

## 操作系统实验二 调试分析 Linux 0.00 多任务切换

### 一、实验目的

- 通过调试一个简单的多任务内核实例，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；

### 二、实验环境

Vmware 17.5.1, Ubuntu 20.04, Bochs 2.4.6

### 三、实验内容

通过调试一个简单的多任务内核实例，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。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。

### 四、实验过程

#### 3.1 当执行完 system\_interrupt 函数，执行 153 行 iret 时，记录栈的变化情况

查阅《Linux 内核完全注释 v5.0》145 页，system\_interrupt 程序内容如下：

```
137 # 系统调用中断 int 0x80 处理程序。该示例只有一个显示字符功能。
138 .align 2
139 system_interrupt:
140     push %ds
141     pushl %edx
142     pushl %ecx
143     pushl %ebx
144     pushl %eax
145     movl $0x10, %edx          # 首先让 DS 指向内核数据段。
146     mov %dx, %ds
147     call write_char          # 然后调用显示字符子程序 write_char，显示 AL 中的字符。
148     popl %eax
149     popl %ebx
150     popl %ecx
151     popl %edx
152     pop %ds
153     iret
```

iret 指令用于从中断处理程序返回，并恢复到中断发生前的程序状态，包括寄存器、堆栈以及特权级别的状态，其执行后会从堆栈弹出以下值：

- CS 和 EIP，以切换到适当特权级别，并恢复被中断程序的代码指针；
- EFLAG，以恢复被中断程序的标志位；
- SS 和 ESP，以恢复被中断程序的堆栈指针；

打开 Bochs 的调试界面，在 0x0 处设置一个断点，此处为 head 程序的入口。对照 head.s 文件找到调用 system\_interrupt 函数的入口为 0x0166，在此处设置一个断点，运行程序会调用系统中断，进入 system\_interrupt 程序，之后单步执行至 iret 指令。

(1) iret 指令执行前：

查看 Stack 内容如下，esp 寄存器值为 0x0e4c，指向栈顶；cs 寄存器值为 0x8，表示当前执行代码的特权级别为 0（内核态）；栈顶为即将弹出的几个值，从栈顶向下依次为 0x10eb（对应 eip 寄存器）、0x000f（对应 cs 寄存器）、0x0246（对应 eflags 寄存器）、0x0bd8（对应 esp 寄存器）。

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic	L.Address	Value	(dec.)	
eax	00000041	65	00000142	(2) 7...	jz .+14 (0x0000152)	00000E4C	000010eb	4331	
ebx	00000000	0	00000144	(5) A...	mov dword ptr ds:0x17d, eax	00000E50	0000000f	15	
ecx	00000080	128	00000149	(7) E...	jmp far 0030:00000000	00000E54	00000246	582	
edx	0000ef00	61184	00000150	(2) E...	jmp .+17 (0x00000163)	00000E58	00000bd8	3032	
esi	00000598	1432	00000152	(10) ...	mov dword ptr ds:0x17d, 0x00000000	00000E5C	00000017	23	
edi	00000998	2456	0000015c	(7) E...	jmp far 0020:00000000	00000E60	00000000	0	
ebp	00000000	0	00000163	(1) 58	pop eax	00000E64	00000000	0	
esp	00000e4c	3660	00000164	(1) 1F	pop ds	00000E68	000003ff	1023	
eip	0000017c	380	00000165	(1) CF	iretd	00000E6C	00c0fa00	12640912	
eflags	00000283		00000166	(1) 1E	push ds	00000E70	000003ff	1023	
cs	0008		00000167	(1) 52	push edx	00000E74	00c0f200	12644864	
ds	0017		00000168	(1) 51	push ecx	00000E78	00000000	0	
es	0000		00000169	(1) 53	push ebx	00000E7C	000010e0	4320	
ss	0010		0000016a	(1) 50	push eax	00000E80	00000010	16	
fs	0000		0000016b	(5) B...	mov edx, 0x00000010	00000E84	00000000	0	
gs	0000		00000170	(2) 8...	mov ds, dx	00000E88	00000000	0	
gdtr	00000998 ( 3f)		00000172	(5) E...	call .+146 (0x000000e5)	00000E8C	00000000	0	
ldtr	00000198 ( 7ff)		00000177	(1) 58	pop eax	00000E90	00000000	0	
ldtr	0be0		00000178	(1) 5B	pop ebx	00000E94	00000000	0	
tr	0bf8		00000179	(1) 59	pop ecx	00000E98	000010f4	4340	
cr0	60000011		0000017a	(1) 5A	pop edx	00000E9C	00000200	512	
cr2	00000000		0000017b	(1) 1F	pop ds	00000EA0	00000000	0	
cr3	00000000		0000017c	(1) CF	iretd	00000EA4	00000000	0	
cr4	00000000		0000017d	(2) 0...	add byte ptr ds:[eax], al	00000EA8	00000000	0	
efer	00000000		0000017f	(2) 0...	add byte ptr ds:[eax], al	00000EAC	00000000	0	
			00000181	(2) 0...	add byte ptr ds:[eax], al	00000EB0	00001308	4872	
			00000183	(2) 0...	add byte ptr ds:[eax], al	00000EB4	00000000	0	
			00000185	(1) 90	nop	00000EB8	00000000	0	

