

# 第4讲 Python建模操作系统

2024.02.28

## 1. OS?

- 从应用程序角度看，就是一条syscall（当然syscall的行为很丰富）的api，应用系统只要调用这个api，就为它提供服务
- 从硬件角度看，就是个c程序，能从cpu reset开始，运行这个状态机，就可以实现把自己的代码加载到计算机中执行

### 回顾：程序/硬件的状态机模型

---

#### 计算机软件

- 状态机 (C/汇编)
  - 允许执行特殊指令 (syscall) 请求操作系统
  - 操作系统 = API + 对象

#### 计算机硬件

- “无情执行指令的机器”
  - 从 CPU Reset 状态开始执行 Firmware 代码
  - 操作系统 = C 程序

从上往下看，操作系统里面有很多对象，比如进程、文件：

- p1,p2: 进程，每一个进程都有很多api，可以访问这些对象
- a.out、a.txt...

## 一个大胆的想法

### 一个大胆的想法

---

无论是软件还是硬件，都是状态机

- 而状态和状态的迁移是可以“画”出来的！
- 理论上说，只需要两个 API
  - dump\_state() - 获取当前程序状态
  - single\_step() - 执行一步
  - gdb 不就是做这个的吗！

我们都说计算机系统是状态机了，而状态和状态的迁移是可以画出来的！

简化我们的模型：

- 在操作系统课上，简化是非常重要的主题（但是simple不代表过度的简化，比如只剩下一个概念，用来考试）
- 否则很容易迷失在细节的海洋中

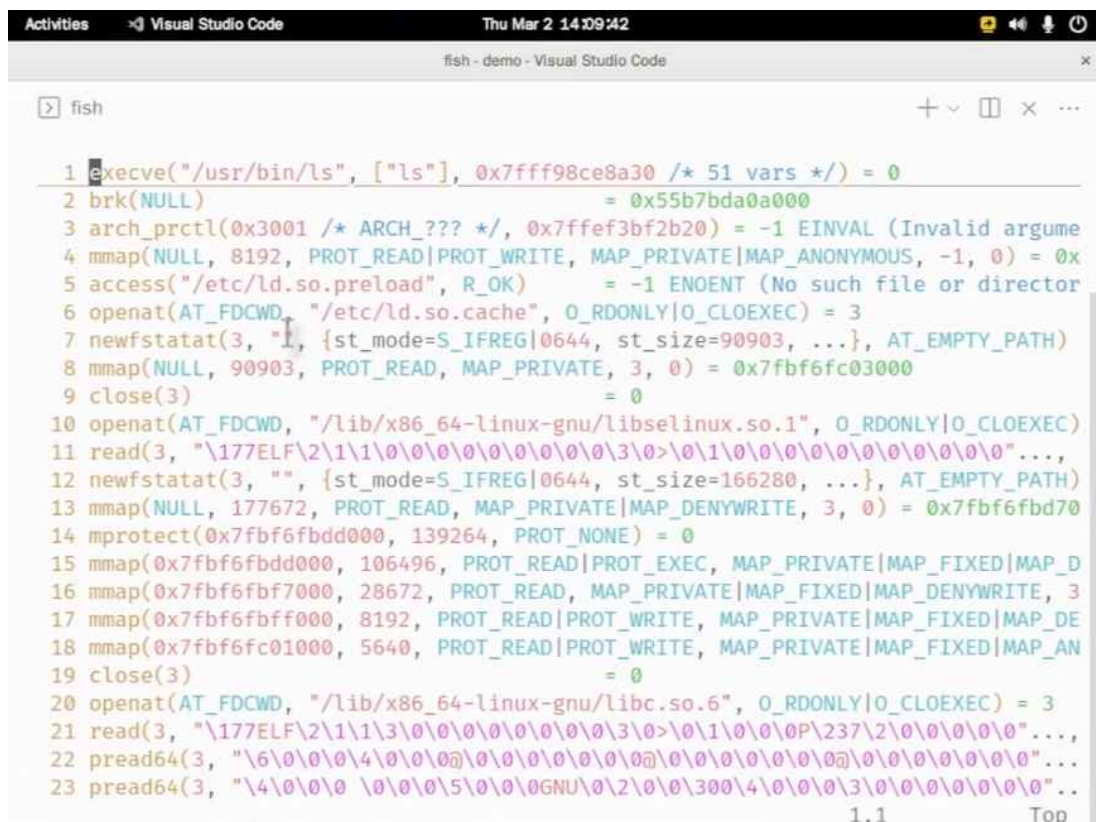
## GDB 检查状态的缺陷

太复杂

- 状态太多 (指令数很多)
- 状态太大 (很多库函数状态)

简化：把复杂的东西分解成简单的东西

- 在《操作系统》课上，简化是非常重要的主题
  - 否则容易迷失在细节的海洋中
  - 一些具体的例子
    - 只关注系统调用 (strace)
    - Makefile 的命令日志
    - strace/Makefile 日志的清理
- strace:
  - 做了一个程序执行流的纵向的简化，我们把指令分成两种：计算用的指令，把寄存器的值拿出来，算一算丢到内存或者寄存器，strace不care这种连续的计算指令，只关心系统调用。
  - 这个纵向的简化使得你能够看到软件和操作系统的边界上发生了什么？



```
1 execve("/usr/bin/ls", ["ls"], 0x7fff98ce8a30 /* 51 vars */) = 0
2 brk(NULL) = 0x55b7bda0a000
3 arch_prctl(0x3001 /* ARCH_??? */, 0x7ffef3bf2b20) = -1 EINVAL (Invalid argume
4 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x
5 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or director
6 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=90903, ...}, AT_EMPTY_PATH)
8 mmap(NULL, 90903, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fbf6fc03000
9 close(3) = 0
