苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
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	2022 年 5 月 10 日	

实 验 名 称　　实验 7　Linux 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. 操作方法和实验步骤

1. 使用内核编译法添加系统调用

（1）在 kernel/sys.c 中加入如下函数：

```
asmlinkage long sys_helloworld(void) {
    printk("hello world, syscall-kernel, by Zhang Hao(5160) !");
    return 1;
}
```

```
asmlinkage long sys_helloworld(void) {
        printk("hello world, syscall-kernel, by Zhang Hao(5160) !");
        return 1;
}
#endif /* CONFIG_COMPAT */
"kernel/sys.c" 2567L, 62198C written
```

（2）向 arch/x86/include/asm/syscalls.h 文件中添加声明：

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

```
asmlinkage long sys_helloworld(void);

"arch/x86/include/asm/syscalls.h" 57L, 1459C written
```

（3）在 arch/x86/entry/syscalls/syscall_64.tbl 中添加一个系统调用号，插入：（第一列为 ID）

```
<ID>   64  helloworld        sys_helloworld
```

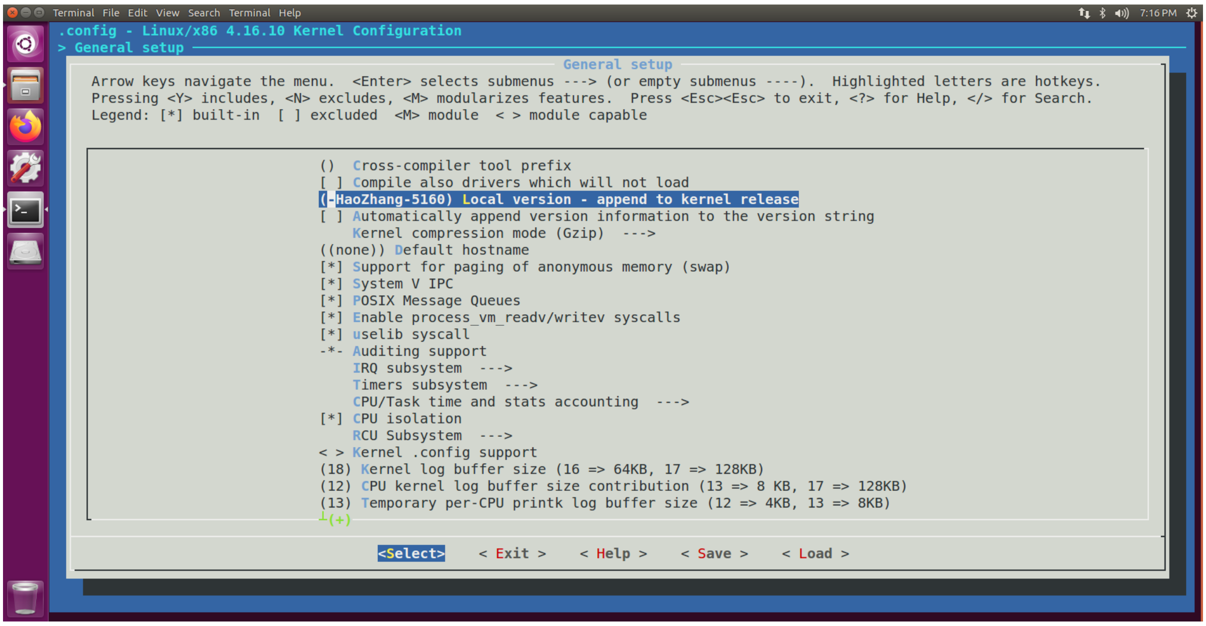
```
331     common  pkey_free               sys_pkey_free
332     common  statx                   sys_statx
333     64      helloworld              sys_helloworld
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
"arch/x86/entry/syscalls/syscall_64.tbl" 382L, 13292C written
```

其中 ID 为：**333**

（4）清除旧的目标文件和配置，重新配置内核

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

为了更明显地看到编译的内核版本，在 make menuconfig 时将 General setup 界面上的 Local version 修改成新的名称-HaoZhang-5160。



