# 基础知识

1. AT&T的汇编语法
2. gcc内联汇编
3. 从源文件到可执行文件的编译过程（源代码 – 汇编代码 – 编译生成对象文件（包含可重定位的机器代码） - 链接生成可执行程序）
4. bochs使用和调试

## 工具集说明

bochs 模拟x86系列CPU的虚拟机

cygwin或者linux 提供下面的程序运行的环境，其安装和使用方法见文档 “56 附件1 Cygwin的安装和使用方法.docx“ 以及 ”56 附件 2 如何在windows下使用linux.docx“

**gcc/g++**  将c/c++源代码转换成汇编代码（编译第一步）

**as** 将汇编代码转换成对象文件（编译第二步）

**ld** 将对象文件中的符号链接成具体的地址（编译第三步）

objcopy 将可执行文件在不同的格式间进行转换，比如将windows下的.exe可执行文件转换成bochs下可直接运行的纯二进制代码

objdump 反汇编工具

注1：gcc/g++ ，as，ld，objcopy，objdump的使用方法见附件 “56 附件3 binutils的使用.docx“,其前面部分介绍的基础用法，建议在阅读和实践下面的例程时，同时参照基础用法部分。或者可以先行略过，不理解的地方再看。

注2：下面所有使用的汇编语言都是AT&T格式。

# 使用汇编语言编写bochs的hello world

此部分主要借助一个简单的汇编程序来帮助理解，当前版本的as 也能生成16位汇编代码，而在最初开发linux的时候，as还没有这个功能。通过这个实例，我们将可以以统一的汇编语言来编写16位和32位的代码，而不必像《linux 0.11源代码剖析》中所述的那样必须借助as86这样的程序来撰写linux的16位启动代码。

as的汇编语法和AT&T所定义的汇编语法一致，但针对具体的应用，as还提供了如下伪指令：

.code16 -- 生成 16位的机器代码

.code32 -- 生成32位机器代码

.code16gcc -- .code16的一个特殊模式，当函数调用生成堆栈框架时，以32位的模式压栈和出栈；主要作用是当与c/c++进行内联汇编时，生成统一的堆栈框架。为了将来和c/c++进行内联汇编，下面编写16位程序时使用.code16gcc。

本部分的hello world程序如下：

.code16gcc

START\_SEG = 0x7c0

.text

ljmp $START\_SEG,$NEXT\_ADDR #设置cs:eip指向下一个指令；ljmp的实际作用就是设置cs和eip的值，如果不使用ljmp，cs可能不指向0x7c0这个段

NEXT\_ADDR: #定义一个符号表示当前地址

mov $START\_SEG,%ax

mov %ax,%ds

mov %ax,%ss

mov $1024,%sp #设置ds,ss,esp， 其中esp指向了1024，也就是将堆栈的栈顶设置为0x7c00后面的第二个扇区末尾，这样第二个扇区就是堆栈的空间，而第一个扇区存放代码

mov $0xb800,%ax

mov %ax,%es #设置es = 0xb800,即es指向屏幕上显示字符的区域的段，这个区域中每个显示字符占有两个字节，第一个字节表示要显示的字符，第二个字节表示显示的颜色、背景、前景等

movb $'h',%al

movb %al,%es:0 #显示h

movb $'e',%al

movb %al,%es:2 #显示e

movb $'l',%al

movb %al,%es:4 #显示l

movb $'l',%al

movb %al,%es:6 #显示l

movb $'o',%al

movb %al,%es:8 #显示o

movb $' ',%al

movb %al,%es:10 #显示空格

movb $'w',%al

movb %al,%es:12 #显示w

movb $'o',%al

movb %al,%es:14 #显示o

movb $'r',%al

movb %al,%es:16 #显示r

movb $'l',%al

movb %al,%es:18 #显示l

movb $'d',%al

movb %al,%es:20 #显示d

jmp . #死循环

.org 510

.word 0xaa55 #设置扇区的最后两个字节为0xaa55,表示可以用于启动

上面的代码保存在附件”asm.s”中，将其复制到cygwin的devel目录下，如下：

1.

