

练习 2：使用 qemu 执行并调试 lab1 中的软件。（要求在报告中简要写出练习过程）

为了熟悉使用 qemu 和 gdb 进行的调试工作，我们进行如下的小练习：

（1）从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。

（2）在初始化位置 0x7c00 设置实地址断点,测试断点正常。

（3）从 0x7c00 开始跟踪代码运行,将单步跟踪反汇编得到的代码与 bootasm.S 和 bootblock.asm 进行比较。

（4）自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

提示：参考附录“启动后第一条执行的指令”

补充材料：

我们主要通过硬件模拟器 qemu 来进行各种实验。在实验的过程中我们可能会遇上各种各样的问题，调试是必要的。qemu 支持使用 gdb 进行的强大而方便的调试。所以用好 qemu 和 gdb 是完成各种实验的基本要素。

默认的 gdb 需要进行一些额外的配置才进行 qemu 的调试任务。qemu 和 gdb 之间使用网络端口 1234 进行通讯。在打开 qemu 进行模拟之后，执行 gdb 并输入

```
target remote localhost:1234
```

即可连接 qemu，此时 qemu 会进入停止状态，听从 gdb 的命令。

另外，可能需要 qemu 在一开始便进入等待模式，则不再使用 make qemu 开始系统的运行，而使用 make debug 来完成这项工作。这样 qemu 便不会在 gdb 尚未连接的时候擅自运行了。

BIOS 首先运行在 16 位实模式下，第一条指令是 ljmp，执行这条指令后会跳到另外一个地方。gdb 默认是 32 位线性地址模式，调试 BIOS 的 16 位代码（短地址）需要手动计算地址，计算公式如下：

$$\text{Linear Addr} = (\text{cs} \ll 4) + \text{ip}$$

如果 CS=0xf000，EIP=0xe05b，则 Linear Address=0xfe05b。

另外，为了正确反汇编 16 位指令，在 gdb 中执行

```
(gdb)set architecture i8086
```

```
(gdb)x/16i 0xfe05b
```

```
0xfe05b: cmpl $0x0,%cs:-0x2f2c
```

```
0xfe062: jne 0xfc792
```

```
:
```

（1）gdb 的地址断点。

在 gdb 命令行中，使用 b * [地址]便可以在指定内存地址设置断点，当 qemu 中的 cpu 执行到指定地址时，便会将控制权交给 gdb。

（2）关于代码的反汇编

有可能 gdb 无法正确获取当前 qemu 执行的汇编指令，通过如下配置可以在每次 gdb 命令行前强制反汇编当前的指令，在 gdb 命令行或配置文件中添加：

```
define hook-stop
```

```
x/i $pc
```

```
end
```

即可。

(3) gdb 的单步命令。

在 gdb 中, 有 next, nexti, step, stepi 等指令来单步调试程序, 他们功能各不相同, 区别在于单步的“跨度”上。

next: 单步到程序源代码的下一行, 不进入函数。

nexti: 单步一条机器指令, 不进入函数。

step: 单步到下一个不同的源代码行 (包括进入函数)。

stepi: 单步一条机器指令。

答: (1) 首先修改 tools/gdbinit 文件为如下内容:

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
set architecture i8086
target remote :1234
```

这个文件中保存的是 gdb 初始化时执行的指令, 其中第一条指令表示 BIOS 进入 8086 的 16 位实模式方式, 第二条指令的作用是与 qemu 通过 TRP 进行连接, 端口号缺省为 1234。

在 lab1 文件夹内执行 make debug, 会弹出 gdb 调试界面如下图:

```
Terminal
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x00000000 in ?? ()
(gdb) █
```

然后可以在 gdb 调试窗口中使用 si 命令进行单步执行, 同时使用 x /2i \$pc 命令来打印断点处的两条指令, 在这个命令中 x 是显示的意思, i 是指令, /2i 表示两条指令, \$pc 代表 EIP 寄存器, 执行结果如图:

```
Terminal
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x00000000 in ?? ()
(gdb) si
0x0000000b in ?? ()
(gdb) x /2i $pc
=> 0xe05b:      add     %al, (%bx, %si)
    0xe05d:      add     %al, (%bx, %si)
(gdb) █
```

在 gdb 调试界面使用 i r 寄存器可以显示寄存器的值:

```
(gdb) i r eip
eip                0xe05b    0xe05b
(gdb) i r cs
cs                 0xf000    61440
(gdb)
```

从图中可以看出, 打印断点处的指令显示的是 32 位线性地址, 不是实模

式下的真实地址。

(2) 再次修改 tools/gdbinit 文件如下：

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
set architecture i8086
target remote :1234
b *0x7c00
continue
x /2i $pc
set architecture i386
```

其中第三条指令代表在 0x7c00 处设置断点，0x7c00 是 bootloader 被加载入内存的起始地址，第四条指令表示开始执行直到遇到断点，最后一条指令表示调试 CPU 切换到 80386 模式，为启动 ucore 做准备。

