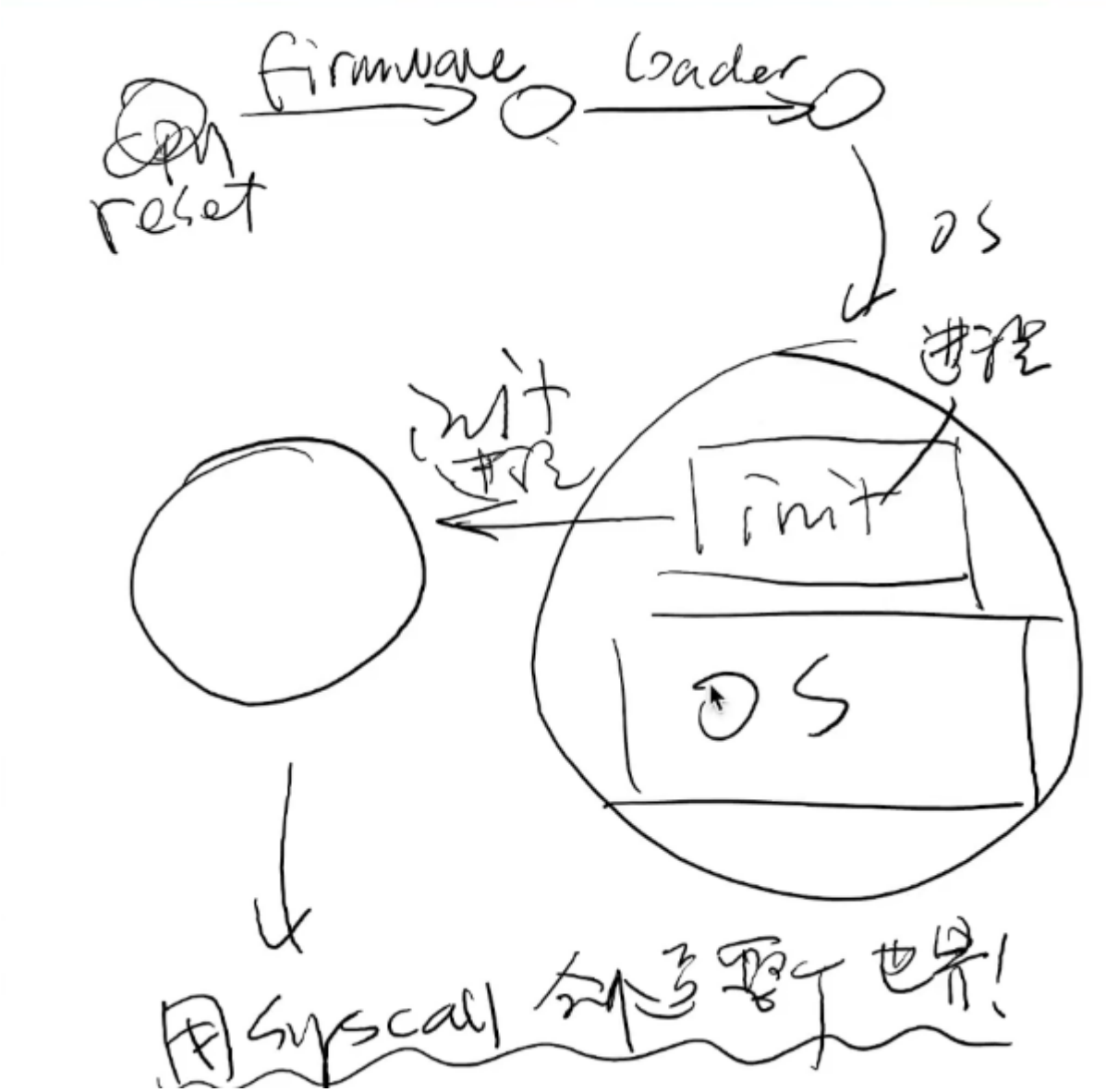
linux 进程

操作系统的初始化可以简化为以下步骤：



先cpurest进行cpu的初始化，然后在跑firmware上的代码，然后跑bootloader，然后跑kernel，然后跑init进程，init开始跑其他程序，此时操作系统退化为一个中断处理程序。

CPU上电后，会从某个地址开始执行。比如MIPS结构的CPU会从0xBFC00000取得第一条指令，而ARM结构的CPU则从地址0x00000000开始。嵌入式开发板中，需要把存储器件ROM或Flash等映射到这个地址，Bootloader就存放在这个地址开始处，这样一上电就可以执行。

