第二章　Linux 运行的硬件基础

我们知道，操作系统是一组软件的集合。但它和一般软件不同，因为它是充分挖掘硬件潜能的软件，也可以说，操作系统是横跨软件和硬件的桥梁。因此，要想深入解析操作系统内在的运作机制，就必须搞清楚相关的硬件机制。

操作系统的设计者必须在硬件相关的代码与硬件无关的代码之间划出清楚的界限，以便将一个操作系统很容易地移植到不同的平台。Linux 的设计就做到了这点，它把与硬件相关的代码全部放在 arch（architecture 一词的缩写，即体系结构相关）目录下，在这个目录下，你可以找到 Linux 目前版本支持的所有平台，例如，Linux 2.4 支持的平台有 arm、alpha,、i386、m68k、mips 等十多种。在这众多的平台中，大家熟悉的就是 i386，即 Inte l80386。因此，我们所介绍的硬件基础也是以此为背景的。

在 X86 系列中，8086 和 8088 是 16 位的处理器，而从 80386 开始为 32 位处理器。这种变化看起来是处理器位数的变化，但实质上是处理器体系结构的变化，从寻址方式上说，就是从“ 实模式 ”到“ 保护模式 ”的变化。从 80386 以后，Intel 的 CPU 经历了 80486、Pentium、Pentium II、Pentium III 等型号，虽然它们在速度上提高了好几个数量级，功能上也有不少改进，但基本上属于同一种系统结构的改进与加强，而无本质的变化。因此，我们用 i386 统指这些型号。

2.1　i386 的寄存器

80386 作为 80X86 系列中的一员，必须保证向后兼容，也就是说，既要支持 16 位的处理器，又要支持 32 位的处理器。在 8086 中，所有的寄存器都是 16 位的，下面我们来看一下 780386 中寄存器有何变化。

- 把 16 位的通用寄存器、标志寄存器以及指令指针寄存器扩充为 32 位的寄存器
- 段寄存器仍然为 16 位。
- 增加 4 个 32 位的控制寄存器。
- 增加 4 个系统地址寄存器。
- 增加 8 个调式寄存器。
- 增加 2 个测试寄存器。

2.1.1　通用寄存器

8 个通用寄存器是 8086 寄存器的超集，它们的名称和用途分别为：

- EAX：一般用作累加器。
- EBX：一般用作基址寄存器（Base)。
- ECX：一般用来计数（Count)。
- EDX：一般用来存放数据（Data)。
- EBP：一般用作堆栈指针（Stack Pointer)。
- EBP：一般用作基址指针（Base Pointer)。
- ESI：一般用作源变址（Source Index)。
- EDI：一般用作目标变址（Destinatin Index)。

8 个通用寄存器中通常保存 32 位数据，但为了进行 16 位的操作并与 16 位机保持兼容，它们的低位部分被当成 8 个 16 位的寄存器，即 AX、BX……DI。为了支持 8 位的操作，还进一步把 EAX、EBX、ECX、EDX 这 4 个寄存器低位部分的 16 位，再分为 8 位一组的高位字节和低位字节两部分，作为 8 个 8 位寄存器。这 8 个寄存器分别被命名为 AH、BH、CH、DH 和 AL、BL、CL、DL。对 8 位或 16 位寄存器的操作只影响相应的寄存器。例如，在做 8 位加法运算时，位 7 的进位并不传给目的寄存器的位 9，而是把标志寄存器中的进位标志（CF）置位。因此，这 8 个通用寄存器既可以支持 1 位、8 位、16 位和 32 位数据运算，也支持 16 位和 32 位存储器寻址。

2.1.2 段寄存器

8086 中有 4 个 16 位的段寄存器：CS、DS、SS、ES，分别用于存放可执行代码的代码段、数据段、堆栈段和其他段的基地址。在 80386 中，有 6 个 16 位的段寄存器，但是，这些段寄存器中存放的不再是某个段的基地址，而是某个段的选择符（Selector)。因为 16 位的寄存器无法存放 32 位的段基地址，段基地址只好存放在一个叫做描述符表（Descriptor）的表中。因此，在 80386 中，我们把段寄存器叫做选择符。下面给出 6 个段寄存器的名称和用途。

- CS：代码段寄存器。
- DS：数据段寄存器。
- SS：堆栈段寄存器。
- ES、FS 及 GS：附加数据段寄存器。

有关段选择符、描述符表及系统表地址寄存器将在段机制一节进行详细描述。

2.1.3 状态和控制寄存器

状态和控制寄存器是由标志寄存器（EFLAGS)、指令指针（EIP）和 4 个控制寄存器组成，如图 2.1 所示。

1．指令指针寄存器和标志寄存器

指令指针寄存器（EIP）中存放下一条将要执行指令的偏移量（offset ），这个偏移量是相对于目前正在运行的代码段寄存器（CS）而言的。偏移量加上当前代码段的基地址，就形成了下一条指令的地址。EIP 中的低 16 位可以分开来进行访问，给它起名叫指令指针 IP

寄存器，用于 16 位寻址。

标志寄存器	EFLAGS
指令指针	EIP
机器状态字	CR0
Intel 预留	CR1
页故障地址	CR2
页目录地址	CR3

图 2.1　状态和控制寄存器

标志寄存器（EFLAGS）存放有关处理器的控制标志，如图 2.2 所示。标志寄存器中的第 1、3、5、15 位及 18～31 位都没有定义。

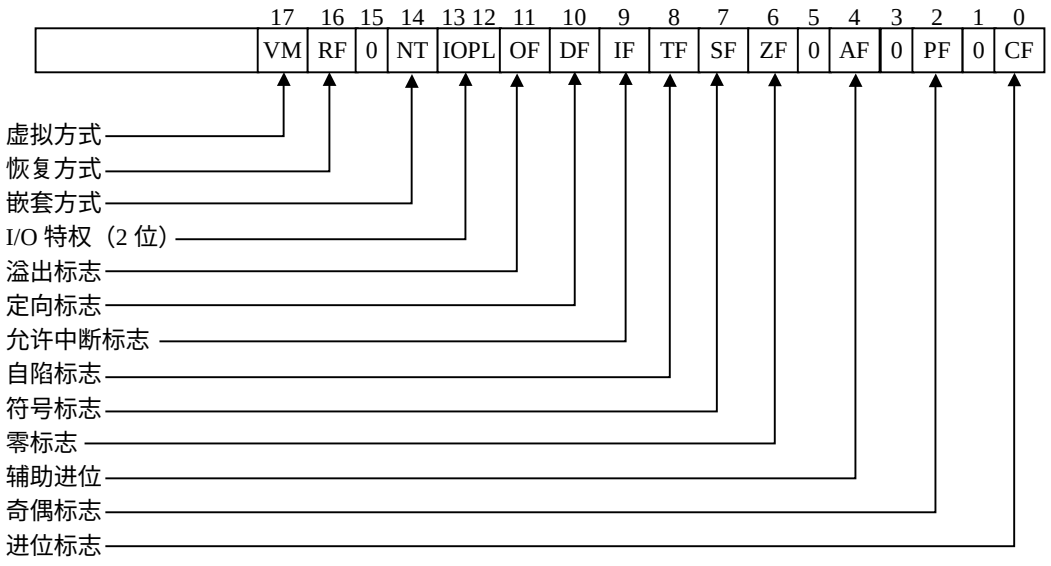

图 2.2　i386 标志寄存器（EFLAGS）

在这些标志位中，我们只介绍在 Linux 内核代码中常用且重要的几个标志位。

第 8 位 TF（Trap Flag）是自陷标志，当将其置 1 时则可以进行单步执行。当指令执行完后，就可能产生异常 1 的自陷（参看第四章）。也就是说，在程序的执行过程中，每执行完一条指令，都要由异常 1 处理程序 （在 Linux 内核中叫做 debug（））进行检验。当将第 8 位清 0 后，且将断点地址装入调试寄存器 DR0～DR3 时，才会产生异常 1 的自陷。

第 12、13 位 IOPL 是输入输出特权级位，这是保护模式下要使用的两个标志位。由于输入输出特权级标志共两位，它的取值范围只可能是 0、1、2 和 3 共 4 个值，恰好与输入输出特权级 0～3 级相对应。但 Linux 内核只使用了两个级别，即 0 和 3 级，0 表示内核级，3 表示用户级。在当前任务的特权级 CPL（Current Privilege Level）高于或等于输入输出特权级时，就可以执行像 IN、OUT、INS、OUTS、STI、CLI 和 LOCK 等指令而不会产生异常 13（即保护异常）。在当前任务特权级 CPL 为 0 时，POPF（从栈中弹出至标志位）指令和中断返回指

令 IRET 可以改变 IOPL 字段的值。

第 9 位 IF（Interrupt Flag）是中断标志位，是用来表示允许或者禁止外部中断（参看第四章）。若第 9 位 IF 被置为 1，则允许 CPU 接收外部中断请求信号；若将 IF 位清 0，则表示禁止外部中断。在保护模式下，只有当第 12、13 位指出当前 CPL 为最高特权级时，才允许将新值置入标志寄存器（EFLAGS）以改变 IF 位的值。

第 10 位 DF（Direction Flag）是定向标志。DF 位规定了在执行串操作的过程中，对源变址寄存器 ESI 或目标变址寄存器 EDI 是增值还是减值。如果 DF 为 1，则寄存器减值；若 DF 为 0，则寄存器值增加。

第 14 位 NT 是嵌套任务标志位。在保护模式下常使用这个标志。当 80386 在发生中断和执行 CALL 指令时就有可能引起任务切换。若是由于中断或由于执行 CALL 指令而出现了任务切换，则将 NT 置为 1。若没有任务切换，则将 NT 位清 0。

第 17 位 VM（Virtual 8086 Mode Flag）是虚拟 8086 方式标志，是 80386 新设置的一个标志位。表示 80386 CPU 是在虚拟 8086 环境中运行。如果 80386 CPU 是在保护模式下运行，而 VM 为又被置成 1，这时 80386 就转换成虚拟 8086 操作方式，使全部段操作就像是在 8086 CPU 上运行一样。VM 位只能由两种方式中的一种方式给予设置，即或者是在保护模式下，由最高特权级（0）级代码段的中断返回指令 IRET 设置，或者是由任务转换进行设置。Linux 内核实现了虚拟 8086 方式，但在本书中我们不准备对此进行详细讨论。

从上面的介绍可以看出，要正确理解标志寄存器（EFLAGS）的各个标志需要很多相关的知识，有些内容在本章的后续部分还会涉及到。在后面的章节中，你会体会如何灵活应用这些标志。

2．控制寄存器

状态和控制寄存器组除了 EFLAGS、EIP，还有 4 个 32 位的控制寄存器，它们是 CR0，CR1、CR2 和 CR3。现在我们详细看看它们的结构，如图 2.3 所示。

