

背景回顾：操作系统有三条主线：“软件 (应用)”、“硬件 (计算机)”、“操作系统 (软件直接访问硬件带来麻烦太多而引入的中间件)”。想要理解操作系统，对操作系统的服务对象 (应用程序) 有精确的理解是必不可少的。

本讲内容：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”？

## 汇编代码和最小可执行文件

# 构造最小的 Hello, World “应用程序”

---

```
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 `objdump -d a.out | less -R`
- `--verbose` 可以查看所有编译选项 (真不少) `gcc hello.c -static --verbose`
  - `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)

`gcc hello.c -static`

```
airmac@debian-x64:2-minimal$ ld hello.o  
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000  
ld: hello.o: in function `main':  
hello.c:(.text+0xf): undefined reference to `puts'
```

# 强行构造最小的 Hello, World?

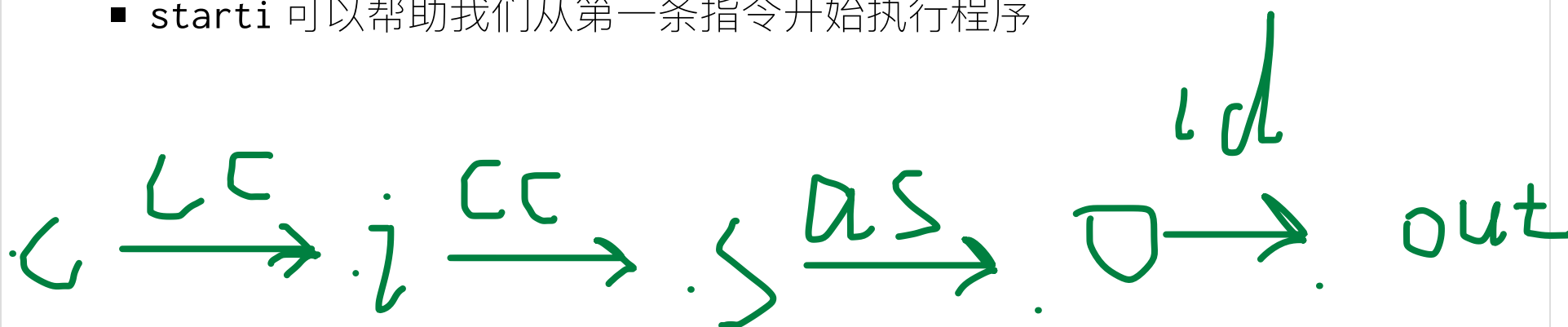
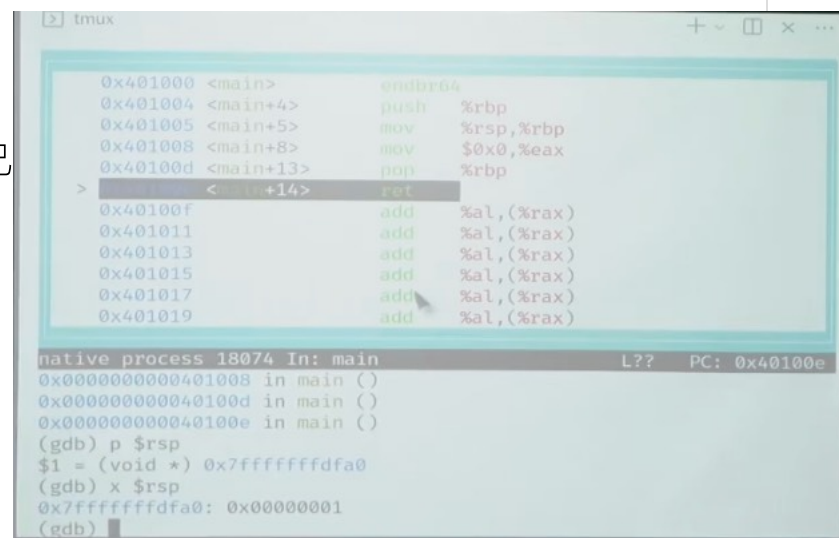
我们可以手动链接编译的文件，直接指定二进制文件的入口

- 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)
  - 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` 可以帮助我们第一条指令开始执行程序

layout src



## 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

# 对一些细节的补充解释

为什么用 gcc 编译?

- gcc 会进行预编译 (可以使用 `__ASSEMBLER__` 宏区分汇编/C 代码)

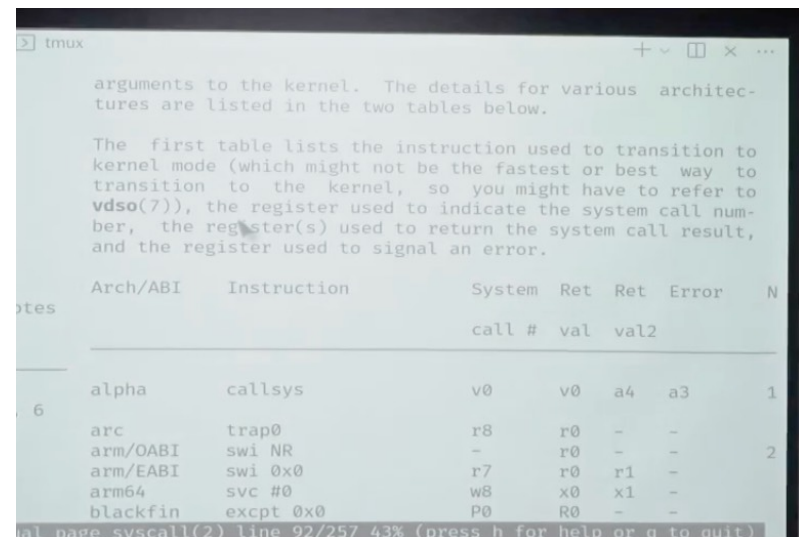
ANSI Escape Code 的更多应用

- vi.c from busybox
- `dialog --msgbox 'Hello, OS World!' 8 32`
- `ssh -o 'HostKeyAlgorithms +ssh-rsa' sshtron.zachlatta.com`

更重要的问题: 怎样才能变强?

- 问正确的问题、用正确的方式找答案
  - `syscall (2), syscalls (2)` -- **RTFM & RTFSC**
  - Q & A 论坛; Q & A 机器人

**man 2 syscall**



