在最低限度上，这些过程应该对任何磁盘起作用。但是我们可以比这更进一步。相同的过程还可以用于管理内存块、文件系统块高速缓存中的块，以及i节点。事实上，它们可以用来分配与回收能够线性编号的任意资源。

4. 重入

重入指的是代码同时被执行两次或多次的能力。在多处理器系统上，总是存在着这样的危险：当一个CPU执行某个过程时，另一个CPU在第一个完成之前也开始执行它。在这种情况下，不同CPU上的两个（或多个）线程可能在同时执行相同的代码。这种情况必须通过使用互斥量或者某些其他保护临界区的方法进行处理。

然而，在单处理器上，问题也是存在的。特别地，大多数操作系统是在允许中断的情况下运行的。否则，将丢失许多中断并且使系统不可靠。当操作系统忙于执行某个过程$P$时，完全有可能发生一个中断并且中断处理程序也调用$P$。如果$P$的数据结构在中断发生的时刻处于不一致的状态，中断处理程序就会注意到它们处于不一致的状态并且失败。

可能发生这种情况的一个显而易见的例子是$P$是调度器。假设某个进程用完了它的时间配额，并且操作系统正将其移动到其队列的末尾。在列表处理的半路，中断发生了，使得某个进程就绪，并且运行调度器。由于队列处于不一致的状态，系统有可能会崩溃。因此，即使在单处理器上，最好是操作系统的大部分为可重入的，关键的数据结构用互斥量来保护，并且在中断不被允许的时刻禁用中断。

5. 蛮力法

使用蛮力法解决问题多年以来获得了较差的名声，但是依据简单性它经常是行之有效的方法。每个操作系统都有许多很少会调用的过程或是具有很少数据的操作，不值得对它们进行优化。例如，在系统内部经常有必要搜索各种表格和数组。蛮力算法只是让表格保持表项建立时的顺序，并且当必须查找某个东西时线性地搜索表格。如果表项的数目很少（例如少于1000个），对表格排序或建立散列表的好处不大，但是代码却复杂得多并且很有可能在其中存在错误。

当然，对处于关键路径上的功能，例如上下文切换，使它们加快速度的一切措施都应该尽力去做，即使可能要用汇编语言编写它们。但是，系统的大部分并不处于关键路径上。例如，许多系统调用很少被调用。如果每隔1秒有一个fork调用，并且该调用花费1毫秒完成，那么即便将其优化到花费0秒也不过仅有0.1%的获益。如果优化过的代码更加庞大且有更多错误，那就不必多此一举了。

6. 首先检查错误

由于各种各样的原因，许多系统调用可能潜在地会失败：要打开的文件属于他人；因为进程表满而创建进程失败；或者因为目标进程不存在而使信号不能被发送。操作系统在执行调用之前必须无微不至地检查每一个可能的错误。

许多系统调用还需要获得资源，例如进程表的空位、i节点表的空位或文件描述符。一般性的建议是在获得资源之前，首先进行检查以了解系统调用能否实际执行，这样可以省去许多麻烦。这意味着，将所有的测试放在执行系统调用的过程的开始。每个测试应该具有如下的形式：

```
if (error_condition) return(ERROR_CODE);
```

如果调用通过了所有严格的测试，那么就可以肯定它将会取得成功。在这一时刻它才能获得资源。

如果将获得资源的测试分散开，那么就意味着如果在这一过程中某个测试失败，到这一时刻已经获得的所有资源都必须归还。如果在这里发生了一个错误并且资源没有被归还，可能并不会立刻发生破坏。例如，一个进程表项可能只是变得永久地不可用。然而，随着时间的流逝，这一差错可能会触发多次。最终，大多数或全部进程表项可能都会变得不可用，导致系统以一种极度不可预料且难以调试的方式崩溃。

许多系统以内存泄漏的形式遭受了这一问题的侵害。典型地，程序调用malloc分配了空间，但是以后忘记了调用free释放它。逐渐地，所有的内存都消失了，直到系统重新启动。

Engler等人（2000）推荐了一种有趣的方法在编译时检查某些这样的错误。他们注意到程序员知道许多定式而编译器并不知道，例如当你锁定一个互斥量的时候，所有在锁定操作处开始的路径都必须包含一个解除锁定的操作并且在相同的互斥量上没有更多的锁定。他们设计了一种方法让程序员将这一事实告诉编译器，并且指示编译器在编译时检查所有路径以发现对定式的违犯。程序员还可以设定已分配