

注：实验一需要的环境，及实验前准备工作

1) 开机选择 win10_3 操作系统，并连接网络。实验过程需要下载相应的依赖库文件。

2) 在桌面上打开 VMware 虚拟机，使用其中的 Ubuntu 18.04 系统。本课程的所有实验均在 VMware 虚拟机上完成。

虚拟机登录密码：admin。

root 用户的密码：root。

2*)若在自己的电脑上进行试验,请安装 VMware 或 VritualBox 虚拟机管理器,推荐使用 Server 版系统减少硬盘占用和安装时间:

第一步:

在 <https://ubuntu.com/download/server> 找到 Alternative downloads 并选择下载 Ubuntu Server 18.04 LTS

若希望使用图形界面系统,请在 <https://ubuntu.com/download/desktop> 下载。**注意:** Desktop 版系统的内核版本高于本实验将要编译的 4.20.13 版本,在安装内核时必须执行额外的步骤切换内核。

推荐为虚拟机分配资源:

2G以上内存

35G以上硬盘容量

可防止启动新内核时出现 kernel panic – not syncing: system is deadlocked 报错,或编译内核时没有足够的硬盘容量存放编译生成的文件。

第二步:

虚拟机第一次启动时选择启动盘/镜像,请选择刚才下载的.iso系统镜像文件。推荐在安装系统的过程中断开网络,防止因下载更新产生额外流量。

Server 版安装时推荐勾选 **Install OpenSSH server** 方便使用 PuTTY 和 WinSCP 等软件传输命令和文件。

Desktop 版安装时推荐在 **更新和其他软件** 页面选择最小安装，并取消勾选 **其他选项** 下的 **安装Ubuntu时下载更新** 和 **安装第三方软件** 两个选项。

其他安装选项选择默认即可。

3) 从教育在线上下载内核源代码 **linux-4.20.13.tar.xz** 和 镜像文件 **sources.list**

4) 为便于查看，指导书上提供的命令，空格间距大，直接复制会出现错误。请手动输入命令。（命令可使用 Tab键 补全）。

5) 打开终端，切换到 root 用户

命令为sudo su 密码: root （或安装系统时自己设定的密码）

6) 若安装依赖库时出现下载速度慢、无法连接等问题，可修改 Ubuntu 下载镜像，改为国内的镜像，如中科大镜像站 (<https://mirrors.ustc.edu.cn>) 清华开源软件镜像站 (<https://mirrors.tuna.tsinghua.edu.cn>)。

使用教育在线提供的 **sources.list** 文件替换源的方法：

第一步，将原来的镜像文件备份：

```
cd /etc/apt/
```

```
mv sources.list sources.list.bak
```

第二步，把新的镜像文件拷贝到 /etc/apt 路径下

```
cp -ri 保存路径/sources.list /etc/apt
```

第三步，运行 **apt-get update**

7) 键入以下命令，安装相关编译支持库。

```
apt install libncurses5-dev libssl-dev
```

```
apt install build-essential openssl
```

```
apt install zlibc minizip
```

```
apt install libidn11-dev libidn11
```

```
apt install bison
```

```
apt install flex
```

8) 安装vim

```
apt-get install vim vim
```

实验报告按照如下内容编写：

实验目的
功能要求
主要功能设计说明
主程序函数与参数说明
程序框图
程序设计实现说明
测试结果与说明

实验一 添加一个新的系统调用

一、 实验目的

理解操作系统内核与应用程序的接口关系；加深对内核空间和用户空间的理解；学会增加新的系统调用。

二、 实验内容与要求

首先增加一个系统调用函数，然后连接新的系统调用，重建新的Linux内核（4.20.13），用新的内核启动系统，使用新的系统调用

三、 实验步骤

查看当前内核版本，可键入如下命令：

```
cat /proc/version
```

```
uname -a
```

1.获得源代码

本次实验的内核版本是 4.20.13，所有操作均在root用户下进行，切换到 root 用户

命令为 `sudo su`，密码: bjut-cs （或键入自己设置的密码）

```
root@linux-exp:~#
```

root用户状态为: root@主机名称:~#

（1）拷贝内核到 `/usr/src` 目录下，输入命令：

```
cp -ri 内核保存路径/ linux-4.20.13.tar.xz /usr/src
```

（2）输入命令：

```
cd /usr/src
```

进入 `/usr/src` 目录（可以输入 `ls` 命令会发现目录下有一个名为

linux-4.20.13.tar.xz 的压缩文件)