(2) iret 指令执行后:

单步执行程序，在 iret 指令执行后再次查看 Stack 内容如下，栈顶为 0x0bd8，对应弹出的 esp 寄存器的值；程序返回到 0x10eb 地址处，执行下一条指令，对应弹出的 eip 寄存器的值；eflags、cs 寄存器的值也与执行前栈内要弹出的值一一对应。

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic	L.Address	Value	(dec.)	
eax	00000041	65	000010eb	(5) B...	mov ecx, 0x00000fff	00000BD8	00000bd8	3032	
ebx	00000000	0	000010f0	(2) E...	loop .-2 (0x000010f0)	00000BDC	90660010	-1872363504	
ecx	00000080	128	000010f2	(2) E...	jmp .-20 (0x000010e0)	00000BE0	00000000	0	
edx	0000ef00	61184	000010f4	(5) B...	mov eax, 0x00000017	00000BE4	00000000	0	
esi	00000598	1432	000010f9	(2) 8...	mov ds, ax	00000BE8	000003ff	1023	
edi	00000998	2456	000010fb	(2) B...	mov al, 0x42	00000BEC	00c0fb00	12647168	
ebp	00000000	0	000010fd	(2) C...	int 0x80	00000BF0	000003ff	1023	
esp	00000bd8	3032	000010ff	(5) B...	mov ecx, 0x00000fff	00000BF4	00c0f300	12645120	
eip	000010eb	4331	00001104	(2) E...	loop .-2 (0x000010f0)	00000BF8	00000000	0	
eflags	00000246		00001106	(2) E...	jmp .-20 (0x000010f4)	00000BFC	00000e60	3680	
cs	000f		00001108	(2) 0...	add byte ptr ds:[eax], al	00000C00	00000010	16	
ds	0017		0000110a	(2) 0...	add byte ptr ds:[eax], al	00000C04	00000000	0	
es	0000		0000110c	(2) 0...	add byte ptr ds:[eax], al	00000C08	00000000	0	
ss	0017		0000110e	(2) 0...	add byte ptr ds:[eax], al	00000C0C	00000000	0	
fs	0000		00001110	(2) 0...	add byte ptr ds:[eax], al	00000C10	00000000	0	
gs	0000		00001112	(2) 0...	add byte ptr ds:[eax], al	00000C14	00000000	0	
gdtr	00000998 ( 3f)		00001114	(2) 0...	add byte ptr ds:[eax], al	00000C18	00000000	0	
ldtr	00000198 ( 7ff)		00001116	(2) 0...	add byte ptr ds:[eax], al	00000C1C	00000000	0	
ldtr	0be0		00001118	(2) 0...	add byte ptr ds:[eax], al	00000C20	00000000	0	
tr	0bf8		0000111a	(2) 0...	add byte ptr ds:[eax], al	00000C24	00000000	0	
cr0	60000011		0000111c	(2) 0...	add byte ptr ds:[eax], al	00000C28	00000000	0	
cr2	00000000		0000111e	(2) 0...	add byte ptr ds:[eax], al	00000C2C	00000000	0	
cr3	00000000		00001120	(2) 0...	add byte ptr ds:[eax], al	00000C30	00000000	0	
cr4	00000000		00001122	(2) 0...	add byte ptr ds:[eax], al	00000C34	00000000	0	
efer	00000000		00001124	(2) 0...	add byte ptr ds:[eax], al	00000C38	00000000	0	
			00001126	(2) 0...	add byte ptr ds:[eax], al	00000C3C	00000000	0	
			00001128	(2) 0...	add byte ptr ds:[eax], al	00000C40	00000000	0	
			0000112a	(2) 0...	add byte ptr ds:[eax], al	00000C44	00000000	0	

