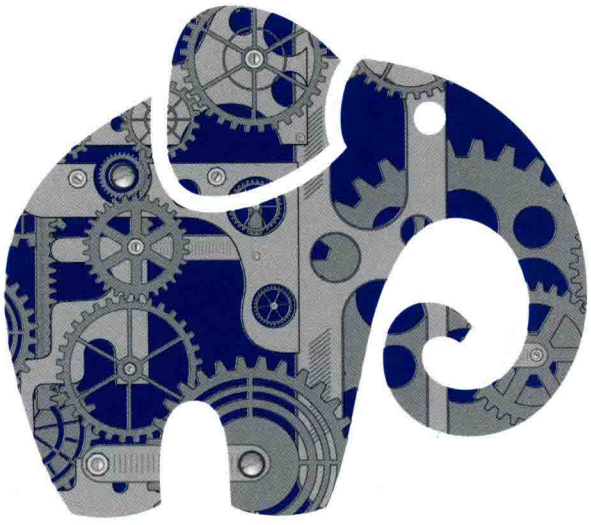
郑钢 ◉ 著

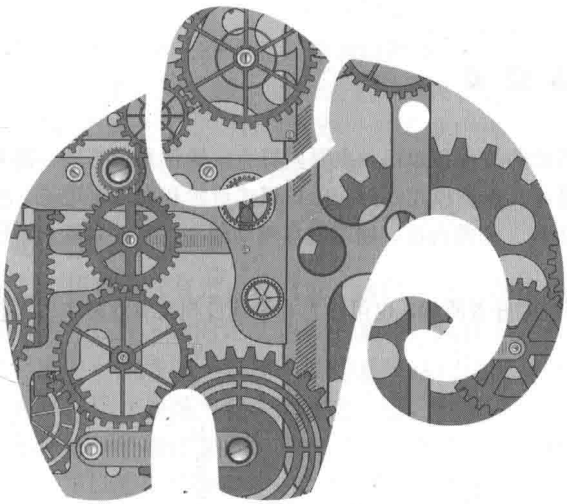
历时19个月，行文60余万字，
用6000多行代码实现了一个完整的操作系统。
彻底剖析操作系统的原理，实现内核线程、特权级变换、
用户进程、任务调度、文件系统等操作系统最基本的组成单元。
用实际代码解释了锁、信号量、生产者、消费者问题。
实现了一个简单的shell，帮助大家理解内部命令、
外部命令、管道等操作。
操作系统并不深奥，本书给予权威解读。



操作系统真象还原

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

郑钢◉著



操作系统真象还原

人民邮电出版社
北京

图书在版编目（CIP）数据

操作系统真象还原 / 郑钢著. -- 北京 : 人民邮电出版社, 2016.3
ISBN 978-7-115-41434-2

Ⅰ. ①操… Ⅱ. ①郑… Ⅲ. ①操作系统－基本知识
Ⅳ. ①TP316

中国版本图书馆CIP数据核字(2016)第016739号

内 容 提 要

本书共分 16 章，讲解了开发一个操作系统需要的技术和知识，主要内容有：操作系统基础、部署工作环境、编写 MBR 主引导记录、完善 MBR 错误、保护模式入门、保护模式进阶和向内核迈进、中断、内存管理系统、线程、输入输出系统、用户进程、完善内核、编写硬盘驱动程序、文件系统、系统交互等核心技术。

本书适合程序员、系统底层开发人员、操作系统爱好者阅读，也可作为大专院校相关专业师生用书和培训学校的教材。

◆ 著 郑 钢
责任编辑 张 涛
责任印制 张佳莹 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市中晟雅豪印务有限公司印刷
◆ 开本：787×1092 1/16
印张：48.25
字数：1 281 千字 2016 年 3 月第 1 版
印数：1－2 500 册 2016 年 3 月河北第 1 次印刷

定价：108.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

前　言

本书面向操作系统基础知识薄弱，但又想把操作系统搞清楚、喜欢刨根问底的技术人，在此向你们致敬，本书用诙谐幽默的语言，把深奥的操作系统尽量讲解清楚，读者在轻松阅读中就学通了深奥的知识，是一本难得的好书。

多数学习操作系统的读者都会有这样的感受：

（1）“太难了，对于操作系统这个庞然大物我简直无从下手”；

（2）“很后悔选了这门课（大学一些专业中操作系统是选修课），甚至不想学习计算机了”；

（3）“上课完全听不懂，我都不想继续听下去了”；

（4）“即使实验做出来了，由于只是完成了局部功能，我依然不明白操作系统是怎样运行起来的，甚至不知道自己在做什么”。

以上的感受我都有过，坦白说，这门课并不是很难，但想把这门课完全搞明白真不容易。我是个喜欢刨根问底的人，为了弄清楚这背后的真相，我花了大量时间学习课程之外的内容，甚至付出了惨痛的代价——大学中第一次考试不及格，操作系统这门课我是第二次才考过的。这确实很“讽刺”——操作系统不及格的人在写操作系统书籍！但转念一想，考试过了的同学并不代表能够写出操作系统，因为试卷上并不是在考如何写一个操作系统。和技术能力相比，卷面成绩并不重要。

