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…

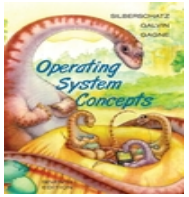
```
main()
{  whoami();}
```

用户程序

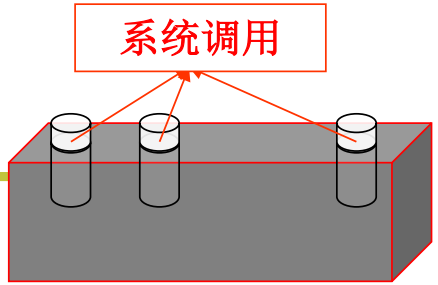
```
whoami()
{
  printf(100, 8);
}

“lizhijun”
```

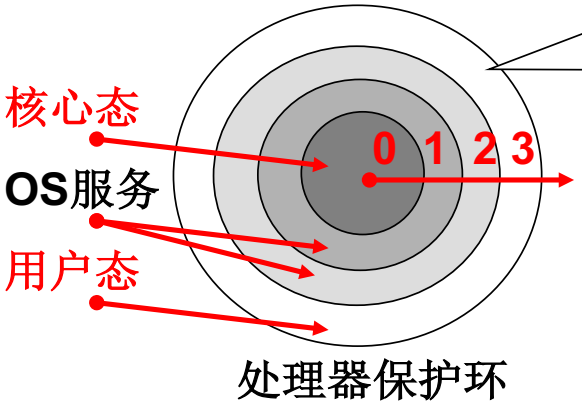
都在内存中，这内存不都是我买的吗…



内核(用户)态，内核(用户)段

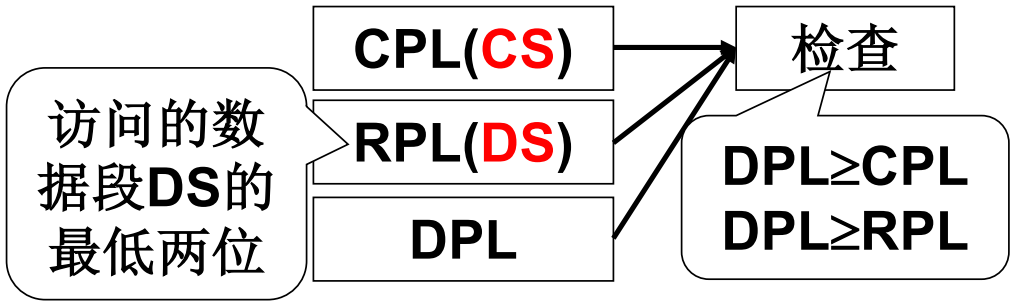


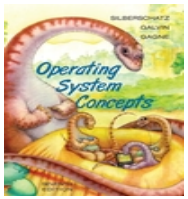
- 将内核程序和用户程序隔离!!!
  - 区分内核态和用户态：一种处理器“硬件设计”



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…

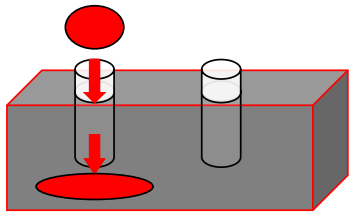




硬件提供了“主动进入内核的方法”

