> - 最后编辑: 2020-2-24

1. 实验要求

本实验通过实现一个简单的引导程序，介绍系统启动的基本过程

1.1. 在实模式下实现一个Hello World程序

在实模式下在终端中打印`Hello, World!`

1.2. 在保护模式下实现一个Hello World程序

从实模式切换至保护模式，并在保护模式下在终端中打印`Hello, World!`

1.3. 在保护模式下加载磁盘中的Hello World程序运行

从实模式切换至保护模式，在保护模式下读取磁盘1号扇区中的Hello World程序至内存中的相应位置，跳转执行该Hello World程序，并在终端中打印`Hello, World!`

2. 相关资料

2.1. CPU、内存、BIOS、磁盘、主引导扇区、加载程序、操作系统

最开始先讨论这样一个问题：

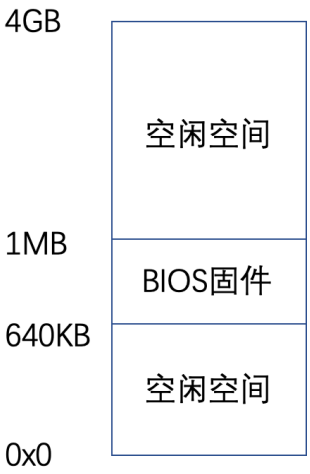
> CPU在加电之后，它的第一条指令在哪?

我们知道，CPU在电源稳定后会将内部的寄存器初始化成某个状态，然后执行第一条指令。第一条指令在哪？答案是**内存**

内存是用来存数据的，但是有过了解的同学都知道，断电后内存中的内容会丢失。那上哪找第一条指令

其实内存除了我们说的内存条这种RAM，还有ROM；在i386机器刚启动时，内存的地址划分如下

- **基本内存** 占据0～640KB地址空间
- **上位内存** 占据640KB～1024KB地址空间。分配给显示缓冲存储器、各适配卡上的ROM和系统ROM BIOS。（这个区域的地址分配给ROM，相应的384KB的RAM被屏蔽掉）
- **扩展内存** 占据1MB以上地址空间



不管是i386还是i386之前的芯片，在加电后的第一条指令都是跳转到BIOS固件进行开机自检，然后将磁盘的主引导扇区（Master Boot Record, MBR；0号柱面，0号磁头，0号扇区对应的扇区，512字节，末尾两字节为魔数`0x55`和`0xaa`）加载到0x7c00。

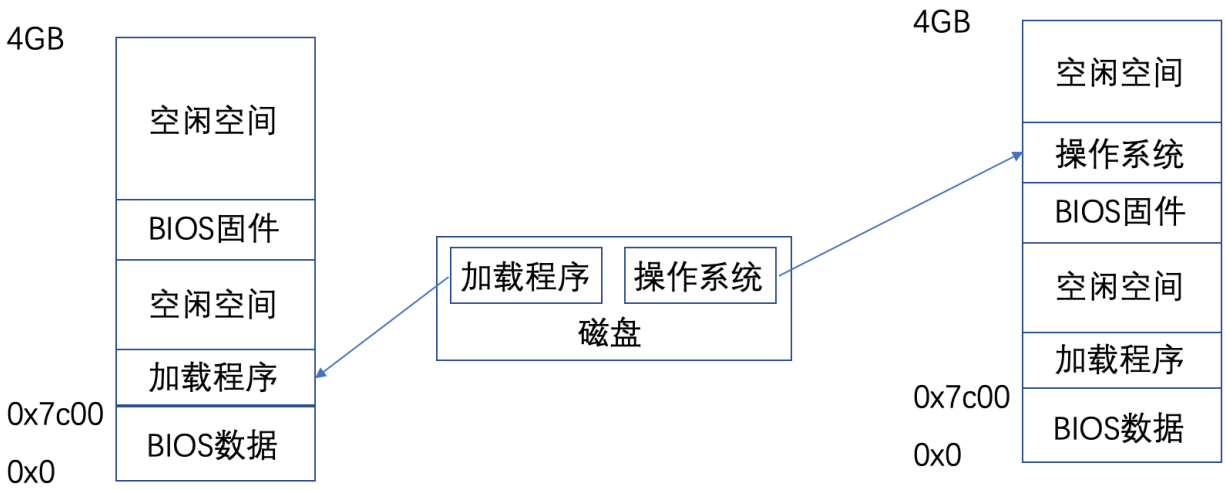
查看磁盘的MBR

在Linux中, 你可以很容易地查看磁盘的MBR(需要root权限):

```
head -c 512 /dev/sda | hd
```

你可以在输出结果的末尾看到魔数`0x55`和`0xaa`.

