第2讲 应用视角的操作系统：程序的状态机模型、编译优化

1. 引入

指令序列和高级语言的状态机模型；回答以下问题：

- 什么是软件（程序）？
- 如何在操作系统上构造最小/一般/图形界面应用程序？
- 什么是编译器？编译器把一段程序翻译成什么样的指令序列才算“正确”？

2. 汇编代码和最小可执行文件

什么是操作系统上的程序？——写一个最小的操作系统上的程序。

构造最小的 Hello, World “应用程序”

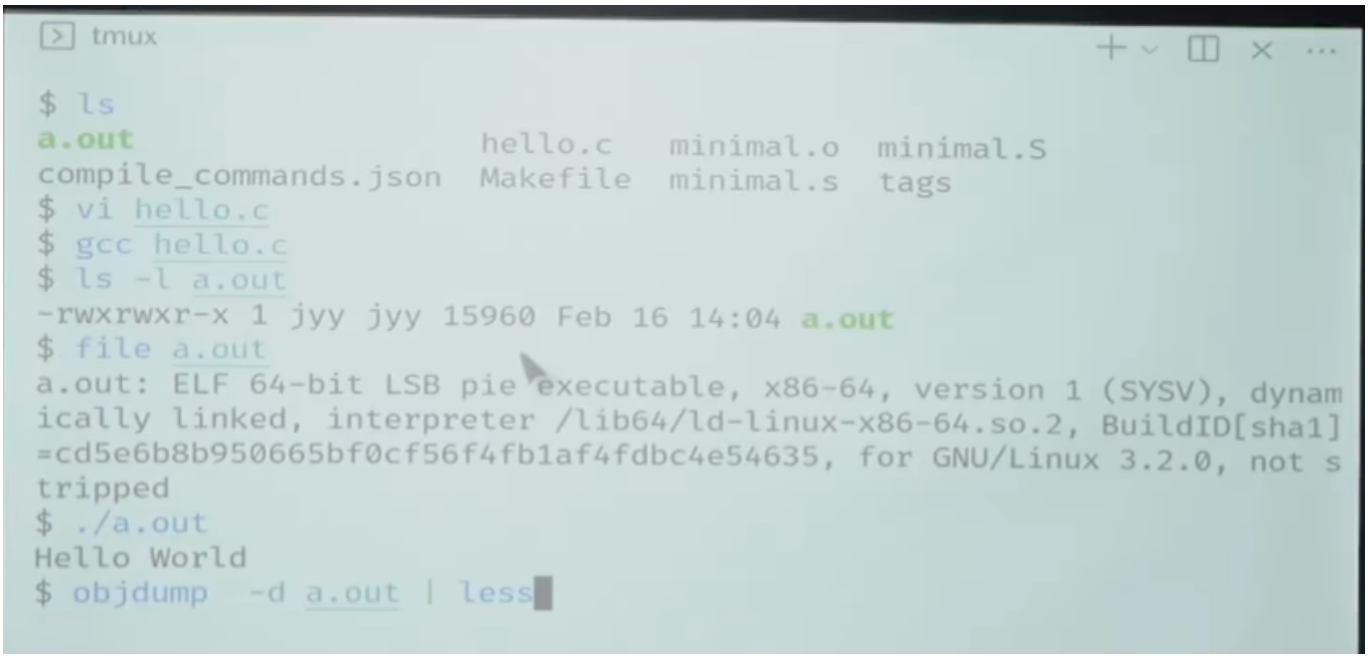
```c
int main() {
  printf("Hello, World\n");
}
```

gcc 编译出来的文件一点也不小

- objdump 工具可以查看对应的汇编代码
- `--verbose` 可以查看所有编译选项 (真不少)
  - printf 变成了 puts@plt
- `-Wl,--verbose` 可以查看所有链接选项 (真不少)
  - 原来链接了那么多东西
  - 还解释了 end 符号的由来
- `-static` 会链接 libc (大量的代码)

`a.out`：assembler的输出

ELF：可执行可链接的（ICS返场）



1. `.c` 经历预编译，变为 `.i`
2. `.i`，编译器编译，变为 `.s`（汇编代码）
3. 然后汇编器，生成 `.o`
4. 然后是链接器，生成 `a.out`

> 所以在计算机程序里面，没有什么魔法，所有一切都是程序一点点生成出来的。

强行构造最小的 Hello, World?