想象一下，如果是爱因斯坦那样的天才给我们讲物理知识，我们会觉得物理更容易理解吗？肯定是不会的，因为在爱因斯坦眼中比较容易的内容也许对我们来说非常深奥，他用 B 解释 A 的时候也许会让我们更迷惑，因为 B 我们也不懂，这就是基础的问题了。幸运的是阅读本书时读者只要有 C 语言和部分汇编语言的基础就行了，涉及的其他方面的知识我都会详细介绍，并以更易懂的方式去解释技术难点，读者不必担心看不懂本书。

回忆一下学车的经历：教练让学员先踩离合器再挂档，然后再踩油门，车子就开动啦。如果学员总是学不会这些，有可能是学员根本不知道什么是离合器，或者不知道离合器的作用是什么。即使把这些操作背下来，也会对驾车感到心有余而力不足，可见，只有了解了背后的原理，才会知道自己在做什么，驾车才变得游刃有余。

以上情况对我们学习操作系统来说也同样存在，比如当老师介绍中断发生时的上下文保护时，我们更多的疑问不是如何保存 CPU 的上下文数据，而是想知道为什么在不同的特权级下会使用不同的栈，这背后的原理是什么，并且这是如何做到的。

诸如此类的疑问需要了解硬件原生支持的运行机制，因为很多操作都是硬件自动完成的，比如处理器进入 0 特权级时，会自动在任务状态段 TSS 中获得 0 特权级的栈地址，这不需要人工干涉，完全由处理器维护。我们想知道的是，硬件在背后自动完成了哪些工作，这样才便于我们理解操作系统的全貌。

操作系统受制于硬件的支持，很大程度上它的能力取决于硬件的能力，因此，要想全面理解操作系统，不仅需要了解上层软件的算法、原理、实现，还要了解很多硬件底层的内容。和硬件相关的知识是在微机接口电路中讲解的，而绝大多数读者在学习这门课时，根本不知道它有何用，只有学习操作系统课程时才用到它，因此，本书内容兼顾相关的硬件知识。

除硬件外，本书还把操作系统中的理论付诸于实践，让读者真正学到包含在操作系统中的实实在在的技术，比如在代码中实现了著名的生产者消费者问题，还有进程、线性、阻塞、信号量、锁、文件系统、目录、shell、管道等。各个章节的代码都可独立运行，方便调试，本书更让读者有成就感的是，我们最终完成的一个操作系统总共代码量只有几千行左右，极大地减少了操作系统源码阅读的工作量。

操作系统还是比较庞大的，因此，大部分介绍操作系统原理的书中，对各个部分都是分拆出来介绍的，这导致我们学习操作系统时犹如盲人摸象、管中窥豹。本书的封面是一个完整的大象的拼图，就像封面展示的那样，本书内容我们不再局部学习，而是把所有局部还原成一个整体，做出一个真正的操作系统。

为了让读者不再惧怕操作系统，同时也为了完成我自己的心愿，我辞职专心进行本书的编写，在此期间也曾拒绝了多份回报丰厚的工作，现在想想真是疯狂……苦了我的父母和女朋友，在这里跟你们说声抱歉，你们“纵容”我的偏执，真心不容易，辛苦啦，我爱你们！

感谢我在北京大学就读期间的 Linux 内核课程老师（同时也是我的研究生导师）荆琦教授和操作系统课程老师陈向群教授，很荣幸能够成为您们的学生，时至今日我常常回想起课堂上您们言传身教并为我解答问题的身影，您们渊博的知识和教学上严谨的态度深深影响了我，仅以此书向我这两位恩师致谢。

感谢父母给予我的理解和宽容，以后我一定加倍努力回报您们的养育之恩！

最后，感谢女朋友给予我的陪伴和照顾，在写此书的过程中我深深体会到：爱并不仅仅体现在相信对方一定能成功，更多是体现在支持对方去做想做的事，即使失败了也不会嫌弃。尽管在这漫长枯燥的 19 个月当中，如果没有你的“唠唠叨叨”本书早就写完了，但恰恰是这种“唠唠叨叨”下的不离不弃让我相信这世上还有真爱。

我爱你王小兔（对我女朋友的昵称），本书是我送给你的礼物。

本书中出现的“兄弟”“大伙儿”“同学”和“咱们”的称谓，是作者为了活泼写作风格故意为之，别无他意，在此说明一下。本书读者交流 QQ 群为：148177180，编辑联系邮箱：zhangtao@ptpress.com.cn。

作　者

于北京大学图书馆

目　录

第0章　一些你可能正感到迷惑的问题

正如计算机中数组下标是从0开始的，我们的内容也从0开始，尽量做到低基础学习（负责地说，不是0基础，而且还只是尽量），解释一些学习过程中经常被问到的问题。

0.1 操作系统是什么

我并没有给你提供教科书上对操作系统的定义，因为解释得太抽象了，看了之后似乎只是获得一些感性认识，好奇心强的读者反而会产生更多迷惑。为了说清楚问题，让我给您举个例子。

