Part1

转载注明出处：[http://www.cnblogs.com/lucasysfeng/p/4846119.html](http://www.cnblogs.com/lucasysfeng/p/4846119.html" \t "_blank)

项目地址：[https://github.com/lucasysfeng/lucasOS](https://github.com/lucasysfeng/lucasOS" \t "_blank)

想动手，但不知从何入手，是学习一门新知识普遍会遇到的尴尬点。笔者喜欢边实践边学习理论，笔者的写作思路是：入门的文章要避免讲一些高深的理论，而应该先抛出demo，从研究demo入手，逐步加深demo的难度，从而学习这个过程中涉及到的理论知识。下面就让我们花10分钟写个“操作系统”。

**第一节 开发环境**

　　我们在linux下制作软盘、编译内核等，因此需要linux开发环境。如果你用windows, 那么在windows下安装VMware, 在VMware中安装ubuntu虚拟机，此ubuntu作为开发环境。

　　注：笔者的开发环境是windows--VMware--ubuntu14.04.

**第二节 计算机启动过程**

　　写操作系统看似是一个复杂的过程，但只要我们将过程分解，完成每一步，那么完成一个操作系统就是水到渠成的事了。好了，我们就看一下计算机的启动过程，看操作系统何时被启动的。

**第一步：读取BIOS**

　　按下电源按钮后，计算机首先读取一块ROM芯片，这块芯片里的程序是"基本输入输出系統"（Basic Input/Output System），即BIOS.

**第二步：硬件自检**

　　BIOS会检查计算机硬件是否满足运行条件，如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。

**第三步：启动顺序**

　　硬件检查完成后，BIOS会将控制权交给下一阶段的启动程序，注意，“下一阶段的启动程序”可能存放在硬盘中，也可能存放在CD/DVD中，或者软盘中等等，可以设置BIOS选择从哪个设备启动。

**第四步：主引导记录**

　　BIOS找到了“下一阶段的启动程序”所在设备，会读取该设备的第一个扇区，即读取最前面的512字节，称为主引导记录。主引导记录会告诉计算机下一步到哪里去找操作系统。

**第五步：bootloader**

　　计算机读取"主引导记录"前面446字节的机器码之后，运行事先安装的“启动管理器”bootloader，由用户选择启动哪个操作系统。如果你安装了多个操作系统，那么就要从这步做出选择了。

**第六步：加载内核**

　　好了，选择操作系统（内核）后，会加载内核，下面就交给内核去处理了。

**第三节 主引导记录**

　  我们使用虚拟机来启动操作系统，上面的第一步和第二步我们不做，由虚拟机去完成；第三步“启动顺序”我们选择从软盘启动（我们用镜像代替，并不是真的软盘），需要对虚拟机做下设置，选择从软盘启动。下面重点来看第四步，我们写一下“主引导记录”，让BIOS读取我们写的主引导记录。

**1. 主引导记录代码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | ; 文件名 boot.asm    org 7c00h                     ; BIOS读入MBR后，从0x7c00h处开始执行    ; 下面部分和10h有关中断，10h中断用来显示字符  mov ax, cs  mov es, ax  mov ax, msg  mov bp, ax                    ; ES:BP表示显示字符串的地址  mov cx, msgLen                ; CX存字符长度  mov ax, 1301h                 ; AH=13h表示向TTY显示字符，AL=01h表示显示方式（字符串是否包含显示属性，01h表示不包含）  mov bx, 000fh                 ; BH=00h表示页号，BL=0fh表示颜色  mov dl, 0                     ; 列  int 10h    msg: db "hello world, welcome to OS!"  msgLen: equ $ - msg           ; 字符串长度  times 510 - ($ - $$) db 0     ; 填充剩余部分  dw 0aa55h                     ; 魔数，必须有这两个字节BIOS才确认是MBR |

**2. 编译**

# nasm boot.asm -o boot.bin

 如果没有nasm，安装它 sudo apt-get install nasm, 执行完上述命令，会生成boot.bin文件，这就是我们的主引导记录二进制。

**第四节 代码解释**

 　　我们再来看下主引导记录的汇编代码，熟悉汇编的读者可忽略本节。

**1. 为什么MBR要从0x7c00h处开始执行？**

　　ORG是伪指令，org 7c00h是告诉编译器，下面代码装入到内存的起始地址0x7c00h处。为什么呢，这是因为BIOS读取主引导记录后，会从0x7c00h处开始执行，那么BIOS为什么会从0x7c00h这个地址开始执行，而不是其他地址呢，这一切都要从大明湖畔的8086cpu说起。

　　时光飞逝，容颜易老，8086却还是那个样子，如图所示：

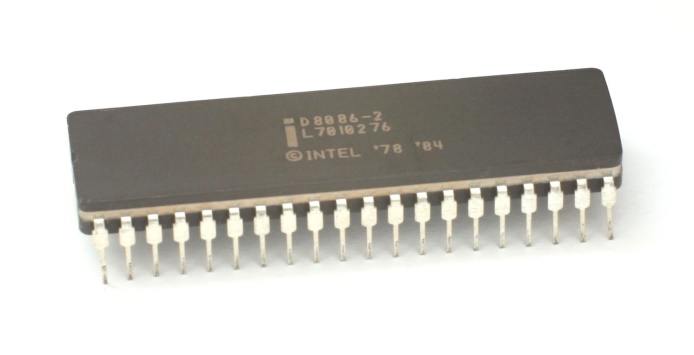


图 8086实物图

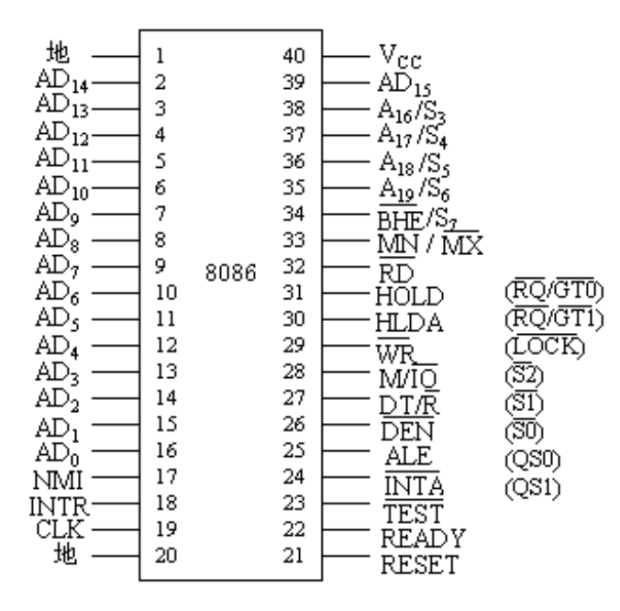


图 8086引脚图

　　正如图中所示，8086cpu的地址总线宽度为20（AD0-AD19），可以传送220的地址信息，即可以定位220（1M）的内存地址空间，那么这1M的内存地址空间是如何分配的呢，见下图所示（图是386的，我们目前只关心实模式即1M内存地址空间分配）：

