

作者：道哥，10+年的嵌入式开发老兵。

公众号：【[IOT物联网小镇](#)】，专注于：C/C++、Linux操作系统、应用程序设计、物联网、单片机和嵌入式开发等领域。 公众号回复【[书籍](#)】，获取 Linux、嵌入式领域经典书籍。

转载：欢迎转载文章，转载需注明出处。

LDT: 局部描述符表

TSS: 任务状态段

TCB: 任务控制块

x86 系统中的保护模式，给系统的安全性提供了很大的保障，但是在我们之前的文章中，一直都[淡化了特权级别](#)这个概念。

例如：在保护模式下的段选择器，我们一直都只把它看做一个段描述符的["索引号"](#)，用来在 GDT (全局描述符表) 中查找一个段描述符，例如：

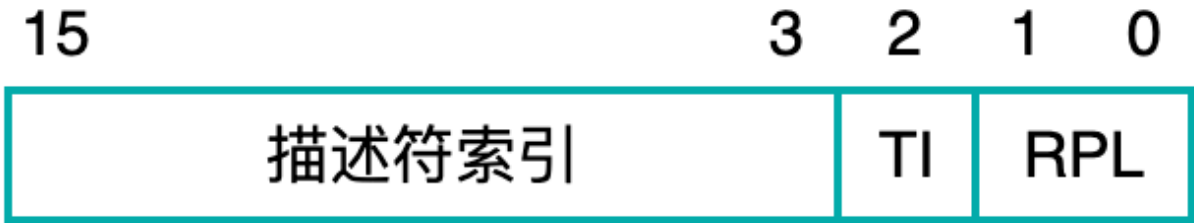


图中：代码段寄存器中的索引号是 4，GDT 中每一个表项占用 4 个字节，于是就在偏移量为 16 的位置，找到了代码段的描述符，进而从描述符中找到代码段的起始地址和长度界限。

数据段、栈段的操作过程也是这样的。

从现在开始，我们需要让用户程序拥有自己私有的描述符表 LDT (Local Descriptor Table)，并且拥有自己的特权级别(总不能让用户程序与操作系统一样，工作在非常高的 0 特权级别)。

因此，我们需要纠正之前的错误：段寄存器中，不仅仅有段的索引号，还有另外两个属性：TI 和 RPL，如下图所示：



段寄存器

- 1. TI 标志位：表示到哪个表中(GDT or LDT)查找描述符;
 - TI = 0: 到 GDT 中查找描述符;
 - TI = 1: 到 LDT 中查找描述符;
 - 2. RPL(Request Privilege Level) 标志位：表示想给段寄存器赋值的请求者(也就是一段代码)，它的特权级别;
- 此时，继续把段寄存器中的内容称作段索引符就不合适了，一般称作：选择子。

