



下载APP



## 02 | 几行汇编几行C：实现一个最简单的内核

2021-05-12 LMOS

操作系统实战45讲

[进入课程 >](#)**讲述：陈晨**

时长 13:26 大小 12.32M



你好，我是 LMOS。

我们知道，在学习许多编程语言一开始的时候，都有一段用其语言编写的经典程序——Hello World。这不过是某一操作系统平台之上的应用程序，却心高气傲地问候世界。

而我们学习操作系统的时候，那么也不妨撇开其它现有的操作系统，基于硬件，写一个最小的操作系统——Hello OS，先练练手、热热身，直观感受一下。

### PC 机的引导流程



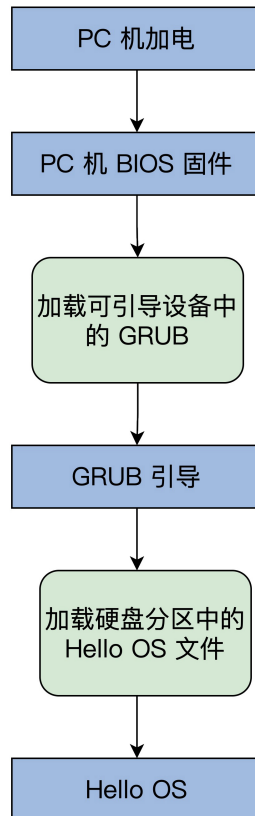
看标题就知道，写操作系统要用汇编和 C 语言，尽管这个 Hello OS 很小，但也要用到两种编程语言。其实，现有的商业操作系统都是用这两种语言开发出来的。

先不用害怕，Hello OS 的代码量很少。

其实，我们也不打算从 PC 的引导程序开始写起，原因是目前我们的知识储备还不够，所以先借用一下 GRUB 引导程序，只要我们的 PC 机上安装了 Ubuntu Linux 操作系统，GRUB 就已经存在了。这会大大降低我们开始的难度，也不至于打消你的热情。



那在写 Hello OS 之前，我们先要搞清楚 Hello OS 的引导流程，如下图所示：



Hello OS引导流程图

简单解释一下，PC 机 BIOS 固件是固化在 PC 机主板上的 ROM 芯片中的，掉电也能保存，PC 机上电后的第一条指令就是 BIOS 固件中的，它负责**检测和初始化 CPU、内存及主板平台**，然后加载引导设备（大概率是硬盘）中的第一个扇区数据，到 0x7c00 地址开始的内存空间，再接着跳转到 0x7c00 处执行指令，在我们这里的情况下就是 GRUB 引导程序。

