操作系统 Operating Systems

L5. 系统调用的实现

System Call?

授课教师：李治军

lizhijun_os@hit.edu.cn
综合楼404室

系统调用的直观实现 问题+直观想法…

- 实现一个whoami系统调用
  - 用户程序调用whoami， 一个字符串“lizhijun”放在操作系统中(系统引导时载入)，取出来打印
  - 不能随意的调用数据，不能随意的jmp。
  - 可以看到root密码，可以修改它…
  - 可以通过显存看到别 100: 人word里的内容…

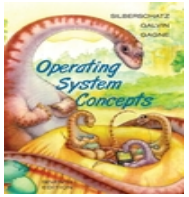
```
main()
{  whoami();}
```

用户程序

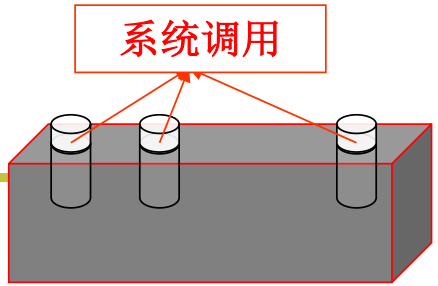
```
whoami()
{
  printf(100, 8);
}

“lizhijun”
```

都在内存中，这内存不都是我买的吗…

为什么需要系统调用来进行隔离？
因为一些设计到系统管理的关键信息不能够让用户直接访问，防制用户进行篡改导致系统崩溃
一些核心的功能必须封装起来。
同时不用APP的内存空间也不能够相互进行随意访问。



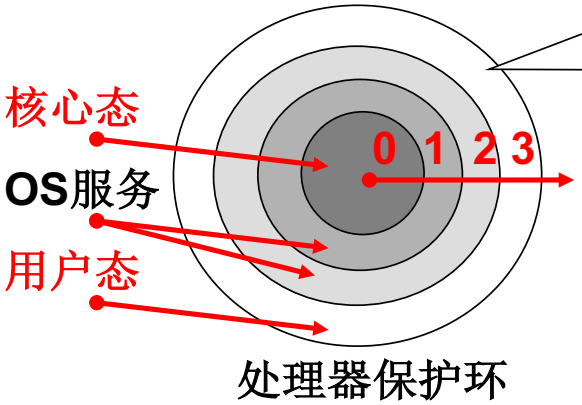
内核(用户)态，内核(用户)段



- 将内核程序和用户程序隔离!!!

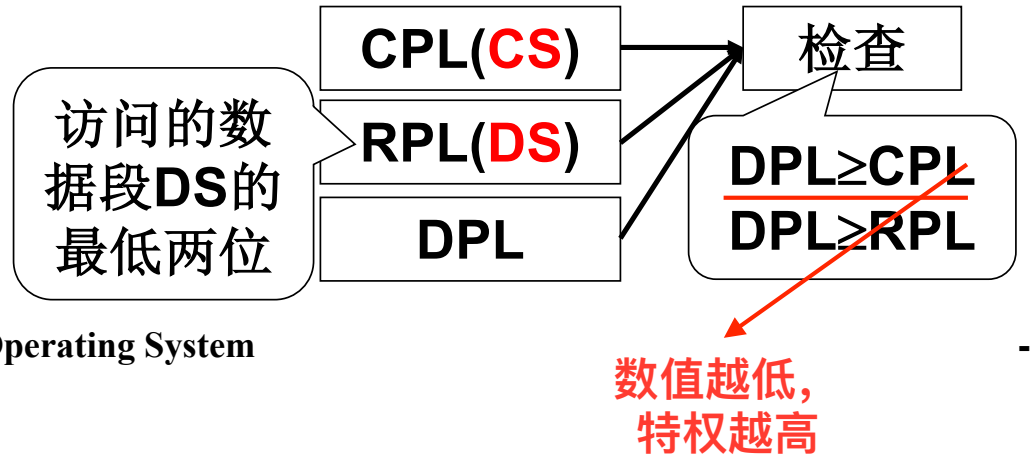
由硬件进行判断
特权关系是否满足

- 区分内核态和用户态：一种处理器“硬件设计”



当前程序执行在什么态(哪层环)?由于CS:IP是当前指令，所以用CS的最低两位来表示: 0是内核态，3是用户态

- 内核态可以访问任何数据，用户态不能访问内核数据
- 对于指令跳转也一样实现了隔离…



CPL表示当前特权级，3表示用户态

DPL表示目标段的特权级
存储在GDT表中，初始化的时候，
操作系统所在的内存被置为了内核态

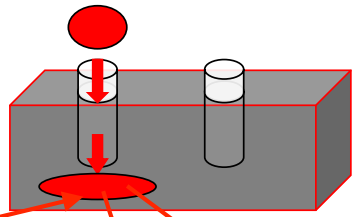
但是int 0x80之后的CPL依然是3，这样才能够让用户调用

硬件提供了“主动进入内核的方法”

只有通过设计的一些中断才能进入内核
不能通过jump，mov等

- 对于Intel x86，那就是中断指令int
  - int指令将使CS中的CPL改成0，“进入内核”
  - 这是用户程序发起的调用内核代码的唯一方式

此时，CPL=3而DPL=0



- 系统调用的核心：

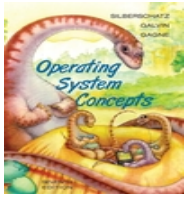
由谁做？库函数!

