调试分析 Linux 0.00 多任务切换

实验目的

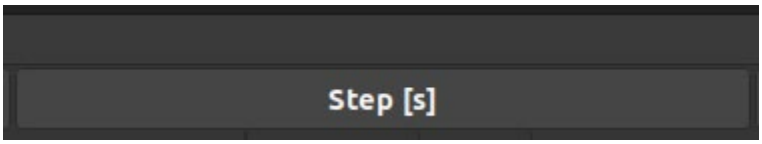
- 通过调试一个简单的多任务内核实例，使大家可以熟练的掌握调试系统内核的方法；
- 掌握 Bochs 虚拟机的调试技巧；
- 通过调试和记录，理解操作系统及应用程序在内存中是如何进行分配与管理的；

实验内容

通过调试一个简单的多任务内核实例，使大家可以熟练的掌握调试系统内核的方法。这个内核示例中包含两个特权级 3 的用户任务和一个系统调用中断过程。我们首先说明这个简单内核的基本结构和加载运行的基本原理，然后描述它是如何被加载进机器 RAM 内存中以及两个任务是如何进行切换运行的。

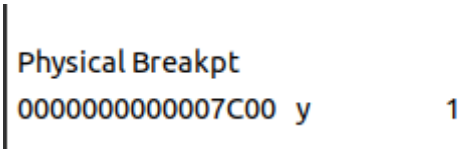
掌握 Bochs 虚拟机的调试技巧

- 如何单步跟踪？

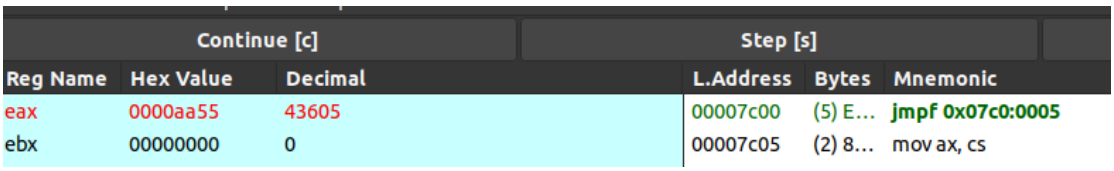


- 如何设置断点进行调试？

Eg：b 0x7c00



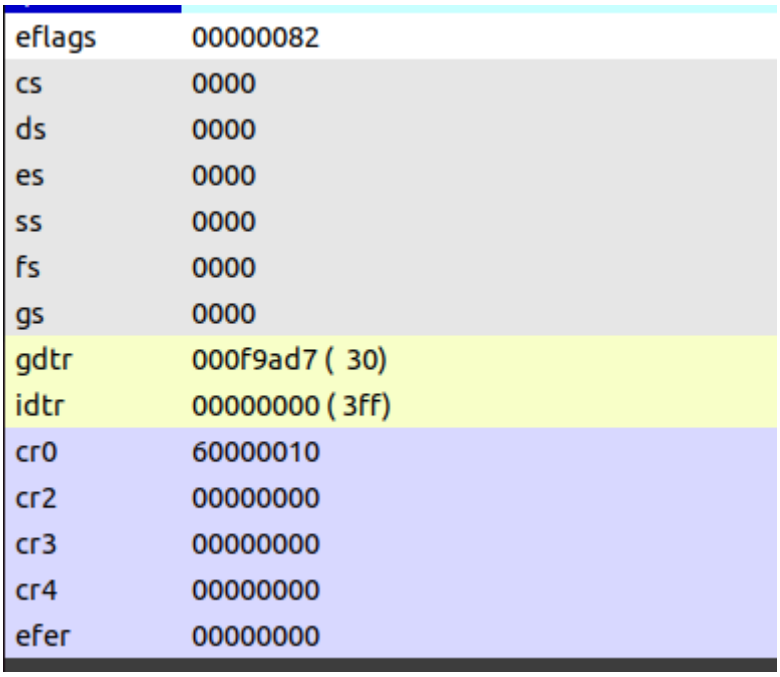
再点击 continue：



- 如何查看通用寄存器的值？

Reg Name	Hex Value	Decimal
eax	0000aa55	43605
ebx	00000000	0
ecx	00090000	589824
edx	00000000	0
esi	000e0000	917504
edi	0000ffac	65452
ebp	00000000	0
esp	0000ffd6	65494
ip	00007c00	31744

- 如何查看系统寄存器的值?



