### 类EXT2文件系统的设计

[类EXT2文件系统的设计](#类ext2文件系统的设计)  
 [实验内容](#实验内容)  
 [实验步骤](#实验步骤)  
 [数据结构定义](#数据结构定义)  
 [底层函数](#底层函数)  
 [命令层函数](#命令层函数)  
 [shell 的设计](#shell-的设计)

#### 实验内容

* 模拟EXT2文件系统原理设计实现一个类EXT2文件系统

#### 实验步骤

* 定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项
* 实现包括分配数据块等底层操作
* 实现命令层函数，包括 dir 等操作
* 完成 shell 的设计
* 测试整个文件系统的功能

#### 数据结构定义

1. // 组描述符
     
   struct group\_desc { // 32 B
     
    char bg\_volume\_name[16]; //文件系统的卷名
     
    unsigned short bg\_block\_bitmap; //块位图的起始块号
     
    unsigned short bg\_inode\_bitmap; //索引结点位图的起始块号
     
    unsigned short bg\_inode\_table; //索引结点表的起始块号
     
    unsigned short bg\_free\_blocks\_count; //本组空闲块的个数
     
    unsigned short bg\_free\_inodes\_count; //本组空闲索引结点的个数
     
    unsigned short bg\_used\_dirs\_count; //组中分配给目录的结点
     
    char bg\_pad[4]; //填充(0xff)
     
   };
     
     
   // 索引结点
     
   struct inode { // 64 B
     
    unsigned short i\_mode; //文件类型及访问权限
     
    unsigned short i\_blocks; //文件所占的数据块个数(0~7), 最大为7
     
    unsigned long i\_size; //文件或目录大小(单位 byte)
     
    unsigned long i\_atime; //访问时间
     
    unsigned long i\_ctime; //创建时间
     
    unsigned long i\_mtime; //修改时间
     
    unsigned long i\_dtime; //删除时间
     
    unsigned short i\_block[8]; //直接索引方式 指向数据块号
     
    char i\_pad[24]; //填充(0xff)
     
   };
     
     
   // 目录项
     
   struct dir\_entry { // 16 B
     
    unsigned short inode; //索引节点号
     
    unsigned short rec\_len; //目录项长度
     
    unsigned short name\_len; //文件名长度
     
    char file\_type; //文件类型(1 普通文件 2 目录.. )
     
    char name[9]; //文件名
     
   };
     
     
     
   // 用户信息
     
   struct user {
     
    char username[10];
     
    char password[10];
     
   }User[USER\_MAX];
   * 因为系统中最多有4096个数据块，所以索引节点最多也是4096。所以索引节点号只需16位（unsigned short）即可。
   * struct inode 中为保证一个索引节点占64位，需要提供char i\_pad[24]，作为填充。

#### 底层函数

##### **读写缓冲区类操作函数**

* // 写gdt  
  static void update\_group\_desc()  
  {  
   fp = fopen("./FS.txt", "rb+");  
   fseek(fp, GDT\_START, SEEK\_SET);  
   fwrite(&gdt, GD\_SIZE, 1, fp);  
   fflush(fp);  
  }  
    
  // 读gdt  
  static void reload\_group\_desc()  
  {  
   fseek(fp, GDT\_START, SEEK\_SET);  
   fread(&gdt, GD\_SIZE, 1, fp);  
  }
* update\_group\_desc函数打开一个名为"FS.txt"的文件，以**读写模式打开（"rb+"）**，然后将文件指针移动到GDT的起始位置（GDT\_START），并将内存中的GDT数据（gdt）写入文件。最后，使用fflush函数刷新文件流，确保数据被写入文件。
* reload\_group\_desc函数假定文件指针fp已经打开，然后将文件指针移动到GDT的起始位置（GDT\_START），并从文件中读取GDT数据到内存中的gdt变量中。
* 其他结构体缓冲区读写函数依次类推：**但需注意何时用指针合适取地址**

##### **分配删除块函数：标注位图**

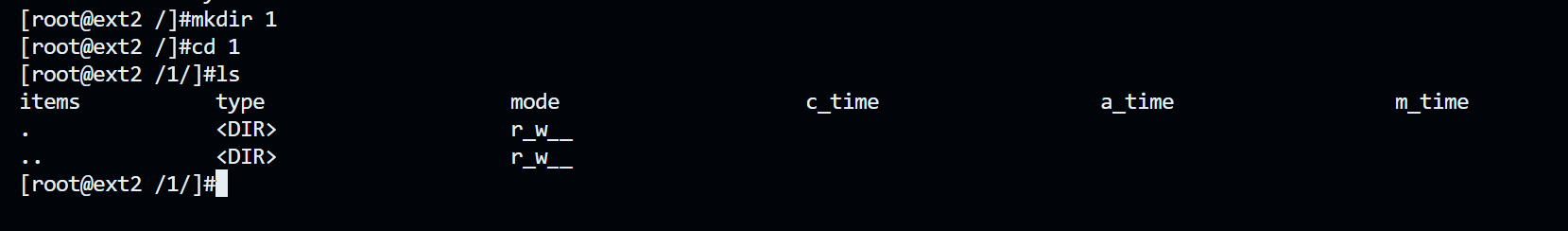
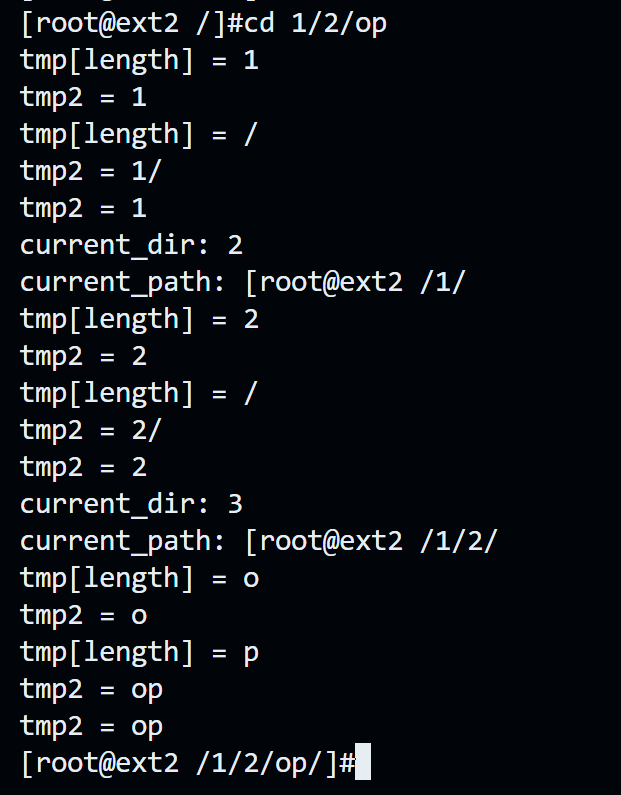
* **分配函数**
* // 分配data\_block  
  static int alloc\_block()  
  {  
   int flag = 0;  
   if (gdt.bg\_free\_blocks\_count == 0)  
   {  
   printf("There is no block to be allocated!\n");  
   return (0);  
   }  
   reload\_block\_bitmap();  
   for (int i = 0; i < 512; i++)  
   {  
   if (bitbuf[i] != 0xff)  
   {  
   for (int j = 0; j < 8; j++)  
   {  
   if ((bitbuf[i] & (1 << j)) == 0)  
   {  
   bitbuf[i] |= (1 << j);  
   last\_alloc\_block = i \* 8 + j;  
   break;  
   }  
   }  
   break;  
   }  
   }  
    
