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?

回顾：计算机硬件

计算机硬件 = 数字电路

数字电路模拟器 (Logisim)

- 基本构件： wire, reg, NAND
- 每一个时钟周期
  - 先计算 wire 的值
  - 在周期结束时把值锁存至 reg

“模拟” 的意义

- 程序是 “严格的数学对象”
- 实现模拟器意味着 “完全掌握系统行为”

计算机硬件的状态机模型

不仅是程序，整个计算机系统也是一个状态机

- 状态：内存和寄存器数值
- 初始状态：手册规定 (CPU Reset)
- 状态迁移
  - 任意选择一个处理器 cpu
  - 响应处理器外部中断
  - 从 cpu.PC 取指令执行

到底谁定义了状态机的行为?

- 我们如何控制“什么程序运行”?

硬件与程序员的约定

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

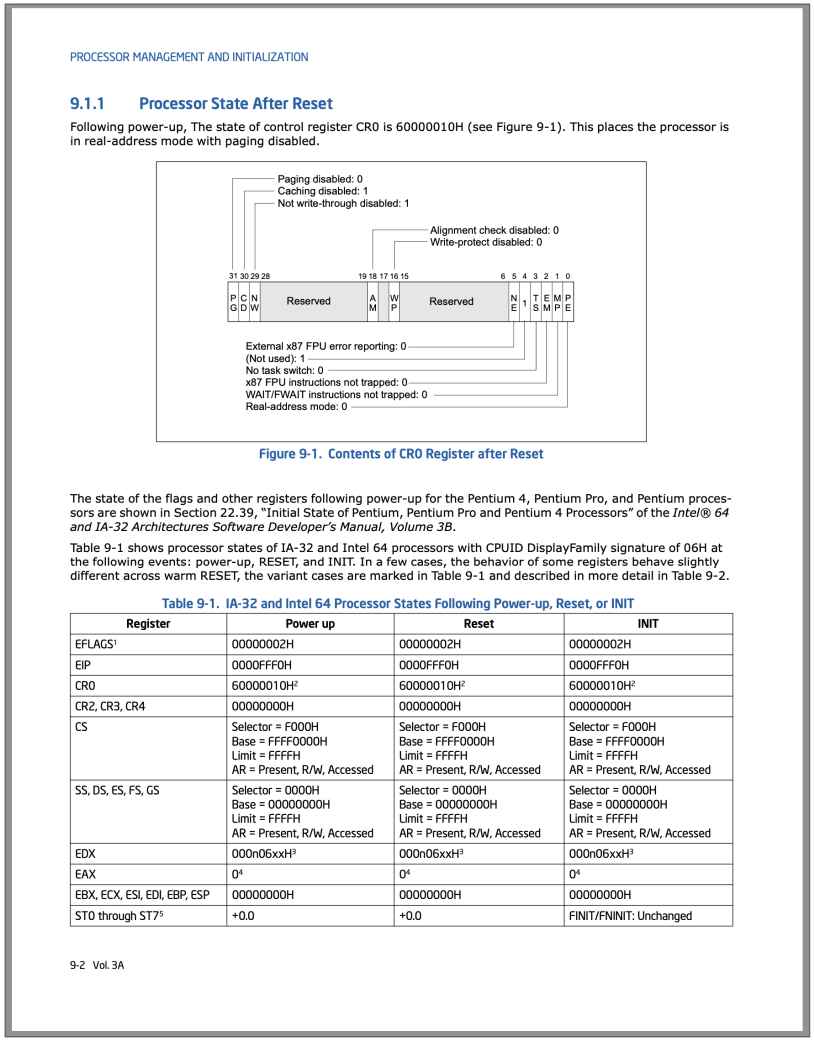
厂商为操作系统开发者提供 Firmware

- 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)

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)



PROCESSOR MANAGEMENT AND INITIALIZATION

9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor is in real-address mode with paging disabled.

Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5]	+0.0	+0.0	FINIT/FNINIT: Unchanged

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**)

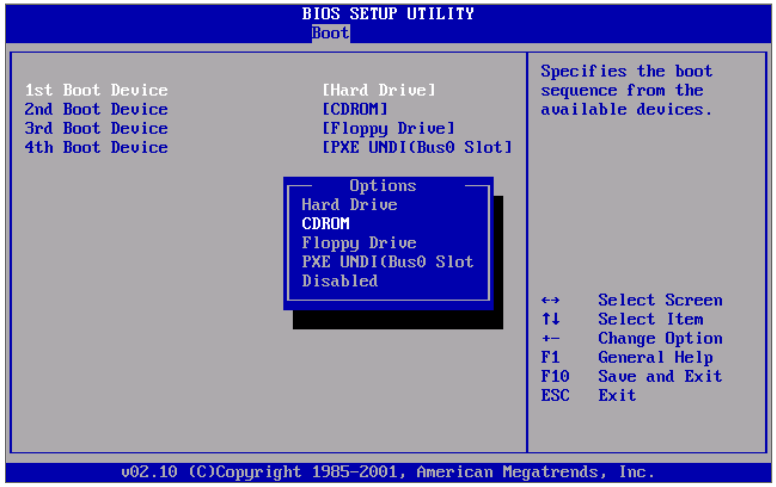
x86 CPU Reset 之后：到底执行了什么?

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**

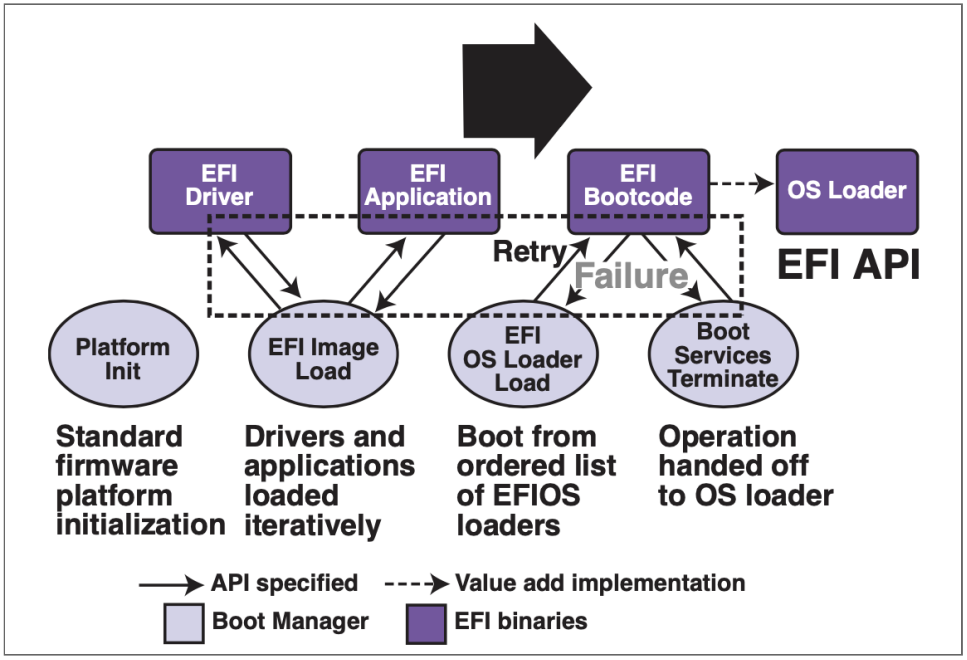
- 一个小 “操作系统”
  - 管理、配置硬件；加载操作系统
- Legacy BIOS (Basic I/O System)
  - IBM PC 所有设备/BIOS 中断是有 specification 的 (成就了 “兼容机”)
- UEFI (Unified Extensible Firmware Interface)



为什么需要 UEFI?

今天的 Firmware 面临麻烦得多的硬件：

- 指纹锁、USB 转接器上的 Linux-to-Go 优盘、山寨网卡上的 PXE 网络启动、USB 蓝牙转接器连接的蓝牙键盘、……
  - 这些设备都需要“驱动程序”才能访问
  - 解决 BIOS 逐渐碎片化的问题



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)

能不能看一下代码?

> Talk is cheap. Show me the code. ——Linus Torvalds

有没有可能我们真的去看从 CPU Reset 以后每一条指令的执行?

