

第15讲 操作系统上的进程

2024.04.15

今天我们正式结束的一部分的内容，进入第二部分虚拟化。

背景回顾：有关状态机、并发和中断的讨论给我们真正理解操作系统奠定了基础，现在我们正式进入操作系统和应用程序的“边界”了。让我们把视角回到单线程应用程序，即“执行计算指令和系统调用指令的状态机”，开始对操作系统和进程的讨论。

1. 进程、线程和操作系统

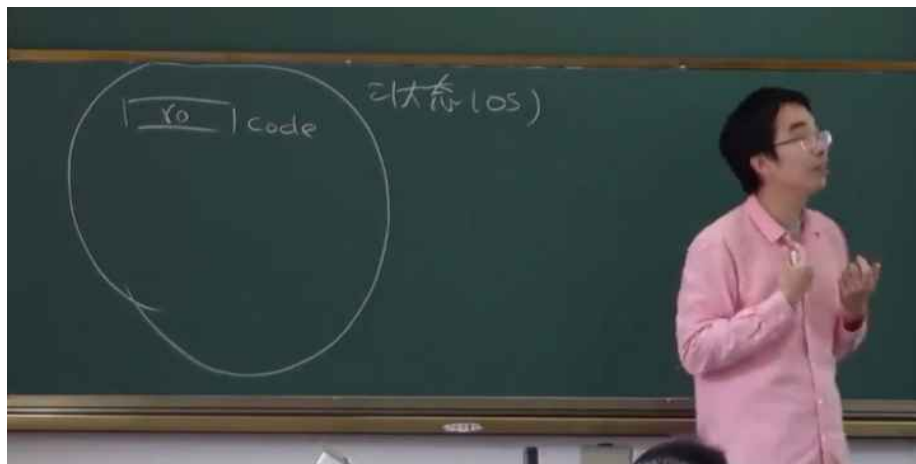
1. Review

ThreadOS 中的线程切换

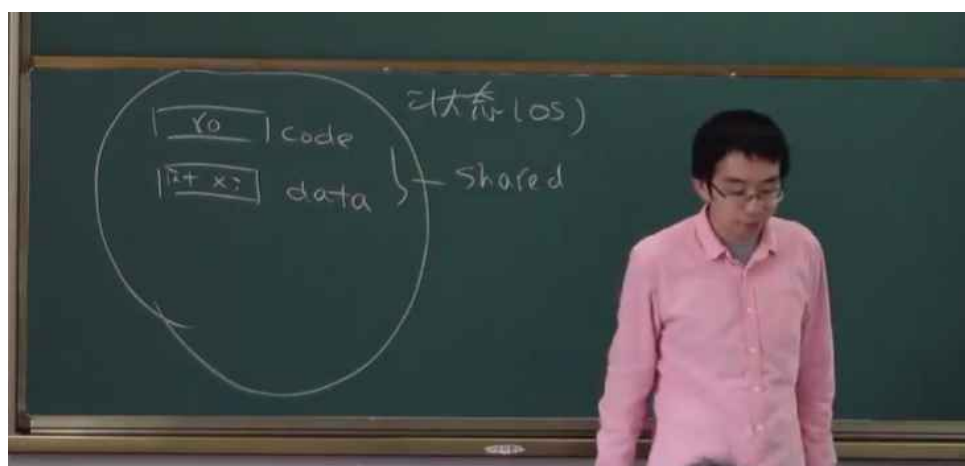
计算机硬件也是状态机

- “共享的内存，容纳了多个状态机”
 - 共享的代码
 - 共享的全局变量
 - 启动代码的堆栈 (仅启动代码可用)
 - T_1 的 Task (仅 T_1 和中断处理程序可用)
 - 堆栈 (中断时，寄存器保存在堆栈上)
 - T_2 的 Task (仅 T_2 和中断处理程序可用)
 - 堆栈 (中断时，寄存器保存在堆栈上)
- 状态迁移
 - 执行指令或响应中断

