

系统编程与基础设施

王慧妍

why@nju.edu.cn

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
...
```



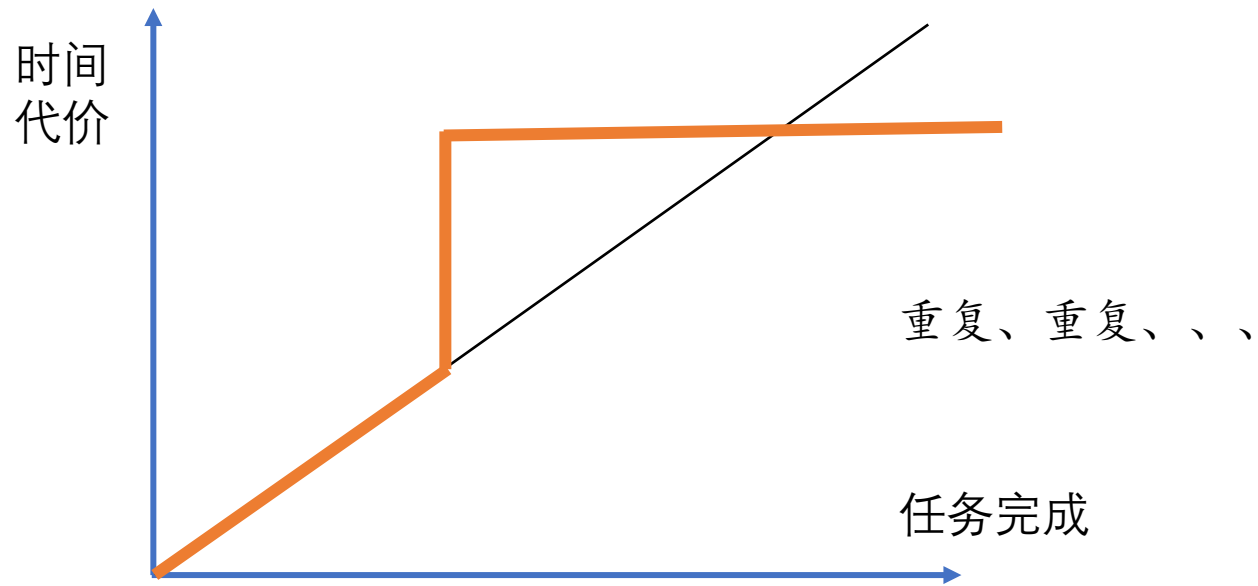
```
$ ./a.out  
3 3  
3 + 3 = 6  
$
```

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
...
```

时间分析



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

```
$ ls
abstract-machine  fceux-am  Makefile  README.md
am-kernels        init.sh    nemu       tags
$
```