我们可以手动链接编译的文件，直接指定二进制文件的入口

- 直接用 ld 链接失败
  - ld 不知道怎么链接 printf
- 不调用 printf 可以链接
  - 但得到奇怪的警告 (可以定义成 `_start` 避免警告)
  - 而且 Segmentation Fault 了
- `while (1);` 可以链接并正确运行

问题：为什么会 Segmentation Fault?

- 当然是观察程序的执行了
  - 初学者必须克服的恐惧：STFW/RTFM (M 非常有用)
  - `starti` 可以帮助我们从第一条指令开始执行程序

为什么while就不会爆栈?

- 猜测：ret的时候出了问题？——调试器！gdb

> 任何计算机的问题，尤其是计算机系统和计算机编程，只要你们能问出正确的问题，ai和搜索引擎都能帮你们找到；所以你们和专家的问题就是——问出正确的问题。

计算机系统实际上是一个状态机。内存的单元、寄存器……

所以，带来的推论是，如果我们的指令都是 `mov` 、 `add` 、 `push` 这种 计算类型的指令，cpu这样的“无情的执行指令的机器”根本不可能停的下来（fetch+decode+execute），最终的结果要么是死循环，要么是crash了。所以我们要提供一条指令能让它停下来。这条指令就是 `syscall` **系统调用**。在系统调用发生的时候，程序会将自己完全交给操作系统。

解决异常退出

有办法让程序“停下来”吗？

- 纯“计算”的状态机：不行
- 没有“停机”的指令

解决办法：用一条特殊的指令请操作系统帮忙

```
movq $SYS_exit,  %rax   # exit(
movq $1,         %rdi   #   status=1
syscall                 # );
```

- 把“系统调用”的参数放到寄存器中
- 执行 syscall，操作系统接管程序
  - 程序把控制权完全交给操作系统
  - 操作系统可以改变程序状态甚至终止程序

有点像做全麻手术。我们就像是操作系统上的进程/程序，全麻以后你就啥都没了，几个小时以后你醒过来看到医生在你面前招手；中间这几个小时就像消失了一样。

在中间这段空白的时间里你完全不知道操作系统对你做了什么，甚至你这个进程从世界上消失了。

汇编代码的状态机模型

Everything is a state machine: 计算机 = 数字电路 = 状态机

- 状态 = 内存 M + 寄存器 R
- 初始状态 = ABI 规定 (例如有一个合法的 %rsp)
- 状态迁移 = 执行一条指令
  - 我们花了一整个《计算机系统基础》解释这件事
  - gdb 同样可以观察状态和执行

操作系统上的程序

- 所有的指令都只能计算
  - deterministic: mov, add, sub, call, ...
  - non-deterministic: rdrand, ...
  - syscall 把 (M, R) 完全交给操作系统

总结：

- 我们用一种底层low level的角度来理解完成了程序——状态机。

3. 理解高级语言程序：C程序的形式语义

你能写一个 C 语言代码的“解释器”吗？

- 如果能，你就完全理解了高级语言
- 和“电路模拟器”、“RISC-V 模拟器”类似
  - 实现 gdb 里的“单步执行”

```
while (1) {
  stmt = fetch_statement();
  execute(stmt);
}
```

“解释器”的例子：用基础结构模拟函数调用和递归

- 试试汉诺塔吧

递归版本的汉诺塔是程序设计中的经典例题——同学们也曾经在理解这个程序的时候遇到困难。这种根本性的困难在于，大家可能并没有建立 “函数调用”、“函数返回” 和 “单步执行” 的正确模型。如果我们清楚地认识到所谓单步执行，指的是从顶部的栈帧 PC 取一条指令执行，就不难用栈模拟递归程序。

```
1  typedef struct {
2   int pc, n;
3   char from, to, via;
4  } Frame;
5  #define call(...) ({ *(++top) = (Frame) { .pc = 0, __VA_ARGS__ }; })
6  #define ret()     ({ top--; })
7  #define goto(loc) ({ f->pc = (loc) - 1; })
8  void hanoi(int n, char from, char to, char via) {
9    Frame stk[64], *top = stk - 1;
10   call(n, from, to, via);
11   for (Frame *f; (f = top) >= stk; f->pc++) {
12     n = f->n; from = f->from; to = f->to; via = f->via;
13     switch (f->pc) {
14       case 0: if (n == 1) { printf("%c -> %c\n", from, to); goto(4); } break;
15       case 1: call(n - 1, from, via, to);   break;
16       case 2: call(    1, from, to,  via);  break;
17       case 3: call(n - 1, via,  to,  from); break;
18       case 4: ret();                        break;
19       default: assert(0);
20     }
21   }
```

