# 怎样理解阻塞非阻塞与同步异步的区别？

## IO 概念区分

四个相关概念：

* 同步（Synchronous）
* 异步( Asynchronous)
* 阻塞( Blocking )
* 非阻塞( Nonblocking)

这四个概念的含义以及相互之间的区别与联系，并不如很多网络博客所写的那么简单, 通过举一些什么商店购物， 买书买报的例子就能讲清楚。

## 进程间通信的同步/异步， 阻塞/非阻塞

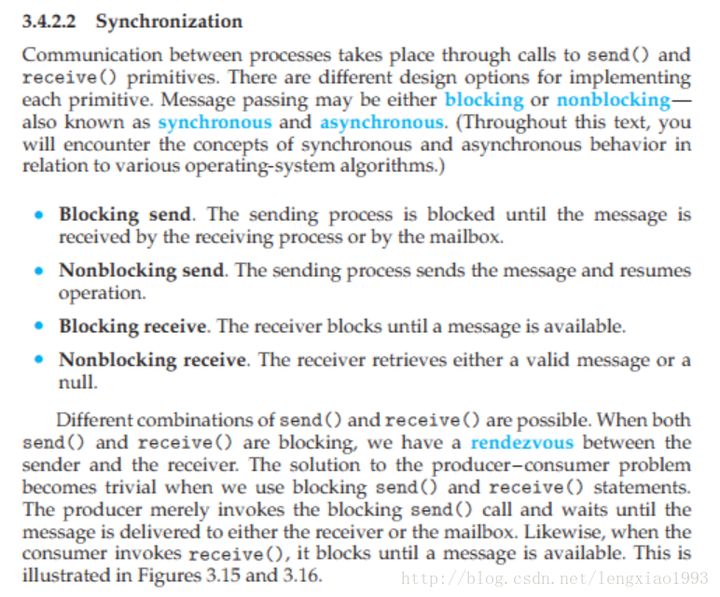
首先强调一点， 网络上很多博文关于同步/异步， 阻塞非阻塞区别的解释其实都经不起推敲。 例如在 **严肃** 的这一高赞回答中 ， 有如下解释（不准确）：

* 同步/异步关注的是消息通信机制 (synchronous communication/ asynchronous communication) 。
* 所谓同步，就是在发出一个调用时，在没有得到结果之前， 该调用就不返回。
* 异步则是相反，调用在发出之后，这个调用就直接返回了，所以没有返回结果
* 阻塞/非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.
* 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
* 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

粗一看， 好像**同步/ 非同步**， **阻塞/非阻塞** 是两种维度的概念， 可以分别对待， 但是稍微推敲一下就会发现上述的解释存在不妥之处。

* 如果**“同步”**是发起了一个调用后， 没有得到结果之前不返回， 那它毫无疑问就是被**“阻塞”**了（即调用进程处于 “waiting” 状态）。
* 如果**“异步”**调用发出了以后就直接返回了， 毫无疑问， 这个进程没有被**“阻塞”**。

所以， 上述的解释是不准确的。 让我们看一下《操作系统概念（第九版）》中有关进程间通信的部分是如何解释的：



翻译一下就是：

进程间的通信是通过 [send](https://www.zhihu.com/search?q=send&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)() 和 receive() 两种基本操作完成的。具体如何实现这两种基础操作，存在着不同的设计。 消息的传递有可能是**阻塞的**或**非阻塞的** – 也被称为**同步**或**异步**的：

* 阻塞式发送（blocking send）. 发送方进程会被一直阻塞， 直到消息被接受方进程收到。
* 非阻塞式发送（nonblocking send）。 发送方进程调用 send() 后， 立即就可以其他操作。
* 阻塞式接收（blocking receive） 接收方调用 receive() 后一直阻塞， 直到消息到达可用。
* 非阻塞式接受（nonblocking receive） 接收方调用 receive() 函数后， 要么得到一个有效的结果， 要么得到一个空值， 即不会被阻塞。

