操作系统第11次作业

1. 实验步骤

1.1 在sys.c里面编写线程查询函数和文件拷贝函数

```
SYSCALL_DEFINE3(copy,char*,buf,char*,source,char*,destination)
{

    char buffer[128];
    struct file*src_file =NULL;
    struct file*dest_file=NULL;
     src_file = filp_open(source, O_RDWR | O_APPEND | O_CREAT, 0644);
    dest_file = filp_open(destination, O_RDWR | O_APPEND | O_CREAT, 0644);

    if (IS_ERR(src_file)) {
        printk("fail to open file");
        return 0;
    }
    if (IS_ERR(dest_file)) {
        printk("fail to open file");
        return 0;
    }
    int len;
    loff_t src=src_file->f_pos;
    loff_t dest=dest_file->f_pos;
    while((len=kernel_read(src_file,buffer,128,&src))>0)
    {
    kernel_write(dest_file,buffer,len,&dest);
    }
    filp_close(src_file,NULL);
    filp_close(dest_file,NULL);


    return 0;
}
```

使用 `filp_open` 函数尝试打开源文件和目标文件。如果文件不存在，则创建它们。 `O_RDWR` 表示读写模式， `O_APPEND` 表示追加数据， `O_CREAT` 表示如果文件不存在则创建， `0644` 是文件权限。接着通过 `while` 循环，使用 `kernel_read` 从源文件中读取数据到 `buffer` ，然后使用 `kernel_write` 将 `buffer` 中的数据写入目标文件。从而实现文件的拷贝。

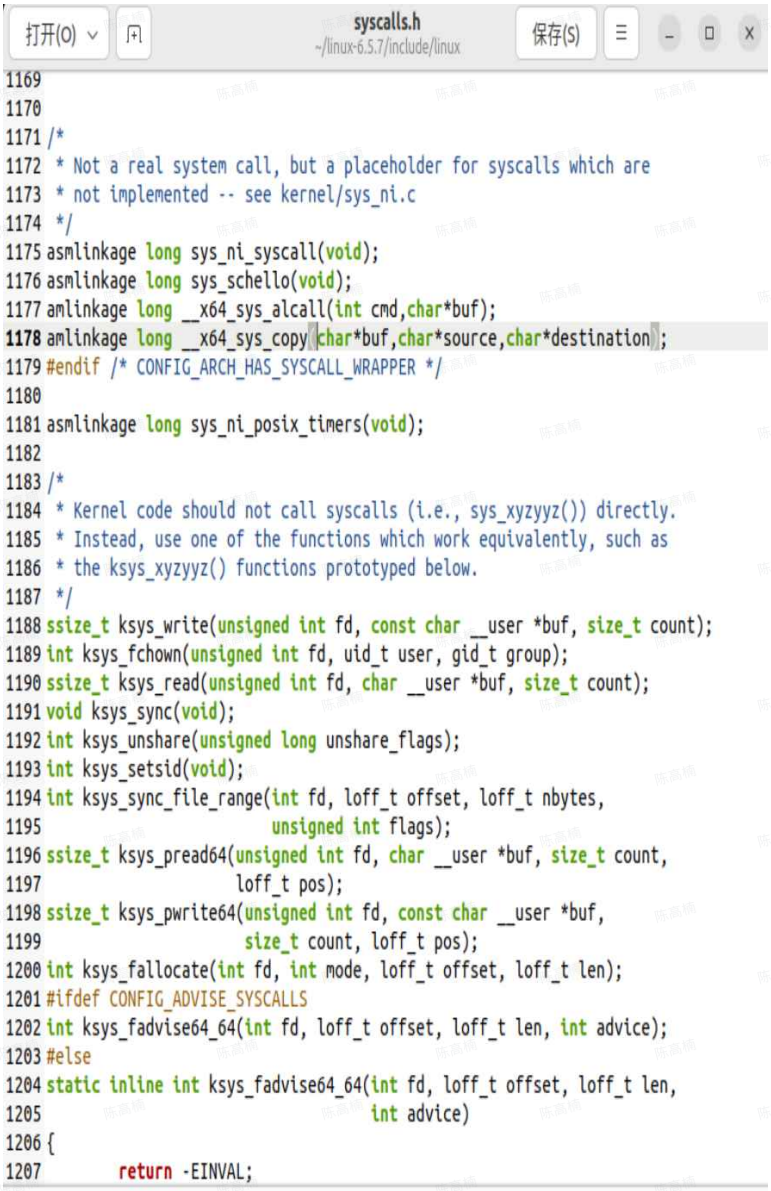
```
SYSCALL_DEFINE2(alcall,int,cmd,char*,buf)
{
int sum=0;
struct task_struct *p;
printk("Hello new system call alcall!  2112966陈高楠\n");
printk("Hello new system call alcall (%d,%x)!\n",cmd,buf);
printk("%-20s %-6s  %-6s\n","Name","Pid","Stat");
for (p = &init_task; (p = next_task(p)) != &init_task;)
{sum+=1;
printk("%-20s  %-6d    %-6ld\n",p->comm,p->pid,p->__state);
}
printk("the number of the process is %d",sum);
return 0;
}
```

