## 

## 操作系统

## 实验报告

# 班号：\_\_\_1603101\_\_

# 学号：\_1161000113\_

# 姓名：\_\_李翼宇\_\_\_\_

1. 请简述head.s 的工作原理。

答：head.s是在32位保护模式下运行的。包括初始化设置与任务执行切换的部分。在初始化设置中，head.s设置了GDT表、设置系统定时芯片、设置IDT表（0x08时钟中断和0x80系统调用中断）以及切换到任务0执行。在任务切换中，head.s不进在任务0和任务1之间做切换，还完成了时钟中断和系统中断。

两个任务，任务0和任务1在LDT中的代码段和数据段的描述符的内容都是基地址为0x0000；段限长值为0x03ff，实际段长度为4MB。因此在线性地址空间中，这个内核的代码段和数据段与任务的代码段和数据段都从线性地址0开始并且由于没有采用分页机制，所以他们都直接对应物理地址0开始处。

为了每隔10ms切换运行的任务，head.s程序中把定时器芯片8253的通道0设置成每隔10ms就向中断控制芯片8259A发送一个时钟中断请求信号。PC机的ROM BIOS开机时已经在8259A中把时钟中断请求信号设置成中断向量8，因此我们需要在中断8的处理过程中执行任务切换操作。任务切换的实现是查看current变量中的当前运行的任务号，如果为0，就利用任务1的TSS选择符作为操作数执行远跳转指令，从而切换到任务1中；否则，就利用任务0的TSS选择符作为操作数执行远跳转指令，从而切换到任务0中。

每个任务在执行时，会首先把一个字符的ASCII码放入寄存器AL中，然后调用系统中断调用int 0x80，而该系统调用处理过程则会调用一个简单的字符写屏子程序，把寄存器AL中的字符显示在屏幕上，同时把字符显示的屏幕的下一个位置记录下来，作为下一次显示字符用。在显示过一个字符后，任务代码会使用循环语句延迟一段时间，然后又跳转到任务代码开始处继续循环执行，直到运行了10ms而发生了定时中断，从而代码会切换到另一个任务执行。对于任务0，寄存器AL中始终存放字符‘A’，而任务1运行时AL钟始终存放字符‘B’。因此程序运行时看到一连串的‘A’和一连串的‘B’间隔的连续不断的显示在屏幕上。若出现了一个‘C’，是由于PC偶然产生了一个不是时钟中断和系统调用中断的其他中断。因为我们已经在程序中给所有其他中断设置了一个默认中断处理程序。当出现一个其他中断时，系统就会运行这个中断处理程序，于是就会在屏幕上显示一个‘C’，然后退出中断。

1. 请记录head.s 的内存分布状况，写明每个数据段，代码段，栈段的起始与终止的内存地址。

答：在0x25处设置断点，跳转到head.s的重新设置IDT和GDT表结束处，看到通用寄存器内容如下：

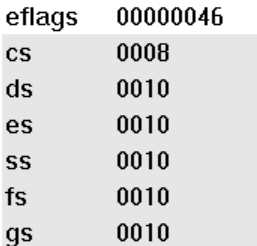


Figure1 head.s在0x25执行完时的通用寄存器数值

看到cs寄存器内容为0008，查看段选择符的结构（如下图所示）可知，cs寄存器的结构为0000 0000 0000 1000，其中，TI为0表示cs描述的是全局描述符表，RPL为00，表示有最高权限，索引index为1，表示应该到全局描述符表GDT中找第一项。

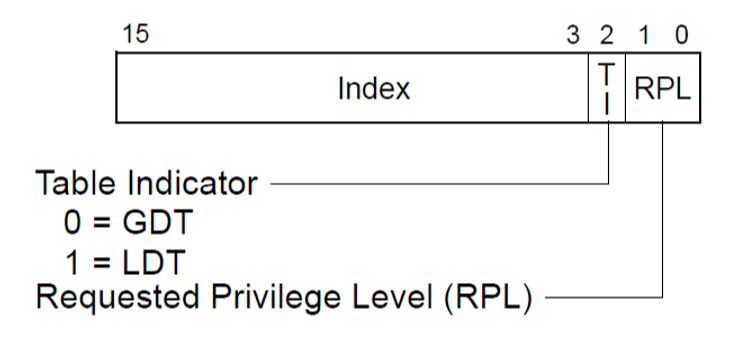


Figure2 段选择符的结构

根据段选择符的结构，可知CS寄存器的内容结构如下：

Figure3 CS寄存器的结构

又由GDTR寄存器的值为0x00000998知道GDT表所在的位置，所以我们需要到GDT表中寻找第1项。



Figure4 GDTR寄存器的值

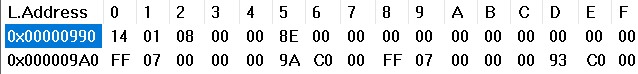


Figure5 内存中的GDT表

根据GDT的开始地址为0x998，且根据段描述符的结构知道，每个段描述符的长度为64位，即8个字节，从0x998开始，往后8个字节为第0项，为0x00 00 00 00 00 00 00 00。再往后的8个字节为第1项，即为我们要的CS指向的值，为0x00 C0 9B 00 00 00 07 FF。

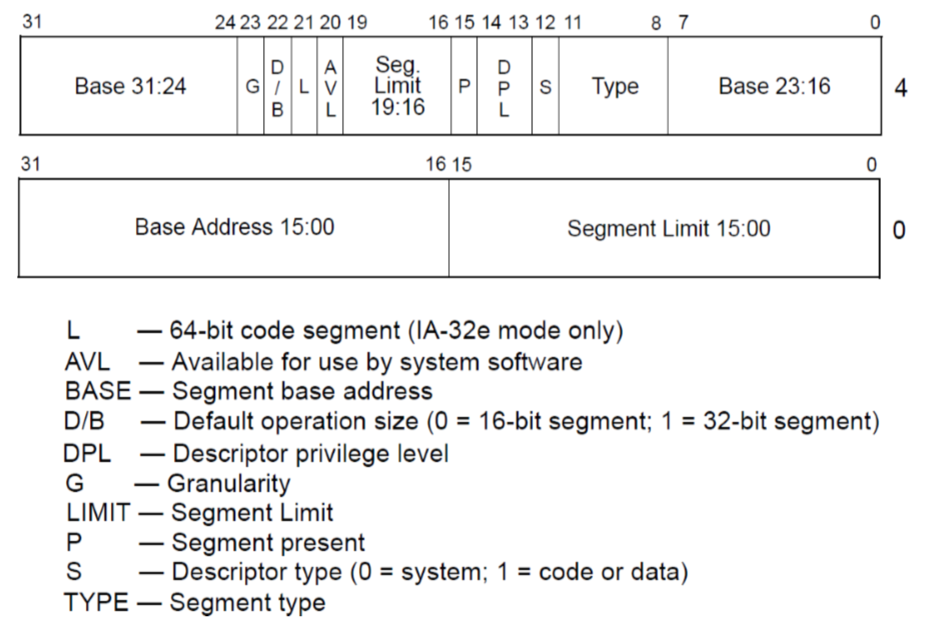


Figure6 段描述符的结构

根据段描述符的结构可知，64位的CS指向的段描述符的结构如下图所示：

Figure7 GDT表第1项的段描述符的结构

由上面的结构分析可知，基地址Base Address为0000 0000 0000 0000 0000 0000 0000 0000b，即为0x0；段限长Segment Limit为0000 0000 0111 1111 1111b，即为0x7ff；颗粒度G为1，所以Segment Limit的单位为4KB，实际段限长为(0x7ff + 1) \* 4KB = 8MB。所以我们可以得知全局代码段的起始地址为0x0，终止地址为0x7f ffff。

