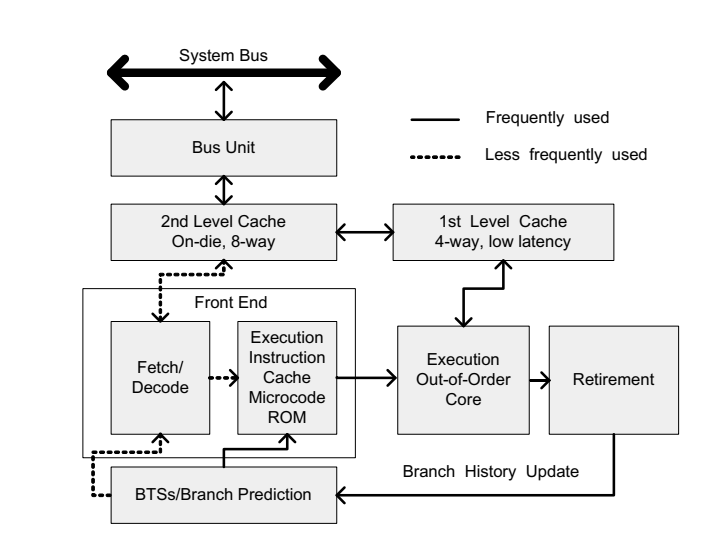
**汇编语言、实模式、保护模式、长模式简介**

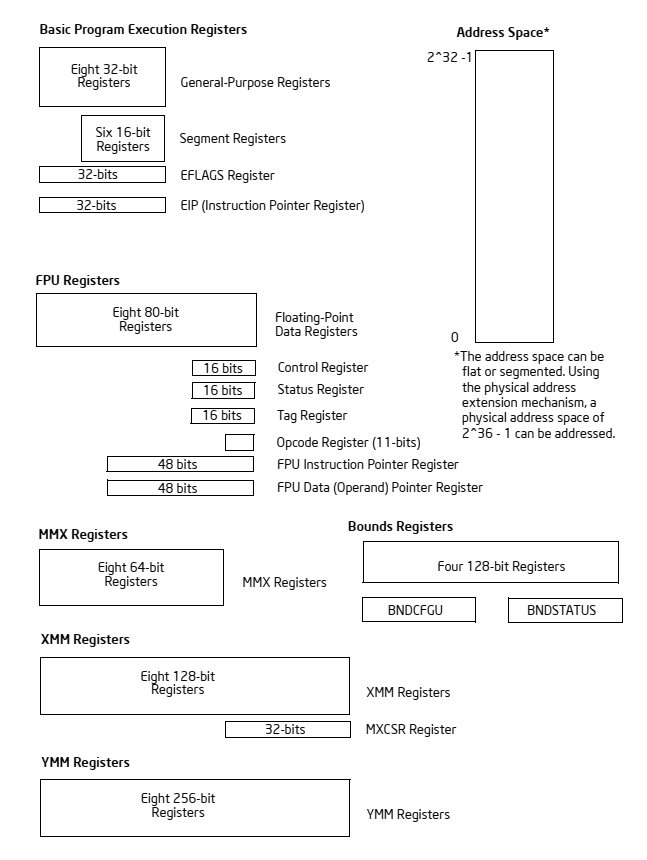


微处理器概念架构（P6）

CPU基本执行过程：

* + 取指令
  + 执行

当代CPU采用了很多并行技术，将汇编代码级别的指令进一步内部分解为微指令，进而实现指令预取、多流水线、乱序、分支预测等功能。每条指令的执行过程又可以被进一步细分为取指令（IF）、指令译码（ID）、执行（EX）、内存访问（MEM）、写回（WB）等。



IA-32 CPU架构基本执行环境

本简介主要介绍上图中的基本程序执行寄存器和地址空间两部分内容，其他部分，请参考Intel IA-32手册。

**一、汇编语言**

汇编语言（assembly language）是一种用于电子计算机、微处理器、微控制器或其他可编程器件的低级语言，亦称为符号语言。在汇编语言中，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址。在不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令（在这里，我们主要讨论Intel x86和x64）。

为了使得运算过程高效和易于实现，CPU硬件内部包含很多寄存器（register），用于指令运算的输入和输出。

如果用汇编语言实现printf()，可能需要执行数千条汇编指令。可见，采用汇编语言，即使要获得最小的结果，实际上也需要很多事情，这就是为什么创建高级语言的原因。汇编语言也特定于CPU体系结构，而像C这样的语言是可移植的。

汇编语言相比于其他高级语言，具有更细微的执行控制能力，同时也是控制CPU运行模式和进行监测的唯一指令级接口，因此，在操作系统底层实现中，通常会包含一部分汇编语言编写的代码，如loader、scheduler、lock等。这也是深入学习操作系统必须要了解汇编语言的原因。

汇编语言具有（相对）少量的命令集，主要包括：

* 在不同位置之间移动数据的命令
* 执行数学算法的命令（简单到复杂）
* 检查条件的命令（如if）
* 其他命令

CPU是执行汇编指令的单元。它们的执行方式取决于处理器的运行模式，共有4种模式：

* 实模式
* 保护模式（包含两种形式，分段和扁平）
* 长模式
* 虚拟化（不完全是一种模式，但是我们稍后会讨论）

**1.1内存**

从物理上讲，内存是一个大数组。如果拥有4GB内存，则可以将其描述为unsigned char mem [4294967295]。但是，其使用方式因处理器模式和操作系统配置的不同而有很大差异。因此，不能以数组的形式对其访问。

**1.2分段**

与C语言不同，内存通常不被视为连续的字节数组（如C数组），而是分割为多个部分。根据CPU模式（实模式，保护模式或长模式），段的含义不同。每个内存地址都首先由一个段寄存器指定，该寄存器保存一个段的起始位置，再通过偏移量指示距段段开头的位移，构成了真正的物理地址。

