

Operating System Lab IV

Teacher: Shuyu Shi

Yikai Zhang, 171840708

2020-1-10

张逸凯, 171840708

查看当前系统所使用的文件系统的类型及版本号

阅读 `Ext3` (或 `Ext4`) 文件系统, 特别是索引节点相关的源代码

原理层面的理解

接下来阅读源码

为内核添加一个新的系统调用 `filesys`, 其从调用者接收一个磁盘文件的全局路径名, 打印该文件占用的所有磁盘块

实现思路

具体实现过程

实验环境:

实验过程

设计并添加系统调用

编译内核

编写用户态测试程序

最后结果如下:

遇到的问题

总结 [🔗](#)

查看当前系统所使用的文件系统的类型及版本号

`parted`命令是由GNU组织开发的一款功能强大的磁盘分区和分区大小调整工具, 它可以处理最常见的分区格式, 包括: `ext2`、`ext3`、`fat16`、`fat32`、`NTFS`、`ReiserFS`、`JFS`、`XFS`、`UFS`、`HFS`以及Linux交换分区。

```
1 | parted
2 | p
3 | print list
```

```
zykoslab@zykoslab-virtual-machine:~$ parted
WARNING: You are not superuser.  Watch out for permissions.
Warning: Unable to open /dev/sr0 read-write (Read-only file system).  /dev/sr0 has been opened read-only.
GNU Parted 3.2
Using /dev/sr0
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) p
Model: NECVMWare VMWare SATA CD01 (scsi)
Disk /dev/sr0: 2083MB
Sector size (logical/physical): 2048B/2048B
Partition Table: mac
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
  1      2048B   6143B   4096B                Apple
  2      2042MB  2045MB  2523kB                EFI
```

或者:

```
1 | df -Th | grep "^/dev"
```

```
zykoslab@zykoslab-virtual-machine:~$ df -Th | grep "^/dev"
/dev/sda1      ext4          30G      24G      4.1G    86% /
/dev/loop0     squashfs      4.2M     4.2M      0    100% /snap/gnome-calculator/406
/dev/loop2     squashfs      3.8M     3.8M      0    100% /snap/gnome-system-monitor/111
/dev/loop1     squashfs      43M      43M      0    100% /snap/gtk-common-themes/1313
/dev/loop3     squashfs      45M      45M      0    100% /snap/gtk-common-themes/1353
/dev/loop4     squashfs      90M      90M      0    100% /snap/core/8213
/dev/loop6     squashfs      4.3M     4.3M      0    100% /snap/gnome-calculator/544
/dev/loop7     squashfs      15M      15M      0    100% /snap/gnome-characters/367
/dev/loop9     squashfs      55M      55M      0    100% /snap/core18/1279
/dev/loop11    squashfs      1.0M     1.0M      0    100% /snap/gnome-logs/81
/dev/loop12    squashfs     157M     157M      0    100% /snap/gnome-3-28-1804/110
/dev/loop13    squashfs     150M     150M      0    100% /snap/gnome-3-28-1804/67
/dev/loop14    squashfs      1.0M     1.0M      0    100% /snap/gnome-logs/61
/dev/loop16    squashfs      55M      55M      0    100% /snap/core18/1288
/dev/sr0       iso9660       2.0G     2.0G      0    100% /media/zykoslab/Ubuntu 18.04.3 LTS amd64
/dev/loop15    squashfs      90M      90M      0    100% /snap/core/8268
/dev/loop8     squashfs      3.8M     3.8M      0    100% /snap/gnome-system-monitor/123
/dev/loop17    squashfs      15M      15M      0    100% /snap/gnome-characters/375
```

使用 `h` 让输出的容量以最合适的单位呈现。

```
zykoslab@zykoslab-virtual-machine:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G   0    3.9G   0% /dev
tmpfs           796M  2.1M  794M   1% /run
/dev/sda1        30G   26G   2.1G  93% /
tmpfs           3.9G   0    3.9G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           3.9G   0    3.9G   0% /sys/fs/cgroup
/dev/loop0       1.0M   1.0M      0 100% /snap/gnome-logs/61
/dev/loop1       3.8M   3.8M      0 100% /snap/gnome-system-monitor/123
/dev/loop2       4.3M   4.3M      0 100% /snap/gnome-calculator/544
/dev/loop3       43M   43M      0 100% /snap/gtk-common-themes/1313
/dev/loop4       55M   55M      0 100% /snap/core18/1288
/dev/loop6      150M  150M      0 100% /snap/gnome-3-28-1804/67
/dev/loop5       90M   90M      0 100% /snap/core/8213
/dev/loop7       90M   90M      0 100% /snap/core/8268
/dev/loop8      157M  157M      0 100% /snap/gnome-3-28-1804/110
/dev/loop9       55M   55M      0 100% /snap/core18/1279
/dev/loop10      15M   15M      0 100% /snap/gnome-characters/375
/dev/loop11      1.0M   1.0M      0 100% /snap/gnome-logs/81
/dev/loop12      45M   45M      0 100% /snap/gtk-common-themes/1353
/dev/loop13      4.2M   4.2M      0 100% /snap/gnome-calculator/406
/dev/loop14      15M   15M      0 100% /snap/gnome-characters/367
/dev/loop15      3.8M   3.8M      0 100% /snap/gnome-system-monitor/111
tmpfs           796M  12K   796M   1% /run/user/121
vmhgfs-fuse     204G  184G   20G   91% /mnt/hgfs
tmpfs           796M  28K   796M   1% /run/user/1000
/dev/sr0         2.0G   2.0G      0 100% /media/zykoslab/Ubuntu 18.04.3 LTS amd64
```

使用 `df` 命令: `df` command reports file system disk space usage, to include the file system type on a particular disk partition, use the `-T` flag.

我们可以看到第二列Type表示文件系统类型:

上面显示的文件系统包含了:

squashfs is a [compressed](#) read-only [file system](#) for [Linux](#).

tmpfs is a temporary file storage paradigm implemented in many Unix-like operating systems.

iso9660 is a file system for optical disc media.

The ext4 journaling file system or fourth extended filesystem is a journaling file system for Linux, developed as the successor to ext3.

综上, 可以发现我的文件系统是 EXT4 格式的, 还有其他的一些特殊情况的不同格式.

