用汇编和 C 编写引导加载程序 - 第 1 部分

阿什奇兰·巴特

2015 年 4 月 20 日 警察

如何使用自己的手写代码在 C 和汇编中启动软盘映像

介绍

我认为这篇文章是关于用 C 和汇编编写引导加载程序的介绍，我不想在编写引导加载程序方面与用 C 和汇编编写的代码进行性能比较。在本文中，我将仅尝试向您介绍如何通过编写自己的代码并注入设备的引导扇区（引导加载程序）来引导软盘映像。在这个过程中，我会将文章分解成不同的部分。很难在一篇文章中解释计算机、可启动设备以及如何编写代码，因此我尽力解释学习计算机的最常见方面以及启动的含义。我试图概括每个阶段的含义和重要性，以便它也易于理解和记忆。如果您需要更详细的解释，

文章的范围是什么?

我将本文的范围限制在如何编写程序代码以及如何将其复制到软盘映像的引导扇区，然后如何使用 x86 测试软盘是否使用您的程序代码启动模拟器，如 Linux 上的 bochs。

我在文章中没有解释的

我没有解释为什么不能用其他语言（如汇编）编写引导加载程序，以及与另一种语言相比，用一种语言编写引导加载程序的缺点。由于这是一篇关于如何编写引导代码的介绍性学习文章，我不想打扰你关于速度、编写更小的代码等更高级的主题。

文章的组织方式

就我而言，我想在文章中介绍一些基础知识，然后用代码进行操作。所以这里是按照我向您介绍如何编写引导加载程序的顺序分解内容。

- 可启动设备简介。
- 开发环境介绍。
- 微处理器简介。
- 在汇编程序中编写代码。
- 在编译器中编写代码。
- 一个显示矩形的迷你项目。

注意:

- 如果您有任何语言的编程经验，这篇文章真的对您有很大帮助。尽管这篇文章似乎相当介绍性，但在启动期间用汇编和 C 编写程序可能是一项艰巨的任务。如果您不熟悉计算机编程，那么我建议您阅读一些介绍编程和计算机基础知识的教程，然后再回到本文。

- 在整篇文章中，我将通过问答的方式向您介绍与计算机相关的各种术语。坦率地说，我会写这篇文章，就像我在向自己介绍这篇文章一样。进行了如此多的问答式对话，以确保我了解它在我日常生活中的重要性和目的。示例：您所说的计算机是什么意思？或者我为什么需要它们，因为我比它们聪明得多？

那么，让我们开始吧.....:)

可启动设备简介

当一台典型的计算机开机时会发生什么？

通常，当计算机打开时，电源按钮会向电源发出信号，以向计算机和其他组件（例如 CPU、显示器、键盘、鼠标）发送适当的电压。CPU 初始化基本输入输出系统只读存储器芯片以加载可执行程序。一旦 BIOS 芯片被初始化，它就会将一个特殊的程序传递给 CPU 来执行，称为 BIOS，下面是它的功能。

- BIOS 是嵌入在 BIOS 芯片中的特殊程序。
- 执行 BIOS 程序，然后执行以下任务。
- 运行开机自检。
- 检查可用的时钟和各种总线。
- 检查 CMOS RAM 中的系统时钟和硬件信息
- 验证系统设置、预配置的硬件设置等，
- 从 RAM、磁盘驱动器、光驱、硬件驱动器等设备开始测试连接的硬件。
- 根据 BIOS 可引导设备信息中的预配置信息，它会根据设置中的可用信息搜索引导驱动器，并开始对其进行初始化以继续进行。

注意：所有 x86 兼容 CPU 在引导期间都以称为实模式的操作模式启动。

什么是可启动设备？

如果设备包含引导扇区或引导块，并且 bios 通过首先将引导扇区加载到内存 (RAM) 中以供执行，然后继续执行，则该设备是可引导设备。

什么是部门？

扇区是可引导磁盘的特定大小的分区。通常一个扇区的大小为 512 字节。在接下来的部分中，我将更多地向您解释如何测量计算机内存以及与之相关的各种术语。

什么是引导扇区？

引导扇区或引导块是可引导设备上的一个区域，其中包含计算机系统的内置固件在初始化期间要加载到 RAM 中的机器代码。它在软盘上为 512 字节。您将在接下来的部分中了解有关字节的更多信息。

可启动设备如何工作？

每当可启动设备被初始化时，bios 都会搜索第一个扇区（称为引导扇区或引导块）并将其加载到 RAM 中并开始执行它。无论引导扇区内的代码是什么，都是您可以编辑的第一个程序，以便在其余时间定义计算机的功能。我在这里的意思是你可以编写自己的代码并将其复制到引导扇区，以使计算机按照你的要求工作。您打算写入设备引导扇区的程序代码也称为引导加载程序。

什么是引导加载程序？

在计算中，引导加载程序是一种特殊的程序，每次可引导设备在计算机开机或重置期间被计算机初始化时都会执行该程序。它是一种可执行的机器代码，非常特定于 CPU 或微处理器类型的硬件架构。

有多少种微处理器可用？

我将主要列出以下内容。

- 16 位
- 32位

- 64位

通常，位数越多，程序访问的内存空间就越多，它们在临时存储等方面的性能就越高。当今有两个主要的微处理器制造商，它们是 Intel 和 AMD。在本文的其余部分，我将仅提及基于 Intel 的系列 (x86) 微处理器。

基于 Intel 的微处理器和基于 AMD 的微处理器有什么区别?

在用于交互的硬件和指令集方面，每家公司都有自己独特的微处理器设计方法。

开发环境介绍。

什么是实模式?

在前面的"计算机启动时会发生什么"一节中，我提到从设备启动时所有 x86 CPU 都以实模式启动。在为任何设备编写引导代码时记下这一点非常重要。实模式仅支持 16 位指令。因此，您为加载到设备的引导记录或引导扇区而编写的代码应仅编译为 16 位兼容代码。在实模式下，指令一次最多可以处理 16 位，例如：一个 16 位的 CPU 将有一个特定的指令，可以在一个 CPU 周期内将两个 16 位数字相加，如果需要的话一个将两个 32 位数字相加的过程，然后将需要更多的周期，从而使用 16 位加法。

什么是指令集?

一个异构的实体集合，它们非常特定于微处理器的体系结构（在设计方面），用户可以使用它与微处理器进行交互。我的意思是一组实体，它包括本机数据类型、指令、寄存器、寻址模式、内存架构、中断和异常处理以及外部 I/O。通常，一组指令对于一系列微处理器是通用的。8086 微处理器是 8086、80286、80386、80486、奔腾、奔腾 I、II、III 家族中的一员。也称为 X86 系列。在本文中，我将参考 x86 系列微处理器的指令集。

如何编写自己的代码来引导设备的扇区?

为了成功完成这项任务，我们需要了解以下内容。

- 操作系统 (GNU Linux)
- 汇编程序（GNU 汇编程序）
- 指令集（x86 系列）
- 在 GNU 汇编器上为 x86 微处理器编写 x86 指令。
- 编译器（C 编程语言 - 可选）
- 链接器（GNU 链接器 ld）
- 一个 x86 模拟器，如用于我们测试目的的 boch。

什么是操作系统?

我将以非常简单的方式解释这一点。由成百上千的专业人士编写的各种程序的大集合，包括帮助全球个人和人们的应用程序和实用程序。从技术角度来看，一般操作系统的一部分主要是为了提供各种应用程序来帮助人们在日常生活中提供很多帮助。比如连接到互联网、聊天、浏览网络、创建文件、保存文件、数据、处理数据等等。我还是不明白。我在这里的意思是你可能想和你的朋友聊天，你可能想在线看新闻，你可能想把一些个人信息写到一个文件中，你可能想看一些电影，你可能想计算一些数学方程式，你可能想玩游戏，您可能想要编写程序等等……所有这些任务都可以通过操作系统来完成。操作系统的工作是提供足够的工具来帮助您并为您服务。您也希望将某些活动进行多任务处理，操作系统的工作是管理硬件并为您提供最佳体验。

另外，请注意，所有现代操作系统都在保护模式下运行。

有哪些不同类型的操作系统?

- 视窗
- Linux
- 苹果电脑

和更多...

什么是保护模式?

与实模式不同，保护模式支持 32 位指令。现在不要太担心它，因为我们不太关心操作系统的工作原理等。

什么是汇编程序?

汇编程序将用户给出的指令转换为机器代码。

甚至编译器也是如此……不是吗?

在更高级别上是的……但它实际上是嵌入在编译器中的汇编器执行此活动。

那为什么编译器不能直接生成机器码呢?

编译器的主要工作主要是将用户编写的指令转换成称为汇编语言指令的中间指令集。然后汇编程序将使用这些指令并将其转换为相应的机器代码。

为什么我需要一个操作系统来编写引导扇区的代码?

现在，我不想进行非常详细的解释，但让我根据本文的范围进行解释。好！前面我提到，为了编写微处理器可以理解的指令，我们需要编译器，而这个编译器是作为操作系统中的实用程序开发的。我告诉过你，操作系统旨在帮助人们提供各种实用程序，而编译器也是实用程序之一。

我可以使用哪种操作系统?

我已经在 Ubuntu 操作系统上编写了从软盘设备启动的程序，所以我推荐 Ubuntu 用于本文。

我应该使用哪个编译器?

我已经使用 GNU GCC 编译器编写了程序，我将介绍如何使用相同的编译器编译代码。如何测试手写代码到设备的引导扇区？我将向您介绍一个 x86 模拟器，它可以在很大程度上帮助我们，而无需每次编辑设备的引导扇区时都重新启动计算机。

微处理器简介

为了学习微处理器编程，首先我们需要学习如何使用寄存器。

什么是寄存器?