LDT：局部描述符表

在上一篇文章中，操作系统把应用程序从硬盘读取到内存中之后，为应用程序创建了三个段描述符，这三个段描述符都放在了 GDT 表中，这是不合理的。

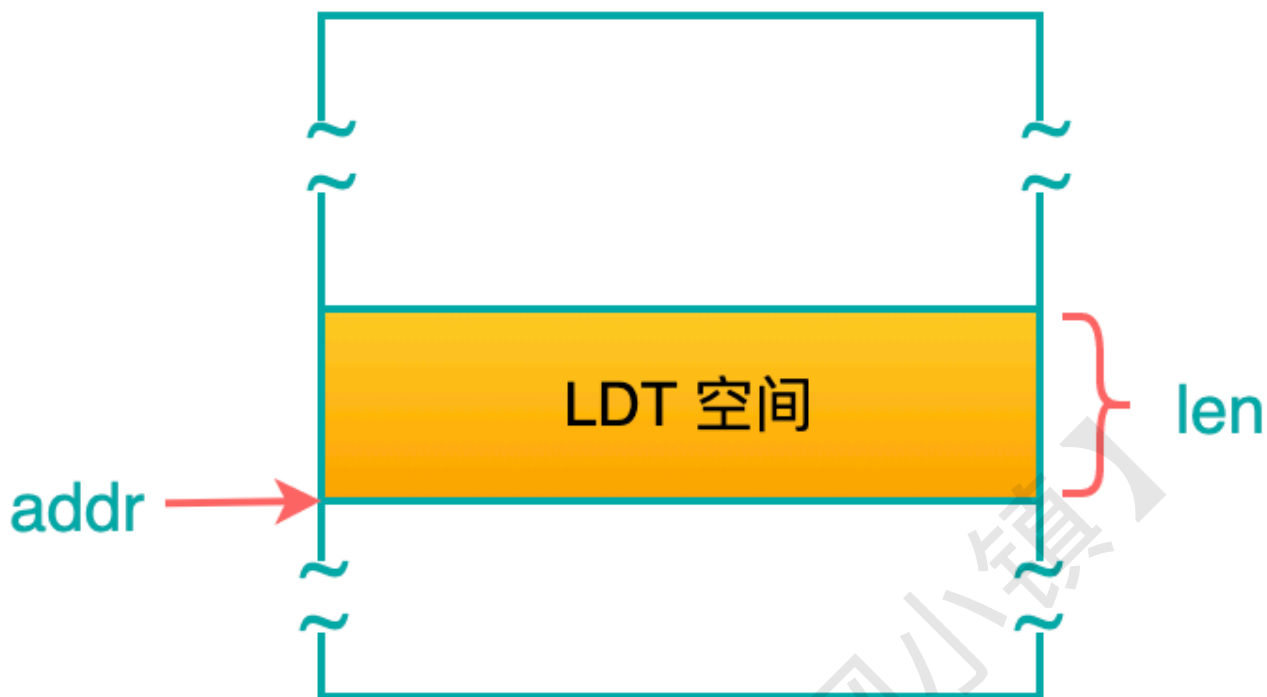
首先，在多任务系统中，应用程序的数量是不确定的，应用程序也会执行结束。

如果把所有应用程序的段描述符都放在 GDT 中，对于操作系统来说，管理这个数据太复杂。

其次，当引入特权级别之后，如果应用程序的段描述符放在 GDT 中，那么就意味着应用程序需要有权限来访问 GDT，而 x86 系统中只有一个 GDT(所以叫做 Global Description Table)，只能被操作系统访问。

因此，操作系统需要为每一个应用程序，单独申请一块空间，用作这个程序自己的段描述附表，称作：LDT(Local Description Table)。

例如：现在系统中有 2 个用户程序: APP1 和 APP2，操作系统在加载每一个应用程序的时候，就会在应用程序自己的内存空间中，申请一块，用作 LDT：



APP1/2 的内存空间

为什么是“应用程序自己的内存空间”？

因为每一个应用程序，都独享 4G 大小的虚拟内存空间。

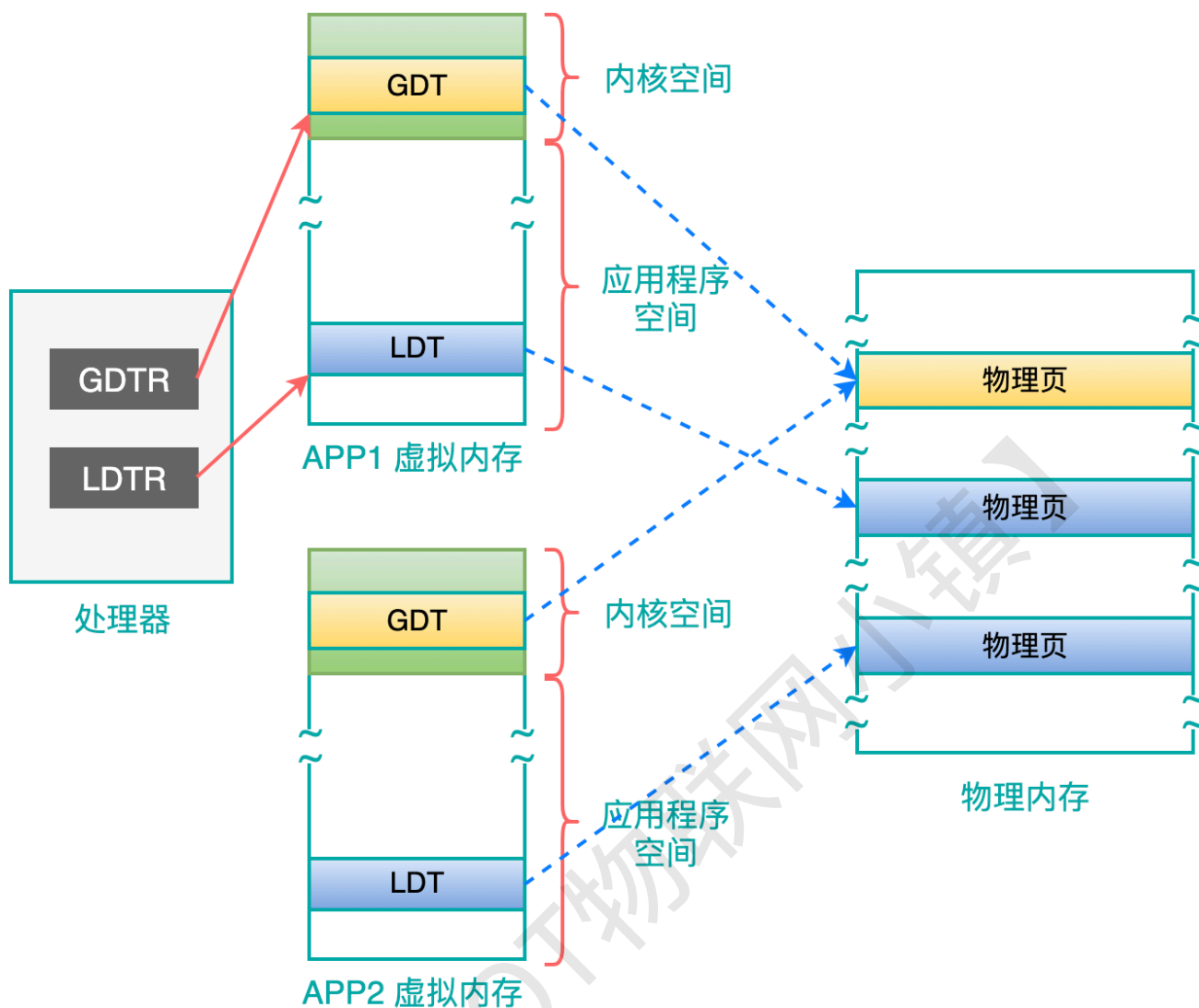
在 LDT 中，存放着当前应用程序自己的段描述符信息，例如：代码段、数据段、栈段。

