各种视角下的操作系统

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

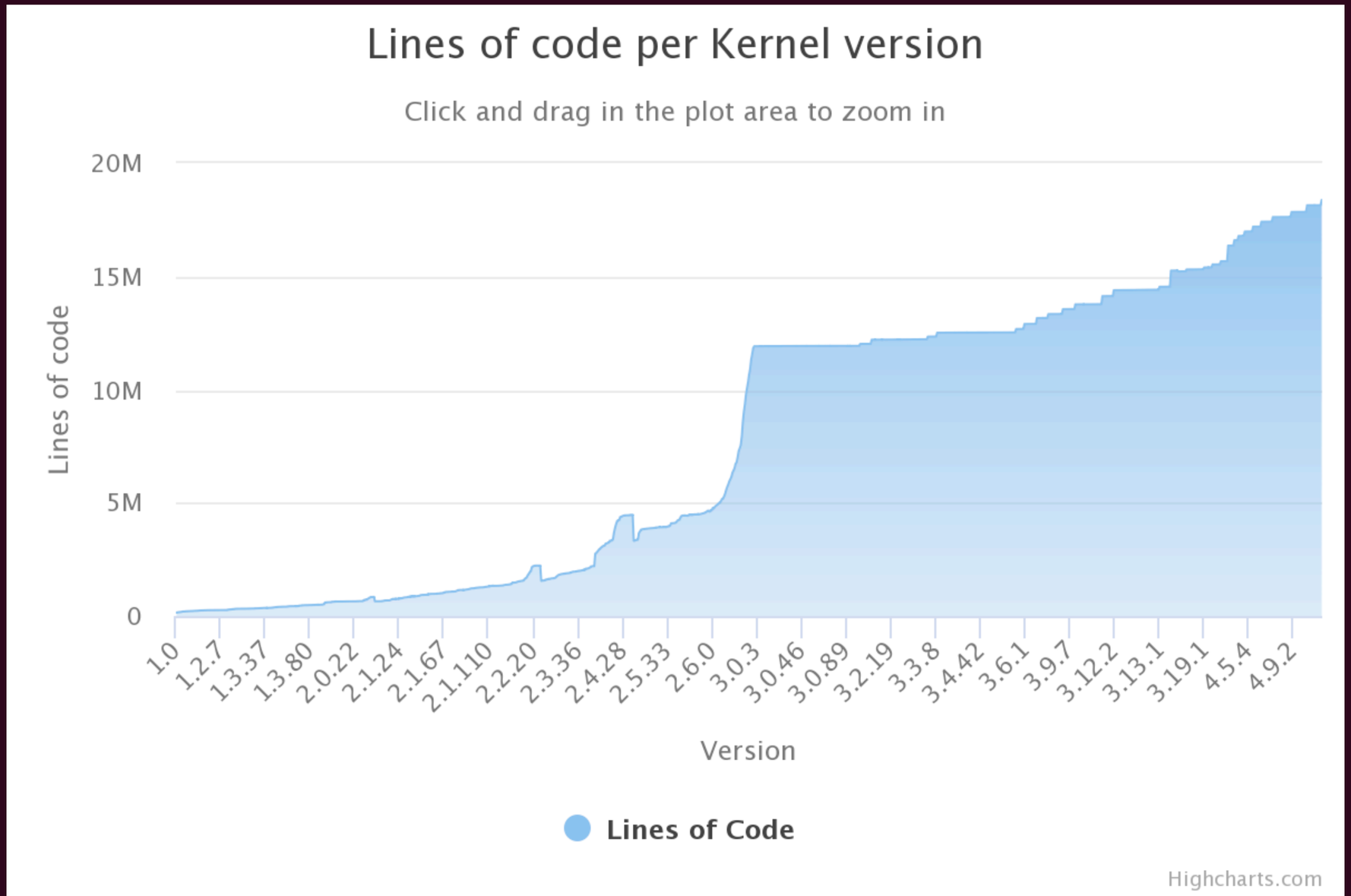
大纲

应用视角下的操作系统

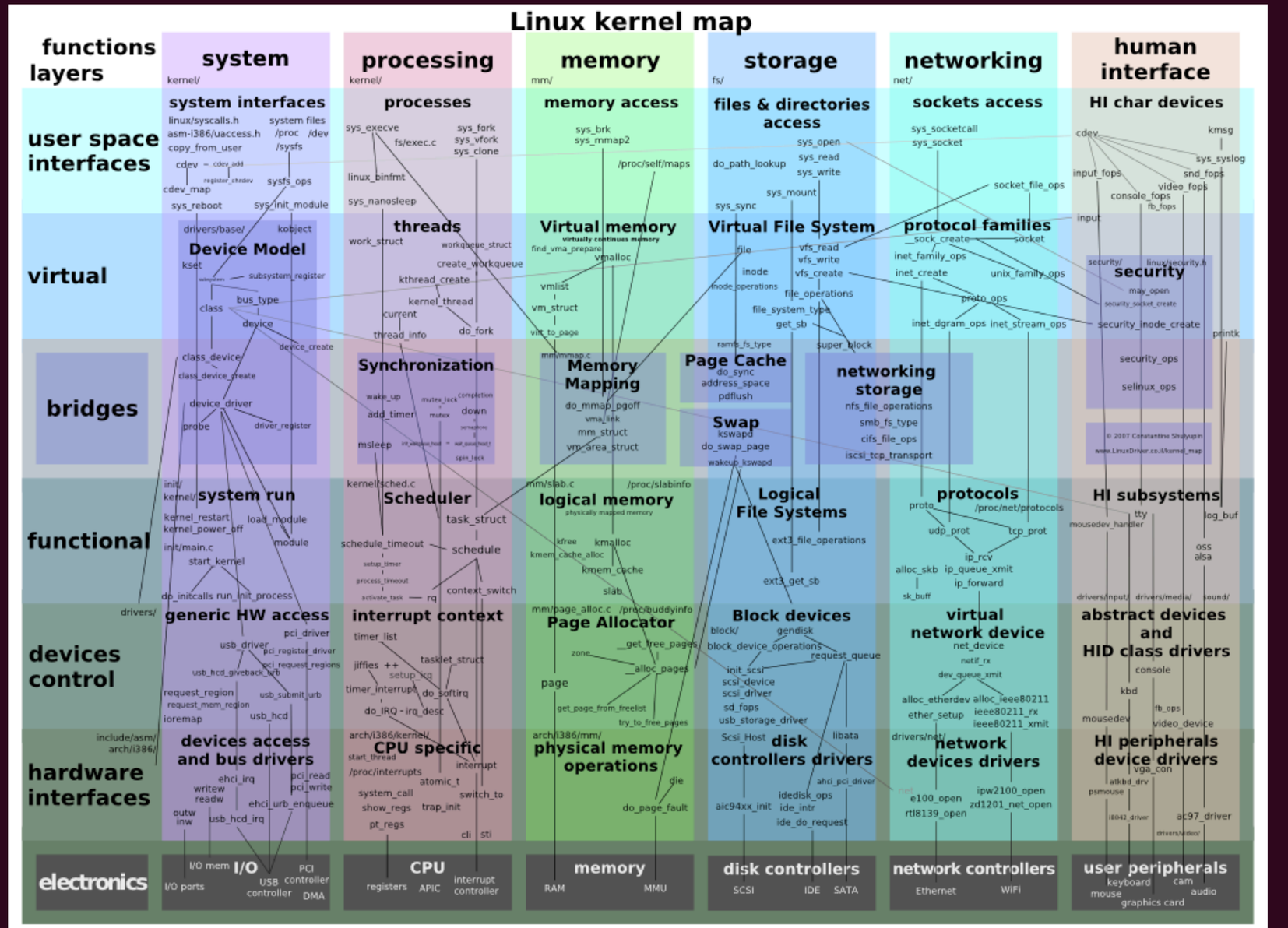
硬件视角下的操作系统

抽象视角下的操作系统

操作系统的复杂程度


Lines of code per Kernel version
Click and drag in the plot area to zoom in
20M
15M
10M
5M
0
Lines of code
1.0
1.2.7
1.3.37
1.3.80
2.0.22
2.1.24
2.1.67
2.1.110
2.2.20
2.3.36
2.4.28
2.5.33
2.6.0
3.0.3
3.0.46
3.0.89
3.2.19
3.3.8
3.4.42
3.6.1
3.9.7
3.12.2
3.13.1
3.19.1
4.5.4
4.9.2
Version
Lines of Code
Highcharts.com

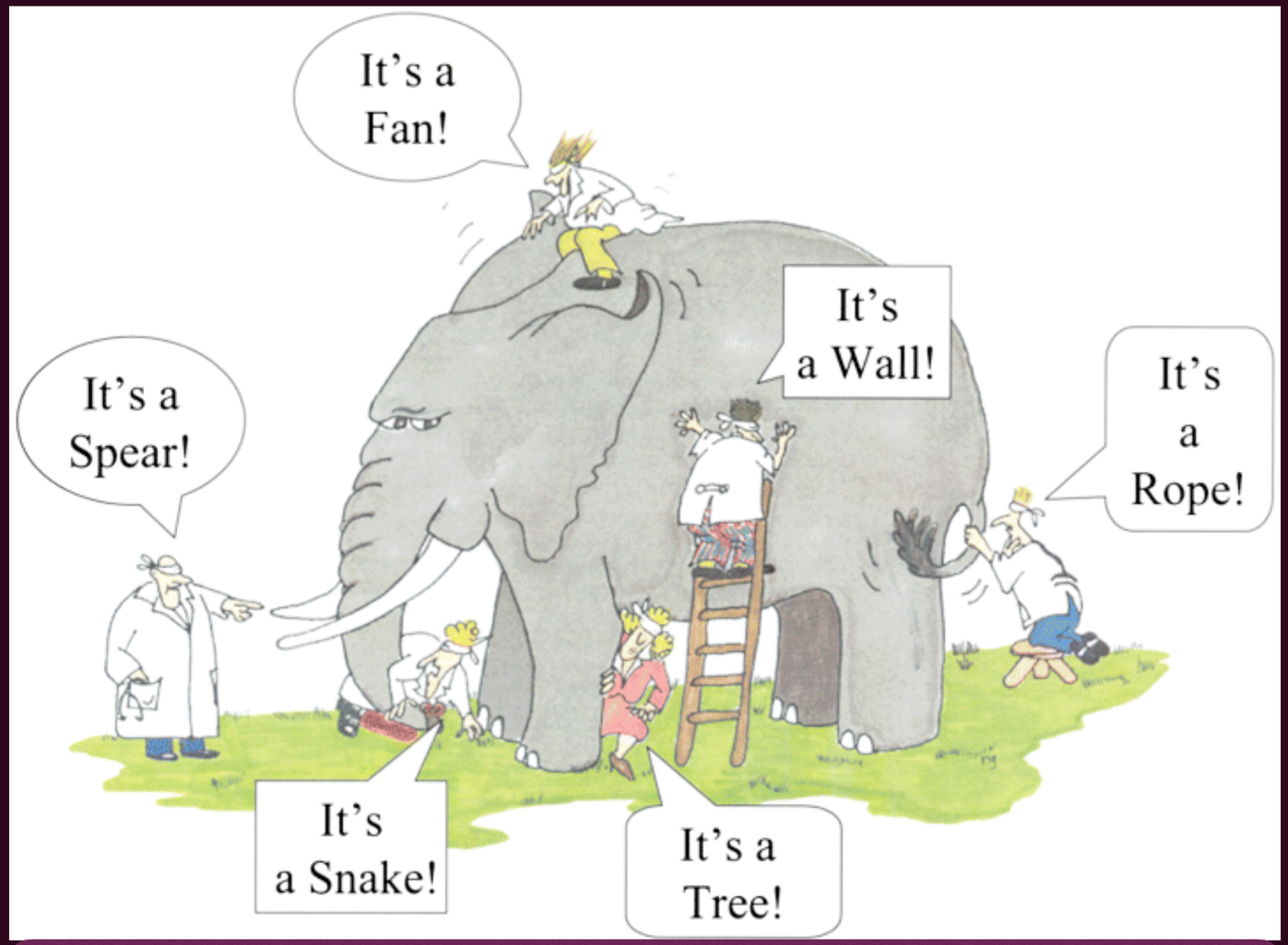
内核图


Linux kernel map
functions
layers
system
processing
memory
storage
networking
human interface
user space interfaces
virtual
bridges
functional
devices control
hardware interfaces
electronics
system interfaces
processes
memory access
files & directories access
sockets access
HI char devices
Device Model
threads
Virtual memory
Virtual File System
protocol families
security
Synchronization
Memory Mapping
Page Cache
networking storage
Swap
system run
Scheduler
logical memory
Logical File Systems
protocols
HI subsystems
generic HW access
interrupt context
Page Allocator
Block devices
virtual network device
abstract devices and HID class drivers
devices access and bus drivers
CPU specific
physical memory operations
disk controllers drivers
network devices drivers
HI peripherals device drivers
CPU
memory
disk controllers
network controllers
user peripherals
I/O
PCI controller
USB controller
DMA
registers
APIC
interrupt controller
RAM
MMU
SCSI
IDE
SATA
Ethernet
WiFi
keyboard
mouse
cam
audio
graphics card

规模的直观感受

- 如果比作一本书
  - 每页50行，一本1000页， 不过5万行
  - 需要100本上述的书才能写下Linux的内核2.x
  - 如果是windows10 （超过5000万行）则更多，需要1000到两千本。

怎样搞懂如此复杂的系统?



