系统编程与基础设施

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

感受到 PA 的恶意了吗?

- 不就是写几行代码么，怎么……怎么写不对啊……
- 这就是为什么 PA 要搞那么麻烦：又是 Makefile，又是各种项目/工具
- 没有适当的基础设施，PA 的完成率会大幅降低

- 开发/调试的基础设施
- 系统编程：基础设施
- Differential testing 代码导读

开发/调试的基础设施

你们是怎么写程序的？

虽然很多同学已经配置好了良好的编程环境，但依然有很多同学在“面向浪费时间编程”

- yzh 推荐 vim/tmux 的原因是大家用 IDE 容易迷失自我 (关注表象，而不知道内在是如何工作的)
- 在你搞清楚的前提下，也可以解放自我

```
$ gcc a.c
a.c: In function ‘main’:
a.c:5:1: error: ‘a’ undeclared (first use in this function)
$ vi a.c
$ gcc a.c
$ ./a.out
1 2             // 你输入的
zsh: segmentation fault (core dumped)  ./a.out
...
```

基础设施的本质

通过适当的配置、脚本减少思维中断的时间，提高连贯性，保持短时记忆活跃

- `make` (fresh build) → 4s，已被打断
- `make` (parallel) → 0.5s

基本原则：如果你认为有提高效率的可能性，一定有人已经做了

- 每次都 `make -j8`?
  - 或者 `alias make='make -j8'`
  - 在 Makefile 里加一行 `MAKEFLAGS += -j 8` (better)
    - 配置一键编译运行 → 1s 内完成

心态分析

本质上，这些都不是难事，STFW 随手即来，但大家通常做不好

- 尚未 GET STFW 的技能
  - 在 hosts 中屏蔽百度 (或修改默认搜索引擎)
- 惰性
  - 键入 `make -j8`: 增加 1s 时间
  - STFW: 至少需要几分钟，而且有不少失败尝试 (短期收益是负的，尤其是上课 workloads 已经很重的前提下)

心态分析 (cont'd)

> 克服惰性可以使你快速成长。

在很多小事上，可能并不带来显著的收益

- 每次键入 `make -j8` 可能并不显著缩短 PA 完成的时间
- 但久而久之成为习惯，你就总是想着改进基础设施

系统编程：建立基础设施

复习：测试/调试的理论

Fault → Error → Failure

对于 PA 来说，failure 是显而易见的：

- Segmentation Fault 了，fail
- HIT BAD TRAP 了，fail
- 马里奥/仙剑不能跑嘛，fail

我已经调了很久了，但就是找不到那个导致 error 的指令啊

- 怎样找到 error 发生的位置？
  - 二分法似乎还是太麻烦了？

抱大腿：一种想法

如果学长/同学已经有一份正确的代码，能不能借助这个代码快速诊断出自己代码的问题？

- 同学们是非常智慧的，在 ICS-PA 创立的初期发明了替换调试法
  - PA 是分模块实现的 (若干个 .c)
  - 从自己的代码开始，逐个替换进大腿同学的 .c
  - 第一个替换后通过测试的文件里有 bug

Cool!

- 可能的改进：以函数级进行替换

Delta Debugging

假设程序 P 会 fail，$P_{\text{大腿}}$ 会 pass

- 并且假设两份程序所有函数的行为都相同
  - 将 $P_{\text{大腿}}$ 中的函数 f 替换成 P 中的 f
    - 依然通过 ➞ f 实现正确
    - 否则 f 中有 bug

把类似的想法用在输入上

- 不断把输入的一部分去掉，直到不能触发 bug 为止
  - Anders Zeller and Ralf Hildebrandt. **Simplifying and isolating failure-inducing input**. *IEEE Transactions on Software Engineering*, 28(2), 2002.

大腿同学代码的另一种用法

大腿同学的代码还可以帮助我们直接定位到错误的指令！

- 只需要大腿的 compiled binary
- 就能指令级定位出错的位置

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

然后找到 log 第一个不一致的地方！

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

基础设施：其实没那么困难

每做的一点自动化都是在给大项目节约维护成本。

程序员们就是喜欢造轮子

- (日常管理) 效率低 → 熟练使用命令行工具/Python
- (项目管理) 你已经会 `gcc a.c` 了，但没法管理几十个文件 → make, 一键编译/运行/测试
- (代码编辑) 在代码里跳来跳去很麻烦 → IDE/配置 Vim/装插件
- (错误检查) 很容易犯低级错误 → `-Wall`, `-Werror`, `fsanitize=address`
- (代码调试) Segmentation Fault了 → gdb

Differential Testing

Differential Testing

> “同一套接口 (API) 的两个实现应当行为完全一致”

大腿同学 & 小腿同学：指令集的两套实现

- 还有什么现实中软件系统的例子？

你能找到两份独立实现的东西，都可以测试

- 浏览器 **Mesbah and Prasad, ICSE'11**
- GCC (vs clang), **Yang et al., PLDI'11**
- 文件系统, **Min et al., SOSP'15**
- 数据库 (2020 年时的预言成真) **Rigger and Su, OSDI'20**
- Gcov (vs llvm-cov), 我们的工作
  - 真的能在 gcc/llvm 里发现很多 bugs

