

设计类大作业：改良 Unix V6++ 文件系统，支持长文件名

学号：1951443

姓名：罗劲桐

1. 实验目的

- 1.1. 分析 FAT32、Linux ext2 文件系统支持长文件名的先进经验，写调研报告
- 1.2. 源码分析 Unix V6++ 系统的原目录结构
- 1.3. 完成系统设计文档
- 1.4. 改进 Unix V6++ 文件系统，实现设计

2. 实验设备及工具

Eclipse IDE 开发环境

Bochs-2.6 虚拟机

UNIX V6++操作系统

3. 调研报告

3.1. FAT32 文件系统支持长文件名的方式

3.1.1. 文件系统结构

FAT32 文件系统由原先的 FAT12 文件系统扩展而成，支持更大的磁盘容量，和长文件名、更大尺寸文件和引导扇区备份。这里主要对长文件名的实现做调研。

FAT32 文件系统由引导扇区、FAT 表、数据区组成。

3.1.1.1. 引导扇区

引导扇区（Boot Sector）位于硬盘第一扇区中，与 Unix 系统中 SuperBlock 作用类似。其中 BPB_RootClus 记录了 FAT32 文件系统根目录对应起始簇号，一般为 2，这是由于 FAT32 文件系统根目录与其他常规目录同样位于 FAT 表和对应的数据区，因此 FAT32 文件系统根目录位于数据区的起始簇中。

3.1.1.2. FSInfo 扇区

FAT32 文件系统特有。由于 FAT32 文件系统 FAT 表的大小巨大，FSInfo 扇区辅助 FAT 表保存文件系统的保留区域。

3.1.1.3. FAT 表

对于 FAT32 文件系统，每个 FAT 表项占 32bit，实际上仅低 28bit 有效，高 4bit 为保留位。FAT[0]、FAT[1]不作为数据区的索引值使用，FAT 表开始为 2。

表13-3 FAT表项取值说明

FAT项	实例值	功能描述
0	0FFFFFF8h	磁盘标示字，低字节与BPB_Media数值一致
1	FFFFFFFFh	第一个簇已经被占用
2	00000003h	x0000000h: 可用簇 x0000002h~xFFFFFFFh: 已用簇，标识下一个簇的簇号 xFFFFFF0h~xFFFFFF6h: 保留簇 xFFFFFF7h: 坏簇 xFFFFFF8h~xFFFFFFFh: 文件的最后一个簇
3	00000004h	
.....	
N	0FFFFFFFh	
N+1	00000000h	
.....	

注：x ∈ (0x0 ~ 0xF)。

3.1.1.4. 根目录和数据区

FAT32 文件系统根目录包含在数据区中，因此能够动态增长长度，取消根目录文件数量限制。同时引入长目录结构，最多支持文件名 255 字符，通过文件属性标志位区别短目录和长目录结构。

3.1.2. 支持长文件名的方式

3.1.2.1. 短目录结构

其中文件名由 8B 的基础名和 3B 的扩展名组成，全长 11B，它们只能保存字母、数字以及有限的几个字符，如果这两部分字符串的长度不足，将使用空格符 (0x20) 补齐。文件名字符串的第一个字节还拥有其他特殊功能，当 DIR_Name[0]为 0xE5、0x00 或 0x05 时，表明此目录项为无效目录项或空闲目录项，而且 DIR_Name[0]的数值不允许为 0x20（空格符），同时基础名字符串间也不允许出现空格符。另外，段目录项仅能保存全部为大写字母的文件名。

具体结构见下表：

表13-4 短目录项结构表

名 称	偏移	长度	功能描述
DIR_Name	0	11	基础名8 B, 扩展名3 B
DIR_Attr	11	1	文件属性: 0x01=ATTR_READ_ONLY (只读) 0x02=ATTR_HIDDEN (隐藏) 0x04=ATTR_SYSTEM (系统文件) 0x08=ATTR_VOLUME_ID (卷标) 0x10=ATTR_DIRECTORY (目录) 0x20=ATTR_ARCHIVE (存档) 0x0F=ATTR_LONG_NAME (长文件名)
DIR_NTRes	12	1	保留使用
DIR_CrtTimeTenth	13	1	文件创建的毫秒级时间戳
DIR_CrtTime	14	2	文件创建时间
DIR_CrtDate	16	2	文件创建日期
DIR_LastAccDate	18	2	最后访问日期
DIR_FstClusHI	20	2	起始簇号 (高字)
DIR_WrtTime	22	2	最后写入时间
DIR_WrtDate	24	2	最后写入日期
DIR_FstClusLO	26	2	起始簇号 (低字)
DIR_FileSize	28	4	文件大小

3.1.2.2. 长目录项

长目录项结构扩展于短目录项, 它将文件名的编码方式从 ASCII 码升级为 Unicode 码, 使得文件名不仅可以区分大小写字母, 同时还支持更多种语言符号。长目录项紧随短目录项之后, 表 13-7 是长目录项的详细结构说明。

LDIR_Name1/2/3 是长目录项的三个字符串存储区域, 字符串中的每个字符用 2B 的 Unicode 码表示, 并以空字符 (NUL) 结尾。剩余字符串空间以 0xPFF 填充, 一个完整的长文件名是由一组连续的长目录项组成。

长目录项序号。长目录项的起始序号为 1, 对于记录长文件名的最后一个长目录项而言, 其序号成员变量的第 6 位必须置位 (LAST_LONG_ENTRY (0x40) | N) 以表示结尾。

表13-7 长目录项结构表

名 称	偏移	长度	功能描述
LDIR_Ord	0	1	长目录项的序号
LDIR_Name1	1	10	长文件名的第1~5个字符, 每个字符占2 B
LDIR_Attr	11	1	文件属性必须为ATTR_LONG_NAME

名 称	偏移	长度	功能描述
LDIR_Type	12	1	如果为0,说明这是长目录项的子项
LDIR_Chksum	13	1	短文件名的校验和
LDIR_Name2	14	12	长文件名的第6~11个字符,每个字符占2 B
LDIR_FstClusLO	26	2	必须为0
LDIR_Name3	28	4	长文件名的第12~13个字符,每个字符占2 B

3.1.3. 特别

创建新文件时,若不满足短目录项名规则,都会创建短目录项和长目录项的组合方式。(或使用 DIR_NTRes 实现简单的基础名、扩展名格式变换)

FAT32 文件系统还要求同一目录里的长短文件名是唯一的。为了保证短文件名的唯一性, FAT32 文件系统以 “部分文件名”+“~N”的方式将重复的短文件名变成唯一的文件名标识。此处的字母 N 代表数字字符,它的取值范围是 1-9999 果重复的短文件名过多, FAT32 文件系统会使用类似的快速法去创建短文件名。

3.1.4. 索引方式

扫描目录项时,若当前目录项偏移为 11 的位置 LDIR_Atrr=ATTR_LONG_NAME (0x0F),则他是上一个短目录项附属的长目录项,此后扫描的长目录项号 LDIR_Ord=1,2...n,直到某一个目录项号 LDIR_Ord 第 6 位为 1,代表长目录项的结尾。

3.2. Linux ext2 文件系统支持长文件名的方式

3.2.1. 文件系统结构

ext2 分区中的块被划分为由连续的块构成的簇,我们称之为块组(block group)。文件系统试图在同一块组中存储相关的数据。由于每个块组中的块都位于磁盘中的毗邻区域内,所以这样的布局可以缩短访问大组相关数据