让我们扯点远的……在盘古开天之际，除动物以外，世界上只有土地、荒草、树木、石头等资源。人们为了躲避天灾、野兽攻击等危险，开始住进了山洞，为了获取食物，用石头和树木等材料打造一些武器。当时所有人都在做这些相同的事。这就是没有组织的人类社会，所有人都在重复“造轮子”。

后来各个地区有了自己权威性的部落，部落都专门找人打造武器，谁需要武器就直接申请领取便可，大部分人不需要自己打造武器了。后来嫌打猎太麻烦了，干脆养一些家畜好了，直接供给人们，谁需要可以过来交换。这就是把大家的重复性劳动集中到了一起，让人们可以专注于自己的事情。

再后来，部落之间为了通信，开始有信使了，这是最原始的通信方式。到后来发展到有社会组织，通信越来越频繁了，干脆搞个驿站吧，谁需要通信，直接写信，由驿站代为送达。

随着人口越来越多，社会组织需要了解到底有多少人，为了方便人口管理，于是就在各地建了“户籍办事”处，人们的生老病死都要到那里登记申报。

说到这我估计您已经猜出我所说的了，上面提到的部落其实就是最原始的操作系统雏形，它将大家都需要的工作找专人负责，大家不用重复劳动。而以上的社会组织其实就是代表现代操作系统，除了把重复性工作集中化，还有了自己的管理策略。

把上面的例子再具体一下，人们想狩猎时，可不可以自己先打造武器，然后拿着自己的武器去狩猎？当然可以，自己制造武器完全没有问题，但部落既然有现成的武器可用，何必自己再费事呢。另外，部落担不担心你随意制造武器会对他人造成伤害？当然会，所以部落不允许你自己制造武器了，人们只有申请的资格，给不给还是要看人家部落的意愿。这就是操作系统提供给用户进程一些系统调用，当用户进程需要某个资源时，直接调用便可，不用自己再费尽心思考虑硬件的事情了，由操作系统把资源获取到后交给用户进程，用户进程可以专注于自己的工作。但操作系统为了保护计算机系统不被损害，不允许用户进程直接访问硬件资源，比如用户进程将操作系统所占据的内存恶意覆盖了，操作系统也就不复存在了，没有操作系统的话，计算机将会瘫痪无法运作。

当人们想和远方的朋友说话时，虽然可以徒步走到亲朋好友身边再对其表达想说的话，但社会组织已经给提供了邮局和电话，何必自己再大老远跑一趟呢。这就是操作系统（社会组织）提供的资源。两个人想在一起生活，要不要一定先结婚呢？完全不用，领不领证都不会阻碍人们在一起生活，但是社会组织为了方便人口管理做了额外约束。不领证的话，至少社会组织无法预测未来人口数量趋势，无法做出宏观调控，甚至这是找到你家人的一种方法。这就如Linux系统中的内存管理，分别要记录哪些页是Active，哪些是“脏页”。不记录会不会影响程序执行，当然不会，记录这些状态还不是为了更好地管理内存吗。

以上说的社会组织和人们之间的关系，正是操作系统和用户进程的关系，希望大家能对操作系统有个初步印象，后面的实践中我们将实例化各个部分。

0.2 你想研究到什么程度

学无止境，学习没有说到头的那天。学习到任何程度都是存有疑惑的，就像中学和大学都讲物理，但学的深度不一样，各个阶段都会产生疑问。我们只是基于一些公认的知识，使其作为学习的起点，并以此展开上层的研究。

比如我对太空很感兴趣，大伙儿都知道地球围绕太阳做周期性公转，后来又知道电子围绕原子核来做周期性公转运动，这和地球绕太阳公转的行为如出一辙，甚至我在想太阳是不是相当于原子核，地球相当于一个电子，我们只是生活在一个电子上……而我们身体里有那么多的原子和电子，对那些我们身体中更为细微的生物来说，我们的身体是不是一个宇宙，无尽的猜想，无尽的疑惑。想法虽然有些荒诞，但基于现有科技目前谁也无法证明这是错的，而且近期已有科学文献证明人的大脑就像个宇宙。如果无止境地刨根问底下去，虽然会对底层科学更加清晰，但这对上层知识的学习非常不利，从而我们需要一个公设，我们认为原子是不可再分的，没有更微小的对象了，一切理论研究以此为基础展开。比如乘法是基于加法的，我们研究 3×4 等于多少，必须要承认 1+1 等于 2，并认为其为真理，不用再去质疑 1+1 为什么等于 2 了，这就是我们的公设，至于为什么 1+1 等于 2，还是由专门研究基础科学的学者们去探究吧。