3.2 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？

模式切换时，操作系统会根据要切换到的特权级别，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。

### 3.1.1 特权级改变

当进入或退出系统调用中断时，特权级的变化具体表现为 CPU 从用户态切换到内核态，或从内核态切换回用户态。

(1) 进入中断时的特权级改变

CPU 从用户空间程序中断执行，并进入内核空间执行中断处理程序。操作系统会利用中断门来触发这一切，CPU 将特权级从 3 提升至 0，cs 寄存器的值由 0x8 变为 0xf，确

保中断处理程序能够访问更高权限的资源（如硬件、内存管理等），并且执行的代码能获得更高的安全性和控制权限。

#### （2）退出中断时的特权级改变

当中断处理程序执行完毕后，CPU 将从内核态切换回用户态，以继续执行原来的用户程序。操作系统会通过中断返回指令（如 `iret`）来实现这一切换，CPU 将特权级从 0 降低至 3，并恢复之前的栈和寄存器状态，`cs` 寄存器的值由 `0x8` 变为 `0xf`。

### 3.1.2 栈切换

中断处理期间需要保存当前的执行上下文，以确保系统能够在中断处理后恢复到正确的状态。因此当进入或退出系统调用中断时，栈也进行切换，具体表现为从用户栈切换到内核栈或从内核栈切换回用户栈。

#### （1）进入中断时的栈切换

进入中断时，处理器硬件会自动根据中断门切换到内核模式，与操作系统内核协调工作，将中断前的状态（如寄存器值、PC 等）保存在内核栈中，之后 CPU 切换到进程的内核栈，进行后续中断处理。

#### （2）退出中断时的栈切换

中断处理程序执行完后，操作系统会使用 `iret` 指令，从内核栈恢复之前保存的用户程序的上下文，来恢复中断前的执行状态。该指令会恢复用户程序的栈指针和 PC 等，切换回用户态，继续用户程序的执行。

## 3.3 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

查阅《Linux 内核完全注释 v5.0》144-145 页，`timer_interrupt` 程序内容如下：

```
116 # 这是定时中断处理程序。其中主要执行任务切换操作。
117 .align 2
118 timer_interrupt:
119     push %ds
120     pushl %eax
121     movl $0x10, %eax           # 首先让 DS 指向内核数据段。
122
123     mov %ax, %ds
124     movb $0x20, %al           # 然后立刻允许其他硬件中断，即向 8259A 发送 EOI 命令。
125     outb %al, $0x20
126     movl $1, %eax             # 接着判断当前任务，若是任务 1 则去执行任务 0，或反之。
127     cmpl %eax, current
128     je 1f
129     movl %eax, current         # 若当前任务是 0，则把 1 存入 current，并跳转到任务 1
130     jmp $TSS1_SEL, $0         # 去执行。注意跳转的偏移值无用，但需要写上。
131 1:    movl $0, current         # 若当前任务是 1，则把 0 存入 current，并跳转到任务 0
132     jmp $TSS0_SEL, $0         # 去执行。
133 2:    popl %eax
134     pop %ds
135     iret
```

