


然而，这样可能会出错，因为它在对等线程的赋值语句和主线程的 `accept` 语句间引入了竞争(race)。如果赋值语句在下一个 `accept` 之前完成，那么对等线程中的局部变量 `connfd` 就得到正确的描述符值。然而，如果赋值语句是在 `accept` 之后才完成的，那么对等线程中的局部变量 `connfd` 就得到下一次连接的描述符值。那么不幸的结果就是，现在两个线程在同一个描述符上执行输入和输出。为了避免这种潜在的致命竞争，我们必须将 `accept` 返回的每个已连接描述符分配到它自己的动态分配的内存块，如第 20~21 行所示。我们会在 12.7.4 节中回过头来讨论竞争的问题。

另一个问题是在线程例程中避免内存泄漏。既然不显式地收回线程，就必须分离每个线程，使得在它终止时它的内存资源能够被收回(第 31 行)。更进一步，我们必须小心释放主线程分配的内存块(第 32 行)。

 **练习题 12.5** 在图 12-5 中基于进程的服务器中，我们在两个位置小心地关闭了已连接描述符：父进程和子进程。然而，在图 12-14 中基于线程的服务器中，我们只在一个位置关闭了已连接描述符：对等线程。为什么？

12.4 多线程程序中的共享变量

从程序员的角度来看，线程很有吸引力的一个方面是多个线程很容易共享相同的程序变量。然而，这种共享也是很棘手的。为了编写正确的多线程程序，我们必须对所谓的共享以及它是如何工作的有很清楚的了解。

为了解 C 程序中的一个变量是否是共享的，有一些基本的问题要解答：1) 线程的基础内存模型是什么？2) 根据这个模型，变量实例是如何映射到内存的？3) 最后，有多少线程引用这些实例？一个变量是共享的，当且仅当多个线程引用这个变量的某个实例。

为了让我们对共享的讨论具体化，我们将使用图 12-15 中的程序作为运行示例。尽管有些人为了的痕迹，但是它仍然值得研究，因为它说明了关于共享的许多细微之处。示例程序由一个创建了两个对等线程的主线程组成。主线程传递一个唯一的 ID 给每个对等线程，每个对等线程利用这个 ID 输出一条个性化的信息，以及调用该线程例程的总次数。

12.4.1 线程内存模型

一组并发线程运行在一个进程的上下文中。每个线程都有它自己独立的线程上下文，包括线程 ID、栈、栈指针、程序计数器、条件码和通用目的寄存器值。每个线程和其他线程一起共享进程上下文的剩余部分。这包括整个用户虚拟地址空间，它是由只读文本(代码)、读/写数据、堆以及所有的共享库代码和数据区域组成的。线程也共享相同的打开文件的集合。

从实际操作的角度来说，让一个线程去读或写另一个线程的寄存器值是不可能的。另一方面，任何线程都可以访问共享虚拟内存的任意位置。如果某个线程修改了一个内存位置，那么其他每个线程最终都能在它读这个位置时发现这个变化。因此，寄存器是从不共享的，而虚拟内存总是共享的。

各自独立的线程栈的内存模型不是那么整齐清楚的。这些栈被保存在虚拟地址空间的栈区域中，并且通常是被相应的线程独立地访问的。我们说通常而不是总是，是因为不同的线程栈是不对其他线程设防的。所以，如果一个线程以某种方式得到一个指向其他线程栈的指针，那么它就可以读写这个栈的任何部分。示例程序在第 26 行展示了这一点，其中对等线程直接通过全局变量 `ptr` 间接引用主线程的栈的内容。