**1.3寄存器**

除了内存之外，每个CPU都有一些辅助器件来存储数据，称为寄存器。可用的寄存器取决于当前的运行模式。有些寄存器有特殊含义，有些则是通用的。

**1.4堆栈和函数**

为了使函数更容易拥有局部变量（例如在C ++中）并在它们之间传输数据，每个应用程序都设置了一个特殊的段，称为“堆栈段”，该段保存用于堆栈的内存的地址。堆栈是一种“后进先出”向量：最后一次压入堆栈的内容通过弹出操作最先得到。从实现上来讲，堆栈只是专门划分出来用于函数调用和执行的普通内存。

下面代码（简化）展示了函数调用时，汇编语言层面大致的执行情况：





变量“a”和“b”首先被“压入”到临时堆栈。调用该函数，然后在清除堆栈的情况下返回，返回值位于ax中。

注意，以上内容大大简化了汇编代码的实际实现。

**1.5中断**

中断是指打断其他正在运行的代码。可以假定它是一个能够在另一个函数中运行的函数。有一些中断由CPU自动生成（硬件或发生异常时），有些则由软件“调用”产生。它们的工作方式取决于运行模式，并且最多可以有255个中断。

**1.6异常**

异常是由CPU（例如，当C ++代码中被零除时，执行int 00功能）执行产生或调用API​​（例如，通过throw关键字）触发的软件中断。在最底层实现中，异常和中断之间没有区别。

**二、CPU寄存器**

**2.1 16位寄存器**

* + AX
  + BX
  + CD
  + DX
  + SI
  + DI
  + BP
  + SP
  + IP
  + 16-bit Flag Register

IP（instruction pointer）指向当前向指令执行的代码地址。IP的值随着指令的执行，由硬件自动更新。

AX, BX, CX, DX既可以被整体访问：

mov ax,1 ; ax is now 1

mov cx,ax ; cx is now also 1

也可以只访问它们中的低8位(al, bl, cl, dl)或高8位(ah, bh, ch, dh)

xor ax,ax ; ax is now 0

mov ah,1 ; ah is now 1, thus ax is now 0100h

mov al,2 ; al is now 2, thus ax is now 0102h

SI、DI总是以16位存储器的形式被访问，并且通常用作数据指针。BP尽管也可以作为通用寄存器，但通常总是用来访问堆栈，SP则拥有指向当前栈顶的指针。

; assume that SP has the value of 0100h

mov ax,3 ; ax is now 0

push ax ; ax is put to the stack, SP has now the value of 00FEh; (100h - 2)

mov dx,5 ; dx is now 5

push dx ; dx is put to the stack, SP is now 00FCh;

pop bx ; bx gets the value from the stack top (5). SP gets back to 00FEh

pop cx ; cx gets the value from the stack top (3). SP gets back to 0100h



如果向占中压入的数据量超出的栈的容量，会引起栈的溢出，程序崩溃。如层次过深的递归调用，就会引起这种问题。

指针访问：类似C语言，如果在C中有一个包含指针的变量，可以使用\*来访问数据。在汇编中，则可以用“[]”完成相同的操作：

; 假定DS指向数据段

mov si,000fh

mov dx,[si] ; 现在dx中包含了地址 DS:000fh中的内容

实模式下，如果尝试访问不存在的内容，就可能会破坏自己的程序或破坏整个操作系统，或两者都破坏。而在保护模式下，不允许访问不存在的内存，因此将触发异常处理程序（否则，程序将被终止）。

标志寄存器（FLAG）包含16位（并非全部使用），这些位会根据不同操作码的操作来改变其值。 JMP命令（JZ，JAE，JB等）可以根据这些标志有条件地跳转。 例如，操作数结果为零时，ZF（零标志）设置为1。

mov ax,bx ; Get value of BX to AX

or ax,ax ; Is AX 0? If yes, or ax,ax will also say 0, so ZF will be 1

je AxIsZero

jmp AxIsNotZero

PUSHF和POPF可以用来读取和设置标志寄存器中的标志：

pushf ; push flags to stack

pop ax ; ax has now the flags

or al,1 ; set the bit 0 to 1

push ax

popf

**2.2 16位段寄存器**

* + CS
  + DS
  + SS
  + ES
  + FS
  + GS

这些寄存器中包含的值用于标识某个段。这些值的解读方式取决于当前的CPU模式（实/保护/长模式）。

CS始终保留当前执行代码的段。不能通过mov cs，ax来设置CS。当调用驻留在另一个段中的函数（FAR call）时，或者跳转到另一个段中（FAR jmp）时，CS的值都会发生变化。

DS保留默认数据段。 下面指令意味着：

mov si,0

mov ax,[si]

mov bx,[1000h]

AX寄存器从DS指向的段中获取数据，其段内偏移量由SI指定，而BX从DS指向的段中获取数据，其段内偏移量为1000h。 如果要使用其他段，则必须明确地这样做：

mov di,0

mov ax,[fs:di]

mov bx,[es:1000h]

当DI用作索引（偏移）时，ES是其默认段。当BP用作索引时，SS是默认段。 在所有其他情况下，DS是默认段。 请注意，并不是每个寄存器都可以在实模式下用作索引，例如，mov ax，[dx]在实模式下无效。

ES，FS和GS是通用辅助段寄存器。SS保存堆栈段的基址。

因此，代码始终在CS：IP处执行，堆栈栈顶由SS：SP指向。

**2.3 32位寄存器**

在所有CPU模式（实模式、保护模式、长模式）下，都可以使用32位寄存器。

* + EAX
  + EBX
  + ECD
  + EDX
  + ESI
  + EDI
  + EBP
  + ESP
  + EIP
  + 32-bit Flag Register

