C 语言拾遗 (2): 编程实践

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

HW1 和 Lab1 已发布。

假设你已经熟练使用各种 C 语言机制 (~~并没有~~)

- 原则上给需求就能搞定任何代码 (~~并不是~~)

本次课程

- 怎样写代码才能从一个大型项目里存活下来？
  - 核心准则：编写可读代码
  - 两个例子

核心准则：编写可读代码

一个极端 (不可读) 的例子

IOCCC'11 best self documenting program

- 不可读 = 不可维护

```
puts(usage: calculator 11/26+222/31
  +~~~~~~~~~~~~~~~~~~~~~~~~~calculator-\
  !                           7.584,367 )
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
  ! clear ! 0 ||l   -x  l   tan  I (/) |
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
  ! 1 | 2 | 3 ||l  1/x  l   cos  I (*) |
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
  ! 4 | 5 | 6 ||l  exp  l  sqrt  I (+) |
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
  ! 7 | 8 | 9 ||l  sin  l   log  I (-) |
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~(0
);
```

一个现实中可能遇到的例子

人类不可读版 (STFW: clockwise/spiral rule)

```
void (*signal(int sig, void (*func)(int)))(int);
```

人类可读版

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int, sighandler_t);
```

编写代码的准则：降低维护成本

> Programs are meant to be read by humans and only incidentally for computers to execute. — *D. E. Knuth*
> (程序首先是拿给人读的，其次才是被机器执行。)

宏观

- 做好分解和解耦 (现实世界也是这样管理复杂系统的)
  - ~~有同学问：PA 是否允许添加额外的文件？~~

微观

- “不言自明”
  - 通过阅读代码能理解一段程序是做什么的 (specification)
- “不言自证”
  - 通过阅读代码能验证一段程序与 specification 的一致性

例子：实现数字逻辑电路模拟器

数字逻辑电路模拟器

假想的数字逻辑电路

- 若干个 1-bit 边沿触发寄存器 (`X`, `Y`, ...)
- 若干个逻辑门
  你会如何设计？
- 基本思路：状态 (存储) 模拟 + 计算模拟
  - 状态 = 变量
    - `int X = 0, Y = 0;`
  - 计算
    - `X1 = !X && Y;`
    - `Y1 = !X && !Y;`
    - `X = X1; Y = Y1;`

通用数字逻辑电路模拟器 (cont'd)

```
#define FORALL_REGS(_)  _(X) _(Y)
#define LOGIC           X1 = !X && Y; \
                        Y1 = !X && !Y;

#define DEFINE(X)       static int X, X##1;
#define UPDATE(X)       X = X##1;
#define PRINT(X)        printf(#X " = %d; ", X);

int main() {
  FORALL_REGS(DEFINE);
  while (1) { // clock
    FORALL_REGS(PRINT); putchar('\n'); sleep(1);
    LOGIC;
    FORALL_REGS(UPDATE);
  }
}
```

使用预编译：Good and Bad

Good

- 增加/删除寄存器只要改一个地方
- 阻止了一些编程错误
  - 忘记更新寄存器
  - 忘记打印寄存器
- “不言自明” 还算不错

Bad

- 可读性变差 (更不像 C 代码了)
  - “不言自证” 还缺一些
- 给 IDE 解析带来一些困难

更完整的实现：数码管显示

logisim.c 和 **display.py**

- 你也可以考虑增加诸如开关、UART 等外设
- 原理无限接近大家数字电路课玩过的 FPGA

等等……FPGA？

- 这玩意不是万能的吗？？？
- 我们能模拟它，是不是就能模拟计算机系统？
  - Yes!
  - 我们实现了一个超级超级低配版 NEMU!

例子：实现 YEMU 全系统模拟器

教科书第一章上的“计算机系统”

存储系统

