操作系统实验（一）

8086寻址方式和指令系统

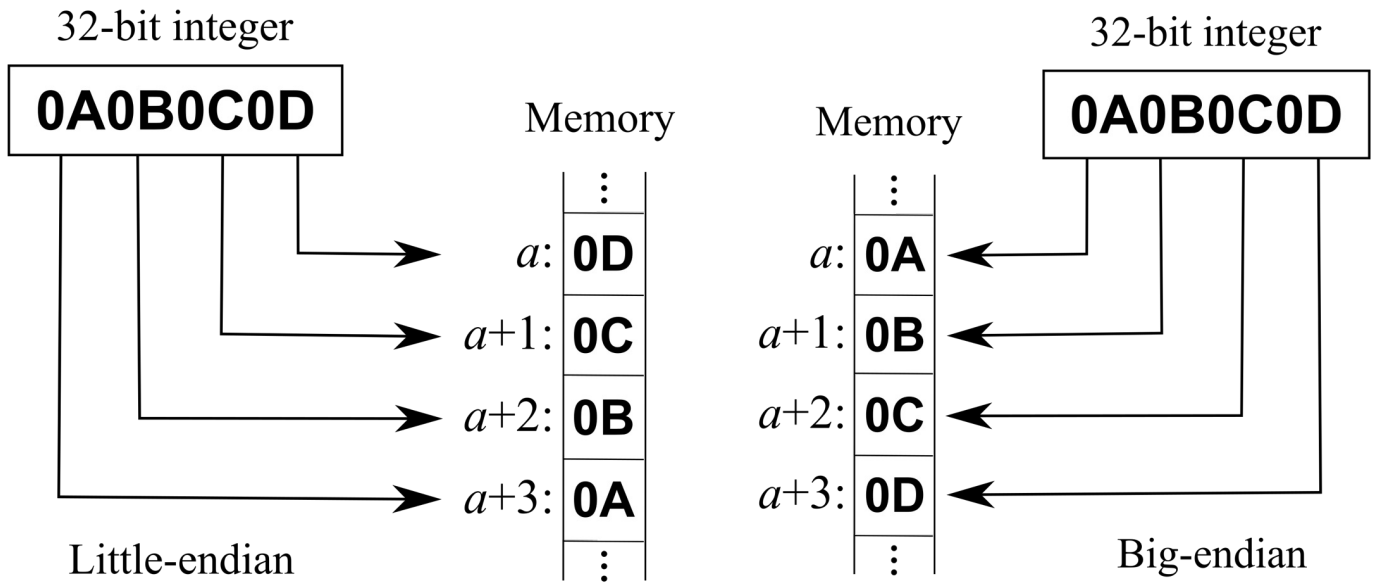
2022年09月26日

80x86简介（1）

- 历史
  - 1978年6月，Intel推出第一款16位微处理器8086，采用20位地址线
  - 1982年发布80286，主频提高至12MHz
  - 1985年发布80386，处理器变为32位，地址线扩展至32位
  - 1989年发布80486，1993年发布80586并命名为奔腾