- 如何查看内存指定位置的值?

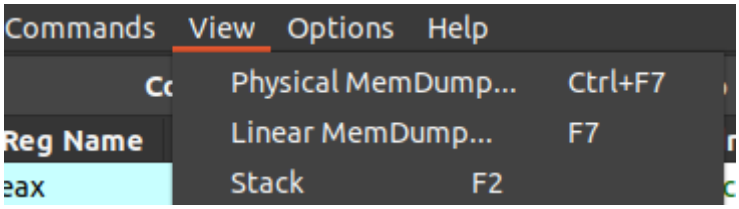
  输入`x/nuf <addr>`命令，其中`<addr>`是要查看的内存地址，按`Enter`键后将显示该地址处的内存值。
- 如何查看各种表，如 gdt，idt，ldt 等?

  `info gdt`：显示全局描述符表（GDT）的内容。

  `info idt`：显示中断描述符表（IDT）的内容。

  `info ldt`：显示本地描述符表（LDT）的内容。
- 如何查看 TSS?

  Info tss
- 如何查看栈中的内容?

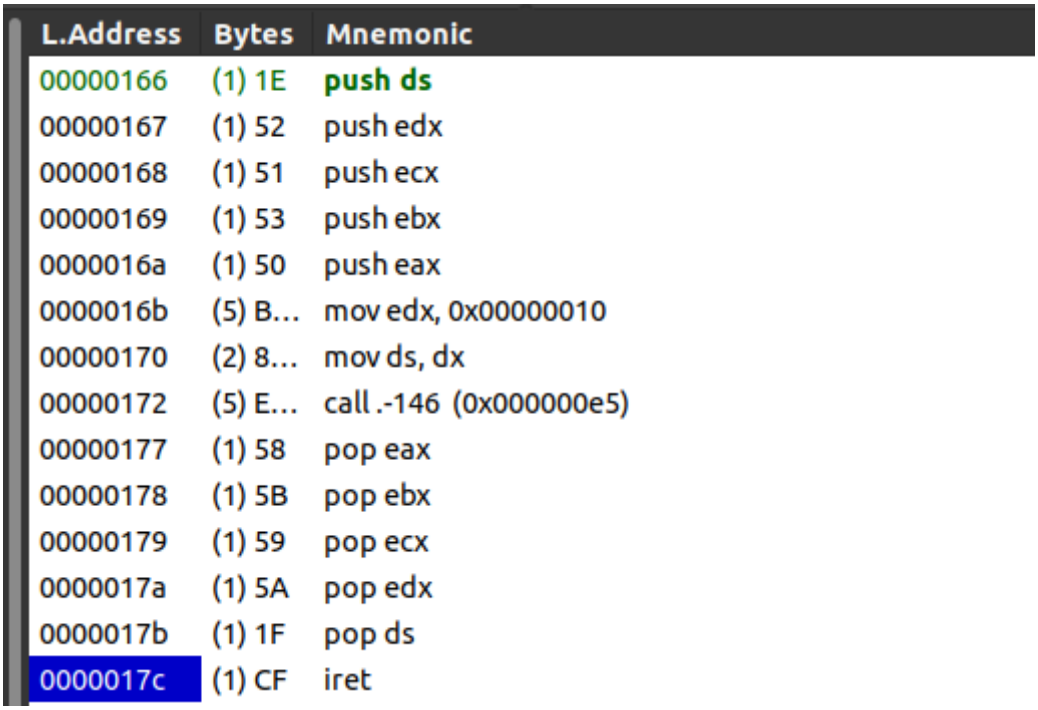


- 如何在内存指定地方进行反汇编?
  u [addr]

