lab5-1718405l4张福翔

邮箱：499480213@qq.com

写在前面（验收相关）

由于本实验的代码量较大以及完成时间匆忙，本人遇到的部分情况说明如下：

- 由于本实验的`gcc`编译参数使用了较为严格的警告等级，而框架代码直接编译会出现`Unused Variable`的警告，而警告直接作为了错误输出。影响到了实验的测试，因此我去掉了`kernel/Makefile`的`gcc`编译条件中的`-Werror -O2`这两个条件，使得框架代码可以直接编译。**该修改仅是因为方便进行调试，本人没有对`Makefile`的其他部分和其他的`Makefile`进行修改。**
- 实际测试过程中发现使用`make clean`后第一次进行`make play`是正常的，但多次进行`make play`后续的`make play`会失效。
- 主函数测试用例中直接使用`ls("/")`有时候会出现未知错误，因此将其暂用一个字符串存储，即将其转化为了`char root[] = "/";`和`ls(root);`两个语句，对测试输出不产生影响。

最后感谢助教抽出时间验收选做实验qwq

1. 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

```
#include "types.h"
#include "lib.h"

int uEntry(void) {
    int fd = 0;
    int i = 0;
    char tmp = 0;

    ls("/");                                    // 列出"/"目录下的所有文件
    ls("/boot/");                               // 列出"/boot/"目录下的所有文件
```

```
    ls("/dev/");                                // 列出"/dev/"目录下的所有文件
    ls("/usr/");                                // 列出"/usr/"目录下的所有文件

    printf("create /usr/test and write alphabets to it\n");
    fd = open("/usr/test", O_RDWR | O_CREAT);  // 创建文件"/usr/test"
    for (i = 0; i < 512; i ++) {                // 向"/usr/test"文件中写入字母表
        tmp = (char)(i % 26 + 'A');
        write(fd, (uint8_t*)&tmp, 1);
    }
    close(fd);
    ls("/usr/");                                // 列出"/usr/"目录下的所有文件
    cat("/usr/test");                           // 在终端中打印"/usr/test"文件的内容

    exit();
    return 0;
}
```

2. 解决思路

2.1 步骤1：搭建镜像 `os.img` 的文件系统

我们使用 `utils/genFS` 中的辅助函数来帮助我们搭建起文件系统模型，我们需要的最终文件系统框架如下：

文件系统的目录结构如下所示，本次实验要求编写格式化程序对其进行构建

```
+/
|---+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
|---...
```

其中 `/boot/initrd` 为用户态初始化程序（对应前4次实验的 `uMain.elf` ）。

已经提供好的辅助函数包括 `mkdir` ， `cp` ， `touch` 等等，分别用来创建一个文件夹，复制一个文件和创建一个新的空文件。我们需要在根目录下创建三个文件夹 `boot` ， `dev` 和 `usr` ，将文件 `app/uMain.elf` 复制到我们文件系统中成为 `/boot/initrd` ，并在 `/dev/` 文件夹下创建两个新设备文件 `stdin` 和 `stdout` ，如果采用 `shell` 命令式的写法，我们可以这样描述：

```
mkdir boot
cp uMain.elf /boot/initrd
mkdir dev
touch stdin
touch stdout
mkdir usr
```

由于辅助函数的参数形式与这些命令相似，我们在此不具体列出代码，需要注意的是我们应当用提供好的字符串处理函数（如 `stringCpy`）来处理上述路径名称。

2.2 步骤2：系统调用 open

系统调用 `open` 用于打开文件目录中的一个文件，并建立一个对应的系统已打开文件表项，函数原型如下：

```
int open(char *path, int flags);
```

两个参数分别为文件路径以及打开标志（我们的实验中只考虑 `O_CREATE`，`O_DIRECTORY`，`O_WRITE`，`O_READ` 这几个标志），`O_CREATE` 表示创建一个新文件，`O_DIRECTORY` 表示打开的是一个目录文件，`O_WRITE` 与 `O_READ` 分别对应读写权限。

