

07 | 软件运行机制及内存管理

2019-05-07 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 13:28 大小 12.35M



你好，我是七牛云许式伟。

操作系统的核心职能是软件治理，而软件治理的一个很重要的部分，就是让多个软件可以共同合理使用计算机的资源，不至于出现争抢的局面。

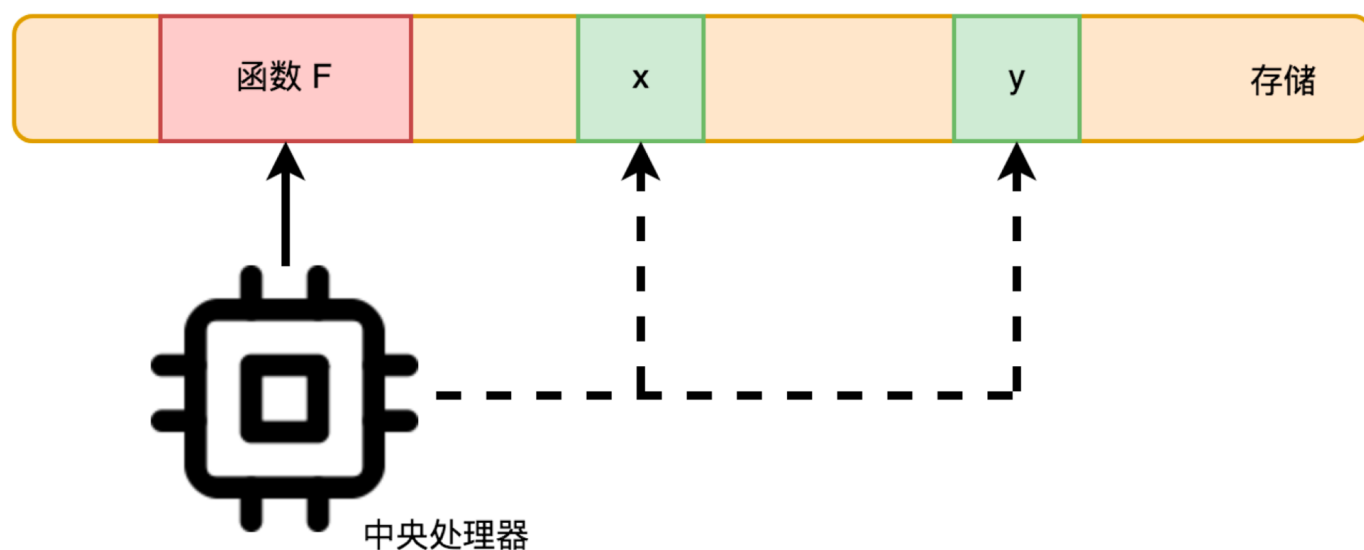
内存作为计算机最基础的硬件资源，有着非常特殊的位置。我们知道，CPU 可以直接访问的存储资源非常少，只有：寄存器、内存（RAM）、主板上的 ROM。

寄存器的访问速度非常非常快，但是数量很少，大部分程序员不直接打交道，而是由编程语言的编译器根据需要自动选择寄存器来优化程序的运行性能。

主板上的 ROM 是非易失的只读的存储。所谓非易失，是计算机重新启动后它里面的数据仍然会存在。这不像内存（RAM），计算机重新启动后它上面的数据就丢失了。ROM 非易失和只读的特点，决定了它非常适合存储计算机的启动程序（BIOS）。

