《操作系统》课第十次实验报告

学院:	软件学院
姓名:	杨万里
学号:	2013774
邮箱:	2013774@mail.nankai.edu.cn
时间:	2022/11/19 修改于 2022/11/20

0. 前言

抱歉老师，我之前理解错了本实验的部分目标（将进程信息返回到用户程序），在 11.19 提交了一份实验报告，我重新理解了该目标，因此完善了实验，在 11.20 重新提交本实验报告。

1. 开篇感言

完成本次实验又是一个痛苦与快乐交织的过程。

通过本次实验，我再次意识到，我们本科生学习计算机的相关知识与实践时，绝大多数时候都是站在巨人的肩膀上：前人/老师给我们参照的方法和框架，我们完成实验。一旦实验的经验过时或者实验相关的经验特别少时，我们就像暴露在真实世界的温室的花朵一样脆弱。如果本次实验中，get_fs()等方法还没有过时，那么想必实验过程会非常顺利。但是实验的挑战就在于这套方法已经被 Linux 新内核抛弃了，网络上关于替代方案的资料和相关知识非常少。因此实验的过程变得很艰难。这也足以想象创建和维护 Linux 内核的程序员的伟大。

本次实验多次涉及到用户空间和内核空间的互相拷贝，让我深刻认识到系统内核的美妙！

并且由于实验编写的是系统调用，不像是在用户态写一个程序就可以轻松运行调试。系统调用的调试非常困难，运行结果的验证也需要重新编译、安装、重启等操作之后才能进行，即使不需要 make clean，也仍然很麻烦。

但是快乐就在于通过自己的努力再次战胜了困难。

《操作系统》课程的实验真有魅力！

2. 实验题目

增加一个带参数的系统调用实现列举所有进程和拷贝文件的功能

3. 实验目标

（1） 在 Linux 内核当中增加一个新的带参数的系统调用

（2） 新的系统调用会返回所有进程的信息到用户程序

（3） 实现内核中文件拷贝

4. 原理方法

（实验源代码附在实验报告结尾处）

（1） 在 Linux 内核中增加系统调用的原理方法：

1） 在 include/linux/syscalls.h 文件当中声明想要添加的系统调用

2） 在 kernel/sys.c 文件当中实现系统调用的具体内容

3） 在 arch/x86/entry/syscalls/syscall_64.tbl 文件当中注册系统调用

4） 重新编译安装内核并重启使之生效

（2） 列举所有进程信息和进程数目的原理方法：

```
char* BUF = kmalloc(sizeof(char)*18000,GFP_ATOMIC);
char* src = kmalloc(sizeof(char)*30,GFP_ATOMIC);
char* tgt = kmalloc(sizeof(char)*30,GFP_ATOMIC);
char temp[45];
int offset = 0;
int i;
p = &init_task;
for_each_process(p)
{
    memset(temp,'\0',45);
    sprintf(temp, "%-20s %6d %4c 2013774\n",p->comm,p->pid,task_state_to_char(p));
    i = 0;
    while(temp[i]!='\0'){
            BUF[offset]=temp[i];
            i++;
            offset++;
    }
    sum += 1;
}
memset(temp,'\0',45);
sprintf(temp, "the number of processes is %d 2013774\n", sum);
i = 0;
while(temp[i]!='\0'){
     BUF[offset]=temp[i];
     i++;
     offset++;
}
//进程信息存储完成
//拷贝到用户空间
value = copy_to_user(buf,BUF,offset);
```

在**内核空间创建缓冲区 BUF**，以及临时存储区 temp。

获取初始进程的指针&init_task，从该指针出发，遍历所有进程，每遇到一个进程就会获取其基本信息，并**将基本信息转化为字符串**，存储在 temp 中，同时将 temp 的内容**添加到 BUF** 当中，并且在统计进程数目上加 1。

遍历所有进程之后，**BUF 存储了所有进程信息**，利用 copy_to_user 将 BUF 的内容**拷贝到用户空间的 buf**（参数传入）指向的地址当中。