(4) 当前目录下(/usr/src), 先后输入命令:

```
xz -d linux-4.20.13.tar.xz
```

```
tar xvf linux-4.20.13.tar
```

解压缩源代码, 命令执行完毕后, 在 /usr/src/ 下, 会出现 **linux-4.20.13** 文件夹。

(5) 进入 /usr/src/ linux-4.20.13路径

```
cd /usr/src/ linux-4.20.13/
```

2.添加系统调用

修改 /usr/src/ linux-4.20.13/ 路径下的三个文件:

(1) 编辑 **sys.c** 文件, 在首部加入 **linkage.h** 的头文件声明, 在尾部加入用户定义的系统调用函数。

```
cd /usr/src/ linux-4.20.13/kernel
```

```
cp sys.c sys.c.bak (备份原文件)
```

```
vim sys.c
```

首先, 在首部加入 linkage.h 的头文件声明

```
1. #include <linux/linkage.h>
```

然后, 在文件末尾加入函数:

```
1. SYSCALL_DEFINE1(mycall, int, number)
2. {
3.     printk("This is my first system call %d ", number);
4.     return number;
5. }
```

注: 在vim编辑器中, **i** 进入编辑; **ESC**退出编辑; **G**跳到文件末尾;

保存并退出: 按**ESC**键后, 输入 **:wq**

只退出不保存: 按**ESC**键, 输入 **:q!**

(2) 编辑 **syscalls.h** 文件，添加函数声明。

```
cd /usr/src/ linux-4.20.13/arch/x86/include/asm/
```

```
cp syscalls.h syscalls.h.bak （备份原文件）
```

```
vim syscalls.h
```

在文件末尾插入：

```
1. asmlinkage long sys_mycall (int);
```

(3) 编辑 **syscall_64.tbl** 文件，添加系统调用号。

```
cd /usr/src/ linux-4.20.13 /arch/x86/entry/syscalls
```

```
cp syscall_64.tbl syscall_64.tbl.bak （备份原文件）
```

```
vim syscall_64.tbl
```

在文件中添加自己的系统调用号： 假设原来common的系统调用号到 334号，那我们自己的系统调用号就设置为 335

在334号后面的一行，添加

```
335      64      mycall      __x64_sys_mycall
```

(4) 需要将修改后的文件恢复原样时，输入命令：

```
cp sys.c.bak sys.c
```

```
cp syscalls.h.bak syscalls.h
```

```
cp syscall_64.tbl.bak syscall_64.tbl
```

将备份文件覆盖修改的文件即可

3. 编译内核

```
cd /usr/src/ linux-4.20.13
```

```
make mrproper
```

```
make clean
```

```
make menuconfig （出现图形对话框后，选择保存，关闭）
```

*若此步骤出现报错并提示 Your display is too small to run Menuconfig! 请调大系统设置中的分辨率，然后重新执行命令

```
sed -ri '/CONFIG_SYSTEM_TRUSTED_KEYS/s/=.+/=""/' .config
```

```
make -j4 （耗时最长，大约60分钟，启用多线程，根据自己电脑的线程数来设置）
```

4. 安装内核

```
make modules_install
```

```
make install
```

执行完毕后，重启虚拟机，直接进入新的内核。

注：由于bootloader默认启动版本号更高的内核，若使用的系统内核版本号高于4.20.13，需要手动设置启动内核，按如下步骤操作（内核版本低于4.20.13可忽略以下内容）：

(1) 终端输入命令：

```
vim /boot/grub/grub.cfg
```

找到 **submenu 'Advanced options for Ubuntu'**，此处列出了系统可用的不同版本的linux内核。若成功编译安装了新内核，可以在这里看到系统原本的内核版本，和新编译的4.20.13版本内核

```
submenu 'Advanced options for Ubuntu' $menuentry_id_option 'gnulinux-advanced-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    menuentry 'Ubuntu, with Linux 4.20.13' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gn  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.20.13 (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class os $menuent  
'gnulinux-4.20.13-recovery-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-171-generic' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id  
'gnulinux-4.15.0-171-generic-advanced-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-171-generic (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class  
'gnulinux-4.15.0-171-generic-recovery-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-167-generic' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id  
'gnulinux-4.15.0-167-generic-advanced-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-167-generic (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class  
'gnulinux-4.15.0-167-generic-recovery-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-156-generic' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id  
'gnulinux-4.15.0-156-generic-advanced-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
menuentry 'Ubuntu, with Linux 4.15.0-156-generic (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class  
'gnulinux-4.15.0-156-generic-recovery-e05126ed-131d-4967-a06c-84dcba8d03c' {  
    ...  
}  
}
```

