注意

请参阅设计20查看设备上运行的进程。

10.6 线程

默认情况下，程序按顺序执行指令，一次处理一个任务。但是，如果程序需要并行执行两个或更多任务呢？例如，假设程序需要在更新用户界面的同时执行一些长时间运行的计算——可能是为了显示一个进度条。如果程序是完全按顺序执行的，那么当程序开始计算时，用户界面就会被忽略，因为分配给程序的CPU时间必须消耗在其他地方。期望的行为是在计算进行时更新UI——这是需要并行执行的两个独立任务。操作系统通过执行线程（简称"线程"）来提供这种功能。线程是进程中可调度的执行单元。线程在进程中运行，它可以执行任何被加载到该进程的程序代码。

线程运行的代码通常包含程序希望完成的特定任务。由于线程属于进程，因此它们与该进程中所有其他线程一起共享地址空间、代码和其他资源。进程从一个线程开始，当有工作需要并行处理时，可以根据需要创建其他线程。每个线程都有一个被称为线程ID或TID的标识符。内核也创建线程来管理其工作。图10-7展示了线程、进程和内核之间的关系。

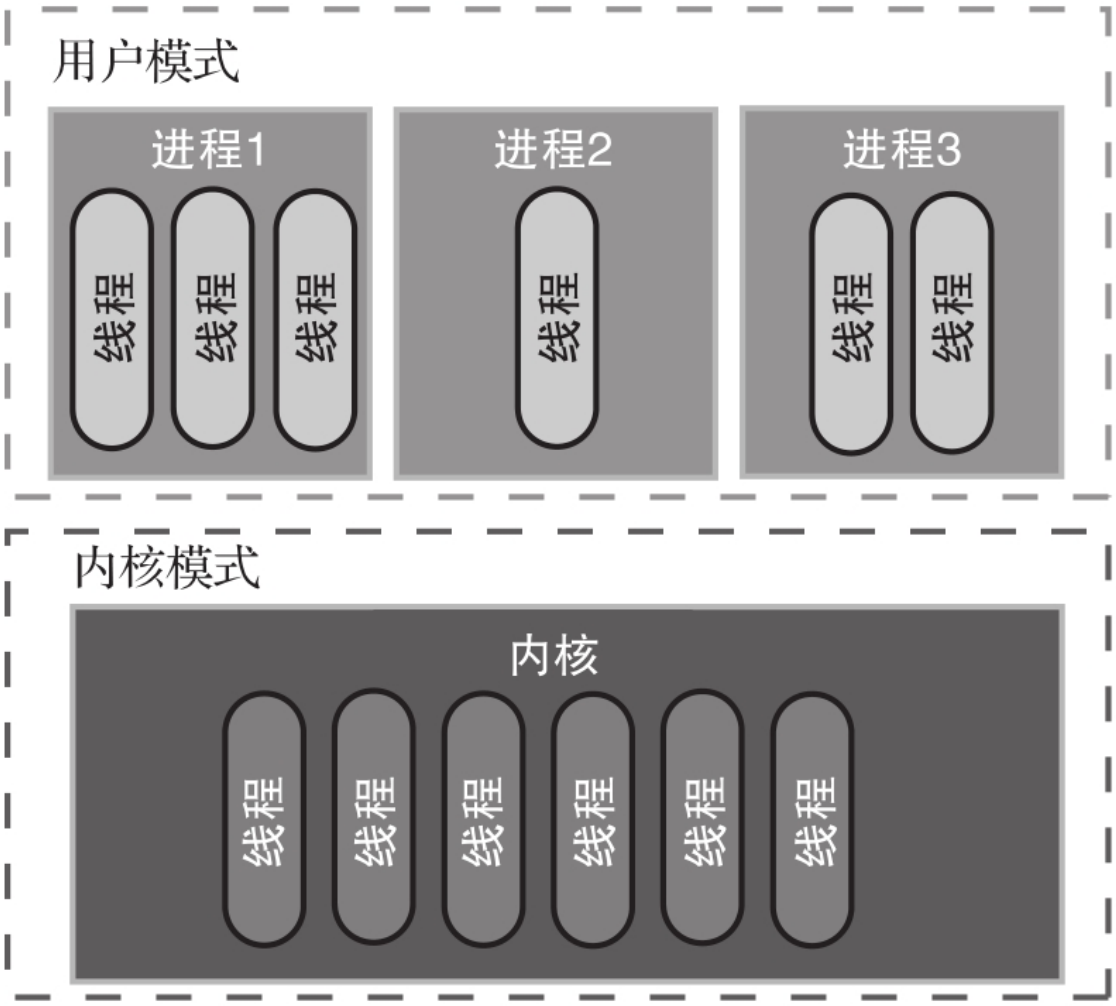


图10-7 属于用户模式进程的线程和属于内核的线程

在Windows中，线程和进程是不同的对象类型。进程对象是一个容器，线程属于进程。在Linux中，这种区别更加微妙。Linux内核使用一个数据类型来表示进程和线程，这个数据类型同时充当进程和线程。在Linux中，共享一个地址空间并有共同进程标识符的一组线程被视为一个进程，没有单独的进程类型。

用来指示进程和线程标识符的Linux术语可能会令人困惑。在用户模式下，进程有进程ID（PID），线程有线程ID（TID），就和Windows一样。但是，Linux内核把线程ID作为PID，而把进程ID作为线程组标识符（TGID）！

注意

请参阅设计21创建自己的线程。

并行运行多个线程的真正含义是什么？假设计算机正在运行10个进程，每个进程有4个线程。那么仅在用户模式下就有40个线程！我们说线程并行运行，但40个线程真的可以同时运行吗？不能，除非计算机有40个处理器核，但它很可能没有。每个处理器核一次只能运行一个线程，所以设备中核的数量决定了能同时运行的线程数。