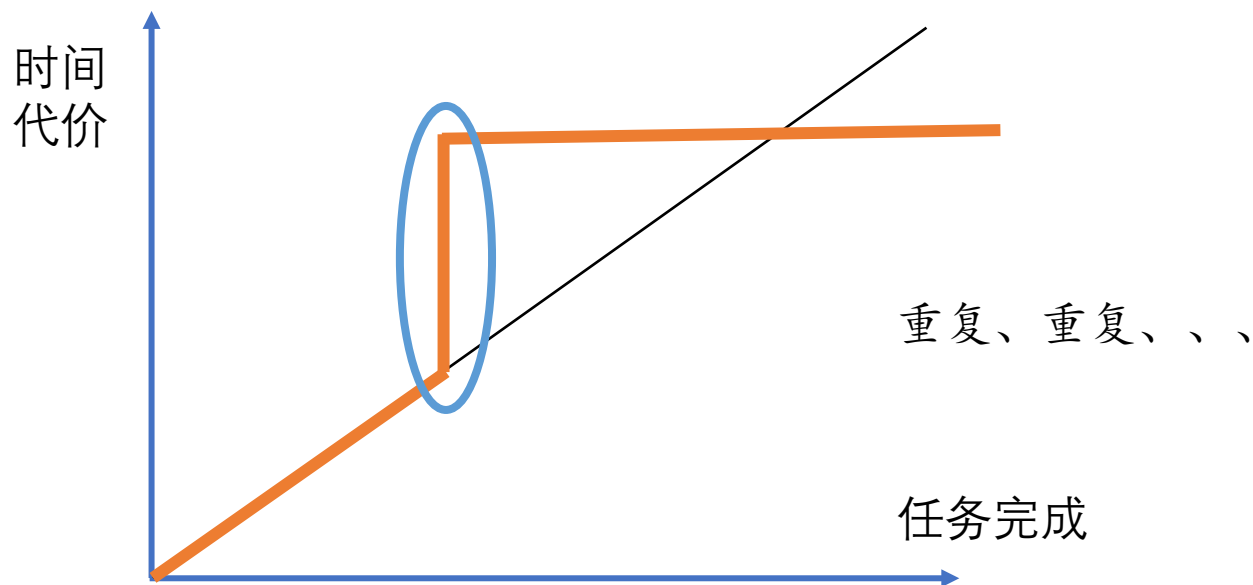
基础设施的本质

- 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

```
# Building fceux-image [x86-nemu]
# Building am-archive [x86-nemu]
# Building klib-archive [x86-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-x86-nemu.a
+ LD -> build/fceux-x86-nemu.elf
# Creating image [x86-nemu]
+ OBJCOPY -> build/fceux-x86-nemu.bin
$ export ARCH=x86-nemu
$ make
# Building fceux-image [x86-nemu]
+ CXX src/emufile.cpp
# Building am-archive [x86-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-x86-nemu.a
# Building klib-archive [x86-nemu]
+ LD -> build/fceux-x86-nemu.elf
# Creating image [x86-nemu]
+ OBJCOPY -> build/fceux-x86-nemu.bin
$ export ARCH=native
$ make
```

心态分析

- 本质上，这都不是难事，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)