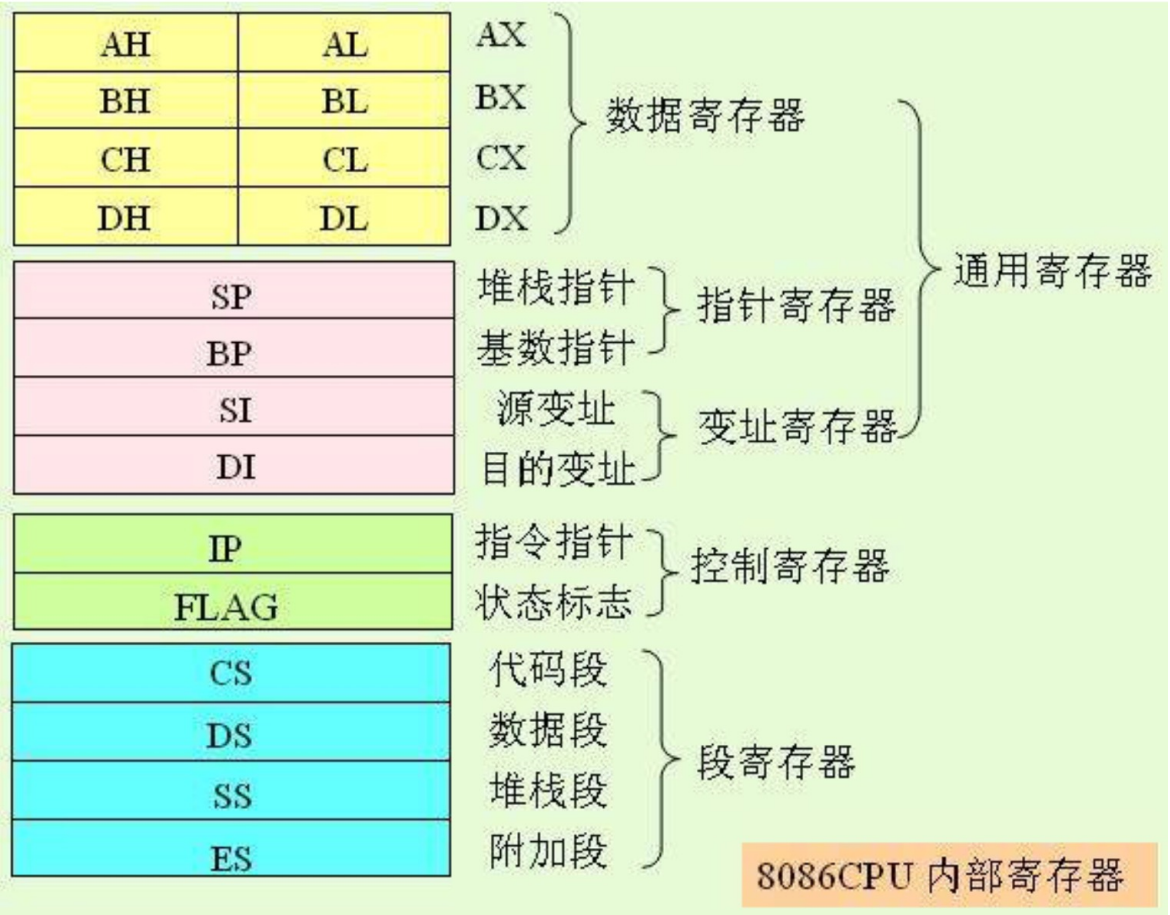
80x86简介（2）

- 特点
  - 采用复杂指令集
  - 小端存储



8086的寄存器

- **SP**: 堆栈指针，与SS配合使用，指向目前的堆栈位置
- **BP**: 基指针寄存器，可用作SS的一个相对基址位置
- **SI**: 源变址寄存器，可用来存放相对于DS段的源变址指针
- **DI**: 目的变址寄存器，可用来存放相对于ES段的目的变址指针



8086的寻址方式（1）

- 寻址
  - 找到操作数的地址(从而能够取出操作数)

- 8086的寻址方式
  - 立即寻址、直接寻址
  - 寄存器寻址、寄存器间接寻址、寄存器相对寻址
  - 基址加变址、相对基址加变址

8086的寻址方式（2）

- 立即寻址
  - `MOV AX 1234H`
  - 直接给出了操作数，事实上没有“寻址”

- 直接寻址
  - `MOV AX [1234H]`
  - 直接给出了地址1234H，用[]符号取数

8086的寻址方式（3）

- 寄存器寻址
  - `MOV AX BX`
  - 操作数在寄存器里，给出寄存器名即可取走操作数
- 寄存器间接寻址
  - `MOV AX [BX]`
  - 操作数有效地址在寄存器之中(SI、DI、BX、BP)
- 寄存器相对寻址
  - `MOV AX [SI+3]`