寄存器就像微处理器的实用程序，用于临时存储数据并根据我们的要求对其进行操作。假设用户想把3和2相加，用户要求计算机将数字3存储在一个寄存器中，将数字2存储在多个寄存器中，然后将这两个寄存器的内容相加，结果由计算机存储在另一个寄存器中。CPU，这是我们希望看到的输出。有四种类型的寄存器，如下所示。

- 通用寄存器
- 段寄存器
- 堆栈寄存器
- 索引寄存器

让我向您介绍每种类型。

通用寄存器：用于存储程序在其生命周期中所需的临时数据。这些寄存器中的每一个都是 16 位宽或 2 个字节长。

- AX - 累加器寄存器
- BX——基地址寄存器
- CX - 计数寄存器
- DX——数据寄存器

段寄存器：为了表示微处理器的内存地址，我们需要注意两个术语：

- Segment：通常是内存块的开始。

- 偏移量：它是内存块的索引。

示例：假设有一个值为“X”的字节，该字节位于起始地址为 0x7c00 的内存块上，并且该字节位于从头开始的第 10 个位置。在这种情况下，我们将段表示为 0x7c00，偏移量表示为 10。
绝对地址为 0x7c00 + 10。

我想列出四个类别。

- CS - 代码段
- SS - 堆栈段
- DS——数据段
- ES——扩展段

但是这些寄存器总是有限制的。您不能直接为这些寄存器分配地址。我们可以做的是，将地址复制到通用寄存器，然后将地址从该寄存器复制到段寄存器。示例：解决定位字节'X'的问题，我们这样做

自动售货机 复制代码

```
movw $0x07c0, %ax
movw %ax    , %ds
movw (0x0A) , %ax
```

在我们的情况下发生的是

- 在 AX 中设置 0x07c0 * 16
- 设置 DS = AX = 0x7c00
- 将 0x7c00 + 0x0a 设置为 ax

我将描述我们在编写程序时需要了解的各种寻址模式。

堆栈寄存器：

- BP——基指针
- SP - 堆栈指针

索引寄存器：

- SI - 源索引寄存器。
- DI - 目标索引寄存器。
- AX：CPU 将其用于算术运算。
- BX：它可以保存过程或变量的地址（SI、DI 和 BP 也可以）。并且还执行算术和数据移动。
- CX：它充当重复或循环指令的计数器。
- DX：它在乘法中保存乘积的高 16 位（也处理除法运算）。
- CS：它保存程序中所有可执行指令的基址。
- SS：它保存堆栈的基本位置。
- DS：它保存变量的默认基址。
- ES：它为内存变量保存额外的基址。
- BP：它包含与 SS 寄存器的假定偏移量。通常由子例程用于定位调用程序在堆栈上传递的变量。
- SP：包含栈顶的偏移量。
- SI：用于弦移动指令。源字符串由 SI 寄存器指向。
- DI：作为字符串移动指令的目的地。

什么是一点?

在计算中，位是可以存储数据的最小单位。位以二进制形式存储数据。1（开）或 0（关）。

更多关于寄存器：

寄存器按从左到右的顺序或位进一步划分如下：

- AX：AX的前8位标识为AL，后8位标识为AH
- BX：BX的前8位标识为BL，后8位标识为BH
- CX：CX的前8位标识为CL，后8位标识为CH
- DX：DX的前8位标识为DL，后8位标识为DH

如何访问 BIOS 功能?

BIOS 提供了一组功能，可以让我们引起 CPU 的注意。人们将能够通过中断访问 BIOS 功能。

什么是中断?

为了中断程序的普通流程并处理需要迅速响应的事件，我们使用中断。计算机的硬件提供了一种称为中断的机制来处理事件。例如，当鼠标移动时，鼠标硬件会中断当前程序来处理鼠标移动（移动鼠标光标等）。中断导致控制传递给中断处理程序。中断处理程序是处理中断的例程。每种类型的中断都分配了一个整数。在物理内存的开头，有一个中断向量表，其中包含中断处理程序的分段地址。中断数本质上是该表的索引。我们也可以将中断称为 BIOS 提供的服务。

我们将在我们的程序中使用哪种中断服务?

Bios 中断 0x10。

在汇编程序中编写代码

GNU 汇编器中有哪些可用的各种数据类型?

一组位，用于表示一个单元以构成各种数据类型。

什么是数据类型?

数据类型用于标识数据的特征。各种数据类型如下。

- 字节
- 单词
- 整数
- ASCII码
- ASCII码

字节：它是八位长。字节被认为是计算机上可以通过编程存储数据的最小单位。

word：它是一个 16 位长的数据单元。

什么是整数?

int 是一种数据类型，表示 32 位长的数据。四个字节或两个字构成一个 int。

什么是ascii?

表示不带空终止符的一组字节的数据类型。

什么是 asciz?

表示以空字符结尾的一组字节的数据类型。

如何通过汇编程序为实模式生成代码?

当 CPU 以实模式（16 位）启动时，从设备启动时我们所能做的就是利用 BIOS 提供的内置功能继续进行。我这里的意思是我们可以利用BIOS的功能来编写我们自己的引导加载程序代码，然后转储到设备的引导扇区，然后引导它。让我们看看如何在汇编器中编写一小段代码，通过 GNU Assembler 生成 16 位 CPU 代码。

复制代码

```
Example: test.S
```

自动售货机 复制代码

```
.code16                 #generate 16-bit code
.text                   #executable code location
     .globl _start;
_start:                 #code entry point
     . = _start + 510   #mov to 510th byte from 0 pos
     .byte 0x55         #append boot signature
     .byte 0xaa         #append boot signature
```

让我解释一下上面代码中的每个语句。

- .code16：它是提供给汇编程序的指令或命令，用于生成 16 位代码而不是 32 位代码。为什么需要这个提示？请记住，您将使用操作系统来利用汇编器和编译器来编写引导加载程序代码。但是，我也提到过操作系统在 32 位保护模式下工作。因此，当您在受保护模式的操作系统上使用汇编程序时，它默认配置为生成 32 位代码而不是 16 位代码，因为我们需要 16 位代码，这并不能达到目的。为避免汇编器和编译器生成 32 位代码，我们使用此指令。
- .text：.text 部分包含构成程序的实际机器指令。
- .globl _start: .global <symbol> 使符号对链接器可见。如果您在部分程序中定义符号，则其值可用于与其链接的其他部分程序。否则，符号从链接到同一程序的另一个文件中的同名符号中获取其属性。
- _start：主代码入口，_start 是链接器的默认入口点。
- . = _start + 510：从头遍历到第 510 个字节
- .byte 0x55：它是标识为引导签名一部分的第一个字节。（第 511 个字节）
- .byte 0xaa：它是标识为引导签名一部分的最后一个字节。（第 512 个字节）

如何编译汇编程序?

将代码保存为*test.S*文件。在命令提示符下键入以下内容：

- 作为 test.S -o test.o
- ld –Ttext 0x7c00 --oformat=binary test.o –o test.bin

无论如何，上述命令对我们意味着什么?

- as test.S –o test.o：此命令将给定的汇编代码转换为相应的目标代码，目标代码是在转换为机器代码之前由汇编程序生成的中间代码。
- --oformat=binary 开关告诉链接器你希望你的输出文件是一个普通的二进制图像（没有启动代码，没有重定位，.....）。
- –Ttext 0x7c00 告诉链接器您希望将“文本”（代码段）地址加载到 0x7c00，从而计算绝对寻址的正确地址。

什么是引导签名?

记得之前我在介绍 BIOS 程序加载的引导记录或引导扇区。BIOS 如何识别设备是否包含引导扇区？为了回答这个问题，我可以告诉你一个引导扇区长 512 个字节，在第 510 个字节中需要一个符号 0x55，在第 511 个字节中需要另一个符号 0xaa。因此，我验证引导扇区的最后两个字节是否为 0x55 和 0xaa，如果是，则将该扇区标识为引导扇区并继续执行引导扇区代码，否则会引发设备不可引导的错误。使用十六进制编辑器，您可以以更易读的方式查看二进制文件的内容，以下是使用 hexedit 工具查看文件时的快照供您参考。

如何将可执行代码复制到可启动设备，然后对其进行测试?