`task0` 程序内容如下：

```
219 # 下面是任务 0 和任务 1 的程序，它们分别循环显示字符'A'和'B'。
220 task0:
221     movl $0x17, %eax          # 首先让 DS 指向任务的局部数据段。
222     movw %ax, %ds             # 因为任务没有使用局部数据，所以这两句可省略。
223     movl $65, %al             # 把需要显示的字符'A'放入 AL 寄存器中。
224     int $0x80                 # 执行系统调用，显示字符。
225     movl $0xffff, %ecx        # 执行循环，起延时作用。
226 1:    loop 1b
227     jmp task0                 # 跳转到任务代码开始处继续显示字符。
```

`task1` 程序内容如下：

```

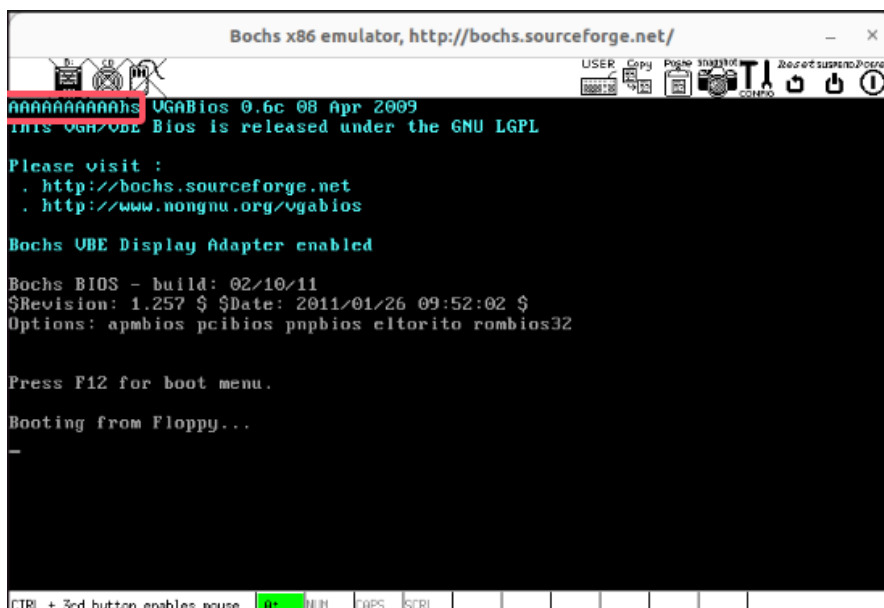
228 task1:
229     movl $66, %al          # 把需要显示的字符' B' 放入 AL 寄存器中。
230     int $0x80              # 执行系统调用, 显示字符。
231     movl $0xffff, %ecx     # 延时一段时间, 并跳转到开始处继续循环显示。
232 1:   loop lb
233     jmp task1
234
235     .fill 128, 4, 0        # 这是任务 1 的用户栈空间。
236 usr_stk1:

```

与查找 system\_interrupt 程序入口方法同理, 找到 timer\_interrupt 程序入口地址为 0x012a。在 0x012c 处设置一个断点, 运行程序进入 timer\_interrupt 程序。

