Lab2 调试分析 Linux 0.00 多任务切换

一、实验目的

1. 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
2. 掌握`Bochs`虚拟机的调试技巧
3. 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。

2.0. 实验环境搭建

2.0.1. 虚拟机环境安装

`VMware + Ubuntu`设置

- 下载安装`VMware`虚拟机
- 安装增强功能
- 网络:`NAT`模式
- 共享文件夹 -> 制定一个需要共享的目录
- 安装完系统，进行快照保存

使用`VMware Workstation Pro`，将`ubuntu-22.04.5-desktop-amd64.iso`作为引导盘，创建`ubuntu22.04`虚拟机。

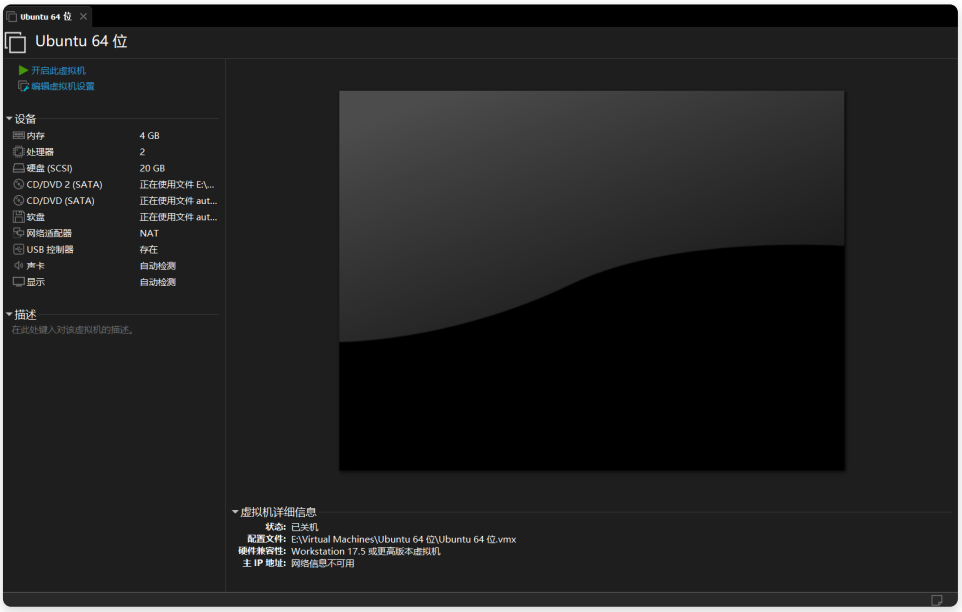

图2-0-1 创建ubuntu虚拟机

2.0.2. Bochs安装及配置

Bochs 是一个能够仿真整个 Intel x86 计算机的程序。 Bochs 能够仿真 386， 486 ，Pentium ， Pentium III ， Pentium 4 乃至 x86-64 CPU 。 Bochs 解释从开机到重启的每一条指令，且支持所有的标准 PC 外设的驱动模型：键盘，鼠标， VGA 显卡/显示器，磁盘，时钟芯片，网卡，无不在其仿真之列。 正因为 Bochs 能够仿真整个 PC 环境， 所以运行在这个仿真环境中的程序会认为其运行在一个真实的计算机上。 因此,很多软件不用修改即可在 Bochs 中运行。

下载 Bochs 的 tarball ，然后自己编译安装。选择 bochs-2.7 版本。

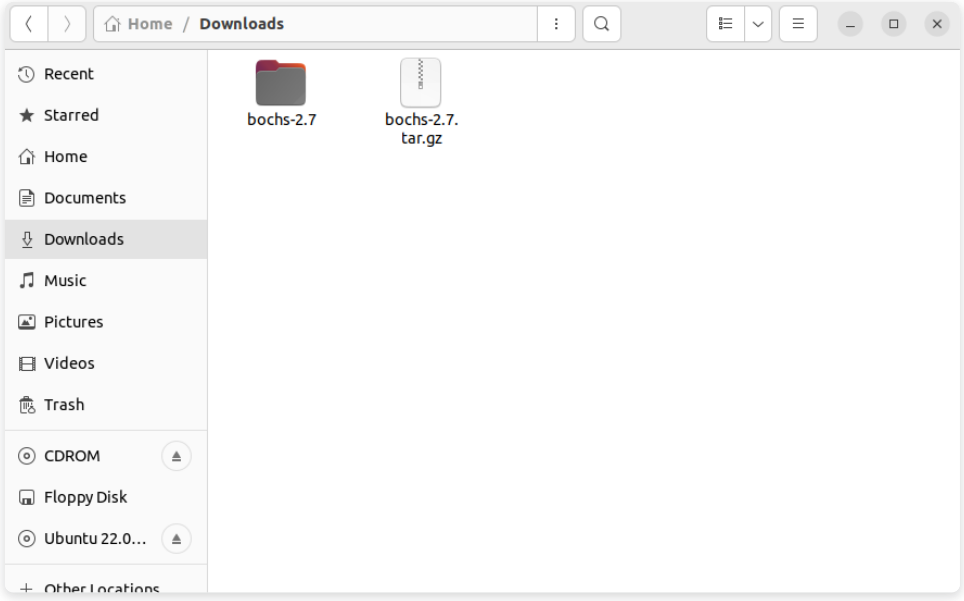

图2-0-2 Bochs-2.7的tarball

安装相关必要的包，避免编译的时候报错。

```
sudo apt-get install build-essential xorg-dev libgtk2.0-dev
```

使用下行命令进行配置。

```
./configure --enable-debugger --enable-disasm
```

编译并安装。

```
# 编译
make
# 安装
sudo make install
```

在 Bochs 的配置文件中加入下面的配置信息，进行可视化配置。

```
display_library: x, options="gui_debug"
```

2.0.3. Linux 0.00

从网站上下载源代码。

```
git clone https://gitee.com/guojunos/Linux000.git
```

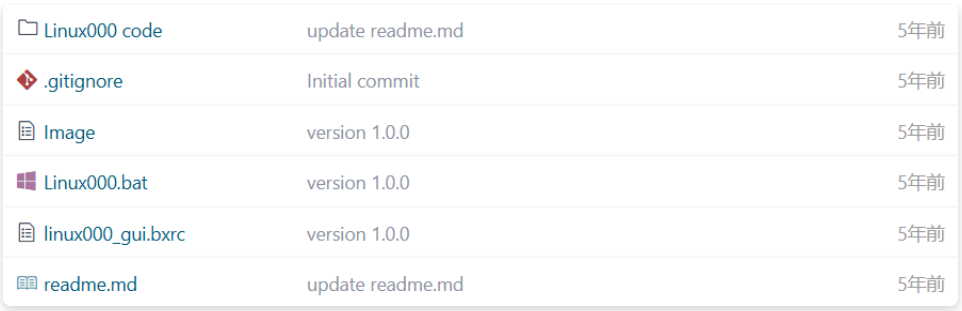

图2-0-3 Linux0.00源代码

其中，`Linux000 code/` 目录下是源码，经修改后可编译成功。`windows` 下可直接运行 `Linux000.bat`。`linux000_gui.bxrc` 是 `Bochs` 配置文件。`Image` 是生成的 `Linux 0.00` 的二进制文件，被加载到软驱，由 `Bochs` 从配置文件读取并启动执行。