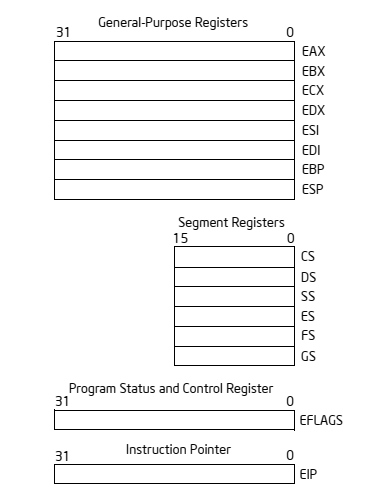
这些寄存器是16位寄存器的扩展。

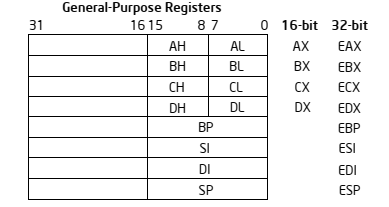
mov eax,0 ; eax is now 0, so ax is also 0.

mov ax,1 ; ax is now 1, eax is also 1

or eax,0FFFF0000; ax is now 1, eax is now 0FFFF0001h

32位寄存器在实模式下也可使用，索引寄存器（EDI、ESI）只能使用低16位（即索引最大值不能超过65535）。





**2.4 64位寄存器**

64位寄存器只有在处理器运行于64位模式下才可用。这些寄存器不能用于实模式、保护模式或兼容模式。

* + RAX
  + RBX
  + RCX
  + RDX
  + RSI
  + RDI
  + RBP
  + RSP
  + RIP
  + 64-bit Flag Register

另外，X64模式还增加了8个64位寄存器（r8,r9,r10,r11,r12,r13,r14,r15）用于辅助存储，和一些128位寄存器处理多媒体向量运算。

**2.5 控制寄存器**

* + CR0
  + CR1
  + CR2
  + CR3
  + CR4

这些寄存器存储关于CPU当前状态的信息。

CR0主要用于设置CPU进入保护模式（位0），和启用分页（位31）

CR1保留

CR2保存缺页异常对应的线性地址（有缺页异常自动填充）

CR3存储页表的位置

CR4定义其他一些标志，如物理地址扩展（Physical Address Extensions）和虚拟86（VM86）模式等

**2.6 调试寄存器**

* + DR0
  + DR1
  + DR2
  + DR3
  + DR6
  + DR7

这些寄存器保存硬件调试信息。DR0-DR3存储4个断点的线性地址，DR6-DR7存储相关的标志信息。

**三、实模式**

实模式是较为古老的模式。DOS就是在这种CPU模式下运行。当使用/ r开关启动时，Windows 3.0也将在这种模式下运行。所有指令的访问或执行都是16位的。这是最弱的一种操作模式（但不是最简单的一种）。 内存由20位地址总线寻址，最多可以访问1MB内存。 超出此限制的可用内存在实模式下无法使用。

**3.1 寻址和分段**

在实模式下，所有数据操作或访问均为16位。不能直接使用从0开始的绝对索引来访问内存，而是将内存分为多个段。每个段的值乘以16代表当前段的起始地址。对于当前段来说，通过段内偏移量来表示距该段起点的距离。通过这两个数据（段：偏移），确定了CPU要访问内存的绝对值。例如：

0000h：0000h：代表，段0，偏移量0，0 \* 16 + 0 = 0=实际内存地址；

0100h：000Fh：代表，段100h，偏移量0Fh，100h \* 16 + 0Fh = 100Fh =实际内存地址

0002h：0000h：2h \* 16 + 0 = 20h=实际内存地址

0001h：0010h：1h \* 16 + 10h = 20h=实际内存地址。（此处所见，内存地址是可以重叠的）

因为段和偏移量仅为16位值，所以此寻址方法可访问的最大内存为0ffffh：0010h = 1MB。指定0ffffh段和大于0010h的偏移量将导致环绕（请参阅保护模式，A20地址线）。并且由于0a000h：0000之后的区域是为系统（如显示器输出等）保留的，所以对于DOS应用程序来说，仅有640 KB可用内存。

此外，所有的段都可以从任何位置进行读/写/执行访问（也就是说，任何程序都可以在任何段中读/写或执行代码）。由于在16位实模式操作系统中，CPU通过上述方式访问内存，所以任何应用程序都可以从内存的任何部分读取或写入，包括操作系统所驻留的部分。这就是为什么实模式OS是单任务系统。

在实模式下，CS：IP保存当前CPU的执行点，DS保存默认数据段，而SS保存堆栈段。具有超过64K的代码或数据段的任何应用程序都必须拆分为多个段。

**3.2中断**

中断是一些特殊的功能，当某些事情发生时（硬件中断）（例如被零除）自动执行，或由软件（通过使用INT指令——软件中断）调用。在实模式下，有256个中断。所有中断的segment：offset存储到一个表中，称为中断向量表，表的起始位位置位于绝对地址0，可以使用LIDT指令（在286+中）将中断向量表放置在其他位置。

某些事件发生时，处理器会自动发出某些中断。在实模式下，最重要的是：

* 中断0，被零除；
* 中断1，单步调试；
* 中断3，断点触发；
* 中断6，无效操作码；
* 中断9，键盘按键。

软件中断可为实模式应用程序提供各种服务。最重要的中断是：

* 0x10，BIOS显示功能
* 0x13，BIOS磁盘功能
* 0x14，BIOS串行端口功能
* 0x16，BIOS键盘功能
* 0x17，BIOS并行端口功能
* 0x21，DOS功能（文件，输入，输出，应用程序，配置等）
* 0x2F，TSR功能
* 0x31，DPMI功能
* 0x33，鼠标功能