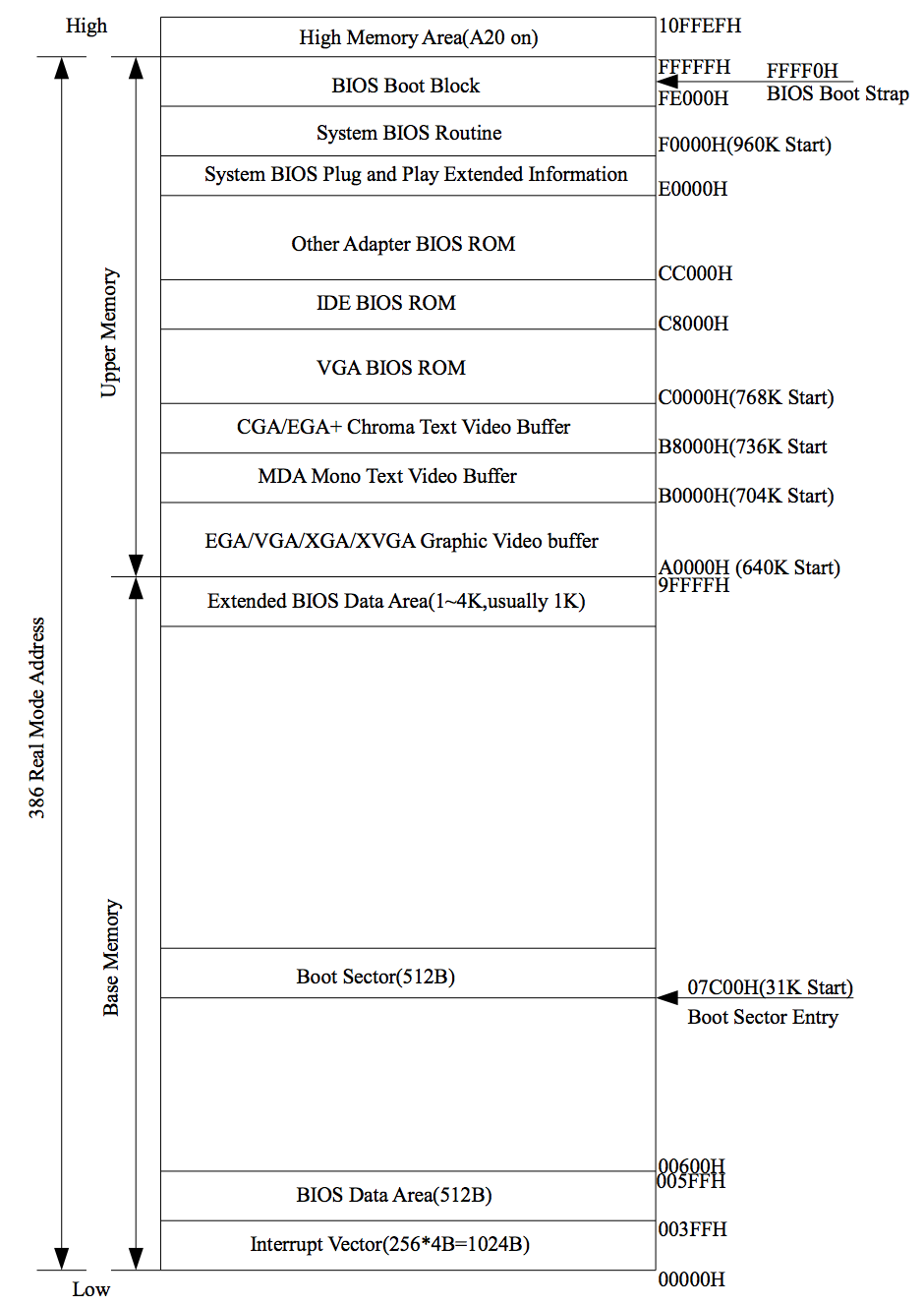


图 实模式内存地址空间分布

　　看到0x7c00h了吗？0x0000h--0x7c00h这一段存的是BIOS中断向量和一些BIOS数据等，至于到底为什么以0x7c00h为界，本文不做讨论，有兴趣看这里http://www.glamenv-septzen.net/en/view/6。

**2. int 10h是干嘛的?**

当出现int 10h中断时，表示要操作显示器了，此时AH寄存器表示如何显示，代码中的AH为13h，表示要在TTY（伪终端）显示字符，此时其他几个寄存器都有一定的含义，如下所示：

ES:BP -- 显示字符串的地址　　　　CX -- 显示字符串的长度

BH -- 页码　　　　                      BL -- 属性(若AL=00H或 01H)

DH -- 行 　　　　　　　　　　　　 DL -- 列

AL -- 显示输出方式

下面一段代码也就不难理解了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | mov ax, cs  mov es, ax  mov ax, msg  mov bp, ax                    ; ES:BP表示显示字符串的地址  mov cx, msgLen                ; CX存字符长度  mov ax, 1301h                 ; AH=13h表示向TTY显示字符，AL=01h表示显示方式（字符串是否包含显示属性，01h表示不包含）  mov bx, 000fh                 ; BH=00h表示页号，BL=0fh表示颜色  mov dl, 0                     ; 列  int 10h |

**3. $和$$是什么意思？**

$   是当前位置$$ 是段开始位置

下面两句就不难理解了：

|  |  |
| --- | --- |
| 1  2 | msgLen: equ $ - msg           ; 字符串长度  times 510 - ($ - $$) db 0     ; 填充剩余部分 |

**4. 为什么要有0xaa55h魔数？**

　　BIOS检查完硬件后，会寻找下一个设备来启动计算机，BIOS找到一个设备后，会读取该设备的第一个扇区，也就是读取最前面的512个字节。如果这512个字节的最后两个字节是0x55和0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给"启动顺序"中的下一个设备。

**第五节 制作软盘镜像，加入主引导记录**

　　如何用dd命令制作软盘，自行google之。

**1. 首先，我们制作一个空的软盘镜像empty.img：**

# dd if=/dev/zero of=empty.img bs=512 count=2880

**2. 之后，我们制作一个包含主引导记录boot.bin的镜像文件lucasOS.img:**

# dd if=boot.bin of=lucasOS.img bs=512 count=1

**3. 然后，将empty.img中1个扇区后的数据拷贝到lucasOS.img的后:**

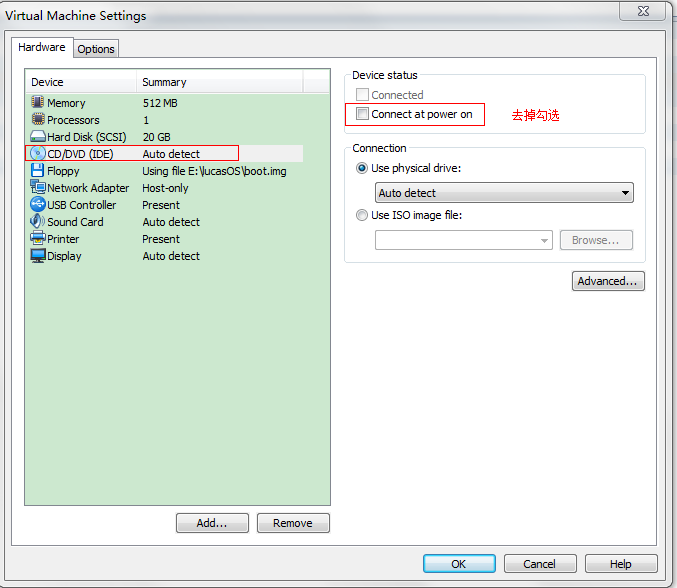
# dd if=empty.img of=lucasOS.img skip=1 seek=1 bs=512 count=2879

