向 linux 内核添加系统调用实现文件夹拷贝

学院：软件学院

学号：2212195

姓名：　乔昊

实验目标

1. **向 linux 内核中添加一个新的系统调用**
2. **测试用户模式下新的系统调用**

实验准备

安装开发工具

本次实验选用 GCC 编译工具

安装 GCC

```
sudo apt-get install build-essential
```

选择开发环境

vscode 编辑器

准备 linux 内核

本次实验使用 linux-6.10.10 版本的 linux 内核

查看系统内核

```
uname -a
```

```
qh2212195@rika-VM:~$ uname -a
Linux rika-VM 6.10.102212195 #3 SMP PREEMPT_DYNAMIC Fri Dec  6 01:18:20 CST 2024 x86_64 x86_64 x86_64 GNU/Linux
qh2212195@rika-VM:~$
```

实验过程

添加 asmlinkage 宏定义

在 /usr/src/linux/include/linux 目录下

1. 进入 **syscalls.h** 文件
2. 找到 **#endif /_ CONFIG_ARCH_HAS_SYSCALL_WRAPPER _/**
3. 添加 **asmlinkage long sys_alcall(char* buf, char* source, char* target);**

```
1192  asmlinkage long sys_ni_syscall(void);
1193  asmlinkage long sys_schello(void);
1194  asmlinkage long sys_alcall(char* buf, char* source, char* target);
```

实现 SYSCALL_DEFINE3(alcall,char* buf, char* source, char* target) 函数

在 /usr/src/linux/kernel 目录下

1. 进入 **sys.c** 文件
2. 找到 **SYSCALL_DEFINE0(gettid)** 函数
3. 添加 **SYSCALL_DEFINE3(alcall,char* buf, char* source, char* target)** 函数

以下是实现列举进程信息的具体代码

```
 998  SYSCALL_DEFINE3(alcall, char *, buf, char *, source, char *, target)
 999  {
1000      int sum = 0;
1001      int value;
1002      int count = 0;
1003      struct task_struct *p;
1004      struct file *src_file = NULL;
1005      struct file *tgt_file = NULL;
1006      loff_t src_pos;
1007      loff_t tgt_pos;
1008      char buffer[128];
1009      char *BUF = kmalloc(sizeof(char) * 18000, GFP_ATOMIC);
1010      char *src = kmalloc(sizeof(char) * 30, GFP_ATOMIC);
1011      char *tgt = kmalloc(sizeof(char) * 30, GFP_ATOMIC);
1012      char temp[45];
1013      int offset = 0;
1014      int i;
1015      p = &init_task;
1016      printk("Hello new system call alcall! 乔昊2212195\n");
```

```
1016    printk("Hello new system call alcall! 乔昊2212195\n");
1017    for_each_process(p)
1018    {
1019        memset(temp, '\0', 45);
1020        sprintf(temp, "%-20s %6d %4c \n", p->comm, p->pid, task_state_to_char(p));
1021        i = 0;
1022        while (temp[i] != '\0')
1023        {
1024            BUF[offset] = temp[i];
1025            i++;
1026            offset++;
1027        }
1028        sum += 1;
1029    }
1030    memset(temp, '\0', 45);
1031    sprintf(temp, "the number of processes is %d \n", sum);
1032    i = 0;
1033    while (temp[i] != '\0')
1034    {
1035        BUF[offset] = temp[i];
1036        i++;
1037        offset++;
1038    }
1039    value = copy_to_user(buf, BUF, offset);
1040    value = copy_from_user(src, source, 30);
1041    value = copy_from_user(tgt, target, 30);
1042    printk("%s to %s ", src, tgt);
1043    src_file = filp_open(src, O_RDWR | O_APPEND | O_CREAT, 0644);
1044    tgt_file = filp_open(tgt, O_RDWR | O_APPEND | O_CREAT, 0644);
1045    if (IS_ERR(src_file))
1046    {
1047        printk("fail to open file ");
1048        return 0;
1049    }
1050    if (IS_ERR(tgt_file))
1051    {
1052        printk("fail to open file ");
1053        return 0;
1054    }
1055    src_pos = src_file->f_pos;
1056    tgt_pos = tgt_file->f_pos;
1057    while ((count = kernel_read(src_file, buffer, 128, &src_pos)) > 0)
1058    {
1059        kernel_write(tgt_file, buffer, count, &tgt_pos);
1060    }
1061    filp_close(src_file, NULL);
1062    filp_close(tgt_file, NULL);
1063    kfree(BUF);
1064    return 0;
1065 }
```