NEMU: 实现 Differential Testing

刚才我们已经给大腿的代码实现了一个简单版本的 diff-testing
真正的大腿：QEMU

- ICS PA = 缩水版 QEMU
  - 实际上 PA 就是简化 (教学) 版的 QEMU

diff-testing 实验

- 以前我们一直都只是自己写自己的程序，调用库函数
- diff-testing 是和其他程序 (QEMU, gdb) 协作/交互的例子

NEMU Differential Testing: 原理

```
# gen-cmds: 不断生成 si; p $eax; p $ebx; ...
diff-stream <(gen-cmds | nemu) <(gen-cmds | qemu-system-i386)
```

改进版的“抱大腿”代码，不需二分查找

- 同时启动两个 NEMU (QEMU)
- 在第一个输出不同时停止

(QEMU Monitor 展示)

- `-serial mon:stdio` 启动 monitor!

NEMU Differential Testing: 原理 (cont'd)

问题分析：我们需要使 QEMU 像 NEMU 一样执行指令！

- QEMU 实现了 gdb 的协议
- 协议格式同 monitor

qemu-diff 实现：

1. 启动 QEMU 并配置它进入与 PA 类似的模式
2. 用 gdb 连接 QEMU
3. 用 `gdb_si()` 在 QEMU 中执行一条指令
4. 比较指令执行后寄存器是否有区别

代码导读

首先 PA 代码 (dut.c) 里没有 diff-test 的实际代码，只有一堆函数指针：

```
void (*ref_difftest_memcpy_from_dut)(paddr_t dest, void *src, size_
void (*ref_difftest_getregs)(void *c) = NULL;
void (*ref_difftest_setregs)(const void *c) = NULL;
void (*ref_difftest_exec)(uint64_t n) = NULL;
```

这些函数封装了参考实现的功能

- 既可以和 QEMU diff-test，也可以和 NEMU diff-test
- exec_wrapper() 中执行一条指令之后直接对比结果就行

```
#if defined(DIFF_TEST)
  difftest_step(ori_pc, cpu.pc);
#endif
```

代码导读 (cont'd)

经过 RTFM/RTFSC：

- `nemu/tools/qemu-diff` 是 differential testing 实际实现的目录

然后 RTFSC，看到了若干有用的函数：

- `gdb_connect_qemu`，看起来就是用来连接到 QEMU 的，创建一个到 127.0.0.1、端口是 1234 的 gdb 连接
- `gdb_si`，和 monitor 一样，单步执行指令
- `gdb_setregs`, `gdb_getregs`，好像复杂一点，不过就是用 `gdb_send()` 和 `gdb_recv()` 发送/接收消息

代码导读 (cont'd)

Differential testing 的初始化

- 我们可以用 QEMU + gdb 调试**这段代码**!
  - `-s -S` 启动 QEMU; gdb `target remote localhost:1234`

```
uint8_t mbr[] = {
  0xfa,                         // cli
  0x31, 0xc0,                   // xorw   %ax,%ax
  0x8e, 0xd8,                   // movw   %ax,%ds
  0x8e, 0xc0,                   // movw   %ax,%es
  0x8e, 0xd0,                   // movw   %ax,%ss
  0x0f, 0x01, 0x16, 0x44, 0x7c, // lgdt   gdtdesc
  0x0f, 0x20, 0xc0,             // movl   %cr0,%eax
  0x66, 0x83, 0xc8, 0x01,       // orl    $CR0_PE,%eax
  0x0f, 0x22, 0xc0,             // movl   %eax,%cr0
  0xea, 0x1d, 0x7c, 0x08, 0x00, // ljmp   $GDT_ENTRY(1),$start32
  ...
};
```

总结

系统编程的困难

在程序规模到达一定程度的时候，代码既难管理，也难写对

- 即便在项目文件之间浏览就已经非常耗时
  - 面对不熟悉的模块/API
  - 需要好的 IDE、代码折叠、第二块屏幕……
- 编程经验可以减少 bug，但很难完全消灭它们
  - 各类肉眼难以发现的低级错误 (`i` vs. `j`, `0x` vs `0b`, `int` with `unsigned`, ...)
  - 读错了手册 (忘记更新某个 flags, 记错 0/符号扩展, ...)
  - 逻辑上的错误 (使用一块已经释放的内存, 虚拟/物理内存地址访问错, ...)

做过 PA 的人就知道，“机器永远是对的”不是开玩笑的

- 你折腾一天，两天，可能就是多打了一个空格

基础设施：帮助你更快更好地生产代码

终极梦想：让计算机自动帮我们写程序

- 告诉计算机需求 → 计算机输出正确的代码
- 计算机科学的 holy grail 之一

现在我们还在软件自动化的初级阶段

- 计算机只能提供有限的自动化 (基础设施)
  - 集成开发环境 (IDE)
  - 静态分析
  - 动态分析
- 但这些基础设施已经从本质上改变了我们的开发效率
  - 你绝对不会愿意用记事本写程序的

基础设施：小结

当轮子都不够用的时候，我们就去造轮子

- 调试困难，有参考实现 → diff-test
- 难以贯通多门实验课 → AbstractMachine

轮子还不够用呢？

- diff-test 每秒只能检查 ~5000 条指令
- 中断和 I/O 具有不确定性
- 没有参考实现呢 → 一个新的研究问题在等着你

End.