下面分析数据段，根据Figure1中所示，DS的值为0x0010，根据Figure2的结构可知，DS选择子的结构如下：

Figure8 DS的段选择符结构

接着查看GDT表的结构：

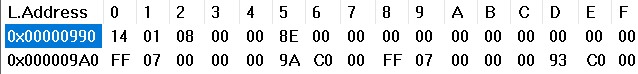


Figure9 内存中的GDT表

查看第2项，可知DS段描述符的内容为0x00 C0 93 00 00 00 07 FF。因此，段描述符的结构如下：

Figure10 GDT表第2项的段描述符的结构

所以，DS段描述符的基地址Base Address为0000 0000 0000 0000 0000 0000 0000 0000b，即为0x0；段限长Segment Limit为0000 0000 0111 1111 1111b，即为0x7ff，颗粒度G为1，单位为4KB，所以实际段限长为(0x7ff + 1) \* 4KB = 8MB。所以DS段指向的数据段的起始地址为0x0，终止地址为0x7f ffff。

在0xac处设置断点，step后进入任务0的代码段。下面分析任务0的代码段和数据段：

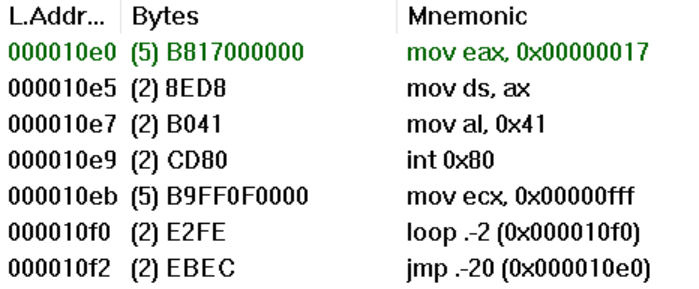


Figure11 任务0的代码段

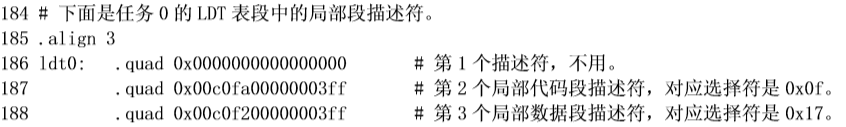


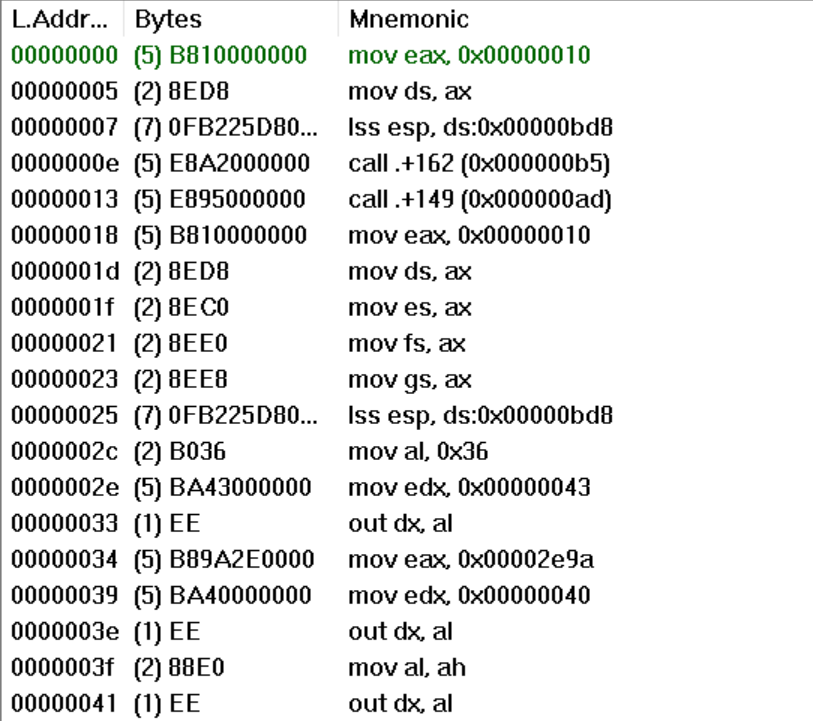
Figure12 任务0的LDT表中的段描述符

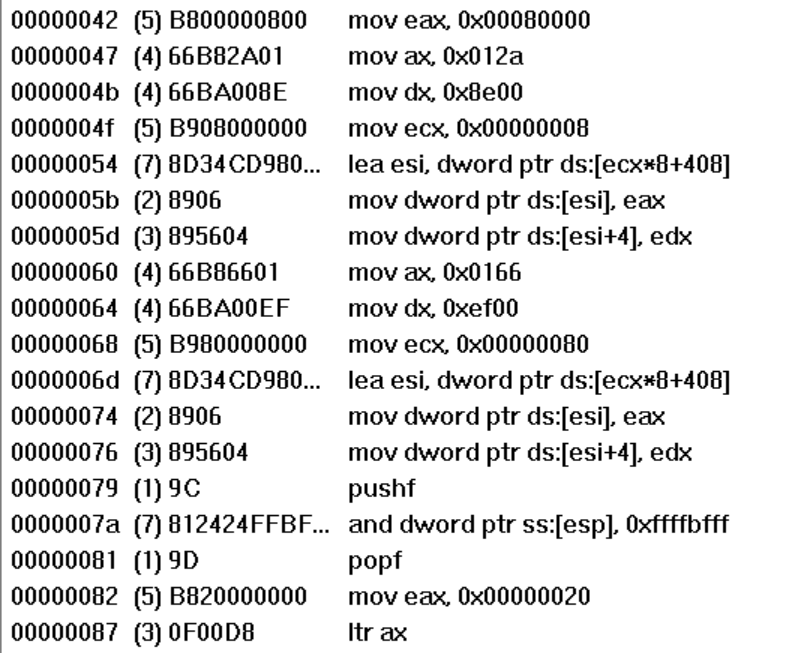
任务0的代码段的段描述符为0x00 c0 fa 00 00 00 03 ff，根据Figure6段描述符的结构，可知任务0的代码段的结构如下：

Figure13 任务0的代码段描述符的结构

可知任务0的代码段的基地址Base Address为0000 0000 0000 0000 0000 0000 0000 0000b，即为0x0；段限长Segment Limit为0000 0000 0011 1111 1111b，即为0x3ff，颗粒度G为1，单位为4KB，所以实际段限长为(0x3ff + 1) \* 4KB = 4MB。所以任务0的代码段的起始地址为0x0，终止地址为0x3f ffff。任务0的数据段的段描述符为0x00 c0 f2 00 00 00 03 ff，与任务0的代码段只有type字段不同，并不影响基地址、段限长和颗粒度的计算，所以任务0的数据段的起始地址也为0x0，终止地址也为0x3f ffff。