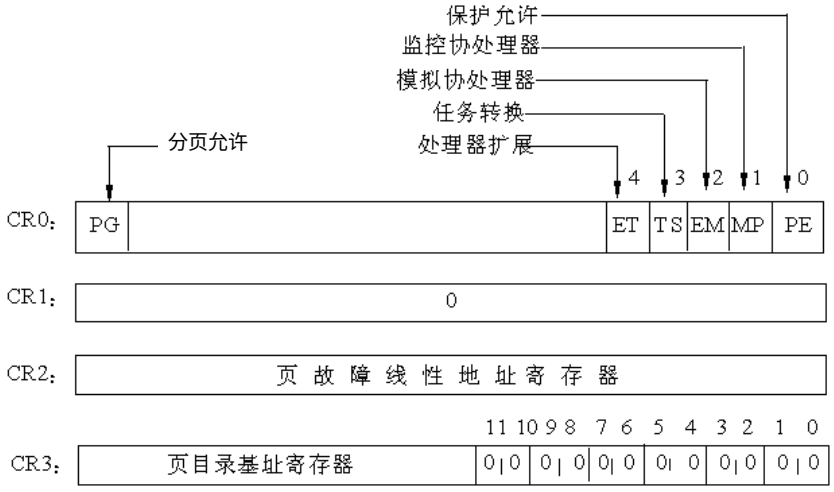

图 2.3　386 中的控制寄存器组

这几个寄存器中保存全局性和任务无关的机器状态。

CR0 中包含了 6 个预定义标志，0 位是保护允许位 PE（Protedted Enable），用于启动保护模式，如果 PE 位置 1，则保护模式启动，如果 PE=0，则在实模式下运行。1 位是监控协处理位 MP（Moniter Coprocessor），它与第 3 位一起决定：当 TS=1 时操作码 WAIT 是否产生一个“协处理器不能使用”的出错信号。第 3 位是任务转换位（Task Switch），当一个任务转换完成之后，自动将它置 1。随着 TS=1，就不能使用协处理器。CR0 的第 2 位是模拟协处理器位 EM （Emulate Coprocessor），如果 EM=1，则不能使用协处理器，如果 EM=0，则允许使用协处理器。第 4 位是微处理器的扩展类型位 ET（Processor Extension Type），其内保存着处理器扩展类型的信息，如果 ET=0，则标识系统使用的是 287 协处理器，如果 ET=1，则表示系统使用的是 387 浮点协处理器。CR0 的第 31 位是分页允许位（Paging Enable），它表示芯片上的分页部件是否允许工作，下一节就会讲到。PG 位和 PE 位定义的操作方式如图 2.4 所示。

PG	PE	方式
0	0	实模式，8080操作
0	1	保护模式，但不允许分页
1	0	出错
1	1	允许分页的保护模式

图 2.4　PG 位和 PE 位定义的操作方式

CR1 是未定义的控制寄存器，供将来的处理器使用。

CR2 是页故障线性地址寄存器，保存最后一次出现页故障的全 32 位线性地址。

CR3 是页目录基址寄存器，保存页目录表的物理地址。页目录表总是放在以 4KB 为单位的存储器边界上，因此，它的地址的低 12 位总为 0，不起作用，即使写上内容，也不会被理会。

这几个寄存器是与分页机制密切相关的，因此，在进程管理及虚拟内存管理中会涉及到这几个寄存器，读者要记住 CR0、CR2 及 CR3 这 3 个寄存器的内容。

2.1.4　系统地址寄存器

80386 有 4 个系统地址寄存器，如图 2.5 所示，它保存操作系统要保护的信息和地址转换表信息。

这 4 个专用寄存器用于引用在保护模式下所需要的表和段，它们的名称和作用如下。

- 全局描述符表寄存器 GDTR（Global Descriptor Table Register），是 48 位寄存器，用来保存全局描述符表（GDT）的 32 位基地址和 16 位 GDT 的界限。
- 中断描述符表寄存器 IDTR（Interrupt Descriptor Table Register），是 48 位寄存器，用来保存中断描述符表（IDT）的 32 位基地址和 16 位 IDT 的界限。

• 局部描述符表寄存器 LDTR（Global Descriptor Table Register），是 16 位寄存器，保存局部描述符表 LDT 段的选择符。

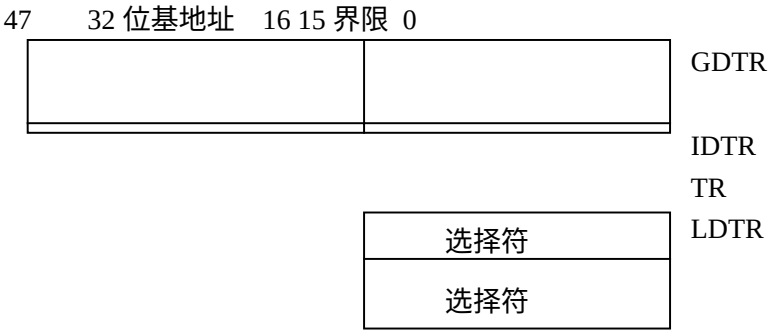

图 2.5　80386 系统地址寄存器

• 任务状态寄存器 TR（Task State Register）是 16 位寄存器，用于保存任务状态段 TSS 段的 16 位选择符。

用以上 4 个寄存器给目前正在执行的任务（或进程）定义任务环境、地址空间和中断向量空间。有关全局描述符表 GST、中断描述符表 IDT、局部描述符表 LDT 及任务状态段 TSS 的具体内容将在稍后进行详细描述。

2.1.5　调试寄存器和测试寄存器

1．调试寄存器

80386 为调试提供了硬件支撑。在 80386 芯片内有 8 个 32 位的调试寄存器 DR0~DR7，如图 2.6 所示。

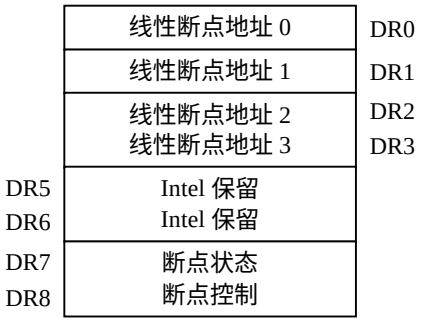

图 2.6　80386 的调试寄存器

这些寄存器可以使系统程序设计人员定义 4 个断点，用它们可以规定指令执行和数据读写的任何组合。DR0~DR3 是线性断点地址寄存器，其中保存着 4 个断点地址。DR4、DR5 是两个备用的调试寄存器，目前尚未定义。DR6 是断点状态寄存器，其低序位是指示符位，

当允许故障调试并检查出故障而进入异常调试处理程序（debug（））时，由硬件把指示符位置 1，调试异常处理程序在退出之前必须把这几位清 0。DR7 是断点控制寄存器，它的高序半个字又被分为 4 个字段，用来规定断点字段的长度是 1 个字节、2 个字节、4 个字节及规定将引起断点的访问类型。低序半个字的位字段用于“ 允许 ”断点和“ 允许 ”所选择的调试条件。

2．测试寄存器

80386 有两个 32 位的测试寄存器 TR6 和 TR7。这两个寄存器用于在转换旁路缓冲器（Translation Lookaside Buffer）中测试随机存储器（RAM）和相联存储器（CAM）。TR6 是测试命令寄存器，其内存放测试控制命令。TR7 是数据寄存器，其内保存转换旁路缓冲器测试的数据。

2.2　内存地址

在任何一台计算机上，都存在一个程序能产生的内存地址的集合。当程序执行这样一条指令时：

```
MOVE  REG，ADDR
```

它把地址为 ADDR（假设为 10000）的内存单元的内容复制到 REG 中，地址 ADDR 可以通过索引、基址寄存器、段寄存器和其他方式产生。

在 8086 的实模式下，把某一段寄存器左移 4 位，然后与地址 ADDR 相加后被直接送到内存总线上，这个相加后的地址就是内存单元的物理地址，而程序中的这个地址就叫逻辑地址（或叫虚地址）。在 80386 的保护模式下，这个逻辑地址不是被直接送到内存总线，而是被送到内存管理单元（MMU）。MMU 由一个或一组芯片组成，其功能是把逻辑地址映射为物理地址，即进行地址转换，如图 2.7 所示。

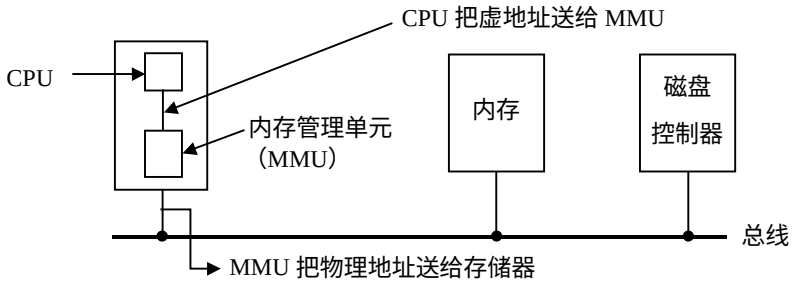

图 2.7　MMU 的位置和功能

当使用 80386 时，我们必须区分以下 3 种不同的地址。

1．逻辑地址

机器语言指令仍用这种地址指定一个操作数的地址或一条指令的地址。这种寻址方式在

Intel 的分段结构中表现得尤为具体，它使得 MS-DOS 或 Windows 程序员把程序分为若干段。每个逻辑地址都由一个段和偏移量组成。

2．线性地址

线性地址是一个 32 位的无符号整数，可以表达高达 2^{32}（4GB）的地址。通常用 16 进制表示线性地址，其取值范围为 0x00000000～0xffffffff。

3．物理地址

物理地址是内存单元的实际地址，用于芯片级内存单元寻址。物理地址也由 32 位无符号整数表示。

从图 2.7 可以看出，MMU 是一种硬件电路，它包含两个部件，一个是分段部件，一个是分页部件，在本书中，我们把它们分别叫做分段机制和分页机制，以利于从逻辑的角度来理解硬件的实现机制。分段机制把一个逻辑地址转换为线性地址；接着，分页机制把一个线性地址转换为物理地址，如图 2.8 所示。

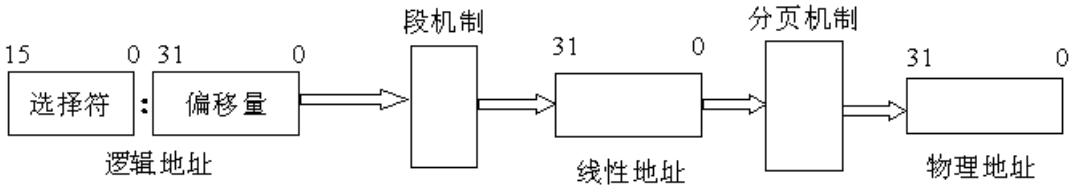

图 2.8　MMU 把逻辑地址转换为物理地址

2.3　段机制和描述符

2.3.1　段机制

在 80386 的段机制中，逻辑地址由两部分组成，即段部分（选择符）及偏移部分。

段是形成逻辑地址到线性地址转换的基础。如果我们把段看成一个对象的话，那么对它的描述如下。

(1) 段的基地址（Base Address）：在线性地址空间中段的起始地址。

(2) 段的界限（Limit）：表示在逻辑地址中，段内可以使用的最大偏移量。

(3) 段的属性（Attribute）： 表示段的特性。例如，该段是否可被读出或写入，或者该段是否作为一个程序来执行，以及段的特权级等。

段的界限定义逻辑地址空间中段的大小。段内在偏移量从 0 到 limit 范围内的逻辑地址，对应于从 Base 到 Base+Limit 范围内的线性地址。在一个段内，偏移量大于段界限的逻辑地址将没有意义，使用这样的逻辑地址，系统将产生异常。另外，如果要对一个段进行访问，

系统会根据段的属性检查访问者是否具有访问权限，如果没有，则产生异常。例如，在 80386 中，如果要在只读段中进行写入，80386 将根据该段的属性检测到这是一种违规操作，则产生异常。

图 2.9 表示一个段如何从逻辑地址空间，重新定位到线性地址空间。图的左侧表示逻辑地址空间，定义了 A、B 及 C 三个段，段容量分别为 $Limit_A$、$Limit_B$ 及 $Limit_C$。图中虚线把逻辑地址空间中的段 A、B 及 C 与线性地址空间区域连接起来表示了这种转换。

