

# 1. 实验要求

本实验为选做，不提供代码框架，需要同学们在实验4的基础上继续完善你的操作系统，实现任意你想实现的功能，这里只给出指导，提供一些可能方向。

## 1.1. 完善格式化程序

实验2中同学们接触到了文件系统格式化程序，其相关文件都放在 `lab/utils/genFS` 目录下。但是代码中删除文件及文件夹的功能没有完成，同学们可以着手完善 `func.c` 中的 `rmdir` 和 `rm` 函数并自行在 `main.c` 中添加代码进行测试（对比添加前和删除后整个文件系统是否一致）。

## 1.2. 内核支持文件读写

在前面的实验中，虽然要求同学们实现了在内核态从文件系统中读文件，但一直没有将相关的接口抽象出来并完成，同学们现在可以将这部分接口实现，并提供相应的系统调用供用户态使用，一般文件系统相关调用有：`open`，`read`，`write`，`lseek`，`close`，`remove` 等。

## 1.3. 用户程序

基于 `open`、`read` 等系统调用实现 `ls`、`cat` 这两个函数，可以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;

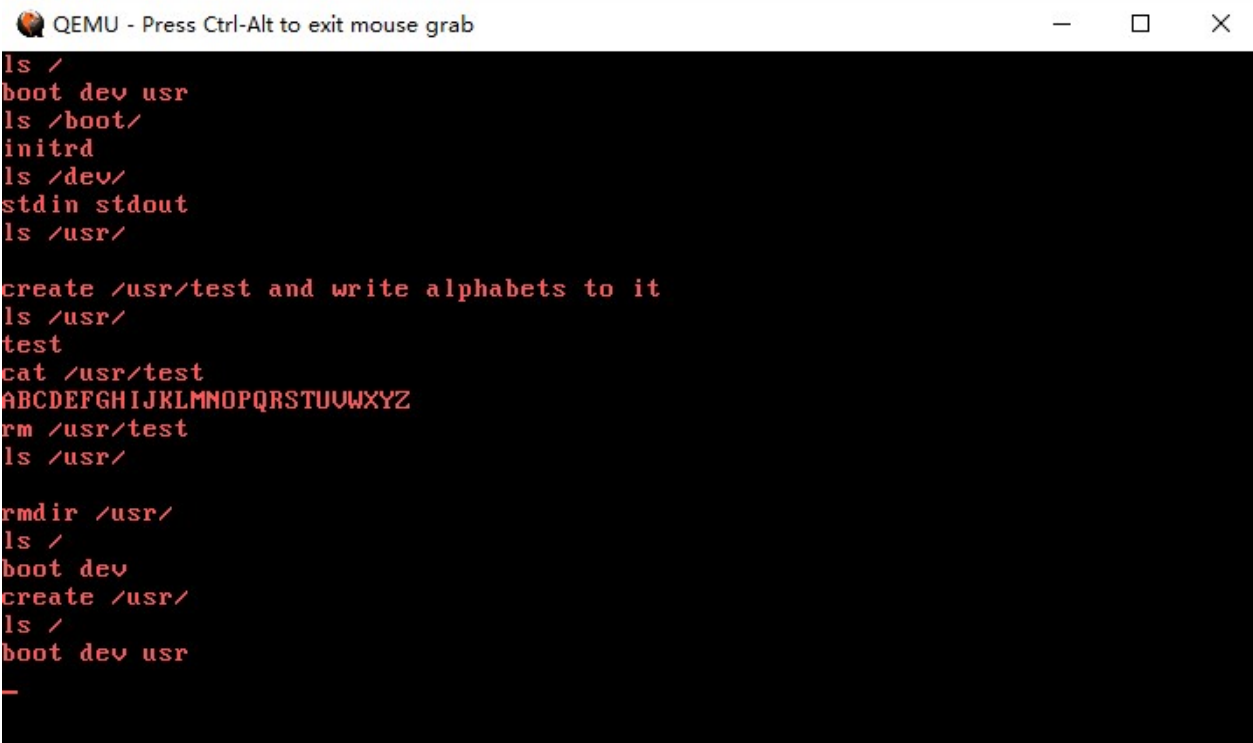
    ls("/");
    ls("/boot/");
    ls("/dev/");
    ls("/usr/");

    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++) {
        tmp = (char)(i % 26 + 'A');
        write(fd, (uint8_t*)&tmp, 1);
    }
    close(fd);
    ls("/usr/");
    cat("/usr/test");
}
```

容