下面具体分析各个代码段、数据段和栈段的起始地址和终止地址：





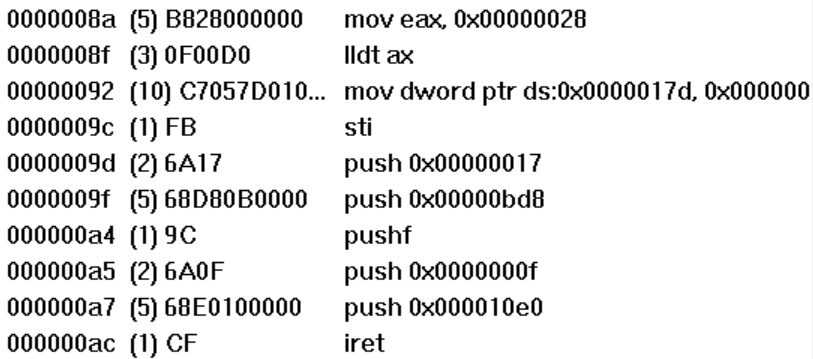


Figure14 startup\_32的代码段

由上图可知，startup\_32的代码段的起始地址为0x0，终止地址为0xac。同理我们可知setup\_gdt的代码段的起始地址为0xad，终止地址为0xb4；setup\_idt的代码段的起始地址为0xb5，终止地址为0xe4；write\_char的代码段的起始地址为0xe5，终止地址为0x113；ignore\_int的代码段的起始地址为0x114，终止地址为0x129；timer\_interrupt的代码段的起始地址为0x12b，终止地址为0x165；system\_interrupt的代码段的起始地址为0x166，终止地址为0x17c；task0的代码段的起始地址为0x10e0，终止地址为0x10f3；task1的代码段的起始地址为0x10f4，终止地址为0x1107；

