漫谈兼容内核之二十:
关于 TLS

毛德操

TLS 是“线程局部存储(Thread Local Storage)”的缩写，“Local”这个词有“局部”、“本地”的意思，所以也可以说是“线程本地存储”。顾名思义，这就是局部于唯一的线程、为具体线程所“私用、专用”的存储空间。这里所说的“空间”并不是“用户空间”、“系统空间”那个意义上的空间，而是指用户空间中个别变量、数组、或数据结构所占据的存储空间。

我们知道，线程并不独立拥有用户空间，用户空间是归进程所有，为同一进程中的所有线程所共享的。所以，用户空间中的任何一个区域，只要有一个线程可以访问，那么同一进程中的所有其它的线程就都能访问。在这个意义上，整个用户空间都是(由同一进程中的)所有线程共享的，不存在只归一个线程使用的变量或数据结构。可是，一般而言，程序对变量或数据结构的访问都是按变量名访问的，经过编译/连接之后就是按地址访问，要是不知道一个变量的地址，实际上就无法正常和正确地加以访问(“地毯式”的扫描一般而言无法辨认数据的边界，所以无法正确读取其内容)。在这个意义上，则只归一个线程使用的变量或数据结构又是可能的。

注意 TLS 只是对全局量和静态变量才有意义。局部量存在于具体线程的堆栈上，而每个线程都有自己的堆栈，所以局部量本来就是“局部”于具体线程的。至于通过动态分配的缓冲区，则取决于保存着缓冲区指针的变量。如果缓冲区指针是全局量，那么同一进程中的所有线程都能访问这个缓冲区；而若是局部量，则别的线程自然就不得其门而入。

那么为什么需要有全局变量(或静态变量)的 TLS 呢？

对于 TLS 的用途，Unix/Linux 的 C 程序库 libc 中的全局变量 errno 是个最典型的例子。

```
if (open (filename, mode) < 0)
{
    error (0, errno, "%s", infile);
    . . . . . .
}
```

当系统调用从内核返回用户空间时，如果 EAX 的值为 0xfffff001 至 0xffffffff 之间即为出错，取其负值(2 的补码)就是出错代码。此时将出错代码写入一个全局量 errno，以供进一步查验，并将 EAX 的值改成-1。这就是从 Unix 时代初期就定下的对于返回整数的系统调用的约定。其好处是写程序时可以略微方便一些，不用每次都在函数中定义一个局部变量 err，再写成“if ((err=open(filename, mode)) < 0”、并因为系统调用出错返回的几率毕竟很小。

在线程的概念出现之前，或者说在一个进程只含有一个线程的时代，这样安排不会有什么问题。这是因为，从启动系统调用的 C 库函数把出错代码写入全局量 errno 以后，到调用者发现返回值为-1、因而从 errno 获取出错代码的这一段时间中 errno 的值不可能改变。即使在这中间发生了中断、并且导致进程调度，从当事进程的角度看也只是时间上的短暂停滞，却不会有谁来改变 errno 的内容；因为全局量 errno 是归进程所有，别的进程不会来打扰。

但是，有了多线程的概念和技术以后，情况就不一样了。因为同一进程中的多个线程是共享一个用户空间的，如果几个线程的程序中都引用全局量 errno，那么它们在运行中实际

访问的就是同一个地址。为说明问题，我们且假定线程 T1 和 T2 属于同一进程，再来考察下述的假想情景：

1. T1 先通过 C 库函数 open()进行系统调用，但是因为所给定的文件名实际上是个目录，所以内核返回出错代码 EISDIR。这个出错代码被写入了全局量 errno。
2. T1 发现 open()的返回值为-1，因而需要以 errno 当时的值、即出错代码为参数调用 error()；但是，在 T1 还没有来得及从 errno 读出之前就发生了中断，而且导致线程调度，调度的结果是线程 T2 获得运行；
3. T2 通过 C 库函数 signal()启动另一次系统调用，但是因为使用参数不当而出错，内核返回的出错代码为 EINVAL，表示参数无效。这个出错代码也被写入全局量 errno，于是 errno 的值变成了 EINVAL。
4. T2 从全局量 errno 读取出错代码，并依此进行相应的(正确)处理。但是 errno 的值仍保持为 EINVAL，此后 T2 没有再进行系统调用。
5. 一段时间之后，T1 又被调度运行，继续其原有的处理，即从 errno 读出、并以 errno 的值为参数调用 error()。但是，errno 的值原来是 EISDIR，而现在已变成 EINVAL。

由此可见，在多线程的环境下，对于某些应用，由多个线程共享一个同名的全局量是有问题的。

怎么解决呢？使用局部量当然是个办法，例如改成“if ((err=open(filename, mode)) < 0”，这似乎轻而易举地就解决了问题。可是，不幸的是，这就要求改变几乎所有 C 库函数的调用界面，而 C 库函数的调用界面早已成为标准。那么另外再定义一个新的 C 程序库怎么样？这也很成问题，新开发的软件固然可以改用新的界面，但是这么多已经存在的代码就都需要重新加以修改了。所以，这是不太现实的。

显然，就 errno 而言，比较好的解决方案是维持原有的代码和界面不变，但是让每个线程都有自己的 errno，或者说都有一个专用的 errno 副本。这就是“线程局部存储”、即 TLS。

于是，使用 TLS 就是自然的选择了。实际上，现在的 C 库函数不是把出错代码写入全局量 errno，而是通过一个函数__errno_location()获取一个地址，再把出错代码写入该地址，其意图就是让不同的线程使用不同的出错代码存储地点，这就是 TLS。而 errno，现在一般已经变成了一个宏定义：

```
#define errno (*__errno_location())
```

这样，原来代码中的“extern int errno;”就变成了“extern int (*__errno_location();”。应该说，这个方案还是挺巧妙的。

相比之下，Win32 API 的调用方式就不同了：

```
Status = NtOpenFile(…);
if(!NT_SUCCESS(Status))
{
}
```