8086的寻址方式（4）

- 基址加变址
  - `MOV AX [BX+DI]`
  - 把一个基址寄存器(BX、BP)的内容，加上变址寄存器(SI、DI)的内容。

- 相对基址加变址
  - `MOV AX [BX+DI+3]`

如何进行函数传参

- 利用寄存器传递参数
  - 缺点：能传递的参数有限，因为寄存器有限
- 利用约定的地址传递参数
- 利用堆栈传递参数（常用）

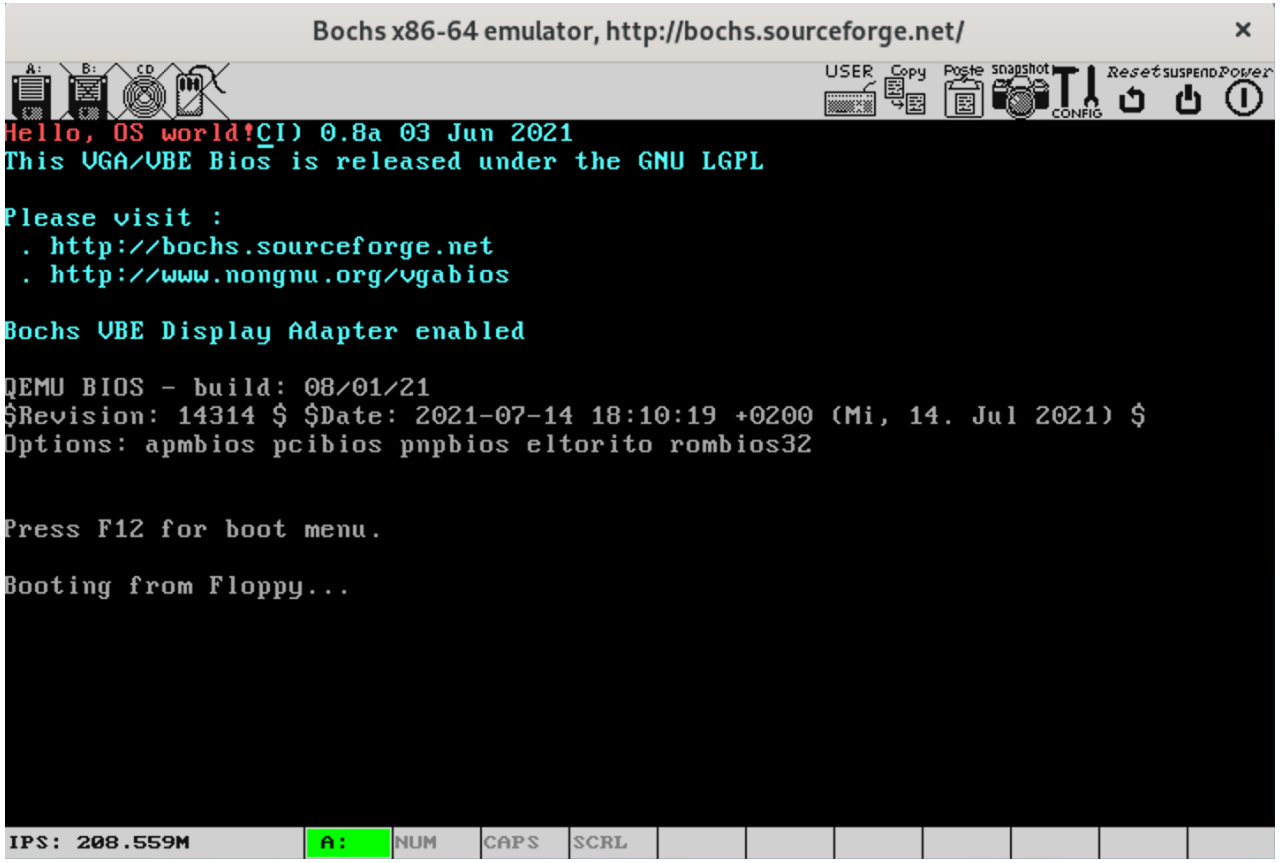
操作系统实验（一）

一个最简操作系统的实现

2022年09月26日

实验任务一

- Hello, OS world!



运行一个操作系统

1.9.2. How do you pronounce "Bochs"?

Phonetically the same as the English word "box". It's just a play on the word "box", since techies like to call their machines a "Linux box", "Windows box", ... Bochs emulates a box inside a box.