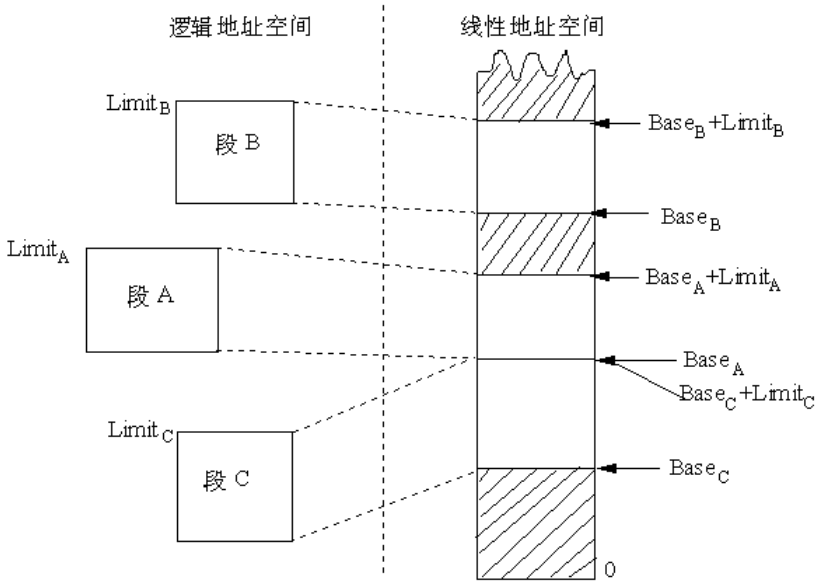

图 2.9　逻辑—线性地址转换

段的基地址、界限及保护属性，存储在段的描述符表中，在逻辑—线性地址转换过程中要对描述符进行访问。段描述符又存储在存储器的段描述符表中，该描述符表是段描述符的一个数组，关于这些内容，我们将在后面详细介绍。

2.3.2　描述符的概念

所谓描述符（Descriptor），就是描述段的属性的一个 8 字节存储单元。在实模式下，段的属性不外乎是代码段、堆栈段、数据段、段的起始地址、段的长度等，而在保护模式下则复杂一些。80386 将它们结合在一起用一个 8 字节的数表示，称为描述符。80386 的一个通用的段描述符的结构如图 2.10 所示。

从图可以看出，一个段描述符指出了段的 32 位基地址和 20 位段界限（即段长）。

第 6 个字节的 G 位是粒度位，当 G=0 时，段长表示段格式的字节长度，即一个段最长可达 1M 字节。当 G=1 时，段长表示段的以 4K 字节为一页的页的数目，即一个段最长可达 1M×4K=4G 字节。D 位表示缺省操作数的大小，如果 D=0，操作数为 16 位，如果 D=1，操作数为 32 位。第 6 个字节的其余两位为 0，这是为了与将来的处理器兼容而必须设置为 0 的位。

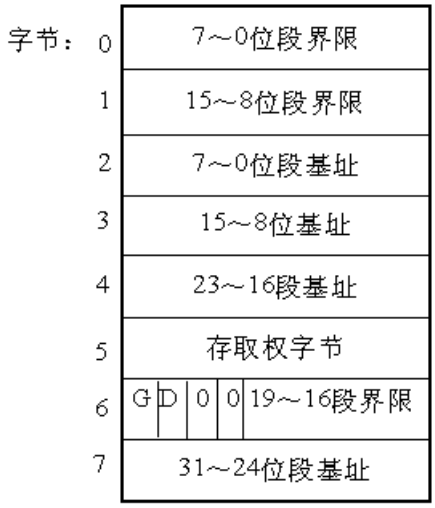

图 2.10　段描述符的一般格式

第 5 个字节是存取权字节，它的一般格式如图 2.11 所示。

图 2.11 存取权字节的一般格式

第 7 位 P 位（Present） 是存在位，表示段描述符描述的这个段是否在内存中，如果在内存中。P=1；如果不在内存中，P=0。

DPL（Descriptor Privilege Level），就是描述符特权级，它占两位，其值为 0～3，用来确定这个段的特权级即保护等级。

S 位（System）表示这个段是系统段还是用户段。如果 S=0，则为系统段，如果 S=1，则为用户程序的代码段、数据段或堆栈段。系统段与用户段有很大的不同，后面会具体介绍。

类型占 3 位，第 3 位为 E 位，表示段是否可执行。当 E=0 时，为数据段描述符，这时的第 2 位 ED 表示扩展方向。当 ED=0 时，为向地址增大的方向扩展，这时存取数据段中的数据的偏移量必须小于或等于段界限，当 ED=1 时，表示向地址减少的方向扩展，这时偏移量必须大于界限。当表示数据段时，第 1 位（W）是可写位，当 W=0 时，数据段不能写，W=1 时，数据段可写入。在 80386 中，堆栈段也被看成数据段，因为它本质上就是特殊的数据段。当描述堆栈段时，ED=0，W=1，即堆栈段朝地址增大的方向扩展。

也就是说，当段为数据段时，存取权字节的格式如图 2.12 所示。

当段为代码段时，第 3 位 E=1，这时第 2 位为一致位（C）。当 C=1 时，如果当前特权级低于描述符特权级，并且当前特权级保持不变，那么代码段只能执行。所谓当前特权级（Current Privilege Level），就是当前正在执行的任务的特权级。第 1 位为可读位 R，当 R=0 时，代码段不能读，当 R=1 时可读。也就是说，当段为代码段时，存取权字节的格式如

图 2.13 所示。

图 2-12 数据段的存取字节

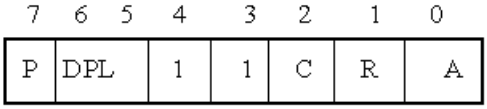

图 2.13　代码段的存取字节

存取权字节的第 0 位 A 位是访问位，用于请求分段不分页的系统中，每当该段被访问时，将 A 置 1。对于分页系统，则 A 被忽略未用。

2.3.3　系统段描述符

以上介绍了用户段描述符。系统段描述符的一般格式如图 2.14 所示。

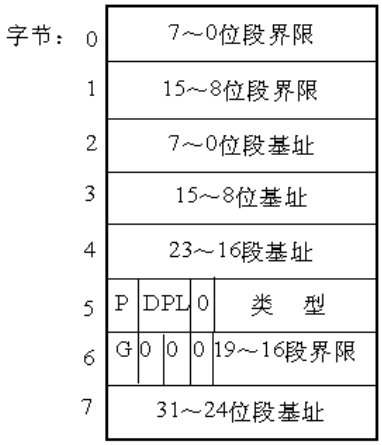

图 2.14　系统段描述符的一般格式

可以看出，系统段描述符的第 5 个字节的第 4 位为 0，说明它是系统段描述符，类型占 4 位，没有 A 位。第 6 个字节的第 6 位为 0，说明系统段的长度是字节粒度，所以，一个系统段的最大长度为 1M 字节。

系统段的类型为 16 种，如图 2.15 所示。

在这 16 种类型中，保留类型和有关 286 的类型不予考虑。

门也是一种描述符,有调用门、任务门、中断门和陷阱门 4 种门描述符。有关门描述符的内容将在第四章中进行具体讨论。

类型号	定义	类型号	定义
0	未定义(Intel公司保留)	8	未定义(Intel公司保留)
1	有效的286TSS	9	有效的386TSS
2	LDT	A	386TSS忙
3	286TSS忙	B	未定义(Intel公司保留)
4	286调用门	C	386调用门
5	任务门	D	未定义(Intel公司保留)
6	286中断门	E	386中断门
7	286陷阱门	F	386陷阱门

图 2.15　系统段的类型

2.3.4　描述符表

各种各样的用户描述符和系统描述符，都放在对应的全局描述符表、局部描述符表和中断描述符表中。

描述符表（即段表）定义了 386 系统的所有段的情况。所有的描述符表本身都占据一个字节为 8 的倍数的存储器空间，空间大小在 8 个字节（至少含一个描述符）到 64K 字节（至多含 8K）个描述符之间。

1．全局描述符表（GDT）

全局描述符表 GDT（Global Descriptor Table），除了任务门，中断门和陷阱门描述符外，包含着系统中所有任务都共用的那些段的描述符。它的第一个 8 字节位置没有使用。

2．中断描述符表（IDT）

中断描述符表 IDT（Interrupt Descriptor Table），包含 256 个门描述符。IDT 中只能包含任务门、中断门和陷阱门描述符，虽然 IDT 表最长也可以为 64K 字节，但只能存取 2K 字节以内的描述符，即 256 个描述符，这个数字是为了和 8086 保持兼容。

3．局部描述符表（LDT）

局部描述符表 LDT（Local Descriptor Table），包含了与一个给定任务有关的描述符，每个任务各自有一个的 LDT。有了 LDT，就可以使给定任务的代码、数据与别的任务相隔离。

每一个任务的局部描述符表 LDT 本身也用一个描述符来表示，称为 LDT 描述符，它包含了有关局部描述符表的信息，被放在全局描述符表 GDT 中。

2.3.5　选择符与描述符表寄存器

在实模式下，段寄存器存储的是真实的段地址，在保护模式下，16 位的段寄存器无法放

下 32 位的段地址，因此，它们被称为选择符，即段寄存器的作用是用来选择描述符。选择符的结构如图 2.16 所示。

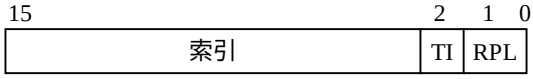

图 2.16　选择符的结构

可以看出，选择符有 3 个域：第 15~3 位这 13 位是索引域，表示的数据为 0~8129，用于指向全局描述符表中相应的描述符。第 2 位为选择域，如果 TI=1，就从局部描述符表中选择相应的描述符，如果 TI=0，就从全局描述符表中选择描述符。第 1、0 位是特权级，表示选择符的特权级，被称为请求者特权级 RPL（Requestor Privilege Level）。只有请求者特权级 RPL 高于（数字低于）或等于相应的描述符特权级 DPL，描述符才能被存取，这就可以实现一定程度的保护。

我们知道，实模式下是直接在段寄存器中放置段基地址，现在则是通过它来存取相应的描述符来获得段基地址和其他信息，这样以来，存取速度会不会变慢呢？为了解决这个问题，386 的每一个段选择符都有一个程序员不可见（也就是说程序员不能直接操纵）的 88 位宽的段描述符高速缓冲寄存器与之对应。无论什么时候改变了段寄存器的内容，只要特权级合理，描述符表中的相应的 8 字节描述符就会自动从描述符表中取出来，装入高速缓冲寄存器中（还有 24 位其他内容）。一旦装入，以后对那个段的访问就都使用高速缓冲寄存器的描述符信息，而不会再重新从表中去取，这就大大加快了执行的时间，如图 2.17 所示。

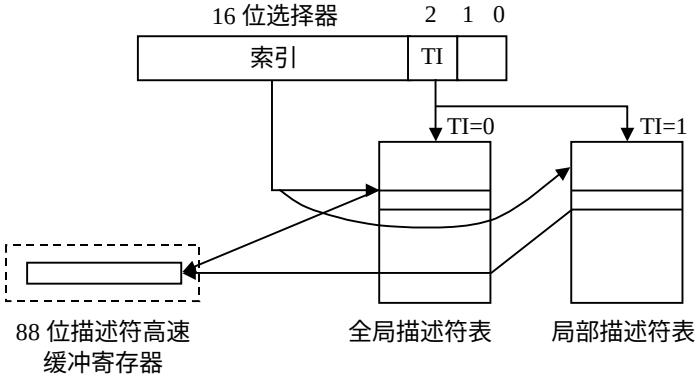

图 2.17　段描述符高速缓冲寄存器的作用

由于段描述符高速缓冲寄存器的内容只有在重新设置选择符时才被重新装入，所以，当你修改了选择符所选择的描述符后，必须对相应的选择符重新装入，这样，88 位描述符高速缓冲寄存器的内容才会发生变化。无论如何，当选择符的值改变时，处理器自动装载不可见部分。