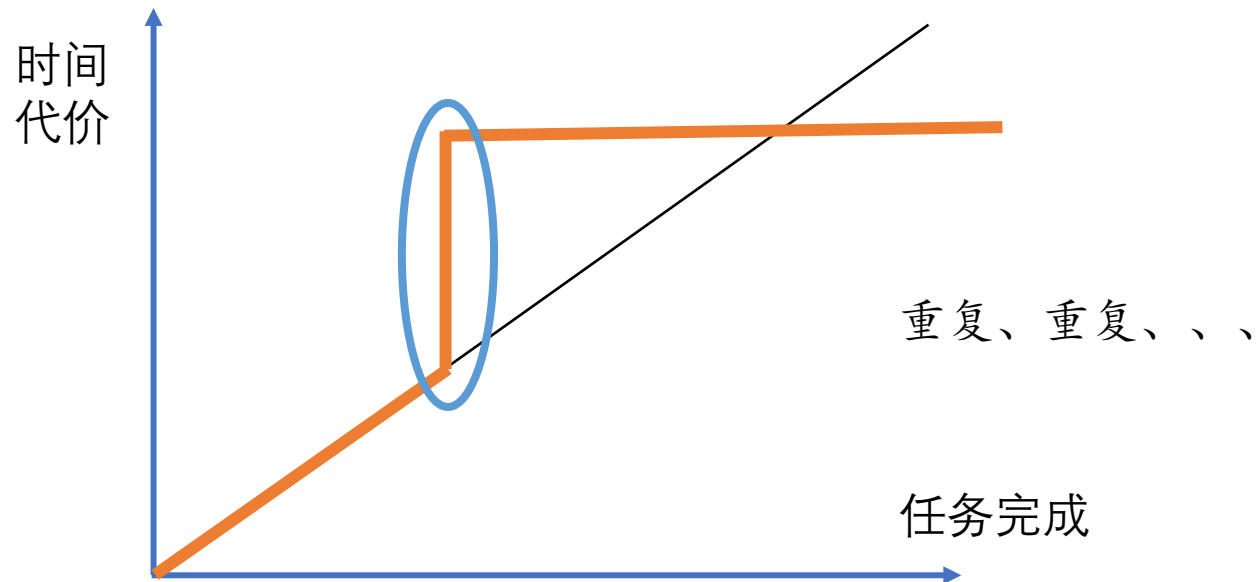


```
$ vim Makefile
```

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



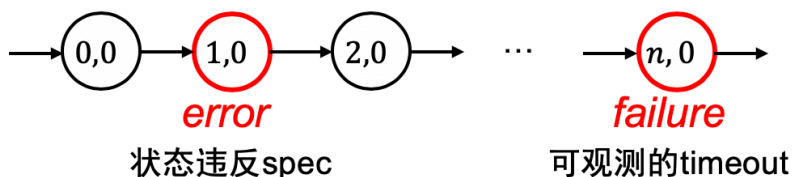
系统编程：建立基础设施

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？

```
for (int i = 0; i < n; i++) fault 程序bug  
for (int j = 0; j < n; i++)  
...
```



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

大腿同学



小腿同学



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 $\rightarrow f$ 实现正确
 - 否则 $\rightarrow f$ 实现有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.
 - 不断把输入的一部分去掉, 直到行为不再保持
 - Md Rafiqul Islam Rabin et al. [Understanding Neural Code Intelligence through Program Simplification](#). ICSE'21

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', ...]]))
```

\$ █

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

```
N=10000
```

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

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 [Rigger and Su, OSDI'20](#)
 - Gcov (vs llvm-cov)真的能在 gcc/llvm 里发现很多 bugs
 - DL library [Pham et al., ICSE'19](#)

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; ...
```

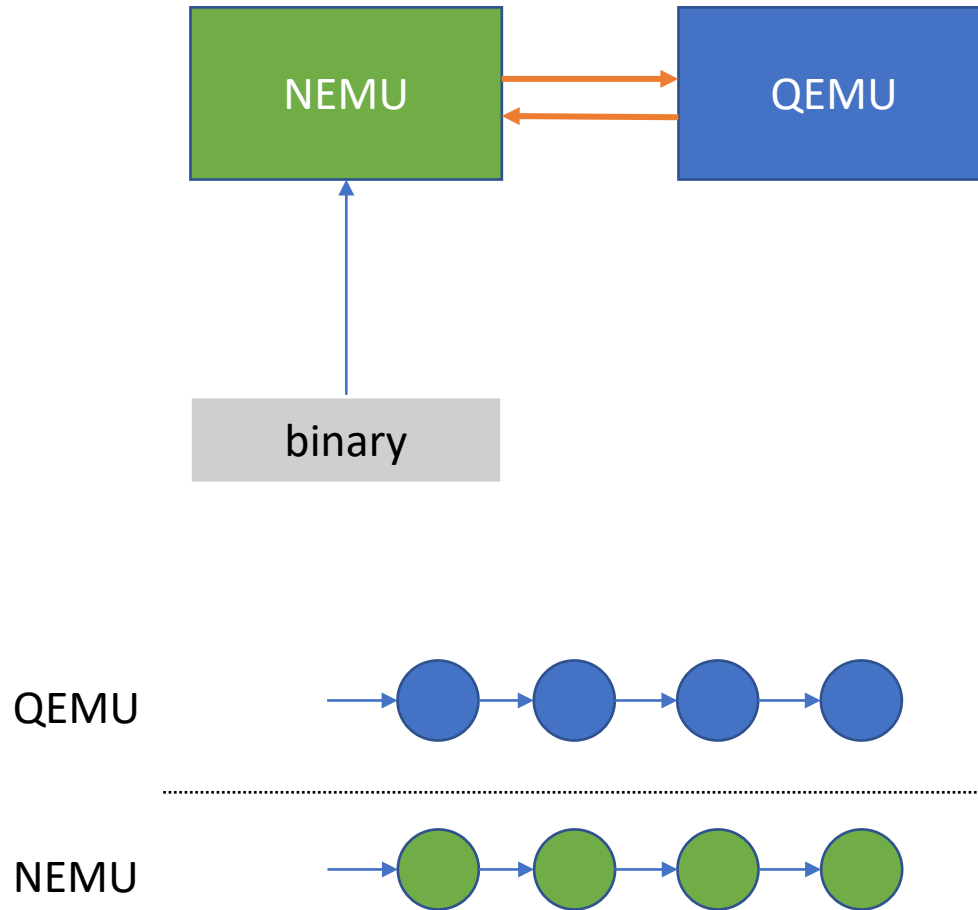
```
diff-stream <(gen-cmds | nemu) <(gen-cmds | qemu-system-i386)
```