（3） 在内核中实现文件拷贝的原理方法：

这一步是本实验的核心，也是个人认为本实验中最难的一部分。

在用户态编写文件拷贝的程序是相对比较简单的，但是在内核当中，文件的打开与读写有更多的限制（这也是为了保护内核）。

```
struct file *file = NULL;
mm_segment_t old_fs;
char buf[128] = "123456";
file = filp_open("/data/test.txt", O_RDWR | O_APPEND | O_CREAT, 0644);
if (IS_ERR(file)) {
      return 0;
}
old_fs = get_fs();
set_fs(KERNEL_DS);
vfs_write(file, buf, sizeof(buf),  0);
set_fs(old_fs);
filp_close(file, NULL);
return 0;
```

get_fs()
filp_open(), vfs_read(),vfs_write(), filp_close()

实验指导书给出的参考框架如上图所示，打开文件采用 filp_open()，读文件用 vfs_read()，写文件用 vfs_write()。

可以了解到：vfs_read 和 vfs_write 这两个函数的第二个参数 buffer 前面都有**__user 修饰符**，这要求 buffer 指针都应该**指向用户空间的内存**，如果对该参数传递内核空间的指针，这两个函数都会返回失败。但在 Kernel 中，我们一般不容易生成用户空间的指针，或者不方便独立使用用户空间内存。**要使这两个读写函数使用 kernel 空间的 buffer 指针也能正确工作**，需要使用 set_fs()函数。**该函数的作用是改变 kernel 对内存地址检查的处理方式**，该函数的参数 fs 只有两个取值：USER_DS，KERNEL_DS，分别代表用户空间和内核空间，默认情况下，kernel 取值为 USER_DS，即对用户空间地址检查并做变换。要在这种对内存地址做检查变换的函数中**使用内核空间地址**，就**需要使用 set_fs(KERNEL_DS)进行设置**。

简单地说就是 vfs_read 和 vfs_write 需要用户空间内存，但是我们使用的是内核空间的内存，因此需要使用 set_fs()的相关设置修改检查处理方式。

参照上述框架编写拷贝文件的程序，如下图所示。

```
int count;
struct file *src_file = NULL;
struct file *tgt_file = NULL;
mm_segment_t old_fs;
char buffer[128];
src_file = filp_open("/home/wanliyang2013774/test.txt", O_RDWR | O_APPEND | O_CREAT, 0644);
tgt_file = filp_open("/home/wanliyang2013774/copy.txt", O_RDWR | O_APPEND | O_CREAT, 0644);
if (IS_ERR(src_file)) {
    return 0;
}
if (IS_ERR(tgt_file)) {
    return 0;
}
old_fs = get_fs();
set_fs(KERNEL_DS);
while((count = vfs_read(src_file,buffer,128,0))>0)
{
    if(vfs_write(tgt_file,buffer,count,0)!= count)
        return 3;
}
if(count == -1) return 4;
set_fs(old_fs);
filp_close(src_file,NULL);
filp_close(tgt_file,NULL);
```

且不论程序本身的正确性，编译内核的时候就会报出下图所示的错误

```
kernel/sys.c:963:9: error: unknown type name ‘mm_segment_t’
  963 |         mm_segment_t fs;
      |         ^~~~~~~~~~~~
kernel/sys.c:964:14: error: implicit declaration of function ‘get_fs’; did you mean ‘sget_fc’? [-
Werror=implicit-function-declaration]
  964 |         fs = get_fs();
      |              ^~~~~~
      |              sget_fc
kernel/sys.c:965:9: error: implicit declaration of function ‘set_fs’; did you mean ‘sget_fc’? [-W
error=implicit-function-declaration]
  965 |         set_fs(get_ds());
      |         ^~~~~~
      |         sget_fc
```

报错**识别不出 mm_segment_t 和 get_fs()以及 set_fs()**，一时摸不着头脑，在网络上查阅了很多资料，涉及到在内核中读写文件基本都使用的是 get_fs()的相关设置。科学上网之后查阅了很多资料和相关讨论，在一个讨论中有人说道：**Linux 内核已经抛弃了 set_fs 和 get_fs**，只能寻找替代方案。

继续查找资料，看到了 **kernel_read 和 kernel_write** 两个函数，虽然它们作为读写文件的函数好像不是 get_fs 等的替代品，但是这两个函数的出现**避免了地址所在空间的检查**，因此也就**不需要 set_fs 和 get_fs 等系列设置**了，这看起来也是 Linux 内核的进步。

