漫谈兼容内核之二十二：
Windows 线程的调度和运行

毛德操

了解 Windows 线程的系统空间堆栈以后，还有必要对 Windows 线程的调度、切换、和运行也有所了解。当然，就兼容内核的开发而言，内核的线程调度/切换/运行机制只能有一套，而且必定是基本上沿用 Linux 的这套机制，而不可能在一个内核中有两套调度/运行机制。但是对于 Windows 这套机制的了解对于兼容内核的开发也很重要，并且还是必须的。举例来说，大家都知道在 Windows 系统中段寄存器 FS 在用户空间指向 TEB、而在系统空间则指向 KPCR，而且 Windows 的 DDK 中也公开了 KPCR 数据结构的定义。这样，设备驱动模块的开发者就有可能在程序中通过段寄存器 FS 获取 KPCR 的地址，并按 KPCR 数据结构的定义访问其中的某些字段。然而如果内核中并不真的有个 KPCR，或者 FS 并不指向 KPCR，那就要乱套了。所以，为了在兼容内核中支持设备驱动界面，就得把 KPCR 等等揉合进去，而那些东西其实就是 Windows 的线程切换/运行机制的一部分。

为此，我们先要了解一下 Windows 内核中有关这一方面的格局，这要从 x86 的系统结构谈起。

所谓“Intel 架构”、即 x86 的系统结构，其最初的设计是在二十多年以前。当初一来是还没有“简约指令集”即 RISC 的概念，二来是把操作系统的设计与实现考虑得太复杂、太繁琐，因而把 CPU 系统结构的设计与实现也考虑得太复杂、太繁琐了。就我们所关心的问题而言，这主要表现在两个方面。

首先，Intel 在 x86 的系统结构中把 CPU 的执行权限分得很细，分成了从 0 环至 3 环共 4 个“环”，并让 CPU 运行于 0 环时具有最高的权限，而运行于 3 环时则权限最低。但是从后来的发展看，无论是 Linux 还是 Windows，实际上都只分系统(即内核)和用户两种状态、或称两个空间就够了，因而只使用了 4 个环中的两个，即 0 环(内核)和 3 环(用户)。

另一方面，Intel 在“任务”(当初的“任务”相当于进程，现在则相当于线程)切换上也动了很多脑筋，其设计意图是让每个任务、即进程或线程、都有一个独立的“任务状态段”TSS，里面包含了几乎所有寄存器的映像，而通过 TSS 的切换来实现任务的切换，而且只要一条指令就能完成这样的切换。这条指令把几乎所有寄存器(除一些“系统寄存器”如 GDTR 等以外)的当前内容都一下子保存到当前任务的 TSS 中；然后通过一个实质上相当于段寄存器的“任务寄存器”TR 切换到目标任务的 TSS，就是使 TR 改而指向目标任务的 TSS；再从这 TSS 中恢复目标任务的寄存器映像，切换就完成了。这整个过程都集成在一条指令中，从程序上看是一步到位。而这功能如此强大的指令，则实际上既可以是 call 指令、也可以是 jmp 指令，还可以是 ret 指令或是中断的发生(由此可见本来应该很简单的 call 指令、jmp 指令，还有 ret 指令的实现变得多么复杂)。

TSS 中还有关于一个任务的其它重要信息，包括从 0 环到 2 环共三个环的堆栈段寄存器和堆栈指针的映像。其设计意图是，当 CPU 从外环(例如 3 环)进入内环(例如 0 环)时，就从当前任务的 TSS 中把内环的堆栈指针(以及段寄存器 SS 的映像)装入 ESP(以及 SS)。此外，TSS 中还有一个“I/O 权限位图”，位图中的每一位都代表着 I/O 地址空间(共 64KB)的一个字节，如果为 0 就表示即使在 3 环中也可以对此字节执行 in、out 等 I/O 指令。

Intel 的这些设计意图都实现了。可是论者却认为这样做真是得不偿失，因为这使 call、jmp、ret 等指令的设计与实现都大大复杂化了，还使指令流水线的设计与实现也大大复杂化了，并且 CPU 芯片上的许多资源都被用来实现这些并非必须的功能。再说，虽然是单指令

“一步到位”，但是这指令的执行时间却大大延长了，实际上也没有带来明显的好处。事实上，jmp 指令在实现任务切换时需要 200 多个时钟周期。有 200 多个周期，在流水线操作的条件下，通过程序和堆栈实现任务切换也差不多了。因此，他们认为 Intel 这是把本来可以简单的事情不必要而且不值得地复杂化了。不过这只是 RISC 拥护者的看法。可是，更有甚者，对于 Intel 如此良苦的用心，本该从中获益的操作系统设计人员竟也不领情。无论是 Linux 还是 Windows，线程的切换都没有使用 Intel 提供的这种单指令手段，而都是通过程序与堆栈实现的，与 TSS 几乎没有多少关系。

但是，尽管没有采用基于 TSS 的线程切换手段，线程切换却离不了 TSS。这主要是因为 CPU 在从用户空间进入系统空间时自动到 TSS 中去获取系统空间的堆栈指针。此外，如果在用户空间执行 in、out 等 I/O 指令，CPU 也要到 TSS 中去核对 I/O 权限位图。所以，在切换线程的时候，TSS 不一定要切换，但是里面的 ESP0 等字段却还是必须要改变，因为不同线程的系统空间堆栈的位置各不相同。至于 I/O 权限位图，也有可能需要改变。可是，在 Intel 的设计中，这毕竟只是 TSS 的作用和功能的很小一部分，所以颇有些“买椟还珠”的意味。

在 Intel 架构中，段寄存器起着重要的作用。在 16 位的“实模式”中段寄存器起着扩大寻址范围和保护的作用，段寄存器的内容为“段地址”、即一个段的起点，每个段的最大长度为 64KB，而整个地址则由段地址和段内位移整合而成。但是，在 32 位保护模式中段寄存器的作用已经改变，变成以保护为主了。此时段寄存器的内容已不再直接与地址有关，而变成了“段选择项”，其主体是用于“段描述表”的下标。下标不同，就选择了描述表中不同的表项，每个表项就是一个“段描述项”，至于段的长度则往往可以覆盖整个 4GB 空间。就线程切换而言，与其密切相关的“段描述表”有两个。一个是“全局描述表(Global Descriptor Table)”GDT，一个是“局部描述表(Local Descriptor Table)”。段描述项可以是针对 GDT 的，也可以是针对 LDT 的。其中 LDT 的设计意图是局部于个别的进程(任务)，并且其本身也是作为一个段而存在的。所以作为“根”的段描述表就是 GDT，而“GDT”中的字符 G 也可以理解为“General”，所以 GDT 就是“总描述表”。CPU 中有个寄存器 GDTR，其内容就是 GDT 起点的 32 位地址。GDT 的最大长度是 64K 字节，最多可以容纳 8192 个描述项(每个描述项 8 个字节)，其中的第一个描述项必须是 0，表示“非法描述项”(所以段选择项不能为 0)。

LDT 和 TSS 都是作为段而存在的(但 GDT 不是)，LDT 可用可不用，但 TSS 是非有不可的；所以 GDT 中必须有 TSS 的描述项，可能还有 LDT 的描述项。既然 LDT 和 TSS 都是作为段而存在，就应该有相应的段寄存器，这就是 LDTR 和 TR，只不过因为作用特殊而不明确地称为段寄存器。

