

**学生姓名 王灏洋**

**学 号 8208181125**

**专业班级 物联网1802**

**指导教师 胡小龙**

**学 院 计算机学院**

**完成时间 2020年6月**

操作系统原理

实 验 报 告

**目录：**

[一.实验题目 3](#_Toc42095877)

[二．Boot相关概念 3](#_Toc42095878)

[1. Boot引导程序之前 3](#_Toc42095879)

[2. BIOS引导原理 3](#_Toc42095880)

[3. FAT12文件系统 4](#_Toc42095881)

[4. 将会用到的BIOS中断服务程序 5](#_Toc42095882)

[三．Boot开发实验过程 8](#_Toc42095883)

[1. 使用简易Boot 8](#_Toc42095884)

[2. 使用完整版Boot 10](#_Toc42095885)

[四．开发Loader程序 17](#_Toc42095886)

[1. 实现测试版Loader引导程序 17](#_Toc42095887)

[五．调试结果 19](#_Toc42095888)

[1. 使用简易Boot，没有Loader时 19](#_Toc42095889)

[2. 使用可以加载Loader的Boot，而没有Loader时。 20](#_Toc42095890)

[3. 使用可以加载Loader的完整版Boot，有可以加载的Loader时。 21](#_Toc42095891)

[六.心得体会 22](#_Toc42095892)

# 一.实验题目

这次操作系统原理的题目，我选择的是：在Bochs上开发一个简易的BootLoader引导程序。BootLoader引导程序由Boot引导程序和Loader引导程序组成，其中boot引导程序将会在开机时启动，然后用于加载Loader程序，进而Loader引导加载程序负责配置完成配置硬件的工作环境，随后引导加载并启动Kernel。本次实验的基础任务是实现Boot程序，进阶实现Loader程序，并模拟内核加载情形。

# 二．Boot相关概念

## 1. Boot引导程序之前

计算机在上电启动后，首先需要进行的，便是一种自检过程，我们将这个过程称为称为BIOS上电自检，在这个过程中，BIOS程序将会检测硬件设备是否存在问题。如果BIOS检查后没有问题，将会按照启动顺序来选择进行引导的设备。BIOS支持的启动设备有软盘，U盘，硬盘，网络等。本次实验选择软盘作为启动项。

## 2. BIOS引导原理

BIOS自检结束后，会根据设置的启动项来选择启动设备，在本次实验的软盘情况下，将会从软盘的第0磁头第0磁道第1扇区开始检测。我们知道，Boot Sertor的标志便是结尾的两个字节分别是0x55和0xaa，所以在进行检测时，也会以该标准来进行检测——第1扇区是否以0x55和0xaa两个字节作为结尾。如果是的话，BIOS就可以确定，这个扇区是一个Boot Sertor（引导扇区），随后将该扇区的数据复制到物理内存地址的0x7c00（此时还是实模式，没有虚拟内存一说），随后将控制权转交给这段程序，也就是设置PC（程序计数器），在实验的x86下应该是CS和IP，并跳转到该处进行执行。

本次实验采用的3.5英寸的1.44MB的软盘，在这种情况下，一个扇区的大小为512B，BIOS需要做的，就是将这一个扇区的数据从软盘拷贝到物理内存中。显然，512B的引导扇区不能装下操作系统，就算是获取硬件信息的程序都装不下，所以我们只能使用Boot作为一个中介，然后使用这个“中介”来将功能更为强大的引导程序Loader加载都内存中。因为Boot是BIOS固定加载的程序，BIOS是由硬件自动执行的，而Loader是由Boot加载的，所以，我们可以将这个过程看作为计算机硬件向计算机软件移交控制权。

在BIOS向引导程序移交控制权之前，BIOS会对处理器进行初始化，这就包括将代码段寄存器CS和指令指针寄存器IP寄存器设置为我们之前加载引导扇区的位置，即CS和IP内的值分别为0x0000和0x7c00。实际的地址就是经过段地址左移四位后加上偏移地址得到，因为目前还是处于实模式，所以不需要经过地址的转化，得到现在所指向的物理地址便是0x7c00。

## 3. FAT12文件系统

通过上文的可知，引导扇区的可使用大小为510B，因为一个扇区的大小为512

B，还需要2个字节存储空间来装载0x55和0xaa作为引导程序的结束标示数据，所以引导扇区中能用的就只有512B – 2B = 510B。在510B的空间下写出加载Loader的汇编程序大小是够用了，但是要如何来存储Loader程序呢？

目前由两种方法来做这件事情：

* + 1. 采用BIOS加载Boot一样的方法：直接将Loader程序写入到固定的扇区，随后，Boot直接到这个固定的位置来加载Loader程序即可。不仅如此，使用这种方法，我们之后也可以这样来加载内核程序了——直接到固定的位置来加载内核代码到物理内存。这样做的好处是一目了然的：Boot引导程序的加载代码较为容易就可以实现，我们只需要明确Loader引导程序的磁头号，磁道号，扇区号和Loader程序所占的扇区数即可顺利加载Loader引导程序，就像BIOS可以在固定的位置顺利加载Boot引导程序一样。

就算是Loader的大小大于一个扇区的大小，我们也不需要担心——如果Loader引导程序的大小大于一个扇区的话，就将其分散到多个扇区中进程存储，如果这些扇区是连续的，那就更不用担心了，所占扇区数的提供就可以解决这个问题。但是如果这些扇区不是连续的呢？其实也需要过多担心，我们只需要在第一个扇区中标明下一个扇区的磁头号，磁道号，扇区号和Loader程序所占的扇区数即可，这样一来，我们就可以像数据结构中链表的访问方式一样，来访问多个扇区即可。不仅如此，使用这种方法，我们甚至可以使用Loader将Kernel也加载到内存中来。

但是，虽然这种方式的优点很多，缺点也是不少的，比如，随着程序的代码量不断增加，Loader和Kernel对扇区的需求数量也会逐步上升，我们每次对Loader和Kernel作出修改后，就需要重新计算起始扇区和所需扇区的大小，随后向存储介质中中写入Loader和Kernel，随着修改次数的增加，多次写入存储介质变得越来越不方便。我们在调试程序的时候，可能需要添加多个内容，这时，我们势必会对Boot引导和Loader程序作出大量的修改，后面在添加功能时，也需要费心的来计算这些繁琐的算式，十分不方便。

