

- [lab1实验报告](#)
  - [lab1.1](#)
  - [lab1.2](#)
  - [lab1.3](#)
  - [思考题](#)
    - [计算机的启动过程](#)
  - [实验结果图](#)

# lab1实验报告

---

231220087 蒋于欣 231220087@smail.nju.edu.cn

实验进度：完成了所有内容。

## lab1.1

---

代码思路：

- 初始化段寄存器
- 设置栈指针为0x7d00
- 重新设置时钟中断处理程序: 将 中断服务例程（ISR）TIMER\_ISR 的地址存储到 0x70（偏移量）
- 设置中断向量表：定时器中断对应的是中断号0x20（IRQ0）。
- 设置8253定时器：每隔20ms左右触发一次时钟中断
  - 将 定时产生中断命令 字节(0x36，表示计数器0，模式3)发送到端口0x43
  - 将 20ms 对应的计数值的字节发送到端口0x40
- **sti**启用中断

编写中断服务例程 TIMER\_ISR：通过陷入屏幕中断调用BIOS打印一行”Hello, World!”，并且间隔1000ms打印新的一行。思路如下：

- 保存寄存器现场
- 循环，每次计数器+1
  - 若计数器<50，则转到end\_itr：发送EOI，弹出寄存器的值，iret返回
  - 若计数器=50，则将计数器归0，调用 **print\_message**:
    - 保存 AX 和 DX 寄存器现场
    - AH = 0x0E：BIOS teletype 功能，允许直接在屏幕上打印字符。

- SI = message: 将字符串 message 的地址加载到 SI 作为数据源。
- **print\_loop**循环:
  - lodsb: 从 SI 指向的内存位置取出一个字节 (字符) 到 AL, 同时 SI 自动递增。
  - testb %al, %al: 检查 AL 是否为 0 (即字符串结束符 \0)。
    - 若 AL 为 0, 结束循环。
    - 否则, 继续执行 int \$0x10 调用 BIOS 中断 0x10/0x0E, 打印 AL 中的字符。
  - done: 打印回车、换行, 恢复寄存器, ret返回
- **end\_itr**:
  - 发送EOI到0x20
  - 恢复寄存器的值
  - **iret**返回
- 恢复寄存器现场

实验修改的代码: start1.s

- 设置中断处理、定时器
- 实现了**TIMER\_ISR**函数、print\_message函数, 用于在一定时间间隔打印字符串

```

TIMER_ISR:
    pusha

    incw cnt    # 计时器计数+1

    cmpw $50, cnt    # 1000ms / 20ms = 50
    jne .end_itr

    movw $0, cnt
    call print_message    # 调用打印函数

.end_itr:
    movb $0x20, %al    # 发送EOI (End of Interrupt)
    out %al, $0x20

    popa
    iret

print_message:
    pusha

    movb $0x0E, %ah    # BIOS teletype function
    mov $message, %si

.print_loop:
    lodsb
    testb %al, %al
    jz .done
  
```

```

    int $0x10 # BIOS 视频中断
    jmp .print_loop
.done: # 打印回车、换行
    movb $0x0D, %al
    int $0x10
    movb $0x0A, %al
    int $0x10

    popa
    ret

```

## lab1.2

实验修改的代码：start2.s

**cli**关中断。

设置CR0的PE位(第0位)为1，表示进入保护模式：

```

movl %cr0, %eax
orl $0x1, %eax
movl %eax, %cr0

```

由于保护模式下，无法像lab1.1一样通过BIOS显示，故通过将参数压入栈，并调用app.s中的**displayStr**函数，该函数通过将字符串写入VGA显存来实现显示字符串。VGA 显示内存的起始地址是 0xb8000。

```

    pushl $13 # pushing the size to print into stack
    pushl $message # pushing the address of message into stack
    calll displayStr # calling the display function

```

设置gdt，第一个表项为空，随后是代码段、数据段、图形段描述符。根据段描述符的组成成分，以及不同类型段描述符的不同属性、基地址填写。

例如，图形段描述符：

```

    # graphics segment entry (0x18)
    .word 0xFFFF
    .word 0x8000
    .byte 0x0B
    .byte 0x92 # P=1, DPL=0, S=1, Type=0010 (RW)