所以你可以看到，内存的地位非常特殊，它是唯一的 CPU 内置支持，且和程序员直接会打交道的基础资源。

内存有什么用？前面我们在 [“02 | 大厦基石：无生有，有生万物”](#) 一节中介绍冯·诺依曼结构的时候，画过一个图：



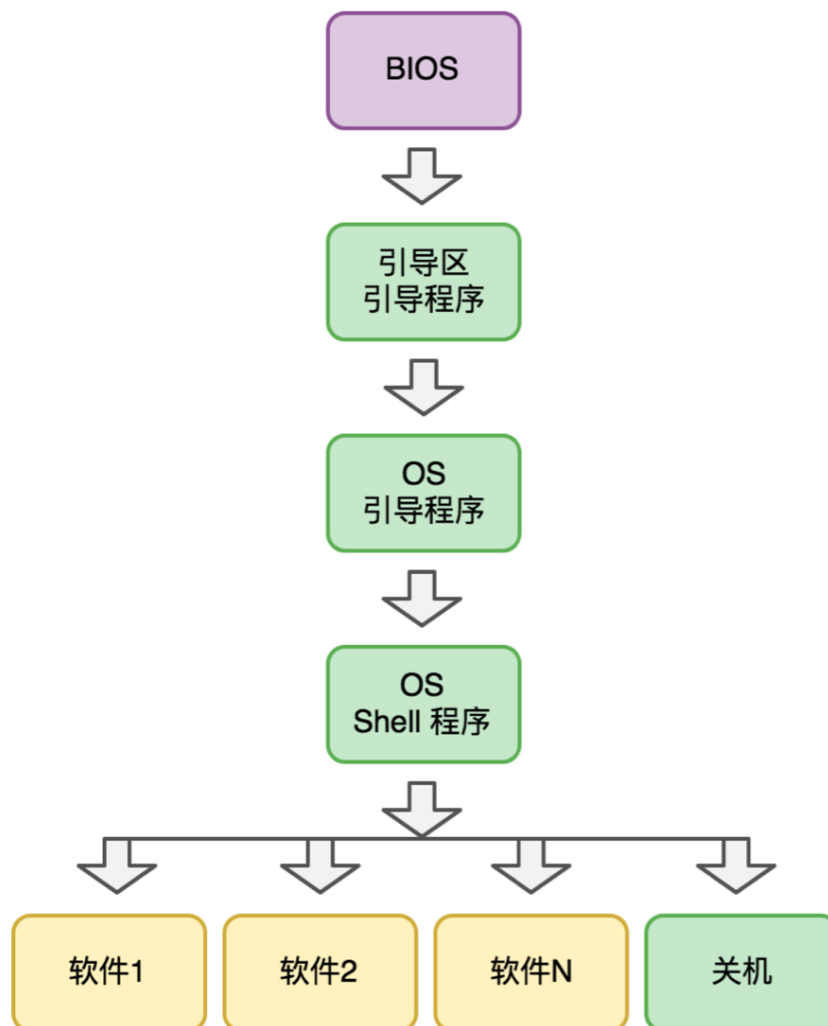
从图中可以看出，存储的作用有两个：一个是作为“计算”的操作对象，输入和输出数据存放的所在；另一个是存放“计算”本身，也就是程序员写的程序。

这里说的存储，主要指的就是内存。

计算机运行全过程

当然，这是从 CPU 角度看到的视图：对于 CPU 来说，“计算”过程从计算机加电启动，执行 BIOS 程序的第一条指令开始，到最后计算机关机，整个就是一个完整的“计算”过程。这个过程有多少个“子的‘计算’过程”，它并不关心。

但是从操作系统的视角来看，计算机从开机到关机，整个“计算”过程，由很多软件，也就是子“计算”过程，共同完成。从时序来说，计算机完整的“计算”过程如下：



整个“计算”过程的每个子过程都有其明确的考量。

首先，BIOS 程序没有固化在 CPU 中，而是独立放到主板的 ROM 上，是因为不同历史时期的计算机输入输出设备很不一样，有键盘 + 鼠标 + 显示器的，有触摸屏的，也有纯语音交互的，外置存储则有软盘，硬盘，闪存，这些变化我们通过调整 BIOS 程序就可以应对，而不需要修改 CPU。

引导区引导程序，则是程序从内置存储（ROM）转到外置存储的边界。引导区引导程序很短，BIOS 只需要把它加载到内存执行就可以，但是这样系统的控制权就很巧妙地转到外置存储了。

引导区引导程序不固化在 BIOS 中，而是写在外置存储的引导区，是为了避免 BIOS 程序需要经常性修改。毕竟 BIOS 还是硬件，而引导区引导程序已经属于软件范畴了，修改起来会方便很多。