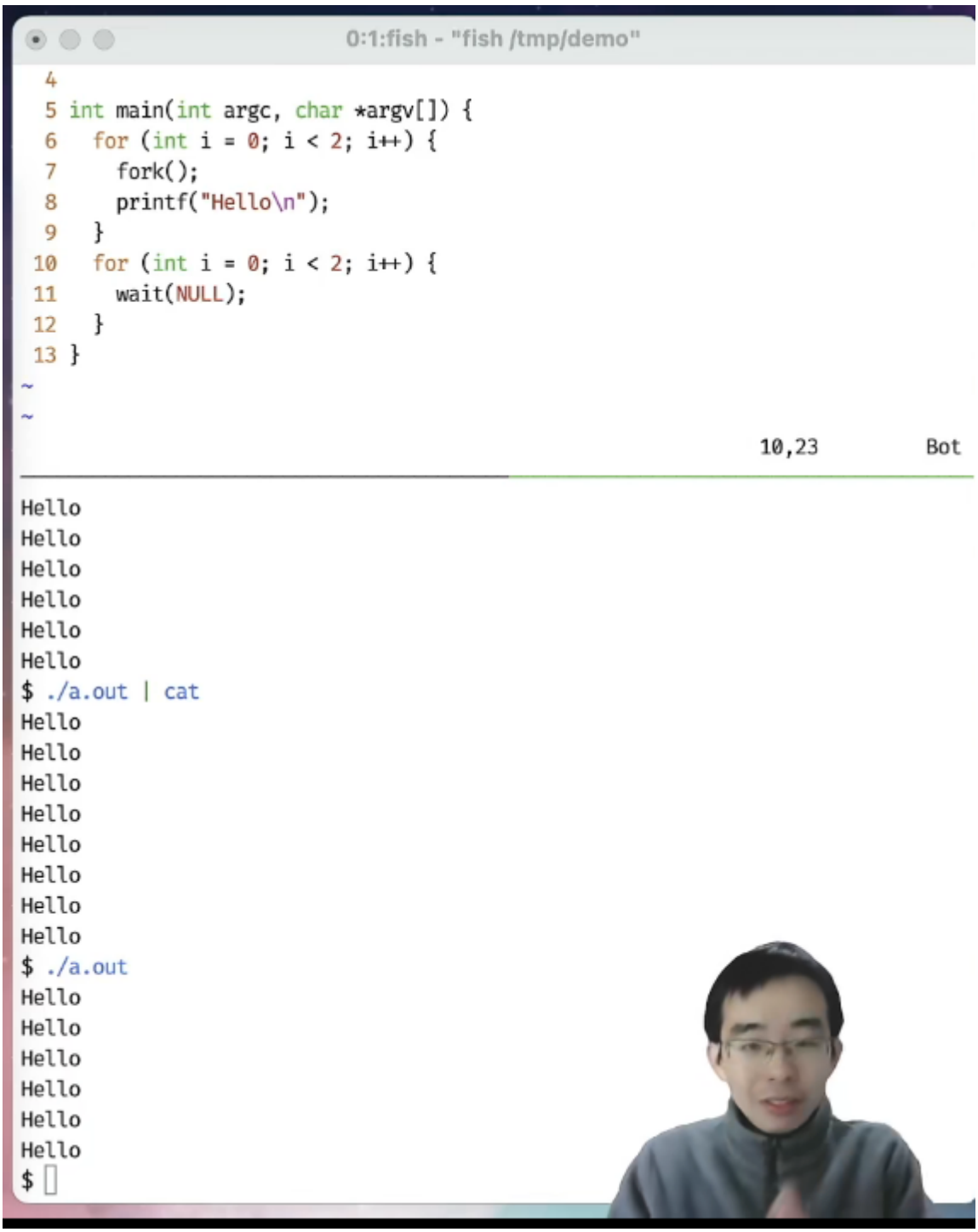
上电后，Bootloader从板子上的某个固定存储设备上将操作系统加载到RAM中运行，整个过程并没有用户的介入。产品发布时，Bootloader工作在这种模式下。

