操作系统课程设计实验报告

实验题目：Linux 内核编译及添加系统调用

姓　　名：王翔宇

学　　号：22200215

组　　号：

专　　业：网络空间安全

班　　级：22270311

老师姓名：张祯

日　期：2024　年　4　月　18　日

(大家注意，目录是自动生成的，页码从正文部分开始，当同学们把正文写完后，只需要右击目录，选择更新域，目录会自动更新)

一 题目介绍

本实验通过修改 Linux 内核源码，添加新的 Linux 系统调用，替换编译后内核，并测试结果，了解 Linux 内核源码的编译方法和内核的安装方法，系统调用的概念、编写步骤和调用方法。

具体实验描述如下：

1. 掌握 Linux 系统调用基本概念
2. Linux 内核源码的编译和安装
3. 添加 Linux 的系统调用
4. Linux 的系统调用的测试方法

二 实验思路

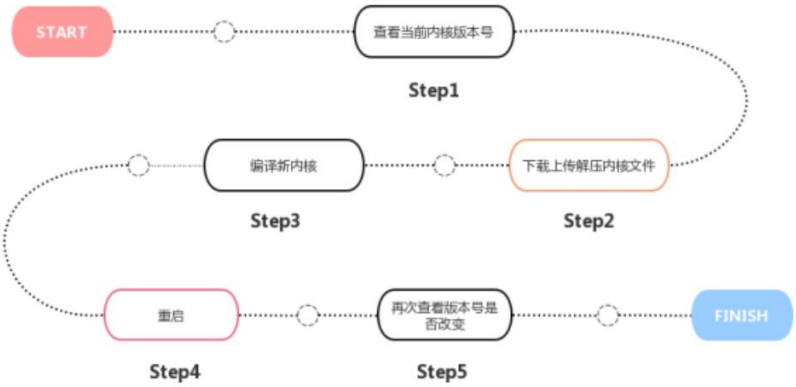

图 1：编译内核流程图

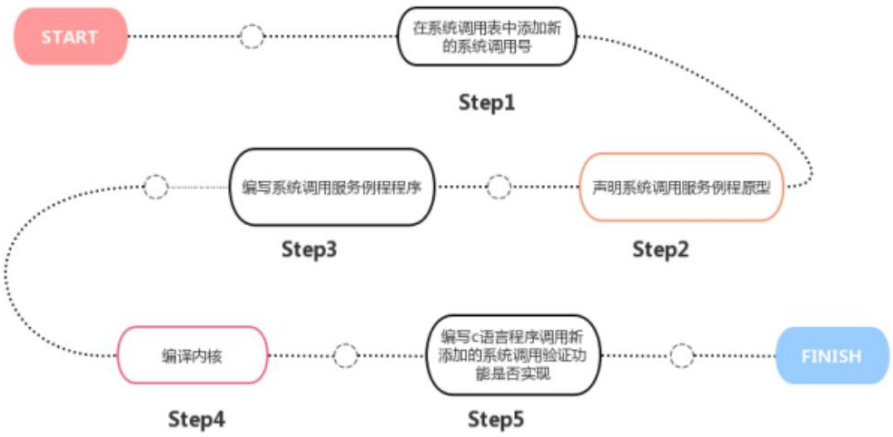

图 2：添加系统调用流程图

内核代码编写实验思路：
修改进程 nice 值：

代码设计思路

定义系统调用：

使用 SYSCALL_DEFINE5 宏定义系统调用，传入五个参数：进程 ID（pid）、标志位（flag）、新的 nice 值（nicevalue）、用于存储当前 nice 值的用户空间指针（nice_ptr）和用于存储当前优先级（prio）的用户空间指针（prio_ptr）。

参数校验：

检查传入的进程 ID（pid）是否有效。

检查用户空间指针（nice_ptr 和 prio_ptr）是否有效，以确保后续可以安全地复制数据到用户空间。

查找进程：

使用内核函数（如 find_get_pid）根据进程 ID 找到对应的 struct pid 结构体。

使用 pid_task 函数从 struct pid 中获取对应的 task_struct。

获取和修改 nice 值：

如果进程存在，根据标志位（flag）决定是仅获取 nice 值和优先级，还是修改 nice 值。

使用 task_nice 和 task_prio 函数获取当前进程的 nice 值和优先级。

如果需要修改 nice 值，使用 set_user_nice 函数进行修改，并重新获取修改后的 nice 值和优先级。

复制数据到用户空间：

使用 copy_to_user 函数将 nice 值和优先级复制到用户空间提供的指针所指向的位置。

错误处理：

如果在查找进程、修改 nice 值或复制数据时发生错误，返回适当的错误码（如 EFAULT）。

修改主机名：

设计思路

参数定义：

系统调用接收一个指向 struct new_utsname 结构的用户空间指针 name 作为参数。这个结构用于存储系统的 UTS 名称信息，如主机名、内核版本等。

