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！

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

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”：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

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()
```

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

🌶️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)

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

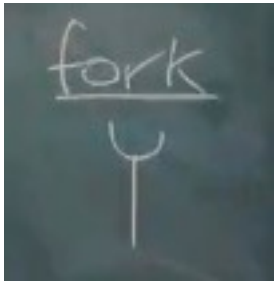
```
void sys_write(const char *s) { printf("%s", s); }
void sys_sched() { usleep(rand() % 10000); }
int sys_choose(int x) { return rand() % x; }

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

建模操作系统

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



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)。

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

模型实现

原理与刚才的 “最小操作系统玩具” 类似

- **mosaic.py** - 500 行建模操作系统
- 进程/线程都是 Generator Object
- 共享内存用 heap 变量访问
  - 线程会得到共享 heap 的指针
  - 进程会得到一个独立的 heap clone

输出程序运行的 “状态图”

- JSON Object
- Vertices: 线程/进程、内存快照、设备历史输出
- Edges: 系统调用
  - 操作系统就是 “状态机的管理者”

建模的意义

我们可以把状态机的执行画出来了！

- 可以直观地理解程序执行的全流程
- 可以对照程序在真实操作系统上的运行结果

这对于更复杂的程序来说是十分关键的

```
void Tsum() {
  for (int i = 0; i < n; i++) {
    int tmp = sum;
    tmp++;
    // 假设此时可能发生进程/线程切换
    sum = tmp;
  }
}
```