　　这样就做成了一个大小为1.44Mb的包含主引导记录的软盘镜像文件lucasOS.img。

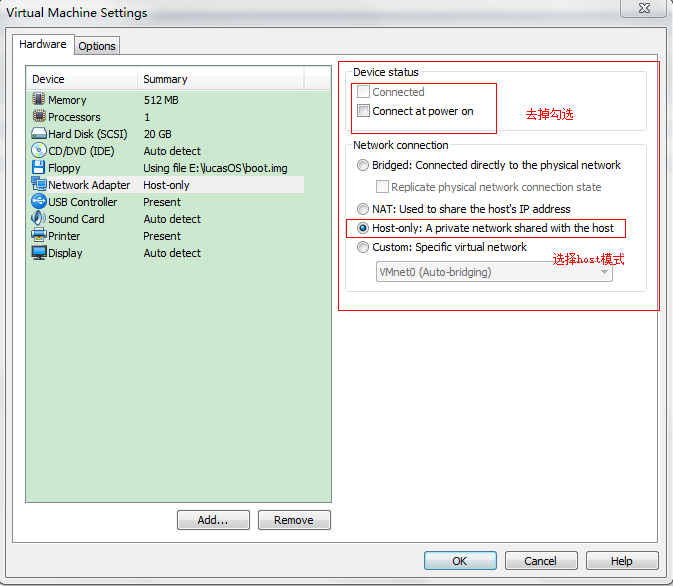
**4. 将虚拟机ubuntu中的文件lucasOS.img拷贝到windows下（鼠标直接拖拽，如果不行google之）。**

**第六节 用软盘镜像lucasOS.img启动一个空的虚拟机**

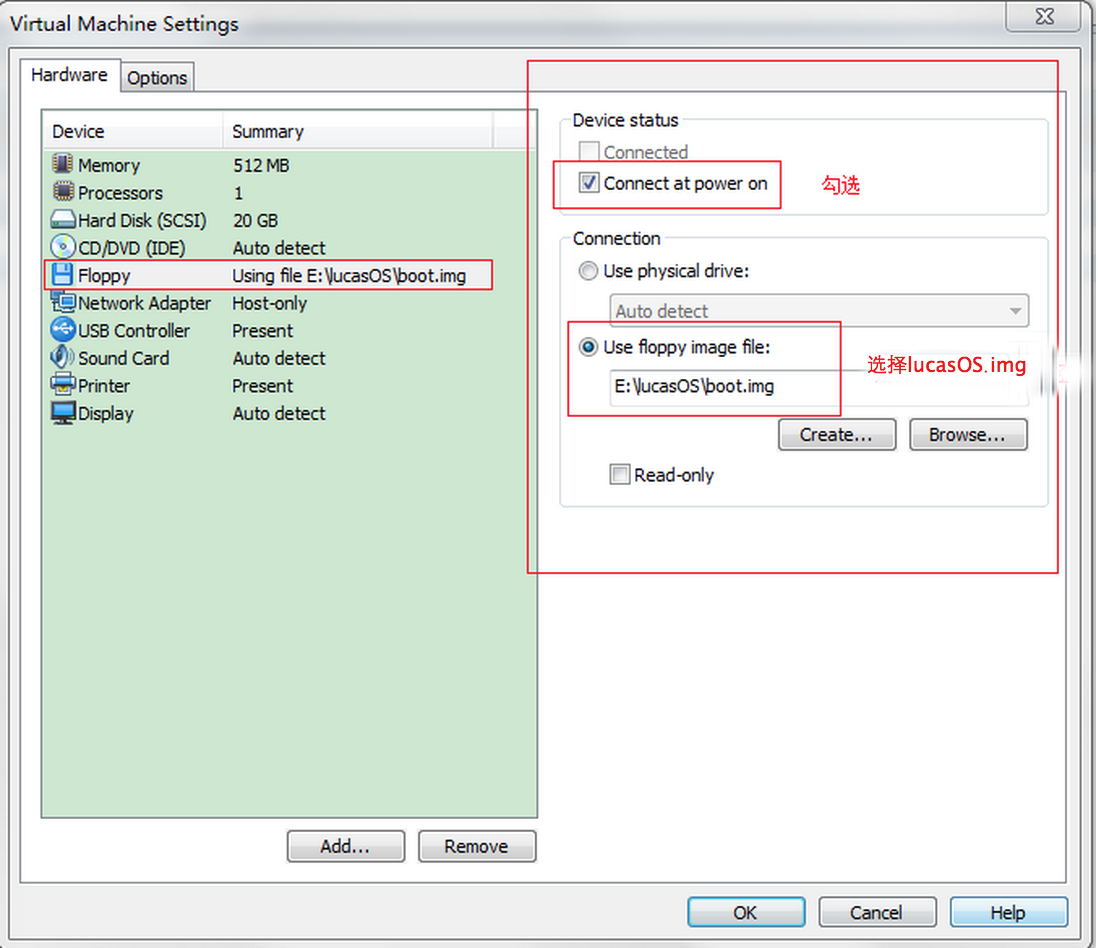
**1. VMware创建空的虚拟机，去掉开机从CD/DVD启动选项。**

****

**2. 网络选择host-only模式。**



**3. 选择从软盘驱动，路径选择上一节已经拷贝到windows下的镜像lucasOS.img.**



**4. 开启虚拟机电源，看到如下画面，恭喜你，成功了。**



　　好了，至此，我们完成了主引导记录，后续会讨论载入内核，并进一步讨论进程管理、内存管理、文件系统和中断等等。

**代码获取**

　　本系列GitHub地址 https://github.com/lucasysfeng/lucasOS.git

　　获取代码:

# git clone https://github.com/lucasysfeng/lucasOS.git

　　本讲的代码是code/chapter1，笔者已经将上面的命令集成到Makefile中了，读者只需进入目录，按ReadMe.txt执行即可。有问题请留言。

part2

# [跟我一起写操作系统(二)——史上最简单的内核](http://www.cnblogs.com/lucasysfeng/p/4847662.html)

　　转载注明出处：<http://www.cnblogs.com/lucasysfeng/p/4847662.html>

　　上一讲地址：[http://www.cnblogs.com/lucasysfeng/p/4846119.html](http://www.cnblogs.com/lucasysfeng/p/4846119.html" \t "_blank)

　　项目地址：[https://github.com/lucasysfeng/lucasOS](https://github.com/lucasysfeng/lucasOS" \t "_blank)

　　上一讲我们介绍了计算机的启动流程，并给出了一份简单的主引导记录代码，此份代码仅仅是显示几个字符，并没有做它本应该做的事--启动内核。本讲我们首先看下内核是如何被启动的，然后写一个简单的内核，用已经实现的主引导记录配合GRUB启动它。

## 如何启动内核

　　前一讲我们说到，计算机读取"主引导记录"前面446字节的机器码之后，会运行事先安装的“启动管理器”bootloader，由用户选择启动哪个内核，之后就会载入内核，将控制权交给内核。GNU GRUB（GRand Unified Bootloader）就是一种bootloader，满足多重引导规范（The Multiboot Specification），GRUB可选择操作系统分区上的不同内核，下图就是GRUB的图形界面：

