“造轮子”的方法与乐趣

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

《计算机系统基础》课到底学了啥?

本讲内容

- 一个关于编译运行的问题
- 那些年我们造过的轮子

一个关于编译运行的问题

回到第一次 C 语言课

在 IDE 里，为什么按一个键，就能编译运行？

- 编译、链接
  - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
- 加载执行
  - `./a.out`

现在，一位同学对这个过程提出了质疑

- 我不信！我就觉得是 gcc 一个程序直接搞定的
  - 道理上完全可以这么实现
- 如何说服这位同学？

答案：用工具！

观察 trace: 查看 gcc 是否调用了其他命令

- `strace -f -qq gcc a.c 2>&1 | vim -`
  - 可以使用 `grep` (shell) 或 `:g!` (vim) 筛选感兴趣的系统调用

调试 gcc: 查看每一个阶段的中间结果

- 在哪里打断点？
  - **gdb-internals.txt**

课堂上见过的轮子们

编译器 (.c → .s)

同“表达式求值问题” (PA1)

- 编译器 (OJ 题)：输入一个字符串，输出计算它的汇编代码
  - 表达式 `(a + b) * (c + d)`
  - → 指令序列 (stack machine)
    - `push $a; push $b; add; push $c; push $d; add; mul`

现代编译器

- 预编译 → 词法分析 → 语法树 → IR 中间代码 → 多趟优化 (内联、传播/折叠、删除冗余) → 指令生成/寄存器分配

汇编器 (.s → .o)

除去预编译，汇编代码和指令几乎一一对应

- 根据指令集手册规定翻译
- 生成符合 ELF 规范的二进制目标文件
  - `printf("\x7f\x45\x4c\x46");...`

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000
00000020: 0000 0000 0000 0000 b81e 0000 0000 0000
00000030: 0000 0000 4000 0000 0000 4000 0f00 0e00
00000040: 5548 89e5 4883 ec10 4889 7df8 4889 75f0
00000050: 488b 55f0 488b 45f8 4801 d048 8905 0000
00000060: 0000 488b 0500 0000 00ba 0d00 0000 4889
00000070: c648 8d3d 0000 0000 b800 0000 00e8 0000
00000080: 0000 488b 0500 0000 0048 8b10 4889 1500
00000090: 0000 008b 5008 8915 0000 0000 0fb6 400c
```

链接器 (.o ➡ a.out)

按照 ld script 指示完成二进制文件构造、符号解析和重定位

- `gcc a.c -Wl,--verbose`
  - “.”代表当前位置；基本功能：粘贴；定义符号

```
SECTIONS {
  . = 0x100000;
  .text   : { *(entry) *(.text*) }
  etext = .; _etext = .;
  .rodata : { *(.rodata*) } .data : { *(.data*) }
  edata = .; _edata = .;
  .bss    : { *(.bss*) }
  _start_start = ALIGN(4096);
  . = _start_start + 0x8000;
  _stack_pointer = .;
  end = .; _end = .;
  _heap_start = ALIGN(4096); _heap_end = 0x8000000;
}
```

加载器

(PA 里做过了)

编译、链接、加载

- 好像没那么难啊 (都是表达式求值 + 翻译)

单元测试框架

YEMU 中的神奇代码

```
#define TESTCASES(_) \
  _(1, 0b11100111, random_rm, ASSERT_EQ(newR[0], oldM[7]) ) \
  _(2, 0b00000100, random_rm, ASSERT_EQ(newR[1], oldR[0]) ) \
  _(3, 0b11100101, random_rm, ASSERT_EQ(newR[0], oldM[5]) ) \
  _(4, 0b00010001, random_rm, ASSERT_EQ(newR[0], oldR[0] + oldR[1]) 
  _(5, 0b11110111, random_rm, ASSERT_EQ(newM[7], oldR[0]) )
```

- ~~相信大家都没有做好单元测试~~

Differential Testing

设置好状态；各走一步

```
for i in range(int(sys.argv[1])):
  print('\n'.join(['si'] + [f'p ${r}' for r in ['eax', ...]]))
```

```
N=10000
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
     <(python3 cmdgen.py $N | ./x86-nemu       img)
```

调试器 GDB

好奇它是怎么实现的？

- 考虑核心功能：在任意 PC 的断点
- 其他功能都可以基于断点实现
  - 单步调试：在下一条指令打断点
  - watch point: 单步调试 + 检查条件

Trace/Profiler

strace: 刚才已经展示过威力了

- 如何实现？

静态分析工具 Lint

大家见到的第一个 lint: `gcc -Wall -Werror`

- cppcheck
- **Cert C Coding Standard**
  - 自带 **checker**

动态分析工具 Sanitizer

本质：运行时额外增加的 `assert()`

- 回顾调试理论

一些非常实用的 assertions

- `assert(IS_GUESTPTR(ptr));`
  - `(PMEM_MAP_START <= (x) && (x) < PMEM_MAP_END)`
- `assert(IS_SMALLINT(x));`
  - `(0 <= (x) && (x) <= 100)`
- `assert(IS_BOOL(ptr));`
  - `((x) == 0 || (x) == 1)`

总结

我们学到了什么？

> “程序是个状态机。”
> “我们可以观测状态机的设计、实现和执行。”

能实现几乎任何工具 (的玩具版)

- 并且能在需要的使用善用它们
  - 编译器 (gcc)
  - 汇编器 (as)
  - 链接器 (ld)
  - 调试器 (gdb)
  - 追踪器 (strace/ltrace)
  - profiler (perf)

Cheers!

(你们完成了了不起的事情！)