```
fd3 = open("/proc/501", O_RDWR);
```

允许进程（尝试）访问进程501的内存，使用文件描述符fd3进行读和写，这在某种程度上是有益的，例如对于一个调试器。

Windows Vista更进一步，它试图使所有一切看起来像是一个对象。一旦一个进程获得了一个指向文件、进程、信号量、邮箱或者其他内核对象的有效句柄，它就可以在其上执行操作。这一范型甚至比UNIX更加一般化，并且比FORTRAN要一般化得多。

统一的范型还出现在其他上下文中，其中在这里值得一提的是Web。Web背后的范型是充满了文档的超空间，每一个文档具有一个URL。通过键入一个URL或者点击被URL所支持的条目，你就可以得到该文档。实际上，许多“文档”根本就不是文档，而是当请求到来时由程序或者命令行解释器脚本生成的。例如，当用户询问一家网上商店关于一位特定艺术家的CD清单时，文档由一个程序即时生成；在查询未做出之前该文档的确并不存在。

至此我们已经看到了4种事例，即所有一切都是磁带、文件、对象或者文档。在所有这4种事例中，意图是统一数据、设备和其他资源，从而使它们更加易于处理。每一个操作系统都应该具有这样的统一数据范型。

13.2.3 系统调用接口

如果一个人相信Corbató的机制最少化的格言，那么操作系统应该提供恰好够用的系统调用，并且每个系统调用都应该尽可能简单（但不能过于简单）。统一的数据范型在此处可以扮演重要的角色。例如，如果文件、进程、I/O设备以及更多的东西都可以看作是文件或者对象，那么它们就都能够用单一的read系统调用来读取。否则，可能就有必要具有read_file、read_proc以及read_tty等单独的系统调用。

在某些情况下，系统调用可能看起来需要若干变体，但是通常更好的实现是具有处理一般情况的一个系统调用，而由不同的库过程向程序员隐藏这一事实。例如，UNIX具有一个系统调用exec，用来覆盖一个进程的虚拟地址空间。最一般的调用是：

```
exec(name, argp, envp);
```

该调用加载可执行文件name，并且给它提供由argp所指向的参数和envp所指向的环境变量。有时明确地列出参数是十分方便的，所以库中包含如下调用的过程：

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

所有这些过程所做的事情是将参数粘连在一个数组中，然后调用exec来做工作。这一安排达到了双赢目的：单一的直接系统调用使操作系统保持简单，而程序员得到了以各种方法调用exec的便利。

当然，试图拥有一个调用来处理每一种可能的情况很可能难以控制。在UNIX中，创建一个进程需要两个调用：fork然后是exec，前者不需要参数，后者具有3个参数。相反，创建一个进程的Win32 API调用CreateProcess具有10个参数，其中一个参数是指向一个结构的指针，该结构具有另外18个参数。

很久以前，有人曾经问过这样的问题：“如果我们省略了这些东西会不会发生可怕的事情？”诚实的回答应该是：“在某些情况下程序员可能不得不做更多的工作以达到特定的效果，但是最终的结果将会是一个更简单、更小巧并且更可靠的操作系统。”当然，主张10+18个参数版本的人可能会说：“但是用户喜欢所有这些特性。”对此的反驳可能会是：“他们更加喜欢使用很少内存并且从来不会崩溃的系统。”在更多功能性和更多内存代价之间的权衡是显而易见的，并且可以从价格上来衡量（因为内存的价格是已知的）。然而，每年由于某些特性而增加的崩溃次数是难于估算的，并且如果用户知道了隐藏的代价是否还会做出同样的选择呢？这一影响可以在Tanenbaum软件第一定律中做出总结：

添加更多的代码就是添加更多的程序错误。

添加更多的功能特性就要添加更多的代码，因此就要添加更多的程序错误。相信添加新的功能特性而不会添加新的程序错误的程序员要么是计算机的生手，要么就是相信牙齿仙女（据说会在儿童掉落在枕边的幼齿旁放上钱财的仙女）正在那里监视着他们。

简单不是设计系统调用时出现的惟一问题。一个重要的考虑因素是Lampson（1984）的口号：