

实验 22: Linux 特定文件系统设计探析

计科 1903 班 19281171 王雨潇

1. 实验目的

探索、分析、理解并掌握 Linux 内核关于特定文件系统的功能支撑设计原理、实现机制和编程要旨。

2. 实验环境

编译环境: GNU 编译器 GCC-1.4.0, 汇编器 as86, 链接器 ld86

运行环境: Bochs-2.2.1 虚拟机, Intel x86 计算机模拟器

测试环境: Bochs-2.2.1 虚拟机和附带的 bochsdbg 调试器

宿主机: 操作系统 Windows 10 家庭版, CPU Intel Core i7-10710 (x86-64 架构)

该实验报告使用 Plantuml 绘制全部流程图, 源码截图使用 VS Code

3. 实验内容

下载 Linux 内核源码, 摘取和研读文件系统相关的源程序。

围绕某一特定文件系统 (譬如 EXT、FAT 或 NTFS) 的功能支撑设计原理, 深入分析和理解文件操作相关系统调用、内核实现机制、对应文件系统标准规范暨数据结构描述等。

完成该 Linux 内核源码的编译、加载和启用, 并通过运行特定命令或程序测试验证自己的分析结果。

目录

实验 22: Linux 特定文件系统设计探析	1
1. 实验目的	1
2. 实验环境	1
3. 实验内容	1
4. 实验流程	3
4.1. MINIX 文件系统设计简析	3
4.2. Linux-0.11 文件系统实现机制	4
4.2.1. 文件系统低层设计	4
4.2.2. 高速缓冲区机制	6
4.2.3. 文件数据操作	9
4.2.4. 文件系统调用接口	10
4.3. Linux-0.11 内核编译启用	14
4.4. 验证分析结果	16
5. 实验结论和心得体会	17

4. 实验流程

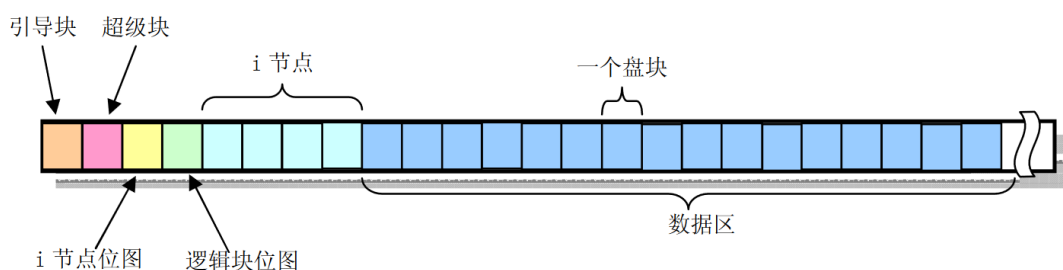
从官网下载并解压一份 devel 开发者版本（包含 include）的 Linux-0.11 内核源码，用于阅读；

Linux-0.11 内核的文件系统主要实现集中在 `\fs\` 目录下，可以分为 4 个模块：

模块	主要源码文件	功能
高速缓冲区管理程序	buffer.c	实现对硬盘等块设备进行数据高速存取
文件系统低层通用函数	bitmap.c	实现文件索引节点的管理、 磁盘数据块分配释放、 文件名与 i 节点转换算法
	inode.c	
	namei.c	
	super.c	
文件数据操作	block_dev.c	实现对各类文件的访问、读、写操作
	file_dev.c	
	char_dev.c	
	pipe.c	
	read_write.c	
文件系统调用接口	open.c	实现文件打开、关闭、创建 以及文件目录的操作
	exec.c	
	fcntl.c	
	ioctl.c	
	stat.c	

4.1. MINIX 文件系统设计简析

Linux-0.11 内核使用的文件系统为 1.0 版本的 MINIX 文件系统，与标准 UNIX 文件系统基本相同，是采用**两级索引分配**结构的文件系统，每个文件或目录映射到一个 i 节点，根据目录逐级访问文件。