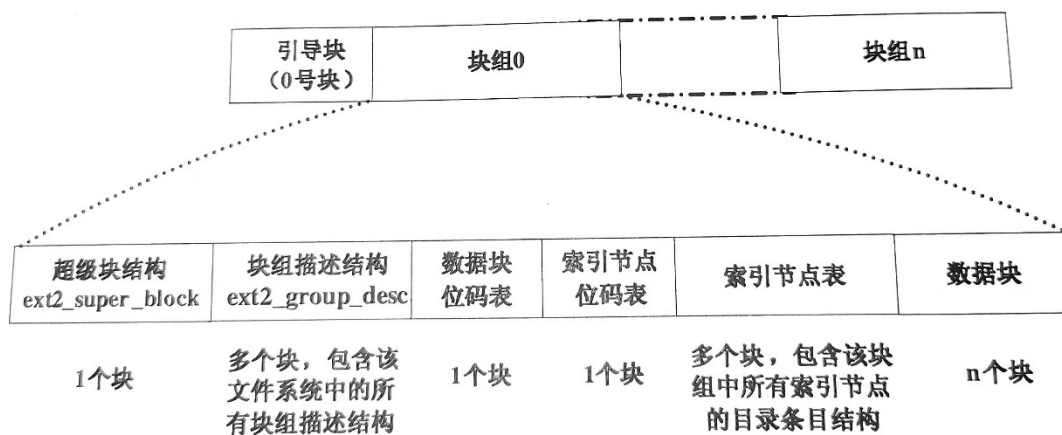


图 20-5 ext2 文件系统磁盘分区格式

3.2.1.1. 超级块

超级块 `ext2_super_block` 与 Unix 中 `SuperBlock` 对应, 包含了整个文件系统的关键信息, 通常有很多副本。

3.2.1.2. 组描述符

组描述结构 `ext2_group_desc` 是每一组的 `SuperBlock`, 包含与索引节点分配位图的位置、块分配位图以及索引节点表相对应的块数量, 在该组中空闲块和空闲索引节点的数量等。

3.2.1.3. 索引结点表

索引节点表 (`inode table`), 它包含了块组中每个索引节点结构。由于索引节点表的大小是固定的, 所以在一个已经格式化的 `ex12` 文件系统中增加索引节点数量的唯一方法是, 增加文件系统的容量。Inode 不包含文件名属性, 只包含文件连接关系、大小和文件指针等信息。

3.2.1.4. 位图

索引节点分配位图 (`inode allocation bitmap`) 的块, 该索引节点分配位图在块组中的作用是, 记录在块组中索引节点的使用情况。分配位图中的每个位都对一个该组索引节点表的条目。

块组使用相同的策略来维护块分配位图 (`block allocation bitmap`), 块分配位图是用来记录每个组的块的使用情况。

3.2.2. ext2 文件系统支持长文件名的方式

在 ext2 文件系统中，目录是作为文件存储的。根目录总是在 inode 表的第二项，而其子目录则在根目录文件的内容中定义。

■ struct ext2_dir_entry_2 ext2 目录条目结构 2

```
__le32  inode: 索引节点号
__le16  rec_len: 目录条目长度，该结构的总长度，必须是 4 的倍数，否则在末尾填充 0，它表示的是同一目录下目录条目结构相对于该目录条目结构的位置（单位为字节数）
__u8    name_len: 名字长度
__u8    file_type: 文件类型编号，ext2 文件系统中定义了 7 种文件类型：普通文件、目录、字符设备、块设备、命名管道（FIFO）、套接字（socket）、符号链接，它们的文件类型编号分别为 1~7
char    name[EXT2_NAME_LEN] : 文件名，变长数组，最大为 EXT2_NAME_LEN (=255)
```

目录的内容实际就是连续存放的 ext2_dir_entry_2 目录条目结构，位于数据块部分存储。由于 name 数组长度可变，可最大支持 255 长度的文件名。
每个目录条目结构描述了该目录下的一个文件(可以是 7 种文件中的任何类型)

表 20-2 ext2 文件系统中的文件类型编号

文件类型	文件类型编号 (file_type)
未知	0
普通文件	1
目录	2
字符设备	3
块设备	4
命名管道 (FIFO)	5
套接字 (socket)	6
符号链接	7

比如一个目录下有 mydir 子目录、tmp 子目录、oldfile 文件以及 buid 子目录，但是 oldfile 经被删除了，那么该目录的内容如下。

相对位置	索引节点号	下一条 目位置	名字 长度	文件 类型	文件名			
0	21	12	1	2	.	\0	\0	\0
12	22	12	2	2	.	.	\0	\0
24	56	16	5	2	m	y	d	i
40	68	28	3	2	t	m	p	\0
52	0	16	7	1	o	l	d	f
68	55	16	5	2	b	u	i	d

图 20-6 ext2 文件系统磁盘目录内容举例

3.2.3. 索引过程

当要访问或搜索一个文件夹下的一个文件（夹）名时，从 ext2_inode 文件索引中取 i_faddr 得到当前目录文件在磁盘的位置。读文件获得 ext2_dir_entry_2 目录条目结构的数组，此时与所有目录条目文件名 name 比较，并获取对应的 inode，就得到了要搜索的文件（夹）的 inode。

4. Unix V6++ 源码分析

4.1. Unix 文件系统结构



4.1.1. 超级块

记录了 Inode 区和数据区的分配情况，用 `s_fsize`、`s_nfree`、`s_free` 表示数据区大小和空闲盘块分布情况，`s_iseize`、`s_ninode`、`s_inode` 表述 Inode 区大小和空闲 Inode 分布情况。

`s_flock`、`s_ilock` 是对这两个区修改时需要上的锁。`s_fmod` 是 SuperBlock 块为脏标记。`s_ronly` 文件系统只读。`s_time` 最近访问时间。`padding` 填充区，无意义。

```
1. class SuperBlock
2. {
3.     /* Functions */
4. public:
5.     /* Constructors */
6.     SuperBlock();
7.     /* Destructors */
8.     ~SuperBlock();
9.
10.    /* Members */
11. public:
12.    int          s_iseize;          /* 外存 Inode 区占用的盘块数 */
13.    int          s_fsize;          /* 盘块总数 */
14.
15.    int          s_nfree;          /* 直接管理的空闲盘块数量 */
16.    int          s_free [100];    /* 直接管理的空闲盘块索引表 */
17.
18.    int          s_ninode;         /* 直接管理的空闲外存 Inode 数量 */
19.    int          s_inode[100];    /* 直接管理的空闲外存 Inode 索引表 */
20.
21.    int          s_flock;          /* 封锁空闲盘块索引表标志 */
22.    int          s_ilock;          /* 封锁空闲 Inode 表标志 */
23.
24.    int          s_fmod;           /* 内存中 super block 副本被修改
    标志，意味着需要更新外存对应的 Super Block */
25.    int          s_ronly;          /* 本文件系统只能读出 */
26.    int          s_time;           /* 最近一次更新时间 */
27.    int          padding[47];      /* 填充使 SuperBlock 块大小等于 1024 字节，
    占据 2 个扇区 */
28. };
```

