《操作系统原理及应用》课程设计报告

虚拟文件系统模拟

|  |  |  |
| --- | --- | --- |
| 课程目标1 | 课程目标2 | 课程目标3 |
|  |  |  |

学院（系）： 计算机科学与工程学院

班 级： 117030704 学号 11703990806

学生姓名： 汤荣杰

同组人员：安鹤松、李东、吴思涵丹、杨森

鲁月、尹明珠、李一凡

目录

**[一、课程设计的内容，要求及实现情况说明 1](#_Toc15932)**

[1.1 课程设计的内容及要求 1](#_Toc22860)

[1.2 课程设计的完成情况 1](#_Toc31585)

**[二、设计方案分析 1](#_Toc5254)**

[2.1 文件系统总体构建方案 1](#_Toc32607)

[2.2 目录/普通文件项的设计 2](#_Toc16041)

[2.3 多种缓冲区的设计 3](#_Toc22843)

[2.4 功能模块的设计 3](#_Toc14435)

[2.5 方案的合理性与可行性分析 4](#_Toc9663)

**[三、选择和使用的现代工具 5](#_Toc14050)**

**[四、课题实现 5](#_Toc19139)**

[4.1 格式化操作 5](#_Toc119)

[4.2 用户模块 6](#_Toc30148)

[4.3 目录操作模块 8](#_Toc21331)

[4.4 普通文件操作模块 13](#_Toc16433)

**[五、课题实现结果 17](#_Toc28453)**

[5.1 用户模块实现结果 17](#_Toc12050)

[5.2 目录操作模块实现结果 18](#_Toc16541)

[5.3 普通文件模块实现结果 19](#_Toc12535)

[5.4 系统格式化实现结果 21](#_Toc26150)

**[六、 结果分析与总结 22](#_Toc5203)**

**[七、 参考文献 23](#_Toc16212)**

**一、课程设计的内容，要求及实现情况说明**

**1.1 课程设计的内容及要求**

设计实现一个虚拟文件系统，所设计的文件系统要求如下：

1. 有文件存储空间的管理和目录管理。
2. 具备文件系统格式化、用户登录、用户注销、显示目录内容、文件创建、文件删除、目录创建和目录删除等功能。
3. 创建的文件不要求格式和内容，但要有相应的权限控制。
4. 要有目录结构的设计和磁盘空间的分配与回收等文件系统核心功能。

**1.2 课程设计的完成情况**

本次课程设计实现了一个多级目录的虚拟文件系统，其构成主要参考了EXT2文件系统。除完成1.1中所述的功能及要求外，还完成了以下内容：

1. 文件打开，文件关闭功能，即模拟文件的打开和关闭过程；
2. 读文件，写文件功能，在文件打开的情况下，能对文件进行内容的写入，能对已有内容进行读取；
3. 用户注册功能，模拟系统能够进行多个用户的注册。

**二、设计方案分析**

**2.1 文件系统总体构建方案**

在对文件系统的总体框架进行设计时，主要参考了EXT2文件系统的结构。其结构比较简单，且考虑到了文件系统访存速度等效率问题。其主要特点之一是使用inode节点记录文件的基础信息，加快了文件系统的速度。

在方案设计时，首先，通过建立了一个文件来模拟物理存储空间。然后定义了超级块，数据块位图，inode节点位图，inode表，数据块的起始位置及大小。其中超级块，数据块位图和inode节点位图的大小定义为512字节。而inode表和数据块的大小则可根据实际需要进行大小的修改，此处将数据块的起始位置定义在10240字节处，保证了足够的inode表的空间，且定义的inode的个数和数据块的个数都为4096。对于每一个inode，定义其大小为64字节，数据块的大小为512字节。总体的示意图如下：

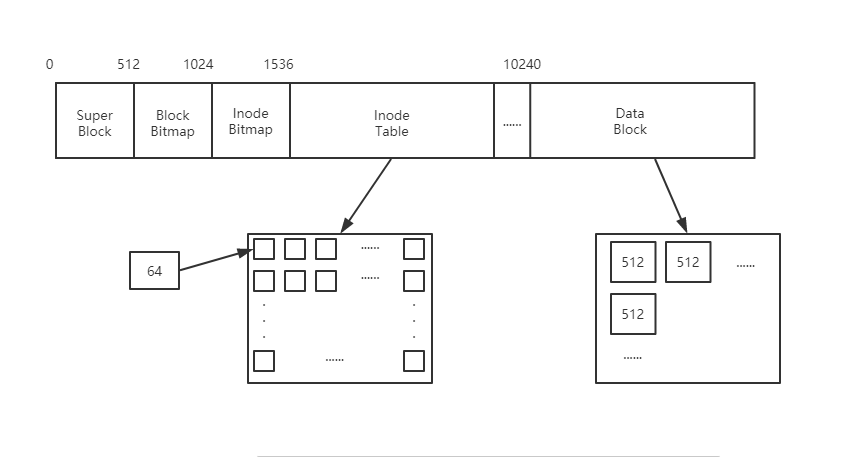


图2-1 文件系统总体构建示例图

对于每一部分，其相关信息和作用如下：

超级块：在该部分中，主要是对本组信息的描述。其中存储的信息包括组名，个分区的起始地址，空闲数据块的个数，空闲inode节点的个数等。记录这些信息，以方便对该组的存储空间进行了解并使用。

数据块位图：本区的主要是对数据区的使用情况进行了描述。数据区被分成大小为512字节的数据块，每一块是否使用都需要进行记录，本区则记录了该信息。当对应的位为1时，代表数据区中的对应数据块已被使用，否则未被使用。

Inode位图：同数据块位图的作用一样，本区记录了inode节点表的使用情况。每一个inode空间都有一个位与之对应，记录该inode空间是否已经被分配，已分配则记录1，否则记录0.