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

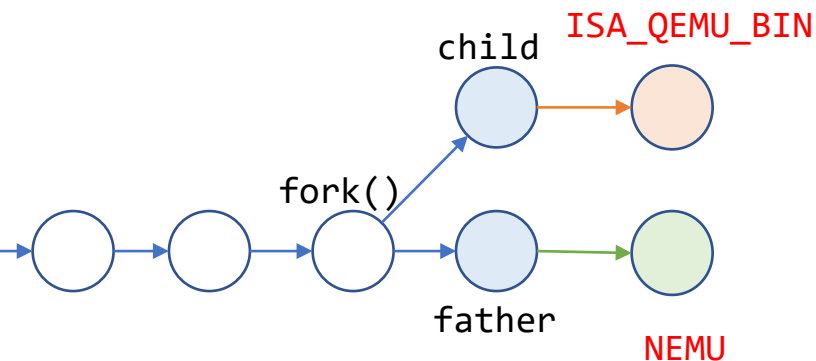
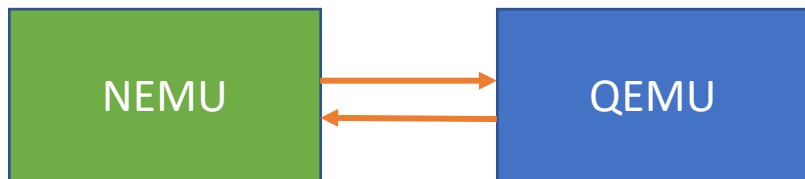
NEMU和QEMU协作



```
$ vim nemu/
```

```
$ █
```

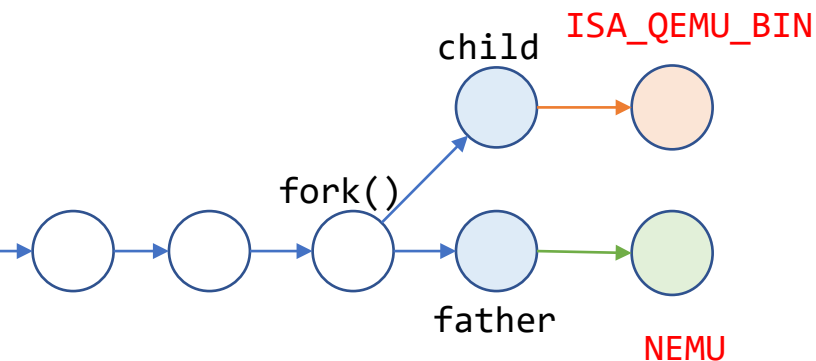
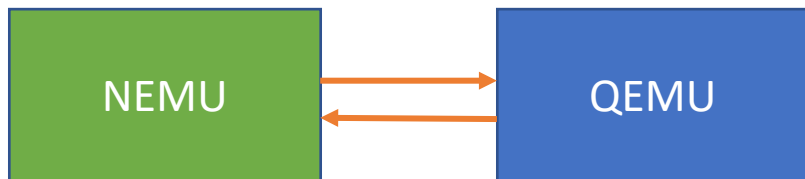
NEMU和QEMU协作



```
9 #define ISA_QEMU_BIN "qemu-system-riscv32"
10 #define ISA_QEMU_ARGS "-bios", "none",
11 #elif defined(CONFIG_ISA_riscv64)
12 #define ISA_QEMU_BIN "qemu-system-riscv64"
```

```
38 void difftest_init(int port) {
39     char buf[32];
40     sprintf(buf, "tcp::%d", port);
41
42     int ppid_before_fork = getpid();
43     int pid = fork();
44     if (pid == -1) {
45         perror("fork");
46         assert(0);
47     }
48     else if (pid == 0) {
49         // child
50
51         // install a parent death signal in the child
52         int r = prctl(PR_SET_PDEATHSIG, SIGTERM);
53         if (r == -1) {
54             perror("prctl error");
55             assert(0);
56         }
57
58         if (getppid() != ppid_before_fork) {
59             printf("parent has died!\n");
60             assert(0);
61         }
62
63         close(STDIN_FILENO);
64         execlp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-gdb", buf,
65              "-serial", "none", "-monitor", "none", NULL);
66         perror("exec");
67         assert(0);
68     }
69     else {
70         // father
71
72         gdb_connect_qemu(port);
73         printf("Connect to QEMU with %s successfully\n", buf);
74
75         atexit(gdb_exit);
76
77         init_isa();
78     }
79 }
```