选对视角很重要!

回归主线

- 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。

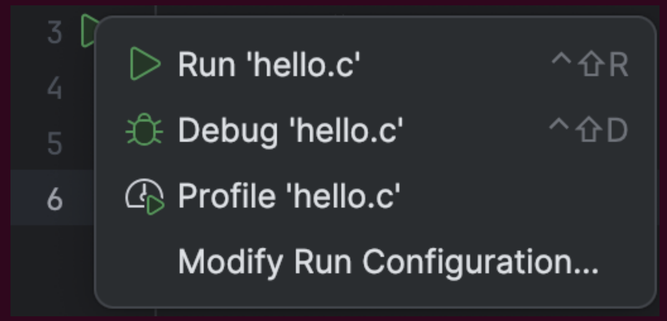
应用视角下的操作系统

什么是程序

- 让我们重新学习 “Hello World”

```c
#include <stdio.h>
int main(){
    printf(”Hello, World\n”);
}
```

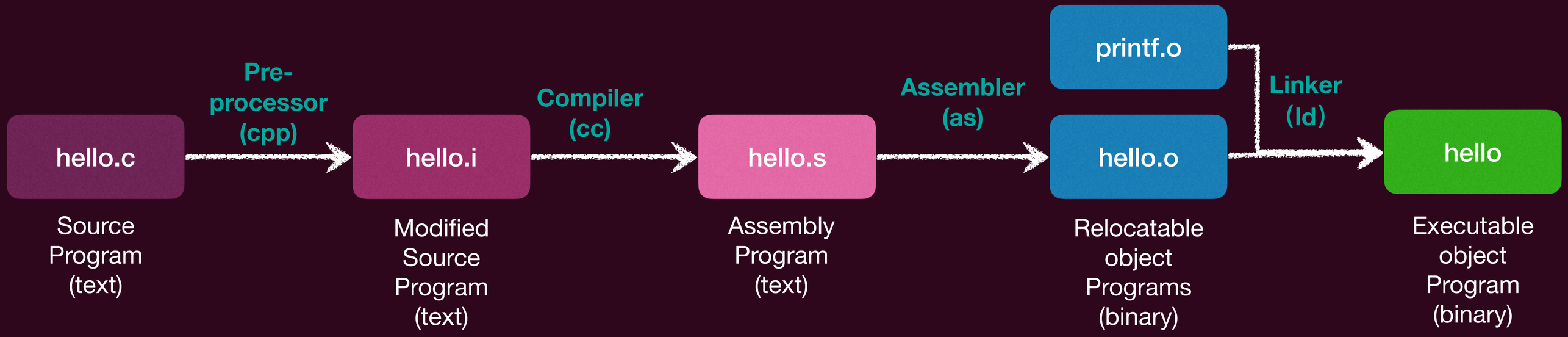
```c
#include <stdio.h>

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




```
Hello World

Process finished with exit code 0
```

什么是程序

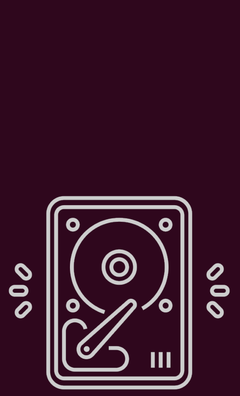
- IDE的好处：简单易用，但...细节呢?
- 到底发生了什么?



编译阶段：text → binary, 目标文件存在磁盘

什么是程序

- IDE的好处：简单易用，但...细节呢?
- 到底发生了什么?



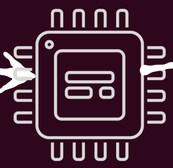
header
mapping info header
Code Instructions
Initialized Data
Symbol table and Info for debugging

Typical ELF executable object file

Loader(加载器)
将可执行文件加载到内存中
并设置好执行entry point

Kernel memory for this application
User Stack
shared libraries
Runtime Heap
Global Data Segment
Read-onlyCode Segment

从entry point开始执行



加载执行阶段: 磁盘→内存→CPU

什么是程序

- 让我们一个个的来看，首先，gcc编译出来的到底是啥?

```
ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ls
Makefile  hello.c  minimal.S
ubuntu@primary:~/Home/OSCodeShow/second/app-view$ gcc hello.c
ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ls -l a.out
-rwxr-xr-x 1 ubuntu ubuntu 15960 Feb 28 17:19 a.out
ubuntu@primary:~/Home/OSCodeShow/second/app-view$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6
4.so.2, BuildID[sha1]=adaa03e6841c6109398f6dd4d1dad268c812d7ee, for GNU/Linux 3.2.0, not stripped
ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ./a.out
Hello World
```

- 我们能看到 a.out的文件到底是什么样吗?
  - objdump 命令

```
objdump -d a.out
```

什么是程序

- gcc 编译出来的文件一点也不小 (试试 `gcc -static`会更有惊喜)
- objdump 工具可以查看对应的汇编代码
- 但是里面似乎有太多我们“没想过的”东西?
- gcc到底做了啥? `--verbose`选项
  - 编译选项不少

这么多东西都是我们想要的吗

- 让我们删掉一些东西看看，看看会不会影响我们的程序
- 毕竟，我们的诉求只是 print hello word
- 怎么做？自己控制编译流程！
- 从预编译开始！

```
cpp hello.c
```

也可以 `gcc -E hello.c`

- #include<stdio.h> 引入太多代码！
  - 删掉#include<stdio.h>！

```
[ubuntu@primary:~/Home/OSCodeShow/second/app-view$ gcc hello.c
hello.c: In function ‘main’:
hello.c:2:3: warning: implicit declaration of function ‘printf
    2 |   printf("Hello World\n");
      |   ^~~~~~
hello.c:1:1: note: include ‘<stdio.h>’ or provide a declaratio
  +++ |+#include <stdio.h>
    1 | int main() {
hello.c:2:3: warning: incompatible implicit declaration of bui
    2 |   printf("Hello World\n");
      |   ^~~~~~
hello.c:2:3: note: include ‘<stdio.h>’ or provide a declaratio
[ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ./a.out
Hello World
[ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ls -l a.out
-rwxr-xr-x 1 ubuntu ubuntu 15960 Feb 28 18:27 a.out
```

这么多东西都是我们想要的吗

- 从预编译开始！

```
cpp hello.c > hello.i
```

- 编译为Assembly programs

```
gcc -c hello.i
```

- 此时，得到hello.o，让我们用objdump看这里面是什么？

```
objdump -d hello.o
```

- 直接链接！

```
ld hello.o
```

这么多东西都是我们想要的吗

- 找不到 ‘puts’ !, 其实这就是gcc在背后帮我们做了这么多！找到了相应的库!
- 没关系，我们删除printf，是不是就可以正常走通了?
  - 当然，此时我们已经稍微背离一点我们的初衷，打印hello world, 但没关系，我们先构造一个程序再说

```
ld hello.o -e main
```

- 此时 a.out 已经生成，可以查看

```
objdump -d a.out
```

让我们运行这个程序

```
./a.out
```

```
[ubuntu@primary:~/Home/OSCodeShow/second/app-view$ ./a.out
 Segmentation fault (core dumped)
```

- 为什么?
  - 在**运行时刻**出错了，即在加载器把a.out这个文件加载到内存中运行出错了，我们应该在运行时刻进行调试
  - 怎么做呢?

调试 Segmentation Fault

- GDB! (一个可以在运行时刻帮我们调试程序的强大工具)
  - `starti` 可以帮助我们从第一条指令开始执行程序
  - `layout asm` 可以更方便地查看汇编
  - `info registers` 可以查看寄存器
  - `p` 查看存储在变量中的值
  - `x` 查看一个内存地址中的值
  - `si` 单步执行

STFW/RTFM or ask chatgpt (但要小心求证)

附：一个GDB的cheetsheet

调试 Segmentation Fault

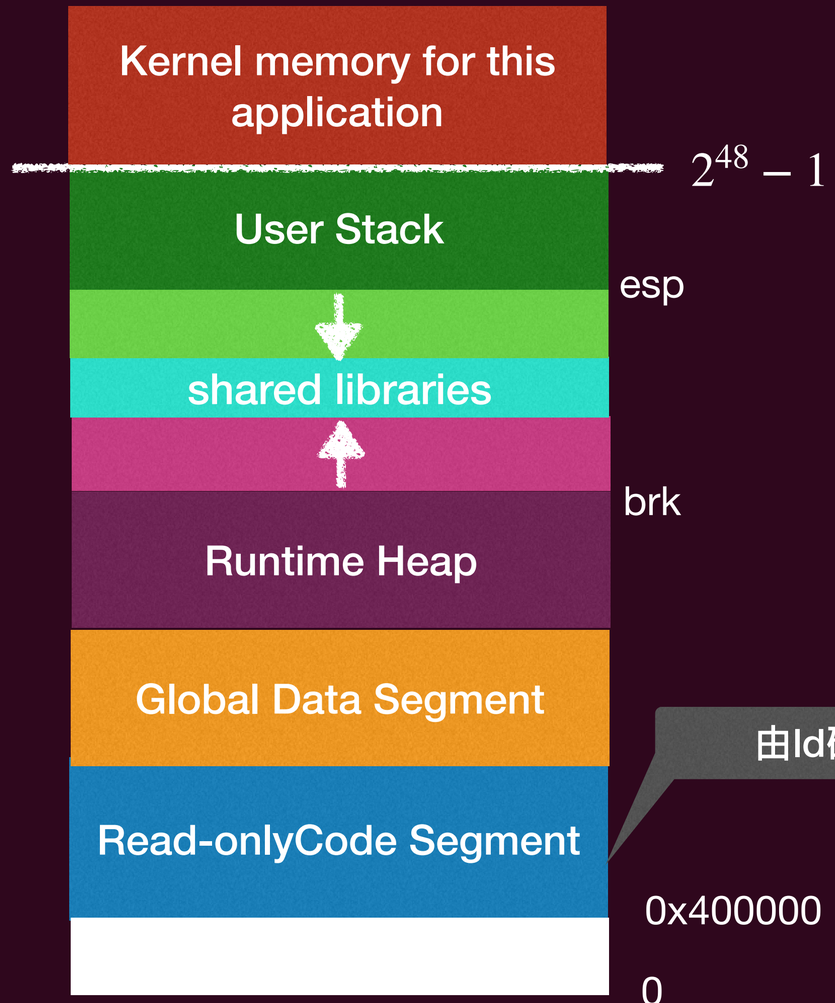
```
gdb a.out
starti
layout asm
info registers
p $sp
x $sp
si
si
si
...
```

```
0x0000000000000001 in ?? ()
(gdb) p $rip
$2 = (void (*)()) 0x1
(gdb)
```

ret 将 sp指针指向的地址pop到了ip上，此时PC = 1，我们取的是地址为 0x1上的指令

为什么PC不能为1?

- 在 Linux x86-64 系统上，代码段从0x400000开始（由底往上）
- 用户栈从 $(2^{48} - 1)$ 开始，由顶往下
- 在地址空间0到0x400000之间是?
  - Low addresses deliberately unmap



怎么修改正确呢?

- 不让他返回到main即可
- 怎么做?
  - 在main函数了加入无限循环!

