

二级文件系统设计报告

学号：1951443

姓名：罗劲桐

1. 实验目的

本次实验的目的是通过学习 UNIX V6++实现文件系统的方式，编写一个 UNIX 文件系统，如何分解文件系统功能，依托开源软件进行二次开发。实现基本文件操作，从而掌握 UNIX 文件系统结构。

2. 实验内容

- 剖析 Unix V6++源代码，深入理解其文件管理模块、高速缓存管理模块和硬盘驱动模块的设计思路和实现技术。
- 裁剪 Unix V6++内核，用以管理二级文件系统。
- 实现磁盘高速缓存，提升系统性能。
- 实现单用户二级文件系统的界面，使文件系统提供 shell，作为用户的字符界面。
- 实现多用户二级文件系统，实现 shell 并发访问内核，允许多个用户同时访问二级文件系统。

3. 实验要求

3.1. 虚拟硬盘

使用一个普通的大文件（如 c: myDisk.img，称之为一级文件）来模拟 UNIX V6++ 一个文件卷（把一个大文件当一张磁盘用）。一个文件卷实际上就是一张逻辑磁盘，磁盘中存储的信息以块为单位，每块 512 字节。

myDisk.img 文件拥有标准的 UNIX 卷格式：

SuperBlock	Inode 区	文件数据区
------------	---------	-------

3.2. libc api

使用 UNIX V6 文件管理系统的内核设计思想。基于上述思想设计文件管理模块实现以下 API：

```
1. void Ls(bool verbose = false);  
2. void Cd(string dirName);
```

```
3.     void Mkdir(string dirName);
4.     void Create(string fileName, string mode);
5.     void Delete(string fileName);
6.     void Open(string fileName, string mode);
7.     void Close(string fd);
8.     void Seek(string fd, string offset, string seek_set);
9.     void Write(string fd, string in, string size);
10.    void Read(string fd, string out, string size);
11.    void Validate(string username, string password);
12.    void AddUser(string username, string password);
13.    void Chmod(string fileName, string mode);
```

3.3. 需要实现的功能

- 将宿主机中的文件存入虚拟磁盘
- 将虚拟磁盘中存放的文件取出，保存在宿主机文件系统中。要求.txt 能够被记事本打开，png 可以显示。
- 创建新目录、文件，删除文件。
- fformat，用来，用来初始化初始化 c:\myDisk.img。
- Unix fs 接收用户输入的文件操作命令，接收用户输入的文件操作命令，测试文件系统和测试文件系统和 API。
- 实现 buffer 缓存
- 实现多用户并发访问
- 实现 BSD 文件锁

4. 概要设计

4.1. shell 输入格式

文件系统采用控制台命令作为输入，按照系统给定的命令即可完成相应的功能。

4.1.1. help

Usage: help[pattern]

Description: 帮助文档，显示有关内置命令的信息。

Exit status: 除非未找到 pattern 或给出无效选项，否则返回成功。

4.1.2. login

Usage: login username password

Description: 用户必须先登录。

Exit status: 用户密码不匹配返回错误, 否则返回成功。

4.1.3. `fformat(root)`

Usage: `fformat`

Description: 将整个文件系统进行格式化, 清空所有文件及目录

Exit status: 当前用户为 `root` 用户且文件系统内无其他正在使用的用户, 否则返回失败

4.1.4. `exit`

Usage: `exit`

Description: 正确退出, 正常退出会调用析构函数, 若有在内存中未更新到磁盘上的缓存会及时更新, 保证正确性。

Exit status: 默认成功

4.1.5. `mkdir`

Usage: `mkdir dir`

Description: ‘`mkdir`’ 根据 `dir` 目录名创建目录。`dir` 如果目录已经存在, 则会出错。

`dir` 可以是相对路径, 也可以是绝对路径。

Exit status: 退出状态为零表示成功, 非零值表示失败。

4.1.6. `cd`

Usage: `cd [dir]`

Description: 改变当前工作目录。将当前目录更改为 `dir`。默认 `dir` 是用户默认 `HOME`。

`dir` 可以是相对路径, 也可以是绝对路径。

Exit status: 目录名过长, 目录路径不存在, 或无访问权限, 返回失败。

4.1.7. `ls`

Usage: `ls [-l]`

Description: 列出当前目录有关文件的信息。

Exit status: 目录名过长, 目录路径不存在, 或无访问权限, 返回失败。

4.1.8. `create`

Usage: `create file`

Description: 以默认文件属性新建一个文件，文件名为 file。

Exit status: 文件名过长，文件不存在，返回失败。

4.1.9. rm

Usage: rm name

Description: 使用 rm 删除 name 指定的一个文件。

Exit status: 文件名过长，目录路径不存在，准备删除一个没有权限的文件，返回失败。

4.1.10. open

Usage: open file [-rw]

Description: 函数 open，打开一个文件。若要进行文件的读写必须先 open。默认为可读可写，可通过-rw 指定。

Exit status: 文件名过长，目录路径不存在，或无访问权限，返回失败。

4.1.11. close

Usage: close fd

Description: 通过 open 获取的 file descriptor 关闭一个文件。可以对已经打开的文件进行关闭。

Exit status: 文件描述符不存在或超出范围，返回失败。

4.1.12. seek

Usage: seek fd offset [pattern]