```
寄存器: PC, R0 (RA), R1, R2, R3 (8-bit)
内存:   16字节 (按字节访问)
```

指令集

```
        7 6 5 4   3 2   1 0
mov    [0 0 0 0] [ rt] [ rs]
add    [0 0 0 1] [ rt] [ rs]
load   [1 1 1 0] [   addr  ]
store  [1 1 1 1] [   addr  ]
```

有“计算机系统”的感觉了？

- 它显然可以用数字逻辑电路实现
- 不过我们不需要在门层面实现它
  - 我们接下来实现一个超级低配版 NEMU……

Y-Emulator (YEMU) 设计与实现

存储模型：内存 + 寄存器 (包含 PC)

- 16 + 5 = 21 bytes = 168 bits
- 总共有 2^{168} 种不同的取值
  - 任给一个状态，我们都能计算出 PC 处的指令，从而计算出下一个状态

理论上，任何计算机系统都是这样的状态机

- (M, R) 构成了计算机系统的状态
- 32 GiB 内存有 $2^{274877906944}$ 种不同的状态……
- 每个时钟周期，取出 $M[R[PC]]$ 的指令；执行；写回
  - 受制于物理实现 (和功耗) 的限制，通常每个时钟周期只能改变少量寄存器和内存的状态
  - (量子计算机颠覆了这个模型：同一时刻可以处于多个状态)

YEMU: 模拟存储

存储是计算机能实现“计算”的重要基础

- 寄存器 (PC)、内存
- 这简单，用全局变量就好了！

```c
#include <stdint.h>
#define NREG 4
#define NMEM 16
typedef uint8_t u8; // 没用过 uint8_t?
u8 pc = 0, R[NREG], M[NMEM] = { ... };
```

- 建议 STFW (C 标准库) → bool 有没有？
- 现代计算机系统：`uint8_t == unsigned char`
  - C Tips: 使用 `unsigned int` 避免潜在的 UB
    - `-fwrapv` 可以强制有符号整数溢出为 wraparound
  - C Quiz: 把指针转换成整数，应该用什么类型？

提升代码质量

给寄存器名字?

```
#define NREG 4
u8 R[NREG], pc; // 有些指令是用寄存器名描述的
#define RA 1    // BUG: 数组下标从0开始
...
```

```
enum { RA, R1, ..., PC };
u8 R[] = {
  [RA] = 0,  // 这是什么语法? ?
  [R1] = 0,
  ...
  [PC]  = init_pc,
};

#define pc (R[PC]) // 把 PC 也作为寄存器的一部分
#define NREG (sizeof(R) / sizeof(u8))
```

从一小段代码看软件设计

软件里有很多隐藏的 dependencies (一些额外的、代码中没有体现和约束的 “规则”)

- 一处改了，另一处忘了 (例如加了一个寄存器忘记更新 NREG...)
- 减少 dependencies → 降低代码耦合程度

```
// breaks when adding a register
#define NREG 5 // 隐藏假设max{RA, RB, ... PC} == (NREG - 1)

// breaks when changing register size
#define NREG (sizeof(R) / sizeof(u8)) // 隐藏假设寄存器是8-bit

// never breaks
#define NREG (sizeof(R) / sizeof(R[0])) // 但需要R的定义

// even better (why?)
enum { RA, ... , PC, NREG }
```

PA 框架代码中的 CPU_state

```
struct CPU_state {
};

// C is not C++
// cannot declare "CPU_state state";

#define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)
#define reg_w(index) (cpu.gpr[check_reg_index(index)]._16)
#define reg_b(index) (cpu.gpr[check_reg_index(index) & 0x3]._8[inde
```

对于复杂的情况，struct/union 是更好的设计

- 担心性能 (check_reg_index)?
  - 在超强的编译器优化面前，不存在的

YEMU: 模拟指令执行

在时钟信号驱动下，根据 (M, R) 更新系统的状态
RISC 处理器 (以及实际的 CISC 处理器实现)：

- 取指令 (fetch): 读出 `M[R[PC]]` 的一条指令
- 译码 (decode): 根据指令集规范解析指令的语义 (顺便取出操作数)
- 执行 (execute): 执行指令、运算后写回寄存器或内存

最重要的就是实现 `idex()`

- 这就是 PA 里你们最挣扎的地方 (囊括了整个手册)

