调试分析 Linux 0.00 引导程序-实验报告

2021113117 王宇轩

1. 执行 `153` 行 `iret` 时，栈的变化情况

使用 `b 0x17c` 指令在需要调试的 `iret` 处设置断点，继续执行，在执行 `iret` 前，栈中内容如下图所示。

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

此时的栈为任务0执行系统调用 `int 0x80` 切换至的任务0的内核栈，从0x00000e60地址处开始，向栈内压入了为任务0保存的原来的用户栈栈顶地址0x17:0x0BD8、标志寄存器值和中断返回地址0x0F:0x10EB。

单步执行 `iret` 指令，之后的栈如下图所示，按照中断返回地址返回到任务0中执行，栈也切换至任务0的用户栈。

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

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

2. `system_interrupt` 模式切换分析

进入和退出系统调用`system_interrupt`时，都发生了模式切换。其中，**特权级别和栈的切换过程如下**：

在任务通过指令`int 0x80`执行系统调用时，处理器根据向量号`0x80`作为索引，在IDT查对应的门描述符，此处的门描述符是在`head.s`引导部分写入的，此门描述符的DPL为`11`，即特权级别3，符合`int`指令跳转CPL≤DPL的要求。随后，处理器获得中断处理过程的段选择子和偏移量，此时偏移量即为`system_interrupt`的起始地址，段选择子为`0x0008`，对应GDT中内核代码段，如下图，其DPL为0。于是，随着`CS`寄存器的更新，进入`system_interrupt`会发生特权级别切换至内核态。

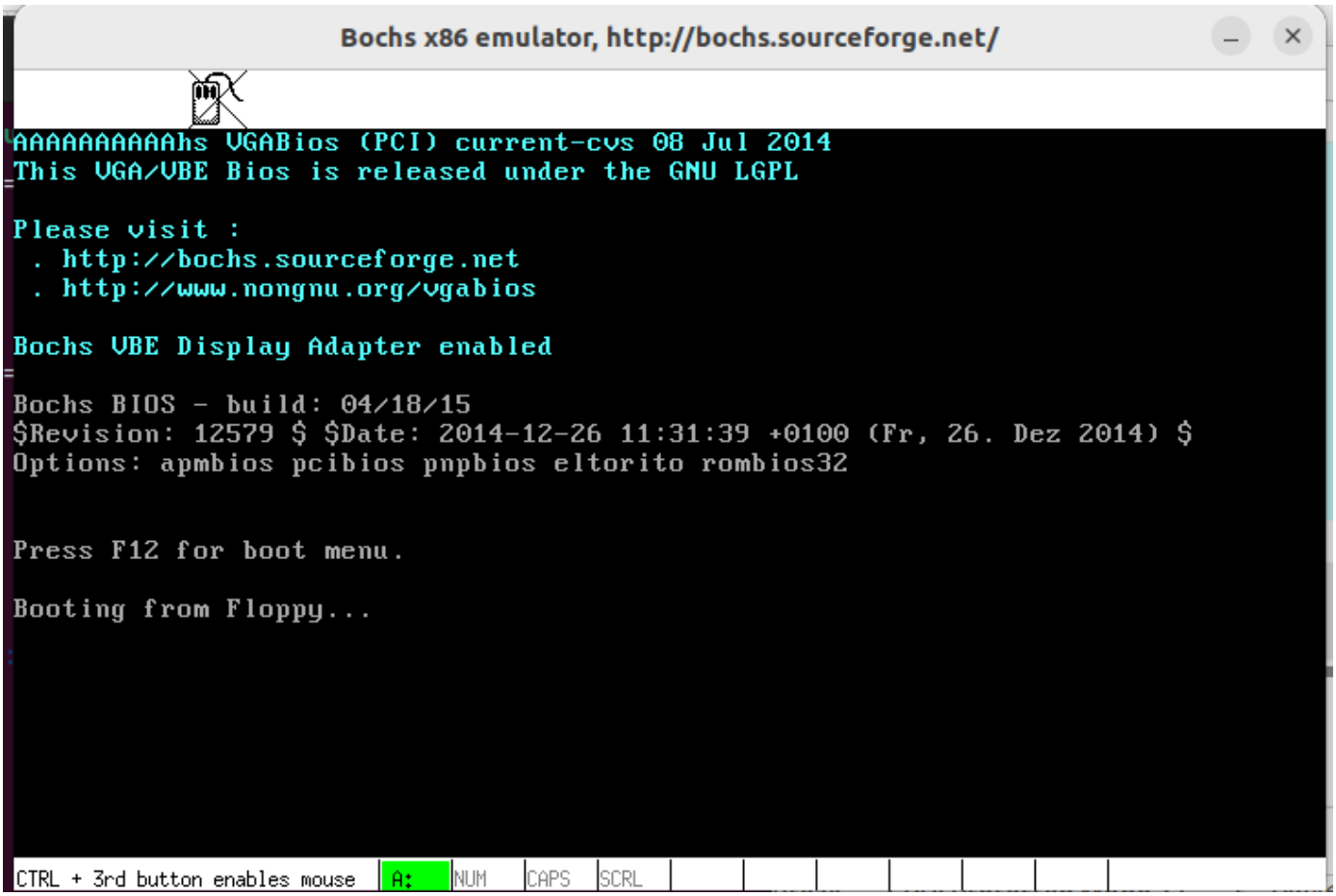
Index	Base Address	Size	DPL	Info
00 (Selector 0x0000)	0x0	0x0	0	Unused
01 (Selector 0x0008)	0x0	0x7FFFFF	0	32-bit code
02 (Selector 0x0010)	0x0	0x7FFFFF	0	32-bit data
03 (Selector 0x0018)	0xB8000	0x2FFF	0	32-bit data
04 (Selector 0x0020)	0xBF8	0x68	3	Available 32bit TSS
05 (Selector 0x0028)	0xBE0	0x40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