当然，这里的 Status 是个局部变量，所以就不存在前述的问题了。这倒不是因为微软的人特别高明、早就预见到了可能发生的问题，而是因为时代不同，须知 Unix 的 C 库程序调用界面比 W32 API 的出现要早十多年。

不过，尽管 Windows 对于系统调用的出错代码这个具体的数据没有线程局部存储的要求，这并不意味着在 Windows 系统中就不需要 TLS。事实上，Windows 也需要 TLS，一些与 Windows 编程有关的文献、资料中对此都有叙述，这里就不重复了。

从实现的方法和形式看，TLS 有两种，一种是“静态 TLS”，另一种是“动态 TLS”。

静态 TLS 的实现依赖于编译工具和运行库，实际上涉及编程语言(例如 C 语言)的语法和语义上的扩充。具体的办法是：编程时在需要作为线程局部存储的 TLS 变量前面加上某种前缀，让编译工具知道这是需要线程局部存储的变量，从而生成出有所不同的汇编代码。例如，在微软的 VC++语言中：

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

这里的前缀__declspec(thread)在“正宗”的 C++语言中是没有的，表示全局量 something 是需要线程局部存储的，或者说是 TLS 变量。加以说明以后，在程序中可以像对待普通变量一样地读/写这个 TLS 变量，所以很是方便。在编译的时候，编译工具把这样的变量都分配到一个.tls 段中。而连接工具则把所有.o 模块的.tls 段都合并在一起，形成整个 EXE 或 DLL 可执行文件的.tls 段。可想而知，每创建一个线程，就要为其所在进程所涉及的所有(例如 EXE 和 DLL)可执行映像的.tls 段(如果有的话)另行分配一块空间。这样，进程中有几个线程，每个 TLS 变量就有几个副本。换言之，对于每个 TLS 变量，每个线程都有其本地的副本。

微软的 VC++是这样，GNU-C 也是如此，只是具体使用的前缀在形式上有所不同。

当然，这还只是事情的一个方面，另一个方面是程序中对这些 TLS 的引用。以前面的 errno 为例，原来直接就可以往这个变量中写，现在却要通过一个函数__errno_location()临时获取其本地副本的地址。总之，原先在连接时就把其所在的地址填写到有关的指令中，现在却要多绕一下，临时才间接地从有关的.tls 段获取其本地副本的地址了。这样，程序执行的效率当然有所下降；但是只要不是大量地反复访问 TLS 变量，就不成为问题。

对每一个 TLS 变量都生成一个类似于__errno_location()的函数未免太麻烦了，所以一般都是把同一模块中的所有 TLS 变量都组装在一个数据结构中，而只是把指向这个数据结构的指针作为 TLS 变量保存，但是每当创建一个线程时就要(通过库程序)为其分配空间并复制这个数据结构的原始副本。这样，需要访问组装在这个数据结构中的变量时就先找到属于当前线程的副本、再通过位移量在结构中找到目标变量，就好像 C 语言中的例如“tlsdata.errno”那样。这里的许多操作都要由库程序承担，所以不光是编译/连接的事，也需要库程序的配套。

原理虽然简单，具体处理起来却也可能很麻烦。试想，如果应用程序在运行中要求动态装入一个带有.tls 段的 DLL，就得扫描已经存在的所有的线程，扩大它们已有的.tls 段，并对新增加的部分进行初始化。而过一回儿若是又要卸载某个 DLL，则又得对每个线程的.tls 段加以调整，这就更麻烦了。

所以，静态 TLS 对于程序员而言固然很是方便，但是也有缺点。实际上，在 EXE 程序中使用静态 TLS 是合适的，在 DLL 中使用就显出缺点来了。

而动态 TLS 正好相反，动态 TLS 是通过一组库函数实现的，与编译没有关系。程序员必须按相应的界面通过这组函数分配、读/写、释放 TLS 变量，而不能直接加以读写，风格上接近于面向对象的程序设计。在 Windows 系统中，这组库函数是：

TlsAlloc() —— 为当前线程分配一个 32 位的动态 TLS 变量，返回一个索引号作为标识。

TlsSetValue() — 以索引号和一个 32 位数值为参数，将给定的数值写入由索引号所标识的动态 TLS 变量中。

TlsGetValue() — 以索引号为参数，读取(返回)由索引号所标识的动态 TLS 变量的值。

TlsFree() — 以索引号为参数，为当前线程释放由索引号所标识的 TLS。

注意由 TlsAlloc()分配的 TLS 变量一定是 32 位长字，可以作为整数使用，也可以作为指针使用。如果需要局部存储的是个数据结构或数组，那就要动态分配一个缓冲区，而把缓冲区指针作为 TLS 变量存储。这样，对于这数据结构中的成分或元素仍可像普通变量一样直接访问，因为这些数据本身并不是 TLS 变量。当然，这样的数据结构或数组存在于本进程公用的空间，在同一进程内所有线程的寻址范围之内，理论上这些线程都可以访问它们。但是指向这些数据的结构指针是 TLS 变量、是局部于具体线程的，别的线程取不到这指针，也就不能正常访问了。

动态 TLS 最适合在 DLL 中使用。

通常，如果一个 DLL 使用动态 TLS，就在以 DLL_PROCESS_ATTACH 为参数调用其 DllMain()的过程中调用 TlsAlloc()，而在以 DLL_PROCESS_DETACH 为参数调用其 DllMain()的过程中调用 TlsFree()。但是当然也可以在程序中随时分配和释放。

