使我们去使用其他一些不那么直观的机制来检测进程的终止。实际上，Stevens 在[110]中就很有说服力地论证了这是规范中的一个错误。

12.3.6　分离线程

在任何一个时间点上，线程是可结合的(joinable)或者是分离的(detached)。一个可结合的线程能够被其他线程收回和杀死。在被其他线程回收之前，它的内存资源(例如栈)是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的。它的内存资源在它终止时由系统自动释放。

默认情况下，线程被创建成可结合的。为了避免内存泄漏，每个可结合线程都应该要么被其他线程显式地收回，要么通过调用 pthread_detach 函数被分离。

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

若成功则返回 0，若出错则为非零。

pthread_detach 函数分离可结合线程 tid。线程能够通过以 pthread_self()为参数的 pthread_detach 调用来分离它们自己。

尽管我们的一些例子会使用可结合线程，但是在现实程序中，有很好的理由要使用分离的线程。例如，一个高性能 Web 服务器可能在每次收到 Web 浏览器的连接请求时都创建一个新的对等线程。因为每个连接都是由一个单独的线程独立处理的，所以对于服务器而言，就很没有必要(实际上也不愿意)显式地等待每个对等线程终止。在这种情况下，每个对等线程都应该在它开始处理请求之前分离它自身，这样就能在它终止后回收它的内存资源了。

12.3.7　初始化线程

pthread_once 函数允许你初始化与线程例程相关的状态。

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

总是返回 0。

once_control 变量是一个全局或者静态变量，总是被初始化为 PTHREAD_ONCE_INIT。当你第一次用参数 once_control 调用 pthread_once 时，它调用 init_routine，这是一个没有输入参数、也不返回什么的函数。接下来的以 once_control 为参数的 pthread_once 调用不做任何事情。无论何时，当你需要动态初始化多个线程共享的全局变量时，pthread_once 函数是很有用的。我们将在 12.5.5 节里看到一个示例。

12.3.8　基于线程的并发服务器

图 12-14 展示了基于线程的并发 echo 服务器的代码。整体结构类似于基于进程的设计。主线程不断地等待连接请求，然后创建一个对等线程处理该请求。虽然代码看似简