代码分析

1. 分别从内核空间向用户空间拷贝数据以及从用户空间向内核空间拷贝数据
2. 使用file_open函数打开两个文件
3. 文件打开成功则获取文件的初始偏移量,循环使用kernel_read从源文件读取数据到buffer
4. 同时使用kernel_write将读取到的数据写入目标文件中

添加 common schello sys_schello 命令

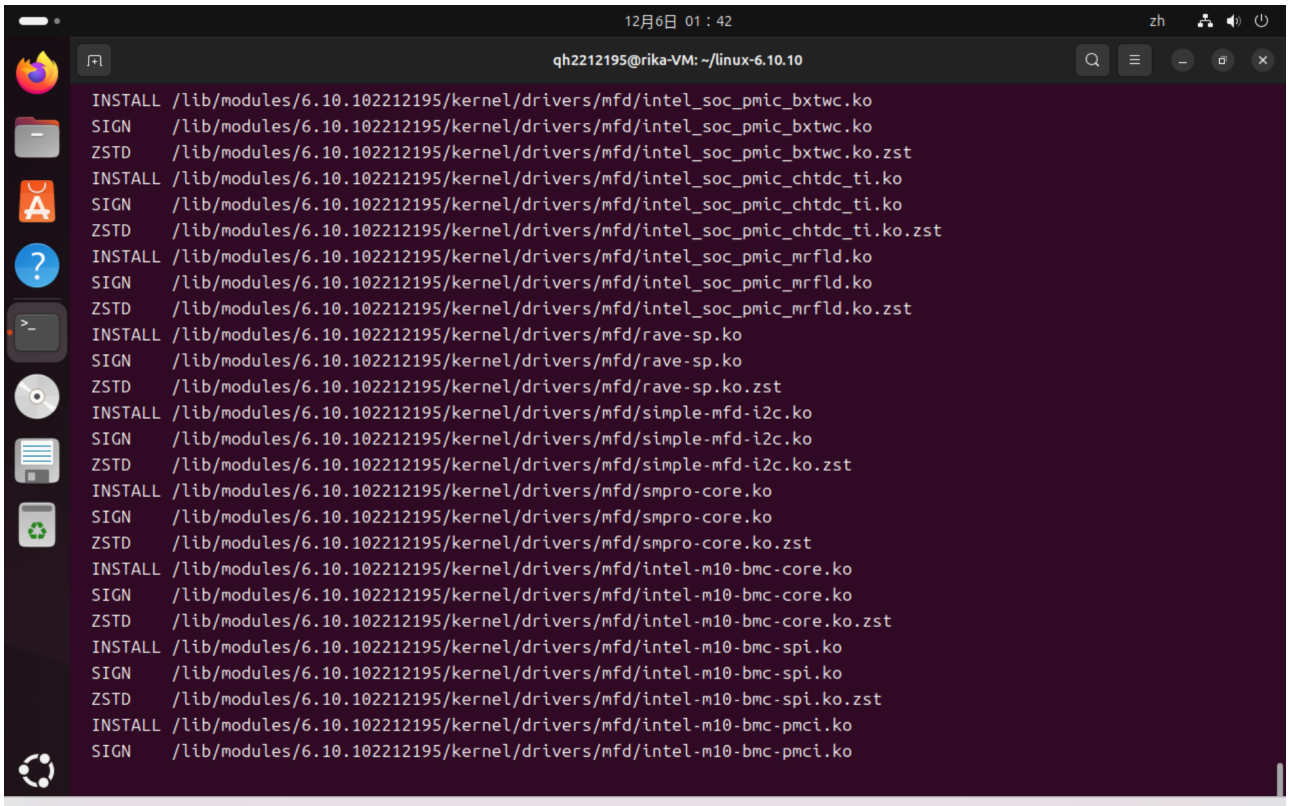
在 /usr/src/linux/arch/x86/entry/syscalls 目录下