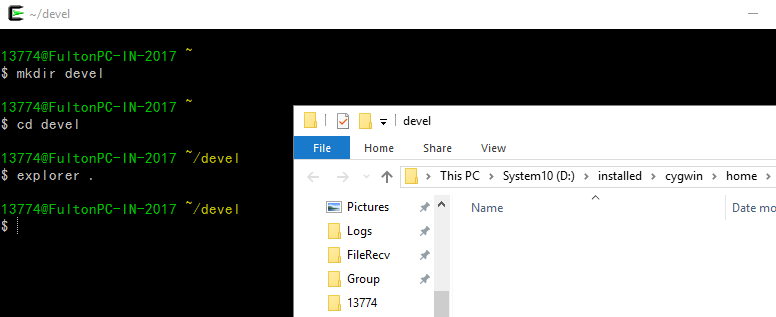
打开cygwin（双击桌面上的cygwin图标）,

创建目录devel ： mkdir devel

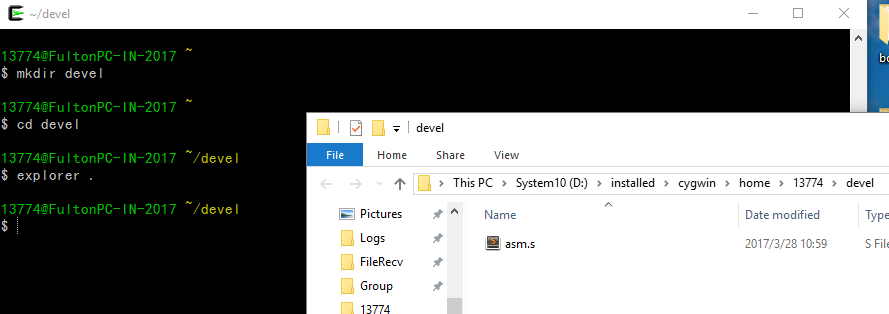
切换到devel目录中： cd devel

使用explorer .打开目录: explorer .

将附件asm.s文件复制到此(鼠标右键复制/剪切)



复制asm.s后



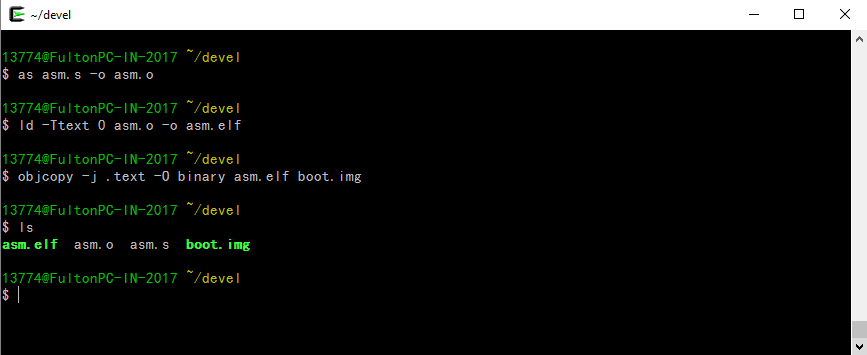
3．现在，将这个汇编文件转换成bochs的启动代码

使用as编译： as asm.s -o asm.o #含义是将asm.s 编译成对象文件asm.o

使用ld链接： ld -Ttext 0 asm.o -o asm.elf #设置.text区的起始地址为0，将asm.o连接成可执行文件asm.elf

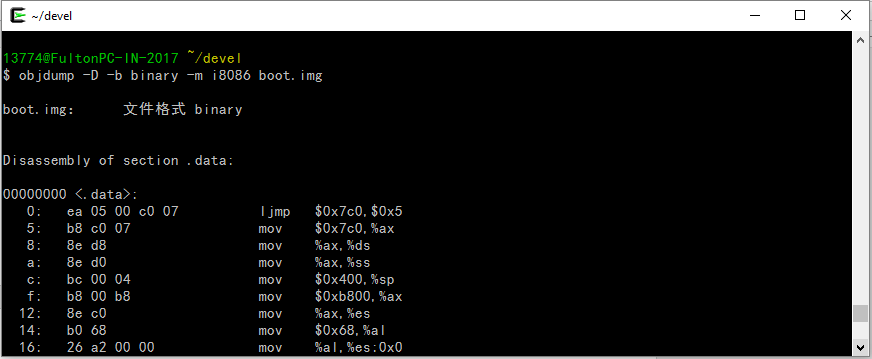
objcopy转换： objcopy -j .text -O binary asm.elf boot.img #从asm.elf文件中仅仅保留.text区，生成二进制的boot.img文件，仅仅保留.text区是因为我们只需要启动代码部分

列出所有的文件： ls



3.现在我们可以使用objdump 来查看boot.img 的反汇编代码：

objdump -D -b binary -m i8086 boot.img #说明：-D 显示所有的节 -b binary输入文件包含纯二进制代码 -m i8086 针对8086，即16位的汇编代码