MINIX 文件系统 1.0 版本支持的最大块设备容量（长度）是 64MB。

外存布局结构如上图所示，其中各部分作用如下：

- (1) **引导块** 是计算机加电启动时 ROM BIOS 自动读入的执行代码和数据；
- (2) **超级块** 负责存放盘设备上文件系统结构的信息，说明各部分的大小；
- (3) **i 节点位图** 每个 bit 代表一个 i 节点的使用情况；
- (4) **逻辑块位图** 每个 bit 代表盘上数据区中的一个数据盘块的使用情况；
- (5) **i 节点** 部分存放文件系统中每一个文件或目录的索引节点，包括对应文件的相关信息；
- (6) **数据区** 存放的是真正的文件数据；

4.2. Linux-0.11 文件系统实现机制

4.2.1. 文件系统低层设计

在 MINIX 文件系统中，一个文件名通过对应的 i 节点映射到磁盘块的相应数据。i 结点各字段信息的结构体定义在 `\include\linux\fs.h` 中，如下图所示：

```
93  struct m_inode {
94      unsigned short i_mode;
95      unsigned short i_uid;
96      unsigned long i_size;
97      unsigned long i_mtime;
98      unsigned char i_gid;
99      unsigned char i_nlinks;
100     unsigned short i_zone[9];
101     /* these are in memory also */
102     struct task_struct * i_wait;
```

`m_inode` 的 `i_zone[9]` 数组字段存放的就分别是文件的直接块号、一次间接块号、二次间接块号。而文件类型、权限信息分别保存在对应 i 节点的 `i_mode` 字段高低位。

6 种文件类型分别是：正规文件、目录名（Linux 内核中万物皆文件，目录也被认为是一种文件）、符号连接、命名管道、字符设备文件、块设备文件。

文件系统的底层设计即内核源码对 i 节点的直接存取操作功能部分，其中.\fs\bitmap.c 主要包括负责在释放和占用数据时修改 i 节点位图、和逻辑块位图使用状态的 4 个函数，如下图所示：

```
47 // 逻辑块位图释放
48 void free_block(int dev, int block)
49 > { ...
74 }
75
76 // 逻辑块位图占用
77 int new_block(int dev)
78 > { ...
107 }
108
109 // i 节点位图释放
110 void free_inode(struct m_inode * inode)
111 > { ...
137 }
138
139 // i 节点位图占用
140 struct m_inode * new_inode(int dev)
141 > { ...
172 }
```

.\fs\inode.c 主要包括对 i 结点进行分配、释放、取内容的关键函数，这些函数内部通过定义在同一文件内的 wait_on_inode()、write_inode()、read_inode()、sync_inodes() 等函数实现具体的底层操作，如下图所示：

```
72 // 根据 i 节点信息取文件数据块在设备上对应的逻辑块号
73 static int _bmap(struct m_inode * inode, int block, int create)
74 > { ...
139 }
...
151 // 释放内存 i 节点
152 void iput(struct m_inode * inode)
153 > { ...
177 > repeat: ...
194 }
246 // 分配 i 节点
247 struct m_inode * iget(int dev, int nr)
248 > { ...
295 }
```

.\fs\namei.c 的 struct m_inode * namei() 函数的输入参数是文件路径, 返回值是 i 节点, 其内部借助 inode.c 定义的 i 节点操作函数实现, 如下图所示:

```
struct m_inode * namei(const char * pathname)
{
    const char * basename;
    int inr, dev, namelen;
    struct m_inode * dir;
    struct buffer_head * bh;
    struct dir_entry * de;

    if (!(dir = dir_namei(pathname, &namelen, &basename)))
        return NULL;
    if (!namelen) /* special case: '/usr/' etc */
        return dir;
    bh = find_entry(&dir, basename, namelen, &de);
    if (!bh) {
        iput(dir);
        return NULL;
    }
    inr = de->inode;
    dev = dir->i_dev;
    brelse(bh);
    iput(dir);
    dir = iget(dev, inr);
    if (dir) {
        dir->i_atime = CURRENT_TIME;
        dir->i_dirt = 1;
    }
    return dir;
}
```

.\fs\super.c 主要包括用于处理文件系统超级块的函数 get_super()、put_super() 和 free_super(), 以及这些函数用到的底层中断、系统调用等功能。

4.2.2. 高速缓冲区机制

由于 I/O 操作开销较大, 为提升操作系统内核性能, Linux-0.11 使用高速数据缓冲区作为用户进程和外存之间的中间层, 高速缓冲区存放最近被使用的数据, 每个缓冲块和磁盘数据块大小相等 (均为 1024 字节), 使用户进程对文件系统的任何访问操作都会先访问缓冲区, 缓冲区未命中时再查找外存设备。