保护 UTS 数据：

使用 down_read(&uts_sem);和 up_read(&uts_sem);来确保在读取 UTS 数据时，其他线程或进程不会同时修改它，保证数据的完整性和一致性。这里使用了读写信号量（semaphore）来保护 UTS 数据结构。

复制 UTS 数据：

使用 memcpy(&tmp, utsname(), sizeof(tmp));将内核中的 UTS 数据（通过 utsname()函数获取）复制到本地临时变量 tmp 中。这样做是为了避免在将数据复制到用户空间时，其他线程或进程修改了 UTS 数据。

将数据复制到用户空间：

使用 copy_to_user(name, &tmp, sizeof(tmp));将临时变量 tmp 中的数据复制到用户空间提供的 name 指针所指向的位置。这是系统调用中常见的步骤，用于将内核空间的数据传递给用户空间。

可选的覆盖功能：

代码中包含两个可选的覆盖函数 override_release 和 override_architecture，它们分别用于覆盖 UTS 信息中的 release（版本信息）和可能的 architecture（体系结构信息）。这两个函数可能是用于特定目的，如隐藏真实的内核版本或修改体系结构信息等。如果这两个函数执行失败（例如，如果 name->release 指向的内存区域不可写），则返回-EFAULT 错误。

错误处理：

如果在复制数据到用户空间或执行覆盖函数时发生错误（如用户空间指针无效或不可写），则返回-EFAULT 错误。这是 Linux 系统调用中常见的错误码，用于指示用户空间参数无效或不可访问。

返回结果：

如果所有操作都成功完成，则返回 0 表示成功。

三 遇到问题及解决方法

1. 内核编译时出错，缺少相关下载包
2. 安装内核时更改了 CPU 核数，加快了编译速度
3. 添加系统调用时，对内核函数实现代码的编写出错几次，最终改写正确

四 核心代码及实验结果展示

1.添加系统调用号

```
[root@ecs-os kernel-4.19.90]# vim include/uapi/asm-generic/unistd.h
```

```
#define __NR_io_pgetevents 292
__SC_COMP(__NR_io_pgetevents, sys_io_pgetevents, compat_sys_io_pgetevents)
#define __NR_rseq 293
__SYSCALL(__NR_rseq, sys_rseq)
#define __NR_czq 294
__SYSCALL(__NR_czq,sys_czq)
#define __NR_mysetnice 295
__SYSCALL(__NR_mysetnice,sys_mysetnice)
#define __NR_mysethostname 296
__SYSCALL(__NR_mysethostname,sys_mysethostname)
#undef __NR_syscalls
#define __NR_syscalls 297
```

2.添加声明系统调用的函数

```
[root@ecs-os kernel-4.19.90]# vim include/linux/syscalls.h
```

```
/*tianjia*/
asmlinkage long sys_zlk(void);
asmlinkage long sys_mysetnice(pid_t pid,int flag,int nicevalue,void __user*prio,void __user*nice);
asmlinkage long sys_mysethostname(char __user  *name, int len);
```

4. 添加系统调用实现代码

```
[root@ecs-os kernel-4.19.90]# vim kernel/sys.c
```

```
/*tianjia*/
SYSCALL_DEFINE0(czq)
{
        printk(KERN_INFO "student number is : 22200215");//前面的参数一定要添加，
        return 0;
}
```

```
SYSCALL_DEFINE5(mysetnice,pid_t,pid,int,flag,int,nicevalue,void __user*,prio,void __user*,nice)
{
        int n;
        int p;
        struct pid * kpid;
        struct task_struct * task;
        kpid = find_get_pid(pid);/*得到pid */
        task = pid_task(kpid, PIDTYPE_PID);/* 返回task_struct */
        n = task_nice(task);/* 返回进程当前nice值 */
        p = task_prio(task);/*返回进程当前prio值*/
        if(flag == 1)
        {
                set_user_nice(task, nicevalue);/* 修改进程nice值 */
                n = task_nice(task);/*重新取得进程nice值*/
                p = task_prio(task);/*重新取得进程prio值*/
                copy_to_user(nice,&n,sizeof(n));/*将nice值拷贝到用户空间*/
                copy_to_user(prio,&p,sizeof(p));/*将prio值拷贝到用户空间*/
                return 0;
        }
        else if(flag == 0)
        {
                copy_to_user(nice,&n,sizeof(n));/*将nice值拷贝到用户空间*/
                copy_to_user(prio,&p,sizeof(p));/*将prio值拷贝到用户空间*/
                return 0;
        }
        return EFAULT;
}
```

```c
SYSCALL_DEFINE2(mysethostname, char __user *, name, int, len)
{
    int errno;
    char tmp[__NEW_UTS_LEN];
    if(len<0 || len>__NEW_UTS_LEN)
        return -EINVAL;
                errno = -EFAULT;
    if(!copy_from_user(tmp, name, len))
        {
                struct new_utsname *u;
                down_write(&uts_sem);
                u = utsname();
                memcpy(u->nodename, tmp, len);
                memset(u->nodename + len, 0, sizeof(u->nodename)- len);
                errno = 0;
                uts_proc_notify(UTS_PROC_HOSTNAME);
                up_write(&uts_sem);
        }
        return errno;
}
```