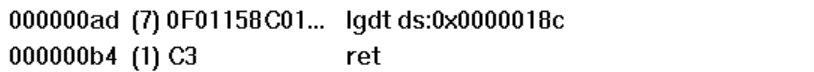


Figure15 setup\_gdt的代码段

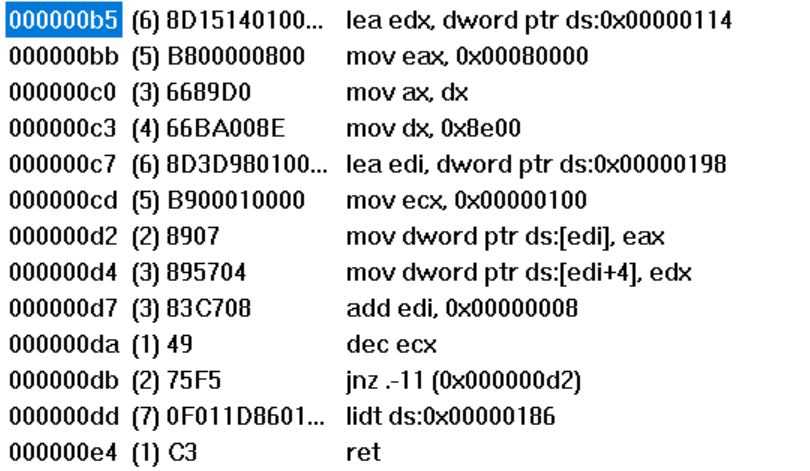


Figure16 setup\_idt的代码段

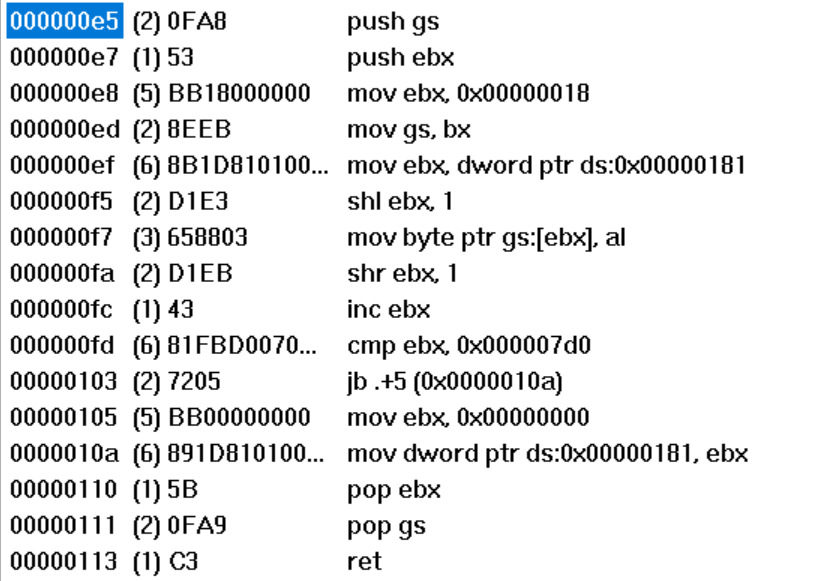


Figure17 write\_char的代码段

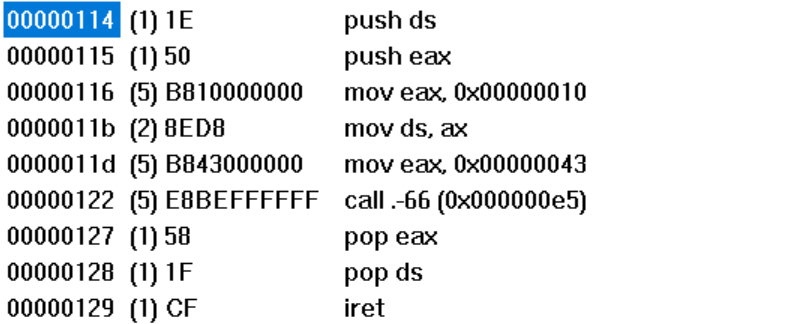


Figure18 ignore\_int的代码段

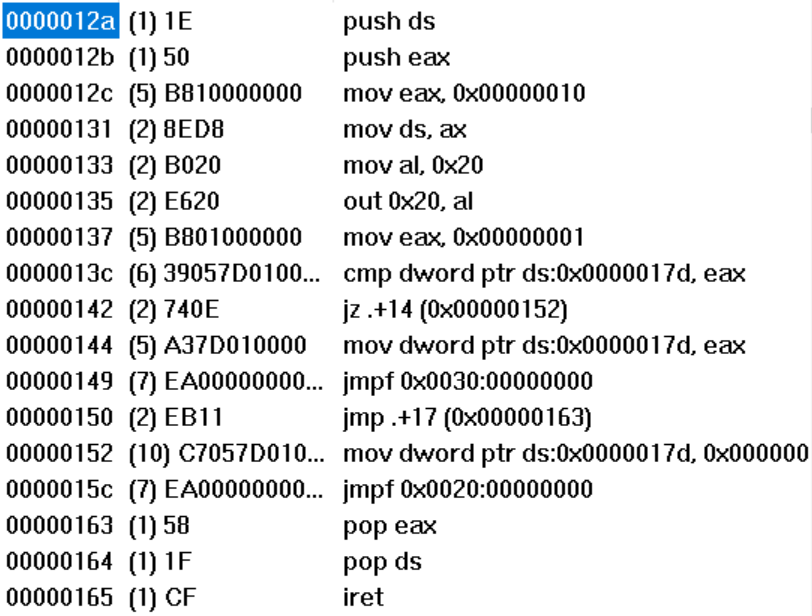


Figure19 timer\_interrupt的代码段

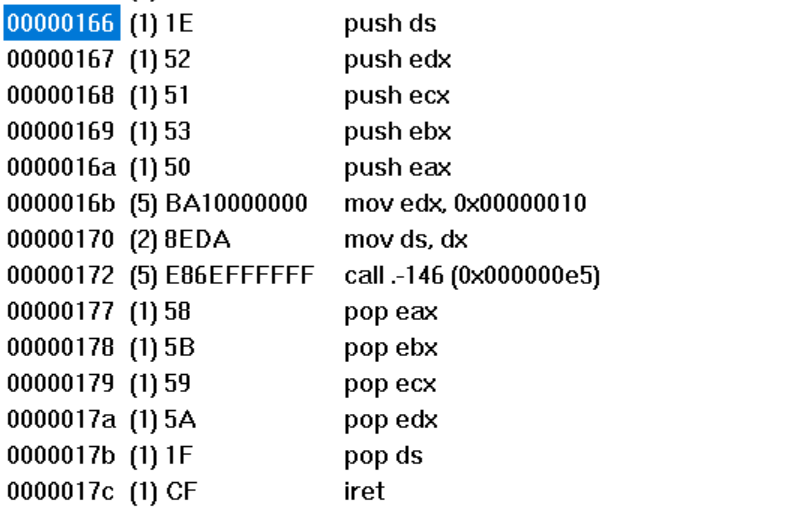


Figure20 system\_interrupt的代码段

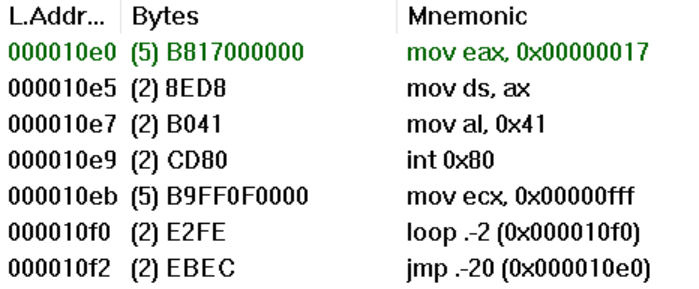


Figure21 task0的代码段

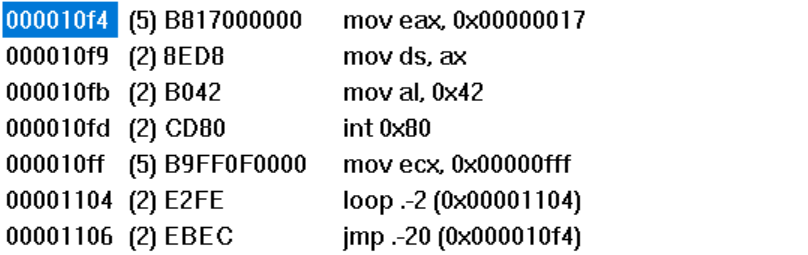


Figure22 task1的代码段

根据current在代码中的位置，在system\_interrupt的末尾iret的位置设置断点，看后面的代码可知，current的数据段为0x17d到0x180，scr\_loc的数据段为0x181到0x184。

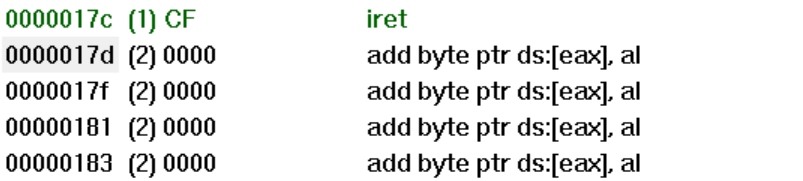


Figure23 current和scr\_loc的数据段

lidt\_opcode从0x186开始，因为占一个word一个long，共6个字节，所以从0x186开始，到0x18b结束。而lgdt\_opcode因为是.align 2，4字节对齐，所以从0x18c开始，同样占6个字节，到0x191结束。

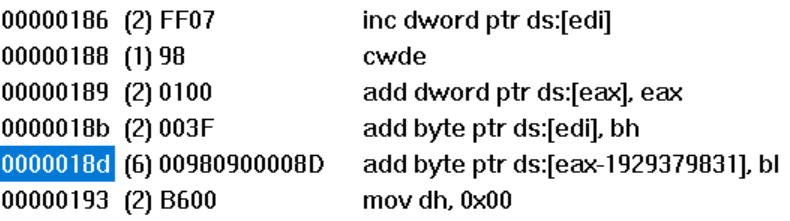


Figure24 lidt\_opcode和lgdt\_opcode的数据段

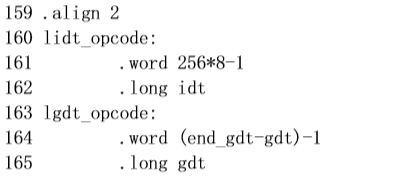


Figure25 lidt\_opcode和lgdt\_opcode的代码

因为idt和gdt的数据段8字节对齐，所以idt段的起始地址为0x198。256个门描述符，每个8字节，总共占4KB的地址空间。所以终止地址为0x997。

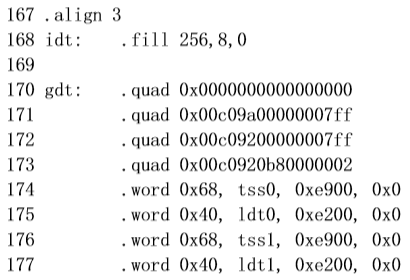


Figure26 idt和gdt的代码

gdt从idt结束的地方开始0x998，刚好也是8字节对齐的。有4个8字节长的段描述符，即32字节长；还有4个4word长的段描述符，也是32字节长，所以gdt总共64字节长，终止地址为0x9d7。同样的方法，分析ldt0，tss0，ldt1，tss1，可得起始地址和终止地址如表所示：

**代码段：**

|  |  |  |  |
| --- | --- | --- | --- |
| 代码段编号 | 名称 | 起始地址 | 终止地址 |
| 1 | startup\_32 | 0x00 | 0xac |
| 2 | setup\_idt | 0xb5 | 0xe4 |
| 3 | setup\_gdt | 0xad | 0xb4 |
| 4 | write\_char | 0xe5 | 0x113 |
| 5 | ignore\_int | 0x114 | 0x129 |
| 6 | timer\_interrupt | 0x12b | 0x165 |
| 7 | system\_interrupt | 0x166 | 0x17c |
| 8 | task0 | 0x10e0 | 0x10f3 |
| 9 | task1 | 0x10f4 | 0x1107 |

