China-pub.com
下载

第21章　线程本地存储器

有时，将数据与对象的实例联系起来是很有帮助的。例如，窗口的附加字节可以使用SetWindowsWord和SetWindowLong函数将数据与特定的窗口联系起来。可以使用线程本地存储器将数据与执行的特定线程联系起来。例如，可以将线程的某个时间与线程联系起来。然后，当线程终止运行时，就能够确定线程的寿命。

C/C++运行期库要使用线程本地存储器（TLS）。由于运行期库是在多线程应用程序出现前的许多年设计的，因此运行期库中的大多数函数是用于单线程应用程序的。函数 strtok就是个很好的例子。应用程序初次调用 strtok时，该函数传递一个字符串的地址，并将字符串的地址保存在它自己的静态变量中。当你将来调用 strtok函数并传递NULL时，该函数就引用保存的字符串地址。

在多线程环境中，一个线程可以调用 strtok，然后，在它能够再次调用该函数之前，另一个线程也可以调用 Strtok。在这种情况下，第二个线程会在第一个线程不知道的情况下，让strtok用一个新地址来改写它的静态变量。第一个线程将来调用 strtok时将使用第二个线程的字符串，这就会导致各种各样难以发现和排除的错误。

为了解决这个问题，C/C++运行期库使用了TLS。每个线程均被赋予它自己的字符串指针，供strtok函数使用。需要予以同样对待的其他C/C++运行期库函数还有asctime和gmtime。

如果你的应用程序需要严重依赖全局变量或静态变量，那么TLS能够帮助解决它遇到的问题。但是编程人员往往尽可能减少对这些变量的使用，而更多地依赖自动（基于堆栈的）变量和通过函数的参数传递的数据。这样做是很好的，因为基于堆栈的变量总是与特定的线程相联系的。

标准的C运行期库一直是由许多不同的编译器供应商来实现和重新实现的。如果 C编译器不包含标准的C运行期库，那么就不值得去购买它。编程员多年来一直使用标准的 C运行期库，并且将会继续使用它，这意味着strtok之类的函数的原型和行为特性必须与上面所说的标准 C运行期库完全一样。如果今天重新来设计C运行期库，那么它就必须支持多线程应用程序的环境，并且必须采取相应的措施来避免使用全局变量和静态变量。

在我的软件开发项目中，我总是尽可能避免使用全局变量和静态变量。如果你的应用程序使用全局变量和静态变量，那么建议你务必观察每个变量，并且了解一下它能否改变成基于堆栈的变量。如果打算将线程添加给应用程序，那么这样做可以节省大量时间，甚至单线程应用程序也能够从中得到许多好处。

在编写应用程序和DLL时，可以使用本章中介绍的两种TLS方法，即动态TLS和静态TLS。但是，当创建DLL时，这些TLS往往更加有用，因为DLL常常不知道它们链接到的应用程序的结构。不过，当编写应用程序时，你通常知道将要创建多少线程以及如何使用这些线程。然后就可以创造一些临时性的方法，或者最好是使用基于堆栈的方法（局部变量），将数据与创建的每个线程联系起来。不管怎样，应用程序开发人员也能从本章讲述的内容中得到一些启发。