下面讲一下在没有分页操作时，寻址一个存储器操作数的步骤。

（1）在段选择符中装入 16 位数，同时给出 32 位地址偏移量（比如在 ESI、EDI 中等）。

（2）根据段选择符中的索引值、TI 及 RPL 值，再根据相应描述符表寄存器中的段地址和

段界限，进行一系列合法性检查（如特权级检查、界限检查），该段无问题，就取出相应的描述符放入段描述符高速缓冲寄存器中。

（4）将描述符中的 32 位段基地址和放在 ESI、EDI 等中的 32 位有效地址相加，就形成了 32 位物理地址。

注意：在保护模式下，32 位段基地址不必向左移 4 位，而是直接和偏移量相加形成 32 位物理地址（只要不溢出）。这样做的好处是：段不必再定位在被 16 整除的地址上，也不必左移 4 位再相加。

寻址过程如图 2.18 所示。

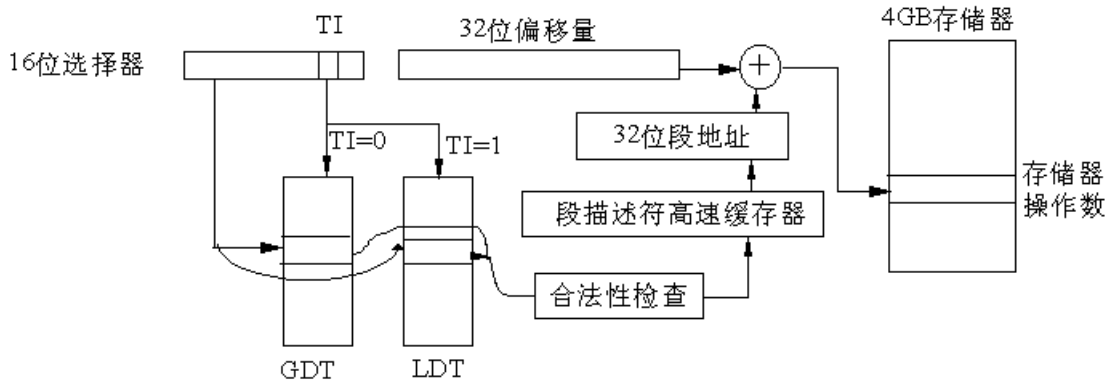

图 2.18　寻址过程

2.3.6　描述符投影寄存器

为了避免在每次存储器访问时，都要访问描述符表，读出描述符并对段进行译码以得到描述符本身的各种信息，每个段寄存器都有与之相联系的描述符投影寄存器。在这些寄存器中，容纳有由段寄存器中的选择符确定的段的描述符信息。段寄存器对编程人员是可见的，而与之相联系的容纳描述符的寄存器，则对编程人员是不可见的，故称之为投影寄存器。图 2.19 中所示的是 6 个寄存器及其投影寄存器。用实线画出的寄存器是段寄存器，用以表示这些寄存器对编程人员可见；用虚线画出的寄存器是投影寄存器，表示对编程人员不可见。

投影寄存器容纳有相应段寄存器寻址的段的基地址、界限及属性。每当用选择符装入段寄存器时，CPU 硬件便自动地把描述符的全部内容装入对应的投影寄存器。因此，在多次访问同一段时，就可以用投影寄存器中的基地址来访问存储器。投影寄存器存储在 80386 的芯片上，因而可以由段基址硬件进行快速访问。因为多数指令访问的数据是在其选择符已经装入到段寄存器之后进行的，所以使用投影寄存器可以得到很好的执行性能。

2.3.7　Linux 中的段

Intel 微处理器的段机制是从 8086 开始提出的， 那时引入的段机制解决了从 CPU 内部 16 位地址到 20 位实地址的转换。为了保持这种兼容性，386 仍然使用段机制，但比以前复杂

得多。因此，Linux 内核的设计并没有全部采用 Intel 所提供的段方案，仅仅有限度地使用了一下分段机制。这不仅简化了 Linux 内核的设计，而且为把 Linux 移植到其他平台创造了条件，因为很多 RISC 处理器并不支持段机制。但是，对段机制相关知识的了解是进入 Linux 内核的必经之路。

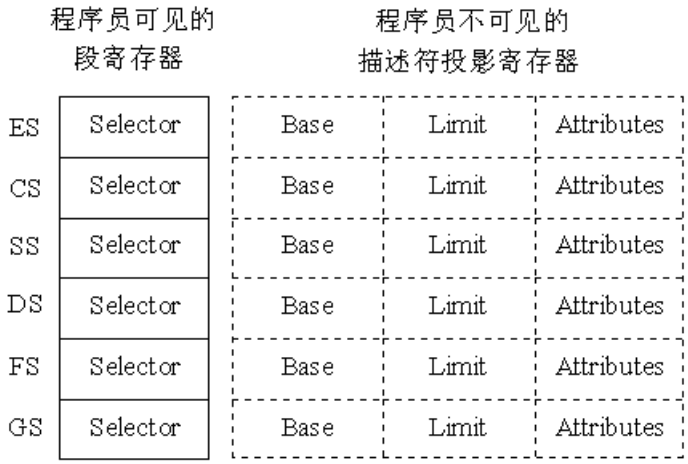

图 2.19 描述符投影寄存器

从 2.2 版开始，Linux 让所有的进程（或叫任务）都使用相同的逻辑地址空间，因此就没有必要使用局部描述符表 LDT。但内核中也用到 LDT，那只是在 VM86 模式中运行 Wine 时，即在 Linux 上模拟运行 Windows 软件或 DOS 软件的程序时才使用。

Linux 在启动的过程中设置了段寄存器的值和全局描述符表 GDT 的内容，段的定义在 include/asm-i386/segment.h 中：

```
#define __KERNEL_CS0x10      ／＊内核代码段，index=2,TI=0,RPL=0＊／
#define __KERNEL_DS0x18      ／＊内核数据段，index=3,TI=0,RPL=0＊／
#define __USER_CS  0x23      ／＊用户代码段，index=4,TI=0,RPL=3＊／
#define __USER_DS  0x2B      ／＊用户数据段，index=5,TI=0,RPL=3＊／
```

从定义看出，没有定义堆栈段，实际上，Linux 内核不区分数据段和堆栈段，这也体现了 Linux 内核尽量减少段的使用。因为没有使用 LDT，因此，TI=0，并把这 4 个段都放在 GDT 中，index 就是某个段在 GDT 表中的下标。内核代码段和数据段具有最高特权，因此其 RPL 为 0，而用户代码段和数据段具有最低特权，因此其 RPL 为 3。可以看出，Linux 内核再次简化了特权级的使用，使用了两个特权级而不是 4 个。

全局描述符表的定义在 arch/i386/kernel/head.S 中：

```
ENTRY (gdt_table)
   .quad 0x0000000000000000   /* NULL descriptor */
   .quad 0x0000000000000000   /* not used */
   .quad 0x00cf9a000000ffff   /* 0x10 kernel 4GB code at 0x00000000 */
   .quad 0x00cf92000000ffff   /* 0x18 kernel 4GB data at 0x00000000 */
   .quad 0x00cffa000000ffff   /* 0x23 user   4GB code at 0x00000000 */
   .quad 0x00cff2000000ffff   /* 0x2b user   4GB data at 0x00000000 */
   .quad 0x0000000000000000   /* not used */
```

```
        .quad 0x0000000000000000   /* not used */
        /*
         * The APM segments have byte granularity and their bases
         * and limits are set at run time.
        */
        .quad 0x0040920000000000   /* 0x40 APM set up for bad BIOS's */
        .quad 0x00409a0000000000   /* 0x48 APM CS    code */
        .quad 0x00009a0000000000   /* 0x50 APM CS 16 code  (16 bit)  */
        .quad 0x0040920000000000   /* 0x58 APM DS    data */
        .fill NR_CPUS*4,8,0   /* space for TSS's and LDT's */
```

从代码可以看出，GDT 放在数组变量 gdt_table 中。按 Intel 规定，GDT 中的第一项为空，这是为了防止加电后段寄存器未经初始化就进入保护模式而使用 GDT 的。第二项也没用。从下标 2～5 共 4 项对应于前面的 4 种段描述符值。对照图 2.10，从描述符的数值可以得出：

- 段的基地址全部为 0x00000000；
- 段的上限全部为 0xffff；
- 段的粒度 G 为 1，即段长单位为 4KB；
- 段的 D 位为 1，即对这 4 个段的访问都为 32 位指令；
- 段的 P 位为 1，即 4 个段都在内存。

由此可以得出，每个段的逻辑地址空间范围为 0～4GB。读者可能对此不太理解，但只要对照图 2.9 就可以发现，这种设置既简单又巧妙。因为每个段的基地址为 0，因此，逻辑地址到线性地址映射保持不变，也就是说，偏移量就是线性地址，我们以后所提到的逻辑地址（或虚拟地址）和线性地址指的也就是同一地址。看来，Linux 巧妙地把段机制给绕过去了，而完全利用了分页机制。

从逻辑上说，Linux 巧妙地绕过了逻辑地址到线性地址的映射，但实质上还得应付 Intel 所提供的段机制。只不过，Linux 把段机制变得相当简单，它只把段分为两种：用户态（RPL＝3）的段和内核态（RPL=0）的段，因此，描述符投影寄存器的内容很少发生变化，只在进程从用户态切换到内核态或者反之时才发生变化。另外，用户段和内核段的区别也仅仅在其 RPL 不同，因此内核根本无需访问描述符投影寄存器，当然也无需访问 GDT，而仅从段寄存器的最低两位就可以获取 RPL 的信息。Linux 这样设计所带来的好处是显而易见的，Intel 的分段部件对 Linux 性能造成的影响可以忽略不计。

在上面描述的 GDT 表中，紧接着那 4 个段描述的两个描述符被保留，然后是 4 个高级电源管理（APM）特征描述符，对此不进行详细讨论。

按 Intel 的规定，每个进程有一个任务状态段（TSS）和局部描述符表 LDT，但 Linux 也没有完全遵循 Intel 的设计思路。如前所述，Linux 的进程没有使用 LDT，而对 TSS 的使用也非常有限，每个 CPU 仅使用一个 TSS。

通过上面的介绍可以看出，Intel 的设计可谓周全细致，但 Linux 的设计者并没有完全陷入这种沼泽，而是选择了简洁而有效的途径，以完成所需功能并达到较好的性能为目标。

2.4　分页机制

分页机制在段机制之后进行，以完成线性—物理地址的转换过程。段机制把逻辑地址转换为线性地址，分页机制进一步把该线性地址再转换为物理地址。

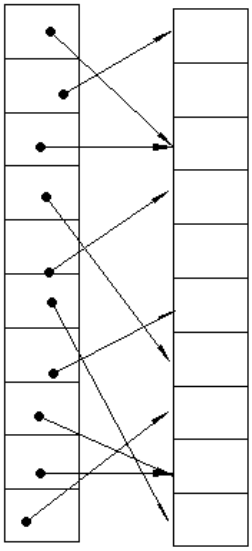

图 2.20　分页机制把线性地址转换为物理地址

分页机制由 CR0 中的 PG 位启用。如 PG=1，启用分页机制，并使用本节要描述的机制，把线性地址转换为物理地址。如 PG=0，禁用分页机制，直接把段机制产生的线性地址当作物理地址使用。分页机制管理的对象是固定大小的存储块，称之为页（page）。分页机制把整个线性地址空间及整个物理地址空间都看成由页组成，在线性地址空间中的任何一页，可以映射为物理地址空间中的任何一页（我们把物理空间中的一页叫做一个页面或页框（page frame））。图 2.20 所示出分页机制如何把线性地址空间及物理地址空间划分为页，以及如何在这两个地址空间进行映射。图 2.20 的左边是线性地址空间，并将其视为一个具有一页大小的固定块的序列。图 2.20 的右边是物理地址空间，也将其视为一个页面的序列。图中，用箭头把线性地址空间中的页，与对应的物理地址空间中的页面联系起来。在这里，线性地址空间中的页与物理地址空间中的页是随意地对应起来的。

