Python 建模操作系统

理解了“软件（应用）”和“硬件（计算机）”之后，操作系统就是直接运行在计算机硬件上的程序，它提供了应用程序执行的支撑和一组 API（系统调用）：

操作系统内核被加载后，拥有完整计算机的控制权限，包括中断和 I/O 设备，因此可以构造出多个应用程序同时执行的“假象”。

目录

- 理解操作系统的新途径
- 使用Python的建模

理解操作系统的新途径

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

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')
```

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。

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()
```

🌶️ 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：

```python
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)

> 作业 了解yield

```python
#!/usr/bin/env python3

import sys
import random
from pathlib import Path

class OperatingSystem():
    """A minimal executable operating system model."""

    SYSCALLS = ['choose', 'write', 'spawn', 'sched']

    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

    def __init__(self, src):
        variables = {}
        exec(src, variables)
        self._main = variables['main']

    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()

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print(f'Usage: {sys.argv[0]} file')
        exit(1)

    src = Path(sys.argv[1]).read_text()
    for syscall in OperatingSystem.SYSCALLS:
        src = src.replace(f'sys_{syscall}',        # sys_write(...)
                          f'yield "{syscall}", ')  #  -> yield 'write', (...)

    OperatingSystem(src).run()
```

建模OS

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
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()/定时被动调用	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	向虚拟磁盘块 $k 写入数据 $v
sys_sync()/sync	将所有向虚拟磁盘的数据写入落盘
sys_crash()/长按电源按键	模拟系统崩溃

使用Python建模做出的简化

模型做出的简化

被动进程/线程切换

- 实际程序随时都可能被动调用 `sys_sched()` 切换

只有一个终端

- 没有 `read()`（用 choose 替代“允许读到任意值”）

磁盘是一个 `dict`

- 把任意 key 映射到任意 value
- 实际的磁盘
  - key 为整数
  - value 是固定大小（例如 4KB）的数据
  - 二者在某种程度上是可以“互相转换”的

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

课后习题/编程作业

1．编程实践

阅读、调试 `os-model.py` 的代码，观察如何使用 `generator` 实现在状态机之间的切换。在现代分时操作系统中，状态机的隔离（通过虚拟存储系统）和切换是一项基础性的基础，也是操作系统最有趣的一小部分代码：在中断或是 `trap` 指令后，通常由一段汇编代码将当前状态机（执行流）的寄存器保存到内存中，完成状态的“封存”。

2．实验作业

开始课程实验：课程实验在在课程网站上发布。实验有一定难度，同时也有实验指导，请大家仔细阅读。