### 实验10 添加syscall打印所有进程

## 1实验目的

掌握用户程序如何利用系统调用与操作系统内核实现通信的方法，加深对系统调用机制的理解。

## 2实验内容

本次实验主要是修改linux内核增加一个系统调用，打印出所有的进程信息，包括系统函数的实现、系统调用表、系统调用入口的修改，对新的内核编译安装，调试新的系统调用。

## 3实验原理

3.1 系统调用

系统调用，顾名思义，说的是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置定时器等。

从逻辑上来说，系统调用可被看成是一个内核与用户态程序交互的接口——它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户态。

系统服务之所以需要通过系统调用来提供给用户态的根本原因是为了对系统进行“保护”，因为我们知道Linux的运行空间分为内核态与用户态，它们各自运行在不同的级别中，逻辑上相互隔离。所以用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户态操作用户数据，调用用户态函数。比如我们熟悉的“hello world”程序（执行时）就是标准的用户态进程，它使用的打印函数printf就属于用户态函数，打印的字符“hello word”字符串也属于用户态数据。

但是很多情况下，用户进程需要获得系统服务（调用系统程序），这时就必须利用系统提供给用户的“特殊接口”——系统调用了，它的特殊性主要在于规定了用户进程进入内核的具体位置；换句话说，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不准许肆意跳入内核。有了这样的陷入内核的统一访问路径限制才能保证内核安全无虞。

3.2 内核态中获取进程信息

Linux内核通过一个被称为进程描述符的task\_struct结构体来管理进程，这个结构体包含了一个进程所需的所有信息，比如pid，进程亲属关系等。

struct task\_struct {

volatile long state;

void \*stack;

unsigned int flags;

int prio, static\_prio;

struct list\_head tasks;

struct mm\_struct \*mm, \*active\_mm;

pid\_t pid;

pid\_t tgid;

struct task\_struct \*real\_parent;

char comm[TASK\_COMM\_LEN];

struct thread\_struct thread;

struct files\_struct \*files;

...

};

init\_task是进程0使用的进程描述符，也是Linux系统中第一个进程描述符，该进程的描述符在arch/powerpc/kernel/init\_task.c中定义，代码片段如下：struct task\_struct init\_task = INIT\_TASK(init\_task);init\_task描述符使用宏INIT\_TASK对init\_task的进程描述符进行初始化，宏INIT\_TASK在include/linux/init\_task.h文件中init\_task是Linux内核中的第一个线程，它贯穿于整个Linux系统的初始化过程中，该进程也是Linux系统中唯一一个没有用kernel\_thread() 函数创建的进程。在init\_task进程执行后期，它会调用kernel\_thread()函数创建第一个核心进程kernel\_init，同时init\_task进程继续对Linux系统初始化。

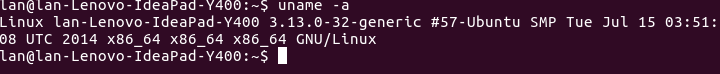
next\_task函数是 sched.h 内的一个宏，它简化了任务列表的迭代（返回下一个任务的 task\_struct 引用）。

通过init\_task和next\_task就可以在内核态中对进程进行遍历了。

## 4实验步骤

4.1增加系统调用

步骤1：uname –a，先查看本机内核版本，以验证后续是否成功安装新的内核。



**图 1 查看原内核版本**

步骤2：下载linux内核，可以从<https://www.kernel.org/>这里下载，不同的内核后续的操作也会有一些不同，本次实验以linux-3.10为例。

步骤3：tar xvJf \*.tar.xz，解压Linux内核。

步骤4：sudo cp –r linux-3.10 /usr/src/ linux-3.10，拷贝至/usr/src/目录下（这个步骤不是必须的，在其他路径也可以完成内核的编译与安装）。

步骤5：sudo gedit /usr/src/linux/kernel/sys.c，在内核中添加新的函数。

步骤6：本次实验添加的两个系统调用，第一个是接收用户态传送的字符串参数并输出，此系统调用较简单，只有输出函数，以此来验证在新内核中添加系统调用是否成功。第二个是打印出所有的进程信息，是本次实验的目标。两个函数的定义方式也有所区别，其中hello系统调用使用旧版本内核中函数定义方式，需要包含头文件<linux/linkage.h>，output\_processes系统调用使用宏来定义。其中宏是在include/linux/syscalls.h头文件中定义的，宏函数名末尾代表着参数数量，系统调用中最多可以有6个参数：

#define SYSCALL\_DEFINE0(sname) \

SYSCALL\_METADATA(\_##sname, 0); \

asmlinkage long sys\_##sname(void)

#define SYSCALL\_DEFINE1(name, ...) SYSCALL\_DEFINEx(1, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE2(name, ...) SYSCALL\_DEFINEx(2, \_##name, \_\_VA\_ARGS\_\_)

在sys.c文件中加入以下函数：

#include <linux/linkage.h>

asmlinkage int sys\_hello(char \* buf) {

printk("====hello\n");

printk("===%s====",buf);

return 1;

}

SYSCALL\_DEFINE0(output\_processes) {

struct task\_struct \*p;

printk("------------输出所有进程------------\n");

printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printk("%-20s %-6s %-6s %-20s\n","Name","pid","state","ParentName");

for(p = &init\_task; (p = next\_task(p)) != &init\_task;)

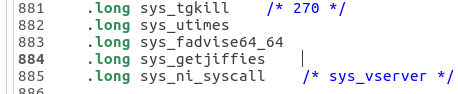
printk("%-20s %-6d %-6d %-20s\n",p->comm , p->pid, p->state, p->parent->comm);

return 1;

