卷3

大话计算机

冬瓜哥◎著

计算机系统底层架构原理极限剖析

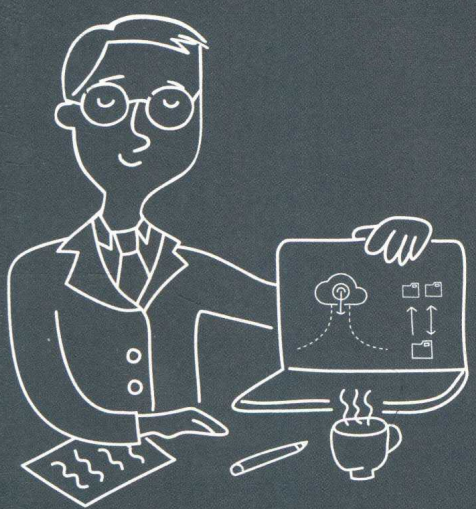
清華大學出版社

大话计算机

计算机系统
底层架构原理极限剖析

冬瓜哥◎著

中国科学技术大学
图书馆藏



清華大學出版社
北京

内 容 简 介

现代计算机系统的软硬件架构十分复杂，是所有IT相关技术的根源。本书尝试从原始的零认知状态开始，逐步从最基础的数字电路一直介绍到计算机操作系统以及人工智能。本书用通俗的语言、恰到好处的疑问、符合原生态认知思维的切入点，来帮助读者洞悉整个计算机底层世界。本书在写作上遵循“先介绍原因，后思考，然后介绍解决方案，最终提炼抽象成概念”的原则。全书脉络清晰，带领读者重走作者的认知之路。本书集科普、专业为一体，用通俗详尽的语言、图表、模型来描述专业知识。

本书内容涵盖以下学科领域：计算机体系结构、计算机组成原理、计算机操作系统原理、计算机图形学、高性能计算机集群、计算加速、计算机存储系统、计算机网络、机器学习等。

本书共分为12章。第1章介绍数字计算机的设计思路，制作一个按键计算器，在这个过程中逐步理解数字计算机底层原理。第2章在第1章的基础上，改造按键计算器，实现能够按照编好的程序自动计算，并介绍对应的处理器内部架构概念。第3章介绍电子计算机的发展史，包括芯片制造等内容。第4章介绍流水线相关知识，包括流水线、分支预测、乱序执行、超标量等内容。第5章介绍计算机程序架构，理解单个、多个程序如何在处理器上编译、链接并最终运行的过程。第6章介绍缓存以及多处理器并行执行系统的体系结构，包括互联架构、缓存一致性架构的原理和实现。第7章介绍计算机I/O基本原理，包括PCIE、USB、SAS三大I/O体系。第8章介绍计算机是如何处理声音和图像的，包括3D渲染和图形加速原理架构和实现。第9章介绍大规模并行计算、超级计算机原理和架构，以及可编程逻辑器件（如FPGA等）的原理和架构。第10章介绍现代计算机操作系统基本原理和架构，包括内存管理、任务调度、中断管理、时间管理等架构原理。第11章介绍现代计算机形态和生态体系，包括计算、网络、存储方面的实际计算机产品和生态。第12章介绍机器学习和人工智能底层原理和架构实现。

本书适合所有IT行业从业者阅读，包括计算机（PC/服务器/手机/嵌入式）软硬件及云计算/大数据/人工智能等领域的研发、架构师、项目经理、产品经理、销售、售前。本书也同样适合广大高中生科普之用，另外计算机相关专业本科生、硕士生、博士生同样可以从本书中获取与课程教材截然不同的丰富营养。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

大话计算机：计算机系统底层架构原理极限剖析 / 冬瓜哥著. — 北京：清华大学出版社，2019（2019.6重印）
ISBN 978-7-302-52647-6

Ⅰ. ①大… Ⅱ. ①冬… Ⅲ. ①计算机系统—基本知识 Ⅳ. ①TP303

中国版本图书馆CIP数据核字（2019）第045444号

责任编辑：栾大成
封面设计：杨玉芳
版式设计：方加青
责任校对：徐俊伟
责任印制：杨 艳

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京亿浓世纪彩色印刷有限公司
经 销：全国新华书店
开 本：188mm×260mm 印 张：96.25 字 数：3546千字
（附海报11张）
版 次：2019年5月第1版 印 次：2019年6月第2次印刷
定 价：598.00元（全三册）

产品编号：082577-01

目 录

第10章 计算机操作系统——舞台幕后的工作者

试读结束：需要全本请在线购买： www.ertongbook.com

第11章 现代计算机系统——形态与生态

第12章 机器学习与人工智能

尾声 狂想计算机——以创造者的名义

后记

随着本书行进至此，我们已经将整个计算机系统建立了一个基本框架，大家目前应该已经可以深刻理解CPU是怎样运行的（第1、2章），数百亿晶体管组成的复杂电路是如何制造出来的（第3章），CPU内部是如何更加优化地执行代码的（第4章），代码是怎么编写并被编译成机器码的，以及程序之间是如何形成各种层级的（第5章），多核心多处理器是如何运行的（第6章），各种外部I/O设备是如何连接到系统中并工作的（第7章），计算机是如何处理声音和图像的（第8章），超级计算到底是怎么计算的（第9章）。

现在是时候将所有这些事物统一地管理、使用起来了。在第5章最后部分，曾提及操作系统的概念，计算机需要有一个操作系统来管理底层资源，这是程序之间分工导致的必然结果，建议读者在阅读本章之前，重新回顾一下第5章的最后部分。

如果说CPU是一个舞台，程序是利用舞台表演的人，那么操作系统就是舞台背后的后勤工作者们。这个后勤团队需要提供：如何发现各种外部设备以及驱动装载过程的管理、Loader/人机交互界面CLI/GUI、市面上主流外部设备的驱动程序、虚拟分页内存分配和管理、文件系统、多线程轮流执行和调度管理、为程序之间相互通信提供支撑、响应各种中断的中断服务程序，以及上面所有这些程序对应的数据结构；用于封装网络包的网络协议栈，比如TCP/IP等；用于封装存储I/O指令的存储协议栈，比如SCSI和NVMe等；供用户态程序执行系统调用时的接口函数；以及一些内置的方便用户管理整个系统的小程序，比如计算器、媒体播放器、压缩解压缩、字处理、文件浏览管理器、网页浏览器等，当然，这些小程序都属于用户态程序，可以将它们删掉，而用自己喜欢的程序替换掉；最后，还需要提供用户认证、多用户等安全功能。

程序员期待着设计良好的操作系统，因为它可以更加方便地利用OS提供的系统调用来获取各种服务，而不用自己去自底向上地实现全套程序。

10.1 内存布局与管理

我们在第5章中大致介绍了一下内存管理方面的基本思想，也就是采用虚拟地址空间的方式，以一个Page（通常定为4KB）为粒度，将虚拟地址空间中的虚拟页映射到物理地址空间的物理页中去，让程序存在于一个由多个虚拟页连续拼接起来的虚拟地址空间中。本节我们就来详细介绍一下内存管理模块。