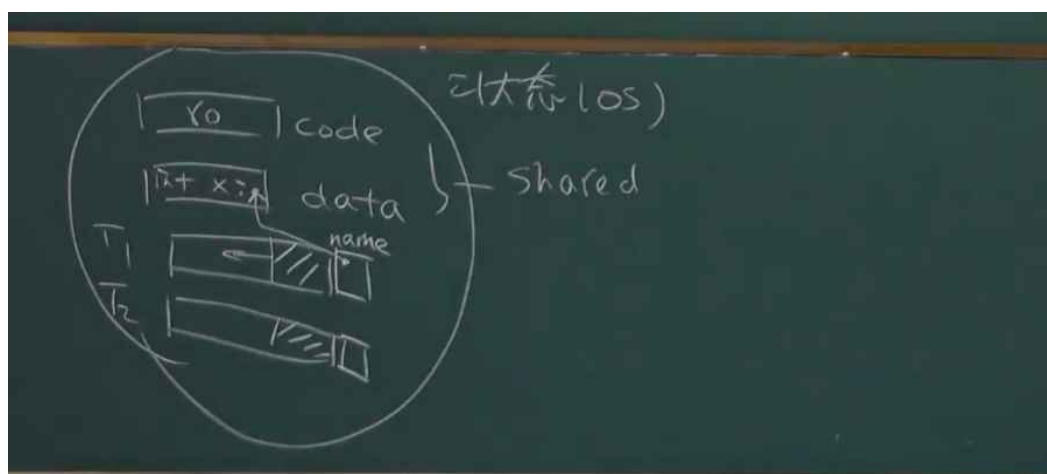
虽然内存的地址空间在硬件上看它是这个平的，但在逻辑上其实它还是分成几个区域的，比如说实际上你是有代码的，所有的汇编指令编译出来的指令都在共享的代码区，那我们一般来讲是假设这个代码是只读的，如果你不小心把这个代码给覆盖了，那很有可能你的程序就出现奇奇怪怪的问题。然后你可能会看到奇怪的字符被打印出来，或者是你的虚拟机在神秘的重启。



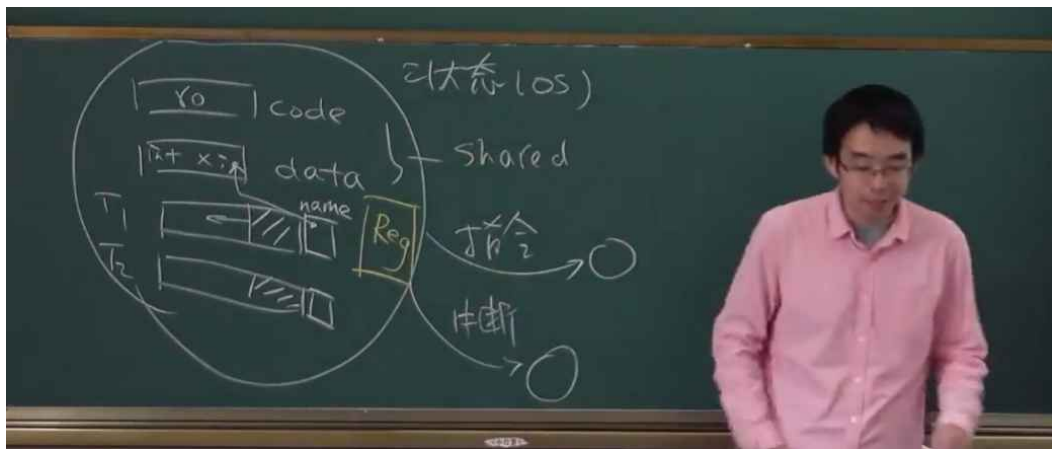
然后是一些共享的全局变量：



还有一些线程：它们就会有自己的线程的状态——堆栈、自己的线程的信息。比如说有名字，它有一个 name，那 name 是一个指针，然后这个指针就应该是指向 data 或者一个 code，或者 read only data 的某一个位置，然后我就可以把这个线程的名字给读出来。



以上是我们计算机系统内存里面很重要的状态。当然还有一个很重要的组成部分——寄存器。



然后指令和中断帮助这个状态机做状态的转移。

2. 什么是操作系统

我们来看一下我们线程操作系统的模型：

- 大家share一份memory
- 每个thread有自己的stack和寄存器



下一个问题：进程是什么？

```

1  #include <sys/syscall.h>
2
3  .globl _start
4  _start:
5      movq $SYS_write, %rax    // write(
6      movq $1, %rdi           // fd=1,
7      movq $st, %rsi          // buf=st,
8      movq $(ed - st), %rdx    // count=ed-st
9      syscall                 // );
10
11     movq $SYS_exit, %rax     // exit(
12     movq $1, %rdi           // status=1
13     syscall                 // );
14
15     st:
16     .ascii "\033[01;31mHello, OS World\033[0m\n"
17     ed:
18