80386 使用 4K 字节大小的页。每一页都有 4K 字节长，并在 4K 字节的边界上对齐，即每一页的起始地址都能被 4K 整除。因此，80386 把 4G 字节的线性地址空间，划分为 1G 个页面，每页有 4K 字节大小。分页机制通过把线性地址空间中的页，重新定位到物理地址空间来进行管理，因为每个页面的整个 4K 字节作为一个单位进行映射，并且每个页面都对齐 4K 字节的边界，因此，线性地址的低 12 位经过分页机制直接地作为物理地址的低 12 位使用。

线性—物理地址的转换，可将其意义扩展为允许将一个线性地址标记为无效，而不是实际地产生一个物理地址。有两种情况可能使页被标记为无效：其一是线性地址是操作系统不

支持的地址；其二是在虚拟存储器系统中，线性地址对应的页存储在磁盘上，而不是存储在物理存储器中。在前一种情况下，程序因产生了无效地址而必须被终止。对于后一种情况，该无效的地址实际上是请求操作系统的虚拟存储管理系统，把存放在磁盘上的页传送到物理存储器中，使该页能被程序所访问。由于无效页通常是与虚拟存储系统相联系的，这样的无效页通常称为未驻留页，并且用页表属性位中叫做存在位的属性位进行标识。未驻留页是程序可访问的页，但它不在主存储器中。对这样的页进行访问，形式上是发生异常，实际上是通过异常进行缺页处理。

2.4.1 分页机构

如前所述，分页是将程序分成若干相同大小的页，每页 4K 个字节。如果不允许分页（CR0 的最高位置 0），那么经过段机制转化而来的 32 位线性地址就是物理地址。但如果允许分页（CR0 的最高位置 1），就要将 32 位线性地址通过一个两级表格结构转化成物理地址。

1．两级页表结构

为什么采用两级页表结构呢？

在 80386 中页表共含 1M 个表项，每个表项占 4 个字节。如果把所有的页表项存储在一个表中，则该表最大将占 4M 字节连续的物理存储空间。为避免使页表占有如此巨额的物理存储器资源，故对页表采用了两级表的结构，而且对线性地址的高 20 位的线性—物理地址转化也分为两部完成，每一步各使用其中的 10 位。

两级表结构的第一级称为页目录，存储在一个 4K 字节的页面中。页目录表共有 1K 个表项，每个表项为 4 个字节，并指向第二级表。线性地址的最高 10 位（即位 31～位 22）用来产生第一级的索引，由索引得到的表项中，指定并选择了 1K 个二级表中的一个表。

两级表结构的第二级称为页表，也刚好存储在一个 4K 字节的页面中，包含 1K 个字节的表项，每个表项包含一个页的物理基地址。第二级页表由线性地址的中间 10 位（即位 21~位 12）进行索引，以获得包含页的物理地址的页表项，这个物理地址的高 20 位与线性地址的低 12 位形成了最后的物理地址，也就是页转化过程输出的物理地址，具体转化过程稍后会讲到，如图 2.21 为两级页表结构。

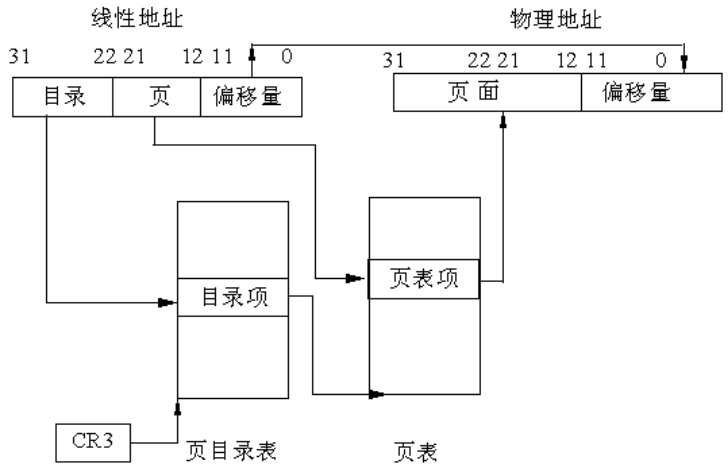

图 2.21　两级页表结构

2．页目录项

图 2-22 所示为页目录表，最多可包含 1024 个页目录项，每个页目录项为 4 个字节，结构如图 2.22 所示。

- 第 31~12 位是 20 位页表地址，由于页表地址的低 12 位总为 0，所以用高 20 位指出 32 位页表地址就可以了。因此，一个页目录最多包含 1024 个页表地址。
- 第 0 位是存在位，如果 P=1，表示页表地址指向的该页在内存中，如果 P=0，表示不在内存中。
- 第 1 位是读/写位，第 2 位是用户/管理员位，这两位为页目录项提供硬件保护。当特权级为 3 的进程要想访问页面时，需要通过页保护检查，而特权级为 0 的进程就可以绕过页保护，如图 2.23 所示。
- 第 3 位是 PWT（Page Write-Through）位，表示是否采用写透方式，写透方式就是既写内存（RAM）也写高速缓存,该位为 1 表示采用写透方式。
- 第 4 位是 PCD（Page Cache Disable）位，表示是否启用高速缓存，该位为 1 表示启用高速缓存。

图 2.22　页目录中的页目录项

U/S	R/W	允许级别3	允许级别0
0	0	无	读/写
0	1	无	读/写
1	0	只读	读/写
1	1	读/写	读/写

图 2.23　由 U/S 和 R/W 提供的保护

- 第 5 位是访问位，当对页目录项进行访问时，A 位=1。
- 第 7 位是 Page Size 标志，只适用于页目录项。如果置为 1，页目录项指的是 4MB 的页面，请看后面的扩展分页。
- 第 9~11 位由操作系统专用，Linux 也没有做特殊之用。

2．页面项

80386 的每个页目录项指向一个页表，页表最多含有 1024 个页面项，每项 4 个字节，包含页面的起始地址和有关该页面的信息。页面的起始地址也是 4K 的整数倍，所以页面的低 12 位也留作它用，如图 2.24 所示。

	7	6	5	4	3	2	1	0
7~0位	0	D	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页面地址				OS专用			0
23~16位	11~4位页面地址							
31~24位	19~12位页面地址							

图 2.24　页表中的页面项

第 31～12 位是 20 位物理页面地址，除第 6 位外第 0～5 位及 9～11 位的用途和页目录项一样，第 6 位是页面项独有的，当对涉及的页面进行写操作时，D 位被置 1。

4GB 的存储器只有一个页目录，它最多有 1024 个页目录项，每个页目录项又含有 1024 个页面项，因此，存储器一共可以分成 1024×1024=1M 个页面。由于每个页面为 4K 个字节，所以，存储器的大小正好最多为 4GB。

3．线性地址到物理地址的转换

当访问一个操作单元时，如何由分段结构确定的 32 位线性地址通过分页操作转化成 32 位物理地址呢？过程如图 2.25 所示。

第一步，CR3 包含着页目录的起始地址，用 32 位线性地址的最高 10 位 A31~A22 作为页目录的页目录项的索引，将它乘以 4，与 CR3 中的页目录的起始地址相加，形成相应页表的地址。

第二步，从指定的地址中取出 32 位页目录项，它的低 12 位为 0，这 32 位是页表的起始地址。用 32 位线性地址中的 A21~A12 位作为页表中的页面的索引，将它乘以 4，与页表的起始地址相加，形成 32 位页面地址。

第三步，将 A11~A0 作为相对于页面地址的偏移量，与 32 位页面地址相加，形成 32 位物理地址。

4．扩展分页

从奔腾处理器开始，Intel 微处理器引进了扩展分页，它允许页的大小为 4MB，如图 2.26 所示。

在扩展分页的情况下，分页机制把 32 位线性地址分成两个域：最高 10 位的目录域和其余 22 位的偏移量。

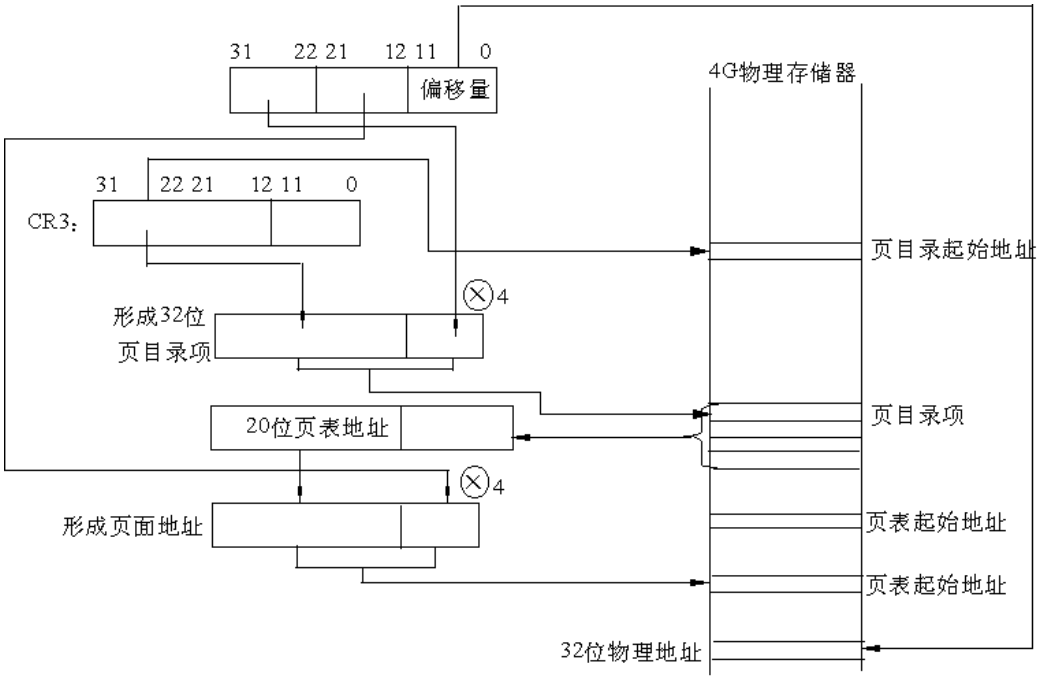

图 2.25　32 位线性地址到物理地址的转换

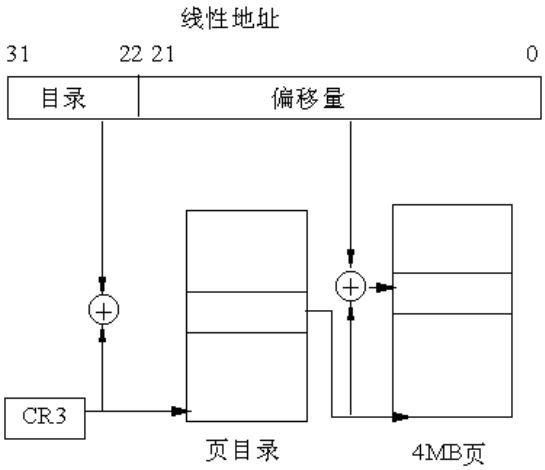

图 2.26　扩展分页

2.4.2　页面高速缓存

由于在分页情况下，每次存储器访问都要存取两级页表，这就大大降低了访问速度。所以，为了提高速度，在 386 中设置一个最近存取页面的高速缓存硬件机制，它自动保持 32 项处理器最近使用的页面地址，因此，可以覆盖 128K 字节的存储器地址。当进行存储器访问时，先检查要访问的页面是否在高速缓存中，如果在，就不必经过两级访问了，如果不在，再进行两级访问。平均来说，页面高速缓存大约有 98%的命中率，也就是说每次访问存储器时，只有 2%的情况必须访问两级分页机构。这就大大加快了速度，页面高速缓存的作用如图 2.27 所示。有些书上也把页面高速缓存叫做“联想存储器”或“转换旁路缓冲器(TLB)”。