实验报告

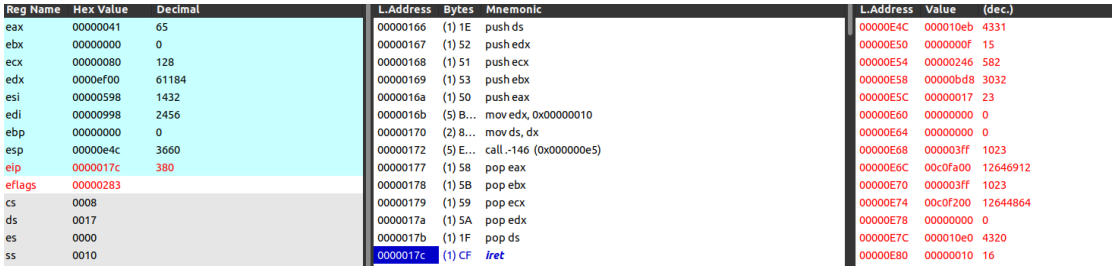
当执行完 system_interrupt 函数，执行 153 行 iret 时，记录栈的变化情况。

先查询实验报告一，发现 system_interrupt 函数的入口地址在 0x166，在这里打断点，然后执行到此:



L.Address	Bytes	Mnemonic
00000166	(1) 1E	push ds
00000167	(1) 52	push edx
00000168	(1) 51	push ecx
00000169	(1) 53	push ebx
0000016a	(1) 50	push eax
0000016b	(5) B...	mov edx, 0x00000010
00000170	(2) 8...	mov ds, dx
00000172	(5) E...	call .-146 (0x000000e5)
00000177	(1) 58	pop eax
00000178	(1) 5B	pop ebx
00000179	(1) 59	pop ecx
0000017a	(1) 5A	pop edx
0000017b	(1) 1F	pop ds
0000017c	(1) CF	iret

发现 iret 在 0x17c 处，再次在这里打断点，执行到此，并观察栈的值:

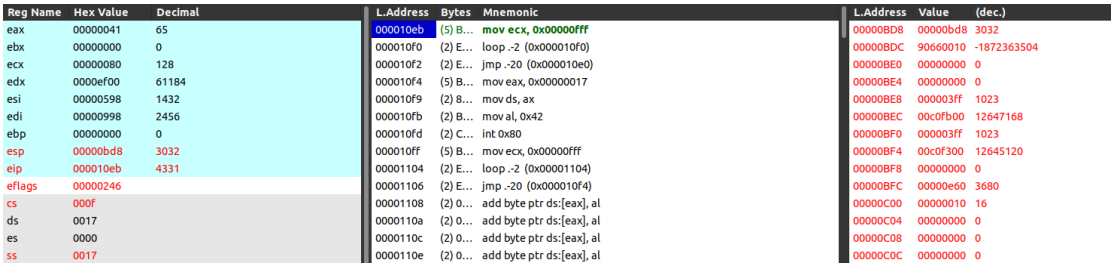


Reg Name	Hex Value	Decimal
eax	00000041	65
ebx	00000000	0
ecx	00000080	128
edx	0000ef00	61184
esi	00000598	1432
edi	00000998	2456
ebp	00000000	0
esp	00000e4c	3660
eip	0000017c	380
eflags	00000283	
cs	0008	
ds	0017	
es	0000	
ss	0010	

L.Address	Bytes	Mnemonic
00000166	(1) 1E	push ds
00000167	(1) 52	push edx
00000168	(1) 51	push ecx
00000169	(1) 53	push ebx
0000016a	(1) 50	push eax
0000016b	(5) B...	mov edx, 0x00000010
00000170	(2) 8...	mov ds, dx
00000172	(5) E...	call .-146 (0x000000e5)
00000177	(1) 58	pop eax
00000178	(1) 5B	pop ebx
00000179	(1) 59	pop ecx
0000017a	(1) 5A	pop edx
0000017b	(1) 1F	pop ds
0000017c	(1) CF	iret