**3.3程序执行**

程序由DOS加载到内存段中，并从EXE头文件中指定的偏移量开始执行（如果它是没有头文件的COM文件，则从0100h开始执行）。之后，该应用程序可以自由执行任何操作，访问任何位置的内存。实模式的“特征”：应用程序拥有整个计算机。另外，允许应用程序直接与任何硬件通信（通过输入/输出操作码），从而绕过操作系统可能具有的任何限制或安全限制。如果应用程序崩溃，则整个系统崩溃，必须重新启动。

**3.4代码**

以下是一个16位“Hello World”程序示例，其中用到了多个段：



上述代码中包含了data16，code16，code16\_2和stack16四个段，汇编器（assembler）是如何确定它们的值的？实际上，汇编阶段，仅仅是为它们创建占位符，并写入到EXE文件的符号表项（Entry）（称为“重定位”）中，以便加载程序将代码复制到内存后，将段的真实值写入指定的占位符（地址）。由于exe文件中包含重定位映射表，因而可以实现多个段的重定位。而COM文件则不能有多个段，即使它们的总和小于64KB。

该程序通过远调用（far call）执行另一个段中的函数ShowMsg，该调用使用DOS功能（09h，INT 21h）显示文本。 该程序还可以通过直接写入视频缓冲区（对于文本模式，驻留在0b000h段中），从而绕过任何操作系统或功能号09h可能进行的安全性检查。 因此，由于每个应用程序都可以轻松地写入任何地方，从而破坏另一个应用程序或OS的数据，因而多任务处理在DOS中是不可能的。

下面是16位COM文件形式的简单“Hello World”示例：



这里的COM文件和exe文件比有什么区别？所有内容（代码，数据，堆栈）必须驻留在一个段中。代码必须从偏移量100h开始（以允许DOS将信息放置到低100h字节中），并且没有定义堆栈段或数据段。

COM文件是直接“内存映射”，并且限制为64KB。前128个字节包含PSP，一个包含信息的数据结构，其余部分包含该程序的所有代码，数据和堆栈存储器。CS = DS = ES = SS。SP设置为0xFFFE以指向段的末尾。从CS：IP = 0x100（在PSP之后）开始执行。

通常，DOS程序由一些代码段，一些数据段和一个堆栈段组成，如上所述。 DOS程序（通过中断）调用DOS和BIOS功能并完成其任务。

**3.5编程模型**

由于段大小限制为64KB（段寄存器是16位的，再乘以16，构成20位数据寻址），因此存在许多编程模型，模型的选择取决于应用程序的要求：

* TIny，所有内容都必须放在一个段中（COM文件）。
* Small，只有一个代码段和一个数据段。调用和跳转全部在段内。
* Medium，当有一个数据段，但有更多代码段时。调用和跳转在段间。
* Compact，当有一个代码段，但有更多数据段时。调用和跳转全部在段内。
* Large，当有更多的代码和数据段时。调用和跳转在段间。
* Huge，当数据结构的大小超过64KB时，必须以编程方式将其拆分为多个段。

最常见的模型是Small和Large类型。

**四、保护模式**

**4.1段保护模式**

保护模式解决了实模式问题，特别是：

* 最多可直接访问16 MB（286）和4GB（386+）；
* 内存访问检查，可以实现保护并分级；
* 如果发生错误，可以将问题隔离出来，其他应用程序不受影响；
* 包括16位保护模式（286+）或32位保护模式（386+）两种。

DOS只在实模式下运行。使用/ s开关启动时，Windows 3.0在16位分段保护模式下运行。Windows 95+，Linux和其他32位操作系统以扁平保护模式运行，在了解偏平保护模式之前，需要先了解保护模式本身的复杂机制。扁平模式在常规分段保护模式下大大简化了许多复杂的检查控制过程。

保护模式引入了“环（ring）”的概念，即特权级。CPU有四个环（ring0、1、2和3），其中环0具有最高特权级，环3具有最低特权级。在特权较低环中运行的代码无法访问（不需要操作系统的监控）较高的环中的代码。

**4.1.1内存**

保护模式下，内存中的每个段不再是64KB大小。段可以具有任何大小，从1字节到4GB。每个段还有自己的保护限制（读，写，执行访问），以及其所属的特权级（环）。

**4.1.2寄存器**

实模式下的寄存器在保护模式下同样可用。另外，任何寄存器都可以用作索引，例如mov ax，[ebx]中的ebx可以作为内存地址索引使用。

**4.1.3全局描述附表**

全局描述符表（GDT）是一组条目，描述了CPU的所有段。 每个表项（条目）的长度为8个字节，格式如下：



注意，代码段不可写。但是，由于段基址可以重叠，因此可以创建具有相同基址和限长的可写数据段。

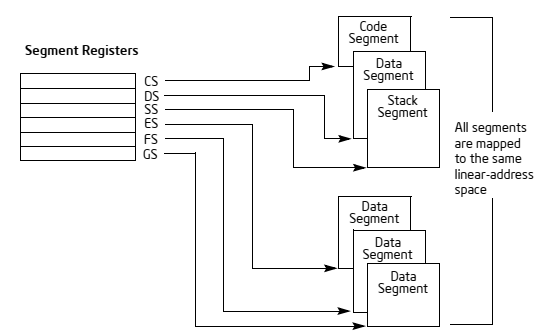
在实模式下，段寄存器（CS，DS，ES，SS，FS，GS）指定实模式段。无论指向何处，都可以向它们放置任何内容，还可以在该段中进行读写和执行。而在保护模式下，这些寄存器都通过段选择子（段选择器、selectors）完成加载。 段选择子是指向GDT的索引，并具有以下格式：



在保护模式下，不能像在实模式下那样，可以向段寄存器放入任意值，必须输入有效值，否则将产生异常。GDT表中的第一个表项是个特例，该表项始终设置为0。CPU不会从条目0中读取信息，因此它被视为“虚拟”表项。从而允许程序员将0值放入段寄存器（DS，ES，FS，GS），而不会引起异常。