(1) 用户程序中包含一段包含int指令的代码

(2) 操作系统写中断处理，获取想调程序的编号

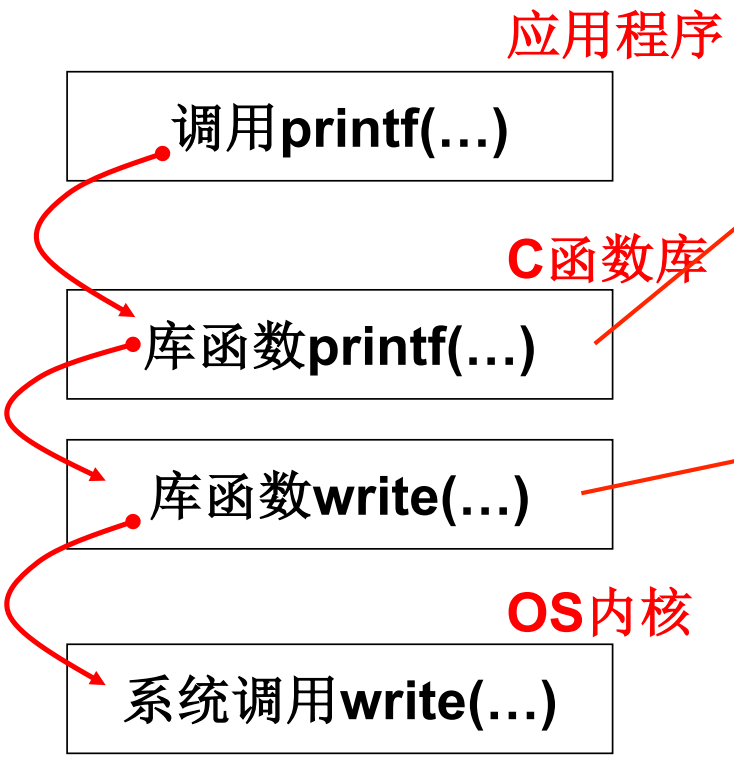
(3) 操作系统根据编号执行相应代码

open代码最终转为一段包含int指令的代码，调用中断



系统调用的实现

库函数将格式化输出转为对应的count等

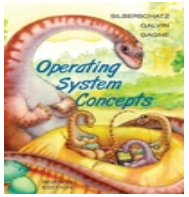


- 最终展开成包含int指令的代码…

```
#include <unistd.h>          在linux/lib/write.c中
_syscall3(int, write, int, fd, const char
*buf, off_t, count)
```

```
#define _syscall3(type, name, ...) type
name(...) \                  在linux/include/unistd.h中
{ __asm__ ("int 0x80" :"=a"(__res)...}
```

只能通过int 0x80才可能进入操作系统内核



Linux系统调用的实现细节!

将关于write的故事完整的讲完...

一段宏，将对应位置替换为传入的参数

_syscall3表示有3个参数

在linux/include/unistd.h中

```
#define _syscall3(type,name,atype,a,btype,b,ctype,c)\
type name(atype a, btype b, ctype c) \
{  long __res;\
   __asm__ volatile("int 0x80":"=a"(__res):""(__NR_##name),
"b"((long)(a)),"c"((long)(b)),"d"((long)(c))); if(__res>=0) return
(type)__res; errno=-__res; return -1;}
```

C内嵌汇编

将系统调用号赋给eax寄存器
用于确定之后执行哪一个处理程序

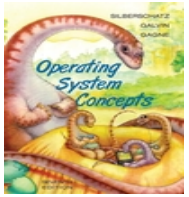
- 显然，__NR_write是系统调用号，放在eax中

在linux/include/unistd.h中

```
#define __NR_write 4 //一堆连续正整数(数组下标，函数表索引)
```

将一堆参数(包括系统调用号等)赋给对应的寄存器，
然后执行int 0x80,进入内核态。
剩下的工作就是操作系统代码的事儿了

- 同时eax也存放返回值，ebx，ecx，edx存放3个参数



int 0x80中断的处理

系统初始化的时候设置好的

调用int 0x80，然后操作系统找到80对应的中断处理函数。

```
void sched_init(void)
{ set_system_gate(0x80,&system_call); }
```

