

本行然后就停止”的恶意客户端的攻击面前，我们的事件驱动服务器显得很脆弱。修改事件驱动服务器来处理部分文本行不是一个简单的任务，但是基于进程的设计却能处理得很好，而且是自动处理的。基于事件的设计另一个重要的缺点是它们不能充分利用多核处理器。

12.3 基于线程的并发编程

到目前为止，我们已经看到了两种创建并发逻辑流的方法。在第一种方法中，我们为每个流使用了单独的进程。内核会自动调度每个进程，而每个进程有它自己的私有地址空间，这使得流共享数据很困难。在第二种方法中，我们创建自己的逻辑流，并利用 I/O 多路复用来显式地调度流。因为只有一个进程，所有的流共享整个地址空间。本节介绍第三种方法——基于线程，它是这两种方法的混合。

线程(thread)就是运行在进程上下文中的逻辑流。在本书里迄今为止，程序都是由每个进程中一个线程组成的。但是现代系统也允许我们编写一个进程里同时运行多个线程的程序。线程由内核自动调度。每个线程都有它自己的线程上下文(thread context)，包括一个唯一的整数线程 ID(Thread ID, TID)、栈、栈指针、程序计数器、通用目的寄存器和条件码。所有的运行在一个进程里的线程共享该进程的整个虚拟地址空间。

基于线程的逻辑流结合了基于进程和基于 I/O 多路复用的流的特性。同进程一样，线程由内核自动调度，并且内核通过一个整数 ID 来识别线程。同基于 I/O 多路复用的流一样，多个线程运行在单一进程的上下文中，因此共享这个进程虚拟地址空间的所有内容，包括它的代码、数据、堆、共享库和打开的文件。

12.3.1 线程执行模型

多线程的执行模型在某些方面和多进程的执行模型是相似的。思考图 12-12 中的示例。每个进程开始生命周期时都是单一线程，这个线程称为主线程(main thread)。在某一时刻，主线程创建一个对等线程(peer thread)，从这个时间点开始，两个线程就并发地运行。最后，因为主线程执行一个慢速系统调用，例如 read 或者 sleep，或者因为被系统的间隔计时器中断，控制就会通过上下文切换传递到对等线程。对等线程会执行一段时间，然后控制传递回主线程，依次类推。

在一些重要的方面，线程执行是不同于进程的。因为一个线程的上下文要比一个进程的上下文小得多，线程的上下文切换要比进程的上下文切换快得多。另一个不同就是线程不像进程那样，不是按照严格的父子层次来组织的。和一个进程相关的线程组成一个对等(线程)池，独立于其他线程创建的线程。主线程和其他线程的区别仅在于它总是进程中第一个运行的线程。对等(线程)池概念的主要影响是，一个线程可以杀死它的任何对等线程，或者等待它的任意对等线程终止。另外，每个对等线程都能读写相同的共享数据。

12.3.2 Posix 线程

Posix 线程(Pthreads)是在 C 程序中处理线程的一个标准接口。它最早出现在 1995

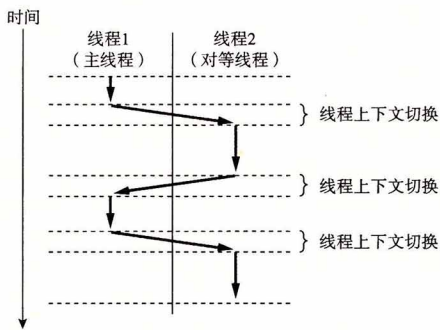


图 12-12 并发线程执行