

08 | 操作系统内核与编程接口

2019-05-10 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 13:39 大小 12.51M



你好，我是七牛云许式伟。

今天我们在开发软件的时候，已经处于一些基础的架构设计之中，像冯·诺依曼计算机体系，像操作系统和编程语言，这些都是我们开发一个应用程序所依赖的基础架构。

在上一节第 7 讲“[软件运行机制及内存管理](#)”中，我们着重介绍了软件是如何被运行起来的。今天，我们着重聊一聊软件如何利用它所依赖的基础架构。

首先是冯·诺依曼计算机体系，它由“中央处理器 + 存储 + 一系列的输入输出设备”构成。这一层，提供了编程接口的是中央处理器（CPU），编程接口是 CPU 指令，但 CPU 指令非常难用。

为此，人们发明了编程语言来降低 CPU 指令的使用门槛。编程语言面向人类，CPU 指令面向机器，编译器负责将人类容易理解和掌握的编程语言的程序，翻译成机器能够理解的 CPU 指令序列。

其次是编程语言。虽然编程语言出现的起因是降低 CPU 指令的使用门槛，第一门编程语言汇编语言的能力也很接近 CPU 指令，但是语言的自然演化会越来越脱离 CPU 所限制的条条框框，大部分语言都会演化出很多基础的算法库。

比如，字符串算法库有：字符串连接（concat）、字符串子串（substring），字符串比较（compare）、字符串长度（length）等等。

系统调用

最后就是操作系统了。

操作系统和前两者非常不同。软件都是某种编程语言写成的，而 CPU 和编程语言的能力，统一以语言的语法或者库体现。

操作系统则属于基础软件，它和我们编写的软件并不在同一个进程（进程是软件的一个运行后产生的实例，同一个软件可以运行多次得到多个进程）中。

如果是实模式下的操作系统，大家都在同一个地址空间下，那么只需要知道操作系统的接口函数地址，理论上就可以直接访问。但是今天主流的操作系统都是保护模式的，操作系统和软件不在同一个进程，软件怎么才能使用操作系统的能力呢？

你可能想说，那就用进程与进程之间的通信机制？

的确，操作系统提供了很多进程与进程之间通讯的机制，后面我们也会涉及。但是今天我们讲的操作系统的编程接口是更为基础的机制，它是所有软件进程使用操作系统能力的基础，包括进程与进程之间通讯的机制，也是建立在这个基础之上。

它应该是一种成本非常非常低的方案，性能上要接近函数调用，否则我们为保护模式付出的成本就太高了。

有这样的机制么？有，就是上一节我们已经提到过的“中断”。

中断的设计初衷是 CPU 响应硬件设备事件的一个机制。当某个输入输出设备发生了一件需要 CPU 来处理的事情，它就会触发一个中断；但是 CPU 也提供了指令允许软件触发一个中断，我们把它叫软中断。

大部分情况下，操作系统的能力通过软中断向我们写的软件开放，为此还专门引入了一个术语叫“系统调用 (syscall)”。

系统调用是怎么工作的？

我们需要先了解下 CPU 的代码执行权限等级。

在保护模式下，CPU 引入了“保护环 (Protection Rings)”的概念。说白了，代码有执行权限等级的，如果权限不够，有一些 CPU 指令就不能执行。

这一点比较容易理解：上一节我们介绍过，从内存管理的角度，虚拟内存机制让软件运行在一个沙盒中，这个沙盒让软件感觉自己在独享系统的内存。但如果不对软件的执行权限进行约束，它就可以打破沙盒，了解到真实的世界。