此外还有一些指令也是和文件系统相关的:

`fsck` 用于检查和可选地修复Linux文件系统, 它还可以在指定的磁盘分区上打印文件系统类型:

```
zykoslab@zykoslab-virtual-machine:~$ fsck -N /dev/sda3
fsck from util-linux 2.31.1
[/sbin/fsck.ext2 (1) -- /dev/sda3] fsck.ext2 /dev/sda3
zykoslab@zykoslab-virtual-machine:~$ fsck -N /dev/sdb1
fsck from util-linux 2.31.1
[/sbin/fsck.ext2 (1) -- /dev/sdb1] fsck.ext2 /dev/sdb1
zykoslab@zykoslab-virtual-machine:~$
```

或者用 `lsblk` 显示块设备, 当与 `-f` 选项一起使用时, 它还在分区上打印文件系统类型:

```
[/sbin/fsck.ext2 (1) -- /dev/sdb1] fsck.ext2 /dev/sdb1
zykoslab@zykoslab-virtual-machine:~$ lsblk -f
NAME        FSTYPE LABEL        UUID                                MOUNTPOINT
loop0       squashfs
loop1       squashfs
loop2       squashfs
loop3       squashfs
loop4       squashfs
loop6       squashfs
loop7       squashfs
loop8       squashfs
loop9       squashfs
loop11      squashfs
loop12      squashfs
loop13      squashfs
loop14      squashfs
loop15      squashfs
loop16      squashfs
loop17      squashfs
sda
└─sda1      ext4          a7ac3bb1-4c1e-4162-8086-7f1da78bd4ab /
sr0         iso9660      Ubuntu 18.04.3 LTS amd64 2019-08-05-19-29-01-00 /media/zykoslab/Ubuntu 18.04.3 LTS amd64
```

或者用 `blkid` 命令查找或打印块设备属性, 只需指定磁盘分区作为参数, 如下所示:

或者在 `/etc/fstab` 下有相关的静态文件信息比如挂载点, 文件系统类型, 挂载选项等.

或者用 `mount` 命令查看在Linux中挂载文件系统, 注意这里是有远程挂载文件系统时才可使用.

阅读 Ext3 (或 Ext4) 文件系统, 特别是索引节点相关的源代码

原理层面的理解

首先先过一遍存储管理大致相关的知识, 也就是文件系统中索引结点为什么要存在, 文件储存在硬盘上, 硬盘的最小存储单位是扇区, 操作系统读取硬盘的时候**不一个个扇区地读取**, 而是一次性连续读取多个扇区, 读取一个块. 所以块是文件存取的最小单位, 一般来说块大小都是 `4KB`.

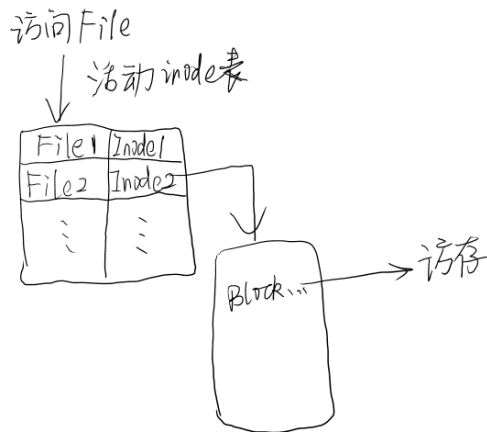
文件数据都储存在块中, 但是我们还必须找到一个地方储存文件的相关信息, 比如文件的创建者、文件的创建日期、文件的大小等等. `inode` **也就是索引结点就是存储这些东西的结构**. 每一个文件都有对应的 `inode`. 在Linux中可以通过inode实现文件的查找定位. 我曾想过把inode看出一个指针, 目前看来并没有什么问题.

每个inode都有一个号码, 操作系统用inode号码来识别不同的文件. 对于系统来说, 文件名只是inode号码便于识别的别称或者绰号. 表面上用户通过文件名打开文件, 实际上, 操作系统首先找到这个文件名对应的inode号码; 其次通过inode号码, 获取inode信息; 最后根据inode信息, 找到文件数据所在的block读出数据.

inode的内容大致有: (这里只是简要地说一下, 下面阅读inode源码有注释具体有什么):

- 文件的字节数
- 文件拥有者的User ID
- 文件的Group ID
- 文件的读、写、执行权限

- 文件的时间戳: ctime指inode上一次变动的时间, mtime指文件内容上一次变动的时间, atime指文件上一次打开的时间.
- 硬链接数, 即有多少文件名指向这个inode, 就是文件共享方式的一种.
- 文件数据block的位置



大致的访存过程如上, 找到inode后根据inode去找磁盘块访存. 所以可以总结出inode的一些性质: 对于有些无法删除的文件可以通过删除inode节点来删除, 移动或者重命名文件, 只是改变了第一层表的映射, 并不需要实际对硬盘操作, 在打开一个文件后, 只需要通过inode来识别文件.

这里我自己以一个文件为例, 查看它的inode信息:

```
zykoslab@zykoslab-virtual-machine:~$ stat maps.txt
  File: maps.txt
  Size: 22118          Blocks: 48          IO Block: 4096   regular file
Device: 801h/2049d    Inode: 409423       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/zykoslab)   Gid: ( 1000/zykoslab)
Access: 2019-12-07 17:09:28.615567692 +0800
Modify: 2019-12-07 17:05:29.316128765 +0800
Change: 2019-12-07 17:09:24.471561419 +0800
 Birth: -
```

blocks: 表示占用的块数目

Access: 表示最后一次访问的时间

Modify: 表示最后一次修改的时间

Change: 表示最后文件状态更改的时间

我琢磨了一下, 其实实践一下inode可以更好地理解它:

```
1 | ln file.txt file.hardlink
2 | ln -s file.txt file.softlink
```

```
/dev/sr0      2.0G  2.0G    0 100% /media/zykoslab/Ubuntu 18.04.3 LTS amd64
zykoslab@zykoslab-virtual-machine:~$ ln maps.txt maps.hardlink
zykoslab@zykoslab-virtual-machine:~$ ln -s maps.txt maps.softlink
zykoslab@zykoslab-virtual-machine:~$
```

注意下面刚刚建立的link:

```
zykoslab@zykoslab-virtual-machine:~$ ls -ll
total 689720
409429 -rw-r--r-- 1 zykoslab zykoslab 2776 1月 10 21:31 data.txt
393435 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 10:05 Desktop
393439 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 10:05 Documents
393436 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 15:03 Downloads
405369 -rw-r--r-- 1 zykoslab zykoslab 665 1月 10 20:59 kk.tbl
409612 drwxr-xr-x 24 zykoslab zykoslab 4096 11月 29 16:30 linux-4.4.205
396886 -rw-r-xr-x 1 zykoslab zykoslab 650383360 12月 5 15:14 linux-4.4.205.tar
464740 -rw-r--r-- 2 zykoslab zykoslab 2296 1月 10 21:33 maps.hardlink
409427 lrwxrwxrwx 1 zykoslab zykoslab 8 1月 10 23:15 maps.softlink -> maps.txt
464740 -rw-r--r-- 2 zykoslab zykoslab 2296 1月 10 21:33 maps.txt
393448 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 10:05 Music
393456 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 7 16:06 Pictures
393438 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 10:05 Public
409414 -rw-r-xr-x 1 zykoslab zykoslab 26464 12月 7 17:10 t
409428 -rw-r--r-- 1 zykoslab zykoslab 634 12月 7 17:10 t.cpp
393437 drwxr-xr-x 2 zykoslab zykoslab 4096 12月 5 10:05 Templates
405351 -rw-r-xr-x 1 zykoslab zykoslab 8776 12月 7 16:41 test
403050 -rw-r--r-- 1 zykoslab zykoslab 2086 12月 5 16:22 test.c
```

可以看到, 创建一个硬链接并没有创建新的inode, 只不过是在maps.txt上加上了一个新的dictionary的对应关系. 当我修改maps.txt的时候, 也就是修改了inode是他自己的那块磁盘, 而硬链接读一样的磁盘, 这样就不难理解为什么硬链接会同步更新.