由于发生了特权级别切换，也要进行从用户栈至内核栈的堆栈切换。进入系统调用时，发生从用户栈至内核栈的切换，此时：从当前正在执行的任务的TSS中获取处理程序要使用的堆栈的段选择器和堆栈指针。对于任务0来说，即TSS0中的`krn_stk0`。在这个新堆栈上，处理器压入中断过程的堆栈段选择器和堆栈指针。然后，处理器将 EFLAGS、CS和 EIP 寄存器的当前状态保存到新堆栈中。

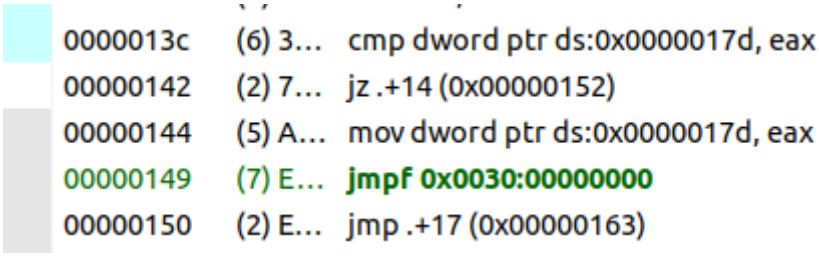
退出系统调用时，特权等级0的代码不能通过跳转进入特权等级3的任务0中继续执行，可通过`iret`指令返回。返回时，处理器从栈中恢复任务0相关的寄存器状态，复原堆栈切换和特权等级。

3. 时钟中断发生时，任务0至任务1的切换过程

用`b 0x12c`在`timer_interrupt`指令处设置断点，运行到此处，可看到任务0已经输出了若干个A，如下图。



在时钟中断程序中，完成了令DS指向数据段，并允许其他硬件中断的工作，此后判断当前执行的是任务0还是任务1，此时执行的是任务.，则把1存入`current`，并通过`jmpl`（图中反汇编结果为`jmpf`）指令执行远跳转，跳转至任务1中执行。