```

```
.byte 0xCF # G=1, D=1, Limit=1111
.byte 0
```

## lab1.3

实验修改的代码：start3.s boot.c

**cli**关闭中断。将CR0的PE位设置为1，填写段描述符，与lab1.2相同。

通过跳转至boot.c中的bootMain函数，在bootMain函数中读取1号扇区的数据到内存中。然后通过**asm volatile("jmp \*%0" :: "r"(0x8C00));**跳转至该程序位置并执行，输出字符串。

```
void bootMain(void) {
    readSect((void *)0x8c00, 1);
    asm volatile("jmp *%0" :: "r"(0x8C00));
}
```

## 思考题

### 计算机的启动过程

下面阐述CPU、内存、BIOS、磁盘、主引导扇区、加载程序、操作系统的含义和他们间的关系。

BIOS: 基本输入输出系统，存储在主板的ROM（只读存储器）或Flash芯片中

MBR（主引导扇区）：位于磁盘的第一个扇区（LBA 0，512 字节），记录磁盘分区信息，决定操作系统如何访问硬盘。主要组成部分：

- Bootloader（引导代码，446 字节）：存放Bootloader的第一阶段代码，由BIOS加载执行。
- 分区表（Partition Table，64 字节）：记录磁盘的分区信息（最多4个）。
- MBR 签名（Magic Number，2 字节）：固定值0x55AA，用来验证MBR是否有效。

关系：BIOS读取mbr并执行其中的bootloader代码

- 加电自检：BIOS检查硬件是否正常

- 加载引导程序：BIOS将主引导扇区从磁盘加载到内存
- 执行引导程序：跳转到0x7c00，执行引导程序(Bootloader)代码
- 引导程序加载操作系统：将操作系统的内核从磁盘加载到内存中
- 初始化操作系统内核：操作系统初始化内存管理、进程调度等核心服务
- 用户空间进程启动

1. 镜像被QEMU加载到内存中，QEMU是如何运行的？你是如何理解bootloader的运行

的？

qemu通过BIOS读取.img文件，将mbr加载至0x7c00处，并跳转至此，将控制权交给mbr中的bootloader程序。由于实验中程序的入口被设定为start，故程序运行时，从start开始。

bootloader由start和boot文件组成，位于mbr中，由qemu的BIOS读取并加载到0x7c00处。它主要负责的功能有：初始化寄存器、栈指针、启动A20总线、加载GDTR、切换至保护模式等。它也可以读取额外扇区(利用boot里的函数)并加载到内存。

2. 初始化栈顶指针。为什么要设置SP=0x7d00？可以是别的吗？SP应该如何设置？有没有可能SP=0x7d00会出现错误？必须大于0x7c00吗？为什么？

SP指向栈顶位置。在makefile中，`x86_64-elf-ld -m elf_i386 -e start -Ttext 0x7c00 start_1.o boot.o -o bootloader.elf`命令会指定程序的起始地址为0x7c00，该位置是引导程序加载时会被放置的内存地址，通常是BIOS启动加载程序的起始位置。

栈是向下增长的，选择0x7d00作为栈顶，可以在有限空间中分配一个大小较合适的栈空间；其次，0x7c00存储了bootloader代码，不能占用代码存储空间。设置SP时，首先应大于0x7c00，其次分配合理的栈空间大小。SP=0x7d00大多数时候不会出现错误，但如果栈一直进行push操作，可能会导致溢出到0x7C00。SP必须大于0x7C00，否则可能覆盖Bootloader代码。

3. MBR的磁盘分区方式。MBR是一种被淘汰的磁盘分区方式，介绍MBR分区的优势与劣势。MBR毕竟是上个世纪的产物，为什么那时候会提出这样的设计？现在有哪些新的分区方式和bootloader？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？

MBR分区优势：

- mbr适用于大部分早期使用BIOS的设备
- mbr存储在磁盘的固定位置，例如第一个扇区，易于修改

劣势：

- 最大只能支持2TB磁盘，容量较小，最多只能有4个主分区
- mbr存储在磁盘的固定位置，记录着硬盘本身的相关信息以及硬盘各个分区的大小及位置信息，是数据信息的重要入口。若该位置损坏，会导致数据丢失