```
    exit();  
    return 0;  
}
```

理想输出如下



```
QEMU - Press Ctrl-Alt to exit mouse grab  
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  
-
```

## 1.4. shell

经过实验4，相信同学们知道了怎么将用户程序放入文件系统，然后再通过 `exec` 执行文件系统中的用户程序，实验5我们以实现一个shell做为结束也做为一个新的起点，同学们可以去尽量移植更多的linux中的命令行用户程序供shell执行。

# 2. 相关资料

## 2.1. 文件系统基础

参照lab2的2.1.层次结构文件系统

## 2.2. 文件系统进阶

### 2.2.1. 什么是文件

这个问题其实不属于操作系统实验的范畴，但对它的讨论有助于理解文件系统的设计与实现中的一些原则。

字节序列

"文件是由文件名标识的一组信息的集合",这是一些操作系统教材中对文件的定义.更具体的,我们可以认为文件是对字节序列的一种抽象.我们已经见识过这种抽象了:在前面的实验中我们要将 `bootloader`、`kMain` 和 `uMain` 三个 `elf` 文件拼接成一个 `img` 做为磁盘.将 `os.img` 抽象成字节序列,因而可以通过数组的方式实现;我们可以为磁盘上每个字节编号,例如第 $x$ 柱面第 $y$ 磁头第 $z$ 扇区中的第 $n$ 字节,把磁盘上所有的字节按照编号的大小进行排序,便得到了一个字节序列...而文件则是字节序列中的一部分,当然也是字节序列.

有很多例子可以让我们相信这种字节序列的抽象是符合直觉的.使用`hd`命令查看编辑的`hello.c`文件:

```
hd hello.c
```

```
00000000  23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e |#include <stdio.|
00000010  68 3e 20 20 0a 0a 69 6e 74 20 6d 61 69 6e 28 29 |h> ..int main(|
00000020  20 7b 0a 09 70 72 69 6e 74 66 28 22 48 65 6c 6c | {..printf("Hell|
00000030  6f 20 57 6f 72 6c 64 21 5c 6e 22 29 3b 0a 09 72 |o World!\n");..r|
00000040  65 74 75 72 6e 20 30 3b 0a 7d 0a                |eturn 0;|.}|
0000004b
```

我们创建第一个用户进程的时候,将二进制文件从磁盘读入内存,实际上也是将文件看成是字节序列,因而可以使用 `memcpy` 等函数(需要自己实现)来直接对文件的内容进行操作.

虽然不同的文件都可以被抽象成字节序列,但直觉上一个文本文件和一个图片文件显然含有不同的内容:我们可以使用图像工具来浏览图片文件,却不能浏览文本文件.这种不同其实取决于如何对字节序列进行解释.

## Everything is a file

如果我们将字节序列的含义赋予文件,那么我们刚才讨论的文件只是一种普通文件.事实上,计算机系统中到处都是字节序列(如果只是无序的字节集合,计算机要如何处理?),我们可以轻松地举出很多例子:

- 配置文件作为一种普通文件,当然也是一种字节序列,对它的解释由使用这个配置文件的程序决定,例如内核的IP转发功能
- 管道是一种先进先出的字节序列,本质上它是内存中的一个队列缓冲区
- `socket`(网络套接字)也是一种字节序列,它有一个缓冲区,负责存放接收到的网络数据包,上层应用将`socket`中的内容看做是字节序列,并使用`receive`操作接收这些内容
- 虚拟设备也可以看做是字节序列,例如 `/dev/random`, `/dev/zero` 等,这些字节序列比较特殊,它们有各自的特点,例如`random`的字节序列是不确定的, `zero`的字节序列则总是0
- 甚至硬件也可以看成是字节序列,我们之前已经见识到,磁盘和内存可以抽象成字节序列,其实各种硬件设备都可以,例如我们在键盘上按顺序敲入按键的编码形成了一个字节序列,显示器上每一个像素的内容按照其顺序也可以看做是字节序列...

Unix将这些五花八门的字节序列全部都看做文件,因此有"Everything is a file"的说法.根据"Everything is a file"的抽象, `iNode` 结构中有用于表示与该文件相关联的设备的属性 `dev_id`,对于存储在磁盘/RAMDISK中的文件,其 `iNode` 结构中的 `dev_id` 属性都是`hda/ram`设备的号码.这种做法最直观的好处就是为不同的事物提供了统一的接口:我们可以使用文件的接口来操作计算机上的一切,而不必对它们进行详细的区分:例如

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

可以通过往配置文件中写入1, 来打开内核的IP转发功能.