可以看到，上图中显示的.data是因为这是二进制文件，没有分区，所以反汇编将所有的代码看成是.data区。对于二进制文件来说，区的名称没有意义，因为只有一个区。

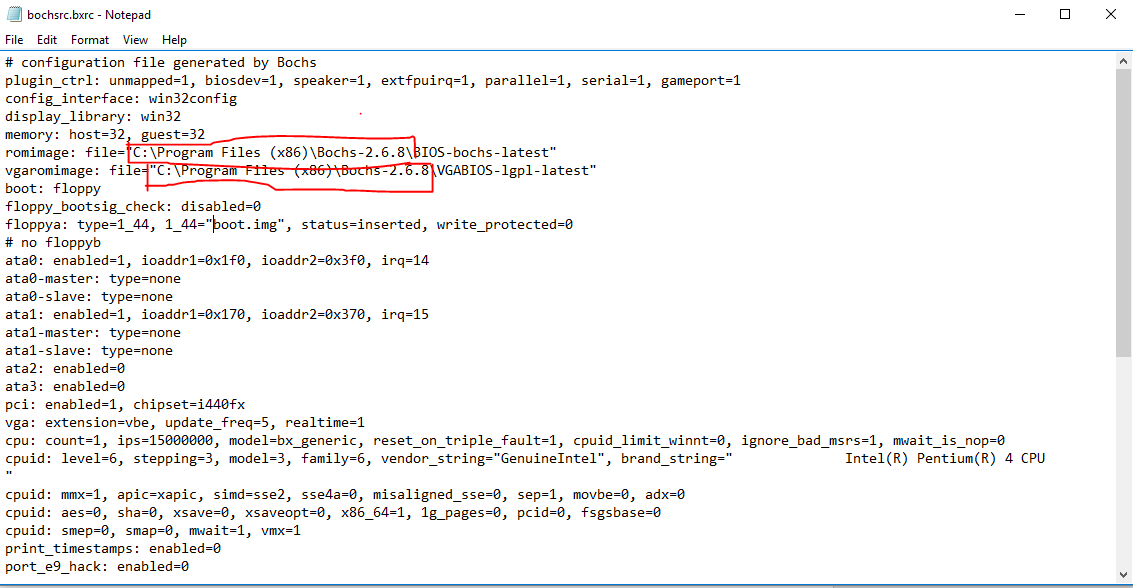
上述反汇编的代码格式为 : 起始地址: 机器代码 汇编代码

我们可以看到上述代码的第一句（0地址处）： ljmp $0x7c0,$0x5 对应的就是原始汇编代码中的第一个ljmp语句，它计算出了NEXT\_ADDR的值为5，恰好指向下一句代码的地址5。

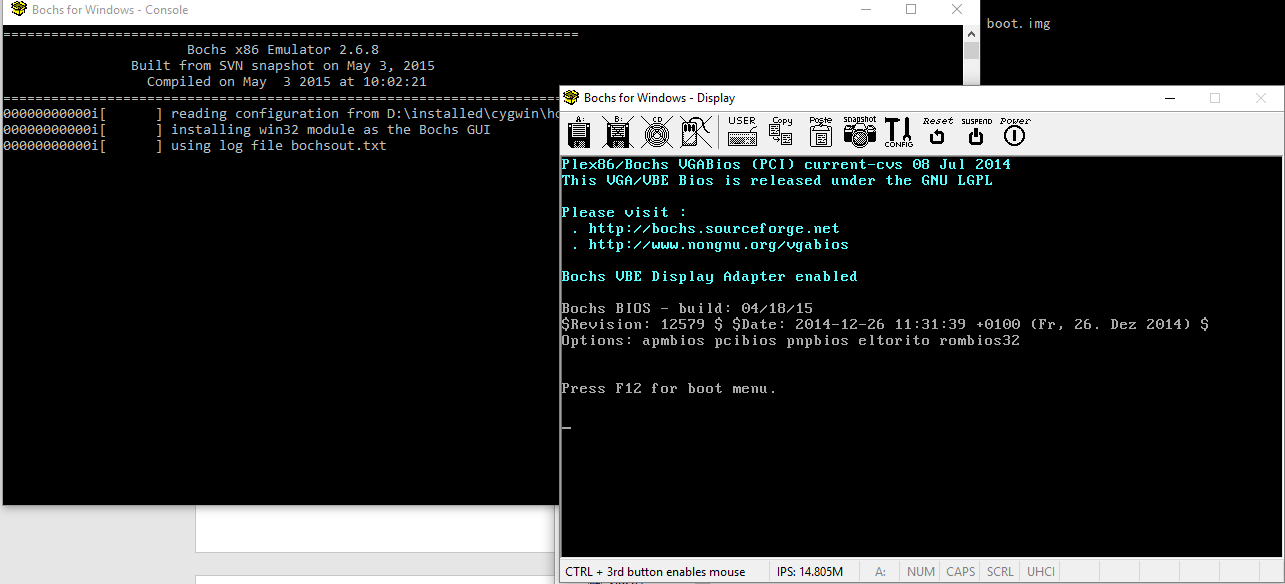
4.现在这个文件已经可以直接在bochs上运行，通过配置bochs的启动文件来启动