那么添加一个硬链接, 显示出来的那个maps.hardlink是从哪来的呢? 添加一个硬链接, 会在目录文件里, 添加一条信息, 文件名-inode号, 所以就是相当于新添加了一个dictionary的对应关系.

而软连接只是创建了一个新的文件, 这个新的文件内容是指向maps.txt, 也就是说, 每次操作软链接, 实际上都是操作maps.txt, 所以软连接不能脱离源文件而存在, 因为如果源文件被删除了那么这个软链接就找不到自己应该指向的那个对象了.

好嘞, 现在感觉从理论层面原理层面已经大致理解了inode了.

接下来阅读源码

在 `/include/linux/fs.h` 中可以发现 inode 数据结构:

```
/*
 * Keep mostly read-only and often accessed (especially for
 * the RCU path lookup and 'stat' data) fields at the beginning
 * of the 'struct inode'
 */
struct inode {
    umode_t          i_mode;
    unsigned short   i_opflags;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int      i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl  *i_acl;
    struct posix_acl  *i_default_acl;
#endif
};
```

同实验三一样是通过[Bootlin网站](#)找到的:

```
/ include / linux / fs.h

1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef _LINUX_FS_H
3  #define _LINUX_FS_H
4
5  #include <linux/linkage.h>
6  #include <linux/wait_bit.h>
7  #include <linux/kdev_t.h>
8  #include <linux/dcache.h>
9  #include <linux/path.h>
10 #include <linux/stat.h>
11 #include <linux/cache.h>
12 #include <linux/list.h>
```

不妨从inode节点结构开始读起.

```
1  /*
2   * Keep mostly read-only and often accessed (especially for
3   * the RCU path lookup and 'stat' data) fields at the beginning
4   * of the 'struct inode'
5   */
6  struct inode {
7      umode_t      i_mode;      // 控制访问模式权限位.
8      unsigned short i_opflags;
9      kuid_t      i_uid;      // 记录被谁使用.
10     kgid_t      i_gid;      // 使用人id的组.
11     unsigned int  i_flags;      // 文件系统标志.
12
13     #ifdef CONFIG_FS_POSIX_ACL
14         struct posix_acl *i_acl;
15         struct posix_acl *i_default_acl;
16     #endif
17
18     const struct inode_operations *i_op; // 索引节点操作表.
19     struct super_block *i_sb;             // super block 超级块.
20     struct address_space *i_mapping;      // 对应的地址映射.
21
22     #ifdef CONFIG_SECURITY
23         void *i_security; // 如果宏CONFIG_SECURITY开启 => Enable
24         different security models, 设置不同的安全模式.
25     #endif
26
27     /* Stat data, not accessed from path walking */
28     unsigned long i_ino; // 当前inode节点号.
29     /*
30      * Filesystems may only read i_nlink directly. They shall use the
31      * following functions for modification:
32      *
33      * (set|clear|inc|drop)_nlink
34      * inode_(inc|dec)_link_count
35      */
36     union {
37         const unsigned int i_nlink; // 注意, 这里就是硬链接的数目. 就是PPT
38         unsigned int __i_nlink;
39     };
40 }
```



```

38     };
39     dev_t          i_rdev;        // 实设备标识符.
40     loff_t         i_size;        // 文件大小(Byte)
41     struct timespec i_atime;       // 记录最后访问时间.
42     struct timespec i_mtime;       // 记录最后modify的时间.
43     struct timespec i_ctime;       // 记录最后change的时间.
44     spinlock_t      i_lock; /* i_blocks, i_bytes, maybe i_size */ // 自旋
    锁
45     unsigned short   i_bytes;      // 当前使用了多少bytes.
46     u8               i_blkbits;    // 块大小 (bit).
47     u8               i_write_hint;
48     blkcnt_t         i_blocks;     // 文件的块数.
49
50 #ifdef __NEED_I_SIZE_ORDERED
51     seqcount_t       i_size_seqcount;
52 #endif
53
54     /* Misc */
55     unsigned long     i_state;      // 状态标志位.
56     struct rw_semaphore i_rwsem;
57
58     unsigned long     dirtied_when; /* jiffies of first dirtying */
59     unsigned long     dirtied_time_when; // 首次修改时间.
60
61     struct hlist_node i_hash;       // 哈希表首端.
62     struct list_head  i_io_list;    /* backing dev IO list */
63 #ifdef CONFIG_CGROUP_WRITEBACK
64     struct bdi_writeback *i_wb;     /* the associated cgroup wb */
65
66     /* foreign inode detection, see wbc_detach_inode() */
67     int               i_wb_frn_winner;
68     u16               i_wb_frn_avg_time;
69     u16               i_wb_frn_history;
70 #endif
71     struct list_head  i_lru;        /* inode LRU list */
72     struct list_head  i_sb_list;
73     struct list_head  i_wb_list;    /* backing dev writeback list */
74     union {
75         struct hlist_head i_dentry;
76         struct rcu_head   i_rcu;
77     };
78     atomic64_t        i_version;
79     atomic_t          i_count;      // 修改计数器, 这里的修改貌似是在特定时候.
80     atomic_t          i_dio_count;  // 限定时间的计数信息.
81     atomic_t          i_writecount; // 写动作计数器.
82 #if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
83     atomic_t          i_readcount; /* struct files open RO */
84 #endif
85     union {
86         const struct file_operations *i_fop; /* former ->i_op-
    >default_file_ops */
87         void (*free_inode)(struct inode *);
88     };
89     struct file_lock_context *i_flctx;
90     struct address_space i_data;
91     struct list_head    i_devices;
92     union {
93         struct pipe_inode_info *i_pipe; // 管道结点.

```



```

94     struct block_device *i_bdev;           // 与硬件设备相关的信息。
95     struct cdev         *i_cdev;
96     char                 *i_link;          // 貌似是某个管道相关的链接指针。
97     unsigned             i_dir_seq;
98 };
99
100     __u32                 i_generation;
101
102 #ifdef CONFIG_FSNOTIFY
103     __u32                 i_fsnotify_mask; /* all events this inode cares about
104     */
105     struct fsnotify_mark_connector __rcu *i_fsnotify_marks;
106 #endif
107
108 #ifdef CONFIG_FS_ENCRYPTION
109     struct fscrypt_info *i_crypt_info;
110 #endif
111
112 #ifdef CONFIG_FS_VERITY
113     struct fsverity_info *i_verity_info;
114 #endif
115
116     void                 *i_private; /* fs or device private pointer */
117 } __randomize_layout;

```

了解 struct inode 结构体里面的信息, 注释如上(带中文的注释都是).

