

# 第一章 概述

## 1.1 嵌入式系统概述

当今时代，人们的生活越来越依赖基于计算机技术和数据通信技术的电子产品，因此，有人说，当今时代是电子产品时代；也有人说，当今时代是互联网时代；还有人说，当今时代是e时代。这些都充分说明了电子产品和互联网技术给人们的生活带来的改变。但这些说法都有些偏颇，一个更接近本质的说法是“当今时代，是嵌入式系统时代”。

嵌入式系统可以简单地理解为“为完成一项功能而开发的、由具有特定功能的硬件和软件组成的一个应用产品或系统”。嵌入式系统在我们的生活中到处可见，例如，手机、PDA、家里的数字电视机、全自动洗衣机等，都是嵌入式系统。当然，在我们日常生活接触不到的领域中，嵌入式系统也被广泛应用。例如，应用于通信网络中的电话交换机、光传输分叉/复用设备、互联网路由器等，都是嵌入式系统的实例。这些实例都有一个共同的特点，那就是“具备特定的用途”。比如，手机只能用于完成移动通信（移动通话、移动短信息等），而不具备数字电视的功能，同样地，数字电视只具备数字电视信号接收、解码和播放功能，以及相关的一些简单附加功能，而不具备洗衣机的功能，等等。因此，嵌入式系统一个最基本的特点，就是“功能专一”。

一般情况下，嵌入式系统是由嵌入式硬件和嵌入式软件两部分组成的。嵌入式硬件，是由完成嵌入式系统功能所需要的机械装置、数字芯片、光/电转换装置等组成，嵌入式硬件决定了嵌入式系统的功能集合，即嵌入式系统的最终功能。嵌入式软件则是附加在嵌入式硬件之上的，驱动嵌入式硬件完成特定功能的逻辑指令。嵌入式软件可以非常简单，比如，在一些简单的自动控制洗衣机中，软件部分可能只有数百行汇编代码，系统功能基本上由硬件完成，软件仅仅起到辅助功能。嵌入式软件也可以非常复杂，比如，手机、大型通信设备等嵌入式系统，软件部分往往由数十万行，甚至数百万行代码组成，这些系统的大部分功能都是由软件逻辑实现的。通过分析这些嵌入式系统，可以发现一个规律，那就是嵌入式软件所占比重越高的嵌入式系统，其灵活性越好，功能也越强大，这很容易理解，因为软件比重大的系统中，大部分功能是由软件完成的，通过迭加更多的软件，就可以实现更多的功能。相反，若一种嵌入式系统由硬件占主导地位，则在这种系统上增加新的功能或配置将非常不方便，因为需要更换硬件。

对于嵌入式系统的软件，可以进一步分为嵌入式操作系统和嵌入式应用软件。其中，嵌入式操作系统是系统软件，是直接接触硬件的一层软件，嵌入式操作系统为应用软件提供了一个统一的接口，屏蔽了不同硬件之间的差别，使得应用软件的开发和调试变得

十分方便。嵌入式应用软件则是真正完成系统功能的软件。当然，这两种软件并不是所有嵌入式系统都必需的，在一些简单的嵌入式系统中，比如在微波炉、自动控制洗衣机等嵌入式系统中，软件功能十分简单，这样就没有必要采用嵌入式操作系统，但在一些复杂的嵌入式系统中，比如在互联网路由器中，嵌入式操作系统则是必不可少的部件，因为这些嵌入式系统的应用软件十分复杂，若不采用嵌入式操作系统来进行支撑，其开发工作将十分困难，甚至无法完成。

总之，嵌入式系统就是由嵌入式硬件和嵌入式软件组成的，具备特定功能的计算机系统，其中，嵌入式软件又可进一步分为嵌入式操作系统和嵌入式应用软件，如图 1-1 所示。

**错误！**

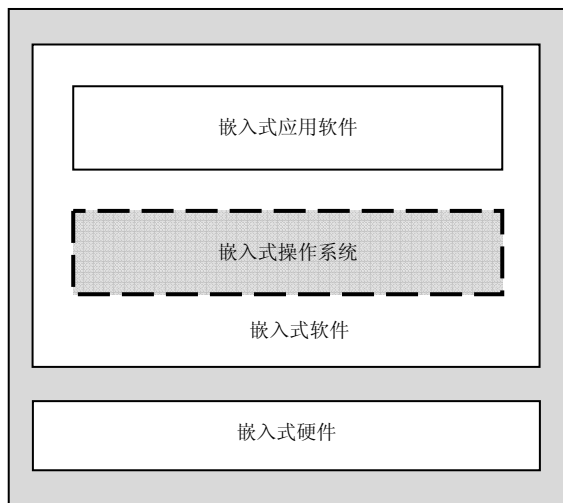


图 1-1 嵌入式系统软硬件之间的关系

嵌入式操作系统是整个嵌入式软件的灵魂，起到承上启下（连接嵌入式硬件和嵌入式应用软件）的作用，而且往往也是嵌入式软件中最复杂的部分。虽然复杂，嵌入式操作系统的功能接口却相对标准化和统一，功能差异很大的嵌入式系统，往往可以采用相同的嵌入式操作系统来进行设计，比如，一台复杂的数字控制机床的控制系统与一架军用飞机的控制系统，可能采用了相同的嵌入式操作系统，仅仅是具体的应用软件不同。因此，嵌入式操作系统可以被理解为通用软件，不同的嵌入式操作系统，除了性能上的差异和实现细节上的差异之外，功能部分往往是相同的。在本书中，我们介绍一个嵌入式操作系统的功能及其功能的实现细节。

## 1.2 嵌入式操作系统概述

从上面的描述中我们知道，嵌入式操作系统是嵌入式系统中的软件部分，且是软件部分的核心内容。嵌入式操作系统在本质上也是一个操作系统，其一些概念与通用计算机操作系统是一致的，但由于应用环境的不同，嵌入式操作系统与通用操作系统有一些区别，且嵌入式操作系统本身具备一些通用操作系统所不具备的特性。在本节中，我们对嵌入式操作系统本身具备的一些特点，以及与通用操作系统的区别进行简单描述。

### 1.2.1 嵌入式操作系统特点

一个典型的嵌入式操作系统应该具备下列特点：

#### 1. 可裁剪性

可裁剪性是嵌入式操作系统最大的特点，因为嵌入式操作系统的目标硬件配置差别很大，有的硬件配置非常高档，有的却因为成本原因，硬件配置十分紧凑，嵌入式操作系统必须能够适应不同的硬件配置环境，具备较好的可裁剪性。在一些配置高、功能要求多的情况下，嵌入式操作系统可以通过加载更多的模块来满足这种需求；而在一些配置相对较低，功能单一的情况下，嵌入式操作系统必须能够通过裁剪的方式，把一些不相关的模块裁剪掉，只保留相关的功能模块。为了实现可裁剪，在编写嵌入式操作系统的时候，就需要充分考虑，进行仔细规划，把整个操作系统的功能进行细致的划分，每个功能模块尽量以独立模块的形式来实现。

对于裁剪的具体实现，可通过两种方式。一种方式是把整个操作系统功能分割成不同的功能模块，进行独立编译，形成独立的二进制可加载映像，这样就可以根据应用系统的需要，通过加载或卸载不同的模块来实现裁剪。另外一种方式，是通过宏定义开关的方式来实现裁剪，针对每个功能模块，定义一个编译开关（`#define`）来进行标志。若应用系统需要该模块，则在编译的时候，定义该标志，否则取消该标志，这样就可以选择需要的操作系统核心代码，与应用代码一起编联，实现可裁剪的目的。其中，第一种方式是二进制级的可裁剪方式，对应用程序更加透明，且无需公开操作系统的源代码，第二种方式则需要应用程序详细了解操作系统的源代码组织。

## 2. 与应用代码一起连接

嵌入式操作系统的另外一个重要特点，就是与应用程序一起，连接成一个统一的二进制模块，加载到目标系统中。而通用操作系统则不然，通用操作系统有自己的二进制映像，可以自行启动计算机，应用程序单独编译连接，形成一个可执行模块，并根据需要在通用操作系统环境中运行。

## 3. 可移植性

通用操作系统的目标硬件往往比较单一，比如，对于 UNIX、Windows 等通用操作系统，只考虑几款比较通用的 CPU 就可以了，比如 Intel 的 IA32 和 Power PC。但在嵌入式开发中却不同，存在多种多样的 CPU 和底层硬件环境，光 CPU，流行的可能就会达到十几款。嵌入式操作系统必须能够适应这种情况，在设计的时候充分考虑不同底层硬件的需求，通过一种可移植的方案来实现不同硬件平台上的方便移植。比如，在嵌入式操作系统设计中，可以把硬件相关部分代码单独剥离出来，在一个单独的模块或源文件中实现，或者增加一个硬件抽象层，来实现不同硬件的底层屏蔽。总之，可移植性是衡量一个嵌入式操作系统质量的重要标志。

## 4. 可扩展性

嵌入式操作系统的另外一个特点，就是具备较强的可扩展性，可以很容易地在嵌入式操作系统上扩展新的功能。比如，随着 Internet 的快速发展，可以根据需要，在对嵌入式操作系统不做大量改动的情况下，增加 TCP/IP 协议功能或 HTTP 协议解析功能。这样必然要求嵌入式操作系统在设计的时候，充分考虑功能之间的独立性，并为将来的功能扩展预留接口。

# 1.2.2 嵌入式操作系统与通用操作系统的区别

## 1. 地址空间上的区别

一般情况下，通用操作系统，充分利用了 CPU 提供的内存管理机制（MMU 单元），实现了一个用户进程（应用程序）独立拥有一个地址空间的功能，比如，在 32 位 CPU 的硬件环境中，每个进程都有自己独立的 4GB 的地址空间。这样每个进程之间相互独立，互不影响，即一个进程的崩溃，不会影响另外的进程，一个进程地址空间内的数据，不能被另外的进程引用。嵌入式操作系统多数情况下不会采用这种内存模型，而是操作系统和应用程序共用一个地址空间，比如，在 32 位硬件环境中，操作系统和应用程序共享 4GB 的地址空间，不同应用程序之间可以直接引用数据。这类似于通用操作系统上的线程模型，即一个通用操作系统上的进程，可以拥有多个线程，这些线程之间共享进程的地址空间。

这样的内存模型实现起来非常简单，且效率很高，因为不存在进程之间的切换（只存在线程切换），而且不同的应用之间可以很方便地共享数据，对于嵌入式应用来说，是十分合适的。但这种模型的最大缺点就是无法实现应用之间的保护，一个应用程序的崩溃，可能直接影响到其他应用程序，甚至操作系统本身。但在嵌入式开发中，这个问题却不是问题，因为在嵌入式开发中，整个产品（包括应用代码和操作系统核心）都是由产品制造商开发完成的，很少，需要用户编写程序，因此整个系统是可信的。而通用操作系统之所以实现应用之间的地址空间独立，一个立足点就是应用程序的不可信任性。因为在一个系统上，可能运行了许多不同厂家开发的软件，这些软件良莠不齐，无法信任，所以采用这种保护模型是十分恰当的。

## 2. 内存管理上的区别

通用的计算机操作系统为了扩充应用程序可使用的内存数量，一般实现了虚拟内存功能，即通过 CPU 提供的 MMU 机制，把磁盘上的部分空间当做内存使用（详细信息请参考本书“Hello China 的内存管理机制”一章）。这样做的好处是可以让应用程序获得比实际物理内存大得多的内存空间，而且还可以把磁盘文件映射到应用程序的内存空间，这样应用程序对磁盘文件的访问，就与访问普通物理内存一样了。

但在嵌入式操作系统中，一般情况下不会实现虚拟内存功能，这是因为：

（1）一般情况下，嵌入式系统没有本地存储介质，或者即使有，数量也很有限，不具备实现虚拟内存功能的基础（即强大的本地存储功能）；

（2）虚拟内存的实现，是在牺牲效率的基础上完成的，一旦应用程序访问的内存内容不在实际的物理内存中，就会引发一系列的操作系统动作，比如引发一个异常、转移到核心模式、引发文件系统读取操作等一系列动作，这样会大大降低应用程序的执行效率，使得应用程序的执行时间无法预测，这在嵌入式系统开发中是无法容忍的。

因此，权衡利弊，嵌入式操作系统首选，是不采用虚拟内存管理机制，这也是嵌入式操作系统与通用的操作系统之间的一个较大的区别。

## 3. 应用方式上的区别

通用的操作系统在使用之前必须先进行安装，安装包括检测并配置计算机硬件、安装并配置硬件驱动程序、配置用户使用环境等过程，这个过程完成之后，才可以正常使用操作系统。但嵌入式操作系统则不存在安装的概念，虽然驱动硬件、管理设备驱动程序也是嵌入式操作系统的主要工作，但与普通计算机不同，嵌入式系统的硬件都是事先配置好的，其驱动程序、配置参数等往往与嵌入式操作系统连接在一起，因此，嵌入式操作系统不必自动检测硬件，因而也无需存在安装的过程。

除了上述特点与区别外，嵌入式操作系统还有一些其他自身特点，在此不再详述，有兴趣的读者可参阅相关资料。

### 1.2.3 嵌入式实时操作系统

另外一个需要提及的概念，就是嵌入式实时操作系统。嵌入式实时操作系统也是嵌入式操作系统的一种，顾名思义，嵌入式实时操作系统一般应用于对时间要求十分苛刻的场合，比如高精度的数字控制机床、通信卫星控制系统等。嵌入式实时操作系统对外部事件的响应时间是有严格控制的，一般有一个底限，在这个底限之内，需要对外部发生的事件进行响应，这样嵌入式实时操作系统在设计的时候，必须充分考虑这些要求。

但需要说明的是，一个实时系统并不是由嵌入式实时操作系统自身决定的，而是由嵌入式硬件、嵌入式操作系统、嵌入式应用软件等共同决定的，单一因素，比如嵌入式操作系统无法决定整个系统的实时性，这很容易理解。

还有一种对嵌入式操作系统的实时性进行描述的说法叫做“半实时操作系统”。这种操作系统不像严格的实时操作系统（姑且叫做硬实时操作系统）对事件的相应有一个严格的底限，但又与普通操作系统对外部事件相应的不确定性有所区别，介于两者之间。这样的操作系统可以满足大部分嵌入式应用的需求，而且当前情况下，一般商用的操作系统都是这种“半实时操作系统”。本书介绍的“Hello China”操作系统也属于半实时操作系统。

操作系统、嵌入式操作系统、实时嵌入式操作系统和半实时操作系统的关系，如图1-2所示。

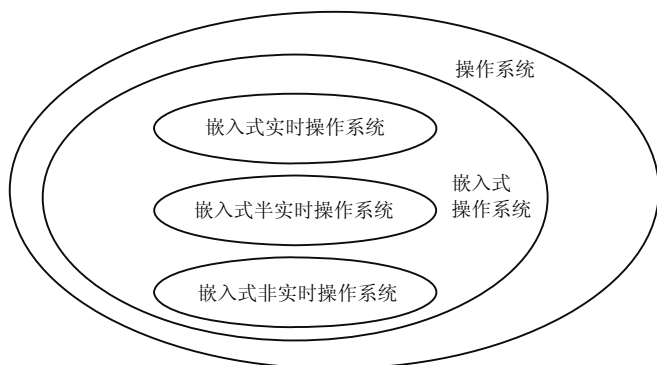


图 1-2 操作系统之间的关系

## 1.3 操作系统的基本概念

为了便于后续内容的理解，我们简单介绍与操作系统相关的几个概念。

1.3.1 微内核与大内核

微内核与大内核是操作系统设计中的两种不同的思想，这与 CPU 的设计中 RISC 和 CISC 构架类似。其中，微内核的思想是，把尽量少的操作系统机制放到内核模块中实现，而把尽量多的操作系统功能以单独进程或线程的方式实现，这样便于操作系统体系结构的扩展。比如，一个常见的设计思路就是，把进程（或线程）调度、进程间通信机制（IPC）与同步、定时功能、内存管理功能、中断调度等功能放到内核中实现，由于这些功能需要的代码量不是很大，所以可使得内核的尺寸很小。另外，把操作系统必须实现的文件系统、设备驱动程序、网络协议栈、IO 管理器等功能作为单独的进程或任务来实现，用户应用程序在需要这些功能的时候，通过核心提供的 IPC 机制（比如消息机制）向这些服务进程发出请求，即一种典型的客户—服务器机制。

这种微内核的实现思路有很明显的优势，比如体系结构更加清晰，扩展性强等，而且由于内核保持很小，移植性（不同 CPU 之间的移植）也很强。但此思路也有很大的弊端，其中最大的一个弊端就是效率相对低下，因为系统调用等服务都是通过 IPC 机制来间接实现的，若服务器进程繁忙，对于客户的请求可能无法及时响应。因此这种微内核的设计方式，不太适合嵌入式操作系统的设计，因为效率是嵌入式操作系统追求的最主要目标。

图 1-3 示意了微内核的设计思路。

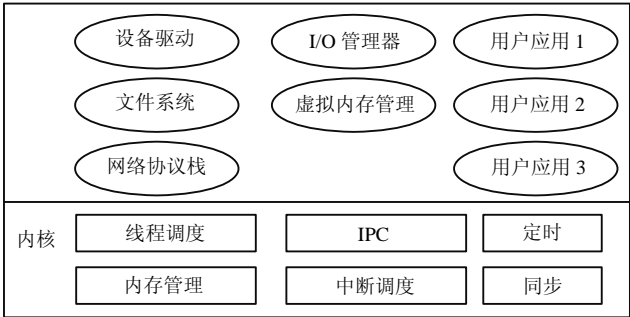


图 1-3 微内核操作系统的设计思想

大内核则是一种相反的设计思路，这种思想是，把尽可能多的操作系统功能拿到内核模块中实现，在操作系统加载的时候，把这些内核模块加载到系统空间中。由于这些系统功能是静态的代码，不像微内核那样作为进程实现，而且为这些代码直接在调用进程的空间中运行，不存在发送消息、等待消息处理、消息处理结果返回等延迟，因此调用这些功能代码的时候，效率特别高。因此，在追求效率的嵌入式操作系统开发中，这种大内核模型是合适的。

但大内核也有一些弊端，最明显的就是内核过于庞大，有时候会使得它的扩展性不好（这可以通过可动态加载模块来部分解决）。但在嵌入式操作系统开发中，这种弊端表现得不是很明显。图 1-4 示意了大内核的开发思想。

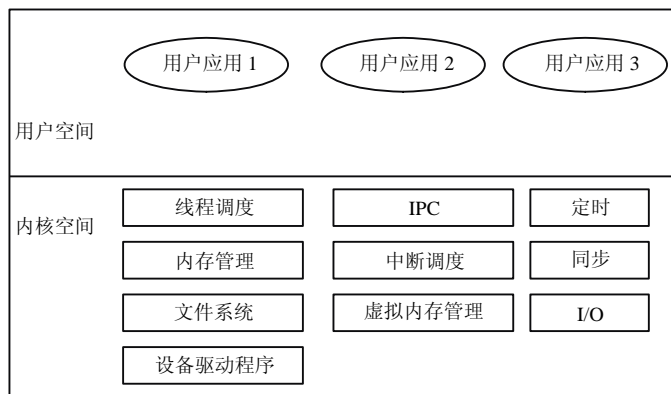


图 1-4 大内核操作系统的设计思想

### 1.3.2 进程、线程与任务

一般情况下，描述操作系统的任务管理机制时存在三个说法。

- **进程** 所谓进程，是一个动态的概念。一个可执行模块（可执行文件）被操作系统加载到内存，分配资源，并加入到就绪队列后，就形成了一个进程。一般情况下，进程有独立的内存空间（比如在典型的 PC 机操作系统中，如果目标 CPU 是 32 位则一个进程就有独立的 4GB 的虚拟内存空间），如果不通过 IPC 机制，进程之间是无法交互任何信息的，因为进程之间的地址空间是独立的，不存在重叠的部分；
- **线程** 一般情况下，线程是 CPU 可感知的最小的执行单元，一个进程往往包含多个线程，这些线程共享进程的内存空间，线程之间可以直接通过内存访问的方式进行通信，线程之间共享同进程的全局变量，但每个线程都有自己的堆栈和硬件寄存器；
- **任务** 概念同线程类似，但与线程不同的是，任务往往是针对没有进程概念的操作系统来说的，比如嵌入式操作系统。这些操作系统没有进程的概念，或者说整个操作系统就是一个进程，这种情况下，任务便成了操作系统中最直接的执行单位。另外一个说法就是，任务往往是一个无限循环，操作系统启时，任务随之启动，然后一直运行到操作系统结束为止。

目前情况下在 Hello China 的实现中是没有进程概念的，但却存在多个执行线索，因此，用任务来描述这些执行线索是最合适的。但考虑到这些执行线索并不一定是无限循

环，而是跟通用操作系统的线程概念一致，可以根据需要创建，运行结束后自动销毁，因此我们也以“线程”来称呼系统中的执行线索，为了区别通用操作系统中普通的用户线程（用户进程中的线程），我们把 Hello China 中的执行线索叫做“核心线程（Kernel Thread）”，或简单称为“线程”。

### 1.3.3 可抢占与不可抢占

在操作系统对进程（或线程）的调度策略中存在两种不同的调度方式：可抢占方式和不可抢占方式。在可抢占方式下，操作系统以时间片（Time Slice）为单位来完成进程调度。针对每个进程，一次只能运行一个或几个时间片，一旦时间片消耗完毕，操作系统就会强行暂停其运行，而选择其他重新获得时间片的进程投入运行。在不可抢占方式下，进程会一直运行，操作系统不会强制剥夺其运行权，而是等待其自行放弃运行为止（或者是发生系统调用）。一般情况下，系统调用时，操作系统才进行新一轮调度。

这两种方式在嵌入式操作系统中都被大规模地应用。但很显然，可抢占调度机制具备更好的实时性，因此在一些对实时性要求很高的场合，一般采用可抢占调度方式。但可抢占调度方式会引发另外一个问题，即多个进程之间对共享资源访问的同步问题。因此，在实现可抢占调度的同时，必须实现互斥机制、同步机制等操作系统机制来解决可抢占性所带来的问题。不可抢占操作系统不存在“资源竞争”的问题，但也存在同步问题。Hello China 操作系统的调度机制是基于可抢占方式进行的。

### 1.3.4 同步机制

有的情况下，线程之间的运行是相互影响的，比如对共享资源的访问就需要访问共享资源的线程相互同步，以免破坏共享资源的连续性。下列线程同步机制可被操作系统实现，用来完成线程的同步和共享资源的互斥访问。

#### 1. 事件（Event）

事件对象是一个最基础的同步对象，事件对象一般处于两种状态：空闲状态和占用状态。一个内核线程可以等待一个事件对象，如果一个事件对象处于空闲状态，那么任何等待该事件对象的线程都不会阻塞。相反，如果一个事件对象处于占用状态，那么任何等待该对象的线程都进入阻塞状态（Blocked）。

一旦事件对象的状态由占用变为空闲，那么所有等待该事件对象的线程都被激活（状态由 Blocked 改变为 Ready，并被插入 Ready 队列），这一点与下面讲述的互斥体不同。

#### 2. 信号量（Semaphore）

信号量也是最基础的同步对象之一，一般情况下，信号量维护一个计数器，假设为 N，每当一个内核线程调用 WaitForThisObject 等待该信号量对象时，N 就减一，如果 N

小于零，那么等待的线程将被阻塞，否则继续执行。

### 3. 互斥体 (Mutex)

互斥体是一个二元信号量，即  $N$  的值为 1，这样最多只有一个内核线程占有该互斥体对象，当这个占有该互斥体对象的线程释放该对象时，只能唤醒另外一个内核线程，其他的内核线程将继续等待。

注意互斥体对象与 Event 对象的不同，在 Event 对象中，当一个占有 Event 对象的线程释放该对象时，所有等待该 Event 对象的线程都将被激活，而 Mutex 对象，则只有一个内核线程被激活。

### 4. 内核线程对象 (KernelThreadObject)

内核线程对象本身也是一个互斥对象，即其他内核线程可以等待该对象，从而实现线程执行的同步。但与普通的互斥对象不同的是，内核线程对象只有当状态是 `TERMINAL` 时才是空闲状态，即如果任何一个线程等待一个状态是非 `Terminal` 的内核线程对象，那么将会一直阻塞，直到等待的线程运行结束（状态修改为 `Terminal`）。

### 5. 睡眠

一个运行的线程可以调用 `Sleep` 函数而进入睡眠状态 (`Sleeping`)，进入睡眠状态的线程将被加入 `Sleeping` 队列。

当睡眠时间(由 `Sleep` 函数指定)到达时，系统将唤醒该睡眠线程(修改状态为 `Ready`，并插入 `Ready` 队列)。

### 6. 定时器

另外一个内核线程同步对象是定时器。定时器是操作系统提供的最基础服务之一，比如线程可以调用 `SetTimer` 函数设置一个定时器，当设置的定时器到时后，系统会给设置定时器的线程发送一个消息。与 `Sleep` 函数不同的是，内核线程调用 `Sleep` 函数后将进入阻塞状态，而调用 `SetTimer` 之后，线程将继续运行。

## 1.4 Hello China 概述

Hello China 是作者开发的一个嵌入式操作系统，目前版本是 V1.5。该操作系统具备一个嵌入式操作系统应该具备的全部核心功能，比如多任务（线程）、线程同步机制、定时机制、中断调度机制、线程睡眠、内存管理、虚拟内存管理等，而且还构架了一个驱动程序支持框架，并提供了一组输入输出 (IO) 接口，设备驱动程序的开发只要遵循 Hello China 提供的构架，将会非常容易，而且对应用程序来说，是透明的。

Hello China V1.5 的开发是在个人计算机 (PC) 硬件平台上完成的。个人计算机就是

一个复杂的嵌入式硬件平台，在 PC 上开发嵌入式操作系统，不仅可以降低成本，节省购买标准开发板的费用，还可以接触到更多的硬件，硬件的扩展也更方便。在 PC 上完成操作系统的开发后，可以通过少量修改，很容易地把操作系统移植到特定的嵌入式硬件平台上。

在本节中，我们对 Hello China 的功能特点、开发环境以及在 PC 机上的使用方式做一个初步介绍，详细的功能特点及其实现、开发环境的详细信息，请参考本书后边章节以及附录。

### 1.4.1 Hello China 的功能特点

当前版本的 Hello China（Version 1.5）具备下列功能：

1. 多线程，Hello China 基于多线程模型可以同时运行多个线索。在嵌入式开发中，可以通过创建多个线程的方式来实现多任务处理；
2. 可抢占调度，线程的调度方式采用了可抢占的方式，这样可使得系统的响应时间非常短暂，对于关键的任务（优先级高的线程）能够尽快地得以运行。当前版本中，Hello China 总共支持 16 个不同的线程优先级；
3. 任务同步，Hello China 实现了完善的任务同步机制，包括事件对象、定时器、线程延迟（睡眠）、核心线程对象等功能，可以很容易地完成多个线程之间的同步运行；
4. 共享资源互斥访问，通过互斥体（MUTEX）、信号量（Semaphore）等核心对象可以实现多个线程之间的共享资源互斥访问；
5. 内存管理，Hello China 实现了完善的内存管理机制，包括物理内存的申请、释放，基于页面的物理内存管理，基于 IA32 构架 MMU 的虚拟内存管理，以及应用程序本地堆（Heap）等功能，还实现了标准 C 运行期库的 malloc、free 等函数来供应用程序直接调用来分配内存。另外，对于基于 PCI 总线的硬件设备，Hello China 还提供了一组内存管理接口，使得设备驱动程序很容易的把设备本地缓冲内存映射到 CPU 的内存空间中，从而完成设备的直接访问；
6. 定时机制，Hello China 实现了毫秒级的定时器机制，一个线程可以通过系统调用 SetTimer 来设定一个定时器，在定时器时间到达后，操作系统会向该线程发送一个消息或调用一个回调函数；
7. 完善的消息机制，每个线程具备一个本地消息队列，其他线程（或操作系统）可以通过系统调用向某个特定的线程发送消息，从而完成线程之间的通信；
8. 中断调度机制，Hello China V1.0 在实现的时候充分考虑了不同 CPU 的中断机制，采用了一种中断向量加中断链表的中断调度机制，可以适应 Intel 等基于中断向量组机制的 CPU，也可以适应 Power PC 等基于单中断向量的 CPU；

9. PCI 总线支持, Hello China 当前版本的实现中, 可以对系统中的单条 PCI 总线进行列举, 从而发现 PCI 总线上的所有物理设备, 并为发现的每个物理设备创建一个管理结构与之对应, 这样设备驱动程序就无需自行检测总线, 只需要向操作系统提出申请, 操作系统就可根据设备 ID, 把设备配置信息传递给驱动程序。这样的体系结构使得设备得以集中管理, 资源得以集中分配;

10. 完善的驱动程序支持框架, 定义了一个通用的设备驱动程序接口规范以及一组应用程序访问驱动程序的接口函数和一个 IO 管理器对象, 使得不论是驱动程序的开发, 还是驱动程序的访问, 都十分方便;

11. 基于 PC 的一些辅助功能, 比如键盘驱动程序、屏幕驱动程序、一个简单的命令行界面等, 另外, 还提供了一组针对 PC 的输入/输出编程接口, 可以直接基于 Hello China 开发 PC 上的应用程序;

12. 一组附加的应用程序, 比如硬件端口读写程序、系统信息诊断程序等, 可供程序开发者作为调试使用。

除此之外, Hello China 还提供了其他操作系统相关的服务, 并初步定义了网络协议栈框架 (尚未实现)。随着本书介绍的深入, 这些特点和服务将会被一一介绍。

### 1.4.2 Hello China 的开发环境

操作系统的开发涉及到各种各样的功能模块, 比如引导功能模块、初始化功能模块、硬件驱动功能模块以及系统核心等, 这些功能模块很难使用同一个开发环境进行代码的编译和编写, 因此, 在 Hello China 的开发过程中, 针对不同的功能模块, 使用了不同的开发环境, 主要有:

- 针对引导功能和硬件驱动程序 (比如键盘驱动程序和字符显示驱动程序) 采用汇编语言编写, 使用 **NASM 编译器** 编译;
- 针对操作系统核心, 为了提高移植性和开发效率, 采用 C 语言编写, 采用 **Microsoft Visual C++** 作为代码的编译和编写环境;
- 由于上述开发环境最终形成的目标格式有时候跟预期的格式不一致, 于是采用 Microsoft Visual C++ 编写了 **目标处理工具**, 这套工具对上述编译器形成的目标文件进行进一步处理, 形成计算机可以直接加载并运行的二进制模块。

Hello China 在开发过程中涉及到的开发环境和开发工具比较多, 但该操作系统的核心代码, 却完全是使用 C 语言编写的, 具备良好的可移植性, 其他非 C 语言编写的代码数量非常少, 相对来说是微不足道的, 根据粗略统计, 非 C 语言编写的代码数量是总体代码数量的 5% 左右。

开发环境的搭建

在 Hello China 的开发过程中，采用 NASM 和 Visual C++ 6.0 等工具完成了不同操作系统模块的开发和编译、连接。其中，NASM 完成汇编语言实现的模块的编译和连接，其他由 C 语言完成的模块，则由 VC 编译和连接。

NASM 的使用

在当前版本的 Hello China 的实现中，汇编语言实现的模块都按照纯二进制格式进行编译，即编译的可执行映像的结果与汇编语言源文件的逻辑完全一致，没有任何编译器附加的头信息。这样的纯二进制模块，可通过下列步骤开发完成：

- 1. 采用任何一个文本编辑器（比如 DOS 操作系统下的 edit）编辑汇编语言源程序，并保存为 .ASM 文件；
- 2. 采用 NSAM 程序编译上述文件，格式为：`nasm -f bin xxx.asm -o xxx.bin`，其中 xxx.asm 是待编译的汇编语言源文件，xxx.bin 是编译的结果文件。

Visual C++的使用

Hello China 的绝大部分核心功能是采用 C 语言编写完成的，采用 Visual C++ 6.0 作为编译连接工具。下列步骤描述了 Hello China 开发过程中开发环境的搭建：

1. 创建一个 DLL 工程：

一般情况下，VC 可以生成 PE 格式的可执行文件、DLL 文件等文件类型，但可执行文件不太适合 OS 映像，因为编译器在编译的时候，会自动在映像文件中加入一些其他代码，比如 C 运行期库的初始化代码等，这样会导致映像文件的体积变大。而 DLL 格式的文件则不会有这个问题，因此，建议从 DLL 开始来建立 OS 映像。

在 Microsoft Visual C++中，按照图 1-5 所示创建一个 DLL 工程：

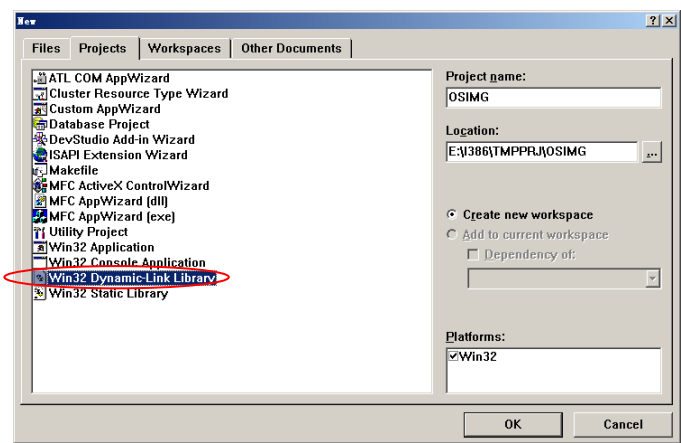


图 1-5 创建一个 DLL 工程

## 2. 设置项目编译与连接选项：

一般情况下，需要对创建的工程设定如下编译连接选项：

1. 对齐方式，在项目选项中，添加/`ALIGN:XXXX` 选项，告诉连接器如何处理目标文件映像在内存在中的对齐方式，一般情况下，需要设置为跟目标文件在磁盘存储时的对齐方式一致，根据经验设置为 16 是可以正常工作的；

2. 设置基址选项，修改默认情况下的加载地址，比如目标文件在我们自己的操作系统中从 0x00100000（1M）处开始加载，则在连接工程选项里面添加/`BASE:0x00100000` 选项；

3. 设置入口地址，如果不设置入口地址，编译器会选择缺省的函数作为入口，比如针对可执行文件是 `WinMain` 或 `main`，针对动态链接库是 `DllMain` 或 `EntryPoint`，等等，采用缺省的入口地址，有时候不能正确的控制映像文件的行为，还可能导致映像文件尺寸变大，因为编译器可能在映像文件中插入一些其他的代码。因此，建议手工设置入口地址，比如，假设我们的操作系统映像的入口地址是 `__init` 函数，则需要设定如下选项：`/entry:?__init@@YAXXZ`，其中，`?__init@@YAXXZ` 是 `__init` 函数被处理后的内部标号，因为 Visual C++ 采用了 C++ 的名字处理模式，而 C++ 支持重载机制，所以编译器可能把原始的函数名变换成内部唯一的标号表示形式，关于如何确定一个函数的内部标号表示，请参考本文的附录。

上述所有的设置，请参考图 1-6：

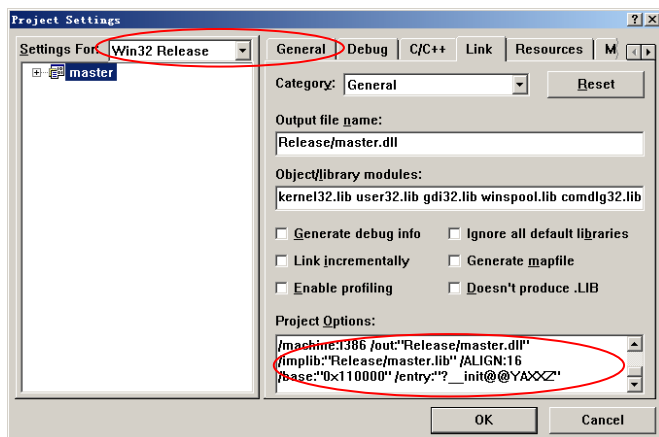


图 1-6 设置编译、链接选项

在 Visual C++ 6.0 中，上述对话框可以从 “project->setting...” 打开，需要注意的是，打开时，是针对 DEBUG 版本设定的，请一定选择 Release 版本进行设定（图中红色圆圈中注明的地方）。

上述步骤完成之后，集成开发环境就设置完毕了，剩下的工作就是直接在该工程中添加 C 语言源文件。

### 编译后模块的进一步处理

上述开发环境搭建完毕之后，就可以在该开发环境下进行操作系统的开发工作了。其中，由 NASM 编译的二进制映像直接可以加载到内存中运行，但由 Visual C++ 编译的二进制模块却是一个 Windows 操作系统下的 DLL 文件，是无法直接加载到内存中运行的，这时候需要对其进行一个预处理。预处理的目的是以及预处理方法的实现机制，请参考本书附录中的相关内容，下面仅仅给出预处理的操作步骤：

1. 把 Visual C++ 编译连接而成的目标文件（Release 版本的 DLL 文件）拷贝到与 process.exe 相同的目录下，其中，process.exe 就是预处理程序；
2. 直接运行 process.exe 程序修改上述 DLL 文件，修改之后，上述 DLL 模块就可以直接加载到内存中运行了。

### 1.4.3 面向对象思想的模拟

虽然在 Hello China 核心模块的开发中使用的是 C 语言，但在开发过程中引入了面向对象的编程与开发思想，把整个核心模块分成一系列对象（比如内存管理器对象、核心线程管理器对象、页框管理器对象、对象管理等）来实现。这样实现起来，独立性更强，而且具备更好的可移植性和可裁减性。

但 C 语言本身是面向过程的语言，因此用 C 语言来进行面向对象的编程，需要对 C 语言做一些简单的预处理。在 Hello China 的开发中，我们先预定义了一系列宏，用来实现面向对象的编程机制，另外，针对 Hello China 的特点，定义了一个对象管理框架，统一归纳所有开发过程中的对象。

在 Hello China 的开发中，我们充分利用 C 语言的宏定义机制，以及函数指针机制，实现了下列简单的面向对象机制。

#### 1. 使用结构体定义实现对象

面向对象开发的一个核心思想就是对象，即把任何可以类型化的东西看作对象，而把程序之间的交互以及调用，以对象之间传递消息（实际上就是对象成员函数的调用）的形式来实现。面向对象的语言（比如 C++）专门引入了对象类型定义机制（比如 class 关键字），C 语言中没有专门针对面向对象的思想，也没有引入对象类型定义机制，但 C 语言中的结构体定义却十分适合定义对象类型（实际上在 C++ 语言中，struct 关键字也用来定义对象类型）。比如在 C++ 语言中，定义一个对象类型如下。

```
class __COMMON_OBJECT
{
private:
    DWORD    dwObjectType;
    DWORD    dwObjectSize;
```

```
Public:
    DWORD    GetObjectType();
    DWORD    GetObjectSize();
};
```

利用 C 语言的 `struct` 关键字，也可以实现类似的对象定义。

```
struct __COMMON_OBJECT
{
    DWORD    dwObjectType;
    DWORD    dwObjectSize;

    DWORD    (*GetObjectType)(__COMMON_OBJECT* lpThis);
    DWORD    (*GetObjectSize)(__COMMON_OBJECT* lpThis);
};
```

与 C++不同的是，C 语言定义的成员函数增加了一个额外参数：`lpThis`，这是最关键的一点。实际上，C++语言在调用成员函数的时候，也隐含了一个指向自身的参数（`this` 指针），因为 C 语言不支持这种隐含机制，因此需要明确的指定指向自身的参数。

这样就可以定义一个对象。

```
__COMMON_OBJECT CommonObject;
```

调用对象的成员函数，在 C++里面代码如下。

```
CommonObject.GetObjectType();
```

而在 C 语言中（参考上述定义），则可以这样：

```
CommonObject.GetObjectType(&CommonObject);
```

使用这种思路，我们简单实现了 C 语言定义对象的基础支撑机制。

## 2. 使用宏定义实现继承

面向对象的另外一个重要思想就是实现继承，而 C 语言不具备这一点。为了实现这个功能，我们在定义一个对象（结构体）的时候，同时也定义一个宏，比如定义如下对象。

```
struct __COMMON_OBJECT
{
    DWORD    dwType;
    DWORD    dwSize;
    DWORD    GetType(__COMMON_OBJECT*);
    DWORD    GetSize(__COMMON_OBJECT*);
};
```

同时，定义如下宏。

```
#define INHERIT_FROM_COMMON_OBJECT \
    DWORD  dwType; \
    DWORD  dwSize; \
    DWORD  GetType(__COMMON_OBJECT*); \
    DWORD  GetSize(__COMMON_OBJECT*);
```

假设另外一个对象从该对象继承，则可以这样定义：

```
struct __CHILD_OBJECT
{
    INHERIT_FROM_COMMON_OBJECT
    .. ...
};
```

这样就实现了对象\_\_CHILD\_OBJECT 从对象\_\_COMMON\_OBJECT 继承的目的。

显然，这样做的一个不利之处是对象尺寸会增大（每个对象的定义都包含了指向成员函数的指针），但相对给开发造成的便利以及增强的代码的可移植性而言，是非常值得的。

### 3. 使用强制类型转换实现动态类型

面向对象语言的一个重要特性就是，子类类型的对象可以适应父类类型的所有情况。为实现这个特点，我们充分利用了 C 语言的强制类型转换机制。比如\_\_CHILD\_OBJECT 对象从\_\_COMMON\_OBJECT 对象继承，那么从理论上说，\_\_CHILD\_OBJECT 可以作为任何参数类型是\_\_COMMON\_OBJECT 的函数的参数。比如下列函数：

```
DWORD GetObjectName(__COMMON_OBJECT* lpThis);
```

那么，以\_\_CHILD\_OBJECT 对象作为参数是可以的：

```
__CHILD_OBJECT Child;
GetObjectName((__COMMON_OBJECT*)&Child);
```

可以看出，上述代码使用了强制的类型转换。

在 Hello China 的开发中，我们使用强制类型转换实现了对象的多态机制。

#### 1.4.4 对象机制

在面向对象的语言（比如 C++）中，实现了一系列对象机制，比如对象的构造函数和析构函数等，另外，一些面向对象的编程框架（比如 OWL 和 MFC 等）对应用程序创建的每个对象都做了记录和跟踪，这样可以实现对象的合理化管理。

但在 C 语言中，缺省情况下却没有这种机制。为了实现这种面向对象的机制，在 Hello China 的开发过程中，根据实际需要建立了一个对象框架，来统一管理系统中创建的对象，

并提供一种机制，对对象创建时的初始化以及销毁时的资源释放做出支持。

在 Hello China 开发过程中实现的对象机制，主要思路如下：

1. 每个复杂的对象（简单的对象，比如临时使用的简单类型等不包含在内），在声明的时候，都声明两个函数：Initialize 和 Uninitialize，其中第一个函数对对象进行初始化，第二个函数对对象的资源进行释放，然后定义一个全局数组，数组内包含了所有对象的初始化函数和反初始化函数；

2. 定义一个全局对象，对系统中所有对象进行管理，这个对象的名字是 ObjectManager（对象管理器），该对象提供 CreateObject、DestroyObject 等接口，代码通过调用 CreateObject 函数创建对象，当对象需要销毁时，调用 DestroyObject 函数。

第一点很容易实现，只要在声明的时候，额外声明两个函数即可（这两个函数的参数是 `__COMMON_OBJECT*`），声明完成之后，把这两个函数添加到全局数组中（该数组包含了系统定义的所有对象相关信息，比如对象的大小、对象的类型、对象的 Initialize 和 Uninitialize 函数等）。

对象管理器 ObjectManager 则维护了一个全局列表，每创建一个对象，ObjectManager 都把新创建的对象插入列表中（实际上是一个以对象类型作为 Key 的 Hash 表）。每创建一个对象 ObjectManager 都申请一块内存（调用 KMemAlloc 函数），并根据对象类型，找到该对象对应的 Initialize 函数（通过搜索对象信息数组），然后调用这个函数初始化对象。

对于对象的销毁，ObjectManager 则调用对象的 Uninitialize 函数，这样就实现了对象的自动初始化和对象资源的自动释放。

在 Hello China 的开发过程中，一直使用这种对象模型，实际上，对象模型不局限于对象的自动初始化和自动销毁，而且还适用于对象枚举、对象统计等具体功能，比如，为了列举出系统中所有的核心线程，可以调用 ObjectManager 的特定函数，该函数就会列举出系统中的所有核心线程对象，因为 ObjectManager 维护自己创建的所有对象的列表，而核心线程对象就是使用 ObjectManager 创建的。

## 1.4.5 Hello China V1.0 版本的源文件构成

图 1-7 示意了 Hello China V1.0 的源文件目录结构。

其中，Debug 和 Release 目录是编译环境（Visual C++）自动生成的，用于存放编译过程文件和最终的编译结果，Hello China 的源文件，存放在下列几个目录中。

1. Drivers 目录，硬件环境的设备驱动程序，比如网络接口卡驱动程序、IDE 接口硬盘驱动程序等；

2. Kernel 目录，Hello China 操作系统核心功能的所有源文件；

3. MISC 目录，一些辅助功能的源文件，比如字符串操作源文件等；

4. NetCore 目录，网络协议栈源代码。

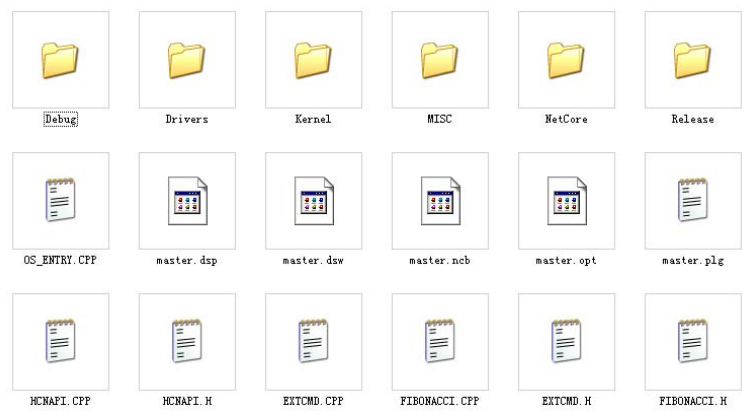


图 1-7 Hello China 的源文件组成结构

另外，在根目录下存在 OS\_ENTRY.CPP 等几个源文件，这些文件可被应用程序直接修改，以迎合开发的需要。比如，为了在操作系统启动的时候，就启动应用程序线程（由用户开发），就需要修改 OS\_ENTRY.CPP 源文件：在操作系统入口点 \_\_init 函数里，添加启动应用程序的代码。

在 Hello China V1.0 里，对设备驱动程序和网络协议栈的实现都很有限，而对操作系统核心功能却做了全面的实现，因此，Kernel 目录是整个 Hello China 的核心目录，该目录下面包含了操作系统核心功能的所有相关源文件，图 1-8 示意了该目录下的文件。



图 1-8 Kernel 目录下的源文件

在本书的后续部分，将对涉及到的每个操作系统核心功能及其源代码进行详细描述。

### 1.4.6 Hello China V1.5 版本的源文件构成

在 Hello China V1.5 的实现中，为了使代码更加容易管理和组织，对源代码的存放目录做了进一步的细分，如下图：

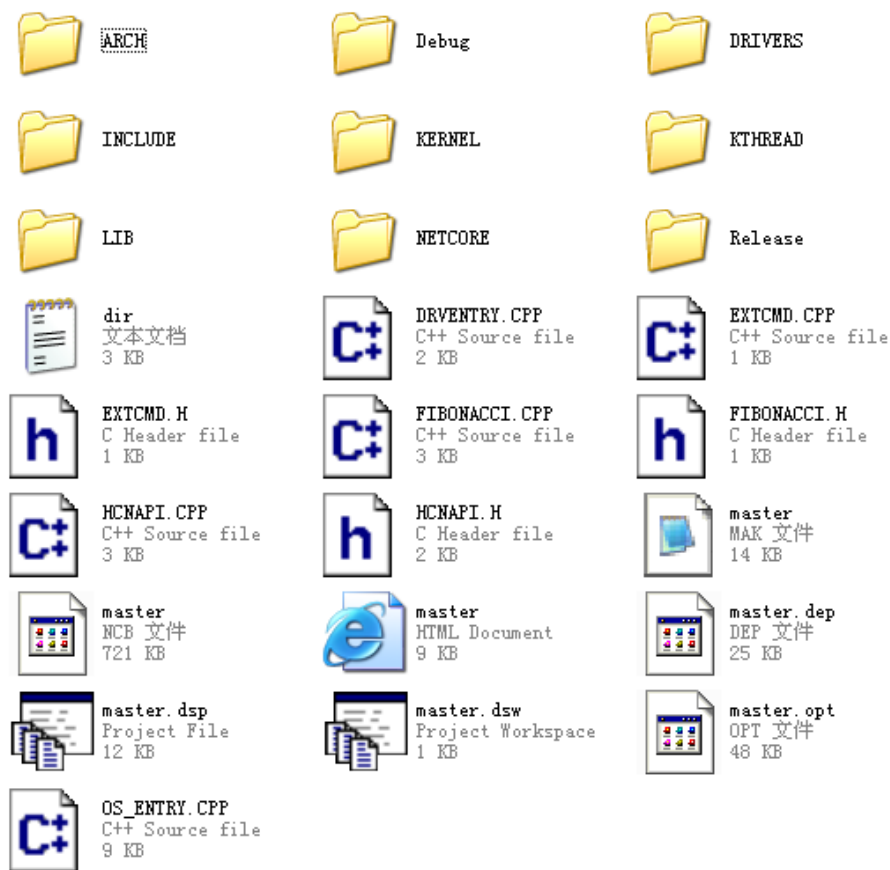


图 1-9 V1.5 版本的源文件组织

在 V1.0 的基础上，增加了下列几个目录：

1、ARCH 目录：存放硬件平台相关的源代码文件。在 V1.5 的实现当中，把与硬件（CPU、IO 等）关系密切的代码，单独拿了出来，组织到单独的文件中，并存放到该目录下。这样可增加 Hello China 的可移植性；

2、INCLUDE 目录：所有源文件的头文件，都组织到了该目录下，这样源代码在包

涵特定的头文件的时候，只需要包涵该目录下的相关文件即可；

3、KTHREAD 目录：V1.5 版本实现的一些应用程序源代码，都存放在了该目录下。比如 sysdiag 程序、IOCTRL 程序、hyptrm/hyptrm2 程序等。这样可使得应用实现代码跟核心代码隔离开；

4、LIB 目录：C 语言库函数的实现目录。在 V1.5 的实现中，增加了一些符合标准 C 运行库的功能，比如字符串操作功能、可变参数个数的 IO 函数等，这些函数的实现代码，都是存放在该目录下的。

另外，在 V1.5 的实现中，把原 1.0 中存在的 NetCore 目录删除掉了，因为 V1.5 和 V1.0 中，都没有真正实现网络协议栈，这项功能会在后续版本中实现。

### 1.4.7 Hello China 的使用

当前版本的 Hello China (Version 1.0), 在 PC 上的操作系统核心模块由下列几个组成:

1. BOOTSECT.BIN, 软盘引导扇区, 用于完成操作系统的引导;
2. REALINIT.BIN, 实模式下的初始化模块, 在转入保护模式 (Intel IA32 CPU) 前, 该模块完成实模式下 PC 硬件设备的初始化;

3. MINIKER.BIN, 保护模式下的初始化代码, 以及键盘、显示驱动程序代码, 该模块也是专门针对 PC 的;

4. MASTER.BIN, 操作系统功能核心模块, 所有操作系统核心功能都是在该模块中实现。

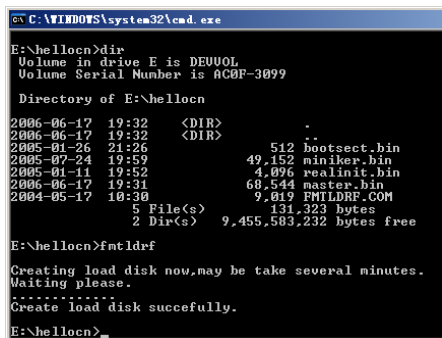
其中, 前三个模块是专门为 PC 平台设计的, 用于完成 PC 系统的初始化、PC 的基本输入/输出 (键盘输入、屏幕字符输出) 等, 在把 Hello China 移植到其他硬件平台上时, 这些模块可能不需要, 但 MASTER.BIN 模块却是操作系统的核心, 该模块完全由 C 语言编写而成, 可以通过重新编译 MASTER.BIN 模块的源代码来移植到其他硬件平台。

另外, 为了在 PC 硬件平台上创建一张引导软盘, 还专门填写了 FMTLDRF.COM 应用程序, 该应用程序把上述四个模块按照一定的格式写入到软盘上, 这样就创建了一张引导软盘, 从而可以正常地启动计算机。

下面给出当前版本的 Hello China 在 PC 上的使用方法:

1. 把上述四个二进制模块以及 FMTLDRF.COM 应用程序, 拷贝到同一个目录下;
2. 把一张新格式化的软盘插入软盘驱动器, 需要注意的是, 一定要确保写保护是关闭的, 即软盘能够被写入;
3. 在上述目录下运行 FMTLDRF.COM 程序, 则会创建一张引导软盘;
4. 使用上述引导软盘重新启动计算机, 则进入 Hello China 操作系统环境, 可以通过 Hello China 提供的一个字符 shell, 完成对话功能。

如图 1-9 所示。



```

C:\WINDOWS\system32\cmd.exe
E:\hellocn>dir
Volume in drive E is DEU00L
Volume Serial Number is AC0F-3099

Directory of E:\hellocn

2006-06-17  19:32    <DIR>        .
2006-06-17  19:32    <DIR>        ..
2005-01-26  21:26                512 bootsect.bin
2005-07-24  19:59            49,152 miniker.bin
2005-01-11  19:52                4,096 realinit.bin
2006-06-17  19:31            68,544 master.bin
2004-05-17  10:30             9,019 FMTLDRF.COM
               5 File(s)      131,323 bytes
               2 Dir(s)      9,455,583,232 bytes free

E:\hellocn>fmtldr.f
Creating load disk now,may be take several minutes.
Waiting please.
.....
Create load disk sucefully.

E:\hellocn>
  
```

图 1-10 Hello China 引导盘的创建

## 1.5 实例：一个简单的 IP 路由器的实现

### 1.5.1 概述

下面我们以一个 IP 路由器的实现为例，来说明如何在嵌入式操作系统环境下开发一个具体的应用。IP 路由器是 IP 网络的核心部件，完成不同 IP 网段之间 IP 报文的转发工作，比如，一个 IP 地址 202.16.0.0，掩码是 24 位的网段，跟另外一个 IP 地址 210.92.0.0，掩码是 24 位的网段之间进行互通，就需要用到路由器，因为这两个网段不属于同一个网段（IP 地址跟掩码进行与运算，得到网络地址，若网络地址相同，则是同一个网段，否则不属于同一个网段，不同网段之间的通信，需要经过路由器进行转接）。其中，路由器中维护一个重要的数据结构作为转发的依据，这个数据结构就是路由表。一旦一个 IP 报文从路由器的一个接口到达，路由器就会接收这个 IP 报文，并从 IP 报文头中得到其目的 IP 地址，然后根据目的 IP 地址查找路由表，查找的结果一般是另外一个接口，这样路由器就会把这个 IP 报文从查找到的接口发送出去，从而完成 IP 报文的路由功能。

路由表是由路由协议计算出来并实时更新的（也可以通过手工配置方式形成），这样在路由器上，就必须有路由协议进程在运行来完成路由表的维护工作。IP 路由器的功能非常复杂，上面介绍的只是路由器主要功能的一个概述，如果读者对路由器感兴趣，可参考数据通信相关的书籍。掌握路由器的工作原理以及路由协议，是深入理解数据通信网络的基础。

### 1.5.2 路由器的硬件结构

在这个实例中，我们重点介绍路由器的软件实现。在介绍软件之前，首先假定一个硬件环境，这样在介绍软件实现的时候才会有基础依据。在我们实现的这个路由器中，硬件结构非常简单，也非常典型，如图 1-10 所示。

一片 CPU 构成了整个系统的核心，作为中央处理部件，该 CPU 完成所有的处理任务。CPU 和内存之间，通过特定 CPU 的总线进行连接，然后通过一片 Host-PCI 桥接芯片把 CPU/内存和 PCI 总线连接在一起，通过这个 Host-PCI 桥接芯片（这片芯片，也叫做北桥芯片），CPU 就可以很容易的跟 PCI 总线上的部件进行通信。有三个接口控制芯片分别对应路由器的三个接口（一台路由器至少有一个物理接口），这三个接口控制芯片直接连接到 PCI 总线上。另外一个桥接芯片 PCI-ISA 桥连接了 PCI 总线和传统的 ISA 总线，这样基于 RS-232 的串行通信接口芯片（COM 口）以及基于 ISA 总线的 Ethernet 接口芯片就可以连接到系统中，从而被 CPU 控制。PCI-ISA 桥接芯片又叫做南桥芯片，完成 PCI 总线和 ISA 总线的协议转换功能。这是一个很简单的硬件体系，但也是一个非常典型的

硬件结构，很多基于软件实现的 IP 路由器，都是由这种结构构成的，所不同的是，可能在 PCI 总线和 ISA 总线上，挂接了更多的设备（控制芯片）而已。对个人计算机（PC）硬件构架熟悉的读者，会发现该硬件构架跟 PC 的硬件是十分类似的，从这个意义上说，PC 本身就是一个典型的嵌入式系统。

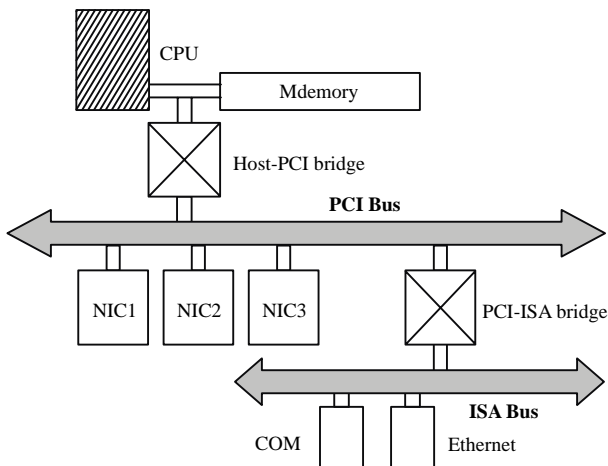


图 1-11 一个简单的路由器硬件构架

### 1.5.3 路由器的软件功能

在这种硬件构架下对 IP 报文的转发，过程如下。

1. 路由器的一个接口卡接收到一个 IP 报文（封装在特定的链路层帧之内），接口卡缓存该报文，并向 CPU 发起一个中断申请；

2. CPU 响应中断请求，在中断处理程序中，把接口卡接收到的报文（报文缓存在接口卡的本地缓冲区内），拷贝到内存中；

3. 这时候，有两种处理方式：一种是直接在中断服务程序中提取 IP 报文头信息（比如目的 IP 地址），然后查找路由表，从另外一个接口转发报文（或丢弃），然后再从中断处理程序中返回。另一种方式是，系统中有一个 IP 转发任务在运行，中断处理程序仅仅是把接收的 IP 报文挂到转发任务的发送队列中，然后直接从中断处理程序中返回。这两种方式各有优缺点，我们的实例采用后者实现；

4. IP 转发任务检查发送队列，发现有一个报文等待转发，于是根据 IP 报文的目的地址，查找路由表，查找到一个出接口后，调用出接口卡的驱动程序提供的发送功能函数，完成 IP 报文的发送；

5. 上述操作全部完成之后，IP 报文转发过程结束。

除了 IP 转发任务（实际上是一个线程）之外，还需要对路由器进行维护，对路由器

的维护，有两种典型的方式。

- 1. 通过 telnet，从远程的一台计算机上登陆到路由器，然后通过命令行界面，对路由器进行维护；
- 2. 直接在路由器本地，通过一条符合 RS-232 标准的串口线缆连接路由器的 COM 接口，对路由器进行维护。

当然，若路由器被应用在电信、大型企业等环境中，则还需要支持 SNMP（简单网路管理协议）等管理协议，来通过网管中心进行维护。在我们的实例中，仅仅考虑通过 Telnet 和串口进行维护的情况。

根据上面的描述，路由器的软件功能至少应该包含下列部分。

- 1. IP 转发功能，完成 IP 报文的转发；
- 2. Telnet 服务器功能，用于接收远程发起的连接请求，完成远程维护工作；
- 3. COM 接口响应服务器功能，用于完成通过 COM 接口进行维护的用户对话；
- 4. 路由协议功能，用于完成路由表的维护，一般情况下，根据作用的范围不同，路由协议又可进一步分为内部网关协议（IGP）和外部网关协议（EGP），其中 OSPF 是一个比较典型的内部网关协议，BGP（Version 4）是最典型的外部网关协议，在我们的实例中，这两种协议都做了考虑；
- 5. TCP/UDP 协议栈，这个协议栈用于路由器本身。

对于上述每个功能，我们创建一个任务（线程）与之对应，这样整个路由器的软件部分，就由图 1-11 中的 6 个任务组成。

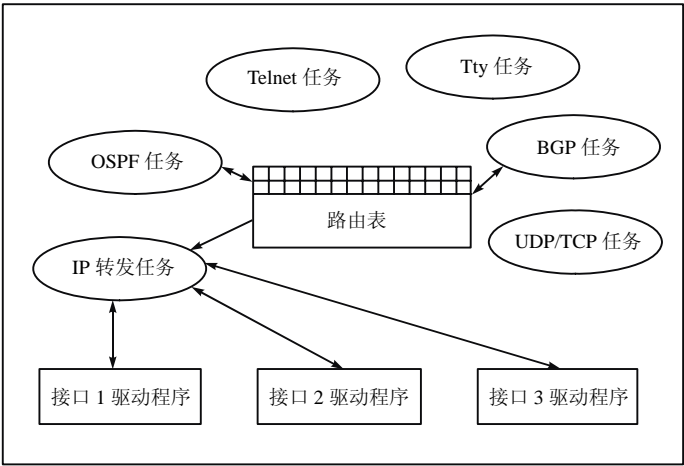


图 1-12 一个简单的路由器软件构架

其中，路由表是一个核心的数据结构，被多个任务所应用，比如 IP 转发任务需要通过路由表来决定转发方向，而路由协议任务（OSPF/BGP）则需要根据网络变化情况，实

时更新路由表，等等。这样路由表就是一个共享的数据结构，在实现的时候，为了确保数据的一致性，需要采用任务同步机制进行保护。

### 1.5.4 各任务的实现

路由器的软件功能被分解、划分成几个相互独立的任务之后，就很容易在支持多任务的嵌入式操作系统上实现了。假设我们的路由器是在 **Hello China** 上进行开发的，而 **Hello China** 支持完善的多任务（多线程）机制和任务同步机制，因此，在实现的时候，我们定义 6 个线程对象用来代表实现路由器功能的 6 个功能模块。

```
__KERNEL_THREAD_OBJECT*    lpIpForwarding;    //Ip forwarding thread.
__KERNEL_THREAD_OBJECT*    lpOspf;            //OSPF thread.
__KERNEL_THREAD_OBJECT*    lpBgp;             //BGP thread.
__KERNEL_THREAD_OBJECT*    lpTelnet;          //Telnet server thread.
__KERNEL_THREAD_OBJECT*    lpConsole;         //COM interface thread.
__KERNEL_THREAD_OBJECT*    lpTcpUdp;          //TCP/UDP thread.
```

**Hello China** 在完成自身初始化后，就应该启动上述六个任务了。在目前的实现中，**Hello China** 的所有初始化功能都是在 `__init` 函数中完成的，因此，一个很好的选择就是修改 `__init` 函数，在该函数的尾部（这时候所有操作系统功能都已经初始化）创建并启动上述线程。相关代码如下。

```
VOID __init()
{
    ... ..
    lpIpForwarding = KernelThreadManager.CreateKernelThread((__COMMON_
OBJECT*)&KernelThreadManager,

                                IpForwarding,
                                NULL,
                                0L,
                                0L,
                                NULL);

    if(NULL == lpIpForwarding)    //Can not create kernel thread.
        goto __TERMINAL;

    //
    //Create other 5 threads here.
    //
    ... ..
}
```

这样，当 **Hello China** 完成自身的初始化后，上述路由器功能的六个线程就会被启动，

目标系统就具备路由器的功能了。

对于每个线程的实现相对来说就比较容易了，因为嵌入式操作系统提供了大量的基础设施，每个具体的功能模块可以充分利用这些基础设施来实现自身功能，比如内存分配、线程同步、消息传递、定时器等功能。我们以 IP 转发线程为例，说明如何利用操作系统提供的同步机制来实现 IP 转发功能。在我们的实例中，IP 转发功能是作为一个单独的线程来实现的，该线程维护一个本地转发队列，队列中存储了等待转发的数据报文。队列中的数据报文，是由接口驱动程序添加的，一旦接口驱动程序接收到一个 IP 报文，则会把该 IP 报文添加到队列中。一旦队列中存在 IP 报文，IP 转发线程就开始工作，依次检查 IP 队列中的每个报文，根据报文的目的 IP 地址，查找路由表，然后从查找到的出接口上发送出去。若队列中没有 IP 报文，则 IP 转发线程进入阻塞状态，以节约系统资源。因此，该线程需要有一个事件对象来配合实现同步功能。该线程的功能描述如下。

```

__EVENT*      lpHasPacket = NULL;

VOID IpForwarding()
{
    ... ..
    lpHasPacket = ObjectManager.CreateObject(&ObjectManager,
                                              NULL,
                                              OBJECT_TYPE_EVENT);    //Create event
object to synchronize itself.
    if(NULL == lpHasPacket)      //Can not create object.
        goto __TERMINAL;

    while(TRUE)
    {
        lpHasPacket->WaitForThisObject((__COMMON_OBJECT*)
lpHasPacket); //Wait for packet to reach.
        while(queue is not empty)
        {
            get a ip packet from the queue;
            look up routing table;
            forward the packet according to routing table;
        }
        lpHasPacket->ResetEvent((__COMMON_OBJECT*)lpHasPacket);
    }
    ... ..
}

```

上述代码中，IpForwarding 函数首先创建一个事件对象，作为 IP 报文队列的指示器，

然后进入一个无限循环。在循环的开始，等待创建的事件对象，若事件对象处于非信号状态，则会导致 IP 转发线程进入阻塞状态。一旦接口驱动程序接收到一个 IP 报文，则驱动程序会把 IP 报文挂到 IP 转发线程的发送队列，然后设置（SetEvent）事件对象。设置事件对象的结果是唤醒 IP 转发线程，IP 转发线程一旦被唤醒，则依次检查转发队列中的 IP 报文，并查找路由表，完成报文的转发，直到 IP 队列变为空（所有 IP 报文处理完毕），然后转发线程复位事件对象，这样在循环的开始处转发线程又会阻塞自己，等待 IP 报文的再次到达。

其他线程的实现与此类似。需要说明的是，对于共享数据结构，比如路由表的访问，需要有一个互斥体对象来进行访问同步。比如在路由器的实现中，定义一个互斥体对象，来完成对路由表的互斥访问，代码如下。

```
__MUTEX*      lpRtMutex = NULL;

... ..
lpRtMutex->WaitForThisObject((__COMMON_OBJECT*)lpRtMutex);
modify routing table;
lpRtMutex->ReleaseMutex((__COMMON_OBJECT*)lpRtMutex);
... ..
```

这样，共享路由表数据结构的各线程之间共享该互斥体对象，在访问路由表的时候，首先获得互斥体对象，然后再进行修改，修改完毕之后，释放互斥体对象。这样可确保路由表的一致性。

上述这个路由器实例，非常简单，仅仅是为了说明如何利用嵌入式操作系统开发一个应用，实际的路由器其功能远不止这些，而且相互之间的关系更加复杂，但开发的基本方法和思路却与此类似。

## 第二章 **Hello China** 的加载和初始化

——The loading and initialization of Hello China

## 2.1 常见嵌入式系统的启动

### 2.1.1 典型嵌入式系统内存映射布局

一个典型的嵌入式系统，至少具备下列存储部件：

- 1、**Boot ROM**：是一片可擦写的只读存储器，一般不会太大（比如，不会超过 1M），用于存放嵌入式系统加电后的初始化代码，在 PC 机上，用于完成加电后检测（POST 功能）的 BIOS，与此类似；
- 2、**Flash**：一块可擦写的存储介质，可用于存储嵌入式系统的操作系统和应用程序映像，以及嵌入式系统的配置数据等，这类介质的尺寸，一般比 **Boot ROM** 要大，比如，可以在 1M 到 64M 之间变化；
- 3、**RAM**：即常规内存，一般情况下，嵌入式系统启动后，执行的代码和数据存放在这个位置。

这三类存储介质，一般直接通过硬件连接的方式，硬性焊接在 CPU 的可寻址空间内，如下图：

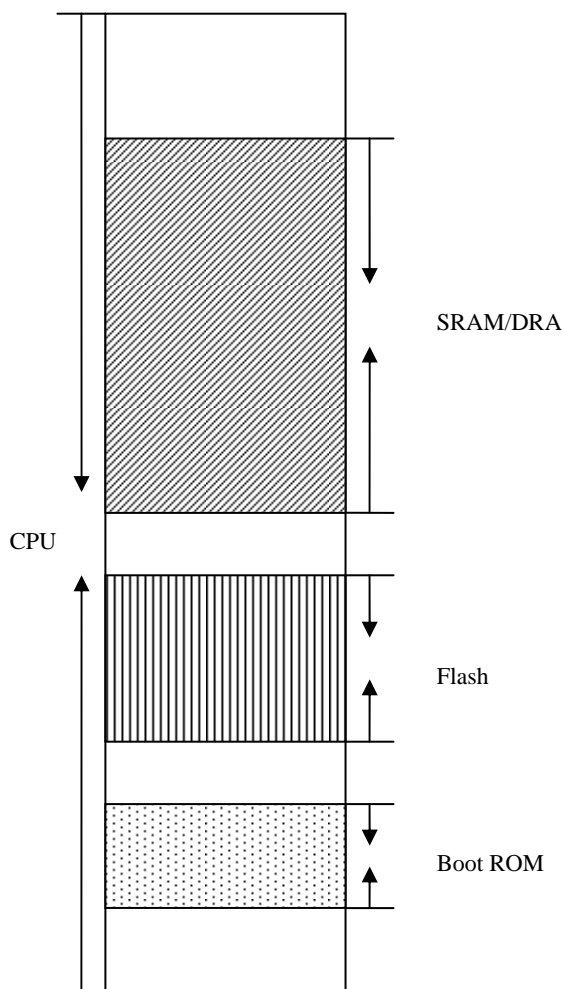


图 2-1 典型嵌入式系统内存布局

这样对于这些存储设备的读写，只需要采用 CPU 的内存读写机制，就可以很方便的完成对这些设备的操作，无需特殊设备驱动程序的支持。

在有的嵌入式系统中，还存在另外一些类型的存储介质，比如 NVROM（非易失性只读存储器）等，这些存储介质往往作为存储设备配置数据的介质而存在，有的情况下，也映射到 CPU 的地址空间中，其操作与 Flash、Boot ROM 等类似。

## 2.1.2 嵌入式系统的启动概述

在嵌入式系统加电后，会触发 CPU 的复位信号（reset），导致 CPU 复位。CPU 复位操作完成之后，一般情况下会直接跳转到内存空间的固定位置，取得第一条指令，开始执行。不同的 CPU，第一条开始执行的指令的位置，是不一样的，比如，在 ARM 系列的 CPU 中，第一条开始执行的指令是地址空间的开始处（即 0x00000000 位置，在 32 位 CPU 情况下），而在 Intel 系列的 IA32 构架 CPU 中，CPU 开始执行的第一条指令，则是位于 0xFFFFFFF0 位置。第一条指令所在的位置（一般称为启动向量），一般情况下是 Boot ROM 所在的位置，在 Boot ROM 中，存放了 CPU 开始执行的第一条指令，一般情况下，这是一条跳转指令，跳转到另外一个固定的位置继续执行。一个很常用的做法是，在 Boot ROM 中，存放了嵌入式系统的初始化代码，启动向量所在的跳转指令，目标跳转地址则是这些初始化代码的开始处。这样嵌入式系统一旦加电，第一部分正式执行的代码，就是硬件系统的初始化代码。

对于硬件系统的初始化，有的情况下会十分复杂，需要初始化的硬件芯片（或硬件设备）非常多，这样必然导致初始化代码十分庞大，这样把这些庞大的初始化代码，放在 Boot ROM 中是不合适的，因此在这种情况下，Boot ROM 里面一般只存放了关键部件的初始化代码，比如 CPU 的初始化（工作模式的选择等）、MMU 的初始化（页表、段表的建立）、中断控制器的初始化、简单输入/输出接口（比如，COM 接口）的初始化等。另外设备的初始化代码，跟嵌入式操作系统放在一起，作为操作系统的一部分代码来实现。

Boot ROM 中的硬件初始化代码执行完毕，对基本的硬件环境完成初始化之后，下一步工作就是加载操作系统和应用软件了（在嵌入式领域，操作系统和应用软件往往编译在一个二进制模块中），这个过程会根据不同的配置，以及不同规模的应用，有不同的实现方式，在接下来的部分当中，我们对常见的加载方式，进行描述。

一般情况下，操作系统和应用代码的映像，存储在 Flash 当中，因此在加载的时候，必然涉及到对 Flash 的操作。在嵌入式系统中，Flash 一般是直接映射到 CPU 的地址空间中，这样对 Flash 的操作，直接通过 CPU 的访问内存指令，就可以完成，无需额外提供 Flash 设备的驱动程序。但这种方式有一个缺点，就是占用了 CPU 的地址空间。若不采取这样的方式，而是把 Flash 当作存储外设（比如，硬盘），则必须提供特定的驱动程序，来支撑这种形态的 Flash 的访问。由于这时候操作系统还没有加载，因此这时候，Flash

的驱动程序只能存放在 Boot ROM 中。

### 2.1.3 常见嵌入式操作系统的加载方式

在本节中，我们对嵌入式操作系统的加载方式，进行一个简单的描述。之所以对嵌入式操作系统的加载方式进行描述，是为了让读者更好的了解常见的嵌入式系统的启动过程，以便可以根据实际需要，把 Hello China 移植到特定的目标系统上。

- **加载方式之一：从 Flash 直接运行代码**

这种加载方式下，嵌入式操作系统映像和应用程序映像，存放在 Flash 当中。在编译的时候，操作系统和应用程序映像的二进制模块，被编译器分成了不同的节，比如 TEXT 节、DATA 节、BSS 节等。其中，不同的节所存放的内容不同，比如，TEXT 节存放了可执行代码，DATA 节存放了已经初始化的全局变量，而 BSS 节是一个预留节，存放了未经初始化的全局变量等。

在这种加载方式下，嵌入式系统的启动过程如下（参考下图）：

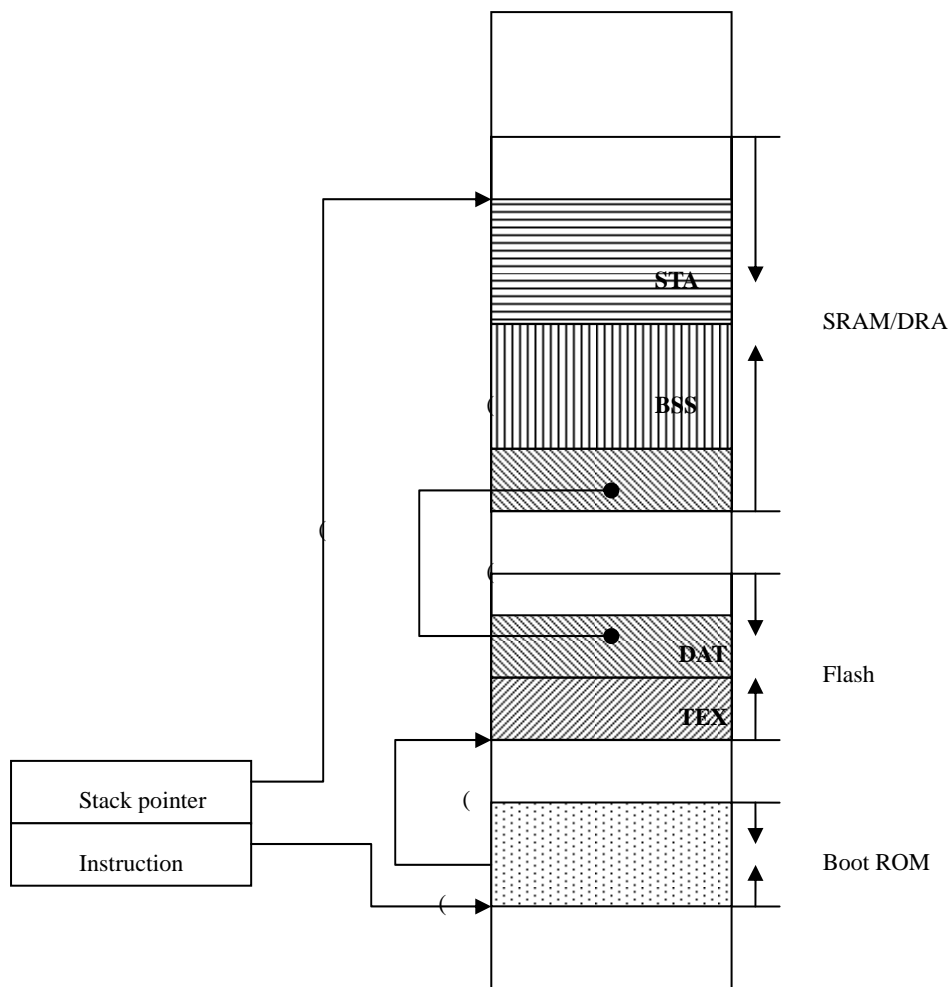


图 2-2 从 Flash 直接运行代码的加载过程

- 1、CPU 复位完成，执行启动向量所在的第一条指令（位于 Boot ROM 内），这条指令往往是一条跳转指令，跳转到 Boot ROM 内的硬件初始化代码位置，执行必须的硬件初始化工作；
- 2、硬件初始化代码完成 CPU 的初始化，比如设置 CPU 的段寄存器、堆栈指针等，以及其它硬件的初始化；
- 3、完成硬件的初始化功能后，会通过一条跳转指令（或函数调用指令），跳转到 Flash 存储器的特定位置开始执行。这个位置，一定是代码段（TEXT 段）中的一个特定位置；

- 4、Flash 中的代码，把 DATA 节从 Flash 中复制到 RAM 中；
- 5、完成 DATA 节的复制后，Flash 中的代码会根据 BSS 节的大小，在 RAM 中预留相应的内存空间，作为未初始化变量使用；
- 6、上述功能完成之后，嵌入式系统执行的环境已经准备完毕，进入操作系统初始化阶段。

需要注意的是，对于 DATA 节的搬迁和 BSS 节的预留工作，也可能是由 Boot ROM 完成的，即 Boot ROM 中的硬件初始化代码执行完毕之后，会通过一些内存搬移指令，把 Flash 中的 DATA 节搬移到 RAM 中，然后再根据 BSS 节的大小，预留 BSS 空间，这些工作完成之后，就可通过一条跳转指令，跳转到 TEXT 节的某个位置（一般是操作系统的入口函数）开始执行。这种情况下，需要 Boot ROM 中的代码，知道 DATA 节和 BSS 节的详细信息（大小、起始地址等）。

在这种加载方式下，所有的执行指令，都是从 Flash 中读取的，RAM 中只是存放了数据和堆栈。这种启动方式，可以节约物理内存空间，因为不需要专门为代码预留内存空间，因此一般应用在内存数量受到限制的系统中。这种加载方式，有以下缺点：

- 1、由于代码直接在 Flash 中执行，一般情况下对 Flash 的访问，速度会比对 RAM 的访问要慢很多，因此性能会受到影响。这个问题，若 CPU 本身携带了较大数量的代码 cache，则会得到一定程度的缓和；
- 2、代码直接从 Flash 中运行，而一般情况下，Flash 是只读的，因此无法实现代码的自我修改功能（即动态修改代码），这样不利于代码级的调试；
- 3、在对操作系统核心和应用程序代码进行编译的时候，必须指定 TEXT 节和 DATA 节、BSS 节的起始地址，这个地址应该跟最终这些节在 Flash 和 RAM 中的位置相同。因此，会给开发带来一定的困难。

总之，这种加载方式，在一些性能要求不是太关键、硬件配置受到限制的系统中，被大量的采用。Internet 发展初期的一些低端路由器，经常采用这种方式。

## ● 加载方式之二：从 RAM 运行代码

与上述方式“从 Flash 直接运行代码”不同的是，这种方式下，代码和数据都被加载到 RAM 中，从 RAM 中运行。这样从 Flash 直接运行的一些弊端，就可以消除了。这种方式下，加载过程如下（参考下图）：

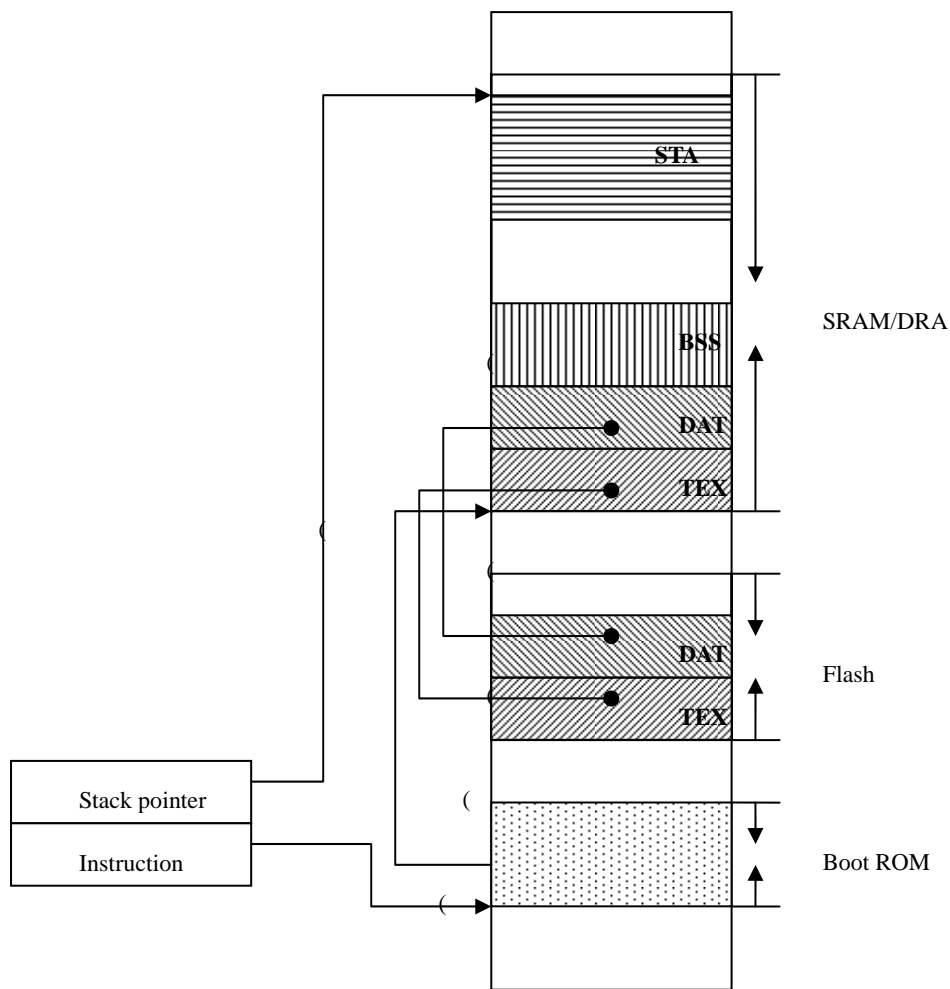


图 2-3 从 RAM 运行代码的加载方式

- 1、CPU 复位完成，执行启动向量所在的第一条指令（位于 Boot ROM 内），这条指令往往是一条跳转指令，跳转到 Boot ROM 内的硬件初始化代码位置，执行必须的硬件初始化工作；
- 2、硬件初始化代码完成 CPU 的初始化，比如设置 CPU 的段寄存器、堆栈指针等，以及其它硬件的初始化；
- 3、位于 Boot ROM 中的代码，把位于 Flash 中的操作系统和应用程序的 TEXT 节（代码节），从 Flash 中搬移到 RAM 中。这个过程，需要 Boot ROM 内的搬移代码，预先知

道 TEXT 节的起始地址和大小，这可以通过在操作系统映象的开始处（该位置往往是固定的），设置一个数据结构，来指明这些节的大小和起始位置，以及目标位置等。这样在搬移的时候，Boot ROM 中的代码，就可以先从这个固定位置，获得节的信息，然后再根据这些信息，来完成搬移工作；

- 4、与第三步类似，Boot ROM 中的代码，把操作系统和应用程序映象的 DATA 节，搬移到 RAM 中；
- 5、Boot ROM 中的启动代码，完成 BSS 节的 RAM 空间预留；
- 6、完成上述所有动作之后，Boot ROM 中的代码通过一条跳转指令，跳转到 RAM 中操作系统的入口点，正式启动操作系统。

这种启动方式，是一种比较常见的启动方式，简便易行，而且克服了“直接从 Flash 中运行代码”的一些弊端。但这种方式需要嵌入式系统配置较多的 RAM 存储器，因为操作系统和应用程序的代码，将直接在 RAM 中执行。

### ● 启动方式之三：从文件系统加载运行

在上面介绍的两种启动方式中，操作系统和应用程序的二进制映象，是存放在 Flash 当中的，而 Flash 直接映射到 CPU 的可寻址空间，因此在加载的时候，直接通过访存指令，就可以完成，无需额外的设备驱动程序。但这种方式，需要系统配置较多的 Flash 存储器，这在操作系统映象很大的情况下，矛盾尤其突出。而下面介绍的一种方式，是从外部存储器加载操作系统和应用程序映象的，可以解决该问题。

这种从文件系统加载操作系统映象的方式，与从 Flash 加载不同的是，操作系统映象和应用程序映象是存放在外部存储器（比如，IDE 接口的硬盘）中的。而对于外部存储器的访问所需要的驱动程序，则存放在 Boot ROM 中。这种方式的加载过程，如下（可参考下图）：

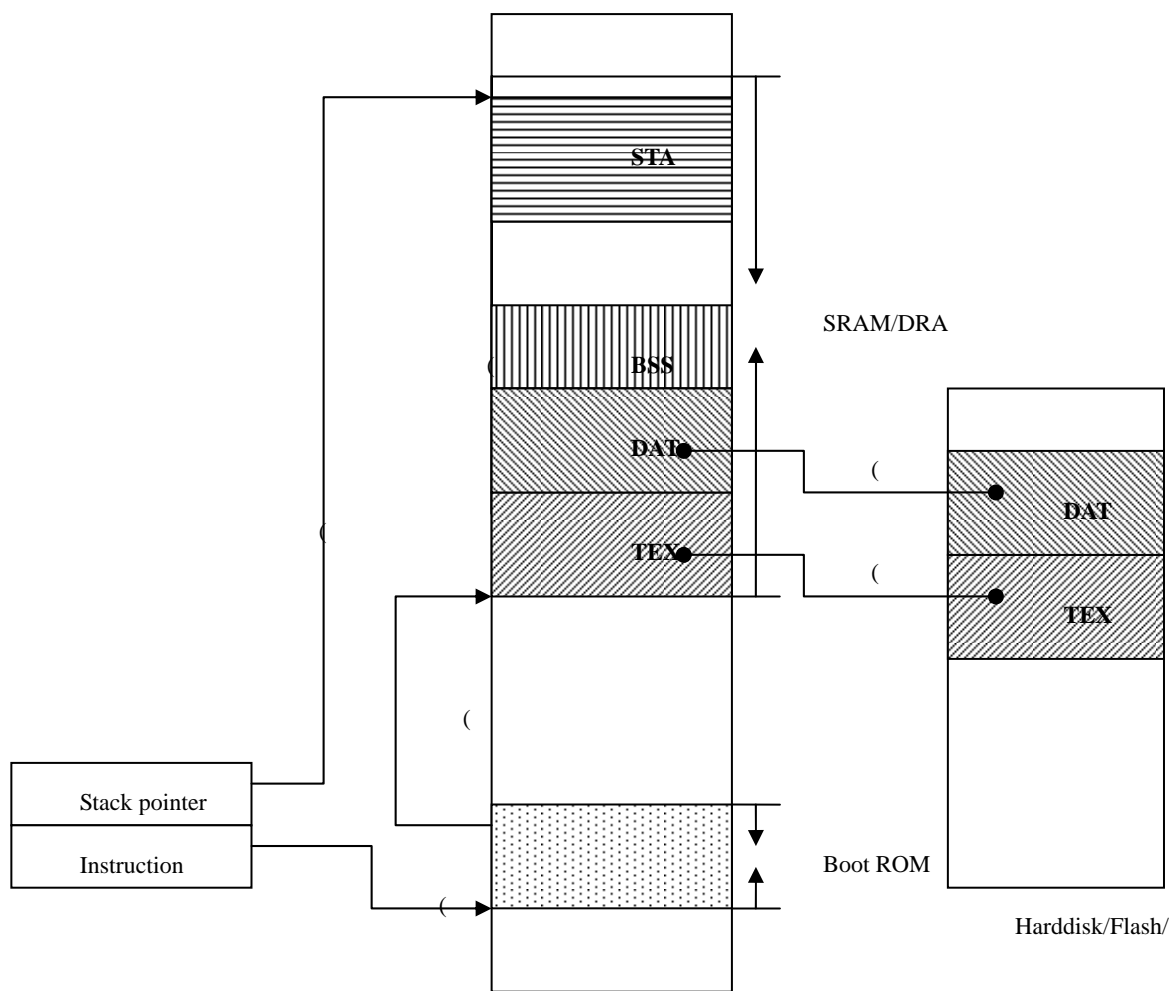


图 2-4 从文件系统加载系统的过程

- 1、CPU 复位完成，执行启动向量所在的第一条指令（位于 Boot ROM 内），这条指令往往是一条跳转指令，跳转到 Boot ROM 内的硬件初始化代码位置，执行必须的硬件初始化工作；
- 2、硬件初始化代码完成 CPU 的初始化，比如设置 CPU 的段寄存器、堆栈指针等，以及其它硬件的初始化；
- 3、Boot ROM 中的引导代码，通过外部设备的驱动程序，读取外部存储器，把操作系统和应用程序的映像，加载到内存 (RAM) 中。这个过程，可以不区分 TEXT 节和 DATA

节，而作为统一的二进制映像进行加载；

- 4、完成二进制映像的加载后，Boot ROM 中的启动代码，需要进一步为操作系统预留 BSS 空间；
- 5、上述一切动作完成，操作系统运行的环境就绪后，Boot ROM 中的代码通过一条转移指令（跳转或 CALL），转移到操作系统的入口点，这样后续系统的运行，就完全在操作系统的控制下进行了。

这种加载方式，把操作系统和应用程序的映像都搬移到了外部存储介质上，不但可以节约 Flash 存储器，而且可以适应大容量的操作系统映像的情况。另外，由于引入了外部存储介质，嵌入式系统运行过程中产生的一些状态数据，比如告警、日志等信息，就可以存储在大容量的外部存储介质上，这样十分便于问题的定位。

但这种方式实现起来相对复杂，需要额外的存储介质访问接口电路（比如，IDE 接口电路），而且需要软件实现针对这些外部存储介质的驱动程序，可能会大大延长开发周期。这种模型，一般应用到一些大型的嵌入式系统设计上，比如，位于 Internet 核心位置的核心路由器（GSR 或 TSR），就可能需要采用这种方式进行设计。

这种方式设计的另外一点好处，就是可以在线升级（系统运行过程中，进行升级操作系统和应用程序软件）。实际上，在第二种方式“从 RAM 中运行代码”中，也可以实现在线升级，但采用外部存储介质的在线升级，会更完善，更利于使用。比如，有的情况下，可能为不同的应用，定制不同的软件和嵌入式操作系统，并编译成不同的模块，这些模块都存放在外部存储器上。在嵌入式设备安装完成后，可以根据需要，选择一种特定的功能进行加载，这时候，就可以通过修改位于外部存储器上的系统配置文件，来告诉系统，缺省情况下加载哪个模块（或版本）。

实际上，PC 就是这样一种系统，对于 PC 机使用的操作系统，一般位于外部存储介质，比如软盘或硬盘上，PC 启动的时候，从这些外部存储介质上加载操作系统。从这点上来说，PC 本身就是一个复杂的嵌入式系统，而且很有典型意义，在嵌入式开发过程中遇到的普遍问题，都可以在 PC 上模拟。因此，对于嵌入式开入门者来说，在 PC 上模拟嵌入式开发环境，以达到学习的目的，是十分可行的。但在嵌入式开发中遇到的一些专业问题，比如特定硬件芯片（比如，网络处理器等）的初始化和应用等，则在 PC 上可能无法模拟。但有了嵌入式开发的基本概念和技能，转而从事这类更专业的开发，所需要的仅仅是一个很短的学习周期而已。

除了我们介绍的几种加载方式之外，在嵌入式开发领域，还有另外的一些加载方式，比如从串口（COM 接口）加载、从以太网接口（Ethernet 接口）加载等，这些加载方式与从外部存储设备加载类似，无非需要 Boot ROM 额外实现一个驱动程序，来完成这些设备数据的读取，在此不进行详细描述。

## 2.1.4 嵌入式系统软件的写入

上面介绍的嵌入式系统软件的加载,都是建立在软件已经被成功的写入到 Flash 或外部存储介质上的基础上的。但这些软件如何被写入这些存储介质上,则是另外一个需要介绍的问题。一般情况下,在硬件电路板开发完成之后,缺省情况下所有存储设备都是空缺的,这些存储设备包括 Boot ROM、Flash、外部存储器等。因此,要想使得硬件运行起来,则必须把相应的软件写入这些存储介质,以完成硬件的控制工作。

这涉及到两个内容:

- 1、Boot ROM 软件的写入,Boot ROM 中的软件是 CPU 复位后,开始执行的第一部分软件,因此只能通过硬件的方式来写入;
- 2、Flash 或外部存储介质软件的写入,即操作系统和应用程序映像的写入。这部分内容在写入前,Boot ROM 中已经被成功写入内容,因此可利用 Boot ROM 软件提供的功能,来把这写系统软件来加载到 Flash 或外部存储介质中。

下面简单介绍这两部分内容的写入方式。

### ● Boot ROM 软件的写入

由于在 Boot ROM 软件写入前,系统是无法启动的,因此无法借用系统提供的软件功能来写入 Boot ROM 代码,只能通过硬件的方式。目前情况下,下列两种方式是常用的两种写入方式:

- 1、通过烧片的方式写入 Boot ROM,即通过特殊的烧片设备,把已经编译好的 Boot ROM 软件,烧入 Boot ROM 芯片中,然后再把 Boot ROM 芯片插入印制电路板上,一般情况下,对于 Boot ROM、Flash 等部件,都是可随意在电路板上插入或拔出的。这种方式简便易行,但需要一种专门的烧片设备,而且需要配以驱动软件;
- 2、通过 JTAG (Join Test Action Group, IEEE 标准) 接口写入。JTAG 接口是一个标准硬件接口,一般的 CPU 都提供。通过 JTAG 接口,可以直接驱动 CPU 执行特定的指令。这样在写入的时候,可以把 JTAG 接口的电缆,通过转换头跟 PC 机的串口或并口进行连接,然后通过 PC 机上的软件,驱动 JTAG 接口,把 Boot ROM 软件写入 Boot ROM 中。

这两种方式都经常使用,但第二种方式更加灵活,用途更加广泛。由于本书内容着重于嵌入式开发的操作系统内容的介绍,因此对这些硬件的实现,不作介绍,感兴趣的读者,可以参考相关的书籍。

## ● 操作系统和应用程序映像的写入

在完成 Boot ROM 的写入后，操作系统和应用程序的映像就可以借助 Boot ROM 提供的服务，来写入 Flash 或外部存储设备了。当然，对于 Flash 设备，也可以采用跟 Boot ROM 类似的方法写入，但通常情况下，是通过调用 Boot ROM 提供的服务写入的。

一般情况下，Boot ROM 软件会提供一些完成操作系统软件加载（这里的加载，指的是把嵌入式软件从所在的计算机，写入嵌入式系统的过程，并不是指操作系统的启动过程）的功能服务，比如串口读写服务、Ethernet 接口驱动、TFTP 服务器等。通过这些服务，操作系统可方便的写入 Flash 或外部存储介质。

在系统启动的时候，Boot ROM 软件会提供一个供用户选择是否更新系统软件的机会，若用户选择了更新软件，则会进入 Boot ROM 更新界面。比如，在 IP 路由器上，启动过程中，若通过串口连接路由器的控制口（比如，Console 接口）和 PC 计算机的中断模拟软件（比如，Windows 操作系统自带的超级终端），则可能会显示如下界面：

```
Booting.....If you want to update system software,please press Ctrl + B in 10 seconds...
```

这样，如果用户在 10 秒之内按了 Ctrl + B 组合键，则会进入如下界面（当然，如果用户不按该组合键，则系统会在等待 10s 之后，进入缺省的启动过程）：

```
Boot loader Interface.  
Please select your option:  
[1] Update software through console  
[2] Update software through Ethernet(tftp)  
[3] Set default boot configuration  
[4] Set Ethernet configuration  
[5] Change bootrom password  
[6] Return to continue boot
```

这样，用户就可以选择是通过 Console 接口（COM 接口）还是以太网接口来更新系统软件。在选择使用 Ethernet 接口更新软件之前，需要首先配置嵌入式系统的以太网接口，比如 IP 地址、网络掩码、缺省网关等参数，配置完成之后，才能通过 TFTP 协议，把软件下载到目标系统上。

上面列出的用户交互内容，都是位于 Boot ROM 内的软件完成输出的，因此，在一个复杂的嵌入式系统中，Boot ROM 的功能往往也十分复杂，因为不但要完成正常的启动

和硬件初始化（这部分代码可能比较少），而且要完成系统软件的更新等功能。更有甚者，还可能在 Boot ROM 中集成软件调试功能。

对于 Boot ROM 代码的编写，一般与硬件相关的内容，比如 CPU 的初始化、系统硬件的初始化等，是由汇编语言完成的，而更复杂的，与硬件无关的代码，则是采用可读性更强的 C 语言编写的。实际上，Boot ROM 软件的开发，也是一项很复杂的工程，有的时候甚至比操作系统和应用软件本身更加复杂。

## 2.2 Hello China 在 PC 机上的启动

### 2.2.1 PC 机启动过程概述

个人计算机（PC）一旦接通电源，设计的电路会出发 CPU 的 reset 引脚，从而导致 CPU 复位，并进入初始化过程。IA32 构架的 CPU 中，CPU 复位后，会根据 CS 寄存器和 EIP 寄存器的值，跳转到地址空间的对应位置，执行 PC 系统的初始化指令。在 IA32 构架的 CPU 中，复位完成之后，各寄存器的初始值如下表：

Register	Pentium 4 and Intel Xeon Processor	P6 Family Processor	Pentium Process
EFLAGS <sup>1</sup>	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H <sup>2</sup>	60000010H <sup>2</sup>	60000010H <sup>2</sup>
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	00000FxxH	000006xxH	000005xxH
EAX	0 <sup>3</sup>	0 <sup>3</sup>	0 <sup>3</sup>
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

图 2-5 Intel CPU 初始化后的寄存器值

可以看出,在复位后,CR0 寄存器的值是 0x60000010,这样的取值,使得 CPU 处于实模式下(禁止分页机制、禁止 cache 和直接写内存),这样 CPU 的寻址范围,就被局限在了 1M 以内,而且对于内存的寻址,在实模式下,也是按照 CS 左移四位,加上 IP 偏移的方式来寻址指令。按照上表中 CS 和 EIP 的值,可以根据这个算法,计算出 CPU 第一条执行的指令的地址是 0xFFFF0。因此,在 PC 计算机系统中,应该把 ROM 系统固化在这个位置上,并把第一条指令放置在 0xFFFF0 处(实际上,IA32 CPU 在复位之后,第一条执行的指令位置,不是通过 CS 偏移加 IP 的方式计算出的,但得出的结果,与这样计算获取的结果一致,因此在此不作实际算法的详细描述)。

在 PC 系统中,CPU 复位之后,跳转到 0xFFFF0 处开始执行,一般情况下,该指令是一条跳转指令,跳转到一段完成硬件初始化的代码处,这段代码完成 PC 计算机所有外围硬件系统的初始化,包括 PCI 总线的初始化、输入/输出设备的初始化等,初始化完成之后,会进入操作系统加载过程。

在本章第一部分中,我们讲述了常见的嵌入式系统启动方式,有一种启动方式,操作系统是从外部存储设备(比如硬盘等)中加载的。实际上,PC 就是这种系统的典型例子,也可以把 PC,看作是一个高度标准化的嵌入式系统。一旦硬件初始化完成,引导程序会根据 BIOS 里面的配置,按照优先级,依次搜索可用的启动设备,比如硬盘、软盘、CD-ROM 等,一旦找到一个可用的启动设备,则 BIOS 尝试把启动设备的第一个扇区读入内存(读入内存的位置为 0x0000:0x07C0),然后判断该扇区是否是启动扇区(通过判断扇区末尾的标志字),若是,则跳转到启动扇区的第一条指令处,开始执行,到此为止,PC 的控制权完全从 BIOS 转移到了操作系统。若不是,则继续检查剩余的可用引导设备。

引导扇区的实现,会因操作系统的不同而有所不同。一般情况下,一个引导扇区的大小是 512 字节,操作系统的初始化操作,不可能放在这么少的代码段中完成。因此,引导扇区的主要工作,是进一步从存储系统上加载其它的操作系统模块,其它的操作系统模块还有可能进一步加载更大的操作系统核心,然后再进入操作系统初始化过程。在操作系统初始化过程中,会根据硬件情况,以及用户的配置,初始化硬件,并加载特定硬件的驱动程序。

### 2.2.2 Hello China 的引导过程

在目前 Hello China 版本的实现中，整个系统是从软盘启动的。根据功能的不同，目前的 Hello China 在 PC 上的实现，由四个二级制模块组成，如下表：

名称	最大大小	用途
BOOTSECT.BIN	512 字节	引导扇区
REALINIT.BIN	4K 字节	实模式下的初始化代码
MINIKER.BIN	48K 字节	保护模式下的初始化代码和输入/输出驱动程序
MASTER.BIN	128K — 560K	操作系统核心功能

表 2-1            Hello China 各组成模块

其中，上述四个二进制模块，被一个特定的程序 FMTLDRF.COM，写到一张标准软盘的固定扇区上（BOOTSECT.BIN 占据了第一个扇区）。BOOTSECT.BIN 是引导扇区，该模块被 BIOS 加载到内存之后，会进一步加载剩余的模块（REALINIT.BIN、MINIKER.BIN 和 MASTER.BIN），完成后，跳转到 REALINIT.BIN 模块处开始执行。

对于一张大小是 1.44M 的高密度软盘，在格式化的时候，被分成了两个盘面，分别对应软驱的两个磁头，每个盘面又进一步被分成了 80 个磁道，每个磁道又被分成 18 个扇区，每个扇区的大小是 512 字节。Hello China 的每个模块，在软盘上的位置（被 FMTLDRF.COM 写入）如下表：

名称	起始位置	结束位置	占用空间大小
BOOTSECT.BIN	0 面 0 道 1 扇区	0 面 0 道 1 扇区	512 字节（1 扇区）
REALINIT.BIN	0 面 0 道 3 扇区	0 面 0 道 10 扇区	4K 字节（8 扇区）
MINIKER.BIN	0 面 0 道 11 扇区	0 面 7 道 4 扇区	64K 字节(128 扇区)
MASTER.BIN	0 面 7 道 5 扇区	0 面 79 道 18 扇区	560K 字节（剩余扇区）

图 2-2 各组成模块在引导盘上的布局

BOOTSECT.BIN 模块根据上述布局，调用 BIOS 提供的软盘读写调用（中断），把 REALINIT.BIN 等三个模块，依次读入内存。读入内存的位置为 0x1000（即 4K 偏移处）。下面是 BOOTSECT.BIN 模块中相关代码（用汇编语言编写，NASM 编译）：

```

gl_start:                                ;;Start label of the boot code.

    cli                                  ;;Mask all maskable interrupts.
    mov ax,DEF_ORG_START
    mov ds,ax
    mov ss,ax
    mov sp,0xffff0

    cld
    mov si,0x0000
    mov ax,DEF_BOOT_START
    mov es,ax
    mov di,0x0000
    mov cx,0x0200                        ;;The boot sector's size is 512B
    rep movsb

    mov ax,DEF_BOOT_START                ;;Prepare the execute context.
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,0x0ffe
    jmp DEF_BOOT_START : gl_bootbgn     ;;Jump to the DEF_BOOT_START to
execute.

```

上述代码是 BOOTSECT.BIN 模块的开始部分，该部分代码的功能，是把 BOOTSECT.BIN 模块，从内存的 0x07C0 偏移处，搬移到 0x9F000 处（即 636K 处），然后跳转到该位置，正式开始执行。其中，DEF\_ORG\_START 是预定义的一个宏，定义为

0x07C0，即引导扇区被 BIOS 加载道内存后的地址，而 DEF\_BOOT\_START 则被定义为 0x9F00，是 BOOTSECT.BIN 被重新搬移到的位置。

gl\_bootbgn 标号处的汇编语句，打印出一串提示信息，然后调用 np\_load 过程，完成操作系统相关模块（即除 BOOTSECT.BIN 之外的三个模块）的加载过程，代码如下：

```
gl_bootbgn:
    call np_printmsg           ;;Print out the process message.
    call np_load
    jmp DEF_RINIT_START / 16 : 0 ;;Jump to the real mode initialize code.
```

加载完毕之后，使用一个远跳转指令，跳转到 REALINIT.BIN 模块处开始执行。下面是加载过程 np\_load 的相关代码：

```
np_load:                                ;;This procedure use the int 13 interrupt
                                        ;;call,load the operating system kernal
                                        ;;into memory.

    push es
    mov ax,0x0000
    mov es,ax
    mov bx,DEF_RINIT_START
    xor cx,cx

.ll_start:
    mov ah,0x02
    mov al,0x02                        ;;Load 2 sector for one time.
                                        ;;So,the sector's number of per track,
                                        ;;the total sectors of the whole system
                                        ;;code must be 2 times.

    mov ch,byte [curr_track]
    mov cl,byte [curr_sector]
    mov dh,byte [curr_head]
    mov dl,0x00
    int 0x013
    jc .ll_error
    dec word [total_sector]
```

```

    dec word [total_sector]
    jz .ll_end

    cmp bx,63*1024                ;;If the buffer reaches 64k boundry,we must
                                   ;;reinitialize it.

    je .ll_inc_es
    add bx,1024
    jmp .ll_continue1
.ll_inc_es
    mov bx,es
    add bx,4*1024
    mov es,bx                    ;;Update the es register to another 64k b-
                                   ;;oundry.

    xor bx,bx

.ll_continue1:
    inc byte [curr_sector]
    inc byte [curr_sector]
    cmp byte [curr_sector],DEF_SECT_PER_TRACK ;;If we have read one track,
                                                ;;must change the track
number.
    jae .ll_inc_track
    jmp .ll_start
.ll_inc_track:
    mov bp,es
    mov word [tmp_word],bp      ;;Print out the process message.
                                   ;;Because of the boring of the
                                   ;;bios call,it use registers
                                   ;;to pass parameter,so here,
                                   ;;we must save the es register
                                   ;;to a variable.

    pop es
    call np_printprocess
    push es
    mov bp,word [tmp_word]

```

```

    mov es,bp

    mov byte [curr_sector],0x01
    inc byte [curr_track]
    cmp byte [curr_track],DEF_TRACK_PER_HEAD
    jae .ll_inc_head
    jmp .ll_start
.ll_inc_head:
    mov byte [curr_track],0x00
    inc byte [curr_head]
    cmp byte [curr_head],0x02
    jae .ll_end
    jmp .ll_start

.ll_error:                                ;;If there is an error,enter a dead loop.
    mov dx,0x03f2
    mov al,0x00
    out dx,al
    pop es
    call np_deadloop
.ll_end:
    mov dx,0x03f2                        ;;The following code shut off the FDC.
    mov al,0x00
    out dx,al
    pop es
    ret                                  ;;End of the procedure.

```

这段代码比较长，但功能比较简单，就是完成 **REALINIT.BIN**、**MINIKER.BIN** 和 **MASTER.BIN** 三个模块的加载工作。之所以导致代码较长，是因为这三个模块分部在软盘的一个整面上，跨越了多个磁道和多个扇区，这样在加载过程中，必须判断是否跨越磁道和盘面。在加载的过程中，每加载两个扇区，就需要打印出一个点，以提示用户加载正在进行。其中，**curr\_sector**、**curr\_track**、**curr\_head** 是定义的三个字节变量，用于存储当前正在读写的起始扇区号、磁道号和盘面号。每完成一次读盘操作，**np\_load** 过程就递增 **curr\_sector** 变量（一次递增 2），若该变量超过了 18（每磁道扇区数），则重新初始

化该变量为 0，并递增 `curr_track` 变量，相应地，若 `curr_track` 变量达到了 80（每盘面最大磁道数），则重新初始化该变量和 `curr_sector` 变量，并递增 `curr_head` 变量。

上述代码中，`DEF_RINIT_START` 是一个预定义的宏，定义为 `0x1000`（4K），这也是三个操作系统模块被加载到内存后的初始地址。需要注意的是，为了方便，`BOOTSECT.BIN` 不区分加载的具体模块，而采取一次读取的策略，把磁盘上 `REALINIT.BIN` 等三个模块一次性读入内存，这也是为什么 `MINIKER.BIN` 实际大小是 48K，而写到磁盘上时，却占用了 64K 空间的原因，就是为了满足三个模块在磁盘上的相对位置，和内存中的相对位置能够保持一致。

具体的磁盘读写操作所采用的 BIOS 调用，在此不作赘述，读者可通过查阅 BIOS 调用手册获取相关信息。

### 2.2.3 实地址模式下的初始化

Hello China 引导完成之后，内存布局如下图：

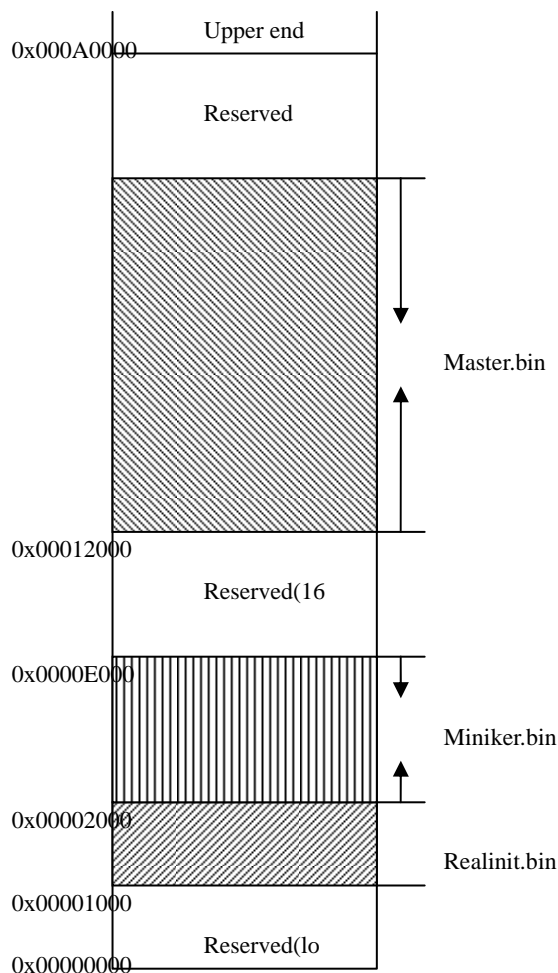


图 2-6 各模块加载到内存后的布局

图中，所有地址都是物理内存地址，由于这时候 CPU 尚工作在实模式下，因此对内存的访问，采用段基址加段偏移的方式，形成 20 位地址，直接定位到物理内存。

引导完成之后，引导程序通过一条 `jmp` 指令，跳转到 `Realinit.bin` 开始处(0x00001000)开始执行，下面是 `Realinit.bin` 的开始部分代码，是采用 NASM 编译的，目标格式为 BIN 格式，即纯粹的二进制可执行文件，不带任何文件头。为了便于理解，我们分段解释：

```

bits 16                                ;;The real mode code.
org 0x0000

%define DEF_RINIT_START 0x01000  ;;Start address of this module.
                                   ;;This code is loaded into memory by
                                   ;;boot sector,and resides at 0x01000.

%define DEF_MINI_START 0x02000  ;;The start address of the mini-kernal
                                   ;;when it be loaded into memory.

```

上面部分代码，除了告诉编译器，代码工作在实模式（16 位），偏移地址为 0 外，还定义了两个宏，其中第一个 `DEF_RINIT_START`，用于指出 `realinit.bin` 被加载到内存后的物理地址，第二个宏 `DEF_MINI_START`，定义了 `miniker.bin` 模块被加载到内存后的物理地址。这两个宏定义，一个用来计算代码段寄存器的值，一个用来作为跳转目标地址，在 `realinit.bin` 执行完毕后，跳转的目标地址。

```

gl_initstart:
    mov ax,DEF_RINIT_START           ;;The following code prepare the execute
                                     ;;context.

    shr ax,4
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,0x0fff

```

上述代码初始化了代码段寄存器、堆栈段寄存器、数据段寄存器和扩展段寄存器。其中，`CS`、`ES`、`SS` 和 `DS` 初始化为相同的值，都是指向 `realinit.bin` 在内存中的起始地址，这样 `realinit.bin` 就可以不用考虑自己在内存中的位置，直接从 0 偏移开始执行了。对于堆栈寄存器的值，设置为 `0x0FFF`，即相对于段寄存器，偏移约 4K 处，如下图：

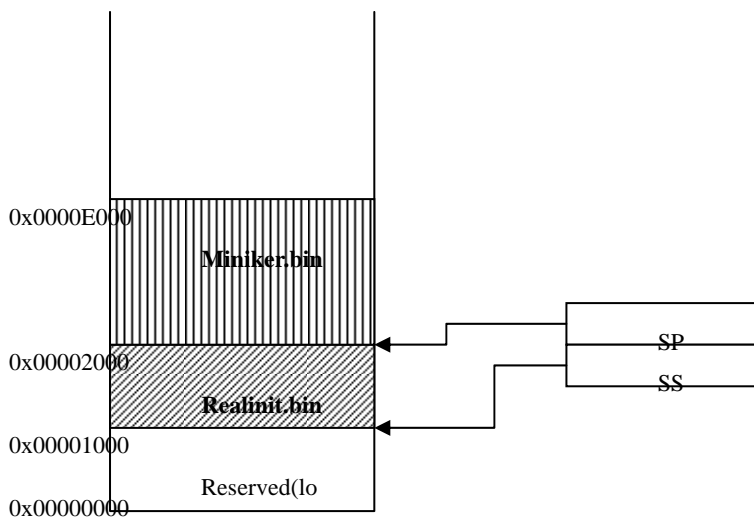


图 2-7 SP 和 SS 寄存器的初始化

因为按照目前的实现，为 `realinit.bin` 预留了 4K 的空间，但实际上，该模块的大小，尚不超过 2K，因此，4K 空间的上面部分 2K 作为堆栈区，是不会有问题的。

```

mov ax,okmsg
call np_strlen
mov ax,okmsg
call np_printmsg
mov ax,initmsg
call np_strlen
mov ax,initmsg
call np_printmsg

```

上述代码调用 `realinit` 模块里面定义的几个函数，打印出一些字符串，提示操作的进度，以及操作的结果。需要注意的是，在 `Hello China` 正常启动过程中，这些字符串是看不到的，不是因为没打印出来，而是因为该模块内定义的功能，很快就执行完了，转

而跳转到 `miniker` 模块，而在 `miniker.bin` 模块的开始处，马上做了一个清屏操作，所以这些信息在正常情况下是看不到的。但若执行过程中发生错误，进入了死循环，这些信息就可以看到了。

```
;;The following code initializes the system hardware.
call np_init_crt           ;;Initialize the crt display.
call np_init_keybrd       ;;Initialize the key board.
call np_init_dmac         ;;Initialize the DMA controller.
call np_init_8259         ;;Initialize the interrupt controller.
call np_init_clk          ;;Initialize the clock chip.
call np_get_syspara       ;;Gather the system parameters.
```

上述代码初始化了系统中的一些关键硬件。在基于 PC 机的实现中，初始化的硬件包括 CRT 显示器、键盘、DMA 控制器、中断控制器（8259 芯片）、时钟，并收集了系统的一些硬件配置信息，比如物理内存的大小等。上述每个操作，都对应 `realinit.bin` 模块内定义的一个函数。若把 `Hello China` 移植到其它的非 PC 系统，也可以在这个地方对特定目标系统的硬件进行初始化。

```
call np_act20addr         ;;First,activate the above 12 address lines.
```

上面这个过程调用，用来激活 A20 地址线。这在 PC 机上十分关键，因为只有激活了 A20 地址线，才能确保 CPU 的实模式下，可以访问到所有的 32 位物理地址。这其中的原因，在很多资料上都有描述，在此不作赘述。

```
xor eax,eax
mov ax,ds
shl eax,0x04
add eax,gl_gdt_content    ;;Form the line address of the gdt content.
mov dword [gl_gdt_base],eax
lgdt [gl_gdt_ldr]        ;;Now,load the gdt register.
```

上述代码完成 gdt 寄存器（全局描述表寄存器）的初始化，gdt 寄存器指向一个全局描述表，这个表定义了 CPU 保护模式下正常工作所需要的段。

```

mov eax,cr0
or eax,0x01                ;;Set the PE bit of CR0 register.
mov cr0,eax                ;;Enter the protected mode.
jmp dword 0x08 : DEF_MINI_START  ;;Transant the control to Mini Kernal.

```

上面的代码，完成 CPU 工作模式的转移功能，即从实地址模式，转移到保护模式。在 IA32 CPU 中，有一个控制寄存器（CR0），该寄存器的第一个比特（PE，Protected Enable），控制了 CPU 工作在何种模式下。若该比特为 1，则 CPU 工作在保护模式下，否则工作在实地址模式下。

完成模式转换之后，通过一条远转移指令，转移到 miniker.bin 模块的开始处，继续运行。这条指令的作用，不但完成执行路径的转换，而且还完成 CPU 上下文的刷新工作，比如，刷新 CPU 的指令预取队列，刷新 CPU 的本地 cache 等。需要注意的是，上述指令是在保护模式下运行的，0x08 指明了代码段在段描述表（全局描述表）中的偏移，而 DEF\_MINI\_START 则指明了 miniker.bin 模块在内存中的偏移地址。在 Hello China 的定义中，代码段的基址是 0，因此，根据代码段基址和偏移，形成的目标地址，就是 DEF\_MINI\_START。

到此为止，realinit.bin 模块就执行完了，总结一下，该模块主要完成下列工作：

- 1、初始化 PC 机中关键的系统硬件；
- 2、转换到保护模式；
- 3、跳转到 MINIKER.BIN 模块开始处，继续执行。

## 2.2.4 保护模式下的初始化

实模式下的初始化完成之后，CPU 已经进入保护模式，并跳转到 MINIKER.BIN 模块的开始处继续执行。下面是 MINIKER.BIN 模块的开始部分代码：

```

bits 32                    ;;The mini-kernal is a pure 32 bits OS kernel.
org 0x00100000

mov ax,0x010
mov ds,ax
mov es,ax

```

```

        jmp gl_sysredirect          ;;The first part of the mini-kernal image is
                                   ;;data section,so the first instruction must
                                   ;;to jump to the actually code section,which
                                   ;;start at gl_sysredirect.

```

上述代码，明确的指示编译器，编译成 32 位指令代码，即保护模式下指令，且偏移地址设定为 1M 开始处。MINIKER.BIN 模块，也是被编译成纯二进制可执行文件格式，没有任何特定的文件头部信息。

代码重新初始化 DS 和 ES 寄存器，在 REALINIT.BIN 中，转换到保护模式之后，仅仅初始化了 CS 寄存器（JMP 指令完成），此处对 DS 和 ES 寄存器进行初始化，为代码的继续执行建立环境。然后，又通过一条跳转指令，跳转到标号为 gl\_sysredirect 的指令处，开始执行。在上述代码和 gl\_sysredirect 标号之间，定义了一些全局的数据结构，包括全局描述表、中断描述表等。

下面是 gl\_sysredirect 标号处的代码：

```

align 4
gl_sysredirect:                ;;Redirect code of mini-kenal,this code
                               ;;moves the mini-kernal from con_org_st-
                               ;;art_addr to con_start_addr.

    mov ecx,con_mini_size + con_mast_size
    shr ecx,0x02
    mov esi,con_org_start_addr  ;;Original address.
    mov edi,con_start_addr      ;;Target address.
    cld
    rep movsd

    mov eax,gl_initgdt
    jmp eax                    ;;After moved mini-kernal to the start
                               ;;address,the mini-kernal then jump to
                               ;;the start entry,laabeled by gl_initgdt.

```

上述代码首先把 MINIKER.BIN 和 MASTER.BIN 两个模块，从最初位置（加载后的位置，位于低端的 1M 内存范围内），重新搬移到 1M 物理内存开始处（此时 CPU 工作在 32 位模式下，可以访问 32 位地址空间的任何位置）。其中，con\_mini\_size 和 con\_mast\_size

是两个预定义的宏，分别指出了 MINIKER.BIN 和 MASTER.BIN 两个模块的长度，这样把这两个模块的长度相加，便是要搬移的大小。然后执行 movsd（转移 32 比特的字符串）指令，并以 rep 为前缀，一次性把 MINIKER.BIN 和 MASTER.BIN 搬移到指定位置。搬移完成后，内存的布局如下图所示：

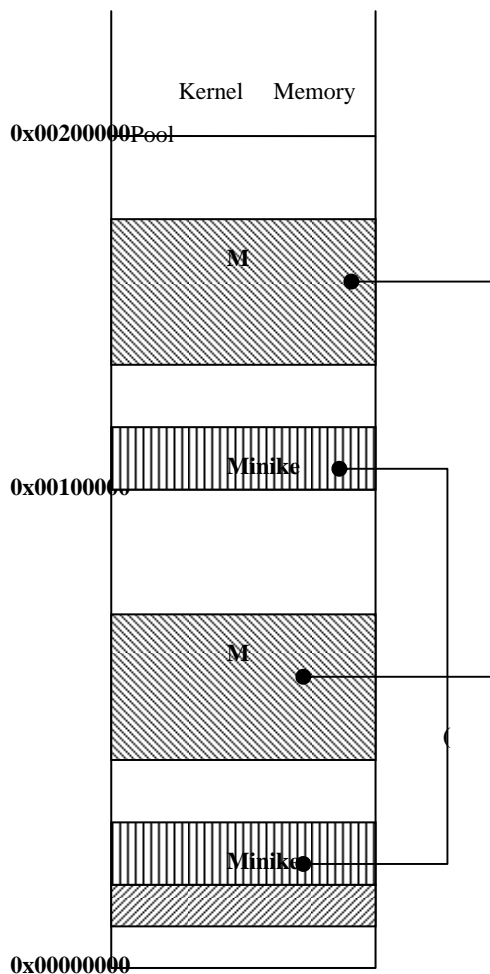


图 2-8 进入保护模式后相关模块的搬移

上述搬移完成之后，紧接着又通过一条跳转指令（绝对位置跳转），跳转到标号为 `gl_initgdt` 位置开始运行。需要注意的是，这时候 `MINIKER.BIN` 已经被搬移到了 1M 开始处，因此，后续的执行，必须也相应的调整到新的 `MINIKER.BIN` 所在的位置。若通过通常的跳转指令，直接以标号为参数，则不能正常工作，因为标号为参数，只能是一个相对位置的跳转，即跳转到当前位置，加上当前位置到标号的偏移量所在的位置，而不是跳转到标号的绝对位置。因此，必须采用绝对跳转，即直接跳转到标号所在位置。这样，采用寄存器作为参数，来执行 `JMP` 指令，可以实现绝对跳转。详细信息，可参考 Intel CPU 的指令手册。

顾名思义，`gl_initgdt` 标号开始的代码，应该是用来完成初始化 GDT 的，如下所示：

```
gl_initgdt:                                ;;The following code initializes the GDT
                                           ;;and all of the segment registers.

    lgdt [gl_gdtr_ldr]                    ;;Load the new gdt content into gdt regis-
                                           ;;ter.

    mov ax,0x010
    mov ds,ax
    mov ax,0x018
    mov ss,ax
    mov esp,DEF_INIT_ESP                  ;;The two instructions,mov ss and mov esp
                                           ;;must resides together.

    mov ax,0x020
    mov es,ax
    mov fs,ax                             ;;Initialize the fs as the same content as
                                           ;;es.If need,we can change the fs's value.

    mov ax,0x020
    mov gs,ax
    jmp dword 0x08 : gl_sysinit            ;;A far jump,to renew the cs register's v-
                                           ;;alue,and clear the CPU's prefetched que-
                                           ;;ue,trans the control to the new squence.
```

这段代码重新初始化了 GDT 寄存器，这时候的初始化，不但对 DS、ES 等寄存器做了初始化，而且还初始化了 SS 寄存器和 ESP 寄存器，这样后续代码就可以执行调用（CALL）指令了。需要注意的是，上述对各段寄存器的初始化，虽然采用了不同的段描

述符，但所有段描述符的基址和长度，都是相同的，因为目前 **Hello China** 的实现，采用了平展模式，每个段都可以完整覆盖整个线性地址空间。详细信息，请参考“**Hello China** 的内存管理机制”一章。

对于堆栈寄存器（ESP）的初始化，是直接把一个预先定义的值 **DEF\_INIT\_ESP** 装入了 ESP 寄存器，这个值目前定义为 **0x13FFFFFF**，即物理内存 20M 地址处。

完成 GDT 的初始化，以及相关寄存器的初始化后，又通过一条远跳转指令，跳转到 **gl\_sysinit** 处继续执行。这条指令不但完成了执行路径的转移，而且更新了 CS 寄存器，并更新了 CPU 的内部上下文，包括指令预取队列、CACHE 等。下面是 **gl\_sysinit** 的实现：

```
gl_sysinit:                                ;;The start position of the init process.
    mov eax,gl_trap_int_handler
    push eax
    call np_fill_idt                        ;;Initialize the IDT table.
    pop eax
    lidt [gl_idtr_ldr]                     ;;Load the idtr.
    call np_init8259                       ;;Reinitialize the interrupt controller.
    sti
    nop
    nop
    nop
    mov eax,con_mast_start
    jmp eax
```

上述代码调用一个本地过程 **np\_fill\_idt**，用来完成中断描述符表的初始化，然后初始化 **idtr** 寄存器（**lidt** 指令），并重新初始化了中断控制寄存器，这样做的目的是，在实模式下，中断控制器的中断向量，是从 0 开始的，而一旦转移到保护模式下，外部中断却是从 32 开始，因此，必须对中断控制器进行重新编程，以产生新的中断向量。

完成上述功能后，该过程使用中断（**sti** 指令），并采用一个绝对跳转指令，跳转到 **con\_mast\_start** 处开始执行。**con\_mast\_start** 是一个预定义的宏，指明了 **MASTER.BIN** 模块所在的内存位置（物理内存位置）。至此，**MINIKER.BIN** 模块执行完毕，控制转移到 **MASTER.BIN** 模块。

至此，采用汇编语言部分实现的功能，已经执行完毕，这部分也是特定硬件平台相关的。后续所有功能，都是采用 C 语言实现的机器无关部分功能。

在嵌入式开发领域，往往把整个软件分成两部分：**BSP** 和应用代码部分。其中 **BSP** 是单板支撑包的缩写，一般情况下，在 **BSP** 中，实现了特定硬件相关的初始化代码，以及特定硬件的驱动程序，这样是为了提升整个系统的可移植性。因为在把代码移植到不同的硬件平台的时候，从理论上说，只需要修改 **BSP** 部分就可以了（实际上远没有这么方便）。在 **Hello China** 的实现中，可以把 **REALINIT.BIN** 和 **MINIKER.BIN** 两个模块，看作是 **BSP**，因为在这两个模块中，完成了对特定硬件平台（PC 机）的硬件初始化功能，而且在 **MINIKER.BIN** 中，还实现了针对标准 PC 显示器和 PC 键盘的驱动程序，这样在 **MASTER.BIN** 模块中，就不用考虑这些硬件的差异，而直接通过特定的接口，调用硬件提供的服务。

**MASTER.BIN** 模块是 **Hello China** 的核心模块，是所有操作系统核心功能的实现模块。该模块也完成一些初始化工作，但这些初始化工作，不是特定硬件的初始化，而是操作系统正常工作的核心数据结构和核心数据对象的初始化。在完成这些初始化工作后，在 PC 平台上，操作系统启动一个 **shell** 线程，用来完成跟用户的交互，到达这一步后，操作系统才算成功启动完毕。

### 2.2.5 操作系统核心功能的初始化

在 **MINIKER.BIN** 模块初始化完成之后，通过一条跳转指令，直接跳转到 **MASTER.BIN** 开始处继续执行。需要注意的是，**MASTER.BIN** 是采用 **Windows** 操作系统下的 **C++** 编译器编译的，编译结果为 **PE** 文件格式，这种文件格式的最开始部分，是一个 **PE** 文件头，并不是可执行的二进制代码。因此，我们通过一个特殊的工具（也是采用 **C** 语言编写的 **PE** 文件处理工具），把 **PE** 文件的开头部分，采用可执行的指令替换，并从 **PE** 头中，提取出文件的入口地址，然后采用一条跳转指令，跳转到入口处执行。下面是 **MASTER.BIN** 文件的开始部分（**MASTER.BIN** 文件已经经过处理，下面是对 **MASTER.BIN** 进行反汇编所得结果的开头部分）：

```
00000000  90                nop
00000001  90                nop
00000002  90                nop
00000003  E9E8500000      jmp 0x50f0
00000008  0400             add al,0x0
0000000A  0000             add [eax],al
```

```

0000000C  FF                db 0xFF
0000000D  FF00             inc dword [eax]
0000000F  00B800000000     add [eax+0x0],bh

```

上述代码中，关键的一条 `jmp` 指令，跳转到了 `MASTER.BIN` 的入口处。

下面是 `MASTER.BIN` 入口函数的实现代码，为了便于阅读，我们分段解释：

```

void __init()
{
    __KERNEL_THREAD_OBJECT*    lpIdleThread    = NULL;
    __KERNEL_THREAD_OBJECT*    lpShellThread   = NULL;
    __KERNEL_THREAD_OBJECT*    lpKeeperThread  = NULL;
    DWORD                      dwKThreadID     = 0;

```

```

DisableInterrupt();    //The following code is executed in no-interruptable environment.

```

`DisableInterrupt` 函数禁止了外部可屏蔽中断，实际上，在 `MINIKER.BIN` 的初始化过程中，已经禁止了中断，在此又重新做一个禁止中断操作，是为了编码上的统一，因为在该函数的尾部，会调用 `EnableInterrupt` 函数启用中断。

```

ClearScreen();          //Print out welcome message.
PrintStr(pszStartMsg1);
PrintStr(pszStartMsg2);
ChangeLine();
GotoHome();

```

上述几个函数做了一个清屏操作，然后打印出了两行提示信息，以指示用户目前系统引导状态。

```

g_keyHandler = SetKeyHandler(_KeyHandler);    //Set key board handler.

