实验报告 1

221275207　喻思文

221275027@smail.nju.edu.cn

一、实验进度

任务 1、2、3 均完成。具体截图在接下来的各个任务中展示。

二、第一小节相关名词含义和关系

• CPU：计算机的核心部件，负责执行指令和处理数据，通过读取内存中的指令来控制计算机的操作。

• 内存：计算机中用于临时存储数据和指令的硬件。分为 RAM（随机存取存储器）和 ROM（只读存储器）。

• BIOS（基本输入输出系统）：存储在主板上的 ROM 中的固件程序，负责初始化硬件并引导计算机启动过程。

• 磁盘：用于长期存储数据的设备

• 主引导扇区（Master Boot Record, MBR）：磁盘上的第一个扇区（512KB），包含引导代码和分区表信息。引导代码用于启动操作系统，分区表用于记录磁盘的分区情况。

• 加载程序（Bootloader）：用于加载操作系统的程序，通常存储在主引导扇区或引导分区中。

• 操作系统：管理计算机硬件和软件资源的系统软件，为用户提供一个方便的计算机使用环境。

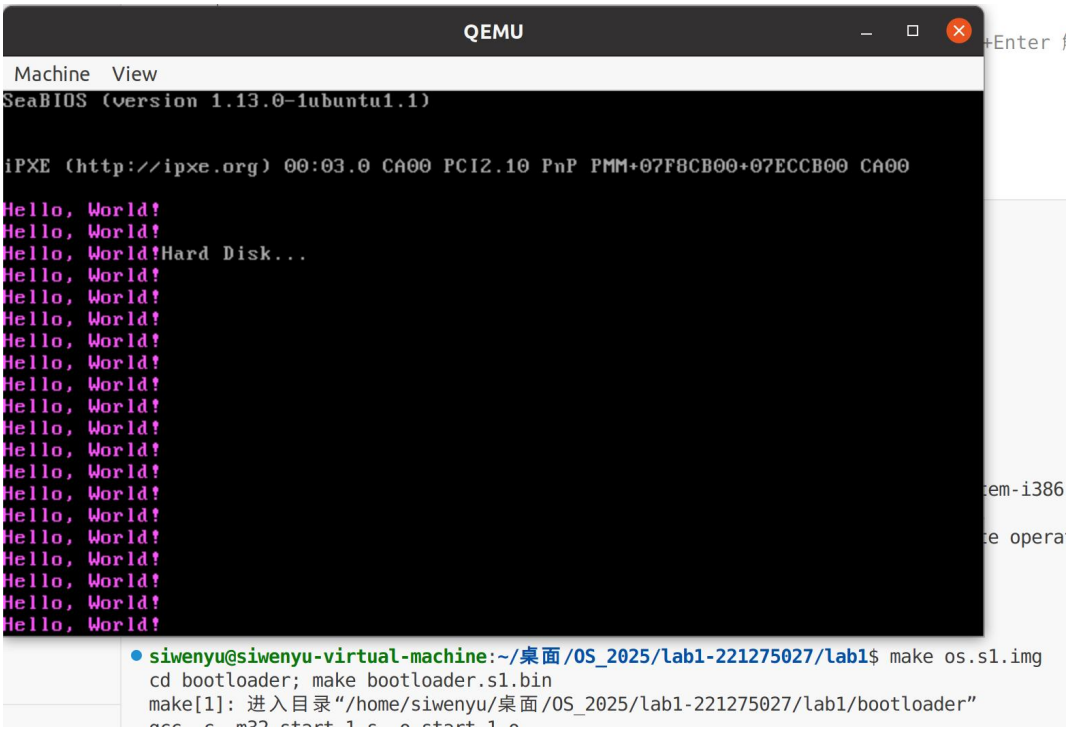
关系阐述

CPU 从内存中读取并执行指令，内存为 CPU 提供运行时所需的程序和数据。BIOS 存储在内存的 ROM 区域，CPU 加电后从内存中特定地址开始执行 BIOS 代码。BIOS 从磁盘的主引导扇区读取启动代码到内存，并跳转执行。MBR 是磁盘上的一个特殊区域，存储着启动计算机所需的引导代码。Bootloader 存储在 MBR 中，负责将操作系统的内核加载到内存。Bootloader 完成操作系统内核的加载后，操作系统开始运行，接管计算机的控制权。