基本理解了inode的组成结构和实现方式之后, 显然接着就是目录项了, 注意目录项也是一个文件, 目录文件的结构非常简单, 就是一系列目录项的列表. 每个目录项由两部分组成: 所包含文件的文件名, 以及该文件名对应的inode号码.

来看源码:

```

1  struct dentry {
2      /* RCU lookup touched fields */
3      unsigned int d_flags;           /* protected by d_lock */           // 目录项标识.
4      seqcount_t d_seq;               /* per dentry seqlock */           // short for sequential lock 的结点
5      struct hlist_bl_node d_hash;    /* lookup hash list */           // 哈希表表项结点指针.
6      struct dentry *d_parent;        /* parent directory */           // 父目录项.
7      struct qstr d_name;
8      struct inode *d_inode;          /* where the name belongs to - NULL is
9      * negative */                 // 与文件名关联的索引节点.
10     unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */       // 取别名.
11
12     /* Ref lookup also touches following */
13     struct lockref d_lockref;        /* per-dentry lock and refcount */
14     const struct dentry_operations *d_op;
15     struct super_block *d_sb;        /* The root of the dentry tree */   // 同 inode 中的意思, super block.
16     unsigned long d_time;            /* used by d_revalidate */
17     void *d_fsdata;                 /* fs-specific data */           // //与文件系统相关的数据.

```

```

18
19     union {
20         struct list_head d_lru;      /* LRU list */      // 为使用链表的指针，这
里和LRU页面替换没有关系。
21         wait_queue_head_t *d_wait; /* in-lookup ones only */
22     };
23     struct list_head d_child; /* child of parent list */
24     struct list_head d_subdirs; /* our children */
25     /*
26      * d_alias and d_rcu can share memory
27      */
28     union {
29         struct hlist_node d_alias; /* inode alias list */
30         struct hlist_bl_node d_in_lookup_hash; /* only for in-lookup ones
*/
31         struct rcu_head d_rcu;
32     } d_u;
33 } __randomize_layout;

```

这里就要说到VFS虚拟文件系统, 这是Linux内核里提供文件系统接口给用户态应用程序的一个虚拟文件系统层. 同时VFS还提供了抽象化的操作接口以方便实现内核的底层文件系统. 当用户模式应用程序使用完整路径比如/root/zykoslav-virtual-machine/test, 启动文件访问操作 (比如 `open()` 库函数) 时, VFS 将需要执行目录查找操作以解码和验证路径中指定的每个组件, 为了有效地查找和转换文件路径中的组件, VFS枚举了dentry, dentry对象包含文件或目录的字符串名称, 指向其inode的指针以及指向父dentry的指针.

接下来就是 `struct file` 了:

`file struct` 对应的结构体代表打开的文件, 在内核空间每一个 `file struct` 都对应着一个打开的文件, 几乎所有的函数或者方法操纵文件都需要通过这个结构体. 在所有的文件实例关闭之后, kernel 会释放这个数据结构, 不像disk file, 这就是inode和file struct的区别之处. 在kernel源码中我发现, 指向 `file struct` 的指针是 `filp`, 好像就是file point的意思.

上源码:

```

1  struct file {
2      mode_t f_mode;          // 读写模式.
3      loff_t f_pos;           // 文件在进程中的偏移量.
4      unsigned short f_flags; // 在打开时设置的flags.
5      unsigned short f_count;  // 标记有多少进程打开了该文件.
6      unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
7      struct file *f_next, *f_prev; // 前驱.
8      int f_owner;             /* pid or -pgrp where SIGIO should be sent */
9      struct inode * f_inode;   // 文件指向的inode节点.
10     struct file_operations * f_op; // 操作类型.
11     unsigned long f_version;
12     void *private_data; /* needed for tty driver, and maybe others */
13 };

```

注意这里一些比较重要典型的成员:

`mode_t f_mode`, 文件模式用来确定文件是可读的或者是可写的, 通过位 `FMODE_READ` 和 `FMODE_WRITE`. 内核在调用方法之前检查. 当文件还没有为那种存取而打开时读或写的企图时会被拒绝.

`loff_t f_pos`, 当前读写位置. 驱动可以读这个值, 如果它需要知道文件中的当前位置, 但是正常地不应该改变它; 读和写应当使用它们作为最后参数而收到的指针来更新一个位置, 代替直接作用于 `filp->f_pos`. 这个规则的一个例外是在 `llseek` 方法中, 它的目的就是改变文件位置.

`unsigned int f_flags` 是文件标志, 例如 `O_RDONLY`, `O_NONBLOCK`, 和 `O_SYNC`. 应当检查 `O_NONBLOCK` 标志来看是否是请求非阻塞操作, 所有的标志在头文件 `<linux/fcntl.h>` 中定义.

`struct file_operations *f_op` 是和文件关联的操作. 内核安排指针作为它的 `open` 实现的一部分, 接着读取它当它需要分派任何的操作时. `filp->f_op` 中的值从不由内核保存为后面的引用.

为内核添加一个新的系统调用 **filesys**, 其从调用者接收一个磁盘文件的全局路径名, 打印该文件占用的所有磁盘块

实现思路

注意到 `ioctl` 函数, 通过这里: <https://stackoverflow.com/questions/38669605/how-to-use-ioctl-with-fs-ioc-fiemap>

查 [Manual](#):

```
1  NAME
2      ioctl - control device
3  SYNOPSIS
4      #include <sys/ioctl.h>
5
6      int ioctl(int fd, unsigned long request, ...);
7  DESCRIPTION
8      The ioctl() system call manipulates the underlying device parameters
9      of special files. In particular, many operating characteristics of
10     character special files (e.g., terminals) may be controlled with
11     ioctl() requests. The argument fd must be an open file descriptor.
12
13     The second argument is a device-dependent request code. The third
14     argument is an untyped pointer to memory. It's traditionally char
15     *argp (from the days before void * was valid C), and will be so
16     named
17     for this discussion.
18
19     An ioctl() request has encoded in it whether the argument is an in
20     parameter or out parameter, and the size of the argument argp in
21     bytes. Macros and defines used in specifying an ioctl() request are
22     located in the file <sys/ioctl.h>.
23  RETURN VALUE
24     Usually, on success zero is returned. A few ioctl() requests use the
25     return value as an output parameter and return a nonnegative value
26     on
27     success. On error, -1 is returned, and errno is set appropriately.
```

主要讲的就是 `ioctl` 函数调用操作特殊文件的底层设备参数, 特别是字符特殊文件(如终端)的许多操作特征可能由 `ioctl()` 请求控制. 参数 `fd` 必须是一个打开的文件描述符. 第二个参数是设备相关的请求代码. 第三个参数是指向内存的无类型指针 (`char *argp`), 自变量是 `in` 参数还是 `out` 参数, `ioctl()` 请求都已在其中进行编码, 参数 `argp` 的大小以字节为单位. 用于指定 `ioctl()` 请求的宏和定义位于文件 `<sys/ioctl.h>` 中.