Inode表：inode表被划分为许多个小的inode节点区域。每个文件都会对应一个inode节点，每个inode节点记录了其对应文件的访问权限，该文件对应的数据块的个数，如果是目录则还有当前目录下文件的大小之和。同时还有指向数据块的指针，这里定义了8个指针，即代表每个文件对应的数据最多占用8个数据块。

数据块区：数据块区的每个数据块大小都为512字节。其主要作用是存储目录文件数据以及普通文件数据。

**2.2 目录/普通文件项的设计**

在本课程设计的内容中，由于所有的操作都是基于文件的，而大多的都会是对目录进行操作，所以对目录文件或普通文件的结构进行了单独的设计。

目录/普通文件项和inode节点相同，也会和文件一一对应，其结构中记录了inode节点号，文件类型，文件名字以及文件的长度。目录/普通文件项的设计即相当于FCB的作用，能更方便的对文件系统进行管理，同时也提高了文件系统的效率。

在模拟时，当前记录的总是当前的相对路径，如果需要访问当前路径下的某个目录或者文件时。首先会根据当前的路径加载出当前目录下的目录/普通文件项，通过在该结构中查找需要访问的路径，从而找到与要访问的目录或者文件的信息以及其inode节点。

**2.3 多种缓冲区的设计**

在设计的方案中，还包括了多个缓冲区的定义与设计。分别是数据缓冲区、位图缓冲区、inode节点缓冲区、目录项缓冲区、已打开文件缓冲区。对于缓冲区的设计，虽然有多种类型，但它们的作用都是相同的，即都是作为中间介质，对文件的相关信息进行暂时的存储，方便文件信息的使用和管理，减少了文件系统访问内存的次数，提高了文件系统的读写效率，同时也对存储的信息提供了一定的保护。

**2.4 功能模块的设计**

对于该虚拟文件系统模块的设计，主要确定了用户模块，目录操作模块和普通文件操作模块。对于虚拟文件的系统的格式化，只需要输入格式化的命令即可。格式化时，则对文件的多个区进行数据的删除，包括数据区的内容清除，位图的置位等。功能模块的关系图如下：

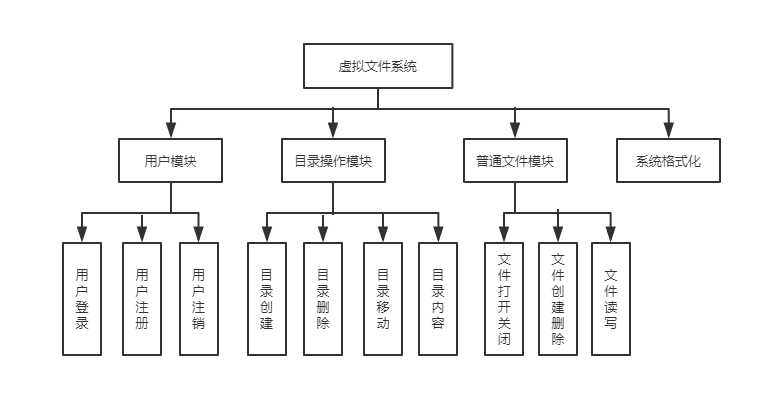


图2-2 文件系统功能模块图

**2.4.1 用户模块**

由题目要求可知，用户方面的操作包括用户登录，用户注销以及用户对文件的权限控制，此外则增加了用户注册的功能。

该模块的设计，保证了用户的正常处理操作，能够简单的模拟文件系统用户的角色。对于用户的权限设置，需要对每一用户指定相应的权限，从而能保证文件的信息的安全，使得用户的访问能安全进行。

**2.4.2 目录操作模块**

该模块中，定义用户对目录的相关操作，包括目录移动，目录创建，目录删除以及当前目录内容四个方面。

对于目录移动，用户可通过该操作的指令进行目录的移动，比如移动到当前目录下的某个目录。同时，还能向父目录进行移动，从而进行目录的相关管理和操作。此外，每个目录都应包含有当前目录和父目录两个目录文件，且能够对不存在的目录进行处理并给予错误信息提示。

对于目录的创建和删除，用户可通过使用目录创建和删除的命令对指定的目录进行创建和删除。创建时，需要对目录申请inode节点，同时进行初始化，即建立当前目录和父目录项，并初始赋予一个数据块。删除时，首先清空目录下的所有在数据块的信息，其次再对目录项的inode节点进行删除并对inode位图进行修改。

对于当前目录内容，用户可使用命令进行当前目录内容的输出。当使用该命令时，通过找到该目录的信息从而找到数据区的内容，对数据区的内容进行合理的输出查看，从而实现该操作。

**2.4.3 普通文件操作模块**

普通文件则没有像目录文件一样较多的结构，其核心则是文件内容。因此包含对文件和数据两者的操作。主要是文件的创建和删除，文件的打开和关闭，文件内容的读取和写入。

对于文件的创建和删除，和目录的操作部分相同，也需要申请一个inode节点及设置相应的节点信息。但普通文件不初始分配数据块，只有当写入时才进行分配。此外在创建文件时，需要对其判断是否已经存在同名文件，这样才能保证同一目录下没有相同名字的文件。

对于文件的打开和删除，在打开一个文件时，首先判断在已打开文件的缓冲区中是否存在该文件的inode节点号，只有不存在时才进行打开。打开后，则将打开文件的inode节点号加入已打开文件的缓冲区中，从而标记其已打开。在关闭一个文件时，其操作则和打开文件相反，首先判断其是否已经打开，若已打开，则从已打开文件的缓冲区中将其打开信息删除。

对于文件内容的读取和写入，通过数据缓冲区进行内容的操作。在读取文件内容时，通过文件的inode节点找到对应的数据块，对其内容进行依次读取，保存到缓冲区中，读取后再从缓冲区输出到控制台即可。在写入文件内容时，首先将用户的输入写入到缓冲区，同时判断用户的输入字节数是否超过定义的文件所能存放的大小，如果超过则停止输入，否则继续输入。输入完成后，对输入内容的大小进行判断，为其分配相应的数据块，保证能存放全部内容。分配数据块后，则将数据从缓冲区写入到数据块中，同时将文件的数据指针指向数据块。从而完成数据的写入操作。

**2.5 方案的合理性与可行性分析**

对于所设计的虚拟文件系统，其模拟了EXT2文件系统总体的文件存储，物理结构划分等方式，虽只进行了简单的模拟，但基础部分都包含在内，同时需要实现的功能都是必要的，是文件系统的必要组成部分，因此，该文件系统的结构划分，相关操作和功能都是合理的，并且各个组成部分虽进行了模块的划分，但都相互联系，没有单独的，不合理的部分。

另外，对于所有需要实现的功能，都是可以通过程序进行实现的，实现语言也有多种，例如C语言，Java语言等。不同的功能或结构模块，都可以通过定义相关的结构或者函数进行功能的实现，因此方案是可以得到具体的实现的，设计的方案是可行的。

综上所述，本虚拟文件的系统符合文件系统设计的要求，能够实现包括目录及文件的创建删除，文件打开和关闭以及文件读写等基本功能，因而对该文件系统的设计方案是合理的，可行的。