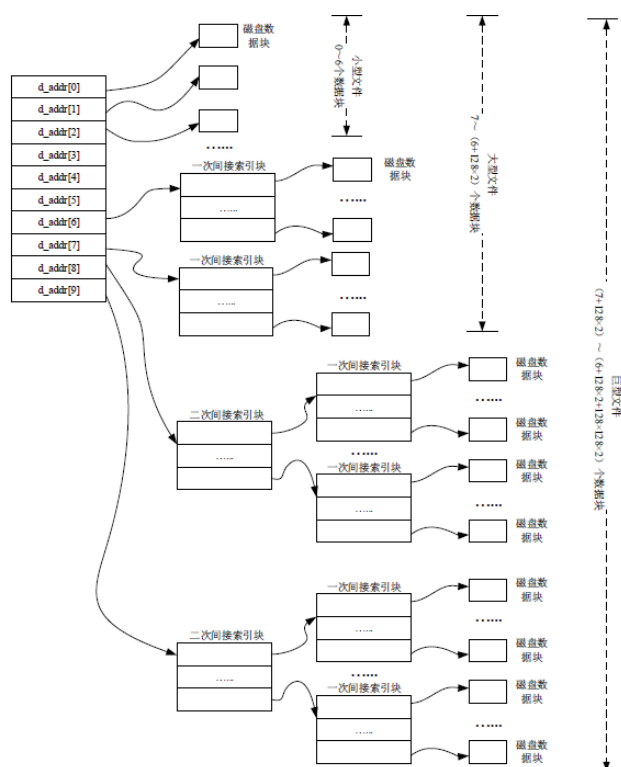
4.1.2. Inode 区

外存索引节点 (DiskInode) 位于外存 Inode 区中。每个文件有唯一对应的

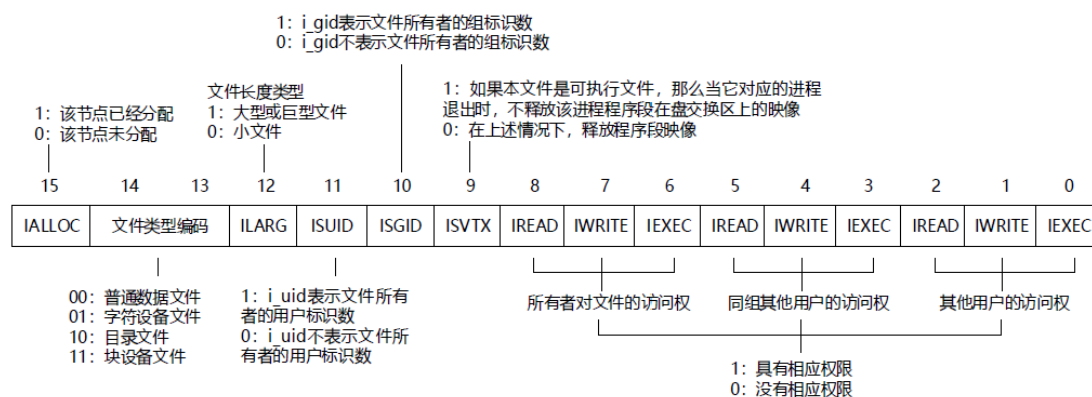
DiskInode，记录了该文件对应的控制信息。DiskInode 对象长度为 64 字节，每个磁盘块可以存放 8 个 DiskInode。当 DiskInode 从磁盘中读入时，使用 Inode::ICopy 复制到 Inode 类中，在内存中组织文件索引。

d_nlink 文件的硬链接数。d_uid 文件所有者的用户 uid。d_gid 文件所有者的组 gid。d_size 文件长度。d_atime 文件的最后访问时刻。d_mtime 文件的最后修改时刻。

d_addr 如图，文件混合索引树的根，指向数据块：



d_mode 如图所示：



```
1. class DiskInode
2. {
3.     /* Functions */
4. public:
```



```

5.  /* Constructors */
6.  DiskInode();
7.  /* Destructors */
8.  ~DiskInode();
9.
10. /* Members */
11. public:
12.  unsigned int d_mode;          /* 状态的标志位, 定义见 enum InodeFlag */
13.  int          d_nlink;         /* 文件联结计数, 即该文件在目录树中不同路径
    名的数量 */
14.
15.  short        d_uid;          /* 文件所有者的用户标识数 */
16.  short        d_gid;          /* 文件所有者的组标识数 */
17.
18.  int          d_size;          /* 文件大小, 字节为单位 */
19.  int          d_addr[10];      /* 用于文件逻辑块好和物理块好转
    换的基本索引表 */
20.
21.  int          d_atime;         /* 最后访问时间 */
22.  int          d_mtime;         /* 最后修改时间 */
23. };
24.

```

4.2. 目录文件和目录项

DirectoryEntry 是外存中保存文件名和 Inode 索引的数据结构。另外 User 结构中保存了目录读写过程中需要保存的部分中间量: u_dent 在每一次比较前记录当前目录的目录项; u_dbuf[DirectoryEntry::DIRSIZ] 是 Pathname 中当前准备进行匹配的路径分量。

```

1.  class DirectoryEntry
2.  {
3.      /* static members */
4.  public:
5.      static const int DIRSIZ = 28; /* 目录项中路径部分的最大字符串长度 */
6.
7.      /* Functions */
8.  public:
9.      /* Constructors */
10.     DirectoryEntry();
11.     /* Destructors */
12.     ~DirectoryEntry();
13.
14.     /* Members */
15.  public:
16.     int m_ino;          /* 目录项中 Inode 编号部分 */
17.     char m_name[DIRSIZ]; /* 目录项中路径名部分 */
18. };

```

4.2.1. 目录项的读取过程

在 NameI 中, 通过 Bread(pInode->i_dev, phyBlkno) 按照目录层级顺序读目录文件, 并使用 pBuf->b_addr + (u.u_IOParam.m_Offset % Inode::BLOCK_SIZE) 按顺序依次获取每个目录项对应的缓存块位置, 并使用 Utility::DWordCopy 拷

贝到&u.u_dent 当前目录项。每次进行路径比较时，比较 u.u_dbuf 和 u.u_dent.m_name，比较成功则取 u.u_dent.m_ino 取下一 Inode 循环继续向下比较，否则路径不存在。

4.2.2. 目录项写回过程

在 WriteDir 中，构造 u_IOParm 写指令，长度为一个目录项，内容为 u_dent 中对应目录项，添加在父目录文件尾部 u.u_pdir->WriteI()。随后，m_InodeTable->IPut 释放当前对 m_InodeTable 的引用。

4.2.3. 目录项删除操作

在 UnLink 中，构造 u_IOParm 指令写入清零后的目录项，m_Offset 向前移动 DirectoryEntry::DIRSIZ + 4 位，u_dent.m_ino 改为 0（表示该项为空），然后父目录对应目录项位置覆盖原记录 pDeleteInode->WriteI()。随后，m_InodeTable->IPut 释放当前对 m_InodeTable 的引用。

5. 系统设计文档

5.1. 总体思想

由于 Unix 中 DiskInode 位于 Inode 区而文件名对应存储结构 DirectoryEntry 存储于数据区，与 Linux ext2 文件系统相似。考虑修改 DirectoryEntry 数据结构，实现可变长度的目录项名。

修改 DirectoryEntry 为如下 DiskDirectoryEntry 结构：

```
1. class DiskDirectoryEntry
2. {
3.     /* static members */
4. public:
5.     static const int DIRSIZ = 24; /* DiskDirectoryEntry 路径部分的长度 */
6.     static const int ADD_DIRSIZ = 128; /* 目录项中路径部分的最大字符串长度 */
7.
8.     /* Functions */
9. public:
10.    /* Constructors */
11.    DiskDirectoryEntry();
12.    /* Destructors */
13.    ~DiskDirectoryEntry();
14.
15.    /* Members */
16. public:
17.    int m_ino; /* 目录项中 Inode 编号部分 */
18.    char m_name[DIRSIZ]; /* 目录项中路径名部分 */
19.    int m_addition_esize; /* 附加段个数 <= (ADD_DIRSIZ/32) */
20.};
```

每个目录项都具有如此一个头项，其中 `m_ino` 记录文件（夹）对应的磁盘 Inode 号，`m_name` 是其前 24 个字符，当其文件名长度 $\leq 24B$ 时，不需要附加段，`m_addition_esize=0`。`m_addition_esize` 表示文件的附加段个数，当文件名长度较长时，在原目录项后增加附加段，每段 32B，全部用于存储目录项存不下的文件名部分。

规定长文件名最长 $ADD_DIRSIZ(4*32)+DIRSIZ(24)=152$ 。因此每个目录项最多添加 4 个附加段。

5.2. User 结构的修改

```
1.  DiskDirectoryEntry u_dent;                      /* 当前目录
   的目录项 */
2.  char addition_entry[DiskDirectoryEntry::ADD_DIRSIZ];
3.  char u_dbuf[DiskDirectoryEntry::ADD_DIRSIZ + DiskDirectoryEntry::DIRSIZ];
   /* 当前路径分量 */
4.  char u_curdir[512];                             /* 当前工作
   目录完整路径 */
```

增加 `addition_entry` 部分，用于暂存 `u_dent` 对应附加段内容。相应的，用于比较的临时数组 `u_dbuf` 增加大小，完整路径 `u_curdir` 也增加大小。

5.3. FileManager::NameI 的修改

5.3.1. 长度修改

对于每次 `u_dbuf` 的引用，将它的读取长度扩展到 `DiskDirectoryEntry::ADD_DIRSIZ + DiskDirectoryEntry::DIRSIZ`，其余相似的修改不再赘述。

```
1.  while ( '/' != curchar && '\0' != curchar && u.u_error ==
   User::NOERROR )
2.  {
3.      if ( pChar < &(u.u_dbuf[DiskDirectoryEntry::ADD_DIRSIZ +
   DiskDirectoryEntry::DIRSIZ]) )
4.      {
5.          *pChar = curchar;
6.          pChar++;
7.      }
8.      curchar = (*func)();
9.  }
```