LDT 所占用的空间也属于内存的一部分，有起始地址和长度界限，因此也需要为它创建一个段描述符，这个描述符就放在 GDT 中。

在 Linux 应用层，我们会严格的区分进程、线程，但是在系统的底层，这样的区分界限已经比较模糊了，用任务 task 来称呼更通用些。

根据刚才的假设，现在系统中有 2 个用户程序，那么处理器怎么知道：当前正在执行的是哪一个应用程序的 LDT 中的代码？

正如处理器中有一个寄存器 GDTR，保存着 GDT 的开始地址和长度，处理器中还有一个寄存器 LDTR，存储着当前正在执行的那个应用程序的 LDT 开始地址和长度：



所有应用程序的虚拟内存的高端地址部分，映射的都是操作系统的内存空间，按照 Linux 中的做法，3G ~ 4G 空间被操作系统使用。

图中的绿色部分，表示操作系统空间(1G)，在分页机制下，它们都映射到相同的物理内存页上(蓝色虚线箭头)。

当操作系统切换到应用程序2时，处理器中的 LDTR 就会被赋值为应用程序2 的 LDT 的线性地址和长度信息。

GDTR 中的内容不变，因为每个应用程序中的 GDT 都是从操作系统“继承”而来的，开始地址和长度都是一样的。

