Linux下增加系统调用的二种方法[zz]

**1.linux&系统调用的基本原理**  
  
   linux的系统调用形式与POSIX兼容，也是一套C语言函数名的集合。然而，linux系统调用的内部实现方式却与DOC的INT 21H相似，它是经过INT 0X80H软中断进入后，再根据系统调用号分门别类地服务。   
  
  
从系统分析的角度，linux的系统调用涉及4个方面的问题。   
(1)与系统调用有关的数据结构和函数   
   函数名以“sys\_”开头，后跟该系统调用的名字。例如，系统调用fork()的响应函数是sys\_fork()(见kernel/fork.c),exit()的响应函数是sys\_exit()(见kernel/fork.c)。   
  
   文件include/asm/unisted.h为每个系统调用规定了唯一的编号。假设用name表示系统调用的名称，那么系统调用号与系统调用响应函数 的关系是：以系统调用号\_NR\_name作为下标，可找出系统调用表sys\_call\_table(见arch/i386/kernel /entry.S)中对应表项的内容，它正好 是该系统调用的响应函数sys\_name的入口地址。系统调 用表sys\_call\_table记录了各sys\_name函数在表中的位 置，共190项。有了这张表，就很容易根据特定系统调用 在表中的偏移量，找到对应的系统调用响应函数的入口地址。系统调用表共256项，余下的项是可供用户自己添加的系统调用空间。   
  
(2)进程的系统调用命令转换为INT 0x80中断的过程   
       宏定义\_syscallN()见include/asm/unisted.h)用于系统调用的格式转换和参数的传递。N取0~5之间的整数。 参数个数为N的系统调用由\_syscallN()负责格式转换和参数传递。系统调用号放入EAX寄存器，启动INT 0x80 后，规定返回值送EAX寄存器。   
  
(3)系统调用功能模块的初始化   
   对系统调用的初始化也就是对INT 0x80的初始化。系统启动时，汇编子程序setup\_idt(见arch/i386/kernel/head.S)准备了1张256项的idt表，由 start\_kernel()(见 init/main.c),trap\_init()(见arch/i386/kernel/traps.c)调用的C语言宏定义 set\_system\_gate(0x80,&system\_call)(见 include/asm/system.h)设置0x80号软中断的服务程序为 system\_call(见arch/i386/kernel/entry.S),system.call就是所有系统调用的总入口。   
  
(4)内核如何为各种系统调用服务   
       当进程需要进行系统调用时，必须以C语言函数的形式写一句系统调用命令。该命令如果已在某个头文件 中由相应的\_syscallN()展开，则用户程序必须包含该文 件。当进程执行到用户程序的系统调用命令时，实际上执 行了由宏命令\_syscallN()展开的函数。系统调用的参数 由各通用寄存器传递，然后执行INT 0x80，以内核态进 入入口地址system\_call。   
  
(5)ret\_from\_sys\_call  
   以ret\_from\_sys\_call入口的汇编程序段在linux进  
程管理中起到了十分重要的作用。所有系统调用结束前以及大部分中断服务返回前，都会跳转至此处入口地址。 该段程序不仅仅为系统调用服务，它还处理中断嵌套、CPU调度、信号等事务。   
  
  
**2.通过修改内核源代码添加系统调用**  
   通过以上分析linux系统调用的过程， 将自己的系统调用加到内核中就是一件容易的事情。下面介绍一个实际的系统调用， 并把它加到内核中去。要增加的系统调用是：inttestsyscall()，其功能是在控制终端屏幕上显示hello world， 执行成功后返回0。  
1编写inttestsyscall()系统调用  
编写一个系统调用意味着要给内核增加1个函数，将新函数放入文件kernel/sys.c中。新函数代码如下：  
asmlingkage sys\_testsyscall()  
{   console\_print("hello world\n");  
return 0;  
}  
  
2连接新的系统调用  
编写了新的系统调用过程后，下一项任务是使内核的其余部分知道这一程序的存在，然后重建包含新的系统调用的内核。  
为了把新的函数连接到已有的内核中去， 需要编辑2个文件：  
  
1).inculde/asm/unistd.h在这个文件中加入  
#define\_NR\_testsyscall 191  
  
2).are/i386/kernel/entry.s这个文件用来对指针数组初始化，在这个文件中增加一行：  
.long SYMBOL\_NAME(\_sys\_tsetsycall)  
将.rept NR\_syscalls-190改为NR\_SYSCALLS-191,然后重新奖励和运行新内核。  
  
3).使用新的系统调用  
在保证的C语言库中没有新的系统调用的程序段，必须自己建立其代码如下  
#inculde<linux/unistd.h>  
\_syscall0(int,testsyscall)  
main()  
{    
   tsetsyscall();  
}  
      在这里使用了\_syscall0（）宏指令，宏指令本身在程序中将扩展成名为syscall()的函数，它在main()函数内部加以调用。 在testsyscall()函数中， 预处理程序产生所有必要的机器指令代码，包括用系统调用参数值加载相应的cpu寄存器， 然后执行int 0x80中断指令。   
  
  
  
**3.利用内核模块添加系统调用**      
   模块是内核的一部分，但是并没有被编译到内核里面去。它们被分别编译并连接成一组目标文件， 这些文件能被插入到正在运行的内核，或者从正在运行的内核中移走。内核模块至少必须有2个函数：int\_module和cleanup\_module。第 一个函数是在把模块插入内核时调用的； 第二个函数则在删除该模块时调用。  
   由于内核模块是内核的一部分，所以能访问所有内核资源。根据对linux系统调用机制的分析， 如果要增加系统调用，可以编写自己的函数来实现，然后在sys\_call\_table表中增加一项，使该项中的指针指向自己编写的函数， 就可以实现系统调用。下面用该方法实现在控制终端上打印“hello world” 的系统调用testsyscall()。  
  
1）编写系统调用内核模块  
#inculde(linux/kernel.h)  
#inculde(linux/module.h)  
#inculde(linux/modversions.h)  
#inculde(linux/sched.h)  
#inculde(asm/uaccess.h)  
#define\_NR\_testsyscall 191  
extern viod \*sys\_call+table[];  
asmlinkage int testsyscall()  
{ printf("hello world\n");  
return 0;  
}  
int init\_module()  
{ sys\_call\_table[\_NR\_tsetsyscall]=testsyscall;  
printf("system call testsyscall() loaded success\n");  
    return 0;  
    }  
void cleanup\_module()  
{  
  
}  
  
2）使用新的系统调用#define<linux/unistd.h>  
#define\_NR\_testsyscall 191  
\_syscall0(int,testsyscall)  
main()  
{   
    testsyscall();  
}  
  
3)编译内核模块并插入内核  
       编译内核的命令为：gcc -Wall -02 -DMODULE -D\_KERNEL\_-C syscall.c  
   -Wall通知编译程序显示警告信息；参数-02 是关于代码优化的设置， 内核模块必须优化； 参数-D\_LERNEL通知头文件向内核模块提供正确的定义； 参数-D\_KERNEL\_通知头文件， 这个程序代码将在内核模式下运行。编译成功后将生成 syscall.0文件。最后使用insmod syscall.o命令将模块插入内核后即可使用增加的系统调用。  
  
    比较以上二种方法，笔者认为采用内核模块的方法较好。因为这种方法可省去编译新内核并用新内核重新 启动的麻烦，这一优点对于代码的调试是非常有价值的， 可以节省大量时间。