Description: 通过 open 获取的 file descriptor，修改读写指针位置。

offset 指定从位移起始位置开始的偏移量可正可负。

pattern 指定位移起始位置，为 0 (SEEK_SET), 1 (SEEK_CUR), 2 (SEEK_END)，默认为 0 (SEEK_SET)。

Exit status: 文件描述符不存在或位移后读写指针超出范围，返回失败。

4.1.13. write

Usage: write fd [content] [-i inFileName] size

Description: 通过 open 获取的 file descriptor，将 content 写入一个已经打开的文件中。通过-i 指定输入的外部文件。

Exit status: 文件描述符不存在，返回失败。

4.1.14. read

Usage: read fd [-o outFileName] size

Description: 通过 open 获取的 file descriptor, 读取一个已经打开的文件。
通过-o 指定将内容显示在 shell 或写入 file 文件。

size 指定读取字节数

file 指定写入的文件

Exit status: 文件描述符不存在或超出范围, 写入的文件已存在, 返回失败。

4.1.15. auto

Usage: auto

Description: 执行自动测试。在系统启动初期帮助测试, 测试不一定所有命令都是正确的。

Exit status: 默认成功

4.1.16. chmod

Usage: chmod mode file

Description: 修改文件 file 的文件属性为 mode

Exit status: 文件名过长, 目录路径不存在, 准备修改一个没有权限的文件, 返回失败。

4.1.17. adduser(root)

Usage: adduser name

Description: 添加一个名为 name 的普通用户, 随后输入用户密码

Exit status: name 用户已存在或 name 过长, 返回失败。

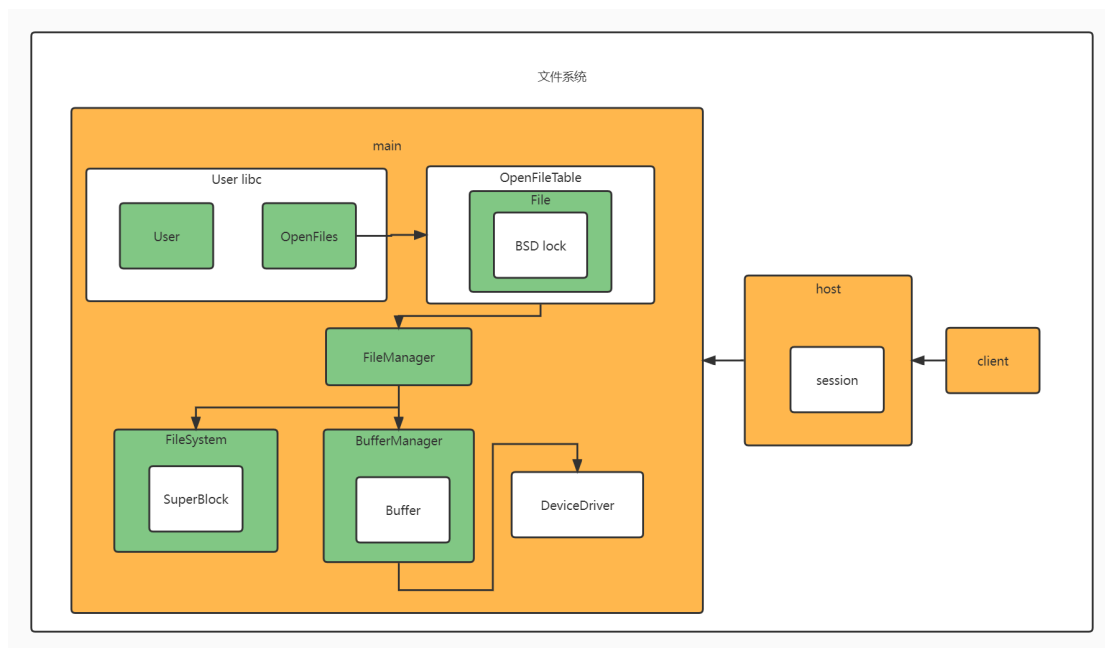
4.2. 程序功能

实验模拟 UNIX 文件操作系统实现了多用户下文件的一系列操作。具体可以实现的功能如下所示:

- 多用户下文件锁
- 多用户下文件、文件夹的创建、删除
- 多用户下文件的打开、关闭、读、写、读写指针移动
- 当前目录文件列表的显示
- 多用户下更改文件属性

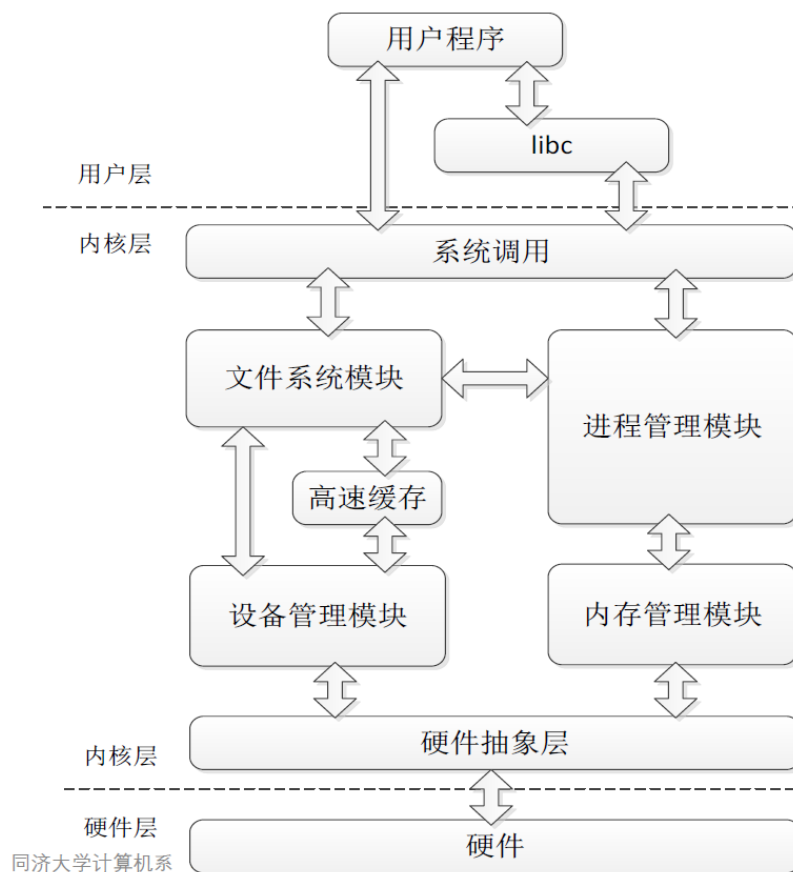
- 用户的创建
- 用户的登录、退出
- 显示所有用户信息
- 格式化文件卷
- 退出文件系统
- 帮助文档