物理核和逻辑核

不是所有的核都具有同等的并行能力。物理核是CPU内部核的硬件实现。逻辑核表示单个物理核一次运行多个线程（每个逻辑核一个线程）的能力。Intel把这个功能称为超线程。例如，我用来写这本书的计算机有两个物理核，每个都有两个逻辑核，所以总共有四个逻辑核。这就意味着我的计算机可以同时运行4个线程，尽管逻辑核无法实现物理核完全的并行性。

如果我们有40个线程需要运行，但只有4个核，那么会发生什么呢？操作系统会实现一个调度器，即负责确保每个线程轮流运行的软件组件。不同的操作系统用不同的方法实现调度，但基本目标都是相同的：给线程运行的时间。一个线程获得一小段时间来运行（称为量子），然后该线程挂起，让另一个线程运行。稍后，再次调度第一个线程，从其中断的地方开始继续执行。通常，这对线程代码和编写应用程序的开发人员是隐藏的。从线程代码的角度来看，它是连续运行的，开发人员编写多线程应用程序时，就好像所有线程都在并行运行一样。

10.7 虚存

操作系统支持多个正在运行的进程，其中的每一个进程都需要使用内存。大多数情况下，一个进程不需要读写另一个进程的内存，实际上，这通常是不可取的。我们不希望行为不端的进程窃取数据或覆盖其他进程的

数据，或者更糟的是，覆盖内核数据。此外，开发人员不希望由于其他进程的使用使得自己进程的地址空间碎片化。出于这些原因，操作系统不会授权用户模式的进程去访问物理内存，相反，为每个进程提供的是虚存（virtual memory）——一种为每个进程提供自己的大容量且私有的地址空间的抽象。

我们在第7章讨论了内存寻址：硬件中的每个物理字节都被分配了一个地址。这种硬件内存地址被称为物理地址。这些地址通常对用户模式进程是隐藏的。操作系统为进程提供了虚存，其中的每个地址都是虚拟地址。每个进程都有自己的虚存空间。对于单个进程，内存看上去就是一个大的地址范围。当进程向某个虚拟地址写入数据时，该地址不会直接引用硬件内存位置。这个虚拟地址在需要时被转换成物理地址，如图10-8所示，但转换的过程对进程是隐藏的。