10.1.1 实模式与保护模式

如图10-1（a）所示的场景为一个预先被载入内存的操作系统，其被放置到物理内存的最底端处，物理内存其余部分留给用户程序使用。操作系统使用内置的Loader程序载入用户程序的可执行文件，对其进行动态链接、地址修正和重定位等操作之后，将其载入到用户内存区执行。如图10-1（b）所示的场景为直接将操作系统固化到BIOS ROM中，CPU直接从ROM芯片中读取代码执行。BIOS本身其实就是一个极其简化的操作系统。有些嵌入式系统比如电动玩具机器人等或许会使用这种方式，因为ROM相比RAM要廉价，虽然读取速度并不如RAM，但是对于这些系统而言已经足够；同时由于这些嵌入式系统都是专用的固定场景，其硬件规格、连接方式等都是固定的，不需要做到灵活适配，所以直接使用BIOS充当操作系统即可。如图10-1（c）所示的场景为同时具有位于ROM中的BIOS和位于RAM中的操作系统，该场景适用于一些需要灵活配置、更加通用的场景，比如个人计算机，BIOS ROM的容量和速度不足以支持拥有较强功能和灵活性的PC操作系统，所以BIOS内部的极简系统仅供在启动计算机初期使用，包括准备好中断向量表、设备信息描述表、加载对应设备的驱动驻留到内存等步骤，然后通过读取硬盘的0扇区引导记录，读出操作系统的bootloader程序执行，由后者负责将位于硬盘上的操作系统代码和数据一步步加载到内存然后最终执行操作系统相关的初始化准备程序准备好操作系统自身使用的大量数据结构，最后整个操作系统被载入RAM的低端地址区域驻留，并执行Loader程序，提供GUI或者CLI与操作员交互，从而加载其他用户程序执行。操作系统可以完全不再依赖BIOS，也可以选择仍然调用BIOS提供的一些驱动程序或者服务函数。为了兼容性和灵活性方面的考虑，现代的PC操作系统普遍都依赖BIOS，这样无论底层硬件规格有何变化，操作系统都不需要重新设计，只需要改变BIOS的设计即可。

DOS操作系统就符合上述的图10-1（c）场景。三个场景都只能运行单进程，无法做到多进程同时执行，因为对应的OS/BIOS并没有将RAM进行分割，所以只能承载一个用户程序执行，该程序退出后返回到

操作系统的Loader程序GUI/CLI，操作员可以启动其他用户程序继续执行。所以DOS是一个**单任务操作系统**。另外，DOS操作系统下的程序可以访问整个物理地址空间的任意位置，因为DOS并没有去限制程序能访问的范围。

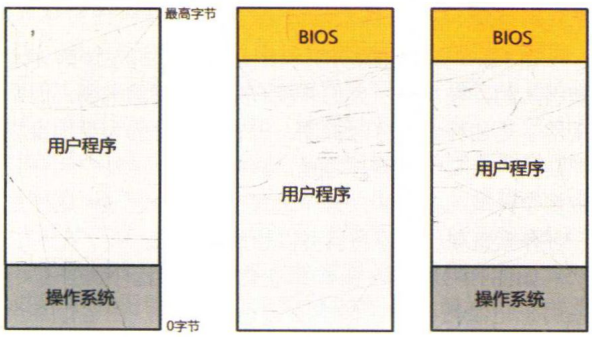


图10-1　早期操作系统的内存分布方式

这里禁不住要问了，所谓“DOS没有限制程序的访存范围”，这意思难道暗指DOS是可以去限制的？怎么限制？当操作系统的Loader程序让CPU跳转到用户程序运行之后，整个CPU就是在运行用户程序了，CPU此时完全受到用户程序代码的控制，让它走东绝不往西，此时操作系统代码只是静静地待在内存里起不到任何作用，此时只有靠CPU来检查和防止越界。要想实现这个功能，必须将当前执行的代码可访问的地址范围限制在某个区域中，比如从地址1024开始的长度2048B的这2KB区域中。为了支持这个功能，CPU必须提供至少两个寄存器，一个用于存放该区域的基地址，也就是上述的地址1024，另一个用于存放长度，也就是上述的2048。在操作系统Loader程序跳转到用户程序执行之前，必须使用对应的机器指令来更新这两个寄存器，告诉CPU：“兄弟，后续任何代码只能在这个区域内执行，一旦越界你就报异常，反过来执行我提供的异常服务程序”，然后再跳转到用户程序执行，此时用户程序就被框住了。然而，这一招只能防君子，却防不了小人。程序（或者说黑客们）是不会善罢甘休的，程序是不是也可以用对应的机器指令来更新这两个寄存器呢？比如将基地址更新为0，长度更新为1GB，从而逃脱限制？这就相当于马路上有个栏杆，守规矩的司机碰到栏杆就会避让，但是不守规矩的可以下车把栏杆往边上移动一下然后说：“你看，我没违规啊”。

设计者早就考虑到这个问题了，CPU的指令集中有一些属于特权指令，只有操作系统的代码可以执行特权指令。任何尝试执行特权指令的用户程序，CPU会直接中断程序的运行，跳转到异常服务程序执行，后者直接把该程序终止掉，并弹出窗口或者文字提示“刚才这个程序不老实想搞事情，被终止了”，当然，说得好听一点儿是“非法操作：尝试访问了×××地址”。更新基地址和长度寄存器的指令，是特权指令，用户程序如果执行特权指令，CPU就会自动报告异常。所以，这一招下去，相当于接下来执行的程序再也无法逃脱出这个框框了。

提示 ▶▶▶

x86处理器的特权指令有：LGDT — Load GDT register；LLDT — Load LDT register；LTR — Load Task Register；LIDT — Load IDT register；MOV (Control Registers) — Load and store control registers；LMSW — Load Machine Status Word；CLTS — Clear task-switched flag in register CR0；MOV (debug registers) — Load and store debug registers；INVD — Invalidate cache, without writeback；WBINVD — Invalidate cache, with writeback；INVLPG —Invalidate TLB entry；HLT— Halt processor；RDMSR — Read Model-Specific Registers；WRMSR —Write Model-Specific Registers；RDPMC — Read Performance-Monitoring Counter；RDTSC — Read Time-Stamp Counter。

用户程序执行完退出后，返回到操作系统代码执行，此时必须有某种机制让CPU从之前的禁止执行特权指令，切换到可以执行特权指令。而操作系统决定执行某个用户进程的代码前，也必须先禁止特权指令的执行权。对于Intel的CPU体系，人们将用户程序运行时所处的权限级别称为Ring3，而将操作系统程序的运行权限级别称为Ring0，若CPU处于Ring0级别下，则其可以不受限制地执行任何指令。程序主动退出时，会调用指定的函数，比如exit()，该函数由操作系统提供，该函数内部会执行系统调用，也就是第5章中5.5.6.4节所述的Int 80软中断指令，CPU执行该指令后，会将权限级别切换到Ring0，也就是进行权限提升操作，具体的提升详见10.3节。

程序如果异常退出，比如遇到错误或者尝试越界，此时CPU会跳转到异常服务程序中执行，这也属于一种中断。记住，只要是中断，不管是程序主动发起的Int软中断指令导致的中断还是CPU自行中断，或是外部设备用电信号硬中断强行中断CPU，中断之后CPU查询中断向量表取出对应的中断向量入口信息后，会根据对应的信息自动将权限级别提升到Ring0（具体过程后文介绍），然后跳转到对应的中断服务程序执行。那你禁不住要问了，如果某个程序把中断向量表给改了，替换成自己的黑客程序，当再中断时就会运行该程序，此时该程序在Ring0权限，所以可以做任何事情，不就完成侵入了么？是的，但是一个用户态程序在操作系统没有漏洞的前提下，是无论如何也改不了中断向量表的，因为操作系统会给你限定访存范围，即使程序知道了中断向量表在哪个地址上，也只能眼看着却碰不到。程序能否跟CPU走个后门串通？不行，除非CPU内部真的有某种奇葩bug。

上述使用访存范围寄存器+权限控制来实现将

用户程序关在笼子里运行的做法，被称为保护模式（Protection Mode）。而不限制程序的访存范围则被称为实模式（Real Mode）。DOS是一个实模式下的操作系统，也就是说，它并没有使用CPU提供的访存范围寄存器以及权限级别功能，而且DOS刚问世时是运行在Intel 8080 CPU上的，该CPU不支持保护模式，一直到1982年的Intel 80286 CPU才开始支持保护模式，但是微软是在8年后的1990年推出的第一个支持保护模式的Wndows版本——Windows 3.0，然而其并不支持80286 CPU，直接使用了80386。