4.3. 程序结构



5. 功能设计

5.1. 模块分解



UNIX 系统架构分为用户层、内核层、硬件层。据此，可以将多用户二级文件系统分为主要的三层，用户交互层、二级文件系统、虚拟硬盘层。

5.2. 用户交互层

用户程序部分：主要负责显示命令行交互系统。用户输入信息处理、将用户输入命令转化为内核操作、反馈信息输出、错误反馈等。

libc 部分：实现系统调用接口函数，提供命令对应接口函数并调用内核实现。

5.3. 二级文件系统

文件系统部分：主要负责文件相关的一系列操作。提供创建、删除文件、文件夹，显示文件列表，打开、关闭文件，读写文件、更改文件指针、更改文件权限等操作的相关接口。通过调用设备管理模块读取硬盘上的数据结构并转换为内存中存储的文件系统数据结构。

设备管理部分：包含子模块高速缓存。主要负责 SuperBlock、Inode、Block 的初始化、申请和释放。按照 512 字节一个 Block 块读取文件系统并将部分块缓存到高速缓存队列中。实现 Inode 相关的索引、读取、上锁、解锁、释放 Block 块等操作。

用户管理部分：读取系统花名册文件/etc/passwd 初始化用户管理模块。主要负责用户相关的一系列操作。提供用户登录、用户登出、创建用户、删除用户、输出用户列表信息等操作的相关接口。

5.4. 虚拟硬盘层

第一级文件系统，读取 myDisk.img 虚拟磁盘文件，提供虚拟磁盘访问接口，读取和写入 myDisk.img。

6. 主要数据结构

6.1. Inode 结构体

```
1.    unsigned int i_flag;          // 状态的标志位，定义见 enum INodeFlag
2.    unsigned int i_mode;          // 文件工作方式信息
3.
4.    int          i_count;          // 引用计数
5.    int          i_nlink;          // 文件联结计数，即该文件在目录树中不同路径
    名的数量
6.
7.    short      i_dev;              // 外存 INode 所在存储设备的设备号
8.    int         i_number;          // 外存 INode 区中的编号
9.
10.   short      i_uid;              // 文件所有者的用户标识数
11.   short      i_gid;              // 文件所有者的组标识数
12.
13.   int         i_size;              // 文件大小，字节为单位
14.   int         i_addr[10];         // 用于文件逻辑块好和物理块好转
    换的基本索引表
15.
16.   int         i_lastr;            // 存放最近一次读取文件的逻辑块号，用于判断
    是否需要预读
```

Inode 结构体重要成员变量：

i_mode：表示 Inode 是文件还是目录；访问权限，由 3 个 3 位权限组成，本次主要使用文件的文件主访问权限和其他用户访问权限。2 位权限分别位 w 写权限、r 读权限。OWNER_R 为文件主可读，OWNER_W 为文件主可写，ELSE_R 为其他用户可读，ELSE_W 为其他用户可写。

i_uid：文件主的 uid

6.2. User 结构体

```
1.     Inode* cdir;                /* 指向当前目录的 Inode 指针 */
2.     Inode* pdir;                /* 指向父目录的 Inode 指针 */
3.
4.     DirectoryEntry dent;        /* 当前目录
   的目录项 */
5.     char dbuf[DirectoryEntry::DIRSIZ]; /* 当前路径分量 */
6.     string curDirPath;          /* 当前工作
   目录完整路径 */
7.
8.     string dirp;                /* 系统调用参数(一般用于
   Pathname)的指针 */
9.     long arg[5];                /* 存放当前系统调用参数 */
10.                                /* 系统调用相关成员 */
11.     unsigned int ar0[5];        /* 指向核心栈现场保护区中 EAX 寄存器
   存放的栈单元，本字段存放该栈单元的地址。
   在 V6 中 r0 存放系统调用的返回值给用户程序，
   x86 平台上使用 EAX 存放返回值，替代 u.ar0[R0] */
15.    ErrorCode u_error;          /* 存放错误码 */
16.
17.    OpenFiles ofiles;           /* 进程打开文件描述符表对象 */
18.
19.    IOParameter IOParam;        /* 记录当前读、写文件的偏移量，用户目标区域和剩余
   字节数参数 */
20.
21.    FileManager* fileManager;
22.
23.    short i_uid;                 // 文件所有者的用户标识数
24.    string username;
25.    string ls;
```

User 结构体重要成员变量：

username：存储用户名，用于显示在命令行界面上

i_uid：保存用户 uid

6.3. DiskDriver 结构体

实现虚拟硬盘驱动。

fp：指向虚拟硬盘文件 myDisk.img 的 fd。

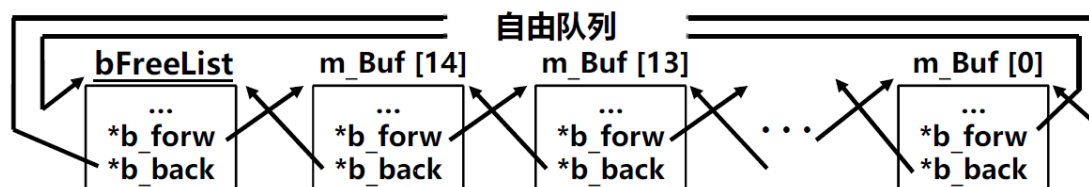
BLOCK_SIZE=512：默认硬盘每一 Block 块设为 512B

read()：模拟硬盘读取过程，每次读取一个 Block

write(): 模拟硬盘写入, 每次写入一个 Block

6.4. BufferManager 结构体

可以参考 Unix V6++ 中 BufferManager 的实现, 由于 DiskDriver 虚拟硬盘的使用, 需要重写某些方法, 例如 Bread, Breada, Bwrite, Bflush 等方法。使用 LRU 方法维护缓存块自由队列。