10 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC)
11 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0\0"...,
12 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=166280, ...}, AT_EMPTY_PATH)
13 mmap(NULL, 177672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fbf6fbd70
14 mprotect(0x7fbf6fbd7000, 139264, PROT_NONE) = 0
15 mmap(0x7fbf6fbd7000, 106496, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_D
16 mmap(0x7fbf6fbd7000, 28672, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3
17 mmap(0x7fbf6fbd7000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DE
18 mmap(0x7fbf6fbd7000, 5640, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_AN
19 close(3) = 0
20 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
21 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0\0"...,
22 pread64(3, "\6\0\0\0\4\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,
23 pread64(3, "\4\0\0\0\0\0\0\0\5\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...
```

大家都是状态机

## 编译器在做什么？

- c程序是有自己的行为的定义的，就是每次执行一个语句，从c语言的一个状态到下一个状态（当然我们这里指的是普通的计算），如果c语言我们禁止你调用任何的库函数，禁止你写汇编，不向外部调用，你甚至都不能终止（除非你程序bug爆了），那么必然是有向外部的系统调用的。
- 所以——不管是高级语言，还是高级语言翻译而成的汇编语言，都是状态机。（刚才我们strace的故事讲过了，我们只关注系统调用）我们认为汇编语言的状态机太复杂，我们就看看c语言的状态机？如果c语言还是太恶心，我们看看.....python？（众所周知，c是高级的汇编——你看着源代码（不优化），你就直接基本上能肉眼翻译成汇编了！哪怕是cpp，模板实例化就足够令你发疯了）

## 一个想法：反正都是状态机.....

---

我们真正关心的概念

- 应用程序 (高级语言状态机)
- 系统调用 (操作系统 API)
- 操作系统内部实现

没有人规定上面三者如何实现

- 通常的思路：真实的操作系统 + QEMU/NEMU 模拟器
- 我们的思路
  - 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

```
def main():  
    sys_write('Hello, OS World')
```