- 显然，set_system_gate用来设置0x80的中断处理

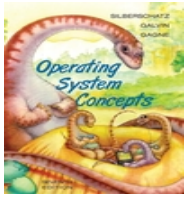
```
在linux/include/asm/system.h中
#define set_system_gate(n, addr) \
_set_gate(&idt[n],15,3,addr); //idt是中断向量表基址
#define _set_gate(gate_addr, type, dpl, addr)\
__asm__("movw %%dx,%%ax\n\t" "movw %0,%%dx\n\t"\
"movl %%eax,%1\n\t" "movl %%edx,%2":\
:"i"((short)(0x8000+(dpl<<13)+type<<8)),"o"(*(( \
char*)(gate_addr))),"o"(*(4+(char*)(gate_addr))),\
"d"((char*)(addr)),"a"(0x00080000))
```

中断就是跳到另一个地方，然后执行

进入到内核态之后，根据系统调用号到IDT表中查找对应的中断处理程序的入口，然后指定对应的中断处理程序即可。中断处理结束就返回到用户的程序。同时对应的返回值存放在固定的寄存器中。

4	处理函数入口点偏移	P	DPL	01110	
0	段选择符 8	处理函数入口点偏移			

IDT表



int 0x80的DPL也是3，这样用户才能够跳进来
跳进来之后再跳了一次，8的末尾是00， 所以变成了内核态
返回之后还会执行指令将末尾变为3，返回到用户态

中断处理程序: system_call

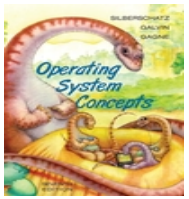
```
在linux/kernel/system_call.s中
nr_system_calls=72
.globl _system_call
_system_call: cmpl $nr_system_calls-1,%eax
  ja bad_sys_call
  push %ds  push %es  push %fs
  pushl %edx  pushl %ecx  pushl %ebx  //调用的参数
  movl $0x10,%edx  mov %dx,%ds  mov %dx,%es //内核数据
  movl $0x17,%edx  mov %dx,%fs  //fs可以找到用户数据
  call _sys_call_table(,%eax,4) //a(,%eax,4)=a+4*eax
  pushl %eax //返回值压栈，留着ret_from_sys_call时用
  ... //其他代码
ret_from_sys_call: popl %eax, 其他pop, iret
```

eax中存放的是系统调用号

```
__asm__ volatile("int 0x80":"=a"(__res)
```

- _sys_call_table+4*%eax就是相应系统调用处理函数入口

每个函数入口地址是4个字节，
所以需要将系统调用号乘4到对应的函数指针



_sys_call_table

在include/linux/sys.h中

```
fn_ptr sys_call_table[]=
{sys_setup, sys_exit, sys_fork, sys_read, sys_write,
  ...};
```

sys_call_table是一个全局函数数组

sys_write对应的数组下标为4，__NR_write=4

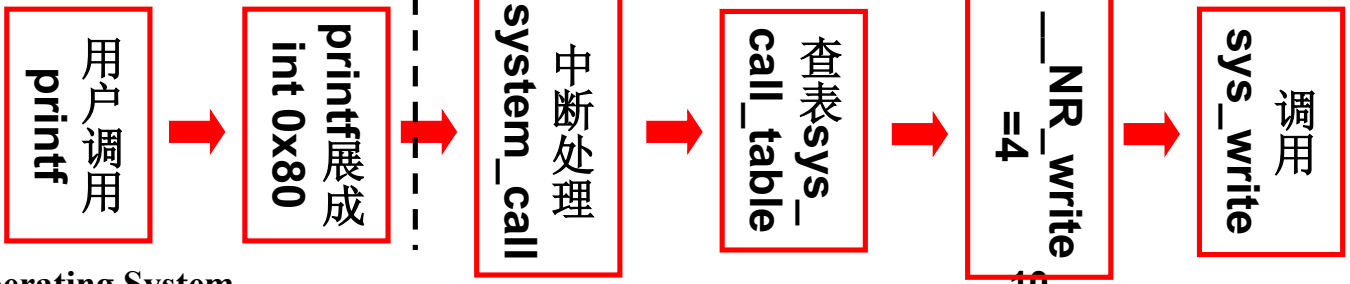
在include/linux/sched.h中

```
typedef int (fn_ptr*)();
```

- call _sys_call_table(,%eax,4)就是call sys_write

eax=4，函数入口地址长度也为4

故事结束!



int 0x80是一个过渡，
是操作系统所有的系统服务的统一入口
同时也是用户态能够跳转到的一个指令

```
main()
{  whoami();}
```

```
whoami()
{
  printf(100, 8);
}
```

100: “lizhijun”

```
main()
{   eax = 72;
    int 0x80;  }
```

用户函数调用int中断

CPL = 3

```
_system_call:
 call sys_whoami
//sys_call_table
+ eax*4
```

int 0x80用户函数传入的编号
确定要到哪个系统调用执行
通过查表找到对应系统调用的地址

CPL = 0

```
sys_whoami()
{
  printk(100, 8);
}
```

操作系统在内核态
中完成对应的操作，完成之后
通过上述调用的路径返回用户态。

100: “lizhijun”