我们通常说的操作系统是很泛的概念。完整的操作系统非常庞大。根据与应用的关系，我们可以把操作系统分为内核与外围。

所谓操作系统内核，其实就是指那些会向我们写的应用程序提供系统服务的子系统的集合，它们管理着计算机的所有硬件资源，也管理着所有运行中的应用软件（进程）。

操作系统内核的执行权限等级，和我们常规的软件进程不同。像 Intel CPU 通常把代码执行权限分为 Ring 0-3 四个等级。

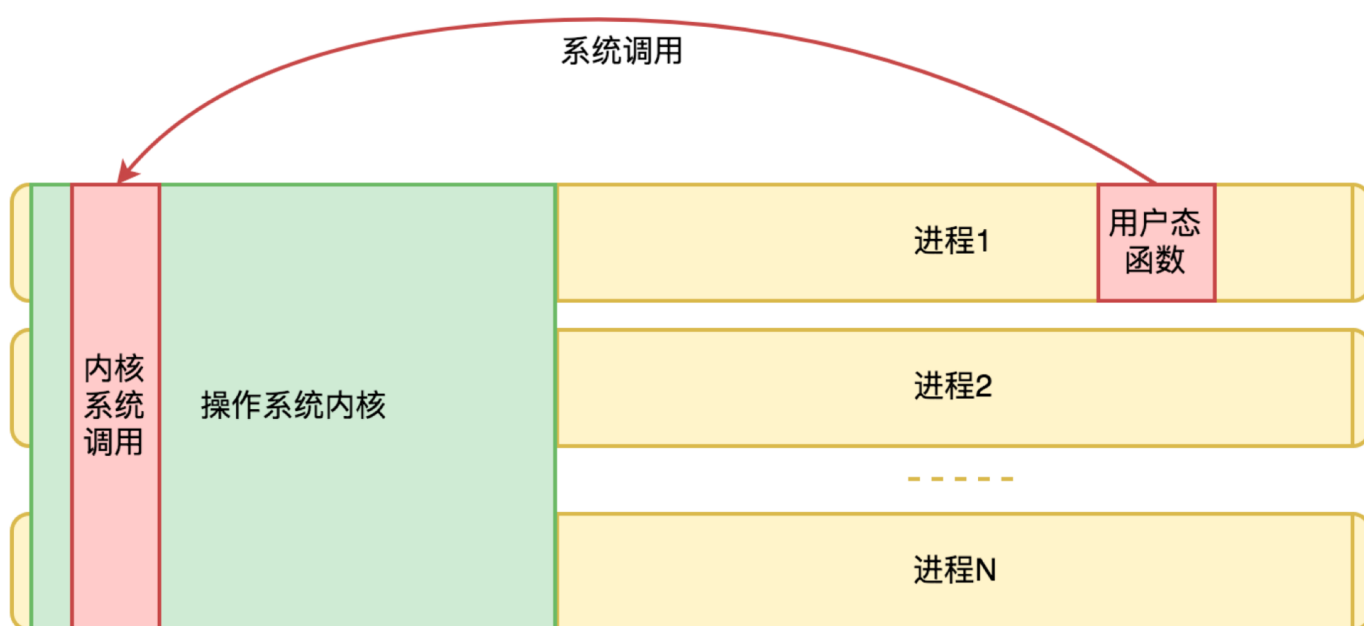
操作系统内核通常运行在 Ring 0，而常规的软件进程运行在 Ring 3（当然近年来虚拟化机制流行，为了更好地提升虚拟化的效率，Intel CPU 又引入了 Ring -1 级别的指令，这些指令只允许虚拟机所在的宿主操作系统才能调用）。

系统调用所基于的软中断，它很像一次间接的“函数调用”，但是又颇有不同。在实模式下，这种区别并不强烈。但是在保护模式下，这种差异会十分明显。

原因在于，我们的应用程序运行在 Ring 3（我们通常叫用户态），而操作系统内核运行在 Ring 0（我们通常叫内核态）。所以一次中断调用，不只是“函数调用”，更重要的是改变了执行权限，从用户态跃迁到了内核态。

但是这似乎不够。我们之前说了，操作系统与我们编写的软件并不同属一个进程，两边的内存地址空间都是独立的，我们系统调用请求是过去了，但是我们传给操作系统的内存地址，对它真的有意义吗？

答案在于，从虚拟内存机制的视角，操作系统内核和所有进程都在同一个地址空间，也就是，操作系统内核，它是所有进程共享的内存。示意如下：



这非常有趣。操作系统内核的代码和数据，不只为所有进程所共享，而且在所有进程中拥有相同的地址。这样无论哪个进程请求过来，对内核来说看起来都是一次本进程内的请求。

从单个进程的视角，中断向量表的地址，以及操作系统内核的地址空间是一个契约。有了中断向量表的地址约定，用户态函数就可以发起一次系统调用（软中断）。

当然你可能要问：**既然操作系统内核和我同属一个地址空间，我是否可以跳过中断，直接访问调用内核函数？**


这不单单是执行权限的问题。你可能会说，也许某个内核函数里面没有调用任何特权指令，我是否可以调用？

当然不能。这涉及虚拟内存中的内存页保护机制。内存页可以设置“可读、可写、可执行”三个标记位。操作系统内核虽然和用户进程同属一个地址空间，但是被设置为“不可读、不可写、不可执行”。虽然这段地址空间是有内容的，但是对于用户来说是个黑洞。