- 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：

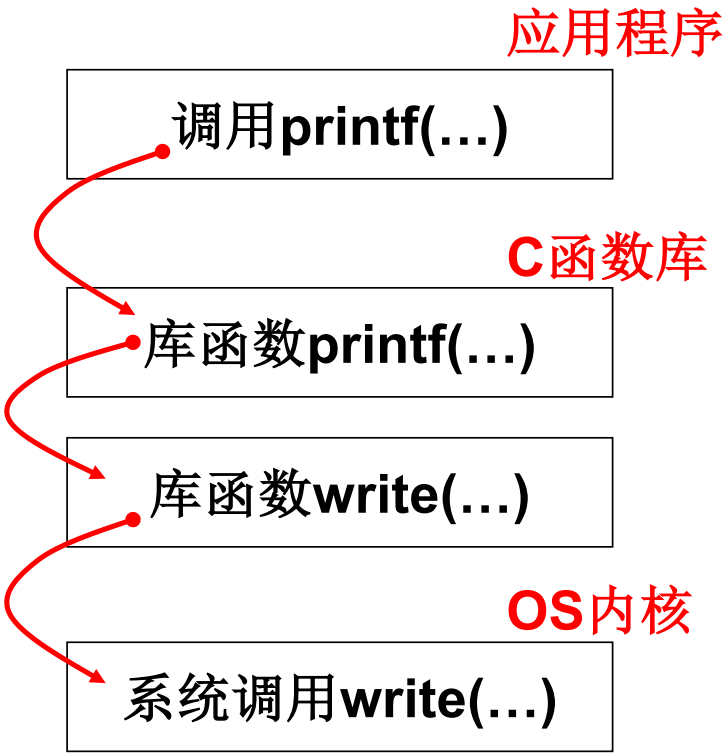
由谁做？库函数!

(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

系统调用的实现



- 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)...}
```

Linux系统调用的实现细节!

将关于write的故事完整的讲完...

_syscall3表示有3个参数

在linux/include/unistd.h中

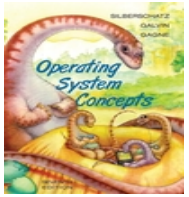
```
#define _syscall3(type,name,atype,a,btype,b,ctype,c)\
type name(atype a, btype b, ctype c) \
{  long __res;\
   __asm__ volatile("int 0x80":"=a"(__res):""(__NR_##name),
"b"((long)(a)),"c"((long)(b)),"d"((long)(c))); if(__res>=0) return
(type)__res; errno=-__res; return -1;}
```

- 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)
```

- 同时eax也存放返回值，ebx，ecx，edx存放3个参数



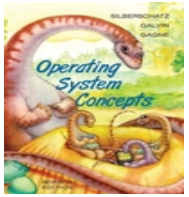
int 0x80中断的处理

```
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))
```

