write，如果write还没有被加载进来，链接器就会查找write并在找到后把它加载进来。当链接器完成任务后，一个可执行二进制文件被写到磁盘，其中包括了所需的全部函数。在库中定义但是没有被调用的函数则不会被加载进去。当程序被装入内存执行时，它需要的所有函数都已经准备就绪了。

假设普通程序需要消耗20~50MB用于图形和用户界面函数。静态链接上百个包括这些库的程序会浪费大量的磁盘空间，在装载这些程序时也会浪费大量的内存空间，因为系统不知道它可以共享这些库。这就是引入共享库的原因。当一个程序和共享库（与静态库有些许区别）链接时，链接器没有加载被调用的函数，而是加载了一小段能够在运行时绑定被调用函数的存根例程（stub routine）。依赖于系统和配置信息，共享库或者和程序一起被装载，或者在其所包含函数第一次被调用时被装载。当然，如果其他程序已经装载了某个共享库，就没有必要再次装载它了——这正是关键所在。值得注意的是，当一个共享库被装载和使用时，整个库并不是被一次性地读入内存。而是根据需要，以页面为单位装载的，因此没有被调用到的函数是不会被装载到内存中的。

除了可以使可执行文件更小、节省内存空间之外，共享库还有一个优点：如果共享库中的一个函数因为修正一个bug被更新了，那么并不需要重新编译调用了这个函数的程序。旧的二进制文件依然可以正常工作。这个特性对于商业软件来说尤为重要，因为商业软件的源码不会分发给客户。例如，如果微软发现并修复了某个标准DLL中的安全错误，Windows更新会下载新的DLL来替换原有文件，所有使用这个DLL的程序在下次启动时会自动使用这个新版本的DLL。

不过，共享库带来了一个必须解决的小问题，如图3-27所示。我们看到有两个进程共享一个20KB大小的库（假设每一方框为4KB）。但是，这个库被不同的进程定位在不同的地址上，大概是因为程序本身的大小不相同。在进程1中，库从地址36K开始；在进程2中则从地址12K开始。假设库中第一个函数要做的第一件事就是跳转到库的地址16。如果这个库没有被共享，它可以在装载的过程中重定位，就会跳转（在进程1中）到虚拟地址的36K+16。注意，库被装载到的物理地址与这个库是否为共享库是没有任何关系的，因为所有的页面都被MMU硬件从虚拟地址映射到了物理地址。

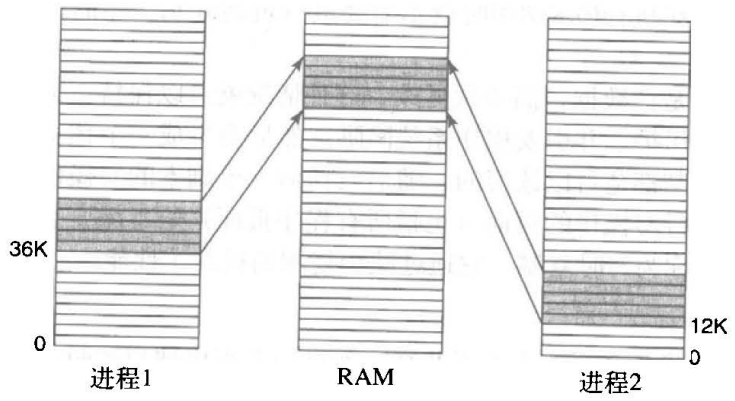

图3-27 两个进程使用的共享库

但是，由于库是共享的，因此在装载时再进行重定位就行不通了。毕竟，当进程2调用第一个函数时（在地址12K），跳转指令需要跳转到地址12K+16，而不是地址36K+16。这就是那个必须解决的小问题。解决它的一个办法是写时复制，并为每一个共享这个库的进程创建新页面，在创建新页面的过程中进行重定位。当然，这样做和使用共享库的目的相悖。

一个更好的解决方法是：在编译共享库时，用一个特殊的编译选项告知编译器，不要产生使用绝对地址的指令。相反，只能产生使用相对地址的指令。例如，几乎总是使用向前（或向后）跳转*n*个字节（与给出具体跳转地址的指令不同）的指令 。不论共享库被放置在虚拟地址空间的什么位置，这种指令都可以正确工作。通过避免使用绝对地址，这个问题就可以被解决。只使用相对偏移量的代码被称作*位置无关代码*（position-independent code）。

3.5.7 内存映射文件

共享库实际上是一种更为通用的机制——*内存映射文件*（memory-mapped file）的一个特例。这种