三、任务 1

（一）运行截图

任务 1 运行截图如下，设定 20ms 一次中断，实验中观察输出一秒钟一行。



（二）功能实现

根据 bootstrap/start_1.s 框架给出的环境和段寄存器，初始化显存段寄存器和时钟中断向量。将显存段地址 0xb800（文本模式显存起始地址）加载到 GS 寄存器，以便后续操作显存。设置时钟中断向量 0x70 的偏移地址为 clock_handle，段地址为 CS，这样当发生时钟中断时，CPU 会跳转到 clock_handle 函数执行。

接着，配置 8253/4 定时器。计数器为二进制计数，初始值为 23863。8253/4 定时器的时钟频率为 1193180Hz，计数值为 1193180/50 = 23863.6，取整后为 23863，这样定时器每隔 50Hz（即每 20ms）产生一次中断。sti 启用中断，允许 CPU 响应中断。

```
    # 初始化显存段寄存器GS为0xB800
    movw $0xB800, %ax
    movw %ax, %gs

    # 设置时钟中断向量0x70 (段地址:偏移地址)
    movw $clock_handle, 0x70    # 偏移地址
    movw %cs, 0x72              # 段地址=CS

    # 配置8253定时器通道0，模式3，50Hz (20ms中断一次)
    movb $0x36, %al             # 控制字: 00110110b
    outb %al, $0x43             # 写入控制寄存器
    movw $23863, %ax            # 计数值=1193180/50 -1
    outb %al, $0x40             # 低字节
    movb %ah, %al
    outb %al, $0x40             # 高字节

    sti                         # 启用中断

loop:
    jmp loop                    # 等待中断
```

因为打印一行行的文字需要不断换行，对 app/app.s 中输出部分进行一些修改。displayStr 函数用于在屏幕上显示字符串。通过 GS 段寄存器访问显存，将字符和属性（修改成 0x0d 颜色）写入显存。根据当前行号计算显存偏移地址，逐个字符写入显存，直到字符串结束。每次显示完一行后，行号加 1，下次显示到下一行。

接下来时钟中断处理（clock_handle）是核心部分。保存所有通用寄存器的值和数据段寄存器。设置 DS 为 CS，确保在中断处理程序中能正确访问数据。然后向 8259 中断控制器发送 EOI（End Of Interrupt）信号，表示中断处理完成。更新中断计数器 count，每发生一次中断，计数器加 1。