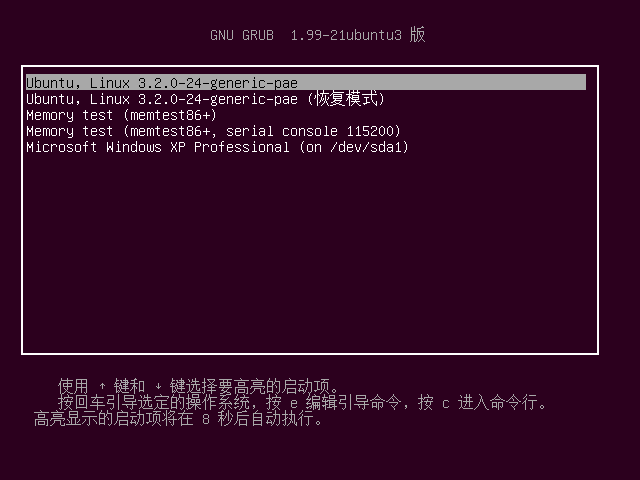


图 GRUB界面

　　能够被GRUB启动的内核需要满足两个的条件:

(1) 内核的前8K字节内必须要包含多重引导规范的头信息（Multiboot Header）；  
(2) 内核要加载在内存地址的1MB以上。

　　那么Multiboot Header是什么样子的呢？它必须包含4字节对齐的3个域（还有其他非必须域，我们不讨论），如下：

魔数域（magic）：标志头的魔数，必须等于 0x1BADB002。。   
标志域（flag）：是否需要引导程序支持某些特性，我们不关心这些特性，这个标志置为0。   
校验域（checksum）：校验等式是否成立（magic + flags + checksum = 0）

 　　本文不讨论GRUB的实现，我们会用前人已经写好的GRUB（笔者会给出），我们要做的是完成符合GRUB启动规范的内核。为了完成这个内核，我们需要写少量的汇编用来在内核中加入Multiboot Header，然后用C语言写内核入口，最后将汇编目标代码和C语言目标代码链接起来生成真正的内核。下面就让我们一步步地完成这些吧！

## 第一步 汇编入口

### 1. 汇编代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | ; 文件名 boot.asm  ; Copyright: www.cnblogs.com/lucasysfeng    MBOOT\_MAGIC  equ 0x1BADB002  ; multiboot magic域，必须为此值  MBOOT\_FLAGS  equ 0x00        ; multiboot flag域, GRUB启动时是否要做一些特殊操作  MBOOT\_CHECKSUM  equ -(MBOOT\_MAGIC + MBOOT\_FLAGS) ; multiboot checksum域，校验上面两个域是否正确    [BITS 32]                    ; 以32位编译    section .text    dd  MBOOT\_MAGIC    dd  MBOOT\_FLAGS    dd  MBOOT\_CHECKSUM    dd  start    [GLOBAL start]  [EXTERN kernel\_main]         ; 内核入口函数, EXTERN表明此符号在外部定义    start:    cli                        ; 禁用中断    call kernel\_main           ; 调用内核入口函数    jmp $                      ; 无限循环 |

　　在上面汇编中，我们定义了GRUB启动需要的域MBOOT\_MAGIC、MBOOT\_FLAGS和MBOOT\_CHECKSUM，并调用了内核入口函数kernel\_main, kernel\_main下一节实现。

### 2. 编译生成目标文件boot.o

# nasm -f elf boot.asm -o boot.o

　　运行上面命令后会生成目标文件boot.o，-f  elf的意思是生成ELF格式的目标代码。

## 第二步 内核入口

### 1. 内核代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  # Copyright(c) www.cnblogs.com/lucasysfeng, all rights reserved  # File        : kernel.c  # Author      : lucasysfeng  # Description : 内核入口函数  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    int kernel\_main()  {      // 显存开始地址      char \*display\_buf = (char\*)0xb8000;        // 清屏      unsigned int i = 0;      const unsigned int total = 80 \* 25 \* 2;      // 一屏25行，每行80个字符，每个字符2个字节      while(i < total)      {          display\_buf[i++] = ' ';          display\_buf[i++] = 0x04;                 // 颜色      }        // 显示字符      const char \*str = "Hello World, welcome to kernel!";      for (i = 0; '\0' != \*str;)      {          display\_buf[i++] = \*(str++);          display\_buf[i++] = 0x04;      }        return 0;  } |

　　0xb8000h是显存开始的地址，读者可以看第一讲（http://www.cnblogs.com/lucasysfeng/p/4846119.html）“实模式内存地址空间分布”那张图，找到0xb8000h这个地址。从0xb8000h这个地址开始，每2个字节表示一个字符，前一个字节是字符的ASCII码，后一个字节是这个字符的颜色和属性，颜色和属性此处先不用关心。这段C代码的其余部分相信读者都能看得懂，我就不过多解释了。

### 2. 编译生成目标文件kernel.o

# gcc -m32 -c -o kernel.o kernel.c

　　运行上面命令后，目标文件kernel.o就生成了。

## 第三步 生成内核

 　　上面讲到了能被GRUB启动的内核需要满足的条件：

(1) 内核的前8K字节内必须要包含多重引导规范的头信息（Multiboot Header）；  
(2) 内核要加载在内存地址的1MB以上。

　　我们将头信息放在了汇编生成的目标文件boot.o中，因此我们需要将boot.o和kernel.o链接到一起生成真正的kernel,并且这个真正的内核要加载到1MB内存上，为此，我们需要下面的链接脚本和命令(关于链接脚本的使用自行google)：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 文件名: link.ld  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    ENTRY(start)  SECTIONS  {      . = 0x100000;        .text :      {          \*(.text)          . = ALIGN(4096);      }      .data :      {          \*(.data)          \*(.rodata)          . = ALIGN(4096);      }  } |