```
int main(){
    while(1);
}
```

- 这样自然就不会执行 `ret`!

程序目前的指令

- 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`, …
    - 但这些指令甚至都无法使程序停下来

解决异常退出

- 有办法让状态机 “停下来” 吗?
  - 纯 “计算” 的状态机：不行
  - 要么死循环，要么 undefined behavior
- 解决办法：交给操作系统 （based on x86-64）

```
#include <sys/syscall.h>
int main() {
    register int p1 asm("rax") = SYS_exit;
    register int p2 asm("rdi") = 1;
    asm( "syscall" );
}
```

Hello, World 的汇编实现

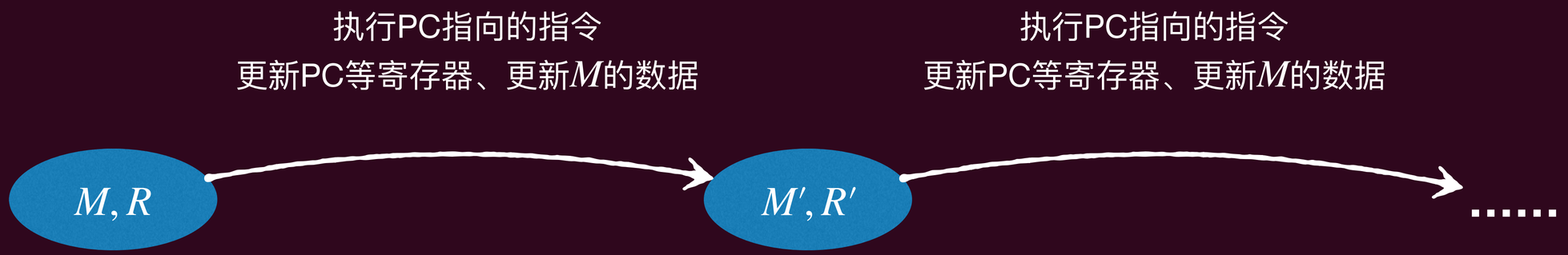
```
#include <sys/syscall.h>
.globl _start
_start:
  movq $SYS_write, %rax    // write(
  movq $1,         %rdi    //    fd=1,
  movq $st,        %rsi    //    buf=st,
  movq $(ed - st), %rdx    //    count=ed-st
  syscall                  // );

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

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

- 输入到minimal.S文本中，执行下面指令（x86-64）
- `cpp minimal.S > minimal.i`
- `as minimal.i -o minimal.o`
- `ld minimal.o`
- `./a.out`

回到“什么是程序”这个问题上来

- 一个理论模型：程序就是**状态机**
- 对于(二进制)程序而言
  - 状态 = M (内存) + E(寄存器)
  - 初始状态 = 程序启动时操作系统给安排的状态
  - 状态转移 = 执行一条指令

程序的状态机模型

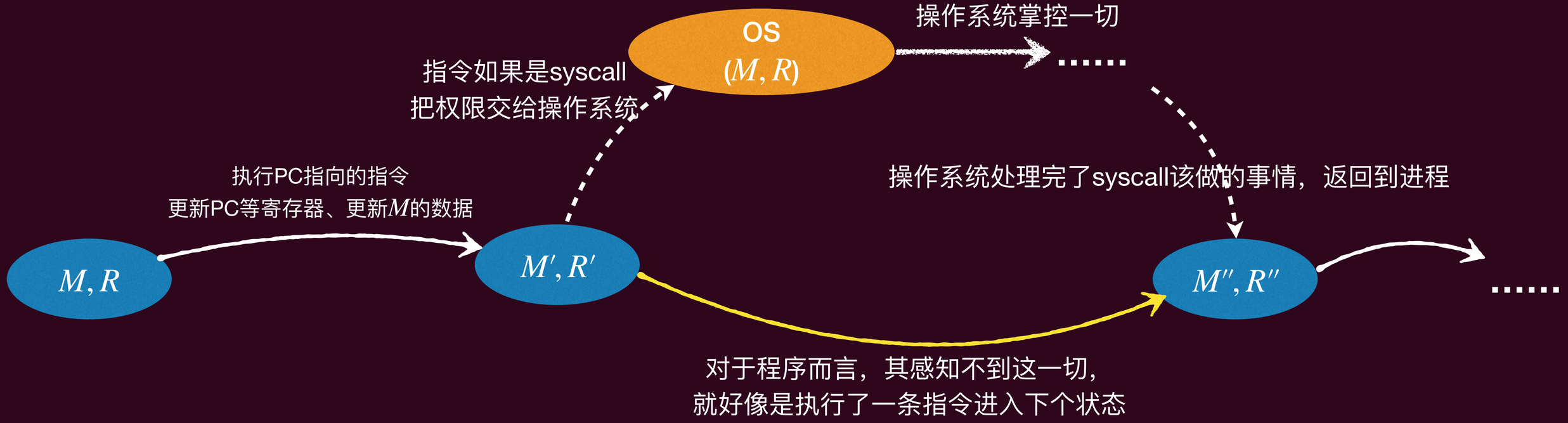


- 假设当前状态是(M, R)
  - 从$R[PC]$取一条指令
  - 解析指令，取出指令所必要的数据
  - 计算结果(可能有非确定性，例如 rdrand)
  - 更新得到(M', R')

一条特殊的指令

- 调用操作系统 syscall
  - 把(M, R)完全交给操作系统，任其修改
- 实现与操作系统中的其他对象交互
  - 读写文件/操作系统状态(例如把文件内容写入M)
  - 改变进程（运行中状态机）的状态，例如创建进程/销毁自己
- 有哪些syscall?
  - 试试 man 2 syscall 以及 man 2 syscalls

程序的状态机模型



- 从这个角度看，程序 = 计算 + syscall，而操作系统就像是syscall的解释器！
  - 普通的指令由cpu解释
  - syscall则由操作系统解释（当然，最终操作系统还是要借助cpu解释）

这也是为什么操作系统可以看成是更加高级的机器！

理解高级语言程序

- 高级语言本质上能描述的计算能力和汇编一致（都是人类能行计算，都逃不开图灵机的限制）
- 因此，事实上C语言写成的程序也可以看成是状态机模型
- 甚至不需要借助编译器，我们完全可以模仿汇编的单步执行
  - 比如我们可以构造一个如下的C语言解释器

```
while(true){
    stmt = fetch_and_parse_statement();
    evaluate(stmt, environment);
}
```

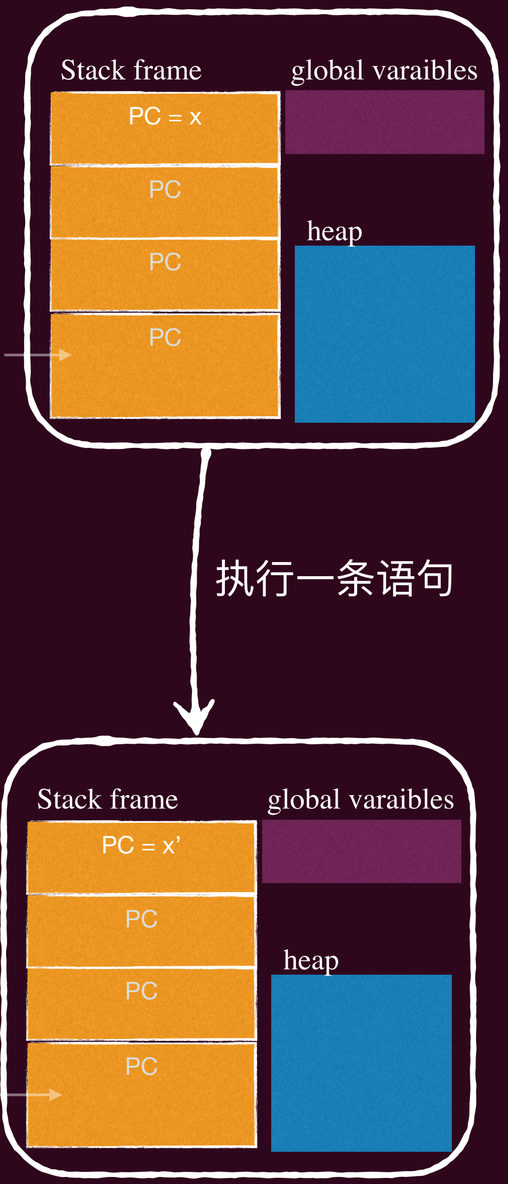
这就是程序状态！

我们在软工1这门课中已经实现过Java的REPL的解释器

理解高级语言程序

- C 程序的状态机模型 (简化语义，semantics)
  - 状态 = 堆 + 栈 + 全局变量
  - 初始状态 = 仅有一个 frame `main(argc, argv)`；全局变量为初始值
  - 迁移 = 执行一条简单语句
    - 执行当前调用栈栈顶的那个栈指针(frames.top.PC处)的简单语句
    - 如果是函数调用的话，本质上就是压入一个新的栈帧(stack frame)，并设置这个新的frame.PC = 所调用函数的入口
    - 函数返回 = pop frame

寄存器呢？C是更高的抽象！



理解高级语言程序

- 至此，我们有两种状态机
  - 汇编指令序列.s
    - 状态是(M, R); 执行指令进行状态迁移
  - 高级语言代码.c
    - 状态是栈、全局变量；状态迁移是执行语句
- 编译器即为二者的桥梁：.s = compile(.c)
  - 不同的优化级别可以产生不同的指令序列！

思考：这里怎么理解 “=”? 🤔

操作系统上的软件（应用程序）

- 操作系统中的**任何**程序本质上和minimal.S无异
  - 即为包含syscall的状态机
- 操作系统管理着所有的硬件/软件资源
  - 只能用操作系统允许的方式访问操作系统中的对象(syscall)
    - 这种集中管理有助于解决资源抢占和冲突

思考: 如果没有这个“中心”管理者，我们还可以解决资源抢占和冲突吗? 🤔

(二进制) 程序也是操作系统中的对象

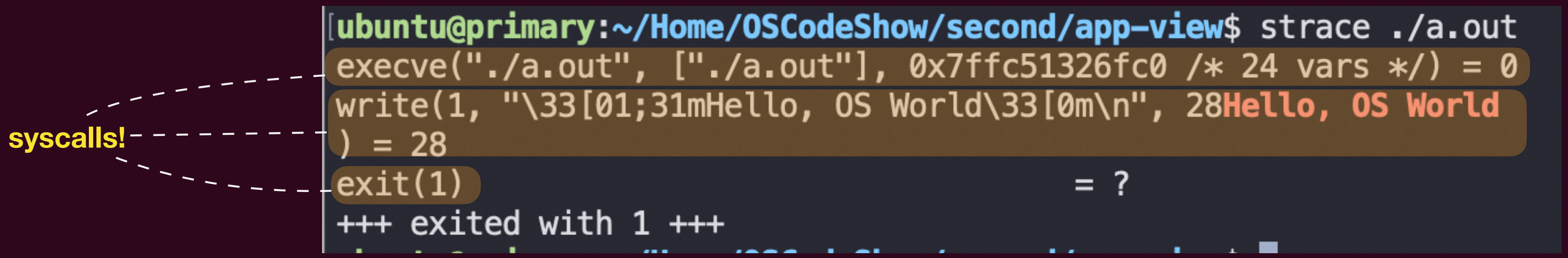
- 可执行文件
  - 与大家日常使用的文件 (a.c, README.txt) 没有本质区
    - 都可以打开、读取、甚至改写！
- 可执行文件的操作
  - `xxd` 可以16进制数直接查看可执行文件
  - vscode安装hex editor插件，可以直接编辑