Table1 代码段的起始地址和终止地址

**数据段：**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据段编号 | 名称 | 起始地址 | 终止地址 |
| 1 | current | 0x17d | 0x180 |
| 2 | scr\_loc | 0x181 | 0x184 |
| 3 | lidt\_opcode | 0x186 | 0x18b |
| 4 | lgdt\_opcode | 0x18c | 0x191 |
| 5 | idt | 0x198 | 0x997 |
| 6 | gtd | 0x998 | 0x9d7 |
| 7 | ldt0 | 0xbe0 | 0xbf7 |
| 8 | tss0 | 0xbf8 | 0xc5f |
| 9 | ldt1 | 0xc60 | 0xe77 |
| 10 | tss1 | 0xe78 | 0xedf |

Table2 数据段的起始地址和终止地址

**栈段：**

|  |  |  |  |
| --- | --- | --- | --- |
| 栈段编号 | 名称 | 起始地址 | 终止地址 |
| 1 | Init\_stack | 0x9d8 | 0xbd8 |
| 2 | Krn\_stk0 | 0xc60 | 0xe60 |
| 3 | Krn\_stk1 | 0xe00 | 0x10e0 |
| 4 | User\_stk1 | 0x1108 | 0x1308 |

Table3 栈段的起始地址和终止地址

1. 简述Head.s 57~62行在做什么？简述iret执行后，pc如何找到下一条指令？并记录iret执行前后，栈是如何变化的。

答：57 pushl $0x17 #把任务0当前局部空间数据段（堆栈段）选择符入栈。

58 pushl $init\_stack #把堆栈指针入栈（也可以直接把ESP入栈）。

59 pushfl #把标志寄存器值入栈。

60 pushl $0x0f #把当前局部空间代码段选择符入栈。

61 pushl $task0 #把代码指针入栈

62 iret #执行中断返回指令，从而切换到特权级3的任务0中执行。

**如何找到下一条指令：**

执行iret之后，将推入堆栈的段地址和偏移地址弹出，使程序返回到原来发生中断的地方。其作用是从中断中回复到中断前的状态，具体作用是：

1. 恢复IP：(IP)🡨((SP)+1: (SP)), (SP)🡨(SP)+2
2. 恢复CS：(CS) 🡨((SP)+1: (SP)), (SP)🡨(SP)+2
3. 恢复中断前的PSW（program status word），即恢复中断前的标志寄存器的状态：(FR) 🡨((SP)+1: (SP)), (SP)🡨(SP)+2
4. 恢复ESP（返回权限发生变化）
5. 恢复SS（返回权限发生变化）

PC根据弹出的CS查找其IDT表形成段基地址0x0，然后再加上偏移地址EIP得到下一条指令的地址0x10e0。

根据程序中的代码段，我们把断点设在iret处（即0xac），然后执行step，即可看到CS和EIP的变化：

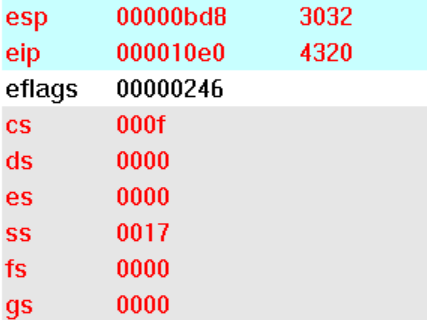


Figure27 CS和EIP的值

看到此时CS的值为0xf，EIP的值为0x10e0。根据Figure2可知，CS寄存器的内容为：

Figure28 CS寄存器的内容

看到此时RPL变为3，已经变到了用户特权级。此时我们需要查看LDT表的第一项：

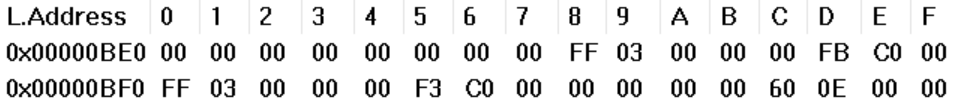


Figure29 LDT表的内容

看到第一项的值为0x00C0 FB00 0000 03FF，即为：

Figure30 LDT表第一项的结构

可知基地址为0x0，再加上偏移地址EIP为0x10e0。所以下一条指令的地址为0x10e0。

**iret执行前后栈的变化：**

执行前，栈的内容就是57-62行指令压入的内容。栈的指针ESP==0xbc4。



Figure31 执行前ESP的值

查看栈内容，看到结果如下：

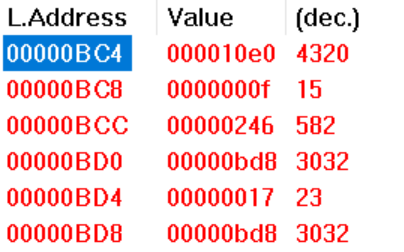


Figure32 执行前栈的内容

执行后，将原先压入的内容全部弹出，使程序返回到原先发生中断的地方。看到栈指针此时为ESP==0xbd8。



Figure33 执行后ESP的值

查看栈的内容，看到结果如下：

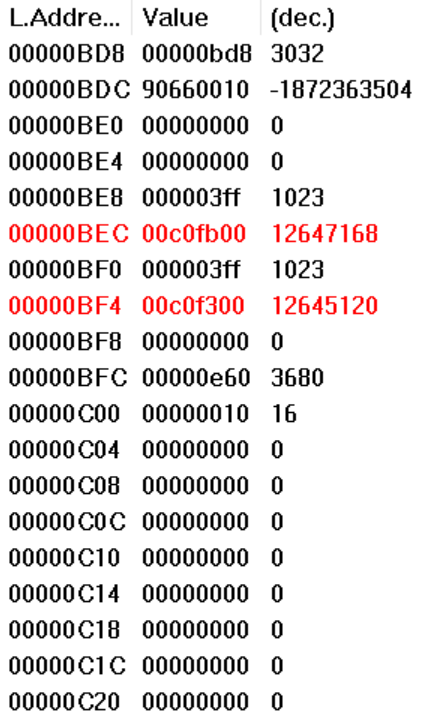


Figure34 执行后栈的内容

1. 当任务进行系统调用时，即int 0x80时，记录栈的变化情况。当执行完system\_interrupt函数，执行153行iret时，记录栈的变化情况。此外，当进入和退出system\_interrupt时，都发生了模式切换，请总结模式切换时，特权级是如何改变的？栈切换吗？如何进行切换的？