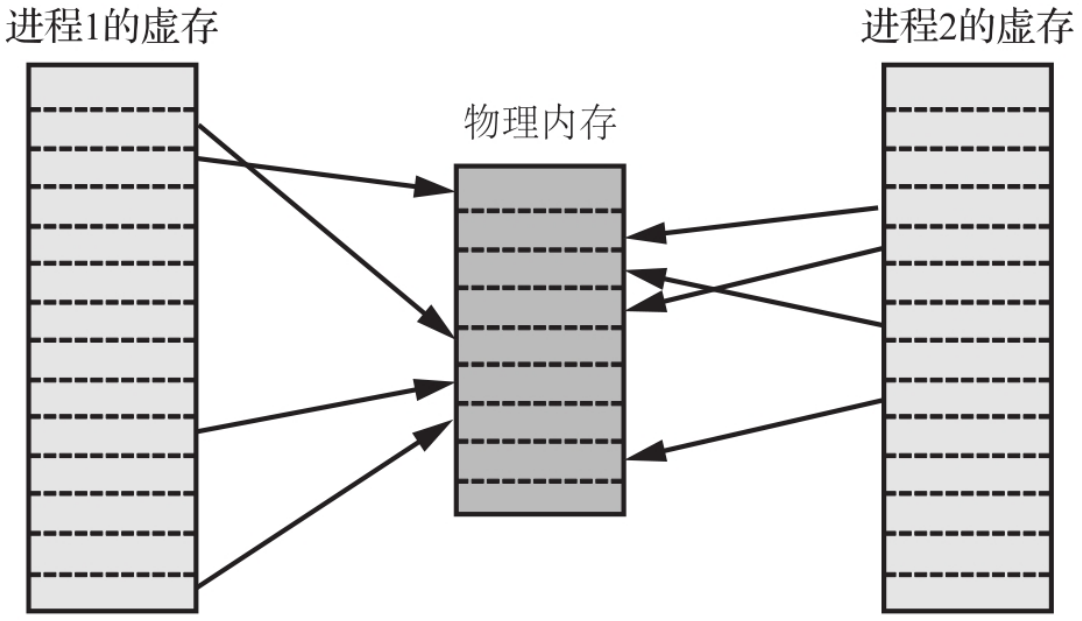


图10-8 每个进程的虚拟地址空间映射到物理内存

这种方法的优点是，给每个进程都提供了一个大的、私有的虚拟地址范围。通常，系统上的每个进程都有相同的内存地址范围。例如，每个进程可能被授予2GB的虚拟地址空间，其地址从0x00000000到0x7FFFFFFF。这看上去似乎有问题：当两个程序试图使用相同内存地址时，会发生什

么？一个程序会覆盖或读取另一个程序的数据吗？幸亏有虚拟寻址，这不是问题。

多个程序的相同虚拟地址映射到不同的物理地址，所以一个程序不会意外访问到内存中另一个程序的数据。这意味着，对于不同的进程，存储在某个虚拟地址中的数据是不同的——虚拟地址可能相同，但存储在其中的数据是不同的。也就是说，如果程序需要共享内存，就会有相应的机制。在旧的操作系统中，内存空间没有进行如此清晰的划分，导致程序有很多机会破坏其他程序甚至操作系统中的内存。幸运的是，所有的现代操作系统都确保了进程间的内存分离。

重要的是要理解尽管进程的地址空间可能是2GB大小，但这并不意味着全部2GB的虚存都可以立即供该进程使用。这些地址中只有一部分是由物理内存支持的。回忆一下你在第8章和第9章完成的设计，那些你正在查看的地址是虚拟内存地址，而不是物理地址。

内核有独立的虚拟地址空间，其地址范围与分配给用户模式进程的地址范围不同。与用户模式地址空间不同，内核地址空间由所有运行于内核模式的代码共享。这意味着任何在内核模式下运行的代码都可以访问内核地址空间中的所有内容。这也让这些代码有机会修改内核内存的内容。这强化了在内核模式下运行的代码必须是可信的这一思想！

那么如何在用户模式和内核模式之间划分虚拟地址空间呢？让我们看一个32位的操作系统。正如第7章所讨论的，对于32位系统，内存地址表示为一个32位数，这意味着地址空间总共有4GB。这个地址空间中的地址必须在内核模式和用户模式之间进行划分。对于4GB的地址空间，Windows和Linux都允许根据配置划分为2GB用户模式虚拟地址/2GB内核模式虚拟地址，或者3GB用户模式虚拟地址/1GB内核模式虚拟地址。图10-9显示了对虚存平均2GB的划分。