创建进程：fork

fork()函数用于创建一个新进程，它是唯一一个能够创建新进程的函数。fork()函数调用一次，返回两次，子进程返回0，父进程返回子进程的pid。

fork系统调用：操作系统把这个状态机的状态完全赋值一遍，执行完之后，会有两个完全一样的进程副本（状态机），除了fork的返回值。

然后就变成一个类似于并发的程序：操作系统可以选择下一步执行哪一个进程。



我们分析时假设某进程“printf”则会立即输出。

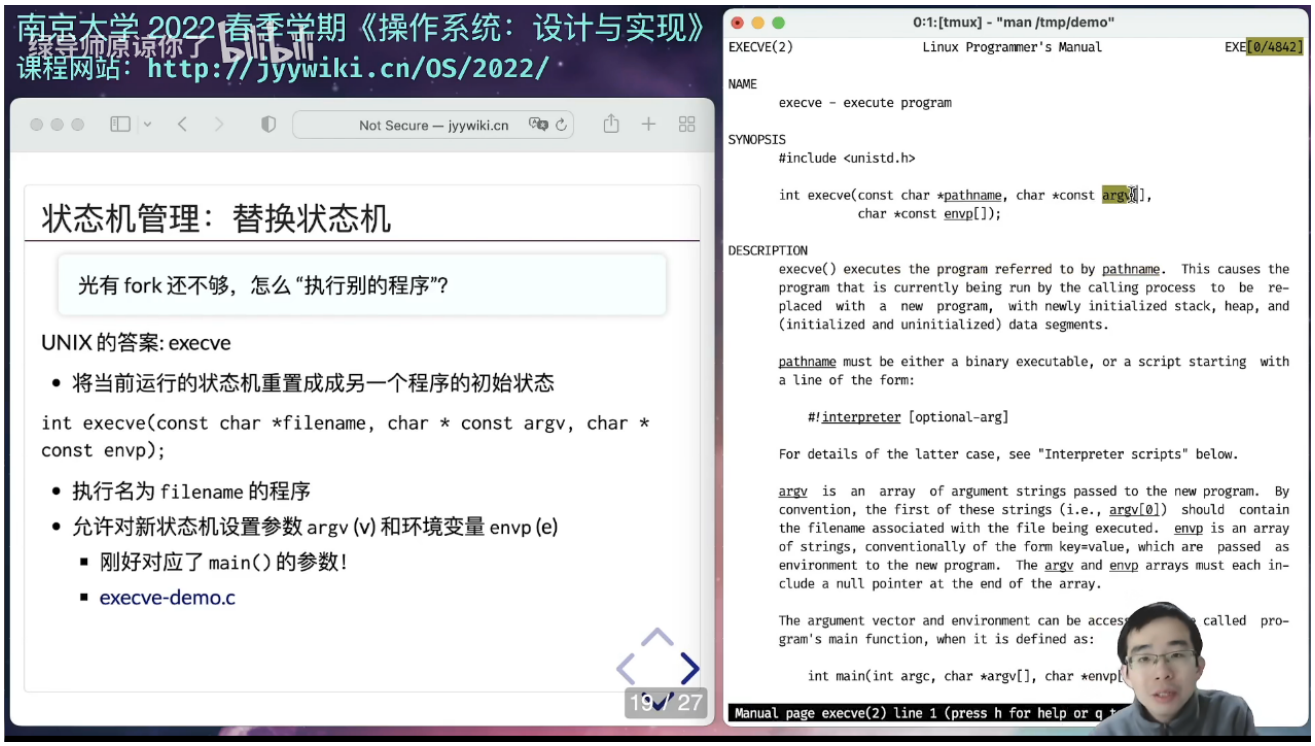
其实我们的printf有缓冲区，对于c的stdout：1.terminal：line buffer（识别到换行符后会打印然后刷新缓冲区）；2.pipe/file：full buffer（满了才会打印）。

