

Hello, OS World

Hello, OS World

Operating System: A body of software, in fact, that is responsible for *making it easy to run programs* (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. (OSTEP)

要想理解“操作系统”，就要理解什么是“程序”

- 那就从 Hello World 开始吧

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

意识到的问题

这个 Hello World 似乎太复杂了

- 一点也不“最小”

相关的工具

- 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)

试图构造最小的 Hello, World

试着手动链接编译的文件

- 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)
 - 但是 Segmentation Fault 了
- 如果改成 `while (1);` 可以正确运行
 - 说明我们写的代码的确被执行了

为什么？怎么办？

我在操作系统课的练习题中写了一个代码，发生了 Segmentation Fault，我阅读了代码，无法定位到问题的地方。程序也没有任何输出。我应该怎么办？

学习任何别人学过的东西，别人都遇到过你遇到过的困难

- 你只需要提出问题
- 随着熟练度的增长，你就可以 RTFM 了
 - 此时 M 变得 friendly
 - 在你问不出问题时，[Manual](#) 的价值就体现了

人类的新时代即将来临

- (你问不出问题的时候，[langchain](#) 可以帮你枚举)

什么是程序？

答案在这里：

```
struct CPUState {  
    uint32_t regs[32], csrs[CSR_COUNT];  
    uint8_t *mem;  
    uint32_t mem_offset, mem_size;  
};
```

处理器：无情的、执行指令的**状态机**

- 从 $M[PC]$ 取出一条指令
- 执行它
- 循环往复

解决异常退出

程序自己是不能“停下来”的 是没有任何一个指令来做这个事情的，而是通过硬件

- 指令集里没有一条关闭计算机的指令，那么操作系统是如何在关闭所有软件后，切断计算机的电源的？

只能借助操作系统

```
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
 - 操作系统可以任意改变程序状态 (甚至终止程序)

Everything (二进制程序) = 状态机

状态

- gdb 内可见的内存和寄存器

初始状态

- 由 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 可以单步观察状态机的执行
- **syscall 指令**: 将状态机 “完全交给” 操作系统

“拆解” 应用程序

操作系统上的应用程序

Core Utilities (coreutils)

- *Standard* programs for text and file manipulation
- 系统中默认安装的是 [GNU Coreutils](#)

系统/工具程序

- bash, [binutils](#), apt, ip, ssh, vim, tmux, gcc, python, ffmpeg, ...
 - 原理不复杂 (例如 apt 是 dpkg 的套壳), 但琐碎
- All-in-one 工具合集: [busybox](#), [toybox](#)

其他各种应用程序

- Vscode、OBS-Studio、浏览器、音乐播放器: 它们在各种工具程序基础上建立起来 (例: ffmpeg)

所以这些程序.....

和 **minimal.S** 有任何区别吗？

- 简短的答案：**没有**
- 任何程序 = minimal.S = 状态机

可执行文件是操作系统中的对象

- 与 minimal 的二进制文件没有本质区别
- 我们甚至可以像文本一样直接编辑可执行文件

为了帮大家记住这一点

死去的记忆复活了



存档用 0x00, 0x01, 0x02 表示状态，嘿嘿嘿.....

Demo: 修改 Hello World

死去记忆的意义

如果那时候有人告诉我.....

- 同样的方式也可以去 hack Windows binary
- Binary 太大？用**正确的工具**把“不在意”的部分屏蔽掉
 - 调试程序，在运行时观察哪里变了
 - 就入门了逆向工程
- (其实大家离“做点不一样的东西”并不远)

对《操作系统》课的反思

- **做减法**：把“不重要”的部分屏蔽掉
- “简单”也可以深刻
- 推广到学习：如果觉得“难”，应该有简化的方法

正确的工具

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.

System call trace (strace)

- “理解程序是如何与操作系统交互的”
 - (观测状态机执行里的 syscalls)
 - Demo: 试一试最小的 Hello World

```
root@hcss-ecs-00dc:~/minimal# strace ./minimal
execve("./minimal", ["/minimal"], 0x7fff3b5a4e80 /* 27 vars */) = 0
write(1, "\33[01;31mHello, OS World\33[0m\n", 28Hello, OS World
) = 28
exit(1)                                = ?
+++ exited with 1 +++
```

操作系统中的“任何程序”

任何程序 = minimal.S = 状态机

- 总是从被操作系统加载开始
 - 通过另一个进程执行 `execve` 设置为初始状态
- 经历状态机执行 (计算 + syscalls)
 - 进程管理: `fork`, `execve`, `exit`, ...
 - 文件/设备管理: `open`, `close`, `read`, `write`, ...
 - 存储管理: `mmap`, `brk`, ...
- 最终调用 `_exit` (`exit_group`) 退出

感到吃惊?

- 浏览器、游戏、杀毒软件、病毒呢? 都是这些 API 吗?

自己动手做实验：观察程序的执行

`strace -f gcc a.c & | vim -`

工具程序代表：编译器 (gcc)

- `strace -f gcc a.c` (gcc 会启动其他进程)
 - 可以管道给编辑器 `vim -`
 - 编辑器里还可以 `%!grep`
 - 对于开发者来说，工具的组合是非常重要的

图形界面程序代表：编辑器 (xedit)

- `strace xedit`
 - 图形界面程序和 X-Window 服务器按照 X11 协议通信
 - 虚拟机中的 xedit 将 X11 命令通过 ssh (X11 forwarding) 转发到 Host

想象“一切应用程序”的实现

应用程序 = 计算 + 操作系统 API

- 窗口管理器
 - 能直接管理屏幕设备 (read/write/mmap)
 - 能画一个点，理论上就能画任何东西
 - 能够和其他进程通信 (send, recv)
- 任务管理器
 - 能访问操作系统提供的进程对象 (M1 - pstree)
- 杀毒软件
 - 文件静态扫描 (read)、主动防御 (ptrace)

操作系统的职责：提供令应用程序舒适的抽象 (对象 + API)

理解高级语言程序

灵魂拷问

既然说“任何程序”都和 **minimal.S** 是一样的

- 为什么我们没有在 C 代码里看到系统调用？
- C 代码是如何变成二进制文件的？
- 到底编译器什么优化能做、什么优化不能做？

Quick Quiz

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 logisim/mini-rv32ima 完全一样

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

递归 v.s. 非递归

实现递归到非递归的转换



这个问题难到 GPT-4 都翻车了