MBR作为上世纪的设计：

- 早期计算机本身存储空间容量较小，小于mbr所能存储的2TB
- mbr为计算机提供了一个标准的引导程序存储方式，支持BIOS读取
- mbr兼容性较高

除了mbr以外，还有GPT（GUID Partition Table）、LVM（Logical Volume Manager，逻辑卷管理）等分区方式。

GPT：

- 支持128个分区和几乎无限的存储容量(8Zib)
- GPT分区表有备份存储
- 通过CRC32校验防止分区表损坏
- 由UEFI(统一可扩展固件接口)引入

LVM：

- 支持动态扩展、缩小分区
- 支持快照，实现备份
- 存储性能较高

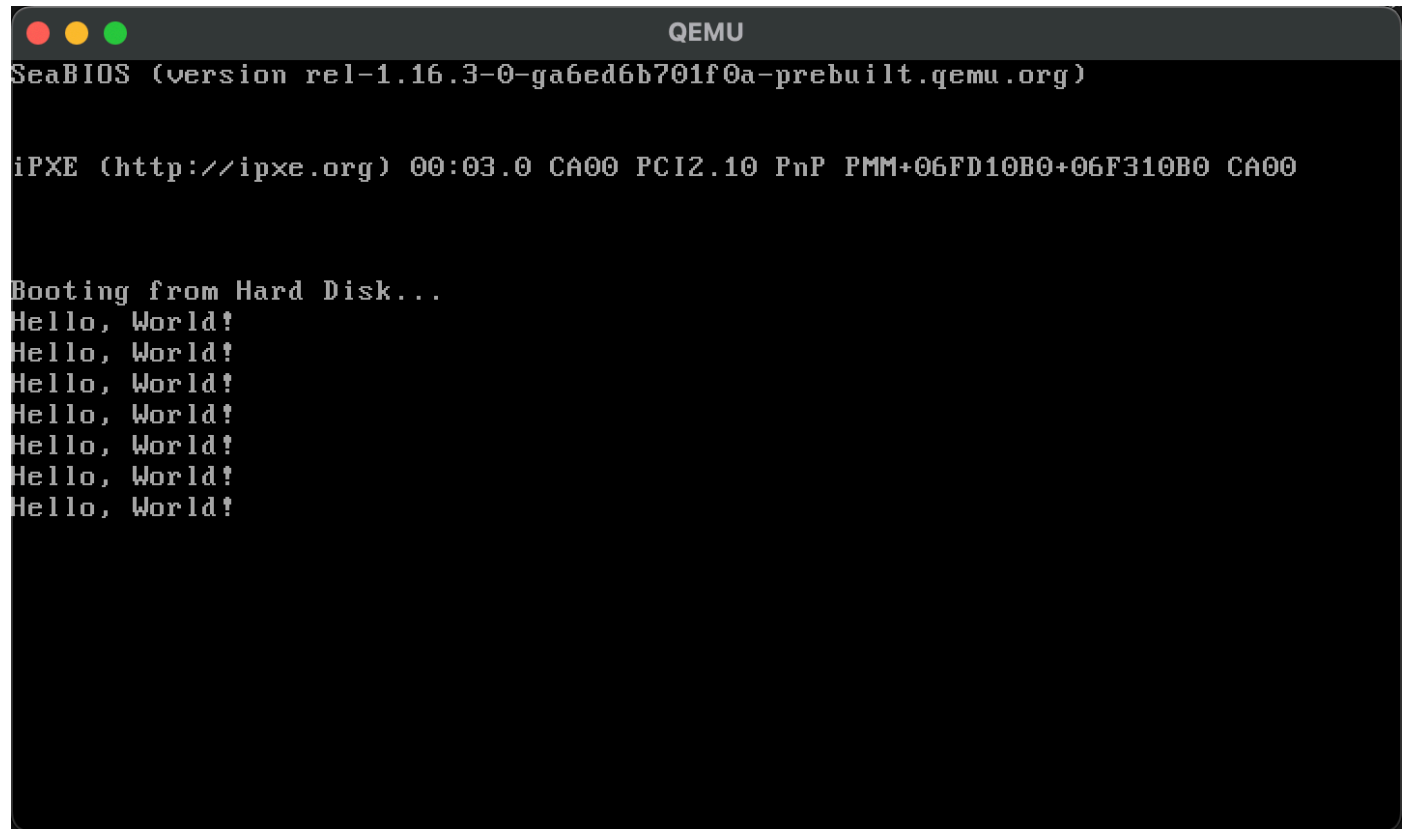
现代bootloader：适用于UEFI设备，例如Windows Boot Manager、GRUB2、rEFInd。

实验中的引导流程较为简单，代码量较小，可以直接在MBR内放Bootloader的代码。现代操作系统的bootloader代码容量较大，需要放在额外的磁盘空间中，由mbr来加载它。