- 计算机（虚拟机Bochs）
  - 安装Bochs虚拟机：**`sudo dnf install bochs`**
  - 安装Bochs的GUI库： **`sudo dnf install SDL2`**
  - Ubuntu中使用`apt-get`

- 硬盘（装有操作系统）
  - 创建虚拟软盘映像：`bximage`命令，涉及的配置
    - `fd`：软盘映像
    - `1.44M`：映像大小 1.44MB
    - `a.img`：映像名称（在往映像中写入操作系统时以及配置计算机时会用到）

开始我们自己的操作系统（1）

- boot.asm

```
    org 07c00h          ; 告诉编译器程序加载到7c00处
    mov ax, cs
    mov ds, ax
    mov es, ax
    call    DispStr         ; 调用显示字符串例程
    jmp $           ; 无限循环
DispStr:
    mov ax, BootMessage
    mov bp, ax          ; ES:BP = 串地址
    mov cx, 16          ; CX = 串长度
    mov ax, 01301h      ; AH = 13,  AL = 01h
    mov bx, 000ch       ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
    mov dl, 0
    int 10h         ; 10h 号中断
    ret
BootMessage:        db  "Hello, OS world!"
times   510-($-$$)  db  0   ; 填充剩下的空间，使生成的二进制代码恰好为512
dw  0xaa55              ; 结束标志
```

开始我们自己的操作系统（2）

- 使用NASM来汇编`boot.asm`生成“操作系统”（`boot.bin`）的二进制代码。

- 首先安装nasm工具：
  - Fedora: **`sudo dnf install nasm`**
  - Ubuntu: **`sudo apt-get install nasm`**
  - macOS: **`brew install nasm`**
- 生成boot.bin：
  - **`nasm boot.asm -o boot.bin`**

开始我们自己的操作系统（3）

- 有了“计算机”，也有了“软盘”，以及“操作系统”（boot.bin），是时候将boot.bin写入“软盘”了。
- 我们需要把boot.bin放在软盘的第一个扇区，为什么?
- 需要理解BIOS与OS的加载。

BIOS

- 开机，从ROM运行BIOS程序，BIOS是厂家写好的。
- BIOS程序检查软盘0面0磁道1扇区，如果扇区以0xaa55结束，则认定为引导扇区，将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。
- 以上的0xaa55以及07c00都是一种约定，BIOS程序就是这样做的，所以我们就需要把我们的OS放在软盘的第一个扇区，填充，并在最末尾写入0xaa55。

```
17  times   510-($-$$)  db  0    ; 填充剩下的空间，使生成的二进制代码恰好为512字节
18  dw  0xaa55                   ; 结束标志
```

写入boot.bin

- 使用dd命令将boot.bin写入软盘中
  - **dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc**
    - if：代表输入文件
    - of：代表输出设备
    - bs：代表一个扇区大小
    - count：代表扇区数
    - conv：代表不作其它处理