```
head -c 512 /dev/sda | hd
```

可以以十六进制的方式查看磁盘上MBR的内容.

```
#include "/dev/urandom"
```

则会将 `urandom` 中的内容包含到头文件中, 由于 `urandom` 是一个长度无穷的字节序列, 提交一个包含上述内容的程序源文件将会令一些检测功能不强的Online Judge平台直接崩溃.

"Everything is a file"的抽象使得我们可以通过标准工具很容易完成一些在Windows下不易完成的工作, 这其实体现了Unix哲学的部分内容: 每个程序采用文本文件作为输入输出, 这样可以使程序之间易于合作. Linux继承自Unix, 也自然继承了这种优秀的特性.

## 2.2.2. 文件操作

### 文件描述符

提到文件操作, 最自然的就是对文件进行读写:

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

其中fd为文件描述符, 是文件的一个标识. 文件作为一种资源(不同的进程需要对不同的文件进行读写), 操作系统需要对其进行管理. 很自然地, 每一个进程都需要拥有一张文件描述符表, 记录了一个进程打开了哪些文件, 每一个有效的文件描述符都对应一个文件的信息.

### 为什么使用文件描述符?

我们可以直接使用文件名做为文件的标识:

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

这样做有什么缺陷吗?

`open(char *path, int flags)` 系统调用用于打开一个文件,它在进程的文件描述符表中寻找一项空闲的,序号最小的表项,填写该表项并返回相应的文件描述符, 其它与文件描述符相关的系统调用有:

```
int close(int fd);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

你可以通过 `man` 来查看他们的功能.

我们不希望每次读写操作都需要从头开始, 于是我们需要为每一个已经打开的文件引入偏移量属性, 来记录目前文件操作的位置. 每次对文件读写了多少个字节, 偏移量就前进多少. 偏移量也可以通过 `lseek` 系统调用来调整, 它的函数原型为:

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

我们希望允许多个进程打开同一个文件(例如使用不同的终端同时浏览同一份文档),但同时希望关系密切的进程(例如父子进程)之间能够共享文件的偏移量.考虑以下父子进程同时写入一个日志文件:

```
int fd = open("log.txt");
if(fork() != 0) {
    write(fd, buf1, len1);
}
else {
    write(fd, buf2, len2);
}
```

若父子进程没有共享文件的偏移量,那么必定有一方写的结果被另一方覆盖.因此,文件的偏移量不应该放在进程的文件描述符表中,否则将不能支持偏移量的共享.另一方面,我们也不希望不相关的进程之间共享同一个偏移量,因此偏移量也不应该作为文件的一个静态属性,否则打开同一个文件的所有进程都将共享同一个偏移量.我们通过引入一张"系统打开文件表"来解决这个问题:打开文件的偏移量放在系统打开文件表中,通过让不同的文件描述符指向系统打开文件表的同一个表项来实现偏移量的共享.

### 进程1打开文件表

fd[0]	
fd[1]	●
fd[2]	
.....	
fd[6]	●
fd[7]	

### 进程2打开文件表

fd[0]	●
fd[1]	
fd[2]	●
.....	
fd[6]	
fd[7]	●

### 系统打开文件表

偏移量	●
偏移量	●
偏移量	●
.....	
偏移量	●
偏移量	●

iNode  
(文件静态信息)

iNode  
(文件静态信息)

iNode  
(文件静态信息)

为了实现上述要求,我们需要对进程的文件描述符表和系统打开文件表进行适当的维护:

- 每当执行 `open` 时,需要在系统打开文件表中添加一项,来增加一个独立的偏移量
- 每当执行 `close` 时,需要对系统打开文件表中相应表项的引用计数器减1.当引用计数器减为0时,代

表没有文件描述符引用该表项, 因此需要对其进行回收

- 每当执行 `fork` 时, 只需要对父进程的文件描述符表进行浅拷贝, 并对系统打开文件表中相应表项的引用计数器加1, 来实现父子进程偏移量的共享
- 每当执行 `exec` 时, 不需要对两者进行任何修改
- 每当执行 `exit` 时, 相当于对文件描述符表中所有有效的表项分别执行了 `close`

### 不需要进行更新的 `exec`

为什么内核在处理 `exec` 时, 不需要对进程文件描述符表和系统文件打开表进行任何修改?