目录项长度修改，如图

```
1.  /* 设置为目录项个数，含空白的目录项*/
2.  u.u_IOParam.m_Count = pInode->i_size / (DiskDirectoryEntry::DIRSIZ +
   8);
```

5.3.2. 为 WriteDir 分配空闲盘块

当获取空闲盘块时。由于使用附加块，首先判断空闲部分是否能够装下新块。

使用 `use_entries=(length-DiskDirectoryEntry::DIRSIZ+sizeof(DiskDirectoryEntry)-1)/sizeof(DiskDirectoryEntry)` 计算新目录项需要的附加块个数和并和 `m_addition_esize` 判断是否能够写入。

如果能够装下，且有剩余空间，需要将原有的目录项和附加段部分分为两部分，前面一部分空闲盘块留给新写入的目录项，后一部分的头部创建一个新的目录项头，用于管理接下来的空闲块。

使用 `DiskDirectoryEntry newEntry` 和 `newEntry.m_addition_esize=u.u_dent.m_addition_esize-use_entries-1` 创建新的目录项头，并写入磁盘缓冲块 `((use_entries*sizeof(DiskDirectoryEntry)+u.u_IOParam.m_Offset) % Inode::BLOCK_SIZE)` 位置。写入前，如果与当前目录项不在同一盘块，需要 `Buf* newBuf` 申请新缓冲块并写入。

```
1.          /* 如果是空闲目录项，记录该项位于目录文件中偏移量 */
2.          if ( 0 == u.u_dent.m_ino )
3.          {
4.              if ( 0 == freeEntryOffset )
5.              {
6.                  int
length=Utility::StringLength(u.u_dirp);
7.                  int use_entries=(length-
DiskDirectoryEntry::DIRSIZ+sizeof(DiskDirectoryEntry)-
1)/sizeof(DiskDirectoryEntry);
8.
if(use_entries<=u.u_dent.m_addition_esize)
9.                {
10.                    freeEntryOffset =
u.u_IOParam.m_Offset;
11.
if(use_entries<=u.u_dent.m_addition_esize)
12.                {
13.
if(use_entries*sizeof(DiskDirectoryEntry)+u.u_IOParam.m_Offset>=Inode::BLOCK_SIZE)
14.                {
15.                    /* 计算要读的物理盘块号 */
16.                    int
phyBlkno =
pInode->Bmap((use_entries*sizeof(DiskDirectoryEntry)+u.u_IOParam.m_Offset) /
Inode::BLOCK_SIZE );
17.                    Buf*
newBuf = bufMgr.Bread(pInode->i_dev, phyBlkno );
18.                    int* src =
(int *) (newBuf->b_addr +
((use_entries*sizeof(DiskDirectoryEntry)+u.u_IOParam.m_Offset) %
Inode::BLOCK_SIZE));
```

```

19.     DiskDirectoryEntry newEntry;
20.     newEntry.m_addition_esize=u.u_dent.m_addition_esize-use_entries-1;
21.     Utility::DWordCopy( src, (int *)&newEntry,
sizeof(DiskDirectoryEntry)/sizeof(int) );
22.                                     if
( NULL != newBuf )
23.                                     {
24.         bufMgr.Brelse(newBuf);
25.                                     }
26.                                     }
27.                                     }
28.                                     }
29.     }
30.     /* 跳过空闲目录项，继续比较下一目录项 */
31.     u.u_IOParam.m_Offset +=
(DiskDirectoryEntry::DIRSIZ + 8) * u.u_dent.m_addition_esize;
32.     continue;
33. }

```

5.3.3. 目录读取

当前目录项有附加块时，即 `u.u_dent.m_addition_esize > 0`。继续向下读取 `m_addition_esize` 个附加块，然后将其拷贝到 `addition_entry` 数组中。

通过 `int* src = (int *) (pBuf->b_addr + (u.u_IOParam.m_Offset % Inode::BLOCK_SIZE))` 计算写入缓存块的目标地址和 `Utility::DWordCopy` 写入缓存块。

```

1.     /* 有附加块，继续向下读取*/
2.     if(u.u_dent.m_addition_esize > 0)
3.     {
4.         int addition_index=1;
5.         while(addition_index <=
u.u_dent.m_addition_esize)
6.         {
7.                                     /* 已读完目录文件的当前盘块，需要读入下一目
录项数据盘块 */
8.                                     if ( 0 == u.u_IOParam.m_Offset %
Inode::BLOCK_SIZE )
9.                                     {
10.                                         if ( NULL != pBuf )
11.                                         {
12.                                             bufMgr.Brelse(pBuf);
13.                                         }
14.                                         /* 计算要读的物理盘块号 */
15.                                         int phyBlkno =
pInode->Bmap(u.u_IOParam.m_Offset / Inode::BLOCK_SIZE );
16.                                         pBuf =
bufMgr.Bread(pInode->i_dev, phyBlkno );
17.                                     }
18.         }

```

```

19.             int* src = (int *) (pBuf->b_addr +
    (u.u_IOParam.m_Offset % Inode::BLOCK_SIZE));
20.             Utility::DWordCopy( src, (int
    *)&u.addition_entry[(addition_index-1)*sizeof(DiskDirectoryEntry)],
    sizeof(DiskDirectoryEntry)/sizeof(int) );
21.
22.             u.u_IOParam.m_Offset +=
    sizeof(DiskDirectoryEntry);
23.             u.u_IOParam.m_Count--;
24.             addition_index++;
25.         }
26.     }

```

5.3.4. 循环比较时，先和 u.u_dent.m_name 比较前部，再和 u.addition_entry 比较后部。

```

1.             if(i < DiskDirectoryEntry::DIRSIZ)
2.             {
3.                 if ( u.u_dbuf[i] !=
    u.u_dent.m_name[i] )
4.                 {
5.                     break;    /* 匹配至某一字符不
    符，跳出 for 循环 */
6.                 }
7.             }
8.             else if(i < DiskDirectoryEntry::DIRSIZ +
    u.u_dent.m_addition_esize * sizeof(DiskDirectoryEntry))
9.             {
10.                if ( u.u_dbuf[i] != u.addition_entry[i-
    DiskDirectoryEntry::DIRSIZ] )
11.                {
12.                    break;    /* 匹配至某一字符不
    符，跳出 for 循环 */
13.                }
14.            }

```

5.4. FileManager::WriteDir 的修改

写盘时，首先进行附加段长度计算 m_addition_esize，然后文件名分段写入 u.u_dent.m_name 和 u.addition_entry。最后调用 u.u_pdir->WriteI() 写回。

```

1.     u.u_dent.m_addition_esize = (Utility::StringLength(u.u_dbuf) -
    DiskDirectoryEntry::DIRSIZ + sizeof(DiskDirectoryEntry)-1) /
    sizeof(DiskDirectoryEntry);
2.     int i;
3.     for ( i = 0; i < DiskDirectoryEntry::DIRSIZ + DiskDirectoryEntry::ADD_DIRSIZ;
    i++ )
4.     {
5.         if(i < DiskDirectoryEntry::DIRSIZ)
6.         {
7.             u.u_dent.m_name[i] = u.u_dbuf[i];
8.         }
9.         else if(i < DiskDirectoryEntry::DIRSIZ + u.u_dent.m_addition_esize *
    sizeof(DiskDirectoryEntry))
10.        {
11.            u.addition_entry[i-DiskDirectoryEntry::DIRSIZ] =
    u.u_dbuf[i];

