西安电子科技大学

计算机学院

实

验

报

告

题目： Linux 系统调用的添加

班级： 1403013

姓名： 谢锦源

学号： 14030130100

1. 理论分析

在 x86 兼容机上，早期的 Linux 内核使用 `INT $0x80` 软中断来进行系统调用处理，用户调用系统调用时，将系统调用编号以及其他参数压入寄存器中，然后触发中断，进入中断处理函数，中断处理函数将 CPU 上下文从用户态切换到内核态，通过中断编号从系统调用表中索引到对应的系统调用函数地址，取出参数进行处理，并获取系统调用的返回值，最后切换回调用的应用程序中。

这种方法涉及到到上下文的切换与恢复,会破坏指令流水线、降低 Cache 的命中率，因而调用的代价较高。在奔腾 II 处理器之后，Intel 引入了新的指令 `SYSCALL/SYSRET` 以及 `SYSENTER/SYSEXIT`，对系统调用的效率进行了优化。如今各种各样架构的处理器都已经具有类似的指令来加速系统调用。

2. 设计与实现

在 Linux 内核代码 `arch/x86/entry/common.c` 中，可以发现这些不同的进行系统调用的方式。其中 `do_syscall_64`、`do_int80_syscall_32` 就分别采用了两种不同的调用机制。

翻阅内核的源代码，在文件 `arch/x86/kernel/syscall_64.c` 中，可以知道内核维护了一个名为 `sys_call_table` 的函数指针数组，其初始化时均指向函数 `sys_ni_syscall()`,在编译时,通过引入 `include/asm/syscalls_64.h` 来初始化，而该文件指向 `include/generated/asm/syscalls_64.h`，由编译预处理脚本利用系统调用表 `syscall_64.tbl` 在编译时生成。

相关部分代码如下：

- `syscall_64.c` 系统调用表初始化部分源代码：

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
      /*
       * Smells like a compiler bug -- it doesn't work
       * when the & below is removed.
       */
      [0 ... __NR_syscall_max] = &sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

- syscall_64.tbl 系统调用表（部分）

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
.
.
.
133     common  mknod                   sys_mknod
134     64      uselib
135     common  personality             sys_personality
```

2.1. 通过重新编译内核来添加系统调用

由上面的分析可知，通过在 arch/x86/syscalls/syscall_64.tbl 表中新增一行，指定系统调用编号，调用函数入口，并实现相应的函数，然后重新编译操作系统内核，最后加载新的系统内核就可以实现添加系统调用的目的。具体实现步骤如下：

- 修改 arch/x86/syscalls/syscall_64.tbl 文件

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
```

```
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
.
.
.
133     common  mknod                   sys_mknod
134     64      uselib
135     common  personality             sys_personality
.
.
.
323     common my_syscall               sys_my_syscall
```

- 实现系统调用函数 sys_my_syscall

```
asmlinkage long sys_my_syscall(void) {
printk(KERN_INFO "The message from my_syscall()");
}
```

该函数向系统内核日志输出一段字符 “The message from my_syscall()”

- 将新增加的文件添加到内核 Makefile 中
- 安装新内核并重启系统

```
# sudo make install
# sudo make install_modules
```

- 测试新增的系统调用
  编写测试程序，使用系统调用号的方式来测试新添加的系统调用。

2.2. 动态修改系统调用表来添加系统调用

2.2.1. 原理分析

前面分析中提到 Linux 维护了一个系统调用表 `sys_call_table[]`，并且注意到，在 `arch/x86/syscalls/syscall_64.tbl` 中，如 134 号等系统调用并没有给出对应的系统调用实现，他们都指向 `sys_ni_syscall()`，那么我们可以在系统运行时来动态修改这个系统调用表的指针来注入一个新的系统调用。

要实现这个想法，有下面的问题需要解决：

1. `sys_call_table[]` 在逻辑地址上处于内核区中，需要取得内核层级的运行权限才能够对该表进行修改；
   该问题的一个比较简单的解决方法是使用 Linux 的内核扩展机制来实现，Linux 的内核扩展通常用来实现 Linux 下的设备驱动，它与内核一样都运行在内核态。

2. 需要找出 `sys_call_table[]` 在运行时的内存地址，并使其变得可写；
   该问题可以采用“扫描内核地址空间的方式来查找 `sys_call_table[]` 的地址，并在通过设置其分页表项使其能够被修改”的方法解决。

实现如下：

```
static inline void protect_memory(void) {
    set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));
}

static inline void unprotect_memory(void) {
    set_pte_atomic(pte, pte_mkwrite(*pte));
}

static unsigned long **find_sys_call_table(void) {
    unsigned long **sys_table;
    unsigned long offset = PAGE_OFFSET;

    while (offset < ULONG_MAX) {
        sys_table = (unsigned long **) offset;
```

```
        if (sys_table[__NR_close] == (unsigned long *) sys_close) {
            return sys_table;
        } else {
            offset += sizeof(void *);
        }
    }

    return NULL;
}
```

有了上面的准备工作，那么就可以着手修改 sys_call_table[] 表，这个过程比较简单：为了不改变原有的系统调用行为，我们在 sys_call_table[] 中查找一个指向 sys_ni_syscall() 函数的位置，然后将这个位置的指针修改为我们自己定义的系统调用。

2.2.2. 实现代码

- dynamic_syscall_loader.c 系统调用动态加载器实现代码
- example/simple_syscall.c 通过加载器动态加载系统调用源代码
- example/test.c 系统调用测试源代码

3. 实验结果

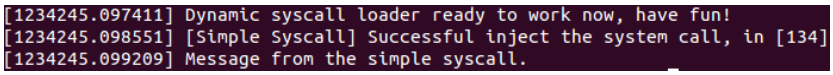
3.1. 方法一

重新编译内核并成功加载新内核后，通过系统调用号调用自定义的系统调用，可以通过 dmesg 命令查看到来自 sys_my_syscall() 的输出。

```
[   40.843174] Message from the simple syscall.
```

3.2. 方法二

首先加载内核模块 dynamic_syscall_loader.ko ,然后加载 simple_syscall.ko。查看内核日志可以看到系统调用的插入位置（即对应的系统调用号）。
编写并运行测试程序,同样可以通过 dmesg 命令查看到来自 sys_my_syscall() 的输出。



4. 心得与收获

在本次实验中，通过阅读 Linux 系统内核的代码，使我对系统调用的实现机制有了更深的理解，明白了系统调用实现的整个流程，掌握了 Ubuntu 系统编译、安装内核的步骤。在探究过程中，掌握了如何编写简单的 Linux 内核模块，了解了 Linux 的内存分页保护机制、内存地址映射机制。

5. 附录

5.1. 实验环境

- 发行版：Ubuntu 14.04 LTS
- 内核版本：Linux x86_64 4.2.8
- GNU C/C++ 编译器版本：4.8.4

5.2. 实验中涉及的主要命令

5.2.1. 安装源代码包

```
sudo apt-get update                              # 更新源
sudo apt-get install build-essential             # 安装编译工具组
sudo apt-get install linux-image-$(uname -r)     # 安装源代码包
sudo apt-get install vim                         # 安装 vim 编辑
器
```

5.2.2. 加载内核模块

```
insmod dynamic_syscall_loader.ko                 # 加
载 dynamic_syscall_loader.ko 内核模块
insmod simple_syscall.ko                         # 加
载 simple_syscall.ko 内核模块
```