

# 操作系统课里的程序设计

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组





# 首先.....

- 程序员两大错觉
  - “**不是我的锅**” ——机器永远是对的
    - 出了错当你怀疑你绝对没问题的时候，都是你的问题。
  - “**肯定没问题**” ——未测代码永远是错的
    - 当你觉得代码写的对的，它往往是错的。



# 首先.....

- 所以就是自己背锅啦!
- 一些可能的情况(非常罕见, 目前尚未遇见)
  - 操作系统/库可能有bug → 那就是按写的C程序执行
  - 编译器可能有bug → 机制也按编译出的二进制代码执行
  - 处理器可能有bug → 但也就是按照有bug的电路(机器)执行
- 总之机器不管多任性, 都是对的, 因为最终物理世界的规律存在但我们并不知道
  - 只是越深的bug越难找



# 然后，背锅的正确姿势是什么？

- 一个bug de两天，应该意识到是方法出了问题
  - fault → failure → error
  - 犯的是fault，看到的是error
  - 要找到最早的failure
- (羞 ♂ 耻的历史) 回到我以前一个bug调两天的时候.....
  - 那时候我还是个面向OJ编程的菜



# Case #1 为什么我连strace都用不了？

- 我都sudo了，还是不行，那我做个毛线的实验？？？

```
# strace ls
strace: ptrace(PTRACE_TRACEME, ...): Operation not
permitted
+++ exited with 1 +++
```

- 亲测：Google, Bing, Baidu都能得到有用的信息
  - Google #1: Unable to strace in container with docker 1.1
  - Bing #1: strace -f strace /bin/l failed
  - Baidu #1: ptrace: Operation not permitted. - CSDN博客
  - 百度的第一条没有提供任何原因和解决方案



## Case #2 为什么strace不结束?

```
16 int pid = fork();
17 if (pid > 0) {
18     close(READ_END(fd));
19     dup2(WRITE_END(fd), STDERR_FILENO);
20     close(WRITE_END(fd));
21     static char *argv[] = {"strace", "ls", NULL};
22     execvpe("strace", argv, environ);
23     perror("execve");
24 } else {
25     close(WRITE_END(fd));
26     while (1) {
27         char buf[1024];
28         nread = read(READ_END(fd), buf, sizeof(buf) - 1);
29         buf[nread] = '\0';
30         if (nread <= 0) break;
31         printf("> %s", buf);
32     }
33 }
```



# 为什么.....

- 我要背这些锅？
- (怒砸键盘)

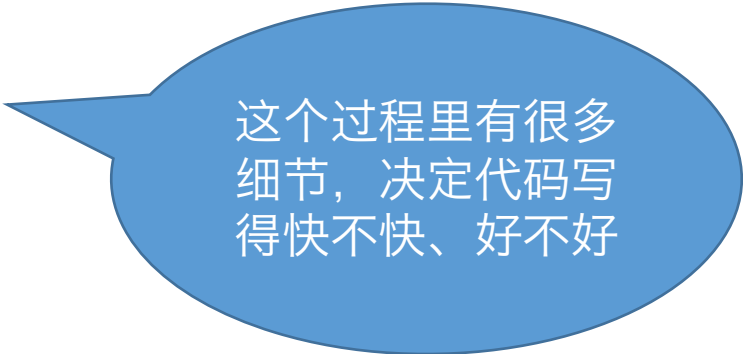
假装上一堂程序设计课





# 一句话背锅准则：用对的方式做对的事

- 这当然是句废话
- 重要的是“**用什么方式**”和“**做什么事**”
- 编程我们都会(甚至没有教过，让大家自学)
  - 用文本编辑器码个程序
  - 在命令行里编译运行
  - 错了再改呗



这个过程里有很多细节，决定代码写得快不快、好不好



# 就拿编译运行的例子.....

- 新手
  - (1) 退出vim (2) make (3) ./perf ls (4) 调bug
  - 命令就打了几十上百次
- 省时间的一万种方法
  - 现代IDE都支持按键映射——一键完成make
  - 一键顺便完成运行
  - 一键都顺便省了 ← 参考代码就是这么编写的



# 编程的正确方式

- Programs are meant to be read by humans and only incidentally for computers to execute —— D. E. Knuth
- “写那些能证明正确性的代码”
  - 在每一个位置都能明确地知道什么是正确的
  - 代码表述的字里行间都透着证明的气息



# 不好的例子 (Anti-Pattern)

- 面向OJ编程
  - 我大概知道用什么来完成这个功能
    - 条件判断、系统调用、库函数.....
  - 那就先写下来试试呗(比如循环终止的条件是 $x \leq y$ )
  - 测试通过了就当它是对的
  - 没通过的话再乱改改(改成 $x < y$ )
- 这代表了一类程序自动化技术
  - 广受诟病的GenProg – xjb改, 改到测试通过为止