显式地调用fflush(stdout)可以强制刷新缓冲区。

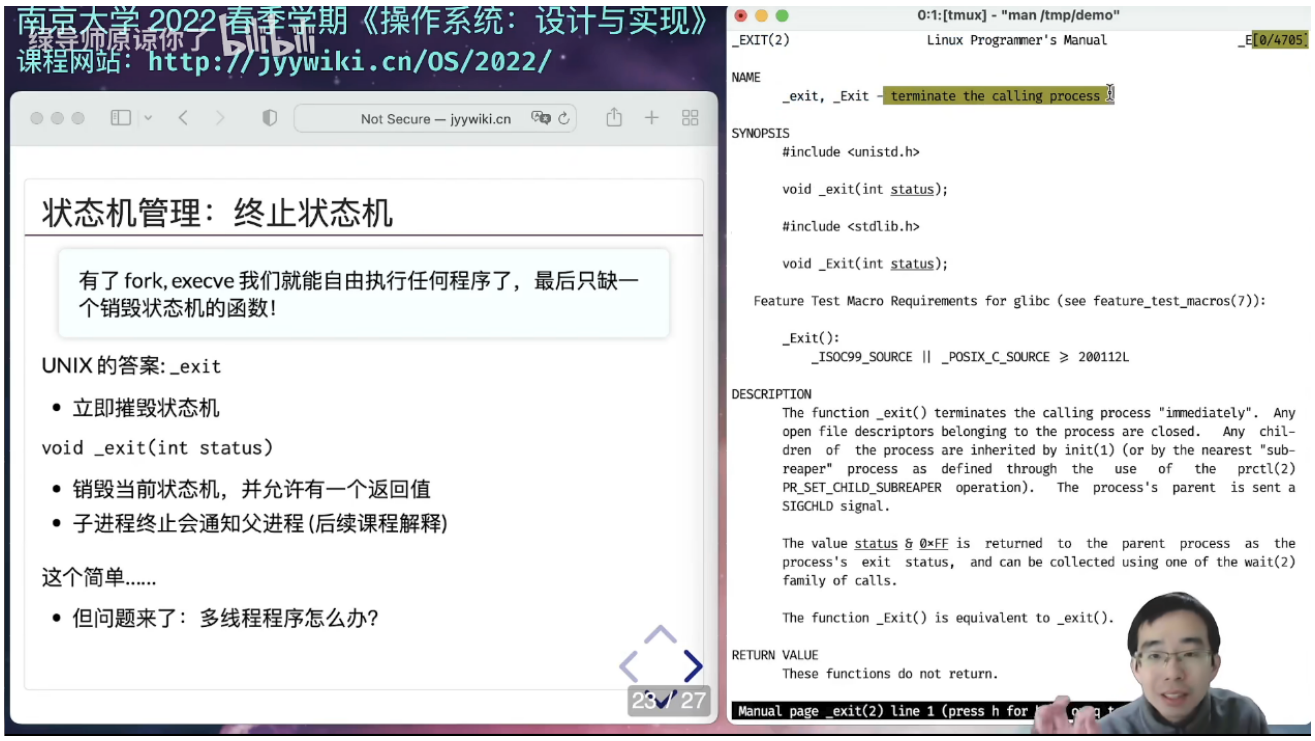
也可以显示的设置 `setbuf(stdout, NULL)`，这样就不会有缓冲区了。