   update\_block\_bitmap();  
   gdt.bg\_free\_blocks\_count--;  
   update\_group\_desc();  
   return last\_alloc\_block;  
  }
* **循环遍历块位图的每个字节（共512个字节）**，然后再遍历每个字节中的每一位做**与运算**。如果某一位为0，表示该数据块未被分配，则将该位设置为1，表示分配了该数据块，并记录下该数据块的编号。最后跳出循环，返回分配的数据块编号。
* **删除块函数**
* // 删除data\_block  
  static void remove\_block(unsigned short del\_num)  
  {  
   unsigned short tmp;  
   tmp = del\_num / 8;  
   reload\_block\_bitmap();  
   switch (del\_num % 8)  
   {   
   case 0:  
   bitbuf[tmp] = bitbuf[tmp] & 127;  
   break;   
   case 1:  
   bitbuf[tmp] = bitbuf[tmp] & 191;  
   break;   
   case 2:  
   bitbuf[tmp] = bitbuf[tmp] & 223;  
   break;  
   case 3:  
   bitbuf[tmp] = bitbuf[tmp] & 239;  
   break;   
   case 4:  
   bitbuf[tmp] = bitbuf[tmp] & 247;  
   break;   
   case 5:  
   bitbuf[tmp] = bitbuf[tmp] & 251;  
   break;   
   case 6:  
   bitbuf[tmp] = bitbuf[tmp] & 253;  
   break;   
   case 7:  
   bitbuf[tmp] = bitbuf[tmp] & 254;  
   break;   
   }  
   update\_block\_bitmap();  
   gdt.bg\_free\_blocks\_count++;  
   update\_group\_desc();  
  }
* 首先计算出要删除的数据块编号在块位图中的位置。接下来，根据要删除的数据块编号在字节中的位置，使用switch语句来设置对应位为0，表示释放该数据块。

##### **配置新节点及初始化目录**

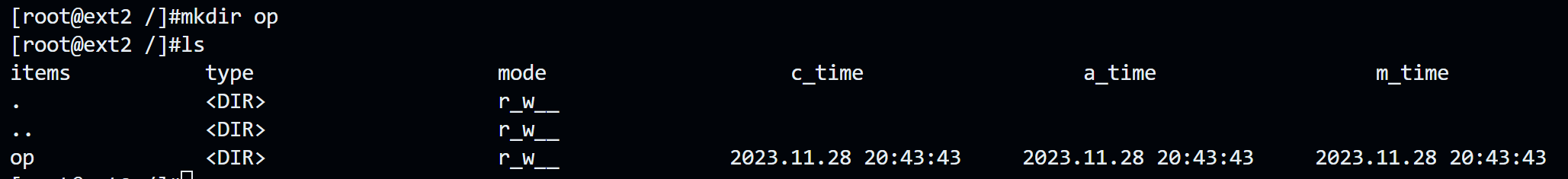
* static void dir\_init(unsigned short tmp, unsigned short len, int type, char name[100])
    
  {
    
   reload\_inode\_entry(tmp);
    
    
   time\_t Time;
    
   time(&Time);
    
   if (type == 2)
    
   { // dir
    
   inode\_buff.i\_size = 32;
    
   inode\_buff.i\_blocks = 1;
    
   inode\_buff.i\_block[0] = alloc\_block();
    
   inode\_buff.i\_ctime = Time;
    
   inode\_buff.i\_mtime = Time;
    
   inode\_buff.i\_atime = Time;
    
   dir[0].inode = tmp;
    
   dir[1].inode = current\_dir;
    
   dir[0].name\_len = len;
    
   dir[1].name\_len = current\_dirlen;
    
   dir[0].file\_type = dir[1].file\_type = 2;
    
    
   for (type = 2; type < 32; type++)
    
   dir[type].inode = 0;
    
   strcpy(dir[0].name, ".");
    
   strcpy(dir[1].name, "..");
    
   update\_dir(inode\_buff.i\_block[0]);
    
    
   inode\_buff.i\_mode = 6;
    
   }
    
   else
    
   {
    
   inode\_buff.i\_size = 0;
    
   inode\_buff.i\_blocks = 0;
    
   inode\_buff.i\_mode = 6;
    
   inode\_buff.i\_ctime = Time;
    
   inode\_buff.i\_mtime = Time;
    
   inode\_buff.i\_atime = Time;
    
   int len = strlen(name);
    
   if (len < 4)
    
   {
    
   inode\_buff.i\_mode |= 1;
    
   }
    
   else
    
   {
    
   char \*lastFour = &name[len - 4];
    
   if (strcmp(lastFour, ".exe") == 0 || strcmp(lastFour, ".bin") == 0 || strcmp(lastFour, ".com") == 0)
    
   {
    
   inode\_buff.i\_mode |= 1;
    
   }
    
   else
    
   {
    
   ;
    
   }
    
   }
    
   }
    
   update\_inode\_entry(tmp);
    
  }
* 函数根据类型type的值来初始化i节点的各个字段。**如果类型为2（表示目录）**，则设置i节点的字段，并初始化目录项dir的内容，并将"."和".."目录项的名称赋值为当前目录和父目录的名称。
* **如果类型不为2（表示文件）**，则设置i节点的大小、块数、创建时间、修改时间、访问时间等字段，并根据**文件名的后缀(扩展名为.exe,.bin,.com及不带扩展名的)**来判断是否为可执行文件，如果是则设置i节点的执行权限。

#### 命令层函数

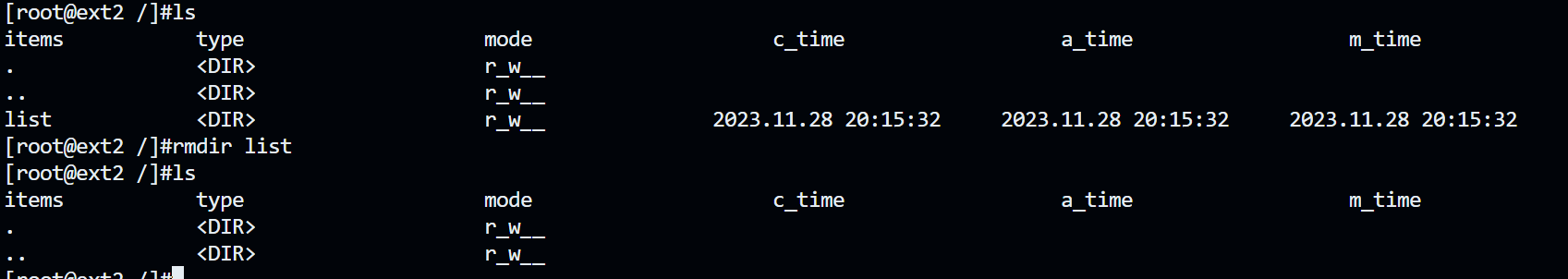
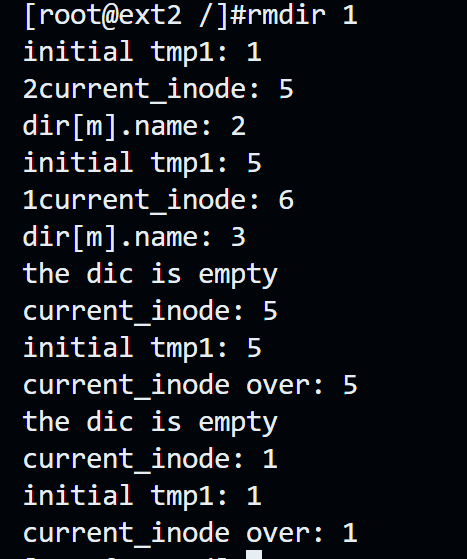
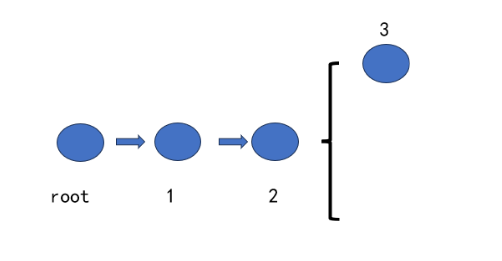
##### cd 改变路径