**三、选择和使用的现代工具**

根据对设计的方案进行分析后，对其实现所需的程序结构进行了分析。因其实现所需要的数据结构的定义，变量等的使用没有过于复杂的操作，因此采用C语言进行本虚拟文件系统的实现。并且，通过C语言进行实现也能更好对相应的命令以及操作进行过程性的展示。因使用C语言进行实现，故通过Windows平台即可对实现的程序进行运行以及相关的测试。

然而，通过C语言实现也具有一定的局限性，第一则是其结果的可视化不够直观和清晰，不能通过相应的点击之类的进行响应，只能输入相关的命令进行控制。第二则是程序量较多，没有其它语言写起来那么简单方便。

**四、课题实现**

**4.1 格式化操作**

initialize\_disk()

**\*功能说明**：磁盘初始化。本函数主要是通过清空缓冲区后，将清空的内容再写入磁盘从而达到清空磁盘的作用。

**\*实现过程**：该函数首先选定当前用户，将当前路径初始化为该用户的根目录。其次，对多个缓冲区进行清空操作，包括已打开文件缓冲区，数据缓冲区，inode节点缓冲区等。然后对磁盘进行清空，将清空后的各缓冲区写入磁盘从而达到清空磁盘数据的作用。最后则是对清空后的各分区进行一些初始化，最重要的则是对当前根目录的初始化，即增加当前目录和父目录信息，增加时，首先申请一个根目录的inode号并分配一个数据块，同时对位图信息进行修改，再对其目录下的两个初始目录添加名字等信息。

**\*关键程序**：

|  |
| --- |
| void initialize\_disk()  {  int i = 0;  printf("Creating the ext2 file system\n");  printf("Please wait ");  while(i < 1){  printf("... ");  sleep(1);  i++;  }  printf("\n");  cur\_user = users[cur\_index];  strcpy(current\_path, "[");  strcat(current\_path, cur\_user.username);  strcat(current\_path, "@localhost/");  //初始化上一次分配时的inode和block号  last\_alloc\_inode = 1;  last\_alloc\_block = 0;  //清空文件打开表， 清空缓冲区  for(i = 0; i < 16; i++)  fopen\_table[i] = 0;  for(i = 0; i < BLOCK\_SIZE; i++)  Buffer[i] = 0;  fp = fopen("FS\_zqw\_zzw.txt", "w+b");  fseek(fp, DISK\_START, SEEK\_SET);  //清空文件，即清空磁盘全部用0填充  for(i = 0; i < 4096; i++)  fwrite(Buffer, BLOCK\_SIZE, 1, fp);  //加载出并初始化组信息，inode以及目录项信息  reload\_group\_desc();  reload\_inode\_entry(1);  reload\_dir(0);  //改组名，并初始化组描述符内容  strcpy(super\_block[0].bg\_volume\_name, "MYSYSTEM");  super\_block[0].bg\_block\_bitmap = BLOCK\_BITMAP;  super\_block[0].bg\_inode\_bitmap = INODE\_BITMAP;  super\_block[0].bg\_inode\_table = INODE\_TABLE;  super\_block[0].bg\_free\_blocks\_count = 4096;  super\_block[0].bg\_free\_inodes\_count = 4096;  update\_group\_desc();  reload\_block\_bitmap();  reload\_inode\_bitmap();  //初始化位图信息  inode\_area[0].i\_mode = 7;  inode\_area[0].i\_blocks = 0;  inode\_area[0].i\_size = 32;  inode\_area[0].i\_block[0] = alloc\_block();  inode\_area[0].i\_blocks++;  //申请一个inode号  current\_dir = get\_inode();  update\_inode\_entry(current\_dir);  //初始化.和..目录  dir[0].inode = dir[1].inode = current\_dir;  dir[0].name\_len = 0;  dir[1].name\_len = 0;  dir[0].file\_type = dir[1].file\_type = 2;  strcpy(dir[0].name, ".");  strcpy(dir[1].name, "..");  update\_dir(inode\_area[0].i\_block[0]);  printf("The ext2 file system has been installed!\n");  } |

**4.2 用户模块**

1. login()

**\*功能说明**：根据输入的用户名和密码实现用户登录。当用户登录成功时，返回1，登录失败则返回0。

**\*实现过程**：首先由用户输入用户名和密码，其次在用户信息表中搜索，查找输入的用户名，并比对密码，如果用户名或者密码错误，则不予登录，并给予错误提示。如果存在该用户且密码输入正确则给予登录。

**\*关键程序:**

|  |
| --- |
| //用户登录  int login(){  struct user cur;  char ch;  int i = 0, j = 0;  printf("用户名：");  scanf("%s", cur.username);  printf("密码：");  while((ch = getch()) != '\r'){  cur.password[i++] = ch;  }  cur.password[i] = '\0';  printf("\n");  //查找是否有登录用户的信息，有则登录，没有则提示无此用户信息  while(j < cur\_user\_num){  if(!strcmp(users[j].username, cur.username) && !strcmp(users[j].password, cur.password)){  cur\_index = j;  cur\_user = users[cur\_index];  return 1;  }  j++;  }  return 0;  } |

1. reg()

**\*功能说明**：用户注册，能够根据命令进行新用户的信息注册。

**\*实现过程**：由用户输入新用户名以及密码，在用户信息表中查找是否已经存在该用户，如果存在则给出已存在提示，不存在时，才进行新用户的注册。注册后将新用户添加到用户信息表中，同时赋予默认的文件读写权限。

**\*关键程序：**

|  |
| --- |
| //用户注册  void reg(){  char ch;  int i = 0;  printf("新用户名：");  scanf("%s", cur\_user.username);  printf("新密码：");  while((ch = getch()) != '\r'){  cur\_user.password[i++] = ch;  }  cur\_user.password[i] = '\0';  //默认权限为只读  cur\_user.auth = 4;//100  printf("\n");  users[cur\_user\_num++] = cur\_user;  } |

1. logout(char str[8])

**\*功能说明**：根据输入的需要注销的用户名，将该用户的信息进行删除。

**\*实现过程**：传入需要删除的用户名，再对其密码进行确认，当密码正确时才进行删除。删除时，将用户的信息从用户信息表中清除。

**\*关键程序：**

|  |
| --- |
| //用户注销  void logout(char str[8]){  struct user u;  strcpy(u.username, str);  char ch;  int i = 0, j;  printf("密码确认：");  while((ch = getch()) != '\r'){  u.password[i++] = ch;  }  u.password[i] = '\0';  printf("\n");  for(i = 0; i < cur\_user\_num; i++){  if(!strcmp(users[i].username, u.username)){  for(j = i; j < cur\_user\_num - 1; j++)  users[j] = users[j+1];  cur\_user\_num--;  }  }  } |