找到 `Linux000 code/` 目录下的 `Makefile`，执行。

```
make all
```

运行后的`Linux000 code/` 目录如下显示。

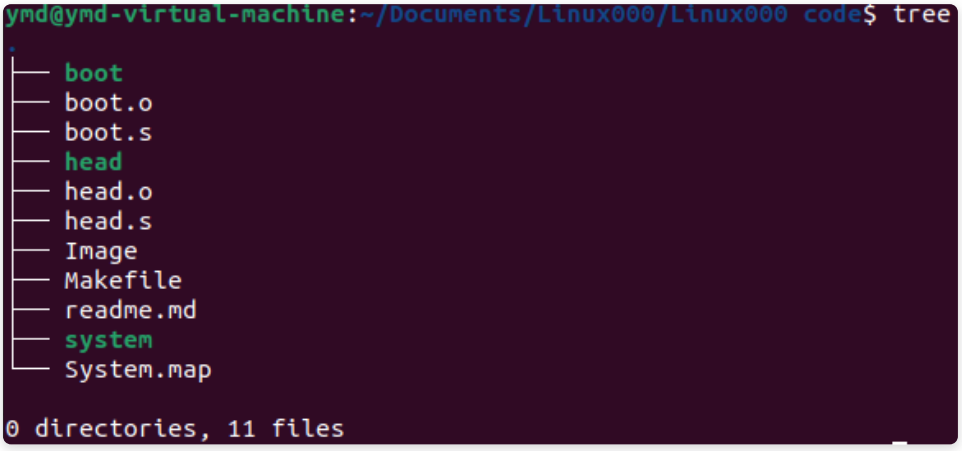

图2-0-4 Linux000 code目录

输入下行命令，即可运行`Bochs`

```
bochs -q -f linux000_gui.bxrc
```

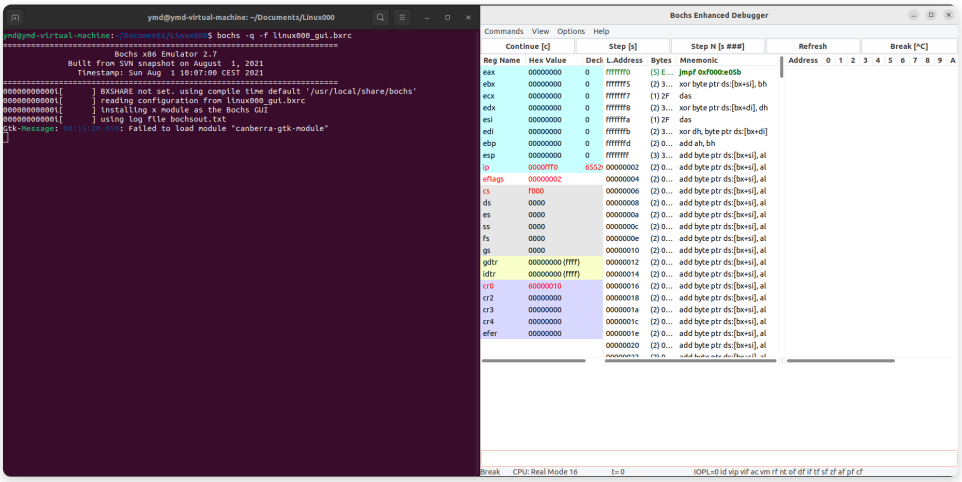

图2-0-5 Bochs运行

2.1. 掌握Bochs虚拟机的调试技巧

2.1.1. 如何单步跟踪

1. 在Bochs界面中选项栏中，点击Step [s]选项

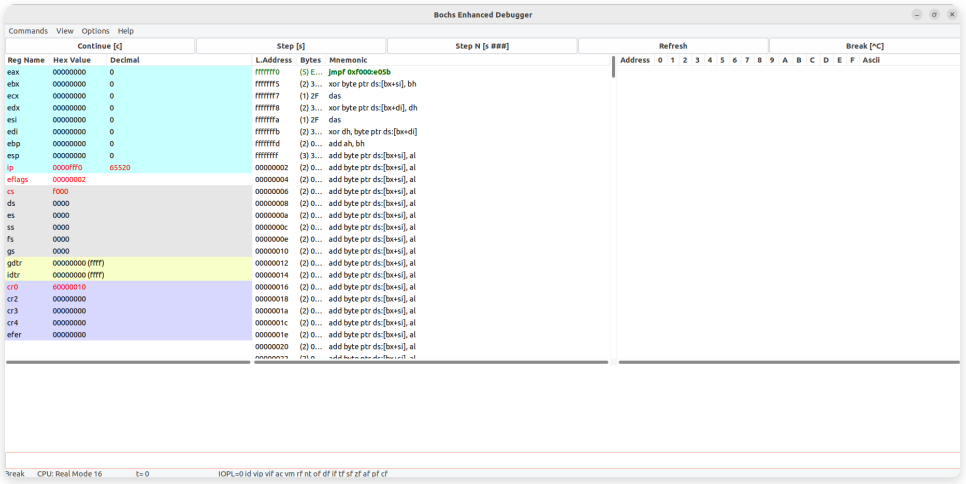

图2-1-1 Bochs界面

2. 在Bochs界面中输入命令S实现单步执行当前指令，然后进入函数调用

2.1.2. 如何设置断点进行调试

1. 设置断点

   输入命令，Bochs在执行到该断点的地址时停止。

```
b <address> #设置内存地址或符号为相应的断点
```

2. 显示断点

   输入命令，显示当前所有断点状态。

```
info break
```

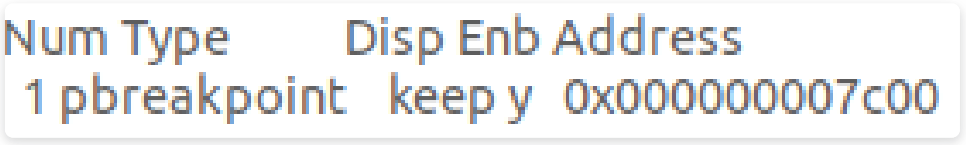

图2-1-2 显示断点

3. 开启断点

```
bpe n
```

4. 关闭断点

```
bpd n
```

5. 删除断点

```
# 第一种
delete n
# 第二种
del n
# 第三种
d n
```

2.1.3. 如何查看通用寄存器的值

使用命令查看通用寄存器的值

```
r <register_name>
```

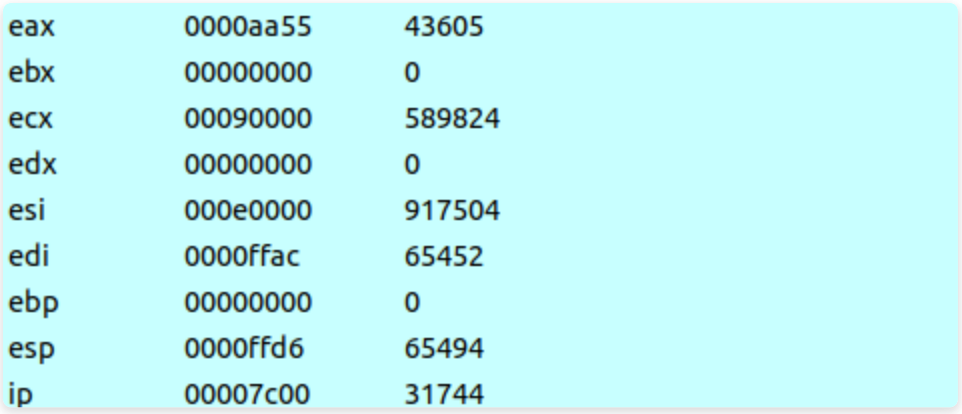

图2-1-3 查看通用寄存器

2.1.4. 如何查看系统寄存器的值

使用命令查看通用寄存器的值

```
info reg <register_name>
```

图2-1-4 查看系统寄存器

2.1.5. 如何查看内存指定位置的值

使用命令查看内存指定位置的值。

1. 查看线性地址

```
x /nuf <address>
# n：指定要显示的内存单元的数量
# u：  显示的内存单元的大小，如下参数之一
#    b 单个字节
#    h 半个字(2 字节)
#    w 一个字(4 字节)
# f：打印的格式如下类型之一：
#    x 按照十六进制形式打印
#    d 按照十进制形式打印
#    u 以无符号的10进制打印
#    o 按照八进制形式打印
#    t 按照二进制行是打印
# <address>：指定要查看的内存地址
```