可以通过执行LDGT命令将GDT加载到CPU，该命令指向一个6字节的数组：

* + 字节0-1包含GDT的完整长度，最大4KB => 4096个条目。
  + 字节2-5包含GDT在内存中的第一个条目的物理地址。



分段式内存管理中的段寄存器使用

**4.1.4中断**

中断向量表中的每条中断向量都是8字节，其结构如下：



操作系统使用LIDT指令加载中断向量表。 IDTR寄存器中包含6个字节的数据，其中2字节用于表的长度，4字节用于内存中的起始物理地址。

**4.1.5局部描述符表**

在多任务环境中，局部描述符表（LDT）为每个应用程序设置一组专用的段，并用LLDT汇编指令加载。段选择子中的LDT位（第2位）指定加载的段是来自GDT还是来自LDT。

**4.1.6 GDT中的系统段**

当GDT中的S位为0时，表示系统段。在这种情况下，GDT表项描述了三种系统段：

* + 任务段(Task Segments)
  + 调用门(Call Gates)
  + 中断门(Interrupt Gates)
  + 陷阱门(Trap Gates)（与中断门相同，但发生陷阱时仍允许中断）

此时，GDT表项中的第40-43位具有以下含义：

* + 0000 - Reserved
  + 0001 - Available 16-bit TSS
  + 0010 - Local Descriptor Table (LDT)
  + 0011 - Busy 16-bit TSS
  + 0100 - 16-bit Call Gate
  + 0101 - Task Gate
  + 0110 - 16-bit Interrupt Gate
  + 0111 - 16-bit Trap Gate
  + 1000 - Reserved
  + 1001 - Available 32-bit TSS
  + 1010 - Reserved
  + 1011 - Busy 32-bit TSS
  + 1100 - 32-bit Call Gate
  + 1101 - Reserved
  + 1110 - 32-bit Interrupt Gate
  + 1111 - 32-bit Trap Gate

**4.1.7调用门**

调用门是一种从低特权级切换到更高特权级的机制，用于用户级代码来调用系统级代码。可以使用以下格式在GDT中指定1100类型的表项：

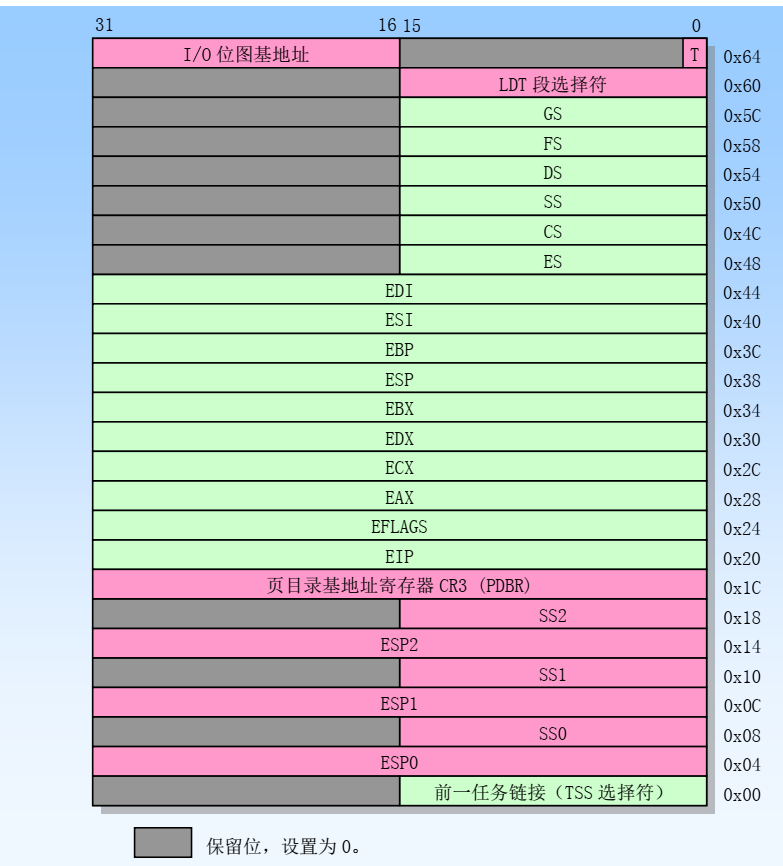


将CALL FAR与调用门的段选择子一起使用（偏移量将被忽略），将切换到该调用门并执行更高特权级别的指令。如果argnum指定要复制的参数，则在SS，ESP，CS，EIP压入内核堆栈之后，系统会将它们也复制到内核堆栈中。执行指令RETF从调用门返回。

调用门是一种CPU特权级之间进行转换的慢速机制。

**4.1.8任务状态段**

用于恢复一个任务执行的处理器状态信息被保存在称为任务状态段 TSS（ Task state segment）的段中。下图给出了 32 位 CPU 使用的 TSS 的格式。 TSS 段中各字段可分成两大类：动态字段和静态字段。



动态字段。当任务切换而被挂起时，处理器会更新动态字段的内容。这些字段包括：

* 通用寄存器字段。用于保存 EAX、 ECX、 EDX、 EBX、 ESP、 EBP、 ESI 和 EDI 寄存器的内容。
* 段选择符字段。用于保存 ES、 CS、 SS、 DS、 FS 和 GS 段寄存器的内容。
* 标志寄存器 EFLAGS 字段。在切换之前保存 EFLAGS。
* 指令指针 EIP 字段。在切换之前保存 EIP 寄存器内容。
* 先前任务连接字段。含有前一个任务 TSS 段选择符（在调用、中断或异常激发的任务切换时更新）。该字段（通常也称为后连接字段（ Back link field））允许任务使用 IRET 指令切换到前一个任务。