L.Address	Value	(dec.)
00000E4C	000010eb	4331
00000E50	0000000f	15
00000E54	00000246	582
00000E58	00000bd8	3032
00000E5C	00000017	23
00000E60	00000000	0
00000E64	00000000	0
00000E68	000003ff	1023
00000E6C	00c0fa00	12646912
00000E70	000003ff	1023
00000E74	00c0f200	12644864
00000E78	00000000	0
00000E7C	000010e0	4320
00000E80	00000010	16

现在栈顶 SS:ESP 指向 0x10:0x0e4c，并且可以看见内核栈栈顶的中断返回地址 0x0f:0x10eb 和用户栈栈顶地址 0x17:0x0bd8。然后单步执行 iret，再观察栈状态的变化:



Reg Name	Hex Value	Decimal
eax	00000041	65
ebx	00000000	0
ecx	00000080	128
edx	0000ef00	61184
esi	00000598	1432
edi	00000998	2456
ebp	00000000	0
esp	00000bd8	3032
eip	000010eb	4331
eflags	00000246	
cs	000f	
ds	0017	
es	0000	
ss	0017	

L.Address	Bytes	Mnemonic
000010eb	(5) B...	mov ecx, 0x00000fff
000010f0	(2) E...	loop .-2 (0x000010f0)
000010f2	(2) E...	jmp .-20 (0x000010e0)
000010f4	(5) B...	mov eax, 0x00000017
000010f9	(2) 8...	mov ds, ax
000010fb	(2) B...	mov al, 0x42
000010fd	(2) C...	int 0x80
000010ff	(5) B...	mov ecx, 0x00000fff
00001104	(2) E...	loop .-2 (0x00001104)
00001106	(2) E...	jmp .-20 (0x000010f4)
00001108	(2) 0...	add byte ptr ds:[eax], al
0000110a	(2) 0...	add byte ptr ds:[eax], al
0000110c	(2) 0...	add byte ptr ds:[eax], al
0000110e	(2) 0...	add byte ptr ds:[eax], al

L.Address	Value	(dec.)
00000BD8	00000bd8	3032
00000BDC	90660010	-1872363504
00000BE0	00000000	0
00000BE4	00000000	0
00000BE8	000003ff	1023
00000BEC	00c0fb00	12647168
00000BF0	000003ff	1023
00000BF4	00c0f300	12645120
00000BF8	00000000	0
00000BFC	00000e60	3680
00000C00	00000010	16
00000C04	00000000	0
00000C08	00000000	0
00000C0C	00000000	0

栈顶 SS:ESP 被切换到了 0x17:0x0bd8，并且中断返回到了 0x0f:0x10eb 处。

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？

进入和退出 system interrupt 时，都发生了模式切换。

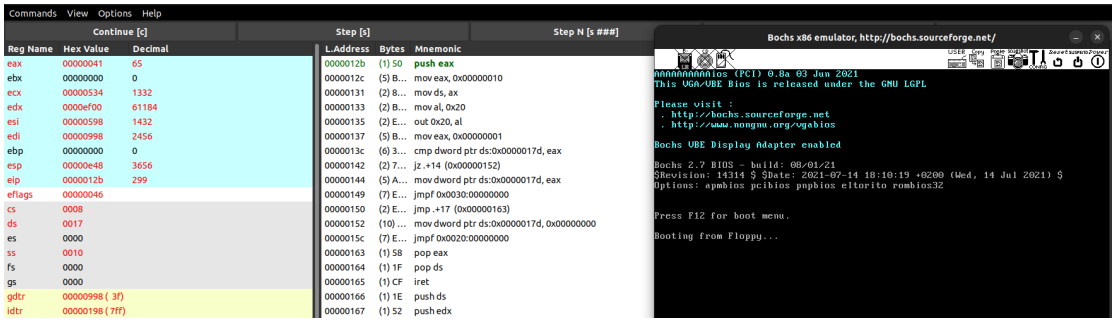
进入系统中断时，硬件将各寄存器的值压入系统栈后，查找其中断向量表后，根据描述符的格式，cs 值变为 0x08，此时，cs 标志着由用户模式切换至内核模式，特权级也由 1 变为 0，栈也切换至系统栈。

