2. 调试分析 Linux 0.00 多任务切换

2.1. 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；

2.2. 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。

2.2.1. 掌握 Bochs 虚拟机的调试技巧

直接 cv 的实验 1 中做完的内容

- 如何单步跟踪？

  输入 s 命令即可单步执行，每次输入 s 命令，虚拟机将执行当前指令并停在下一条指令之前。

- 如何设置断点进行调试？

  使用 b +地址命令设置断点，再点击运行，bochs 就会在断点位置停止

- 如何查看通用寄存器的值？

  调试界面左边蓝色部分：

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？

  调试界面左侧紫色部分：

cr0	60000010
cr2	00000000
cr3	00000000
cr4	00000000
efer	00000000

- 如何查看内存指定位置的值？

  使用如下命令（x）：

  x /<count><format> <address>

  # <count>：指定要查看的数据项数量。

  # <format>：指定数据的显示格式，如 b（字节）、w（字）、d（双字）等。

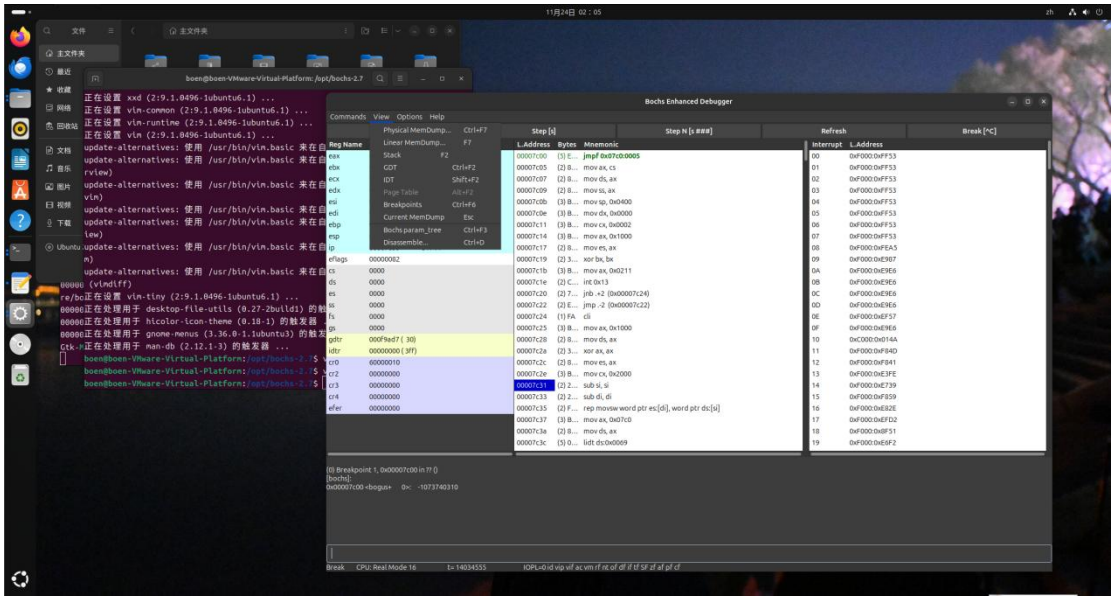
  # <address>：指定要查看的内存地址。

X /1wd 0x7c00:

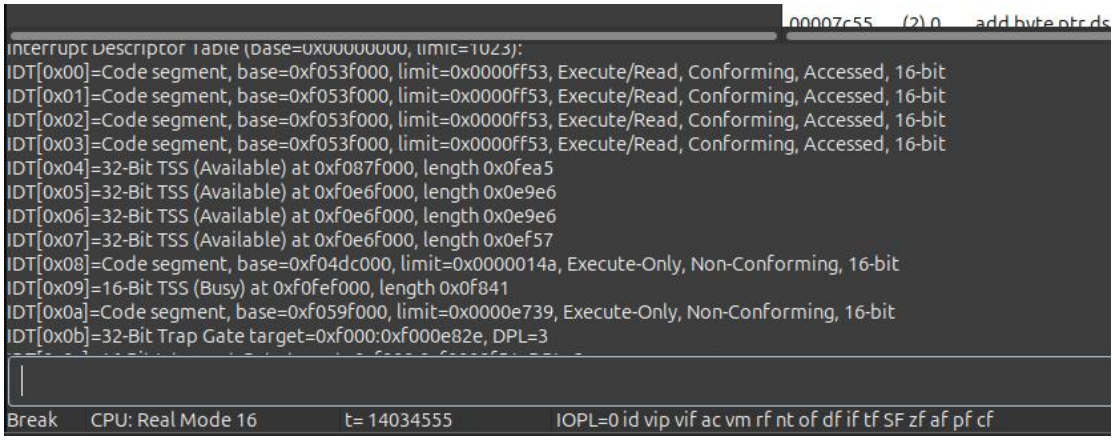
```
[bochs]:
0x00007c00 <bogus+     0>:   -1073740310
```

· 如何查看各种表，如 gdt ， idt ， ldt 等？

点击调试界面左上角的 view ， 即可选择查看 GDT ， IDT ， STACK 等，打开后其内容显示在调试器右边部分的界面，LDT ， TSS 可以从 GDT 中得到基址间接查看。