6.5. File 结构体

```
1.  unsigned int flag;           /* 对打开文件的读、写操作要求 */
2.  int count;                   /* 当前引用该文件控制块的进程数量 */
3.  INode* inode;                /* 指向打开文件的内存 INode 指针 */
4.  int offset;                  /* 文件读写位置指针 */
5.  short lock_uid;
```

记录在打开文件表中, 存储打开文件信息。其包含的成员如下:

inode: 文件的 inode, 用于访问 Inode 表, 实现读写请求。也是 File 结构体唯一对应 Inode 的判断条件。

count: 文件的引用计数。i_count>1 时, 代表打开使用, 此时 lock_uid 表示对其上锁的 uid 用户。close 时, 如果 i_count=1, 代表全部关闭了。

lock_uid: 文件打开用户, 当文件系统锁的粒度为单个文件时, 表示当前使用文件的用户, 不允许其他用户再打开。

6.6. Session 结构体

```
1.  class Session
2.  {
3.  public:
4.      User s_User;
5.      int testNo;
6.      int autoFlag;
7.      int login;
8.      stringstream isstream, osstream;
9.      Session() {
```

```

10.         autoFlag = 0;
11.         testNo = 0;
12.         login = 0;
13.     }
14.     ~Session() {
15.
16.     }
17. };

```

s_User 用于切换内核运行的 user 对象。

Stringstream 保留用户对应的输入输出流。

autoFlag、testNo 记录自动运行信息。

7. 详细设计

7.1. Socket 流程

通过 socket 连接收发信息，通过 p_session 区分不同登录的用户

```

1. Session* p_session = new Session;
2. int nSend = send(clientSock, tmp.str().c_str(), tmp.str().length(), 0);
3. const stringstream& ss = mainloop("", p_session);
4. nSend = send(clientSock, ss.str().c_str(), ss.str().length(), 0);
5. //向连接的客户端收发数据
6. while (true)
7. {
8.     auto begin = listSock.begin();
9.     auto end = listSock.end();
10.    //收发数据
11.    char buffer[1024]{ 0 };
12.    int nRecv = recv(clientSock, buffer, 1024, 0);
13.    if (nRecv <= 0)
14.        break;
15.    buffer[nRecv - 1] = '\0';
16.    cout << "from socket: " << clientSock << " received " << nRecv << " length of
        data: " << buffer << endl;
17.    const stringstream& ss = mainloop(buffer, p_session);
18.    //for (; begin != end; begin++)
19.    //int nSend = send(*begin, "acknowledged", strlen("acknowledged"), 0);
20.    int nSend = send(clientSock, ss.str().c_str(), ss.str().length(), 0);
21. }

```

7.2. 指令的解析和运行 libc 入口

User 结构实现指令解析和所有指令的入口：以下以登录和 fformat 为例

```
1. if (!p_Session->login) {
2.     if (cmd == "login") {
3.         user->Validate(arg1, arg2);
4.         if(user->u_error == User::U_NOERROR){
5.             p_Session->login = true;
6.             user->username = arg1;
7.             ostream << "login success!" << endl;
8.         }
9.         else{
10.            ostream << "wrong username or password!" << endl;
11.        }
12.    }
13.    else{
14.        ostream << "use login command to log in." << endl;
15.    }
16. }
17. else
18. {
19.     if (cmd == "help") {
20.         man(arg1.empty() ? "help" : arg1);
21.     }
22.     else if (cmd == "fformat") {
23.         g_OpenFileTable.Format();
24.         g_INodeTable.Format();
25.         g_BufferManager.FormatBuffer();
26.         g_FileSystem.FormatDevice();
27.         g_INodeTable = INodeTable();
28.         g_FileManager = FileManager();
29.         g_User = User();
30.         g_User.username = "root";
31.         user->Mkdir("bin");
32.         user->Mkdir("etc");
33.         user->Mkdir("home");
34.         user->Mkdir("dev");
35.         user->Cd("etc");
36.         user->Create("passwd", "-rw");
37.         user->Open("passwd", "-rw");
38.         unsigned int fd = user->ar0[User::EAX];
39.         user->Seek(to_string(fd), "0", "0");
40.         user->Write(to_string(fd), "", "1 root root");
41.         user->Close(to_string(fd));
42.         user->Cd("..");
```

```
43.     }
44.     else if (cmd == "exit") {
45.         exit(0);
46.     }
```

7.3. Login 密码验证

验证密码，读/etc/passwd 文件，使用 CheckPwd 读取所有用户密码信息并匹配验证。

```
1. void User::Validate(string username,string password) {
2.     if (!checkPathName("/etc/passwd")) {
3.         return;
4.     }
5.     int md = FileMode("-r");
6.     if (md == 0) {
7.         ostream << "this mode is undefined ! \n";
8.         return;
9.     }
10.
11.     arg[1] = md;
12.     fileManager->Open();
13.     if (u_error != U_NOERROR) {
14.         u_error = U_NOERROR;
15.         return;
16.     }
17.     unsigned int fd = ar0[EAX];
18.     Seek(to_string(fd), "0", "0");
19.
20.     int usize = 1024;
21.     char* buffer = new char[usize+1];
22.     arg[0] = fd;
23.     arg[1] = (long)buffer;
24.     arg[2] = usize;
25.     fileManager->Read();
26.     if (IsError())
27.         return;
28.     Close(to_string(fd));
29.
30.     buffer[ar0[User::EAX]] = '\0';
31.     short uid = CheckPwd(username, password, buffer);
32.     delete[]buffer;
33.
34.     if (uid==0)
```

```

35.     {
36.         u_error = User::U_EACCES;
37.     }
38.     i_uid = uid;
39.     return;
40. }

