第3讲 硬件视角的操作系统

背景回顾：操作系统有三条主线：“软件 (应用)”、“硬件 (计算机)”、“操作系统 (软件直接访问硬件带来麻烦太多而引入的中间件)”。我们已经理解了操作系统上的应用程序的本质 (状态机)。而程序最终是运行在计算机硬件上的；因此有必要理解什么是计算机硬件，以及如何为计算机硬件编程。

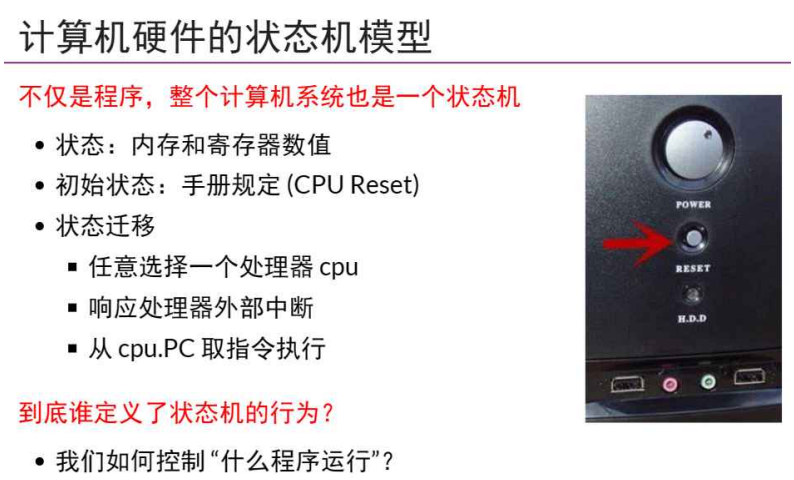
本讲内容：计算机硬件的状态机模型；回答以下问题：

- 什么是计算机硬件?
- 计算机硬件和程序员之间是如何约定的?
- 听说操作系统也是程序。那到底是鸡生蛋还是蛋生鸡?

1. CPU复位与固件

模拟器的意义

上节课实现了一个模拟数码管；其实我们可以将其看作一个七个像素的屏幕；我们写硬件模拟（比如ICS）的意义其实是告诉我们，计算机系统里面没有魔法，所有程序、硬件都是严格的数学对象，可以理解为一个函数，这个函数是可以被定义的！这个函数就是我们的模拟器。实现模拟器，就意味着完全掌握系统行为。



我们常常忽略的是状态机的初始状态！因为初始状态决定了我们有没有可能将自己的程序放到计算机系统上执行。远古时期的电脑上都有一个reset键（热启动）。冷启动就是从没有电的地方加电启动，reset就是已经有电了，然后你把状态机的状态恢复到初始状态。

计算机系统就是靠着reset的时候的状态，建立起了程序员和计算机系统之间的桥梁。

硬件和程序员的约定

CPU是由CPU的厂商生产的（废话），但是你并没有买过英特尔的电脑（甚至是主板！）

所以这里有一个很有意思的问题——生产CPU的厂商和生产主板的厂商bu'shi然后你如果打开一个惠普的电脑，开机的时候会有一个惠普的logo，所以这件事情是谁做的？

CPU规定好了reset后的值（比如PC，英特尔在生产的时候写死了，reset以后又会被强行拨回去了，然后就开始无情地执行指令了！）。

主板生产厂家只要在那个cpu-reset地址放上代码就可以了！

Bare-metal 与程序员的约定

Bare-metal 与厂商的约定

- CPU Reset 后的状态 (寄存器值)
  - 厂商自由处理这个地址上的值
  - Memory-mapped I/O

厂商为操作系统开发者提供 Firmware

- 管理硬件和系统配置
- 把存储设备上的代码加载到内存
  - 例如存储介质上的第二级 loader (加载器)
  - 或者直接加载操作系统 (嵌入式系统)