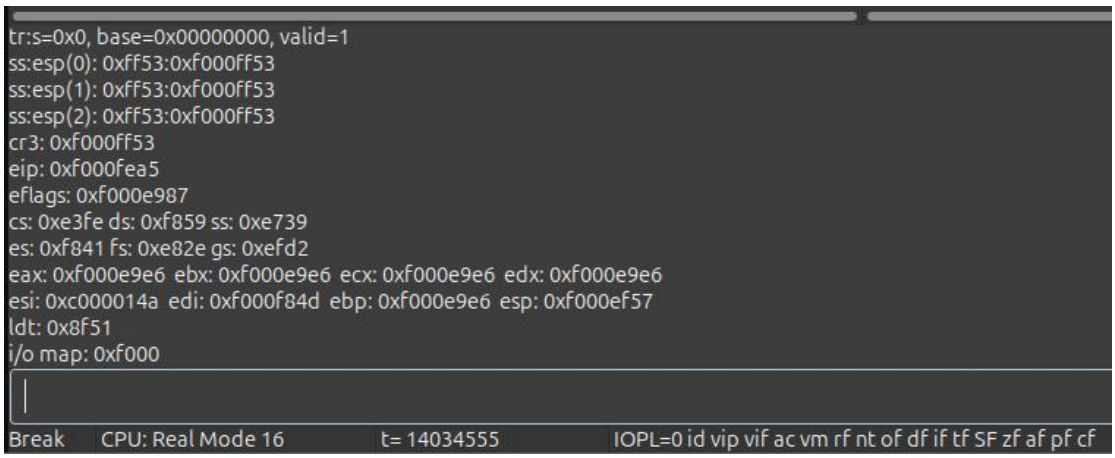


可以使用 info+表名查看表中的内容（info idt（小写））：



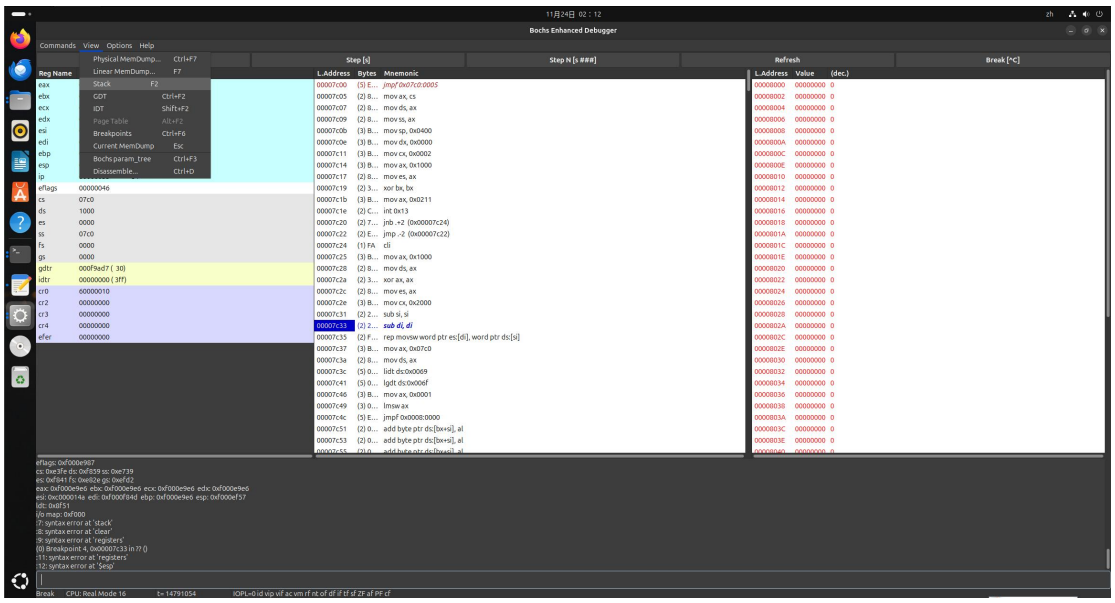
· 如何查看 TSS?

同上 info tss（小写）：



· 如何查看栈中的内容？

同同上上，在断点时，使用 view 中的 stack 查看：



· 如何在内存指定地方进行反汇编？

使用 disasm 命令：

disasm <address> <count>

<address>：指定要反汇编的内存地址。

<count>：指定要反汇编的指令数量。

disasm 0x1234 10:

2.3. 实验报告

通过仔细的调试与跟踪程序，完成以下任务：

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

首先阅读理解 system_interrupt 中的内容：

```
155     system_interrupt:
156         push %ds
157         pushl %edx
158         pushl %ecx
159         pushl %ebx
160         pushl %eax
161         movl $0x10, %edx    #让DS指向内核数据段
162         mov %dx, %ds
163         call write_char     #调用显示字符子程序，显示AL中的字符
164         popl %eax
165         popl %ebx
166         popl %ecx
167         popl %edx
168         pop %ds
169         iret
```