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L Address	Bytes	Mnemonic				Base Address
eax	00000041	65	00000129	(1) CF	iretd				0x0
ebx	00000000	0	0000012a	(1) 1E	push ds				0x0
ecx	00000530	1328	0000012b	(1) 50	push eax				0x0
edx	0000e000	61184	0000012c	(3) B...	mov eax, 0x00000010				0x0
edi	00000998	2456	00000131	(2) B...	mov ds, ax				0x0
edi	00000998	2456	00000133	(2) B...	mov al, 0x0				0x0
ebp	00000000	0	00000135	(2) E...	out 0x20, al				0x0
esp	00000e44	3652	00000137	(5) B...	mov eax, 0x00000001				0x0
esp	0000012c	300	0000013c	(6) 3...	cmp dword ptr ds:0x17d, eax				0x0
eflags	00000046		00000142	(2) 7...	jz +14 (0x00000152)				0x0
cs	0008		00000144	(5) A...	mov dword ptr ds:0x17d, eax				0x0
ds	0017		00000149	(7) E...	jmp far 0020:0x00000000				0x0
es	0000		00000150	(2) E...	jmp +17 (0x00000163)				0x0
fs	0010		00000152	(10) ...	mov dword ptr ds:0x17d, 0x00000000				0x0
fs	0000		0000015c	(7) E...	jmp far 0020:0x00000000				0x0
gs	0000		00000163	(1) 5B	pop eax				0x0
gdt	00000998 ( 3f)		00000164	(1) 1F	pop ds				0x0
ldtr	00000198 ( 7ff)		00000165	(1) CF	iretd				0x0
ldtr	0be0		00000166	(1) 1E	push ds				0x0
tr	0b76		00000167	(1) 52	push edx				0x0
cr0	60000011		00000168	(1) 51	push ecx				0x0
cr2	00000000		00000169	(1) 53	push ebx				0x0
cr3	00000000		0000016a	(1) 50	push eax				0x0
cr4	00000000		0000016b	(5) B...	mov edx, 0x00000010				0x0
efer	00000000		00000170	(2) B...	mov ds, dx				0x0
			00000172	(5) E...	call -146 (0x000000e5)				0x0
			00000177	(1) 5B	pop eax				0x0
			00000178	(1) 5B	pop ebx				0x0
			00000179	(1) 59	pop ecx				0x0
			0000017a	(1) 5A	pop edx				0x0
			0000017b	(1) 1F	pop ds				0x0
			0000017c	(1) CF	iretd				0x0
			0000017d	(2) 0...	add byte ptr ds:[eax], al				0x0

从输出窗口可以看到程序执行 task0, 打印了 10 个 A, 但并未打印 B, 即还未执行过 task1。



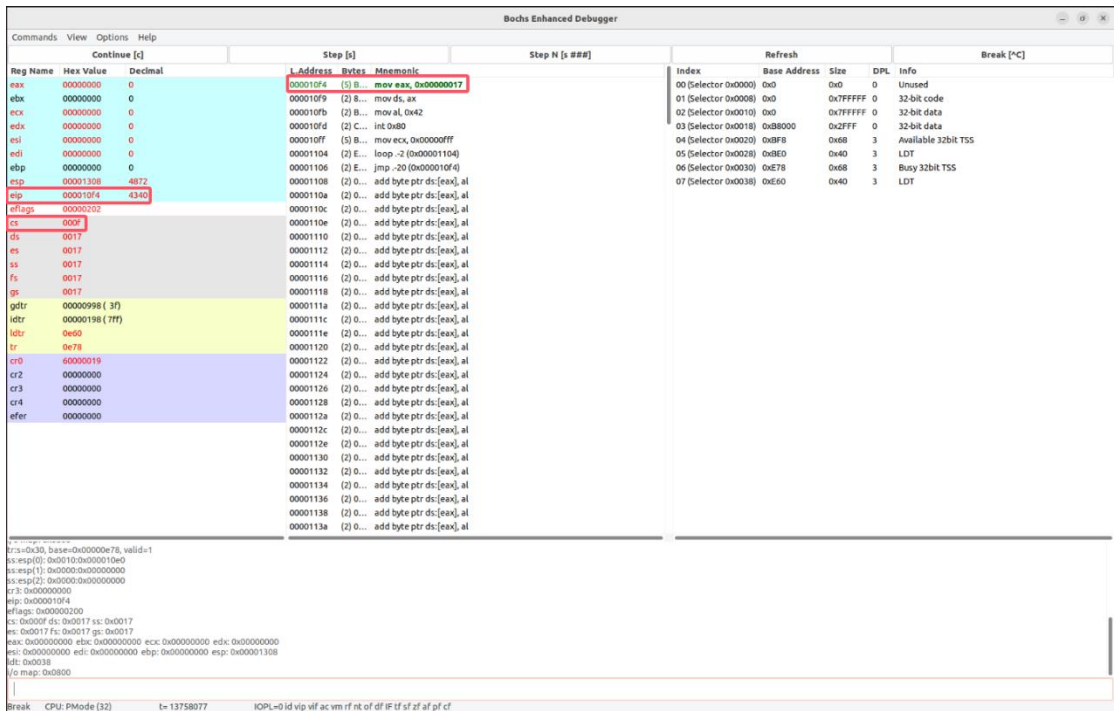
在下方命令行输入 “info tss”，查看任务切换前 task0 的 TSS。

```
tss=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
idt: 0x0028
i/o map: 0x0800
```