首先确保计算机中已经安装了bochs，并找到安装目录，例如 C:\Program Files (x86)\Bochs-2.6.8

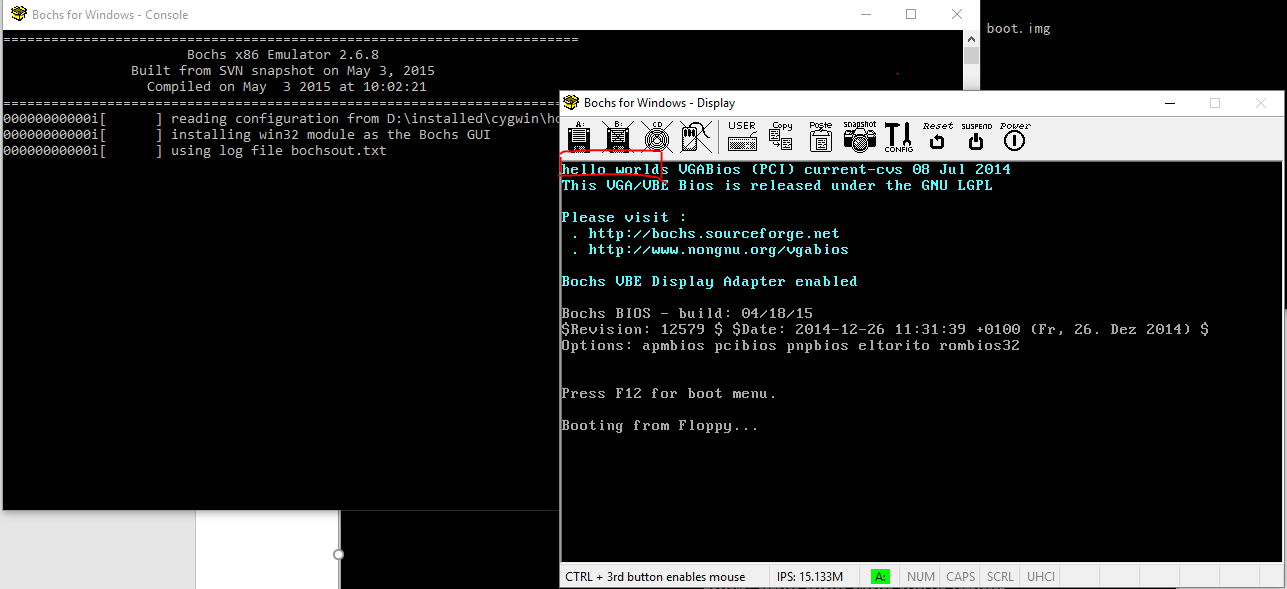
然后将附录中的配置文件”bochsrc.bxrc”复制到devel目录下（使用刚才的方法或者任何其他可行的方法），然后打开修改其中的配置，如下红线所标识：



将图中所标注的两行替换成计算机上的bochs安装目录。保存，然后双击这个配置文件运行，即可看到启动bochs的画面以及运行结果：



下图中的红线部分标识打印出来的hello world（对比上图可知）。



# 使用c语言编写bochs的hello world

经过上面的例子，我们已经知道as有能力编译16位汇编代码。现在，我们将通过c语言来完成一个与上面相当类似的程序。

首先明确一点，我们将要使用的编译器是gcc。各个编译器对内联汇编的支持不同，比如一条mov $0,%eax语句，在gcc中可能如下：

\_\_asm\_\_(“mov $0, %eax \n\t”);

而面向微软的vc编译器可能如下：

\_asm{

mov eax,0;

}

gcc支持两种形式的内联汇编，一种称作基本内联汇编，一种称作扩展内联汇编。

#### 基础内联汇编

格式：

\_\_asm\_\_(

“语句1 \n”

“语句2 \n”

…

);

注意语句中的 换行符不可省略。原因是gcc处理内联汇编时，是将代码直接复制到生成的汇编代码中。

上一节中的hello world程序可以用完全的内联汇编实现（简单地加上双引号和换行,换行后面的\t是为了缩进对齐， 注释照样可以保留）：

//这是一个c语言程序，使用基本内联汇编编写

\_\_asm\_\_(

“.code16gcc \n\t”

“START\_SEG = 0x7c0 \n\t”

“.text \n\t”

“ljmp $START\_SEG,$NEXT\_ADDR #设置cs:eip指向下一跳指令；ljmp的实际作用就是设置cs和eip的值，如果使用ljmp，cs可能不指向0x7c0这个段 \n\t”

“NEXT\_ADDR: #定义一个地址 \n\t”

“mov $START\_SEG,%ax \n\t”

“mov %ax,%ds \n\t”

…

“jmp . #死循环 \n\t”

“.org 510 \n\t“

“.word 0xaa55 #设置扇区的最后两个字节为0xaa55,表示可以用于启动 \n\t”