（5）编译和安装内核

```
# make -j8
# make modules -j8
# make modules_install
# make install
```

```
root@hao-zhang:/usr/src/linux-4.16.10# make install
sh ./arch/x86/boot/install.sh 4.16.10-HaoZhang-5160 arch/x86/boot/bzImage \
        System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
update-initramfs: Generating /boot/initrd.img-4.16.10-HaoZhang-5160
run-parts: executing /etc/kernel/postinst.d/pm-utils 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
run-parts: executing /etc/kernel/postinst.d/update-notifier 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 4.16.10-HaoZhang-5160 /boot/vmlinuz-4.16.10-HaoZhang-5160
Generating grub configuration file ...
Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is set is no longer supported.
Found linux image: /boot/vmlinuz-4.16.10-HaoZhang-5160
Found initrd image: /boot/initrd.img-4.16.10-HaoZhang-5160
Found linux image: /boot/vmlinuz-4.16.10
Found initrd image: /boot/initrd.img-4.16.10
Found linux image: /boot/vmlinuz-4.15.0-142-generic
Found initrd image: /boot/initrd.img-4.15.0-142-generic
Found linux image: /boot/vmlinuz-4.15.0-112-generic
Found initrd image: /boot/initrd.img-4.15.0-112-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
root@hao-zhang:/usr/src/linux-4.16.10#
```

（6）重启并查看新版本内核

```
# reboot
$ uname -r
```

```
holger@hao-zhang:~$ uname -r
4.16.10-HaoZhang-5160
holger@hao-zhang:~$
```

2. 使用内核模块法添加系统调用

（1）查询 sys_call_table 的地址

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

```
holger@hao-zhang:~$ sudo cat /proc/kallsyms | grep sys_call_table
[sudo] password for holger:
ffffffff8e0001a0 R sys_call_table
ffffffff8e001560 R ia32_sys_call_table
holger@hao-zhang:~$
```

记录得到的地址为：ffffffff8e0001a0

（2）编写 hello.c 文件：

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/unistd.h>
#include <linux/sched.h>
MODULE_LICENSE("Dual BSD/GPL");
#define SYS_CALL_TABLE_ADDRESS 0xffffffff8e0001a0  //sys_call_table 地址
#define NUM 223  //系统调用号为 223
int orig_cr0;  //用来存储 cr0 寄存器原来的值
unsigned long *sys_call_table_my=0;
static int(*anything_saved)(void);  //定义一个函数指针，用来保存一个系统调用

static int clear_cr0(void) {  //使 cr0 寄存器的第 17 位设置为 0（内核空间可写）
    unsigned int cr0=0;
    unsigned int ret;
    //将 cr0 寄存器的值移动到 eax 寄存器中，同时输出到 cr0 变量中
    asm volatile("movq %%cr0,%%rax":"=a"(cr0));
    ret=cr0;
    //将 cr0 变量值中的第 17 位清 0,将修改后的值写入 cr0 寄存器
    cr0&=0xfffffffffffeffff;
    //将 cr0 变量的值作为输入，输入到寄存器 eax 中，同时移动到寄存器 cr0 中
    asm volatile("movq %%rax,%%cr0"::"a"(cr0));
    return ret;
}

static void setback_cr0(int val) {  //使 cr0 寄存器设置为内核不可写
    asm volatile("movq %%rax,%%cr0"::"a"(val));
}

asmlinkage long sys_mycall(void) { //定义自己的系统调用
    printk("** SYSCALL Hao Zhang ** pid=%d, comm:%s\n",
    current->pid, current->comm);
    printk("hello world, syscall-module, by Zhang Hao(5160) !\n");
    return current->pid;
```

```
}

static int __init call_init(void) {
    sys_call_table_my=(unsigned long*)(SYS_CALL_TABLE_ADDRESS);
    printk("call_init......\n");
    //保存系统调用表中的 NUM 位置上的系统调用
    anything_saved=(int(*)(void))(sys_call_table_my[NUM]);
    orig_cr0=clear_cr0();//使内核地址空间可写
    //用自己的系统调用替换 NUM 位置上的系统调用
    sys_call_table_my[NUM]=(unsigned long) &sys_mycall;
    setback_cr0(orig_cr0);//使内核地址空间不可写
    return 0;
}

static void __exit call_exit(void) {
    printk("call_exit......\n");
    orig_cr0=clear_cr0();
    sys_call_table_my[NUM]=(unsigned long)anything_saved;//将系统调用恢复
    setback_cr0(orig_cr0);
}

module_init(call_init);
module_exit(call_exit);
```

其中 sys_call_table 的地址需要修改为上述步骤（1）中查询到的计算机中的地址。

（3）编写 Makefile 文件：

