OS第 5 次实验

文件系统

151220030 → 高子腾 : gzt@outlook.com

2017 年 6 月 18 日

https://github.com/sebgao/sebopesyslab

第一部分 实验环境

编译虚拟机环境 Debian 32bit，注意只能在 Debian 32bit 下编译的镜像文件才能正确运行（不能在 Ubuntu 32bit 下编译）. 运行 QEMU 的平台则没有限制.

GCC 编译版本 gcc version 4.9.2 (Debian 4.9.2-10)

第二部分 实验结果

完成了所有要求，

1. 具有文件系统的磁盘镜像
2. 实现文件相关系统调用并封装
3. 实现 3 个小工具，formatter，read_myfs，copy2myfs
4. 有演示程序，见 /app

第三部分 详细实验结果

1 磁盘映像组织

文件系统实现方式类似讲义的实现方式,但只实现了一级目录,一级索引,磁盘映像组织图示可以在 fmt/file.h 里找到

bootloader 占 1 个 sector，bitmap 占 256 个 sector，dir 目录块每个占 1 个 sector，但所有 dir 目录块都属于根目录范畴，此处为了与 bitmap 保持一致，有 256 个 dir 目录块。inode 节点也每个占用 1 个 sector，为了保持一致，也有 256 个 inode 节点。随后是 data 块区，每个 data 块占 1 个 sector。具体详见 fmt/file.h 里的示意图。

2 磁盘工具工具

fmt 文件夹下有如下源文件

file.h 描述磁盘映像结构，定义各种结构，定义各种辅助函数

formatter.c 的实现还是比较简单的，首先把 boot.bin 复制进 bootloader 区，然后将 bitmap 区清零，初始化 dir，即把每个 dir 的 entry 的 inode_offset 置为-1，表示此处空闲。用法是直接

```
./formatter
```

copy2myfs.c 首先从 dir 的 entry 里找到与当前文件名相同或者是空闲的 entry，分配 inode 块，再在 inode 块里分配 data 块，进行每 512B 的写入，详见 copy2myfs.c。使用方法为

```
./copy2myfs 宿主机文件名 磁盘映像里的文件名
```

read_myfs.c 读出在磁盘映像中的文件信息，并输出，也可以去掉注释输出文件内容（不建议）。使用方法为

```
./read_myfs 磁盘映像里的文件名
```

在 Makefile 中，生成磁盘映像的操作为

```
$(IMAGE): $(BOOT) $(KERNEL) $(GAME) $(APP) formatter
  ./formatter
  ./copy2myfs $(KERNEL) kernel
  ./copy2myfs $(GAME) game
  ./copy2myfs $(APP) app
  ./copy2myfs test.txt test.txt
```

3 系统调用

系统调用底层具体代码见 kernel/src/fs

系统的底层代码复用了绝大部分 fmt 小工具里的函数，实现起来异常通顺，kernel/include/file.h 也基本照搬 fmt/file.h，只不过实现读写的工程细节比较多，buffer 偏移啊什么的，还有最麻烦的是跨多个 512B 的读和写，可以看到非常多的代码在处理这种情况。

另外我封装这些函数也封装了好几层，以 read 为例，

```
int fs_read_base_kr(int fd, void* buf, int32_t len)
```

是最底层的 read，不提供 len 越界检测，而

```
int fs_read_kr(int fd, void* buf, int32_t len)
```

加了越界检测，

```
int fs_read_md(int fd, void *buf, int len)
```

呃这层是包装一下，其他的 open 有封装其他功能，以上的 fd 是指内核中全局共有的文件描述符，而接下来的事情就不一样了

```
int fs_read_port(int fd, void *buf, int len)
```

这里的 fd 是用户进程的 FCB 数组的文件描述符，此函数取 FCB 对应的项的内核文件描述符，并进行 fs_read_md，关于 FCB 结构定义为

```
typedef struct FCB {
  int32_t fd_kr;
} FCB;
```

就是包括一个内核文件描述符的结构。

为什么要用用户进程和内核之间独立的文件描述符，又把它们对应起来呢？理由是这种方法比较配合进程和 fork 操作，fork 只复制了 FCB 对应的 fd_kr，而底层内核文件描述符的 offset 是由子进程和父进程共享的，这也比较符合现代操作系统。

后缀为 port 的 fs 函数即为系统调用的对应函数，在用户态文件函数为 open，read，write，lseek，close。

第四部分　综合

为了骄傲的展示程序正确性，呃，还是那个 app/src/main.c，有如下代码

```
int fd = fs_open("singer.txt", FS_RWC);
char buf[300];
fs_read(fd, buf, 300);
printf("APP#READ singer.txt: %s\n", buf);
fs_lseek(fd, 0, SEEK_SET);

fork();

char buf2[300];
strcpy(buf2, "Hello from the other side!");
fs_write(fd, buf2, 26);
printf("APP#WRITE singer.txt: %s\n", buf2);
```

可以看出逻辑是先读取一个文件，选项 FS_RWC 代表着不存在就创建空文件，输出文件内容。然后 fork，分别一串字符，但不含结束符。

运行时可以看到，第一次运行读取时输出的是空字符串，结束，不再重新编译，运行第二次的时候读取时输出的是两个字符串拼接的结果，这也证实了文件系统实现的正确性。

第五部分　一些问题

由于我只实现了一级 inode 索引，单个文件最多是 64KB，而 kernel 是 170 多 KB，一个 inode 块装不下，我就选择了比较妥协的实现机制，inode 块索引 64KB 为止，但 kernel 文件是全复制进去，bitmap 也进行相应修改，而且是连续存放，因为 bootloader 要加载 kernel，bootloader 512B 的大小限制又存不进文件系统的逻辑，就只能以一个固定偏移量进行读取 kernel 了。

第六部分　一些用户接口

```
int open(char *pathname, int flags); //flags可为FS_RW (如不存在返回-1), FS_RWC (如不存在创建文件)
int read(int fd, void *buf, int len);
int write(int fd, void *buf, int len);
int lseek(int fd, int offset, int whence); //whence可为SEEK_SET, SEEK_CUR, SEEK_END, 含义同 linux
int close(int fd);
```

第七部分　实验感想

实现一个文件系统，我第一个反应是惊呆的。真的因为文件系统涉及面太大了，不可能面面俱到。而在我们这个玩具系统里面的玩具文件系统里，只实现了简单的五个函数，这五个函数已经足够复杂了，比如说 lseek 到一个超出文件长度的 offset，然后再进行 write 操作，这种情况实现起来远远复杂的多，其他 unexpected 情况很多很多，所以我就都运用了 KISS 原则，嘿嘿嘿。