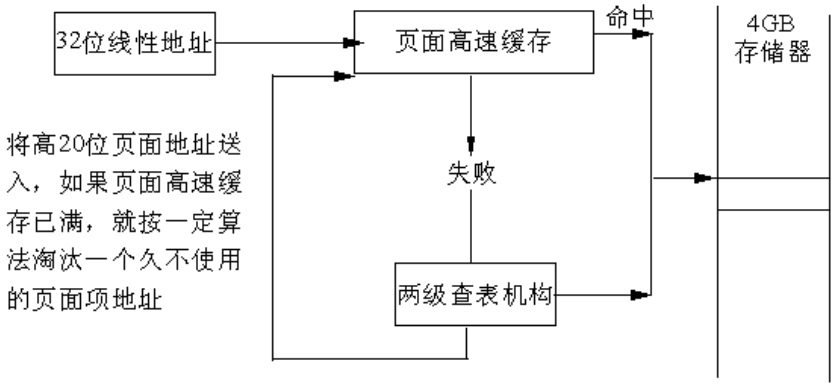

图 2.27　子页面高速缓存

2.5　Linux 中的分页机制

如前所述，Linux 主要采用分页机制来实现虚拟存储器管理，原因如下。

- Linux 的分段机制使得所有的进程都使用相同的段寄存器值，这就使得内存管理变得简单，也就是说，所有的进程都使用同样的线性地址空间（0～4GB）。
- Linux 设计目标之一就是能够把自己移植到绝大多数流行的处理器平台。但是，许多 RISC 处理器支持的段功能非常有限。

为了保持可移植性，Linux 采用三级分页模式而不是两级，这是因为许多处理器（如康柏的 Alpha，Sun 的 UltraSPARC，Intel 的 Itanium）都采用 64 位结构的处理器，在这种情况下，两级分页就不适合了，必须采用三级分页。如图 2.28 所示为三级分页模式，为此，Linux 定义了 3 种类型的页表。

- 总目录 PGD（Page Global Directory）
- 中间目录 PMD（Page Middle Derectory）
- 页表 PT（Page Table）

尽管 Linux 采用的是三级分页模式，但我们的讨论还是以 Intel 奔腾处理器的两级分页模式为主，因此，Linux 忽略中间目录层，以后，我们把总目录就叫页目录。

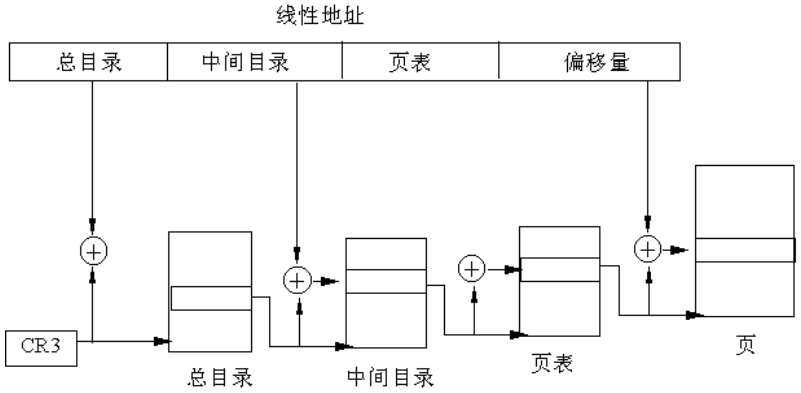

图 2.28　Linux 的三级分页

2.5.1　与页相关的数据结构及宏的定义

上一节讨论的分页机制是硬件对分页的支持，这是虚拟内存管理的硬件基础。要想使这种硬件机制充分发挥其功能，必须有相应软件的支持，我们来看一下 Linux 所定义的一些主要数据结构，其分布在 include/asm-i386/目录下的 page.h，pgtable.h 及 pgtable-2level.h 三个文件中。

1．表项的定义

如上所述，PGD、PMD 及 PT 表的表项都占 4 个字节，因此，把它们定义为无符号长整数，

分别叫做 pgd_t、pmd_t 及 pte_t（pte 即 Page table Entry），在 page.h 中定义如下：

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
```

可以看出，Linux 没有把这几个类型直接定义长整数而是定义为一个结构，这是为了让 gcc 在编译时进行更严格的类型检查。另外，还定义了几个宏来访问这些结构的成分，这也是一种面向对象思想的体现：

```
#define pte_val (x)   ( (x) .pte_low)
#define pmd_val (x)   ( (x) .pmd)
#define pgd_val (x)   ( (x) .pgd)
```

从图 2.22 和图 2.24 可以看出，对这些表项应该定义成位段，但内核并没有这样定义，而是定义了一个页面保护结构 pgprot_t 和一些宏：

```
typedef struct { unsigned long pgprot; } pgprot_t;
#define pgprot_val (x)     ( (x) .pgprot)
```

字段 pgprot 的值与图 2.24 页面项的低 12 位相对应，其中的 9 位对应 0～9 位，在 pgtalbe.h 中定义了对应的宏：

```
#define _PAGE_PRESENT   0x001
#define _PAGE_RW   0x002
#define _PAGE_USER 0x004
#define _PAGE_PWT  0x008
#define _PAGE_PCD  0x010
#define _PAGE_ACCESSED  0x020
#define _PAGE_DIRTY0x040
#define _PAGE_PSE  0x080   /* 4 MB  (or 2MB)  page, Pentium+, if present.. */
#define _PAGE_GLOBAL    0x100   /* Global TLB entry PPro+ */
```

在你阅读源代码的过程中你能体会到，把标志位定义为宏而不是位段更有利于编码。

另外，页目录表及页表在 pgtable.h 中定义如下：

```
extern pgd_t swapper_pg_dir[1024];
extern unsigned long pg0[1024];
```

swapper_pg_dir 为页目录表，pg0 为一临时页表，每个表最多都有 1024 项。

2．线性地址域的定义

Intel 线性地址的结构如图 2.29 所示。

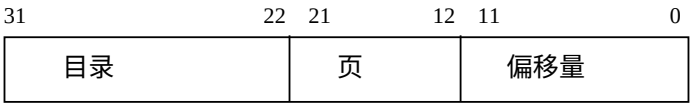

图 2.29　32 位的线性地址结构

偏移量的位数

```
#define PAGE_SHIFT 12
#define PAGE_SIZE   (1UL << PAGE_SHIFT)
#define PTRS_PER_PTE    1024
#define PAGE_MASK   (~ (PAGE_SIZE-1) )
```

其中 PAGE_SHIFT 宏定义了偏移量的位数为 12，因此页大小 PAGE_SIZE 为 2^{12}＝4096 字节；

PTRS_PER_PTE 为页表的项数；最后 PAGE_MASK 值定义为 0xfffff000，用以屏蔽掉偏移量域的所有位（12 位）。

PGDIR_SHIFT

```
#define PGDIR_SHIFT22
#define PTRS_PER_PGD    1024
#define PGDIR_SIZE  (1UL << PGDIR_SHIFT)
#define PGDIR_MASK  (~ (PGDIR_SIZE-1) )
```

PGDIR_SHIFT 是页表所能映射区域线性地址的位数，它的值为 22（12 位的偏移量加上 10 位的页表）；PTRS_PER_PGD 为页目录目录项数；PGDIR_SIZE 为页目录的大小，为 2^{22}，即 4MB；PGDIR_MASK 为 0xffc00000，用于屏蔽偏移量位与页表域的所有位。

(3) PMD_SHIFT

```
#define PMD_SHIFT  22
#define PTRS_PER_PMD    1
```

PMD_SHIFT 为中间目录表映射的地址位数，其值也为 22，但是因为 Linux 在 386 中只用了两级页表结构，因此，让其目录项个数为 1，这就使得中间目录在指针序列中的位置被保存，以便同样的代码在 32 位系统和 64 位系统下都能使用。后面的讨论我们不再提及中间目录。

2.5.2　对页目录及页表的处理

在 page.h，pgtable.h 及 pgtable-2level.h3 个文件中还定义有大量的宏，用以对页目录、页表及表项的处理，我们在此介绍一些主要的宏和函数。

1．表项值的确定

```
static inline int pgd_none (pgd_t pgd)  { return 0; }
static inline int pgd_present (pgd_t pgd)    { return 1; }
#define pte_present (x)    ( (x) .pte_low &  (_PAGE_PRESENT | _PAGE_PROTNONE) )
```

pgd_none () 函数直接返回 0，表示尚未为这个页目录建立映射，所以页目录项为空。pgd_present () 函数直接返回 1，表示映射虽然还没有建立，但页目录所映射的页表肯定存在于内存（即页表必须一直在内存）。

pte_present 宏的值为 1 或 0，表示 P 标志位。如果页表项不为 0，但标志位为 0，则表示映射已经建立，但所映射的物理页面不在内存。

2．清相应表的表项