采用这两个函数重新编写程序，如下图所示

```
value = copy_from_user(src,source,30);
value = copy_from_user(tgt,target,30);
printk("%s to %s 2013774",src,tgt);
src_file = filp_open(src, O_RDWR | O_APPEND | O_CREAT, 0644);
tgt_file = filp_open(tgt, O_RDWR | O_APPEND | O_CREAT, 0644);
if (IS_ERR(src_file)) {
    printk("fail to open file 2013774");
    return 0;
}
if (IS_ERR(tgt_file)) {
    printk("fail to open file 2013774");
    return 0;
}
src_pos = src_file->f_pos;
tgt_pos = tgt_file->f_pos;
while((count = kernel_read(src_file,buffer,128,&src_pos))>0)
{
        kernel_write(tgt_file,buffer,count,&tgt_pos);
}
filp_close(src_file,NULL);
filp_close(tgt_file,NULL);
```

这一步做了改进，拷贝源文件和目标**文件来自系统调用的参数**，这样可以**通过命令行输入来确定源文件和目标文件**，更具灵活性。

但是要注意，需要**把来自用户空间的文件路径拷贝到内核空间**，才可以使用。

这里需要非常关注 f_pos 这个属性，这标记了**读写文件的当前偏移量**，是文件指针自带的属性（此前不知道这个属性吃了很大的亏，程序一直有 bug），这个标记会随着文件读写自动偏移。

接着就是循环读写文件直至结束，最后关闭文件。

5. 具体步骤

在开始实验之前，需要更新内核到最新的 Linux6.0.9

检查当前内核版本，为 5.19.10

```
wanliyang2013774@wanliyang2013774-virtual-machine:/usr/src/linux$ uname -a
Linux wanliyang2013774-virtual-machine 5.19.10 #4 SMP PREEMPT_DYNAMIC Sat Oct 15 19:
46:41 CST 2022 x86_64 x86_64 x86_64 GNU/Linux
```

下载最新内核

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ wget -c https://cdn.kernel.org/
pub/linux/kernel/v6.x/linux-6.0.9.tar.xz
--2022-11-18 16:55:14--  https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.0.9.ta
r.xz
Resolving cdn.kernel.org (cdn.kernel.org)... 146.75.113.176, 2a04:4e42:8c::432
Connecting to cdn.kernel.org (cdn.kernel.org)|146.75.113.176|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 133911648 (128M) [application/x-xz]
Saving to: ‘linux-6.0.9.tar.xz’

linux-6.0.9.tar.xz  100%[===================>] 127.71M  30.9KB/s    in 96m 39s

2022-11-18 18:31:54 (22.6 KB/s) - ‘linux-6.0.9.tar.xz’ saved [133911648/133911648]
```

解压内核并移动到/usr/src/linux 中

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ sudo ln -s `pwd`/linux-6.0.9 /u
sr/src/linux
[sudo] password for wanliyang2013774:
```

编译内核：make -j5

安装内核并重启

```
  AS      arch/x86/boot/compressed/piggy.o
  LD      arch/x86/boot/compressed/vmlinux
  ZOFFSET arch/x86/boot/zoffset.h
  OBJCOPY arch/x86/boot/vmlinux.bin
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
Kernel: arch/x86/boot/bzImage is ready  (#1)
wanliyang2013774@wanliyang2013774-virtual-machine:/usr/src/linux$
```

检查内核版本，为 Linux6.0.9，是最新版本

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ uname -a
Linux wanliyang2013774-virtual-machine 6.0.9 #1 SMP PREEMPT_DYNAMIC Fri Nov 18 1
8:45:32 CST 2022 x86_64 x86_64 x86_64 GNU/Linux
wanliyang2013774@wanliyang2013774-virtual-machine:~$
```

开始正式实验

（1） 在 include/linux/syscalls.h 文件中声明新的系统调用

```
#include <asm/syscall_wrapper.h>
asmlinkage long sys_alcall(char* buf,char* source,char* target);
#endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
```

（2） 在 kernel/sys.c 文件中实现系统调用