## 2.2.3. 标准输入/输出

为了方便进程进行标准输入输出, 操作系统准备了三个默认的文件描述符:

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

我们经常使用的 `printf`, 最终会调用 `write(STDOUT_FILENO, buf, len)` 进行输出; 而 `scanf` 将会通过调用 `read(STDIN_FILENO, buf, len)` 进行读入.

当内核载入第一个用户进程的时候, 需要手动在文件描述符表和系统打开文件表中填写这三个默认文件描述符的表项. 从此以后, 文件描述符表和系统打开文件表将通过系统调用来维护, 内核不必额外处理.

理解了"Everything is a file"的特性之后, 我们可以很容易地理解标准输入/输出的工作过程. 在上面的例子中, `printf` 将会输出到一个叫 `stdout` 的文件中. 你可以在 `/dev` 目录下找到它, 它是一个软链接, 追根溯源后你会发现, 它指向了一个设备文件, 这个设备文件代表的正是终端. 这样, 输出到代表终端的设备文件实际上是调用了终端驱动程序的 `dev_write`, 于是屏幕上就出现了输出的内容.

## 2.2.4. 重定向和管道

在 `fork-exec` 的连锁机制下, 加上"Everything is a file"的抽象, 重定向的实现就变得十分容易. 例如命令

```
ls > a.txt
```

在shell中的实现大致如下:

```
if((pid = fork()) == 0) {
    // child
    close(STDOUT_FILENO);
    open("a.txt");
    exec("ls");
}
else {
    // father
    waitpid(pid);
}
```



这样, `ls` 在执行之前, 它的标准输出已经被重定向到文件 `a.txt` 了, 但在执行的时候它却毫不知情.

管道能够将一个进程的输出作为另一个进程的输入, 在这里我们只讨论匿名管道. 创建一个管道的系统调用为:

```
int pipe(int pipefd[2]);
```

若执行成功, 它返回两个文件描述符, 其中 `pipefd[0]` 为读端, `pipefd[1]` 为写端.

从实现的角度来看, 往管道里写入数据后, 数据不一定马上被读出, 因此管道实际上是一个先进先出的缓存:

```
typedef struct pipe {
    int front, rear;
    pid_t read_wait_queue[LEN], write_wait_queue[LEN];
    uint8_t buf[BUF_LEN];
} Pipe;
```

需要注意的是, 当管道读写失败时(缓存中数据过少或过多), 需要将请求读写的进程放置到相应的等待队列中, 当其它进程进行了读写操作后, 再唤醒阻塞的进程. 这与生产者消费者问题十分相似, 你想到解决的办法了吗?

我们仍然将管道抽象成文件, 但我们并不需要像普通文件那样读写磁盘, 因为管道的缓冲区实际上是在内存中, 应该由文件系统进行管理. 因此我们需要为系统打开文件表的 `dev_id` 属性增加一种 `PIPE` 类型, 可以将其值设置为 -1 (设备号总是非负的), 用于指示一个管道文件. 另外, 由于管道中的数据总是先进先出的, 不能随机访问, 因此偏移量对管道来说是没有意义的, 故管道两端的文件描述符均可指向系统打开文件表中的同一项, 并且当其引用计数器减为 0 时, 需要回收相应的管道.

### 实现输出到缓冲区的 `system` 函数

`system` 函数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?

这是一个对重定向和管道使用的练习, 你可以尝试在 Linux 下实现这一功能.

## 2.2.5 绝对路径和相对路径 (不强求).

你应该听说过文件的绝对路径和相对路径. 绝对路径是指从根目录到文件所在位置的完整路径, 但如果每次指定一个文件都需要使用绝对路径, 操作起来就会显得十分繁琐, 尤其是目录层数很深的时候. 为了方便用户使用, 文件系统引入了相对路径的概念, 允许通过当前工作目录来定位文件. 那么当前工作目录在哪里呢? 聪明的你应该想到, 当前工作目录不应该是全局的, 否则不同进程都通过相对路径来访问文件的时候将会产生干扰; 因此, 每个进程都应该拥有它的工作目录, 并且

- 通过 `fork` 创建子进程时, 子进程将会继承父进程的工作目录
- 通过 `exec` 执行新程序时, 工作目录将会改为新程序所在的目录
- 文件系统需要向用户进程提供一个切换目录的系统调用

`cd` 程序在哪里

我们可以通过 `which` 命令来查看一个shell命令的程序所在的绝对路径, 例如

```
$ which gcc
/usr/bin/gcc
```

输出结果告诉我们, 我们平时使用的gcc命令的程序其实位于 `/usr/bin/` 目录下. 但对于我们使用得最多的 `cd` 命令, `which` 却找不到它所在的目录, 你知道这是为什么吗? 如果你感到困惑, 请到互联网上搜索相关资料.