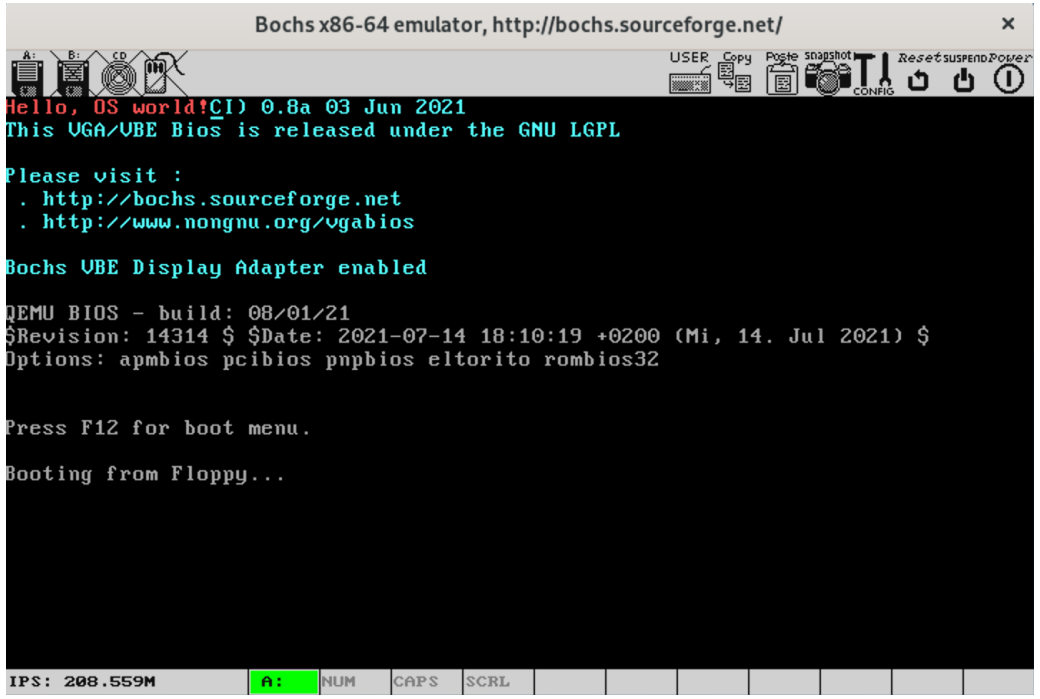
启动虚拟机（1）

- 启动还需要Bochs的配置文件。告诉Bochs，你希望你的虚拟机是什么样子的，如内存多大，硬盘映像和软盘映像都是哪些文件等内容。
  - `display_library`：Bochs使用的GUI库
  - `megs`：虚拟机内存大小 (MB)
  - `floppya`：虚拟机外设，软盘为`a.img`文件
  - `boot`：虚拟机启动方式，从软盘启动

```
megs:32
display_library: sdl2
floppya: 1_44=a.img, status=inserted
boot: floppy
```

启动虚拟机（2）

- 配置文件保存为`bochsrc`，和`a.img`以及`boot.bin`放在同一目录下。
- 启动Bochs：`bochs -f bochsrc`



org 07c00h (1)

- 为什么需要org 07c00h这一行代码?
  - BIOS将软盘内容放在07c00h位置，并不是由这行代码决定的。

- org是伪指令
  - 伪指令是指，不生成对应的二进制指令，只是汇编器使用的。也就是说，boot.bin文件里面，没有07c00h这个东西，BIOS不是因为这条指令才把代码放在07c00h的。
  - mov这种指令，就会生成二进制代码，可以直接告诉CPU该做什么。

org 07c00h (2)

- `org 07c00h`的作用：告诉汇编器，当前这段代码会放在`07c00h`处。所以，如果之后遇到需要绝对寻址的指令，那么绝对地址就是`07c00h`加上相对地址。
  - 绝对地址：内存的实际位置（先不考虑内存分页一类逻辑地址）。
  - 相对地址：当前指令相对第一行代码的位置。

- 在第一行加上`org 07c00h`只是让编译器从相对地址`07c00h`处开始编译第一条指令，相对地址被编译加载后就正好和绝对地址吻合。

org 07c00h (3)

- 如果去掉这一行会发生什么?
- 反编译boot1.bin
- **ndisasm boot1.bin -o 0**

```
00000000  8CC8              mov ax,cs
00000002  8ED8              mov ds,ax
00000004  8EC0              mov es,ax
00000006  E80200            call word 0xb
00000009  EBFE              jmp short 0x9
0000000B  B81E00            mov ax,0x1e
0000000E  89C5              mov bp,ax
00000010  B91000            mov cx,0x10
00000013  B80113            mov ax,0x1301
00000016  BB0C00            mov bx,0xc
00000019  B200              mov dl,0x0
0000001B  CD10              int 0x10
0000001D  C3                ret
0000001E  48                dec ax
```

0000000B B81E00 mov ax,0x1e

取字符串，由之前所说，第一行可执行代码加载到内存的0x7c00处，故字符串地址应该为0x7c00+0x1e

org 07c00h (4)

