苏州大学实验报告

院、系	计算机学院	年级专业	20计科	姓名	柯骅	学号	2027405033
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无		实验日期	2023.04.13	

实验名称 实验6 系统调用

一、 实验目的

1. 学习 Linux 内核的系统调用方法。
2. 理解并掌握 Linux 系统调用的实现框架、用户界面、参数传递、进入/返回过程。

二、 实验内容

使用编译内核法和内核模块法添加一个不用传递参数的系统调用，其功能是简单输出类似“hello world! ”这样的字符串。

三、 实验步骤

1. 使用内核编译法添加系统调用

使用这种方法需要重新编译内核，常见问题详见实验5。

(1) 获取 root 权限

```
# sudo -i
```

(2) 进入 kernel 目录

```
# cd /usr/src/linux-4.16.10/kernel
```

(3) 打开 sys.c 并在其中加入如下函数：

```
1. asmlinkage long sys_helloworld(void){
2.     printk("hello world! by Ke Hua(2027405033)");
3.     return 1;
4. }
```

使用“gedit sys.c”进入图形化界面编辑并保存：

```
asmlinkage long sys_helloworld(void){
        printk("hello world! by Ke Hua(2027405033)");
        return 1;
}

#endif /* CONFIG_COMPAT */
```

或使用“vim sys.c”打开并进入编辑模式进行插入，使用“:wq”保存并推出：

```
asmlinkage long sys_helloworld(void){
        printk("hello world! by Ke Hua(2027405033)");
        return 1;
}

#endif /* CONFIG_COMPAT */
```

(4) 添加声明

```
# cd /usr/src/linux-4.16.10/arch/x86/include/asm/
# vim syscalls.h
```

在 syscalls.h 中加入如下函数声明：

```
1. asmlinkage long sys_helloworld(void);
```

```
/* Common in X86_32 and X86_64 */
/* kernel/ioport.c */
asmlinkage long sys_ioperm(unsigned long, unsigned long, int);
asmlinkage long sys_iopl(unsigned int);
asmlinkage long sys_helloworld(void);
```

(5) 添加一个系统调用号

```
# cd /usr/src/linux-4.16.10/arch/x86/entry/syscalls
# vim syscall_64.tbl
```

在系统调用表（其最前面的属性是 id）中添加一个 id 为 333 的系统调用号（如下图光标所处行），添加后保存 syscall_64.tbl 文件，系统调用号添加示例：

```
1. 333     64      helloworld              sys_helloworld
```

```
329     common  pkey_mprotect           sys_pkey_mprotect
330     common  pkey_alloc              sys_pkey_alloc
331     common  pkey_free               sys_pkey_free
332     common  statx                   sys_statx
333     64      helloworld              sys_helloworld
```

(6) 配置内核

清除旧目标文件并重新配置内核，依次执行以下语句（与实验 5 类似）：

```
# cd /usr/src/linux-4.16.10
# make mrproper
# make clean
# make menuconfig
```

为了更明显的看到编译内核的版本，在 makeconfig 时将 General setup 界面上的 Local version 修改成新的名称，如 myKernel，如下图所示：

```
()  Cross-compiler tool prefix (NEW)
[ ] Compile also drivers which will not load
(myKernel) Local version - append to kernel release
[ ] Automatically append version information to the version string
    Kernel compression mode (Gzip)  --->
((none)) Default hostname
```

选择<Save>保存为.config 后选择<Exit>退出 menuconfig 界面，然后输入如下指令：

```
# sudo gedit .config
```

使用 Ctrl+F,搜索 CONFIG_SYSTEM_TRUSTED_KEYS 和 CONFIG_SYSTEM_REVOCATION_KEYS, 删除这两条引号中的内容(而不包括引号)，随后保存，以防编译失败。

(7) 编译和安装内核（与实验 5 类似）

依次输入以下几条指令对内核与模块进行编译与安装：

```
# sudo make -j16
# sudo make modules_install
# sudo make install
```