后面读者将会看到，静态 TLS 与动态 TLS 本质上是一样的，不同的只是形式。

静态 TLS 的实现

先看静态 TLS 是怎么实现的。

前面讲过，静态 TLS 需要编译工具的配合。但那只是为了尽可能地为使用者提供方便，并不是非有不可，事实上通过对宏定义的预处理也可以达到目的。再说，像 errno 这样的变量，已经广泛存在于已有的代码中，要在编译之前都先加上 TLS 前缀也不太现实。所以，除使用经过扩充的编译工具和 TLS 前缀之外，对于 errno 这些变量一般是在 C 库里面实现线程局部存储的，而并不依赖于编译工具和 TLS 前缀。而明白了 errno 的线程局部存储是如何实现的，就可以推而广之，不必深入到编译工具例如 gcc 的代码中就可以理解对静态 TLS 的实现机理了。

如前所述，errno 可以定义为(*__errno_location ())，这很简单。但是，从哪儿获取本线程的出错代码副本的地址呢？实际上，对于一个变量而言，取其值与取其指针的值并没有什么本质的不同，对于 TLS 变量也是一样。所以一个圈子又绕回了原点，我们仍然面临着相同的问题：怎样让执行着相同的代码、要访问同一个变量的不同线程实际访问的是这个变量的不同副本。

我们可以把一个进程的所有 TLS 变量全都组装在一个数据结构中，让每一个线程都有这个数据结构的一份副本，这并不困难，困难仍旧在于怎样让每个线程都有自己的指针，指向这个数据结构的不同副本。

所以，静态 TLS 的实现问题可以归结到仅仅一个指针的线程局部存储。有了这样一个指针，其余的就好办了。

怎么办呢？有下面这么几个思路：

一、跟堆栈挂勾。每个线程都有自己的堆栈，如果能从堆栈上划出一小片空间，就可以用来实现局部于具体线程的存储。

我们不妨以 Linux 内核中的 task_struct 数据结构为例来说明这个思路。在 Linux 内核中，形式上 current 是个全局的指针，总是指向当前线程的 task_struct 数据结构，例如 current->state 就是当前线程目前的状态。当然，每个线程都有自己的 task_struct 数据结构，里面都有 state

这个字段，但是所有的线程都执行同样的程序，使用同一个“指针”current。所以 state 以及 task_struct 结构中的其它字段就都相当于 TLS 变量，它们都组装在一个数据结构中，每个线程只要有自己版本的指针 current 就可以了。所以内核中 task_struct 数据结构的实现与用户空间 TLS 的实现是可以类比的，后者可以借鉴前者的实现。那么 current 是怎样实现的呢？看下面的代码：

```
#define   current   get_current()

static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
    return current;
 }
```

可见，current 其实是个宏操作，具体就是把当前的堆栈指针跟 0xffff2000 相与，把最低的 13 位屏蔽掉，所得到的就是当前线程的 task_struct 结构指针。这是因为：首先当前的堆栈一定是当前线程的(系统空间)堆栈；其次是每当创建一个线程时内核都为其分配与 8KB 边界对齐的两个连续页面，并把这线程的 task_struct 数据结构放在地址较低的那个页面中，其页面起点就是 task_struct 数据结构的起点，而地址较高的那个页面的顶端则就是堆栈的起点(向下伸展)。这样，不管当前的堆栈指针究竟指在什么位置上，只要将其最低的 13 位屏蔽掉就得到了指向当前线程的 task_struct 结构的指针。这里的关键在于，每个线程的 task_struct 结构与其堆栈是互相挂勾、互相绑定的。

当然，使二者挂勾的办法并不止这么一种，只要挂起勾来就行。显然，在用户空间也可以使用类似的方法，即与用户空间堆栈挂勾。这是办法之一。

二、除用户空间堆栈以外，Windows 用户空间的“线程环境块”TEB 也是具体线程所独有的，事实上 TEB 数据结构中的每一项都是 TLS。所以，与 TEB 挂勾也是办法。

三、另一个办法是使用段寄存器。在采用某个段寄存器加段内位移的方法寻址时，尽管位移相同，只要让不同线程有不同的段寄存器设置，就可以让形式上相同的地址(例如%fs:0x18)实际上指向不同的地方。这又是一种办法。

其实，这几种办法实质上是一样的。因为堆栈的位置是由堆栈段寄存器和堆栈指针共同确定的；而 TEB 的实现本来就是依赖于段寄存器的，每个线程的 TEB 上实际上就是一个段。所不同的只在于是否把段寄存器加位移的寻址方式转换成线性地址。

除此以外还有没有别的办法呢？比方说，要是通过系统调用可以获取当前线程的 TID，并且可以很方便地把 TID 转换成一个下标，用来访问一个指针数组，从中获得本线程的 TLS 结构指针，那倒也是个办法。Linux 现在有个系统调用 gettid()，但是就目前而言所返回的就是统一编号的 PID，而不是同一进程里面的线程序号，难于把它转换成数组下标。另一方面，即使这个系统调用果真能返回如 0、1、2、3 等 TID，因而可直接用作下标，效率也还是太低(每次都得系统调用)。所以，除了上述的与堆栈挂勾、与 TEB 挂勾、采用段寄存器这几种方法外，别的恐怕也找不到什么更好的方法了。

总之，TLS 的实现必须与某项由具体线程所独有的、唯一的资源挂勾。堆栈、TEB 就不必说了，在采用段寄存器的方案里，写入段寄存器的值就是具体线程所独有的，所以都满足这个条件。

事实上，在 libc 中，前述的函数__errno_location()的定义之一为：

```
int *__errno_location (void)
{
  pthread_descr self = thread_self();
  return THREAD_GETMEM (self, p_errnop);
}
```

说是“定义之一”，是因为 libc 在编译时有很多条件编译选项，选择不同的选项，其静态 TLS 的具体实现也就有所不同。

关键在于 thread_self()。据说人类的一个关键问题是认识自身，而这里的关键问题显然是让当前线程认识其自身。这个函数的定义之一是这样：

```
static inline pthread_descr thread_self (void)
{
  return (pthread_descr)((unsigned long)sp &~ (STACK_SIZE-1));
}
```

