加以测试。这一设计方式存在的问题是，如果最初没有什么可以运转，可能难于分离出一个或多个模块是否工作不正常，或者一个小组是否误解了某些其他模块应该做的事情。尽管如此，如果有大型团队，还是经常使用该方法使程序设计工作中的并行程度最大化。

13.3.8 实用技术

我们刚刚了解了系统设计与实现的某些抽象思想，现在将针对系统实现考察一些有用的具体技术。这方面的技术很多，但是篇幅的限制使我们只能介绍其中的少数技术。

1. 隐藏硬件

许多硬件是十分麻烦的，所以只好尽早将其隐藏起来（除非它要展现能力，而大多数硬件不会这样）。某些非常低层的细节可以通过如图13-2所示的HAL类型的层次得到隐藏。然而，许多硬件细节不能以这样的方式来隐藏。

值得尽早关注的一件事情是如何处理中断。中断使得程序设计令人不愉快，但是操作系统必须对它们进行处理。一种方法是立刻将中断转变成别的东西，例如，每个中断都可以转变成即时弹出的线程。在这一时刻，我们处理的是线程，而不是中断。

第二种方法是将每个中断转换成在一个互斥量上的unlock操作，该互斥量对应正在等待的驱动程序。于是，中断的惟一效果就是导致某个线程变为就绪。

第三种方法是将一个中断转换成发送给某个线程的消息。低层代码只是构造一个表明中断来自何处的消息，将其排入队列，并且调用调度器以（潜在地）运行处理程序，而处理程序可能正在阻塞等待该消息。所有这些技术，以及其他类似的技术，都试图将中断转换成线程同步操作。让每个中断由一个适当的线程在适当的上下文中处理，比起在中断碰巧发生的随意上下文中运行处理程序，前者要更加容易管理。当然，这必须高效率地进行，而在操作系统内部深处，一切都必须高效率地进行。

大多数操作系统被设计成运行在多个硬件平台上。这些平台可以按照CPU芯片、MMU、字长、RAM大小以及不能容易地由HAL或等价物屏蔽的其他特性来区分。尽管如此，人们高度期望拥有单一的一组源文件用来生成所有的版本，否则，后来发现的每个程序错误必须在多个源文件中修改多次，从而有源文件逐渐疏远的危险。

某些硬件的差异，例如RAM大小，可以通过让操作系统在引导的时候确定其取值并且保存在一个变量中来处理。内存分配器可以利用RAM大小变量来确定构造多大的数据块高速缓存、页表等。甚至静态的表格，如进程表，也可以基于总的可用内存来确定大小。

然而，其他的差异，例如不同的CPU芯片，就不能让单一的二进制代码在运行的时候确定它正在哪一个CPU上运行。解决一个源代码多个目标机的问题的一种方法是使用条件编译。在源文件中，定义了一定的编译时标志用于不同的配置，并且这些标志用来将独立于CPU、字长、MMU等的代码用括号括起。例如，设想一个操作系统运行在Pentium和UltraSPARC芯片上，这就需要不同的初始化代码。可以像图13-6a中那样编写init过程的代码。根据CPU的取值（该值定义在头文件config.h中），实现一种初始化或其他的初始化过程。由于实际的二进制代码只包含目标机所需要的代码，这样就不会损失效率。

```
#include "config.h"

init( )
{
#if (CPU == PENTIUM)
/*此处是Pentium的初始化*/
#endif

#if (CPU == ULTRASPARC)
/*此处是UltraSPARC的初始化*/
#endif
}
```

a)

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

b)

图13-6 a) 依赖CPU的条件编译；b) 依赖字长的条件编译