高速缓冲区的数据结构是：以 1024 字节（与外存的数据块大小相同）为最小单位，将整个缓冲区划分成小缓冲块，采用哈希表 `hash_table` 和空闲缓冲块链表 `free_list` 两种数据结构共同管理。这种组合方式的好处是，可以用哈希表实现 $O(1)$ 的缓冲头查询、用双向链表实现 $O(1)$ 插入和删除头尾结点。

`buffer_head` 结构体的定义在 `./include/linux/fs.h` 中，如下图所示：

```
struct buffer_head {
    char * b_data;           /* pointer to data block (1024 bytes) */
    unsigned long b_blocknr;  /* block number */
    unsigned short b_dev;     /* device (0 = free) */
    unsigned char b_uptodate;
    unsigned char b_dirt;     /* 0-clean,1-dirty */
    unsigned char b_count;    /* users using this block */
    unsigned char b_lock;     /* 0 - ok, 1 -locked */
    struct task_struct * b_wait;
    struct buffer_head * b_prev;
    struct buffer_head * b_next;
    struct buffer_head * b_prev_free;
    struct buffer_head * b_next_free;
};
```

实际使用的缓冲头队列和表变量、以及对高速缓冲区进行操作的代码，主要集中在 `./fs/buffer.c` 中。

```
extern int end;
struct buffer_head * start_buffer = (struct buffer_head *) &end;
struct buffer_head * hash_table[NR_HASH];
static struct buffer_head * free_list;
static struct task_struct * buffer_wait = NULL;
int NR_BUFFERS = 0;
```

`./fs/buffer.c` 提供了 3 个高速缓冲区读取函数，分别是块读取函数 `bread()`，块提前预读函数

`breada()`，页块读取函数 `bread_page()`，它们的共同功能都是调用对应的块读写函数，区别如下：

函数原型	行号	功能
<code>struct buffer_head * bread(int dev,int block)</code>	267	返回一个指定的缓存块
<code>void bread_page(unsigned long address, int dev,int b[4])</code>	296	一次读取 4 块缓冲块到内存指定地址。
<code>struct buffer_head * breada(int dev,int first, ...)</code>	322	返回指定设备读取指定的一些块

.\fs\buffer.c 还提供了缓冲区的同步方式。getblk()函数在获取缓冲块时，如果发现缓冲块 dirt 字段为 1（即缓冲块有数据未同步到磁盘），就会调用 sync_dev()函数将数据同步到硬盘。

如下图所示，sync_dev()函数内部是调用 ll_rw_block()将缓冲块数据写入磁盘的。

```
int sync_dev(int dev)
{
    int i;
    struct buffer_head * bh;

    bh = start_buffer;
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        if (bh->b_dev != dev)
            continue;
        wait_on_buffer(bh);
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    sync_inodes();
    bh = start_buffer;
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        if (bh->b_dev != dev)
            continue;
        wait_on_buffer(bh);
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    return 0;
}
```

.\fs\buffer.c 还包括缓冲区的置换方法，当调用 getblk()函数时，该函数会采用 LRU 算法，调用定义在同一文件下的 remove_from_queues()函数，从 hash 表中和空闲链表中移除要被清理的缓冲区。

```
static inline void remove_from_queues(struct buffer_head * bh)
{
    /* remove from hash-queue */
    if (bh->b_next)
        bh->b_next->b_prev = bh->b_prev;
    if (bh->b_prev)
        bh->b_prev->b_next = bh->b_next;
    if (hash(bh->b_dev,bh->b_blocknr) == bh)
        hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
    /* remove from free list */
    if (!(bh->b_prev_free) || !(bh->b_next_free))
        panic("Free block list corrupted");
    bh->b_prev_free->b_next_free = bh->b_next_free;
    bh->b_next_free->b_prev_free = bh->b_prev_free;
    if (free_list == bh)
        free_list = bh->b_next_free;
}
```