5. 内核编译安装

(5) 编译内核

[root@openEuler ~]# cd kernel

[root@openEuler kernel]# make openeuler_defconfig

在这里，我们按源代码文件kernel/arch/arm64/configs/openeuler_defconfig的配置配置内核，此外，建议大家可以试试make menuconfig，看一下编译内核有哪些可配置项（https://blog.csdn.net/howiexue/article/details/76696631）。

[root@openEuler kernel]# make help | grep Image

* Image.gz - Compressed kernel image (arch/arm64/boot/Image.gz)

Image - Uncompressed kernel image (arch/arm64/boot/Image)

这一步查看了可编译的Image。

[root@openEuler kernel]# make -j 4 Image modules dtbs

这一步是编译内核的Image、modules和dtbs，make -j 4表示4个线程编译（可以根据CPU核数调整）

(6) 安装内核

[root@openEuler kernel]# make modules_install

```
[root@ecs-os ~]# uname -a
Linux ecs-os 4.19.90 #2 SMP Thu Apr 18 11:48:31 CST 2024 aarch64 aarch64 aarch64 GNU/Linux
[root@ecs-os ~]#
```

6. 编写并添加调用函数

（1） 输出学号

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/syscall.h>
int main()
{
        if(syscall(294)==0)
                printf("success!\n");
        else
                printf("fail!\n");
        return 0;
}
~
~
```

（2） 修改进程优先级

```c
#define _GNU_SOURCE
#include<unistd.h>
#include<sys/syscall.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
        pid_t pid;
        int nicevalue;
        int flag;
        int n=0;
        int p=0;
        int *prio;
        int *nice;
        prio = &p;
        nice = &n;
        printf("请输入pid: \n");
        scanf("%d",&pid);
        printf("pid输入成功\n请输入nice值:\n");
        scanf("%d",&nicevalue);
        printf("nice输入成功\n请输入flag(flag为1时修改,为0时查看):\n");
        scanf("%d",&flag);
        syscall(295,pid,flag,nicevalue,prio,nice);
        printf("现在的nice为%d,prio为%d\n",n,p);
        return 0;
}

~
~
```

（3） 修改主机名为自定义字符串

```c
#define _GUN_SOURCE
#include<unistd.h>
#include<sys/syscall.h>
#include<stdio.h>
int main()
{
        syscall(296,"haha",4);
        return 0;
}
```

实验结果：

五 个人实验改进与总结

5.1 个人实验改进

在进行该实验时：

1. 进行了 boot 目录的备份，以防后续更新内核失败
2. 获取 openEuler 内核源码时登录相关网站进行相关代码的下载并上传到云服务器
3. 编译内核时切换 CPU 核数，加快编译速度
4. 添加系统调用函数时请教了别的同学，并且参考了网络上的源代码

5.2 个人实验总结

本次实验旨在通过修改 Linux 内核源码，添加新的系统调用，并替换编译后的内核以测试新功能。通过这一过程，我深入了解了 Linux 内核源码的编译方法、内核的安装方法，以及系统调用的概念、编写步骤和调用方法。

首先，我学习了如何获取和配置 Linux 内核源码。这包括了从官方网站下载源码包、解压源码、配置编译选项等步骤。在配置编译选项时，我根据自己的需求和硬件环境进行了适当的调整，以确保编译出的内核能够在我的系统中正常运行。

接下来，我开始了系统调用的编写工作。系统调用是用户空间程序与内核空间交互的接口，通过它，用户空间程序可以请求内核执行一些特权操作。我仔细阅读了内核文档和相关资料，了解了系统调用的编写规范和步骤。然后，我根据自己的需求编写了一个新的系统调用，并在内核源码中进行了相应的修改。

完成系统调用的编写后，我开始了内核的编译工作。编译内核是一个复杂的过程，需要耐心和细心。我按照编译指南的步骤，依次执行了清理、配置、编译和安装等命令。在编译过程中，我遇到了一些问题，如依赖缺失、编译错误等，但通过查阅资料和不断尝试，我最终成功编译出了新的内核。

最后，我替换了系统中的旧内核，并重启了计算机。重启后，我编写了一个简单的测试程序来调用我添加的新系统调用。通过测试，我发现新系统调用能够正常工作，达到了预期的效果。

通过这次实验，我深刻体会到了 Linux 内核的复杂性和强大性。同时，我也学到了很多关于内核开发和系统调用的知识，这将对我未来的学习和工作产生积极的影响。总的来说，这是一次非常有意义和收获丰富的实验。

六 参考文献

1.Linux 内核编译及添加系统调用_向 openeuler 系统增加一个系统调用模块-CSDN 博客

2.第四十三期-向 openEuler 内核中增加一个系统调用 - 知乎 (zhihu.com)