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到特权级了的代码中执行，可利用中断返回指令`iret`来启动第一个任务。在初始堆栈`init stack`中人工设置一个返回环境即可。

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 中以后，把任务 0 的用户栈指针和代码指针以及标志寄存器值压入栈中，然后执行中断返回指令`iret`。该指令会恢复任务 0 的标志寄存器内容，并且弹出栈中指针放入 CS:EIP 中，从而开始执行任务 0 的代码，完成了从特权级 0 到特权级了代码的控制转移，模式由内核模式转换为用户模式。

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 的过程。

查询实验报告一：得知 timer_interrupt 的入口地址在 0x12a 处，为了进入该程序，在 0x12b 处打断点，并执行到此处：



可以发现第一次执行的 task0 已经打印出了几个 A，还没有 B；

接下来单步执行到远跳指令 jmpf:

L.Address	Bytes	Mnemonic
0000012b	(1) 50	push eax
0000012c	(5) B...	mov eax, 0x00000010
00000131	(2) 8...	mov ds, ax
00000133	(2) B...	mov al, 0x20
00000135	(2) E...	out 0x20, al
00000137	(5) B...	mov eax, 0x00000001
0000013c	(6) 3...	cmp dword ptr ds:0x0000017d, eax
00000142	(2) 7...	jz .+14 (0x00000152)
00000144	(5) A...	mov dword ptr ds:0x0000017d, eax
00000149	(7) E...	**jmpf 0x0030:00000000**

这条远跳指令会将一个 TSS 选择子装入 CS，实际上就是为了将任务切换到这个 TSS。可以查看到这条指令的选择子是 0x30，这个时候可以再在底下框内输入调试命令“info tss”，来查看任务切换前的 TSS:

```
cr3: 0x00000000
eip: 0x00000000
eflags: 0x00000000
cs: 0x0000 ds: 0x0000 ss: 0x0000
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00000000
ldt: 0x0028
i/o map: 0x0800
```

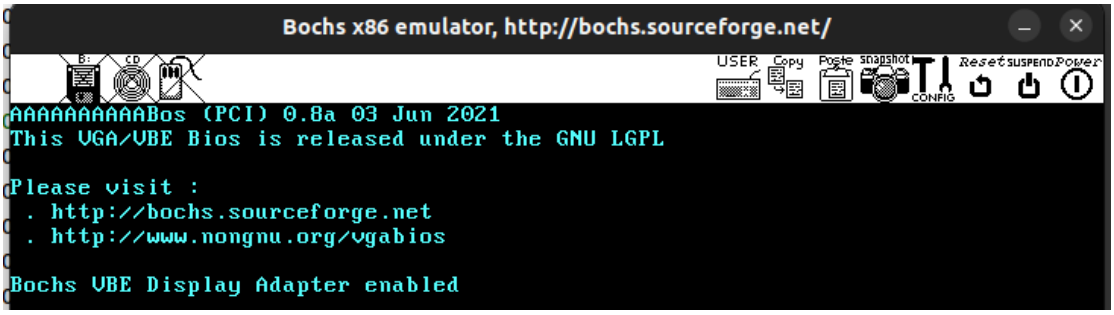
然后点击单步执行，再观察 tss 的值与寄存器的值:

Continue [c]

Reg Name	Hex Value	Decimal
eax	00000000	0
ebx	00000000	0
ecx	00000000	0
edx	00000000	0
esi	00000000	0
edi	00000000	0
ebp	00000000	0
esp	00001308	4872
eip	000010f4	4340
eflags	00000202	
cs	000f	
ds	0017	
es	0017	
ss	0017	
fs	0017	
gs	0017	
gdtr	00000998 (3f)	
idtr	00000198 (7ff)	
ldtr	0e60	
tr	0e78	
cr0	60000019	
cr2	00000000	
cr3	00000000	
cr4	00000000	

```
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00001308
ldt: 0x0038
i/o map: 0x0800
```