你自然会想到，如果CPU和操作系统都支持了保护模式，就可以支持多任务同时安全地被执行了，只要给每个进程设置并记录对应的访存范围，让这多个访存范围位于内存的不同物理区域，不重叠，支持这种模式的操作系统就属于多任务操作系统。在运行某个任务之前，将该任务被分配的访存范围基地址和长度更新到对应的寄存器，然后根据该基地址对程序中的绝对地址引用进行地址修正（基地址重定向，见第5章），然后跳转过去直接执行即可。当决定切换到其他任务执行时，将当前任务所运行到的位置（也就是PC指针）以及各种栈指针寄存器等保存下来到一张表中，然后载入其他任务的PC、栈指针等寄存器以及访存范围寄存器，跳转执行即可。这就可以实现多任务轮流执行，轮流的时间间隔取决于时钟中断频度以及调度算法了，这在上文中略有介绍。这种将物理内存分隔成多个区域的方式被称为分区式内存管理。

提示 ▶▶▶

如果不采用分区机制并不意味着只能实现单任务，也可以通过其他笨办法实现多任务。比如，当决定暂停当前任务而切换到另一个任务时，将该任务占用的内存以及相关寄存器值全部复制到硬盘上，然后载入另一个任务执行，下次切换时再将之前任务的内存数据整体从硬盘上复制回内存，然后载入之前保存的寄存器值，然后继续执行。这种做法被称为Swap（交换）。其切换时将会非常慢，因为硬盘很慢。

10.1.2 分区式内存管理

第5章中介绍过的分页机制是在计算机发展后期才被引入的。在早期，人们并没有采用分页的机制，而是采用了更加朴素的分区机制。说它朴素是因为更加直观和简洁，上文中也曾思考过，直接把内存地址空间分隔成多个区域（分区），每个进程占用一个区域，这便是一种朴素直观，很容易就可以想到的方法。

如图10-2所示，操作系统内存管理程序维护了一张用于记录内存分区的内存分区记录表，在运行某个程序之前，操作系统的Loader程序分析该用户程序的初始内存耗费情况，然后为其分配对应大小的内存分区，做地址修正后开始运行。程序退出后，该分区会被标记为空闲。当多个程序轮流运行之后，难免产生内存分区碎片，如果某个程序耗费的内存大于任何一个空闲分区容量，则无法运行，所以操作系统内存管理程序需要实现相应的空闲分区合并机制，相当于文件系统的碎片整理功能，只不过文件系统就算不整理碎片也能继续存入文件，只要总空闲容量足够即可，而内存要求必须是连续的才能容纳一个程序，这也是为何后来人们改为采用分页机制管理内存的最主要的原因。图中右侧可以看到一个用于追踪各个进程状态、记录之前执行过但是被从CPU卸下或者说从CPU被调度下来的进程的各种运行现场信息，该表由操作系统的进程调度程序来维护。其中，显示进程E将要被重新调度到CPU上执行，其之前被进程调度程序从#1分区整体被Swap到了硬盘上，进程调度程序决定重新继续运行E进程，所以将其数据再Swap回#1分区，但是尚未开始执行，因为目前该CPU上的4个核心已经被A、B、C、D4个进程占用了，所以E进程被置为Ready状态。而F和G进程也被Swap到了硬盘上，但是

分区ID	分区基地址（扇区号）	分区长度（扇区数）	状态	进程名和ID
0	100	400	占用	A, 000
1	500	300	空闲	
2	800	300	占用	B, 101
3	1100	100	空闲	
4	1200	100	占用	C, 110
5	1300	100	空闲	
6	1400	100	占用	D, 111

进程名和ID	被保存的PC	被保存的栈指针	被保存的其他寄存器	所在分区ID	运行状态
A , 000				0	Running
B , 101				2	Running
C , 110				4	Running
D , 111				6	Running
E , 001	512	788	.....	1	Ready
F , 010	555	799	.....	1	Swapped
G , 100	486	780	.....	1	Swapped

图10-2 朴素的内存分区管理方式

由于其对内存耗费比较大，而此时只有#1分区有足够容量，同时内存管理程序还没来得及对内存空闲碎片进行合并，所以F和G只能等待E运行一段时间之后被调度执行。

这种分区方式有个问题就是程序运行之后所动态申请的内存，也就是Heap（堆），只能被限定在其所被分配的分区之内，无法被分配到分区外面，因为CPU会根据分区长度寄存器来限定该程序访存范围。但是，Loader程序一开始似乎并不会知道该用户程序会申请多少内存（比如程序根据某些判断条件来决定申请多少内存，比如用户每按一次某个键就申请一部分内存，这根本无法预先判断），所以Loader只会根据该程序的静态内存耗费容量来分配分区，那么该程序将无法动态要求新分配内存。解决办法是要么Loader为每个程序在其分区内预留一部分堆内存空间，要么就需要改变CPU的寄存器设置，新增一个用于追踪当前进程可访问的堆内存空间的基地址和长度寄存器，并且每次为该进程分配了堆内存之后都要更新这两个寄存器，从而让CPU放开该进程对堆空间的访问。

历史上IBM System/360 Operating System，UNIVAC 1108，Burroughs Corporation B5500，PDP-10以及GE-635等计算机系统的操作系统使用了分区式内存管理方式。不过由于分区式管理在进程数量较少、尺寸较少并且比较均匀的时候也算合理，但是随着进程数量猛增、进程大小不一的时候，会带来很大的管理开销，所以后续该方式已经无人使用，被分页管理方式取代了。

10.1.3　8086分段+实模式

在第5章中的图5-68中给出了一个程序文件的结构示意图，程序文件内部是分段的，包含代码段、数据段以及其他一些没有介绍的段，分段的一个重要原因是为了让缓存命中率足够高，因为缓存对空间和时间局部性较高的场景才具有更高命中率，所以将程序中相似、相邻的信息都集中在一起，便形成了段。

分段式内存管理的初衷，就是将内存的分配与程序中的这些段联系起来，每个段分配一块内存区域，对应的内存区域也称为段。Intel最早在其8086 CPU上使用了分段机制。8086 CPU内部设置了对应的段基地址寄存器（都是16位），包括CS（Code Segment）、DS（Data Segment）、SS（Stack Segment）和ES（Extension Segment），此外还有FS（Flag Segment）和GS（Global Segment）段基地址寄存器。但是，其没有提供长度寄存器，这一点导致8086无法实现保护模式，也就意味着，程序可以访问任意内存地址。但是即便如此，也能够提升内存的利用率，协助实现多任务，比如一个程序可以按照段为粒度被切分放置在内存的不同区域，不必连续；多个不同进程对应的程序可以被切分为段，见缝插针，如图10-3所示。

图10-3　分段示意图

这样，系统最大可以支持2^{16}=65536个段。操作系统的Loader程序为用户程序分配好对应的段之后，会将各自段基地址采用对应指令比如lds（Load DS）、lss等载入相应寄存器（没有lds指令，更改DS寄存器需要使用Jmp类指令），在做完另外一些准备以及地址重定位修正之后，便跳转到该程序入口地址执行。**在后续的执行过程中，CPU会自动将下一条指令的PC指针与CS寄存器保存的基地址相加，将得到的结果作为最终访存地址**，因为程序的代码段被整体搬移到了以CS基地址开始的段中，所以PC指针就成为基于CS段的Offset偏移量而存在，如果PC=4，则此时需要从物理地址的CS+4处取回代码执行。

对于访存指令，CPU会自动将指令中给出的地址与DS寄存器中的基地址相加得出最终的物理地址来访存，比如对于8086 CPU的一条汇编指令：mov ax [si]，其含义为将si寄存器中存储的值作为指针来寻址内存，将取回的数据写入ax寄存器，该指令语法与冬瓜哥在第2章中给出的指令集描述方式是不同的，但是本质都相同，也希望大家不要被冬瓜哥的自创指令集所迷惑。CPU执行该指令时，会用DS中的基地址+si中的地址，用得出的地址来访存。