```

```

12.         }
13.     }

```

5.5. FileManager::UnLink 的修改

对当前目录项执行清空写入时，首先写指针向前移动到目录项头部，然后清空 `m_ino=0`，表示该目录项不占有空间，即删除掉了。但 `m_addition_esize` 需要保留，因为目录项头部数据仍记录着之后附加块的数量。

```

1.  /* 写入清零后的目录项 */
2.  u.u_IOParam.m_Offset -= (DiskDirectoryEntry::DIRSIZ + 8 +
    u.u_dent.m_addition_esize * sizeof(DiskDirectoryEntry));
3.  u.u_IOParam.m_Base = (unsigned char *)&u.u_dent;
4.  u.u_IOParam.m_Count = DiskDirectoryEntry::DIRSIZ + 8 +
    u.u_dent.m_addition_esize * sizeof(DiskDirectoryEntry);
5.
6.  for (int i = 0; i < DiskDirectoryEntry::ADD_DIRSIZ; i++ )
7.  {
8.      u.addition_entry[i] = '\0';
9.  }
10. u.u_dent.m_ino = 0;
11. //u.u_dent.m_addition_esize = 0;      此项保留
12. Diagnose::Write("u.u_IOParam.m_Offset: %d\n",u.u_IOParam.m_Offset);
13. Diagnose::Write("UnLink::u.u_dent::m_name: %s,
    m_addition_esize: %d\n",u.u_dent.m_name,u.u_dent.m_addition_esize);
14. Diagnose::Write("UnLink::u.addition_entry: %s\n",u.addition_entry);
15. pDeleteInode->WriteI();

```

5.6. ls 命令的修改

5.6.1. 首先需要增大 item 目录项缓存的大小，修改为 160。

5.6.2. 空目录项处理

读到空目录项 `m_ino=0` 时，根据 `m_addition_esize` 的大小向后 `seek()` 移动指针到下一个目录项

```

1.         if(*((int*)item)==0)//This is very important!
2.             //When some dir has been deleted, their dir name may
3.             //still exit in the item. The fact that the inode number
4.             //of an directory is "0" indicates that the dir is invalid.
5.             {
6.
7.                 for(j=0;j<160;j++)
8.                     item[j]='\0';
9.                 int addition_esize=*((int*)(item+28));
10.                //printf("addition_esize: %d\n",addition_esize);
11.                seek(fd,addition_esize*32,1);
12.                count=read(fd,item,32);
13.            }

```

5.6.3. 读取正常目录

读到正常目录项时, 获取 `addition_esize`, 并按顺序向下读 `addition_esize*32B`, 并粘贴在 `item` 字符串后部 (即 `item+28` 位置)

```
1.             int addition_esize=((int*)(item+32-4));
2.
3.             //printf("addition_esize: %d\n" ,addition_esize);
4.             int k;
5.             for(k=1;k<=addition_esize;k++)
6.             {
7.                 count=read(fd,item+28+(k-1)*32,32);
8.                 if(count!=32)
9.                 {
10.                     printf("error reading long
11. name: count %d\n",count);
12.                 }
13.             }
14.             printf("%s\t" ,item+4);
15.             for(j=0;j<160;j++)
16.             item[j]='\0';
17.             count=read(fd,item,32);
```

5.6.4. `ls` 的另一部分代码 (`ls -l`) 也向上如此修改, 不再赘述。

5.7. `rm` 命令修改

5.7.1. 目录项读取

对于 `rm -r`, 方法与 `ls -l` 一致, 分为空目录项 `m_ino=0` 的处理和真目录项的向后读取, 并添加到 `pathbuf` 路径。

```
1.             if(*((int*)item)==0)
2.             {
3.                 for(j=0;j<160;j++)
4.                 item[j]='\0';
5.                 int addition_esize=((int*)(item+28));
6.                 //printf("addition_esize: %d\n" ,addition_esize);
7.                 seek(fd,addition_esize*32,1);
8.                 count=read(fd,item,32);
9.             }
10.            else
11.            {
12.
13.                 int addition_esize=((int*)(item+32-
14. 4));
15.                 //printf("addition_esize: %d\n" ,addition_esize);
16.                 int k;
17.                 for(k=1;k<=addition_esize;k++)
18.                 {
19.                     count=read(fd,item+28+(k-
20. 1)*32,32);
21.                     if(count!=32)
22.                     {
```



```

20.                                     printf("error
    reading long name" ,item+4);
21.                                     }
22.                                     }
23.
24.         strcpy(pathbuf,path);//Combine parent path
25.             //and current item path
26.         strcat(pathbuf,"/");
27.         strcat(pathbuf,item+4);//to get real path
28.
29.         rm_r(pathbuf);
30.         //clear pathbuf
31.         for(j=0;j<100;j++)
32.             pathbuf[j]='\0';
33.         //clear item:
34.         for(j=0;j<160;j++)
35.             item[j]='\0';
36.         count=read(fd,item,32);
37.     }

```

5.7.2. rm 命令的无参过程似乎有误，对返回结果判断不正确，做出了修改

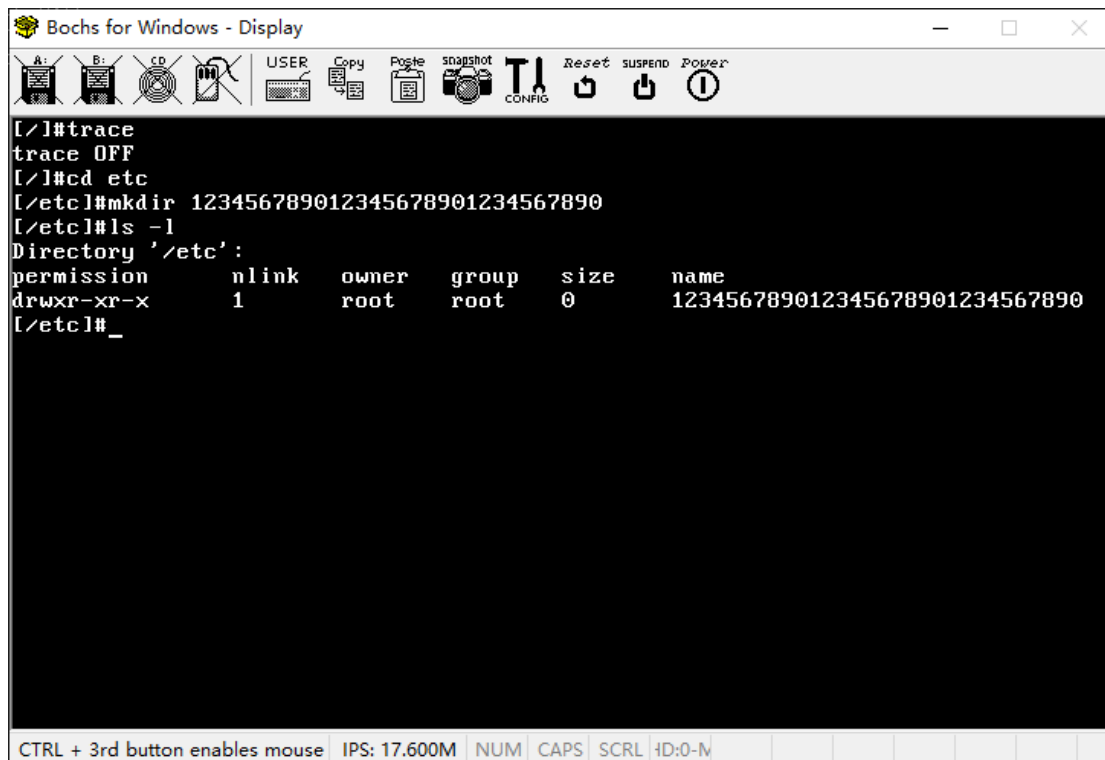
```

1.         int stat_ret=stat(path[i],&inode);
2.         if(stat_ret==-1)
3.             {
4.                 //printf("stat_ret: %d\n",stat_ret);
5.                 printf("Wrong file \'%s\'!\n",path[i]);
6.             }
7.         else
8.             {
9.                 if((inode.st_mode&0x4000)==0)
10.                    unlink(path[i]);
11.                 else
12.                    printf("\'%s\' is a directory!\n",path[i]);
13.
14.             }

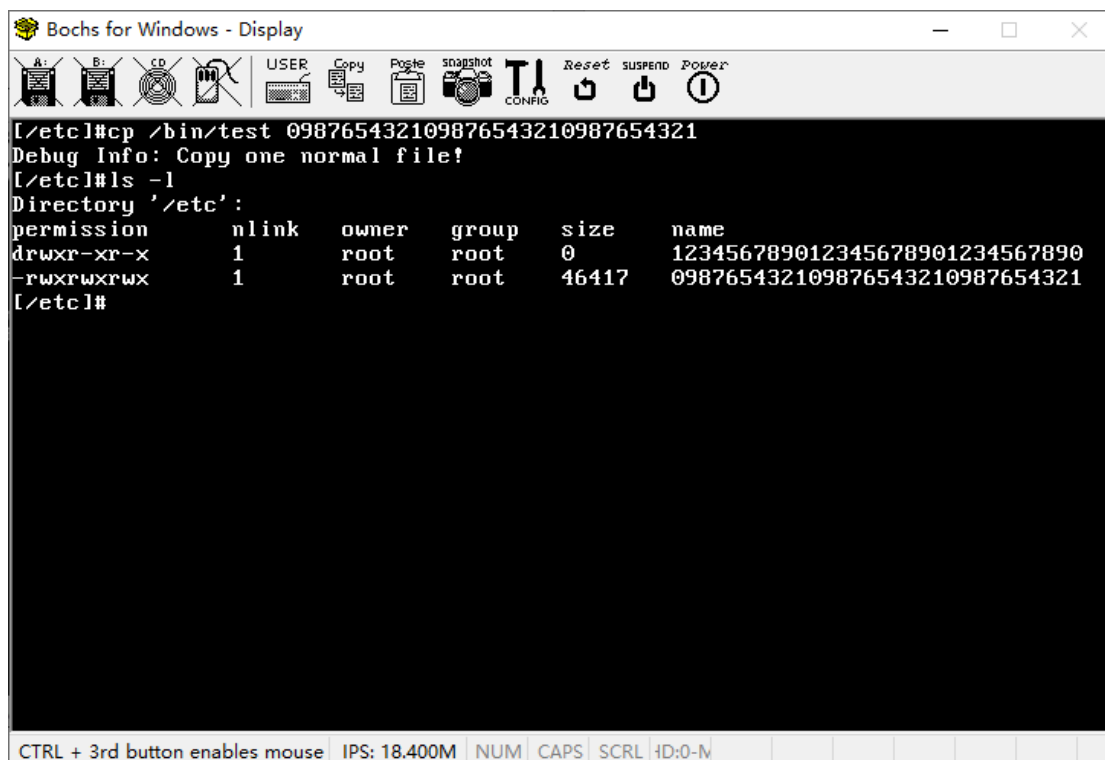
```