2. 查看物理地址

```
xp /nuf <address>
# n：指定要显示的内存单元的数量
# u：  显示的内存单元的大小，如下参数之一
#    b 单个字节
#    h 半个字(2 字节)
#    w 一个字(4 字节)
# f：打印的格式如下类型之一：
#    x 按照十六进制形式打印
#    d 按照十进制形式打印
#    u 以无符号的10进制打印
#    o 按照八进制形式打印
#    t 按照二进制行是打印
# <address>：指定要查看的内存地址
```

2.1.6. 如何查看各种表，如`gdt`，`idt`，`ldt`等

1. 查看`gdt`

在`Bochs`的`View`选项中选择`GDT`，即可将其内容显示在可视化界面中。

Index	Base Address	Size	DPL	Info
00 (Selector 0x0000)	0x0	0x0	0	Unused
01 (Selector 0x0008)	0x0	0x0	0	
02 (Selector 0x0010)	0x0	0xFFFFFFFF	0	32-bit code
03 (Selector 0x0018)	0x0	0xFFFFFFFF	0	32-bit data
04 (Selector 0x0020)	0xF0000	0xFFFF	0	16-bit code
05 (Selector 0x0028)	0x0	0xFFFF	0	16-bit data

图2-1-5 查看GDT(方法一)

运行命令查看`GDT`

```
info gdt
```

```
GDT[0x0000]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x0008]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x0010]=Code segment, base=0x00000000, limit=0xffffffff, Execute/Read, Non-Conforming, Accessed, 32-bit
GDT[0x0008]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x0010]=Code segment, base=0x00000000, limit=0xffffffff, Execute/Read, Non-Conforming, Accessed, 32-bit
GDT[0x0018]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x0020]=Code segment, base=0x000f0000, limit=0x0000ffff, Execute/Read, Non-Conforming, Accessed, 16-bit
GDT[0x0028]=Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
```

图2-1-6 查看GDT(方法二)

2. 查看`idt`

在`Bochs`的`View`选项中选择`IDT`，即可将其内容显示在可视化界面中。

Interrupt	L.Address
00	0xF000:0xFF53
01	0xF000:0xFF53
02	0xF000:0xFF53
03	0xF000:0xFF53
04	0xF000:0xFF53
05	0xF000:0xFF53
06	0xF000:0xFF53
07	0xF000:0xFF53
08	0xF000:0xFEA5
09	0xF000:0xE987
0A	0xF000:0xE9E6
0B	0xF000:0xE9E6
0C	0xF000:0xE9E6
0D	0xF000:0xE9E6
0E	0xF000:0xEF57
0F	0xF000:0xE9E6
10	0xC000:0x014A
11	0xF000:0xF84D
12	0xF000:0xF841
13	0xF000:0xE3FE
14	0xF000:0xE739
15	0xF000:0xF859
16	0xF000:0xE82E
17	0xF000:0xEFD2

图2-1-7 查看IDT(方法一)

运行命令查看`IDT`

```
info idt
```

```
IDT[0x00]=Code segment, base=0xf053f000, limit=0x0000ff53, Execute/Read, Conforming, Accessed, 16-bit
IDT[0x01]=Code segment, base=0xf053f000, limit=0x0000ff53, Execute/Read, Conforming, Accessed, 16-bit
IDT[0x02]=Code segment, base=0xf053f000, limit=0x0000ff53, Execute/Read, Conforming, Accessed, 16-bit
IDT[0x03]=Code segment, base=0xf053f000, limit=0x0000ff53, Execute/Read, Conforming, Accessed, 16-bit
IDT[0x04]=32-Bit TSS (Available) at 0xf087f000, length 0x0fea5
IDT[0x05]=32-Bit TSS (Available) at 0xf0e6f000, length 0x0e9e6
IDT[0x06]=32-Bit TSS (Available) at 0xf0e6f000, length 0x0e9e6
IDT[0x07]=32-Bit TSS (Available) at 0xf0e6f000, length 0x0ef57
IDT[0x08]=Code segment, base=0xf04dc000, limit=0x0000014a, Execute-Only, Non-Conforming, 16-bit
```

图2-1-8 查看IDT(方法二)

3. 查看`ldt`

运行命令查看`LDT`

```
info ldt
```

```
LDT[0x1fe8]=??? descriptor hi=0x10000a2a, lo=0x11001100
LDT[0x1fe9]=16-Bit TSS (Busy) at 0x11000050, length 0x30019
LDT[0x1fea]=Data segment, base=0x00f7ff5e, limit=0x000aff00, Read/Write, Expand-down, Accessed
LDT[0x1feb]=??? descriptor hi=0xff820002, lo=0x00000000
LDT[0x1fec]=16-Bit TSS (Busy) at 0xfff20000, length 0xc0206
LDT[0x1fed]=??? descriptor hi=0x00000082, lo=0xf000ef97
LDT[0x1fee]=Data segment, base=0x020076ed, limit=0x00060300, Read-Only
LDT[0x1fef]=??? descriptor hi=0xff880000, lo=0xff9b0040
LDT[0x1ff0]=??? descriptor hi=0x00400094, lo=0x00008683
```

图2-1-9 查看LDT

2.1.7. 如何查看`TSS`

运行命令查看`TSS`

```
info tss
```

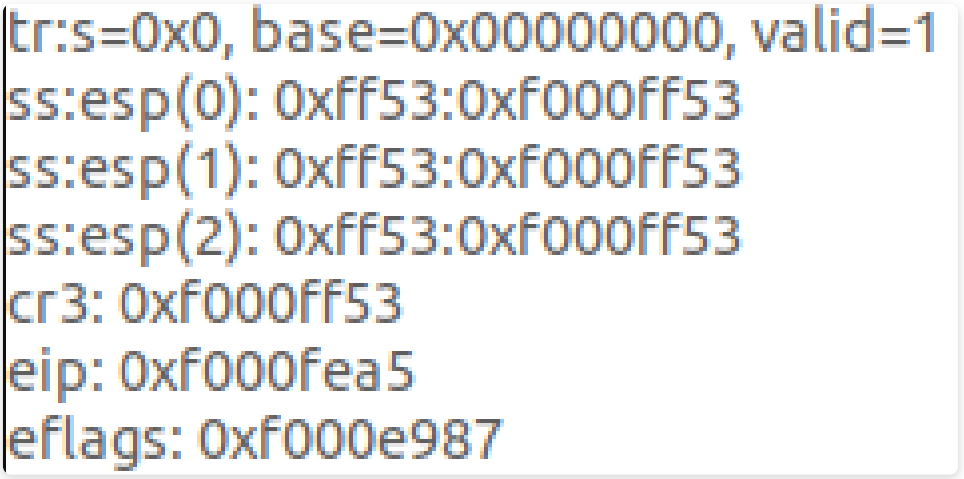

图2-1-10 查看TSS

```
cs: 0xe3fe ds: 0xf859 ss: 0xe739
es: 0xf841 fs: 0xe82e gs: 0xefd2
eax: 0xf000e9e6  ebx: 0xf000e9e6  ecx: 0xf000e9e6  edx: 0xf000e9e6
esi: 0xc000014a  edi: 0xf000f84d  ebp: 0xf000e9e6  esp: 0xf000ef57
ldt: 0x8f51
i/o map: 0xf000
```

图2-1-11 查看TSS(续)

2.1.8. 如何查看栈中的内容

1. 在`Bochs`的`View`选项中选择`STACK`，即可将其内容显示在可视化界面中。

L.Address	Value	(dec.)
0000FFD6	ffff8f51	-28847
0000FFD8	fffff000	-4096
0000FFDA	00000001	1
0000FFDC	00000000	0
0000FFDE	00000000	0
0000FFE0	00000000	0
0000FFE2	00000000	0
0000FFE4	00000000	0
0000FFE6	00000000	0
0000FFE8	00000000	0
0000FFEA	ffffffff	-1
0000FFEC	00000000	0
0000FFEE	00007c00	31744
0000FFF0	00000000	0
0000FFF2	00000000	0
0000FFF4	00000000	0
0000FFF6	ffff9fc0	-24640
0000FFF8	fffffff6	-10
0000FFFA	ffff8f87	-28793
0000FFFC	00000000	0
0000FFFE	00000000	0
00010000	00000000	0
00010002	00000000	0
00010004	00000000	0