操作系统中常见的应用程序

- Core Utilities (coreutils)
  - standard programs for text and file manipulation
    - 默认为GNU Coreutils，小巧的替代品：busybox
- 系统/工具程序
  - bash, binutils, apt, ip, ssh, vim, tmux, jdk, python, …
- 其他各种应用程序
    - 浏览器、音乐播放器、游戏…

打开程序的执行：Trace (追踪)

- 一个很重要的工具：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



为什么我们能够“追踪”进程

- 因为所有进程都在“操作系统”的监控之下！
  - 进程某种意义上是运行在“操作系统”这样的一个虚拟机中，而不是直面硬件，所以操作系统清楚进程的一切，也能修改其一切！
- 操作系统为进程提供了`ptrace`系统调用，其可以帮助一个进程去查看另外一个进程，甚至是修改另外进程的运行时的寄存器，以至于植入代码！
  - Strace、GDB背后都是`ptrace`系统调用！
  - 凭借这个系统调用，你们甚至可以实现自己的“动态”程序分析器！

操作系统中“任何程序”的一生

- 初始状态
  - 执行`execve`加载到内存，设置初始状态
- 状态机执行
  - 进程管理(-e trace=%process)：`fork, execve, exit, ...`
  - 文件/设备管理(-e trace=%file)：`open, close, read, write, ...`
  - 存储管理(-e trace=%memory)：`mmap, brk, ...`
  - 调用`exit`或者`exit_group`退出

操作系统中“任何程序”的一生

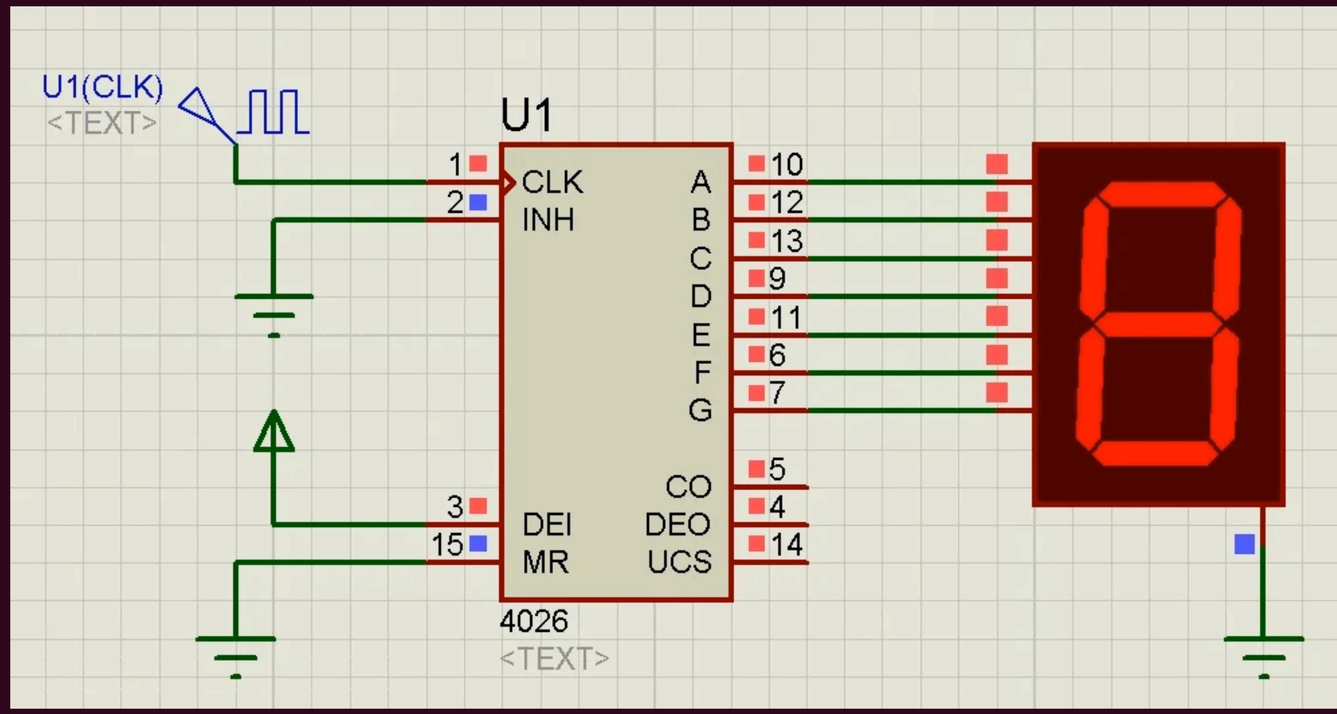
- 所有的这些程序都是在操作系统 API (syscall) 和操作系统中的对象上构建
  - 本质都是调用 syscall 的状态机
  - 对于这些应用而言，操作系统就是一个帮助解释这些syscall的存在而已，其和能够帮他们解释普通计算指令的CPU无异！

简而言之，在应用眼中，操作系统就是syscall的解释器！

硬件视角下的操作系统

数字电路与状态机

- 硬件的核心是一堆数字电路
  - 状态 = 寄存器保存的值（flip-flop）
  - 初始状态 = REST
  - 迁移 = 组合逻辑电路(NAND, NOT, AND, OR, NOR…)计算寄存器下一时钟周期的值



计算机硬件的状态机模型

- 整个计算机系统也是一个状态机
  - 状态：内存和寄存器数值
  - 初始状态：CPU Reset
  - 状态迁移：
    - 任意选择一个处理器CPU
    - 响应处理器外部中断
    - 从CPU的PC取指令执行

计算机硬件的状态机模型

- 为了让“操作系统”这个程序能够正确启动，计算机硬件系统必定和程序员之间存在**约定**：
  - 首先就是 Reset 的状态。
  - 然后是 Reset 以后执行的程序应该做什么。
- 这么想，计算机硬件和操作系统就一点也不神秘了。

硬件与程序员的约定

- Bare-metal 与厂商的约定
  - CPU Reset 后的状态 (寄存器值PC)
    - 根据PC指向的地址取指令、译码、执行
    - 厂商自由处理这个地址上的值
      - 通过Memory-mapped I/O将其映射到主板上的存储器ROM，ROM中的代码（Firmware，固件）会执行

- 厂商为操作系统开发者提供Firmware
  - 管理硬件和系统配置
  - 把存储设备上的代码加载到内存
    - 例如存储介质上的第二级 loader (加载器)
    - 或者直接加载操作系统 (比如嵌入式系统)

x86 Family: CPU Reset

- CPU Reset (Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A/3B)
  - 寄存器会有确定的初始状态
    - `EIP = 0x0000fff0`
    - `CR0 = 0x60000010`
      - 处理器处于实模式(real mode or real address mode)，页表机制关闭
      - 此时CPU处在16bit的状态
    - `EFLAGS = 0x00000002`
      - Interrupt disabled

9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor is in real-address mode with paging disabled.

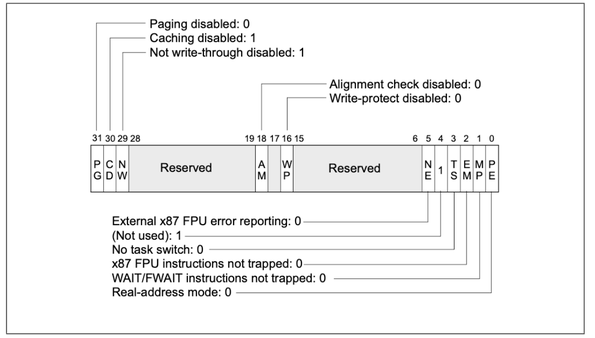


Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

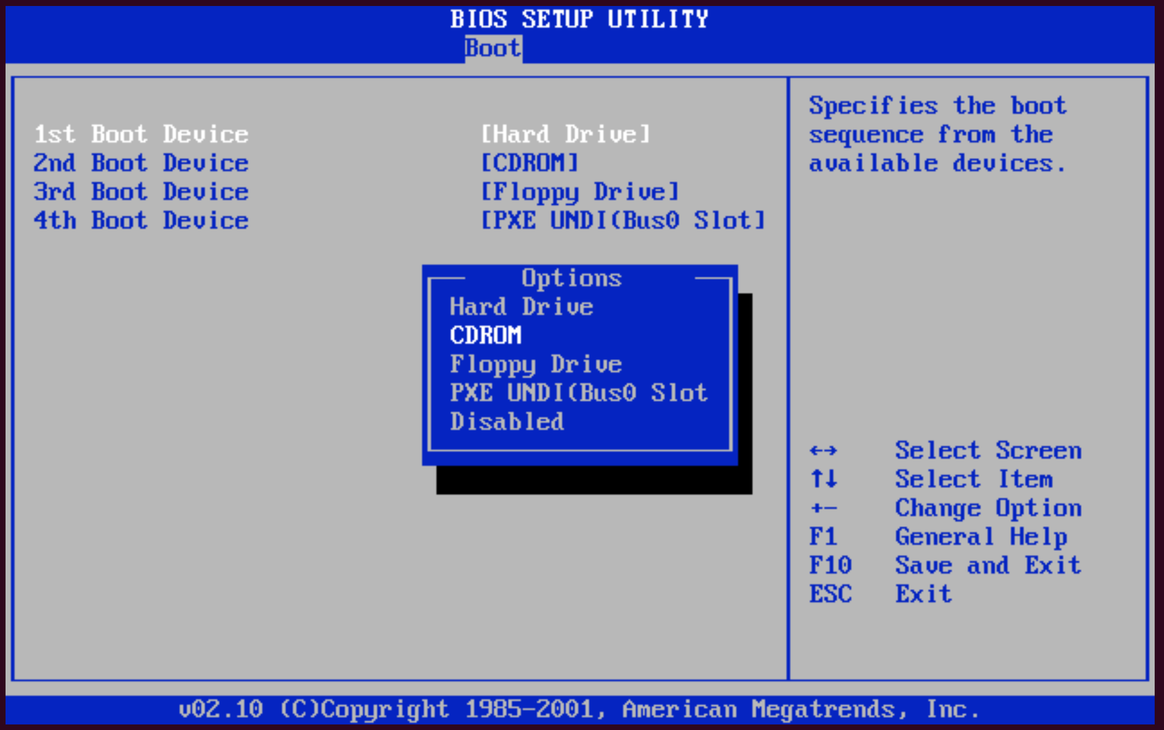
Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5]	+0.0	+0.0	FINIT/FNINIT: Unchanged

其他平台上的 CPU Reset

- Reset 后处理器都从固定地址 (Reset Vector) 启动
  - MIPS: `0xbfc00000`
  - ARM: `0x00000000` (允许配置 Reset Vector Base Address Register)
  - RISC-V: Implementation defined (给厂商最大程度的自由)
- 然后执行Firmware中的代码，加载操作系统