当然，更先进的 [UEFI BIOS](#) 则不同，这里就不深入其中了，你可以通过链接自行了解。


## Hello OS 引导汇编代码

明白了 PC 机的启动流程，下面只剩下我们的 Hello OS 了，我们马上就去写好它。

我们先来写一段汇编代码。这里我要特别说明一个问题：为什么不能直接用 C？

**C 作为通用的高级语言，不能直接操作特定的硬件，而且 C 语言的函数调用、函数传参，都需要用栈。**

栈简单来说就是一块内存空间，其中数据满足**后进先出**的特性，它由 CPU 特定的栈寄存器指向，所以我们要先用汇编代码处理好这些 C 语言的工作环境。

 复制代码

```
1 ;彭东 @ 2021.01.09
2 MBT_HDR_FLAGS EQU 0x00010003
3 MBT_HDR_MAGIC EQU 0x1BADB002 ;多引导协议头魔数
4 MBT_HDR2_MAGIC EQU 0xe85250d6 ;第二版多引导协议头魔数
5 global _start ;导出_start符号
6 extern main ;导入外部的main函数符号
7 [section .start.text] ;定义.start.text代码节
8 [bits 32] ;汇编成32位代码
9 _start:
10 jmp _entry
11 ALIGN 8
12 mbt_hdr:
13 dd MBT_HDR_MAGIC
14 dd MBT_HDR_FLAGS
15 dd -(MBT_HDR_MAGIC+MBT_HDR_FLAGS)
16 dd mbt_hdr
17 dd _start
18 dd 0
19 dd 0
20 dd _entry
21 ;以上是GRUB所需要的头
22 ALIGN 8
23 mbt2_hdr:
24 DD MBT_HDR2_MAGIC
25 DD 0
26 DD mbt2_hdr_end - mbt2_hdr
27 DD -(MBT_HDR2_MAGIC + 0 + (mbt2_hdr_end - mbt2_hdr))
28 DW 2, 0
29 DD 24
30 DD mbt2_hdr
31 DD _start
32 DD 0
33 DD 0
34 DW 3, 0
35 DD 12
36 DD _entry
37 DD 0
38 DW 0, 0
39 DD 8
40 mbt2_hdr_end:
41 ;以上是GRUB2所需要的头
42 ;包含两个头是为了同时兼容GRUB、GRUB2
43 ALIGN 8
44 _entry:
45 ;关中断
46 cli
```

```
47 ;关不可屏蔽中断
48 in al, 0x70
49 or al, 0x80
50 out 0x70,al
51 ;重新加载GDT
52 lgdt [GDT_PTR]
53 jmp dword 0x8 :_32bits_mode
54 _32bits_mode:
55 ;下面初始化C语言可能会用到的寄存器
56 mov ax, 0x10
57 mov ds, ax
58 mov ss, ax
59 mov es, ax
60 mov fs, ax
61 mov gs, ax
62 xor eax,eax
63 xor ebx,ebx
64 xor ecx,ecx
65 xor edx,edx
66 xor edi,edi
67 xor esi,esi
68 xor ebp,ebp
69 xor esp,esp
70 ;初始化栈, C语言需要栈才能工作
71 mov esp,0x9000
72 ;调用C语言函数main
73 call main
74 ;让CPU停止执行指令
75 halt_step:
76 halt
77 jmp halt_step
78 GDT_START:
79 knull_dsc: dq 0
80 kcode_dsc: dq 0x00cf9e000000ffff
81 kdata_dsc: dq 0x00cf92000000ffff
82 k16cd_dsc: dq 0x00009e000000ffff
83 k16da_dsc: dq 0x000092000000ffff
84 GDT_END:
85 GDT_PTR:
86 GDTLEN dw GDT_END-GDT_START-1
87 GDTBASE dd GDT_START
```

以上的汇编代码 (/lesson01/HelloOS/entry.asm) 分为 4 个部分：

1. 代码 1~40 行，用汇编定义的 GRUB 的多引导协议头，其实就是一定格式的数据，我们的 Hello OS 是用 GRUB 引导的，当然要遵循 **GRUB 的多引导协议标准**，让 GRUB 能识别我们的 Hello OS。之所以有两个引导头，是为了兼容 GRUB1 和 GRUB2。

2. 代码 44~52 行，关掉中断，设定 CPU 的工作模式。你现在可能不懂，没事儿，后面 CPU 相关的课程我们会专门再研究它。


3. 代码 54~73 行，初始化 CPU 的寄存器和 C 语言的运行环境。

4. 代码 78~87 行，GDT\_START 开始的，是 CPU 工作模式所需要的数据，同样，后面讲 CPU 时会专门介绍。

## Hello OS 的主函数

到这，不知道你有没有发现一个问题？上面的汇编代码调用了 main 函数，而在其代码中并没有看到其函数体，而是从外部引入了一个符号。

那是因为这个函数是用 C 语言写的在 (/lesson01/HelloOS/main.c) 中，最终它们分别由 nasm 和 GCC 编译成可链接模块，由 LD 链接器链接在一起，形成可执行的程序文件：

 复制代码

```
1 //彭东 @ 2021.01.09
2 #include "vgastr.h"
3 void main()
4 {
5     printf("Hello OS!");
6     return;
7 }
```

以上这段代码，你应该很熟悉了吧？不过这不是应用程序的 main 函数，而是 Hello OS 的 main 函数。

其中的 printf 也不是应用程序库中的那个 printf 了，而是需要我们自己实现了。你可以先停下歇歇，再去实现 printf 函数。

## 控制计算机屏幕

接着我们再看下显卡，这和我们接下来要写的代码有直接关联。

计算机屏幕显示往往是显卡的输出，显卡有很多形式：集成在主板上的叫集显，做在 CPU 芯片内的叫核显，独立存在通过 PCIE 接口连接的叫独显，性能依次上升，价格也是。

