围的）以及许多其他内核表格也是有效的。

一个替代的策略是将进程表建立为一个小型表的链表，最初只有一个表。如果该表被填满，可以从全局存储池中分配另一个表并且将其链接到前一个表。这样，在全部内核内存被耗尽之前，进程表不可能被填满。

另一方面，搜索表格的代码会变得更加复杂。例如，在图13-5中给出了搜索一个静态进程表以查找给定PID，pid的代码。该代码简单有效。对于小型表的链表，做同样的搜索则需要更多的工作。

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

图13-5 对于给定PID搜索进程表的代码

当存在大量的内存或者当表的利用可以猜测得相当准确时，静态表是最佳的。例如，在一个单用户系统中，用户不太可能立刻启动64个以上的进程，并且如果试图启动第65个进程失败了，也并不是一个彻底的灾难。

还有另一种选择是使用一个固定大小的表，但是如果该表填满了，就分配一个新的固定大小的表，比方说大小是原来的两倍。然后将当前的表项复制到新表中并且把旧表返回空闲存储池。这样，表总是连续的而不是链接的。此处的缺点是需要某些存储管理，并且现在表的地址是变量而不是常量。

对于内核栈也存在类似的问题。当一个线程切换到内核模式，或者当一个内核模式线程运行时，它在内核空间中需要一个栈。对于用户线程，栈可以初始化成从虚拟地址空间的顶部向下生长，所以大小不需要预先设定。对于内核线程，大小必须预先设定，因为栈占据了某些内核虚拟地址空间并且可能存在许多栈。问题是：每个栈应该得到多少空间？此处的权衡与进程表是类似的。

另一个静态-动态权衡是进程调度。在某些系统中，特别是在实时系统中，调度可以预先静态地完成。例如，航空公司在班机启航前几周就知道它的飞机什么时候要出发。类似地，多媒体系统预先知道何时调度音频、视频和其他进程。对于通用的应用，这些考虑是不成立的，并且调度必须是动态的。

还有一个静态-动态问题是内核结构。如果内核作为单一的二进制程序建立并且装载到内存中运行，情况是比较简单的。然而，这一设计的结果是添加一个新的I/O设备就需要将内核与新的设备驱动程序重新链接。UNIX的早期版本就是以这种方式工作的，在小型计算机环境中它相当令人满意，那时添加新的I/O设备是十分罕见的事情。如今，大多数操作系统允许将代码动态地添加到内核之中，随之而来的则是所有额外的复杂性。

13.3.7 自顶向下与自底向上的实现

虽然最好是自顶向下地设计系统，但是在理论上系统可以自顶向下或者自底向上地实现。在自顶向下的实现中，实现者以系统调用处理程序为开端，并且探究需要什么机制和数据结构来支持它们。接着编写这些过程等，直到触及硬件。

这种方法的问题是，由于只有顶层过程可用，任何事情都难于测试。出于这样的原因，许多开发人员发现实际上自底向上地构建系统更加可行。这一方法需要首先编写隐藏底层硬件的代码，特别是图11-6中的HAL。中断处理程序和时钟驱动程序也是早期就需要的。

然后，可以使用一个简单的调度器（例如轮转调度）来解决多道程序设计问题。在这一时刻，测试系统以了解它是否能够正确地运行多个进程应该是可能的。如果运转正常，此时可以开始仔细地定义贯穿系统的各种各样的表格和数据结构，特别是那些用于进程和线程管理以及后面内存管理的表格与数据结构。I/O和文件系统在最初可以等一等，用于测试和调试目的的读键盘与写屏幕的基本方法除外。在某些情况下，关键的低层数据结构应该得到保护，这可以通过只允许经由特定的访问过程来访问而实现——实际上这是面向对象的程序设计思想，不论采用何种程序设计语言。当较低的层次完成时，可以彻底地测试它们。这样，系统自底向上推进，很像是建筑商建造高层办公楼的方式。

如果有一个大型团队可用，那么替代的方法是首先做出整个系统的详细设计，然后分配不同的小组编写不同的模块。每个小组独立地测试自己的工作。当所有的部分都准备好时，可以将它们集成起来并