要创建 1.4mb 大小的软盘映像，请在命令提示符下键入以下内容。

- dd if=/dev/zero of=floppy.img bs=512 count=2880

要将代码复制到软盘映像文件的引导扇区，请在命令提示符下键入以下内容。

- dd if=test.bin of=floppy.img

要测试程序，请在命令提示符下键入以下内容

- 博克斯

如果没有安装 bochs 那么你可以输入以下命令

- sudo apt-get install bochs-x

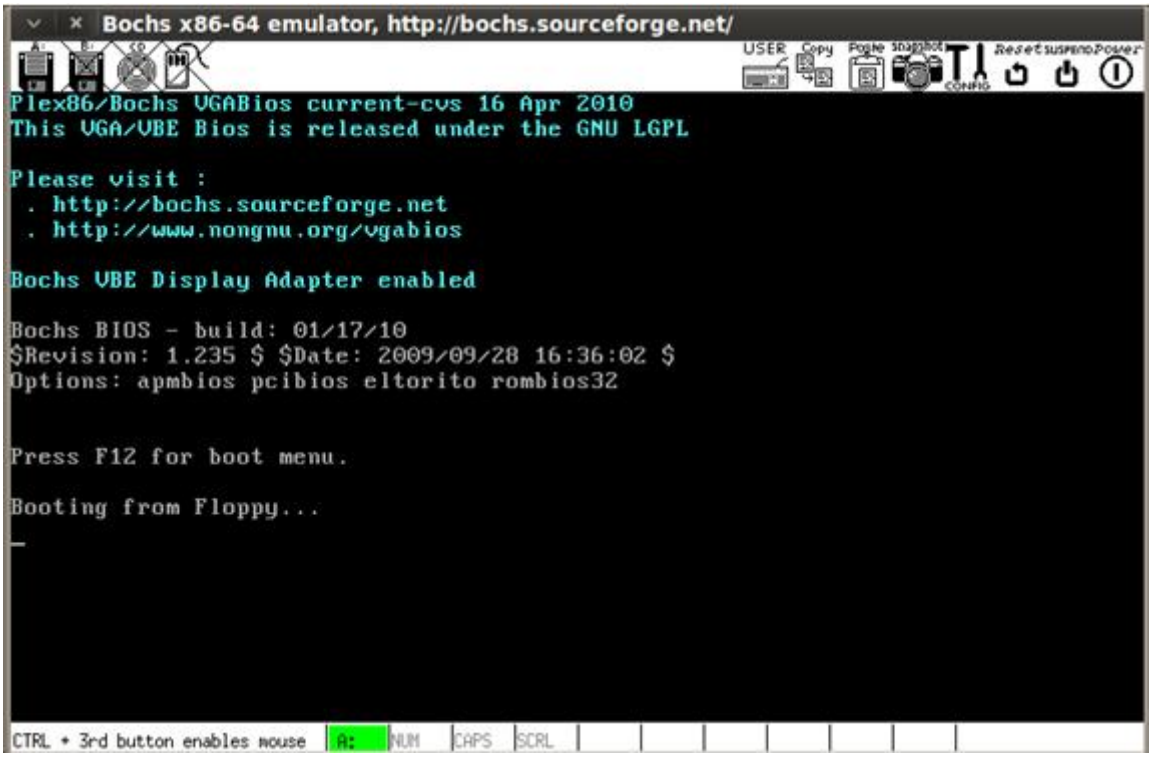
复制代码

```
Sample bochsrc.txt file
```

自动售货机 复制代码

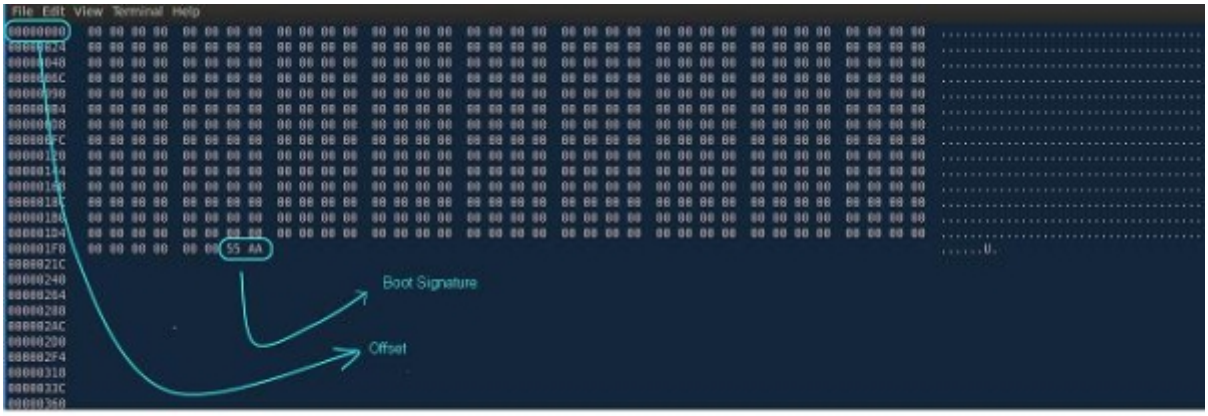
```
megs: 32
#romimage: file=/usr/local/bochs/1.4.1/BIOS-bochs-latest, address=0xf0000
#vgaromimage: /usr/local/bochs/1.4.1/VGABIOS-elpin-2.40
floppya: 1_44=floppy.img, status=inserted
boot: a
log: bochsout.txt
mouse: enabled=0
```

您应该会看到一个典型的 boch 模拟窗口，如下所示。



观察：

现在，如果您在十六进制编辑器中查看 test.bin 文件，您将看到引导签名位于第 510 个字节的末尾，以下是供您参考的屏幕截图。



什么都没有发生，因为我们没有写任何东西在我们的代码中显示在屏幕上。所以你只会看到一条消息“从软盘启动”。让我们再看几个在汇编器上编写汇编代码的例子。

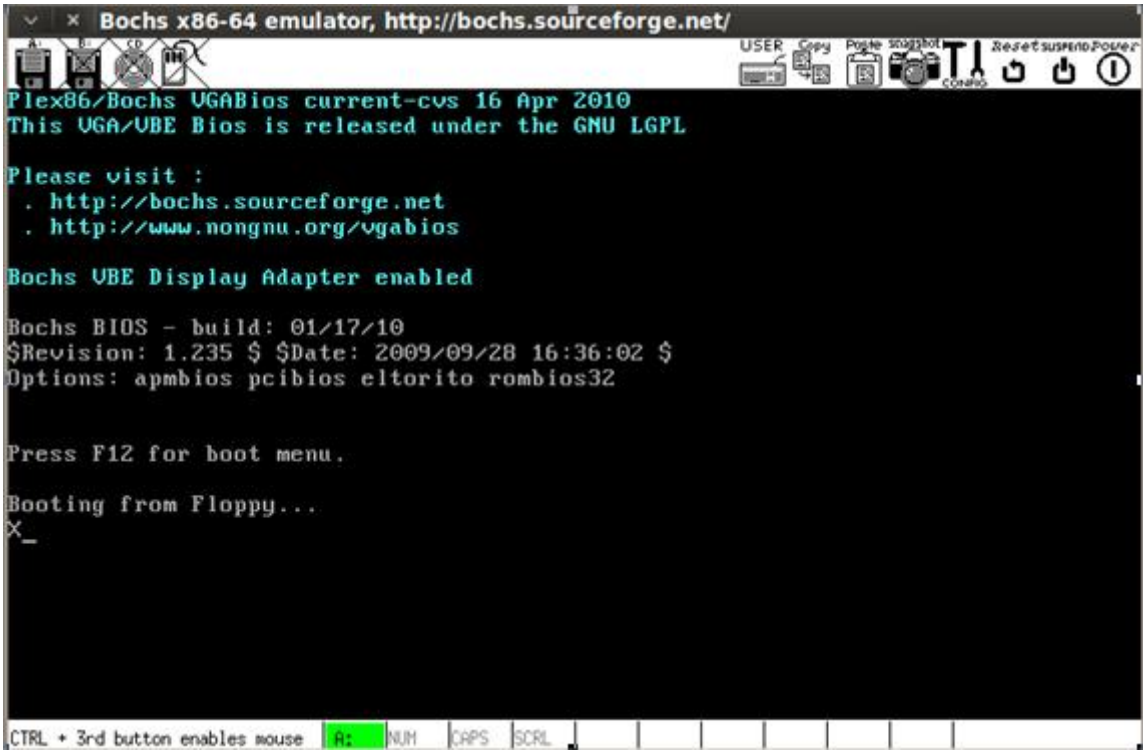
复制代码

```
Example: test2.S
```

自动售货机 复制代码

```
.code16                     #generate 16-bit code
.text                       #executable code location
     .globl _start;
_start:                     #code entry point

     movb $'X' , %al        #character to print
     movb $0x0e, %ah        #bios service code to print
     int  $0x10             #interrupt the cpu now

     . = _start + 510       #mov to 510th byte from 0 pos
     .byte 0x55             #append boot signature
     .byte 0xaa             #append boot signature
```

键入以上内容后，保存到 test2.S 中，然后通过更改源文件名按照之前的说明进行编译。当您编译并成功将此代码复制到引导扇区并运行 bochs 时，您应该看到以下屏幕。在命令提示符下键入 bochs 以查看结果，您应该会在屏幕上看到字母“X”，如下面的屏幕截图所示。