运行成功后结果如下：

```
LD [M]  sound/usb/bcd2000/snd-bcd2000.ko
LD [M]  sound/usb/line6/snd-usb-line6.ko
LD [M]  sound/usb/hiface/snd-usb-hiface.ko
LD [M]  sound/usb/line6/snd-usb-podhd.ko
LD [M]  sound/usb/line6/snd-usb-pod.ko
LD [M]  sound/usb/line6/snd-usb-toneport.ko
LD [M]  sound/usb/line6/snd-usb-variax.ko
LD [M]  sound/usb/snd-usb-audio.ko
LD [M]  sound/usb/misc/snd-ua101.ko
LD [M]  sound/usb/snd-usbmidi-lib.ko
LD [M]  sound/usb/usx2y/snd-usb-us122l.ko
LD [M]  sound/usb/usx2y/snd-usb-usx2y.ko
LD [M]  sound/x86/snd-hdmi-lpe-audio.ko
```

```
INSTALL sound/usb/hiface/snd-usb-hiface.ko
INSTALL sound/usb/line6/snd-usb-line6.ko
INSTALL sound/usb/line6/snd-usb-pod.ko
INSTALL sound/usb/line6/snd-usb-podhd.ko
INSTALL sound/usb/line6/snd-usb-toneport.ko
INSTALL sound/usb/line6/snd-usb-variax.ko
INSTALL sound/usb/misc/snd-ua101.ko
INSTALL sound/usb/snd-usb-audio.ko
INSTALL sound/usb/snd-usbmidi-lib.ko
INSTALL sound/usb/usx2y/snd-usb-us122l.ko
INSTALL sound/usb/usx2y/snd-usb-usx2y.ko
INSTALL sound/x86/snd-hdmi-lpe-audio.ko
DEPMOD  4.16.10myKernel
```

```
Found linux image: /boot/vmlinuz-4.16.10
Found initrd image: /boot/initrd.img-4.16.10
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

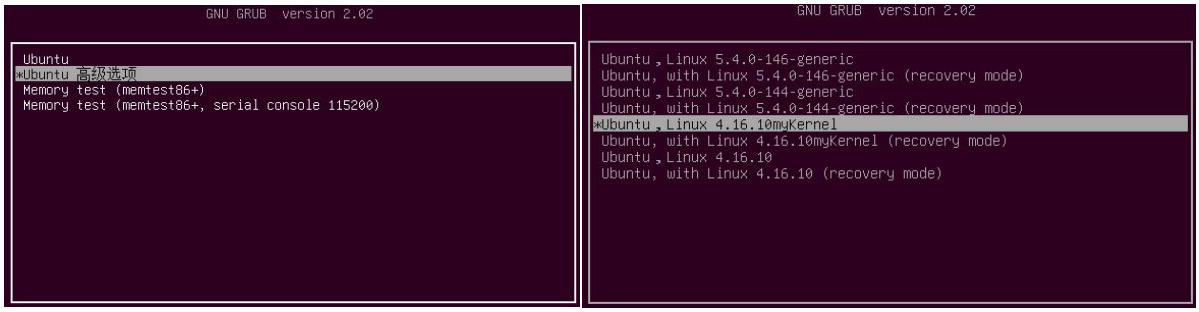
(8) 重启系统

输入指令“reboot”重新启动，并长按“Shift”键进入 GRUB 菜单：

```
# reboot
# //长按 shift
```

使用“⬆”“⬇”键选择，使用“Enter”键进入“Ubuntu 高级选项”

选择选项“Ubuntu Linux4.16.10myKernel”按下“Enter”回车后启动此内核。



使用“uname -r”查看此时的内核版本，结果如下：

2. 使用内核模块法添加系统调用

(1) 查询 sys_call_table 的地址

使用如下命令来查询 sys_call_table 的地址

```
# sudo cat /proc/kallsyms | grep sys_call_table
```

查询结果如下：

得到的地址为:ffffffffadc00180,用该地址替换/module/hello.c 中的 SYS_CALL_TABLE_ADDRESS

(2) 编写 hello.c 文件，内容详见附件/module/hello.c

需要将 hello.c 文件中 sys_call_table 的地址修改为用户各自计算机中第（1）步查询的地址

关键代码如下：

```
#define SYS_CALL_TABLE_ADDRESS 0xffffffffadc00180  //sys_call_table对应的地址
```

(3) 编写 Makefile 文件，可参考附件/module/Makefile

需要注意的是，LINUX_KERNEL_PATH 需要被改成当前使用的 kernel 目录

(4) 依次执行以下命令，编译 hello 模块并将其装入系统。

```
# sudo make
# sudo insmod hello.ko
# lsmod //查看所有模块，从而检查 hello 模块是否被装进系统
```

运行结果如下所示：

```
kh@ubuntu:~/test/test6/module$ sudo make
make -C /usr/src/linux-headers-5.4.0-146-generic M=/home/kh/test/test6/module modules
make[1]: 进入目录“/usr/src/linux-headers-5.4.0-146-generic”
  CC [M]  /home/kh/test/test6/module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/kh/test/test6/module/hello.mod.o
  LD [M]  /home/kh/test/test6/module/hello.ko
make[1]: 离开目录“/usr/src/linux-headers-5.4.0-146-generic”
kh@ubuntu:~/test/test6/module$ sudo insmod hello.ko
kh@ubuntu:~/test/test6/module$ lsmod
Module                  Size  Used by
hello                  16384  0
```

可以发现，hello 模块已经被装进系统。

四、 实验结果

(1) 内核编译法

为了验证系统调用是否成功，可使用附件/kernel/hello.c 来验证：

使用如下命令编译示例代码：