* 
* // 进入某个目录，实际上是改变当前路径  
  void cd(char tmp[100])  
  {  
   unsigned short i, j, k, flag;  
   flag = research\_file(tmp, 2, &i, &j, &k);  
   if (flag)  
   {  
    
   if (!strcmp(tmp, "..") && dir[k - 1].name\_len)   
   {  
   current\_dir = i;  
   current\_path[strlen(current\_path) - dir[k - 1].name\_len - 1] = '\0';  
   current\_dirlen = dir[k].name\_len;  
   // 修改访问时间  
   reload\_inode\_entry(current\_dir);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(current\_dir);  
   return;  
   }  
   else if (!strcmp(tmp, "..") && !dir[k - 1].name\_len)   
   {  
   return;  
   }  
   else if (!strcmp(tmp, "."))  
   {  
   // 修改访问时间  
   current\_dir = i;  
   reload\_inode\_entry(current\_dir);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(current\_dir);  
   return;  
   }  
   else if (!strcmp(tmp, "."))  
   {  
   // 修改访问时间  
   current\_dir = i;  
   reload\_inode\_entry(current\_dir);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(current\_dir);  
   return;  
   }  
   }  
   // 以/为分隔符，可以多级cd  
   int length = 0;  
   int ii = 0;  
   char tmp2[100];  
   while (tmp[length] != '\0')  
   {  
   // 以/为分隔符  
   char p[1];  
   p[0] = tmp[length];  
   tmp2[ii] = tmp[length];  
   tmp2[ii + 1] = '\0';  
   // printf("tmp[length] = %c\n", tmp[length]);  
   // printf("tmp2 = %s\n", tmp2);  
    
   if (!strcmp(p, "/"))  
   {  
   tmp2[ii] = '\0';  
   unsigned short i, j, k, flag;  
   // printf("tmp2 = %s\n", tmp2);  
   flag = research\_file(tmp2, 2, &i, &j, &k);  
   if (flag)  
   {  
   current\_dir = i;  
    
   // printf("current\_dir: %d\n", current\_dir);  
   current\_dirlen += strlen(tmp2);  
   strcat(current\_path, tmp2);  
   strcat(current\_path, "/");  
   // printf("current\_path: %s\n", current\_path);  
   // 修改访问时间  
   reload\_inode\_entry(current\_dir);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(current\_dir);  
   }  
   else  
   {  
   printf("The directory %s not exists!\n", tmp2);  
   }  
   char tmp2[100];  
   ii = -1;  
   }  
   else  
   {  
   ;  
   }  
   length++;  
   ii++;  
   }  
   tmp2[ii] = '\0';  
   // printf("tmp2 = %s\n", tmp2);  
   flag = research\_file(tmp2, 2, &i, &j, &k);  
   if (flag)  
   {  
   current\_dir = i;  
   current\_dirlen += strlen(tmp2);  
   strcat(current\_path, tmp2);  
   strcat(current\_path, "/");  
   // 修改访问时间  
   reload\_inode\_entry(current\_dir);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(current\_dir);  
   }  
   else  
   {  
   printf("The directory %s not exists!\n", tmp2);  
   }  
  }
* cd 分为三种情况：主要考虑更改 **current\_dir**和**current\_path**
  1. **传送到上一级目录（..）**但不在根目录上
  2. 在根目录上执行 cd ..（直接return）
  3. **传送到子目录（考虑多级）:** **以/为分隔符**,截取每一级的路径，搜索其索引节点及路径，不断循环。
  + 
  + 该图展示了分割路径的路径，通过遍历路径寻找 **“/”**分割每一层路径。

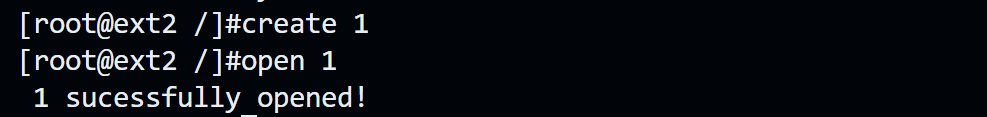
##### mkdir 创建目录

* 
* // 创建目录  
  void mkdir(char tmp[100], int type)  
  {  
   unsigned short tmpno, i, j, k, flag;  
   reload\_inode\_entry(current\_dir);  
   if (!research\_file(tmp, type, &i, &j, &k))  
   {  
   if (inode\_buff.i\_size == 4096)  
   {   
   printf("Directory has no room to be alloced!\n");  
   return;  
   }  
   flag = 1;  
   if (inode\_buff.i\_size != inode\_buff.i\_blocks \* 512)  
   {  
   i = 0;  
   while (flag && i < inode\_buff.i\_blocks)  
   {  
   reload\_dir(inode\_buff.i\_block[i]);  
   j = 0;  
   while (j < 32)  
   {  
   if (dir[j].inode == 0)  
   {  
   flag = 0;   
   break;  
   }  
   j++;  
   }  
   i++;  
   }  
   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\_buff.i\_block[i - 1]);  
   }  
   else  
   { // 全满 新增加块  
   inode\_buff.i\_block[inode\_buff.i\_blocks] = alloc\_block();  
   inode\_buff.i\_blocks++;  
   reload\_dir(inode\_buff.i\_block[inode\_buff.i\_blocks - 1]);  
   tmpno = dir[0].inode = get\_inode();  
   dir[0].name\_len = strlen(tmp);  
   dir[0].file\_type = type;  
   for (flag = 1; flag < 32; flag++)  
   {  
   dir[flag].inode = 0;  
   }  
   update\_dir(inode\_buff.i\_block[inode\_buff.i\_blocks - 1]);  
   }  
   inode\_buff.i\_size += 16;  
   update\_inode\_entry(current\_dir);  
   // 为新增目录分配 dir\_entry  
   dir\_prepare(tmpno, strlen(tmp), type, tmp);  
   }  
   else  
   {   
   printf("Directory has already existed!\n");  
   }  
  }