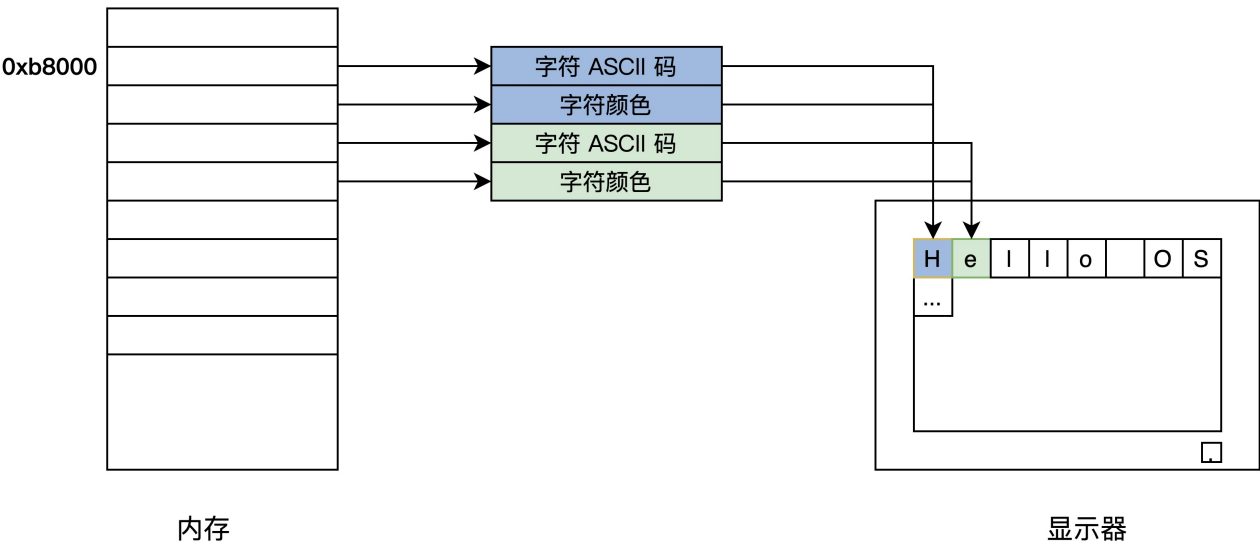
独显的高性能是游戏玩家们所钟爱的，3D 图形显示往往要涉及顶点处理、多边形的生成和变换、纹理、着色、打光、栅格化等。而这些任务的计算量超级大，所以独显往往有自己的 RAM、多达几百个运算核心的处理器。因此独显不仅仅是可以显示图像，而且可以执行大规模并行计算，比如“挖矿”。

我们要在屏幕上显示字符，就要编程操作显卡。

其实无论我们 PC 上是什么显卡，它们都支持一种叫 **VESA** 的标准，这种标准下有两种工作模式：字符模式和图形模式。显卡们为了兼容这种标准，不得不自己提供一种叫 VGABIOS 的固件程序。

下面，我们来看看显卡的字符模式的工作细节。

它把屏幕分成 24 行，每行 80 个字符，把这 (24\*80) 个位置映射到以 0xb8000 地址开始的内存中，每两个字节对应一个字符，其中一个字节是字符的 ASCII 码，另一个字节为字符的颜色值。如下图所示：



计算机显卡文本模式

明白了显卡的字符模式的工作细节，下面我们开始写代码。

这里先提个醒：**C 语言字符串是以 0 结尾的，其字符编码通常是 utf8，而 utf8 编码对 ASCII 字符是兼容的，即英文字符的 ASCII 编码和 utf8 编码是相等的**（关于 [utf8](#) 编码你可以自行了解）。

```
1 //彭东 @ 2021.01.09
2 void _strwrite(char* string)
3 {
4     char* p_strdst = (char*)(0xb8000); //指向显存的开始地址
5     while (*string)
6     {
7         *p_strdst = *string++;
8         p_strdst += 2;
9     }
10    return;
11 }
12
13 void printf(char* fmt, ...)
14 {
15     _strwrite(fmt);
16     return;
17 }
```