学习操作系统也一样，不必纠结于硬件内部是如何工作的，我们只要认为给硬件一个输入，硬件就会给我一个输出就行了，因为即使你学到了硬件内部电子电路，随着你不断进步，钻研不断深入，也许有一天你的求知欲到了物理领域，并产生了物理科学方面的质疑……这让我想到一个笑话，某人准备去买自行车，结果被销售人员不断劝说，加点钱就能买摩托啦，等决定买摩托时，销售人员又说既然都决定买摩托车了，不如再加点买汽车吧，给出了各种汽车方面的优势，欲望需求不断升级，不断被销售劝说，最后居然花了几百万元买车，最后才想起自己是来买自行车的，甚至他还没有驾照……于是，咱们赶紧就此打住，我们是来学操作系统的。

你想学到哪个程度呢，你的公设是什么，要不咱们还是走一步说一步吧。

0.3 写操作系统，哪些需要我来做

首先应该明确，在计算机中有分层的概念，也就是说，计算机是一个大的组合物，由各个部分组合成一个系统。每个部分就是一层功能模块，各司其职，它只完成一定的工作，并将自己的工作结果（也就是输出）交给下一层的模块，这里的模块指的是各种外设、硬件。

这样，各种工作成果不断累加，通过这种流水线式的上下游协作，便实现了所谓的系统。可见，系统就是各种功能组合到一起后，产生最终输出的组合物。就像人的身体，胃负责搅拌食物，将这些食物变食糜后交给小肠，因为小肠只能处理流食，所以上游的输出一定要适合作为下游的输入，是不是有点类似管道操作了，哈哈，分工协作是大自然的安排，并不是只有计算机世界才有。我们人类的思想是大自然安排好的，所以人类创造的事物也是符合大自然规律的。

好，赶紧回到正题，操作系统是管理资源的软件，操作系统能做什么，取决于主机上硬件的功能。就像用 Maya 造一个人体模型出来，首先我得知道 Maya 这个软件提供曲线曲面各种建模方法才行，换句话说，对于人体建模，你不可能会想到用 QQ，因为它不是干这个的。我想说的是硬件不支持的话，操作系统也没招……操作系统一直是所谓的底层，拥有至高无上的控制权，一副牛气轰轰的样子，原来也要依仗他人啊。是啊，操作系统毕竟是软件，而软件的逻辑是需要作用在硬件上才能体现出来的。

所以说，写操作系统需要了解硬件，这些硬件提供了软件方面的接口，这样我们的操作系统通过软件（计算机指令）就能够控制硬件。我们需要做的就是知道如何通过计算机指令来控制硬件，参考硬件手册这下少不了啦。

0.4 软件是如何访问硬件的

硬件是各种各样的，发展速度还是非常快的。各个硬件都有自己的个性，操作系统不可能及时更新各

种硬件的驱动方法吧。比如，刚出来某个新硬件，OS 开发者们便开始为其写驱动，这不太现实，会把人累死的。于是乎，便出现了各种硬件适配设备，这就是 IO 接口。接口其实就是标准，大家生产出来的硬件按照这个标准工作就实现了通用。

硬件在输入输出上大体分为串行和并行，相应的接口也就是串行接口和并行接口。串行硬件通过串行接口与 CPU 通信，反过来也是，CPU 通过串行接口与串行设备数据传输。并行设备的访问类似，只不过是通过并行接口进行的。

访问外部硬件有两个方式。

（1）将某个外设的内存映射到一定范围的地址空间中，CPU 通过地址总线访问该内存区域时会落到外设的内存中，这种映射让 CPU 访问外设的内存就如同访问主板上的物理内存一样。有的设备是这样做的，比如显卡，显卡是显示器的适配器，CPU 不直接和显示器交互，它只和显卡通信。显卡上有片内存叫显存，它被映射到主机物理内存上的低端 1MB 的 0xB8000～0xBFFFF。CPU 访问这片内存就是访问显存，往这片内存上写字节便是往屏幕上打印内容。看上去这么高大上的做法是怎么实现的，这个我们就不关心了，前面说过，计算机中处处是分层，我们要充分相信上一层的工作。

（2）外设是通过 IO 接口与 CPU 通信的，CPU 访问外设，就是访问 IO 接口，由 IO 接口将信息传递给另一端的外设，也就是说，CPU 从来不知道有这些设备的存在，它只知道自己操作的 IO 接口，你看，处处体现着分层。

于是问题来了，如何访问到 IO 接口呢，答案就是 IO 接口上面有一些寄存器，访问 IO 接口本质上就是访问这些寄存器，这些寄存器就是人们常说的端口。这些端口是人家 IO 接口给咱们提供的接口。人家接口电路也有自己的思维（系统），看到寄存器中写了什么就做出相应的反应。接口提供接口，哈哈，有意思。不过这是人家的约定，没有约定就乱了，各干各的，大家都累，咱们只要遵循人家的规定就能访问成功。