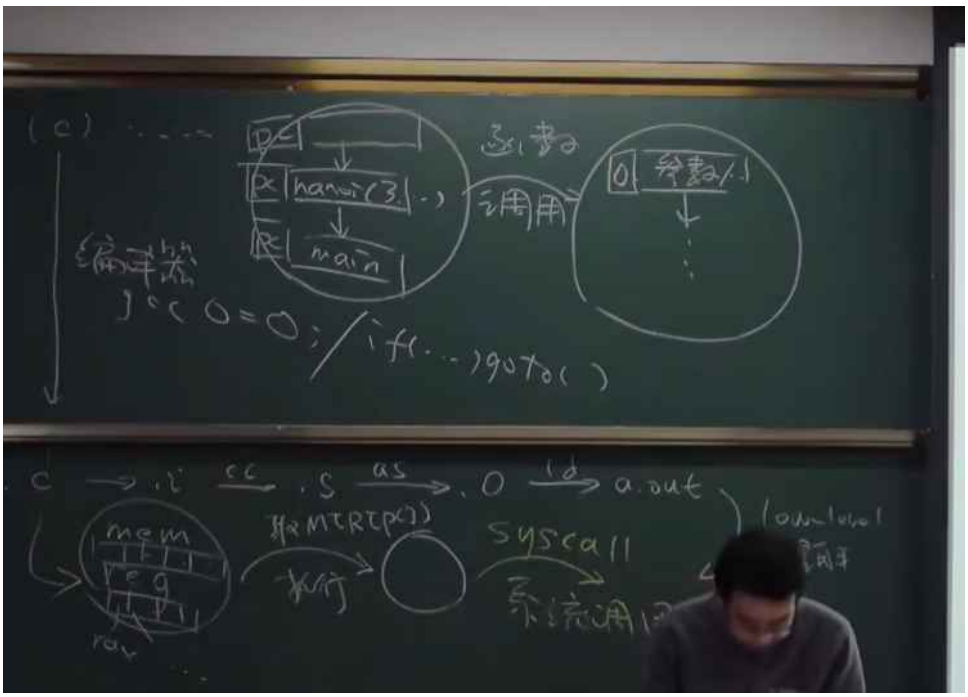
```
22 }
23
```

这个程序的秘诀就藏在三个宏里面。不仅汇编语言的代码是状态机，高级语言的程序也是一个状态机。所以当我们讨论C语言状态机的状态是什么的时候，我们发现，你写不出非递归的汉诺塔，是因为你不知道什么是函数调用：你知道手工模拟一个栈，但是你不知道栈里面有什么。

实际上，程序是由很多栈帧组成的。与此同时，每一个栈帧都有一个PC。所以：

1. 什么是函数调用？在栈帧的顶上再加一个stack，pc=0，把参数扔上去。
2. 什么是函数的返回？把顶上的stack抹掉。
3. 什么是执行一条语句？取top most的pc上的语句执行。

这就是C的形式语义。有这样的概念模型，我们就可以把任何C语言代码翻译成为更简单的C语言代码，甚至没有循环。你给我一个C语言代码，我可以将其改写为一个行为和它等价、但是只有顺序语句执行和if+goto的东西。你就实现了一个编译器！



对于任意的 C 语言代码，我们都可以把它解析成语法树的结构 (类似于表达式树，在《计算机系统基础》的 Programming Assignment 中包含了类似的实验)。C 程序的语义解释执行 “一条语句” 的更严谨说法是解释执行当前语句中 “优先级最高的节点”。

简单 C 程序的状态机模型 (语义)

状态

- Stack frame 的列表 + 全局变量

初始状态

- 仅有一个 frame: `main(argc, argv)`；全局变量为初始值

状态迁移

- 执行 frames.top.PC 处的简单语句
- 函数调用 = push frame (frame.PC = 入口)
- 函数返回 = pop frame

然后看看我们的非递归汉诺塔 (更本质)