* + 1. 我们与其每次都调整Boot程序和Loader程序，不如为软盘创建一个简单的文件系统，方便后续的使用。简单的文件系统并不难创建，我们选择较为容易实现的FAT12软盘型文件系统，实现该文件系统，只需要一些简单的逻辑。

## 4. 将会用到的BIOS中断服务程序

在实验中，将使用到BIOS中断服务程序INT 10h的主功能编号有06h，02h和13h，他们的功能及参数说明如下：

* 重置磁盘驱动器

BIOS中断服务程序INT 10h的主功能编号AH = 00h可以实现软盘驱动器的复位，即重新初始化一次软盘驱动器，将软盘驱动器的磁头移动到默认位置。

INT 13h， AH = 00h功能：重置软盘驱动器，为下一次读写软盘做准备。

1. DL = 驱动器号， 00h ～ 7fh： 软盘； 80h ～ 0ffh：硬盘。
   1. DL = 00h代表第一个软盘驱动器；
   2. DL = 01h代表第二个软盘驱动器；
   3. DL = 80h代表第一个硬盘驱动器；
   4. DL = 81h代表第二个软盘驱动器。

* 软盘扇区的读取操作

BIOS中断服务程序INT 10h的主功能编号AH = 02h实现软盘扇区的读取操作。

INT 13h， AH = 02h功能：读取磁盘扇区。

1. AL = 读入的扇区数；
2. CH = 磁道号的低8位；
3. CL = 扇区号1～63（bit 0～5），磁道号（柱面号）的高2位；
4. DH = 磁头号；
5. DL = 驱动器号；
6. ES：BX = 数据缓冲区。

* 设置屏幕光标位置

BIOS中断服务程序INT 10h的主功能号AH=02h可以实现屏幕光标位置的设置功能，具体的寄存器参数说明如下：

INT 10h，AH = 02h功能：设定光标位置。

1. DH = 游标的列数；
2. DL = 游标的行数；
3. BH = 页码。

使用方式：设置屏幕的光标位置在屏幕的左上角（0，0）处。为了实现该功能，我们设置DH = DL = 0.因为屏幕的列坐标0点和行坐标0点都在屏幕的左上角，屏幕的X轴和Y轴分别向左向下延伸，即屏幕的原点在屏幕的左上角。

* 上卷制定范围的窗口（用于清屏）

BIOS中断服务程序INT 10h的主功能号AH=06h可以实现按指定范围滚动窗口的功能，该功能可以被用于清屏，具体的寄存器参数说明如下：

INT 10h，AH = 06h功能：按指定范围滚动窗口。

1. AL = 滚动的列数，若为0则实现清空屏幕的功能；
2. BH = 滚动后空出位置放入的属性；
3. CH = 滚动范围的左上角坐标列号；
4. CL = 滚动范围的左上角坐标行号；
5. DH = 滚动范围的右下角坐标列号；
6. DL = 滚动范围的右下角坐标行号；
7. BH = 颜色属性。
   1. Bit 0～2:字体颜色
   2. Bit 3：字体亮度
   3. Bit 4～6: 背景颜色
   4. Bit 7: 字体是否闪烁

该命令主要用户按指定范围滚动窗口，因为我们希望使用该调用来清屏，所以我们主要使用的是AL = 0时的功能。当AL = 0时，BX， CX， DX寄存器中的值都将不起作用。

* 显示字符

BIOS中断服务程序INT 10h的主功能号AH=0Eh是在屏幕上显示一个字符。

INT 10h，AH = 0Eh功能：在屏幕上显示一个字符

1. AL = 待显示字符；
2. BL = 前景色。

* 显示字符串

BIOS中断服务程序INT 10h的主功能号AH=13h可以实现字符串的显示功能，具体的寄存器参数说明如下：

INT 10h， AH = 13h，功能：显示一行字符串。

1. AL = 写入模式。
   1. AL = 00h：字符串的属性由BL寄存器提供，而CX寄存器提供字符串长度（B为单位），显示后光标位置不变，即显示前的光标位置。
   2. AL = 01h：同AL = 00h，但光标会移动到字符串尾端位置。
   3. AL = 02h：字符串属性由每个字符后面紧跟的字节提供，故CX寄存器提供的字符串长度改成以Word为单位，显示后光标位置不变。
   4. AL = 03h：同AL = 02h，但是显示后光标会移动到字符串尾端位置。
2. CX = 字符串的长度；
3. DH = 游标的坐标行行号；
4. DL = 游标的坐标列号；
5. DS：BP = 需要显示的字符串存放在内存中的位置；
6. BH = 页码；
7. BL = 字符属性/颜色属性。
   1. Bit 0～2:字体颜色
   2. Bit 3：字体亮度
   3. Bit 4～6: 背景颜色
   4. Bit 7: 字体是否闪烁

字符串的显示功能，在本次实验中使用较为频繁。使用该功能不仅可以显示

字符串，设定字体的前景色和背景色，还可以设置待显示字符串的坐标位置。该功能比较适合显示不同的日志信息。

# 三Loader原理

Loader引导加载程序负责检测硬件信息，完成处理器模式的转换，像内核传递数据三部分工作，之所以做这些工作是为了给内核的初始化提供信息及功能支持，以便完成内核的初始化工作后能够正常运行。

## 1. 检测硬件信息

Loader引导程序需要检测很多硬件的信息，而这种检测主要是通过BIOS中断服务程序来获取和检测硬件的信息。但是，因为BIOS在上电自检时得到的信息大部分都只能在实模式下获取，而内核又不运行在实模式下，所以我们需要在进入内核程序之前就将这些信息进行检测，然后将他们作为参数提供给内核来使用。

而在要提供内核的信息中，最为重要的是物理地址空间信息，只有正确解析出物理地址空间信息，我们才可以知道ROM，RAM，设备寄存器空间呃内存空洞等资源的物理地址范围，只有知道了这些资源的正确地址，我们才能将他们交给内存管理单元模块来进行维护，才有可能进行后面复杂的操作。