0.5 应用程序是什么，和操作系统是如何配合到一起的

应用程序是软件（似乎是废话，别急，往后看），操作系统也是软件。CPU 会将它们一视同仁，甚至，CPU 不知道自己在执行的程序是操作系统，还是一般应用软件，CPU 只知道去 cs：ip 寄存器中指向的内存取指令并执行，它不知道什么是操作系统，也无需知道。

操作系统是人想出来的，为了让自己管理计算机方便而创造出来的一套管理办法。

应用程序要用某种语言编写，而语言又是编译器来提供的。其实根本就没有什么语言，有的只是编译器。是编译器决定怎样解释某种关键字及某种语法。语言只是编译器和大家的约定，只要写入这样的代码，编译器便将其翻译成某种机器指令，翻译成什么样取决于编译器的行为，和语言无关，比如说 C 语言的 printf 函数，它的功能不是说一定要把字符打印到屏幕上，这要看编译器对这种关键字的处理。

编译器提供了一套库函数，库函数中又有封装的系统调用，这样的代码集合称之为运行库。C 语言的运行库称为 C 运行库，就是所谓的 CRT（C Runtime Library）。

应用程序加上操作系统提供功能才算是完整的程序。由于有了操作系统的支持，一些现成的东西已经摆在那了，但这些是属于操作系统的，不是应用程序的，所以咱们平时所写的应用程序只是半成品，需要调用操作系统提供好的函数才能完整地做成一件事，而这个函数便是系统调用。

用户态与内核态是对 CPU 来讲的，是指 CPU 运行在用户态（特权 3 级）还是内核态（特权 0 级），很多人误以为是对用户进程来讲的。

用户进程陷入内核态是指：由于内部或外部中断发生，当前进程被暂时终止执行，其上下文被内核的中断程序保存起来后，开始执行一段内核的代码。是内核的代码，不是用户程序在内核的代码，用户代码怎么可能在内核中存在，所以“用户态与内核态”是对 CPU 来说的。

当应用程序陷入内核后，它自己已经下 CPU 了，以后发生的事，应用程序完全不知道，它的上下文环境已经被保存到自己的 0 特权级栈中了，那时在 CPU 上运行的程序已经是内核程序了。所以要清楚，内核代码并不是成了应用程序的内核化身，操作系统是独立的部分，用户进程永远不会因为进入内核态而变身为操作系统了。

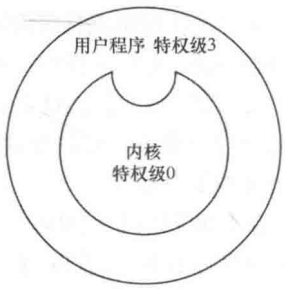
应用程序是通过系统调用来和操作系统配合完成某项功能的，有人可能会问：我写应用程序时从来没写什么系统调用的代码啊。这是因为你用到的标准库帮你完成了这些事，库中提供的函数其实都已经封装好了系统调用，你需要跟下代码才会看到。其实也可以跨过标准库直接执行系统调用，对于 Linux 系统来说，直接嵌入汇编代码“int 0x80”便可以直接执行系统调用，当然要提前设置好系统调用子功能号，该子功能号用寄存器 eax 存储。

会不会有人又问，编译器怎么知道系统调用接口是什么，哈哈，您想啊，下载编译器时，是不是要选择系统版本，编译器在设计时也要知道自己将来运行在哪个系统平台上，所以这都是和系统绑定好的，各个操作系统都有自己的系统调用号，编译器厂商在代码中已经把宿主系统的系统调用号写死了，没什么神奇的。

0.6 为什么称为“陷入”内核

前面提到了用户进程陷入内核，这个好解释，如果把软件分层的话，最外圈是应用程序，里面是操作系统，如图 0-1 所示。

应用程序处于特权级 3，操作系统内核处于特权级 0。当用户程序欲访问系统资源时（无论是硬件，还是内核数据结构），它需要进行系统调用。这样 CPU 便进入了内核态，也称管态。看图中凹下去的部分，是不是有陷进去的感觉，这就是“陷入内核”。

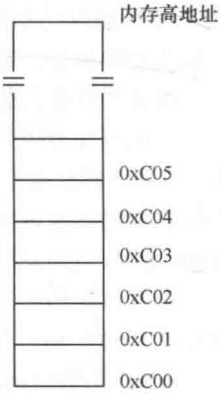


▲图 0-1　陷入内核

0.7 内存访问为什么要分段

按理说咱们应该先看看段是什么，不过了解段是什么之前，先看看内存是什么样子，如图 0-2 所示。

内存按访问方式来看，其结构就如同上面的长方形带子，地址依次升高。为了解释问题更明白，我们假设还在实模式下，如果读者不清楚什么是实模式也不要紧，这并不影响理解段是什么，故暂且先忽略。



▲图 0-2　内存示例

内存是随机读写设备，即访问其内部任何一处，不需要从头开始找，只要直接给出其地址便可。如访问内存 0xC00，只要将此地址写入地址总线便可。问题来了，分段是内存访问机制，是给 CPU 用的访问内存的方式，只有 CPU 才关注段，那为什么 CPU 要用段呢，也就是为什么 CPU 非得将内存分成一段一段的才能访问呢？