根据 `open` 的打开参数不同，我们可以先分成两类考虑，一类是创建新文件的，另一类是打开原有文件的。因此，我们先根据路径查找对应的 `iNode`，如果 `iNode` 查找成功，说明该文件已存在，否则，该文件不存在，我们就需要创建一个这样的文件。而创建文件需要找到其父目录，我们可以使用辅助函数 `stringChrR` 来找到其最后一个 `/` 的分隔符，需要注意的是目录路径可能以 `/` 结尾，因此我们需要在使用之前删除末尾的 `/` 符号。

```
if (mode & O_DIRECTORY)
{
    if (str[stringLen(str) - 1] == '/')
        str[stringLen(str) - 1] = 0;
}
ret = stringChrR(str, '/', &size);
```

分隔最后一个 `/` 号后我们可以提取出该文件的父目录：

```
char fatherPath[NAME_LENGTH];
if (size == 0)
{
    fatherPath[0] = '/';
    fatherPath[1] = 0;
}
else
    stringCpy(str, fatherPath, size);
```

由于 `stringChrR` 的 `size` 返回最后一个 `/` 前面的字符数，如果最后一个 `/` 即为根目录，则返回的 `size` 为 `0`，这不符合我们的需求，因此我们应当手动将其改为根目录。完成父目录路径获取后，我们根据需要创建的文件类型，调用函数 `allocInode` 为父目录创建对应文件名的文件并分配 `iNode`。

```
int type = 0;
if (mode & O_DIRECTORY)
    type = DIRECTORY_TYPE;
else
    type = REGULAR_TYPE;
ret = allocInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
&destInodeOffset, str + size + 1, type);
```

由此我们完成了关于文件系统方面的问题，下面需要做的就是根据对应的`iNode`建立起一项对应的系统已打开文件表项。我们给定的系统已打开文件表项数据结构如下：

```
struct File { // file
    int state;
    int inodeOffset; //XXX inodeOffset in filesystem, for syscall open
    int offset; //XXX offset from SEEK_SET
    int flags;
};
```

其中`state`字段表示该表项是否生效，`inodeoffset`字段表示对应`iNode`的偏移量，`offset`表示当前文件指针的位置，而`flags`即为文件的打开标志位。我们先遍历一遍打开文件表`file`，找到其中处于空闲的表项，并正确设置其相关字段即可：

```
for (i = 0; i < MAX_FILE_NUM; i++)
    if (file[i].state == 0)
        break;
if (i == MAX_FILE_NUM)
{
    sf->eax = -1;
    return;
}

file[i].state = 1;
file[i].inodeOffset = destInodeOffset;
file[i].offset = 0;
file[i].flags = mode;
```

最后我们向用户进程返回文件描述符，由于设备号占用了`4`个文件描述符，因此普通打开文件的描述符需要在系统已打开文件表项索引的基础上加上`4`的大小。

```
sf->eax = i + MAX_DEV_NUM;
return;
```

返回值即为对应打开表项的文件描述符。

2.3 步骤3：系统调用`write`

系统调用`write`用于从当前文件指针开始写入字节流，其原型如下：

```
int write(int fd, void *buffer, int size);
```

其中 `fd` 为文件描述符，`buffer` 为需要写入的字节流，`size` 为写入字节流长度。我们需要首先对 `fd` 文件描述符的合法性进行检查，包括该文件表项是否打开、是否允许写模式访问等等。

随后，我们从文件表项中获取到对应的 `iNode` 偏移量，由此获取 `iNode`，判断当前偏移量处于的文件物理块索引：

```
int quotient = file[sf->ecx - MAX_DEV_NUM].offset / sBlock.blockSize;
int remainder = file[sf->ecx - MAX_DEV_NUM].offset % sBlock.blockSize;
Inode inode;
diskRead(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
```