```c
int main() {
  while (!is_halt(M[pc])) {
    idex();
  }
}
```

代码例子 1

```
void idex() {
  if ((M[pc] >> 4) == 0) {
    R[(M[pc] >> 2) & 3] = R[M[pc] & 3];
    pc++;
  } else if ((M[pc] >> 4) == 1) {
    R[(M[pc] >> 2) & 3] += R[M[pc] & 3];
    pc++;
  } else if ((M[pc] >> 4) == 14) {
    R[0] = M[M[pc] & 0xf];
    pc++;
  } else if ((M[pc] >> 4) == 15) {
    M[M[pc] & 0xf] = R[0];
    pc++;
  }
}
```

代码例子 2

是否好一些?

- 不言自明? 不言自证?

```
void idex() {
  u8 inst = M[pc++];
  u8 op = inst >> 4;
  if (op == 0x0 || op == 0x1) {
    int rt = (inst >> 2) & 3, rs = (inst & 3);
    if      (op == 0x0) R[rt]  = R[rs];
    else if (op == 0x1) R[rt] += R[rs];
  }
  if (op == 0xe || op == 0xf) {
    int addr = inst & 0xf;
    if      (op == 0xe) R[0]    = M[addr];
    else if (op == 0xf) M[addr] = R[0];
  }
}
```

代码例子 3 (YEMU 代码)

```
typedef union inst {
  struct { u8 rs  : 2, rt: 2, op: 4; } rtype;
  struct { u8 addr: 4,        op: 4; } mtype;
} inst_t;
#define RTYPE(i) u8 rt = (i)->rtype.rt, rs = (i)->rtype.rs;
#define MTYPE(i) u8 addr = (i)->mtype.addr;

void idex() {
  inst_t *cur = (inst_t *)&M[pc];
  switch (cur->rtype.op) {
  case 0b0000: { RTYPE(cur); R[rt]   = R[rs];   pc++; break; }
  case 0b0001: { RTYPE(cur); R[rt]  += R[rs];   pc++; break; }
  case 0b1110: { MTYPE(cur); R[RA]   = M[addr]; pc++; break; }
  case 0b1111: { MTYPE(cur); M[addr] = R[RA];   pc++; break; }
  default: panic("invalid instruction at PC = %x", pc);
  }
}
```

有用的 C 语言特性

Union / bit fields

```
typedef union inst {
  struct { u8 rs  : 2, rt: 2, op: 4; } rtype;
  struct { u8 addr: 4,        op: 4; } mtype;
} inst_t;
```

指针

- 内存只是个字节序列
- 无论何种类型的指针都只是地址 + 对指向内存的解读

```
inst_t *cur = (inst_t *)&M[pc];
  // cur->rtype.op
  // cur->mtype.addr
  // ...
```

小结

如何管理“更大”的项目 (YEMU)?

- 我们分多个文件管理它
  - `yemu.h` - 寄存器名；必要的声明
  - `yemu.c` - 数据定义、主函数
  - `idex.c` - 译码执行
  - `Makefile` - 编译脚本 (能实现增量编译)
- 使用合理的编程模式
  - 减少模块之间的依赖
    - `enum { RA, ... , NREG }`
  - 合理使用语言特性，编写可读、可证明的代码
    - `inst_t *cur = (inst_t *)&M[pc]`
- NEMU 就是加强版的 YEMU

更多的计算机系统模拟器

`am-kernels/litenes`

- 一个“最小”的 NES 模拟器
- 自带硬编码的 ROM 文件

`fceux-am`

- 一个非常完整的高性能 NES 模拟器
- 包含对卡带定制芯片的模拟 (`src/boards`)

`QEMU`

- 工业级的全系统模拟器
  - 2011 年发布 1.0 版本
  - 有兴趣的同学可以 **RTFSC**
- 作者：传奇黑客 **Fabrice Bellard**

End.

永远不要停止对好代码的追求

~~再编下去就要单身一辈子了~~