Linux 下系统调用原理解析及增加系统调用的方法

The principle of system call and the method of add new system call

胡盼盼　（华中科技大学计算机学院 湖北武汉 430074）

摘要：系统调用是应用程序和操作系统之间进行交互的接口。程序的运行离不开系统调用。本文讨论讨论了 linux 下系统调用的实现机理，并以一个简单的例子说明了在 linux 下增加系统调用的实现方法，本文的讨论基于 linux 2.4 内核版本。

关键词：linux　系统调用

1　Linux 系统调用概述

系统调用是内核提供的，功能十分强大的一系列函数。这些函数在 Linux 的内核中实现，用来完成一些系统级的功能。通过一定的方式呈现给用户，是用户程序与内核交互的一个接口，在用户程序每调用一次系统调用函数，都将进行一次用户空间到内核空间的转换。如 Linux 中常用的 open，read 等函数，都属于系统调用。

在 Linux 系统中，系统调用是作为一种异常类型实现的，它将执行相应的机器代码指令来产生异常信号。在执行系统调用异常指令时，自动地将系统切换为核心态（模式切换，mode switch），并安排异常处理程序的执行。用户态的程序只有通过门（gate）陷入（trap）到系统内核中去（执行 int 指令），才能执行一些具有特权的内核函数。系统调用完成后，系统执行另一组特征指令（iret 指令）将系统返回到用户态，控制权返回给进程。Linux 用来实现系统调用异常的实际指令是：int $0x80，这一指令使用中断/异常向量号 128（即 16 进制的 80）将控制权转移给内核（进行模式切换）。为达到在使用系统调用时不必用机器指令编程，在标准的 C 语言库中为每一系统调用提供了一段短的子程序，完成机器代码的编程工作。这段子程序所要做的工作只是将送给系统调用的参数加载到 CPU 寄存器中，接着执行 int $0x80 指令。然后运行系统调用，系统调用的返回值将送入 CPU 的一个寄存器中，标准的库子程序取得这一返回值，并将它送回用户程序。

2　Linux 系统调用结构分析

2.1　部分重要的宏的说明

我们以 Linux2.4 内核为例，跟系统调用相关的内核代码主要有：

- /usr/src/linux－2.4/arch/i386/kernel/entry. S
- /usr/src/linux－2.4/arch/i386/kernel/trap. c
- /usr/src/linux－2.4/arch/i386/kernel/unistd. h

这三个内核文件中定义了系统调用所需的一些信息。下面将说明这些代码文件中与系统调用相关的一些重要的宏定义，便于后面对系统调用过程的说明。

SAVE_ALL

SAVE_ALL 宏定义在 entry. s 中，代码如下：

```
#define SAVE_ALL \
cld; \
pushl % es; \
pushl % ds; \
pushl % eax; \
pushl % ebp; \
pushl % edi; \
pushl % esi; \
pushl % edx; \
pushl % ecx; \
pushl % ebx; \
movl $ (__KERNEL_DS),% edx; \
movl % edx,% ds; \
```

```
movl % edx,% es;
```

我们可以看到 SAVE_ALL 宏主要是保存寄存器信息,即进行现场保留。

```
RESTORE_ALL
```

RESTORE_ALL 也定义在 entry. S 中。其定义如下:

```
popl % ebx; \
popl % ecx; \
popl % edx; \
popl % esi; \
popl % edi; \
popl % ebp; \
popl % eax; \
popl % ds; \
popl % es; \
addl $4,% esp; \
iret; \
```

可以看出,RESTORE_ALL 与 SAVE_ALL 的过程是刚好相反的,主要完成现场恢复。iret 是返回指令。

```
__NR_syscallname NNN
```

这个宏定义在 unistd. h 中,其中 syscallname 表示系统调用函数名,NNN 表示系统调用序号。

```
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
.
.
.
```

Linux 系统给每个系统调用都编了号,当执行一个系统调用时,将按照系统调用号来索引系统调用。

```
_syscallN ( type, name, parameter1 type, parameter1, ……. )
```

这是一个带参数宏,定义在 unistd. h 中,用来展开系统调用,N 是可变的,表示系统调用函数的参数个数。宏参数分别为系统调用函数的返回类型,系统调用函数名,系统调用参数类型,参数名。可以加多个参数。其展开后如下(假设此系统调用没有参数):

```
#define _syscall0 ( type,name )
type name( void )
{
    Long __res;
    __asm__ volatile ( "int $0x80
        : "=a" (__res)
        :"" (__NR_##name) );
    __syscall_return( type,__res) ;
}
```