```

`SetKeyHandler` 函数用于设置键盘中断处理程序，在 `Hello China` 目前版本的实现中，对于键盘驱动程序，是在 `MINIKER.BIN` 模块里实现的，这样用户按键消息，最初会被

MINIKER.BIN 模块捕获，为了把按键消息传递给 MASTER.BIN 模块，设计了一个回掉机制，即在 MASTER.BIN 中实现一个处理函数（该函数就是\_KeyHandler），把该函数的地址，传递给 MINIKER.BIN 模块中的一个变量（该变量位于 MINIKER.BIN 末尾的特定位置），这样一旦发生键盘中断事件，MINIKER.BIN 模块就以适当的参数调用该函数，这样 MASTER.BIN 就可以接收到这个按键事件，从而做进一步处理。

```
*(__PDE*)PD_START = NULL_PDE;
```

上述代码用于完成页索引对象的初始化工作，详细信息，请参考“Hello China 的内存管理机制”一章。

```
#ifdef __ENABLE_VIRTUAL_MEMORY    //Should enable virtual memory model.

    lpVirtualMemoryMgr                                =
    (__VIRTUAL_MEMORY_MANAGER*)ObjectManager.CreateObject(&ObjectManager,
        NULL,
        OBJECT_TYPE_VIRTUAL_MEMORY_MANAGER);          //Create virtual
memory manager object.
    if(NULL == lpVirtualMemoryMgr)    //Failed to create this object.
        goto __TERMINAL;

    if(!lpVirtualMemoryMgr->Initialize((__COMMON_OBJECT*)lpVirtualMemoryMgr))
        goto __TERMINAL;
#endif
```

上述代码创建针对整个系统的虚拟内存管理器，并对之进行初始化。在 Hello China 的实现中，为了对虚拟内存进行管理，特实现了一个虚拟内存管理器（Virtual Memory Manager）的对象，用于对系统或单个进程的地址空间进行管理。目前的实现，尚没有实现进程机制，因此整个系统只有一个虚拟内存管理器。但在未来的实现中，可能会引入进程模型，这样系统中就可能存在多个虚拟内存管理器对象（每进程一个），因此，没有把虚拟内存管理器对象作为全局对象实现，而是作为一个核心对象来实现，虽然目前情况下，整个系统只有一个虚拟内存管理器对象。另外需要注意的是，虚拟内存功能（在 IA32 构架 CPU 的实现中，表现为分页机制）是一个可选的实现模块，通过预先定义的一个宏 \_\_ENABLE\_VIRTUAL\_MEMORY 来控制，若在代码中定义了该宏，在编译操作

系统核心的时候，虚拟内存管理功能就会被包含，若没有定义该宏，则不会包含虚拟内存管理功能。

```

if(!KernelThreadManager.Initialize((__COMMON_OBJECT*)&KernelThreadManager))
    goto __TERMINAL;

if(!System.Initialize((__COMMON_OBJECT*)&System))
    goto __TERMINAL;

if(!PageFrameManager.Initialize((__COMMON_OBJECT*)&PageFrameManager,
    (LPVOID)0x02000000,
    (LPVOID)0x09FFFFFF))
    goto __TERMINAL;

if(!IOManager.Initialize((__COMMON_OBJECT*)&IOManager))
    goto __TERMINAL;

if(!DeviceManager.Initialize(&DeviceManager))
    goto __TERMINAL;

lpIdleThread = KernelThreadManager.CreateKernelThread(    //Create system idle
thread.
    (__COMMON_OBJECT*)&KernelThreadManager,
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_LOWEST,                                //Lowest
priority level.
    SystemIdle,
    (LPVOID>(&dwIdleCounter),
    NULL);

if(NULL == lpIdleThread)
{

```

```

        //PrintLine("Can not create idle kernel thread,please restart the system.");
        __ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,NULL);
        goto __TERMINAL;
    }

    lpShellThread = KernelThreadManager.CreateKernelThread(    //Create shell thread.
        (__COMMON_OBJECT*)&KernelThreadManager,
        0L,
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_NORMAL,
        SystemShell,
        NULL,
        NULL);

    if(NULL == lpShellThread)
    {
        //PrintLine("Can not create system shell thread,please restart the system.");
        __ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,NULL);
        goto __TERMINAL;
    }

    g_lpShellThread = lpShellThread;    //Initialize the shell thread global variable.

    if(!DeviceInputManager.Initialize((__COMMON_OBJECT*)&DeviceInputManager,
        NULL,
        (__COMMON_OBJECT*)lpShellThread))
        //Initialize the DeviceInputManager object.
    {
        __ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,NULL);
        goto __TERMINAL;
    }
}

```

上述代码完成了下列两项初始化功能：

- 1、 创建了空闲线程（Idle Thread）和用户交互线程（Shell Thread）。其中，空闲线程在 CPU 空闲的时候被调度，用户线程用于完成用户界面功能。其中，Idle 线程必须被

创建，以完成 CPU 空闲时的处理，而 shell 线程则根据需要创建。在基于 PC 环境的 Hello China 中，shell 用于完成用户输入/输出功能，若移植 Hello China 到其它硬件环境，shell 线程则可根据需要决定是否创建；

- 2、完成全局对象的初始化。所谓全局对象，就是整个系统运行环境只存在一个的对象，这些对象一般用于对整个系统中特定部分资源的统一管理。任何一个全局对象初始化失败，都将会导致系统停止启动，进入死循环。下表列举了上述初始化的全局对象，以及这些对象的功能：

名称	变量名	功能
核心线程管理器	KernelThreadManager	完成线程的管理工作，比如创建、挂起、恢复运行等，并为应用程序提供接口。
系统对象	System	完成系统资源的统一管理工作，比如中断管理、定时器管理等。
页框管理器	PageFrameManager	用于完成物理内存页面的分配、回收等工作。
输入/输出管理器	IOManager	输入输出管理，并提供应用程序接口。
设备管理器	DeviceManager	物理设备管理，完成系统资源（IO 端口、内存映射区域等）的统一分配和回收，并统一管理系统中的所有硬件设备。
设备输入管理器	DeviceInputManager	完成键盘、鼠标等主动输入设备的输入管理，把这些设备的输入，定向到当前焦点线程。

表 2-3 核心设备管理对象

这些对象的详细功能，以及其实现方式等，将会在后面章节进行详细介绍，这也是本书的重点内容。需要注意的是，DeviceInputManager 对象是在 shell 线程创建之后才初始化的，因为该对象的初始化函数，需要有一个具体的线程作为当前焦点线程（也可以不指定焦点线程），这样后续的任何主动输入（键盘、鼠标等用户交互设备的输入），都可以被定向到当前焦点线程。在当前的实现中，shell 线程被作为当前焦点线程，即任何用户输入，首先被 shell 感知，然后由 shell 做进一步处理，这是符合 shell 线程的功能的。当然，可以根据需要，采用其它线程来替代 shell 线程，以作为当前焦点线程。比如，可

以把 **Hello China** 移植到一个手持设备上, 这样需要实现一个交互式的图形界面。这时候, 可以把这个交互式的界面, 以一个线程的形式实现, 并把该线程作为当前焦点线程, 这样任何输入, 都可以定向到该线程, 从而完成用户和设备的交互。

```
#ifdef __ENABLE_VIRTUAL_MEMORY
//
//Now,we enable the page mechanism.
//
__asm{
    push eax
    mov eax,PD_START
    mov cr3,eax
    mov eax,cr0
    or eax,0x80000000
    mov cr0,eax
    pop eax
}
#endif
```

上述代码完成了 IA32 CPU 环境下, 分页机制的使能。在此之前, 所有对内存的访问, 都是把线性地址直接映射到物理地址的, 在使能分页功能之后, 对内存的访问, 将经过分页机制的映射。在当前的实现中, 把线性地址空间的前 20M, 依然映射到物理内存的前 20M, 这样可实现分页功能对操作系统代码的透明程度。当然, 分页机制是否使能, 是可以通过定义宏 `__ENABLE_VIRTUAL_MEMORY` 来进行控制的。

```
SetTimerHandler(GenericIntHandler);
```

上述代码用于连接通用中断处理程序和中断。在当前的实现中, 对所有的中断处理, 都是采用同一个函数 `GenericIntHandler` 作为入口的, 然后 `GenericIntHandler` 再调用 `System` 对象的相应函数, 完成中断的进一步分发。在 **Hello China** 的当前实现中, `GenericIntHandler` 是在 `MASTER.BIN` 模块中实现的, 而所有的中断描述表 (IDT), 则是定义在 `MINIKER.BIN` 中, `SetTimerHandler` 函数, 完成连接 `GenericIntHandler` 和 `MINIKER.BIN` 模块中的中断处理程序的功能。从名字上看, 该函数似乎是完成时钟中断的连接, 这是由于历史原因造成, 目前情况下, `SetTimerHandler` 完成任何中断和通用中断处理程序的

连接。对于中断的详细信息，请参考“中断和定时处理机制的实现”一章。

```
StrCpy("[system-view",&HostName[0]);
EnableInterrupt();
DeadLoop();

//The following code will never be executed if corrected.
__TERMINAL:
ChangeLine();
GotoHome();
//PrintStr("STOP : An error occured,please power off your computer and restart it.");
__ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,"Initializing process failed!");

DeadLoop();
}
```

上述代码打印出提示符（机器名），并启用中断，然后进入一个死循环。这个死循环的作用，是为了等待一个时钟中断发生后，开始正式调度线程。实际上，系统初始化过程的代码，包括 **REALINIT.BIN**、**MINIKER.BIN** 等模块，不属于任何线程，或者可以看作是一个初始化线程，系统一旦初始化完毕，这个初始化线程就算运行完了，这时候，如果不进入一个死循环，则 `__init` 函数返回后，可能会导致 CPU 进入一个不确定的状态，从而导致系统崩溃。需要注意的是，这个死循环，并不会真正导致系统死循环，一旦时钟中断发生，线程调度程序会选择一个状态为就绪的线程（`Idel` 或 `shell`），重新投入运行，这样初始化线程就算正式结束了。

最后部分的代码是出错处理部分。在初始化过程中，遇到任何一个错误，就可能导致初始化失败，然后 `__init` 函数跳转到 `__TERMINAL` 标号处，打印出一个出错信息，然后进入死循环，这时候必须采用关闭电源的方式，对计算机进行重新引导。

到此为止，**Hello China** 的启动就算完成了，这之后，`shell` 线程将得到调度，从而完成用户和计算机之间的交互。

## 第三章 Hello China 的 SHELL

——The shell of Hello China

## 3.1 Hello China 的 SHELL

### 3.1.1 Shell 的启动和初始化

在 Hello China 初始化过程中，完成全局对象的初始化后，会创建一个 shell 线程，用于完成用户交互功能，如下：

```
lpShellThread = KernelThreadManager.CreateKernelThread(    //Create shell thread.
    (__COMMON_OBJECT*)&KernelThreadManager,
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_NORMAL,
    SystemShell,
    NULL,
    NULL);
```

其中，CreateKernelThread 函数是 KernelThreadManager 提供的一个接口，用于完成核心线程的创建功能，在“Hello China 的线程”一章中，会进行详细描述。该函数创建一个用户界面（shell）线程，线程的入口点，即线程的功能函数，是 SystemShell。该函数是一个封装函数，直接调用了 EntryPoint 例程。在 EntryPoint 例程中，事先了线程的消息处理功能，代码如下：

```
DWORD EntryPoint()
{
    //__KTHREAD_MSG Msg;
    __KERNEL_THREAD_MESSAGE KernelThreadMessage;

    PrintPrompt();

    while(TRUE)
    {
        if(GetMessage(&KernelThreadMessage))
        {
            if(KTMSG_THREAD_TERMINAL == KernelThreadMessage.wCommand)
```

```

        goto __TERMINAL;
    DispatchMessage(&KernelThreadMessage,EventHandler);
}
}

__TERMINAL:                                //Terminal code here.

return 0L;
}

```

其中，`KernelThreadMessage` 是一个核心线程消息数据结构，线程之间的消息交互，都是通过该数据结构进行的。该结构的定义如下：

```

struct __KERNEL_THREAD_MESSAGE{
    WORD        wCommand; //Message type,such as Keyboard,Mouse,etc.
    WORD        wParam;
    DWORD       dwParam;
};

```

### 3.1.2 Shell 的消息处理过程

缺省情况下，一个线程在创建时候，会同时创建一个消息队列，该消息队列跟线程的控制结构（核心线程对象）进行绑定。其它的线程，可以调用 `SendMessage`，向该线程发送消息，发送的消息，其类型就是 `__KERNEL_THREAD_MESSAGE`。线程可以调用 `GetMessage` 函数，从自己的消息队列中获取消息，该函数以 `KernelThreadMessage` 的地址（指针）为参数。若该函数成功的从消息队列中获取了一个消息，则返回 `TRUE`，`KernelThreadMessage` 结构体里面，就存储了所获得的消息的内容，否则，若当前线程消息队列为空，则该函数会阻塞当前线程。对于线程之间的消息，有许多中类型，比如键盘消息、鼠标消息，以及用户自己定义的消息。在 `EntryPoint` 的事先中，首先打印出用户提示标识符，然后进入一个循环，循环调用 `GetMessage`，从自己的线程队列中获得消息，若调用成功，则判断消息的类型，是否是线程终止消息（一个线程可以给另外一个线程发送终止消息 `KTMSG_THREAD_TERMINAL`，来终止另外线程的运行）。若是，则

跳出循环，从而导致线程运行结束，否则，调用 `DispatchMessage` 函数，来分发消息（处理消息）。

`GetMessage` 和 `SendMessage` 函数的实现，在另外的章节中会有描述，`DispatchMessage` 函数完成消息分发的功能，当前情况下，该函数的实现十分简单，只是把 `KernelThreadMessage` 中的参数分离出来，然后调用消息处理函数 `EventHandler`。其中，`EventHandler` 函数作为一个指针，传递给 `DispatchMessage`，该函数的实现代码如下：

```

BOOL EventHandler(WORD wCommand, WORD wParam, DWORD dwParam)
{
    WORD wr = 0x0700;
    BYTE bt = 0x00;
    BYTE Buffer[12];

    switch(wCommand)
    {
    case MSG_KEY_DOWN:
        bt = LOBYTE(LOWORD(dwParam));
        if(VK_RETURN == bt)
        {
            if(BufferPtr)
                DoCommand();
            PrintPrompt();
            break;
        }
        if(VK_BACKSPACE == bt)
        {
            if(0 != BufferPtr)
            {
                GotoPrev();
                BufferPtr --;
            }
            break;
        }
    else

```

```

    {
        if(MAX_BUFFER_LEN - 1 > BufferPtr)
        {
            CmdBuffer[BufferPtr] = bt;
            BufferPtr++;
            wr += LOBYTE(LOWORD(dwParam));
            PrintCh(wr);
        }
    }
    break;
default:
    break;
}
return 0L;
}

```

当前的实现中，EventHandler 只处理键盘消息。其中，KernelThreadMessage 的 wCommand 变量中，存储了具体的消息类型，目前情况下，定义了两种键盘消息：MSG\_KEY\_DOWN 和 MSG\_KEY\_UP，分别在用户按下键盘和放开按键的时候，由键盘驱动程序发送。其中，EventHandler 只处理键盘按下消息 MSG\_KEY\_DOWN。wCommand 成员标明了消息类型，而 dwParam 变量（KernelThreadMessage 的另一个成员）则包含了对应于特定消息的相关参数，比如，在按键消息中，dwParam 的最低一个字节，存放了键盘被按下的 ASCII 码，这样 EventHandler 就可以从 dwParam 的最低一个字节，确定是哪个键被按下。根据所按下的键的不同，分三种情况进行处理：

- 1、若按下的键是回车键（VK\_RETURN），则 EventHandler 判断当前键盘缓冲队列（CmdBuffer 是一个键盘缓冲队列，BufferPtr 则指明了当前队列中元素的数量）是否为空，若为空（BufferPtr 为 0），则只会换一行，重新打印出提示字符，然后就返回了。若不为空，则说明用户已经在提示符下输入了命令，这时候 EventHandler 会调用 DoCommand 函数，来处理用户输入的命令。在目前的实现中，CmdBuffer 是一个全局变量数组，因此 DoCommand 函数可直接访问该数组，不需要任何参数。DoCommand 函数的实现，在后面进行详细介绍；
- 2、若按下的键是一个退格键（BACKSPACE），则判断当前命令缓冲区（CmdBuffer）是否为空，若为空，则不作任何处理，若不为空，则删除缓冲区的最后一个元素，并把显示器上的光标后移一格，最终的表现就是，用户按了 BACKSPACE 键，删除了输入的一个字符；

- 3、若按下的键不是上述两者之一，则 `EventHandler` 会判断当前命令缓冲区是否满（长度是否达到了 `MAX_BUFFER_LEN`，目前定义为 512）。若已经满了，则不作任何处理，直接返回，否则，会把用户按下的键的 ASCII 字符，存到 `CmdBuffer` 当中，并更新 `BufferPtr`。

与 DOS 命令提示符类似，`EventHandler` 实际上是实现了一个简单的行编辑器，用户可以输入字符，通过 `BACKSPACE` 键删除字符，并通过回车键确认输入的命令，引发操作系统的执行。

上面讲到，若用户按下的键是回车键，且当前命令缓冲区不为空，则 `EventHandler` 会调用 `DoCommand` 函数，处理用户输入的命令。`DoCommand` 函数会对 `CmdBuffer`（全局的命令缓冲区）进行分析，然后根据命令的不同，调用合适的处理函数。该函数的实现代码如下：

```
VOID DoCommand()
{
    DWORD wIndex = 0x0000;
    BOOL bResult = FALSE;           //If find the correct command object,then
                                    //This flag set to TRUE.
    BYTE tmpBuffer[36];

    CmdBuffer[BufferPtr] = 0x00; //Prepare the command string.
    BufferPtr = 0;

    while((' ' != CmdBuffer[wIndex]) && CmdBuffer[wIndex] && (wIndex < 32))
    {
        tmpBuffer[wIndex] = CmdBuffer[wIndex];
        wIndex ++;
    }
    tmpBuffer[wIndex] = 0;

    for(DWORD dwIndex = 0; dwIndex < CMD_OBJ_NUM; dwIndex ++)
    {
        if(StrCmp(&tmpBuffer[0], CmdObj[dwIndex].CmdStr))
```

```

        {
            CmdObj[dwIndex].CmdHandler(&CmdBuffer[wIndex]); //Call the command
handler.

            bResult = TRUE;          //Set the flag.
            break;
        }
    }
    if(!bResult)
    {
        bResult = DoExternalCmd(&CmdBuffer[0]);          //Call the external command
parser.

        if(!bResult)
        {
            DefaultHandler(NULL);                          //Call
default event handler.

        }
    }
    return;
}

```

DoCommand 函数是整个 shell 命令处理的入口点。在目前的实现中，对于系统命令，分成两类：

- 1、内部命令，即操作系统自带的一些功能命令，比如设置时间、设置日期等，这些命令在运行的时候，是直接在 shell 线程的上下文中运行的，不需要额外创建核心线程。这类命令的处理过程往往很短，且不会引起阻塞；
- 2、外部命令，这类命令实际上是一些应用程序，由用户编写（也有几个是操作系统自带的），一般实现一些特定的功能。与内部命令不同的是，所有外部命令，都需要额外创建一个线程，即外部命令在执行的时候，都有自己的上下文空间，且可以被阻塞。

### 3.1.3 内部命令的处理过程

对于内部命令，当前的实现十分简单，只是通过定义一个内部命令数组，把内部命令的命令字符串和处理函数关联起来。如下：

```
#define CMD_OBJ_NUM 18

__CMD_OBJ CmdObj[CMD_OBJ_NUM] = {
    {"version" ,    VerHandler},
    {"memory"   ,    MemHandler},
    {"sysinfo"  ,    SysInfoHandler},
    {"sysname"  ,    SysNameHandler},
    {"help"     ,    HlpHandler},
    {"date"     ,    DateHandler},
    {"time"     ,    TimeHandler},
    {"cpuinfo"  ,    CpuHandler},
    {"support"  ,    SptHandler},
    {"runtime"  ,    RunTimeHandler},
    {"test"     ,    TestHandler},
    {"untest"   ,    UnTestHandler},
    {"memview"  ,    MemViewHandler},
    {"sendmsg"  ,    SendMsgHandler},
    {"ktview"   ,    KtViewHandler},
    {"ioctl"    ,    IoCtrlApp},
    {"sysdiag"  ,    SysDiagApp},
    {"cls"      ,    ClsHandler}
};
```

其中\_\_CMD\_OBJ 是预先定义的一个结构体，如下：

```
struct __CMD_OBJ{
    LPSTR          CmdStr;                //Command text.
    __CMD_HANDLER  CmdHandler;           //Command handler.
};
```

这样，Hello China 在实现内部命令的时候，只需要编写一个处理函数（该函数的原型，应该是\_\_CMD\_HANDLER 定义的，一个只接受字符串作为参数，并返回 VOID 的函数），并在上述数组中增加一项即可。

DoCommand 函数首先处理内部命令，处理过程如下：

- 1、根据用户输入的命令字符串为索引，依次检索上述数组，若能够找到一个合适的，则采用用户输入的命令参数（字符串）为参数，调用该函数；
- 2、若上述查找过程失败，即上述数组中不存在跟用户输入对应的字符串，则会进入外部命令处理阶段。

需要说明的是，用户的输入，被操作系统按照“命令字符串 参数 1 参数 2 .....”的形式解释的，即用户按下回车后，操作系统会把整个字符串传递给 DoCommand，DoCommand 则提取字符串前面的命令部分（命令字符串），然后把后面的参数部分作为参数，传递给命令处理函数。按照目前的实现，命令字符串跟参数之间是通过空格的方式分割的，但命令字符串也有一个最大长度，即 32 字节，比如，假设用户输入下列字符串后按下回车：

```
[system-view]thisismycommandthisismycommandthis hello china <ENTER>
```

则命令字符串会被截取为“thisismycommandthisismycommandth”，因为命令字符串超过了 32 个比特。另外，在调用该命令的处理函数时，传递的参数将是“hello china”。

下面我们以实现一个内部命令为例，说明内部命令的实现方式。

- 1、假设实现的内部命令为 mycommand，首先编写一个内部命令的功能函数，可以直接在 SHELL.CPP 文件中添加，也可以通过另外的模块实现，然后在 SHELL.CPP 中，包含实现的命令函数的头文件。假设 mycommand 命令的实现函数如下：

```
VOID mycommand(LPSTR)
{
    ChangeLine();
    PrintLine("Hello,World!");
    ChangeLine();
}
```

该函数的功能十分简单，打印出“Hello,World!”字符串，这也是大多数编程语音的一个入门示例；

- 2、将该命令字符串和命令函数，添加到内部命令列表中，并更改 CMD\_OBJ\_NUM 宏为原来的值加一，因为新增加了一个内部命令。如下：

```

#define CMD_OBJ_NUM 18
#define CMD_OBJ_NUM 19
__CMD_OBJ CmdObj[CMD_OBJ_NUM] = {
    {"version" ,    VerHandler},
    {"memory" ,    MemHandler},
    {"sysinfo" ,    SysInfoHandler},
    {"sysname" ,    SysNameHandler},
    {"help" ,       HlpHandler},
    {"date" ,       DateHandler},
    {"time" ,       TimeHandler},
    {"cpuinfo" ,    CpuHandler},
    {"support" ,    SptHandler},
    {"runtime" ,    RunTimeHandler},
    {"test" ,       TestHandler},
    {"untest" ,     UnTestHandler},
    {"memview" ,    MemViewHandler},
    {"sendmsg" ,    SendMsgHandler},
    {"ktview" ,     KtViewHandler},
    {"ioctl" ,      IoCtrlApp},
    {"sysdiag" ,    SysDiagApp},
    {"mycommand" ,  mycommand},
    {"cls" ,        ClsHandler}
};

```

3、重新编译连接（rebuild）整个操作系统核心，并重新制作引导盘，引导系统。成功启动后，在命令行提示符下，输入 `mycommand` 并回车，就可以看到 `mycommand` 的输出了。

### 3.1.4 外部命令的处理过程

之所以增加外部命令和内部命令之分，一个原因是为了编程上的方便。在当前的实现中，所有内部命令，都是在 `SHELL` 所在的源文件内实现的，而外部命令，则统一以

EXTCMD.CPP 为接口, 实现的时候, 用户可用通过另外的模块实现功能部分, 然后只需要在 EXTCMD.CPP 文件当中, 所一个简单的修改即可。另外的一个因素, 是所有内部命令都是以函数的形式实现的, 即内部命令的运行上下文, 跟 SHELL 共享, 但外部命令的实现, 则是通过创建单独的核心线程实现的, 有自己的线程上下文。

为了实现外部命令, 特定义如下的数据结构:

```
__BEGIN_DEFINE_OBJECT(__EXTERNAL_COMMAND)
    LPSTR          lpszCmdName;
    LPSTR          lpszHelpInfo;
    BOOL           bBackground;
    DWORD          (*ExtCmdHandler)(LPVOID);
__END_DEFINE_OBJECT()
```

其中, lpszCmdName 是外部命令的命令字符串, lpszHelpInfo 是外部命令的帮助信息, 通过执行 help 命令, 可把外部命令的相关帮助信息打印出来。bBackground 变量是一个指示变量, 告诉操作系统, 该外部命令是在后台执行, 还是在前台执行。若是在后台执行, 则操作系统创建外部命令的执行线程, 然后就返回用户界面 (SHELL 线程), 这时候, 用户界面仍然正常响应用户需求。若是在前台执行, 则操作系统会创建外部命令的执行线程后, 将一直等待外部命令执行结束, 然后才返回 SHELL, 这个过程, 用户只能与外部命令提供的用户接口进行交互。ExtCmdHandler 则是具体的外部命令入口点, 可以看出, 该函数的原型, 是跟线程入口点的函数原型匹配的。

定义如下数组, 来管理所有的外部命令:

```
__EXTERNAL_COMMAND ExtCmdArray[] = {
    {"extcmd1", "The first external command.", TRUE, ExtCmd1},
    {NULL, NULL, FALSE, NULL}
};
```

在实现外部命令的时候, 只需要在上述数组中, 加入对应的内容, 就可以被系统识别, 这时候, 在命令提示符下, 只需输入外部命令字符串, 外部命令就可以得到执行。另外, 通过执行 help 命令, 可以打印出外部命令的帮助信息。

对于外部命令的处理, 是由 DoExternalCmd 函数完成的, 该函数被 DoCommand 调用 (参见上面内部命令实现的描述), 代码如下:

```

BOOL DoExternalCmd(LPSTR lpszCmd)
{
    DWORD wIndex      = 0x0000;
        DWORD i        = 0;
        LPSTR lpszParam  = NULL;
        __KERNEL_THREAD_OBJECT* lpExtCmd = NULL;
    BOOL bResult      = FALSE;           //If find the correct command object,then
                                           //This flag set to TRUE.

    BYTE tmpBuffer[36];

    while((' ' != lpszCmd[wIndex]) && lpszCmd[wIndex] && (wIndex < 32))
    {
        tmpBuffer[wIndex] = lpszCmd[wIndex];
        wIndex ++;
    }
    tmpBuffer[wIndex] = 0;

    while(ExtCmdArray[i].lpszCmdName)
    {
        if(StrCmp(&tmpBuffer[0],ExtCmdArray[i].lpszCmdName)) //Find the correct
external command entry.
        {
            //
            //Handle external command here.
            //
            if(tmpBuffer[wIndex + 1]) //Have parameters.
            {
                lpszParam
                =
                (LPSTR)KMemAlloc(StrLen(&tmpBuffer[wIndex + 1] + 1),KMEM_SIZE_TYPE_ANY);
                if(NULL == lpszParam) //Can not allocate memory.
                {
                    bResult = FALSE;
                    break;
                }
            }
        }
    }
}

```

```

        StrCpy(lpszParam,&tmpBuffer[wIndex + 1]);
    }
    lpExtCmd = KernelThreadManager.CreateKernelThread(

(__COMMON_OBJECT*)&KernelThreadManager,
        OL,
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_NORMAL,
        ExtCmdArray[i].ExtCmdHandler,
        lpszParam,
        NULL);
    if(NULL == lpExtCmd) //Can not create thread.
    {
        bResult = FALSE;
        break;
    }
    if(!ExtCmdArray[i].bBackground)        //Should run in
foreground.

    {
        DeviceInputManager.SetFocusThread(

(__COMMON_OBJECT*)&DeviceInputManager,

(__COMMON_OBJECT*)lpExtCmd); //Set focus.

lpExtCmd->WaitForThisObject((__COMMON_OBJECT*)lpExtCmd); //Blocking.
    }
    bResult = TRUE;        //Set the flag.
    break;
}
}
return bResult;
}

```

与内部命令处理类似，首先从命令行中，分离出具体的外部命令，然后根据外部命令字符串，匹配外部命令列表，若找不到匹配的项，则返回 `FALSE`，从而导致 `DefaultHandler` 被调用（参考 `DoCommand`）。若匹配成功，则进一步判断，用户是否输入了作用于该外部命令的参数。若有额外的参数，则该函数申请一块内存，并把额外的参数拷贝到申请的内存中。然后调用 `CreateKernelThread` 函数，创建一个线程，线程的入口点，就是用户提供的外部命令处理函数。需要注意的是，在存在额外参数的情况下，由于 `DoExternalCmd` 函数申请了内存，然后把该内存传递到了新创建的函数，因此这块内存的释放，应该是外部命令处理函数的工作。若外部命令处理函数不释放参数占用的内存，则可能会引起内存泄漏。

在完成外部命令执行线程的创建后，会进一步判断，该外部命令是否需要在后台执行。若是后台执行程序，则 `DoExternalCmd` 函数直接返回，若是非后台执行程序，则 `DoExternalCmd` 函数需要等待外部命令线程的执行，进入阻塞状态。这种情况下，还把当前的输入焦点设置为外部命令执行线程，以便外部命令执行线程能够接受用户输入。

## 第四章 Hello China 的线程

### 4.1 线程概述

线程是 Hello China 操作系统的任务模型，在本章中，我们对这个任务模型进行详细描述，并对其实现机制和代码进行分析。在此之前，有必要对操作系统中关于线程、进程和任务的一些基本概念进行描述，以便读者更好地理解本章内容。

#### 4.1.1 进程、线程和任务

进程是操作系统演进中的革命性概念，所谓进程，比较通俗的一个说法就是“一个运行起来的程序”，这就是说，一个进程，首先是一个程序，即一段代码，而且是一段已经运行起来的代码。比如，位于磁盘上的应用程序不是进程，因为它还没有运行起来，一旦该程序运行起来（比如，在命令行界面中输入该程序的名字及相关参数，然后敲回车键），就可以称为进程了。总之，一个进程有下列特性。

（1）首先是一个程序，即是一段可执行的代码（这段代码可能位于内存中，也可能位于存储介质上）；

（2）该程序正在运行，即已经被操作系统加载到内存中（或者本来就位于内存中），并创建了相应的控制结构（比如，进程控制块、地址空间等）。

一般情况下，一个进程有自己独立的地址空间，比如，在 32 位硬件平台上，进程的地址空间是 4GB。可执行代码可以访问这个地址空间内的任何数据（不考虑操作系统的保护机制），但不能访问其他进程的数据，因为不同的进程其地址空间是不重叠的。

线程则是归属于进程的可调度单位。一般情况下，一个进程由多个线程组成，线程是操作系统能够知晓的最小的调度单位，而且一般情况下，操作系统都是按照线程来调度的。属于同一个进程的多个线程共享进程的地址空间，这样线程之间就可以很容易地通过这个公共的地址空间进行通信。每个线程都有自己的堆栈和上下文，用于保存运行过程数据。



而任务是嵌入式操作系统中的一个概念，其本质就是一个线程，但特别的是，任务是一个一直运行的循环，一旦启动，就一直运行，不会中途退出（除非发生异常被操作系统强行中止，或者被人工强行中止）。由于任务本质上是一个线程，具备线程所有的特点，而线程的内涵更丰富一些，因此，在 **Hello China** 的实现中，只引用了线程的概念，没有引入任务概念。实际上，把一个线程的功能函数编码成一个死循环，该线程就成了一个任务。比如，下列函数就可以作为一个任务运行。

```
DWORD TaskRoutine(LPVOID lpData)
{
    .....
    while(TRUE)
    {
        GetMessage(...); //Get message from message queue.
        ProcessMsg(...); //Process message.
    };
    return 0L;
}
```

该函数一旦被投入运行，就以循环的方式检查自己的消息队列，若有消息，则做进一步处理，然后又进入新一轮循环。

## 4.2 Hello China V1.0 版本的线程实现

与大多数嵌入式操作系统一样，**Hello China** 实现了多任务、多线程的构架。但在 **Hello China** 的实现中，只引用了线程的概念，没有引用任务的概念，因为从本质上讲，任务就是一个线程，所不同的是，任务是一个无限循环，因此，实现线程比实现任务具有更广泛的适应性。

### 4.2.1 核心线程管理对象

在 **Hello China** 的实现中，一个全局对象 **KernelThreadManager**（核心线程管理对象）用来完成对整个操作系统线程的管理，包括线程的组织、创建、销毁、修改优先级等操作，该对象的定义如下。

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_MANAGER)
    //DWORD                                dwCurrentIRQL;
    __KERNEL_THREAD_OBJECT*                lpCurrentKernelThread; //Current
kernel thread.

    __PRIORITY_QUEUE*                      lpRunningQueue;
```

```

__PRIORITY_QUEUE*          lpReadyQueue;
__PRIORITY_QUEUE*          lpSuspendedQueue;
__PRIORITY_QUEUE*          lpSleepingQueue;
__PRIORITY_QUEUE*          lpTerminalQueue;

DWORD                      dwNextWakeupTick;

BOOL                      (*Initialize)(__COMMON_OBJECT*
lpThis);

__KERNEL_THREAD_OBJECT*    (*CreateKernelThread)(
                        __COMMON_OBJECT*          lpThis,
                        DWORD                      dwStackSize,
                        DWORD                      dwStatus,
                        DWORD                      dwPriority,
                        __KERNEL_THREAD_ROUTINE    lpStartRoutine,
                        LPVOID                    lpRoutineParam,
                        LPVOID                    lpReserved);

VOID                      (*DestroyKernelThread)(__COMMON_OBJECT* lpThis,
                        __COMMON_OBJECT*          lpKernelThread
                        );

BOOL                      (*SuspendKernelThread)(
                        __COMMON_OBJECT*          lpThis,
                        __COMMON_OBJECT*          lpKernelThread
                        );

BOOL                      (*ResumeKernelThread)(
                        __COMMON_OBJECT*          lpThis,
                        __COMMON_OBJECT*          lpKernelThread
                        );

VOID                      (*ScheduleFromProc)(
                        __KERNEL_THREAD_CONTEXT*  lpContext
                        );

VOID                      (*ScheduleFromInt)(
                        __COMMON_OBJECT*          lpThis,
                        LPVOID                    lpESP
                        );

DWORD                    (*SetThreadPriority)(
                        __COMMON_OBJECT*          lpKernelThread,
                        DWORD                      dwNewPriority
                        );

```

```

DWORD          (*GetThreadPriority)(
    __COMMON_OBJECT*          lpKernelThread
    );

DWORD          (*TerminalKernelThread)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
    );

BOOL           (*Sleep)(
    __COMMON_OBJECT*          lpThis,
    //__COMMON_OBJECT*        lpKernelThread,
    DWORD                    dwMilliSecond
    );

BOOL           (*CancelSleep)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
    );

DWORD          (*GetLastError)(
    __COMMON_OBJECT*          lpKernelThread
    );

DWORD          (*SetLastError)(
    __COMMON_OBJECT*          lpKernelThread,
    DWORD                    dwNewError
    );

DWORD          *GetThreadID)(
    __COMMON_OBJECT*          lpKernelThread
    );

DWORD          (*GetThreadStatus)(
    __COMMON_OBJECT*          lpKernelThread
    );

DWORD          (*SetThreadStatus)(
    __COMMON_OBJECT*          lpKernelThread,
    DWORD                    dwStatus
    );

BOOL           (*SendMessage)(
    __COMMON_OBJECT*          lpKernelThread,
    __KERNEL_THREAD_MESSAGE*   lpMsg
    );

BOOL           (*GetMessage)(
    __COMMON_OBJECT*          lpKernelThread,
    __KERNEL_THREAD_MESSAGE*   lpMsg
    );

```

```

BOOL                                     (*LockKernelThread)(
    __COMMON_OBJECT*                    lpThis,
    __COMMON_OBJECT*                    lpKernelThread);

VOID                                     (*UnlockKernelThread)(
    __COMMON_OBJECT*                    lpThis,
    __COMMON_OBJECT*                    lpKernelThread);

END_DEFINE_OBJECT() //End of the kernel thread manager's definition.

```

这是一个比较大的对象，其中，五个优先级队列用于对系统中的线程进行管理，每个队列的用途如表 4-1 所示。

表 4-1 Hello China 的线程队列

队 列	用 途	对应的线程状态
lpRunningQueue	所有当前运行的线程对象，被存储在该队列（注 1）	运行
lpSuspendedQueue	所有被手工挂起的线程，存储在该队列	挂起
lpSleepingQueue	所有处于睡眠状态的线程，存储在该队列	睡眠
lpReadyQueue	所有准备就绪的线程，存储在该队列	就绪
lpTerminalQueue	所有运行结束，尚未被释放的线程对象	终止

注 1：对于单 CPU 的情况，有且只有一个线程处于运行状态（即任何时刻，只有一个线程获得 CPU 资源，处于运行状态），这种情况下，该队列未被使用。但在多处理器情况下，任何一个时刻，有跟系统中 CPU 数量相同的线程在运行，这样为了便于管理，设置此队列，用于管理多 CPU 情况下的运行态线程。

需要注意的是，还有一种线程状态——阻塞状态没有体现在上述队列中。因为线程的阻塞状态一般是因为该线程要请求一个共享资源，而该共享资源又不可用（被其他线程占有），这时候线程进入阻塞状态。进入阻塞状态的线程会被暂时存放在共享资源的阻塞队列中，因此没有必要专门设置一个全局队列来管理阻塞状态的线程。详细信息在介绍同步对象的时候会提到。

该对象还提供了大量的接口，用于完成对线程的操作。表 4-2 给出了操作动作和对应的操作函数。

表 4-2 核心线程管理对象提供的接口

操作函数	操作动作
CreateKernelThread	创建一个核心线程
DestroyKernelThread	销毁一个核心线程
SuspendKernelThread	手工挂起一个核心线程
ResumeKernelThread	恢复一个核心线程对象
GetThreadPriority	获得线程优先级
SetThreadPriority	设置线程优先级

续表

操作函数	操作动作
Sleep	使当前线程进入睡眠状态
CancelSleep	唤醒一个处于睡眠状态的线程
GetLasterError	获得当前线程的最后错误状态
SetLasterError	设置当前线程的最后错误状态
GetThreadID	获得一个线程的线程 ID
GetThreadStatus	获得一个线程的线程状态
SetThreadStatus	设置一个线程的线程状态（不建议直接调用）
SendMessage	向一个特定线程发送一条消息
GetMessage	从线程消息队列中获取消息
LockKernelThread	锁住当前线程，避免被调度（被锁住的线程不会被其它线程中断）
UnlockKernelThread	解开锁住的线程

上述函数可以被应用程序直接调用，来完成对 Hello China 线程的操作。另外的几个函数，比如 ScheduleFromProc、ScheduleFromInt 等，用于操作系统核心完成线程切换的功能函数。这些函数被操作系统核心代码调用，一般不建议应用程序直接调用这些函数，但为了简便起见，把这些函数也纳入 KernelThreadManager 的管理范围。

另外，dwNextWakeupTick 变量用于管理线程的睡眠功能。该变量记录了需要最早唤醒的线程和应该唤醒的时刻（tick 数目）。比如，当前的时钟 tick 是 1000，有三个线程，分别调用了 Sleep 函数，代码如下。

```
Thread1:
... ..
Sleep(500);
... ..

Thread2:
... ..
Sleep(300);
... ..

Thread3:
... ..
Sleep(600);
... ..
```

并假设这三个线程调用 Sleep 函数的次序发生在同一个时间片之间，这样需要最早唤醒的线程应该是 Thread2。于是，Sleep 函数在执行的时候，把以毫秒（ms）为单位的参数，转换为时钟 tick 数，然后跟当前的系统 tick 数（System 对象维护，详细信息请参考

本书“中断和定时处理机制的实现”部分）相加，并赋给 `dwNextWakeupTick` 变量。

这样每次时钟中断处理的时候，操作系统把 `dwNextWakeupTick` 跟当前的时钟 `tick` 数进行比较，若一致，则说明已经到达唤醒线程的时刻，于是从睡眠队列（`lpSleepingQueue`）中取出所有需要唤醒的线程，插入就绪队列（`lpReadyQueue`）。

另外需要注意的是，`KernelThreadManager` 是一个全局对象，整个系统中只有一个这样的对象，因此，该对象没有从 `_COMMON_OBJECT` 对象派生。对于该对象提供的功能函数（比如 `CreateKernelThread` 等），为了保持面向对象的语义，其参数列表也与其它对象一样，第一个参数是 `lpThis`（一个指向自己的指针），实际上，可以不用这个参数，而直接引用 `KernelThreadManager` 对象。

### 4.2.2 线程的状态及其切换

Hello China 的线程可以处于以下几种状态。

**1. Ready:** 所有线程运行的条件就绪，线程进入 `Ready` 队列，如果 `Ready` 队列中没有比该线程优先级更高的线程，那么下一次调度程序运行时（时钟中断或系统调用），该线程将会被选择投入运行；

**2. Suspended:** 线程被挂起，这是线程执行 `SuspendKernelThread` 的结果，或者该线程创建时就指定初始状态为 `Suspended`，处于这种状态的线程，只有另外的线程调用 `ResumeThread` 时才能把该线程的状态改变为 `Ready`；

**3. Running:** 线程获取了 CPU 资源正在运行。在单 CPU 环境下，任何时刻只有一个线程的状态是 `Running`，但在多 CPU 环境下，假设有 `N` 个 CPU，那么任何时刻，最多的时候，可能有 `N` 个线程的状态是 `Running`；

**4. Sleeping:** 线程处于睡眠状态，一般情况下，处于运行状态（`Running`）的线程调用 `Sleep` 函数，则该线程进入睡眠队列，当定时器（由 `Sleep` 函数指定）到时后，处于该状态的线程被系统从 `Sleeping` 队列中删除，并插入 `Ready` 队列，相应地，其状态修改为 `Ready`；

**5. Blocked:** 线程不具备运行的条件，比如，线程正在等待某个共享资源，那么该线程就会进入该共享资源的队列中，其状态也会被修改为 `Blocked`。当共享资源被其它资源释放的时候，会重新修改当前等待该共享资源的线程（`Blocked` 线程），将其状态修改为 `Ready`，并放入 `Ready` 队列；

**6. Terminal:** 线程运行结束，但用于对该线程进行控制的线程对象（`Kernel Thread Object`）还没有被删除，处于这种状态的线程对象被放入 `Terminal` 队列，直到另外的线程明确地调用 `DestroyKernelThread` 删除该线程对象为止。

其中，每种状态的线程对象被组织在一个队列中（在单 CPU 环境中，状态是 `Running` 的线程不进入任何队列，在多 CPU 环境中，状态是 `Running` 的线程，进入 `lpRunningQueue`

队列)，每个队列都是一个优先队列对象，因此，位于其中的线程对象可以按照优先级进行排序，对于状态是 **Blocked** 的线程，被组织在共享资源（或同步对象）的本地等待队列中，调度程序只选择 **Ready** 队列中的线程投入运行，整个系统的线程对象队列模型参考图 4-1。

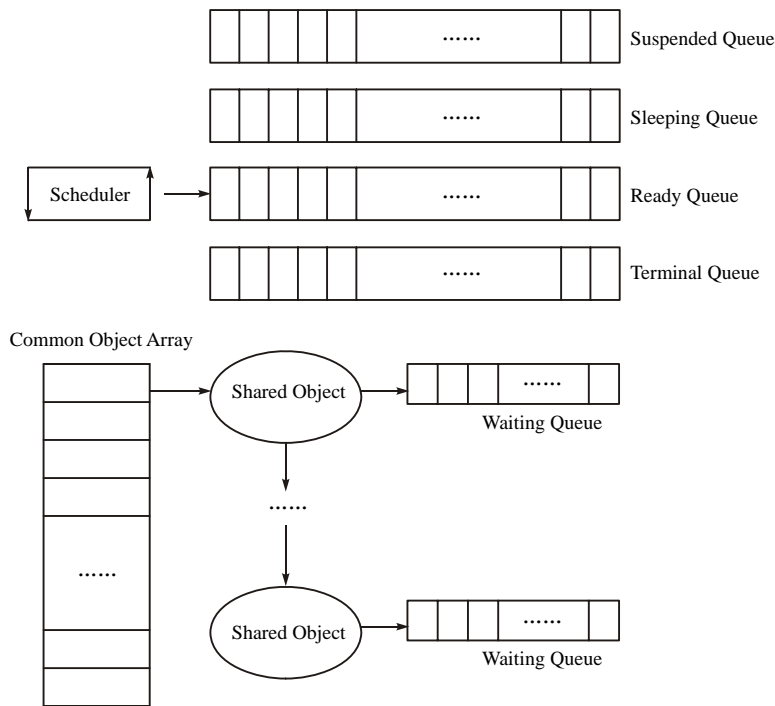


图 4-1 Hello China 的线程队列

从图中可以看出，系统总共维护四个队列（**lpSleepingQueue**、**lpSuspendedQueue**、**lpReadyQueue** 和 **lpTerminalQueue**），调度程序（**Scheduler**）只从 **lpReadyQueue** 中选择线程投入运行。对于共享对象（**SharedObject**），在 **Hello China** 的实现中，被组织成了一条链表（**ObjectManager** 维护），每个共享对象维持一个等待队列（**Waiting Queue**），凡是进入这些队列的线程对象，其状态必然是 **Blocked**。

图 4-2 给出了线程各个状态之间的转换图示。其中，箭头表示转换方向，单向的箭头代表状态转换是单向的，比如 **Running** 到 **Suspended** 的单向箭头，代表一个处于 **Running** 状态的线程，可以切换到 **Suspended** 状态，反之则不行。

从图 4-2 可以看出，**Running** 状态只能从 **Ready** 状态转换过来，**Blocked** 状态和 **Suspended** 状态也只能从 **Running** 状态转换过来。

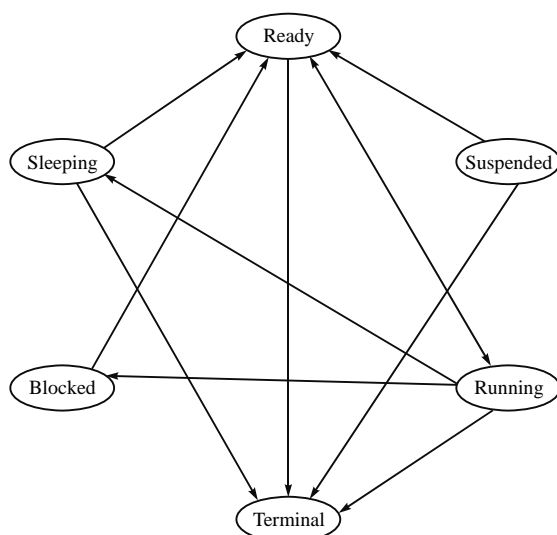


图 4-2 核心线程的状态及其之间的转换

表 4-3 列出了各状态线程之间的切换条件。

表 4-3 线程转换发生的条件

初始状态 目标状态	Ready	Suspended	Running	Blocked	Sleeping	Terminal
Ready	——	ResumeKernelThread	时间片用完	获得共享资源	定时器到时	——
Suspended	——	——	SuspendKernelThread 自己调用	——	——	——
Running	获得 CPU 资源	——	——	——	——	——
Blocked	——	——	请求共享资源	——	——	——
Sleeping	——	——	调用 Sleep 函数	——	——	——
Terminal	TerminalKernelThread 其它线程调用	TerminalKernelThread 其它线程调用	运行完	——	TerminalKernelThread 其它线程调用	——

表 4-3 中，横的一栏为初始状态，对应的表格为转换原因，竖的一栏为目标状态。

### 4.2.3 核心线程对象

核心线程对象（KernelThreadObject）用于管理 Hello China 操作系统中的所有线程，该对象记录了每个线程的上下文信息、堆栈指针（初始指针）、最后错误信息、线程的消

息队列等。注意核心线程对象与核心线程管理对象的区别。下面是核心线程对象的定义。

```

BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    INHERIT_FROM_COMMON_SYNCHRONIZATION_OBJECT
    __KERNEL_THREAD_CONTEXT          KernelThreadContext;
    DWORD                            dwThreadID;
    DWORD                            dwThreadStatus;
    __PRIORITY_QUEUE*                lpWaitingQueue;
    DWORD                            dwThreadPriority;
    DWORD                            dwScheduleCounter;
    DWORD                            dwReturnValue;

    DWORD                            dwTotalRunTime;
    DWORD                            dwTotalMemSize;

    __KERNEL_FILE*                    lpCurrentDirectory;
    __KERNEL_FILE*                    lpRootDirectory;
    __KERNEL_FILE*                    lpModuleDirectory;

    __COMMON_OBJECT*                 StartTime;

    BOOL                             bUsedMath;

    DWORD                            dwStackSize;
    LPVOID                           lpInitStackPointer;

    DWORD                            (*KernelThreadRoutine)(LPVOID);
    LPVOID                           lpRoutineParam;

    //The following four members are used to manage the message queue
of the
    //current kernel thread.
    __KERNEL_THREAD_MESSAGE           KernelThreadMsg[MAX_KTHREAD_
MSG_NUM];
    UCHAR                            ucMsgQueueHeader;
    UCHAR                            ucMsgQueueTrial;
    UCHAR                            ucCurrentMsgNum;
    UCHAR                            ucAlignent;
    __EVENT*                          lpMsgEvent;

    DWORD                            dwLastError;

```

```

__KERNEL_THREAD_OBJECT_TABLE      KernelObjectTable;
//Object's table.
__KERNEL_THREAD_OBJECT*           lpParentKernelThread;
END_DEFINE_OBJECT( )

```

为了版面上的清晰，省略了源代码中的相关注释。该对象用于管理每个核心线程，因此该对象的成员变量包含了核心线程相关的方方面面。表 4-4 中，按照变量的用途进行归类，并对每个变量的含义进行了解释。

表 4-4 核心线程对象各成员的含义

类 别	类别含义	对应的变量	变量含义
硬件上下文	保存 CPU 相关的寄存器	KernelThreadContext	一个结构类型，跟特定 CPU 相关
线程属性	线程的标识、优先级等属性信息	dwLastError	最后错误信息
		dwThreadID	线程 ID
		dwThreadStatus	线程的状态
		dwRetVal	线程的返回值
堆栈信息	记录或控制线程的堆栈	dwStackSize	堆栈大小
		lpInitStackPointer	初始堆栈指针
消息队列信息	完成线程的消息队列控制功能	KernelThreadMsg	消息数组
		ucMsgQueueHeader	消息头索引
		ucMsgQueueTail	消息尾索引
		ucCurrentMsgNum	当前消息队列中的消息数
		lpMsgEvent	同步消息队列的事件对象
同步信息	核心线程对象本身是一个同步对象	lpWaitingQueue	等待队列
调度信息	线程调度的依据	dwThreadPriority	线程的优先级
		dwScheduleCounter	调度计数
资源占用信息	描述线程对内存、CPU 等资源的占用情况，以及创建的核心对象情况	dwTotalRunTime	总共运行时间
		dwTotalMemSize	内存占用数量（物理内存）
		StartTime	开始运行时间
		bUsedMath	是否使用数学协处理器
		KernelObjectTable	记录创建的核心对象
线程函数	线程功能函数相关信息	KernelThreadRoutine	功能函数指针
		lpRoutineParam	功能函数参数

下面对线程对象的上述类别信息进行粗略的描述，详细的描述，请参考本章后续章节。

**1. 件上下文：**这是一个\_\_KERNEL\_THREAD\_CONTEXT 类型的结构变量，用于记录线程的硬件上下文信息。所谓的硬件上下文，就是线程所运行的 CPU 的硬件寄存器，

比如指令指针寄存器、堆栈寄存器等，这些寄存器信息在线程切换的时候需要保存或恢复。在线程从运行状态切换到其它状态（比如，就绪状态）的时候，调度程序就会把当前线程所运行的 CPU 的寄存器信息保存到这个数据结构中，在线程被再次调度运行的时候，调度程序从这个数据结构中，恢复对应的寄存器信息。需要注意的是，这个结构的定义是 CPU 特定的，即不同的 CPU，该结构的定义也不一样。详细情况请参考“线程的上下文”一节；

**2. 线程属性信息：**包含了线程 ID、线程的当前状态、线程的返回值（线程函数的返回值）以及线程的最后错误信息等内容；

**3. 堆栈信息：**堆栈是线程运行过程中，保存临时数据和临时变量的地方，在核心线程对象中，记录了线程堆栈的大小（dwStackSize）和线程堆栈的初始地址（lpInitStackPointer）。需要注意的是，lpInitStackPointer 不是线程的堆栈指针，线程的堆栈指针在线程的运行过程中不断变化，详细信息请参考“线程的堆栈”一节；

**4. 消息队列信息：**每个线程都有一个本地消息队列，用于存储别的线程（或者自己）发过来的消息，在 Hello China 的当前实现中，消息队列是一个环形队列，ucMsgQueueHeader 和 ucMsgQueueTail 两个变量记录了唤醒队列的头和尾，ucCurrentMsgNum 则记录了当前队列中的消息数目。线程采用 GetMessage 函数从线程队列中获取信息，别的线程采用 SendMessage 函数给一个特定的线程发送信息；

**5. 同步信息：**与其它同步对象（比如 Event、Mutex 等）一样，核心线程对象也是一个同步对象，所不同的是，核心线程对象的状态不能人为的通过 API 函数控制，而只能根据线程的运行状态来自行控制。一个线程对象只有其状态成为 Terminal（KERNEL\_THREAD\_STATUS\_TERMINAL）的时候，才是发信号状态（可用状态），所有其它状态都为不可用状态。比如，另外一个线程（假设为 A）调用 WaitForThisObject 函数，等待一个线程对象（假设为 B），则该线程 A 将一直处于阻塞状态（被放入线程 B 的 lpWaitingQueue），直到线程 B 运行完毕，状态变化为 Terminal 的时候，线程 A 才会被唤醒；

**6. 调度信息：**包含两个变量 dwScheduleCounter 和 dwThreadPriority，这两个变量是线程调度策略的依据；

**7. 资源占用信息：**描述了线程的资源占用情况，比如创建的核心对象、占用的物理内存大小、占用的 CPU 时间（运行时间）、是否使用了数学协处理器等；

**8. 线程函数信息：**线程的功能函数和其参数，线程函数是实现线程功能的主体，由应用程序编写。线程函数的参数是一个无类型指针（LPVOID），可以通过该指针传递任何参数信息。

## 4.2.4 线程的上下文

线程的上下文（Context）是一个类型为 \_\_KERNEL\_THREAD\_CONTEXT 的结构体，

该结构体与特定的硬件平台（CPU）有强关联关系，不同的硬件平台，该结构体的定义不同。在本文中，我们重点关注 Intel IA32 构架的 CPU。在这种硬件平台下，该结构体的定义如下。

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_CONTEXT)
#ifdef __I386
    DWORD        dwEFlags;
    WORD          wCS;
    WORD          wReserved;
    DWORD         dwEIP;
    DWORD         dwEAX;
    DWORD         dwEBX;
    DWORD         dwECX;
    DWORD         dwEDX;
    DWORD         dwESI;
    DWORD         dwEDI;
    DWORD         dwEBP;
    DWORD         dwESP;

//
//The following macro is used to initialize a kernel thread's execution
//context.
//

#define INIT_EFLAGS_VALUE 512
#define INIT_KERNEL_THREAD_CONTEXT_I(lpContext,initEip,initEsp) \
    (lpContext)->dwEFlags      = INIT_EFLAGS_VALUE;           \
    (lpContext)->wCS           = 0x08;                         \
    (lpContext)->wReserved     = 0x00;                         \
    (lpContext)->dwEIP         = initEip;                      \
    (lpContext)->dwEAX         = 0x00000000;                   \
    (lpContext)->dwEBX         = 0x00000000;                   \
    (lpContext)->dwECX         = 0x00000000;                   \
    (lpContext)->dwEDX         = 0x00000000;                   \
    (lpContext)->dwESI         = 0x00000000;                   \
    (lpContext)->dwEDI         = 0x00000000;                   \
    (lpContext)->dwEBP         = 0x00000000;                   \
    (lpContext)->dwESP         = initEsp;

//
//In order to access the members of a context giving it's base address,
```

```
//we define the following macros to make this job easy.
//

#define CONTEXT_OFFSET_EFLAGS      0x00
#define CONTEXT_OFFSET_CS          0x04
#define CONTEXT_OFFSET_EIP        0x08
#define CONTEXT_OFFSET_EAX        0x0C
#define CONTEXT_OFFSET_EBX        0x10
#define CONTEXT_OFFSET_ECX        0x14
#define CONTEXT_OFFSET_EDX        0x18
#define CONTEXT_OFFSET_ESI        0x1C
#define CONTEXT_OFFSET_EDI        0x20
#define CONTEXT_OFFSET_EBP        0x24
#define CONTEXT_OFFSET_ESP        0x28
#else
#endif
END_DEFINE_OBJECT()
```

该结构体的定义可以分为三部分。

- 1. 数据成员部分，与 IA32 硬件平台的相关寄存器相应；
  - 2. 初始化宏 (INIT\_KERNEL\_THREAD\_CONTEXT\_I)，这个宏用于初始化一个 IA32 构架下的上下文结构，其中第一个参数 (lpContext) 给出了要初始化的上下文结构的指针，initEip 和 initEsp 是赋予 dwEIP 和 dwESP（分别与 EIP 和 ESP 寄存器对应）的初始化值，上下文结构中 dwEflags 初始化为 512，wCS 初始化为 8，所有其它变量初始化为 0。这样初始化的含义请参考本文后续部分；
  - 3. 成员访问偏移，为了访问一个上下文结构的成员变量（比如，dwESP 等），采用上下文结构的指针（基地址）再加上特定的宏，这应用于嵌入式汇编语言的编程中。
- 表 4-5 给出了对应的 IA32 硬件平台的硬件寄存器与上述结构中成员的对应关系。

表 4-5 核心线程各寄存器的初始化值

寄 存 器	对应的变量	初始化值
EFlags	dwEflags	512
CS	wCS	8
EIP	dwEIP	线程特定
EAX	dwEAX	0
EBX	dwEBX	0
ECX	dwECX	0
EDX	dwEDX	0

续表

寄存器	对应的变量	初始化值
ESI	DwESI	0
EDI	dwEDI	0
EBP	dwEBP	0
ESP	dwESP	线程特定

按照 IA32 的体系结构，EFLAGS 寄存器的内容如图 4-3 所示。

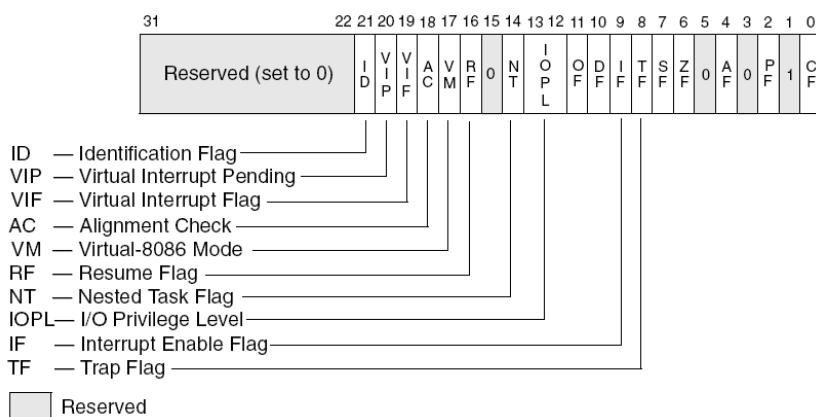


图 4-3 EFLAGS 寄存器各比特的含义

每个核心线程初始化的时候，我们把 EFLAGS（dwEflags）寄存器的值设置为 512，也就是在上述比特中，只把 IF（中断允许标记）标志设置为 1，这样允许中断，所有其它标志。均设置为 0。

目前 Hello China 的实现没有引入进程的概念，整个操作系统只有一个地址空间，所有核心线程共享这个地址空间，因此，所有线程对应的 CS 寄存器值相同，都为 8（代码段的索引值）。针对每个线程，操作系统都创建一个堆栈，该堆栈实际上是一块物理内存，在初始化的时候，lpESP 指向了该物理内存的末端（不是首地址，因为堆栈是按照从上往下的方向增长的，但也不是严格的末地址，而是在末地址的基础上，再减去 8，详细信息，请参考“线程的创建”部分）。

而对于 EIP 寄存器的值，设置为线程起始函数的地址。需要注意的是，这个起始函数并不是线程工作函数，线程工作函数被线程起始函数调用，来完成具体的工作，而在调用线程工作函数之前，线程起始函数还需要做一些其它的工作，比如初始化核心线程对象等。下面是 Hello China 实现的线程起始函数的部分代码。

```
static VOID KernelThreadWrapper(__COMMON_OBJECT* lpKThread)
{
```

```

    __KERNEL_THREAD_OBJECT*    lpKernelThread        = NULL;
    __KERNEL_THREAD_OBJECT*    lpWaitingThread       = NULL;
    __PRIORITY_QUEUE*          lpWaitingQueue        = NULL;
    __PRIORITY_QUEUE*          lpReadyQueue          = NULL;
    DWORD                       dwRetValue           = 0L;
    DWORD                       dwFlags              = 0L;

    if(NULL == lpKThread)      //Parameter check.
        goto __TERMINAL;

    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpKThread;

    if(NULL == lpKernelThread->KernelThreadRoutine)    //If the main
routine is empty.
        goto __TERMINAL;

    dwRetValue =
lpKernelThread->KernelThreadRoutine(lpKernelThread->lpRoutineParam);

    //ENTER_CRITICAL_SECTION();
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    lpKernelThread->dwReturnValue = dwRetValue;    //Set the return
value of this thread.
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_TERMINAL;
//Change the status.
    //LEAVE_CRITICAL_SECTION();
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    ... ..
}

```

注意上述代码中的黑体部分，该部分实际上是调用了线程的功能函数（以用户提供的参数为参数，在应用程序调用 `CreateKernelThread` 的时候，`CreateKernelThread` 创建一个核心线程对象，并把用户提供的线程功能函数和参数存储到该对象中，详细信息请参考线程的创建部分）。需要注意的是，在从功能函数调用返回后，线程起始函数并没有马上返回，而是做了一些收尾处理，比如设置线程的返回值，设置线程核心对象的状态（设置为 `TERMINAL`）等。

从上述代码中还可看出，应用程序可以创建一个没有任何功能函数的“空线程”，因为在调用线程的功能函数前，线程起始函数先做了检查，若功能函数为空，则直接跳转到末尾，否则再调用功能函数。图 4-4 显示了线程起始函数和线程功能函数之间的关系。

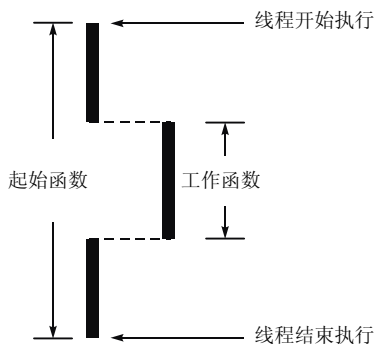


图 4-4 核心线程的生命周期

### 4.2.5 线程的优先级与调度

Hello China 当前版本的实现中，线程的调度算法是抢占式的基于优先级调度算法。在核心线程对象中，有两个变量。

```
DWORD                                     dwThreadPriority;
DWORD                                     dwScheduleCounter;
```

这两个变量是实现线程调度的基础。

在当前版本的实现中，只实现了一种算法，即优先级相关的时间片调度算法，该算法执行流程如下。

- (1) 线程创建时，dwScheduleCounter 初始化为 dwThreadPriority，即线程优先级的值；
- (2) 线程每运行一个时间片（系统时钟中断间隔），调度程序把 dwScheduleCounter 的值减一，并作为新的优先级；
- (3) 如果 dwScheduleCounter 减一后，结果是 0，那么重新初始化为 dwThreadPriority 的值。

根据这个算法，优先级越高的线程获得的 CPU 资源越多，而且优先级高的线程可以完全抢占优先级低的线程而投入运行。但这个算法也避免了优先级低的线程出现所谓“饿死”现象，即 CPU 时钟被优先级高的线程占有，优先级低的线程无法占有 CPU 而得不到运行。因为随着运行时间的增加，优先级高的线程，其 dwScheduleCounter 逐渐降低，在降低到一定程度时，优先级低的线程就有机会运行了。

比如，按照这个算法，系统中存在三个线程。

**线程 A，优先级为 6；**

**线程 B，优先级为 4；**

**线程 C，优先级为 2。**

假设 A 先运行，按照这个算法，系统将发生以下进程切换动作。

- 1. 一个时钟中断：调度程序减少 A 的 dwScheduleCounter（初始为 6，结果为 5），然后跟 Ready 队列的第一个线程（B）比较，发现 A 比 B 的优先级（dwScheduleCounter）大，于是继续运行 A；
- 2. 第二个时钟中断：调度程序减少 A 的 dwScheduleCounter，然后跟 Ready 队列的第一个线程（B）比较，发现仍然不小于（只有小于才发生切换），于是继续运行 A；
- 3. 第三个时钟中断：调度程序减少 A 的 dwScheduleCounter，跟 B 比较，发现小于 B，于是 A 入 Ready 队列（以 dwScheduleCounter 为关键字，此时为 3），然后恢复 B 的上下文，B 开始运行；
- 4. 第四个时钟中断：调度程序减少 B 的 dwScheduleCounter（结果为 3），然后跟 Ready 队列中的第一个线程（A）比较，发现不小于，于是继续运行；
- 5. 第五个时钟中断：调度程序减少 B 的 dwScheduleCounter（结果为 2），然后跟 A 比较，发现小于，于是 B 入 Ready 队列，恢复 A 上下文，A 继续运行；
- 6. 第六个时钟中断：调度程序减少 A 的 dwScheduleCounter（结果为 2），然后跟 Ready 队列的第一个线程（C）比较，发现不小于，于是继续运行；
- 7. 第七个时钟中断：调度程序减少 A 的 dwScheduleCounter（结果为 1），然后跟 C 比较，发现小于 C，于是 C 运行。

这三个线程运行的序列如图 4-5 所示：

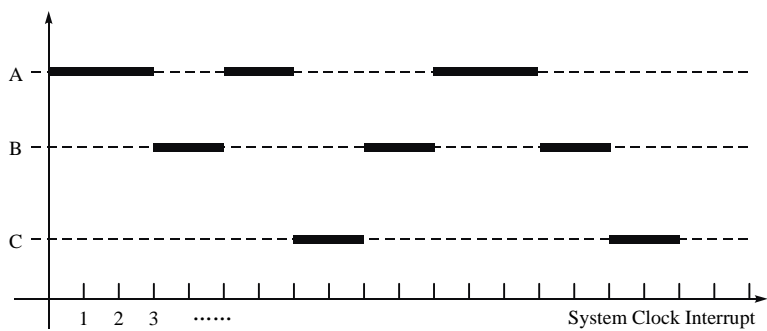


图 4-5 示例：线程运行时序

通过上面的描述可以看出，该算法是严格抢先的，比如，又有一个核心线程 D 被创建，D 的优先级为 8，那么，在下一个调度时刻（时钟中断），D 将会被优先选择执行，因为 D 的优先级在当前就绪线程队列中是最高的。

需要注意的是，上述算法仅仅是 Hello China 可以实现的调度算法中的一种，读者完全可以根据自己的需要，开发出适合自己应用的另外的调度算法。比如，上述调度算法为了兼顾低优先级的线程，采用了“优先级递减”的思路，随着高优先级任务的不断运行，其调度优先级将持续递减，直到为 0，然后又恢复为初始值。但这样的调度算法，有

的情况下可能满足不了需求。比如，有的系统要求严格优先，即优先级高的线程，将一直保持高优先级，不向低优先级的线程让步，这样读者就可以把优先级递减相关的代码删除，而实现一种严格优先的调度算法。

另外，为了实现上的方便，虽然 `dwThreadPriority` 可以取任何整数值，但为了规范起见，目前版本的 **Hello China**，只定义了下列几个可取值。

```
#define PRIORITY_LEVEL_REALTIME          0x00000020 //Kernel thread's
priority level.
#define PRIORITY_LEVEL_CRITICAL          0x00000010
#define PRIORITY_LEVEL_IMPORTANT         0x00000008
#define PRIORITY_LEVEL_NORMAL            0x00000004
#define PRIORITY_LEVEL_LOW               0x00000002
#define PRIORITY_LEVEL_LOWEST            0x00000001
#define PRIORITY_LEVEL_INVALID           0x00000000
```

线程的优先级可以在线程创建（`CreateKernelThread` 函数）的时候指定，建议应用程序在赋予线程优先级的时候，只采用上述定义值。

## 4.2.6 线程的创建

`CreateKernelThread` 函数完成线程的创建功能，该函数执行下列动作。

- (1) 调用 `CreateObject`（`ObjectManager` 提供）函数，创建一个核心线程对象；
- (2) 初始化该核心线程对象；
- (3) 创建线程堆栈，并初始化堆栈；
- (4) 把核心线程对象插入就绪队列（初始状态为就绪）或挂起队列（初始状态为挂起）。

下面是该函数的实现代码，为了方便阅读，我们分段列举解释。

```
static __KERNEL_THREAD_OBJECT* CreateKernelThread(__COMMON_OBJECT*
lpThis,
            DWORD                dwStackSize,
            DWORD                dwStatus,
            DWORD                dwPriority,
            __KERNEL_THREAD_ROUTINE lpStartRoutine,
            LPVOID               lpRoutineParam,
            LPVOID               lpReserved)
{
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    __KERNEL_THREAD_MANAGER* lpMgr        = NULL;
    LPVOID                   lpStack       = NULL;
```

```

        BOOL                                     bSuccess                                     = FALSE;

        if((NULL==lpThis)|| (NULL==lpStartRoutine))    //Parameter check.
            goto __TERMINAL;

        if((KERNEL_THREAD_STATUS_READY != dwStatus) && (KERNEL_THREAD_
STATUS_SUSPENDED != dwStatus))
            /*The initial status of a kernel thread should only be READY or SUSPENDED.
If the initial status is READY,then the kernel thread maybe scheduled to
run in the NEXT schedule circle(please note the kernel thread does not be
scheduled immediately),else,the kernel thread will be suspended,the kernel
thread in this status can be activated by ResumeKernelThread routine.*/
            goto __TERMINAL;

```

跟注释描述的一样，上述代码主要是检查创建线程的初始状态（也可以认为是参数合法性检查），本函数只创建状态为 **READY**（就绪）或 **SUSPENDED**（挂起）的线程，所有以其它值调用该函数的尝试，都将会失败。

```

        lpMgr = (__KERNEL_THREAD_MANAGER*)lpThis;
        lpKernelThread =
            (__KERNEL_THREAD_OBJECT*)ObjectManager.CreateObject(&ObjectMana
ger,
                                                                NULL,
                                                                OBJECT_TYPE_KERNEL_THREAD);
        if(NULL == lpKernelThread)    //If failed to create the kernel thread
object.
            goto __TERMINAL;
        if(!lpKernelThread->Initialize((__COMMON_OBJECT*)lpKernelThread))
//Failed to initialize.
            goto __TERMINAL;

```

上述代码调用 **ObjectManager** 提供的 **CreateObject** 函数创建核心线程对象。在目前的实现中，对核心线程对象也归纳到 **ObjectManager** 的管理框架中，即系统中创建的任何核心线程对象都会被 **ObjectManager** 记录，这样便于管理。

```

        if(0 == dwStackSize)    //If the dwStackSize is zero,then allocate
the default size's
            //stack.
            dwStackSize = DEFAULT_STACK_SIZE;
        else
        {
            if(dwStackSize < KMEM_MIN_ALLOCATE_BLOCK)    //If dwStackSize is

```

```

too small.

        dwStackSize = KMEM_MIN_ALLOCATE_BLOCK;
    }
    lpStack = KMemAlloc(dwStackSize,KMEM_SIZE_TYPE_ANY);
    if(NULL == lpStack)    //Failed to create kernel thread stack.
        goto __TERMINAL;

```

上述代码完成线程堆栈的创建。线程的堆栈实际上就是一块物理内存，对于堆栈的大小（dwStackSize），用户可以根据需要自行指定（在 CreateKernelThread 函数调用中通过参数传递），也可以不指定。若用户不指定堆栈大小（dwStackSize 参数设为 0），则系统创建一个缺省大小（DEFAULT\_STACK\_SIZE，目前定义为 16KB）的堆栈，否则根据用户指定的大小创建。但若用户指定的堆栈尺寸太小（小于 KMEM\_MIN\_ALLOCATE\_BLOCK），则系统会采用 KMEM\_MIN\_ALLOCATE\_BLOCK 代替用户指定的值。

若堆栈创建失败（内存申请失败），则 CreateKernelThread 函数会以失败告终。

```

//The following code initializes the kernel thread object created just
now.
    lpKernelThread->dwThreadID          = lpKernelThread->dwObjectID;
    lpKernelThread->dwThreadStatus       = dwStatus;
    lpKernelThread->dwThreadPriority     = dwPriority;
    lpKernelThread->dwScheduleCounter    = dwPriority; //***** CAUTION!!!
*****
    lpKernelThread->dwReturnValue        = 0L;
    lpKernelThread->dwTotalRunTime       = 0L;
    lpKernelThread->dwTotalMemSize       = 0L;
    lpKernelThread->lpCurrentDirectory   = NULL;        //Maybe updated in
the future.
    lpKernelThread->lpRootDirectory      = NULL;
    lpKernelThread->lpModuleDirectory    = NULL;

    lpKernelThread->bUsedMath             = FALSE;      //May be updated in
the future.
    lpKernelThread->dwStackSize          = dwStackSize ? dwStackSize :
DEFAULT_STACK_SIZE;
    lpKernelThread->lpInitStackPointer   = (LPVOID)((DWORD)lpStack +
dwStackSize);
    lpKernelThread->KernelThreadRoutine = lpStartRoutine; //Will
be updated.
    lpKernelThread->lpRoutineParam       = lpRoutineParam;
    lpKernelThread->ucMsgQueueHeader     = 0;
    lpKernelThread->ucMsgQueueTrial      = 0;

```

```

lpKernelThread->ucCurrentMsgNum      = 0;
lpKernelThread->dwLastError           = 0L;
lpKernelThread->lpParentKernelThread = NULL;      //Will be updated.

```

上述代码完成核心线程对象的部分初始化（有一些成员，需要进一步初始化），包括设置线程的 ID、优先级、当前状态等，需要注意的是，对线程核心对象中 `lpInitStackPointer` 的设置，不是设置为堆栈的起始地址，而是设置为堆栈的终止地址，因为堆栈是从高地址到低地址增长的。

```

//
//The initializing value of Instruction Pointer Register is
KernelThreadWrapper,
//this routine has a parameter,lpKernelThread,in order to pass this
parameter to the
//routine,we must build the stack frame correctly.
//The following code is used to build the kernel thread's stack
frame.
//It first 'push' the lpKernelThread into the stack,and subtract
8 from the stack
//pointer's value to simulate a procedure call,then initializes the
context of the
//kernel thread created just now by using INIT_KERNEL_THREAD_
CONTEXT_X macros.
//

*(DWORD*)((DWORD)lpStack + dwStackSize - 4) = (DWORD)lpKernelThread;
// "Push" the lpKernelThread into it's stack.

#ifdef __I386
    INIT_KERNEL_THREAD_CONTEXT_I(&(lpKernelThread->KernelThreadCont
ext), //Initialize context.
        (DWORD)KernelThreadWrapper,
        ((DWORD)lpStack + dwStackSize - 8)) //Please
notice the 8 is necessary.
#endif

```

上述代码比较关键，完成线程堆栈的初始化，以及线程硬件上下文的初始化。对于一个创建的线程，其起始运行地址并不是 `CreateKernelThread` 函数中用户指定的函数地址（`lpStartRoutine`），而是由系统提供的一个通用函数 `KernelThreadWrapper` 的地址，在 `KernelThreadWrapper` 函数中，调用了用户提供的线程功能函数 `lpStartRoutine`。因此，在初始化线程硬件上下文（`INIT_KERNEL_THREAD_CONTEXT_I` 宏）的时候，对于 EIP

寄存器的设置，是设置为 `KernelThreadWrapper` 函数的首地址的，这样一旦线程得到调度，将会从 `KernelThreadWrapper` 函数开始执行。对于线程硬件上下文初始化的详细过程，请参考本章中“线程的上下文”一节。

`KernelThreadWrapper` 函数的原型如下。

```
static VOID KernelThreadWrapper(__COMMON_OBJECT* lpKThread);
```

若线程一旦得到调度，将会开始执行该函数，而该函数却有一个线程核心对象指针（即刚创建的线程核心对象）为参数，如何把这个参数传递到该函数呢？显然，函数参数的获取是直接从堆栈中进行的，因此，为了让该函数能够访问 `lpKThread` 参数，我们必须把这个指针压入线程的堆栈，这就是上述代码中黑体部分的意义了。由于堆栈是从高往低增长的，因此必须把该函数压入线程堆栈所在内存的末尾。图 4-6 显示了这个过程。

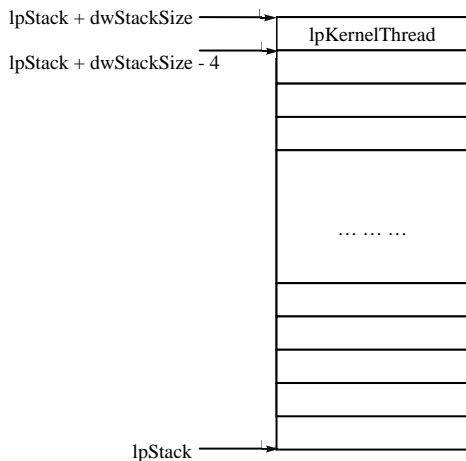


图 4-6 线程功能函数在堆栈中的起始位置

其中，`lpStack` 指向了堆栈所在内存的基地址，而 `lpStack+dwStackSize` 则指向了堆栈所在内存的高地址，为了向堆栈中最后一个双字中写入 `lpKernelThread`，必须以 `lpStack + dwStackSize - 4` 为指针。

这样 `KernelThreadWrapper` 函数的参数被压入堆栈了。另外一个问题就是，新创建的线程一旦得到调用，其起始执行地址将直接跳转到 `KernelThreadWrapper` 函数开始执行，需要注意的是，这个过程不是函数调用，整个过程中没有 `CALL` 指令。而通常情况下，对于函数参数的访问是建立在函数调用基础上的，对于函数调用，CPU 会执行如下的操作。

1. 把 `CALL` 指令后的第一条指令（下一条执行指令）的地址压入堆栈；
2. 跳转到目标函数。

这个过程会导致堆栈中被压入一个双字的元素，堆栈指针会变化。为了迎合这个过程，在初始化堆栈指针的时候，需要再减去 4，这即是 `INIT_KERNEL_THREAD_CONTEXT_I` 宏调用中，第三个参数的含义了。减去 8 的理由是这样的。

- (1) 在堆栈中压入 `lpKernelThread`，需要减去 4；
- (2) 为迎合 `CALL` 指令压入一个返回地址的操作，虚拟一个 `PUSH` 动作，使堆栈再减去 4。

因此，线程创建完毕后，其堆栈初始状态如图 4-7 所示。

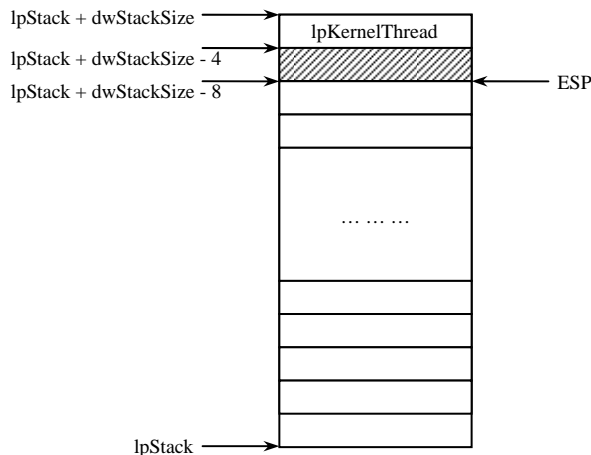


图 4-7 线程创建完毕后的堆栈状态

在这种堆栈框架下，线程一旦被调度执行，`KernelThreadWrapper` 函数就可以访问到自己的参数 (`lpKernelThread`) 了。

需要注意的是，在堆栈中压入了一个“空”双字，不会导致异常的发生，因为这仅仅是为了迎合 `CALL` 指令的动作而完成的一个虚拟操作。实际上，`KernelThreadWrapper` 函数永远不会返回，在 `KernelThreadWrapper` 函数的结尾处，已经把当前的核心线程对象从就绪队列中删除，这样该函数就不可能被再次调度，从而没有返回的机会，该“伪位置”就不可能被访问，详细信息，请参考本章“线程的结束”一节。

```
if(KERNEL_THREAD_STATUS_READY == dwStatus) //Add into
Ready Queue.
{

    if(!lpMgr->lpReadyQueue->InsertIntoQueue((__COMMON_OBJECT*)lpMgr->
lpReadyQueue,
        (__COMMON_OBJECT*)lpKernelThread,dwPriority))
        goto __TERMINAL;
```

```

    }
    else //Add into Suspended
Queue.
    {
        if(!lpMgr->lpSuspendedQueue->InsertIntoQueue((__COMMON_
OBJECT*)lpMgr->lpSuspendedQueue,
        (__COMMON_OBJECT*)lpKernelThread,dwPriority))
            goto __TERMINAL;
    }
    bSuccess = TRUE; //Now,the TRANSACTION of create a kernel thread
is successfully.
__TERMINAL:
    ... ..//Deal with some errors.
    return lpKernelThread;
}

```

上述代码根据创建的线程的状态（READY 或 SUSPENDED），把线程插入对应的队列。若线程的初始状态为 READY，CreateKernelThread 函数会把线程插入 lpReadyQueue，这样一旦下一个调度时机（时钟中断或系统调用发生）到达，该线程就可能会被调度执行（根据线程的优先级确定）。若线程的初始状态为 SUSPENDED，则该线程会被放入 lpSuspendedQueue，除非该线程被手工恢复（ResumeKernelThread），否则不会被调度。

### 4.2.7 线程的结束

线程的结束有两种方式。

- (1) 线程执行完功能函数，自然结束；
- (2) 被其它线程调用 TerminalKernelThread 函数强行终止。

其中，第一种结束情况属正常情况，在这种情况下，不会发生资源泄漏等情况，而在第二种情况下，被结束线程申请的系统资源可能得不到释放，从而造成资源的消耗。因此，一般情况下，不建议采用第二种方式结束一个线程。

上文中多次提到，一个新创建的线程刚开始被调度投入运行的时候，是从 KernelThreadWrapper 函数开始运行的。该函数的上半部分在“线程的上下文”中有介绍，在本节中，我们重点关注该函数的下半部分，因为这是线程的结束部分。

下面是该函数的相关代码，为了便于阅读，我们分段解释。

```

static VOID KernelThreadWrapper(__COMMON_OBJECT* lpKThread)
{
    __KERNEL_THREAD_OBJECT*      lpKernelThread      = NULL;
    __KERNEL_THREAD_OBJECT*      lpWaitingThread      = NULL;
    __PRIORITY_QUEUE*            lpWaitingQueue       = NULL;

```

```

        __PRIORITY_QUEUE*          lpReadyQueue          = NULL;
        DWORD                     dwRetValue              = 0L;
        DWORD                     dwFlags                 = 0L;

        ... ..
        dwRetValue =
lpKernelThread->KernelThreadRoutine(lpKernelThread->lpRoutineParam
);

```

上述代码中，黑体部分调用了线程的功能函数，在线程从功能函数返回的时候并没有结束，而是继续执行以下代码。

```

//ENTER_CRITICAL_SECTION();
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
lpKernelThread->dwReturnValue = dwRetValue;    //Set the return
value of this thread.
lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_TERMINAL;
//Change the status.
//LEAVE_CRITICAL_SECTION();
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

```

执行完功能函数后，该函数首先设置线程核心对象的返回值，以及线程状态(TERMINAL)。

```

//
//The following code wakeup all kernel thread(s) who waiting for this
kernel thread
//object.
//
lpWaitingQueue = lpKernelThread->lpWaitingQueue;
lpReadyQueue = KernelThreadManager.lpReadyQueue;
lpWaitingThread =
((__KERNEL_THREAD_OBJECT*)lpWaitingQueue->GetHeaderElement((__COMMON_OB
JECT*)lpWaitingQueue,
    NULL));
while(lpWaitingThread)
{
    lpWaitingThread->dwThreadStatus = KERNEL_THREAD_STATUS_READY;
    lpReadyQueue->InsertIntoQueue((__COMMON_OBJECT*)lpReadyQueue,
        (__COMMON_OBJECT*)lpWaitingThread,
        lpWaitingThread->dwScheduleCounter);
    lpWaitingThread = (__KERNEL_THREAD_OBJECT*)lpWaitingQueue->
    GetHeaderElement(

```

```

        (__COMMON_OBJECT*)lpWaitingQueue,
        NULL);
}

```

上述代码唤醒所有等待当前核心线程对象的其它线程。核心线程对象本身也是一个同步对象，其它线程可以等待核心线程对象。一旦核心线程对象的状态被设置为 **TERMINAL**，所有等到该对象的其它线程将被激活（类似 **EVENT** 对象的 **SetEvent** 调用）。上述代码就是用来激活所有等待该核心线程对象的其他线程的，这部分代码的详细含义，请参考“核心同步对象”相关的章节。

```

__TERMINAL:
    KernelThreadManager.lpTerminalQueue->InsertIntoQueue((__COMMON_
OBJECT*)KernelThreadManager.lpTerminalQueue,
        (__COMMON_OBJECT*)lpKernelThread,
        0L);    //Insert the current kernel thread object into TERMINAL
queue.

```

上述代码把当前线程核心对象插入终止队列（**lpTerminalQueue**）。下面的代码，从就绪队列（**lpReadyQueue**）中，提取一个就绪线程，并切换到就绪线程，这样当前线程宣告正式结束。需要注意的是，此后当前线程由于不会出现在就绪队列，因此永远得不到调度。处于结束队列（**lpTerminalQueue**）的核心线程对象，在合适的时机，将会被系统删除。

```

//
//The following code fetch the first READY kernel thread from Ready
Queue,restore it's
//context,and switch to this kernel thread to continue running.
//

lpKernelThread = (__KERNEL_THREAD_OBJECT*)KernelThreadManager.
lpReadyQueue->GetHeaderElement(
    (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
    NULL);

if(NULL == lpKernelThread) //If this condition is occurs,the
system will crash.
{
    PrintLine("In KernelThreadWrapper.");
    PrintLine(lpszCriticalMsg);
    return;
}

```

上述代码完成当前线程和就绪线程的切换，在 `SwitchTo` 函数被调用后，当前的执行线索将转移到从就绪队列提取的就绪线程，`SwitchTo` 函数不会返回，从而最后一条指令（`return`）永远得不到执行。需要注意的是，`__ENTER_CRITICAL_SECTION(NULL, dwFlags)`宏被调用，但在本函数中，并没有一个`__LEAVE_CRITICAL_SECTION(NULL, dwFlags)`与之对应，实际上，`__LEAVE_CRITICAL_SECTION`宏的功能在 `SwitchTo` 函数中已经做了实现。详细的切换信息，请参考本文中“线程的切换”相关章节。

#### 4.2.8 线程的消息队列

消息队列是一个由数组结构构成的循环队列，即核心线程对象（\_\_KERNEL\_THREAD\_OBJECT）定义的 `KernelThreadMsg` 数组，为方便阅读，把核心线程对象定义中关于线程消息队列的部分代码列举如下。

```

... ..
__KERNEL_THREAD_MESSAGE
        KernelThreadMsg[MAX_KTHREAD_MSG_NUM];
UCHAR                                     ucMsgQueueHeader;
UCHAR                                     ucMsgQueueTrial;
UCHAR                                     ucCurrentMsgNum;
UCHAR                                     ucAligment;
__EVENT*                                 lpMsgEvent;
... ..

```

`KernelThreadMsg` 数组是一个类型为 `__KERNEL_THREAD_MESSAGE` 结构的数组，根据目前的定义，该数组大小是 32 (`MAX_KTHREAD_MSG_NUM = 32`)。`ucAlignment` 是为了实现数据对齐（32 比特对齐），`ucQueueHeader` 和 `ucQueueTail` 分别指向队列的头部和尾部，其中，`ucQueueHeader` 指向队列的第一个非空元素（若队列非空的话），而 `ucQueueTail` 指向了消息队列中第一个空元素（若队列不满的话）。`ucCurrentMsgNum` 则指出了当前队列中消息的个数，如图 4-8 所示。

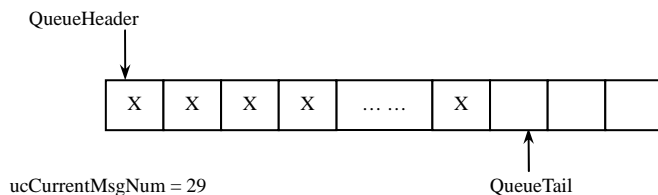


图 4-8 线程的消息队列

系统中的核心线程可以通过 `SendMessage` 函数调用向队列中发送消息，如果队列不满，则消息被存储在 `ucQueueTail` 所指向的位置，同时 `ucQueueTail` 后移一个元素（指向下一个非空位置），`ucCurrentMsgNum` 增加 1，如图 4-9 所示。

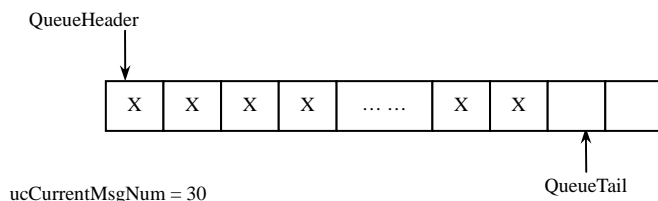


图 4-9 线程消息队列的添加操作

线程本身可以调用 `GetMessage` 函数，从自己的消息队列中获取消息，若当前消息队列为空，则 `GetMessage` 函数阻塞（通过等待一个 `EVENT` 核心对象），直到有其它线程向本线程的消息队列中发送消息。若消息队列非空，则 `GetMessage` 函数取走 `ucQueueHeader` 所指位置的消息，然后 `ucQueueHeader` 向后移动一个位置，`ucCurrentMsgNum` 减 1，如图 4-10 所示。

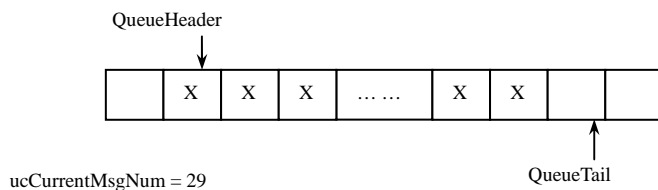


图 4-10 线程消息队列的删除操作

队列当前的状态（空或满）可以通过判断 `ucCurrentMsgNum` 的大小获得。

`lpMsgEvent` 是一个 `__EVENT` 内核对象，该对象用来完成消息操作的同步。在当前 `Hello China` 的实现中，`GetMessage` 函数是按照同步操作实现的，即若队列中有消息，则该函数立即返回，并把队列中的消息返回给用程序，若队列中没有消息，则该函数阻塞，直到有消息到达。阻塞操作就是通过等待该事件对象实现的，下面是 `GetMessage` 函

数的相关代码。

```
static BOOL MgrGetMessage(__COMMON_OBJECT*
lpThread, __KERNEL_THREAD_MESSAGE* lpMsg)
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread = NULL;
    DWORD                       dwFlags       = 0L;

    if((NULL == lpThread) || (NULL == lpMsg)) //Parameters check.
        return FALSE;
    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpThread;
    if(MsgQueueEmpty(lpThread))
    {
        lpKernelThread->lpMsgEvent->WaitForThisObject(
            (__COMMON_OBJECT*)(lpKernelThread->lpMsgEvent));
//Block the current thread.
    }
    ... ..
    return TRUE;
}
```

在上述实现中，GetMessage 函数首先判断线程的消息队列是否为空，若为空，则调用 lpMsgEvent 对象的 WaitForThisObject 函数等待 lpMsgEvent 对象。

而 lpMsgEvent 对象是被 SendMessage 函数唤醒的，SendMessage 函数的相关实现代码如下。

```
static BOOL MgrSendMessage(__COMMON_OBJECT*
lpThread, __KERNEL_THREAD_MESSAGE* lpMsg)
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread = NULL;
    BOOL                       bResult       = FALSE;
    DWORD                       dwFlags       = 0L;

    if((NULL == lpThread) || (NULL == lpMsg)) //Parameters check.
        return bResult;

    if(MsgQueueFull(lpThread))                //If the queue is full.
        return bResult;
    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpThread;

    ... .. //Put the message into kernel thread's message queue.
```

```

//
//Set the signal to indicate there is one message to be get at least.
//
lpKernelThread->lpMsgEvent->SetEvent((__COMMON_OBJECT*)(lpKernelThread->lpMsgEvent));

return bResult;
}

```

在上述实现中，每向线程队列发送一个消息，就会调用 `SetEvent` 函数设置事件对象的状态，这样若当前线程因为调用 `GetMessage` 函数阻塞，在此时刻就会被唤醒。

对于线程的消息队列，最后需要解释的就是 `__KERNEL_THREAD_MESSAGE` 结构本身了，顾名思义，该结构用来装载具体的消息，定义如下。

```

BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_MESSAGE)
    WORD            wCommand;
    WORD            wParam;
    DWORD           dwParam;
    //DWORD          (*MsgAssocRoutine)(__KERNEL_THREAD_MESSAGE*);
END_DEFINE_OBJECT()

```

`wCommand` 是一个命令字，指出具体的消息类型，比如键盘按下、鼠标按下等，也可以由用户自己定义。`wParam` 和 `dwParam` 是两个跟 `wCommand` 关联的参数，比如，跟“键盘按下”这样一个消息相关联，可以是具体被按下的键的 ASCII 码（可以通过 `wParam` 设置）。

消息队列机制的应用十分广泛，也十分灵活，从理论上说，任何基于多线程通信的应用模型都可以使用消息队列来实现。

## 4.2.9 线程的切换——中断上下文

在 `Hello China` 的当前实现中，采用的是可抢占式的线程调度方式，即每个时钟中断发生后，中断处理程序会打断当前执行的线程，检查线程的就绪队列（`lpReadyQueue`），选择一个优先级最高的线程投入运行。这样的调度机制，可确保优先级最高的线程能够马上得到调度。

这样，就涉及到一个问题，在中断上下文中，如何保存当前的线程上下文状态，并选择另外一个线程，恢复其上下文，并投入运行。在本节中，我们对这个问题进行详细描述。

首先，在进入正式讨论前，先介绍 Intel IA32 CPU 的一条指令——`iretd`。这条指令的用途很广泛，最基础的用途是从中断中返回。

在 IA32 构架的 CPU 中，每次中断发生的时候，CPU 会做如下动作（没有考虑不同层级之间的转换，只考虑在核心保护模式下的情况）。

- 1. 把当前执行的线程所在的代码段寄存器（CS）、EIP 寄存器和标志寄存器（EFLAGS），以及一个可选的错误代码压入当前堆栈；
- 2. 根据中断向量号，查找中断描述表（IDT），并跳转到 IDT 指定的中断处理程序；
- 3. 中断处理程序执行完毕后，执行一条 iretd 指令，该指令恢复先前在堆栈中保存的 CS、EFlags、EIP 寄存器信息，并继续执行。

因此，当中断发生，CPU 跳转到中断处理程序前后，当前线程堆栈的堆栈框架如图 4-11 所示。

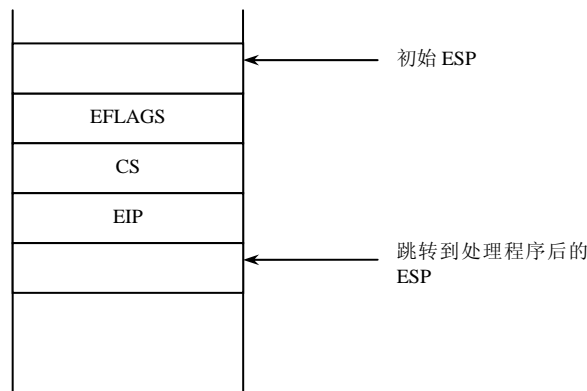


图 4-11 中断发生后的堆栈框架

当中断处理程序执行完毕，最后一条指令 iretd 恢复上述保存在堆栈中的寄存器，然后继续执行中断发生前的代码。可以看出，iretd 指令的动作是一次性从堆栈中恢复 EFlags、CS 和 EIP。

该指令除了用于从通常中断中返回之外，还用于任务的切换。假设在中断发生前，运行的线程是 T1，这时候发生一次时钟中断，CPU 按照上述方式，在 T1 的堆栈中保存 T1 的相关寄存器（EFlags、CS、EIP），然后跳转到中断处理程序。中断处理程序在执行具体的任务前，首先保存 T1 线程的其它相关寄存器（EAX/EBX 等通用寄存器），然后才开始执行具体的中断处理任务（定时器处理、睡眠线程唤醒等）。在执行完毕后，中断处理程序会从就绪队列中选择一个优先级最高的线程，假设为 T2，然后恢复其寄存器信息（包括 EAX 等通用寄存器，还包括线程 T2 的堆栈寄存器 ESP），并建立上述堆栈框架（这时候的上述寄存器，就不是线程 T1 的，而是新选择的线程 T2 的），这时候的目标堆栈，也不是 T1 的，而是 T2 的，上述堆栈框架建立完毕后，执行 iretd 指令，这样恢复运行的就不再是线程 T1，而是新选择的线程 T2。



```
out 0xa0,al
pop eax
iret
```

入口程序首先保存 EAX 寄存器，然后判断 `gl_general_int_handler` 是否为 0，该标号实际上就是采用 C 语言实现的中断处理程序。若该标号为 0，则说明对应的 C 语言实现的中断处理程序不存在（可能 Master 没有加载），这样直接跳转到 `ll_contiune` 编号处，恢复中断控制器后从中断中返回。

若 `gl_general_int_handler` 不为 0，则说明存在对应的 C 语言处理函数，于是该中断入口程序首先保存当前线程的通用寄存器信息，然后把当前中断向量号压入堆栈，并调用 `gl_general_int_handler` 函数。在调用 `gl_general_int_handler` 函数前，当前线程各寄存器在堆栈中的框架如图 4-12 所示。

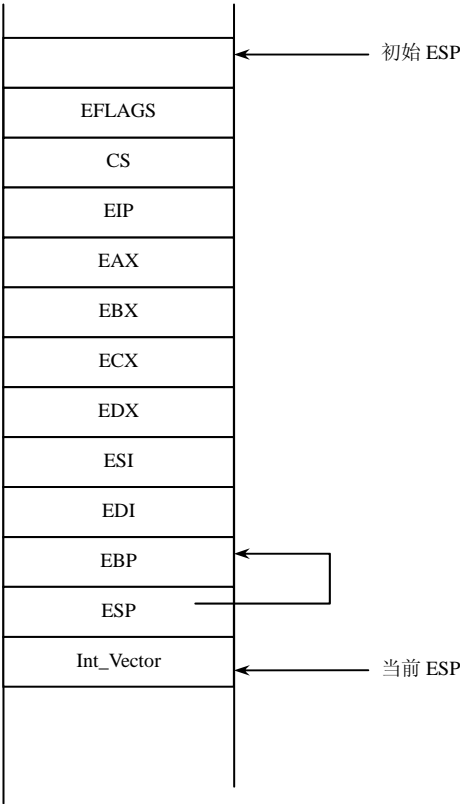


图 4-12 当前线程的各寄存器在堆栈中的布局

因为 ESP 是一个动态变化的指针，每次向堆栈中压入一个变量，ESP 就增加对应的

字节，因此，在上述堆栈框架中，保存的 ESP 寄存器的值，是在压入 EBP 后 ESP 的值。只所以保存该值，是因为 `gl_general_int_handler` 函数可以通过该值来访问堆栈框架。

`gl_general_int_handler` 函数的原型如下。

```
VOID GeneralIntHandler(DWORD dwVector, LPVOID lpEsp);
```

可以看出，该函数有两个参数，即对应的中断向量号和堆栈框架指针。其中，堆栈向量号就是上述代码中压入的向量号，而堆栈框架指针则就是上述堆栈框架中保存的 ESP 的值。需要注意的是，中断处理函数是在当前线程的堆栈中执行的。这样通过上述两个参数，`GeneralIntHandler` 函数就可以访问中断向量号和堆栈框架。

`GeneralIntHandler` 函数根据中断向量号，再调用对应的中断处理程序。比如，时钟中断的中断向量号是 0x20，则 `GeneralIntHandler` 函数会根据该向量号，查找一个数组，在该数组中，保存了每个中断处理例程的地址，找到对应的例程后，`GeneralIntHandle` 函数 `r` 就会调用对应的例程。对于线程的调度，目前只在时钟中断中进行处理，因此，只有 0x20 号中断发生后，才会发生线程的重新调度。

在 0x20 号中断（时钟中断）的处理程序中，所有线程相关的调度工作，通过一个函数 `ScheduleFromInt` 来实现，时钟中断处理在处理完所有其它任务后，在程序最后调用该函数。下面是该函数的实现代码，为了阅读方便起见，我们分段进行解释。

```
static VOID ScheduleFromInt(__COMMON_OBJECT* lpThis, LPVOID lpESP)
{
    __KERNEL_THREAD_OBJECT*      lpNextThread    = NULL;
    __KERNEL_THREAD_OBJECT*      lpCurrentThread = NULL;
    __KERNEL_THREAD_MANAGER*      lpMgr           = NULL;
    __KERNEL_THREAD_CONTEXT*      lpContext       = NULL;

    if((NULL == lpThis) || (NULL == lpESP))    //Parameters check.
        return;

    lpMgr = (__KERNEL_THREAD_MANAGER*)lpThis;

    if(NULL == lpMgr->lpCurrentKernelThread)  //The routine is called
first time in the

                                                //initialization process.
                                                //In this case, the routine
does not need

                                                //to save the current kernel's
context,

                                                //it only fetch the first ready
kernel thread
```

```

//from Ready Queue, and switch to
this kernel

//thread.
{
    lpNextThread =
    (__KERNEL_THREAD_OBJECT*)KernelThreadManager.lpReadyQueue->
    GetHeaderElement(
        (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
        NULL);
    if(NULL == lpNextThread)                //If this case is occurs,
the system is crash.
    {
        PrintLine("In ScheduleFromInt, lpCurrentKernelThread ==
NULL.");
        PrintLine(lpszCriticalMsg);
        return;
    }
    KernelThreadManager.lpCurrentKernelThread = lpNextThread;
//Update the current kernel thread pointer.
    lpNextThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
//Update the status.

    lpContext = &lpNextThread->KernelThreadContext;
    SwitchTo(lpContext);                //Switch to the next
kernel thread.
}

```

在操作系统刚刚启动，还没有发生线程切换（时钟中断被禁止）的时候，是在一个叫做初始化上下文中执行的，这时候的代码属初始化代码（也可以认为是一个初始化线程）。但 **Hello China** 的实现不把这部分代码作为任何线程，因此这时候，**lpCurrentKernelThread** 是空值。就绪队列中却不是空的，因为初始化代码创建了 **shell**、**IDLE** 等线程，这些线程被放入就绪队列。

一旦初始化代码执行完毕，就会使能时钟中断，这时候，一旦发生时钟中断，该函数就会被调用。若 **lpCurrentKernelThread** 是空值，说明该函数是第一次被调用，这时候，该函数会从就绪队列中取出第一个线程对象（优先级最高的线程对象），并调用 **SwitchTo** 函数，切换到这个线程。**SwitchTo** 函数是实现线程切换的汇编语言函数，在后面我们会详细描述，现在只要知道，一旦以目标线程的上下文信息（**lpContext**）调用了 **SwitchTo** 函数，就会切换到目标线程开始运行。

```
else
```

```

{
    lpCurrentThread = KernelThreadManager.lpCurrentKernelThread;
    lpContext = &lpCurrentThread->KernelThreadContext;
    SaveContext(lpContext, (DWORD*)lpESP);    //Save the current
kernel thread's context.

```

若 `lpCurrentKernelThread` 不是空，则说明该函数不是第一次被调用（有且只有第一次被调用的时候，`lpCurrentKernelThread` 为空），当中断发生的时候，已经有线程在运行了。这种情况下，该函数首先获得当前运行的线程（实际上是中断发生前运行的线程）的上下文结构（`lpContext`），然后调用 `SaveContext` 函数保存当前线程的上下文信息。`SaveContext` 函数的具体实现，在后面我们再详细介绍。

保存线程上下文信息后，该函数会根据线程的状态，做进一步判断。

```

switch(lpCurrentThread->dwThreadStatus)
{
case KERNEL_THREAD_STATUS_BLOCKED:
case KERNEL_THREAD_STATUS_TERMINAL:
case KERNEL_THREAD_STATUS_SLEEPING:

    //ENTER_CRITICAL_SECTION();
    lpCurrentThread->dwScheduleCounter -= 1;
    if(0 == lpCurrentThread->dwScheduleCounter)
    {
        lpCurrentThread->dwScheduleCounter =
            lpCurrentThread->dwThreadPriority;
    }
    lpCurrentThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
    lpContext = &lpCurrentThread->KernelThreadContext;
    SwitchTo(lpContext);
    break;                                //This instruction will
never reach.

default:
    break;
}

```

一般情况下，线程的状态应该是 `RUNNING`，因为被打断的时候，线程是处于运行状态的。但下列三种状态在线程被时钟中断打断的时候也可能出现。

**(1) `KERNEL_THREAD_STATUS_BLOCKED`**。这种状态下的线程，是正在执行一个等待共享资源的操作（`WaitForThisObject`），在等待共享资源的时候，线程的状态首先

被设置为 **BLOCKED**，然后被插入共享资源的本地等待队列。这个时候，若线程在被插入本地等待队列前发生中断，则其状态会为 **BLOCKED**。这种情况下，中断调度程序不会再选择其它线程投入运行，而是继续恢复 **BLOCKED** 线程的上下文，让该线程继续执行。在线程被成功插入共享对象的本地等待队列后，会再次发生一次线程切换，这时候，当前优先级最高的就绪线程会被调度执行；

**(2) KERNEL\_THREAD\_STATUS\_SLEEPING**。与上述类似，若当前线程执行 Sleep 调用，则线程的状态会首先被设置为 **KERNEL\_THREAD\_STATUS\_SLEEPING**，然后会被插入睡眠队列。在插入睡眠队列前，若有时钟中断发生，则线程的状态会为 **SLEEPING**，这种情况下，调度程序也不会重新调度其它线程，而是直接恢复当前线程，因为当前线程马上就会停止运行（被插入睡眠队列后，会引发一次线程调度）；

**(3) KERNEL\_THREAD\_STATUS\_TERMINAL**。在线程执行结束，但线程的“扫尾”工作还没有完成的时候，会发生这种情况。这时候，线程的状态已经被设置为 **TERMINAL**，但还在处理一些线程的扫尾工作，比如唤醒等待该线程对象的其它线程等，这时候若发生时钟中断，则会出现线程状态为 **TERMINAL** 的线程。这个时候，时钟中断仍然继续恢复该线程的上下文，使得该线程继续执行。因为该线程马上就可以执行完毕，从而引发另一次线程的切换。

若当前线程的状态不是上述几种情况，则时钟中断会重新调度。这时候，时钟中断处理程序会把当前线程的状态设置为 **KERNEL\_THREAD\_STATUS\_READY**，并递减其调度计数，若调度计数达到 0，则重新设置其调度计数（详细信息请参考本文“线程的调度”一节），然后把当前线程对象插入就绪队列（lpReadyQueue）。由于就绪队列是一个优先队列，因此在插入的时候，使用当前线程的调度计数（dwScheduleCounter）为优先字段，插入就绪队列。这种情况下，当前线程的 dwScheduleCounter 决定了在就绪队列的位置。如果当前线程的 dwScheduleCounter 为所有线程中最大的，则该线程仍然会被排在队列头部，这样下一次被选择调度的线程，仍然是当前线程。相关如下代码。

```
lpCurrentThread->dwThreadStatus      =
    KERNEL_THREAD_STATUS_READY;
lpCurrentThread->dwScheduleCounter   -= 1;
if(0 == lpCurrentThread->dwScheduleCounter)
{
    lpCurrentThread->dwScheduleCounter      =
lpCurrentThread->dwThreadPriority;
}
lpCurrentThread->dwTotalRunTime       += SYSTEM_TIME_SLICE;

KernelThreadManager.lpReadyQueue->InsertIntoQueue(    //Insert into
```

ready queue.

```
(__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
(__COMMON_OBJECT*)lpCurrentThread,
lpCurrentThread->dwScheduleCounter
);
```

在把当前线程插入就绪队列后，调度程序会重新检查就绪队列，从就绪队列中选择第一个线程（优先级最高的线程）投入运行。按照 Hello China 的实现，就绪队列中至少有一个线程，即 IDLE 线程，因此，若从就绪队列中获取线程操作失败，则是一种严重异常的情况，这种情况下，会打印出一串告警信息，然后系统会进入死循环。代码如下。

```
lpNextThread = (__KERNEL_THREAD_OBJECT*)
    KernelThreadManager.lpReadyQueue->GetHeaderElement( //Get next
ready one.
    (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
    NULL);

if(NULL == lpNextThread)    //If this case occurs,the system will crash.
{
    PrintLine("In ScheduleFromInt,lpCurrentKernelThread != NULL.");
    PrintLine(lpszCriticalMsg);
    Dead();
    return;
}
```

获得下一步该调度的线程后，首先设置其状态为 `KERNEL_THREAD_STATUS_RUNNING`，然后把 `lpCurrentKernelThread` 设置为该线程，并调用 `SwitchTo` 函数，切换到该线程。代码如下。

```
lpNextThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
KernelThreadManager.lpCurrentKernelThread = lpNextThread;
//Update the current kernel thread.
lpContext = &lpNextThread->KernelThreadContext;
SwitchTo(lpContext);    //Switch to the new kernel thread.
}
}
```

至此，对中断上下文中的线程调度就解释完了。在此总结一下。

(1) 目前的实现，在所有的硬件中断中，Hello China 只在时钟中断中实现了线程调度，实际上，大多数操作系统都是这么做的；

(2) 当前线程的上下文信息，是在中断处理程序的入口处（采用汇编语言编写的代

码)进行保存的;

(3) 在中断处理程序中, 调用 `ScheduleFromInt` 函数来实现线程的调度, 需要注意的是, 这个函数在中断处理程序的最后部分被调用, 因为该函数不会返回, 直接切换到目标线程开始运行;

(4) 对线程的切换, 在 IA32 CPU 上采用 `iretd` 指令实现;

(5) `ScheduleFromInt` 函数调用 `SaveContext` 函数保存当前线程的上限文, 调用 `SwitchTo` 函数来切换到目标线程;

(6) 对于状态是 `BLOCKED`、`TERMINAL`、`SLEEPING` 的线程, 不做调度, 而是恢复其上下文, 使得这些线程继续运行。因为处于这些状态的线程, 都是临时状态, 很快就会被切换出去。

两个底层的函数 `SaveContext` 和 `SwitchTo`, 实现线程上下文的保存和切换工作, 在本文的后面部分进行详细描述。

#### 4.2.10 线程的切换——系统调用上下文

除了在系统时钟中断处理程序中完成线程的调度(线程切换)外, 在运行的线程试图获取共享资源(调用 `WaitForThisObject` 函数), 而共享资源当前状态为不可用的时候, 也需要发生切换, 这时候, 当前线程(获取共享资源的线程)会阻塞, 并插入共享资源的本地线程队列, 然后再从就绪队列中提取优先级最高的线程投入运行。这个过程不是发生在中断上下文中的, 而是发生在系统调用上下文中, 这个时候的线程切换, 我们称为“系统调用上下文中的切换”。

系统调用上下文中的线程切换, 与时钟中断上下文中的线程切换基本上是一样的, 唯一的区别是, 系统调用上下文的线程切换, 其切换前建立的堆栈框架不一样。在中断上下文中的切换, 堆栈框架的建立是 CPU 自己完成的(即中断发生后, CPU 把当前线程的 CS、EFlags 和 EIP 寄存器自动压入堆栈), 而在系统调用上下文中, 堆栈框架的建立, 则是由 `CALL` 指令建立的。我们通过一个例子说明这个问题。比如, 一个 `EVENT` 对象的 `WaitForThisObject` 函数实现如下。

```
static DWORD WaitForEventObject(__COMMON_OBJECT* lpThis)
{
    __EVENT*          lpEvent          = NULL;
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    __KERNEL_THREAD_CONTEXT* lpContext    = NULL;
    DWORD              dwFlags          = 0L;

    if(NULL == lpThis)
        return OBJECT_WAIT_FAILED;
```

```

lpEvent = (__EVENT*)lpThis;
//ENTER_CRITICAL_SECTION();
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
if(EVENT_STATUS_FREE == lpEvent->dwEventStatus)
{
    //LEAVE_CRITICAL_SECTION();
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return OBJECT_WAIT_RESOURCE;
}
else
{
    lpKernelThread = KernelThreadManager.lpCurrentKernelThread;
    //ENTER_CRITICAL_SECTION();
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_BLOCKED;
    //LEAVE_CRITICAL_SECTION();
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    lpEvent->lpWaitingQueue->InsertIntoQueue(
        (__COMMON_OBJECT*)lpEvent->lpWaitingQueue,
        (__COMMON_OBJECT*)lpKernelThread,
        0L);
    lpContext = &lpKernelThread->KernelThreadContext;
    KernelThreadManager.ScheduleFromProc(lpContext);
}
return OBJECT_WAIT_RESOURCE;
}

```

该函数首先判断当前事件对象的状态，若当前状态为 **FREE** (**EVENT\_STATUS\_FREE**)，则函数等待成功，直接返回，否则，说明当前事件对象处于未发信号状态，需要等待，这个时候，当前线程首先把自己的状态设置为 **KERNEL\_THREAD\_STATUS\_BLOCKED**（把当前线程状态设置为 **BLOCKED** 后，该线程会一直执行，直到阻塞。详细信息请参考“线程的切换—中断上下文”一节），然后插入当前对象的等待队列。在插入等待队列之后，使用当前线程的上下文对象 (**lpContext**)，调用 **ScheduleFromProc**（该函数是 **KernelThreadManager** 的一个成员函数）函数，来引发一个重新调度。

若把调用 **ScheduleFromProc** 函数的 C 代码翻译成汇编代码，应该是下面这个样子。

```

push lpContext    //Prepare parameter for ScheduleFromProc routine.
call ScheduleFromProc    //Call this routine.
mov eax,OBJECT_WAIT_RESOURCE
ret

```

即首先把 lpContext 压入堆栈，然后调用 ScheduleFromProc 函数。上述指令执行完毕之后，当前堆栈框架的样子如图 4-13 所示。

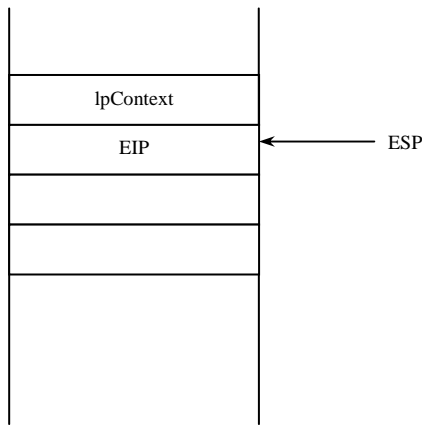


图 4-13 调用 ScheduleFromProc 函数前的堆栈框架

之所以深入分析堆栈框架，是因为这是线程切换的关键。可以看出，ScheduleFromProc 函数是这个过程的关键，该函数的功能是先保存当前线程的硬件上下文，把当前线程的硬件上下文保存到 lpContext 对象中（这也是为什么该函数需要当前线程上下文对象指针作为参数的原因），然后再从就绪队列中选择一个线程，并恢复所选线程的上下文，使所选线程投入运行。下面是该函数的代码，为方便起见，分段进行解释。

```
__declspec(naked) static VOID ScheduleFromProc(__KERNEL_THREAD_CONTEXT*
lpContext)
{
```

首先该函数使用 \_\_declspec(naked) 来进行修饰，这样的目的是防止编译器生成任何附加的汇编代码，以免对当前线程的堆栈框架造成影响。这个修饰关键字的详细含义，请参考本书附录部分相关内容。

```
#ifdef __I386__
__asm{
    push ebp
    mov ebp,esp
    add ebp,0x08           //add ebp,0x04 ??????
    push ebp               //Save the ESP register.
    sub ebp,0x08           //sub ebp,0x04 ??????
    push eax
    push ebx
```

```
        push ecx
        push edx
        push esi
        push edi
        pushfd
    }    //Now,we have saved the current kernel's context into stack
        successfully.
    #else
    #endif
```

上述汇编代码完成了针对 IA32 CPU 上当前线程上下文（硬件寄存器）的保存（保存到当前线程的堆栈中）。上述代码执行完毕后，当前线程的堆栈框架应该如图 4-14 所示。

错误！

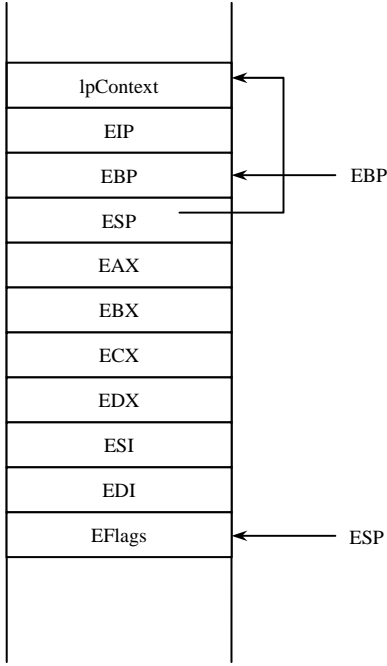


图 4-14 当前线程各寄存器在堆栈中的布局

上述代码执行完毕之后，ESP 指向堆栈中 EFlags 寄存器所在的位置，EBP 则指向堆栈中 EBP 寄存器所在的位置，这样通过 EBP 寄存器的值，就可以很容易地访问到 lpContext 变量的值，这是十分重要的。

当前线程的上下文保存到堆栈中之后，需要进一步从堆栈中保存到当前线程对象的 Context 数据结构中。之所以先保存到堆栈中，再从堆栈中保存到 Context 数据结构中，下面的代码完成了从堆栈到 Context 的保存。

//The following code saves the current kernel thread's context into kernel thread object.

```
#ifdef __I386__
__asm{
    mov eax,dword ptr [ebp + 0x08]    //Now,the EAX register
countains the lpContext.
    mov ebp,esp

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_EFLAGS],ebx    //Save
eflags.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_EDI],ebx    //Save EDI.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_ESI],ebx    //Save ESI.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_EDX],ebx    //Save EDX.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_ECX],ebx    //Save ECX.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_EBX],ebx    //Save EBX.
    add ebp,0x04

    mov ebx,dword ptr [EBP]
    mov dword ptr [eax + CONTEXT_OFFSET_EAX],ebx    //Save EAX.
    add ebp,0x04

    mov ebx,dword ptr [ebp]
    mov dword ptr [eax + CONTEXT_OFFSET_ESP],ebx    //Save ESP.
    add ebp,0x04
```

```

        mov ebx,dword ptr [ebp]
        mov dword ptr [eax + CONTEXT_OFFSET_EBP],ebx        //Save EBP.
        add ebp,0x04

        mov ebx,dword ptr [ebp]
        mov dword ptr [eax + CONTEXT_OFFSET_EIP],ebx        //Save EIP.
        add ebp,0x04
    } //Now,we have saved the current kernel thread's context into
kernel thread object.
    #else
    #endif

    ChangeContext(); //Call ChangeContext to re-schedule all kernel
threads.
}

```

上述代码十分简单，首先把 `lpContext` 的值从堆栈中拷贝到 `EAX` 寄存器（通过 `EBP` 寄存器访问堆栈），由于 `lpContext` 是一个指针，因此可以通过间接寻址的方式，通过 `EAX` 寄存器访问线程上下文数据结构（`Context`）的各成员。需要注意的是，这个过程一直是通过 `EBP` 寄存器来从当前线程堆栈拷贝寄存器数据的。

把当前线程的上下文信息保存完毕之后，就需要执行一个线程切换动作，从就绪队列中选择一个就绪的线程替换当前线程了。`ChangeContext` 函数就是完成这项工作的，该函数代码如下。

```

static VOID ChangeContext()
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread    = NULL;
    __KERNEL_THREAD_CONTEXT*    lpContext        = NULL;
    BYTE                        strThread[12];
    DWORD                       dwThread        = 0L;
    DWORD                       dwFlags         = 0L;

    lpKernelThread = (__KERNEL_THREAD_OBJECT*)
        KernelThreadManager.lpReadyQueue->GetHeaderElement(
            (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
            NULL);
    if(NULL == lpKernelThread) //If this case occurs,the system may crash.
    {
        PrintLine("In ChangeContext routine.");
        PrintLine(lpszCriticalMsg);
        PrintLine("Current kernel thread: ");
    }
}

```

```

        strThread[0] = ' ';
        strThread[1] = ' ';
        strThread[2] = ' ';
        strThread[3] = ' ';

        dwThread = (DWORD)KernelThreadManager.lpCurrentKernelThread;
        Hex2Str(dwThread,&strThread[4]);
        PrintLine(strThread);
        return;
    }

    lpContext = &lpKernelThread->KernelThreadContext;

    //ENTER_CRITICAL_SECTION();           //Here,the interrupt must be
disabled.
    __ENTER_CRITICAL_SECTION(NULL,dwFlags)
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
    KernelThreadManager.lpCurrentKernelThread = lpKernelThread;
    SwitchTo(lpContext);
}

```

该函数十分简单，直接从就绪队列中，提取第一个（优先级最高的）线程，修改其状态，并把当前线程设置为选择的线程，然后调用 **SwitchTo** 函数，切换到目标线程。需要注意的是，该函数也执行了一个错误检查，即若从就绪队列提取线程失败（就绪队列中无任何线程），则打印一个错误信息，然后死机。就绪队列中没有任何线程，是一种严重的系统错误，一般情况下是不可能发生的，只有因为编程错误、堆栈溢出等发生的情况下才可能发生。

至此，从系统调用上下文中切换线程的过程就介绍完了。

#### 4.2.11 上下文保存和切换的底层函数

在上面的描述中，提到了两个完成线程切换和上下文保护的底层函数 **SwitichTo** 和 **SaveContext**，在本节中，我们对这两个函数的实现进行描述。

**SaveContext** 函数完成线程上下文的保护，该函数的代码如下。

```

static VOID SaveContext(__KERNEL_THREAD_CONTEXT* lpContext,DWORD*
lpdwEsp)
{
    if((NULL == lpContext) || (NULL == lpdwEsp))           //Parameters
check.

        return;
}

```

```

    DisableInterrupt();
#ifdef __I386__                //i386's implementation.
    lpContext->dwEBP = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwEDI = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwESI = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwEDX = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwECX = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwEBX = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwEAX = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwEIP = *lpdwEsp;
    lpdwEsp ++;
    lpdwEsp ++;                //Skip the CS's space.
    lpContext->dwEFlags = *lpdwEsp;
    lpdwEsp ++;
    lpContext->dwESP = (DWORD)lpdwEsp;
#else
#endif
    EnableInterrupt();

    return;
}

```

代码十分简单，没有采用任何汇编语句，因为作为参数之一的 `lpdwEsp` 指针实际上指向了被时钟中断处理程序保存的当前线程的上下文信息，这些信息位于当前线程的堆栈中。因此，该函数直接把堆栈中的上下文信息拷贝到当前线程的上下文数据结构中即可。从此也可以看出，该函数只能被中断处理程序调用，用来完成当前线程的上下文保护。

另外一个函数 `SwitchTo`，既可以被中断处理程序调用，也可以被系统调用函数调用，完成线程的切换。该函数接受一个线程上下文结构指针作为参数，先把要切换到的目标线程的上下文恢复到其堆栈中，然后执行 `iretd` 指令，切换到目标线程。代码如下所示，为了便于阅读，我们分段解释。

```

__declspec(naked) static VOID SwitchTo(__KERNEL_THREAD_CONTEXT* lpContext)
{

```

```
#ifdef __I386__                                     //Intel's x86 CPU
implementation.
    __asm{
        cli
        push ebp
        mov ebp,esp                                //Build the
stack frame.
        mov eax,dword ptr [ebp + 0x08]
```

到此为止，是在当前线程（调用 SwitchTo 函数的线程）堆栈中执行的，即 ESP 寄存器的值，指向的是当前线程的堆栈位置。上述代码通过 EBP 寄存器的值来访问 lpContext 参数（目标线程的上下文信息指针），把 lpContext 参数拷贝到 EAX 寄存器中。下面的代码的执行堆栈则切换到了目标线程的堆栈上（注意下面黑体部分的代码），但在执行 iretd 之前，所有的代码仍然属于当前线程（即当前线程的执行堆栈切换了）：

```
mov esp,dword ptr [eax + CONTEXT_OFFSET_ESP]
push dword ptr [eax + CONTEXT_OFFSET_EFLAGS]
xor ebx,ebx
mov bx,word ptr [eax + CONTEXT_OFFSET_CS]
push ebx
push dword ptr [eax + CONTEXT_OFFSET_EIP] //Push EIP to stack.
                                           //Now,we have built the
correct
                                           //stack frame.
```

上述代码首先切换到目标线程的堆栈，然后把目前线程的 EFlags 寄存器、CS 寄存器和 EIP 寄存器的值恢复到堆栈中。完成上述操作后，目标线程的堆栈情形如图 4-15 所示。

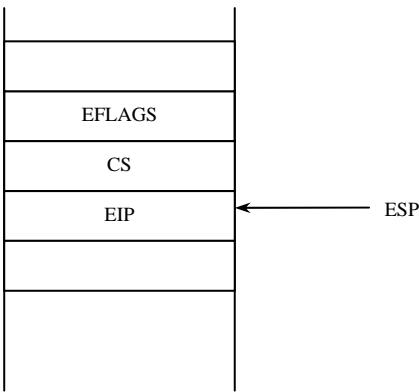


图 4-15 切换线程前建立的堆栈框架

这个堆栈框架读者应该是十分熟悉的，即中断发生后 CPU 建立的堆栈框架（只考虑

相同执行权限下的情况)。在这个堆栈框架下，只要执行一条 `iretd` 指令，就可以完成线程的切换了。但这时候还不是切换的时候，因为 `EAX`、`EBX` 等通用寄存器的值还没有恢复到目标线程所对应的值，因此，下面的代码完成这个过程。

```

push dword ptr [eax + CONTEXT_OFFSET_EAX]           //Save eax to stack.
push dword ptr [eax + CONTEXT_OFFSET_EBX]           //Save ebx to stack.
push dword ptr [eax + CONTEXT_OFFSET_ECX]           //ecx
push dword ptr [eax + CONTEXT_OFFSET_EDX]           //edx
push dword ptr [eax + CONTEXT_OFFSET_ESI]           //esi
push dword ptr [eax + CONTEXT_OFFSET_EDI]           //edi
push dword ptr [eax + CONTEXT_OFFSET_EBP]           //ebp

mov ebp,dword ptr [esp]                             //Restore the ebp register.
add esp,0x04
mov edi,dword ptr [esp]                             //Restore edi
add esp,0x04
mov esi,dword ptr [esp]                             //Restore esi
add esp,0x04
mov edx,dword ptr [esp]                             //Restore edx
add esp,0x04
mov ecx,dword ptr [esp]                             //Restore ecx
add esp,0x04
mov ebx,dword ptr [esp]                             //Restore ebx
add esp,0x04

```

上述代码很简单，就是从目标线程的 `Context` 中，把保存的通用寄存器的值拷贝到堆栈中，然后再从堆栈中恢复到通用寄存器中。只所以从堆栈中恢复，而不是直接从 `Context` 结构中恢复，是考虑到实现上的统一（因为在保存这些寄存器的时候，也是从堆栈中保存的）。下面的代码首先解除硬件中断（通知中断控制器），然后执行 `iretd` 指令，切换到目标线程。

[illegible]

```

        retn                                //This instruction will
never be reached.
    }
    #else
    #endif
}

```

在 `iretd` 指令执行完毕后，当前的执行线程就已经是目标线程了，因此，`iretd` 后面的指令（`retn`）永远不会被执行，即该函数永远不会返回。实际上，该函数没有返回前就已经终止执行了，当然，当前线程就已经被挂起或终止了。在当前线程再次被恢复执行的时候，开始执行地点是当初保存的 `EIP` 寄存器所指向的位置，而不是继续执行上述函数中的 `retn` 指令。

#### 4.2.12 线程的睡眠与唤醒

线程在执行的过程中可以调用 `Sleep` 函数，暂时进入睡眠状态，一定时间之后，再继续运行，睡眠的时间由 `Sleep` 函数的参数指定。`Sleep` 函数把当前线程对象插入睡眠队列（`lpSleepingQueue`），然后引发一个线程重调度。

每次时钟中断，中断处理程序都会检查当前睡眠队列中，是否有睡眠时间到的线程，若有这样的线程，则时钟中断处理程序会从睡眠队列中把这些睡眠的线程删除，然后插入就绪队列，这样在合适的时刻，这些线程就会又被调度执行。

可以看出，睡眠时间虽然可以采用 `Sleep` 函数的参数以毫秒（`ms`）为单位进行指定，但实际的睡眠时间粒度应该是系统时钟批频率。假设系统时钟中断周期为  $T$ （`ms`），而线程调用 `Sleep` 函数的时候，指定一个参数为  $t$ （`ms`），则实际上，该线程的睡眠时间应该为：

$$(t/T)*T + T (t \% T \neq 0)$$

其中， $t/T$  为  $t$  除  $T$  所得的结果的整数部分，而  $t\%T$  则是  $t$  对  $T$  取模所得的结果。当然，如果  $t$  刚好能够被  $T$  整除，则睡眠时间就是  $t$ 。

上述所谓的睡眠时间是线程处于睡眠状态的时间（在睡眠队列中的时间），上述时间到达后，线程并不一定马上得到调度，因为线程仅仅被重新插入就绪队列。若线程的优先级不是就绪队列中最高的，则可能不会被马上调度；若线程的优先级是最高的，则可以马上得到调度。

## 4.3 V1.5 版本中核心线程的实现

### 4.3.1 概述

在 Hello China 的 V1.0 版本中, 采用的是基于优先级递减的轮转调度算法, 这样可充分照顾到各个优先级的核心线程, 又可保证高优先级的核心线程获得更多的调度机会。但这种调度方式, 在时间要求非常严格的实时系统中, 却可能存在不足。因为随着运行时间的增加, 核心线程的优先级是递减的, 当一个处理关键实时任务的核心线程, 优先级递减到普通核心线程的优先级的时候, 其优先权就跟普通核心线程一样。

在 V1.5 的实现中, 增加了另外的严格按照优先级进行的调度算法, 只要系统中有更高优先级的线程, 较低优先级的核心线程便不会被调度, 而且核心线程的优先级不会随着运行时间的增加而降低。这样可充分保证高优先级的任务得到处理, 符合实时系统的需要。

在 V1.5 的实现中, 对于调度算法, 也以单独的函数模块进行实现, 调度程序只是调用这个调度算法函数, 获取下一个可调度的核心线程, 而不是直接参与调度算法。这样可使得系统很容易的容纳其它调度算法。比如, 可通过替代 V1.5 现有的调度算法实现函数, 增加另外的定制调度算法。

### 4.3.2 核心线程调度时机

在 V1.0 的实现中, 只有在两个时机会发生核心线程的重调度:

- 1、系统时钟中断处理程序结束后;
- 2、调用 WaitForThisObject 或 WaitForThisObjectEx 函数, 等待系统核心对象的时候。

而在 V1.5 的实现中, 运行调度程序的时机比 V1.0 的实现增加了很多, 这样可大大提高系统的实时性。在 V1.5 的实现中, 下列情况下会发生核心线程的重调度:

- 1、系统时钟中断;
- 2、等待共享资源 (调用 WaitForThisObject 或 WaitForThisObjectEx 函数);
- 3、释放共享资源;
- 4、线程睡眠;
- 5、线程睡眠结束;
- 6、线程创建;
- 7、线程结束;
- 8、任何设备的中断服务程序结束时。

一旦重新调度, 系统中处于 `KERNEL_THREAD_STATUS_READY` 状态的具有最高

优先级的核心线程便可得到 CPU，从而尽快投入运行。

## 4.4 V1.5 核心线程管理器（KernelThreadManager）的实现

在 Hello China V1.5 中，对核心线程管理对象进行了优化，优化后的定义如下（V1.0 版本的定义，请参考本章 V1.0 版本核心线程的实现）：

```
typedef DWORD (*__KERNEL_THREAD_ROUTINE)(LPVOID);

BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_MANAGER)
    __KERNEL_THREAD_OBJECT*      lpCurrentKernelThread;
    __PRIORITY_QUEUE*             lpRunningQueue;
    __PRIORITY_QUEUE*             lpSuspendedQueue;
    __PRIORITY_QUEUE*             lpSleepingQueue;
    //__PRIORITY_QUEUE*           lpReadyQueue;
    __PRIORITY_QUEUE*             lpTerminalQueue;
    __PRIORITY_QUEUE*
ReadyQueue[MAX_KERNEL_THREAD_PRIORITY + 1];

    DWORD                          dwNextWakeupTick;

    BOOL                           (*Initialize)(__COMMON_OBJECT*
lpThis);

    __KERNEL_THREAD_OBJECT*        (*CreateKernelThread)(
    __COMMON_OBJECT*              lpThis,
    DWORD                          dwStackSize,
    DWORD                          dwStatus,
    DWORD                          dwPriority,
    __KERNEL_THREAD_ROUTINE
lpStartRoutine,
    LPVOID                          lpRoutineParam,
    LPVOID                          lpReserved);
    __KERNEL_THREAD_OBJECT*        (*GetScheduleKernelThread)(
    __COMMON_OBJECT*              lpThis,
    DWORD                          dwPriority);
    VOID                           (*AddReadyKernelThread)(
```

```

        __COMMON_OBJECT* lpThis,
        __KERNEL_THREAD_OBJECT* lpThread);

VOID
lpThis,
        __COMMON_OBJECT*          lpKernelThread
    );

BOOL
(*SuspendKernelThread)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
);

BOOL
(*ResumeKernelThread)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
);

VOID
(*ScheduleFromProc)(
    __KERNEL_THREAD_CONTEXT*   lpContext
);

VOID
(*ScheduleFromInt)(
    __COMMON_OBJECT*          lpThis,
    LPVOID                    lpESP
);

DWORD
(*SetThreadPriority)(
    __COMMON_OBJECT*          lpKernelThread,
    DWORD                     dwNewPriority
);

DWORD
(*GetThreadPriority)(
    __COMMON_OBJECT*          lpKernelThread
);

DWORD
(*TerminalKernelThread)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
);

BOOL
(*Sleep)(
    __COMMON_OBJECT*          lpThis,
    DWORD                     dwMilliSecond
);

BOOL
(*CancelSleep)(
    __COMMON_OBJECT*          lpThis,
    __COMMON_OBJECT*          lpKernelThread
);

DWORD
(*GetLastError)(

```

```

        __COMMON_OBJECT*      lpThis
    );
    DWORD (*SetLastError)(
        __COMMON_OBJECT*      lpThis,
        DWORD                   dwNewError
    );
    DWORD (*GetThreadID)(
        __COMMON_OBJECT*      lpThis
    );
    BOOL (*SendMessage)(
        __COMMON_OBJECT*
lpKernelThread,

        __KERNEL_THREAD_MESSAGE* lpMsg
    );
    BOOL (*GetMessage)(
        __COMMON_OBJECT*
lpKernelThread,

        __KERNEL_THREAD_MESSAGE* lpMsg
    );
    BOOL (*MsgQueueFull)(
        __COMMON_OBJECT*      lpKernelThread
    );
    BOOL (*MsgQueueEmpty)(
        __COMMON_OBJECT*      lpKernelThread
    );
    BOOL (*LockKernelThread)(
        __COMMON_OBJECT*      lpThis,
        __COMMON_OBJECT*
lpKernelThread);
    VOID (*UnlockKernelThread)(
        __COMMON_OBJECT*      lpThis,
        __COMMON_OBJECT*
lpKernelThread);
    END_DEFINE_OBJECT() //End of the kernel thread manager's
definition.