有了这个魔数, BIOS就可以很容易找到可启动设备了: BIOS依次将设备的首扇区加载到内存`0x7c00`的位置, 然后检查末尾两个字节是否为`0x55`和`0xaa`. `0x7c00`这个内存位置是BIOS约定的, 如果你希望知道为什么采用`0x7c00`, 而不是其他位置, 这里可以给你提供一些线索. 如果成功找到了魔数, BIOS将会跳到`0x7c00`的内存位置, 执行刚刚加载的启动代码, 这时BIOS已经完成了它的使命, 剩下的启动任务就交给MBR了; 如果没有检查到魔数, BIOS将会尝试下一个设备; 如果所有的设备都不是可启动的, BIOS将会发出它的抱怨: "找不到启动设备".



BIOS加载主引导扇区后会跳转到`CS:IP=0x0000:0x7c00`执行加载程序，这就是我们操作系统实验**开始**的地方。在我们目前的实验过程中，主引导扇区和加载程序（bootloader）其实代表一个东西。但是现代操作系统中，他们往往不一样，请思考一下为什么？

主引导扇区中的加载程序的功能主要是

- 将操作系统的代码和数据从磁盘加载到内存中
- 跳转到操作系统的起始地址

其实真正的计算机的启动过程要复杂很多，有兴趣请自行了解。

> 你弄清楚本小结标题中各种名词的含义和他们间的关系了吗？请在实验报告中阐述。

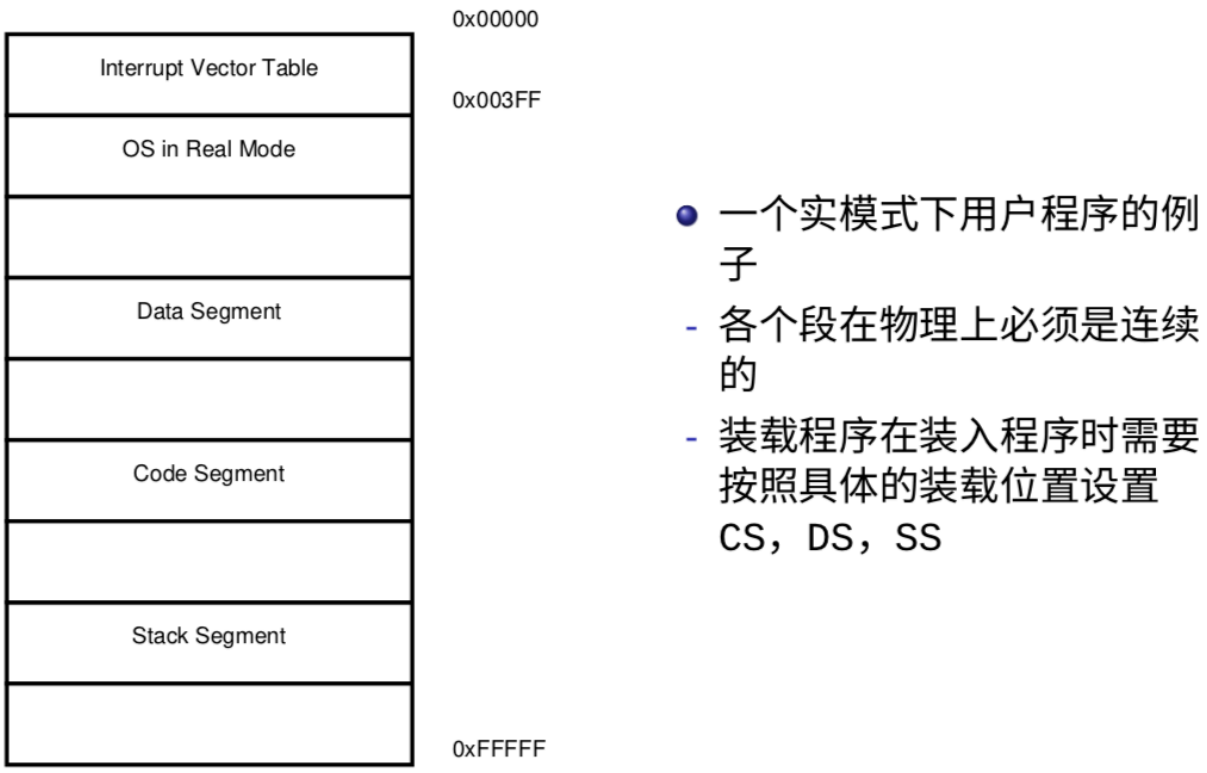
2.2. IA-32的存储管理

在IA-32下，CPU有两种工作模式：源于8086的实模式与源于80386的保护模式；

2.2.1. 实模式简介

8086为16位CPU，有16位的寄存器（Register），16位的数据总线（Data Bus），20位的地址总线（Address Bus），寻址能力为1MB

- 8086 的寄存器集合
  - 通用寄存器(16 位): AX，BX，CX，DX，SP，BP，DI，SI
  - 段寄存器(16 位): CS，DS，SS，ES
  - 状态和控制寄存器(16 位): FLAGS，IP
- 寻址空间与寻址方式
  - 采用实地址空间进行访存，寻址空间为 `2^20`
  - 物理地址 = 段寄存器 << 4 + 偏移地址
  - CS=0x0000:IP=0x7C00 和 CS=0x0700:IP=0x0C00 以及 CS=0x7C0:IP=0x0000 所寻地址是完全一致的



- 8086的中断