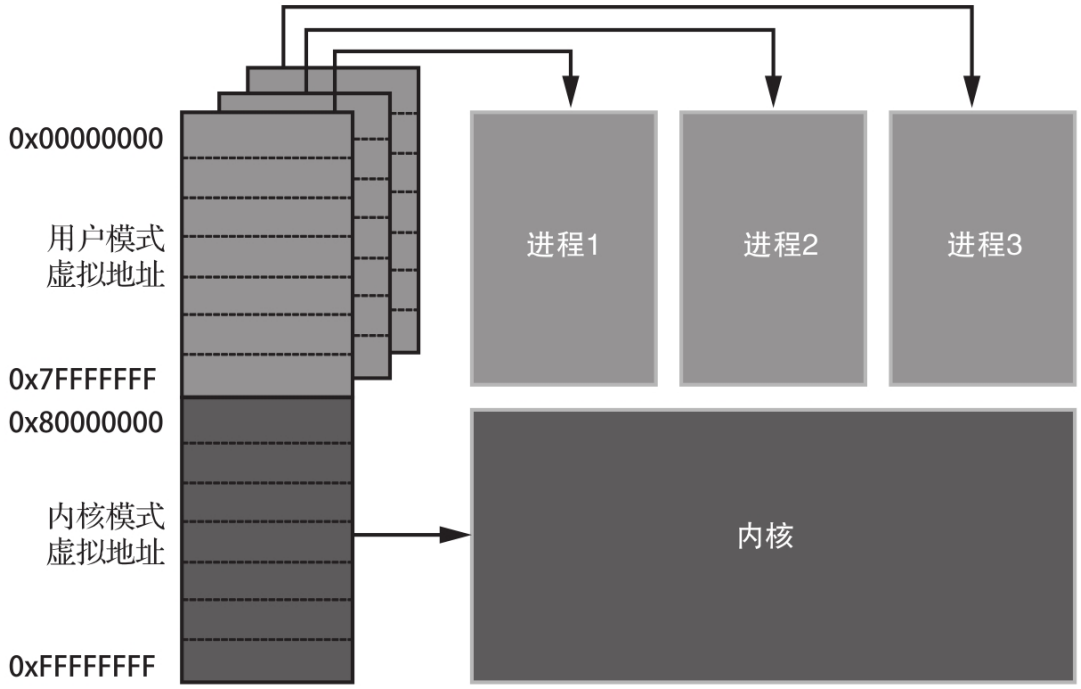


图10-9 32位系统虚拟地址空间的平均划分

请注意，这里我们只关注虚拟地址。不论其有多少物理内存，32位系统都有4GB的虚拟地址空间。假设一台计算机只有1GB RAM，在32位操作系统下，它还是有4GB的虚拟地址空间。回忆一下，虚拟地址范围不代表映射的物理内存，只表示可以映射的物理内存的范围。也就是说，内核与所有的运行中进程所请求的虚存当然有可能大于RAM的总大小。在这种情况下，操作系统可以把一些内存字节内容移动到辅存，以便为新请求的内存腾出RAM空间，这个过程称为分页。通常，最少被使用的内存首先被分页，这样频繁使用的内存可以保留在RAM中。当需要被分页的内存时，操作系统必须把它加载回RAM。分页允许使用更大的虚存，但代价是在辅存和RAM之间移动字节时产生的性能损失。请记住，辅存比RAM慢得多。

注意

请参阅设计22查看虚存。

随着64位处理器和操作系统的到来，有可能出现更大的地址空间。如果我们用完整的64位来表示内存地址，那么虚拟地址空间将是32位地址空

间的40亿倍！但是，现在不需要这么大的地址空间，所以64位操作系统使用较少的位数来表示地址。不同处理器上不同的64位操作系统表示地址的位数不同。64位Linux和64位Windows都支持48位地址，这相当于256TB的虚拟地址空间，大约是32位地址空间的65000倍——对于现在的典型应用程序而言，这个空间已经足够了。

10.8 应用程序编程接口

当大多数人想到操作系统时，他们会想到用户界面，即壳。壳是人们所看到的，它会影响人们如何看待这个系统。例如，Windows用户一般把Windows看作任务栏、“开始”菜单、桌面等。但用户界面实际上只是操作系统代码的一小部分，它只是个界面，是系统与用户相遇的地方。从应用程序（或软件开发人员）的角度来看，与操作系统的交互不由UI定义，而是由操作系统的应用程序编程接口（Application Programming Interface，API）来定义。API不仅适用于操作系统，任何想以编程方式进行交互的软件都可以提供API，但这里我们重点关注操作系统API。

操作系统API是一种规范，它在源代码中定义，在文档中描述，它详细说明了程序应如何与操作系统交互。典型的操作系统API包括与操作系统交互所需的函数（含它们的名称、输入和输出）以及数据结构列表。操作系统中包含的软件库提供了API规范的实现。软件开发人员所说的“调用”或“使用”API是一种简洁地表示他们的代码正在调用API中指定（并在软件库中实现）的函数的方式。