显然，这就是与用户空间堆栈挂勾。

再看动态 TLS 的实现，我们着重看 Windows、因而 ReactOS 的动态 TLS。

为实现动态 TLS，微软在 PEB 和 TEB 数据结构中都作了必要的安排。先看 PEB 数据结构中的有关定义：

```
typedef struct _PEB
{
    . . . . . .
    RTL_USER_PROCESS_PARAMETERS *ProcessParameters;   /* 10 */
    . . . . . .
    HANDLE                      ProcessHeap;          /* 18 */
    . . . . . .
    PRTL_BITMAP                 TlsBitmap;            /* 40 */
    ULONG                       TlsBitmapBits[2];     /* 44 */
    . . . . . .
    ULONG                       SessionId;            /* 1d4 */
} PEB, *PPEB;
```

显而易见，这里的指针 TlsBitmap 和数组 TlsBitmapBits[]就是为 TLS 而设的。其中的指针 TlsBitmap 指向一个 RTL_BITMAP 数据结构，其定义如下：

```
typedef struct _RTL_BITMAP
{
  ULONG SizeOfBitMap;
  PULONG Buffer;
```

} **RTL_BITMAP**, *PRTL_BITMAP;

显然，其目的是要提供一个位图，而真正的位图在 Buffer 所指的地方，一般这就是 PEB 中的 TlsBitmapBits[2]，但是有需要时也不排斥采用别的缓冲区，因为两个 32 位长字只能提供 64 个标志位。

PEB 是进程的资源，不是线程的资源，更不成为具体线程所独有的资源。所以这两个结构成分本身不能构成 TLS。如前所述，具体的 TLS 只能与堆栈、TEB 挂勾，或者就采用段寄存器。事实上，TEB 中为动态 TLS 的实现提供了手段。

```
typedef struct _TEB
{
    NT_TIB Tib;                             /* 00h */
    PVOID EnvironmentPointer;               /* 1Ch */
    . . . . . .
    PVOID ThreadLocalStoragePointer;        /* 02c */
    PPEB Peb;                               /* 30h */
    . . . . . .
    PVOID TlsSlots[0x40];                   /* E10h */
    LIST_ENTRY TlsLinks;                    /* F10h */
    . . . . . .
} TEB, *PTEB;
```

TlsSlots[]是个无类型指针数组，其大小为 0x40、即 64。这就是说，一个线程同时存在的动态 TLS 不能超过 64 项。另一个字段 ThreadLocalStoragePointer 用于静态 TLS，后面会讲到这个指针。

如果一项动态 TLS 数据的大小不超过 4 个字节，那么直接就可以存储在这个数组中，作为这个数组的一个元素。而若是大于 4 个字节的数据结构，那就需要为之动态分配存储缓冲区，而把缓冲区的地址存储在这个数组中。当然，这就大大扩充了动态 TLS 的容量。

需要创建一个动态 TLS 变量时，应用程序先通过 TlsAlloc()分配空间：

```
DWORD STDCALL
TlsAlloc(VOID)
{
    ULONG Index;

    RtlAcquirePebLock();
    Index = RtlFindClearBitsAndSet (NtCurrentPeb()->TlsBitmap, 1, 0);
    if (Index == (ULONG)-1)
    {
        SetLastErrorByStatus(STATUS_NO_MEMORY);
    }
    else
    {
        NtCurrentTeb()->TlsSlots[Index] = 0;
```

```
    }
    RtlReleasePebLock();
    return(Index);
}
```

这里的操作很简单，先通过 RtlFindClearBitsAndSet()检查所属进程的 PEB 中的位图，从中找到一个为 0 的标志位、即空闲的 TLS 变量，并把它设置成 1，表示已经占用，然后就根据标志位所在的位置推算出相应的下标。而当前线程 TEB 中数组 TlsSlots[]的相应元素就是所分配的动态 TLS 变量，这里把它初始化成 0。

关键是 NtCurrentTeb()的实现，即如何找到当前线程的 TEB。这个 inline 函数的代码为：

```
static __inline__ struct _TEB * NtCurrentTeb(void)
{
    struct _TEB *ret;

    __asm__ __volatile__ (
        "movl %%fs:0x18, %0\n"
        : "=r" (ret)
        : /* no inputs */
    );
    return ret;
}
```

当 CPU 运行于用户空间时，段寄存器总是(通过相应的段描述项)指向当前线程的 TEB，TEB 的起点就是段的起点。TEB 中位移为 0x18 处是个指针 Self，指向其所在 TEB 的起点，所以这里的%%fs:0x18 就是当前 TEB 的起点(注意%%fs:0 是 TEB 起点处的内容，但是不能以&(%%fs:0)取起点的地址，也不存在可以直接达到这个目的的指令)。

由 TlsAlloc()返回的索引号、即下标、代表着一个动态 TLS 变量，一般都是保存在一个全局量或静态变量中，为同一进程中的所有线程所共见和共用。但是不同线程在使用这个共同下标时访问的数组是不同的。当然，若要将下标保存在一个局部变量中也并无不可，但是那样就浪费了一个 TLS 变量，因为局部变量本来就已经是局部的、不能为其它线程所见。

需要读/写这个动态 TLS 变量的时候，就使用这个下标。作为 TlsSetValue()或 TlsGetValue()的参数之一，其作用就像是目标 TLS 变量的地址。于是，不同的线程使用相同的地址，访问的却是不同的变量、即局部于不同线程的变量。

明白了这些，函数 TlsSetValue()的代码就是直截了当的了：

```
BOOL STDCALL
TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue)
{
    if (dwTlsIndex >= TLS_MINIMUM_AVAILABLE)
    {
      SetLastErrorByStatus(STATUS_INVALID_PARAMETER);
      return(FALSE);
    }
```

```
    NtCurrentTeb()->TlsSlots[dwTlsIndex] = lpTlsValue;
    return(TRUE);
}
```

明白了 TlsAlloc()和 TlsSetValue()的代码，TlsGetValue()和 TlsFree()的代码就更是不值一提。可见，动态 TLS 的实现相当简洁。

