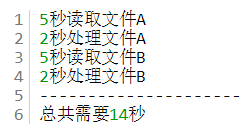
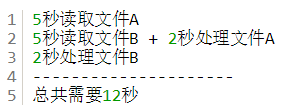
## 优点

* 资源利用率更好

比如一个应用程序需要从本地文件系统中读取和处理文件，从磁盘读取一个文件需要5秒，处理一个文件需要2秒。处理两个文件则需要：



从磁盘读取文件的时候，大部分的CPU时间用于等待磁盘去读取数据。在这段时间里CPU非常的空闲。它可以做一些别的事情。通过改变操作的顺序，就能够更好的使用CPU资源。



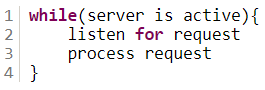
总的来说，CPU能够在等待IO的时候做一些其他的事情。这个不一定就是磁盘IO。也可以是网络的IO，或者用户输入。通常情况下，网络和磁盘的IO比CPU和内存的IO慢的多。

* 程序设计在某些情况下更简单

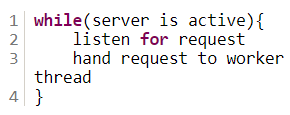
在单线程应用程序中，如果你想编写程序手动处理上面所提到的读取和处理的顺序，你必须记录每个文件读取和处理的状态。相反，你可以启动两个线程，每个线程处理一个文件的读取和操作。线程会在等待磁盘读取文件的过程中被阻塞。在等待的时候，其他的线程能够使用CPU去处理已经读取完的文件。其结果就是，磁盘总是在繁忙地读取不同的文件到内存中。这会带来磁盘和CPU利用率的提升。而且每个线程只需要记录一个文件，因此这种方式也很容易编程实现。

* 程序响应更快

将一个单线程应用程序变成多线程应用程序的另一个常见的目的是实现一个响应更快的应用程序。设想一个服务器应用，它在某一个端口监听进来的请求。当一个请求到来时，它去处理这个请求，然后再返回去监听。



如果一个请求需要占用大量的时间来处理，在这段时间内新的客户端就无法发送请求给服务端。只有服务器在监听的时候，请求才能被接收。另一种设计是，监听线程把请求传递给工作者线程(workerthread)，然后立刻返回去监听。而工作者线程则能够处理这个请求并发送一个回复给客户端。



这种方式，服务端线程迅速地返回去监听。因此，更多的客户端能够发送请求给服务端。这个服务也变得响应更快。

## 代价

* 设计更复杂

在多线程访问共享数据的时候，这部分代码需要特别注意。线程之间的交互往往非常复杂。不正确的线程同步产生的错误非常难以被发现，并且重现以修复。

* 上下文切换的开销

当CPU从执行一个线程切换到另外一个线程的时候，它需要先存储当前线程的本地的数据，程序指针等，然后载入另一个线程的本地数据，程序指针等，最后才开始执行。这种切换称为“上下文切换（context switch）”。CPU会在一个上下文中执行一个线程，然后切换到另外一个上下文中执行另外一个线程。如无必要应减少上下文切换。

* 增加资源消耗

线程在运行的时候需要从计算机里面得到一些资源。除了CPU，线程还需要一些内存来维持它本地的堆栈。它也需要占用操作系统中一些资源来管理线程。我们可以尝试编写一个程序，让它创建100个线程，这些线程什么事情都不做，只是在等待，然后看看这个程序在运行的时候占用了多少内存。