代码很简单，printf 函数直接调用了 \_strwrite 函数，而 \_strwrite 函数正是将字符串里每个字符依次定入到 0xb8000 地址开始的显存中，而 p\_strdst 每次加 2，这也是为了跳过字符的颜色信息的空间。

到这，Hello OS 相关的代码就写好了，下面就是编译和安装了。你可别以为这个事情就简单了，下面请跟着我去看一看。

## 编译和安装 Hello OS

Hello OS 的代码都已经写好，这时就要进入安装测试环节了。在安装之前，我们要进行系统编译，即把每个代码模块编译最后链接成可执行的二进制文件。

你可能觉得我在小题大做，编译不就是输入几条命令吗，这么简单的工作也值得一说？

确实，对于我们 Hello OS 的编译工作来说特别简单，因为总共才三个代码文件，最多四条命令就可以完成。

但是以后我们 Hello OS 的文件数量会爆炸式增长，一个成熟的商业操作系统更是多达几万个代码模块文件，几千万行的代码量，是这世间最复杂的软件工程之一。所以需要有一个牛逼的工具来控制这个巨大的编译过程。



## make 工具

make 历史悠久，小巧方便，也是很多成熟操作系统编译所使用的构建工具。

在软件开发中，make 是一个工具程序，它读取一个叫 “makefile” 的文件，也是一种文本文件，这个文件中写好了构建软件的规则，它能根据这些规则自动化构建软件。


makefile 文件中规则是这样的：首先有一个或者多个构建目标称为 “target”；目标后面紧跟着用于构建该目标所需要的文件，目标下面是构建该目标所需要的命令及参数。

与此同时，它也检查文件的依赖关系，如果需要的话，它会调用一些外部软件来完成任任务。

第一次构建目标后，下一次执行 make 时，它会根据该目标所依赖的文件是否更新决定是否编译该目标，如果所依赖的文件没有更新且该目标又存在，那么它便不会构建该目标。这种特性非常有利于编译程序源代码。

任何一个 Linux 发行版中都默认自带这个 make 程序，所以不需要额外的安装工作，我们直接使用即可。

为了让你进一步了解 make 的使用，接下来我们一起看一个有关 makefile 的例子：

 复制代码

```
1 CC = gcc #定义一个宏CC 等于gcc
2 CFLAGS = -c #定义一个宏 CFLAGS 等于-c
3 OBJJS_FILE = file.c file1.c file2.c file3.c file4.c #定义一个宏
4 .PHONY : all everything #定义两个伪目标all、everything
5 all:everything #伪目标all依赖于伪目标everything
6 everything :$( OBJJS_FILE) #伪目标everything依赖于OBJJS_FILE, 而OBJJS_FILE是宏会被
7 #替换成file.c file1.c file2.c file3.c file4.c
8 %.o : %.c
9 $(CC) $(CFLAGS) -o $@ $<
```

我来解释一下这个例子：

make 规定 “#” 后面为注释，make 处理 makefile 时会自动丢弃。

makefile 中可以定义宏，方法是在一个字符串后跟一个 “=” 或者 “:=” 符号，引用宏时要用 “\$(宏名)”，宏最终会在宏出现的地方替换成相应的字符串，例如：\$(CC) 会被替换成 gcc，\$(OBJ\_FILE) 会被替换成 file.c file1.c file2.c file3.c file4.c。

.PHONY 在 makefile 中表示定义伪目标。所谓伪目标，就是它不代表一个真正的文件名，在执行 make 时可以指定这个目标来执行其所在规则定义的命令。但是伪目标可以依赖于另一个伪目标或者文件，例如：all 依赖于 everything，everything 最终依赖于 file.c file1.c file2.c file3.c file4.c。