图2-1-12 查看STACK(方法一)

2. 运行命令查看栈中的内容

```
print-stack[num]
# 显示堆栈，num默认为 16，表示打印的栈条目数。 输出的栈内容是栈顶在上，低地址在上，高地址在下。这和栈的实际扩展方向相反，这一点请注意。
```

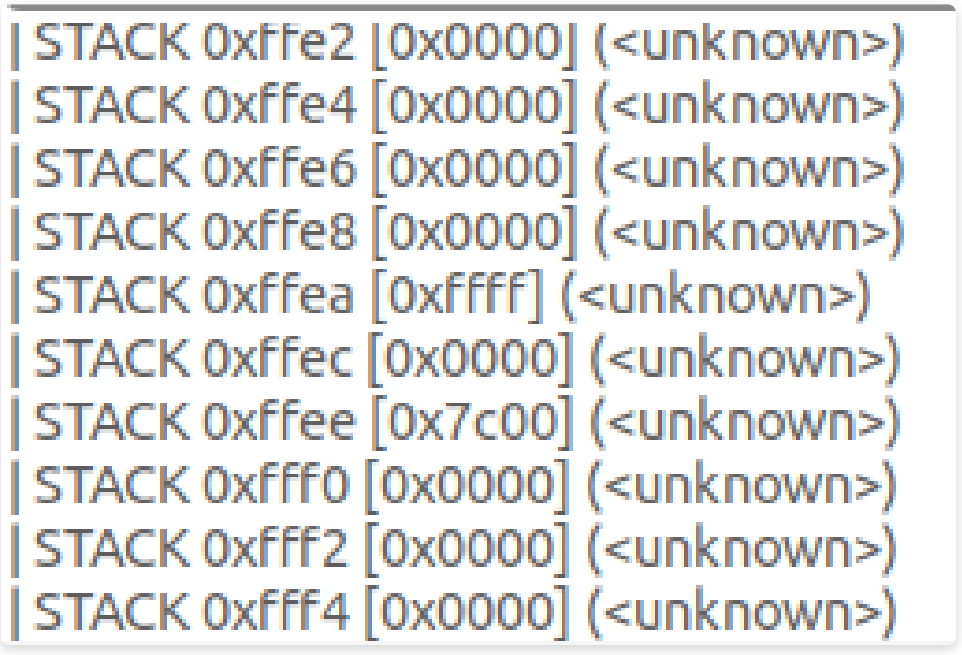

图2-1-13 查看STACK(方法二)

2.1.9. 如何在内存指定地方进行反汇编

1. 运行命令

```
u addr1 addr2
```

在给定的线性地址范围内反汇编指令，包含start处指令，不包含end处指令

2. 运行命令

```
u swith-mode
```

在Intel和 AT&T两种汇编风格之间切换

3. 运行命令

```
u size=n
```

设定反汇编命令的位数，使用0，16，32。值0意思是使用当前的`cs`段寄存器，默认值是0

4. 运行命令

```
set u on
```

每次停止执行时就自动反汇编当前的指令

5. 运行命令

```
set u off
```

每次停止执行时就不自动反汇编当前的指令

三、实验报告

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

查看 `Linux 0.00` 的 `head.s` 汇编代码，发现 `system_interrupt` 函数的完整代码实现如下图所示。

```
153 /* system call handler */
154 .align 2
155 system_interrupt:
156         push %ds
157         pushl %edx
158         pushl %ecx
159         pushl %ebx
160         pushl %eax
161         movl $0x10, %edx
162         mov %dx, %ds
163         call write_char
164         popl %eax
165         popl %ebx
166         popl %ecx
167         popl %edx
168         pop %ds
169         iret
```

图3-1-1 system_interrupt函数

3.1.1. `system_interrupt` 函数分析

对该函数进行分析，发现这是一个中断处理程序的实现，通常用于处理 `int 0x80` 中断，其中

```
.align 2
```

表示将后续代码对齐到4字节边界，`2` 表示对齐到 `2^2=4` 字节边界，可以提高CPU的执行效率；

```
system_interrupt:
```

是一个标签，表示中断处理程序的入口，在程序中，其他部分可以跳转到这个标签来调用该中断处理程序；

```
push %ds
```

表示将当前的数据段寄存器（`%ds`）压入栈中，保存当前的段寄存器值。因为中断发生时，当前的段寄存器可能指向用户态的数据段，而内核需要切换到内核数据段；

```
pushl %edx
push %ecx
pushl %ebx
pushl %eax
```

这几条指令将寄存器 `edx`、`ecx`、`ebx` 和 `eax` 的值压入栈中。中断处理程序需要保存所有使用的寄存器的状态，以便在中断处理完毕后恢复原来状态，避免破坏正在运行的程序的寄存器值；

```
movl $0x10, %edx
```

将常数 `0x10` (即十六进制的16) 移动到 `edx` 寄存器中。这个值是一个段选择符，表示内核数据段。它将用于切换到内核数据段；

```
mov %dx, %ds
```

将 `edx` 寄存器的低16位值 (即 `0x10`) 移动到 `ds` 寄存器中。此时， `ds` 被设置为指向内核数据段，这样接下来就可以访问内核的数据结构；

```
call write_char
```

调用显示字符子程序 `write_char` ， 显示 `AL` 中的字符；

```
popl %eax
popl %ebx
popl %ecx
popl %edx
```

这些指令从栈中恢复寄存器的值，恢复中断前的寄存器状态。这是为了保证中断处理程序执行完毕后，程序能够恢复到中断前的状态；

```
pop %ds
```

从栈中恢复 `ds` 寄存器的值，恢复中断前的数据段寄存器。这样可以恢复用户态程序的数据段；

```
iret
```

`iret` 指令用于从中断返回。在执行过程中，为了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 `EIP` 寄存器的值；为了恢复中断或异常处理程序返回到的代码段，从堆栈中弹出 `CS` 寄存器的值；为了恢复标志寄存器的状态，从堆栈中弹出标志寄存器 `EFLAGS` 的值；如果在中断或异常处理程序执行期间切换了堆栈，为了恢复原始堆栈， `iret` 会从堆栈中弹出新的 `ESP` 寄存器的值；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，实现不同特权级别之间切换。

3.1.2. `iret` 执行前栈分析

在 `0x17b` 设置断点，并运行(`Continue`)至此处后，在点击下一步运行(`Step`)，堆栈状态如下图所示：

1. 下一条指令的内存地址 `EIP` 是 `0x17c`
2. 代码段寄存器 `CS` 是 `0x8` ，所以当前执行的代码段和执行代码的特权级别是0内核态
3. 栈顶指针 `ESP` 是 `0xe4e`
4. 观察 `Stack` 状态栏，可知：
    - 即将弹出的 `EIP` 寄存器的值是 `0x10eb`
    - 即将弹出的 `CS` 寄存器的值 `0x000f`
    - 即将弹出的 `EFLAGS` 寄存器的值 `0x0246`
    - 即将弹出的 `ESP` 寄存器的值 `0x0bd8`