**4.3 目录操作模块**

（1）ls()

**\*功能说明**：显示当前目录下的所有内容，包括目录和普通文件。

**\*实现过程**：首先加载出当前目录的inode节点号，其次根据inode节点中的信息找到数据块中的内容。将数据块中的内容加载到目录/文件项缓冲区中，依次访问当前目录下的目录/文件项（系统中设定的项数量的最大值为32），加载其inode号，根据inode节点的信息进行内容的输出。如果是目录文件，则输出表示为目录的信息。如果是普通文件则输出普通文件的标志。对于文件的权限，则由文件本身所设置的权限和用户的权限共同决定。根据共同作用后的结果输出结果。

**\*关键程序：**

|  |
| --- |
| //列出当前目录内容  void ls()  {  printf("items type mode size\n");  unsigned short i, j, k, tmpno, no;  i = 0;  //加载当前目录的inode号  reload\_inode\_entry(current\_dir);  //依次访问当前目录的数据块  while(i < inode\_area[0].i\_blocks){  k = 0;  reload\_dir(inode\_area[0].i\_block[i]);  while(k < 32){  if(dir[k].inode){  printf("%s", dir[k].name);  if(dir[k].file\_type == 2){  j = 0;  //加载出该文件的inode号，得到其文件信息  reload\_inode\_entry(dir[k].inode);  if(!strcmp(dir[k].name, ".."))  printf(" ");  else if(!strcmp(dir[k].name, "."))  printf(" ");  else while(j++ < 15-dir[k].name\_len)  printf(" ");  printf("<DIR> ");  switch(cur\_user.auth&7){  case 1: printf("d--x"); break;  case 2: printf("d-w-"); break;  case 3: printf("d-wx"); break;  case 4: printf("dr--"); break;  case 5: printf("dr-x"); break;  case 6: printf("drw-"); break;  case 7: printf("drwx"); break;  }  printf(" ----");  }  else if(dir[k].file\_type == 1){  j = 0;  reload\_inode\_entry(dir[k].inode);  while(j++ < 15-dir[k].name\_len)  printf(" ");  printf("<FILE> ");  switch(cur\_user.auth&7){  case 1: printf("---x"); break;  case 2: printf("--w-"); break;  case 3: printf("--wx"); break;  case 4: printf("-r--"); break;  case 5: printf("-r-x"); break;  case 6: printf("-rw-"); break;  case 7: printf("-rwx"); break;  }  printf(" %d bytes ", inode\_area[0].i\_size);  }  printf("\n");  }  k++;  reload\_inode\_entry(current\_dir);  }  i++;  }  } |

（2）reserch\_file(char tmp[9], int file\_type, unsigned short \*inode\_num, unsigned short \*block\_num, unsigned short \*dir\_num)

**\*功能说明：**给定文件名，在当前目录下查找指定文件的inode节点号，所在目录节点的数据块号（0~7）、目录/文件项所在号。

**\*实现过程**：首先加载出当前路径的inode节点号，并找到当前目录的数据块，加载到目录/文件项缓冲区中，并在其中查找是否有指定的文件名，有则记录其所在的数据块号，其inode节点号和在目录/文件项缓冲区中的下标并返回1表示查找成功，否则返回0表示查找失败。

**\*关键程序：**

|  |
| --- |
| //查找文件的inode节点号，所在目录节点的数据块号（0~7）、目录项所在号  unsigned short reserch\_file(char tmp[9],int file\_type,unsigned short \*inode\_num,unsigned short \*block\_num,unsigned short \*dir\_num){  unsigned short j, k;  reload\_inode\_entry(current\_dir);  j = 0;  while(j < inode\_area[0].i\_blocks){  //数据块大小为512，此时目录缓冲区最多有32个文件项，默认每个文件大小16  reload\_dir(inode\_area[0].i\_block[j]);  k = 0;  while(k < 32){  if(!dir[k].inode || dir[k].file\_type!=file\_type || strcmp(dir[k].name,tmp))  k++;  else{  \*inode\_num = dir[k].inode;  \*block\_num = j;  \*dir\_num = k;  return 1;  }  }  j++;  }  return 0;  } |

（3）cd(char tmp[9])

**\*功能说明**：从当前目录移动到指定的目录之下，包括父目录和指定的当前目录下的某个目录。

**\*实现过程**：首先调用函数reserch\_file找到指定目录的inode节点号，所在的数据块号以及在目录/文件项缓冲区中的下标。如果能找到该文件则进行判断需要移动到的目录，分为.目录，..目录和其它目录三种情况。移动到指定目录后则将其inode节点号作为当前的inode节点号，并修改当前路径。如果没有找到指定的目录则给予不存在的提示。

**\*关键程序：**

|  |
| --- |
| //目录移动  void cd(char tmp[9]){  unsigned short i, j, k, flag;  flag = reserch\_file(tmp, 2, &i, &j, &k);  if(flag){  current\_dir = i;  //根目录的namelen为0,此时不能再继续向上回退,若当前的命令为cd ..  //则dir[k-1]为目录.，即当前目录，故使用cd ..需保证当前目录长度不为0  if(!strcmp(tmp, "..") && dir[k-1].name\_len){  current\_path[strlen(current\_path)-dir[k-1].name\_len-1] = '\0';  current\_dirlen = dir[k].name\_len;  }  else if(!strcmp(tmp,"."));  else if(strcmp(tmp,"..")){  current\_dirlen=strlen(tmp);  strcat(current\_path,tmp);  strcat(current\_path,"/");  }  }  else printf("The directory %s not exists!\n", tmp);  } |

（4）mkdir(char tmp[9], int type)

**\*功能说明**：在当前目录下新建一个目录或者文件，目录名为传入的字符串。文件类型根据传入的type而定。

**\*实现过程**：首先加载出当前目录的inode节点号，根据inode节点中的i\_size信息判断空间是否已满。已满则给予提示并退出创建，否则继续创建，如果对于需要新建的目录或者普通文件能够找到其对应的inode号，说明相同名字的目录已经存在，则也结束创建。创建时，根据当前的inode信息找到空闲的数据块位置。找到空闲位置后，首先为新目录或者文件申请一个inode节点，并初始化inode节点的信息，同时对新建的目录或文件以及位图等信息进行相应的修改。