那么ES寄存器又是干什么用的呢？如果指令为mov ax [di]，则CPU会默认用ES中的基地址+di中的值作为最终访存地址。ES其名称为扩展段，意思就是这个，也就是提供除了DS段之外的额外的数据存储空间。在基于8086 CPU的MS-DOS操作系统下，提供了malloc和farmalloc函数，供程序调用以分配内存，其中，malloc函数只返回一个offset，说明DOS为当前程序分配的内存就处于当前DS寄存器中的基地址所表示的段中，所以无须返回段基地址；但是farmalloc函数会返回一个新分配的段基地址和一个offset，此时程序可以将这个新的段地址写到DS中，此时CPU就会切换到以这个新的段为基准来寻址后续访存动作，但是

有时候程序既要使用之前DS段的内容，又要使用新分配段中的内容，ES额外的段基地址就为此而生，程序可以将新分配的段基地址写入ES寄存器，访存时使用di寄存器来盛放offset，CPU会自动以ES基地址与其相加，从而访存。所以，CPU被设计为si寄存器与DS默认对应，di寄存器与ES默认对应。当然，也可以用segment：offset的形式强行指定，比如：mov ax es:[si]或者mov ax ds:[di]，那CPU就会用指定的基地址来与offset相加。

由于程序在运行过程中可以擅自改变各个段基地址寄存器中的值，比如用户程序可以执行lds或者mov指令来改变DS寄存器值，由于8086 CPU并没有实现保护模式，没有设置访存范围，所以程序可以肆无忌惮地访问到任何地址上的数据，正因如此，不可靠的程序会导致系统崩溃，以及各种病毒程序泛滥。

由于8086 CPU并没有主动限制段的长度，所以每个段可以是任意长度，当然有个最大值，也就是64KB。因为offset的值，也就是程序中给出的访存地址的值是16位，2^{16}=64KB。8086 CPU的内部寄存器以及数据总线位宽为16位，但是地址总线却有20位，这样，其可寻址的最大字节数为2^{20}B=1MB。然而由于其内部寄存器为16位，给不出20位的地址，于是8086这样来设计，针对每个段基地址，强行将其左移4位，也就是在16位值的尾部填上4个0从而变成20位，比如原本某个段的基地址为2000h，强行左移后会变为20000h，然后用该值与代码中给出的offset值相加，比如20000h+1F60h=21F60h，用该地址作为物理地址来访存。但是这样做之后，段基地址的数量最大依然是64kB个，但是再加上16位的offset一起，就可以将1MB的地址空间全覆盖了。

提示 ▶▶▶

8086汇编指令集中有几种不同的跳转指令，比如短跳转（跳转距离在-128～127B）、近跳转（跳转距离在-32～32KB）以及长跳转（跳转目标地址与当前处在不同段中）。短跳转指令只需要携带1B长度的地址即可，因为8位足以描述256B的范围；近跳转需要携带2B的地址来描述64KB（2^{16}，2B为16位）的距离范围。但是短跳和近跳都属于内跳转，也就是跳转目标地址必须处在当前的CS段内部某处，如果跨了段，必须用长跳转或者说远跳转指令，该指令会在后台自动将跳转目标地址所在的段的基地址导入到CS寄存器中，然后从目标地址开始执行。

如果CS段基地址为FFFFh，offset为FFFFh，那么前者左移4位之后为FFFF0h，与后者相加=10FFEFh，由于8086的CPU地址线只有20位，10FFEFh中的最高的4位：0001，会溢出而被丢弃，只保留0FFEFh这20位，而地址0FFEFh表示的是64KB-16B字节处。其实，F800h+8000h=100000h，就已经溢出了，实际送到地址线上的信号为00000h，也就是访问1MB空间中的第一个字节。所以，程序以及编译器必须熟知这一点，否则会得到错误的结果，或者误覆盖数据导致崩溃。这个不能完全算作bug的问题，引发了一段奇葩历史事件。

提示 ▶▶▶

正犹如没有段子手编不出来段子的事件，也永远不要低估程序员们的调皮。在当时，有些程序竟然利用了这个“特性”来实现一些特殊效果，比如提升性能。他们设置了一个段：F800:0000～F800:FFFF，显然，这个段的前半部分，也就是F800:0000～F800:7FFF区间，也就是物理地址0x000F8000～0x000FFFFF区间，其位于1MB的末尾处，程序员将自己的程序代码放置到这里；另一半，F800:8000～F800:FFFF区间，溢出折回到地址0处继续开始，也就是对应了物理地址的0x00000000～0x00007FFF区间，而这里恰好存放的是操作系统维护的一些关键数据，其中有一些是I/O缓冲区，比如键盘缓冲区等，这样，程序可以在不切换CS段基地址的情况下，既运行自己的代码，又访问这些缓冲区，比如将操作键盘的程序放置在前半部分，这样就避免了段基地址寄存器切换，可以节省指令，提升性能。要知道，在当时，“免费”得到了可榨取哪怕一丁点儿性能的方法，都能让程序员满足很长时间。就连当时的DOS操作系统都使用了该“特性”来做一些事情。然而，程序员们沾沾自喜没多久，Intel就发布了80286 CPU，其具备24根地址线，这下好了，之前的地址不会再溢出了，因为第21根信号线（如果从0开始算的话应该是Address #20号地址线，简称A20）终于存在了。

80286号称兼容8086程序，但是，其设计者或者是真的不知道程序员们之前所做的，或者是忘记了，既然兼容8086，那么就应该在兼容模式下自动将A20信号线强制为0，但是设计者却并没有实现这个机制。这样，当之前的8086上的程序真的采用溢出的地址来访存时，却真的得到了“正确”的地址，而不是溢出后折回到0～（64K-16）B区间的地址，结果这些之前利用这个缺陷获得收益的程序却运行异常，开始声讨Intel 80286 CPU出现了bug。这应该算是修复之前的“bug”而引出了新“bug”？到底谁才是bug已经说不清了，Intel当时一定捂着脸委屈到“我……我哪知道你们这帮奇葩之前竟然这么玩啊！”，总之，让人哭笑不得。奇葩还需奇葩治，当时PC大佬IBM放了个奇葩招解决了这个问题。如图10-4所示，其在主板上将80286 CPU的A20地址信号接入一个与门的一个输入端，另一个输入端与当时广为采用的Intel 8042键盘控制器的某个闲置信号相接，通过写入8042控制器对应的寄存器可

试读结束：需要全本请在线购买： www.ertongbook.com

以控制该信号输出0还是1，当8042控制器的该信号输出1时，与门的输入将与A20原生输出一致，但是当8042控制器的信号输出为0时，A20 Gate输出总为0，也就可以模拟8086的20根地址线效果了。将这个过程做成一个选项，在BIOS阶段供用户配置即可。这个做法流传开来，一直到Intel赛扬CPU时代的PC上还曾见过，不过后续的PC基本上已经不再使用8042来连接A20地址线，而是采用其他更便捷的方式，因为使用8042来控制的话，要先写寄存器把键盘控制器禁用，这样才能清空其缓冲区内容，然后再写寄存器下发对应命令启用A20 Gate。采用更快方式启用禁用A20 Gate的主板BIOS选项中会称之为比如“Fast A20 Gate”，此时主板会采用专用控制硬件来控制A20 Gate，并向外暴露操作地址0x92，对该地址的第1个位写1即可使能A20 Gate。注意，Fast A20 Gate中的“Fast”并不代表开启了A20 Gate系统就变快了，哈哈，当年冬瓜哥可真是这么认为的。这世界似乎真不缺奇葩，一茬接一茬的出现，挑逗着你的神经。当然，CPU在设计和演进时，各种设计失误和奇葩事件会持续发生，只不过由于当代的CPU集成度越来越高，这些事件就逐渐不为人知了，都被掩盖在CPU内部了，比如通过更新微码的方式来解决一些bug等。而随着整个计算机产业的发展，目前人们多数都聚焦在上层的比如大数据、云计算、人工智能等领域去了，计算机硬件和底层软件成为宏伟建筑的砖头和钢筋水泥，它们被埋没到了漂亮的外表和豪华的室内装饰之下，再也不被人所看到和关注，甚至一旦看到赶紧封装起来，成为与上层角色格格不入的存在。