## 2. 处理器模式的切换

在BIOS的时候，处理器是运行在实模式下的，所有的信息都是可以直接通过地址来进行获取的，但是32位操作系统使用的保护模式，再到现在的64位操作系统使用的IA-32e（长模式），Loader引导加载程序必须确保这个转换的过程可以顺利的完成，所以在Lodaer引导程序中，必须要经历这个三个模式，才能使处理器也运行中64位IA-32e模式。在各个模式的切换过程中，Loader引导程序必须手动创建各运行模式的临时数据，然后按照标准执行模式间的跳转。这其中有配置系统临时页表的工作，这个工作的主要目的是根据各个阶段的页表特性设置临时页表项，还要保证页表覆盖的地址空间满足程序的使用要求。

## 3. 向内核传递数据

Loader引导加载程序可以向内核传递两类数据，一类是控制信息，另一类是硬件数据信息。这些数据一方面控制内核程序的执行流程，另一方面为内核程序的初始化提供数据信息支持。

* 控制信息

控制信息一般用于控制内核执行流程或者限制内核的某些功能。这些数据（参数）是与内核程序商定好的协议，属于纯软件控制逻辑，例如启动模式（命令行界面或者是图形界面），启动方式（从何处读取内核信息），终端重定向（串口或是显示器）等信息。

* 硬件数据信息

正如前面所述，硬件数据信息是指Loader引导加载程序检测出的硬件数据信息。Loader引导程序将这些数据信息保存在固定的内存地址中，并将数据起始内存和数据长度作为参数传递给内核，以供内核程序在初始化时分析，配置和使用。

Loader引导程序的内容过于庞杂，同时，也出于时间的紧迫，只能实现一个用于测试Boot引导程序是否正确的，关于上述原理而实现的Loader，会在暑假期间进行实践。

# 四．Boot开发实验过程

## 1. 使用简易Boot

首先使用org 0x7c00设置起始地址。如果没有使用该指令，那么编译器将会使用0x0000作为程序的起始地址。程序的起始地址将会影响绝对地址寻址指令，而我们在使用BIOS向Boot移交控制权时，就是使用绝对地址寻址的方式，如果不确定地址，就会造成编译得出的地址是BIOS找不到的。

代码中的标识符BaseOfStack用于为栈指针寄存器SP提供栈基地址。

以下是Boot引导程序的主体代码，主要是用BIOS中断服务程序INT 10h实现屏幕信息显示相关操作。

org 0x7c00 *;设定引导起始地址*

BaseOfStack equ 0x7c00 *;设定常量BaseOfstack为0x7c00*

Label\_Start:

mov **ax**, **cs**

mov **ds**, **ax**

mov **es**, **ax**

mov **ss**, **ax**

mov **sp**, BaseOfStack *;栈顶设置为0x7c00*

*;======= clear screen*

mov **ax**, 0600h *;ah=06h时，int 10h中断设置滚动窗口的功能， al=0则实现清屏*

mov **bx**, 0700h

mov **cx**, 0

mov **dx**, 0184fh

int 10h

*;======= set focus*

mov **ax**, 0200h *;ah=02h时，int 10h中断为设置光标*

mov **bx**, 0000h

mov **dx**, 0000h

int 10h

*;======= display on screen : Start Booting......*

mov **ax**, 1301h *;ah=13h时，int 10h中断为显示一行字符串*

mov **bx**, 000fh

mov **dx**, 0000h

mov **cx**, 10

push **ax**

mov **ax**, **ds**

mov **es**, **ax**

pop **ax**

mov **bp**, StartBootMessage

int 10h

*;======= reset floppy*

xor **ah**, **ah**

xor **dl**, **dl**

int 13h

jmp $

StartBootMessage: db "Start Boot"