OS 引导程序，则是外置存储接手计算机控制权的真正开始。这里 OS 是操作系统（Operating System）的缩写。操作系统从这里开始干活了。这个过程发生了很多很多事

情，这里我们先略过。但是最终所有的初始化工作完成后，操作系统会把执行权交给 OS Shell 程序。

OS Shell 程序负责操作系统与用户的交互。最早的时候，计算机的交互界面是字符界面，OS Shell 程序是一个命令行程序。DOS 中叫 `command.com`，而在 Linux 下则叫 `sh` 或者 `bash` 之类。这里的 `sh` 就是 `shell` 的缩写。

这个时期启动一个软件的方式就是在 Shell 程序中输入一个命令行，Shell 负责解释命令理解用户的意图，然后启动相应的软件。到了图形界面时期，在 Shell 中启动软件就变成点点鼠标，或者动动手指（触摸屏）就行了，交互范式简化了很多。

在了解了计算机从开机到关机的整个过程后，你可能很快会发现，这里面有一个很关键的细节没有交代：计算机是如何运行外置存储上的软件的？

这和内存管理有关。

结合内存的作用，我们谈内存管理，只需要谈清楚两个问题：

如何分配内存（给运行中的软件，避免它们发生资源争抢）；

如何运行外置存储（比如硬盘）上的软件？

在回答这两个问题之前，我们先了解一个背景知识：CPU 的实模式和保护模式。这两个模式 CPU 对内存的操作方式完全不同。在实模式下，CPU 直接通过物理地址访问内存。在保护模式下，CPU 通过一个地址映射表把虚拟的内存地址转为物理的内存地址，然后再去读取数据。

相应的，工作在实模式下的操作系统，我们叫实模式操作系统；工作在保护模式下的操作系统，我们叫保护模式操作系统。

实模式下的内存管理

先看实模式操作系统。

在实模式操作系统下，所有软件包括操作系统本身，都在同一个物理地址空间下。在 CPU 看来，它们是同一个程序。操作系统如何分配内存？至少有两种可行的方法。

其一，把操作系统内存管理相关的函数地址，放到一个大家公认的地方（比如 0x10000 处），每个软件要想申请内存就到这个地方取得内存管理函数并调用它。

其二，把内存管理功能设计为一个中断请求。所谓中断，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。

内存的全局有一个中断向量表，本质上就是在一个大家公认的地方放了一堆函数地址。比如键盘按了一个键，它会触发 9 号中断。在 CPU 收到中断请求时，它会先停下手头的活来响应中断请求（到中断向量表找到第 9 项对应的函数地址并去执行它），完成后再回去干原来的活。

中断机制设计之初本来为响应硬件事件之用，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。比如我们约定 77 号中断为内存管理中断，操作系统在初始化时把自己的内存管理函数写到中断向量表的第 77 项。

所以，上面两种方法实质上是同一个方法，只是机制细节有所不同而已。中断机制远不止是函数向量表那么简单。比如中断会有优先级，高优先级中断可以打断低优先级中断，反之则不能。

那么，在实模式下，操作系统如何运行外置存储（比如硬盘）上的软件？

很简单，就是把软件完整从外置存储读入到内存然后执行它。不过，在执行前它干了一件事情，把浮动地址固定下来。为什么会有浮动地址？因为软件还没有加载到内存的时候并不知道自己会在哪里，所以有很多涉及数据的地址、函数的地址都没法固定下来，要在操作系统把它加载到内存时来确定。