1. 进入 **syscall_64.tbl**文件
2. 添加 **336 common schello __x64_sys_alcall**命令

```
345  334 common  rseq            sys_rseq
346  335 common  schello         sys_schello
347  336 common  alcall          __x64_sys_alcall
```

重新编译内核

1. **清理项目**`make clean`
2. **编译内核**`make -j5`
3. **拷贝编译模块**`sudo make modules_install`
4. **安装内核映像**`sudo make install`
5. **重新启动**`reboot`



查看系统内核

执行`uname -a`命令

```
qh2212195@rika-VM:~$ uname -a
Linux rika-VM 6.10.102212195 #3 SMP PREEMPT_DYNAMIC Fri Dec  6 01:18:20 CST 2024 x86_64 x86_64 x86_64 GNU/Linux
qh2212195@rika-VM:~$
```

编写用户态测试程序

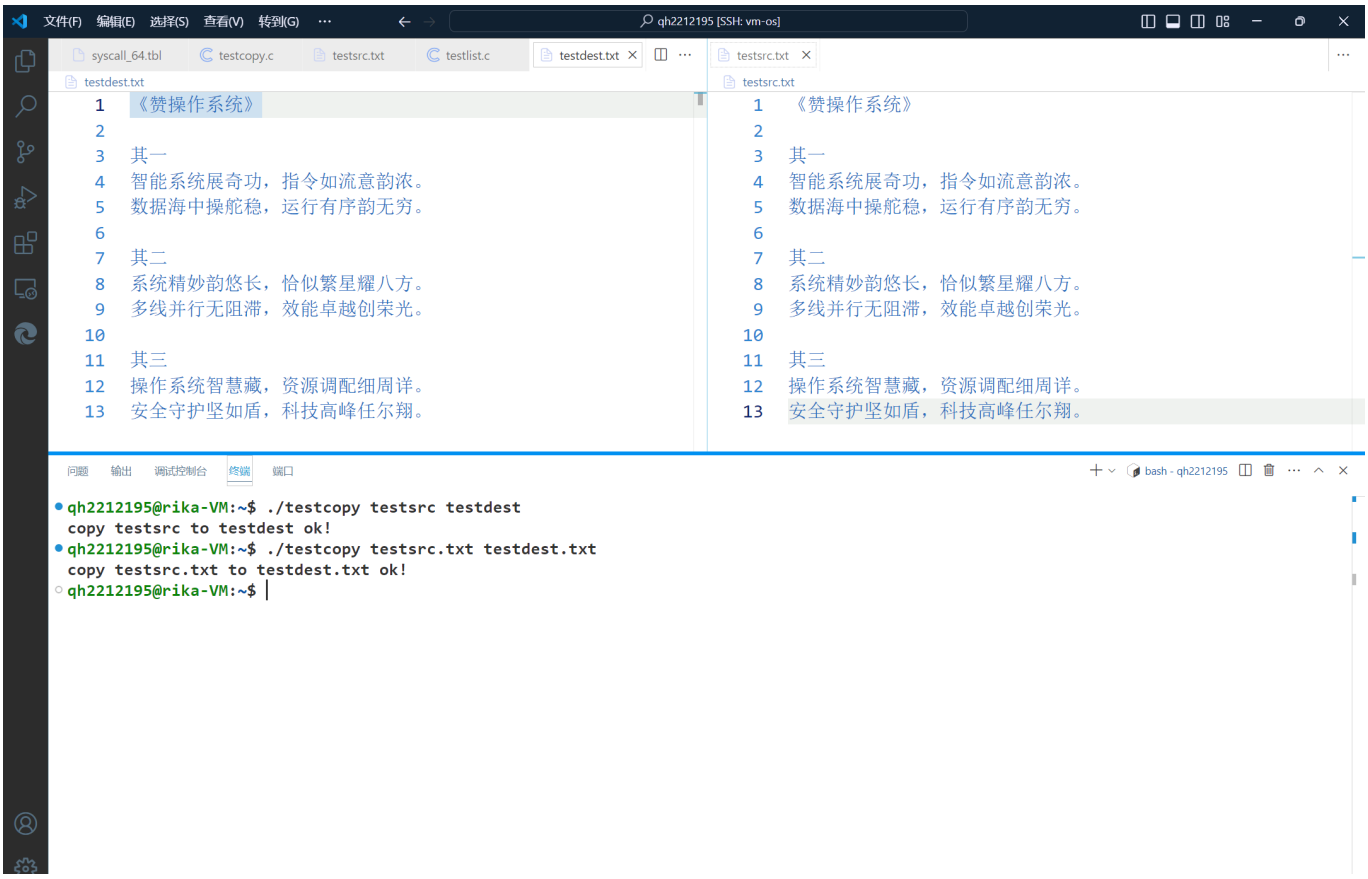
测试代码

```
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define __NR_alcall 336
long alcall(char *buf, char *source, char *destination)
{
    return syscall(__NR_alcall, buf, source, destination);
}
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);
    alcall(buf, source, destination);
    printf(" ok! \n");
    free(buf);
    return 0;
}
```