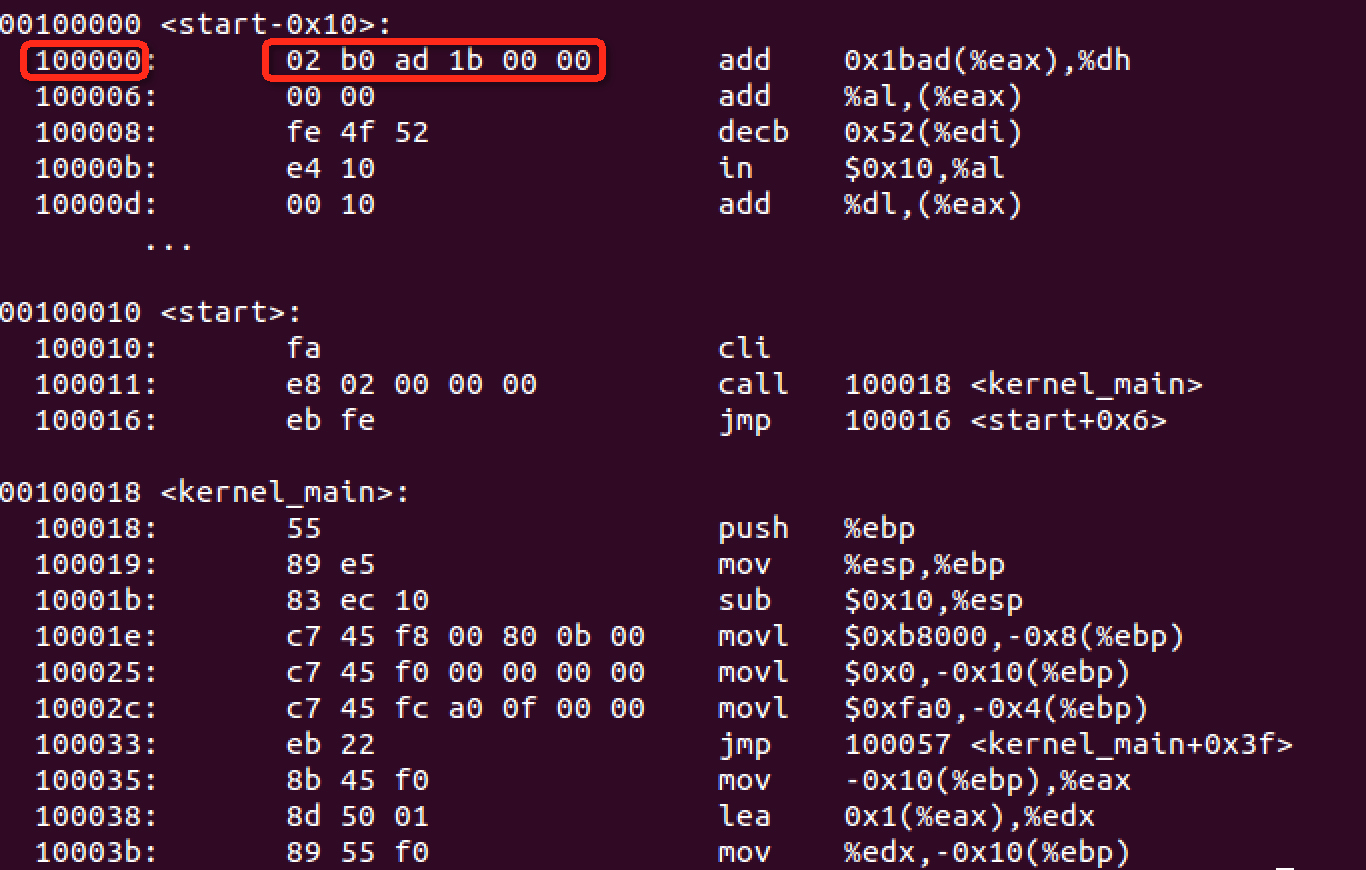
　　我们用ld命令链接目标文件boot.o和kernel.o，指明使用链接脚本link.ld:

# ld -T link.ld -m elf\_i386 -nostdlib boot.o kernel.o -o kernel

　　运行上面命令后，会生成我们要启动的真正的内核kernel，那么这个kernel是否满足GRUB启动规范呢？我们可以通过反汇编来看一下：

# objdump -d  kernel | head -n30

　　结果如下图所示，我们看到100000了吗，这个就是.text段起始的地址即1M，看到02 b0 ad 1b 00 00了吗，这个就是GRUB魔数域1b ad b0 02（大小端问题，反向存储）



## 第四步 将内核拷贝到软盘镜像

　　我们这里不制作软盘镜像，而是使用已经制作好的软盘镜像，镜像名称lucasOS.img，已经放在github上了。我们也无需制作GRUB，这个软盘镜像已经包含了GRUB.我们要做的是把内核文件kernel拷贝到软盘镜像lucasOS.img中。

### 1. 获取lucasOS.img软盘镜像。

　　lucasOS目录下的lucasOS.img就是我们要的软盘镜像。

# git clone https://github.com/lucasysfeng/lucasOS.git

### 2. 创建挂载点。

# sudo mkdir /mnt/lucasOS

### 3. 挂载软盘镜像。

　　注意把lucasOS.img改为你的lucasOS.img所在路径。

# sudo mount lucasOS.img /mnt/lucasOS

### 4. 把内核文件拷贝到软盘镜像中。

　　注意把kernel改为你的kernel所在路径。

# sudo cp kernel /mnt/lucasOS/kernel

### 5. 卸载软盘镜像。

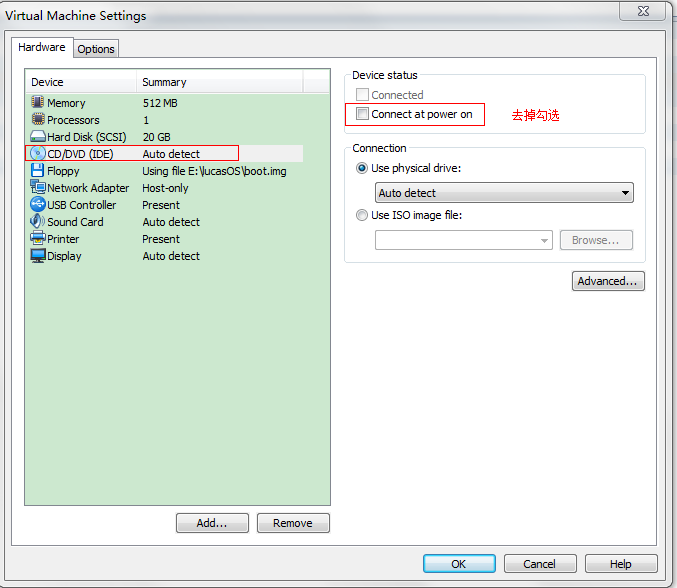
# sudo umount /mnt/lucasOS

## 

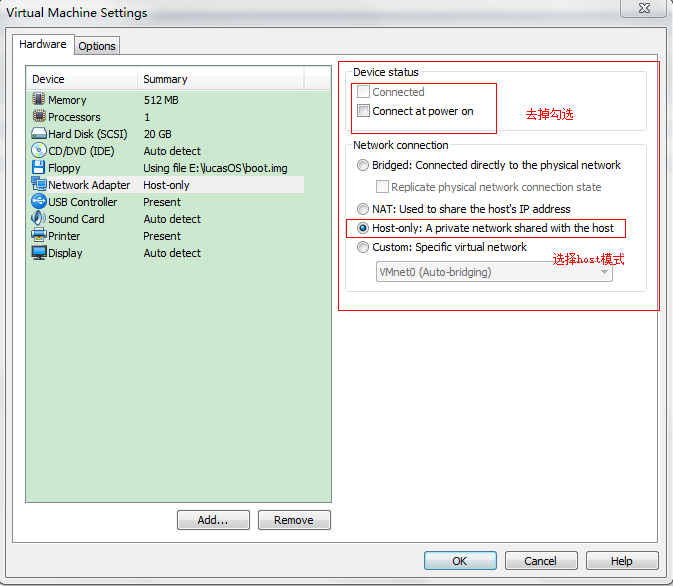
## 第五步 启动内核

 　　上一讲我们用软盘镜像启动了一个空的虚拟机，下面用同样地操作启动虚拟机，要记得把软盘镜像lucasOS.img从ubuntu拷贝到windows下。这里我们使用VMware创建虚拟机，当然也可以使用其他软件创建虚拟机。