编程接口

理解了操作系统内核，以及它的调用方法“系统调用”，我们来聊一聊操作系统的编程接口。

自然，最原始的调用方式，是用软中断指令。在汇编语言里面通常是：

 复制代码

```
1 int < 中断号 > ; 对每个操作系统来说中断号是固定的，比如 Linux 是 0x80
```

这里的 int 不是整数 (integer) 的缩写，而是中断 (interrupt) 的缩写。

当然用汇编语言来写软件并不是一个好主意。大部分高级语言都实现了操作系统编程接口的封装。

前面我们说，操作系统（内核）有六大子系统：存储管理、输入设备管理、输出设备管理、进程管理、网络管理、安全管理。除了安全管理是一个“润物细无声”的能力外，其他子系统都会有所包装。

我们以 C 语言和 Go 语言为例给一个简表，方便大家索引：

子系统	子类别	Go语言	C语言
存储	内存	语言支持 GC, 不手工管理	stdlib.h - malloc, free, etc.
	外存(文件系统)	os - Mkdir, Open, etc.	direct.h - mkdir, opendir, etc.
	外存(文件)	os - Rename, Remove, Create/Open, etc. fmt - Fscan, Fprint, etc.	stdio.h - rename, remove, fopen, fscanf, fprintf, etc.
输入/输出	命令行	fmt - Scan, Print, etc.	stdio.h - scanf, printf, etc.
	窗口系统/GDI	跨平台难, 标准库中缺失	跨平台难, 标准库中缺失
进程(内)	执行体	goroutine	thread
	互斥体	sync - Mutex, RWMutex	pthread.h - pthread_mutex_t, pthread_rwlock_t
	等待组	sync - WaitGroup	无对应, pthread_join 实现相关能力
	条件变量	sync - Cond	pthread.h - pthread_cond_t
	管道	io - Pipe	unistd.h - pipe
	消息传递	channel , 语言内建支持	标准库中缺失
进程(间)	进程启动	os - StartProcess os/exec - Command	unistd.h - fork, execl, etc.
	文件锁	os - OpenFile	fcntl.h - open
	信号	os/signal - Notify	signal.h - signal
	信号量	认为过时, 标准库中缺失	sys/sem.h - semget
	共享内存	syscall - Map, Unmap	sys/mman.h - mmap, munmap
	匿名管道	os - Pipe	unistd.h - pipe
	命名管道	syscall - Mkfifo, Unlink os - OpenFile	sys/stat.h - mkfifo
	UNIX 域	net - Dial	sys/socket.h - socket
网络	DNS	net - Resolver	sys/socket.h - getaddrinfo
	UDP/TCP/IP	net - Dial	sys/socket.h - socket
	HTTP	net/http - Handle, Get, Post, etc.	标准库中缺失

这些标准库的能力, 大部分与操作系统能力相关, 但或多或少进行了适度的包装。

例如, HTTP 是应用层协议, 和操作系统内核关联性并不大, 基于 TCP 的编程接口可以自己实现, 但由于 HTTP 协议细节非常多, 这个网络协议又是互联网世界最为广泛应用的应

用层协议，故此 Go 语言提供了对应的标准库。

进程内通讯最为复杂。虽然操作系统往往引入了 thread 这样的概念，但 Go 语言自己搞了一套 goroutine 这样的东西，原因是什么，我们在后面讨论“进程管理”相关的内容时，再做详细讨论。

动态库

从操作系统的角度来说，它仅提供最原始的系统调用是不够的，有很多业务逻辑的封装，在用户态来做更合适。但是，它也无法去穷举所有的编程语言，然后——为它们开发各种语言的基础库。那怎么办？