执行结果



实验总结

本次实验着重于在 Linux 内核中添加系统调用以实现文件拷贝功能。首先在 /usr/src/linux/include/linux 的 syscalls.h 文件中处理相关声明（此次无需重复添加 asmlinkage 宏定义）。核心步骤为在 /usr/src/linux/kernel 的 sys.c 文件里实现 SYSCALL_DEFINE3 (alcall, char* buf, char* source, char* target) 函数。此函数先从内核空间向

用户空间拷贝数据以及反向拷贝，接着运用 file_open 函数打开源文件与目标文件，若文件打开成功，则获取其初始偏移量，通过循环使用 kernel_read 从源文件读取数据至 buffer，同时借助 kernel_write 将数据写入目标文件，以此达成文件拷贝目的。之后在 /usr/src/linux/arch/x86/entry/syscalls 的 syscall_64.tbl 文件中添加注册命令。最后执行重新编译内核操作，涵盖清理项目、编译内核、安装模块、安装内核映像以及重启系统等一系列步骤，确保新系统调用能正常运作。

本次实验,我掌握了在 sys.c 文件中编写代码实现文件拷贝功能，熟练运用内核文件操作函数如 file_open、kernel_read 和 kernel_write 等，深入理解其参数含义与使用场景。同时，完整地熟悉了内核编译与安装流程，知晓每个步骤的作用与可能出现的问题及解决方法。在编程思维与能力提升维度，进一步强化了 C 语言在内核环境下的运用能力，包括对指针、缓冲区操作以及函数返回值处理的精准把握。

本次实验开启了对 Linux 内核文件处理领域深入探索的大门，激发了我对内核其他文件相关功能开发的浓厚兴趣。深刻体会到系统级开发中对系统架构深入理解的极端重要性，在文件拷贝这一相对基础的功能开发中，就涉及到内核多模块、多函数的协同工作，任何对架构理解的偏差都可能导致功能无法实现或系统不稳定。这也促使我在今后的开发过程中必须始终保持严谨的编程态度，对每一个代码细节都要精益求精，确保代码的准确性与可靠性。