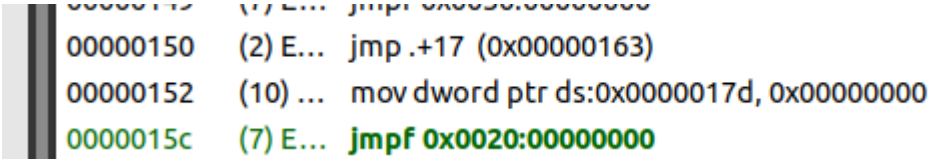
都对应相等，说明任务切换时会根据 tss 的各个字段改变对应寄存器的值；
再让程序执行几步：



也是打印出了 B，说明 task1 也开始执行了，任务切换成功。

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？

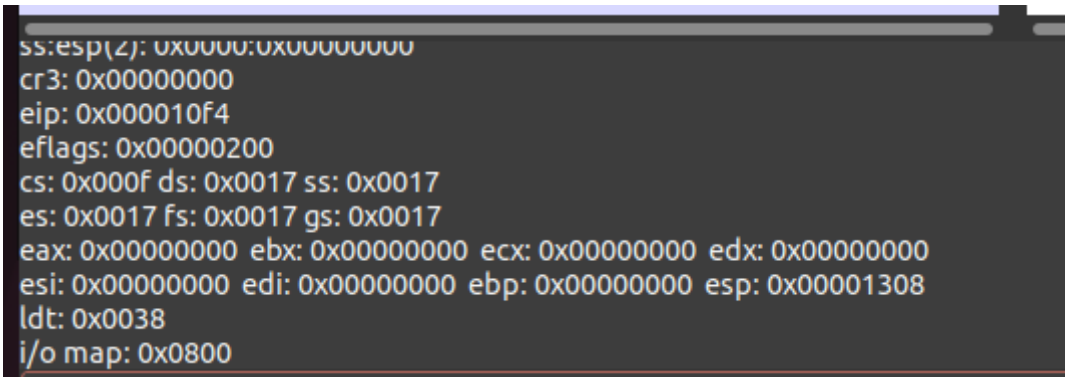
因为已经在程序开始打了断点，因此直接点击：c，让程序执行到断点处；再次单击执行到远跳指令 jmpf，此次要跳转的地址为 0x20：



观察 GDT 表：得知 0x20 就是另一个 tss 选择子；

Index	Base Address	Size	DPL	Info
00 (Selector 0x0000)	0x0	0x0	0	Unused
01 (Selector 0x0008)	0x0	0x7FFFFF	0	32-bit code
02 (Selector 0x0010)	0x0	0x7FFFFF	0	32-bit data
03 (Selector 0x0018)	0xB8000	0x2FFF	0	32-bit data
04 (Selector 0x0020)	0xBF8	0x68	3	Available 32bit TSS
05 (Selector 0x0028)	0xBE0	0x40	3	LDT
06 (Selector 0x0030)	0xE78	0x68	3	Busy 32bit TSS
07 (Selector 0x0038)	0xE60	0x40	3	LDT

