首要的近似，内核所做的全部工作只是调度进程以及处理在进程之间传递的消息。因此，只需要两个系统调用：send发送一条消息，而receive接收一条消息。第三个调用sendrec只是为了效率的原因而做的优化，它使得仅用一次内核陷阱就可以发送一条消息并且请求应答。其他的一切事情都是通过请求某些其他进程（例如文件系统进程或磁盘驱动程序）做相应的工作而完成的。

Amoeba甚至更加简单。它仅有一个系统调用：执行远程过程调用。该调用发送一条消息并且等待一个应答。它在本质上与MINIX的sendrec相同。其他的一切都建立在这一调用的基础上。

原则3：效率

第三个指导方针是实现的效率。如果一个功能特性或者系统调用不能够有效地实现，或许就不值得包含它。对于程序员来说，一个系统调用的代价有多大也应该在直觉上是显而易见的。例如，UNIX程序员会认为lseek系统调用比read系统调用要代价低廉，因为前者只是在内存中修改一个指针，而后者则要执行磁盘I/O。如果直觉的代价是错误的，程序员就会写出效率差的程序。

13.2.2 范型

一旦确定了目标，就可以开始设计了。一个良好的起点是考虑客户将怎样审视该系统。最为重要的问题之一是如何将系统的所有功能特性良好地结合在一起，并且展现出经常所说的*体系结构一致性*（architectural coherence）。在这方面，重要的是区分两种类型的操作系统“客户”。一方面，是用户，他们与应用程序打交道；另一方面，是程序员，他们编写应用程序。前者主要涉及GUI，后者主要涉及系统调用接口。如果打算拥有遍及整个系统的单一GUI，就像在Macintosh中那样，设计应该在此处开始。然而，如果打算支持许多可能的GUI，就像在UNIX中那样，那么就应该首先设计系统调用接口。首先设计GUI本质上是自顶向下的设计。这时的问题是GUI要拥有什么功能特性，用户将怎样与它打交道，以及为了支持它应该怎样设计系统。例如，如果大多数程序在屏幕上显示图标然后等待用户在其上点击，这暗示着GUI应该采用事件驱动模型，并且操作系统或许也应该采用事件驱动模型。另一方面，如果屏幕主要被文本窗口占据，那么进程从键盘读取输入的模型可能会更好。

首先设计系统调用接口是自底向上的设计。此时的问题是程序员通常需要哪些种类的功能特性。实际上，并不是需要许多特别的功能特性才能支持一个GUI。例如，UNIX窗口系统X只是一个读写键盘、鼠标和屏幕的大的C程序。X是在UNIX问世很久以后才开发的，但是并不要求对操作系统做很多修改就可以使它工作。这一经历验证了这样的事实：UNIX是十分完备的。

1. 用户界面范型

对于GUI级的接口和系统调用接口而言，最重要的方面是有一个良好的范型（有时称为隐喻），以提供观察接口的方法。台式计算机的许多GUI使用我们在第5章讨论过的WIMP范型。该范型在遍及接口的各处使用定点-点击、定点-双击、拖动以及其他术语，以提供总体上的体系结构一致性。对于应用程序常常还有额外的要求，例如要有一个具有文件（FILE）、编辑（EDIT）以及其他条目的菜单栏，每个条目具有某些众所周知的菜单项。这样，熟悉一个程序的用户就能够很快地学会另一个程序。

然而，WIMP用户界面并不是惟一可能的用户界面。某些掌上型计算机使用一种程式化的手写界面。专用的多媒体设备可能使用像VCR一样的界面。当然，语音输入具有完全不同的范型。重要的不是选择这么多的范型，而是存在一个单一的统领一切的范型统一整个用户界面。

不管选择什么范型，重要的是所有应用程序都要使用它。因此，系统设计者需要提供库和工具包给应用程序开发人员，使他们能够访问产生一致的外观与感觉的过程。用户界面设计非常重要，但它并不是本书的主题，所以我们现在要退回到操作系统接口的主题上。

2. 执行范型

体系结构一致性不但在用户层面是重要的，在系统调用接口层面也同样重要。在这里区分执行范型和数据范型常常是有益的，所以我们将讨论两者，我们以前者为开始。

两种执行范型被广泛接受：算法范型和事件驱动范型。*算法范型*（algorithmic paradigm）基于这样的思想：启动一个程序是为了执行某个功能，而该功能是事先知道的或者是从其参数获知的。该功能可能是编译一个程序、编制工资册，或者是将一架飞机飞到旧金山。基本逻辑被硬接线到代码当中，而程序则时常发出系统调用获取用户输入、获得操作系统服务等。图13-1a中概括了这一方法。