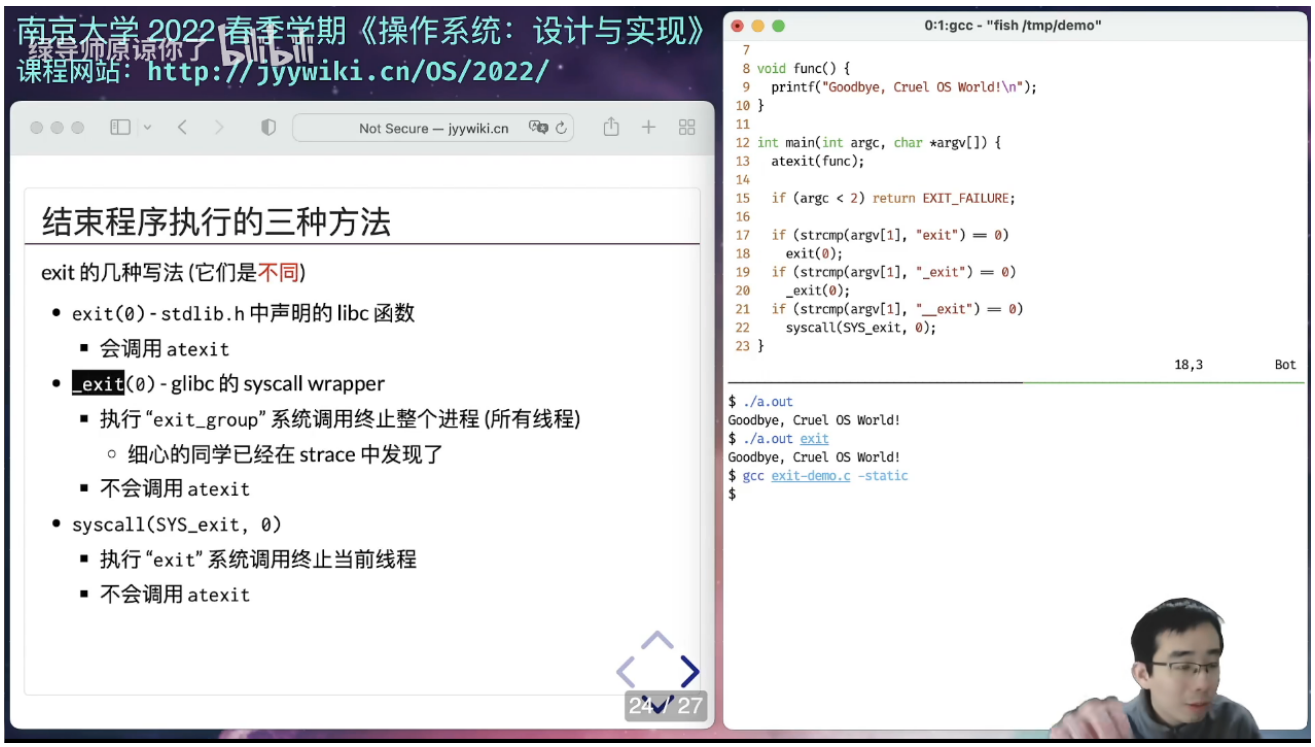
重置状态机：execve

把某个状态机重置为我们指定的某个程序的初始状态



状态机的销毁：exit





如图所示有三种结束程序执行的方法：

1. exit(0)-stdlib.h 中声明的libc函数，会调用atexit（return 也会调用这个）
2. _exit(0)-glibc的syscall wrapper：执行 exit_group syscall 终止整个进程（所有线程）不会调用 atexit：因此不会刷新缓冲区（在atexit中会刷新缓冲区）
3. syscall(SYS_exit, 0): 执行 “exit” 系统调用终止当前线程

进程的地址空间

在汇编语言中，看不到栈帧等概念，只能看到一个连续的内存空间以及寄存器。

由pmap指令可以查看某个进程的地址空间：我们可以发现某个进程的地址空间被划分成若干个连续的段。每个段都被标记上了权限，比如只读，可读可写等。

地址空间被划分为：Text，Data（在程序运行初已经对变量进行初始化的数据），BSS（在程序运行初未对变量进行初始化的数据）， Heap，Stack。

Linux的虚拟地址空间范围为0～4G，Linux内核将这4G字节的空间分为两部分， 将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF）供内核使用，称为“内核空间”。而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF）供各个进程使用，称为“用户空间。因为每个进程可以通过系统调用进入内核，因此，Linux内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。

vdso（virtual dynamically linked shared object)

例如：Linux里面的time的实现：在不进入操作系统内核的前提下进行系统调用。

具体实现:每当内核加载一个ELF可执行程序时，内核都会在其进程地址空间中建立一个叫做vDSO mapping的内存区域。vDSO是virtual dynamic shared object的缩写，表示这段mapping实际包含的是一个**ELF共享目标文件**，也就是俗称的.so。(即操作系统为每个进程中专门开一个VMA（virtual memory area）这段区域用来管理vDSO的物理页就是这个物理文件，只要操作系统改变ELF文件就可以了)

> 补充:elf文件格式 ELF（Executable and Linkable Format）可执行文件是一种常见的二进制文件格式，用于存储程序的可执行代码、数据和符号表等信息。ELF可执行文件是Linux和其他类UNIX系统中常见的可执行文件格式。

> ELF可执行文件通常包含程序的入口点、段表、符号表、重定位表等节（section）信息。程序入口点指的是程序的起始执行点，段表描述了可执行文件中各个段的地址范围、访问权限等信息，符号表则记录了程序中所有符号的信息，如变量名、函数名等，重定位表则用于描述程序需要进行的地址重定位操作。

> ELF可执行文件的特点是它可以通过静态链接或动态链接的方式加载到内存中运行。静态链接将程序的所有依赖库打包进可执行文件中，使得程序独立运行，而动态链接则将依赖库链接到程序运行时的内存空间中，使得程序运行时可以共享依赖库。