整体来说，实模式内存管理的机制是非常容易理解的。因为它毕竟实质上是一个程序被拆分为很多个软件（程序代码片段），实现了程序代码片段的动态加载而已。

保护模式下的内存管理

但实模式有两个问题。

其一是安全性。操作系统以及所有软件都运行在一起，相互之间可以随意修改对方的数据甚至程序指令，这样搞破坏就非常容易。

其二是支持的软件复杂性低，同时可运行的软件数量少。

一方面，软件越复杂，它的程序代码量就越多，需要的存储空间越大，甚至可能出现单个软件的大小超过计算机的可用内存，这时在实模式下就没法执行它。

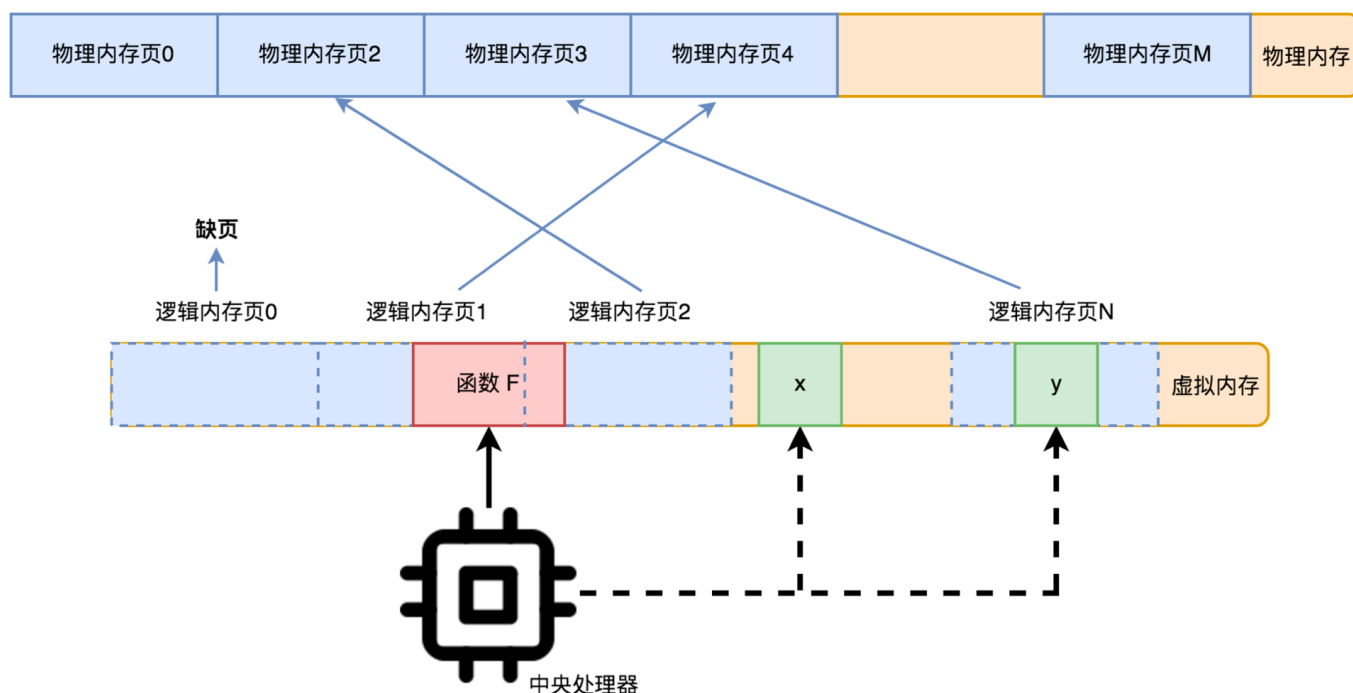
另一方面，哪怕单个软件可运行，但是一旦我们同时运行的软件多几个，操作系统对内存的需求量就会急剧增加。相比这么多软件加起来的内存需求量，内存的存储空间往往仍然是不足的。

但是为什么平常我们可以毫无顾忌地不断打开新的软件，从来不曾担心过内存会不足呢？

这就是保护模式的作用了。保护模式下，内存访问不再是直接通过物理内存，而是基于虚拟内存。虚拟内存模式下，整个内存空间被分成很多个连续的内存页。每个内存页大小是固定的，比如 64K。

这样，每次 CPU 访问某个虚拟内存地址中的数据，它都会先计算出这是要访问哪个内存页，然后 CPU 再通过一个地址映射表，把虚拟的内存地址转为物理的内存地址，然后到这个物理内存地址去读取数据。地址映射表是一个数组，下标是内存页页号，值是该内存页对应的物理内存首地址。

