[**系统调用机制分析**](https://www.cnblogs.com/joey-hua/p/5570691.html)

【版权所有，转载请注明出处。出处：http://www.cnblogs.com/joey-hua/p/5570691.html 】

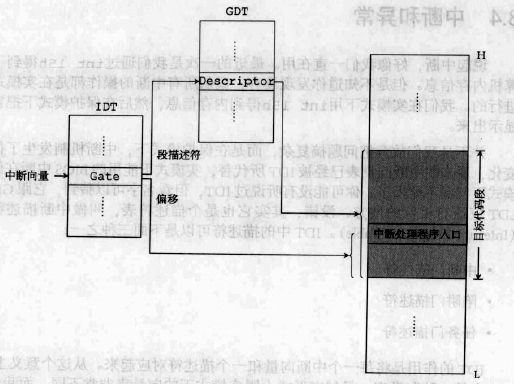
Linux内核从启动到初始化也看了好些个源码文件了，这次看到kernel文件夹下的system\_call.s，这个文件主要就是系统调用的过程。但说到系统调用，不只是这一个文件这么简单，里面牵扯到的内容太多，这里就做个笔记记录一下从建立中断到最终调用系统调用的完整机制。

假设就从write这个函数作为系统调用来解释。

系统调用的本质就是用户进程需要访问内核级别的代码，但用户进程的权限是最低的，内核代码是权限最高的，不允许直接访问，需要通过中断门作为媒介来实现权限的跳转。简单讲就是用户进程调用一个中断，这个中断再去访问内核代码。这里就来学习一下Linux内核具体是怎么做的。

**1.建立中断描述符表IDT**

因为要用到中断，所以首先要建立中断描述符表IDT，作用如下图：



在head.s文件中，建立好了IDT，比如要使用int 0x80，就从\_idt开始找到偏移为0x80的地方执行代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | .align 3                        # 按8 字节方式对齐内存地址边界。  \_idt:   .fill 256,8,0           # idt is uninitialized# 256 项，每项8 字节，填0。    idt\_descr:                  #下面两行是lidt 指令的6 字节操作数：长度，基址。      .word 256\*8-1           # idt contains 256 entries      .long \_idt        lidt idt\_descr                          # 加载中断描述符表寄存器值。 |

**2.建立0x80号中断**

所有的系统调用都是通过0x80号中断来实现的，所以接下来就是建立第0x80号中断，在sched.c中：

|  |  |
| --- | --- |
| 1  2 | // 设置系统调用中断门。    set\_system\_gate (0x80, &system\_call); |

这里通过set\_system\_gate这个宏定义就把0x80中断和函数system\_call关联上了，这里先不管system\_call，先看set\_system\_gate，在system.h中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //// 设置系统调用门函数。  // 参数：n - 中断号；addr - 中断程序偏移地址。  // &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是15，特权级是3。  #define set\_system\_gate(n,addr) \_set\_gate(&idt[n],15,3,addr)      //// 设置门描述符宏函数。  // 参数：gate\_addr -描述符地址；type -描述符中类型域值；dpl -描述符特权层值；addr -偏移地址。  // %0 - (由dpl,type 组合成的类型标志字)；%1 - (描述符低4 字节地址)；  // %2 - (描述符高4 字节地址)；%3 - edx(程序偏移地址addr)；%4 - eax(高字中含有段选择符)。  #define \_set\_gate(gate\_addr,type,dpl,addr) \  \_\_asm\_\_ ( "movw %%dx,%%ax\n\t" \    // 将偏移地址低字与选择符组合成描述符低4 字节(eax)。    "movw %0,%%dx\n\t" \      // 将类型标志字与偏移高字组合成描述符高4 字节(edx)。    "movl %%eax,%1\n\t" \     // 分别设置门描述符的低4 字节和高4 字节。  "movl %%edx,%2":  :"i" ((short) (0x8000 + (dpl << 13) + (type << 8))),    "o" (\*((char \*) (gate\_addr))),    "o" (\*(4 + (char \*) (gate\_addr))), "d" ((char \*) (addr)), "a" (0x00080000)) |

这里参考[**中断门结构图**](http://www.cnblogs.com/joey-hua/p/5388208.html)可知，这里设置特权级是3，用户进程也是3，就可以直接访问此中断，偏移地址对应的上面的system\_call，也就是说如果调用中断int 0x80，那么就会去访问system\_call函数。注意这里的n就是0x80，也就是idt数组的[0x80]，idt在head.h中声明，编译后会变成符号\_idt，在head.s中定义的，就此关联上。