这个函数的实现在 `/linux/fs/ioctl.c` 中.

```
1  static int file_ioctl(struct file *filp, unsigned int cmd,
```

```

2     unsigned long arg)
3 {
4     struct inode *inode = file_inode(filp);
5     int __user *p = (int __user *)arg;
6
7     switch (cmd) {
8     case FIBMAP:
9         return ioctl_fibmap(filp, p);
10    case FIONREAD:
11        return put_user(i_size_read(inode) - filp->f_pos, p);
12    case FS_IOC_RESVSP:
13    case FS_IOC_RESVSP64:
14        return ioctl_preallocate(filp, p);
15    }
16
17    return vfs_ioctl(filp, cmd, arg);
18 }

```

就像实验三一样, 系统调用不能引用的函数, 自己写一遍. 按照它的逻辑来即可.

这里可以看到获取磁盘号的函数, 这个比实验三好多了, 实验三太多层页表了, 这里清晰简单很多, 那就再看看 `ioctl_fibmap()` 函数:

```

1 static int ioctl_fibmap(struct file *filp, int __user *p)
2 {
3     struct address_space *mapping = filp->f_mapping;
4     int res, block;
5
6     /* do we support this mess? */
7     if (!mapping->a_ops->bmap)
8         return -EINVAL;
9     if (!capable(CAP_SYS_RAWIO))
10        return -EPERM;
11    res = get_user(block, p);
12    if (res)
13        return res;
14    res = mapping->a_ops->bmap(mapping, block);
15    return put_user(res, p);
16 }

```

这几个函数在一起就可以获取文件信息之后先计算出分配块有多少, 然后遍历这些块, 一个一个用 `ioctl()` 函数去测试, 如果可以就输出:

综上所述, 可以实现如下系统调用:

```

1 #include <linux/types.h>
2 #include <linux/ioctl.h>
3 #include <linux/stat.h>
4 #include <linux/fcntl.h>
5 #include <linux/hdreg.h>
6 #include <linux/module.h>
7 #include <linux/uaccess.h>
8 #include <linux/capability.h>
9 #include <linux/mm_types.h>
10 #include <linux/capability.h>
11 #include <linux/fs.h>
12 #include <linux/rbtree.h>

```

```

13 #include <linux/init.h>
14 #include <linux/pid.h>
15 #include <linux/rwsem.h>
16 #include <linux/errseq.h>
17 #include <linux/ioprio.h>
18 #include <linux/fs_types.h>
19 #include <linux/stddef.h>
20 #include <linux/fiemap.h>
21 #include <linux/rculist_bl.h>
22 #include <linux/atomic.h>
23 #include <linux/shrinker.h>
24 #include <linux/migrate_mode.h>
25 #include <linux/uidgid.h>
26
27 asmlinkage long sys_filesys(const char __user *para){
28     struct inode *myInode = NULL;
29     struct file *myFilp = NULL;
30
31     myFilp = filp_open(para, O_RDONLY, 0);
32
33     myInode = fp->f_path.dentry->d_inode;    // 相当于找到指向文件的指针。
34     int blockNum = myInode->i_blocks;        // 占有多少块。
35     int fileSize = myInode->i_size;          // 文件大小，根据上面阅读源码注释的
inode结构体。
36
37     int okBlock = (fileSize + (3 << node->i_blkbits) - 1) / blockSize; //
有效磁盘块数。(3 << node->i_blkbits)是块大小
38
39     printk("File: %s START!\n", para);
40
41     struct address_space *myMapping = filp->f_mapping;
42     for(i = 0; i < okBlock; i++) {
43         printk("%3d %10d\n", i, myMapping->a_ops->bmap(myMapping, i));
44     }
45     filp_close(myFilp, NULL);
46
47     return 1;
48 }

```

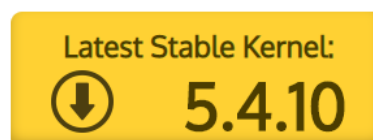
具体实现过程

实验环境：



内核源码是[The Linux Kernel Archives](https://www.kernel.org/)下载的 5.4.10 (Latest Stable Kernel).

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

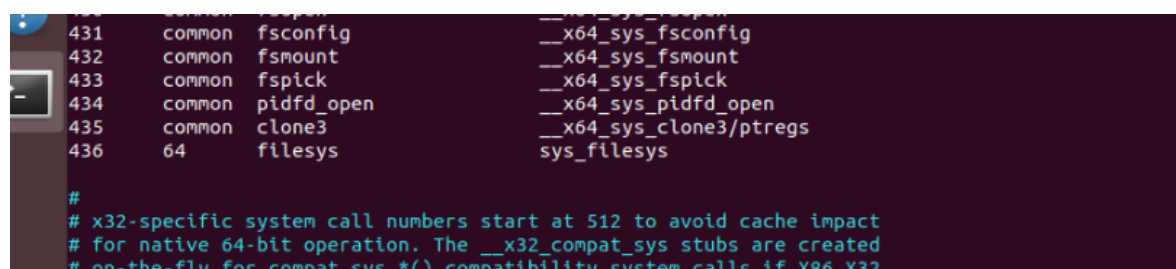


实验过程

设计并添加系统调用

1. 分配系统调用号, 修改系统调用表

```
./arch/x86/entry/syscalls/syscall_64.tbl
```



2. 声明系统调用原型

```
./include/linux/syscalls.h
```

```

        unsigned long fd, unsigned long pgoff);
asmlinkage long sys_old_mmap(struct mmap_arg_struct __user *arg);

/*
 * Not a real system call, but a placeholder for syscalls which are
 * not implemented -- see kernel/sys_ni.c
 */
asmlinkage long sys_ni_syscall(void);

#endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */

asmlinkage long sys_filesys(const char __user *para);

/*
 * Kernel code should not call syscalls (i.e., sys_xyzzyz()) directly.
 * Instead, use one of the functions which work equivalently, such as
 * the ksys_xyzzyz() functions prototyped below.
 */

int ksys_mount(const char __user *dev_name, const char __user *dir_name,
               const char __user *type, unsigned long flags, void __user *data);
int ksys_umount(char __user *name, int flags);
int ksys_dup(unsigned int fd);
~