这样，CPU 中的段寄存器一共是 8 个，即 CS、DS、SS、ES、FS、GS、LDTR、和 TR，其中 LDTR 和 TR 为系统段寄存器。相比之下，GDT 中最多可以有 8191 个有效的段描述项，所以只要改变段寄存器中的选择项即下标就可以使其灵活地指向不同的地址段。例如，段寄存器 FS 在用户空间时的内容是 TEB_SELECTOR，而进入内核时就改成 PCR_SELECTOR。这就在 GDT 中选择了不同的表项，从而分别指向当前线程用户空间的 TEB 和内核中的 KPCR 数据结构；而 FS 实际上起着指针的作用，同时也有对于越界访问的保护作用。当然，GDT 中的描述项都是事先设置好了的。

对于具体的进程，还可以为其建立一个 LDT，在 LDT 中又可以有多达 8191 个有效段描述项。在切换线程时，如果要保持 LDTR 的内容不变，则只要改变 GDT 中的相应表项，就可以使其切换到目标进程的 LDT；或者也可以通过改变 LDTR 的内容选择 GDT 中的不同表项，同样达到切换 LDT 的目标。这个思路无疑是很好的，但是实际上却很少使用 LDT，

可能是因为一般的软件并没有复杂到这样的程度。

注意 GDTR 与“段寄存器”的区别在于：在 32 位保护模式下，CS、SS、DS、ES、FS、GS 这些寄存器的内容并不包含目标段的起点和长度，而只是描述表、即全局描述表 GDT 或局部描述表 LDT 中某个表项的下标；但是 GDTR 的内容却不是下标，48 位的 GDTR 中就含有 GDT 的 32 位线性基地址和 16 位的长度。用于中断向量段的 IDTR 也是一样。

相比之下，TR 和 LDTR 就与一般的 CS、SS、DS、ES、FS、GS 这些段寄存器很相似了。TR 和 LDTR 表面上都是 16 位的，实际上却都伴随着隐藏的 64 位段描述项。当然，LDTR 和 TSS 都必须在 GDT 中有相应的表项，每当写入 TR 或 LDTR 的时候，CPU 就自动根据写入的下标从 GDT 中将相应的表项装载到 TR 或 LDTR 的隐藏部分，作为高速缓存。

但是即使是 TR 和 LDTR 也不同于一般的段寄存器，因为对 TR、LDTR、GDTR、以及 IDTR 的操作都有专用的指令，例如 STR/LTR、SLDT/LLDT、SGDT/LGDT 就分别是对于 TR、LDTR、GDTR 的读/写指令，这里的 S 表示“Store”、L 表示“Load”。而一般的段寄存器，则都可以通过 mov 指令进行读/写。

如前所述，“任务状态段”TSS 的设计意图是保存各个任务的执行环境和状态，而当前任务的 TSS 选择项就存储在 TR 中。当一个任务暂时放弃或被剥夺运行时，其当前状态、即所有通用寄存器的映像、就保存在 TSS 中。读者也许要问，这当前状态岂不就是运行现场？那不是保存在系统空间堆栈中吗？是的，但是当初的设计意图并非如此。另一方面，即使现在都是用系统空间堆栈保存运行现场，但这还不是任务状态的全部。这里有个问题：当前进程的系统空间堆栈又在那里？大家知道，任务切换只发生于系统空间，所以一个任务(线程)只有(主动或被动)进入了系统空间才能暂时放弃或者被剥夺运行。既然进入系统空间，就一定要用到系统空间堆栈，而且是在切换的瞬间就得用到，根本就不可能通过一段系统空间的程序去获取当前进程的系统空间堆栈指针，所以必须由 CPU 自动获取这个指针。那么从哪里去获取呢？可见至少得要有个固定的地方，这就是在本任务的 TSS 里面。实际上，除系统空间堆栈指针外，还有些别的信息也保存在 TSS 中。所以，某种形式的 TSS 的存在是确有必要的，只不过是否把运行现场都保存在 TSS 里面则值得商榷。

TSS 中有三个堆栈指针，即 ESP0、ESP1、ESP2、，分别用于 0 环、1 环、和 2 环。当 CPU 从外环通过调用、陷阱、中断、异常进入某个内环时，CPU 就从 TSS 取得该内环的堆栈指针。由于 3 环处于最外围，所以 CPU 不可能从某个更外围的环进入 3 环，所以 TSS 中没有用于 3 环的堆栈指针(见手册第三卷 4.8.5 节)。而所谓 CPU 在系统调用、中断、异常时切换到系统空间堆栈，实际上就是从 TSS 中把 0 环的 SS0 和 ESP0 装入 SS 和 ESP，再把原来 3 环的 SS 和 ESP 压入 0 环堆栈。所以，CPU 中物理的堆栈指针 ESP 只有一个，而逻辑的堆栈指针却最多可以有 4 个。不过，在实际的使用中，无论是 Linux 还是 Windows，都只使用了 0 环和 3 环，即其中的两个。

不同线程的系统空间堆栈处于不同的位置，当然就得有不同的 ESP0。如前所述，当初的设计意图是让每个任务都有自己的 TSS，切换任务时就改变寄存器 TR 的指，使其指向不同任务的 TSS。但是，现在实际上都把运行现场保存在堆栈上，光是为了一个 ESP0 就切换整个 TSS 未免不划算，还不如只用一个固定的 TSS，保持 TR 不变，但是在切换任务时改变一下 TSS 中的 ESP0，使其指向目标线程的系统空间堆栈。

现在可以转入正题，即 Windows 如何调度和切换线程了。

在 Windows 操作系统中，当调度一个线程运行、并实际切换到这个线程时，都需要做些什么、改变些什么呢？结合上面介绍的背景材料，我们先归纳一下，然后再看有关的代码。

注意下面有时候说线程、有时候说进程，这是因为有些特性是属于进程、可能为多个线程所共有的：

- 由于所有线程(不管属于哪一个进程)都共用同一个 TSS 数据结构，在切换线程时就需要改变 TSS 中的一些字段、即某些寄存器的映像。特别地：
  1. 各个线程的系统空间堆栈位置各不相同，所以字段 ESP0 的字段是必须改变的。注意需要恢复的只是系统空间的堆栈指针，用户空间的堆栈指针已经保存在系统空间堆栈的陷阱框架中，返回用户空间时自然就会恢复。
  2. 各个进程可能有自己的 I/O 权限位图，这个位图的起点位移保存在进程的 EPROCESS 结构中，切换线程时要把它复制到 TSS 中。
- 各个进程的内存映射各不相同，各有自己的页面目录。各进程页面目录的起始地址保存在 KPROCESS 数据结构中，需要把它设置到控制寄存器 CR3 中。
- 每个线程在用户空间都有个“线程环境块”TEB，这就是进入用户空间时段寄存器 FS 所指的存储段。所以，GDT 中的相应段描述项也需要加以改变。
- 如果用到 LDT 的话，GDT 中的 LDT 段描述项的映像也需要改变。
- FPU 状态，如果使用浮点处理器的话，在切换线程时也要切换浮点运算的上下文，就是保存和恢复浮点处理器的状态(FX_SAVE_AREA)。