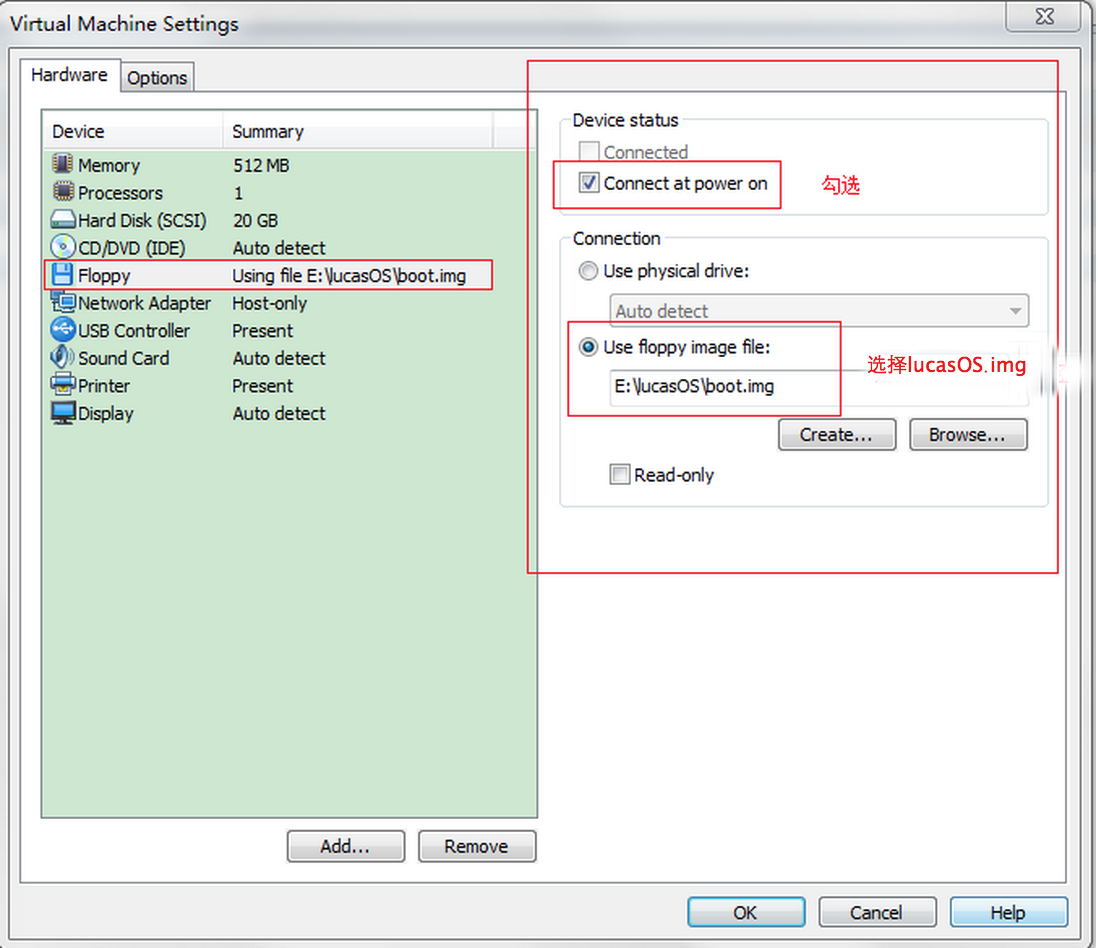
### 1. 创建空的虚拟机，去掉开机从CD/DVD启动选项。



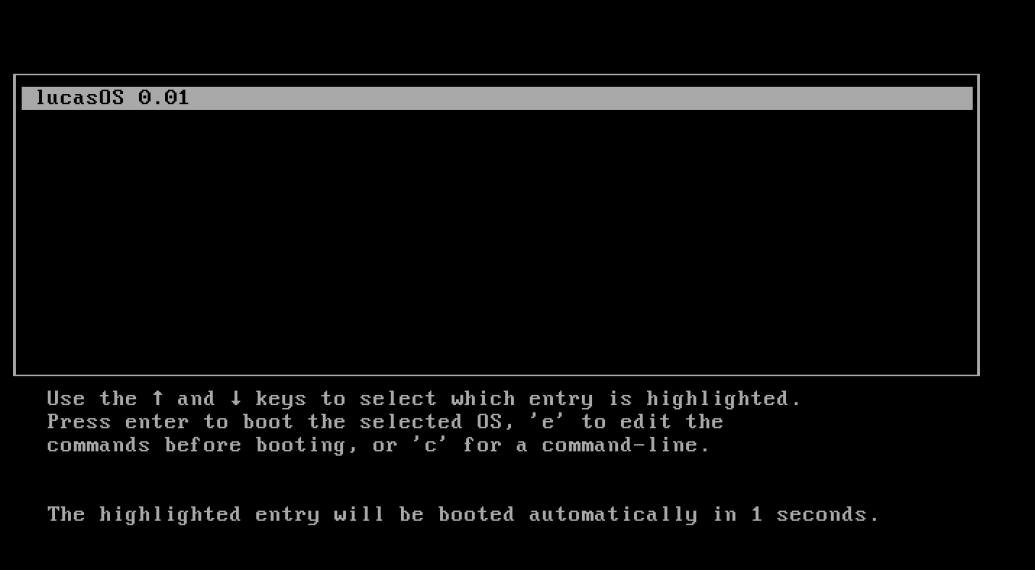
### 2. 网络选择host-only模式。



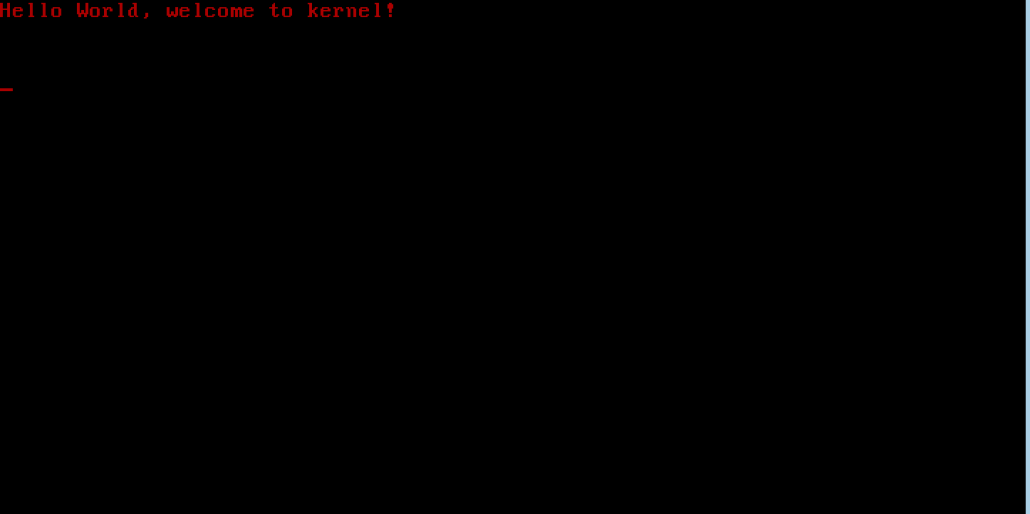
### 3. 选择从软盘驱动，路径选择已经拷贝到windows下的镜像lucasOS.img.