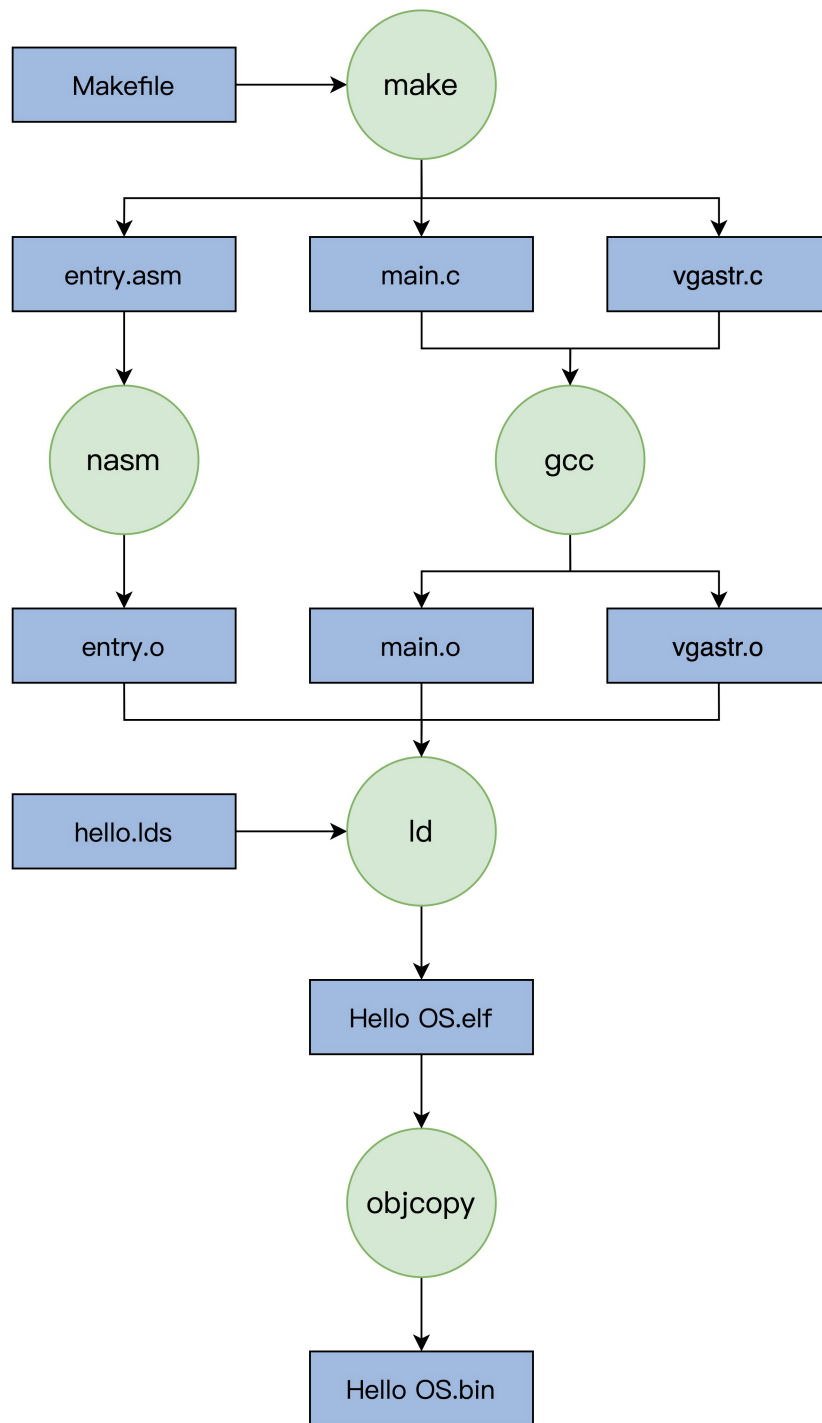
虽然我们会发现，everything 下面并没有相关的执行命令，但是下面有个通用规则：“%.o : %.c”。其中的 “%” 表示通配符，表示所有以 “.o” 结尾的文件依赖于所有以 “.c” 结尾的文件。

例如：file.c、file1.c、file2.c、file3.c、file4.c，通过这个通用规则会自动转换为依赖关系：file.o: file.c、file1.o: file1.c、file2.o: file2.c、file3.o: file3.c、file4.o: file4.c。

然后，针对这些依赖关系，分别会执行：\$(CC) \$(CFLAGS) -o \$@ \$< 命令，当然最终会转换为：gcc -c -o xxxx.o xxxx.c，这里的 “xxxx” 表示一个具体的文件名。

## 编译

下面我们用一张图来描述我们 Hello OS 的编译过程，如下所示：



Hello OS编译过程

## 安装 Hello OS

经过上述流程，我们就会得到 `Hello OS.bin` 文件，但是我们还要让 GRUB 能够找到它，才能在计算机启动时加载它。这个过程我们称为安装，不过这里没有写安装程序，得我们手动来做。