```
# gcc hello.c
```

执行验证代码，可输入下列命令：

```
# ./a.out
```

运行结果如下：

```
kh@ubuntu:~/test/test6/kernel$ ./a.out
System call sys_helloworld reutrn 1
```

这表明成功添加了系统调用，并成功实现了调用

(2) 内核模块法

创建测试程序 test.c 以测试新增的系统调用是否可以正常工作。test.c 测试程序中的关键代码如下，详见附件/module/test.c。

```
1. x = syscall(223);        //测试 223 号系统调用
2. printf("syscall result: %ld\n", x);
```

依次执行以下命令，编译并运行测试程序：

```
# gcc -o test test.c
# ./test
```

运行结果如下：

```
kh@ubuntu:~/test/test6/module$ gcc -o test test.c
kh@ubuntu:~/test/test6/module$ ./test
syscall result: 3866
```

这表示 test.c 成功调用了新增的 hello 模块

此外，可以利用如下 dmesg 命令查看系统日志输出

```
# dmesg | tail
```

结果如下图，其中展示了模块初始化、系统调用添加的系统日志输出，这表明试验成功，即成功初始化模块、添加系统调用。

```
[ 1200.945495] 模块系统调用-当前pid: 3866, 当前comm:test
[ 1200.945496] hello world!by Ke Hua(2027405033)
```

依次运行以下指令，可以测试模块在被卸载时是否正常：

```
# sudo rmmod hello    //删除 hello 模块
# lsmod | grep hello //查看 hello 是否存在，即是否被卸载
# dmesg | tail       //查看日志
```

运行结果如下：

```
kh@ubuntu:~/test/test6/module$ sudo rmmod hello
kh@ubuntu:~/test/test6/module$ lsmod | grep hello
kh@ubuntu:~/test/test6/module$ dmesg |tail
[ 1004.184790] call_init......
[ 1007.689978] 模块系统调用-当前pid: 3838, 当前comm:test
[ 1007.689979] hello world!by Ke Hua(2027405033)
[ 1116.262119] call_exit......
[ 1196.613906] call_init......
[ 1200.945495] 模块系统调用-当前pid: 3866, 当前comm:test
[ 1200.945496] hello world!by Ke Hua(2027405033)
[ 1531.113295] call_exit......
```

最后一行结果为“call_exit”表示模块正常卸载，表明实验成功。

五、 实验思考与总结

1. 思考 1：vim 的使用

vim 中有三种模式：命令模式，编辑模式，末行模式

命令模式：

“vim［文件名］”打开后的默认模式，在这个模式下，可以进行光标移动、搜索替换、复制粘贴删除等操作，其中常用指令如下：

指令	作用
h 或 左方向键（←）	光标向左移动一个字符
l 或 右方向键（→）	光标向右移动一个字符
k 或 上方向键（↑）	光标向上移动一个字符
j 或 下方向键（↓）	光标向下移动一个字符
gg	移动到文件的第一行
G	移动到文件的最后一行

编辑模式：

命令模式按 i 进入编辑模式，此模式可对内容进行编辑，常用命令如下：

指令	作用
i	进入输入模式，进入后显示 – INSTER–
o	进入输入模式，在光标下一行插入新行，进入后显示 – INSTER –
R	进入取代模式，输入的值会取代光标所在的内容，进入后显示 – REPLACE –
esc键	退出编辑模式（输入模式）

末行模式：

英文状态的”:”键进入末行模式，常用指令如下：

指令	作用
:w	保存
:q	退出
:wq	保存后退出
:q!	不保存，强制退出

2. **思考 2：dmesg 命令**

dmesg 是一种程序，用于检测和控制内核环缓冲。程序用来帮助用户了解系统的启动信息。

常见用法：

```
# dmesg | tail (-[行数 n])
# dmesg | head (-[行数 n])
```

查看 dmesg 命令的头部（head）或末尾（tail）n 行日志

```
# dmesg | grep (-i) [string]
```

过滤出包含 string 字符串的日志,-i 表示忽略大小写

```
# dmesg -c
```

清空 dmesg 日志

```
# watch “dmesg | tail -20”
```

实时监控 dmesg 日志输出

3. **实验总结**

本次实验使用了内核编译法和内核添加法成功添加了一个不用传递参数的系统调用：内核编译法是在内核编译之前在相关文件中添加一个函数，并将其添加到系统调用，在编译安装好内核后，进入内核检验是否正确添加了系统调用。

内核添加法是在当前内核中,利用事先打包好的 Makefile 文件将相关函数添加到系统调用模块,随后使用 test.c 程序和 dmesg 命令两种方式检验是模块否正确加载，调用与卸载。

在本次实验中，学习了Linux内核的系统调用方法，理解并掌握了Linux系统调用的实现框架、用户界面、参数传递、进入/返回过程，同时对于Linux系统调用有了更深的理解，在遇到问题时，可以通过问题的提示进行操作或搜索相关技术来解决，对问题的处理有了更多的经验。