**主要内容：**

什么是系统调用

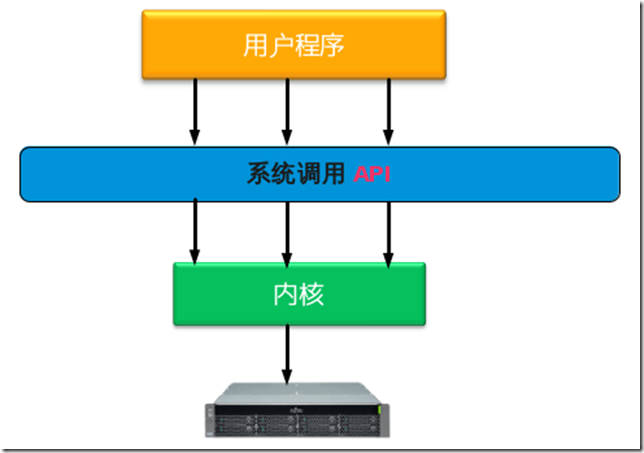
Linux上的系统调用实现原理

一个简单的系统调用的实现

### ****1. 什么是系统调用****

简单来说，系统调用就是用户程序和硬件设备之间的桥梁。用户程序在需要的时候，通过系统调用来使用硬件设备。

用户程序，系统调用，内核，硬件设备的调用关系如下图：

[](http://images.cnblogs.com/cnblogs_com/wang_yb/201209/201209170957496310.png)

### ****2. Linux上的系统调用实现原理****

要想实现系统调用，主要实现以下几个方面：

1. 通知内核调用一个哪个系统调用
2. 用户程序把系统调用的参数传递给内核
3. 用户程序获取内核返回的系统调用返回值

下面看看Linux是如何实现上面3个功能的。

#### **2.1 通知内核调用一个哪个系统调用**

每个系统调用都有一个系统调用号，系统调用发生时，内核就是根据传入的系统调用号来知道是哪个系统调用的。

系统调用号定义在内核代码：arch/alpha/include/asm/unistd.h 中，可以看出linux的系统调用不是很多。

#### **2.2 用户程序把系统调用的参数传递给内核**

一般的系统调用都是通过C库(最常用的是glibc库)来访问的，Linux内核提供一个从用户程序直接访问系统调用的方法。

参见内核代码：arch/cris/include/arch-v10/arch/unistd.h

里面定义了6个宏，分别可以调用参数个数为0～6的系统调用

\_syscall0(type,name)

\_syscall1(type,name,type1,arg1)

\_syscall2(type,name,type1,arg1,type2,arg2)

\_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)

\_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)

\_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)

\_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)

超过6个参数的系统调用很罕见，所以这里只定义了6个。

### ****3. 一个简单的系统调用的实现****

了解了Linux上系统调用的原理，下面就可以自己来实现一个简单的系统调用。

#### **3.1 环境准备**

为了不破坏现有系统，我是用虚拟机来实验的。

主机：fedora16 x86\_64系统 + kvm(一种虚拟技术，就像virtualbox，vmware等)

虚拟机: 也是安装fedora16 x86\_64系统(通过virt-manager很容易安装一个系统)

下载内核源码：www.kernel.org  下载最新的就行

#### **3.2 修改内核源码中的相应文件**

主要修改以下文件：

arch/x86/ia32/ia32entry.S

arch/x86/include/asm/unistd\_32.h

arch/x86/include/asm/unistd\_64.h

arch/x86/kernel/syscall\_table\_32.S

include/asm-generic/unistd.h

include/linux/syscalls.h

kernel/sys.c

我在sys.c中追加了2个函数:sys\_foo和sys\_bar

如果是在x86\_64的内核中增加一个系统调用，只需修改 arch/x86/include/asm/unistd\_64.h，比如sys\_bar。

修改内容参见下面的diff文件：

diff -r new/arch/x86/ia32/ia32entry.S old/arch/x86/ia32/ia32entry.S

855d854

< .quad sys\_foo

diff -r new/arch/x86/include/asm/unistd\_32.h old/arch/x86/include/asm/unistd\_32.h

357d356

< #define \_\_NR\_foo 349

361c360

< #define NR\_syscalls 350

---

> #define NR\_syscalls 349

diff -r new/arch/x86/include/asm/unistd\_64.h old/arch/x86/include/asm/unistd\_64.h689,692d688

< #define \_\_NR\_foo 312

< \_\_SYSCALL(\_\_NR\_foo, sys\_foo)

< #define \_\_NR\_bar 313

< \_\_SYSCALL(\_\_NR\_bar, sys\_bar)

diff -r new/arch/x86/kernel/syscall\_table\_32.S old/arch/x86/kernel/syscall\_table\_32.S

351d350

< .long sys\_foo

diff -r new/include/asm-generic/unistd.h old/include/asm-generic/unistd.h694,695d693

< #define \_\_NR\_foo 272

< \_\_SYSCALL(\_\_NR\_foo, sys\_foo)

698c696

< #define \_\_NR\_syscalls 273

---

> #define \_\_NR\_syscalls 272

diff -r new/kernel/sys.c old/kernel/sys.c1920,1928d1919

<

< asmlinkage long sys\_foo(void)

< {

< return 1112223334444555;< }

< asmlinkage long sys\_bar(void)

< {

< return 1234567890;< }

#### **3.3 编译内核**

#cd linux-3.2.28

#make menuconfig (选择要编译参数，如果不熟悉内核编译，用默认选项即可)

#make all (这一步真的时间很长......)

#make modules\_install

#make install (这一步会把新的内核加到启动项中)

#reboot (重启系统进入新的内核)

#### **3.4 编写调用的系统调用的代码**

#include <unistd.h>

#include <sys/syscall.h>

#include <string.h>

#include <stdio.h>

#include <errno.h>

#define \_\_NR\_foo 312

#define \_\_NR\_bar 313

int main()

{

printf ("result foo is %ld\n", syscall(\_\_NR\_foo));

printf("%s\n", strerror(errno));

printf ("result bar is %ld\n", syscall(\_\_NR\_bar));

printf("%s\n", strerror(errno));

return 0;

}

编译运行上面的代码：

#gcc test.c -o test

#./test

运行结果如下：

result foo is 1112223334444555

Success

result bar is 1234567890

Success