- 修改boot1.asm

```
 mov ax,cs
 mov ds,ax
 mov es,ax
 call DispStr
 jmp $
DispStr:
 mov ax,BootMessage+07c00h
 mov bp,ax
 mov cx,16
 mov ax,01301h
 mov bx,000ch
 mov dl,0
 int 10h
 ret
BootMessage:    db "Hello, OS world!"
times  510-($-$$) db 0
dw 0xaa55
```

```
000  8CC8              mov ax,cs
00000002  8ED8         mov ds,ax
00000004  8EC0         mov es,ax
00000006  E80200       call word 0xb
00000009  EBFE         jmp short 0x9
0000000B  B81E7C       mov ax,0x7c1e
0000000E  89C5         mov bp,ax
00000010  B91000       mov cx,0x10
00000013  B80113       mov ax,0x1301
00000016  BB0C00       mov bx,0xc
00000019  B200         mov dl,0x0
0000001B  CD10         int 0x10
0000001D  C3           ret
```

OS的加载

- 首先回忆一下我们在BIOS中提到的“操作系统”的加载过程：
  - BIOS程序检查软盘0面0磁道1扇区，如果扇区以`0xaa55`结束，则认定为引导扇区，将其512字节的数据加载到内存的`07c00`处，然后设置PC，跳到内存`07c00`处开始执行代码。

- 其实我们所提到的只是真正的操作系统内核加载过程中的第一步，真正的加载过程大概是这样：“引导-加载内核入内存-跳入保护模式-开始执行内核”，那是不是只要我们将加载内核入内存的代码写入我们之前写的`boot.asm`中就行了呢？

突破512字节的限制

- 其实，除了加载内核，我们要做的事还有准备保护模式（等讲到保护模式后再来理解）等，512字节显然不够，为了突破512字节的限制，我们引入另外一个重要的文件`loader.asm`，引导扇区只负责把loader加载入内存并把控制权交给他，这样将会灵活得多。