当然，也有可能某一个内存页对应的物理内存地址还不存在，这种情况叫缺页，没法读取数据，这时 CPU 就会发起一个缺页的中断请求。



这个缺页的中断请求会被操作系统接管。发生缺页时，操作系统会为这个内存页分配物理的内存，并恢复这个内存页的数据。如果没有空闲的物理内存可以分配，它就会选择一个最久没有被访问的内存页进行淘汰。

当然，淘汰前会把这个内存页的数据保存起来，因为下次 CPU 访问这个被淘汰的内存页时一样会发生缺页中断请求，那时操作系统还要去恢复数据。

通过这个虚拟内存的机制，操作系统并不需要一上来就把整个软件装进内存中，而是通过缺页中断按需加载对应的程序代码片段。多个软件同时运行的问题也解决了，内存不够用的时候，就把最久没有用过的内存页淘汰掉，腾出物理内存出来。

运行软件的问题解决了。那么，操作系统如何分配内存给运行中的软件？

其实，内存分配的问题也解决了，并不需要任何额外的机制。反正内存地址空间是虚拟的，操作系统可以一上来就给要运行的软件分配超级大的内存，你想怎么用随你。软件如果不用某个内存页，什么都不发生。软件一旦用了某个内存页，通过缺页中断，操作系统就分配真正的物理内存给它。

通过引入虚拟内存及其缺页机制，CPU 很好地解决了操作系统和软件的配合关系。

每个运行中的软件，我们把它叫进程，都有自己的地址映射表。也就是说，虚拟地址并不是全局的，而是每个进程有一个自己独立的虚拟地址空间。

在保护模式下，计算机的基础架构体系和操作系统共同在努力做的一件事情，就是让每个软件“感觉”自己在独占整个计算机的资源。独立的虚拟地址空间很好地伪装了这一点：看起来我独自在享用所有内存资源。在实模式下的浮动地址的问题也解决了，软件可以假设自己代码加载的绝对地址是什么，不需要在加载的时候重新调整 CPU 指令操作的地址。

这和实模式很不一样。在实模式下，所有进程都在同在物理内存的地址空间里，它们相互可以访问对方的数据，修改甚至破坏对方的数据，进而导致其他进程（包括操作系统本身的进程）崩溃。内存是进程运行的基础资源，保持进程基础资源的独立性，是软件治理的最基础的要求。这也是保护模式之所以叫“保护”模式的原因。

架构思维上我们学到什么？

虚拟内存它本质上要解决这样两个很核心的需求。

其一，软件越来越大，我们需要考虑在外置存储上执行指令，而不是完整加载到内存中。但是外置存储一方面它的数据 CPU 并不知道怎么读；另一方面就算知道怎么读，也不知道它的数据格式是什么样的，这依赖文件系统的设计。让 CPU 理解外置存储的实现细节？这并不是一个好的设计。

其二，要同时运行的软件越来越多，计算机内存的供给与软件运行的内存需求相比，捉襟见肘。怎么才能把有限的内存的使用效率最大化？一个很容易想到的思路是把不经常使用的内存数据交换到外置存储。但是问题仍然是，CPU 并不了解外置存储的实现细节，怎么能把内存按需交换出去？

通过把虚拟内存地址分页，引入缺页中断，我们非常巧妙地解决了这个问题。缺页中断很像是 CPU 留给操作系统的回调函数，通过它对变化点实现了很好的开放性设计。

结语

总结一下。我们今天先概要地阐述了计算机运行的全过程，并对每一步的核心意义做了简单的介绍。然后我们把话题转到我们这一节的重心：内存管理。

谈内存管理，需要谈清楚两个核心问题：

如何分配内存（给运行中的软件，避免它们发生资源争抢）；

如何运行外置存储（比如硬盘）上的软件？

我们分别就在实模式下和保护模式下的内存管理进行了讨论。


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 06 | 操作系统进场

下一篇 08 | 操作系统内核与编程接口

精选留言 (60)

 写留言



coder

2019-05-08

 11

我觉得许大大的这个专栏写的挺好的。从计算机底层的角度去思考和总结架构的观点, 让人觉得眼前一亮。

btw, 看到评论区有人问cache的事情, 补充一下自己的看法。

cache一般对programmer和软件来说确实是透明的, 但是它也有缺点, 比如说占用了大...