聪明的操作系统设计者们想了一个好办法：动态库。几乎所有主流操作系统都有自己的动态库设计，包括：

Windows 的 dll (Dynamic Link Library) ；

Linux/Android 的 so (shared object) ；

Mac/iOS 的 dylib (Mach-O Dynamic Library) 。

动态库本质上是实现了一个语言无关的代码复用机制。它是二进制级别的复用，而不是代码级别的。这很有用，大大降低了编程语言标准库的工作量。

动态库的原理其实很简单，核心考虑两个东西。

浮动地址。动态库本质上是在一个进程地址空间中动态加载程序片段，这个程序片段的地址显然在编译阶段是没法确定的，需要在加载动态库的过程把浮动地址固定下来。这块的技术非常成熟，我们在实模式下加载进程就已经在使用这样的技术了。

导出函数表。动态库需要记录有哪些函数被导出 (export)，这样用户就可以通过函数的名字来取得对应的函数地址。

有了动态库，编程语言的设计者实现其标准库来说就多了一个选择：直接调用动态库的函数并进行适度的语义包装。大部分语言会选择这条路，而不是直接用系统调用。

操作系统与编程语言

我们这个专栏从计算机硬件结构讲起，然后再到编程语言，到现在开始介绍操作系统，有些同学可能会觉得话题有那么一些跳跃。虽然每一节的开头，我其实对话题的脉络有所交代，但是，今天我还是有必要去做一个梳理。

编程语言和操作系统是两个非常独立的演化方向，却又彼此交融，它们有点像是某种“孪生关系”。虽然操作系统的诞生离不开编程语言，但是操作系统和 CPU 一样，是编程语言背后所依赖的基础设施。

和这个话题相关的，有这么一些有趣的问题：

先有编程语言，还是先有操作系统；

编程语言怎么做到自举的（比如用 C 语言来实现 C 语言编译器）；

操作系统开发的环境是什么样的，能够做到操作系统自身迭代本操作系统（自举）么？

对于**第一个问题：先有编程语言，还是先有操作系统？**这个问题的答案比较简单，先有编程语言。之所以有这个疑问，是因为两点：

其一，大部分人习惯认为运行软件是操作系统的责任。少了责任方，软件是怎么跑起来的？但实际上软件跑起来是很容易的，看 BIOS 程序把控制权交给哪个软件。

其二，大部分常见的应用程序都直接或间接依赖操作系统的系统调用。这样来看，编程语言编译出来的程序是无法脱离操作系统而存在的。但是实际上常见的系统级语言（比如 C 语言）都是可以编写出不依赖任何内核的程序的。

对于**第二个问题：编程语言怎么做到自举的？**

从鸡生蛋的角度，编译器的进化史应该是这样的：先用机器码直接写第一个汇编语言的编译器，然后汇编语言编译器编出第一个 C 语言编译器。有了 C 语言编译器后，可以反过来用 C 语言重写汇编语言编译器和 C 语言编译器，做更多的功能增强。