恭喜！！！

观察：

如果在十六进制编辑器中查看，您将看到字符“X”位于起始地址的第二个位置。

现在让我们做一些不同的事情，比如在屏幕上打印文本。

复制代码

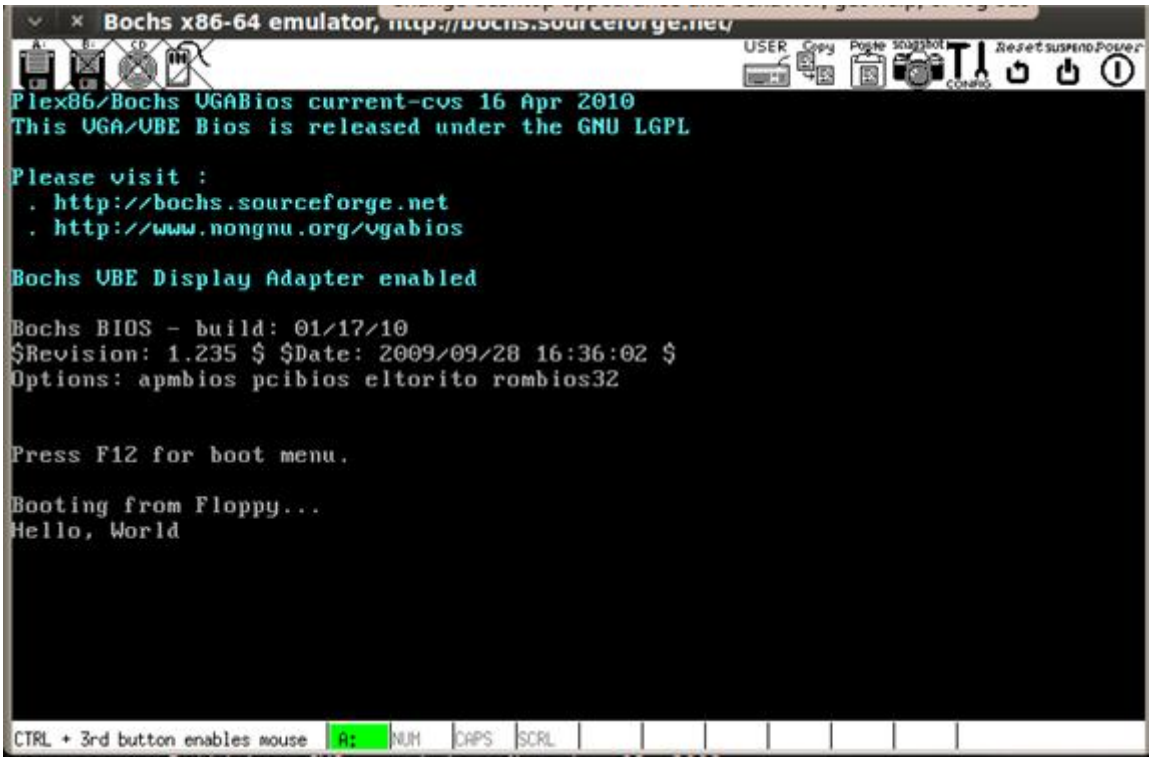
```
Example: test3.S
```

自动售货机 缩小▲ 复制代码

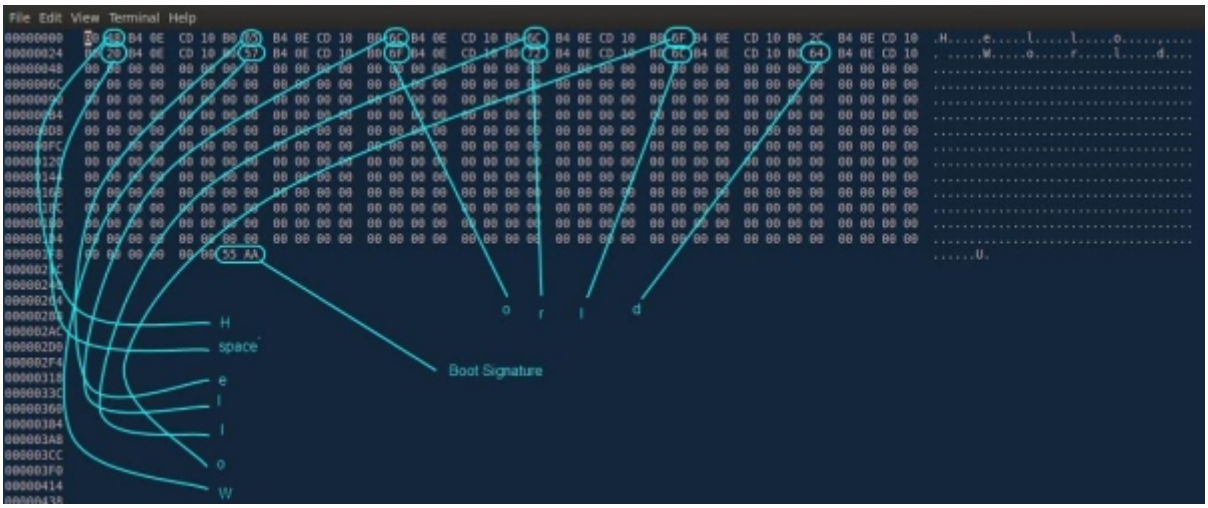
```
.code16                 #generate 16-bit code
.text                   #executable code location
     .globl _start;

_start:                 #code entry point

     #print letter 'H' onto the screen
     movb $'H' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'e' onto the screen
     movb $'e' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'l' onto the screen
     movb $'l' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'l' onto the screen
     movb $'l' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'o' onto the screen
     movb $'o' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter ',' onto the screen
     movb $',' , %al
     movb $0x0e, %ah
     int  $0x10

     #print space onto the screen
     movb $' ' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'W' onto the screen
     movb $'W' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'o' onto the screen
     movb $'o' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'r' onto the screen
     movb $'r' , %al
     movb $0x0e, %ah
     int  $0x10

     #print letter 'l' onto the screen
     movb $'l' , %al
     movb $0x0e, %ah
     int  $0x10
```

```
    #print letter 'd' onto the screen
    movb $'d' , %al
    movb $0x0e, %ah
    int  $0x10

    . = _start + 510      #mov to 510th byte from 0 pos
    .byte 0x55            #append boot signature
    .byte 0xaa            #append boot signature
```

将其另存为*test3.S*。当您编译并成功将此代码复制到引导扇区并运行 bochs 时，您应该看到以下屏幕。



观察：



好的.....现在我们做了一些与之前程序不同的事情。

复制代码

```
Let us write an assembly program to print the letters "Hello, World" onto the screen.
```

我们还将尝试定义函数和宏，通过它们我们将尝试打印字符串。

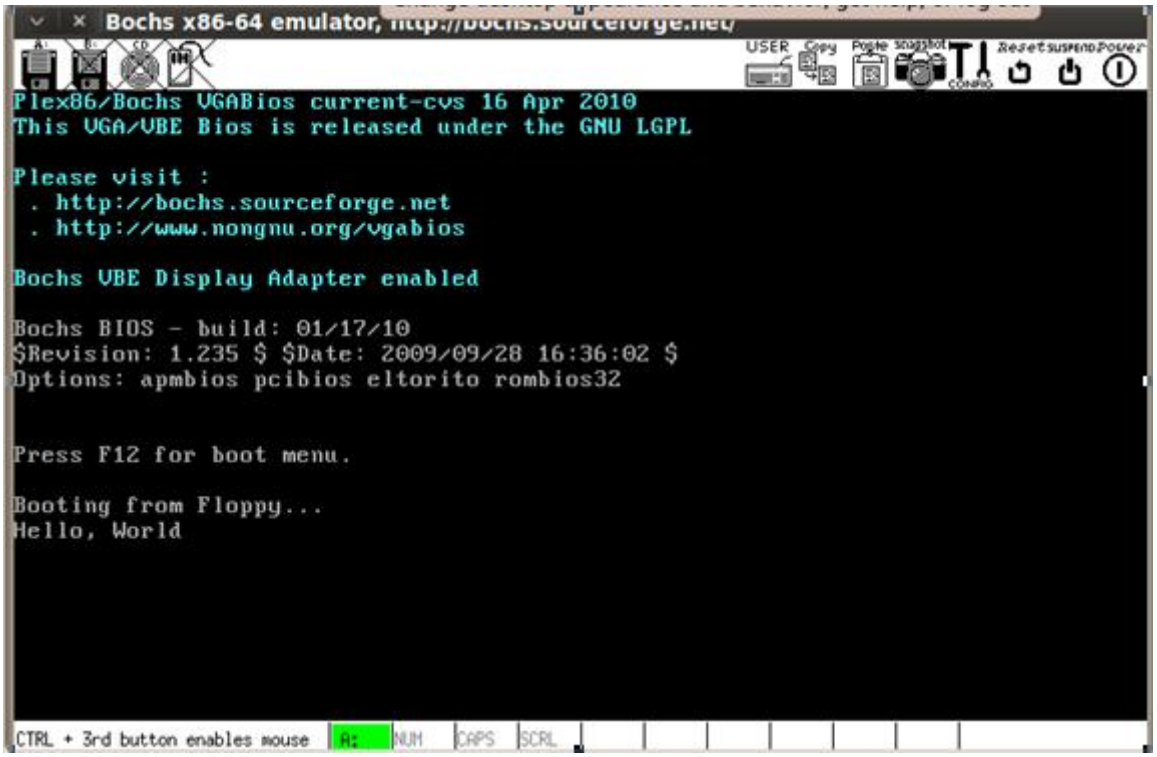
复制代码

```
Example: test4.S
```

自动售货机 缩小▲ 复制代码

```
#generate 16-bit code
.code16

#hint the assembler that here is the executable code located
.text
.globl _start;
#boot code entry
_start:
      jmp _boot                          #jump to boot code
      welcome: .asciz "Hello, World\n\r"  #here we define the string

     .macro mWriteString str             #macro which calls a function to print a string
          leaw  \str, %si
          call .writeStringIn
     .endm

     #function to print the string
     .writeStringIn:
          lodsb
          orb  %al, %al
          jz   .writeStringOut
          movb $0x0e, %ah
          int  $0x10
          jmp  .writeStringIn
     .writeStringOut:
     ret

_boot:
     mWriteString welcome

     #move to 510th byte from the start and append boot signature
     . = _start + 510
     .byte 0x55
     .byte 0xaa
```

将其另存为*test4.S*。当您编译并成功将此代码复制到引导扇区并运行 bochs 时，您应该看到以下屏幕。



好！！！如果您确实理解了我所做的并且能够编写类似的程序，那么再次恭喜您！

观察：

什么是函数?

函数是具有名称和可重用属性的代码块。

什么是宏?

宏是一段代码，它被赋予了名字。无论何时使用该名称，它都会被宏的内容替换。

宏和函数在语法方面有什么区别?

要调用函数，我们使用以下语法。

- push <参数>
- 调用 <函数名>

要调用宏，我们使用以下语法

- 宏名 <参数>

但是与函数的调用和使用语法相比，宏的调用和使用语法非常简单。所以我更喜欢写一个宏并使用它而不是在主代码中调用一个函数。您可以在网上参考更多关于如何在 GNU Assembler 上编写汇编代码的资料。

在 C 编译器中编写代码

什么是C?

在计算领域，C 是一种通用编程语言，最初由丹尼斯·里奇 (Dennis Ritchie) 于 1969 年至 1973 年间在 AT&T 贝尔实验室开发。

为什么要使用 C？一种依赖机器的语言，但用 C 编写的程序通常很小且执行起来很快。该语言包括通常仅在汇编语言或机器语言中可用的低级功能。C 是一种结构化的编程语言。

为什么我需要用 C 编写代码?

好吧，如果您想编写较小的程序并希望它们非常快，那就去做吧。

用C语言编写代码需要什么?

好吧，我们将使用名为 gcc 的 GNU C 编译器来编写 C 代码。

如何在 C 中的 GCC 编译器中编写程序?

让我们编写一个程序来看看它的样子。

复制代码

```
Example: test.c
```

C++ 复制代码

```
__asm__(".code16\n");
__asm__("jmpl $0x0000, $main\n");

void main() {
}
```

复制代码

```
File: test.ld
```

C++ 复制代码

```
ENTRY(main);
SECTIONS
{
    . = 0x7C00;
    .text : AT(0x7C00)
    {
        *(.text);
    }
    .sig : AT(0x7DFE)
    {
        SHORT(0xaa55);
    }
}
```

如何编译C程序？在命令提示符下键入以下内容：

- gcc -c -g -Os -march=i686 -ffreestanding -Wall -Werror test.c -o test.o
- ld -static -Ttest.ld -nostdlib --nmagic -o test.elf test.o
- objcopy -O 二进制 test.elf test.bin

无论如何，上述命令对我们意味着什么？

此命令将给定的 C 代码转换为相应的目标代码，目标代码是在转换为机器代码之前由编译器生成的中间代码。

- gcc -c -g -Os -march=i686 -ffreestanding -Wall -Werror test.c -o test.o：

每个标志是什么意思？

- -c：用于在不链接的情况下编译给定的源代码。
- -g：生成供 GDB 调试器使用的调试信息。
- -Os：代码大小优化
- -march：为特定的 CPU 架构生成代码（在我们的例子中是 i686）
- -ffreestanding：独立环境是一种标准库可能不存在的环境，程序启动可能不一定在“main”。
- -Wall：启用所有编译器的警告消息。应始终使用此选项，以便生成更好的代码。
- -Werror：启用被视为错误的警告
- test.c：输入源文件名
- -o：生成目标代码
- test.o：输出目标代码文件名。

通过编译器的所有上述标志组合，我们尝试生成目标代码，帮助我们识别错误、警告，并为 CPU 类型生成更有效的代码。如果您不指定 March=i686，它会为您拥有的机器类型生成代码，否则它会在移植时更好地指定您的目标 CPU 类型。

- ld -static -Ttest.ld -nostdlib --nmagic test.elf -o test.o：

这是从命令提示符调用链接器的命令，我在下面解释了我们试图用链接器做什么。

每个标志是什么意思?

- -static：不链接共享库。
- -Ttest.ld：此功能允许链接器遵循链接器脚本中的命令。
- -nostdlib：此功能允许链接器通过链接非标准 C 库启动函数来生成代码。
- --nmagic：此功能允许链接器生成没有 _start_SECTION 和 _stop_SECTION 代码的代码。
- test.elf：输入文件名（存储可执行文件的平台依赖文件格式 Windows：PE，Linux：ELF）
- -o：生成目标代码
- test.o：输出目标代码文件名。

什么是链接器?