展开 ▾

作者回复:  挺好的补充





黄海峰

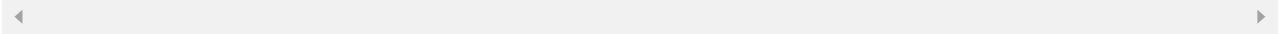
2019-05-07

👍 8

这个专栏应该叫“计算机导论”而不是叫“架构”。。。

展开 ▾

作者回复: 不是的，这个是架构课。你看计算机基础只占了一章，而且我们是从架构角度来讲计算机基础，很多细节都被抽象掉了。



川杰

2019-05-08

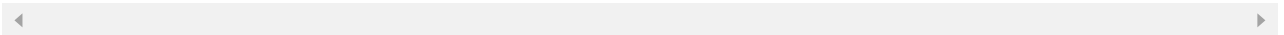
👍 6

老师您好，有两个问题希望解答：1、淘汰的内存页数据保存在哪里？是保存在外置存储设备中吗？2、CPU加载对应程序的代码段到内存中，那么CPU是如何知道这个对应程序的代码段在什么位置的呢？

展开 ▾

作者回复: 1、是的，保存在外置存储中。对于unix系的系统往往是swap分区；windows则是一个隐藏属性的.swp文件。

2、代码段在哪里，是操作系统约定的，因为负责加载的人是操作系统，它设计的程序文件的数据格式。



云学

2019-05-13

👍 5

看前几篇还没共鸣，直到这篇才体会到了这种宏观视角解读的好处，精述每个模块的输入输出以及核心功能，怎么串联到一起，如果从头再来该怎么设计，真正构建出整个计算机知识脉络，看完这些再看其他专栏就如降维打击。

展开 ▾



涵

2019-05-07

👍 5

请问老师内存管理作为操作系统的一个重要且基本的功能，其本身就是一段程序需要在启动时加载在内存中并且永远不被交换到外置存储对吗？那么这段程序在保护机制下是如果一直停留在物理内存中的呢？是写入特定的内存页，永远不会被交换出去，还是由于内存管理功能总是被所以程序调用，由于访问频繁因此永远不会被换出去？另外，操作系统是如何管理内存和寄存器之间的数据交换呢？有点儿想象不出来这个接口。比命令将内存地址1中...

展开 ▾

作者回复: 1、内存管理模块负责内存页的淘汰规则，它当然不会自己把自己淘汰掉。
2、内存和磁盘的数据交换吧？我们这里核心交代思想，要了解细节可以学习一下一些操作系统课程。我后面在第一章的总结与回顾中也会介绍一些参考资料。

◀ ▶



晓凉

2019-05-12

👍 4

从“缺页中断”看架构设计的开闭原则，CPU是硬件，生产出来就不能修改，本身是封闭的，设计上要考虑的是扩展的开放性，缺页中断解决了开放性问题。软件的设计，因为源代码是对程序员开放的，可随便修改，封闭性不能物理上保证，所以需要同时考虑封闭性和开放性。服务化隔离不同的代码模块，一定程度上实现了类似物理上的封闭性。

展开 ▾

作者回复: 挺好的分析

◀ ▶



kirogiyi

2019-05-08

👍 4

真的是很好的架构课程，这这里我们可以了解计算机的发展历史，解析计算机的基础原理，理清软件架构的原始脉络，一步步构建出属于自己的软件架构知识体系，并不会只知其然而不知其所以然，这样构建的软件架构才会更加坚实牢靠。



苟范儿

2019-05-07

👍 4

从硬件架构到其上的操作系统。但是不得不佩服前人在这些层面给出的架构，BIOS 引导、OS、Shell 等系统级方案，仔细想象，每天都在用，各类软件的开发都在这些架构之上很好的运行。