在远跳指令中，提供了跳转目的地的段选择子及偏移量，此处，段选择子是TSS1（GDT中的0x0030选择子条目），而偏移量无用。由于任务1还未被执行过，此次跳转至任务1的起始处代码开始执行。跳转过程如下：

1. 由于任务切换是通过跳转发生的，处理器会进行数据访问权限检查，要求当前（旧）任务的CPL（0）和新任务的段选择子的RPL（0）必须小于或等于被引用的 TSS 描述符或任务门的DPL（3）。
2. .检查新任务的 TSS 描述符是否标记为“存在”（present）并且具有有效界限值（大于等于67H）。GDT中的TSS1描述符如下，其present=1，界限为0x68，是符合的。

```
.word 0x68,tss1,0xe900,0x0    #第7个是TSS1段的描述符。其选择符是0x30
```

3. 检查新任务是否可用、是否忙（通过TSS描述符中的标志位）。
4. 在当前任务的 TSS 中保存当前（旧）任务的状态。处理器在任务寄存器中找到当前TSS的基地址，然后将以下寄存器的状态复制到当前 TSS 中：所有通用寄存器，来自段寄存器们的段选择子们，EFLAGS寄存器的临时保存映像和指令指针寄存器 (EIP)。如，跳转前的tss如下（用指令 `info tss` 查看）：

```
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
```

跳转后，用指令 `x/100 0x0bf8` 查看TSS0的，其中0x0bf8是TSS0的基地址，如下，对照TSS格式，与跳转前相比，可以观察到前文所述的改 变，如EIP寄存器值变为了 `0x00000150`，也就是 `jmpf` 指令执行前EIP的值，可见于上一张图片。

```
[bochs]:
0x00000bf8 <bogus+   0>:  0x00000000  0x00000e60  0x00000010  0x00000000
0x00000c08 <bogus+  16>:  0x00000000  0x00000000  0x00000000  0x00000000
0x00000c18 <bogus+  32>:  0x00000150  0x00000097  0x00000001  0x00000530
0x00000c68 <bogus+ 112>:  0x00000000  0x00000000  0x00000000  0x00000000
0x00000c48 <bogus+  80>:  0x00000010  0x00000010  0x00000000  0x00000000
0x00000c58 <bogus+  96>:  0x00000028  0x08000000  0x00000000  0x00000000
0x00000c68 <bogus+ 112>:  0x00000000  0x00000000  0x00000000  0x00000000
```

5. 在新任务的TSS描述符中设置 B 标志为1，将新任务的 TSS 的段选择子和描述符加载到任务寄存器中，并加载TSS状态。下图是跳转后的寄存器状态，与任务1执行时 `info tss` 的结果是相符合的，如下。