这是编译的最后阶段。ld(linker) 将一个或多个目标文件或库作为输入，并将它们组合起来生成一个（通常是可执行的）文件。在此过程中，它解析对外部符号的引用，为过程/函数和变量分配最终地址，并修改代码和数据以反映新地址（称为重定位的过程）。

还要记住，我们的代码中没有标准库和所有花哨的函数。

- objcopy -O 二进制 test.elf test.bin

该命令用于生成平台无关代码。请注意，Linux 以不同于 Windows 的方式存储可执行文件。每个人都有自己的文件存储方式，但我们只是开发了一个小代码来启动，目前不依赖于任何操作系统。因此，我们不依赖任何一个，因为我们不需要操作系统在启动时运行我们的代码。

为什么在 C 程序中使用汇编语句?

在实模式下，可以使用汇编语言指令通过软件中断轻松访问 BIOS 功能。这导致在我们的 C 代码中使用内联汇编。

如何将可执行代码复制到可启动设备，然后对其进行测试?

要创建 1.4mb 大小的软盘映像，请在命令提示符下键入以下内容。

- dd if=/dev/zero of=floppy.img bs=512 count=2880

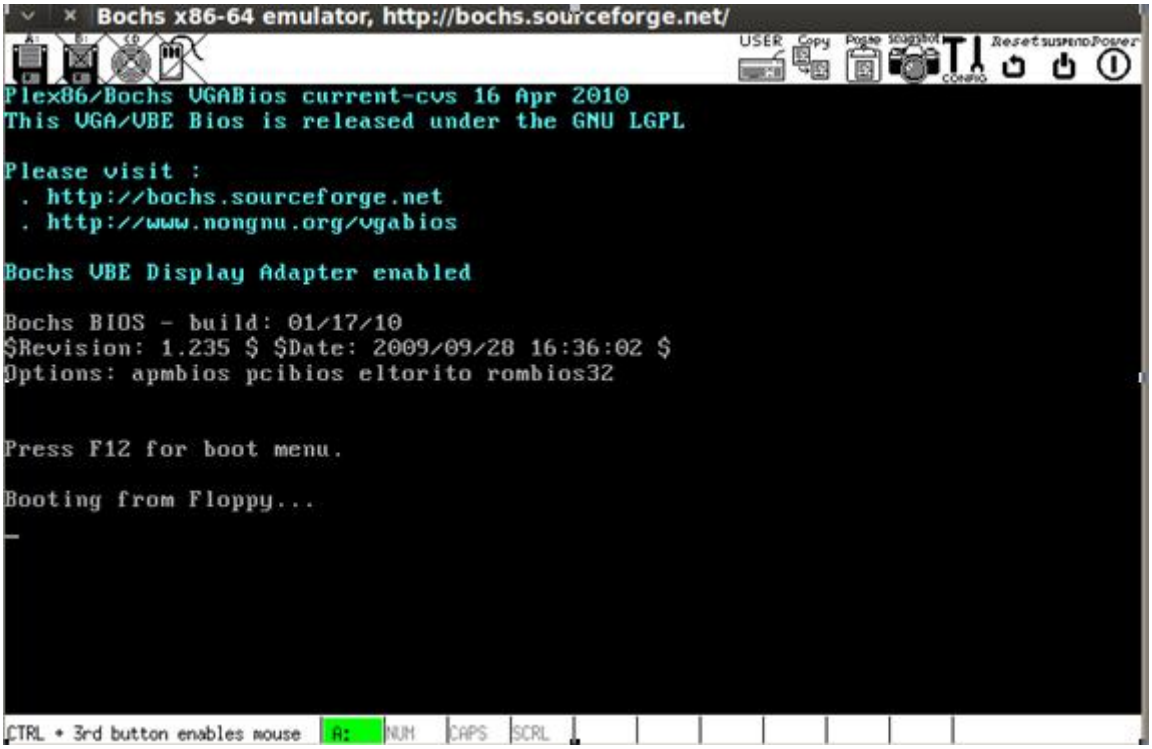
要将代码复制到软盘映像文件的引导扇区，请在命令提示符下键入以下内容。

- dd if=test.bin of=floppy.img

要测试程序，请在命令提示符下键入以下内容

- 博克斯

您应该会看到一个典型的 boch 模拟窗口，如下所示。



观察：什么都没有发生，因为我们没有写任何东西在我们的代码中显示在屏幕上。所以你只会看到一条消息“从软盘启动”。恭喜！！！

- 我们使用__asm__关键字将汇编语言语句嵌入到 C 程序中。该关键字提示编译器识别它是用户给出的汇编指令。
- 我们还__volatile__用来提示汇编器不要修改我们的代码，让它保持原样。

这种在 C 代码中嵌入汇编代码的方式称为内联汇编。

让我们再看几个在编译器上编写代码的例子。

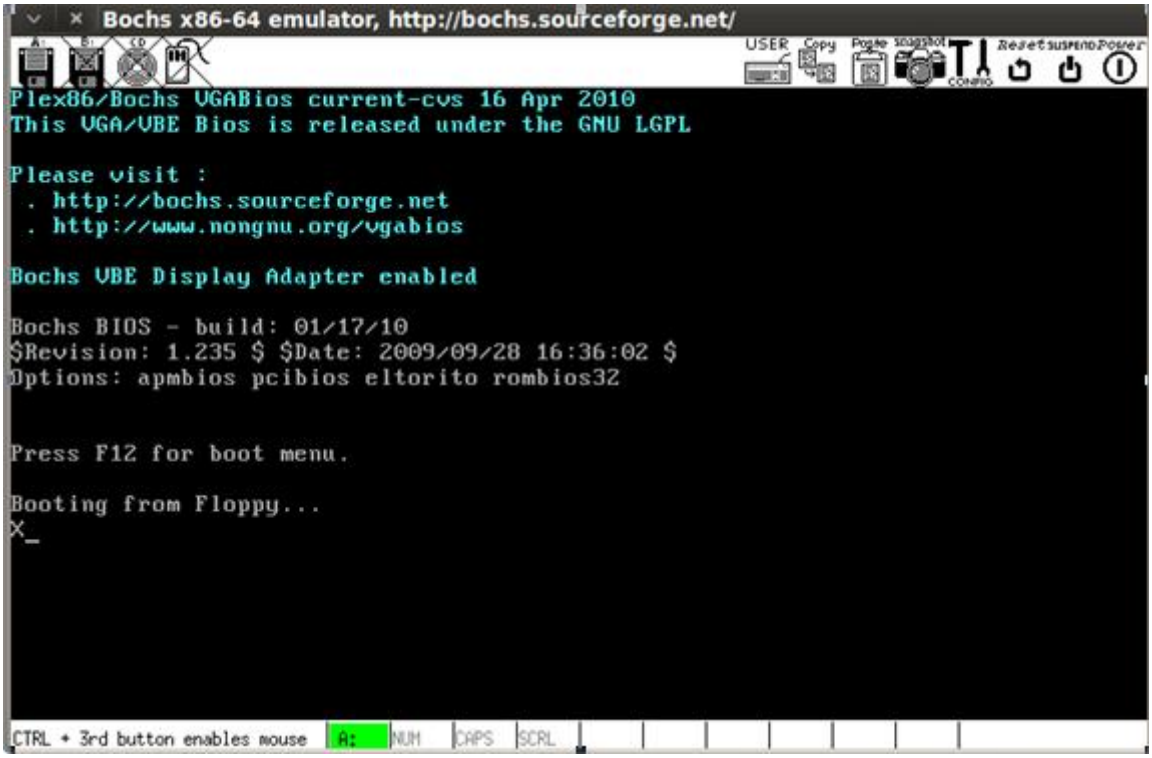
让我们编写一个汇编程序将字母“X”打印到屏幕上。

示例：test2.c

C++ 复制代码

```
__asm__(".code16\n");
__asm__("jmpl $0x0000, $main\n");

void main() {
     __asm__ __volatile__ ("movb $'X'  , %al\n");
     __asm__ __volatile__ ("movb $0x0e, %ah\n");
     __asm__ __volatile__ ("int $0x10\n");
}
```

键入以上内容后，保存到*test2.c* 中，然后通过更改源文件名按照之前的说明进行编译。当您编译并成功将此代码复制到引导扇区并运行 bochs 时，您应该看到以下屏幕。在命令提示符下键入 bochs 以查看结果，您应该会在屏幕上看到字母“X”，如下面的屏幕截图所示。



