

2022spring 《操作系统》lab5实验报告

姓名：武雅琛

学号：201220102

exercise3: task: 2 challenge 1 😊

一、PART1: exercise

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。

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

- 因为在一个程序中一个文件可以被同一个进程或者多个进程**多次打开**，分别访问文件的**不同字节部分**(offset)，执行不同的操作。
- 文件指针fd为一个非负整数，指向了**当前进程文件已打开表**对应的一个表项。

对于同一个文件在不同进程中分别open/create，可以用文件描述符查找对应进程FCB中的进程已打开表。

对于同一个文件在**同一个进程中多次打开**，可以用**不同的fd**指向**不同的进程已打开表项来标识区分**，并且，指向不同的系统已打开表项也可以设置不同的flag和offset来区分打开的类型和文件指针的位置。

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。（可以先往下看再回答，或者阅读一下Xv6的shell）

```
1 //shell
2 while((fd = open("console", O_RDWR)) >= 0){
3     if(fd >= 3){
4         close(fd);
5         break;
6     }
7 }
```

- 首先可以看出，shell作为Xv6载入的第一个用户进程，承担了和**用户交互访问文件和执行程序**的任务。
- 在shell的main函数开始时第一件事就是打开**标准输入输出文件**，并且填写好了系统已打开文件表和进程已打开文件表，由于之前未打开任何文件，它们的文件描述符恰好为0，1，2。按照约定，在**执行标准输入输出时就会对文件描述符为0、1、2的文件操作**。

```

1 //shell/main()
2 while(getcmd(buf, sizeof(buf)) >= 0){
3     ...
4     }
5     if(fork1() == 0)
6         runcmd(parsecmd(buf));
7     ...
8 //runcmd
9 case EXEC:
10    ...
11    exec(ecmd->argv[0], ecmd->argv);
12    ...
13    break;

```

- 接下来注意到在shell命令行执行一个输入的命令时，会创建一个子进程，在子进程中调用exec()函数执行命令对应的程序。这样由shell命令行产生的子进程都在进程已打开文件表中默认打开标准输入输出文件在进程已打开文件表的0、1、2项，并且可以共享对标准输入输出文件的同一个系统已打开表项的偏移量和对应的同一个设备文件。
- 方便了内核统一处理多进程的标准输入输出。

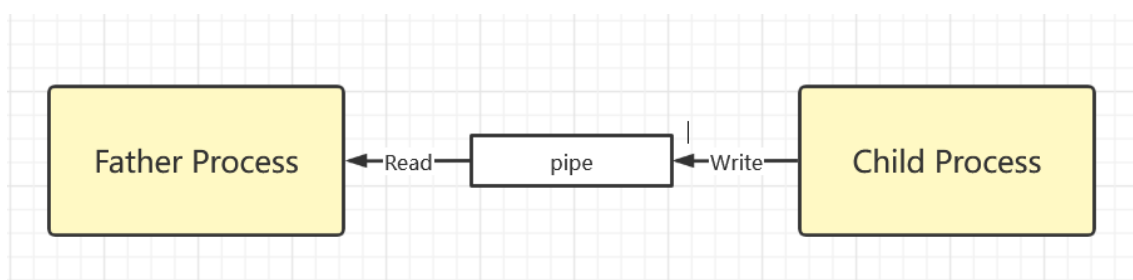
exercise3

- 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。）
- 当我们执行which ls的时候相当于输出 ls 这个程序在目录结构下的绝对路径，体现的是程序文件在磁盘中的相对位置。
- 当我们执行ls的时候相当于把这个程序从它的绝对路径下装载到内存上创建一个新的进程执行，这个新的进程是通过shell命令行进程fork产生的，并通过exec系统调用执行了ls程序，它们分别继承父进程和原进程的工作目录，所以继承了shell命令行的工作目录，而ls命令的功能就是得到当前工作目录下的文件列表。

二、PART2:Challenge

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？



我采用了重定位和管道通信的方法实现了以上的功能。

首先在父进程中用dup函数拷贝标准输入输出的进程文件表项，然后关闭0号和1号文件，创建管道，将标准输入输出替换为管道文件的输入输出，实现重定向输入输出。

然后通过fork创建子进程。在父进程中关闭写管道文件描述符，在子进程中关闭读管道文件描述符。

那么在子进程中执行 `system("ls -l")` 语句时，输出就会重定向到管道中，继而在父进程中执行 `read` 函数读出管道内容到 `buf` 数组当中。