从 `iNode` 中我们可以获取很多有用的信息，我们通过循环进行写入过程，其流程大致如下：

- 先读入当前需要写入块的数据，存储在 `buffer` 里。
- 如果当前写入块超出了该文件拥有的总块数，说明我们需要为该文件新分配块进行写入，这可以通过函数 `allockBlock` 进行，如果分配失败，说明文件系统已经没有空闲块了，写入到此为止，返回已写入的字节数。
- 计算该物理块剩余可写入字节数 `wbytes`，如果其不小于剩余需写入字节数 `left`，则说明剩余的字节只分布在当前块中，我们写完剩下的字节，修改文件指针，将当前物理块写回磁盘，判断文件大小比写入前是否有所增加，并据此修改 `inode.size` 即文件大小，返回写入的字节数。
- 如果 `wbytes` 比剩余需写入字节数 `left` 小，则说明该块不足以存储所有应当写入的数据，我们先写完当前物理块，修改文件指针，将当前物理块写回磁盘，然后读取下一个物理块继续循环。

```
while (left)
{
    if (cur > inode.blockCount) // write to a new block
    {
        ret = allocBlock(&sBlock, gDesc, &inode, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
        if (ret < 0) // no more free block
        {
            inode.size = old_off + size - left > inode.size ? old_off + size - left :
inode.size;
            sf->eax = size - left; // bytes have written
            diskWrite(&inode, sizeof(Inode), 1, file[sf->ecx -
MAX_DEV_NUM].inodeOffset);
            return;
        }
    }
    ret = readBlock(&sBlock, &inode, cur, buffer);
    int wbytes = sBlock.blockSize - start;
    if (wbytes >= left)
    {
        for (int k = 0; k < left; k++)
        {
            buffer[j] = str[i++];
            j++;
        }
```

```
            writeBlock(&sBlock, &inode, cur, buffer);
            inode.size = (old_off + size > inode.size) ? (old_off + size) : inode.size;
            file[sf->ecx - MAX_DEV_NUM].offset += left;
            left = 0;
            sf->eax = size;
            diskWrite(&inode, sizeof(Inode), 1, file[sf->ecx - MAX_DEV_NUM].inodeOffset);
            return;
        }
        else
        {
            for (int k = 0; k < left; k++)
            {
                buffer[j] = str[i++];
                j++;
            }
            writeBlock(&sBlock, &inode, cur, buffer);
            file[sf->ecx - MAX_DEV_NUM].offset += wbytes;
            left -= wbytes;
            cur++; // block index ++
            start = 0; // start from a new block
            j = start;
            readBlock(&sBlock, &inode, cur, buffer);
        }
    }
```

仅当文件系统中没有空闲块可供分配时，我们无法写入指定大小的数据，否则，返回值应当都与给定的参数`size`一致。

2.4 步骤4：系统调用`read`

系统调用`read`用于从文件指针位置读入一定长度的字节流，其原型如下：

```
int read(int fd, void *buffer, int size);
```

`read`函数需要对文件描述符采用与`write`类似的合法性检查过程，在此不多做赘述。我们再读取对应的`iNode`，同样计算出文件指针位置对应的物理块，然后采取循环方式读取：

- 如果当前指针位置已经大于等于文件大小，返回值`-1`表明已经读取到文件结尾。否则，根据文件指针读取对应索引的块，存到缓存`buffer`里。
- 如果剩余需要读取字节数`left`超过了文件大小，读取到文件大小为止的字节，修改文件指针，并返回读取的字节数。
- 如果当前块可读字节数`rbytes`大于等于剩余需读字节数`left`，则读完当前块的`left`长度字节即可结束，修改文件指针并返回读取的字节数。
- 否则，我们读取完当前块的内容，修改文件指针，然后继续读取下一个物理块内容，重复循环。

```
while (left)
{
    int rbytes = sBlock.blockSize - start;
    if (rbytes + file[sf->ecx - MAX_DEV_NUM].offset >= inode.size)
```

```
        {
            for (int k = 0; k < inode.size - file[sf->ecx - MAX_DEV_NUM].offset; k++)
            {
                str[i] = buffer[j++];
                i++;
            }
            file[sf->ecx - MAX_DEV_NUM].offset = inode.size;
            sf->eax = size - left + inode.size - file[sf->ecx - MAX_DEV_NUM].offset;
            left = 0;
            return;
        }
        if (rbytes >= left)
        {
            for (int k = 0; k < left; k++)
            {
                str[i] = buffer[j++];
                i++;
            }
            file[sf->ecx - MAX_DEV_NUM].offset += left;
            left = 0;
            sf->eax = size;
            return;
        }
        else
        {
            for (int k = 0; k < rbytes; k++)
            {
                str[i] = buffer[j++];
                i++;
            }
            file[sf->ecx - MAX_DEV_NUM].offset += rbytes;
            left -= rbytes;
            cur++;
            start = 0;
            j = start;
            readBlock(&sBlock, &inode, cur, buffer);
        }
    }
```

可以看出，除了读取到文件结尾的情况，否则成功调用下返回值应当等同于参数`size`。

2.5 步骤5：系统调用`lseek`

`lseek`函数用于调整文件指针的位置，其函数原型如下：

```
int lseek(int fd, int offset, int whence);
```