答：系统调用0x80之前，栈为task0或者task1的用户栈，栈的内容为：

task0的用户栈为：

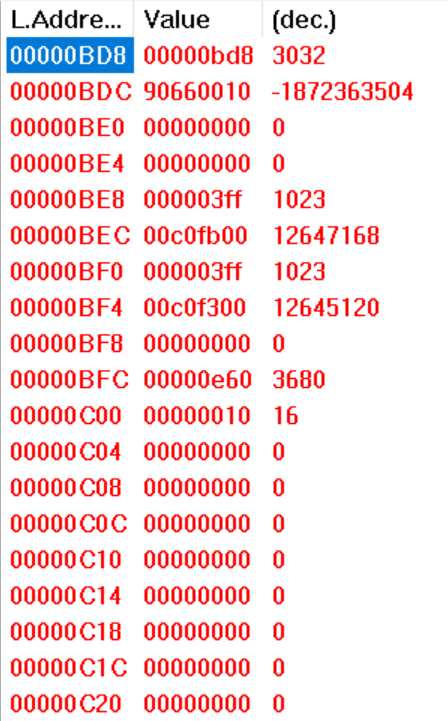


Figure35 task0的用户栈

task1的用户栈为：

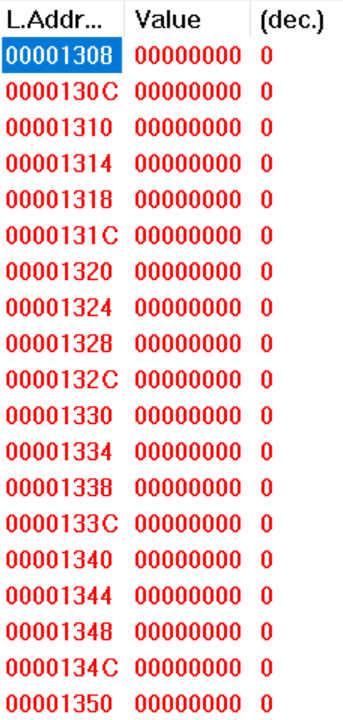


Figure36 task1的用户栈

执行了系统调用int 0x80之后，堆栈变为此任务的系统栈，而在这系统栈中自动压入了发生中断之前的SS、栈指针、标志寄存器、CS和下一条指令的EIP。task0和task1在调用int 0x80后的系统栈如下：

task0的系统栈：

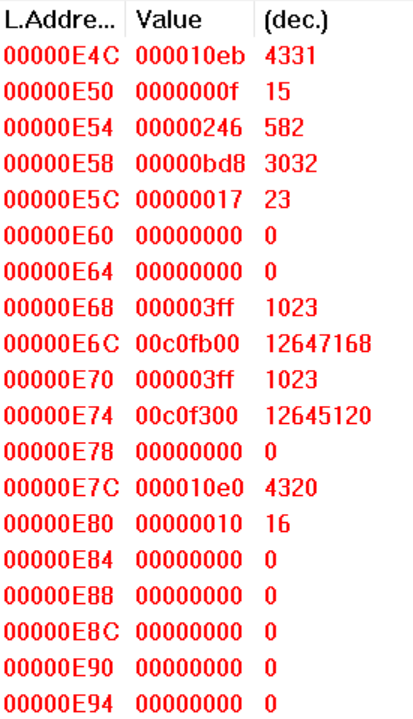


Figure37 task0的系统栈

task1的系统栈：

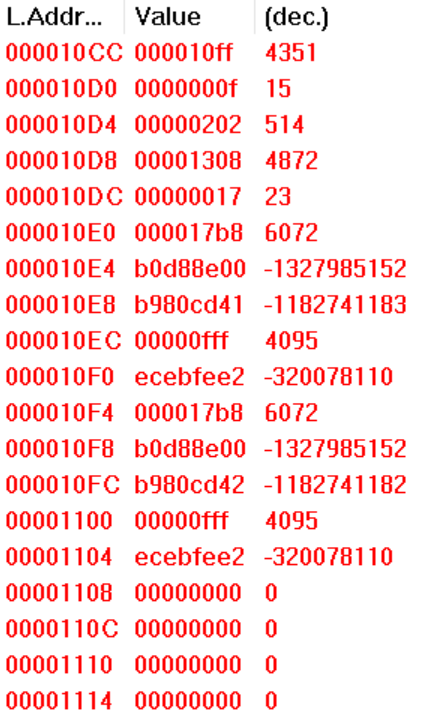


Figure38 task1的系统栈

当执行完system\_interrupt函数，执行153行iret前，栈为任务0的系统栈，ESP==0xe4c。



Figure39 执行完系统中断，未执行iret返回时的ESP

此时栈的内容为：

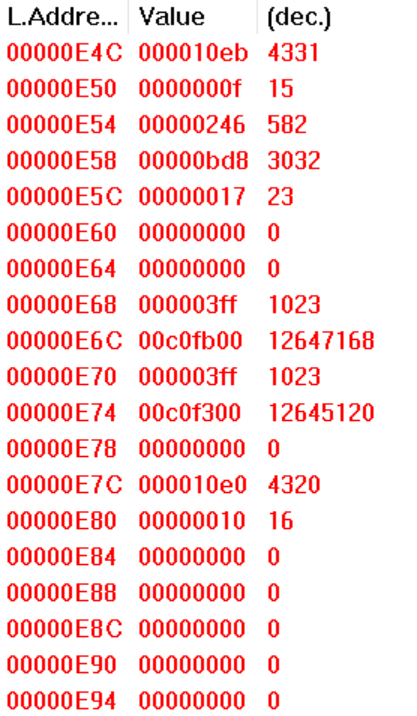


Figure40 执行完系统中断，未执行iret返回时的栈内容

执行完153行的iret后，系统栈弹出下一条指令的地址以及各个寄存器的值，弹完后系统栈为空。栈变化为任务1的用户栈：ESP==0x1308。

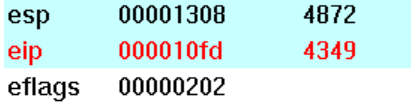


Figure41 执行完iret返回时的ESP

栈的内容为：

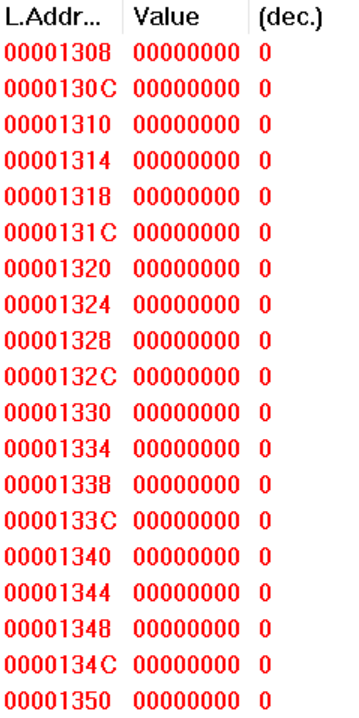


Figure42 执行完iret返回时的栈内容

**总结模式切换：**

当进入和退出系统中断system\_interrupt时，都发生了模式切换。

当进入系统中断时，硬件将各个寄存器的值压入系统栈后，根据IDTR为0x198。

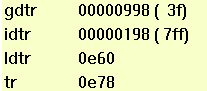


Figure43 进入系统中断时的IDTR

再加上8\*0x80，得到内存地址为0x218。寻址到0x598，得到0x00 00 EF 00 00 08 01 66：

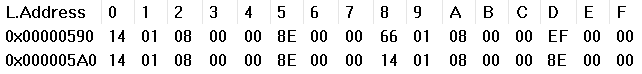


Figure44 进入系统中断时的IDT对应项

根据描述符的格式：

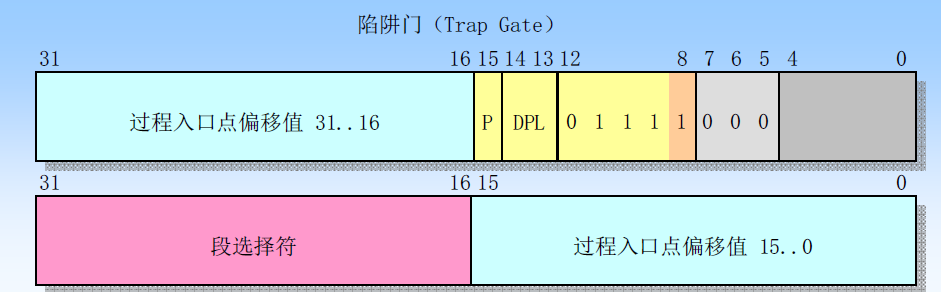


Figure45 描述符的格式

可知偏移值为0x166，段选择符为0x8，DPL为3。此时CS由原来的0xf变为0x8，因为程序要把控制转移到另一个具有不同特权级的代码段中时，处理器会改变CPL，因此当进入系统中断时，原来的特权级3就会变为特权级0。