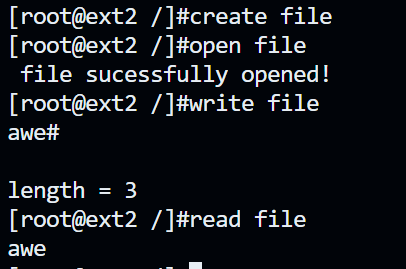
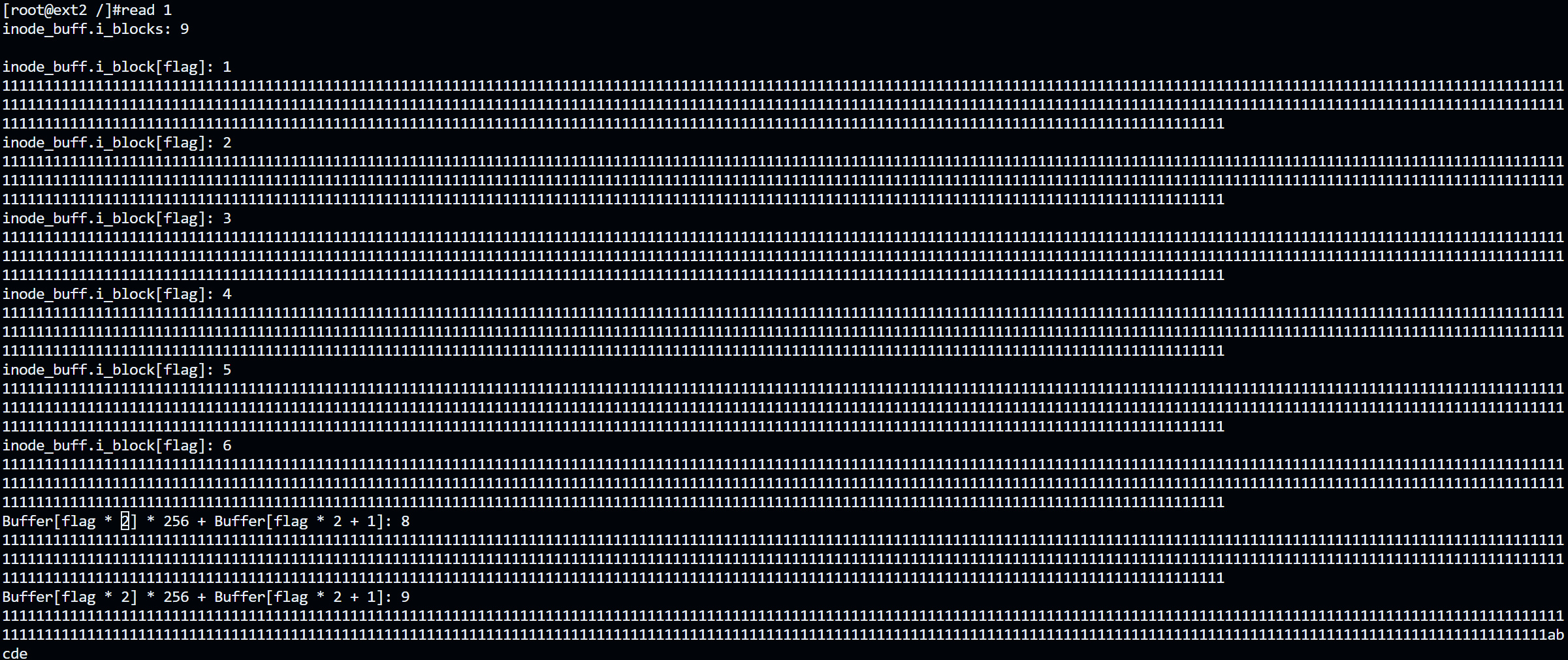
##### rmdir 删除目录

* 
* // 删除目录  
  void rmdir(char tmp[100])  
  {  
   unsigned short i, j, k, flag;  
   unsigned short m, n;  
   unsigned short tmp1 = current\_dir;  
   if (!strcmp(tmp, "..") || !strcmp(tmp, "."))  
   {  
   printf("The directory can not be deleted!\n");  
   return;  
   }  
   flag = research\_file(tmp, 2, &i, &j, &k);  
   unsigned short tmp2 = i;  
   if (flag)  
   {  
   reload\_inode\_entry(dir[k].inode); // 加载要删除的节点  
   if (inode\_buff.i\_size == 32)  
   { // 只有.and ..  
   inode\_buff.i\_size = 0;  
   inode\_buff.i\_blocks = 0;  
    
   remove\_block(inode\_buff.i\_block[0]);  
   // 更新 tmp 所在父目录  
   reload\_inode\_entry(current\_dir);  
   reload\_dir(inode\_buff.i\_block[j]);  
   remove\_inode(dir[k].inode);  
   dir[k].inode = 0;  
   update\_dir(inode\_buff.i\_block[j]);  
   inode\_buff.i\_size -= 16;  
   flag = 0;  
   m = 1;  
   while (flag < 32 && m < inode\_buff.i\_blocks)  
   {  
   flag = n = 0;  
   reload\_dir(inode\_buff.i\_block[m]);  
   while (n < 32)  
   {  
   if (!dir[n].inode)  
   {  
   flag++;  
   }  
   n++;  
   }  
   // 如果删除过后，整个数据块的目录项全都为空。类似于在数组中删除某一个位置  
   if (flag == 32)  
   {  
   remove\_block(inode\_buff.i\_block[m]);  
   inode\_buff.i\_blocks--;  
   while (m < inode\_buff.i\_blocks)  
   {  
   inode\_buff.i\_block[m] = inode\_buff.i\_block[m + 1];  
   ++m;  
   }  
   }  
   }  
   update\_inode\_entry(current\_dir);  
   return;  
   }  
   else  
   {  
   for (int l = 0; l < inode\_buff.i\_blocks; l++)  
   {  
   reload\_dir(inode\_buff.i\_block[l]);  
   for (m = 0; m < 32; m++)  
   {  
   if (!strcmp(dir[m].name, ".") || !strcmp(dir[m].name, "..") || dir[m].inode == 0)  
   continue;  
   if (dir[m].file\_type == 2)  
   {  
   current\_dir = i;  
   rmdir(dir[m].name);  
   }  
   else if (dir[m].file\_type == 1)  
   {  
   current\_dir = i;  
   del(dir[m].name);  
   current\_dir = i;  
   }  
   }  
   if (inode\_buff.i\_size == 32)  
   {  
    
   current\_dir = tmp1;  
   rmdir(tmp);  
   }  
   }  
   return;  
   }  
   }  
   else  
   {  
   printf("Directory to be deleted not exists!\n");  
   }  
  }
  1. 首先检查要删除的目录是否为"."或".."，如果是则无法删除，否则继续执行。
  2. 如果目录下只有"."和".."两个项，则直接删除该目录，更新父目录信息，并释放相关的数据块和节点。
  3. 如果目录下还有其他文件或子目录，则递归调用rmdir函数，依次删除其中的文件和子目录。最后更新父目录信息。
  + 在递归调用时要注意更新 **current\_dir**，与当前删除的一级的目录相对应。
  + 
  + 图中表示的是**current\_dir**的变化过程。该文件的构成为
* 有图中可知结点从根节点先层层递进到达目录2，删除文件3后（判断目录是否已空）再删除目录本身。从子节点再递归删除返回。