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



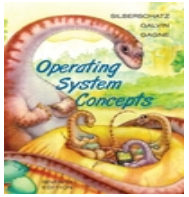
中断处理程序: system_call

```
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



_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

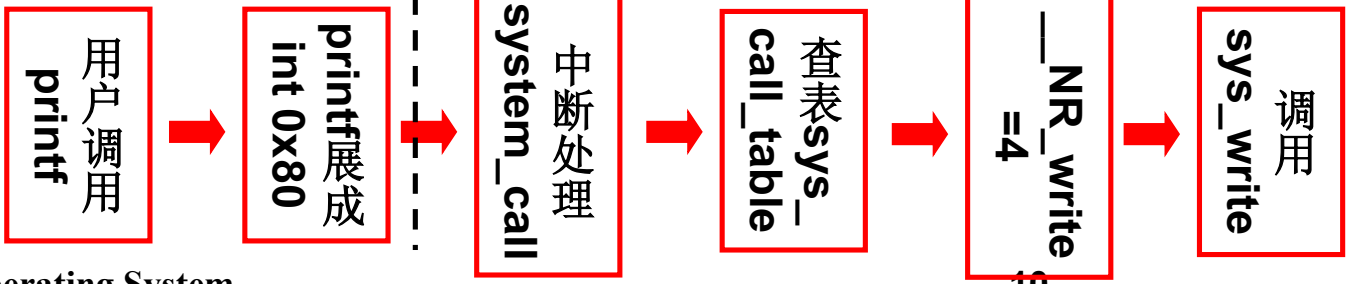
在include/linux/sched.h中

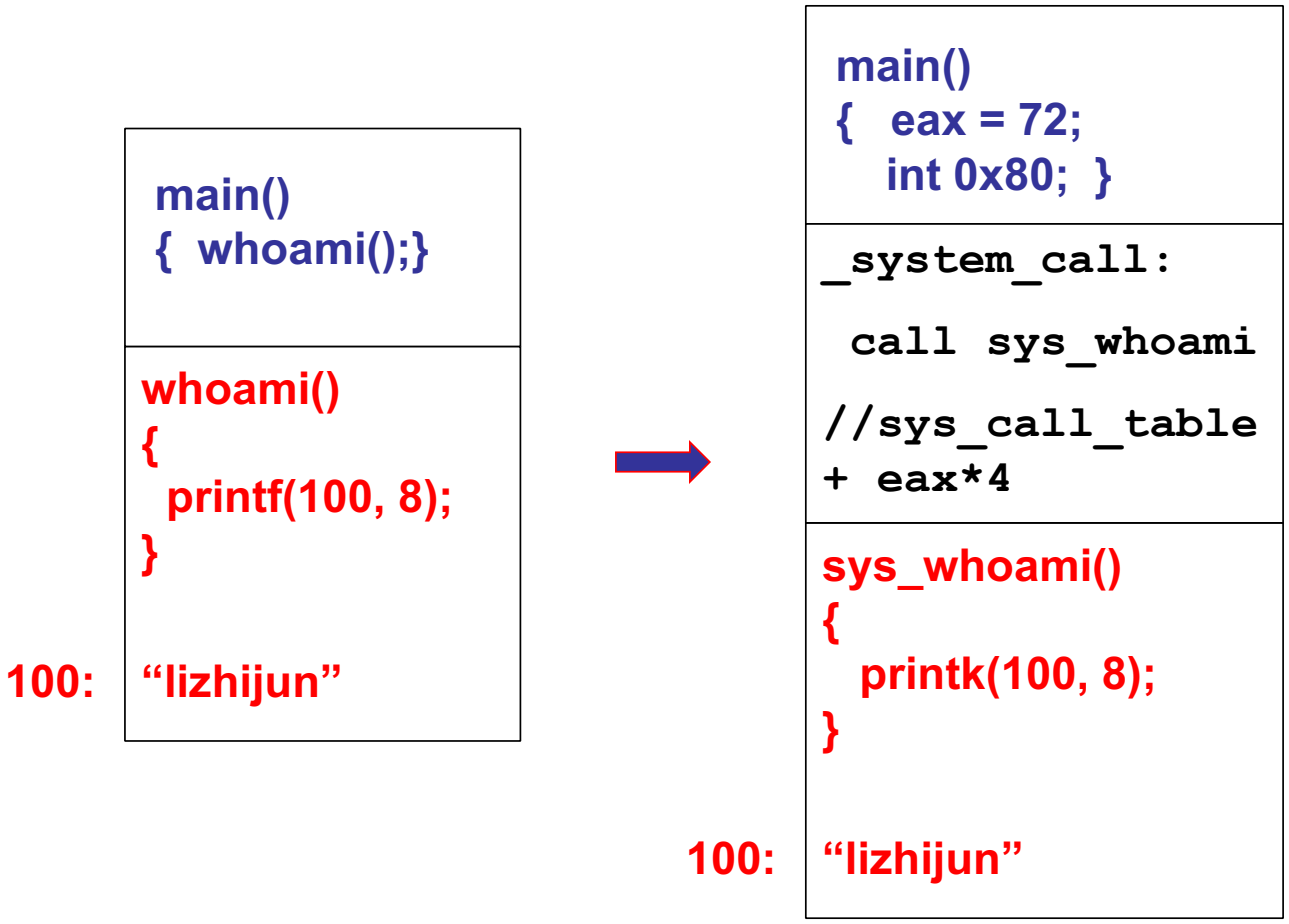
```
typedef int (fn_ptr*)();
```

- 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!




main()
{ whoami();}
whoami()
{
printf(100, 8);
}
100:
“lizhijun”
main()
{ eax = 72;
int 0x80; }
_system_call:
call sys_whoami
//sys_call_table
+ eax*4
sys_whoami()
{
printk(100, 8);
}
100:
“lizhijun”


Operating
System
Concepts