4.2.3. 文件数据操作

.\fs\目录下, block_dev.c、file_dev.c、char_dev.c、pipe.c 这 4 个文件分别是块设备、字符设备、管道设备和普通文件的外部调用接口。其中 block_dev.c、file_dev.c、pipe.c 的读写函数大同小异, 如下图所示:

```
14  int block_write(int dev, long * pos, char * buf, int count)
15 > { ...
45  }
46
47  int block_read(int dev, unsigned long * pos, char * buf, int count)
48 > { ...
73  }
17  int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 > { ...
46  }
47
48  int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
49 > { ...
56 > /* ...
90  }
13  // inode: 管道对应的 i 节点, buf: 数据缓冲区指针, count: 读取的字节数。
14  int read_pipe(struct m_inode * inode, char * buf, int count)
15 > { ...
40  }
41
42  int write_pipe(struct m_inode * inode, char * buf, int count)
43 > { ...
70  }
71
72  int sys_pipe(unsigned long * fildes)
73 > { ...
112 }
```

.\fs\read_write.c 文件是字符设备文件 (包括控制台终端 tty、串口终端 ttyx、内存字符设备) 的读写函数, 系统调用 read()和 write()的内部实现就是调用 char_dev.c 中的 rw_char()函数来操作。

```
int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
{
    crw_ptr call_addr;
    if (MAJOR(dev)>=NRDEVS)
        return -ENODEV;
    if (!(call_addr=crw_table[MAJOR(dev)]))
        return -ENODEV;
    return call_addr(rw,MINOR(dev),buf,count,pos);
}
```

4.2.4. 文件系统调用接口

这部分代码是文件系统调用的对接上层功能的实现。

.\fs\open.c 中名称以 “sys_” 开头的函数实现的都是文件的创建、打开和关闭，属性修改等系统调用。

```
43 > /* ...
47 int sys_access(const char * filename,int mode)
48 > { ...
73 }
74
75 int sys_chdir(const char * filename)
76 > { ...
88 }
89
90 int sys_chroot(const char * filename)
91 > { ...
103 }
104
105 int sys_chmod(const char * filename,int mode)
106 > { ...
119 }
```

.\fs\exec.c 中最关键的部分是 int do_execve(unsigned long * eip,long tmp,char * filename, char

** argv, char ** envp) 系统中断调用函数。第一步，它取得可执行程序的文件名对应的 i 节点号，如下图：

```
struct m_inode * inode;
struct buffer_head * bh;
struct exec ex;
unsigned long page[MAX_ARG_PAGES];
int i,argc,envc;
int e_uid, e_gid;
int retval;
int sh_bang = 0;
unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;

if ((0xffff & eip[1]) != 0x000f)
    panic("execve called from supervisor mode");
for (i=0 ; i<MAX_ARG_PAGES ; i++) /* clear page-table */
    page[i]=0;
if (!(inode=namei(filename))) /* get executables inode */
    return -ENOENT;
argc = count(argv);
envc = count(envp);
```

第二步，检查当前进程对该可执行文件有没有执行权限；

```
restart_interp:
    if (!S_ISREG(inode->i_mode)) { /* must be regular file */
        retval = -EACCES;
        goto exec_error2;
    }
    i = inode->i_mode;
    e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
    e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
    if (current->euid == inode->i_uid)
        i >>= 6;
    else if (current->egid == inode->i_gid)
        i >>= 3;
    if (!(i & 1) &&
        !((inode->i_mode & 0111) && suser())) {
        retval = -ENOEXEC;
        goto exec_error2;
    }
    if (!(bh = bread(inode->i_dev, inode->i_zone[0]))) {
        retval = -EACCES;
        goto exec_error2;
    }
}
```

第三步，处理可执行文件的执行头，.\fs\exec.c 的 225-318 行都是这部分的解析和出错处理，此处

不一一列举；

```
225     ex = *((struct exec *) bh->b_data); /* read exec-header */
226 >     if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) { ...
297     }
298     brelse(bh);
299 >     if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize || ...
304     }
305 >     if (N_TXTOFF(ex) != BLOCK_SIZE) { ...
309     }
310 >     if (!sh_bang) { ...
317     }
318     /* OK, This is the point of no return */
```

