访问。如果硬件允许，可以让每个驱动程序进程仅访问它需要的那些I/O设备。例如，对于内存映射的I/O，每个驱动程序进程可以拥有页面将它的设备映射进来，但是没有其他设备的页面。如果I/O端口空间可以部分地加以保护，就可以保证只有相应的正确部分对每个驱动程序可用。

即使没有硬件帮助可用，仍然可以设法使这一思想可行。此时需要的是一个新的系统调用，该系统调用仅对设备驱动程序进程可用，它提供一个（端口，取值）对列表。内核所做的是首先进行检查以了解进程是否拥有列表中的所有端口，如果是，它就将相应的取值复制到端口以发起设备I/O。类似的调用可以用一种受保护的方式读I/O端口。

这一方法使设备驱动程序避免了检查（并且破坏）内核数据结构，这（在很大程度上）是一件好事情。一组类似的调用可以用来让驱动程序进程读和写内核表格，但是仅以一种受控的方式并且需要内核的批准。

这一方法的主要问题，并且一般而言是针对微内核的主要问题，是额外的上下文切换导致性能受到影响。然而，微内核上的所有工作实际上是许多年前当CPU还非常缓慢的时候做的。如今，用尽CPU的处理能力并且不能容忍微小性能损失的应用程序是十分稀少的。毕竟，当运行一个字处理器或Web浏览器时，CPU可能有95%的时间是空闲的。如果一个基于微内核的操作系统将一个不可靠的3GHz的系统转变为一个可靠的2.5GHz的系统，可能很少有用户会抱怨。毕竟，仅仅在几年以前当他们得到具有1GHz的速度（就当时而言十分惊人）的系统时，大多数用户是相当快乐的。

4. 可扩展的系统

对于上面讨论的客户-服务器系统，思想是让尽可能多的东西脱离内核。相反的方法是将更多的模块放到内核中，但是以一种“受保护的”方式。当然，这里的关键字是“受保护的”。我们在9.5.6节中研究了某些保护机制，这些机制最初打算用于通过Internet引入小程序，但是对于将外来的代码插入到内核中的过程同样适用。最重要的是沙盒技术和代码签名，因为解释对于内核代码来说实际上是不可行的。

当然，可扩展的系统自身并不是构造一个操作系统的方法。然而，通过以一个只是包含保护机制的最小系统为开端，然后每次将受保护的模块添加到内核中，直到达到期望的功能，对于手边的应用而言一个最小的系统就建立起来了。按照这一观点，对于每一个应用，通过仅仅包含它所需要的部分，就可以裁剪出一个新的操作系统。Paramecium就是这类系统的一个实例（Van Doorn, 2001）。

5. 内核线程

此处，另一个相关的问题是系统线程，无论选择哪种结构模型。有时允许存在与任何用户进程相隔离的内核线程是很方便的。这些线程可以在后台运行，将脏页面写入磁盘，在内存和磁盘之间交换进程，如此等等。实际上，内核本身可以完全由这样的线程构成，所以当一个用户发出系统调用时，用户的线程并不是在内核模式中运行，而是阻塞并且将控制传给一个内核线程，该内核线程接管控制以完成工作。

除了在后台运行的内核线程以外，大多数操作系统还要启动许多守护进程。虽然这些守护进程不是操作系统的组成部分，但是它们通常执行“系统”类型的活动。这些活动包括接收和发送电子邮件，并且对远程用户各种各样的请求进行服务，例如FTP和Web网页。

13.3.2 机制与策略

另一个有助于体系结构一致性的原理是机制与策略的分离，该原理同时还有助于使系统保持小型和良好的结构。通过将机制放入操作系统而将策略留给用户进程，即使存在改变策略的需要，系统本身也可以保持不变。即使策略模块必须保留在内核中，如果可能，它也应该与机制相隔离，这样策略模块中的变化就不会影响机制模块。

为了使策略与机制之间的划分更加清晰，让我们考虑两个现实世界的例子。第一个例子，考虑一家大型公司，该公司拥有负责向员工发放薪水的工资部门。该部门拥有计算机、软件、空白支票、与银行的契约以及更多的机制，以便准确地发出薪水。然而，策略——确定谁将获得多少薪水——是完全与机制分开的，并且是由管理部门决定的。工资部门只是做他们被吩咐做的事情。

第二个例子，考虑一家饭店。它拥有提供餐饮的机制，包括餐桌、餐具、服务员、充满设备的厨房、与信用卡公司的契约，如此等等。策略是由厨师长设定的，也就是说，厨师长决定菜单上有什么。如果厨师长决定撤掉豆腐换上牛排，那么这一新的策略可以由现有的机制来处理。