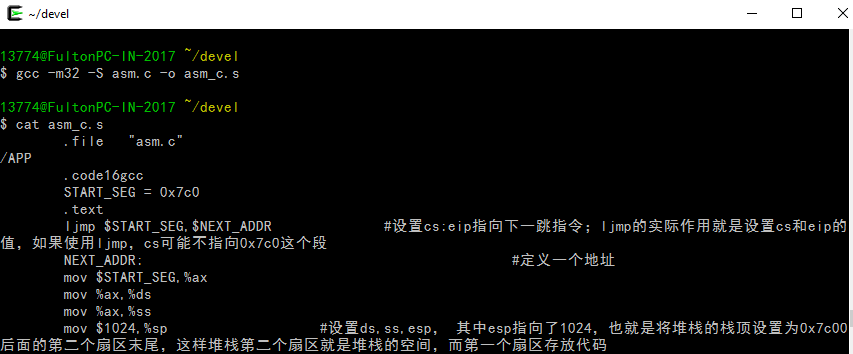
);//内联汇编结束

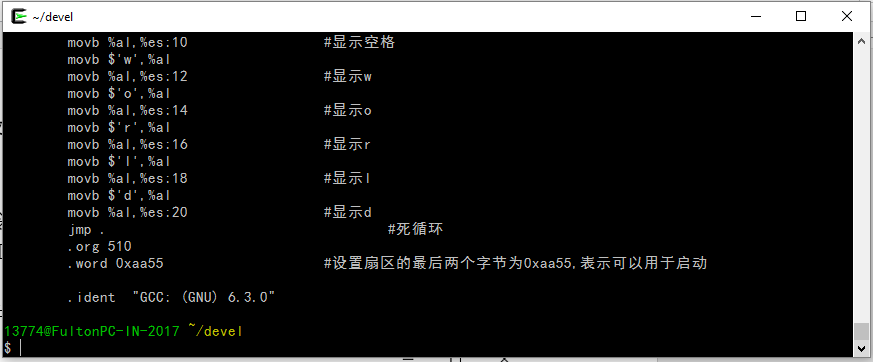
上面的代码在附件asm.c中提供

使用命令gcc -m32 -S asm.c -o asm\_c.s可以将asm.c转换成汇编代码 asm\_c.s，

选项的含义： -S 生成汇编代码 -m32 为16位或者32位机器生成代码。可以使用gcc –help查看帮助；

下面是编译成汇编语言后使用cat查看文件内容；





仔细观察生成的文件的头部，多了两行：

.file “asm.c”

/APP

尾部多了一行：

.ident “GCC: (GNU) 6.3.0”

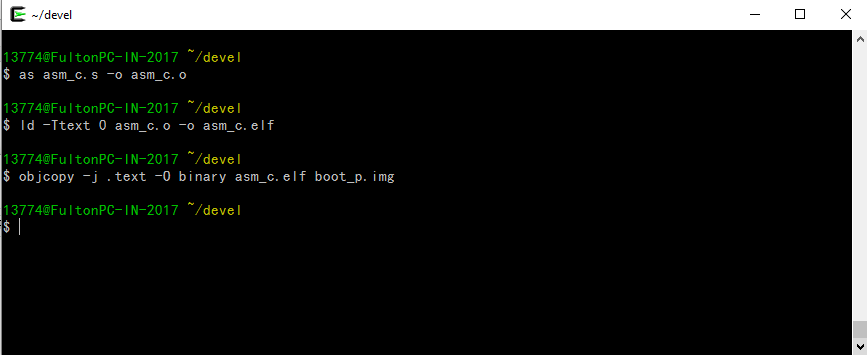
这些信息是用于调试或者其他目的伪指令，对程序主体没有影响。

现在，我们继续采用上面使用的as, ld, objcopy来生成一个bochs可运行的文件boot\_p.img.