第四步，释放旧程序 i 节点，让进程的 executable 指向新程序 i 节点，为新执行程序准备进程堆栈，

函数结束；

```
/* OK, This is the point of no return */
if (current->executable)
    iput(current->executable);
current->executable = inode;
for (i=0 ; i<32 ; i++)
    current->sigaction[i].sa_handler = NULL;
for (i=0 ; i<NR_OPEN ; i++)
    if ((current->close_on_exec>>i)&1)
        sys_close(i);
current->close_on_exec = 0;
free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
if (last_task_used_math == current)
    last_task_used_math = NULL;
current->used_math = 0;
p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
p = (unsigned long) create_tables((char *)p,argc,envc);
current->brk = ex.a_bss +
    (current->end_data = ex.a_data +
    (current->end_code = ex.a_text));
current->start_stack = p & 0xfffff000;
current->euid = e_uid;
current->egid = e_gid;
i = ex.a_text+ex.a_data;
while (i&0xfff)
    put_fs_byte(0,(char *) (i++));
eip[0] = ex.a_entry;      /* eip, magic happens :- ) */
eip[3] = p;               /* stack pointer */
return 0;
```

.\fs\fcntl.c 实现了文件控制系统调用 `sys_fcntl()` 和两个文件句柄复制系统调用 `sys_dup()` 和 `sys_dup2()`，主要用在文件的标准输入/输出重定向和管道操作中；

```
int sys_dup2(unsigned int oldfd, unsigned int newfd)
{
    sys_close(newfd);
    return dupfd(oldfd, newfd);
}

int sys_dup(unsigned int fildes)
{
    return dupfd(fildes, 0);
}

int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
    struct file * filp;

    if (fd >= NR_OPEN || !(filp = current->filp[fd]))
        return -EBADF;
    switch (cmd) {
        case F_DUPFD:
            return dupfd(fd, arg);
        case F_GETFD:
            return (current->close_on_exec >> fd) & 1;
        case F_SETFD:
            if (arg & 1)
                current->close_on_exec |= (1 << fd);
    }
}
```

.\fs\ioctl.c 文件实现了输入/输出控制系统调用 `sys_ioctl()`，功能是对终端的 I/O 进行控制。

```
30 int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
31 {
32     struct file * filp;
33     int dev, mode;
34
35     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
36         return -EBADF;
37     mode = filp->f_inode->i_mode;
38     if (!S_ISCHR(mode) && !S_ISBLK(mode))
39         return -EINVAL;
40     dev = filp->f_inode->i_zone[0];
41     if (MAJOR(dev) >= NRDEVS)
42         return -ENODEV;
43     if (!ioctl_table[MAJOR(dev)])
44         return -ENOTTY;
45     return ioctl_table[MAJOR(dev)](dev, cmd, arg);
46 }
```

.\fs\stat.c 文件用于实现取文件状态信息的系统调用 sys_stat()和 sys_fstat(), 分别是利用文件名取信息和使用文件句柄(描述符)来取信息, 如下图所示:

```
int sys_stat(char * filename, struct stat * statbuf)
{
    struct m_inode * inode;

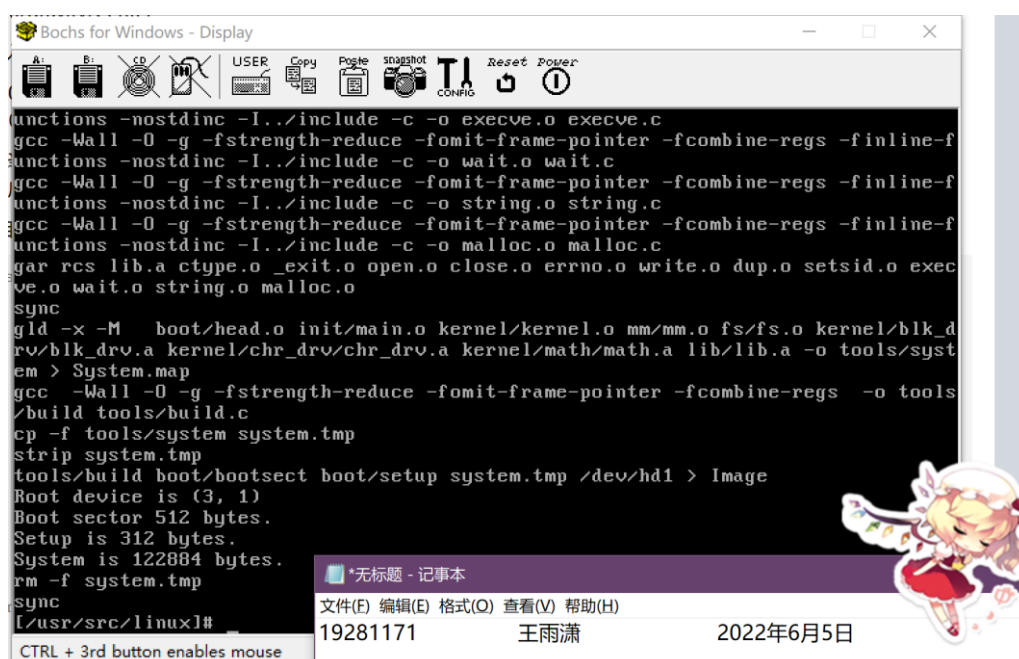
    if (!(inode=namei(filename)))
        return -ENOENT;
    cp_stat(inode,statbuf);
    iput(inode);
    return 0;
}

int sys_fstat(unsigned int fd, struct stat * statbuf)
{
    struct file * f;
    struct m_inode * inode;

    if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
        return -EBADF;
    cp_stat(inode,statbuf);
    return 0;
}
```