# 系统编程的正确思路

- 用充分理解行为的API，充分理解程序
  - 我不知道用什么API：看别人怎么用的
  - 我不知道怎么用API：看别人怎么用的 & RTFM
  - 我不知道怎么处理错误：看别人怎么用的

```
16 int pid = fork();
17 if (pid > 0) {
22     execvpe("strace", argv, environ);
23     perror("execve");
24 } else {
26     while (1) {
28         nread = read(STDIN_FILENO, buf, sizeof(buf) - 1);
32     }
33 }
```

什么？让老爸去  
干马仔活，儿子  
潇洒走一回？

比  
printf("execv  
e failed\n");  
更科学点



# 程序员的嗅觉

- 如何理解程序的行为？
- 如何知道程序是否按照既定的方向直行？
- 出错以后如何根据各种信息诊断bug的位置？



# It's a Long Way

- 初学编程的时候对API没有任何理解
  - 只知道内存、数据、数据结构.....
- 时间久了以后就知道API中有很多约定
  - 返回是指针的，NULL表示失败
  - 返回是int的， $< 0$  (-1)表示失败，设置errno
- 每个参数、概念都有意义

万一  
strace在  
别处？

此处有黑  
人问号

```
execve("/usr/bin/strace", p_argv, NULL);
```



# API: 水面下的冰山

这就是为什么建议大家不要极限操作

- `man exec` – 提示了 `environ` 的存在

ENVIRON(7) Linux Programmer's Manual

## NAME

`environ` – user environment

## SYNOPSIS

```
extern char **environ;
```

## DESCRIPTION

The variable `environ` points to an array of pointers to strings called the "environment".



# UNIX笑话重读

- **Unix is user-friendly** — it's just choosy about who its friends are.
- 成为UNIX的朋友，从RTFM/RTFSC开始



# 示例分析

pattern?  
pattern1?

```
54 regcomp(&reg, pattern1,  
cflags);  
55 status = regexec(&reg, buf,  
nmatch, pmatch, 0);  
56 if(status == REG_NOMATCH)  
57     printf("No match\n");  
58 if(status == 0){  
59     for(chk = pmatch[0].rm_so;  
        chk < pmatch[0].rm_eo;  
        ++chk){  
60         temp_name[chk - pmatch[0].  
            .rm_so] = buf[chk];  
61     }  
62 }  
63 regfree(&reg);
```

遥远的代码保证了  
temp\_xxx  
x以\0结束

一眼望去有  
些迷的循环

```
64 regcomp(&reg, pattern,  
65 status = regexec(&reg,  
nmatch, pmatch, 0);  
66 if(status == REG_NOMATCH)  
67     printf("No match\n");  
68 else if(status == 0){  
69     for(chk = pmatch[0].rm_so;  
        chk < pmatch[0].rm_eo;  
        ++chk){  
70         temp_name[chk - pmatch[0].  
            .rm_so] = buf[chk];  
71 // putchar(buf[chk]);  
72     }  
73 //printf("\n");  
74 }
```

起不到足  
够作用的  
调试信息

代码克隆  
万一有个  
bug.....



# 编写容易证明的代码:

- 我们也就读出一些数据
  - 假如遥远的地方定义了 `char buf[1024] = {0};`
  - `assert(buf[nread] == '\0');` 是否成立?
- 但如果写成 `buf[nread] = '\0';` 正确性就一目了然了
- 另外还需要 `assert(nread < sizeof(buf));`
- 所以如果 `read(fd, buf, sizeof(buf) - 1);` 正确性也一目了然了

一个有点长的逻辑链条

把证明写在程序里



## 再比如.....

- 啰嗦 vs 简洁

```
18 close(READ_END(fds));  
19 dup2(WRITE_END(fds), /* -> */ STDERR_FILENO);  
20 close(WRITE_END(fds));
```

```
18 close(fds[0]);  
19 dup2(fds[1], 2);  
20 close(fds[1]);
```

- 左边读起来能更快反应过来我在做什么
- 右边：我也能.....但我要想一想.....
  - 如果有20行这样的代码，你应该想不清了
  - 你也不愿意再仔细check一下了



# 想要证明它.....你觉得呢.....?

- BST的旋转操作(来自某CSDN博客)

```
1.// node 为结点类型, 其中ch[0]表示左结点指针, ch[1]表示右结点指针
2.// pre 表示指向父亲的指针
3.void Rotate(node *x, int c) // 旋转操作, c=0 表示左旋, c=1 表示右旋
4.{
5.    node *y = x->pre;
6.    y->ch[!c] = x->ch[c];
7.    if (x->ch[c] != Null) x->ch[c]->pre = y;
8.    x->pre = y->pre;
9.    if (y->pre != Null)
10.        if (y->pre->ch[0] == y) y->pre->ch[0] = x;
11.        else y->pre->ch[1] = x;
12.    x->ch[c] = y, y->pre = x;
13.    if (y == root) root = x; // root 表示整棵树的根结点
14.}
```