## 实验结果图

---

lab1.1



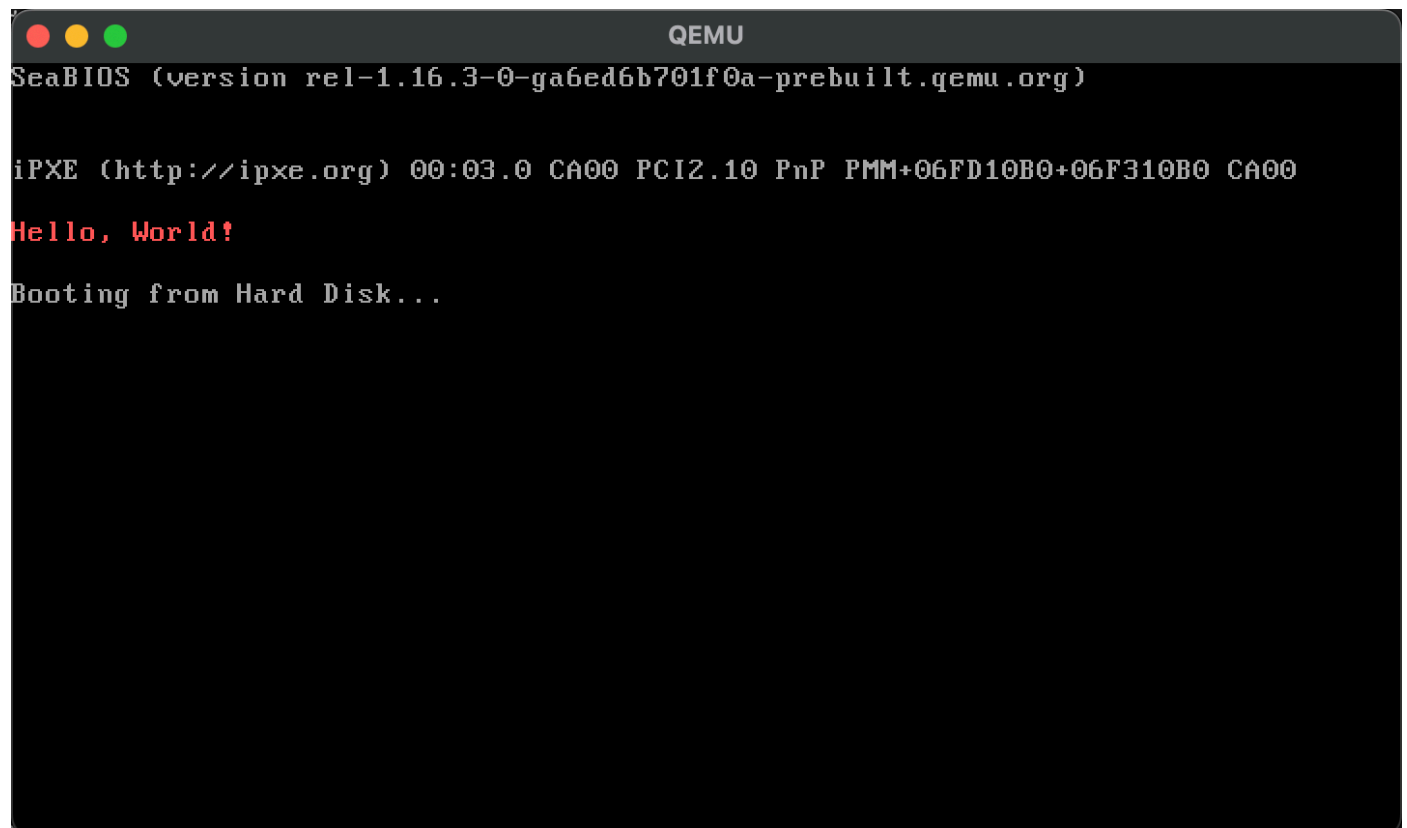
A screenshot of a QEMU terminal window. The title bar is dark gray with three colored window control buttons (red, yellow, green) on the left and the text "QEMU" in the center. The terminal background is black with white text. The text displayed is as follows:

```
SeaBIOS (version rel-1.16.3-0-ga6ed6b701f0a-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FD10B0+06F310B0 CA00

Booting from Hard Disk...
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

lab1.2



A screenshot of a QEMU terminal window, similar to the one above. The title bar is dark gray with three colored window control buttons (red, yellow, green) on the left and the text "QEMU" in the center. The terminal background is black with white text. The text displayed is as follows:

```
SeaBIOS (version rel-1.16.3-0-ga6ed6b701f0a-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FD10B0+06F310B0 CA00

Hello, World!

Booting from Hard Disk...
```