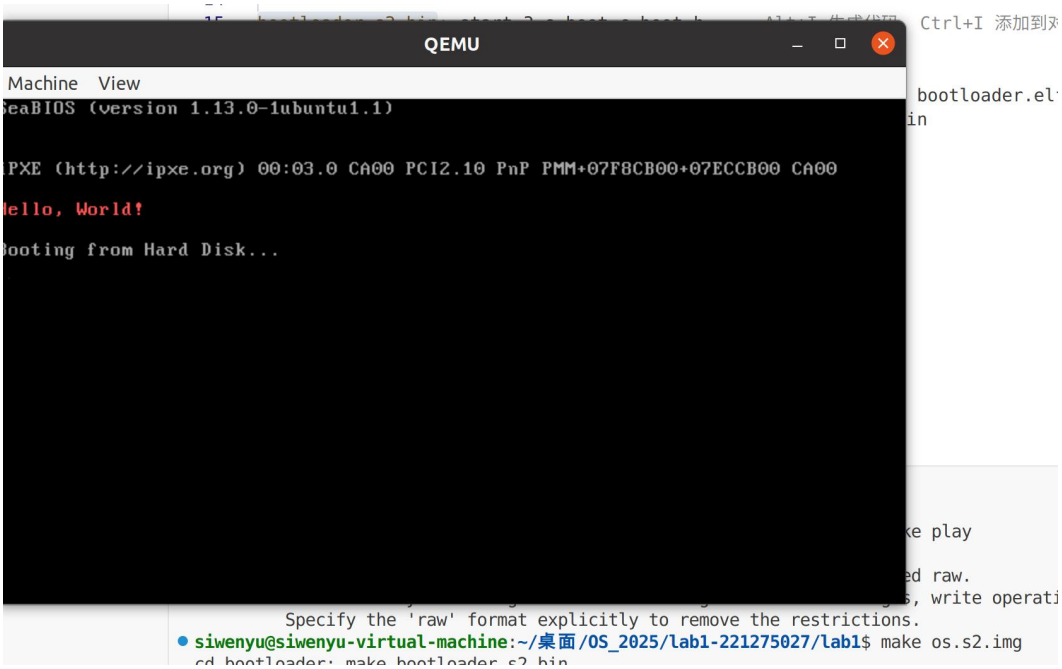
当计数器达到 50 时（即 1 秒），重置计数器，并调用 displayStr 函数显示"Hello, World!"。最后恢复寄存器和数据段寄存器，通过 iret 返回从中断前的状态继续执行。

```
# 时钟中断处理程序
clock_handle:
    pusha                     # 保存寄存器
    push %ds

    movw %cs, %ax
    movw %ax, %ds             # 设置DS=CS

    # 发送EOI到8259 PIC
    movb $0x20, %al
    outb %al, $0x20

    # 更新中断计数器
    incw (count)
    cmpw $50, (count)         # 检查是否达到50次（1秒）
    jb .done

    # 打印"Hello, World!"
    movw $0, (count)          # 重置计数器
    pushw $13                 # 字符串长度（2字节压栈）
    pushw $message            # 字符串地址（2字节压栈）