Firmware标准

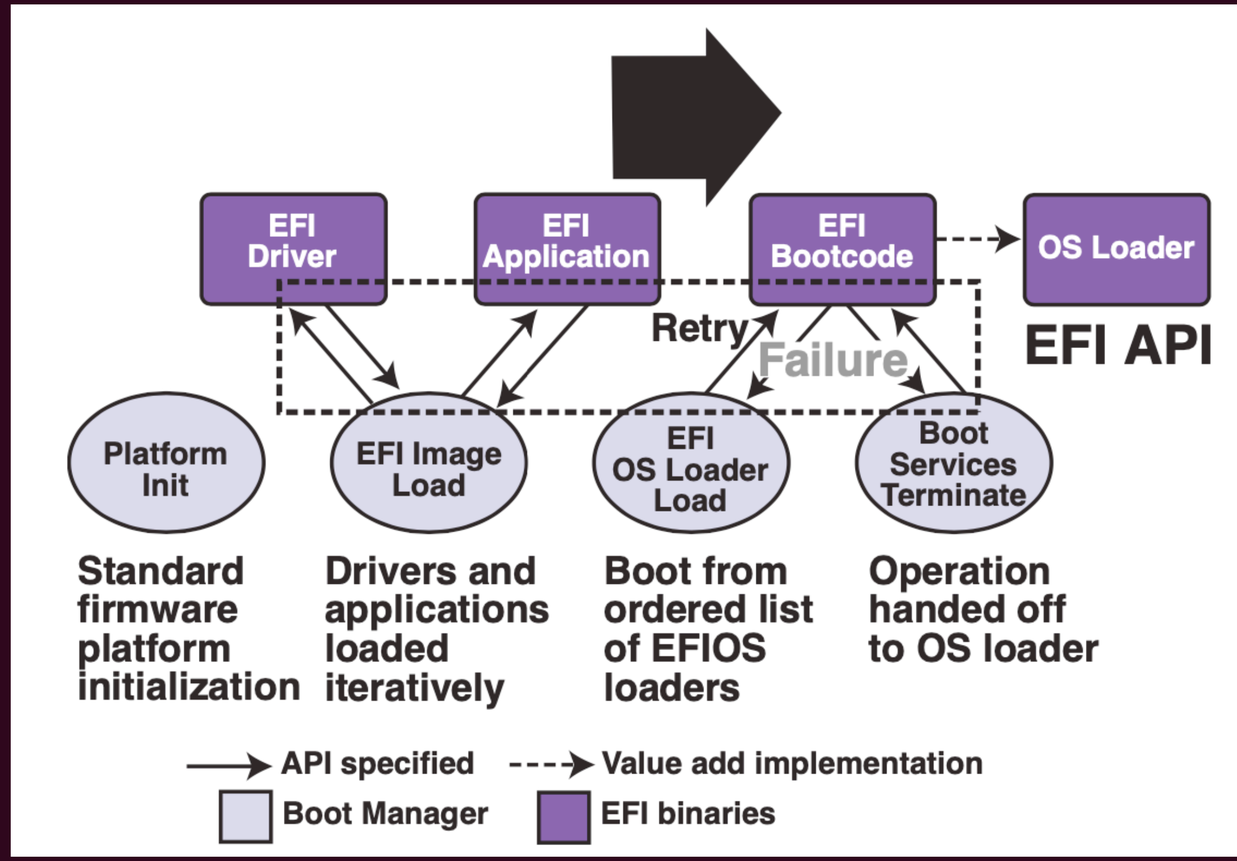
- BIOS(**B**asic **I**nput/**O**utput **S**ystem.)



- UEFI(**U**nified **E**xtensible **F**irmware **I**nterface)

为什么需要 UEFI?

- 今天的 Firmware 面临麻烦得多的硬件：
  - 指纹锁、USB 转接器上的 Linux-to-Go 优盘、USB 蓝牙转接器连接的蓝牙键盘、…
    - 这些设备都需要 “驱动程序” 才能访问
    - 而传统BIOS只能支持有限的硬件
- 除此之外，UEFI更加灵活，支持更多的模式（比如secure boot等），性能也更好！

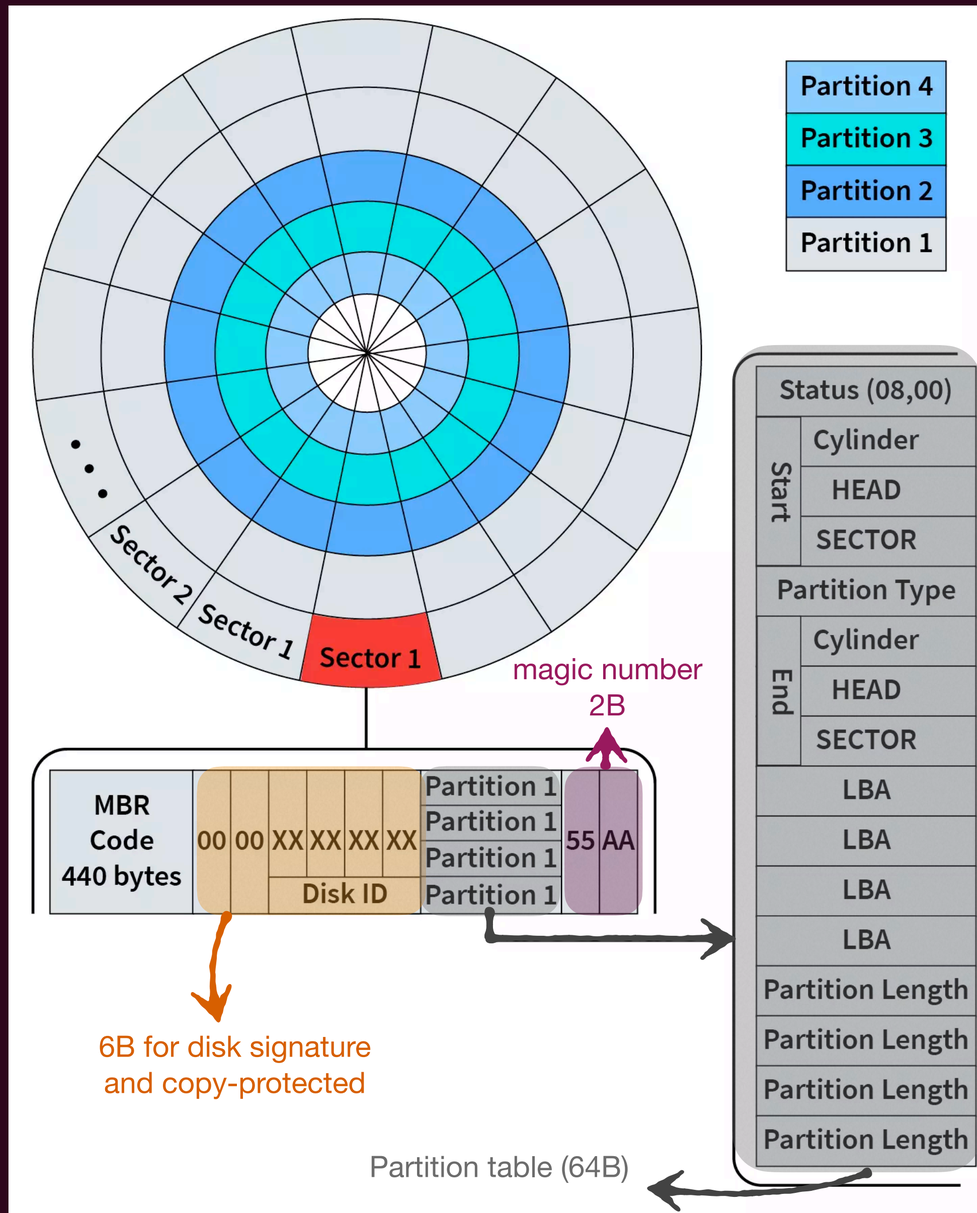


回到 Legacy BIOS: 约定

- BIOS 提供机制，将程序员的代码载入内存
  - Legacy BIOS 把第一个可引导设备的第一个 512 字节加载到物理内存的 `7c00` 位置
  - 此时处理器处于 16-bit 模式
  - 规定 `CS:IP = 0x7c00`
  - 其他没有任何约束
- 虽然最多只有 446 字节代码 (64B 分区表 + 2B 标识)，但控制权已经回到程序员手中

Master Boot Record (MBR)

Magic Number



UEFI 上的操作系统加载

- 标准化的加载流程
  - 磁盘必须按 GPT (GUID Partition Table) 方式格式化
  - 预留一个 FAT32 分区 (可以用命令lsblk/fdisk 查看)
  - Firmware 能够加载任意大小的 PE 可执行文件`.efi`
    - EFI 应用可以再次返回 firmware

CPU Reset之后的世界

- 可以查看CPU Reset 之后的每一条指令的执行?
  - 如何观察一个计算机系统的指令运行?
  - 答案：在其之上!
- 模拟方案：QEMU （大佬 Fabrice Bellard的作品）

  - QEMU模拟硬件系统，在模拟的硬件之上再加载操作系统，那么宿主机（运行QEMU的机器）即可观察这一切
  - QEMU的技术已经被应用于Android Virtual Device、KVM、XEN、VirtualBox等多个虚拟化项目中。

CPU Reset之后的世界

- QEMU配合GDB

启动gdb服务器，端口tcp::1234

启动后CPU先暂停

编译后的MBR

```
qemu-system-x86_64 -s -S mbr.img
```

```
gdb
target remote localhost:1234
```

gdb连接qemu开的服务器进行调试

- 可以通过gdb调试看到：
  - CPU Reset之后的PC指向的地址0xfff0
  - MBR的第一条指令被加载到0x7c00
  - 利用GDB的watch point可以看到是谁把MBR加载到内存的之处

可以定制gdb脚本加速这些流程

CPU Reset之后的世界

- 当然，MBR不是为了打印“Hello,World!”而存在的，而是为了加载操作系统存在的
  - 因此MBR里的程序叫做bootloader(加载器)
  - 当然，在真实环境中，MBR会加载二级bootloader而不是操作系统，比如对于Linux而言，bootloader程序GRUB就是两阶段的
    - 先执行GRUB在MBR里512字节里的程序，然后再从磁盘加载剩余的程序到内存，并执行这个二级bootloader，最后由这个二级bootloader加载磁盘中的操作系统内核到内存中！

加载器的工作细节

- 假设 MBR 包含的是一级bootloader，做完一些必要的处理器初始设置之后
  - 将16-bit → 32-bit模式
  - 跳转到 ELF32/64 的加载器
    - 按照约定的磁盘镜像格式加载操作系统内核 （一个编译完成的ELF文件）

```
if (elf32->e_machine == EM_X86_64) {
  ((void(*)())(uint32_t)elf64->e_entry)();
} else {
  ((void(*)())(uint32_t)elf32->e_entry)();
}
```

比如对于 am/src/x86/qemu/boot/start.S 和 main.c 最后跳转到这里

具体例子：实现最小“操作系统”

- 我们已经可以让机器运行任意不超过 512 字节的指令序列 (本门课为了简单，默认固件为BIOS)
- 而操作系统就是一个C程序而已
  - 用 512 字节的指令将磁盘的C程序加载内存
  - 初始化 C 程序的执行环境
  - 操作系统就开始运行了！

Bare-metal 上的 C 代码

- 要在一个Bare-metal运行一个简单的C程序我们需要准备什么?
  - Master Boot Record (MBR) 上的 “启动加载器” (Boot Loader) 加载C程序
  - 我们可以通过编译器控制 C 程序的行为
    - 静态链接
    - Freestanding (不使用任何标准库)
      - 编译器 (gcc) 提供了选项(-fno-hosted)帮我们生成不依赖操作系统的目标文件