单步执行到 CS:EIP 0x08:0x0149 处准备执行 `jmp far 0x30:0` 指令，即远跳转到 0x30:0，将一个 TSS 选择子（0x30）装入 `cs` 寄存器，而由 `head.s` 代码文件可知 0x30 恰好为 task1 的 TSS 选择子。

5	SCRN_SEL	=	0x18
6	TSS0_SEL	=	0x20
7	LDT0_SEL	=	0x28
8	TSS1_SEL	=	0x30
9	LDT1_SEL	=	0x38

单步执行后切换到任务 1 跳转到 CS:EIP 0xf:0x10f4 处，由于任务 1 是第一次执行，故直接跳转到了 task1 程序的入口处。



The screenshot shows the Bochs Enhanced Debugger interface. The 'Registers' window displays the state of various registers, including CS (000f), EIP (000110f4), and ESP (000110f4). The 'Memory' window shows the TSS structure at 0x00000bf8, with fields like TSS0\_SEL, LDT0\_SEL, TSS1\_SEL, and LDT1\_SEL. The 'Command Line' window shows the command 'info tss' and the output of the 'info tss' command, which shows the TSS structure for task 1.

此时再次输入 “info tss”，TSS 已经变化为任务切换后 task1 的 TSS，且与寄存器中的值一一对应，这说明任务切换时会根据目标任务的 TSS 的各个字段修改寄存器。



```

tss=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
idt: 0x0038
i/o map: 0x0800

```

3.4 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 0x015c 处设置一个断点，然后运行程序，准备执行 `jmp far 0x20:0` 指令。

The screenshot shows the Bochs Enhanced Debugger interface. The main window displays the state of the system at a specific point in time. The registers window on the left shows the values of various registers, including `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, `esp`, `eip`, `eflags`, `cr0`, `cr2`, `cr3`, `cr4`, and `efer`. The memory window on the right shows the contents of memory at various addresses. The bottom window shows the list of breakpoints, including a breakpoint at address 0x0000015c.

Reg Name	Hex Value	Decimal	Step [s]	Step N [s #]	Refresh	Break [C]
eax	00000001	1	0000015c (7) E...	0000015c (7) E...	00	0
ebx	00000000	0	00000163 (1) 58	00000163 (1) 58	0x7FFFFFFF	0
ecx	00000216	1318	00000164 (1) 1F	00000164 (1) 1F	0x7FFFFFFF	0
edx	00000000	0	00000165 (1) CF	00000165 (1) CF	0x2FFFFF	0
esi	00000000	0	00000166 (1) 1E	00000166 (1) 1E	0x8000	0
edi	00000000	0	00000167 (1) 52	00000167 (1) 52	0x80	3
ebp	00000000	0	00000168 (1) 51	00000168 (1) 51	0x80	3
esp	000010c4	4292	00000169 (1) 53	00000169 (1) 53	0x80	3
eip	0000015c	348	0000016a (1) 50	0000016a (1) 50	0x80	3
eflags	00000046	70	0000016b (5) B...	0000016b (5) B...	0x80	3
cr0	0000	0	00000170 (2) 8...	00000170 (2) 8...	0x80	3
cr2	0010	10	00000172 (5) E...	00000172 (5) E...	0x80	3
cr3	0017	7	00000177 (1) 58	00000177 (1) 58	0x80	3
cr4	0010	10	00000178 (1) 5B	00000178 (1) 5B	0x80	3
efer	0017	7	00000179 (1) 59	00000179 (1) 59	0x80	3

从当前输出窗口可以看到打印了 10 个 B，说明程序执行了 task1。

The screenshot shows the Bochs x86 emulator window. The main window displays the BIOS boot screen, which includes the text "Bochs x86 emulator, http://bochs.sourceforge.net/", "0.6c 08 Apr 2009", "This UGA/VE Bios is released under the GNU LGPL", "Please visit: http://bochs.sourceforge.net, http://www.nongnu.org/ugabios", "Bochs UBE Display Adapter enabled", "Bochs BIOS - build: 02/10/11", "\$Revision: 1.257 \$ \$Date: 2011/01/26 09:52:02 \$", "Options: apmbios pcibios pnphbios eltorito rombios32", "Press F12 for boot menu.", and "Booting from Floppy...". The bottom status bar shows the keyboard layout and the current state of the emulator.