### 4. 开启虚拟机电源，看到如下的画面（约停留2s），恭喜你，GRUB成功了！



### 5. GRUB启动成功后，会自动载入内核，出现下面画面，恭喜你，载入内核成功！



　　好了，至此，我们完成了内核的启动流程，下一讲我们开始内核之旅！

## 代码获取

　　本系列GitHub地址 <https://github.com/lucasysfeng/lucasOS>，用下面命令获取代码:

# git clone https://github.com/lucasysfeng/lucasOS.git

本讲的代码是code/chapter2，笔者已经将上面的命令集成到Makefile中了，读者只需进入目录，按ReadMe.txt说明执行即可，有问题请留言。

part3

# [跟我一起写操作系统(三)——打印函数和qemu虚拟机](http://www.cnblogs.com/helloweworld/p/5036562.html)

　　转载注明出处：<http://www.cnblogs.com/lucasysfeng/p/4847662.html>

　　上一讲地址：[http://www.cnblogs.com/lucasysfeng/p/4847662.html](http://www.cnblogs.com/lucasysfeng/p/4847662.html" \t "_blank)

　　项目地址：[https://github.com/lucasysfeng/lucasOS](https://github.com/lucasysfeng/lucasOS" \t "_blank)

　　本讲主要涉及以下三个方面：1. 打印函数封装；2. qemu的使用；3. 项目目录结构调整。 理由如下：1. 后续开发无疑会遇到打印变量的操作，因此有必要封装打印函数（内核开发阶段，无法调用库函数）；2. 前两节，我们都是生成镜像文件，然后由VMvare启动，过程较繁琐，后续我们使用qemu虚拟机，可以直接在Linux环境下启动qemu；3. 随着项目的复杂，有必要调整目录结构，并调整Makefile.

　　本讲没有过多讲解，看代码完全可以理解。本讲只是列出了部分代码，全部代码对应<https://github.com/lucasysfeng/lucasOS>上的code/chapter3.

# 清屏

　　0xb8000h是显存开始的地址，因此打印就是操作0xb8000h开始的地址，从0xb8000h这个地址开始，每2个字节表示一个字符，前一个字节是字符的ASCII码，后一个字节是这个字符的颜色和属性。

　　lib/kprint.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | // 一屏25行  #define LINE\_COUNT 25  // 一行80个字符  #define LINE\_LENGTH 80  // 字符颜色  #define COLOR 0x04    // 显存开始地址  static unsigned short \*display\_buf = (unsigned short \*) 0xb8000;  // 字符属性  static const unsigned short char\_attr = COLOR << 8;    // 当前光标的坐标  static unsigned char cursor\_x = 0;  static unsigned char cursor\_y = 0;    /\*\*   \* 清屏   \*/  void print\_clear()  {      unsigned int i;      for (i = 0; i < LINE\_COUNT \* LINE\_LENGTH; i++)      {          display\_buf[i] = 0x20 | char\_attr;      }        cursor\_x = 0;      cursor\_y = 0;  } |

# 打印字符

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*\*   \* 打印一个字符（为简化逻辑，某些特殊字符均作为换行符处理）   \*/  void print\_char(const char c)  {      if (c >= ' ')      {          display\_buf[cursor\_y \* LINE\_LENGTH + cursor\_x] = c | char\_attr;          cursor\_x++;      }      else      {          cursor\_x = 0;          cursor\_y++;      }        // 每行80个字符，满80个字符换行      if (cursor\_x >= 80)      {          cursor\_x = 0;          cursor\_y++;      }        scroll();  // 满一屏时，向上滚屏  } |

# 打印字符串

　　调用打印字符函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*   \* 打印以\0结尾的字符串   \*/  void print\_str(const char\* str)  {      while (\*str)      {          print\_char(\*str++);      }  } |

# 内核入口

 　　print\_dec()   print\_hex()在github上，此处没给出实现。

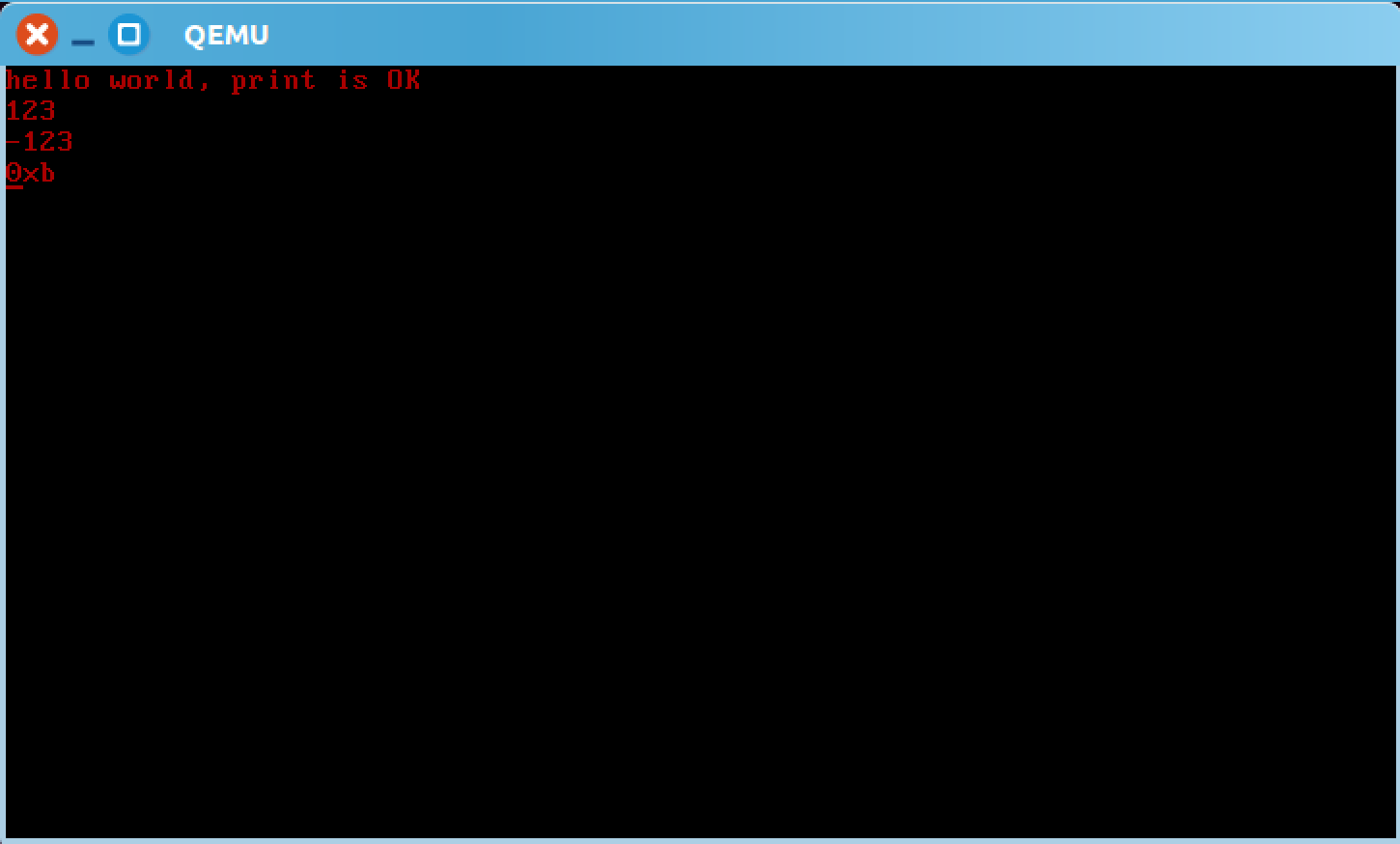
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include "kprint.h"    int kernel\_main()  {      print\_clear();        print\_str("hello world, print is OK\n");      print\_dec(123);      print\_char('\n');      print\_dec(-123);      print\_char('\n');      print\_hex(11);        return 0;  } |