## 2. 操作系统玩具：设计与实现

### 操作系统玩具：API

---

四个“系统调用”API

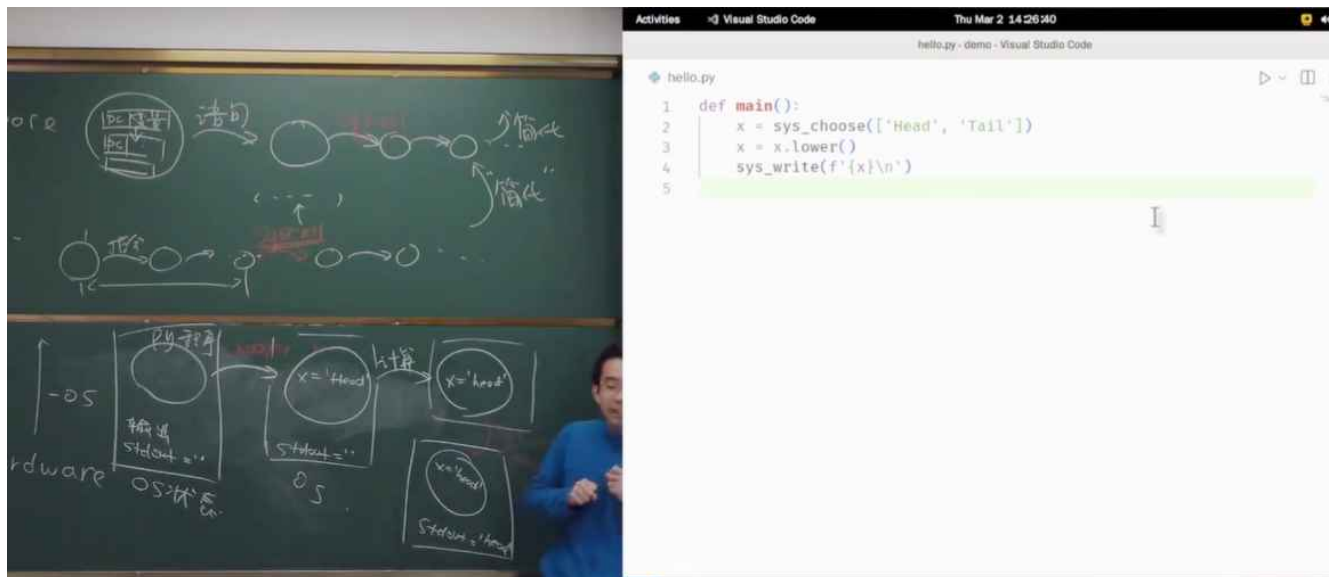
- choose(xs): 返回 xs 中的一个随机选项
- write(s): 输出字符串 s
- spawn(fn): 创建一个可运行的状态机 fn
- sched(): 随机切换到任意状态机执行

除此之外，所有的代码都是确定 (deterministic) 的纯粹计算

- 允许使用 list, dict 等数据结构

- **choose**：如果你只能做真正纯粹的计算，你要想随机数，必须是要外界来帮你的（比如os）。当然现在x86硬件能够提供实现随机数的功能了，这是后话。

## 操作系统是什么？



我们执行的时候，圆圈里面就是我们的程序；当我们执行到sys\_write的时候，我们发现：（从程序的视角）状态似乎没有改变啊！这个时候我们要关注的就是os本身——屏幕上也输出了！os单独我们也可以拉出来认为是一个状态机（操作系统的状态，一个更大的状态机）。

同时，操作系统里面有很多很多程序！所以操作系统可以认为是状态机的管理者！

- **spawn**：创建一个新的状态机出来。我们可以给它传一个函数，可以给这个函数传一个参数。  
**spawn** 的行为是？（感觉你在创建一个进程/线程对不对？）当前的状态机的pc动了，同时在操作系统的世界里，多出来了一个状态机！
  - 哦！这就是今天的操作系统！