```

在光标的地方可以发现添加了系统调用原型的声明。

3. 实现系统调用

如上代码:

```

1  #include <linux/types.h>
2  #include <linux/ioctl.h>
3  #include <linux/stat.h>
4  #include <linux/fcntl.h>
5  #include <linux/hdreg.h>
6  #include <linux/module.h>
7  #include <linux/uaccess.h>
8  #include <linux/capability.h>
9  #include <linux/mm_types.h>
10 #include <linux/capability.h>
11 #include <linux/fs.h>
12 #include <linux/rbtree.h>
13 #include <linux/init.h>
14 #include <linux/pid.h>
15 #include <linux/rwsem.h>
16 #include <linux/errseq.h>
17 #include <linux/ioprio.h>
18 #include <linux/fs_types.h>
19 #include <linux/stddef.h>
20 #include <linux/fiemap.h>
21 #include <linux/rculist_bl.h>
22 #include <linux/atomic.h>
23 #include <linux/shrinker.h>
24 #include <linux/migrate_mode.h>
25 #include <linux/uidgid.h>
26
27 asmlinkage long sys_filesys(const char __user *para){
28     struct inode *myInode = NULL;
29     struct file *myFilp = NULL;
30
31     myFilp = filp_open(para, O_RDONLY, 0);
32
33     myInode = fp->f_path.dentry->d_inode;    // 相当于找到指向文件的指针。
34     int blockNum = myInode->i_blocks;        // 占有多少块。
35     int fileSize = myInode->i_size;          // 文件大小，根据上面阅读源码注释的
inode结构体。
36

```



```

37     int okBlock = (fileSize + (3 << node->i_blkbits) - 1) / blockSize; //
    有效磁盘块数. (3 << node->i_blkbits)是块大小
38
39     printk("File: %s START!\n", para);
40
41     struct address_space *myMapping = filp->f_mapping;
42     for(i = 0; i < okBlock; i++) {
43         printk("%3d %10d\n", i, myMapping->a_ops->bmap(myMapping, i));
44     }
45     filp_close(myFilp, NULL);
46
47     return 1;
48 }

```

传入参数是文件路径;

编译内核

1. 清除残留的.config 和.o 文件

```
1 make mrproper
```

```

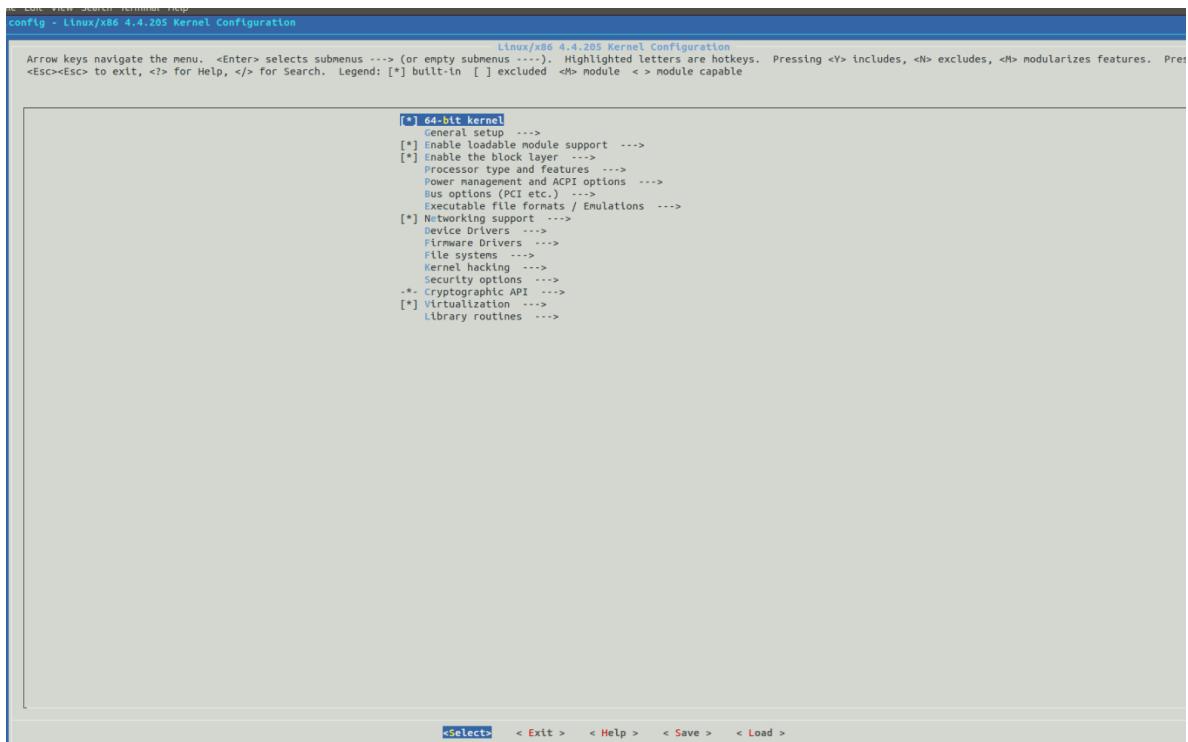
zykoslab@zykoslab-virtual-machine:/usr/src/linux-5.4.10$ make mrproper
zykoslab@zykoslab-virtual-machine:/usr/src/linux-5.4.10$

```

(上面是第1次编译了, 所以CLEAN之后几乎没有)

2. 配置内核

```
1 make menuconfig
```



```

zykoslab@zykoslab-virtual-machine: /usr/src/linux-5.4.10$ sudo make menuconfig
HOSTCC scripts/basic/fixdep
UPD scripts/kconfig/mconf-cfg
HOSTCC scripts/kconfig/mconf.o
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/util.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c

```

3. 编译内核, 生成启动映像文件

```
1 | make
```

```

LD [M] arch/x86/crypto/serpent-avx-x86_64.o
AS [M] arch/x86/crypto/camellia-aesni-avx2-asm_64.o
CC [M] arch/x86/crypto/camellia-aesni-avx2_glue.o
LD [M] arch/x86/crypto/camellia-aesni-avx2.o
AS [M] arch/x86/crypto/serpent-avx2-asm_64.o
CC [M] arch/x86/crypto/serpent_avx2_glue.o
LD [M] arch/x86/crypto/serpent-avx2.o
AS arch/x86/entry/entry_64.o
AS arch/x86/entry/thunk_64.o
CC arch/x86/entry/syscall_64.o
CC arch/x86/entry/common.o
CC arch/x86/entry/vdso/vma.o
CC arch/x86/entry/vdso/vdso32-setup.o
LDS arch/x86/entry/vdso/vdso.lds
AS arch/x86/entry/vdso/vdso-note.o
CC arch/x86/entry/vdso/vclock_gettime.o
CC arch/x86/entry/vdso/vgetcpu.o
VDISO arch/x86/entry/vdso/vdso64.so.dbg
HOSTCC arch/x86/entry/vdso/vdso2c
OBJCOPY arch/x86/entry/vdso/vdso64.so
VDISO2C arch/x86/entry/vdso/vdso-image-64.c
CC arch/x86/entry/vdso/vdso-image-64.o
LDS arch/x86/entry/vdso/vdso32.lds