- 最终，由loader将内核kernel加载入内存，才开始了真正操作系统内核的运行。

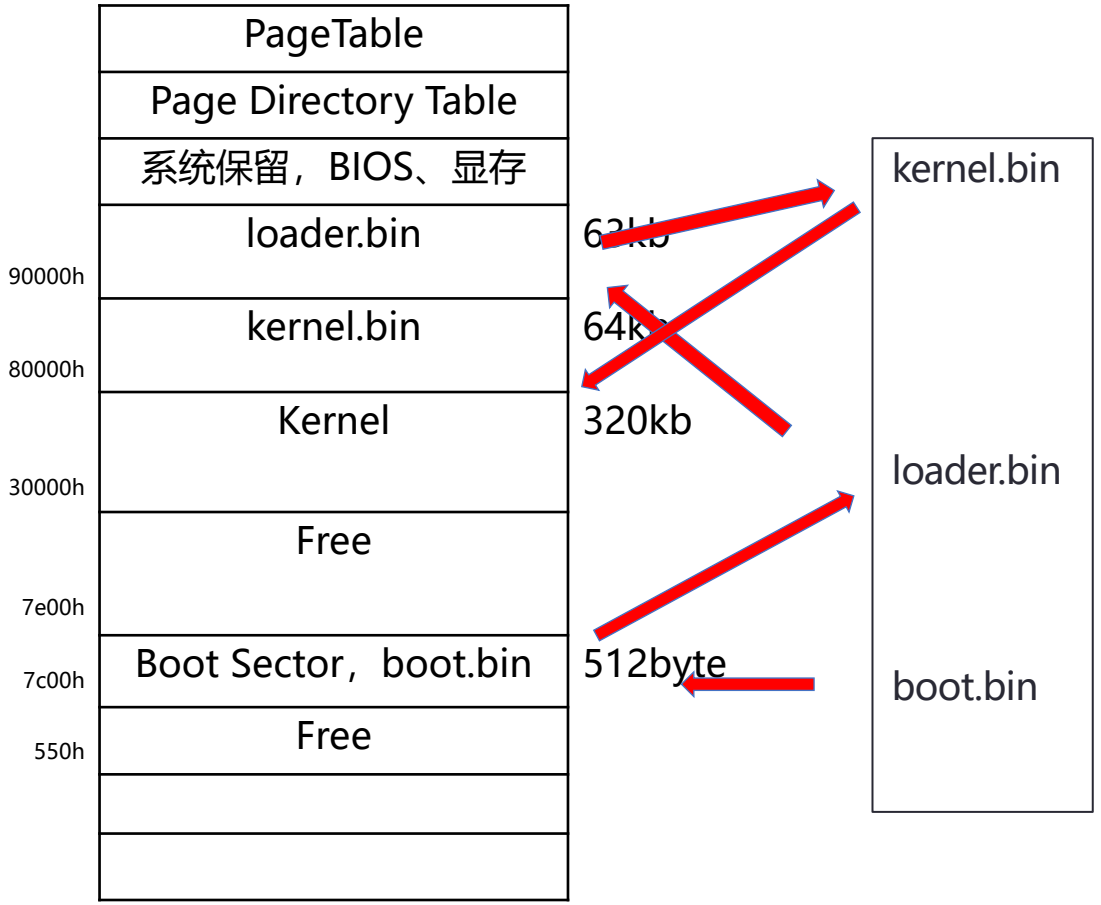
Loader

- 跳入保护模式
  - 最开始的x86处理器16位，寄存器用ax, bx等表示，称为实模式。后来扩充成32位，eax，ebx等，为了向前兼容，提出了保护模式
  - 必须从实模式跳转到保护模式，才能访问1M以上的内存。
- 启动内存分页
- 从kernel.bin中读取内核，并放入内存，然后跳转到内核所在的开始地址，运行内核
  - 跟boot类似，使用汇编直接在软盘下搜索kernel.bin
  - 但是，不能把整个kernel.bin放在内存，而是要以ELF文件的格式读取并提取代码。

Kernel

- 这才是真正的操作系统
- 内存管理，进程调度，图像显示，网络访问等等，都是内核的功能。
- 内核的开发使用高级语言
  - C语言可以更高效的编写内核
  - 但我们是在操作系统层面上编写另一个操作系统，于是生成的内核可执行文件是和当前操作系统平台相关的。比如Linux下是ELF格式，有许多无关信息，于是，内核并不能像`boot.bin`或`loader.bin`那样直接放入内存中，需要loader从`kernel.bin`中提取出需要放入内存中的部分。

启动流程和内存分布


PageTable
Page Directory Table
系统保留，BIOS、显存
loader.bin
90000h
kernel.bin
80000h
Kernel
30000h
Free
7e00h
Boot Sector，boot.bin
7c00h
Free
550h
320kb
512byte
kernel.bin
loader.bin
boot.bin

操作系统实验（一）

总结

2022年09月26日

环境搭建（1）

- Linux（以Fedora 36为例）
  - 安装GCC（一般情况下系统自带）
  - 安装NASM
  - 安装虚拟机Bochs
  - 安装GUI库SDL2
  - 在终端使用DNF安装上述内容（Ubuntu使用APT-GET）
    - `sudo dnf install nasm bochs SDL2`
- macOS
  - 在终端使用Homebrew安装上述内容
    - `brew install nasm bochs sdl2`

环境搭建（2）

- Windows
  - 安装虚拟机（如VirtualBox，VMWare）
  - 在虚拟机中安装Linux
  - 接下和Linux中搭建环境一样
- 说明
  - 实验任务一（Hello OS world）在Bochs环境中运行，因此无论什么操作系统、架构，只要支持Bochs就行。
  - 实验任务二（大数运算）在本机运行，而不是在Bochs中运行。
    - 装载Apple Silicon芯片的Mac*可能*会在汇编、运行时出现问题，建议使用X86-64的机器完成实验。

具体步骤

- Step 1. 使用NASM汇编`boot.asm`生成“操作系统”的二进制代码
  - **`nasm boot.asm -o boot.bin`**
- Step 2. 使用`bximage`命令生成虚拟软盘.
  - **`bximage -> fd -> 1.44 -> a.img`**
- Step 3. 使用`dd`命令将操作系统写入软盘
  - **`dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc`**
- Step 4. 配置bochs
- Step 5. 启动bochs
  - **`bochs -f bochsrc`**
  - 如果一开始没显示，说明是debug模式，那么只需要按c即可显示。

评分说明

- 实验任务一总分1分。
- 助教检查时，只保留`boot.asm`和`bochsrc`文件，在不看本讲义的情况下复现实验。
- 如果需要助教提示或查看讲义才能复现实验，酌情扣0.5～1分。