静态字段。处理器会读取静态字段的内容，但通常不会改变它们。这些字段内容是在任务被创建时设置的。这些字段有：

* LDT 段选择符字段。含有任务的 LDT 段的选择符。
* CR3 控制寄存器字段。含有任务使用的页目录物理基地址。控制寄存器 CR3 通常也被称为页目录基地址寄存器 PDBR（ Page directory base register）。
* 特权级 0、 1 和 2 的堆栈指针字段。这些堆栈指针由堆栈段选择符（ SS0、 SS1 和 SS2）和栈中偏移量指针（ ESP0、 ESP1 和 ESP2）组成。注意，对于指定的一个任务，这些字段的值是不变的。因此，如果任务中发生堆栈切换，寄存器 SS 和 ESP 的内容将会改变。
* 调试陷阱（ Debug Trap） T 标志字段。该字段位于字节 0x64 比特 0 处。当设置了该位时，处理器切换到该任务的操作将产生一个调试异常。
* I/O 位图基地址字段。该字段含有从 TSS 段开始处到 I/O 许可位图处的 16 位偏移值。

如果使用了分页机制，那么在任务切换期间应该避免处理器操作的 TSS 段中（前 104 字节中）含有内存页边界。如果 TSS 这部分包含内存页边界，那么该边界处两边的页面都必须同时并且连续存在于内存中。另外，如果使用了分页机制，那么与原任务 TSS 和新任务 TSS 相关的页面，以及对应的描述符表表项应该是可读写的。

**4.1.9 TSS描述符、任务门和硬件多任务**

CPU具有将任务状态段保存在GDT和LDT中的功能，因此可以进行任务切换。在任务状态段中，CPU保存有关本地任务（当前寄存器）的信息。使用指向GDT中TSS的选择子执行FAR JMP或CALL（偏移量像在调用门中一样被忽略）将“切换”到该任务，恢复保存的寄存器。TSS描述符用于指定要从中加载新CPU状态的TSS的基址和限制。 CPU有一个名为“任务寄存器”（TR）的寄存器，该寄存器告知哪个TSS将接收旧的CPU状态。当TR寄存器中加载有LTR指令时，CPU会查看GDT条目（由LTR指定），并用GDT条目加载TR的可见部分，并用GDT条目的基址和限长加载隐藏部分。保存CPU状态后，将TR的隐藏部分将被启用。

除了far call和jmp外，还可以使用任务门描述符（Task Gate Descriptor）触发上下文切换。与TSS描述符不同，任务门描述符可以位于GDT，LDT或IDT中（因此可以在发生中断时强制执行任务切换）。

**4.1.10进入保护模式**

CPU进入保护模式需要经历以下几个步骤：

* 开启A20地指线
* 设置GDT
* 设置IDT（如果需要处理保护模式中断）
* 设置CR0进入保护模式

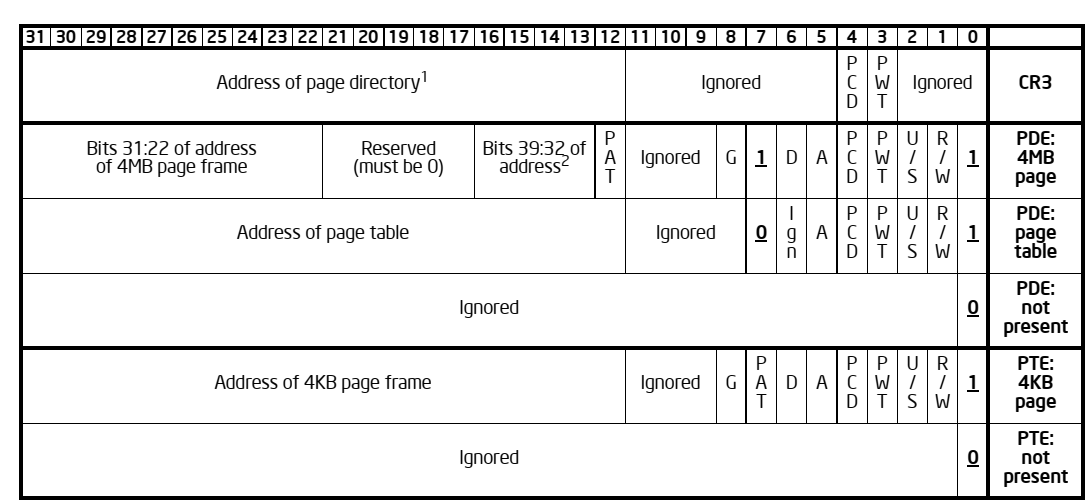
尽管分段模式提供了多任务、地址空间隔离、保护等多种能力，但任务切换和特权级切换速度很慢。

**4.2 扁平模式**

**4.2.1分页**

分页是将一种地址重定向到另一种地址的方法。请求的地址称为线性地址，目标地址称为物理地址。当线性地址与物理地址相同时，我们说我们处于“透视”区域。

为了完成分页，使用了两个表：页目录表和页表。



页目录表是由1024个32位表项组成的数组，格式如下：

* + P - Page is present in memory. This flag allows the OS to cache the pages back to disk , clear P, and reload them when a page fault is generated when software attemps to access the page.
  + R - Page is Read Write if set, else Read only. This restriction applies only to ring 3 unless the WP bit in CR0 is set.
  + U - If unset, only ring 0 can access this page.
  + W - If set, write-through is enabled.
  + D - If set, the page will not be cached. The CPU caches the page tables in it's Translation Lookaside Buffer (TLB).
  + A - Set when the page is accessed (not automatically, like the GDT bit).
  + N - Set to 0.
  + S - Set to 0. If Page Size Extensions (PSE) are enabled, S can be 1, in which case the page size is 4MB instead, and the pages must be 4MB aligned. This mode is introduced to avoid lots of small pages, at the expense of more memory wasted if the needed memory is somewhat larger than 4MB. Fortunately, modes can be mixed.
  + G - Set to 0.
  + Addr - The upper 20 bits (the lower 12 are ignored because it must be 4096- aligned) of the Page Table entry that this Page Directory entry points to.