```

这就是进程。

进程就是一个状态机，然后这个状态机有一个初始状态，每一次执行指令，做状态切换。

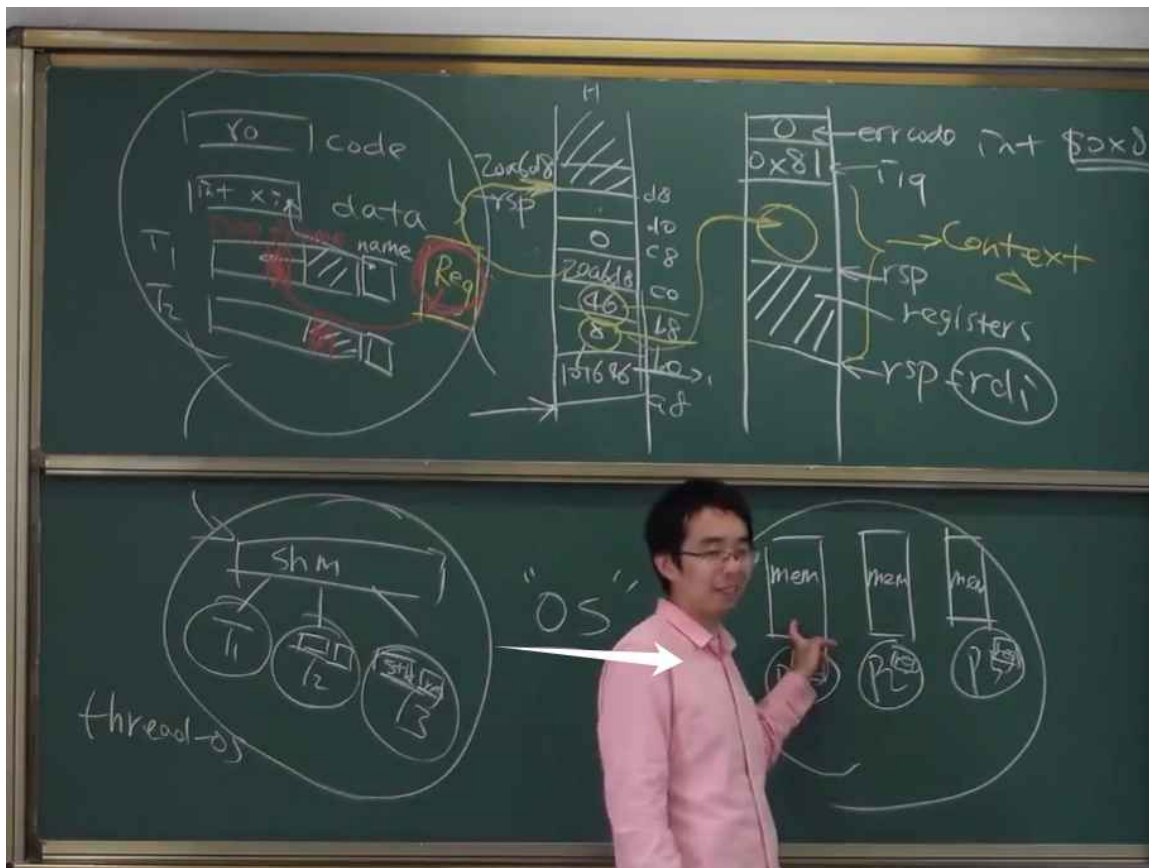
好，那么问题来了，我们的进程模型，也就是 Linux 操作系统上面的进程和这个线程模型有什么区别？唯一的区别就是首先每个线程都有独立的寄存器和堆栈，每个进程也有它独立的寄存器和堆栈，但是进程之间不共享内存，对吧？

所以这是我们的thread-os，也就是我们的框架代码。我们如何将其变为一个真正的操作系统呢？

- **我们让线程不共享内存就行了。**

我们还是叫P1、P2和P3，然后每一个P1和P2和P3都有一个独立的内存，他不共享。当然它们各自也有寄存器。

这就是一个真正的操作系统！



所以你理解了一个线程版本的操作系统——多线程之间是怎么切换的？你回头想很简单，**给每一个线程分配一个地址空间，然后使得这个线程被加载到 CPU 上运行的时候，这个线程看不到其这个世界上所有其他的地方，这是通过一个特别的寄存器实现的，对吧？你们知道我在说 CR3。**