最后为了将字符数组的内容输出在终端，使用 `dup2` 函数将0号和1号文件重新替换为标准输入输出文件。

- 终端输出结果符合预期：

```
oslab@oslab-VirtualBox:~$ gcc redir.c -o redir
oslab@oslab-VirtualBox:~$ ./redir
This is father process!
bufout:
总用量 68
-rwxrwxr-x 1 oslab oslab 8744 5月 25 03:15 a.out
drwxrwxrwx 4 oslab oslab 4096 5月 23 21:52 lab5
-rwxrwxr-x 1 oslab oslab 8744 5月 25 03:36 redir
-rw-rw-r-- 1 oslab oslab 1069 5月 25 03:36 redir.c
-rw-rw-r-- 1 oslab oslab 3072 5月 25 03:13 redir.o
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 公共的
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 模板
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 视频
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 图片
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 文档
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 下载
drwxr-xr-x 2 oslab oslab 4096 2月 22 14:26 音乐
drwxr-xr-x 2 oslab oslab 4096 2月 25 21:09 桌面
oslab@oslab-VirtualBox:~$ This is Child process!
```

三、PART3:task

task1

- 完成 `irqHandle.c` 里面有关文件的系统调用的内容，都用 `TODO` 标好了，并且添加了提示。

☑ `TODO: open`

```
1 //lib/syscall.c
2 return syscall(SYS_OPEN, (uint32_t)path, (uint32_t)flags, 0, 0, 0);
```

可以看到 `open` 传入了两个参数：**path 文件路径**（包括文件名）以及**文件控制标志**。

■ 文件路径

文件路径在打开已存在文件时直接可以传入 `readInode` 函数读取活动索引节点。

但是在创建新文件时要拆分为父亲目录的路径和文件名，可以利用提供的函数来将**字符串划分并且拷贝到两个字符数组**重用，实现如下：

```
1 //irqhandle.c/syscallOpen()
2 stringChrR(str, '/', &pos); //pos赋值为从右数起第一个 '/' 出现的位置索引
3 char fatherstr[128];
4 char filename[128];
5 stringCpy(str + pos + 1, filename, stringLen(str) -
pos - 1);
6 stringCpy(str, fatherstr, pos);
```

■ 文件控制标志

这一部分我主要通过掩码的方式来取出各个比特位的内容获取文件控制信息。

这些信息可能用于对比os内保存的文件信息来**判断操作是否合法**或者用于创建新的文件时**初始化文件控制信息**。

比如： `(flags & 0x8) == 0x8` 可以判断打开的文件是否是目录文件

另外从目的上来说，分为创建新文件和打开已有文件。

■ 创建新文件

创建新文件要处理path得到fatherpath以及filename，利用readInode读出目录节点，传入allocInode写入目录项并分配新的节点。最后注意也要在系统文件已打开表中打开新创建的文件。

■ 打开已有文件

为了防止重复打开，需要遍历系统已打开文件表确认是否已经打开文件。然后在系统已打开表项中打开文件。

✅ TODO: Write

```
1 //lib/syscall.c
2 return syscall(SYS_WRITE, fd, (uint32_t)buffer, size, 0, 0);
```

可以看到传入了三个参数：文件描述符，字符缓存和写入的字符数目。

■ 文件描述符

文件描述符主要涉及了错误处理：访问了超出文件范围或者未打开的文件则返回-1结束。

```
1 if ((fd - MAX_DEV_NUM) < 0 && (fd - MAX_DEV_NUM) > MAX_FILE_NUM)
2 ...
3 if (file[fd - MAX_DEV_NUM].state == 0)
4 ...
```

■ 字符缓存和写入的字符数目

这一步分主要涉及到如何块内写入，如果跨块写入的问题。

首先要获得写入文件的对应位置的**块号**和**块内偏移量**。

```
1 int quotient = file[sf->ecx - MAX_DEV_NUM].offset /
  sBlock.blockSize;
2 int remainder = file[sf->ecx - MAX_DEV_NUM].offset %
  sBlock.blockSize;
```