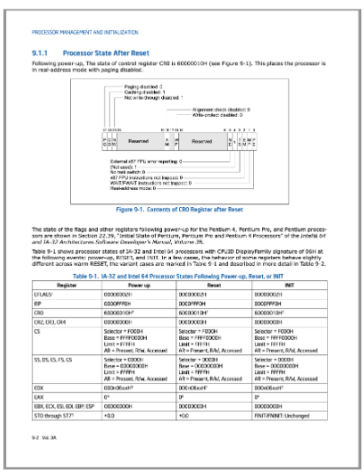
如果在做windows update的时候，你会看到一段时间叫做固件更新（firmware update），如果更新以后重启，你可能会进入一个你从来没见过的界面，同时警告你千万不要关机，它会有个读进度条，同时你什么也不能干，后面就一切正常了，你再也看不到这个界面了。实际上是，厂商会在主板上放上一个小的存储器（可读可写），然后这个存储器放了代码和数据，比如刚才重启电脑的时候看到的惠普的logo。

厂商可能还会和应用程序员或者操作系统开发者再有一层约定，比如firmware固件的代码会把存储设备代码加载到内存执行。就是实际上是有一个好几层的加载。

x86 Family: CPU Reset

CPU Reset (Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A/3B)

- 寄存器会有确定的初始状态
  - `EIP = 0x0000fff0`
  - `CR0 = 0x60000010`
    - 处理器处于 16-bit 模式
  - `EFLAGS = 0x00000002`
    - Interrupt disabled
- TFM (5,000 页+)
  - 最需要的 Volume 3A 只有 ~400 页 (我们更需要 AI)



其他平台上的 CPU Reset

Reset 后处理器都从固定地址 (Reset Vector) 启动

- MIPS: 0xbfc00000
  - Specification 规定
- ARM: 0x00000000
  - Specification 规定
  - 允许配置 Reset Vector Base Address Register
- RISC-V: Implementation defined
  - 给厂商最大程度的自由

Firmware 负责加载操作系统

- 开发板：直接把加载器写入 ROM
- QEMU：`-kernel` 可以绕过 Firmware 直接加载内核 (RTFM)

x86 CPU Reset 之后：到底执行了什么？

状态机 (初始状态) 开始执行

- 从 PC 取指令、译码、执行……
- 开始执行厂商“安排好”的 Firmware 代码
  - x86 Reset Vector 是一条向 Firmware 跳转的 jmp 指令

Firmware: BIOS vs. UEFI

- 一个小“操作系统”
  - 管理、配置硬件；加载操作系统
- Legacy BIOS (Basic I/O System)
  - IBM PC 所有设备/BIOS 中断是有 specification 的 (成就了“兼容机”)
- UEFI (Unified Extensible Firmware Interface)

如果我们想要配置一些计算机的硬件，比如我们的cpu是有很多可以配置的地方的——比如有的cpu可能有超线程，就是有一个cpu的物理核心，但是可以分裂成两个逻辑核心来使用，对于操作系统看来就是两个完全一样的核心，这个功能被英特尔设置为可以开也可以关闭。这个功能的配置不是由操作系统完成的，是操作系统运行之前的硬件配置完成的，就是所谓的bios。

- bios：配置和硬件、主板相关的信息，比如禁用、启动主板的端口，这部分代码是由厂商完成的。比如你还可以设置插上u盘之后优先从u盘启动os！你就可以从u盘搞一个linux！
- 所以firmware本质上可以理解为一个小的os，它上面也可以运行一些os，当然它上面的运行的软件是比较固定的，它的作用基本就是管理和配置硬件，加载操作系统。

很快，随着计算机的进步，小小的bios已经不能满足我们的需求了，在ibm pc之后，硬件厂商面临越来越多的问题——比如计算机上的指纹锁……指纹锁要驱动（设备都需要驱动程序），或者比如你要从u盘上启动一个os，usb蓝牙转换器链接的蓝牙键盘，如果没有键盘机器可能会拒绝启动，然后出现下面的若志信息：

> Keyboard not found, press F1 to continue.