iret 用于在处理器状态转移期间从中断或异常处理程序返回到被中断的程序，还原被中断程序的执行环境，包括寄存器、堆栈以及特权级别的状态。

iret 指令执行以下操作：

从堆栈中弹出 EIP 寄存器的值，以恢复中断或异常处理程序返回到的下一条指令的地址。

从堆栈中弹出 CS 寄存器的值，以恢复中断或异常处理程序返回到的代码段。

从堆栈中弹出标志寄存器 EFLAGS 的值，以恢复标志寄存器的状态。

如果在中断或异常处理程序执行期间切换了堆栈，iret 会从堆栈中弹出新的 ESP 寄存器的值，以恢复原始堆栈。

根据从堆栈中弹出的 CS 寄存器的值，恢复到适当的特权级别。这允许在不同特权级别（如内核态和用户态）之间切换。

进入 bochs，调试找到 iret 的地址为 0x17，在 0x17c 处设置断点，后运行至断点处：

如下图，可以观察到在执行 iret（也就是 0x17c）之前的栈状态：

CPU 正在执行的下一条指令的内存地址 EIP：0x17c（即将执行 iret）

代码段寄存器 CS：0x8（当前执行的代码段以及执行代码的特权级别为 0 内核态）

栈顶指针 ESP：0xe4e

此时 stack 中的值从栈顶开始是：

0x10eb（即将弹出的 EIP 寄存器的值） 0x000f （即将弹出的 CS 寄存器的值）

0x0246（即将弹出的 EFLAGS 寄存器的值） 0x0bd8（即将弹出的 ESP 寄存器的值）

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	
fs	0000	
gs	0000	
gdtr	00000998 (3f)	
idtr	00000198 (7ff)	
ldtr	0be0	
tr	0bf8	
cr0	60000011	

L.Address	Bytes	Mnemonic
0000017c	(1) CF	iret
0000017d	(2) 0...	add byte ptr ds:[eax], al
0000017f	(2) 0...	add byte ptr ds:[eax], al
00000181	(2) 0...	add dword ptr ds:[eax], eax
00000183	(2) 0...	add byte ptr ds:[eax], al
00000185	(1) 90	nop
00000186	(2) F...	inc dword ptr ds:[edi]
00000188	(1) 98	cwde
00000189	(2) 0...	add dword ptr ds:[eax], eax
0000018b	(2) 0...	add byte ptr ds:[edi], bh
0000018d	(6) 0...	add byte ptr ds:[eax-1879048183], bl
00000193	(5) 2...	lea esi, cs:[esi]
00000198	(2) 1...	adc al, 0x01
0000019a	(2) 0...	or byte ptr ds:[eax], al
0000019c	(6) 0...	add byte ptr ds:[esi+18087936], cl
000001a2	(2) 0...	or byte ptr ds:[eax], al
000001a4	(6) 0...	add byte ptr ds:[esi+18087936], cl
000001aa	(2) 0...	or byte ptr ds:[eax], al
000001ac	(6) 0...	add byte ptr ds:[esi+18087936], cl
000001b2	(2) 0...	or byte ptr ds:[eax], al
000001b4	(6) 0...	add byte ptr ds:[esi+18087936], cl

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
00000E84	00000000	0
00000E88	00000000	0
00000E8C	00000000	0
00000E90	00000000	0
00000E94	00000000	0
00000E98	000010f4	4340
00000E9C	00000200	512

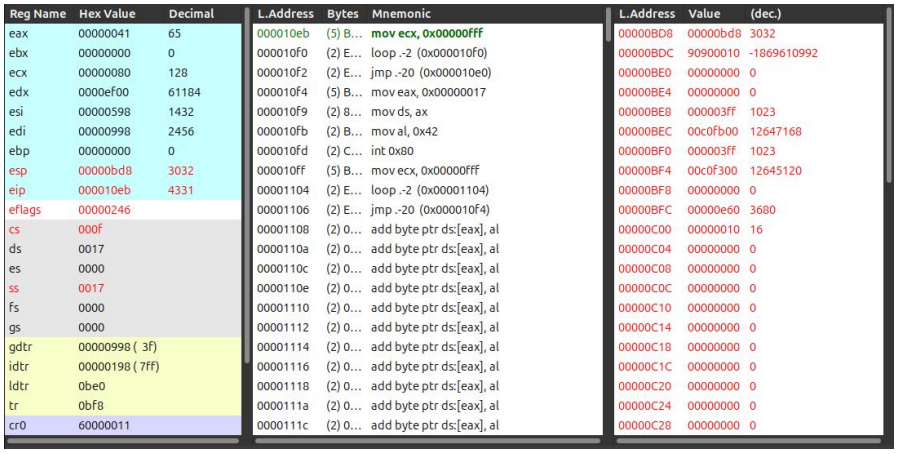
单步运行，检查执行 iret 命令之后 cpu 状态：

执行命令之后，如下图，栈的变化情况为：

CPU 正在执行的下一条指令的内存地址 EIP：0x10eb

代码段寄存器 CS：0xf（当前执行的代码段以及执行代码的特权级别为 3 用户态）

栈顶指针 ESP：0xbd8



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	
fs	0000	
gs	0000	
gdtr	00000998 (3f)	
idtr	00000198 (7ff)	
ldtr	0be0	
tr	0bf8	
cr0	60000011	

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
00001110	(2) 0...	add byte ptr ds:[eax], al
00001112	(2) 0...	add byte ptr ds:[eax], al
00001114	(2) 0...	add byte ptr ds:[eax], al
00001116	(2) 0...	add byte ptr ds:[eax], al
00001118	(2) 0...	add byte ptr ds:[eax], al
0000111a	(2) 0...	add byte ptr ds:[eax], al
0000111c	(2) 0...	add byte ptr ds:[eax], al