```
#define pgd_clear (xp)   do { } while  (0)
#define pte_clear (xp)    do { set_pte (xp, __pte (0) ) ; } while  (0)
```

pgd_clear 宏实际上什么也不做，定义它可能是为了保持编程风格的一致。pte_clear 就是把 0 写到页表表项中。

3．对页表表项标志值进行操作的宏

这些宏的代码在 pgtable.h 文件中，表 2.1 给出宏名及其功能。

表 2.1　　对页表表项标志值进行操作的宏及其功能

宏名	功能
Set_pte ()	把一个具体的值写入表项
Pte_read ()	返回 User/Supervisor 标志值（由此可以得知是否可以在用户态下访问此页）
Pte _write ()	如果 Present 标志和 Read/Write 标志都为 1，则返回 1（此页是否存在并可写）
Pte _exec ()	返回 User/Supervisor 标志值
Pte _dirty ()	返回 Dirty 标志的值（说明此页是否被修改过）
Pte _young ()	返回 Accessed 标志的值（说明此页是否被存取过）
Pte _wrprotect ()	清除 Read/Write 标志
Pte _rdprotect ()	清除 User/Supervisor 标志
Pte _mkwrite	设置 Read/Write 标志
Pte _mkread	设置 User/Supervisor 标志
Pte _mkdirty ()	把 Dirty 标志置 1
Pte _mkclean ()	把 Dirty 标志置 0
Pte _mkyoung	把 Accessed 标志置 1
Pte _mkold ()	把 Accessed 标志置 0
Pte _modify (p,v)	把页表表项 p 的所有存取权限设置为指定的值 v
Mk_pte ()	把一个线性地址和一组存取权限合并来创建一个 32 位的页表表项
Pte _pte_phys ()	把一个物理地址与存取权限合并来创建一个页表表项
Pte _page ()	从页表表项返回页的线性地址

实际上页表的处理是一个复杂的过程，在这里我们仅仅让读者对软硬件如何结合起来有一个初步的认识，有关页表更多的内容我们将在第六章接着讨论。

2.6　Linux 中的汇编语言

在阅读 Linux 源代码时，你可能碰到一些汇编语言片段，有些汇编语言出现在以.S 为扩展名的汇编文件中，在这种文件中，整个程序全部由汇编语言组成。有些汇编命令出现在以.c 为扩展名的 C 文件中，在这种文件中，既有 C 语言，也有汇编语言，我们把出现在 C 代码中的汇编语言叫所“嵌入式”汇编。不管这些汇编代码出现在哪里，它在一定程度上都成为阅读源代码的拦路虎。

尽管 C 语言已经成为编写操作系统的主要语言，但是，在操作系统与硬件打交道的过程中，在需要频繁调用的函数中以及某些特殊的场合中，C 语言显得力不从心，这时，繁琐但又高效的汇编语言就粉墨登场了。因此，在了解一些硬件的基础上，必须对相关的汇编语言知识也有所了解。

读者可能有过在 DOS 操作系统下编写汇编程序的经历，也具备一定的汇编知识。但是，在 Linux 的源代码中，你可能看到了与 Intel 的汇编语言格式不一样的形式，这就是 AT&T 的 386 汇编语言。

2.6.1　AT&T 与 Intel 汇编语言的比较

我们知道，Linux 是 UNIX 家族的一员，尽管 Linux 的历史不长，但与其相关的很多事情都发源于 UNIX。就 Linux 所使用的 386 汇编语言而言，它也是起源于 UNIX。UNIX 最初是为 PDP-11 开发的，曾先后被移植到 VAX 及 68000 系列的处理器上，这些处理器上的汇编语言都采用的是 AT&T 的指令格式。当 UNIX 被移植到 i386 时，自然也就采用了 AT&T 的汇编语言格式，而不是 Intel 的格式。尽管这两种汇编语言在语法上有一定的差异，但所基于的硬件知识是相同的，因此，如果你非常熟悉 Intel 的语法格式，那么你也可以很容易地把它“移植”到 AT&T。下面我们通过对照 Intel 与 AT&T 的语法格式，以便于你把过去的知识能很快地“移植”过来。

1．前缀

在 Intel 的语法中，寄存器和和立即数都没有前缀。但是在 AT&T 中，寄存器前冠以“%”，而立即数前冠以“$”。在 Intel 的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在 AT&T 中，十六进制立即数前冠以“0x”，如表 2.2 所示给出几个相应的例子。

2．操作数的方向

Intel 与 AT&T 操作数的方向正好相反。在 Intel 语法中，第一个操作数是目的操作数，第二个操作数是源操作数。而在 AT&T 中，第一个数是源操作数，第二个数是目的操作数。由此可以看出，AT&T 的语法符合人们通常的阅读习惯。

例如：在 Intel 中，mov　　eax,[ecx]

在 AT&T 中，movl　(%ecx),%eax

表 2.2　　Intel 与 AT&T 前缀的区别

Intel 语法	AT&T 语法
mov　eax,8	movl $8,%eax
mov　ebx,0ffffh	movl $0xffff,%ebx
int　80h	int　$0x80

3．内存单元操作数

从上面的例子可以看出，内存操作数也有所不同。在 Intel 的语法中，基寄存器用“[]”括起来，而在 AT&T 中，用“()”括起来。

例如：在 Intel 中，mov　eax,[ebx+5]

在 AT&T，movl　5(%ebx),%eax

4．间接寻址方式

与 Intel 的语法比较，AT&T 间接寻址方式可能更晦涩难懂一些。Intel 的指令格式是 segreg:[base+index*scale+disp]，而 AT&T 的格式是%segreg:disp(base,index,scale)。

其中 index/scale/disp/segreg 全部是可选的，完全可以简化掉。如果没有指定 scale 而指定了 index，则 scale 的缺省值为 1。segreg 段寄存器依赖于指令以及应用程序是运行在实模式还是保护模式下，在实模式下，它依赖于指令，而在保护模式下，segreg 是多余的。在 AT&T 中，当立即数用在 scale/disp 中时，不应当在其前冠以“ $ ”前缀，表 2.3 给出其语法及几个相应的例子。

表 2.3　　内存操作数的语法及举例

Intel 语法	AT&T 语法
指　　令　　foo,segreg:[base+index*scale+disp]	指令 %segreg:disp（base,index,scale）,foo
mov　eax,[ebx+20h]	Movl　0x20（%ebx）,%eax
add　eax,[ebx+ecx*2h	Addl　（%ebx,%ecx,0x2）,%eax
lea　eax,[ebx+ecx]	Leal　（%ebx,%ecx）,%eax
sub　eax,[ebx+ecx*4h-20h]	Subl　-0x20（%ebx,%ecx,0x4）,%eax

从表中可以看出，AT&T 的语法比较晦涩难懂，因为[base+index*scale+disp]一眼就可以看出其含义，而 disp（base,index,scale）则不可能做到这点。

这种寻址方式常常用在访问数据结构数组中某个特定元素内的一个字段，其中，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素还是一个结构，则 disp 为具体字段在结构中的位移。

5．操作码的后缀

在上面的例子中你可能已注意到，在 AT&T 的操作码后面有一个后缀，其含义就是指出操作码的大小。“ l ”表示长整数（32 位），“ w ”表示字（16 位），“ b ”表示字节（8 位）。而在 Intel 的语法中，则要在内存单元操作数的前面加上 byte ptr、word ptr 和 dword ptr，“ dword ”对应“ long ”。表 2.4 给出了几个相应的例子。

表 2.4　　操作码的后缀举例

Intel 语法	AT&T 语法
Mov al,bl	movb　%bl,%al
Mov ax,bx	movw　%bx,%ax
Mov eax,ebx	movl　%ebx,%eax
Mov eax, dword ptr [ebx]	movl　（%ebx）,%eax

2.6.2　AT&T 汇编语言的相关知识

在 Linux 源代码中，以.S 为扩展名的文件是“ 纯 ”汇编语言的文件。这里，我们结合具

体的例子再介绍一些 AT&T 汇编语言的相关知识。

1．GNU 汇编程序 GAS（GNU Assembly）和连接程序

当你编写了一个程序后，就需要对其进行汇编（assembly）和连接。在 Linux 下有两种方式，一种是使用汇编程序 GAS 和连接程序 ld，一种是使用 gcc。我们先来看一下 GAS 和 ld:

GAS 把汇编语言源文件（.o）转换为目标文件（.o），其基本语法如下：

```
as filename.s -o filename.o
```

一旦创建了一个目标文件，就需要把它连接并执行，连接一个目标文件的基本语法为：

```
ld filename.o -o filename
```

这里 filename.o 是目标文件名，而 filename 是输出（可执行） 文件。

GAS 使用的是 AT&T 的语法而不是 Intel 的语法，这就再次说明了 AT&T 语法是 UNIX 世界的标准，你必须熟悉它。

如果要使用 GNC 的 C 编译器 gcc，就可以一步完成汇编和连接，例如：

```
gcc -o example example.S
```

这里，example.S 是你的汇编程序，输出文件（可执行文件）名为 example。其中，扩展名必须为大写的 S，这是因为，大写的 S 可以使 gcc 自动识别汇编程序中的 C 预处理命令，像#include、#define、#ifdef、#endif 等，也就是说，使用 gcc 进行编译，你可以在汇编程序中使用 C 的预处理命令。

2．AT&T 中的节（Section）

在 AT&T 的语法中，一个节由.section 关键词来标识，当你编写汇编语言程序时，至少需要有以下 3 种节。

section .data：这种节包含程序已初始化的数据，也就是说，包含具有初值的那些变量，例如：

```
    hello: .string "Hello world!\n"
    hello_len : .long 13
```

.section .bss：这个节包含程序还未初始化的数据，也就是说，包含没有初值的那些变量。当操作系统装入这个程序时将把这些变量都置为 0，例如：

```
name : .fill 30    # 用来请求用户输入名字
name_len : .long  0    # 名字的长度（尚未定义）
```

当这个程序被装入时，name 和 name_len 都被置为 0。如果你在.bss 节不小心给一个变量赋了初值，这个值也会丢失，并且变量的值仍为 0。

使用.bss 比使用.data 的优势在于，.bss 节不占用磁盘的空间。在磁盘上，一个长整数就足以存放.bss 节。当程序被装入到内存时，操作系统也只分配给这个节 4 个字节的内存大小。

注意，编译程序把.data 和.bss 在 4 字节上对齐（align），例如，.data 总共有 34 字节，那么编译程序把它对齐在 36 字节上，也就是说，实际给它 36 字节的空间。

section .text ：这个节包含程序的代码，它是只读节，而.data 和.bss 是读／写节。

3. 汇编程序指令（Assembler Directive）

上面介绍的.section 就是汇编程序指令的一种，GNU 汇编程序提供了很多这样的指令（directiv），这种指令都是以句点（.）为开头，后跟指令名（小写字母），在此，我们只介绍在内核源代码中出现的几个指令（以 arch/i386/kernel/head.S 中的代码为例）。

（1）.ascii "string"...

.ascii 表示零个或多个（用逗号隔开）字符串，并把每个字符串（结尾不自动加“0“ 字节）中的字符放在连续的地址单元。

还有一个与.ascii 类似的.asciz，z 代表“0”，即每个字符串结尾自动加一个“0”字节，例如：

```
int_msg:
    .asciz "Unknown interrupt\n"
```

（2）.byte 表达式

.byte 表示零或多个表达式（用逗号隔开），每个表达式被放在下一个字节单元。

（3）.fill 表达式

形式：.fill repeat , size , value

其中，repeat、size 和 value 都是常量表达式。Fill 的含义是反复拷贝 size 个字节。repeat 可以大于等于 0。size 也可以大于等于 0，但不能超过 8，如果超过 8，也只取 8。把 repeat 个字节以 8 个为一组，每组的最高 4 个字节内容为 0，最低 4 字节内容置为 value。

size 和 value 为可选项。如果第 2 个逗号和 value 值不存在，则假定 value 为 0。如果第 1 个逗号和 size 不存在，则假定 size 为 1。

例如，在 Linux 初始化的过程中，对全局描述符表 GDT 进行设置的最后一句为：

```
.fill NR_CPUS*4,8,0    /* space for TSS's and LDT's */
```

因为每个描述符正好占 8 个字节，因此，.fill 给每个 CPU 留有存放 4 个描述符的位置。

（4）.globl symbol

.globl 使得连接程序（ld）能够看到 symbl。如果你的局部程序中定义了 symbl，那么，与这个局部程序连接的其他局部程序也能存取 symbl，例如：

```
.globl SYMBOL_NAME (idt)
.globl SYMBOL_NAME (gdt)
```

定义 idt 和 gdt 为全局符号。

（5）.quad bignums

.quad 表示零个或多个 bignums（用逗号分隔），对于每个 bignum，其缺省值是 8 字节整数。如果 bignum 超过 8 字节，则打印一个警告信息；并只取 bignum 最低 8 字节。

例如，对全局描述符表的填充就用到这个指令：

```
.quad 0x00cf9a000000ffff   /* 0x10 kernel 4GB code at 0x00000000 */
.quad 0x00cf92000000ffff   /* 0x18 kernel 4GB data at 0x00000000 */
.quad 0x00cffa000000ffff   /* 0x23 user   4GB code at 0x00000000 */
.quad 0x00cff2000000ffff   /* 0x2b user   4GB data at 0x00000000 */
```

（6）.rept count

把.rept 指令与.endr 指令之间的行重复 count 次，例如

```
   .rept   3
   .long   0
```

```
    .endr
```

相当于

```
    .long   0
    .long   0
    .long   0
```

(7) .space size , fill

这个指令保留 size 个字节的空间，每个字节的值为 fill。size 和 fill 都是常量表达式。如果逗号和 fill 被省略，则假定 fill 为 0，例如在 arch/i386/bootl/setup.S 中有一句：