页表具有相似的结构

* The C bit is the same as the previous D bit
* The D bit is used to mark dirty pages (pages that have been written) by the OS.
* The G flat, if set, prevents caching in the TLB.
* The Addr is the 4096-aligned physical address that this entry points to. The virtual address is calculated from the offset in the page directory and the offset in the page table.

启用分页：

* + 使用页目录表中第一个表项的地址加载CR3（必须为4096对齐）。
  + 设置CR0位31。需要在保护模式下完成。

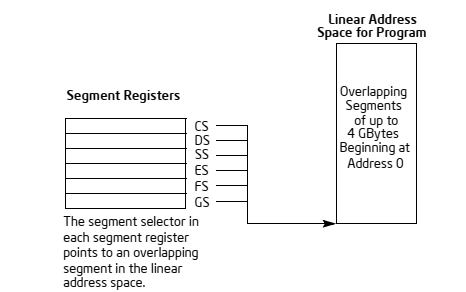
加载页目录表后，硬件会将它们缓存到TLB中。 重新加载CR3将重置缓存。 486+还具有INVLPG指令，仅重置特定的页面缓存，而不是重置整个TLB。

**4.2.2架构**

分段保护模式非常复杂（基于段的内存分配、保护、隔离等）。启用分页后，保护模式可以是扁平的（flat），进而实现以下功能：

* + 所有进程都获得4GB的虚拟地址空间。实现分页级别的保护。所有段均为4GB，所有段选择子始终指向同一个段（即起始地址为0，大小4GB）；
  + 编程非常简单，因为只需要“近”指针即可；
  + 操作系统可以将共享库（驻留在物理内存中一次）映射到多个应用程序的虚拟空间中；
  + 应用程序只看到分页到其自己虚拟地址空间的内存，因此进程直接受硬件保护；

此外，所有现代OS现在仅使用4个环中的2个，内核使用环0，所有用户应用程序使用环3。即两个特权级。



扁平地址模式下的段寄存器使用

**4.2.3 SYSENTER / SYSEXIT**

为了使用户模式（环3）和内核模式（环0）之间的转换更快，必须实现除调用门以外的方法。SYSENTER / SYSEXIT指令是从环3切换到环0的当前方法。需要使用WRMSR为CS（0x174），ESP（0x175）和EIP（0x176）设置新值。ECX必须保留SYSEXIT的环3堆栈指针，而EDX包含SYSEXIT的环3 EIP。为CS存储的表项（代码段描述符）必须是4个段选择子的索引，第一个是ring 0代码，第二个是ring 0数据，第三个是ring 3代码，第四个是ring 4数据。这些值是固定的，因此要使用SYSENTER，GDT表必须以这种格式包含这些条目。

这些操作码仅支持在环3和环0之间切换，但是它们的速度要快得多。当代操作系统使用它们来代替较慢的调用门。

注：MSR（Model Specific Register）处理器提供的多种机器专用的寄存器，用于控制和报告处理器性能（如温度控制，性能监控等）。所有MSR都处理与系统相关的功能，应用程序无法访问。

**4.2.4 软件多任务**

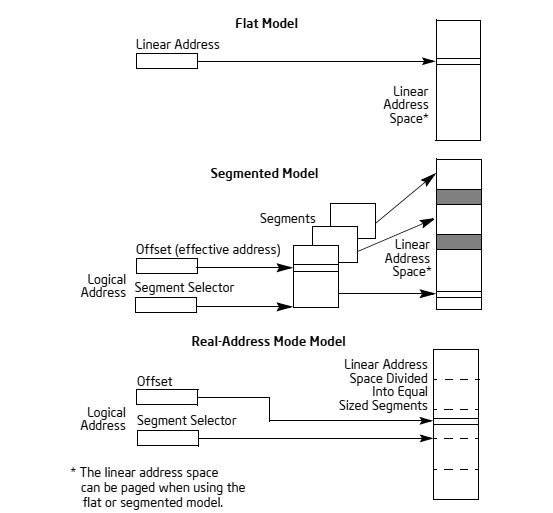
当代的操作系统不再使用任务门，而是采用软件形式的多任务处理，实现进程（线程）之间进行切换：

* + 运行“调度程序”（中断计时器）；
  + 根据线程和进程优先级切换堆栈和EIP。

由于软件形式的调度程序仅保存任务切换所需的内容，所以它比分段模式硬件切换要快得多。

**4.3 内存管理模型比较**

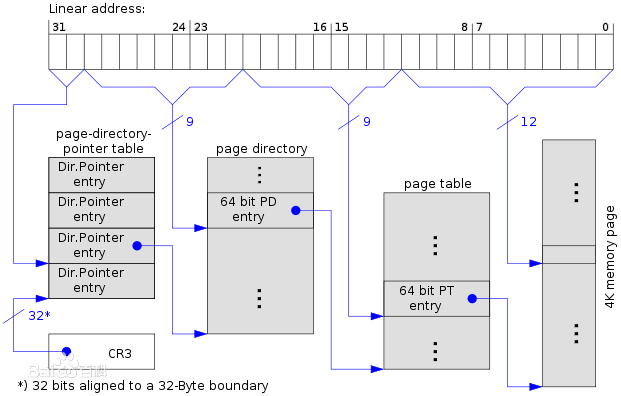
到目前为止，我们已经了解实模式、段保护模式和扁平模式的实现机制。下图展示了三者之间的区别。