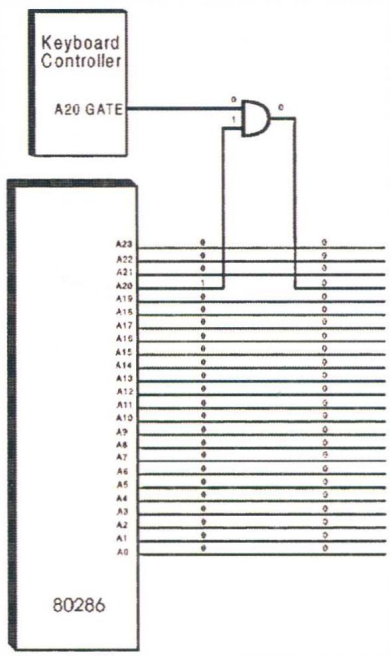


图10-4　A20 Gate示意图

如图10-5所示为8086体系寄存器和段布局示意图，大家可以结合之前的内容具体体会一下。

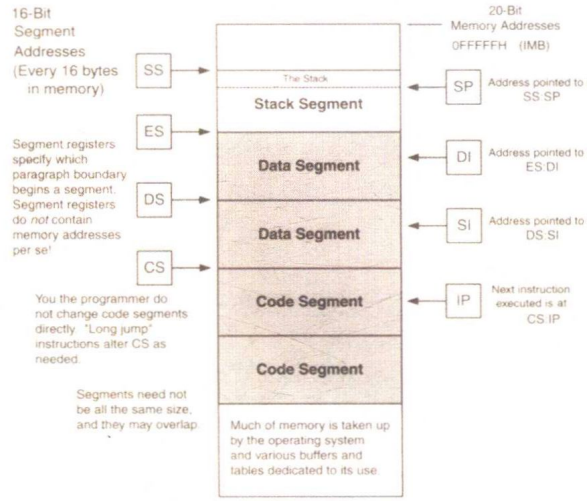


图10-5　8086体系寄存器和段布局示意图

提示 ▶▶▶

8086对内存的分段管理模式可以与exe/elf可执行文件里的那些分段概念匹配起来，exe代码载入内存之后，可以完全按照相应的布局来放置，代码中的地址也根据其访问的区域，使用对应的段基地址来做长/远跳转。然而由硬件完全控制内存布局的做法不灵活，所以在后来的分页模式下，分段机制基本成了摆设，代码中也不再有所谓远跳转的概念。虽然程序依然遵循elf/exe定义的布局格式，但是每个区域的描述和访问控制完全放在了软件中来处理。

线性/逻辑/物理地址

在8086体系下，代码中出现的segment：offset（如果是代码段的话则俗称CS：IP，IP就是PC指针地址Instruction Pointer的意思）的地址组合（或者只给出Offset，CPU会默认使用当前段寄存器中的值作为基地址），被称为**逻辑地址**；segment左移4位+offset之后的地址被称为**线性地址**；线性地址也就是最终用于放置在20位物理总线上的地址，所以其也就是最终的**物理地址**。在下文中会看到，当启用了带有保护模式的分段，或者启用了分页技术之后，线性地址还需要经过一次转换映射，才被转换成最终的物理地址。

总之，8086的分段的做法本身就比较奇葩，第一是比较乱，比如2000h：1F60h、2100h：0F60h、21F0h：0060h、21F6h：0000h、1F00h：2F60h、……数不胜数，这多种segment：offset组合都可以表示同一个物理地址：21F60h。每一个物理地址都可以有大量不同的segment：offset组合来表示，因为同一个物理地址可以被表达成两个值之和，所以多个段描述的

区域可以是重叠（Overlap）的。再加上代码中可以强行指定段基地址+偏移量来访问任何地址，这会让程序员疑惑。第二是没有提供保护模式，也就是虽然分了段，但是却不禁止越界行为，导致实现多任务时基本不现实，这就好比上车要系安全带一样，因为即便你自己开的慢但是如果别人不靠谱一样会把你撞废，而8086不提供任何安全保障。于是，Intel从80286开始，采用了彻底改进的分段方式来解决上述两个问题。

10.1.4 80286分段+保护模式

要想实现保护模式，必须让不同任务/进程的地址区间不发生重叠，同时还要禁止进程越界访问。80286使用下述设计思路来解决这两个问题。为了保持对之前程序的透明，80286仍然接受segment:offset模式的寻址方式，但是，CPU并不会向8086那样将segment左移4位+offset然后用这个地址直接去访问物理内存，而是需要操作系统的内存管理模块先在内存中设立一张表，并将该表的基地址告诉CPU（CPU将其保存到一个新设立的专用基地址寄存器中），CPU每次拿到一个访存地址，便将该地址的segment部分（16位）作为一个索引号去查找该表中第segment行上的条目，从中读出一个基地址值，然后利用这个值，与offset相加，得出最终用于访存的物理地址，该条目内同时还存有一个限长值，offset如果超过这个值就直接报异常。大家可能已经体会到了，只要操作系统的内存管理模块在这个表对应的条目中放置分配好的地址，就可以实现“我指哪你才能打哪”的效果，从而将程序框住在由操作系统分配的内存区间来运行。此时，由于物理地址必须加上一个偏移量之后才被拿去访存，所以黑客程序就会干瞪眼，即便它知道另外一个程序的访存地址，也不会知道这个访存地址最终的真实位置，除非它可以读出这张表，但这是不可能的，见下文。

那么，如果程序非要打一个操作系统没有指向的地方呢？比如假设这个表中的第FFFFh项并没有被分配任何基地址，程序将segment号FFFFh存入CS寄存器，此时80286处理器会寻找到一个空项目，则报告异常。那好，程序如果先擅自把一个地址写入表中的这行呢？对不起，办不到，因为这个表所在的地址必须不能被分配给任何用户程序，那么程序就访问不了这个表。那好，如果程序自己创建一个表，然后将这个表的基地址更新到CPU内部对应的基地址寄存器中呢？对不起，做不到，因为用于更新该基地址寄存器的指令，是一条特权指令。那好，如果程序随便载入某个值到CS/DS寄存器中，而这个值对应的表中的条目真的被分配了某个基地址，但是是给其他程序分配的，此时该程序不就可以访问到其他程序的数据了么？为了解决这个问题，内存管理模块需要为表中的每个项目设置一个访问权限，我们下文中再描述。

可以看到，所有的路都被封死了，程序只能在层层保密下执行，这就是所谓的保护模式。既然如此，之前的8086程序就无法在80286下运行了？是的，因为其中很可能包含一些特权指令。但是80286提供了两种运行模式，一种是实模式，加电后默认运行在实模式，此时最多访问1MB内存，模拟8086的行为，所以该模式下不做特权级限制，可以兼容8086程序；第二种是保护模式，通过对特定的控制寄存器写入对应的值（通过mov指令将CR0寄存器里的“PE”位改为0或者1来控制），将CPU切换到保护模式运行，此时会切换到上述分段和特权级检查模式下运行。

提示 ▶▶

mov指令按理说是个非特权指令，但是为何可以用来操作敏感的CR寄存器呢？值得一提的是，CPU内部并不是仅根据指令的Opcode字段来判断其特权，也要查看目标寄存器，比如CPU一看是要操作CR寄存器，而当前的运行特权级假设为Ring3，那就报异常。

所以，正如前文中所述，实现保护模式，需要第一，必须通过某种方法限定访存范围；第二，必须将指令加上特权级限制并提供特权级切换机制（也就是前文中所述的，在Int指令执行后会切换到Ring0权级，返回到用户进程前切换到Ring3权级，以及程序主动使用lds/les/mov等指令更改DS/ES寄存器时，要做权限匹配）。