4. 理解编译器

理解编译器

我们有两种状态机

- 高级语言代码 $.c$
  - 状态：栈、全局变量；状态迁移：语句执行
- 汇编指令序列 $.s$
  - 状态：(M, R)；状态迁移：指令执行
- 编译器是二者之间的桥梁：

$$.s = \text{compile}(.c)$$

那到底什么是编译器？

- 不同的优化级别产生不同的指令序列
- 凭什么说一个 $.s = \text{compile}(.c)$ 是“对的”还是“错的”？

$.s = \text{compile}(.c)$：编译正确性

$.c$ 执行中所有外部观测者可见的行为，必须在 $.s$ 中保持一致

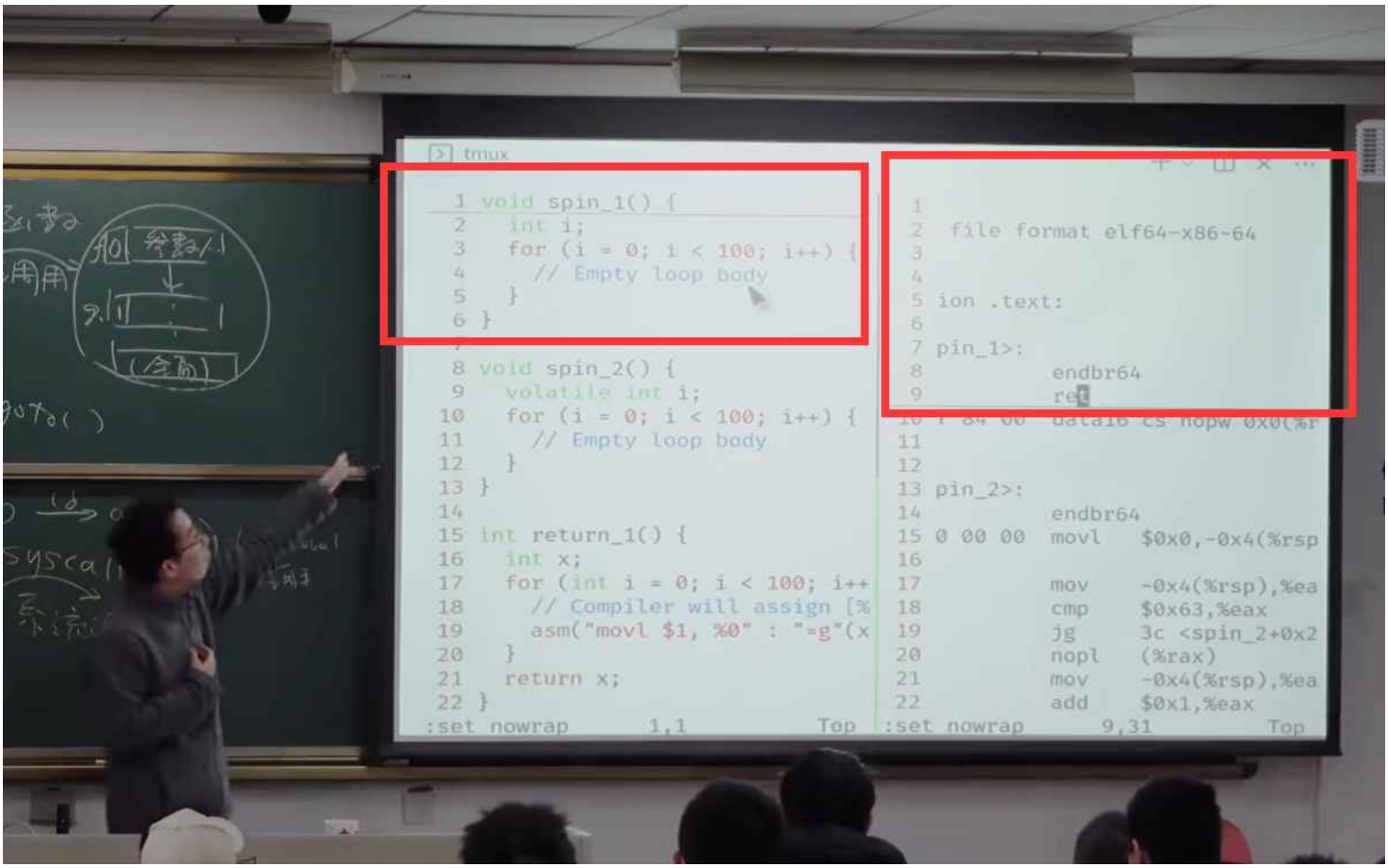
- External function calls (链接时确定)
  - 如何调用由 Application Binary Interface (ABI) 规定
  - 可能包含系统调用，因此不可更改、不可交换
- 编译器提供的“不可优化”标注
  - `volatile` [load | store | inline assembly]
- Termination
  - $.c$ 终止当且仅当 $.s$ 终止

在此前提下，任何翻译都是合法的 (例如我们期望更快或更短的代码)