你们都知道我们的计算机有一个重要的机制叫虚拟内存。

我举个例子就是戴上一个 VR 眼镜，而且是强制的，你摘不掉的。有时候你们想我们生活在我们的物理世界里，是不是也带着一种 VR 眼镜，对吧？我们到底是生活在真实的世界里，还是在生活在虚拟的世界里？你分不清。

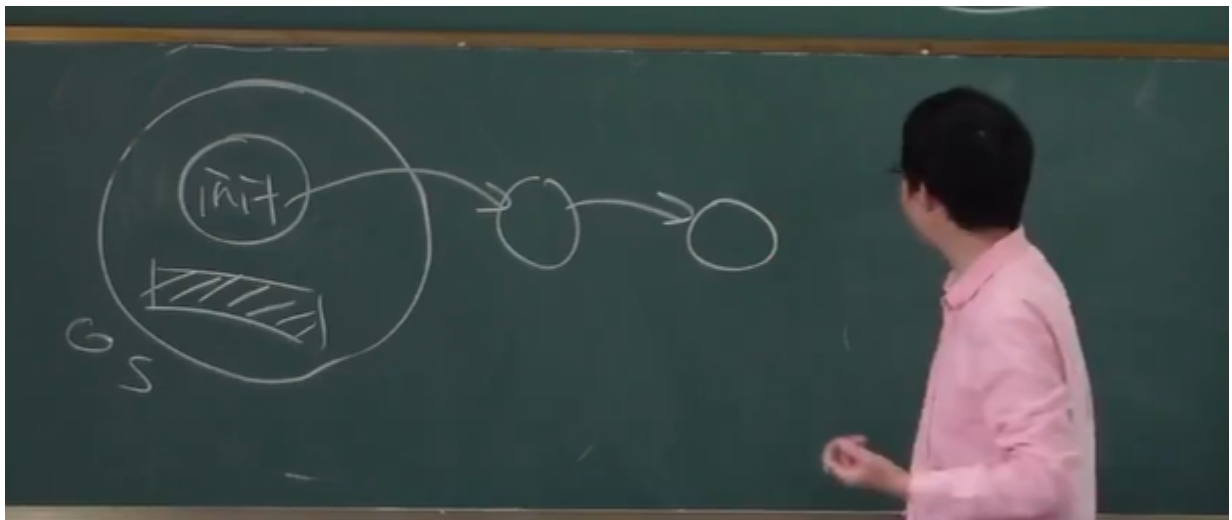
然后我们的进程的实现实际上就是给这个操作系统里面的线程通过虚拟内存这样的一个 VR 眼镜，它戴上以后，那么戴上以后它就从世界的剩下的部分给隔离开了，他只能看到自己的内存，虚拟的内存，然后他有他自己的寄存器，但是他对系统的剩下的部分，第一看不见，第二他没有权限。

然后有这一部分是我处理器，就相当于我的这个处理器可见，但是不能改。然后更多的比如说其他人的内存我是看不见的。而给每个线程戴上这个 VR 眼镜，你就得到了一个多进程的操作系统，所以操作系统真的没有什么神秘的。

其实我第一个月的时间或者一个多月的时间我都在卖关子，对吧？我在讲好像和操作系统没有关系的内容。

你通过上下文切换，通过中断把好多多个状态机放到同一个物理计算机上，因为我们的内存是一个内存条，那个内存条是物理内存，然后通过 CPU 给我们的虚拟内存的机制，可以把好多多个状态机同时放到操作系统里，所以操作系统就是状态机的管理者，对吧？就这一句话就讲完了，什么是操作系统。

有趣的事情就又发生了，我们的操作系统是一个状态机的管理者，而操作系统它自己也是一个状态机，对吧？所以什么是操作系统呢？他自己也是一个状态资源，然后他自己有一些操作系统的内存和数据，对吧？他自己私有的数据。像我刚才那些 struct，每一个线程或者每一个进程它的名字是什么？他运行了多少时间？各种各样的信息，他自己总会留一部分内存，然后剩下的内存它其实是分配给状态机的，对吧？那个状态机的而初始的时候。操作系统代码在完成了整个操作系统的初始化以后，它会加载一个进程。我们的操作系统初始化完了以后，某种程度上它也有一个初始状态。这个初始状态就是一个进程，然后当我的操作系统到达这个状态以后，它就会把这个进程加载到内存里执行，相当于是把这个进程的内存和寄存器加载到处理器上面执行，剩下所有的事情，整个你看到的我们操作系统上面的这个世界，所有的程序应用程序的世界都是由这一个程序创建出来的，**操作系统在完成这个初始化以后，它就会变成一个中断和系统调用的处理程序。**