L.Address	Value	(dec.)
00000BD8	00000bd8	3032
00000BDC	90900010	-1869610992
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
00000C10	00000000	0
00000C14	00000000	0
00000C18	00000000	0
00000C1C	00000000	0
00000C20	00000000	0
00000C24	00000000	0
00000C28	00000000	0

·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？

1）特权级

在进入和退出 system_interrupt 时发生了特权级的模式切换。

进入 system_interrupt：

用户态（特权级 3）切换到内核态（特权级 0），中断服务在内核态中执行。

处理器将特权级从 3 切换到 0，CS 的值由 0xf 变为 0x8。

退出 system_interrupt：

内核态（特权级 0）切换回用户态（特权级 3）。

iret 执行之后，处理器将特权级从 0 切换到 3，CS 的值由 0x8 变为 0xf。

2）栈

在进入和退出 system_interrupt 时同时会发生栈的切换。

进入 system_interrupt：

发生堆栈切换，以便在内核态中使用内核堆栈，来保护用户堆栈的完整性。

退出 system_interrupt：

发生堆栈切换，以返回到原特权级的堆栈，并继续用户态的执行。

3）如何切换

这种特权级的切换和堆栈切换是操作系统内核和处理器硬件协同工作的结果。

保存当前上下文：在模式切换之前，当前执行的任务的上下文以及当前特权级需要被保存压入栈中

选择新特权级别：根据要切换到的特权级别，操作系统选择新的代码段描述符和堆栈描述符，切换堆栈和特权级，并将它们加载到相应的寄存器中。

在进入内核态时，通常会使用内核堆栈，以避免破坏用户堆栈。

在退出内核态时，特权级和堆栈状态会恢复，以确保程序的正常执行。

·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。

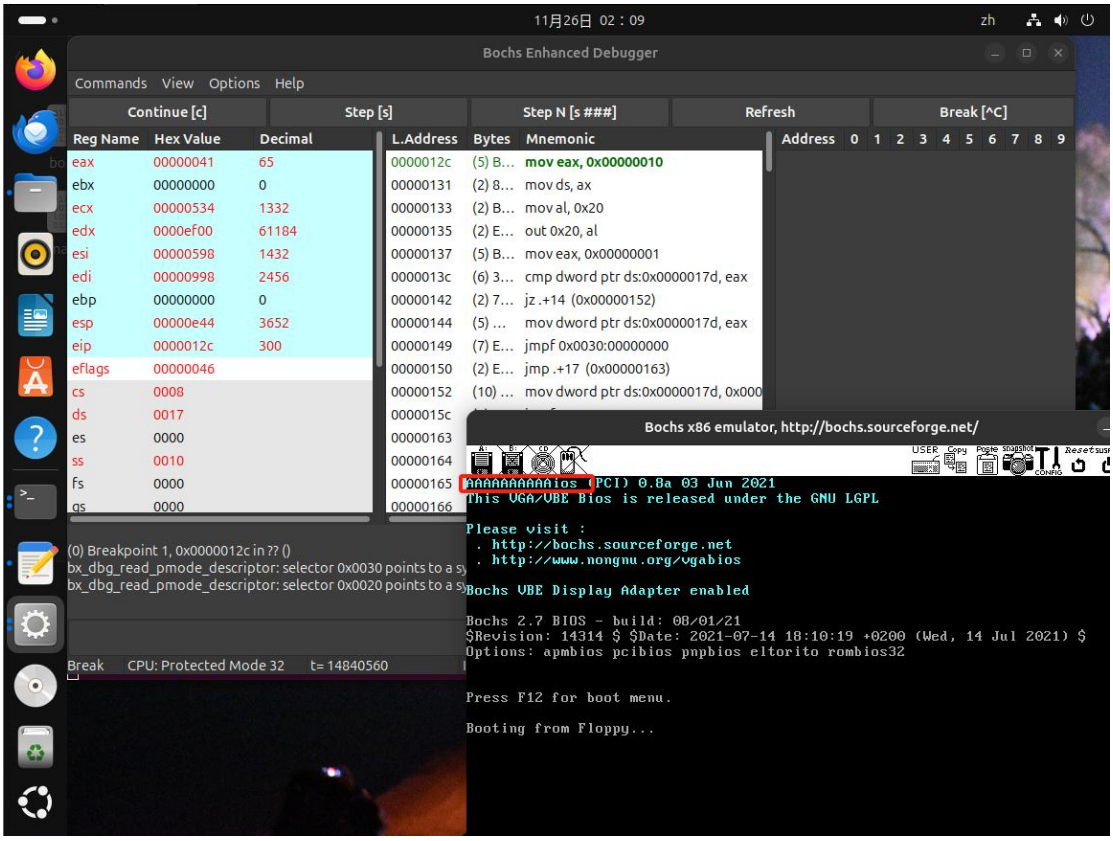
首先阅读理解 timer_interrupt 中的内容：

```
timer_interrupt:
    push %ds
    pushl %eax
    movl $0x10, %eax        #让DS指向内核数据段
    mov %ax, %ds
    movb $0x20, %al         #允许其他硬件中断（向8259A发送EOI命令）
    outb %al, $0x20
    movl $1, %eax           #判断当前任务，如果是任务1就去执行任务0，是任务0就去执行任务1
    cmpl %eax, current
    je 1f
    movl %eax, current      #如果当前任务是0，把1存入current，跳转到1
    ljmp $TSS1_SEL, $0      #执行跳转
    jmp 2f
1:  movl $0, current        #如果...(和上面的全反过来)
    ljmp $TSS0_SEL, $0      #执行跳转
2:  popl %eax
    pop %ds
    iret
```