6. 实验结果

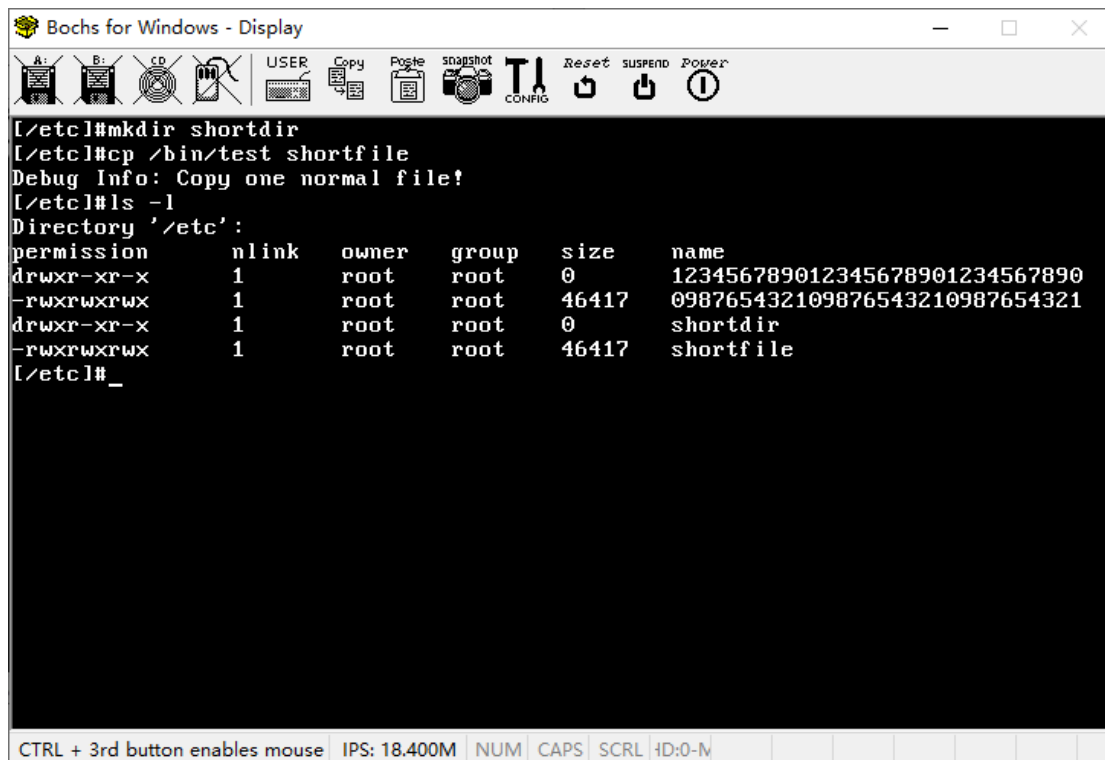
6.1. 添加长文件夹



6.2. 添加长文件



6.3. 添加短文件（夹）

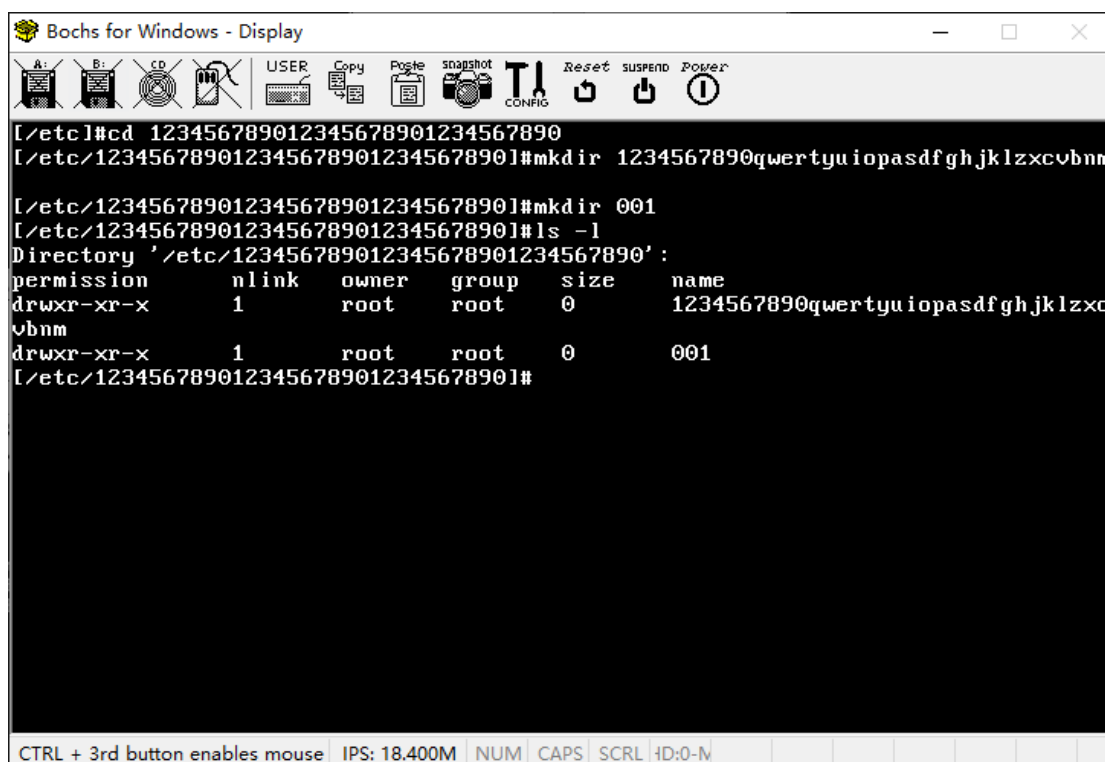


The image shows a Bochs for Windows - Display window. The terminal output is as follows:

```
[/etc]#mkdir shortdir
[/etc]#cp /bin/test shortfile
Debug Info: Copy one normal file!
[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
drwxr-xr-x  1      root   root    0      123456789012345678901234567890
-rwxrwxrwx  1      root   root   46417  098765432109876543210987654321
drwxr-xr-x  1      root   root    0      shortdir
-rwxrwxrwx  1      root   root   46417  shortfile
[/etc]#_
```