其实是真的是做了那么多铺垫以后，我们终于来到了操作系统世界里面的这样一个事实，就是所有的进程都是和 minimal.s 一样的状态机。它有两种状态迁移的方式，一种是 syscall——它会回到我们的操作系统，要么就是普通的指令。

我们看到的这整个操作系统世界都是由第一个状态机创建出来的。

In all:

实际上，**在 UNIX/Linux 系统内核完成初始化后，只有一个 init 进程被启动，从此以后，操作系统内核就化身为了一个事件驱动的程序、状态机的管理者，仅在中断和系统调用发生时开始执行。**我们看到的“花花世界”，完全是由进程 (也就是状态机管理) 管理的 API (系统调用) 创建出来的。

这个事实可能会有一点点惊讶，但是你回过头来想一想，其实也对吧？我们要做的事情就是提供系统调用，然后你们会想我们应该提供什么样的系统调用，对吧？除了我们可以把字符打印出来，我们以往设备上面画画以外，我们首先需要的 API 系统，这样的 API 就是状态机的管理。因为在初始的时候，我们的系统设计成只有一个状态机，所以我们的 system call 就必须能够创建状态机，也要能够销毁状态机。

好，那我们就来看，嗯，Unix 世界是怎么来管理这些状态机的。

什么是操作系统？

虚拟化：操作系统同时保存多个状态机

- C 程序 = 状态机
 - 初始状态：main(argc, argv)
 - 状态迁移：指令执行
 - 包括特殊的系统调用指令 syscall
- 有一类特殊的系统调用可以管理状态机
 - CreateProcess(exec_file)
 - TerminateProcess()

从线程到进程：虚拟存储系统

- 通过虚拟内存实现每次“拿出来一个执行”
- 中断后进入操作系统代码，“换一个执行”

2. 状态机管理-创建状态机：fork

fork是unix里面创建状态机唯一的方法，你只能用这种方式创建状态机。

状态机管理：创建状态机

如果要创建状态机，我们应该提供什么样的 API？

UNIX 的答案: fork

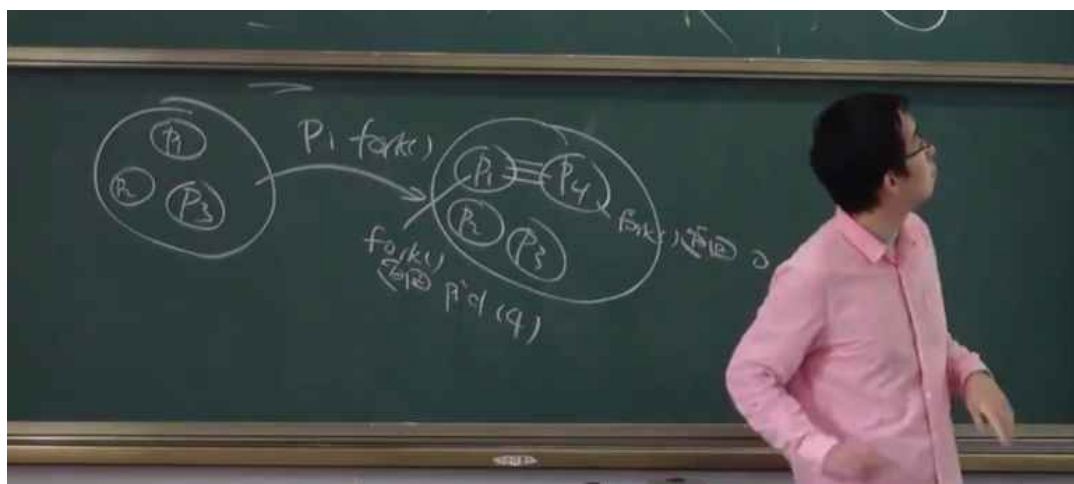
- 做一份状态机完整的复制 (内存、寄存器现场)