```

7.4. 添加新用户

添加用户密码，读入所有信息并比较，决定更新或添加

```

1.  short AddPwd(string username, string password, char* buffer) {
2.      stringstream in(buffer),out;
3.      short uid,max_uid=1,user_id=0;
4.      string name, pwd;
5.      while (true)
6.      {
7.          in >> uid >> name >> pwd;
8.          if (in.fail())
9.          {
10.             if (user_id == 0) {
11.                 user_id = max_uid + 1;
12.                 out << user_id << ' ' << username << ' ' << password << '\n';
13.                 break;
14.             }
15.         }
16.         else if(name== username)
17.         {
18.             max_uid = uid;
19.             user_id = max_uid;
20.             out << uid << ' ' << name << ' ' << password << '\n';
21.         }
22.         else
23.         {
24.             max_uid = uid;
25.             out << uid << ' ' << name << ' ' << pwd<<'\n';
26.         }
27.     }
28.     strcpy(buffer, out.str().c_str());
29.     return user_id;
30. }

```

7.5. BSD 锁

允许多个 fd 但只有一个 File 结构体。BSD 锁，i_count>1 时，代表打开使用，此时 lock_uid 表示对其上锁的 uid 用户。close 时，如果 i_count=1，代表全部关闭了

```
1. if (inodeAddr) {
2.     for (int i = 0; i < OpenFileTable::MAX_FILES; ++i) {
3.         if ((unsigned int)this->m_File[i].inode == inodeAddr) {
4.             if
5.                 (! (this->m_File[i].lock_uid==0 || this->m_File[i].lock_uid==u.i_uid)){
6.                     u.u_error = User::U_EACCES;
7.                     return NULL;
8.                 }
9.             else{
10.                 this->m_File[i].lock_uid = u.i_uid;
11.                 u.ofiles.SetF(fd, &this->m_File[i]);
12.                 this->m_File[i].count++;
13.                 return (&this->m_File[i]);
14.             }
15.         }
16.     }
17. void OpenFileTable::CloseF(File* pFile) {
18.     pFile->count--;
19.     if (pFile->count <= 1) {
20.         pFile->lock_uid = 0;
21.     }
22. if (pFile->count <= 0) {
23.     g_INodeTable.IPut(pFile->inode);
24. }
25. }
```

7.6. 虚拟磁盘读写

在 buffer 块读出和换入过程中，使用 DeviceDriver 的读写功能，写入到磁盘中文件 img。

```
1. void DeviceDriver::write(const void* buffer, unsigned int size, int offset,
2.     unsigned int origin) {
3.     if (offset >= 0) {
4.         fseek(fp, offset, origin);
5.     }
6.     fwrite(buffer, size, 1, fp);
7. }
```

```
6. }
7.
8. void DeviceDriver::read(void* buffer, unsigned int size, int offset, unsigned int
   origin) {
9.     if (offset >= 0) {
10.         fseek(fp, offset, origin);
11.     }
12.     fread(buffer, size, 1, fp);
13. }
```

8. 调试分析

8.1. 自动测试指令

自动测试指令涵盖了单用户文件系统所有的指令的测试,多用户指令测试使用手动方式进行。

```
1. static const char* test[] = {
2.     "cd /home",
3.     "mkdir testdir",
4.     "create testfile1",
5.     "cd ..",
6.     "create /home/testfile2",
7.     "cd home",
8.     "ls -l",
9.     "chmod testfile1 700",
10.    "rm /home/testfile2",
11.    "ls -l",
12.    "open testfile1 -rw",
13.    "write 8 -i testInputFile.txt 50",
14.    "seek 8 30 0",
15.    "read 8 20",
16.    "seek 8 30 0",
17.    "read 8 -o testOutputFile.txt 20",
18.    "seek 8 -20 1",
19.    "read 8 100",
20.    "cd testdir",
21.    "ls",
22.    "create testpng",
23.    "open testpng -rw",
24.    "write 10 -i test.png 50000",
25.    "seek 10 0 0",
26.    "read 10 -o testOutput.png 50000",
27.    "adduser name1 pass1",
```



```
28.     ""
29.     };
```

8.2. 编译和运行

8.2.1. 集成环境编译

推荐使用 VS2019 集成环境进行编译，分别将两个文件夹创建为两个项目，并分别编译得到 Host.exe 和 Cilent.exe。

8.2.2. Makefile

由于 Makefile.win 文件预置了环境路径，所以首先需要在 Host 和 Cilent 文件夹下，修改 Makefile.win 对应的 g++ 环境路径，才能正确运行编译。需要修改以下的 LIBS、INCS、CXXINCS 目录路径。