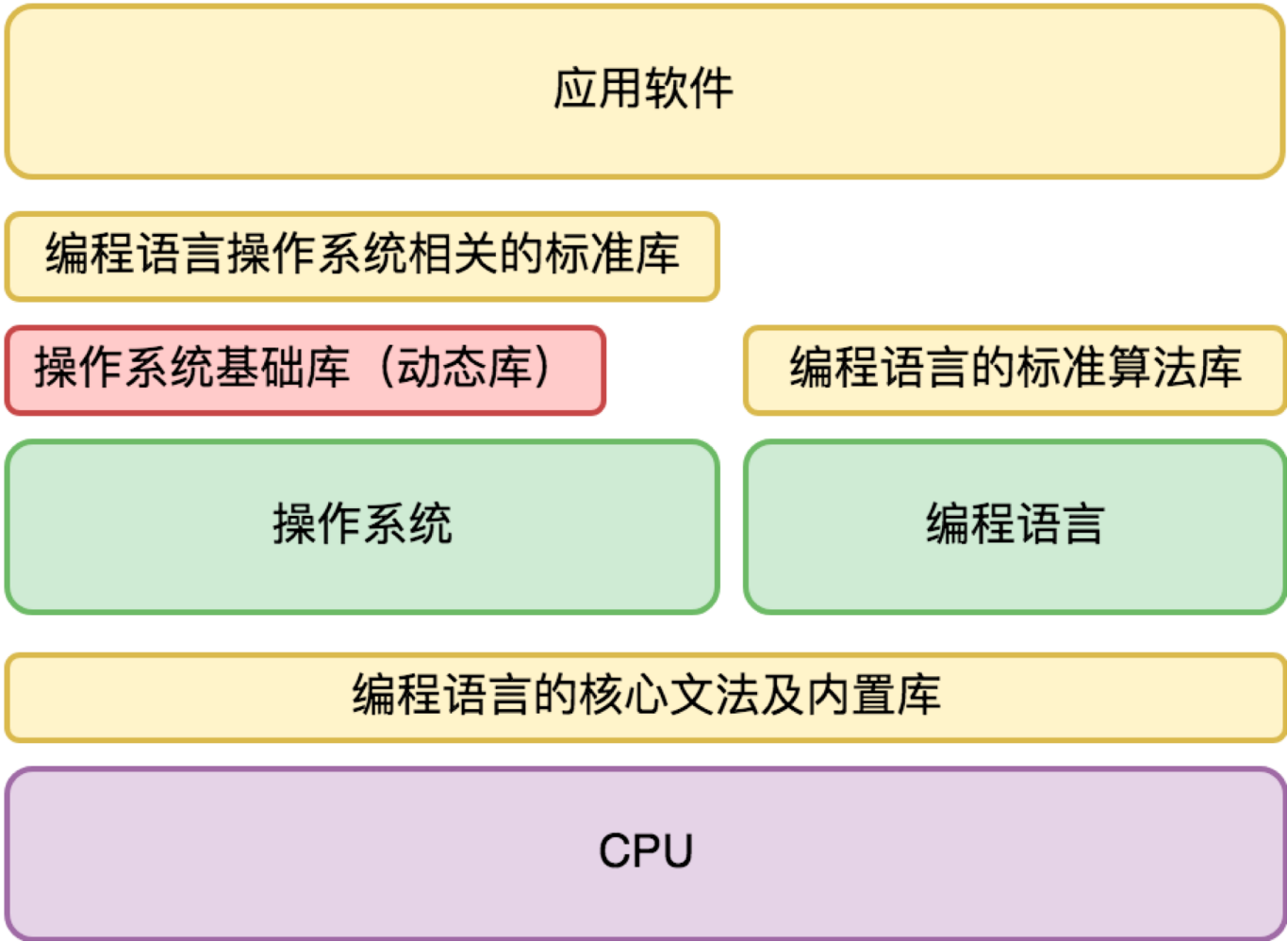
这个过程理论上每出现一种新 CPU 指令集、新操作系统，就需要重新来一遍。但是人是聪明的。所以交叉编译这样的东西产生了。所谓交叉编译就是在一种“CPU + 操作系统”架构下，生成另一种“CPU + 操作系统”架构下的软件。这就避免了需要把整个编译器进化史重新演绎一遍。

对于**第三个问题：操作系统能够做到自身迭代本操作系统（自举）么？**

当然可以。通常一门新的操作系统开发之初，会用上面提到的交叉编译技术先干出来，然后等到新操作系统稳定到一定程度后再实现自举，也就是用本操作系统自己来做操作系统的后续迭代开发。

结语

这一节我们介绍了我们的基础架构：中央处理器（CPU）、编程语言、操作系统这三者对应用软件开放的编程接口。总结来看就是下面这样一幅图：



其中，我们着重介绍的是操作系统的系统调用背后的实现机理。通过系统调用这个机制，我们很好地实现了操作系统和应用软件的隔离性和安全性，同时仍然保证了极好的执行性能。


如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

许式伟的架构课

从源头出发, 带你重新理解架构设计

许式伟
七牛云 CEO



新版升级: 点击「 请朋友读」, 20位好友免费读, 邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 软件运行机制及内存管理

下一篇 09 | 外存管理与文件系统

精选留言 (34)

 写留言



:)

2019-05-10

 22

老师讲得太好了, 让我对操作系统有了更加深入的理解, 个人认为理解好老师所讲的内容, 有如下关键点: 1. 操作系统和我们写的程序运行于两个进程中 2. 我们写的程序无法直接访问操作系统所在的内存 3. 连接操作系统进程和用户进程的桥梁是中断地址。可以设想在还没有出现操作系统的上古时代, 有两个普通的程序A和B在cpu上跑着, 那么程序A和程序B如何交互呢? 一种简单的办法就是程序A直接跳到程序B的函数地址上, cpu去执行该函...
展开



SuperFrank...