`int fork();`

- 立即复制状态机 (完整的内存)
 - 复制失败返回 -1 (errno)
- 新创建进程返回 0
- 执行 fork 的进程返回子进程的进程号



fork 是一个两叉的水果叉——你把一个状态机分成两个完全相同的状态机，除了这个 fork 函数的返回值是不同的。



灵异事件？

```
fish
Hello
$ ./a.out | wc -l
8
$ ./a.out
Hello
Hello
Hello
Hello
Hello
Hello
Hello
$ ./a.out | cat
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
$ ./a.out
Hello
Hello
Hello
Hello
Hello
Hello
$
```

The screenshot shows a tmux session with two panes. The top bar indicates the current window is named 'tmux'. The left pane contains the following commands:

```
1 write(1, "Hello\nHello\n", 12 <unfin
2 write(1, "Hello\nHello\n", 12 <unfin
3 write(1, "Hello\nHello\n", 12) = 12
4 write(1, "Hello\nHello\n", 12)
```

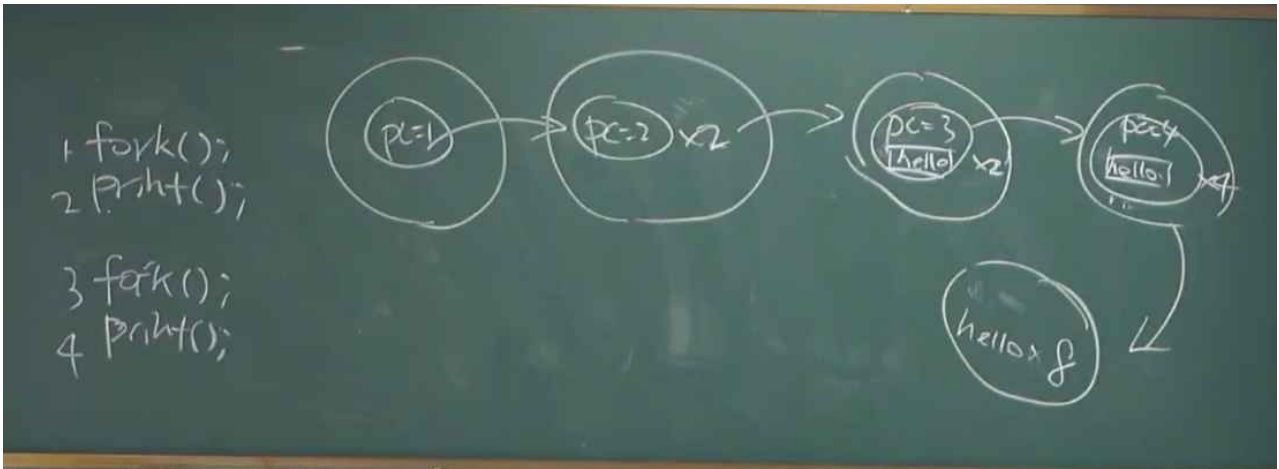
The right pane shows the same commands but with additional output from strace, indicating process creation and completion:

```
1 write(1, "Hello\n", 6) = 6
2 write(1, "Hello\n", 6strace: Process
3 write(1, "Hello\n", 6 <unfinished ..
4 write(1, "Hello\n", 6strace: Process
5 write(1, "Hello\n", 6 <unfinished ..
6 write(1, "Hello\n", 6 <unfinished ..
```

At the bottom of each pane, there are navigation keys: 'b', 'h', 'j', '...', and 'k'. The status bar at the very bottom shows the current pane's position and name: '3,1 All' for the left pane and '6,14 All' for the right pane.

你应该猜到结果了，是因为我们的 `printf` 在打印的时候，不总是直接打印到标准输出的，不是直接调用系统调用把它写出来的，它会根据标准输出连的那个东西是我的终端还是一个管道，它会做不同的行为。如果我的标准输出连到一个管道，它就会把我的输出给 `buffer`了，放到一个缓冲区里，而这个缓冲区是 `Lib c`的，你看不见。在 `PC` 等于 2 的时候，它并不是打印出了两个 `hello`，而是往缓冲区里面扔了两个 `hello`。

p.s. 我感觉jyy这板书buggy



```
1 def main():
2     heap.buf = ''
3     for _ in range(2):
4         sys_fork()
5         heap.buf += '★'
6     sys_write(heap.buf)
7
8 # Outputs:
9 # ★★★★★★★
```