TSS: 任务状态段

顾名思义，任务状态段就是用来存储和恢复任务的状态信息。

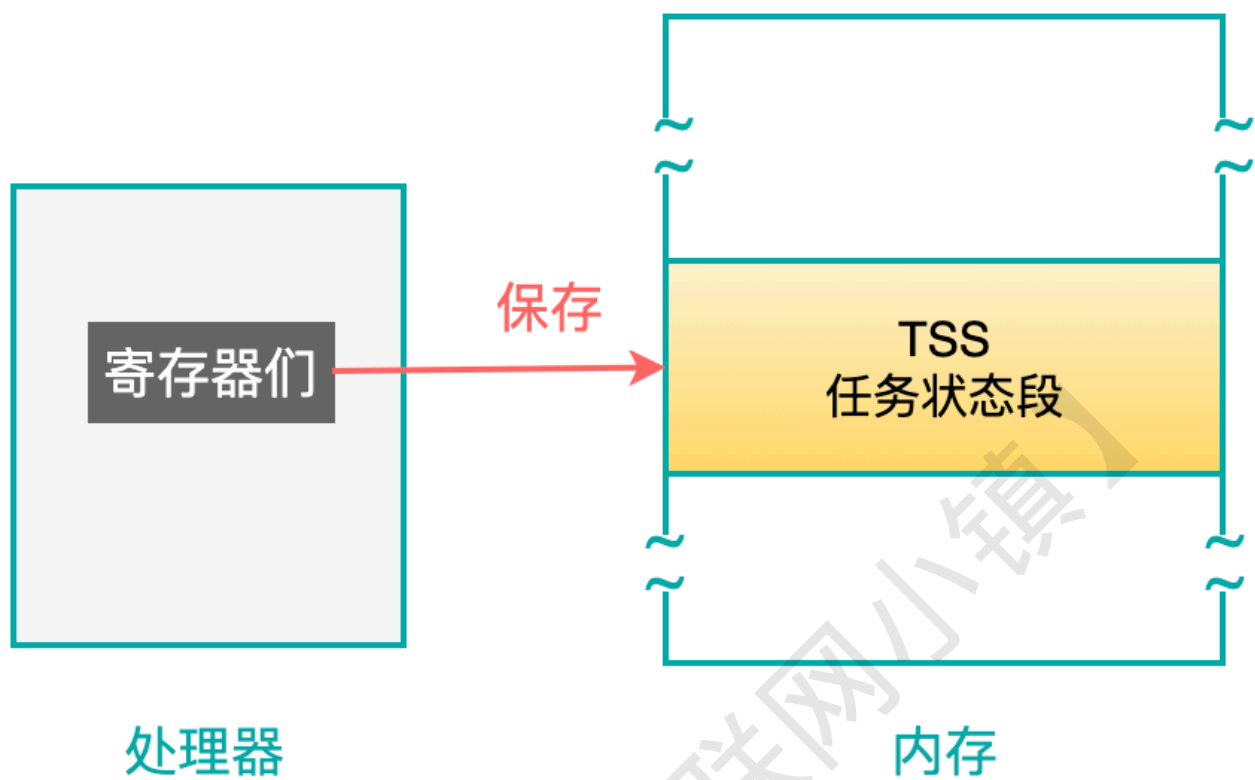
经常听到一个术语：任务上下文。

所谓的上下文，就是体现一个任务正在被执行时的环境信息，主要就是处理器中的各种寄存器内容，也就是下面这张图中的寄存器们：

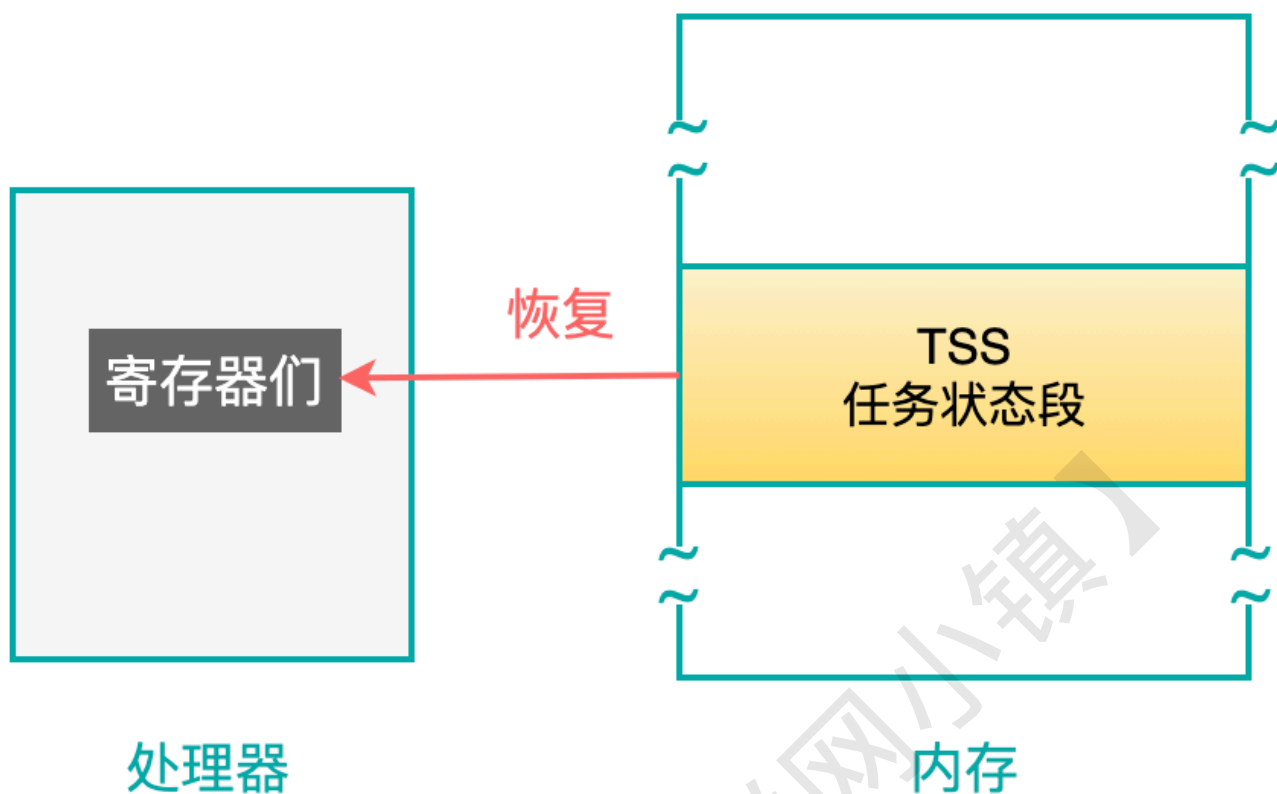
31	15	0
I/O Map Base Address		T 100
	LDT Segment Selector	96
	GS	92
	FS	88
	DS	84
	SS	80
	CS	76
	ES	72
EDI		68
ESI		64
EBP		60
ESP		56
EBX		52
EDX		48
ECX		44
EAX		40
EFLAGS		36
EIP		32
CR3 (PDBR)		28
	SS2	24
ESP2		20
	SS1	16
ESP1		12
	SS0	8
ESP0		4
	Previous Task Link	0