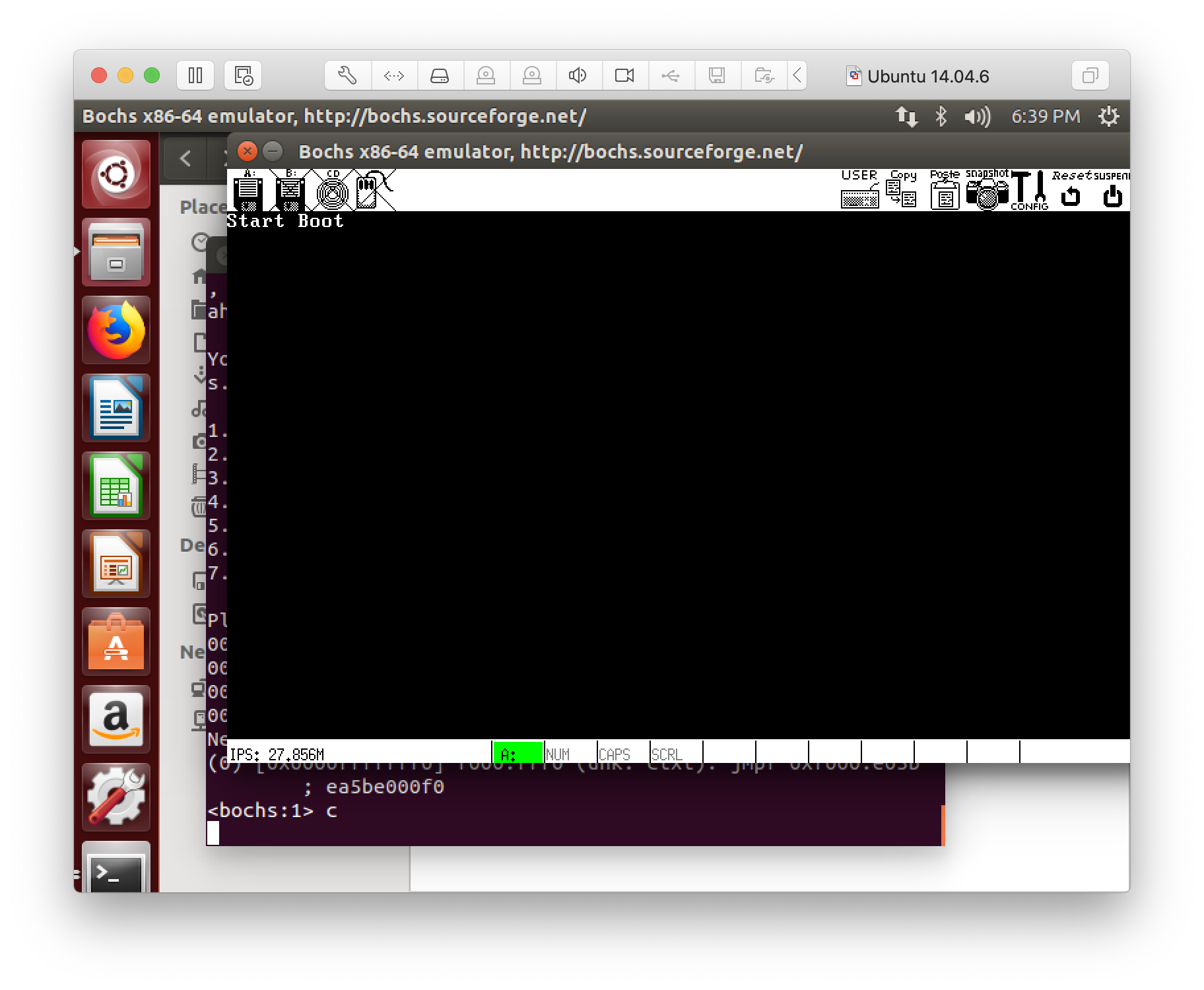
*;======= fill zero until whole sector*

times 510 - ($ - $$) db 0 *;不够512字节的用0填充*

dw 0xaa55

汇编代码 times 510 - （$ - $$）db 0中，$ - $$标明将当前行被编译后的地址（机器码地址）减去本节程序的起始地址，在此处义为计算出当前程序生成的机器码长度，随后将510 - （$ - $$）长度填充为0。此处为通过times伪指令填充引导扇区剩余空间，保障生成文件大小为512B。

接下来通过bochs虚拟机创建boot.img，再将上述汇编代码生成的二进制文件写入磁盘镜像文件。便得到如下结果。



此时可以看出，仅有软盘A在线，其他的软盘B，光驱，键盘等都未连接，这正是我们设置的软盘启动。

将软盘格式化为FAT12文件系统的过程中，FAT类文件系统会对软盘中的扇区进行处理，将软盘划分为引导扇区，FAT表，根目录区和数据区。

在加载Loader程序后，在Label\_File\_Loaded中移交控制权，即跳转到物理地址0x10000处执行loader.bin。以下是boot.asm的完整代码。

## 2. 使用完整版Boot

Boot.asm

org 0x7c00

BaseOfStack equ 0x7c00

BaseOfLoader equ 0x1000

OffsetOfLoader equ 0x00

RootDirSectors equ 14

SectorNumOfRootDirStart equ 19

SectorNumOfFAT1Start equ 1

SectorBalance equ 17

jmp **short** Label\_Start

nop

BS\_OEMName db 'MINEboot'

BPB\_BytesPerSec dw 512

BPB\_SecPerClus db 1

BPB\_RsvdSecCnt dw 1

BPB\_NumFATs db 2

BPB\_RootEntCnt dw 224

BPB\_TotSec16 dw 2880

BPB\_Media db 0xf0

BPB\_FATSz16 dw 9

BPB\_SecPerTrk dw 18

BPB\_NumHeads dw 2

BPB\_HiddSec dd 0

BPB\_TotSec32 dd 0

BS\_DrvNum db 0

BS\_Reserved1 db 0

BS\_BootSig db 0x29

BS\_VolID dd 0

BS\_VolLab db 'boot loader'

BS\_FileSysType db 'FAT12 '

Label\_Start:

mov **ax**, **cs**

mov **ds**, **ax**

mov **es**, **ax**

mov **ss**, **ax**

mov **sp**, BaseOfStack

*;======= clear screen*

mov **ax**, 0600h

mov **bx**, 0700h

mov **cx**, 0

mov **dx**, 0184fh

int 10h

*;======= set focus*

mov **ax**, 0200h

mov **bx**, 0000h

mov **dx**, 0000h

int 10h

*;======= display on screen : Start Booting......*

mov **ax**, 1301h

mov **bx**, 000fh

mov **dx**, 0000h

mov **cx**, 10

push **ax**

mov **ax**, **ds**

mov **es**, **ax**

pop **ax**

mov **bp**, StartBootMessage

int 10h

*;======= reset floppy*

xor **ah**, **ah**

xor **dl**, **dl**

int 13h

*;======= search loader.bin*

mov word [SectorNo], SectorNumOfRootDirStart

Lable\_Search\_In\_Root\_Dir\_Begin:

cmp word [RootDirSizeForLoop], 0

jz Label\_No\_LoaderBin

dec word [RootDirSizeForLoop]

mov **ax**, 00h

mov **es**, **ax**

mov **bx**, 8000h

mov **ax**, [SectorNo]

mov **cl**, 1

call Func\_ReadOneSector

mov **si**, LoaderFileName

mov **di**, 8000h

cld

mov **dx**, 10h

Label\_Search\_For\_LoaderBin:

cmp **dx**, 0

jz Label\_Goto\_Next\_Sector\_In\_Root\_Dir

dec **dx**

mov **cx**, 11

Label\_Cmp\_FileName:

cmp **cx**, 0

jz Label\_FileName\_Found

dec **cx**

lodsb

cmp **al**, byte [**es**:**di**]

jz Label\_Go\_On

jmp Label\_Different

Label\_Go\_On:

inc **di**

jmp Label\_Cmp\_FileName

Label\_Different:

and **di**, 0ffe0h

add **di**, 20h

mov **si**, LoaderFileName

jmp Label\_Search\_For\_LoaderBin

Label\_Goto\_Next\_Sector\_In\_Root\_Dir:

add word [SectorNo], 1

jmp Lable\_Search\_In\_Root\_Dir\_Begin

*;======= display on screen : ERROR:No LOADER Found*

Label\_No\_LoaderBin:

mov **ax**, 1301h

mov **bx**, 008ch

mov **dx**, 0100h

mov **cx**, 21

push **ax**

mov **ax**, **ds**

mov **es**, **ax**

pop **ax**

mov **bp**, NoLoaderMessage

int 10h

jmp BaseOfLoader:OffsetOfLoader

*;======= found loader.bin name in root director struct*

Label\_FileName\_Found:

mov **ax**, RootDirSectors

and **di**, 0ffe0h

add **di**, 01ah

mov **cx**, word [**es**:**di**]

push **cx**

add **cx**, **ax**

add **cx**, SectorBalance

mov **ax**, BaseOfLoader

mov **es**, **ax**

mov **bx**, OffsetOfLoader

mov **ax**, **cx**

Label\_Go\_On\_Loading\_File:

push **ax**

push **bx**

mov **ah**, 0eh

mov **al**, '.'