}

其中输出进程的系统调用不需要参数，所以使用SYSCALL\_DEFINE0宏。这里的中文输出在后续验证时会发现没有正常地输出，在系统调用中输出的字符串中不要有中文字符。

步骤6：添加系统调用表，在系统调用表中为新函数安排空间。系统调用表在不同版本的内核有所不同，在linux-3.10中/usr/src/linux-3.10/arch/x86/syscalls/syscall\_32.tbl（64位机则为syscall\_64.tbl）。在linux-3.6.8中，系统调用表路径为arch/x86/kernel/syscall\_table\_32.S。在linux-2.6.0则是linux- 2.6.0/arch/i386/kernel/entry.S。在以前的内核版本中该文件有类似如下的清单：.long SYMBOL\_NAME（）该清单用来对sys\_call\_table[]数组进行初始化。该数组包含指向内核中每个系统调用的指针。这样就在数组中增加了新的内核函数的指针。我们在清单最后添加一行：.long sys\_getjiffies。



**图 2 添加系统调用表**

在linux-3.10中则还要把系统调用号添加进来，格式为：

#The format is:

# <number> <abi> <name> <entry point>

#

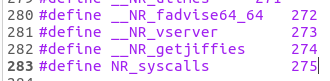
# The abi is "common", "64" or "x32" for this file.

320 64 hello sys\_hello

321 64 output\_processes sys\_output\_processes

把之前定义的两个系统调用添加到系统调用表中，系统调用号适当的选择之前没有选择过的即可。

步骤7：添加系统调用入口。在之前的版本中是在这里添加系统调用号，修改linux-2.6.0/include/asm-i386/unistd.h文件，该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码。我们把添加的新函数加到最后#define \_\_NR\_getjiffies 274，并把代表着表大小的常量NR\_syscalls顺势后移并加1。



**图 3 添加系统调用入口**

在linux-3.10中则是在include/linux/syscalls.h中添加两个系统调用的函数声明（这个步骤不是必须的）：

asmlinkage int sys\_hello(char \*);

asmlinkage long sys\_output\_processes(void);

4.2重新编译内核

步骤1：cd usr/src/linux-3.10，到内核根目录，进行内核的编译安装。

步骤2：sudo make mrproper，如果不是第一次进行内核编译，需要对之前的设置进行清理，清理以前编译留下的临时文件。

步骤3：sudo make menuconfig，生成配置文件。这一步可能会报错，应该是交叉编译工具gcc版本过高，或缺少一些依赖库，报错的话请降低gcc版本或尝试安装ncurses库。为了简便起见，配置文件也可以直接使用当前内核版本的配置文件。cp /boot/config-<Tab> .config，把默认配置文件拷贝到根目录中。sudo make olddefconfig，根据已有配置文件生成新的配置文件。

步骤4：sudo make，可以加-j2，多线程编译加快编译的速度，建议线程数为cpu核数的2倍，这个过程会很漫长，请耐心等待。

步骤5：sudo make modules，编译模块。编译

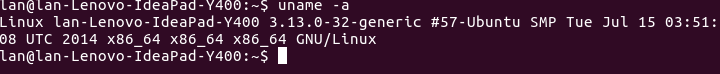
步骤6：sudo make modules\_install，安装模块，安装之前最好备份/lib/modules中相应的模块，因为会把同名内核路径的模块覆盖。

步骤7：sudo make install，会在/boot中生成相应文件，并修改grub.cfg启动配置文件。

步骤8：编译完成，reboot重启选择新的内核即可。

4.3调试系统调用

步骤1：uname –a，验证是否为新内核。



**图 4 查看内核版本**

步骤1：在用户态编程使用系统调用。

步骤2：参考代码。

#include<stdio.h>

#define HELLO\_WORLD 320

#define OUTPUT\_PROCESSES 321

int main()

{

int rc;

char \*buf = “mycall”;

rc = syscall(HELLO\_WORLD, buf);

if (rc == 1) {

printf("syscall %d succeed!\n", HELLO\_WORLD);

} else {

printf("error!\n");

}

rc = syscall(OUTPUT\_PROCESSES);

if (rc == 1) {

printf("syscall %d succeed!\n", OUTPUT\_PROCESSES);

} else {

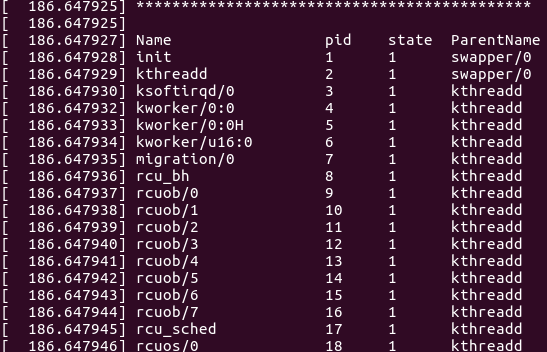
printf("error!\n");

}

步骤3：dmesg，查看printk所输出的内容。

X:\hello.png

**图 5 hello系统调用**



**图 6 output\_processes系统调用**

## 5实验总结

本次实验主要分为三个步骤：下载linux内核，修改内核，添加新的函数，修改系统调用表，添加系统调用入口；修改完内核，对内核进行重新编译；编译成功后使用新的内核启动，在用户态编程使用新的系统调用，验证整个是否添加成功。主要使学生掌握用户态如何利用系统调用与操作系统内核实现通信的方法，加深对系统调用机制的理解。

## 6参考文献

[Linux内核添加系统调用](http://blog.csdn.net/rk2900/article/details/8281335)：以内核linux-3.8.6为例添加一个系统调用。

## 7实验拓展

再添加一个系统调用，函数返回值为系统自启动开始过了多少秒