这张图反映了一个任务上下文的所有寄存器信息。

当任务被调度器中止执行之前，需要把这些寄存器中的值都保存下来，相当于做一个快照。



当这个任务以后又被恢复执行时，再把这个快照中保存的信息，原样的赋值给图中的所有寄存器，这样就称作**恢复任务上下文**，这个任务就上次被中止的地方继续执行(因为指令指针寄存器 EIP 被恢复了)。



就如同 LDT 一样，TSS 也是操作系统为应用程序分配的一块内存空间，只不过这块空间是位于操作系统的势力范围内，只能由操作系统来操作。

TSS 也有起始地址和长度界限，也需要为它在 GDT 中创建一个段描述符。

与 LDT 类似，在处理器中也有一个寄存器 TR，用来指向当前正在执行的那个任务的 TSS。

当进行任务切换的时候：

1. 首先，把处理器中的寄存器内容，存储到 TR 寄存器指向的 TSS 段中(即将被停止的任务)；
2. 然后，把新的任务的 TSS 段中的内容，复制到处理器的各寄存器中，并且把 TSS 地址赋值给 TR 寄存器；

TCB: 任务控制块

任务控制块，可以说是系统中用来管理任务的最重要的数据结构了，操作系统用来管理任务的所有信息都可以放在这里。

看一下 Linux 2.6 内核代码中的结构体：struct task_struct{ ... }，就知道 TCB 有多复杂了，有些书籍上也称之为 PCB(Process Control Block，进程控制块)。