NEMU和QEMU协作



```
bool gdb_connect_qemu(int port) {  
    // connect to gdbserver on localhost port 1234  
    while ((conn = gdb_begin_inet("127.0.0.1", port)) == NULL) {  
        usleep(1);  
    }  
    return true;  
}
```

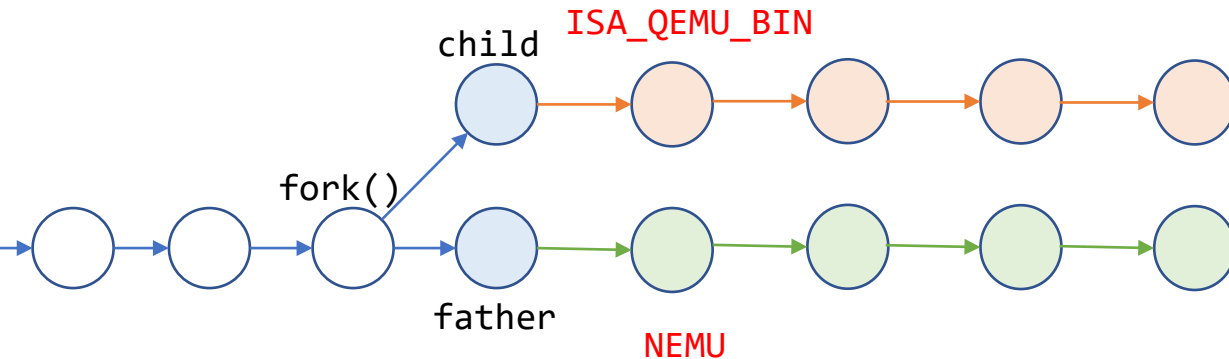
```
38 void difftest_init(int port) {  
39     char buf[32];  
40     sprintf(buf, "tcp::%d", port);  
41  
42     int ppid_before_fork = getpid();  
43     int pid = fork();  
44     if (pid == -1) {  
45         perror("fork");  
46         assert(0);  
47     }  
48     else if (pid == 0) {  
49         // child  
50  
51         // install a parent death signal in the child  
52         int r = prctl(PR_SET_PDEATHSIG, SIGTERM);  
53         if (r == -1) {  
54             perror("prctl error");  
55             assert(0);  
56         }  
57  
58         if (getppid() != ppid_before_fork) {  
59             printf("parent has died!\n");  
60             assert(0);  
61         }  
62  
63         close(STDIN_FILENO);  
64         execlp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-gdb", buf,  
65             "-serial", "none", "-monitor", "none", NULL);  
66         perror("exec");  
67         assert(0);  
68     }  
69     else {  
70         // father  
71  
72         gdb_connect_qemu(port);  
73         printf("Connect to QEMU with %s successfully\n", buf);  
74  
75         atexit(gdb_exit);  
76  
77         init_isa();  
78     }  
79 }
```