# 某位同学的血泪史

- 会输出在终端里直接输入cmd arg输出的内容，忘记重定向了
- 一开始我是用指针一位一位扫buf来看有没有“(”再读取前面的内容作为名称，父进程最后一次读取的是“+++ exit with 0+++”然后匹配不上，死循环了就会segmentation fault
- 然后改成了正则表达式(掉进我挖的更大坑里了)，无法匹配，产生了bus error
  - 最后我在while语句开始的时候把buf前三位和+++匹配了一下，就行了
- 如果用execve来执行文件 在./perf find的时候会出现问题 但是用execvp就不会 emm不知道为什么

```
regcomp(&reg,  
"([a-zA-Z0-9_]+)\\((.*)\\).* = .* <([0-9]+\\. [0-9]+)>\\n$",  
REG_EXTENDED);
```



# 怎么写好程序？

- 让每一个部分都**可控**并且尽可能地**正确**
  - 相信机器永远是对的
  - 理解API的规约
  - 编写人类可读的代码
  - 完成足够的测试
- 单独保证每个部分的正确性，再去考虑全局的正确性



## 另外，当程序变大.....

- “在某个program point发生了什么” 越来越难描述
- 人类使用抽象和封装应对这个问题
  - 改为描述模块的状态、模块与外界的交互.....
  - 例如AM中的dev->id, dev->read, dev->write
  - libc中常见的功能函数
  - STL, boost, C++11/14/17新特性
  - Batteries included Python





# 当然了，凡事都有例外

- QA工程师走进酒吧：
  - 要了一杯啤酒
  - 要了0杯啤酒
  - 要了999999999杯啤酒
  - 要了一只蜥蜴
  - 要了-1杯啤酒
  - 要了一个sfdeljknesv
- 酒保从容应对，QA工程师很满意
- 接下来，一名顾客来到了同一个酒吧，问厕所在哪，酒吧顿时起了大火，然后整个建筑坍塌了



# 抽象/封装的两面

- 抽象和封装的确降低了“理解程序”的难度
  - 否则软件根本不可能被构造出来了
- 但并不意味着系统的复杂性被降低了
  - 每个模块有 $m$ 个状态， $n$ 个模块就有 $m^n$ 种组合
  - 模块之间可能产生意料之外的交互
    - 在顺序情况下，AM Device API工作得很好
    - 但如果在设备操作时发生了中断/异常，它们的行为如何？
    - 模块之间可能发生并发执行

当我的程序出bug了， 我怎么办？



# Bug到底是什么？

- Fault

- 把for (...;  $i < 10$ ; ...)写成了for (...;  $i \leq 10$ ; ...)
- 给envp传递了NULL

- Error

- 运行时，真的走到了 $i=10$ 的循环，悄悄做了一些不该做的事情  
( $M, R$ )的状态发生了不正确的改变，但没有人知道发生了什么

- Failure ← 我们在这里

- 发生了可观测的问题，crash，错误的结果，strace说找不到ls



# 调试的基本原则

- 我们已经尽力让程序正确了.....但.....
- 反正机器永远是对的
  - 善用手边的工具
  - 弄清楚发生了什么(error)
  - 找到fault



# 用对的工具

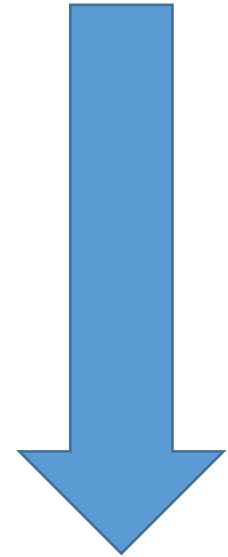
- 你也许已经吃过苦头，也许已经听别人说了解决的办法，但是如果你是给老板打工，孤立无援呢.....
  - 你有没有想过，还可以strace -f trace ls?
  - 原来strace里到各种地方去找了ls啊
    - 上哪里找的呢..... 这肯定跟“找不到ls”有关系啊
    - 好像上课讲过PATH，讲过环境变量
    - 那环境变量是怎么给子进程的呢.....?

```
stat("/usr/local/sbin/ls", 0x7fff0767f570) = -1
stat("/usr/local/bin/ls", 0x7fff0767f570) = -1
stat("/usr/sbin/ls", 0x7fff0767f570)      = -1
stat("/usr/bin/ls", 0x7fff0767f570)      = -1
stat("/sbin/ls", 0x7fff0767f570)         = -1
stat("/bin/ls", {st_mode=S_IFREG|0755, ...}) = 0
```



# 你们手中有的工具(按代价从小到大)

- valgrind – 完全不花钱，但只能检查特定类型错误
- 脑子是个好东西 – 花大价钱，终身享受
- assert – 花一次钱，终身享受
- printf – 花一次钱，错了要仔细看
- strace – 花一次钱，错了要仔细看
- gdb – 花大价钱，但什么都搞不定的时候有奇效



- 思考题：它们在fault → error → failure的链条上都做了什么？