The status bar at the bottom shows: CTRL + 3rd button enables mouse | IPS: 18.400M | NUM | CAPS | SCRL | ID:0-N

6.4. 进入长文件夹，并添加子文件



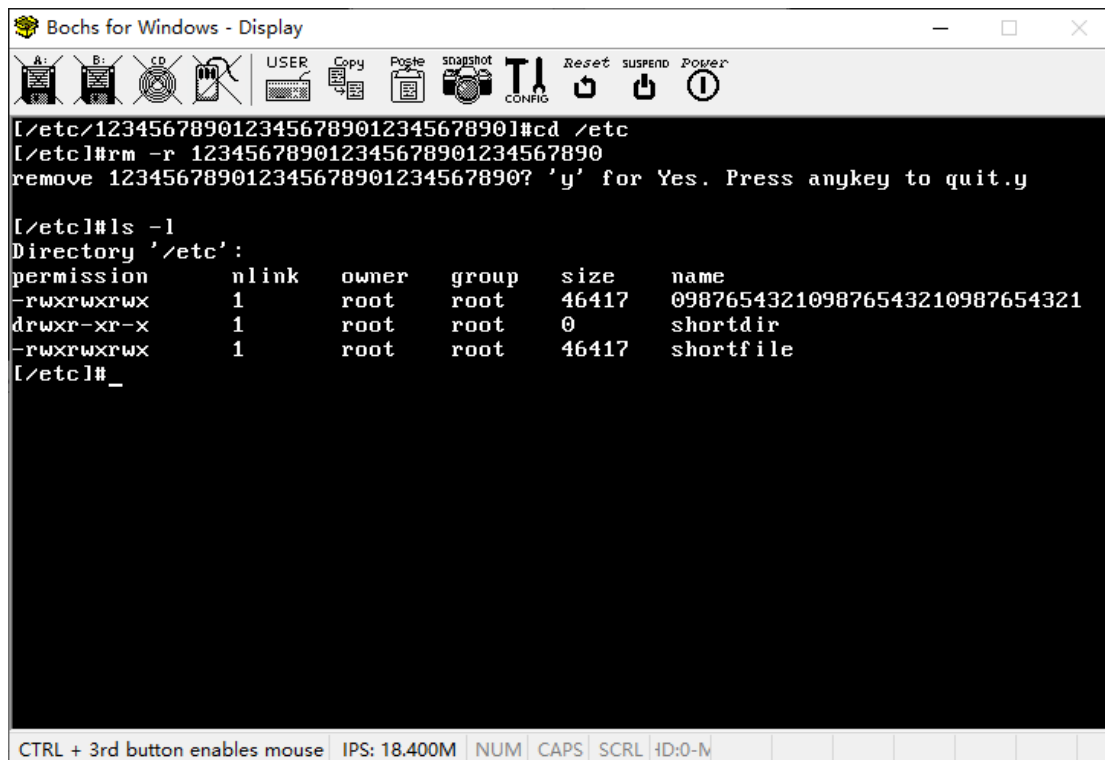
The image shows a Bochs for Windows - Display window. The terminal output is as follows:

```
[/etc]#cd 123456789012345678901234567890
[/etc/123456789012345678901234567890]#mkdir 1234567890qwertyuiopasdfghjklzxcvbnm

[/etc/123456789012345678901234567890]#mkdir 001
[/etc/123456789012345678901234567890]#ls -l
Directory '/etc/123456789012345678901234567890':
permission  nlink  owner  group  size  name
drwxr-xr-x  1      root   root    0      1234567890qwertyuiopasdfghjklzxcvbnm
drwxr-xr-x  1      root   root    0      001
[/etc/123456789012345678901234567890]#
```

The status bar at the bottom shows: CTRL + 3rd button enables mouse | IPS: 18.400M | NUM | CAPS | SCRL | ID:0-N

6.5. 递归删除长文件夹



Bochs for Windows - Display

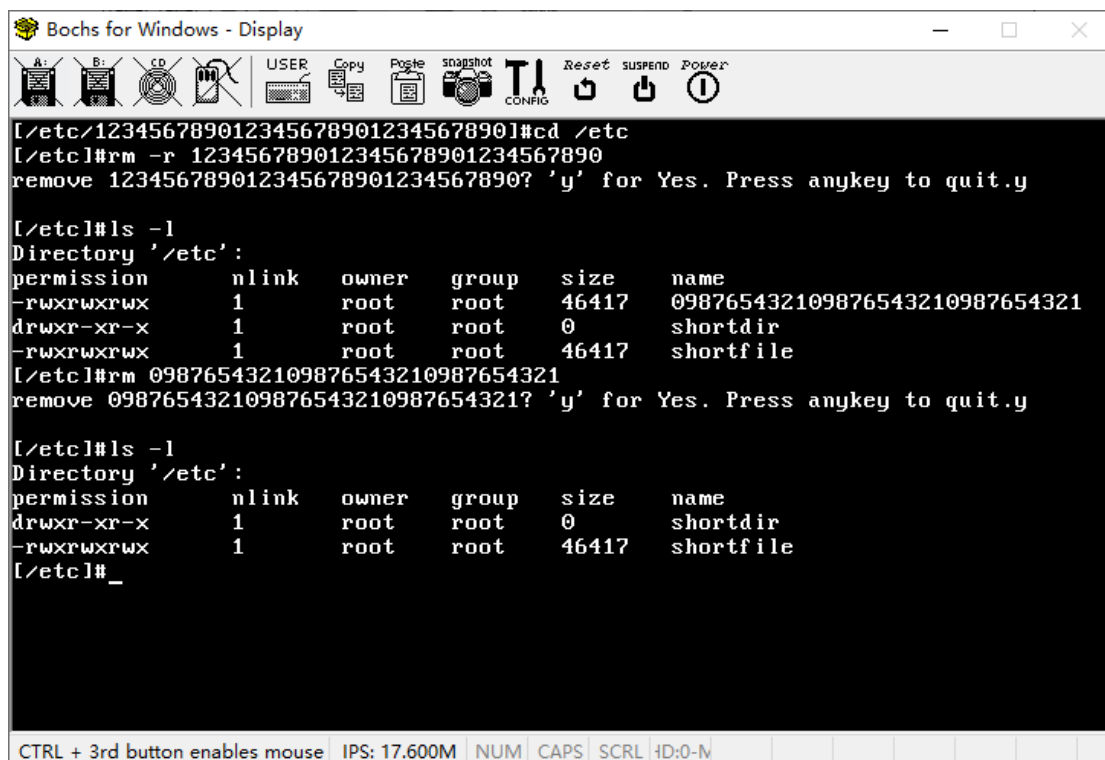
USER Copy Paste snapshot CONFIG Reset SUSPEND Power

```
[/etc/1234567890123456789012345678901#cd /etc
[/etc]#rm -r 123456789012345678901234567890
remove 123456789012345678901234567890? 'y' for Yes. Press anykey to quit.y

[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
-rwxrwxrwx   1   root   root   46417  098765432109876543210987654321
drwxr-xr-x   1   root   root    0     shortdir
-rwxrwxrwx   1   root   root   46417  shortfile
[/etc]#_
```