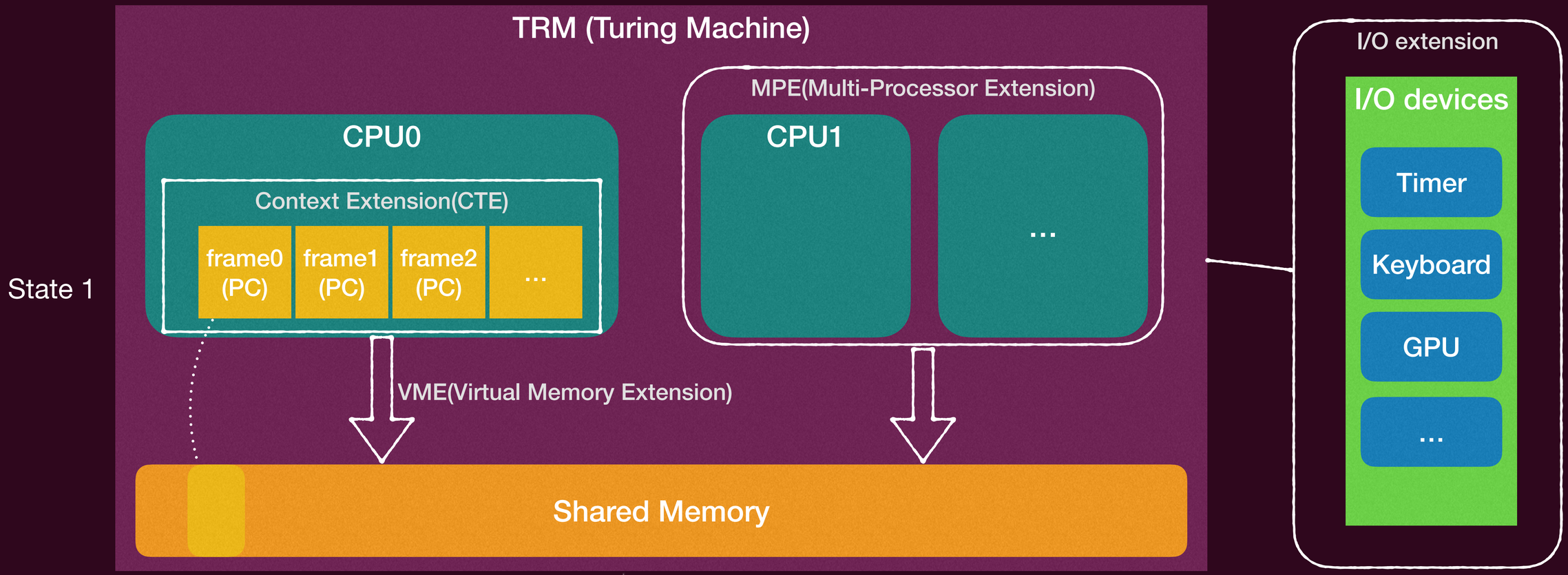
但是没有标准库，我们很多事情做不了！ 没有 printf、malloc… 我们需要实现他们!

不同的硬件体系

- 实现这个操作系统的第一个难题就是硬件体系太多(x86, MIPS, RISC-V), 在本课程中我们不想每个体系都实现一遍，如何屏蔽这些不同?
  - 硬件抽象! 本门课程中我们应该在一个硬件抽象层上实现我们的操作系统，这个抽象层能够帮助C程序访问硬件
    - 业界已经有很多这样的抽象了，本门课提供的抽象机器是由蒋炎岩和余子濠两位老师提供的AM(Abstract Machine)，其设计文档可见：https://jyywiki.cn/OS/AbstractMachine/index.html

```
git clone https://github.com/NJU-ProjectN/abstract-machine.git
```

AM



State 1

execute one instruction (AM is also a state machine)

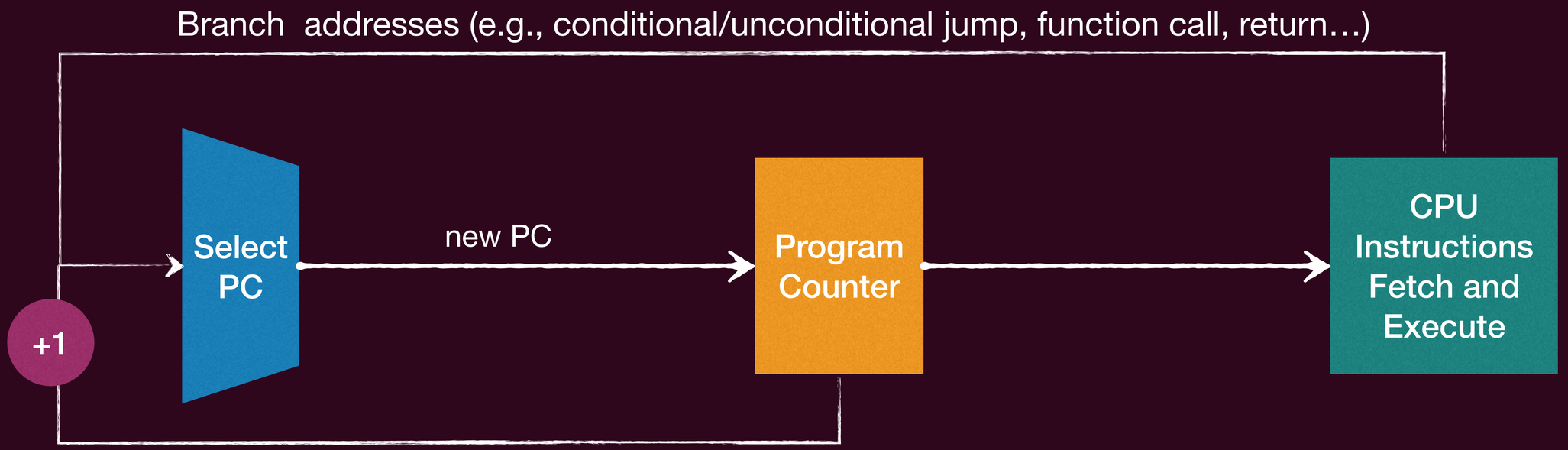
- Instructions in a normal flow
- Instructions to handle interrupts (after an event happens)

State 2

.....

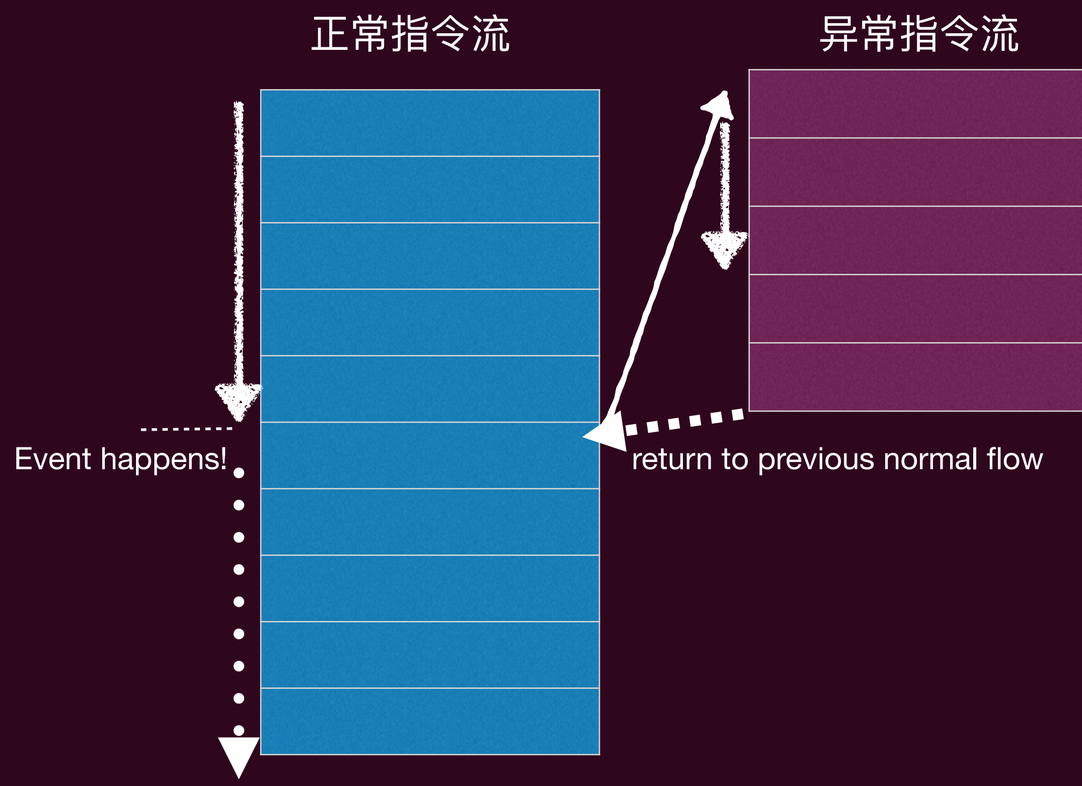
两种执行流

- 普通控制流(normal control flow)：
  - 即遵循冯诺依曼结构的指令循环



两种执行流

- 异常控制流(Exceptional Control Flow):
  - 不是线性的正常的指令流（PC的转移不受正常的自增、或者branch instrunctions控制）
  - 由“外界”强制转移到另一块指令入口（强制赋予PC）
    - 比如：遇到中断、异常、自陷指令时，CPU会转移到另外的指令流入口



比如，发生某个中断时，实地址模式下CPU会查看中断向量表(Interrupt Vector Table (IVT))相应的中断处理程序（ISR）的入口，并转向该入口，如果在保护模式下（所有现实操作系统）CPU会查看中断描述表（Interrupt Descriptor Table (IDT)）

异步事件处理

- 很多用户应用也有打断正常执行流的需求
  - 游戏里需要响应玩家的键盘和鼠标事件
  - 网络聊天应用需要响应网络包达到的事件
  - …
- 这些都需要操作系统在背后支持以及硬件支持！
  - 比如应用会被 Ctrl+C终止：就是发生了一个键盘输入中断，操作系统中断处理程序处理了这个Ctrl+C，发现是Ctrl+C之后给相应的应用程序发送SIGINT信号，相应的应用程序终止当前正常执行指令流，转向自己的SIGINT信号处理函数，最终终止自己！

Event-driven programming!

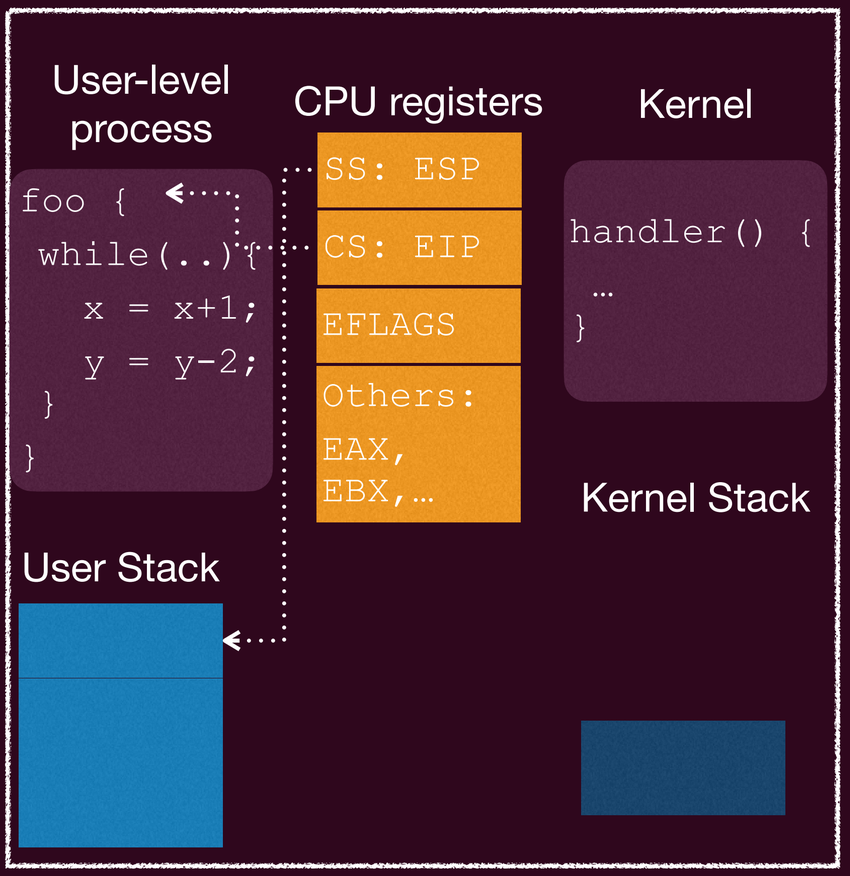
上下文切换（Context Switch）

- 当然进入另外入口前，CPU还要做“现场保护”，否则就回不来了！
- 所谓的现场就是当前CPU的一些寄存器的值（因为这些不像是栈、堆保存在内存中，一旦由于为了执行另外的指令流而赋予了其他值，旧的值就“消失”了）
- 这就是上下文！
  - 不同的指令架构上下文会有些区别（AM统一抽象为Context结构体），大体上包含：
    - PC寄存器（CS、IP）、栈指针（SP、SS）、控制寄存器、 virtual address translation入口寄存器、其他一些数据寄存器
  - 这个“现场”将会保存在内核栈中（安全可靠！），等异常控制流处理完毕时，并再次调度这个执行流时，这个“现场”将会被弹出到CPU上，恢复之前的执行！