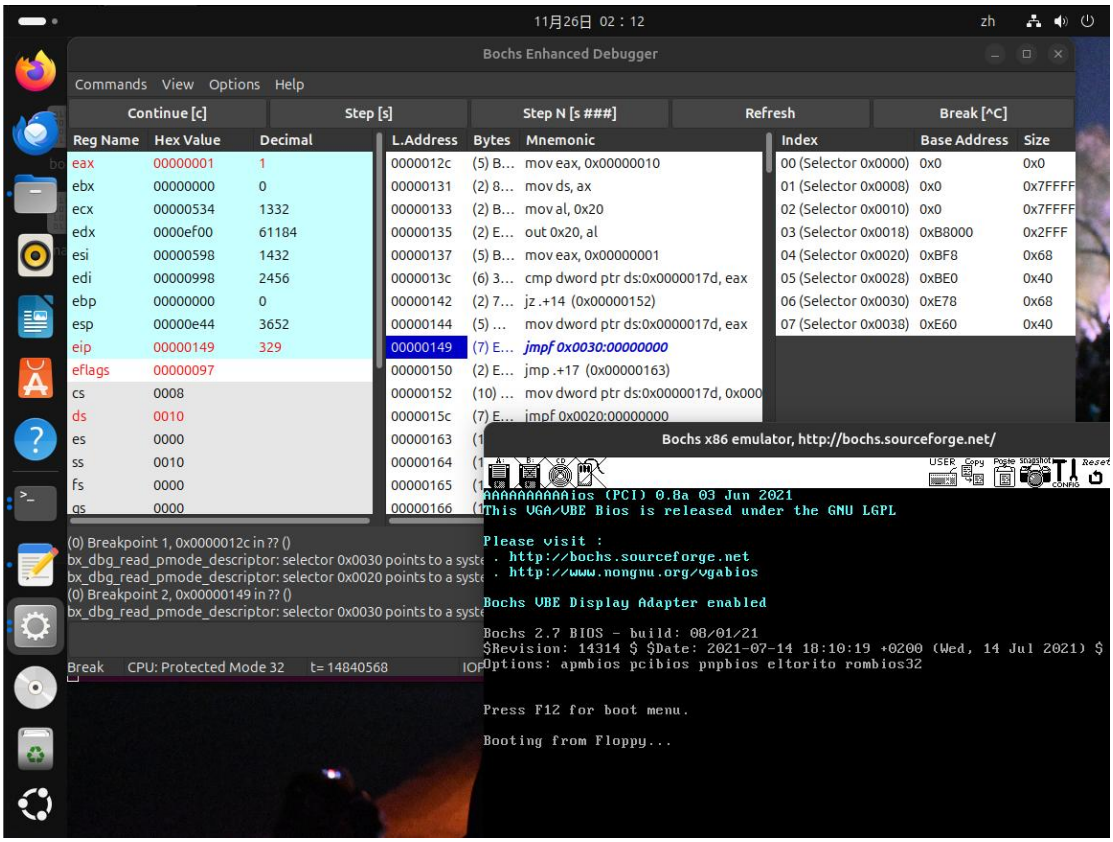
发现有 task0 和 1 的出现，观察发现 task0 和 1 分别是打印 A 和 B（字母），如下图：

```
    /*****************************************/
task0:
    movl $0x17, %eax        #让DS指向任务的局部数据段
    movw %ax, %ds           # ?
    movb $65, %al              /* print 'A' */
    int $0x80               #显示打印的字符
    movl $0xfff, %ecx       #循环 延时
1:  loop 1b
    jmp task0               #继续打印显示

task1:
    movl $0x17, %eax
    movw %ax, %ds           # ?
    movb $66, %al              /* print 'B' */
    int $0x80               #显示
    movl $0xfff, %ecx
1:  loop 1b
    jmp task1

    .fill 128,4,0           #看起来是栈空间
usr_stk1:
```

推测是先将任务 0 的执行结果存储到寄存器中，再运行任务 1，先在首部（0x12c 处）设置断点观察运行结果：



如图（框住的部分）可知程序进行了 A 的打印但是没有打印 B，也就是只执行了 task0 还未执行 task1，单步运行到 CS:EIP 0x08:0x0149 处准备执行指令 jmpf 0x30:0 ，即远跳转到 0x30:0，将一个 TSS 选择子（0x30）装入 CS，实际上就是为了将任务切换到这个 TSS，而 0x30 恰好是 task1 的 TSS，如下图：