    call displayStr           # 正确调用函数
    add $4, %sp               # 清理栈（2个pushw共4字节）

.done:
    pop %ds
    popa
    iret                      # 中断返回
```

四、任务 2

（一）运行截图

任务 2 运行截图如下：



（二）功能实现

增加的代码位置参考 todo 要求。关闭中断使用 cli（Clear Interrupt）实现。通过读取端口 0x92 的值并设置第 2 位为 1，启用 A20 总线。A20 总线用于允许访问内存的高 1MB 区域，因为这是进入保护模式的必要条件。读取 CR0 寄存器，设置其第 0 位（PE 位）为 1，表示进入保护模式，然后将修改后的值写回 CR0。输出 hello world 部分可以完全参考 app.s

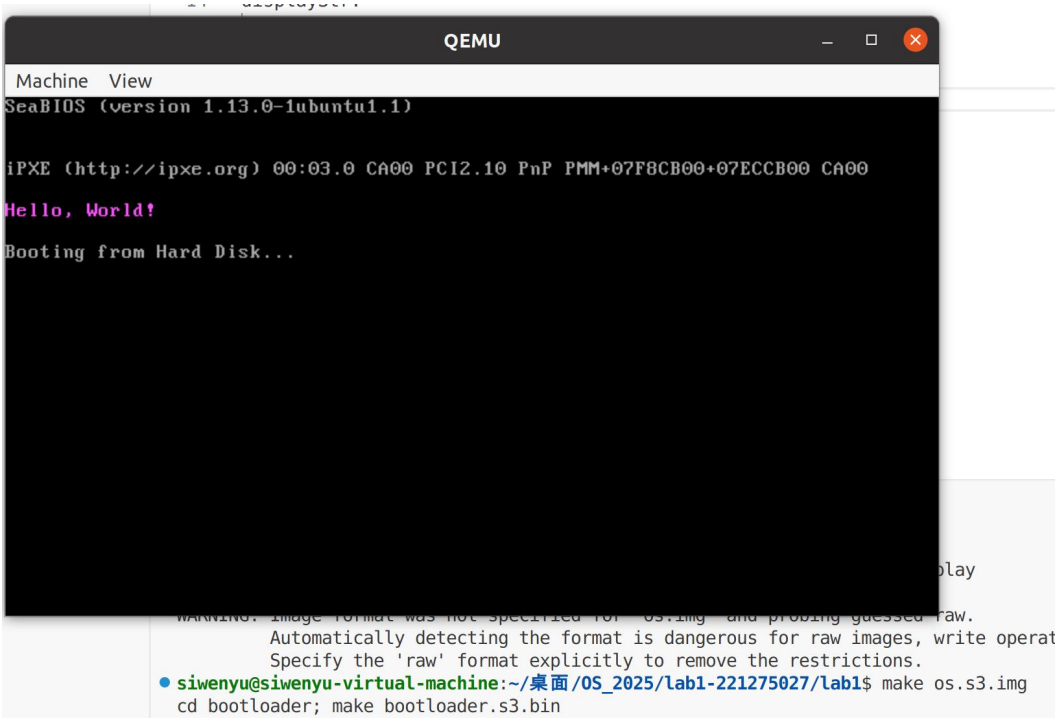
GDT 表项确定：代码段和数据段：基地址设为 0，界限设为 0xFFFF，结合 4KB 粒度标志位，实际覆盖整个 4GB 内存空间。图形数据段：基地址设为 0x8000，界限设为 0xFFFF，用于访问显存。其他各个部分根据讲义中的段描述符和代码中的位置提示可以推算出。