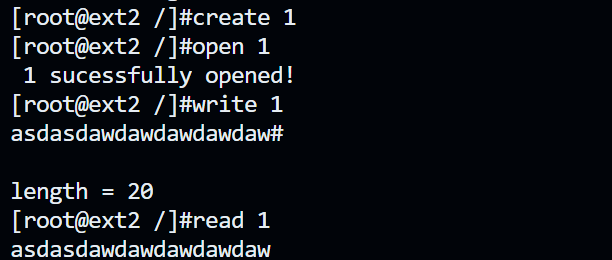
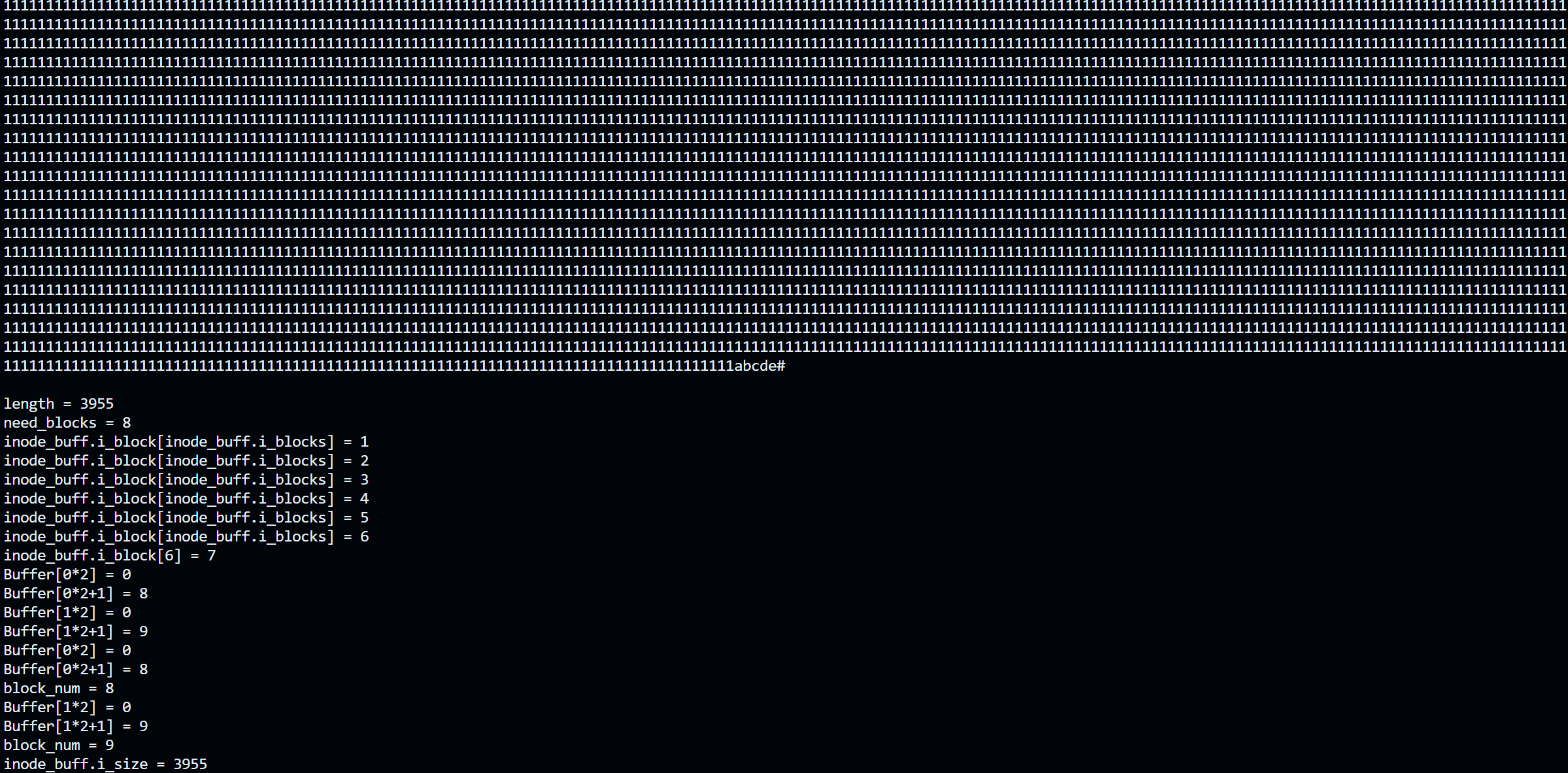
##### open\_file 打开文件

* 
* void open\_file(char tmp[100])  
  {  
   unsigned short flag,inode\_num , block\_num, dir\_num;  
   flag = research\_file(tmp, 1, &inode\_num, &block\_num, &dir\_num);  
   if (flag)  
   {  
   if (search\_file(dir[dir\_num].inode))  
   {  
   printf("The file %s has opened!\n", tmp);  
   }  
   else  
   {  
   flag = 0;  
   while (fopen\_table[flag])  
   {  
   flag++;  
   }  
   fopen\_table[flag] = (short)dir[dir\_num].inode;  
   // 更新文件的访问时间  
   reload\_inode\_entry(dir[dir\_num].inode);  
   time\_t t;  
   time(&t);  
   inode\_buff.i\_atime = t;  
   update\_inode\_entry(dir[dir\_num].inode);  
    
   printf(" %s sucessfully opened!\n", tmp);  
   }  
   }  
   else  
   printf("The file %s does not exist!\n", tmp);  
  }
* 首先查找文件，如果找到了文件，则检查该文件是否已经被打开，如果已经被打开，则输出提示信息。如果文件没有被打开，则在 fopen\_table 数组中找到一个空闲的位置，将该文件的 inode 号存入该位置。然后更新文件的访问时间。