因为系统中断处理过程在高特权级上执行，因此会发生堆栈切换操作。堆栈切换过程如下：

1. 处理器从当前执行任务的TSS段中得到中断或异常处理过程使用的堆栈的段选择符和栈指针，即tss0.ss0和tss0.esp0，分别为0x10和0xc60，然后处理器会把被中断程序，即任务0的站选择符和栈指针压入新栈中。
2. 接着处理器会把EFLAGS、CS和EIP寄存器的当前值也压入新栈中。
3. 如果异常会产生一个错误号，那么该错误号也会把最后压入新栈中。

经过以上操作，此时已经由用户模式切换至内核模式，特权级由3变为0，用户栈也切换至系统栈。

退出系统中断时，由于处于特权级0的代码不能直接把控制权转移给特权级为3的代码中执行，但中断返回操作iret是可以的，可以利用中断返回指令iret来启动第一个任务。具体的实现方法是在初试堆栈init\_stack中人工设置一个返回环境，把任务0的TSS段选择符加载到任务寄存器LTR中，LDT段选择符加载到LDTR中，然后把任务0的用户栈指针（0x17：init\_stack）和代码指针（0x0f：task0）以及标志寄存器值压入栈中，然后执行中断返回指令iret。该指令会弹出堆栈上的堆栈指针作为任务0的用户指针，恢复假设的任务0的标志寄存器内容，并且弹出栈中代码指针放入CS:EIP寄存器中，从而开始执行任务0的代码，完成了从特权级0到特权级3的代码的控制转移，模式由内核模式转换到用户模式。

1. 当时钟中断发生，进入到timer\_interrupt程序，请**详细记录**从任务0切换到任务1，以及又过了10ms，从任务1切换回到任务0，整个流程是怎样的？TSS是如何变化的？各个寄存器的值是如何变化的？请详细总结任务切换的过程。

答：从任务0切换到任务1，断点设在0x12c：

1、将任务0的各寄存器的值和状态保存到TSS0中。

2、在timer\_interrupt中，判断current的值，由于此时current为0，则通过ljmp $TSS1\_SEL,$0获取TSS1的段选择符。通过TSS1\_SEL和LDT1\_SEL选择符查找GDT的TSS1和LDT1段描述符，从而得到TSS1和LDT1的地址，将新任务的TSS状态加载进处理器。这包括LDTR寄存器、PDBR（CR3）寄存器、EFLAGS寄存器、EIP寄存器以及通用寄存器和段选择符。在此期间检测到的任何错误都将出现在新任务的上下文中。

3、根据TSS1的段描述符获取TSS1中保存的cs和eip的值，再根据cs=0xf从LDT1中获取基地址，（其中LDT1的地址可以从TSS1中保存的LDT1\_SEL选择符获取GDT中保存的LDT1的段描述符，因此可得到LDT1的基地址），基地址加上eip就得到了Task1的地址，由此跳转到任务1中。此时TSS0中保存着任务0在跳转前的对应各寄存器的值。

跳转前TSS0,TSS0起始地址是0xbf8，终止地址是0xc5f：

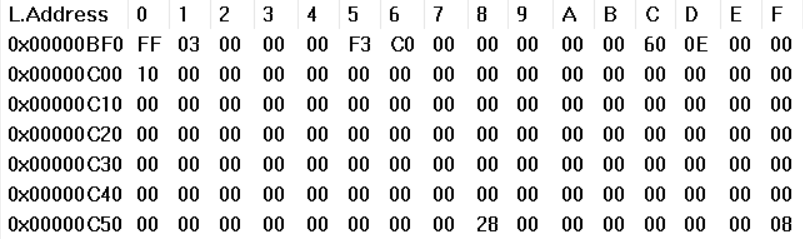


Figure46 跳转前的TSS0

跳转后数据或指针加载到对应各寄存器后的TSS0：

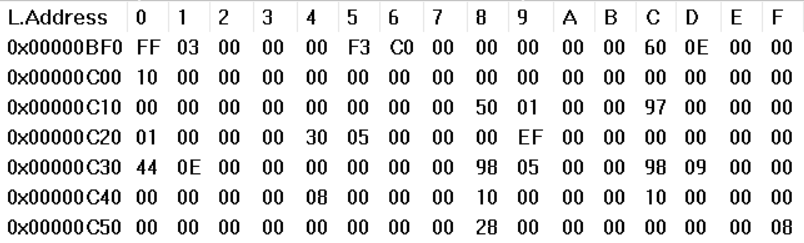


Figure47 跳转后的TSS0

跳转后各寄存器的值是TSS1中对应的各寄存器，TSS1起始地址是0xe78，终止地址是0xedf。

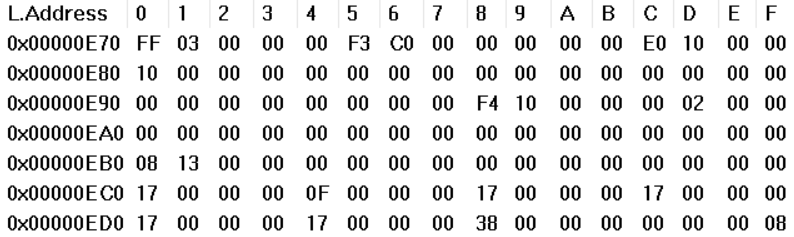


Figure48 跳转后的TSS1

此时各寄存器值为：

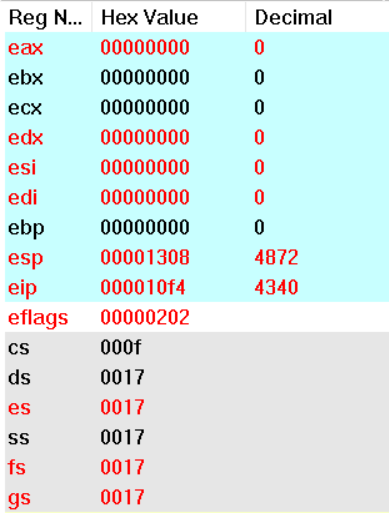


Figure49 跳转后的各寄存器值

当过10ms后，从任务1切换到任务0：执行跳转指令时

1、将任务1的各寄存器的值和状态保存到TSS1中。

2、在timer\_interrupt中，判断current的值，由于此时current为1，则通过ljmp $TSS0\_SEL,$0获取TSS0的段选择符。通过TSS0\_SEL和LDT0\_SEL选择符查找GDT的TSS0和LDT0段描述符，从而得到TSS0和LDT0的地址，将新任务的TSS状态加载进处理器，这包括LDTR寄存器、PDBR（CR3）寄存器、EFLAGS寄存器、EIP寄存器以及通用寄存器和段选择符。在此期间检测到的任何错误都将出现在新任务的上下文中。其中EIP的值为0x150是前面跳转到任务0时下一条指令的地址，即 timer\_interrupt中的jmp 2f，此时寄存器保存的是TSS0的状态。栈切换为任务0的系统栈，如下图。

|  |  |  |
| --- | --- | --- |
| 地址 | 寄存器值 | 寄存器名称 |
| 0xe44 | 0x41 | eax |
| 0xe48 | 0x17 | ds |
| 0xe4c | 0x10f0 | eip |
| 0xe50 | 0xf | cs |
| 0xe54 | 0x246 | eflags |
| 0xe58 | 0xbd8 | esp |
| 0xe5c | 0x17 | ss |