(2) 在 root 用户下输入命令：

```
vim /etc/default/grub
```

打开文件，找到

```
1. ...
2. GRUB_DEFAULT=0
3. GRUB_TIMEOUT_STYLE=hidden
4. GRUB_TIMEOUT=2
5. ...
```

将 **GRUB_DEFAULT=0** 改为:

GRUB_DEFAULT='Advanced options for Ubuntu>Ubuntu, with Linux 4.20.13'

(3) 更新grub并重启，输入命令:

update-grub

reboot

(4) 重启后再次查看当前的内核版本

uname -a

5.测试新系统调用是否成功

(1) 编写测试程序

testcall.c

```
1. #include<stdio.h>
2. #include<linux/kernel.h>
3. #include<unistd.h>
4. #include<sys/syscall.h>
5.
6. int main()
7. {
8.     long a;
9.     a = syscall(335, 666) ; //调用第 335 号系统调用，即 sys_mycall();
10.    printf("The number from syscall is %ld\n",a);
11.    return 0;
12. }
```

终端编译 **gcc testcall.c -o testcall**

运行 **./testcall**

查看结果： 终端输出 **The number from syscall is 666** 表明系统调用添加成功。

实验二 内核模块的添加和卸载

1.了解模块的编程方法。并编写HelloWorld模块。

(1) 首先编写一个HelloWorld.c程序，如图1:

```
1  /* HelloWorld.c */
2
3  #include <linux/module.h>
4  #include <linux/init.h>
5
6  static int moduletest_init(void)
7  {
8      printk("HelloWorld!\n");
9      return 0;
10 }
11
12 static void moduletest_exit(void)
13 {
14     printk("HelloWorld exit!\n");
15 }
16
17 module_init(moduletest_init);
18 module_exit(moduletest_exit);
19
20 MODULE_LICENSE("GPL");
21
```

两个头文件： linux/module.h 和 linux/init.h是编写模块所必须的头文件

第一个函数 moduletest_init() 实现模块加载时的动作；第二个函数 moduletest_exit()实现模块卸载时的动作。这两个函数的函数名可以随意指定。

module_init 和 module_exit 是两个宏。括号里面的函数名对应的函数才是模块加载和模块卸载时要执行的操作。

最后一句表示模块遵循的公共许可证。

(2) 编译内核模块

✓ 编写Makefile文件

在与HelloWorld.c相同的目录下，新建文件名为Makefile的文件（注意M要大写）。编辑以下内容并保存。

```
1  obj-m := HelloWorld.o
2  all:
3      make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
4  clean:
5      make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

✓ 编译内核模块

在与Makefile文件相同的目录下，执行命令： make

命令成功执行后，会在当前目录下生成许多新的文件：

HelloWorld.o HelloWorld.ko HelloWorld.mod.o HelloWorld.mod.c

Modules.symvers

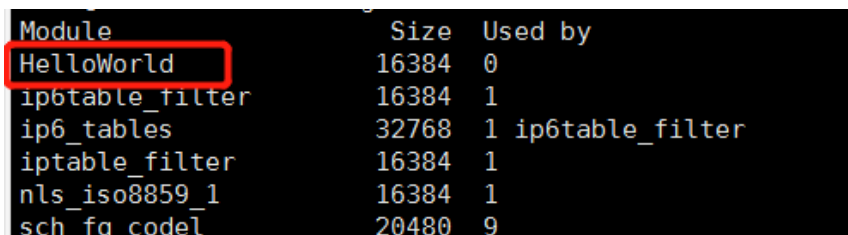
HelloWorld.ko就是我们要加载的模块。

✓ 加载模块

在当前目录下，输入命令：

`insmod ./HelloWorld.ko`

输入命令：`lsmod`，看能否找到名为HelloWorld的模块，若有，则说明模块已经加载成功。



Module	Size	Used by
HelloWorld	16384	0
ip6table_filter	16384	1
ip6_tables	32768	1 ip6table_filter
iptable_filter	16384	1
nls_iso8859_1	16384	1
sch_fq_codel	20480	9

输入命令：`dmesg`，查看最后一行，会有模块加载时调用的函数输出。

✓ 卸载模块

输入命令：`rmmod HelloWorld`,卸载模块

使用命令`lsmod`查看时，已经找不到HelloWorld,说明模块已经被卸载。

输入命令 `dmesg`，查看模块卸载时调用的函数输出。