比如 data segment entry，整个段描述符[31:0]为 0x0000 ffff，23-16 位为 0000 0000，type 为 0010（正好对应 code 的 1010），limit[19:16]为 1111，base[31:24]为 00000000，按照注释中计算得出。

五、任务 3

（一）运行截图

任务 3 运行截图如下：



其中，为了区分任务 2 和 3 的内存来源，我将 app/app.s 中字体颜色（位于 displayStr 函数的 movb $0x0c, %ah 一行）改为 movb $0x0d, %ah，因此不是黑底红字。

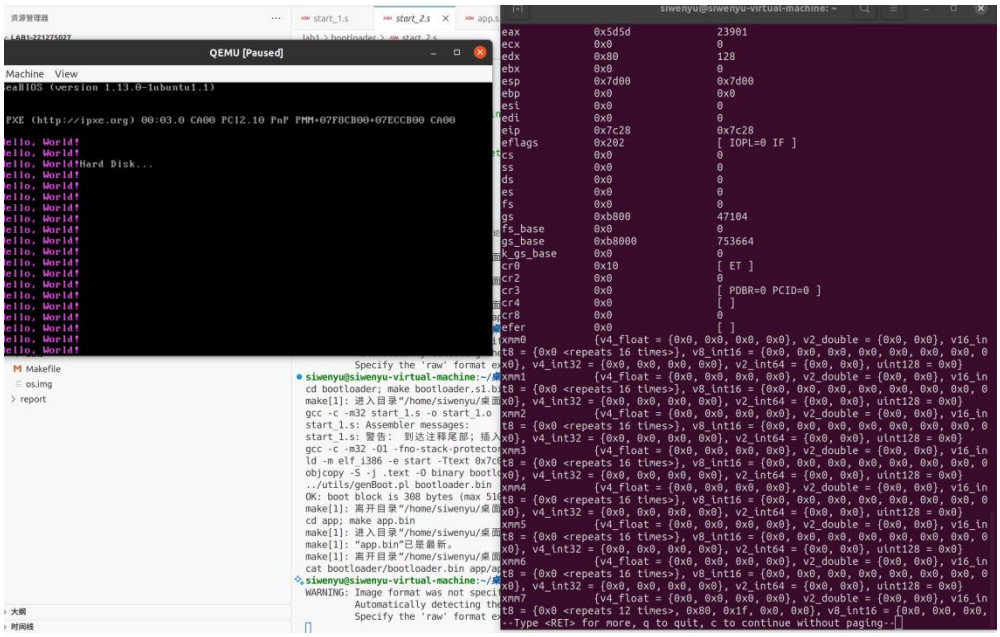
（二）功能实现

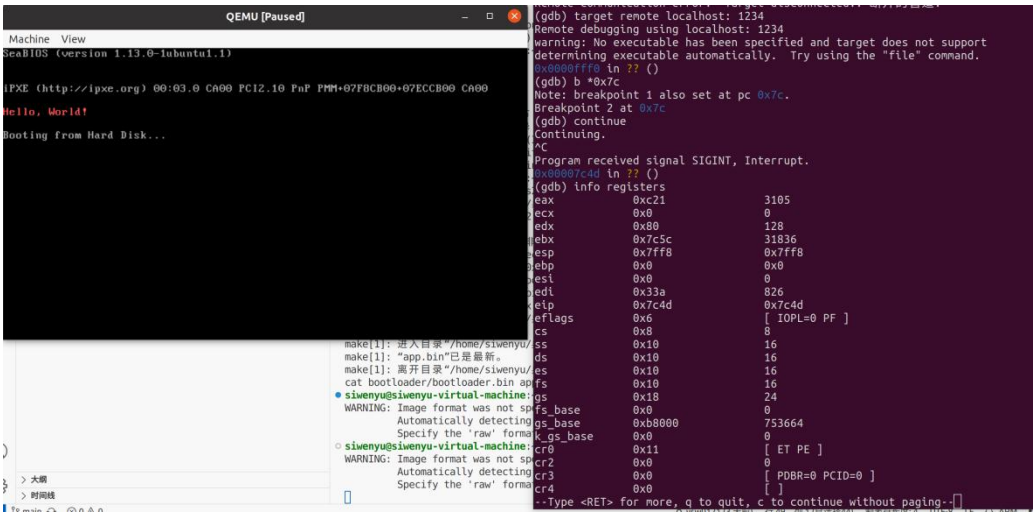
start 部分基本与任务 2 一致，而 start32 部分需要 jump to bootMain，加载磁盘 1 号扇区，也就是 app 文件夹内容，不需要像 task2 另行写输出。对 bootloader/boot.c 中的 bootMain 函数，参考 app/Makefile 中 "ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf" 可得，我们要一个指向函数的指针，设置 entry 指向内存地址 0x8c00，用于加载第二阶段的引导程序。调用 readSect 函数，将磁盘的第一个扇区（offset = 1）读取到 0x8c00 开始的内存位置。最后调用 entry 指针所指向的函数，执行从磁盘读取到内存的代码。