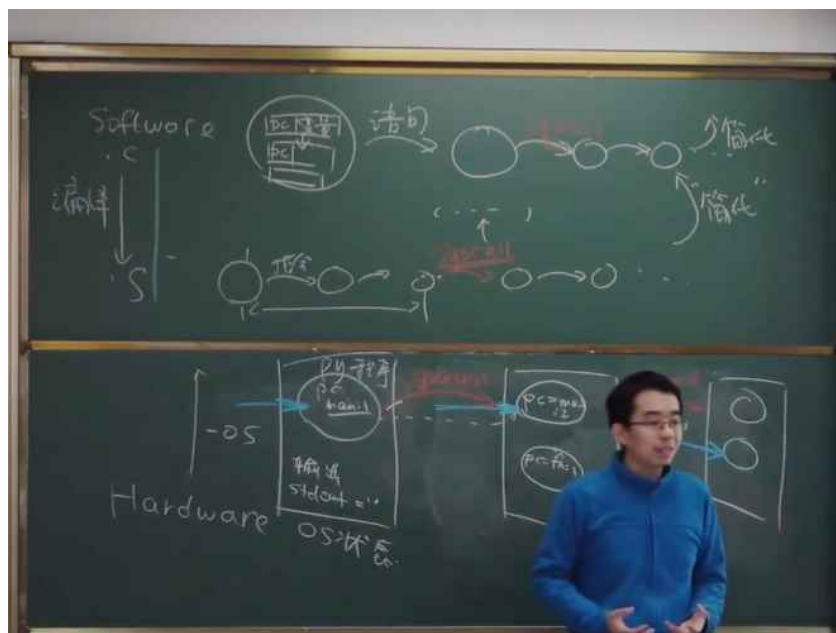
再回顾一下：

- 我们今天的操作系统的概念模型：程序是状态机；操作系统为我们的程序提供了api和对象。
- 当我们一开始讲操作系统的对象的时候，大家会想：哦，文件是对象。想要读写文件需要一个指针（文件描述符），它其实就是指向文件对象的指针。
- 还有一种非常重要的对象就是进程和线程（执行的状态机）。我们的toy里面就体现了这一点。

现在就有意思了：现在有两个状态机了，现在该谁执行了？？？

我们的玩具是怎么办的？

- `sched`：我们有一个蓝色的指针指向当前的状态机，不管我们怎么执行系统调用，这个蓝色的指针都是不变的；但是如果我们执行 `schedule` 调度，蓝色的指针就随机漂移一波。



## 操作系统玩具：API

四个“系统调用”API

- `choose(xs)`: 返回 `xs` 中的一个随机选项
- `write(s)`: 输出字符串 `s`
- `spawn(fn)`: 创建一个可运行的状态机 `fn`
- `sched()`: 随机切换到任意状态机执行

除此之外，所有的代码都是确定 (deterministic)