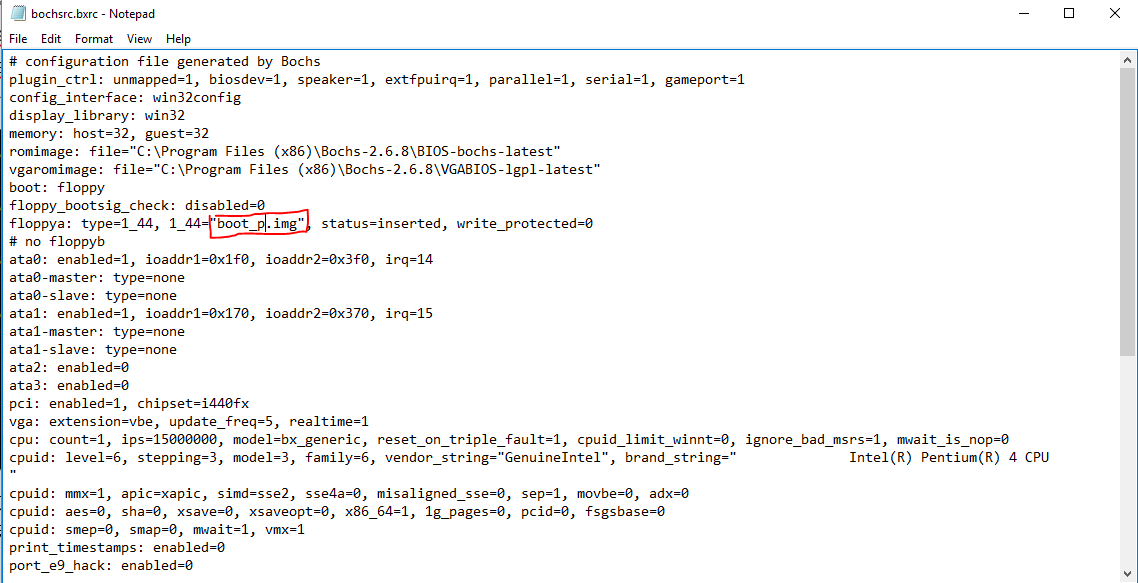
as asm\_c.s -o asm\_c.o

ld -Ttext 0 asm\_c.o -o asm\_c.elf

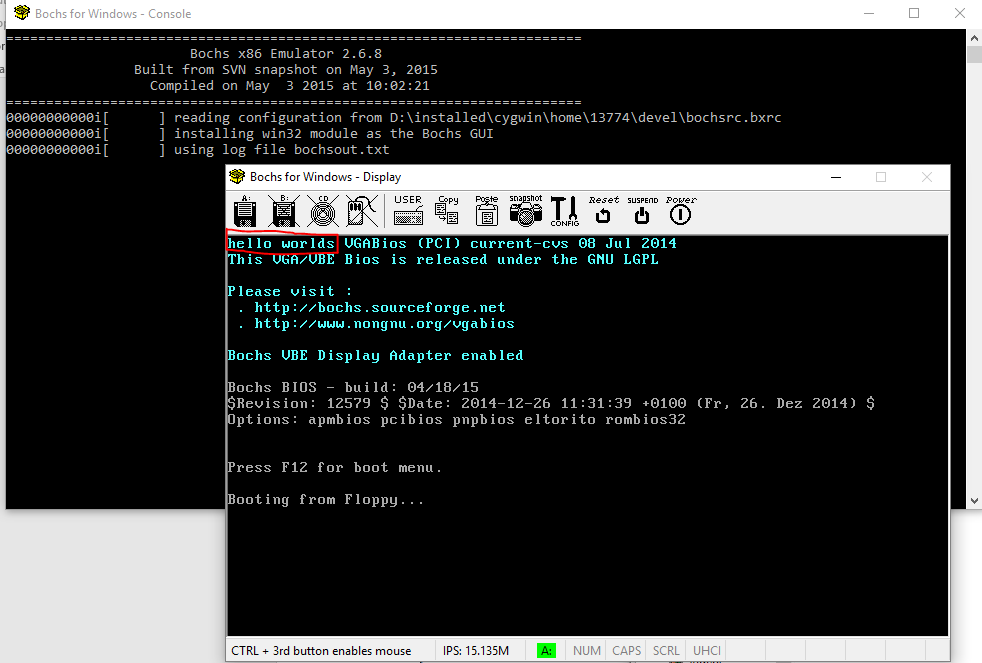
objcopy -j .text -O binary asm\_c.elf boot\_p.img



然后配置bochs.bxrc文件，用“windows记事本“（notepad）程序打开该文件，然后修改下图中标识的内容为boot\_p.img,保存之后启动



可以看到hello world被打印出来



## 扩展内联汇编

扩展内联汇编的基本格式为:

\_\_asm\_\_ \_\_volatile\_\_(

“语句1 \n“

“语句2 \n”

…

“语句n \n”

:输出列表

:输入列表

:修改的寄存器或内存列表

);

下面我们借助扩展内联汇编来优化上面打印hello world部分的程序，我们将其改成循环结构和函数调用来实现，通过这个例子，我们可以看到c语言相对于汇编强大的表达能力和模块化结构。

extern "C" void printStr(const char \*ch); //声明一个打印字符串的函数

extern "C" void printHelloWorld(); //声明打印hello world的程序

\_\_asm\_\_(

".code16gcc \n\t"

"START\_SEG = 0x7c0 \n\t"

".text \n\t"

"ljmp $START\_SEG,$NEXT\_ADDR \n\t"

"NEXT\_ADDR: \n\t"

"mov $START\_SEG,%ax \n\t"

"mov %ax,%ds \n\t"

"mov %ax,%ss \n\t"

"mov $1024,%sp \n\t"

"mov $0xb800,%ax \n\t"

"mov %ax,%es \n\t"

"call \_printHelloWorld \n\t" //调用printHelloWorld函数

"jmp . \n\t"

);

void printHelloWorld()

{

char str[]={"hello world"};

printStr(str);

}

void printStr(const char \*ch)

{

const char \*p=ch;

int i=0;

while(\*p) //如果p!=’\0’则继续打印

{

\_\_asm\_\_ \_\_volatile\_\_(

"mov %%al,%%es:(%%bx) \n\t"

://输出列表为空

:"a"(\*p),"b"(i)

:"memory"

);

p++;

i+=2;

}

}

\_\_asm\_\_(

".org 510 \n\t"

".word 0xaa55 \n\t"

);

现在说明一下程序的改变，首先，在程序头部声明了c语言函数，printStr和printHellWorld, 可以在程序的中间部分看到这两个函数的定义，printHelloWorld调用printStr打印”hello world”，因此打印字符的程序在printStr中。