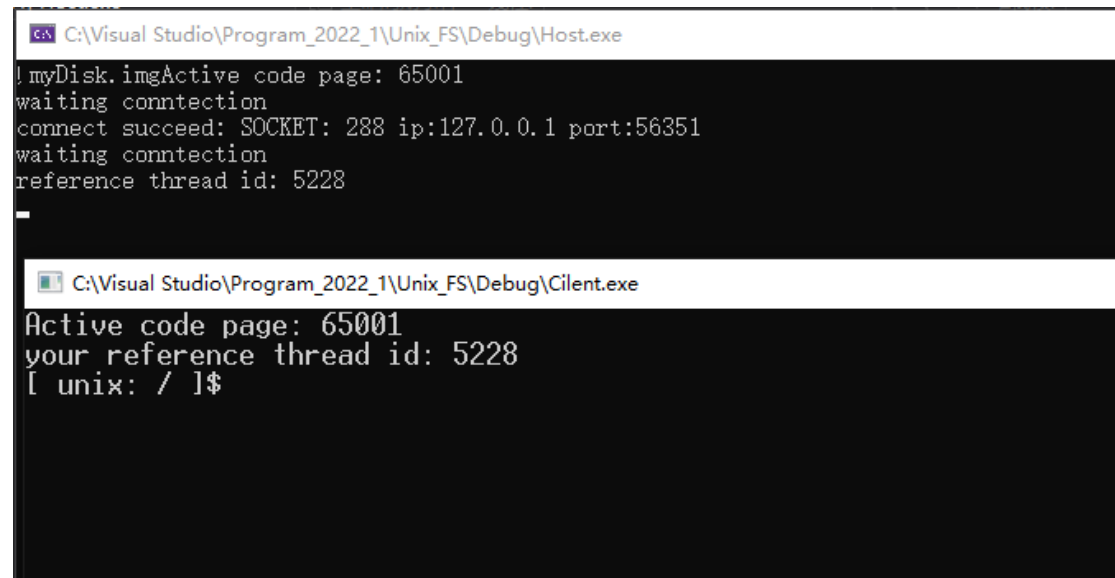
```
1. CPP      = g++.exe -D__DEBUG__
2. CC       = gcc.exe -D__DEBUG__
3. WINDRES  = windres.exe
4. OBJ      = client.o
5. LINKOBJ  = client.o
6. LIBS     = -L"C:/C++/Dev C++/Dev-Cpp/MinGW64/x86_64-mingw32/lib32" -lws2_32 -
m32 -g3
7. INCS     = -I"C:/C++/Dev C++/Dev-Cpp/MinGW64/include" -I"C:/C++/Dev C++/Dev-
Cpp/MinGW64/x86_64-mingw32/include" -I"C:/C++/Dev C++/Dev-
Cpp/MinGW64/lib/gcc/x86_64-mingw32/4.9.2/include"
8. CXXINCS  = -I"C:/C++/Dev C++/Dev-Cpp/MinGW64/include" -I"C:/C++/Dev C++/Dev-
Cpp/MinGW64/x86_64-mingw32/include" -I"C:/C++/Dev C++/Dev-
Cpp/MinGW64/lib/gcc/x86_64-mingw32/4.9.2/include" -I"C:/C++/Dev C++/Dev-
Cpp/MinGW64/lib/gcc/x86_64-mingw32/4.9.2/include/c++"
9. BIN      = cilent.exe
10. CXXFLAGS = $(CXXINCS) -m32 -std=c++11 -g3
11. CFLAGS   = $(INCS) -m32 -std=c++11 -g3
12. RM       = rm.exe -f
13.
14. .PHONY: all all-before all-after clean clean-custom
15.
16. all: all-before $(BIN) all-after
17.
18. clean: clean-custom
19.     ${RM} $(OBJ) $(BIN)
20.
21. $(BIN): $(OBJ)
22.     $(CPP) $(LINKOBJ) -o $(BIN) $(LIBS)
23.
24. client.o: client.cpp
25.     $(CPP) -c client.cpp -o client.o $(CXXFLAGS)
```

运行以下进行编译

```
1. mingw32-make.exe -f ".\Makefile.win" clean all
```

8.2.3. 运行

8.2.3.1. 首先打开 host.exe, 然后开启 cilent.exe 使用用户登录

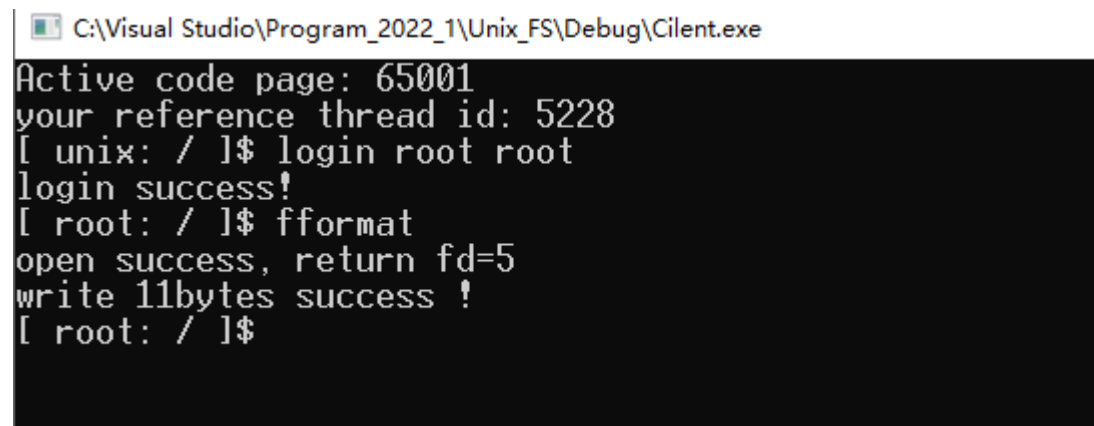


The screenshot shows two overlapping command prompt windows. The top window, titled 'C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Host.exe', displays the following text: 'myDisk.imgActive code page: 65001', 'waiting conntection', 'connect succeed: SOCKET: 288 ip:127.0.0.1 port:56351', 'waiting conntection', and 'reference thread id: 5228'. The bottom window, titled 'C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe', displays: 'Active code page: 65001', 'your reference thread id: 5228', and a shell prompt '[unix: /]\$'.

```
C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Host.exe
myDisk.imgActive code page: 65001
waiting conntection
connect succeed: SOCKET: 288 ip:127.0.0.1 port:56351
waiting conntection
reference thread id: 5228

C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe
Active code page: 65001
your reference thread id: 5228
[ unix: / ]$
```

8.2.3.2. Login 连接和格式化硬盘



The screenshot shows a single command prompt window titled 'C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe'. It displays the following text: 'Active code page: 65001', 'your reference thread id: 5228', a shell prompt '[unix: /]\$', and the execution of 'login root root' (resulting in 'login success!'), 'fformat' (resulting in 'open success, return fd=5' and 'write 11bytes success !'), and another shell prompt '[root: /]\$'.

```
C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe
Active code page: 65001
your reference thread id: 5228
[ unix: / ]$ login root root
login success!
[ root: / ]$ fformat
open success, return fd=5
write 11bytes success !
[ root: / ]$
```