执行 make clean 清除上次 make 产生的文件，再次执行 make debug，这一次 gdb 直接停在了 0x7c00 处：

```
Terminal
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000ffff in ?? ()
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:      cli
    0x7c01:      cld
The target architecture is assumed to be i386
(gdb) █
```

程序正常执行到了断点处。

(3) 为了方便跟踪代码的运行，我们对 Makefile 文件进行了修改，将 debug 处的内容修改如下：

```
debug: $(UCOREIMG)
      $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D
$(BINDIR)/q.log -par      allel stdio -hda $< -serial null"
      $(V)sleep 2
      $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
```

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
197 .DEFAULT_GOAL := TARGETS
198
199 .PHONY: qemu qemu-nox debug debug-nox
200 qemu: $(UCOREIMG)
201     $(V)$(QEMU) -parallel stdio -hda $< -serial null
202
203 qemu-nox: $(UCOREIMG)
204     $(V)$(QEMU) -serial mon:stdio -hda $< -nographic
205 TERMINAL :=gnome-terminal
206 debug: $(UCOREIMG)
207     $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel stdio -hda $< -serial null"
208     $(V)sleep 2
209     $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
210
211 debug-nox: $(UCOREIMG)
212     $(V)$(QEMU) -S -s -serial mon:stdio -hda $< -nographic &
213     $(V)sleep 2
214     $(V)$(TERMINAL) -e "gdb -q -x tools/gdbinit"
215
216 .PHONY: grade touch
217
218 GRADE_GDB_IN := .gdb.in
-- INSERT --
```

206,1

85%

主要变化是第 207 行增加了 `-d in_asm -D q.log` 参数，可以保存运行的汇编指令到 `q.log` 文件中，方便后续比较。此次 `make debug` 后得到了 `bin/q.log` 文件，其部分内容如下：

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1

-----
IN:
0x00007c00: cli

-----
IN:
0x00007c00: cli

-----
IN:
0x00007c01: cld
0x00007c02: xor    %ax,%ax
0x00007c04: mov    %ax,%ds
0x00007c06: mov    %ax,%es
0x00007c08: mov    %ax,%ss

-----
IN:
0x00007c0a: in     $0x64,%al

-----
IN:
0x00007c0c: test   $0x2,%al
0x00007c0e: jne    0x7c0a

-----
IN:
0x00007c10: mov    $0xd1,%al
0x00007c12: out    %al,$0x64
0x00007c14: in     $0x64,%al
0x00007c16: test   $0x2,%al
0x00007c18: jne    0x7c14

-----
IN:
0x00007c1a: mov    $0xdf,%al

chy@chy-VirtualBox:~/ucore_lab/code/lab1$
```

图中展示了 0x7c00 后执行的部分指令，我们打开 obj/bootblock.asm 查看代码如下：

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
1
2 obj/bootblock.o:      file format elf32-i386
3
4
5 Disassembly of section .text:
6
7 00007c00 <start>:
8
9 # start address should be 0:7c00, in real mode, the beginning address of the
  running bootloader
10 .globl start
11 start:
12 .code16                # Assemble for 16-bit mode
13     cli                # Disable interrupts
14     7c00:      fa          cli
15     cld                # String operations increment
16     7c01:      fc          cld
17
18     # Set up the important data segment registers (DS, ES, SS).
19     xorw %ax, %ax      # Segment number zero
20     7c02:      31 c0      xor    %eax,%eax
21     movw %ax, %ds      # -> Data Segment
22     7c04:      8e d8      mov    %eax,%ds
23     movw %ax, %es      # -> Extra Segment
24     7c06:      8e c0      mov    %eax,%es
25     movw %ax, %ss      # -> Stack Segment
26     7c08:      8e d0      mov    %eax,%ss
27
28 00007c0a <seta20.1>:
29     # Enable A20:
30     # For backwards compatibility with the earliest PCs, physical
31     # address line 20 is tied low, so that addresses higher than
32     # 1MB wrap around to zero by default. This code undoes this.
33 seta20.1:
34     inb $0x64, %al      # Wait for not busy(8042
                          input buffer empty).
1,0-1 Top
```

然后再打开 boot/bootasm.S 查看代码如下：

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
1 #include <asm.h>
2
3 # Start the CPU: switch to 32-bit protected mode, jump into C.
4 # The BIOS loads this code from the first sector of the hard disk into
5 # memory at physical address 0x7c00 and starts executing in real mode
6 # with %cs=0 %ip=7c00.
7
8 .set PROT_MODE_CSEG,      0x8          # kernel code segment selector
9 .set PROT_MODE_DSEG,      0x10         # kernel data segment selector
10 .set CR0_PE_ON,           0x1          # protected mode enable flag
11
12 # start address should be 0:7c00, in real mode, the beginning address of the
   running bootloader
13 .globl start
14 start:
15 .code16                          # Assemble for 16-bit mode
16     cli                          # Disable interrupts
17     cld                          # String operations increment
18
19     # Set up the important data segment registers (DS, ES, SS).
20     xorw %ax, %ax                # Segment number zero
21     movw %ax, %ds                # -> Data Segment
22     movw %ax, %es                # -> Extra Segment
23     movw %ax, %ss                # -> Stack Segment
24
25     # Enable A20:
26     # For backwards compatibility with the earliest PCs, physical
27     # address line 20 is tied low, so that addresses higher than
28     # 1MB wrap around to zero by default. This code undoes this.
29 seta20.1:
30     inb $0x64, %al               # Wait for not busy(8042
   input buffer empty).
31     testb $0x2, %al
32     jnz seta20.1
```