10.1.4.1 全局描述符表

上文所述的这个表，被称为GDT（Global Descriptor Table，全局描述符表），用于保存GDT在内存中的基地址的寄存器被称为GDTR。表中的每个项目被称为一个描述符，其不但描述了内存中的某个段基地址、长度，还需要加入一些权限等属性的描述，比如该段中保存的数据/代码是哪个特权级才能访问的、该段保存的是用户数据/代码还是操作系统底层数据/代码、该段保存的是数据还是代码、该段是否可被读/写/执行等信息。

如图10-6左侧所示，当访问内存时，CPU先用各个段基地址寄存器中的值去寻址GDT（需要将该值与GDTR寄存器中的值相加才能读取到正确条目），找到对应的描述符，读出描述符中的Base Address，利用这个Base Address+offset（程序代码中给出）寻址物理内存。描述符中存有基地址、长度和Access属性信息（上述）。你会发现，此时再称CS/DS等寄存器为“段基地址寄存器”已经不合适了，其存储的已经不是段基地址（段基地址存储在段描述符中），而存储的是GDT中段描述符的索引，人们将其简称为段选择子（Segment Selector）。利用段选择子索引GDT

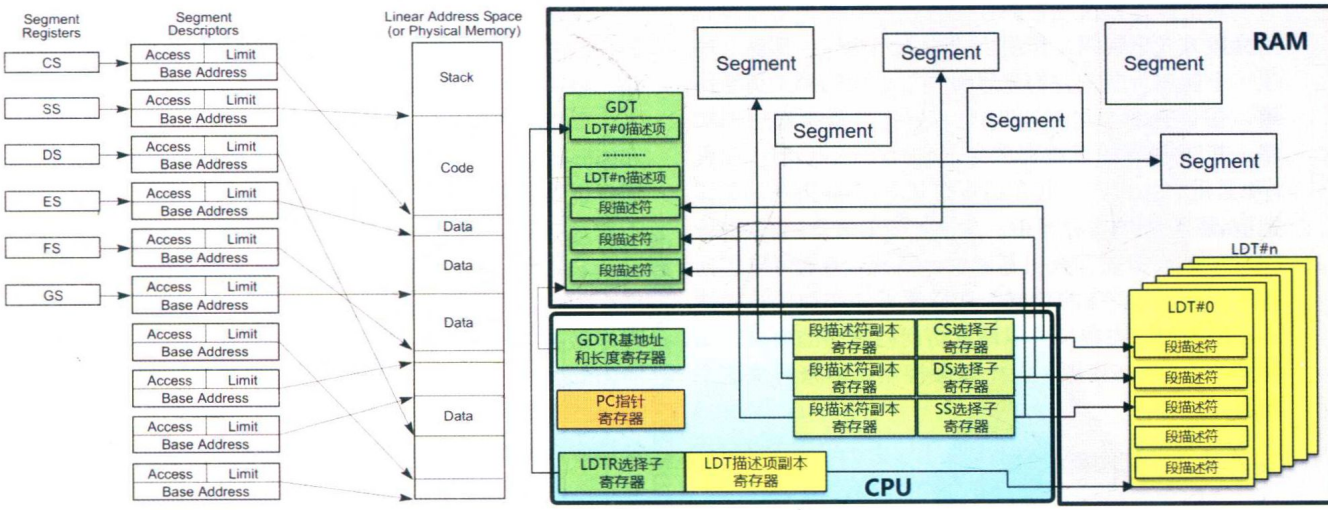


图10-6　段选择子、GDT、LDT示意图

读出的描述符，会被CPU自动存储到内部的一个专用寄存器（该寄存器不可被程序操作，仅供CPU后续使用），这样，在DS/CS等寄存器被程序改成其他值之前，CPU后续的执行不需要重复地去读GDT来拿到段基地址，直接用上一次保存在这里的基地址与offset相加即可，也就是图中的“段描述符副本寄存器”。

提示

80286处理器为了兼容8086程序，其加电启动之后首先运行在实模式，需要靠特殊的指令将其切换到保护模式。但是，286处理器的保护模式是模拟出来的，具体来说，就是它运行在实模式时并不像8086处理器那样用CS/DS等段寄存器来左移4位作为段基地址，而是使用上述的“段描述符副本寄存器”中的基地址来寻址，也就是说，当80286处理器运行在实模式时，其会自动将CS/DS等段寄存器中的值左移4位然后写入这个副本寄存器，后续拿着它作为段基地址寻址。相当于，在保护模式下286处理器使用段寄存器来寻址GDT拿到段基地址然后写入副本寄存器，在实模式下其是直接将段寄存器的值左移4位写入副本寄存器。**不管在实模式还是保护模式，CPU都是拿着该副本寄存器中的基地址来寻址的**。这里容易忽略的是，认为在实模式下CPU是拿着CS寄存器中的值直接输送到一个移位4位的移位器然后输出地址，其实并不是的。而8086处理器中并没有这个副本寄存器。这么说，对于80286处理器，由于其地址线为24根，其副本寄存器中的基地址长度也为24位，当其运行在实模式下，如果能够用某种方式强行将某个24位的地址值载入这个副本寄存器，那么即便在实模式下，也可以访问到超过1MB的内存地址。在实模式下是无论如何也无法载入超过20位的地址到副本寄存器的，但是，如果先让CPU进入保护模式，然后在GDT中捏造一条描述符，将基地址改为全00000h，长度改为最大值FFFFFFh（80286的最大24位寻址范围），然后使用jmp CS：IP指令（将CS值设置为GDT中刚才捏造的项目的序号）让CPU载入CS寄存器，CPU将自动将GDT中捏造的地址自动载入副本寄存器，然后再使用指令从保护模式切换回实模式，80286以及后续的处理器在切换模式时并不会自动清空这个副本寄存器（这被一些人认为是设计上的一个漏洞），所以这个描述符就会留在副本寄存器中，此时CPU可以访问全部的地址空间，后续程序只需要使用Jmp IP这种方式来寻址即可，不需要再给出CS，一旦给出新CS，则CPU会用给出的CS左移4位写入副本寄存器，之前的24位地址就会变成20位，被限制到1MB的寻址范围了。这个技巧在80386处理器（支持32位寻址）时代得到了广泛的应用。后来有人认为该设计并非漏洞，因为80386处理器加电后自动运行在实模式，其第一个取指令的地址是FFFFFFF0h，也就是4GB-16B处，去寻址BIOS ROM，而这显然超出了实模式的1MB寻址范围限制，其内部电路强行将该地址载入副本寄存器中，仅当程序代码中第一次尝试载入CS寄存器后，CPU的寻址范围才会立即被立即限制到1MB内（当然，可以打开A20地址线，这样可以多寻址64KB-16B=48B的内存）再也跳不出了，除非切换到保护模式。所以看上去是有意这样设计的。但是如果说它是漏洞也合理，因为完全可以被设计为从保护模式退出到实模式时自动清空副本寄存器，但是却没有这么做。所以这个技巧也算是一个免费赠送。CPU内部的用于存放描述符副本的寄存器，在虚拟机模式下会被暴露出来，比如CS.Base、CS.Limit等，可使用特殊指令（比如VMREAD指令）来分别操作该寄存器内的基地址部分和长度部分。在实模式下寻址所有内存空间又被人戏称为“Unreal Mode”。这个技巧后来甚至被微软在MS-DOS操作系统中用于访问扩展内存，详见下文。

对于操作系统底层的程序代码和数据，也可以使用分段方式来组织，作用步骤与上述相同。那就会产生一个问题。现在我们来思考一下上面的那个遗留问题，假设有两个进程A和B，A为操作系统底层的程序，其DS段基地址被存放到了GDT中的第a项；而程序B是用户态程序，其在运行时执行了lds指令，尝试把值a装入到DS寄存器中，这样，CPU就会用a来寻址GDT中的段描述符找出基地址，这样，B就可以访问到系统底层程序A的数据。这样就无法实现保护模式了，因为用户态程序可以肆意访问内核态的数据。如何解决？80286使用了三个权限控制字段来解决这个问题。