现在，让我们编写 ac 程序将字母“Hello, World”打印到屏幕上。

我们还将尝试定义函数和宏，通过它们我们将尝试打印字符串。

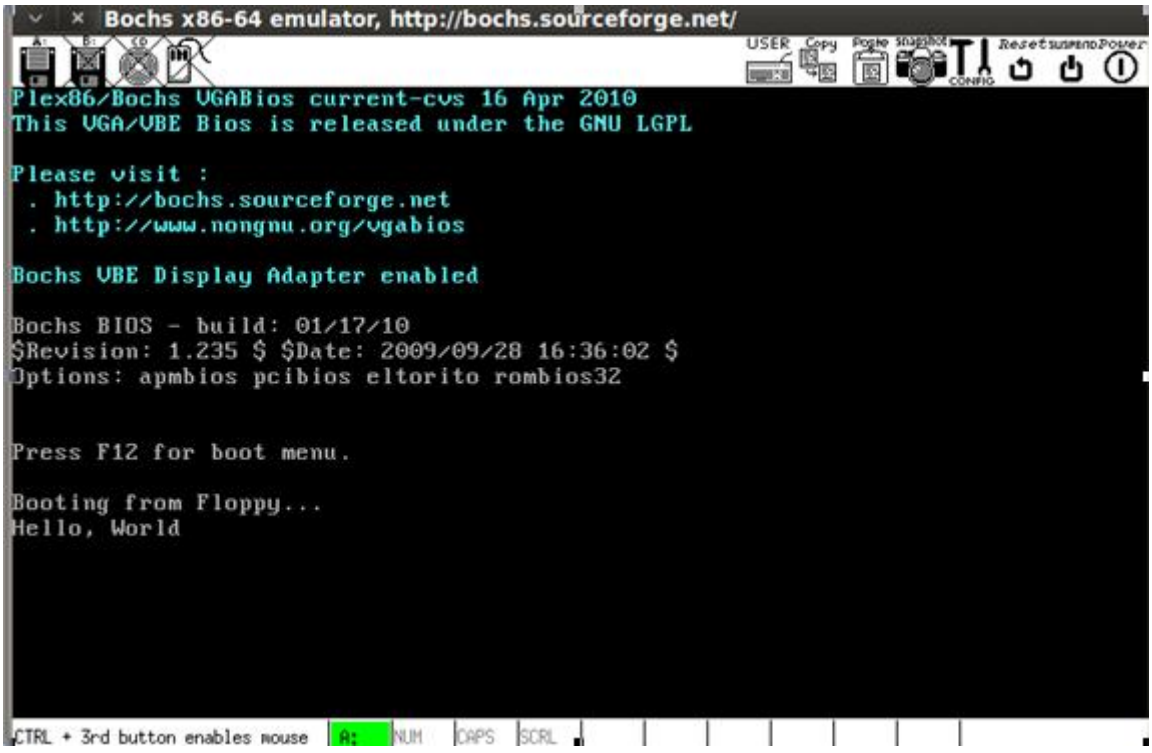
示例：test3.c

C++ 缩小▲ 复制代码

```
/*generate 16-bit code*/
__asm__(".code16\n");
/*jump boot code entry*/
__asm__("jmpl $0x0000, $main\n");

void main() {
     /*print letter 'H' onto the screen*/
     __asm__ __volatile__("movb $'H' , %al\n");
     __asm__ __volatile__("movb $0x0e, %ah\n");
     __asm__ __volatile__("int  $0x10\n");

     /*print letter 'e' onto the screen*/
     __asm__ __volatile__("movb $'e' , %al\n");
     __asm__ __volatile__("movb $0x0e, %ah\n");
     __asm__ __volatile__("int  $0x10\n");

     /*print letter 'l' onto the screen*/
     __asm__ __volatile__("movb $'l' , %al\n");
     __asm__ __volatile__("movb $0x0e, %ah\n");
     __asm__ __volatile__("int  $0x10\n");

     /*print letter 'l' onto the screen*/
     __asm__ __volatile__("movb $'l' , %al\n");
     __asm__ __volatile__("movb $0x0e, %ah\n");
     __asm__ __volatile__("int  $0x10\n");

     /*print letter 'o' onto the screen*/
     __asm__ __volatile__("movb $'o' , %al\n");
     __asm__ __volatile__("movb $0x0e, %ah\n");
     __asm__ __volatile__("int  $0x10\n");

     /*print letter ',' onto the screen*/
```

```
    __asm__ __volatile__("movb $',' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter ' ' onto the screen*/
    __asm__ __volatile__("movb $' ' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter 'W' onto the screen*/
    __asm__ __volatile__("movb $'W' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter 'o' onto the screen*/
    __asm__ __volatile__("movb $'o' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter 'r' onto the screen*/
    __asm__ __volatile__("movb $'r' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter 'l' onto the screen*/
    __asm__ __volatile__("movb $'l' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");

    /*print letter 'd' onto the screen*/
    __asm__ __volatile__("movb $'d' , %al\n");
    __asm__ __volatile__("movb $0x0e, %ah\n");
    __asm__ __volatile__("int  $0x10\n");
}
```

现在将上面的代码保存为*test3.c*，然后按照修改输入源文件名给出的编译说明，按照给出的说明将编译后的代码复制到软盘的引导扇区。现在观察结果。如果一切正常，您应该会看到以下屏幕输出。



让我们编写一个 C 程序，将字母“Hello, World”打印到屏幕上。

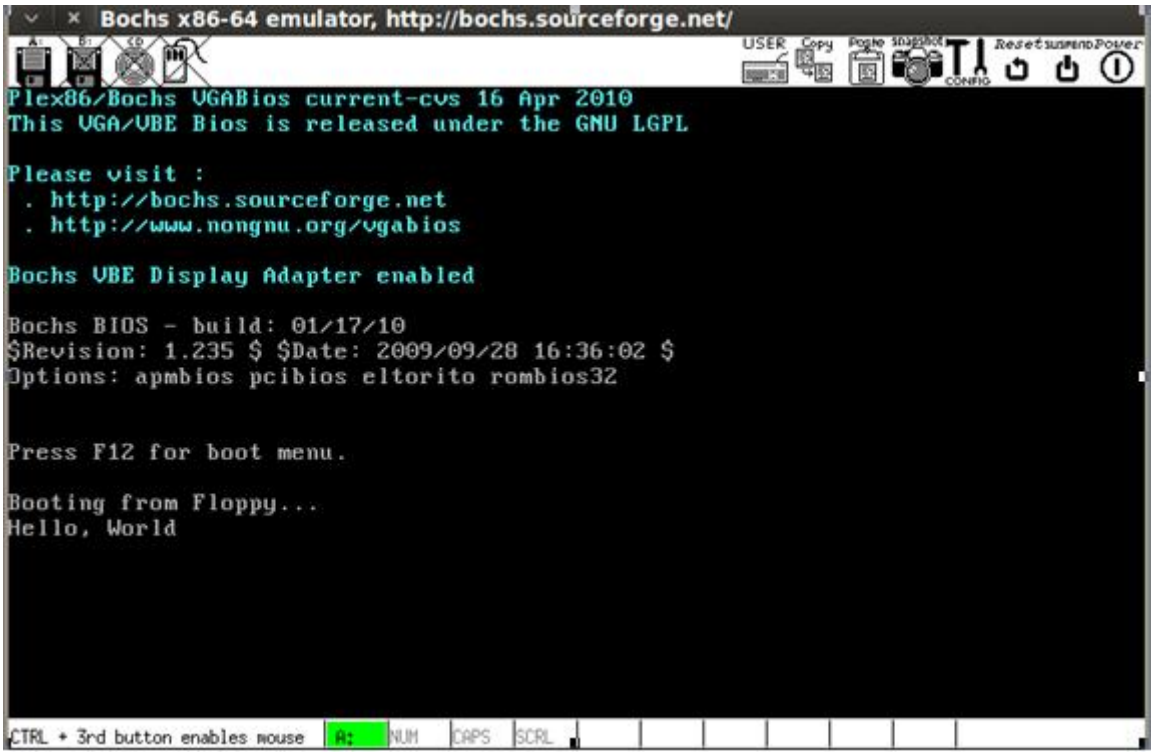
我们还将尝试定义我们将尝试打印字符串的函数。

示例：test4.c

C++ 复制代码

```
/*generate 16-bit code*/
__asm__(".code16\n");
/*jump boot code entry*/
__asm__("jmpl $0x0000, $main\n");

/* user defined function to print series of characters terminated by null character */
void printString(const char* pStr) {
     while(*pStr) {
          __asm__ __volatile__ (
               "int $0x10" : : "a"(0x0e00 | *pStr), "b"(0x0007)
          );
          ++pStr;
     }
}

void main() {
     /* calling the printString function passing string as an argument */
     printString("Hello, World");
}
```

现在将上面的代码保存为*test3.c*，然后按照修改输入源文件名给出的编译说明，按照给出的说明将编译后的代码复制到软盘的引导扇区。现在观察结果。如果一切正常，您应该会看到以下屏幕输出。



我想提请你注意一点。我们所做的只是通过学习将之前编写的汇编程序转换为 C 程序。到现在为止，您应该可以轻松地用汇编和 C 编写程序，并且也很清楚如何编译和测试它们。