系统调用表 sys_call_table

在 entry. S 文件中还有一个重要的表就是系统调用表,系统调用表保存着所有系统调用的函数指针,以方便进行系统调用的索引。其定义如下:

```
ENTRY( sys_call_table )
    . long SYMBOL_NAME( sys_ni_syscall )
    . long SYMBOL_NAME( sys_exit )
    . long SYMBOL_NAME( sys_fork )
    . long SYMBOL_NAME( sys_read )
        .
        .
        .
```

2.2 系统调用过程解析

我们以 getuid( ) 这个系统调用为示例. getuid( ) 的功能是获取当前进程的 ID 号。测试程序如下:

```
#include < linux/unistd. h >
int main( ) {
    int i = getuid( ) ;
    printf( "my uid is : % d\n",i) ;
}
```

这个程序调用 getuid( ) 函数,来获得当前进程的 ID 号,并在终端输出。当程序调用 getuid( ) 这个函数时,便会调用头文件 unistd. h 中的定义的_syscall0 (int , getuid) 这个宏。这是个带参数宏,在 unistd. h 头文件中定义。通过查看 unistd. h 头文件,会发现 getuid 会被展开成以下代码:

```
getuid
int getuid( void) {
    long __res;
    __asm__ volatile ( "int $0x80" )
        : "=a" (__res)
        :"" (__NR_getuid) ) ;
    __syscall_return( int,__res) ;
}
```

从中可以看出，程序通过调用宏 __NR_getuid 得到调用号，放到寄存器 eax 中，然后执行中断指令，“int $0x80”。

我们继续深入下去，看看 int $ 0x80 后面到底做了什么。因为这是一条软中断指令，那就要看看系统规定的这条中断指令的处理程序是什么。在 traps. c 文件中，有这样一行代码：

set_ system _ gate (SYSCALL _ VECTOR, &system _ call)，其中宏 SYSCALL_VECTOR 在内核中是被定义成 0x80 的。所以，在进入 int $0x80 中断后，系统将会自动进行模式和堆栈切换，然后将控制转移到 system_call 的入口点。

在 entry. S，中，我们可以找到 system_call 的入口点：

```
ENTRY( system_call)
    pushl % eax
    SAVE_ALL
    GET_CRRENT( % ebx)
    testb  $0x02,tsk_ptrace( % ebx)
    jne tracesys
    cmpl  $ (NR_syscalls) ,% eax
    jae badsys
    call \ * SYMBOL_NAME( sys_call_table) ( ,% eax,
4)
    movl % eax,EAX( % ebx)
ENTRY( ret_from_sys_call)
    cli
cmpl  $0,need_resched( % ebx)
    jnereschedule
    cmpl  $0,sigpending( % ebx)
    jne sygnal_return
Retore_all:
    RESTORE_ALL
```

这段代码的主要功能就是保留系统调用号拷贝，SAVE_ALL 保存现场，得到当前用户进程结构指针，并保存在 ebx 中，检测系统调用号是否合法，最后根据系统调用号索引 sys_call_table，并找到最终的内核函数 sys_getuid16。我们在 linux 终端下打开../kernel/uid16. c，便可看到这个系统调用函数的原始定义。

```
Asmlinkage long sys_getuid16( void) {
    returnhigh21owuid( current - > uid) ;
}
```

这个内核系统调用函数仅返回一个当前进程的 uid。然后将得到的 uid 保存在 eax 中，最后调用 RESTORE_ALL 宏恢复现场。最后进程返回到用户态，用户程序最终执行完成。整个系统调用过程也就完成。整个过程可以用下图来表示：