说来话长，现实行业中有很多问题都是历史遗留问题，计算机行业也不能例外。分段是从 CPU 8086 开始的，限于技术和经济，那时候电脑还是非常昂贵的东西，所以 CPU 和寄存器等宽度都是 16 位的，并不是像今天这样寄存器已经扩展到 64 位，当然编译器用的最多的还是 32 位。16 位寄存器意味着其可存储的数字范围是 2 的 16 次方，即 65536 字节，64KB。那时的计算机没有虚拟地址之说，只有物理地址，访问任何存储单元都直接给出物理地址。

编译器在编译程序时，肯定要根据 CPU 访问内存的规则将代码编译成机器指令，这样编译出来的程序才能在该 CPU 上运行无误，所以说，在直接以绝对物理地址访问内存的 CPU 上运行程序，该程序中指令的地址也必须得是绝对物理地址。总之，要想在该硬件上运行，就要遵从该硬件的规则，操作系统和编译器也无一例外。

若加载程序运行，不管其是内核程序，还是用户程序，程序中的地址若都是绝对物理地址，那该程序必须放在内存中固定的地方，于是，两个编译出来地址相同的用户程序还真没法同时运行，只能运行一个。于是伟大的计算机前辈们用分段的方式解决了这一问题，让 CPU 采用“段基址+段内偏移地址”的方式来访问任意内存。这样的好处是程序可以重定位了，尽管程序指令中给的是绝对物理地址，但终究可以同时运行多个程序了。

什么是重定位呢，简单来说就是将程序中指令的地址改写成另外一个地址，但该地址处的内容还是原

地址处的内容。

CPU 采用“段基址+段内偏移地址”的形式访问内存，就需要专门提供段基址寄存器，这些是 cs、ds、es 等。程序中需要用到哪块内存，只要先加载合适的段到段基址寄存器中，再给出相对于该段基址的偏移地址便可，CPU 中的地址单元会将这两个地址相加后的结果用于内存访问，送上地址总线。

注意，很多读者都觉得段基址一定得是 65536 的倍数（16 位段基址寄存器的容量），这个真的不用，段基址可以是任意的。这就是段可以重叠的原因。

举个例子，看图 0-2，假设段基址为 0xC00，要想访问物理内存 0xC01，就要将用 0xC00：0x01 的方式来访问才行。若将段基址改为 0xc01，还是访问 0xC01，就要用 0xC01：0x00 的方式来访问。同样，若想访问物理内存 0xC04，段基址和段内偏移的组合可以是：0xC01：0x03、0xC02：0x02、0xC00：0xC04 等，总之要想访问某个物理地址，只要凑出合适的段基地址和段内偏移地址，其和为该物理地址就行了。这时估计有人会问这样行不行，0xC05：-1，能这样提问的同学都是求知欲极强的，可以自己试一下。

说了这么多，我想告诉你的是只要程序分了段，把整个段平移到任何位置后，段内的地址相对于段基址是不变的，无论段基址是多少，只要给出段内偏移地址，CPU 就能访问到正确的指令。于是加载用户程序时，只要将整个段的内容复制到新的位置，再将段基址寄存器中的地址改成该地址，程序便可准确无误地运行，因为程序中用的是段内偏移地址，相对于新的段基址，该偏移地址处的内存内容还是一样的，如图 0-3 所示。

▲图 0-3 段的重定位

所以说，程序分段首先是为了重定位，我说的是首先，下面还有其他理由呢。

偏移地址也要存入寄存器，而那时的寄存器是 16 位的，也就是一个段最多可以访问到 64KB。而那时的内存再小也有 1MB，改变段基址，由一个段变为另一个段，就像一个段在内存中飘移，采用这种在内存中来回挪位置的方式可以访问到任意内存位置。

所以说，程序分段又是为了将大内存分成可以访问的小段，通过这样变通的方法便能够访问到所有内存了。

但想一想，1M 是 2 的 20 次方，1MB 内存需要 20 位的地址才能访问到，如何做到用 16 位寄存器访问 20 位地址空间呢？

在 8086 的寻址方式中，有基址寻址，这是用基址寄存器 bx 或 bp 来提供偏移地址的，如“mov [bx], 0x5;”指令便是将立即数 0x5 存入 ds：bx 指向的内存。