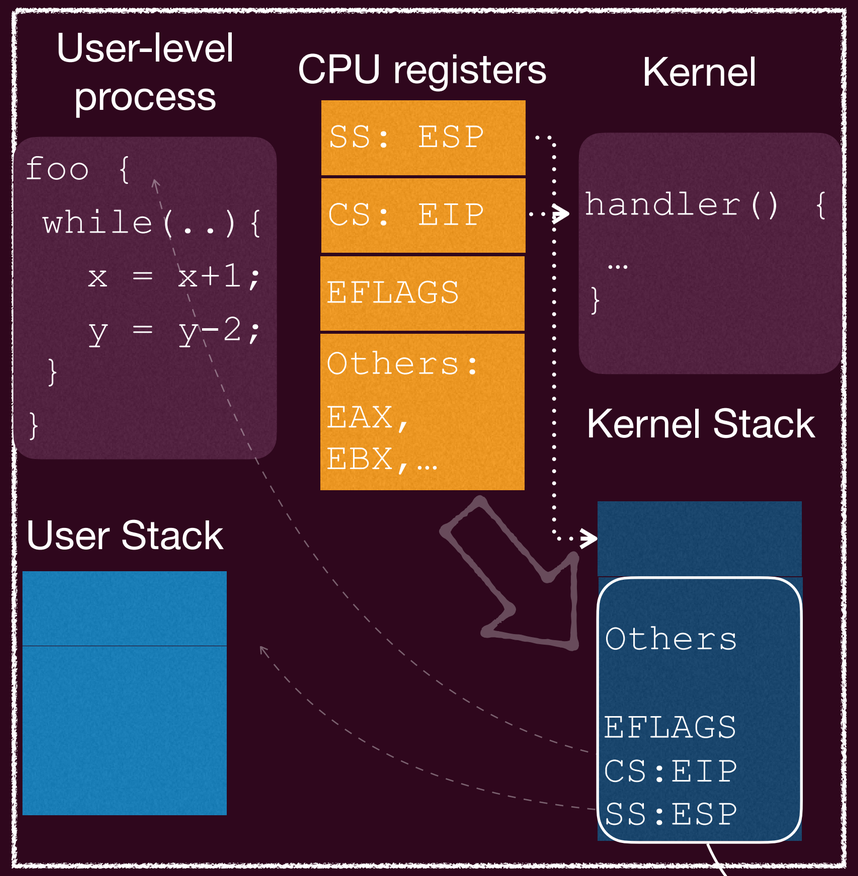
上下文切换（Context Switch）

X86上下文切换

在中断处理时出现中断怎么办？🤔 —disable it (it will be stored in a queue) during interrupt handling, and enable it when handling is ending

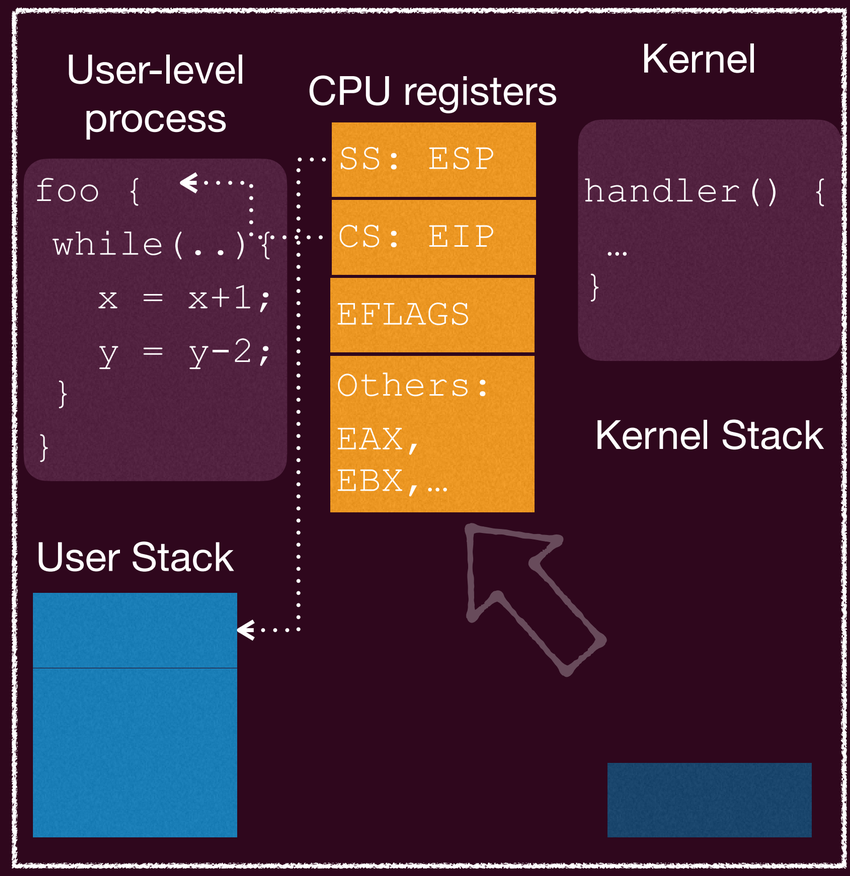


Before exception event



After happening of an exception event

Context!

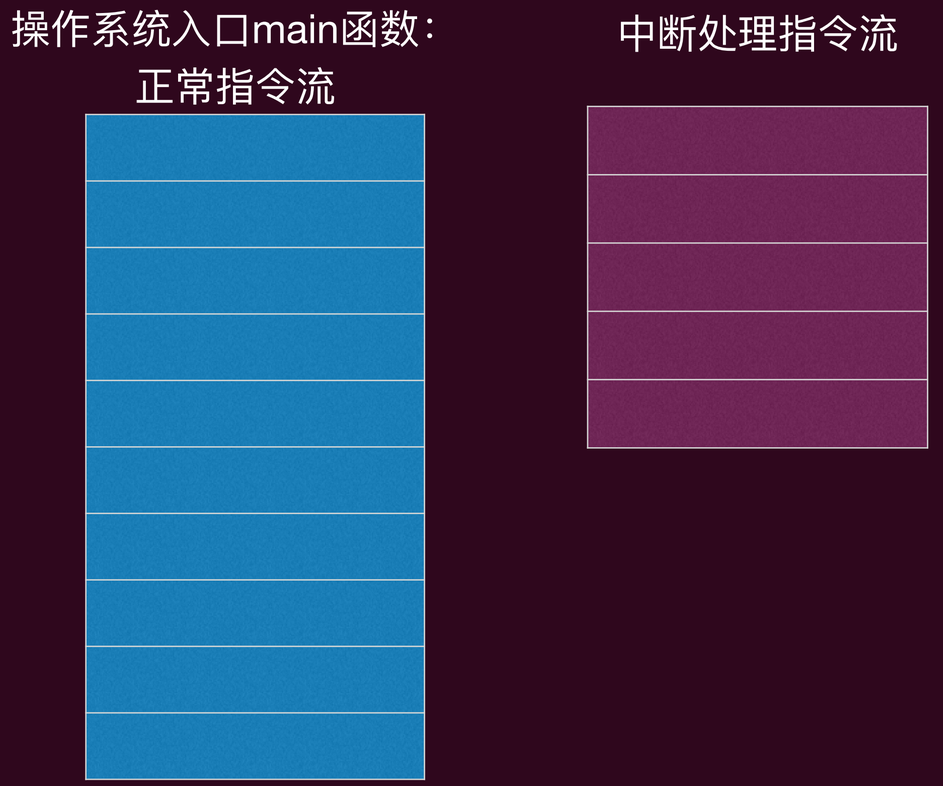


Exception handling ends and the user process get scheduled

Context给了我们有多个CPU的虚像！是分时操作系统的核心

我们怎么在AM上实现操作系统?

- 本质上，操作系统只能控制两个部分：
  - 自己的main函数 (经历CPU reset—> Bios —> MBR —>C环境初始化之后才最终到达的函数)
    - 一般用来初始化整个计算系统环境
  - 中断响应的函数
    - 响应异步事件的指令流，会打断当前的正常的指令流，强制转移到处理中断事件的指令流，一般用来实现各种服务
- 看看https://github.com/NJU-ProjectN/am-kernels里的一些例子吧！



这些就是硬件眼中的操作系统！

我们得到了硬件下的操作系统视角

- 计算机系统的一切行为都是可观测、可理解的。
  - 处理器的任务就是重复着进行着取指令和执行指令
  - 厂商配置好处理器 Reset 后的行为：先运行 Firmware，再加载操作系统
  - 操作系统就是一个C程序！
    - 只不过其拥有完整计算机的控制权限，包括中断和 I/O 设备

简而言之，在硬件眼中，操作系统就是个 C 程序，只是能直接访问计算机硬件

抽象视角下的操作系统

我还是没明白什么是操作系统?

- 我们拥有了两个视角，但都是从侧面
  - 应用的视角（自顶向下）
  - 硬件的视角（自底向上）
- 我们没有真正从全局去看操作系统，没有对它的一个整体的概念
  - 但操作系统太复杂了，怎么全局去看?
    - 抽象的视角!
    - 操作系统本身就是状态机!

操作系统的一生！

初始化阶段

Many user processes will start from here!

Bootloader transfer to load kernel

initialize and configures memory, processors, I/O, and storage devices, sets up interrupt handling (IRQs).

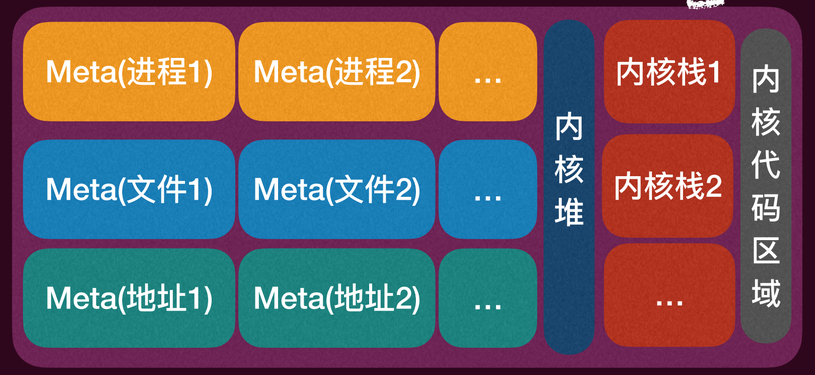
Mount the initrd and load essential drivers then Unmount initrd

Mount the root filesystem

init

Start the first user-level process: **init** (the newest version of **init** in linux is **systemd**)

yield cpu



形成这样一个状态机

操作系统的一生！

运行阶段

用户进程1

用户进程2

...

自陷(trap)

返回用户空间，用户进程继续

syscall API

执行相应系统调用代码

执行相中断处理程序

Meta(进程1)	Meta(进程2)	...	内核堆	内核栈1	内核栈2	...	内核代码区域
Meta(文件1)	Meta(文件2)	...					
Meta(地址1)	Meta(地址2)	...					

s

Meta(进程1)	Meta(进程2)	...	内核堆	内核栈1	内核栈2	...	内核代码区域
Meta(文件1)	Meta(文件2)	...					
Meta(地址1)	Meta(地址2)	...					

s'

Meta(进程1)	Meta(进程2)	...	内核堆	内核栈1	内核栈2	...	内核代码区域
Meta(文件1)	Meta(文件2)	...					
Meta(地址1)	Meta(地址2)	...					

s''

可能需要硬件（如disk）辅助

发生某些中断事件（如硬盘读写完成）

硬件1

硬件2

...