10.1.4.2 实现权限检查

前文中提到过，Intel CPU提供了4个级别的Ring权限，用户程序运行在Ring3最低权限下，操作系统自身的底层程序运行在Ring0最高权限下。那么如何体现这种权限级别？如图10-7所示，CS和各种其他段选择子寄存器中，其实不仅存放选择子，还存放2位长度的CPL（Current Privilege Level）控制字。操作系统启动之初，CPU依然处于实模式，可运行所有特权代码，操作系统启动到一定过程会将CPU切换到保护模式运行，然后自己会在Ring0来运行自己的代码，此时，CS寄存器中的CPL字段会为00，也就是表示Ring0，表示当前的运行权级为Ring0最高级。当Loader程序加载某个用户程序时（比如Linux操作系统下的命令行Shell程序），首先为程序分配内存，然后将要用户程序的入口地址，以及对应的CS寄存器选择子值（包含CPL值并设置为3）压入栈中，然后使用iret指令（详见后文），让CPU将栈中的这些参数装载到CS寄存器中，然后跳转到指定的用户程序处执行。

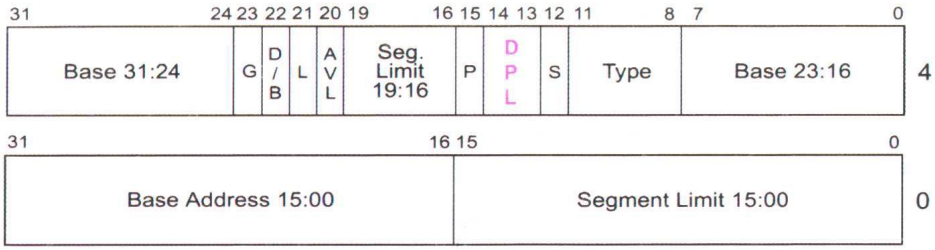
当程序代码需要访问的数据位于与当前段不同的段中时，比如希望跳到另一个代码段，或者访问另一个数据段中的内容时，程序代码中需要给出【新段选择子】：【offset】，新的段选择子中的权限控制字被称为RPL（Requested Privilege Level），之所以被称为Requested的原因是因为当前程序“要求”跳转到该段。CPU根据新段选择子从GDT中选出对应的描述符，而描述符中也存放有权限控制信息，被称为DPL（Descriptor Privilege Level）。于是，这个场景就是：某个运行在CPL级别的人（当前段寄存器中的CPL），拿着印有RPL级别的通行证（欲切换到的目标段选择子中的RPL），欲进入只有不低于DPL级别才能进入的场所（目标段描述符中的DPL）。门卫如果此时只检查该人出示的通行证上的RPL的话，那就太傻了，我完全可以伪造一张Ring0的RPL通行证，因为代码中可以直接给出被编辑成任何值的段选择子。所以，门卫应当先看你的身份证（当前的CS寄存器，印有CPL级别），再看你的RPL通行证，取其中权限较低的值，再与DPL比较。整个判断过程如图10-8所示。

既然如此，印有RPL的通行证好像根本是多余的，CPU只把CPL与DPL进行对比不就可以了么？但是，设想这样一个场景：一个拥有Ring0最高级别的人，要想进入某场所，但是该场所中有Ring3、Ring2和Ring1三个级别的区域，而本次办的事只需要Ring3通行证访问Ring3区域即可，如果此时门卫只看脸而不看通行证，那么Ring2和Ring1区域你也能进去，这会导致潜在的问题。比如，一个运行在Ring3的程序A执行了某种系统调用（Int指令），委托Ring0的程序做某些事情，Int指令被执行之后，会载入对应的Ring0级CS描述符从而处于Ring0权级执行，也就是CS中的CPL=0，在执行过程中，Ring0的程序可能会访问到其他的Ring1、Ring2、Ring3程序中的一些数据，或是A故意设计好的，或是不经意的或者各种bug。而此时CPU会全部予以放行，为什么？因为当前的CPL=0，最高特权，可以肆意访问任何其他特权级的数据，而这便等效于：委托Ring0做事的Ring3程序挟天子以令诸侯，四两拨千斤，而它原本是没有权限去访问其他的Ring3程序中的数据的，更没有权限访问Ring2级的数据。正因如此才会设置RPL，当发生上述情况时，Ring0如果要访问比如某个DS，会强行将该DS的RPL字段设置为程序A的CPL，也就是3，那么就有了这种效果：当前正在运行的是Ring0程序，CS中的CPL=0，但是要访问的DS的RPL=3，这样，CPU拿着这个DS去选择GDT中的描述符，如果目标描述符的DPL=2，则不予放行，因为当前DS的RPL级别比DPL低。而如果当前运行的程序为Ring3，CPL=3，其给出一个RPL=2的DS试图越权怎么办？所以，CPU最终会判断DPL值是否≥max{CPL, RPL}，而不能仅判断RPL或者CPL。

10.1.4.3 本地/局部描述符表

经过这样设计之后，只要操作系统将自身的数据/代码段在GDT的描述符中的DPL设置为0，Ring3的进程就无法访问到这些描述符。但是仍然有个问题，Ring3的进程A是否可以擅自访问Ring3的进程B的数据段呢？按照上述场景，如果进程A强制给出一个CS选择子，去选择GDT中进程B的描述符，那么CPU在做权限检查时发现CPL=目标DPL，则予以放行，最终A可以窥探B。CPU分不清GDT中的哪个描述符隶属于当前程序，这些需要由操作系统来做，而CPU只能提供硬件上的鉴别辅助，当然，让CPU全做了也可以，但是会增加CPU设计负担而且不利于灵活性。当然，80286的设计者是不会有这个漏洞的，其做法是引入另一个表LDT（Local Descriptor Table，本地/局部描述符表）。

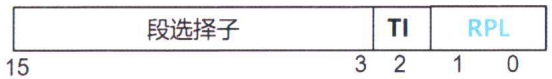
操作系统在分配内存时，将每个用户进程的所有段的描述符都被放置到LDT中，每个进程一个LDT。所有的LDT，每个都作为一个GDT中的描述符所描述的段而存在，于是，GDT至此有了两种段：放数据