块内写入：将block的内容从外部存储读出，使用 `Memcpy` 将特定位置的内容用 `buffer` 的内容拷贝进去，然后再写入外存。

跨块写入：要注意写道前一个块的最后一个字节后写回前一个块，读出后一个块写入对应的内容。

✅ TODO:Read

read和write处理相近，不做过多叙述。

✅ TODO:lseek

```
1 | return syscall(SYS_LSEEK, fd, offset, whence, 0, 0);
```

传入三个参数：**文件描述符**，**偏移量**，**模式参数**。

这一部分只要求了解lseek的三种模式参数：

SEEK_SET：从文件开头开始偏移

SEEK_CUR：从系统已打开文件表当前偏移量计算偏移

SEEK_END：从文件的结尾处计算偏移

☑ TODO:close

```
1 | return syscall(SYS_CLOSE, fd, 0, 0, 0, 0);
```

只传入了文件描述符，对应的处理和write一致，做错误判断即可。

另外，因为我们的文件系统只有系统已打开文件表和固定索引节点，在关闭文件时只需要将系统已打开文件表中的表项移除，也就是把对应的表项的state修改清零。

☑ TODO:remove

```
1 | return syscall(SYS_REMOVE, (uint32_t)path, 0, 0, 0, 0);
```

删除文件的过程并不复杂，大部分和之前重合并调用接口freeInode。

这里要注意**已经打开的文件不允许删除**，所以要先遍历文件已打开表查看对应的文件是否正在打开。

task2

- 在app里面完善简易的ls和cat函数。

☑ ls函数

将目录文件的内容读取到buffer之后，将buffer的数组指针类型修改为**目录项的指针类型**采用下标读取。

特别要注意的是，如果读取到的目录项**inode为0**，则说明**目录文件已空**，结束循环。

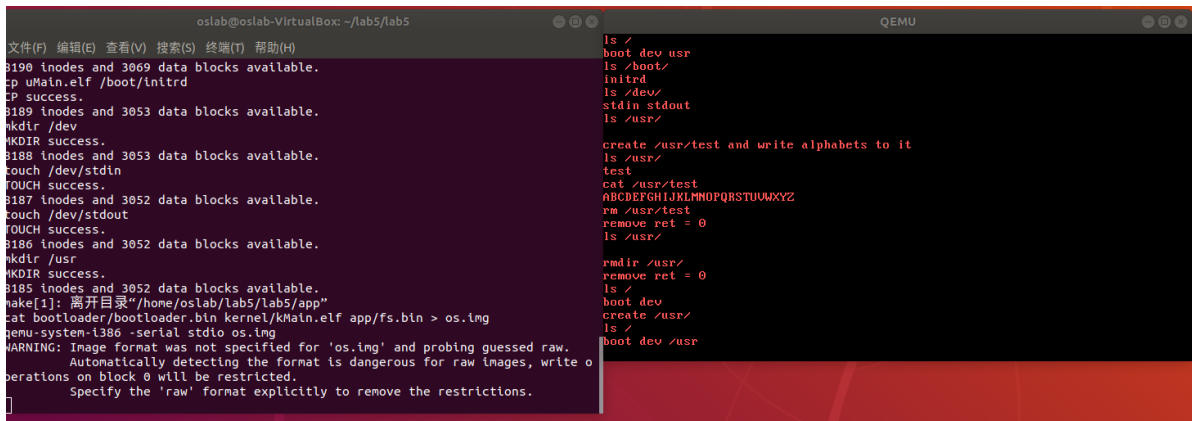
☑ cat函数

这里将读出的有效内容拷贝到Printbuffer中输出即可。

特别要注意根据ret的内容判断是否读取到最后一个块，**如果ret < blocksize说明读到最后一个块，输出之后结束。**

否则输出当前的内容之后再次对文件读取直到文件末尾。

Test



```
oslab@oslab-VirtualBox: ~/lab5/lab5
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
3190 inodes and 3069 data blocks available.
cp uMain.elf /boot/initrd
CP success.
3189 inodes and 3053 data blocks available.
mkdir /dev
MKDIR success.
3188 inodes and 3053 data blocks available.
touch /dev/stdin
TOUCH success.
3187 inodes and 3052 data blocks available.
touch /dev/stdout
TOUCH success.
3186 inodes and 3052 data blocks available.
mkdir /usr
MKDIR success.
3185 inodes and 3052 data blocks available.
make[1]: 离开目录"/home/oslab/lab5/lab5/app"
cat bootloader/bootloader.bin kernel/kMain.elf app/fs.bin > os.img
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

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
remove ret = 0
ls /usr/

rmdir /usr/
remove ret = 0
ls /
boot dev
create /usr/
ls /
boot dev /usr
```

四、PART4: feeling

- oslab完结撒花🌸🌸
- 非常感谢zjgg的陪伴，虽然因为网课基本没有见过zjgg本人，但是从实验手册上来看可以看出zjgg们的辛勤付出，再次感谢，我们有缘江湖再见~