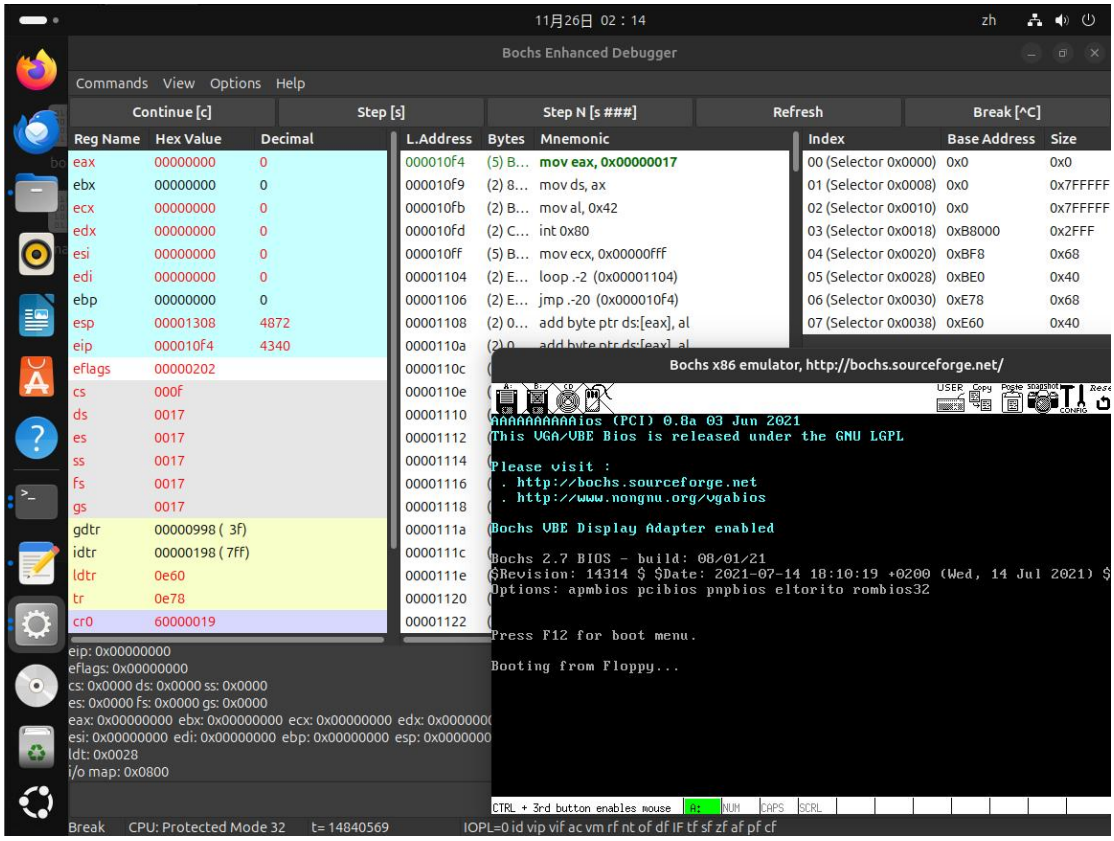


输入输入 info tss，记录任务切换前 task0 的 TSS：

```
tr:s=0x20, base=0x00000bf8, valid=1
ss:esp(0): 0x0010:0x00000e60
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
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
```

单步执行后，寄存器状态如下图 CS:EIP 0xf:0x10f4，现在已经切换到任务 1，由于任务 1 是第一次执行，直接跳转到了用户程序 task1 的入口处：