大家看，bx 寄存器是 16 位的，它最大只能表示 0～0xFFFF 的地址空间，即 64KB，也就是单一的一个寄存器无法表示 20 位的地址空间——1MB。也许有人会说，段基址和段内偏移地址都搞到最大，都为 0xFFFF，对不起，即使不溢出的话，其结果也只是由 16 位变成了 17 位，即两个 n 位的数字无论多大，其相加的结果也超不过 n+1 位，因为即使是两个相同的数相加，其结果相当于乘以 2，也就是左移一位而已，依然无法访问 20 位的地址空间。也许读者又有好建议了：CPU 的寻址方式又不是仅仅这一种，上面的限制是因为寄存器是 16 位，只要不全部通过寄存器不就行了吗。既然段寄存器必须得用，那就在偏移地址上下功夫，不要把偏移地址写在寄存器里了，把它直接写成 20 位立即数不就行啦。例如 mov ax，[0x12345]，这样最终的地址是 ds+0x12345，肯定是 20 位，解决啦。不错，这种是直接寻址方式，至少道理上讲得通，这是通过编程技巧来突破这一瓶颈的，能想到这一点我觉得非常 nice。但是作为一个严谨的 CPU，既然宣称支持了通过寄存器来寻址，那就要能够自圆其说才行，不能靠程序员的软实力来克服 CPU 自身的缺陷。于是，一个大胆的想法出现了。

16 位的寄存器最多访问到 64KB 大小的内存。虽然 1MB 内存中可容纳 1MB/64KB=16 个最大段，但这只是可以容纳而已，并不是说可以访问到。16 位的寄存器超过 0xffff 后将会回卷到 0，又从 0 重新开始。20 位宽度的内存地址空间必然只能由 20 位宽度的地址来访问。问题又来了，在当时只有 16 位寄存器的情况下是如何做到访问 20 位地址空间的呢？

这是因为 CPU 设计者在地址处理单元中动了手脚，该地址部件接到“段基址+段内偏移地址”的地址后，自动将段基址乘以 16，即左移了 4 位，然后再和 16 位的段内偏移地址相加，这下地址变成了 20 位了吧，行

啦，有了 20 位的地址便可以访问 20 位的空间，可以在 1MB 空间内自由翱翔了。

0.8 代码中为什么分为代码段、数据段？这和内存访问机制中的段是一回事吗

首先，程序不是一定要分段才能运行的，分段只是为了使程序更加优美。就像用饭盒装饭菜一样，完全可以将很多菜和米饭混合在一起，或者搅拌成一体，哈哈，但这样可能就没什么胃口啦。如果饭盒中有好多小格子，方便将不同的菜和饭区分存放，这样会让我们胃口大开增加食欲。

x86 平台的处理器是必须要用分段机制访问内存的，正因为如此，处理器才提供了段寄存器，用来指定待访问的内存段起始地址。我们这里讨论的程序代码中的段（用 section 或 segment 来定义的段，不同汇编编译器提供的关键字有所区别，功能是一样的）和内存访问机制中的段本质上是一回事。在硬件的内存访问机制中，处理器要用硬件——段寄存器，指向软件——程序代码中用 section 或 segment 以软件形式所定义的内存段。

分段是必然的，只是在平坦模型下，硬件段寄存器中指向的内存段为最大的 4GB，而在多段模式下编程，硬件段寄存器中指向的内存段大小不一。

对于在代码中的分段，有的是操作系统做的，有的是程序员自己划分的。如果是在多段模型下编程，我们必然会在源码中定义多个段，然后需要不断地切换段寄存器所指向的段，这样才能访问到不同段中的数据，所以说，在多段模型下的程序分段是程序员人为划分的。如果是在平坦模型下编程，操作系统将整个 4GB 内存都放在同一个段中，我们就不需要来回切换段寄存器所指向的段。对于代码中是否要分段，这取决于操作系统是否在平坦模型下。

一般的高级语言不允许程序员自己将代码分成各种各样的段，这是因为其所用的编译器是针对某个操作系统编写的，该操作系统采用的是平坦模型，所以该编译器要编译出适合此操作系统加载运行的程序。由于处理器支持了具有分页机制的虚拟内存，操作系统也采用了分页模型，因此编译器会将程序按内容划分成代码段和数据段，如编译器 gcc 会把 C 语言写出的程序划分成代码段、数据段、栈段、.bss 段、堆等部分。这会由操作系统将编译器编译出来的用户程序中的各个段分配到不同的物理内存上。对于目前咱们用高级语言编码来说，我们之所以不用关心如何将程序分段，正是由于编译器按平坦模型编译，而程序所依赖的操作系统又采用了虚拟内存管理，即处理器的分页机制。像汇编这种低级语言允许程序员为自己的程序分段，能够灵活地编排布局，这就属于人为将程序分成段了，也就是采用多段模型编程。

这么说似乎不是很清楚，一会再用例子和大伙儿解释就明白了。在这之前，先和大家明确一件事。

CPU 是个自动化程度极高的芯片，就像心脏一样，给它一个初始的收缩，它将永远地跳下去。突然想到 Intel 的广告词：给你一颗奔腾的心。

只要给出 CPU 第一个指令的起始地址，CPU 在它执行本指令的同时，它会自动获取下一条的地址，然后重复上述过程，继续执行，继续取址。假如执行的每条指令都正确，没有异常发生的话，我想它可以运行到世界的尽头，能让它停下来的唯一条件就是断电。