8.2.3.3. auto 运行自动测试

```
C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe
[ root: /home/ ]$ seek 8 30 0
[ root: /home/ ]$ read 8 20
read 20 bytes success :
33333333
44444444

[ root: /home/ ]$ seek 8 30 0
[ root: /home/ ]$ read 8 -o testOutputFile.txt 20
read 20 bytes success :
read to testOutputFile.txt done !
[ root: /home/ ]$ seek 8 -20 1
[ root: /home/ ]$ read 8 100
read 20 bytes success :
33333333
44444444

[ root: /home/ ]$ cd testdir
[ root: /home/testdir/ ]$ ls

[ root: /home/testdir/ ]$ create testpng
[ root: /home/testdir/ ]$ open testpng -rw
open success, return fd=10
[ root: /home/testdir/ ]$ write 10 -i test.png 50000
write 50000bytes success !
[ root: /home/testdir/ ]$ seek 10 0 0
[ root: /home/testdir/ ]$ read 10 -o testOutput.png 50000
read 50000 bytes success :
read to testOutput.png done !
[ root: /home/testdir/ ]$ adduser name1 pass1
[ root: /home/testdir/ ]$
```

8.2.3.4. 打开另一个 client，登录测试并发

<pre>C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe [root: /home/]\$ seek 8 30 0 [root: /home/]\$ read 8 20 read 20 bytes success : 33333333 44444444 [root: /home/]\$ seek 8 30 0 [root: /home/]\$ read 8 -o testOutputFile.txt 20 read 20 bytes success : read to testOutputFile.txt done ! [root: /home/]\$ seek 8 -20 1 [root: /home/]\$ read 8 100 read 20 bytes success : 33333333 44444444 [root: /home/]\$ cd testdir</pre>	<pre>C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe Active code page: 65001 your reference thread id: 16960 [unix: /]\$ login name1 pass1 login success! [name1: /]\$ ls bin etc home dev [name1: /]\$</pre>
---	---

8.2.3.5. 使用另一个 client 尝试打开无权限文件

C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe

```
[ name1: /home/ ]$ ls -l
d----- 32B testdir
-rwx----- 50B testfile1

[ name1: /home/ ]$ open testfile1 -r
errno = 13 Permission denied
[ name1: /home/ ]$
```

8.2.3.6. 使用另一个 client 尝试被前一个 client 上锁的文件

C:\Visual Studio\Program_2022_1\Unix_FS\Debug\Cilent.exe

```
[ name1: /home/testdir/ ]$ ls -l
-rw----rw- 50000B testpng

[ name1: /home/testdir/ ]$ open testpng -r
File locked by other user:
errno = 13 Permission denied
[ name1: /home/testdir/ ]$
```

8.2.3.7. 前一个 client 关闭文件后（解锁），另一个 client 才能访问

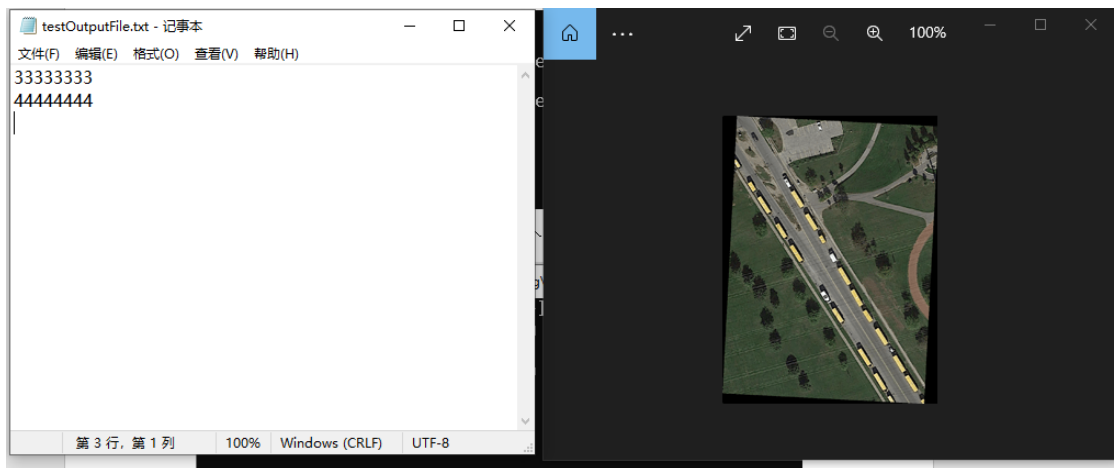
```
[ root: /home/testdir/ ]$ ls
-rw----rw- 50000B testpng

[ root: /home/testdir/ ]$ create testpng
[ root: /home/testdir/ ]$ open testpng -rw
open success, return fd=10
[ root: /home/testdir/ ]$ write 10 -i test.png
write 50000bytes success !
[ root: /home/testdir/ ]$ seek 10 0 0
[ root: /home/testdir/ ]$ read 10 -o testOutput
read 50000 bytes success :
read to testOutput.png done !
[ root: /home/testdir/ ]$ adduser name1 pass1
[ root: /home/testdir/ ]$ close 10

[ name1: /home/testdir/ ]$ open testpng -r
File locked by other user:
errno = 13 Permission denied
[ name1: /home/testdir/ ]$ open testpng -r
File locked by other user:
errno = 13 Permission denied
[ name1: /home/testdir/ ]$ open testpng -r
open success, return fd=0
[ name1: /home/testdir/ ]$
```

8.2.3.8. Exit 退出即可保存文件到 img

8.2.3.9. 测试输出的文件



9. 文件锁调研

9.1. 简介

Linux 支持两种主要的文件锁：咨询锁和强制锁。

咨询锁是 Unix 风格的锁，是建议性的锁。它们仅在进程显式获取和释放锁时起作用，由进程主动判断是否存在锁。如果进程不知道锁，它们将被忽略。

9.2. 咨询锁

9.2.1. Linux 中有几种类型的咨询锁

BSD 锁 (flock)