可见，在切换线程的时候需要设置不少的寄存器，而这牵涉到不少的指针和数据结构。把这些信息保存在许多离散的变量中，而把对所有这些变量的引用“硬编码”在程序中当然也是可以的，但是最好还是把它们集中保存在一个数据结构中。这样，程序中只需要访问一个变量，就取得了指向这个数据结构的指针，然后就可以“顺藤摸瓜”获取其它的信息了。特别地，当系统中有不止一个 CPU 时，这样做就更有必要了，因为在这样的系统中每个 CPU 都有一套这样的数据，还有许多从属于具体 CPU 的信息，如果分散保存就更不好办了。所以 Windows 内核中为此定义了一套以“处理器控制区(Processor Control Region)”KPCR 为枢纽的数据结构，使每个 CPU 都有个 KPCR 结构，用来保存与线程切换有关的全局信息。KPCR 数据结构的定义如下：

```
typedef struct _KPCR {
   KPCR_TIB    Tib;                /* 00 */
   struct _KPCR    *Self;          /* 1C */
   struct _KPRCB    *Prcb;         /* 20 */
   KIRQL    Irql;                  /* 24 */
   ULONG    IRR;                   /* 28 */
   ULONG    IrrActive;             /* 2C */
   ULONG    IDR;                   /* 30 */
   PVOID    KdVersionBlock;        /* 34 */
   PUSHORT    IDT;                 /* 38 */
   PUSHORT    GDT;                 /* 3C */
   struct _KTSS    *TSS;           /* 40 */
   USHORT    MajorVersion;         /* 44 */
   USHORT    MinorVersion;         /* 46 */
   KAFFINITY    SetMember;         /* 48 */
   ULONG    StallScaleFactor;      /* 4C */
   UCHAR    DebugActive;           /* 50 */
```

```
    UCHAR   ProcessorNumber;        /* 51 */
    UCHAR   Reserved;               /* 52 */
    UCHAR   L2CacheAssociativity;   /* 53 */
    ULONG   VdmAlert;               /* 54 */
    ULONG   KernelReserved[14];     /* 58 */
    ULONG   L2CacheSize;            /* 90 */
    ULONG   HalReserved[16];        /* 94 */
    ULONG   InterruptMode;          /* D4 */
    UCHAR   KernelReserved2[0x48]; /* D8 */
    KPRCB   PrcbData;               /* 120 */
} KPCR, *PKPCR;
```

数据结构的定义取自 ReactOS 的代码，不过微软在 DDK 中也公开了这种数据结构的定义，只是比这里的小了一些，从位移 0x54 以后的字段都没有了。

KPCR 结构中的第一个成分 TIB 也是个数据结构，即 KPCR_TIB 数据结构：

```
typedef struct _KPCR_TIB {
    PVOID   ExceptionList;          /* 00 */
    PVOID   StackBase;              /* 04 */
    PVOID   StackLimit;             /* 08 */
    PVOID   SubSystemTib;           /* 0C */
    _ANONYMOUS_UNION union {
        PVOID   FiberData;          /* 10 */
        DWORD   Version;            /* 10 */
    } DUMMYUNIONNAME;
    PVOID   ArbitraryUserPointer;   /* 14 */
    struct _NT_TIB *Self;           /* 18 */
} KPCR_TIB, *PKPCR_TIB;             /* 1C */
```

这是 KPCR 结构中至关重要的成分。首先其中 StackBase 和 StackLimit 的重要性显而可见的，而 ExceptionList 则是实现“结构化异常处理(SEH)”所必不可少的，这我以后还要专门加以介绍。

KPCR 结构中的第二个成分 self 是个指针，指向其所在 KPCR 结构的起点，之所以这样安排的原因后面会讲到。

KPCR 结构中的第三个成分 Prcb 又是个指针，指向一个“处理器控制块”、即 KPRCB 数据结构。这个结构中的信息可就多了，下面所列的只是一部分：

```
/* ProcessoR Control Block */
typedef struct _KPRCB {
    USHORT MinorVersion;
    USHORT MajorVersion;
    struct _KTHREAD *CurrentThread;
    struct _KTHREAD *NextThread;
    struct _KTHREAD *IdleThread;
```

```
    . . . . . .
    UCHAR CpuType;
    UCHAR CpuID;
    USHORT CpuStep;
    KPROCESSOR_STATE ProcessorState;
    . . . . . .
    PVOID LockQueue[33];        // Used for Queued Spinlocks
    struct _KTHREAD *NpxThread;
    ULONG InterruptCount;
    ULONG KernelTime;
    ULONG UserTime;
    . . . . . .
    struct _KEVENT *DpcEvent;
    UCHAR ThreadDpcEnable;
    BOOLEAN QuantumEnd;
    . . . . . .
    LONG MmPageReadCount;
    LONG MmPageReadIoCount;
    LONG MmCacheReadCount;
    LONG MmCacheIoCount;
    LONG MmDirtyPagesWriteCount;
    . . . . . .
    FX_SAVE_AREA NpxSaveArea;
    PROCESSOR_POWER_STATE PowerState;
} KPRCB, *PKPRCB;
```

里面的指针 CurrentThread 指向当前线程的 KTHREAD 数据结构，而 NextThread 指向已经预先调度运行的下一个线程，还有 IdleThread 则指向系统中的空转线程。

KPCR 中还有个指针 TSS，指向一个 KTSS 数据结构，这就是任务状态段 TSS，每个 CPU 都有一个自己的 TSS。

显然，从 KPCR 开始的这一套数据结构并不是专为线程调度和切换而设的，里面还包含着许多统计信息；还有如 LockQueue[33]则是用于 Spinlock 的数组，它把可能需要用到的 Spinlock 集中到了一起。但是，既然有了 KPCR，Windows 的线程调度和切换也就离不开这些数据结构了。

在单 CPU 的系统中只有一个 KPCR 数据结构，其位置固定在地址为 KPCR_BASE、即 0xFF000000 的地方。而在多 CPU 系统中、即在 SMP 结构的系统中则是个数组，数组的起点地址也是 KPCR_BASE，用 CPU 的逻辑编号作为下标就可以取得该 CPU 的 KPCR 数据结构。

```
#define KPCR_BASE                       0xFF000000
```

当一个线程被调度在某个 CPU 上运行、并且运行于系统空间时，段寄存器 FS 的内容、

即段选择项、就总是设置成 PCR_SELECTOR，而 GDT 中的相应描述项则总是指向这个 CPU 的 KPCR 数据结构。以前读者看到 CPU 因系统调用或中断、异常进入系统空间时总是要把 FS 的内容换成 PCR_SELECTOR，就是这个道理。这样，在系统空间，只要以 FS 加位移的方式寻址，就可以方便地访问其所在 CPU 的 KPCR 结构中的各个字段，以及实际上还有紧随在 KPCR 结构后面的 KPRCB 结构。例如，%fs:0 就总是指向本线程的 KPCR。但是，在汇编指令中“%fs:0”是 KPCR 的第一个 32 位长字的内容，却无法取它的地址，因为这不是一个变量。当然，根据段寄存器 FS 的内容，在 GDT 中可以找到相应的表项，从而找到其起始地址，但是那毕竟太麻烦了。这就是为什么要在 KPCR 中放上一个指针 Self 的原因。这个指针在 KPCR 中的位移是 0x1c，所以“%fs:0x1c”就是 KPCR 的起点，下面这一小段代码就说明了这一点：

```
static __inline struct _KPCR * KeGetCurrentKPCR(VOID)
{
   ULONG Value;
   __asm__ __volatile__ ("movl %%fs:0x1C, %0\n\t"
         : "=r" (Value)
      : /* no inputs */);
   return (struct _KPCR *) Value;
}
```