- 允许使用 `list`, `dict` 等数据结构

## 应用程序



# 操作系统玩具：应用程序

操作系统玩具：我们可以动手把状态机画出来！

```
count = 0

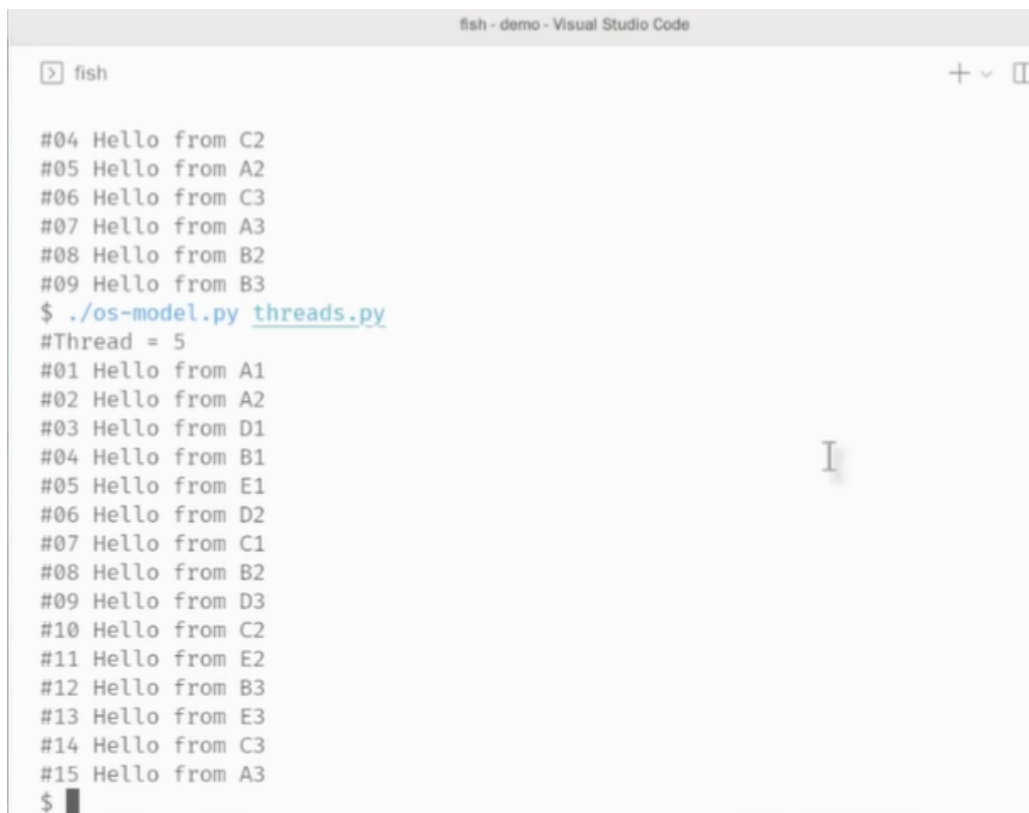
def Tprint(name):
    global count
    for i in range(3):
        count += 1
        sys_write(f'#{count:02} Hello from {name}{i+1}\n')
        sys_sched()

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()
```

每个状态机执行的时候能共享这个count，所以严格来说它是线程。

进程和线程之间的区别是？

- 线程是可以共享内存的；进程是独立内存的。
- 比如你的安卓手机应用能随便访问另一个应用的数据，就很危险了。



```
fish - demo - Visual Studio Code

fish

#04 Hello from C2
#05 Hello from A2
#06 Hello from C3
#07 Hello from A3
#08 Hello from B2
#09 Hello from B3
$ ./os-model.py threads.py
#Thread = 5
#01 Hello from A1
#02 Hello from A2
#03 Hello from D1
#04 Hello from B1
#05 Hello from E1
#06 Hello from D2
#07 Hello from C1
#08 Hello from B2
#09 Hello from D3
#10 Hello from C2
#11 Hello from E2
#12 Hello from B3
#13 Hello from E3
#14 Hello from C3
#15 Hello from A3
$
```

## 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)
```

有些就困难了

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

我们需要

- 封存当前状态机的状态
- 恢复另一个“被封存”状态机的执行
  - 没错，我们离真正的“分时操作系统”就只差这一步

当我们参与项目的时候，写一个 `raise NotImplementedError` 是非常好的practice。

你搭了一个很大的框架，比如你类的设计已经做好了，肯定一开始就是到处写TODO，如果你在犄角旮旯里面return了一个奇怪的返回值比如0/null，有可能就会造成误解。

封存当前状态机的状态：

- 在我们的程序中，调度的时候，我们要封存当前所有的状态（比如这个 `i` 不能没了）。

## 借用 Python 的语言机制

Generator objects (无栈协程/轻量级线程/...)

```
def numbers():
    i = 0
    while True:
        ret = yield f'{i:b}' # “封存”状态机状态
        i += ret
```

使用方法：