再 info tss 看一下 tss 的值：



再单步执行：观察 tss 和寄存器的值：

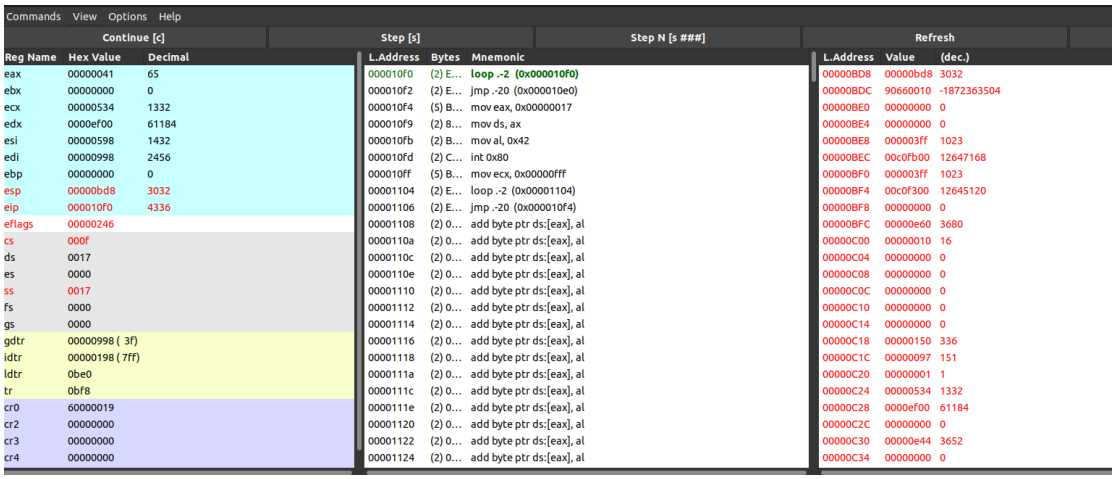
Reg Name	Hex Value	Decimal
eax	00000001	1
ebx	00000000	0
ecx	00000534	1332
edx	0000ef00	61184
esi	00000598	1432
edi	00000998	2456
ebp	00000000	0
esp	00000e44	3652
eip	00000150	336
eflags	00000097	
cs	0008	
ds	0010	
es	0000	
ss	0010	
fs	0000	
gs	0000	
gdtr	00000998 (3f)	
idtr	00000198 (7ff)	
ldtr	0be0	
tr	0bf8	
cr0	60000019	
cr2	00000000	
cr3	00000000	
cr4	00000000	

```
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x00000150
eflags: 0x00000097
cs: 0x0008 ds: 0x0010 ss: 0x0010
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000001 ebx: 0x00000000 ecx: 0x00000534 edx: 0x0000ef00
esi: 0x00000598 edi: 0x00000998 ebp: 0x00000000 esp: 0x00000e44
ldt: 0x0028
i/o map: 0x0800
```

第二次任务切换后，由于第一次任务切换时将寄存器现场保存到了 TSS0 里，因此将 TSS0 切换回来后，CS:EIP 会指向第一次任务切换的下一条地址，也就是 0x08:0x0150；

和上述分析一样，寄存器值和 TSS 的字段是一致的，虽然 TSS0 的大部分字段初值都是 0，但是在这里却并非全是，因为它们保存了第一次任务切换时的现场。

然后需要再单步执行几下到 iret 指令，单步执行，中断返回：



返回了 task0 用户之前发生中断的地方，好像什么都没发生，但实际上已经发生两次任务切换了。

请详细总结任务切换的过程。

task0->task1

第一次在 227 行发生时钟中断，此时任务号为 0，当跳入时钟中断后，从 125 行开始进行转换，首先将 1 写入寄存器 eax，然后判断，因为此时任务号为 0，所以不执行 127 行，执行 128 行，将任务号改为 1，然后在 129 行跳转到任务 1 的代码段。

task1->task0

从代码段 228 执行，在 232 循环，在循环时进行时间中断发生，在 126 行比较时，由于此时任务号为 1，所以跳转到 131 行；将任务号设为 0，在 132 行处，跳转到任务 0，由于任务 0 执行到 129 行，所以从 130 执行，再跳转到 133 行，弹栈后通过 iret 返回，由于上次为在 226 行触发中断，所以回到 226 行。

task0->task1

然后在此行，再次遇到时间中断，从 125 行开始进行转换，首先将 1 写入寄存器 eax，然后判断，因为此时任务号为 0，所以不执行 127 行，执行 128 行，则将任务号改为 1，然后在 129 行跳转到任务 1。由于任务 1 上次执行至 132 行，所以从 133 行开始，到 iret 返回至 232 行。

task1->task0

在 232 行遇到时钟中断，执行到 126 行比较后跳转到 131 行，将任务号变为 0，回到任务 0 的代码段。