现在我们将继续编写循环并使它们在函数内工作，并且还会看到更多的 bios 服务。

一个显示矩形的小项目

现在让我们转向更大的东西……比如显示图形。

示例：test5.c

C++ 缩小▲ 复制代码

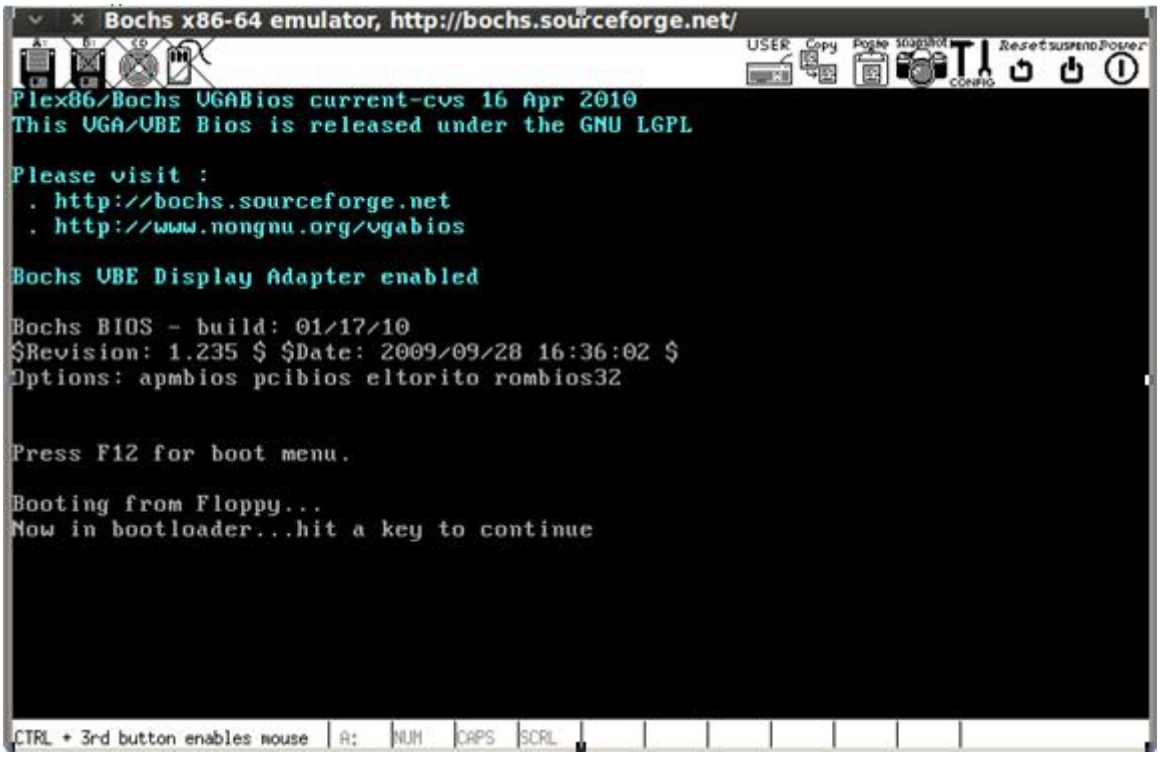
```
/* generate 16 bit code                                                 */
__asm__(".code16\n");
/* jump to main function or program code                                */
__asm__("jmpl $0x0000, $main\n");

#define MAX_COLS     320 /* maximum columns of the screen               */
#define MAX_ROWS     200 /* maximum rows of the screen                  */

/* function to print string onto the screen                             */
/* input ah = 0x0e                                                      */
/* input al = <character to print>                                      */
/* interrupt: 0x10                                                      */
/* we use interrupt 0x10 with function code 0x0e to print               */
/* a byte in al onto the screen                                         */
/* this function takes string as an argument and then                   */
/* prints character by character until it founds null                   */
/* character                                                            */
void printString(const char* pStr) {
     while(*pStr) {
          __asm__ __volatile__ (
               "int $0x10" : : "a"(0x0e00 | *pStr), "b"(0x0007)
          );
          ++pStr;
     }
}

/* function to get a keystroke from the keyboard                        */
/* input ah = 0x00                                                      */
/* input al = 0x00                                                      */
/* interrupt: 0x10                                                      */
/* we use this function to hit a key to continue by the                 */
/* user                                                                 */
void getch() {
     __asm__ __volatile__ (
          "xorw %ax, %ax\n"
          "int $0x16\n"
     );
}

/* function to print a colored pixel onto the screen                    */
/* at a given column and at a given row                                 */
/* input ah = 0x0c                                                      */
/* input al = desired color                                             */
/* input cx = desired column                                            */
/* input dx = desired row                                               */
/* interrupt: 0x10                                                      */
void drawPixel(unsigned char color, int col, int row) {
     __asm__ __volatile__ (
          "int $0x10" : : "a"(0x0c00 | color), "c"(col), "d"(row)
     );
}

/* function to clear the screen and set the video mode to               */
/* 320x200 pixel format                                                 */
/* function to clear the screen as below                                */
/* input ah = 0x00                                                      */
/* input al = 0x03                                                      */
/* interrupt = 0x10                                                     */
/* function to set the video mode as below                              */
/* input ah = 0x00                                                      */
/* input al = 0x13                                                      */
/* interrupt = 0x10                                                     */
```

```
void initEnvironment() {
     /* clear screen                                                */
     __asm__ __volatile__ (
          "int $0x10" : : "a"(0x03)
     );
     __asm__ __volatile__ (
          "int $0x10" : : "a"(0x0013)
     );
}

/* function to print rectangles in descending order of              */
/* their sizes                                                      */
/* I follow the below sequence                                      */
/* (left, top)     to (left, bottom)                                */
/* (left, bottom)  to (right, bottom)                               */
/* (right, bottom) to (right, top)                                  */
/* (right, top)    to (left, top)                                   */
void initGraphics() {
     int i = 0, j = 0;
     int m = 0;
     int cnt1 = 0, cnt2 =0;
     unsigned char color = 10;

     for(;;) {
          if(m < (MAX_ROWS - m)) {
               ++cnt1;
          }
          if(m < (MAX_COLS - m - 3)) {
               ++cnt2;
          }

          if(cnt1 != cnt2) {
               cnt1  = 0;
               cnt2  = 0;
               m     = 0;
               if(++color > 255) color= 0;
          }

          /* (left, top) to (left, bottom)                          */
          j = 0;
          for(i = m; i < MAX_ROWS - m; ++i) {
               drawPixel(color, j+m, i);
          }
          /* (left, bottom) to (right, bottom)                      */
          for(j = m; j < MAX_COLS - m; ++j) {
               drawPixel(color, j, i);
          }

          /* (right, bottom) to (right, top)                        */
          for(i = MAX_ROWS - m - 1 ; i >= m; --i) {
               drawPixel(color, MAX_COLS - m - 1, i);
          }
          /* (right, top)   to (left, top)                          */
          for(j = MAX_COLS - m - 1; j >= m; --j) {
               drawPixel(color, j, m);
          }
          m += 6;
          if(++color > 255)  color = 0;
     }
}

/* function is boot code and it calls the below functions           */
/* print a message to the screen to make the user hit the           */
/* key to proceed further and then once the user hits then          */
/* it displays rectangles in the descending order                   */
void main() {
     printString("Now in bootloader...hit a key to continue\n\r");
     getch();
     initEnvironment();
```

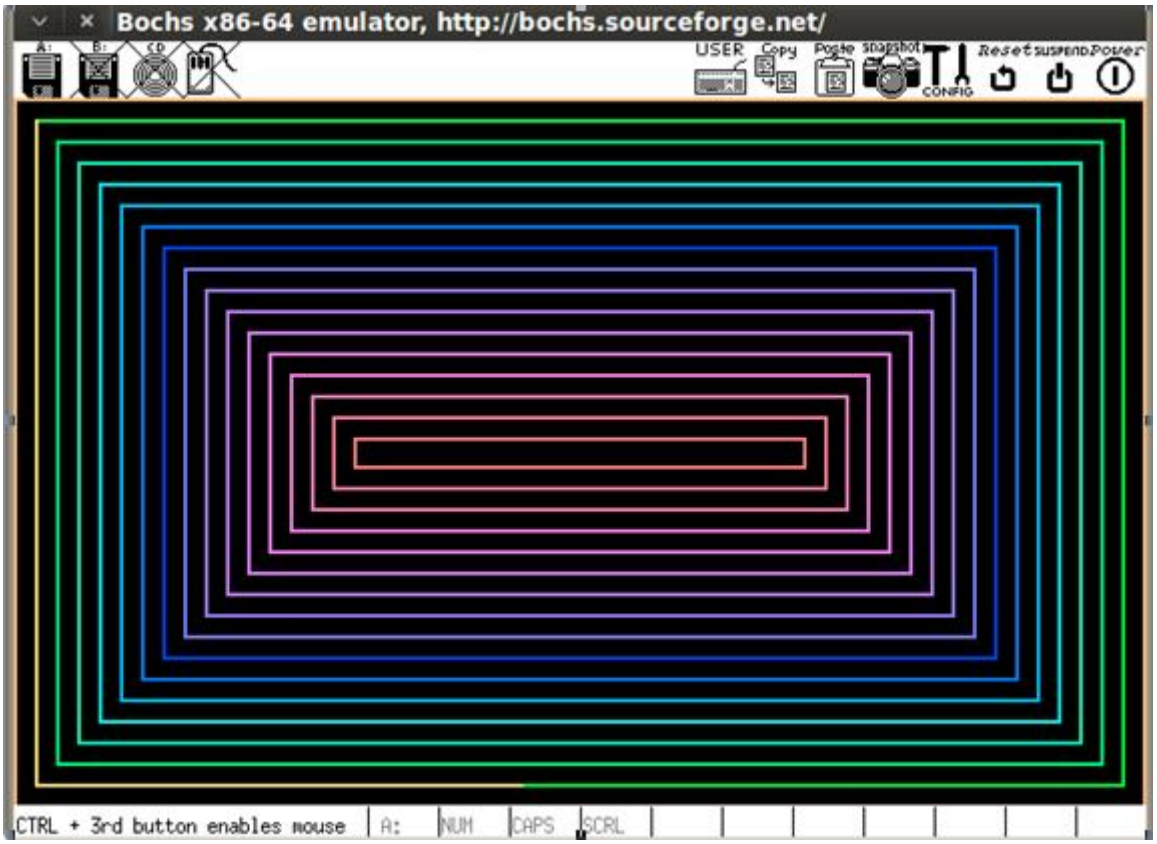
```
    initGraphics();
}
```

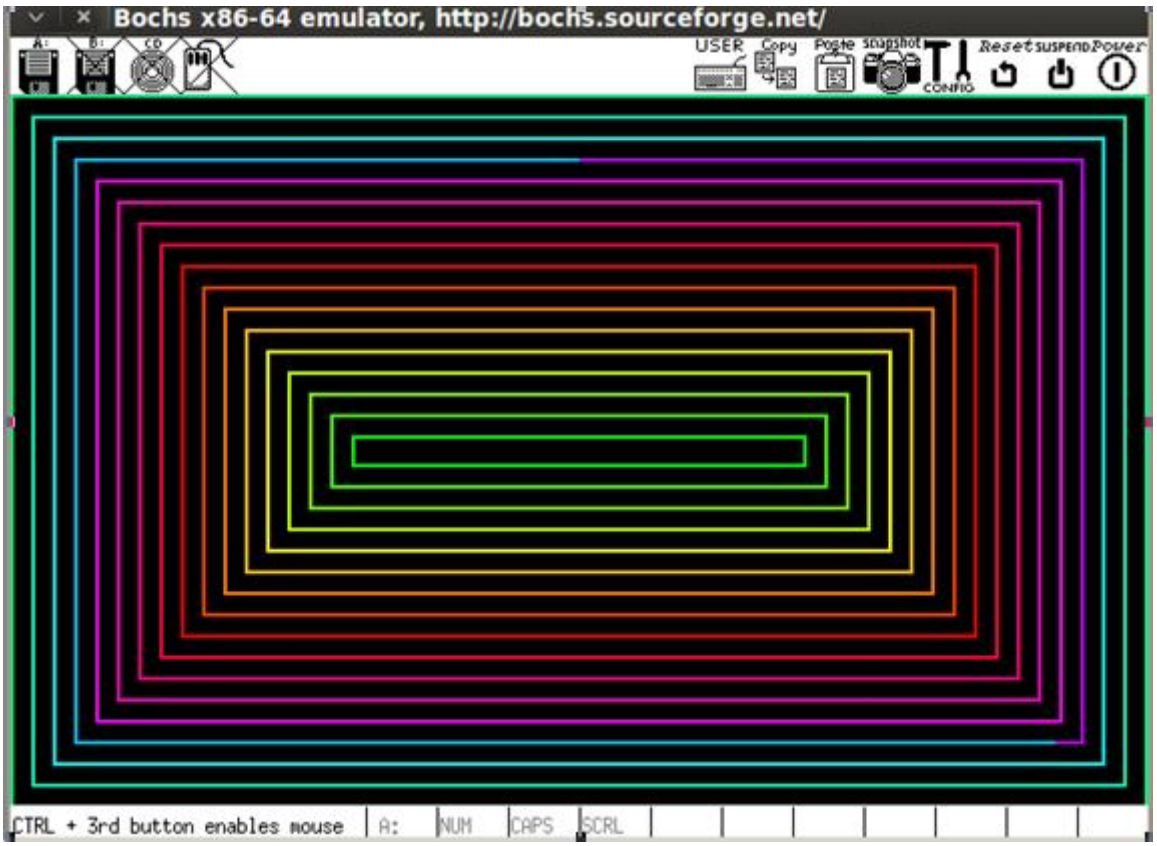
现在将上面的代码保存为test5.c，然后按照修改输入源文件名给出的编译说明，按照给出的说明将编译后的代码复制到软盘的引导扇区。