4.3. Linux-0.11 内核编译启用

首先启动 bochsrc-hd.bxrc, 到./usr/src/linux 目录下编译 Linux-0.11 内核源码



```
Bochs for Windows - Display
functions -nostdinc -I../include -c -o execve.o execve.c
gcc -Wall -O -g -fstrength-reduce -fomit-frame-pointer -fcombine-regs -finline-f
unctions -nostdinc -I../include -c -o wait.o wait.c
gcc -Wall -O -g -fstrength-reduce -fomit-frame-pointer -fcombine-regs -finline-f
unctions -nostdinc -I../include -c -o string.o string.c
gcc -Wall -O -g -fstrength-reduce -fomit-frame-pointer -fcombine-regs -finline-f
unctions -nostdinc -I../include -c -o malloc.o malloc.c
gar rcs lib.a ctype.o _exit.o open.o close.o errno.o write.o dup.o setsid.o exec
ve.o wait.o string.o malloc.o
sync
gld -x -M boot/head.o init/main.o kernel/kernel.o mm/mm.o fs/fs.o kernel/blk_d
rv/blk_drv.a kernel/chr_drv/chr_drv.a kernel/math/math.a lib/lib.a -o tools/syst
em > System.map
gcc -Wall -O -g -fstrength-reduce -fomit-frame-pointer -fcombine-regs -o tools
/build tools/build.c
cp -f tools/system system.tmp
strip system.tmp
tools/build boot/bootsect boot/setup system.tmp /dev/hd1 > Image
Root device is (3, 1)
Boot sector 512 bytes.
Setup is 312 bytes.
System is 122884 bytes.
rm -f system.tmp
sync
[usr/src/linux]#
```

备份一份 bootimage-0.11-hd 命名为 bootimage-0.11-hd-copy, 备用;

bootimage-0.11	2004/8/17 14:48	11 文件
rootimage-0.11	2005/6/18 16:49	11 文件
bootimage-0.11-fd	2004/3/4 21:56	11-FD 文件
bootimage-0.11-hd	2004/4/29 23:22	11-HD 文件
bootimage-0.11-hd-copy	2004/4/29 23:22	11-HD-COPY 文件
bootimage-0.12-fd	2002/8/29 6:50	12-FD 文件
bootimage-0.12-hd	2004/2/23 21:20	12-HD 文件
VGABIOS-elpin-2.40	2004/9/3 1:15	40 文件
bochsrc-fda.bxrc		
bochsrc-fdb.bxrc		
bochsrc-hd.bxrc		
bochsrc-hdboot.bxrc		



*无标题 - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)


19281171 | 王雨潇 | 2022年6月5日

输入命令 `dd bs=8192 if=Image of=/dev/fd0`, 把新的引导启动文件输出到 bootimage-0.11-hd;

```

cp -f tools/system system.tmp
strip system.tmp
tools/build boot/bootsect boot/setup system.tmp /dev/hd1 > Image
Root device is (3, 1)
Boot sector 512 bytes.
Setup is 312 bytes.
System is 122884 bytes.
rm -f system.tmp
sync
[/usr/src/linux]# dd bs=8192 if=Image of=/dev/fd0
15+1 records in
15+1 records out
[/usr/src/linux]#