```

4. 编译模块

```
1 | make modules
```

5. 安装内核

```
1 | make install
```

```
Activities Terminal 六 11:58 zykoslab@zykoslab-virtual-machine: /usr/src/linux
File Edit View Search Terminal Help
INSTALL drivers/hwmon/tmp401.ko
INSTALL drivers/hwmon/tmp421.ko
INSTALL drivers/hwmon/via-cputemp.ko
INSTALL drivers/hwmon/via686a.ko
INSTALL drivers/hwmon/vt1211.ko
INSTALL drivers/hwmon/vt8231.ko
INSTALL drivers/hwmon/w83627ehf.ko
INSTALL drivers/hwmon/w83627hf.ko
INSTALL drivers/hwmon/w83781d.ko
INSTALL drivers/hwmon/w83791d.ko
INSTALL drivers/hwmon/w83792d.ko
INSTALL drivers/hwmon/w83793.ko
INSTALL drivers/hwmon/w83795.ko
INSTALL drivers/hwmon/w83l785ts.ko
INSTALL drivers/hwmon/w83l786ng.ko
INSTALL drivers/hwmon/wm831x-hwmon.ko
INSTALL drivers/hwmon/wm8350-hwmon.ko
INSTALL drivers/hwtracing/intel_th/intel_th.ko
INSTALL drivers/hwtracing/intel_th/intel_th_gth.ko
INSTALL drivers/hwtracing/intel_th/intel_th_msu.ko
INSTALL drivers/hwtracing/intel_th/intel_th_pci.ko
INSTALL drivers/hwtracing/intel_th/intel_th_pti.ko
INSTALL drivers/hwtracing/intel_th/intel_th_sth.ko
INSTALL drivers/hwtracing/stm/dummy_stm.ko
INSTALL drivers/hwtracing/stm/stm_console.ko
INSTALL drivers/hwtracing/stm/stm_core.ko
INSTALL drivers/i2c/algos/i2c-algo-bit.ko
INSTALL drivers/i2c/algos/i2c-algo-pca.ko
INSTALL drivers/i2c/busses/i2c-ali1535.ko
INSTALL drivers/i2c/busses/i2c-ali1563.ko
INSTALL drivers/i2c/busses/i2c-ali15x3.ko
INSTALL drivers/i2c/busses/i2c-amd756-s4882.ko
INSTALL drivers/i2c/busses/i2c-amd756.ko
INSTALL drivers/i2c/busses/i2c-amd8111.ko
INSTALL drivers/i2c/busses/i2c-cbus-gpio.ko
INSTALL drivers/i2c/busses/i2c-cros-ec-tunnel.ko
INSTALL drivers/i2c/busses/i2c-designware-pci.ko
INSTALL drivers/i2c/busses/i2c-diolan-u2c.ko
INSTALL drivers/i2c/busses/i2c-dln2.ko
INSTALL drivers/i2c/busses/i2c-gpio.ko
INSTALL drivers/i2c/busses/i2c-i801.ko
INSTALL drivers/i2c/busses/i2c-isch.ko
INSTALL drivers/i2c/busses/i2c-ismt.ko
INSTALL drivers/i2c/busses/i2c-kempld.ko
INSTALL drivers/i2c/busses/i2c-nforce2-s4985.ko
INSTALL drivers/i2c/busses/i2c-nforce2.ko
INSTALL drivers/i2c/busses/i2c-ocores.ko
INSTALL drivers/i2c/busses/i2c-parport-light.ko
INSTALL drivers/i2c/busses/i2c-parport.ko
INSTALL drivers/i2c/busses/i2c-pca-platform.ko
INSTALL drivers/i2c/busses/i2c-piix4.ko
INSTALL drivers/i2c/busses/i2c-robotfuzz-ostf.ko
INSTALL drivers/i2c/busses/i2c-scmi.ko
```

6. 配置 grub 引导

```
1 | sudo update-grub2
```

7. 重启

```
1 | reboot
```

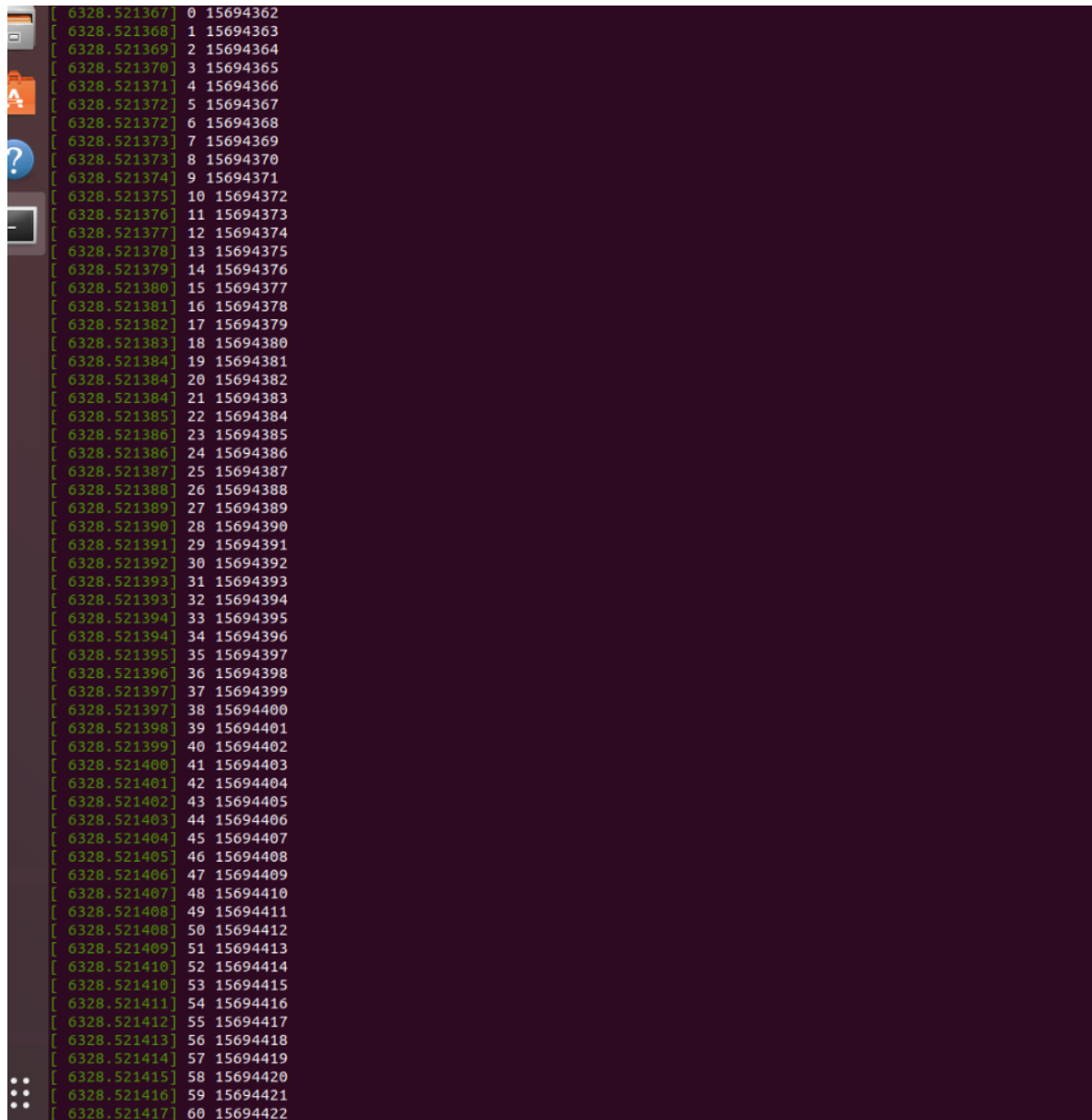
编写用户态测试程序

```
1 | #define _GNU_SOURCE
2 | #include <unistd.h>
3 | #include <sys/syscall.h>
4 | #include <stdio.h>
5 | #define __NR_memsys 436 // 系统调用号
6 |
7 | int main() {
8 |     syscall(__NR_memsys, "/home/zykoslab/Downloads/The Matrix Calculus You
   Need For Deep Learning.pdf");
9 |
10 |     return 0;
11 | }
```