这个函数通过for循环遍历进程并输出，统计进程数并输出。

1.2 在syscalls.h里面定义函数接口

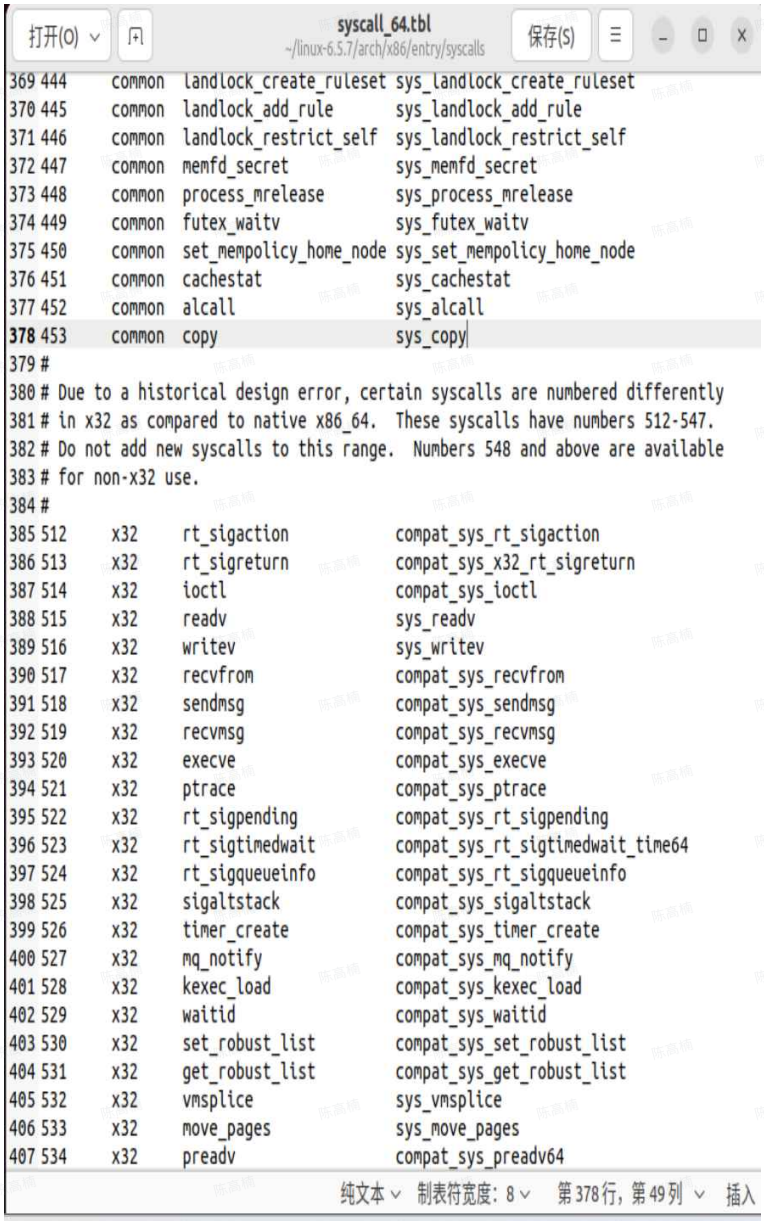
写入asmlinkage long __x64_sys_alcall(int cmd,char*buf),