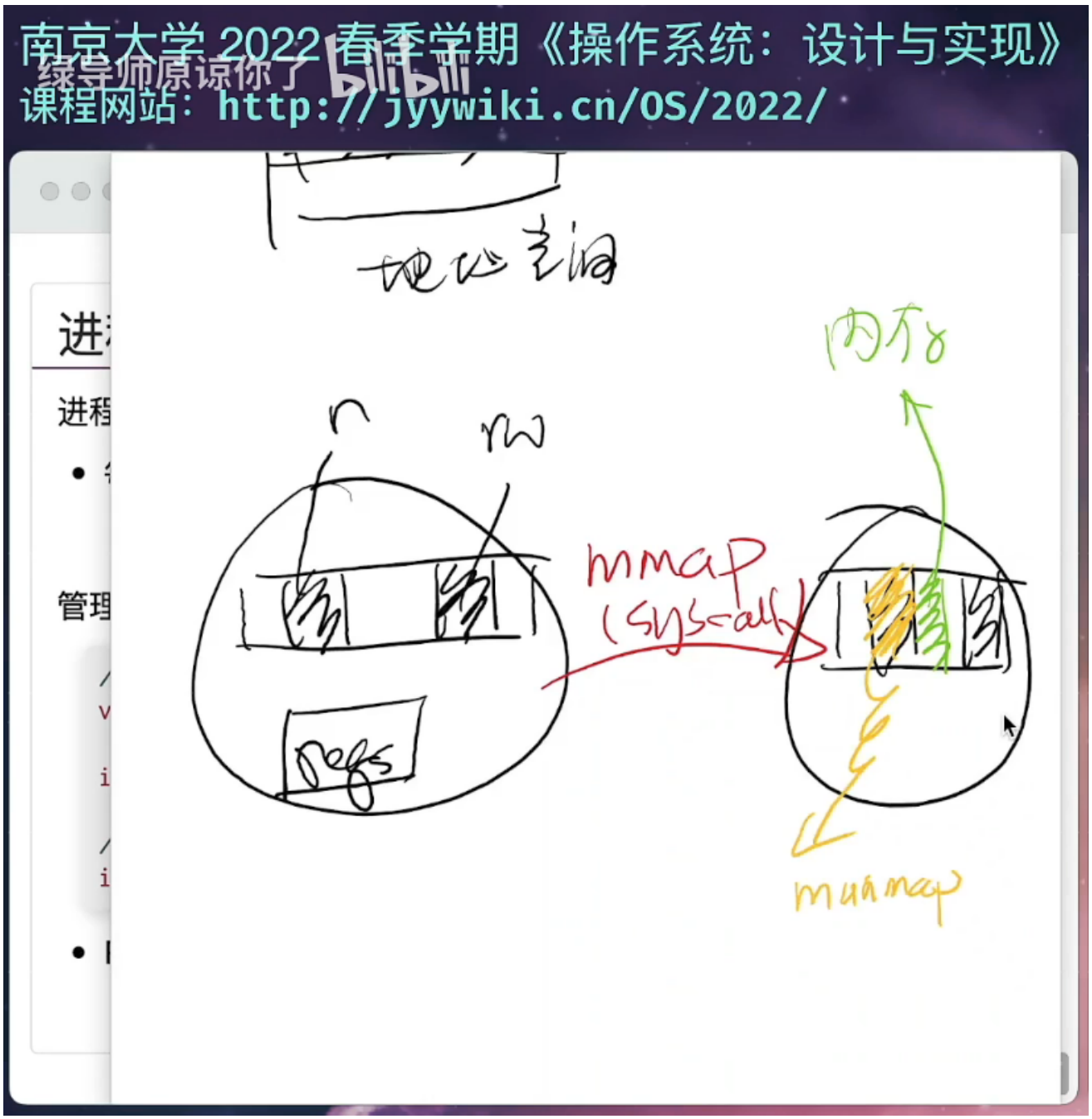
> 总的来说，ELF可执行文件是一种常见的二进制文件格式，它是Linux和其他类UNIX系统中常见的可执行文件格式，具有灵活的链接和运行方式。

补充：sycall的实现：

> 1.用户空间程序通过汇编指令或C库函数进行系统调用syscall。 2.操作系统内核中存在一个系统调用表（syscall table），其中存储了每个系统调用的函数指针。 3.内核将系统调用号传递给系统调用处理程序，该处理程序将根据系统调用号从系统调用表中获取相应的系统调用函数指针。 4.内核执行相应的系统调用函数，并将返回值传递回用户空间程序。 除此之外，再执行系统调用过程中，内核还可能会进行一些额外的检查。