```
n = numbers() # 封存状态机初始状态
n.send(None) # 恢复封存的状态
n.send(0) # 恢复封存的状态（并传入返回值）
```

完美适合我们实现操作系统玩具 (os-model.py)

## Python: yield

生成器：

生成器的发明使得python模仿协同程序的概念得以实现。

- 协同程序：可以运行的独立函数调用，函数可以暂停或者挂起，并且在需要的时候从离开的地方继续或者重新开始。对于调用的普通python函数，一般是从函数的第一行开始执行，执行到return或者异常或者执行完毕所有代码，将控制权交还给调用者。这就意味着全部结束了，此时**函数所做的所有工作和保存至局部变量当中的数据都将丢失，再次调用函数时，一切从头开始。**
- 生成器可以暂时挂起函数，并且保留函数的局部变量等数据，再次调用函数的时候，从上次暂停的位置继续运行下去。

原文链接：<https://blog.csdn.net/mieleizhi0522/article/details/82142856>

首先，如果你还没有对yield有个初步认识，那么你先把yield看做“return”，这个是直观的，它首先是个return，普通的return是什么意思，就是在程序中返回某个值，返回之后程序就不再往下运行了。看做return之后再把它看做一个是生成器（generator）的一部分（带yield的函数才是真正的迭代器），好了，如果你对这些不明白的话，那先把yield看做return,然后直接看下面的程序，你就会明白yield的全部意思了：

```
1 def foo():
2     print("starting...")
3     while True:
4         res = yield 4
5         print("res:",res)
6 g = foo()
7 print(next(g))
8 print("*"*20)
9 print(next(g))
```

就这么简单的几行代码就让你明白什么是yield，代码的输出这个：

```
1 starting...
2 4
3 *****
4 res: None
5 4
```

我直接解释代码运行顺序，相当于代码单步调试：

1. 程序开始执行以后，因为foo函数中有yield关键字，所以foo函数并不会真的执行，而是先得得到一个生成器g(相当于一个对象)



2. 直到调用 `next` 方法，`foo` 函数正式开始执行，先执行 `foo` 函数中的 `print` 方法，然后进入 `while` 循环
3. 程序遇到 `yield` 关键字，然后把 `yield` 想想成 `return`，`return` 了一个4之后，程序停止，并没有执行赋值给 `res` 操作，此时 `next(g)` 语句执行完成，所以输出的前两行（第一个是 `while` 上面的 `print` 的结果,第二个是 `return` 出的结果）是执行 `print(next(g))` 的结果
4. 程序执行 `print("*"20)`，输出20个 `*`
5. 又开始执行下面的 `print(next(g))`，这个时候和上面那个差不多，不过不同的是，这个时候是从刚才那个 `next` 程序停止的地方开始执行的，也就是要执行 `res` 的赋值操作，这时候要注意，这个时候赋值操作的右边是没有值的（因为刚才那个是 `return` 出去了，并没有给赋值操作的左边传参数），所以这个时候 `res` 赋值是 `None`，所以接着下面的输出就是 `res:None`，
6. 程序会继续在 `while` 里执行，又一次碰到 `yield`，这个时候同样 `return` 出4，然后程序停止，`print` 函数输出的4就是这次 `return` 出的4.

到这里你可能就明白 `yield` 和 `return` 的关系和区别了，带 `yield` 的函数是一个生成器，而不是一个函数了，这个生成器有一个函数就是 `next` 函数，`next` 就相当于“下一步”生成哪个数，这一次的 `next` 开始的地方是接着上一次的 `next` 停止的地方执行的，所以调用 `next` 的时候，生成器并不会从 `foo` 函数的开始执行，只是接着上一步停止的地方开始，然后遇到 `yield` 后，`return` 出要生成的数，此步就结束。

```
1 def foo():
2     print("starting...")
3     while True:
4         res = yield 4
5         print("res:",res)
6 g = foo()
7 print(next(g))
8 print("*"*20)
9 print(g.send(7))
```

再看一个这个生成器的 `send` 函数的例子，这个例子就把上面那个例子的最后一行换掉了，输出结果：