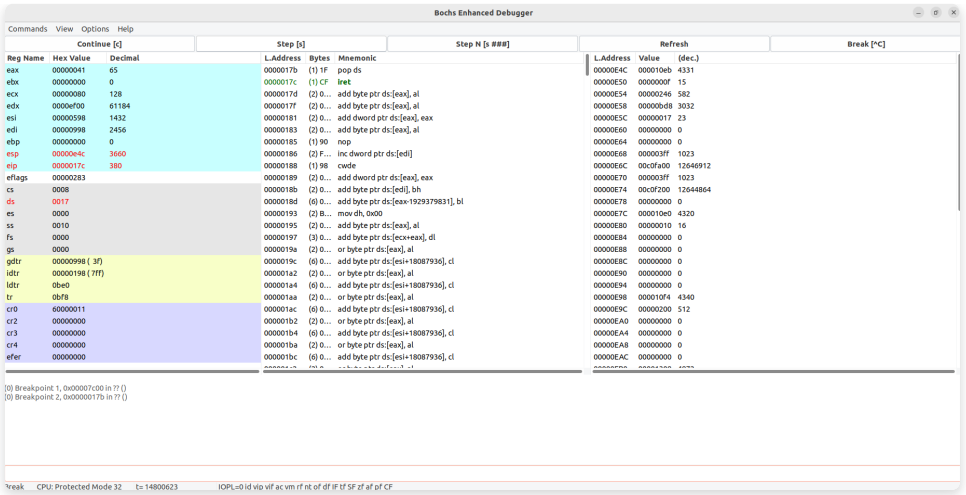

图3-1-2 iret执行前

3.1.3. `iret`执行后栈分析

在`0x17c`处点击下一步运行(`Step`)，即运行`iret`指令，堆栈状态如下图所示：

1. 下一条指令的内存地址`EIP`是`0x10eb`
2. 代码段寄存器`CS`是`0xf`，所以当前执行的代码段和执行代码的特权级别是3内核态
3. 栈顶指针`ESP`是`0xbd8`
4. 观察`Stack`状态栏，可知：
    - 即将弹出的`EIP`寄存器的值是`0x10eb`
    - 即将弹出的`CS`寄存器的值`0x000f`
    - 即将弹出的`EFLAGS`寄存器的值`0x0246`
    - 即将弹出的`ESP`寄存器的值`0x0bd8`

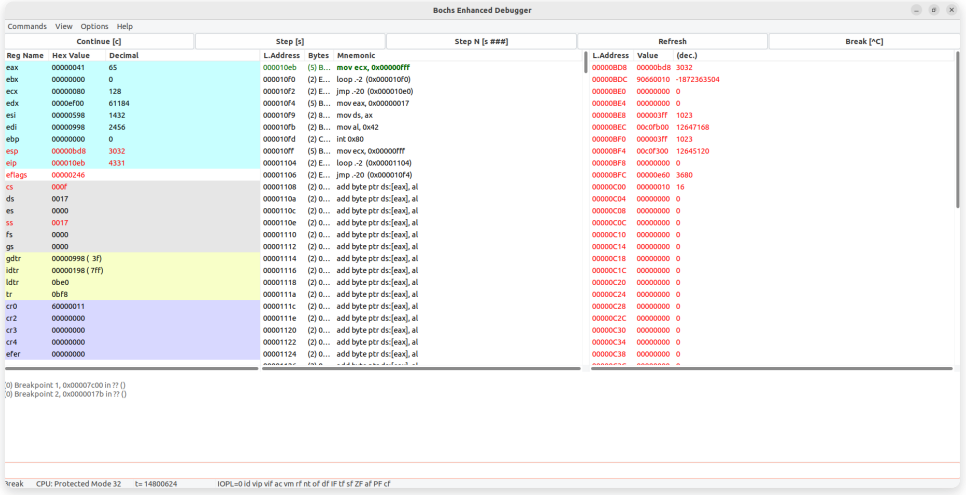

图3-1-3 iret执行后

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?

在进入和退出`system_interrupt`时既会发生发生了特权级的模式切换，也会发生栈切换。

3.2.1. 特权级的改变

当进入`system_interrupt`时，会自动从用户模式切换到内核模式，故程序执行从用户空间切换到内核空间。针对特权级分析，特权级将会从3（用户模式）切换到0（内核模式），并切换到内核栈，`CS`寄存器的值由`0xf`变到`0x8`。

当退出`system_interrupt`时，会自动从内核模式切换回用户模式。针对特权级分析，特权级将会从0（内核模式）恢复到3（用户模式），并在需要时恢复到内核栈，`CS`寄存器的值由`0x8`变到`0xf`，`iret`指令将自动恢复`EIP`、`CS`、`EFLAGS`等寄存器的值，这些寄存器值存储了之前中断前的执行状态。

3.2.2. 栈的改变

当进入`system_interrupt`时，发生堆栈切换，从用户栈切换到内核栈。

当退出`system_interrupt`时，发生堆栈切换，从内核栈切换回用户栈。

3.2.3. 如何进行切换

特权级改变和栈切换是由操作系统内核和处理器硬件共同实现的。在切换过程中，首先要将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；之后操作系统根据切换到的特权级别，选择新的代码段描述符和堆栈描述符，切换堆栈和特权级，并将其加载到相应的寄存器中；最后为了避免破坏用户堆栈，会使用内核堆栈进入内核模式；为了确保程序的正常执行，退出内核态时，特权级和堆栈状态要进行恢复。

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

`timer_interrupt`程序和两个任务的代码和相关执行顺序如下图所示。

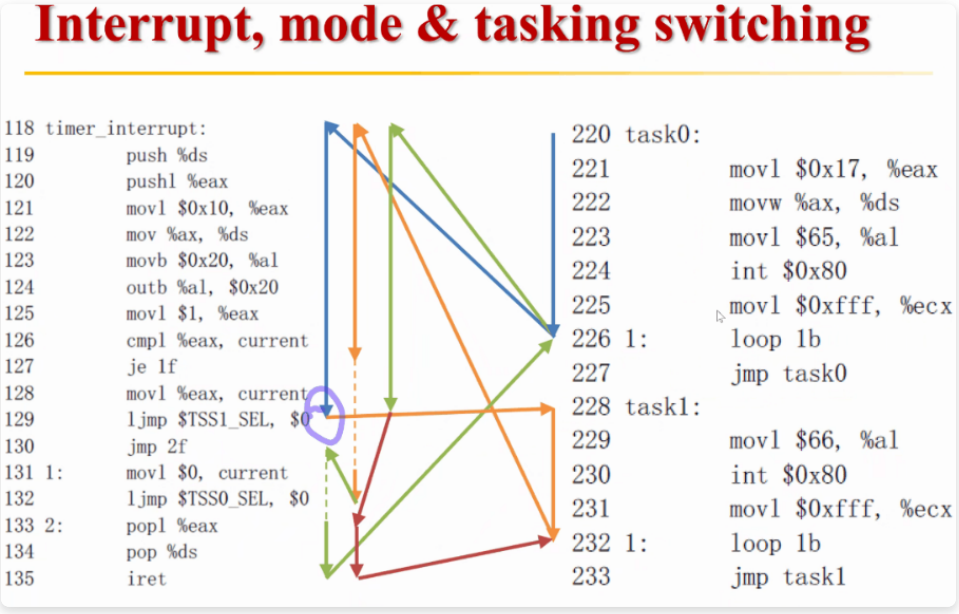

图3-3-1 timer_interrupt程序和两个任务的执行顺序

3.3.1. `timer_interrupt`程序分析

查看`Linux 0.00`的`head.s`汇编代码，发现`time_interrupt`函数的完整代码实现如下图所示。

```
132 /* Timer interrupt handler */
133 .align 2
134 timer_interrupt:
135         push %ds
136         pushl %eax
137         movl $0x10, %eax
138         mov %ax, %ds
139         movb $0x20, %al
140         outb %al, $0x20
141         movl $1, %eax
142         cmpl %eax, current
143         je 1f
144         movl %eax, current
145         ljmp $TSS1_SEL, $0
146         jmp 2f
147 1:      movl $0, current
148         ljmp $TSS0_SEL, $0
149 2:      popl %eax
150         pop %ds
151         iret
```