**上述不同类型的发送方式和不同类型的接收方式，可以自由组合。**

* 也就是说， 从进程级通信的维度讨论时， 阻塞和同步（非阻塞和异步）就是一对同义词， 且需要针对**发送方**和**接收方**作区分对待。

**先修知识**

* 用户空间和内核空间
* 进程切换
  + 系统调用（system call）
  + 中断（interrupt）
* 进程的阻塞

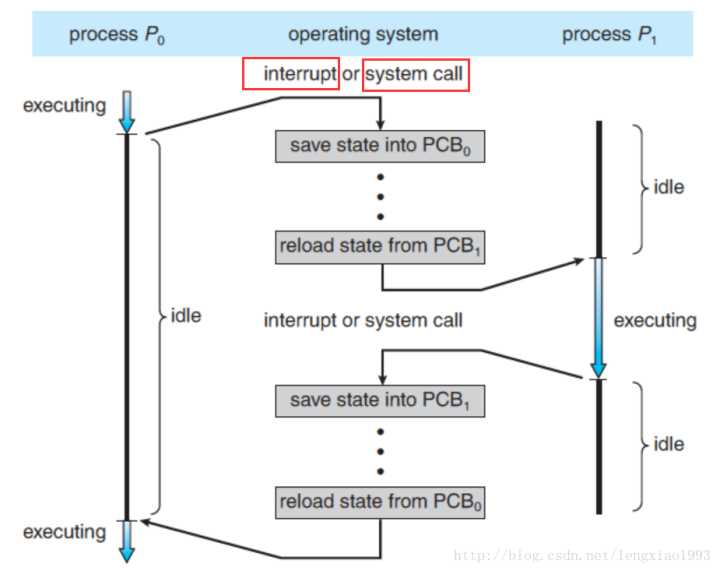
**用户空间和内核空间**

操作系统为了支持多个应用同时运行，需要保证不同进程之间相对独立（一个进程的崩溃不会影响其他的进程 ， 恶意进程不能直接读取和修改其他进程运行时的代码和数据）。 因此操作系统内核**需要拥有高于普通进程的权限**， 以此来调度和管理用户的应用程序。

于是内存空间被划分为两部分，一部分为内核空间，一部分为用户空间，内核空间存储的代码和数据具有更高级别的权限。内存访问的**相关硬件**在程序执行期间会进行访问控制（ Access Control），使得用户空间的程序不能直接读写内核空间的内存。

* 有《[微机原理](https://www.zhihu.com/search?q=%E5%BE%AE%E6%9C%BA%E5%8E%9F%E7%90%86&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)》 课程基础同学可以 Google 搜索 DPL, CPL 这两个关键字了解硬件层面的内存访问权限控制细节

## 进程切换



上图展示了进程切换中几个最重要的步骤：

1. 当一个程序正在执行的过程中， 中断（interrupt） 或 系统调用（system call） 发生可以使得 CPU 的控制权会从当前进程转移到操作系统内核。
2. 操作系统内核负责保存进程 i 在 CPU 中的上下文（程序计数器， 寄存器）到 PCBi （操作系统分配给进程的一个[内存块](https://www.zhihu.com/search?q=%E5%86%85%E5%AD%98%E5%9D%97&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)）中。
3. 从 PCBj 取出进程 j 的CPU 上下文， 将 CPU 控制权转移给进程 j ， 开始执行进程 j 的指令。

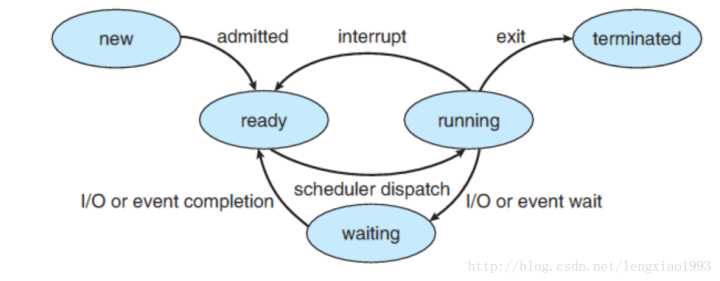
几个底层概念的通俗（不严谨）解释：

* 中断（interrupt）
  + CPU [微处理器](https://www.zhihu.com/search?q=%E5%BE%AE%E5%A4%84%E7%90%86%E5%99%A8&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)有一个中断信号位， 在每个CPU时钟周期的末尾, CPU会去检测那个中断信号位是否有中断信号到达， 如果有， 则会根据中断优先级决定是否要暂停当前执行的指令， 转而去执行处理中断的指令。 （其实就是 CPU 层级的 while 轮询）
* 时钟中断( Clock Interrupt )
  + 一个硬件时钟会每隔一段（很短）的时间就产生一个中断信号发送给 CPU，CPU 在响应这个中断时， 就会去执行操作系统内核的指令， 继而将 CPU 的控制权转移给了操作系统内核， 可以由操作系统内核决定下一个要被执行的指令。
* 系统调用（system call）
  + system call 是操作系统提供给应用程序的接口。 用户通过调用 systemcall 来完成那些需要操作系统内核进行的操作， 例如硬盘， 网络接口设备的读写等。