```
.space  1024
```

表示保留 1024 字节的空间，并且每个字节的值为 0。

(8) .word expressions

这个表达式表示任意一节中的一个或多个表达式（用逗号分开），表达式的值占两个字节，例如：

```
gdt_descr:
.word GDT_ENTRIES*8-1
```

表示变量 gdt_descr 的值为 GDT_ENTRIES*8-1

(9) .long expressions

这与.word 类似

(10) .org new-lc , fill

把当前节的位置计数器提前到 new-lc（New Location Counter)。new-lc 或者是一个常量表达式，或者是一个与当前子节处于同一节的表达式。也就是说，你不能用.org 横跨节：如果 new-lc 是个错误的值，则.org 被忽略。.org 只能增加位置计数器的值，或者让其保持不变；但绝不能用.org 来让位置计数器倒退。

注意，位置计数器的起始值是相对于一个节的开始的，而不是子节的开始。当位置计数器被提升后，中间位置的字节被填充值 fill（这也是一个常量表达式)。如果逗号和 fill 都省略，则 fill 的缺省值为 0。

例如：.org 0x2000

ENTRY (pg0)

表示把位置计数器置为 0x2000，这个位置存放的就是临时页表 pg0。

2.6.3　gcc 嵌入式汇编

在 Linux 的源代码中，有很多 C 语言的函数中嵌入一段汇编语言程序段，这就是 gcc 提供的“ asm ”功能，例如在 include/asm-i386/system.h 中定义的，读控制寄存器 CR0 的一个宏 read_cr0 ()：

```
#define read_cr0 ()   ({ \
    unsigned int __dummy; \
    __asm__ ( \
  "movl %%cr0,%0\n\t" \
  :"=r"  (__dummy) ) ; \
    __dummy; \
```

})

这种形式看起来比较陌生，这是因为这不是标准 C 所定义的形式，而是 gcc 对 C 语言的扩充。其中__dummy 为 C 函数所定义的变量；关键词__asm__表示汇编代码的开始。括弧中第一个引号中为汇编指令 movl，紧接着有一个冒号，这种形式阅读起来比较复杂。

一般而言，嵌入式汇编语言片段比单纯的汇编语言代码要复杂得多，因为这里存在怎样分配和使用寄存器，以及把 C 代码中的变量应该存放在哪个寄存器中。为了达到这个目的，就必须对一般的 C 语言进行扩充，增加对编译器的指导作用，因此，嵌入式汇编看起来晦涩而难以读懂。

1．嵌入式汇编的一般形式

```
__asm__ __volatile__ ("<asm routine>" : output : input : modify) ;
```

其中，__asm__表示汇编代码的开始，其后可以跟__volatile__（这是可选项），其含义是避免“ asm ”指令被删除、移动或组合；然后就是小括弧，括弧中的内容是我们介绍的重点。

- "<asm routine>"为汇编指令部分，例如，"movl %%cr0,%0\n\t"。数字前加前缀“ % “ ，如%1，%2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数量，如 Intel 可以有 8 个。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的处理。由于这些样板操作数的前缀使用了“ % ”，因此，在用到具体的寄存器时就在前面加两个“ % ”，如%%cr0。
- 输出部分（output），用以规定对输出变量（目标操作数）如何与寄存器结合的约束（constraint）,输出部分可以有多个约束，互相以逗号分开。每个约束以“ = ”开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：

:"=r" (__dummy)

“ =r ”表示相应的目标操作数（指令部分的%0）可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

：“ =m ” (__dummy)

“ =m ”就表示相应的目标操作数是存放在内存单元__dummy 中。

表示约束条件的字母很多，表 2.5 给出了几个主要的约束字母及其含义。

表 2.5　　主要的约束字母及其含义

字母	含义
m, v,o	表示内存单元
R	表示任何通用寄存器
Q	表示寄存器 eax、ebx、ecx、edx 之一
I, h	表示直接操作数
E, F	表示浮点数
G	表示“ 任意 ”
a, b.c d	表示要求使用寄存器 eax/ax/al，ebx/bx/bl，ecx/cx/cl 或 edx/dx/dl

S，D	表示要求使用寄存器 esi 或 edi
I	表示常数（0～31）

输入部分（Input）：输入部分与输出部分相似，但没有“ ＝ ”。如果输入部分一个操作数所要求使用的寄存器，与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号（如“ 1 ”,“ 2 ”等）放在约束条件中，在后面的例子中，我们会看到这种情况。

修改部分（modify）:这部分常常以“ memory ”为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。

注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，分号“ :”要保留，当“ memory ”存在时，三个分号都要保留，例如 system.h 中的宏定义__cli ()：

```
#define __cli ()  __asm__ __volatile__ ("cli": : :"memory")
```

2．Linux 源代码中嵌入式汇编举例

Linux 源代码中，在 arch 目录下的.h 和.c 文件中，很多文件都涉及嵌入式汇编，下面以 system.h 中的 C 函数为例，说明嵌入式汇编的应用。

（1）简单应用

```
#define __save_flags (x)     __asm__ __volatile__ ("pushfl ; popl %0":"=g" (x) : /* no input
*/)
#define __restore_flags (x)  __asm__ __volatile__ ("pushl %0 ; popfl": /* no output */
 :"g" (x) :"memory", "cc")
```

第 1 个宏是保存标志寄存器的值，第 2 个宏是恢复标志寄存器的值。第 1 个宏中的 pushfl 指令是把标志寄存器的值压栈。而 popl 是把栈顶的值（刚压入栈的 flags）弹出到 x 变量中，这个变量可以存放在一个寄存器或内存中。这样，你可以很容易地读懂第 2 个宏。

（2）较复杂应用

```
static inline unsigned long get_limit (unsigned long segment)
{
    unsigned long __limit;
    __asm__ ("lsll %1,%0"
  :"=r" (__limit) :"r" (segment) ) ;
   return __limit+1;
}
```

这是一个设置段界限的函数，汇编代码段中的输出参数为__limit（即%0），输入参数为 segment（即%1）。lsll 是加载段界限的指令，即把 segment 段描述符中的段界限字段装入某个寄存器（这个寄存器与__limit 结合），函数返回__limit 加 1，即段长。

（3）复杂应用

在 Linux 内核代码中，有关字符串操作的函数都是通过嵌入式汇编完成的，因为内核及用户程序对字符串函数的调用非常频繁，因此，用汇编代码实现主要是为了提高效率（当然是以牺牲可读性和可维护性为代价的）。在此，我们仅列举一个字符串比较函数 strcmp，其代码在 arch/i386／string.h 中。

```
static inline int strcmp (const char * cs,const char * ct)
{
int d0, d1;
register int __res;
__asm__ __volatile__ (
    "1:\tlodsb\n\t"
    "scasb\n\t"
    "jne 2f\n\t"
    "testb %%al,%%al\n\t"
    "jne 1b\n\t"
    "xorl %%eax,%%eax\n\t"
    "jmp 3f\n"
    "2:\tsbbl %%eax,%%eax\n\t"
    "orb $1,%%al\n"
    "3:"
    :"=a" (__res) , "=&S" (d0) , "=&D" (d1)
  :"1" (cs) ,"2" (ct) ) ;
return __res;
}
```

其中的“ \n ”是换行符，“ \t ”是 tab 符，在每条命令的结束加这两个符号，是为了让 gcc 把嵌入式汇编代码翻译成一般的汇编代码时能够保证换行和留有一定的空格。例如，上面的嵌入式汇编会被翻译成：

```
1：   lodsb   //装入串操作数，即从[esi]传送到 al 寄存器，然后 esi 指向串中下一个元素
 scasb//扫描串操作数，即从 al 中减去 es:[edi]，不保留结果，只改变标志
 jne2f//如果两个字符不相等，则转到标号 2
 testb %al  %al
 jne 1b
 xorl %eax %eax
 jmp 3f
2:    sbbl %eax %eax
 orb $1 %al
3:
```

这段代码看起来非常熟悉，读起来也不困难。其中 3f 表示往前（forword）找到第一个标号为 3 的那一行，相应地，1b 表示往后找。其中嵌入式汇编代码中输出和输入部分的结合情况为：

- 返回值__res，放在 al 寄存器中，与%0 相结合；
- 局部变量 d0，与%1 相结合，也与输入部分的 cs 参数相对应，也存放在寄存器 ESI 中，即 ESI 中存放源字符串的起始地址。
- 局部变量 d1，与%2 相结合，也与输入部分的 ct 参数相对应，也存放在寄存器 EDI 中，即 EDI 中存放目的字符串的起始地址。

通过对这段代码的分析我们应当体会到，万变不利其本，嵌入式汇编与一般汇编的区别仅仅是形式，本质依然不变。因此，全面掌握 Intel 386 汇编指令乃突破阅读低层代码之根本。

2.6.4　Intel386 汇编指令摘要

在阅读 Linux 源代码时，你可能遇到很多汇编指令，有些是你熟悉的，有些可能不熟悉，在此简要列出一些常用的 386 汇编指令及其功能。

1．位操作指令

指令	功能
BT	位测试
BTC	位测试并求反
BTR	位测试并复位
BTS	位测试并置位

2．控制转移类指令

指令	功能
CALL	调用过程
JMP	跳转
LOOP	用 ECX 计数器的循环
LOOPNZ/LOOPNE	用 ECX 计数器且不为 0 的循环／用 ECX 计数器且不等的循环
RET	返回

3．数据传输指令

指令	功能
IN	从端口输入
LEA	装入有效地址
MOV	传送
OUT	从段口输出
POP	从堆栈中弹出
POPA／POPAD	从栈弹出至所有寄存器
PUSH	压栈
PUSH／PUSHAD	所有通用寄存器压栈
XCHG	交换

4．标志控制类指令

指令	功能
CLC	清 0 进位标志
CLD	清 0 方向标志
CLI	清 0 中断标志
LAHF	将标志寄存器装入 AH 寄存器
POPF／POPFD	从栈中弹出至标志位

PUSHF／PUSHFD	将标志压栈
SAHF	将 AH 寄存器存入标志寄存器
STC	置进位标志
STD	置方向标志
STI	置中断标志

5．逻辑类指令

指令	功能
NOT	与
AND	非
OR	或
SAL/SHL	算术左移／逻辑左移
SAR	算术右移
SHLD	逻辑右移
TEST	逻辑比较
XOR	异或

6．串操作指令

指令	功能
CMPS/CMPSB/CMPSW/CMPSD	比较串操作数
INS/INSB/INSW/INSD	输入串操作数
LODS/LODSB/LODSW/LODSD	装入串操作数
MOVS/MOVSB/MOVSW/MOVSD	传送串操作数
REP	重复
REPE/REPZ	相等时重复／为 0 时重复
SCAS/SCASB/SCASW/SCASD	扫描串操作数
STOS/STOSB/STOSW/STOSD	存储串操作数

7．多段类操作指令

指令	功能
CALL	过程调用
INT	中断过程的调用
INTO	溢出中断过程的调用
IRET	中断返回
JMP	跳转
LDS	将指针转入 DS
LES	将指针转入 ES
LFS	将指针转入 FS
LGS	将指针转入 GS

LSS	将指针转入 SS
MOV	段寄存器的传送
POP	从栈弹出至段寄存器
PUSH	压栈
RET	返回

8．操作系统类指令

指令	功能
APPL	调整请求特权级
ALTS	清任务切换标志
HLT	暂停
LAR	加载访问权
LGDT	加载全局描述符表
LIDT	加载中断描述符表
LLDT	加载局部描述符表
LMSW	加载机器状态字
LSL	加载段界限
LTR	加载任务寄存器
MOV	特殊寄存器的数据传送
SGDT	存储全局描述符表
SIDT	存储中断描述符表
SMSW	存储机器状态字
STR	存储任务寄存器