**操作系统分析——main.c(2) move\_to\_user\_mode()函数**

今天来看看move\_to\_user\_mode()函数

这个宏函数是在main（）函数开启中断以后调用的。这个函数的作用是什么呢？经过sti()以及之上的语句，进程0就已经创建完毕了。其中最主要的就是创建了类型为task\_union 的init\_task。

static union task\_union init\_task = {INIT\_TASK,};

1

其中INIT\_TASK是linus硬编码进去的，它的值为

#define INIT\_TASK \

/\* state etc \*/ { 0,15,15, \

/\* signals \*/ 0,{{},},0, \

/\* ec,brk... \*/ 0,0,0,0,0,0, \

/\* pid etc.. \*/ 0,-1,0,0,0, \

/\* uid etc \*/ 0,0,0,0,0,0, \

/\* alarm \*/ 0,0,0,0,0,0, \

/\* math \*/ 0, \

/\* fs info \*/ -1,0022,NULL,NULL,NULL,0, \

/\* filp \*/ {NULL,}, \

{ \

{0,0}, \

/\* ldt \*/ {0x9f,0xc0fa00}, \

{0x9f,0xc0f200}, \

}, \

/\*tss\*/ {0,PAGE\_SIZE+(long)&init\_task,0x10,0,0,0,0,(long)&pg\_dir,\

0,0,0,0,0,0,0,0, \

0,0,0x17,0x17,0x17,0x17,0x17,0x17, \

\_LDT(0),0x80000000, \

{} \

}, \

}

其中，我们重点关注它的三个ldt的值：

{ \

{0,0}, \

/\* ldt \*/ {0x9f,0xc0fa00}, \

{0x9f,0xc0f200}, \

}, \

ldt[0]是一个空描述符。

ldt[1]是进程0的代码段描述符，根据之前的分析（LDT0分析 进程0的代码段的基地址其实就是0！也就是说进程0和内核共用同一个代码段！但是它的特权级为3。

而ldt[2]是进程0的数据段描述符。

其TSS的内容为：

/\*tss\*/ {0,PAGE\_SIZE+(long)&init\_task,0x10,0,0,0,0,(long)&pg\_dir,\

0,0,0,0,0,0,0,0, \

0,0,0x17,0x17,0x17,0x17,0x17,0x17, \

\_LDT(0),0x80000000, \

{} \

},

可以看到第二项为PAGE\_SIZE+(long)&init\_task，而这个值就是那个联合体的最后一个字节，这个值tss\_struct就是0特权的esp。

task\_union是一个联合体。其定义为：

union task\_union {

struct task\_struct task;//小于4096

char stack[PAGE\_SIZE];//4096B

};

每个task\_union都是包含4096字节，这刚好是一个页的大小。而这个task\_union的前104个字节是一个task\_struct结构。也就是说每个task\_union的布局其实是这样的：

|---------------------------------------4096个字节----------------------------------------------------|

|-------task\_struct----------|----------------------剩余未用-----------------------------------------|

为什么会有这么多的剩余未用空间呢？如果这部分是闲置不用的。对于追求最高效利用内存资源的OS来说，不太可能如此浪费资源。其实这个剩余未用部分，就作为一个个进程的0特权栈来使用。我们都晓得，进程在运行期间很有可能使用调用system call函数的，而一调用system call函数，CPL就会从原来的3特权，翻转到0特权。而翻转特权以后，原来3特权的栈将不得被0特权的内核使用，于是Intel要求每个进程在创建之初都得指定一个0特权的栈，用于将来进程进入0特权时使用。

并让task[0]指向这个task\_struct。

sched.c:

struct task\_struct \* task[NR\_TASKS] = {&(init\_task.task), };

1

创建一个task\_struct\* 类型的数组task,这个task里面的每个元素都是task\_struct\*类型的。在声明的同时，把&(init\_task.task)地址（其实就是刚刚那个init\_task的首地址）给task[0]，而task[1…NR\_TASKS]的值就都为NULL。

进程0的一些东西都准备完毕了，现在就需要把进程0的特权级由0特权转移到3特权。这是因为操作系统规定进程都必须运行在3特权上。

而这个move\_to\_user\_mode（）就是这个作用！

move\_to\_user\_mode()也是一个宏，它的定义为

#define move\_to\_user\_mode() \

\_\_asm\_\_ (

"movl %%esp,%%eax\n\t" \

"pushl $0x17\n\t" \

"pushl %%eax\n\t" \

"pushfl\n\t" \

"pushl $0x0f\n\t" \

"pushl $1f\n\t" \

"iret\n" \

"1:\tmovl $0x17,%%eax\n\t" \

"movw %%ax,%%ds\n\t" \

"movw %%ax,%%es\n\t" \

"movw %%ax,%%fs\n\t" \

"movw %%ax,%%gs" \

:::"ax")

我们一行行的分析。前面一部的压栈其实是在模拟INT n过程中需要执行的依次将ss,esp,eflag,cs,eip压栈。

movl %%esp,%%eax将esp内容给eax，esp是当前栈的栈顶，而当前用的是哪个栈呢？就是那个大小为一个页的user\_stack的那个栈。

pushl $0x17，将0x17压入栈

pushl %%eax，将eax压入栈，也就是刚刚的那个旧的栈顶。

"pushfl\n\t" \,将eflag压栈

"pushl $0x0f\n\t" \将0x0f压栈

"pushl $1f\n\t" \ 将指令1f的偏移压栈（它是iret的下一条指令）

看到这些push其实还不知道它要干什么，可以当看到iret时，我们就知道。

iret会隐含的执行以下指令：

popl eip

popl cs

popl eflag

popl esp

popl ss

而这些pop就是与上面的push一一对应。因此，我们再回过头看看这些出栈以后寄存器保存的是啥。

首先eip 就是标号为1的指令的偏移，这是iret返回的下一条指令的偏移。

然后cs为0x0f,这是一个段选择子。0x0f=0000 1111,表示以RPL=3去选择LDT里面的第2项。而我们知道，在main（）前面的sched\_init函数里面的lldt(0)指令，可以知道ldtr保存的是进程0的ldt基地址。所以LDT的第2项就是进程0的代码段。

再然后是eflag为eflag，这个没啥好说的。

然后再是esp为刚刚的保存的旧的栈顶，OS设计可谓是精打细算，一个栈的字节它都不浪费！咦此时不是会把旧的esp弹给这个esp嘛，那这样的话，cs的值就会是错误的！

这样子就把特权级从3转到0了，其他的东西依然没变化。

之后就是一波ds,es,fs,gs的对齐，都对齐到0x17选择子上。这个选择子0x17=0001 0111,即以RPL=3选择LDT的第3项(二进制10为2）。

---------------------

作者：Icoding\_F2014

来源：CSDN

原文：https://blog.csdn.net/jmh1996/article/details/83068544

版权声明：本文为博主原创文章，转载请附上博文链接！