就像UI为用户定义操作系统的“个性”一样，API也为应用程序定义操作系统的个性。图10-10展示了用户和应用程序是如何与操作系统交互的。

如图10-10所示，用户与操作系统的用户界面（也称为壳）进行交互。壳把用户命令转换成API调用。然后，API再调用内部操作系统代码来执行请求的操作。应用程序无须通过UI，它们只需直接调用API即可。从这个角度来说，壳与操作系统API的交互就像任何其他应用程序一样。

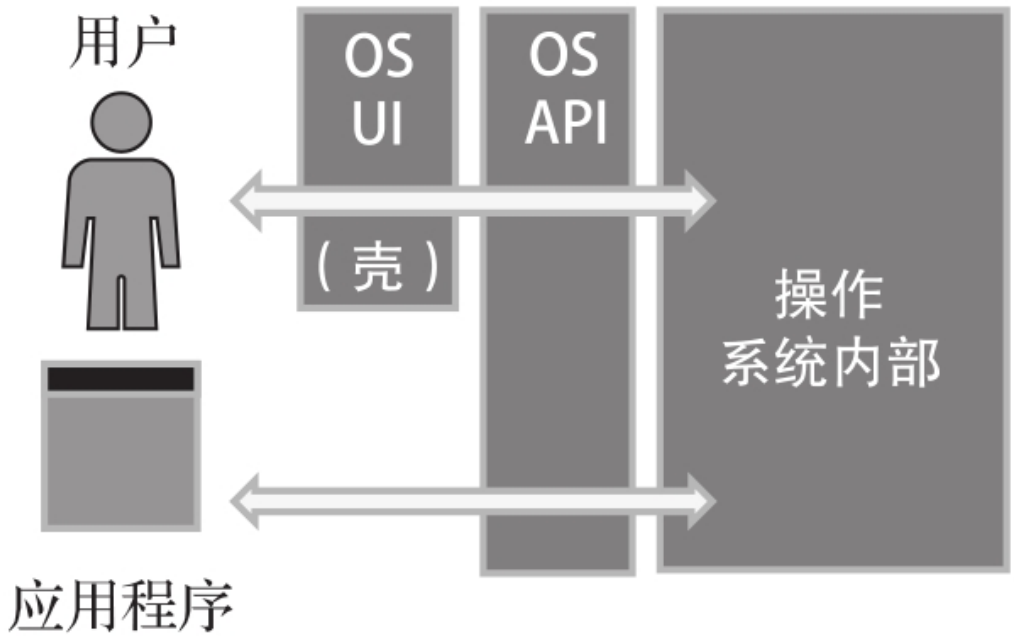


图10-10 操作系统界面：UI对用户，API对应用程序

让我们来看一个通过API与操作系统交互的例子。创建文件是操作系统的常见功能，是用户和应用程序都需要的。图形壳和命令行壳为用户提供了创建文件的简单方式。但是，应用程序不需要通过GUI或CLI来创建文件。让我们看看应用程序如何以编程方式来创建文件。

对于UNIX或Linux系统，你可以用API函数open来创建文件。下面的C语言例子使用open函数创建了名为hello.txt的新文件。O_WRONLY标志指示只写操作，O_CREAT指示要创建一个文件：

```
open("hello.txt", O_WRONLY|O_CREAT);
```

同样的操作在Windows中可以用CreateFileA API函数来实现：

```
CreateFileA("hello.txt", GENERIC_WRITE, 0, NULL,
    CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
```

这两个例子都使用C语言。操作系统通常是用C语言编写的，所以它们的API自然就适合在C程序中使用。对于用其他语言编写的程序，在程序运

行时仍然必须调用操作系统API，但编程语言把该API调用封装在自己的语法中，对开发人员隐藏了API的细节。这使得代码可以在不同的操作系统间移植。甚至C语言也是这样的，它提供了能在任何操作系统上使用的标准函数库。反过来，这些函数在运行时必须调用特定于操作系统的API。再考虑一下创建文件的例子，在C语言中，我们可以换用fopen函数，如下面的代码所示，这个函数是C语言标准库的一部分，可以在任何操作系统上运行：

```
fopen("hello.txt", "w");
```