打开一个终端, 监视并且打印当前系统的**日志信息**. 也就是测试程序的日志输出.

```
1 while true
2 do
3     sudo dmesg -c
4     sleep 1
5 done
```

最后结果如下:

A terminal window with a dark purple background and light green text. The left sidebar shows icons for a file manager, a terminal, and a help icon. The terminal displays a continuous stream of dmesg log messages, each on a new line. Each line starts with a bracketed timestamp, followed by a kernel message ID in square brackets, and then the message content. The messages are repetitive, showing kernel boot logs and hardware initialization details. The output is truncated on the right side of the terminal window.

```
[ 6328.521367] 0 15694362
[ 6328.521368] 1 15694363
[ 6328.521369] 2 15694364
[ 6328.521370] 3 15694365
[ 6328.521371] 4 15694366
[ 6328.521372] 5 15694367
[ 6328.521372] 6 15694368
[ 6328.521373] 7 15694369
[ 6328.521373] 8 15694370
[ 6328.521374] 9 15694371
[ 6328.521375] 10 15694372
[ 6328.521376] 11 15694373
[ 6328.521377] 12 15694374
[ 6328.521378] 13 15694375
[ 6328.521379] 14 15694376
[ 6328.521380] 15 15694377
[ 6328.521381] 16 15694378
[ 6328.521382] 17 15694379
[ 6328.521383] 18 15694380
[ 6328.521384] 19 15694381
[ 6328.521384] 20 15694382
[ 6328.521384] 21 15694383
[ 6328.521385] 22 15694384
[ 6328.521386] 23 15694385
[ 6328.521386] 24 15694386
[ 6328.521387] 25 15694387
[ 6328.521388] 26 15694388
[ 6328.521389] 27 15694389
[ 6328.521390] 28 15694390
[ 6328.521391] 29 15694391
[ 6328.521392] 30 15694392
[ 6328.521393] 31 15694393
[ 6328.521393] 32 15694394
[ 6328.521394] 33 15694395
[ 6328.521394] 34 15694396
[ 6328.521395] 35 15694397
[ 6328.521396] 36 15694398
[ 6328.521397] 37 15694399
[ 6328.521397] 38 15694400
[ 6328.521398] 39 15694401
[ 6328.521399] 40 15694402
[ 6328.521400] 41 15694403
[ 6328.521401] 42 15694404
[ 6328.521402] 43 15694405
[ 6328.521403] 44 15694406
[ 6328.521404] 45 15694407
[ 6328.521405] 46 15694408
[ 6328.521406] 47 15694409
[ 6328.521407] 48 15694410
[ 6328.521408] 49 15694411
[ 6328.521408] 50 15694412
[ 6328.521409] 51 15694413
[ 6328.521410] 52 15694414
[ 6328.521410] 53 15694415
[ 6328.521411] 54 15694416
[ 6328.521412] 55 15694417
[ 6328.521413] 56 15694418
[ 6328.521414] 57 15694419
[ 6328.521415] 58 15694420
[ 6328.521416] 59 15694421
[ 6328.521417] 60 15694422
```

```
[ 6328.521418] 61 15694423
[ 6328.521419] 62 15694424
[ 6328.521420] 63 15694425
[ 6328.521421] 64 15694426
[ 6328.521421] 65 15694427
[ 6328.521421] 66 15694428
[ 6328.521422] 67 15694429
[ 6328.521423] 68 15694430
[ 6328.521424] 69 15694431
[ 6328.521425] 70 15694432
[ 6328.521426] 71 15694433
[ 6328.521426] 72 15694434
[ 6328.521427] 73 15694435
[ 6328.521428] 74 15694436
[ 6328.521429] 75 15694437
[ 6328.521430] 76 15694438
[ 6328.521431] 77 15694439
[ 6328.521432] 78 15694440
[ 6328.521433] 79 15694441
[ 6328.521434] 80 15694442
[ 6328.521434] 81 15694443
[ 6328.521435] 82 15694444
[ 6328.521435] 83 15694445
[ 6328.521435] 84 15694446
[ 6328.521435] 85 15694447
[ 6328.521436] 86 15694448
[ 6328.521437] 87 15694449
[ 6328.521438] 88 15694450
[ 6328.521439] 89 15694451
[ 6328.521439] 90 15694452
[ 6328.521440] 91 15694453
[ 6328.521441] 92 15694454
[ 6328.521442] 93 15694455
[ 6328.521442] 94 15694456
[ 6328.521443] 95 15694457
[ 6328.521443] 96 15694458
[ 6328.521444] 97 15694459
[ 6328.521445] 98 15694460
[ 6328.521446] 99 15694461
[ 6328.521447] 100 15694462
[ 6328.521447] 101 15694463
[ 6328.521448] 102 15694464
[ 6328.521449] 103 15694465
[ 6328.521450] 104 15694466
[ 6328.521451] 105 15694467
[ 6328.521452] 106 15694468
[ 6328.521452] 107 15694469
[ 6328.521453] 108 15694470
[ 6328.521454] 109 15694471
[ 6328.521455] 110 15694472
[ 6328.521456] 111 15694473
[ 6328.521457] 112 15694474
[ 6328.521458] 113 15694475
[ 6328.521458] 114 15694476
[ 6328.521459] 115 15694477
[ 6328.521460] 116 15694478
```

这里使用的是700K+的文件, 在这些截图之后还有一些.

遇到的问题

vim中修改文件时报错: `E212 can't open file for writing`, 保存文件时用 `:w !sudo tee %` 即可, 其中 `tee` 用于读取输入文件, 同时保存, `%` 表示当前编辑文件;

还有各种编译错误, 因为实验三已经被虐得很惨所以都习惯啦.

还有很多没有记录下来的问题, 最后一个实验确实时间紧压力大, 不过总算完成啦!

总结

谢谢老师一学期的教导, 操作系统确实是很有内容的一门课, 在实验过程中看源码看文档的能力也提升了不少, 很充实很快乐.

祝老师新年快乐! 也祝我能在未来的计算机科学领域踏实地前行!