mov **bl**, 0fh

int 10h

pop **bx**

pop **ax**

mov **cl**, 1

call Func\_ReadOneSector

pop **ax**

call Func\_GetFATEntry

cmp **ax**, 0fffh

jz Label\_File\_Loaded

push **ax**

mov **dx**, RootDirSectors

add **ax**, **dx**

add **ax**, SectorBalance

add **bx**, [BPB\_BytesPerSec]

jmp Label\_Go\_On\_Loading\_File

Label\_File\_Loaded:

jmp BaseOfLoader:OffsetOfLoader

*;======= read one sector from floppy*

*;/\**

*; \*AX = 待读取的磁盘起始扇区号*

*; \*CL = 读取的扇区数量*

*; \*ES: BX = 缓冲区*

*; \*/*

Func\_ReadOneSector:

push **bp**

mov **bp**, **sp**

sub **esp**, 2

mov byte [**bp** - 2], **cl**

push **bx**

mov **bl**, [BPB\_SecPerTrk]

div **bl**

inc **ah**

mov **cl**, **ah**

mov **dh**, **al**

shr **al**, 1

mov **ch**, **al**

and **dh**, 1

pop **bx**

mov **dl**, [BS\_DrvNum]

Label\_Go\_On\_Reading:

mov **ah**, 2

mov **al**, byte [**bp** - 2]

int 13h

jc Label\_Go\_On\_Reading

add **esp**, 2

pop **bp**

ret

*;======= get FAT Entry*

*;AH = FAT表项号*

Func\_GetFATEntry:

push **es**

push **bx**

push **ax**

mov **ax**, 00

mov **es**, **ax**

pop **ax**

mov byte [Odd], 0

mov **bx**, 3

mul **bx**

mov **bx**, 2

div **bx**

cmp **dx**, 0

jz Label\_Even

mov byte [Odd], 1

Label\_Even:

xor **dx**, **dx**

mov **bx**, [BPB\_BytesPerSec]

div **bx**

push **dx**

mov **bx**, 8000h

add **ax**, SectorNumOfFAT1Start

mov **cl**, 2

call Func\_ReadOneSector

pop **dx**

add **bx**, **dx**

mov **ax**, [**es**:**bx**]

cmp byte [Odd], 1

jnz Label\_Even\_2

shr **ax**, 4

Label\_Even\_2:

and **ax**, 0fffh

pop **bx**

pop **es**

ret

*;======= tmp variable*

RootDirSizeForLoop dw RootDirSectors

SectorNo dw 0

Odd db 0

*;======= display messages*

StartBootMessage: db "Start Boot"

NoLoaderMessage: db "ERROR:No LOADER Found"