作为另一个例子，我们可以用如下Python代码来创建新文件。这行代码能在任何安装了Python解释器的操作系统上运行。Python解释器负责代表应用程序来调用合适的操作系统API。

```
open('hello.txt', 'w')
```

对于类UNIX操作系统，API根据UNIX或Linux的特定风格以及内核的版本会有所不同。但是，大多数类UNIX操作系统完全或部分符合标准规范。这个标准被称为可移植操作系统接口（Portable Operating System Interface，POSIX），它不仅为操作系统API提供了标准，而且为壳的行为及其包含的实用程序提供了标准。POSIX为类UNIX操作系统提供了一个基线，但现代类UNIX操作系统一般都有自己的API。Cocoa是Apple macOS的API，iOS也有类似的API，称为Cocoa Touch。Android也有自己的一组编程接口，统称为Android平台API。

另一个主要的操作系统系列Windows有自己的API。Windows API一直在增长和扩展。Windows API的最初版本是16位的，现在称为Win16。当Windows在20世纪90年代升级为32位操作系统时，32位版本的API，即Win32也发布了。现在，Windows是64位操作系统，也有相应的Win64 API。微软还在Windows 10中引入了一个新的API，即通用Windows平台（Universal Windows Platform，UWP），其目的是让应用程序的开发在运行Windows的各类设备上保持一致。