我们查看printStr的函数体，它总体是一个循环，打印每个字符串直到遇到’\0’字符。在循环内部包含了一个扩展内联汇编，它唯一的一条语句是：

mov %%al, %%es: (%%bx) #在扩展内联汇编的语句中，我们对常规寄存器的访问需要多加一个%，这是因为有些引用形式是%1,%2,…，为了将它们区分开来，所以多使用一个%。

这条语句的含义是：将al的内容送到 es:bx处，那么al, bx的值是多少呢？

往下看第一个冒号，后面是空，表明没有输出；

再往下可以看到第二个冒号之后的内容：

:”a”(\*p),”b”(i)

这一行对应的是扩展内联汇编的输入部分，”a”表示输入到ax寄存器，”b”表示输入到bx寄存器，所以，我们可以知道,ax 是当前要打印的字符，而bx是当前打印字符的偏移。

下面一行：

:”memory”

这一行是修改列表；”memory”告知gcc，内存被修改; 如果我们在语句内部修改了某些通用寄存器，则就要在这里告知gcc，而在输入输出列表中的寄存器不必列在此处。

看完了这两个函数，现在来看一下开始部分的基础内联汇编，它将原来一个字符一个字符搬运到显示区域的所有代码，换成了一个函数调用：

call \_printHelloWorld

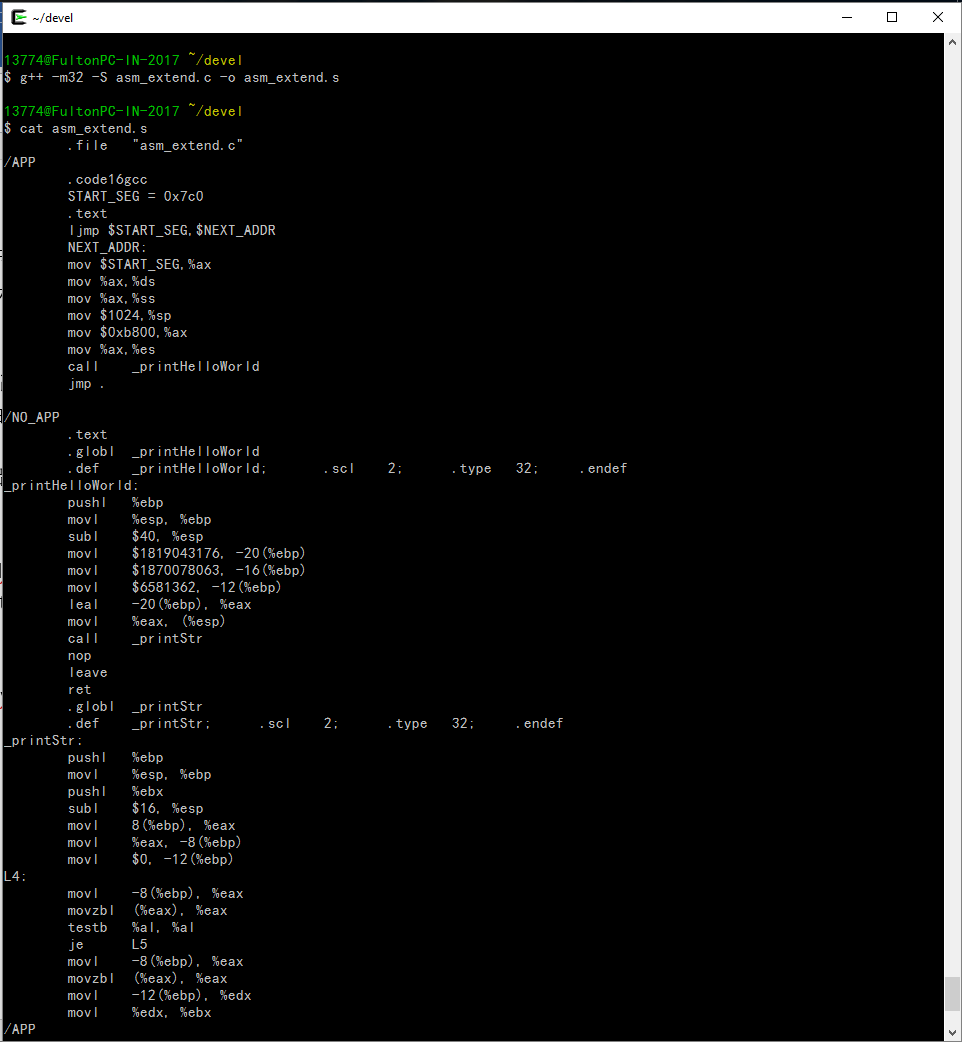
\_printHelloWorld是怎么来的？为什么不是printHelloWorld? 这是gcc做的转换，我们在文件开头声明了extern “C” void printHelloWorld(), 其中的extern “C”前缀表示将这个符号转换成汇编语言时，加上一个下划线(\_)。

