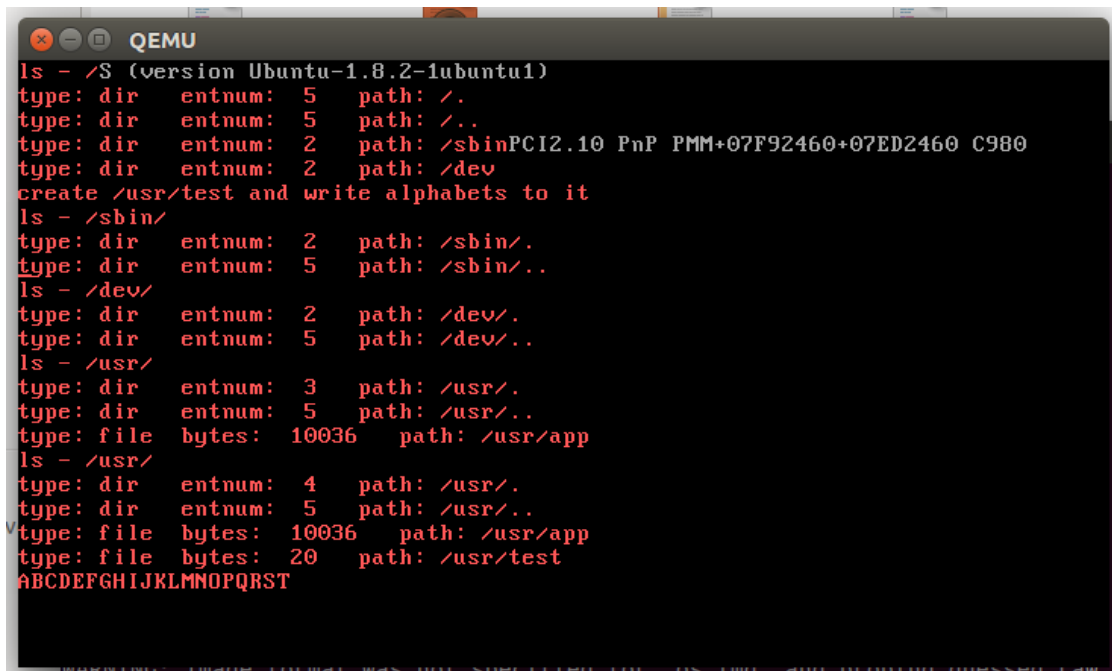
```

void Log(char *format, ...)
{
    va_list ap;
    va_start(ap, format);
    while (*format)
    {
        if (*format != '%')
        {
            putwrite(format, 1);
            format++;
        }
        else
        {

```

三、实验结果

运行结果如下：（当然由于我设置的写文件函数中限制了每次读写的文件长度，所以发现并不能完整的实现 $l = 1024$ 时的输出，所以我将 `app/main.c` 中的 $l = 1024$ 改为了 20，因此输出结果如下）



```
ls - /S (version Ubuntu-1.8.2-1ubuntu1)
type: dir   entnum: 5   path: /.
type: dir   entnum: 5   path: /..
type: dir   entnum: 2   path: /sbinPCI2.10 PnP PMM+07F92460+07ED2460 C980
type: dir   entnum: 2   path: /dev
create /usr/test and write alphabets to it
ls - /sbin/
type: dir   entnum: 2   path: /sbin/.
type: dir   entnum: 5   path: /sbin/..
ls - /dev/
type: dir   entnum: 2   path: /dev/.
type: dir   entnum: 5   path: /dev/..
ls - /usr/
type: dir   entnum: 3   path: /usr/.
type: dir   entnum: 5   path: /usr/..
type: file  bytes: 10036 path: /usr/app
ls - /usr/
type: dir   entnum: 4   path: /usr/.
type: dir   entnum: 5   path: /usr/..
type: file  bytes: 10036 path: /usr/app
type: file  bytes: 20   path: /usr/test
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

时间限制，还有一些不足，但真的尽力了。感谢老师和助教一学期的帮助，终于完成了。开心。