注意

请参阅设计23尝试与Linux操作系统API进行交互。

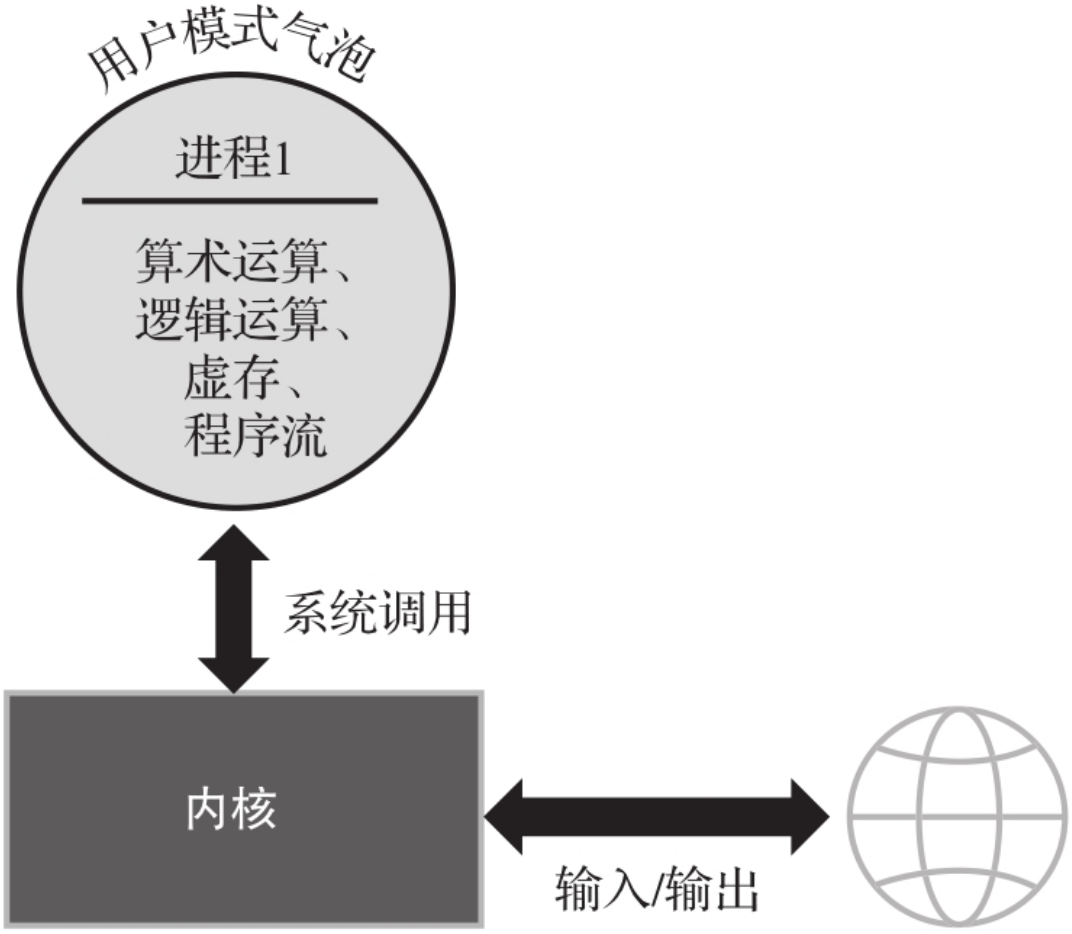
10.9 用户模式气泡和系统调用

如前所述，在用户模式下运行的代码访问系统是受限的。那么，用户模式代码能做些什么呢？它可以读写自己的虚存，执行算术和逻辑运算，还可以控制自己代码的程序流。但用户模式代码不能访问物理内存地址，包括用于内存映射I/O的地址。这意味着它无法独自将文本输出到控制台窗口、从键盘接收输入、把图形绘制到屏幕、播放声音、接收触摸屏输入、通过网络进行通信，或从硬盘读取文件！我喜欢说"用户模式代码运行在气泡中"（见图10-11）。它不能与外界进行交互，至少在没有帮助的情况下不能。另一种说法是，用户模式代码不能直接执行I/O。这样做的实际效果是，在用户模式下运行的代码可以执行有用的工作，但它不能在没有帮助的情况下分享这项工作的成果。

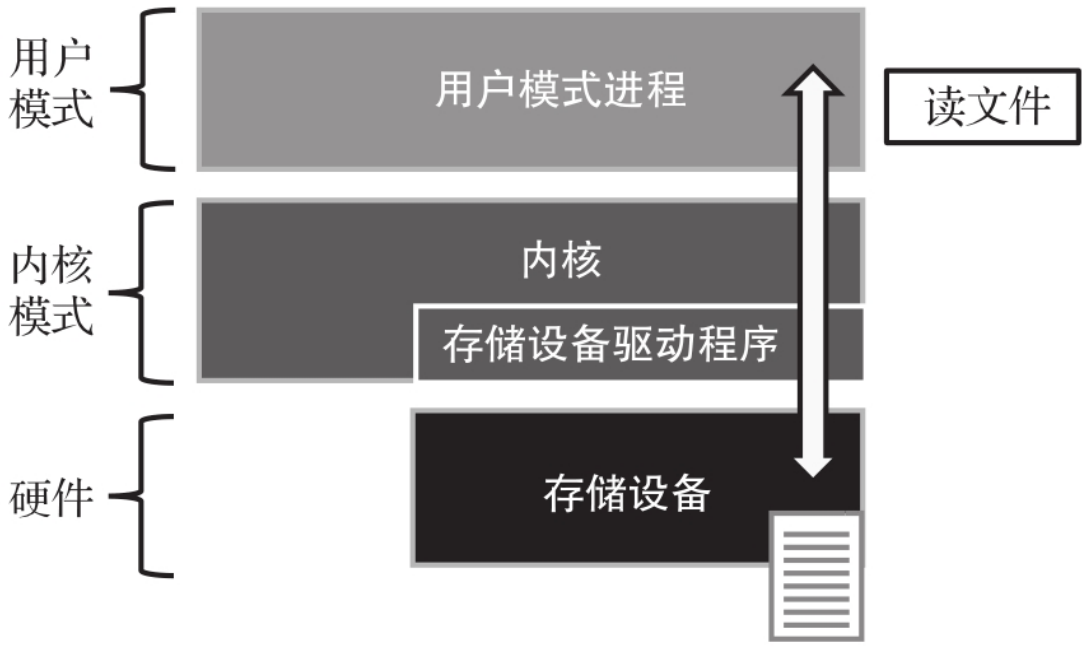
图10-11 进程在用户模式气泡中运行。它可以执行算术运算、逻辑运算，可以访问虚存，还可以控制程序流，但它不能直接与外界进行交互

你可能好奇用户模式应用程序是如何与用户交互的。当然，应用程序能够以某种方式与外界进行交互，但这是如何实现的呢？答案是用户模式代码有另一个重要功能：它可以请求内核模式代码代表它执行工作。当用户模式代码请求内核模式代码代表它执行特权操作时，这个过程被称为系统调用，如图10-12所示。

例如，如果用户模式代码需要读取文件，它会通过系统调用来请求内核从某个文件中读取某些字节。内核与存储设备驱动程序配合使用，执行必要的I/O来读取文件，然后把被请求数据提供给用户模式进程，如图10-13所示。



