lab5 report

文件系统

任课教师：叶保留　　助教：尹熙喆，水兵

学院	计算机科学与技术系	专业	计算机科学与技术
学号	201220154	姓名	王紫萁
Email	201220154@smail.nju.edu.cn	开始/完成时间	2022. 5.18/2022.5.22

1. exercises

exercise 1

exercise1：想想为什么我们不使用文件名而使用文件描述符作为文件标识。

int read(const char *filename, void *buffer, int size);

int write(const char *filename, void *buffer, int size);

用户的读写之前必须经过文件打开的操作，因此使用文件描述符可以在用户打开文件表中快速索引到相应的文件。另外，用户打开文件表中可能有文件名相同但路径不同的文件，并且系统打开文件表中可能有同一个文件的不同进程的FCB，所以文件名不能作为某一文件的唯一标识。而文件描述符是每个文件在打开之后特有的编号，因此之后对该文件的读写操作都可以通过该唯一编号快速找到正确的文件。

exercise 2

exercise2：为什么内核在处理exec的时候，不需要对进程描述符表和系统文件打开表进行任何修改。（可以先往下看看再回答，或者**阅读一下Xv6的**shell）

我们知道，exec函数作用是将一段全新的程序加载到内存并替代调用exec的进程执行新的程序。新程序执行的PID和文件描述符等信息都保持不变，因此无需修改进程描述符表或者系统文件打开表。使得新运行的程序仍然能够通过文件描述符读写在该进程中运行的前一段程序打开的文件。

exercise 3

> exercise3：我们可以通过which指令来找到一个程序在哪里，比如 which ls，就输出ls程序的绝对路径（看下面，绝对路径是/usr/bin/ls）。那我在/home/yxz这个目录执行ls的时候，为什么输出/home/yxz/路径下的文件列表，而不是/usr/bin/路径下的文件列表呢？（请根据上面的介绍解释。）

`ls` 是注册在环境变量中的一个普通的可执行程序，当不指定参数时，ls默认将当前目录的路径作为查询 `inode` 的键值。找到 `inode` 后（可能在系统打开文件表或者需要提前从磁盘 `inode` 区加载），根据inode的指示读出目录文件并调用 `opendir` 函数，读取目录文件中记录的子文件信息，通过控制台终端打印每个文件的名称。不同的参数可以得到文件的大小，建立时间，归属等信息。

2. challenges

challenge 1

> challenge1：system函数（自行搜索）通过创建一个子进程来执行命令。但一般情况下, system的结果都是输出到屏幕上，有时候我们需要在程序中对这些输出结果进行处理。一种解决方法是定义一个字符数组，并让 system 输出到字符数组中。如何使用重定向和管道来实现这样的效果?
>
> Hint：可以用pipe函数（自行搜索）、read函数（你们都会）.....
>
> （在Linux系统下自由实现，不要受约束）

我们用 `dup2` 实现输出文件的重定向，具体来讲，就是希望将 `STDOUT` 指定的文件描述符替换成管道的入口 `pipefd` [1]，做法就是通过 `dup2` 替换文件描述符，这样原本应该出现在控制台上的输出就会输入管道，接着我们通过 `read` 管道出口 `pipefd[0]` 读取到 `buffer` 中就能处理system执行指令的输出了。值得注意的是，为了使父进程能够正常在控制台上输出，我们需要恢复STDOUT的文件描述符。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#define MAX_BUF_LEN 512