```
arguments to the kernel. The details for various architectures
are listed in the two tables below.

The first table lists the instruction used to transition to
kernel mode (which might not be the fastest or best way to
transition to the kernel, so you might have to refer to
vdso(7)), the register used to indicate the system call number,
the register(s) used to return the system call result,
and the register used to signal an error.

Arch/ABI      Instruction      System  Ret  Ret  Error  N
               call #   val  val2
-----
6  alpha      callsys         v0      v0   a4   a3      1
   arc        trap0           r8      r0   -    -      2
   arm/OABI    swi NR          -      r0   -    -      2
   arm/EABI    swi 0x0         r7      r0   r1   -      2
   arm64       svc #0          w8      x0   x1   -      2
   blackfin    excpt 0x0       P0      R0   -    -      2

al page syscall(2) line 92/257 43% (press h for help or q to quit)
```

# 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: `rand`, ...
  - `syscall` 把  $(M, R)$  完全交给操作系统

# 理解高级语言程序

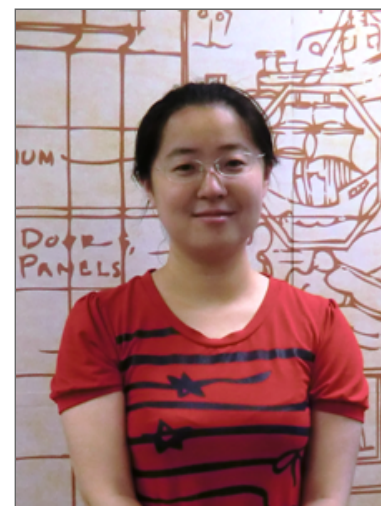
# 什么是程序？

---

Hmm....

你需要 《程序设计语言的形式语义》

- by 梁红瑾 🎩
- $\lambda$ -calculus, operational semantics, Hoare logic, separation logic
- 入围“你在南京大学上过最牛的课是什么？”知乎高票答案
  - ~~当然，我也厚颜无耻地入围了~~





## ~~不，你不需要~~

---

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

编译器，就是把这种递归的汉诺塔，把他翻译成更简单的函数调用，没有递归，没有循环，改写成跟行文等价，至于if...goto...的逻辑

## 这个问题已经超出了 90% 程序员的能力范围

---

ChatGPT 竟然改写对了！而且给出了非常优雅 (但也有缺陷) 的实现

```
void hanoi_non_recursive(int n, char from, char to, char v
    struct Element { int n; char from; char to; char via; };
    std::stack<Element> elements;
    elements.push({n, from, to, via});
    while (!elements.empty()) {
        auto e = elements.top();
        elements.pop();
        if (e.n == 1) {
            printf("%c -> %c\n", e.from, e.to);
        } else {
            elements.push({e.n - 1, e.via, e.to, e.from});
            elements.push({1, e.from, e.to, e.via});
            elements.push({e.n - 1, e.from, e.via, e.to});
        }
    }
```

}

}



## 当然，ChatGPT 也没能完全理解

---

给了他一个更难的题，果然翻车了 (思路基本正确，但不再优雅)

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

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

(你们会写这个的非递归吗)

# 简单 C 程序的状态机模型 (语义)

---

对 C 程序做出简化

- 简化：改写成每条语句至多一次运算/函数调用的形式
  - 真的有这种工具 (C Intermediate Language) 和解释器

状态机定义

- 状态 = 堆 + 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

(这还只是“粗浅”的理解)

- **Talk is cheap. Show me the code.** (Linus Torvalds)
  - 任何真正的理解都应该落到可以执行的代码

# 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)

# 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 = 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

**操作系统上的软件(应用程序)**

# 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 `vim` 中输入 `:%!xxd`
  - `Vscode` 有 `binary editor` 插件

# 系统中常见的应用程序

---

## Core Utilities (coreutils)

- *Standard* programs for text and file manipulation
- 系统中安装的是 **GNU Coreutils**
  - 有较小的替代品 **busybox**

## 系统/工具程序

- bash, **binutils**, apt, ip, ssh, vim, tmux, jdk, python, ...
  - 这些工具的原理不复杂 (例如 apt 是 dpkg 的套壳), 但琐碎
  - **Ubuntu Packages** (和 apt-file 工具) 支持文件名检索

## 其他各种应用程序

- Vscode, 浏览器、音乐播放器.....

## 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, ...)

# 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！

# 动手实验：观察程序的执行

---

工具程序代表：编译器 (gcc)

- 主要的系统调用： `execve`, `read`, `write`
- `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` (细节/技巧)

图形界面程序代表：编辑器 (xedit)

- 主要的系统调用： `poll`, `recvmsg`, `writev`
- `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

# 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.....