**\*关键程序：**

|  |
| --- |
| //新建目录文件  void mkdir(char tmp[9], int type)  {  unsigned short tmpno, i, j, k, flag;  //将当前目录的索引节点加载到缓冲区inode\_area中  reload\_inode\_entry(current\_dir);  if(!reserch\_file(tmp, type, &i, &j, &k)){  //目录项限制空间已满  if(inode\_area[0].i\_size == 4096){  printf("Directory has no room to be alloced!\n");  return;  }  flag = 1;  if(inode\_area[0].i\_size != inode\_area[0].i\_blocks\*512){  i = 0;  while(flag && i < inode\_area[0].i\_blocks){  reload\_dir(inode\_area[0].i\_block[i]);  j = 0;  //找到空闲位置区  while(j < 32){  if(dir[j].inode == 0){  flag = 0;  break; }  j++;}  i++;}  //此时j所在的位置即新建目录文件的位置  //对其申请一个inode号，tmpno记录申请的inode号  tmpno = dir[j].inode = get\_inode();  dir[j].name\_len = strlen(tmp);  dir[j].file\_type = type;  strcpy(dir[j].name, tmp);  update\_dir(inode\_area[0].i\_block[i-1]);  }  else{  inode\_area[0].i\_block[inode\_area[0].i\_blocks] = alloc\_block();  inode\_area[0].i\_blocks++;  //对新建的目录文件进行初始化  reload\_dir(inode\_area[0].i\_block[inode\_area[0].i\_blocks-1]);  tmpno = dir[0].inode = get\_inode();  dir[0].name\_len = strlen(tmp);  dir[0].file\_type = type;  strcpy(dir[0].name, tmp);  //初始化新块  for(flag = 1; flag < 32; flag++)  dir[flag].inode = 0;  update\_dir(inode\_area[0].i\_block[inode\_area[0].i\_blocks-1]);  }  inode\_area[0].i\_size += 16;  update\_inode\_entry(current\_dir);  dir\_prepare(tmpno, strlen(tmp), type);  }  else{//已经有相同文件  if(type == 1)  printf("File has already existed!\n");  else printf("Directory has already existed!\n");  }  } |

（5）rmdir(char tmp[9])

**\*功能说明**：删除指定的目录。

**\*实现过程：**首先判断是否.目录和..目录，如果是则不允许删除。当不是时，首先找到需要删除的目录的inode节点号，如果其目录下还有其它数据文件则不予删除。若没有则进行目录的删除。将要删除的目录的inode信息进行清除，清除其数据块信息。其次加载出要删除目录所在的目录的inode节点信息，修改其数据信息，包括其数据块以及其目录下的目录/文件项信息。同时，根据删除的目录所在的位置进行目录/文件项数据的移动，以填充空闲位置，避免产生较多较小的碎片。

**\*关键程序：**

|  |
| --- |
| // 删除目录文件  void rmdir(char tmp[9]){  unsigned short i, j, k, flag;  unsigned short m, n;  //.和..目录不能删除  if(!strcmp(tmp, "..") || !strcmp(tmp,".")){  printf("The directory can not be deleted!\n");  return;  }  flag = reserch\_file(tmp, 2, &i, &j, &k);  if(flag){  //找到要删除的目录的节点并加载到缓冲区  reload\_inode\_entry(dir[k].inode);  if(inode\_area[0].i\_size == 32){  inode\_area[0].i\_size = 0;  inode\_area[0].i\_blocks = 0;  remove\_block(inode\_area[0].i\_block[0]);  //得到当前目录的inode号并更改当前目录项  reload\_inode\_entry(current\_dir);  remove\_inode(dir[k].inode);  dir[k].inode = 0;  update\_dir(inode\_area[0].i\_block[j]);  inode\_area[0].i\_size -= 16;  flag = 0;  m = 1;  //对当前目录项进行修改  while(flag < 32 && m < inode\_area[0].i\_blocks){  flag = n = 0;  reload\_dir(inode\_area[0].i\_block[m]);  while(n < 32){  if(!dir[n].inode)  flag++;  n++;  }  //当flag为32时，说明该数据块已经全空，则进行删除  if(flag == 32){  remove\_block(inode\_area[0].i\_block[m]);  inode\_area[0].i\_blocks--;  //删除之后，对已存在的数据块进行合并移动  while(m < inode\_area[0].i\_blocks)  inode\_area[0].i\_block[m] = inode\_area[0].i\_block[++m];  }  }  update\_inode\_entry(current\_dir);  }  else printf("Directory is not null!\n");  }  else printf("Directory to be deleted not exists!\n");  } |

**4.4 普通文件操作模块**

（1）mkdir(char tmp[9], int type)

**\*说明**：和目录创建所使用的功能函数相同，只需修改传入的文件类型，指定为普通文件的创建。其实现过程和关键程序则和目录创建处是相同的。

1. del(char tmp[9])

**\*功能说明**：根据输入的文件名删除当前目录下的指定文件。

**\*实现过程：**在当前目录下查找输入的文件名。若不存在则给予错误提示。若存在，首先判断文件是否被打开，只有未被打开时才可删除。删除时，首先加载出要删除的文件的inode节点信息，依次清空其数据区的内容。清空数据后再对inode节点进行删除。删除了指定文件后，对其所在的目录进行数据的修改。首先加载出其inode节点信息，找到数据块，加载到目录/文件项，再对其目录/文件项进行修改，包括清空需要删除的文件项的信息以及进行空闲分区的合并等操作。

**\*关键程序：**

|  |
| --- |
| //删除文件  void del(char tmp[9])  {  unsigned short i, j, k, m, n, flag;  m = 0;  flag = reserch\_file(tmp, 1, &i, &j, &k);  if(flag){  flag = 0;  while(fopen\_table[flag] != dir[k].inode && flag < 16)  flag++;  if(flag < 16)  fopen\_table[flag] = 0;  //加载需要删除的文件的inode号  reload\_inode\_entry(i);  //依次清空数据区并初始化其参数  while(m < inode\_area[0].i\_blocks)  remove\_block(inode\_area[0].i\_block[m++]);  inode\_area[0].i\_blocks = 0;  inode\_area[0].i\_size = 0;  remove\_inode(i);  //对文件所在的当前目录进行修改  reload\_inode\_entry(current\_dir);  dir[k].inode = 0;  update\_dir(inode\_area[0].i\_block[j]);  inode\_area[0].i\_size -= 16;  m = 1;  while(m < inode\_area[i].i\_blocks){  flag = n = 0;  reload\_dir(inode\_area[0].i\_block[m]);  while(n < 32){  if(!dir[n].inode)  flag++;  n++; }  //当flag为32时，说明该数据块已经全空，则进行删除  if(flag == 32){  remove\_block(inode\_area[i].i\_block[m]);  inode\_area[i].i\_blocks--;  //当几个数据块中间被删除时，将后面的数据块向前移动  while(m < inode\_area[i].i\_blocks)  inode\_area[i].i\_block[m] = inode\_area[i].i\_block[++m];  }}  update\_inode\_entry(current\_dir); }  else printf("The file %s not exists!\n",tmp);  } |