Table4 栈切换为任务0的系统栈

3、接着程序跳转到timer\_interrupt中的jmp 2f，接着跳转到popl %eax中执行，最后根据iret返回，将栈中的EAX,DS,EIP,CS,EFLAGS,ESP,SS弹出，此时EIP=0x10f0，CS=0xf，因此地址为0x10f0，即task0中的loop lb语句，跳转后栈切换为task0的用户栈。跳转之后，TSS1变为跳转到任务0前任务1对应寄存器的值。此时TSS0中保存着任务0在跳转前的对应各寄存器的值。

任务1切换到任务0之前的TSS1：

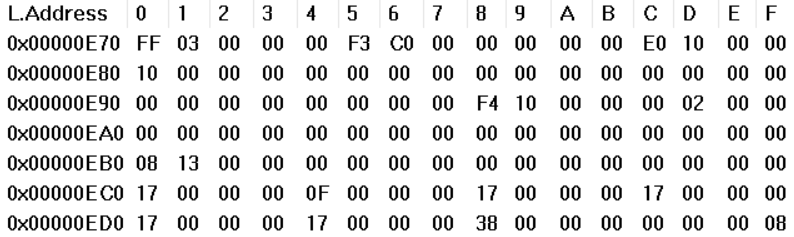


Figure50 任务1切换到任务0之前的TSS1

跳转之后的TSS1：

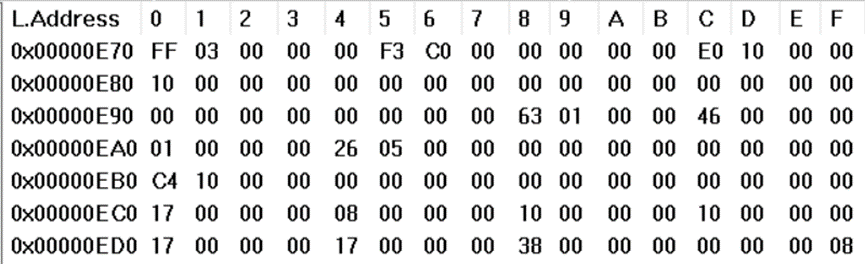


Figure51 跳转后的TSS1

此后若再从task0跳转到task1，则EIP的值不再是0x10f4，而是0x163，是前面跳转到任务1时下一条指令的地址，即 timer\_interrupt中的popl %eax，栈切换为任务1的系统栈，执行iret后，CS=0xf，EIP=0x1104，栈切换为任务1的用户栈。

各寄存器的值从Task0跳转到Task1的变化如下：

|  |  |  |
| --- | --- | --- |
| 寄存器名称 | 跳转前值 | 跳转后值 |
| eax | 0x41 | 0x42 |
| ebx | 0x0 | 0x0 |
| ecx | 0x0 | 0x0 |
| edx | 0xef00 | 0x0 |
| esi | 0x598 | 0x0 |
| edi | 0x998 | 0x0 |
| ebp | 0x0 | 0x0 |
| esp | 0xbd8 | 0x1308 |
| eip | 0x150 | 0x163 |
| eflags | 0x246 | 0x202 |
| cs | 0xf | 0xf |
| ds | 0x17 | 0x17 |
| es | 0x0 | 0x17 |
| ss | 0x17 | 0x17 |
| fs | 0x0 | 0x17 |
| gs | 0x0 | 0x17 |

Table5 各寄存器从task0跳转到task1的变化

从Task1跳转到Task0寄存器的变化是上述过程的逆过程。

总结任务切换的过程：

1、从作为JMP或CALL指令操作数中，或者从任务门中，或者从当前TSS的前一任务链接字段（对于由IRET引起的任务切换）中取得新任务的TSS段选择符。

2、检查当前任务是否允许切换到新任务。把数据访问特权级规则应用到JMP和CALL指令上。当前任务的CPL和新任务段选择符的RPL必须小于或等于TSS段描述符的DPL，或者引用的是一个任务门。无论目标任务门或TSS段描述符的DPL是何值，异常、中断（除了使用INT n指令产生的中断）和IRET指令都允许执行任务切换。对于INT n指令产生的中断将检查DPL。

3、检查新任务的TSS描述符是标注为存在的（P=1），并且TSS段长度有效（大于0x67）。当试图执行会产生错误的指令时，都会恢复对处理器状态的任何改变。这使得异常处理过程的返回地址指向出错指令，而非出错指令随后的一条指令。因此异常处理过程可以处理出错条件并且重新执行任务。异常处理过程的介入处理对应用程序来说是完全透明的。

4、如果任务切换产生自JMP或IRET指令，处理器就会把当前任务（老任务）TSS描述符中的忙标志B复位；如果任务切换是由CALL指令，异常或中断产生，则忙标志B不动。

5、如果任务切换由IRET产生，则处理器会把临时保存的EFLAGS映像中的NT标志复位；如果任务切换由CALL、JMP指令或者异常或中断产生，则不用改动上述NT标志。

6、把当前任务的状态保存到当前任务的TSS中，处理器会从任务寄存器中取得当前任务TSS的基地址，并且把以下寄存器内容复制到当前TSS中：所有通用寄存器、段寄存器中的段选择符、标志寄存器EFLAGS以及指令指针EIP。

7、如果任务切换是由CALL指令、异常或中断产生，则处理器就会把新任务中加载的EFLAGS中的NT标志置位。如果任务切换产生自JMP或IRET指令，就不改动新加载EFLAGS中的标志。

8、如果任务切换由CALL、JMP指令或者异常或中断产生，处理器就会设置新任务TSS描述符中的忙标志B。如果任务切换由IRET产生，则不去改动B标志。

9、使用新任务TSS的段选择符和描述符加载任务寄存器TR（包括隐藏部分）。设置CR0寄存器的TS标志。

10、把新任务的TSS状态加载进处理器。这包括LDTR寄存器、PDBR（CR3）寄存器、EFLAGS寄存器、EIP寄存器以及通用寄存器和段选择符。在此期间检测到的任何错误都将出现在新任务的上下文中。

11、开始执行新任务（对于异常处理过程，新任务的第一条指令显现出还没有执行）。

**请修改模板的文件名，并在第一页上填写你的个人信息，然后回答第二页上的5个问题，每个问题的回答都要新起一页，标清序号。**