AbstractMachine 选讲

蒋炎岩

南京大学　　计算机科学与技术系　　计算机软件研究所

本讲概述

困惑：为什么需要 AbstractMachine?

- 复杂系统的构造与解释
- 理解计算机系统
- 理解 AbstractMachine
- 代码导读

复杂系统的构造与解释

《计算机系统基础》到底学什么？

一句话的 take-away message: “计算机系统是一个状态机”。

更具体一点？

- 状态机的状态是什么？
  - 内存 + 寄存器 + (外部设备状态)
- 状态机的行为是什么？
  - 取指令 + 译码 + 执行
  - 输入/输出设备访问
  - 中断 + 异常控制流
  - 地址转换
- (是否感到驾驭不了？)

复杂中隐藏的秩序

理解/构造一个复杂系统 (操作系统/处理器/航母)？

- USS Midway (CV-41); 1945 年首航，“沙漠风暴” 行动后退役
  - 舰船配置；资源管理/调度；容错……

(picture: www.seaforces.org; familyvacationhub.com)

复杂中隐藏的秩序 (cont'd)

航母不是一天造成的。

(picture: history.navy.mil)

复杂中隐藏的秩序 (cont'd)

采矿船继承了航海时代的设计。

(picture: onlyinyourstate.com)

复杂中隐藏的秩序 (cont'd)

> 复杂系统的演化通常是 *evolutionary* 的而不是 *revolutionary* 的。

无法第一次就设计出“绝对完美”的复杂系统

- (因为环境一直在变)

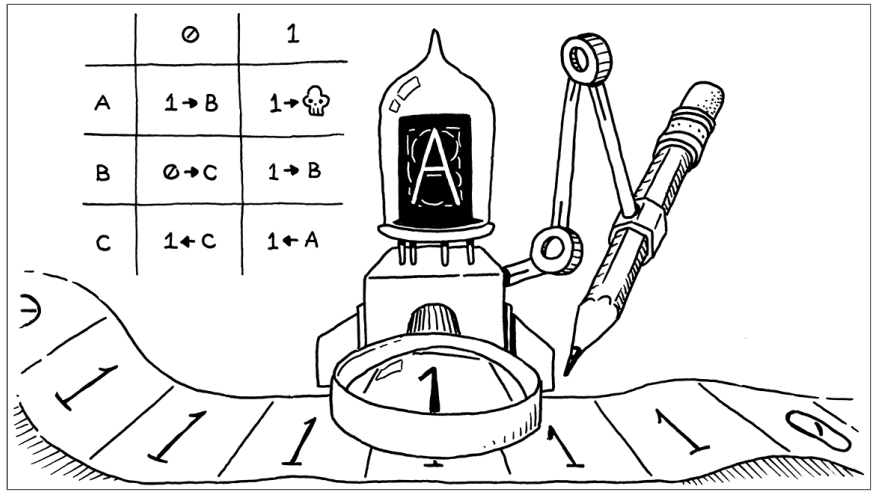
实际情况：从 minimal, simple, and usable 的系统不断经过 local modifications (trial and errors)

- 计算机硬件
- 操作系统
- 编译器/程序设计语言
  - 需求和系统设计/实现螺旋式迭代

理解计算机系统

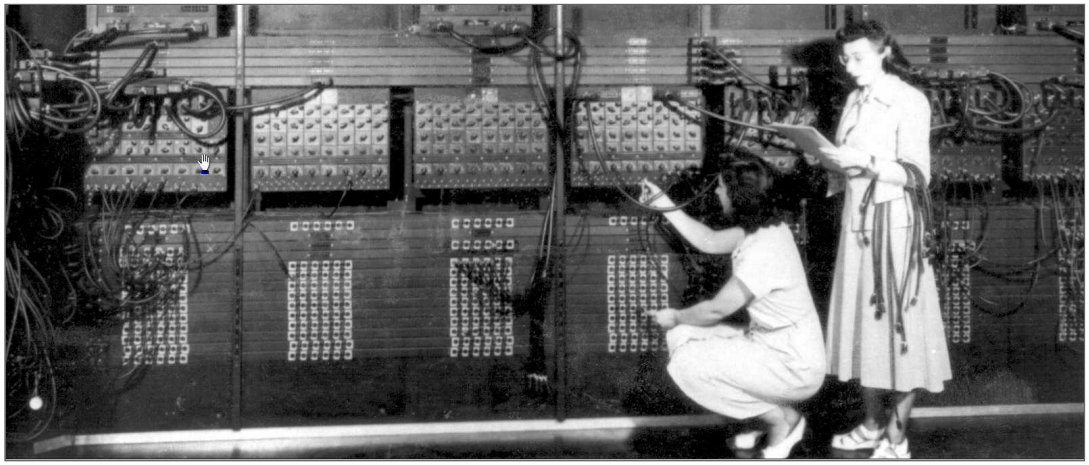
在计算机诞生之前……

Alan Turing's “machine” (1936)