图3-3-2 timer_interrupt程序

对该函数进行分析，发现这是`Linux 0.00`的定时器中断处理程序的实现，通常用于实现多任务调度和时间片轮转。其中

```
.align 2
```

表示将后续代码对齐到4字节边界，`2`表示对齐到`2^2=4`字节边界，可以提高`CPU`的执行效率；

```
timer_interrupt:
```

是中断处理程序的入口标签。当定时器中断触发时，`CPU`会跳转到这个位置执行代码；

```
pushl %eax
```

保存通用寄存器`eax`的值到栈中，避免破坏中断前程序的寄存器内容；

```
movl $0x10, %eax
```

表示让`DS`指向内核数据段；

```
mov %ax, %ds
```

表示将`eax`的低`16`位（即`0x10`）加载到`ds`，切换到内核数据段。此后可以安全访问内核数据；

```
movb $0x20, %al
```

表示将值`0x20`（中断结束信号）加载到`al`；

```
outb %al, $0x20
```

表示将`al`的值发送到主$8259A$中断控制器的命令端口（端口号`0x20`），通知中断控制器当前中断已处理完成；

```
movl $1, %eax
cmpl %eax, current
je 1f
```

表示将常量 `1` 加载到 `eax`，表示任务 `TSS1` 的标识；比较 `eax` 的值和 `current`（当前任务变量）；如果 `current` 的值等于 `1`（当前任务已经是任务 `TSS1`），跳转到标签 `1`；

```
movl %eax, current
ljmp %TSS1_SEL, $0
```

表示将 `eax` 的值（即 `1`）存储到 `current`，表示切换到任务 `TSS1`；通过长跳转（`ljmp` 指令）切换到任务状态段选择符 `TSS1_SEL` 指定的任务。这会触发任务切换机制，切换到任务 `TSS1`；

```
jmp 2f
```

表示跳过后续代码，直接跳转到标签 `2`；

```
1:  movl $0, current
    ljmp $TSS0_SEL, $0
```

表示将值 `0` 写入 `current`，表示当前任务是任务 `TSS0`；通过长跳转切换到任务状态段选择符 `TSS0_SEL` 指定的任务，切换到任务 `TSS0`；

```
2:  popl %eax
    pop %ds
```

表示从栈中恢复之前保存的 `eax` 值；从栈中恢复之前保存的 `ds` 值，恢复到中断发生前的段寄存器状态；

```
iret
```

表示从中断返回。

3.3.2. `task0` 分析

查看 `Linux 0.00` 的 `head.s` 汇编代码，发现 `task0` 的完整代码实现如下图所示。

```
238 task0:
239         movl $0x17, %eax
240         movw %ax, %ds
241         movb $65, %al              /* print 'A' */
242         int $0x80
243         movl $0xfff, %ecx
244 1:      loop 1b
245         jmp task0
```

图3-3-3 task0

这段代码定义了用户任务 `task0`，其中

```
task0:
```

定义任务 `task0` 的入口标签，表明 `task0` 的起始地址；

```
movl $0x17, %eax
```

表示让DS指向任务的局部数据段；

```
movw %ax, %ds
```

表示将 eax 的低16位（0x17）加载到段寄存器 ds，设置当前任务的数据段为用户数据段；

```
movl $65, %al
```

表示把需要显示的字符A放入AL寄存器中；

```
int $0x80
```

表示执行系统调用，显示字符；

```
movl $0xfff, %ecx
```

表示将常量 0xfff 加载到计数寄存器 ecx，用于延时；

```
1:  loop 1b
```

表示定义循环的标签，执行循环操作，减少 ecx 的值并判断是否为零。如果不为零，跳转到标签 1；否则继续往下执行；

```
jmp task0
```

表示跳转到任务代码开始出继续显示字符。

3.3.3. task1分析

查看Linux 0.00的head.s汇编代码，发现 task0 的完整代码实现如下图所示。

```
247 task1:
248         movl $0x17, %eax
249         movw %ax, %ds
250         movb $66, %al               /* print 'B' */
251         int $0x80
252         movl $0xfff, %ecx
253 1:      loop 1b
254         jmp task1
255
256         .fill 128,4,0
257 usr_stk1:
```