21.1　动态TLS

若要使用动态TLS，应用程序可以调用一组 4个函数。这些函数实际上是 DLL用得最多的函数。图21-1显示了Windows用来管理TLS的内部数据结构。

China-pub.com 下载

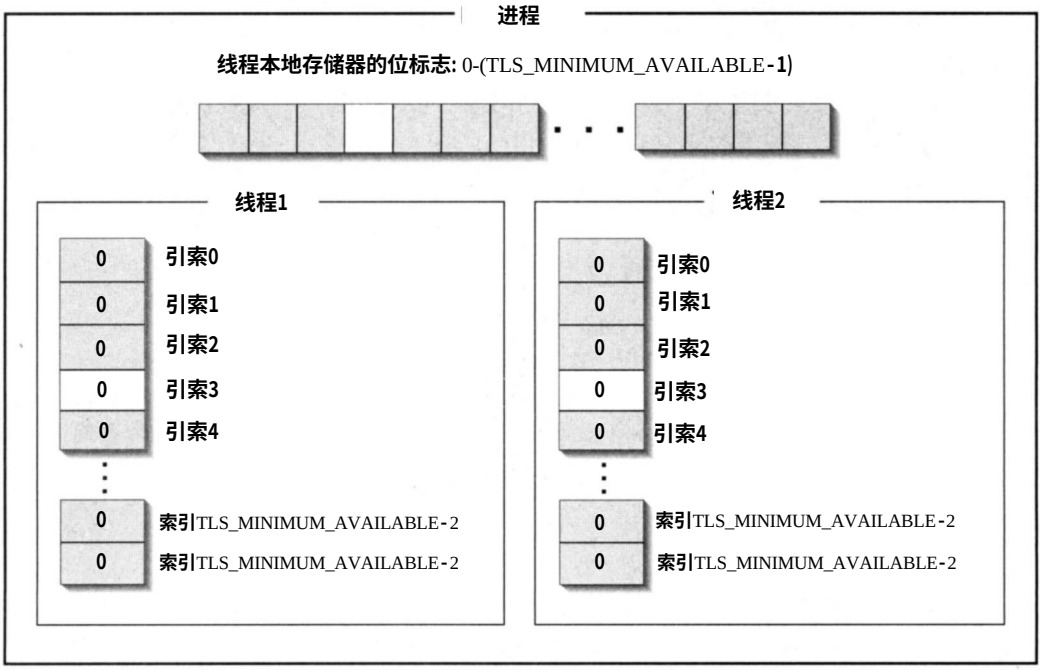

图21-1 用于管理TLS的内部数据结构

该图显示了系统中运行的线程正在使用的一组标志。每个标志均可设置为 FREE或者INUSE，表示TLS时隙(slot)是否正在使用。Microsoft保证至少TLS_MINIMUM_AVAILABLE位标志是可供使用的。另外，TLS_MINIMUM_AVAILABLE在WinNT.h中被定义为64。Windows 2000将这个标志数组扩展为允许有1000个以上的TLS时隙。对于任何一个应用程序来说，这个时隙数量足够了。

若要使用动态TLS，首先必须调用TlsAlloc函数：

```
DWORD TlsAlloc();
```

这个函数命令系统对进程中的位标志进行扫描，并找出一个 FREE标志。然后系统将该标志从FREE改为INUSE，并且TlsAlloc返回位数组中的标志的索引。DLL（或应用程序）通常将该索引保存在一个全局变量中。这是全局变量作为一个较好选择的情况之一，因为它的值是每个进程而不是每个线程使用的值。

如果TlsAlloc在该列表中找不到 FREE标志，它就返回 TLS_OUT_OF_INDEXES（在WinBase.h中定义为0xFFFFFFFF）。当TlsAlloc第一次被调用时，系统发现第一个标志是FREE，并将该标志改为INUSE，同时TlsAlloc返回0。TlsAlloc这样运行的概率是99%。下面介绍在另外的1%的概率下TlsAlloc是如何运行的。

当创建一个线程时，便分配一个 TLS_MINIMUM_AVAILABLEPVOID值的数组，并将它初始化为0，然后由系统将它与线程联系起来。如图21-1所示，每个线程均得到它自己的数组，数组中的每个PVOID可以存储任何值。

在能够将信息存储在线程的PVOID数组中之前，必须知道数组中的哪个索引可供使用，这就是前面调用TlsAlloc所要达到的目的。按照设计概念，TlsAlloc为你保留了一个索引。如果TlsAlloc返回索引3，那么就说明目前在进程中运行的每个线程中均为你保留了索引 3，而且在将来创建的线程中也保留了索引3。

China-pub.com
下载

若要将一个值放入线程的数组中，可以调用TlsSetValue函数：

```
BOOL TlsSetValue(
   DWORD dwTlsIndex,
   PVOID pvTlsValue);
```

该函数将一个PVOID值（用pvTlsValue参数标识）放入线程的数组中由dwTlsIndex参数标识的索引处。PvTlsValue的值与调用TlsSetValue的线程相联系。如果调用成功，便返回TRUE。

线程在调用TlsSetValue时，可以改变它自己的数组。但是它不能为另一个线程设置TLS值。我希望有另一个Tls函数能够用于使一个线程将数据存储到另一个线程的数组中，但是不存在这样一个函数。目前，将数据从一个线程传递到另一个线程的唯一方法是，将单个值传递给CreateThread或_beginthreadex，然后该函数将该值作为唯一的参数传递给线程的函数。

当调用TlsSetValue时，始终都应该传递较早的时候调用的 TlsAlloc函数返回的索引。Microsoft设计的这些函数能够尽快地运行，在运行时，将放弃错误检查。如果传递的索引是调用TlsAlloc时从未分配的，那么系统将设法把该值存储在线程的数组中，而不进行任何错误检查。