（3）open\_file(char tmp[9])

**\*功能说明：**对于输入的文件名，找到并打开文件。

**\*实现过程：**首先载入当前目录的inode节点信息，在当前目录下查找是否存在需要打开的文件。若不存在则提示错误，若存在，首先判断在已打开文件的缓冲区中是否存在该文件的inode号，若存在则表示已经打开，若不存在则将其打开，并在缓冲区中记录下该文件的inode号，并提示已经打开。

**\*关键程序：**

|  |
| --- |
| //打开文件  void open\_file(char tmp[9])  {  unsigned short flag, i, j, k;  flag = reserch\_file(tmp, 1, &i, &j, &k);  if(flag){  //判断文件是否已经打开，若已经打开则不再进行操作  if(search\_file(dir[k].inode))  printf("The file %s has opened!\n", tmp);  else {  //未打开时，在文件打开表中做相应记录  flag = 0;  while(fopen\_table[flag])  flag++;  fopen\_table[flag] = dir[k].inode;  printf("File %s! opened\n", tmp);  } }  else printf("The file %s does not exist!\n", tmp);  } |

（4）close\_file(char tmp[9])

**\*功能说明**：根据输入的文件名，模拟关闭文件。

**\*实现过程**：首先载入当前目录的inode节点信息，在当前目录下查找是否存在需要关闭的文件。存在时，则在已打开文件缓冲区中查找是否有需要被关闭的文件，如果存在则将其已打开信息删除，标记并提示已经关闭该文件。

**\*关键程序：**

|  |
| --- |
| //关闭文件  void close\_file(char tmp[9])  {  unsigned short flag, i, j, k;  flag = reserch\_file(tmp, 1, &i, &j, &k);  if(flag){  //文件已打开，此时才可执行关闭操作  if(search\_file(dir[k].inode)){  flag = 0;  while(fopen\_table[flag] != dir[k].inode)  flag++;  fopen\_table[flag] = 0;  printf("File: %s! closed\n", tmp);  }  else printf("The file %s has not been opened!\n", tmp); }  else printf("The file %s does not exist!\n", tmp);  } |

（5）read\_file(char tmp[9])

**\*功能说明**：对已打开文件进行内容读取。

**\*实现过程**：首先载入当前目录的inode节点信息，找到需要读取的文件的inode号，判断其是否打开。若已经打开，首先判断当前用户是否具有读取的权限，有时则继续读取。根据其inode号找到数据块，对数据块依次读取到数据缓冲区。读取完成后再进行内容的输出展示。

**\*关键程序：**

|  |
| --- |
| //读文件  void read\_file(char tmp[9])  {  unsigned short flag, i, j, k;  //找到文件目录项的inode号  flag = reserch\_file(tmp, 1, &i, &j, &k);  if(flag){  //文件已打开  if(search\_file(dir[k].inode)){  reload\_inode\_entry(dir[k].inode);  if(!(cur\_user.auth&4)){  printf("The file %s can not be read!\n", tmp);  return;  }  //循环读取已有数据块  for(flag = 0; flag < inode\_area[0].i\_blocks; flag++){  reload\_block(inode\_area[0].i\_block[flag]);  Buffer[512] = '\0';  printf("%s", Buffer);  }  //flag未加，表示无数据  if(flag == 0)  printf("The file %s is empty!\n", tmp);  else  printf("\n"); }  else  printf("The file %s has not been opened!\n", tmp); }  else  printf("The file %s not exists!\n", tmp);  } |

（6）write\_file(char tmp[9])

**\*功能说明**：对已打开的文件写入内容。

**\*实现过程**：首先载入当前目录的inode节点信息，在其目录下找到需要写入内容的文件名并加载出其inode号。然后判断当前用户是否具有写入权限。当具有写入权限时则进行数据的写入，首先将输入的数据写入到数据缓冲区，同时实时判断输入的内容是否超过文件所能装入的最大字节数。输入内容以‘#’号结束。当内容写入到缓冲区后，根据写入内容的大小进行数据块的分配，分配数据块后将缓冲区中的内容再写入到磁盘中。最后对各位图及inode的数据块指针进行更新。

**\*关键程序：**

|  |
| --- |
| //写文件  void write\_file(char tmp[9])  {  unsigned short flag, i, j, k, size = 0, need\_blocks;  //找到对应文件的目录项信息，flag为1表示找到，否则未找到  flag = reserch\_file(tmp, 1, &i, &j, &k);  if(flag){  if(search\_file(dir[k].inode)){  reload\_inode\_entry(dir[k].inode);  if(!(cur\_user.auth&2)){  printf("The file %s can not be writed!\n", tmp);  return;  }  while(1){  tempbuf[size] = getchar();  //以#号结束文件输入  if(tempbuf[size] == '#'){  tempbuf[size] = '\0';  break;  }  //超过文件最大长度  if(size >= 4096){  printf("Sorry,the max size of a file is 4KB!\n");  tempbuf[size] = '\0';  break; }  size++; }  //需要的数据块数量  need\_blocks = strlen(tempbuf)/512;  //还有未满512的部分需再申请一块空间  if(strlen(tempbuf)%512)  need\_blocks++;  if(need\_blocks < 9){  //分配需要的数据块  while(inode\_area[0].i\_blocks < need\_blocks){  inode\_area[0].i\_block[inode\_area[0].i\_blocks] = alloc\_block();  inode\_area[0].i\_blocks++; }  j = 0;  while(j < need\_blocks){  if(j != need\_blocks-1){  //加载出数据块位图信息  reload\_block(inode\_area[0].i\_block[j]);  memcpy(Buffer, tempbuf+j\*BLOCK\_SIZE, BLOCK\_SIZE);  //更新位图信息  update\_block(inode\_area[0].i\_block[j]); }  else{  reload\_block(inode\_area[0].i\_block[j]);  memcpy(Buffer,tempbuf+j\*BLOCK\_SIZE, strlen(tempbuf)-j\*BLOCK\_SIZE);  //若有多余部分，需要对size重新设置  if(strlen(tempbuf) > inode\_area[0].i\_size){  Buffer[strlen(tempbuf)-j\*BLOCK\_SIZE] = '\0';  inode\_area[0].i\_size = strlen(tempbuf);  }  update\_block(inode\_area[0].i\_block[j]);  }  j++; }  update\_inode\_entry(dir[k].inode); }  else printf("Sorry,the max size of a file is 4KB!\n"); }  else printf("The file %s has not opened!\n",tmp); }  else printf("The file %s does not exist!\n",tmp);  } |