图3-3-4 task1

这段代码定义了用户任务task1，其中

```
task1:
```

定义任务 task1 的入口标签，表明 task1 的起始地址；

```
movl $0x17, %eax
```

表示让`DS`指向任务的局部数据段；

```
movw %ax, %ds
```

表示将`eax`的低`16`位（`0x17`）加载到段寄存器`ds`，设置当前任务的数据段为用户数据段；

```
movl $66, %al
```

表示把需要显示的字符`B`放入`AL`寄存器中；

```
int $0x80
```

表示执行系统调用，显示字符；

```
movl $0xfff, %ecx
```

表示将常量`0xfff`加载到计数寄存器`ecx`，用于延时；

```
1:  loop 1b
```

表示定义循环的标签，执行循环操作，减少`ecx`的值并判断是否为零。如果不为零，跳转到标签`1`；否则继续往下执行；

```
jmp task1
```

表示跳转到任务代码开始出继续显示字符。

```
.fill 128,4,0
```

表示任务1的用户栈空间；

```
usr_stk1:
```

定义标签`usr_stk1`，标记用户任务堆栈的起始位置。它可以被任务描述符或任务切换机制引用。

3.3.4. 切换过程分析

通过反汇编程序窗口发现反向找到时钟中断处理程序的入口，在`0x12c`处设置断点，运行程序观察状态，如下图所示。

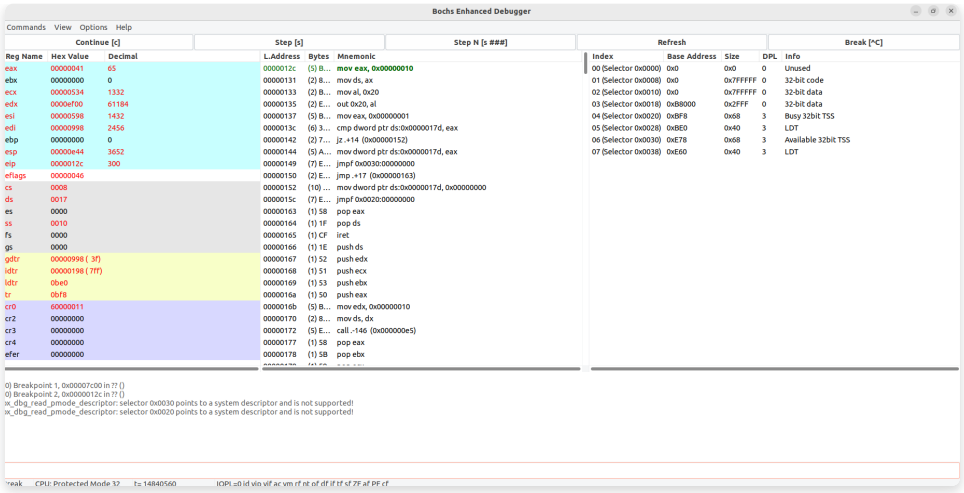

图3-3-5 运行至0x12c

运行结果如下所示，可以发现程序执行`task0`时，打印10个`A`，但是没有`B`被打印出来，所以推理出程序还未执行过`task1`。

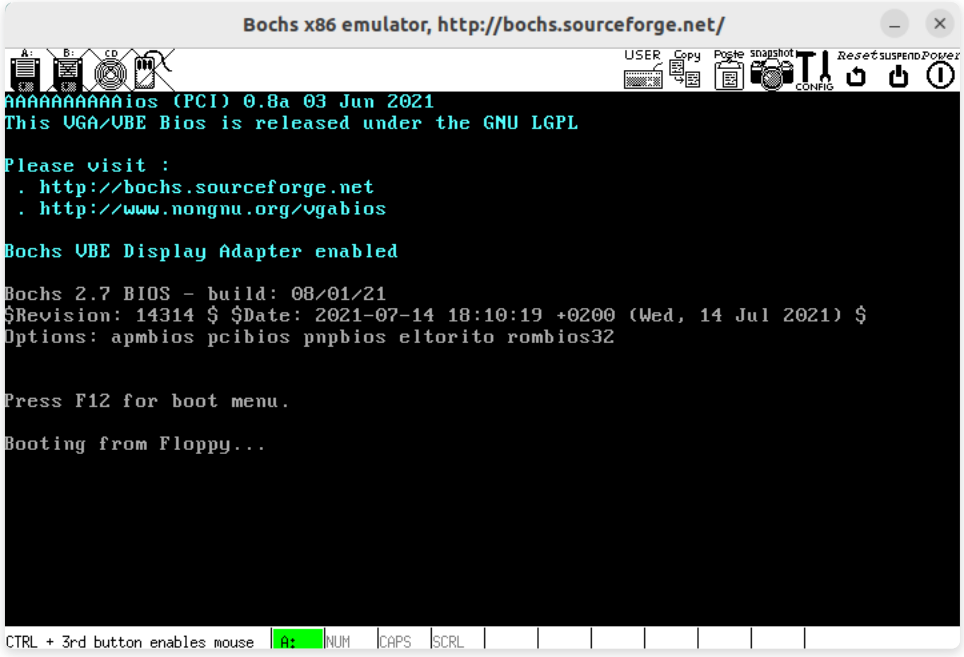

图3-3-6 打印状态

单步运行至`CS:EIP 0x08:0x0149`处准备执行转移指令`jmpf 0x30:0`，并且记录`GDT`。转移指令将一个`TSS`选择子（`0x30`）装入`CS`，实际上将任务切换到这个`TSS`。该指令的选择子是`0x30`，和右边`GDT`对照，`0x30`恰好是一个未执行的`TSS`选择子。

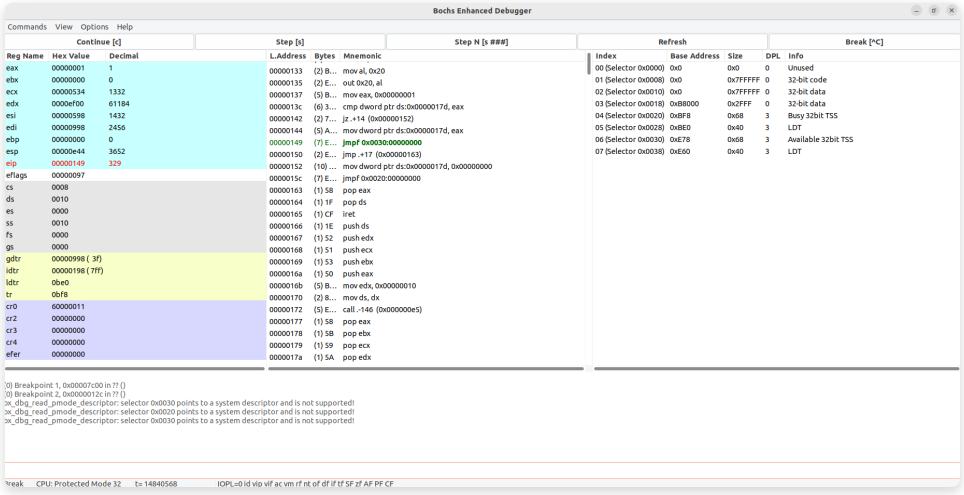

图3-3-7 运行至0x149

在任务切换之前，即未跳转之前，在`bochs`中输入`info tss`，查看`task 0`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
```

图3-3-8 task0的TSS(1)

```
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
```

图3-3-9 task0的TSS(2)

单步执行该指令后，观察程序状态，如下图所示。发现寄存器状态为`CS:EIP 0xf:0x10f4`，表示程序切换到任务1，并且跳转到用户程序`task1`的入口处，表示任务1是第一次执行。

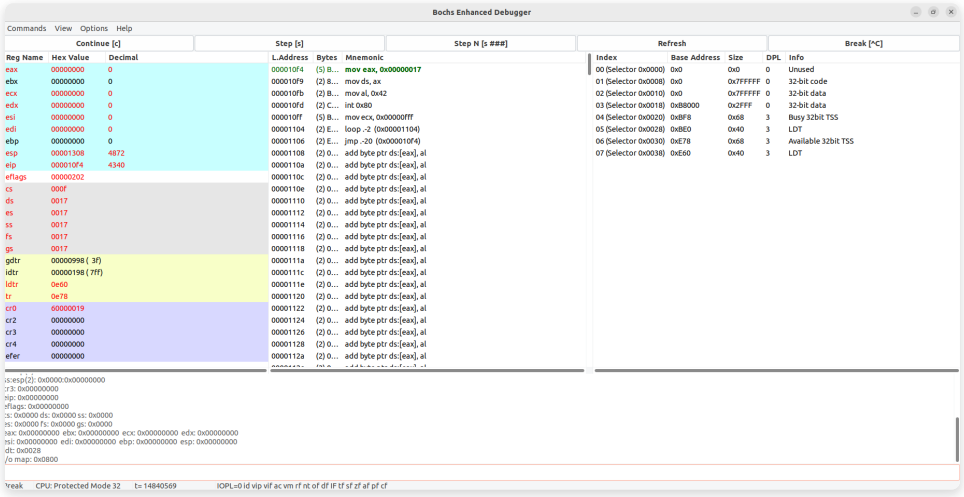

图3-3-10 运行至0x10f4

在任务切换之后，即跳转之后，在`bochs`中输入`info tss`。发现`TTS`发生变化，并且`task 1`的`TSS`与与寄存器中的值相同。证明任务切换的时候会按照`TSS`的各个字段进行寄存器中变量值的修改。

```
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
```

图3-3-11 task1的TSS(1)