值得一提的倒是 TLS 的初始化。

TLS 的初始化可以分成两部分，即静态 TLS 的初始化和动态 TLS 的初始化。可想而知，动态 TLS 的初始化是很简单的，而静态 TLS 的初始化则比较复杂。

我们从__true_LdrInitializeThunk()开始看 TLS 的初始化，因为新创建的线程都是从这里开始执行的。特别地，我们从创建进程时主线程、即第一个线程的初始化开始。以前在介绍装入/执行 PE 可执行文件和创建进程/线程的过程时讲解过这个函数，但那时还没有到要关心 TLS 的地步，所以当时都把与 TLS 有关的代码跳过了。

```
VOID STDCALL
__true_LdrInitializeThunk (ULONG Unknown1, ULONG Unknown2,
                           ULONG Unknown3, ULONG Unknown4)
{
    . . . . . .

    DPRINT("LdrInitializeThunk()\n");
    if (NtCurrentPeb()->Ldr == NULL || NtCurrentPeb()->Ldr->Initialized == FALSE)
    {
        Peb = (PPEB)(PEB_BASE);
        DPRINT("Peb %x\n", Peb);
        ImageBase = Peb->ImageBaseAddress;
        . . . . . .
        . . . . . .
        /* initialize tls bitmap */
        RtlInitializeBitMap (&TlsBitMap, Peb->TlsBitmapBits,
                             TLS_MINIMUM_AVAILABLE);
        Peb->TlsBitmap = &TlsBitMap;
        Peb->TlsExpansionCounter = TLS_MINIMUM_AVAILABLE;
        . . . . . .
        NtModule->TlsIndex = -1;
        . . . . . .
        EntryPoint = LdrPEStartup((PVOID)ImageBase, NULL, NULL, NULL);
        ExeModule->EntryPoint = (ULONG)EntryPoint;
        . . . . . .
    }
    /* attach the thread */
    RtlEnterCriticalSection(NtCurrentPeb()->LoaderLock);
    LdrpAttachThread();
```

```
    RtlLeaveCriticalSection(NtCurrentPeb()->LoaderLock);
}
```

进程中的第一个线程需要执行 if 语句中的内容，以后的线程就跳过这一部分，只是执行后面的 LdrpAttachThread()了。

RtlInitializeBitMap()的作用就是建立本进程的动态 TLS，即 TLS 位图。这里的 TlsBitMap 是个 RTL_BITMAP 数据结构，但是这个数据结构里面并不包括实际用作位图的缓冲区，这缓冲区就是 Peb->TlsBitmapBits，位图的大小则是 TLS_MINIMUM_AVAILABLE，即 64。设置好 TlsBitMap 以后，将其指针填写到 PEB 中，并相应设置 PEB 的其它几个有关的字段，本进程的动态 TLS 机制就建立起来了，可见动态 TLS 的初始化是很简单的。

下面是 LdrPEStartup()，这就涉及静态 TLS 的初始化了。先要介绍一下调用这个函数的背景。此时目标 EXE 映像和 ntdll.dll 的映像均已装入，现在 CPU 是在 ntdll.dll 的映像中运行。本进程 PEB 中的指针 Ldr 指向一个 PEB_LDR_DATA 数据结构，这个数据结构中有个“已装入模块”队列 InLoadOrderModuleList，里面是代表着已装入模块的 LDR_MODULE 数据结构，此刻这个队列中已经有了两个模块。

[__true_LdrInitializeThunk() > LdrPEStartup()]

```
PEPFUNC LdrPEStartup (PVOID ImageBase, HANDLE SectionHandle,
                      PLDR_MODULE* Module, PWSTR FullDosName)
{
    . . . . . .
    .PLDR_MODULE tmpModule;

    . . . . . .
    if (Module != NULL)
    {
        *Module = LdrAddModuleEntry(ImageBase, NTHeaders, FullDosName);
        (*Module)->SectionHandle = SectionHandle;
    }
    else
    {
        Module = &tmpModule;
        Status = LdrFindEntryForAddress(ImageBase, Module);
        if (!NT_SUCCESS(Status))
        {
            return NULL;
        }
    }
    . . . . . .

    /*
     * If the DLL's imports symbols from other modules, fixup the imported calls entry points.
     */
```

```
    DPRINT("About to fixup imports\n");
    Status = LdrFixupImports(NULL, *Module);
    . . . . . .
    Status = LdrpInitializeTlsForProccess();
    if (NT_SUCCESS(Status))
    {
       Status = LdrpAttachProcess();
    }
    . . . . . .

    /* Compute the DLL's entry point's address. */
    . . . . . .
    return EntryPoint;
}
```

调用参数 ImageBase 指向 EXE 映像装入本进程用户空间后的地址。

回过去看一下从__true_LdrInitializeThunk()里面调用这个函数时的实际参数，就可知道此时除 ImageBase 以外的其余三个参数、包括 Module、都是 NULL。所以，这里使用一个临时的 LDR_MODULE 结构指针 tmpModule，通过 LdrFindEntryForAddress()从“已装入模块队列”中根据地址 ImageBase 找到其所属的模块，并返回其 LDR_MODULE 结构指针。当然，这就是 EXE 模块。然后就以此为参数调用 LdrFixupImports()，从 EXE 模块开始，通过有可能是递归的 LdrFixupImports()调用，完成对所有 DLL 模块的装入和动态连接(只有 ntdll.dll 已经装入)。而在连接的过程中，就可能要涉及其.tls 段了。

注意这里调用 LdrFixupImports()时的参数是*Module，而不是 Module。Module 的类型定义是“PLDR_MODULE* Module”，所以是双重指针，但是 tmpModule 的类型却是 LDR_MODULE 指针。