计算机系统公理：你想到的就一定有人做到

- 模拟方案：QEMU
  - 传奇黑客、天才程序员 **Fabrice Bellard** 的杰作
    - **QEMU, A fast and portable dynamic translator** (USENIX ATC'05)
    - Android Virtual Device, VirtualBox, ... 背后都是 QEMU
- 真机方案：JTAG (Joint Test Action Group) debugger
  - 一系列 (物理) 调试寄存器，可以实现 gdb 接口 (!!!)

🌶️UEFI 上的操作系统加载

标准化的加载流程

- 磁盘必须按 GPT (GUID Partition Table) 方式格式化
- 预留一个 FAT32 分区 (lsblk/fdisk 可以看到)
- Firmware 能够加载任意大小的 PE 可执行文件 `.efi`
  - 没有 legacy boot 512 字节限制
  - EFI 应用可以返回 firmware

更好的程序支持

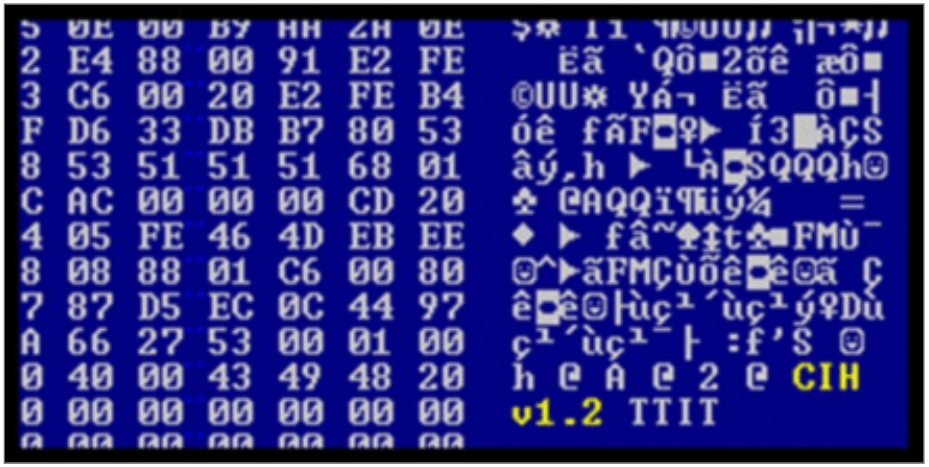
- 设备驱动框架
- 更多的功能，例如 Secure Boot，只能启动 “信任” 的操作系统

小插曲：Hacking Firmware (1998)

Firmware 通常是只读的 (当然……)

- Intel 430TX (Pentium) 芯片组允许写入 Flash ROM
  - 只要向 Flash BIOS 写入特定序列，Flash ROM 就变为可写
    - 留给 Firmware 更新的通道
  - 要得到这个序列其实并不困难
    - 似乎文档里就有 🤔 Boom……

CIH 病毒的作者陈盈豪被逮捕，但并未被定罪



实现最小“操作系统”

我们已经获得的能力

为硬件直接编程

- 可以让机器运行任意不超过 510 字节的指令序列
- 编写任何指令序列 (状态机)
  - 只要能问出问题，就可以 RTFM/STFW/ChatGPT 找到答案
    - “如何在汇编里生成 n 个字节的 0”
    - “如何在 x86 16-bit mode 打印字符”

操作系统：就一个 C 程序

- 用 510 字节的指令完成磁盘 → 内存的加载
- 初始化 C 程序的执行环境
- 操作系统就开始运行了！

实现操作系统：觉得“心里没底”？

心路历程

- 曾经的我：哇！这也可以？
- 现在的我：哦。呵呵呵。

大学的真正意义

- 迅速消化数十年来建立起的学科体系
  - 将已有的思想和方法重新组织，为大家建立好“台阶”
- 破除“写操作系统很难”、“写操作系统很牛”类似的错误认识
  - 操作系统真的就是个 C 程序
  - 你只是需要“被正确告知”一些额外的知识
    - 然后写代码、吃苦头
    - 从而建立正确的“专业世界观”

Bare-metal 上的 C 代码

为了让下列程序能够“运行起来”：

```c
int main() {
  printf("Hello, World\n");
}
```

我们需要准备什么？

- MBR 上的“启动加载器” (Boot Loader)
- 我们可以通过编译器控制 C 程序的行为
  - 静态链接/PIC (位置无关代码)
  - Freestanding (不使用任何标准库)
  - 自己手工实现库函数 (putch, printf, ...)
    - 有亿点点细节： RTFSC!