从上述描述中，可以看出来， 操作系统在进行进切换时，需要进行一系列的内存读写操作，这带来了一定的开销：

* 对于一个运行着 UNIX 系统的现代 PC 来说， 进程切换通常至少需要花费 300 us 的时间

## 进程阻塞



上图展示了一个进程的不同状态：

* New. 进程正在被创建.
* Running. 进程的指令正在被执行
* Waiting. 进程正在等待一些事件的发生（例如 I/O 的完成或者收到某个信号）
* Ready. 进程在等待被操作系统调度
* Terminated. 进程执行完毕（可能是被强行终止的）

我们所说的 “阻塞”是指进程在**发起了一个系统调用**（System Call） 后， 由于该系统调用的操作不能立即完成，需要等待一段时间，于是内核将进程挂起为**等待 （waiting）**状态， 以确保它不会被调度执行， 占用 CPU 资源。

* 友情提示： **在任意时刻， 一个 CPU 核心上（processor）只可能运行一个进程** 。

## I/O System Call 的阻塞/非阻塞， 同步/异步

这里再重新审视 **阻塞/非阻塞 IO** 这个概念， 其实**阻塞和非阻塞**描述的是进程的一个操作是否会使得进程转变为“等待”的状态， 但是为什么我们总是把它和 IO 连在一起讨论呢？

原因是， **阻塞**这个词是与系统调用 System Call 紧紧联系在一起的， 因为要让一个进程进入 等待（waiting） 的状态, 要么是它主动调用 wait() 或 sleep() 等挂起自己的操作， 另一种就是它调用 System Call, 而 System Call 因为涉及到了 I/O 操作， 不能立即完成， 于是内核就会先将该进程置为等待状态， 调度其他进程的运行， 等到 它所请求的 I/O 操作完成了以后， 再将其状态更改回 [ready](https://www.zhihu.com/search?q=ready&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D) 。

操作系统内核在执行 System Call 时， CPU 需要与 IO 设备完成一系列物理通信上的交互， 其实再一次会涉及到阻塞和非阻塞的问题， 例如， 操作系统发起了一个读硬盘的请求后， 其实是向硬盘设备通过总线发出了一个请求，它即可以阻塞式地等待IO 设备的返回结果，也可以非阻塞式的继续其他的操作。 在现代计算机中，这些物理通信操作基本都是异步完成的， 即发出请求后， 等待 I/O 设备的中断信号后， 再来读取相应的设备[缓冲区](https://www.zhihu.com/search?q=%E7%BC%93%E5%86%B2%E5%8C%BA&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)。 但是，大部分操作系统默认为用户级应用程序提供的都是阻塞式的系统调用 （blocking systemcall）接口， 因为阻塞式的调用，使得应用级代码的编写更容易（代码的执行顺序和编写顺序是一致的）。

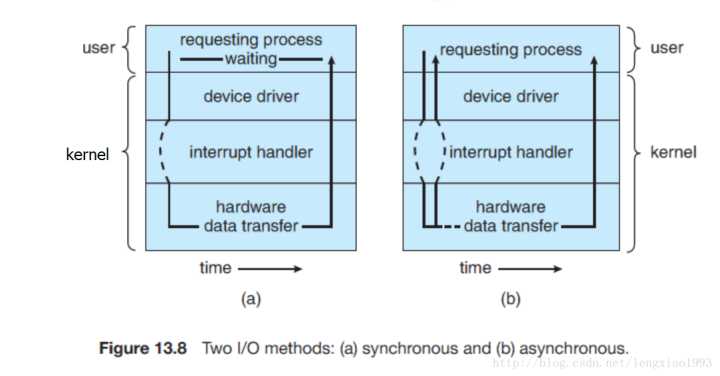
但同样， 现在的大部分操作系统也会提供非阻塞I/O 系统调用接口（Nonblocking I/O system call）。 一个非阻塞调用不会挂起调用程序， 而是会立即返回一个值， 表示有多少bytes 的数据被成功读取（或写入）。

非阻塞I/O 系统调用( nonblocking system call )的另一个替代品是 **异步I/O系统调用 （asychronous system call）**。 与非阻塞 I/O 系统调用类似，asychronous system call 也是会立即返回， 不会等待 I/O 操作的完成， 应用程序可以继续执行其他的操作， 等到 I/O 操作完成了以后，操作系统会通知调用进程（设置一个用户空间特殊的[变量值](https://www.zhihu.com/search?q=%E5%8F%98%E9%87%8F%E5%80%BC&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D) 或者 触发一个 signal 或者 产生一个软中断 或者 调用应用程序的回调函数）。