```

最大的修改，增加了一个优先队列数组，该数组以核心线程的优先级作为索引，每个元素都是一个优先队列对象，特定优先级的核心线程，全部存放在对应的优先队列内。详细的实现介绍如下。

### 4.4.1 V1.5 核心线程队列的实现

在 Hello China V1.0 的实现中，所有处于就绪状态的队列，都被放在了一个队列 `lpReadyQueue` 中。而在 Hello China V1.5 的实现中，则定义了一个长度是最大线程优先级的队列数组，每个处于就绪状态的核心线程，都按照其优先级，放到对应的队列中。在 Hello China V1.5 的实现中，定义了 `MAX_KERNEL_THREAD_PRIORITY` 个线程优先级，这样就绪队列数组就由 `MAX_KERNEL_THREAD_PRIORITY` 个元素组成，每个线程按照其优先级作为索引，找到队列数组对应的元素，然后插入队列。下图示意了这个优先级队列数组的组织结构：

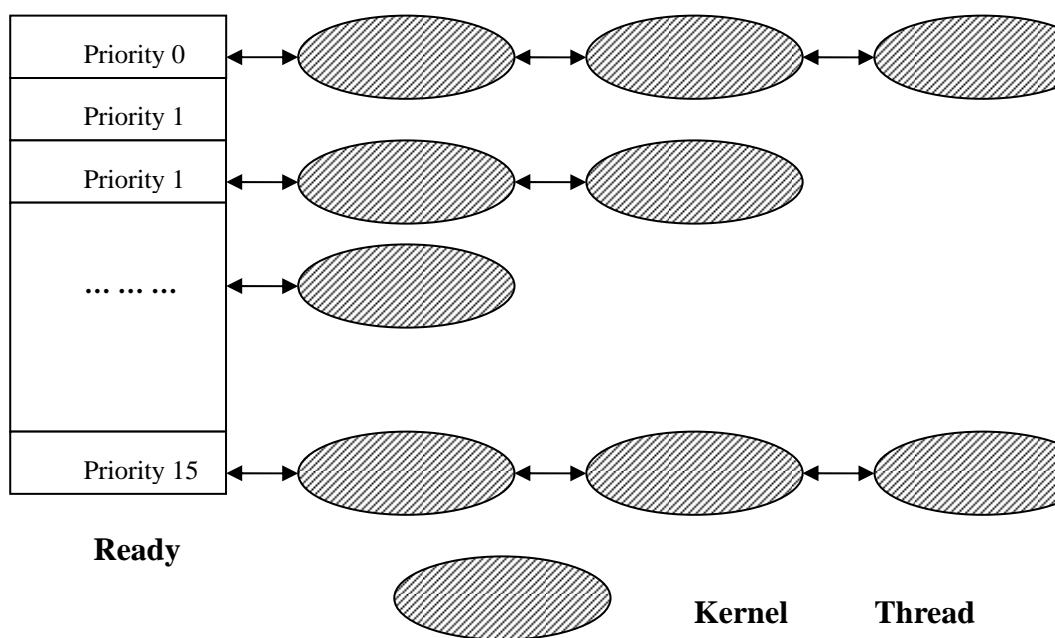


图 4-16 1.5 版本的核心线程队列

其它的线程队列，比如挂起队列、睡眠队列等，都按照原来的组织。

按照这样一种组织结构实现，可很容易的实现按照优先级的线程调度策略。在 V1.5 的实现中，实现了下列严格按照优先级进行调度的调度算法：

1、任何情况下，总是选择优先级最高的线程投入运行；

- 2、对于相同优先级的线程，按照先到先服务的原则，即队头的线程对象首先得到运行（这样就要求线程队列是一个优先队列）。

为了实现上的独立性，单独把调度算法拿出来，当作一个独立的函数实现，这样需要调度的时候，只需要调用该调度函数，获得下一个要投入运行的线程即可。在将来改变调度算法的时候，只需要替换该函数即可。

下列是 V1.5 实现的调度算法接口函数，该函数接受一个线程优先级数值和一个指向核心线程管理对象的指针，返回一个优先级大于或等于所接受优先级的核心线程对象。若没有这样的对象（优先级大于或等于给出的优先级），则返回 NULL：

```
__KERNEL_THREAD_OBJECT* GetScheduleKernelThread(
    __KERNEL_THREAD_MANAGER* lpThis,
    DWORD dwPriority)
{
    if((NULL == lpThis) || (dwPriority > MAX_KERNEL_THREAD_PRIORITY))
    {
        return NULL;
    }

    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    for(DWORD i = 0; i <= MAX_KERNEL_THREAD_PRIORITY; i++)
    {
        lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpMgr->
            ReadyQueue[MAX_KERNEL_THREAD_PRIORITY -
i]->GetHeaderElement(
                (__COMMON_OBJECT*)ReadyQueue[i - 1],
                NULL); //Get the header element.
        if(lpKernelThread) //Get one.
        {
            return lpKernelThread;
        }
    }
    return lpKernelThread; //Can not find.
}
```

上述函数完成了从就绪队列中选择一个核心线程的功能，AddReadyKernelThread 则实现相反的功能—往就绪队列中添加一个核心线程对象。该函数也十分简单，如下：

```
static void AddReadyKernelThread(__COMMON_OBJECT* lpThis,
```

```

    __KERNEL_THREAD_OBJECT* lpKernelThread)
{
    if((NULL == lpThis) || (NULL == lpKernelThread)) //Invalid
parameters.
    {
        return;
    }
    //Validate this kernel thread's priority.
    if(lpKernelThread->dwThreadPriority >
MAX_KERNEL_THREAD_PRIORITY)
    {
        __BUG(); //Print out this position.
        return;
    }
    __PRIORITY_QUEUE* lpReadyQueue =
        ((__KERNEL_THREAD_MANAGER*)lpThis)->ReadyQueue[
            lpKernelThread->dwThreadPriority];
    lpReadyQueue->InsertIntoQueue(
        (__COMMON_OBJECT*)lpReadyQueue,
        (__COMMON_OBJECT*)lpKernelThread,
        0); //Insert into queue.
}

```

上述函数首先检查核心线程对象优先级的合法性，这是非常有必要的，因为当前的实现，是把线程的优先级作为索引，来直接索引就绪队列数组的。因此，万一出现核心线程对象的优先级超出预定范围，会导致数组越界访问。当然，优先级超出预定范围的核心线程对象，也必然是一个非法的核心线程对象。

另外，在核心线程管理对象的初始化函数（Initialize）中，需要对就绪队列数组进行初始化，即创建每一个就绪队列数组对象。

## 4.5 V1.5 核心线程对象（KernelThreadObject）的实现

在 Hello China V1.5 的实现中，对核心线程对象（\_\_KERNEL\_THREAD\_OBJECT）进行了优化，删除了其中不必要的数据成员，并对其中的一些成员进行了更改。相应地，为了实现等待多个同步对象等功能，又增加了部分数据成员。优化后的核心线程对象如下：

[illegible]

相对 V1.0 版本，主要的修改内容有：

- 1、删除了在 V1.0 中用于调度的 `dwScheduleCounter` 成员，因为在 V1.5 的实现中，是严格按照优先级来进行调度的，无需该成员；
- 2、在 V1.0 的实现中，该对象包含了一个类型为 `__KERNEL_THREAD_CONTEXT` 的数据结构，用于保存当前线程的硬件上下文。在 V1.5 的实现中，则把该变量替换成了一个指针，硬件的上下文直接保存在当前线程的堆栈中，新增加的指针指向核心线程的堆栈中的上下文信息。详细信息，请参考本文后续描述；
- 3、在 V1.0 的实现中，采用了一个事件对象来同步核心线程的消息队列，而在 V1.5 的实现中，则直接添加了一个消息队列的等待队列（`lpMsgWaitingQueue`）对象，当核心线程调用 `GetMessage` 试图获得消息的时候，若消息队列为空，则会阻塞在该队列中。

### 4.5.1 V1.5 版本中硬件上下文的保存

在 V1.0 的实现中，对硬件上下文的保存，是直接保存在核心线程对象的 `KernelThreadContext` 成员中的。每当线程切换发生的时候，原来的线程的硬件上下文，保存在该结构体中，而新选择投入运行的线程，则从该结构体中获得其硬件上下文。当时的设计，是考虑到线程上下文可能被程序更改的情况，这样单独把线程上下文拿出来，会带来很多的方便。但实际表明，这种需求是非常小的，因此在 V1.5 中做了更改，线程上下文直接保存在线程的堆栈中，而核心线程对象则保存了一个指向核心线程上下文的指针，这样通过该指针，也可以方便的访问到线程的上下文。

而对线程上下文本身的定义，也做了更改，更改后的定义如下：

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_CONTEXT)
#ifdef __I386
    DWORD        dwEFlags;
    WORD          wCS;
    WORD          wReserved;
    DWORD         dwEIP;
    DWORD         dwEAX;
    DWORD         dwEBX;
    DWORD         dwECX;
    DWORD         dwEDX;
    DWORD         dwESI;
    DWORD         dwEDI;
    DWORD         dwEBP;
    //DWORD        dwESP;
```

```
END_DEFINE_OBJECT( )
```

与 V1.0 不同的是，删除了硬件上下文中的一个数据成员 dwESP。在中断或异常发生后，由汇编语言编写的中断处理程序入口模块，会建立如下的堆栈框架：

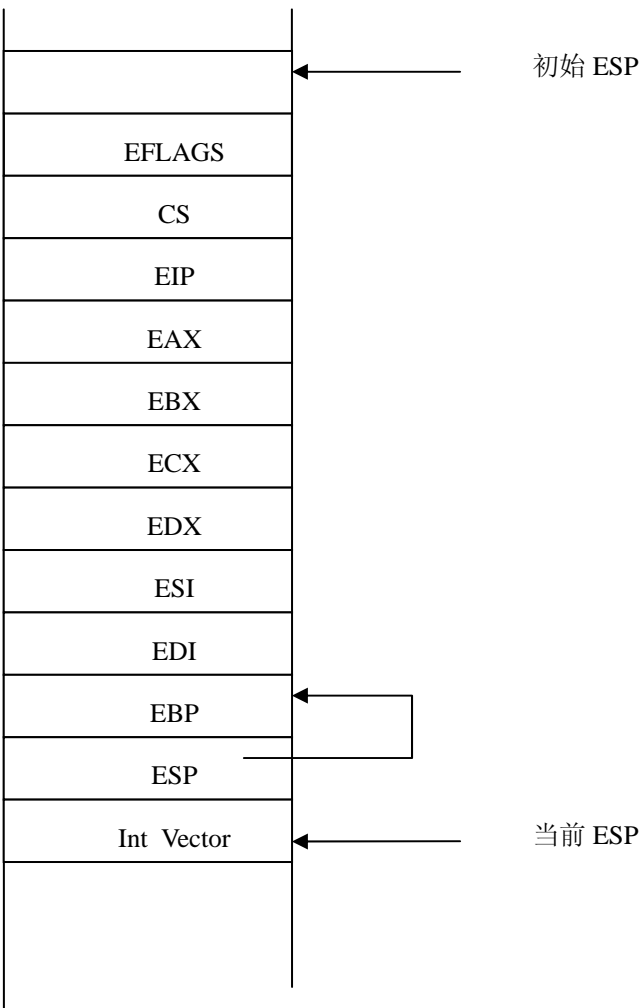


图 4-17 1.5 版本的核心线程堆栈框架

然后调用 GeneralIntHandler 函数，该函数接受两个参数：一个是中断向量，另外一

个是保存的堆栈指针，即图中的 ESP 值（该值指向了 EBP 寄存器在堆栈中的位置）。显然，这个堆栈框架，跟核心线程上下文的定义（\_\_KERNEL\_THREAD\_CONTEXT）是吻合的，因此，我们只要在 GeneralIntHandler 里面，把传递过来的 ESP 指针保存到核心线程对象的 lpKernelThreadContext 里面就可以了。

在切换到新的线程时，只需要把新线程的 lpKernelThreadContext 装载到 ESP 寄存器中，就切换到了新线程的堆栈，然后恢复所有寄存器，并执行 iretd 指令即可，下面的汇编代码，示意了该过程：

```
__declspec(naked) VOID __SwitchTo(__KERNEL_THREAD_CONTEXT*
lpContext)
{
__asm{
    push ebp
    mov ebp,esp
    mov esp,dword ptr [ebp + 0x08] //Switched to new thread.
    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx

    mov al,0x20
    out 0x20,al
    out 0xa0,al

    pop eax
    iretd
}
}
```

上述切换函数，只能在中断上下文中调用，因为该切换函数解除了 8259 中断控制器的中断请求。

但在进程上下文中切换的时候，却有些麻烦，因为需要建立上述形式的堆栈框架。在过程上下文中，定义了一个函数\_\_SaveAndSwitch，来完成当前线程上下文的保存，并切换到新选择的核心线程。该函数原型如下：

```
__declspec(naked) VOID __SaveAndSwitch(__KERNEL_THREAD_CONTEXT**
```

```
lppOldContext,__KERNEL_THREAD_OBJECT** lppNewContext);
```

该函数被 `ScheduleFromProc` 函数调用，用于完成核心线程在过程上下文中的调度。因此，在调用该函数前，必须获得当前线程的上下文，以及待调度线程的上下文，这些工作都是 `ScheduleFromProc` 函数完成的。

`__SaveAndSwitch` 被调用后，当前线程的堆栈框架中只保存了两个参数——`lppOldContext` 和 `lppNewContext`，以及函数返回地址，如下：

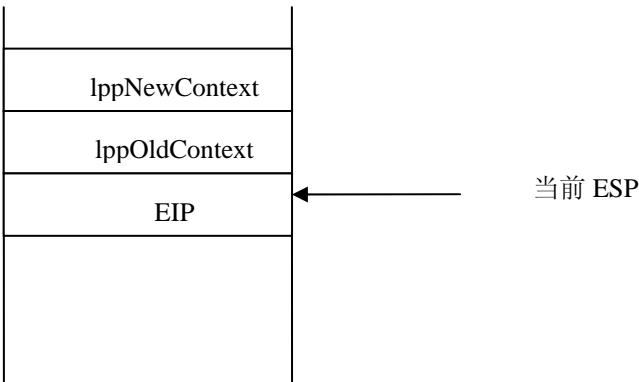


图 4-18 `__SaveAndSwitch` 调用后的堆栈框架

为了建立目标堆栈框架，在 `__SaveAndSwitch` 函数所在的源文件内，定义了两个静态全局变量，并借助这两个静态全局变量实现了当前线程堆栈框架的保存。如下：

```
static DWORD dwTmpEip = 0;
static DWORD dwTmpEax = 0;
static DWORD dwTmpEbp = 0;

__declspec(naked) void __SaveAndSwitch(__KERNEL_THREAD_CONTEXT**
    lppOldContext,__KERNEL_THREAD_CONTEXT** lppNewContext)
{
    __asm{
        mov dwTmpEbp,esp //Save ESP to global variable.
        pop dwTmpEip //Save EIP to global variable.
        push eax
```

```

    pop dwTmpEax //Save EAX to global variable.
    pushfd       //Save EFlags.
    xor eax,eax
    mov ax,cs
    push eax     //Save CS.
    push dwTmpEip //Restore EIP
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ebp

    //Now, we have built the target stack frame, save it to
    *lppOldContext.

    mov ebp, dwTmpEbp
    mov ebx, dword ptr [ebp + 0x04]
    mov dword ptr [ebx], esp //Now, save ESP to *lppOldContext.

    //Restore the new thread's context, and switch to it.
    mov ebx, dword ptr [ebp + 0x08]
    mov esp, dword ptr [ebx] //Restored the ESP register.
    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx
    pop eax
    iretd
}
}

```

需要注意的是，该函数被调用的时候，当前核心线程还在执行，尚未切换到新的核心线程。该函数首先保存当前核心线程的一些寄存器信息（保存到当前正在运行的核心线程的堆栈中），然后把堆栈指针保存在\*lppOldContext 变量中（该变量实际上就是指向当前核心线程对象的 lpKernelThreadContext 变量）。

保存完当前核心线程的上下文信息之后，通过\*lppNewContext 变量，获得新核心线

程的上下文信息的指针（实际上就是待运行核心线程的堆栈指针），然后把 ESP 寄存器的值恢复为新核心线程的堆栈指针，这时候操作的堆栈，已经是新核心线程的堆栈了。通过连续的几条 POP 指令，把新核心线程的上下文进行恢复，然后执行一条 iretd 指令，就切换到新核心线程被换出的位置并开始运行了。

## 4.5.2 线程的调度—中断上下文

在 Hello China V1.0 的实现中，只会在系统时钟中断发生的时候，才会对线程进行重新调度。这样对普通的应用可能是可以满足的，但对于一些实时的应用，则可能会有问题。因此，在 Hello China V1.5 的实现中，对线程的调度，不再局限于时钟中断中，而安排在所有中断中，在任何中断（包括系统时钟中断）处理程序执行完毕，返回用户线程之前，都会做一个线程调度。

在 V1.0 的实现中，对于所有线程调度的功能，都是在 ScheduleFromInt 函数中实现的，该函数原型如下：

```
static VOID ScheduleFromProc(__COMMON_OBJECT* lpThis, LPVOID lpEsp);
```

在 V1.5 的实现中，仍然保留了该函数，但对其进行了更改，下面是 V1.5 中，ScheduleFromProc 的实现代码：

```
static VOID ScheduleFromProc(__COMMON_OBJECT* lpThis, LPVOID lpEsp)
{
    if((NULL == lpThis) || (NULL == lpEsp)) //Invalid parameters.
    {
        return;
    }

    __KERNEL_THREAD_MANAGER* lpMgr =
(__KERNEL_THREAD_MANAGER*)lpThis;

    __KERNEL_THREAD_OBJECT* lpThread = NULL;
    __KERNEL_THREAD_OBJECT* lpCurrentThread = NULL;

    lpCurrentThread = lpMgr->lpCurrentKernelThread;
    if(NULL == lpCurrentThread) //Called first time.
    {
        lpThread =
lpMgr->GetScheduleKernelThread((__COMMON_OBJECT*)lpMgr,
                                0); //Get one kernel thread from ready queue to schedule.
```

```

        if(NULL == lpThread)    //Not any kernel thread exists,fatal
error!
    {
        PrintLine("Fatal error!No kernel thread has been created!");
        return;
    }
    lpMgr->lpCurrentKernelThread = lpThread;
    lpThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
    lpThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
    __SwitchTo(lpThread->lpKernelThreadContext); //Switch to new
thread.
    return; //Should not reach here.
}
else
{
    lpCurrentThread->lpKernelThreadContext =
        (__KERNEL_THREAD_CONTEXT*)lpEsp; //Save context.
    switch(lpCurrentThread->dwThreadStatus)
    {
        //Allow the thread with following status to continue to run.
        case KERNEL_THREAD_STATUS_READY:
        case KERNEL_THREAD_STATUS_SUSPENDED:
        case KERNEL_THREAD_STATUS_SLEEPING:
        case KERNEL_THREAD_STATUS_TERMINAL:
        case KERNEL_THREAD_STATUS_BLOCKED:
        {
            lpCurrentThread->dwTotalRunTime +=
                SYSTEM_TIME_SLICE; //Update time slice.
            __SwitchTo((__KERNEL_THREAD_CONTEXT*)lpEsp);
            return; //Should not reach here.
        }
        case KERNEL_THREAD_STATUS_RUNNING: //Should schedule.
        {
            lpThread = lpMgr->GetScheduleKernelThread(
                (__COMMON_OBJECT*)lpMgr,
                lpCurrentThread->dwThreadPriority);
            if(NULL == lpThread) //Current thread is most priority.
            {
                lpCurrentThread->dwTotalRunTime
SYSTEM_TIME_SLICE;
                __SwitchTo((__KERNEL_THREAD_CONTEXT*)lpEsp);
            }
        }
    }
}

```



放入挂起队列，然后从就绪队列中选择另外一个优先级最高的核心线程投入运行。

若在当前核心线程的状态刚刚被设置为 `SUSPENDED`，还没有放入挂起队列的时候，发生了中断，这样当前核心线程的状态就是 `KERNEL_THREAD_STATUS_SUSPENDED`。处于这种状态的核心线程，是一种临时状态，会在很短的时间内被换出 CPU。因此，若发生中断的时候，当前核心线程处于这种状态，则不作任何调度，而是恢复当前核心线程，继续让其执行（采取“放行”的策略）。因为在很短的时间内，又会发生一次线程调度。

### **`KERNEL_THREAD_STATUS_SLEEPING:`**

核心线程在调用 `Sleep` 函数，但还未完全进入睡眠状态的时候，会处于正在运行，但状态为 `SLEEPING` 的情况。因为 `Sleep` 函数会首先把当前核心线程的状态设置为 `SLEEPING`，然后插入睡眠队列，并从就绪队列中选择另外一个状态为 `KERNEL_THREAD_STATUS_READY` 的线程投入运行。

若核心线程的状态刚刚被设置为 `KERNEL_THREAD_STATUS_SLEEPING`，还没有来得及被插入睡眠队列，这时候发生中断，则当前线程就是 `SLEEPING` 状态。对处于这种状态的核心线程，调度程序也不会打断，而是恢复其上下文，继续让其执行。因为在很短的时间内，该线程就会被切换出 CPU。

### **`KERNEL_THREAD_STATUS_TERMINAL:`**

在核心线程结束的时候，会处于 `KERNEL_THREAD_STATUS_TERMINAL`。在核心线程结束运行的时候，首先会把自己的状态设置为 `KERNEL_THREAD_STATUS_TERMINAL`，然后试图从就绪队列中选择另外一个状态为 `READY` 的线程投入运行。若这个过程中有中断发生，则在中断处理程序看来，当前核心线程会处于 `TERMINAL` 状态。对于这种状态的核心线程，也采取放行策略。

### **`KERNEL_THREAD_STATUS_BLOCKED:`**

在核心线程等待一个核心对象的时候，会处于这种状态。核心线程调用 `WaitForThisObject` 或 `WaitForThisObjectEx` 函数，等待一个共享对象。在这些函数的处理中，会首先把当前核心线程的状态设置为 `KERNEL_THREAD_STATUS_BLOCKED`，然后把当前线程插入共享对象的等待队列。但若在插入等待队列前发生中断，则被中断的核心线程（当前核心线程）就会处于这种状态。

对于这种状态的核心线程，也是采取放行的策略。

### 4.5.3 线程的调度—程序上下文

在 Hello China V1.0 的实现中，在程序上下文中完成线程调度，是通过调用 ScheduleFromProc 函数来完成的，在 V1.5 的实现中，继续保留了该函数，但对于该函数的实现，做了一些变更，下面是 V1.5 中，实现的 ScheduleFromProc 函数：

```
static VOID ScheduleFromProc(__KERNEL_THREAD_CONTEXT* lpContext)
{
    __KERNEL_THREAD_OBJECT* lpCurrent = NULL;
    __KERNEL_THREAD_OBJECT* lpNew     = NULL;
    DWORD                    dwFlags;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    lpCurrent = KernelThreadManager.lpCurrentKernelThread;
    switch(lpCurrent->dwThreadStatus)
    {
        case KERNEL_THREAD_STATUS_RUNNING:
        {
            lpNew = KernelThreadManager.GetScheduleKernelThread(
                (__COMMON_OBJECT*)&KernelThreadManager,
                lpCurrent->dwThreadPriority); //Try to get a new one.
            if(NULL == lpNew) //Current one is the most priority.
            {
                lpCurrent->dwTotalRunTime += SYSTEM_TIME_SLICE;
                __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
                return; //Allow current thread to continue to run.
            }
            //If got a new kernel thread successfully,then should
schedule
                //the new one to run.
            else
            {
                lpCurrent->dwThreadStatus =
KERNEL_THREAD_STATUS_READY;
                KernelThreadManager.AddReadyKernelThread(
                    (__COMMON_OBJECT*)&KernelThreadManager,
                    lpCurrent); //Add to ready queue.
                lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
                lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
                KernelThreadManager.lpCurrentKernelThread = lpNew;
                __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
```

one.

```

        &lpNew->lpKernelThreadContext); //Switch to new

        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return;
    }
}
case KERNEL_THREAD_STATUS_READY:
{
    lpNew = KernelThreadManager.GetScheduleKernelThread(
        (__COMMON_OBJECT*)&KernelThreadManager,
        lpCurrent->dwThreadPriority);
    if(NULL == lpNew) //Should not occur.
    {
        BUG();
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return;
    }
    if(lpNew == lpCurrent) //The same one.
    {
        lpCurrent->dwTotalRunTime += SYSTEM_TIME_SLICE;
        lpCurrent->dwThreadStatus = KERNEL_THREAD_RUNNING;
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return; //Allow current continue to run.
    }
    else
    {
        lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
        lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
        KernelThreadManager.lpCurrentKernelThread = lpNew;
        __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
            &lpNew->lpKernelThreadContext);
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return;
    }
}
case KERNEL_THREAD_STATUS_BLOCKED:
case KERNEL_THREAD_STATUS_SLEEPING:
case KERNEL_THREAD_STATUS_TERMINAL:
{
    lpNew = KernelThreadManager.GetScheduleKernelThread(
        (__COMMON_OBJECT*)&KernelThreadManager,

```

```

        0); //Current thread must be swapped out.
    if(NULL == lpNew) //Should not occur.
    {
        BUG();
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return;
    }
    lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
    lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
    KernelThreadManager.lpCurrentKernelThread = lpNew;
    __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
        &lpNew->lpKernelThreadContext);
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return;
}
default: //Should not occur.
{
    BUG();
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return;
}
}
}
}

```

该函数可以在任何非中断上下文中被调用，用于完成核心线程的重调度。这样，该函数必须判断当前线程的状态，以确定进一步的动作。需要注意的是，当前线程的状态，不一定是 **RUNNING**，而很多情况下，都是非 **RUNNING** 的“临时”状态，比如，当前线程等待一个共享对象，而该共享对象又是不可使用的，于是当前线程就需要把自己插入共享对象的等待队列，然后把状态设置为 **BLOCKED**，并调用 **ScheduleFromProc** 重新调度，这样就出现了当前线程状态是 **BLOCKED** 状态的情况。

下列对各种临时状态进行解释，包括其发生的条件，以及采取的动作等。

### **KERNEL\_THREAD\_STATUS\_RUNNING:**

在当前核心线程调用 **WaitForThisObject** 等系统调用的时候，若试图等待的共享资源可用，则当前线程的状态不会被修改。但 **Hello China V1.5** 采用的是抢占式的调度方式，因此在任何系统调用中，会重新检查系统就绪队列，看是否存在比当前优先级更高的核心线程，即执行一个核心线程调度过程。

这种情况下，在调用 **ScheduleFromProc** 的时候，就会出现当前核心线程是 **RUNNING**

的情况。对于这种情况，ScheduleFromProc 做如下处理：

- 1、调用 GetScheduleKernelThread 函数，试图从就绪队列中选择一个可调度线程。在调用该函数的时候，会以当前核心线程的优先级作为参数，遮掩 GetScheduleKernelThread 会返回比当前核心线程优先级更高的核心线程，若没有，则返回 NULL；
- 2、若返回 NULL，说明当前就绪队列中没有核心线程比当前线程优先级更高，于是直接返回，这样可导致当前核心线程继续运行；
- 3、若能够找到一个比当前核心线程优先级更高的线程，则把当前核心线程状态修改为 READY，并放入就绪队列。然后增加刚刚获取的核心线程的运行时间片信息，修改其状态为 KERNEL\_THREAD\_STATUS\_RUNNING，并修改当前核心线程指针指向该线程，调用 \_\_SaveAndSwitch 函数，切换到该线程。这样当前核心线程就会被打断，从而“让路”给更高优先级的核心线程。

这种调度方式，可确保任何比当前核心线程优先级高的线程，能够在最快的时间内得到调度，从而提升系统的整体实时性。

## KERNEL\_THREAD\_STATUS\_READY:

在操作系统刚刚完成初始化，还没有选择任何核心线程运行的时候，当前核心线程会被设置为这种状态。在系统初始化的过程中，会创建 shell、IDLE 等系统核心线程。在初始化完成后，会把当前核心线程设置为创建的任何一个核心线程，不论设置为哪个核心线程，其状态都是 KERNEL\_THREAD\_STATUS\_READY。

系统初始化完成之后，会调用 ScheduleFromProc 函数，以切换到一个优先级最高的线程。实际上，系统初始化过程，是不属于任何核心线程的，但也可以看作是一个初始化核心线程。一旦初始化完成，切换到其它的核心线程，则这个“初始化核心线程”也就运行结束了。

这样初始化完成，调用 ScheduleFromProc 的时候，当前核心线程就是 READY 状态。针对这种状态，调度程序做如下处理：

- 1、调用 GetScheduleKernelThread 函数，从就绪队列中提取一个核心线程。在调用该函数的时候，会以当前核心线程的优先级为参数，这样就约束了 GetScheduleKernelThread 函数，只能返回大于或等于当前核心线程优先级的就绪线程；
- 2、若 GetScheduleKernelThread 返回 NULL，说明系统发生问题了。因为当前核心线程被创建的时候，一定是加入到就绪队列的，GetScheduleKernelThread 函数至少应该返回当前核心线程。若返回 NULL，则打印出调试信息（BUG()函数），并返回；
- 3、若返回的核心线程对象，跟当前核心线程是同一个，则说明当前核心线程就是系统

中优先级最高的，于是增加当前核心线程的时间片计数，并修改其状态为 `RUNNING`，直接返回。这样可导致当前核心线程继续执行；

- 4、若返回的核心线程对象不是当前核心线程对象，则增加新核心线程的时间片计数，修改其状态，并切换到该线程开始执行。

## **KERNEL\_THREAD\_STATUS\_SUSPENDED:**

若当前核心线程对象的状态为 `KERNEL_THREAD_STATUS_SUSPENDED`，则说明当前核心线程对象调用了 `SuspendKernelThread` 函数，试图挂起自己。`SuspendKernelThread` 函数在把当前核心线程设置为 `SUSPENDED` 状态之后，会把当前核心线程插入挂起队列，并调用 `ScheduleFromProc` 函数，重新调度线程。

若当前核心线程处于该状态，则调度程序执行下列动作：

- 1、调用 `GetScheduleKernelThread` 函数，试图从当前就绪队列中选择一个状态为就绪的核心线程。需要注意的是，这时候调用 `GetScheduleKernelThread` 函数，是以参数 0 作为第二个参数的，这样可导致该函数返回就绪队列中任何优先级大于或等于 0 的核心线程，即只要就绪队列中有核心线程对象存在，就会返回一个核心线程对象；
- 2、若上述函数返回 `NULL`，说明系统出现了问题。因为就绪队列中肯定会有核心线程存在的，至少有 `IDLE` 线程存在；
- 3、若上述调用返回了一个合法的核心线程对象，则修改返回的核心线程状态信息，增加其运行时间片计数，调用 `__SaveAndSwitch` 函数，保存当前核心线程的上下文信息，并切换到新的核心线程开始运行。

## **KERNEL\_THREAD\_STATUS\_SLEEPING:**

当前核心线程调用 `Sleep` 函数，试图睡眠的时候，会发生当前核心线程状态是 `SLEEPING` 的情况。因为 `Sleep` 函数首先把当前核心线程设置为 `KERNEL_THREAD_STATUS_SLEEPING` 状态，并插入睡眠队列，然后调用 `ScheduleFromProc` 函数。对于这种状态的核心线程，`ScheduleFromProc` 的处理机制，与当前核心线程状态为 `KERNEL_THREAD_STATUS_SUSPENDED` 的处理机制一样，请参考上面“`KERNEL_THREAD_STATUS_SUSPENDED`”一节获取详细信息。

## **KERNEL\_THREAD\_STATUS\_TERMINAL:**

线程运行结束的时候，会首先设置自己的状态为 `TERMINAL`，并调用 `ScheduleFromProc` 函数。`ScheduleFromProc` 函数对当前线程是该状态的处理动作，与上面“`KERNEL_THREAD_STATUS_SUSPENDED`”的处理机制一样，请参考上面一节获取详细信息。

## 核心线程的创建和初始化

相对于 Hello China V1.0, V1.5 的实现中, CreateKernelThread 函数也有所不同。V1.5 的实现代码如下, 其中黑体标注部分, 是与 V1.0 的实现有显著差别的地方。为了便于描述, 对每个黑体标注的差别部分, 采用 “//POSITION x” 进行标注, 以方便后面的描述:

```
static __KERNEL_THREAD_OBJECT* CreateKernelThread(
    __COMMON_OBJECT*          lpThis,
    DWORD                     dwStackSize,
    DWORD                     dwStatus,
    DWORD                     dwPriority,
    __KERNEL_THREAD_ROUTINE
lpStartRoutine,
    LPVOID
lpRoutineParam,
    LPVOID                    lpReserved,
    //POSITION 1
    LPSTR                     lp.szName)
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread    =
NULL;

    __KERNEL_THREAD_MANAGER*    lpMgr             = NULL;
    LPVOID                      lpStack           = NULL;
    BOOL                        bSuccess          = FALSE;
    DWORD*                      lpStackPtr        = NULL;
    DWORD                       i;

    if((NULL == lpThis) || (NULL == lpStartRoutine))
        goto __TERMINAL;

    if((KERNEL_THREAD_STATUS_READY != dwStatus) &&
        (KERNEL_THREAD_STATUS_SUSPENDED != dwStatus))
        goto __TERMINAL;

    lpMgr = (__KERNEL_THREAD_MANAGER*)lpThis;
    lpKernelThread
=
    (__KERNEL_THREAD_OBJECT*)ObjectManager.CreateObject(&ObjectManager,
        NULL,
        OBJECT_TYPE_KERNEL_THREAD);

    if(NULL == lpKernelThread)
```

```

        goto __TERMINAL;

    if(!lpKernelThread->Initialize((__COMMON_OBJECT*)lpKernelTh
read))
        goto __TERMINAL;

    if(0 == dwStackSize)
    {
        dwStackSize = DEFAULT_STACK_SIZE;
    }
    else
    {
        if(dwStackSize < MIN_STACK_SIZE)
        {
            dwStackSize = MIN_STACK_SIZE;
        }
    }
    lpStack = KMemAlloc(dwStackSize,KMEM_SIZE_TYPE_ANY);
    if(NULL == lpStack)    //Failed to create kernel thread stack.
    {
        goto __TERMINAL;
    }

    //The following code initializes the kernel thread object
    created just now.
    lpKernelThread->dwThreadID                =
lpKernelThread->dwObjectID;
    lpKernelThread->dwThreadStatus              = dwStatus;
    lpKernelThread->dwThreadPriority            = dwPriority;
    lpKernelThread->dwScheduleCounter          = dwPriority;    //*****
CAUTION!!! *****
    lpKernelThread->dwReturnValue               = 0L;
    lpKernelThread->dwTotalRunTime              = 0L;
    lpKernelThread->dwTotalMemSize              = 0L;

    lpKernelThread->bUsedMath                   = FALSE;
    lpKernelThread->dwStackSize                 = dwStackSize ?
dwStackSize : DEFAULT_STACK_SIZE;
    lpKernelThread->lpInitStackPointer          =
(LPVOID)((DWORD)lpStack + dwStackSize);
    lpKernelThread->KernelThreadRoutine        = lpStartRoutine;

```

```

lpKernelThread->lpRoutineParam      = lpRoutineParam;

lpKernelThread->ucMsgQueueHeader     = 0;
lpKernelThread->ucMsgQueueTail      = 0;
lpKernelThread->ucCurrentMsgNum      = 0;

lpKernelThread->dwLastError           = 0L;
lpKernelThread->dwWaitingStatus      = OBJECT_WAIT_WAITING;

//Copy kernel thread name.
//POSITION 2
if(lp.szName)
{
    for(i = 0; i < MAX_THREAD_NAME - 1; i++)
    {
        if(lp.szName[i] == 0) //End.
        {
            break;
        }
        lpKernelThread->KernelThreadName[i] = lp.szName[i];
    }
}
lpKernelThread->KernelThreadName[i] = 0; //Set string's
terminator.

//
//The following routine initializes the hardware context
//of the kernel thread.
//It's implementation depends on the hardware platform,so
//this routine is implemented in ARCH directory.
//
//POSITION 3
InitKernelThreadContext(lpKernelThread,KernelThreadWrapper)
;

if(KERNEL_THREAD_STATUS_READY == dwStatus)
{
    lpMgr->AddReadyKernelThread((__COMMON_OBJECT*)lpMgr,
        lpKernelThread);
}
else //Add into Suspended Queue.

```

```

    {
        if(!lpMgr->lpSuspendedQueue->InsertIntoQueue(
            (__COMMON_OBJECT*)lpMgr->lpSuspendedQueue,
            (__COMMON_OBJECT*)lpKernelThread,dwPriority))
            goto __TERMINAL;
    }

    //Call the create hook.
    //POSITION 4
    lpMgr->CallThreadHook(THREAD_HOOK_TYPE_CREATE,lpKernelThrea
d,
        NULL);
    bSuccess = TRUE;

__TERMINAL:
    if(!bSuccess)
    {
        //First,release the resources created successfully.
        if(NULL != lpKernelThread)

            ObjectManager.DestroyObject(&ObjectManager,(__COMMON_OBJECT*)lpKer
nelThread);

        if(NULL != lpStack)
            KMemFree(lpStack,KMEM_SIZE_TYPE_ANY,0L);
        return NULL;
    }
    else
        return lpKernelThread;
}

```

与 V1.0 相比，主要有三个不同的地方：

- 1、V1.5 的实现中，允许为核心线程指定一个名称（对应代码中 POSITION1 和 POSITION2）。这样可对核心线程进行有意义的标识，在 CPU 统计等一些系统输出中，也会同时输出核心线程的名字和 ID，这样使得每个核心线程的标识更加直观。当然，在 V1.5 的实现中，为核心线程对象的定义增加了一个字符串数组成员，用于存放核心线程的名字；
- 2、对核心线程上下文的初始化（对应代码中 POSITION3）。在 V1.0 的实现中，使用了一个宏 INIT\_KERNEL\_THREAD\_CONTEXT 完成对核心线程上下文对象的初始化。而在 V1.5 的实现中，由于硬件上下文直接保存在堆栈中，因此在创建线程过程中，其硬件上下文的初始化方式需要改变。在 V1.5 的实现中，只需要建立合适的堆栈框架即可，这个堆栈框架，应该是符合\_\_SwitchTo 函数要求的，因为线程刚创建完毕

后，并不会马上投入运行，而是插到就绪队列中，直到下一次调度时机（比如，中断或另外线程的系统调用）到来的时候，才会被调度。在 V1.5 的实现中，通过一个 `InitKernelThreadContext` 函数，完成了对核心线程上下文的初始化。详细的实现，请参考后文；

- 3、调用了线程创建回调函数（对应代码中 **POSITION4**）。线程回调函数是 V1.5 引入的一个机制，可在核心线程被创建、被销毁、被换出 CPU、被调入 CPU 的四个时刻得到调用，从而完成一些系统级的任务。

`InitKernelThreadContext` 函数初始化了一个核心线程的硬件上下文，该函数代码如下：

```
VOID InitKernelThreadContext(__KERNEL_THREAD_OBJECT* lpKernelThread,
                             __KERNEL_THREAD_WRAPPER lpStartAddr)
{
    DWORD*      lpStackPtr = NULL;
    DWORD       dwStackSize = 0;

    if((NULL == lpKernelThread) || (NULL == lpStartAddr))
    {
        return;
    }

    lpStackPtr = (DWORD*)lpKernelThread->lpInitStackPointer;

    __PUSH(lpStackPtr,lpKernelThread);        //Push lpKernelThread to
stack.
    __PUSH(lpStackPtr,NULL);        //Push a new return address,simulate
a call.
    __PUSH(lpStackPtr,INIT_EFLAGS_VALUE);    //Push EFlags.
    __PUSH(lpStackPtr,0x00000008);          //Push CS.
    __PUSH(lpStackPtr,lpStartAddr); //Push start address.
    __PUSH(lpStackPtr,0L);                  //Push eax.
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);
    __PUSH(lpStackPtr,0L);

    //Save context.
    lpKernelThread->lpKernelThreadContext =
        (__KERNEL_THREAD_CONTEXT*)lpStackPtr;
```

```
    return;  
}
```

其中，**PUSH** 是预定义的一个宏，如下：

```
#define __PUSH(stackptr, val) \  
do{ \  
    (DWORD*)(stackptr) -= 1; \  
    *((DWORD*)stackptr) = (DWORD)(val); \  
}while(0)
```

这个宏模拟了一个堆栈 **PUSH** 动作，首先把堆栈指针减去 1（实际上是减去四个字节），然后把 **val** 存放在栈顶。

**InitkernelThreadContext** 函数通过 **PUSH** 宏，建立了如下的堆栈框架：

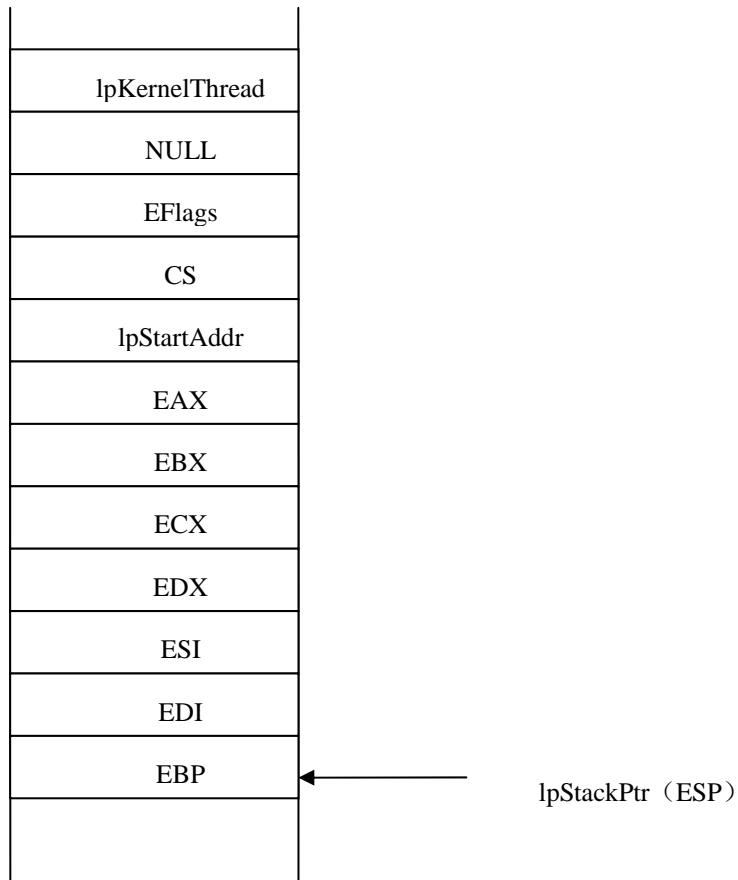


图 4-19 新创建的核心线程的堆栈框架

其中, `lpStartAddr` 就是 Hello China 提供的核心线程封装函数(`KernelThreadWrapper`)。堆栈框架建立完成之后, 就把 `lpStackPtr` 赋值给当前核心线程对象的 `lpKernelThreadContext` 变量。待该核心线程得到调度的时候, 通过 `lpKernelThreadContext` 变量, 就可得到上述堆栈框架的栈顶指针, 然后依次恢复通用寄存器, 并执行 `iretd` 指令, 就可跳转到 `lpStartAddr` 位置处 (即 `KernelThreadWrapper` 函数处, 这是所有核心线程的统一入口点)。

由于 `KernelThreadWrapper` 是一个函数, 接受一个核心线程对象作为其参数, 因此我们在开始的时候, 压入了当前核心线程对象的指针和一个 `NULL` 值, 以模拟一个 `CALL` 指令执行过程。

## 中断处理程序结束后的线程调度

在 V1.0 的实现中，在中断上下文中对线程的调度，是发生在时钟中断结束的时候，而在 V1.5 的实现中，对线程的调度，则发生在任何中断处理程序结束后。而 Hello China 的中断处理过程是这样的：

- 1、CPU 特定的汇编语言处理部分，主要是保存线程硬件上下文等；
- 2、汇编语言完成硬件上下文的保存后，调用 `GeneralIntHandler` 函数，该函数是 C 语言实现的，从该函数往后，所有的处理动作，都是由 C 语言完成（不考虑驱动程序内部采用的汇编语言处理）；
- 3、`GeneralIntHandler` 再调用 `system` 对象提供的 `DispatchInterrupt` 函数；
- 4、`DispatchInterrupt` 函数再调用系统中注册的中断处理程序。

在 V1.0 的实现中，对线程的调度，是发生在时钟中断处理程序之后、返回 `DispatchInterrupt` 函数之前的，因此只会在时钟中断中，对系统中的线程进行调度。在 V1.5 的实现中，把对线程的调度，放在了 `DispatchInterrupt` 函数中，该函数完成特定中断处理程序的调用后，将调用 `ScheduleFromInt` 函数，重新对线程进行调度。V1.5 的实现如下：

```
static VOID DispatchInterrupt(__COMMON_OBJECT* lpThis,
                             LPVOID          lpEsp,
                             UCHAR ucVector)
{
    __INTERRUPT_OBJECT*   lpIntObject = NULL;
    __SYSTEM*             lpSystem    = NULL;

    if((NULL == lpThis) || (NULL == lpEsp))
        return;

    lpSystem = (__SYSTEM*)lpThis;
    lpIntObject = lpSystem->lpInterruptVector[ucVector];

    //POSITION 1
    lpSystem->ucIntNestLevel += 1;    //Increment nesting level.

    if(NULL == lpIntObject) //The current interrupt vector has not
        handler object.
    {
```

```

        DefaultIntHandler(lpEsp,ucVector);
        return;
    }

    while(lpIntObject)    //Travel the whole interrupt list of this
vector.
    {
        if(lpIntObject->InterruptHandler(lpEsp,
            lpIntObject->lpHandlerParam))    //If an interrupt object
handles the interrupt,then returns.
        {
            break;
        }
        lpIntObject = lpIntObject->lpNextInterruptObject;
    }

    //POSITION 2
    lpSystem->ucIntNestLevel -= 1;
    if(0 == lpSystem->ucIntNestLevel) //The most outside interrupt.
    {
        KernelThreadManager.ScheduleFromInt(
            (__COMMON_OBJECT*)&KernelThreadManager,
            lpEsp); //Re-schedule kernel thread.
    }
    else
    {
        BUG();
    }
    return;
}

```

其中黑体标注部分，是与 Hello China V1.0 的实现不同的。在中断处理程序结束后，会调用 `ScheduleFromInt` 函数，完成一次线程调度。需要注意的是，在 V1.5 的实现中，对 `System` 对象也做了少量的修改，引入了一个 `ucIntNestLevel` 的成员变量，用于表示中断的嵌套级别。每当进入中断调度程序的时候，该变量会增加 1（代码中 **POSITION1** 位置处），每当完成一个中断处理程序的调用（退出中断处理程序）的时候，该变量减 1（对应代码中 **POSITION 2** 位置处）。若该变量值大于 1，说明发生了嵌套中断（即当前中断处理过程中，又被更高优先级的中断打断）。对于核心线程的调度，只有在最外层中断处理程序结束后，才会引发。这样是符合实际情况的，因为如果在不是最外层的中断处理

程序结束后调度线程，可能会导致外层中断无法得到处理，严重情况下会引起系统崩溃。

虽然引入了 `ucIntNestLevel` 变量标识中断嵌套级别，但 V1.5 版本的 Hello China 没有针对中断嵌套做特殊的优化，因此暂时可做“不支持嵌套中断”的处理。但实际上，经过测试，在中断嵌套的情况下，Hello China V1.5 的表现也是很好的☺。



## 第五章 内存管理机制

——Memory Management Mechanism of Hello  
China

## 5.1 内存管理机制概述

目前版本的 Hello China，其内存管理机制的实现，分成了两部分：

- 1、物理内存的管理，这部分主要实现了“纯粹”的物理内存的管理，不考虑任何基于硬件（比如，MMU）的内存管理机制，这部分的焦点，集中在几个重要的算法上；
- 2、虚拟内存管理，基于 Intel 32 位 CPU（本文中称为 IA32 结构）的内存管理机制，实现了一个分页的虚拟内存管理机制。

在本章中，我们首先讨论 IA32 CPU 的内存管理机制（硬件 MMU），在了解 IA32 内存管理机制的基础上，再详细介绍 Hello China 的物理内存管理方法和虚拟内存管理方法。需要说明的是，IA32 实现的内存管理机制，是十分典型的，其它类型或厂家的 CPU 的内存管理机制，跟 IA32 都有相通之处，至少一些概念是通用的，因此，掌握了 IA32 的内存管理机制，就可以很容易的通过阅读特定 CPU 的技术资料，掌握其它类型的 CPU 的内存管理机制。在本章中，为了进行比较，我们对 Power PC 的内存管理机制也进行一个简要的介绍。

## 5.2 IA32 CPU 内存管理机制

### 5.2.1 IA32 CPU 内存管理机制概述

IA32 的内存管理机制由两部分组成：分段和分页，其中，分段提供了一种机制，使得应用程序或操作系统的数据、代码、堆栈等，可以相互隔离，避免相互影响，在多任务（多进程）的情况下，每个进程都有自己特定的段，这样每个进程之间也不会相互影响。而分页机制则提供了按需内存分配、虚拟内存等机制，有了这些机制的支持，就可以实现应用程序的部分装入（应用程序执行映像只加载特定的部分到内存中，就可以开始执行）等功能。当然，分页机制也可以用于应用程序之间的隔离（或保护）。在 IA32 体系构架的 CPU 中，是否启用分页机制，是一个可选项，通过设置 CR0 寄存器（控制寄存器）的某一个比特，可以禁止或启用分页机制，而分段机制则不然，任何情况下都是启用的，没有一种方法可以禁止分段功能。

IA32 CPU 提供的这种内存管理机制十分灵活，最简单的情况下，采用平展段模式，禁止分页，可以实现最简单的跟物理内存一样的内存管理模型，最复杂的情况下，采用独立的段管理不同进程（或操作系统）的不同数据（代码、数据、堆栈等），采用分页机

制实现虚拟内存、按需内存分配等，可以实现最完整的程序保护，可以确保操作系统不受任何应用程序的影响，而应用程序之间，也相互不影响，而同一个应用程序内，采用段保护机制，也不会出现堆栈溢出、非法访问等异常情况。

下图很清楚的表示了这种内存管理机制：

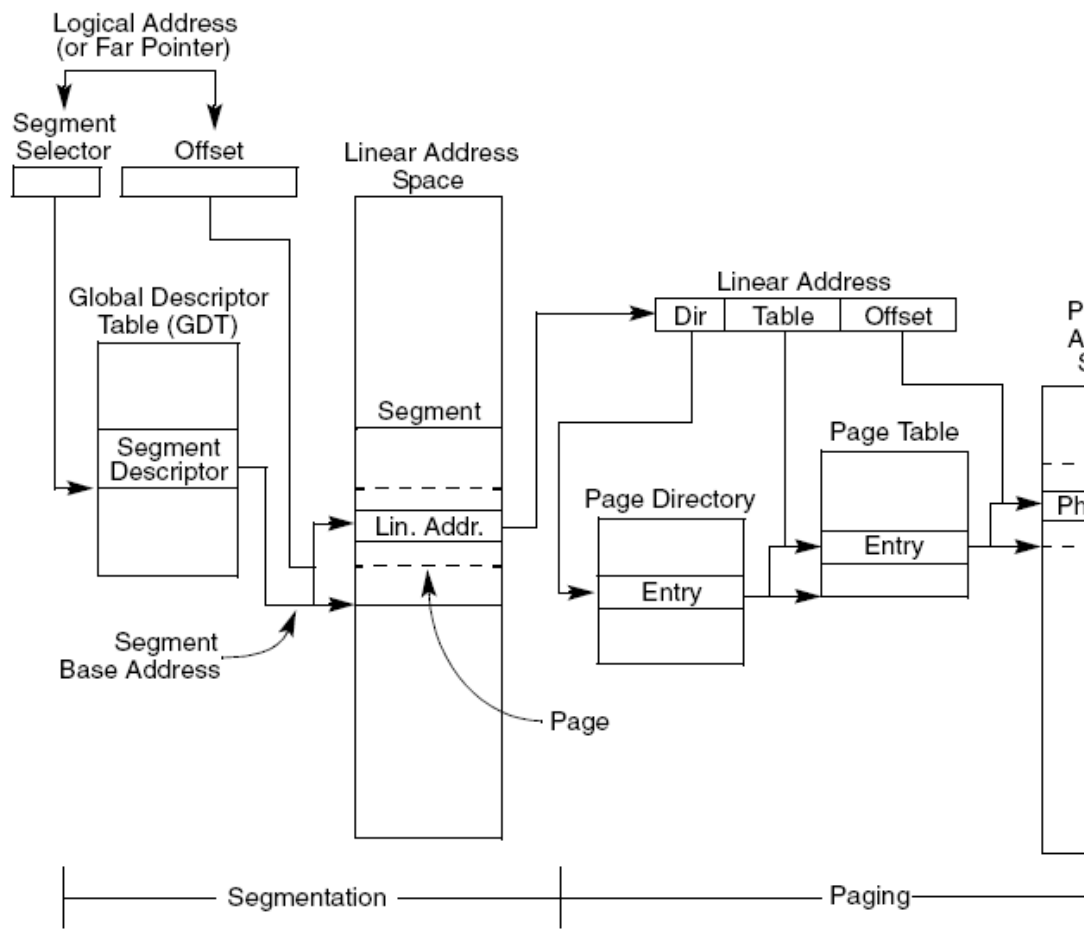


图 5-1 Intel CPU（IA32）内存管理机制

从图中可以看出，通过分段的方式，把 CPU 可寻址的整个地址空间（此处叫做线性地址空间）分成了若干部分，每部分对应一个段。可以看出，要描述每个段，需要知道

这个段在线性地址空间中的基地址（起始地址），以及该段的长度（界限），对于不同的段（比如代码段、数据段等），还有不同的访问方式（只读、读写等），所有这些数据存放在一个段描述表中（上图中对应的 GDT），段描述表的每一项（称为段描述符），描述了一个特定的段。可以看出，要访问一个段内的特定字节，需要给出两个数据：该段的描述符（用以确定段的基地址），以及该字节在段中的偏移（相对于段基地址），在 IA32 的实现中，段描述符表存放在物理内存中，整个段描述符表的初始地址，存放在一个特定的寄存器 GDTR 中，因此，在访问段描述符的时候，需要通过 GDTR 查找到描述符表对应的物理内存起始地址，然后再根据描述符在描述符表中的索引，定位到具体的段描述符的物理地址。段描述符在描述符表中的索引，就称为段选择子，这样，对于所有的段，由于 GDTR 是固定的，因此，给出一个段选择子，就可以准确定位到具体的段，也就是说，段描述符和段选择子是一一对应的，因此，要访问特定段内的一个字节，只需要给出一个段选择子和该字节在该段中的偏移位置即可。这两者的组合（段选择子和段内偏移），就称为逻辑地址。在 IA32 的段寄存器（CS、DS 等）中，存放的实际上就是段选择子（段描述符在描述符表中的索引）。下图示意了逻辑地址和线性地址的关系：

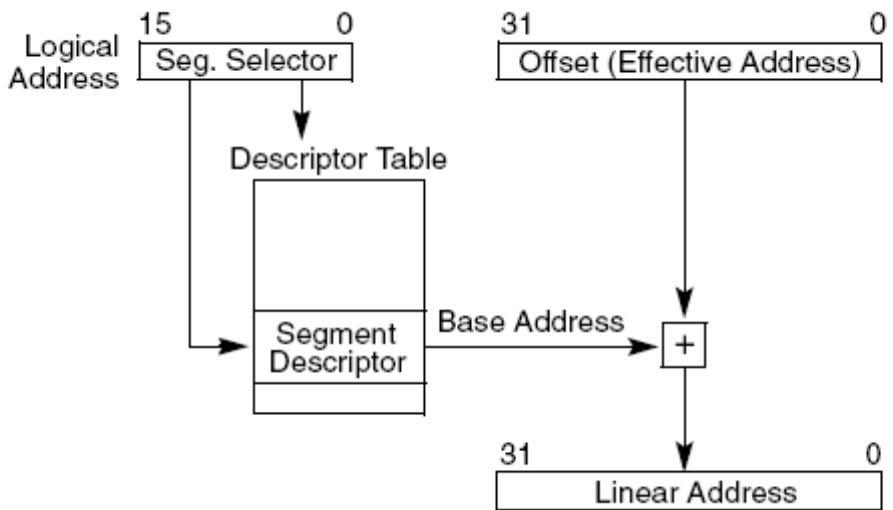


图 5-2 逻辑地址和线性地址的关系

给出一个逻辑地址后，CPU 就根据段选择子，查找到对应的段描述符，从段描述符中，获得段的基地址（在线性地址空间中），然后把基地址跟逻辑地址的字节偏移部分相

加，就获得一个线性地址。若没有启用分页机制，这个线性地址就可以直接映射到物理地址了。可以看出，这个过程是复杂的，需要多次访问物理内存，这样势必造成效率上的折扣。为了解决这个问题，IA32 构架的 CPU 采用段寄存器来存储段选择子，在访问段内的数据的时候，逻辑地址的选择子部分，直接从段寄存器中获取。按照当前的实现，代码段选择子从 CS 寄存器内获得，数据段选择子从 DS 寄存器内获得，堆栈段选择子从 SS 寄存器内获得。因此，在访问具体的数据的时候，需要根据数据所在的段，先把段选择子装入特定的段寄存器。为了进一步的提高效率，IA32 还实现了影子寄存器的机制，CS/DS 等段寄存器，还包含了程序员不可见的影子部分，影子部分存储了当前段的起始地址和段界限，这些数据在初始化 CS/DS 等段寄存器的时候，由 CPU 统一初始化，这样在访问段内的数据的时候，就不用再从物理内存中获取段描述符，然后再获得段基址了，而是直接从影子寄存器内获得段基址。这样的机制，实际上访问一个段内的一个字节，只通过一次内存访问操作就完成了，大大提高了效率。段寄存器和影子寄存器的关系如下：

Visible Part		Hidden Part	
Segment Selector		Base Address, Limit, Access Information	
			CS
			SS
			DS
			ES
			FS
			GS

图 5-3 段寄存器和影子寄存器的关系

5.2.2 几个重要的概念

在上面的介绍中，涉及到了逻辑地址等几个重要的概念，这几个概念在 IA32 的内存管理体系中，十分重要，在本节中，再次强调一下：

- 逻辑地址，段选择符和段内偏移一起组成逻辑地址，逻辑地址是 CPU 内的“第一层”地址，任何内存的访问，都是以逻辑地址的形式给出来的，比如，内存中的代码，其逻辑地址是通过 CS 寄存器存储的代码段选择符，跟 EIP 寄存器存储的指令指针（位置）共同组成的，CPU 在读取内存中的指令的时候，首先通过 CS 寄存器的影

子部分，获得段基址，然后跟 EIP 寄存器的指令偏移组合，形成逻辑地址，可以看出，逻辑地址是 48 位的（16 位的段选择符和 32 位的段内偏移）；

- 有效地址（Effective address），在 IA32 构架的内存管理机制中，把段内偏移称为有效地址，LEA 指令，操作的就是有效地址；
- 线性地址，段选择符跟段内偏移共同组成了逻辑地址，由段选择符可以唯一确定一个段描述符，进而确定一个特定的段，在段描述符内，存储了该段的基址，线性地址就是段基址跟段偏移相加，形成的地址。线性地址是 32 位的，线性地址跟逻辑地址的关系，并不是一一对应的关系，一个逻辑地址，唯一的对应一个线性地址，而一个线性地址，却可能对应多个逻辑地址。需要注意的是，线性地址是 CPU 内部的第二层地址，也可以理解为 CPU 的地址空间；
- 物理地址，物理地址就是 CPU 可以通过地址总线，直接寻址的地址。需要注意的是，线性地址并不是物理地址，在 CPU 根据逻辑地址，获得线性地址后，并不是根据线性地址，直接通过地址总线进行寻址的，而是把线性地址再次变换，变换成物理地址，然后通过地址总线寻址。这个线性地址到物理地址的变换，就是分页机制，这样，若在不启用分页机制的情况下，线性地址跟物理地址是一一对应的，即线性地址就是物理地址，但若启用了分页机制，则 CPU 根据线性地址，查找页目录和页表，获得物理地址，再通过地址总线进行寻址。

下图示意了上述各地址之间的关系：

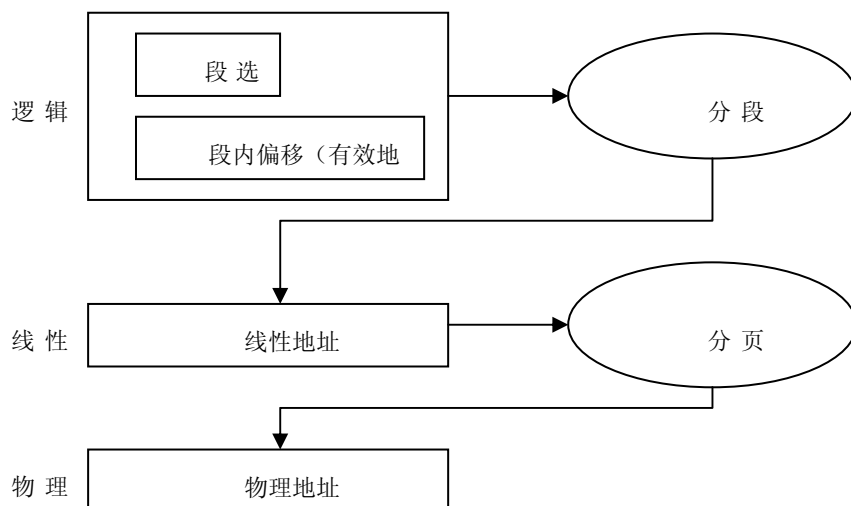


图 5-4 各地址概念之间的关系

还存在一种“虚拟地址”的概念，其实在 IA32 构架的 CPU 中，并没有引入该概念，但虚拟地址的概念，却经常出现，在本文中，我们也把线性地址叫做虚拟地址。

### 5.2.3 分段机制的应用

IA32 构架 CPU 提供的分段机制，应用十分灵活。从最简单的基本平展段模式，到复杂的多段模式，以及多段模式和分页机制的结合，都可以被操作系统采用，以完成不同的需求。在本节中，我们对不同的段模式进行介绍。

#### 5.2.3.1 基本平展段模式

所谓平展段模式，指的是系统中数据段、代码段、堆栈段等相互重叠，并且每个段都跟整个线性地址空间重叠。这样的段模式，使得应用程序和操作系统可以访问整个线性地址空间，而不用考虑段的存在。实际上，这种应用模式，把 IA32 CPU 的段机制屏蔽了。下图示意了这种基本的平展段模式：

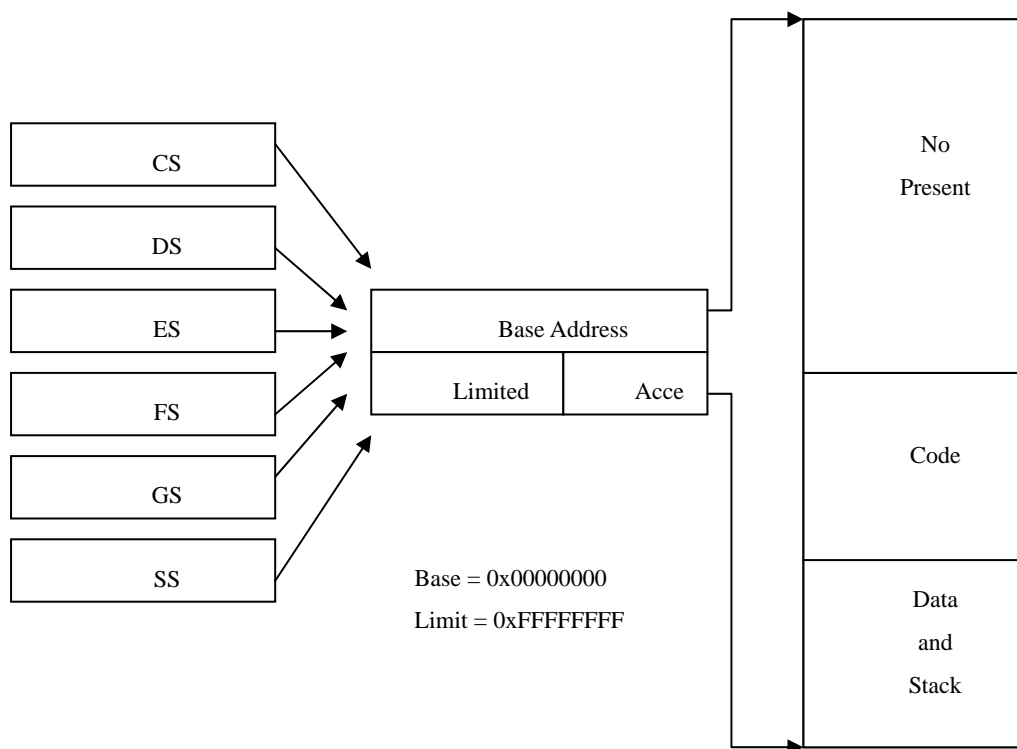


图 5-5 平展段应用模式

这种模式下，至少创建两个段描述符，一个为代码段描述符，该描述符的选择子，存放到 CS 寄存器，另外一个为数据段描述符，其选择子存放到 DS 和 SS 寄存器（堆栈段与数据段重合）。这两个数据段的访问方式、基址和界限都相同，唯一不同的是标志字段。

这种段模式，是一种最基本的段模式，但又是一种最通用的方式，因为按照这种方式实现的操作系统，可以很容易的移植到其它的 CPU，甚至是一些没有实现段机制的 CPU。目前版本的 Hello China，就是按照这种模式实现的。

实际上，许多流行的操作系统，都是按照这种方式实现的，只不过额外增加了两个段：

- 用户程序代码段，用来保存用户程序的代码；

- 用户程序数据段，用来保存用户程序的数据。  
所有的段的基址和界限，都是重叠的，覆盖了整个线性地址空间，为了实现保护功能，这些流行的操作系统采用了 IA32 的分页机制。

5.2.3.2 保护平展段模式

与基本平展段模式类似的，是保护平展段模式，在基本平展段模式中，每个段的界限（Limited）字段被设置为最大（0xFFFFFFFF），这样即使实际物理内存很小，在 CPU 出现内存访问溢出（访问的物理地址，比实际配置的物理地址大）的时候，仍然是可行的，不会引起异常。而保护平展段模式则不同，这种模式下，把段的界限和基址，根据需要，设置为合适的值，但整个系统中仍然存在两个段：代码段和数据段（堆栈段与数据段合一）。下图示意了这种结构：

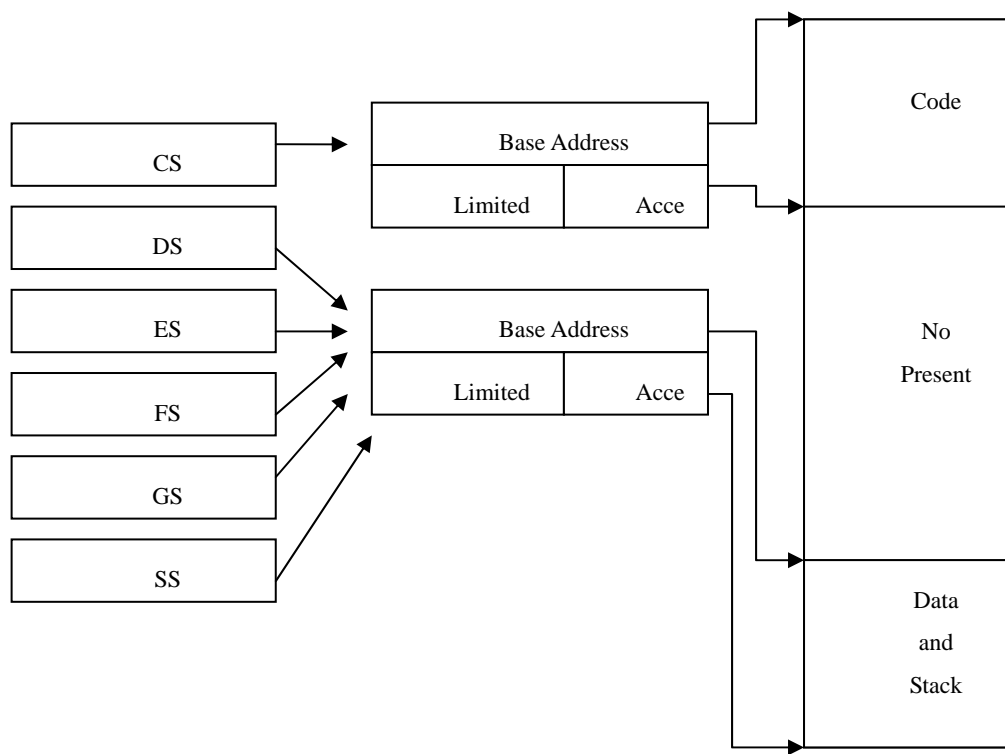


图 5-6 保护平展段应用模式

在这种段模型中，系统中也是有两个段：

- 代码段，该段的基地址设置为代码的实际开始地址，界限设置为代码段的长度；
- 数据段，该段的基地址设置为数据的实际开始地址（0x00000000），界限设置为数据和堆栈的长度的和，堆栈段和数据段合一。

这样，就可以避免两种错误情况的出现：

- 1、内存访问越界，对数据或代码的访问，超出了实际配置的物理内存，一旦出现这种情况，就会引起异常；
- 2、代码段被改写，因为数据段和代码段是不重合的，绝对不会出现代码段被改写的情况。

### 5.2.3.3 多段模式

多段模式是最复杂的一种模式，对于不同的数据结构，采用不同的段来表示。这种模式，最完整的应用了 IA32 CPU 提供的段机制，这样基于段机制的一些保护措施，可以很好的应用。这种段模式如下图：

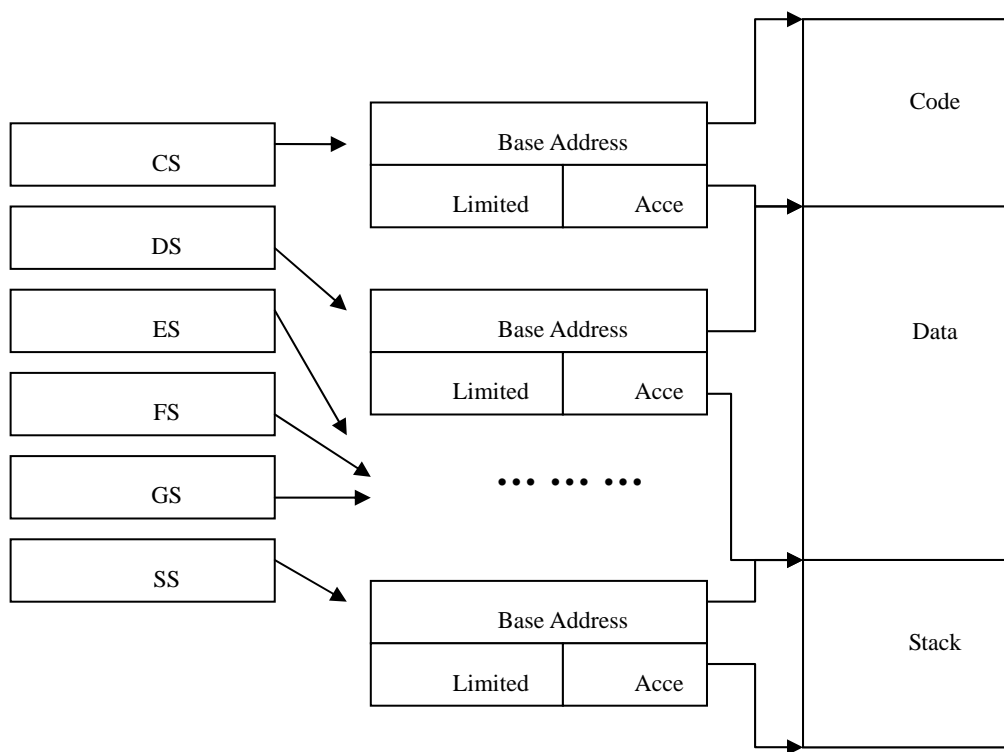


图 5-7 多段应用模式

系统中对不同的数据结构（数据、代码、堆栈等），设置了不同的段，每个段具有不同的基址和界限，分别跟实际的数据结构在内存中的位置对应，并把不同的段选择子装载到不同的段寄存器。

这种结构可以充分应用 CPU 提供的段保护基址，比如，这种情况下，不可能出现堆栈溢出的情况，因为一旦溢出，会引发异常。但这种结构也有一个缺点，就是 CPU 依赖性太强，按照这种模型实现的操作系统，很难移植到其它的与 IA32 段机制不同的 CPU 上。

### 5.2.4 分页机制的应用

分页机制是现代 CPU 的最基本的特征，基于分页机制，一些现代操作系统的功能才

能得以实现，比如虚拟内存、部分程序装入、按需内存分配、代码共享等。但分页机制也有一个缺陷，就是可能会导致效率下降，因为分页机制启用后，CPU 访问内存需要经过一系列的查表操作，而这些查表操作需要进一步从内存中读取数据。因此，分页机制一般应用在通用操作系统（比如，基于 PC 计算机的操作系统）中，而在实时的嵌入式操作系统中，一般很少使用。

IA32 CPU 实现了完善的分页机制，在 Hello China 的实现中，也实现了基于 IA32 CPU 的分页功能，提供简单的内存保护、高速缓存控制等功能。不过，目前版本 Hello China 实现的分页功能，是作为一个可选择模块实现的，在编译的时候，可以通过注释掉一个预定义选项，而取消分页功能。

但不管怎么说，深入理解分页机制及其应用，是深入理解现代操作系统的基础，在本章中，我们对 IA32 CPU 的分页机制进行描述，并列举几个典型的应用，这些应用，都是现代通用操作系统实现的最基本功能。

## 分页机制概述

IA32 CPU 根据逻辑地址，计算出线性地址，这个时候，若没有启用分页机制（通过 CR0 寄存器中的一个比特判断），则直接把线性地址映射到物理地址，也就是说，直接把线性地址送到物理地址总线上，以完成一个内存访问动作。但若启用了分页机制，则 CPU 就不会直接根据线性地址访问物理地址了，而是再根据线性地址，查找页目录和页表，最终获得物理地址。下图示意了根据线性地址查找物理地址的过程（算法）：

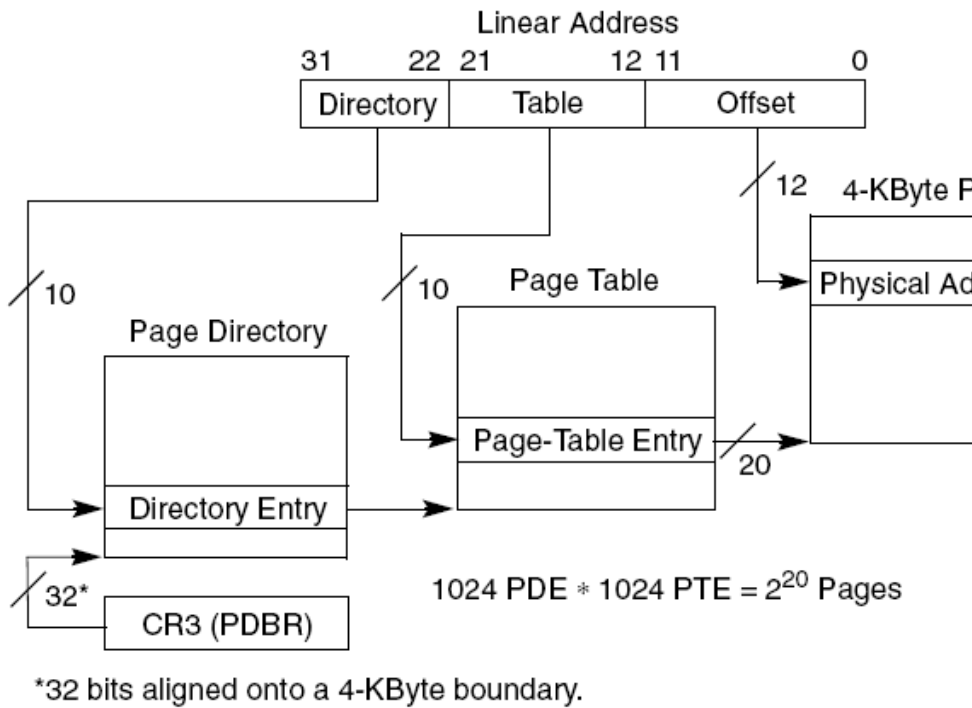


图 5-8 地址转换算法

- 1、CPU 根据 CR3 寄存器的值，获得页目录（第一级页表）所在的物理地址，从而获得页目录在内存中的位置；
- 2、CPU 根据线性地址的前 10 比特（22 到 31 比特）形成一个索引值，根据这个索引值，查找页目录，得到页表（二级页表）的位置（物理地址）；
- 3、CPU 再根据线性地址的中间 10 比特（12 到 21 比特），形成页表内索引，根据这个索引，查找步骤二获得的页表，从而获得页框的物理内存；
- 4、CPU 以线性地址的最后 12 比特（0 到 11 比特）为偏移，以步骤三获取的物理地址为基址，相加得到实际的物理地址。

可见，上述过程是较为复杂的，在最终获得正确的内存位置前，需要经过两次内存访问，两次查表操作，这显然是需要消耗时间的。为了提高效率，现代 CPU 都实现了一种 TLB（后备转换存储器）的机制，即把部分页表和页目录缓存到 CPU 的片上缓存中，在查找的时候，首先从 TLB 中查找，若查找失败，再启动内存查找，并更新 TLB。

下面是 IA32 CPU 的页目录、页表和页框的结构：

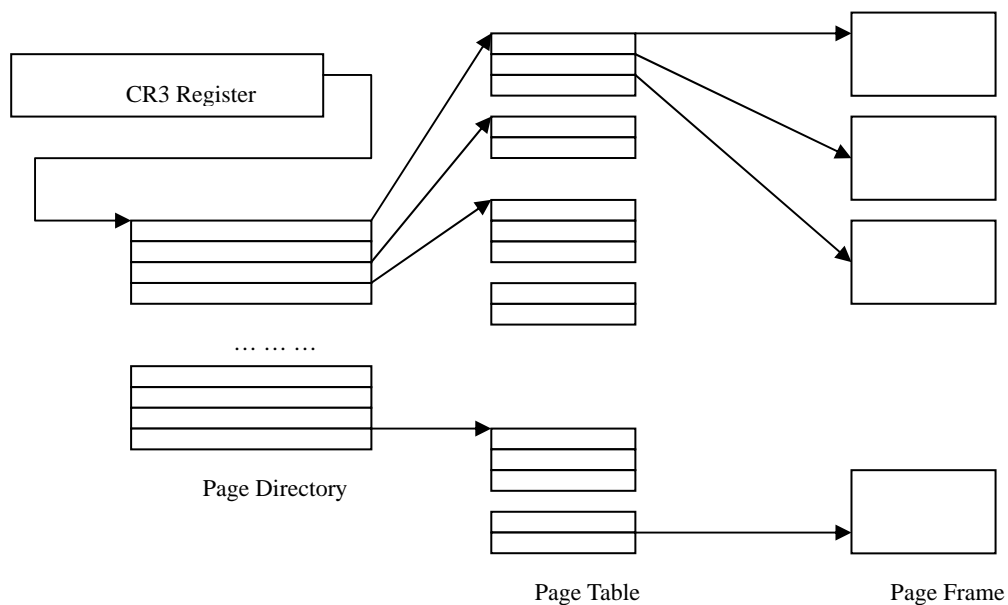


图 5-9 页目录、页表和页框的关系

其中，页目录是由页目录项组成的，每个页目录项是一个 32 比特的结构，如下：

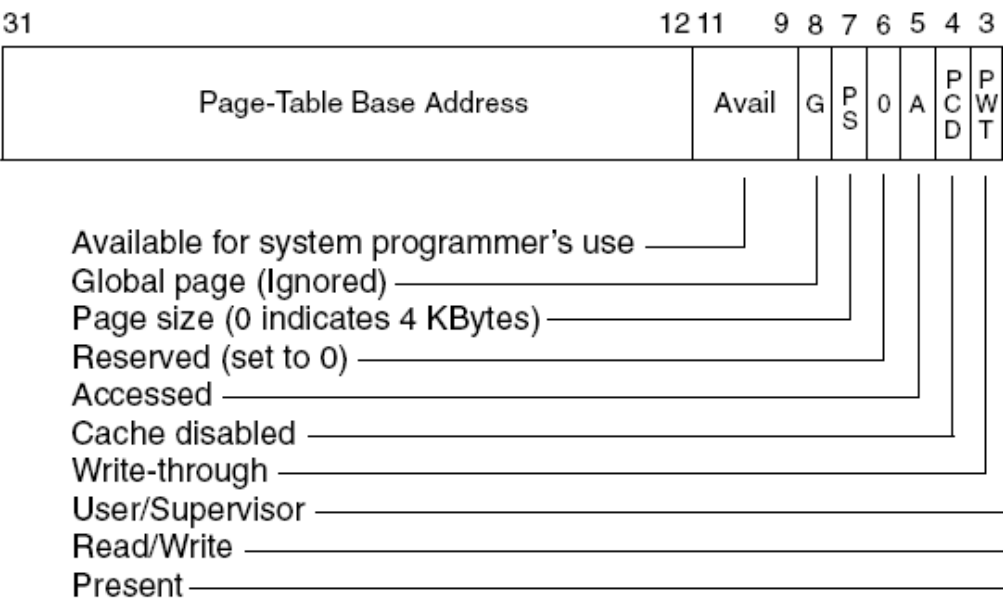


图 5-10 页目录项的组成

其中，页表基址部分，指出了跟该页目录项对应的页表的基地址，其它比特的含义，请参考下面的表格，详细的含义，请参考 Intel CPU 的用户编程手册：

比特	含义
P	存在标志，若该页目录项对应的页表存在于内存中，则设置该比特，否则设置为 0。
R/W	读/写标志，设置为 0，意味着对应的页表只读，否则为可读写
U/S	特权标志，设置为 1，任何特权都可访问，否则只有特权代码可访问。
PWT	Cache 控制标志
PCD	Cache 控制标志
A	访问标志，对应的页表一旦被访问，CPU 设置该标志
Reserved	保留，设置为 0
PS	页框大小标志，0 意味着页框大小为 4K，否则为 4M。
G	全局标志

Avail	供应用程序使用。
-------	----------

表 5-1 页目录项各比特的含义

这样，线性地址的前 10 比特是页目录的索引，因此可以最大确定 1024 个页目录项，每个页目录项的大小是 4byte，因此，整个页目录的大小是 4K。

同样地，页表也是由页表项组成的，页表项的结构如下：

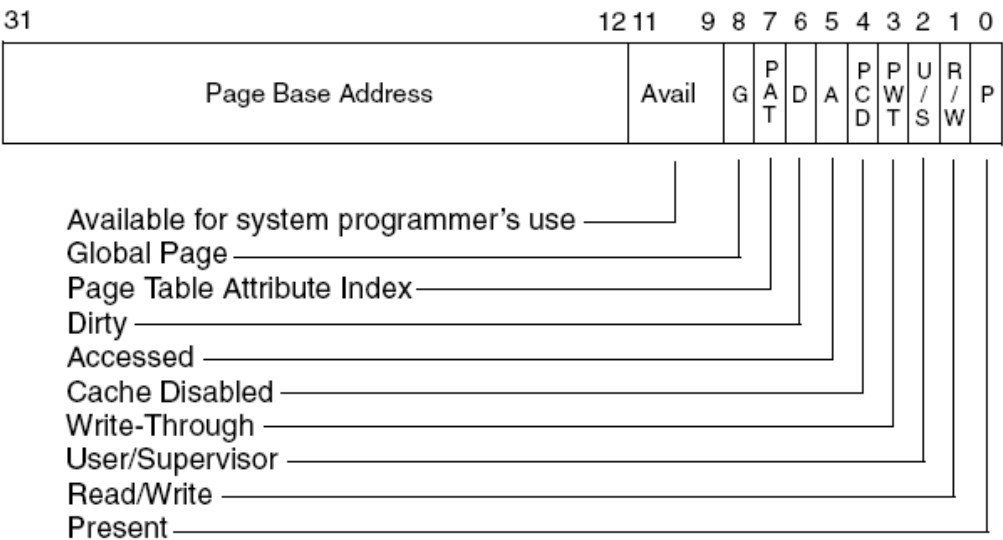


图 5-11 页表项的组成

其中，Page Base Address 给出了页框的物理地址（基地址），该字段是 20 比特，因此，每个页框是 4K 对齐的，且其大小是 4K，其它标志的含义，请参考下列表格：

比特	含义
P	存在标志，若该页目录项对应的页框存在于内存中，则设置该比特，否则设置为 0。
R/W	读/写标志，设置为 0，意味着对应的页框只读，否则为可读写

U/S	特权标志，设置为 1，任何特权都可访问，否则只有特权代码可访问。
PWT	Cache 控制标志
PCD	Cache 控制标志
A	访问标志，对应的或页框一旦被访问，CPU 设置该标志
D	修改标志，若对应的页框被修改，则 CPU 设置为 1
PS	页框大小标志，0 意味着页框大小为 4K，否则为 4M。
G	全局标志
Avail	供应用程序使用。

表 5-2 页表项各比特的含义

由于线性地址的中间 10 比特是页表项索引，因此，一个页表项的大小，也是 4K(1024 个页表项，每个页表项 4 字节)。这样，32 位线性地址空间，采用这种页目录和页表结构完整表示下来，需要内存的数量为：

$4\text{K}(\text{页目录}) + 1024 * 4\text{K} = 4100\text{K}$

其中，第一个 4K，是页目录所占用的空间，1024\*4K，则是所有页表占用的空间（一个页目录项对应一个页表）。但一般情况下，不需要把整个线性地址空间表示完，大多数情况下，表示其中的一部分，就可以满足应用需要了，因此，这种情况下，页目录和页表所占用的空间大大减少。

采用分页机制，可以完成线性空间内任意地址，跟物理地址空间内任意地址的映射，而且页表项（或页目录项）提供了页面的访问属性，通过这种映射关系，以及访问属性控制，可以实现很灵活的操作系统特性。在页表项和页目录项中，存在一个 P 标志，该标志指出了对应的页框或页表框是否存在（位于内存中）。当试图访问一个 P 标志为 0 的页表或页框时，将引发一个异常。除了访问 P 标志为 0 的页面，会引发异常以外，还有一些其它的组合情况，也会引发异常。下面的表格，列举了会引发异常的情况以及访问方式：

当前特权模式	页表特权模式	访问标志	访问方式	是否引发异常
Supervisor	User	Read-only	Write	是（注）
Supervisor	Supervisor	Read-only	Write	是

Supervisor	User	Read/Write	Write	否
Supervisor	Supervisor	Read/Write	Write	否
Supervisor	User	Read-only	Read	否
Supervisor	Supervisor	Read-only	Read	否
Supervisor	User	Read/Write	Read	否
Supervisor	Supervisor	Read/Write	Read	否
User	User	Read-only	Write	是
User	Supervisor	Read-only	Write	是
User	User	Read/Write	Write	否
User	Supervisor	Read/Write	Write	是
User	User	Read-only	Read	否
User	Supervisor	Read-only	Read	是
User	User	Read/Write	Read	否
User	Supervisor	Read/Write	Read	是

表 5-3            特权模式和访问模式之间的组合

上述情况中，没有考虑页表和页目录的对应标志不同的情况（在这种情况下会更加复杂，详细信息请参考 Intel CPU 的用户手册）。另，按照 Intel 的软件编程手册上的描述，不同类型的 CPU，其页面级保护方式也不一样，比如，有的 CPU，若当前模式是特权模式，也可以直接写入访问属性是 Read-Only、页面特权模式是用户的页面。但在本文的描述中，这些特殊情况不会带来影响。

通过分页机制，以及基于页面的保护机制，操作系统可以实现许多应用价值非常高的功能特性，比如内核保护、虚拟内存、按需内存分配、代码共享等，下面简单介绍了几个比较典型的机制。

操作系统核心的保护

采用 IA32 CPU 提供的页面级保护机制，可以实现操作系统核心代码的保护功能，即防止应用程序破坏操作系统核心代码或数据，从而导致整个系统故障。假设有一个操作系统，采用保护平展段模式，来使用 IA32 CPU 的分段机制，即整个系统设置四个段：

- 1. 操作系统核心代码段，特权级为 0（最高特权级），基址为 0x00000000，界限为 4G，即覆盖整个 CPU 的线性地址空间；

- 2. 操作系统核心数据段，特权级为 0，覆盖整个 CPU 的线性地址空间；
- 3. 用户程序代码段，特权级为 3（最低特权级），覆盖整个 CPU 的线性地址空间；
- 4. 用户程序数据段，特权级为 3，覆盖整个 CPU 的线性地址空间。

上述四个段彼此重合，但访问的特权级不同。操作系统在实现的时候，把自己的代码和数据，映射到线性地址的高端，比如 E0000000H—FFFFFFFFH，低端的 3G 线性地址空间，供应用程序使用。这样，操作系统需要为自己的这 1G 空间建立页表（实际上，只为实际装入代码和数据的内存，建立部分页表项即可），在建立页表的时候，把页表的访问特权级设置为 Supervisor。每当创建一个进程（加载一个应用程序），操作系统会根据应用程序的要求，把应用程序的代码、数据和堆栈，映射到低端的 3G 空间中，并对实际使用的线性地址，建立页表项，这时候，把页表项的特权级，设置为 User，最后，把操作系统对应的页表项，追加到应用程序页表上（不改变其特权级别），这样应用程序就可以“看到”操作系统的相关代码和数据了（但不能直接访问），如下图：

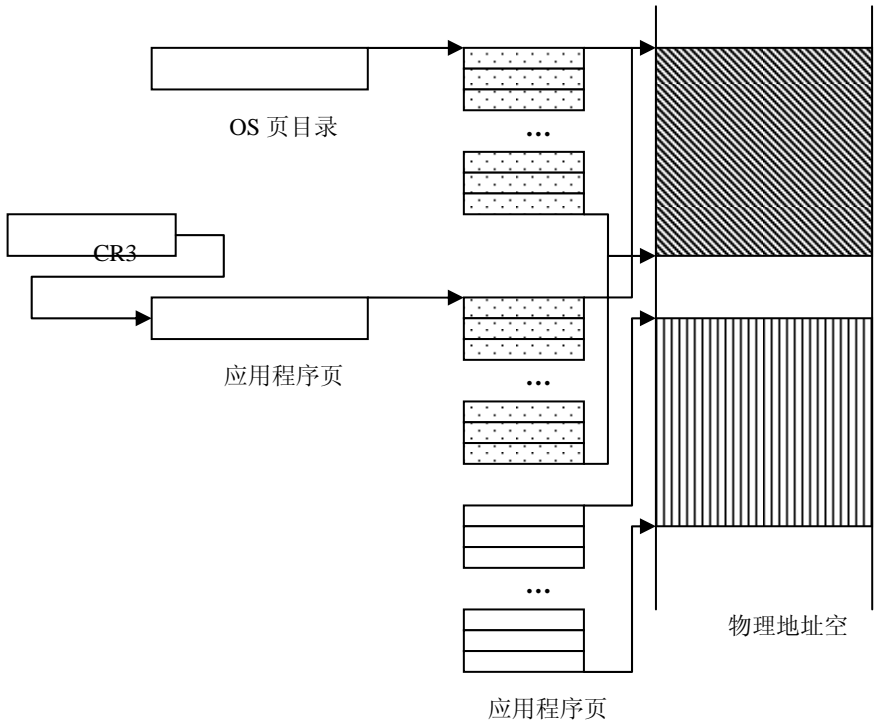


图 5-12 应用程序和操作系统地址空间

其中，应用程序的页表项，包含了操作系统的页表项，这样正常情况下，应用程序的运行，涉及到的线性地址空间，仅仅是应用程序所特有的（不是操作系统占用的线性地址空间），而且以特权级 3 来运行。一旦应用程序试图访问操作系统占用的线性空间，这样在线性地址转换为物理地址的时候，由于操作系统页表访问特权级是 0，而目前的特权级是 3，这样就会引发越权访问，导致一个页面失效（Page-Fault）异常，从而阻止应用程序对操作系统代码或数据的破坏。

当然，若应用程序要访问操作系统提供的服务，只能通过 IA32 CPU 提供的门机制，来提升当前的运行特权级（提升为 0），从而可以顺利访问操作系统的代码或数据。

在这个模型中，每个进程都有独立的地址空间（每创建一个进程，都需要创建对应的页目录和页表），但操作系统所占用的线性地址空间，却映射到所有应用程序的地址空间中的相同位置。在进程切换的时候，只需要把 CR3 寄存器（页目录基址寄存器）切换为新的进程，就实现了进程地址空间的切换。实际上，目前许多流行的操作系统，比如 Linux、MS Windows，都是按照这种方式实现的，或者与此方式类似。

## 虚拟内存的实现

现代计算机操作系统，一般都实现了虚拟内存功能，即把永久性存储介质（比如，硬盘）中的一部分空间开辟出来，虚拟成物理内存来使用，这样对应用程序来说，物理内存大大的增加了，这样可使得计算机系统，能够运行实际尺寸比物理内存大得多的应用程序。

虚拟内存的实现，也是建立在 CPU 的分页机制上的。在页表项中，有一个重要比特—P 比特，该比特指明了页表项对应的页框是否存在于物理内存中。若 P 比特为 0，则说明该页表项对应的页框不在内存中。这种情况下，若访问的线性地址刚好落到了这个不在物理内存中的页框内，则会引发一个缺页异常（page-fault）。在缺页异常处理程序中，操作系统会把该页表项对应的页框，从永久性存储介质上重新装入内存，并更改 P 标志为 1，然后从异常处理程序中返回（详细信息，请参考本书“中断和定时处理机制的实现”一章）。返回后，原先引起内存访问异常的指令，会再次执行。这时候，由于对应的页框已经被换回了内存，而且 P 标志被修改成了 1，这样就不会再次出现缺页异常了。

下面的图示，简单的说明了这种情况：

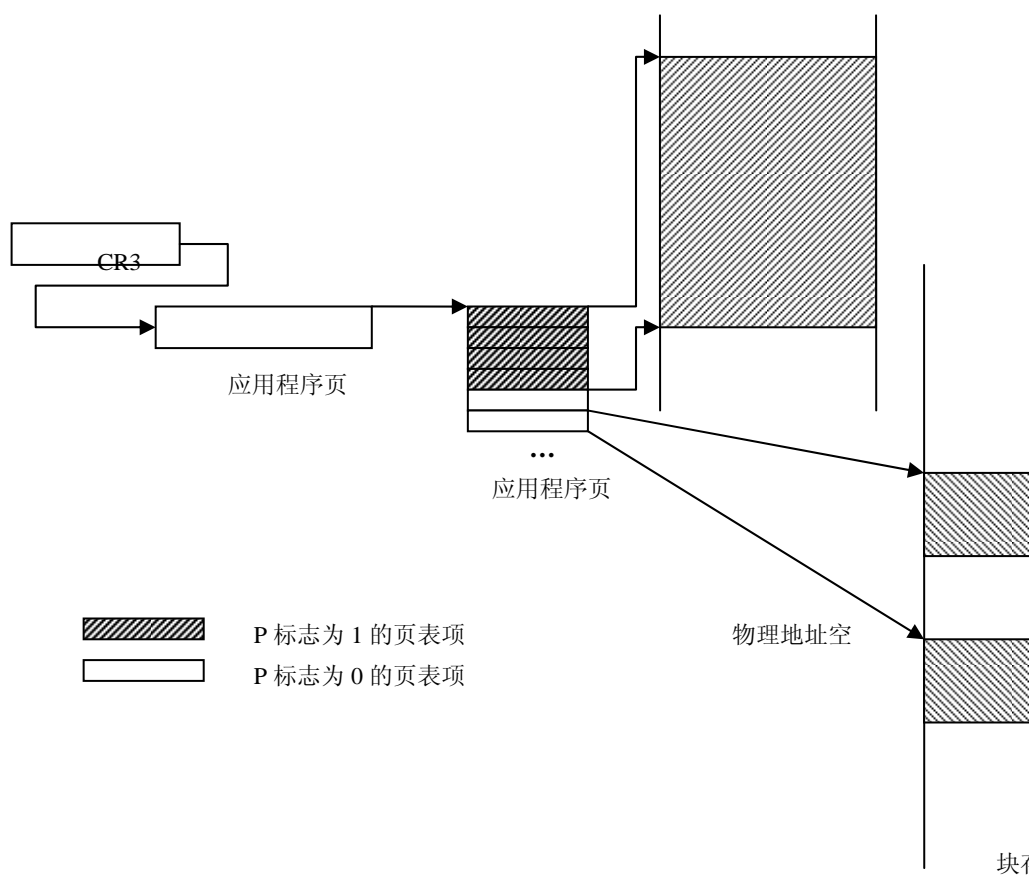


图 5-13 虚拟内存的实现机制

应用程序的页表项中，有的 P 比特为 1，该页表项跟唯一的物理内存页框对应，而有的 P 标志为 0，这种情况下，标明跟该页表项对应的物理内存框不在内存中，在虚拟内存的情况下，该页表项对应的页框位于块状存储介质中。这种情况下，页表项的前 31 个比特，可以用来指明该页框在物理介质上的具体位置（或者可以理解为一个文件的偏移，以页长度为单位），这样的情况下，若应用程序（代码或数据）访问了这种 P 标志为 0 的页框，会引发一个缺页异常，在缺页异常处理程序中，操作系统会重新分配一块物理内存页框，并把所缺的页，从磁盘中读会分配的物理页框，然后更改对应的页表项（若需要，还可能更改页目录项），所有这些操作完成之后，操作系统从异常处理程序中返回，

这样引起异常的程序得以继续执行。由于操作系统更改了页目录项，因此不会再次引起异常。

需要补充的是，在页表项中，若 P 标志为 0，则 CPU 对页表项的其它 31 比特是不作定义的，这样，剩余的 31 比特，可以用来存储对应的页框，在页面文件（虚拟内存存在磁盘上对应的文件）中的位置。但是，P 标志为 0，有的时候并不代表该页面被换出了内存，还有一种情况是，该页面尚未分配具体的物理内存（参考下面按需内存分配）。这种情况下，可以对剩余的 31 比特全部设置为 0，来表示这种情况，这样，为了区分这两种情况（按需内存分配和虚拟内存），在虚拟内存的情况下，页表项的剩余 31 比特，必须不能全为 0，这样的—个结果就是，页面文件的第—个页面大小的块，将不被使用。

### 按需内存分配

按需内存分配是分页机制的另外一个应用，其作用是尽可能的把应用程序对内存的需求，延迟到必须的时候才分配，以尽可能的提高内存利用率，尤其是应用程序申请内存的数量较大的时候。比如，—个应用程序，申请了 1M 的内存，—般情况下，应用程序不可能—下子把 1M 内存都使用完，这样为了提高内存使用效率，操作系统会给应用程序返回—个分配成功的信息，但实际上，只为应用程序分配有限数量的内存（比如，几个页框），但应用程序请求的内存所对应的页表项，都已经创建，不过页表项的 P 比特设置为 0（除了已经分配的—些页框）。

这样若应用程序访问了 P 标志为 0 的页表项，会导致—个缺页异常，在缺页异常中，操作系统会重新为应用程序分配内存，并更新页表项（需要的时候，—步更新页目录项）。这个过程跟虚拟内存类似，为了区别这两种情况，采用了—个特殊的标识方法—页表项除了 P 标志之外的 31 比特全部为零，则表示页表项对应—个未分配页框，否则对应虚拟内存的情况。

### 代码共享机制

应用程序之间可以通过分页机制，共享相同的代码，这样可使得共享代码在内存中仅仅保留—份拷贝，不用为每个应用程序进行单独加载。因此可减少内存空间的使用，提高内存使用效率。具体实现机制非常简单，即把共享的代码，映射到共享它的应用程序地址空间的相同位置（通过页表机制，可以实现—点）。

### 部分装入机制

采用分页机制，可以实现一种叫做“部分应用程序装入”的操作系统功能，用来运行尺寸比物理内存大得多的应用程序。为了实现这种功能，操作系统在装载应用程序的时候，会根据应用程序的代码段、数据段以及堆栈的需求，建立全部的页表项（以及页目录项），但只把应用程序代码和数据的前面很少部分（比如，几个页面）装入物理内存，其它剩余部分，仍然保留在硬盘上，装入内存的页面所对应的页表项和页目录项的 **P** 标志，设置为 1，其它的设置为 0，这样应用程序就可以从开始位置运行了。运行过程中，若指令位置或数据位置超出了装入内存的部分，则会引发一个缺页异常，从而导致操作系统把程序的另外一部分代码和数据，再装入内存，这样可使得应用程序继续得以执行。

## Power PC CPU 的内存管理机制

在嵌入式开发领域，Power PC、ARM 等功耗相对较低的 CPU，应用得比较广泛，在此简单介绍一下 Power PC 的内存管理机制，作为对 Intel CPU 内存管理机制内容的一个补充。

下图示意了 PPC CPU 的内存管理机制：

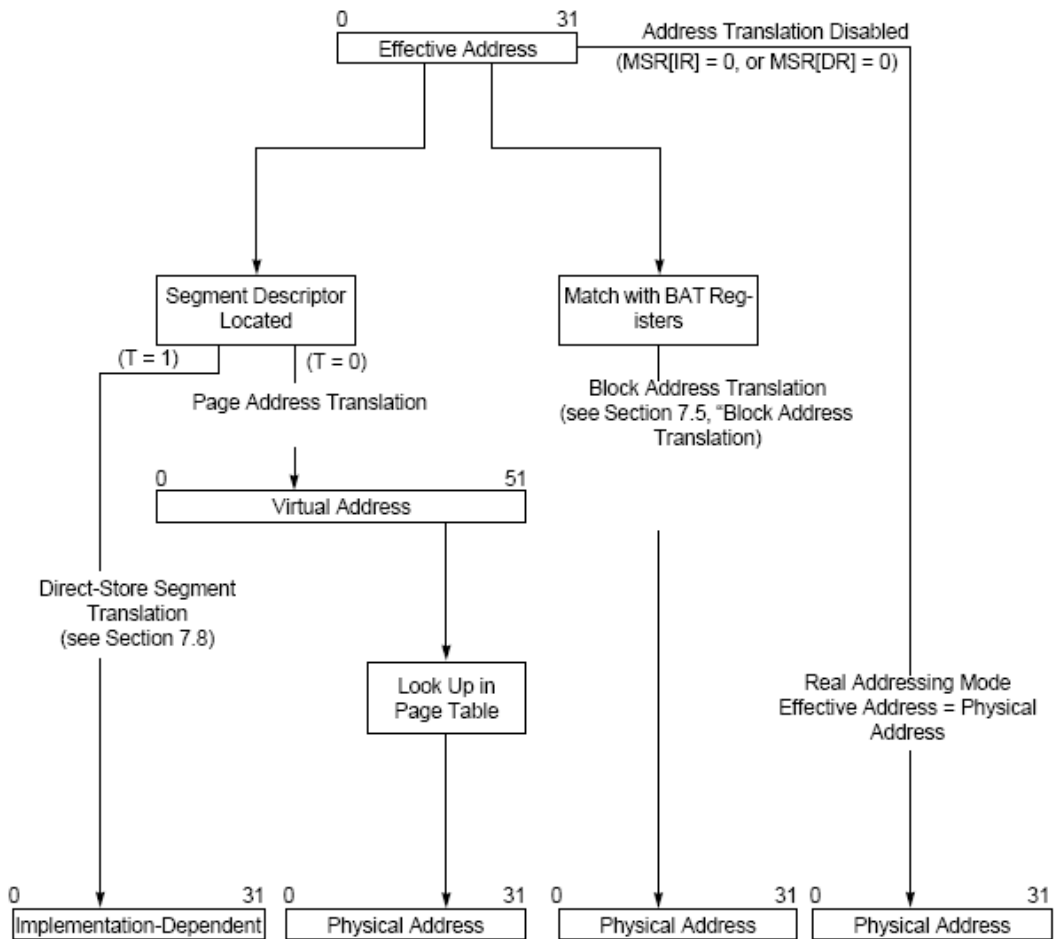


图 5-14 Power PC 的内存转换机制

在 PPC 的指令系统中，最初的地址叫做 Effective address，在 32 位 CPU 中，Effective address 是 32 位的，实际上，Effective 跟 IA32 内存管理机制中的线性地址类似，在 PPC 中，所有的编程层面涉及的地址，包括对指令和数据的寻址，都是以有效地址进行的。

在 PPC 的实现中，可以把内存分成 4K 大小的页面，以分页机制进行管理，还可以把物理内存分成长度可变化（但最小长度不能低于 128K）的块（block），以块为单位进行管理。在以页为基础的内存管理机制中，进一步把页组织成段，整个地址空间被分割成了 256M 大小的 16 个段，分别对应 16 个段描述符，与 IA32 的一个不同点是，PPC 的

段数量是固定的，整个系统就是 16 个。以块（Block）为基础的管理机制中，对每个块，有一个 BAT 与之对应，系统中的 BAT 组成一个寄存器数组。在进行内存寻址的时候，CPU 会对一个 Effective address 同时进行 BAT 匹配和段匹配（以 Effective address 的高 4 比特来索引一个段描述符），其中 BAT 优先级更高，即若 Effective 地址匹配 BAT 成功，则对段的匹配将被忽略，否则使用段匹配结果进行寻址。

我们重点介绍一下 PPC 的段页管理机制，在 CPU 寻址的过程中，根据 Effective 地址的高四个比特，定位到一个段描述符之后，会根据段描述符中的一个特殊的标志（T 标志），来确定进一步的动作。若 T 标志为 0，则进入分页机制的处理，根据段描述符以及 Effective 地址的剩余比特，形成一个虚拟地址（虚拟地址的长度是 52 比特），然后根据虚拟地址，查找段表，进而获得物理地址。若 T 标志为 1，则进入一种叫做 Direct-Store 的处理程序，这种处理程序是老式 PPC 处理器上的一种加快设备访问的机制，在新的 Power PC CPU 中，将会被淘汰，因此不用太关注。下图示意了 PPC CPU 根据 Effective 地址，获取物理地址的过程：

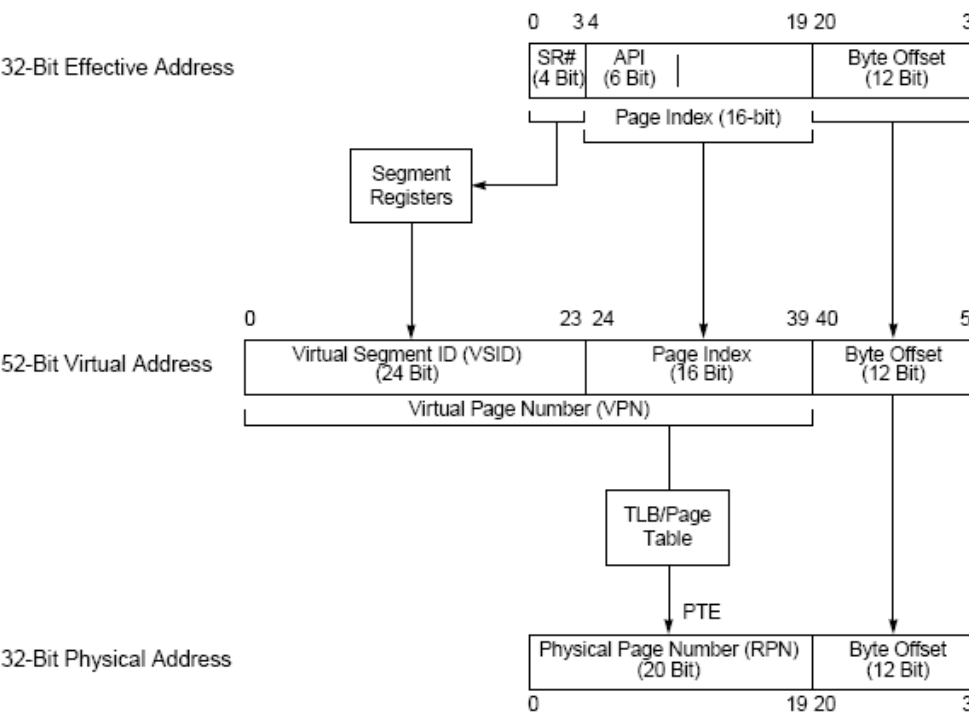


图 5-15 Power PC 的段页管理机制

这样，在查找段表的过程中，会把 52 比特的虚拟地址中的 VPN（虚拟页面号，实际上是 VPN 字段中的一部分比特），通过一个特定的 HASH 算法，获得目标页表项在页表中的物理地址，进而分析页表项，从中获得物理地址。既然是 HASH 算法，就可能会出现冲突，在不冲突的情况下，一次就可以获得正确的页表项，若出现了冲突，则进一步采用冲突处理算法，依次比较冲突的目标项，从中选择匹配的一项。通过合理的设计页表组织结构和大小（由操作系统完成），可以大大的提高命中概率。

需要注意的是，与 IA32 CPU 一样，PPC CPU 也有一种实地址模式（Read address mode），在这种模式下，PPC 的有效地址，直接映射为物理地址，与 IA32 的实地址模式不同的是，IA32 的实地址模式，其实是 16 位模式。

## Hello China 内存管理模型

### Hello China 的内存管理模型

按照目前 Hello China 版本的实现，在 Intel 32 位 CPU 上，采用的是最简单的平展模式，即数据段、代码段和堆栈段相互重叠，覆盖 CPU 的整个线性地址空间，下面的汇编代码，是代码断、数据段和堆栈段的段描述符的定义：

```
gl_sysgdt:                                ;;The start address of GDT.
                                           ;;In order to load the mini-kernal,the sys-
                                           ;;tem loader program,such as sysldrd.com(f-
                                           ;;or DOS) or sysldrb(for DISK),have initia-
                                           ;;lized the GDT,and make the code segment
                                           ;;and data segment can address the whole 32
                                           ;;bits linear address.
                                           ;;After the mini-kernal loaded,the control
                                           ;;transform to the OS kernal,so the kernal
                                           ;;will initialize the GDT again,this initi-
                                           ;;alization will make the GDT much proper.

gl_gdt_null    dd 0    ;;The first entry of GDT must be NULL.
               dd 0
```

```
gl_gdt_syscode                ;;The system code segment's GDT entry.
                                dw 0xFFFF
                                dw 0x0000
                                db 0x00
                                dw 0xCF9B
                                db 0x00

gl_gdt_sysdata                ;;The system data segment's GDT entry.
                                dw 0xFFFF
                                dw 0x0000
                                db 0x00
                                dw 0xCF93
                                db 0x00

gl_gdt_sysstack               ;;The system stack segment's GDT entry.
                                dw 0xFFFF
                                dw 0x0000    ;;The stack's base address is
                                                ;;0x01000000
                                db 0x00
                                dw 0xCF93
                                db 0x00
```

结合 IA32 CPU 的段描述符的定义，可以看出，目前 Hello China 实现的段属性如下：

段名称	起始线性地址	结束线性地址	访问属性
代码段	0x00000000	0xFFFFFFFF	读、写、执行
数据段	0x00000000	0xFFFFFFFF	读、写
堆栈段	0x00000000	0xFFFFFFFF	读、写

表 5-4 Hello China 的段属性

这样的实现，实际上是一种最简单、最通用的实现，相当于是忽略了 IA32 的段机制。

在嵌入式开发中，这种情况最为常见，因此，按照这种模型实现的操作系统，可移植性要高一些。另外，按照这种模型实现，符合 C 语言的内存管理模型，因为按照 C 语言的标准，一个指针应该能够寻址地址空间中的任何对象，若采用不重合的段模型，则可能会出现问題。比如，有下列两个函数：

```
VOID Function1(DWORD* lpdwResult,DWORD dw1,DWORD dw2)
{
    *lpdwResult = dw1 + dw2;
}

VOID Function2()
{
    DWORD dw1 = 100;
    DWORD dw2 = 200;
    DWORD dwResult = 0;

    Function1(&dwResult,dw1,dw2);
}
```

在第二个函数中，dwResult 的位置，实际上是在堆栈段里面的，这样在调用第一个函数（Function1）的时候，传递过去的参数（&dwResult），实际上是堆栈段的一个地址（偏移），但是在第一个函数中，在引用 lpdwResult 的时候，缺省情况下是按照数据段内的地址来引用的。这样若堆栈段和数据段不重叠，就不会引用到正确的位置，从而导致执行结果不正确，严重的时候，还会引起系统崩溃。只所以产生这个问题，是因为一般的编译器在实现的时候，对于指针参数的传递，只传递了段偏移部分，没有传递段选择子。

在当前的实现中，Hello China 没有实现进程，只实现了线程，而且在实现的时候，所用的线程，以及操作系统核心代码和数据，共享统一的线性空间。这样的实现方式，也是大多数嵌入式操作系统实现的方式，这种实现方式，效率会比进程模型高，因为在线程切换的时候，没有必要切换段寄存器（这会引引起整个 CPU Cache 的刷新），只需要完成堆栈、通用寄存器的切换即可。但也有一个弊端，就是保护功能稍微若一些，一个线程的崩溃，可能会导致整个系统的崩溃。

虽然没有充分采用 IA32 CPU 的分段机制，但目前 Hello China 的实现，却充分采用了 CPU 的分页机制，用来完成内存保护功能。通过分页功能，可以很容易的把线性地址空间内的内存地址，映射到物理内存当中，而且还可以实现按需内存分配功能，对于内存资源的充分利用，提供了保证。

在下面的章节中，我们首先对操作系统启动后的内存布局进行描述，然后对系统中的下列两个物理内存区域进行描述：

- 1、核心内存池，供操作系统和设备驱动程序使用，进一步分成 4K 区（以 4K 为单位进行分配）和任意尺寸区（以任何尺寸进行分配）；
- 2、分页管理区，供应用程序使用，以分页的方式进行管理。

上述两个区域都是物理内存区域，在完成上面两个区域的管理方式的描述后，将详细介绍 Hello China 的虚拟内存实现方法。在当前的版本中，虚拟内存的实现，是建立在 CPU 的分页机制上的。

## Hello China 的内存布局

按照目前的实现，Hello China 启动完成后，内存布局如下：

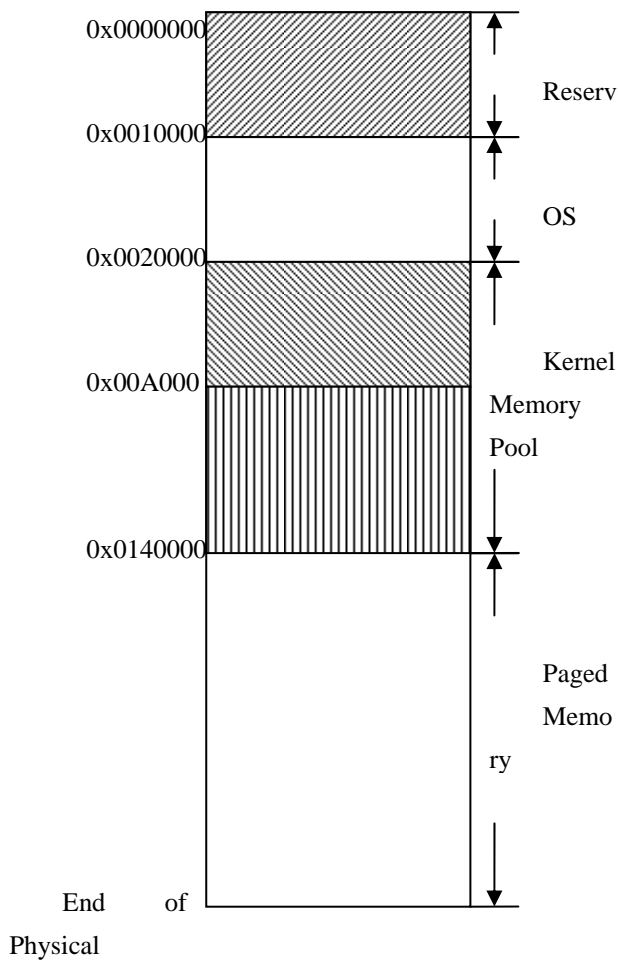


图 5-16 Hello China 的内存布局

对上述内存布局，描述如下：

范围	大小	用途
0x00000000 – 0x000FFFFF	1M	预留
0x00100000 –	1M	操作系统核心代码、静态数据等

0x001FFFFFFF		
0x00200000 – 0x009FFFFFFF	8M	操作系统核心内存池，以 4K 为单位进行分配
0x00A00000 – 0x013FFFFFFF	10M	操作系统核心内存池，以任何尺寸进行分配
0x01400000 – END	—	分页管理区

表 5-5      Hello China 的内存布局

其中，核心内存池用于操作系统和设备驱动程序使用，比如，操作系统运行过程中创建的核心对象（同步对象、核心线程对象等），都从核心内存池中进行分配。核心内存池又进一步分成两部分，一部分以 4K（页面大小）为大小进行分配，适用于系统中内存需求比较大的场合，比如驱动程序的缓存等，另一部分以任何大小的尺寸（不能大于该区域大小）进行分配。而分页管理区则采用分页机制进行管理，即按照页面大小（4K）为单位进行分配、回收，一般情况下，应用程序所需要的内存，从这一部分物理内存中分配。

核心内存池的管理

在 Hello China 当前的实现中，核心内存池又进一步分成了两部分：

- 1、4K 区域，以 4K 为单位进行分配和回收的区域，这部分内存池，一般供驱动程序采用，用来当作设备的数据缓冲区；
- 2、任意尺寸区域，以任意尺寸进行分配（实际上，在当前的实现中，最小的分配单位是 16 字节），供操作系统核心和驱动程序使用。操作系统在运行过程中创建的核心对象，比如核心线程对象、同步对象等，都是从该区域内分配内存。

对于任意尺寸的内存区域，采用空闲链表的方式来进行管理。空闲链表算法简单描述如下：

- 1、系统维护一个空闲链表，连接所有的空闲内存块。开始的时候，整个核心内存区域作为一个空闲块连接到空闲链表中；
- 2、每当有一个内存分配申请到达，内存管理函数遍历空闲链表，寻找一块空闲内存，该内存的大小大于请求的内存（或等于）；

- 3、如果不能找到，则返回空指针（NULL）；
- 4、如果找到，判断寻找到的内存的大小，如果跟请求的内存大小一致，或比请求的内存大少许（比如，16 字节），那么内存管理函数就把整个内存块返回给用户，然后把该空闲块从内存中删除；
- 5、如果找到的内存比用户请求的内存大许多（比如，大于 16 字节），那么内存管理函数把该空闲块分成两块，一块仍然作为空闲块插入空闲链表中，另外一块返回用户。

对于内存回收算法，如下：

- 1、回收函数（KMemAlloc）把释放的内存插入空闲链表；
- 2、在插入的同时，回收函数判断跟该空闲块相邻的下一块是否可以跟当前块合并（合并成更大的块）；
- 3、如果可以合并（地址连续），那么回收函数将合并两块空闲内存块，然后作为一块更大的内存块重新插入空闲链表；
- 4、如果不能合并，则简单返回。

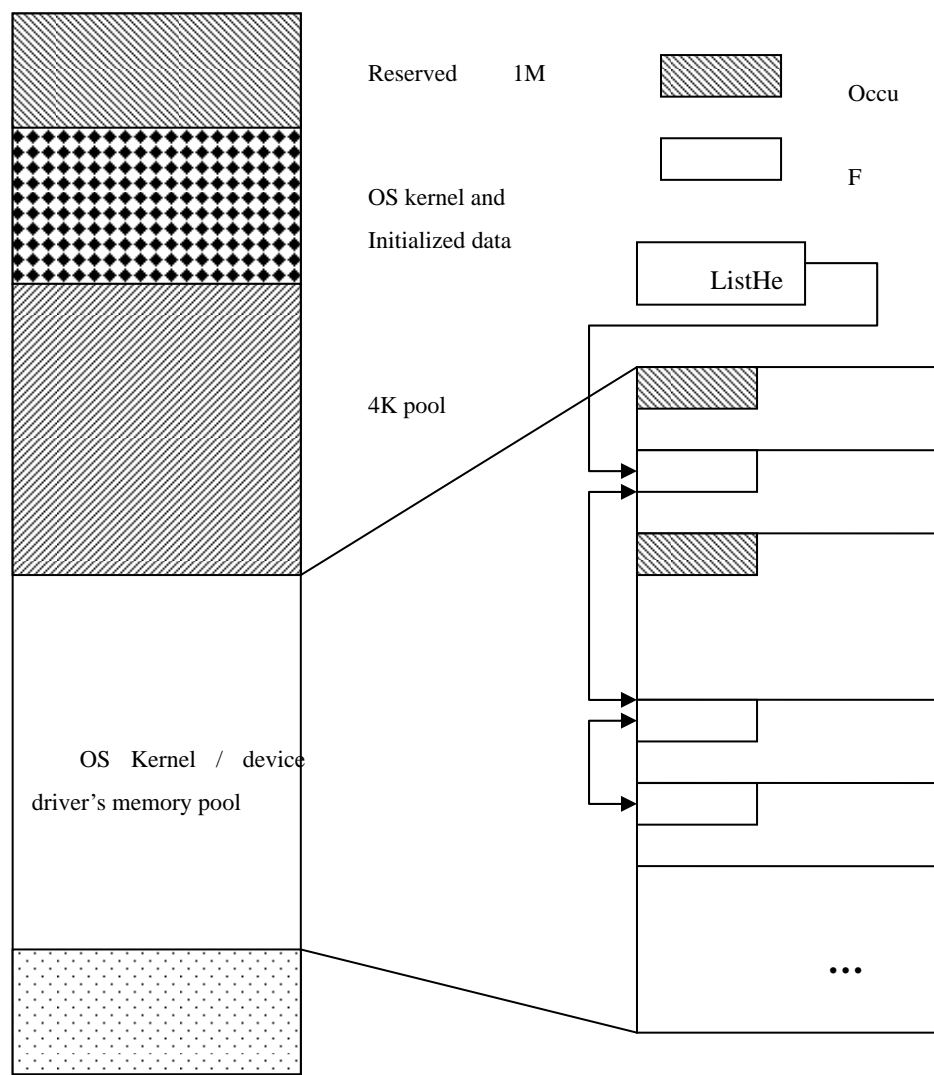


图 5-17 Hello China 的空闲链表算法

为了维护这些空闲块，必须为每块空闲块分配一个控制结构，然后这个控制结构指定了特定的空闲内存块。在分配和回收的时候，需要对空闲块的控制结构进行修改，因此，必须有一种方法，能够快速的定位控制结构。

为了解决这个问题，我们把空闲块的控制结构放在空闲块的前端，这样给定一个内存地址，就可以很容易的索引到其控制块，比如，假设给定的内存地址为 `lpStartAddr`，

空闲内存控制结构为 `__FREE_BLOCK_CONTROL_BLOCK`，那么对应该空闲块的控制结构可以这样获取：

```
__FREE_BLOCK_CONTROL_BLOCK* lpControlBlock =
    (__FREE_BLOCK_CONTROL_BLOCK*)((DWORD)lpStartAddr -
    sizeof(__FREE_BLOCK_CONTROL_BLOCK));
```

这在内存释放（`KMemFree`）的时候特别有用。

在 **Hello China** 当前版本的实现中，空闲链表算法使用的是初次适应算法，即把第一次发现的空闲块分配给用户，而不管这个内存块是否太大，这样往往会造成内存碎片，即随着分配次数的增加，内存中零碎的内存片数量逐渐增多，到了一定的程度，整个内存中全部是零碎的内存片，如果用户请求一块大的内存，往往以失败而告终。但这些缺点仅仅是理论上的，试验表面，首次适应算法能很好的满足实际需求，实际上，很多操作系统的内存分配算法就是使用这种方式实现的，运行效果十分理想。

对于 4K 区域，采用位图算法进行管理，即把整个 4K 区域以 4K 为单位进行划分，对于每个单位，有一个比特与之对应，若该比特的值为 1，则说明该比特对应的内存区域（4K）已经分配，若为 0，则说明该区域尚未分配。如下图：

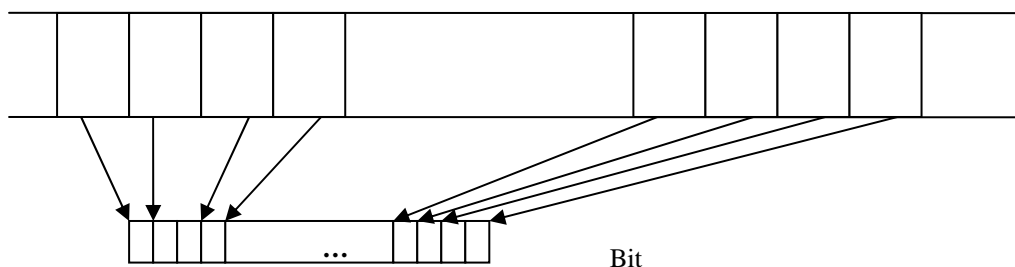


图 5-18 Hello China 的位图算法示意

其中，位图实际上是一个静态定义的全局数组，在操作系统初始化的时候，会对位图数据进行适当的初始化。在内存分配的时候，分配函数会根据请求的大小，检索整个位图，以找到空闲的能够满足请求的内存块。若能够找到符合条件的内存区域，则设置该区域对应的位图，并把首地址返回给应用程序，否则返回 `NULL`。在内存释放的时候，则清除相应的位图标志。

对于核心内存的申请和释放，统一由下列两个函数来完成：

### **KMemAlloc:**

该函数完成核心内存的分配，原型如下：

```
LPVOID KMemAlloc(DWORD dwSize,DWORD dwAllocType);
```

其中，`dwAllocType` 参数指明了要从 4K 区域申请，还是从任何尺寸区域申请，若该参数为 `KMEM_SIZE_TYPE_4K`，则 `KmemAlloc` 从 4K 区域内分配内存，若该参数为 `KMEM_SIZE_TYPE_ANY`，则从任意尺寸区域内分配内存。

### **KmemFree:**

该函数完成核心内存的释放，原型如下：

```
VOID KmemFree(LPVOID lpAddr,DWORD dwAllocType,DWORD dwSize);
```

其中，`lpAddr` 参数指出了要释放的核心内存的首地址，`dwAllocType` 参数指明了内存的位置（与 `KmemAlloc` 一样），对于 4K 区域内的内存块，在释放的时候，需要指定尺寸，即最后一个参数 `dwSize`。

## 页框管理对象（PageFrameManager）

页框管理对象用于对物理内存的分页区域（即 `0x01400000` 到物理内存的末端）进行管理。为了管理上的方便，对物理内存进行分页管理，每页的大小为 `PAGE_FRAME_SIZE`，可以定义为 4K 或 8K 等，根据不同的 CPU 类型确定。为了管理每个页框，使用一个页框对象来描述，如下定义：

```
BEGIN_DEFINE_OBJECT(__PAGE_FRAME)
    __PAGE_FRAME*      lpNextFrame;
```

```
__PAGE_FRAME*      lpPrevFrame;
DWORD              dwKernelThreadNum;
DWORD              dwFrameFlag;
__COMMON_OBJECT*   lpOwner;
__PRIORITY_QUEUE*   lpWaitingQueue;
__ATOMIC_T         Reference;
LPVOID             lpVirtualAddr;
END_DEFINE_OBJECT()
```

上述变量都是不言而喻的，其中，lpVirtualAddr 用来描述该页框被映射到的虚拟内存地址（虚拟地址），如果页框没有被映射，则该虚拟地址不做任何设置。

把物理内存分成一个一个的页框，每个页框的大小为 PAGE\_FRAME\_SIZE（当前版本中，该数字定义为 4K），对每一个页框，分配一个页框对象，系统中所有页框的页框对象结构组合在一起，形成一个数组，整体结构请参考下图：

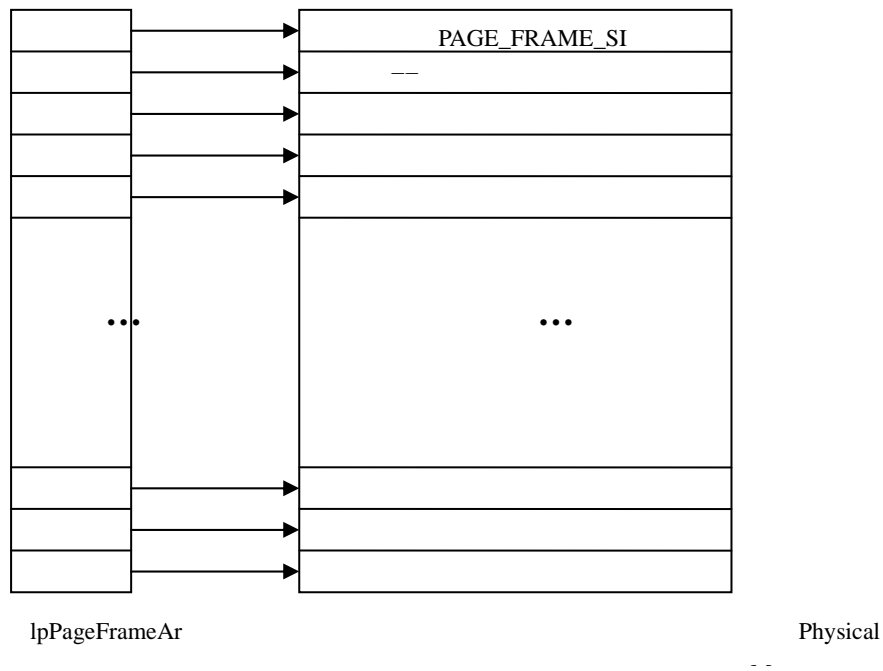


图 5-19 Hello China 的页框管理

注意的是，页框对象数组是动态申请的，即通过 `KMemAlloc`（以 `KMEM_SIZE_TYPE_ANY` 为参数）从核心内存池中分配，这是因为不同的硬件配置，物理内存的数量也不一样，因此无法事先确定页框数组的大小。在操作系统初始化的时候，根据检测到的物理内存的数量，计算出页框数组所需要的尺寸，然后动态申请。申请页框数组内存完成之后，操作系统根据内存情况，初始化页框管理数组，并且记录下物理内存的起始地址，这样就建立了页框管理数组和物理内存的一一对应关系，因此，给定一个页框对象的索引，就可以唯一的确定一块物理内存（尺寸为 `PAGE_FRAME_SIZE`），相反，给定任何一个物理地址，就可以确定该物理地址对应的页框对象。比如，给定一个页框索引为 `N`，那么相应的物理内存块初始地址可以这样计算：

$$\text{lpPageFrameAddr} = \text{lpStartAddr} + \text{PAGE\_FRAME\_SIZE} * N;$$

相反，给定一个物理地址，假设为 `lpPageFrameAddr`，那么对应的页框对象在页框数组内的索引可以这样确定：

$$\text{dwIndex} = (\text{lpPageFrameAddr} - \text{lpStartAddr}) / \text{PAGE\_FRAME\_SIZE};$$

其中，`lpStartAddr` 为物理内存的起始地址（物理地址）。

然而实际上，对内存的请求往往不是一个页框，而是许多页框组成的块，因此，为了更有效的利用内存，我们采用伙伴算法（buddy system algorithm）来对物理页框进行进一步的管理，伙伴算法的一个核心思想就是，通过尽量的合并小的块，来形成大的块，以避免内存浪费。有关伙伴算法的具体流程，请参考数据结构相关的书籍。

在伙伴算法中，把数量不同的连续的物理页框组合成块，在进行分配的时候，根据请求的大小，选择合适的块分配给请求线程。在当前的实现中，一个线程可以请求下列尺寸大小的内存块：

- 1、4K；
- 2、8K；
- 3、16K；

- 4、 32K;
- 5、 64K;
- 6、 128K;
- 7、 256K;
- 8、 512K;
- 9、 1024K;
- 10、 2048K;
- 11、 4096K;
- 12、 8192K。

可以看出，申请的内存块的尺寸，是 4K 的二次方倍数。

在系统核心数据区，维护了下面一个数据结构（参考图）：

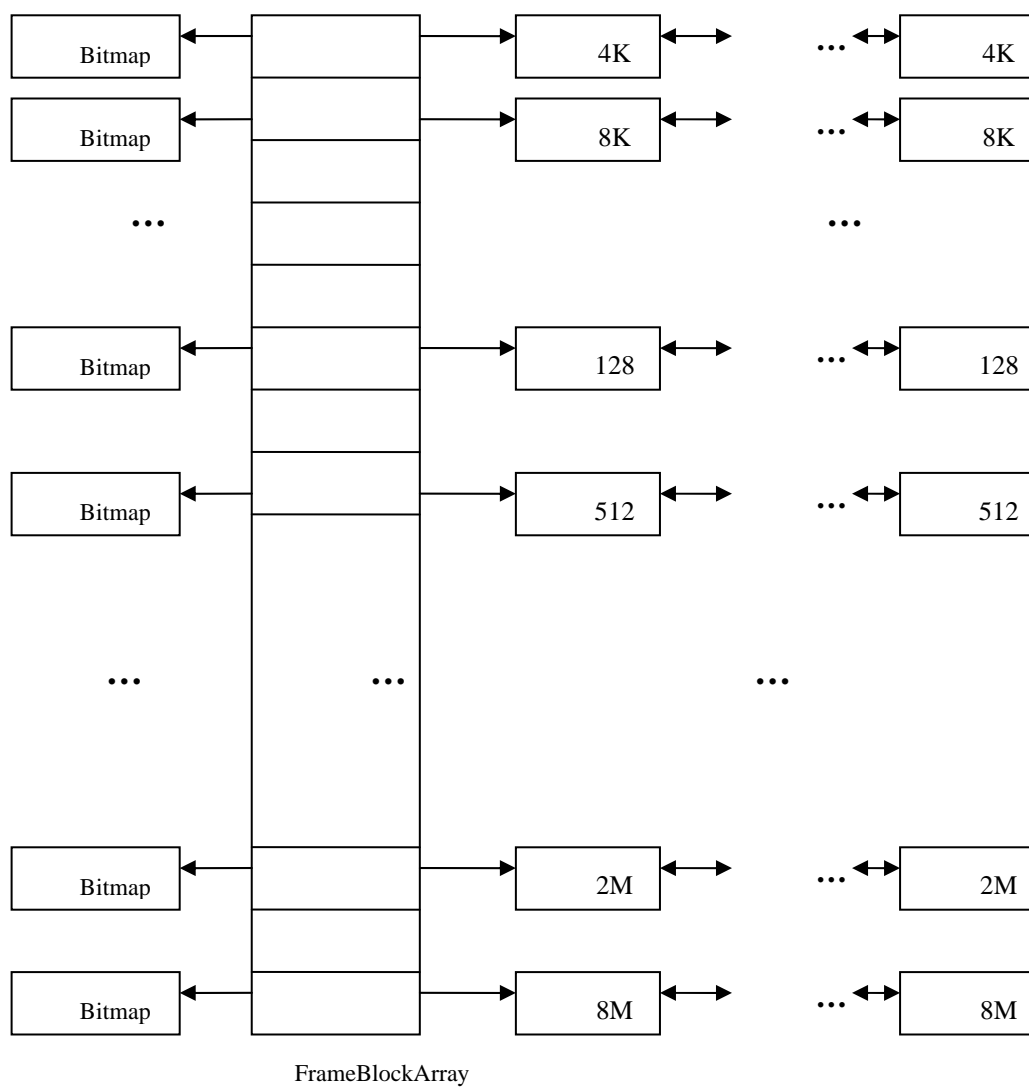


图 5-20 Hello China 的物理页框管理结构

有一个单独的管理对象—页框管理器（`PageFrameManager`）管理所有的页框以及页块，当一个线程请求一块页块时，页框管理器进行下列动作：



DWORD dwSize);

END\_DEFINE\_OBJECT()

其中, `FrameBlockArray` 数组用来描述不同大小的页框, 按照目前的实现, 页框大小按照尺寸组织成 4K、8K 等总共 12 种, 所以该数组的大小 (`PAGE_FRAME_BLOCK_NUM` 的大小) 为 12。 `lpPageFrameArray` 则是一个页框对象类型的数组, 该数组由操作系统在启动的过程中, 根据检测到的物理内存的数量动态分配, 系统中的物理内存被分割成以 `PAGE_FRAME_SIZE` 为大小的页框块 (除去 OS 核心占用的内存), 每块物理内存对应一个页框对象 (`__PAGE_FRAME`), 因此, 建设系统中物理内存的大小为 32M, OS 核心占用了 20M, 这时候, 系统中剩下的 12M 内存, 以 4K 为单位进行分割, 分割成  $12\text{M}/4\text{K} = 3\text{K}$  块, 因此, `lpPageFrameArray` 数组的大小就是  $3\text{K} \times \text{sizeof}(\text{__PAGE\_FRAME})$ 。

`Initialize` 函数是该对象的初始化函数, 该对象是一个全局对象, 即整个系统中只存在一个, 因此, 其初始化函数在操作系统初始化的过程中被调用。 `lpStartAddr` 和 `lpEndAddr` 参数是系统中可用于分页管理的物理内存的起始地址和结束地址, `Initialize` 函数根据这两个参数, 计算出系统中需要分页管理的物理内存的大小, 并根据这两个参数以及计算出来的大小, 初始化页框管理对象的相关变量 (`lpPageFrameArray`、`FrameBlockArray` 等)。在 `Hello China` 目前的实现中, 需要分页管理的物理内存, 起始地址定义为 `0x01400000` (20M 以后), 结束地址根据检测的物理内存数量设定, 比如, 检测到系统中有 64M 的物理内存, 则操作系统初始化的时候, 这样调用该函数:

```
PageFrameManager.Initialize(&PageFrameManager, 0x01400000, 0x03FFFFFF);
```

`FrameAlloc` 和 `FrameFree` 两个函数, 用于具体的页框申请和页框释放操作。 `FrameAlloc` 函数根据调用程序给出的页框需求大小, 分配一个或多个连续的页框, 返回所分配的页框的初始物理地址, 若分配失败 (比如, 系统中没有足够的页框), 则返回 `NULL`。 `FrameFree` 函数则用于释放 `FrameAlloc` 分配的页框, 除了指定要释放的页框的首地址外, 还需要指定要释放的页框的尺寸 (`dwSize` 参数)。

需要注意的是, 若启用了 CPU 提供的分页机制, 则页框管理对象提供的这两个函数 (`FrameAlloc` 和 `FrameFree`), 一般情况下应用程序不要直接调用, 而应该调用虚拟内存管理对象 (下面介绍) 提供的 `VirtualAlloc` 函数, 来具体分配内存。因为直接调用这两个函数, 不会更新系统中的页表和页目录, 会造成系统数据的不一致, 而且若直接访问 `FrameAlloc` 返回的内存地址, 可能会引起异常 (因为这时候系统页表没有更新)。

若没有启用 CPU 的分页机制，则页框管理器提供的这两个页面分配函数可以由应用程序调用，用来完成物理内存的分配。但这两个函数，只能完成以 4K 大小为粒度的物理内存的分配，若需要更小粒度的物理内存，则这两个函数无法满足需求。按照目前版本的 Hello China 的实现，没有实现这种用户应用程序层面的小粒度的内存分配函数，这种情况下，可考虑由应用程序编写者自己编写一个内存分配器，下面是一种可选择的思路：

- 1、应用程序初始化的时候，调用页框管理器提供的 FrameAlloc 函数，分配一定数量的物理内存（比如，32K）；
- 2、把申请的上述 32K 内存，作为应用程序内存池，然后使用空闲链表算法，自己设计一个内存分配器（提供 malloc 和 free 等标准 C 库函数）；
- 3、应用程序每次申请小于 4K 的内存的时候，就调用应用程序开发者自己编写的 malloc 函数分配；
- 4、在内存池不足的情况下，内存分配器可以通过再次调用 FrameAlloc 函数，分配更多的内存。

实际上，很多操作系统的实现，对内存的管理都是以页大小为基础进行的，没有提供更小粒度的内存分配器，更小粒度的内存分配器，在应用程序层面实现。

到此为止，Hello China 物理内存的管理机制，就介绍完毕了，后续部分将详细介绍虚拟内存（基于 IA32 提供的分页机制）的实现机制。下面的表格，对物理内存管理机制的要点，进行了总结：

内存区域	子区域	分配算法	分配单位	分配接口	释放接口
核 心 内存池	4K 池	位 图 算法	4K 倍 数	KMemAlloc	KMemFree
	任 意 尺寸池	空 闲 链表	任 意 大小	KMemAlloc	KMemFree
分 页 管理区		伙 伴 算法	4K 倍 数	FrameAlloc	FrameFree

表 5-6            Hello China 的内存访问服务接口

## 页面索引对象

所谓页索引对象，指的是用于完成虚拟地址和物理地址转换功能的数据结构，比如页目录、页表等，这种数据结构，因不同的硬件平台（CPU）而不同，比如，针对 Intel IA32 系列的 CPU，页目录和页表构成了页索引对象，而对于象 PowerPC 等 RISC 结构的 CPU，则采用 HASH 算法，根据虚拟地址完成物理地址的计算，这样又有了另外一套索引对象（索引数据结构），因此，在 Hello China 的实现中，把所有这些功能使用同一个对象——页索引管理器（PageIndexManager）来进行封装。

PageIndexMgr 用来管理页索引，这个对象的功能，以及内部实现，是与具体的处理器平台密切关联的，比如，针对 Intel 的 IA32 构架，该对象完成该平台下的页目录、页表以及页目录项和页表项的管理，该对象定义如下：

```
BEGIN_DEFINE_OBJECT(__PAGE_INDEX_MANAGER)
    INHERIT_FROM_COMMON_OBJECT
    __PDE*          dwPdAddress;
    BOOL            (*Initialize)(__COMMON_OBJECT*);
    VOID            (*Uninitialize)(__COMMON_OBJECT*);
    LPVOID          (*GetPhysicalAddress)(__COMMON_OBJECT*,LPVOID);
    BOOL            (*ReservePage)(__COMMON_OBJECT*,
                                   LPVOID,LPVOID,DWORD);
    BOOL            (*SetPageFlag)(__COMMON_OBJECT*,
                                   LPVOID,
                                   LPVOID,
                                   DWORD);
    VOID            (*ReleasePage)(__COMMON_OBJECT*,LPVOID);
    /*__PTE*        (*GetPte)(__COMMON_OBJECT*,LPVOID);
    __PDE*          (*GetPde)(__COMMON_OBJECT*,LPVOID);
    VOID            (*SetPteFlags)(__COMMON_OBJECT*,__PTE*,DWORD);
    VOID
    (*SetPdeFlags)(__COMMON_OBJECT*,__PDE*,DWORD);*/
END_DEFINE_OBJECT()
```

其中，dwPdAddress 是页目录的物理地址，在 Intel 构架的 CPU 中，加载到 CR3 寄

寄存器中，用来定位页目录。下面对该对象提供的主要接口函数进行讲解。

## Initialize

该函数是页索引管理对象的初始化函数，在当前版本的实现中，该函数做如下工作：

1、初始化页目录，并把内核占用的头 20M 空间所占用的页目录项和页表项填满。  
在当前的实现中，整个系统只有一个页目录（没有实现不同地址空间的进程），把页目录固定的放置在物理内存 PD\_START 开始的内存处（PD\_START 定义为 0x00200000 - 0x00010000），占用 4K 的物理地址空间，然后把 20M 物理地址空间所占用的页表（共 5 个页表框，20K 内存）项，紧接着页目录存放，并进行填充。如下图：

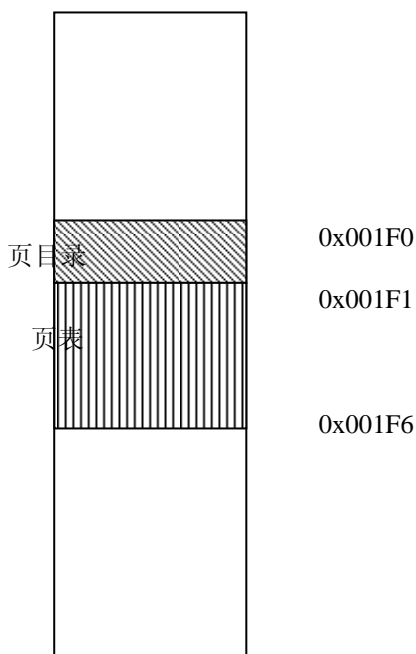


图 5-21 Hello China 的核心页目录和页表在内存中的位置

因为前 20M 是操作系统和设备驱动程序代码、数据占用的空间，所以需要事先填写好。需要注意的是，对于页目录，没有占用的目录项，都使用 `EMPTY_PDE_ENTRY` 填充，即都填充成空。前 20M 物理内存跟线形地址空间的映射关系，采用的是“照实映射”，即映射到线形地址空间内的相同的位置（线形地址空间的前 20M 处），这样做的好处是，

分页机制可以透明的显示给操作系统核心代码，在编写、编译操作系统核心代码的时候，无需考分页机制；

2、完成页目录和页表的填充后，设置 `dwPdAddress` 为 `PD_START`，并加载到 `CR3` 寄存器中（这个时候系统仍然工作在非分页模式，直到所有初始化任务结束后，系统才设置 `CR0` 寄存器，使得整个系统转到分页模式工作。

需要注意的是，在目前的实现中，系统中只有一个 `PageIndexManager` 对象（因为没有实现独立虚拟内存空间的进程模型，整个系统只有一个虚拟内存空间，各核心线程共享这个虚拟内存空间），而这个虚拟内存空间的页索引数据结构（页表、页目录等），被固定在了 `PD_START` 的位置（物理内存），因此不用调用 `KMemAlloc` 函数额外分配页目录和页表。但如果是对多进程模型（为将来考虑，在多进程模型下，每个进程需要有一个页索引管理器），则该函数应该调用 `KMemAlloc` 函数为新创建的进程，分配页索引对象（页表、页目录等），并初始化这些页索引对象（包括把内核的前 20M 内存空间映射到新创建进程的虚拟地址空间中，以使得这前 20M 地址空间，可以被任何进程访问）。

这样一个问题就出现了，就是页索引管理对象如何判断，自己是在操作系统初始化过程中调用（这是第一个页索引对象），还是在操作系统运行过程中，创建进程的时候调用？可以通过下列办法解决：

在第一次调用的时候，页索引管理器对象需要完成 `FD_START` 位置的页目录的初始化，这样该位置的页目录项，将会是一个合法的目录项，后续相关调用，可以检查该位置，是否是一个合法的目录项，或者是一个空目录项（系统初始化时候填充成空目录项），如果是空，则是第一次调用，直接初始化即可，否则，则需要调用 `KMemAlloc` 函数分配页索引对象空间。

### Uninitialize

与 `Initialize` 对应，该函数判断自己是针对进程调用，还是针对系统中的唯一页索引管理器对象调用，如果是后者，不需要做任何事情，如果是前者，则需要释放所有由页索引对象占用的地址空间。

由于 `lpPdAddress` 已经包含了页目录的物理地址，因此，只要对比该地址是否是 `PD_START`，就可以决定，是不是针对进程调用。

### GetPhysicalAddress

该函数完成虚拟地址到物理地址的转换。该函数根据 CPU 特定的转换机制，通过把

虚拟地址进行适当的分割,然后查找页索引而得到物理地址。在 Intel 的 IA32 构架的 CPU 上,该函数这样处理:首先,把虚拟地址的开始 10bit,作为页目录的索引,找到一个页目录项,从页目录项中,得到页表的物理地址,然后利用虚拟地址的中间 10bit,作为页表的索引,找到一个页表项,通过页表项,找到页框的物理地址,然后以虚拟地址的最后 12bit 为偏移,加上页框的物理地址,就可以得到该虚拟地址对应的物理地址。当然,上述能够操作的前提是,该虚拟地址对应的页框存在于物理内存中。如果不存在,或者存在,但被调换出去(后续版本实现),则返回一个 NULL 值。

### ReservePage:

该函数为虚拟地址分配一个页表项,原型如下:

```
BOOL ReservePage(__COMMON_OBJECT* lpThis,LPVOID lpVirtualAddr,LPVOID
lpPhysicalAddr,DWORD dwFlags);
```

其中,lpVirtualAddr 是虚拟地址,而 lpPhysicalAddr 则是物理地址,dwFlags 是页表项的属性。该函数的任务,就是在页索引对象中,通过设置合适的页表和页目录项,完成 lpVirtualAddr 和 lpPhysicalAddr 的映射。

dwFlags 可以取下列值:

```
#define PTE_FLAG_PRESENT    0x001
#define PTE_FLAG_RW        0x002
#define PTE_FLAG_USER      0x004
#define PTE_FLAG_PWT       0x008
#define PTE_FLAG_PCD       0x010
#define PTE_FLAG_ACCESSED  0x020
#define PTE_FLAG_DIRTY     0x040
#define PTE_FLAG_PAT       0x080
#define PTE_FLAG_GLOBAL    0x100
#define PTE_FLAG_USER1     0x200
#define PTE_FLAG_USER2     0x400
#define PTE_FLAG_USER3     0x800
```

如果 dwFlags 包含了 PTE\_FLAG\_PRESENT 位,但 lpPhysicalAddr 为 NULL,则认为是一个错误,直接返回 FALSE。否则,该函数完成下列操作:

- 1、根据页目录索引（虚拟地址的前 10bit），找到对应的页目录项，判断该页目录项是否存在（或者是否已经使用，通过调用 `EMPTY_PDE_ENTRY` 宏来判断），如果没有使用，则说明该页目录对应的页表也不存在，于是先调用 `KMemAlloc` 分配一个页表（4K），对该内存进行清 0，并根据页表的物理地址初始化页目录项；
- 2、根据页目录项找到对应的页表，根据页表索引（虚拟地址的中间 10bit），找到对应的页表项，判断该页表项是否存在（通过调用 `EMPTY_PTE_ENTRY` 宏判断），如果存在，则直接返回 `TRUE`；
- 3、如果页表项不存在，则预留该页表项，并根据 `dwFlags` 的值，设置该页表项的标记（`FLAG`），并进一步判断 `dwFlags` 是否包含 `PTE_FLAG_PRESENT` 位，如果包含，则使用 `lpPhysicalAddr` 设置页表项的页框物理地址；
- 4、上述所有步骤完成之后，返回 `TRUE`。

需要格外说明的是，该函数参数的两个地址（`lpVirtualAddr` 和 `lpPhysicalAddr`），都需要是 4K 边界对其的，否则函数会直接返回 `FALSE`。这个条件，需要调用者保证（即在调用该函数前，首先确保上述两个地址满足 4K 边界对其的要求）。

## SetPageFlag

该函数用于设置页表项的标记属性，原型如下：

```
BOOL SetPageFlag(__COMMON_OBJECT* lpThis, LPVOID lpVirtualAddr, LPVOID lpPhysicalAddr, DWORD dwFlags);
```

其中，`lpVirtualAddr` 是虚拟地址，用于设置由该虚拟地址对应的页表项属性，需要注意的是，该虚拟地址一定是 4K 边界（页长度边界）对齐的，否则该函数将直接返回 `FALSE`，`dwFlags` 是希望设置的属性标志，如果 `dwFlags` 包含了 `PTE_FLAG_PRESENT` 标志位，则 `lpPhysicalAddr` 一定不能为 `NULL`，该数值包含了该页表项对应的页框物理地址，也是 4K 边界对齐的。

该函数进行如下操作：

- 1、判断该虚拟地址对应的页目录项和页表项是否存在，任何一个不存在，都将导致该函数直接返回 `FALSE`；
- 2、找到对应的页表项后，首先检查原页表项的标记是否与 `dwFlags` 一致，如果一致，直接返回 `TRUE`；
- 3、使用 `dwFlags` 的值代替原页表的标记属性，如果 `dwFlags` 设置了 `PTE_FLAG_PRESENT` 位，则再使用 `lpPhysicalAddr` 设置页表项对应的页框的物理地址；
- 4、上述所有操作完成之后，返回 `TRUE`。

## ReleasePage

该函数释放虚拟地址占用的页表项，原型如下：

```
VOID ReleasePage(__COMMON_OBJECT* lpThis, LPVOID lpVirtualAddr);
```

其中，`lpVirtualAddr` 就是要释放的页表项，所对应的虚拟地址。该函数首先检查对应的页表项是否存在，如果不存在，直接返回，否则，把对应的页表项设置成一个空页表项，然后检查该页表项对应的页表框是否还有保留的页表项（通过调用 `EMPTY_PTE_ENTRY` 判断），如果该页表框已经没有保留的页表项了，则释放该页表框占用的内存，并设置对应的页目录项为空，然后返回。

之所以在该函数中，检查并释放页表框，是为了与 `ReservePage` 对应，确保系统中不存在内存浪费。

## 页索引管理器的应用

页索引管理器对特定 CPU 的分页机制进行了封装，使得不同的分页机制，对外表现出相同的处理接口，这样便于代码的移植。需要注意的是，页索引管理器仅仅完成页索引的操作，比如预留、释放、设置标志等，一般情况下，应用程序不要直接调用这些操作接口，以免引起系统的崩溃。该对象提供的接口函数，被虚拟内存管理器调用，用来完成线形地址空间内的页面和物理地址空间内的页面之间的映射。

## 虚拟内存管理对象（VirtualMemoryMgr）

### 虚拟区域

采用虚拟区域（Virtual Area）来表示线形内存空间中的一个区域，需要注意的是，这个区域仅仅是线形内存空间的一部分，不一定跟物理内存存在映射关系，这个时候，如果引用该虚拟内存空间，由于没有跟物理内存对应，所以会导致访问异常。因此，对于虚拟内存区域，在使用前，一定要通过 API 调用，来完成虚拟内存到物理内存的映射。

但需要说明的是，虚拟内存区域不仅仅可以与物理内存之间完成映射，甚至可以与存储系统上的文件、硬件设备的内存映射区域等映射。比如，可以把一个存储系统文件的部分内容（或全部内容），映射到一个进程（或系统）的虚拟地址空间中，这个时候，只要按通常的内存访问方式，就可以访问文件中的内容了，十分方便。对于内存和文件之间的同步，由操作系统保证，对应用程序来说是透明的。

还有一个应用就是，把设备的内存映射区域映射到虚拟空间中，这时候，只要访问

虚拟内存中的相关区域，就可以直接访问设备了。

在当前版本的实现中，Hello China 只实现了虚拟内存的基本功能，即可以把虚拟地址空间中的一个区域（由虚拟区域描述），映射到物理内存或设备的 IO 内存映射区域中，通过对虚拟内存的访问，来完成对设备或物理内存的访问。

针对每块虚拟区域，有一个虚拟区域描述符进行描述、管理，虚拟区域描述符（Virtual Area Descriptor）的定义如下：

```
DECLARE_PREDEFINED_OBJECT(__VIRTUAL_MEMORY_MANAGER);

BEGIN_DEFINE_OBJECT(__FILE_OPERATIONS)
    DWORD          (*FileRead)(__VIRTUAL_MEMORY_DESCRIPTOR*);
    DWORD          (*FileWrite)(__VIRTUAL_MEMORY_DESCRIPTOR*);
END_DEFINE_OBJECT() //This object is not used currently ,but maybe used in the
future.

BEGIN_DEFINE_OBJECT(__VIRTUAL_AREA_DESCRIPTOR)
    __VIRTUAL_MEMORY_MANAGER*    lpManager;
    LPVOID                      lpStartAddr;
    LPVOID                      lpEndAddr;
    __VIRTUAL_AREA_DESCRIPTOR*   lpNext;
    DWORD                      dwAccessFlags;
    DWORD                      dwCacheFlags;
    DWORD                      dwAllocFlags;
    __ATOMIC_T                  Reference;
    DWORD                      dwTreeHeight;
    __VIRTUAL_AREA_DESCRIPTOR*  lpLeft;
    __VIRTUAL_AREA_DESCRIPTOR*  lpRight;
    UCHAR
strName[MAX_VA_NAME_LEN];
    __FILE*                    lpMappedFile;
    DWORD                      dwOffset;
    __FILE_OPERATIONS*         lpOperations;
END_DEFINE_OBJECT()
```

在当前的实现中，把描述每块虚拟区域的虚拟区域描述符，通过链表的方式连接在一起（lpNext 指针），形成了下面的结构：

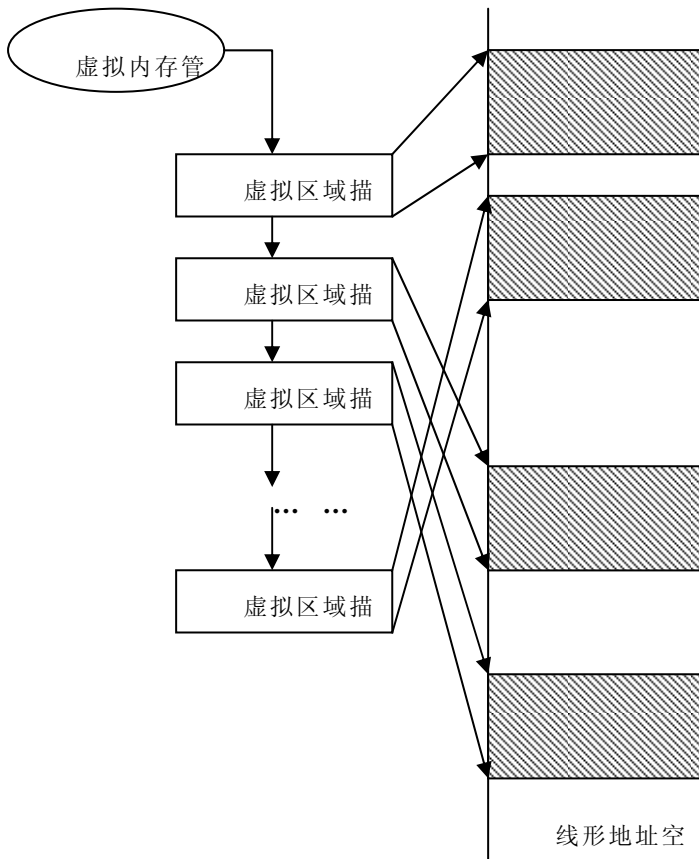


图 5-22 Hello China 的虚拟区域管理结构

之所以把虚拟区域（线形地址空间）进行统一管理，是因为线形地址空间也是一种重要的系统资源，许多实体，比如设备驱动程序等，都需要一块线形地址空间来完成设备寄存器的映射，这时候若不进行统一管理、统一分配，则可能会导致冲突，即两个实体占用了同一块线形地址空间区域。进行统一管理后，操作系统提供统一的接口给应用

程序或驱动程序，用以申请线形地址空间的某一块区域。在受理虚拟区域申请的时候，操作系统首先检索整个线形地址空间的分配情况（通过遍历虚拟区域描述符链表），从未分配的区域中，选择一块合适的分配给应用程序或设备驱动程序，并设置该虚拟区域对应的页表和页目录（通过页索引对象）。在 32 位线形地址空间中，整个线形地址空间的大小为 4G，这是一个庞大的空间，这样如果有大量的虚拟区域描述符存在，则遍历虚拟区域描述符链表将是一件非常耗时间的东西，因为遍历链表花费的时间，跟链表元素的数量是成正比例关系的。这种情况下，为了提高系统的效率，Hello China 采用了两种数据结构管理虚拟区域描述符，除了上述的链表方式外，还采用了平衡二叉树的方式，对虚拟区域描述符进行管理。在目前的实现中，若虚拟区域描述符的数量小于 64，则采用链表进行管理，若一旦虚拟区域描述符的数量超过了这个数值，则切换到平衡二叉树进行管理。

上述虚拟区域组成的链表（或二叉树），是由虚拟内存管理器（Virtual Memory Manager）进行管理的。每个具有独立地址空间的进程，有一个对应的虚拟内存管理器对象，当前的实现中，由于没有引入进程的概念，因此只有一个全局的虚拟内存管理器，用来管理整个系统的虚拟内存，所有的内核线程对象都共享这个虚拟内存管理器，进而共享整个的虚拟内存空间。

lpStartAddr 和 lpEndAddr 指明了本虚拟区域的起始虚拟地址和结束虚拟地址，而 dwAccessFlags 和 dwCacheFlags 则指明了本虚拟区域的访问属性和缓存属性。访问属性可以取下列值：

```
#define VIRTUAL_AREA_ACCESS_READ      0x00000001
#define VIRTUAL_AREA_ACCESS_WRITE     0x00000002
#define VIRTUAL_AREA_ACCESS_RW        0x00000004
#define VIRTUAL_AREA_ACCESS_EXEC      0x00000008
```

上述各值，在请求该虚拟区域的时候指定（一般是调用者指定）。

而缓存属性可以取下列值：

```
#define VIRTUAL_AREA_CACHE_NORMAL     0x00000001
#define VIRTUAL_AREA_CACHE_IO         0x00000002
#define VIRTUAL_AREA_CACHE_VIDEO      0x00000004
```

其中, `VIRTUAL_AREA_CACHE_NORMAL` 指明了, 当前虚拟区域如果跟物理内存进行关联, 则使用缺省的内存缓冲策略 (也就是物理内存跟 L1、L2 和 L3 等处理器 cache 之间的缓冲/替换策略), 一般情况下, 缺省的缓存策略为回写方式, 即对于读操作, 直接从 `CACHE` 里面读取, 如果没有命中, 则引发一个 `CACHE` 行更新, 对于写操作, 写入 `CACHE` 的同时, 直接写入物理内存, 即写操作所影响的数据, 不会在 `CACHE` 中缓存, 而是直接反映到物理内存中。但需要注意的是, 对于写操作, 处理器可能会使用内部的写合并 (Write Combine) 缓冲区。

`VIRTUAL_AREA_CHCCE_IO` 则指明了当前的虚拟区域是一个 IO 设备的映射区域, 这样对该区域的 `CACHE` 策略, 应该是禁用系统 `CACHE`, 并禁用投机读等提高效率的策略, 而应该严格按照软件编程顺序对虚拟区域进行访问。这是因为设备映射的 IO 区域, 一般情况下是跟物理设备的寄存器对应的, 而这些物理设备的寄存器, 可能处于不断的变化当中, 若采用 `cache` 缓存的读策略, 则可能出现数据不一致的情况, 因此在物理设备驱动程序的实现中, 若需要申请虚拟区域, 一定要采用 `VIRTUAL_AREA_CACHE_IO` 来作为申请标志。一般情况下, 对于 PCI 设备的内存映射区域, 应该设置这种缓冲策略。

在这里说一点题外话, `Hello China` 由于定位于嵌入式的操作系统, 即使运行在 PC 上, 也是常驻内存的, 不会发生物理内存和存储设备之间的内存替换, 而且也没有必要引入进程概念, 因此, 没有必要实现分页机制。而且按照通常的说法, 实现分页机制, 会导致系统的整体效率大大下降 (因为如果实现了分页机制, 对于一次内存的访问, 可能需要多次实际的内存读写才能完成, 因为 CPU 要根据页表和页目录来完成实际物理内存的定位, 尽管采用 TLB 等缓冲策略, 可以提高访问效率, 但相对不分页来说, 系统效率还是会大大降低), 但后来的一些设备, 比如网卡、显示卡等物理设备, 需要把内部寄存器映射到存储空间, 而且这些映射到的存储空间, 还不能采用缺省的内存缓冲策略, 这样就必须采用一些额外机制, 保证这些内存映射区域的完整性 (不会因为提前读而导致数据不一致), 而分页是一种最通用的内存控制策略, 可以在页级对虚拟内存区域属性进行控制, 因此, 在目前的 `Hello China` 版本中, 实现了基于分页机制的虚拟内存管理系统, 而且这个系统是可裁减的, 即通过调整适当的编译选项, 可以选择编译后的内核, 是否包含该系统。

后续 `Hello China` 的实现中, 可能会因为额外的需要, 实现一个更完整的虚拟内存系统 (比如, 增加文件和虚拟内存的映射、页面换出等功能), 但至少目前还没有这个必要。

另外, 在 Intel 的处理器上, 可以通过设置一些控制寄存器, 比如 `MTTR` 等, 来控制

缓存策略，但不作为一种通用的方式，在当前 Hello China 的实现中，不作考虑。

VIRTUAL\_AREA\_CACHE\_VIDEO 是另外一种 CACHE 策略，这种策略可以针对 VIDEO 的特点，进行额外优化，在这里不做详细描述。

为了将来扩充方便，在当前虚拟区域的定义中，也引入了相关变量，来描述虚拟区域和存储系统文件之间的映射关系（`lpMappedFile`、`dwOffset` 和 `lpOperations` 三个变量），但在当前版本中，没有实现该功能，其一是因为没有必要（就目前 **Hello China** 的应用来说），其二是因为 **Hello China** 没有实现文件系统（将来的版本中，会实现）。

最后要说明的是，为便于描述每个虚拟区域，在虚拟区域描述对象中，引入了虚拟区域名字变量，其最大长度是 `MAX_VA_NAME_LEN`（目前定义为 32），该变量在分配虚拟区域的时候，被虚拟内存管理器填写，当然，最初的来源，仍然是由用户指定（参考 `VirtualAlloc` 的定义）。

## 虚拟内存管理器 (Virtual Memory Manager)

虚拟内存管理器是 Hello China 的虚拟内存管理机制的核心对象,提供了应用程序(或设备驱动程序)可以直接调用的接口,用来完成虚拟内存(线形地址空间)的分配。另外,该对象还维护了虚拟区域描述符链表(或二叉树)等数据。在当前的实现中,由没有实现进程模型,因此整个系统中只有一个虚拟内存管理器,用以对虚拟地址空间进行管理,若实现了进程模型,则每个进程需要有自己的虚拟内存管理器对象。

下面就是虚拟内存管理器对象的定义:

```
BEGIN_DEFINE_OBJECT(__VIRTUAL_MEMORY_MANAGER)
    INHERIT_FROM_COMMON_OBJECT
    __PAGE_INDEX_OBJECT*                lpPageIndexMgr;
    __VIRTUAL_AREA_DESCRIPTOR*           lpListHdr;
    __VIRTUAL_AREA_DESCRIPTOR*           lpTreeRoot;
    __ATOMIC_T                           Reference;
    DWORD                                dwVirtualAreaNum;
    __LOCK_T                             SpinLock;

    BOOL (*Initialize)(__COMMON_OBJECT*);
```

```

VOID          (*Uninitialize)(__COMMON_OBJECT*);
LPVOID        (*VirtualAlloc)(__COMMON_OBJECT*,
                                LPVOID,      //Desired start virtual address
                                DWORD,       //Size
                                DWORD,       //Allocation flags
                                DWORD,       //Access flags.
                                UCHAR*,      //Virtual area name.
                                LPVOID);     //Reserved.
VOID          (*VirtualFree)(__COMMON_OBJECT*,
                                LPVOID);
DWORD         (*GetPdeAddress)(__COMMON_OBJECT*);
END_DEFINE_OBJECT()

```

其中，lpPageIndexMgr 指向了一个页面索引管理对象（\_\_PAGE\_INDEX\_MANAGER），用来管理该虚拟内存空间的页索引对象（页表、页目录等）。在系统核心，页面管理对象（PageFrameManager）、页索引管理对象（PageIndexManager）和虚拟内存管理对象（VirtualMemoryManager）的数量关系是，整个系统一个 PageFrameManager，用来管理整个系统的物理内存（因为在单处理系统或对称多处理器系统中，整个系统只有一个共享的物理内存池，当然，不考虑多处理器、多内存池的情况），而一个线形地址空间，对应一个虚拟内存管理对象，一个线形地址空间，对应一个页面索引管理对象，在当前的实现中，由于整个系统只有一个虚拟内存空间（没有引入进程的概念），因此，整个系统中，这三种对象都只有一个。经来如果引入了进程的概念，每个进程一个虚拟内存空间，那么系统中就会存在多个虚拟内存管理对象和多个页面管理对象（每个进程一个），但仍然只有一个页面管理对象。这三种类型的内存管理对象的关系如下：

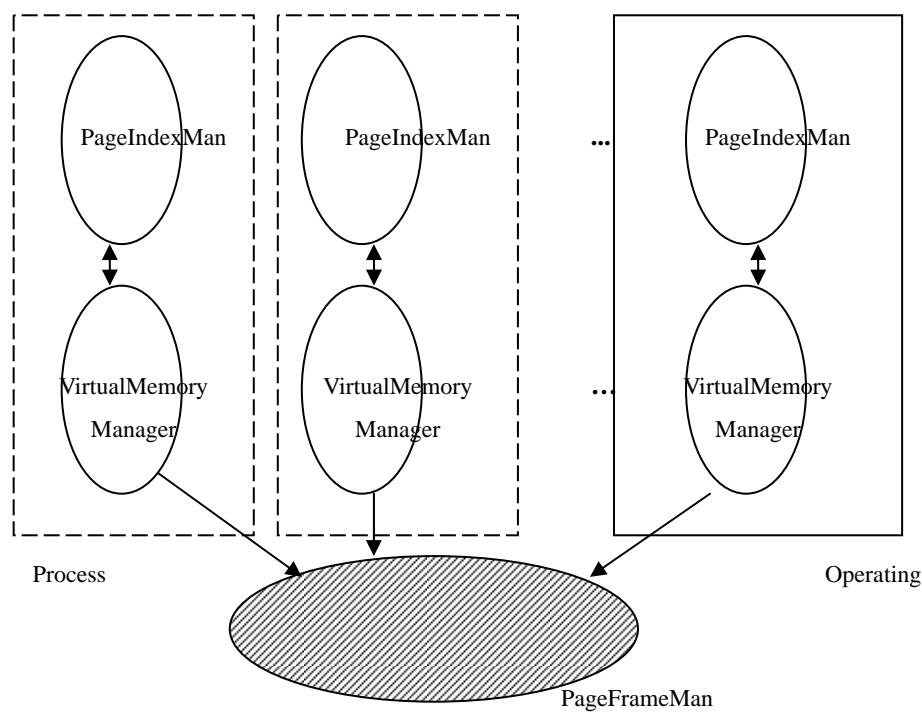


图 5-23 页框管理器、虚拟内存管理器和页索引管理器

由于当前没有实现多线程模型，因此系统中只有一个操作系统内存空间，图中实线矩形表示当前已经实现的操作系统内存空间，而虚线矩形则表示了每个进程的虚拟地址空间。

`lpListHdr` 指向虚拟区域链表，`lpTreeRoot` 也是用来维护虚拟区域的，在当前的实现中，如果虚拟区域的数量少于 `MAX_VIRTUAL_AREA_NUM` 个（当前，定义为 64），则使用线性表进行管理，如果超出了 `MAX_VIRTUAL_ZREA_NUM` 个，则使用平衡二叉树进行管理，以加快查找等操作的速度。

`SpinLock` 用在 `SMP`（对称多处理系统）上，以同步对虚拟内存管理器对象的访问（在单处理器系统上，没有任何用途），`dwVirtualAreaNum` 是目前已经分配的虚拟区域的数量。

下面介绍虚拟内存管理器提供的函数，其中，这些仅仅是对外可用的，还有一些函数，作为内部辅助函数，没有对外提供，因此，在这里不作介绍。这些函数中，最重要的一个，就是 **VirtualAlloc** 函数，这个函数是虚拟内存管理系统对外的最主要接口，也是用户线程（或实体）请求虚拟内存服务的唯一接口。

## Initialize

这是该对象的初始化函数，目前来说，该函数完成下列功能：

- 1、设置该对象的函数指针值；
- 2、创建第一块虚拟区域（Virtual Area，通过调用 **KMemAlloc** 函数），起始地址为 0，终止地址为 0x013FFFFFFF，长度为 20M（该内存区域被操作系统核心数据和代码、核心内存池等占用），访问属性为 **VIRTUAL\_AREA\_ACCESS\_RW**，缓冲策略为 **VIRTUAL\_AREA\_CACHE\_NORMAL**，并把该虚拟区域对象插入虚拟区域列表；
- 3、调用 **ObjectManager** 的 **CreateObject** 方法，创建一个 **PageIndexManager** 对象；
- 4、调用 **PageIndexManager** 的初始化函数（该函数完成系统空间的页表预留工作）；
- 5、设置 **dwVirtualAreaNum** 为 1；
- 6、如果上述一切正常，返回 **TRUE**，否则返回 **FALSE**。

操作系统在初始化的时候，调用该函数（**Initialize**），如果该函数失败（返回 **FALSE**），则直接导致操作系统初始化不成功。

## Uninitialize

目前情况下，该函数不作任何工作，因为该函数只能在操作系统关闭的时候调用（系统中只有一个虚拟内存管理器对象），但是如果在多进程的环境下，该函数调用 **DestroyObject**（**ObjectManager** 对象提供）函数，释放 **PageIndexManager** 对象，并删除所有创建的 **Virtual Area** 对象。

## VirtualAlloc

该函数用来分配虚拟内存空间中的内存，是虚拟内存管理器提供给应用程序的最重要接口，该函数原型如下：

```
LPVOID VirtualAlloc(__COMMON_OBJECT* lpThis,
                    LPVOID lpDesiredAddr,
                    DWORD dwSize,
                    DWORD dwAllocationFlag,
```

```
DWORD dwAccessFlag,
    UCHAR* lpVaName,
    LPVOID lpReserved);
```

其中，lpDesiredAddr 是应用程序希望地址，即应用程序希望能够得到 lpDesiredAddr 开始，dwSize 大小的一块虚拟内存，dwAllocationFlag 则指出了希望的分配类型，有下列可取值：

```
#define VIRTUAL_AREA_ALLOCATE_RESERVE    0x00000001
#define VIRTUAL_AREA_ALLOCATE_COMMIT    0x00000002
#define VIRTUAL_AREA_ALLOCATE_IO        0x00000004
#define VIRTUAL_AREA_ALLOCATE_ALL        0x00000008
```

各标志的含义如下：

- **VIRTUAL\_AREA\_ALLOCATE\_RESERVE**：该标志指明了应用程序只希望系统能够预留一部分线形内存空间，不需要分配实际的物理内存，VirtualAlloc 函数在处理这种类型的请求时，只会检索虚拟区域描述符表，查找一块未分配的虚拟内存区域，并返回给用户，同时，调用 PageIndexManager 提供的接口，建立刚刚分配的虚拟内存对应的页表，需要注意的是，这个时候建立的页表项，其 P 标志（存在标志）被设置为 0，表明该虚拟内存区域尚未分配具体的物理内存，这样，对这一块虚拟内存的访问，会引起异常；
- **VIRTUAL\_ALLOCATE\_COMMIT**：使用该标志调用 VirtualAlloc 的应用程序，希望完成预先预留的（以 VIRTUAL\_AREA\_ALLOCATE\_RESERVE 调用 VirtualAlloc）虚拟内存空间的物理内存分配工作，即为预先分配的虚拟内存预留物理内存空间，并完成页表的更新，这时候，访问对应的虚拟内存的时候，就不会引起异常了；
- **VIRTUAL\_AREA\_ALLOCATE\_IO**：使用该标志调用 VirtualAlloc 函数，说明调用者希望预留的虚拟内存区域，是用于 IO 映射，这种情况下，系统不但需要预留虚拟内存空间，而且还要完成系统页索引结构的初始化，即根据预留结果，填写页表，这时候，预留的线形地址空间的地址，跟采用页索引机构映射到物理地址空间的地址是一样的，直接映射到设备的“寄存器地址空间”；
- **VIRTUAL\_AREA\_ALLOCATE\_ALL**：采用该标志调用 VirtualAlloc 函数，说明应用程序既需要预留一部分虚拟内存空间，也需要为对应的虚拟内存空间分配物理内存，并完成两者之间的映射（填写页面索引数据结构），页就是说，VIRTUAL\_AREA\_ALLOCATE\_ALL 是 VIRTUAL\_AREA\_ALLOCATE\_RESERVE 和 VIRTUAL\_AREA\_ALLOCATE\_COMMIT 两个标志的结合。

dwAccessFlags 说明了调用者希望的访问类型，可以取下列值：

```
#define VIRTUAL_AREA_ACCESS_READ      0x00000001
#define VIRTUAL_AREA_ACCESS_WRITE     0x00000002
#define VIRTUAL_AREA_ACCESS_RW        0x00000004
#define VIRTUAL_AREA_ACCESS_EXEC      0x00000008
```

上述各取值的含义，都是很明确的。lpVaName 指明了虚拟区域的名字，一般用来描述虚拟区域，lpReserved 用于将来使用，当前情况下，用户调用的时候，一定要设置为 NULL。

下面根据不同的分配标志，对 VirtualAlloc 的动作进行详细描述。

## **VIRTUAL\_AREA\_ALLOCATE\_IO**

设置了这个标志，说明分配者希望分配一块内存映射 IO 区域，作为访问 IO 设备使用。一般情况下，用户指定了 lpDesiredAddr 参与，希望系统能够在 lpDesiredAddr 开始的地方开始分配。

这种情况下，VirtualAlloc 进行如下处理：

- 1、向下舍入 lpDesiredAddr 地址到 PAGE\_FRAME\_SIZE 边界，在 dwSize 上增加向下舍入的部分，并向上舍入 dwSize，到 FRAME\_PAGE\_SIZE 边界；
- 2、检查从 lpDesiredAddr 开始，长度为 dwSize 的虚拟内存空间是否已经分配，或者是否与现有的区域重叠，这项检查通过遍历虚拟区域链表或虚拟区域 AVL 树来完成；
- 3、如果所请求的区域既没有分配，也没有与现有区域重叠，则创建一个虚拟区域描述对象（\_\_VIRTUAL\_AREA\_DESCRIPTOR），设置该对象的相关成员；
- 4、如果请求的区域已经分配，或者与现有的区域有重叠，则需要在虚拟地址空间中，重新寻找一块区域，如果寻找成功，则创建虚拟区域描述对象，否则，直接返回 NULL，指示操作失败；
- 5、把上述区域描述对象插入链表或 AVL 树（根据目前虚拟区域数量决定）；
- 6、递增 dwVirtuanAreaNum；
- 7、以 FRAME\_PAGE\_SIZE 为递增单位，循环调用 lpPageIndexMgr 的 ReservePage 函数，在系统页表中增加对新增加区域的页表项，页表项的虚拟地址和物理地址相同（都是 lpDesiredAddr），页表项的属性为 PTE\_FLAG\_PRESENT、PTE\_FLAG\_NOCACHE，期访问对象，根据 dwAccessFlags 标志，设置为 PTE\_FLAG\_READ、PTE\_FLAG-WRITE 或 PTE\_FLAG\_RW；
- 8、设置 dwAllocFlags 为 VIRTUAL\_AREA\_ALLOCATE\_IO，以指明没有为该虚拟内存区域，分配物理内存；
- 9、如果上述一切成功，则返回 lpDesiredAddr（注意，该数值可能是最初用户调用的时

候设置的数值，也可能是由 VirtualAlloc 重新分配的数值)，以指明调用成功。

下面是上述实现的详细代码：

```
static LPVOID VirtualAlloc(__COMMON_OBJECT* lpThis,
                           LPVOID          lpDesiredAddr,
                           DWORD           dwSize,
                           DWORD           dwAllocFlags,
                           DWORD           dwAccessFlags,
                           UCHAR*          lpVaName,
                           LPVOID          lpReserved)
{
    switch(dwAllocFlags)
    {
        case VIRTUAL_AREA_ALLOCATE_IO:    //Call DoIoMap only.
            return DoIoMap(lpThis,
                           lpDesiredAddr,
                           dwSize,
                           dwAllocFlags,
                           dwAccessFlags,
                           lpVaName,
                           lpReserved);
            break;
        case VIRTUAL_AREA_ALLOCATE_RESERVE:
            ... ..
        default:
            return NULL;
    }
    return NULL;
}
```

VirtualAlloc 函数判断分配标志，根据不同的标志，再进一步调用特定的实现函数。在分配标志是 VIRTUAL\_AREA\_ALLOCATE\_IO 的情况下，调用了 DoIoMap 函数，该函数实际完成预留功能，下面是该函数的实现代码，由于函数较长，我们分段解释：

```

static LPVOID DoIoMap(__COMMON_OBJECT* lpThis,
                    LPVOID          lpDesiredAddr,
                    DWORD            dwSize,
                    DWORD            dwAllocFlags,
                    DWORD            dwAccessFlags,
                    UCHAR*           lpVaName,
                    LPVOID           lpReserved)
{
    __VIRTUAL_AREA_DESCRIPTOR*      lpVad      = NULL;
    __VIRTUAL_MEMORY_MANAGER*        lpMemMgr    =
(__VIRTUAL_MEMORY_MANAGER*)lpThis;
    LPVOID                           lpStartAddr = lpDesiredAddr;
    LPVOID                           lpEndAddr   = NULL;
    DWORD                            dwFlags      = 0L;
    BOOL                             bResult      = FALSE;
    LPVOID                           lpPhysical   = NULL;
    __PAGE_INDEX_MANAGER*             lpIndexMgr  = NULL;
    DWORD                            dwPteFlags   = NULL;

    if((NULL == lpThis) || (0 == dwSize))    //Parameter check.
        return NULL;
    if(VIRTUAL_AREA_ALLOCATE_IO != dwAllocFlags)    //Invalidate flags.
        return NULL;
    lpIndexMgr = lpMemMgr->lpPageIndexMgr;
    if(NULL == lpIndexMgr)    //Validate.
        return NULL;

```

上述代码完成了本地变量的定义、参数合法性检查等工作，以确保参数的合法性。

```

lpStartAddr = (LPVOID)((DWORD)lpStartAddr & ~(PAGE_FRAME_SIZE - 1));
//Round up to page.

```

```

lpEndAddr    = (LPVOID)((DWORD)lpDesiredAddr + dwSize );

```

```

lpEndAddr    = (LPVOID)((((DWORD)lpEndAddr & (PAGE_FRAME_SIZE - 1)) ?
                        (((DWORD)lpEndAddr & ~(PAGE_FRAME_SIZE - 1)) + PAGE_FRAME_SIZE -
1)
                        : ((DWORD)lpEndAddr - 1)); //Round down to page.

dwSize       = (DWORD)lpEndAddr - (DWORD)lpStartAddr + 1; //Get the actually
size.

```

上述代码把起始地址（应用程序可以指定一个期望预留的起始地址）舍入到页面长度边界，并根据长度，计算预留的虚拟地址空间的结束地址，计算出结束地址后，也舍入到页面边界，最后计算出实际预留长度（因为经过上面两次舍入，预留长度可能会变化）。

```

lpVad                                               =
(__VIRTUAL_AREA_DESCRIPTOR*)KMemAlloc(sizeof(__VIRTUAL_AREA_DESCRIPTOR),
KMEM_SIZE_TYPE_ANY); //In order to avoid calling KMemAlloc routine in the
                        //critical section,we first call it here.
if(NULL == lpVad)                                   //Can not allocate memory.
    goto __TERMINAL;
lpVad->lpManager      = lpMemMgr;
lpVad->lpStartAddr     = NULL;
lpVad->lpEndAddr       = NULL;
lpVad->lpNext          = NULL;
lpVad->dwAccessFlags   = dwAccessFlags;
lpVad->dwAllocFlags    = dwAllocFlags;
__INIT_ATOMIC(lpVad->Reference);
lpVad->lpLeft          = NULL;
lpVad->lpRight         = NULL;
if(lpVaName)
{
    if(StrLen((LPSTR)lpVaName) > MAX_VA_NAME_LEN)
        lpVaName[MAX_VA_NAME_LEN - 1] = 0;
    StrCpy((LPSTR)lpVad->strName[0],(LPSTR)lpVaName); //Set the virtual area's

```

```

name.
}
else
    lpVad->strName[0] = 0;
lpVad->dwCacheFlags = VIRTUAL_AREA_CACHE_IO;

```

上述代码调用 `KMemAlloc` 函数，创建了一个虚拟区域描述符对象，并根据应用程序提供的参数，对虚拟区域描述符对象进行了初始化。

```

//
//The following code searches virtual area list or AVL tree,to check if the lpDesiredAddr
//is occupied,if so,then find a new one.
//
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
if(lpMemMgr->dwVirtualAreaNum < SWITCH_VA_NUM) //Should search in the list.
    lpStartAddr =
SearchVirtualArea_l((__COMMON_OBJECT*)lpMemMgr,lpStartAddr,dwSize);
else //Should search in the AVL tree.
    lpStartAddr =
SearchVirtualArea_t((__COMMON_OBJECT*)lpMemMgr,lpStartAddr,dwSize);
if(NULL == lpStartAddr) //Can not find proper virtual area.
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    goto __TERMINAL;
}

```

上述代码根据目前虚拟区域描述符数量的大小，检索虚拟区域描述符链表或二叉树，以找到一个满足用户需求的虚拟区域。这时候，有三种情况，一种情况是，能够找到一个满足用户需求大小的虚拟区域，但其起始地址跟用户提供的期望的起始地址不一致。这时候，`VirtualAlloc` 仍然预留找到的虚拟区域，并把预留的虚拟区域的起始地址返回给用户。另外一种情况是，查找到的虚拟区域完全满足用户的需求，即大小和起始地址都适合（用户期望预留的虚拟区域尚未被占用），这种情况下，`VirtualAlloc` 也是以预留成功处理，返回用户预留的起始地址。若未能找到满足用户要求的结果（即系统线形空间中，没有足够大的连续区域，能够满足用户需求），则 `VirtualAlloc` 调用将以失败而告终，返

回用户一个 NULL。

```

lpVad->lpStartAddr = lpStartAddr;
lpVad->lpEndAddr    = (LPVOID)((DWORD)lpStartAddr + dwSize -1);
lpDesiredAddr       = lpStartAddr;

if(lpMemMgr->dwVirtualAreaNum < SWITCH_VA_NUM)
    InsertIntoList((__COMMON_OBJECT*)lpMemMgr,lpVad); //Insert into list or
tree.
else
    InsertIntoTree((__COMMON_OBJECT*)lpMemMgr,lpVad);

```

上述代码把找到的满足用户需求的虚拟区域，插入虚拟区域描述符表或二叉树。

```

//
//The following code reserves page table entries for the committed memory.
//
dwPteFlags = PTE_FLAGS_FOR_IOMAP;    //IO map flags,that is,this memory range
will not use hardware cache.
lpPhysical = lpStartAddr;

while(dwSize)
{
    if(!lpIndexMgr->ReservePage((__COMMON_OBJECT*)lpIndexMgr,
        lpStartAddr,lpPhysical,dwPteFlags))
    {
        PrintLine("Fatal Error : Internal data structure is not consist.");
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        goto __TERMINAL;
    }
    dwSize -= PAGE_FRAME_SIZE;
    lpStartAddr = (LPVOID)((DWORD)lpStartAddr + PAGE_FRAME_SIZE);
    lpPhysical  = (LPVOID)((DWORD)lpPhysical  + PAGE_FRAME_SIZE);
}

```

```
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
bResult = TRUE;    //Indicate that the whole operation is successfully.
```

跟注释指明的那样，上述代码完成了页表的预留，然后使用与线形地址相同的地址，来填充页表。因为一次预留的虚拟区域的长度，有可能是多个页面长度的倍数，因此上述代码是一个循环，每次循环预留一个页表项，直到 dwSize 递减为 0。

```
__TERMINAL:
if(!bResult)    //Process failed.
{
    if(lpVad)
        KMemFree((LPVOID)lpVad,KMEM_SIZE_TYPE_ANY,0L);
    if(lpPhysical)

PageFrameManager.FrameFree((__COMMON_OBJECT*)&PageFrameManager,
        lpPhysical,
        dwSize);
    return NULL;
}
return lpDesiredAddr;
}
```

上述代码完成最后的处理，若整个操作失败（bResult 为 FALSE），则释放所有已经申请的资源，并返回 NULL，否则，返回预留的虚拟区域的首地址。

需要注意的是，上述操作，凡涉及到对共享变量的操作，都是在互斥对象保护下进行的，确保同时只有一个线程在进行相关操作，否则可能会导致数据不一致。

## **VIRTUAL\_AREA\_ALLOCATE\_RESERVE**

当调用者使用该标志调用 VirtualAlloc 的时候，说明调用者仅仅想预留一部分虚拟地址空间，以备将来使用。

当使用该标志调用 VirtualAlloc 的时候，相关操作如下：

- 1、 向下舍入 lpDesiredAddr 到 FRAME\_PAGE\_SIZE 边界，并在 dwSize 上，增加舍入部分，然后向上舍入 dwSize 到 FRAME\_PAGE\_SIZE 边界；
- 2、 检查从 lpDesiredAddr 开始，长度为 dwSize 的虚拟内存区域，是否已经被分配，或者

与已经被分配的虚拟区域重叠；

- 3、如果没有分配，也没有重叠，则调用 KMemAlloc 函数，创建一个新的虚拟区域描述对象（\_\_VIRTUAL\_AREA\_DESCRIPTOR），根据调用参数等初始化该对象；
- 4、如果上述区域已经被分配，或者与现有的已经分配的区域重叠，则 VirtualAlloc 重新寻找一块连续区域，满足上述长度，如果能够找到，创建一个虚拟区域描述对象，并初始化，如果没找到，说明虚拟内存空间已经被消耗完毕，直接返回 NULL，调用失败；
- 5、把上述区域描述对象插入虚拟区域链表或者 AVL 树，增加 dwVirtualAreaNum，设置 dwAllocFlags 为 VIRTUAL\_AREA\_ALLOCATE\_RESERVE，并返回新创建的虚拟区域的初始地址。

上述功能的实现代码，与 VIRTUAL\_AREA\_ALLOCATE\_IO 类似，在此不做赘述，不同的是，VIRTUAL\_AREA\_ALLOCATE\_IO 预留了页表项，而当以该标志调用 VirtualAlloc 的时候，却没有预留页表项，仅仅返回预留的虚拟区域的首地址。这时候，若引用这个地址，会出现内存访问异常。

### **VIRTUAL\_AREA\_ALLOCATE\_COMMIT**

当使用该参数调用 VirtualAlloc 的时候，说明用户先前已经预留了虚拟内存空间（通过 VIRTUAL\_AREA\_ALLOCATE\_RESERVE 调用 VirtualAlloc 函数），本次调用的目的，是想为先前已经预留的虚拟地址空间，具体分配物理内存。这个时候，lpDesiredAddr 必须不能为 NULL，否则直接返回。

相关操作如下：

- 1、遍历虚拟区域列表或 AVL 树，查找 lpDesiredAddr 是否已经存在，如果不能找到，说明该地址没有被预留，则直接返回 NULL；
- 2、如果虚拟地址空间已经被预留，则判断预留大小，当前情况下，如果已经预留的空间的大小，比 dwSize 大，则以预留的虚拟地址空间尺寸为准，申请物理内存，否则（dwSize 大于预留的虚拟空间大小），返回 NULL，以指示操作失败；
- 3、调用 PageFrameManager 的 FrameAlloc 函数，分配物理内存。按照当前的实现方式，只调用一次 FrameAlloc 函数为虚拟内存分配物理内存，这样由于一次调用 FrameAlloc 函数，最多可以分配的物理内存是 8M（目前的实现），因此，若采用本标志调用 VirtualAlloc 函数，目标虚拟区域的大小不能大于 8M，否则会失败；
- 4、物理内存分配成功之后，调用 ReservePage，为新分配的物理内存，以及虚拟内存建立对应关系；
- 5、如果上述操作一切顺利，则设置虚拟区域描述符的 dwAllocFlags 值为 VIRTUAL\_AREA\_ALLOCATE\_COMMIT，返回 lpDesiredAddr，否则，返回 NULL，指示分配失败。

这种情况下，可实现一种所谓“按需分配”的内存分配策略，即开始的时候，为调

用者分配部分内存，比如，一个物理内存页，这个时候，如果用户访问了没有分配物理内存的虚拟内存地址空间，则会引发一个访问异常，系统的页面异常处理程序会被调用，这样异常处理程序，就会继续为用户分配需要的物理地址空间。

在当前的实现中，为提高系统的效率，没有采用这种按需分配的内存分配策略，而是直接按照用户需求的大小，分配内存页面，如果能够成功，则返回 `lpDesiredAddr`，指示操作成功，否则，返回 `NULL`，指示操作失败。这样就不会频繁的导致内存访问异常，导致系统效率大大下降。

## VIRTUAL\_AREA\_ALLOCATE\_ALL

当设定 `VIRTUAL_AREA_ALLOCATE_ALL` 标志调用 `VirtualAlloc` 的时候，其效果与以 `VIRTUAL_AREA_ALLOCATE_RESERVE` 和 `VIRTUAL_AREA_ALLOCATE_COMMIT` 联合调用效果相同。

当以该标志调用 `VirtualAlloc` 的时候，相关操作流程如下：

- 1、向下舍入 `lpDesiredAddr` 到 `FRAME_PAGE_SIZE` 边界，并对 `dwSize` 增加舍入数值，并向上舍入 `dwSize` 到 `PAGE_FRAME_SIZE` 边界；
- 2、检查虚拟区域链表或 AVL 树，以确定以 `lpDesiredAddr` 为起始地址，`dwSize` 为长度的虚拟内存区域是否已经分配，或者是否与已经分配的虚拟区域重合；
- 3、如果上述区域没有分配，也没有重合，则创建一个新的虚拟区域描述对象，根据 `lpDesiredAddr`、`dwSize` 等数值初始化该虚拟区域描述对象；
- 4、如果上述虚拟区域已经分配，或者与现有的虚拟区域重合，则 `VirtualAlloc` 重新寻找一块虚拟区域（长度为 `dwSize`），如果寻找成功，则设置 `lpDesiredAddr` 为新寻找的区域的起始地址，并创建虚拟区域描述对象，初始化；
- 5、把上述新创建的虚拟区域对象，插入虚拟区域链表，或 AVL 树，具体插入哪个数据结构，根据 `dwVirtualAreaNum` 决定，上述对虚拟区域链表或 AVL 树的检查、虚拟区域描述对象（`__VIRTUAL_AREA_DESCRIPTOR`）的创建、虚拟区域插入链表或 AVL 树等操作，构成一个原子操作（关闭中断、`SpinLock` 保护等），以保证链表或 AVL 树的完整性；
- 6、调用 `PageFrameManager` 的 `AllocFrame` 函数，分配一块大小可以容纳 `dwSize` 的物理内存区域，如果分配成功，把取得的物理内存的地址，存放在一个变量中，假设为 `lpPhysicalAddr`；
- 7、如果分配不成功，则重新调用 `AllocFrame` 函数，分配一页物理内存（`PAGE_FRAME_SIZE`），并存放在 `lpPhysicalAddr` 变量内，如果分配失败，转失败处理流程；
- 8、根据分配的物理内存的大小，调用 `PageIndexMgr` 的 `ReservePage` 函数，完成页表的填充（完成虚拟地址和物理地址的映射），如果分配的物理内存，大小等于或超过 `dwSize`，则以 `FRAME_PAGE_SIZE` 为单位，每次完成 `ReservePage` 函数后，递增

lpDesiredAddr 和 lpPhysicalAddr 变量（因为 ReservePage 每次填充一个页面），并递减 dwSize，直到 dwSize 为 0。如果分配的物理内存的尺寸小于 dwSize（实际上，只有 FRAME\_PAGE\_SIZE）大小，则只在第一次调用 ReservePage 的时候，给出物理内存，后续的调用，只给出虚拟内存地址，并设定 PTE\_FLAG\_NOTPRESENT 标志，以指明内存尚未分配，这个循环，也是直到 dwSize 递减到 0（FRAME\_PAGE\_SIZE 为递减单位）；

- 9、上述所有操作成功完成之后，设置目标虚拟区域描述符的 dwAllocFlags 标志为 VIRTUAL\_AREA\_ALLOCATE\_COMMIT，返回 lpDesiredAddr，作为成功标志，否则转失败处理流程（以下步骤）；
- 10、如果处理过程转到该步骤及以下步骤，说明操作过程中发生了错误，需要还原到系统先前的状况，并释放所有已经分配的资源；
- 11、检查是否已经分配虚拟区域描述符对象，如果已经分配，则从链表或 AVL 树中删除该对象，并释放该对象；
- 12、如果已经分配物理内存，则调用 PageFrameManager 的 FreeFrame 函数，释放物理内存；
- 13、如果出现预留页表项不成功的情况，则说明系统内部出现问题（数据不连续），这个时候直接给出严重警告，并停机。

上述功能的实现代码如下，为了方便理解，我们分段解释：

```
static LPVOID DoReserveAndCommit(__COMMON_OBJECT* lpThis,
                                LPVOID          lpDesiredAddr,
                                DWORD            dwSize,
                                DWORD            dwAllocFlags,
                                DWORD            dwAccessFlags,
                                UCHAR*           lpVaName,
                                LPVOID           lpReserved)
{
    __VIRTUAL_AREA_DESCRIPTOR* lpVad      = NULL;
    __VIRTUAL_MEMORY_MANAGER*  lpMemMgr   =
(__VIRTUAL_MEMORY_MANAGER*)lpThis;
    LPVOID                      lpStartAddr = lpDesiredAddr;
    LPVOID                      lpEndAddr   = NULL;
    DWORD                      dwFlags      = 0L;
    BOOL                        bResult      = FALSE;
    LPVOID                      lpPhysical  = NULL;
    __PAGE_INDEX_MANAGER*       lpIndexMgr  = NULL;
```

```

DWORD                                     dwPteFlags = NULL;

if((NULL == lpThis) || (0 == dwSize))    //Parameter check.
    return NULL;
if(VIRTUAL_AREA_ALLOCATE_ALL != dwAllocFlags)    //Invalidate flags.
    return NULL;
lpIndexMgr = lpMemMgr->lpPageIndexMgr;
if(NULL == lpIndexMgr)    //Validate.
    return NULL;

lpStartAddr = (LPVOID)((DWORD)lpStartAddr & ~(PAGE_FRAME_SIZE - 1));
//Round up to page.

lpEndAddr = (LPVOID)((DWORD)lpDesiredAddr + dwSize );
lpEndAddr = (LPVOID)((DWORD)lpEndAddr & (PAGE_FRAME_SIZE - 1)) ?
    (((DWORD)lpEndAddr & ~(PAGE_FRAME_SIZE - 1)) + PAGE_FRAME_SIZE -
1)
    : ((DWORD)lpEndAddr - 1)); //Round down to page.

dwSize = (DWORD)lpEndAddr - (DWORD)lpStartAddr + 1; //Get the actually
size.

lpVad = (LPVOID)0;
(__VIRTUAL_AREA_DESCRIPTOR*)KMemAlloc(sizeof(__VIRTUAL_AREA_DESCRIPTOR),
OR),
    KMEM_SIZE_TYPE_ANY); //In order to avoid calling KMemAlloc routine in the
    //critical section, we first call it here.
if(NULL == lpVad)    //Can not allocate memory.
{
    PrintLine("In DoReserveAndCommit: Can not allocate memory for VAD.");
    goto __TERMINAL;
}
lpVad->lpManager = lpMemMgr;
lpVad->lpStartAddr = NULL;

```

```

    lpVad->lpEndAddr      = NULL;
    lpVad->lpNext          = NULL;
    lpVad->dwAccessFlags   = dwAccessFlags;
    lpVad->dwAllocFlags     =    VIRTUAL_AREA_ALLOCATE_COMMIT;
//dwAllocFlags;
    __INIT_ATOMIC(lpVad->Reference);
    lpVad->lpLeft           = NULL;
    lpVad->lpRight          = NULL;
    if(lpVaName)
    {
        if(StrLen((LPSTR)lpVaName) > MAX_VA_NAME_LEN)
            lpVaName[MAX_VA_NAME_LEN - 1] = 0;
        StrCpy((LPSTR)lpVad->strName[0],(LPSTR)lpVaName);    //Set the virtual area's
name.
    }
    else
        lpVad->strName[0] = 0;
    lpVad->dwCacheFlags = VIRTUAL_AREA_CACHE_NORMAL;

```

上述代码与 VIRTUAL\_AREA\_ALLOCATE\_IO 相同，完成参数的检查、虚拟区域描述符的分配以及初始化等工作。

```

    lpPhysical =
PageFrameManager.FrameAlloc((__COMMON_OBJECT*)&PageFrameManager,
    dwSize,
    0L);    //Allocate physical memory pages.In order to reduce the
time

    //in critical section,we allocate physical memory here.
    if(NULL == lpPhysical)    //Can not allocate physical memory.
    {
        PrintLine("In DoReserveAndCommit: Can not allocate physical memory.");
        goto __TERMINAL;
    }

```

上述代码完成实际的物理内存分配功能，调用 `FrameAlloc` 函数，并以最终计算的 `dwSize` 为大小，来申请物理内存。若申请成功，则继续下一步的操作，否则，会打印出“无法分配物理内存”的信息，并跳转到该函数的最后，这样会导致该函数以失败告终，返回用户一个 `NULL`。

```
//
//The following code searches virtual area list or AVL tree,to check if the lpDesiredAddr
//is occupied,if so,then find a new one.
//
lpEndAddr = lpStartAddr;    //Save the lpStartAddr,because the lpStartAddr may
changed

                                //after the SearchVirtualArea_X is called.
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
if(lpMemMgr->dwVirtualAreaNum < SWITCH_VA_NUM) //Should search in the list.
    lpStartAddr =
SearchVirtualArea_l((__COMMON_OBJECT*)lpMemMgr,lpStartAddr,dwSize);
else //Should search in the AVL tree.
    lpStartAddr =
SearchVirtualArea_t((__COMMON_OBJECT*)lpMemMgr,lpStartAddr,dwSize);
if(NULL == lpStartAddr) //Can not find proper virtual area.
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    goto __TERMINAL;
}
```

上述代码查找虚拟区域描述符链表或二叉树，以试图找到一个满足需要的虚拟内存区域，若查找失败，则会导致该函数以失败返回。需要注意的是，该操作也尝试以用户提供的期望地址，为用户分配虚拟内存区域，若尝试失败，则选择另外一个满足大小的，但期望地址不是用户给出的地址的虚拟区域，返回给用户。

```
lpVad->lpStartAddr = lpStartAddr;
lpVad->lpEndAddr = (LPVOID)((DWORD)lpStartAddr + dwSize -1);
if(!(lpStartAddr == lpEndAddr)) //Have not get the desired area.
    lpDesiredAddr = lpStartAddr;
```

```

if(lpMemMgr->dwVirtualAreaNum < SWITCH_VA_NUM)
    InsertIntoList((__COMMON_OBJECT*)lpMemMgr,lpVad); //Insert into list or
tree.
else
    InsertIntoTree((__COMMON_OBJECT*)lpMemMgr,lpVad);

```

上述代码把符合用户需求的虚拟区域，插入到虚拟区域描述符链表或二叉树。

```

//
//The following code reserves page table entries for the committed memory.
//
dwPteFlags = PTE_FLAGS_FOR_NORMAL;    //Normal flags.

while(dwSize)
{
    if(!lpIndexMgr->ReservePage((__COMMON_OBJECT*)lpIndexMgr,
        lpStartAddr,lpPhysical,dwPteFlags))
    {
        PrintLine("Fatal Error : Internal data structure is not consist.");
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        goto __TERMINAL;
    }
    dwSize -= PAGE_FRAME_SIZE;
    lpStartAddr = (LPVOID)((DWORD)lpStartAddr + PAGE_FRAME_SIZE);
    lpPhysical  = (LPVOID)((DWORD)lpPhysical  + PAGE_FRAME_SIZE);
}
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
bResult = TRUE;    //Indicate that the whole operation is successfully.

```

上述代码完成虚拟内存地址和物理内存地址的映射，即调用 `PageIndexManager` 提供的接口函数，创建页表。操作成功完成后，设置 `bResult` 为 `TRUE`，这表示该函数最终操作成功。

```

__TERMINAL:
if(!bResult)    //Process failed.
{
    if(lpVad)
        KMemFree((LPVOID)lpVad,KMEM_SIZE_TYPE_ANY,0L);
    if(lpPhysical)

PageFrameManager.FrameFree((__COMMON_OBJECT*)&PageFrameManager,
        lpPhysical,
        dwSize);
    return NULL;
}
return lpDesiredAddr;
}

```

上述代码是该函数的最后处理代码，根据 **bResult** 的结果，做不同的处理，若 **bResult** 为 **TRUE**，则说明一切操作成功，返回成功预留的虚拟地址，否则，释放一切内存资源，包括虚拟区域描述符占用的资源、申请的物理页面等，然后返回 **NULL**。

需要注意的是，上述操作也可采用“按需分配”的原则，即如果用户请求的内存数量太大，则暂缓分配全部物理内存，而是只分配一个物理页面，这样后续用户访问未分配物理页面的虚拟内存时，会引发一个访问异常，这样在异常处理程序中，会继续为没有分配到物理内存的虚拟内存，分配物理页面。

需要注意的是，在当前版本的实现中，考虑到系统效率等因素，没有实现“按需分配”的内存分配策略，而是采用“一次全部分配”的原则，即一次分配所有需要的物理内存，如果成功，则设置页表，并返回成功标志，否则直接返回失败标志。这个时候，用户应用程序可以尝试改变请求的大小，再次调用 **VirtualAlloc** 函数。

## VirtualFree

该函数是 **VirtualAlloc** 的反向操作，用于释放调用 **VirtualAlloc** 函数分配的虚拟区域。该函数执行流程如下：

- 1、根据调用者提供的虚拟地址，查找虚拟区域列表或 AVL 树，找到对应的虚拟区域描述符；
- 2、如果不能找到，说明该区域不存在，直接返回；

- 3、根据 dwAllocFlags 的不同取值，完成不同的操作：
- 4、如果 dwAllocFlags 的值是 VIRTUAL\_AREA\_ALLOCATE\_RESERVE，则仅仅从链表或 AVL 树中删除该虚拟区域描述符，并释放该虚拟区域描述符占用的内存，直接返回；
- 5、如果 dwAllocFlags 的值是 VIRTUAL\_AREA\_ALLOCATE\_IO，则从链表中删除虚拟区域描述符，调用 PageIndexMgr 的 ReleasePage 函数，释放预留的页表，最后释放虚拟区域描述符占用的物理内存，并返回；
- 6、如果 dwAllocFlags 的值是 VIRTUAL\_AREA\_COMMIT，则说明已经为该虚拟区域分配了实际的物理内存，这种情况下，就需要释放物理内存。首先，从链表或 AVL 树中删除虚拟区域描述符对象，调用 PageIndexMgr 的 GetPhysicalAddr 函数，获取虚拟地址对应的物理地址，然后调用 PageFrameManager 的 FreeFrame 函数，释放物理页面，最后依次调用（以 FRAME\_PAGE\_SIZE 为递减单位递减 dwSize，直到 dwSize 为 0）PageIndexMgr 的 ReleasePage 函数，释放预留的页表，所有这些操作完成之后，函数返回。

需要注意的是，上述所有操作，包括对虚拟区域链表或 AVL 树的删除、页表的释放等操作，都需要在一个原子操作内完成，以免发生系统级的数据结构不一致。实现该部分功能的代码比较简单，在此不作详细描述。

### GetPdeAddress

该函数返回页目录的物理地址，用于设置 CPU 的特定寄存器，比如，针对 Intel 的 IA32 构架 CPU，需要设置 CR3 寄存器，这时候就需要知道页目录的物理地址。

该函数直接读取 PageIndexMgr 的页目录物理地址，并返回给调用者。



## 第六章线程本地堆的实现

——Heap's implementation

## 6.1 Heap 概述

在前面的介绍中提到，一个用户线程，可以通过两种方式获取内存：调用 `KMemAlloc` 函数，从内核内存空间中分配内存，这种方式下，可以申请页面尺寸大小（在 IA32 构架下，为 4K）的内存，也可以申请任何大小的内存。但一般情况下，核心内存是供操作系统核心和设备驱动程序使用的，不建议用户应用程序直接申请。另外一种方式是调用 `VirtualAlloc`，从整个系统的线性空间中分配内存，这种方式下，需要在调用 `VirtualAlloc` 函数的时候，提供一个 `VIRTUAL_ALLOC_ALL` 标志，这样就使得操作系统把申请的线性地址空间区域，跟物理内存对应起来，从而可以当作常规内存使用（若申请的线性地址空间区域，没有对应的物理内存，则在访问这些内存的时候，会导致异常）。但采用 `VirtualAlloc` 来分配内存，最小尺寸也是页面尺寸（IA32 构架下是 4K），不能申请更小尺寸的内存。

因此，上述两种内存分配方式，都不适合应用程序直接采用，一个比较可行的做法就是，另外实现一个内存分配器，这个分配器通过 `VirtualAlloc` 函数，从线性空间中申请大块内存，然后作为一种资源自己管理，并提供用户接口，供应用程序调用来申请小块内存。这种管理内存的方式，我们称作“堆”（heap）。在本章中，我们就对堆的实现，进行详细描述。

## 6.2 堆的功能需求定义

在实现具体的堆功能前，需要详细考虑如何定义堆的功能，在软件工程中，所谓“需求分析”。这个过程是十分关键的，在这个过程中，需要把待实现的系统（或一个简单的功能模块）的具体功能，进行完整、详尽的定义和描述，且一旦固定（比如，通过了技术评审），将不被改变。这样在该系统的实现过程中，可避免频繁修改功能需求，导致的大量反复工作。在堆的实现中，我们充分考虑用户程序的方便，并向标准的 C 运行库靠拢，我们这样定义堆功能：

- 1、堆是基于线程实现的，即一个堆只属于一个线程，但一个线程可以具备多个堆；
- 2、为了管理和实现上的方便，采用一个统一的接口—堆管理器（`HeapManager`），来管理系统中所有的堆；
- 3、堆是一个内存池，根据用户需要，从系统中“批发”申请内存，并“零售”给用户；
- 4、除了堆的创建、销毁接口之外，堆管理器还应该提供“内存分配”和“内存释放”两个接口，供应用程序调用；

- 5、应用程序调用“内存分配”和“内存释放”函数的时候，所操作的堆对象，只能是当前线程的堆对象；
- 6、其中，“内存分配”和“内存释放”接口函数所需要的参数，能够与标准 C 运行库函数 `malloc` 和 `free` 的参数相互映射，这样可通过函数封装或宏定义的方式，来通过堆的“内存释放”和“内存分配”函数，来实现 `free` 和 `malloc` 函数；
- 7、在存在多个堆的情况下，应该有一个缺省堆，来对应 `malloc` 和 `free` 函数，这样这两个函数，可以从缺省堆中分配内存；
- 8、除非出现系统内存不足的情况，否则堆功能函数“内存申请”和“内存释放”函数不能失败。

其中，上述第一条的含义在于，一个线程可能有多个堆对象，而一个堆对象只能属于一个线程。这样的实现，可以具有更大的灵活性，一个线程（或用户应用程序）可能是由若干功能模块组成的，这些功能模块可能互不交叉，比如一个文字处理系统的编辑模块和打印模块等，这样为了实现上的一致性和清晰性，每个功能模块可以单独创建一个自己的内存堆，在申请内存的时候，可从自己的内存堆中申请。当然，这仅仅是一种可选的实现，一个线程的不同模块，完全可以公用一个堆，完全可以调用 `malloc` 和 `free` 函数（这些函数都是作用在线程的缺省堆上的）来实现内存管理。

在上述第三条的功能定义中，堆作为一个内存池，在初始化（创建过程中）的时候，就需要事先从系统中申请一部分内存（比如，16K），作为一个内存池，一旦用户有内存分配需求，就从该内存池中进行分配。若内存池中的内存分配完毕，则需要通过调用 `VirtualAlloc` 函数，从系统中再次申请内存，并加入内存池中。若用户释放内存，则释放的内存会被重新加入内存池，在积累到一定程度的时候，堆对象会堆内存池进行清理，把暂时用不到的大块内存，返回系统。这样做的一个好处是，可以实现系统内存的按需分配，不至于出现大规模的内存浪费现象。

上述第五条定义中，应用程序只能操作自己的堆，而不能从其它应用程序（线程）的堆中申请内存。这样的实现是符合逻辑的，且实现起来相对简便，无需考虑多线程之间的同步。另，在中断处理程序中，也不能调用堆功能函数，来从堆中分配内存，而应该调用 `KMemAlloc` 或 `VirtualAlloc` 来分配内存。

`malloc` 和 `free` 函数，是标准 C 运行库提供的接口函数，实现这两个函数，对于代码的移植（把其它操作系统上的应用程序代码，移植到 `Hello China` 上）非常有帮助。而且一般的程序员都十分熟悉这两个接口函数，鉴于此，在 `Hello China` 当前的堆实现中，通过引入一个默认堆（缺省堆）的对象，通过标准的堆操作接口，实现了这两个 C 运行库

函数。

在上述第八条功能描述中，实际上是提出了一种“按需分配”的实现方式，即开始的时候，堆对象先从系统中申请少量内存，作为内存池，等该内存池分配完毕，或用户提出的申请的内存尺寸大于这个内存池的时候，堆对象再调用 `VirtualAlloc` 函数，从系统空间中申请额外内存池。在这种实现方式下，堆管理器可以根据需要来申请系统内存池，从而做到尽可能的节约系统内存。堆对象也可以被认为是一个内存申请代理机构和缓冲机构，对于数量小的内存块申请，堆直接从本地内存池中分配，而对于一些大块内存的申请，堆管理器也可以作为一个中专代理，从系统中申请。当然，建议应用程序开发者，在需要数量比较大的内存的时候（比如，超过页面尺寸 4K 大小），直接调用 `VirtualAlloc` 函数申请，因为经过堆申请，实际上堆也可能调用 `VirtualAlloc`，这样经过了堆的中专，会导致性能的少量下降。当然，对于小块的内存申请，若堆的内存池可以满足要求，则不会存在这个问题。

## 6.3 堆的实现概要

按照上述定义的功能，我们定义堆管理器对象，作为堆功能的对外接口。堆管理器对象定义如下：

```
BEGIN_DEFINE_OBJECT(__HEAP_MANAGER)
    __HEAP_OBJECT*          (*CreateHeap)(DWORD dwInitSize); //Create a
heap object.
    VOID                    (*DestroyHeap)(__HEAP_OBJECT*); //Destroy a
heap object.
    VOID                    (*DestroyAllHeap)(); //Destroy all heaps.
    LPVOID                  (*HeapAlloc)(__HEAP_OBJECT*,DWORD); //Allocate
memory from heap.
    VOID                    (*HeapFree)(LPVOID,__HEAP_OBJECT*); //Free the
memory block.
END_DEFINE_OBJECT()
```

其中，`__HEAP_OBJECT` 是一个堆对象，该对象定义如下：

```

BEGIN_DEFINE_OBJECT(__HEAP_OBJECT)
    __KERNEL_THREAD_OBJECT*      lpKernelThread;
    __FREE_BLOCK_HEADER           FreeBlockHeader;
    __VIRTUAL_AREA_NODE*         lpVirtualArea;
    __HEAP_OBJECT*                lpPrev;
    __HEAP_OBJECT*                lpNext;
END_DEFINE_OBJECT()

```

在堆管理器对象提供的接口中，`CreateHeap` 用于创建一个堆对象，`dwInitSize` 是初始的缓冲池的大小，即在创建堆的时候，从系统中申请多少内存，作为堆的内存池。`DestroyHeap` 用于销毁一个堆对象，而 `DestroyAllHeap` 函数，则是销毁当前线程的所有堆对象。这个操作一般是在线程运行结束后，被操作系统调用，用来销毁线程创建的堆对象的。顾名思义，`HeapAlloc` 和 `HeapFree` 两个接口用于完成内存的分配和释放。其中，这两个函数除了接受一个 `DWORD` 类型的参数，用于指明需要申请的内存数量外，还接受一个堆对象指针参数，这个堆对象指针指明了从哪个堆中分配内存。在当前的实现中，`HeapAlloc` 只能从当前线程（调用这个函数的线程）的堆对象中分配内存。

堆对象管理器仅仅是一个操作接口，是为了采用面向对象的实现思路而引入的。

按照堆的功能描述，一个线程可能有多个堆，而一个堆只能属于一个线程。因此，必须有一种机制，可以管理多个堆的情况。在当前的实现中，修改了 `__KERNEL_THREAD_OBJECT` 对象（核心线程对象，用于保存线程的特定信息，每个核心线程对应这样一个对象），在该对象的定义中，添加了两个变量：`lpHeapObject` 和 `lpDefaultHeap`，这两个变量的类型，都是 `LPVOID`，之所以这样安排，是为了不把堆对象的特定数据结构定义，引入核心线程的实现中，这样可保持代码的清晰和独立。其中，第一个变量指向一个堆对象链表，该链表把当前线程中的所有堆对象连接在了一起。在 `CreateHeap` 函数被调用的时候，该函数就创建一个堆对象，并插入该堆对象链表。第二个变量 `lpDefaultHeap`，则指向了一个缺省的堆对象，所谓的缺省堆对象，就是 `malloc` 函数和 `free` 函数所操作的堆对象。

在当前的实现中，采用空闲链表算法，来完成堆对内存池的管理。所谓空闲链表，指的是把所有空闲的内存块，连接在一起，作为一个统一的空闲内存池，在分配内存的时候，遍历整个空闲链表，选择一块合适的空闲内存块，返回给调用者，并把该空闲内存块从空闲链表中删除。在释放内存的时候，释放函数把释放的内存块再重新插入空闲

链表。这样，每个堆对象需要维护一个空闲链表（FreeBlockHeader 变量就是该空闲链表的头指针），用于完成空闲内存块的管理。开始的时候（堆被创建的时候），堆对象申请的初始内存池，作为空闲链表中的第一个对象（也是唯一一个空闲块）被插入空闲链表，随着内存分配的持续，空闲链表中的元素（空闲块）数量，会增加或减少。

堆对象除了维护一个空闲链表外，还维护一个虚拟区域对象链表，每个虚拟区域，实际上就是系统线性地址空间中的一块连续区域，而且这块区域已经跟实际的物理内存完成了映射（CPU MMU 的内存管理数据结构，已经被成功填充）。虚拟区域列表实际上就是堆对象的内存池，堆对象初始化的时候，会申请一块特定大小的虚拟内存区域，作为内存池，随着分配的持续，申请的虚拟内存区域可能已经分配完了，或者用户请求的内存大小，已经超过了剩余的可分配的内存空间，这时候，堆对象会再次调用 VirtualAlloc 函数，申请额外的虚拟区域对象。申请成功后，将把该虚拟区域对象插入虚拟区域列表。在堆对象的定义（\_\_HEAP\_OBJECT）中，lpVirtualArea 就是虚拟区域链表的头指针，需要注意的是，虚拟区域链表是一个单项链表，每个链表元素是一个 \_\_VIRTUAL\_AREA\_NODE 结构，该结构定义如下：

```
BEGIN_DEFINE_OBJECT(__VIRTUAL_AREA_NODE)
    LPVOID                lpStartAddress;    //The start address.
    DWORD                 dwAreaSize;        //Virtual area's size.
    __VIRTUAL_AREA_NODE* lpNext;             //Pointing to next node.
END_DEFINE_OBJECT()
```

可见，该结构的定义非常简单，仅仅保留了虚拟区域的起始地址，以及该虚拟区域的大小。

按照上述实现方式，一个核心线程的堆对象组成一个链表，其中，每个堆对象中，又维护两个链表：空闲块链表（双向）和虚拟区域链表（单项），如下图：

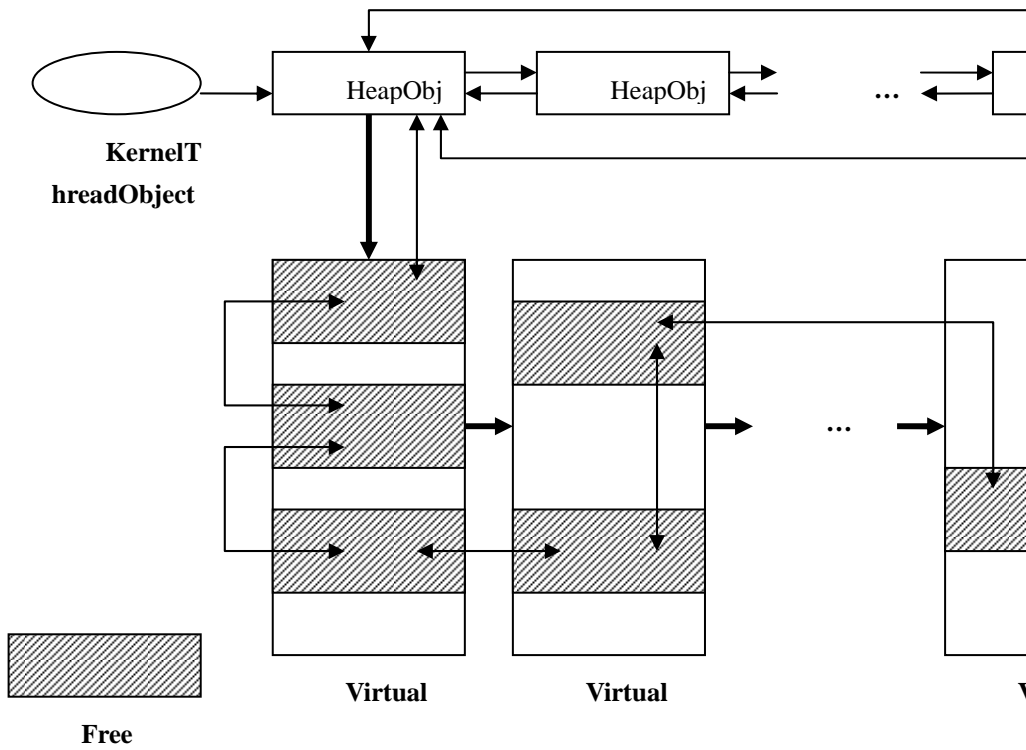


图 6-1 线程本地堆的整体数据结构

需要注意的是，空闲链表的空闲块，跟虚拟区域链表中的区域是重合的，这很容易理解，因为空闲块是从虚拟区域中分配的，虚拟区域作为空闲块的原始资源。

对于空闲链表算法，相对比较简单，基本思路就是，把系统中的所有空闲内存块，通过链表的方式串连在一起，开始的时候，空闲链表中只有一块空闲块，那就是最初申请的虚拟区域。对于内存分配操作，该算法做如下动作：

- 1、从空闲链表头部开始，依次检查空闲链表中的空闲块，大小是否满足要求的尺寸；
- 2、若能够找到这样的一块空闲块，则判断该空闲块的大小，是否应该拆分。在空闲内存块比用户的申请尺寸大不太多（当前情况下，定义为 16 字节）的情况下，就把整个空闲块分配给用户，否则把找到的空闲块进行拆分，把其中剩余的部分再次插入

空闲链表，然后把拆分出来的另外一块，分配给用户；

- 3、若无法找到满足要求的空闲块，则再次调用 `VirtualAlloc` 函数，从系统空间中申请一块虚拟区域，然后把这块虚拟区域当作空闲块对待，从中摘取头部的一部分，返回用户，然后把剩余的部分（若用户申请的内存足够大，则整个虚拟区域直接返回用户），当作一块空闲块，插入空闲链表。当然，若调用 `VirtualAlloc` 失败，则会返回用户一个 `NULL` 指针，以指明本次操作失败。

可以看出，上述空闲链表采用的是首次适应算法。所谓首次适应算法，即从开始遍历空闲链表，只要找到一块尺寸大于要求的空闲块，就停止继续查找，直接把这块内存块返回给用户。这样做的优点是，实现起来相对方便，且效率高，当然，首次适应算法也有一个缺点，就是随着分配的深入，空闲链表中可能出现大量的“碎片”（尺寸很小的空闲块），这样一旦遇到申请内存数量很大的请求，就可能失败。但这样的分析只是在理论上的，实际上，许多成熟的计算机操作系统，都采用了首次适应算法，工作得都很好。而且在 `Hello China` 的实现中，对于堆，只是为了满足少量内存的申请而实现的，对于大块内存的分配，可直接调用 `VirtualAlloc` 函数来分配，这样就可避免首次适应算法带来的问题。

对于释放操作，在当前的实现中，分两步进行：

- 1、把释放的内存块，插入空闲链表；
- 2、调用一个合并操作，把空闲链表中的相互邻接的小空闲块，合并成一个大的空闲块。

其中，第一个步骤比较简单，只需要根据待释放内存的物理地址，找到该块空闲内存块的控制头，然后把该空闲内存块插入空闲链表即可。对于空闲块的合并操作，当前的实现中，是以虚拟区域为单位进行操作的。因为一个堆对象，可能申请了多块虚拟内存区域（这些区域被连接成单向链表），而一块空闲内存块，只属于一块虚拟区域，这样在释放内存的时候，只需根据待释放的内存地址，找到该空闲内存块所属的虚拟区域，然后从虚拟区域的开头，来开始空闲块的合并工作。需要注意的是，对于空闲块的合并，是按照地址相邻（而不是空闲块在空闲链表中相邻，两块空闲块，其在内存中的位置收尾相接，但在空闲链表中的位置可能不会收尾相接）进行的，算法如下：

- 1、以虚拟区域的第一块内存块为起始（虚拟区域的起始地址，对应于该虚拟区域所包含的第一块内存块的控制头地址），作为当前内存块，判断该块是否空闲（通过判断控制头的一个标志字段），若不空闲，则把当前内存块更新为与当前内存块相邻接（地址邻接）的下一块内存块，这可以通过在当前内存块指针上，增加当前内存块的大小（可从控制头中获取）；
- 2、判断当前内存块的相邻内存块是否空闲，若不空闲，则更新当前内存块为相邻内存

- 块，并从上述步骤（1）重新开始操作；
- 3、若当前内存块的相邻内存块空闲，则进行一个合并动作，合并操作比较简单，只需要把当前内存块的相邻内存块从空闲链表中删除，然后在当前内存块的“大小”字段上，增加当前相邻内存块的大小即可（实际上还需要增加相邻内存块的控制头大小），如下图：

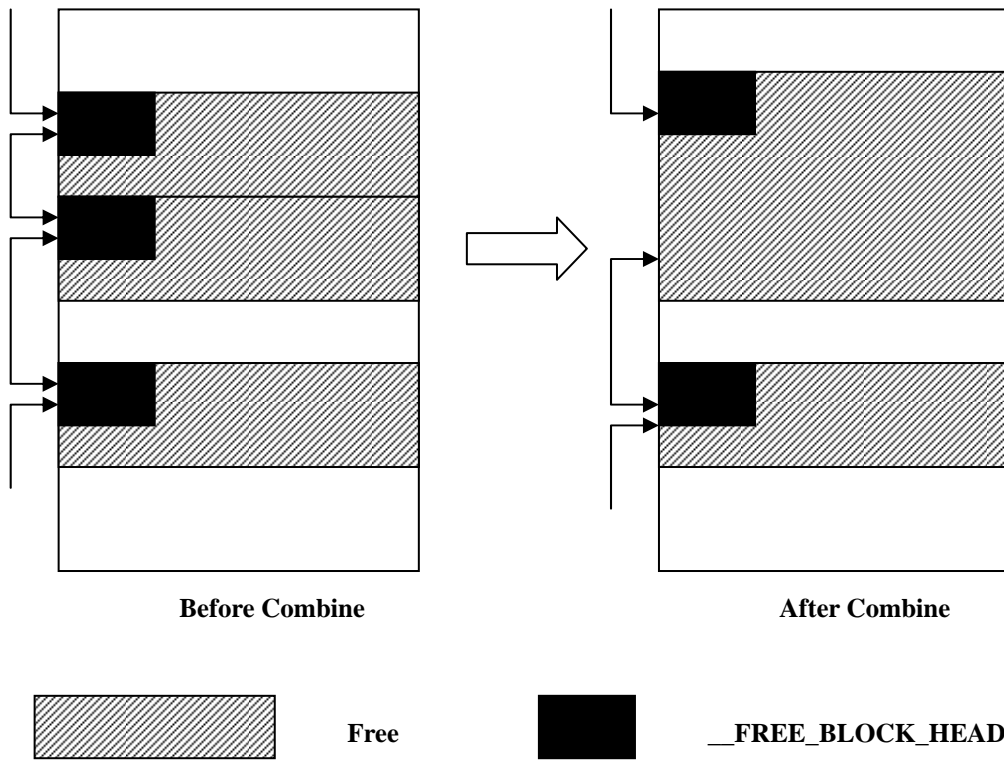


图 6-2 空闲内存块的合并操作

- 4、重复上述动作，直到当前内存块地址到达虚拟区域的尾部（这时，对于当前虚拟区域的合并操作完成）。

把对空闲块的合并操作，限制在一个虚拟区域内部，可以达到增加效率的目的，因为无需遍历整个空闲链表。

从上述的描述中可以看出，在当前 **Hello China** 对堆的实现中，由于采用了空闲链表算法，对内存的分配和回收，时间是无法预测的，在理想的情况下，可能很快就能完成内存的分配和释放，但如果空闲链表很长，而且“碎片”众多的情况下，分配和回收操作都可能耗费较长的时间。因此，基于这种实现，对于一些对时间要求非常严格的应用，可能无法满足要求，但对于大多数的非“硬实时”的应用，这样的实现是足够的。若程序员面对的是对时间要求十分苛刻的硬实时系统，则可考虑实现自己的内存分配器，这时候，只需要调用 **VirtualAlloc** 函数，事先获得一个内存池，然后再这个内存池的基础上，实现能够满足要求的分配算法。

对于空闲块的管理，是通过一个空闲控制块结构来进行的，该结构定义如下：

```
BEGIN_DEFINE_OBJECT(__FREE_BLOCK_HEADER)
    DWORD                dwFlags;                //Flags.
    DWORD                dwBlockSize;            //The size of this block.
    __FREE_BLOCK_HEADER* lpPrev;                //Pointing to previous free
block.
    __FREE_BLOCK_HEADER* lpNext;                //Pointing to next free
block.
END_DEFINE_OBJECT()
```

在该结构中，记录了空闲块的尺寸，并提供了一个标志字段，来标志当前块的状态，比如空闲或占用。若当前块的状态为占用，则说明当前内存块已经分配给了用户应用程序，不在空闲链表之中。**lpPrev** 和 **lpNext** 指针把空闲块连接在双向空闲链表中。

对于每个内存块，都预留开始的 16 字节，作为控制头，如下所示：

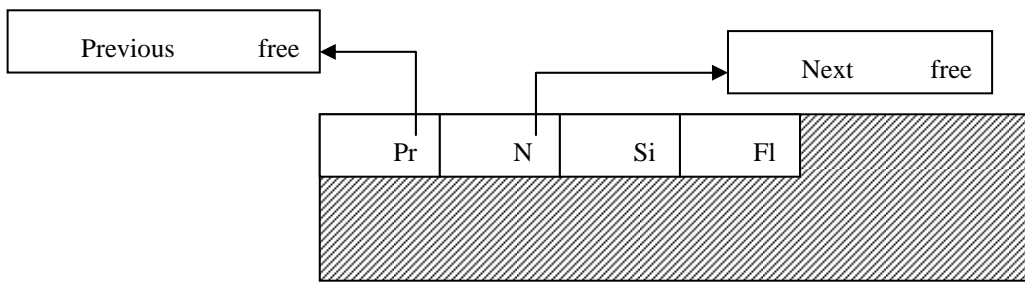


图 6-3 空闲内存块的控制头结构及位置

这样，在进行内存分配的时候，只需要从空闲链表中找到一块合适的空闲块，这时候的块指针，指向的是控制头基地址，在返回用户的时候，只需要在控制头地址基础上，增加 16（控制头结构长度）即可，当然，还需要修改内存块的标记字段。在释放内存的时候，根据用户提供的地址，减少 16 字节，就可获得控制头的基地址，然后把控制头中的标记字段修改为空闲，并插入空闲链表即可。

## 6.4 堆的详细实现

下面我们对当前版本 Hello China 的堆的详细实现进行描述。在当前的实现中，对于堆的管理，是通过一个堆管理器（Heap Manager）进行的，堆管理器提供五个接口函数：创建堆函数（CreateHeap）、销毁堆函数（DestroyHeap）、销毁所有堆（DestroyAllHeap）、堆内存分配函数（HeapAlloc）和堆内存释放函数（HeapFree）。

### 6.4.1 堆的创建

堆的创建过程比较简单，主要过程如下：

1、分配一块虚拟区域，虚拟区域的大小，根据用户提供的参数 dwInitSize 来决定，如果 dwInitSize 大于预定义的 DEFAULT\_VIRTUAL\_AREA\_SIZE（当前定义为 16K），则按照 dwInitSize 申请内存，否则按照 DEFAULT\_VIRTUAL\_AREA\_SIZE 来申请内存；

- 2、创建一个虚拟区域头节点，来管理刚刚申请的虚拟区域对象；
- 3、申请核心内存（KMemAlloc），创建一个堆对象；
- 4、把虚拟区域节点插入堆的虚拟区域链表；
- 5、把申请的虚拟区域，作为第一块空闲内存块，插入空闲链表；
- 6、然后把上述初始化完成的堆对象，插入当前线程的堆对象链表；
- 7、如果上述所有操作成功，则返回堆的起始地址，否则返回 NULL，指示失败。

代码如下所示，为了方便，我们分段解释：

```
static __HEAP_OBJECT* CreateHeap(DWORD dwInitSize)
{
    __HEAP_OBJECT*          lpHeapObject    = NULL;
    __HEAP_OBJECT*          lpHeapRoot      = NULL;
    __VIRTUAL_AREA_NODE*    lpVirtualArea   = NULL;
    __FREE_BLOCK_HEADER*    lpFreeHeader    = NULL;
    LPVOID                  lpVirtualAddr   = NULL;
    BOOL                    bResult         = FALSE;
    DWORD                   dwFlags         = 0;

    if(dwInitSize > MAX_VIRTUAL_AREA_SIZE) //Requested size too big.
        return NULL;
    if(dwInitSize < DEFAULT_VIRTUAL_AREA_SIZE)
        dwInitSize = DEFAULT_VIRTUAL_AREA_SIZE;

    //
    //Now,allocate the virtual area.
    //
    lpVirtualAddr = GET_VIRTUAL_AREA(dwInitSize);
    if(NULL == lpVirtualAddr) //Can not get virtual area.
        goto __TERMINAL;
    lpFreeHeader = (__FREE_BLOCK_HEADER*)lpVirtualAddr;
    lpFreeHeader->dwFlags      = BLOCK_FLAGS_FREE;
    lpFreeHeader->dwBlockSize  = dwInitSize - sizeof(__FREE_BLOCK_HEADER);
```

//Caution!!!

上述代码调用 VirtualAlloc(GET\_VIRTUAL\_AREA 实际上是 VirtualAlloc 的宏定义), 申请一块虚拟区域, 该虚拟区域的大小, 为 dwInitSize 和 DEFAULT\_VIRTUAL\_AREA\_SIZE 中最大者, 然后把申请的虚拟区域看作是一块空闲内存块, 并初始化其头部。需要注意的是, 这块空闲块的大小, 是虚拟区域的大小减去空闲块控制头的大小 (16 字节), 因为要考虑空闲块控制头所占用的大小。

```
//
//Now,create a virtual area node object,to manage virtual area.
//
lpVirtualArea = (__VIRTUAL_AREA_NODE*)GET_KERNEL_MEMORY(
    sizeof(__VIRTUAL_AREA_NODE));
if(NULL == lpVirtualArea) //Can not get memory.
    goto __TERMINAL;
lpVirtualArea->lpStartAddress = lpVirtualAddr;
lpVirtualArea->dwAreaSize      = dwInitSize;
lpVirtualArea->lpNext          = NULL;
```

上述代码创建了一个虚拟区域节点对象, 这个节点对象的唯一用途, 就是把虚拟区域连接到堆的虚拟区域链表中。

```
lpHeapObject =
(__HEAP_OBJECT*)GET_KERNEL_MEMORY(sizeof(__HEAP_OBJECT));
if(NULL == lpHeapObject) //Can not allocate memory.
    goto __TERMINAL;
lpHeapObject->lpVirtualArea = lpVirtualArea; //Virtual area node
list.
lpHeapObject->FreeBlockHeader.dwFlags |= BLOCK_FLAGS_FREE;
lpHeapObject->FreeBlockHeader.dwFlags &= ~BLOCK_FLAGS_USED;
lpHeapObject->FreeBlockHeader.dwBlockSize = 0;
lpHeapObject->lpPrev = lpHeapObject; //Pointing to itself.
lpHeapObject->lpNext = lpHeapObject; //Pointing to itself.
```

```

//__ENTER_CRITICAL_SECTION(NULL,dwFlags); //Critical section here.
lpHeapObject->lpKernelThread      = CURRENT_KERNEL_THREAD;
lpHeapRoot = (__HEAP_OBJECT*)CURRENT_KERNEL_THREAD->lpHeapObject;
if(NULL == lpHeapRoot) //Has not any heap yet.
{
    CURRENT_KERNEL_THREAD->lpHeapObject = (LPVOID)lpHeapObject;
}
else //Has at least one heap object,so insert it into the list.
{
    lpHeapObject->lpPrev = lpHeapRoot->lpPrev;
    lpHeapObject->lpNext = lpHeapRoot;
    lpHeapObject->lpNext->lpPrev = lpHeapObject;
    lpHeapObject->lpPrev->lpNext = lpHeapObject;
}
//__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

```

上述代码创建了一个堆对象（\_\_HEAP\_OBJECT），并把该堆对象初始化。其中，FreeBlockHeader 是空闲链表的头节点，这个头节点仅仅是用来完成空闲块的连接，不代表任何空闲块，因此其尺寸设置为 0。完成堆对象的创建，并初始化后，把创建的堆对象插入当前线程的堆链表。需要注意的是，由于堆是基于线程创建的，不存在与其它线程竞争资源的情况，而且当前线程对象的堆链表，也不会被中断处理程序修改，因此上述代码无需保护。

下面的代码，把申请的虚拟区域，加入了当前堆的虚拟区域列表，然后返回返回创建的堆的基地址。当然，如果上述处理中发生错误，则会导致该函数失败，在这种情况下，函数返回前，需要撤销已经申请的资源。这种采用事务式的处理方式，在 Hello China 的实现中非常常见。

```

//
//Now,add the virtual area into the heap object's free list.
//
lpFreeHeader->lpPrev = &(lpHeapObject->FreeBlockHeader);
lpFreeHeader->lpNext = &(lpHeapObject->FreeBlockHeader);

```

```

lpHeapObject->FreeBlockHeader.lpPrev      = lpFreeHeader;
lpHeapObject->FreeBlockHeader.lpNext      = lpFreeHeader;

bResult = TRUE;    //The whole operation is successful.

__TERMINAL:
if(!bResult)      //Failed.
{
    if(lpVirtualAddr) //Should release it.
        RELEASE_VIRTUAL_AREA(lpVirtualAddr);
    if(lpHeapObject)
        RELEASE_KERNEL_MEMORY((LPVOID)lpHeapObject);
    if(lpVirtualArea)
        RELEASE_KERNEL_MEMORY((LPVOID)lpVirtualArea);
    lpHeapObject = NULL; //Should return a NULL flags.
}
return lpHeapObject;
}

```

## 6.4.2 堆的销毁

堆的销毁分两种情况：一种情况是销毁单个堆，另外一种情况是，销毁当前线程的所有堆。其中，第二种情况，一般在线程结束的时候，由线程结束收尾函数（`KernelThreadWrapper` 函数）调用，用来释放当前线程尚未释放的所有堆对象。对于第一种情况，一般是由应用程序编写者调用，用来撤销自己创建的堆。

其中，销毁当前线程所有堆（`DestroyAllHeap`）的实现，也是调用了销毁单个堆的实现算法，因此，在此只简单介绍销毁单个堆的实现代码。对于堆的撤销操作，比较简单，所需要完成的，只是内存资源的回收而已。目前的实现中，在 `DestroyHeap` 的实现中，完成下列动作：

- 1、从当前线程的堆链表中，删除要销毁的堆对象（该对象作为函数的参数传递）；
- 2、释放该堆对象申请的所有虚拟区域；
- 3、然后把虚拟区域链表所占用内存释放，即虚拟区域节点占用内存；

4、最后，释放堆对象本身。

代码如下：

```
static VOID DestroyHeap(__HEAP_OBJECT* lpHeapObject)
{
    __VIRTUAL_AREA_NODE*      lpVirtualArea  = NULL;
    __VIRTUAL_AREA_NODE*      lpVirtualTmp   = NULL;
    LPVOID                     lpVirtualAddr  = NULL;
    DWORD                      dwFlags       = 0L;

    if(NULL == lpHeapObject) //Parameter check.
        return;

    if(lpHeapObject == lpHeapObject->lpNext) //Only one heap object in current thread.
    {
        __ENTER_CRITICAL_SECTION(NULL,dwFlags);
        lpHeapObject->lpKernelThread->lpHeapObject = NULL;
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    }
    else //Delete itself from the kernel thread's heap list.
    {
        __ENTER_CRITICAL_SECTION(NULL,dwFlags);
        if(lpHeapObject->lpKernelThread->lpHeapObject == lpHeapObject)
        {
            lpHeapObject->lpKernelThread->lpHeapObject =
(LPVOID)lpHeapObject->lpNext;
        }
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);

        lpHeapObject->lpPrev->lpNext = lpHeapObject->lpNext;
        lpHeapObject->lpNext->lpPrev = lpHeapObject->lpPrev;
    }

    lpVirtualArea = lpHeapObject->lpVirtualArea;
```

```
while(lpVirtualArea)
{
    lpVirtualTmp = lpVirtualArea;
    lpVirtualArea = lpVirtualArea->lpNext;
    RELEASE_VIRTUAL_AREA(lpVirtualTmp->lpStartAddress);    //Release the
virtual area.
    RELEASE_KERNEL_MEMORY((LPVOID)lpVirtualTmp);
}

//
//Now,should release the heap object itself.
//
RELEASE_KERNEL_MEMORY((LPVOID)lpHeapObject);

return;
}
```

### 6.4.3 堆内存申请

HeapAlloc 函数完成堆内存的申请，该函数接受两个参数：目标堆对象和待申请内存的大小。若申请成功，则返回成功申请的内存基地址，否则返回 NULL。该函数完成如下动作：

- 1、首先，做一个堆归属的合法性检查，即判断用户提供的堆对象，是不是属于当前线程。若属于当前线程，则继续下一步的动作，否则直接返回 NULL；
- 2、检查空闲链表，以寻找一块大小大于用户请求的空闲内存块；
- 3、如果能够找到这样的内存块，则根据内存块的大小，确定是否需要进一步拆分，还是直接返回用户；
- 4、如果需要拆分，则把找到的内存块拆分成两块，然后把拆分后的两块内存块中的后一块，重新插入空闲链表，把第一块返回用户。若不需要拆分，则把找到的空闲块，直接从空闲链表中删除，然后返回用户；
- 5、如果从空闲链表中无法找到满足要求的空闲块，则调用 VirtualAlloc 函数，重新分配一个虚拟区域，并把该区域当成一块空闲块，进行上述处理；

6、返回用户分配的内存块地址，或者在失败的情况下，返回 NULL。

代码如下：

```
static LPVOID HeapAlloc(__HEAP_OBJECT* lpHeapObject,DWORD dwSize)
{
    __VIRTUAL_AREA_NODE*          lpVirtualArea    = NULL;
    __FREE_BLOCK_HEADER*           lpFreeBlock      = NULL;
    __FREE_BLOCK_HEADER*           lpTmpHeader      = NULL;
    LPVOID                         lpResult          = NULL;
    DWORD                         dwFlags            = 0L;
    DWORD                         dwFindSize        = 0L;

    if((NULL == lpHeapObject) || (0 == dwSize)) //Parameter check.
        return lpResult;

    //__ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(lpHeapObject->lpKernelThread != CURRENT_KERNEL_THREAD) //Check the
heap's owner.
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return lpResult;
    }
    //__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

    if(dwSize < MIN_BLOCK_SIZE)
        dwSize = MIN_BLOCK_SIZE;
    dwFindSize = dwSize + MIN_BLOCK_SIZE + sizeof(__FREE_BLOCK_HEADER);
```

上述代码完成了堆归属检查，并重新计算了用户所申请的内存块的大小。在当前的实现中，对于堆内存的申请，最小尺寸是 MIN\_BLOCK\_SIZE（定义为 16 字节），若用户请求的内存小于该数值，则调整为该数值。其中，dwFindSize 是待查找的目标空闲块的大小。只所以在 dwSize 的基础上，增加 MIN\_BLOCK\_SIZE 和控制头的尺寸，是为了确保任何空闲块的可用尺寸，都应该大于 MIN\_BLOCK\_SIZE。在找到一块空闲块之后，若

该空闲块的大小大于 `dwFindSize`，则需要对该空闲块进行分割，否则无需分割，直接返回给用户。

```
//
//Now,check the free list,try to find a free block.
//
lpFreeBlock = lpHeapObject->FreeBlockHeader.lpNext;
while(lpFreeBlock != &lpHeapObject->FreeBlockHeader)
{
    if(lpFreeBlock->dwBlockSize >= dwSize) //Find one.
    {
        if(lpFreeBlock->dwBlockSize >= dwFindSize) //Should split it into two free
blocks.
        {
            lpTmpHeader = (__FREE_BLOCK_HEADER*)((DWORD)lpFreeBlock
+ dwSize
            + sizeof(__FREE_BLOCK_HEADER)); //Pointing to second
part.

            lpTmpHeader->dwFlags = BLOCK_FLAGS_FREE;
            lpTmpHeader->dwBlockSize = lpFreeBlock->dwBlockSize - dwSize
            - sizeof(__FREE_BLOCK_HEADER); //Calculate second
part's size.

            //
            //Now,should replace the lpFreeBlock with lpTmpHeader.
            //
            lpTmpHeader->lpNext = lpFreeBlock->lpNext;
            lpTmpHeader->lpPrev = lpFreeBlock->lpPrev;
            lpTmpHeader->lpNext->lpPrev = lpTmpHeader;
            lpTmpHeader->lpPrev->lpNext = lpTmpHeader;

            lpFreeBlock->dwBlockSize = dwSize;
            lpFreeBlock->dwFlags |= BLOCK_FLAGS_USED;
            lpFreeBlock->dwFlags &= ~BLOCK_FLAGS_FREE; //Clear the
free flags.
```

```

        lpFreeBlock->lpPrev        = NULL;
        lpFreeBlock->lpNext        = NULL;
        lpResult                    = (LPVOID)((DWORD)lpFreeBlock
            + sizeof(__FREE_BLOCK_HEADER));
        goto __TERMINAL;
    }
    else //Now need to split,return the block is OK.
    {
        //
        //Delete the free block from free block list.
        //
        lpFreeBlock->lpNext->lpPrev = lpFreeBlock->lpPrev;
        lpFreeBlock->lpPrev->lpNext = lpFreeBlock->lpNext;
        lpFreeBlock->dwFlags        |= BLOCK_FLAGS_USED;
        lpFreeBlock->dwFlags        &= ~BLOCK_FLAGS_FREE; //Clear
free bit.

        lpFreeBlock->lpNext        = NULL;
        lpFreeBlock->lpPrev        = NULL;
        lpResult                    = (LPVOID)((DWORD)lpFreeBlock
sizeof(__FREE_BLOCK_HEADER));
        goto __TERMINAL;
    }
}

lpFreeBlock = lpFreeBlock->lpNext; //Check the next block.
}

if(lpResult) //Have found a block.
    goto __TERMINAL;

```

上述代码完成空闲块链表的搜索，一旦找到一块满足要求尺寸（dwSize）的空闲块，则进一步判断该空闲块的大小，是否大于 dwFindSize。若大于，则可以进行进一步拆分，否则直接返回用户找到的内存块。在拆分的情况下，把拆分后的第二块内存，重新插入空闲链表。

这时候，就可以很清楚的解释为什么只有在大于 dwFindSize 的时候，才需要拆分了。因为在拆分后，实际上还需要在空闲块的开头，预留 16 字节（空闲控制头的大小），作

为空闲块的控制头，这样若找到的空闲块小于 `dwFindSize`，则无法保证拆分后空闲块的大小，会大于 `MIN_BLOCK_SIZE`（这是空闲块的最小尺寸）。

如果从空闲链表中无法找到满足的空闲块，则需要扩充对的内存池了，这时候，需要调用 `VirtualAlloc` 函数，从系统空间中重新申请一个虚拟区域，然后把该虚拟区域当作一个空闲块对待，插入堆的空闲链表，这时候，还需要把虚拟区域插入对的虚拟区域链表。代码如下：

```
//
//If can not find a statisfying block in above process,we should allocate a
//new virtual area according to dwSize,insert the virtual area into free list,
//then repeat above process.
//
lpVirtualArea =
(__VIRTUAL_AREA_NODE*)GET_KERNEL_MEMORY(sizeof(__VIRTUAL_AREA_NO
DE));
if(NULL == lpVirtualArea) //Can not allocate kernel memory.
    goto __TERMINAL;
lpVirtualArea->dwAreaSize = ((dwSize + sizeof(__FREE_BLOCK_HEADER))
    > DEFAULT_VIRTUAL_AREA_SIZE) ?
    (dwSize + sizeof(__FREE_BLOCK_HEADER)) :
    DEFAULT_VIRTUAL_AREA_SIZE;

lpVirtualArea->lpStartAddress = GET_VIRTUAL_AREA(lpVirtualArea->dwAreaSize);
if(NULL == lpVirtualArea->lpStartAddress) //Can not get virtual area.
{
    RELEASE_KERNEL_MEMORY((LPVOID)lpVirtualArea);
    goto __TERMINAL;
}
//
//Insert the virtual area node object into virtual area list
//of current heap object.
//
lpVirtualArea->lpNext = lpHeapObject->lpVirtualArea;
lpHeapObject->lpVirtualArea = lpVirtualArea;
```

```
//
//Now,insert the free block(virtual area) into free list of the
//heap object.
//
lpTmpHeader = (__FREE_BLOCK_HEADER*)lpVirtualArea->lpStartAddress;
lpTmpHeader->dwFlags |= BLOCK_FLAGS_FREE;
lpTmpHeader->dwFlags &= ~BLOCK_FLAGS_USED; //Clear the used flags.
lpTmpHeader->dwBlockSize          =          lpVirtualArea->dwAreaSize          -
sizeof(__FREE_BLOCK_HEADER);

lpTmpHeader->lpNext  = lpHeapObject->FreeBlockHeader.lpNext;
lpTmpHeader->lpPrev  = &lpHeapObject->FreeBlockHeader;
lpTmpHeader->lpNext->lpPrev = lpTmpHeader;
lpTmpHeader->lpPrev->lpNext = lpTmpHeader;
```

若调用 `VirtualAlloc` 也失败，则 `HeapAlloc` 只能返回 `NULL` 了，否则，在成功调用 `VirtualAlloc` 的情况下，堆的空闲链表中会增加一块满足要求的内存空闲块（新申请的虚拟区域），这时候，重新调用 `HeapAlloc`，肯定是成功的，因此，`HeapAlloc` 递归调用自己，然后把返回的结果，返回给用户。

```
//
//Now,call HeapAlloc again,this time must successful.
//
lpResult = HeapAlloc(lpHeapObject,dwSize);

__TERMINAL:
return lpResult;
}
```

## 6.4.4 堆内存释放

`HeapFree` 函数完成堆内存的释放，该函数接受两个参数：待释放内存的起始地址，

以及所属的堆对象。内存释放函数完成下列操作：

- 1、首先，把待释放的内存（实际上是一个空闲块），修改标志并插入空闲链表；
- 2、找到该空闲块所属于的虚拟区域；
- 3、对该虚拟区域，发起一个合并空闲内存块的操作；
- 4、完成块的合并操作后，检查当前虚拟内存区域是否是一个完整的内存块，即不包含尚未释放的内存，若是，则调用 `VirtualFree` 函数，把该虚拟区域返回系统，否则返回。

之所以增加第四步操作，是为了实现一种“按需分配”的目的，一个策略就是，尽量返回不用的资源给操作系统，这样不会造成资源的浪费。如果不进行上述第四步操作，则可能当前线程申请了大量的虚拟区域而闲置不用，其它线程申请内存失败的情况。

`HeapFree` 函数代码如下：

```
static VOID HeapFree(LPVOID lpStartAddr, __HEAP_OBJECT* lpHeapObj)
{
    __FREE_BLOCK_HEADER*    lpFreeHeader  = NULL;
    __VIRTUAL_AREA_NODE*    lpVirtualArea = NULL;
    __VIRTUAL_AREA_NODE*    lpVirtualTmp  = NULL;

    if((NULL == lpStartAddr) || (NULL == lpHeapObj)) //Invalid parameter.
        return;

    lpFreeHeader = (__FREE_BLOCK_HEADER*)((DWORD)lpStartAddr
        - sizeof(__FREE_BLOCK_HEADER)); //Get the block's header.
    if(!(lpFreeHeader->dwFlags & BLOCK_FLAGS_USED)) //Abnormal case.
        return;
```

上述代码根据待释放内存的起始地址，找到该内存块对应的控制头，然后检查控制头的标记，若标记不是空闲，则属于一种异常情况，否则继续执行。

```
//
//Now,check the block to be freed belong to which virtual area.
//
lpVirtualArea = lpHeapObj->lpVirtualArea;
while(lpVirtualArea)
{
```

```

if(((DWORD)lpStartAddr > (DWORD)lpVirtualArea->lpStartAddress) &&
   ((DWORD)lpStartAddr < (DWORD)lpVirtualArea->lpStartAddress
    + lpVirtualArea->dwAreaSize)) //Belong to this virtual area.
{
    break;
}
lpVirtualArea = lpVirtualArea->lpNext;
}
if(NULL == lpVirtualArea) //Can not find a virtual area that countains
                           //the free block to be released.
{
    //printf("\r\nFree a invalid block: can not find virtual area.");
    return;
}

```

上述代码检查待释放内存属于哪个虚拟区域，找到对应的虚拟区域的目的，是为了完成一个合并操作。若待释放内存无法跟一个虚拟区域对应，则可能是一个不正常的操作，因此直接返回。

```

//
//Now,should insert the free block into heap object's free list.
//
lpFreeHeader->dwFlags |= BLOCK_FLAGS_FREE;
lpFreeHeader->dwFlags &= ~BLOCK_FLAGS_USED; //Clear the used flags.
lpFreeHeader->lpPrev    = &lpHeapObj->FreeBlockHeader;
lpFreeHeader->lpNext    = lpHeapObj->FreeBlockHeader.lpNext;
lpFreeHeader->lpPrev->lpNext = lpFreeHeader;
lpFreeHeader->lpNext->lpPrev = lpFreeHeader;

CombineBlock(lpVirtualArea,lpHeapObj); //Combine this virtual area.

```

上述代码把待释放的空闲块，插入了当前对的空闲链表，然后发起一个合并操作，合并的对象，就是待释放内存所属的虚拟区域。对于合并过程，后面会详细解释。

```

//
//Now,should check if the whole virtual area is a free block.
//If so,delete the free block from free list,then release the
//virtual area to system.
//
lpFreeHeader = (__FREE_BLOCK_HEADER*)(lpVirtualArea->lpStartAddress);
if((lpFreeHeader->dwFlags & BLOCK_FLAGS_FREE) &&
    (lpFreeHeader->dwBlockSize + sizeof(__FREE_BLOCK_HEADER)
    == lpVirtualArea->dwAreaSize))
{
    //
    //Delete the free block from free list.
    //
    lpFreeHeader->lpPrev->lpNext = lpFreeHeader->lpNext;
    lpFreeHeader->lpNext->lpPrev = lpFreeHeader->lpPrev;
    //
    //Now,should delete the virtual area node object from
    //heap object's virtual list.
    //
    lpVirtualTmp = lpHeapObj->lpVirtualArea;
    if(lpVirtualTmp == lpVirtualArea) //The first virtual node.
    {
        lpHeapObj->lpVirtualArea = lpVirtualArea->lpNext;
    }
    else //Not the first one.
    {
        while(lpVirtualTmp->lpNext != lpVirtualArea)
        {
            lpVirtualTmp = lpVirtualTmp->lpNext;
        }
        lpVirtualTmp->lpNext = lpVirtualArea->lpNext; //Delete it.
    }
    //
    //Then,should release the virtual area and virtual area node

```

```

    //object.
    //
    RELEASE_VIRTUAL_AREA((LPVOID)lpVirtualArea->lpStartAddress);
    RELEASE_KERNEL_MEMORY((LPVOID)lpVirtualArea);
}

return;
}

```

在完成了虚拟区域的合并之后，则检查当前虚拟区域是否本身是一个完整的空闲块。若是，则调用 `VirtualFree` 函数释放该虚拟区域，否则函数返回。对于虚拟区域本身是否是一个空闲块的判断方式十分简单，只需要判断虚拟区域的第一个内存块的长度，加上控制头，是否等于整个虚拟区域大小即可。若是，则整个虚拟区域是一块空闲块，否则就不是。

下面的 `CombineBlock` 函数，完成了特定虚拟区域的空闲内存块合并工作。该函数操作过程如下：

- 1、从第一个空闲块开始，判断是否有连续的两块内存块，其状态都是空闲。这里的连续，是内存块地址的连续（相邻），而不是空闲块在空闲链表中的连续；
- 2、如果发现这样的两块内存，则把第二块从空闲链表中删除，合并到第一块中；
- 3、继续执行上述操作，直到到达虚拟区域的末端。

代码如下：

```

static          VOID          CombineBlock(__VIRTUAL_AREA_NODE*
lpVirtualArea,__HEAP_OBJECT* lpHeapObj)
{
    __FREE_BLOCK_HEADER*      lpFirstBlock    = NULL;
    __FREE_BLOCK_HEADER*      lpSecondBlock   = NULL;
    LPVOID                    lpEndAddr       = NULL;

    if((NULL == lpVirtualArea) || (NULL == lpHeapObj)) //Invalid parameters.
        return;

    lpEndAddr    = (LPVOID)((DWORD)lpVirtualArea->lpStartAddress

```

```
+ lpVirtualArea->dwAreaSize);
```

其中，lpEndAddr 是一个结束标志，一旦待合并内存块的控制头地址跟该地址相同，则说明合并已经结束。

```
lpFirstBlock  = (__FREE_BLOCK_HEADER*)lpVirtualArea->lpStartAddress;
lpSecondBlock = (__FREE_BLOCK_HEADER*)((DWORD)lpFirstBlock
    + sizeof(__FREE_BLOCK_HEADER)
    + lpFirstBlock->dwBlockSize); //Now,lpSecondBlock pointing to the second
block.
```

初始化 lpFirstBlock 和 lpSecondBlock 变量，使得 lpFirstBlock 指向当前虚拟区域中的第一个内存块，lpSecondBlock 指向第二块内存块，然后进入下面的循环。

```
while(TRUE)
{
    if(lpEndAddr == (LPVOID)lpSecondBlock) //Reach the end of the virtual area.
        break;
    if((lpFirstBlock->dwFlags & BLOCK_FLAGS_FREE) &&
        (lpSecondBlock->dwFlags & BLOCK_FLAGS_FREE)) //Two blocks all
free,combine it.
    {
        lpFirstBlock->dwBlockSize += lpSecondBlock->dwBlockSize;
        lpFirstBlock->dwBlockSize += sizeof(__FREE_BLOCK_HEADER);
        //
        //Delete the second block from free list.
        //
        lpSecondBlock->lpNext->lpPrev = lpSecondBlock->lpPrev;
        lpSecondBlock->lpPrev->lpNext = lpSecondBlock->lpNext;

        lpSecondBlock = (__FREE_BLOCK_HEADER*)((DWORD)lpFirstBlock
            + sizeof(__FREE_BLOCK_HEADER)
            + lpFirstBlock->dwBlockSize); //Update the second block.
        continue; //Continue to next round.
```

```

    }
    if((lpFirstBlock->dwFlags & BLOCK_FLAGS_USED) ||
        (lpSecondBlock->dwFlags & BLOCK_FLAGS_USED)) //Any block is used.
    {
        lpFirstBlock  = lpSecondBlock;
        lpSecondBlock = (__FREE_BLOCK_HEADER*)((DWORD)lpFirstBlock
            + sizeof(__FREE_BLOCK_HEADER)
            + lpFirstBlock->dwBlockSize);
        continue;
    }
}
}

```

上述循环比较简单，只是判断 `lpFirstBlock` 的标志字段，是否是空闲（FREE），若是，则进一步判断 `lpSecondBlock` 是否也空闲，如果是，则具备合并条件，合并 `lpFirstBlock` 和 `lpSecondBlock`，然后更新 `lpSecondBlock`，重新进入循环，否则（`lpFirstBlock` 和 `lpSecondBlock` 中至少一个是非空闲块），则更新 `lpFirstBlock` 为 `lpSecondBlock`，更新 `lpSecondBlock` 为 `lpSecondBlock` 的相邻块，继续下一轮迭代。

## 6.4.5 malloc 和 free 的实现

`malloc` 和 `free` 是标准 C 运行期库函数，实现这两个函数，对于代码的移植十分有意义。很明显，采用 `HeapAlloc` 和 `HeapFree` 函数可以很容易的模拟这两个函数，但 `HeapAlloc` 和 `HeapFree`，都接受一个堆对象指针作为参数，而 `malloc` 和 `free` 则没有。因此，为了屏蔽这个差异，我们在核心线程对象（`__KERNEL_THREAD_OBJECT`）中引入一个缺省堆的变量，所有 `malloc` 和 `free` 函数的内存申请，都从缺省堆中进行。

下面是 `malloc` 的实现代码：

```

LPVOID malloc(DWORD dwSize)
{
    DWORD          dwFlags    = NULL;
    LPVOID          lpResult   = NULL;

```

```

__HEAP_OBJECT*   lpHeapObj  = NULL;
DWORD            dwHeapSize = 0L;

if(0 == dwSize) //Invalid parameter.
    return lpResult;

if(NULL == CURRENT_KERNEL_THREAD->lpDefaultHeap) //Should create one.
{
    dwHeapSize = (dwSize + sizeof(__FREE_BLOCK_HEADER) >
DEFAULT_VIRTUAL_AREA_SIZE) ?
(dwSize + sizeof(__FREE_BLOCK_HEADER)) :
DEFAULT_VIRTUAL_AREA_SIZE;
    lpHeapObj = CreateHeap(dwHeapSize);
    if(NULL == lpHeapObj) //Can not create heap object.
        return lpResult;
    //__ENTER_CRITICAL_SECTION(NULL,dwFlags);
    CURRENT_KERNEL_THREAD->lpDefaultHeap = (LPVOID)lpHeapObj;
    //__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
}
else
{
    lpHeapObj =
(__HEAP_OBJECT*)CURRENT_KERNEL_THREAD->lpDefaultHeap;
}
return HeapAlloc(lpHeapObj,dwSize);
}

```

首先，该函授判断当前线程的缺省堆对象（lpDefaultHeap）是否存在，若存在（不为 NULL），则直接以缺省堆为目标堆，调用 **HeapAlloc** 函数，把该函数的结果返回给用户，否则，该函数首先调用 **CreateHeap**，创建一个缺省堆，并初始化当前线程的 lpDefaultHeap 变量，在此基础上，再调用 **HeapAlloc** 函数。

很明显，只有 **malloc** 函数第一次调用的时候，可能会发生 lpDefaultHeap 为 NULL 的情况，后续调用的时候，不会出现这种情况。

对于 free 函数，实现起来就更简单了：

```
VOID free(LPVOID lpMemory)
{
    __HEAP_OBJECT*      lpHeapObj = NULL;

    if(NULL == lpMemory)
        return;

    lpHeapObj = (__HEAP_OBJECT*)CURRENT_KERNEL_THREAD->lpDefaultHeap;
    if(NULL == lpHeapObj)    //Invalid case.
        return;
    HeapFree(lpMemory,lpHeapObj);
}
```

首先，该函数确保当前线程的缺省堆是存在的，然后调用 **HeapFree** 来释放具体的内存。否则可能是一种不正常操作，直接导致 free 返回。

## 第七章 互斥和同步机制的实现

### —— The implementation of synchronization

## 7.1 互斥和同步概述

在操作系统的设计中，尤其是引入多任务的情况下，需要考虑下列可能的几种“竞争”状态：

- 1、中断处理程序和应用程序之间的竞争，比如，一个共享的数据对象，既可以被应用程序修改，也可以被中断处理程序修改，这时候就需要充分考虑应用程序和中断处理程序之间的竞争关系，因为中断随时可能发生；
- 2、应用程序与应用程序之间的竞争，比如，两个应用线程共享同一个数据结构，且可以任意修改，这样就需要充分考虑如何避免修改过程中出现的冲突（一个线程修改了一半，被替换出去，另一个线程继续修改同样的数据结构）；
- 3、在多 CPU 的情况下，除了考虑上述竞争关系之外，还需要考虑多个 CPU 之间的冲突关系。

在 Hello China 当前版本的实现中，对这几种可能的冲突情况，都做了充分考虑，并实现了相应机制，来处理这种竞争关系。虽然当前版本的 Hello China 只支持单 CPU，但在设计的时候，对多 CPU 的情况也做了充分考虑，并在数据结构和代码的设计中，留下了余地，以便于后续向多 CPU 平滑过渡。

在当前版本的实现中，关键区段用于完成中断处理程序和应用程序之间的同步，而应用程序之间的互斥与同步，则实现了事件对象（Event Object）、互斥体对象（Mutex Object）和信号量对象（Semaphore Object），这些对象可支持标准的同步或互斥语义，也支持超时等待机制，限于篇幅，在本章中，我们仅仅对信号量对象的实现做详细描述，其它对象的实现，与信号量对象的实现类似，读者可自行查阅代码。

## 7.2 关键区段概述

所谓 Critical Section（关键区域），是一段可执行的代码，在这段代码的执行过程中，不能被打断，否则可能会产生严重问题。一般情况下，关键区域内的代码，往往对全局的数据进行修改或读取，这些数据为系统中所有的线程共享，这样如果在修改的过程中被打断，可能会导致这些全局的数据处于一种不一致的状态。

在当前版本的 Hello China 的设计中，为了确保操作系统核心数据结构的一致性，实现了关键区段。如果在一段代码中，涉及到修改全局变量，比如内存管理数据结构等，就需要把这段代码作为关键区段来对待。下列两个宏：

```
__ENTER_CRITICAL_SECTION(objptr,flags)
__LEAVE_CRITICAL_SECTION(objptr,flags)
```

用来完成关键区段的保护，在一段代码的开始部分，调用 `__ENTER_CRITICAL_SECTION` 宏，就进入了关键区段，执行过程中，不会收到任何干扰，一旦代码执行完毕，调用 `__LEAVE_CRITICAL_SECTION` 宏，离开关键区段。其中，`objptr` 是一个对象指针，目前没有使用（代码中，可以把该变量设置为 `NULL`，该变量的作用，在本文后续部分介绍），`flags` 变量是一个本地声明的变量，用来保存 `EFLAGS` 寄存器的值。比如，`nKernelObject` 是一个全局变量，在修改的时候，需要使用关键区段进行保护，可以采用如下代码：

```
DWORD dwFlags;
__ENTER_CRITICAL_SECTION(NULL,dwFlags)
nKernelObject += 100;
__LEAVE_CRITICAL_SECTION(NULL,dwFlags)
```

在接下来的几节中，我们详细介绍关键区段产生的原因，以及当前版本 **Hello China** 的关键区段的实现方式。在本章的后面部分，我们对 **Power PC** 环境下的互斥机制进行了简单的描述，以加深读者对该部分的理解。

## 7.3 关键区段产生的原因

关键区段（**Critical Section**）的最根本原因，是确保系统数据结构的一致性。而数据的不一致性，一般是由于竞争修改引起的。所谓竞争修改，指的是多个实体（比如，线程）试图同时修改该数据，下列原因，是操作系统中常见的竞争修改：

### 7.3.1 多个线程之间的竞争

比如，假设在一个多线程单 **CPU** 的环境中，有一个记录所有核心对象数量的变量 `nKernelObject`，这个变量可以被所有的线程共享，假设一个线程 **A**，创建了一个核心对象，并假设 `nKernelObject` 当前的值为 100，这时候，该线程需要增加这个变量的值，以便反映这种情况（核心对象数量增加），于是可能会通过下列代码进行修改：