POSIX 记录锁 (fcntl, lockf)

Open file description 打开文件描述锁 (fcntl)

除 lockf 函数外的所有锁都是读写锁，即支持独占和共享模式。

9.2.2. 基本特征

所有锁都支持阻塞和非阻塞操作。

仅允许对文件进行锁定，但不允许对目录进行锁定。

当进程退出或终止时，锁会自动移除。系统内核保证如果获取了锁，获取锁的进程还活着。

	BSD 锁	lockf 函数	POSIX 记录锁	打开文件描述锁
有关联	文件对象	[i-node, pid] 对	[i-node, pid] 对	文件对象
应用于一个字 节范围	不	是的	是的	是的
支持独占和共	是的	不	是的	是的

享模式				
原子模式开关	不	-	是的	是的

9.2.3. BSD 锁 (flock)

最简单和最常见的文件锁，函数为 `int flock(int fd, int operation);`

- 特征：

在 POSIX 中未指定，但在各种 Unix 系统上广泛可用

与文件对象相关联、总是锁定整个文件

不保证锁定模式（独占和共享）之间的原子切换

NFS 上的 flock() 锁使用 fcntl() POSIX 记录整个文件的字节范围锁来模拟（除非在 NFS 挂载选项中禁用了模拟）

- 锁获取与文件对象相关联，即：

重复的文件描述符，例如使用 dup2or 创建 fork 的，共享锁获取；

独立的文件描述符，例如使用两个 open 调用创建的（即使是同一个文件），不共享锁获取；

这意味着使用 BSD 锁，线程或进程不能在相同或重复的文件描述符上同步，但是，两者都可以在独立的文件描述符上同步。

- flock() 不保证原子模式切换。从手册页：

转换锁（共享到独占，反之亦然）不能保证是原子的：先移除现有锁，然后建立新锁。在这两个步骤之间，可能会授予另一个进程的挂起锁请求，如果指定了 LOCK_NB，则转换要么阻塞，要么失败。

9.2.4. POSIX 记录锁 (fcntl)

POSIX 记录锁，也称为与进程相关的锁，函数为 `int fcntl(int fd, int cmd, ... /* arg */);`

- 特征：

在 POSIX（基本标准）中指定

可以应用于一个字节范围 (byte range)

与 [i-node, pid] 而不是文件对象相关联

保证锁定模式（独占和共享）之间的原子切换

- 锁获取与 [i-node, pid] 对相关联，即：

由同一进程为同一文件打开的文件描述符共享锁获取（甚至是独立的文件描述符，例如使用两次 open 调用创建）；

不同进程打开的文件描述符不共享锁获取；

- 这意味着使用 POSIX 记录锁，可以同步进程，但不能同步线程。属于同一个进程的所有线程总是共享一个文件的锁获取，这意味着：

某个线程通过某个文件描述符获取的锁可能被另一个线程通过另一个文件描述符释放；

当任何线程调用 close 引用给定文件的任何描述符时，即使有其他打开的描述符引用该文件，也会为整个进程释放锁。

9.2.5. lockf 函数

lockf 函数是 POSIX 记录锁的简化版本，函数为 `int lockf(int fd, int cmd, off_t len);`

- 特征：

lockf POSIX 没有指定与其他类型的锁之间的交互。在 Linux 上，lockf 只是 POSIX 记录锁的包装器，直接使用 POSIX 记录锁实现。

由于 lockf 锁与[i-node, pid]对相关联，因此它们具有与上述 POSIX 记录锁相同的问题。

9.2.6. Open file description 打开文件描述锁 (fcntl)

打开文件描述锁是特定于 Linux 的，它结合了 BSD 锁和 POSIX 记录锁的优点。函数为 `int fcntl(int fd, int cmd, ... /* arg */);` 与 POSIX 记录锁 API 相同，但是唯一的区别在于 fcntl 命令名称，使用 Open file description 的 cmd 代替 POSIX 记录锁的 cmd：

F_OFD_SETLK 代替 F_SETLK

F_OFD_SETLKW 代替 F_SETLKW

F_OFD_GETLK 代替 F_GETLK

9.3. 强制锁定

Linux 对强制文件锁定的支持有限。通过满足以下条件时，将为文件激活强制锁定：

该分区是使用该 mand 选项挂载 (mount) 的。

文件的 set-group-ID 位为 on，而 group-execute 位为 off。(set-group-ID 位的常规含义是在 group-execute 位打开时提升特权，而在 group-execute 位关闭时启用强制锁定的特殊含义。)

获得了 POSIX 记录锁。

- 当强制锁被激活时，它会影响文件上的常规系统调用：

当获得独占或共享锁时，所有修改文件的系统调用（例如 open() 和 truncate()）都会被阻塞，直到锁被释放。

当获得独占锁时，所有从文件中读取的系统调用（例如 `read()`）都会被阻塞，直到锁被释放。然而，Linux 当前未实现强制锁定，特别是 `read()` 当与 `write()` 同时获取锁时，使用 `mmap()` 时，产生竞争冒险

- 特征：

特定于 Linux，在 POSIX 中未指定

可以应用于一个字节范围

与文件对象相关联

保证锁定模式（独占和共享）之间的原子切换

因此，打开文件描述锁结合了 BSD 锁和 POSIX 记录锁的优点：它们提供了锁定模式之间的原子切换，以及同步线程和进程的能力。

10. 总结

本次实验完成了二级文件系统，实现了将宿主机中的文件存入、取出虚拟磁盘，创建新目录、文件，删除文件，接收用户输入的文件操作命令，测试文件系统和测试文件系统和 API。实现 buffer 缓存，实现多用户并发访问，实现 BSD 文件锁。

通过本次操作系统课程设计，我觉得无论是系统分析和设计架构的能力，编程语言 C++ 的运用和理解，还是寻找项目代码 bug 的能力，分析错误调试程序的能力都有较大的提升。