```
SYSCALL_DEFINE3(alcall,char*,buf,char*,source,char*,target)
{
    int sum = 0;
    int value;
    int count = 0;
    struct task_struct *p;
    struct file* src_file = NULL;
    struct file* tgt_file = NULL;
    loff_t src_pos;
    loff_t tgt_pos;
    char buffer[128];
    char* BUF = kmalloc(sizeof(char)*18000,GFP_ATOMIC);
    char* src = kmalloc(sizeof(char)*30,GFP_ATOMIC);
    char* tgt = kmalloc(sizeof(char)*30,GFP_ATOMIC);
    char temp[45];
    int offset = 0;
    int i;
    p = &init_task;
    for_each_process(p)
    {
        memset(temp,'\0',45);
        sprintf(temp, "%-20s %6d %4c 2013774\n",p->comm,p->pid,task_state_to_char(p));
        i = 0;
        while(temp[i]!='\0'){
                BUF[offset]=temp[i];
                i++;
                offset++;
        }
        sum += 1;
    }
    memset(temp,'\0',45);
    sprintf(temp, "the number of processes is %d 2013774\n", sum);
    i = 0;
    while(temp[i]!='\0'){
         BUF[offset]=temp[i];
         i++;
         offset++;
    }
    value = copy_to_user(buf,BUF,offset);
    value = copy_from_user(src,source,30);
    value = copy_from_user(tgt,target,30);
    printk("%s to %s 2013774",src,tgt);
    src_file = filp_open(src, O_RDWR | O_APPEND | O_CREAT, 0644);
    tgt_file = filp_open(tgt, O_RDWR | O_APPEND | O_CREAT, 0644);
    if (IS_ERR(src_file)) {
        printk("fail to open file 2013774");
        return 0;
    }
    if (IS_ERR(tgt_file)) {
        printk("fail to open file 2013774");
        return 0;
    }
    src_pos = src_file->f_pos;
    tgt_pos = tgt_file->f_pos;
    while((count = kernel_read(src_file,buffer,128,&src_pos))>0)
    {
            kernel_write(tgt_file,buffer,count,&tgt_pos);
    }
    filp_close(src_file,NULL);
    filp_close(tgt_file,NULL);
    kfree(BUF);
    return 0;
}
```

（3） 在/usr/src/linux/arch/x86/entry/syscalls/syscall_64.tbl 中注册系统调用

系统调用号为 448

```
446     common  landlock_restrict_self  sys_landlock_restrict_self
447     common  memfd_secret            sys_memfd_secret
448     common  alcall                  __x64_sys_alcall
449     common  process_mrelease        sys_process_mrelease
450     common  futex_waitv             sys_futex_waitv
451     common  set_mempolicy_home_node sys_set_mempolicy_home_node
```

（4）编译（不需要 clean）、安装修改后的内核，并重启机器

```
wanliyang2013774@wanliyang2013774-virtual-machine:/usr/src/linux$ make -j5
  SYNC    include/config/auto.conf.cmd
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/confdata.o
  HOSTCC  scripts/kconfig/expr.o
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
  HOSTCC  scripts/kconfig/menu.o
```

```
wanliyang2013774@wanliyang2013774-virtual-machine:/usr/src/linux$ sudo make modules_
install
[sudo] password for wanliyang2013774:
  INSTALL /lib/modules/6.0.9/kernel/arch/x86/crypto/aesni-intel.ko
  SIGN    /lib/modules/6.0.9/kernel/arch/x86/crypto/aesni-intel.ko
  INSTALL /lib/modules/6.0.9/kernel/arch/x86/crypto/crc32-pclmul.ko
  SIGN    /lib/modules/6.0.9/kernel/arch/x86/crypto/crc32-pclmul.ko
```

……

（4） 编写测试程序

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#define _NR_alcall 448
long alcall(char* buf,char* source,char* target){
 return syscall(_NR_alcall,buf,source,target);
}
int main(int argc, char *argv[])
{
 char* source = malloc(sizeof(char)*30);
 char* target = malloc(sizeof(char)*30);
 source = argv[1];
 target = argv[2];
 char* buf = malloc(sizeof(char)*18000);
 printf("the program will copy %s to %s",source,target);
 alcall(buf,source,target);
 printf("ok! success!\n");
 printf("%s",buf);
 free(buf);
 return 0;
}
```