所以有了今天的UEFI，相当于就是一个小的os，规定了驱动的格式、规范，你的应用程序要是什么样的格式、规范。

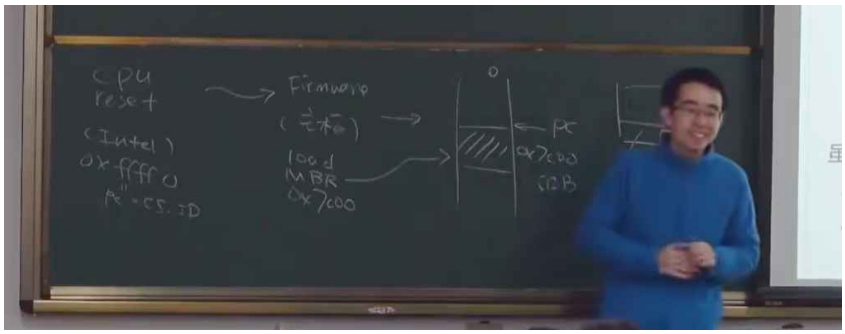
回到 Legacy BIOS: 约定

BIOS 提供机制，将程序员的代码载入内存

- Legacy BIOS 把第一个可引导设备的第一个 512 字节加载到物理内存的 `7c00` 位置
  - 此时处理器处于 16-bit 模式
  - 规定 `CS:IP = 0x7c00,(R[CS] << 4) | R[IP] == 0x7c00`
    - 可能性1：`CS = 0x07c0, IP = 0`
    - 可能性2：`CS = 0, IP = 0x7c00`
  - 其他没有任何约束

虽然最多只有 446 字节代码 (64B 分区表 + 2B 标识)

- 但控制权已经回到程序员手中了！
- 你甚至可以让 ChatGPT 给你写一个 Hello World
  - 当然，他是抄作业的 (而且是有些小问题的)



2. 调试MBR代码

示例：firmware | Makefile | mbr.S | init.gdb

```
#define SECT_SIZE  512

.code16  // 16-bit assembly

// Entry of the code
.globl _start
_start:
  lea   (msg), %si   // R[si] = &msg;

again:
  movb  (%si), %al   // R[al] = *R[si]; <--+
  incw  %si          // R[si]++;           |
  orb   %al, %al     // if (!R[al])        |
  jz    done         //   goto done; --+   |
  movb  $0x0e, %ah   // R[ah] = 0x0e;  |   |
  movb  $0x00, %bh   // R[bh] = 0x00;  |   |
  int   $0x10        // bios_call();   |   |
  jmp   again        // goto again; ---+---+
                     //                |
done:                //                |
  jmp   .            // goto done; <---+

// Data: const char msg[] = "...";
msg:
  .asciz "This is a baby step towards operating systems!\r\n"

// Magic number for bootable device
.org SECT_SIZE - 2
.byte 0x55, 0xAA
```

示例：firmware | Makefile | mbr.S | init.gdb

```
# Kill process (QEMU) on gdb exits
define hook-quit
  kill
end

# Connect to remote
target remote localhost:1234
file a.out
break *0x7c00
layout src
continue
```

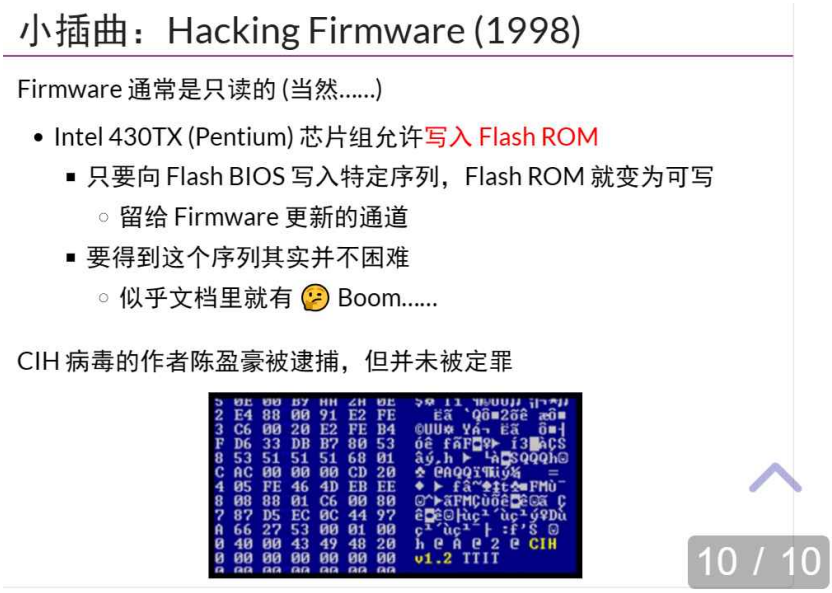
init.gdb 大幅简化了修改代码-执行调试-观测结果的流程。在这类基础设施上的投入是绝对值得的——让我们来假设一个 (固件工程师) 经常面临的场景：代码在模拟器中一切正常，但在真机 (或是无比简陋的开发板) 上却无法通过。此时，你的工作流程是：