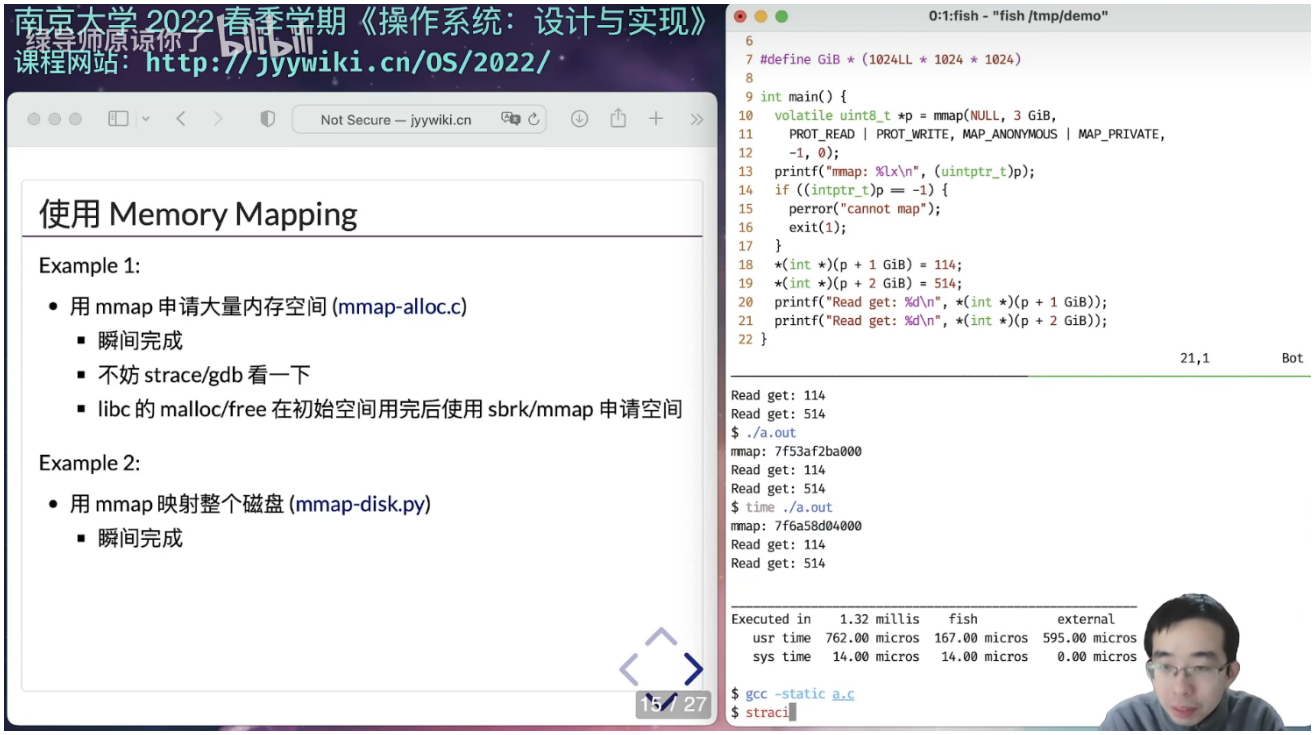
mmap

从状态机角度理解：



mmap把某个地址空间的某个段映射到文件的某个段上

使用实例：



“瞬间完成”的原因：只是标记上了映射关系，直到真正缺页的时候才分配。

系统调用和shell

shell：与系统交互的一个程序，是一个用户态的程序。shell是一门把用户指令翻译成系统调用的语言。

shell其实是包装了一下terminal。

fork 和 execve

fork会赋值一个状态机，这包括某个进程所持有的所有操作系统的对象。

execve会把某个状态机重置为某个初始状态，但是继承持有的所有操作系统对象：比如文件描述符。（也就是fd不会重置，这也就实现了父子进程的通信）。

关于文件描述符：理解为一个打开的文件。

而且打开文件需要“偏移量”。文件描述符会记住上次文件读写的位置。

```
fd = open("a.txt", O_WRONLY | O_CREAT); assert(fd > 0);
pid_t pid = fork(); assert(pid >= 0);
if (pid == 0) {
  write(fd, "Hello");
} else {
  write(fd, "World");
}
```