**五、课题实现结果**

**5.1 用户模块实现结果**

**5.1.1 用户登录**

本虚拟文件系统在初始时默认设置了一个root用户，对其进行登录，密码为不回显。得到其登录过程及运行结果如下图所示：

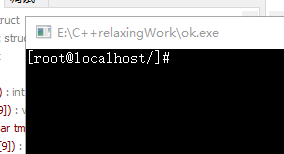
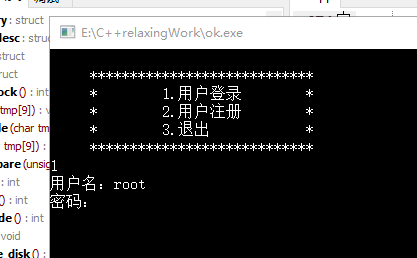


图5-1 root用户登录界面 图5-2 root用户登录结果

**5.1.2 用户注册**

注册一个名为dac的用户并使用其名字和密码登录文件系统，其结果如下：

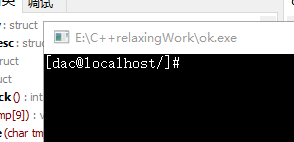
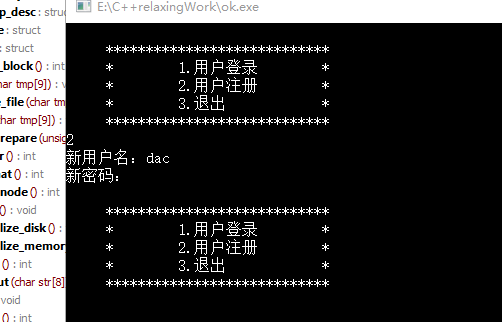


图5-3 新用户注册结果 图5-4 新用户登录结果

**5.1.3 用户注销**

对dac用户使用注销命令，确认密码后将其注销，再次登录提示用户名或密码错误，验证了注销程序。其结果如下：

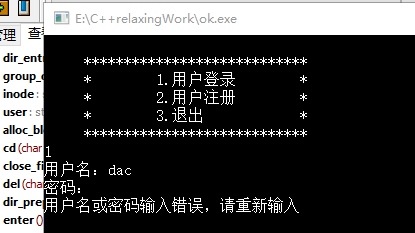
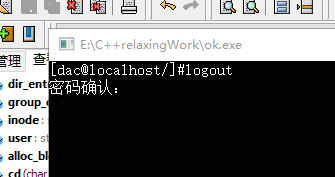


图5-5 注销用户dac 图5-6 注销程序验证结果

**5.2 目录操作模块实现结果**

**5.2.1 新建目录与当前目录显示**

使用root用户登录文件系统，在登录后的目录下使用目录创建命令mkdir创建两个目录文件file与tfg，使用ls命令显示当前目录的内容，查看创建结果，其结果如下图所示：

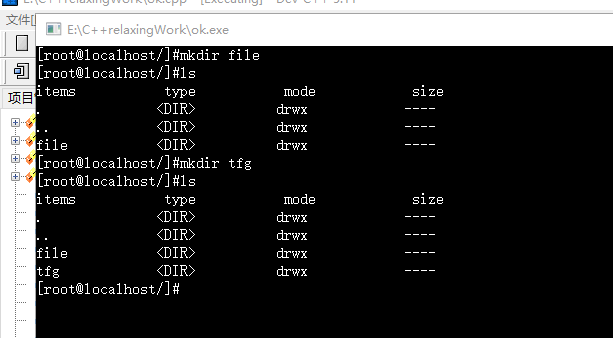


图5-7 目录创建与目录内容显示测试

**5.2.2 删除目录**

使用目录删除命令rmdir删除目录文件tfg，再使用ls命令查看删除结果，如下图所示：

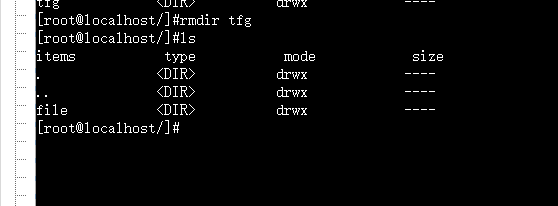


图5-8 目录删除测试

**5.2.3 目录移动**

使用cd命令进行目录移动的测试。首先测试进行其子目录的情况，使用命令cd file进入file目录，然后使用cd ..返回到父目录，其测试结果如下图：

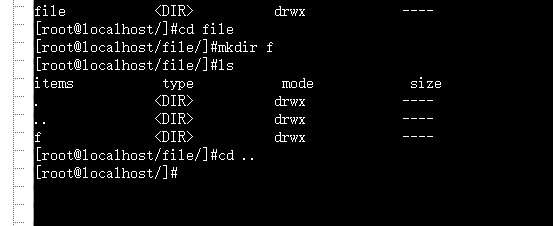


图5-9 目录移动程序测试

**5.3 普通文件模块实现结果**

**5.3.1 创建文件**

使用命令mkf创建文件m.c和文件test.c，再使用ls命令查看创建结果，如下图所示：



图5-10 文件创建测试

**5.3.2 删除文件**

使用命令rm删除文件m.c，并使用ls命令查看删除结果，如下图：

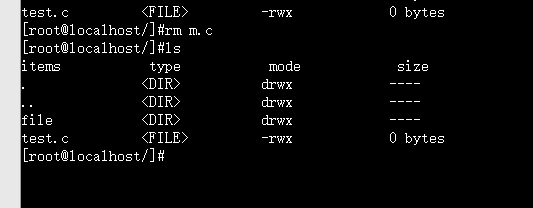


图5-11 文件删除测试

**5.3.3 打开文件与关闭文件**

使用open命令打开已创建的文件test.c，系统会提示相关的打开信息。当再次使用打开命令时，系统则会提示文件已经被打开，不能再打开。使用close命令对打开的文件进行关闭，正确操作时会给予已关闭的信息提示。若文件是在没有打开的情况下关闭，则会提示文件未打开。测试结果如下入所示：