- 编译优化的实际实现：(context-sensitive) rewriting rules
- 代码示例：观测编译器优化行为和 compiler barrier

比如，`printf` 打印一个东西，就是外部观测者可见的，这玩意是无法优化的，改了就错了。这里写了几种不能“改”的东西。

所以我们看红色的代码直接被编译器优化成了个return：



因为 `i` 这个局部变量并不会被任何人看到！随着函数返回的时候就消失了！并没有执行任何外部可见的操作。

但如果加上了 `volatile`，我们会发现右边就去栈上开始对 `x` 进行读写了。

5. 操作系统上的软件（应用程序）

示例：strace | hello.c | Makefile | minimal.S

```
#include

.globl _start
_start:
  movq $SYS_write, %rax   // write(
  movq $1,         %rdi   //   fd=1,
  movq $st,        %rsi   //   buf=st,
  movq $(ed - st), %rdx   //   count=ed-st
  syscall                 // );

  movq $SYS_exit,  %rax   // exit(
  movq $1,         %rdi   //   status=1
  syscall                 // );

st:
  .ascii "\033[01;31mHello, OS World\033[0m\n"
ed:
```

操作系统中的任何程序

> 任何程序 = minimal.S = 调用 syscall 的状态机

可执行文件是操作系统中的对象

- 与大家日常使用的文件 (a.c, README.txt) 没有本质区别
- 操作系统提供 API 打开、读取、改写 (都需要相应的权限)

查看可执行文件

- vim, cat, xxd 都可以直接“查看”可执行文件
  - vim 中二进制的部分无法“阅读”，但可以看到字符串常量
  - 使用 xxd 可以看到文件以 "\x7f" "ELF" 开头
  - Vscode 有 binary editor 插件

`a.out` 和 `README.md` 没有任何本质区别。

这节课全部想说的，就是：你在操作系统里面看到的所有的一切，比如vim，vsc，浏览器，其实都是和我们一开始看到的只有三行汇编的 `minimal.S` 是一样的！

我们可以把应用程序打开，我们的应用程序会在运行的过程中不断地向操作系统请求系统调用 `syscall`，让os给它一些反馈。我们先不管应用里面的指令是如何执行了，我们只关注它和os的交互，我们也能勾勒出应用程序运行的整个的过程。

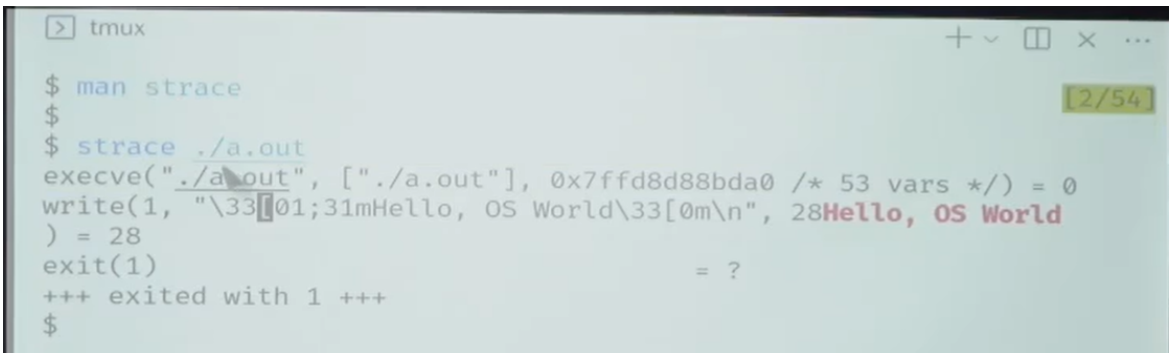
打开程序的执行：Trace (追踪)