此处， 非阻塞I/O 系统调用( nonblocking system call ) 和 异步I/O系统调用 （asychronous system call）的区别是：

* 一个**非阻塞I/O 系统调用 read()** 操作立即返回的是任何可以立即拿到的数据， 可以是完整的结果， 也可以是不完整的结果， 还可以是一个空值。
* 而**异步I/O系统调用** read（）结果必须是完整的， 但是这个操作完成的通知可以延迟到将来的一个时间点。

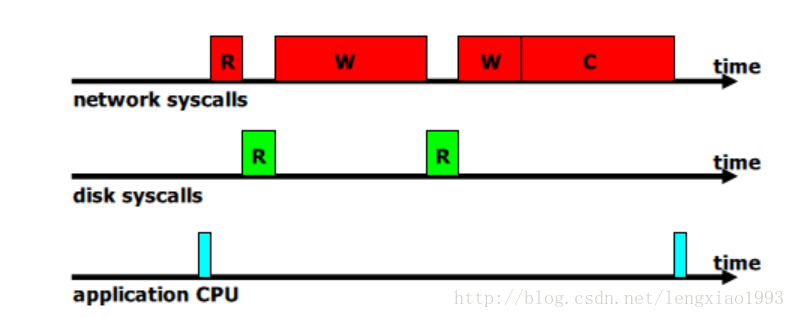
下图展示了同步I/O 与 异步 I/O 的区别 （非阻塞 IO 在下图中没有绘出）.



注意， 上面提到的 **非阻塞I/O 系统调用( nonblocking system call )** 和 **异步I/O系统调用** 都是非阻塞式的行为（non-blocking behavior）。 他们的差异仅仅是返回结果的方式和内容不同。

## 非阻塞 I/O 如何帮助服务器提高[吞吐量](https://www.zhihu.com/search?q=%E5%90%9E%E5%90%90%E9%87%8F&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)

考虑一个**单进程**服务器程序， 收到一个 Socket 连接请求后， 读取请求中的文件名，然后读请求的文件名内容，将文件内容返回给客户端。 那么一个请求的处理流程会如下图所示。



* R 表示读操作
* W 表示写操作
* C 表示关闭操作

在这个过程中， 我们可以看到， CPU 和 硬盘IO 的资源大部分时间都是闲置的。 此时， 我们会希望在等待 I/O 的过程中继续处理新的请求。

方案一： [多进程](https://www.zhihu.com/search?q=%E5%A4%9A%E8%BF%9B%E7%A8%8B&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D)

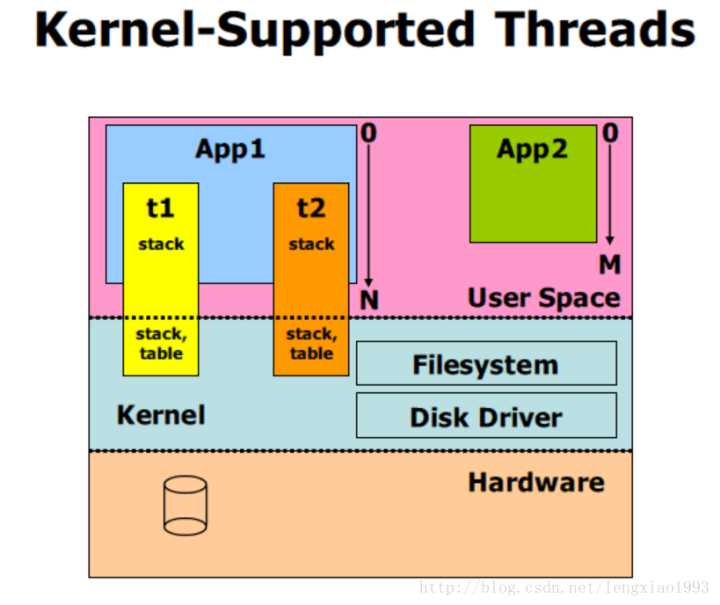
* 每到达一个请求， 我们为这个请求新创建一个进程来处理。 这样， 一个进程在等待 IO 时， 其他的进程可以被调度执行， 更加充分地利用 CPU 等资源。
* 问题： 每新创建一个进程都会消耗一定的内存空间， 且进程切换也会有时间消耗， 高并发时， 大量进程来回切换的时间开销会变得明显起来。