查看此时的 TSS 如下所示，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。

```
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
```

单步执行 `jmp far 0x20:0` 指令，远跳转到 `0x20:0`，将 TSS 选择子 `0x20` 装入 `cs` 寄存器，而 `0x20` 也就是 `task0` 的 TSS 选择子。

由于第一次任务切换时将寄存器现场保存到了 `task0` 的 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`。

The screenshot shows the Bochs Enhanced Debugger interface. The CPU registers window displays the state after a task switch. The `CS` register is set to `0008`, and the `EIP` register is set to `00001150`. The `ESP` register is also set to `00001150`. The memory window shows the TSS structure for task0 at address `0x00000000`. The TSS structure includes fields for `tr`, `ss`, `esp`, `cr3`, `eip`, `eflags`, `cs`, `ds`, `es`, `fs`, `gs`, `gdtr`, `ldtr`, `ldt`, `tr`, `cr0`, `cr2`, `cr3`, `cr4`, `efer`, and `i/o map`.

查看此时的 TSS 如下所示。

```
tr:s=0x20, base=0x00000bf8, valid=1
ss:esp(0): 0x0010:0x0000e60
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x00001150
eflags: 0x00000097
cs: 0x0008 ds: 0x0010 ss: 0x0010
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000001 ebx: 0x00000000 ecx: 0x00000530 edx: 0x0000ef00
esi: 0x00000598 edi: 0x00000998 ebp: 0x00000000 esp: 0x0000e44
ldt: 0x0028
i/o map: 0x0800
```

### 3.5 请详细总结任务切换的过程

#### (1) 准备工作

任务切换通常由时钟中断触发。在 `head.s` 中，时钟中断处理程序 `timer_interrupt` 负责在两个任务之间进行切换。

#### (2) 保存当前任务状态

当时钟中断发生时，当前任务的上下文将被保存：

- **EIP**：当前任务的指令指针，即下一条要执行的指令的地址；

- ESP: 当前任务的堆栈指针;
- EFLAGS: 标志寄存器, 包含处理器的状态标志;
- CS: 代码段寄存器, 包含当前任务的代码段地址。

这些状态被保存在当前任务的堆栈上。

### (3) 切换到新任务

中断处理程序 `timer_interrupt` 会检查当前任务编号, 并决定切换到哪个新任务。它通过以下步骤实现切换:

- 更新任务编号: 更新全局变量 `current`, 该变量记录当前正在执行的任务编号;
- 加载新任务的 TSS: 使用 `ltr` 指令加载新任务的 TSS 选择子, 这会更新任务状态段寄存器, 指向新任务的 TSS;
- 加载新任务的 LDT: 使用 `lldt` 指令加载新任务的 LDT 选择子, 这会更新局部描述符表寄存器, 指向新任务的 LDT。

### (4) 恢复新任务状态

`iret` 指令用于从中断返回, 它从堆栈中恢复新任务的状态:

- EIP: 新任务的指令指针;
- CS: 新任务的代码段;
- EFLAGS: 新任务的标志寄存器;
- ESP: 新任务的堆栈指针。

### (5) 执行新任务

一旦 `iret` 指令执行, CPU 会跳转到新任务的入口点 (`eip` 寄存器指向的地址), 并开始执行新任务的代码。

## 五、实验结果

通过调试 Linux0.00 的 `head.s` 中一个简单的多任务内核实例, 掌握了 Bochs 的简单调试方法和技巧, 掌握了调试系统内核的方法。理解了操作系统及应用程序在内存中是如何进行分配与管理的, 对系统调用中断、时钟中断、任务和模式切换的过程有了更深刻的理解。