现在我们可以看线程切换的代码了。在 ReactOS 中，线程切换是由 Ki386ContextSwitch() 实现的。这个函数有两个调用参数，就是新、老两个线程的 KTHREAD 结构指针。

[Ki386ContextSwitch()]

```
_Ki386ContextSwitch:
   pushl   %ebp
   movl    %esp, %ebp
   /* Save callee save registers. */
   pushl   %ebx
   pushl   %esi
   pushl   %edi
   /* This is a critical section for this processor. */
   cli
#ifdef CONFIG_SMP
   . . . . . .
#endif /* CONFIG_SMP */

   /* Get the pointer to the new thread. */
   movl   8(%ebp), %ebx

   /* Set the base of the TEB selector to the base of the TEB for this thread. */
   pushl   %ebx
```

```
pushl   KTHREAD_TEB(%ebx)
pushl   $TEB_SELECTOR
call    _KeSetBaseGdtSelector
addl    $8, %esp
popl    %ebx
```

首先设置框架指针 EBP。设置了框架指针以后，第一个参数 8(%ebp)就是目标线程、即新线程的 KTHREAD 结构指针；而第二个参数 12(%ebp)则是当前线程、即老线程的 KTHREAD 结构指针。

切换线程的过程当然不能受中断干扰，所以要把这个过程置于关闭中断的条件下进行。

这里跳过了#ifdef CONFIG_SMP 下面用于多处理器 SMP 系统结构的代码，先把单处理器结构下的线程切换搞清楚。

我们知道，当 CPU 回到用户空间时，段寄存器 FS 的内容应该指向当前线程的 TEB。但是同一个进程中每个线程的 TEB 的位置都是不同的，显然需要在切换线程时加以改变。然而段寄存器 FS 是个 16 位寄存器，在保护模式下并不直接含有地址，而只是一个“段选择符”，由一个(13 位)下标、一个描述表选择位、和两位的优先级构成。具体到 TEB_SELECTOR，其定义为(0x38 + 0x3)，说明下标为 7，选择的是 GDT，作用于用户空间。至于具体的地址，则编码在 GDT 的相应表项中。所以这里就通过 KeSetBaseGdtSelector()设置这个表项。调用参数 TEB_SELECTOR 指明了需要改变的表项，而 KTHREAD_TEB(%ebx)则取自目标进程 KTHREAD 结构中的 TEB 字段。注意 KeSetBaseGdtSelector()改变的是 GDT 数据结构中的表项，数据结构的地址取自当前 CPU 的 KPCR 结构，而 CPU 中的 GDTR 也指向这个 GDT 数据结构。但是这并不意味着每次通过段寄存器访问内存时都要先访问 GDT 中的描述项，那样当然效率太低了。程序员所见到的 16 位 FS 寄存器实际上只是这个段寄存器的可见部分，CPU 中的 FS 还有一个隐藏部分。每当将一个“段选择符”装入 FS 时，CPU 就会根据段选择符中的信息和 GDTR 的内容找到相应的段描述项，并将这个描述项装入 FS 的隐藏部分。这样，通过段寄存器访问内存时就无须访问 GDT 数据结构了。FS 是这样，其它段寄存器也是一样。那么什么时候把选择符 TEB_SELECTOR 装入 FS 呢？这个选择符是在用户空间才使用的，在 CPU 进入系统空间，形成不管是因为系统调用、中断、还是异常的陷阱框架的时候，都有这么几条指令：

```
. . . . . .
pushl   %fs

/* Load PCR Selector into fs */
movw    $PCR_SELECTOR, %bx
movw    %bx, %fs
. . . . . .
```

就是说，在进入系统空间时保存原来指向 TEB 的段选择符，并且把指向 KPCR 的段选择符装入 FS，这就蕴含着把指向 KPCR 的段描述项装入了 FS 的隐藏部分。而在返回用户空间时则通过“popl %fs”指令恢复原来的段选择符，这时候就又把指向 TEB 的段描述符装入了 FS 的隐藏部分。至于在创建线程的时候，则在 Ke386InitThreadWithContext()中把 TEB_SELECTOR 预先设置在虚构的陷阱框架中。

我们继续往下看代码：

[Ki386ContextSwitch()]

```
   /* Load the PCR selector. */
   movl   $PCR_SELECTOR, %eax
   movl   %eax, %fs

   /* Set the current thread information in the PCR. */
   movl   %ebx, %fs:KPCR_CURRENT_THREAD

   /* Set the current LDT */
   xorl    %eax, %eax
   movl   KTHREAD_APCSTATE_PROCESS(%ebx), %edi     //
   testw   $0xFFFF, KPROCESS_LDT_DESCRIPTOR0(%edi)
   jz 0f

   pushl   KPROCESS_LDT_DESCRIPTOR1(%edi)
   pushl   KPROCESS_LDT_DESCRIPTOR0(%edi)
   pushl   $LDT_SELECTOR
   call    _KeSetGdtSelector
   addl    $12, %esp
   movl   $LDT_SELECTOR, %eax
0:
   lldtw   %ax
```

在内核中，段寄存器的内容总是 PCR_SELECTOR，所选择的是 GDT 中以此为下标的段描述项，实际上总是指向本 CPU 的 KPCR 数据结构，而%fs:KPCR_CURRENT_THREAD 则为离 KPCR 数据结构起点的位移为 KPCR_CURRENT_THREAD、即 0x124 处的字段。但是 KPCR 数据结构的大小一共才 0x120，所以这实际上是 KPRCB 数据结构中的结构指针 CurrentThread。这样，就使这个指针指向了目标线程的 KTHREAD 结构。注意这里有个前提，就是 KPRCB 数据结构必须是紧跟在 KPCR 数据结构的后面(这样的程序设计实在不敢恭维)；然而 KPCR 数据结构中却又有个指针，指向与其配对的 KPRCB 数据结构，似乎意在让 KPRCB 数据结构可以独立存在。

由于段寄存器 FS 在系统空间总是指向当前进程的 KPCR，为了提高效率，常常用汇编指令通过 FS 访问 KPCR 结构中的字段，例如：

```
static inline PKPRCB KeGetCurrentPrcb(VOID)
{
   ULONG value;

#if defined(__GNUC__)
   __asm__ __volatile__ ("movl %%fs:0x20, %0\n\t"
         : "=r" (value)
       : /* no inputs */
```

```
    );
#elif defined(_MSC_VER)
  ......
#endif
  return((PKPRCB)value);
}
```

这里 fs:0x20 就是 KPCR 数据结构中的指针 Prcb，指向相应的 KPRCB 数据结构。这样的代码，如果在维护中需要作一些改变，那可真是牵一发动全身。

不管怎么说，现在 KeGetCurrentPrcb()->CurrentThread 已经指向目标线程了。可见，KPRCB 数据结构中的这个指针总是指向目标线程程的 ETHREAD 数据结构。事实上，底层函数 KeGetCurrentThread()的实现在单 CPU 结构的系统中就是返回这个指针：

```
PKTHREAD STDCALL KeGetCurrentThread(VOID)
{
#ifdef CONFIG_SMP
    ULONG Flags;
    PKTHREAD Thread;
    Ke386SaveFlags(Flags);
    Ke386DisableInterrupts();
    Thread = KeGetCurrentPrcb()->CurrentThread;
    Ke386RestoreFlags(Flags);
    return Thread;
#else
    return(KeGetCurrentPrcb()->CurrentThread);
#endif
}
```