经研究发现，GRUB 在启动时会加载一个 `grub.cfg` 的文本文件，根据其中的内容执行相应的操作，其中一部分内容就是启动项。

GRUB 首先会显示启动项到屏幕，然后让我们选择启动项，最后 GRUB 根据启动项对应的信息，加载 OS 文件到内存。

下面来看看我们 Hello OS 的启动项：

[复制代码](#)

```
1 menuentry 'HelloOS' {  
2     insmod part_msdos #GRUB加载分区模块识别分区  
3     insmod ext2 #GRUB加载ext文件系统模块识别ext文件系统  
4     set root='hd0,msdos4' #注意boot目录挂载的分区，这是我机器上的情况  
5     multiboot2 /boot/HelloOS.bin #GRUB以multiboot2协议加载HelloOS.bin  
6     boot #GRUB启动HelloOS.bin  
7 }
```

如果你不知道你的 boot 目录挂载的分区，可以在 Linux 系统的终端下输入命令：df /boot/，就会得到如下结果：

[复制代码](#)

1	文件系统	1K-块	已用	可用	已用%	挂载点
2	/dev/sda4	48752308	8087584	38158536	18%	/

其中的“sda4”就是硬盘的第四个分区，但是 GRUB 的 menuentry 中不能写 sda4，而是要写“hd0,msdos4”，这是 GRUB 的命名方式，hd0 表示第一块硬盘，结合起来就是第一块硬盘的第四个分区。

把上面启动项的代码插入到你的 Linux 机器上的 /boot/grub/grub.cfg 文件中，然后把 Hello OS.bin 文件复制到 /boot/ 目录下，最后重启计算机，你就可以看到 Hello OS 的启动选项了。

选择 Hello OS，按下 Enter 键，这样就可以成功启动我们自己的 Hello OS 了。

## 重点回顾

有没有很开心？我们终于看到我们自己的 OS 运行了，就算它再简单也是我们自己的 OS。下面我们再次回顾下这节课的重点。

首先，我们了解了从按下 PC 机电源开关开始，PC 机的引导过程。它从 CPU 上电，到加载 BIOS 固件，再由 BIOS 固件对计算机进行自检和默认的初始化，并加载 GRUB 引导程序，最后由 GRUB 加载具体的操作系统。

其次，就到了我们这节课最难的部分，即用汇编语言和 C 语言实现我们的 Hello OS。

第一步，用汇编程序初始化 CPU 的寄存器、设置 CPU 的工作模式和栈，最重要的是**加入了 GRUB 引导协议头**；第二步，切换到 C 语言，用 C 语言写好了**主函数和控制显卡输出的函数**，其间还了解了显卡的一些工作细节。

最后，就是编译和安装 Hello OS 了。我们用了 make 工具编译整个代码，其实 make 会根据一些规则调用具体的 nasm、gcc、ld 等编译器，然后形成 Hello OS.bin 文件，你把这个文件写复制到 boot 分区，写好 GRUB 启动项，这样就好了。

这里只是上上手，下面我们还会去准备一些别的东西，然后就真正开始了。但你此刻也许还有很多问题没有搞清楚，比如重新加载 GDT、关中断等，先不要担心，我们后面会一一解决的。

本节课的**配套代码**，你可以从 [这里](#) 下载。

## 思考题

以上 printf 函数定义，其中有个形式参数很奇怪，请你思考下：为什么是 “...” 形式参数，这个形式参数有什么作用？

欢迎你在留言区分享你的思考或疑问。

我是 LMOS，我们下节课见！

30 人觉得很赞 | [提建议](#)

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | 程序的运行过程：从代码到机器运行

下一篇 03 | 黑盒之中有什么：内核结构与设计

## 精选留言 (78)

写留言



pedro 神 置顶

2021-05-12

不想在物理机上搞的同学们，可以参考一下我的 repo。如下：

```
``sh
git clone https://gitee.com/gaopedro/pos
cd pos
make && make umount_image...
```

展开 v

作者回复：很牛啊

10

22



陈诚 置顶

2021-05-12

写了一个关于这节课实验的笔记，VMware + Ubuntu 16.04环境，供大家参考  
<https://blog.csdn.net/chenchengwudi/article/details/116707122>