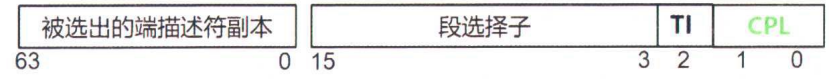
描述符

L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

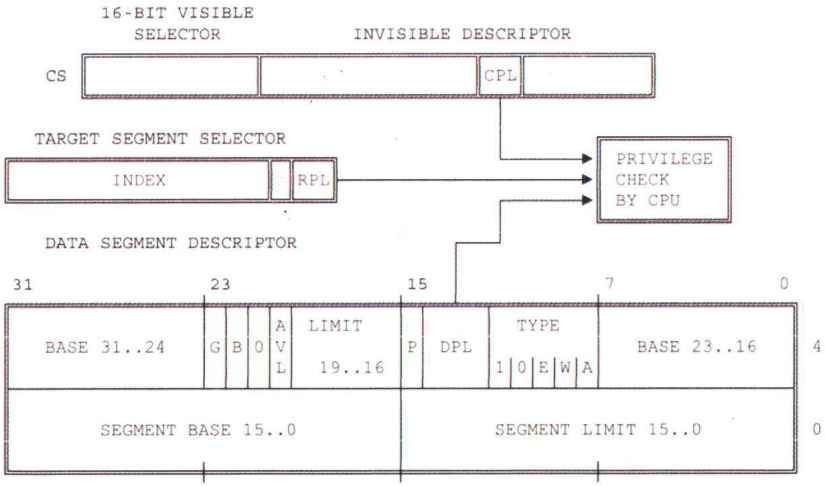


代码中给出的、欲载入寄存器的CS/DS/SS/ES/FS/GS选择子



当前寄存器中的CS/DS/SS/ES/FS/GS选择子

图10-7　引入DPL、RPL和CPL三个权限控制字段



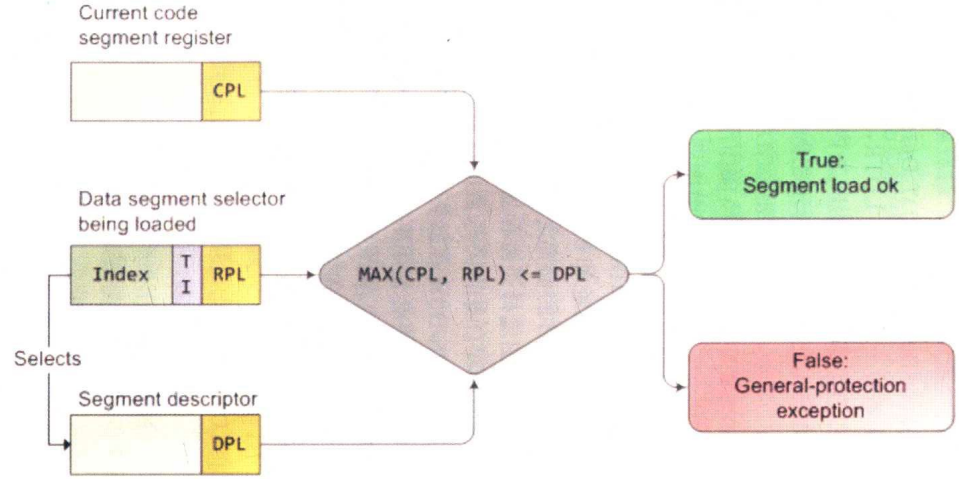


图10-8　CPU对RPL/CPL/DPL的权限鉴别机制示意图

/代码的段、放LDT表的段，对应的描述符也分别被称为Segment Descriptor、LDT Descriptor。然后，在CPU内部增设一个寄存器：LDTR，用来存放当前运行程序的、用于从GDT中选出LDT描述符的LDT选择子，其结构与图10-7下方所示相同，其TI=0（Table Indicator），表示该选择子要去GDT中选出描述符，其CPL/RPL字段为3，表示当前为用户态进程，其剩余的13位作为索引，去GDT中选出的就是对应的LDT描述符，并将其缓存到对外不可见的副本寄存器中。LDT描述符的结构如图10-7中左上角所示相同，其中包含LDT的基地址和长度，从而让CPU能够寻址LDT。知道了LDT的位置之后，CPU从而再用CS/DS等代码/数据段的选择子寄存器中的值，再去索引LDT中记录的该进程自身的代码/数据段描述符，将拿到的描述符放置到CS/DS寄存器旁边的副本缓存中，后续所有的访存请求将会使用副本缓存中给出的段基地址+offset来寻址物理内存。

每个进程加载之前，操作系统为其分配内存，并生成一张LDT，将分配好的所有段的描述符放入表中，然后再在GDT中开辟一个新描述符条目项，将LDT的基地址记录进去，然后采用lldt（Load LDT）特权指令将LDT选择子载入LDTR寄存器，这个动作会触发CPU在后台自动利用该选择子去GDT中读出对应的LDT段描述符载入自己内部的副本缓存，从而用该基地址去寻址LDT，然后将程序入口的CS：IP中的CS载入CS寄存器，IP载入PC指针寄存器，这样CPU就可以根据CS选择子从LDT中选出CS段描述符，从而得到CS段基地址，与IP相加，拿着相加后的地址访存，就可以拿到程序入口的指令，然后就可以开始执行了。这里面的映射关系非常复杂，一层套着一层，不容易梳理清楚。可以结合图10-6中给出的示意图仔细推敲。操作系统的进程管理模块也需要为每个进程记录其各自的LDT选择子，切换进程之前必须将对应进程的LDT选择子装载到LDTR寄存器中。

每个选择子寄存器（包括LDT/CS/DS/ES/FS/GS）的第三位，也就是位2（见图10-7中的TI，Table Indicator），来表示当前的选择子是要去从GDT（TI=0）还是LDT（TI=1）中选出描述符。也就是说，进程可以自主决定使用GDT还是使用LDT来获取段基地址。如果使用LDT来获取段基地址，那么CPU根据LDTR旁的描述符副本缓冲器中的LDT基地址来找到LDT，然后用CS/DS等段选择子来索引表中对应的描述符，获取段基地址并缓冲到CS/DS旁的副本缓冲器中，以供后续访存计算地址使用。

由于lldt指令为特权指令，所以用户程序要么只能访问由操作系统指定的LDT中的描述符中的基地址指向的段的访存范围，要么只能访问GDT中任意与当前进程CPL相同或者级别权级更低的描述符中的基地址指向的段范围。一般来讲，操作系统会将自身的数据和代码所在的段描述在GDT中的描述符中，并将DPL设置为0，以及将S字段设置为0以表示该段为系统段，这样，用户进程尝试访问这些描述符时就会被CPU给禁掉并报异常。只要操作系统确保所有的用户进程都采用LDT来存储段描述符，而不是直接放到GDT中，就可以做到Ring3之间的隔离。LDT的另一个作用是可以更清晰地将每个进程的描述符归拢，而不是分散在GDT中各处，也便于管理。

那么，为何依然可以用GDT来存放用户进程的数据/代码段描述符呢，都放LDT不好么？GDT的存在，是为了方便多个Ring3的程序共享数据用的。比如，操作系统可以分配一段内存，让其DPL=3，然后将该描述符放置到GDT中，这样，多个Ring3的程序就都可以访问它了，当然，也可以在每个进程的LDT中分别放置一份同样的描述符，其段基地址指向同一段内存，但是这样做就麻烦了一些，但是却有更高的可控性，因为如果将共享数据放到GDT中，那么所有进程都可以访问，如果想做到只让某个或者某些进程访问，那就需要放到对应进程的LDT中而不是GDT中。

当程序运行动态申请内存时，会由内存管理模块在当前段分配新的区域，或者开辟新的段，在GDT或者LDT里开辟一条空描述项，将分配好对应的段基地址写入，然后将段基地址返回给用户程序，用户程序使用$CS^{新分配}$：IP来访问申请到的内存，CPU会拿着$CS^{新分配}$去到LDT/GDT中找到对应描述项，从而找到基地址。

至此，Ring3程序不可直接访问Ring0的数据，也不可直接访问其他Ring3的数据，做到了彻底的保护模式。

如图10-9所示为不同的特权级别示意图，图中使用了Level而不是Ring这个词来表示特权级别，Level是Intel在早期使用的词汇。目前，基本上没有代码跑在Ring1/2特权级，基本上只使用Ring3和Ring0。

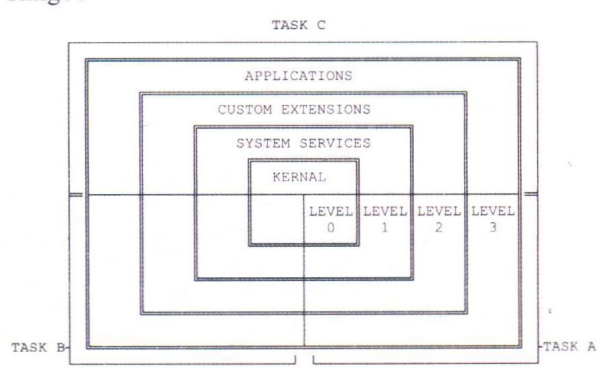


图10-9　不同级别特权示意图

解决了多任务+保护模式这个需求之后，一个新的需求就要开始酝酿了，那就是，每个段在内存中必须是连续存放的，如果内存中的空闲空间被碎片

试读结束：需要全本请在线购买：www.ertongbook.com