**3.声明系统调用函数**

以write系统函数为例，在write.c中声明此函数：

|  |  |
| --- | --- |
| 1 | \_syscall3 (int, write, int, fd, const char \*, buf, off\_t, count) |

\_syscall3又是一个宏定义，在unistd.h中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // 有3 个参数的系统调用宏函数。type name(atype a, btype b, ctype c)  // %0 - eax(\_\_res)，%1 - eax(\_\_NR\_name)，%2 - ebx(a)，%3 - ecx(b)，%4 - edx(c)。  #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; \  } |

所以翻译过来就是在write.c中可以写成：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int write(int fd,const char\* buf,off\_t count) \  { \  long \_\_res; \  \_\_asm\_\_ volatile ( "int $0x80" \  : "=a" (\_\_res) \  : "" (\_\_NR\_write), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count))); \  if (\_\_res>=0) \  return (type) \_\_res; \  errno=-\_\_res; \  return -1; \  } |

是不是一下子就清晰明朗了，也就是说，如果一个用户进程要使用write函数，就会去调用int 0x80中断，然后把三个参数fd、buf、count分别存入ebx、ecx、edx寄存器，还有个最关键的是\_NR\_write，会把这个值存入eax寄存器，具体做什么用等会再说，这个是在unistd.h中定义的：

|  |  |
| --- | --- |
| 1 | #define \_\_NR\_write 4 |

好，现在各种初始化和声明都完成了，万事俱备只欠东风！

**4.系统调用过程**

用户进程调用函数write，就会调用int 0x80中断，上面第2点已经说了，如果调用中断int 0x80会去访问system\_call函数，sched.c：

|  |  |
| --- | --- |
| 1 | extern int system\_call (void);  // 系统调用中断处理程序(kernel/system\_call.s,80)。 |

是在system\_call中定义，注意编译后头部会加上\_，以下代码只截取了前半部分：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | \_system\_call:      cmpl $nr\_system\_calls-1,%eax    # 调用号如果超出范围的话就在eax 中置-1 并退出。      ja bad\_sys\_call      push %ds                                            # 保存原段寄存器值。      push %es      push %fs      pushl %edx                      # ebx,ecx,edx 中放着系统调用相应的C 语言函数的调用参数。      pushl %ecx                      # push %ebx,%ecx,%edx as parameters      pushl %ebx                      # to the system call      movl $0x10,%edx             # set up ds,es to kernel space      mov %dx,%ds                 # ds,es 指向内核数据段(全局描述符表中数据段描述符)。      mov %dx,%es      movl $0x17,%edx             # fs points to local data space      mov %dx,%fs                     # fs 指向局部数据段(局部描述符表中数据段描述符)。  # 下面这句操作数的含义是：调用地址 = \_sys\_call\_table + %eax \* 4。参见列表后的说明。  # 对应的C 程序中的sys\_call\_table 在include/linux/sys.h 中，其中定义了一个包括72 个  # 系统调用C 处理函数的地址数组表。  call \_sys\_call\_table(,%eax,4)  pushl %eax                                      # 把系统调用号入栈。（这个解释错误，是函数返回值入栈）  movl \_current,%eax                      # 取当前任务（进程）数据结构地址??eax。 |

注意从pushl %edx开始的三句代码，是前面第3点提到的三个参数依次从右向左入栈。重点是call \_sys\_call\_table(,%eax,4)这句代码，翻译过来就是call [eax\*4 + \_sys\_call\_table]，根据第3点，eax存的是\_NR\_write的值也就是4，因为\_sys\_call\_table是sys.h中的一个int (\*)()类型的数组，里面存的是所有的系统调用函数地址，所以再翻译一下就是访问sys\_call\_table[4]也就是sys\_write函数：

|  |  |
| --- | --- |
| 1  2  3 | // 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。  fn\_ptr sys\_call\_table[] = { sys\_setup, sys\_exit, sys\_fork, sys\_read,    sys\_write, ...} |

sys\_write在fs下的read\_write.c：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int  sys\_write (unsigned int fd, char \*buf, int count)  {    struct file \*file;    struct m\_inode \*inode;  ...  } |

好了，到这里为止才明白千回百转最终调用的就是这个sys\_write函数。至此分析结束！