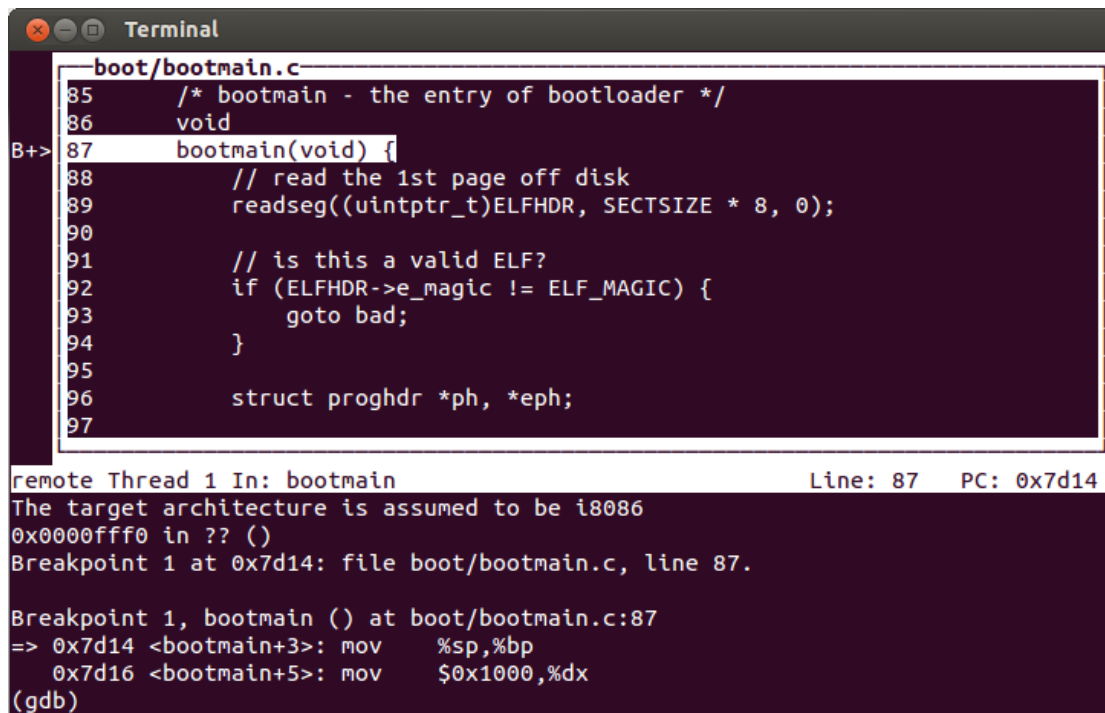
1,1 Top

容易发现，q.log 显示断点之后的代码与 bootasm.S、bootblock.asm 中的代码是一样的。也就是说，bootblock 被安装在了 0x7c00 处的内存空间中。以 Intel 80386 为例，计算机加电后，CPU 从物理地址 0xFFFFF0（由初始化的 CS:EIP 确定，此时 CS 和 IP 的值分别是 0xF000 和 0xFFF0）开始执行。在 0xFFFFF0 这里只是存放了一条跳转指令，通过跳转指令跳到 BIOS 例行程序起始点。BIOS 做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区（即主引导扇区或启动扇区）到内存一个特定的地址 0x7c00 处，然后 CPU 控制权会转移到那个地址继续执行。

（4）要在 0x7d14 处设置断点，修改 tools/gdbinit 文件如下：

```
chy@chy-VirtualBox: ~/ucore_lab/code/lab1
file obj/bootblock.o
set architecture i8086
target remote :1234
break *0x7d14
continue
x /2i $pc
```

执行 make debug 后可以发现，此处是 bootmain 函数的入口地址：



```
boot/bootmain.c
85  /* bootmain - the entry of bootloader */
86  void
B+> 87  bootmain(void) {
88      // read the 1st page off disk
89      readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
90
91      // is this a valid ELF?
92      if (ELFHDR->e_magic != ELF_MAGIC) {
93          goto bad;
94      }
95
96      struct proghdr *ph, *eph;
97
remote Thread 1 In: bootmain                               Line: 87   PC: 0x7d14
The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
Breakpoint 1 at 0x7d14: file boot/bootmain.c, line 87.

Breakpoint 1, bootmain () at boot/bootmain.c:87
=> 0x7d14 <bootmain+3>: mov     %sp,%bp
    0x7d16 <bootmain+5>: mov     $0x1000,%dx
(gdb)
```

参考资料：

[1] 唐源棕. ucore

lab1[OL]. <https://blog.csdn.net/tangyuanzong/article/details/78595854>, 2017-11-21.