```
nKernelObject += 1;
```

如果被编译成汇编语言，可能是下面这个样子：

```
mov eax,nKernelObject
inc eax
mov nKernelObject,eax
```

假设在上述第二条指令（`inc eax`）完成后，该线程的时间片用完，被临时阻塞，这时候，`nKernelObject` 的值由于还没有被修改，因此仍然是 100，但 `eax` 寄存器的值已经是 101 了。线程 A 被阻塞后，另外一个线程 B 被唤醒，继续运行，而线程 B 同样创建了一个核心对象，也需要对 `nKernelObject` 进行递增，这时候，在线程 B 中，会执行同样的代码。假设线程 B 在执行完上述最后一条指令（`mov nKernelObject,eax`）后，时间片用完，被阻塞，这时候，由于 B 修改了 `nKernelObject`，使得 `nKernelObject` 变成了 101，而不是原来的 100 了。

线程 A 又被唤醒，继续运行，由于线程 A 被挂起的时候，是刚刚执行完 `inc eax` 指令，于是线程 A 会继续执行 `mov nKernelObject,eax` 指令，这样原先存储在 `eax` 内的值（101），就覆盖了当前 `nKernelObject` 的值，于是当前 `nKernelObject` 的值，仍然是 101，于是不一致就产生了（正确的情况下，`nKernelObject` 应该是 102）。

可以看出，产生上述问题的原因，就是线程 A 在修改 `nKernelObject` 变量的时候，被打断了。因此，为了解决这个问题，必须确保线程 A 在修改 `nKernelObject` 变量的时候，作为一个原子操作进行，不能被打断，即把上述修改 `nKernelObject` 的区域，作为一个关键区域来进行对待。

### 7.3.2 中断服务程序与线程之间的竞争

在单 CPU 环境下，另外一种可能产生不一致的状态，是线程跟中断处理程序之间的竞争。比如，假设线程 A 执行完 `inc eax` 指令后（这时候，`nKernelObject` 是 100，`eax` 的值是 101），一个中断发生，这时候，系统将把线程 A 临时挂起，把控制转移到中断处理程序，在中断处理程序中，也创建了一个核心对象，同样，`nKernelObject` 被增加了一（这时候，`nKernelObject` 的值是 101），当中断处理程序返回后，线程 A 继续执行，于是，`mov nKernelObject,eax` 指令被执行，这样由于线程 A 被打断的时候，`eax` 的值是 101，因此，这时候 `nKernelObject` 被 `eax` 覆盖，仍然是 101（正确的情况下，应该是 102）。

7.3.3 多个 CPU 之间的竞争

在多 CPU 的 SMP（对称多处理）环境下，这种情形还要复杂，因为不但是多个线程之间的竞争，还存在多个 CPU 之间的竞争。仍然以上述假设，nKernelObject 记录了系统中核心对象的数量，A 和 B 是系统中的两个线程，分别创建了一个系统核心对象，这时候，线程 A 需要对 nKernelObject 进行加一操作，与单处理系统不同的是，在单处理系统中，只有在线程 A 时间片用完的情况下，才会出现上述不一致的情况，因为如果线程 A 的运行时间片没有用完，那么线程 A 不会被挂起，上述不一致就不会产生了（不考虑线程 A 被中断打断的情况），但在多处理器系统中，即使线程 A 时间片没有用完，仍然在运行，也可能产生不一致的现象，因为线程 B 可能与线程 A 同时在运行（两个线程分别在两个不同的 CPU 上运行），这个时候，如果出现下列运行序列，就会产生不一致现象：

时 刻	线程 A 执行的指令	线程 B 执行的指令	nKernelObject 的值
N	... ..	mov eax,nKernelObject	100
N + 1	mov eax,nKernelObject	inc eax	100
N + 2	inc eax	mov nKernelObject,eax	101
N + 3	mov nKernelObject,eax	... ..	101
N + 4	... ..		

表 7-1 一个产生访问冲突的指令序列

可以看出，在多 CPU 的环境下，即使线程不被打断，也可能产生不一致状态，当然，在多 CPU 环境下，线程因为被打断而可能产生的不一致状态，仍然存在。

## 单 CPU 下关键区段的实现

在单 CPU 环境下，通过上述分析可以看出，有两个原因可能导致数据不一致：

- 1、线程切换，即一个线程正在试图修改全局数据的时候，切换到另外一个试图修改同一数据的线程；
- 2、中断发生，即一个试图修改全局变量的线程，正在试图修改全局数据的时候，发生中断，而该中断的服务程序，也试图修改同一变量。

因此，在单 CPU 环境下，只要在修改关键数据结构的代码段执行过程中，避免上述两类事件发生，就可以实现关键区段。其中，在修改关键数据的时候，一般不会涉及到系统调用，在这种情况下，线程切换也是由于中断导致（系统时钟中断），因此，在单 CPU 环境下，要实现关键区段，只要临时禁止中断即可。一种可能的实现就是，在关键代码段的开始，禁止中断，在关键代码段执行完毕后，重新开启中断，如下：

```
__asm{
    cli
}
nKernelObject += 100;
__asm{
    sti
}
```

这样做的一个弊端就是，在代码的最后，又硬性的重新打开了中断（sti 指令），考虑这样一种情况：

```
VOID IncreaseGlobal()
{
    __asm{
        cli
    }
    nKernelObject ++;
    __asm{
        sti
    }
}
```

```

    }

    VOID ModifyGlobal()
    {
        __asm{
            cli
        }
        IncreaseGlobal();
        AnotherRoutine();
        __asm{
            sti
        }
    }

```

在 `ModifyGlobal` 函数中, 认为 `IncreaseGlobal` 和 `AnotherRoutine` 都是关键区段中的代码, 因此在调用这两个函数的时候, 首先禁止了中断。但在 `IncreaseGlobal` 函数中, 在实际执行关键代码 (`nKernelObject ++`) 的时候, 也首先禁止了中断, 完成修改后, 又恢复了中断 (`sti`), 这样问题就产生了: 从 `IncreaseGlobal` 函数返回后, 中断实际上已经打开, 这样 `AnotherRoutine` 实际上已经不在关键区段里面了!

产生上述问题的原因, 就是在每个函数中, 都硬性的关闭或打开了中断, 而没有考虑调用自己的更上级的函数的实际情况。由此可见, 通过直接关闭或打开中断的方式来实现关键区段, 是不完整的, 一个合理的实现是, 在关键区段的开始处, 首先保存 `EFLAGS` 寄存器的值, 然后再关闭中断, 在关键代码段的结束, 恢复先前保存的 `EFLAGS` 的值, 这样就可以确保不影响原始 `EFLAGS` 寄存器的值, 因为开启中断或者关闭中断, 不过是对 `EFLAGS` 寄存器的一个标志位的清除或设置而已。如下:

```

VOID IncreaseGlobal()
{
    __asm{
        pushfd    //Save EFLAGS register.
        cli
    }
    nKernelObject ++;
    __asm{

```

```

        popfd    //Restore EFLAGS register.
    }
}

```

这样从 `IncreaseGlobal` 函数返回后，原来设置的中断标志，得到了保存，从而使得调用函数的关键区段，得以连续。

在 `Hello China` 的实现中，就是遵循了这个原则，不过是把所有上述汇编语言完成的功能，使用一个宏定义来代替，以增强代码的可移植性：

```

#define __ENTER_CRITICAL_SECTION(objptr,flags) \
    __asm{
        push eax                \
        pushfd                  \
        pop eax                 \
        mov flags,eax           \
        pop eax                 \
        cli                     \
    }

#define __LEAVE_CRITICAL_SECTION(objptr,flags) \
    __asm{
        push flags              \
        popfd                   \
    }

```

为了不破坏代码的堆栈框架，我们使用一个局部变量 `flags`，来保存 `EFLAGS` 寄存器的值（如果直接使用 `PUSHFD` 指令，可能会破坏调用上述两个宏的函数的堆栈框架）。因此，在调用上述两个宏的时候，必须首先声明一个局部变量，如下：

```

DWORD dwFlags;
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
nKernelObject += 100;
... .. //Other critical code here.
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

```

`__ENTER_CRITICAL_SECTION` 和 `__LEAVE_CRITICAL_SECTION` 宏还有另外一个参数 `objptr`，是一个对象指针（`__COMMON_OBJECT` 指针），用来完成在多 CPU 下关键区段的实现，在目前版本的 Hello China 中，由于尚不支持多 CPU，因此该参数可以设置为 `NULL`。

## 多 CPU 下关键区段的实现

### 多 CPU 环境下的实现方式

在多 CPU（比如，对称多处理 **SMP**）的情况下，情况要稍微复杂一些，因为不但要考虑在单 CPU 下可能发生的问题（线程竞争、中断竞争），还要考虑多个 CPU 的并发竞争问题。因此，仅仅采用关闭中断的方式，来实现关键区段，已经不能满足要求。

这种情况下，一种可行的解决方案就是，为每个需要保护的全局数据结构，设置一个保护标志，每当对这些全局数据进行修改的时候，首先检测该保护标志，如果该保护标志没有被设置（说明没有其它线程正在修改数据），于是首先设置该保护标志，并进入修改数据的关键代码段，一旦完成修改（即在关键代码段的最后），再恢复该保护标志。如果试图修改全局数据的线程，检测到保护标志被设置，则进入等待状态（忙等待），直到检测到该标志被释放（清除）。上述过程，可以采用伪代码进行如下描述：

```
disable interrupt;           //Disable interrupt first.
while(protecting flags == 1); //Waiting.
Protecting flags = 1;        //Set this flags, get lock.
Modify global data structures;
Protecting flags = 0;        //Clear flags, release lock.
Enable interrupt.
```

假设有两个线程 A 和 B，都试图修改同一个全局变量，A 在修改前，先禁止中断（这样就可以确保在 A 运行的 CPU 上，不会发生线程切换和中断），然后检测全局变量的保护标志，由于这时候没有其它线程正在修改该全局变量，于是保护标志处于清除（空闲）状态，于是 A 设置保护标志为 1（占有状态），并开始修改全局数据。

这时候，线程 B 也试图修改全局数据，执行跟 A 相同的过程：首先禁止本地中断（B

所在的 CPU 的中断)，然后检测保护标志，但这时候 A 正在修改全局数据，已经设置了保护标志为 1（占有状态），所以会导致线程 B 将一直处于检测状态（忙等待）。

当 A 完成对全局数据的修改后，设置保护标志为 0（释放保护标志），这时候，线程 B 会检测到保护标志被释放（线程 B 一直在检测该标志），于是线程 B 重新获得该标志，进入修改全局数据的过程。可以看出，采用这种方式，可以很好的完成多个线程之间的同步。

仔细的读者会发现，上述过程仍然是存在问题的，假设线程 A 在检测到保护标志空闲，但还没有完成设置的时候，线程 B 也刚好执行到这里，发现保护标志空闲（因为 A 还没有完成设置），于是 B 也认为没有其它线程正在修改全局数据，这样就产生不一致了，A 和 B 会同时进入修改全局数据的过程。

产生上述问题的原因，就是对保护标志的检测和设置，是分开执行的，而不是一个整体的操作。为解决上述问题，单靠软件是不行的，需要靠 CPU 的支持，即需要 CPU 提供一种能够在一个原子操作内完成上述工作（检测和设置）的指令，这种指令就是有名的 testing-and-setting 指令。在 INTEL CPU 中，可以采用 BTS 指令完成上述过程。

BTS 指令的格式为：

BTS bitstr, bitoffset

其中，bitstr 是一个比特串，而 bitoffset 则是一个偏移，指出了 bitstr 中的一个比特。该指令首先把 bitstr 中的第 bitoffset 个比特，保存在 EFLAGS 寄存器的 CF 位中，然后设置 bitstr 中的 bitoffset 个比特为 1。所有这些操作都是原子的，即在操作的过程中，不会发生中断（中断只是在指令的边界被引发）。在多 CPU 的环境下，可以在该指令的前面，加上一个总线锁定指示符（LOCK），让该指令执行的时候，锁定总线，这样在该指令执行的过程中，其它 CPU 也无法中断。在操作系统的实现中，采用该指令完成多个 CPU 之间的同步。

下面是用该指令完成保护的过程：

```
__TRY_AGAIN:
    LOCK BTS flags,0
    JC __TRY_AGAIN
    CRITICAL CODE
```

其中 flags 是保护全局数据的标记，BTS 指令把 flags 中的第一个比特（比特 0），读

入 CF 标记位，然后设置 flags 的第一个比特为 1。JC 指令判断 EFLAGS 寄存器的 CF 标志是否为 1，如果为 1，则跳转到 \_\_TRY\_AGAIN 标号处执行，这样如果原来 flags 的第一比特为 1（被占用），则上述指令会一直循环，如果一旦另外一个线程清除了 flags 的第一个比特（释放锁），则上述代码会检测到这个改变，然后进入 CRITICAL CODE 段执行。可以看出，上述过程就是一个不断检测，并加锁的过程。LOCK 指示符指示 CPU，在执行 BTS 指令的时候，锁住总线，这样就实现了多 CPU 下的原子操作。

对于保护标记的释放操作，十分简单，只需要使用适当的指令清除保护标记的适当 bit 即可，在此不做赘述。

## Hello China 的未来实现

需要说明的是，在 Hello China 当前版本的实现中，是针对单 CPU 环境的，没有实现支持多 CPU 的版本。但为实现多 CPU 的版本，预留了扩展接口。我们知道，目前情况下，所有内核对象都是从 \_\_COMMON\_OBJECT 继承的，这样就可以直接在 \_\_COMMON\_OBJECT 的定义中，增加一个保护标志，用于保护多 CPU 环境下，对内核对象的同步修改，这个保护标志就可以被所有的继承自 \_\_COMMON\_OBJECT 的内核对象继承。在 \_\_ENTER\_CRITICAL\_SECTION 宏定义中，预留了一个参数 (objptr，请参考本章前面相关内容)，该参数就是一个指向 \_\_COMMON\_OBJECT 的指针。比如，在多 CPU 环境下，为了完成对一个 \_\_KERNEL\_THREAD\_OBJECT 对象 KernelThread 的属性的修改，可以这样进行：

```
DWORD dwFlags;  
__ENTER_CRITICAL_SECTION((__COMMON_OBJECT*)&KernelThread,dwFlags);  
Critical code here.  
__LEAVE_CRITICAL_SECTION((__COMMON_OBJECT*)&KernelThread,dwFlags);
```

这样就实现了多 CPU 环境下的数据保护（线程同步）。在 \_\_ENTER\_CRITICAL\_SECTION 的定义中，就是采用 BTS 指令，实现了等待/加锁过程。

## Power PC 下关键区段的实现

在单 CPU 环境下，Power PC 关键区段也可以采用关闭中断的方式来实现，在 Power PC CPU 中，通过清楚 MSR（机器状态寄存器）中的 EE 比特（External Exception），可以禁止外部可屏蔽中断，以达到同步的目的。但对于多处理器环境下的同步，Power PC 提供了跟 IA32 不一样的机制来完成，本节我们简单介绍这种机制，并给出几个实例。

### Power PC 提供的互斥访问机制

Power PC 提供了几个用于互斥操作的指令，采用这些指令，可以完成多处理器环境下的同步操作。最典型的是下列两个指令：

1. **lwarx**（Load word and reserve index）指令，该指令的作用是，读取内存中的一个特定字（在 Power PC 中，字的长度是 32 比特的，而在 IA32 中，字的长度定义为 16 比特）到一个特定的寄存器，然后设置一个 Reserve 位，并启动内存窥探机制，对上述内存位置进行监控。一旦该内存位置被修改（可能是其它的处理器），则清除 Reserve 比特，否则一直保持 Reserve 比特。该指令的格式为：**lwarx rD,rA,rB**（其中，rA 和 rB 寄存器给出了 Effective 地址，该指令把 Effective 地址位置的字，读入 rD 寄存器，该 Effective 地址也是内存窥探机制作用的目标地址）；
2. **stwcx**（Store word conditional indexed）指令，该指令与上述 **lwarx** 指令对应，用来协同完成同步机制。该指令的作用是，判断 Reserve 比特是否为 1，若为 1，则设置指定地址位置的字为指定字（指定内存位置和指定字，由该指令的操作数寄存器指定，一般情况下，指定的内存位置与 **lwarx** 指令指定的有效地址相同），清除 Reserve 标志，并设置一个成功标志（CR0 寄存器的特定比特），若 Reserve 标志为 0，则该指令不作任何其它操作，仅仅设置一个失败标志（CR0 寄存器的特定比特）。该指令的格式与 **lwarx** 类似：**stwcx rS,rA,rB**，其中，rA 和 rB 两个通用寄存器指定 Effective 地址，rS 寄存器指定要存储的值。

当然，上述简单的描述，仅仅是这两个指令的基本用途，这两个指令还有一些其它的用途，在此不作赘述。

许多较复杂的原子操作和同步操作，都可以采用上述两条指令实现。比如，下列代码实现了一个简单的 Test-and-set 例程：

```
loop: lwarx r5,0,r3
      cmpwi r5,0
      bne $+12
      stwcx r4,0,r3
      bne- loop
```

```
continue:
    ... ..
```

上述代码中，假设 r3 寄存器存放了要测试的内存位置（Effective 地址），lwarx 执行后，把该位置的一个字，读入了 r5 寄存器，并设置了 Reserve 比特。第二条指令（cmpwi），用于比较 r5 寄存器的值是否为 0（即原内存位置的字是否为 0），若不为零，则跳转到 continue 标号处执行，若为 0，则继续执行。这时候，stwcx 指令根据 Reserve 比特的值，来确定是否把 r4 寄存器的值（非 0）存放到上述内存位置。若成功（Reserve 比特为 1，成功的把 r4 寄存器的值，存放到 r3 指定的有效地址处），则跳出循环，继续执行，否则跳转到 loop 标号处，重新执行上述操作。

上述操作是一个原子操作，考虑在多 CPU 的环境下，一种访问冲突的情况：假设有两个 CPU（CPU1 和 CPU2）试图同时对同一个位置进行 Test and set 操作，则下列情况可能发生：

CPU1 指令序列	CPU2 指令序列
-----	Lwarx r5,0,r3
Lwarx r5,0,r3	Cmpwi r5,0
Cmpwi r5,0	Bne \$+12
Bne \$+12	Stwcx r4,0,r3
Stwcx r4,0,r3	-----
Bne- loop	

表 7-2      一个产生访问冲突的指令序列（双 CPU）

在上述序列下，CPU1 和 CPU2 会把 r3 指定的内存位置处的数字，读入 r5 寄存器，由于这时该位置还没有被修改，因此 cmpwi 指令的执行结果，都会比较成功，从而导致 stwcx 指令的执行。这时候，CPU2 可以成功执行该指令，因为在 CPU2 执行前，该位置尚未被修改，但 CPU1 却执行失败，因为在 CPU1 执行 stwcx 执行前，CPU2 已经修改了该位置（CPU1 通过内存窥探机制得知），从而导致 Reserve 标志被清除。这样目标为同一位置的 test and set 操作，只有一个 CPU set 成功，其它 CPU 都没有设置成功，从而实现了共享资源的同步和保护。

## 多 CPU 环境下的互斥机制

在多 CPU 环境下，对共享资源的访问需要一种互斥锁（spin lock）来进行同步，尤其是在中断上下文环境中，对共享资源进行访问的情况下。Power PC CPU 通过 `lwarx` 和 `stwcx` 指令，可以实现一个 `test and set` 的原子操作，利用这个原子操作，可以构造一个自旋锁来保护共享数据。应用程序在访问共享数据的时候，首先试图获取保护该数据结构的自旋锁，若获取，则访问共享的数据结构，否则进入忙等待状态。

下面给出一个典型的自旋锁的实现，首先看该自旋锁的获取操作：

`Acquire_lock:`

`loop:`

```
    li r4,1
    bl test_and_set
    bne- loop
    isync
    blr
```

第一条指令，`li r4,1`，用于初始化 `r4` 寄存器，然后调用 `test-and-set` 例程（该例程的实现，参考上节）。在 `test and set` 例程里，会对自旋锁（一个内存字）进行判断，若自旋锁当前状态为可获取状态（为 0），则 `test and set` 设置自旋锁为 1（`r4` 寄存器），并设置 `CR0` 寄存器中的特定比特（该比特用于标志比较结果），若自旋锁处于被占用状态，则 `test and set` 例程直接返回。在上述代码中，若 `test-and-set` 例程没有获得自旋锁，则 `bne` 指令会被执行，从而导致该例程有一次被调用，直到自旋锁获得为止。

下面是自旋锁的释放代码：

`Release_lock:`

```
    sync
    li r1,0
    stwcx r1,0,r3
    blr
```

代码比较简单，`stwcx` 指令把自旋锁清零，并清除 `Reserve` 比特。需要注意的是，`stwcx` 与 `Acquire_lock` 中的 `lwarx` 指令（实际上在 `test and set` 例程中）对应。

在上述实现中，还涉及到了两条指令 `isync` 和 `sync`，这两条指令用于完成上下文的同

步，在此不作详细描述，详细内容可参考 Power PC 的用户编程手册。

## 关键区段使用注意事项

在使用关键区段，对关键代码段进行保护的时候，需要遵循下列注意事项：

- 1、关键代码断不能太长，如果太长，可能会影响系统的整体效率。比如，在多 CPU 环境下，关键区段是采用忙等待的方式实现的，即如果有一个线程，试图修改已经被占有的全局数据，那么它必须首先等待，如果正在修改的线程长时间的占有保护锁，则会导致其它线程长时间的等待，浪费 CPU 资源。在单 CPU 环境下，关键区段是采用关闭中断的方式实现的，这样如果长时间的进行关键代码段的执行，会导致中断丢失，在实时系统中，这是不允许的；
- 2、在关键区段保护的代码段中，不能调用可能阻塞线程继续执行的系统调用。比如，不能在代码段中等待一个内核对象，这样可能会导致严重的资源浪费（CPU 资源），甚至可能导致死锁发生；
- 3、对于关键区段，只建议在操作系统核心的实现代码中使用，对于普通的应用程序（应用线程），不建议直接使用关键区段，而建议使用内核对象，比如 Event、Semaphore、Mutex 等，来实现线程之间的同步和关键资源的保护。

## Semaphore 概述

信号（semaphore）是一个内核同步对象，用来同步或保护共享资源的访问。通常情况下，对于 semaphore，设定一个初始值，比如为 N，对于每个请求等待 semaphore 的核心线程（通过调用 WaitForThisObject 函数），操作如下：

- 1、判断 N 是否小于或等于 0，如果不是，则  $N = N - 1$ ，并从等待操作中，直接返回，这时候，等待 semaphore 的线程得以继续执行；
- 2、如果 N 小于或等于 0，说明当前没有可利用资源，于是等待过程把请求的线程插入等待队列，然后执行一个重调度过程，需要注意的时，这种情况下，也对 N 进行递减操作，不过这时候 N 已经成为负值，因此，如果 N 大于 0，N 的数值代表了当前可用资源的数量，如果 N 小于 0，则 N 的绝对值，代表了当前正在等待 semaphore 资源的线程的数量（等待队列的长度）。

对于已经获得信号的线程，在执行完关键代码后，必须释放申请的 semaphore 资源（通过调用 ReleaseSemaphore 函数），释放资源的操作（ReleaseSemaphore）过程如下：

- 1、对 N 进行递增（加 1）操作，如果 N 大于 0，则说明当前没有线程等待 semaphore 资源，直接从释放函数中返回；

- 2、如果 N 小于或等于 0（递增之后），说明仍然有线程在等待该信号资源，于是从等待队列中，提取一个线程，把该线程修改为就绪（`KERNEL_THREAD_STATUS_READY`）状态，并插入就绪队列（这样，该线程在合适的时刻会被调度运行），然后返回。

可以看出，一次释放 semaphore 的操作，最多会唤醒一个等待线程，这与事件对象（EVENT）不同，事件对象的一次设置操作，可以唤醒所有等待该事件的线程。而如果 semaphore 的 N 的值为 1，则 semaphore 则演变为一个简单的 mutex 对象（互斥体对象），可以用户保护临界区的并发访问。只所以用简单的 mutex 对象来形容 N 为 1 的情况，是因为 semaphore 对象不支持 mutex 对象支持的所有功能（比如递归调用等），而仅仅支持 mutex 对象功能的一个子集。可参阅 mutex 对象的描述和实现部分，来了解这些差异。

semaphore 的等待操作，支持超时等待的方式，即等待的线程可以设定一个时间长度，如果能立即获取资源（等待成功），则直接返回，如果不能立即获取资源，而被阻塞，则在设定的时间超时后，如果资源仍然不能获得，则也会返回继续执行。这个功能在网络协议的实现中十分有用。

## Semaphore 对象的定义

Semaphore 是一个内核对象，因此需要从 `__COMMON_OBJECT` 对象继承，以利用 `__COMMON_OBJECT` 提供的服务，也便于将来向 MP（多处理器）扩展，而且 semaphore 是一个同步对象，因此需要从 `__COMMON_SYNCHRONIZATION_OBJECT` 继承，以提供一个通用的同步对象访问接口（当前版本中，主要是继承 `WaitForThisObject` 函数接口），因此，semaphore 的定义如下：

```
BEGIN_DEFINE_OBJECT(__SEMAPHORE)
    INHERIT_FROM_COMMON_OBJECT
    INHERIT_FROM_COMMON_SYNCHRONIZATION_OBJECT
    INT                                nCounter;
    __PRIORITY_QUEUE*                lpWaitingQueue;
    INT                                (*SetSemaphoreCounter)(__COMMON_OBJECT*
lpThis,INT nNewCounter);
    DWORD
```

```
(*WaitForThisObjectEx)(__COMMON_OBJECT*,DWORD);  
    INT                                (*ReleaseSemaphore)(__COMMON_OBJECT*);  
END_DEFINE_OBJECT()
```

其中，nCounter 是 semaphore 当前资源计数（即概述部分的 N 值），lpWaitingQueue 是一个等待队列，所有等待 semaphore 对象的线程，在没有获得资源（nCounter <= 0）的情况下，都将被阻塞，并排在该等待队列中。

由于 Semaphore 遵循 Hello China 的对象语义，可以通过 CreateObject(ObjectManager 的成员函数) 函数创建，而目前版本下，该函数(CreateObject) 并没有提供设置 nCounter 初始值的参数，因此，缺省情况下，每当一个 semaphore 对象被创建完成，其 nCounter 成员变量初始化为 1，为了改变这个缺省值，SetSemaphoreCounter 函数可以被调用，用来设置新的 nCounter 值，该函数(SetSemaphoreCounter) 在设置新的 nCounter 值的同时，返回原先 nCounter 的数值。

Semaphore 对象从 \_\_COMMON\_SYNCHRONIZATION\_OBJECT 对象继承，从而继承了 WaitForThisObject 方法，该方法用来完成 semaphore 对象的资源请求（等待），但是在当前版本下，该函数没有实现超时功能（函数参数中没有一个 dwTimeOut 的参数，用来指定超时时间），因此，为了实现超时功能，重新引入一个函数 WaitForThisObjectEx，该函数实现了 WaitForThisObject 的所有功能，并增加了超时功能。

ReleaseSemaphore 函数用来完成 semaphore 资源的释放工作，这个函数的调用，必须跟在 WaitForThisObject 之后，或者跟在 WaitForThisObjectEx 之后，但 WaitForThisObjectEx 的返回原因，不能是超时（即如果是因为超时返回，则不能调用该函数）。如果调用者先前没有调用 WaitForThisObject 或 WaitForThisObjectEx，而直接调用了 ReleaseSemaphore，则可能会导致资源不一致，产生问题。因此，目前版本的实现中，semaphore 不是一个支持安全调用的对象，使用者应该谨慎。

同样地，semaphore 对象从 \_\_COMMON\_OBJECT 对象继承了通用对象都必须实现的方法和变量，比如 Initialize、Uninitialize 函数等。在实现的时候，必须单独实现这些函数。

## Semaphore 对象的实现

在本节中，我们对 semaphore 对象的实现，进行详细描述，主要包括其初始化（Initialize）和非初始化（Uninitialize）操作、等待操作（WaitForThisObject 和 WaitForThisObjectEx）和释放操作（ReleaseSemaphore）进行描述。

### Initialize 和 Uninitialize 实现

Initialize 和 Uninitialize 两个函数，是 Hello China 的对象语义定义的，用于完成对象的一致创建和销毁。其中，Initialize 函数在对象被创建完成后调用，用来完成对象的初始化工作，而 Uninitialize 函数则在对象的销毁过程中调用，用来释放对象占用的资源。一般情况下，这两个函数执行相反的操作。

在 Semaphore 对象的实现中，Initialize 函数主要完成下列工作：

- 1、创建等待队列对象（\_\_PRIORITY\_QUEUE），并初始化该对象；
- 2、设置 semaphore 对象的 nCounter 成员变量为缺省值（当前缺省值为 1）；
- 3、设置函数指针 WaitForThisObject、WaitForThisObjectEx、ReleaseSemaphore、SetSemaphoreCounter 等函数指针的值，一般情况下，这些函数都作为静态函数，在一个模块（源文件）内实现。

下面是 Initialize 函数的实现：

```
static BOOL SemInitialize(__COMMON_OBJECT* lpObject)
{
    __SEMAPHORE*                lpSem                =
(__SEMAPHORE*)lpObject;

    __PRIORITY_QUEUE*            lpWaitingQueue        = NULL;
    BOOL                          bResult               = FALSE;

    if(NULL == lpObject)        //Parameter check.
        return FALSE;

    lpWaitingQueue = ObjectManager.CreateObject(&ObjectManager,
                                                NULL,
                                                OBJECT_TYPE_PRIORITY_QUEUE);
    if(NULL == lpWaitingQueue)    //Failed to create waiting queue.
```

```

        goto __TERMINAL;
    if(!lpWaitingQueue->Initialize((__COMMON_OBJECT*)lpWaitingQueue))
//Failed to initialize.
        goto __TERMINAL;
    lpSem->lpWaitingQueue = lpWaitingQueue;
    lpSem->nCounter        = DEFAULT_SEMAPHORE_COUNTER;
    lpSem->WaitForThisObject = WaitForSemObject;
    lpSem->WaitForThisObjectEx = WaitForSemObjectEx;
    lpSem->SetSemaphoreCounter = SetSemaphoreCounter;
    bResult  = TRUE;    //Indicate the whole process is successful.
__TERMINAL:
    if(!bResult)        //Initialize failed.
    {
        if(lpWaitingQueue)    //Have created the waiting queue,must destroy it.
        {
            ObjectManager.DestroyObject(&ObjectManager,(__COMMON_OBJECT*)lpWaitingQueue,
                )
        }
    }
    return bResult;
}

```

其中, `DEFAULT_SEMAPHORE_COUNTER` 是一个预先定义的宏, 当前版本下, 该数字定义为 1, 在 semaphore 创建完成后, 线程可以调用 `SetSemaphoreCounter` 函数, 来重新设置一个 `nCounter`, 以满足实际需要。

`Uninitialize` 函数的实现, 与 `Initialize` 的实现相反, 在 `Uninitialize` 函数中, 完成等待队列对象 (`lpWaitingQueue`) 的销毁工作, 在此不进行赘述。

## WaitForThisObject 的实现

该函数供内核线程调用, 用来完成 semaphore 对象资源的请求工作。该函数实现流程如下:

- 1、对 `nCounter` 施行递减操作, 即 `nCounter = nCounter - 1`, 并判断 `nCounter` 的结果, 如果结果大于或等于 0, 则说明当前尚有资源, 于是当前线程等待成功, 该函数返回成

功标志，使得当前线程不阻塞的继续运行；

- 2、如果 `nCounter` 递减后的结果小于 0，说明当前已经没有资可供使用，于是把当前线程的状态设置为阻塞状态，并放入等待队列 (`lpWaitingQueue`)，然后执行一个重新调度操作，选择其它就绪的线程投入运行。

需要注意的是，上述操作（包括递减 `nCounter`、判断 `nCounter` 的结果、阻塞当前线程等）是在一个原子操作内完成的，因为如果不这样，很可能产生不一致状态。

下面是该函数的实现：

```
static DWORD WaitForSemObject(__COMMON_OBJECT* lpObject)
{
    __SEMAPHORE*          lpSem          = lpObject;
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    DWORD                 dwFlags        = 0L;

    if(NULL == lpObject)    //Parameter check.
        return 0L;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);    //Enter atomic operations.
    lpSem->nCounter --;
    if(lpSem->nCounter >= 0)    //Have resource now.
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return 1L;
    }
    //
    //Now,there is not enough resource ,so block the current kernel thread.
    //
    lpKernelThread = KernelThreadManager.lpCurrentKernelThread;
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_BLOCKED;

    lpSem->lpWaitingQueue->InsertIntoQueue((__COMMON_OBJECT*)lpSem->lpWaitingQueue,
    e,
    (__COMMON_OBJECT*)lpKernelThread,
    0L);
```

```
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);  
KernelThreadManager.ScheduleFromProc(&lpKernelThread->Context);  
return 0L;  
}
```

需要注意的是，在上述实现中，从递减 `nCounter`、判断 `nCounter`、修改当前线程状态、把当前线程插入等待队列等操作，都是在一个临界区内完成的，即这些操作的过程中，不会被中断，这样可以充分确保所有这些操作能够原子的完成，可以看出，按照这样的实现，`semaphore` 对象不但是线程安全的，而且是中断安全的，即可以在中断上下文中调用 `semaphore` 的操作函数。

## WaitForThisObjectEx 的实现

与 `WaitForThisObject` 不同的是，`WaitForThisObjectEx` 支持超时等待，即调用者（通常情况下是一个线程）可以设定一个时间参数，如果资源能够在时间参数超时内（从调用开始计时）可用，则返回资源可用指示，否则，如果在定时器超时前，资源仍不可用，则返回定时器超时指示。

如果把超时参数设置为 0，则该函数的行为，与 `WaitForThisObject` 类似，永远等待，直到资源变得可用。因此，对于超时参数为 0 的情况，该函数直接调用 `WaitForThisObject`，而对于超时参数不为 0 的情况，该函数需要完成下列操作：

- 1、递减 `nCounter` 变量，并判断递减结果；
- 2、如果递减后的结果大于或者等于 0，则说明尚有足够的资源，于是直接返回资源成功获取指示（返回 1）；
- 3、如果递减后的结果小于 0，则说明当前已经没有足够的资源可供使用，于是准备超时等待操作；
- 4、首先，该函数设定一个一次性定时器，该定时器的超时参数设置为调用者传递过来的超时参数，并设定一个回调函数，以在定时器超时时调用；
- 5、然后该函数设置当前线程状态为阻塞，并把当前线程插入等待队列；
- 6、调用 `ScheduleFromProc` 例程，重新调度；
- 7、`ScheduleFromProc` 返回后（当前线程重新被激活），判断激活原因是超时，还是已经获得了资源；
- 8、如果激活原因是获得了资源，则返回资源获得信息（返回 1），否则，返回超时信息 `SEMAPHORE_TIMEOUT`，以指示用户等待超时。

该函数调用返回后，调用者需要判断该函数的返回原因（是获得了资源，还是因为超时返回），如果是因为在超时前获得了请求的资源，则在后续的某个恰当的地方，需要

执行 `ReleaseSemaphore`，以释放获得的资源，如果是因为超时返回，则不需要调用 `ReleaseSemaphore` 函数，因为当前线程从来就没有获得过 semaphore 资源。

在上述描述中，提到了一个定时器超时时，调用的回调函数，该函数定义如下：

```
DWORD SemCallBack(LPVOID lpParam);
```

一旦定时器超时，该函数就会被调用（在时钟中断上下文中被调用，因此，该函数在实现的时候，一定要短小紧凑，以减少处理时间），该函数完成如下功能：

- 1、从 semaphore 对象的等待队列中，把设定该定时器的线程（即调用 `WaitForThisObjectEx`，并设定超时参数的线程）删除，并修改该线程的状态为 `READY`，插入就绪队列，其实是一个唤醒过程；
- 2、设置线程的唤醒原因为超时。

这样被唤醒的线程在合适的时候，就会被调度运行。细心的读者可能发现，上述回调函数的处理，存在一个问题，假设在定时器没有超时前，有另外一个占有 semaphore 对象的线程，释放了 semaphore 资源（调用 `ReleaseSemaphore` 函数），这样处于等待状态的线程（假设该线程为 TA）被唤醒，并插入就绪队列，但还未被调度运行，这个时候定时器超时，即上述事件按照下列时序发生：

- 1、另外一个占有 semaphore 资源的线程释放 semaphore 资源；
- 2、等待 semaphore 资源（超时等待）的线程（TA）被唤醒，并插入就绪队列，但还未被调度运行；
- 3、TA 设定的定时器（其实是 TA 以超时方式调用 `WaitForThisObjectEx` 函数，`WaitForThisObjectEx` 参数设定定时器）超时，回调函数在中断上下文中被调用；
- 4、TA 被重新调度，投入运行。

在上述情形中，回调函数在从 semaphore 对象的就绪队列中删除 TA 的时候，就会失败（因为 TA 已经不在 semaphore 对象的等待队列里面了）。这个时候，说明 TA 已经获得了 semaphore 资源，被重新调度，因此，需要设置 TA 被唤醒的原因为获得了资源，而不应该是超时。

综上所述，回调函数的处理过程应该如下：

- 1、调用 `DeleteFromQueue` 函数，从 semaphore 对象的等待队列中，删除设定超时等待的线程；
- 2、如果 `DeleteFromQueue` 返回 `TRUE`，则说明该线程尚未被唤醒，则回调函数设置该线程状态为就绪（`KERNEL_THREAD_STATUS_READY`），插入就绪队列，并设置线

程唤醒原因为超时 (SEMAPHORE\_WAIT\_TIMEOUT);

- 3、如果 DeleteFromQueue 返回 FALSE, 说明等待队列中已经不存在 TA 线程, 即 TA 线程可能已经获得了 semaphore 资源, 早已被唤醒, 因此, 这个时候, 就需要设定返回原因为获得资源 (SEMAPHORE\_WAIT\_RESOURCE), 并返回。

另外一种可能的时序, 就是:

- 1、等待 semaphore 资源的线程 TA, 由于获得了资源, 而被唤醒;
- 2、在一个时钟中断内, TA 得到调度投入运行, 这个时候 TA 设定的定时器还未超时。

这种情况下, 在 TA 投入运行后, 第一件事情就是, 判断被唤醒的原因 (得到资源或者超时), 如果是得到资源, 则删除定时器对象 (因为这个时候, 定时器还未超时, 定时器对象仍然存在), 如果是超时, 则说明定时器已经超时, 回调函数已经被调用, 而且定时器对象已经被删除 (一次定时器在超时后, 立即被删除), 因此这个时候函数只需要返回即可。

综上所述, semaphore 的回调函数实现方式如下:

```
static DWORD SemCallBack(LPVOID lpParam)
{
    __SEMAPHORE_CALLBACK_PARAM* lpSemParam    = lpParam;
    __KERNEL_THREAD_OBJECT*      lpKernelThread = NULL;
    __SEMAPHORE*                 lpSem         = NULL;

    if(NULL == lpParam)          //Parameter check.
        return 0L;

    lpKernelThread = lpSemParam->lpKernelThread;
    lpSem          = lpSemParam->lpSemaphore;

    if(!lpSem->lpWaitingQueue->DeleteFromQueue((__COMMON_OBJECT*)lpSem->lpWaiting
    Queue,(__COMMON_OBJECT*)lpKernelThread)) //The kernel thread is not exist in waiting
    queue.

        return;
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_READY;
    KernelThreadManager.lpReadyQueue->InsertIntoQueue(
```

```

    (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
    (__COMMON_OBJECT*)lpKernelThread,
    lpKernelThread->dwScheduleCounter);    //Insert into ready queue.

    lpSemParam->dwWakeupReason = SEMAPHORE_WAIT_TIMEOUT;
    return 1L;
}

```

可以看出，该函数首先视图从等待队列中删除等待线程，如果成功，则设置超时原因，否则不做任何动作，直接返回。

`__SEMAPHORE_CALL_BACK` 结果是一个局部定义的数据结构，用来完成 callback 函数的参数传递工作，该数据结构包含三个成员：等待线程（`lpKernelThread`）、线程被唤醒的原因（`dwWakeupReason`）以及 semaphore 对象指针（`lpSem`）。

下列是 `WaitForThisObjectEx` 函数的实现：

```

static  DWORD  WaitForSemObjectEx(__COMMON_OBJECT*  lpObject,DWORD
dwTimeOut)
{
    __SEMAPHORE*          lpSem          = lpObject;
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    __SEMAPHORE_CALLBACK_PARAM*
                                lpCallbackParam = NULL;
    DWORD                  dwFlags          = 0L;
    __TIMER_OBJECT*        lpTimerObject    = NULL;

    if(0 == dwTimeOut)        //Without time out waiting, so the same as
WaitForThisObject.
        return WaitForSemObject(lpObject);

    if(NULL == lpObject) //Parameter check.
        return 0L;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);

```

```

    lpSem->nCounter --;
    if(lpSem >= 0)    //There is enough resource.
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return SEMAPHORE_WAIT_RESOURCE;
    }
    //
    //There is not enough resource,so must block the current kernel thread.
    //
    lpKernelThread = KernelThreadManager.lpCurrentKernelThread;
    lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_BLOCKED;
    lpCallbackParam = (__SEMAPHORE_CALLBACK_PARAM*)KMemAlloc(
        sizeof(__SEMAPHORE_CALLBACK_PARAM),
        KMEM_SIZE_TYPE_ANY);
    if(NULL == lpCallbackParam)    //Can not allocate memory.
    {
        lpKernelThread->dwThreadStatus =
        KERNEL_THREAD_STATUS_RUNNING;
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return SEMAPHORE_WAIT_FAILED;
    }
    lpCallbackParam->lpSem = lpSem;
    lpCallbackParam->lpKernelThread = lpKernelThread;
    lpCallbackParam->dwWakeupReason = SEMAPHORE_WAIT_RESOURCE;
    lpTimerObject = (__TIMER_OBJECT*)System.SetTimer(
        (__COMMON_OBJECT*)&System,
        lpKernelThread,
        SEMAPHORE_TIMER_ID,
        dwTimeOut,
        SemCallback,    //Call back routine.
        lpCallbackParam,
        TIMER_FLAGS_ONCE);
    if(NULL == lpTimerObject)    //Failed to set timer object.
    {

```

```

    lpKernelThread = KERNEL_THREAD_STATUS_RUNNING;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    KMemFree((LPVOID)lpCallbackParam,KMEM_SIZE_TYPE_ANY,0L);
    return SEMAPHORE_WAIT_FAILED;
}
lpSem->lpWaitingQueue->InsertIntoQueue((__COMMON_OBJECT*)
    lpSem->lpWaitingQueue,
    (__COMMON_OBJECT*)lpKernelThread,
    0L);
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

```

KernelThreadManager.ScheduleFromProc(&lpKernelThread->KernelThreadContext);

//

**//The following code will be executed after the current kernel thread is waken**

**up.**

//

```
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
```

```
switch(lpCallbackParam->dwWakeupReason)
```

```
{
```

```
    case SEMAPHORE_WAIT_RESOURCE:        //The thread is waken up
```

because of resource available.

```
    System.CancelTimer((__COMMON_OBJECT*)&System,
```

```
        (__COMMON_OBJECT*)lpTimerObject);    //Cancel
```

timer.

```
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
```

```
KMemFree((LPVOID)lpCallbackParam,KMEM_SIZE_TYPE_ANY,0L);
```

```
return SEMAPHORE_WAIT_RESOURCE;
```

```
case SEMAPHORE_WAIT_TIMEOUT:
```

```
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
```

```
    KMemFree((LPVOID)lpCallbackParam,KMEM_SIZE_TYPE_ANY,0L);
```

```
    return SEMAPHORE_WAIT_TIMEOUT;
```

```
default:    //Should not occur for ever.
```

```
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
```

```

        KMemFree((LPVOID)lpCallbackParam,KMEM_SIZE_TYPE_ANY,0L);
        return SEMAPHORE_WAIT_FAILED;
    }
}

```

该函数首先检查是否有足够的可用资源，如果是，则直接返回成功标志 (SEMAPHORE\_WAIT\_RESOURCE)，否则，设置定时器，并阻塞当前线程。

一个注意的地方就是，该函数完成 lpCallbackParam 的内存分配后，把 dwWakeupReason 初始化为 SEMAPHORE\_WAIT\_RESOURCE，并把 lpCallbackParam 作为参数传递给 SetTimer 函数，这样如果 SemCallback 被调用，则 dwWakeupReason 将被修改为 TIMEOUT，否则，会一致保持 SEMAPHORE\_WAIT\_RESOURCE，这样当该线程被唤醒之后（代码中黑色字体注释部分开始），就可以根据 dwWakeupReason 的当前值，确定不同的动作流程。

## ReleaseSemaphore 的实现

该函数的实现相对简单，在这个函数中，首先对 nCounter 施行递增操作，然后判断递增后的结果是否是大于 0。如果大于 0，则说明当前没有线程在等待资源，于是接返回，否则，说明有线程在等待资源，于是从等待队列中，提取第一个等待的线程，标记为就绪状态，插入就绪队列，然后再返回。实现如下：

```

static INT ReleaseSemaphore(__COMMON_OBJECT* lpObject)
{
    __SEMAPHORE*          lpSem      = lpObject;
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    DWORD                dwFlags = 0L;

    if(NULL == lpObject)    //Parameter check.
        return -1;
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    lpSem->nCounter ++;
    if(lpSem->nCounter > 0)    //No kernel thread waiting for this semaphore now.

```

```

{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return 0;
}
//
//There is one kernel thread waiting for this object at least.
//
lpKernelThread = lpSem->lpWaitingQueue->GetHeaderElement(
    (__COMMON_OBJECT*)lpSem->lpWaitingQueue),
if(NULL == lpKernelThread)    //----- ** BUG!!!! ** -----
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return -1;
}
lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_READY;
KernelThreadManager.lpReadyQueue->InsertIntoQueue((__COMMON_OBJECT*)
    KernelThreadManager.lpReadyQueue,
    (__COMMON_OBJECT*)lpKernelThread,
    0L);    //Insert into ready queue.
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
return 0;
}

```

需要注意的是，该函数没有进行对应性安全检查，即如果一个线程事先没有调用 `WaitForThisObject` 或 `WaitForThisObjectEx`，而直接调用该函数，则会出现 semaphore 内部状态的不一致，导致异常行为，因此，在使用该对象的时候，一定要注意这一点。

# 中断和定时处理机制的实现

## —— The implementation of Interrupt and Timer

**by GarryXin**  
**2005/10/05**

目录

中断和异常概述	7
硬件相关部分处理	7
IA32 中断处理过程	7
IDT初始化	7
硬件无关部分处理	8
系统对象和中断对象	8
中断调度过程	8
缺省中断处理函数	8
对外服务接口	8
几个注意事项	8
Power PC的异常处理机制	8
PPC 异常处理机制概述	8
Power PC异常的分类	9
异常的处理和返回	9
定时器概述	9
SetTimer调用	9
CancelTimer调用	9
ResetTimer调用	9
设置定时器操作	9
定时器超时处理	9
定时器取消处理	10
定时器复位	10
定时器注意事项	10

中断和异常处理以及定时机制，是操作系统必须实现的最重要的核心功能之一。在 **Hello China V1.0** 的实现中，实现了一个可以移植到不同 CPU 上的中断和异常处理机制模型，在本章中，我们将对这个模型进行详细介绍，另外，对 **Power PC CPU** 的中断和异常机制，也做了一个简单的介绍。

在本章的后面部分，我们对 **Hello China** 的定时机制及其应用，做了详细介绍。

## 中断和异常概述

中断和异常是系统在运行过程中，可能发生的外部或内部事件。一般情况下，中断是由外部设备引起的，比如，键盘设备，每当计算机系统的使用者按下一个键，或者放开一个键，键盘设备（严格来说，应该是控制键盘的芯片）就会生成一个中断，并通知 CPU。

而异常则一般是由软件造成的，泛指软件运行过程中，产生的不正常的事件。比如，最容易理解的是除法运算，如果出现了除数为 0 的情况，则会引发一个异常。另外一个很常见的异常，就是缺页异常。为了实现虚拟内存模型，操作系统可以把内存的一部分内容，暂时存储在外部存储设备上（比如，硬盘），从而腾出更多的内存空间为软件的运行服务。这样一旦一条指令访问了不在物理内存中的内存地址（比如，已经被暂时存储在硬盘上的数据），则会引发一个缺页异常。

一旦检测到中断或异常发生，CPU 就会中断当前的处理顺序，并根据中断或异常的类型，转移到相应的处理程序。需要注意的是，并不是中断或异常发生后，CPU 马上跳转到相应的处理程序，而是只有在指令的执行边界，CPU 才会检查是否有异常或中断发生，如果没有，CPU 继续执行下一条指令，即一旦有中断或异常发生，会引起 CPU 设置内部的相应寄存器位，然后继续执行当前指令（中断发生时，正在执行的指令，或者引发异常的指令），只有在当前指令执行完毕之后，相应的中断或异常，才有机会得到处理。

不同的 CPU 类型，其异常或中断的处理机制是不同的，比如，针对 **Intel CPU**，其对系统中可能产生的异常进行了编号，一旦异常发生，则 CPU 根据异常类型，找到对应的编号，然后再根据异常编号，查找一个叫做中断描述符表（**IDT**）的表格，找到对应的处理程序，然后跳转到具体的处理程序，对于中断，也是采取类似的方式，不过中断的编号（俗称中断向量号），是由硬件决定的。而 **Power PC**，则不论对异常，还是对中断，都调用同一个处理程序（所有的中断和异常，都调用同一个处理程序），然后处理程序再检测中断或异常的类型，进行进一步的分类处理，在此，我们称这种处理方式中断处理链方式。

在 Hello China 的设计中，充分考虑了这两种典型的中断和异常模型，采用了中断向量表跟中断处理链方式进行结合的实现方式，即首先有一个中断向量表，这样如果目标 CPU 是 Intel 系列的 CPU，则直接根据中断或异常的向量号，定位到一个中断向量表项，在每个中断向量表的表项中，又保存了一个中断对象链表，这样就可以实现链表方式的中断处理模型。因此，其可移植性较好。

在本部分中，我们以 Intel CPU（IA32 构架）为目标 CPU，详细介绍 Hello China 的中断处理机制，并介绍中断处理机制的服务提供接口。在本章的最后，简单介绍 Power PC CPU 的异常处理机制，以跟 IA32 的异常处理进行比较。

## 硬件相关部分处理

### IA32 中断处理过程

所谓硬件相关部分处理，指的是针对不同的 CPU 平台，所采用的不同的中断处理方法。比如，针对 Intel IA32 构架的 CPU，需要建立 IDT（中断描述符表）表，并填写每个表项，针对 PPC 的 CPU，则需要初始化特定的寄存器等。在这一部分中，我们针对 IA32 构架的 CPU，进行描述，需要说明的是，硬件相关处理，仅仅是为了迎合不同的硬件的中断和异常模型，而引入的“第一层”处理，这部分处理比较简单，一般使用汇编语言实现，这部分的主要功能就是，完成硬件部分的处理，跟 Hello China 本身中断处理机制之间的连接。

IA32 CPU 采用中断描述符表（IDT）的方式，对中断和异常进行处理。IDT 是位于内存中的数据结构，该结构由 256 个中断描述符（或异常描述符）组成，每个中断描述符是一个 64 比特的数据结构，其每个比特的内容及含义，都是硬件严格定义的，在 CPU 内部，有一个很重要的寄存器：IDTR，即中断描述符表寄存器，这个寄存器保存了中断描述符表的起始物理地址（物理内存地址），一旦 CPU 检测到中断或异常发生，则进入如下的中断或异常处理流程：

- 1、把下一条将要执行的指令地址（EIP 寄存器）和代码段寄存器（CS），以及标志寄存器（EFLAGS）压入堆栈（当前线程堆栈）；
- 2、根据中断号或异常号，定位到具体的中断描述符（具体定位方法为：中断/异常向量号乘以 8，加上 IDTR 寄存器的值）；
- 3、中断描述符里面，存放了处理该中断或异常的处理程序的起始地址（以及所在的代码段），CPU 把起始地址读入到 EIP 寄存器（并更新代码段寄存器 CS），然后开始执行中断处理程序（实际上是一个跳转的过程）；
- 4、处理程序处理完毕后，使用 iret 指令，从中断处理程序中返回（该指令的含义，请参考本书“Hello China 线程的实现”一章）；

5、CPU 继续执行中断处理前的任务。

需要注意的是，上述处理过程，仅仅是一个简单的描述，实际上，根据不同的任务模型和地址模型，以及不同的 CPU 工作模式，中断处理方式各不相同，且十分复杂，但由于 Hello China 的设计，充分抽象了 CPU 的具体特征，最小化的利用了 CPU 的硬件特性，而把相关的特性放到软件中完成（便于移植到不同的 CPU），因此，上述简单的处理过程描述，在 Hello China 的实现中，已经足够。

这样可以看出，为了完成对 IA32 平台的中断处理模块的初始化，只需要完成下列两件事情：

- 1、 正确形成 IDT；
- 2、 把 IDT 的物理地址，填写到 IDTR 寄存器中。

其中，第二步非常简单，只需要一条指令就可完成任务：

```
lidt idt_addr
```

其中，idt\_addr 就是 IDT 的物理地址。接下来，我们对 IDT 的填写进行详细描述。

IDT 初始化

下面是 IA32 体系构架定义的中断描述符（Interrupt Descriptor）的结构：

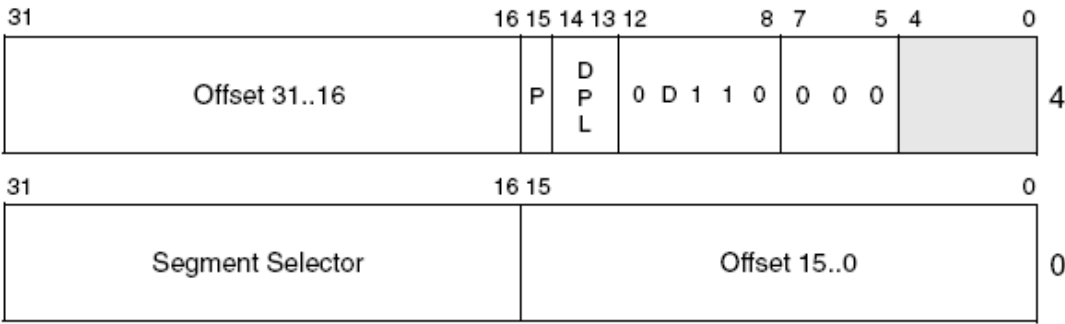


图 8-1 中断描述符的结构

其中，各个字段定义如下：

- **Segment Selector**: 段选择符，指明了中断处理程序所在的代码段；
- **Offset**: 中断处理程序在段内的偏移量；
- **DPL: Descriptor Privilege Level**，即可以调用该中断描述符的当前处理级；
- **P: Present flags**，如果段描述符在内存内，则该标志置为 1，如果当前描述符不存在，则设置为 0；
- **D**: 门尺寸，如果为 0，则是一个 16bit 的段描述符，否则是 32bit 描述符。

需要注意的是，**Offset** 字段长 32 比特，被分成了两部分。

在 **Hello China** 的当前实现中，所有中断和异常处理程序，都位于代码断内，即段描述符索引为 0x08（详细信息参考 **Hello China** 的初始化部分），而当前 **Hello China** 的实现中，没有使用 **IA32** 提供的优先级保护，因此，所有涉及到 **DPL** 字段的地方，都设置为 0，**D** 比特设置为 1（32 位操作系统），**P** 比特也设置为 1，因为当前版本的实现中，所有中断描述符都是合法的（存在于内存中的）。

剩下的唯一需要确定的字段，就是 **Offset** 字段，即中断处理程序在代码段中的位置。在当前版本的实现中，方便起见，为中断描述符中前 48 个中断描述符项，定义了 48 个处理程序（按照 **IA32** 的定义，目前中断描述符表中，前 32 个（0—31）为异常处理描述符，后面 224 个（32—255）为用户定义的中断描述符，因此，在当前版本的实现中，只定义了 48 个中断和异常处理程序，因为一般的 PC 机上，只有 16 个外部中断，分别对应到中断描述符表中的第 32 到 47 个中断描述符），目前情况下，这些中断或异常处理程序，完成的功能非常有限，主要完成下列功能：

- 1、保存所有的通用寄存器；
- 2、把当前中断向量号或异常向量号，压入堆栈；
- 3、把当前堆栈指针（**ESP**）压入堆栈；
- 4、调用同一个中断或异常处理程序（这个处理程序，采用 C 语言实现）；
- 5、调用通用处理程序返回后，恢复保存的通用寄存器；
- 6、如果是中断处理程序，则通知中断控制器（8259 芯片），中断处理完毕；
- 7、从中断中返回。

下面是一个典型的中断处理程序：

```
np_int20:
    push eax
    cmp dword [gl_general_int_handler],0x00000000
    jz .ll_continue
```

```

push ebx                                ;;The following code saves the general
                                        ;;registers.

push ecx
push edx
push esi
push edi
push ebp
mov eax,esp
push eax
mov eax,0x20
push eax
call dword [gl_general_int_handler]
pop eax                                ;;Restore the general registers.
pop eax
mov esp,eax
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
.ll_continue:

mov al,0x20                            ;;Indicate the interrupt chip we have fin-
                                        ;;ished handle the interrupt.
                                        ;;;-)

out 0x20,al
out 0xa0,al
pop eax
iret

```

这是一段汇编代码，采用 NASM 进行编译。在这段程序的开始，首先压入 EAX 寄存器，然后判断 gl\_general\_int\_handler 是否为 0，如果为 0，则直接跳转到.ll\_continue（这是一个局部标号）处。

`gl_general_int_hander` 是在 `MINIKER.ASM` 文件中定义的一个 32 比特的全局变量，这个变量用来保存通用的中断和异常处理程序，这个通用的中断和异常处理程序，采用 C 语言编写，在操作系统初始化的时候，使用通用处理程序的地址，初始化 `gl_general_int_handler`。缺省情况下（未初始化的情况下），`gl_general_int_handler` 的值是 0，这样上述汇编代码就很容易理解了，首先判断，是否对 `gl_general_int_handler` 进行了初始化，如果没有，则直接跳转到 `.ll_continue` 处，直接解除中断，如果进行了初始化，即 `gl_general_int_handler` 的值不为 0，则进行正常的处理操作，包括保存通用寄存器，压入中断向量号（黑体标出的汇编语句），然后调用通用的中断和异常处理程序 `gl_general_int_handler`。从 `gl_general_int_handler` 返回后，再执行反向的恢复寄存器操作，然后解除中断（通过向中断控制芯片 8259 发送解除信号），并从中断中返回。需要注意的是，所有的中断处理程序（32—47），都是类似的，唯一不同的是，不同的中断处理程序，压入堆栈的中断向量号不同（代码中黑色部分）。

一个外部中断发生后，CPU 依次把 `EFLAGS`、`CS`、`EIP` 压入堆栈，然后根据中断向量号，从中断描述符表中，找到中断描述符，从中断描述符中，提取出中断处理程序的代码段选择符（`segment selector`）和中断处理程序，然后跳转到中断处理程序，也就是上述汇编语言编写的程序，继续执行。因此，在中断发生后，上述处理程序还未执行前，中断发生后的堆栈框架如下：

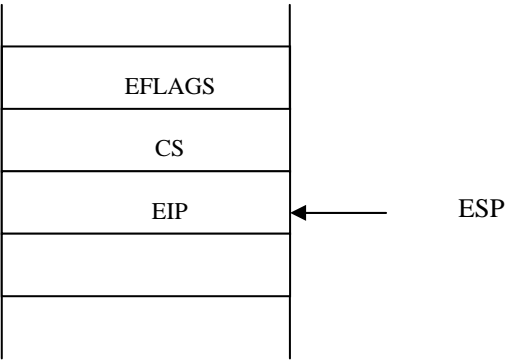


图 8-2 中断发生后的堆栈框架

一旦中断处理程序（上面描述的汇编语言程序）被执行，在实际调用 `gl_general_int_handler` 之前，形成如下的堆栈框架：

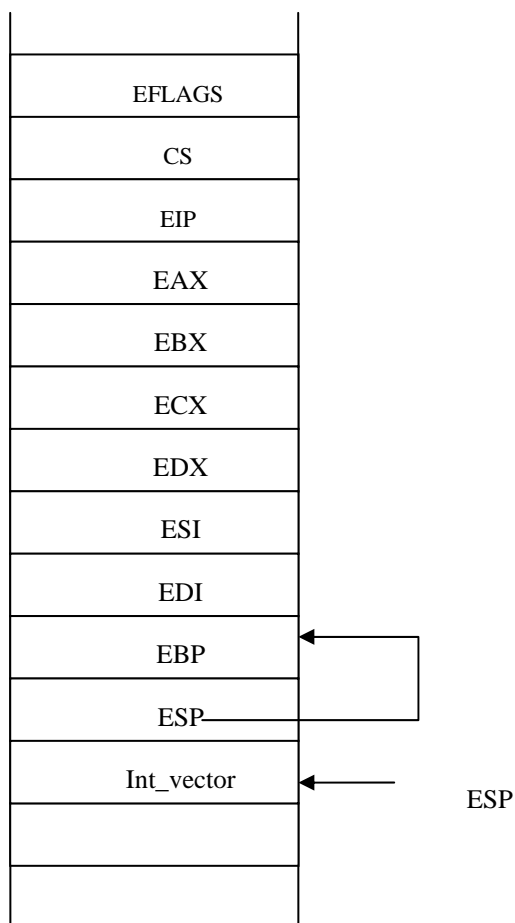


图 8-3 执行中断处理程序前建立的堆栈框架

其中，在上述堆栈框架中，保存的 ESP 的值，是压入 EBP 之后的 ESP 的值（图形中的折线示意位置），之所以保存 ESP 的值，是把 ESP 作为通用中断和异常处理程序的一个参数，来传递给通用中断和异常处理程序，这样通用的中断和异常处理程序，就可以访问堆栈框架了。下面是通用异常和中断处理程序的原型：

```
VOID GeneralIntHandler(DWORD dwVector,LPVOID lpEsp);
```

这样一切就明了了，汇编语言编写的中断处理程序，只是建立起一个堆栈框架（保存了通用寄存器的值），然后把 `esp` 寄存器的值和当前发生的中断的中断向量号，压入堆栈，作为 `GeneralIntHandler` 的参数，最后调用 `GeneralIntHandler (gl_general_int_handler)`。在 `GeneralIntHandler` 中，就可以通过 `dwVector` 和 `lpEsp` 来直接访问中断向量号和堆栈框架了。

对于异常的处理，与中断处理类似，唯一不同的是，对于异常的处理，在异常处理程序的最后，不用向中断控制器芯片发命令，来解除中断，下面是一个异常处理程序：

```
np_int0E:
    push eax
    cmp dword [gl_general_int_handler],0x00000000
    jz .ll_continue
    push ebx                                ;;The following code saves the general
                                           ;;registers.

    push ecx
    push edx
    push esi
    push edi
    push ebp
    mov eax,esp
    push eax
    mov eax,0x0E
    push eax
    call dword [gl_general_int_handler]
    pop eax                                ;;Restore the general registers.
    pop eax
    mov esp,eax
    pop ebp
    pop edi
    pop esi
```

```

    pop edx
    pop ecx
    pop ebx
.ll_continue:
    pop eax
    iret

```

需要进一步说明的是，对于异常的处理，IA32 CPU 会根据异常类型的不同，有选择的往堆栈中压入一个异常错误号，这样就导致了异常发生后的堆栈框架，与中断发生后的堆栈框架不一致（因为异常发生后，堆栈中比中断发生后多了一个错误号），因此需要特殊的处理。但在当前 **Hello China** 的实现中，没有考虑这种情况，因为一旦异常发生，按照当前版本的实现，只是打印出引发异常的上下文信息，然后停机（HLT 指令）。后续版本的实现中，需要充分考虑这种区别，按照现在的设计，充分考虑了这种特殊的情况，后续如果需要，这种特殊的处理，是很容易被支持的。

这样把各个中断和异常处理程序编写完成之后，剩下的任务就十分简单了，只需要把每个中断和异常的处理程序的地址（偏移），填写在相应的中断描述符的 Offset 字段即可。这项任务十分简单，在当前版本的实现中，是通过一个汇编语言例程（np\_fill\_idt）来实现的，在此不做赘述。

最后，我们把 gl\_general\_int\_handler 和 GeneralIntHandler 之间的关系说明一下。gl\_general\_int\_handler 是在 MINIKER.ASM 文件中声明的一个全局变量，并被初始化为 0，与其它全局变量不同的是，gl\_general\_int\_handler 被声明在了固定的位置，即 MINIKER.BIN 文件的末尾处，而当前版本下，MINIKER.BIN 文件（MINIKER.ASM 编译后形成的二进制文件）大小固定为 48K，因此，gl\_general\_int\_handler 相对于 MINIKER.BIN 文件的偏移，就是 48K - 4 位置处。

而 GeneralIntHandler 则是一个 C 语言函数，被编译在 MASTER.BIN 文件中。在操作系统初始化的时候，MINIKER.BIN 被加载到内存地址 1M 开始的地方，而 MASTER.BIN 则被加载到物理内存 0x00110000 位置处（1M+64K），如下所示：

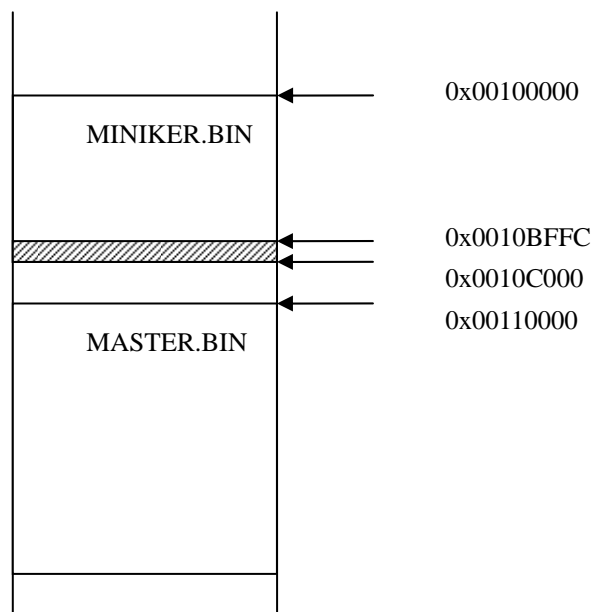


图 8-4 Hello China 相关模块在内存中的位置

其中，阴影部分是在 MINIKER.ASM 中，声明的 `gl_general_int_handler` 变量的位置。这样在 MASTER.BIN 初始化的时候，直接把 `GeneralIntHandler` 的值（其实是一个函数指针值）填写在 `gl_general_int_handler` 处，这样就完成了 `gl_general_int_handler` 的初始化。在 Hello China 的当前实现中，为了连接 MINIKER.BIN 和 MASTER.BIN 两个模块，大量的采用了这种方式。

## 硬件无关部分处理

### 系统对象和中断对象

当前版本的 Hello China，大量的采用了面向对象的思想进行设计，对于系统相关的一些功能和任务，比如定时处理、中断处理等，都封装到一个系统对象中，即 `System` 对象。该对象维护了所有中断处理相关的数据结构、函数等。

下面是 `System` 对象的定义：



```
lpKernelThread,

                                DWORD                dwTimerID,
                                DWORD                dwTimeSpan,
                                __DIRECT_TIMER_HANDLER
DirectTimerHandler,

                                LPVOID
lpHandlerParam,

                                DWORD
dwTimerFlags

                                );
VOID                (*CancelTimer)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpTimer);

END_DEFINE_OBJECT()
```

其中，用黑色字体标出来的相关代码，是中断处理相关的定义，包括一个中断对象数组（lpInterruptVector），三个完成中断调度以及中断服务相关的函数。

在这里，我们先介绍 lpInterruptVector。这是一个中断对象数组，所谓中断对象，是Hello China 定义，用来描述一个中断处理函数的对象，该对象定义如下：

```
BEGIN_DEFINE_OBJECT(__INTERRUPT_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    __INTERRUPT_OBJECT*        lpPrevInterruptObject;
    __INTERRUPT_OBJECT*        lpNextInterruptObject;
    UCHAR                      ucVector;
    BOOL                        (*InterruptHandler)(LPVOID lpParam, LPVOID
lpEsp);
    LPVOID                      lpHandlerParam;
END_DEFINE_OBJECT()
```

lpPrevInterruptObject和lpNextInterruptObject是用来把中断对象连接到双向链表中的指针，该对象从\_\_COMMON\_OBJECT对象继承，因此，遵循Hello China的对象语义，可以通过CreateObject方法（ObjectManager对象提供）创建。ucVector是中断向量，InterruptHandler则是真正的中断处理函数，lpHandlerParam则是中断处理函数的参数。

一般情况下,一个设备驱动程序(或者其它实体)如果想连接一个中断,则调用System对象提供的ConnectInterrupt函数,在这个函数里,指定了ucVector变量、InterruptHandler变量和相关的参数,ConnectInterrupt于是调用CreateObject函数,创建一个中断对象,然后把相关的变量初始化(包括ucVector、InterruptHandler等),并以ucVector变量为索引,定位到lpInterruptVector数组的一个特定元素(lpInterruptVector[ucVector]),这个元素是一个中断对象指针,指向一个中断对象链表,于是ConnectInterrupt就把新创建的对象,插入到这个链表中。

按照这种处理方式,系统中的所有中断对象,按照下列形式组织:

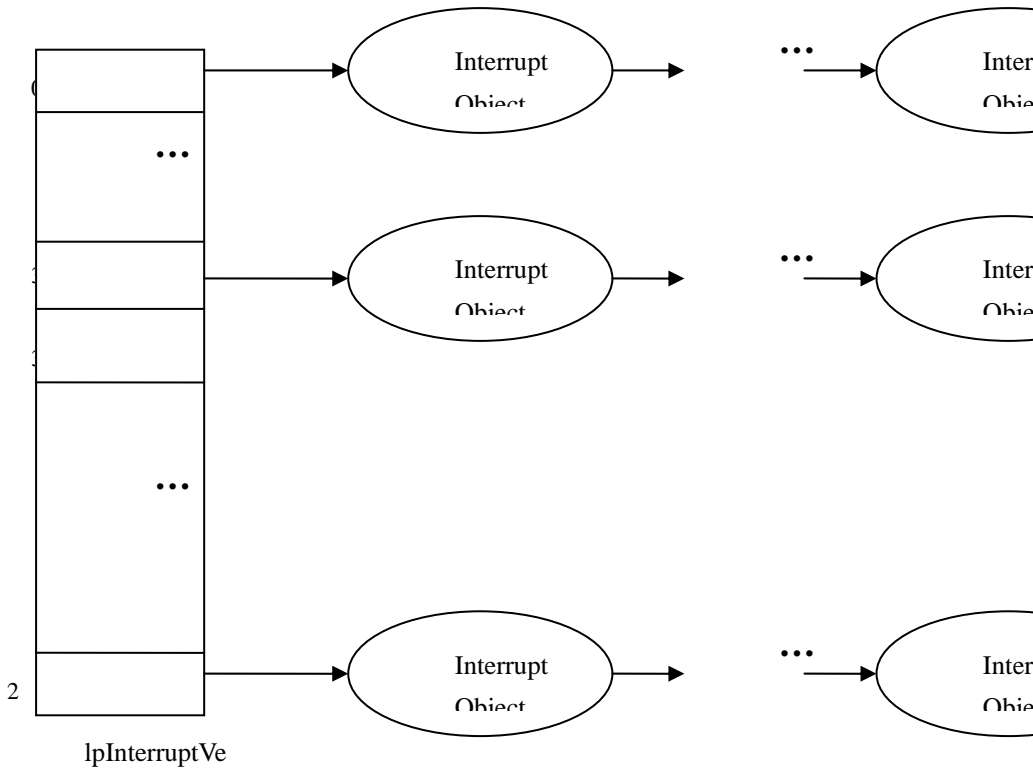


图 8-5 Hello China 中断对象的组织

其中, `lpInterruptVector` 是一个数组 (System 对象的一个成员), 其元素是中断对

象指针，系统中的所有中断对象，按照中断向量号（ucVector），被插入到相应的链表中。

## 中断调度过程

一旦中断或异常发生，CPU 首先根据中断或异常向量号，找到对应的描述符，从描述符中提取出中断或异常处理程序，然后把控制转移到中断或异常处理程序。这里的中断或异常处理程序，就是上面介绍的使用汇编语言编写的中断处理程序。

根据上面的描述，所有的异常和中断的处理，都会调用一个通用的中断异常处理程序 `gl_general_int_handler`，而目前情况下，该处理程序在 `MASTER.BIN` 中，使用 C 语言实现，即 `GeneralIntHandler` 函数，该函数的实现如下：

```
VOID GeneralIntHandler(DWORD dwVector, LPVOID lpEsp)
{
    UCHAR    ucVector = LOBYTE(LOWORD(dwVector));

    System.DispatchInterrupt((__COMMON_OBJECT*)&System,
        lpEsp,
        ucVector);
}
```

可见，该函数没有进行过多的处理，而是调用了 `System` 对象的 `DispatchInterrupt` 函数（见 `System` 对象的定义）。`DispatchInterrupt` 函数做如下处理：

- 1、根据中断向量号 `ucVector`，定位到特定的中断对象链表；
- 2、如果中断链表中没有任何中断对象，则调用一个缺省的中断或异常处理程序；
- 3、否则，依次调用该链表中，中断对象的中断处理函数，直到有一个中断处理函数返回 `TRUE`。

代码如下：

```
static VOID DispatchInterrupt(__COMMON_OBJECT* lpThis,
                             LPVOID          lpEsp,
                             UCHAR ucVector)
{
```

```

__INTERRUPT_OBJECT*    lpIntObject  = NULL;
__SYSTEM*              lpSystem     = NULL;

if((NULL == lpThis) || (NULL == lpEsp))
    return;

lpSystem = (__SYSTEM*)lpThis;
lpIntObject = lpSystem->lpInterruptVector[ucVector];
if(NULL == lpIntObject) //The current interrupt vector has not handler object.
{
    DefaultIntHandler(lpEsp,ucVector);
    return;
}
while(lpIntObject) //Travel the whole interrupt list of this vector.
{
    if(lpIntObject->InterruptHandler(lpEsp,
        lpIntObject->lpHandlerParam))
    {
        break;
    }
    lpIntObject = lpIntObject->lpNextInterruptObject;
}
return;
}

```

可见，这种中断处理的实现，支持中断共享，即支持多个设备使用同一条中断引脚。一个中断发生后，操作系统会轮询所有相同中断向量的中断对象（中断对象链表），并调用其处理函数，如果处理函数返回 **TRUE**，则说明该中断已经得到处理，于是直接返回，否则会一直遍历整个中断对象链表。

下面是中断对象的中断处理函数的定义：

```

BOOL    InterruptHandler(LPVOID lpParam);

```

lpParam 由驱动程序指定，该函数也在驱动程序中实现。需要注意的是，一旦该函数

被调用，该函数需要尽快的判断，自己是不是对应的中断处理程序（可以通过读取硬件寄存器判断），如果是，则进行进一步处理，最后必须返回 **TRUE**，否则，为了不影响系统的整体效率，需要中断处理程序尽快的返回 **FALSE**。

## 缺省中断处理函数

从 **DispatchInterrupt** 函数的处理过程看出，如果中断对象链表为空（即不包含任何中断对象），则该函数会调用一个 **DefaultIntHandler** 的函数。目前情况下，该函数实现下列功能：

- 1、打印出 “Unhandled interrupt or Exception!” 提示信息；
- 2、打印出相应的中断向量号；
- 3、把堆栈中前 12 个双字（**DWORD**）打印出来，这 12 个双字包含了通用寄存器信息、异常错误号信息等，通过这些信息，可以进行进一步的判断异常发生的原因；
- 4、如果是异常，则进入一个死循环；
- 5、否则，直接返回。

下面是该函数的实现代码：

```
static VOID DefaultIntHandler(LPVOID lpEsp, UCHAR ucVector)
{
    BYTE          strBuffer[16] = {0};
    DWORD         dwTmp          = 0L;
    DWORD         dwLoop         = 0L;
    DWORD*        lpdwEsp        = NULL;

    PrintLine("  Unhandled interrupt or Exception!"); //Print out this message.
    PrintLine("  Interrupt Vector:");

    dwTmp    = ucVector;
    lpdwEsp = (DWORD*)lpEsp;
    strBuffer[0] = ' ';
    strBuffer[1] = ' ';
    strBuffer[2] = ' ';
    strBuffer[3] = ' ';
```

```
Hex2Str(dwTmp,&strBuffer[4]);

PrintLine(strBuffer); //Print out the interrupt or exception's vector.
PrintLine(" Context:"); //Print out system context information.
for(dwLoop = 0;dwLoop < 12;dwLoop ++){
    dwTmp = *lpdwEsp;
    Hex2Str(dwTmp,&strBuffer[4]);
    PrintLine(strBuffer);
    lpdwEsp ++;
}

#define IS_EXCEPTION(vector) ((vector) <= 0x20)
if(IS_EXCEPTION(ucVector))
    DEAD_LOOP();

return;
}
```

## 对外服务接口

设备驱动程序（或其它实体）可以通过 `ConnectInterrupt` 函数连接一个中断，该函数是 `System` 对象的一个服务接口，声明如下：

```

__COMMON_OBJECT*      (*ConnectInterrupt)(__COMMON_OBJECT*
lpThis,

__INTERRUPT_HANDLER InterruptHandler,
LPVOID                lpHandlerParam,
UCHAR                  ucVector,
UCHAR                  ucReserved1,
UCHAR                  ucReserved2,
UCHAR                  ucInterruptMode,
BOOL                   blfShared,

```

```
DWORD dwCPUMask);
```

目前版本的实现中，驱动程序只需要给出 InterruptHandler、lpHandlerParam、ucVector 即可，其它参数都是为了将来便于扩展而保留的，目前版本下，只需要设置为 NULL 即可。该函数完成下列功能：

- 1、调用 CreateObject（ObjectManager 提供的服务接口）函数，创建一个中断对象；
- 2、根据该函数的参数，初始化中断对象；
- 3、把中断对象插入到特定的链表中(使用 ucVector 为索引，简缩 lpInterruptVector 数组)。

一旦中断连接成功，后续如果发生对应的中断，相应的处理程序就会被调用。需要注意的是，该函数返回创建的中断对象的指针，建议设备驱动程序保存这个指针，以便后续使用。

与 ConnectInterrupt 函数相反，DisconnectInterrupt 断开连接的中断，该函数原型如下：

```
VOID (*DisconnectInterrupt)(__COMMON_OBJECT* lpThis,
                             __COMMON_OBJECT* lpIntObj);
```

其中，第一个参数是 System 本身指针，第二个参数则是 ConnectInterrupt 函数返回的中断对象指针。该函数完成下列操作：

- 1、从对应的中断对象链表中，把 lpIntObj 删除；
- 2、销毁相应的中断对象。

一般情况下，在设备驱动程序被卸载，或者修改了中断向量，需要重新连接时，调用该函数。

## 几个注意事项

根据 Hello China 目前版本的实现，其中断处理模块具有下列特点：

- 1、支持中断共享，多个设备可以共享同一条中断引脚；
- 2、可移植，充分考虑了向量式中断模型和链表式中断模型，兼容两者，可以很方便的相互移植；
- 3、不支持中断嵌套。即一个中断发生后，如果在当前中断处理过程中，另外有其它的中断发生，后发生的中断不会马上被响应，而是直到当前中断处理完毕，才会响应后续中断。后续版本中，可实现中断嵌套；
- 4、目前的版本中，中断处理程序没有使用单独的堆栈，而是使用中断发生时，正在运

行的核心线程的堆栈。

因此，在设备驱动程序的编写过程中，需要充分考虑这些特点，建议驱动程序的中断处理部分，遵循下列原则：

- 1、 中断处理程序尽可能短；
- 2、 中断处理程序中，不能调用可能引起阻塞的系统调用，比如 WaitForThisObject 等；
- 3、 中断处理程序应首先判断发生的中断，是不是自己的（通过读取硬件寄存器可以获得），如果不是自己的，需要尽快返回 FALSE；
- 4、 在中断处理过程中，建议不要显式的打开中断，即显式的插入“sti”等指令，或调用包含“sti”指令的函数。

## Power PC 的异常处理机制

### PPC 异常处理机制概述

Power PC CPU 实现了完善的异常机制。在 PPC 中，对于 IA32 CPU 构架中的中断和自陷（trap），也称为异常，因此，异常这个称呼，在 PPC 中，泛指一切可打断当前程序处理的外部或内部事件。

与 IA32 构架类似，PPC 也定义了一个异常处理程序入口表，这个表放在内存中的固定位置（比如，放在物理内存 0x000000000 开始处，或者另外一个固定的位置，根据一个内部控制寄存器的标志位确定），而且对于每个可能发生的异常，都在异常处理程序描述表中对应一个特定的位置，该位置放置了对应的异常的处理程序。这样一旦对应的异常发生，CPU 就打断当前正在执行的程序，跳转到异常处理程序，来完成异常的处理。

与 IA32 不同的是，在 PPC 中，对于计算机上下文的保护，是放在两个特定的寄存器（SRR0/SRR1，Save-Restore Register）里面的，而在 IA32 构架中，对机器上下文的保存（比如引起异常或被打断的指令、机器状态字等），是放在堆栈中的。

下面的表格，给出了 Power PC 定义的异常，以及每种异常在异常处理程序表中的偏移：

异常类型	处理程序 偏移	产生原因
系统重启(System Reset)	00100	不同的型号的 PPC，对该异常有不同的触发实现

机 器 检 查 （ Machine Check）	00200	一般情况下，由总线奇偶错误或者访问不物理内存引起。
DSI（数据访问异常）	00300	非法数据访问引起，比如，访问权限不匹
ISI（指令访问异常）	00400	非法指令访问引起，比如，访问权限不匹
外 部 中 断 （ External Interrupt）	00500	外部设备引起，所有的外设，都使用同一输入引脚。
对齐（Alignment）	00600	访问的数据地址，跟数据大小不能对其的引发。
程序异常（Program）	00700	应用程序异常，一般由浮点部件、越权非法指令等引起。
浮 点 不 可 用 （Floating-point unavailable）	00800	执行浮点运算指令，但浮点部件不存在的会引发该异常。
定时器（Decrementer）	00900	CPU 内部定时器到时引发。
Reserved	00A00	保留。
Reserved	00B00	保留。
系统调用（System Call）	00C00	系统调用指令（sc）引发。
跟踪（Trace）	00D00	用于调试使用，跟踪每条指令执行情况。
浮 点 处 理 部 件 （Floating-point assist）	00E00	用于完成软件浮点处理模拟程序。
Reserved	00E10-00FFFF	保留。
Reserved	01000-02FFFF	保留。

表 8-1 Power PC 定义的异常

## Power PC 异常的分类

在 IA32 CPU 构架中，对系统中发生的异常分为异常、中断和自陷三类，其中异常是同步事件，即是由指令执行过程中，产生的不正常事件引起，自陷则是由应用程序通过特殊的指令（比如，int）引发，而中断则是由外部设备引发的一种异步事件。在 IA32 CPU 中，对于中断，又进一步分为可屏蔽中断和不可屏蔽中断。在 PPC CPU 的异常机制中，对系统中可能产生的异常，也按照类似方法，分成了四类：

- 1、**异步不可屏蔽异常**，这类异常由系统硬件引起，比如总线错误、机器内部状态错误等，在 PPC 中，系统复位（System Reset）和机器检查（Machine Check）两种异常，属于异步不可屏蔽异常。这类异常与 IA32 构架异常机制中的“不可屏蔽中断”相对应；
- 2、**异步可屏蔽异常**，这类异常由计算机的外设（外部硬件）引发，与 IA32 CPU 一样，这类异常是可屏蔽的，通过设置机器状态字（MSR）中的一个特定比特，可以屏蔽这些外部设备引发的异常。这类异常与 IA32 构架 CPU 的异常处理机制中，“可屏蔽外部中断”对应；
- 3、**同步精确异常（Asynchronous Precise）**，这类异常由指令执行过程中产生的错误引发，与 IA32 构架中的异常类似。所谓精确（Precise），指的是一旦 CPU 确定了某一条指令在执行过程中产生了异常，马上停止下来（同时完成一个上下文同步），然后直接调用异常处理程序。这类异常的特点是，可以准确的判断，异常到底是由哪条指令引起；
- 4、**同步非精确异常（Asynchronous imprecise）**，这类异常也是指令执行过程中产生错误引发的，与同步精确异常不同的是，这类异常发生后，CPU 有可能不会马上相应（调用异常处理程序），而有可能继续执行，在完成一些指令的执行后，再对发生的异常作出相应。这样就有可能无法确定到底是哪条指令引发了该异常，所以叫做非精确异常。当然，如果异常发生后，CPU 保存足够的信息，还是能够确定异常是由哪条指令引发的。在 PPC 的异常机制中，目前只有浮点使能（Floating-point enable）异常属于这类异常。

不论哪类异常发生，CPU 的处理过程都是类似的，所不同的是，对于不同的异常，CPU 保存的机器状态可能会不同。比如，对于同步异常（指令执行过程中的不正常事件引发的异常），CPU 可能会保存引发异常的指令，这样在异常处理程序中，就有机会对引发该指令异常的原因进行修复，从而使得 CPU 可以恢复运行，最典型的情况，就是缺页异常。而对于异步异常（外部事件引发的异常），CPU 则保存下一条要执行的指令（若当

前指令是一个分叉指令，即跳转指令，则 CPU 保存要跳转到的指令位置）。

## 异常的处理和返回

当异常发生的时候，PPC 完成下列动作：

- 1、根据异常的类型，把特定的指令指针，保存在 SRR0 寄存器中，比如，对于异步异常，保存下一条要执行的指令；
- 2、根据异常的类型，设置 SRR1 寄存器的特定比特。对于 SRR1 寄存器的其它比特，根据 MSR 寄存器（机器状态寄存器）进行设定；
- 3、合适的设置 MSR 寄存器，使得 CPU 工作在异常处理环境下。在这种环境下，地址翻译（MMU）被取消，直接使用物理地址进行寻址，这样使得异常处理程序一开始执行，就处于一种一致的环境中；
- 4、根据异常类型，计算得到对应的异常处理程序所在的地址，然后直接跳转到该地址，执行异常处理程序。这个时候，若异常处理程序需要启用地址翻译功能，则必须重新设置 MSR 的特定比特（IR 和 DR 比特）。

进入异常处理程序后，异常处理程序就可以根据特定寄存器中保存的值，来进一步确定异常发生的原因，并作出对应的处理。在异常处理结束后，执行一条 rfi（Return From Interrupt）指令，从异常处理程序中返回。这条指令可以完成上下文同步工作，并把 SRR1 寄存器的特定比特，重新拷贝回 MSR 寄存器，然后继续执行 SRR0 寄存器指定的指令。

## 定时器概述

定时功能是操作系统必备的一项重要服务，一般情况下，定时服务被用户线程（或任务）使用，来定时完成一项任务，在操作系统核心中，一些同步对象，比如事件对象（Event Object）、信号量对象（Semaphore）、互斥对象（Mutex）等，支持超时等待操作（即等待这些对象的时候，可以设定一个超时值），为实现超时等待操作，也需要使用操作系统核心提供的定时功能。

当前情况下，Hello China 提供定时器（Timer）对象，来实现定时服务，一个定时器对象也是一个核心对象，可以通过 Hello China 的对象框架来管理，并提供给用户统一的接口，来访问定时服务。在当前版本的 Hello China 中，提供了三个应用编程接口：

- 1、SetTimer，用来设置一个定时器；
- 2、CancelTimer，取消设置的定时器对象；
- 3、ResetTimer，复位定时器对象。

## SetTimer 调用

SetTimer 函数用来设定一个定时器，该函数原型如下：

```
__COMMON_OBJECT* SetTimer(
    DWORD          dwTimerID,
    DWORD          dwTimeSpan,
    DWORD          (*DirectHandler)(LPVOID),
    LPVOID         lpParam,
    DWORD          dwTimerFlags);
```

其中，dwTimerID 是定时器对象的标识符，用来标识定时器，这个参数的作用是为了让应用程序能够区分定时器对象。很多情况下，一个应用程序可能设定多个定时器，这样为了区分这多个定时器，需要为每个定时器设定一个不同的 ID。

dwTimeSpan 则是以毫秒 (ms) 为单位的超时间隔，超时间隔从设定开始计时，每次时钟中断，都会递减该超时间隔，一旦超时间隔递减为 0，定时器超时，会根据情况采取相应的动作。

DirectHandler 是一个函数指针，如果用户在设定定时器的时候，同时指定了该参数，则当定时器超时时，由操作系统核心调用该函数，lpParam 则是该函数的参与，如果用户在设定定时器的时候，没有指定该参数，则当定时器超时的时侯，操作系统会给设定定时器的线程（当前线程）发送一个定时器超时消息。

dwTimerFlags 可以指定设置什么样的定时器。当前版本中，Hello China 支持两种类型的定时器：一次定时器和永久定时器。一次定时器是只有一次超时处理的定时器，一旦定时器超时，则超时处理之后，该定时器对象将被删除，而永久定时器则是一个反复循环的定时器，一旦定时器超时，引发超时处理（发送消息或调用回调函数），超时处理完成之后，操作系统继续保留该定时器，并重新设置超时值，除非用户调用 CancelTimer 取消该定时器，否则该定时器会一直存在。如果 dwTimerFlags 设置了 TIMER\_FLAGS\_ONCE，则设置一个一次定时器，如果 dwTimerFlags 设置为 TIMER\_FLAGS\_ALWAYS，则设定一个永久定时器。

需要说明的是，定时器超时处理，有两种方式，一种方式是给调用 SetTimer 设定定时器的线程，发送一个消息，另外一个处理方式是，调用 SetTimer 设定的一个超时函数。这两种方式是互斥的，即如果用户在调用 SetTimer 的时候，设定了一个超时回调函数 (DirectHandler)，则当定时器超时时，只会调用该函数（以 lpParam 为参数），而不会再

给用户线程发送一个消息，相反，如果用户调用 `SetTimer` 的时候，指定 `DirectHandler` 为 `NULL`，则超时时，直接给用户线程发送一个消息。下面是 `Hello China` 目前定义的消息格式：

```
typedef struct __KERNEL_THREAD_MESSAGE{  
    WORD            wCommand;  
    WORD            wParam;  
    DWORD           dwParam;  
};
```

在超时处理过程中，操作系统核心会为用户线程发送一个消息，消息内容中的 `wCommand` 字段，设置为 `KERNEL_MESSAGE_TIMER`，`dwParam` 字段，设置为定时器标识（即 `SetTimer` 调用中的 `dwTimerID` 参数）。

## CancelTimer 调用

`CancelTimer` 用来取消已经设定的定时器对象。该函数原型如下：

```
VOID CancelTimer(__COMMON_OBJECT* lpTimerObject);
```

其中，`lpTimerObject` 是调用 `SetTimer` 函数返回的对象指针，该指针指向了创建的定时器对象。一般情况下，该函数是用来取消永久定时器的，该函数调用后，线程设定的永久定时器会被删除。

对于一次定时器对象，该函数需要在定时器尚未超时前调用，用来取消已经设定的定时器对象，倘若定时器已经超时，再调用该函数，可能会引发异常，或者造成系统紊乱，但有的时候，用户线程可能不知道定时器已经超时（比如，定时器已经超时，并且给设定定时器的线程发送了一个消息，但该消息还没有来得及被处理，这时候设定定时器的线程就可能不知道定时器已经超时），这种情况下，建议用户不要调用该函数，而让该定时器超时，然后系统会自动删除定时器对象。



```

DWORD
(*DirectHandler)(LPVOID),
LPVOID lpParam,
DWORD
dwTimerFlags);
VOID
(*CancelTimer)(__COMMON_OBJECT*lpSystem,
__COMMON_OBJECT*lpTimerObject);
VOID (*ResetTimer)(__COMMON_OBJECT*
lpSystem,
__COMMON_OBJECT*
lpTimerObject);
... ..
END_DEFINE_OBJECT()
```

可以看出，lpTimerQueue 用来维护定时器对象，SetTimer、CancelTimer 和 ResetTimer 分别对应前面介绍的三个系统调用（在转换为系统调用的时候，对前两个参数 lpSystem 和 lpKernelThread，进行了省略，因为系统中只有一个 System 全局对象，所以缺省情况下，取系统中这个全局对象为第一个参数，第二个参数则是调用该函数的当前线程）。

- SetTimer 调用可以用来设定一个定时器对象，该函数操作过程如下：
- 1、调用 CreateObject（ObjectManager 对象提供的接口）函数，创建一个定时器对象；
  - 2、初始化该定时器对象（根据用户传递过来的参数）；
  - 3、把定时器对象插入定时器对象维护队列；
  - 4、重新设定调度时刻。

这里有两个问题，需要细致说明一下，第一个问题是，如何确定该定时器对象的超时刻问题。在当前版本的实现中，定时器的超时判断，是在时钟中断处理程序里进行的，系统维护一个全局变量 dwClockTickCount，用来记录时钟中断数量，即每发生一次时钟中断，该变量递增一，同样地，系统也维护了另外一个变量，dwNextTimerTick，用来表示下一个定时器超时的时钟中断个数，这样每次时钟中断的时候，中断处理程序就会比较这两个变量，如果这两个变量相同，则说明在这个时钟中断中，有至少一个定时器对象超时了，因此需要做进一步处理。对于 dwNextTimerTick 变量的设置，是这

样进行的（在 SetTimer 函数调用中）：

```
... ..
dwPriority = dwTimeSpan / SYSTEM_TIME_SLICE;
dwPriority += dwClockTickCounter;
if(dwNextTimerTick > dwPriority)
    dwNextTimerTick = dwPriority;
... ..
```

其中，dwPriority 是一个临时变量，SYSTEM\_TIME\_SLICE 是系统定义的时间片，即每个时钟中断之间的时间间隔（以 ms 为单位，当前定义为 10），上述处理中，首先把 dwTimeSpan（定时器设定的等待时间，ms 为单位）换算成时钟中断个数（tick 个数），然后与当前 tick 计数(dwClockTickCounter)相加，这样就得到了当前定时器超时时的 tick。所有这些完成之后，再跟 dwNextTimerTick 比较，如果小于 dwNextTimerTick，则设置当前 dwNextTimerTick 为 dwPriority，否则，保留 dwNextTimerTick 不变。dwPriority 大于 dwNextTimerTick，说明先前已经有定时器被设置，而且设置的定时器超时时间，比当前设置的要提前。

第二个问题是，定时器维护队列（lpTimerQueue）是一个优先队列，在插入该队列的时候，可以提供一个优先数值，以让队列确定插入对象应该在的位置，优先级越高，对象在优先队列中的位置越靠前。对于定时器对象，理想的处理方式是，离超时时刻越近，应该越提前，这样但这样跟优先队列的优先级确定方式刚好相反（优先级越大，越靠前），因此，为了把最先超时的定时器对象插入优先队列的最前面，采用下列方式：

```
dwPriority = MAX_DWORD_VALUE - dwPriority;
lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
                                (__COMMON_OBJECT*)lpTimerObject,
                                dwPriority);
```

其中，lpTimerObject 是 SetTimer 创建的定时器对象，而 MAX\_DWORD\_VALUE 则是一个最大的类型为 DWORD 的常数，在 32 位硬件平台上，定义为 0xFFFFFFFF，这样变换之后，就可以实现上述目的（即把最先超时的对象，插入优先队列的最前端）。

## 定时器超时处理

当前情况下，定时器超时处理是在时钟中断中完成的。每次时钟中断发生后，中断处理程序被调用，在中断处理程序中，会比较 `dwClockTickCounter` 和 `dwNextTimerTick` 两个变量，如果这两个变量相同，则说明当前至少有一个定时器对象超时，这时候，时钟中断处理程序会进行如下操作（是一个循环操作）：

- 1、从定时器对象队列 (`lpTimerQueue`) 中，提取第一个定时器对象（定时器对象已经按照超时先后进行排序，排在最前面的定时器对象，是最先超时的定时器）；
- 2、获取定时器对象的超时时刻（即定时器对象在优先队列中的优先级，参考 `SetTimer` 函数的处理），然后跟当前 `dwNextTimerTick` 比较，如果不等于 `dwNextTimerTick`，则跳出当前循环，如果等于 `dwNextTimerTick`，则说明该定时器对象超时，于是执行下列操作：
  - a) 判断回调函数 (`DirectHandler`) 是否为空，如果不为空，则调用该回调函数，以 `lpHandlerParam` 为参数)；
  - b) 如果为空，则给设置该定时器对象的线程发送一个消息 (`KERNEL_MESSAGE_TIMER`)；
- 3、完成上述超时处理后，再判断当前定时器对象是一次性定时器，还是永久定时器，如果是一次定时器，则删除该定时器对象，如果是永久定时器，则重新更新该定时器的超时时间，并重新插入定时器对象队列（其插入优先级的确定方式与 `SetTimer` 操作雷同）；
- 4、然后再从定时器队列中提取下一个定时器对象，并转到步骤 1，进行循环处理。

在定时器对象的超时时刻（以 tick 计）不等于 `dwNextTimerTick` 的时候，上述循环结束，这时候当前定时器对象，将是目前所有的定时器对象中，最先超时的定时器对象，因此，需要根据该定时器的超时时刻，重新计算 `dwNextTimerTick` 的值，并更新 `dwNextTimerTick`，然后把该定时器对象重新插入定时器对象队列（因为该定时器对象虽然已被从定时器队列中提取出来，但由于没有超时，所以没有被处理，需要重新插入定时器对象队列，等待下一个超时时刻）。下面是定时器对象的处理代码：

```
if(System.dwClockTickCounter == System.dwNextTimerTick)    //Should schedule
timer.
{
    lpTimerQueue = System.lpTimerQueue;
    lpTimerObject = (__TIMER_OBJECT*)lpTimerQueue->GetHeaderElement(
        (__COMMON_OBJECT*)lpTimerQueue,
        &dwPriority);
```

```

if(NULL == lpTimerObject)
    goto __CONTINUE_1;
dwPriority = MAX_DWORD_VALUE - dwPriority;
while(dwPriority == System.dwNextTimerTick)
{
    if(NULL == lpTimerObject->DirectTimerHandler) //Send a message to the
kernel thread.
    {
        Msg.wCommand = KERNEL_MESSAGE_TIMER;
        Msg.dwParam  = lpTimerObject->dwTimerID;
        KernelThreadManager.SendMessage(
            (__COMMON_OBJECT*)lpTimerObject->lpKernelThread,
            &Msg);
    }
    else
    {
        lpTimerObject->DirectTimerHandler(
            lpTimerObject->lpHandlerParam);    //Call the associated handler.
    }

    switch(lpTimerObject->dwTimerFlags)
    {
        case TIMER_FLAGS_ONCE:                //Delete the timer object processed just
now.
            ObjectManager.DestroyObject(&ObjectManager,
                (__COMMON_OBJECT*)lpTimerObject);
            break;
        case TIMER_FLAGS_ALWAYS:              //Re-insert the timer object into timer
queue.
            dwPriority  = lpTimerObject->dwTimeSpan;
            dwPriority /= SYSTEM_TIME_SLICE;
            dwPriority += System.dwClockTickCounter;
            dwPriority  = MAX_DWORD_VALUE - dwPriority;

```

```

lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
                               (__COMMON_OBJECT*)lpTimerObject,
                               dwPriority);

    break;

default:
    break;
}

lpTimerObject = (__TIMER_OBJECT*)lpTimerQueue->GetHeaderElement(
    (__COMMON_OBJECT*)lpTimerQueue,
    &dwPriority);    //Check another timer object.
if(NULL == lpTimerObject)
    break;
dwPriority = MAX_DWORD_VALUE - dwPriority;
}

```

上面的处理中，是一个循环操作，从定时器队列中，提取第一个定时器对象，计算定时器对象的超时时刻（以 tick 计，超时时刻等于 MAX\_DWORD\_VALUE 减去定时器对象在优先队列中的优先级，请参考 SetTimer 函数的处理），并判断提取的定时器对象的超时时刻是否跟 dwNextTimerTick 相同，如果相同，说明该定时器已经超时，然后进一步处理，如果不相同，则说明超时的定时器对象已经处理完毕（只要优先队列中，队头对象没有超时，后边的对象肯定不会超时，因为定时器对象是按照超时先后顺序，插入优先队列的），这时候需要跳出循环。

下面的代码，更新了下一个超时时刻（dwNextTimerTick），并把当前定时器对象重新插入定时器对象队列：

```

if(NULL == lpTimerObject) //There is no timer object in queue.
{
    ENTER_CRITICAL_SECTION();
    System.dwNextTimerTick = 0L;
    LEAVE_CRITICAL_SECTION();
}
else
{

```

```

ENTER_CRITICAL_SECTION();
System.dwNextTimerTick = dwPriority;    //Update the next timer tick
counter.

LEAVE_CRITICAL_SECTION();
dwPriority = MAX_DWORD_VALUE - dwPriority;
lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
    (__COMMON_OBJECT*)lpTimerObject,
    dwPriority);
}
}

```

## 定时器取消处理

定时器取消处理比较简单：

- 1、首先从定时器对象队列中，把取消的处理器对象删除；
- 2、调用 DestroyObject 函数（ObjectManager 提供的接口），销毁 timer 对象；
- 3、更新 dwNextTimerTick 变量。

如下代码：

```

static VOID CancelTimer(__COMMON_OBJECT* lpThis,__COMMON_OBJECT*
lpTimer)
{
    __SYSTEM*          lpSystem      = NULL;
    DWORD              dwPriority     = 0L;
    __TIMER_OBJECT*    lpTimerObject = NULL;

    if((NULL == lpThis) || (NULL == lpTimer))
        return;
}

```

```

    lpSystem = (__SYSTEM*)lpThis;
    lpSystem->lpTimerQueue->DeleteFromQueue((__COMMON_OBJECT*)lpSystem->lpT
imerQueue,
        lpTimer);
    ObjectManager.DestroyObject(&ObjectManager,lpTimer);

    lpTimerObject = (__TIMER_OBJECT*)
        lpSystem->lpTimerQueue->GetHeaderElement(
            (__COMMON_OBJECT*)lpSystem->lpTimerQueue,
            &dwPriority);
    if(NULL == lpTimerObject)    //There is not any timer object to be processed.
        return;

    //
    //The following code updates the tick counter that timer object should be processed.
    //
    dwPriority = MAX_DWORD_VALUE - dwPriority;
    if(dwPriority > lpSystem->dwNextTimerTick)
        lpSystem->dwNextTimerTick = dwPriority;
    dwPriority = MAX_DWORD_VALUE - dwPriority;
    lpSystem->lpTimerQueue->InsertIntoQueue(
        (__COMMON_OBJECT*)lpSystem->lpTimerQueue,
        (__COMMON_OBJECT*)lpTimerObject,
        dwPriority);    //Insert into timer object queue.

    return;
}

```

上面的处理中，首先从定时器对象队列中删除要取消的定时器对象，并删除该对象，然后尝试从定时器队列中提取第一个对象（最先超时的定时器对象），如果能成功，说明目前尚有定时器对象，于是根据获取的对象的超时时刻，更新 `dwNextTimerTick` 变量，如果提取失败（返回 `NULL`），说明目前定时器队列中，已经没有定时器对象，这时候，只需要返回即可。

## 定时器复位

对于定时器复位操作，相对简单，主要完成下列工作：

- 1、从定时器队列中，删除要复位的定时器对象；
- 2、重新计算定时器对象的超时时刻，并重新插入定时器队列；
- 3、更新 `dwNextTimerTick` 变量。

## 定时器注意事项

考虑到定时器的实现机制，以及目标系统的性能要求，在实际应用中，使用定时器服务的时候，下列规则需要遵循：

- 1、在使用定时器服务时，如果不是十分必要，建议不要使用回调通知机制（即设定一个回调函数）。因为定时器的回调函数，是在时钟中断处理程序中被调用的，如果有大量的回调函数存在，会大大降低系统性能。下面这个测试，是在一台 **Pentium IV 2.5G** 处理器上，做的测试结果：

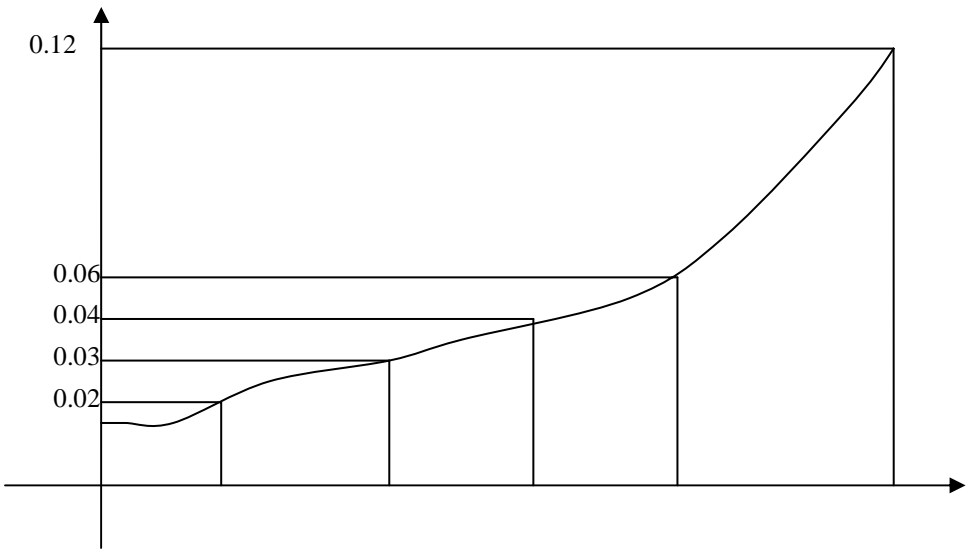


图 8-6 一个测试结果

其中，横坐标是一次定时器个数，其中，每个一次定时器的超时处理，都采用回调函数的方式，每个回调函数，所做的处理操作都一样，都是从一个优先队列中删除一个对象（该优先队列只有一个对象），因此，可认为回调函数的处理时间，都是一样的。纵坐标则是处理器的处理时间（通过记录处理器时钟周期个数来计算），可以看出，当一次定时器个数在 20 个的时候，处理时间达到了 0.06ms，如果上升到 30 个，则处理时间达到了 0.12ms。

但如果一次定时器采用发送消息的超时处理机制（即发送一个消息给设定定时器的线程），则如果处理时间为 0.12ms，需要设定 250 多个一次定时器。因此，可以看出，采用回调函数超时机制的定时器，其开销比采用消息通知机制的定时器大得多。

- 2、如果一定要采用回调函数机制来处理超时，建议回调函数的处理时间不能太长，一般情况下，处理时间不能大于 0.01ms；
- 3、对于采用消息通知作为超时处理机制的定时器，会引入误差。比如，假设设定的定时器，在 T1 时刻超时，则在 T1 时刻，设定定时器的线程会收到一个定时器超时消息，但真正处理该定时器超时消息的时间，可能会延迟到 T2，如下图所示：

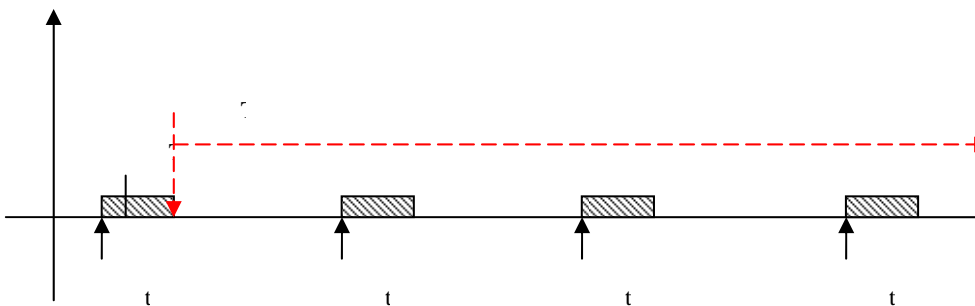


图 8-7 定时器消息的处理延时

图中， $t_N$  ( $t_1, t_2, t_3, \dots$ ) 是时钟中断发生时刻，其中， $t_N$  之间间隔均匀，在  $T_1$  时刻，系统给设定定时器的线程发送一个消息，然后继续执行时钟中断处理程序。中断处理程

序执行完毕之后，系统会选择线程就绪队列中，优先级最高的线程投入运行。这个时候，如果设定定时器的线程是优先级最高的线程，那么可能马上投入运行，这时候，定时器超时消息可能会被处理，因此，图中的 T2 时刻，是定时器超时消息得到处理的最早时刻。如果设定定时器对象的线程，优先级不是最高的，那么这个时候，就不会被调度，其它优先级更高的就绪线程会被调度，因此这个时候，定时器消息将一直存放在设定它的线程队列中，除非设定线程得到调度，否则定时器超时消息将一直不能被处理。因此，定时器超时消息的处理时间，可能会进一步延迟，一种最糟糕的情况就是，定时器超时消息可能永远得不到处理（在设定线程永远得不到调度的时候发生）。

但是对于采用回调函数作为超时处理机制的定时器，在 T1 时刻，回调函数就可以被调用，即超时消息马上被处理。因此，如果是一些时间要求十分严格的场合，建议使用回调机制处理超时。

## 第八章 中断和定时处理机制的实现

### ——The implementation of Interrupt and Timer

中断和异常处理以及定时机制，是操作系统必须实现的最重要的核心功能之一。在 Hello China V1.0 的实现中，实现了一个可以移植到不同 CPU 上的中断和异常处理机制模型，在本章中，我们将对这个模型进行详细介绍，另外，对 Power PC CPU 的中断和异常机制，也做了一个简单的介绍。

在本章的后面部分，我们对 Hello China 的定时机制及其应用，做了详细介绍。

## 8.1 中断和异常概述

中断和异常是系统在运行过程中，可能发生的外部或内部事件。一般情况下，中断是由外部设备引起的，比如，键盘设备，每当计算机系统的使用者按下一个键，或者放开一个键，键盘设备（严格来说，应该是控制键盘的芯片）就会生成一个中断，并通知 CPU。

而异常则一般是由软件造成的，泛指软件运行过程中，产生的不正常的事件。比如，最容易理解的是除法运算，如果出现了除数为 0 的情况，则会引发一个异常。另外一个很常见的异常，就是缺页异常。为了实现虚拟内存模型，操作系统可以把内存的一部分内容，暂时存储在外部存储设备上（比如，硬盘），从而腾出更多的内存空间为软件的运行服务。这样一旦一条指令访问了不在物理内存中的内存地址（比如，已经被暂时存储在硬盘上的数据），则会引发一个缺页异常。

一旦检测到中断或异常发生，CPU 就会中断当前的处理顺序，并根据中断或异常的类型，转移到相应的处理程序。需要注意的是，并不是中断或异常发生后，CPU 马上跳转到相应的处理程序，而是只有在指令的执行边界，CPU 才会检查是否有异常或中断发生，如果没有，CPU 继续执行下一条指令，即一旦有中断或异常发生，会引起 CPU 设置内部的相应寄存器位，然后继续执行当前指令（中断发生时，正在执行的指令，或者引发异常的指令），只有在当前指令执行完毕之后，相应的中断或异常，才有机会得到处理。

不同的 CPU 类型，其异常或中断的处理机制是不同的，比如，针对 Intel CPU，其对系统中可能产生的异常进行了编号，一旦异常发生，则 CPU 根据异常类型，找到对应的编号，然后再根据异常编号，查找一个叫做中断描述符表（IDT）的表格，找到对应的处理程序，然后跳转到具体的处理程序，对于中断，也是采取类似的方式，不过中断的编号（俗称中断向量号），是由硬件决定的。而 Power PC，则不论对异常，还是对中断，都调用同一个处理程序（所有的中断和异常，都调用同一个处理程序），然后处理程序再检测中断或异常的类型，进行进一步的分类处理，在此，我们称这种处理方式中断处理链方式。

在 Hello China 的设计中，充分考虑了这两种典型的中断和异常模型，采用了中断向量表跟中断处理链方式进行结合的实现方式，即首先有一个中断向量表，这样如果目标 CPU 是 Intel 系列的 CPU，则直接根据中断或异常的向量号，定位到一个中断向量表项，在每个中断向量表的表项中，又保存了一个中断对象链表，这样就可以实现链表方式的中断处理模型。因此，其可移植性较好。

在本部分中，我们以 Intel CPU（IA32 构架）为目标 CPU，详细介绍 Hello China 的中断处理机制，并介绍中断处理机制的服务提供接口。在本章的最后，简单介绍 Power PC CPU 的异常处理机制，以跟 IA32 的异常处理进行比较。

## 8.2 硬件相关部分处理

### 8.2.1 IA32 中断处理过程

所谓硬件相关部分处理，指的是针对不同的 CPU 平台，所采用的不同的中断处理方法。比如，针对 Intel IA32 构架的 CPU，需要建立 IDT（中断描述符表）表，并填写每个表项，针对 PPC 的 CPU，则需要初始化特定的寄存器等。在这一部分中，我们针对 IA32 构架的 CPU，进行描述，需要说明的是，硬件相关处理，仅仅是为了迎合不同的硬件的中断和异常模型，而引入的“第一层”处理，这部分处理比较简单，一般使用汇编语言实现，这部分的主要功能就是，完成硬件部分的处理，跟 Hello China 本身中断处理机制之间的连接。

IA32 CPU 采用中断描述符表（IDT）的方式，对中断和异常进行处理。IDT 是位于内存中的数据结构，该结构由 256 个中断描述符（或异常描述符）组成，每个中断描述符是一个 64 比特的数据结构，其每个比特的内容及含义，都是硬件严格定义的，在 CPU 内部，有一个很重要的寄存器：IDTR，即中断描述符表寄存器，这个寄存器保存了中断描述符表的起始物理地址（物理内存地址），一旦 CPU 检测到中断或异常发生，则进入如下的中断或异常处理流程：

- 6、把下一条将要执行的指令地址（EIP 寄存器）和代码段寄存器（CS），以及标志寄存器（EFLAGS）压入堆栈（当前线程堆栈）；
- 7、根据中断号或异常号，定位到具体的中断描述符（具体定位方法为：中断/异常向量号乘以 8，加上 IDTR 寄存器的值）；
- 8、中断描述符里面，存放了处理该中断或异常的处理程序的起始地址（以及所在的代码段），CPU 把起始地址读入到 EIP 寄存器（并更新代码段寄存器 CS），然后开始执行中断处理程序（实际上是一个跳转的过程）；
- 9、处理程序处理完毕后，使用 iret 指令，从中断处理程序中返回（该指令的含义，请参考本书“Hello China 线程的实现”一章）；

10、CPU 继续执行中断处理前的任务。

需要注意的是，上述处理过程，仅仅是一个简单的描述，实际上，根据不同的任务模型和地址模型，以及不同的 CPU 工作模式，中断处理方式各不相同，且十分复杂，但由于 Hello China 的设计，充分抽象了 CPU 的具体特征，最小化的利用了 CPU 的硬件特性，而把相关的特性放到软件中完成（便于移植到不同的 CPU），因此，上述简单的处理过程描述，在 Hello China 的实现中，已经足够。

这样可以看出，为了完成对 IA32 平台的中断处理模块的初始化，只需要完成下列两件事情：

- 3、 正确形成 IDT；
- 4、 把 IDT 的物理地址，填写到 IDTR 寄存器中。

其中，第二步非常简单，只需要一条指令就可完成任务：

```
lidt idt_addr
```

其中，idt\_addr 就是 IDT 的物理地址。接下来，我们对 IDT 的填写进行详细描述。

8.2.2 IDT 初始化

下面是 IA32 体系构架定义的中断描述符（Interrupt Descriptor）的结构：

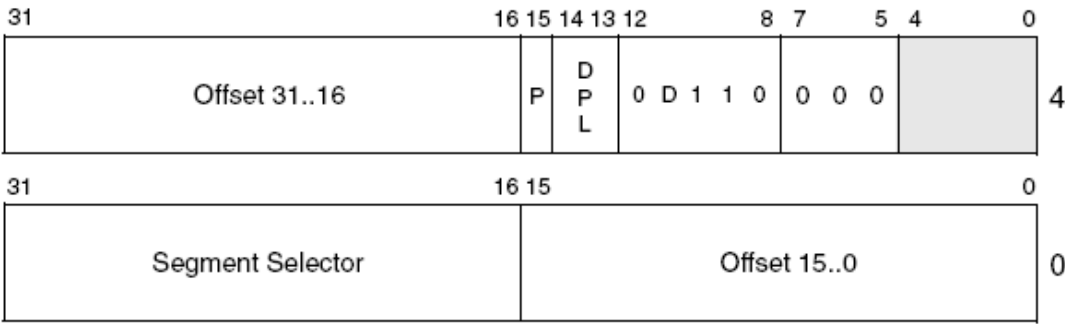


图 8-1 中断描述符的结构

其中，各个字段定义如下：

- **Segment Selector**: 段选择符，指明了中断处理程序所在的代码段；
- **Offset**: 中断处理程序在段内的偏移量；
- **DPL: Descriptor Privilege Level**, 即可以调用该中断描述符的当前处理级；
- **P: Present flags**, 如果段描述符在内存内, 则该标志置为 1, 如果当前描述符不存在, 则设置为 0;
- **D**: 门尺寸, 如果为 0, 则是一个 16bit 的段描述符, 否则是 32bit 描述符。

需要注意的是, **Offset** 字段长 32 比特, 被分成了两部分。

在 **Hello China** 的当前实现中, 所有中断和异常处理程序, 都位于代码断内, 即段描述符索引为 0x08 (详细信息参考 **Hello China** 的初始化部分), 而当前 **Hello China** 的实现中, 没有使用 **IA32** 提供的优先级保护, 因此, 所有涉及到 **DPL** 字段的地方, 都设置为 0, **D** 比特设置为 1 (32 位操作系统), **P** 比特也设置为 1, 因为当前版本的实现中, 所有中断描述符都是合法的 (存在于内存中的)。

剩下的唯一需要确定的字段, 就是 **Offset** 字段, 即中断处理程序在代码段中的位置。在当前版本的实现中, 方便起见, 为中断描述符中前 48 个中断描述符项, 定义了 48 个处理程序 (按照 **IA32** 的定义, 目前中断描述符表中, 前 32 个 (0—31) 为异常处理描述符, 后面 224 个 (32—255) 为用户定义的中断描述符, 因此, 在当前版本的实现中, 只定义了 48 个中断和异常处理程序, 因为一般的 PC 机上, 只有 16 个外部中断, 分别对应到中断描述符表中的第 32 到 47 个中断描述符), 目前情况下, 这些中断或异常处理程序, 完成的功能非常有限, 主要完成下列功能:

- 8、保存所有的通用寄存器;
- 9、把当前中断向量号或异常向量号, 压入堆栈;
- 10、把当前堆栈指针 (**ESP**) 压入堆栈;
- 11、调用同一个中断或异常处理程序 (这个处理程序, 采用 **C** 语言实现);
- 12、调用通用处理程序返回后, 恢复保存的通用寄存器;
- 13、如果是中断处理程序, 则通知中断控制器 (8259 芯片), 中断处理完毕;
- 14、从中断中返回。

下面是一个典型的中断处理程序:

```
np_int20:
    push eax
    cmp dword [gl_general_int_handler],0x00000000
    jz .ll_continue
```

```

push ebx                                ;;The following code saves the general
                                        ;;registers.

push ecx
push edx
push esi
push edi
push ebp
mov eax,esp
push eax
mov eax,0x20
push eax
call dword [gl_general_int_handler]
pop eax                                ;;Restore the general registers.
pop eax
mov esp,eax
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
.ll_continue:

mov al,0x20                            ;;Indicate the interrupt chip we have fin-
                                        ;;ished handle the interrupt.
                                        ;;;-)

out 0x20,al
out 0xa0,al
pop eax
iret

```

这是一段汇编代码，采用 NASM 进行编译。在这段程序的开始，首先压入 EAX 寄存器，然后判断 gl\_general\_int\_handler 是否为 0，如果为 0，则直接跳转到.ll\_continue（这是一个局部标号）处。

`gl_general_int_hander` 是在 `MINIKER.ASM` 文件中定义的一个 32 比特的全局变量，这个变量用来保存通用的中断和异常处理程序，这个通用的中断和异常处理程序，采用 C 语言编写，在操作系统初始化的时候，使用通用处理程序的地址，初始化 `gl_general_int_handler`。缺省情况下（未初始化的情况下），`gl_general_int_handler` 的值是 0，这样上述汇编代码就很容易理解了，首先判断，是否对 `gl_general_int_handler` 进行了初始化，如果没有，则直接跳转到 `.ll_continue` 处，直接解除中断，如果进行了初始化，即 `gl_general_int_handler` 的值不为 0，则进行正常的处理操作，包括保存通用寄存器，压入中断向量号（黑体标出的汇编语句），然后调用通用的中断和异常处理程序 `gl_general_int_handler`。从 `gl_general_int_handler` 返回后，再执行反向的恢复寄存器操作，然后解除中断（通过向中断控制芯片 8259 发送解除信号），并从中断中返回。需要注意的是，所有的中断处理程序（32—47），都是类似的，唯一不同的是，不同的中断处理程序，压入堆栈的中断向量号不同（代码中黑色部分）。

一个外部中断发生后，CPU 依次把 `EFLAGS`、`CS`、`EIP` 压入堆栈，然后根据中断向量号，从中断描述符表中，找到中断描述符，从中断描述符中，提取出中断处理程序的代码段选择符（`segment selector`）和中断处理程序，然后跳转到中断处理程序，也就是上述汇编语言编写的程序，继续执行。因此，在中断发生后，上述处理程序还未执行前，中断发生后的堆栈框架如下：

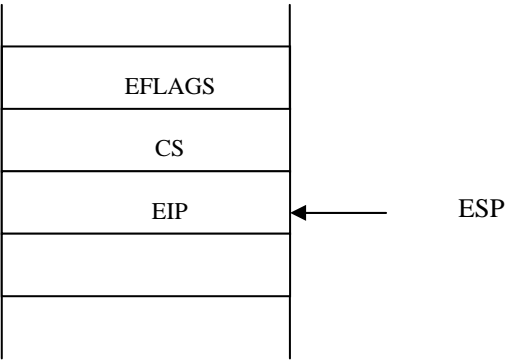


图 8-2 中断发生后的堆栈框架

一旦中断处理程序（上面描述的汇编语言程序）被执行，在实际调用 `gl_general_int_handler` 之前，形成如下的堆栈框架：

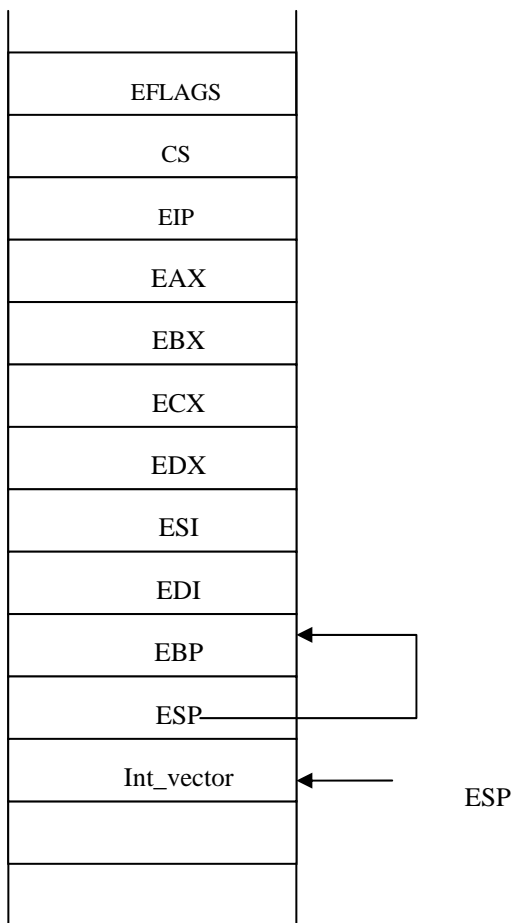


图 8-3 执行中断处理程序前建立的堆栈框架

其中，在上述堆栈框架中，保存的 ESP 的值，是压入 EBP 之后的 ESP 的值（图形中的折线示意位置），之所以保存 ESP 的值，是把 ESP 作为通用中断和异常处理程序的一个参数，来传递给通用中断和异常处理程序，这样通用的中断和异常处理程序，就可以访问堆栈框架了。下面是通用异常和中断处理程序的原型：

```
VOID GeneralIntHandler(DWORD dwVector,LPVOID lpEsp);
```

这样一切就明了了，汇编语言编写的中断处理程序，只是建立起一个堆栈框架（保存了通用寄存器的值），然后把 `esp` 寄存器的值和当前发生的中断的中断向量号，压入堆栈，作为 `GeneralIntHandler` 的参数，最后调用 `GeneralIntHandler (gl_general_int_handler)`。在 `GeneralIntHandler` 中，就可以通过 `dwVector` 和 `lpEsp` 来直接访问中断向量号和堆栈框架了。

对于异常的处理，与中断处理类似，唯一不同的是，对于异常的处理，在异常处理程序的最后，不用向中断控制器芯片发命令，来解除中断，下面是一个异常处理程序：

```
np_int0E:
    push eax
    cmp dword [gl_general_int_handler],0x00000000
    jz .ll_continue
    push ebx                                ;;The following code saves the general
                                           ;;registers.

    push ecx
    push edx
    push esi
    push edi
    push ebp
    mov eax,esp
    push eax
    mov eax,0x0E
    push eax
    call dword [gl_general_int_handler]
    pop eax                                ;;Restore the general registers.
    pop eax
    mov esp,eax
    pop ebp
    pop edi
    pop esi
```

```
    pop edx
    pop ecx
    pop ebx
.ll_continue:
    pop eax
    iret
```

需要进一步说明的是，对于异常的处理，IA32 CPU 会根据异常类型的不同，有选择的往堆栈中压入一个异常错误号，这样就导致了异常发生后的堆栈框架，与中断发生后的堆栈框架不一致（因为异常发生后，堆栈中比中断发生后多了一个错误号），因此需要特殊的处理。但在当前 **Hello China** 的实现中，没有考虑这种情况，因为一旦异常发生，按照当前版本的实现，只是打印出引发异常的上下文信息，然后停机（**HLT** 指令）。后续版本的实现中，需要充分考虑这种区别，按照现在的设计，充分考虑了这种特殊的情况，后续如果需要，这种特殊的处理，是很容易被支持的。

这样把各个中断和异常处理程序编写完成之后，剩下的任务就十分简单了，只需要把每个中断和异常的处理程序的地址（偏移），填写在相应的中断描述符的 **Offset** 字段即可。这项任务十分简单，在当前版本的实现中，是通过一个汇编语言例程（**np\_fill\_idt**）来实现的，在此不做赘述。

最后，我们把 **gl\_general\_int\_handler** 和 **GeneralIntHandler** 之间的关系说明一下。**gl\_general\_int\_handler** 是在 **MINIKER.ASM** 文件中声明的一个全局变量，并被初始化为 0，与其它全局变量不同的是，**gl\_general\_int\_handler** 被声明在了固定的位置，即 **MINIKER.BIN** 文件的末尾处，而当前版本下，**MINIKER.BIN** 文件（**MINIKER.ASM** 编译后形成的二进制文件）大小固定为 48K，因此，**gl\_general\_int\_handler** 相对于 **MINIKER.BIN** 文件的偏移，就是 48K - 4 位置处。

而 **GeneralIntHandler** 则是一个 C 语言函数，被编译在 **MASTER.BIN** 文件中。在操作系统初始化的时候，**MINIKER.BIN** 被加载到内存地址 1M 开始的地方，而 **MASTER.BIN** 则被加载到物理内存 0x00110000 位置处（1M+64K），如下所示：

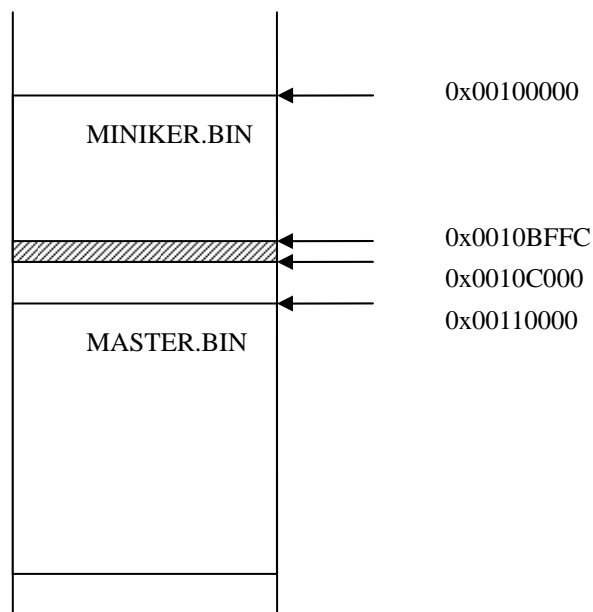


图 8-4 Hello China 相关模块在内存中的位置

其中，阴影部分是在 MINIKER.ASM 中，声明的 `gl_general_int_handler` 变量的位置。这样在 MASTER.BIN 初始化的时候，直接把 `GeneralIntHandler` 的值（其实是一个函数指针值）填写在 `gl_general_int_handler` 处，这样就完成了 `gl_general_int_handler` 的初始化。在 Hello China 的当前实现中，为了连接 MINIKER.BIN 和 MASTER.BIN 两个模块，大量的采用了这种方式。

## 8.3 硬件无关部分处理

### 8.3.1 系统对象和中断对象

当前版本的 Hello China，大量的采用了面向对象的思想进行设计，对于系统相关的一些功能和任务，比如定时处理、中断处理等，都封装到一个系统对象中，即 `System` 对象。该对象维护了所有中断处理相关的数据结构、函数等。

下面是 `System` 对象的定义：



```
lpKernelThread,

                                DWORD                dwTimerID,
                                DWORD                dwTimeSpan,
                                __DIRECT_TIMER_HANDLER
DirectTimerHandler,

                                LPVOID
lpHandlerParam,

                                DWORD
dwTimerFlags

                                );
VOID                (*CancelTimer)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpTimer);

END_DEFINE_OBJECT()
```

其中，用黑色字体标出来的相关代码，是中断处理相关的定义，包括一个中断对象数组（lpInterruptVector），三个完成中断调度以及中断服务相关的函数。

在这里，我们先介绍 lpInterruptVector。这是一个中断对象数组，所谓中断对象，是Hello China 定义，用来描述一个中断处理函数的对象，该对象定义如下：

```
BEGIN_DEFINE_OBJECT(__INTERRUPT_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    __INTERRUPT_OBJECT*        lpPrevInterruptObject;
    __INTERRUPT_OBJECT*        lpNextInterruptObject;
    UCHAR                      ucVector;
    BOOL                        (*InterruptHandler)(LPVOID lpParam, LPVOID
lpEsp);
    LPVOID                      lpHandlerParam;
END_DEFINE_OBJECT()
```

lpPrevInterruptObject和lpNextInterruptObject是用来把中断对象连接到双向链表中的指针，该对象从\_\_COMMON\_OBJECT对象继承，因此，遵循Hello China的对象语义，可以通过CreateObject方法（ObjectManager对象提供）创建。ucVector是中断向量，InterruptHandler则是真正的中断处理函数，lpHandlerParam则是中断处理函数的参数。

一般情况下,一个设备驱动程序(或者其它实体)如果想连接一个中断,则调用System对象提供的ConnectInterrupt函数,在这个函数里,指定了ucVector变量、InterruptHandler变量和相关的参数,ConnectInterrupt于是调用CreateObject函数,创建一个中断对象,然后把相关的变量初始化(包括ucVector、InterruptHandler等),并以ucVector变量为索引,定位到lpInterruptVector数组的一个特定元素(lpInterruptVector[ucVector]),这个元素是一个中断对象指针,指向一个中断对象链表,于是ConnectInterrupt就把新创建的对象,插入到这个链表中。

按照这种处理方式,系统中的所有中断对象,按照下列形式组织:

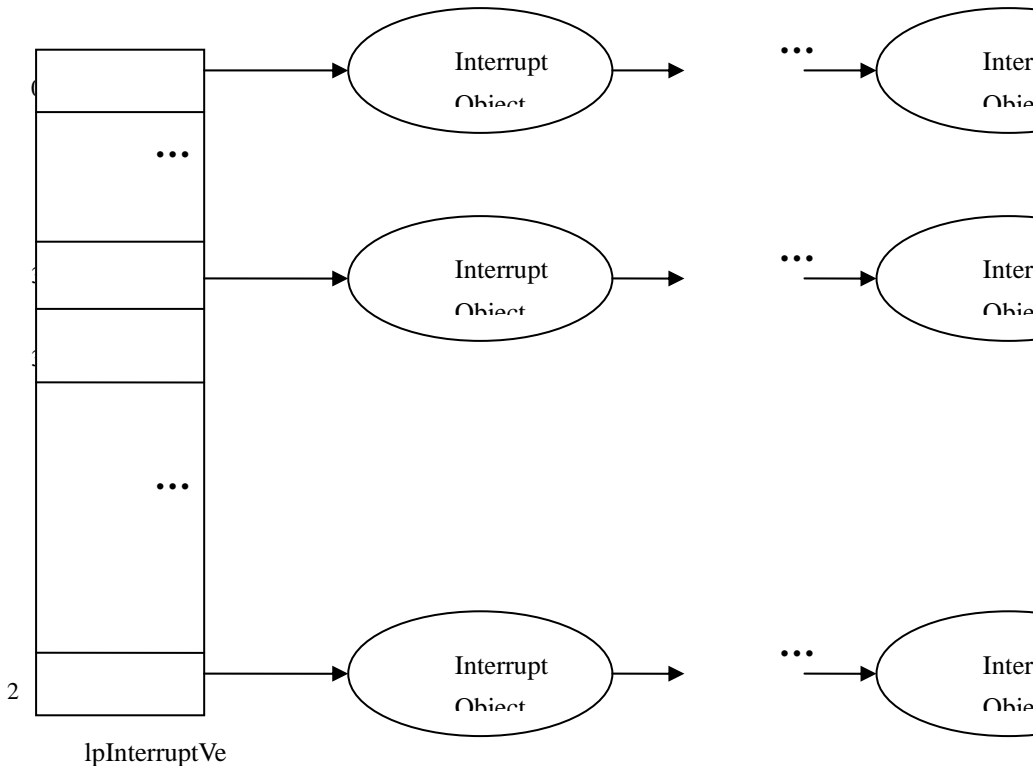


图 8-5 Hello China 中断对象的组织

其中, `lpInterruptVector`是一个数组(System对象的一个成员),其元素是中断对

象指针，系统中的所有中断对象，按照中断向量号（ucVector），被插入到相应的链表中。

### 8.3.2 中断调度过程

一旦中断或异常发生，CPU 首先根据中断或异常向量号，找到对应的描述符，从描述符中提取出中断或异常处理程序，然后把控制转移到中断或异常处理程序。这里的中断或异常处理程序，就是上面介绍的使用汇编语言编写的中断处理程序。

根据上面的描述，所有的异常和中断的处理，都会调用一个通用的中断异常处理程序 `gl_general_int_handler`，而目前情况下，该处理程序在 `MASTER.BIN` 中，使用 C 语言实现，即 `GeneralIntHandler` 函数，该函数的实现如下：