若要从线程的数组中检索一个值，可以调用TlsGetValue：

```
PVOID TlsGetValue(DWORD dwTlsIndex);
```

该函数返回的值将与索引 dwTlsIndex处的TLS时隙联系起来。与 TlsSetValue一样，TlsGetValue只查看属于调用线程的数组。还有， TlsGetValue并不执行任何测试，以确定传递的索引的有效性。

当在所有线程中不再需要保留TLS时隙的位置的时候，应该调用TlsFree：

```
BOOL TlsFree(DWORD dwTlsIndex);
```

该函数简单地告诉系统该时隙不再需要加以保留。由进程的位标志数组管理的INUSE标志再次被设置为FREE。如果线程在后来调用TlsAlloc函数，那么将来就分配该INUSE标志。如果TlsFree函数运行成功，该函数将返回TRUE。如果试图释放一个没有分配的时隙，将产生一个错误。

使用动态TLS

通常情况下，如果 DLL使用TLS，那么当它用 DLL_PROCESS_ATTACH标志调用它的DllMain函数时，它也调用TlsAlloc。当它用DLL_PROCESS_DETACH调用DllMain函数时，它就调用TlsFree。对TlsSetValue和TlsGetValue的调用很可能是在调用DLL中包含的函数时进行的。

将TLS添加给应用程序的方法之一是在需要它时进行添加。例如，你的DLL中可能有一个运行方式类似strtok的函数。第一次调用这个函数时，线程传递一个指向40字节的结构的指针。必须保存这个结构，这样，将来调用函数时就可以引用它。可以像下面这样对你的函数进行编码：

```
DWORD g_dwTlsIndex; // Assume that this is initialized
                    // with the result of a call to TlsAlloc.

 :
 :

void MyFunction(PSOMESTRUCT pSomeStruct) {
   if (pSomeStruct != NULL) {
      // The caller is priming this function.

      // See if we already allocated space to save the data.
      if (TlsGetValue(g_dwTlsIndex) == NULL) {
         // Space was never allocated. This is the first
         // time this function has ever been called by this thread.
         TlsSetValue(g_dwTlsIndex,
```

China-pub.com 下载

```
            HeapAlloc(GetProcessHeap(), 0, sizeof(*pSomeStruct));
        }

        // Memory already exists for the data;
        // save the newly passed values.
        memcpy(TlsGetValue(g_dwTlsIndex), pSomeStruct,
           sizeof(*pSomeStruct));

    } else {

        // The caller already primed the function. Now it
        // wants to do something with the saved data.

        // Get the address of the saved data.
        pSomeStruct = (PSOMESTRUCT) TlsGetValue(g_dwTlsIndex);

        // The saved data is pointed to by pSomeStruct; use it.
        .
        .
        .
    }
```

如果你的应用程序的线程从来不调用 MyFunction，那么也就从来不用为该线程分配内存块。

64个TLS位置看来超出了你的需要。但是请记住，应用程序可以动态地链接到若干个 DLL。第一个DLL可以分配10个TLS索引，第二个DLL可以分配5个TLS索引，依此类推。减少你需要的TLS索引始终是个好思路。减少所用 TLS索引的最好的办法，是采用前面的代码中的MyFunction使用的那种方法。当然，可以在多个TLS索引中保存全部40个字节，但是这样做不仅很浪费，而且使数据的操作很困难。相反，应该为数据分配一个内存块，并且像 MyFunction那样，只将指针保存在单个TLS索引中。正如前面讲过的那样，Windows 2000允许设置1000多个TLS时隙。Microsoft增加了时隙的数量，因为许多编程人员对时隙的使用采取一种只顾自己不顾其他的态度，不给其他DLL分配时隙，从而导致它们运行失败。

前面介绍TlsAlloc函数时，只描述了该函数能够实现的 99%的功能。为了帮助了解剩下的1%的功能，让我们观察一下下面的代码段：

```
DWORD dwTlsIndex;
PVOID pvSomeValue;

      .
      .
      .

dwTlsIndex = TlsAlloc();
TlsSetValue(dwTlsIndex, (PVOID) 12345);
TlsFree(dwTlsIndex);

// Assume that the dwTlsIndex value returned from
// this call to TlsAlloc is identical to the index
// returned by the earlier call to TlsAlloc.
dwTlsIndex = TlsAlloc();

pvSomeValue = TlsGetValue(dwTlsIndex);
```

在这个代码运行之后，你认为pvSomeValue将包含什么信息呢？包含12345？答案是包含0。TlsAlloc在返回之前，要遍历进程中的每个线程，在每个线程的数组中的新分配索引处放入 0。