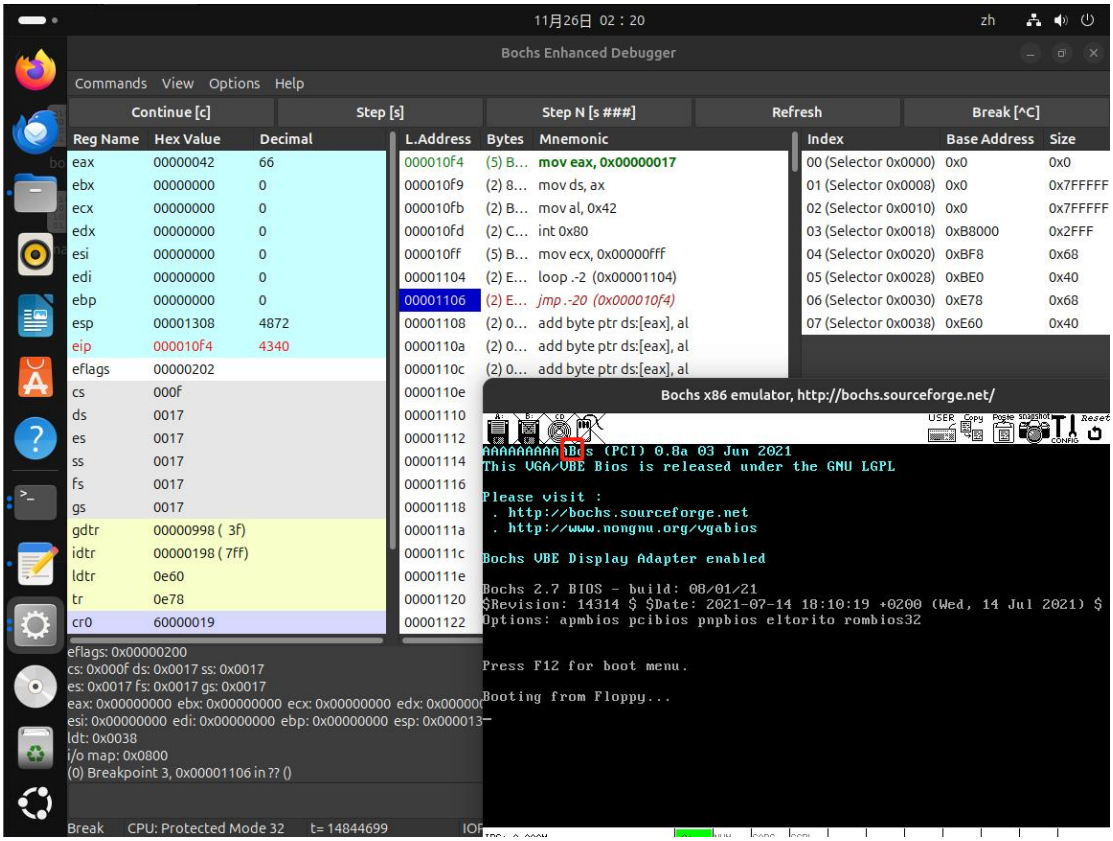


输入 info tss，查看此时的 TSS，发现 TSS 变化为 task1 的 TSS，并且与寄存器中的值相同，这说明了任务切换时会根据对应任务的 TSS 的各个字段修改寄存器。

```
tr:s=0x30, base=0x00000e78, valid=1
ss:esp(0): 0x0010:0x000010e0
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
```

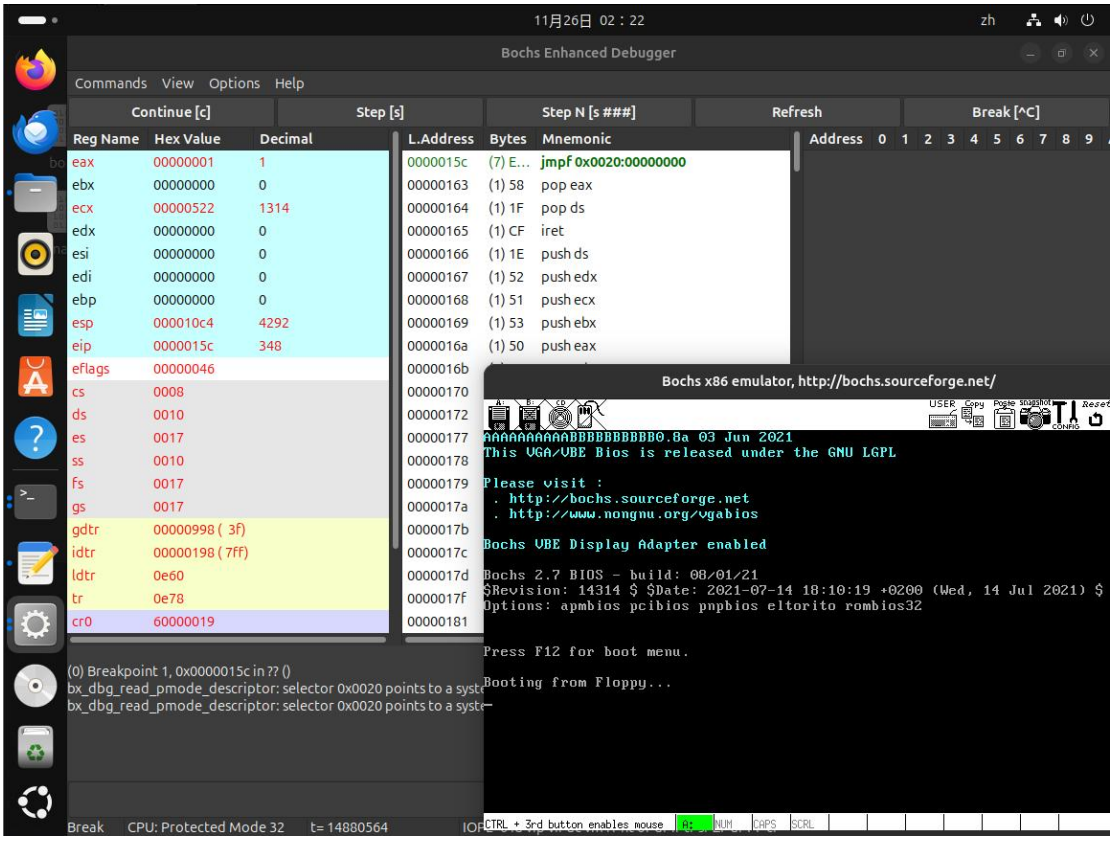
```
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00001308
ldt: 0x0038
i/o map: 0x0800
```