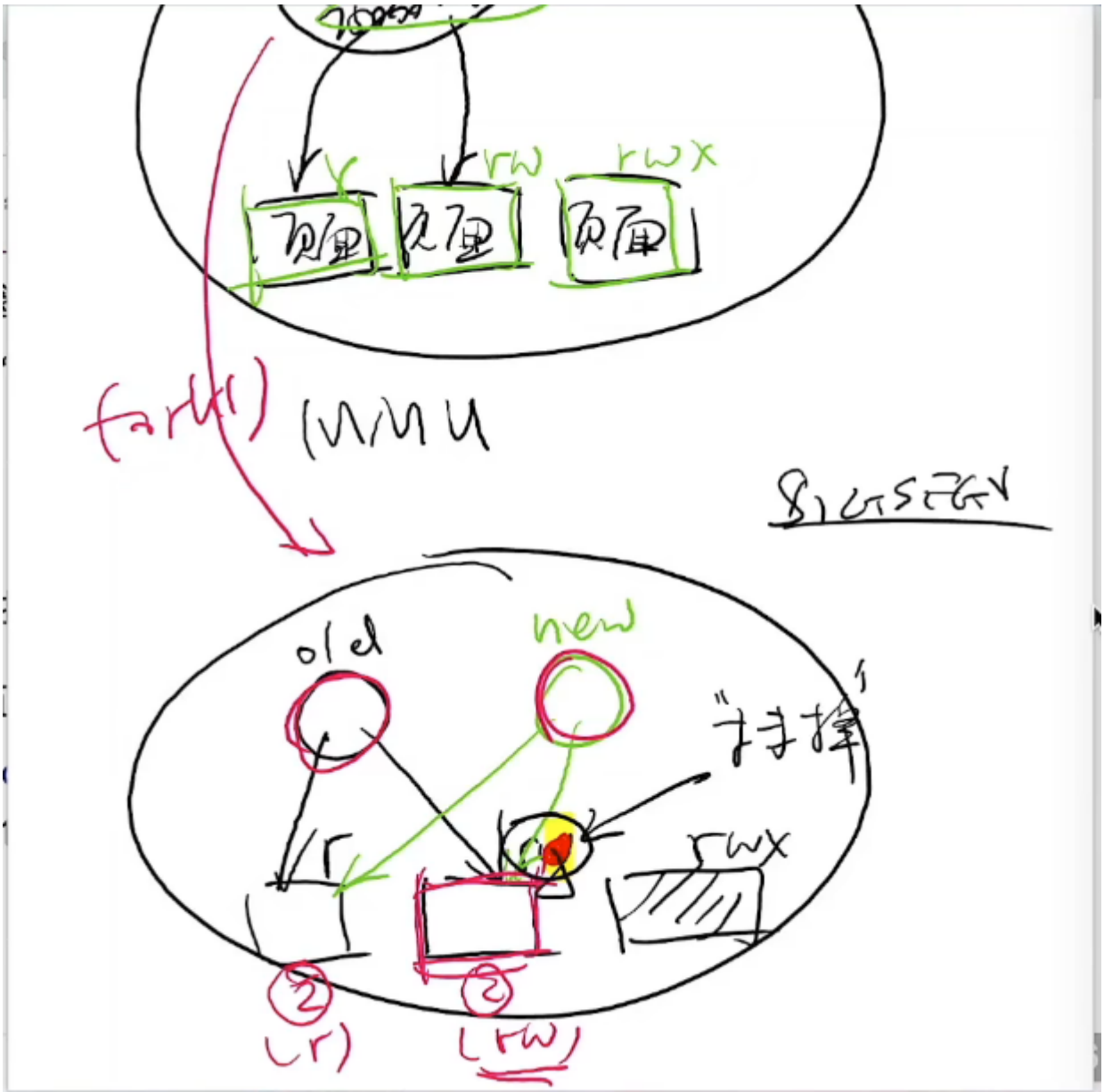
例：上图应该是什么结果？

我们预期的应该是不会产生覆盖的情况。这使得操作系统的设计变的复杂了。POSIX。1-2008规定：Thread Interactions with Regular File Operations. 也就是说，写文件的操作应该是原子的。

fork 的实现

概念上，状态机被复制，但实际上复制后**内存都被共享。**

采用COPY-ON-WRITE（写时复制）的实现，只有在写入某个页面时才会复制一份。概念上，进程持有页面，但是实际上进程只持有页面的引用（一个映射表）。只有操作系统才持有页面。



其实操作系统中的页面是这样的：old，new表示两个进程，fork之后其实只复制了进程的映射表，而没有复制页面，然后操作系统会偷偷地把“可写”的权限抹掉，同时记录每个页面被引用的次数。只有当进程要写如某个页面时，操作系统才会复制一份页面，然后把引用计数减一，新复制的页面的引用计数为1，只有引用计数为1的页面才可以被写入，否则需要先复制再写入。

所以，统计进程占用的内存其实是一个伪命题，我们可以统计进程占用的“虚拟内存”（这里指的不是系统中的虚存）。

如果多个进程都有同样的的一个初始化过程，那么我们可以在初始化之后再fork，这样很多个进程就可以共享同一个初始化过程，省掉时间（Android app就是这样设计的，chrome浏览器也是这样的）。

除此之外，fork还可用于备份与容错。

当有多线程的时候，只有执行fork的线程会被复制，其他线程不会被复制。