# 编译和运行

 　　我们使用qemu启动虚拟机，所以先要安装qemu，qemu的安装和使用此处略。

$ make

$ make qemu

　　make后会生成kernel内核，并将内核拷贝到lucasOS.img中，运行make qemu就会加载lucasOS.img，并出现下面窗口，表示成功了：　　　　　　　　

## 代码获取

　　本系列GitHub地址 <https://github.com/lucasysfeng/lucasOS>，本讲的代码是code/chapter3.

part4

# http://www.cnblogs.com/lucasysfeng/p/5059767.html

转载注明出处：[http://www.cnblogs.com/lucasysfeng/p/4847662.html](http://www.cnblogs.com/lucasysfeng/p/4847662.html" \t "_blank)

上一讲地址：[http://www.cnblogs.com/lucasysfeng/p/5036562.html](http://www.cnblogs.com/lucasysfeng/p/5036562.html" \t "_blank)

项目地址：[https://github.com/lucasysfeng/lucasOS](https://github.com/lucasysfeng/lucasOS" \t "_blank)

　　我们知道，内存管理是操作系统的重要组成部分，在学习内存管理之前，首先要解决一个问题：如何获取物理内存？在前几讲我们谈到，内核是由GRUB启动的，因此要在内核中获取物理内存时，我们可以通过GRUB获取。

## multiboot\_t结构体

 　　GRUB将内存的分布放到了multiboot\_t结构体里，关于该结构体的相关信息和字段介绍可以看这里[http://www.uruk.org/orig-grub/boot-proposal.html](http://www.uruk.org/orig-grub/boot-proposal.html" \t "_blank)，该结构体如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | typedef struct multiboot\_t  {      uint32\_t flags;      uint32\_t mem\_lower;      uint32\_t mem\_upper;      uint32\_t boot\_device;      uint32\_t cmdline;      uint32\_t mods\_count;      uint32\_t mods\_addr;      uint32\_t num;      uint32\_t size;      uint32\_t addr;      uint32\_t shndx;      uint32\_t mmap\_length;      uint32\_t mmap\_addr;      uint32\_t drives\_length;      uint32\_t drives\_addr;      uint32\_t config\_table;      uint32\_t boot\_loader\_name;      uint32\_t apm\_table;      uint32\_t vbe\_control\_info;      uint32\_t vbe\_mode\_info;      uint32\_t vbe\_mode;      uint32\_t vbe\_interface\_seg;      uint32\_t vbe\_interface\_off;      uint32\_t vbe\_interface\_len;  }\_\_attribute\_\_((packed)) multiboot\_t; |

　　我们没有必要了解每个字段，重点来关注下mmap\_addr和mmap\_length， mmap\_addr是缓冲区的地址，mmap\_length是缓冲区的总大小。缓冲区由一个或者多个下面的结构对组成：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | typedef struct mmap\_entry\_t  {      uint32\_t size;      uint32\_t base\_addr\_low;      uint32\_t base\_addr\_high;      uint32\_t length\_low;      uint32\_t length\_high;      uint32\_t type;  }\_\_attribute\_\_((packed)) mmap\_entry\_t; |

　　size是相关结构的大小，单位是字节，它可能大于最小值20. base\_addr\_low是启动地址的低32位，base\_addr\_high是高32位，启动地址总共有64位。length\_low是内存区域大小的低32位，length\_high是内存区域大小的高32位，总共是64位。type是相应地址区间的类型，1代表可用RAM，所有其它的值代表保留区域。

## 获取物理内存

　　我们在内核代码里打印出地址，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void show\_memory\_map()  {      uint32\_t mmap\_addr = glb\_mboot\_ptr->mmap\_addr;      uint32\_t mmap\_length = glb\_mboot\_ptr->mmap\_length;        mmap\_entry\_t \*mmap = (mmap\_entry\_t \*) mmap\_addr;      for (mmap = (mmap\_entry\_t \*) mmap\_addr;              (uint32\_t) mmap < mmap\_addr + mmap\_length; mmap++)      {          print\_hex((uint32\_t) mmap->base\_addr\_low);          print\_char('\n');      }  } |

# 编译和运行

 　　我们使用qemu启动虚拟机。

$ make

$ make qemu

　　make后会生成kernel内核，并将内核拷贝到lucasOS.img中，运行make qemu就会加载lucasOS.img，并出现下面窗口，表示成功了：

## http://images2015.cnblogs.com/blog/443349/201512/443349-20151219211518991-1897659947.png

## 代码获取

　　本系列GitHub地址 <https://github.com/lucasysfeng/lucasOS>，本讲的代码是code/chapter4.

part5

# [跟我一起写操作系统(五)——分配物理内存](http://www.cnblogs.com/helloweworld/p/5061763.html)

转载注明出处：<http://www.cnblogs.com/lucasysfeng/p/4847662.html>

上一讲地址：[http://www.cnblogs.com/lucasysfeng/p/5059767.html](http://www.cnblogs.com/lucasysfeng/p/4846119.html" \t "_blank)

项目地址：[https://github.com/lucasysfeng/lucasOS](https://github.com/lucasysfeng/lucasOS" \t "_blank)

　　上一讲我们获取到了物理内存，本讲我们来看如何分配物理内存。为简化问题，我们不使用伙伴内存分配算法，而是把可用内存按页存入栈中，每次分配从栈中取就可以了。

　　本讲的主要代码在lib/pmm.c中，下面给出部分代码。本讲内容不多，读者看代码即可。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | void init\_pmm()  {      mmap\_entry\_t \*mmap\_start\_addr = (mmap\_entry\_t \*) glb\_mboot\_ptr->mmap\_addr;      mmap\_entry\_t \*mmap\_end\_addr = (mmap\_entry\_t \*) glb\_mboot\_ptr->mmap\_addr              + glb\_mboot\_ptr->mmap\_length;        mmap\_entry\_t \*map\_entry;        for (map\_entry = mmap\_start\_addr; map\_entry < mmap\_end\_addr; map\_entry++)      {            // type为1表示可用内存，其它指保留区域，参考上一讲的multiboot\_t结构          if (map\_entry->type == 1 && map\_entry->base\_addr\_low == 0x100000)          {                // 可用内存要减去内核本身占用的内存              uint32\_t page\_addr = map\_entry->base\_addr\_low                      + (uint32\_t) (kern\_end - kern\_start);              uint32\_t length = map\_entry->base\_addr\_low + map\_entry->length\_low;                while (page\_addr < length && page\_addr <= PMM\_MAX\_SIZE)              {                  pmm\_free\_page(page\_addr);                  page\_addr += PMM\_PAGE\_SIZE;                  phy\_page\_count++;              }          }      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5 | void pmm\_free\_page(uint32\_t p)  {      assert(pmm\_stack\_top != PAGE\_MAX\_SIZE, "out of pmm\_stack stack\n");      pmm\_stack[++pmm\_stack\_top] = p;  } |

## 代码获取

　　本系列GitHub地址 <https://github.com/lucasysfeng/lucasOS>，本讲的代码是code/chapter5.