在这个结构中，一些常用的信息包括：

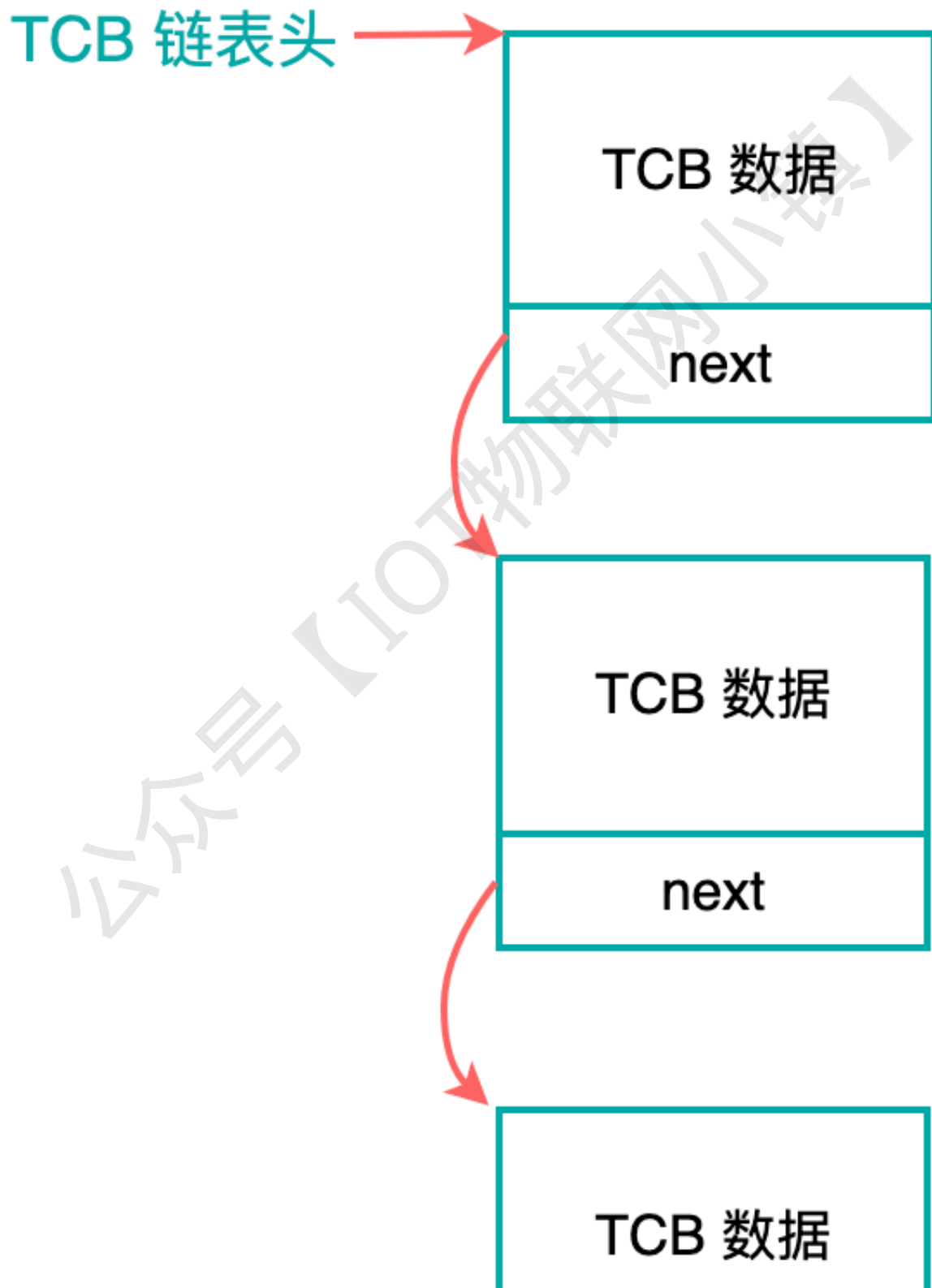
1. 程序的加载地址；
2. 任务的优先级；
3. 任务的当前状态；
4. 任务打开的一些资源：网络、文件设备等；

。 。 。

需要注意的是：上面的 LDT、TSS，是 x86 处理器中设计的运行机制，是处理器要求这样的。

而 TCB 不是处理器要求的，它是操作系统的实现者自己来构建的，因此可以根据自己的需要来进行设计。

每一个应用程序需要一个 TCP 结构，所有的 TCB 结构就可以构成一个链表，便于操作系统来管理。



NULL

比如：在发生任务切换的时候，就可以顺着链表头，一次扫描链表上的每一个 TCB 节点。

如果找到了当前正在被执行(即将被中止)的任务，就把这个任务的状态标记为暂停，并移动到链表的末尾，然后把链表头部的第一个处于 ready 状态的任务，加载到处理器中去执行。

当然，Linux 系统中的处理过程更为复杂，它把每一个任务按照优先级放在不同的等待队列中，然后利用哈希桶算法来查找任务。

----- End -----

x86 处理器中的这三个概念，对于理解任务切换非常重要。

写到这里，我总是觉得以上的文字描述还是有点朦朦胧胧，也许是自己还需要进一步的理解其中的脉络。

就先这样吧，以后想到更好的描述方式了再与大家分享，谢谢！

推荐阅读

- 【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
- 【2】一步步分析-如何用C实现面向对象编程
- 【3】原来gdb的底层调试原理这么简单
- 【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、C语言、Linux操作系统、应用程序设计、物联网



微信搜一搜



IOT物联网小镇

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言 Linux 操作系统、应用程序开发设计



扫码关注公众号



道哥 个人微信

喜欢请**分享**，满意点个**赞**，最后点**在看**。