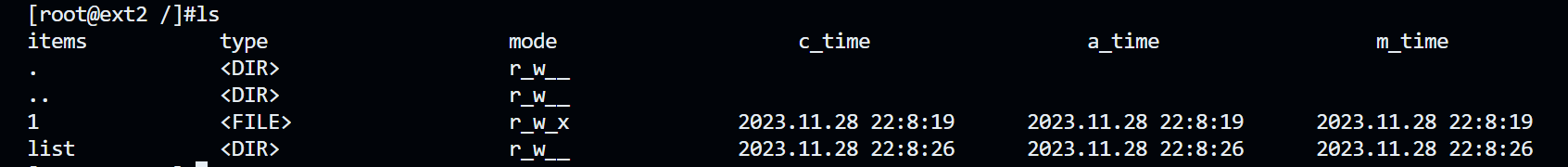
##### read\_file 读文件

* 
* void read\_file(char tmp[100])  
  {  
   unsigned short flag, i, j, k, t;  
   flag = research\_file(tmp, 1, &i, &j, &k);  
   if (flag)  
   {  
   if (search\_file(dir[k].inode)) // 读文件的前提是该文件已经打开  
   {  
   reload\_inode\_entry(dir[k].inode);  
   // 判断是否有读的权限  
   if (!(inode\_buff.i\_mode & 4)) // i\_mode:111b:读,写,执行  
   {  
   printf("The file %s can not be read!\n", tmp);  
   return;  
   }  
   // 读文件直接索引  
   if (inode\_buff.i\_blocks <= 6)  
   {  
   for (flag = 0; flag < inode\_buff.i\_blocks; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[flag]);  
   for (t = 0; t < inode\_buff.i\_size - flag \* 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   }  
   // 读文件一级索引  
   else if (inode\_buff.i\_blocks < 262)  
   {  
   printf("inode\_buff.i\_blocks: %d\n", inode\_buff.i\_blocks);  
   for (flag = 0; flag < 6; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[flag]);  
   // printf("\ninode\_buff.i\_block[flag]: %d\n", inode\_buff.i\_block[flag]);  
   for (t = 0; t < 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   reload\_block(inode\_buff.i\_block[6]);  
   for (flag = 0; flag < inode\_buff.i\_blocks - 7; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[6]);  
   int t = Buffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1];  
   // printf("\nBuffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1]: %d\n", Buffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1]);  
   reload\_block(t);  
   for (t = 0; t < 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   }  
   // 读文件二级索引  
   else if (inode\_buff.i\_blocks < 4072)  
   {  
   for (flag = 0; flag < 6; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[flag]);  
   for (t = 0; t < 6 \* 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   reload\_block(inode\_buff.i\_block[6]);  
   for (flag = 0; flag < 256; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[6]);  
   reload\_block(Buffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1]);  
   for (t = 6 \* 512; t < inode\_buff.i\_size - 6 \* 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   reload\_block(inode\_buff.i\_block[7]);  
   for (flag = 0; flag < inode\_buff.i\_blocks - 262; flag++)  
   {  
   reload\_block(inode\_buff.i\_block[7]);  
   reload\_block(Buffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1]);  
   for (t = 0; t < inode\_buff.i\_size - (flag + 262) \* 512; ++t)  
   {  
   printf("%c", Buffer[t]);  
   }  
   }  
   }  
   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);  
  }
* 创建文件需满足两个条件：
  1. 该文件是可读的 2.该文件已打开 open
* 若满足条件即可读。读时首先读取i\_blcoks的个数。根据数据块个数选择索引方式:
  1. （i\_blcoks< 6）直接索引将数据块的位置信息存储在文件的inode项中的i\_blocks数组中**。每个数组元素对应一个数据块的位置**。直接读取i\_blcok[]数组中的块号，然后直接读取块号指向的数据块的内容。
  2. （i\_blcoks< 262）一级子索引是一个额外的索引块，其中每个索引项指向一个数据块的位置。索引块中用unsigned int 16位变量，即2字节表示1个块号。索引块都用来存放块号。读取时若想获得一个块号需读取两个Buffer[]，代表最终块号的高8位和低8位。
  + int t = Buffer[flag \* 2] \* 256 + Buffer[flag \* 2 + 1];读取对应块号的内容。
  1. （i\_blcoks< 4072）读入二级索引块。代码通过循环读入一级索引块的块号，**并在每个一级索引块中再次循环读入块号，将这些块的块号对应的数据块载入读取**。
* 

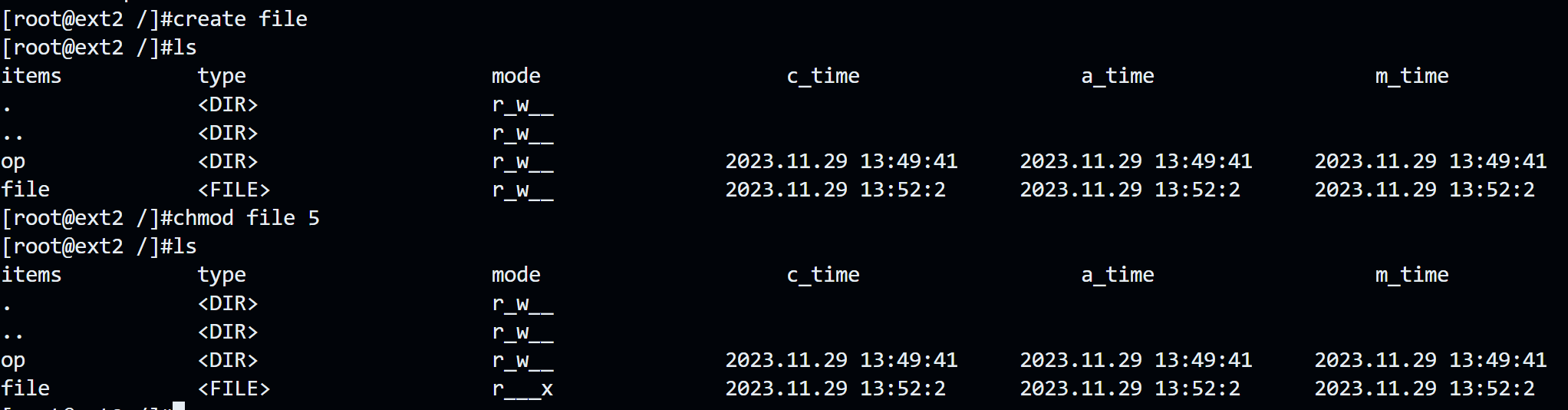
##### write\_file 写文件

* 
* void write\_file(char tmp[100])  
  {  
   unsigned short flag, i, j, k = 0, need\_blocks;  
   unsigned long size = 0, length;  
   flag = research\_file(tmp, 1, &i, &j, &k);  
   if (flag)  
   {  
   if (search\_file(dir[k].inode))  
   {  
   reload\_inode\_entry(dir[k].inode);  
   if (!(inode\_buff.i\_mode & 2)) // i\_mode:111b:读,写,执行  
   {  
   printf("The file %s can not be writed!\n", tmp);  
   return;  
   }  
   // fflush(stdin);  
   while (1)  
   {  
   tempbuf[size] = getchar();  
   if (tempbuf[size] == '#')  
   {  
   tempbuf[size] = '\0';  
   break;  
   }  
   if (size >= 4096 \* 512)  
   {  
   printf("Sorry,the max size of a file is 2MB!\n");  
   break;  
   }  
   size++;  
   }  
   if (size >= 4096 \* 512)  
   {  
   length = 4096 \* 512;  
   }  
   else  
   {  
   length = strlen(tempbuf);  
   }  
   printf("\nlength = %d\n", length);  
   // 计算需要的数据块数目  
   need\_blocks = length / 512;  
   if (length % 512)  
   {  
   need\_blocks++;  
   }  
   else if (need\_blocks < 262)  
   {  
   inode\_buff.i\_size = length;  
   printf("need\_blocks = %d\n", need\_blocks);  
   if (inode\_buff.i\_blocks <= 6)  
   {  
   while (inode\_buff.i\_blocks < 6)  
   {  
   inode\_buff.i\_block[inode\_buff.i\_blocks] = alloc\_block();  
   inode\_buff.i\_blocks++;  
   }  
   inode\_buff.i\_block[6] = alloc\_block();  
   inode\_buff.i\_blocks++;  
   }  
   // 写一级索引 2字节表示1个块号  
   reload\_block(inode\_buff.i\_block[6]);  
   printf("inode\_buff.i\_block[6] = %d\n", inode\_buff.i\_block[6]);  
   for (j = 0; j < need\_blocks - 6; j++)  
   {  
   short block\_num = alloc\_block();  
   inode\_buff.i\_blocks++;  
   Buffer[j \* 2] = block\_num / 256;  
   Buffer[j \* 2 + 1] = block\_num % 256;  
   }  
   update\_block(inode\_buff.i\_block[6]);  
   // 写数据块  
   j = 0;  
   while (j < 6)  
   {  
   reload\_block(inode\_buff.i\_block[j]);  
   memcpy(Buffer, tempbuf + j \* BLOCK\_SIZE, BLOCK\_SIZE);  
   update\_block(inode\_buff.i\_block[j]);  
   j++;  
   }  
    