方案二：多线程

* 和多进程方案类似，为每一个请求新建一个线程进行处理，这样做的重要区别是， 所有的线程都共享同一个进程空间
* 问题： 需要考虑是否需要为特定的逻辑使用锁。

引申问题： 一个进程中的某一个线程发起了 system call 后， 是否造成整个进程的阻塞？ 如果会， 那么多线程方案与单进程方案相比就没有明显的改善。

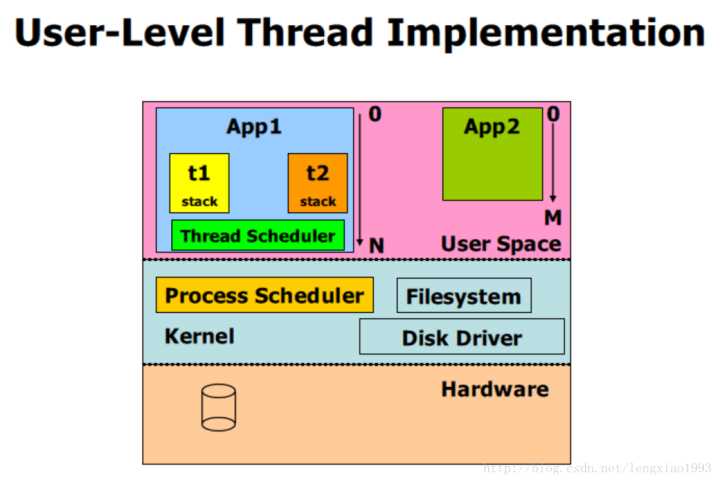
* 解决办法1：内核支持的线程（kenerl supported threads）
  + 操作系统内核能够感知到线程， 每一个线程都会有一个内核调用栈（kenerl stack） 和 保存CPU 寄存器下文的 table 。



在这种方案中，如果 CPU 是多核的，不同的线程还可以运行在不同的 CPU processor 上。既实现了IO 并发，也实现了 CPU 并发。

问题：内核支持线程可移植性差， 其实现对于不同的操作系统而言有所差别。

* 解决办法2： 用户支持的线程（user supported threads）
  + 内核感知不到用户线程， 每一个用户的进程拥有一个调度器， 该调度器可以感知到线程发起的系统调用， 当一个线程产生系统调用时， 不阻塞整个进程， 切换到其他线程继续运行。 当 I/O 调用完成以后， 能够重新唤醒被阻塞的线程。
  + 实现细节：
    - 应用程序基于[线程库](https://www.zhihu.com/search?q=%E7%BA%BF%E7%A8%8B%E5%BA%93&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A241673170%7D) thread libray 编写
    - 线程库中包含 “虚假的” read(), write(), accept()等系统调用。
    - 线程库中的 read(), write(), accept() 的底层实现为非阻塞系统调用（Non-blocking system call）， 调用后，由于可以立即返回， 则将特定的线程状态标记为 waiting, 调度其他的可执行线程。 内核完成了 IO 操作后， 调用线程库的回调函数， 将原来处于 waiting 状态的线程标记为 runnable.



从上面的过程可以看出，用户级支持线程（User-Supported Threads）的解决方案基于非阻塞IO系统调用( non-blocking system call) ， 且是一种基于操作系统内核事件通知（event-driven）的解决方案， 该方案可以降低系统处理并发请求时的进程切换开销。 基于这个方案， 可以引申到更为宽泛的 event-driven progreamming 话题上。 但是这里就不作赘述了。

## 总结：

1. 阻塞/非阻塞， 同步/异步的概念要注意讨论的上下文：

* 在进程通信层面， 阻塞/非阻塞， 同步/异步基本是同义词， 但是需要注意区分讨论的对象是发送方还是接收方。
* 发送方阻塞/非阻塞（同步/异步）和接收方的阻塞/非阻塞（同步/异步） 是互不影响的。
* 在 IO 系统调用层面（ IO system call ）层面， **非阻塞 IO 系统调用** 和 **异步 IO 系统调用**存在着一定的差别， 它们都不会阻塞进程， 但是返回结果的方式和内容有所差别， 但是都属于非阻塞系统调用（ non-blocing system call ）

2. 非阻塞系统调用（non-blocking I/O system call 与 asynchronous I/O system call） 的存在可以用来实现线程级别的 I/O 并发， 与通过多进程实现的 I/O 并发相比可以减少内存消耗以及进程切换的开销。