回到前面的汇编代码中。

下面的指令“movl KTHREAD_APCSTATE_PROCESS(%ebx), %edi”使寄存器 EDI 指向了目标线程所属进程的 KPROCESS 数据结构。这个数据结构中有个 ULONG 数组 LdtDescriptor[2]，如果其第一个元素的低 16 位非 0 就是一个有效的 LDT 段描述项，那就说明目标线程使用了 LDT，因此就要把这个指向其 LDT 段的描述项设置到 GDT 中，其下标为 LDT_SELECTOR。同时还要通过指令 lldt 把这个下标作为段选择项装入到寄存器 LDTR 中。如前所述，LDTR 实质上是个段寄存器，其结构与普通的段寄存器相同。这就是说，它在 CPU 中有个 16 位的可见部分，还有个 64 位的隐藏部分。而 lldt 指令，则一方面把 16 位的段选择项置入 LDTR 的可见部分，同时也从 GDT 中把相应的表项装入了 LDTR 的隐藏部分。这样：

- 当用户程序把 FS、GS 等段寄存器设置成使用 LDT 时(选择项中的表选择位为 0 表示使用 GDT，为 1 表示使用 LDT)，根据 LDTR 就可以找到 LDT，而不用再去访问 GDT 中的描述项。
- 根据置入 FS、GS 等段寄存器的选择项和 LDT 的内容，把 LDT 中的相应表项装入 FS、GS 等段寄存器的隐藏部分
- 于是，当用户程序通过 FS、GS 等段寄存器访问内存时，就无需再去访问 LDT 中

的描述项。

注意在不使用 LDT 时装入 LDTR 的段选择项是 0，而 GDT 中下标为 0 处是个非法段描述项，所以要是以后企图访问 LDT(例如把段寄存器 FS 设置成使用 LDT)就会导致异常。

我们再往下看。

[Ki386ContextSwitch()]

```
  /* Get the pointer to the old thread. */
  movl    12(%ebp), %ebx

  /* FIXME: Save debugging state. */

  /* Load up the iomap offset for this thread in preparation for setting it below. */
  movl    KPROCESS_IOPM_OFFSET(%edi), %eax

  /* Save the stack pointer in this processors TSS */
  movl    %fs:KPCR_TSS, %esi
  pushl   KTSS_ESP0(%esi)

  /* Switch stacks */
  movl    %esp, KTHREAD_KERNEL_STACK(%ebx)
  movl    8(%ebp), %ebx
  movl    KTHREAD_KERNEL_STACK(%ebx), %esp
  movl    KTHREAD_STACK_LIMIT(%ebx), %edi
  movl    %fs:KPCR_TSS, %esi

  /* Set current IOPM offset in the TSS */
  movw  %ax, KTSS_IOMAPBASE(%esi)
```

这里使 EBX 指向了老线程，其实要使用这个指针的地方还在后面。此时的 EDI 仍指向新线程(目标线程)所属进程的 KPROCESS 数据结构，而 KPROCESS_IOPM_OFFSET(%edi)就是 KPROCESS 结构中字段 IopmOffset 的内容。这个字段说明该进程的 IO 权限位图在 TSS 中的位置，这里先把它装入了 EAX，后面会把它写入 TSS 中的 IoMapBase 字段。

随后的指令“movl %fs:KPCR_TSS, %esi”把 KPCR 中的指针 TSS 装入 ESI，使其指向了 KTSS 数据结构，接着就把这个结构中字段 Esp0 的当前值压入堆栈。如前所述，TSS 中的这个字段总是指向当前线程系统空间堆栈的原点，当一个线程不再成为当前线程时就得把它保存起来。保存在哪里呢？办法当然不止一种，例如保存在 KTHREAD 结构中也未尝不可，而这里选择的是保存在堆栈中，那当然也可以。

至此，已经为堆栈的切换作好了准备，下面就是切换堆栈了，注意此时 EBX 指向老线程的 KTHREAD 结构。首先把此刻的堆栈指针记录在老线程 KTHREAD 结构中的字段 KernelStack 中。这就是老线程在切换点上的系统空间堆栈指针。然后又使 EBX 指向新线程的 KTHREAD 结构，从中恢复其保存着的堆栈指针，读者不妨回顾一下上一篇漫谈中所讲的这个字段的作用。注意这里的保存堆栈指针和恢复堆栈指针是分别针对两个不同线程、两个不同 KTHREAD 数据结构的操作。

从现在起，程序就开始使用另一个线程的系统空间堆栈了。读者也许心中疑虑，就这么把堆栈换了，会不会给程序的运行带来断裂？不会的。对于老线程，已经在堆栈上的内容或者是要到返回的时候才会用到，或者是在程序中需要这些数据时才会用到，但是那都发生在下一次当这个线程又被调度运行的时候。对于新线程，则下面要用到的堆栈内容都是在上一次当这个线程被调度停止运行时压入堆栈的。

下一条 mov 指令把目标进程 KTHREAD 结构中字段 StackLimit 的值置入 EDI，但是这似乎是多余的。注意此前 EDI 指向新线程的 KPROCESS 数据结构，现在则变成了新线程的 StackLimit，可是后面没有看到此项数据的使用。

再下一条 mov 指令把 KPCR 结构中的指针 TSS 置入 ESI，使其指向本 CPU 的 KTSS 数据结构。但是这似乎又是多余的，因为在切换堆栈的那几条指令的前后 FS 的内容并未改变，GDT 中的相应表项也未改变，又是在同一个 CPU 上，所以前后两条以 ESI 为目标的 mov 指令应该有着相同的效果。话虽如此，读者要是看出这里面有甚么奥妙就请发个 Email 给作者。

下面是把当前进程的 IopmOffset 位移写入 KTSS 数据结构中的 IoMapBase 字段。这里寄存器 EAX 的内容在切换堆栈之前来自目标进程的 KPROCESS 结构，现在则把它写入 KTSS 结构的 IoMapBase 字段中。这个字段的值是个 16 位的位移量，说明 IO 权限位图在 TSS 中的位置。这样，KTSS 数据结构中的 IoMapBase 就总是指向当前进程的 IO 权限位图，或者说明当前进程没有 IO 权限位图(如果 IoMapBase 为 0xffff)。需要说明的是，大部分 Windows 进程都没有 IO 权限位图，因而只有在内核中才能进行 I/O 操作；有 IO 权限位图的只是特殊的进程，一般是运行于 V86 模式的进程。

系统空间堆栈的切换意味着 CPU 的执行已由老线程转到新进程，已经恢复了新线程在系统空间的运行。但是，除非是内核线程，一般而言新线程最后还得回到用户空间，而此刻的用户空间映射还是老线程的，所以还得切换用户空间的映射。继续往下看：

[Ki386ContextSwitch()]

```
   /* Change the address space */
   movl    KTHREAD_APCSTATE_PROCESS(%ebx), %eax
   movl    KPROCESS_DIRECTORY_TABLE_BASE(%eax), %eax
   movl    %eax, %cr3

   /* Restore the stack pointer in this processors TSS*/
   popl    KTSS_ESP0(%esi)

   /* Set TS in cr0 to catch FPU code and load the FPU state when needed
    * For uni-processor we do this only if NewThread != KPCR->NpxThread */
#ifndef CONFIG_SMP
   cmpl    %ebx, %fs:KPCR_NPX_THREAD
   je 4f
#endif /* !CONFIG_SMP */
   movl    %cr0, %eax
   orl     $X86_CR0_TS, %eax
   movl    %eax, %cr0
4:
```

```
    /* FIXME: Restore debugging state */
    /* Exit the critical section */
    sti

    call       @KeReleaseDispatcherDatabaseLockFromDpcLevel@0

    cmpl    $0, _PiNrThreadsAwaitingReaping
    je 5f
    call       _PiWakeupReaperThread@0
5:

    /* Restore the saved register and exit */
    popl    %edi
    popl    %esi
    popl    %ebx

    popl    %ebp
    ret
```