同学们应该看出来了, 由于一些历史遗留问题, 操作系统实验框架代码没有完全按照Unix思想实现。

## 3. 解决思路

### 3.1. 完善格式化程序

注意删除文件或目录后需要更新 `superBlock`, `inodeBitmap`, `blockBitmap`。

### 3.2. 内核支持文件读写

#### 3.2.1. 文件控制块

内核使用FCB（File Control Block, 文件控制块）这一数据结构对进程打开的文件进行管理, FCB中需要记录其对应的是哪个文件, 该文件是以哪种方式打开的（读、写），该文件的读写偏移量

FCB的索引号称为文件描述符, 每个进程的PCB中对其打开的文件的文件描述符进行记录

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

推荐的文件控制块数据结构如上, 同学们可根据需求自行修改

#### 3.2.2. 库函数

`open`为Linux提供的系统原语, 其用于打开（或创建）由路径 `path` 指定的文件, 并返回文件描述符 `fd`, `fd` 为该文件对应的内核数据结构FCB的索引, 参数 `flags` 用于对该文件的类型、访问控制进行设置

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



**read**为Linux提供的系统原语，其用于从 `fd` 索引的FCB中的文件读写偏移量处开始，从文件中读取 `size` 个字节至从 `buffer` 开始的内存中，并返回成功读取的字节数，若文件支持seek操作，则同时修改该FCB中的文件读写偏移量

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

**write**为Linux提供的系统原语，其用于向 `fd` 索引的FCB中的文件读写偏移量处开始，向文件中写入从 `buffer` 开始的内存中的 `size` 个字节，并返回成功写入的字节数，若文件支持seek操作，则同时修改该FCB中的文件读写偏移量

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

**lseek**为Linux提供的系统原语，其用于修改 `fd` 索引的FCB中的文件读写偏移量（若文件支持seek操作），返回修改后的偏移量

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

**close**为Linux提供的系统原语，其用于关闭由 `fd` 索引的FCB

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

**remove**为C标准库的函数，其用于删除 `path` 指定的文件

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

## flags详解

本实验只考虑文件打开的四种权限

```
#define O_WRITE 0x01
#define O_READ 0x02
#define O_CREATE 0x04
#define O_DIRECTORY 0x08
```

⇒ 自己加到框架代码中

这四种权限可以相互补充，设置了 `O_WRITE` 才能写文件，设置了 `O_READ` 才能读文件，设置了 `O_CREAT` 才能在打开的文件不存在时新建文件，设置 `O_DIRECTORY` 表示读、写或新建的为文件夹，而不是文件。

## whence详解

所有打开的文件都有一个当前文件偏移量 `cfo`，`lseek` 参数 `offset` 的含义取决于参数 `whence`：

```
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
```

1. 如果 `whence` 是 `SEEK_SET`，文件偏移量将被设置为 `offset`。
2. 如果 `whence` 是 `SEEK_CUR`，文件偏移量将被设置为 `cfo` 加上 `offset`，`offset` 可以为正也可以为负。
3. 如果 `whence` 是 `SEEK_END`，文件偏移量将被设置为文件长度加上 `offset`，`offset` 可以为正也可以为负。

↓  
一般 offset 为负。

### 3.2.3 Everything is a file

#### 文件、目录、设备、管道、套接字、链接

对类Unix系统，其文件系统中除却真实存储在磁盘上的通常文件与目录这两种类型的文件外，将物理硬件本身、内核提供的功能等等，也作为文件归入文件系统之中，这些作为文件也都有inode与之对应

类Unix系统的文件系统通常将所有文件划分为通常文件（Regular File）、目录文件（Directory）、块设备文件（Block Device）、字符设备文件（Character Device）、管道文件（FIFO）、套接字文件（Socket）、链接文件（Symbolic Link）

其中，块设备文件与字符设备文件是按照存取方式进行的划分，前者例如 `/dev/sda`、`/dev/loop0`，后者例如 `/dev/tty1`、`/dev/random`，前者支持随机访存，后者仅支持顺序访存，一般看来，前者支持Seek操作，后者不支持Seek操作

对于设备文件同样可以按照其是否具有物理实体进行划分，即物理设备（对实际存在的物理硬件的抽象，例如 `/dev/sda`）与虚拟设备（内核提供的功能，例如 `/dev/loop0`）