  - 中断向量表存放在物理内存的开始位置(0x0000至0x03FF)
  - 最多可以有 256 个中断向量

- 0x00 至 0x07 号中断为系统专用
- 0x08 至 0x0F，0x70 至 0x77 号硬件中断为 8259A 使用

8086的中断处理是交给BIOS完成的，这也是为什么我们看到**2.1节**内存0x0处是BIOS数据。

实模式下可以通过`int $0x10`中断进行屏幕上的字符串显示，具体细节请参考BIOS中断向量表或自行查找资料。

实模式或者说8086本身有一些缺点

- 安全性问题
  - 程序采用物理地址来实现访存，无法实现对程序的代码和数据的保护
  - 一个程序可以通过改变段寄存器和偏移寄存器访问并修改不属于自己的代码和数据
- 分段机制本身的问题
  - 段必须是连续的，从而无法利用零碎的空间
  - 段的大小有限制(最大为 64KB)，从而限制了代码的规模

2.2.2. 保护模式

80386开始，Intel处理器步入32位CPU；80386有32位地址线，其寻址空间为`2^32=4GB`；为保证兼容性，实模式得以保留，PC启动时CPU工作在实模式，并由Bootloader迅速完成从实模式向保护模式的切换

- 保护模式带来的变化
  - 通用寄存器(从 16 位扩展为 32 位): EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP
  - 段寄存器(维持 16 位): CS，DS，SS，ES，FS，GS
  - 状态和控制寄存器(32/64 位): EFLAGS，EIP，CR0，CR1， CR2，CR3
  - 系统地址寄存器: GDTR，IDTR，TR，LDTR
  - 调试与测试用寄存器: DR0，···，DR7，TR0，···，TR7

	8086 寄存器	80386 寄存器
通用寄存器	AX，BX，CX，DX SP，BP，DI，SI	EAX，EBX，ECX，EDX ESI，EDI，EBP，ESP
段寄存器	CS，DS，SS，ES	CS，DS，SS，ES，FS，GS
段描述符寄存器	无	对程序员不可见
状态和控制寄存器	FLAGS，IP	EFLAGS，EIP CR0，CR1，CR2，CR3
系统地址寄存器	无	GDTR，IDTR，TR，LDTR
调试寄存器	无	DR0，···，DR7
测试寄存器	无	TR0，···，TR7