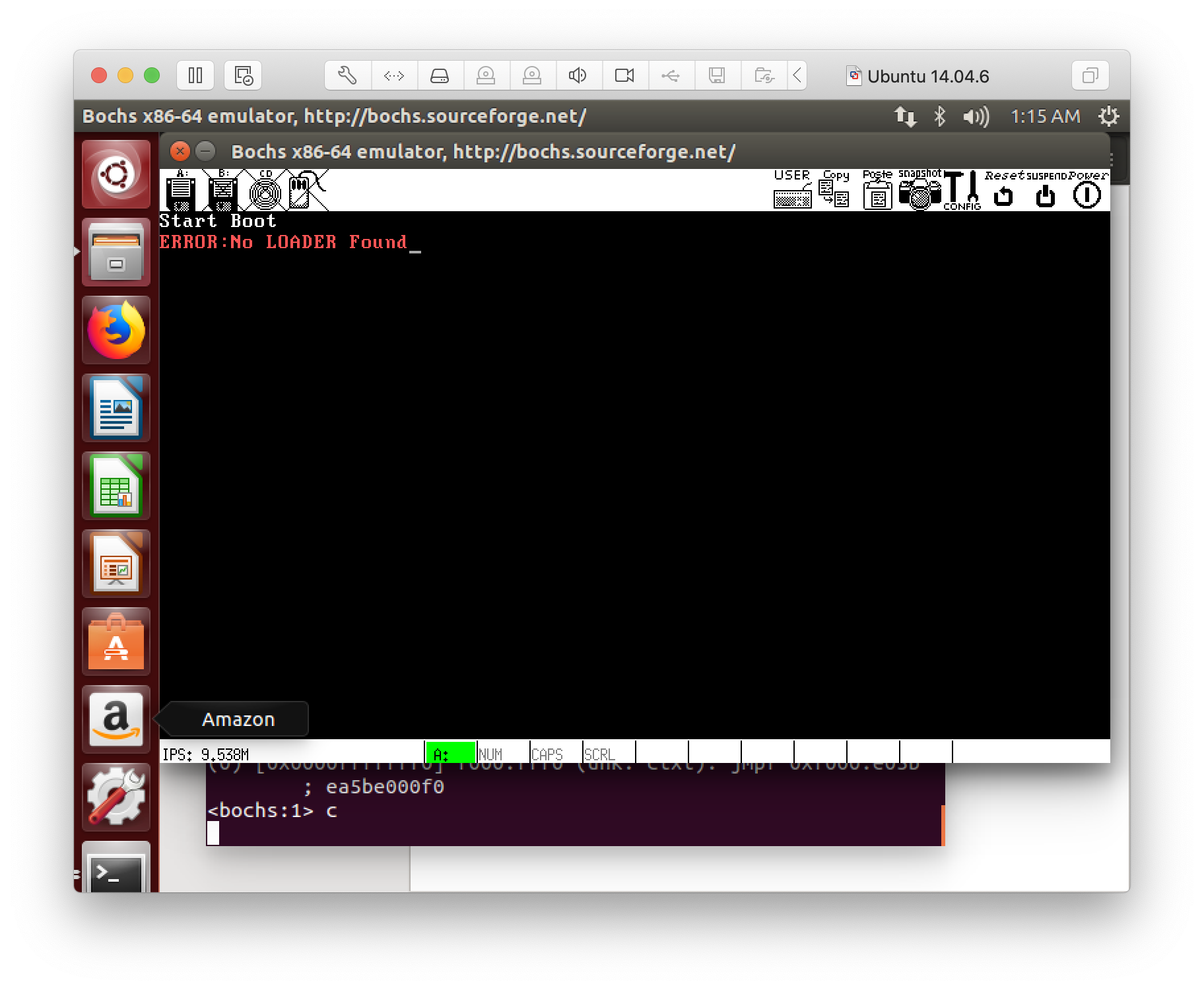
LoaderFileName: db "LOADER BIN",0

*;======= fill zero until whole sector*

times 510 - ($ - $$) db 0

dw 0xaa55

在此时，我们没有加入Loader，故结果如下：



# 五．开发Loader程序

## 1. 实现测试版Loader引导程序

我们在此添加一个用测试的Loader程序，用于测试效果。

Loader\_test.asm

org 10000h

mov **ax**, **cs**

mov **ds**, **ax**

mov **es**, **ax**

mov **ax**, 0x00

mov **ss**, **ax**

mov **sp**, 0x7c00

*;======= display on screen : Start Loader......*

mov **ax**, 1301h

mov **bx**, 000fh

mov **dx**, 0200h

mov **cx**, 12

push **ax**

mov **ax**, **ds**

mov **es**, **ax**

pop **ax**

mov **bp**, StartLoaderMessage

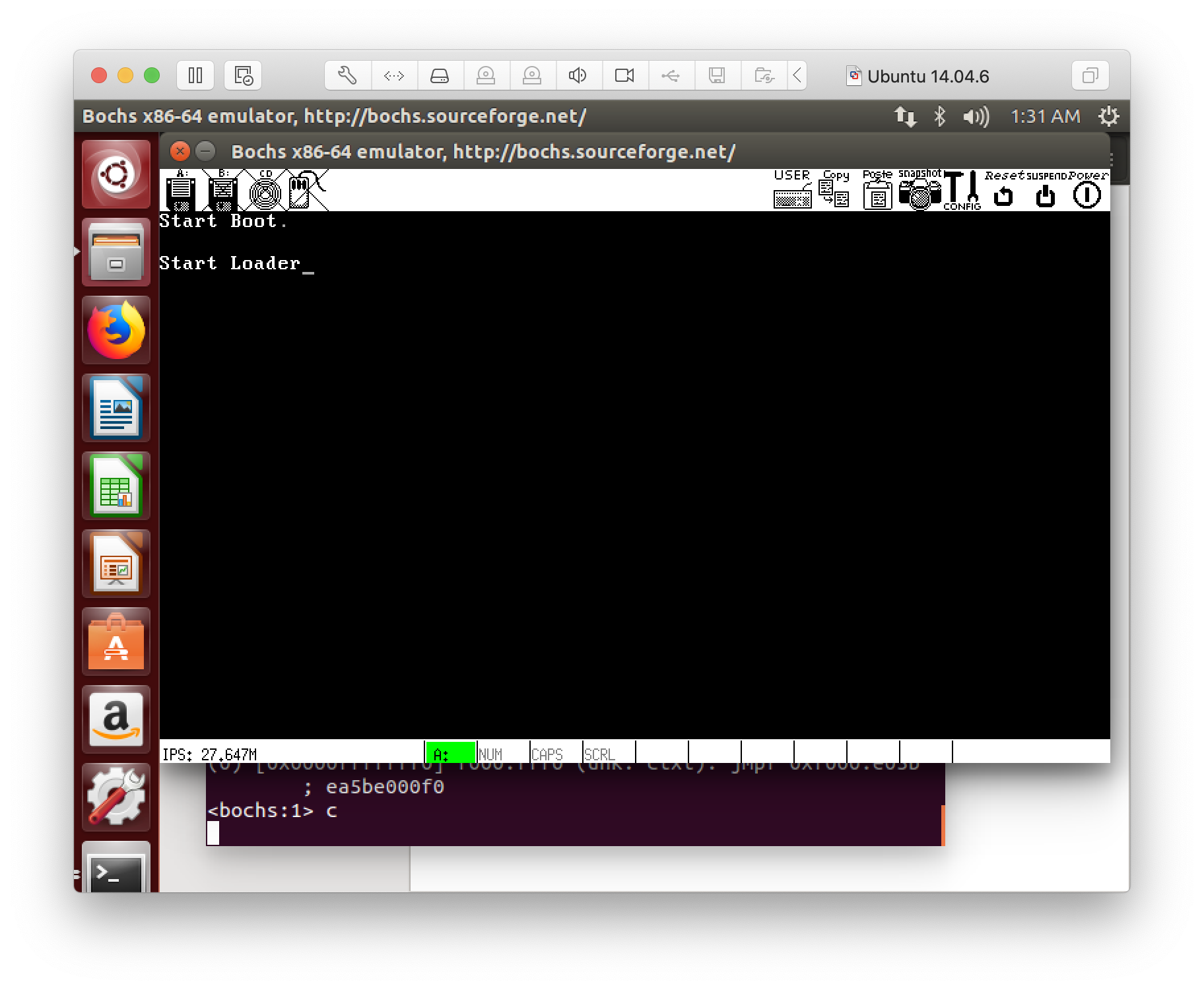
int 10h

jmp $

*;======= display messages*

StartLoaderMessage: db "Start Loader"

因为此时，我们已经使用了FAT12文件系统，将boot.img挂载到/media/下后，我们只需要使用cp将loader.bin复制过去即可。随后运行bochs。