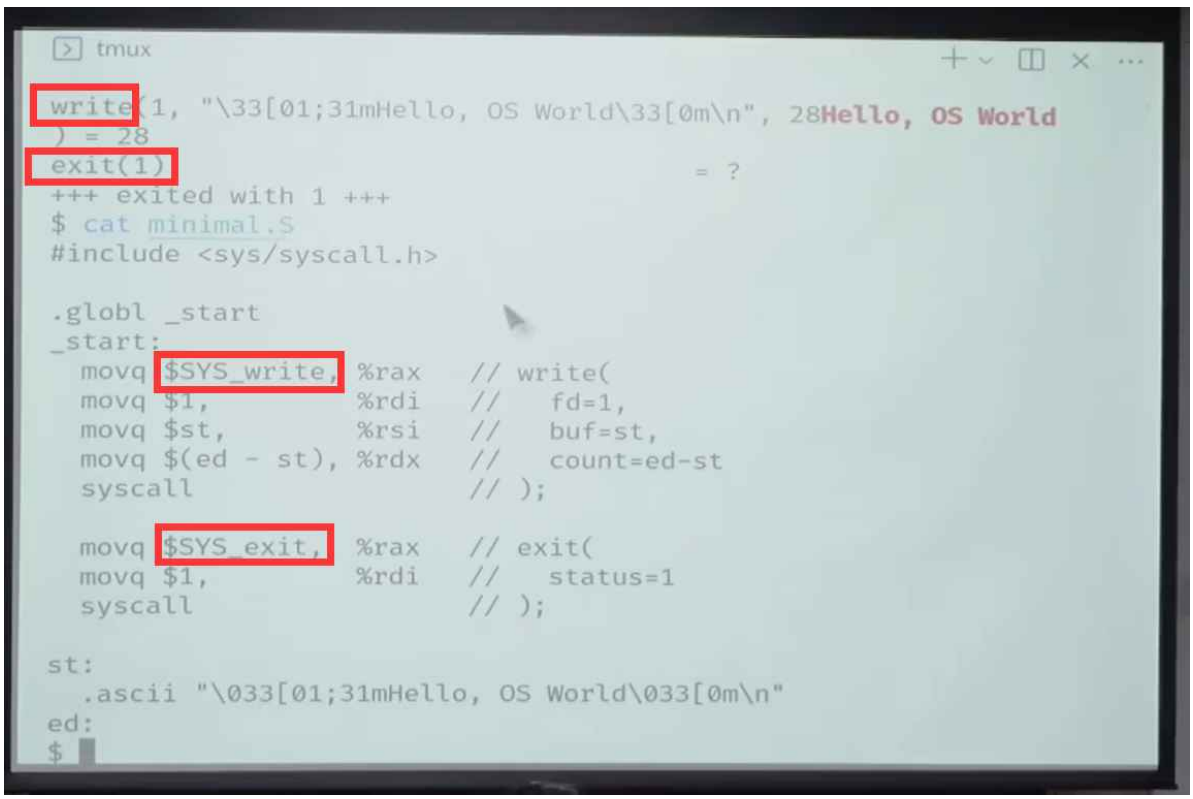
> In general, trace refers to the process of following *anything* from the beginning to the end. For example, the `traceroute` command follows each of the network hops as your computer connects to another computer.

这门课中很重要的工具：strace

- System call trace
- 允许我们观测状态机的执行过程
  - Demo: 试一试最小的 Hello World
  - 在这门课中，你能理解 strace 的输出并在你自己的操作系统里实现相当一部分系统调用 (mmap, execve, ...)

这里就引出了很重要的工具：strace(system call trace)。





操作系统中“任何程序”的一生

> 任何程序 = minimal.S = 调用 syscall 的状态机

- 被操作系统加载
  - 通过另一个进程执行 execve 设置为初始状态
- 状态机执行
  - 进程管理：fork, execve, exit, ...
  - 文件/设备管理：open, close, read, write, ...
  - 存储管理：mmap, brk, ...
- 调用 _exit (exit_group) 退出

(初学者对这一点会感到有一点惊讶)

- 说好的浏览器、游戏、杀毒软件、病毒呢？都是这些 API 吗？
- 我们有 strace，就可以自己做实验了！

各式各样的应用程序

都在操作系统 API (syscall) 和操作系统中的对象上构建

- 窗口管理器
  - 管理设备和屏幕 (read/write/mmap)
  - 进程间通信 (send, recv)
- 任务管理器
  - 访问操作系统提供的进程对象 (readdir/read)
  - 参考 gdb 里的 `info proc *`
- 杀毒软件
  - 文件静态扫描 (read)
  - 主动防御 (ptrace)
  - 其他更复杂的安全机制……

6. 总结

Take-away Messages

无论是汇编代码还是高级语言程序，它们都可以表示成状态机：

- 高级语言代码 .c
  - 状态：栈、全局变量；状态迁移：语句执行
- 汇编指令序列 .s
  - 状态：(M, R)；状态迁移：指令执行
- 编译器实现了两种状态机之间的翻译

应用程序与操作系统沟通的唯一桥梁是系统调用指令 (例如 x86-64 的 syscall)。计算机系统不存在玄学；一切都建立在确定的机制上

- 理解操作系统的重要工具：gcc, binutils, gdb, strace