它为什么能够取得下一条指令地址？也就是说为什么知道下一条指令在哪里。这是因为程序中的指令都是挨着的，彼此之间无空隙。有同学可能会问，程序中不是有对齐这回事吗？为了对齐，编译器在程序中塞了好多 0。是的，对齐确实是让程序中出现了好多空隙，但这些空隙是数据间的空隙，指令间不存在空隙，下一条指令的地址是按照前面指令的尺寸大小排下来的，这就是 Intel 处理器的程序计数器 cs：eip 能够自动获得下一条指令的原理，即将当前 eip 中的地址加上当前指令机器码的大小便是内存中下一条指令的起始地址。即使指令间有空隙或其他非指令的数据，这也仅仅是在物理上将其断开了，依然可以用 jmp 指令将非指令部分跳过以保持指令在逻辑上连续，我们在后面会通过实例验证这一原理。

为了让程序内指令接连不断地执行，要把指令全部排在一起，形成一片连续的指令区域，这就是代码段。这样 CPU 肯定能接连不断地执行下去。指令是由操作码和操作数组成的，这对于数据也一样，程序运行不仅要有操作码，也得有操作数，操作数就是指程序中的数据。把数据连续地并排在一起存储形成的

段落，就称为数据段。

指令大小是由实际指令的操作码决定的，也就是说 CPU 在译码阶段拿到了操作码后，就知道实际指令所占的大小。其实说来说去，本质上就是在解释地址是怎么来的。这部分在第 3 章中的“什么是地址”节中有详解。

给大家演示个小例子，代码没有实际意义，是我随便写的，只是为方便大家理解指令的地址，代码如下。

code_seg.S

```
1      mov ds,ax
2      mov ax,[var]
3  label:
4      jmp label
5      var dw 0x99
```

本示例一共就 5 行，简单纯粹为演示。将其编译为二进制文件，程序内容是：

```
8E D8 A1 07 00 EB FE 99 00
```

就这 9 个字节的内容，有没有觉得一阵晕炫。如果没有，目测读者兄弟的技术水平远在我之上，请略过本书。

其实这 9 个字节的内容就是机器码。为了让大家理解得更清晰，给大家列个机器码和源码对照表，见表 0-1。

表 0-1　　机器码和源码对照表

地　址	机 器 码	源　码
00000000	8ED8	mov ds，ax
00000002	A10700	mov ax，[0x7]
00000005	EBFE	jmp short 0x5
00000007		var dw 0x99
00000008		

表 0-1 第 1 行，地址 0 处的指令是“mov ds，ax”，其机器码是 8ED8，这是十六进制表示，可见其大小是 2 字节。前面说过，下一条指令的地址是按照前面指令的尺寸排下来的，那第 2 行指令的起始地址是 0+2=2。在第 2 行的地址列中，地址确实是 2。这不是我故意写上去的，编译器真的就是这样编排的。第 2 列的指令是“mov ax，[0x7]”（0x7 是变量 var 经过编译后的地址），其机器码是 A10700，这是 3 字节大小。所以第 3 条指令的地址是 2+3=5。后面的指令地址也是这样推算的。程序虽然很短，但麻雀虽小，五脏俱全，完美展示了程序中代码紧凑无隙的布局。

现在大伙儿明白为什么 CPU 能源源不断获取到指令了吧，如前所述，原因首先是指令是连续紧凑的，其次是通过指令机器码能够判断当前指令长度，当前指令地址+当前指令长度=下一条指令地址。

上面给出的例子，其指令在物理上是连续的，其实在 CPU 眼里，只要指令逻辑上是连续的就可以，没必要一定得是物理上连续。所以，明确一点，即使数据和代码在物理上混在一起，程序也是可以运行的，这并不意味着指令被数据“断开”了。只要程序中有指令能够跨过这些数据就行啦，最典型的就是用 jmp 跳过数据区。

比如这样的汇编代码：

```
1    jmp start        ;跳转到第三行的 start，这是 CPU 直接执行的指令
2    var dd 1         ;定义变量 var 并赋值为 1。分配变量不是 CPU 的工作
                      ;汇编器负责分配空间并为变量编址
3    start:           ;标号名为 start，会被汇编器翻译为某个地址
4    mov ax,0     ; 将 ax 赋值为 0
```

这几行代码没有实际意义，只是为了解释清楚问题，咱们只要关注在第 2 行的定义变量 var 之前为什么要 jmp start。如果将上面的汇编代码按纯二进制编译，如果不加第 1 行的 jmp，CPU 也许会发出异常，显示无效指令，也许不知道执行到哪里去了。因为 CPU 只会执行 cs：ip 中的指令，这两个寄存器记录的是下一条待执行指令的地址，下一个地址 var 处的值为 1，显然我们从定义中看出这只是数据，但指令和数据都是二进制数字，CPU 可分不出这是指令，还是数据。保不准某些“数据”误打误撞恰恰是某种指令也说不定。既然 var 是我们定义的数据，那么必须加上 jmp start 跳过这个 var 所占的空间才可以。