其中`fd`为文件描述符，`offset`是需要调整的文件指针偏移量，而`whence`是基址位置，其有三种取值：`SEEK_SET`表示文件开头，`SEEK_CUR`表示文件当前位置，`SEEK_END`表示文件结尾位置，将该基址位置加上偏移量即得到新的文件指针位置：

```
int off = sf->edx;
int whence = sf->ebx;if (whence == SEEK_SET)
{
    file[fd].offset = off;
}
else if (whence == SEEK_CUR)
{
    file[fd].offset += off;
}
else if (whence == SEEK_END && off <= 0)
{
    file[fd].offset = inode.size + off;
}
else
{
    sf->eax = -1;
    return;
}
sf->eax = file[fd].offset;
return;
```

需要注意的是我们应当根据已打开文件表项中存储的 `iNode` 信息来获取文件 `iNode` 的总大小信息：

```
diskRead(&inode, sizeof(Inode), 1, file[fd].inodeOffset);
```

函数返回值即为修改后的文件指针位置，需要注意的是当 `whence` 取 `SEEK_END` 时，偏移量应当是非正数（否则即超出了文件大小）。

2.6 步骤6：系统调用 `close`

`close` 函数用于关闭一个已打开的文件，其函数原型如下：

```
int close(int fd);
```

由于我们实现的是立即写回磁盘的 `write` 调用，因此不需要额外考虑内外存的同步，在正常调用情况下直接将对应的系统已打开文件表项清除即可。

```
file[fd].state = 0;
sf->eax = 0;
return;
```

函数返回 `0` 作为正常调用，当提供的文件描述符不可操作时返回 `-1`。

2.7 步骤7：系统调用 `remove`

`remove` 函数用于删除一个文件系统中的指定文件，其函数原型如下：

```
int remove(char *path);
```

参数 `path` 即为对应的文件路径，因此我们需要进行的操作即为，找到文件路径对应的 `iNode`，找到文件路径的父目录对应的 `iNode`，释放该文件的 `iNode`，并删除其父级目录中的对应目录项。我们仍然需要使用到寻找父目录的方法，该方法我们在系统调用 `open` 中已经描述过，在此不再赘述。找到当前文件 `iNode` 和父目录的 `iNode` 以后，我们利用函数 `freeInode` 释放对应的文件 `iNode` 空间，具体的文件类型由 `iNode` 中的属性取得。

```
int type = destInode.type;
freeInode(&sBlock, gDesc, &fatherInode, fatherInodeOffset, &destInode,
&destInodeOffset, str + size + 1, type);
```

函数正常结束返回值 `0`。

2.8 步骤8：指令 `ls`

`ls` 命令用于打印出一个目录下的所有文件，其原型如下：

```
int ls(char *destFilePath);
```

尽管我们将其放置在用户函数下，但显然这个功能的实现是需要依赖于对文件系统的访问的，由于我们没有实现如 `POSIX` 标准中的 `stat` 之类的函数，我们没有很好的系统调用方法来获取 `ls` 函数所需的需求，因此我们额外实现了一个 `ls` 指令相关的系统调用，其原型如下：

```
int do_ls(char *path, char *res);
```

参数 `path` 即需要查询的目录，而 `res` 是一个足够大小的字符串空间，用于存放该目录下的所有文件，文件之间使用空格连接。这样我们就将 `ls` 的查询过程转换到了内核态，将查询结果存放到 `res` 字符串后，再借由 `printf` 输出命令输出到标准输出流。

然后我们需要查询该目录文件下的所有文件，首先我们先判断给定的是否的确是一个目录信息，否则 `ls` 命令失效，返回 `-1`。如果是一个目录文件，我们就可以利用辅助函数 `getDirEntry` 来遍历目录下的文件，并将每个目录名拼接到 `res` 字符串中并返回。

```
char *p = res;
DirEntry dir;
for (i = 0; getDirEntry(&sBlock, &inode, i, &dir) == 0; i++)
{
    stringCpy(dir.name, p, stringLen(dir.name));
    p += stringLen(dir.name);
    *p = ' ';
    p++;
}
*p = 0;
```

由此我们即实现了 `ls` 函数以查询目录下的文件信息。

2.9 步骤9：指令`cat`