进入细节的海洋

好消息：我们提供了运行 C 程序的框架代码和库
坏消息：框架代码也太复杂了吧

- 被 ICS PA 支配的恐惧再次袭来……
  - 读懂 Makefile 需要 STFW, RTFM，大量的精力
  - 读了很久都没读到重要的地方 → 本能地放弃

花一点时间想“有更好的办法吗？”

- 花几分钟创建一个小工具：“构建理解工具”
  - UNIX Philosophy
  - 把复杂的构建过程分解成流程 + 可理解的单点
- *Get out of your comfort zone*

(0) 当然是先试试 AI 了

AI 已经正确“消化”了 Makefile 的主干 (说明我们写得“好读”)

- 但是忽略了其中的彩蛋；大多数是我们不想要的套话

This is a Makefile used to build AbstractMachine Kernels and Libraries. The Makefile includes various targets for compiling, linking, and building binaries and archives. It also includes checks for the environment, including ensuring that the environment variable `$AM_HOME` is set to an AbstractMachine repo and checking if `$ARCH` is in the list of supported architectures.

The Makefile has the following sections:

1. Basic Setup and Checks: This section sets the default goal to create a bare-metal kernel image and checks for the environment variable `$AM_HOME` and `$ARCH`.
2. General Compilation Targets: This section creates the destination directory and specifies the compilation targets.
3. General Compilation Flags: This section specifies the compiler flags.
4. Arch-Specific Configurations: This section includes arch-specific configurations.
5. Compilation Rules: This section specifies the rules for compiling the code.

The Makefile includes several rules for compiling code, including a rule for compiling a single `.c` file to a `.o` file using gcc and a rule for compiling a single `.cc` file to a `.o` file using g++. The Makefile also includes rules for creating a binary image and archive, and for cleaning up the build directory.

(1) 生成镜像和启动虚拟机

观察 AbstractMachine 程序编译过程的正确方法：

```
make -nB \
  | grep -ve '^\(\#\|echo\|mkdir\|make\)' \
  | sed "s#$AM_HOME#\$AM_HOME#g" \
  | sed "s#$PWD#.#g" \
  | vim -
```

- Command line tricks
  - `make -nB` (RTFM)
  - grep: 文本过滤，省略了一些干扰项
    - echo (提示信息), mkdir (目录建立), make (sub-goals)
  - sed: 让输出更易读
    - 将绝对路径替换成相对路径
  - vim: 更舒适的编辑/查看体验

make -nB \
| grep -ve '^\(\#\|echo\|mkdir\|make\)' \ | sed 's#$AM_HOME#\
$AM_HOME#g' \ | sed 's#$PWD#.#g' \ | vim -