下面就是从 EXE 模块开始执行 LdrFixupImports()的过程了：

[__true_LdrInitializeThunk() > LdrPEStartup() > LdrFixupImports()]

```
static NTSTATUS
LdrFixupImports(IN PWSTR SearchPath OPTIONAL, IN PLDR_MODULE Module)
{
    . . . . . .

    /* Check for tls data */
    TlsDirectory = (PIMAGE_TLS_DIRECTORY)
                        RtlImageDirectoryEntryToData(Module->BaseAddress, TRUE,
                                                     IMAGE_DIRECTORY_ENTRY_TLS, NULL);
    if (TlsDirectory)
    {
        TlsSize = TlsDirectory->EndAddressOfRawData
                        - TlsDirectory->StartAddressOfRawData
                        + TlsDirectory->SizeOfZeroFill;
```

```
        if (TlsSize > 0 && NtCurrentPeb()->Ldr->Initialized)
        {
            TRACE_LDR("Trying to load dynamicly %wZ which contains a tls directory\n",
                      &Module->BaseDllName);
            return STATUS_UNSUCCESSFUL;
        }
    }

    . . . . . .

    if (TlsDirectory && TlsSize > 0)
    {
        LdrpAcquireTlsSlot(Module, TlsSize, FALSE);
    }
    return STATUS_SUCCESS;
}
```

LdrFixupImports()的主要作用固然是完成对 DLL 的装入和动态连接(可能需要递归)，却同时也兼顾了对于静态 TLS 的初始化处理。这里一开始就检查当前模块(在这一层是 EXE 模块)的映像中是否有.tls 段、即类型为 IMAGE_DIRECTORY_ENTRY_TLS 的段。如果有的话，就根据映像头部所提供的信息计算出.tls 段的长度。当然，我们在这里只关心.tls 段长度非 0 的情况。

有意思的是，如果(整个进程的)初始化已经完成，此时就提示“试图动态装入含有 TLS 目录的 XX 模块”并出错返回。注意这里讲的是动态装入、而不是动态连接。DLL 模块一般是在创建进程、启动执行 EXE 映像的时候装入的，那是静态装入；但是也可以在运行的过程中根据需要随时装入，那就是动态装入。不管是静态装入还是动态装入都需要动态连接，所以都需要调用 LdrFixupImports()。前面讲过，把动态装入的模块的.tls 段合并到已在运行中的进程的 TLS 机制中是个比较麻烦的事。显然，至少 ReactOS 目前尚未实现此项功能；Windows 怎么样不好说，估计也是一样。另一方面，这也解释了为什么有关的资料都鼓励在 DLL 中使用动态 TLS、而不鼓励使用静态 TLS。

对于静态装入的 DLL，则后面通过 LdrpAcquireTlsSlot()为其分配一个非负的索引序号 TlsIndex，并将其.tls 段的大小纳入统计：

[__true_LdrInitializeThunk() > LdrPEStartup() > LdrFixupImports() > LdrpAcquireTlsSlot()]

```
static inline
VOID LdrpAcquireTlsSlot(PLDR_MODULE Module, ULONG Size, BOOLEAN Locked)
{
    if (!Locked)
    {
        RtlEnterCriticalSection (NtCurrentPeb()->LoaderLock);
    }
    Module->TlsIndex = (SHORT)LdrpTlsCount;
    LdrpTlsCount++;
```

```
    LdrpTlsSize += Size;
    if (!Locked)
    {
        RtlLeaveCriticalSection(NtCurrentPeb()->LoaderLock);
    }
}
```

显然，LdrpTlsCount 表示.tls 段的个数，而 LdrpTlsSize 表示累计的大小，将来要根据这两项数据为本进程建立静态 TLS。

对于 LdrFixupImports()的调用有可能(更确切地说是必然)是递归的，对于启动执行目标 EXE 映像所涉及的每个模块都会调用一次，所以最后就把所有.tls 段的大小都统计进来了。当 CPU 从最外层针对 EXE 模块的 LdrFixupImports()返回时，LdrpTlsSize 中已经有了总计的 TLS 大小。

回到 LdrPEStartup()的代码，下一步是对于 LdrpInitializeTlsForProccess()的调用。

[__true_LdrInitializeThunk() > LdrPEStartup() > LdrpInitializeTlsForProccess()]

```
static NTSTATUS
LdrpInitializeTlsForProccess(VOID)
{
   PLIST_ENTRY ModuleListHead;
   PLIST_ENTRY Entry;
   PLDR_MODULE Module;
   PIMAGE_TLS_DIRECTORY TlsDirectory;
   PTLS_DATA TlsData;

   if (LdrpTlsCount > 0)
   {
      LdrpTlsArray = RtlAllocateHeap(RtlGetProcessHeap(),0,
                                                LdrpTlsCount * sizeof(TLS_DATA));
      if (LdrpTlsArray == NULL)
      {
            return STATUS_NO_MEMORY;
      }
      ModuleListHead = &NtCurrentPeb()->Ldr->InLoadOrderModuleList;
      Entry = ModuleListHead->Flink;
      while (Entry != ModuleListHead)
      {
         Module =
                  CONTAINING_RECORD(Entry, LDR_MODULE, InLoadOrderModuleList);
         if (Module->LoadCount == -1 && Module->TlsIndex >= 0)
         {
            TlsDirectory = (PIMAGE_TLS_DIRECTORY)
                        RtlImageDirectoryEntryToData(Module->BaseAddress, TRUE,
```

```
                                                IMAGE_DIRECTORY_ENTRY_TLS, NULL);
            assert(Module->TlsIndex < LdrpTlsCount);
            TlsData = &LdrpTlsArray[Module->TlsIndex];
            TlsData->StartAddressOfRawData = (PVOID)TlsDirectory->StartAddressOfRawData;
            TlsData->TlsDataSize = TlsDirectory->EndAddressOfRawData -
                                                    TlsDirectory->StartAddressOfRawData;
            TlsData->TlsZeroSize = TlsDirectory->SizeOfZeroFill;
            if (TlsDirectory->AddressOfCallBacks)
              TlsData->TlsAddressOfCallBacks = *TlsDirectory->AddressOfCallBacks;
            else
              TlsData->TlsAddressOfCallBacks = NULL;
            TlsData->Module = Module;
            /* FIXME: Is this region allways writable ? */
            *(PULONG)TlsDirectory->AddressOfIndex = Module->TlsIndex;
            CHECKPOINT1;
          }
          Entry = Entry->Flink;
        }
      }
      return STATUS_SUCCESS;
}
```