Meta为元信息，比如对于进程而言，操作系统只在内核中维护进程的元信息（进程控制表，存放如进程id、进程栈指针、PC...）

操作系统的一生！

- 操作系统本身就是**状态机**
  - 内部状态为用户进程的元信息、内核栈、内核堆、操作系统代码区
  - 操作系统在硬件加载完毕和初始化之后就变成了就成为了 interrupt/trap/fault handler
  - 操作系统的状态是**被动**迁移的，用户程序执行syscall才会改变操作系统状态、硬件中断事件发生后（如时钟中断）才会改变操作系统状态

让我们用代码实现一个抽象的操作系统！

- 我们真正关心的是：
  - 应用程序
  - 系统调用 (操作系统 API)
  - 操作系统内部实现
- 一个Toy的实现思路：
  - 应用程序 = 纯粹计算的 Python 代码 + 系统调用
  - 操作系统 = Python 系统调用实现，有 “假想” 的 I/O 设备

操作系统玩具：API

- 四个“系统调用” API
  - `sys_choose(xs)` 返回 `xs` 中的一个随机选项
  - `sys_write(s)` 输出字符串 `s`
  - `sys_spawn(fn)` 创建一个可运行的状态机 `fn`
  - `sys_sched()` 随机切换到任意状态机执行
- 为了足够“toy”，我们先不关心硬件部分

玩具系统编程

- 有了这四个“系统调用”，我们的“应用程序”可以完成如下的编程

```
count = 0

def Tprint(name):
    global count
    for i in range(3):
        count += 1
        sys_write(f'#{count:02} Hello from {name}{i+1}\n')
        sys_sched()

def main():
    n = sys_choose([3, 4, 5])
    sys_write(f'#Thread = {n}\n')
    for name in 'ABCDE'[:n]:
        sys_spawn(Tprint, name)
    sys_sched()
```

实现系统调用

- 有些“系统调用”的实现是显而易见的

```
def sys_write(s): print(s)
def sys_choose(xs): return random.choice(xs)
def sys_spawn(t): runnables.append(t)
```

- 有些就不那么容易了

```
def sys_sched():
    raise NotImplementedError('No idea how')
```

一个好的编程习惯，未实现先填一个raise message

实现系统调用

- 切换状态机是“分时”操作系统的核心
  - 其对于被切换的状态机必须是“透明”的，其不知从运行中被切换了，也不知被切换回来继续运行。
  - 为此，我们需要:
    - 封存当前状态机的状态（保护现场，使其能够在未来被切换回来时，对于其而言，运行环境没有变化）
    - 恢复另一个 “被封存” 状态机的执行

借用 Python 的语言机制

- 在Python里什么对象可以被暂时终止运行，然后还可以被再度唤醒继续执行?
- Generator objects(生成器)
  - `yield` 关键词可以暂停当前执行流，返回给 `next()` 的调用者！然后下一次调用 `next()` 时继续正常执行
    - 能做到这一点，当然是Python帮助封存了yield处的执行环境了！然后在下一次调用时切换回来

```
def number(a):
    while True:
        a += 1
        yield
```

```
>>> def number(a):
...     while True:
...         a+= 1
...         yield
...
>>> c = number(3)
>>> next(c)
>>> next(c)
```

借用 Python 的语言机制

- 生成器可以“**产生**”（a.k.a. yield）某个值给外界调用者，有点像其在**让出**（a.k.a. yield）CPU使用时给外界的信息！

```
def number(a):
    while True:
        a += 1
        yield a
```

```
>>> def number(a):
...     while True:
...         a+= 1
...         yield a
...
>>> c = number(3)
>>> next(c)
4
>>> next(c)
5
```

- 我们能不能在其再次获得CPU的时候从外界得到点什么呢?

借用 Python 的语言机制

- 我们还可以在执行时向内传递信息！

```
def number(a):
    while True:
        a += 1
        a = yield a
```

next再次启动这个生成器时，由于没有从外界得到值，a得到了一个None！无法做加法！

```
>>> def number(a):
...     while True:
...         a+= 1
...         a = yield a
...
>>> c = number(3)
>>> next(c)
4
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in number
TypeError: unsupported operand type(s) for +=: 'NoneType' and 'int'
```

NoneType?

使用send向生成器传递信号，并让其继续执行！

```
>>> c = number(3)
>>> next(c)
4
>>> c.send(5)
6
>>> c.send(10)
11
>>> c.send(100)
101
```

Send(None)就等价于 Next, 就是直接恢复执行，不向其传递信息

```
>>> c = number(3)
>>> c.send(None)
4
```

我们可以构建一个操作系统toy了！

- 我们实现操作系统玩具 (os-model.py)
  - 关键点1：利用系统构建的数据结构Thread来支撑目标程序运行

```
class Thread:
    """A "freezed" thread state."""
    def __init__(self, func, *args):
        self._func = func(*args)
        self.retval = None

    def step(self):
        """Proceed with the thread until its next trap."""
        syscall, args, *_ = self._func.send(self.retval)
        self.retval = None
        return syscall, args
```

根据目标程序创建生成器

运行该程序直到下个内核陷入

我们可以构建一个操作系统toy了！

- 我们实现操作系统玩具 (os-model.py)
  - 关键点2：操作系统就初始化之后就变成了“事件”的处理者（这里简化为syscall的解释器）

```
def run(self):
    threads = [OperatingSystem.Thread(self._main)]
    while threads:  # Any thread lives
        try:
            match (t := threads[0]).step():
                case 'choose', xs:  # Return a random choice
                    t.retval = random.choice(xs)
                case 'write', xs:  # Write to debug console
                    print(xs, end='')
                case 'spawn', (fn, args):  # Spawn a new thread
                    threads += [OperatingSystem.Thread(fn, *args)]
                case 'sched', _:  # Non-deterministic schedule
                    random.shuffle(threads)
        except StopIteration:  # A thread terminates
            threads.remove(t)
            random.shuffle(threads)  # sys_sched()
```

系统初始化工作！

interrupt/trap/fault的handler!

*当然我们可以有更完整的Toy

- 一个更完整的系统玩具 (mosaic.py)

mosaic系统调用	Linux对应	作用
sys_spawn(fn)	pthread_create	创建从 fn 开始执行的线程
sys_fork()	fork	创建当前状态机的完整复制
sys_sched()	调度器	切换到随机的线程/进程执行
sys_choose(xs)	rand	返回一个 xs 中的随机的选择
sys_write(s)	printf	向调试终端输出字符串 s
sys_bread(k)	read	读取虚拟设磁盘块k的数据
sys_bwrite(k, v)	write	向虚拟设磁盘块k的写入数据v
sys_sync()	sync	将所有向虚拟磁盘的数据写入落盘

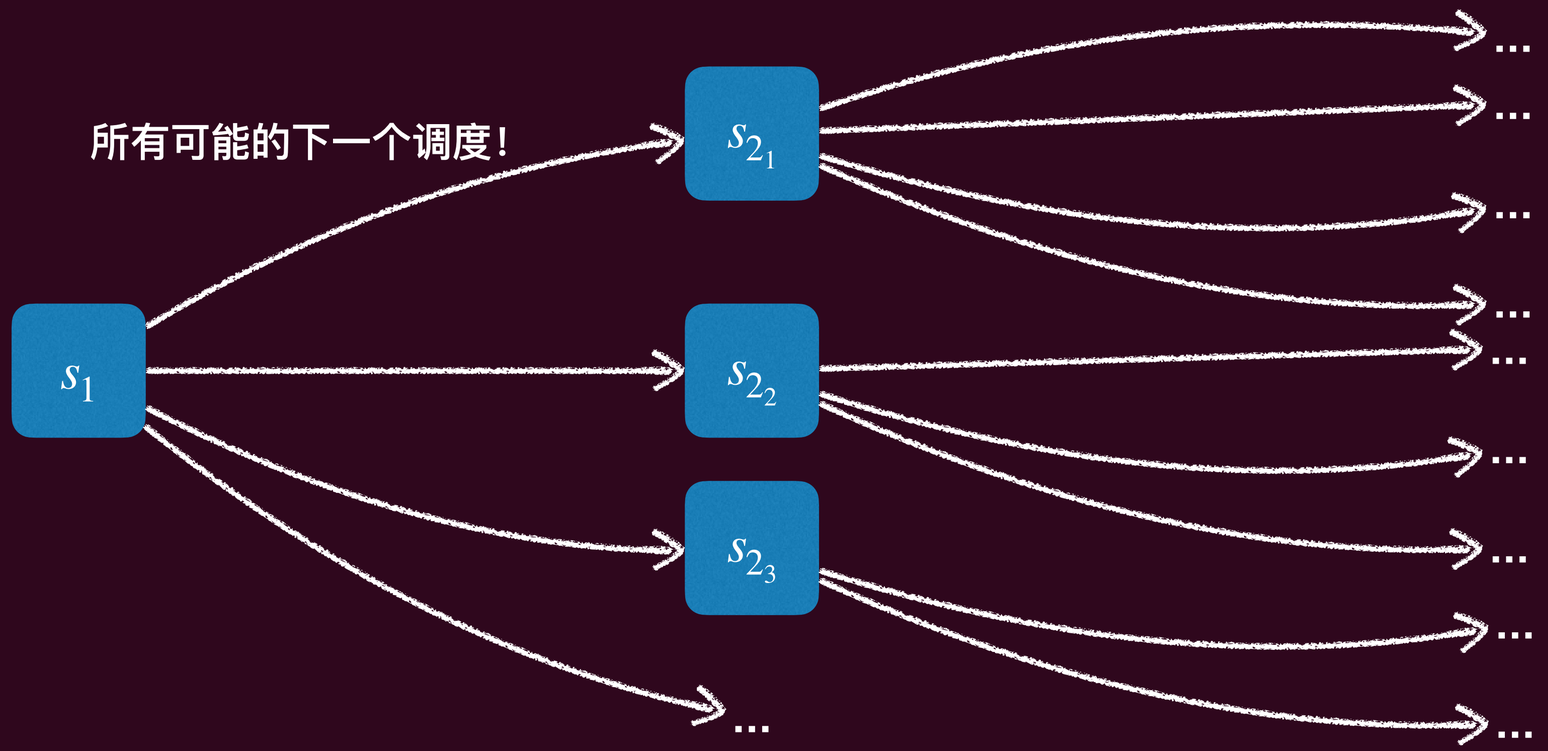
*当然我们可以有更完整的Toy

- 原理和之前的相似
  - 进程/线程都是生成器对象
  - 有了共享内存（线程），进程是独立的heap clone
- 但还是简化模型
  - 实际系统中程序随时都可能被“外部事件”打断（比如时间中断）
  - 磁盘就是一个dict

简化的玩具让我们对操作系统又了一个宏观的感受！当然之后我们会回归细节

*当然我们可以有更完整的Toy

- mosaic操作系统的一个彩蛋：
  - 当执行*sys_sched()*，其可以枚举所有可能的调度序列 （这样才能证明所有可能的状态下程序是正确的）



另一个角度：Nondeterministic Turing Machine！

总结

- 本身是状态机(内部的状态为操作系统管理的资源状态，被动进行状态迁移)
- 为在其之上的应用程序服务：应用程序的解释器
- 直接跑在硬件之上的C程序：为硬件中断处理程序

阅读材料

- [CSAPP] 第1，7，8章
- [OSPP] 第1，2章
- 浏览 GNU Coreutils 和 GNU Binutils 的网站，建立 “手边有哪些可用的命令行工具” 的一些印象。
- 浏览 gdb 文档的目录，找到你感兴趣的章节了解，例如——“Reverse Execution”、“TUI: GDB Text User Interface”……
- 对于你有兴趣的命令行工具，可以参考 busybox 和 toybox 项目中与之对应的简化实现。