- 寻址方式的变化
  - 在保护模式下，分段机制是利用一个称作段选择子（Selector）的偏移量到全局描述符表中找到需要的段描述符，而这个段描述符中就存放着真正的段的物理首地址，该物理首地址加上偏离量即可得到最后的物理地址
  - 一般保护模式的寻址可用 0xMMMM:0xNNNNNNNN 表示，其中 0xMMMM 表示段选择子

的取值，16 位(其中高 13 位表示其对应的段描述符在全局描述符表中的索引，低 3 位表示权限等信息)，0xNNNNNNNN 表示偏移量的取值，32 位
    - 段选择子为 CS，DS，SS，ES，FS，GS 这些段寄存器

全局描述符表（Global Descriptor Table），即GDT，GDT中由一个个被称为段描述符的表项组成，表项中定义了段的起始32位物理地址，段的界限，属性等内容；段描述符的结构如下图所示

```
       DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

 31                23                15                7               0
+-----------------+-+-+-+-+---------+-+-----+-+-----+-+-----------------+
|                 | |D| |A|         | |     |S|     | |                 |
|   BASE 31..24   |G|/|O|V| LIMIT   |P| DPL | | TYPE|A|  BASE 23..16    | 4
|                 | |B| |L| 19..16  | |     |1|     | |                 |
|-----------------+-+-+-+-+---------+-+-----+-+-----+-+-----------------|
|                                   |                                   |
|        SEGMENT BASE 15..0         |       SEGMENT LIMIT 15..0         | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

               DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS

 31                23                15                7               0
+-----------------+-+-+-+-+---------+-+-----+-+-------+-----------------+
|                 | | | |A|         | |     |S|       |                 |
|   BASE 31..24   |G|X|O|V| LIMIT   |P| DPL | |  TYPE |  BASE 23..16    | 4
|                 | | | |L| 19..16  | |     |0|       |                 |
|-----------------+-+-+-+-+---------+-+-----+-+-------+-----------------|
|                                   |                                   |
|        SEGMENT BASE 15..0         |       SEGMENT LIMIT 15..0         | 0
|                                   |                                   |
+-----------------+-----------------+-----------------+-----------------+

          A      - ACCESSED
          AVL    - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
          DPL    - DESCRIPTOR PRIVILEGE LEVEL
          G      - GRANULARITY
          P      - SEGMENT PRESENT
```

- 每个段描述符为 8 个字节，共 64 位
- 段基址为第 2，3，4，7 字节，共 32 位
- 段限长为第 0，1 字节及第 6 字节的低 4 位，共 20 位，表示该段的最大长度
- 当属性 G 为 0 时，20 位段限长为实际段的最大长度(最大为 1MB)；当属性 G 为 1 时，该 20 位段限长左移 12 位后加上 0xFFF 即为实际段的最大长度(最大为 4GB)
- D/B: 对于不同类型段含义不同
    - 在可执行代码段，该位叫做 D 位，D 为 1 使用 32 位地址和 32/8 位操作数，D 为 0 使用 16 位地址和 16/8 位操作数

    - 在向下扩展的数据段中，该位叫做 B 位，B 为 1 段的上界为 4GB，B 为 0 段的上界为 64KB
    - 在描述堆栈段的描述符中，该位叫做 B 位，B 为 1 使用 32 位 操作数，堆栈指针用 ESP，B 为 0 使用 16 位操作数，堆栈指针用 SP
- AVL：Available and Reserved Bit，通常设为 0
- P：存在位，P 为 1 表示段在内存中
- DPL：描述符特权级，取值 0-3 共 4 级;0 特权级最高，3 特权级最低，表示访问该段时 CPU 所处于的最低特权级，后续实验会详细讨论
- S：描述符类型标志，S 为 1 表示代码段或数据段，S 为 0 表示系统段(TSS，LDT)和门描述符
- TYPE：当 S 为 1，TYPE 表示的代码段，数据段的各种属性 如下表所示

bit 3	Data/Code	0 (data)
bit 2	Expand-down	0 (normal) 1 (expand-down)
bit 1	Writable	0 (read-only) 1 (read-write)
bit 0	Accessed	0 (hasn't) 1 (accessed)

bit 3	Data/Code	1 (code)
bit 2	Conforming	0 (non-conforming) 1 (conforming)
bit 1	Readable	0 (no) 1 (readable)
bit 0	Accessed	0 (hasn't) 1 (accessed)