int main() {
    char buffer[MAX_BUF_LEN] = { 0 };
    int pipefd[2];
    pid_t id = getpid();
    printf("father pid: %d\n", id);
    pipe(pipefd);
    int recoverfd = dup(STDOUT_FILENO); // 记录STDOUT
    dup2(pipefd[1], STDOUT_FILENO); //重定向到管道入口
    system("ls");
    read(pipefd[0], buffer, MAX_BUF_LEN);   //从管道出口读取
    dup2(recoverfd, STDOUT_FILENO); //恢复STDOUT
    printf("ls print in pid <%d>: %s", getpid(), buffer);
}
```

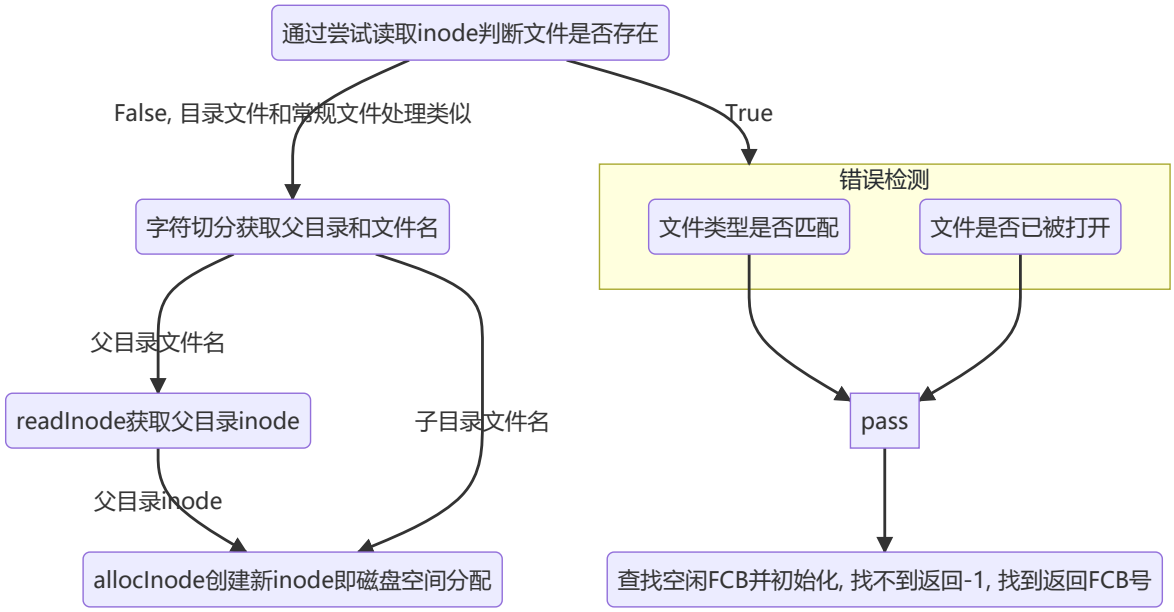
执行结果