▲图10-12 在内核的帮助下，通过系统调用，用户模式进程可以与外界交互



▲图10-13 内核充当用户模式代码的中介，这些代码需要访问硬件资源，比如辅存

用户模式代码不需要了解关于物理存储设备或相关设备驱动程序的任何信息。内核提供了抽象，把细节封装起来，并允许用户模式代码简单地完成工作。我们之前介绍的示例API函数open和CreateFileA在后台的工作方式是这样的：通过系统调用请求特权操作。当然，内核允许与否是有限制的。例如，用户模式进程不能读取其无权访问的文件。

CPU提供专用指令以方便系统调用。在ARM处理器上，使用SVC指令（以前是SWI），它被称为管理程序调用。在x86处理器上，可使用SYSCALL和SYSENTER指令。Linux和Windows都实现了大量的系统调用，每个调用都用唯一的编号进行标识。例如，在ARM的Linux上，write系统调用（写入文件）的编号是4。要进行系统调用，程序需要把所需的系统调用的编号加载到某个处理器寄存器，并把任何附加参数放入其他特定寄存器中，然后再执行系统调用指令。

尽管软件开发人员可以直接用机器码或汇编语言进行系统调用，但幸运的是，在大多数情况下不需要这样做。操作系统和高级编程语言为程序员提供了用自然方式进行系统调用的方法，通常是通过操作系统API或编程语言的标准库。程序员只需简单地编写代码便可执行一个操作，甚至都可能没有意识到后台正在执行一个系统调用。

注意

请参阅设计24观察程序进行的系统调用。

10.10 API和系统调用

前面我们讨论了操作系统API，并且只讨论了系统调用。那么，操作系统API和系统调用有什么不同呢？这两者是相关的，但它们不等价。系统调用为用户模式代码请求内核模式服务定义了一种机制。API描述了应用程序与操作系统交互的一种方式，无论是否调用了内核模式代码。一些API函数使用了系统调用，而其他API函数不需要系统调用，具体情况取决于操作系统。

让我们先来看看Linux。如果我们把Linux的定义限制为内核，我们可以说Linux API是使用Linux系统调用的有效规范，因为系统调用是内核的编程接口。然而，基于Linux的操作系统不仅仅只是内核。以Android为例，它使用的就是Linux内核。Android有它自己的一套编程接口，即Android平台API。

对于Microsoft Windows，Windows NT内核通过被称为本地API（Native API）的接口提供了一组系统调用。应用程序开发人员很少直接使用本地API，它是供操作系统组件使用的。相反，开发人员使用Windows API，它充当本地API的包装器。但是，并不是所有的Windows API函数都需要系统调用。让我们看几个Windows API的例子。Windows API函数CreateFileW创建或打开一个文件。它是本地API NtCreateFile的包装器，NtCreateFile对内核进行系统调用。相比之下，Windows API函数PathFindFileNameW（在路径中查找文件名）不与本地

API交互或进行任何系统调用。创建文件需要内核的帮助，而在路径字符串中查找文件名则只需要访问虚存，这可以在用户模式下进行。

总结一下，操作系统API描述了操作系统的编程接口。系统调用为用户模式代码提供了一种请求特权内核模式操作的机制。某些API函数依赖于系统调用，而另一些则不依赖。

10.11 操作系统软件库

如前所述，操作系统API描述了到操作系统的编程接口。尽管技术接口描述对于程序员而言是有用的，但当程序运行时，它还是需要一个具体方法来调用API。这是通过软件库实现的。操作系统的软件库是操作系统中的代码集合，它提供了操作系统API的实现。也就是说，库所包含的代码执行的是在API规范中描述的操作。在第9章中，我们讨论了编程语言可用的库：该语言的标准库以及使用该语言的开发人员社区维护的其他库。这里讨论的软件库与之类似，唯一的区别是这些库是操作系统的一部分。

操作系统库类似于可执行程序，它是包含机器码字节的文件。不过，它一般没有入口点，所以通常不能自行运行。相反，库导出（提供）一组可以被程序使用的函数。使用软件库的程序从库中导入函数，这被称为链接到该库。

操作系统包含了一组库文件，它们导出API定义的各种函数。一些函数只是立即进行内核系统调用的包装器。一些函数是在库文件自身所含的用户模式代码中完全实现的。其他的则介于两者之间，在用户模式代码中实现一些逻辑，同时也进行一个或多个系统调用，如图10-14所示。