2000年的时候，Linux系统引入了一个新的、强大的系统调用clone，模糊了进程和线程的区别，甚至使得两个概念的重要性被倒置。任何其他UNIX系统的版本中都没有clone函数。传统观念上，当一个新线程被创建的时候，之前的线程和新线程除了寄存器内容之外共享所有的信息。特别是，已打开文件的文件描述符、信号处理函数、警报信号和其他每个进程（不是每个线程）都具有的全局属性。clone函数可以设置这些属性是进程特有的还是线程特有的。它的调用方式如下：

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

调用这个函数可以在当前进程或新的进程中创建一个新线程，具体依赖于参数sharing_flags。如果新线程在当前进程中，它将与其他已存在的线程共享地址空间，任何一个线程对地址空间做出修改对于同一进程中的其他线程而言都是立即可见的。换句话说，如果地址空间不是共享的，新线程会获得地址空间的完整副本，但是新线程对这个副本进行的修改对于旧的线程来说是不可见的。这些语义同POSIX的fork函数是相同的。

在这两种情况下，新线程都从function处开始执行，并以arg作为惟一的参数。同时，新线程还拥有私有堆栈，其中私有堆栈的指针被初始化为stack_ptr。

参数sharing_flags是一个位图，这个位图允许比传统的UNIX系统更加细粒度的共享。每一位可以单独设置，且每一位决定了新线程是复制一些数据结构还是与调用clone函数的线程共享这些数据结构。图10-9显示了根据sharing_flags的设置，哪些项可以共享，哪些项需要复制。

CLONE_VM位决定了虚拟内存（即地址空间）是与旧的线程共享还是需要复制。如果该位置1，新线程加入到已存在的线程中去，即clone函数在一个已经存在的进程中创建了一个新线程。如果该位清零，新线程会拥有私有的地址空间。拥有自己的地址空间意味着存储的操作对于之前已经存在的线程而言是不可见的。这与fork函数很相似，除了下面提到的一点。创建新的地址空间事实上就定义了一个新的进程。

标志	置位时的含义	清除时的含义
CLONE_VM	创建一个新线程	创建一个新进程
CLONE_FS	共享umask、根目录和工作目录	不共享
CLONE_FILES	共享文件描述符	复制文件描述符
CLONE_SIGHAND	共享信号句柄表	复制该表
CLONE_PID	新线程获得旧的PID	新线程获得自己的PID
CLONE_PARENT	新线程与调用者有相同的父亲	新线程的父亲是调用者

图10-9 sharing-flags位图中的各个位

CLONE_FS位控制着是否共享根目录、当前工作目录和umask标志。即使新线程拥有自己的地址空间，如果该位置1，新、旧线程之间也可以共享当前工作目录。这就意味着即使一个线程拥有自己的地址空间，另一个线程也可以调用chdir函数改变它的工作目录。在UNIX系统中，一个线程通常会调用chdir函数改变它所在进程中其他线程的当前工作目录，而不会对另一进程中的线程做这样的操作。所以说，这一位引入了一种传统UNIX系统不可能具有的共享性。

CLONE_FILES位与CLONE_FS位相似。如果该位置1，新线程与旧线程共享文件描述符，所以一个线程调用lseek函数对另一个线程而言是可见的。通常，这样的处理是对于同属一个进程的线程，而不是不同进程的线程。相似的，CLONE_SIGHAND位控制是否在新、旧线程间共享信号句柄表。如果信号处理函数表是共享的，即使是在拥有不同地址空间的线程之间共享，一个线程改变某一处理函数也会影响另一个线程的处理函数。CLONE_PID位控制新线程是拥有自己的PID还是与父进程共享PID。这个特性在系统启动的时候是必需的。用户进程不允许对该位进行设置。

最后，每一个进程都有一个父进程。CLONE_PARENT位控制着哪一个线程是新线程的父线程。父线程可以与clone函数调用者的父线程相同（在这种情况下，新线程是clone函数调用者的兄弟），也可以是clone函数调用者本身，在这种情况下，新线程是clone函数调用者的子线程。还有另外一些控制其他项目的位，但是它们不是很重要。

由于Linux系统为不同的项目维护了独立的数据结构（见10.3.3小节，如调度参数、内存映射等），