除非内核线程，每个线程都在其所属进程的空间中运行，因而使用某个特定的页面目录。不同页面目录中系统空间页面的映射都是相同的，所不同的是用户空间页面的映射。至于内核线程则只有系统空间页面的映射，而没有用户空间页面的映射。所以，切换线程的时候也要切换页面目录。而页面目录属于进程，这里 KTHREAD_APCSTATE_PROCESS(%ebx)实际上是 KTHREAD 数据结构内部 KAPC_STATE 结构中的字段 Process 的值。这是个指针，指向其所属进程的 KPROCESS 数据结构。把这个指针赋给 EAX 以后，KPROCESS_DIRECTORY_TABLE_BASE(%eax) 就是 KPROCESS 数据结构中字段 DirectoryTableBase 的值，这又是个指针(物理地址)，指向该进程的页面目录。把这个值写入控制寄存器 CR3，就引起了地址映射的切换，不过此刻的程序是在系统空间执行，而所有进程的系统空间都是相同的，所以这种切换并不影响程序的继续执行，其作用要倒 CPU 回到用户空间时才表现出来。

下面从堆栈恢复 TSS 中的 ESP0，这条 pop 指令与切换堆栈之前的 push 指令相对应，但却是针对不同的堆栈。前面的 push 指令是针对老线程的系统空间堆栈，而后面的 pop 指令则是针对新线程的系统空间堆栈。而所谓“新线程”，很可能是从前的某一次线程切换中的“老线程”。也就是说，当一个线程不被执行时，其 ESP0 存放在它的系统空间堆栈上，到被调度运行并切换时再把 ESP0 写入 TSS。这样，CPU 在需要从用户空间进入系统空间时才能知道当前线程的系统空间堆栈在哪里。注意在修改了 KTSS 的某些内容后并不需要重新装入段寄存器 TR，TR 还是指向原来的地方，因为 TSS 还是原来的 TSS，而所改变的内容都是在实际用到时才由 CPU 到 TSS 中获取，例如 Esp0 就是要到 CPU 从用户空间进入系统空间时才会用到的。

此后的几条指令与浮点运算有关。控制寄存器 CR0 的一些标志位控制着 CPU 许多方面的运行状态，例如是否开启页面映射、是否启用高速缓存等等都是由 CR0 控制的。但是这里所关心的是其中与浮点运算有关的标志位 X86_CR0_TS：

```
#define X86_CR0_TS    0x00000008    /* enable exception on FPU instruction for task switch */
```

这是一个控制/标志位，TS 表示“Task Switched”，其作用是使浮点处理器 FPU 的上下文不必立即加以保存，因为新的目标线程在运行中未必会用到 FPU，每次切换时都加以保存/恢复就造成浪费。这里的程序中把这一位设成 1 以后，如果新的线程真的用到 FPU，就会在首次使用 FPU 时导致一次异常，在相应的异常处理程序中再来处理 FPU 的切换就可以了(详见 Intel 的软件开发手册第三卷)。

至此，线程切换的关键操作都已完成，可以打开中断了。后面的 KeReleaseDispatcherDatabaseLockFromDpcLevel()显然是解锁，与其相对应的加锁操作在上一层的程序中，所以在这里看不到。至于 PiWakeupReaperThread()，则是唤醒一个内核线程，让它来“收割”那些已经退出运行的线程，实际上就是释放它们的数据结构。

我们不妨考察一下在这整个过程中的堆栈操作。首先所有的 push 操作和 pop 操作显然是平衡的、即数量相等，这在任何函数中都是一样。进一步，除少数例外，绝大多数的 push 操作和 pop 操作也是配对的，例如前面有“pushl KTSS_ESP0(%esi)”，后面就有“popl KTSS_ESP0(%esi)”。最后，至关重要的是，在这个函数内部，这些 push 操作和 pop 操作实际上作用于两个不同的堆栈，即两个不同线程的系统空间堆栈。所以，与前面的 push 操作配对的确实就是后面的那些 pop 操作，但是这些 pop 指令的执行却是“老线程”在下一次被调度运行、并因此而而执行 Ki386ContextSwitch()、变成了“新线程”时的时候。

新创建的线程是个特例。新建线程系统空间堆栈上的这些数据当然不可能是在切换线程时保存进去的，而是预先安排好的。我们再回顾一下 Ke386InitThreadWithContext()中的这几行代码：

```
KernelStack[0] = (ULONG)Thread->InitialStack - sizeof(FX_SAVE_AREA); /* TSS->Esp0 */
KernelStack[1] = 0;          /* EDI */
KernelStack[2] = 0;          /* ESI */
KernelStack[3] = 0;          /* EBX */
KernelStack[4] = 0;          /* EBP */
KernelStack[5] = (ULONG)&PsBeginThreadWithContextInternal;     /* EIP */
```

比较一下前面的那些 pop 语句，就可以知道当新建线程被调度运行时被恢复到 KTSS_ESP0(%esi)、即 TSS 中 ESP0 字段的是系统空间堆栈的原点，即堆栈区间顶部减去一个 FX_SAVE_AREA 数据结构以后的边界上。而寄存器 EDI、ESI、EBX、和 EBP 的初值则为 0。

新建线程开始运行时 TSS 中的 ESP0 字段被设置成系统空间堆栈原点。这样，在 CPU 回到用户空间之后，如果发生中断、异常、或者系统调用，CPU 就会把 TSS 中的 ESP0 装入堆栈指针寄存器 ESP，使其指向系统空间堆栈的原点。系统空间堆栈的 SS 也取自 TSS，但是那实际上一经初始化以后便不再改变，所有线程在系统空间都使用同一个堆栈段。从此以后，这个线程的系统空间堆栈原点就会永远保持下去，作为当前线程运行时这个数值在 TSS 的 ESP0 中，不运行时则保存在其自己的系统空间堆栈中。

值得注意的还有 KernelStack[5]、即返回地址，这是 PsBeginThreadWithContextInternal。本来，调用 Ki386ContextSwitch()的地方是 PsDispatchThreadNoLock()，从 Ki386ContextSwitch()返回时应该返回到那里去，但是那样就得在新建线程的堆栈上构建出包含多个函数调用框架的整个上下文，因为 PsDispatchThreadNoLock()又是受别的函数调用的。对于新建的线程，那样做既麻烦又无必要，所以这里让它抄近路“返回”到 PsBeginThreadWithContextInternal。

在那里，读者在前一篇漫谈中已经看到，稍作处理就直接跳转到了_KiServiceExit。

读者也许要问，这里保存在堆栈上的寄存器才那么几个，这就够了吗？是的。须知线程切换一定是在 Ki386ContextSwitch()进行的，首先这是在系统空间，所有寄存器在用户空间的内容都已经在进入系统空间时保存在陷阱框架中。而这些寄存器在线程切换前夕的内容，如果需要的话，也已经保存在系统空间堆栈上，或者本来就在系统空间堆栈上(作为局部变量的值)，或者保存在有关的数据结构中。所以到进入 Ki386ContextSwitch()的时候实际上已经没有什么需要保存的了，这里之所以要保存 EDI、ESI、EBX、和 EBP 的值，只是因为在切换的过程中需要用到这几个寄存器。除此以外，真正需要保存/恢复的数据其实只有一项，那就是 KTSS_ESP0(%esi)，因为每个线程的系统空间堆栈的位置是不同的。