讲的很简单，说 fork 就是状态机的复制，但这个状态机的复制现在你们记得了，是连同所有的代码，所有的数据，不管是你的库函数，还是你的你自己 Malloc new 出来的内存，所有的一切都会被复制一份，无条件的。

3. 重置状态机：execve

你想一想，我现在可以复制状态机了，我如果想要把一个状态机变成另外一个程序，这是一个正常的需求，对吧？我想创建一个shell，我想创建一个浏览器，所以它提供了 fork 复制状态，以后它只要再提供一个 reset 状态机的syscall，我们就可以实现创建状态机的功能。

状态机管理：重置状态机

UNIX 的答案: `execve`

- 将当前进程重置成一个可执行文件描述状态机的初始状态

```
int execve(const char *filename,
           char * const argv[], char * const envp[]);
```

`execve` 行为

- 执行名为 `filename` 的程序
- 允许对新状态机设置参数 `argv (v)` 和环境变量 `envp (e)`
 - 刚好对应了 `main()` 的参数！
- `execve` 是**唯一能够“执行程序”的系统调用**
 - 因此也是一切进程 `strace` 的第一个系统调用

你在重置一个状态机的时候，你就可以给这个重置的状态机传参数了。我们说重置一个状态机，你可以想象成是 `minimal.s`，你也可以把这个重置的状态机想象成是任何一个 `c` 程序，任何一个二进制文件。

`EXECVE` 是唯一的能够执行程序的系统调用，所以你们看到的是所有的进程的 `strace`，它的第一个系统调用都是 `execve`。

环境变量

“应用程序执行的环境”

- 使用 `env` 命令查看
 - `PATH`: 可执行文件搜索路径
 - `PWD`: 当前路径
 - `HOME`: home 目录
 - `DISPLAY`: 图形输出
 - `PS1`: shell 的提示符
- `export`: 告诉 shell 在创建子进程时设置环境变量
 - 小技巧: `export ARCH=x86_64-qemu` 或 `export ARCH=native`
 - 上学期的 `AM_HOME` 终于破案了

看起来很正常的一件事情，你们每天都在做的事情，其实背后是有疑问的，如果你有两个屏幕，它应该在哪个屏幕上显示出来呢？比如你有个服务器，两个同学同时远程连接了两个shell。大家都是同一台机器上运行的shell啊！