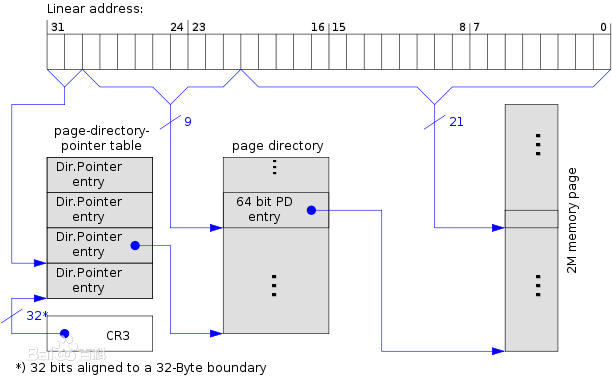


**4.4物理地址扩展（Physical Address Extensions）**

PAE是x86具有使用36位地址而不是32位地址的能力。这将可用内存从4GB增加到64GB。32位应用程序仍然只能看到4GB地址空间，但是OS可以（通过窗口投影）将内存从高区域映射到较低的4GB地址空间。在64位CPU出现之前，此扩展已添加到x86中，以应对4GB地址空间的限制。

启用PAE（CR4位5）意味着现在具有3个分页级别：除了页目录表和页面之外，还有PDTD——页目录指针表，它具有四个64位表项。每个PDTD表项都指向4KB的页目录（与普通页面一样）。现在，新页目录中的每个表项都是64位长（因此只有512个表项）。新的页目录中的每个表项都指向一个4KB的页表（就像在普通页面中一样），新页表中的每个条目现在也都是64位长，因此也只有512个表项。由于只能达到原始映射范围的四分之一，这就是为什么PDTD中包含4个表项的原因。第一个表项映射第一个1GB，第二个映射第二个1GB，第三个映射第三个1GB，最后，第四个条目映射第四个1GB。





但是现在PDT中的“ S”位具有不同的含义：如果未设置，则表示页面表项为4KB；如果设置，则表示该表项未指向PT表项，但其自身描述为2MB页。因此，根据S位，可以具有不同级别的页大小。

页目录表项中还有一个新标志NX位（位63），如果将其置位，则阻止该页中的代码执行。

PAE允许操作系统处理超过4GB的内存，但是由于地址空间仍为4GB，因此每个进程仍限于4GB。内存最大可以为64GB，但是进程无法看到整个内存。

PAE模式下，直接进行内存访问的驱动程序有一个问题，因为它们不使用分页内存。如果以32位模式工作，驱动程序必须自行管理页表，以便能够处理超过4GB的内存，这意味着与操作系统不兼容，除非操作系统向驱动程序公开了安全的DMA API。由于这个原因，PAE迅速淡出了对64位操作系统的支持。

**五、长模式（Long Mode）**

**5.1架构**

无论采用什么方法来克服x86的4GB限制，最终都会导致使用完整的64位处理器。了解了所有保护模式的复杂性之后，马上可以看到，x64 CPU体系结构更加简单。x64 CPU具有3种操作模式：

* 实模式
* 保护模式（称为旧模式）
* 长模式，包含两个子模式：
  + 兼容模式，32位。这使得64位OS可以本机运行32位应用程序。
  + 64位模式

要在长模式下工作，程序员必须考虑以下事实：

* 与可以在有或没有分页的情况下运行的保护模式不同，长模式仅在PAE和分页以及扁平模式下运行。所有段都是平坦的，从0到0xFFFFFFFFFFFFFFFF，所有内存寻址都是线性的。DS，ES，SS被忽略。在长模式下，扁平模式是唯一有效的模式，没有分段。
* 通过在一条指令中启用保护模式和长模式，可以直接从实模式进入长模式（这是由于在实模式下，可以直接访问控制寄存器）。
* 尽管从理论上讲任何64位值都可以用作地址，但实际上目前为止还不需要2 ^ 64的内存。因此，当前的CPU实现仅实现48位寻址，这会强制所有指针的47-63位全为0或全为1。这意味着有2个有效的“规范”地址范围，一个范围为0到0x00007FFF'FFFFFFFF，另一个为从0xFFFF8000'00000000到0xFFFFFFFF'FFFFFFFF，总空间为256TB。大多数操作系统为内核保留上部区域，为用户空间保留下部区域。

**5.2寄存器**

在64位模式下运行时，可以使用以下64位扩展：

     RAX，RBX，RCX，RDX，RSI，RDI，RSP，RBP，RIP

同时还添加了8个新的64位寄存器：R8至R15。可以分别访问R8D-R15D格式的低32位，R8W-R14W格式的高8位，R8B-R14B格式的低8位。

这些寄存器仅在64位模式下可用。在所有其他模式（包括兼容模式）中，它们都不可用。

**5.3长模式分页**

在长模式下，分页系统添加了一个新的顶层结构，即PML4T，它具有512个64位表项，这些表项指向一个PDPT，而现在PDPT也具有512个表项（而不是在x86 PAE模式下为4个）。这意味着一个PT条目管理4KB，一个PDT表项管理2MB（4KB \* 512 PT表项），一个PDPT表项管理1GB（2MB \* 512 PDT表项），一个PML4T表项管理512 GB（ 1GB \* 512个PDPT表项）。由于存在512个PML4T表项，因此总共可以寻址256TB（512GB \* 512个PML4T表项）。

这也是不使用整个64位进行寻址的另一个原因，否则的话，将迫使系统采用6个分页级别，而现在只需要4个分页级别就足够了。

PDPT / PDT中的每个“S”位可以为0，表示下面存在较低级别的结构，也可以为1表示遍历到此结束。如果PDPT S标志为1并且CPU支持这种设置，则页面大小为1GB。

英特尔有一份有关PML5的草案，该草案加入了一种新的顶层结构，当CPU支持56位寻址时，将允许5级分页。

**六、多核**

To be continued.

**七、虚拟化**

To be continued.