明白了线程切换的过程，剩下来的问题是什么时侯切换。这不用说当然是调度的时侯切换，于是问题变成了什么时候调度。事实上，很多情况都会引起线程调度：

- 当前线程通过 NtYieldExecution()系统调用自愿礼让。
- 当前线程在别的系统调用中因操作受阻而半自愿地交出运行权。
- 当前线程通过 NtSetInformationThread()等系统调用改变了自身或其它线程/进程的优先级，使得自己可能不再具有最高的运行优先级。
- 当前线程通过 NtSuspendThread()挂起其自身的运行。
- 当前线程通过 NtResumeThread()恢复了其它线程的运行，使得自己可能不再具有最高的运行优先级。
- 当前线程通过进程间通信/线程间通信唤醒了别的进程，使得自己可能不再具有最高的运行优先级。
- 对时钟中断的处理发现当前线程已经用完时间配额，因而调度其它线程运行。
- 其它中断的发生导致某个/某些线程被唤醒，从而使得当前线程可能不再具有最高的运行优先级。

明白了在什么时侯调度，剩下的就是怎样调度、特别是根据什么准则调度的问题了。

下面我们通过一个实际的情景来解答这个问题。为简单起见，我们从系统调用 NtYieldExecution()着手来看这整个过程。这个系统调用的作用是使当前线程暂时放弃运行，但又不进入睡眠，实际上就是为别的线程让一下路，相当与 Linux 中的 yield()。

```
NTSTATUS STDCALL
NtYieldExecution(VOID)
{
  PsDispatchThread(THREAD_STATE_READY);
  return(STATUS_SUCCESS);
}
```

由于只是暂时退让，当前线程并不被阻塞，其运行状态仍为 THREAD_STATE_READY、仍处于就绪状态，而只是通过 PsDispatchThread()启动一次线程调度，但是当前线程自己并不参与竞争。

[NtYieldExecution() > PsDispatchThread()]

```
VOID STDCALL PsDispatchThread(ULONG NewThreadStatus)
{
```

```
    KIRQL oldIrql;

    if (!DoneInitYet || KeGetCurrentPrcb()->IdleThread == NULL)
    {
        return;
    }
    oldIrql = KeAcquireDispatcherDatabaseLock();
    PsDispatchThreadNoLock(NewThreadStatus);
    KeLowerIrql(oldIrql);
}
```

实际的调度是由 PsDispatchThreadNoLock()完成的。调度的过程需要排它地进行，不能在中途又因为别的原因而再次进入调度的过程，所以要对这整个进程加上锁。特别地，调度的过程中涉及许多队列操作，而队列操作是必须排它进行的。至于所谓 Database，实际上就是指这些队列。要不然，如果不加锁的话，例如要是中途发生了一次时钟中断，而时钟中断服务程序发现当前进程已经用完了时间配额，就又会启动线程调度，这就乱了套。而 PsDispatchThreadNoLock()，正如其函数名所示，是不管加锁这事的，所以这里要先加上锁。

但是读者也许会问，既然在调用 PsDispatchThreadNoLock()之前先上了锁，那么理应在从这个函数返回以后开锁，怎么这里看不到呢？这是因为当前线程对 PsDispatchThreadNoLock()的调用并不是在完成了调度以后就立即返回，而要到它下一次又被调度运行时才会返回，中间还夹着其它线程的运行。所以，到从 PsDispatchThreadNoLock()返回的时侯才开锁，那就错了。

[NtYieldExecution() > PsDispatchThread() > PsDispatchThreadNoLock()]

```
VOID PsDispatchThreadNoLock (ULONG NewThreadStatus)
{
    KPRIORITY CurrentPriority;
    PETHREAD Candidate;
    ULONG Affinity;
    PKTHREAD KCurrentThread = KeGetCurrentThread();
    PETHREAD CurrentThread =
                                CONTAINING_RECORD(KCurrentThread, ETHREAD, Tcb);

    . . . . . .

    CurrentThread->Tcb.State = (UCHAR)NewThreadStatus;
    switch(NewThreadStatus)
    {
      case THREAD_STATE_READY:
          PsInsertIntoThreadList(CurrentThread->Tcb.Priority, CurrentThread);
          break;
      case THREAD_STATE_TERMINATED_1:
          PsQueueThreadReap(CurrentThread);
```

```
        break;
}

Affinity = 1 << KeGetCurrentProcessorNumber();
for (CurrentPriority = HIGH_PRIORITY;
      CurrentPriority >= LOW_PRIORITY; CurrentPriority--)
{
   Candidate = PsScanThreadList(CurrentPriority, Affinity);
   if (Candidate == CurrentThread)
   {
        Candidate->Tcb.State = THREAD_STATE_RUNNING;
        KeReleaseDispatcherDatabaseLockFromDpcLevel();
        return;
   }
   if (Candidate != NULL)
   {
        PETHREAD OldThread;
        PKTHREAD IdleThread;

        DPRINT("Scheduling %x(%d)\n",Candidate, CurrentPriority);

        Candidate->Tcb.State = THREAD_STATE_RUNNING;

        OldThread = CurrentThread;
        CurrentThread = Candidate;
        IdleThread = KeGetCurrentPrcb()->IdleThread;

        if (&OldThread->Tcb == IdleThread)
        {
            IdleProcessorMask &= ~Affinity;
        }
        else if (&CurrentThread->Tcb == IdleThread)
        {
            IdleProcessorMask |= Affinity;
        }

        MmUpdatePageDir(PsGetCurrentProcess(),
                    (PVOID)CurrentThread->ThreadsProcess, sizeof(EPROCESS));

        KiArchContextSwitch(&CurrentThread->Tcb, &OldThread->Tcb);
        return;
    }
}
CPRINT("CRITICAL: No threads are ready (CPU%d)\n", KeGetCurrentProcessorNumber());
```

```
    PsDumpThreads(TRUE);
    KEBUGCHECK(0);
}
```

一进入这个函数，就使局部量 KcurrentThread 指向当前线程的 KTHREAD 数据结构，并使 CurrentThread 指向相应的 ETHREAD 数据结构。宏定义 CONTAINING_RECORD 根据指向数据结构内部成分的指针和外层结构的类型推算出指向外层结构的指针。实际上，KTHREAD 数据结构是 ETHREAD 结构内部的第一个成分，所以这两个指针其实是一样的，不过这样做更为安全(万一有谁修改了 ETHREAD 的定义)。

作为参数传下来的是当前线程的新的状态。如果当前线程被阻塞，那就是 THREAD_STATE_BLOCKED，但是在我们这个情景中是 THREAD_STATE_READY。

状态为 THREAD_STATE_READY 的线程应该挂入系统的就绪队列。所谓就绪队列，实际上是一组队列，每一种线程优先级都有一个队列。为此，内核中有个 LIST_ENTRY 结构数组 PriorityListHead[MAXIMUM_PRIORITY]，数组的大小 MAXIMUM_PRIORITY 定义为 32，对应着 32 种不同的线程优先级。