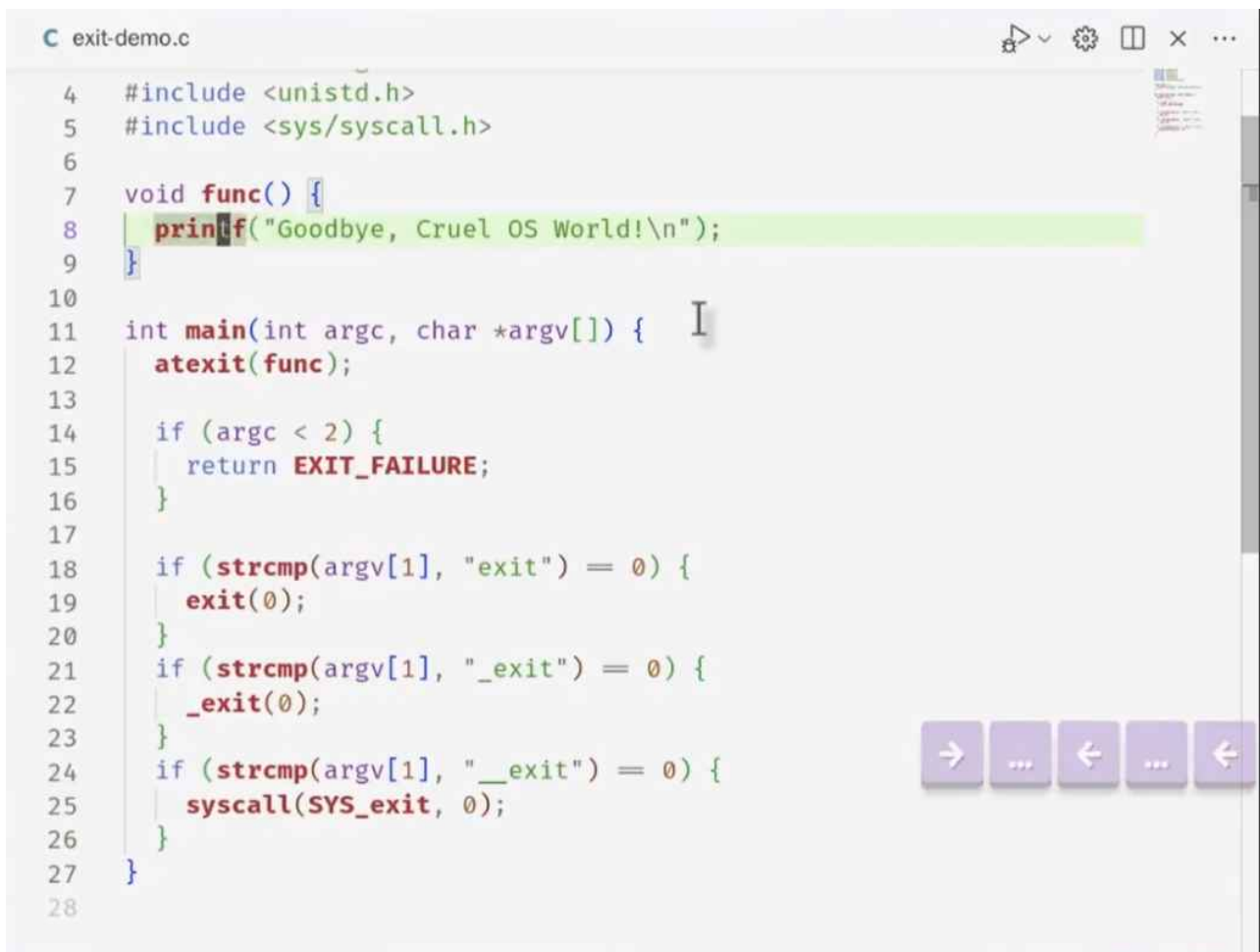
这就是我们的每一个运行的程序share，不管是 shell 还是任何一个程序，除了我给他的命令行参数以外，还有它运行的环境。环境变量也是由 EXECVE 这个系统调用传进去的。

环境变量比如：

- 包括我的语言。当前语言，有的时候你们的系统是中文的，然后有的时候有的程序又输出了英文，谁来决定到底应该给你英文的出错信息还是中文的出错信息，对吧？

4. 销毁状态机：_exit

因为exit是c标准库，所以加了个下划线。



```
C exit-demo.c
4  #include <unistd.h>
5  #include <sys/syscall.h>
6
7  void func() {
8      printf("Goodbye, Cruel OS World!\n");
9  }
10
11 int main(int argc, char *argv[]) {
12     atexit(func);
13
14     if (argc < 2) {
15         return EXIT_FAILURE;
16     }
17
18     if (strcmp(argv[1], "exit") == 0) {
19         exit(0);
20     }
21     if (strcmp(argv[1], "_exit") == 0) {
22         _exit(0);
23     }
24     if (strcmp(argv[1], "__exit") == 0) {
25         syscall(SYS_exit, 0);
26     }
27 }
28
```

系统调用是绝对粗暴的，直接把状态机从我们的世界里面抹除了。C库是优雅的，帮我们实现了一个 graceful shutdown。

结束程序执行的三种方法

exit 的几种写法 (它们是不同)

- `exit(0)` - `stdlib.h` 中声明的 libc 函数
 - 会调用 `atexit`
- `_exit(0)` - glibc 的 syscall wrapper
 - 执行“`_exit_group`”系统调用终止整个进程 (所有线程)
 - 细心的同学已经在 `strace` 中发现了
 - 不会调用 `atexit`
- `syscall(SYS_exit, 0)`
 - 执行“`exit`”系统调用终止当前线程
 - 不会调用 `atexit`

5. Take away messages

因为“程序 = 状态机”，操作系统上进程 (运行的程序) 管理的 API 很自然地就是状态机的管理。在 UNIX/Linux 世界中，这是通过以下三个最重要的系统调用实现的：

- `fork`: 对当前状态机状态进行完整复制
- `execve`: 将当前状态机状态重置为某个可执行文件描述的状态机
- `exit`: 销毁当前状态机

在对这个概念有了绝对正确且绝对严谨的理解后，操作系统也就显得不那么神秘了。