asmlinkage long __x64_sys_copy(char*buf,char*source,char*destination)。




```
1169
1170
1171 /*
1172  * Not a real system call, but a placeholder for syscalls which are
1173  * not implemented -- see kernel/sys_ni.c
1174  */
1175 asmlinkage long sys_ni_syscall(void);
1176 asmlinkage long sys_schello(void);
1177 amlinkage long __x64_sys_alcall(int cmd,char*buf);
1178 amlinkage long __x64_sys_copy(char*buf,char*source,char*destination);
1179 #endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
1180
1181 asmlinkage long sys_ni_posix_timers(void);
1182
1183 /*
1184  * Kernel code should not call syscalls (i.e., sys_xyzyyz()) directly.
1185  * Instead, use one of the functions which work equivalently, such as
1186  * the ksys_xyzyyz() functions prototyped below.
1187  */
1188 ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count);
1189 int ksys_fchown(unsigned int fd, uid_t user, gid_t group);
1190 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count);
1191 void ksys_sync(void);
1192 int ksys_unshare(unsigned long unshare_flags);
1193 int ksys_setsid(void);
1194 int ksys_sync_file_range(int fd, loff_t offset, loff_t nbytes,
1195                          unsigned int flags);
1196 ssize_t ksys_pread64(unsigned int fd, char __user *buf, size_t count,
1197                      loff_t pos);
1198 ssize_t ksys_pwrite64(unsigned int fd, const char __user *buf,
1199                       size_t count, loff_t pos);
1200 int ksys_fallocate(int fd, int mode, loff_t offset, loff_t len);
1201 #ifdef CONFIG_ADVISE_SYSCALLS
1202 int ksys_fadvise64_64(int fd, loff_t offset, loff_t len, int advice);
1203 #else
1204 static inline int ksys_fadvise64_64(int fd, loff_t offset, loff_t len,
1205                                     int advice)
1206 {
1207         return -EINVAL;
```

1.3 在syscall_64.tbl里面写入



1.4 编写测试文件，编译后进行测试

首先使用make命令进行重新编译

cd /usr/src/linux进入目录，之后make clean，然后make -j5。

编译完成后输入sudo make modules_install,sudo make install，之后输入reboot重启。

接着编写测试文件。

这是测试拷贝函数的c文件

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include<stdlib.h>
#define __NR_copy 453
long mycopy(char*buf,char*source,char*destination){
return syscall(__NR_copy,buf,source,destination);
```

```c
}
int main(int argc, char *argv[])
{

char *source=malloc(sizeof(char)*30);
char*destination=malloc(sizeof(char)*30);
source=argv[1];
destination=argv[2];
char*buf=malloc(sizeof(char)*16000);
printf("copy %s to %s",source,destination);
mycopy(buf,source,destination);
printf("  ok!  \n");
free(buf);
return 0;
}

```

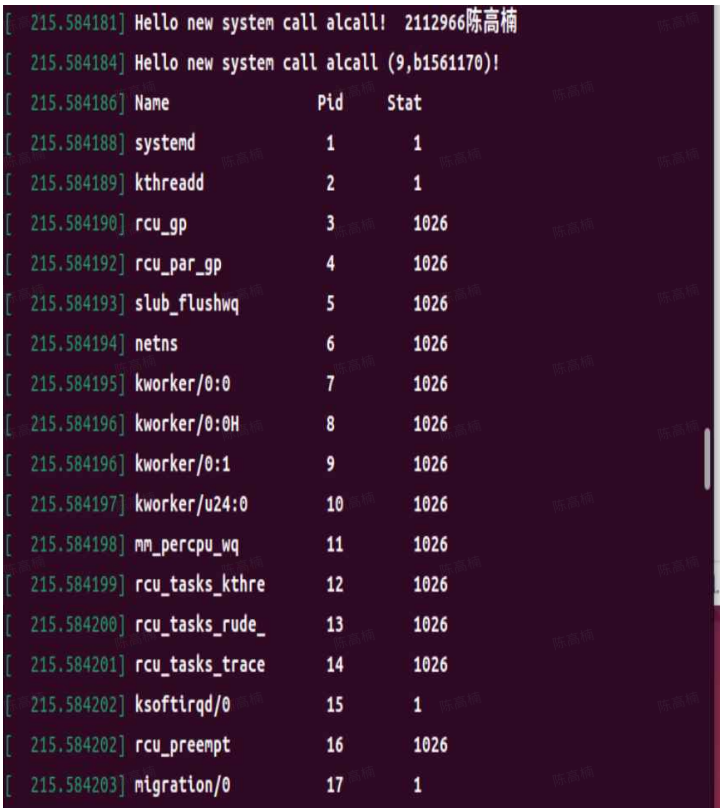
这是线程查询的c文件

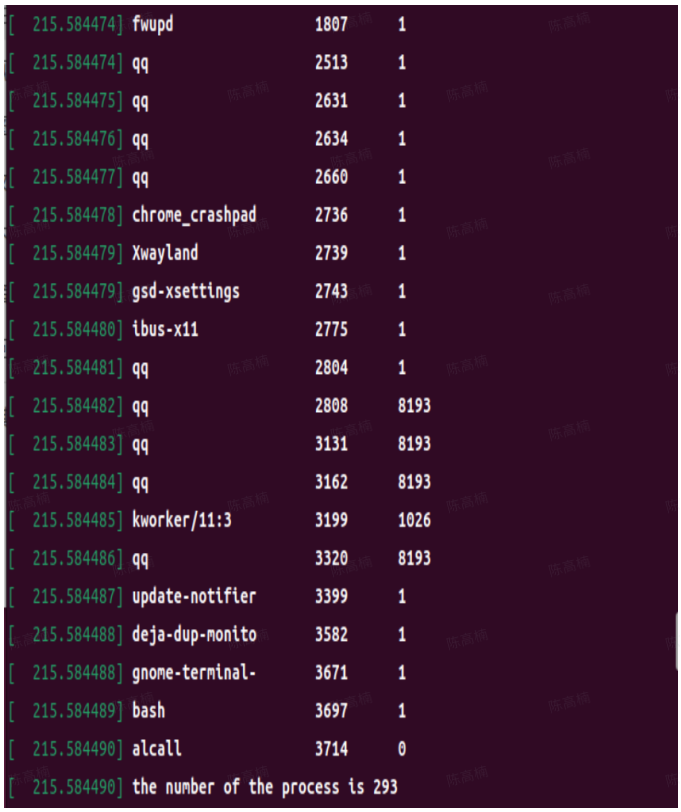
```c
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#define __NR_alcall 452
long alcall(int cmd,char*buf){
return syscall(__NR_alcall,cmd,buf);