```




*无标题 - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

19281171 | 王雨潇

按 Reset 重启模拟器, 截图如下所示:

Bochs for Windows - Display



```

Plex86/Bochs VGABios 0.5c 07 Jul 2005
This VGA/VE Bios is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.138.2.1 $ $Date: 2005/07/06 19:30:36 $


ata0 master: Generic 1234 ATA-2 Hard-Disk (121 MBytes)

Booting from Floppy...

Loading system ...

HD-controller reset failed: 00
Partition table ok.
39064/62000 free blocks
19520/20666 free inodes
3462 buffers = 3545088 bytes buff
Free mem: 12582912 bytes
Ok.
[/usr/root]#

```



*无标题 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

19281171 | 王雨潇 | 2022年6月5日

4.4. 验证分析结果

分别运行 ls, cd, mkdir 等文件目录管理命令，截图如下所示：

```

Bochs x86 Emulator 2.6.0
Build from CVS snapshot on 2022-06-05

000000000000i[ ] reading configuration file
000000000000i[ ] WARNING: syntax has changed
000000000000e[ ] D:\Program Files (x86)\Bochs\
000000000000i[ ] installing win32 module
000000000000i[ ] using log file bochsout.

[ ]# find . -name "*.c"
find: command not found
[ ]# ls
Image      dev      image     shoelace  usr
bin        etc      mnt       tmp       var
[ ]# cd mnt
[ ]# ls
Image      bin      dev      etc      image
[ ]# cd bin
[ ]# ls
agetty      chsh      faillog   groupdel  mkfs      passwd    sulogin   usermod
awk         clock     false     groupmod  mknod     ps        sync      vi
bash        crond     fdisk     id        mkpasswd  rm        true      zdump
bash0       date      fsck      init      mkswap    sh        umount    zic
chage       dpasswd   gawk      kdb       mount     sh0       update
chfn        echo      gpasswd   login     newgrp    sleep     useradd
chpasswd    env       groupadd  logoutd   newusers  su        userdel

[ ]#
  
```

```

bc: command not found
[ ]# cd bin
[ ]# pwd
/mnt/bin
[ ]# mkdir wyx
[ ]# ls
agetty      chsh      faillog   groupdel  mkfs      passwd    sulogin   usermod
awk         clock     false     groupmod  mknod     ps        sync      vi
bash        crond     fdisk     id        mkpasswd  rm        true      zdump
bash0       date      fsck      init      mkswap    sh        umount    zic
chage       dpasswd   gawk      kdb       mount     sh0       update
chfn        echo      gpasswd   login     newgrp    sleep     useradd
chpasswd    env       groupadd  logoutd   newusers  su        userdel

[ ]#
  
```

再运行 chmod 777 命令修改文件权限，运行 ls -al 查看文件权限信息，可以发现文件夹“wyx”所有权限已被启用；

```

[ ]# cd wyx
[ ]# ls
[ ]# cd ..
[ ]# chmod 777 wyx
[ ]# ls -al wyx
total 2
drwxrwxrwx  2 root    root          32 Jun  5 22:52 .
drwxr-xr-x  3 root    root          896 Jun  5 22:52 ..

[ ]#
  
```


5. 实验结论和心得体会

在本次实验中，我通过阅读 Linux-0.11 的内核源码，了解了最初版本的 Linux 的文件系统设计实现机制，也巩固了课上所学的理论知识。阅读源码可以看出，Linux-0.11 作为最简操作系统，其内核的文件系统有着非常巧妙的层次结构。

作为用户程序和文件、目录、磁盘之间的抽象层，对于上层结构，Linux-0.11 的文件系统为用户封装了只需文件路径就可以操作 i 结点的接口，方便用户使用；对于底层结构，文件系统又分别实现了操作块设备（磁盘）、字符设备、管道、普通文件数据的方法，完成了文件管理的基本目标。我们应当熟悉 Linux-0.11 内核中的文件管理实现，为以后编写出更好的程序打下基础。