上面的程序在asm\_extend.c中提供；将其复制到devel目录。

现在我们将其汇编，看看其汇编生成的程序：

gcc -m32 -S asm\_extend.c -o asm\_extend.s

cat asm\_extend.s





我们可以看到现在生成的两个函数标号：

\_printHelloWorld

\_printStr

我们可以看看\_printStr函数的那一部分:

# 38 "asm\_extend.c" 1

mov %al,%es:(%bx)

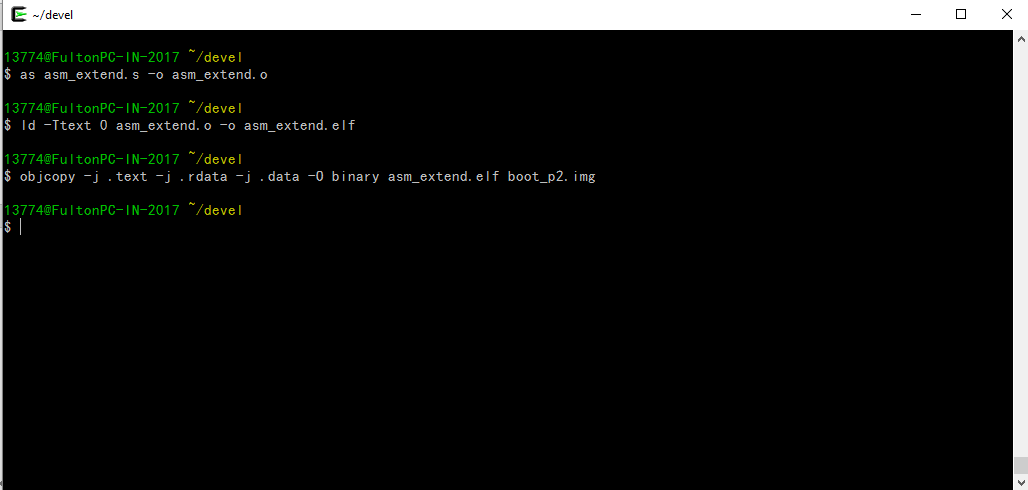
这里对应的就是c语言文件中的内联汇编部分。

现在我们将其编译、链接和转换成二进制文件

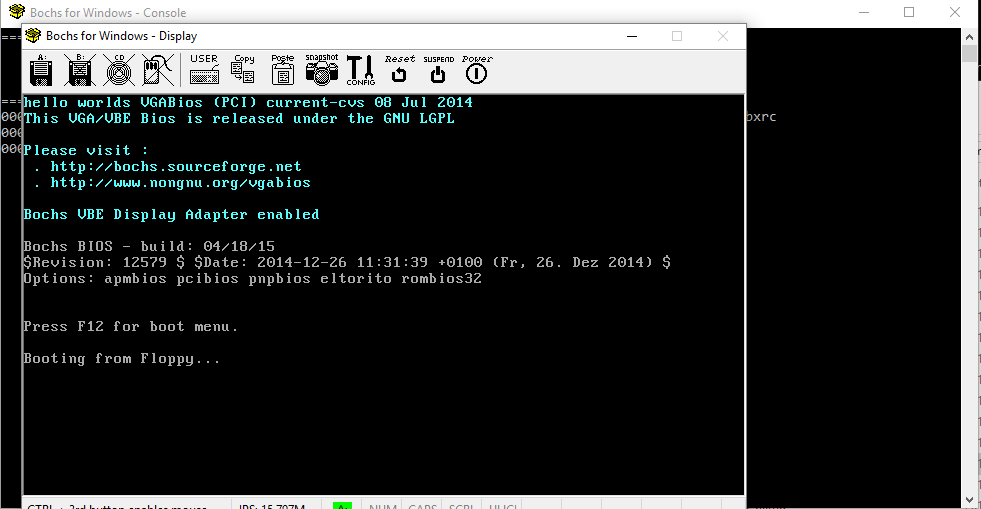
as asm\_extend.s -o asm\_extend.o

ld -Ttext 0 asm\_extend.o -o asm\_extend.elf

objcopy -j .text -j .rdata -j .data -O binary asm\_extend.elf boot\_p2.img



修改bochs.bxrc中的启动文件为boot\_p2.img,双击启动



同样可以看到hello world被打印出来。

# 小结

至此，我们已经基本了解了：

如何使用汇编语言编写bochs启动代码，

如何使用c语言基础内联汇编编写bochs启动代码

如何使用c语言扩展内联汇编编写bochs启动代码

这些方法的关键是了解汇编语法，掌握一定的gcc, as, ld, objcopy的使用方法。

上面的内容也是一个对c语言生成的代码的探索。掌握这些方法，如果有需要，我们可以自由地从c语言转换到汇编语言，编写效率更高的代码或者使用专有指令；或者相反地，从汇编语言到c语言，使用更强更具表达能力的高级语言来高效率地编写代码，然后正确地将其和汇编语言联合起来。