```
obj-m := hello.o
CURRENT_PATH:=$(shell pwd)
LINUX_KERNEL_PATH:= /usr/src/linux-$(shell uname -r | cut -d '-' -f1)
all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
```

注意 LINUX_KERNEL_PATH 要设置成为计算机中的内核源码路径。

（4）编译 hello 模块并将其装入系统：

```
$ sudo make
$ sudo insmod hello.ko
```

```
holger@hao-zhang:~/code/exp07/module$ sudo make
make -C /usr/src/linux-4.16.10 M=/home/holger/code/exp07/module modules
make[1]: Entering directory '/usr/src/linux-4.16.10'
  CC [M]  /home/holger/code/exp07/module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/holger/code/exp07/module/hello.mod.o
  LD [M]  /home/holger/code/exp07/module/hello.ko
make[1]: Leaving directory '/usr/src/linux-4.16.10'
holger@hao-zhang:~/code/exp07/module$ sudo insmod hello.ko
```

四. 实验结果和分析

1. 内核编译法

使用如下代码调用系统调用，验证是否成功：

```
/* test-kernel.c */
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#define SYSCALL_ID 333

int main() {
    long ret = syscall(SYSCALL_ID);
    printf("System call sys_helloworld reutrn %ld\n", ret);
    return 0;
}
```

编译：

```
$ gcc test-kernel.c -o test-kernel
```

运行：

```
holger@hao-zhang:~/code/exp07$ gcc test-kernel.c -o test-kernel
holger@hao-zhang:~/code/exp07$ ./test-kernel
System call sys_helloworld reutrn 1
holger@hao-zhang:~/code/exp07$
```

查看系统日志输出：

```
$ sudo dmesg | tail
```

```
holger@hao-zhang:~/code/exp07$ sudo dmesg | tail
[  553.617284] e1000: ens33 NIC Link is Down
[  565.704575] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[  567.724884] e1000: ens33 NIC Link is Down
[  571.848730] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[  573.864003] e1000: ens33 NIC Link is Down
[  577.895766] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 1702.253641] perf: interrupt took too long (17561 > 16615), lowering kernel.perf_event_max_sample_rate to 11250
[ 2390.735026] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 5.158 msecs
[ 2924.199452] perf: interrupt took too long (22597 > 21951), lowering kernel.perf_event_max_sample_rate to 8750
[ 3141.494450] hello world, syscall-kernel, by Zhang Hao(5160) !
```

可见系统调用成功执行。

2. 内核模块法

首先检查模块是否正确装入：

```
holger@hao-zhang:~/code/exp07/module$ lsmod | grep hello
hello                  16384  0
holger@hao-zhang:~/code/exp07/module$
```

可见 hello 模块已经装入系统内。

使用如下代码调用系统调用，验证是否成功：

```
/* test-module.c */
#include<stdio.h>
#include<stdlib.h>
#include<linux/kernel.h>
#include<sys/syscall.h>
```

```
#include<unistd.h>
#define SYSCALL_ID 223

int main() {
    unsigned long ret = syscall(SYSCALL_ID);
    printf("System call reutrn %ld\n", ret);
    return 0;
}
```

编译：

```
$ gcc test-module.c -o test-module
```

运行：

```
holger@hao-zhang:~/code/exp07/module$ gcc test-module.c -o test-module
holger@hao-zhang:~/code/exp07/module$ ./test-module
System call reutrn 5121
holger@hao-zhang:~/code/exp07/module$
```

查看系统日志输出：

```
$ sudo dmesg | tail
```

```
[ 4137.481109] hello: loading out-of-tree module taints kernel.
[ 4137.481222] hello: module verification failed: signature and/or required key mis
sing - tainting kernel
[ 4137.490075] call_init......
[ 4212.187371] ** SYSCALL Hao Zhang ** pid=5121, comm:test-module
[ 4212.187403] hello world, syscall-module, by Zhang Hao(5160) !
holger@hao-zhang:~/code/exp07/module$
```

从系统日志中可以得知模块验证失败（module verification failed），经查阅得知 Linux 内核自 3.7 版本后加入了内核模块签名检查机制[1]。

首先卸载内核模块：