```
VOID GeneralIntHandler(DWORD dwVector, LPVOID lpEsp)
{
    UCHAR    ucVector = LOBYTE(LOWORD(dwVector));

    System.DispatchInterrupt((__COMMON_OBJECT*)&System,
        lpEsp,
        ucVector);
}
```

可见，该函数没有进行过多的处理，而是调用了 `System` 对象的 `DispatchInterrupt` 函数（见 `System` 对象的定义）。`DispatchInterrupt` 函数做如下处理：

- 4、根据中断向量号 `ucVector`，定位到特定的中断对象链表；
- 5、如果中断链表中没有任何中断对象，则调用一个缺省的中断或异常处理程序；
- 6、否则，依次调用该链表中，中断对象的中断处理函数，直到有一个中断处理函数返回 `TRUE`。

代码如下：

```
static VOID DispatchInterrupt(__COMMON_OBJECT* lpThis,
                                LPVOID          lpEsp,
                                UCHAR ucVector)
{
```

```

__INTERRUPT_OBJECT*    lpIntObject  = NULL;
__SYSTEM*              lpSystem     = NULL;

if((NULL == lpThis) || (NULL == lpEsp))
    return;

lpSystem = (__SYSTEM*)lpThis;
lpIntObject = lpSystem->lpInterruptVector[ucVector];
if(NULL == lpIntObject) //The current interrupt vector has not handler object.
{
    DefaultIntHandler(lpEsp,ucVector);
    return;
}
while(lpIntObject) //Travel the whole interrupt list of this vector.
{
    if(lpIntObject->InterruptHandler(lpEsp,
        lpIntObject->lpHandlerParam))
    {
        break;
    }
    lpIntObject = lpIntObject->lpNextInterruptObject;
}
return;
}

```

可见，这种中断处理的实现，支持中断共享，即支持多个设备使用同一条中断引脚。一个中断发生后，操作系统会轮询所有相同中断向量的中断对象（中断对象链表），并调用其处理函数，如果处理函数返回 **TRUE**，则说明该中断已经得到处理，于是直接返回，否则会一直遍历整个中断对象链表。

下面是中断对象的中断处理函数的定义：

```

BOOL    InterruptHandler(LPVOID lpParam);

```

lpParam 由驱动程序指定，该函数也在驱动程序中实现。需要注意的是，一旦该函数

被调用，该函数需要尽快的判断，自己是不是对应的中断处理程序（可以通过读取硬件寄存器判断），如果是，则进行进一步处理，最后必须返回 **TRUE**，否则，为了不影响系统的整体效率，需要中断处理程序尽快的返回 **FALSE**。

### 8.3.3 缺省中断处理函数

从 **DispatchInterrupt** 函数的处理过程看出，如果中断对象链表为空（即不包含任何中断对象），则该函数会调用一个 **DefaultIntHandler** 的函数。目前情况下，该函数实现下列功能：

- 6、打印出 “Unhandled interrupt or Exception!” 提示信息；
- 7、打印出相应的中断向量号；
- 8、把堆栈中前 12 个双字（**DWORD**）打印出来，这 12 个双字包含了通用寄存器信息、异常错误号信息等，通过这些信息，可以进行进一步的判断异常发生的原因；
- 9、如果是异常，则进入一个死循环；
- 10、否则，直接返回。

下面是该函数的实现代码：

```
static VOID DefaultIntHandler(LPVOID lpEsp, UCHAR ucVector)
{
    BYTE          strBuffer[16] = {0};
    DWORD         dwTmp          = 0L;
    DWORD         dwLoop        = 0L;
    DWORD*        lpdwEsp       = NULL;

    PrintLine("  Unhandled interrupt or Exception!"); //Print out this message.
    PrintLine("  Interrupt Vector:");

    dwTmp    = ucVector;
    lpdwEsp = (DWORD*)lpEsp;
    strBuffer[0] = ' ';
    strBuffer[1] = ' ';
    strBuffer[2] = ' ';
    strBuffer[3] = ' ';
```

```
Hex2Str(dwTmp,&strBuffer[4]);

PrintLine(strBuffer); //Print out the interrupt or exception's vector.
PrintLine(" Context:"); //Print out system context information.
for(dwLoop = 0;dwLoop < 12;dwLoop ++){
    dwTmp = *lpdwEsp;
    Hex2Str(dwTmp,&strBuffer[4]);
    PrintLine(strBuffer);
    lpdwEsp ++;
}

#define IS_EXCEPTION(vector) ((vector) <= 0x20)
if(IS_EXCEPTION(ucVector))
    DEAD_LOOP();

return;
}
```

## 8.4 对外服务接口

设备驱动程序（或其它实体）可以通过 `ConnectInterrupt` 函数连接一个中断，该函数是 `System` 对象的一个服务接口，声明如下：

```

__COMMON_OBJECT*          (*ConnectInterrupt)(__COMMON_OBJECT*
lpThis,

__INTERRUPT_HANDLER InterruptHandler,
LPVOID                    lpHandlerParam,
UCHAR                     ucVector,
UCHAR                     ucReserved1,
UCHAR                     ucReserved2,
UCHAR                     ucInterruptMode,
BOOL                      blfShared,

```

```

        DWORD          dwCPUMask
    );

```

目前版本的实现中，驱动程序只需要给出 `InterruptHandler`、`lpHandlerParam`、`ucVector` 即可，其它参数都是为了将来便于扩展而保留的，目前版本下，只需要设置为 `NULL` 即可。该函数完成下列功能：

- 4、调用 `CreateObject` (`ObjectManager` 提供的服务接口) 函数，创建一个中断对象；
- 5、根据该函数的参数，初始化中断对象；
- 6、把中断对象插入到特定的链表中(使用 `ucVector` 为索引，简缩 `lpInterruptVector` 数组)。

一旦中断连接成功，后续如果发生对应的中断，相应的处理程序就会被调用。需要注意的是，该函数返回创建的中断对象的指针，建议设备驱动程序保存这个指针，以便后续使用。

与 `ConnectInterrupt` 函数相反，`DisconnectInterrupt` 断开连接的中断，该函数原型如下：

```

VOID          (*DisconnectInterrupt)(__COMMON_OBJECT* lpThis,
                                     __COMMON_OBJECT* lpIntObj);

```

其中，第一个参数是 `System` 本身指针，第二个参数则是 `ConnectInterrupt` 函数返回的中断对象指针。该函数完成下列操作：

- 3、从对应的中断对象链表中，把 `lpIntObj` 删除；
- 4、销毁相应的中断对象。

一般情况下，在设备驱动程序被卸载，或者修改了中断向量，需要重新连接时，调用该函数。

## 8.5 几个注意事项

根据 `Hello China` 目前版本的实现，其中断处理模块具有下列特点：

- 5、支持中断共享，多个设备可以共享同一条中断引脚；
- 6、可移植，充分考虑了向量式中断模型和链表式中断模型，兼容两者，可以很方便的相互移植；
- 7、不支持中断嵌套。即一个中断发生后，如果在当前中断处理过程中，另外有其它的中断发生，后发生的中断不会马上被响应，而是直到当前中断处理完毕，才会响应后续中断。后续版本中，可实现中断嵌套；
- 8、目前的版本中，中断处理程序没有使用单独的堆栈，而是使用中断发生时，正在运

行的核心线程的堆栈。

因此，在设备驱动程序的编写过程中，需要充分考虑这些特点，建议驱动程序的中断处理部分，遵循下列原则：

- 5、中断处理程序尽可能短；
- 6、中断处理程序中，不能调用可能引起阻塞的系统调用，比如 WaitForThisObject 等；
- 7、中断处理程序应首先判断发生的中断，是不是自己的（通过读取硬件寄存器可以获得），如果不是自己的，需要尽快返回 FALSE；
- 8、在中断处理过程中，建议不要显式的打开中断，即显式的插入“sti”等指令，或调用包含“sti”指令的函数。

## 8.6 Power PC 的异常处理机制

### 8.6.1 PPC 异常处理机制概述

Power PC CPU 实现了完善的异常机制。在 PPC 中，对于 IA32 CPU 构架中的中断和自陷（trap），也称为异常，因此，异常这个称呼，在 PPC 中，泛指一切可打断当前程序处理的外部或内部事件。

与 IA32 构架类似，PPC 也定义了一个异常处理程序入口表，这个表放在内存中的固定位置（比如，放在物理内存 0x000000000 开始处，或者另外一个固定的位置，根据一个内部控制寄存器的标志位确定），而且对于每个可能发生的异常，都在异常处理程序描述表中对应一个特定的位置，该位置放置了对应的异常的处理程序。这样一旦对应的异常发生，CPU 就打断当前正在执行的程序，跳转到异常处理程序，来完成异常的处理。

与 IA32 不同的是，在 PPC 中，对于计算机上下文的保护，是放在两个特定的寄存器（SRR0/SRR1，Save-Restore Register）里面的，而在 IA32 构架中，对机器上下文的保存（比如引起异常或被打断的指令、机器状态字等），是放在堆栈中的。

下面的表格，给出了 Power PC 定义的异常，以及每种异常在异常处理程序表中的偏移：

异常类型	处理程序 偏移	产生原因
系统重启(System Reset)	00100	不同的型号的 PPC，对该异常有不同的触发实现

机 器 检 查 （ Machine Check）	00200	一般情况下，由总线奇偶错误或者访问不物理内存引起。
DSI（数据访问异常）	00300	非法数据访问引起，比如，访问权限不匹
ISI（指令访问异常）	00400	非法指令访问引起，比如，访问权限不匹
外 部 中 断 （ External Interrupt）	00500	外部设备引起，所有的外设，都使用同一输入引脚。
对齐（Aligment）	00600	访问的数据地址，跟数据大小不能对其的引发。
程序异常（Program）	00700	应用程序异常，一般由浮点部件、越权非法指令等引起。
浮 点 不 可 用 （Floating-point unavailable）	00800	执行浮点运算指令，但浮点部件不存在的会引发该异常。
定时器（Decrementer）	00900	CPU 内部定时器到时引发。
Reserved	00A00	保留。
Reserved	00B00	保留。
系统调用（System Call）	00C00	系统调用指令（sc）引发。
跟踪（Trace）	00D00	用于调试使用，跟踪每条指令执行情况。
浮 点 处 理 部 件 （Floating-point assist）	00E00	用于完成软件浮点处理模拟程序。
Reserved	00E10-00FF	保留。
Reserved	01000-02FFF	保留。

表 8-1 Power PC 定义的异常

## Power PC 异常的分类

在 IA32 CPU 构架中，对系统中发生的异常分为异常、中断和自陷三类，其中异常是同步事件，即是由指令执行过程中，产生的不正常事件引起，自陷则是由应用程序通过特殊的指令（比如，int）引发，而中断则是由外部设备引发的一种异步事件。在 IA32 CPU 中，对于中断，又进一步分为可屏蔽中断和不可屏蔽中断。在 PPC CPU 的异常机制中，对系统中可能产生的异常，也按照类似方法，分成了四类：

- 5、**异步不可屏蔽异常**，这类异常由系统硬件引起，比如总线错误、机器内部状态错误等，在 PPC 中，系统复位（System Reset）和机器检查（Machine Check）两种异常，属于异步不可屏蔽异常。这类异常与 IA32 构架异常机制中的“不可屏蔽中断”相对应；
- 6、**异步可屏蔽异常**，这类异常由计算机的外设（外部硬件）引发，与 IA32 CPU 一样，这类异常是可屏蔽的，通过设置机器状态字（MSR）中的一个特定比特，可以屏蔽这些外部设备引发的异常。这类异常与 IA32 构架 CPU 的异常处理机制中，“可屏蔽外部中断”对应；
- 7、**同步精确异常（Asynchronous Precise）**，这类异常由指令执行过程中产生的错误引发，与 IA32 构架中的异常类似。所谓精确（Precise），指的是一旦 CPU 确定了某一条指令在执行过程中产生了异常，马上停止下来（同时完成一个上下文同步），然后直接调用异常处理程序。这类异常的特点是，可以准确的判断，异常到底是由哪条指令引起；
- 8、**同步非精确异常（Asynchronous imprecise）**，这类异常也是指令执行过程中产生错误引发的，与同步精确异常不同的是，这类异常发生后，CPU 有可能不会马上相应（调用异常处理程序），而有可能继续执行，在完成一些指令的执行后，再对发生的异常作出相应。这样就有可能无法确定到底是哪条指令引发了该异常，所以叫做非精确异常。当然，如果异常发生后，CPU 保存足够的信息，还是能够确定异常是由哪条指令引发的。在 PPC 的异常机制中，目前只有浮点使能（Floating-point enable）异常属于这类异常。

不论哪类异常发生，CPU 的处理过程都是类似的，所不同的是，对于不同的异常，CPU 保存的机器状态可能会不同。比如，对于同步异常（指令执行过程中的不正常事件引发的异常），CPU 可能会保存引发异常的指令，这样在异常处理程序中，就有机会对引发该指令异常的原因进行修复，从而使得 CPU 可以恢复运行，最典型的情况，就是缺页异常。而对于异步异常（外部事件引发的异常），CPU 则保存下一条要执行的指令（若当

前指令是一个分叉指令，即跳转指令，则 CPU 保存要跳转到的指令位置）。

## 异常的处理和返回

当异常发生的时候，PPC 完成下列动作：

- 5、根据异常的类型，把特定的指令指针，保存在 SRR0 寄存器中，比如，对于异步异常，保存下一条要执行的指令；
- 6、根据异常的类型，设置 SRR1 寄存器的特定比特。对于 SRR1 寄存器的其它比特，根据 MSR 寄存器（机器状态寄存器）进行设定；
- 7、合适的设置 MSR 寄存器，使得 CPU 工作在异常处理环境下。在这种环境下，地址翻译（MMU）被取消，直接使用物理地址进行寻址，这样使得异常处理程序一开始执行，就处于一种一致的环境中；
- 8、根据异常类型，计算得到对应的异常处理程序所在的地址，然后直接跳转到该地址，执行异常处理程序。这个时候，若异常处理程序需要启用地址翻译功能，则必须重新设置 MSR 的特定比特（IR 和 DR 比特）。

进入异常处理程序后，异常处理程序就可以根据特定寄存器中保存的值，来进一步确定异常发生的原因，并作出对应的处理。在异常处理结束后，执行一条 rfi（Return From Interrupt）指令，从异常处理程序中返回。这条指令可以完成上下文同步工作，并把 SRR1 寄存器的特定比特，重新拷贝回 MSR 寄存器，然后继续执行 SRR0 寄存器指定的指令。

## 定时器概述

定时功能是操作系统必备的一项重要服务，一般情况下，定时服务被用户线程（或任务）使用，来定时完成一项任务，在操作系统核心中，一些同步对象，比如事件对象（Event Object）、信号量对象（Semaphore）、互斥对象（Mutex）等，支持超时等待操作（即等待这些对象的时候，可以设定一个超时值），为实现超时等待操作，也需要使用操作系统核心提供的定时功能。

当前情况下，Hello China 提供定时器（Timer）对象，来实现定时服务，一个定时器对象也是一个核心对象，可以通过 Hello China 的对象框架来管理，并提供给用户统一的接口，来访问定时服务。在当前版本的 Hello China 中，提供了三个应用编程接口：

- 4、SetTimer，用来设置一个定时器；
- 5、CancelTimer，取消设置的定时器对象；
- 6、ResetTimer，复位定时器对象。

## SetTimer 调用

SetTimer 函数用来设定一个定时器，该函数原型如下：

```
__COMMON_OBJECT* SetTimer(
    DWORD          dwTimerID,
    DWORD          dwTimeSpan,
    DWORD          (*DirectHandler)(LPVOID),
    LPVOID         lpParam,
    DWORD          dwTimerFlags);
```

其中，dwTimerID 是定时器对象的标识符，用来标识定时器，这个参数的作用是为了让应用程序能够区分定时器对象。很多情况下，一个应用程序可能设定多个定时器，这样为了区分这多个定时器，需要为每个定时器设定一个不同的 ID。

dwTimeSpan 则是以毫秒（ms）为单位的超时间隔，超时间隔从设定开始计时，每次时钟中断，都会递减该超时间隔，一旦超时间隔递减为 0，定时器超时，会根据情况采取相应的动作。

DirectHandler 是一个函数指针，如果用户在设定定时器的时候，同时指定了该参数，则当定时器超时，由操作系统核心调用该函数，lpParam 则是该函数的参与，如果用户在设定定时器的时候，没有指定该参数，则当定时器超时的时，操作系统会给设定定时器的线程（当前线程）发送一个定时器超时消息。

dwTimerFlags 可以指定设置什么样的定时器。当前版本中，Hello China 支持两种类型的定时器：一次定时器和永久定时器。一次定时器是只有一次超时处理的定时器，一旦定时器超时，则超时处理之后，该定时器对象将被删除，而永久定时器则是一个反复循环的定时器，一旦定时器超时，引发超时处理（发送消息或调用回调函数），超时处理完成之后，操作系统继续保留该定时器，并重新设置超时值，除非用户调用 CancelTimer 取消该定时器，否则该定时器会一直存在。如果 dwTimerFlags 设置了 TIMER\_FLAGS\_ONCE，则设置一个一次定时器，如果 dwTimerFlags 设置为 TIMER\_FLAGS\_ALWAYS，则设定一个永久定时器。

需要说明的是，定时器超时处理，有两种方式，一种方式是给调用 SetTimer 设定定时器的线程，发送一个消息，另外一个处理方式是，调用 SetTimer 设定的一个超时函数。这两种方式是互斥的，即如果用户在调用 SetTimer 的时候，设定了一个超时回调函数（DirectHandler），则当定时器超时，只会调用该函数（以 lpParam 为参数），而不会再

给用户线程发送一个消息，相反，如果用户调用 `SetTimer` 的时候，指定 `DirectHandler` 为 `NULL`，则超时时，直接给用户线程发送一个消息。下面是 `Hello China` 目前定义的消息格式：

```
typedef struct __KERNEL_THREAD_MESSAGE{  
    WORD          wCommand;  
    WORD          wParam;  
    DWORD         dwParam;  
};
```

在超时处理过程中，操作系统核心会为用户线程发送一个消息，消息内容中的 `wCommand` 字段，设置为 `KERNEL_MESSAGE_TIMER`，`dwParam` 字段，设置为定时器标识（即 `SetTimer` 调用中的 `dwTimerID` 参数）。

## CancelTimer 调用

`CancelTimer` 用来取消已经设定的定时器对象。该函数原型如下：

```
VOID CancelTimer(__COMMON_OBJECT* lpTimerObject);
```

其中，`lpTimerObject` 是调用 `SetTimer` 函数返回的对象指针，该指针指向了创建的定时器对象。一般情况下，该函数是用来取消永久定时器的，该函数调用后，线程设定的永久定时器会被删除。

对于一次定时器对象，该函数需要在定时器尚未超时前调用，用来取消已经设定的定时器对象，倘若定时器已经超时，再调用该函数，可能会引发异常，或者造成系统紊乱，但有的时候，用户线程可能不知道定时器已经超时（比如，定时器已经超时，并且给设定定时器的线程发送了一个消息，但该消息还没有来得及被处理，这时候设定定时器的线程就可能不知道定时器已经超时），这种情况下，建议用户不要调用该函数，而让该定时器超时，然后系统会自动删除定时器对象。



```

DWORD
(*DirectHandler)(LPVOID),
LPVOID lpParam,
DWORD
dwTimerFlags);
VOID
(*CancelTimer)(__COMMON_OBJECT*lpSystem,
__COMMON_OBJECT*lpTimerObject);
VOID (*ResetTimer)(__COMMON_OBJECT*
lpSystem,
__COMMON_OBJECT*
lpTimerObject);
... ..
END_DEFINE_OBJECT()
```

可以看出，lpTimerQueue 用来维护定时器对象，SetTimer、CancelTimer 和 ResetTimer 分别对应前面介绍的三个系统调用（在转换为系统调用的时候，对前两个参数 lpSystem 和 lpKernelThread，进行了省略，因为系统中只有一个 System 全局对象，所以缺省情况下，取系统中这个全局对象为第一个参数，第二个参数则是调用该函数的当前线程）。

- SetTimer 调用可以用来设定一个定时器对象，该函数操作过程如下：
- 5、调用 CreateObject（ObjectManager 对象提供的接口）函数，创建一个定时器对象；
  - 6、初始化该定时器对象（根据用户传递过来的参数）；
  - 7、把定时器对象插入定时器对象维护队列；
  - 8、重新设定调度时刻。

这里有两个问题，需要细致说明一下，第一个问题是，如何确定该定时器对象的超时刻问题。在当前版本的实现中，定时器的超时判断，是在时钟中断处理程序里进行的，系统维护一个全局变量 dwClockTickCount，用来记录时钟中断数量，即每发生一次时钟中断，该变量递增一，同样地，系统也维护了另外一个变量，dwNextTimerTick，用来表示下一个定时器超时时时钟中断个数，这样每次时钟中断的时候，中断处理程序就会比较这两个变量，如果这两个变量相同，则说明在这个时钟中断中，有至少一个定时器对象超时了，因此需要做进一步处理。对于 dwNextTimerTick 变量的设置，是这

样进行的（在 SetTimer 函数调用中）：

```
... ..
dwPriority = dwTimeSpan / SYSTEM_TIME_SLICE;
dwPriority += dwClockTickCount;
if(dwNextTimerTick > dwPriority)
    dwNextTimerTick = dwPriority;
... ..
```

其中，dwPriority 是一个临时变量，SYSTEM\_TIME\_SLICE 是系统定义的时间片，即每个时钟中断之间的时间间隔（以 ms 为单位，当前定义为 10），上述处理中，首先把 dwTimeSpan（定时器设定的等待时间，ms 为单位）换算成时钟中断个数（tick 个数），然后与当前 tick 计数(dwClockTickCount)相加，这样就得到了当前定时器超时时的 tick。所有这些完成之后，再跟 dwNextTimerTick 比较，如果小于 dwNextTimerTick，则设置当前 dwNextTimerTick 为 dwPriority，否则，保留 dwNextTimerTick 不变。dwPriority 大于 dwNextTimerTick，说明先前已经有定时器被设置，而且设置的定时器超时时间，比当前设置的要提前。

第二个问题是，定时器维护队列（lpTimerQueue）是一个优先队列，在插入该队列的时候，可以提供一个优先数值，以让队列确定插入对象应该在的位置，优先级越高，对象在优先队列中的位置越靠前。对于定时器对象，理想的处理方式是，离超时时刻越近，应该越提前，这样但这样跟优先队列的优先级确定方式刚好相反（优先级越大，越靠前），因此，为了把最先超时的定时器对象插入优先队列的最前面，采用下列方式：

```
dwPriority = MAX_DWORD_VALUE - dwPriority;
lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
                                (__COMMON_OBJECT*)lpTimerObject,
                                dwPriority);
```

其中，lpTimerObject 是 SetTimer 创建的定时器对象，而 MAX\_DWORD\_VALUE 则是一个最大的类型为 DWORD 的常数，在 32 位硬件平台上，定义为 0xFFFFFFFF，这样变换之后，就可以实现上述目的（即把最先超时的对象，插入优先队列的最前端）。

## 定时器超时处理

当前情况下，定时器超时处理是在时钟中断中完成的。每次时钟中断发生后，中断处理程序被调用，在中断处理程序中，会比较 `dwClockTickCounter` 和 `dwNextTimerTick` 两个变量，如果这两个变量相同，则说明当前至少有一个定时器对象超时，这时候，时钟中断处理程序会进行如下操作（是一个循环操作）：

- 5、从定时器对象队列（`lpTimerQueue`）中，提取第一个定时器对象（定时器对象已经按照超时先后进行排序，排在最前面的定时器对象，是最先超时的定时器）；
- 6、获取定时器对象的超时时刻（即定时器对象在优先队列中的优先级，参考 `SetTimer` 函数的处理），然后跟当前 `dwNextTimerTick` 比较，如果不等于 `dwNextTimerTick`，则跳出当前循环，如果等于 `dwNextTimerTick`，则说明该定时器对象超时，于是执行下列操作：
  - a) 判断回调函数（`DirectHandler`）是否为空，如果不为空，则调用该回调函数，以 `lpHandlerParam` 为参数）；
  - b) 如果为空，则给设置该定时器对象的线程发送一个消息（`KERNEL_MESSAGE_TIMER`）；
- 7、完成上述超时处理后，再判断当前定时器对象是一次性定时器，还是永久定时器，如果是一次定时器，则删除该定时器对象，如果是永久定时器，则重新更新该定时器的超时时间，并重新插入定时器对象队列（其插入优先级的确定方式与 `SetTimer` 操作雷同）；
- 8、然后再从定时器队列中提取下一个定时器对象，并转到步骤 1，进行循环处理。

在定时器对象的超时时刻（以 tick 计）不等于 `dwNextTimerTick` 的时候，上述循环结束，这时候当前定时器对象，将是目前所有的定时器对象中，最先超时的定时器对象，因此，需要根据该定时器的超时时刻，重新计算 `dwNextTimerTick` 的值，并更新 `dwNextTimerTick`，然后把该定时器对象重新插入定时器对象队列（因为该定时器对象虽然已被从定时器队列中提取出来，但由于没有超时，所以没有被处理，需要重新插入定时器对象队列，等待下一个超时时刻）。下面是定时器对象的处理代码：

```
if(System.dwClockTickCounter == System.dwNextTimerTick)    //Should schedule
timer.
{
    lpTimerQueue = System.lpTimerQueue;
    lpTimerObject = (__TIMER_OBJECT*)lpTimerQueue->GetHeaderElement(
        (__COMMON_OBJECT*)lpTimerQueue,
        &dwPriority);
```

```

if(NULL == lpTimerObject)
    goto __CONTINUE_1;
dwPriority = MAX_DWORD_VALUE - dwPriority;
while(dwPriority == System.dwNextTimerTick)
{
    if(NULL == lpTimerObject->DirectTimerHandler) //Send a message to the
kernel thread.
    {
        Msg.wCommand = KERNEL_MESSAGE_TIMER;
        Msg.dwParam  = lpTimerObject->dwTimerID;
        KernelThreadManager.SendMessage(
            (__COMMON_OBJECT*)lpTimerObject->lpKernelThread,
            &Msg);
    }
    else
    {
        lpTimerObject->DirectTimerHandler(
            lpTimerObject->lpHandlerParam);    //Call the associated handler.
    }

    switch(lpTimerObject->dwTimerFlags)
    {
        case TIMER_FLAGS_ONCE:                //Delete the timer object processed just
now.
            ObjectManager.DestroyObject(&ObjectManager,
                (__COMMON_OBJECT*)lpTimerObject);
            break;
        case TIMER_FLAGS_ALWAYS:              //Re-insert the timer object into timer
queue.
            dwPriority  = lpTimerObject->dwTimeSpan;
            dwPriority /= SYSTEM_TIME_SLICE;
            dwPriority += System.dwClockTickCounter;
            dwPriority  = MAX_DWORD_VALUE - dwPriority;

```

```

lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
                               (__COMMON_OBJECT*)lpTimerObject,
                               dwPriority);

    break;

default:
    break;
}

lpTimerObject = (__TIMER_OBJECT*)lpTimerQueue->GetHeaderElement(
    (__COMMON_OBJECT*)lpTimerQueue,
    &dwPriority);    //Check another timer object.
if(NULL == lpTimerObject)
    break;
dwPriority = MAX_DWORD_VALUE - dwPriority;
}

```

上面的处理中，是一个循环操作，从定时器队列中，提取第一个定时器对象，计算定时器对象的超时时刻（以 tick 计，超时时刻等于 MAX\_DWORD\_VALUE 减去定时器对象在优先队列中的优先级，请参考 SetTimer 函数的处理），并判断提取的定时器对象的超时时刻是否跟 dwNextTimerTick 相同，如果相同，说明该定时器已经超时，然后进一步处理，如果不相同，则说明超时的定时器对象已经处理完毕（只要优先队列中，队头对象没有超时，后边的对象肯定不会超时，因为定时器对象是按照超时先后顺序，插入优先队列的），这时候需要跳出循环。

下面的代码，更新了下一个超时时刻（dwNextTimerTick），并把当前定时器对象重新插入定时器对象队列：

```

if(NULL == lpTimerObject)    //There is no timer object in queue.
{
    ENTER_CRITICAL_SECTION();
    System.dwNextTimerTick = 0L;
    LEAVE_CRITICAL_SECTION();
}
else
{

```

```

ENTER_CRITICAL_SECTION();
System.dwNextTimerTick = dwPriority;    //Update the next timer tick
counter.

LEAVE_CRITICAL_SECTION();
dwPriority = MAX_DWORD_VALUE - dwPriority;
lpTimerQueue->InsertIntoQueue((__COMMON_OBJECT*)lpTimerQueue,
    (__COMMON_OBJECT*)lpTimerObject,
    dwPriority);
    }
}

```

## 定时器取消处理

定时器取消处理比较简单：

- 4、首先从定时器对象队列中，把取消的处理器对象删除；
- 5、调用 DestroyObject 函数（ObjectManager 提供的接口），销毁 timer 对象；
- 6、更新 dwNextTimerTick 变量。

如下代码：

```

static VOID CancelTimer(__COMMON_OBJECT* lpThis,__COMMON_OBJECT*
lpTimer)
{
    __SYSTEM*          lpSystem      = NULL;
    DWORD              dwPriority     = 0L;
    __TIMER_OBJECT*    lpTimerObject = NULL;

    if((NULL == lpThis) || (NULL == lpTimer))
        return;
}

```

```

    lpSystem = (__SYSTEM*)lpThis;
    lpSystem->lpTimerQueue->DeleteFromQueue((__COMMON_OBJECT*)lpSystem->lpT
imerQueue,
        lpTimer);
    ObjectManager.DestroyObject(&ObjectManager,lpTimer);

    lpTimerObject = (__TIMER_OBJECT*)
        lpSystem->lpTimerQueue->GetHeaderElement(
            (__COMMON_OBJECT*)lpSystem->lpTimerQueue,
            &dwPriority);
    if(NULL == lpTimerObject)    //There is not any timer object to be processed.
        return;

    //
    //The following code updates the tick counter that timer object should be processed.
    //
    dwPriority = MAX_DWORD_VALUE - dwPriority;
    if(dwPriority > lpSystem->dwNextTimerTick)
        lpSystem->dwNextTimerTick = dwPriority;
    dwPriority = MAX_DWORD_VALUE - dwPriority;
    lpSystem->lpTimerQueue->InsertIntoQueue(
        (__COMMON_OBJECT*)lpSystem->lpTimerQueue,
        (__COMMON_OBJECT*)lpTimerObject,
        dwPriority);    //Insert into timer object queue.

    return;
}

```

上面的处理中，首先从定时器对象队列中删除要取消的定时器对象，并删除该对象，然后尝试从定时器队列中提取第一个对象（最先超时的定时器对象），如果能成功，说明目前尚有定时器对象，于是根据获取的对象的超时时刻，更新 `dwNextTimerTick` 变量，如果提取失败（返回 `NULL`），说明目前定时器队列中，已经没有定时器对象，这时候，只需要返回即可。

# 定时器复位

对于定时器复位操作，相对简单，主要完成下列工作：

- 4、从定时器队列中，删除要复位的定时器对象；
- 5、重新计算定时器对象的超时时刻，并重新插入定时器队列；
- 6、更新 dwNextTimerTick 变量。

# 定时器注意事项

考虑到定时器的实现机制，以及目标系统的性能要求，在实际应用中，使用定时器服务的时候，下列规则需要遵循：

- 4、在使用定时器服务时，如果不是十分必要，建议不要使用回调通知机制（即设定一个回调函数）。因为定时器的回调函数，是在时钟中断处理程序中被调用的，如果有大量的回调函数存在，会大大降低系统性能。下面这个测试，是在一台 Pentium IV 2.5G 处理器上，做的测试结果：

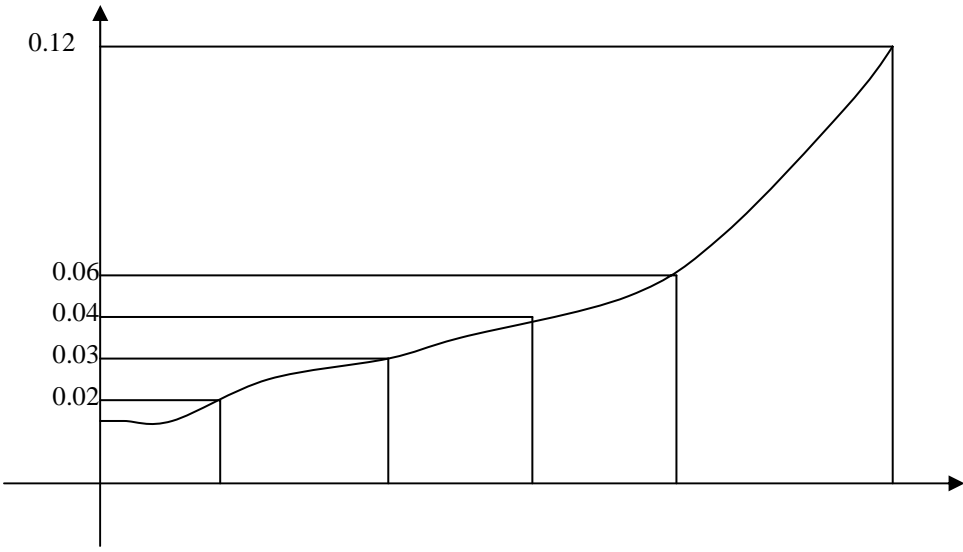


图 8-6 一个测试结果

其中，横坐标是一次定时器个数，其中，每个一次定时器的超时处理，都采用回调函数的方式，每个回调函数，所做的处理操作都一样，都是从一个优先队列中删除一个对象（该优先队列只有一个对象），因此，可认为回调函数的处理时间，都是一样的。纵坐标则是处理器的处理时间（通过记录处理器时钟周期个数来计算），可以看出，当一次定时器个数在 20 个的时候，处理时间达到了 0.06ms，如果上升到 30 个，则处理时间达到了 0.12ms。

但如果一次定时器采用发送消息的超时处理机制（即发送一个消息给设定定时器的线程），则如果处理时间为 0.12ms，需要设定 250 多个一次定时器。因此，可以看出，采用回调函数超时机制的定时器，其开销比采用消息通知机制的定时器大得多。

- 5、如果一定要采用回调函数机制来处理超时，建议回调函数的处理时间不能太长，一般情况下，处理时间不能大于 0.01ms；
- 6、对于采用消息通知作为超时处理机制的定时器，会引入误差。比如，假设设定的定时器，在 T1 时刻超时，则在 T1 时刻，设定定时器的线程会收到一个定时器超时消息，但真正处理该定时器超时消息的时间，可能会延迟到 T2，如下图所示：

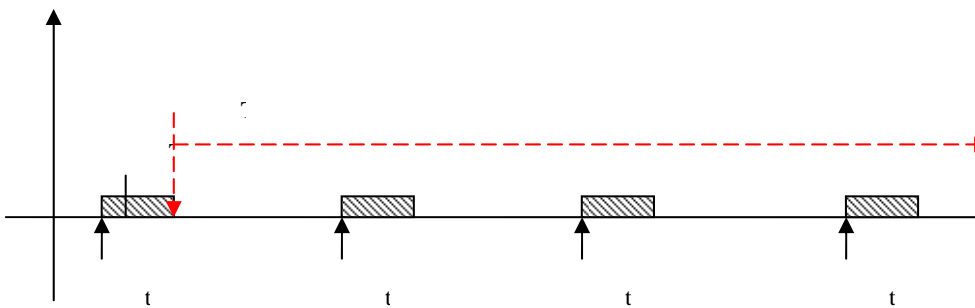


图 8-7 定时器消息的处理延时

图中， $t_N$  ( $t_1, t_2, t_3, \dots$ ) 是时钟中断发生时刻，其中， $t_N$  之间间隔均匀，在  $T_1$  时刻，系统给设定定时器的线程发送一个消息，然后继续执行时钟中断处理程序。中断处理程

序执行完毕之后，系统会选择线程就绪队列中，优先级最高的线程投入运行。这个时候，如果设定定时器的线程是优先级最高的线程，那么可能马上投入运行，这时候，定时器超时消息可能会被处理，因此，图中的 T2 时刻，是定时器超时消息得到处理的最早时刻。如果设定定时器对象的线程，优先级不是最高的，那么这个时候，就不会被调度，其它优先级更高的就绪线程会被调度，因此这个时候，定时器消息将一直存放在设定它的线程队列中，除非设定线程得到调度，否则定时器超时消息将一直不能被处理。因此，定时器超时消息的处理时间，可能会进一步延迟，一种最糟糕的情况就是，定时器超时消息可能永远得不到处理（在设定线程永远得不到调度的时候发生）。

但是对于采用回调函数作为超时处理机制的定时器，在 T1 时刻，回调函数就可以被调用，即超时消息马上被处理。因此，如果是一些时间要求十分严格的场合，建议使用回调机制处理超时。

## 第九章 系统总线管理

### ——System BUS management

## 9.1 系统总线概述

### 9.1.1 系统总线

系统总线是计算机系统的枢纽，所有的外部设备，都是连接在系统总线上的，甚至可以把 CPU、内存等部件，也看作是系统总线上的“设备”。因此，对系统总线的管理，以及对系统总线上设备的管理，是操作系统的一个十分核心的功能。

一般情况下，操作系统都定义一个管理框架，对系统总线和总线上的设备进行统一管理。这个管理框架的好与坏，对于操作系统的可扩展性、性能、编程的难易程度等，影响十分关键。一个好的设备和总线管理框架，可以使得物理设备的管理十分简便，使得驱动程序的编写和调试十分简便，也可以大大提高系统的运行效率。相反，一个不好的总线管理框架，可能使得系统无法扩展，无法支持更多的物理设备，也使得设备驱动程序编写工作十分烦杂，效率也得不到保证。在 Hello China 的设计当中，充分考虑了系统总线和设备的管理框架，建立了一种相对开放的体系结构，使得该框架可以很容易的容纳新的设备，并严格定义了管理框架和设备驱动程序之间的接口，使得驱动程序的编写相对容易。

目前，在计算机系统中，存在很多类型的总线，比如，ISA、PCI、EISA、USB 等，甚至 RS-232 标准，也可以认为是一种总线。在 Hello China 的设计中，定义的总线管理框架可以容纳上述各种类型的总线，但目前的版本中，只对 ISA 和 PCI 两种类型的总线，进行了实现，因为这是目前最常见的总线类型。在后续版本的实现中，将继续增加对 USB 等总线的支持。在本章中，我们对 Hello China 的总线管理框架（模型）进行描述，并对 PCI 总线驱动程序的实现，做了详细描述。

### 9.1.2 总线管理模型

在当前版本 Hello China 的实现中，所有对总线和物理设备的管理功能，抽象到一个全局对象 DeviceManager 中。该对象定义如下：

```
BEGIN_DEFINE_OBJECT(__DEVICE_MANAGER)
#define MAX_BUS_NUM 16
    __SYSTEM_BUS          SystemBus[MAX_BUS_NUM];
    __RESOURCE             FreePortResource;
    __RESOURCE             UsedPortResource;
    BOOL                   (*Initialize)(__DEVICE_MANAGER*);
```

```

        __PHYSICAL_DEVICE*      (*GetDevice)(__DEVICE_MANAGER*,
                                           DWORD
dwBusType,
                                           __IDENTIFIER*
lpIdentifier,
                                           __PHYSICAL_DEVICE* lpStart);
        __RESOURCE*              (*GetResource)(__DEVICE_MANAGER*,
                                           DWORD
dwBusType,
                                           DWORD
dwResType,
                                           __IDENTIFIER*
lpIdentifier);
        BOOL                      (*AppendDevice)(__DEVICE_MANAGER*,
                                           __PHYSICAL_DEVICE*);
        VOID                      (*DeleteDevice)(__DEVICE_MANAGER*,
                                           __PHYSICAL_DEVICE*);
        BOOL                      (*CheckPortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*);
        __RESOURCE*              (*ReservePortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*,
                                           DWORD dwLength);
        VOID                      (*ReleasePortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*);

END_DEFINE_OBJECT()
```

由于这是一个全局对象，系统中存在一个，因此没有从\_\_COMMON\_OBJECT 继承。这个对象完成所有系统总线的管理，以及相应设备的管理。在操作系统初始化的时候，这个对象的 Initialize 函数被调用，用来完成该对象的初始化。如果初始化成功（Initialize 函数返回 TRUE），则系统会继续进行下一步的工作，如果初始化失败，则系统停止启动，进入死循环。

当前版本的实现中，对系统总线进行了抽象，定义了一个统一的数据结构 \_\_SYSTEM\_BUS，用来描述任何系统总线（包括 PCI、ISA、EISA、USB 等），该对象定义如下：

```

BEGIN_DEFINE_OBJECT(__SYSTEM_BUS)
    __SYSTEM_BUS*          lpParentBus;
    __PHYSICAL_DEVICE*      lpDevListHdr;
    __PHYSICAL_DEVICE*      lpHomeBridge;
    __RESOURCE               Resource;

    DWORD                   dwBusNum;
    DWORD                   dwBusType;
END_DEFINE_OBJECT()

```

在 DeviceManager 对象中，维护了一个数组 SystemBus，这个数组用来保存系统中当前存在的总线。在 DeviceManager 初始化的时候，会对系统中的总线进行检测，每检测到一条总线，就占用 SystemBus 数组的一个元素，在这个元素内，记录检测到的总线的相关信息。注意 \_\_SYSTEM\_BUS 定义中的 dwBusType 字段，该字段指出了当前系统总线的类型，目前有如下定义：

```

#define BUS_TYPE_NULL      0x00000000
#define BUS_TYPE_PCI       0x00000001
#define BUS_TYPE_ISA       0x00000002
#define BUS_TYPE_EISA      0x00000004
#define BUS_TYPE_USB       0x00000008

```

如果 dwBusType 的值为 BUS\_TYPE\_NULL，则说明当前系统总线对象尚没有被初始化。这样总线驱动程序如果要向 SystemBus 数组中，加入一条总线，那么就可以通过这个字段，来判断 SystemBus 数组中，哪个元素尚未被占用。

为了以模块化的形式，实现对总线和设备的管理，对于总线，也当作设备来看待，在 Hello China 当前版本的实现中，对于目前流行的总线类型，都编写了对应的总线驱动程序，对于总线的检测、总线设备的枚举以及总线设备的配置等，都是由特定的总线驱动程序来完成的。这些总线驱动程序被静态编译到 Hello China 的内核中。

这样在 DeviceManager 对象的初始化过程中，只需要加载相应的总线驱动程序，由总线驱动程序完成特定总线的检测和设备的枚举。比如，目前版本的 Hello China 中，实现了两种总线—ISA 和 PCI—的总线驱动程序，这样 DeviceManager 在初始化的时候，就需要执行下列操作：

```

BOOL Initialize(__DEVICE_MANAGER* lpDeviceMgr)
{
    BOOL    bResult = FALSE;

    if(NULL == lpDeviceMgr)
        return bResult;

    if(PciDriver(lpDeviceMgr))
    {
        bResult = TRUE;
    }
    if(IsaDriver(lpDeviceMgr))
    {
        bResult = TRUE;
    }
    //
    //Load more bus drivers here.
    //
    return bResult;
}

```

上述处理中，只要有一种类型的总线驱动程序加载成功，`Initialize` 函数就会成功返回。如果任何一类总线的驱动程序都没有加载成功，则 `Initialize` 返回失败，这种情况下，系统无法启动。实际上，根据 `Hello China` 目前的实现，ISA 总线的驱动程序，总是会成功加载的，因为该总线肯定是存在的，如果该总线不存在，则 `Hello China` 根本无法引导（从软驱或硬盘）。

每种不同的总线驱动程序，所完成的工作主要是检测总线是否存在，如果存在，则填写一个 `SystemBus` 数组的元素，然后对该总线上的设备进行枚举。对于连接在总线上的设备，也以抽象的数据结构 `__PHYSICAL_DEVICE` 进行描述。该结构的定义如下：

```

BEGIN_DEFINE_OBJECT(__PHYSICAL_DEVICE)
    __IDENTIFIER                DevId;
    CHAR                        strName[MAX_DEV_NAME];

```

```

#define MAX_RESOURCE_NUM    7

__RESOURCE                  Resource[MAX_RESOURCE_NUM];
__PHYSICAL_DEVICE*          lpNext;
//__PHYSICAL_DEVICE*        lpPrev;
__SYSTEM_BUS*               lpHomeBus;
__SYSTEM_BUS*               lpChildBus;
LPVOID                      lpPrivateInfo;

END_DEFINE_OBJECT()

```

在上述定义中，每个设备所占用的资源，包括中断向量号、内存映射地址、IO 端口映射地址等，都被记录在 **Resource** 数组内，当前版本的实现中，**Resource** 数组的最大长度为 7，这可以满足常见的总线类型的需要。**lpPrivateInfo** 成员是一个可以保存任何数据结构的指针变量，用于保存不同的设备类型所特有的信息。比如，针对 **PCI** 设备，这个成员指向一个保存 **PCI** 接口设备的数据结构。

在总线对象（**\_\_SYSTEM\_BUS**）的定义中，定义了一个成员变量 **DeviceListHeader**，这个变量把位于该总线上的所有设备，连接成一个双向链表，而每个物理设备对象的定义中，也包含一个成员变量 **lpHomeBus**，指向该设备所在的总线对象。在物理设备定义中，另外一个总线指针 **lpChildBus**，用在该设备是一个总线桥设备的情况，用来指出该设备所连接的下一级总线。而在总线对象（**\_\_SYSTEM\_BUS**）的定义中，也有一个成员变量 **lpHomeBridege**，用来表明该总线所连接到的桥设备。需要注意的是，对总线设备的枚举，都是在总线驱动程序内完成的。这样当系统中所有的总线驱动程序被加载，并成功完成初始化后，将建立如下的数据结构：

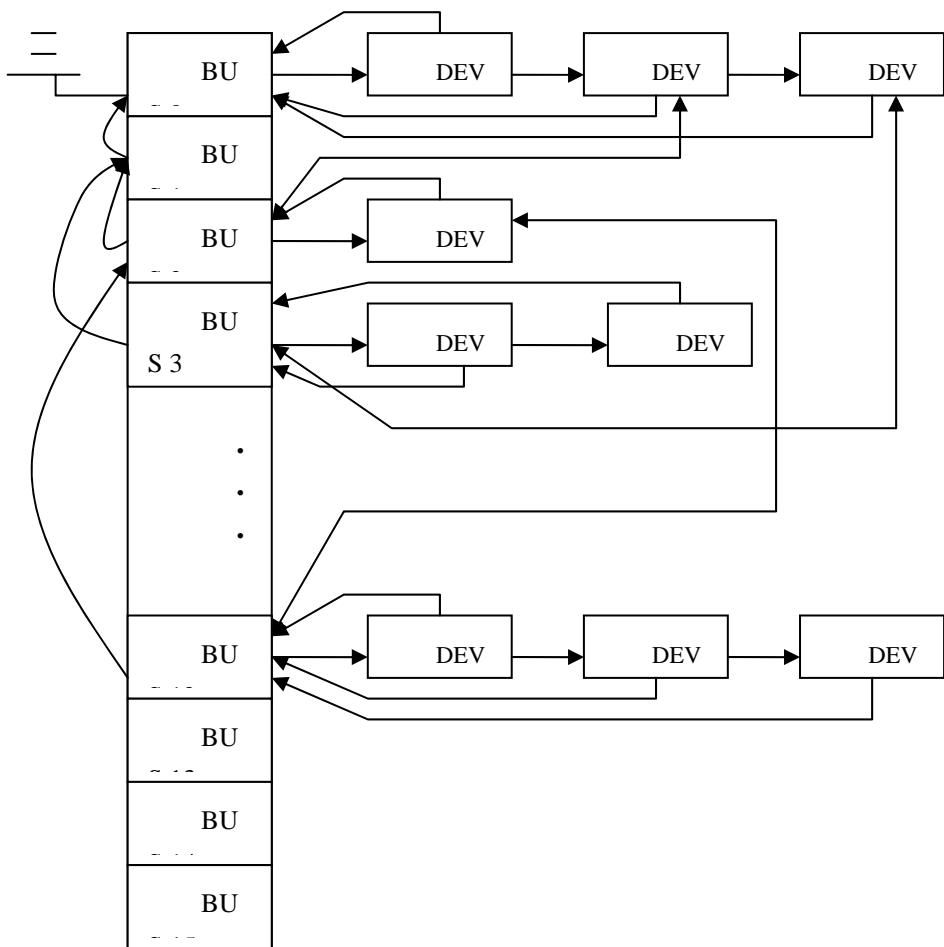


图 9-1 Hello China 的系统总线管理结构

其中，BUS0 是所有总线的父总线，BUS1 是 BUS2 和 BUS3 的父总线，而 BUS2 则是 BUS12 的父总线。每条总线上的设备，都以双向链表的方式，串接在一起，每个设备，都包含了一个指向所在总线的指针。总线 0（BUS0）的第二和第三个设备，是两个总线桥接设备，其中，第一个桥接设备，其下一级总线（lpChildBus）是总线 2（BUS2），而第二个桥接设备，其下一级总线是总线 3。在总线二（BUS2）上的第一个设备，也是桥接设备，其下一级总线是总线 12。

需要说明的是，有一些总线类型，是无法自动枚举连接在其上的设备的，比如 ISA 总线。对于这种总线，只能由驱动程序来探测相应的设备是否存在，如果驱动程序能够探测到设备存在，则由驱动程序创建一个物理设备对象，然后调用 `AppendDevice` 函数 (`DeviceManager` 对象提供)，静态的添加到系统总线上。

### 9.1.3 设备标识符

为了标识总线上的设备，定义一个设备标识符对象，用来对不同总线上的设备进行标识，如下：

```
BEGIN_DEFINE_OBJECT(__IDENTIFIER)
    DWORD          dwBusType;
    union{
        struct{
            UCHAR    ucMask;
            WORD      wVendor;
            WORD      wDevice;
            DWORD     dwClass;
            UCHAR     ucHdrType;
            WORD       wReserved;
        }PCI_Identifier;
        struct{
            DWORD     dwDevice;
        }ISA_Identifier;
    }
END_DEFINE_OBJECT()
```

这个对象的解释，是与总线类型相关的，即该对象的第一个成员变量 `dwBusType`，决定了具体的标识符内容。在具体的总线设备驱动程序的描述中，会对该对象进行详细的叙述。

## 9.2 系统资源管理

所谓系统资源，指的是 IO 端口、内存映射区域、中断引脚（向量）等被所有设备共享的资源。有些 CPU 类型，比如 Power PC，没有 IO 端口的概念，这个时候的资源管理，就只限于内存映射区域。

在系统范围内，这些资源的数量是有限的，比如针对 IO 端口资源，可使用的范围为 0—65535（16 比特范围），如果不对这些资源进行统一管理和分配，则可能会出现资源冲突的情况，比如，两个物理设备占用了同一范围的 IO 端口。这样不但设备无法正常工作，严重情况下，还可能造成整个系统崩溃。因此，需要系统统一对这些资源进行调配，以保证资源的充分利用，并保证资源的分配不会出现冲突。

在 Hello China 的实现中，对于内存映射区域资源，由虚拟内存管理器（VirtualMemoryMgr）对象统一管理，设备如果想使用一段虚拟内存区域（内存映射区域），必须调用 VirtualAlloc 函数来进行分配，该函数可以保证资源不会出现冲突，因为如果设备申请的资源已经被占用，则该函数会反户 NULL。对于中断资源，由于目前 Hello China 的中断机制，支持中断嵌套，因此，如果驱动程序严格按照 Hello China 的中断机制定义的协议进行编写，则不会出现中断冲突的现象，即使不同的设备连接到了同一条中断引脚上，也不会出现问题。因此，目前情况下，需要进行统一调配的资源，就只有 IO 端口资源。

目前的实现中，完成对 IO 端口进行统一分配和管理的对象，就是 DeviceManager 对象。在该对象中，定义了两个成员变量：FreePortResource 和 UsedPortResource，这两个变量分别把系统中可以使用的 IO 端口，以及已经使用的 IO 端口资源，以双向链表的形式，串连在一起。

### 9.2.1 资源描述对象

为了对系统中的资源进行描述，定义如下的资源对象：

```
BEGIN_DEFINE_OBJECT(__RESOURCE)
```

```

__RESOURCE*    lpPrev;
__RESOURCE*    lpNext;
DWORD          dwResType;
union{
    struct{
        WORD    wStartPort;
        WORD    wEndPort;
    }
};

```

```

        }IOPort;
    struct{
        LPVOID    lpStartAddr;
        LPVOID    lpEndAddr;
    }MemoryRegion;
    UCHAR        ucVector;
};
END_DEFINE_OBJECT()

```

这个对象的定义中，dwResType 用来决定当前对象所装载的资源类型，dwResType 的取值如下：

```

#define RESOURCE_TYPE_IO            0x00000001
#define RESOURCE_TYPE_INTERRUPT    0x00000002
#define RESOURCE_TYPE_MEMORY       0x00000004
#define RESOURCE_TYPE_EMPTY        0x00000000

```

如果 dwResType 的值是 RESOURCE\_TYPE\_MEMORY，则当前的资源对象中，保存的是内存映射区域资源。顾名思义，lpPrev 和 lpNext 两个指针，把资源对象串连在一个双向链表中。另外，RESOURCE\_TYPE\_EMPTY 是一个标记，标明了当前资源描述对象不包含任何资源信息。在物理设备对象的定义中，为了描述该设备所占用的资源信息，专门定义了一个数组—Resource，来管理设备所占用的资源。在当前版本的实现中，这个数组的元素，固定为 MAX\_RESOURCE\_NUM（当前定义为 7）个，这样如果设备所占用的资源，少于该数值，这样剩余的 Resource 数组的元素，其 dwResType 就设置为 RESOURCE\_TYPE\_EMPTY。比如，一个物理设备，仅仅占用了—个端口范围，一个中断向量号，这样，Resource 数组的前两项就描述了端口范围和中断向量两项资源，后续的 5 个元素，其 dwResType 标志都设置为 RESOURCE\_TYPE\_EMPTY。

### 9.2.3 IO 端口资源管理

在当前版本的 Hello China 实现中，由 DeviceManager 对象对 IO 端口资源进行了统一管理。在 DeviceManager 对象的定义中，定义了两个成员变量：UsedPortResource 和 FreePortResource。其中，UsedPortResource 把已经使用的端口资源，连接成一个双向链表，而 FreePortResource，则把当前尚且空闲的资源，连接成一个双向链表。例如：

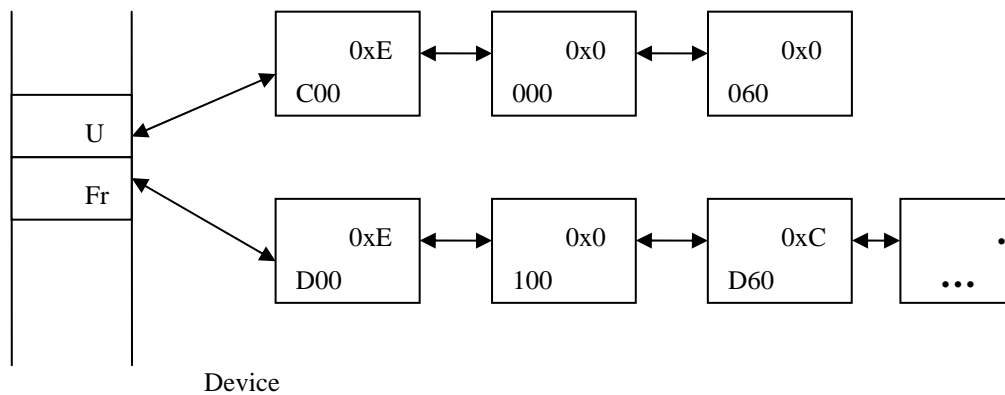


图 9-2 Hello China 的端口资源管理

开始的时候，UsedPortResource 链表为空，即没有任何 IO 端口资源被使用，而 FreePortResource 则包含了所有可用的 IO 端口资源（0x0000 — 0xFFFF）。一旦有驱动程序被加载，则驱动程序会调用 ReservePortRegion 函数，以预留 IO 端口资源，这样如果请求预留的端口资源，没有被使用（位于 FreePortResource 链表内），则 DeviceManager 会创建一个 \_\_RESOURCE 对象，根据 ReservePortRegion 的参数，对该 Resource 对象初始化，并把对象插入到 UsedPortResource 链表中。需要注意的是，在插入使用链表的同时，会从 FreePortResource 链表中删除相应的资源。如果请求预留的资源，是一块连续端口资源的一部分，比如，用户请求预留资源 0x0010 到 0x001F，而位于空闲链表中的资源，是包含用户请求资源的更大范围的空闲端口，比如 0x0000 到 0x01FF，则 DeviceManager 就会执行一个拆分动作，把 0x0000 到 0x01FF 的资源，拆分成三部分：0x0000—0x000F、0x0010—0x001F、0x0020—0x01FF，然后把中间部分，返回给用户（添加到使用链表中），把剩余的两部分重新插入空闲链表。

相反，如果驱动程序释放了一个端口范围，则 DeviceManager 会把释放的端口范围，插入到空闲资源链表中，并进行一个合并操作，把零散的（但是连续的）端口范围，尽量合并成一个连续的端口范围。



### 9.3.3 CheckPortRegion

该函数用来检查一段 IO 端口区域是否已经被占用。有些总线类型，比如 ISA，不支持自动配置，这样总线驱动程序就无法为总线上的设备统一分配 IO 端口资源，这种情况下，就需要设备驱动程序自己分配 IO 端口资源。这样为了不导致资源冲突，设备驱动程序在实际使用端口范围前，首先使用该函数来确认，自己即将使用的端口资源是否已经被占用。如果被占用，则该函数返回 FALSE，否则返回 TRUE。在端口资源占用的情况下，设备驱动程序必须放弃使用相应的端口资源，或者选择另外的端口范围，或者停止工作。

该函数原型如下：

```
BOOL CheckPortRegion(__DEVICE_MANAGER* lpThis,  
                     __RESOURCE* lpResource);
```

其中，lpResource 指向一个资源描述对象 (\_\_RESOURCE)，该资源对象的资源类型，必须为 RESOURCE\_TYPE\_IO。

### 9.3.4 ReservePortRegion

顾名思义，该函数用于预留部分端口资源，原型如下：

```
__RESOURCE ReservePortRegion(__DEVICE_MANAGER* lpThis,  
                             __RESOURCE* lpResource,  
                             DWORD dwLength);
```

其中，lpResource 是一个资源描述对象，调用者可以使用该对象，指定一个期望预留的端口范围，这样 ReservePortRegion 函数会优先判断，lpResource 指定的端口资源是否已经被使用。如果已经被使用，则重新为调用者预留 dwLength 的资源，否则，预留调用者指定的端口资源。当然，如果调用者指定的资源已经被占用，而且 FreePortResource 链表中又没有足够的端口范围（长度大于或等于 dwLength），则返回 NULL。

如果驱动程序必须使用自己指定的资源，而 ReservePortRegion 函数返回了另外的端口范围（指定的范围已经被占用），则驱动程序必须调用 ReleasePortRegion 函数，释放返回的资源。

另，该函数返回一个资源描述对象的地址，该对象由 ReservePortRegion 创建，设备

驱动程序从该对象中提取出相应的资源信息后（保存在自己创建的资源描述对象中，或保存在本地变量中），必须调用 `KMemFree` 函数，销毁该对象。

### 9.3.5 ReleasePortRegion

该函数释放 `ReservePortRegion` 函数预留的端口范围，原型如下：

```
VOID ReleasePortRegion(__DEVICE_MANAGER*   lpThis,
                      __RESOURCE*         lpResource);
```

### 9.3.6 AppendDevice

一些不支持设备自动发现的总线类型，比如 `ISA` 等，需要设备驱动程序来静态的检测设备的存在，并进行配置。对于这类设备驱动程序，在检测到一个实际的物理设备的时候，必须调用该函数，向 `DeviceManager` 注册相应的设备。这样做的目的，是使得 `DeviceManager` 能够统一的管理系统中的所有硬件设备。该函数原型如下：

```
BOOL AppendDevice(__DEVICE_MANAGER* lpThis,
                  __PHYSICAL_DEVICE* lpDev);
```

其中，`lpDev` 是设备驱动程序创建的一个物理设备对象，并由设备驱动程序对之进行初始化。在设备驱动程序存在的时间内（设备驱动程序被卸载前），必须一致保持这个物理设备对象的有效性（即不能销毁该对象），在设备驱动程序被卸载的时候，销毁该物理设备对象，但在销毁前，一定要调用 `DeleteDevice` 函数，来从系统中删除该物理设备，否则可能会导致系统异常，甚至崩溃。

### 9.3.7 DeleteDevice

该函数用于删除通过 `AppendDevice` 函数向系统中增加的物理设备对象，原型如下：

```
VOID DeleteDevice(__DEVICE_MANAGER*   lpThis,
                  __PHYSICAL_DEVICE* lpDev);
```

## 9.4 PCI 总线驱动程序概述

### 9.4.1 PCI 总线概述

PCI (Peripheral Component Interconnect)总线是一种同步的、独立于处理器的、32 位或 64 位局部总线，其目的是在高集成度的外设控制器件、扩展板(add-in board)、和处理器/存储器系统之间提供一种内部连接机制，下图是一个典型 PCI 系统框图：

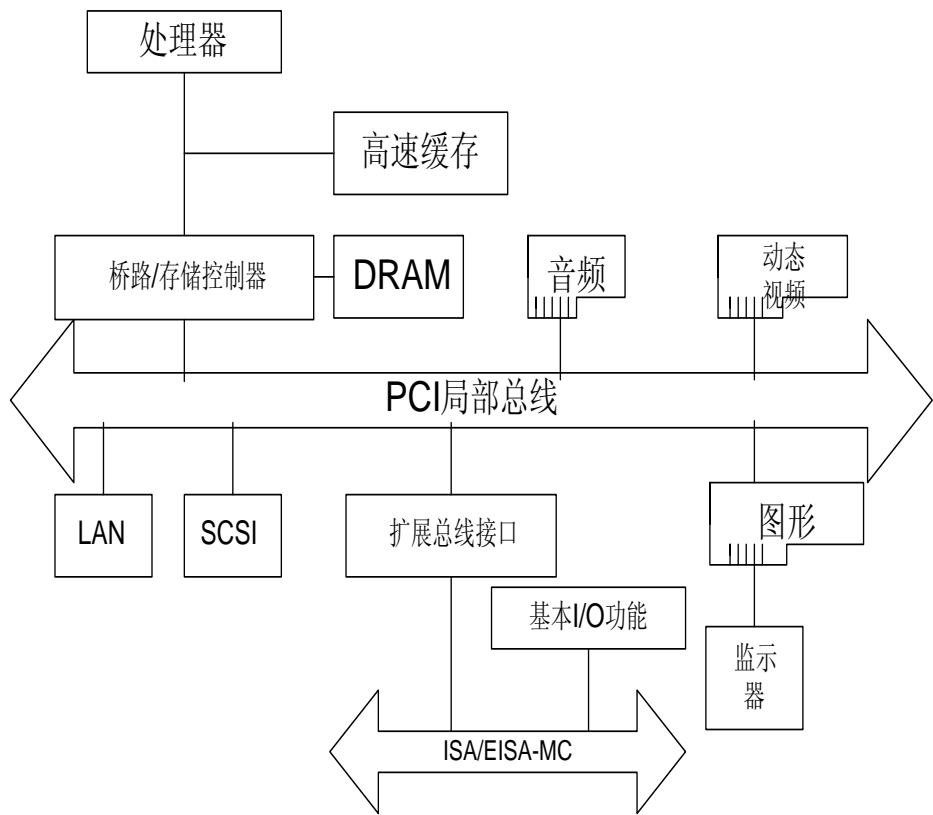


图 9-3 一个典型的基于 PCI 总线的计算机结构

可以看出，处理器、高速缓存、存储器子系统通过桥路与 PCI 总线相连接，该桥路提供一条低时间延迟的通道，通过它，处理器能直接操作任何映射到存储器或 I/O 空间

的设备，它也提供一条高带宽通道，使 PCI 总线主控设备能直接操作主存储器。该桥路可以选择包括以下功能：数据缓存/停驻和 PCI 核心功能(即仲裁)，一般情况下，这种桥路称为 HOST-PCI 桥，也俗称“北桥”，相应地，PCI 总线上连接 ISA 总线的桥路，称为“南桥”。

## 9.4.2 PCI 设备的配置空间

PCI 规范规定了配置空间以满足现在及将来系统配置机制的要求。这种配置机制反映了设备的功能和状态，提供了无用户参与的安装、配置和引导，全部设备重定位，由独立于设备的软件统一完成设备的配置，包括分配 IO 端口资源、内存映射资源，以及分配（或配置）中断向量等。一般情况下，这种配置软件集成在操作系统核心，在目前 Hello China 的实现中，对 PCI 总线设备的枚举、配置等操作，都是由 PCI 总线驱动程序实现的。

按照 PCI 规范的定义，设备的配置空间必须是任何时候都可操作的，不仅是在系统引导期间，在系统正常运行的过程中，也可以通过软件对配置空间的内容进行修改。PCI 总线配置软件需要扫描 PCI 总线以确定当前总线上存在哪些设备，0xFFFF 是无效的设备供应商（Vendor）标识符，因而，如果在对应的 PCI 槽位上不存在一个实际的物理设备，PCI 的总线桥路可以返回一个全“1”的值。

PCI 总线规范规定了 256 字节的配置空间，这个空间分为 64 字节的预定义首部和 192 字节的设备相关首部。在每个字段中，设备只需要实现必要的和相关的寄存器。其中，预定义的 64 字节的首部，也会因为设备类型的不同而不同，进一步分为 0 型头部和 1 型头部。比如，针对普通的设备（0 型头部），预定义的 64 字节首部组织如下：

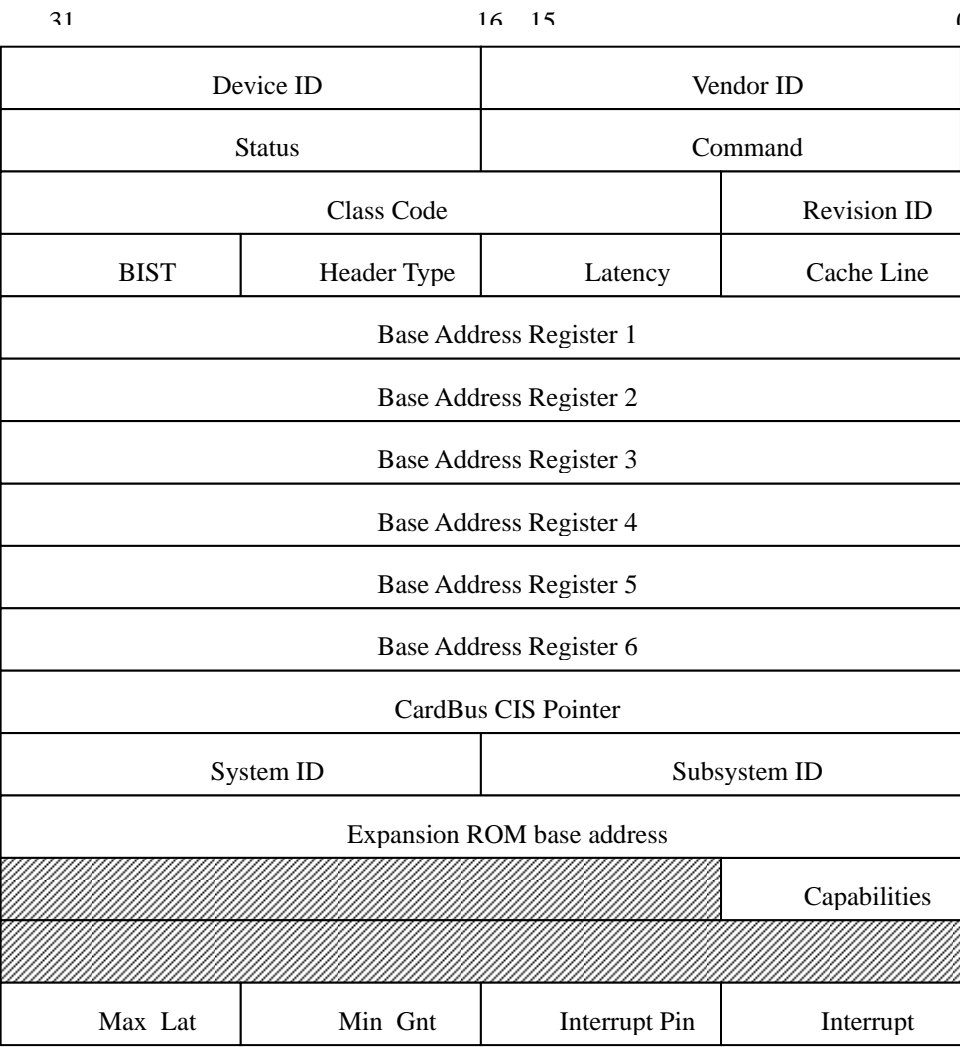


图 9-4 PCI 配置空间布局（0 型头部）

而对于 PCI-PCI 总线（1 型头部），则预定义的首部结构如下：

31		16		15		0	
Device ID				Vendor ID			
Status				Command			
Class Code						Revision ID	
BIST		Header Type					
Base Address Register1							
Base Address Register 2							
Latency		Subordinate		Secondary		Primary	
Secondary status				IO Limit		IO Base	
Memory Limit				Memory Base			
Prefetch memory Limit				Prefetch memory Base			
Prefetch memory base upper 32							
Prefetch memory limit upper 32							
IO limit upper 16				IO Base upper 16			
						Capabilities	
Bridge control				Interrupt Pin		Interrupt	

图 9-5 PCI 配置空间布局（1 型头部）

另外，在 PCI 规范 2.2 中，还定义了一种头部类型，即 PCI-CardBus 桥接设备头部（2 型头部），在当前版本的 Hello China 中，没有涉及到，在此不作赘述。

### 9.4.3 配置空间关键字段的说明

下面对配置空间中，一些关键的字段进行简要说明，详细的字段含义，以及本文没有描述到的字段含义，请参考 PCI 总线规范。

#### 9.4.3.1 Device ID 和 Vendor ID

其中，Vendor ID 用来标识设备的制造厂商，而 Device ID 则标识特定的设备，Vendor ID 是由 PCI 规范组织统一分配的，因此不会重复（类似以太网接口卡的 MAC 地址），Device ID 则是由厂家自行分配的，一般情况下，Device ID 和厂家 ID 结合起来，可以精确识别一种设备。在 Hello China 的当前实现中，定义了下列结构，来对总线上的物理设备（并不一定是 PCI 接口设备）进行标识：

```
BEGIN_DEFINE_OBJECT(__IDENTIFIER)
    DWORD          dwBusType;
    union{
        struct{
            UCHAR    ucMask;
            WORD      wVendor;
            WORD      wDevice;
            DWORD     dwClass;
            UCHAR     ucHdrType;
            WORD      wReserved;
        }PCI_Identifier;
        struct{
            DWORD     dwDevice;
        }ISA_Identifier;
    }
END_DEFINE_OBJECT()
```

其中，总线类型（dwBusType）字段，确定该结构中 union 部分的具体含义。比如，如果总线类型为 BUS\_TYPE\_PCI，则按 PCI\_Identifier 结构解释该标识对象，于是在 PCI\_Identifier 结构中，wVendor 就是 Vendor ID，wDevice 则是 Device ID 字段。为了进一步区分设备，又把 Class code 字段和 Header Type 字段纳入进来（dwClass 成员，实际上只有高 24 比特有效，最低的 8 比特是 Revision ID），这样这些字段结合起来，可以作为 PCI 设备的唯一标识。有些情况下，可能需要一种“模糊”标识，比如，要标识一个特

定厂家（比如 Intel）的所有设备，这样就只需要 Vendor ID 字段就可以了，其它字段不需要给出。因此，ucMask 字段指明了 PCI\_Identifier 结构中哪个字段有效。比如，如果 ucMask 字段取值 IDENTIFIER\_MASK\_PCI\_VENDOR，则该结构中，只有 Vendor ID 字段有效，如果 wMask 字段取值 IDENTIFIER\_MASK\_PCI\_VENDOR | IDENTIFIER\_MASK\_PCI\_DEVICE，则 wVendor 和 wDevice 字段同时有效。

### 9.4.3.2 Class code

设备类代码，用来描述设备的功能。这个字段长 24 比特，进一步分成三部分：

- 1、最高的一个字节（偏移在 0x0B 处），是一个基类编号，粗略的描述了一个 PCI 设备的功能。比如，如果该字节是 0x01，则说明对应的 PCI 设备是一个大容量存储设备的控制器，如果该字节是 0x02，则对应的 PCI 设备是一个网络控制器，等等；
- 2、中间一个字节（偏移在 0x0A 处），是一个子类编号，进一步描述了设备的功能。比如，如果基类编号是 0x02，子类编号是 0x00，则说明对应的设备是以太网接口控制器，如果子类编号是 0x01，则说明对应的设备是令牌环接口控制器；
- 3、最后一个字节（偏移在 0x09 处），是一个编程接口字段，该字段可以用来进一步的对 PCI 设备的功能做出区分，一般情况下，这个字节可以保持为 0，也可以由 PCI 设备制造商根据自己的定义指定。

在对 PCI 设备进行枚举的时候，就是根据 Class code 字段，判断设备功能的。该字段也用来作为设备标识符的一部分。

### 9.4.3.3 Header type

头部类型。根据 PCI 规范的定义，对所有 PCI 设备，保留了 256 字节的配置空间，其中前 64 字节是预定义的，后续 192 字节，则根据设备的具体情况具体实现。对于预先定义的 64 字节首部，根据 PCI 规范（Revision 2.2），目前有三种情形：

- 1、普通的 PCI 设备，这类设备的头部组织，在本文中已经给出；
- 2、PCI-PCI 桥接设备（PCI-PCI 桥），这类设备的头部组织，在文中也给了出来；
- 3、PCI-CardBus 桥接设备（PCI-CardBus 桥），这类设备的组织，本文中没有给出。

其中，上述三种情形的预定义头部，其开始的 16 字节的组织结构，都是一样的，从 16 字节往后（17—64 字节），根据不同的头部类型，有不同的组织。Header type 字段，就是用来标识不同的头部的。该字段位于 PCI 配置空间偏移 0x0E 处，其中，该字段的最高比特（第七比特），用来标明当前的 PCI 设备是单功能设备（该比特为 0），还是多功能设备（该比特为 1）。0 到 6 比特则用来区分不同的 PCI 配置空间头部类型。如果这 7 比特为 0，则说明配置空间的头部是 0 型头部（普通的 PCI 设备头部），如果这 7 比特的

值为 1，说明配置空间的头部类型是 1，即 PCI-PCI 桥设备的配置头部，如果是 2，则是 PCI-CardBus 头部。因此，如果当前的 PCI 设备是一个多功能的 PCI-PCI 桥接设备，则 Header Type 的数值应该为 0x81，如果是一个普通的单功能 PCI 设备，则该字段的值是 0x00。

#### 9.4.3.4 Base Address Register

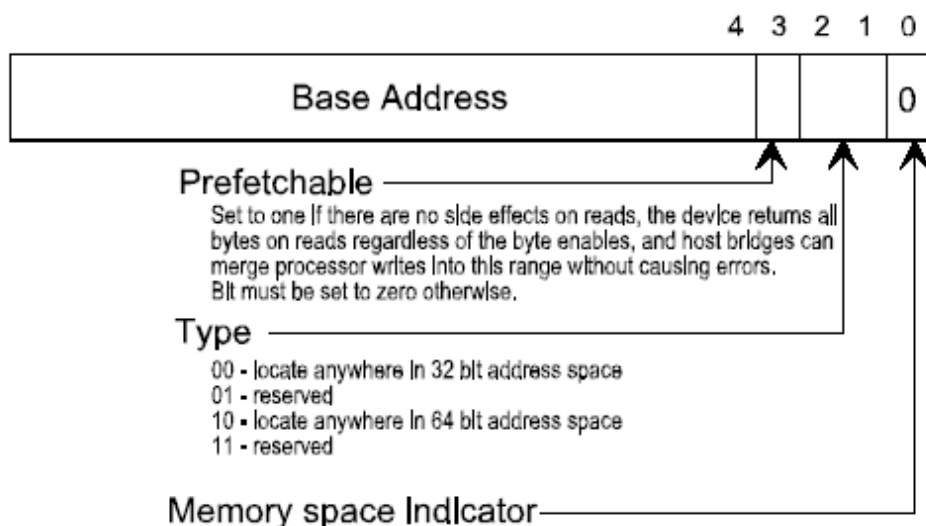
PCI 接口的硬件设备，一个最突出的特性，就是动态配置能力，即用于操作设备的 IO 端口、内存映射位置等，可以不用事先固定，而推迟到系统引导的时候，由软件根据系统资源情况，进行统一配置。而传统的基于 ISA 总线的设备，则在安装的时候，必须静态的完成 IO 端口分配，然后通过硬件跳线的方式，把硬件设备连接到计算机总线上，这样十分不方便，而且很容易出现冲突。

PCI 设备就是通过 Base Address Register 寄存器来实现动态配置的。配置软件把分配给 PCI 设备的 IO 端口范围，或内存映射地址范围，写入这些寄存器，这样后续对设备的操作，就可以通过写入的端口范围或寄存器进行。开始的时候，这些寄存器保持缺省值，并且设备所需要的 IO 端口范围大小（或内存映射区域的大小），也包含在这些寄存器中。对于普通的 PCI 设备，共有六个 Base Register Address 寄存器，这样一个普通的 PCI 设备，可以申请 6 个 IO 地址范围或内存映射空间。而对于 PCI-PCI 桥，则只有两个 Base Address Register，而且一般情况下，不使用这两个寄存器（有的桥设备也需要 IO 端口或内存映射空间，来进行操作，这时候这两个寄存器就得到了应用）。

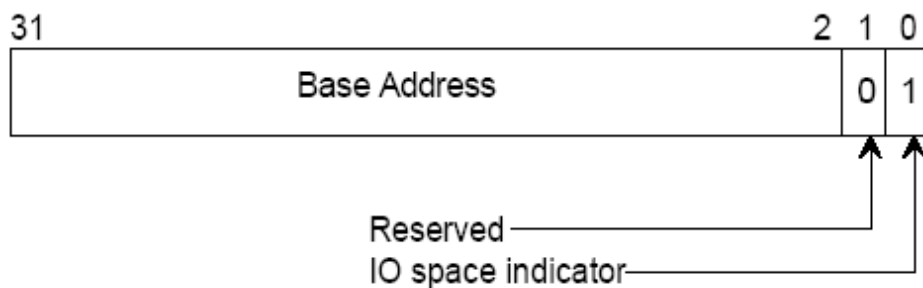
一般情况下，对 Base Address Register 的操作有两种：

- 1、获得 PCI 所需要的 IO 端口范围大小，或内存映射区域的数量；
- 2、向 Base Address Register 寄存器写入分配给设备的 IO 端口范围，或者内存区域。

在进行上述两种操作前，一个很重要的前提就是，如何得知 Base Address Register 寄存器是寄存了 IO 端口范围，还是内存映射空间。按照 PCI 的规范，每个 Base Address Register，必须符合下列结构：



**Figure 6-5: Base Address Register for Memory**



**Figure 6-6: Base Address Register for I/O**

图 9-6 Base Address Register 的结构

其中，每个寄存器的最低比特（比特 0）确定了该寄存器是映射到 IO 空间，还是内存空间。如果该比特为 0，则说明对应的寄存器映射到内存空间，如果为 1，则映射到 IO 端口空间。对于映射到内存空间的 Base Address Register，其次低的三个比特（比特 1、2、3）是控制比特，对所映射的内存区域特性进行描述。下列是这三个比特的取值，以及对

应的含义：

取值	含义
000	该寄存器映射到 32 位的内存空间内，而且映射的空间范围，是不可预取的。
001	保留
010	该寄存器映射到 64 位的内存空间，而且映射的空间范围，是不可预取的。
011	保留
100	该寄存器映射到 32 位的内存空间内，而且映射的空间范围，是可预取的。
101	保留
110	该寄存器映射到 64 位的内存空间内，而且映射的空间范围，是可预取的。
111	保留

表 9-1          三个控制比特的含义

对于“可预取（Prefetch）”，在这里进行进一步说明。许多情况下，为了提高效率，CPU 都实现了 cache 机制，即在 CPU 的本地，设置一个本地缓存，大小可以从 512K 到 2M 不等（有的甚至更大），有的 CPU，可能设置了 2 级，甚至 3 级 cache，这样每当 CPU 要读取一个内存单元的数据时，首先检索本地 cache，因为从 cache 中读取数据的速度，比从内存中读取数据的速度，快几个数量级。如果能够从 cache 中获得期望的内容（叫做 cache 命中），则就省略了从内存中读取相应内容花费的时间，提高了整体效率。当然，如果要读取的数据，不在 cache 中（叫做 cache 不命中），则 CPU 会到物理内存中读取相应的数据，在读取的同时，还把与该数据单元相邻的一块连续内存（比如，4K），读入

CPU 的本地 cache，这样后续 CPU 如果再读取同样的内存单元，或者跟该内存单元相邻的内存单元，就可以直接从 cache 中读取了。这种结构得以实现的核心基础，就是所谓的“局部性原理”。

当然，这种加快读取或写入操作的方式，对于通常的内存来说，是没有问题的，CPU 硬件可以保持数据的一致性。但对于映射到内存空间中的设备寄存器，则有的情况下，可能不适合这样的操作。比如，有的硬件设备的寄存器，读取之后，就进行复位，然后根据设备的状态，进一步设置位其它的值。这样的寄存器，读取的时机就十分关键了，在时刻 T1 的时候进行读取，跟在时刻 T2 的时候进行读取，得到的结果可能是不一样的。因此，如果有这类特征的硬件寄存器被映射到 CPU 的内存空间，那么就不适合于预先读取（或写入）。但有的设备寄存器，却没有这种限制。因此，为了兼顾这两种情况，在 Base Address Register 寄存器中，专门设置了比特位，来区分这两种情况。

在当前版本 Hello China 的实现中，对于可预取的设备映射内存，采取跟普通的物理内存一样的访问策略（可预读，写的时候直接写入，详细信息请参考“Hello China 的内存管理机制”一章），而对于不可预读的设备映射内存，则采取另外的禁止缓冲的访问策略，所有对该区域的内存读取操作，直接从内存中读取（而不经 over cache），在 IA32 硬件平台上，这可以通过设置合适的页面标志来实现。

而对于映射到 IO 端口空间的 Base Address Register，情况相对简单，最后一个比特为 1，第二个比特为 0，其它的比特，则保存了映射到的 IO 端口范围。

根据 PCI 规范（Revision 2.2）的描述，可以通过下列方式来获取 IO 端口范围大小或内存映射区域大小：

- 1、保存原来 Base Address Register 的值；
- 2、写入 0xFFFFFFFF 到对应的 Base Address Register；
- 3、再次读取对应的寄存器的值。

假设最后一次读取的内容为 size，则 IO 端口范围大小或内存映射范围大小可以按照下列方法计算：

- 1、清除掉保留的比特，对于映射到内存的寄存器，清除 0—3 比特，对于映射到 IO 端口的寄存器，清除 0 比特；
- 2、对经过上述步骤处理的结果，取反，然后加 1；
- 3、得到的 32 比特数值，就是内存映射范围的大小，如果是映射到 IO 端口范围内的大小，则忽略最高的 16 比特（16—31 比特）。

在当前版本的 Hello China 的实现中，实现上述计算的函数如下：

```
DWORD GetRangeSize(DWORD dwValue)
{
    DWORD dwTmp = dwValue;

    if(dwTmp & 0x00000001) //This range is IO port range.
    {
        dwTmp &= 0xFFFFFFF0; //Clear the lowest bit.
        dwTmp = ~dwTmp; //NOT calculation.
        dwTmp += 1;
        dwTmp &= 0xFFFF; //Reserve the low 16 bits only.
        return dwTmp;
    }
    else
    {
        dwTmp &= 0xFFFFFFF0;
        dwTmp = ~dwTmp;
        dwTmp += 1;
        return dwTmp;
    }
    return dwTmp;
}
```

计算出 Base Address Register 的尺寸后，就可以根据系统资源的情况，为这些寄存器分配资源了。分配好资源之后，再把分配的资源（IO 端口范围或内存映射范围）写入对应的寄存器，这样后续对设备的访问，就可以直接通过分配的 IO 端口，或内存映射区域进行。

最后，PCI 规范对普通的 PCI 设备，定义了六个 Base Address Register，这样从理论上说，一台 PCI 设备最多可以定义 6 个 IO 端口或内存映射空间，用于对设备的操作。但实际上用不了这么多的空间资源，比如，一般的设备，可能仅仅使用两个 Base Address Register 寄存器，一个寄存器用来指定 IO 端口范围，另外一个寄存器用来指定内存映射范围，这样剩余的没有使用的寄存器，就全部设置为 1。为了便于移植，按照 PCI 规范的建议，对于 PCI 设备，尽量采用内存映射区域，因此，在 Hello China 的设计中，对于

设备驱动程序的编写，尽量建议采用内存区域对设备进行控制。

## Interrupt Line 和 Interrupt Pin

这两个字段，给出了 PCI 设备的中断连接信息。其中，第一个字段，Interrupt Line，描述了设备的中断输入，与中断控制器的哪条引脚连接。比如，在 PC 机上，中断控制器一般采用两块 8259 芯片，向外提供 15 个中断输入，这样 Interrupt Line 字段就给出了当前的 PCI 设备，具体连接到 8259 的哪条引脚上。一般情况下，这个字段由 BIOS（或固件）填写，操作系统和设备驱动程序可以读取这个字段，从而获得设备的中断向量号。在 PC 机上，这个字段的值可以是 0—15 之间的任何数字，16—254 保留，如果是 255，则说明该设备没有中断输入。

另外一个字段，Interrupt Pin，则说明了对应的 PCI 设备的中断输入，连接到 PCI 总线的哪条中断输入上。PCI 总线有四条中断连接线—INTA、INTB、INTC、INTD，PCI 设备的中断输入，可以跟这四条连接线的任何一条连接（按照 PCI 规范定义，对于单功能设备，强烈建议连接到 INTA，对于多功能设备，则可以连接到 INTA—INTD 中的任何一条或几条），这四条中断连接线，又跟中断控制器（PC 机上的 8259 芯片）的四条中断输入引脚进行连接。Interrupt Pin 字段就指出了，对应的 PCI 设备的中断输入，跟 INTA—INTD 的哪条连接。如果该字段的值为 1，则是跟 INTA 连接，如果是 2，则是跟 INTB 连接，如果设备没有中断输入，则该字段设置为 0，5—255 是保留的设置。

在 Hello China 的当前实现中，对于 PCI 设备的这个字段，只进行读取操作，即采用 BIOS 分配的默认值，而不会另外分配新的数值。如果设备驱动程序要获取设备的中断向量号，建议调用 DeviceManager 对象的特定接口（函数）来获取，不建议直接读取 Interrupt Line 字段来获取。

## Primary、Secondary 和 Subordinate 总线号

Primary、Secondary 和 Subordinate 三个字段，目前只会在 01 型头部（PCI-PCI 桥接器设备）中出现。一般情况下，PCI-PCI 桥设备连接了两条 PCI 总线，一条总线是主总线（Primary BUS），就是 PCI-PCI 桥所在的总线，另外一条总线就叫做第二总线（Secondary BUS），这样 Primary 字段和 Secondary 字段，就是主总线号和第二总线号。

但是 PCI-PCI 桥设备的第二总线，还可能又连接了另外一个桥接器，另外一个桥接器又连接了第三条总线，...这样依次下去，会组成一个复杂的树形结构（按照 PCI 规范，最多可以有 256 条总线通过桥设备连接在一起），相对每个 PCI-PCI 桥来说，其 Secondary 总线所连接的所有总线，都称为该 PCI-PCI 桥的下级总线。这样 Subordinate 字段，就是

一个 PCI-PCI 桥设备所连接的下级总线中，总线号最大的那条总线的总线标识号。下图是一个由两个 PCI-PCI 桥设备组成的一个典型的硬件配置结构，总共有三条 PCI 总线：

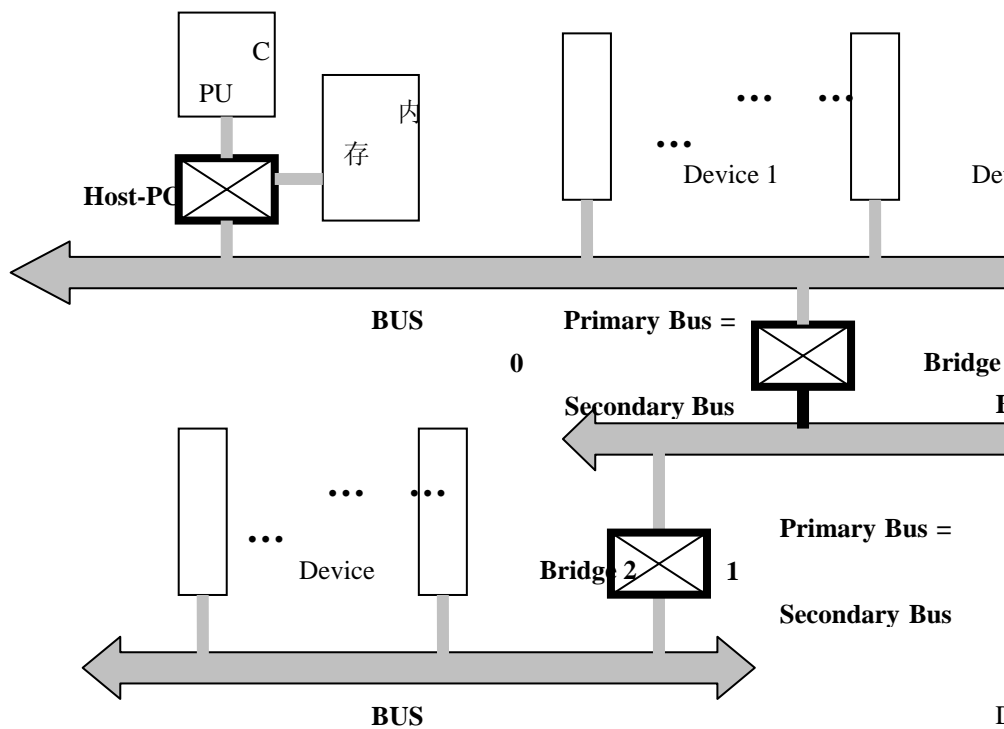


图 9-7 一个由三条 PCI 总线组成的总线结构

在这个硬件系统中，HOST-PCI 桥设备连接了总线 0 (BUS 0)，Bridge 1 作为一个 PCI 设备，出现在总线 0 上，因此 Bridge 1 桥接设备的主总线 (Primary BUS) 就是总线 0。Bridge 1 连接了总线 1，因此，其第二总线号 (Secondary 字段) 就是 1，同样地，在 BUS 1 上，Bridge 2 作为一个普通的 PCI 设备出现，因此，Bridge 2 的主总线号就是 1，Bridge 2 所连接的总线 2，就是其第二总线 (Bridge 2 的 Secondary BUS 是 2)。由于 Bridge 2 的第二总线上，没有再继续连接其它的桥接设备，因此 Bridge 2 的 Subordinate 就是 2，同样地，Bridge 1 的 Subordinate 字段的值应该是 2，因为在 Bridge 1 的下级总线上，最大的总线号是 2。

之所以在 PCI-PCI 桥设备中,记录 Subordinate 总线号,是为了访问的方便。一旦 CPU 发起一个配置空间的访问(该访问使用总线号、设备号和功能号来定位一个具体的 PCI 功能设备),PCI-PCI 桥设备把访问请求的目标总线号跟 Secondary 总线号和 Subordinate 总线号进行对比,如果发现请求的目标设备所在的总线号在 Secondary 和 Subordinate 之间,则该 PCI-PCI 桥会转发该访问,如果不在上述两个数字之间,则说明 CPU 访问的目标设备,不在该 PCI-PCI 桥以下的总线上,因此该 PCI-PCI 桥就不作任何处理。

这些字段都是在 PCI 总线初始化的时候填写的,在 PC 机上,这个初始化操作由 BIOS 软件完成。可靠起见,在 Hello China 的实现过程中,会重新对 PCI 总线进行初始化,不过在初始化的过程中,如果发现 PCI 设备已经配置,则接收 BIOS 的配置,不再进行更改。但如果有的 PCI 设备(或者 PCI-PCI 桥)没有被 BIOS 配置(很可能发生这种情况),则 Hello China 会配置这些没有经过 BIOS 配置的 PCI 设备。详细信息,请参考“PCI 总线初始化”一节。

## IO Base 和 IO Limit

这两个字段出现在 1 型预定义头部中(PCI-PCI 桥设备的头部),其含义与 PCI-PCI 桥设备的 Secondary 和 Subordinate 字段类似,用于过滤对连接在 PCI 桥设备的下级总线上的设备的读写。一旦 PCI 总线上的主设备发起一个读写请求(不是配置空间的读写,而是通过 IO 端口直接访问 PCI 设备),PCI 桥设备就会判断,请求的目标地址是否在 IO Base 和 IO Limit 限定的范围内。如果在限定的范围内,则 PCI 桥会向下级设备“转接”这个读写请求,否则,如果请求的地址不在 IO Base 和 IO Limit 范围内,则 PCI 桥设备不作任何动作。

其中,IO Base 字段给出了 IO 端口的起始地址,而 IO Limit 字段则给出了 IO 端口的范围大小,这样两者结合起来,就可以确定一个端口范围,对 PCI 设备的访问,凡是目标地址落在这个范围之内请求,都会被 PCI 桥设备“转接”。因此,IO Base 和 IO Limit 所确定的端口范围,应该是 PCI 桥接器所有下级 PCI 设备的 IO 端口范围的“并集”,即 IO Base 确定的端口地址,应该是所有该 PCI 桥下面的 PCI 设备的端口范围下界的最小值,而 IO Base + IO Limit 则是所有该 PCI 桥下面的 PCI 设备的端口范围上界的最大值。

需要注意的是,IO Base 和 IO Limit 都是以 256 字节为边界的,即如果 IO Base 取值为 0xAB,则确定的 IO 端口范围的下界是 0xAB00,同样地,IO Limit 也是以 256 字节为边界的,如果 IO Limit 的取值为 0x01,则确定的端口范围实际上是 0x0100,即 256 字节。这样如果 IO Base 取值为 0xAB00,IO Limit 取值为 0x01,则实际确定的 IO 端口范围是 [0xAB00, 0xABFF]。

这样 IO Base 和 IO Limit 就可以确定 16 比特的 IO 端口范围，这在 Intel 的硬件平台上，是足够了，但有些 CPU 的端口范围，可能是 32 位的，因此，1 型头部中的 IO Base Uper 16 字段和 IO Limit Uper 16 字段，就作为 32 位 IO 端口范围的扩展，跟 IO Base 和 IO Limit 结合起来，共同确定一个 32 比特的 IO 端口范围。在 Hello China 目前版本的实现中，目标 CPU 是 IA32，因此没有考虑这种 32 位端口的情况，但 Hello China 的相关数据结构的设计，却充分考虑了这种存在，将来如果要实现支持 32 位端口范围的版本，将会十分容易。

Memory base 和 Memory Limit

这两个字段也是出现在 1 型预定义头部中（PCI-PCI 桥设备头部），其含义与 IO Base 和 IO Limit 相同，用于完成对当前桥设备下的 PCI 设备的内存映射区域的访问过滤。

Prefetch memory base 和 Prefetch memory limit

这两个字段也是出现在 1 型预定义头部中（PCI-PCI 桥设备头部），其含义与 IO Base 和 IO Limit 相同，用于完成对当前桥设备下的 PCI 设备的可预取内存映射区域的访问过滤。

PCI 配置空间的读取与设置

在个人计算机（PC）上，HOST-PCI 桥接器（北桥）直接连接了 CPU 和内存，对于 PCI 上设备配置空间的读取，也是通过 HOST-PCI 桥接器进行的，按照 PCI 规范的定义，HOST-PCI 桥接器使用端口范围 0xCF8—0xCFF，这样就可以通过读写这些端口号，对 PCI 设备的配置空间进行读取或写入操作。在读写 PCI 设备的配置空间的时候，分两步进行：1、首先向预留端口 CF8（十六进制），输出要读取的 PCI 设备标识信息（包括 PCI 总线号、设备号、功能号等），以及要读取的配置空间的偏移，所有这些信息，组织成一个 32 比特的长字，如下：

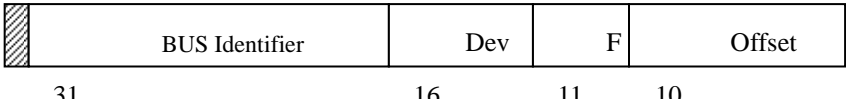


图 9-8 PCI 总线操作命令字结构

其中，第 31 比特固定为 1，16 到 30 比特是总线标识符，11 到 15 比特则是设备号，8 到 10 比特是功能号（一个 PCI 设备，最多可以支持 8 种不同的功能），最后 8 比特则是要读取的配置空间数据的偏移。比如，一块 PCI 接口的网络接口卡，位于 PCI 总线 1 上，设备 ID 是 8，功能 ID 为 0（只有一种功能），要读取配置空间的第一个双字（4byte），则首先需要向 CF8 端口输出 80014000（十六进制）；

2、然后读取 CFC 端口，以双字的方式，就可以获得上述设置的配置空间内相应的内容。

对于直接连接在 HOST-PCI 桥接器的 PCI 总线，总线号总是 0。比如，要读取 PCI 设备配置空间中的 Vendor ID 和 Device ID 字段，可以如下操作（假设设备位于总线 0、设备号 8、功能号 0）：

```
mov dx,CF8
out dx,80004000
mov dx,CFC
in eax,dx
```

对于 PCI 设备配置空间的写入操作，与此类似，首先向端口 CF8 写入目标地址（包含总线号、设备号、功能号、偏移等），然后向 CFC 端口写入一个长字。比如，要写入 PCI 设备配置空间的 Interrupt Line 字段，可用这样进行：

```
mov dx,CF8
out dx,8000403C
mov dx,CFC
in eax,dx
and eax,FFFFFFF0    ;Clear the lowest 4 bits.
add eax,8
out dx,eax
```

这样就把“8”写入了位于 PCI 总线 0 上的设备 8、功能 0 的配置空间中的 Interrupt Line 寄存器内。需要注意的是，一般建议以 32 比特为单位，进行读写，因此，在进行写入操作的时候，如果写入的字段小于 4 字节（上面 Interrupt Line 只有 8 比特），则首先读出四个字节，然后修改要写入的部分，再把四个字节整体写入配置空间。

## PCI 总线驱动程序的实现

在当前版本 Hello China 的实现中，实现了 PCI 总线和 ISA 总线两种常见个人计算机总线的驱动程序，也就是说，目前版本的 Hello China，内嵌支持 PCI 和 ISA 总线。所谓内嵌支持，只的是在操作系统核心中，已经集成了这类总线的驱动程序，不需要额外的总线驱动程序。因此，如果要进一步支持其它的总线类型，比如 USB（在当前的 Hello China 版本上），就需要专门编写一个 USB 总线驱动程序，并跟操作系统一起加载。在后续版本中，Hello China 会进一步支持更广泛的总线类型，比如 USB 等。

按照目前的设计，系统总线驱动程序的结构，没有纳入普通的设备驱动程序体系结构中，即总线驱动程序遵循单独的编写规范。为了实现上的简便，总线驱动程序只需输出一个函数 XXXBusDriver 即可。比如，对于 PCI 总线驱动程序，该函数的原型如下：

```
BOOL PciBusDriver(__DEVICE_MANAGER* lpDevMgr);
```

其中，DeviceManager 对象作为该函数的参数（DeviceManager 是一个全局对象）。

这个函数在 DeviceManager 初始化的时候被调用，如果初始化成功，则返回 TRUE，否则返回 FALSE，如果返回 FALSE，有可能导致操作系统引导失败。

当前版本的实现中，PCI 总线驱动程序完成下列工作：

- 1、探测 PCI 总线是否存在，如果不存在，返回 FALSE；
- 2、如果存在，对 PCI 总线进行枚举，包括枚举所有的 PCI 设备及下级总线；
- 3、如果定义了 CONFIG\_PCI，则对 PCI 设备进行配置（分配内存映射区域或 IO 端口范围），否则仅仅完成设备信息收集工作（这时候设备的配置，依靠 BIOS 完成）。

下面对上述过程进行详细说明。

### 探测 PCI 总线是否存在

对于 PCI 总线的探测，目前做的十分简单：

- 1、向端口 0xCF8 写入 0x80000000（总线 0、设备 0、功能 0、配置空间偏移 0）；
- 2、读取 0xCFC 端口（双字）；
- 3、如果读取的结果是 0xFFFFFFFF，则说明系统中不存在 PCI 总线，否则，认为 PCI 总线存在。

上述判断的依据是，如果 PCI 总线存在，那么必然会有一个 HOST-PCI 桥设备存在，

而 HOST-PCI 桥设备一般是作为 PCI 总线的第 0 个设备（0 号功能），这样上述读取操作实际上是读取了 HOST-PCI 桥的 Vender ID 和 Device ID。按照 PCI 规范，如果对应的设备不存在，则返回 0xFFFFFFFF。因此，上述操作可以判断 PCI 总线的存在。

在大多数情况下，上述判断可以很好的工作，但是有一种情况必须考虑，那就是 PCI 总线不存在，而恰好有另外一个设备，使用了端口 0xCF8 到 0xCFF，这样上述读取操作，获得的结果可能不是 0xFFFFFFFF，但实际上 PCI 总线是不存在的。这种情况极少发生，实际上，在现代的计算机体系结构中，PCI 总线是必不可少的，这也是把 PCI 探测做的很简单的主要原因。

对于 PCI 总线的探测，实现代码如下：

```
static BOOL PciBusProbe()
{
    DWORD    dwInit  = 0x80000000;
    __outd(0xCF8,dwInit);
    dwInit = __ind(0xCFC);
    if(0xFFFFFFFF == dwInit)    //The HOST-PCI bridge does not exist.
        return FALSE;
    return TRUE;
}
```

## 对普通 PCI 设备进行枚举

一旦探测到 PCI 总线的存在，就需要对 PCI 总线进行枚举，以收集连接到该总线的 PCI 设备信息。目前情况下，对 PCI 设备的枚举，采用下列算法：

- 1、首先从 DeviceManager 的 SystemBus 数组中，找到一个空闲的（尚未被占用的）总线对象（\_\_SYSTEM\_BUS），设置该总线对象的总线类型为 BUS\_TYPE\_PCI；
- 2、从当前总线上第 0 号设备、第 0 号功能开始，依次探测对应的设备（或功能）是否存在；
- 3、如果设备存在，则创建一个 \_\_PHYSICAL\_DEVICE 对象，把设备相关的信息（所占用的资源、中断向量号、设备类型等）填写到 \_\_PHYSICAL\_DEVICE 对象中，然后把该物理设备对象插入设备列表（由总线对象维护）；
- 4、探测完毕之后，再对该总线上的 PCI-PCI 桥设备，进行初步配置，然后对桥设备的下级总线，完成同样的探测。

上述过程的实现方式如下：

```

DWORD PciScanBus(__DEVICE_MANAGER* lpDeviceMgr, __PHYSICAL_DEVICE*
lpBridge, DWORD dwBusNum)
{
    DWORD                dwLoop                = 0;
    DWORD                dwFlags                = 0;
    PCI_DEVICE_INFO*     lpBusInfo              = NULL;
    __PHYSICAL_DEVICE*   lpDevice              = NULL;
    DWORD                dwSubNum               = dwBusNum;

    if(NULL == lpDeviceMgr)    //Invalidate paremter.
        return MAX_DWORD_VALUE;

    __ENTER_CRITICAL_SECTION(NULL, dwFlags);
    for(dwLoop = 0; dwLoop < MAX_BUS_NUM; dwLoop++) //Find a empty bus
object.
    {
        if(lpDeviceMgr->SystemBus[dwLoop].dwBusType == BUS_TYPE_NULL)
            break;
    }
    if(MAX_BUS_NUM == dwLoop)    //Can not find a empty system bus object.
    {
        __LEAVE_CRITICAL_SECTION(NULL, dwFlags);
        return MAX_DWORD_VALUE;
    }
    lpDeviceMgr->SystemBus[dwLoop].dwBusType = BUS_TYPE_PCI;    //Set bus
type.

    lpDeviceMgr->SystemBus[dwLoop].lpHomeBridge = lpBridge;
    lpDeviceMgr->SystemBus[dwLoop].dwBusNum    = dwBusNum;
    if(lpBridge)
    {
        lpDeviceMgr->SystemBus[dwLoop].lpParentBus = lpBridge->lpHomeBus;
    }
}

```

```

        lpBridge->lpChildBus = &lpDeviceMgr->SystemBus[dwLoop];
    }

    PciScanDevices(&lpDeviceMgr->SystemBus[dwLoop]);    //Scan devices on this
bus.

    lpDevice = lpDeviceMgr->SystemBus[dwLoop].lpDevListHdr;
    while(lpDevice)
    {
        if(PCI_DEVICE_TYPE_BRIDGE ==
(PCI_DEVICE_INFO*)lpDevice->lpPrivateInfo->dwDeviceType)    //This is a PCI-PCI
bridge.

            dwSubNum = PciScanBus(lpDeviceMgr,lpDevice,++dwBusNum);

#ifdef PCI_CONFIG    //Configure the PCI bridge device.
            SetPciBridgeSubNum(lpDevice,dwSubNum);    //Set Subordinate bus number
of the bridge.
#endif

        lpDevice = lpDevice->lpNext;
    }

    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return dwSubNum;
}

```

上述过程是一个递归过程（黑色字体标出的代码），总体思路是，首先对当前总线上的 PCI 设备，进行搜索（PciScanDevices 函数完成），并针对每个存在的 PCI 设备，创建一个物理设备对象（\_\_PHYSICAL\_DEVICE），添加到当前总线的设备列表中。然后再次遍历当前总线的所有 PCI 设备，如果发现一个 PCI 设备是一个 PCI-PCI 桥，则进一步扫描该桥接设备的二级（Secondary）总线。

如果当前总线上没有任何 PCI-PCI 桥设备，则该函数返回当前总线号，如果有 PCI-PCI 桥设备，则该函数返回的数值，就是该桥设备下，所有总线中的最大的总线号，因此，该函数的返回值，就可以作为上级总线的 Subordinate 值。

为了对 PCI 设备进行更进一步的描述，定义下列对象：

```
BEGIN_DEFINE_OBJECT(__PCI_DEVICE_INFO)
    DWORD                dwDeviceType;
    DWORD                DeviceNum : 5;
    DWORD                FunctionNum : 3;
    DWORD                dwClassCode;
    UCHAR                ucPrimary;
    UCHAR                ucSecondary;
    UCHAR                ucSubordinate;
END_DEFINE_OBJECT()
```

其中，dwDeviceType 字段，给出了该对象的类型。目前情况下，定义了下列三个取值：

```
#define PCI_DEVICE_TYPE_BRIDGE    0x00000001
#define PCI_DEVICE_TYPE_NORMAL    0x00000002
#define PCI_DEVICE_TYPE_CARDBUS   0x00000004
#define PCI_DEVICE_TYPE_EMPTY     0x00000000
```

这个字段的设置，是根据 HdrType 字段（PCI 设备配置空间）进行的。在对 PCI 设备进行枚举的时候，每当发现一个设备，就创建这样一个对象，并连接在 \_\_PHYSICAL\_DEVICE 对象中（\_\_PHYSICAL\_DEVICE 的 lpPrivateInfo 字段，保存了这个对象的指针）。

PciScanDevices 函数，完成对当前总线上所有设备的枚举和配置工作，该函数实现如下：

```
VOID PciScanDevices(__SYSTEM_BUS* lpSysBus)
{
    __PHYSICAL_DEVICE*    lpDevice = NULL;
    DWORD                dwFlags;
    DWORD                dwConfigAddr = 0x80000000;
    DWORD                dwTmp = 0L;
```

```

if(NULL == lpSysBus)
    return;

dwConfigAddr += (lpSysBus->dwBusNum << 16);
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
for(DWORD dwLoop = 0;dwLoop < 0x100;dwLoop++) //Scan all devices and
functions.
{
    dwConfigAddr &= 0xFFFF0000;    //Clear it.
    dwConfigAddr += (dwLoop << 8);
    __outd(0xCF8,dwConfigAddr);
    dwTmp = __ind(0xCFC);
    if(0xFFFFFFFF == dwTmp)        //The device(functions) is not exist.
        continue;
    //
    //Now, a PCI devices is found.
    //
    PciAddDevice(dwTmp,lpSysBus)
}
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
return;
}

```

这个函数从功能 0、设备 0 开始，依次检测所有 PCI 总线上可能出现的设备（或功能），一旦检测到一个存在的设备或功能，就调用 `PciAddDevice` 函数，该函数创建一个物理设备对象（`__PHYSICAL_DEVICE`），初始化，并插入到系统总线的设备列表中。需要注意的是，该函数（`PciAddDevice`）会根据头部类型的不同，分别处理普通的 PCI 设备和 PCI-PCI 桥设备，如果定义了 `PCI_CONFIG`，则该函数会重新配置物理设备（重新分配 IO 端口范围、内存映射空间等），对于 PCI-PCI 桥，该函数仅仅完成 `Primary` 和 `Secondary` 字段的配置工作，`Subordinate` 字段则一直留到 `PciScanBus` 函数中配置。该函数的实现如下：

```

VOID PciAddDevice(DWORD dwConfigReg,__SYSTEM_BUS* lpSysBus)
{

```

```

__PCI_DEVICE_INFO*          lpDevInfo          = NULL;
__PHYSICAL_DEVICE*          lpPhyDev           = NULL;
DWORD                       dwFlags            = 0L;
BOOL                         bResult            = FALSE;
DWORD                       dwLoop             = 0L;
DWORD                       dwTmp              = 0L;

if((NULL == lpSysBus) || (0 == dwConfigReg)) //Invalidate parameters.
    return;

lpPhyDev =
KMemAlloc(KMEM_SIZE_TYPE_ANY, sizeof(__PHYSICAL_DEVICE));
if(NULL == lpPhyDev) //Can not allocate memory for physical device object.
    goto __TERMINAL;

lpDevInfo =
KMemAlloc(KMEM_SIZE_TYPE_ANY, sizeof(__PCI_DEVICE_INFO));
if(NULL == lpDevInfo) //Failed to allocate memory for device information
object.
    goto __TERMINAL;

lpPhyDev->lpPrivateInfo = (LPVOID)lpDevInfo;
lpDevInfo->FunctionNum = (dwConfigReg >> 8) & 0x00000007;
lpDevInfo->DeviceNum = (dwConfigReg >> 11) & 0x0000001F;

//
//The following code initializes the physical device by reading configuration space.
//
dwConfigReg &= 0xFFFFF00; //Clear lowest 8 bits.
dwConfigReg += PCI_CONFIG_OFFSET_ID;
__outd(0xCF8, dwConfigReg);
dwTmp = __ind(0xCFC);

lpPhyDev->DevId.dwBusType = BUS_TYPE_PCI;
lpPhyDev->DevId.PCI_Identifier.ucMask = PCI_IDENTIFIER_MASK_ALL;
lpPhyDev->DevId.PCI_Identifier.wVendor = (LOWORD)(dwTmp);

```

```
lpPhyDev->DevId.PCI_Identifier.wDevice = (LOWORD)(dwTmp >> 16);
```

```
dwConfigReg &= 0xFFFFFFFF00
```

```
dwConfigReg += PCI_CONFIG_OFFSET_CLASSCODE;
```

```
__outd(0xCF8,dwConfigReg);
```

```
dwTmp = __ind(0xCFC);
```

```
lpPhyDev->DevId.PCI_Identifier.dwClass = dwTmp;
```

```
lpDevInfo->dwClassCode = dwTmp;    //Also save this information to information
```

object.

```
dwConfigReg &= 0xFFFFFFFF00;
```

```
dwConfigReg += PCI_CONFIG_OFFSET_HEADERTYPE;
```

```
__outd(0xCF8,dwConfigReg);
```

```
dwTmp = __ind(0xCFC);
```

```
lpPhyDev->DevId.PCI_Identifier.ucHdrType = (LOBYTE(LOWORD(dwTmp >>
```

16)));

```
//
```

```
//The following code initializes the resource information of physical device object.
```

```
//
```

```
switch(lpPhyDev->DevId.PCI_Identifier.ucHdrType & 0x7F)
```

```
{
```

```
    case 0:    //The header type is 0.
```

```
        lpDevInfo->dwDeviceType = PCI_DEVICE_TYPE_NORMAL;
```

```
        dwConfigReg &= 0xFFFFFFFF00;
```

```
        PciFillDevResource(dwConfigReg,lpPhyDev);
```

```
        bResult = TRUE;
```

```
        break;
```

```
    case 1:    //The header type is 1.
```

```
        lpDevInfo->dwDeviceType = PCI_DEVICE_TYPE_BRIDGE;
```

```
        dwConfigReg& = 0xFFFFFFFF00;
```

```
        PciFillBridgeResource(dwConfigReg,lpPhyDev);
```

```

        bResult = TRUE;
        break;
    default:    //In current version,only 0 and 1 header type are supported.
        break;
}

//
//Once finished initializing the physical device object, we insert it into device list of
the bus.
//
lpPhyDev->lpHomeBus = lpSysBus;
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
lpPhyDev->lpNext = lpSysBus->lpDevListHdr;
lpSysBus->lpDevListHdr = lpPhyDev;
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);

__TERMINAL:
if(!bResult)    //Some errors occurred.
{
    if(lpPhyDev)
        KMemFree((LPVOID)lpPhyDev,KMEM_SIZE_TYPE_ANY,0L);
    if(lpDevInfo)
        KMemFree((LPVOID)lpDevInfo,KMEM_SIZE_TYPE_ANY,0L);
}
return;
}

```

可以看出，上述函数（PciAddDevice）首先创建一个物理设备对象（\_\_PHYSICAL\_DEVICE）和一个 PCI 设备信息对象（\_\_PCI\_DEVICE\_INFO），然后把 PCI 设备信息对象连接到物理对象中，并根据传递过来的参数，初始化设备信息对象的部分成员。

接下来，该函数读取设备的配置空间，根据读取的结果，初始化设备的 ID、头部类型等字段，然后根据头部类型，进一步判断是 1 型头部还是 0 型头部，对于 0 型头部（普通的 PCI 设备），则调用 PciFillDevResource 函数，完成设备所需资源的填充，对于 1 型

头部（PCI-PCI 桥设备头部），则调用 `PciFillBridgeResource` 函数，完成 PCI-PCI 桥设备的资源填充。

首先看 `PciFillDevResource` 函数，该函数实现如下：

```
static VOID PciFillDevResource(DWORD dwConfigReg, __PHYSICAL_DEVICE*
lpPhyDev)
{
    DWORD          dwLoop = 0L;
    DWORD          dwTmp = 0L;
    DWORD          dwOrg  = 0L;
    DWORD          dwSize = 0L;
    DWORD          dwIndex = 0L;

    if((NULL == lpPhyDev) || (0 == dwConfigReg))    //Invalid parameters.
        return;
    dwConfigReg &= 0xFFFFF00;    //Clear the offset part.
    dwConfigReg += PCI_CONFIG_OFFSET_BASEADDR;

    for(dwLoop = 0; dwLoop < 6; dwLoop++)
    {
        __outd(0xCF8, dwConfigReg);
        dwOrg = __ind(0xCFC);
        __outd(0xCF8, 0xFFFFFFFF);
        dwTmp = __ind(0xCFC);
        if((0 == dwTmp) || (0xFFFFFFFF == dwTmp)) //The base address register is not
used.
        {
            dwConfigReg += 4;    //Prepare to read the next base address register.
            __outd(0xCF8, dwOrg); //Restore original value.
            continue;
        }

        __outd(0xCF8, dwOrg);    //Restore original value.
```

```

        if(dwOrg & 0x00000001)    //IO Port range.
        {
            dwSize = GetRange(dwTmp);
            dwOrg &= 0xFFFFFFF0;    //Clear the lowest bit.
            lpPhyDev->Resource[dwIndex].lpNext = NULL;
            lpPhyDev->Resource[dwIndex].lpPrev = NULL;
            lpPhyDev->Resource[dwIndex].dwResType = RESOURCE_TYPE_IO;
            lpPhyDev->Resource[dwIndex].IOPort.wStartPort = LOWORD(dwOrg);
            lpPhyDev->Resource[dwIndex].IOPort.wEndPort =
                LOWORD(dwOrg) + dwSize - 1;
        }
        else
        {
            dwSize = GetRange(dwTmp);
            dwOrg &= 0xFFFFFFF0;    //Clear the lowest 4 bits.
            lpPhyDev->Resource[dwIndex].lpNext = NULL;
            lpPhyDev->Resource[dwIndex].lpPrev = NULL;
            lpPhyDev->Resource[dwIndex].dwResType = RESOURCE_TYPE_MEMORY;
            lpPhyDev->Resource[dwIndex].lpStartAddr = (LPVOID)dwOrg;
            lpPhyDev->Resource[dwIndex].lpEndAddr = (LPVOID)(dwOrg + dwSize
- 1);
        }
        dwIndex ++;
        dwConfigReg += 4;
    }

    dwConfigReg &= 0xFFFFFFF0;
    dwConfigReg += PCI_CONFIG_OFFSET_INTERRUPT;
    __outd(0xCF8,dwConfigReg);
    dwTmp = __ind(0xCFC);
    if(0xFF == LOBYTE(LOWORD(dwTmp)))    //No interrupt vector is present.
        return;

    lpPhyDev->Resource[dwIndex].dwResType = RESOURCE_TYPE_INTERRUPT;

```

```

        lpPhyDev->Resource[dwIndex].ucVector = LOBYTE(LOWORD(dwTmp));
    return;
}

```

上述函数十分简单，仅仅是一个循环，把设备所有的 **Base Address Register** 寄存器读取一遍（按照本文概述中介绍的方法），然后把设备的资源信息，存储到物理设备对象的 **Resource** 数组中。需要注意的是，在计算 **IO** 端口区间范围或内存映射区域大小的时候，调用了 **GetRange** 函数。最后，该函数读取设备的中断向量信息，并填充到资源数组中。

对于 **PCI-PCI** 桥设备，其资源设置方式，与普通的 **PCI** 设备有所不同，如下所示：

```

static VOID PciFillBridgeResource(DWORD dwConfigReg,__PHYSICAL_DEVICE*
lpPhyDev)
{

}

```

## 配置 **PCI** 桥接设备

对于 **PCI** 桥设备的配置，与普通的 **PCI** 设备不同，需要单独考虑。主要原因是，在 **PCI-PCI** 桥设备的配置空间中，有几个字段，需要根据该总线的二级总线上的设备配置情况，进行配置，这几个字段是：

- 1、**IO Base** 和 **IO Limit** 字段，这两个字段，应该是该 **PCI-PCI** 桥设备的二级总线及更下级总线上，所有分配的 **IO** 端口范围的并集，因此，为了对这两个字段进行配置，需要简缩当前总线上的所有 **PCI** 设备，并计算他们的并集；
- 2、**Memory Base** 和 **Memory Limit** 字段，这两个字段是该 **PCI** 桥设备的二级总线及更下级总线上所有 **PCI** 设备内存映射范围的并集，与 **IO Base** 和 **IO Limit** 含义类似，因此，要配置这两个字段，必须搜索本 **PCI** 总线及其下级总线的所有设备；
- 3、**Prefetch Memory Base** 和 **Prefetch Memory Limit** 字段，这两个字段与上述四个字段含义类似，需要采用相同的方式进行配置；
- 4、**Subordinate** 字段，这个字段描述了该 **PCI-PCI** 桥设备的所有下级总线中，总线号最大的总线，因此，对于该字段的配置，也需要搜索整个 **PCI** 树（以该 **PCI-PCI** 桥为根）。

在 **Hello China** 当前版本的实现中，对上述几个字段的配置，除了 **Subordinate** 字段，都推迟到设备初始化完成之后进行，而对于 **Subordinate** 字段，则在扫描总线的时候，就

## 110 自己动手写嵌入式操作系统

已经做了配置（参考 `PciScanBus` 函数的实现）。



## 第十章 驱动程序管理框架

*(The architecture of Hello China,loadable  
module and device driver)*

## 10.1 设备驱动程序管理框架

### 10.1.1 概述

设备管理是操作系统最核心的管理任务之一，实际上，在一个典型的成熟的操作系统当中，设备管理部分的实现代码，可能占整个操作系统实现代码的一半，可见设备管理部分的复杂性和重要性。

由于硬件设备的多样性，操作系统不可能对每种硬件都自己直接驱动，而是采用一种分层的结构，即由硬件设备制造商（Vendor）提供针对特定硬件的设备驱动程序（Driver），设备驱动程序安装在计算机上，由操作系统调用设备驱动程序来控制硬件。这样就可以避免各种各样的麻烦，保持操作系统良好的可扩充性。

因此，一般来说，操作系统面临的是设备驱动程序，而设备驱动程序再进一步对设备进行驱动或管理，操作系统一般不直接面向具体的硬件设备，要实现这个功能，那么操作系统必须提供一个标准的接口给设备驱动程序，以便设备驱动程序可以向上跟操作系统交互，操作系统也根据自己提供的这个接口，来调用设备驱动程序的一些功能函数，来简介的操纵硬件。

另外，在设备管理过程当中，还有一些问题要解决，诸如：

- 1、**如何标识和命名一个设备**，如果一个设备得到良好的命名，那么用户程序就可以直接根据设备名字来打开设备，进而请求设备提供的服务，相反，如果无法正常的命名设备，那么用户将无法通知操作系统，自己想操纵哪个设备，因此，设备的命名机制，实际上是用户应用程序跟操作系统之间的接口；
- 2、**如何处理设备中断**，一般情况下，设备是通过中断的方式来通知操作系统，特定的事件已经发生（有些情况下，是被动通知的，即操作系统需要被动的查询设备），操作系统在收到中断通知之后，会根据中断号，调用合适的中断处理程序，一般情况下，中断处理程序是在设备驱动程序中的，因此，设备如何把自己特定的中断处理程序注册到操作系统中，也是需要解决的一个问题；
- 3、**硬件资源的分配问题**，所谓硬件资源，即包括中断号、输入/输出端口号、DMA 通道、内存映射区域在内的系统资源。这些系统资源的分配，可以有两种方式：其一，静态分配，由人工的方式，手工设置设备所使用的资源情况，然后这些设置保存在一个配置文件中，设备驱动程序分析这个配置文件，获得资源分配情况，或者由设备管理单元通知设备驱动程序，这种方式的好处是，不需要操作系统核心做额外的事情，所有硬件资源都由计算机管理者手工分配，但有一个问题，就是操作起来比较麻烦，尤其是对于初级用户，可能会是一个很大的挑战，而且有可能出现资源冲突的现象，比如，由于错误的分配，两个硬件设备占用了同一个中断。另外一种分配方式，就是动态分配，操作系统核心在加载的时候，通过某种总线协议，动态的检测存在总线上的设备，并集中分配系统资源，然后通过某种协议通知设备驱动程序，显然，这种方式是一种理想的方式，不需要用户过多的干预，而且由于集中分配，

避免了资源冲突问题。但这种方式需要系统总线的支持，有些情况下，系统总线是不支持自动发现设备的，这时候就需要通过手工的方式，静态分配系统资源；

- 4、**用户线程对设备的调用问题**，用户如何通过一种统一的接口，调用各种不同的设备驱动程序提供的服务，也是设备管理模块在设计过程中，需要重点考虑的问题之一。一般情况下，不可能为每种不同的设备，都提供一套特定的调用接口，由用户线程调用，而是由设备管理框架提供一套统一的接口，用户线程通过这套统一的接口，调用不同的设备驱动程序提供的服务。

一般情况下，一个操作系统的设备管理部件，需要通盘考虑上述问题，并针对上述问题，提供合理的解决方案，来实现一个可以真正使用的设备管理部件。

Hello China 的设计过程中，对于设备管理部件，在设计初期，也充分考虑了上述问题，并采取了合适的手段，对上述问题进行了解决。在本文中，我们就 Hello China 的设备管理模块，进行详细的描述，包括其总体框架模型、每个组成对象的详细设计、各模块之间的接口等。

在本文中，我们把 Hello China 实现的设备管理体系成为“设备管理框架”，从名字上看出，该软件模块其实是一个框架（Frame），因为该软件模块定义了一些标准接口，这些接口的具体实现，则是在用户编制的程序中实现的，比如设备驱动程序等，设备管理体系仅仅根据用户的请求，调用适当的标准接口，符合框架的概念。

## 10.1.2 设备管理器和 IO 管理器

### 10.1.2.1 通用的设备管理机制

在正式描述 Hello China 的设备管理框架前，有必要对通用的操作系统（比如 Windows 系列、Linux 系列等）的设备管理机制进行描述，以便于读者理解这些通用操作系统的设备管理机制，以此为基础，从而可以更好的理解 Hello China 的设备管理框架。若读者对通用的操作系统设备管理机制非常熟悉，可跳过此节，直接阅读下一节。

为了对设备进行管理，操作系统必须充分收集系统的硬件配置信息，并建立相应的数据库。一般情况下，这个收集设备硬件信息、建立设备信息数据库的过程，是在操作系统启动的过程中进行的。在操作系统启动的过程中，会首先从系统总线开始探测，比如，针对总线，操作系统引导代码会读取适当的端口（比如，CF8H 或 CF0H），根据读取结果，来判断对应的 PCI 总线是否存在。若存在，则跳转到 PCI 总线驱动程序代码，PCI 总线驱动程序代码完成 PCI 设备的枚举和检测。当然，对于 ISA 等总线类型，也执行类似的操作。

为了维护设备硬件信息，操作系统一般维护一个特定格式的数据库，在操作系统加载期间，由初始化代码检测系统硬件配置，根据检测的信息，填写这个数据库。比如，针对 PCI 总线，PCI 总线驱动程序会依次枚举总线上的设备，并读取设备配置信息（PCI 相关的详细信息，请参考“系统总线管理”一章），然后针对每个系统中存在的设备，都会在硬件信息数据库中创建相应的对象（数据结构），然后使用读取的配置信息填充这个对象，这样针对系统中的每条总线，都会进行这样一个检测操作，最终的结果是，操作系统把所有的系统硬件信息都进行了收集，并存放到数据库中进行统一管理。比如，下列图示表示了一个典型的操作系统的硬件数据信息库：

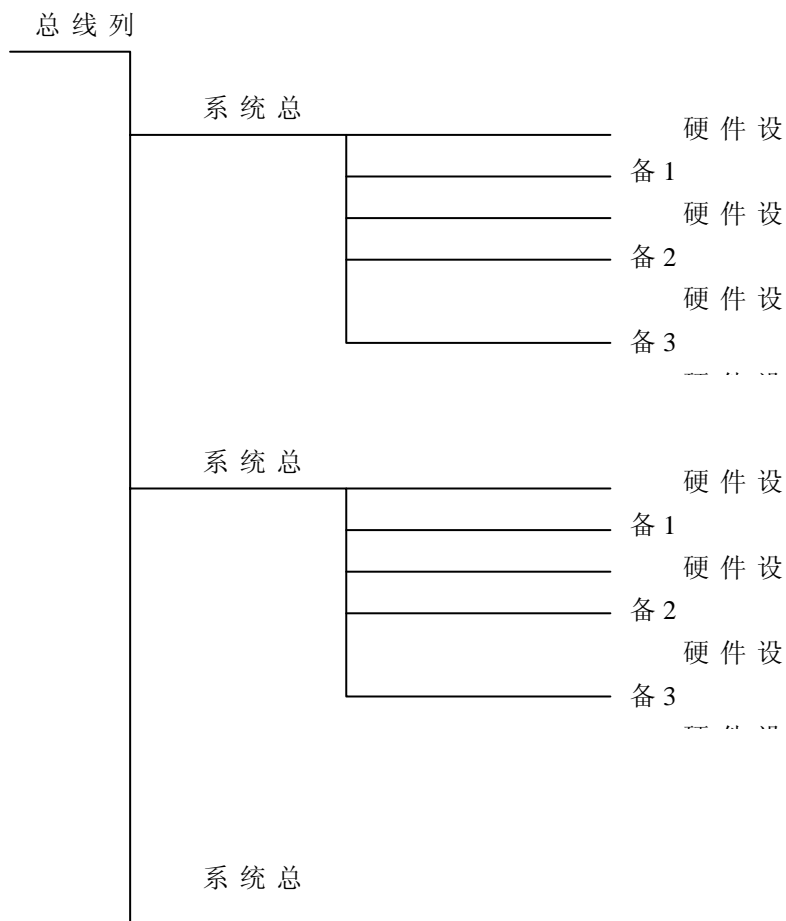


图 10-1 操作系统维护的硬件信息数据库

这样，如果要标识一个屋里设备，一种可选的方式是，为每条总线分配一个数字，作为总线号，并为位于该总线上的所有设备，分配唯一的设备号，来表示该总线上的不同设备，这样可以通过总线号+设备号，形成一个唯一的设备标识 ID，来定位到具体的设备。

这个硬件信息数据库的位置，会根据操作系统实现上的不同，而位于不同的位置，比如，Windows 操作系统，就把这些硬件设备配置信息写到磁盘上（注册表内），而有些操作系统，则会把这些硬件配置信息保存在内存中，一旦计算机重新启动，就会丢失。

完成设备硬件信息的枚举之后，下一步工作，就是为硬件设备分配系统资源了，比如中断向量号、内存映射空间、IO 端口地址、DMA 通道等，我们称这一步为设备配置。由于这些资源信息都是从一个统一的资源空间中分配，因此操作系统必须保证，为设备分配的所有资源信息，不能出现冲突的情况。举例来说，假设一台计算机外设，采用 IO 端口的方式进行通信，提供 PCI 接口，这个时候，在枚举该设备的时候，操作系统可能只会得到该设备所需要的端口范围大小，因此，操作系统需要为该设备分配端口资源范围，这就是设备配置的任务了。当然，有的时候，BIOS 可能已经为所有的硬件设备分配了 IO 端口资源，这时候操作系统需要确认，BIOS 为硬件分配的资源不会出现冲突。很可能出现的一种情况就是，BIOS 为两个不同的物理设备，分配了相同的系统资源（比如，都分配了 IO 端口 8E0H — 8EFH），这可能是由 BIOS 软件错误引起的，也可能是由于设备物理硬件原因引起的，操作系统必须能够检测到这种错误，并进行处理（比如，为其中的一台物理设备保留端口号 8E0H-8EFH，为另一台物理设备另外分配不同的 IO 端口资源）。

在对设备完成配置之后，设备还不能被正常使用，因为还没有加载设备的驱动程序。因此，完成设备配置之后，要进入正式使用步骤之前，必须加载设备驱动程序。一般来说，操作系统维护一个硬件设备 ID（比如，PCI 设备的设备 ID）和对应设备的驱动程序的一个映射文件，在完成设备的枚举和配置之后，物理设备的设备 ID 已经得到（针对不同的总线类型，设备 ID 的标识方式也不一致），这样操作系统就可以根据设备 ID 查找映射文件，找到对应的设备驱动程序的文件名，然后把设备驱动程序加载到内存中，这样设备就可以被驱动，从而可以正常使用了，对应的设备驱动程序，提供了对实际物理设备进行操作的软件代码，并以函数指针的形式表现出来。

到目前为止，从理论上说，设备已经可以使用了，因为设备已经被操作系统配置好，而且设备驱动程序已经被加载。但实际上，仅仅靠这些信息，用户应用程序是无法使用

设备的，试想，用户应用程序希望通过 Ethernet 网络接口卡发送一个数据报文，而实际系统中存在两张以太网接口卡，这种情况下，必须采用一种机制，让用户可以唯一指定一个特定的 Ethernet 接口卡，并唯一指定一个特定的操作（发送操作而不是接收操作）。对于功能的指定，可以通过不同的函数来进行，比如，针对以太网接口卡，驱动程序提供发送、接收、重新启动等一系列接口函数，这样用户就可以调用不同的功能函数，实现特定的功能。而对于设备的标识方式，一种可以选择的方式是，使用设备的物理 ID（设备 ID）直接进行标识，这样方式在理论上是可行的，但不直观，用户将面临一系列无任何特定意义的数字，非常不容易使用。因此，可以考虑采用字符串的形式，对设备进行标识。

一般操作系统的做法是，对每个具体的设备，都分配一个字符串（具有很明显的描述含义），用来唯一指定一台设备，用户可以直接看到这些设备标识字符串。这个字符串可以由操作系统在枚举设备的时候指定，也可以由设备驱动程序自己指定。设备描述字符串往往需要跟设备驱动程序进行关联，因为只有这样，才可以通过设备标识字符串，直接找到具体的设备，并定位到具体的操作函数，因此，一般情况下，操作系统会单独维护另外一个数据库，这个数据库是由一系列的设备标识字符串加上对应的设备驱动程序操作函数所组成的对象的集合，比如，下列就是一个典型的数据库元素：