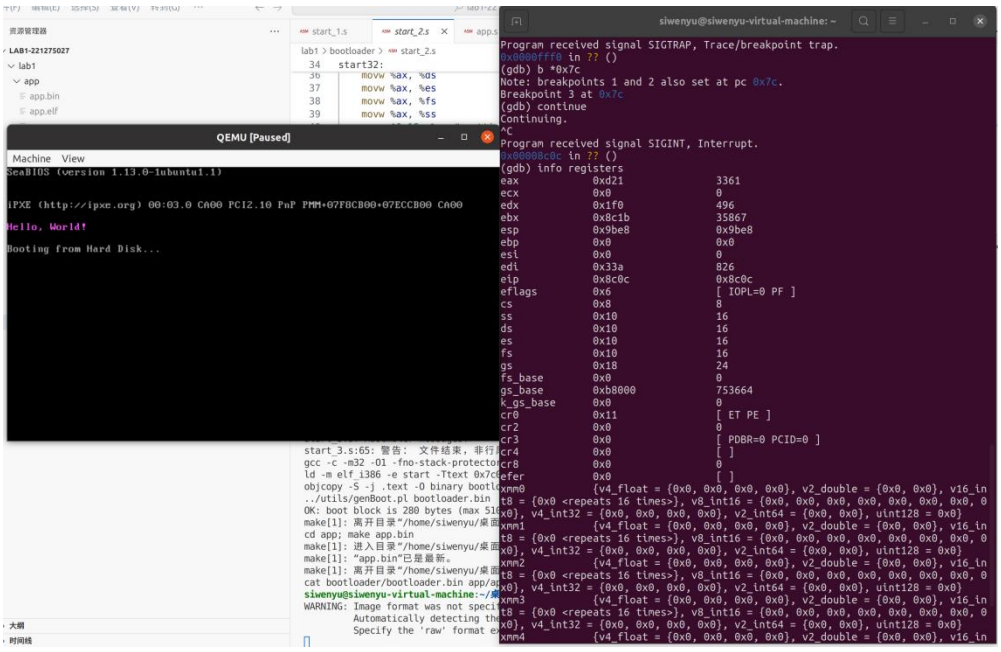
六、思考题

（一）gdb 调试

任务 1、2、3 的 gdb 调试（依照 introduction 过程）依次如下：


QEMU [Paused]
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00
Hello, World!
Booting from Hard Disk...
siwenyu@siwenyu-virtual-machine: ~
eax 0x5d5d 23901
ecx 0x0 0
edx 0x80 128
ebx 0x0 0
esp 0x7d00 0x7d00
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x7c28 0x7c28
eflags 0x202 [IOPL=0 IF]
cs 0x0 0
ss 0x0 0
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0xb800 47104
fs_base 0x0 0
gs_base 0xb8000 753664
k_gs_base 0x0 0
cr0 0x10 [ET]
cr2 0x0 0
cr3 0x0 [PDBR=0 PID=0]
cr4 0x0 [ ]
cr8 0x0 0
efer 0x0 [ ]
--Type <RET> for more, q to quit, c to continue without paging--


QEMU [Paused]
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00
Hello, World!
Booting from Hard Disk...
(gdb) target remote localhost: 1234
Remote debugging using localhost: 1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000fff0 in ?? ()
(gdb) b *0x7c
Note: breakpoint 1 also set at pc 0x7c.
Breakpoint 2 at 0x7c
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x00007c4d in ?? ()
(gdb) info registers
eax 0xc21 3105
ecx 0x0 0
edx 0x80 128
ebx 0x7c5c 31836
esp 0x7ff8 0x7ff8
ebp 0x0 0x0
esi 0x0 0
edi 0x33a 826
eip 0x7c4d 0x7c4d
eflags 0x6 [IOPL=0 PF]
cs 0x8 8
ss 0x10 16
ds 0x10 16
es 0x10 16
fs 0x10 16
gs 0x18 24
fs_base 0x0 0
gs_base 0xb8000 753664
k_gs_base 0x0 0
cr0 0x11 [ET PE]
cr2 0x0 0
cr3 0x0 [PDBR=0 PID=0]
cr4 0x0 [ ]
--Type <RET> for more, q to quit, c to continue without paging--


QEMU [Paused]
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00
Hello, World!
Booting from Hard Disk...
siwenyu@siwenyu-virtual-machine: ~
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000fff0 in ?? ()
(gdb) b *0x7c
Note: breakpoints 1 and 2 also set at pc 0x7c.
Breakpoint 3 at 0x7c
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x00008c0c in ?? ()
(gdb) info registers
eax 0xd21 3361
ecx 0x0 0
edx 0x1f0 496
ebx 0x8c1b 35867
esp 0x9be8 0x9be8
ebp 0x0 0x0
esi 0x0 0
edi 0x33a 826
eip 0x8c0c 0x8c0c
eflags 0x6 [IOPL=0 PF]
cs 0x8 8
ss 0x10 16
ds 0x10 16
es 0x10 16
fs 0x10 16
gs 0x18 24
fs_base 0x0 0
gs_base 0xb8000 753664
k_gs_base 0x0 0
cr0 0x11 [ET PE]
cr2 0x0 0
cr3 0x0 [PDBR=0 PID=0]
cr4 0x0 [ ]
cr8 0x0 0
efer 0x0 [ ]

均选择的 0x7c 断点，没发现什么大的问题。

• QEMU 的运行机制

1. 加载镜像文件：将指定的镜像文件加载到虚拟内存中。

2. 模拟硬件环境：QEMU 模拟计算机的硬件环境，包括 CPU、内存、硬盘、显卡等。相当于创建一个虚拟的硬件平台，使得操作系统和应用程序能够在其中运行。

3. 启动引导过程：镜像文件中读取引导扇区（通常是前 512KB），并将控制权交给引导程序。引导程序负责初始化硬件、加载操作系统内核等。

4. 执行操作系统内核：引导程序加载操作系统内核到内存中，并跳转到内核入口点开始执行。操作系统内核会继续初始化系统，最终进入用户界面。

• bootloader 运行理解

当计算机启动时，BIOS 会从硬盘的 MBR 读取 bootloader 到内存中。然后 bootloader 会初始化硬件设备。接着，bootloader 从硬盘或其他存储设备中**读取操作系统内核到内存**中。最后 bootloader 将控制权交给操作系统内核的入口点，**内核开始执行初始化过程。**完成之后系统进入操作系统的核心部分，最终进入用户界面。