- 并没有真正 “造出来”
- 纸带 + 自动机
  - 纸带 `map<int,int> mem`
  - 读写头 pos
  - 自动机 (程序)
    - (移动读写头) `pos++`, `pos--`
    - (写内存) `mem[pos] = 0`, `mem[pos] = 1`
    - (读内存) `if (mem[pos]) { } else { }`
    - (跳转) `goto label`
    - (终止) `halt()`

ENIAC: 人类可用的 Turing Machine



ENIAC Simulator by Brian L. Stuart

- 20 word (not bit) memory
- 自带 “寄存” 的状态 (寄存器)
- 支持算数运算
- 支持纸带 (串行) 输入输出

von Neumann Machine: 存储程序控制

可以把状态机的形态保存在存储器里，而不要每次重新设置。

计算机的设计受到数字电路实现的制约

- 执行一个动作 (指令) 只能访问有限数量的存储器
  - 常用的临时存储 (寄存器；包括 PC)
  - 更大的、编址的内存
  - (是不是想起了 YEMU?)

von Neumann Machine: 更多的 I/O 设备

存储程序的通用性真正掀起了计算机走向全领域的革命。

- 只要增加 in 和 out 指令，就可以和物理世界建立无限的联系
  - 持久存储 (磁带、磁盘……)
  - 读卡器、打印机……

中断：弥补 I/O 设备的速度缺陷

CPU cycles 实在太珍贵了

- 不能用来浪费在等 I/O 设备完成上
  - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了

中断 = 硬件驱动的函数调用

- 相当于在每条语句后都插入

```
if (pending_io && int_enabled) {
  interrupt_handler();
  pending_io = 0;
}
```

- (硬件上好像不太难实现)
  - 于是就有了最早的操作系统：管理 I/O 设备的库代码

中断 + 更大的内存 = 分时多线程

```
void foo() { while (1) printf("a"); }
void bar() { while (1) printf("b"); }
```

能否让 foo() 和 bar() “同时” 在处理器上执行?

- 借助每条语句后被动插入的 interrupt_handler() 调用

```
void interrupt_handler() {
  dump_regs(current->regs);
  current = (current->func == foo) ? bar : foo;
  restore_regs(current->regs);
}
```

- 操作系统背负了 “调度” 的职责

分时多线程 + 虚拟存储 = 进程

让 `foo()` 和 `bar()` 的执行互相不受影响

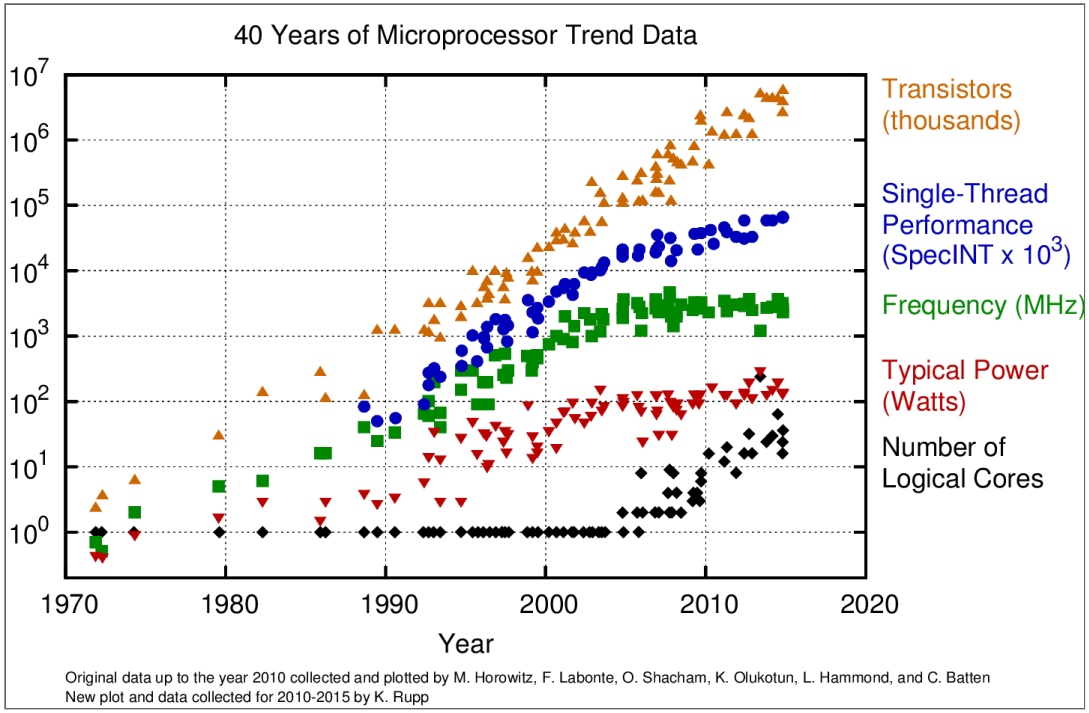
- 在计算机系统里增加映射函数 $f_{\texttt{foo}}, f_{\texttt{bar}}$
  - `foo` 访问内存地址 m 时，将被重定位到 $f_{\texttt{foo}}(m)$
  - `bar` 访问内存地址 m 时，将被重定位到 $f_{\texttt{bar}}(m)$