China-pub.com
下载

这是很不错的，因为应用程序可能调用LoadLibrary来加载DLL，而DLL则调用TlsAlloc来分配索引，然后，线程调用FreeLibrary来删除DLL。DLL应该通过调用TlsFree来释放它的索引，但是谁知道DLL的代码将哪些值放入线程的数组中呢？接着，线程调用LoadLibrary，将另一个DLL加载到内存中。该DLL启动时也调用TlsAlloc，并获得与前面的DLL相同的索引。如果TlsAlloc没有为进程中的所有线程设置返回的索引，那么线程就可能看到一个老的值，而代码则无法正确地运行。

例如，这个新DLL可以查看对TlsGetValue函数的调用是否曾经为一个线程分配了内存，就像前面的代码段显示的那样。如果TlsAlloc没有删除每个线程的数组项目，那么第一个DLL的老数据仍然可以使用。如果线程调用MyFunction，那么MyFunction就会认为内存块已经分配，并且调用memcpy函数，将新数据拷贝到它所认为的内存块中。这可能造成灾难性的后果，不过，幸好TlsAlloc会对数组元素进行初始化，使这样的灾难永远不会发生。

21.2 静态TLS

与动态TLS一样，静态TLS也能够将数据与线程联系起来。但是，静态TLS在代码中使用起来要容易得多，因为不必调用任何函数就能够使用它。

比如说，你想要将起始时间与应用程序创建的每个线程联系起来。只需要将起始时间变量声明为下面的形式：

```
__declspec(thread) DWORD gt_dwStartTime = 0;
```

__declspec(thread)的前缀是Microsoft添加给Visual C++编译器的一个修改符。它告诉编译器，对应的变量应该放入可执行文件或DLL文件中它的自己的节中。__declspec(thread)后面的变量必须声明为函数中（或函数外）的一个全局变量或静态变量。不能声明一个类型为__declspec(thread)的局部变量。这不应该是个问题，因为局部变量总是与特定的线程相联系的。我将前缀gt_用于全局TLS变量，而将st_用于静态TLS变量。

当编译器对程序进行编译时，它将所有的TLS变量放入它们自己的节，这个节的名字是.tls。链接程序将来自所有对象模块的所有.tls节组合起来，形成结果的可执行文件或DLL文件中的一个大的.tls节。

为了使静态TLS能够运行，操作系统必须参与其操作。当你的应用程序加载到内存中时，系统要寻找你的可执行文件中的.tls节，并且动态地分配一个足够大的内存块，以便存放所有的静态TLS变量。你的应用程序中的代码每次引用其中的一个变量时，就要转换为已分配内存块中包含的一个内存位置。因此，编译器必须生成一些辅助代码来引用该静态TLS变量，这将使你的应用程序变得比较大而且运行的速度比较慢。在x86 CPU上，将为每次引用的静态TLS变量生成3个辅助机器指令。

如果在进程中创建了另一个线程，那么系统就要将它捕获并且自动分配另一个内存块，以便存放新线程的静态TLS变量。新线程只拥有对它自己的静态TLS变量的访问权，不能访问属于其他线程的TLS变量。

这就是静态TLS变量如何运行的基本情况。现在让我们来看一看DLL的情况。你的应用程序很可能要使用静态TLS变量，并且链接到也想使用静态TLS变量的一个DLL。当系统加载你的应用程序时，它首先要确定应用程序的.tls节的大小，并将这个值与你的应用程序链接的DLL中的任何.tls节的大小相加。当在你的进程中创建线程时，系统自动分配足够大的内存块来存放应用程序需要的所有TLS变量和所有隐含链接的DLL。

China-pub.com 下载

下面让我们来看一下当应用程序调用LoadLibrary，以便链接到也包含静态TLS变量的一个DLL时，将会发生什么情况。系统必须查看进程中已经存在的所有线程，并扩大它们的 TLS内存块，以便适应新DLL对内存的需求。另外，如果调用FreeLibrary来释放包含静态TLS变量的DLL，那么与进程中的每个线程相关的的内存块应该被压缩。

对于操作系统来说，这样的管理任务太重了。虽然系统允许包含静态 TLS变量的库在运行期进行显式加载，但是TLS数据没有进行相应的初始化。如果试图访问这些数据，就可能导致访问违规。这是使用静态TLS的唯一不足之处。当使用动态TLS时，不会出现这个问题。使用动态TLS的库可以在运行期进行加载，并且可以在运行期释放，根本不会产生任何问题。