

# 哈工大操作系统-L5笔记

## 哈工大操作系统-L5笔记

1. 不能随意访问内核
  - 1.1 为何
  - 1.2 内核用户态/内核用户段
  - 1.3 中断(硬件提供的主动请求进入内核的方法)
    - 1.3.1 系统调用的核心
    - 1.3.2 以printf为例子
    - 1.3.3 write()如何实现
    - 1.3.4 INT0X80干了什么
2. 总结

本节课讲接口是如何实现的。即操作系统是怎么样来提供这些重要的函数的。

## 1. 不能随意访问内核

### 1.1 为何

如果能随意访问，那所有程序都可以随意访问存在内存中其他程序的数据，不安全。因此不能随意 jump。

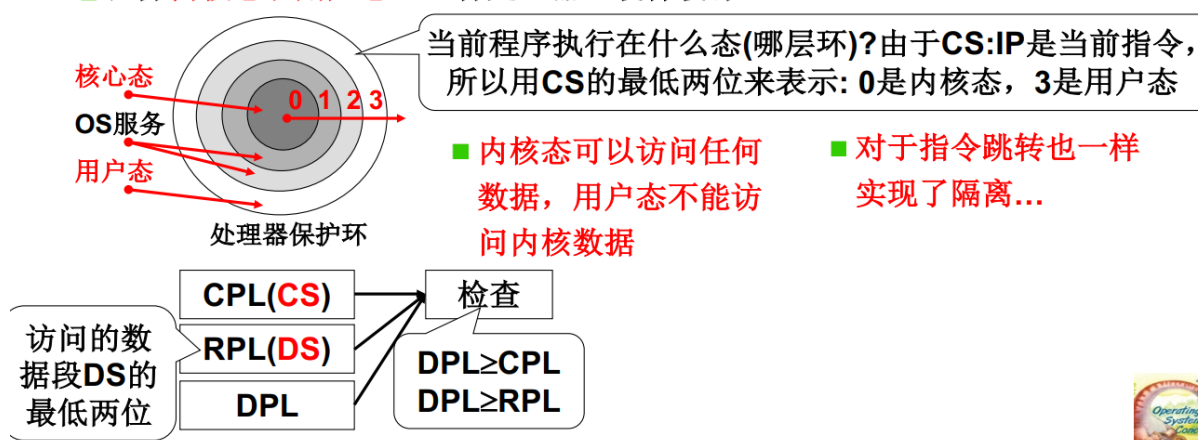
### 1.2 内核用户态/内核用户段

硬件设计将内核程序 and 用户程序隔离

## 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”



- 内存分为：内核段和用户段。
- 指令执行的状态分为：内核态和用户态
- 数据段中会记录该段是内核段还是用户段。(gdt表中的ds的dpl为0/3)

- **保护:** 想要访问数据时, 会根据指令找gdt去找数据, 然后系统会核查dpl和cpl的关系, 来保护内存段。
- 当前的指令也会记载目前是处于内核态还是用户态。(cs的cpl为0/3)
- 只有内核态可以访问任何数据。用户态访问不了内核段。

## 1.3 中断(硬件提供的主动请求进入内核的方法)

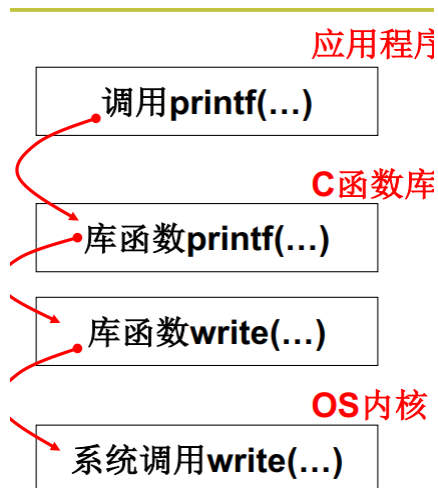
**中断: 用户主动调用内核代码的唯一方式.**

对于x86cpu即为INT指令, INT指令执行前,先将该INT代码的数据的DS的DPL改为3, 以允许用户代码通过idt找到该INT中断以进入内核;然后将进入内核的代码的CPL改为0, 以允许对内核的访问。一般为INT0x80。

### 1.3.1 系统调用的核心

- 用户程序中包含一段包含int指令的代码(这段代码一般是库函数)
- 操作系统写中断处理, 获取想调程序的编号
- 操作系统根据编号执行相应代码

### 1.3.2 以printf为例子



- 用户写程序时使用库函数printf()
- 而printf()中封装了另一个库函数write() (write()包含一段用宏展开的内嵌汇编代码)
- printf()和write(), 将用户输入的格式化输出参数, 转变为系统调用write()可以接受的参数。进而调用接口write()

### 1.3.3 write()如何实现

# 将关于write的故事完整的讲完...

在linux/include/unistd.h中

\_\_syscall3表示有3个参数

返回值

赋值

```
#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;}
```

这一句会展开成 NR\_write  
由根据宏定义, 这个  
NR\_write是4  
因此会将4放入eax寄存器中

■ 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

- write()中包含一段由宏定义展开的内嵌汇编代码, 使用INT0X80中断进入内核, 并传入中断号4。

## 1.3.4 INT0X80干了什么

调用相应的中断处理程序。

## 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

内核的代码段

\_\_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

Operating System

- 9 -

每个入口函数占4个字节

## \_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

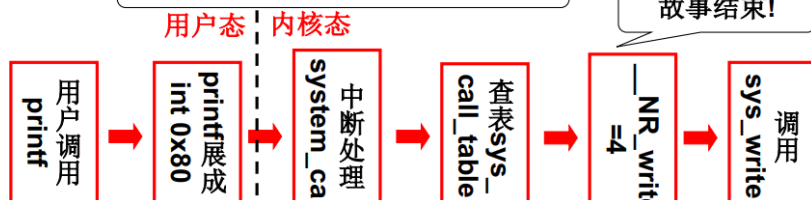
```
typedef int (fn_ptr*)();
```

这段代码可以找到sys\_write的中断处理函数

■ 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!



- 可以认为sys\_write是一个系统接口

## 2. 总结

- 为了保护数据, 将内核数据和用户数据分开。指令分成内核态和数据态, 内存分为内核段和数据段。
  - 只有内核态可以访问所有数据
- 因为隔离开, 所以用户只能通过中断来主动进入内核。如INT0x80中断。
- 用户主动进入内核的流程为:
  - 写一段包含库函数的代码
  - 库函数中封装了使用宏展开的汇编代码, 汇编代码中包含INT0x80中断
  - 执行库函数后, 系统进入中断, 然后去执行中断处理函数
    - INT0x80的执行过程中, 为了让用户执行这个中断, 会在中断的执行之前将0x80的idt表设置为用户段, 因此用户能访问中断的程序, 进而执行这个中断。

- 中断处理函数会去查表，执行相应的系统接口。比如INT0X80的4号中断，就是表中的4号系统接口system\_write