```
1 starting...
2 4
3 *****
4 res: 7
5 4
```

先大致说一下 `send` 函数的概念：此时你应该注意到上面那个紫色的字，还有上面那个 `res` 的值为什么是 `None`，这个变成了7，到底为什么，这是因为，`send` 是发送一个参数给 `res` 的，因为上面讲到，`return` 的时候，并没有把4赋值给 `res`，下次执行的时候只好继续执行赋值操作，只好赋值为 `None` 了，而如果用 `send` 的话，开始执行的时候，先接着上一次（`return 4` 之后）执行，先把7赋值给了 `res`，然后执行 `next` 的作用，遇见下一回的 `yield`，`return` 出结果后结束。

1. 程序执行 `g.send(7)`，程序会从 `yield` 关键字那一行继续向下运行，`send` 会把7这个值赋值给 `res` 变量
2. 由于 `send` 方法中包含 `next()` 方法，所以程序会继续向下运行执行 `print` 方法，然后再次进入 `while` 循环
3. 程序执行再次遇到 `yield` 关键字，`yield` 会返回后面的值后，程序再次暂停，直到再次调用 `next` 方法或 `send` 方法。

借助操作系统里面的知识理解编程语言：



```
python3
#Thread = 3
#01 Hello from C1
#02 Hello from B1
#03 Hello from A1
#04 Hello from A2
#05 Hello from C2
#06 Hello from A3
#07 Hello from C3
#08 Hello from B2
#09 Hello from B3
$ python3
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def numbers():
...     i = 0
...     while True:
...         ret = yield f'{i:b}' # "封存" 状态机状态
...         i += ret
...
>>>
>>> n = numbers()
>>> type(n)
<class 'generator'>
>>>
```

- 这里看起来是函数调用，其实是一个生成器。现在generator其实就是一个状态机：
  - 两种操作：第一种是纯粹的计算，`i = 0`，`i++`，还有一种是系统调用，就是这个 `yield`，可以把这个状态机里面的信息暴露到外面，然后把控制流交给外面，外面也可以把信息传回来。

## Show me the f\*\*king code

```
1 #!/usr/bin/env python3
2
```

```

3 import sys
4 import random
5 from pathlib import Path
6
7 class OperatingSystem():
8     """A minimal executable operating system model."""
9
10     SYSCALLS = ['choose', 'write', 'spawn', 'sched']
11
12     class Thread:
13         """A "freezed" thread state."""
14
15         def __init__(self, func, *args):
16             self._func = func(*args)
17             self.retval = None
18
19         def step(self):
20             """Proceed with the thread until its next trap."""
21             syscall, args, *_ = self._func.send(self.retval)
22             self.retval = None
23             return syscall, args
24
25         def __init__(self, src):
26             variables = {}
27             exec(src, variables)
28             self._main = variables['main']
29
30         def run(self):
31             """
32             操作系统是主循环，如果现在还有可以执行的状态机，我们每次选择一个状态机，把控制流给
33             它，直到它给我一个yield，把控制流还给我；
34             在现在真实的os中，除了应用程序可以主动将控制权交还，硬件中断（时钟的定时中断）也
35             会强行夺回控制权，这就奠定了os的霸主地位。
36             """
37             threads = [OperatingSystem.Thread(self._main)]
38             while threads: # Any thread lives
39                 try:
40                     match (t := threads[0]).step():
41                         case 'choose', xs: # Return a random choice
42                             t.retval = random.choice(xs)
43                         case 'write', xs: # Write to debug console
44                             print(xs, end='')
45                         case 'spawn', (fn, args): # Spawn a new thread
46                             threads += [OperatingSystem.Thread(fn, *args)]
47                         case 'sched', _: # Non-deterministic schedule
48                             random.shuffle(threads)
49                     except StopIteration: # A thread terminates

```

```

48         threads.remove(t)
49         random.shuffle(threads) # sys_sched()
50
51 if __name__ == '__main__':
52     if len(sys.argv) < 2:
53         print(f'Usage: {sys.argv[0]} file')
54         exit(1)
55
56     src = Path(sys.argv[1]).read_text()
57     for syscall in OperatingSystem.SYSCALLS:
58         src = src.replace(f'sys_{syscall}',          # sys_write(...)
59                           f'yield "{syscall}"', ' ') # -> yield 'write', (...)
60
61     OperatingSystem(src).run()

```

## 玩具的意义

我们并没有脱离真实的操作系统

- “简化”了操作系统的 API
  - 在暂时不要过度关注细节的时候理解操作系统
- 细节也会有的，但不是现在
  - 学习路线：先 100% 理解玩具，再理解真实系统和玩具的差异