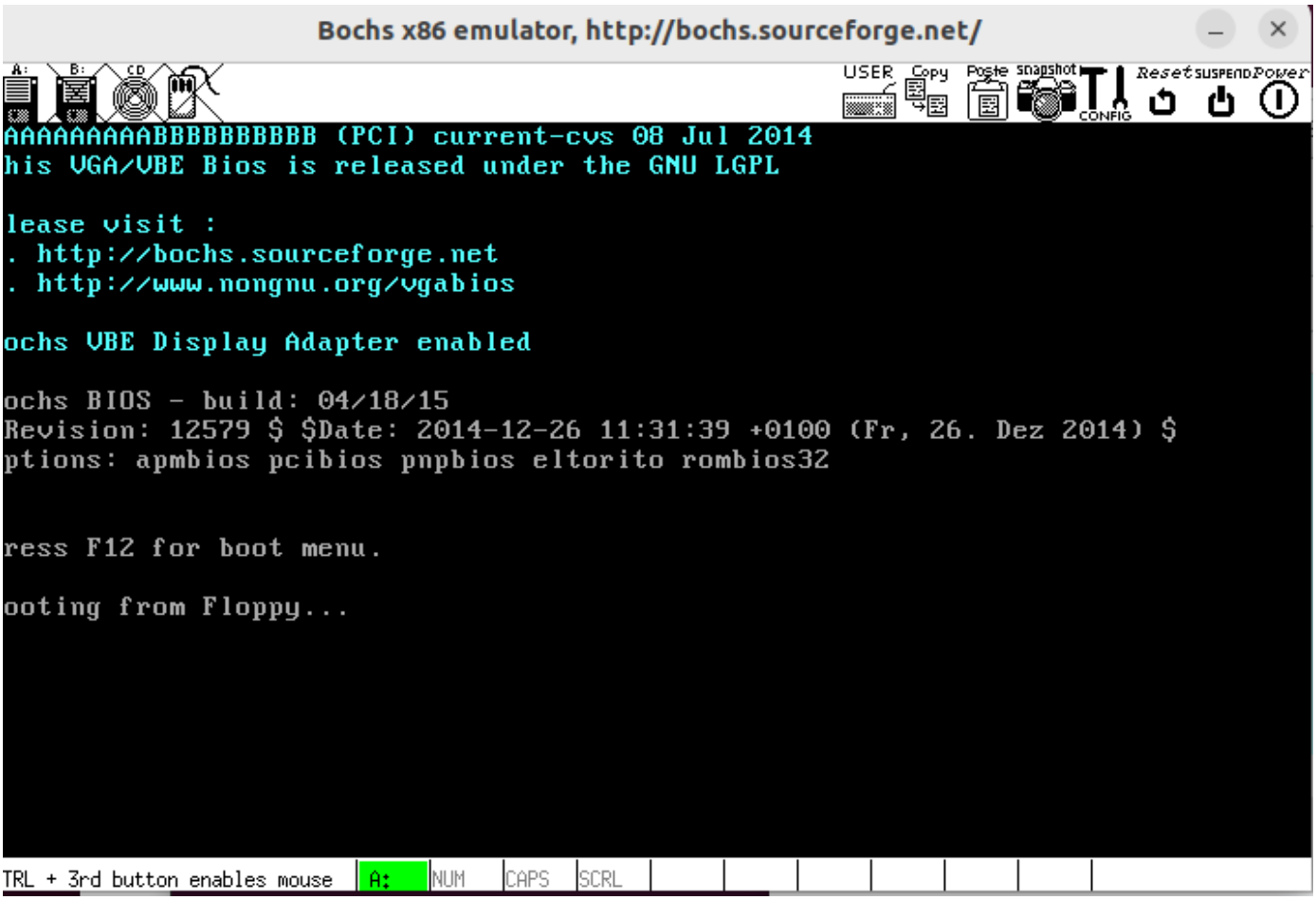
Reg Name	Hex Value	Decimal
eax	00000000	0
ebx	00000000	0
ecx	00000000	0
edx	00000000	0
esi	00000000	0
edi	00000000	0
ebp	00000000	0
esp	00001308	4872
eip	000010f4	4340
eflags	00000202	
cs	000f	
ds	0017	
es	0017	
ss	0017	
fs	0017	
gs	0017	
gdtr	00000998 (3f)	
idtr	00000198 (7ff)	
ldtr	0e60	
tr	0e78	
cr0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

```
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
```

6. 开始执行新任务。

4. 再次切换至任务0的分析

保持断点设置不变，继续运行，又过了10ms，时钟中断发生，执行到`timer_interrupt`指令处暂停，此时任务1也打印出了几个B。



从任务1切换至任务0的整体流程类似于上述从任务0切换至任务1的过程，唯一的不同是，由于任务0此前被执行过，所以这一次任务切换时，处理器会按照第一次任务切换时更新的TSS0来恢复任务0的上下文，包括各个寄存器以及EIP指针指向任务0被中断前即将运行的下一条指令的地址。