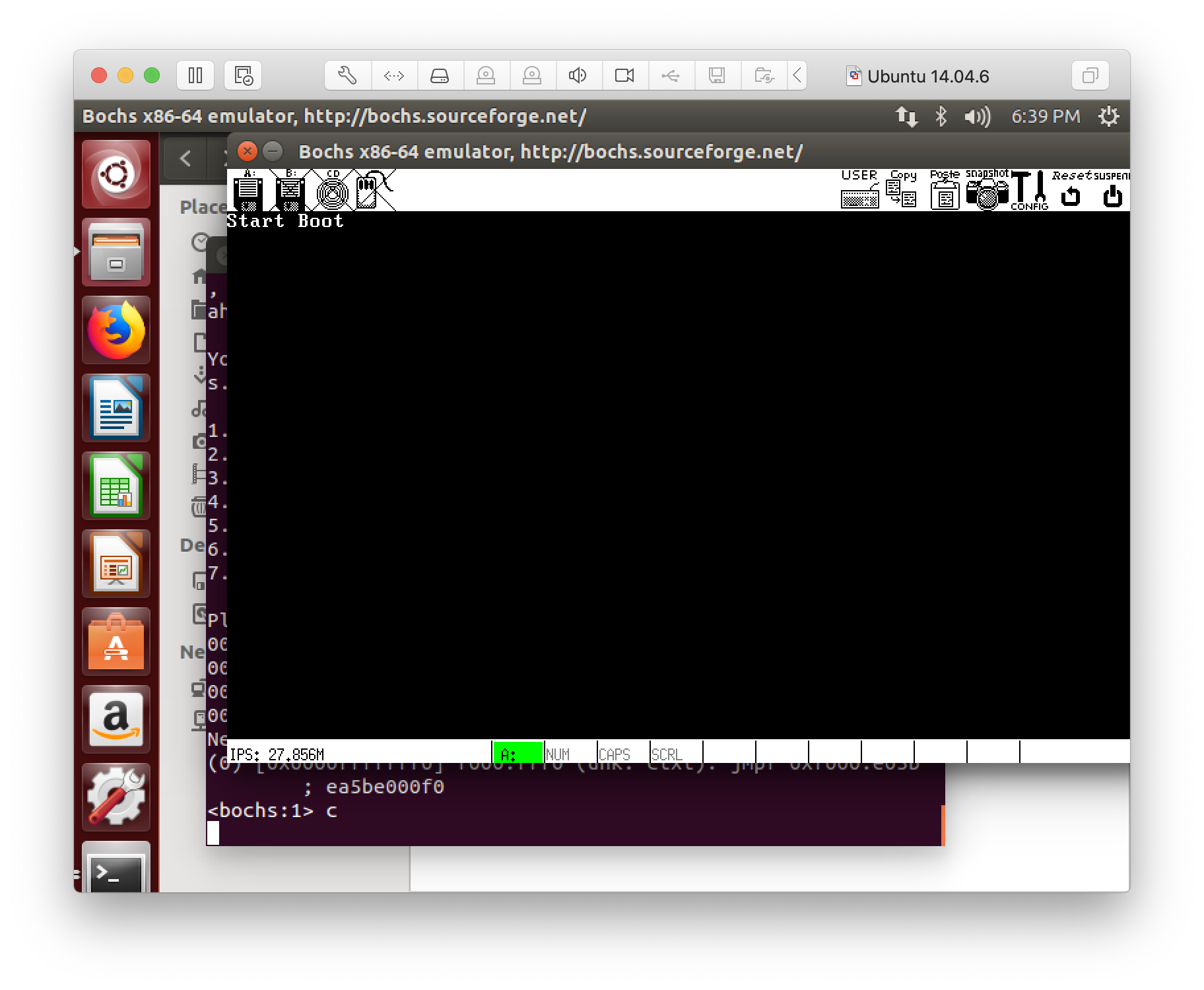


由此可见，Loader加载成功。

# 六．调试结果

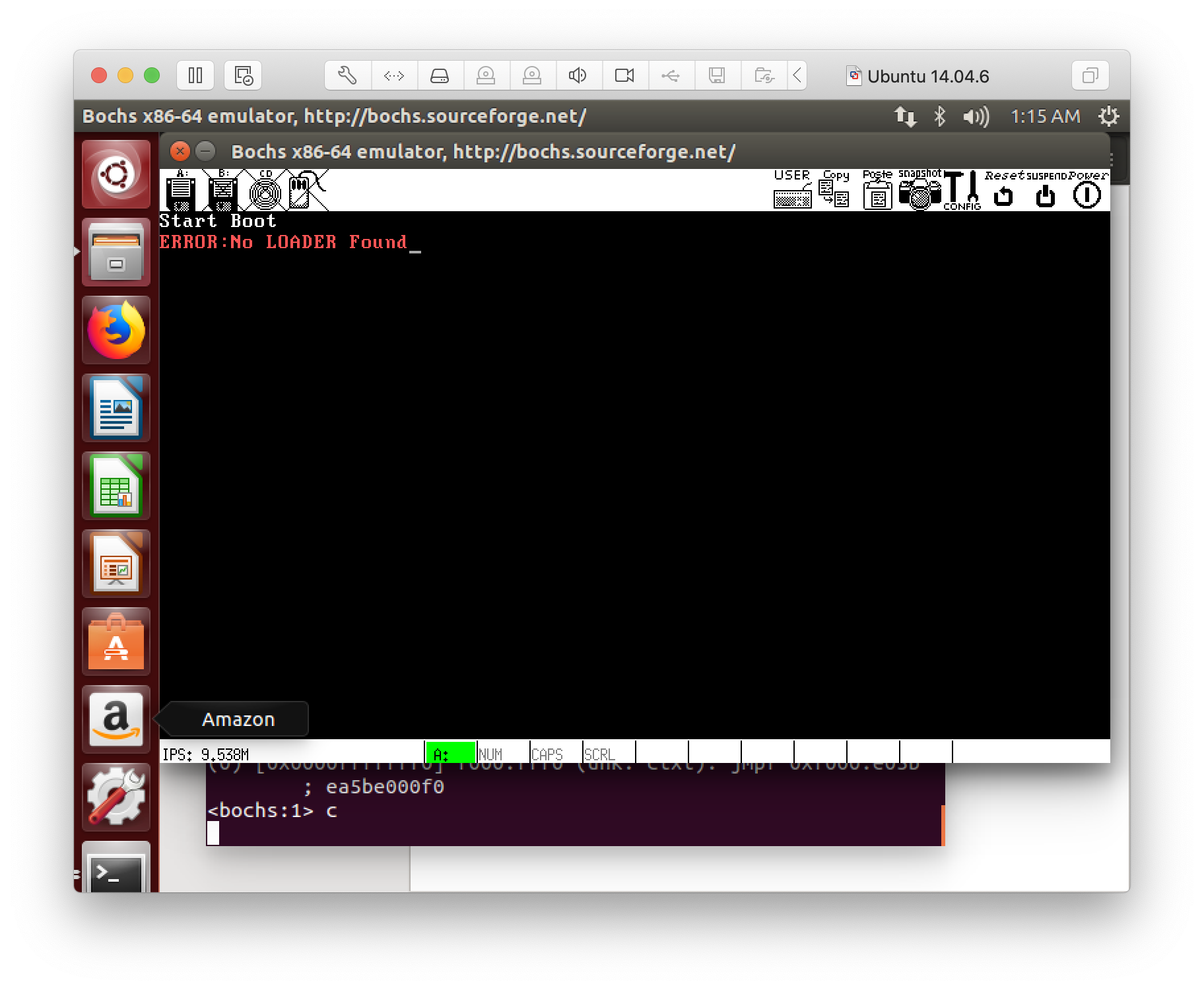
## 1. 使用简易Boot，没有Loader时

使用简易Boot时，可以正确被BIOS加载。



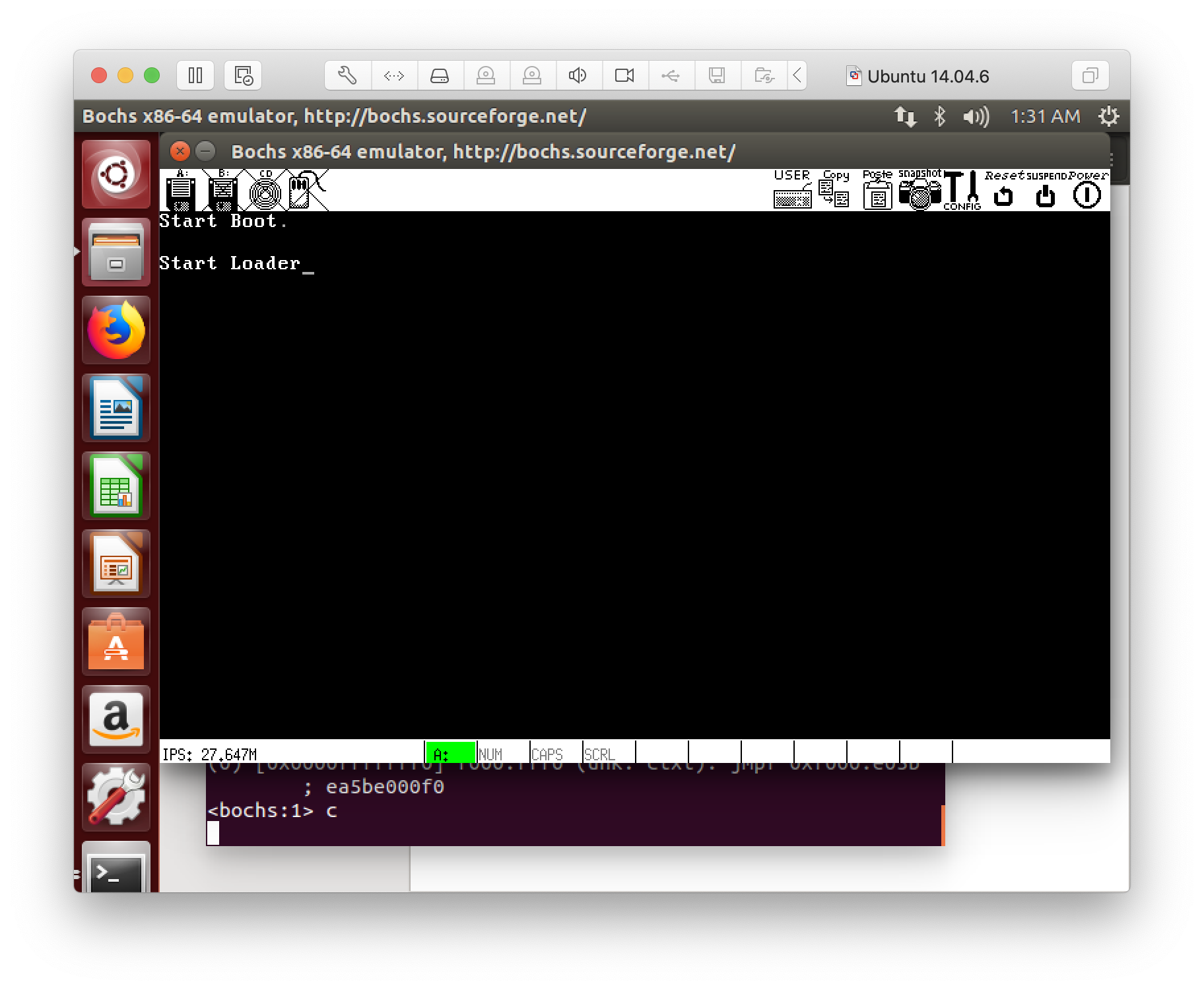
## 2. 使用可以加载Loader的Boot，而没有Loader时。

使用可以加载Loader的完整版的Boot时，因为在FAT12中，没有可以加载的Loader程序，所以显示错误。



## 3. 使用可以加载Loader的完整版Boot，有可以加载的Loader时。

使用可以加载Loader的完整版Boot时，存在可以加载的测试版Loader，所以完成了加载。



# 七.心得体会

本次课程实验中，我将学习到的关于Boot的知识在一定程度上付诸实践，比较可惜的是，确实没有时间来做Loader这一块了，我会尽量在以后的时间中把这些缺陷补上。以前，我写过一些关于BootLoader的文章，那时候只是知道，奥，BootLoader就是用来上电自检，然后把内核拿到内存里面去，之后开始运行就万事大吉了，在系统运行起来后，BootLoader就被抛到一旁了。在这次实验之后，我对BootLoader有了更为深入的了解，虽然这次用的是软盘，但是好歹也满足了下心愿。

最后，非常感谢胡老师对这个题目的认可，让我学到了很多有意思的知识。