末尾设置断点，运行至断点后发现已经打印了 B，再次单步运行，跳回 10f4，与之前对代码的注释以及猜想相同。



· 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？

在 0x15c 处设置断点，然后运行程序：



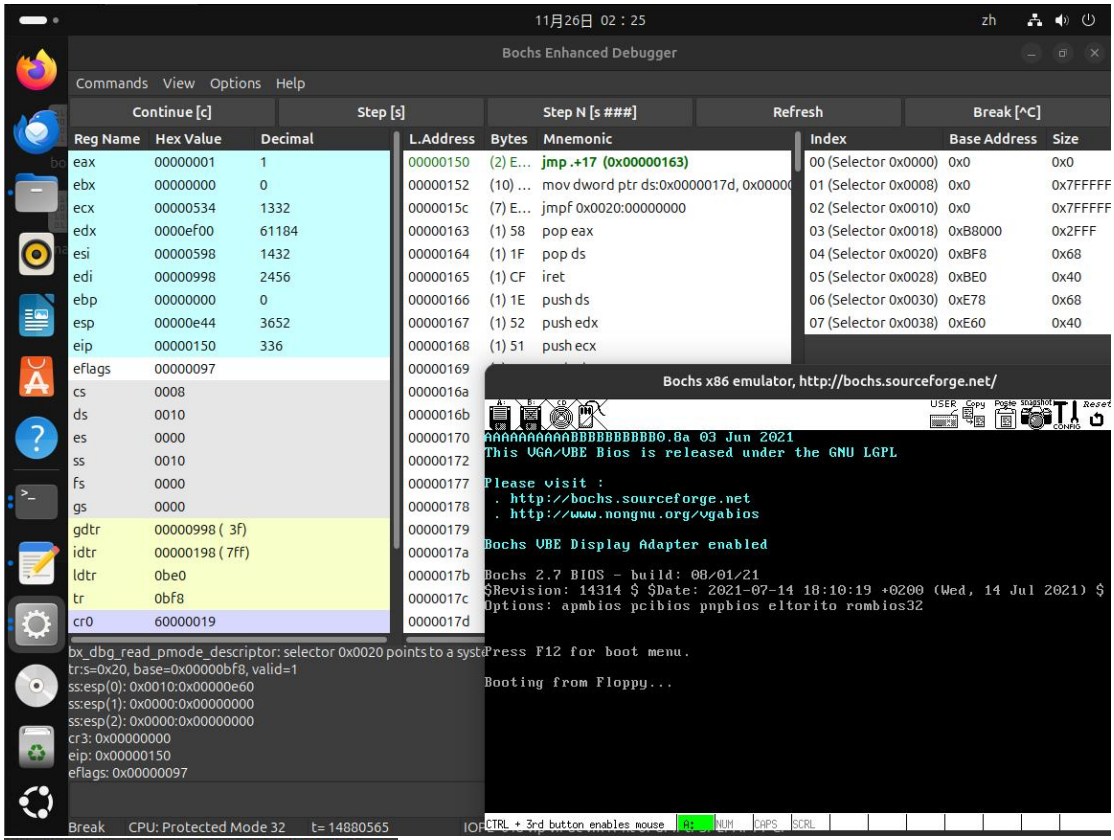
可以看到 AB 均被打印，可知 程序已经执行了 task1，输入 info tss 检查 TSS 中内容：

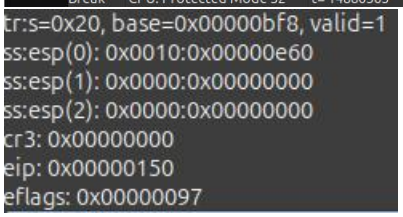
```
tr:s=0x30, base=0x00000e78, valid=1
ss:esp(0): 0x0010:0x000010e0
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
```

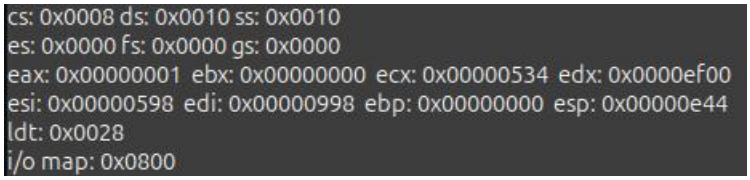
```
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00001308
ldt: 0x0038
i/o map: 0x0800
```

与寄存器内容相同，说明已经将 task1 的上下文保存在 task1 的 TSS 中。

然后点击单步运行，执行 jmpf 0x20:0，将一个 TSS 选择子（0x20）装入 CS，即 task0 的 TSS，由于第一次任务切换时将寄存器现场保存到了 task0 TSS 里，因此将 TSS 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 0x8:0x150，运行结果以及 TSS 信息如下：







· 请详细总结任务切换的过程。

时钟中断触发：任务切换通常是由系统中的时钟中断 timer_interrupt 触发的。时钟中断以固定的时间间隔（每 10 毫秒）发生一次，它是多任务处理的触发点。

保存当前任务上下文：当时钟中断触发时，操作系统会执行时钟中断处理程序。在处理程序执行之前，当前正在执行的 taskA 的上下文会被保存。EAX、ECX、EDX、EFLAGS、ESP、CS、EIP 等寄存器的状态会保存在 taskA 的 TSS 中。

选择下一个任务：在时钟中断处理程序中，操作系统会选择下一个要执行的 taskB ，任务 B 的上下文信息存储在 taskB 的 TSS 中。

加载下一个任务上下文：时钟中断处理程序通过 taskB 的 TSS 将 taskB 的上下文信息（寄存器的值）加载到处理器中，处理器现在准备执行 taskB 。

切换堆栈：如果 taskA 和 taskB 使用不同的内核堆栈，堆栈指针寄存器 ESP 将从 taskA 的内核堆栈切换到 taskB 的内核堆栈，以确保 taskB 可以正常执行内核代码。

特权级别切换：如果 taskA 和 taskB 属于不同的特权级别，时钟中断处理程序会执行特权级别切换操作。

taskB 开始执行： taskB 的上下文准备就绪并加载到处理器中， taskB 开始执行。

时钟中断返回：时钟中断处理程序执行完毕后，处理器返回到 taskB 的执行点， taskB 继续执行。

在这个过程中，TSS 寄存器中的值将发生变化，以反映新任务的上下文。旧任务的上下文信息已经被保存，以便在未来的任务切换中恢复。其他寄存器的值也会根据任务上下文的不同而变化，以确保任务切换的正确执行。这个过程允许多个任务在同一个系统中轮流执行，实现多任务处理。

2.3.1. 评分标准

记录描述要详细完整，前 4 项每题 20%，总共 80%
总结任务切换过程，10%
格式规范美观，10%