被更新过的TSS0如前文所述用指令`x/100 0x0bf8`查看的结果，切换回任务0后，`info tss`查看当前TSS的结果如下，与更新过的TSS0内容一致。

```
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: 0x00000150
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: 0x00000e44
ldt: 0x0028
i/o map: 0x0800
```

寄存器状态如下，也与TSS0中保存的信息一致。

Reg Name	Hex Value	Decimal
eax	00000001	1
ebx	00000000	0
ecx	00000530	1328
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	
efer	00000000	

5. 详细总结任务切换的过程

任务切换时，处理器执行如下操作：

1. 从任务门或（对于由`IRET`指令启动的任务切换：）前任务字段获取新任务的TSS段选择子，作为`JMP`或`CALL`指令的操作数。
2. 检查是否允许当前（旧）任务切换到新任务。数据访问权限规则适用于`JMP`和`CALL`指令。当前（旧）任务的 CPL 和新任务的段选择子的 RPL 必须小于或等于被引用的 TSS 描述符或任务门的 DPL。异常、中断（除由 INT n 指令生成的中断）和`IRET`指令无视目标任务门或 TSS 描述符的 DPL，允许切换任务。对于 INT n 指令产生的中断，需要检查 DPL。
3. 检查新任务的 TSS 描述符是否标记为“存在”（present）并且具有有效界限值（大于等于 67H）。
4. 检查新任务是否可用（调用、跳转、异常或中断）或忙（`IRET`返回）。
5. 检查任务切换中使用的当前（旧）TSS、新 TSS 和所有段描述符是否已分页到系统内存中。
6. 如果任务切换是用`JMP`或`IRET`指令启动的，处理器会清除当前（旧）任务的 TSS 描述符中的 B 标志；如果使用`CALL`指令、异常或中断启动：B 标志保持1。 （见**4. 任务链**中的**表 7-2**）

7. 如果任务切换是用`IRET`指令启动的，处理器会清除 EFLAGS 寄存器临时保存映像中的 NT 标志；如果使用 `CALL` 或 `JMP` 指令、异常或中断启动，则保存的 EFLAGS 映像中的 NT 标志将保持不变。
8. 在当前任务的 TSS 中保存当前（旧）任务的状态。处理器在任务寄存器中找到当前TSS的基地址，然后将以下寄存器的状态复制到当前 TSS 中：所有通用寄存器，来自段寄存器们的段选择子们，EFLAGS寄存器的临时保存映像和指令指针寄存器 (EIP)。
9. 如果任务切换是用`CALL`指令、异常或中断启动的，处理器将在从新任务加载的 EFLAGS 中设置 NT 标志为1。如果用 `IRET` 指令或 `JMP` 指令启动，NT 标志将反映从新任务加载的 EFLAGS 中 NT 的状态（见**表 7-2**）。
10. 如果任务切换是通过 `CALL` 指令、 `JMP` 指令、异常或中断启动的，则处理器会在新任务的 TSS 描述符中设置 B 标志为1；如果使用 `IRET` 指令启动，则 B 标志保持设置状态。
11. 将新任务的 TSS 的段选择子和描述符加载到任务寄存器中。
12. TSS 状态被加载到处理器中。包括 LDTR 寄存器、PDBR（控制寄存器 CR3）、EFLAGS 寄存器、EIP 寄存器、通用寄存器和段选择子们。加载此状态期间的故障可能会破坏体系结构状态。 （如果未启用分页，则会从新任务的 TSS 中读取 PDBR 值，但不会将其加载到 CR3 中。）
13. 加载并限定与段选择子相关的描述符。与此加载和资格相关的任何错误都会发生在新任务的上下文中，并且可能会破坏体系结构状态。
14. 开始执行新任务。 （对于异常处理程序，新任务的第一条指令似乎没有被执行。）