   for (j = 0; j < need\_blocks - 6; j++)  
   {  
   reload\_block(inode\_buff.i\_block[6]);  
   unsigned short block\_num = (int)Buffer[j \* 2] \* 256 + (int)Buffer[j \* 2 + 1];  
   reload\_block(block\_num);  
   memcpy(Buffer, tempbuf + (j + 6) \* BLOCK\_SIZE, BLOCK\_SIZE);  
   update\_block(block\_num);  
   }  
   update\_inode\_entry(dir[k].inode);  
   reload\_inode\_entry(dir[k].inode);  
   }  
     
   }  
    
   else  
   {  
   printf("Sorry,the max size of a file is 2MB!\n");  
   }  
   }  
   else  
   {  
   printf("The file %s has not opened!\n", tmp);  
   }  
   }  
   else  
   {  
   printf("The file %s does not exist!\n", tmp);  
   }  
  }
* 创建文件需满足两个条件：
  1. 该文件是可写的 2.该文件已打开 open
* 若满足条件即可写入。写入时首先计算所需块数need\_blocks。根据索引个数选择索引方式:
  1. （need\_blocks < 6）直接索引将数据块的位置信息存储在文件的inode项中的i\_blocks数组中**。每个数组元素对应一个数据块的位置**。在代码中，通过更新inode项的i\_blocks数组来分配或释放直接索引所需的数据块。然后，使用循环将数据块中的数据逐块写入磁盘。
  2. （need\_blocks < 262）一级子索引是**一个额外的索引块，其中存储的块号指向一个数据块的位置。索引块中用unsigned int 16位变量**，**即2字节表示1个块号**。索引块都用来存放块号，可以存放512/2＝256个。因为Buffer为char型（8位），所以每得到一个数据块需要取两个Buffer[]代表高 8位和低8位。
  + Buffer[j \* 2] = block\_num / 256;  
     Buffer[j \* 2 + 1] = block\_num % 256;
  1. （need\_blocks < 4072）写入二级索引块。代码通过循环分配一级索引块的块号，**并在每个一级索引块中再次循环分配块号，将这些块的块号存储在二级索引块中的相应位置**。写入数据块。代码通过循环读取tempbuf中的数据，并将其写入对应的数据块中。前6个数据块写入直接索引块，接下来的256个数据块写入一级索引块指向的块，剩余的数据块按照二级索引块的结构依次写入。
  + 