CTRL + 3rd button enables mouse IPS: 18.400M NUM CAPS SCRL ID:0-M

6.6. 删除长文件



Bochs for Windows - Display

USER Copy Paste snapshot CONFIG Reset SUSPEND Power

```
[/etc/1234567890123456789012345678901#cd /etc
[/etc]#rm -r 123456789012345678901234567890
remove 123456789012345678901234567890? 'y' for Yes. Press anykey to quit.y

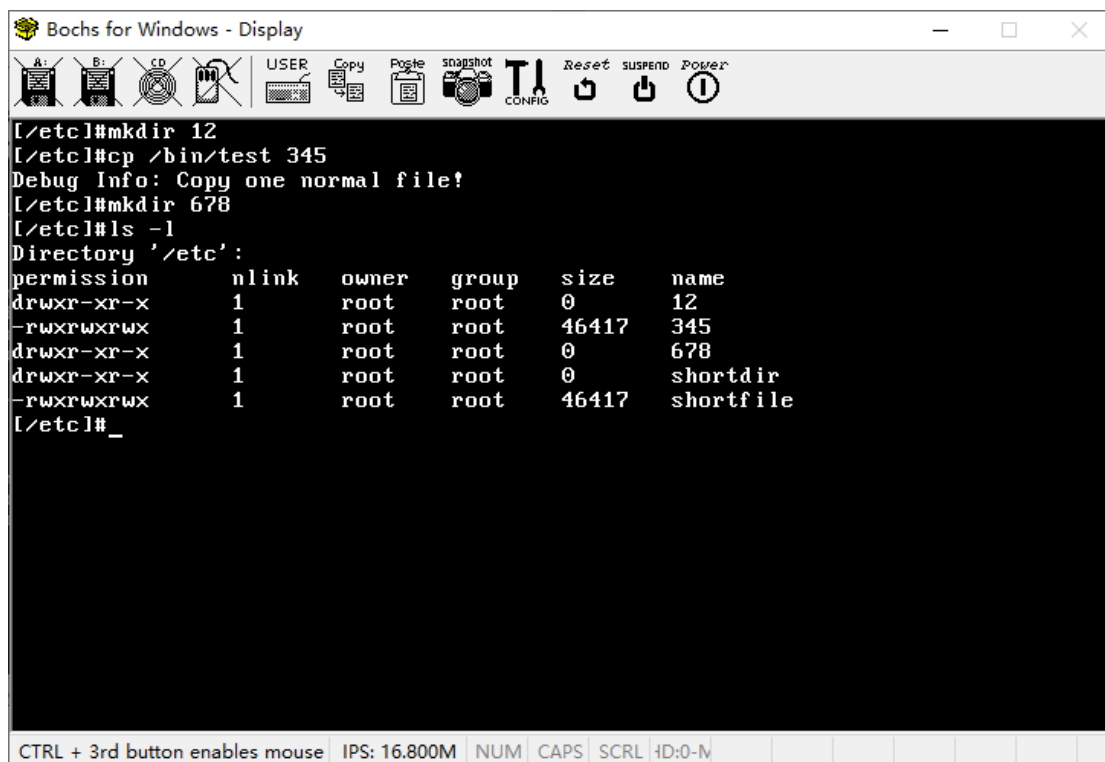
[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
-rwxrwxrwx   1   root   root   46417  098765432109876543210987654321
drwxr-xr-x   1   root   root    0     shortdir
-rwxrwxrwx   1   root   root   46417  shortfile
[/etc]#rm 098765432109876543210987654321
remove 098765432109876543210987654321? 'y' for Yes. Press anykey to quit.y

[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
drwxr-xr-x   1   root   root    0     shortdir
-rwxrwxrwx   1   root   root   46417  shortfile
[/etc]#_
```

CTRL + 3rd button enables mouse IPS: 17.600M NUM CAPS SCRL ID:0-M

6.7. 在添加短文件夹（填充原来长文件夹位置）

通过 ls 可以看出，新的短文件名目录项填充在原来目录项的位置（共 4*32，使用了 3*32）

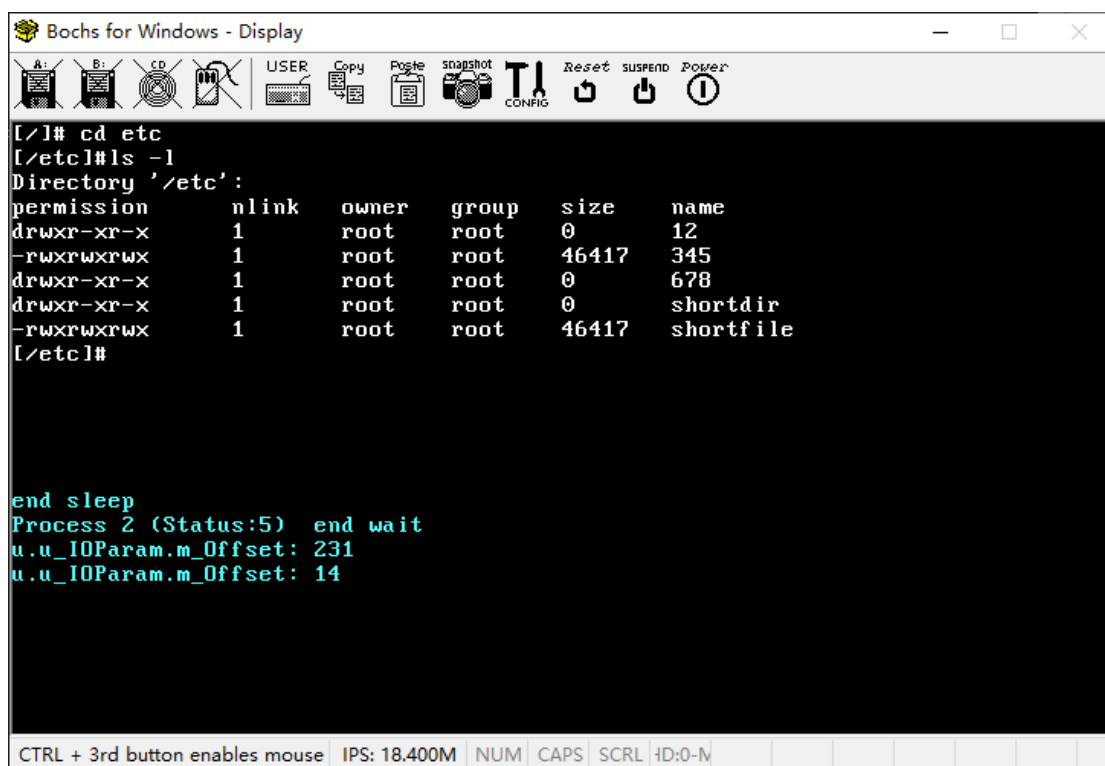


```
[/etc]#mkdir 12
[/etc]#cp /bin/test 345
Debug Info: Copy one normal file!
[/etc]#mkdir 678
[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
drwxr-xr-x   1    root   root    0      12
-rwxrwxrwx   1    root   root  46417  345
drwxr-xr-x   1    root   root    0      678
drwxr-xr-x   1    root   root    0    shortdir
-rwxrwxrwx   1    root   root  46417  shortfile
[/etc]#_
```

CTRL + 3rd button enables mouse IPS: 16.800M NUM CAPS SCRL ID:0-M

6.8. 再次进入后，磁盘保存

可以看出原先建立的目录项和文件都在。



```
[/]# cd etc
[/etc]#ls -l
Directory '/etc':
permission  nlink  owner  group  size  name
drwxr-xr-x   1    root   root    0      12
-rwxrwxrwx   1    root   root  46417  345
drwxr-xr-x   1    root   root    0      678
drwxr-xr-x   1    root   root    0    shortdir
-rwxrwxrwx   1    root   root  46417  shortfile
[/etc]#

end sleep
Process 2 (Status:5) end wait
u.u_IOParam.m_Offset: 231
u.u_IOParam.m_Offset: 14
```

CTRL + 3rd button enables mouse IPS: 18.400M NUM CAPS SCRL ID:0-M

7. 参考文献

- [1]田宇. 一个 64 位操作系统的设计与实现. 北京: 人民邮电出版社, 2018. 05.
- [2]王洪辉. 嵌入式系统 Linux 内核开发实战指南 ARM 平台. 北京: 电子工业出版社, 2009. 03.
- [3]Harver M. Deitel, Paul J. Deitel, David R. Choffnes. 操作系统 第 3 版. 北京: 清华大学出版社, 2007. 08.