```
[ 1129.839160] HelloWorld!  
[ 1201.913856] HelloWorld exit!
```

可以用`make clean` 命令清除编译生成的文件，以便重新编译。

实验三 编写系统时钟模块

一、clock模块的加载和卸载

1. 新建一个名为 **clock** 的文件夹，在clock文件夹下编写 **clock.c** 源程序。

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/uaccess.h>
#include <linux/seq_file.h>

//Prototypes
static int init_clock(void);
static void cleanup_clock(void);
static ssize_t proc_read_clock(struct file *file, char __user *pszPage, size_t size, loff_t *off);

//define MODULE
#define USER_MODULE_VERSION "1.0"
#define USER_ROOT_DIR "clock"
#define MODULE_NAME "my_clock"

//user defined directory
struct proc_dir_entry *my_clock_dir;
struct proc_dir_entry *my_clock_file;

static const struct file_operations my_clock_fops =
{
    .owner = THIS_MODULE,
    .read = proc_read_clock,
};

//Functions
static int init_clock(void)
{
    //Create user root dir under /proc
    printk("clock: init_module()\n");
    my_clock_dir= proc_mkdir(USER_ROOT_DIR,NULL);
    my_clock_file = proc_create(MODULE_NAME,0,my_clock_dir, &my_clock_fops);
    printk(KERN_INFO"%s %s has initialized.\n",MODULE_NAME,USER_MODULE_VERSION);
    return 0;
}
```

```
static void cleanup_clock(void)
{
    printk("clock: cleanup_module()\n");
    remove_proc_entry(MODULE_NAME,my_clock_dir);
    remove_proc_entry(USER_ROOT_DIR,NULL);
    printk(KERN_INFO"%s %s has removed.\n",MODULE_NAME,USER_MODULE_VERSION);
}
```

```
static ssize_t proc_read_clock(struct file *file,char __user *pszPage,size_t size,loff_t *off)
{
    int len = 0;
    struct timeval tv;
    do_gettimeofday(&tv);
    len = sprintf(pszPage,"%ld %ld\n",tv.tv_sec,tv.tv_usec);
    return len;}

module_init(init_clock);
module_exit(cleanup_clock);

MODULE_DESCRIPTION("clock module for gettimeofday of proc.");
MODULE_LICENSE("GPL");
```

2. 编译和加载clock模块

编译和加载模块的方法同实验二。加载clock模块后，进入 /proc 目录，用ls命令，会发现存在一个名为 clock 的文件。

3. 编写测试函数 testclock.c

```
#include<sys/time.h>
#include<unistd.h>

#include<stdio.h>。、

#include<stdlib.h>
int main()
{
    struct timeval tv;
    FILE *fp;
    char info[128];
    fp=fopen("/proc/clock","r");
    if(fp==NULL)
    {
        printf("no module");
        exit(0);
    }
    fgets(info,30,fp);
```

```
printf("system time:%s\n",info);
gettimeofday(&tv,0);
printf("%ld %ld\n",tv.tv_sec,tv.tv_usec);
fclose(fp);
}
```

编译test.c测试程序，对模块进行测试。

4. 卸载clock模块

模块卸载的方法同实验二。卸载模块后，再查看 /proc ,将找不到clock模块

二、程序分析

1. 模块程序

模块加载时调用 proc_create 函数，

```
static inline struct proc_dir_entry *proc_create(
    const char *name,
    umode_t mode,
    struct proc_dir_entry *parent,
    const struct file_operations *proc_fops)
{return proc_create_data(name, mode, parent, proc_fops, NULL);}
```

name:名字

mod:模式

parent:父 entry, 为 NULL 的话, 默认父 entry 是 /proc

proc_fops:操作函数表

该函数会创建一个 PROC entry, 用户可以通过此文件, 和内核进行数据交互。

详细内容请查看 /usr/src/linux-自己的内核版本号/include/linux/proc_fs.h 文件。

2. linux在源码/include/linux/time.h 中定义了关于时间的数据结构:

```
struct timespec{
    time_t tv_sec; //秒
    long tv_nsec; //纳秒
}

struct timeval{
    time_t tv_sec; //秒
    long tv_usec; //微秒
}
```

Linux 中定义了一个全局系统变量 `struct timeval xtime`, 用来保存当前时间;
`proc_read_clock` 函数的第二个参数指向了 `/proc/clock/my_clock`文件的缓冲区,
因此调用了`sprintf`后, 系统时间`xtime`的值被格式化输入到`my_clock`文件中。