为进入保护模式，需要在内存中开辟一块空间存放GDT表；80386提供了一个寄存器`GDTR`用来存放GDT的32位物理基地址以及表长界限；在将GDT设定在内存的某个位置后，可以通过`LDGT`指令将GDT的入口地址装入此寄存器

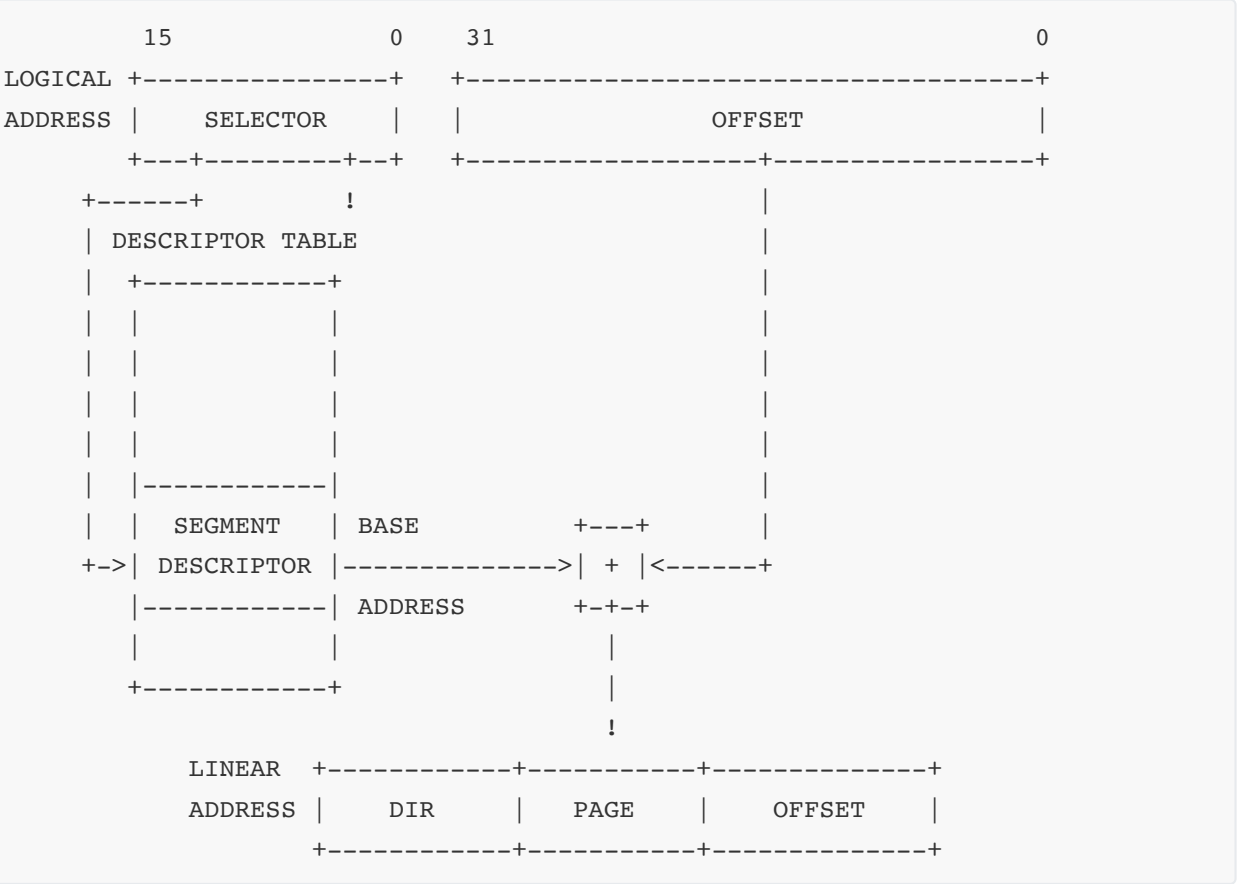
```
          GDT REGISTER

                  15             0
                 +---------------+
                 |   GDT LIMIT   |
+----------------+---------------|
|            GDT BASE            |
+--------------------------------+
 31                              0
```