该程序读取命令行参数，第一个参数是拷贝源文件的路径，第二个参数是拷贝目标文件的路径，并且参数会传递到系统调用当中使用。

（5） 进行测试

测试源文件 test.txt

```
Open    test.txt    Save
this is a test txt from ywl 2013774
this is OSLab10
may you good luck!

1. 尽管这是一部谈论跑步的书，却不是谈论健康方法的书。我并非要在这里高谈阔论、振臂一呼：“来呀!让我们每天跑步，永葆健康吧!”归根结底，这些都不过是思索片段，抑或自问自答—对我个人而言．坚持跑步究竟有何意味。仅此而已。

2. 萨默赛特·毛姆写道：“任何一把剃刀都自有其哲学。”大约是说．无论何等微不足道的举动．只要日日坚持．从中总会产生出某些类似观念的东西来。

3. Pain is inevitable. Suffering is optional.

4. 我下决心写一本关于跑步的书，说起来也是十多年前的事了。自那以后便苦苦思索，觉得这样不行那样也不成，始终不曾动笔，任烟花空散岁月空流。虽只是“跑步”一事，然而这个主题太过茫然，究竟该写什么，如何去写，思绪实在纷纭杂乱，无章无法。 然而有一次，我忽然想到，将自己感到的想到的，就这般原模原样、朴素自然地写成文章得了。恐怕舍此别无捷径。

5.现在是坚忍地累积奔跑距离的时期，所以眼下还不必介意成绩如何，只消默默地花上时间累积距
```

测试目标文件 copy.txt（暂时为空）

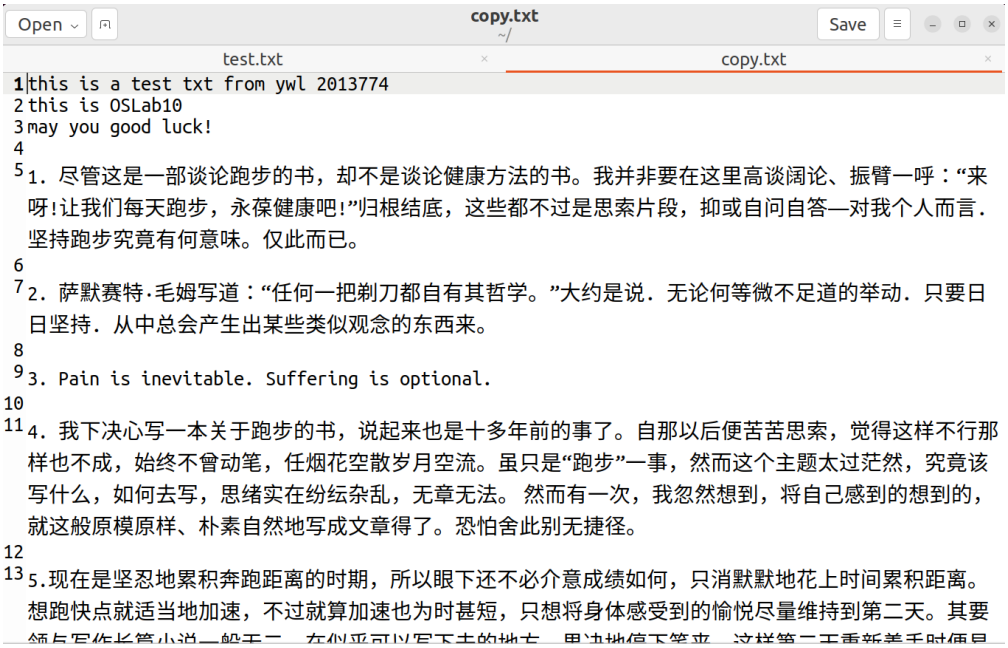
编译并执行测试程序

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab10_code$ ./test /home/wanliyang2013774/test.txt /home/wanliyang2013774/copy.txt
the program will copy /home/wanliyang2013774/test.txt to /home/wanliyang2013774/copy.txtok! success!
systemd                 1    S 2013774
kthreadd                2    S 2013774
rcu_gp                  3    I 2013774
rcu_par_gp              4    I 2013774
slub_flushwq            5    I 2013774
netns                   6    I 2013774
kworker/0:0             7    I 2013774
kworker/0:0H            8    I 2013774
kworker/u256:0          9    I 2013774
mm_percpu_wq           10    I 2013774
rcu_tasks_kthre        11    I 2013774
rcu_tasks_rude_        12    I 2013774
```

……