展开 v

作者回复：666666

3

17



杨军 置顶

2021-05-12

老师讲的详细，醍醐灌顶，后面会将如何用QEMU BOSCH之类的模拟器运行OS内容吗？

展开 v

作者回复：会的，敬请期待

**胡永** 置顶

2021-05-13

补充两个我遇到的坑：

1. 如何确定boot挂载的分区，参考Geek\_993581提供的方法，在grub命令行里确认。

set root='hd0,msdos4' #注意boot目录挂载的分区，修改为自己的分区。

我自己的是 (hd0,gpt2) 。

把insmod part\_msdos 修改为 part\_gpt。 ...

展开 ∨

作者回复: 你是对的 正确

**Viz** 置顶

2021-05-15

我也搞了份不需要 GRUB 可以直接用 QEMU 或者其他虚拟机软件跑的版本

[https://github.com/vizv/learn\\_os/tree/master/hello-os](https://github.com/vizv/learn_os/tree/master/hello-os)

克隆仓库后在 hello-os 目录里

...

展开 ∨

编辑回复: 6666，佩服佩服，感谢你的分享！教学相长，期待之后课程里，你还有更多的输出。

**第九天魔王** 置顶

2021-05-13

实际上boot目录只是一个目录，helloos文件可以放到其它目录，只要把grub.cfg里面相应位置改下就行了。把内核文件放到boot目录是一个既定的做法，目录下面有个vmlinuz-xxx就是Ubuntu的内核镜像。另外grub本身带有ext驱动，所以可以访问文件系统，取出你的内核镜像执行。

展开 ∨

编辑回复: 对的，666！期待你的更多精彩分享~



3

**DoubleL** 置顶

2021-05-13

在输入df /boot/ 时，要注意下【挂载点】的值是/还是/boot，东哥这里是/；如果是/boot，就要注意在修改/boot/grub/grub.cfg文件时，multiboot2这一项直接填写/HelloOS.bin即可，而不需要在前面加/boot了，因为/boot被划分为单独的分区了，即：  
multiboot2 /HelloOS.bin #GRUB以multiboot2协议加载HelloOS.bin

...

展开 ∨

编辑回复: 感谢老铁的优质分享!



2

**Eevee** 置顶

2021-05-13

将自己遇到的几个小问题总结下，希望能帮助到其他同学（linux新手）

1. 将HelloOs.bin移动到boot文件夹时无权限

解决办法：打开终端，运行命令sudo nautilus就可以打开一个管理员权限的文件管理器

2.df/boot/失效

解决办法：df-h 查看所有文件挂载，找到boot即可...

展开 ∨

作者回复: 谢谢



1

**冷板凳** 置顶

2021-05-12

从开机选项中看到自己的OS，很有成就感。会一点C和make，gcc方面不熟，nasm和ld需要额外了解。grub.cfg文件的挂载分区，一个参考命令，也可以参考里面已有的menuentry

展开 ∨

编辑回复: 太优秀了，再接再厉，这里带大家找找感觉。后面讲解更精彩，敬请期待。



1



Geek 002504



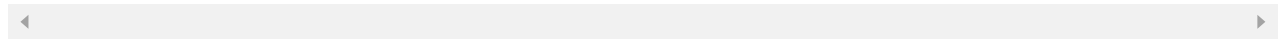


Geek 995581  
2021-05-12

终于试成功了，超开心！  
'Hello OS'

我是使用虚拟机  
df /boot/...  
展开 ∨

编辑回复: 太赞了，不但自己实战，还分享给大伙经验了，良性循环，666



2

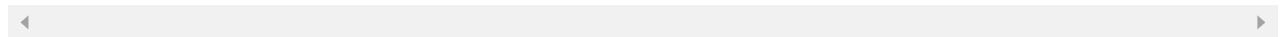
20



Strangeen  
2021-05-15

我是小白，只会java，没学过c，当时也只是抱着了解下操作系统的想法报了课程，学到第2课，发现完全听不懂...  
不过跟着课程和陈诚的博客走 (<https://blog.csdn.net/chenchengwudi/article/details/116707122>)，然后下载了老师的代码，最终居然还是跑出Hello OS!来了，虽然还不太明白汇编、Makefile、hello.ld代码的含义.....  
展开 ∨