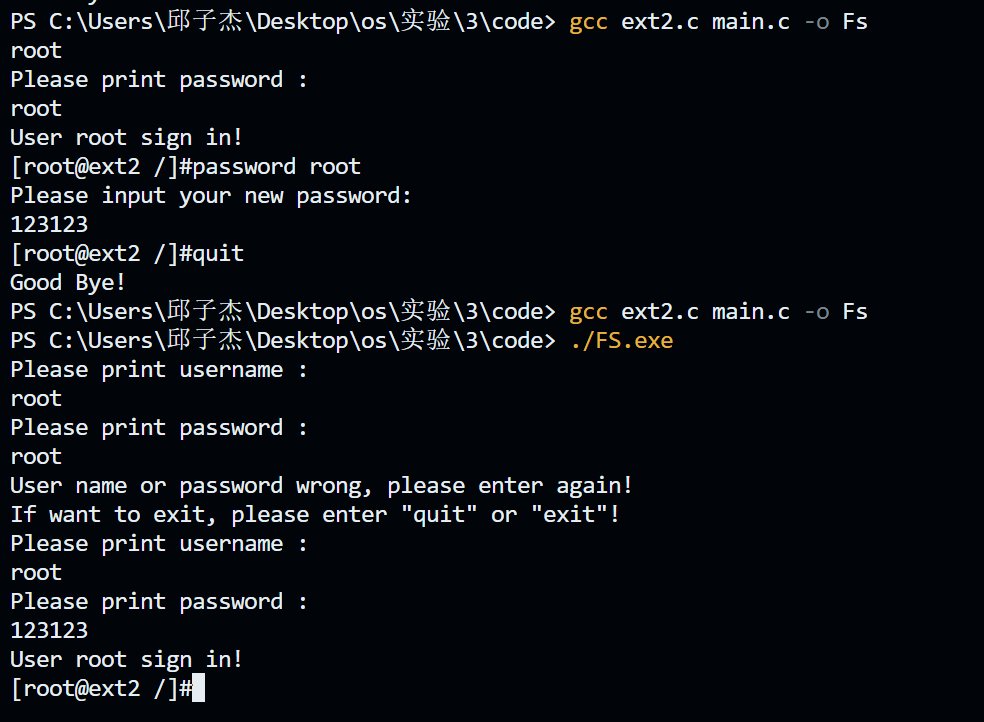
##### ls 显示指定工作目录下之内容

* 

##### chmod 更改文件权限

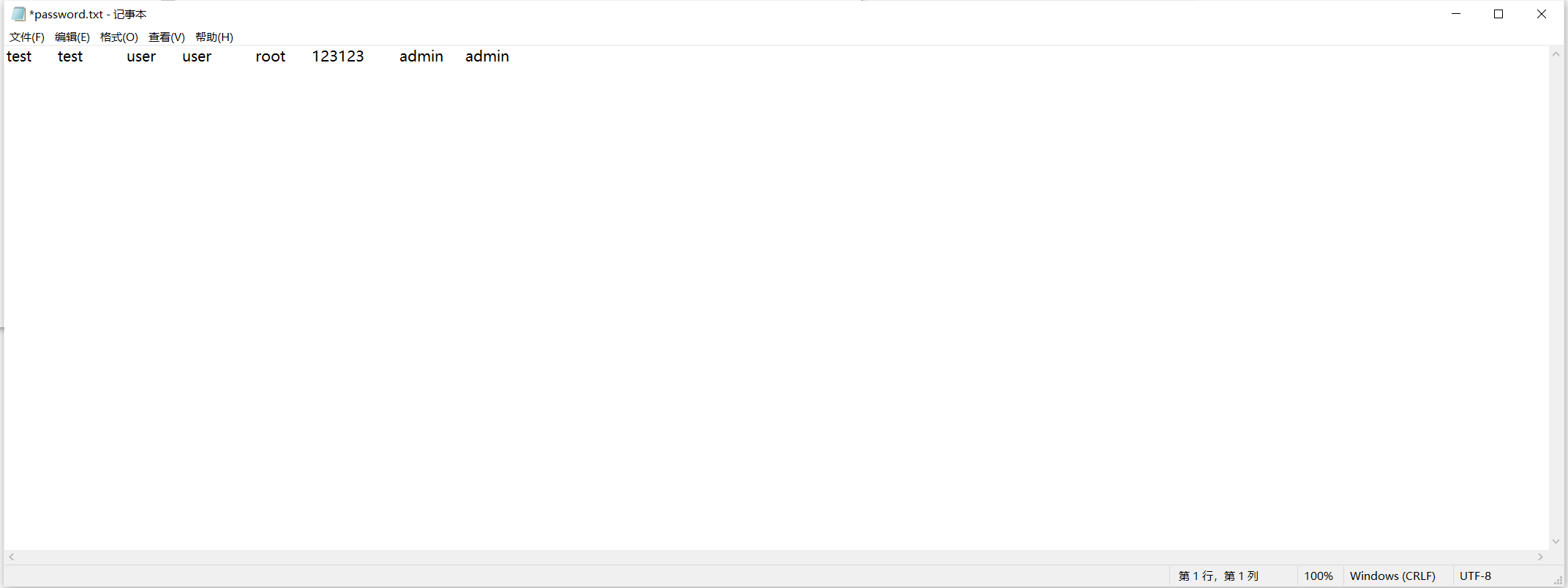
* 
* void chmod(char tmp[100], unsigned short mode)  
  {  
   unsigned short flag, i, j, k;  
   flag = research\_file(tmp, 1, &i, &j, &k);  
   if (flag)  
   {  
   if (mode < 0 || mode > 7)  
   {  
   printf("Wrong mode!\n");  
   return;  
   }  
   reload\_inode\_entry(dir[k].inode);  
   inode\_buff.i\_mode = mode;  
   update\_inode\_entry(dir[k].inode);  
   }  
   else  
   printf("The file %s does not exist!\n", tmp);  
  }
* 检查权限模式是否在0到7之间，如果不在范围内则输出错误信息并返回。如果权限模式在范围内，则重新加载文件的索引节点信息，更新权限模式，并更新索引节点信息。如果文件不存在，则输出文件不存在的错误信息。

##### initialize\_user 初始化用户 **password\_change** 更改密码

* 

// 初始化用户信息  
void initialize\_user()  
{  
 // 创建password.txt  
 FILE \*fp;  
 fp = fopen("./password.txt", "r+");  
 if (fp == NULL)  
 {  
 fp = fopen("./password.txt", "w+");  
 // 初始化用户信息  
 strcpy(User[0].username, "test");  
 strcpy(User[0].password, "test");  
  
 strcpy(User[1].username, "user");  
 strcpy(User[1].password, "user");  
  
 strcpy(User[2].username, "root");  
 strcpy(User[2].password, "root");  
  
 strcpy(User[3].username, "admin");  
 strcpy(User[3].password, "admin");  
 fwrite(User, sizeof(struct user), USER\_MAX, fp);  
 printf("The password.txt has been created!\n");  
 fclose(fp);  
 }  
 // 读取password.txt  
 fp = fopen("./password.txt", "r+");  
 fread(User, sizeof(struct user), USER\_MAX, fp);  
 fclose(fp);  
 return;  
}  
  
// 修改密码  
void password\_change(char username[10], char password[10])  
{  
 for (int i = 0; i < USER\_MAX; i++)  
 {  
 if (!strcmp(User[i].username, username))  
 {  
 strcpy(User[i].password, password);  
 fp = fopen("./password.txt", "w+");  
 fwrite(User, sizeof(struct user), USER\_MAX, fp);  
 break;  
 }  
 }  
 return;  
}

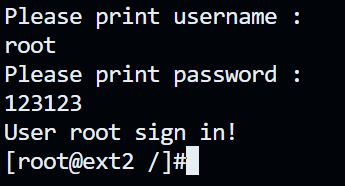
这两个函数共同完成了用户信息的初始化和密码修改的功能。initialize\_user()用于初始化用户信息并创建文件，而password\_change()用于修改特定用户的密码并更新文件

1. initialize\_user():
   * 该函数用于初始化用户信息并创建/读取一个名为password.txt的文件。将预定义的用户信息写入User数组中。
   * 接下来，使用fwrite将User数组中的用户信息写入文件中，并关闭文件。
   * 最后，再次打开password.txt文件以读取其中的用户信息，并将其存储到User数组中
2. password\_change(char username[10], char password[10]):
   * 以写入模式重新打开password.txt文件，并使用fwrite将更新后的User数组写入文件。
   * 
   * password.txt

#### shell 的设计

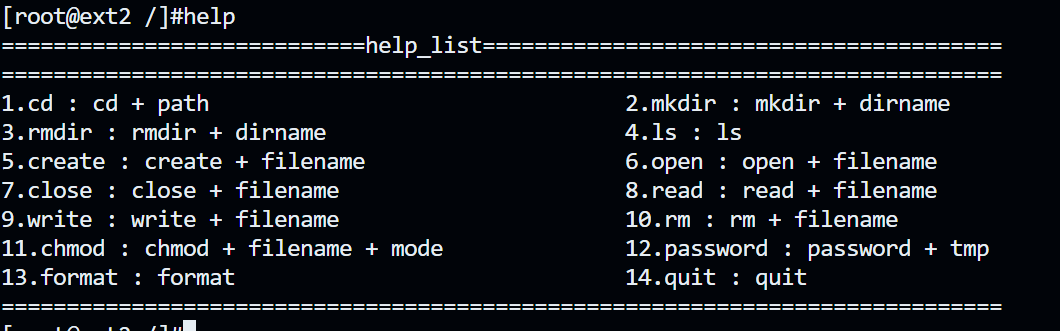
int main()  
{  
 char command[10], temp[100];  
 char username[10], password[10];  
 initialize\_user();  
 while (1)  
 {  
 printf("Please print username : \n");  
 scanf("%s", username);  
 if (!strcmp(username, "quit") || !strcmp(username, "exit"))  
 return 0;  
 printf("Please print password : \n");  
 scanf("%s", password);  
 if (login(username, password))  
 {  
 strcpy(current\_user, username);  
 strcpy(current\_path, "[");  
 strcat(current\_path, current\_user);  
 strcat(current\_path, "@ext2 /");  
 printf("User %s sign in!\n", username);  
 break;  
 }  
 else  
 {  
 printf("User name or password wrong, please enter again!\n");  
 printf("If want to exit, please enter \"quit\" or \"exit\"!\n");  
 }  
 }  
 initialize\_memory();  
 while (1)  
 {  
   
 printf("%s]#", current\_path);  
 scanf("%s", command);  
 if (!strcmp(command, "cd"))  
 { // 进入当前目录下  
 scanf("%s", temp);  
 cd(temp);  
 }  
 else if (!strcmp(command, "mkdir"))  
 { // 创建目录  
 scanf("%s", temp);  
 mkdir(temp, 2);  
 }

**1.shell层进行用户登陆**

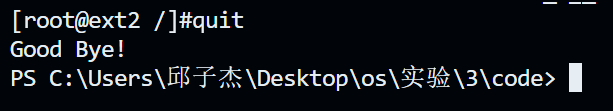


用户登陆界面

**2.执行命令行（包括 “help” 查询操作用法）**



**3.退出文件系统**



**4.格式化系统**