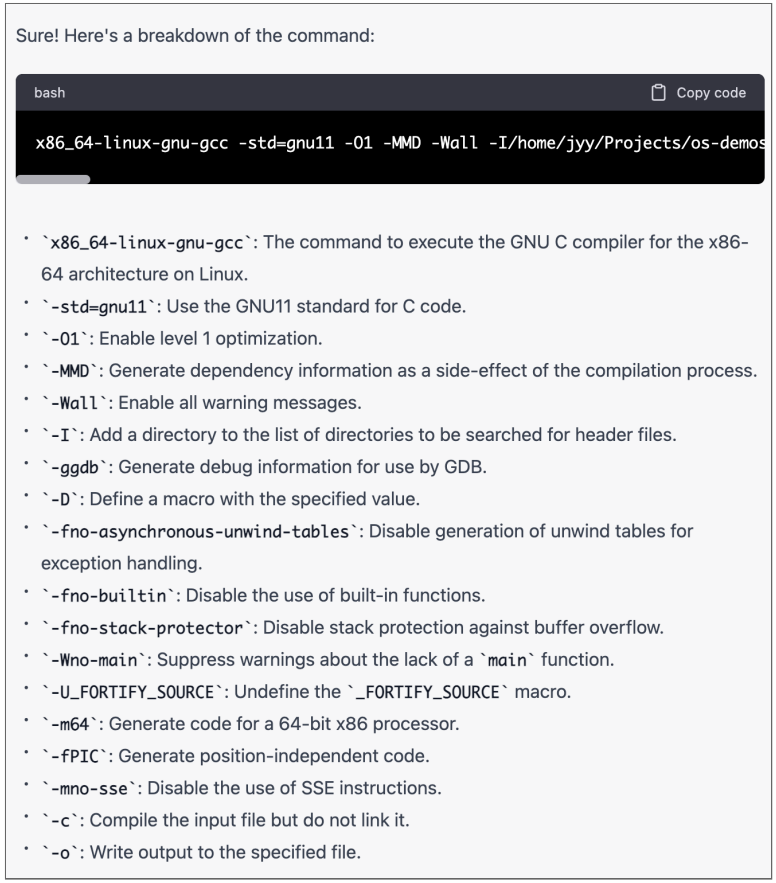
(2) 改进文本可读性

想要看得更清楚一些?

- `:%s/ /\r /g`
  - 每一个命令就像“一句话”
  - AI 落后，一代人就落后
    - 我的学习历程 (~2010)：看书、在床上刷 Wikipedia

编译/链接

- `-std=gnu11`, `-m64`, `-mno-sse`, `-I`, `-D`, ...
  - 它们是导致 vscode 里红线的原因
- `-melf_x86_64`, `-N`, `-Ttext-segment=0x00100000`
  - 链接了需要的库 (`am-x86_64-qemu.a`, `klib-x86_64-qemu.a`)



Sure! Here's a breakdown of the command:

bash Copy code

```
x86_64-linux-gnu-gcc -std=gnu11 -O1 -MMD -Wall -I/home/jyy/Projects/os-demos
```

- `x86_64-linux-gnu-gcc`: The command to execute the GNU C compiler for the x86-64 architecture on Linux.
- `-std=gnu11`: Use the GNU11 standard for C code.
- `-O1`: Enable level 1 optimization.
- `-MMD`: Generate dependency information as a side-effect of the compilation process.
- `-Wall`: Enable all warning messages.
- `-I`: Add a directory to the list of directories to be searched for header files.
- `-ggdb`: Generate debug information for use by GDB.
- `-D`: Define a macro with the specified value.
- `-fno-asynchronous-unwind-tables`: Disable generation of unwind tables for exception handling.
- `-fno-builtin`: Disable the use of built-in functions.
- `-fno-stack-protector`: Disable stack protection against buffer overflow.
- `-Wno-main`: Suppress warnings about the lack of a `main` function.
- `-U_FORTIFY_SOURCE`: Undefine the `_FORTIFY_SOURCE` macro.
- `-m64`: Generate code for a 64-bit x86 processor.
- `-fPIC`: Generate position-independent code.
- `-mno-sse`: Disable the use of SSE instructions.
- `-c`: Compile the input file but do not link it.
- `-o`: Write output to the specified file.

(3) 启动加载器 (Boot Loader)

假设 MBR 后紧跟 ELF Binary (真正的的加载器有更多 stages)

- 16-bit → 32-bit
- ELF32/64 的加载器
  - 按照约定的磁盘镜像格式加载

代码讲解:

- `am/src/x86/qemu/boot/start.S` 和 `main.c`
  - 最终完成了 C 程序的加载 (它们都可以调试)

```
if (elf32->e_machine == EM_X86_64) {
  ((void(*)())(uint32_t)elf64->e_entry)();
} else {
  ((void(*)())(uint32_t)elf32->e_entry)();
}
```

我们承诺的“操作系统”

就是一个 C 程序

- 只不过调用了更多的 API (之后解释)
  - 使用了正确的工具，就没什么困难的

支持固定的“线程”

- T_a - `while (1) printf("a");`
- T_b - `while (1) printf("b");`
  - 允许并发执行

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