```
$ sudo rmmod hello
$ lsmod | grep hello
```

```
holger@hao-zhang:~/code/exp07/module$ sudo rmmod hello
[sudo] password for holger:
holger@hao-zhang:~/code/exp07/module$ lsmod | grep hello
holger@hao-zhang:~/code/exp07/module$
```

对于此问题有两种解决方案。

一种是编译内核时，在 Makefile 中增加一行，并重新编译[2]。

```
CONFIG_MODULE_SIG=n
```

一种是对已编译的内核模块进行签名[3]：

```
$ /usr/src/linux-4.16.10/scripts/sign-file sha512 \
   /usr/src/linux-4.16.10/certs/signing_key.pem \
   /usr/src/linux-4.16.10/certs/signing_key.x509 hello.ko
```

这里选择后一种方案，运行后重新装入内核模块：

```
$ sudo insmod hello.ko
$ lsmod | grep hello
```

[1] https://lishiwen4.github.io/linux-kernel/linux-kernel-module-signing
[2] https://blog.csdn.net/caoyahong114/article/details/51744748
[3] https://www.cnblogs.com/rivsidn/p/9481037.html

运行 test-module:

```
$ ./test-module
```

```
holger@hao-zhang:~/code/exp07/module$ sudo rmmod hello
[sudo] password for holger:
holger@hao-zhang:~/code/exp07/module$ lsmod | grep hello
holger@hao-zhang:~/code/exp07/module$ /usr/src/linux-4.16.10/scripts/sign-file sha5
12 /usr/src/linux-4.16.10/certs/signing_key.pem /usr/src/linux-4.16.10/certs/signin
g_key.x509 hello.ko
holger@hao-zhang:~/code/exp07/module$ sudo insmod hello.ko
holger@hao-zhang:~/code/exp07/module$ lsmod | grep hello
hello                  16384  0
```

查看系统日志输出:

```
$ sudo dmesg | tail
```

```
holger@hao-zhang:~/code/exp07/module$ ./test-module
System call reutrn 5901
holger@hao-zhang:~/code/exp07/module$ sudo dmesg | tail
[ 5899.678807] ** SYSCALL Hao Zhang ** pid=5594, comm:test-module
[ 5899.678839] hello world, syscall-module, by Zhang Hao(5160) !
[ 5958.875907] ** SYSCALL Hao Zhang ** pid=5604, comm:test-module
[ 5958.875939] hello world, syscall-module, by Zhang Hao(5160) !
[ 5992.965946] call_exit......
[ 6034.213372] e1000: ens33 NIC Link is Down
[ 6052.436024] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: Non
e
[ 6069.571167] call_init......
[ 6192.588367] ** SYSCALL Hao Zhang ** pid=5901, comm:test-module
[ 6192.588407] hello world, syscall-module, by Zhang Hao(5160) !
holger@hao-zhang:~/code/exp07/module$
```

可见系统调用成功执行。

卸载内核模块:

```
$ sudo rmmod hello
$ lsmod | grep hello
```

查看系统日志输出:

```
$ sudo dmesg | tail
```

```
holger@hao-zhang:~/code/exp07/module$ sudo rmmod hello
holger@hao-zhang:~/code/exp07/module$ lsmod | grep hello
holger@hao-zhang:~/code/exp07/module$ sudo dmesg | tail
[ 5899.678839] hello world, syscall-module, by Zhang Hao(5160) !
[ 5958.875907] ** SYSCALL Hao Zhang ** pid=5604, comm:test-module
[ 5958.875939] hello world, syscall-module, by Zhang Hao(5160) !
[ 5992.965946] call_exit......
[ 6034.213372] e1000: ens33 NIC Link is Down
[ 6052.436024] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: Non
e
[ 6069.571167] call_init......
[ 6192.588367] ** SYSCALL Hao Zhang ** pid=5901, comm:test-module
[ 6192.588407] hello world, syscall-module, by Zhang Hao(5160) !
[ 6482.433020] call_exit......
holger@hao-zhang:~/code/exp07/module$
```

五. 讨论、心得

通过这次实验我学会了 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。实验过程中较为顺利，遇到了一个并不影响实验结果的小问题，通过搜索引擎查找原因并解决了。