- 修改你的代码，利用有限的机制向外输出信息
- 在模拟器中调试代码确认无误
- 部署到真实机器上运行，不断缩小问题的范围，直到定位到问题

如果没有基础设施的帮助，你的流程会被大幅拉长，直到折磨得你最终放弃。因此，《操作系统》课程中的另一个重要主题是不断提醒大家，我们应该**思考做事的方式、大胆地问出好的问题，并且小心地去求证**。

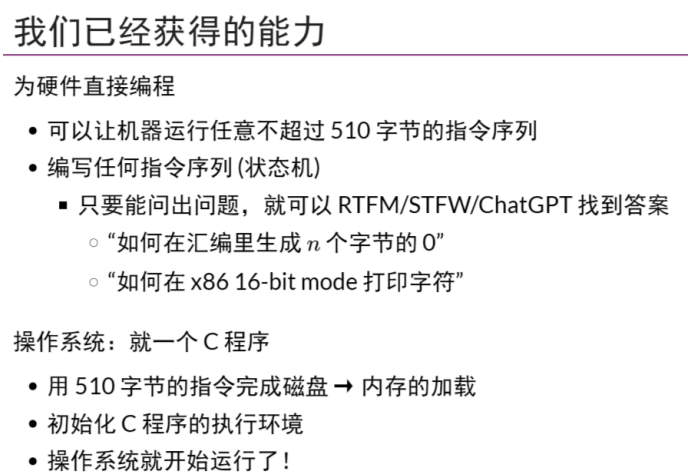
例如，在我们从 “概念上” 理解了计算机的启动过程后，就可以尝试 “代码级” 的深入理解了。通过检查 CPU Reset 时整个计算机系统的状态，我们就能发现 0x7c00 位置并没有出现 MBR 的内容——

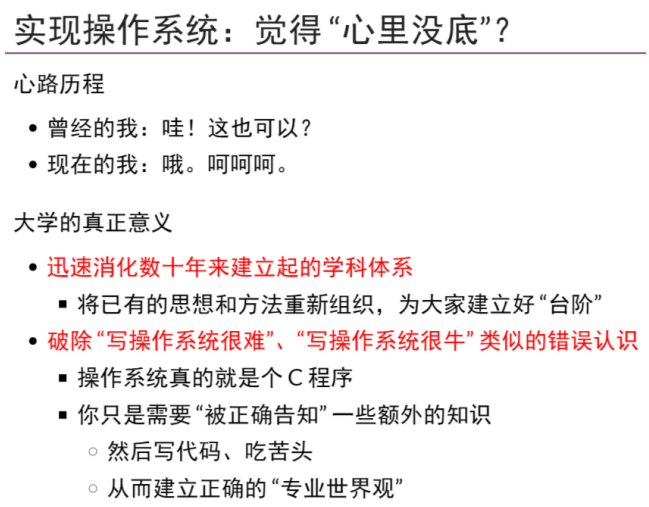
这确认了理论课部分的内容：MBR 是由 Firmware 加载的，而不是系统启动时就存在的。同时，我们也可以用 watchpoint 机制证明这一点，定位到加载 MBR 的精确汇编指令。计算机系统中没有魔法：其中绝大部分行为都是完全确定 (deterministic) 的，即便是不确定 (non-deterministic) 的部分，计算机系统依然是**严格的数学对象**，我们可以列举出不确定性的所有可能。



根据这个故事就能记得firmware是什么了，改了以后电脑就真的变成板砖了。

3. 实现os：编译运行操作系统内核





总结

本讲其实就是破除一些思想，os就是一个c程序，和平时我们写的c没有任何区别！你只需要被正确地告知某些api的行为和知识。

计算机系统是严格的数学对象：没有魔法；计算机系统的一切行为都是可观测、可理解的。

- 处理器是无情的执行指令的机器
- 厂商配置好处理器 Reset 后的行为：先运行 Firmware，再加载操作系统
- 厂商逐渐形成了达成共识的 Firmware Specification (IBM PC “兼容机”、UEFI、……)
- 操作系统真的就是个 C 程序，只是能直接访问计算机硬件