```
int f(int n) { return (n<=1) ? 1 : f(n-1) + g(n-2); }  
int g(int n) { return (n<=1) ? 1 : f(n+1) + g(n-1); }
```

让我们先做一点简化

试图把 C 代码改写成 “SimpleC”

- 成每条语句至多一次运算 (函数调用也是运算)
- 条件语句中不包含运算
 - 真的有这种工具 (C Intermediate Language) 和解释器
- (暂时假设没有指针和内存分配)

Everything (C 程序) = 状态机

- 状态 = 变量数值 + 栈
- 初始状态 = main 的第一条语句
- 状态迁移 = 执行一条语句中的一小步
 - 为什么你觉得还写不出汉诺塔？

真正的“理解”

“状态机”是拥有**严格数学定义**的对象。这意味着你可以把定义写出来，并且用**数学严格**的方法理解它 —— 形式化方法

状态

- [StackFrame, StackFrame, ...] + 全局变量

初始状态

- 仅有一个 StackFrame(main, argc, argv, **PC=0**)
- 全局变量全部为初始值

状态迁移

- 执行 frames[-1].PC 处的简单语句

重新理解编译器

什么是编译器？

编译器的输入

- 高级语言 (C) 代码 = 状态机

编译器的输出

- 汇编代码 (指令序列) = 状态机

编译器 = 状态机之间的翻译器

SimpleC: 直接翻译

运算

- 把操作数 load 到寄存器、执行运算、store 写回结果

分支/循环

- 使用条件跳转分别执行代码

函数调用

- 专门留一个寄存器给栈 (SP, Stack Pointer)
- 将 Stack frame 的信息保存在内存里
 - 通过 SP 可以访问当前栈帧的变量、返回地址等

(SimpleC 编译器很适合作为《计算机系统基础》的编程练习)

SimpleC: 直接翻译 (cont'd)

所以，C 被称为高级汇编语言

- 存在 C 代码到指令集的直接对应关系
 - 状态机和迁移都可以“直译”
 - 于是计算机系统里多了一个抽象层 (“一生二、二生三、三生万物”)
- 更“高级”的语言就很难了
 - C++ `virtual void foo();`
 - Python `[1, 2, 3, *rest]`
 - Javascript `await fetch(...)`

C 语言能实现对机器更好的控制 (例子: **Inline Assembly**)

编译优化

C 语言编译器在进行代码优化时，遵循的基本准则是在不改变程序的语义 (即程序的行为和输出结果) 的前提下，提高程序的执行效率和/或减少程序的资源消耗

```
int foo(int x) {  
    int y = x + 1;  
    return y - 1;  
}
```

一些“不改变语义”的例子 (编译优化中最重要的“三板斧”):

- **函数内联**: 将函数调用替换为函数体本身的内容
- **常量传播**: 在编译时计算常量表达式的值并替换
- **死代码消除**: 删除永远不会被执行到的代码

但如果我们问一个更本质的问题.....

给两个程序 A, B ，编译器到底允许不允许把 A 编译成 B ？

- 刚才的优化怎么就“不改变语义”了？
- 什么才算“改变语义”？

考虑一个特殊情况

- 一个程序没有任何系统调用
 - 它甚至不能终止
 - `main` 函数直接返回是 undefined behavior
 - 无论它有多少代码
 - 优化成 `while (1)` 就好了

编译正确性

系统调用是使程序计算结果**可见**的唯一方法

- 不改变语义 = 不改变可见结果
- 状态机的视角：满足**C/汇编状态机生成的所有 syscall 序列完全一致**，任何优化都是允许的

C 代码中的**不可优化部分**

- External function calls (链接时才能确定到底是什么代码)
 - 未知的代码可能包含**系统调用**
 - 因此不可删除、移出循环等，且要保证参数传递完全一致
- 编译器提供的“不可优化”标注
 - `volatile` [load | store | inline assembly]

有没有觉得这个定义保守了? 🌶️🌶️🌶️

凭什么系统调用不能被优化?

```
if (n <= 26) {  
    for (int i = 0; i < n; i++) {  
        putchar('A' + i);  
    }  
}
```

- 凭什么不能合并成一个 `printf`?

把状态机的一部分直接放到操作系统里运行

- 把代码放进操作系统运行: [XRP](#)
- 单个应用就是操作系统: [Unikernel](#)