可想而知，对于把所有.tls 段合并在一起的静态 TLS，每个线程都需要有个副本；但是线程是随时都可以创建的，所以必须把来自各模块映像.tls 段的原始映像合并保存起来。然而各个.tls 段既然分别存在于已被装入的模块映像中，而这些.tls 段又不属于任何具体的线程，在运行不会被修改，就无需再单独存储一份合并以后的版本，但是需要有个合并的目录，以便需要时可以方便地找到这些原始的版本。这正是 LdrpInitializeTlsForProccess()要做的。

所以，为实现静态 TLS，每个进程的用户空间需要一个类似于目录的指针数组 LdrpTlsArray[]，其中的每个指针都指向一个 TLS_DATA 数据结构，而每个 TLS_DATA 数据结构则代表着一个模块的.tls 段。

```
typedef struct _TLS_DATA
{
    PVOID StartAddressOfRawData;
    DWORD TlsDataSize;
    DWORD TlsZeroSize;
    PIMAGE_TLS_CALLBACK TlsAddressOfCallBacks;
    PLDR_MODULE Module;
} TLS_DATA, *PTLS_DATA;
```

这个数据结构本身并不提供实际的 TLS，其指针 StartAddressOfRawData 才指向实际的存储空间、即各个模块的.tls 段。每个.tls 段的开头可以有一些非 0 初值的 TLS 变量，TlsDataSize 说明了这一部分的大小。其余则都是初值为 0 的 TLS 变量，字段 TlsZeroSize 说明了这一部分的大小。此外，含有.tls 段的模块可能同时提供一些用于初始化或需要在特定

条件下执行的“回调”函数，指针 TlsAddressOfCallBacks 指向一个用来提供有关函数指针的 IMAGE_TLS_CALLBACK 数据结构。至于指针 Module，则只是说明这个.tls 段来自哪一个模块，同时也便于从具体模块的数据结构中获取更多、更详尽的有关信息。例如，其中的 FullDllName 就提供了该模块的映像文件名。

上面的程序中根据.tls 段的个数 LdrpTlsCount 分配了数组 LdrpTlsArray[]所需的空间。注意 LdrpTlsCount 只反映了静态装入的.tls 段的个数，而动态装入的模块到底会有多少是无法预测的。

然后扫描本进程的“已装入模块队列”、即 PEB 下面的 InLoadOrderModuleList，为每个模块的.tls 段(如果有的话)建立 LdrpTlsArray[]中的目录项。注意此时所有静态装入的模块都已经在这个队列中，所以这是一次完全的扫描。另一方面，凡是带有.tls 段、或者说.tls 段的大小非 0 的模块，前面都已经为其分配了 TLS 索引号 TlsIndex，所以在目录中的位置也已经确定。

从程序中可以看出，具体的原始 TLS 映像仍在各模块的内存映像中，但是现在有了一个统一的目录 LdrpTlsArray[]。

再回到 LdrPEStartup()的代码，下面的 LdrpAttachProcess()主要是以常数 DLL_PROCESS_ATTACH 为参数调用各个 DLL 模块的 DllMain()，与静态 TLS 的实现没有什么关系。而 LdrpTlsCallback()则是以 DLL_PROCESS_ATTACH 为参数调用各模块的 TLS 回调函数(如果有的话)，我们在这里就不深入下去了。

这些操作都是进程一级的，所以只是进程中的第一个线程才加以执行，以后创建的线程就不再执行这些操作了。

从 LdrPEStartup()返回到__true_LdrInitializeThunk()，在完成了所有别的操作以后，对于静态 TLS 而言，就是线程一级的操作了，这是每个线程都要执行的，目的在于为当前线程复制一个本地的静态 TLS 副本。这是在 LdrpAttachThread()里面完成的：

[__true_LdrInitializeThunk() > LdrpAttachThread()]

```
NTSTATUS
LdrpAttachThread (VOID)
{
   PLIST_ENTRY ModuleListHead;
   PLIST_ENTRY Entry;
   PLDR_MODULE Module;
   NTSTATUS Status;

   DPRINT("LdrpAttachThread() called for %wZ\n",
            &ExeModule->BaseDllName);

   RtlEnterCriticalSection (NtCurrentPeb()->LoaderLock);

   Status = LdrpInitializeTlsForThread();

   if (NT_SUCCESS(Status))
      {
         ModuleListHead = &NtCurrentPeb()->Ldr->InInitializationOrderModuleList;
```

```
        Entry = ModuleListHead->Flink;

        while (Entry != ModuleListHead)
          {
            Module = CONTAINING_RECORD
                                (Entry, LDR_MODULE, InInitializationOrderModuleList);
            if (Module->Flags & PROCESS_ATTACH_CALLED &&
                !(Module->Flags & DONT_CALL_FOR_THREAD) &&
                !(Module->Flags & UNLOAD_IN_PROGRESS))
              {
                TRACE_LDR("%wZ - Calling entry point at %x for thread attaching\n",
                          &Module->BaseDllName, Module->EntryPoint);
                LdrpCallDllEntry(Module, DLL_THREAD_ATTACH, NULL);
              }
            Entry = Entry->Flink;
          }

        Entry = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink;
        Module = CONTAINING_RECORD(Entry, LDR_MODULE, InLoadOrderModuleList);
        LdrpTlsCallback(Module, DLL_THREAD_ATTACH);
      }

    RtlLeaveCriticalSection (NtCurrentPeb()->LoaderLock);

    DPRINT("LdrpAttachThread() done\n");

    return Status;
}
```