- 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)
```



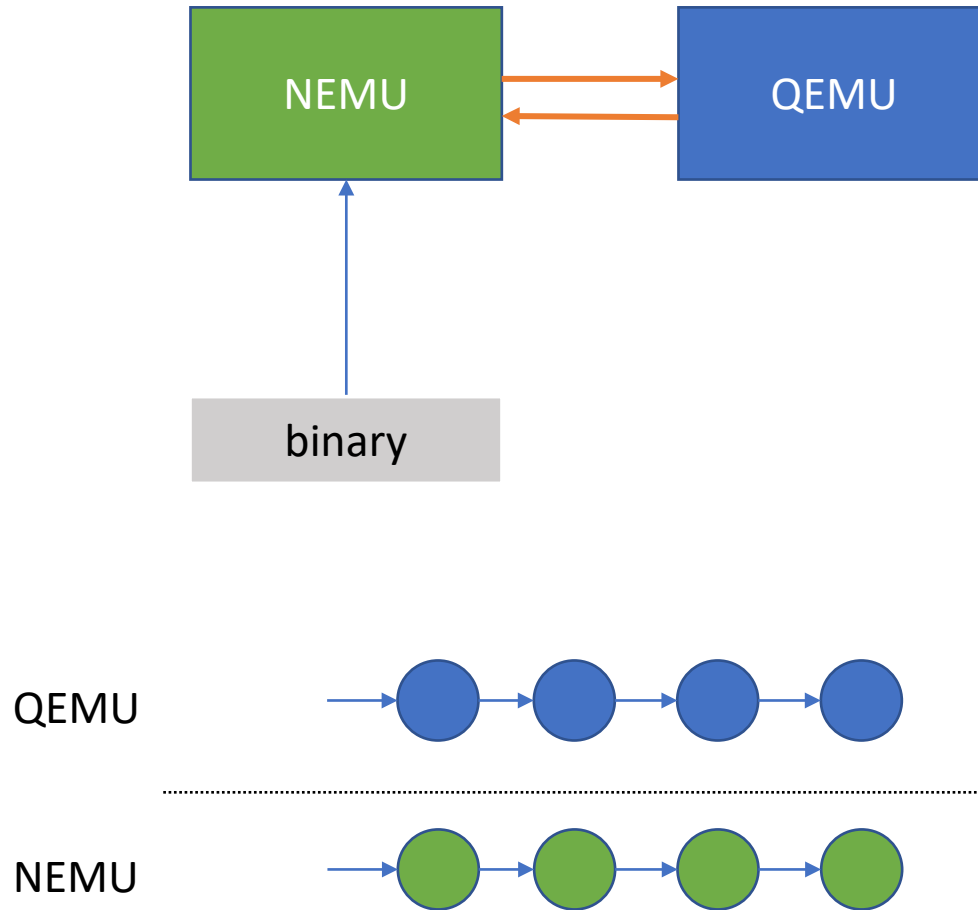
```
ssfully\n", buf);
```

```
int64_t n) {
    i();
```

```
47 bool gdb_getregs(union isa_gdb_regs *r) {
48     gdb_send(conn, (const uint8_t *) "g", 1);
49     size_t size;
50     uint8_t *reply = gdb_recv(conn, &size);
51
52     int i;
53     uint8_t *p = reply;
54     uint8_t c;
55 +--- 7 lines: for (i = 0; i < sizeof(union isa_gdb_
```

```
bool gdb_si() {
    char buf[] = "vCont;s:1";
    gdb_send(conn, (const uint8_t *) buf, strlen(buf));
    size_t size;
    uint8_t *reply = gdb_recv(conn, &size);
    free(reply);
    return true;
}
```


NEMU和QEMU协作



\$

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course14

why@why-VirtualBox: ~/Documents/jyy-wiki-git/wiki

\$

英 简 拼

Right Shift + Right Alt

代码导读

- 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 gdt_desc  
    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  
    ...  
};
```

```
$ ls  
a.cpp  a.out  in.txt  mbr  
$
```

```
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000150: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000160: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000170: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000180: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000190: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa .....U.
```

\$ qemu-system-i386

^Cqemu-system-i386: terminating on signal 2

\$ qemu-system-i386 mbr

WARNING: Image format was not specified for 'mbr' and probing guessed raw.

Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.

Specify the 'raw' format explicitly to remove the restrictions.