```
DeviceIdentifyString
    ReadOperations;
    WriteOperations;
    ResetOperations;
    OtherOperations.
```

其中，DeviceIdentifyString 是设备的标识字符串，而接下来的一系列函数指针，则是对应驱动程序所提供的功能函数的地址。这样用户在访问具体的设备的时候，就可以通过设备字符串很容易的定位到上述数据库元素，并根据操作需求，定位到具体的操作函数。比如，用户发出一个操作请求：

```
OperateRequest("Ethernet0",SendPacket,...);
```

这样操作系统就可以查找上述数据库（根据“Ethernet0”，找到以后，并根据操作类型(SendPacket)，来定位到具体的函数，然后以剩下的参数为调用参数，来调用 SendPacket 函数。

下列的图形，示例了设备标识字符串数据库的格式：

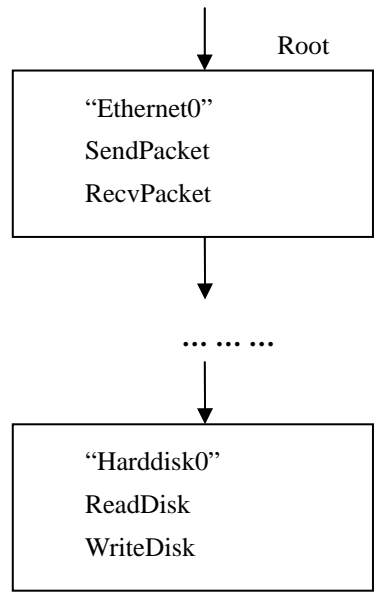


图 10-2 设备标识字符串数据库格式

对于上述数据库（链表）中的每个元素（结构），我们称为设备对象，这个存放设备对象的数据库（一般情况下，采用链表进行存放，因此有时候也称为“设备对象链表”），一般称为设备对象数据库。

显然，上述实现方式中，一个很重要的问题就是，针对不同的物理设备，需要定义不同的操作函数，这样在实现起来，显然是十分困难的，而且几乎是不可能的，因为操作系统无法预先知道所有的硬件设备。为了克服这个问题，操作系统对物理设备进行了抽象，抽象出了一组通用的函数，来操作所有的设备，其中，最典型的两个抽象出来的操作，就是 Read 和 Write，这样任何设备的驱动程序，只需要支持通用的抽象操作（其实是把设备特定的操作，以通用操作的函数原型来实现）即可，比如，针对物理硬盘和 Ethernet 网卡，都支持 Read 和 Write 操作，对于硬盘，在这两个操作中，只需要完成通常的读和写操作即可，但对于 Ethernet 网卡，则需要在读操作中，实现 ReceivePacket 功能，而在写操作中，实现 SendPacket 功能。这样实现后，对于用户程序的接口，也不用提供一个抽象的“OperateRequest”函数了，只需要提供有限的跟抽象操作对应的函数即可，比如，提供给用户一个 Read 和 Write 函数，这两个函数跟物理设备对应的驱动程序

提供的操作相对应，每当用户针对特定的设备，调用这两个函数的时候，操作系统就会把这种调用映射到对应设备驱动程序的相应函数，从而实现设备的透明访问。一个比较典型的例子就是，Windows 操作系统提供的 `ReadFile` 函数和 `WriteFile` 函数，这两个函数不但可以用于完成文件的读写操作，也可以完成设备的读写操作，实际上，在对物理设备（非文件）调用这两个函数的时候，操作系统就把这些函数的调用，传递到了设备驱动程序相关函数的调用上。

到此为止，用户就可以很容易的访问硬件设备了，比如，用户调用 `Read("Harddisk0",...)`（函数参数中，省略的部分为传递的参数），操作系统就会根据设备标识字符串“Harddisk0”查找设备对象链表，找到“Harddisk0”对应的设备对象，然后以用户传递过来的参数为参数（或稍做调整），调用设备驱动程序提供的 `Read` 函数。

一切似乎都很完善，但细心的读者可能发现，对于任何针对设备的操作，如果按照上述形式，都需要操作系统完成一个字符串查找工作（根据函数提供的设备标识字符串，查找设备对象数据库），这显然是十分低效的，尤其是设备操作十分频繁的时候。目前，大多数操作系统都提供一个打开（`Open`）操作，用户在这个函数调用中，指定设备标识字符串作为参数，函数返回的时候，返回一个句柄（`Handle`），在实现上，这个句柄可能是设备对象的指针，或者其它可以快速检索到设备对象的数据，而后续操作（比如 `Read`、`Write` 等），则不必提供设备标识字符串，而使用 `Open` 函数返回的句柄作为参数，这样操作系统就可以省略查找过程，而直接通过句柄快速定位到设备对象，从而调用设备对象的相关操作。比如，对一个物理硬盘的访问，遵循下列顺序：

```
HANDLE hHardDisk = NULL_HANDLE;
hHardDisk = Open("Harddisk0",...);
if(NULL_HANDLE == hHardDisk)    //Can not open this device.
    return FALSE;
Read(hHardDisk,...);            //Read device using handle.
Close(hHardDisk);               //Close this device.
```

在对设备的操作完成之后，为保险或节约系统资源起见，一般需要采用 `Close` 函数（由操作系统提供）关闭打开的设备。

显然，这样对设备的访问，就十分完善了，如果读者对 Windows API 十分熟悉，通过上面的叙述，就应该对 Windows 操作系统提供的 `CreateFile`、`ReadFile`、`WriteFile`、`CloseHandle` 等函数的实现机制，有一定了解了，这些函数，分别与上面介绍的 `Open`、`Read`、`Write`、`Close` 对应。

最后补充一点，引入设备对象数据库（设备对象链表）的另外一个目的，是用于存储多设备实例情况下，单个设备实例的特定状态数据。比如，计算机系统配备了两个 IDE 接口的物理硬盘，由于这两个物理硬盘都是 IDE 接口，因此只需要一个 IDE 驱动程序即可，这样为了存储这两个物理硬盘的相关信息，就可以创建两个设备对象，这两个设备对象分别具有不同的标识字符串（比如，“IDEHD0”和“IDEHD1”），以及不同的状态参数（比如，当前磁头的位置、当前需要读写的扇区个数等），但这两个设备对象提供的操作函数，却是一样的，都是 IDE 接口硬盘驱动程序提供的参考函数。下面是设备对象数据库的一个更详尽的示例：

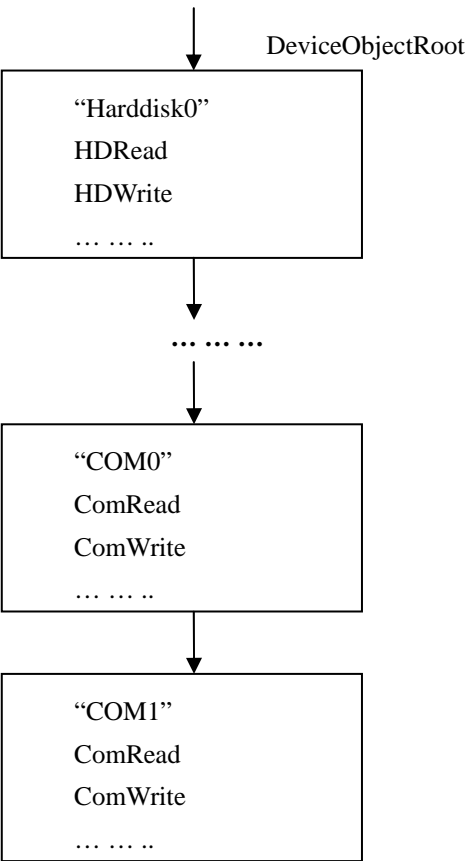


图 10-3 一个设备对象数据库的例子

在这个例子中，所有的设备对象被存储在一个链表数据结构中，第一个设备对象“Harddisk0”，是一个硬盘设备，HDRead 和 HDWrite 函数是硬盘驱动程序提供的操作方法，这两个操作方法的原型，必须与操作系统抽象的一致，而 COM1 和 COM0 则是两个串口对象，这两个物理设备对象采用同一个设备驱动程序（COM 通信端口驱动程序）提供的操作方法，需要注意的是，ComRead 和 HDRead 的函数原型（参数）必须一样，都必须与操作系统抽象的操作函数保持一致。

综上所述，为了对计算机系统中的设备进行有效管理，操作系统一般情况下需要维护两套数据结构（数据库）：硬件信息数据库和设备对象数据库（设备对象链表），其中，硬件信息数据库是操作系统在引导的时候，通过收集硬件信息而建立，用于维护系统中的硬件配置信息以及硬件物理参数，而设备对象数据库，则是由操作系统建立，用于完成特定的设备跟其操作方法（驱动程序）的关联，并提供设备标识字符串，用以标识设备，还用来保存设备运行过程中的状态信息。

### 10.1.2.2 Hello China 的设备管理机制

为了对物理设备进行管理，一般情况下，操作系统需要维护两个数据库：硬件信息数据库和设备对象数据库，并且提供对这两个数据库的相应操作，比如在数据库中添加记录、删除记录等维护工作。根据面向对象的思想，一个比较简便的实现形式就是，创建两个对象，分别对应上述两个数据库的数据结构以及操作（方法）。

在 Hello China 的实现中，物理设备管理模块就是遵循这种思路实现的。在当前版本中，实现了下列两个对象：设备管理器对象和 IO 管理器对象，其中设备管理对象用于管理物理设备的硬件信息数据库，而 IO 管理器对象则用于完成设备对象数据库的维护。

#### 10.1.2.2.1 设备管理器 (DeviceManager)

设备管理器对象的定义如下：

```
BEGIN_DEFINE_OBJECT(__DEVICE_MANAGER)
#define MAX_BUS_NUM 16
    __SYSTEM_BUS                SystemBus[MAX_BUS_NUM];
    __RESOURCE                   FreePortResource;
```

```

        __RESOURCE                UsedPortResource;
        BOOL                      (*Initialize)(__DEVICE_MANAGER*);
        __PHYSICAL_DEVICE*       (*GetDevice)(__DEVICE_MANAGER*,
                                           DWORD
dwBusType,
                                           __IDENTIFIER*
lpIdentifier,
                                           __PHYSICAL_DEVICE* lpStart);
        __RESOURCE*              (*GetResource)(__DEVICE_MANAGER*,
                                           DWORD
dwBusType,
                                           DWORD
dwResType,
                                           __IDENTIFIER*
lpIdentifier);
        BOOL                     (*AppendDevice)(__DEVICE_MANAGER*,
                                           __PHYSICAL_DEVICE*);
        VOID                     (*DeleteDevice)(__DEVICE_MANAGER*,
                                           __PHYSICAL_DEVICE*);
        BOOL                     (*CheckPortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*);
        __RESOURCE*              (*ReservePortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*,
                                           DWORD dwLength);
        VOID                     (*ReleasePortRegion)(__DEVICE_MANAGER*,
                                           __RESOURCE*);

    END_DEFINE_OBJECT()

```

这个对象维护了一个类型是\_\_SYSTEM\_BUS 的数组(目前定义该数组的大小是16), 这个数组用于存储系统中配置的所有总线, 对于总线上的物理设备, 都是以物理设备对象(\_\_PHYSICAL\_DEVICE)的形式, 连接到系统总线对象上。上述对象也对系统资源进行了统一管理, FreePortResource 和 UsedPortResource 两个数组, 用于记录当前空闲的 IO 端口号范围和已经占用的 IO 端口号范围, 对于中断和虚拟内存空间的管理, 为了实现上的方便, 在另外的两个对象(\_\_SYSTEM 对象和\_\_VIRTUAL\_MEMORY\_MANAGER

对象)中进行管理,在此不做赘述。

这个对象还提供了用于操作物理设备信息数据库的方法,比如 `AppendDevice`、`DeleteDevice`、`ReservePortRegion` 等,这些函数完成物理设备的添加、删除、端口号的预留等工作,这些函数的目的,是为设备驱动程序提供接口,让设备驱动程序能够自主的配置硬件(比如,为硬件保留 IO 端口信息),这些设备不支持自动配置功能,因此需要设备驱动程序自己完成配置工作。而对于支持自动配置功能的设备,比如 PCI 总线接口的设备,其配置工作无需调用这些函数,而是在 `Initialize` 函数中,被该对象一并检测并完成配置,因此, `Initialize` 函数的功能比较复杂,该函数检测系统中的所有总线,并完成总线上设备的枚举和配置,一旦该函数成功执行完毕,物理硬件数据库(实际上是 `SystemBus` 数组)中就建立了全面的物理硬件拓扑结构,以及物理硬件的配置信息,当然,这个过程无法检测到不支持自动配置的物理设备,对于不支持自动配置的物理设备,需要驱动程序的初始化代码调用 `AppendDevice`、`ReservePortRegion` 等函数,手工的添加到物理设备信息数据库中。

`Initialize` 函数成功执行完毕之后,就可以根据收集到的物理设备信息,加载驱动程序了,由于 `Hello China` 的设计目标是一个嵌入式的操作系统,系统中的硬件事先已经配置好,设备驱动程序也一般会跟操作系统核心一起编译连接,因此,在当前的实现中,无需实现驱动程序加载功能(从存储设备上读取驱动程序,并加载到内存中),但如果将来需要实现这项功能,也比较方便,就是在 `Initialize` 函数中,另外添加部分代码,完成物理设备驱动程序的加载工作。

对于 `_DEVICE_MANAGER` 对象的详细信息,请参考“系统总线管理”章节。

#### 10.1.2.2.2 I/O 管理器 (IOManager)

IO 管理器用于管理设备对象数据库(设备对象列表),并提供一组函数(方法),用于操作这个数据库,比如,提供了设备对象的创建、销毁等函数,供硬件驱动程序调用。对于设备驱动程序的管理,也是由 `IOManager` 进行,在当前版本的 `Hello China` 的实现中,设备驱动程序的加载,采用静态方式,即设备驱动程序代码和操作系统核心连接到一起,作为一个统一的软件模块,加载到目标系统中,这是因为 `Hello China` 的设计目标,是一个嵌入式操作系统。但这不影响驱动程序的管理构架,也就是说,即使是采用动态加载驱动程序的方式,也遵循同样的驱动程序管理构架。当前版本的实现中,对于驱动程序的管理,按照下列方式进行:

- 1、`IOManager` 针对每个驱动程序,创建一个驱动程序对象(`_DRIVER_OBJECT`),初始化,然后以该对象为参数,调用驱动程序提供的 `DriverEntry` 函数(每个驱动程序必须输出一个 `DriverEntry` 函数,作为驱动程序的入口函数);
- 2、驱动程序使用输出的操作函数(`Read`、`Write` 等),填写驱动程序对象的相关成员变

量；

- 3、驱动程序在 `DriverEntry` 中，检查系统中对应的物理设备（通过调用 `DeviceManager` 提供的 `GetDevice` 函数），针对每个自己支持的设备，驱动程序必须创建一个设备对象（`_DEVICE_OBJECT`，通过调用 `IOManager` 提供的函数创建），并初始化该物理设备对象，比如赋予物理设备对象标识字符串等；
- 4、第三步完成之后，由驱动程序创建的物理设备对象，就会插入设备对象链表（由 `IOManager` 维护）中，一旦插入设备对象链表，就对应用程序可见了，应用程序就可以采用 `Open` 系统调用，打开这个设备，并调用 `Read`、`Write` 等函数，对设备进行操作了。

另，该对象也提供了应用程序调用的标准接口，比如 `Read`、`Write` 等，这些函数跟驱动程序实现的一组标准接口对应，用户调用这些函数的时候，`IOManager` 会把用户的调用，映射到驱动程序提供的相应函数上，对于调用的参数，直接从用户调用的参数传递到驱动程序提供的函数里，或者稍做调整，然后再传递到驱动程序提供的函数上。

### 10.1.3 Hello China 的设备管理框架

在 `Hello China` 的实现中，对于设备的管理，是按照下列策略来实现的：

- 1、采用一个统一的设备管理对象（`IOManager`），来集中管理所有的设备、设备驱动程序以及系统资源，并实现系统资源的分配和回收，以及设备驱动程序的加载和卸载；
- 2、在设备管理对象和设备驱动程序之间定义了一个标准的交互接口，设备驱动程序和 `IOManager` 就是通过这个规定好的接口进行通信；
- 3、在设备管理对象和用户线程（核心线程）之间，也定义了一个良好的交互接口，用户线程（核心线程）直接调用 `IOManager` 的用户侧接口，请求设备管理对象的服务（最终实现对设备的功能调用）；
- 4、把文件系统的实现也纳入设备管理框架里面，对文件的访问，也是通过 `IOManager` 的用户侧接口进行的；
- 5、文件系统实现的时候，也作为驱动程序来实现，遵循设备驱动程序的体系结构，也遵循设备驱动程序与操作系统的通信机制；
- 6、设备驱动程序代码和 `IOManager` 代码必须是可重入的，即多个用户线程可以同时调用同一个设备驱动程序功能函数（或 `IOManager` 函数），而不会发生不一致的资源访问问题。为了实现这个功能，需要在 `IOManager` 的实现中，引入互斥机制，在设备驱动程序的实现中，需要考虑自己可能管理多个设备的情况，并为每个设备建立一套单独的数据环境，来充分保证代码的可重入性；
- 7、实现设备的动态发现和枚举，比如，针对 `PCI` 总线，`IOManager` 可以动态的发现连接在总线上的设备，并为之分配系统资源（中断号、端口号、内存映射区域等），动态加载这些设备的驱动程序；
- 8、即插即用（`PnP`），实时监视总线状态，对于实时出现在总线上的设备，`IOManager` 会及时做出响应，比如分配资源、加载驱动等，对于从总线上实时拆离的设备，

IOManager 会及时卸载掉已经加载的驱动程序，并释放这些设备所占用的资源。

上述两点功能，根据最新设计，将在 DeviceManager（设备管理器）中实现。

在满足上述功能模型的情况现下，Hello China 的设备管理框架如下所示：

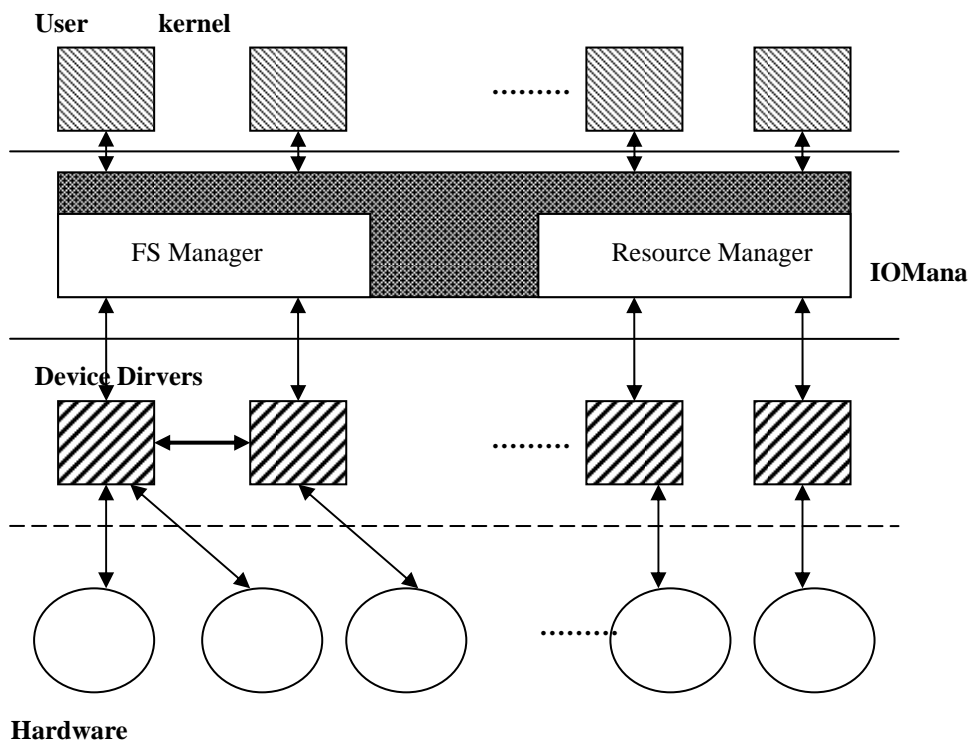


图 10-4 Hello China 的设备管理框架

可以看出，在整个设备管理框架中，IOManager 成为核心部件，IOManager 提供了两个规范的接口，对于用户核心线程，称为用户接口（或上行接口，图中蓝色箭头表示），对于设备驱动程序，称为设备接口（或下行接口，图中红色箭头表示），用户线程通过用户接口调用 IOManager，进而获得设备服务，设备驱动程序通过设备接口，来调用

IOManager 提供的服务，或通知 IOManager 自己的存在。

需要注意的是，设备驱动程序之间也可能相互调用彼此的服务（图中紫色的双向箭头），这种情况的一个典型应用，就是文件系统的实现。比如，用户通过 IOManager 提供的调用接口，来访问一个文件（打开、读写等），IOManager 把这种调用转化为对相应文件系统的调用，相应的文件系统在完成内部表格的修改后，需要对实际的物理存储设备进行操作，这个时候，文件系统驱动程序（在 Hello China 的实现中，文件系统作为一种特殊的驱动程序来实现）就需要调用实际的物理设备驱动程序，来对实际的物理设备进行操作。

对于中断管理，设备驱动程序在获得系统为自己分配的资源（中断号、端口号、内存映射区域、DMA 通道等）后，可以以中断号为参数，调用 ConnectInterrupt 函数（该函数由核心提供），直接注册自己的中断处理函数，当然，在设备卸载的时候，系统也可以调用 DisconnectInterrupt 函数，解除自己注册的中断调用，从而释放中断资源。

需要注意的是，一个设备驱动程序，可能管理多个设备（或逻辑的设备功能），设备和设备驱动程序之间的交互接口，由设备驱动程序实现，在 IOManager 中不作任何假设，另外，之所以把设备和设备驱动程序之间的交互表示为双向（图中的双向黑色箭头），是因为设备可能通过中断的方式，跟设备驱动程序交互。

在下面的各部分中，我们详细介绍该管理框架中涉及到的模块，以及模块之间的接口。

### 10.1.4 I/O 管理器（IOManager）

IO 管理器（IOManager）是系统中的全局对象之一，整个系统中，只存在一个这样的对象，该对象提供了面向应用的接口，比如 CreateFile，ReadFile 等函数，供用户线程调用，来访问具体的设备，该对象还提供了面向设备驱动程序的接口，供设备驱动程序调用，完成诸如创建设备、预留资源、销毁设备等操作。

系统中所有加载的设备驱动程序都归该对象管理，系统中所有用户可以使用的设备，也归该对象管理，因此，该对象可以认为是设备管理框架的核心对象。

#### 10.1.4.1 驱动程序对象和设备对象

在 Hello China 的实现中，对于每个加载的设备驱动程序，系统都为之创建一个驱动

程序对象，并调用驱动程序的 **DriverEntry** 函数（参考本文中设备驱动程序一节）来初始化这个驱动程序对象。驱动程序对象保存了对设备进行操作的所有函数，比如对设备的读函数、对设备的写函数等。

驱动程序对象可以理解为管理设备驱动程序的数据结构，而设备对象则对应于具体的物理设备，即设备对象是对物理设备进行直接管理的数据结构。设备对象由驱动程序创建，一般情况下，是在设备驱动程序加载并初始化的时候创建，一个比较合适的时机，就是在 **DriverEntry** 函数中创建。

在设备对象中，有一个指向对应于该设备的设备驱动程序对象的指针，对于设备的所有操作，都是由驱动程序对象提供的函数完成的，通过设备对象指向驱动程序对象的指针，可以找到特定的设备操作函数，进而完成对设备的操作。

为了说明设备对象和设备驱动程序对象的关系，在这里以磁盘驱动程序为例，来说明整个流程：

- 1、在系统启动的时候，根据配置文件，或总线检测结果，加载硬盘驱动程序；
- 2、完成驱动程序的加载（加载过程包括读入驱动程序文件、重定位、根据文件头找到 **DriverEntry** 函数的入口地址等）后，**IOManager** 创建一个设备驱动程序对象，并以该对象为参数，调用硬盘驱动程序的 **DriverEntry** 函数；
- 3、驱动程序（实际上是 **DriverEntry** 函数）根据 **IOManager** 提供的配置，对设备进行检测，比如，系统中硬盘的数量、每个硬盘的分区情况等，都在这个检测过程中完成，实际上，检测是一个收集数据的过程；
- 4、硬盘驱动程序根据收集的数据，比如，系统中的硬盘数量，以及每个硬盘的分区情况等，创建相应的设备对象（通过调用 **IOManager** 提供的 **CreateDevice** 函数），一般情况下，针对每个硬盘、每个硬盘分区，分别创建设备对象，比如，假设系统中安装了一个硬盘，该硬盘划分了四个分区，则 **DriverEntry** 创建五个设备对象（分别为硬盘设备对象、分区一设备对象、分区二设备对象、分区三设备对象和分区四设备对象）；
- 5、上述步骤完成之后，硬盘就可以供具体的应用线程使用了。

比如，假设有一个用户线程读取硬盘数据，则具体的过程如下：

- 1、用户调用 **ReadFile** 函数，发起一个硬盘读取请求（该函数的参数提供了硬盘对象设备对象的地址）；
- 2、**IOManager** 根据 **ReadFile** 提供的设备对象的地址，找到该对象对应的驱动程序对象（设备对象保存了指向驱动程序对象的指针）；
- 3、**IOManager** 创建一个 **DRCB** 对象（参考 **DRCB** 对象一节），初始化，然后调用驱动程序对象中，特定的函数（**DeviceRead** 函数）；
- 4、该函数完成具体的硬盘读写操作，并返回；
- 5、**IOManager** 根据返回的结果，填充用户缓冲区，然后返回给用户线程。

需要指出的是，在调用 `ReadFile` 函数，读取设备内容的时候，需要首先打开设备（调用 `CreateFile` 函数）。

#### 10.1.4.2 IOManager 对设备对象和设备驱动程序的管理

在当前版本 `Hello China` 的实现中，所有驱动程序对象和设备对象，都是由 `IOManager` 直接管理的，在实现中，`IOManager` 维护了两个双向链表，一个链表把系统中所有的驱动程序对象连接在一起，另一个链表把系统中所有的设备对象连接在一起，在 `IOManager` 的定义中，有两个成员变量：

```
__DEVICE_OBJECT*    lpDeviceRoot;  
__DRIVER_OBJECT*    lpDriverRoot;
```

这两个变量指向两个双向链表的头节点。

整体构架请参考下图：

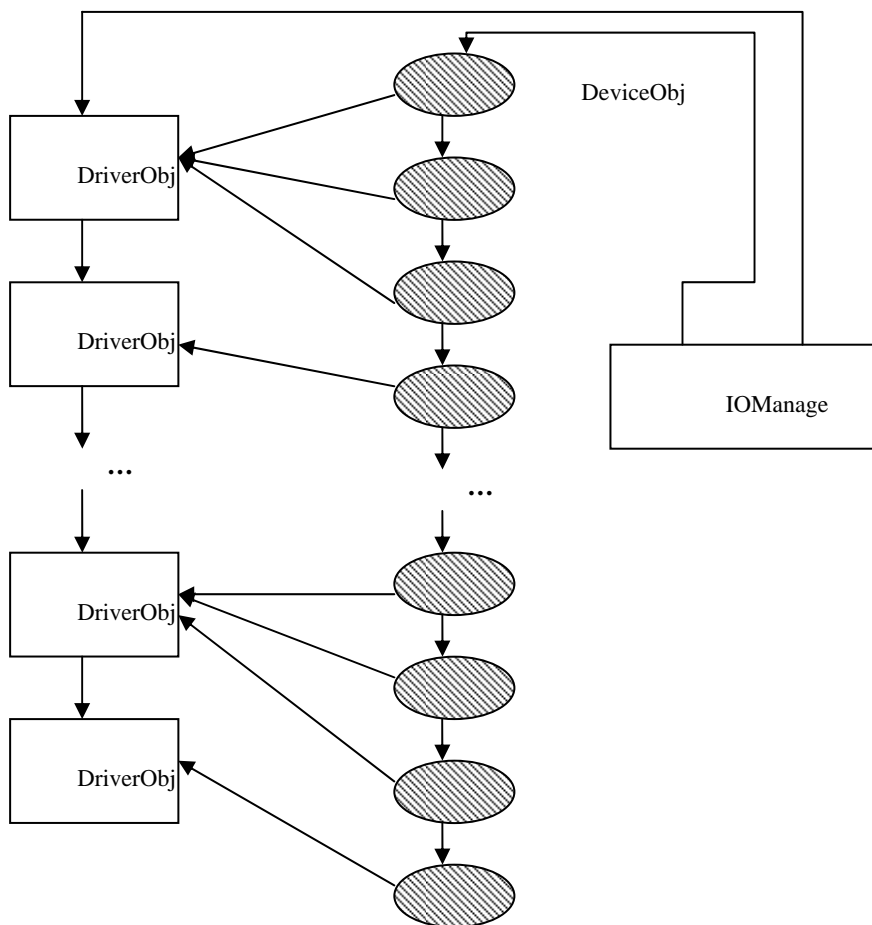


图 10-5 Hello China 的设备对象和设备驱动程序对象

### 10.1.4.3 IOManager 的实现框架

IOManager 的定义如下：

```
BEGIN_DEFINE_OBJECT(__IO_MANAGER)
    __DEVICE_OBJECT*          lpDeviceRoot;
```

```

__DRIVER_OBJECT*      lpDriverRoot;

//Initializing routine.
BOOL                  (*Initialize)(__COMMON_OBJECT*   lpThis);

//The interface(functions) to user kernel thread.
__COMMON_OBJECT*      (*CreateFile)(__COMMON_OBJECT*
lpThis,

                                LPSTR
                                lpFileName,

                                DWORD          dwAccessMode,
                                DWORD
                                dwOperationMode,

                                LPVOID         lpReserved);
BOOL                  (*ReadFile)(__COMMON_OBJECT*
lpThis,

                                __COMMON_OBJECT* lpFileObj,
                                DWORD          dwByteSize,
                                LPVOID         lpBuffer,
                                DWORD*         lpReadSize);
BOOL                  (*WriteFile)(__COMMON_OBJECT*   lpThis,
__COMMON_OBJECT*
lpFileObj,

                                DWORD          dwWriteSize,
                                LPVOID         lpBuffer,
                                DWORD*         lpWrittenSize);
VOID                  (*CloseFile)(__COMMON_OBJECT*   lpThis,
__COMMON_OBJECT*
lpFileObj);
BOOL                  (*IoControl)(__COMMON_OBJECT*   lpThis,
__COMMON_OBJECT*
lpFileObj,

                                DWORD
                                dwCommand,
                                dwCommand,

```

```

                                DWORD
dwInputLen,

                                LPVOID
lpInputBuffer,

                                DWORD
lpOutBufferLen,

                                LPVOID
lpOutBuffer);
        BOOL                                (*SetFilePointer)(__COMMON_OBJECT*
lpThis,

                                __COMMON_OBJECT*
lpFileObj,

                                DWORD
dwWhereBegin,

                                INT            dwOffset);
        VOID                                (*FileFlush)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT*
lpFileObj);

        //The interface(functions) to device drivers.
        __DEVICE_OBJECT*                    (*CreateDevice)(__COMMON_OBJECT*
lpThis,

                                LPSTR
lpDevName,

                                __COMMON_OBJECT*
lpDrv,

                                LPVOID
lpDevExtension,

        /* __RESOURCE_DESCRIPTOR*
                                lpResDesc,

                                DWORD
dwDevType,

                                DWORD

```

```
dwBlockSize*/

);

VOID (*DestroyDevice)(__COMMON_OBJECT*
lpThis,
__COMMON_OBJECT*
lpDevObj);

END_DEFINE_OBJECT() //End to define __IO_MANAGER.
```

可以看出，IOManager 的定义比较复杂，涉及到比较多的函数，但这些对外函数（或接口）总体上可以分为三类：

- 1、初始化函数（Initialize），系统初始化的时候，调用该函数初始化 IOManager；
- 2、对用户的接口，由用户调用，用来访问设备；
- 3、对设备驱动程序的接口，由设备驱动程序调用，来获得 IOManager 的服务。

在下面的部分中，分别对这三类函数进行描述。

10.1.4.4 初始化函数（Initialize）

初始化函数（Initialize）用来进行一些初始化工作，在 Hello China 启动的时候调用，该函数完成一些预定的初始化工作，在目前的实现中，该函数完成下列功能：

- 1、初始化设备驱动程序，当前版本的 Hello China，设计目标为嵌入式操作系统，这样就不需要动态的加载设备驱动程序，设备驱动程序事先已经跟操作系统内核编译在一起了。但设备驱动程序所遵循的框架，也跟动态加载的设备驱动程序一致，不同的是，少了加载的步骤（动态加载设备驱动程序包括从存储设备读入驱动程序、重定位等步骤）。在 IOManager 的 Initialize 函数中，会调用每个连接到操作系统核心的设备驱动程序的 DriverEntry 函数；
- 2、其它相关工作。

上述所有工作顺利完成之后，Initialize 函数将返回 TRUE，若该函数返回 FALSE，会导致系统停止引导。

另外一个问题就是，对于跟操作系统核心连接在一起的驱动程序，Initialize 函数如何确定其入口点（DriverEntry 函数）。为了解决这个问题，当前版本的 Hello China 定义了下列一个数据结构：

```

BEGIN_DEFINE_OBJECT(__DRIVER_ENTRY_MAP)
    LPSTR                lpszDriverName;
    BOOL                 (*DriverEntry)(__DRIVER_OBJECT*);
END_DEFINE_OBJECT()

```

并定义了一个全局数组：

```

__DRIVER_ENTRY_MAP DriverEntryMap[] = {
    {"Ide hard disk",IdeDriverEntry},
    {"Mouse",MouseDriverEntry},
    {"Keyboard",KeyboardDriverEntry},
    {"Screen",ScreenDriverEntry},
    ... ..
    {NULL, NULL}
};

```

这样，在 `Initialize` 的实现中，就会遍历这个数组，对于数组中的每个元素，创建一个 `__DRIVER_OBJECT` 对象，然后调用对应的 `DriverEntry` 函数：

```

BOOL IoMgrInitialize(__IO_MANAGER* lpThis)
{
    BOOL                bResult                = FALSE;
    __DRIVER_OBJECT*    lpDriver                = NULL;
    DWORD               dwLoop                = 0L;

    ... ..
    while(DriverEntryMap[dwLoop].lpszDriverName)
    {
        lpDriver = ObjectManager.CreateObject(&ObjectManager,
                                              NULL,
                                              OBJECT_TYPE_DRIVER_OBJECT);

        if(NULL == lpDriver)    //Failed to create driver object.
            goto __TERMINAL;

        if(!(DriverEntryMap[dwLoop].DriverEntry)(lpDriver))    //Failed to initialize

```

driver.

```

        {
            PrintLine("Unable to initialize driver.");
            PrintLine(DriverEntryMap[dwLoop].lpzDriverName);
        }
        dwLoop ++
    }
    ... ..
    bResult = TRUE;
__TERMINAL:
    return bResult;
}

```

因此，对于每个需要静态编联并加载的设备驱动程序，程序开发者都需要在 `DriverEntryMap` 数组中手工添加一条记录，该数组的最后一条空记录（`{NULL,NULL}`），是该数组结束的标记。

虽然目前情况下，Hello China 没有实现动态设备驱动程序的加载功能，但将来的时候，如果需要，可以按照下列思路，来实现动态设备驱动程序的加载功能：

通过另外一个帮助函数，`GetDriverEntry`，得到需要动态加载的设备驱动程序的入口地址，该函数原型如下：

```

LPVOID      GetDirverEntry(__DEVICE_VENDOR*      lpDevVendor,LPSTR
lpDrvName);

```

其中，`lpDevVender` 指向一个设备厂家 ID 结构，该结构描述了 `IOManager` 想要加载的设备的厂家信息，而 `lpDrvName` 则指明了加载的设备驱动程序的名字（可以为空）。`GetDriverEntry` 根据厂家信息，查询系统的一个配置文件，找到对应的驱动程序的文件名，然后调用 `ModuleManager` 的特定函数，`ModuleManager` 根据文件名，在存储设备上找到合适的驱动程序，然后加载到内存（加载过程包括了重定位、名字解析、初始化等操作），并返回给加载模块的起始地址（返回给 `GetDriverEntry`）。

其中，`ModuleManager` 是模块管理器，用来完成把磁盘上的代码（可执行模块，比如动态链接库、应用程序可执行文件等）加载到内存中，并重定位等功能。

在实现动态设备驱动程序加载的时候，`IOManager` 的 `Initialize` 函数，需要调用 `DeviceManager` 的相关函数，遍历系统中的硬件配置，对于检索到的每一个硬件，根据该硬件的 `__DEVICE_VENDOR` 标识，调用 `GetDriverEntry` 函数。

#### 10.1.4.5 IOManager 对应用的接口

IOManager 提供了两个方向的接口：对应用程序的接口和对设备驱动程序的接口。其中，对应用程序的接口被应用线程调用，用来访问具体的设备，下列接口（函数）是对应用的接口（函数）：

- ✓ **CreateFile**，用于打开一个文件或设备，在 Hello China 当前版本的实现中，所有的设备和文件同等对待，都是用名字来标识，该函数既可以打开某一文件系统中的特定文件，也可以打开一个特定的物理设备；
- ✓ **ReadFile**，从文件或设备中读取数据。在当前版本的实现中，该函数采用同步操作模式，即该函数一直等待设备操作完成，而不是中途返回（在 Windows API 中，实现了一种所谓的异步操作模式，即该函数向操作系统提交一个读取事务，然后直接返回，当操作系统完成事务指定的读写动作后，向发起事务的进程发送一个消息，进程处理该消息，最后完成读写操作），在这个过程中，调用该函数的线程可能被阻塞；
- ✓ **WriteFile**，向设备或文件写入数据，实现机制与 ReadFile 类似；
- ✓ **CloseFile**，CreateFile 的反向操作，用于关闭 CreateFile 打开的设备或文件，在这个函数的实现中，如果操作目标是一个文件，则系统直接把相应的文件对象销毁，如果操作的对象是物理设备，则该对象不被销毁，而是递减对象的引用计数；
- ✓ **IOControl**，完成设备驱动程序独特的操作，有些操作是不能通过 Read、Write 等来抽象的，比如针对音频设备的快进、重复播放等，系统提供了该函数，相当于提供了一个万能的接口给用户程序，用户程序可以通过该函数调用，完成任意驱动程序特定的操作功能；
- ✓ **SetFilePointer**，移动文件的当前指针；
- ✓ **FlushFile**，把位于缓冲区中的文件内容写入磁盘，一般情况下，文件系统的实现大量的使用了缓冲机制，即对文件的写操作先在内存中完成，然后积累到一定的程度后，再由设备驱动程序统一递交到物理设备，这样可以大大提高操作效率，但有的情况下，应用程序可能需要立即把改写的文件内容，写到物理存储设备上，比如，应用程序关闭的时候，这样就需要调用该函数来主动的同步缓存和物理存储介质，需要说明的是，CloseFile 在实际关闭文件对象前，总是调用 FlushFile 来同步缓冲区和物理存储介质。

有的操作系统提供了 LockFile 函数，该函数用于把打开的文件加锁，实现互斥的访问，在 Hello China 当前的实现中，没有提供该函数功能，主要是考虑到该函数同能用途可能不是很大，而且可以通过一些替代方式来完成，比如，应用程序可以独占的打开一个文件，也可以在打开文件的时候，指定另外的打开标志，只允许其它应用程序只读的打开文件，等等。

下面对这些函数的实现，进行详细描述。

10.1.4.6 CreateFile 的实现

CreateFile 的函数原型如下：

```
__COMMON_OBJECT* (*CreateFile)(__COMMON_OBJECT*
lpThis,
LPSTR
lpzFileName,
DWORD dwAccessMode,
DWORD
dwOperationMode,
LPVOID lpReserved);
```

其中，第一个参数lpThis是一个指向IOManager全局对象的指针，第二个参数lpzFileName，则指明了要打开的设备或文件的名称，dwAccessMode和dwOperationMode两个参数，用于对打开的文件进行控制，对于目标是物理设备的情形不适用，最后一个参数保留，用于将来使用。不同的命名规则，用来标识打开的目标对象是设备还是文件，对于设备，lpzFileName的形式遵循 [\\.\devname](#) 的规则，即开始是两个反向斜线，接着一个点号（或DEV字符串），后面再跟着一个反向斜线，最后是设备的名字（即设备标识字符串），而对于文件，则按照文件系统名称+文件路径+文件名的格式，比如“C:\HCN\DATA1.BIN”。详细的命名规则，请参考“设备对象命名规则”一节。

如果调用该函数打开物理设备，则该函数执行下列动作：

- 1、从lpzFileName中提取设备标识字符串（lpzFileName头部包含了 [\\.](#) 字符串，而这部分内容不属于设备标识字符串内容）；
- 2、以设备标识字符串检索设备对象（\_\_DEVICE\_OBJECT）链表，一旦找到匹配的设备对象，则停止查找；
- 3、若找到匹配的物理设备对象，递增设备对象的引用计数（dwRefCounter），然后返回设备对象的指针；
- 4、若不能查找到物理设备对象，则返回 NULL，表示该函数操作失败。

调用该函数打开文件的相关过程，请参考“文件系统的实现”相关内容。

10.1.4.7 ReadFile 的实现

ReadFile 函数是 IOManager 提供给应用程序的一个最重要函数，所有对物理设备或

文件的读取访问，都是通过该函数进行的，原型如下：

```

        BOOL                                     (*ReadFile)(__COMMON_OBJECT*
lpThis,
                                                __COMMON_OBJECT* lpFileObj,
        DWORD                                     dwByteSize,
        LPVOID                                    lpBuffer,
        DWORD*                                   lpReadSize);

```

该函数用于从已经打开的设备（通过调用 `CreateFile` 函数）中读取部分数据，其中，`lpFileObj` 参数指定了打开的设备，`dwByteSize` 指定了读取的字节数，而 `lpBuffer` 则是一个缓冲区指针，从设备中读取的数据，将被存储在该缓冲区内。若该函数操作成功，则返回 `TRUE`，实际读取的字节数在 `lpReadSize` 参数中返回，若该函数操作失败，则返回 `FALSE`。

在 `CreateFile` 函数的描述中，我们知道，`lpFileObj` 实际上是一个设备对象的地址，而设备对象包含了一个指向驱动程序对象的指针，通过这个指针，可以直接找到驱动程序提供的 `DeviceRead` 函数，并调用这个函数。这个过程如下：

```

... ..
__DRIVER_OBJECT*      lpDrvObj = NULL;
lpDrvObj = (__DEVICE_OBJECT*)lpFileObj->lpDriverObject;
bResult = lpDrvObj->DeviceRead((__COMMON_OBJECT*)lpDrvObj,
                               (__COMMON_OBJECT*)lpFileObj,
                               lpDrcb);
... ..

```

在 `DeviceRead` 函数的参数中，第一个是指向驱动程序对象的指针，第二个则是设备对象的指针，因为特定的设备状态，都是存储在设备对象中，而 `DeviceRead` 函数需要这些状态数据。第三个参数是一个 `__DRCB` 对象，该对象用于跟踪设备请求操作。对于 `__DRCB` 对象的初始化过程如下：

```

__DRCB*      lpDrcb = NULL;
lpDrcb = ObjectManager.CreateObject(&ObjectManager,

```

```

        NULL,
        OBJECT_TYPE_DRCB);
If(NULL == lpDrcb)    //Can not create DRCB object.
    goto __TERMINAL;

lpDrcb->dwDrcbStatus = DRCB_STATUS_PENDING; //Pend the DRCB object.
lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_READ; //Read operation.
lpDrcb->dwOutputLen = dwByteSize;
lpDrcb->dwOutputBuffer = lpBuffer;

```

在上述代码中，把 DRCB 对象的状态（dwDrcbStatus）初始化为 DRCB\_STATUS\_PENDING，因为该 DRCB 对象即将被排队（一旦调用 DeviceRead 函数，DRCB 对象就会被排入驱动程序维护的读取或写入队列中），把 DRCB 对象的请求模式（dwRequestMode，即操作类型）初始化为 DRCB\_REQUEST\_MODE\_READ，初始化 DRCB 的输出缓冲区指针，以及缓冲区的大小，完成上述操作之后，就可以使用该 DRCB 对象为参数，调用 DeviceRead 函数了。

上面描述的是一种理想的情况，实际的操作，涉及到一个比较麻烦的问题，就是数据分片问题，比如，应用程序打开了一个以块形式进行操作的物理设备，比如 IDE 接口的硬盘，这时候，对硬盘的读取操作，必须是一块一块的进行，比如，以 512 字节为一个块单元。这样如果应用程序采用下列形式调用 ReadFile 函数：

```

ReadFile((__COMMON_OBJECT*)&IOManager,
        lpFileObj,
        1024,
        lpBuffer,
        &dwReadSize);

```

就会出现一个问题：应用程序请求的数据大小，比物理设备所支持的块大小要大。这种情况下，ReadFile 就需要把这种大于设备物理块请求，进行分片，依次调用 DeviceRead 函数，来完成读取操作。这样在 ReadFile 函数中，就需要进行额外的处理，来适应这种情况，这样，ReadFile 的完整实现，如下：

```

BOOL ReadFile(__COMMON_OBJECT* lpThis, __COMMON_OBJECT* lpFileObj,

```

```

        DWORD dwByteSize,LPVOID lpBuffer,DWORD* lpdwReadSize)
{
    BOOL                                bResult                = FALSE;
    __DRIVER_OBJECT*                   lpDrvObj                = NULL;
    __DEVICE_OBJECT*                   lpDevObj                = NULL;
    __DRCB*                            lpDrcb                 = NULL;
    DWORD                              dwBlockSize             = 0L;
    LPVOID                             lpTmpBuffer            = NULL;
    LPVOID                             lpTmp                  = NULL;
    DWORD                              dwOrginalSize           = dwByteSize;

    if((NULL == lpThis) || (NULL == lpFileObj) || (0 == dwByteSize) || (NULL ==
lpBuffer))

        goto __TERMINAL;
    lpDevObj = (__DEVICE_OBJECT*)lpFileObj;
    lpDrvObj = lpDevObj->lpDriverObject;

    dwBlockSize = lpDevObj->dwMaxReadSize;
    dwByteSize = (0 == dwByteSize % dwBlockSize) ? dwByteSize :
(dwByteSize + dwBlockSize - (dwByteSize % dwBlockSize));

    lpDrcb = ObjectManager.CreateObject(&ObjectManager,
        NULL,
        OBJECT_TYPE_DRCB);
    if(NULL == lpDrcb)    //Can not create DRCB object.
        goto __TERMINAL;
    lpTmpBuffer = MemAlloc(KMEM_SIZE_TYPE_ANY,dwByteSize);
    if(NULL == lpTmpBuffer)
        goto __TERMINAL;
    lpTmp = lpTmpBuffer;    //In order to destroy this buffer,first save it,because the
lpTmpBuffer maybe changed in this routine.
    if(lpdwReadSize)
        *lpdwReadSize = 0;

```

```

while(dwByteSize)
{
    lpDrcb->dwStatus = DRCB_STATUS_PENDING;
    lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_READ;
    lpDrcb->dwOutputLen = dwBlockSize;
    lpDrcb->lpOutputBuffer = lpTmpBuffer;
    lpTmpBuffer += dwBlockSize;    //Adjust the buffer.
    if(!lpDrvObj->DeviceRead((__COMMON_OBJECT*)lpDrvObj,
                            (__COMMON_OBJECT*)lpDevObj,
                            lpDrcb))

        break;

    dwByteSize -= dwBlockSize;    //Read next block.
    *lpdwReadSize += dwBlockSize;
}

if(0 == dwByteSize)    //Read successfully.
{
    MemCopy(lpBuffer,lpTmp,dwOriginalSize); //Save the read data to application's
buffer.

    *lpdwReadSize = dwOriginalSize;
    bResult = TRUE;
    goto __TERMINAL;
}

if(0 == *lpdwReadSize)    //Can not read any data.
    goto __TERMINAL;

//
//Read several blocks,but not all acquired blocks,in case of reach the end of device.
//

MemCopy(lpBuffer,lpTmp,*lpdwReadSize); //Save the read data to applications's
buffer.

bResult = TRUE;
goto __TERMINAL;

__TERMINAL:

```



其中, `lpThis` 参数指向 `IOManager`, `lpFileObj` 是一个指向打开的设备对象(或文件对象)的指针, `dwWriteSize` 参数指定了希望写入目标设备或文件的字节的数量, 而 `lpBuffer` 参数则存储了具体的写入内容。 `lpWrittenSize` 参数则是一个返回参数, 函数成功返回(返回 `TRUE`)后, 该参数指定了实际写入的字节数量。正常情况下, 可能该参数的返回值等于 `dwWriteSize`, 但如果出现设备已经满, 或者到达文件结尾的情况, 则可能只写入了一部分内容, 这个时候, `lpWrittenSize` 参数告诉调用者, 多少数据被写入了目标设备或文件。

与 `ReadFile` 函数一样, 该函数通过 `lpFileObj` 参数, 找到该设备对象所对应的设备驱动程序对象, 然后调用设备驱动程序对象提供的 `DeviceWrite` 函数, 如下:

```
__DRIVER_OBJECT* lpDrvObj = NULL;
lpDrvObj = (__DEVICE_OBJECT*)lpFileObj->lpDriverObject;
bResult = lpDrvObj->DeviceWrite(lpDrvObj, lpFileObj, lpDrcb);
```

其中, `lpDrcb` 是一个指向 `DRCB` 对象的指针, 该对象被 `WriteFile` 函数创建, 用于传递参数、跟踪请求过程。代码如下:

```
__DRCB*    lpDrcb = NULL;
lpDrcb = ObjectManager.CreateObject(&ObjectManager,
                                     NULL,
                                     OBJECT_TYPE_DRCB);
If(NULL == lpDrcb)    //Can not create DRCB object.
    goto __TERMINAL;

lpDrcb->dwDrcbStatus = DRCB_STATUS_PENDING;    //Pend the DRCB object.
lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_WRITE;    //Read operation.
lpDrcb->dwInputLen = dwWriteSize;
lpDrcb->dwInputBuffer = lpBuffer;
```

与 `ReadFile` 不同的是, `DRCB` 对象的 `dwRequestMode` 被初始化为 `DRCB_REQUEST_MODE_WRITE`, 用来指明这是一个写操作。

与 `ReadFile` 一样, `WriteFile` 也会存在写入字节数量跟目标设备块大小不一致, 需要协调的情况, 也一样会出现不完全写入(只写入一部分)的情况, 因此, `WriteFile` 也必须象 `ReadFile` 一样, 综合考虑这些情况。下面是 `WriteFile` 的完整实现代码:

```

BOOL WriteFile(__COMMON_OBJECT* lpThis,__COMMON_OBJECT* lpFileObj,
               DWORD dwWriteSize,LPVOID lpBuffer,DWORD* lpdwWrittenSize)
{
    BOOL                                bResult                = FALSE;
    __DRIVER_OBJECT*                   lpDrvObj                = NULL;
    __DEVICE_OBJECT*                   lpDevObj                = NULL;
    __DRCB*                            lpDrcb                 = NULL;
    DWORD                              dwBlockSize             = 0L;
    LPVOID                             lpTmpBuffer            = NULL;
    LPVOID                             lpTmp                   = NULL;
    DWORD                              dwOrginalSize           = dwWriteSize;

    if((NULL == lpThis) || (NULL == lpFileObj) || (0 == dwWriteSize) || (NULL ==
lpBuffer))

        goto __TERMINAL;
    lpDevObj = (__DEVICE_OBJECT*)lpFileObj;
    lpDrvObj = lpDevObj->lpDriverObject;

    dwBlockSize = lpDevObj->dwMaxWriteSize;
    dwWriteSize = (0 == dwWriteSize % dwBlockSize) ? dwWriteSize :
(dwWriteSize + dwBlockSize - (dwWriteSize % dwBlockSize));

    lpDrcb = ObjectManager.CreateObject(&ObjectManager,
                                       NULL,
                                       OBJECT_TYPE_DRCB);
    if(NULL == lpDrcb)    //Can not create DRCB object.
        goto __TERMINAL;
    lpTmpBuffer = MemAlloc(KMEM_SIZE_TYPE_ANY,dwWriteSize);
    if(NULL == lpTmpBuffer)
        goto __TERMINAL;
    lpTmp = lpTmpBuffer;    //In order to destroy this buffer,first save it,because the
lpTmpBuffer maybe changed in this routine.
    if(lpdwReadSize)

```

```

        *lpdwReadSize = 0;
        MemCopy(lpTmpBuffer,lpBuffer,dwOriginalSize);    //Copy the content to be written to
a new buffer.

```

```

        while(dwWriteSize)
        {
            lpDrcb->dwStatus = DRCB_STATUS_PENDING;
            lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_WRITE;
            lpDrcb->dwInputLen = dwBlockSize;
            lpDrcb->lpInputBuffer = lpTmpBuffer;
            lpTmpBuffer += dwBlockSize;    //Adjust the buffer.
            if(!lpDrvObj->DeviceWrite((__COMMON_OBJECT*)lpDrvObj,
                                    (__COMMON_OBJECT*)lpDevObj,
                                    lpDrcb))

                break;

            dwWriteSize -= dwBlockSize;    //Read next block.
            *lpdwWrittenSize += dwBlockSize;
        }

        if(0 == dwWriteSize)    //Write successfully.
        {
            *lpdwReadSize = dwOriginalSize; //Set returned written size to original request
size.

            bResult = TRUE;
            goto __TERMINAL;
        }
        if(0 == *lpdwReadSize)    //Can not read any data.
            goto __TERMINAL;

        //
        //Written several blocks,but not all acquired blocks,in case of reach the end of device.
        //

        bResult = TRUE;
        goto __TERMINAL;

```

```

__TERMINAL:
    if(!lpDrcb)
        ObjectManager.DestroyObject(&ObjectManager,
                                     (__COMMON_OBJECT*)lpDrcb);    //Destroy
DRCBObject.
    if(lpTmp)
        KMemFree((LPVOID)lpTmp,KMEM_SIZE_TYPE_ANY,0L);
    return bResult;
}

```

该函数的实现中，首先把要写入的字节数，舍入到目标设备的最大写入块的倍数，然后创建一个新的缓冲区，并把原始缓冲区的内容，拷贝到新的缓冲区，之所以这样做，是因为实际在把写入字节舍入为块的倍数的时候，可能会比原始请求的字节数大（比如，原始请求写入的字节数为 108，而块设备的最大写入块大小为 256，则舍入的写入块大小将为 256）。这种情况下，若仍然使用原来的缓冲区，则可能会出现“超读”的现象，假设即原始缓冲区大小为 108 字节，而实际写入的时候，需要读取 256 字节（一个块），这样可能会出现异常。因此，需要 WriteFile 自己创建一个跟舍入写入尺寸相同的缓冲区，然后把用户请求数据复制到自己创建的缓冲区，在实际写入（调用 DeviceWrite 函数）的时候，直接从 WriteFile 自己创建的缓冲区内读取，而不用读取调用者提供的缓冲区。

#### 10.1.4.9 CloseFile 的实现

CloseFile 函数用来关闭打开的设备或文件，与 CreateFile 的功能相反，该函数的原型如下：

```

VOID                                     (*CloseFile)(__COMMON_OBJECT* lpThis,
                                     __COMMON_OBJECT*
lpFileObj);

```

在目前 Hello China 的实现中，该函数没有做很多的工作，而近近是把设备对象的引用计数递减，然后直接返回。实现如下：

```

VOID CloseFile(__COMMON_OBJECT* lpThis,__COMMON_OBJECT* lpFileObj)

```

```

{
    DWORD                dwFlags = 0L;
    __DEVICE_OBJECT* lpDevObj = (__DEVICE_OBJECT*)lpFileObj;
    if(NULL == lpDevObj)
        return;
    __ENTER_CIRITICAL_SECTION(NULL,dwFlags);
    lpDevObj->dwRefCounter --;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return;
}

```

#### 10.1.4.10 IOControl 的实现

ReadFile 和 WriteFile 可以用来完成对设备的读写操作，这对于大多数的存储设备，或许已经足够了，但对于一些其它类型的设备，只采用这两个函数抽象所有的操作，可能会力不从心，比如媒体播放设备，提供了快进、倒退功能，这样的功能，通过读写的操作，是无法实现控制的，因此还必须实现另外的接口。另外，还有一些功能未知的设备，操作系统无法事先预知设备功能，因此也就无法预先定义接口。这种情况下，就需要操作系统定义一种“透传”接口，把应用程序请求的操作，直接透传给设备驱动程序，操作系统不做任何功能上的操作。IOControl 函数就是为这种需求而提出的，这个函数的原型如下：

```

        BOOL                (*IoControl)(__COMMON_OBJECT*  lpThis,
                                           __COMMON_OBJECT*
lpFileObj,
                                           DWORD
dwCommand,
                                           DWORD
dwInputLen,
                                           LPVOID
lpInputBuffer,
                                           DWORD
lpOutputLen,
                                           LPVOID

```

lpOutBuffer);

lpFileObj 函数是已经打开的设备对象，dwCommand 函数则是设备特定的命令代码（操作系统对该参数的含义一无所知），lpInputBuffer 和 lpInputLen 共同确定了一个输入缓冲区，作为 dwCommand 命令代码的输入参数，lpOutputLen 参数和 lpOutBuffer 参数共同确定了一个缓冲区，该缓冲区用来保存输出结果。

在 Hello China 当前的实现中，对于该函数，只做了一个参数转换功能（把输入的参数，通过一个 DRCB 对象统一管理），然后直接调用设备驱动程序对象提供的 IOControl 函数。代码如下：

```

BOOL IOControl(__COMMON_OBJECT* lpThis,
               __COMMON_OBJECT* lpFileObj,
               DWORD dwCommand,
               DWORD dwInputLen,
               LPVOID lpInputBuffer,
               DWORD dwOutputLen,
               DWORD dwOutputBuffer)
{
    __DEVICE_OBJECT* lpDevObj = (__DEVICE_OBJECT*)lpFileObj;
    __DRIVER_OBJECT* lpDrvObj = NULL;
    __DRCB*          lpDrcb  = NULL;

    if(NULL == lpDevObj)    //Invalid parameter.
        return FALSE;

    lpDrcb = ObjectManager.CreateObject(&ObjectManager,
                                       NULL,
                                       OBJECT_TYPE_DRCB);

    if(NULL == lpDrcb)
        return FALSE;

    lpDrcb->dwStatus = DRCB_STATUS_PENDING;    //The DRCB object will be
    pending.

    lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_IOCTL;
    lpDrcb->dwCtrlCommand = dwCommand;
    lpDrcb->dwInputLen = dwInputLen;

```

```

    lpDrcb->lpInputBuffer = lpInputBuffer;
    lpDrcb->dwOutputLen = dwOutputLen;
    lpDrcb->lpOutputBuffer = lpOutputBuffer;

    lpDrvObj = lpDevObj->lpDriverObject;
    return lpDevObj->IoControl(lpDrvObj,
                               lpDevObj,
                               lpDrcb);
}

```

为了处理上的简便，DRCB 对象的定义中，专门预留了一个成员 dwCtrlCommand，用来为 IoControl 函数服务，该变量直保存了 IoControl 调用者给定的操作命令。

#### 10.1.4.11 SetFilePointer 的实现

该函数用来指定新的读写位置，比如，在文件系统的实现中，对于每个打开的文件，都有一个当前位置指针与之对应，这样每次读取或写入一定数量的字节，指针跟着向前移动对应数量的字节，这样的实现是合理的。但有的时候，在写入文件或设备的时候，需要写入特定的位置，这样就需要通过该函数，来有目的的移动文件指针。该函数的原型如下：

```

BOOL                                     (*SetFilePointer)(__COMMON_OBJECT*
lpThis,
                                     __COMMON_OBJECT*
lpFileObj,
                                     DWORD
dwWhereBegin,
                                     INT      dwOffset);

```

参数中，lpFileObj 是已经打开的设备或文件的指针，dwWhereBegin 是一个起始位置标识符，可以取下列各值：

- FILE\_POSITION\_CURRENT: 从当前位置开始移动；
- FILE\_POSITION\_BEGIN: 从文件或设备的起始位置开始移动；
- FILE\_POSITION\_END: 从文件或设备的结束位置开始移动。

而 dwOffset 参数则指定了具体的移动字节数量，而移动的方向，则是从文件或对象



```

if(NULL == lpDrcb) //Can not create DRCB object.
    return;
lpDrcb->dwStatus = DRCB_STATUS_PENDING;
lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_FLUSH;
lpDrvObj->DeviceFlush(lpDrvObj,
                      lpDevObj,
                      lpDrcb);

return;
}

```

#### 10.1.4.13 IOManager 对设备驱动程序的接口

下列函数供设备驱动程序调用：

- ✓ **CreateDevice**，该函数创建一个设备对象，并根据函数参数完成初步的初始化功能，一般情况下，设备驱动程序加载完毕，进入初始化阶段（DriverEntry 函数）之后，设备驱动程序会检测设备，根据检测结果，来创建相应的设备对象。比如，网卡驱动程序被加载之后，驱动程序会检测系统上是否安装了网卡，如果能够检测到网卡，那么驱动程序会创建一个网卡设备对象；
- ✓ **DestroyDevice**，该函数销毁 CreateDevice 函数创建的设备对象。

为了进一步理解上述几个函数的功能，下面描述一个比较典型的设备驱动程序加载、初始化过程，假设设备驱动程序为硬盘驱动程序：

- 1、操作系统加载硬盘驱动程序文件，并完成诸如重定位等工作；
- 2、IOManager 调用硬盘驱动程序的入口函数（DriverEntry），硬盘驱动程序进入初始化工作；
- 3、在硬盘驱动程序的 DriverEntry 函数内部，会检测系统的硬盘安装情况，比如安装硬盘的个数、每个硬盘的分区情况等；
- 4、根据检测结果，预留系统资源（通过调用 ReserveResource 函数），有的情况下，IOManager 会通过 DriverEntry 函数传递给驱动程序相应的设备资源，这种情况下，驱动程序必须使用系统分配的资源，但也必须显示的通过 ReserveResource 预留系统分配的资源，算是一个资源确认操作；
- 5、根据检测的结果，调用 CreateDevice 创建相应的设备对象；
- 6、如果上述过程一切顺利，初始化结束，设备可以使用。

#### 10.1.4.14 驱动程序入口 (DriverEntry)

驱动程序被加载到内存后，IOManager 首先通过某种方式（具体参考下面的章节），找到一个所谓“入口函数”的地址，然后调用这个函数。这个函数就是所谓的驱动程序“入口”。

驱动程序的入口原型如下：

```

        BOOL                                     DriverEntry(__DRIVER_OBJECT*
        lpDriverObject, __RESOURCE_DESCRIPTOR* lpResDesc);

```

其中，lpDriverObject 是 IOManager 创建的一个驱动程序对象，而 lpResDesc 则是描述系统资源的数据结构指针。

该函数的具体实现，是由驱动程序本身完成的，一般情况下，驱动程序可以在这个函数内初始化全局数据结构，创建设备对象（根据 IOManager 分配的系统资源），并设置驱动程序对象的一些变量（比如，各个函数指针等），如果该函数成功执行，那么返回 TRUE，这个时候，IOManager 就认为驱动程序初始化成功，否则，返回 FALSE，那么 IOManager 就会认为驱动程序初始化失败，于是就卸载掉该驱动程序，释放创建的驱动程序对象。

#### 10.1.4.15 设备驱动程序的卸载

所谓设备驱动程序卸载，指的是把不再使用的驱动程序从内存中删除掉，以释放内存，供其它应用程序适应。设备驱动程序的卸载发生在操作系统关闭、设备消失（被拔出等）等情况下，在设备驱动程序被卸载的时候，系统（IOManager）调用设备驱动程序的 UnloadEntry 函数，该函数释放驱动程序申请的资源。

从 UnloadEntry 返回后，IOManager 会删除该驱动程序对应的驱动程序对象。

## 10.2 文件系统的实现

在 Hello China 驱动程序管理框架的实现中，把文件系统的实现也纳入驱动程序管理框架范围之内，把文件系统作为一种特殊的设备驱动程序来看待。实际上，IOManager 提供给用户的接口（CreateFile、ReadFile 等）可以直接用来打开、读写文件（从名字的命名上也可以看出）。

在本节中，我们对文件系统的实现，以及跟设备驱动程序管理框架的融合，进行详

细的描述。

### 10.2.1 文件系统与文件的命名

在当前版本的 **Hello China** 的实现中，采用类 **Windows** 文件命名格式，即使用英文字母（**A**、**B**、**C**、... ...）加上冒号（**:**）来标识一个物理卷，一个物理卷可以是一个硬盘的分区，也可以是一个硬盘，甚至可以是多个硬盘分区（或多个硬盘）的逻辑组合。比如，系统中第一个硬盘分区为 **C:**，第二个硬盘分区为 **D:**，第三个为 **C:**，.....

对于分区上的文件，分两类对待：一类是目录，这类文件可以理解为一个容器，里面装载了文件，目录文件的内容即是容器内部文件的文件名，比如，一个目录下有三个文件：

file1.dat  
file2.dat  
file3.dat

那么，该目录文件的文件内容可能是这个样子：

File	File1 的控制信息	File	File2 的控制信	File	File
------	-------------	------	------------	------	------

图 10-6 目录文件的内容

即目录文件也作为普通文件对待，不同的是，目录文件的内容，就是存在于目录下面实际文件（也可能包含目录）的文件名（以及其它控制信息）的集合。

目录可以是嵌套的，比如，一个目录文件（假设为 **Directory1**）下面，包含了另外三个文件和一个目录文件（假设为 **Directory2**），**Directory2** 下面又包含了一个数据文件 **file1.dat**，而且假设这些文件都位于系统中第二个分区上（相应的表示符为 **D:**），那么，**file1.dat** 可以这样表示：

D:\Directory1\Directory2\file1.dat

在当前版本的实现中，一个文件的名称，可以由字母和数字组成，也可以由汉字组成，文件名可以使用点（.）来分割成几个部分，最后一部分成为文件的扩展名，一般情况下，文件的扩展名不超过四个字节。当前情况下，下列字符不能出现在文件的命名中：

/\ = \* ?

### 10.2.3 文件系统驱动程序

在当前版本的 Hello China 的实现中，文件系统作为一种特殊类型的设备驱动程序来实现，这样文件系统就可以统一纳入 IOManager 的管理框架，而且对用户来说，与普通的设备驱动程序是一致的，所不同的是，文件系统驱动程序可能又调用了实际的存储设备驱动程序提供的功能，来访问存储设备驱动程序。

文件系统驱动程序的加载，是在普通设备驱动程序加载之后，再进行的。一般情况下，IOManager 首先检测系统总线，判断哪些设备连接到了系统总线上，然后配置相应的资源，并加载设备驱动程序，等这个过程完成之后，IOManager 会遍历所有加载的设备驱动程序创建的设备对象，来判断该设备对象是存储设备还是非存储设备。如果是存储设备，那么 IOManager 会调用适当的函数来读取该存储设备的一些头部信息（比如，一个硬盘分区的前几个扇区），根据存储设备的头部信息，IOManager 就会判断出该存储设备被格式化成的文件系统格式（FAT、FAT32、NTFS、EXT2 等），根据判断的结果，再加载合适的文件系统驱动程序。

IOManager 在把文件系统驱动程序加载到内存之后，会使用合适的参数调用文件系统驱动程序的 DriverEntry 函数，给文件系统驱动程序一个机会，完成自己的初始化工作，然后，IOManager 调用文件系统驱动程序提供的 CreateFileSystem 函数（每个文件系统驱动程序都需要提供该函数），来创建一个文件系统设备对象，一些存储设备相关的参数（比如，该存储设备位于系统中的第几块硬盘上，是该硬盘的第几个逻辑分区，该逻辑分区的起始扇区号、扇区数量、每扇区的大小等）会通过 CreateFileSystem 的参数传递给文件系统驱动程序，这样，文件系统驱动程序就会根据传递过来的资源，调用 CreateDevice 函数来创建一个设备对象（该设备对象的类型是 DEV\_TYPE\_FILE\_SYSTEM），这样就完成了该存储设备文件系统的初始化工作。

需要注意的是，IOManager 只有在检测到后续同类型的存储设备的时候，就不需要再加载文件系统驱动程序了，因为该驱动程序已经被加载并初始化，IOManager 只要直接调用 CreateFileSystem 函数，创建另外的文件系统设备对象即可。

## 10.2.4 打开一个文件的操作流程

在 Hello China 当前版本的实现中，把文件跟普通的设备同等对待，即每当打开一个文件，在操作系统核心内部，实际上是增加了一个设备对象（文件设备对象），对于文件的读写等操作，直接调用设备对象管理的驱动程序提供的服务函数。

用户通过调用 IOManager 提供的 CreateFile 函数来打开一个文件，该函数的参数指明了要打开的文件名、打开方式（只读、读写等）等，IOManager 按下列步骤进行操作：

- 1、首先根据文件名以及命名规范，来确定请求打开的对象是文件还是普通的设备对象（文件名是以文件系统标识符开头的，而设备对象则以字符串 [\\.\devicename](#) 开头）；
- 2、如果判断结果是普通的设备对象，则转到设备对象打开流程（参考其它章节）；
- 3、如果判断结果是文件对象，则 IOManager 首先遍历系统中已经打开的设备链表，用文件名来匹配每个设备名字，如果匹配成功，则说明了该文件已经打开，于是进一步检查该文件的打开方式（先前已经打开的方式），如果允许按照本次请求的打开方式重复打开，则直接给用户返回已经打开的文件对象的地址，如果不允许重复打开，则返回失败标志；
- 4、如果遍历打开设备的链表后，没有找到对应的文件，则说明该文件没有被打开，于是从文件名中提取文件系统标识符（比如，C:,D:,E:等）；
- 5、根据文件系统标识符，来遍历打开的设备链表（实际上，在实现的时候，这两个遍历可以同步执行，在这里是为说明上的方便），尝试寻找对应的文件系统对象（文件系统也作为一个普通设备对象对待）；
- 6、如果不能找到对应的文件系统对象，则说明目前系统中不存在这个文件，返回失败标志；
- 7、如果可以找到对应的文件系统对象，则 IOManager 调用文件系统对象的 DeviceOpen 函数（以文件名或 DRCB 为参数），来打开该文件；
- 8、如果文件系统对象的 DeviceOpen 函数执行成功，则返回被打开文件的句柄，否则返回一个失败标志（NULL），在这里，文件系统对象进一步调用存储设备对象的相应函数，来读取实际的设备数据；
- 9、CreateFile 根据 DeviceOpen 函数的返回结果，来给初始调用返回适当的数值。

上面的描述，是为了方便理解起见，进行的一个简化版的描述，实际上，在实现的时候，上述所有过程只需要一个遍历操作就可以完成，在开始的时候，CreateFile 首先确定打开的文件是文件，还是普通的设备，如果是普通的设备，则采用设备名直接查找设备链表，采用精确匹配，反之，如果是文件，则在查询设备链表的时候，采用的是最长匹配，因为这样可以大大提高打开效率。比如，设备链表中有下列已经打开的文件（或设备）：

E:

E:\DATA\DATA1.BAT

E:\DATA

这个时候，如果需要打开文件 E:\DATA\DATA2.DAT，则最长匹配的结果为 E:\DATA，于是系统调用这个设备对象的 DeviceOpen 函数，并以该对象的基地址为参数，来打开目标文件。再比如，如果要打开文件 E:\SYS\DATA.DAT，则最长匹配的结果是 E:（文件系统设备对象），然后调用这个设备对象的 DeviceOpen 函数，来打开目标文件。

## 10.3 设备驱动程序框架

### 10.3.1 设备请求控制块（DRCB）

设备请求控制块（Device Request Control Block）是 Hello China 的 I/O 管理框架中的核心数据结构（对象），该对象用来跟踪所有对设备的请求操作，一般由 IOManager 创建，然后通过设备驱动程序提供的服务函数（比如，DeviceRead、DeviceWrite 等）传递给设备驱动程序，然后设备驱动程序根据 DRCB 里面的参数，就可以确定本次操作的一些特定数据，比如，设备读取的开始地址，读取数据的长度，以及数据读取后应存放的缓冲区位置等，可以认为，DRCB 是 Hello China 设备驱动管理框架的核心。

该对象（数据结构）的定义如下：

```
BEGIN_DEFINE_OBJECT(__DRCB)
    INHERIT_FROM_COMMON_OBJECT
    __EVENT*                lpSynObject; //Synchronization object.
    __KERNEL_THREAD_OBJECT* lpKernelThread; //The kernel thread
originates this I/O request.
    DWORD                   dwDrcbFlag;
    DWORD                   dwStatus;
    DWORD                   dwRequestMode;
//Read,Write,Control,etc.
    DWORD                   dwCtrlCommand;

//Output buffer's information.
    DWORD                   dwOffset;
```

```

        DWORD                dwOutputLen;
        LPVOID                lpOutputBuffer;

//Input buffer(parameters)'s information.
        DWORD                dwInputLen;
        LPVOID                lpInputBuffer;

        __DRCB*               lpNext;
        __DRCB*               lpPrev;

        DRCB_WAITING_ROUTINE  WaitForCompletion;
        DRCB_COMPLETION_ROUTINE OnCompletion; //Called when request is
completed.
        DRCB_CANCEL_ROUTINE   OnCancel; //Called when request is
canceled.

        DWORD                DrcbExtension[0];
END_DEFINE_OBJECT();

```

在这个对象的定义中，大多数成员意义很明确，下列三个数据成员需要着重说明一下：

- 1、WaitForCompletion;
- 2、OnCompletion;
- 3、OnCancel。

这三个都是函数指针，指向了驱动程序管理框架实现的三个函数，这三个函数分别被驱动程序调用。其中，第一个函数（WaitForCompletion）用于等待请求的操作完成，比如，设备驱动程序根据 DRCB 对象提供的信息，提交了一个物理设备读取请求，由于这个物理设备的执行速度比 CPU 慢很多，因此，设备驱动程序不能采用等待的方式，来等待设备操作完成，而是采用一种中断的方式，即设备操作完成之后，通过中断的方式通知设备驱动程序，然后设备驱动程序再采取进一步的动作。而这个函数（WaitForCompletion）就是用于这个目的，该函数调用后，相应的用户线程（发起 IO 请求的线程）就会进入阻塞状态，直到对应的操作完成。这样做的好处是，大大节约了 CPU 的资源，如果不采用这种方式，而是采用一种忙等待的方式等待设备操作完成，那么会

浪费很多 CPU 资源。

第二个函数是跟第一个函数对应的，这个函数在设备驱动程序完成设备操作后调用，一般情况下是在设备驱动程序的中断处理中调用的，这个函数执行后，就会唤醒原来等待 IO 操作完成的线程（投入到 Ready 队列），这样当下一次调度的时候，如果这个线程（发起 IO 请求的线程）的优先级足够高，那么该线程就可以继续执行了。

为了进一步理解上述过程和配合关系，我们举一个硬盘读写的例子进行说明，假设用户线程想读取一个文件，于是用户线程发起了一个 ReadFile 的函数调用，后续的执行过程如下：

- 1、IOManager 创建一个 DRCB 对象，根据 ReadFile 函数的参数对该对象进行初始化，然后再根据文件名，找到合适的文件对象（其实是一个设备对象），调用设备对象所对应的驱动程序（文件系统驱动程序）的相应函数（DeviceRead 函数，以创建的 DRCB 对象为参数）；
- 2、文件系统程序根据 ReadFile 传递过来的 DRCB 对象，以及设备对象的设备扩展，找到实际的硬盘设备对象，然后文件系统驱动程序另外创建一个 DRCB 对象，初始化这个 DRCB 对象，再以这个新创建的 DRCB 对象为参数，来调用硬盘设备对象的 DeviceRead 函数；
- 3、硬盘设备对象的 DeviceRead 函数根据传递过来的 DRCB 对象，初始化一个硬盘读请求事务，并把该 DRCB 对象放到驱动程序的等待队列中，然后调用 DRCB 对象中的 WaitForCompletion 函数；
- 4、硬盘驱动器（控制器）执行实际的读操作，完成以后，给 CPU 发一个中断；
- 5、中断调度机制根据中断号调用实际的中断处理函数（硬盘驱动程序的中断处理函数），中断处理函数从中断控制器读取数据，填充在 DRCB 指定的缓冲区内，然后把该 DRCB 对象从等待队列中删除，并调用 OnCompletion 函数；
- 6、OnCompletion 函数唤醒等待的线程（返回到硬盘驱动程序的 DeviceRead 函数继续执行），于是 DeviceRead 函数把从硬盘上读取的数据填充到 IOManager 发送过来的 DRCB 对象中，销毁自己创建的 DRCB，并返回；
- 7、IOManager 把从硬盘读取的数据填充到用户线程指定的缓冲区内，然后从 ReadFile 函数返回。

可以看出，这个过程比较复杂，而且在上面的描述中，我们省略了数据尺寸不匹配的情况（比如，用户请求 4K 的数据，而硬盘驱动程序一次只能读取一个扇区的字节，一般情况下为 512byte，这种情况下，就需要文件系统驱动程序对原始请求进行分割），实际情况可能比上述情况更加复杂。

最后一个函数用于取消一个 IO 请求。一般情况下，设备驱动程序可能维护了多个 IO 请求任务（比如，系统中多个线程同时读取同一个硬盘上的文件），这些 IO 请求任务使

用 DRCB 对象进行跟踪，并被设备驱动程序以队列的形式进行维护。这样可能出现一种情况，就是一个请求任务可能被取消（比如，对应的用户线程取消了读取请求），这个时候，设备驱动程序就可以直接把对应的 DRCB 对象从等待队列中删除，然后调用 OnCancel 函数，来通知上层模块（IOManager）这个取消请求动作。在目前版本的 Hello China 的实现中，暂不支持 IO 请求的取消服务。

需要说明的是，上述三个函数的参数，都是对应的 DRCB 对象，比如，可以这样调用 OnCompletion 函数：

```
lpDrcb->OnCompletion((__COMMON_OBJECT*)lpDrcb);
```

由于 DRCB 对象中包含了发起该 IO 请求的线程对象（lpKernelThread 成员）和一个事件同步对象（lpSynObject），所以这些函数可以很容易的实现线程的阻塞、唤醒等操作。对于一些特定功能的参数，可以通过 DRCB 对象的 DrcbExtension 成员访问，该成员用于访问一些驱动程序特定的参数，类似于设备对象的设备扩展。

另外，在 \_\_DRCB 对象的定义中，为了标志 DRCB 对象的状态，定义了 dwDrcbStatus 变量，这个变量可以取下列值：

- **DRCB\_STATUS\_INITIALIZED**: DRCB 对象已经被初始化，但尚未被任何线程应用，一般情况下，调用 CreateObject 函数，创建一个 DRCB 对象后，所创建的 DRCB 对象状态被设置为该值；
- **DRCB\_STATUS\_PENDING**: DRCB 处于排队状态，等待对应的操作完成。比如，读取磁盘上的一个数据块，相应的设备操作命令已经发出，但还没有收到最终相应，这个时候，dwDrcbStatus 被设置为该值；
- **DRCB\_STATUS\_COMPLETED**: DRCB 对象跟踪的设备请求操作已经被成功完成；
- **DRCB\_STATUS\_FAILED**: DRCB 对象跟踪的设备请求操作失败，比如，设备在长时间内没有响应，会导致这种情况出现；
- **DRCB\_STATUS\_CANCELED**: DRCB 对象跟踪的设备请求，在完成前被用户取消。比如，用户读取一个硬盘上的数据，驱动程序已经发出请求（这时候，DRCB 对象的状态设置为 DRCB\_STATUS\_PENDING），在完成前，用户取消了该读取操作，这时候，操作系统会把该 DRCB 对象的状态设置为 DRCB\_STATUS\_CANCELED。

另外一个比较重要的成员变量，是 dwRequestMode，指明了该 DRCB 跟踪的请求类型，可以取下列值：

- **DRCB\_REQUEST\_MODE\_READ**: 该 DRCB 对象跟踪的请求类型是一个读取操作，比如，读取存储设备上的一块数据；
- **DRCB\_REQUEST\_MODE\_WRITE**: 对应写入设备的操作，欲写入设备的具体数据，

由 dwInputLen 和 lpInputBuffer 两个参数指定；

- DRCB\_REQUEST\_MODE\_CONTROL: IO Control 操作，dwCtrlCommand 指明了具体的操作类型，一般情况下，dwCtrlCommand 取值的具体含义，由设备驱动程序自己定义；
- DRCB\_REQUEST\_MODE\_SEEK: 在调用 DeviceSeek 函数的时候，设定该值；
- DRCB\_REQUEST\_FLUSH: 在调用 DeviceFlush 函数的时候，设定该值。

### 10.3.2 设备驱动程序的文件组织结构

设备驱动程序编译后，以文件的形式存在系统存储设备（比如，硬盘）上，下图示意了典型的驱动程序文件组织结构：

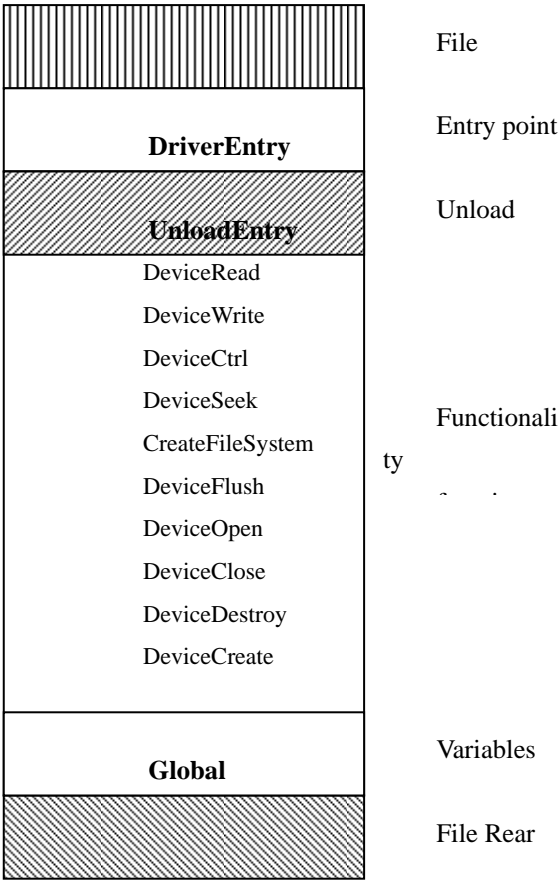


图 10-7 设备驱动程序文件的存储结构

其中，DriverEntry 和 UnloadEntry 是用于初始化和资源释放调用的，其它的函数和全局变量，用于实现该设备驱动程序的特定功能。相应地，DriverEntry 和 UnloadEntry 是直接输出的函数，而其它的函数，则在 DriverEntry 中隐式的输出给 IOManager（参见 DriverEntry 的实现部分）。

### 10.3.3 设备驱动程序的功能实现

从上面的设备驱动程序文件组织结构中看出，设备驱动程序实现了一个统一的设备操作框架（一组标准函数），这些标准框架函数由 IOManager 调用（设备驱动程序在初始化时，通过 DriverEntry 函数通知 IOManager 这些函数的地址，参考下面相关章节的描述），设备驱动程序实际对设备的操作，就是在这些框架函数中实现的。

一般情况下，对设备的操作可以抽象为读写操作和打开/关闭操作，对应框架中的 DeviceRead/DeviceWrite、DeviceOpen、DeviceClose 函数，但也有一些其它的操作，比如定位当前位置（DeviceSeek），特殊的控制命令（DeviceCtrl）等，因此，标准框架也为这些特殊的操作定义了接口。

对设备的打开和关闭操作相对比较简单，在设备驱动程序实现的时候，针对打开操作，一般是创建一个设备对象，初始化，并连接到操作系统（确切的说，是 IOManger）维护的设备对象链表中，对于关闭操作，设备驱动程序释放关闭设备所对应的系统资源，从系统设备链表中删除该设备对象。

比较复杂的是对设备的读写操作和控制操作（对应 DeviceRead/DeviceWrite/DeviceCtrl 函数），在本节中，我们对这几个操作进行比较详细的实现描述。需要说明的是，设备不同，这些函数实现的方式和具体功能也不同，在这里描述的是一个相对通用的框架，或者可以看作是一种实现的特例，作为实现具体的设备驱动程序时的参考。

#### 10.3.3.1 读操作（DeviceRead）的实现

读操作的发起者，可以是用户线程、系统线程，也可以是设备驱动程序（比如，文件系统驱动程序，就需要读硬盘数据），但不论是哪种方式，其入口却只有一个，即所有的读操作都通过 IOManager 提供的 ReadFile 函数来展现，当然，在读一个设备的时候，该设备必须已经打开（即建立了设备对象）。在这里，假设一个用户线程发起一个读请求操作，从串行接口读取一个字节的的数据，相应的流程如下：

- 1、用户线程调用 IOManager 提供的 ReadFile 函数（通过系统调用）；
- 2、ReadFile 函数根据用户提供的参数，创建一个 DRCB（Device Request Control Block）

对象，并初始化该对象，然后 `ReadFile` 函数根据用户提供的设备对象地址，进而找到该设备对象对应的设备驱动程序（设备对象维护了指向设备对象驱动程序的后向指针），调用设备驱动程序对象对应的函数，到此为止，所有的操作都是由操作系统核心完成的（确切的说，是 `IOManager`），下面的操作将由设备驱动程序自己完成：

- 3、设备驱动程序维护了一个设备请求控制对象队列（`DRCB` 队列），该队列中缓存了所有未完成的设备请求，当 `DeviceRead` 函数被调用后，该函数会检查队列的状态是否为空，如果是空，则该函数把该设备请求控制对象插入队列，然后根据 `DRCB` 提供的参数，发起一个设备操作请求（操作实际的设备），如果队列不为空，则说明现在仍然有一些请求正在执行中，于是该函数会把该 `DRCB` 对象插入队列，然后调用 `WaitForCompletion` 函数（该函数由 `IOManager` 提供，其指针保存在 `DRCB` 对象里面）；
- 4、在设备驱动程序对象发起一个设备请求操作后，会根据设备工作方式的不同（中断方式或轮询方式），来确定是否调用 `WaitForCompletion` 函数，如果是中断方式，则调用该函数，对应的线程进入阻塞状态，等待操作完成，如果是轮询方式，则不调用该函数，直接等待操作的完成（这个时候，其实是进入一个循环），在轮询方式下，设备操作完成或失败后，`DeviceRead` 函数填充 `DRCB` 提供的缓冲区，设置 `DRCB` 对应的状态字段，然后返回给调用者（`ReadFile` 函数）；
- 5、在中断方式下，由 `DeviceRead` 调用了 `WaitForCompletion` 函数，该函数会把当前线程挂起，等待设备操作完成。当设备操作完成之后，设备控制器会发起一个中断，通知操作系统该操作的完成，操作系统会调用该设备对应的中断处理程序，设备驱动程序的中断处理程序从 `DRCB` 队列中摘取一个 `DRCB` 对象，填充该对象（根据设备的操作结果），调用该对象的 `OnCompletion` 函数（该函数唤醒等待该操作的线程），然后检查 `DRCB` 队列是否为空，若是，则从中断中返回，否则，会从队列中获取一个 `DRCB` 对象，根据该对象指明的操作，再次发起一个设备操作，然后从中断中返回。下图反映了上述调用关系：

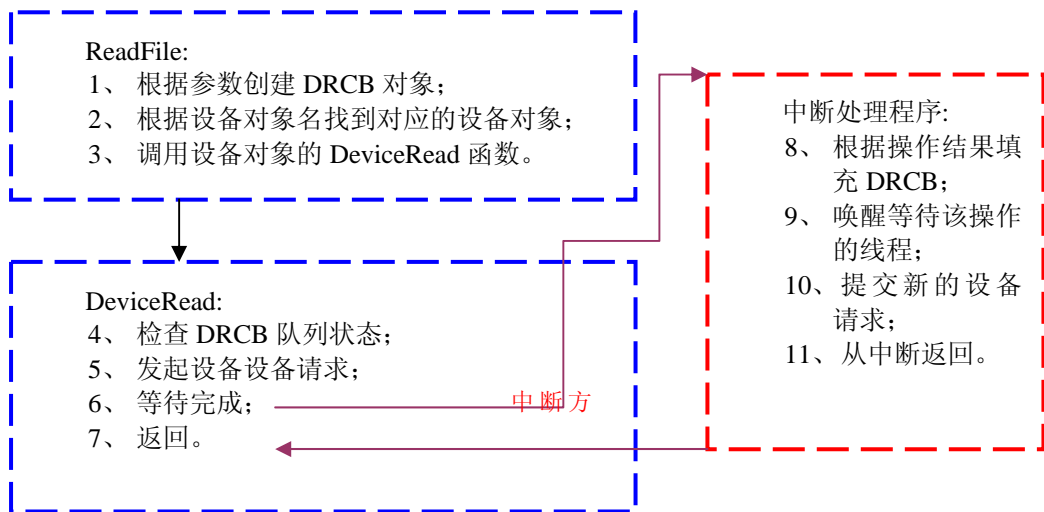


图 10-8 设备读取操作的组成步骤

上面描述的是没有缓冲的情况，实际上，为了提高访问速度，大多数的设备驱动程序提供了缓冲功能，在内存中创建缓冲区，用于缓存设备上的数据，当读请求到达时，设备驱动程序首先检查请求的内容是否位于本地缓冲区内，如果在，则直接从缓冲区内读出，这样可以大大提高读操作的速度。在实现缓冲的设备驱动程序中，上述流程略有不同，就是在上述第三步中，驱动程序首先检查本地缓冲区，如果读取的内容位于本地缓冲区内，则直接从缓冲区内读出，返回给用户，否则，再发起一个实际的设备操作。

### 10.3.3.2 写操作（DeviceWrite）的实现

写操作的过程，跟读操作基本一致，所不同的是，驱动程序提交一个设备的写操作，然后根据设备操作模式（中断模式或轮询模式）来等待或阻塞请求的线程，直到该操作完成。

### 10.3.3.3 设备控制（DeviceCtrl）的实现

读写操作不能抽象所有可能的设备操作，比如，对一个音频设备，可能需要控制诸

如暂停、重新开始、快进、倒退等操作，这种情况下，读写操作就无法胜任了，还有一种读写操作无法胜任的就是，设备的读写单位可能不一样，比如，针对串行接口，可能是一个字节一个字节的读写，而针对硬盘、光盘等存储设备，则可能是一个数据块一个数据块的读写，在 UNIX 的实现中，对这两种不同类型的设备分别做了处理（对应于 UNIX 的字符设备和块设备），但在 Hello China 的实现中，则没有进行区分，而进行了统一对待，但在读写的时候，客户程序必须首先确定每次操作的字节数（一个字节还是多个字节，具体是多少字节等），客户如何获取每次操作的字节数量，读写操作也是无法胜任的。因此，引入了设备控制操作（DeviceCtrl 函数）。

设备控制操作是通过 DeviceCtrl 函数来实现的，用户通过 IOControl 函数调用（由 IOManager 提供）来实现对设备的 DeviceCtrl 函数的访问。

对于 DeviceCtrl 功能的输入参数和输出参数，在 DRCB 对象中做了完善的提供，客户程序在请求设备的控制功能的时候，首先使用合适的参数调用 IOManager 的 IOControl 函数，该函数创建一个 DRCB，根据 IOControl 的参数初始化这个对象，然后进一步的调用设备驱动程序对象的 DeviceCtrl 函数。

DRCB 对象中的 dwCtrlCommand 成员用来指出，设备驱动程序应该执行哪个功能，然后调用适当的功能函数。一般情况下，下列控制功能必须实现：

- 1、CONTROL\_COMMAND\_GET\_READ\_BLOCK\_SIZE，获得设备每次读操作的数据大小，比如，针对串行接口，可以是一个字节，针对磁盘，可以是 512 字节；
- 2、CONTROL\_COMMAND\_GET\_WRITE\_BLOCK\_SIZE，获得设备每次写操作的数据大小；
- 3、CONTROL\_COMMAND\_GET\_DEVICE\_ID，获取设备的唯一 ID，针对不同的设备，该功能的实现也不一样，而且 ID 也没有一个统一的编配，这种情况下，系统一致认为所有设备的 ID 是一个字符串，因此，设备驱动程序可以有选择的实现该功能，如果不能实现，则简单的返回失败结果；
- 4、CONTROL\_COMMAND\_GET\_DEVICE\_DESC，获取设备的描述信息，设备驱动程序可以在描述信息中，对设备的具体型号、厂家、功能特点等进行描述，比如，“Broadcom 570x Gigabit Integrated Controller”。

其它的功能，设备根据实际的需要来自己定义，比如，针对一个音频控制设备，驱动程序可以定义诸如快进、倒退、循环播放等功能命令，进而完成实现。

### 10.3.4 设备驱动程序对象

BEGIN\_DEFINE\_OBJECT(\_\_DRIVER\_OBJECT)

## INHERIT\_FROM\_COMMON\_OBJECT

__DRIVER_OBJECT*	lpPrev;
__DRIVER_OBJECT*	lpNext;
DWORD	(*DeviceRead)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
DWORD	(*DeviceWrite)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
DWORD	(*CreateFileSystem)(__COMMON_OBJECT* lpDev, __COMMON_OBJECT* lpDrv, DRCB* lpDrcb);
DWORD	(*DeviceCtrl)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
VOID	(*DeviceFlush)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
DWORD	(*DeviceSeek)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
DWORD	(*DeviceOpen)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
VOID	(*DeviceClose)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);
DWORD	(*DeviceCreate)(__COMMON_OBJECT* lpDrv, __COMMON_OBJECT* lpDev, DRCB* lpDrcb);

```

        DWORD                                (*DeviceDestroy)(__COMMON_OBJECT* lpDrv,
                                                                __COMMON_OBJECT* lpDev,
                                                                DRCB*                lpDrcb);

END_DEFINE_OBJECT();

```

### 10.3.5 DriverEntry 的实现

DriverEntry 是 IOManager 调用的函数，该函数给设备驱动程序一个机会，用来做一些初始化工作，同时也注册（向 IOManager）对设备进行操作的标准函数。

一般情况下，该函数内可以做下列工作：

- 1、初始化驱动程序的全局变量；
- 2、注册用来对设备进行操作的标准函数；
- 3、创建自己管理的设备对象。

一般情况下，直接把驱动程序实现的一些对设备操作的标准函数指针赋值给驱动程序对象（作为该函数的参数传递）即可，如下：

```

lpDriverObject->DeviceRead    = DeviceRead;
lpDriverObject->DeviceWrite   = DeviceWrite;
lpDriverObject->DeviceCtrl    = DeviceControl;
... ..

```

另一项重要的工作，就是创建设备对象，可以通过调用 IOManager 提供的 CreateDevice 函数来完成。一般情况下，设备驱动程序只能创建自己可以管理的对象（创建自己不能管理的设备对象也是可以的，但没有任何意义），对对象的资源分配，有两种方式：

- 1、接受由 IOManager 传递过来的资源分配方案，把这些资源分配给设备；
- 2、如果设备驱动程序管理的设备，系统资源固定（比如，对于键盘、显示器、IDE 接口硬盘等设备，其资源基本上固定），那么可以不接受 IOManager 提供的资源分配方案，而自己硬性的给设备分配资源，这种方式下，很有可能出现资源冲突。

在创建设备对象的时候，一个很重要的事情就是指定设备的设备扩展，所谓设备扩展，就是紧跟随设备对象后面的一段存储空间，该空间内存储了跟设备相关的一些数据，比如，设备的类型，设备块的大小，当前指针位置，设备的尺寸等。设备扩展的大小只

有设备驱动程序知道，因此需要设备驱动程序来指定。

下面是一个典型的创建设备对象，并对之初始化的例子：

```
__DEVICE_OBJECT*    lpIdeHardDisk  = NULL;

lpIdeHardDisk = CreateDevice("IDE Hard Disk 0",    //Device name.
                             lpDriverObject,      //Driver object.
                             NULL,                //Resource descriptor.
                             NULL,                //Device extension.
                             DEVICE_TYPE_STORAGE,
                             512);                //Device block size.

If(NULL == lpIdeHardDisk)    //Failed to create device.
    Return FALSE;

InitializeIdeHardDisk(lpIdeHardDisk);    //Initialize it.
... ..
```

### 10.3.6 UnloadEntry 的实现

当 IOManager 要卸载一个设备驱动程序的时候，会调用设备驱动程序提供的 UnloadEntry 函数，一般情况下，设备驱动程序需要在这个函数中做如下事情：

- 1、调用 DestroyDevice 函数，销毁自己创建的（但没有销毁的）所有设备对象；
- 2、释放设备驱动程序运行过程中申请的内存资源；
- 3、释放所有通过 ReserveResource 函数预留的资源。

如果设备驱动程序在系统运行的整个过程中都存在，那么该函数可以不做任何事情，简单返回即可，但如果设备驱动程序有可能被动态的加载或卸载，比如一些可移动存储介质的驱动程序，那么设备驱动程序就必须在这个函数中释放所有的资源。如果设备驱动程序在运行的过程中申请了系统资源（比如内存等），但在卸载的时候没有释放，那么会造成资源泄漏。

## 10.4 设备对象

系统中任何打开的设备，都对应一个设备对象，该对象用来记录特定设备的相关信息，也包含了指向该设备驱动程序的后向指针，在 `CreateFile`（`IOManager` 提供的用户侧接口调用）函数成功返回后，就是返回打开或创建的设备对象的地址。

设备对象是用名字来唯一标识的，用户线程通过 `CreateFile` 调用打开文件的时候，需要明确指定设备的名字，`IOManager` 就是靠这个名字来找到具体的设备的。

### 10.4.1 设备对象的定义

在 `Hello China` 当前版本的实现中，设备对象的定义如下：

```
BEGIN_DEFINE_OBJECT(__DEVICE_OBJECT)
    INHERIT_FROM_COMMON_OBJECT

    __DEVICE_OBJECT*      lpPrev;
    __DEVICE_OBJECT*      lpNext;
    UCHAR                  DevName[MAX_DEV_NAME_LEN];
    __KERNEL_THREAD_OBJECT*

        lpOwner;
    DWORD                  dwDevType;
    __DRIVER_OBJECT*      lpDriverObject;
    /*DWORD                  dwStartPort;
    DWORD                  dwEndPort;
    DWORD                  dwDmaChannel;
    DWORD                  dwInterrupt;
    LPVOID                  lpMemoryStartAddr;
    DWORD                  dwMemLen;*/
    DWORD                  dwRefCount;
    DWORD                  dwBlockSize;
    DWORD                  dwMaxReadSize;
    DWORD                  dwMaxWriteSize;
    //DWORD                  DevExtension[0];
    LPVOID                  lpDevExtension;
```

END\_DEFINE\_OBJECT();

在下面的几节中，我们对该定义中的几个重要字段（变量或成员）进行说明。

## 10.4.2 设备对象的命名

在当前版本的 Hello China 实现中，对所有的设备，采用设备名唯一标识，这样就需要定义一套规范的命名方式，来对系统中可能存在的设备进行命名。

在当前版本的实现中，系统中可能存在下列几类设备：

- 1、普通文件，即存储在存储设备（比如，硬盘、光盘、FLASH 卡等）上的数据文件，在 Hello China 中，数据文件也作为设备对待，与普通设备不同的是，文件设备的驱动程序，是文件系统；
- 2、实际存在的物理设备，比如显示卡、网卡、串口、鼠标/键盘等，这些设备是实实在在的物理硬件设备，有相应的驱动程序进行驱动，每种物理设备，完成一项具体的功能；
- 3、系统虚拟的设备，这类设备是操作系统（或设备驱动程序）虚拟出来的一种设备，比如，命名管道、RAM 存储设备等，这些设备不对应于具体的物理外设，但却完成某项特定的功能，比如，命名管道可以完成进程间通信的功能，RAM 存储设备可以把内存的一部分预留出来，虚拟成一个文件系统，供操作系统临时保存文件使用，等等；
- 4、网络文件系统，比如，可以通过映射，来把远程计算机上的一个文件（或目录），映射为本地的一个文件系统，这样只要对本地虚拟的文件系统进行访问，就可以间接的访问到远程计算机的文件系统，比较典型的如 NFS 等。

针对上述几种设备类型，分别定义其命名形式如下：

- 1、针对普通文件，采用的命名格式为文件系统标识符加文件路径的方式，比如，系统中存在三个硬盘分区，则每个分区被格式化为一个文件系统，相应的文件系统标识符为（缺省情况下）C:，D:，E:，比如，在文件系统 C:下有一个目录 Hello China，该目录下有一个名字为 cat.dat 的文件，于是该文件可以这样命名：C:\Hello China\cat.dat；
- 2、对于实际存在的物理设备，采用这样的命名格式：\\dev\device\_name，其中两个反斜线和后面的dev，是固定部分，操作系统根据这个固定部分来确定该命名是实际存在的物理设备命名，简化起见，一般把dev省略掉，简化为 \\device\_name的形式，这样省略的另外一个目的，就是避免与下面网络文件系统的命名冲突；
- 3、对于系统虚拟的设备，其命名按照实际存在的物理设备的格式，比如，在当前的实现中，对命名管道的命名为 \\."20ADC1F6-5194-416e-97AB-962A03472410"，其中，后面device\_name部分，是采用一个GUID转换来的唯一字符串；
- 4、对于远程文件系统，命名格式如下：\\server\_name\file\_path\_name，其中server\_name指明了具体的服务器名字，也就是远程计算机的名字，而file\_path\_name则是远程计算机上的文件路径名。比如，命名服务器shanghai的一个共享文件：shanghai.map，

结果为: [\\shanghai\shanghai.map](#)。

设备的命名, 构成了系统 IOManager 管理设备的基础, 但当前版本的实现中, 操作系统在加载设备驱动程序, 并创建设备的时候, 却不对设备名字做任何检查。因此, 如果驱动程序不按照上述规则为系统中的设备命名, 则可能会一起混乱。

### 10.4.3 设备对象的类型

为了管理上的方便, 我们把物理设备根据其功能划分成特定的类别, 这样就可以对一种设备进行更细致的划分, 进而提供更细致的管理和监控。在 Hello China 当前版本的实现中 (V1.0), 我们把设备分成以下几类:

- 1、**DEVICE\_TYPE\_STORAGE**: 存储设备, 能够提供永久存储功能的功能部件, 比如软盘、硬盘 (基于 IDE 或 SCSI 接口)、光盘、USB 接口的存储设备等, 之所以这样划分, 是因为这些设备都需要有文件系统进行支撑;
- 2、**DEVICE\_TYPE\_FILE\_SYSTEM**: 文件系统对象, 针对系统中存在的每个文件系统, 操作系统都创建一个文件系统对象;
- 3、**DEVICE\_TYPE\_NORMAL**: 普通设备对象, 所有不属于上述类别的设备, 都属于普通设备对象;
- 4、**DEVICE\_TYPE\_FILE**, 文件对象, 任何打开的文件系统中的文件, 都被赋予这个对象属性。

在设备驱动程序加载完毕后, IOManager 会根据设备的类型 (存储设备或非存储设备) 来决定是否进行进一步的初始化。针对存储设备, IOManager 会尝试为该设备加载一种文件系统, 比如, 针对硬盘 (严格来说, 应该是硬盘的每个分区), 操作系统会读取硬盘分区的相关信息, 判断该硬盘分区的文件系统类型, 如果判断的结果是目前能够支持的文件系统, 则操作系统就会加载该文件系统的驱动程序, 并调用文件系统驱动程序提供的 CreateFileSystem 函数, 创建一个文件系统设备对象, 如果判断结果未知 (即该分区或者没有被格式化, 或者被格式化成了系统目前不支持的文件系统), 那么操作系统将会放弃为该分区加载文件系统的尝试。

对于文件系统的实现流程, 总结如下:

- 1、IOManager 遍历已经创建的所有设备对象, 检查该对象是否是存储设备 (设备对象类型为 **DEVICE\_TYPE\_STORAGE**);
- 2、若是, 则会读取该设备对象的相关信息 (使用设备驱动程序提供的读写函数), 根据这些信息判断该设备被格式化成的文件类型;
- 3、如果判断的结果 (特定的文件系统类型) 为操作系统当前不支持的文件系统, 则放

弃当前设备，进行下一个设备的检测，若当前操作系统支持该设备对应的文件系统，则会加载对应文件系统的驱动程序，并调用文件系统驱动程序的 `DriverEntry` 函数，给文件系统一个初始化的机会；

- 4、调用文件系统驱动程序提供的 `CreateFileSystem` 函数，并把检测到的存储设备的物理参数（比如，硬盘分区的起始扇区号、所在的硬盘号、扇区数量、每个扇区的大小等）传递给该函数，创建一个文件系统设备对象（实际上是创建一个设备对象，该设备对象的对象类型为 `DEV_TYPE_FILE_SYSTEM`）；
- 5、完成当前设备对象的检测后，继续进行下一个设备对象的检测，直到所有的设备对象都检测完毕。

### 10.4.4 设备对象的设备扩展

设备扩展是设备对象定义中最后一个变量（成员）`lpDevExtension`，可以说该变量是设备对象定义中最重要的一个变量，它为不同的设备，预留了保存各自数据的空间。

正常情况下，物理设备是各种各样的，这些设备之间的差异，最终表现在设备的不同状态数据上，比如，对于硬盘，需要保存诸如硬盘大小、分区个数、扇区大小、扇区数量、操作方式（LBA、CHS 等）等等数据，对于网卡，则需要保存诸如 MAC 地址、MTU 大小、缓冲区大小、工作方式（全双工/半双工）、工作速率（1000M/100M/10 等）等数据，设备对象不能囊括所有这些不同设备的不同要求，因此只把每种设备必须具有的数据抽象出来，做了明确定义，比如设备名、所占用的系统资源等，而对于设备特定的数据，设备对象没有做明确定义（也不可能定义），而是预留了一个指针，该指针指向特定的设备状态数据，这样不同的设备，其公共部分是相同的（设备对象定义部分），而差异数据，则由设备驱动程序组织，并存放在 `lpDevExtension` 指向的存储空间中，这个存储空间就称为设备对象扩展。

在当前版本的实现中，设备对象扩展是由 `IOManager` 的 `CreateDevice` 函数创建的，在 `CreateDevice` 函数的参数中，有一个参数是 `dwExtSize`，该数据指出了设备扩展的尺寸，由设备驱动程序提供。在当前的实现中，`CreateDevice` 仅仅通过调用 `KMemAlloc` 申请一块内存地址空间，然后赋予 `lpDevExtension` 成员，具体的设备扩展内容，由设备驱动程序自己填写。

### 10.4.5 设备的打开操作

在 `Hello China` 当前版本的实现中，设备的打开操作是通过调用 `CreateFile` 函数实现

的。该函数是由 `IOManager` 提供给用户线程，用户线程访问设备之前，使用该函数打开待访问的设备。该函数不但可以用于打开物理设备，而且还可以用来打开或创建普通的数据文件（从该函数的名字也可以看出这一点），对于打开文件的详细流程，在前面的章节中已经做了介绍，下面的流程，描述了如何打开一个物理设备：

- 1、用户调用该函数，其中待打开设备的设备名字作为参数之一；
- 2、`CreateFile`函数（也可以说是`IOManager`）分析设备名字，如果发现该设备名字是一个普通文件（以文件系统标识符开头，比如`C:`，`D:`等），则启用文件打开流程，如果分析结果表示，待打开的设备对象是一个物理设备（以 `\\.\devicename` 开头），则继续下面的设备打开操作流程；
- 3、`CreateFile` 查询设备对象链表，以设备名字作为索引关键字，从头开始遍历设备对象链表；
- 4、如果遍历完整个链表，没有查找到目标设备对象，则说明对应的设备没有安装，或者安装了但没有启用，这种情况下，`CreateFile` 返回一个空值（`NULL`），表示打开设备失败；
- 5、如果能够在设备对象链表中找到对应的设备，则判断设备的当前状态（打开还是未打开），如果状态为打开，则判断该设备是否允许共享打开（允许两个或以上的线程同时打开该设备），如果允许，则增加设备打开计数，然后返回设备对象指针，如果不允许，则仍然返回一个空值（`NULL`），指明该操作失败；
- 6、调用 `CreateFile` 的线程如果得到一个失败的操作结果，可以通过 `GetLastError` 调用，获取错误原因。

可以看出，上述操作的关键，也是遍历整个系统设备对象链表，跟文件的打开方式基本一致。另外还可以看出，对设备的打开操作，也是以设备名作为唯一关键字来查询设备的，因此，`Hello China` 当前版本的实现中，要求系统中的所有设备，必须具有不同的设备名字。要达到这个要求，如果任意取设备名，可能会存在冲突的可能，因此，必须采用一些特殊的命名措施，来确保系统中所有设备的名字没有冲突。下面，我们介绍几种可用的设备命名策略，供设备驱动程序实现者参考。

## 设备命名策略

在 `Hello China` 当前版本的实现中，对设备的区分，是按照名字来进行的，也就是说，名字是设备唯一的标识。这样如果设备是由多家厂商提供的，那么就可能产生命名冲突，为解决这个问题，建议设备驱动程序编写者在为设备命名的时候，采用能够产生全球唯一设备名字（字符串）的算法，来产生设备名字，而不要随意的命名。需要指出的是，设备名字不同于设备描述，如果设备供应商想对自己的设备做一些简单的描述，那么可

以在设备描述里面进行，系统提供了函数接口，可以让用户很容易的得到设备描述信息。

下面列举了几种可以采用的方法，来生成全球唯一的字符串，作为设备的名字，设备驱动程序编写者可以采用下列命名方式中的一种，对设备进行命名（如果不按照下列给出的命名方式，则可能产生冲突）：

### 采用全球唯一标识符来命名设备

Microsoft 公司提供的一个小程序 GUIDGEN.EXE（随 Microsoft Visual Studio 一起发行）可以产生长度为 32 字节的全球唯一标识符（GUID），比如，下面是该程序产生的一个 GUID：

**{9EABB977-9872-4cfc-A381-2E4D52864FA5}**

由于采用了独特的算法，可以确保生成的 GUID 全球唯一，因此，设备驱动程序开发商可以把上述 GUID 转换成字符串，来命名自己的设备，比如，对于上述 GUID，可以转换成下列形式：

**“9EABB977-9872-4cfc-A381-2E4D52864FA5”**

也可以转换成下列形式：

**“9EABB97798724cfcA3812E4D52864FA5”**

总之，只要把 GUID 表示成字符串形式，然后作为全局变量定义在设备驱动程序中，作为设备的名字，可以确保不会产生冲突。

### 采用网络接口卡硬件地址命名设备

另外一种可以作为唯一标识符种子的就是以以太网接口卡的物理地址（也称为 MAC 地址）。以太网接口卡的物理地址有统一的组织管理并分配，可以确保全球唯一，该地址由 48 比特（6 字节）组成，其中前面三字节是分配给特定厂家的，而后面三个字节，则由该厂家分配。在产生设备驱动标识符的时候，可以直接把一块网络接口卡的 MAC 地址转换成字符串后，作为设备的标识符，比如，网络接口卡的 MAC 地址为：

**00-11-43-98-90-AB**

则可以把上述标识符转换成字符串：

**“00-11-43-98-90-AB”**

直接作为设备的标识符，也可以在此基础上，增加一些额外信息作为设备标识符，比如，可以这样操作：

**“IDE Hard Disk 00-11-43-98-90-AB”**

## 10.5 设备的中断管理

特定设备的中断处理函数，由设备驱动程序提供，在当前版本的实现中，Hello China 提供了完善的中断连接机制，可以通过 `ConnectInterrupt` 调用，把特定的中断处理函数跟相应的中断向量连接。

一般情况下，这个连接是在 `DriverEntry` 函数内完成的，在设备初始化的时候，就已经确定了设备所使用的中断号（系统分配，或设备驱动程序自己检测），中断号确定之后，就可以直接调用 `ConnectInterrupt` 函数，连接中断处理程序和中断向量了。在当前版本的实现中，对于设备的中断处理程序（中断处理函数），其原型必须符合下列形式：

```
BOOL InterruptHandler(LPVOID lpParam, LPVOID lpEsp);
```

其中，第一个参数为传递给中断处理程序的特定参数，第二个参数则是中断程序发生之后，堆栈框架的指针，中断处理程序通过该参数（`lpEsp`），可以访问到中断发生之后的系统堆栈框架。

一般情况下，中断处理程序在开始的时候，需要先决定该中断是不是自己的，因为在许多硬件设计环境中，可能多个设备共同使用一个中断向量（即多个设备连接到中断控制器的同一条输入引脚上），在 Hello China 当前版本的实现中，对于共同使用一个中断向量的中断处理函数，使用链表的方式连接在一起（串连在一起），每当中断发生的时候，操作系统从链表的头部开始，依次调用中断处理程序，根据中断处理程序的返回（`TRUE` 或 `FALSE`），来决定中断是否得到了处理。如果调用的中断处理程序返回了 `FALSE`，则说明该中断不是由提供刚刚调用的中断处理程序的设备驱动程序来处理，于是会继续调用下一个中断处理程序，直到有一个中断处理程序返回 `TRUE`，或到达链表的末尾。

因此，在编写设备的中断处理程序的时候，在函数的开始处，建议马上采用简单的代码来判断该中断是不是针对自己的，如果是，则进一步处理，否则，马上返回 `FALSE`，以便系统尽快的把中断调度到正确的处理程序上。

那么，中断处理程序如何才能知道该中断是针对自己的，还是不是自己的呢？一般情况下，设备驱动程序可以通过读取设备的寄存器得到，或者通过内部的一操作情况来

确定。比如，假设 IDE 硬盘和网络通信控制器（NIC）连接到同一条中断线上，那么当中断发生的时候，网卡驱动程序就可以读取网卡的状态寄存器，来判断是否有报文到达，如果是，则说明该中断是由网卡发起的，则进行进一步处理，并返回 **TRUE**（表示中断得到了正确的处理），否则，直接返回 **FALSE**。对于 IDE 接口的硬盘驱动器，则可以采用内部状态来确定是否是自己的中断。因为正常情况下，必须由设备驱动程序发起了请求，比如一个读取操作，当操作结束时，设备才会发生中断，因此，这种情况下，驱动程序会保持一个未完成的操作事务，当收到中断的时候，驱动程序就可以结合事务的状态，并读取适当的状态寄存器，来判断是否是 IDE 硬盘发起的中断。当然，不同的设备有不同的判断方式，需要结合具体的设备来完成。

# 第十一章 应用编程接口

—— Application Programming Interface



11.1 核心线程操作接口

CreateKernelThread

原型	HANDLE CreateKernelThread(DWORD dwStackSize, DWORD dwStatus, DWORD dwPriority, __KERNEL_THREAD_ROUTINE lpStartRoutine, LPVOID lpRoutineParam, LPVOID lpReserved);	
用途	创建一个核心线程对象。	
参数	取值	含义
dwStackSize	小于 8M 的整数	待创建线程的堆栈大小
dwStatus	KERNEL_THREAD_STATUS_READY	初始状态为准备就绪
	KERNEL_THREAD_STATUS_SUSPEND	初始状态为挂起
dwPriority	KERNEL_THREAD_PRIORITY_CRITICAL	关键优先级（最高）
	KERNEL_THREAD_PRIORITY_HIGH	高优先级
	KERNEL_THREAD_PRIORITY_NORMAL	中优先级
	KERNEL_THREAD_PRIORITY_LOW	低优先级
lpStartRoutine	__KERNEL_THREAD_ROUTINE	线程的功能主体函数
lpRoutineParam	LPVOID	功能函数参数
lpReserved	LPVOID	保留参数
返回值类型	HANDLE	
返回值取值	含义	
NULL	创建核心线程失败	
非 NULL	创建核心线程成功，返回所创建的核心线程对象指针	
注意事项	若 dwStackSize 参数取值 0，则使用缺省的堆栈尺寸（16K）为	

	线程创建堆栈。该参数十分重要，需要根据应用仔细评估，若出现堆栈溢出的情况，可能会导致系统崩溃。
--	---

应用范例：

## DestroyKernelThread

原型	VOID DestroyKernelThread(HANDLE hThread);	
用途	销毁创建的核心线程对象。	
参数	取值	含义
hThread	HANDLE	待销毁的线程句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	调用 CreateKernelThread 函数创建的核心线程对象，必须调用该函数进行销毁，否则会出现内存泄漏。	

应用范例：

```
typedef struct {
    DWORD dwInitNumber;    //The initial number to calculate.
    DWORD dwResult;    //Result.
}FibonacciBlock;

DWORD Fibonacci(LPVOID lpFiboBlock)    //Calculate Fibonacci number.
{
    FibonacciBlock* lpBlock = (FibonacciBlock*)lpFiboBlock;
    DWORD dwFirst = 0;    //First element in Fibonacci sequence.
    DWORD dwSecond = 1;    //Second element in Fibonacci sequence.
    DWORD dwThird = 0;    //Third element in Fibonacci sequence.
    for(DWORD i = 0;i < lpBlock->dwInitNumber;i ++)
    {
        dwThird = dwFirst + dwSecond;
        dwFirst = dwSecond;
        dwSecond = dwThird;
    }
}
```

```

    }
    lpFiboBlock->dwResult = dwFirst; //Return result.
    return 0L;
}

void Main()
{
    FibonacciBlock Block[3];
    HANDLE hThread[3];
    Block[0].dwInitNumber = 20;
    Block[1].dwInitNumber = 30;
    Block[2].dwInitNumber = 40;
    for(DWORD i = 0; i < 3; i++)
    {
        hThread[i] = CreateKernelThread(0L,
                                         KERNEL_THREAD_STATUS_READY,
                                         KERNEL_THREAD_PRIORITY_NORMAL,
                                         Fibonacci,
                                         (LPVOID)&Block[i],
                                         NULL);
        if(NULL == hThread[i]) //Can not create kernel thread.
            break;
    }

    WaitForThisObject(hThread[0]);
    WaitForThisObject(hThread[1]);
    WaitForThisObject(hThread[2]);
    DumpoutResult(Block); //Print out the calculate result.
    DestroyKernelThread(hThread[0]);
    DestroyKernelThread(hThread[1]);
    DestroyKernelThread(hThread[2]);
    return;
}

```

上述代码示意了 `CreateKernelThread` 和 `DestroyKernelThread` 两个函数的简单应用。其中，线程功能函数 `Fibonacci` 用于完成 `Fibonacci` 序列的计算，为了传递参数，获得计算结果，定义了一个结构体 `FibonacciBlock`，用于向 `Fibonacci` 传递待计算的序列的项数，并保存返回结果。

在 `Main` 函数里，创建三个线程，分别用于计算 `Fibonacci` 数列第 20、30 和 40 项的结果。线程创建完毕之后，`Main` 函数（主线程）将等待三个线程完成（通过调用 `WaitForThisObject`，详细含义请参考本章后续部分），计算完成后并输出结果（`DumpoutResult`）后，调用 `DestroyKernelThread` 销毁创建的核心线程对象。

### SendMessage

原型	BOOL SendMessage(HANDLE hThread,MSG* lpMsg);	
用途	向一个特定的核心线程发送消息	
参数	取值（或类型）	含义
hThread	HANDLE	目标线程的句柄
lpMsg	MSG	待发送消息的指针
返回值类型	BOOL	
返回值取值	含义	
TRUE	发送消息成功，消息已经挂入目标线程的本地消息队列	
FALSE	发送消息失败，当前情况下，若目标线程消息队列已满，则返回该值。	
注意事项	该函数是一个非阻塞函数，若目标线程消息队列不满，则发送成功，否则会返回失败。	

示例程序：

```
MSG msg;
msg.wCommand = CMD_USER + 1;
msg.wParam   = 0;
msg.dwParam  = 1024;
SendMessage(hThread,&msg); //Send message to hThread.
... ..
```

GetMessage

原型	BOOL GetMessage(MSG* lpMsg);	
用途	从线程本地消息队列读取消息	
参数	取值（或类型）	含义
lpMsg	MSG*	存放消息的消息结构地址
返回值类型	BOOL	
返回值取值	含义	
TRUE	获取消息成功	
FALSE	获取消息失败（实际上，当前的实现是一个阻塞操作，因此不会出现这种情况，但将来可能会实现一种非阻塞操作，则可能返回 FALSE）。	
注意事项	该函数是一个阻塞函数，若消息队列中无任何消息，则当前线程将进入阻塞状态，直到有消息到达。	

```
示例程序：
MSG Msg;
while(TRUE)
{
    if(GetMessage(&Msg)) //Get message from local message queue.
    {
        if(Msg.wCommand == MSG_EXIT)
            break;
        //
        //Process message here.
        //
    }
}
```

SetKernelThreadPriority

原型	DWORD SetKernelThreadPriority(HANDLE hThread,DWORD dwPriority);	
用途	设置一个线程的优先级	
参数	取值（或类型）	含义

hThread	HANDLE	待设定优先级的目标线程
dwPriority	KERNEL_THREAD_PRIORITY_CRITICAL	实时优先级（最高）
	KERNEL_THREAD_PRIORITY_HIGH	高优先级
	KERNEL_THREAD_PRIORITY_NORMAL	普通优先级
	KERNEL_THREAD_PRIORITY_LOW	最低优先级
返回值类型	DWORD	
返回值取值	含义	
四个优先级之一	目标线程的原始优先级	
注意事项		

## GetKernelThreadPriority

原型	DWORD GetKernelThreadPriority(HANDLE hThread);	
用途	获得线程的优先级	
参数	取值（或类型）	含义
hThread	HANDLE	目标核心线程句柄
返回值类型	DWORD	
返回值取值	含义	
KERNEL_THREAD_PRIORITY_CRITICAL	实时优先级	
KERNEL_THREAD_PRIORITY_HIGH	高优先级	
KERNEL_THREAD_PRIORITY_NORMAL	普通优先级	
KERNEL_THREAD_PRIORITY_LOW	最低优先级	
注意事项	若 hThread 参数设置为 NULL，则获取当前线程（调用该函数的线程）的优先级。	

GetKernelThreadID

原型	DWORD GetKernelThreadID(HANDLE hThread);	
用途	获得一个线程的 ID	
参数	取值（或类型）	含义
hThread	核心线程对象句柄	目标线程的句柄
返回值类型	DWORD	
返回值取值	含义	
任意整数	目标线程的线程 ID	
注意事项		

内存操作接口

KMemAlloc

原型	LPVOID KMemAlloc(DWORD dwSize,DWORD dwSizeType);	
用途	在操作系统核心空间中申请内存	
参数	取值	含义
dwSize	小于 4M 的整数	申请内存的大小
dwSizeType	KMEM_SIZE_TYPE_ANY	申请任意尺寸大小内存
	KMEM_SIZE_TYPE_4K	申请 4K 倍数大小内存
返回值类型	LPVOID	
返回值取值	含义	
NULL	申请内存失败	
非 NULL	申请成功，返回申请的内存的起始地址	
注意事项	只建议设备驱动程序使用，用来申请缓冲区空间。	

实例程序：

```
LPVOID lpBuffer = NULL;
lpBuffer = KMemAlloc(4096,KMEM_SIZE_TYPE_ANY);
if(NULL == lpBuffer) //Failed to allocate kernel memory.
return FALSE;
```

... ..

KMemFree

原型	VOID KMemFree(LPVOID lpStartAddr,DWORD dwSize,DWORD dwSizeType);	
用途	释放由 KMemAlloc 申请的核心内存	
参数	取值（或类型）	含义
lpStartAddr	LPVOID	待释放内存的起始地址
dwSizeType	KMEM_SIZE_TYPE_ANY	申请任意尺寸大小内存
	KMEM_SIZE_TYPE_4K	申请 4K 倍数大小内存
dwSize	小于 4M 的任意值	待释放的内存大小，只有在 dwSizeType 取 值为 KMEM_SIZE_TYPE_4K 的时候有效
返回值类型	无返回值（VOID）	
返回值取值	含义	
—	—	
注意事项	只有在释放内存大小是 4K 整数倍(KMEM_SIZE_TYPE_4K 标志设置)的时候，才需要给出 dwSize 参数，否则 dwSize 参数设置为 0。	

实例程序：

```
LPVOID lpBuffer = NULL;
lpBuffer = KMemAlloc(4096,KMEM_SIZE_TYPE_ANY);
if(NULL == lpBuffer) //Failed to allocate kernel memory.
return FALSE;
... ..
KMemFree(lpBuffer,KMEM_SIZE_TYPE_ANY,0L);
... ..
```

VirtualAlloc

原型	LPVOID VirtualAlloc(LPVOID lpDesiredAddr,
----	---

	DWORD dwSize, DWORD dwAllocFlags, DWORD dwAccessFlags, LPSTR lpszName, LPVOID lpReserved);	
用途	从系统线性地址空间中分配一个连续的区域（内存）。	
参数	取值（或类型）	含义
lpDesiredAddr	LPVOID	期望的开始地址
dwSize	DWORD	申请的内存区域的大小
dwAllocFlags	VIRTUAL_AREA_ALLOCATE_RESERVE	保留一块虚拟区域
	VIRTUAL_AREA_ALLOCATE_COMMIT	提交保留的虚拟区域
	VIRTUAL_ARAE_ALLOCATE_ALL	保留并提交内存区域
	VIRTUAL_AREA_ALLOCATE_IO	为 IO 设备分配内存映射区
dwAccessFlags	VIRTUAL_AREA_ACCESS_READ	Read Access
	VIRTUAL_AREA_ACCESS_WRITE	Write Access
	VIRTUAL_AREA_ACCESS_RW	读写访问
	VIRTUAL_AREA_ACCESS_EXEC	执行访问
lpszName	LPSTR	该内存区域的名字字符串
lpReserved	LPVOID	保留将来使用。
lpStartAddr	LPVOID	待释放的内存起始地址
返回值类型	LPVOID	
返回值取值	含义	
NULL	申请虚拟区域失败。	
非 NULL	申请成功，返回具体的虚拟区域地址。	

注意事项	该函数只能以页面尺寸（比如，IA32 CPU 下为 4K）为单位申请内存，可供应用程序和设备驱动程序调用。
------	---

## VirtualFree

原型	VOID VirtualFree(LPVOID lpStartAddr);	
用途	释放申请的虚拟区域。	
参数	取值（或类型）	含义
lpStartAddr	LPVOID	待释放的内存起始地址
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	必须是调用 VirtualAlloc 函数申请的内存，才能使用该例程释放。	

## malloc

原型	LPVOID malloc(DWORD dwSize);	
用途	从线程本地堆（缺省堆）中申请内存	
参数	取值（或类型）	含义
dwSize	任意整数值	待申请的内存大小
返回值类型	LPVOID	
返回值取值	含义	
NULL	申请内存失败	
非 NULL	申请内存成功，返回申请的内存的地址	
注意事项	该函数从线程的本地默认堆中申请内存，是应用程序申请内存的主要方式。	

示例程序：

```
LPVOID lpBuffer = NULL;
```

```
lpBuffer = malloc(4096);    //Allocate 4096 bytes memory.
```

```
if(NULL == lpBuffer)       //Failed to allocate memory.
```

```
return FALSE;
... ..
```

free

原型	VOID free(LPVOID lpStrartAddr);	
用途	释放从线程本地堆中申请的内存	
参数	取值（或类型）	含义
lpStartAddr	LPVOID	待释放的内存起始地址
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	必须是调用 malloc 函数申请的内存，才能使用该例程释放。	

示例程序：

```
LPVOID lpBuffer = NULL;
lpBuffer = malloc(4096);      //Allocate 4096 bytes memory.
if(NULL == lpBuffer)          //Failed to allocate memory.
return FALSE;
... ..
free(lpBuffer);              //Release the memory block.
```

CreateHeap

原型	HEAP CreateHeap(DWORD dwInitSize);	
用途	创建一个线程本地堆	
参数	取值（或类型）	含义
dwInitSize	小于 8M 的整数	起始堆内存池的大小
返回值类型	HEAP	
返回值取值	含义	
NULL	创建堆失败	
非 NULL	创建堆成功，返回堆对象的句柄	
注意事项		

## DestroyHeap

原型	VOID DestroyHeap(HEAP hHeap);	
用途	销毁一个线程本地堆	
参数	取值（或类型）	含义
hHeap	HEAP	待销毁的堆对象句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	所销毁的堆对象，必须是通过 CreateHeap 函数创建的。	

## HeapAlloc

原型	LPVOID HeapAlloc(HEAP hHeap,DWORD dwSize);	
用途	从一个线程本地堆中申请内存	
参数	取值（或类型）	含义
hHeap	HEAP	目标堆句柄
dwSize	小于 8M 的整数	待申请的内存大小
返回值类型	LPVOID	
返回值取值	含义	
NULL	申请内存失败	
非 NULL	申请内存成功，返回申请的内存首地址	
注意事项	若 hHeap 为 NULL，则从线程默认堆中申请内存，效果与 malloc 相同。	

## HeapFree

原型	VOID HeapFree(HEAP hHeap,LPVOID lpStartAddr);	
用途	释放一块堆内存	
参数	取值（或类型）	含义
hHeap	HEAP	待释放内存所在的堆的句柄
lpStartAddr	LPVOID	待释放的内存的起始地址
返回值类型	无	
返回值取值	含义	

—	—
注意事项	待释放的内存，必须是通过 HeapAlloc 函数申请的，且必须是同一个堆对象。

定时器操作接口

SetTimer

原型	HANDLE SetTimer(DWORD dwTimerID, DWORD dwTimeSpan, __DIRECT_TIMER_HANDLER lpHandler, LPVOID lpHandlerParam, DWORD dwTimerFlags);	
用途	设置一个定时器。	
参数	取值（或类型）	含义
dwTimerID	DWORD	设置的定时器对象 ID
dwTimeSpan	DWORD	以 ms 计的定时器间隔
lpHandler	__DIRECT_TIMER_HANDLER	回调函数
lpHandlerParam	LPVOID	回调函数参数
dwTimerFlags	TIMER_FLAGS_ONCE	定时器一次失效
	TIMER_FLAGS_ALWAYS	定时器一直有效，直到通过 CancelTimer 函数撤销
返回值类型	HANDLE	
返回值取值	含义	
NULL	设置定时器失败。	
非 NULL	设置定时器成功，返回定时器对象的 HANDLE。	
注意事项	若 lpHandler 不为 NULL，则定时器到时后，直接调用 lpHandler，否则给当前线程发送一个消息。	

CancelTimer

原型	VOID CancelTimer(HANDLE hTimer);
用途	取消由 SetTimer 设置的定时器

参数	取值（或类型）	含义
hTimer	HANDLE	待撤销的定时器对象句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	所撤销的定时器对象，在创建的时候，必须是以 TIMER_FLAGS_ALWAYS 标记创建的。	

```

示例程序：
void Main()
{
HANDLE hTimer1;
HANDLE hTimer2;
MSG msg;
hTimer1 = SetTimer(100,1000,NULL,NULL,TIMER_FLAGS_ALWAYS);
hTimer2 = SetTimer(200,2000,NULL,NULL,TIMER_FLAGS_ONCE);

while(TRUE)
{
    if(GetMessage(&msg)) //Get message from message queue.
    {
        switch(msg.wCommand)
        {
            case MSG_TIMER: //Timer message.
                if(msg.dwParam == 100) //First timer.
                {
                    PrintLine("First timer is OK.");
                }
                else //Second timer.
                {
                    PrintLine("Second timer is OK");
                }
                break;
            }
        }
    }
}
```

```

        case MSG_TERMINAL:
        default:
            goto __END; //Exit the loop.
    }
}
__END:
CancelTimer(hTimer1);
}

```

上述代码创建了两个定时器，一个定时器的 ID 是 100，间隔是 1s，使用标志 `TIMER_FLAGS_ALWAYS` 创建，这样该定时器将一直存在，每间隔 1s 时间，会向当前线程发送一个 `MSG_TIMER` 消息，另外一个定时器的 ID 是 200，间隔 2s，创建为一次定时器，这样一旦第二个定时器到时，则会给当前线程发送一个 `MSG_TIMER` 消息，然后自行销毁。因此，在 `Main` 函数的最后，只销毁了采用 `TIMER_FLAGS_ALWAYS` 创建的定时器。

上述程序的输出，将会是如下格式：

```

First timer is OK
First timer is OK
Second timer is OK
First timer is OK
First timer is OK
... ..

```

即第二个定时器只会发生一次。

另外，创建上述两个定时器的时候，`lpHandler` 参数（`SetTimer` 函数）没有指定，这样定时器到达后，会以消息的形式，通知当前线程，消息对象（`MSG`）的 `dwParam` 成员，保存了定时器的 ID，这样若一个线程设置了多个定时器，可以通过不同的 ID 来进行区分。若在调用 `SetTimer` 的时候，设置了 `lpHandler` 例程，则定时器到时后，会调用 `lpHandler` 函数，而不会发送消息。

## 核心线程同步操作接口

### Sleep

原型	VOID Sleep(DWORD dwMillionSeconds);	
用途	当前线程进入睡眠状态，时间由参数决定	
参数	取值（或类型）	含义
dwMillionSeconds	任意整数	以毫秒计算的待睡眠时间
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	当前的实现中，睡眠的最小时间是一个时钟中断周期（在 PC 上，为 10ms），因为线程的唤醒操作，只会在系统时钟中断处理程序中发生。	

示例程序：

```
while(!_inp(IDE_STATUS) & 0x80)
{
    dwCounter--;
    if(!dwCounter)
        break;
    Sleep(100);
}
```

上述代码检查 IDE 接口控制器的状态寄存器，判断控制器是否处于忙（BUSY）状态，若是，则睡眠 100ms，然后再次尝试。另外，为了避免出现死循环（比如，硬盘控制器坏，导致状态寄存器一直处于忙状态），还设定了一个计数器，在尝试指定的次数后，将放弃。

### CreateMutex

原型	HANDLE CreateMutex();
用途	创建一个互斥体对象。

参数	取值（或类型）	含义
—	—	—
返回值类型	HANDLE	
返回值取值	含义	
NULL	创建互斥体失败。	
非 NULL	成功创建一个互斥体对象，返回其 HANDLE。	
注意事项	当前的实现中，该对象不支持递归调用和安全删除功能。	

ReleaseMutex

原型	VOID ReleaseMutex(HANDLE hMutex);	
用途	释放互斥体对象	
参数	取值（或类型）	含义
hMutex	HANDLE	待释放的互斥体对象的句柄。
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	释放获取的互斥体对象，线程调用 WaitForThisObject(Ex)，获得对象之后，必须调用该函数释放 Mutex 对象。	

DestroyMutex

原型	VOID DestroyMutex(HANDLE hMutex);	
用途	销毁创建的 Mutex 对象。	
参数	取值（或类型）	含义
hMutex	HANDLE	待销毁的互斥体对象的句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	互斥体对象销毁前，必须确保没有线程等待该对象，否则会有问题，因为目前的实现，该对象不支持安全删除功能。	

## CreateEvent

原型	HANDLE CreateEvent(BOOL bInitStatus);	
用途	创建一个事件对象（用于同步）。	
参数	取值（或类型）	含义
bInitStatus	TRUE	事件对象创建完成后，处于发信号状态
	FALSE	事件对象创建完成后，处于非发信号状态
返回值类型	HANDLE	
返回值取值	含义	
NULL	创建事件对象失败	
非 NULL	创建事件对象成功，返回该对象的 HANDLE。	
注意事项		

## SetEvent

原型	DWORD SetEvent(HANDLE hEvent);	
用途	设置事件对象状态为发信号状态	
参数	取值（或类型）	含义
hEvent	HANDLE	待设置的事件对象句柄
返回值类型	DWORD	
返回值取值	含义	
整数值	该函数调用前，信号量的状态。	
注意事项	可以对一个信号量进行多次设置操作。	

## ResetEvent

原型	DWORD ResetEvent(HANDLE hEvent);	
用途	复位事件对象状态，使得事件对象处于非发信号状态。	
参数	取值（或类型）	含义
hEvent	HANDLE	待复位的事件对象的句柄。
返回值类型	DWORD	
返回值取值	含义	

整数值	ResetEvent 函数调用前该事件对象的状态。
注意事项	该函数也可多次应用到同一个事件对象。

DestroyEvent

原型	VOID DestroyEvent(HANDLE hEvent);	
用途	销毁一个事件对象	
参数	取值（或类型）	含义
hEvent	HANDLE	待销毁的事件对象句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	必须确保没有任何线程等待该事件对象，也必须确保没有任何线程会进一步引用该事件对象，因为当前的实现，不支持安全删除功能。	

WaitForThisObject

原型	DWORD WaitForThisObject(HANDLE hHandle);	
用途	等待一个同步或互斥对象	
参数	取值（或类型）	含义
hHandle	非 NULL 的 HANDLE 值	等待的同步对象的句柄。
返回值类型	DWORD	
返回值取值	含义	
0	尝试等待的时候发生阻塞，再次被唤醒而返回，则返回 0	
1	尝试等待的资源可用，直接返回的情况下，返回该值。	
注意事项	该函数不支持超时等待操作。	

示例程序：

```
HANDLE hMutex = NULL; //Declare as global variable.  
DWORD dwGlobalVar = 10000L; //Global variable shared by two threads.
```

```

DWORD UserThread(LPVOID) //First user thread.
{
while(TRUE)
{
    WaitForThisObject(hMutex);
    if(dwGlobalVar > 0)
        dwGlobalVar --;
    ReleaseMutex(hMutex);
    if(0 == dwGlobalVar)
        break;
}
return 0L;
}

VOID Main()
{
hMutex = CreateMutex(); //Create a mutex object.
if(NULL == hMutex) //Can not create mutex object.
    return;

HANDLE hThread1 = CreateKernelThread(UserThread);
HANDLE hThread2 = CreateKernelThread(UserThread);

WaitForThisObject(hThread1); //Block to wait for thread,until the kernel thread run
over.
WaitForThisObject(hThread2);
DestroyKernelThread(hThread1); //Destroy kernel thread object.
DestroyKernelThread(hThread2);
DestroyMutex(hMutex); //Destroy the mutex object.
return;
}

```

上述代码非常简单，Main 函数创建了两个线程（公用一段代码），每个线程都试图

递减一个全局变量（共享变量）dwGlobalVar，当递减到 0 的时候，线程退出。由于两个线程都试图递减该变量，因此，为了防止出现冲突，使用互斥体对象（Mutex）来完成线程之间的互斥访问。在 Main 函数的最后，删除了创建的两个线程，以及互斥体对象。

上述代码只是作为示例，实际上，对于简单的递减操作，在单 CPU 的情况下是不会出现共享问题的，因为一个递减操作，可能会被编译成一条指令（比如，IA32 构架下的 DEC 指令）。但如果共享的是一个涉及到多个变量的结构体，则需要有上述共享机制。

WaitForThisObjectEx

原型	DWORD WaitForThisObjectEx(HANDLE hHandle,DWORD dwMillionSeconds);	
用途	等待一个同步对象。	
参数	取值（或类型）	含义
hHandle	HANDLE	等待的目标对象
dwMillionSeconds	DWORD	尝试等待的时间
返回值类型	DWORD	
返回值取值	含义	
0	等待的时候发生阻塞，返回该值	
1	等待的资源直接可用，没有发生阻塞，直接返回	
2	等到超时	
注意事项	该函数支持超时等待功能，若 dwMillionSeconds 设置为 0，则功能与 WaitForThisObject 相同。	

示例程序：

```
HANDLE hEvent = NULL; //Global variable shared by device driver and user application.

