这些进程使用相同的I空间页表和不同的D空间页表。在一个比较典型的使用这种方式来支持共享的实现中，页表与进程表数据结构无关。每个进程在它的进程表中都有两个指针：一个指向I空间页表，一个指向D空间页表，如图3-26所示。当调度程序选择一个进程运行时，它使用这些指针来定位合适的页表，并使用它们来设立MMU。即使没有分离的I空间和D空间，进程也可以共享程序（或者有时为库），但要使用更为复杂的机制。

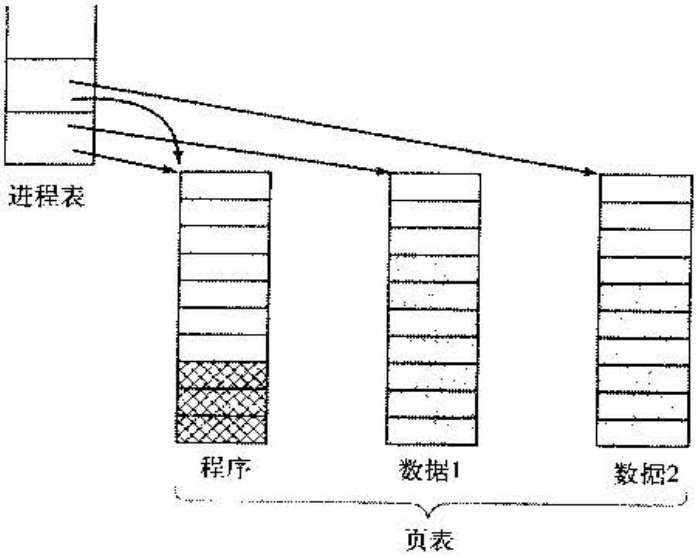

图3-26 两个进程通过共享程序页表来共享同一个程序

在两个或更多进程共享某些代码时，在共享页面上存在一个问题。假设进程A和进程B同时运行一个编辑器并共享页面。如果调度程序决定从内存中移走A，撤销其所有的页面并用一个其他程序来填充这些空的页框，则会引起B产生大量的缺页中断，才能把这些页面重新调入。

类似地，当进程A结束时，能够发现这些页面仍然在被使用是非常必要的，这样，这些页面的磁盘空间才不会被随意释放。查找所有的页表，考察一个页面是否共享，其代价通常比较大，所以需要专门的数据结构记录共享页面，特别地，如果共享的单元是单个页面（或一批页面），而不是整个页表。

共享数据要比共享代码麻烦，但也不是不可能。特别是在UNIX中，在进行fork系统调用后，父进程和子进程要共享程序文本和数据。在分页系统中，通常是让这些进程分别拥有它们自己的页表，但都指向同一个页面集合。这样在执行fork调用时就不需要进行页面复制。然而，所有映射到两个进程的数据页面都是只读的。

只要这两个进程都仅仅是读数据，而不做更改，这种情况就可以保持下去。但只要有一个进程更新了一点数据，就会触发只读保护，并引发操作系统陷阱。然后会生成一个该页的副本，这样每个进程都有自己的专用副本。两个复制都是可以读写的，随后对任何一个副本的写操作都不会再引发陷阱。这种策略意味着那些从来不会执行写操作的页面（包括所有程序页面）是不需要复制的，只有实际修改的数据页面需要复制。这种方法称为写时复制，它通过减少复制而提高了性能。

3.5.6 共享库

可以使用其他的粒度取代单个页面来实现共享。如果一个程序被启动两次，大多数操作系统会自动共享所有的代码页面，而在内存中只保留一份代码页面的副本。代码页面总是只读的，因此这样做不存在任何问题。依赖于不同的操作系统，每个进程都拥有一份数据页面的私有副本，或者这些数据页面被共享并且被标记为只读。如果任何一个进程对一个数据页面进行修改，系统就会为此进程复制这个数据页面的一个副本，并且这个副本是此进程私有的，也就是说会执行“写时复制”。

现代操作系统中，有很多大型库被众多进程使用，例如，处理浏览文件以便打开文件的对话框的库和多个图形库。把所有的这些库静态地与磁盘上的每一个可执行程序绑定在一起，将会使它们变得更加庞大。

一个更加通用的技术是使用共享库（在Windows中称作DLL或动态链接库）。为了清楚地表达共享库的思想，首先考虑一下传统的链接。当链接一个程序时，要在链接器的命令中指定一个或多个目标文件，可能还包括一些库文件。以下面的UNIX命令为例：

```
ld *.o -lc -lm
```

这个命令会链接当前目录下的所有的.o（目标）文件，并扫描两个库：/usr/lib/libc.a和/usr/lib/libm.a。任何在目标文件中被调用了但是没有被定义的函数（比如，printf），都被称作未定义外部函数（undefined externals）。链接器会在库中寻找这些未定义外部函数。如果找到了，则将它们加载到可执行二进制文件中。任何被这些未定义外部函数调用了但是不存在的函数也会成为未定义外部函数。例如，printf需要