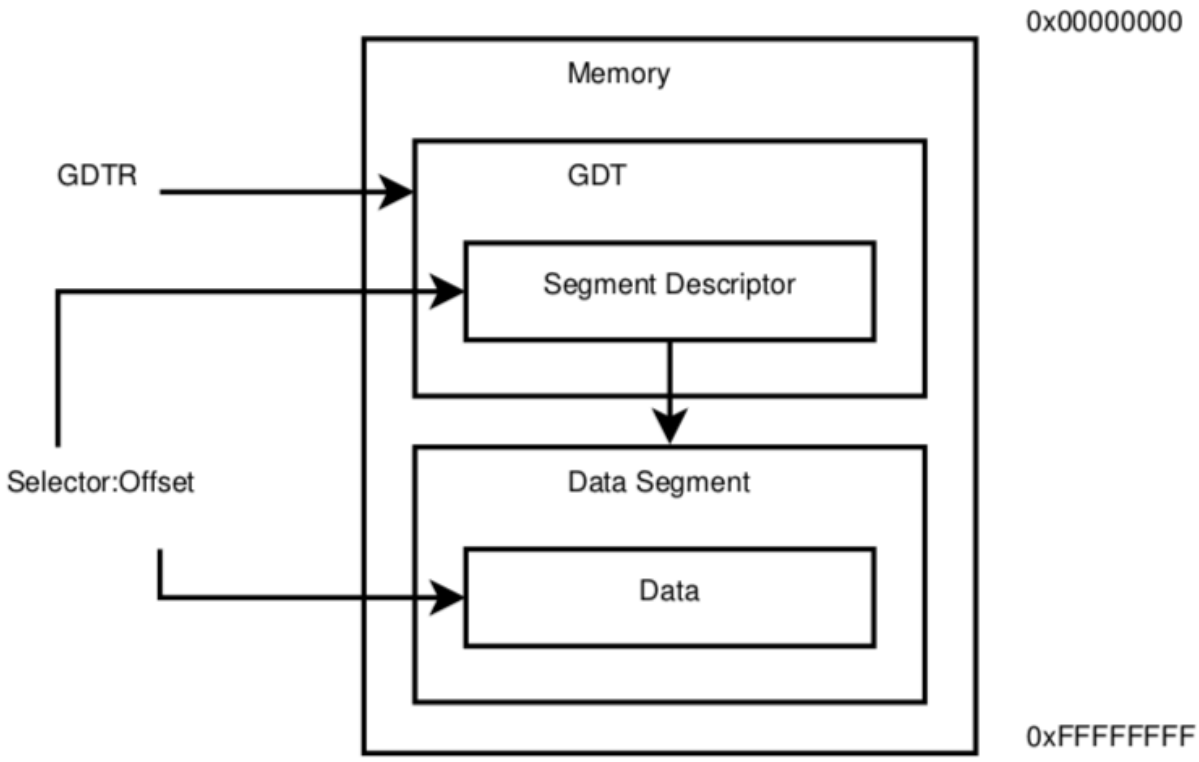
由`GDTR`访问GDT是由段选择子来完成的；为访问一个段，需将段选择子存储入段寄存器，比如数据段选择子存储入`DS`，代码段选择子存储入`CS`；其数据结构如下

```
 15                      3   1 0
+-------------------------+-+---+
|                         |T|   |
|           INDEX         | |RPL|
|                         |I|   |
+-------------------------+-+---+

 TI  - TABLE INDICATOR, 0 = GDT, 1 = LDT
 RPL - REQUESTOR'S PRIVILEGE LEVEL
```