2019-05-11

 12

cpu不需要检查是否发生了中断, 它的原理类似于开关和灯泡的关系, 当开关合上, 灯泡就

会亮，灯泡不需要定期检查开关是否合上了

作者回复: 



Fz

2019-05-10

 9

交叉编译不是很理解

展开 

作者回复: 其实理解清楚一个实质: 编译器就是把高级语言翻译成为机器码, 更抽象说, 它其实就是格式转换器。目标格式是不是编译器正在运行的环境并不重要, 只不过如果目标格式刚好是当前机器的CPU+操作系统, 那么目标格式就可以直接执行, 否则就编译出一个当前环境下无法执行的目标格式, 这种情况就叫交叉编译。



杨怀

2019-05-13

 5

老师好, 我这有个问题, 就是有中断必有对应的中断处理程序, 那么执行中断处理程序会不会像普通线程那样抢占cpu资源呢, 如果没抢到还要等一等? 还有就是一下同时来了十几个中断, 那么怎么处理呢?

展开 

作者回复: 软中断你可以把它理解为虚函数调用, 本来就占着cpu资源呢, 不需要等。一下子来很多中断是可能的, 在硬件中断的情况下, 这时候会根据中断优先级响应。




云学

2019-05-12

 5

看完有种苹果砸到脑袋上的感觉, 好多问题之前总想不透, 现在贯通了, 许老师可以创立个品牌专门教小孩编程, 以老师的功底肯定能教会

作者回复: 正在教自家小孩 



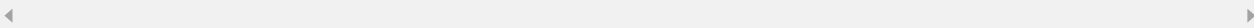
long.mr

2019-05-12

👍 4

许老师，问一哈在画架构图的时候有没有一些约定俗成的准则呢，比如虚线 实线 方框 圆角矩形的选择。业界有一些权威的参考吗？

作者回复: UML 里面有一些约定，可以参考。但是更重要的是团队内的约定。



QQ怪

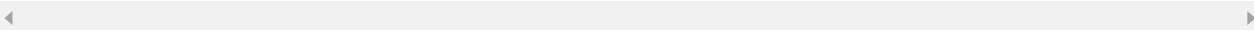
2019-05-10

👍 4

老师能否讲下协程的优势?不太理解

展开 ▾

作者回复: 很快要讲到了



行者

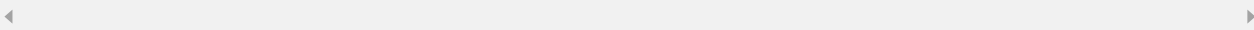
2019-05-13

👍 3

关于动态库，是不是可以理解为：提供动态库不是操作系统的责任（因为其他语言完全可以通过系统调用来自己实现动态库的功能），只是操作系统为了方便其他语言（减少不必要的冗余）而做的多余的事？

展开 ▾

作者回复: 是的



晓凉

2019-05-12

👍 3

优秀的架构设计能带来的好处可能超出架构师最初的预想，例如Linux系统的容器特性，现在成了云计算领域的重要基础技术，成就了一个领域。基于前辈大师的优秀架构，现在的信息世界才能如此生生不息。优秀的架构不仅能解决当前的实际问题，更具有理论上的优雅，像一种真理，可照亮未来。

作者回复: 我们日常所见明明有无数设计精良的例子，所以我们其实不需要一上来去设计新的例子，从这些最宏大的例子学起





李文斌

2019-05-11

👍 3

老师 但是实际上常见的系统级语言（比如 C 语言）都是可以编写出不依赖任何内核的程序的。这句话是说我们编写的程序 没有操作系统也可以跑起来是吗

作者回复: 要让引导程序给你执行权才行，如果你写一个不依赖操作系统的程序，但是要让操作系统执行它是行不通的，因为操作系统接管了所有的资源，你不依赖它什么也干不了（没有权限）。所以这种写法基本上只适合写另一个操作系统。



K战神

2019-05-10

👍 3

k8s是不是也是类似于软件治理平台？

展开 ∨

作者回复: 是的



dong

2019-05-10

👍 3

这是我看过的讲的最清楚的计算机组成原理和操作系统。期待更新。

展开 ∨



M

2019-05-10

👍 3

请问一下cpu是如何检查是否有中断的。是怎么及时知道发生了中断？每执行完一条指令都去检查一次吗？

作者回复: 挺好的问题。硬件中断和软中断不一样。硬件中断你可以理解为总是会定期检查。软中断本身是一条指令，所以不存在检查这样的概念。



泉水叮咚...