```

void sys_write(const char *s) { printf("%s", s); }
void sys_sched() { usleep(rand() % 10000); }
int sys_choose(int x) { return rand() % x; }

void sys_spawn(void (*fn)(void *), void *args) {
    pthread_create(&threads[nthreads++], NULL, fn, args);
}

```

## 3. 一个更大的玩具

在我们的操作系统模型中，应用程序 (状态机) 被分为两部分：确定性 (deterministic) 的本地计算，和可能产生非确定性的系统调用 (以 sys 开头的函数)。操作系统提供以下 API：

## 一个更“全面”的操作系统模型

进程 + 线程 + 终端 + 存储 (崩溃一致性)

系统调用/Linux 对应	行为
<code>sys_spawn(fn)/pthread_create</code>	创建从 <code>fn</code> 开始执行的线程
<code>sys_fork()/fork</code>	创建当前状态机的完整复制
<code>sys_sched()/</code> 定时被动调用	切换到随机的线程/进程执行
<code>sys_choose(xs)/rand</code>	返回一个 <code>xs</code> 中的随机的选择
<code>sys_write(s)/printf</code>	向调试终端输出字符串 <code>s</code>
<code>sys_bread(k)/read</code>	读取虚拟磁盘块 <code>k</code> 的数据
<code>sys_bwrite(k, v)/write</code>	向虚拟磁盘块 <code>k</code> 写入数据 <code>v</code>
<code>sys_sync()/sync</code>	将所有向虚拟磁盘的数据写入落盘
<code>sys_crash()/</code> 长按电源按键	模拟系统崩溃

- 进程不共享内存；共享内存的好处就是一个人算出来的东西马上人家就能用了。

我们有一个假想的磁盘，它是按块来读写的，比如说第一块的数据是什么？第二块的数据是什么？第三块的数据是什么？那我就可以读一块，写一块。但是更有意思的是我们今天的磁盘和我们的共享内存也有点像，这也是后面我会讲到的。

今天的磁盘你说你写了不代表他就真的写进去了，你们有没有遇到过这样的情况？就是你的 u 盘，如果你在写的时候，比如说你刚你文件已经拷完了，比如你保存了，关掉了，然后你把 u 盘拔掉，windows 是不是给你一个警告，对吧？告诉你不要这样做。然后以及你插上这个盘以后，他可能会说这个盘可能存在问题，问你是不是要修复？我不知道你们有没有因为这样的差和拔丢过数据，今天的 Windows 对 u 盘的写策略是比较保守的，如果你 u 盘比较先进，也有可能会有这样的问题。

你们现在的闪存的速度，尤其是 u 盘上的闪存的速度，可能是跟不上你的这个系统的写入的速度，所以你的 u 盘上很有可能有一块小的内存，这个内存很快。然后你的所有的写的数据都写到你的 u 盘里，然后 u 盘我收到了，因为一旦 u 盘和你的计算机确认说我收到了，你就可以写下一个给他，所以你就看起来很快，你们应该观察过这个现象。你的 u 盘可能在前几秒钟可以写的速度很快，然后就对这个现象大家是见过的，对吧？就在前几秒钟你会看我很开心这个一分钟就写完，然后突然哇那个曲线就过下来了。这就是因为我们的磁盘的 u 盘里面是有一个缓存的，而这个缓存就意味着如果你在这个时候把 u 盘从电脑上拔掉，或者这个电脑发生了断电，你的数据就不完全不是按照你想的方式写了。

## 总结

### Take-away Messages

我们可以用“简化”的方式把操作系统的概念用可执行模型的方式呈现出来：

- 程序被建模为高级语言 (Python) 的执行和系统调用
- 系统调用的实现未必一定需要基于真实或模拟的计算机硬件

- 操作系统的“行为模型”更容易理解