```
eflags: 0x00000200
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00001308
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00001308
ldt: 0x0038
i/o map: 0x0800
```

图3-3-12 task1的TSS(2)

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？

通过反汇编程序窗口,在0x15c处设置断点，运行程序观察状态，如下图所示。

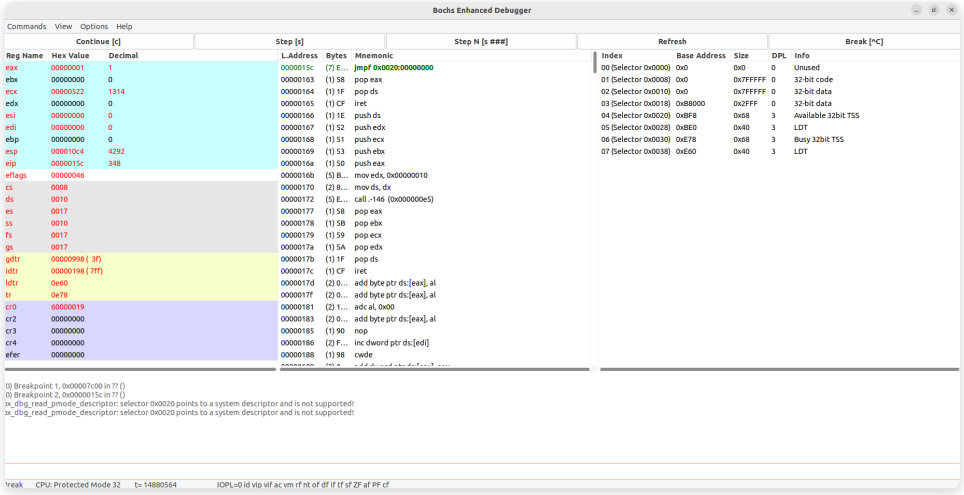

图3-4-1 运行至0x15c

运行结果如下所示，可以发现程序执行时，打印10个A和10个B，所以推理出程序执行了task1。

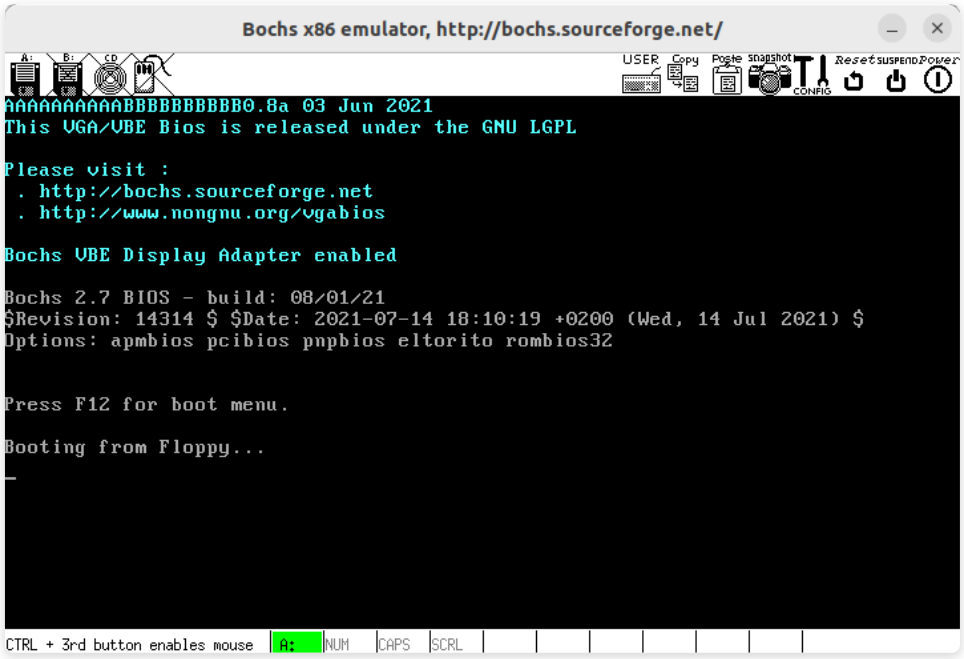

图3-4-2 打印状态

在bochs中输入info tss，观察下图，查看TSS与寄存器的内容相同，证明task的TSS中已经保存了task 1的上下文。

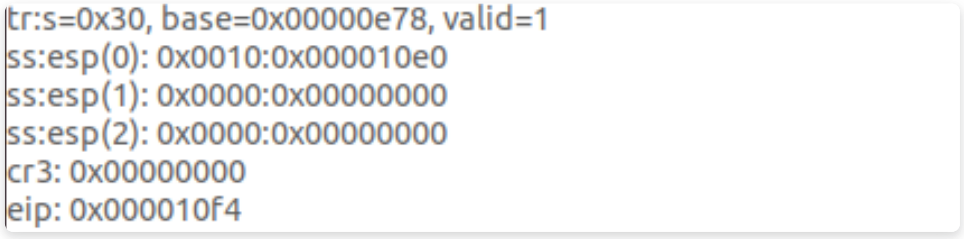

图3-4-3 task1的TSS(1)

```
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
```

图3-4-4 task1的TSS(2)

单步执行 `jmp .+17(0x0163)` ，即之前任务0的`TTS`保存的指令执行地址。状态结果如下所示。发现程序将`task 0`的`TSS`选择子（`0x20`）装入`CS`寄存器中。将`TSS`切换后，`CS:EIP`指向`0x8:0x150`，即第一次任务切换的下一条地址，因为第一次任务切换时将寄存器现场保存到了`task0`的`TSS`中。

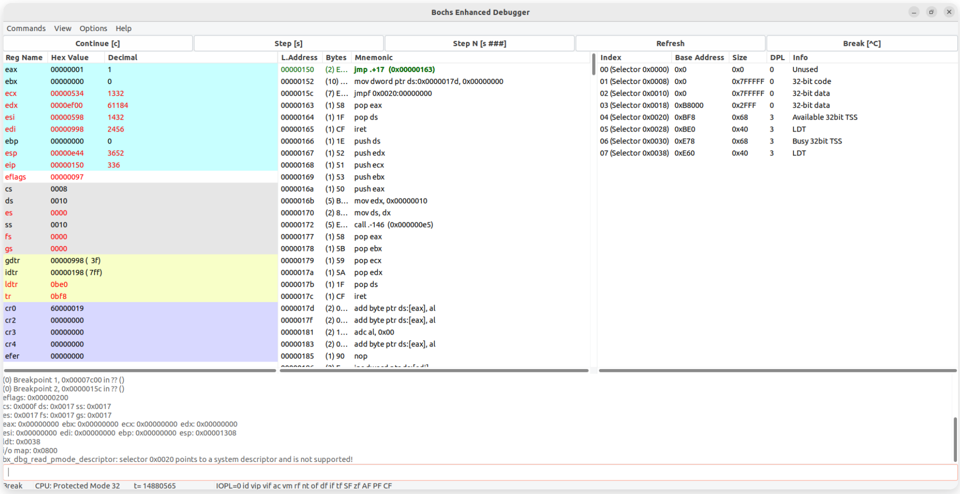

图3-4-5 运行至0x150

在`bochs`中输入`info tss`，观察下图，发现保存着执行前任务1的寄存器值。

```
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
```

图3-4-6 task0的TSS(1)

```
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
```

图3-4-7 task0的TSS(2)

3.5. 请详细总结任务切换的过程

1. 时钟中断触发。中断触发的目的是通知操作系统检查当前任务的运行时间，并决定是否需要切换到其他任务。时钟中断作为触发点，时钟中断由定时器硬件发出，系统会进入中断处理程序`timer_interrupt`。
2. 保存当前任务上下文。中断处理程序首先保存当前任务的运行状态，这一步骤确保当前任务可以在下次运行时继续执行。
    - 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。

3. 选择下一个任务。在保存当前任务上下文后，中断处理程序会根据调度算法选择下一个任务。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。
   - 检查任务优先级、时间片、状态等条件，选择下一个最优的任务。
   - 如果当前任务仍符合运行条件（例如时间片未用尽），可能会选择继续执行当前任务。
   - 调度的结果是确定下一个任务的`TSS`。
4. 加载下一个任务上下文。通过`TSS`切换，将新任务的上下文加载到处理器中。
   - 根据调度结果，加载`taskB`的`TSS`段选择子到任务寄存器。
   - 检查新任务`TSS`的合法性，包括段描述符的权限和段状态。
   - 从`taskB`的`TSS`中加载其上下文信息到处理器，包括：`EIP`、`SS:ESP`、`EFLAGS`、通用寄存器和特权级别信息。
   - 设置任务切换标志，标记当前任务切换完成。
5. 切换堆栈。目的是切换到新任务的堆栈，确保其内核态和用户态的执行环境正确。
   - 如果`taskA`和`taskB`使用不同的内核堆栈，则堆栈指针寄存器（ESP）会更新为`taskB`的堆栈指针。
   - 确保内核栈中已包含中断处理程序的返回地址。
6. 特权级别切换。当新任务（`taskB`）与当前任务（`taskA`）的特权级别不同（如从用户态切换到内核态），完成特权级别切换。
   - 更新段寄存器以反映新的特权级别。
   - 切换到新任务的用户栈或内核栈，确保任务在正确的地址空间中运行。
   - 特权级别切换通常由硬件完成，但操作系统需确保`TSS`中的特权级别信息合法。
7. 恢复并执行新任务。启动新任务的执行，`taskB`的上下文准备就绪并加载到处理器中，`taskB`开始执行。
8. 时钟中断返回。时钟中断处理程序执行完毕后，通过`iret`指令返回到`taskB`的执行点，`taskB`继续执行。

四、总结

通过本次实验，通过完成调试分析`Linux 0.00`多任务切换，我调试一个简单的多任务内核实例，熟练的掌握调试系统内核的方法充分熟悉实验环境；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。在实验过程中，我还不断学习`linux0.00`中`head.s`汇编代码，了解其中的作用和含义。通过本次实验还锻炼了我解决问题的能力。