\$

```
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 0000  . . . . .
```

0000

0000

0000

0000

0000
00000000
0000

0000

0000

0000

0000

0000

0000

0000

0000

[21]

[2]

\$ qe

^Cqe

\$ qe

WARN

100

bloo

100

 $\wedge$ case

†

\$ 46
WADN

WARI

—

b loc

[illegible]

[

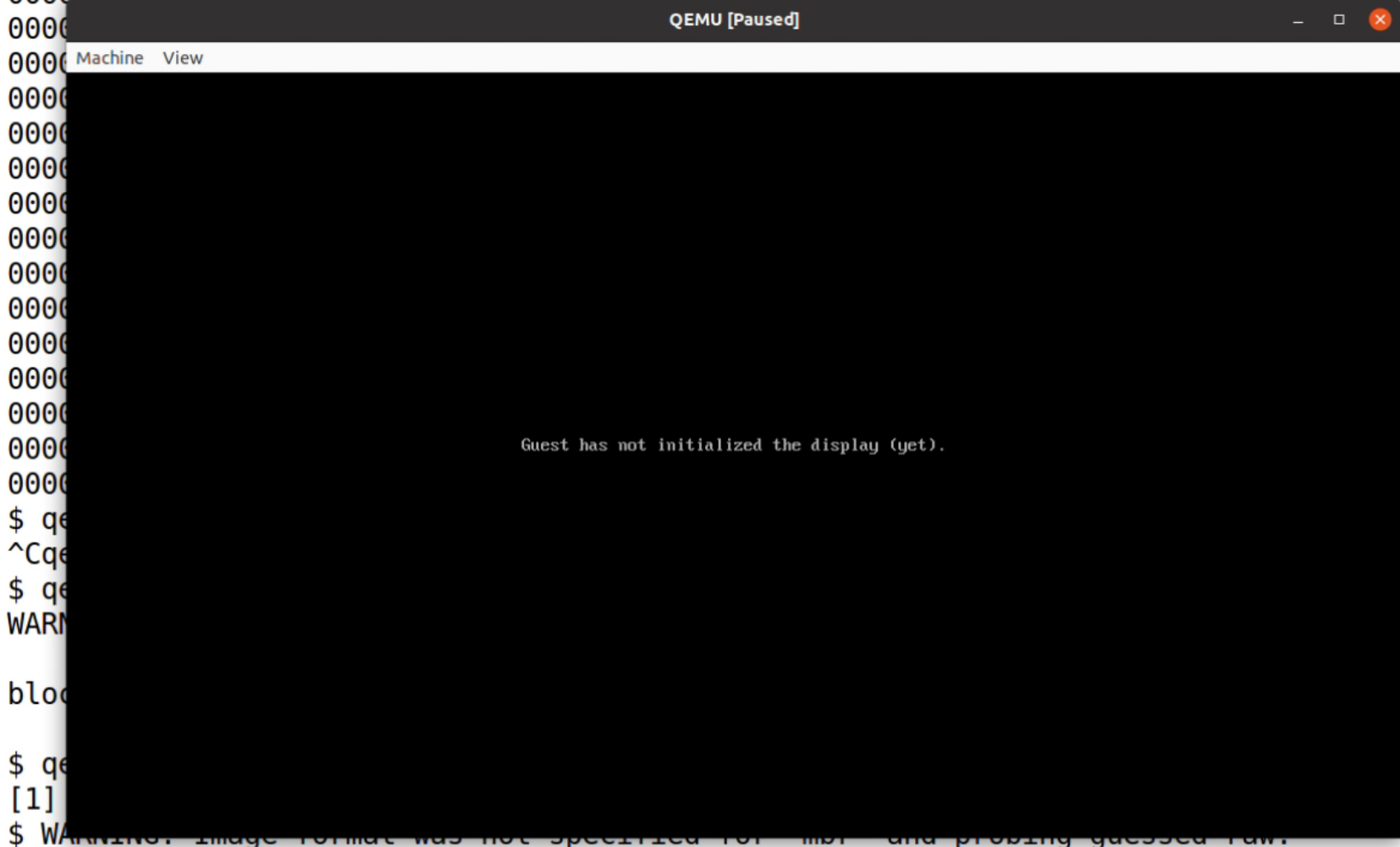
QEMU [Paused] - Press Ctrl+Alt+G to release grab

Machine View

Guest has not initialized the display (yet).