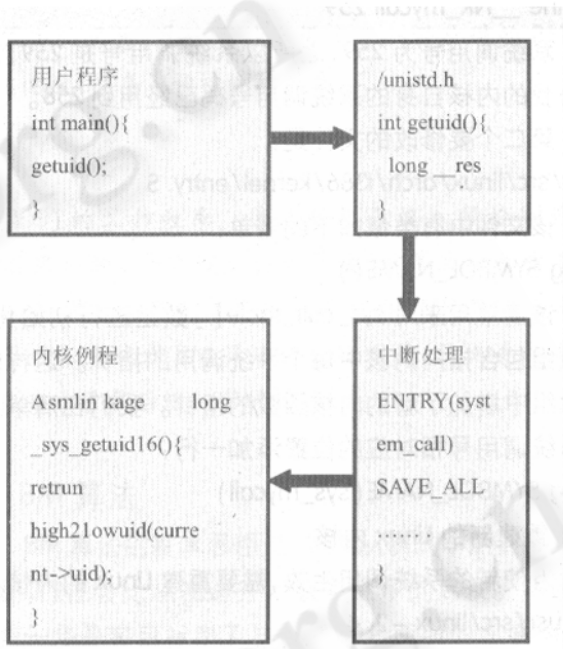

图 1

3 增加系统调用

3.1 添加源代码

首先是编写加到内核中的源程序，即将要加到一个内核文件中去的一个函数，该函数的名称应该是新的系统调用名称前面加上 sys_标志。在/usr/src/linux/kernel/sys. c 文件中添加源代码，如下所示：

```
asmlinkage int sys_mycall( int number)
{
    return number;
}
```

作为一个最简单的例子，我们新加的系统调用仅仅返回一个整型值。

3.2 连接新的系统调用

添加新的系统调用后，下一个任务是使 Linux 内核的其余部分知道该程序的存在。为了从已有的内核程

序中增加到新的函数的连接，需要编辑前面提到过的两个文件。

在我们所用的 Linux 内核版本(2.4.18)中，第一个要修改的文件是：

/usr/src/linux/include/asm－i386/unistd.h

文件中找到系统调用号的宏定义，然后在后面加上一行：

#define __NR_mycall 259

系统调用号为 259，之所以系统调用号是 259，是因为我的内核自身的系统调用号码已经用到 258。

第二个要修改的文件是：

/usr/src/linux/arch/i386/kernel/entry.S

该文件中有类似如下的清单：

.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。我们在清单上与系统调用号相对应的位置添加一行：

.long SYMBOL_NAME(sys_mycall)

3.3 重建新的 Linux 内核

为使新的系统调用生效，需要重建 Linux 的内核。

cd /usr/src/linux－2.4

超级用户在当前工作目录(/usr/src/linux)下，才可以重建内核。

#make xconfig，进行内核配置。

#make dep，检测关联性。

#make clean，清除不用的组件。

#make bzImage，生成内核映像文件。

编译完毕后，系统生成一可用于安装的、压缩的内核映象文件：/usr/src/linux/arch/i386/boot/bzImage

3.4 用新的内核启动系统

将/usr/src/linux/arch/i386/boot/bzImage 拷贝到/boot/bzImage

配置启动文件：

假设所用的启动文件为 grub，修改/boot/grub/grub.conf，添加新的引导内核：

```
title Linuxtest
root(hd0,4)
kernel /boot/bzImage ro root=LABEL=/
initrd /boot/initrd－2.4.20－8.img
```

至此，新的 Linux 内核已经建立，新添加的系统调用已成为操作系统的一部分，重新启动，进入 Linux-test，就可以在应用程序中使用该系统调用了。

3.5 使用新的系统调用

在应用程序中使用新添加的系统调用 mycall。同样为实验目的，我们写了一个简单的例子 test.c。

```
#include <errno.h>
#include </usr/src/linux/include/asm－i386/unistd.h>
_syscall1(int,mycall,int,ret) //系统调用函数声明
main( )
{
printf("%d\n",mycall(100));
}
```

编译该程序：

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

执行：

```
#./test
```

结果：

```
#100
```

4 结束语

本文中我们解析了 linux 下系统调用的整个实现过程，通过一个系统调用的实例分析一个系统调用在内核的完整实现，加深了对系统调用的认识。然后又通过一个简单的例子，叙述了在 linux 下增加系统调用的方法。Linux 下的系统调用具有强大的功能。通过增加系统调用，我们可以根据自己的需求，来给内核增加特定的功能，方便我们的使用。

参考文献

1 陈莉君，<<Linux 操作系统内核分>>[M]北京>>:人民邮电出版社，2003.

2 毛德操、胡希明，<<Linux 内核源代码情景分析>>[M]，杭州:浙江大学出版社，2001.

3 李善平、刘文峰、王伟波等，<<Linux 内核源代码分析大全>>[M] 北京:机械工业出版社，2002.

4 李善平、陈文智等，<<边干边学－linux 内核指导>>[M]，杭州:浙江大学出版社.