```
father pid: 43
ls print in pid <43>: c1
challenge1.c
```

3. 实现原理

3.1. open

作为文件操作的基础，open负责对传入路径的文件执行打开和设置权限的操作。具体流程如下



有一个需要注意的细节：对于不同类型的文件（常规文件和目录文件），文件名的切分略有不同，由于常规文件路径不已'/'结尾（如果检测到返回错误），因此直接向前寻找最后一个'/'即可。而目录文件末尾有无'/'都可以，所以查询 `inode` 时用'name/'，保存 `inode` 时保存目录名为'name'，后期删除 `inode` 时也是会定位'name'。

3.2. write

参考diskWrite函数，三个重要的变量含义分别如下

- `quotient`：文件已经占用的完整磁盘块个数
- `remainder`：文件指针在最后一个磁盘块的偏移量
- `j`：新增完整磁盘块数。
- `buffer`：存放一整个盘块的信息。

写操作有如下几个步骤：

- 合法性检验：检查文件是否打开，是否可写（设备和没有写权限的不可写），写入大小是否合法。
- 取出 `inode`

- 读取最后一个盘块放入 `buffer`：通过 `readBlock` 取出最后一个盘块
- 将待写入的字符放入 `buffer` 中：`buffer[(remainder + stroff) % block_size] = str[stroff];`
- 检查是否需要切换到下一个盘块即缓冲区 `buffer` 已满
    - 检查为该文件分配的盘块是否够用：如果 `新增完整盘块数j + 已有完整盘块数quotient == 文件允许分配的总块数inode.blockCount` 说明应该再分配一块新的块。
    - 将buffer写入最后一个盘块中，
    - 新增盘块数+1
    - 读取新盘块的内容到buffer中
- 直到将size个字符全部写入

存操作：

- 如果写入字符使得文件大小增大了（有可能会有修改之前字符，这种情况就不用修改文件大小），修改文件大小为 `FCB.offset + size`（写入字符数）
- 将信息更新后的inode存入disk
- FCB的offset 累加 size， 返回size。

3.3. read & lseek & close：

- `read` 和 `write` 过程相似，不需要分配新块或者写入 `buffer`，与 `write` 相反是一个 `buffer` 到 `str` 的过程，只需要处理切换新块的情况即可。其余过程完全照搬 `write` 的实现。
- `lseek`：过程也很简单，不同 `whence` 的 `offset` 设置如下：
    - `SEEK_SET`: ofs = offset;
    - `SEEK_CUR`: ofs = file[FCBindex].offset + offset;
    - `SEEK_END`: ofs = inode.size + offset;
- `close`：经过一些列错误检查后，将指定FCB的state置为0表示关闭即可。

3.4. remove

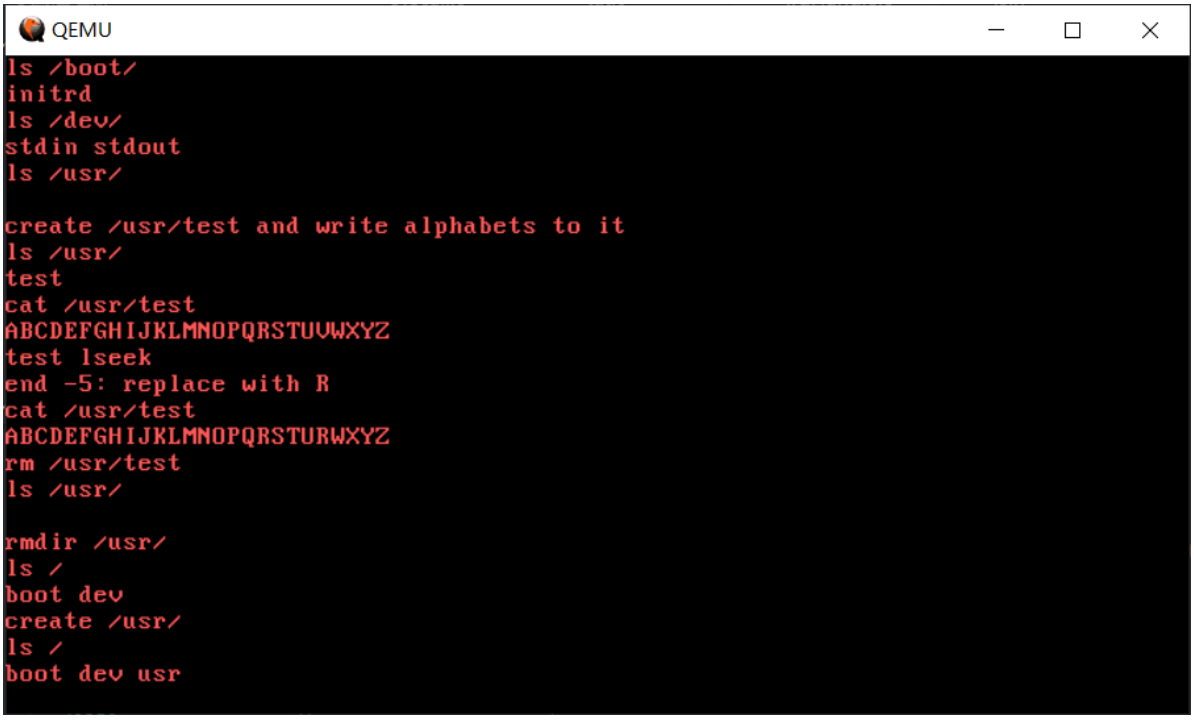
和 `open` 的实现几乎相同，唯一不同的是open为新创建的文件分配inode（`allocInode`），而 `remove` 为已存在且未打开的 文件撤销 `inode`。

3.5. ls & cat

`ls` 指令会去寻找目标路径的目录文件，通过 `open` 打开目录文件后读取其中的内容，每条文件信息在目录文件中以128字节为一个单位，可以作为一个结构体 `DirEntry`，我们用前4字节表示inode号，紧接着64字节表示子文件名，未用部分可保存其他信息，于是我们将目录文件中的内容按块完整读取出来后，就可以通过访问 `DirEntry` 的数组元素访问得到每一个文件名并输出，需要注意首先判断文件是否存在，即inode是否非零。

`cat` 指令我们实现的相对简单，就是将目标文件打开并输出其中的内容。打开文件 → 读取磁盘到 `buffer` → 打印 `buffer` → 读取下一个磁盘到 `buffer` → 打印 `buffer` →...→直到读取完毕返回→关闭文件

4. 测试结果



我们新增了对于 `lseek` 的测试，对于相对易错的 `SEEK_END` 做了测试，在offset向前移动5个单位的位置替换为字母R，发现结果也是正确的。

5. 总结

实现了一套简易的文件系统，理解了文件系统的基本结构和文件类型以及文件增删改查的步骤和实现方法。从lab1到lab5，从boot loader，到中断处理再到进程和进程通信，最后来到文件系统，我们的玩具OS总算大功告成了！感谢老师和两位助教的耐心解惑，debug固然痛苦但最后收获满满🎉