Specify the 'raw' format explicitly to remove the restrictions.

```
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000110: 0000 0000 0000 0000 0000 0000 0000 0000
```



```
$ qe
^Cqe
$ qe
WARN
rations on
blo
$ qe
[1]
$ WARN: Image format was not specified for 'img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on
block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

□


```

B+>0x7c00 cli
0x7c01 xor %eax,%eax
0x7c03 mov %eax,%ds
0x7c05 mov %eax,%es
0x7c07 mov %eax,%ss
0x7c09 lgdtl (%esi)
0x7c0c inc %esp
0x7c0d jl 0x7c1e
0x7c0f and %al,%al
0x7c11 or $0x1,%ax
0x7c15 mov %eax,%cr0
0x7c18 ljmp $0xb866,$0x87c1d
0x7c1f adc %al, (%eax)
0x7c21 mov %eax,%ds
0x7c23 mov %eax,%es
0x7c25 mov %eax,%ss
0x7c27 jmp 0x7c27
0x7c29 lea 0x0(%esi),%esi

```

remote Thread 1.1 In:

L?? PC: 0x7c00

Breakpoint 1 at 0x7c00

(gdb) c

Continuing.

Breakpoint 1, 0x00007c00 in ?? ()

(gdb) x/10x 0x7c00

0x7c00:	0x8ec031fa	0x8ec08ed8	0x16010fd0	0x200f7c44
0x7c10:	0xc88366c0	0xc0220f01	0x087c1dea	0x10b86600
0x7c20:	0x8ed88e00	0xebd08ec0		

(gdb) █

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)(paddr_t addr, void *buf, size_t n, bool direction) = NULL;  
void (*ref_difftest_regcpy)(void *dut, bool direction) = NULL;  
void (*ref_difftest_exec)(uint64_t n) = NULL;  
void (*ref_difftest_raise_intr)(uint64_t NO) = 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

总结

系统编程的困难

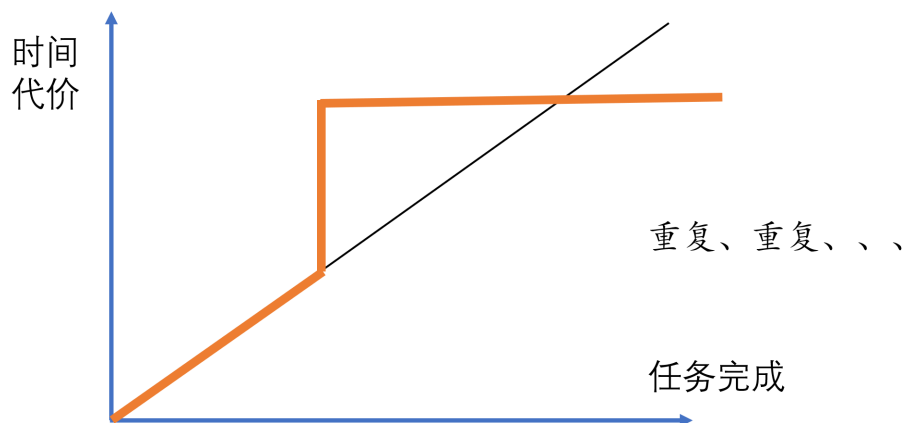
- 在程序规模到达一定程度的时候，代码既难管理，也难写对
 - 即便在项目文件之间浏览就已经非常耗时
 - 面对不熟悉的模块/API
 - 需要好的 IDE、代码折叠、第二块屏幕.....
 - 编程经验可以减少 bug，但很难完全消灭它们
 - 各类肉眼难以发现的低级错误 (i vs. j, 0x vs 0b, int with unsigned, ...)
 - 读错了手册 (忘记更新某个 flags, 记错 0/符号扩展, ...)
 - 逻辑上的错误 (使用一块已经释放的内存, 虚拟/物理内存地址访问错, ...)
- 做过 PA 的人就知道，“机器永远是对的”不是开玩笑的
 - 你折腾一天，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 → Abstract Machine



- 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.