TI位表示该段选择子为全局段还是局部段，PRL表示该段选择子的特权等级，13位Index表示描述符表中的编号

Selector:Offset表示的逻辑地址可如下图所示转化为线性地址，倘若不采用分页机制，则该线性地址即物理地址





2.3. AT&T汇编

参考`AT&T汇编语言.ppt`（有时间可以讲）

2.4. 显存映射

前文我们提到，在实模式下可以通过BIOS中断在屏幕上显示文字，切换到保护模式后有一个令人震惊的事实：**切换到32位保护模式的时候，我们不能再使用 BIOS 了**。

切换到32位碰到的第一个问题是如何在屏幕上打印信息。之前我们请求 BIOS 在屏幕上打印一个 ASCII 字符，但是它是如何做到将合适的像素展示在计算机屏幕恰当的位置上的呢？目前，只要知道显示设备可以用很多种方式配置成两种模式：文本模式和图像模式。屏幕上展示的内容只是某一特定区域的内存内容的视觉化展示。所以为了操作屏幕的展示，我们必须在当前的模式下管理内存的某特定区域。显示设备就是这样子的一种设备，和内存相互映射的硬件。

当大部分计算机启动时候，虽然它们可能有更先进的图像硬件，但是它们都是先从简单的视频图像数组（VGA，video graphics array）颜色文本模式，尺寸80*25，开始的。在文本模式，编码人员不需要为每个字符渲染每一个独立的像素点，因为一个简单的字体已经在 VGA 显示设备内部内存中定义了。每一个屏幕上字符单元，在内存中通过两字节表示，第一个字节被展示字符的 ASCII 编码，第二个字节包含字符的一些属性，比如字符的前景色和背景色，字符是否应该闪烁等。

所以，如果我们想在屏幕上展示一个字符，那么我们需要为当前的 VGA 模式，在正确的内存地址处设置一个 ASCII 码值，通常这个地址是 `0xb8000`。

3. 解决思路

3.1. 实模式Hello World程序

通过陷入屏幕中断调用BIOS打印字符串`Hello, World!`

```
movw $message, %ax
movw %ax, %bp
movw $13, %cx                           #打印的字符串长度
movw $0x1301, %ax                       #AH=0x13 打印字符串
movw $0x000c, %bx                       #BH=0x00 黑底 BL=0x0c 红字
movw $0x0000, %dx                       #在第0行0列开始打印
int $0x10                               #陷入0x10号中断

message:
       .string "Hello, World!"
```

屏幕中断的相关内容可以查阅`BIOS中断表.xlsx`

通过写显存打印字符`H`

```
movl $((80*5+0)*2), %edi                #在第5行第0列打印
movb $0x0c, %ah                         #黑底红字
movb $42, %al                           #42为H的ASCII码
movw %ax, %gs:(%edi)                    #写显存
```

3.2. 实模式切换保护模式

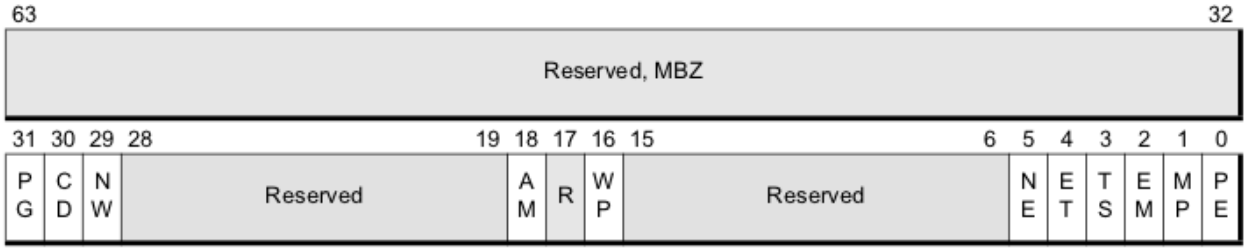
关闭中断，打开A20数据总线，加载`GDTR`，设置`CR0`的PE位（第0位）为`1b`，通过长跳转设置`CS`进入保护模式，初始化`DS`，`ES`，`FS`，`GS`，`SS`