显然，这里 LdrpInitializeTlsForThread()的作用就是为当前线程复制一个“本地”的 TLS 副本：

[__true_LdrInitializeThunk() > LdrpAttachThread() > LdrpInitializeTlsForThread()]

```
static NTSTATUS
LdrpInitializeTlsForThread(VOID)
{
    PVOID* TlsPointers;
    PTLS_DATA TlsInfo;
    PVOID TlsData;
    ULONG i;

    DPRINT("LdrpInitializeTlsForThread() called for %wZ\n", &ExeModule->BaseDllName);
```

```
    if (LdrpTlsCount > 0)
    {
        TlsPointers = RtlAllocateHeap(RtlGetProcessHeap(),0,
                                      LdrpTlsCount * sizeof(PVOID) + LdrpTlsSize);
        if (TlsPointers == NULL)
        {
            DPRINT1("failed to allocate thread tls data\n");
            return STATUS_NO_MEMORY;
        }

        TlsData = (PVOID)TlsPointers + LdrpTlsCount * sizeof(PVOID);
        NtCurrentTeb()->ThreadLocalStoragePointer = TlsPointers;

        TlsInfo = LdrpTlsArray;
        for (i = 0; i < LdrpTlsCount; i++, TlsInfo++)
        {
            TRACE_LDR("Initialize tls data for %wZ\n", &TlsInfo->Module->BaseDllName);
            TlsPointers[i] = TlsData;
            if (TlsInfo->TlsDataSize)
            {
                memcpy(TlsData, TlsInfo->StartAddressOfRawData, TlsInfo->TlsDataSize);
                TlsData += TlsInfo->TlsDataSize;
            }
            if (TlsInfo->TlsZeroSize)
            {
                memset(TlsData, 0, TlsInfo->TlsZeroSize);
                TlsData += TlsInfo->TlsZeroSize;
            }
        }
    }
    DPRINT("LdrpInitializeTlsForThread() done\n");
    return STATUS_SUCCESS;
}
```

注意这里分配的空间大小是 LdrpTlsCount * sizeof(PVOID) + LdrpTlsSize，这是一个大小为 LdrpTlsCount 的指针数组加上合并以后的整个静态 TLS 的大小。然后就根据静态 TLS 目录把所有.tls 段的原始副本收集汇总并复制到所分配的缓冲区中。这样，就为一个线程构建了一个合并的静态 TLS 副本。至于指针数组 TlsPointers[ ]中的每个指针，则各自指向相应.tls 段的数据在这个副本中的起点。

还要注意，指针 TlsPointers 填写在当前线程 TEB 的字段 ThreadLocalStoragePointer 中。所以，找到了一个线程的 TEB，就可以根据这个指针找到其静态 TLS 副本。

由于每个线程在启动时都要执行__true_LdrInitializeThunk()，因而都要执行这个函数，所以就都会有各自的静态 TLS 副本。

回到 LdrpAttachThread()，后面以 DLL_THREAD_ATTACH 为参数对各个模块调用

LdrpCallDllEntry()和 LdrpCallDllEntry()，这里就不看了。

那么，在应用程序中怎样访问静态 TLS 变量呢？如前所述，这需要编译工具和库程序的配合。可以想像，库程序应提供一个类似于__errno_location()那样的函数，但是以目标变量所在模块的索引号和目标变量在其.tls 段内部的位移为参数。这样，只要先找到当前线程的 TEB，就可以找到其静态 TLS 副本。然后，以具体 TLS 变量所在模块的索引号为下标可以通过其指针数组找到目标所在的.tls 段副本，再用目标 TLS 变量在.tls 段内的位移就可以找到其地址。

可见静态 TLS 其实一点不比动态 TLS 简单，反倒复杂得多。

早期的程序员也许觉得使用动态 TLS 需要在程序中通过 TlsSetValue()、TlsGetValue()一类的函数才能访问 TLS 变量，不如使用静态 TLS 那样方便；但是现在大家都已经习惯了面向对象的程序设计，而面向对象的程序设计本来就得通过对象所提供的“方法(method)”进行访问，因此也就不感到不方便了。

作为一个例子，我们再看一下 errno。本来，W32 API 的界面并不需要使用 errno，也不存在为此而需要使用 TLS 的问题；但是微软为了实现对 POSIX 子系统的支持而只好也来实现一个标准 C 程序库的界面，因此就也有了如何实现 errno 的问题，下面的代码仍取自 ReactOS：

```
#define errno (*__PdxGetThreadErrNum())
```

显然，__PdxGetThreadErrNum()相当于__errno_location()。再看它是如何实现的：

```
int * __PdxGetThreadErrNum(void)
{
 return &(((__PPDX_TDATA)
          (NtCurrentTeb()->TlsSlots[__PdxGetProcessData()->TlsIndex]) )->ErrNum);
}
```

可见，这实际上是在用动态 TLS 来实现静态 TLS，因为 TEB 中的 TlsSlots[]是用于动态 TLS 的。

最后，表面上 TLS 的实现都是用户空间的事，似乎与内核无关；但是实际上内核为用户空间 TLS 的实现提供了基础。这是因为，无论是与 TEB 挂勾，还是与用户空间堆栈挂勾，还是使用段寄存器寻址，实际上都是建立在分段存储的基础上；但实际上正是内核在维持段描述表 GDT 或 LDT 中段描述项的切换，并使段寄存器 FS 或 GS 在 CPU 进入用户空间时持有相应的选择项。

此外，在系统调用 NtSetInformationThread()中有两个选项是与 TLS 有关的，一个是 ThreadZeroTlsCell，另一个是 ThreadSetTlsArrayAddress。前者的作用据说是将一个(大概是静态)TLS 变量清 0。后者则是为“TLS 数组”、应该是当事线程的静态 TLS 副本吧、指定一个地址。为此，在 KTHREAD 数据结构中还设了一个字段 TlsArray，但是在 ReactOS 的代码中未见使用。