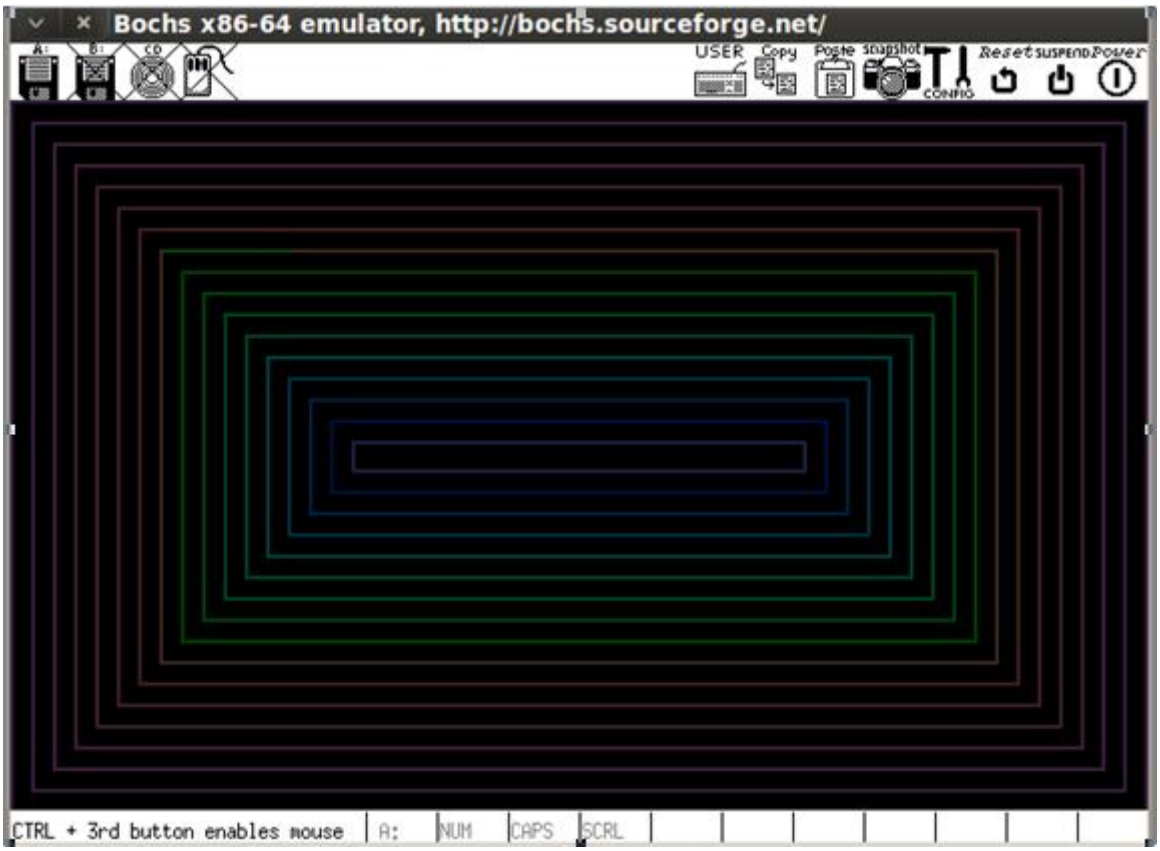
现在观察结果。如果一切正常，您应该会看到以下屏幕输出。

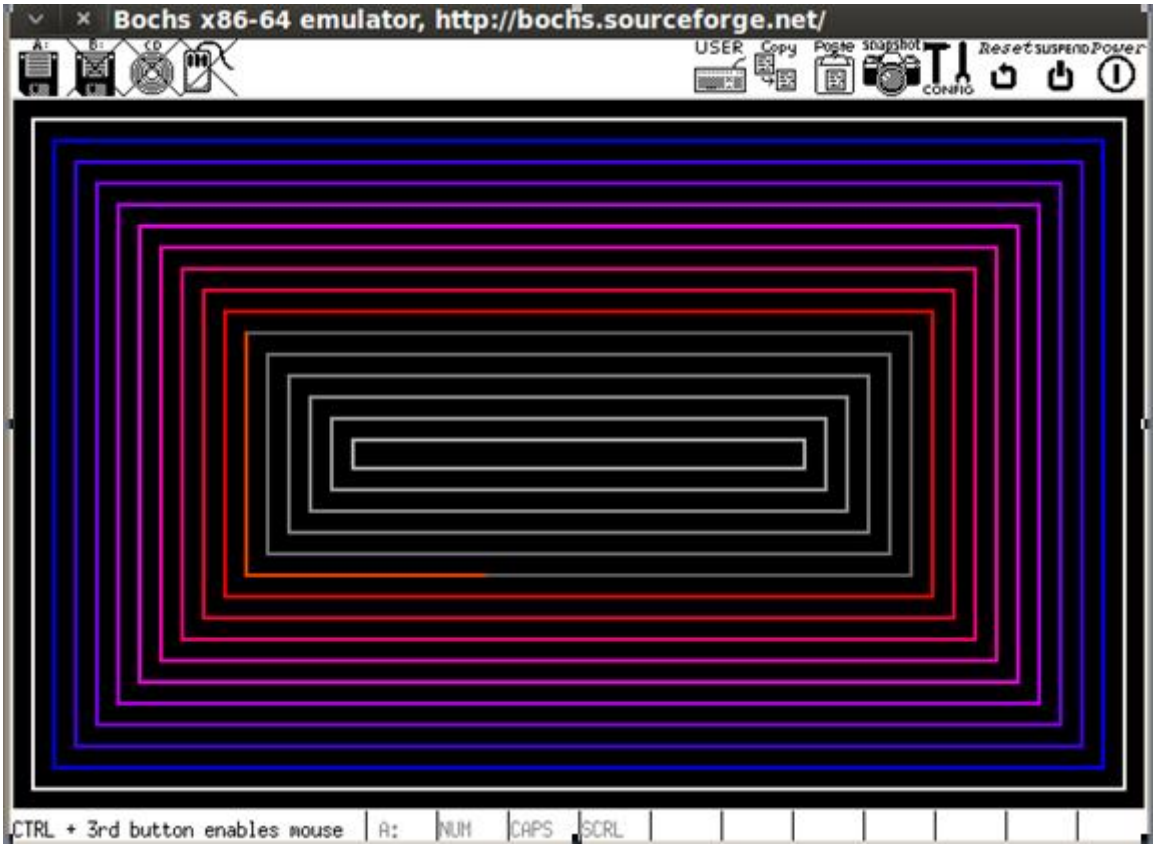


现在按下一个键，看看会发生什么。




Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER
Copy
Paste
Snapshot
CONFIG
Reset
Suspend
Power
CTRL + 3rd button enables mouse
A:
NUM
CAPS
SCRL


Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER
Copy
Paste
Snapshot
CONFIG
Reset
Suspend
Power
CTRL + 3rd button enables mouse
A:
NUM
CAPS
SCRL



观察：

如果您仔细查看可执行文件的内容，您会发现我们的空间几乎用完了。由于引导扇区只有 512 字节，我们只能在我们的程序中嵌入很少的功能，例如初始化环境，然后打印彩色矩形，但不能更多，因为它需要超过 512 字节的空间。下面是截图供大家参考。

这就是本文的全部内容。玩得开心并编写更多程序来探索实模式，您会发现使用 bios 中断在实模式下编程是非常有趣的。在下一篇文章中，我将尝试解释用于访问数据的寻址模式、读取软盘及其架构，以及为什么引导加载程序主要用汇编而不是 C 编写，以及用 C 编写引导加载程序的限制是什么代码生成条款:)

执照

分享

关于作者

阿什奇兰·巴特
软件开发人员
美国
手表该会员

Ashakiran 来自印度海得拉巴，目前在美国担任软件工程师。他是一名业余程序员，喜欢编写代码。

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

关于软盘
Member 14761267 17-Mar-20 14:30

我的5票
coeus 1-Aug-19 12:53

谢谢！
Derek James Sisco 4-Feb-19 22:49

无法编译 test.s
Member 14130364 27-Jan-19 18:55

运行 test4.c 时出错只打印单个 S
Member 13367529 2-Nov-17 18:37

Re: Error running test4.c only prints single S
Member 14496679 17-Jun-19 9:44

Graphics code doesn't run properly
Member 13335272 29-Jul-17 20:39

Re: Graphics code doesn't run properly
Akarsh Gupta 10-Nov-17 17:37

Have problems with the linker
Member 12896585 10-Dec-16 13:51

"int $0x10" : : "a"(0x0e00 | *pStr), "b"(0x0007)
Member 11657648 2-May-15 2:19

Re: "int $0x10" : : "a"(0x0e00 | *pStr), "b"(0x0007)
ecoae2 4-Feb-16 1:32

function getch
Member 11657648 2-May-15 2:16

My vote of 5
Midnight489 22-Apr-15 2:49

Running it on hardware
adnrv 3-Apr-15 3:01

Test4.c - Runtime Error
RoNaK gOhel 8-Dec-14 15:47

Re: Test4.c - Runtime Error
Manoj Sharma 11-Apr-16 22:39

Re: Test4.c - Runtime Error
Сергей Солнцев 29-Dec-16 20:03

Re: Test4.c - Runtime Error
Сергей Солнцев 29-Dec-16 21:13

compile time error
Member 10893260 29-Aug-14 3:34

Re: compile time error
AshakiranBhatter 1-Sep-14 6:11

Re: compile time error
Manoj Sharma 11-Apr-16 21:43

Re: 编译时错误
Member 13367529 2-Nov-17 18:44

不错，但是...
Florian Schneidereit 2-Aug-14 16:56

回复：很好，但是...
AshakiranBhatter 1-Sep-14 6:11

链接错误
Member 10289316 3-Apr-14 17:36

刷新 1 2 3 4 下一页 ▷