展开 ▾

作者回复: 值得细细体会，感悟

◀ ▶





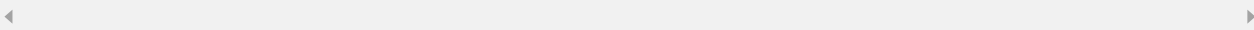
1900
2019-05-07

👍 4

这里说的“虚拟内存”是内存还是磁盘啊？感觉应该还是内存。

展开 ▾

作者回复: 内存和磁盘的配合



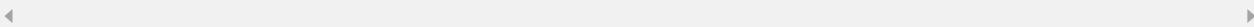
keshawn
2019-05-08

👍 3

表面上是在讲技术，实际上是在分析需求

展开 ▾

作者回复: 😊



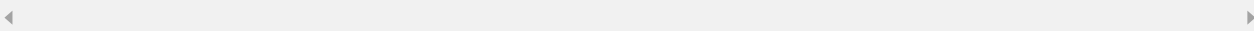
Enthusiasm
2019-05-07

👍 3

老师可以多加一些：按照历史发展的时间顺序来讲技术间因果关系，以便于构建完整的知识体系架构，而细节上可以用给出参考链接或书籍的方式来让我们自己去索骥。比如这一章，由于之前了解过汇编语言，我知道8086CPU一开始是不支持保护模式的，后来386之后才开始支持。明显CPU和操作系统是经历了一段磨合，不是一上来就采用了这种保护模式，CPU也不是一上来就支持保护模式，而操作系统厂商也不是一开始就想到要多任务...

展开 ▾

作者回复: dos面临的个人市场刚开始资源太匮乏了，所以没有做多任务是很正常的决策



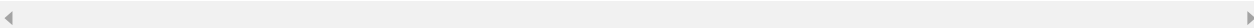
逆流的鱼
2019-05-07

👍 3

如果都是保护模式，比如Java进程，启动指定了堆或者内存大小为4G，那其实内存实际没有分配吗？linux服务器可用内存没有减少4G？运行实际占不到4G？！

展开 ▾

作者回复: 这种疑问，可以实际动手测试试试，验证你的推理对不对





松鼠君

2019-05-07

👍 3

现代操作系统，通过内存管理机制的改变，提升了软件运行的安全性和效率。

展开 ▾



乘风

2019-05-16

👍 2

求许老师帮助

保护模式下非完整读取，意味着运行时动态读取（从磁盘读取），节省了内存空间但出现了运行卡顿的问题，因为内存和磁盘的读取速率相差极大，这个问题如何解决？

作者回复: 1、预读（预测cpu执行）；2、用高速磁盘，现在主流笔记本都用固态硬盘了，也是有这方面原因



Geek_f026c...

2019-05-12

👍 2

缺页异常思想上有点类似依赖注入

展开 ▾



zhangtnt...

2019-05-08

👍 2

许老师, 您的文章很棒。有一点小小的建议, 能否您亲自录音频, 现在的音频老师录得也很好, 但总觉得少了点计算机、软件、架构的一丝丝底蕴（个人浅见）。有时音频对于读者也是很重要的，辛苦老师考虑一下。

展开 ▾

作者回复: 录音频我不太擅长，我给音频老师反馈一下您的意见。



晓明

2019-05-08

👍 2

请问老师怎么保证不同进程通过独立的映射表得到的物理地址相互之间不冲突呢？

作者回复: 操作系统保证的，只有操作系统内核才能修改地址映射表。



涵

2019-05-08

👍 2

第二个问题是关于内存和寄存器(RAM和CPU)的数据交换。确实是如老师所说，有再学一学操作系统的冲动。学着老师的架构课感觉到本科时的课程设计是很好的，可惜当时见识太浅，每门课只是孤立的在学习，为了完成作业拿个好分数而学习，从未整体的去思考课程设计背后计算机的完整架构，真的是很有意思的事情。

展开 ∨

作者回复: 确实，整体去看信息科技世界，里面有太多精彩。非常值得从头再学一遍。加油👊



WadeYu

2019-05-07

👍 2

现在的主流操作系统都是运行在保护模式下吧

展开 ∨

作者回复: 是的，实模式是个历史产物



徐云天

2019-05-07

👍 2

一直不知道虚拟内存为什么有用，现在知道了。

展开 ∨