类Unix系统将物理硬件设备与内核提供的功能作为文件归入文件系统是为统一用户操作接口，例如以下代码，通过向标准输出的设备文件中写入 `Hello World!`，即可实现在当前终端中打印出该字符串

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (int argc, char *argv[]) {
    int fd = open("/dev/stdout", O_WRONLY);
    write(fd, (void*)"Hello World!\n", 13);
    close(fd);
    return 0;
}
```

我们实现的简单文件系统中同样可以将物理硬件与内核提供的功能作为文件进行抽象，创建诸如 `/dev/stdin`、`/dev/stdout` 这些文件

实验要求完成 `stdin`、`stdout` 和共享内存的文件抽象，数据结构上要做如下变化：

```

struct Device {
    int state;
    int value;
+ int inodeOffset; //XXX inodeOffset in filesystem, for syscall open
    struct ListHead pcb; // link to all pcb ListHead blocked on this device
};
typedef struct Device Device;

```

而在文件系统中也要真正加入 `/dev/stdin`、`/dev/sdtout`、`/dev/shmem` 等文件

这样实验4中直接调用 `read`、`write` 进行共享内存读写就行不通，现在需要先 `open`，再进行读写

### 3.2.4. 完善系统调用处理

```

#define SYS_WRITE 0
#define SYS_FORK 1
#define SYS_EXEC 2
#define SYS_SLEEP 3
#define SYS_EXIT 4
#define SYS_READ 5
#define SYS_SEM 6
#define SYS_GETPID 7
+ #define SYS_OPEN 8
+ #define SYS_LSEEK 9
+ #define SYS_CLOSE 10
+ #define SYS_REMOVE 11

```

```

void syscallHandle(struct TrapFrame *tf) {
    switch(tf->eax) { // syscall number
        case SYS_WRITE:
            syscallWrite(tf);
            break; // for SYS_WRITE
        case SYS_READ:
            syscallRead(tf);
            break; // for SYS_READ
        case SYS_FORK:
            syscallFork(tf);
            break; // for SYS_FORK
        case SYS_EXEC:
            syscallExec(tf);
            break; // for SYS_EXEC
        case SYS_SLEEP:
            syscallSleep(tf);
            break; // for SYS_SLEEP
        case SYS_EXIT:
            syscallExit(tf);
            break; // for SYS_EXIT
        case SYS_SEM:

```

```

        syscallSem(tf);
        break; // for SYS_SEM
    case SYS_GETPID:
        syscallGetPid(tf);
        break; // for SYS_GETPID
+   case SYS_OPEN:
+       syscallOpen(tf);
+       break; // for SYS_OPEN
+   case SYS_LSEEK:
+       syscallLseek(tf);
+       break; // for SYS_SEEK
+   case SYS_CLOSE:
+       syscallClose(tf);
+       break; // for SYS_CLOSE
+   case SYS_REMOVE:
+       syscallRemove(tf);
+       break; // for SYS_REMOVE
    default: break;
}
}

```

需要添加四个处理函数 `syscallOpen`、`syscallLseek`、`syscallClose`、`syscallRemove`，完善两个处理函数 `syscallRead`、`syscallWrite`，其他函数根据需求自行修改。

### 3.3. 用户程序 *(参照 lab 4)*

---

略

### 3.4. shell *输入命令 → fork 一个子进程 → 用其执行程序*

---

略

### 3.5. 文件系统目录结构

---

如下文件系统的目录结构如下可供参考

```
+/  
|---+boot  
|   |---initrd  
|   |---...  
|---+dev  
|   |---stdin          #标准输入设备文件  
|   |---stdout         #标准输出设备文件  
|   |---shmem          #共享内存文件  
|   |---...  
|---+usr  
|   |---...            #用户文件  
|---...
```

## 4. 相关资源

---

- [EXT4文件系统](#)

## 5. 作业提交

---

- 本次作业需提交可通过编译的实验相关源码与报告,提交前请确认 `make clean` 过.
- 请大家在提交的实验报告中注明你的邮箱, 方便我们及时给你一些反馈信息.
- **学术诚信:** 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 请你在实验截止前务必确认你提交的内容符合要求(**压缩包命名**, 格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除一定的分数, 最高可达50%.
- 实验不接受迟交, 一旦迟交按**学术诚信**给分.
- 其他问题参看 `index.pdf` 中的**作业规范与提交**一章
- 本实验最终解释权归助教所有

截止时间: 2020-6-22 23:55