```
gsd-xsettings           2145    S 2013774
gjs                     2156    S 2013774
ibus-x11                2194    S 2013774
fwupd                   2245    S 2013774
gnome-terminal-         2680    S 2013774
bash                    2710    S 2013774
update-notifier         2796    S 2013774
gedit                   3084    S 2013774
test                    3117    R 2013774
the number of processes is 343 2013774
```

查看拷贝的目标文件，拷贝完成！



通过 diff 命令比较两文件异同

```
wanliyang2013774@wanliyang2013774-virtual-machine:~$ diff test.txt copy.txt
wanliyang2013774@wanliyang2013774-virtual-machine:~$
```

显然，两个文件完全相同

至此，实验成功！

6. 总结心得

（1） 通过本次实验加深了我对 Linux 系统调用的理解以及带参数的系统调用的编写和使用；

（2） 本次实验中拷贝文件的代码虽然简短，但是出错和调试的过程很漫长，因此也巩固了我关于文件读取和写入的知识；

（3） 在内核中拷贝文件让我对内核态、用户态以及内核空间和用户空间有了更多的认识，此前的 get_fs()等设置也是为了保障这个机制的运转，而现在弃用了这个方法显然是有更好的保护方式出现了。

（4） 通过本次实验意识到，内核中的系统调用也无法直接访问用户空间的内容，需要拷贝到内核空间才能正常使用。

7. 参考资料

实验指导书《os_lab_addnewsyscallargREADME.pdf》

实验指导书《Lab10aAddNewSyscallWithArgs.pdf》

Linux 系统调用手册

8. 实验源代码

（kernel/sys.c 文件中的添加内容）

```
1.  SYSCALL_DEFINE3(alcall,char*,buf,char*,source,char*,target)
2.  {
3.      int sum = 0;
4.      int value;
5.      int count = 0;
6.      struct task_struct *p;
7.      struct file* src_file = NULL;
```

```
8.      struct file* tgt_file = NULL;
9.      loff_t src_pos;
10.     loff_t tgt_pos;
11.     char buffer[128];
12.     char* BUF = kmalloc(sizeof(char)*18000,GFP_ATOMIC);
13.     char* src = kmalloc(sizeof(char)*30,GFP_ATOMIC);
14.     char* tgt = kmalloc(sizeof(char)*30,GFP_ATOMIC);
15.     char temp[45];
16.     int offset = 0;
17.     int i;
18.     p = &init_task;
19.     for_each_process(p)
20.     {
21.         memset(temp,'\0',45);
22.         sprintf(temp, "%-
    20s %6d %4c 2013774\n",p->comm,p->pid,task_state_to_char(p));
23.         i = 0;
24.         while(temp[i]!='\0'){
25.                 BUF[offset]=temp[i];
26.                 i++;
27.                 offset++;
28.         }
29.         sum += 1;
30.     }
31.     memset(temp,'\0',45);
32.     sprintf(temp, "the number of processes is %d 2013774\n", sum);
33.     i = 0;
34.     while(temp[i]!='\0'){
35.          BUF[offset]=temp[i];
36.          i++;
37.          offset++;
38.     }
39.     //进程信息存储完成
40.     //拷贝到用户空间
41.     value = copy_to_user(buf,BUF,offset);
42.     value = copy_from_user(src,source,30);
43.     value = copy_from_user(tgt,target,30);
44.     printk("%s to %s 2013774",src,tgt);
45.     src_file = filp_open(src, O_RDWR | O_APPEND | O_CREAT, 0644);
46.     tgt_file = filp_open(tgt, O_RDWR | O_APPEND | O_CREAT, 0644);
47.     if (IS_ERR(src_file)) {
48.         printk("fail to open file 2013774");
49.         return 0;
50.     }
```

```
51.     if (IS_ERR(tgt_file)) {
52.         printk("fail to open file 2013774");
53.         return 0;
54.     }
55.     src_pos = src_file->f_pos;
56.     tgt_pos = tgt_file->f_pos;
57.     while((count = kernel_read(src_file,buffer,128,&src_pos))>0)
58.     {
59.             kernel_write(tgt_file,buffer,count,&tgt_pos);
60.     }
61.     filp_close(src_file,NULL);
62.     filp_close(tgt_file,NULL);
63.     kfree(BUF);
64.     return 0;
65. }
```