（二）栈指针初始化

• 必须大于 0x7c00 的原因

为了避免覆盖引导扇区。引导扇区通常被加载到 0x7c00 处，如果栈顶指针设置为小于或等于 0x7c00，栈的增长可能会覆盖引导扇区的代码和数据，导致程序崩溃或行为异常。

• 本实验设置成 0x7d00 原因

1. 内存布局约定

在实模式下，x86 架构的计算机通常将中断向量表放置在内存的前 1024 字节（0x0000-0x03ff），BIOS 数据区通常位于 0x0000-0x07ff，可用内存通常从

0x0800 开始，引导扇区通常被加载到 0x7c00 处，因此，将栈顶指针设置为 0x7d00 是为了在引导扇区之后留出一定的空间，避免覆盖引导代码和数据。

2. 在 x86 架构中，栈是向下增长的，即 SP 在压栈时会减小。将 SP 设置为 0x7d00 意味着栈将从 0x7d00 开始向下增长，留出从 0x7d00 到 0x7c00 的空间给引导程序使用。

• 是否可以设置其他值

应该可以，但是要确保所选的栈顶位置不会覆盖到已使用的内存区域，如引导扇区（0x7c00-0x7dff）或其他关键数据区域。

而且栈的大小需要足够大以满足引导程序的需求。如果引导程序较为复杂，可能需要更大的栈空间。并且合乎一些系统的约定。

• 0x7d00 是否可能出错

可能出错，比如因为栈空间不足，或者如果其他数据或代码区域被放置在 0x7d00 附近，可能会与栈发生冲突，导致数据被覆盖或程序崩溃。或者设置的 SP 值不符合系统约定，可能导致兼容问题。

• SP 应该如何设置

根据上面思考，主要考虑内存布局/扇区设置、内存可用、栈大小需求和系统约定等因素。

（三）MBR 磁盘分区

• 实验中仍使用的原因

MBR 是一种被广泛支持的分区方式，几乎所有的操作系统和硬件设备都支持 MBR 分区。这使得在实验环境中使用 MBR 可以确保兼容性，避免因分区方式不兼容导致的问题。

而且 MBR 分区方式相对简单，易于理解和操作，可以让我们更直观地理解磁盘分区和引导过程的基本原理。

• MBR 提出背景

当时的技术环境和需求与现在有很大不同。第一，当时的磁盘容量相对较小，2TB 的限制在当时是足够的。设计时无需考虑更大容量磁盘的分区需求。第二当时的操作系统和应用程序对分区的需求相对简单，4 个主分区或 3 个主分区加 1 个扩展分区的设计足以满足大多数用户的存储需求。加上当时技术有限，软硬件技术限制了分区方式的复杂度，而 MBR 的简洁高效符合当时的技术水平和需求。

• 新的分区方式和 bootloader

1. GPT（GUID 分区表）：支持更大的磁盘容量，提供了更好的数据完整性和恢复能力。

2. UEFI（统一可扩展固件接口）：UEFI 是一种替代传统 BIOS 的固件接口，支持更复杂的引导过程和更大的磁盘分区，通常与 GPT 分区方式结合使用。

3. GRUB2 等现代 bootloader 支持多种分区方式和操作系统，提供了更灵活和强大的引导功能。

七、总结

1. 深入理解操作系统启动过程：不仅了解了操作系统的启动流程，还掌握了如何通过引导程序初始化硬件和加载内核。这让我对计算机系统的启动机制有了更全面的认识。

2. 掌握中断处理与定时器操作：学会了如何设置和处理中断，特别是时钟中断，通过配置定时器芯片实现精确的时间控制。

3. 理解内存管理与程序加载：对内存模型和程序加载过程有了更深入的理解。学会了如何在内存中分配空间、加载程序以及管理内存资源。