- `foo` 和 `bar` 本身无权管理 f
  - 操作系统需要完成进程、存储、文件的管理
  - UNIX 诞生 (就是我们今天的进程; Android app; ...)
    - `gcc a.c`
    - `readelf -a a.out` ➡ 二进制文件的全部信息

故事其实没有停止……

面对有限的功耗、难以改进的制程、无法提升的频率、应用的需求

- 多处理器、big.LITTLE、异构处理器 (GPU, NPU, ...)
- 单指令多数据 (MMX, SSE, AVX, ...), 虚拟化 (VT; EL0/1/2/3), 安全执行环境 (TrustZone; SGX), ...



演化视角的 AbstractMachine

如果我们只是为了使用计算机系统，而不关心它的实现？

Turing Machine (TRM)

> 任何编程语言编译后都表达了“内存上的计算”。

一段 C (C++ w/o RTTI, Rust, ...) 程序

- 它也是一个状态机
  - 回到了课程的 take-away message
- 从 `main` 开始
  - 一段自由内存 heap
  - 随时可以终止 `halt`
  - 随时可以输出 `putch`

I/O Extension (IOE)

> 计算机在演化的过程中多了 in 和 out 指令。

我们也配上 read 和 write 不就好了？

一个非常简化的设备模型

- 但足够支持许多 non-trivial 的软件了 (例如 LiteNES)

Context Extension (CTE)

相当于在每条指令之后都插入了异常/中断的检查

- 我们的代码会自动保存异常/中断返回的处理器状态 (context)

```
// after each instruction
if (has_exception) {
  exception_handler();
}
if (has_interrupt && int_enabled) {
  interrupt_handler();
}
```

- 此时“计算机系统”对应的状态机发生了怎样的改变？
  - **thread-os.c**

AbstractMachine 的设计取舍

提供最少机制以实现现代计算机系统软件

- TRM (程序所需最小的运行环境)
- IOE (仅提供一些系统无关的设备抽象)
- CTE (简化、统一、相对低效的中断处理)
- VME (基于页的映射，忽略硬件实现)
- MPE (假设 race-freedom、简化的系统模型)

我们的设计做了怎样的取舍？

- 得到：统一简洁的接口
  - 适合教学；跨体系结构存活 (甚至有 native)
- 失去：实际系统的特性支持
  - 不连续的内存、热插拔内存、Access/Dirty Bit……

代码导读

总结

八卦 (1): PA 的由来

yzh 觉得……好像只有 OS 不够劲啊

- 而且 OSLab 上来就把 x86 的手册丢给你好像不太好玩

那就让大家好好读读 x86 手册吧……那……

- 就做个模拟器好啦，反正就是照着手册写一遍的事
- 于是就有了 PA: 简易 x86 全系统模拟器
- 目的是让大家知道“什么是计算机”

八卦 (2): PA 的后续

觉得虽然体系里的确什么都有了，但还是不够劲啊，不如玩个大的?

自己写个 CPU

上面跑自己的 OS
上面跑自己编译器编译出来的应用程序
应用程序可以是 NEMU
NEMU 又跑自己的 OS……

好吧我承认这有点炫酷

- 但用 Verilog (Chisel) 写个 x86 的 CPU 好像夸张了点……

AbstractMachine: 演化观点的系统抽象

抽象带来的好处

- 写操作系统再也不用汇编了
  - 如果你们看过一些“自己动手写操作系统”类的书，前面讲 x86 的部分就根本不想看下去了
- 在正确的抽象层上写代码能减少细节纠结
- 软件正确性可以互相验证
  - 软件在 native 调试，调试好了再上体系结构运行

在抽象层上工作带来的额外好处

- trace
- model checking
- formal verification

End. (这是计算机系统的完整故事)

本学期 (ICS)
实现 x86/riscv/mips 上的 AbstractMachine API

下学期 (OS)
基于 AbstractMachine 实现多处理器操作系统