址24，那里是一条MOV指令。第二个程序一开始跳转到地址28，那里是一条CMP指令。与讨论无关的指令没有画出来。当两个程序被连续地装载到内存中从0开始的地址时，内存中的状态就如同图3-2c所示。在这个例子里，我们假设操作系统是在高地址处，图中没有画出来。

程序装载完毕之后就可以运行了。由于它们的内存键不同，它们不会破坏对方的内存。但在另一方面会发生问题。当第一个程序开始运行时，它执行了JMP 24指令，然后不出预料地跳转到了相应的指令，这个程序会正常运行。

但是，当第一个程序已经运行了一段时间后，操作系统可能会决定开始运行第二个程序，即装载在第一个程序之上的地址16 384处的程序。这个程序的第一条指令是JMP 28，这条指令会使程序跳转到第一个程序的ADD指令，而不是事先设定的跳转到CMP指令。由于对内存地址的不正确访问，这个程序很可能在1秒之内就崩溃了。

这里关键的问题是这两个程序都引用了绝对物理地址，而这正是我们最需要避免的。我们希望每个程序都使用一套私有的本地地址来进行内存寻址。下面我们会展示这种技术是如何实现的。IBM 360对上述问题的补救方案就是在第二个程序装载到内存的时候，使用静态重定位的技术修改它。它的工作方式如下：当一个程序被装载到地址16 384时，常数16 384被加到每一个程序地址上。虽然这个机制在不出错误的情况下是可行的，但这不是一种通用的解决办法，同时会减慢装载速度。而且，它要求给所有的可执行程序提供额外的信息来区分哪些内存字中存有（可重定位的）地址，哪些没有。毕竟，图3-2b中的“28”需要被重定位，但是像

```
MOV REGISTER1, 28
```

这样把数28送到REGISTER1的指令不可以被重定位。装载器需要一定的方法来辨别地址和常数。

最后，正如我们在第1章中指出的，计算机世界的发展总是倾向于重复历史。虽然直接引用物理地址对于大型计算机、小型计算机、台式计算机和笔记本电脑来说已经成为很久远的记忆了（对此我们深表遗憾），但是缺少内存抽象的情况在嵌入式系统和智能卡系统中还是很常见的。现在，像收音机、洗衣机和微波炉这样的设备都已经完全被（ROM形式的）软件控制，在这些情况下，软件都采用访问绝对内存地址的寻址方式。在这些设备中这样能够正常工作是因为，所有运行的程序都是可以事先确定的，用户不可能在烤面包机上自由地运行他们自己的软件。

虽然高端的嵌入式系统（比如手机）有复杂的操作系统，但是一般的简单嵌入式系统并非如此。在某些情况下可以用一种简单的操作系统，它只是一个被链接到应用程序的库，该库为程序提供I/O和其他任务所需要的系统调用。操作系统作为库实现的常见例子如流行的e-cos操作系统。

3.2 一种存储器抽象：地址空间

总之，把物理地址暴露给进程会带来下面几个严重问题。第一，如果用户程序可以寻址内存的每个字节，它们就可以很容易地（故意地或偶然地）破坏操作系统，从而使系统慢慢地停止运行（除非有特殊的硬件进行保护，如IBM 360的锁键模式）。即使在只有一个用户进程运行的情况下，这个问题也是存在的。第二，使用这种模型，想要同时（如果只有一个CPU就轮流执行）运行多个程序是很困难的。在个人计算机上，同时打开几个程序是很常见的（一个文字处理器，一个邮件程序，一个网络浏览器，其中一个当前正在工作，其余的在按下鼠标的时候才会被激活）。在系统中没有对物理内存的抽象的情况下，很难做到上述情景，因此，我们需要其他办法。

3.2.1 地址空间的概念

要保证多个应用程序同时处于内存中并且不互相影响，则需要解决两个问题：保护和重定位。我们来看一个原始的对前者的解决办法，它曾被用在IBM 360上：给内存块标记上一个保护键，并且比较执行进程的键和其访问的每个内存字的保护键。然而，这种方法本身并没有解决后一个问题，虽然这个问题可以通过在程序被装载时重定位程序来解决，但这是一个缓慢且复杂的解决方法。

一个更好的办法是创造一个新的内存抽象：地址空间。就像进程的概念创造了一类抽象的CPU以运行程序一样，地址空间为程序创造了一种抽象的内存。地址空间是一个进程可用于寻址内存的一套地址集合。每个进程都有一个自己的地址空间，并且这个地址空间独立于其他进程的地址空间（除了在一些特殊情况下进程需要共享它们的地址空间外）。