}
int main(int argc, char *argv[])
{ int cmd;
char buf[256];
cmd=9;
alcall(cmd,buf);
printf("ok! run dmesg|grep alcall in terminal\n");
return 0;
}

```

2. 实验结果

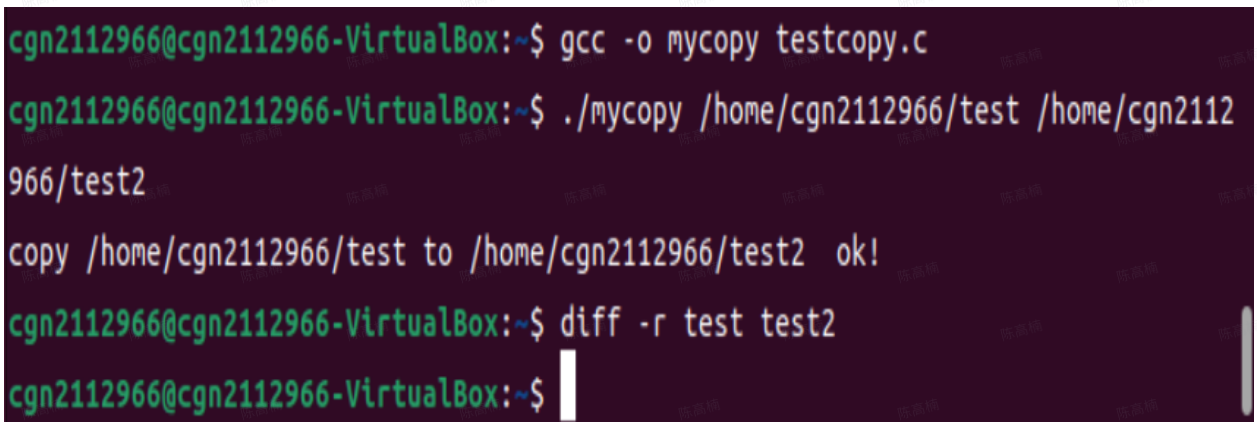
在命令行执行测试文件效果如下：

查看线程以及线程数：





拷贝文件并检查是否成功拷贝



由结果可得实验结果成功，可以显示线程以及线程数目，也可以拷贝文件。

3. 实验心得

在本次操作系统实验中，我探索了Linux内核，特别是系统调用的实现和扩展。在这一过程中，我增强了对操作系统核心机制的理解。我通过实现文件拷贝函数，我学习了如何在内核空间打开、读取、写入和关闭文件。这加深了我对操作系统核心概念的理解。