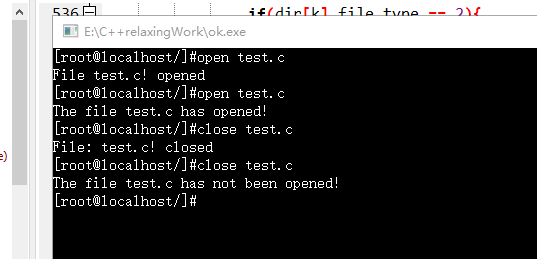


图5-12 文件打开与关闭测试

**5.3.4 写入文件与读取文件**

使用write命令对文件test.c进行写操作，由于写文件之前未打开文件，此时会提示文件test.c未打开。因此首先打开文件，再进行写操作，此时便可正确写入。写入后对文件执行关闭命令。其结果如下图所示：

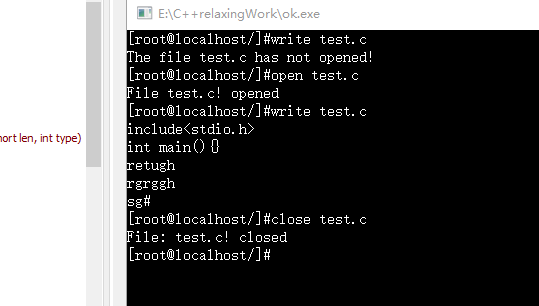


图5-13 文件内容写入测试

写入内容到文件后，使用read命令对文件内容进行读取，同样在文件未打开时读取会提示文件未打开的信息。将文件test.c打开，使用read命令对其内容读取，读取后显示到了控制台，其结果如下图所示：

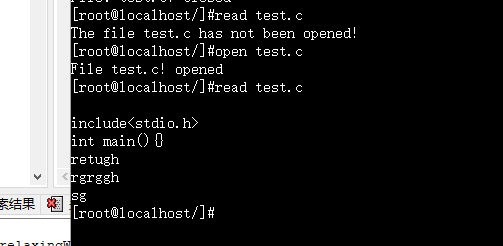


图5-14 文件内容读取测试

**5.4 系统格式化实现结果**

使用format命令对系统进行格式化，系统会给予相应的提示信息，以确认格式化操作。格式化操作之后所有信息都恢复了初始情况。使用ls命令可以查看到当前目录下只有当前目录.和父目录..两个目录，系统中也没有其余的任何文件。格式化后的结果如下图所示：

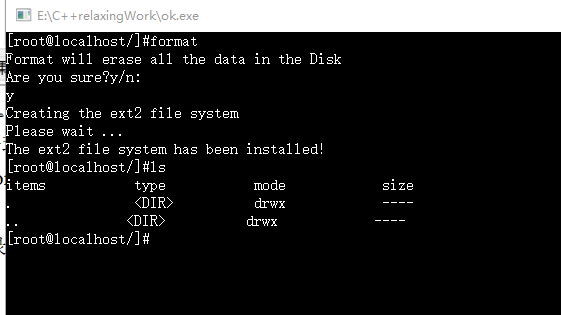


图5-15 系统格式化测试

1. **结果分析与总结**

通过对方案的实现以及测试后，验证了所有实现的操作的正确性。虽都完成了相应的功能，但也有较多的不足。

在用户模块中，只是简单的实现了用户登录，注册与注销三个功能，对于已经实现的部分，能够模拟用户登录系统，模拟用户的注册和注销。并且在操作的过程中能够给予相应的操作提示信息。但对于自己实现的用户部分，不能对新建的用户等信息进行保存，而只是在线的，所以这一部分也还有很多的不足。其次则是使用C语言实现使得界面看起来不够美观。

对于目录操作模块，从测试的结果来看，结果还是可观的。目录移动能够实现在多级目录之间转换，目录文件的创建和删除命令也能得到正确结果，目录内容显示则可对当前目录下的多个目录或者普通文件进行正确的显示。通过对该模块的测试，也较为全面的进行了程序实现的验证。从得到的实现结果可以看出，程序通过输入的命令也进行了较好的内容展示。

对于普通文件的操作模块，模拟了文件的打开关闭、读写以及创建和删除功能。从实现的结果中可以看出，每个功能都得到了正确的执行，由此可以验证程序的正确性。同时，在不同操作的过程中，根据用户的输入给予不同的提示信息。例如在未打开文件时对文件进行读操作，系统会给予文件未打开的信息提示。通过文件的读写结果验证也可以看到，读写功能正确的进行了内容的读取和写入，能够正确写入文件内容，正确读出文件的内容。

当然，对于目录文件和普通文件两个部分的操作也有较多的不足，其中文件的读写权限便是其一。对于设计的文件系统，只有一个用户的三位权限，即读、写、执行三种。没有区分文件主和其它使用用户。其次，也没有设计文件权限的修改命令，以致于文件的权限固定单一。

从总体的设计来看，本虚拟文件系统在物理结构上对几个主要的部分进行了划分，包括超级块，位图以及数据区等，能够简单的模拟相应的文件操作以及用户操作。因此，相反的也是不足的，没有进行细致的划分，比如EXT2文件系统中的组描述符表GDT等，因而也产生了些较小的缺陷。不过从设计到得到实现结果来说，还是比较满意的，当然对于不足的部分也需要后续不断的学习进行补充。

1. **参考文献**
2. 郭学理, 韦智, 潘松. Linux的Ext2文件系统[J]. 计算机应用研究(05):128-130.
3. 包怀忠. EXT2文件系统分析[J]. 计算机工程与设计, 2005, 26(4):1022-1024.
4. 王红. Linux文件系统结构分析[J]. 潍坊学院学报(2):34-36.
5. <https://www.jianshu.com/p/3355a35e7e0a>.Ext2文件系统简单剖析,2018.
6. <https://blog.csdn.net/YuZhiHui_No1/article/details/50256713>.ext2文件系统结构分析,2015.
7. <https://cloud.tencent.com/developer/article/1008537>.Ext2文件系统布局,2017.