2019-05-22

👍 2

老师解释的真是很明白，尤其是系统调用的工作原理，我一直都没有想明白普通用户自定义函数调用和系统调用之间到底有什么区别，原来以为是垮进程内存地址传递和内存共享，原来压根儿就是在同一个内存地址空间里啊！OS内核地址空间被所有用户进程自己操作系统自己共享啊。

上学时学习操作系统课程，上面说虚拟内存低地址空间已经分给操作系统内核使用，这...

展开 ▾



Yayu

2019-05-13

👍 2

请问老师，变成语言提供的内置库（SDK）可大体氛围两类：1、对（操作系统各个子系统的）系统调用的封装；2、业务（逻辑）层的工具。那我们这里说的SDK是文章末尾分层图中的【编程语言操作系统相关的标准库】和【编程语言的算法库】对吧？那【编程语言的核心文法及内置库】这一层如何理解？落实到具体的指代哪些库呢？

展开 ▾

作者回复: 编程语言的核心文法及内置库主要就是语言各种基础特性，主要就是语言文法部分，比如变量、控制语句（if、for、switch）、函数定义等等



Barry

2019-05-13

👍 2

看来有人提了朗读者之前读速有点快的问题，这节就慢了下来。这个语速挺好的



陈光

2019-05-12

👍 2

老师，CPU划定权限范围，操作系统负责分配权限，可以这样理解吗？另外，为什么CPU会需要定期检查“硬中断”而不需要定期检查“软中断”？是因为软中断是更“高”一层的中断吗？

展开 ▾

作者回复: 软中断就是一条cpu指令，遇到了执行就行了。硬件中断我觉得其实也不存在定期检查，所谓检查只是从软件思维去理解硬件而已，前面有人举了开关合上灯就亮了是一个挺恰当的比喻，灯并没有去不停检查开关是否合上，所谓检查只是从软件去理解这种现象的一种想象。



honnkyou

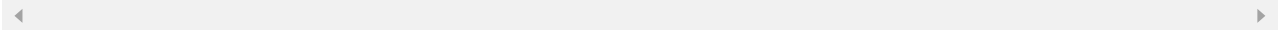
2019-05-10

👍 2

老师，cpu是怎么知道什么代码可以在ring0中执行，什么代码需要在ring3中执行的呢？是在程序中标注的吗？

展开 ▾

作者回复: 不是程序标注，是一个cpu当前的状态值（比如用一个寄存器专门表示当前的执行权限）



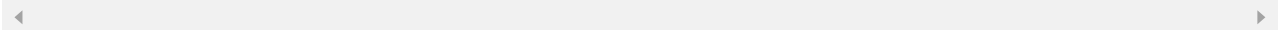
pawhrmyki

2019-05-10

👍 2

假设可寻址范围4G，用户空间：0 ~ 3G，内核3G ~ 4G，那对于保护模式下的用户程序，他能访问到的虚拟的4个G中，其实真正属于他的只有3G，还有1G是和内核以及其他所有进程共享的，而这1G的地址对于所有进程（包括操作系统）都是一样的，可以这么理解吗

作者回复: 1G就是保留个内核，对所有进程来说一样



82

2019-05-10

👍 2

编程语言只是一个工具，而操作系统作为接口平台，干了很多复杂的事情。其实语言本身也是可以直接跳过操作系统去对接硬件。

操作系统作为聚合平台其实也不是必须的，只是为了提升某块的效率，对吧

展开 ▾

作者回复: 操作系统并不是简单的辅助工具，而是软件治理的平台。所以软件正常是没法绕过操作系统的，除非是找到漏洞提权了。