这里设置了三个GDT表项，其中代码段与数据段的基地址都为`0x0`，视频段的基地址为`0xb8000`

```
.code16
start:
        cli                                 #关闭中断
        inb $0x92, %al                      #启动A20总线
        orb $0x02, %al
        outb %al, $0x92
        data32 addr32 lgdt gdtDesc          #加载GDTR
        movl %cr0, %eax                     #启动保护模式
        orb $0x01, %al
        movl %eax, %cr0                     #设置CR0的PE位（第0位）为1
        data32 ljmp $0x08, $start32         #长跳转切换至保护模式

.code32
start32:
        ...                                 #初始化DS ES FS GS SS 初始化栈顶指针ESP
        jmp bootMain                        #跳转至bootMain函数 定义于boot.c

gdt:
        .word 0,0                           #GDT第一个表项必须为空
        .byte 0,0,0,0

        .word 0xffff,0                      #代码段描述符
        .byte 0,0x9a,0xcf,0

        .word 0xffff,0                      #数据段描述符
        .byte 0,0x92,0xcf,0

        .word 0xffff,0x8000                 #视频段描述符
        .byte 0x0b,0x92,0xcf,0
        ...

gdtDesc:
        .word (gdtDesc - gdt -1)
        .long gdt
```

cr0寄存器

CR0寄存器的结构如下图所示:



Bits	Mnemonic	Description	R/W
63–32	Reserved	Reserved, Must be Zero	
31	PG	Paging	R/W
30	CD	Cache Disable	R/W
29	NW	Not Writethrough	R/W
28–19	Reserved	Reserved	
18	AM	Alignment Mask	R/W
17	Reserved	Reserved	
16	WP	Write Protect	R/W
15-6	Reserved	Reserved	
5	NE	Numeric Error	R/W
4	ET	Extension Type	R
3	TS	Task Switched	R/W
2	EM	Emulation	R/W
1	MP	Monitor Coprocessor	R/W
0	PE	Protection Enabled	R/W

CR0的PE位(protection enable)即保护位表示是否开启了段级保护, 一旦置为1, 就表示开启了保护模式, 而开机加电时默认是置为0的. 同理如果要开启分页机制, 就需要把PG位, 即分页标志位置为1, 此次不要求分页, 对CR0寄存器感兴趣的同学可以课后深入探索.

3.3. 加载磁盘中的程序并运行

由于中断关闭，无法通过陷入磁盘中断调用BIOS进行磁盘读取，本次实验提供的代码框架中实现了`readSec(void *dst, int offset)`这一接口（定义于`bootloader/boot.c`中），其通过读写（`in`，`out`指令）磁盘的相应端口（Port）来实现磁盘特定扇区的读取

通过上述接口读取磁盘MBR之后扇区中的程序至内存的特定位置并**跳转执行**（注意代码框架`app/Makefile`中设置的该Hello World程序入口地址）

4. 代码框架

本次实验提供一个示范代码框架

```
lab1-STUID                          #待修改
├── lab1
│   ├── Makefile
│   ├── app
│   │   ├── Makefile
│   │   └── app.s                   #用户程序
│   ├── bootloader
│   │   ├── Makefile
│   │   ├── boot.c                  #加载磁盘上的用户程序
│   │   ├── boot.h                  #磁盘I/O接口
```

```
│   │   └── start.s                    #引导程序
│   └── utils
│       └── genboot.pl                 #生成MBR
└── report
    └── 171220000.pdf                  #待替换
```

5. 作业提交

- 本次作业仅需提交保护模式下加载磁盘中的Hello World程序并运行的相关源码.
- **请大家在提交的实验报告中注明你的邮箱, 方便我们及时给你一些反馈信息.**
- **学术诚信**: 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 请你在实验截止前务必确认你提交的内容符合要求(格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除一定的分数, 最高可达50%.
- 实验不接受迟交，一旦迟交按**学术诚信**给分.
- 其他问题参看`index.pdf`中的**作业规范与提交**一章
- **本实验最终解释权归助教所有**

截止时间：2020-3-9 23:59:59