[NtYieldExecution() > PsDispatchThread() > PsDispatchThreadNoLock()
> PsInsertIntoThreadList ()]

```
static VOID
PsInsertIntoThreadList(KPRIORITY Priority, PETHREAD Thread)
{
    ......
    InsertTailList(&PriorityListHead[Priority], &Thread->Tcb.QueueListEntry);
    PriorityListMask |= (1 << Priority);
}
```

显然，这是通过 KTHREAD 结构中的 QueueListEntry 将其挂入给定优先级的就绪队列。

与 32 个就绪队列相对应，内核中还有个位图 PriorityListMask，只要某个优先级的就绪队列非空，这个位图中相应的标志位就设置成 1。这样，只要看一下位图，就知道有没有某个优先级的线程在等待被调度运行了。

那么，不在就绪状态的线程怎么办呢？不在就绪状态的线程当然不在就绪队列中，但是仍通过 ETHREAD 结构中的 ThreadListEntry 链接在其所属进程的线程队列中，这是在创建之初通过 PsInitializeThread()中挂入这个队列的。所以，一个线程，不管是否就绪，其 ETHREAD 数据结构总是挂在其所属进程的线程队列中。不过，到一个线程结束了运行、状态变成 THREAD_STATE_TERMINATED_1 的时候则又是特例，对于这样的线程要通过 PsQueueThreadReap()“收割”这个线程的数据结构。

还有个问题，挂入就绪队列的线程什么时候从队列中脱离出来呢？下面就会看到，当调度一个线程运行时，就把它的 ETHREAD 结构从就绪队列中摘除下来。正因为这样，前面才把状态为就绪的线程又挂回就绪队列。

回到 PsDispatchThreadNoLock()的代码，下面就是调度了。在多处理器 SMP 结构的系统中，有些线程是指定只能在某个或某几个 CPU 上运行的，这种关系叫做 Affinity，即“亲和”，或者也可以说“绑定”。有关的信息以位图的形式记录在具体线程的 KTHREAD 数据结构中，所以在调度时得要明确这是为哪一个 CPU 在做调度。

具体的调度就是按从高到低的次序扫描各个优先级的就绪队列，从中找出第一个愿意在目标 CPU 上运行的线程。这就是程序中那个 for 循环的作用。对于具体优先级的就绪队列，则通过 PsScanThreadList()加以扫描。

[NtYieldExecution() > PsDispatchThread() > PsDispatchThreadNoLock() > PsScanThreadList()]

```
static PETHREAD PsScanThreadList(KPRIORITY Priority, ULONG Affinity)
{
    PLIST_ENTRY current_entry;
    PETHREAD current;
    ULONG Mask;

    Mask = (1 << Priority);
    if (PriorityListMask & Mask)
    {
        current_entry = PriorityListHead[Priority].Flink;
        while (current_entry != &PriorityListHead[Priority])
        {
            current = CONTAINING_RECORD(current_entry, ETHREAD,
                                                    Tcb.QueueListEntry);
            if (current->Tcb.State != THREAD_STATE_READY)
            {
                DPRINT1("%d/%d\n", current->Cid.UniqueThread, current->Tcb.State);
            }
            . . . . . .
            if (current->Tcb.Affinity & Affinity)
            {
               PsRemoveFromThreadList(current);
               return(current);
            }
            current_entry = current_entry->Flink;
        }
    }
    return(NULL);
}
```

就这样，按优先级从高到底的次序对每个就绪队列执行 PsScanThreadList()；一找到合适的线程，循环就结束了。

回到前面 PsDispatchThreadNoLock()的代码，调度的结果无非是这么几种：

- 目标线程 Candidate 就是当前线程本身，这就不需要切换了。将其状态改回 THREAD_STATE_RUNNING，就可以返回了。注意返回前的解锁操作。
- 目标线程 Candidate 就是空转线程 IdleThread，而当前线程不是空转线程。此时将位图 IdleProcessorMask 中对应于当前 CPU 的标志位设成 1，表示空转线程在本 CPU 上运行。

- 目标线程 Candidate 不是空转线程，而当前线程是空转线程。此时此时将位图 IdleProcessorMask 中对应于当前 CPU 的标志位清 0，表示空转线程已不在本 CPU 上运行。
- 目标线程 Candidate 和当前线程都不是空转线程，这是常规的线程调度。
- 没有找到任何可以在此 CPU 上执行的线程，这一定是程序中发生了什么严重的错误，因为空转线程永远都是就绪的，而且每个 CPU 都有一个(每个 CPU 的 KRCB 数据结构中的指针 IdleThread 指向其空转线程)。系统在这种情况下不能继续运行了，所以执行 KEBUGCHECK()。

当然，在正常的情况下总是有线程可以运行的，所以下面就是切换的事了。这里有两步操作，第一步是对 MmUpdatePageDir()的调用；第二步是宏操作 KiArchContextSwitch()，对于 x386 处理器这就是前面的 Ki386ContextSwitch()。

先看对 MmUpdatePageDir()的调用。其第一个参数是 PsGetCurrentProcess()，注意这是指切换前的当前线程所属的进程。第二个参数是 CurrentThread->ThreadsProcess，这却是目标线程所属进程的 EPROCESS 结构指针。第三个参数则是 sizeof(EPROCESS)。调用这个函数的目的是要确保目标线程所属进程的 ETHREAD 数据结构在当前进程的页面映射表中有映射。每个进程都有个页面映射表，而其所在的地址则在该进程的 EPROCESS 结构中。每个进程的 EPROCESS 结构都在物理页面中。但是，在 ReactOS 内核中(估计在 Windows 内核中也是一样)，一个进程的 EPROCESS 结构所占的物理页面却未必映射到别的进程的系统(虚存)空间。换言之，从一个进程的虚存空间可能访问不到别的进程的 EPROCESS 结构。这在平时没有什么问题，但是在一些需要跨进程访问的特殊场合下就有问题了。需要跨进程访问的场合主要就在线程切换的过程中以及需要进程挂靠的时候。

其实，在线程切换的过程中需要访问的并非“新”线程所属进程的整个 EPROCESS 结构，而是作为其一部分的 KPROCESS 结构。回顾一下前面 Ki386ContextSwitch()的代码，就可以看到：

- 为设置 LDTR，需要访问 KPROCESS 结构中的映像 LdtDescriptor[2]。
- KPROCESS 结构中的 IopmOffset。
- 为设置 CR3，需要访问 KPROCESS 结构中的字段 DirectoryTableBase。

目标进程的 EPROCESS 结构所在页面在当前进程的页面映射表中未必有映射，如果不补上这些页面的映射，就会在访问这些数据的时候发生页面异常，所以需要先通过 MmUpdatePageDir()补上这些映射。

注意这里说的是 EPROCESS 结构，而不是 ETHREAD 结构，后者无论在哪一个进程的页面目录中都是有映射的，要不然线程调度就没法做了。

现在已是万事俱备了，下面就是 Ki386ContextSwitch()，这前面已经看过了。

从代码中可以看出，把线程挂入就绪队列的过程和从就绪队列中选择线程的过程都是很简单而直截了当的，实际上就是根据优先级、即 KTHREAD 结构中 Priority 字段的值。有多个相同优先级的就绪线程时，则轮流(Roubd Robin)执行，因为把就绪线程挂入队列时总是挂在尾部。

线程的运行优先级可以通过系统调用加以改变，也可能因为别的原因而受到改变，但是那又属于另一个话题了。