作者回复: 你太用功了



1

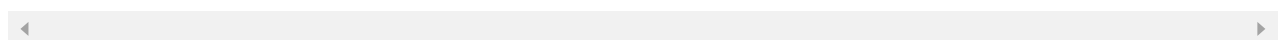
10



陈诚  
2021-05-12

如果看过王爽老师的《汇编语言》和李忠老师的《X86汇编语言：从实模式到保护模式》，这段的理解将非常简单，建议大家可以配合使用  
展开 ∨

作者回复: 目前不需要搞懂 先感受一下，我后面课程也会介绍的



1

7

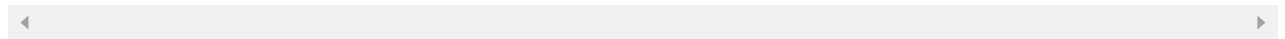


云师兄  
2021-05-12

终于看到操作系统从无到有的过程了！简直是完美啊！

展开 ▾

作者回复: 谢谢，一起折腾



💬 1

👍 7



**guojiun**

2021-05-12

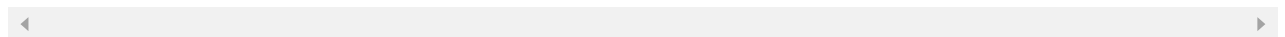
提供給大家參考我的配置 (可以成功看到 Hello OS!):

```
$ df -h /boot/
```

```
Filesystem Size Used Avail Use% Mounted on  
/dev/nvme0n1p1 468G 211G 234G 48% /...
```

展开 ▾

作者回复: 牛逼



💬

👍 6



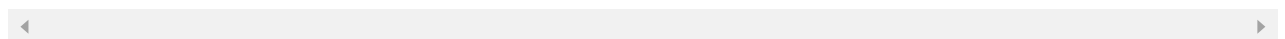
2021-05-12

... 在很多语言中是可变参数。在函数调用使用堆栈的前提下，参数的提取是从后往前。所以可以通过第一个参数的地址，拿到后续参数。

不过这里 main 函数中 printf 只有一个传参，printf 实现中也没用到。感觉可能是后续扩充代码使用的。

展开 ▾

作者回复: 对 对 对



💬

👍 5



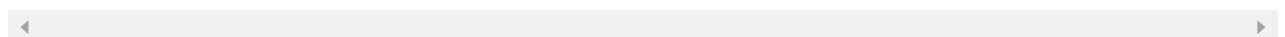
**大妖怪**

2021-05-12

学完之后，我还要再买本自制cpu，然后把我的hello os 跑在我自制的7cm进制的cpu上

展开 ▾

编辑回复: 加油。千里之行，始于足下！



3

4

**尼欧**

2021-05-12

打卡

展开 ▾

编辑回复: 加油, 尼欧同学~

3

4

**Geek\_032c4a**

2021-05-12

请教下各位, 在grub阶段还没有加载系统, 为啥有ext2文件系统, 难道是grub自带的吗

展开 ▾

作者回复: grub自己能被别 文件系统

1

2

**鲍勃恪**

2021-05-12

请问会不会有视频讲解之类的, 还有这个需要自己配一个专门的电脑么, 用虚拟机运行的方式能操作吗?

展开 ▾

编辑回复: 图文+音频讲解哦, 有不懂的可以留言讨论。后续会安虚拟机, 莫着急, 到时会详细说明。

2

2

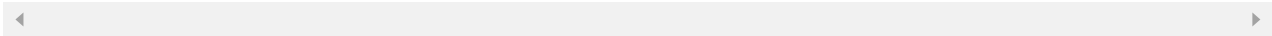
**尼欧**

2021-05-12

舉手, 我要問個問題, `char* p_strdst = (char*)(0xb8000);` //指向显存的开始地址, C語言中通過這句去讀寫顯存的時候, 這裏的0xb8000是段內偏移還是綫性地址呢? 在最終生成可執行文件的時候, C編譯器會對這個0xb8000做什麼轉換麼? 比如轉換成某個段的基地址和段內偏移的形式?

展开 ▾

作者回复: 这是线性地址 汇编代码已经切换到保护模式了



 1

 2