DWORD DrvInterrupt() //Device driver's interrupt handler.
{
    for(DWORD i = 0;i < 256;i ++)
        Buffer[i] = _inp(IDE_DATA);
    SetEvent(hEvent);
}
```

```

}

VOID Main()
{
    DWORD dwResult = 0L;
    hEvent = CreateEvent(FALSE);
    InstallIdeDriver(DrvInterrupt);
    dwResult = WaitForThisObjectEx(hEvent,1000);
    switch(dwResult)
    {
        case 0:
        case 1:
            PrintLine("Read OK");
            break;
        case 2:
            PrintLine("Can not read device,waiting time out.");
            break;
    }
    UninstallDriver();
    DestroyEvent(hEvent);
}

```

上述代码说明了如何使用 `WaitForThisObjectEx` 函数。其中，`DrvInterrupt` 是一个中断处理程序，用于完成设备的读取操作，并设置事件状态。`Main` 函数首先创建事件对象用于完成中断和应用的同步，安装中断处理程序（加载设备驱动程序），然后调用 `WaitForThisObjectEx`，进入超时等待。若超过 1000ms 没有收到中断，则超时返回，否则说明设备读取操作成功。需要注意的是，在创建 `Event` 对象的时候，其初始状态为非信号状态，这样可确保应用程序开始的时候，进入阻塞状态（超时等待），只有在中断发生，设置事件对象之后，应用程序线程才会被唤醒。

另外一个需要注意的地方，就是在销毁事件对象前，必须确保没有任何其它线程（或中断处理程序）操作该事件对象，否则会发生问题。

系统中断操作接口

ConnectInterrupt

原型	HANDLE      ConnectInterrupt(__INTERRUPT_HANDLER lpHandler, LPVOID lpHandlerParam, UCHAR ucVector, DWORD dwReserved);	
用途	连接一个中断向量	
参数	取值（或类型）	含义
lpHandler	__INTERRUPT_HANDLER	中断处理函数
lpHandlerParam	LPVOID	中断处理函数参数
ucVector	UCHAR	中断向量
dwReserved	DWORD	保留将来使用
返回值类型	HANDLE	
返回值取值	含义	
NULL	连接中断向量失败	
非 NULL	连接中断向量成功，返回一个句柄，这个句柄用于取消中断连接操作。	
注意事项		

DisconnectInterrupt

原型	VOID DisconnectInterrupt(HANDLE hInterrupt);	
用途	取消中断向量的连接	
参数	取值（或类型）	含义
hInterrupt	HANDLE	取消的中断对象句柄
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	hInterrupt 必须是 ConnectInterrupt 函数的返回值。	

示例程序：

```

BOOL IDEIntHandler(LPVOID)    //Interrupt handler of IDE controller's.
{
    UCHAR ucStatus = __inp(IDE_STATUS);
    if(IDE_STATUS & HAS_INTERRUPT) //The interrupt is trigged by IDE.
        Process();
    else
        return FALSE;
    return TRUE;
}

```

```

HANDLE hInterrupt = NULL; //Used to save interrupt object's HANDLE.

```

```

BOOL DriverEntry(__DRIVER_OBJECT* lpDriver)
{
    hInterrupt = ConnectInterrupt(IDEIntHandler,NULL,0x14,0L);
    ... ..
    return TRUE;
}

```

```

VOID UnloadDriver(__DRIVER_OBJECT* lpDriver)
{
    DisconnectInterrupt(hInterrupt);
}

```

上述代码简单描述了一个 IDE 接口控制器的驱动程序，其中 `DriverEntry` 是驱动程序的入口点，在 `DriverEntry` 函数中，连接了中断向量 `0x14` 和中断处理程序。对于中断处理程序，在被调用后，首先必须判断该中断是否是本驱动程序所驱动的设备触发的（因为有可能多个设备共享同一个中断向量），若不是，则尽快返回 `FALSE`，这样系统会继续调用同一中断向量的其它中断处理程序，若是，则进行处理，处理完毕后，返回 `TRUE`，这样操作系统会停止检查其它的中断处理程序。

最后，在驱动程序的卸载例程（`UnloadDriver`）里，调用 `DisconnectInterrupt`，撤销中断向量和中断处理程序的连接。

输入/输出（IO）接口

CreateFile

原型	HANDLE CreateFile(LPSTR lpszFileName, DWORD dwAccessMode, DWORD dwOperationMode, DWORD dwReserved);	
用途	打开一个文件（或设备）对象。	
参数	取值（或类型）	含义
lpszFileName	LPSTR	待打开的文件或设备名
dwAccessMode	FILE_ACCESS_READ	以只读方式打开
	FILE_ACCESS_WRITE	以只写入方式打开
	FILE_ACCESS_RW	以读写方式打开
dwOperationMode	FILE_OPERATION_READ	其它线程可以只读方式打开同一文件
	FILE_OPERATION_WRITE	其它线程可以只写方式打开同一文件
	FILE_OPERATION_RW	其它线程可以读写方式打开同一文件
	FILE_OPERATION_NOACCESS	不允许其它线程打开该文件
dwReserved	DWORD	保留
返回值类型	HANDLE	
返回值取值	含义	
NULL	打开文件失败	
非 NULL	打开文件成功，返回打开的文件对象句柄。	
注意事项	可使用该函数打开文件或设备进行操作，当前 Hello China 的实现（V1.0），只支持打开设备的操作。	

## ReadFile

原型	BOOL ReadFile(HANDLE hFile, DWORD dwByteSize, LPVOID lpBuffer, DWORD* lpdwReadSize);	
用途	从文件（或设备）对象中读取数据。	
参数	取值（或类型）	含义
hFile	HANDLE	待读取的文件或设备对象句柄
dwByteSize	DWORD	待读取的数据尺寸
lpBuffer	LPVOID	缓冲区指针
lpdwReadSize	DWORD*	实际读取的字节数
返回值类型	BOOL	
返回值取值	含义	
TRUE	读取文件成功	
FALSE	读取文件失败	
注意事项	<p>若当前文件指针到达了文件末尾，则该函数仍然返回 TRUE，不过实际读取的字节数（*lpdwReadSize）为 0。</p> <p>若试图读取一个不存在的文件，或传递了一个非法的参数（比如非法句柄、缓冲区指针为 NULL 等），则该函数返回 FALSE。</p>	

示例程序：

## WriteFile

原型	BOOL WriteFile(HANDLE hFile, DWORD dwByteSize, LPVOID lpBuffer, DWORD* lpdwWrittenSize);	
用途	向一个文件对象或设备对象中写入数据。	
参数	取值（或类型）	含义
hFile	HANDLE	待写入的文件或设备对象句柄
dwByteSize	DWORD	待写入的数据大小，以字节计
lpBuffer	LPVOID	待写入数据所在的缓冲区

lpdwWrittenSize	DWORD*	实际写入的字节数量
返回值类型	BOOL	
返回值取值	含义	
TRUE	写入成功	
FALSE	写入失败	
注意事项	若因为存储介质满等原因导致写入失败，该函数仍然返回 TRUE，不过这时候，实际写入的字节数为 0（*lpdwWrittenSize 为 0）。 只有在参数错误、文件句柄无效等情况下，该函数才返回 FALSE。	

IoControl

原型	BOOL IoControl(HANDLE hFile, DWORD dwCommand, DWORD dwInputLen, LPVOID lpInputBuff, DWORD dwOutputLen, LPVOID lpOutputBuff);	
用途	实现特殊的文件或设备控制功能。	
参数	取值（或类型）	含义
hFile	HANDLE	待控制的文件或设备对象句柄
dwCommand	DWORD	控制命令（由特定设备决定）。
dwInputLen	DWORD	输入缓冲区大小
lpInputBuff	LPVOID	输入缓冲区首指针
dwOutputLen	DWORD	输出缓冲区大小
lpOutputBuff	LPVOID	输出缓冲区首指针
返回值类型	BOOL	
返回值取值	含义	
TRUE	函数操作成功	
FALSE	函数操作失败	
注意事项	具体的操作命令，由特定应用程序和特定设备驱动程序决定，操作系统只起到一个透传作用。	

## SetFilePointer

原型	BOOL SetFilePointer(HANDLE hFile, DWORD dwWhereBegin, INT nOffset);	
用途	移动文件当前指针	
参数	取值（或类型）	含义
hFile	HANDLE	待移动指针的文件或设备对象句柄。
dwWhereBegin	FILE_POSITION_CURRENT	从当前位置移动
	FILE_POSITION_BEGIN	从文件开头移动
	FILE_POSITION_END	相对文件结尾开始移动
nOffset	INT	待移动的偏移量
返回值类型	BOOL	
返回值取值	含义	
TRUE	移动文件指针成功	
FALSE	移动文件指针失败	
注意事项	若相对于文件末尾来移动，则 nOffset 必须为负值。	

## FlushFile

原型	VOID FlushFile(HANDLE hFile);	
用途	用于把文件或设备中的临时数据冲刷入永久存储介质	
参数	取值（或类型）	含义
hFile	HANDLE	待冲刷的文件或设备对象句柄。
返回值类型	无	
返回值取值	含义	
—	—	
注意事项		

CloseFile

原型	VOID CloseFile(HANDLE hFile);	
用途	关闭打开的文件（或设备）对象。	
参数	取值（或类型）	含义
hFile	HANDLE	待关闭的文件或设备对象句柄。
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	hFile 必须是调用 CreateFile 打开的文件或设备句柄对象。	

示例程序：

```
HANDLE hFileSrc = NULL;
HANDLE hFileDes = NULL;
UCHAR  Buffer[512];
BOOL bResult = FALSE;
DWORD dwReadSize = 0;
DWORD dwWrittenSize = 0;
```

```
hFileSrc =
CreateFile(“\\\\.\\IDE0”,FILE_ACCESS_READ,FILE_OPERATION_NOACCESS,0L);
if(NULL == hFile) //Open file failed.
goto __END;
hFileDes =
CreateFile(“\\\\.\\IDE1”,FILE_ACCESS_WRITE,FILE_OPERATION_NOACCESS,0L);
if(NULL == hFile)
goto __END;

do{
bResult = ReadFile(hFileSrc,512,(LPVOID)Buffer,&dwReadSize);
if(!bResult) //Read file failed.
{
PrintLine(“Exception encounter.”);
```

```
        break;
    }
    bResult = WriteFile(hFileDes,512,(LPVOID)Buffer,&dwWrittenSize);
    if(!bResult)
    {
        PrintLine("Exception encounter.");
        break;
    }
}while(dwReadSize == 512);

__END:
if(hFileSrc)
CloseFile(hFileSrc);
if(hFileDes)
CloseFile(hFileDes);
```

上述代码完成一个简单的磁盘拷贝功能，首先，以只读方式打开 IDE0，然后以只写方式打开 IDE1，随后进入一个循环，每次从 IDE0 中读取一个扇区，然后写入 IDE1 中，直到 dwReadSize 不等于 512，这时候，实际上是已经到达了磁盘的末尾。

在上述代码的最后，调用 CloseFile 关闭了打开的两个文件（设备对象）。需要注意的是，除非发生异常，比如设备打开期间被拔出，或者传递了一个非法的文件句柄，ReadFile 和 WriteFile 不会返回 FALSE，即使到了文件（或设备）的末尾。判断到达文件或设备末尾的唯一方式，就是判断实际读写的字节数目（dwReadSize 或 dwWrittenSize）。

## 设备驱动程序接口

### CreateDevice

原型	__DEVICE_OBJECT* CreateDevice(LPSTR lpszDevName, __DRIVER_OBJECT* lpDrvObj, LPVOID lpDevExtension);
用途	创建一个设备对象，一般由设备驱动程序调用。

参数	取值（或类型）	含义
lpDevName	LPSTR	创建的设备名字
lpDevObj	__DRIVER_OBJECT*	驱动程序对象，由 IOManager 提供
lpDevExtention	LPVOID	设备扩展，用于存储设备私有数据
返回值类型	__DEVICE_OBJECT*	
返回值取值	含义	
NULL	创建设备对象失败	
非 NULL	创建设备对象成功，返回创建的设备对象指针	
注意事项	该函数一般由设备驱动程序调用，用于完成自身管理的设备的注册。	

DestroyDevice

原型	VOID DestroyDevice(__DEVICE_OBJECT* lpDevObj);	
用途	销毁创建一个设备对象。	
参数	取值（或类型）	含义
lpDevObj	__DEVICE_OBJECT*	待销毁的设备对象指针
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	该函数由设备驱动程序调用，完成设备对象的销毁，销毁的设备对象，必须是调用 CreateDevice 创建的，否则会出问题。	

示例程序：

```
typedef struct{
    __DEVICE_OBJECT* lpDevObj;
    WORD wSectorSize; //Bytes per sector.
    DWORD dwSectorNum; //Total sector of this partition.
    DWORD dwStartSector; //Start sector in IDE disk.
    DWORD dwCurrentSector; //Current position of driver header.
```

```

}IDE_EXTENSION;

BOOL DriverEntry(__DRIVER_OBJECT* lpDrvObj)
{
    __DEVICE_OBJECT* lpDevObj = NULL;
    IDE_EXTENSION* lpExtension = NULL;

    lpExtension = KMemAlloc(sizeof(IDE_EXTENSION),KMEM_SIZE_TYPE_ANY);
    InitExtension(lpExtension);
    ... ..
    lpDevObj = CreateDevice("\\\\.\\IDE0",lpDrvObj,(LPVOID)lpExtension);
    if(NULL == lpDevObj) //Failed to create device object.
        goto __END;
    lpExtension->lpDevObj = lpDevObj;

__END:
    if(NULL == lpDevObj) //Create device failed.
    {
        KMemFree(lpExtension,KMEM_SIZE_TYPE_ANY,0L);
        return FALSE;
    }
    return TRUE;
}

VOID UnloadDriver(__DRIVER_OBJECT* lpDrvObj)
{
    __DEVICE_OBJECT*lpDevObj = lpDrvObj->lpDevObj;
    ... ..
    DestroyDevice(lpDevObj);
    ... ..
    return;
}

```

上述代码示意了如何在设备驱动程序中调用 CreateDevice 和 DestroyDevice 注册设

备。首先，设备扩展是一个设备专用的数据结构，用于存放设备本身特定的数据，为了保存设备对象，在设备扩展中专门设置一个变量（IDE\_EXTENSION 定义中的 lpDevObj 变量），用于记录创建的设备对象。一旦驱动程序调用 CreateDevice 注册了设备对象，应用程序就可以调用 CreateFile 打开该对象进行操作了（通过指定设备名字），因此，在设备驱动程序的初始化（DriverEntry 函数）过程中，一般是在设备初始化全部完成之后，才创建设备驱动程序。

在驱动程序被卸载的时候，有调用 DestroyDevice，销毁了创建的设备对象，这样应用程序如果再指定相应的设备名，调用 CreateFile 函数，则会取得一个失败的结果。

相关辅助功能接口

StrLen

原型	DWORD StrLen(LPSTR lpszStr);	
用途	获取一个字符串的长度，字符串以 0 结尾。	
参数	取值（或类型）	含义
lpszStr	LPSTR	字符串首地址
返回值类型	DWORD	
返回值取值	含义	
任意整数	字符串的长度	
注意事项	若以 lpszStr = NULL 调用该函数，仍然返回 0。	

StrCpy

原型	DWORD StrCpy(LPSTR lpszDes,LPSTR lpszSrc);	
用途	字符串拷贝函数	
参数	取值（或类型）	含义
lpszDes	LPSTR	目标位置首地址
lpszSrc	LPSTR	源字符串首地址
返回值类型	DWORD	
返回值取值	含义	
任意整数	拷贝的字符串长度（字符个数）。	
注意事项	目的缓冲区长度，必须大于源字符串长度，该函数不做缓冲	

	区长度检查。
--	--------

MemZero

原型	VOID MemZero(LPVOID lpMem,DWORD dwSize);	
用途	缓冲区清零操作	
参数	取值（或类型）	含义
lpMem	LPVOID	待清零的缓冲区的首地址
dwSize	DWORD	以字节计的缓冲区长度
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	—	

MemCpy

原型	VOID MemCpy(LPVOID lpDes,LPVOID lpSrc,DWORD dwSize);	
用途	内存块拷贝操作	
参数	取值（或类型）	含义
lpDes	LPVOID	待拷贝的目标缓冲区
lpSrc	LPVOID	源缓冲区
dwSize	DWORD	待拷贝长度（字节计）
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	源和目标缓冲区长度，必须大于或等于 dwSize，该函数不作缓冲区长度合法性检查。	

PC 服务接口

PrintLine

原型	VOID PrintLine(LPSTR lpszMsg);	
用途	在计算机屏幕上输出一个字符串	
参数	取值（或类型）	含义
lpszMsg	LPSTR（字符串指针）	待输出的字符串首地址，字符串以 0 结束
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	仅在 PC 机上有效	

示例程序：

```
... ..
PrintLine(“Hello, China!”);
... ..
```

PrintChar

原型	VOID PrintChar(UCHAR ucForeClr,UCHAR ucBackClr,CHAR cChar);	
用途	在计算机屏幕的当前位置，打印一个字符，可以指定前景和背景颜色。	
参数	取值（或类型）	含义
ucForeClr	0—15 之间的整数	前景颜色
ucBackClr	0—15 之间的整数	背景颜色
ucChar	任意 8 比特数据	要输出的字符
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	该函数用于在 PC 机上输出一个字符，非个人计算机平台上无效。	

### ChangeLine

原型	VOID ChangeLine();	
用途	换行，在计算机的屏幕上，由当前行更换到下一行的同一位置	
参数	取值（或类型）	含义
—	—	—
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	目标平台必须是 PC 平台，其它硬件平台上无效。	

### GotoHome

原型	VOID GotoHome();	
用途	把 PC 屏幕上光标的位置移动到当前行的起始处。	
参数	取值（或类型）	含义
—	—	—
返回值类型	无	
返回值取值	含义	
—	—	
注意事项	目标平台必须是 PC 平台，其它硬件平台上无效。	

示例程序：

```

... ..
ChangeLine(); //Change to next line.
GotoHome(); //Move to begin of this line.
PrintChar(CLR_BLACK,CLR_WHITE,'H');
PrintChar(CLR_BLACK,CLR_WHITE,'e');
PrintChar(CLR_BLACK,CLR_WHITE,'l');
PrintChar(CLR_BLACK,CLR_WHITE,'l');
PrintChar(CLR_BLACK,CLR_WHITE,'o');
PrintChar(CLR_BLACK,CLR_WHITE,',');

```

```
PrintChar(CLR_BLACK,CLR_WHITE,'C');  
PrintChar(CLR_BLACK,CLR_WHITE,'h');  
PrintChar(CLR_BLACK,CLR_WHITE,'i');  
PrintChar(CLR_BLACK,CLR_WHITE,'n');  
PrintChar(CLR_BLACK,CLR_WHITE,'a');  
PrintChar(CLR_BLACK,CLR_WHITE,'!');  
... ..
```

上述示例程序的执行结果为，换行，并退到所换行的开始处，然后打印出白底黑字的“Hello,China!”字符串。



## 第十二章 **Hello China** 的应用开发方法

## 12.1 Hello China 的开发方法概述

嵌入式操作系统一个最大的特点，就是操作系统核心代码（模块）跟应用程序一起链接，这样在嵌入式系统上开发应用程序，必须把操作系统的核心代码（或核心代码编译而成的中间模块）作为应用程序整体工程的一部分。在 Hello China 的开发过程中，也是如此。

本章介绍如何在 Hello China 基础上定制一个简单的应用程序，以及如何修改 Hello China 的内核，包括在内核中添加功能模块等。

## 12.2 在 Hello China 基础上开发一个简单的应用程序

在 Hello China 操作系统核心基础上，开发一个简单的应用程序，包含下列步骤：

1. 启动集成编程环境，并把 Hello China 导入，如下图：

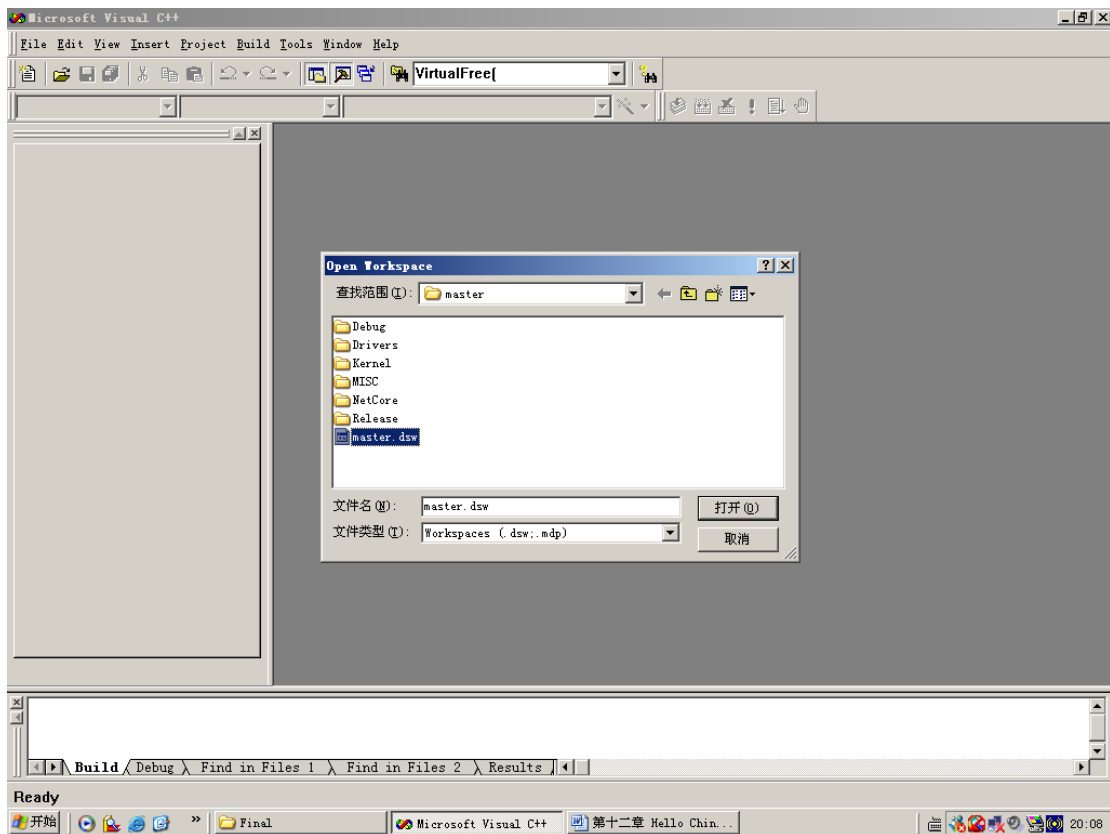


图 12-1 倒入 Hello China 的工作区文件

通过在 Visual C++ 6.0 集成开发环境中，选择 File -> Open workspace...菜单，可打开工作区。

- 2. 在工作区中新建一个 C（或 C++）源文件，可通过选择 File -> New...来完成，如下图：

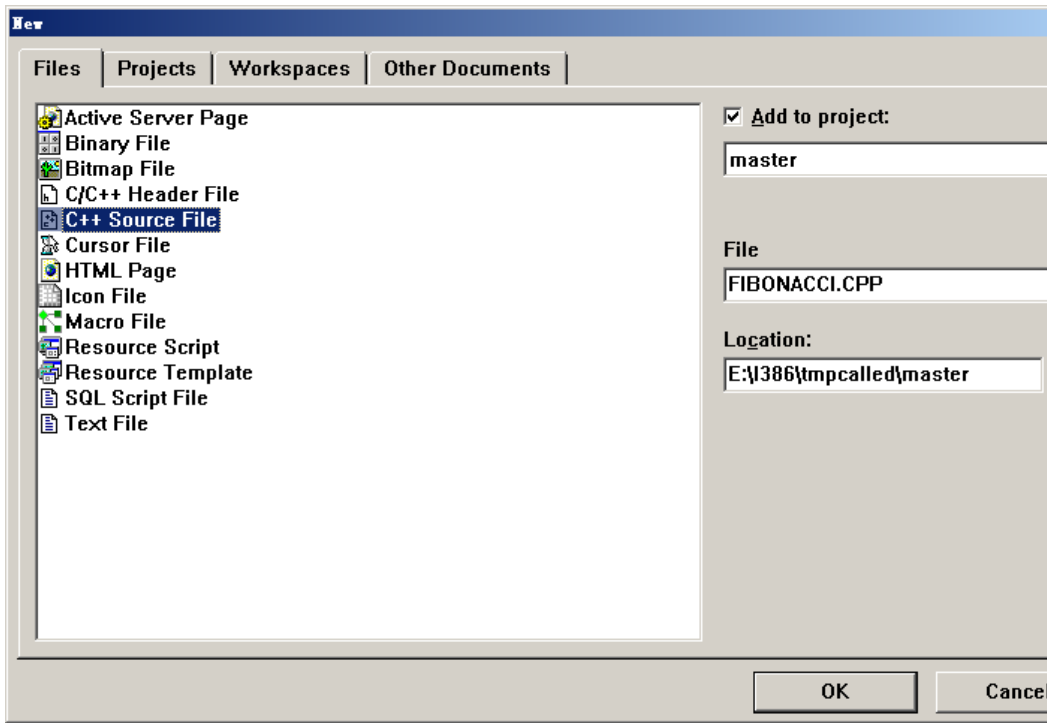


图 12-2 在工作区中添加一个新文件

- 3. 完成源代码文件的添加之后，就可以在此基础上编译代码了，需要注意的是，一定要把应用程序的源文件，放在 Hello China 的跟目录（即 Kernel、Drivers 等所在的目录）下。

4. 若引用 Hello China 提供的 API，则必须在源文件中包含头文件“HCNAPI.H”，下面的代码，实现了一个简单的计算 Fibonacci 数列的程序：

```
//
//Application simple of Hello China.
//

#include "HCNAPI.H"
#include "FIBONACCI.H"

typedef struct{
    DWORD dwInitNum;
    DWORD dwResult;
}__FIBONACCI_CONTROL_BLOCK;

DWORD CalculateThread(LPVOID lpParam) //Calculate fibonacci sequence's element.
{
    __FIBONACCI_CONTROL_BLOCK* lpControlBlock =
    (__FIBONACCI_CONTROL_BLOCK*)lpParam;
    DWORD dwS0 = 0;
    DWORD dwS1 = 1;
    DWORD dwS2 = 0;

    if(NULL == lpControlBlock)
        return 0L;
    for(DWORD i = lpControlBlock->dwInitNum; i > 0; i --)
    {
        dwS2 = dwS0 + dwS1;
        dwS0 = dwS1;
        dwS1 = dwS2;
    }
    lpControlBlock->dwResult = dwS0;
    return 1L;
}
```

```

DWORD Fibonacci(LPVOID lpParam)
{
    LPSTR lpszParam = (LPSTR)lpParam;
    __FIBONACCI_CONTROL_BLOCK ControlBlock[5] = {0};
    HANDLE hThread[5] = {NULL};
    CHAR Buffer[12];
    DWORD dwCounter;
    DWORD dwIndex,i;

    PrintLine("Fibonacci application running...");
    ChangeLine();
    GotoHome();

    if(NULL == lpszParam) //Invalidate parameter.
        return 0L;

    dwCounter = 0;
    for(i = 0;i < 5;i ++)
    {
        while(' ' == lpszParam[dwCounter])
            dwCounter ++; //Skip the space character(s).

        dwIndex = 0;
        while((lpszParam[dwCounter] != ' ') && lpszParam[dwCounter])
        {
            Buffer[dwIndex] = lpszParam[dwCounter];
            dwIndex ++;
            dwCounter ++;
        }

        Buffer[dwIndex] = 0;
        Str2Hex(Buffer,&ControlBlock[i].dwInitNum); //Convert the parameter to

```

integer.

```

        if(!lpSzParam[dwCounter])
            break;
    }

    i = 5;
    for(i;i > 0;i --)
    {
        hThread[i - 1] = CreateKernelThread(
            0L,      //Stack size,use default.
            KERNEL_THREAD_STATUS_READY, //Status.
            PRIORITY_LEVEL_NORMAL,
            CalculateThread, //Start routine.
            (LPVOID)&ControlBlock[i - 1],
            NULL);

        if(NULL == hThread[i - 1]) //Failed to create kernel thread.
        {
            PrintLine("Create kernel thread failed.");
            break;
        }
    }

    //
    //Waiting for the kernel thread to over.
    //
    WaitForThisObject(hThread[0]);
    WaitForThisObject(hThread[1]);
    WaitForThisObject(hThread[2]);
    WaitForThisObject(hThread[3]);
    WaitForThisObject(hThread[4]);

    //
    //Now,we have calculated the fibonacci number,print them out.

```

```

//
for(i = 0;i < 5;i ++)
{
    Int2Str(ControlBlock[i].dwInitNum,Buffer);
    PrintStr(Buffer);
    PrintStr("'s result is: ");
    Int2Str(ControlBlock[i].dwResult,Buffer);
    PrintStr(Buffer);
    ChangeLine();
    GotoHome();
}

//
//Close the kernel thread.
//
DestroyKernelThread(hThread[0]);
DestroyKernelThread(hThread[1]);
DestroyKernelThread(hThread[2]);
DestroyKernelThread(hThread[3]);
DestroyKernelThread(hThread[4]);

return 1L;
}

```

其中，FIBONACCI.H 文件，声明了 Fibonacci 函数，该文件应该被 EXTCMD.CPP 文件包含，以能够把 Fibonacci 函数，连接到 Hello China 的外部命令列表中。

5. 在外部命令列表（EXTCMD.CPP 文件）中，添加一项，即修改 EXTCMD.CPP 文件，如下：

```

#include ".\Kernel\StdAfx.h"
#include "EXTCMD.H"
#include "FIBONACCI.H"

```

```
__EXTERNAL_COMMAND ExtCmdArray[] = {  
{"fibonacci",NULL,FALSE,Fibonacci},  
{NULL,NULL,FALSE,NULL}  
};
```

具体的外部命令工作方式，请参考本书“Hello China 在 PC 机上的 shell”一章。

上述工作完成之后，Hello China 的工作区中，所包含的文件如下图：

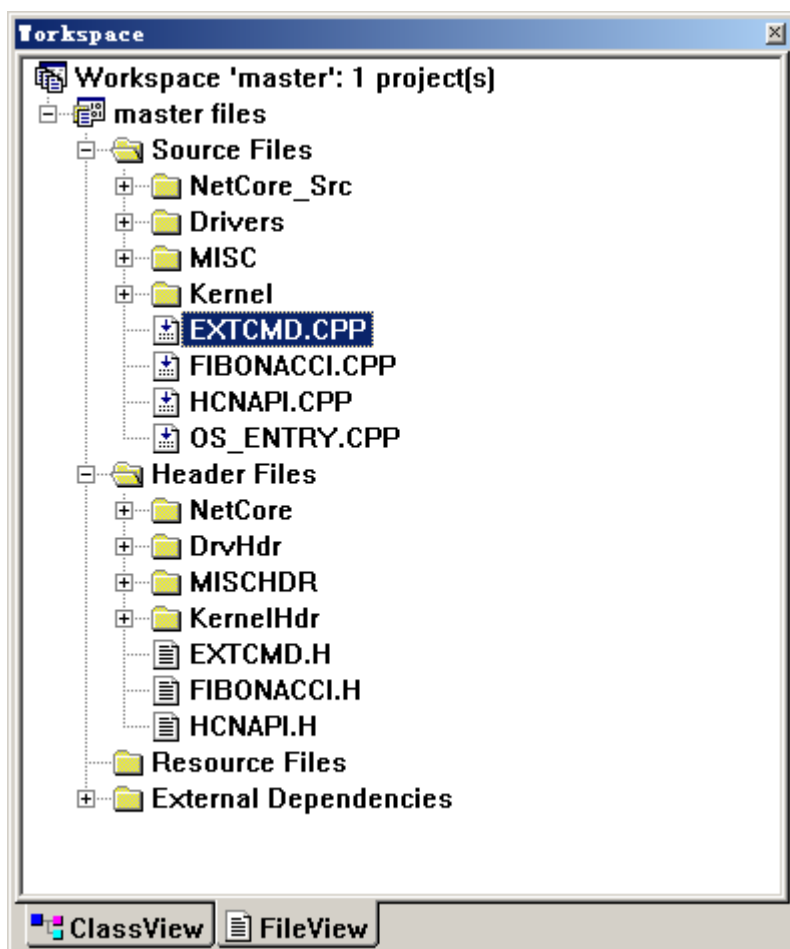


图 12-3 工作区示意

其中，FIBONACCI.CPP 和 FIBONACCI.H 是新增加的应用程序文件，为了把新增加的应用程序连接到 Hello China 的外部命令列表，需要修改 EXTCMD.CPP 文件。

6. 重新编译连接 Hello China，选择编译 Release 版本，如下：

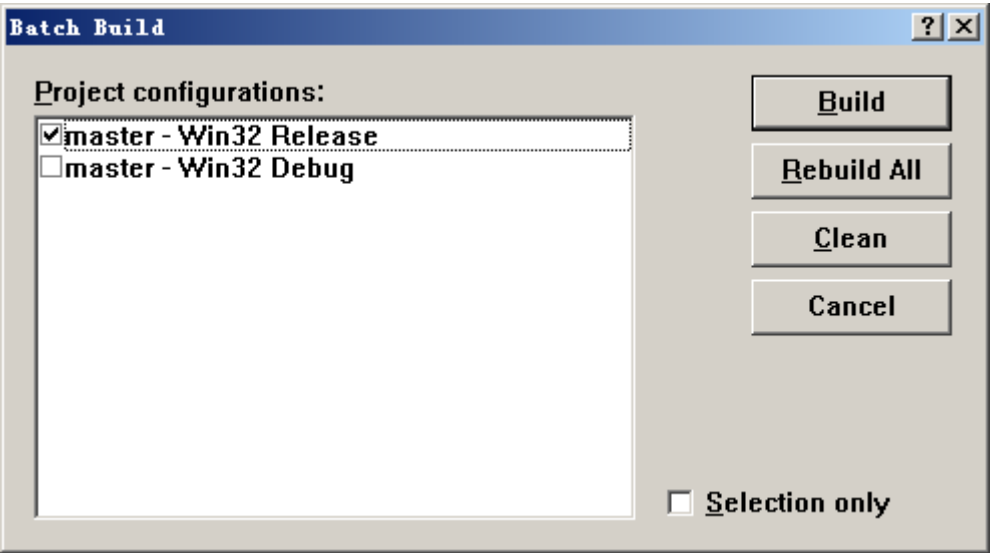


图 12-4 对整个项目进行编译连接

7. 编译完成之后，把编译之后的 MASTER.DLL 文件拷贝到 Hello China 目标模块所在目录（BOOTSECT.BIN、REALINIT.BIN、MINIKER.BIN、FMTLDRF.COM、PROCESS.EXE 等文件都在该目录下），然后运行 PROCESS.EXE，处理 MASTER.DLL 模块，并手工把 MASTER.DLL 改名为 MASTER.BIN。上述操作也可通过编写一个批处理命令完成；
8. 运行 FMTLDRF.COM，重新创建引导软盘，并引导计算机，引导成功之后，在 Hello China 的命令行模式下，敲入“Fibonacci 10 20 30 40 50”，应该可以得到 Fibonacci 第 16、32、48、64、80 项的计算结果，需要注意的是，在命令行模式下，输入的数字都是按 16 进制解释的。

对 Hello China 操作系统核心功能进行修改，与此类似，所不同的是，建议把添加的功能模块源代码文件，放在 Kernel 目录下，并在 StdAfx.H 文件中，包含添加的模块的.H

文件，这样操作系统核心的其它功能模块，就可以直接引用添加的功能模块了。

# 附录 A 串口交互程序及其实现

## 串行通信接口概述

PC 微机的串行通信使用的异步串行通信芯片是 INS 8250 或 NS16450 兼容芯片，统称为 UART(通用异步接收发送器)。对 UART 的编程实际上是对其内部寄存器执行读写操作。因此可将 UART 看作是一组寄存器集合，包含发送、接收和控制三部分。UART 内部有 10 个寄存器，供 CPU 通过 IN/OUT 指令对其进行访问。

这些寄存器的端口和用途见下表所示。其中端口 0x3f8-0x3fe 用于微机上 COM1 串行口，0x2f8-0x2fe 对应 COM2 端口。条件 DLAB(Divisor Latch Access Bit)是除数锁存访问位，是指线路控制寄存器的位 7。

下表给出了 PC 机上串行通信接口的访问端口地址,以及对应的寄存器 bit 位的含义：

访问端口 (COM1/COM2)	访问 方式	DLAB 位 状态	各寄存器 bit 用途
0x3F8/0x2F8	R	0	写发送保持寄存器。含有将发送的字符。
	W	0	读接收缓存寄存器。含有收到的字符。
	R/W	1	读/写波特率因子低字节 (LSB) 。
0x3F9/0x2F9	R/W	1	读/写波特率因子高字节 (MSB) 。
	R/W	0	读/写中断允许寄存器。 位7-4 全0 保留不用； 位3=1 modem 状态中断允许； 位2=1 接收器线路状态中断允许； 位1=1 发送保持寄存器空中断允许；

			位0=1 已接收到数据中断允许。
0x3FA/0x2FA	R	X	<p>读中断标识寄存器。中断处理程序用以判断此次中断是4 种中的那一种。</p> <p>位7-3 全0（不用）；</p> <p>位2-1 确定中断的优先级；</p> <p>= 11 接收状态有错中断，优先级最高；</p> <p>= 10 已接收到数据中断，优先级第2；</p> <p>= 01 发送保持寄存器空中断，优先级第3；</p> <p>= 00 modem 状态改变中断，优先级第4。</p> <p>位0=0 有待处理中断；=1 无中断。</p>
0x3FB/0x2FB	W	X	<p>写线路控制寄存器。</p> <p>位7=1 除数锁存访问位（DLAB）。</p> <p>0 接收器，发送保持或中断允许寄存器访问；</p> <p>位6=1 允许间断；</p> <p>位5=1 保持奇偶位；</p> <p>位4=1 偶校验；=0 奇校验；</p> <p>位3=1 允许奇偶校验；=0 无奇偶校验；</p> <p>位2=1 1 位停止位；=0 无停止位；</p> <p>位1-0 数据位长度：</p> <p>= 00 5 位数据位；</p> <p>= 01 6 位数据位；</p> <p>= 10 7 位数据位；</p>

			= 11 8 位数据位。
0x3FC/0x2FC	W	X	写modem 控制寄存器。 位7-5 全0 保留； 位4=1 芯片处于循环反馈诊断操作模式； 位3=1 辅助用户指定输出2，允许INTRPT 到系统； 位2=1 辅助用户指定输出1，PC 机未用； 位1=1 使请求发送RTS 有效； 位0=1 使数据终端就绪DTR 有效。
0x3FD/0x2FD	R	X	读线路状态寄存器。 位7=0 保留； 位6=1 发送移位寄存器为空； 位5=1 发送保持寄存器为空，可以取字符发送； 位4=1 接收到满足间断条件的位序列； 位3=1 帧格式错误； 位2=1 奇偶校验错误； 位1=1 超越覆盖错误； 位0=1 接收器数据准备好，系统可读取。
0x3FE/0x2FE	R	X	读modem 状态寄存器。δ 表示信号发生变化。 位7=1 载波检测(CD)有效； 位6=1 响铃指示(RI)有效； 位5=1 数据设备就绪(DSR)有效； 位4=1 清除发送(CTS)有效； 位3=1 检测到δ 载波； 位2=1 检测到响铃信号边沿；

			位1=1    δ    数据设备就绪 (DSR) ; 位0=1    δ    清除发送(CTS)。
--	--	--	--

现对上述表格中比较重要的几个寄存器比特进行简要介绍:

- 1、 **DLAB**: 除数锁存访问位, 设置为 1 的时候, 用于设置波特率因子。设置为 0 的时候, 用于设置或读取其它寄存器的值。也可以理解为一个标志位, 用于对前两个寄存器 (0x3F8/0x2F8、0x3F9/0x2F9) 的用途进行区别。若为 1, 则这两个寄存器为波特率因子, 否则这两个寄存器被用于发送/接收保持 (缓存) 寄存器和中断控制寄存器;
- 2、 中断允许寄存器: 用端口地址 0x3F9/0x2F9 进行访问, 用于设置或读取 COM 端口的中断控制比特。该寄存器的 0—3 比特用于控制特定的中断源的状态, 4—7 比特保留 (为 0)。其中, 0—3 比特中, 一个比特对应一个特定的中断源, 若设置该比特为 1, 则对应的中断源会打开, 这样一旦有对应的事件发生, COM 接口的控制芯片就会通过 IRQ4 (对于 COM1) 或 IRQ3 (对于 COM2) 中断输入引脚, 给 CPU 发出中断请求。相反, 若设置为 0, 则对应的中断源会被屏蔽。这样即使对应的事件发生, 串行通信控制器也不会引发中断请求。这个时候, 就需要程序自行读取相应的寄存器, 来判断发生的事件及其信息;
- 3、 中断标识寄存器 (0x3FA/0x2FA 地址): 一旦发生中断, 软件可通过读取这个寄存器的值, 来判断发生的中断的类型。需要注意的是, 该寄存器的最后一个比特, 说明是否有中断发生。因为在系统设计的时候, 有可能串行接口跟其它系统模块共用一个中断输入 (比如, 都采用 IRQ4), 这样在发生中断的时候, 可通过该比特来判断发生的中断是否是串行接口发生的。若是, 则进行处理, 否则把处理过程转让给其它设备的中断处理程序。Hello China 的设计支持这种中断共用的模型;
- 4、 线路控制寄存器: 该寄存器用于控制串行接口的工作模式。比如, 在 Windows 操作系统提供的超级终端 (HYPERTRM.EXE 程序) 程序中, 可通过下列对话框设置 COM 接口的工作属性, 实际上内部就是通过修改线路控制寄存器来实现的 (每秒位数除外, 该属性是通过修改波特率因子寄存器实现的);

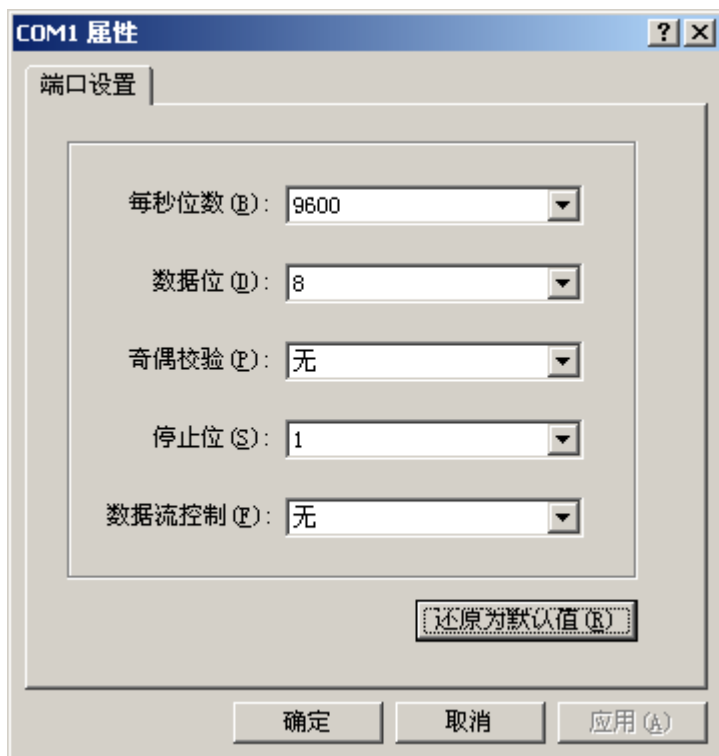


图 13-1 超级终端中串口通信参数的设置

- 5、 波特率因子：用于确定串口工作的波特率，该因子越大，实际的工作速率越小。一般情况下，按照下列表格来设置串口的波特率因子：

Bits Per Second	3F9/3F9 Value	3F8/2F8 Value
110	4	17h
300	1	80h
600	0	C0h
1200	0	60h
1800	0	40h
2400	0	30h
3600	0	20h
4800	0	18h
9600	0	0Ch
19.2K	0	6
38.4K	0	3
56K	0	1

图 13-2 波特率跟波特率因子之间的对应关系

需要注意的是，波特率因子是 16 位的，通过设置 3F9/2F9 和 3F8/2F8 两个寄存器来完成。在设置前，首先应该设置 DLAB 比特为 1，这样在访问上述地址的寄存器的时候，才会访问到波特率因子寄存器。

## 串行通信编程方式

介绍了上述内容之后，对串行接口进行编程就非常简单了，大致可分为下列几个步骤：

### 串口初始化

在开始用发送或接收数据前，必须对串口进行初始化，主要是初始化串口的发送/接收波特率、中断允许方式、奇偶校验、停止位数量、数据位长度等。

比如，下列代码把串口 1 的工作模式，设置为缺省的 Windows 超级终端工作模式（波

特率为 9600，无校验，停止位 1，数据位 8，无流控）：

```
void InitializeCom1()
{
    __outb(0x80,0x3FB); //Set DLAB bit to 1,thus the baud rate
divisor can be set.
    __outb(0x0C,0x3F8); //Set low byte of baud rate divisor.
    __outb(0x0,0x3F9); //Set high byte of baud rate divisor.
    __outb(0x07,0x3FB); //Reset DLAB bit to 0,and set data bit
to 8,1 stop bit,without parity check.
    __outb(0x0D,0x3F9); //Set all interrupts but write buffer
empty.
    __inb(0x3FB); //Read one byte from data port,to reset the
data port.
}
```

## 数据发送

对于个人计算机外部设备的数据发送和接收，一般情况下有两种方式：中断方式和轮询方式（对于一些专用的大型计算机，还可以通过 IO 通道方式进行数据传输）。

轮询方式是发送或接收程序不停的检查外设的状态，一旦发现外设状态可用，便启动数据发送或接收过程。这样即使在外设不可能的时间内，发送或接收程序也不停的运行，会导致无用的 CPU 占用，降低系统效率。但轮询方式编程非常简单。

中断方式则会大大提高系统整体效率。发送或接收程序启动一个发送或接收过程，然后进入睡眠状态。在设备完成发送或接收数据后，通过中断的方式通知 CPU，然后对应的发送或接收程序会被唤醒，从而继续进行数据传输。这个过程对 CPU 的利用会大大降低。但中断方式编程相对复杂。

在下面的部分中，我们对这两种方式下的串口数据传输都做描述。

### *基于轮询方式的数据发送*

轮训方式的数据发送操作比较简单，主要思路是，在发送数据前，检查串口是否准备好，若准备好，则启动发送，否则继续检测。但需要考虑一点，就是避免陷入死循环。下列是一个简单的发送程序，主要完成一个字节字符的发送：

```
BOOL ComSendByte(WORD wPort,BYTE bt)
```

```

{
    UINT nCounter1 = 1024;
    UINT nCounter2 = 3;

    while(nCounter2-- > 0) //Outer loop,for three times.
    {
        while(nCounter1-- > 0) //Inner loop,for 1024 times.
        {
            if(__inb(wPort + 5) & 32) //Send register empty.
            {
                __outb(wPort,bt); //Send out the byte.
                return TRUE;
            }
        }
        nCounter1 = 1024;
        __MicroDelay(1000); //Delay 1ms.
    }
    return FALSE; //Don't wait anymore.
}

```

上面的代码比较简单。首先，定义了两个变量：nCounter1 和 nCounter2，用于控制循环。然后代码开始进入内层循环，首先检查发送保持寄存器是否为空（LSR 寄存器的第 5 个 bit 是否为 1）。若为空，则发送对应的字节，然后返回成功结果。否则一直循环。若内层循环超出了预定的循环次数，则进入外层循环。外层循环通过调用\_\_MicroDelay 函数，来延迟 1ms，然后又可重新进入内层循环。

若外层循环结束（3 次），则该函数将不再试图发送，而是以失败返回。

### *基于中断方式的数据发送*

基于中断方式的数据发送稍微有些复杂。主要实现思路是，首先判断发送保持寄存器是否为空。若是，则直接发送。否则，则发送线程进入睡眠状态，等待发送寄存器恢复。在发送寄存器可用的时候，串口控制芯片会通过中断通知 CPU，从而唤醒发送线程，进而完成数据的发送。这时候，需要考虑两个问题：

- 1、为了提高效率，不要一检测到发送保持寄存器不可用就睡眠，而是稍微等待一段时间，这样可大大提升整体效率。因为线程睡眠涉及到线程上下文的切换，也是十分消耗系统资源的；

- 2、发送线程在睡眠的时候，防止进入永久睡眠，即假如串口发送寄存器始终不可用，发送线程应该能够在一段时间的睡眠之后被唤醒。

下面是一个实现示例：

```

BOOL ComSendByte(WORD wPort, BYTE bt)
{
    UINT nCount = 16; //Used for short delay.
    DWORD dwFlags;
    ResetEvent(g_hEvent) ;
    __ENTER_CRITICAL_SECTION(NULL, dwFlags);
__REPEAT :
    while(nCount-- > 0)
    {
        if(__inb(wPort + 5) & 32) //Send holding register
empty
        {
            __outb(wPort, bt);
            __LEAVE_CRITICAL_SECTION(NULL, dwFlags);
            return TRUE;
        }
    }
    //The sending holding register still occupied after
    //wait a short time, so go to sleep to wait more time.
    DWORD dwResult = WaitForThisObject(g_hEvent, 2000);
    if(OBJECT_WAIT_RESOURCE == dwResult) //Wait
successful.
    {
        nCount = 16;
        goto __REPEAT; //Try again.
    }
    __LEAVE_CRITICAL_SECTION(NULL, dwFlags);
    return FALSE; //Wait time out.
}

```

`g_hEvent` 是一个全局范围内的事件对象，应该在程序开始的时候，完成创建和初始化工作。上述发送过程十分简单，首先检查一下当前串口芯片的发送保持寄存器是否为空（只有为空的时候才能发送）。若为空，则直接通过写入端口，完成数据发送工作，然后返回。需要注意的是，为了提升效率，在检查串口发送保持寄存器状态的时候，采用的是连续检查的策略。即一旦检测到不可用，则会通过循环进行第二次检查，然后是第三次，…。完成十六次检查后，若仍然不可用，则读取线程进入睡眠状态。这样可避免一次检查失败，就导致线程进入睡眠。因为睡眠操作是很费时的，而串口保持寄存器的状态，变化十分迅速。若一次检查不成功，后续检查可能会成功。这样就可以大大提升系统效率。

需要注意的是，上述操作是一个关键区段操作，即不允许中断。因为若这个操作过程中发生中断，可能会导致线程永久睡眠。设想在上述代码中，黑体部分代码执行前，发生一次发送保持寄存器空的中断。中断处理程序会重置事件对象，但此时线程还未进入睡眠状态。这样中断处理程序结束后，写入线程会继续执行，从而进入睡眠状态。这样就有可能永远不会被唤醒了。因此，上述操作，必须在一个关键区段内完成。

下面就是具体的中断处理程序：

```
DWORD Com1IntHandler(LPVOID)
{
    UCHAR isr = __inb(0x3FA);
    while(isr & 1) //Has interrupt to process.
    {
        if(isr & 0x2) //Sending holding register empty.
        {
            SetEvent(g_hEvent); //Set the event to wakeup
                                //sending thread.
        }
        else{
            if(isr & 0x4) //Received data.
            {
            }
        }
        isr = __inb(0x3FA); //Check again.
    }
}
```

```

        return 0L;
    }

```

首先判断发生的中断是否就是由 COM 接口控制芯片引发的中断（判断中断状态寄存器的第一个比特是否为 1）。若是，则进一步判断是什么类型的中断。因为 COM 接口控制芯片可在多种情况下引发中断。然后根据中断类型，做进一步处理。在这里，重点关注中断类型是发送保持寄存器为空中断（ISR 的第二个比特为 1）。若是该类型的中断，则调用 SetEvent 函数，恢复 g\_hEvent 的信号状态，这会唤醒所有阻塞在该信号上的核心线程。

从中断返回后，若有核心线程阻塞在 g\_hEvent 时间对象上，则统统会被唤醒。在合适的调度时机，就会被重新调度执行。这样由于串口控制器的发送保持寄存器已经空了，所以就可顺利完成发送任务。

需要注意的是，中断处理函数对 ISR 的检查是循环的，因为有可能发生中断嵌套的情况，即第一个中断得到处理后，又一个后续的中断立即发生。这样连续的两个中断可在同一个中断处理程序中进行，大大提高系统效率。

## 数据接收

### *基于轮询方式的数据接收*

基于轮询方式的数据发送，与基于轮询方式的数据接收程序类似。下面是一个实现示例：

```

BOOL ComRecvByte(WORD wPort, BYTE* pbt)
{
    UINT nCounter1 = 1024;
    UINT nCounter2 = 3;

    while(nCounter2-- > 0) //Outer loop, for three times.
    {
        while(nCounter1-- > 0) //Inner loop, for 1024 times.
        {
            if(__inb(wPort + 5) & 1) //Data available.
            {
                *pbt = __inb(wPort); //Read the byte.
            }
        }
    }
}

```

```

        return TRUE;
    }
}

nCounter1 = 1024;
__MicroDelay(1000); //Delay 1ms.
}

return FALSE; //Don't wait anymore.
}

```

程序首先判断线路状态寄存器的第一个比特是否为 1。若是 1，则说明数据寄存器中已有接收到的数据，于是通过 `__inb` 函数，把数据读取出来，然后返回。需要注意的是，读取数据寄存器的数据，会导致线路状态寄存器的第一个比特清零。

若数据寄存器中没有数据（线路状态寄存器的第一个比特为 0），则会进入首轮循环（1024 次）。若首轮循环结束后，仍然没有数据到达，则进入外层循环。外层循环会调用 `__MicroDelay` 函数，延迟 1ms，然后再重新尝试。

如果在外层循环结束后，仍然没有数据到达，则会返回 `FALSE`。调用者应该根据该函数的返回结果，确定是否有正确的数据被取回。

### 基于中断方式的数据接收

基于中断方式的数据接收程序，与基于中断方式的数据发送程序类似，也是分两步实现：

- 1、由应用程序主动调用的数据接收函数。该函数检查串口芯片的数据寄存器是否有数据可用，若有，则直接读取后返回。否则进入等待过程。为了提升效率，可在进入等待过程前，多做几次检查，以提升效率；
- 2、中断处理程序。在数据到达的时候，串口控制芯片会通过中断的方式通知 CPU。这时候，CPU 需要唤醒等待读取数据的核心线程。

下面是数据接收函数的实现示例：

```

BOOL ComRecvByte(WORD wPort, BYTE* bt)
{
    UINT nCount = 16; //Used for short delay.
    DWORD dwFlags;
    ResetEvent(g_hEvent) ;
    __ENTER_CRITICAL_SECTION(NULL, dwFlags);

```

```

__REPEAT :
    while(nCount-- > 0)
    {
        if(__inb(wPort + 5) & 1) //Send holding register empty
        {
            *bt = __inb(wPort);
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            return TRUE;
        }
    }
    //The data register still unavailable after
    //wait a short time,so go to sleep to wait more time.
    DWORD dwResult = WaitForThisObject(g_hEvent,2000);
    if(OBJECT_WAIT_RESOURCE == dwResult) //Wait
successful.
    {
        nCount = 16;
        goto __REPEAT; //Try again.
    }
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return FALSE; //Wait time out.
}

```

实现的思路与基于中断方式的数据发送程序类似，在此不做赘述。

## 串口交互程序的实现

在 Hello China V1.5 中，实现了一个串口交互程序，类似于 Windows 自带的超级终端程序，可通过串口，控制外部设备。

对于这个串口通信程序，分别采用基于轮询方式的编程方式和基于中断方式的编程方式进行实现，因此存在两个版本。在 Hello China 启动完成后，通过输入 `hypertrm` 和 `hyptrm2` 命令，就可启动串口输入/输出程序。其中 `hypertrm` 对应轮询方式的实现，而 `hyptrm2` 则是中断方式的实现版本。这两个版本的功能是一致的，但基于中断方式的实现，可大大节约 CPU 资源，但其复杂性也大大增加了。

本章对这两种实现进行详细描述。这两个程序的实现很有典型意义，从中不但可以看到 **Hello China** 的编程方法以及步骤，而且也可以深入体会中断方式和轮询方式的设备控制方式，对于编写任何操作系统的设备驱动程序，都是十分有参考价值的。

## 串口交互程序的使用

在介绍其实现之前，先简单介绍一下串口交互程序的使用，这样可加深读者印象。可通过两种方法验证串口交互程序：

*通过 PC 机的串口，控制特定功能的设备。*

比如，大多数的数据通信设备（路由器、以太网交换机等），都是通过串口来完成配置和管理的。这种通信模型如下：

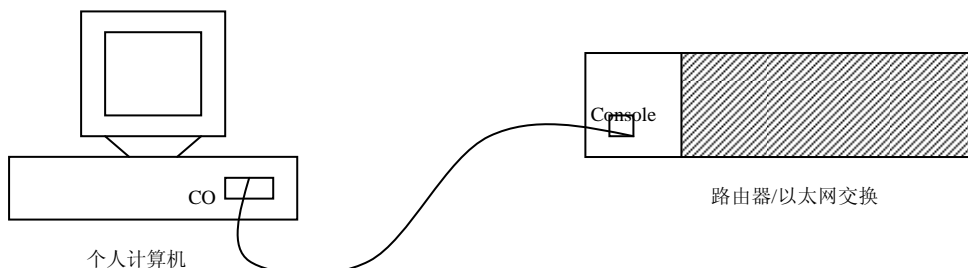


图 13-3 通过 PC 的串口控制网络设备

通过一条特殊的串口线（一头为 RJ45 接头，连接网络设备。另外一头为 DB-9 接头，连接计算机的 COM 接口），连接网络设备的控制端口（一般称为 Console 接口）和 PC 机的 COM 接口。设置合适的通信参数（波特率、数据位数等），在 PC 上启用一个终端模拟软件（比如 Windows 操作系统自带的超级终端），就可对网络设备进行控制了。

在这种方式下，可采用 **Hello China** 启动个人计算机，连接好串口线，然后输入 **hypertrm** 或 **hyptm2**，就可启用串口交互程序。这时候，输入回车键或其它键，就可看到输出了。这种情况下，串口驱动程序实际上是替代了 Windows 的超级终端程序。

需要注意的是，为了实现上的简便，**Hello China V1.5** 实现的串口交互程序，在初始化的时候就设定了串口的波特率和数据位数等参数（波特率 9600、数据位 8 位、无奇偶校验、一位停止位）。幸运的是，大多数网络设备，都是在这种工作模式下工作的。若遇到特殊情况，可通过修改串口交互程序的初始化参数，重新编译来实现。

若要退出 **hypertrm** 或 **hyptm2**，只要输入字母 'z' 就可以了，该字符用于指示线程运行结束。这是不符合实际应用需求的，实际当中，用户可输入 **Ctrl + C** 组合键，用于退

出一个命令行程序。但由于 **Hello China** 当前版本键盘驱动程序的功能缺陷，无法识别 **Ctrl + C** 组合键（实际上无法识别任何组合键），因此暂时用这种简便的方式替代了☺。后续版本中，会完善键盘驱动程序，该问题就可得到解决。

通过串口连接两台计算机，实现点对点通信

在上述应用场景中，需要有一台被控制的设备。但很多情况下，可能没有这样的试验设备供使用。这时候要验证串口交互程序，就可采用两台计算机直连的方式。通过一条直连串口线，连接两台 PC 的 COM1 接口，然后在两台电脑上分别启用 **hypertrm** 或 **hyptm2** 程序，在一台电脑上输入的数据，就可显示在另外一台电脑上，或者相反。这实际上是一个点对点的通信程序。这种情形如下图：

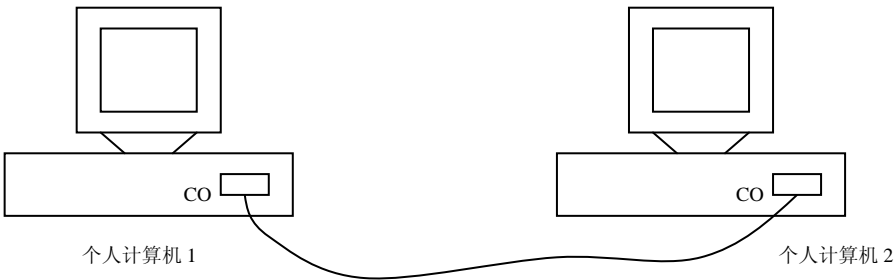


图 13-4 通过串口实现两台个人计算机之间的通信

连接两台电脑的串口线，可采用自行制作，也可到电子市场上购买。若自行制作，则可参考下面介绍的串口线制作方法。

直连串口线的制作方法

在介绍之前先对一些市场常用名词做出解释。现在所有的接头都可以分为公头和母头两大类：

公头：泛指所有针式的接头。

母头：泛指所有插槽式的接头。

按照不同的协议，所有接头的针脚有统一规定，连接时要注意查看。在接线时没有提及的针脚都可悬空。下面给出串口、并口各针脚的功能列表，其中最常用的是 9 针和 25 针的：

25 针串口功能一览

针脚	功能
2	发送数据（TXD）

3	接收数据 (RXD)
4	发送请求 (RTS)
5	发送清除 (CTS)
6	数据准备好 (DSR)
7	信号地 (GND)
8	载波检测 (DCD)
20	数据终端准备好 (DTR)
22	振铃指示 (RI)

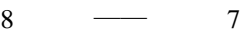
### 9 针串口功能一览表

针脚	功能
1	载波检测 (DCD)
2	接收数据 (RXD)
3	发送数据 (TXD)
4	数据终端准备好 (DTR)
5	信号地 (GND)
6	数据准备好 (DSR)
7	发送请求 (RTS)
8	发送清除 (CTS)
9	振铃指示 (RI)

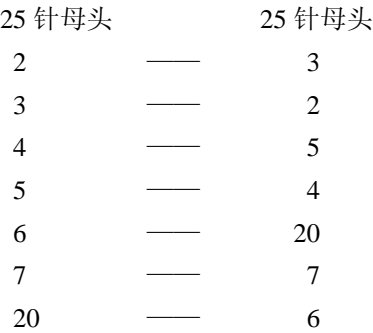
串口直连线主要用于直接把两台电脑的 com 口连接在一起。比较早一点的 AT 架构的计算机的串口有 9 针和 25 针两种，而现在的 XT 架构的电脑两个串口全部是 9 针。于是联机线就分为 3 种 (9 针对 9 针串口直连线, 9 针对 25 针串口直连线, 25 针对 25 针串口直连线)。下面给出每种组合的线缆对应方式:

### 9 针对 9 针串口连接

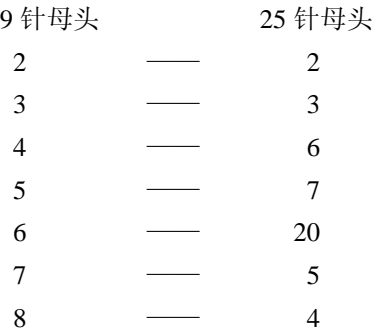
9 针母头		9 针母头
2	——	3
3	——	2
4	——	6
5	——	5
6	——	4
7	——	8



25 针对 25 针串口连接



9 针对 25 针串口连接



现在大多数个人计算机，都是基于 IBM PC/XT 构架的个人计算机，即提供的 COM 接口都是 9 针的公头接口。这样只需用两个 9 针的母头以及对应的线缆，根据上述 9 针对 9 针的连接顺序连接起来，就可以使用了。

轮询模式的串口交互程序实现

轮询模式的 IO 交互程序，有下列几个基本功能模块组成：

- 1、初始化功能，完成串口的初始化工作；
- 2、数据发送功能。该功能接收用户输入，并采用轮询方式发送到串口；
- 3、数据接收功能。该功能模块采用轮询方式，读取到达串口的数据，并打印到屏幕上；
- 4、轮询模式的程序入口，这是一个符合 Hello China 定义的核心线程入口函数，该函数调用串口初始化功能模块，并创建两个核心线程：接收线程和发送线程，然后准备就绪，进入阻塞状态，直到用户退出。

初始化功能模块由 InitComPort 一个函数组成，该函数代码如下：

```
static void InitComPort(WORD base)
{
    if((base != COM1_BASE) && (base != COM2_BASE))
    {
        return;
    }
    __outb(0x80,base + 3); //Set DLAB bit to 1,thus the baud rate
divisor can be set later.
    __outb(0x0C,base);    //Set low byte of baud rate divisor.
    __outb(0x0,base + 1); //Set high byte of baud rate divisor.
    __outb(0x07,base + 3); //Reset DLAB bit,and set data bit to
8,one stop bit,without parity check.
    __outb(0x0,base + 1); //Disable all interrupt enable bits.
    __inb(base); //Reset data register.
}
```

该函数完成串口的初始化工作，其中函数的参数 base，指定了要初始化的串口端口地址。该函数首先判断串口地址是否是 COM1 和 COM2 的端口地址（分别为 0x3F8 和 0x2F8），若不是，则直接返回。因为通常情况下，PC 提供两个串口，每个串口采用固定的端口地址。

完成端口地址的确认后，进入初始化过程。首先设置线路控制寄存器的 DLAB 比特为 1，这样就可写入串口工作的波特率因子。在目前的实现中，波特率硬性设置为 9600，这可适应大多数的应用场景。

完成波特率因子的设置后，恢复 DLAB 比特，并设置数据位为 8 位，一位停止位，不做任何校验。这都是通过写入线路控制寄存器完成的。完成上述设置后，InitComPort 函数禁止了 COM 接口的所有中断，因为该实现是基于轮询方式的，没有安装对应的中断处理程序。若不禁止 COM 接口的中断，则可能会引发系统异常。

最后，通过读取数据寄存器，来对数据寄存器进行复位。

下面是数据发送模块的实现。数据发送模块由两个函数组成，一个函数完成实际的数据发送功能（ComSendByte），另外一个函数是发送线程的入口函数。下面

是数据发送函数，该函数采用轮询的方式，向串口发送一个字节。实现代码如下：

```
static BOOL ComSendByte(UCHAR bt,WORD port)
{
    DWORD dwCount1 = 1024;
    DWORD dwCount2 = 3;

    while((dwCount2 --) > 0)
    {
        while((dwCount1 --) > 0)
        {
            if(__inb(port + 5) & 32) //Send register empty.
            {
                __outb(bt,port);
                return TRUE;
            }
        }
        dwCount1 = 1024;
        __MicroDelay(1024); //Delay 1s and try again.
    }
    return FALSE;
}
```

该函数的实现，与本文中 [基于轮询方式的数据发送](#) 一节描述的类似，采用两轮循环，判断数据发送保持寄存器是否为空（通过判断线路状态寄存器）。若为空，则发送数据并返回。若经过两轮循环的等待后，数据发送保持寄存器仍然不为空，则取消发送，返回FALSE。

组成发送模块的另外一个函数是 PollSend，该函数是发送核心线程的入口函数，在该函数中，调用了 ComSendByte 函数。代码如下：

```
static DWORD PollSend(LPVOID lpData)
{
    __BASE_EVENT* lpbe = (__BASE_EVENT*)lpData;
```

```

__KERNEL_THREAD_MESSAGE msg;
UCHAR bt;
DWORD count;
BOOL bSendResult = FALSE;

if(NULL == lpbe) //Invalid parameter.
{
    return 0L;
}
while(TRUE)
{
    if(GetMessage(&msg))
    {
        if(MSG_KEY_DOWN == msg.wCommand){
            bt = LOBYTE(LOWORD(msg.dwParam));
            if(QUIT_CHARACTER == bt) //Should quit.
            {

lpbe->lpEvent->SetEvent((__COMMON_OBJECT*)lpbe->lpEvent);
                return 0L;
            }
            bSendResult = FALSE;
            for(count = MAX_SEND_TIMES;count > 0;count --)
            {
                if(ComSendByte(bt,lpbe->wPortBase))
                {
                    bSendResult = TRUE;
                    break;
                }
            }
            if(!bSendResult) //Failed to send out.
            {
                PrintLine("Failed to send out,connection may
break.");
            }
        }
    }
}

```

```

        }
    }
}
return 0L;
}

```

该函数进入一个无限循环，并调用 GetMessage 函数，从当前核心线程的消息队列中获取消息（消息由系统输入对象发送到该线程的消息队列）。获取到消息后，对消息的类型进行判断，若是按键消息，则会根据用户按下的具体键，做不同的处理：

- 1、若用户按下的键是 QUIT\_CHARACTER（定义为 'z'），则设置一个事件对象。这个事件对象是另外一个线程—接收线程等待的事件对象。若该事件对象被设置，则等待线程会退出。设置完事件对象后，则返回，意味着发送线程退出运行；
- 2、若用户按下的键是其它形式的键，则会获取按键的 ASCII 码，然后通过串口发送出去。在发送的时候，调用了 SendComByte 函数。在发送的时候，做了 MAX\_SEND\_TIMES 次发送尝试。若所有发送尝试都不成功，则打印出 "Failed to send out, connection may break." 字符串，放弃此次发送。一般情况下，发送会立即完成的，只有在连接中断或串口芯片出现故障的时候，才可能出现发送失败的情形。

\_\_BASE\_EVENT 是临时定义的一个结构体，用于传递参数。因为按照 Hello China 目前的定义，一个核心线程入口函数，只能接受一个类型是 VOID 的指针作为参数，为了传递更多的参数，可通过定义结构体来实现。下面是 \_\_BASE\_EVENT 的定义：

```

typedef struct{
    WORD wPortBase;
    __EVENT* lpEvent;
}__BASE_EVENT;

```

其中，wPortBase 是串口的输入/输出端口号地址，lpEvent 是一个事件对象指针，用于完成发送核心线程和接收核心线程之间的同步操作。该事件对象由 hypertrm 的主入口函数创建，并作为参数传递给发送线程和接收线程。在用户输

入 `QUIT_CHARACTER` 键的时候，发送线程会设置该信号。而接收线程则不断检查该信号的状态，一旦该信号为有信号状态（被设置），则会结束运行。采用 `__BASE_EVENT` 结构传递参数，在发送模块中也会用到。

这样发送过程就很清晰了，总结如下：

- 1、 发送模块由两个函数：`ComSendByte` 和 `PollSend` 组成。其中 `ComSendByte` 完成轮询状态下的串口发送工作，`PollSend` 是发送核心线程的入口函数，该函数调用 `ComSendByte`，向串口发送用户输入的字符。同时，该线程还根据用户输入的字符，来判断是否应该结束运行；
- 2、 发送核心线程也不是一直在运行的，在没有用户输入的时候，发送核心线程阻塞在 `GetMessage` 函数上。只有在有用户输入（`GetMessage` 能够获取到消息）的时候，发送核心线程才被唤醒。因此，发送核心线程的效率是可以得到保证的。

下面是接收模块的实现。与发送模块类似，接收模块也是由两个函数组成的，`ComRecvByte` 用于从串口接收一个字节，而 `PollRecv` 则是接收核心线程的入口函数，该函数不断调用 `ComRecvByte`，若能够接收到数据，则打印在屏幕上。下面是 `ComRecvByte` 函数的实现代码：

```
static BOOL ComRecvByte(UCHAR* pbt,WORD port)
{
    DWORD nCount1 = 1024;
    DWORD nCount2 = 3;

    while(nCount2 -- > 0)
    {
        while(nCount1 -- > 0)
        {
            if(__inb(port + 5) & 1) //Data available.
            {
                *pbt = __inb(port);
                return TRUE;
            }
        }
    }
}
```

```

        nCount1 = 1024;
        __MicroDelay(1024); //Delay 1s and try again.
    }
    return FALSE;
}

```

该函数也是采用双层循环的方式，不断检查线路寄存器的状态。一旦发现有字符到达，则调用\_\_inb，把字符从串口中读取出来，并返回 TRUE。若两层循环后仍然没有取得数据，则放弃接收操作，返回 FALSE，指示本次接收失败。

下面是接收核心线程的入口函数：

```

static DWORD PollRecv(LPVOID lpData)
{
    __BASE_EVENT* lpbe = (__BASE_EVENT*)lpData;
    UCHAR bt;
    DWORD count;
    WORD wr = 0x0700;

    while(TRUE)
    {
        for(count = MAX_RECV_TIMES; count > 0; count --)
        {
            if(ComRecvByte(&bt, lpbe->wPortBase))
            {
                switch(bt)
                {
                    {
                        case '\r':
                            ChangeLine();
                            break;
                        case '\n':
                            GotoHome();
                            break;
                        default:
                            wr += bt;

```

```

        PrintCh(wr);
        wr = 0x0700; //Reset the background color.
        break;
    }
}
}
if(OBJECT_WAIT_RESOURCE ==
    lpbe->lpEvent->WaitForThisObjectEx(
        (__COMMON_OBJECT*)lpbe->lpEvent,0L))    //Should
terminate.
{
    return 0L;
}
}
return 0L;
}

```

该函数进入一个无限循环，不断调用 ComRecvByte 函数，试图从串口接收字符。若调用成功，则打印出此字符。在每个循环结束的时候，调用 WaitForThisObjectEx 函数，检查事件对象的状态。若事件状态被设置，则函数返回，导致接收线程结束。有两个地方需要解释一下：

- 1、 WaitForThisObjectEx 是一个超时等待函数，第二个参数给出了超时值（以毫秒计）。若在超时前，等待的对象可用（比如，事件对象被设置），则返回 OBJECT\_WAIT\_RESOURCE，若超过等待事件后对象仍不可用，则返回 OBJECT\_WAIT\_TIMEOUT。若以参数 0 调用该函数，则不会进入阻塞操作，而只是判断一下等待对象的状态。若状态可用，则直接返回 OBJECT\_WAIT\_RESOURCE，若对象不可用，则直接返回 OBJECT\_WAIT\_TIMEOUT。在该实现中，无需等待事件对象，只需判断一下对象的状态即可。只要事件对象被设置，则意味着用户按下了 QUIT\_CHARACTER 键，接收线程会直接结束运行；
- 2、 获取到串口的字符后，需要进一步判断字符是否为回车或换行符。若是，则调用 ChangeLine 和 GotoHome，换行或回车（回到一行的起始处）。若是其它字符，则直接调用 PrintCh 打印出来。PrintCh 是一个 PC 屏幕输出函数，接收一个 WORD（两字节）类型的参数，其中参数的高字节指明了输出到屏幕上的前景和背景颜色，而低字节则指明了要输出的字符的 ASCII 码。

下面是 hypertrm 应用程序的主入口函数，也是主入口线程。hypertrm 是作为 Hello China 的外部命令实现的，Hello China 在执行外部命令的时候，都是以外部命令提供的入口函数为主函数，创建一个核心线程。因此，下列函数也应该遵循 Hello China 定义的核心线程入口函数原型（以 LPVOID 为参数，返回 DWORD）。

hypertrm 的主入口函数实现了下列功能：

- 1、初始化 COM 接口。在当前的实现中，hypertrm 直接操作 COM1 接口。也可通过传入命令行的方式，根据用户输入选择不同的 COM 接口，但目前为了实现上的方便，采用固定编码的方式，直接操作 COM1 接口。这可适应大多数的应用场合；
- 2、初始化接收线程和发送线程用到的内核对象，比如事件对象等，然后创建接收核心线程和发送核心线程；
- 3、等待两个核心线程运行结束，然后完成清理工作，并返回。

下面是其实现代码：

```
DWORD Hypertrm(LPVOID lpData)
{
    __BASE_EVENT be;
    __KERNEL_THREAD_OBJECT* lpSendThread = NULL;
    __KERNEL_THREAD_OBJECT* lpRecvThread = NULL;

    //Print application information.
    PrintLine(" ----- Hypertrm for Hello China is running
----- ");
    ChangeLine();
    GotoHome();

    be.lpEvent
=
    (__EVENT*)ObjectManager.CreateObject(&ObjectManager,
        NULL,
        OBJECT_TYPE_EVENT); //Create event object.
    if(NULL == be.lpEvent) //Can not create object.
    {
        PrintLine("Can not create event object.");
        goto __TERMINAL;
```

```

    }
    if(!be.lpEvent->Initialize((__COMMON_OBJECT*)be.lpEvent)
)
    {
        PrintLine("Can not initialize the event object.");
        goto __TERMINAL;
    }
    be.lpEvent->ResetEvent((__COMMON_OBJECT*)be.lpEvent);
    be.wPortBase = COM1_BASE; //Use COM1 as default port.

    //Initialize the COM port.
    InitComPort(be.wPortBase);

    //Now create the receive and send kernel thread.
    lpSendThread =
    (__KERNEL_THREAD_OBJECT*)KernelThreadManager.CreateKernelThr
    ead(
        (__COMMON_OBJECT*)&KernelThreadManager,
        0L,
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_NORMAL,
        PollSend,
        (LPVOID)&be,
        NULL,
        "COMSEND");
    if(NULL == lpSendThread)
    {
        PrintLine("Can not create send kernel thread.");
        goto __TERMINAL;
    }

    lpRecvThread =
    (__KERNEL_THREAD_OBJECT*)KernelThreadManager.CreateKernelThr
    ead(

```

```

        (__COMMON_OBJECT*)&KernelThreadManager,
        0L,
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_NORMAL,
        PollRecv,
        (LPVOID)&be,
        NULL,
        "COMRECV");
if(NULL == lpRecvThread) //Can not create receive thread.
{
    PrintLine("Can not create receive kernel thread.");
    goto __TERMINAL;
}

//Give the current focus to send thread.
DeviceInputManager.SetFocusThread((__COMMON_OBJECT*)&DeviceInputManager,
    (__COMMON_OBJECT*)lpSendThread);

//Now wait the two kernel threads to finish.
WaitForThisObject((HANDLE)lpSendThread);
WaitForThisObject((HANDLE)lpRecvThread);

__TERMINAL:
if(NULL != be.lpEvent)
{
    DestroyEvent((HANDLE)be.lpEvent);
}
if(NULL != lpSendThread)
{
    DestroyKernelThread((HANDLE)lpSendThread);
}
if(NULL != lpRecvThread)
{

```

```

        DestroyKernelThread((HANDLE)lpRecvThread);
    }
    return 0L;
}

```

需要注意的是，在完成发送线程和接收线程的创建之后，调用了 `SetFocusThread`，设置了当前输入焦点线程为发送核心线程。这样的结果是，用户的任何输入（目前来说，只有键盘输入），操作系统核心都会发送给发送线程。这样发送线程就可通过串口发送出去。若不调用该函数，则当前的输入焦点线程是系统 `shell` 线程，用户的键盘输入，会被 `shell` 线程获得，而不会被 `COM` 接口发送线程获得，因此无法实现交互。

上述代码组形成了整个 `hypertrm` 应用程序。最后，需要在 `EXTCMD.CPP` 文件中，增加一行代码，把 `hypertrm` 的命令字符串和入口函数添加到外部命令列表。这样一旦用户输入“`hypertrm`”字符串，`shell` 线程就会查找内部和外部命令列表，最终会匹配到刚刚添加的项，于是 `shell` 会以 `hypertrm` 为入口函数，创建一个核心线程，这样最终会导致 `hypertrm` 应用程序被启动起来。

下面总结一下 `hypertrm` 的启动和运行过程：

- 1、用户在命令行界面下，输入 `hypertrm` 字符串，然后回车；
- 2、`shell` 线程获取输入，以输入的字符串为关键字，搜索内部命令列表和外部命令列表；
- 3、在外部命令列表搜索中，命中刚才添加的项；
- 4、`shell` 以 `Hypertrm` 为入口点，创建一个核心线程，并把当前输入焦点（通过调用 `SetFocusThread` 函数）设置为刚刚创建的核心线程；
- 5、`Hypertrm` 作为 `hypertrm` 应用的主入口函数，初始化 `COM1` 接口，并创建发送核心线程和接收核心线程，然后重新设置当前输入焦点为发送核心线程，并等待发送线程和接收线程运行结束（等待的过程中，主线程处于阻塞状态）；
- 6、发送核心线程调用 `GetMessage` 函数，检查消息队列，把用户通过键盘输入的字符，发送到 `COM` 接口。需要注意的是，发送核心线程是阻塞在 `GetMessage` 函数上的，只有消息到达的时候，才会被唤醒；
- 7、接收线程不断的检查 `COM` 接口的线路状态寄存器，若发现 `COM` 接口有字符到达，则读取并打印到屏幕上。另，在每个检查循环结束的时候，接收线程还检查一个事件对象（该事件对象用于同步发送和接收线程），一旦发现事件对象被设置，则退出运行；
- 8、一旦用户输入 `QUIT_CHARACTER`（当前定义为‘`Z`’），发送核心线程将设置事件对象（这会导致接收线程也退出），并退出运行；
- 9、当发送核心线程和接收核心线程都退出运行的时候，`hypertrm` 的主线程

会恢复运行，这时候，主线程做一些收尾工作，释放相应的资源，并退出运行。

## 中断模式的串口交互程序实现

采用轮询方式的 `hypertrm` 程序，其发送线程是输入驱动的，即只有用户有键盘输入的时候，才会被唤醒，若没有输入，则会阻塞在消息队列上，不会空循环而导致 CPU 资源浪费。但接收线程却是一直在运行的，即使 COM 接口没有任何数据到达，接收线程也不会阻塞，而是一直处于检查 COM 接口的状态之中。显然，接收线程会大大浪费 CPU 资源。这种情形无法避免，这也是轮询方式驱动程序（或应用程序）的最大缺点。

采用中断方式可解决该问题。对于发送线程，会由键盘输入事件驱动，与轮询方式一致，不会浪费任何 CPU 资源。但对于接收方式，也会由中断驱动。在 COM 接口有数据到达的时候，COM 接口芯片会引发中断，在中断处理程序中，会唤醒接收线程。这样就可避免了轮询方式中 CPU 资源浪费的现象。

但中断方式的数据接收线程，相对轮询方式，其编程方式也会复杂得多。在下面的部分中，我们会对中断方式的接收线程实现，进行详细描述。

需要注意的是，不论对于接收还是发送，都可由中断驱动。对于数据接收，中断驱动是很自然的事情，因为无法预期什么时候会有数据到达。但对于发送，却是可预期的，因为发送是主动的（用户输入的时候会引发发送过程）。但有的情况下，也需要中断来配合。因为在发送的时候，需要 COM 接口的发送保持寄存器处于空状态。这样若发送频率太高，超出了 COM 接口的处理能力，若不查询 COM 接口的状态而直接发送，会导致信息丢失。这样就需要中断来配合了，在发送大量数据的时候，应用程序可先启动发送一个字符，然后进入等待状态。在 COM 接口芯片完成发送后，会通过中断通知发送线程，其发送保持寄存器为空，这时候发送线程可启动后续字符的发送，一直持续到数据发送完毕，这样就不会导致信息发送丢失了。

但我们实现的 `hyptm2` 程序，却不会出现发送大量数据的情形，每次只会发送一个字符。而且唯一的发送来源，就是键盘输入。即使输入得再快，也不可能超过 COM 接口的发送能力和计算机的处理能力。因此对于发送过程，我们仍然采用跟轮询方式一致的处理方式。但对于接收过程，采用中断处理方式。

`hyptm2` 的代码结构与 `hypertrm` 结构类似，不同的是增加了一个中断处理程序，用于处理中断。但在一些实现细节和采用的数据结构上，则有较大不同。`hypertrm` 没有采用任何复杂的数据结构，只是简单的把接收到或待发送的数据存

储到一个本地变量中，然后直接处理。但对于中断方式的处理程序，却需要涉及到中断处理程序和接收线程的数据交互和同步，我们采用了环形缓冲区（ring buffer）内核对象。该对象的实现机制，请参考“优先队列和环形缓冲区”一章。

下面是中断方式下串口的初始化函数：

```
static void InitComPort2(WORD base)
{
    if((base != COM1_BASE) && (base != COM2_BASE))
    {
        return;
    }
    __outb(0x80,base + 3); //Set DLAB bit to 1,thus the baud rate
divisor can be set.
    __outb(0x0C,base); //Set low byte of baud rate divisor.
    __outb(0x0,base + 1); //Set high byte of baud rate divisor.
    __outb(0x07,base + 3); //Reset DLAB bit,and set data bit to
8,one stop bit,without parity check.
    __outb(0x01,base + 1); //Enable data available interrupt.
    __outb(0x0B,base + 4); //Enable DTR,RTS and Interrupt.
    __inb(base); //Reset data register.
}
```

与轮询方式不同的是黑体标出的部分。在黑体标出的第一行代码中，启用了数据可用中断（设置中断允许寄存器的第一个比特），所有其它中断，包括发送保持寄存器空中断、发送状态错误中断等，都是禁止的。这样可简化程序的实现，而且功能上也不会有什么影响。若需要补充其它类型的中断，只需要设置相应的比特，并在中断处理程序中添加处理代码即可。

黑体标出的第二行允许 COM 接口芯片引发中断，同时设置了 DTR 和 RTS 标记。黑体代码第一行的作用，仅仅是允许哪些事件可引发中断，若没有第二行黑体代码，则 COM 接口芯片仍然不会引发 CPU 中断，即有可引发中断的事件发生。这是由 PC/XT 设计结构导致的，感觉有些多余☺。

设置好上述寄存器之后，若一旦有数据到达串口，串口就会引发 CPU 中断了。

下面是中断模式下的发送线程。hyptrm2 的实现中，对于发送过程，只采用了一个函数实现，这个函数也是发送核心线程的入口函数。代码如下：

```
static DWORD IntSend(LPVOID lpData)
{
    __BASE_AND_EVENT2* pbe2 = (__BASE_AND_EVENT2*)lpData;
    __KERNEL_THREAD_MESSAGE msg;
    UCHAR bt;
    DWORD count;
    BOOL bSendResult = FALSE;

    if(NULL == pbe2) //Invalid parameter.
    {
        return 0L;
    }
    while(TRUE)
    {
        if(GetMessage(&msg))
        {
            if(MSG_KEY_DOWN == msg.wCommand) //Key press event.
            {
                bt = LOBYTE(LOWORD(msg.dwParam));
                if(QUIT_CHARACTER == bt) //Should quit.
                {
                    SetEvent(pbe2->hTerminateEvent);
                    //Indicate the receiving thread
                    //to exit.

                    return 0L;
                }
                //bSendResult = FALSE;
                __outb(bt,pbe2->wBasePort);
            }
        }
    }
}
```

```
return 0L;
}
```

该函数比较简单，在一个无限循环当中检查消息队列，若有键盘输入消息，则会被唤醒进行处理。若用户按下的键是 `QUIT_CHARACTER`，则设置事件对象，以指示接收线程退出，然后返回，这样就导致自己结束运行。若接收到的字符是其它字符，则调用 `__outb` 函数，输送到串口。这里采用了一种最简单的形式，没有判断 COM 接口线路寄存器的状态。这实际上有些冒险，因为有可能 COM 接口尚未准备好发送。但在绝大多数情况下，都是可以正常工作的。对于要求十分苛刻的场合，可增加这部分判断，并增加多次尝试（与轮询方式发送类似）。

需要注意的是，若用户不做任何键盘输入，则该发送线程会处于阻塞状态，不会消耗任何 CPU 资源。这跟轮询方式下的发送过程类似。

上述代码中 `__BASE_AND_EVENT2` 是临时定义的一个数据结构，用于完成线程之间的参数传递，定义如下：

```
typedef struct{
    WORD wBasePort;          //The COM port's base address(port).
    HANDLE hTerminateEvent;  //The event object to indicate
    terminate.
    HANDLE hSendRb;          //Sending ring buffer.
    HANDLE hRecvRb;          //Receiving ring buffer.
}__BASE_AND_EVENT2;
```

其中，`wBasePort` 是 COM 接口的端口基地址，`hTerminateEvent` 是一个事件对象，用于同步发送线程和接收线程。该事件对象由发送线程设置，用于通知接收线程结束运行。`hSendRb` 和 `hRecvRb` 是两个环形缓冲区（`__RING_BUFFER`）对象，用于完成中断处理程序和发送/接收核心线程的数据交换和同步。在当前的实现中，数据发送线程没有用到环形缓冲区对象，因为没有采用中断方式。而数据接收核心线程却需要使用环形缓冲区对象缓存数据。

发送模块也只有一个函数，该函数也是发送核心线程的主入口函数。实现代码如下：

```

static DWORD IntRecv(LPVOID lpData) //Receive thread
routine in interrupt mode.
{
    __BASE_AND_EVENT2* pbe2 = (__BASE_AND_EVENT2*)lpData;
    WORD wr = 0x0700;
    DWORD element;

    while(TRUE)
    {

if(GetRingBuffElement(pbe2->hRecvRb,&element,MAX_RECV_WAIT))
    {
        switch((UCHAR)element)
        {
            case '\r':
                ChangeLine();
                break;
            case '\n':
                GotoHome();
                break;
            default:
                wr += (UCHAR)element;
                PrintCh(wr);
                wr = 0x0700; //Reset the background color.
                break;
        }
    }

    if(OBJECT_WAIT_RESOURCE ==
WaitForThisObjectEx(pbe2->hTerminateEvent,
    0L)) //Should terminate.
    {
        PrintLine("Receiving kernel thread exit now.");
        return 0L;
    }
}

```

```

    }
}
return 0L;
}

```

该函数进入一个无限循环，调用 `GetRingBuffElement`，试图从环形缓冲区中取得数据。环形缓冲区是由 `hypترم2` 的主入口线程创建的，中断处理程序会向该环形缓冲区中放置数据。若环形缓冲区中有数据，则接收线程会被唤醒，根据获取的数据，做不同的处理：

- 1、 若接收的字符是一个换行符，则调用 `ChangeLine` 函数，移动当前光标到下一行；
- 2、 若接收到的字符是一个回车符，则调用 `GotoHome` 函数，把光标返回到当前行的起始位置；
- 3、 若是其它字符，则调用 `PrintCh` 打印该字符到屏幕上。

完成一轮检查之后，再调用 `WaitForThisObjectEx` 函数，检查事件对象是否被设置。若是，则直接返回，从而导致接收线程退出，否则进入下一轮循环。

需要注意的是，若环形队列中没有任何数据处理，则该线程会阻塞到环形队列上，不会浪费任何 CPU 资源。这是中断方式最大的优点。

与轮询方式实现的 `hyperترم` 最大的不同，就是在 `hypترم2` 的实现中，多增加了一个中断处理程序。该中断处理程序处理 COM 接口芯片引发的中断。下面是中断处理程序的实现代码：

```

static BOOL ComIntHandler(LPVOID,LPVOID lpParam)
{
    __BASE_AND_EVENT2* pbe2 = (__BASE_AND_EVENT2*)lpParam;

    if(NULL == pbe2)
    {
        BUG();
        return FALSE;
    }
}

```

```

    UCHAR isr = __inb(pbe2->wBasePort + 2); //Read interrupt
status register.
    UCHAR bt;
    if(isr & 1) //No interrupt to process.
    {
        return FALSE;
    }
    if(__inb(pbe2->wBasePort + 5) & 1) //Data available.
    {
        bt = __inb(pbe2->wBasePort); //Read the byte.
        AddRingBuffElement(pbe2->hRecvRb,(DWORD)bt); //Add to
ring buffer.
    }
    return TRUE;
}

```

中断处理程序非常简单，首先通过读取中断状态寄存器，判断是否有中断发生。若中断状态寄存器的第一个比特为 0，则说明有中断发生，若为 1，则说明无中断发生，直接返回 `FALSE`。

在有中断待处理的情况下，进一步通过读取线路状态寄存器，判断数据寄存器是否有数据待读取。若是，则调用 `__inb` 函数，从 `COM` 的数据寄存器中读取一个字节的数 据，然后调用 `AddRingBuffElement`，添加到环形队列中。`AddRingBuffElement` 函数会环形阻塞在该环形队列上的核心线程。我们知道，接收线程就是通过调用 `GetRingBuffElement` 阻塞在环形队列上的，在中断程序中，接收线程会被唤醒。

有两个地方需要做进一步解释：

- 1、中断程序的返回值。在没有中断要处理的情况下，会直接返回 `FALSE`。这样中断调度程序会认为该中断不是当前中断处理程序对应的设备发出的，于是会继续调用其它的中断处理程序（这些中断处理程序对应的设备，跟 `COM` 接口共享同一中断输入）。若返回 `TRUE`，则标明当前中断处理程序已成功的处理了中断，于是中断调度程序不会再调用其它设备的中断处理程序了；
- 2、只所以会出现中断程序被调用，但中断状态寄存器却标明没有中断发生（最低比特为 1）的情况，是因为多种设备可共享同一条中断输入。比如，另外一个计算机外设跟 `COM` 接口芯片连接到了同一条中断输入

引脚上(8259 芯片的相同引脚), 则另外的设备引发中断的时候, COM 接口的中断处理程序也可能被调用。因此, 需要进一步判断中断是否 COM 接口芯片引发的, 若是, 就做进一步处理并返回 TRUE, 否则返回 FALSE, 以便另外设备的中断处理程序会被调用。

下面是 hyptrm2 应用程序的主入口函数。hyptrm2 作为 Hello China 的外部命令实现, 其主入口函数也需要符合核心线程入口函数的原型定义。下面是其实现代码:

```
//Main entry for HYPTRM2 application.
DWORD Hyptrm2(LPVOID lpData)
{
    __BASE_AND_EVENT2 be21;
    __BASE_AND_EVENT2 besend;
    __BASE_AND_EVENT2 berecv;
    HANDLE hSendThread = NULL;
    HANDLE hRecvThread = NULL;
    HANDLE hTerminateEvent = NULL;
    HANDLE hIntHandler = NULL;
    HANDLE hSendRb = NULL;
    HANDLE hRecvRb = NULL;

    //Print out application title information.
    PrintLine(" ----- Hyptrm2 for Hello China is running
----- ");
    ChangeLine();
    GotoHome();

    hTerminateEvent = CreateEvent(FALSE);
    if(NULL == hTerminateEvent)
    {
        PrintLine("Can not create hTerminateEvent.");
        goto __TERMINAL;
    }
}
```

```

hSendRb = CreateRingBuff(0);
if(NULL == hSendRb)
{
    PrintLine("Can not create sending ring buffer.");
    goto __TERMINAL;
}
hRecvRb = CreateRingBuff(0);
if(NULL == hRecvRb)
{
    PrintLine("Can not create receiving ring buffer.");
    goto __TERMINAL;
}

be21.hSendRb      = hSendRb;
be21.hRecvRb      = hRecvRb;
be21.wBasePort    = COM1_BASE;

//Connect interrupt handler.
hIntHandler =
ConnectInterrupt(ComIntHandler, (LPVOID)&be21, COM1_INT_VECTOR
);
if(NULL == hIntHandler)
{
    PrintLine("Can not set COM's interrupt handler.");
    goto __TERMINAL;
}

//Initialize the COM interface.
InitComPort2(COM1_BASE);

//Create sending kernel thread now.
besend.wBasePort = COM1_BASE;
besend.hTerminateEvent = hTerminateEvent;

```

```

besend.hSendRb    = hSendRb;
hSendThread = CreateKernelThread(
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_NORMAL,
    IntSend,
    (LPVOID)&besend,
    NULL,
    "COMSEND_INT");
if(NULL == hSendThread) //Failed to create sending kernel
thread.
{
    PrintLine("Can not create sending thread.");
    goto __TERMINAL;
}

//Create receiving kernel thread.
berecv.hTerminateEvent = hTerminateEvent;
berecv.wBasePort        = COM1_BASE;
berecv.hRecvRb          = hRecvRb;
hRecvThread = CreateKernelThread(
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_NORMAL,
    IntRecv,
    (LPVOID)&berecv,
    NULL,
    "COMRECV_INT");
if(NULL == hRecvThread) //Can not create receiving thread.
{
    PrintLine("Can not create receiving kernel thread.");
    goto __TERMINAL;
}

```

```

        //Set sending kernel thread as current focus thread.
        DeviceInputManager.SetFocusThread((__COMMON_OBJECT*)&DeviceInputManager,
            (__COMMON_OBJECT*)hSendThread);

        //Wait for receiving and sending kernel threads to terminate.
        WaitForThisObject(hSendThread);
        WaitForThisObject(hRecvThread);

__TERMINAL:
    if(NULL != hIntHandler)
    {
        DisconnectInterrupt(hIntHandler);    //Disconnect the
        interrupt handler.
    }
    if(NULL != hTerminateEvent)
    {
        DestroyEvent(hTerminateEvent);
    }

    if(NULL != hSendThread)
    {
        DestroyKernelThread(hSendThread);
    }
    if(NULL != hRecvThread)
    {
        DestroyKernelThread(hRecvThread);
    }

    if(NULL != hRecvRb)
    {
        DestroyRingBuff(hRecvRb);
    }
    if(NULL != hSendRb)

```

```

{
    DestroyRingBuff(hSendRb);
}
return 0L;
}

```

该函数比较长，但却比较简单，主要有三部分组成：

- 1、 初始化部分代码。在该部分代码中，创建了用于同步接收核心线程和发送核心线程的事件对象，以及用于中断处理程序和发送/接收线程的环形缓冲区对象。然后调用 `ConnectInterrupt` 函数，把 COM 接口的中断处理程序安装到系统中。完成这些之后，才调用 `InitComPort2`，初始化了串口芯片。需要注意的是，一定要在中断处理程序安装完成之后，才能初始化 COM 接口。因为在初始化 COM 接口的时候，其中断是被打开的，这时候若还没有安装中断处理程序，一旦 COM 接口引发一个中断，则会导致系统打印出系统诊断信息而停机；
- 2、 核心线程创建代码。创建了接收和发送核心线程，并调用 `WaitForThisObject`，等待两个核心线程运行结束。在接收和发送线程运行过程中，主线程是被阻塞的，不作任何处理；
- 3、 运行结束后的清理代码。这部分代码释放了创建的事件对象和核心线程对象，调用 `DisconnectInterrupt` 函数取消了安装在系统中的中断处理程序，并退出运行。需要注意的是，一定要调用 `DisconnectInterrupt` 函数取消中断，否则中断处理程序还可能被调用。这时候由于环形缓冲区对象已被销毁，可能会导致内存紊乱。

## 串行通信编程总结

### 轮询方式和中断方式编程的对比

通过上述描述可知，轮询方式的设备驱动程序，比中断方式的设备驱动程序消耗更多的 CPU 资源，因为轮询方式的驱动程序，需要 CPU 不停的去查询设备的状态，并做出适当的处理。而中断方式则不然，设备驱动线程只需被动的等待即可，一旦设备有输入，则会引发中断，在中断处理程序中完成设备 IO，并唤醒等待的线程。在普通的操作系统环境中，比如个人计算机的操作系统实现中，选择中断方式的设备驱动程序是合适的，因为这可大大节约系统整体资源。

但在嵌入式操作系统中，选择中断方式的设备驱动实现，可能会存在问题。因为嵌入式系统对系统的响应时间要求十分苛刻。若采用中断方式的驱动程序，则正在处理关

键任务的核心线程可能会被设备的中断打断，从而延误关键的事件处理。更糟糕的是，若系统外设硬件故障，导致外设不断的引发中断，这样可能会使系统一直忙于处理不重要的外部中断，无法响应其它的系统事件。

轮询方式的设备驱动方式可避免此类问题。这时候，对设备的处理代码，实际上是一个核心线程，合适设置该核心线程的优先级，可使得系统中所有核心线程都处于一种有序的、可预测的调度顺序，这样即使外设不断发生中断，也不会对系统中其它关键的线程造成影响，因为关键的线程会优先得到调度。

因此，在设备驱动程序的实现中，可采用轮询方式加中断方式结合的策略：

- 1、对于非常重要的且认为可靠的外部设备，可采用中断方式实现其驱动程序。这样可以提高系统的整体性能；
- 2、对于非关键的、不可靠的外部设备，可采用轮询方式实现其驱动程序。这样可以提高系统的整体鲁棒性。

Hello China 的线程模型和中断调度模型，可满足上述实现的需求。

### 串口交互程序的其它实现方式

在 `hypertrm` 和 `hyptrm2` 的实现中，是直接采用核心线程的方式进行的，没有采用 Hello China 的驱动程序框架。虽然实现上的思路没有什么本质的不同，但采用 Hello China 的驱动程序框架来实现串口交互程序，会使得软件代码的整体结构和可复用性更高。若采用 Hello China 的驱动程序框架模型来实现串口交互程序，可参考下列步骤：

- 1、为串口编写一个驱动程序，该驱动程序需要符合 Hello China 定义的驱动程序接口规范；
- 2、在驱动程序里，实现对硬件设备（串口）的读写和控制工作；
- 3、在系统中注册该驱动程序，可通过向系统驱动程序注册数组中添加对应的项来完成。详细信息，请参考“Hello China 的设备驱动模型”一章；
- 4、编写具体的实现线程。该实现线程也是串口交互程序的主入口线程，该线程通过 Hello China 提供的 `ReadFile`、`WriteFile` 等 API 函数，完成 COM 接口数据的输入/输出。在调用上述 API 函数前，先通过调用 `CreateFile` 函数，打开 COM 接口。当然，可采用不同的核心线程，分别处理输入和输出；
- 5、结束后，调用 `CloseFile` 函数，关闭 COM 接口文件对象。

显然，这种方式实现了设备驱动代码和应用程序代码的隔离。设备驱动代码采用单独的设备驱动程序实现，应用程序代码采用统一的接口，访问设备驱动程序。这样的实现，可使得这两部分代码相互独立，对设备驱动程序的修改，不会影响到应用程序的代码，多个应用程序，也可访问相同的设备驱动代码。



# 附录 B 核心线程 CPU 占用率统计功能

## CPU 占用率概述

在操作系统的设计和实现中，对每个线程或进程的 CPU 占用率进行统计，是一项非常重要且清晰的工作。这样可使最终用户很清晰的看到每个线程或进程的 CPU 占用情况，对于故障定位、程序优化非常重要。

一般情况下，对 CPU 占用率的统计，是在内核层面进行的，因为只有内核，才清楚什么时刻调度了什么进程（或线程），每个线程或进程的运行时间，等等。线程或进程本身是无法统计的，因为根本不知道自己什么时候在运行，也不知道究竟实际运行了多长时间。因为线程或进程随时可能被抢占。

即使是采用非抢占方式调度的操作系统中，线程或进程本身也很难统计准确的 CPU 占用率。虽然线程或进程知道被调度出 CPU 或重新调度的时刻（一般是在系统调用的前后发生），但仍然无法预料中断的发生，而且中断的发生，对线程或进程来说是透明的。因此，在非抢占方式调度的操作系统中，线程或进程仍然难以统计自己的准确执行时间。但若把中断或异常执行时间也考虑进当前线程或进程，则可采用下列方式，大致计算自己的执行时间：

- 1、 在线程或进程开始执行的时候，记录下当前时间（精确起见，可记录 CPU 的内部计数器）；
- 2、 做任何系统调用前，再次记录当前时间（CPU 计数器），然后取本次记录的时间值跟前一次记录的差值，该差值就是线程或进程在本次系统调用前所运行的时间（或占用的 CPU 节拍）；
- 3、 系统调用完后，开始正常的功能操作前，重新记录当前时间。

这样把上述所有记录的时间片断累加起来，就是当前线程或进程所占用的 CPU 时间了（其中包含了发生在当前线程或进程中的中断处理时间）。这样可粗略估计当前线程或进行的执行时间。需要注意的是，上述统计方式，没有考虑系统调用的执行时间。

Hello China 目前采用的是抢占调度方式，因此线程或进程本身无法自行统计执行时间。但 Hello China 的内核提供了该项功能。其实现方式，跟上面描述的思路类似，就是在线程开始执行的时候（或被重新调度执行的时候），记录一次 CPU 的当前计数器（记录了 CPU 自初始化以来到目前的节拍数量），在线程被换出 CPU 的时候，再记录一次 CPU 的计数器，然后两者相减，就得到了当前线程最近一次被调度，所占用的 CPU 时钟数。该数值再除以 CPU 的时钟频率，就可得到当前核心线程的执行时间。但更多的情况下，用户或开发人员关注的是线程或进程的 CPU 占用率（即线程或进程的执行时间，占 CPU 总运行时间的百分比），因此只要把每个线程占用的 CPU 时钟节拍数记录下来就

可以了，通过比值就可计算出每个线程的 CPU 占用率，无需换算成真正的时间，因为这种换算，可能会涉及浮点运算，而目前版本的 Hello China，还没有对 CPU 的 FPU（浮点处理单元）进行初始化，尚不支持浮点运算。

在本章中，我们对 Hello China 实现的核心线程 CPU 占用率统计功能进行描述。

## 核心线程 CPU 占用率统计的实现

### 统计周期和统计算法

统计周期指的是对核心线程 CPU 占用率进行计算的时间间隔。虽然从理论上来说，核心线程的 CPU 占用率是任何时刻都可以统计的，但实际上在实现的时候，为了提高效率，一般选择每隔一段时间，进行一次统计，这一段时间就是统计周期。

假设统计周期为  $T$ ，系统中有  $N$  个核心线程，在时间间隔（周期） $T$  内，统计的每个核心线程的 CPU 占用时间分别为  $(T_1, T_2, T_3, \dots, T_n)$ ，则第  $i$  个核心线程的 CPU 占用率  $R_i$  应该为：

$$R_i = T_i / T \times 100\%$$

假设针对一个核心线程  $i$ ，连续统计了  $M$  个统计周期（统计周期为  $T$ ），这个  $M$  个周期中，该线程运行的时间分别为  $(T_{i1}, T_{i2}, T_{i3}, \dots, T_{im})$ ，对应的 CPU 占用率分别为  $(R_{i1}, R_{i2}, R_{i3}, \dots, R_{im})$ 。对于该核心线程，第  $j$  个统计周期内的 CPU 占用率  $R_{ij}$  应该为：

$$R_{ij} = T_{ij} / T \times 100\%$$

对  $M$  个统计周期内，该核心线程的所有 CPU 占用率进行累加：

$$\begin{aligned} & R_{i1} + R_{i2} + R_{i3} + \dots + R_{im} \\ &= T_{i1}/T + T_{i2}/T + T_{i3}/T + \dots + T_{im}/T \\ &= (T_{i1} + T_{i2} + T_{i3} + \dots + T_{im}) / T \end{aligned}$$

对上述结果两边同时除以  $M$ ，则得到：

$$\begin{aligned} & (\mathbf{Ri1} + \mathbf{Ri2} + \mathbf{Ri3} + \dots + \mathbf{Rim}) / \mathbf{M} \\ & = (\mathbf{Ti1} + \mathbf{Ti2} + \mathbf{Ti3} + \dots + \mathbf{Tim}) / (\mathbf{M} * \mathbf{T}) \end{aligned}$$

其中，等式的右边，刚好是在间隔为  $M$  个统计周期内，该核心线程的 CPU 占用率 ( $M$  个统计周期的运行时间，除以  $M$  个统计周期)。而等式的左边，则是所有  $M$  个统计周期中，该线程在所有统计周期内的 CPU 占用率的平均值。

上述结果非常重要。根据上述推导结果，只要我们保存了连续  $M$  个统计周期内单核心线程的 CPU 占用率，则任意小于  $T \times M$  ( $N$  为统计周期) 时间内的核心线程 CPU 占用率，都可以通过上述公式计算得出。

在 Hello China 核心线程 CPU 占用率统计的实现中，定义的统计周期为 1000ms(1s)，并针对每个核心线程，定义了一个包含 300 个元素的数组，该数组用于存储连续 300 个统计周期内，该核心线程的 CPU 占用率。这样最后 5 分钟 ( $1000\text{ms} \times 300 = 5$  分钟) 内，任意时间内的核心线程 CPU 占用率都可通过计算得到。但一般情况下，关注最后 1s 内的 CPU 占用率、最后 1 分钟内的 CPU 占用率和最后 5 分钟内的 CPU 占用率三个数值就足够了，因此在目前版本的 Hello China 实现中，只针对上述三个典型时刻输出了统计值。

## 核心线程统计对象

核心线程统计对象用于记录单个线程的 CPU 占用率相关信息，一个核心线程统计对象，跟一个核心线程对应，定义如下：

```
struct __THREAD_STAT_OBJECT{
    __THREAD_STAT_OBJECT*    lpPrev;
    __THREAD_STAT_OBJECT*    lpNext;

    __U64                      TotalCpuCycle;
    __U64                      CurrPeriodCycle;
    __U64                      PreviousTsc;

    __KERNEL_THREAD_OBJECT*    lpKernelThread;

    WORD                       wQueueHdr;
    WORD                       wQueueTail;
```

```
WORD                                RatioQueue[MAX_STAT_PERIOD];
WORD                                wOneMinuteRatio; //CPU ratio in
last 1 minute.
WORD                                wMaxStatRatio;    //CPU ratio in
last maximal statistics
//period,such as 5
minutes.
WORD                                wCurrPeriodRatio;
WORD                                wReserved;
};
```

其中，lpNext 和 lpPrev 两个指针变量，把系统中的所有核心线程统计对象连接起来，形成一个链表。该对象组成的链表，由 CPU 统计对象管理，如下图：

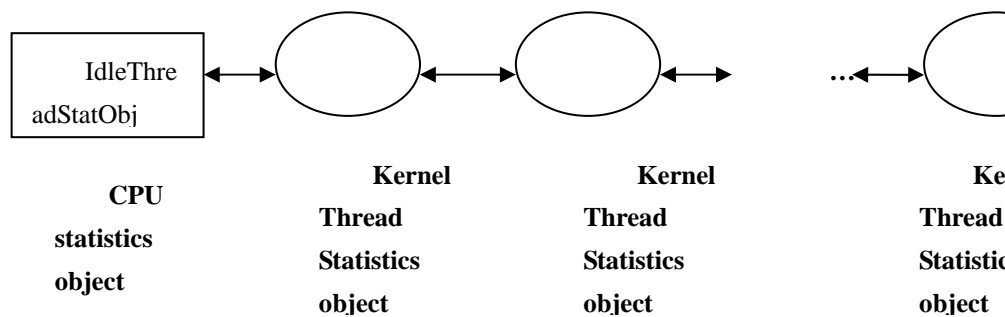


图 11-1 核心线程 CPU 统计对象的组织

其中，CPU 统计对象的详细信息，请参考本章“CPU 统计对象”一节。在 CPU 统计对象中，定义了一个名字为 IdleThreadStatObj 的核心线程统计对象，用于记录 Idle 线程的 CPU 占用率。所有其它创建的核心线程对应的统计对象，都挂接在 IdleThreadStatObj 的后面，这样系统中所有核心线程的统计对象，就连接成了一个双向链表。

- 该对象其它成员的含义如下：
- TotalCpuCycle: 与之对应的核心线程自运行以来，占用的 CPU 周期数的总和；
  - CurrPeriodCycle: 与之对应的核心线程最近一次被调度，所运行的 CPU 周期数；

- **PreviousTsc**: 核心线程被调入 CPU 运行的时候, 对应的 CPU 计数 (Time Stamp Counter);
- **lpKernelThread**: 该核心线程统计对象对应的核心线程;
- **wQueueHdr**、**wQueueTail** 和 **RatioQueue** 三个变量, 实现了一个循环队列, 其中 **wQueueHdr** 指向队列的头位置, **wQueueTail** 指向队列的尾, **RatioQueue** 则是存储队列元素的数组。按照当前的实现, 该队列共由 **MAX\_STAT\_PERIOD** 个元素 (定义为 300) 组成, 用于存放连续 300 个统计周期的 CPU 占用率。这样通过该队列记录的 CPU 占用率, 就可得出最近 5 分钟内任何时间段内, 核心线程的 CPU 占用率。需要注意的是, 这是一个循环队列, 一旦记录的统计周期多余 300 个, 队列尾部的元素就会被覆盖, 从而反映最新的统计结果;
- **wOneMinuteRatio**: 对应的核心线程在最近一分钟内的 CPU 占用率;
- **wMaxStatRatio**: 对应的核心线程在最大统计时间 (当前 5 分钟) 内的 CPU 占用率;
- **wCurrPeriodRatio**: 对应的核心线程在当前统计周期内的 CPU 占用率。

最后的一个变量, 用于填充整个结构体, 使得该结构体能够以 4 字节对齐。这也可以通过编译器的伪指令实现。

**Hello China** 对 CPU 占用率的统计动作, 分布在下列四个时刻。这四个时刻的动作综合起来, 可完成特定线程的 CPU 占用时间的记录工作 (具体的统计工作, 是另外一个单独的核心线程实现的, 详细信息请参考本章后续部分)。对 CPU 占用率的统计, 采用了 **Hello China V1.5** 版本实现的回调机制, 分别安装了下列四个时刻的回调函数:

- 1、核心线程创建的时候。CPU 统计对象 (参考本章 “CPU 统计对象” 一节) 在初始化的时候, 安装了线程创建回调函数。这样一旦一个核心线程被创建, 则会调用线程创建回调函数, 在线程回调函数中, 为创建的核心线程额外创建了一个核心线程统计对象, 并跟创建的线程绑定在一起;
- 2、核心线程被调度入 CPU, 准备执行的时候。CPU 统计对象安装了线程开始调度的回调函数, 这样一旦一个核心线程被调入 CPU 执行, 相应的回调函数会被调用。在回调函数中, 会记下当前的 CPU 时间戳计数器 (Time Stamp Counter, 后续简称为 TSC);
- 3、核心线程被换出 CPU, 结束当前运行时间片的时候。CPU 统计对象安装了结束调度回调函数, 在一个特定核心线程被换出 CPU 的时候, 结束调度回调函数会被调用。在这个回调函数中, 会再次记录当前的 CPU 时间戳计数器 (TSC), 并跟先前的记录结果相减, 累加到核心线程统计对象中 (**TotalPeriodCycle** 变量);
- 4、核心线程结束的时候。在该时刻, 核心线程结束回调函数会被调用, 从而记录当前的 CPU TSC 值, 并计算出最后一个时间片的执行时间, 累加到核心线程统计对象中。

详细的回调机制及其实现信息, 请参考本书 “**Hello China** 的回调机制” 一章。其中, 四个回调函数, 是由 CPU 统计对象在初始化的时候安装的, 请参考本文 “CPU 统计对象” 一节。下面详细解释这四个回调函数的实现, 这是这四个回调函数, 详细记录了核心线

程的执行时间（严格来说，应该是消耗的 CPU 周期数，即 TSC 计数的数量）。

### 核心线程创建回调函数

该函数在一个核心线程被创建的时候调用，由 CPU 统计对象在初始化的时候注册。实现代码如下：

```
static      DWORD      CreateHook(__KERNEL_THREAD_OBJECT*
lpKernelThread,DWORD* lpdwUserData)
{
    __THREAD_STAT_OBJECT*          lpStatObj    =
&StatCpuObject.IdleThreadStatObj;
    DWORD                          dwFlags;

    if((NULL == lpdwUserData) || (NULL == lpKernelThread))
//Invalid parameter.
    {
        return 0L;
    }
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(NULL == lpStatObj->lpKernelThread) //This stat object
was not used yet.
    {
        lpStatObj->lpKernelThread = lpKernelThread;
        *lpdwUserData              = (DWORD)lpStatObj;
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return 1;
    }
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    //
    //Should create a kernel stat object.
    //
    lpStatObj                                =
    (__THREAD_STAT_OBJECT*)GET_KERNEL_MEMORY(sizeof(__THREAD_STAT_
OBJECT));
```

```

if(NULL == lpStatObj) //Can not allocate memory.
{
    return 0L;
}
//Initialize this object.
lpStatObj->lpKernelThread    = lpKernelThread;
lpStatObj->CurrPeriodCycle.dwHighPart = 0;
lpStatObj->CurrPeriodCycle.dwLowPart  = 0;
lpStatObj->TotalCpuCycle.dwHighPart   = 0;
lpStatObj->TotalCpuCycle.dwLowPart    = 0;

lpStatObj->wQueueHdr    = 0;
lpStatObj->wQueueTail   = 0;
lpStatObj->lpPrev       = NULL;
lpStatObj->lpNext       = NULL;
lpStatObj->wMaxStatRatio = 0;
lpStatObj->wCurrPeriodRatio = 0;
lpStatObj->wOneMinuteRatio = 0;
MemZero((LPVOID)lpStatObj->RatioQueue, sizeof(lpStatObj->
RatioQueue)); //Clear memory.

*lpdwUserData          = (DWORD)lpStatObj; //Save this
object.

__ENTER_CRITICAL_SECTION(NULL, dwFlags);
//Insert this object into stat list.
lpStatObj->lpNext          =
StatCpuObject.IdleThreadStatObj.lpNext;
lpStatObj->lpPrev = &StatCpuObject.IdleThreadStatObj;

StatCpuObject.IdleThreadStatObj.lpNext->lpPrev          =
lpStatObj;
StatCpuObject.IdleThreadStatObj.lpNext          =
lpStatObj;

```

```
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
return 1L;
}
```

该函数完成的任务比较简单，首先判断 CPU 统计对象（一个全局对象，名字为 StatCpuObject）中的第一个线程统计对象（IdleThreadStatObj）是否被占用（若该统计对象的 lpKernelThread 为 NULL，则说明未被占用）。若未被占用，则初始化该对象，使得该对象作为当前创建的核心线程（由 lpKernelThread 指定）的统计对象。若该对象已被占用，则创建一个核心线程统计对象，初始化，并挂接到 IdleThreadStatObj 的后面，然后设定为当前创建的核心线程的统计对象。

在 Hello China V1.5 的实现中，第一个被创建的核心线程，就是 Idle 线程。因此，CPU 统计对象中定义的核心线程统计对象（IdleThreadStatObj），肯定会被 Idle 线程占用，这也是为什么这样命名的原因。

- 对于核心线程对象和核心线程统计对象的绑定关系，是通过两个变量进行的：
- 1、在核心线程统计对象中，定义了 lpKernelThread 指针，该指针被初始化为指向对应的核心线程对象；
  - 2、在核心线程对象的定义中，V1.5 版本的实现增加了一个额外的变量 dwUserData，该变量可被回调函数使用。实际上，在调用回调函数的时候，第一个参数是当前核心线程的指针，第二个参数（lpdwUserData），则是核心线程的 dwUserData 指针。这样只要把该指针指向的变量，设置为创建的核心线程统计对象，就可以把该统计对象连接到对应的核心线程对象上了。

下图示意了上面这种绑定关系：

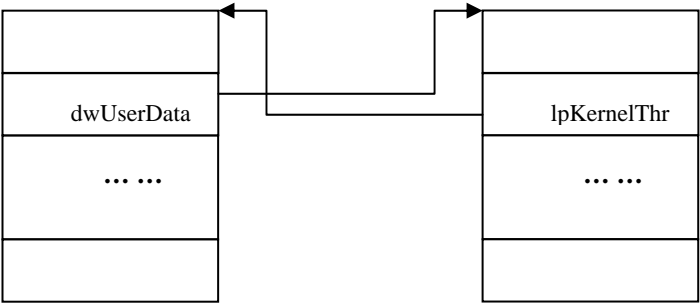


图 11-2 核心线程对象和核心线程统计对象的绑定关系

### 核心线程调入 CPU 回调函数

在核心线程得到调度，调入 CPU 执行的时候，操作系统核心会调用调入 CPU 回调函数。该函数实现代码如下：

```
static    DWORD    BeginScheduleHook(__KERNEL_THREAD_OBJECT*
lpKernelThread,

                                DWORD*

lpdwUserData)
{
    if((NULL == lpKernelThread) || (NULL == lpdwUserData))
    {
        return 0L;
    }

    __THREAD_STAT_OBJECT*    lpStatObj    =
(__THREAD_STAT_OBJECT*)(*lpdwUserData);
    __GetTsc(&lpStatObj->PreviousTsc);    //Save current time
stamp counter.

    return 1L;
}
```

该函数的功能十分简单，主要是通过获取当前核心线程对应的统计对象指针，并记录当前 CPU 的时间戳计数器。\_\_GetTsc 函数是一个与底层硬件相关的函数，该函数采用汇编语言实现，读取了 CPU 当前的 TSC 计数器，并保存到参数指定的变量中。

在 Hello China V1.5 的实现中，为了尽可能的减少代码和硬件平台的耦合程度，对所有需要汇编语言实现的功能，都单独拿了出来，采用独立的函数实现。这些函数的名称前面，都有两条连续的下划线，表示这样的函数是跟硬件相关的。其它类似的函数还有 \_\_SwitchTo 等。

### 核心线程调出 CPU 回调函数

在核心线程运行完了当前的时间片，准备切换出 CPU 的时候，该回调函数被调用。下面是调出 CPU 回调函数的实现代码：

```

static    DWORD    EndScheduleHook(__KERNEL_THREAD_OBJECT*
lpKernelThread,

                                DWORD*

lpdwUserData)
{
    if((NULL == lpKernelThread) || (NULL == lpdwUserData))
    {
        return 0L;
    }
    __THREAD_STAT_OBJECT*          lpStatObj          =
(__THREAD_STAT_OBJECT*)(*lpdwUserData);
    __U64                          currTsc;

    __GetTsc(&currTsc); //Get current time stamp counter.
    u64Sub(&currTsc,&lpStatObj->PreviousTsc,&currTsc); //Get
the difference.

    //Add the difference to current period cycle counter.
    u64Add(&lpStatObj->CurrPeriodCycle,&currTsc,&lpStatObj->
CurrPeriodCycle);

    //Add the difference to total CPU cycle counter.
    u64Add(&lpStatObj->TotalCpuCycle,&currTsc,&lpStatObj->To
talCpuCycle);
    return 0L;
}

```

该函数的功能也十分简单，主要完成下列工作：

- 1、调用\_\_GetTsc 函数，获取当前 CPU 的 TSC；
- 2、把当前的 TSC，跟前一次保存的 TSC（线程调入 CPU 运行时保存）相减，就可得到当前线程最近一次的运行时间（严格说，应该是 CPU 节拍数量）；
- 3、把该运行时间累加到当前统计周期内（lpStatObj->CurrPeriodCycle 变量），并累加到核心线程所有运行时间变量中（lpStatObj->TotalCpuCycle）。

这样核心线程最近一次的运行时间，就被记录了下来。

另，该函数涉及到一些 64 位整数的操作，以 u64 开头的函数，就是用于完成 64 位整数加、减、乘、除运算的辅助函数。由于当前版本的 Hello China 是 32 位的，且采用的编译环境也是 32 位的，无法直接支持 64 位整数的运算，因此在 Hello China 的开发中，专门实现了 64 位无符号整数的数据类型（\_\_U64）和对应的运算。

### 核心线程结束回调函数

在一个核心线程运行结束的时候，操作系统核心会调用该函数。该函数由 CPU 统计对象注册，代码如下：

```
static      DWORD      TerminalHook(__KERNEL_THREAD_OBJECT*
lpKernelThread,

                                DWORD*

lpdwUserData)
{
    if((NULL == lpKernelThread) || (NULL == lpdwUserData))
//Invalid parameters.
    {
        return 0L;
    }

    __THREAD_STAT_OBJECT*      lpStatObj      =
(__THREAD_STAT_OBJECT*)(*lpdwUserData);
    DWORD                      dwFlags;

    //Delete this statistics object from stat object list.
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    lpStatObj->lpNext->lpPrev = lpStatObj->lpPrev;
    lpStatObj->lpPrev->lpNext = lpStatObj->lpNext;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);

    //Free this object.
    FREE_KERNEL_MEMORY(lpStatObj);
    return 1L;
```

```
}

```

该函数非常简单，主要是从线程统计对象链表中，把结束运行的核心线程对应的统计对象删除掉，并释放对应的内存。

## CPU 统计对象

### *CPU 统计对象的定义*

CPU 统计对象是 Hello China 当前版本实现的用于完成核心线程 CPU 占用率统计的对象，该对象是一个全局对象，核心线程 CPU 统计相关的功能，都是在该对象内实现的。原型定义如下：

```
struct __STAT_CPU_OBJECT{
    __U64      PreviousTsc;//Previous time stamp counter.
    __U64      CurrPeriodCycle;//CPU cycle conter in this stat
period.
    __U64      TotalCpuCycle;        //Total CPU cycle counter
since
                                     //system startup.
    __THREAD_STAT_OBJECT      IdleThreadStatObj;

    BOOL      (*Initialize)(__STAT_CPU_OBJECT*);//Initialize
routine.
    __THREAD_STAT_OBJECT*      (*GetFirstThreadStatObj)();
    __THREAD_STAT_OBJECT*
(*GetNextThreadStatObj)(__THREAD_STAT_OBJECT*);
    VOID      (*DoStat)();
    VOID      (*ShowStat)();
};

```

各成员含义如下：

- **PreviousTsc**：上一个统计周期结束时的 CPU TSC 计数。每个统计周期结束的时候，都会更新该变量，这样在下一个统计周期结束的时候，就可以通过再次读取 CPU TSC 计数，并跟 PreviousTsc 相减，从而获得该统计周期的 CPU 节拍数。因为在 Hello China 当前版本的实现中，统计周期是通过定时器实现的一个时间间隔，该时间间隔大致

维持在 1s 左右，但不会十分精确。因此直接用统计周期除以 CPU 频率获得运行时间或节拍的方法，是不准确的。而通过获取 CPU TSC 计数器，是最准确的方法；

- **CurrPeriodCycle**: 当前统计周期的 CPU 节拍数；
- **TotalCpuCycle**: 系统自启动以来到上一统计周期结束的时候的 CPU TSC 计数。该变量实际上就是存储了 CPU 的 TSC 寄存器的值，只不过不是实时变化的，而是在每个统计周期结束的时候更新；
- **IdleThreadStatObj**: 内置的核心线程统计对象。该对象作为线程统计对象链表的头元素，也可用于作为系统中第一个核心线程的统计对象。因为在目前的实现中，系统中第一个创建的核心线程就是 Idle 线程，所以把该对象命名为 IdleThreadStatObj(Idle 线程统计对象)。虽然大多数情况下，该对象作为 Idle 线程的统计对象使用，但也可以作为其它核心线程的统计对象使用。
- **Initialize** 函数: CPU 统计对象的初始化函数，在操作系统初始化过程中被调用，用于完成 CPU 统计对象的初始化功能；
- **GetFirstThreadStatObj**: 获取 CPU 统计对象中的第一个核心线程统计对象。该函数和 **GetNextThreadStatObj** 联合使用，用于完成 CPU 统计对象中核心线程统计对象链表的遍历；
- **GetNextThreadStatObj**: 与 **GetFirstThreadStatObj** 联合使用，用于完成 CPU 统计对象中线程统计对象链表的遍历；
- **DoStat**: 该函数在一个统计周期结束的时候被调用，用于完成该统计周期内，各核心线程的 CPU 占用率统计。该函数更新系统中所有核心线程的线程统计对象，从而反映该统计周期结束时，各核心线程的 CPU 占用率情况；
- **ShowStat**: 打印出所有核心线程的 CPU 占用率信息。

CPU 统计对象是一个静态的对象，也就是说，该对象仅提供一些功能函数和变量，这些功能函数会被被动的调用。在 Hello China V1.5 的实现中，通过一个专门的统计核心线程(STAT\_S)实现了 CPU 占用率的统计功能。在该线程中，设定了一个定时器(1000ms)，用于实现统计周期。在每个定时器消息的处理函数内，都会调用 **DoStat** 完成各核心线程的 CPU 占用率的计算工作，并保存在各核心线程的线程统计对象里面。

### *CPU 统计对象的初始化*

CPU 统计对象的初始化工作较简单，主要就是完成了核心线程回调函数(Hook 函数)的设定工作，其中统计对象链表的初始化、读取当前 CPU TSC 计数器等工作，也包含在该函数里面，代码如下：

```
static BOOL Initialize(__STAT_CPU_OBJECT* lpStatObj)
{
    lpCreateHook          = CreateHook;
    lpBeginScheduleHook   = BeginScheduleHook;
```

```
        lpEndScheduleHook      = EndScheduleHook;
        lpTerminalHook         = TerminalHook;

        //Install hook routines.
        KernelThreadManager.SetThreadHook(THREAD_HOOK_TYPE_CREATE,CreateHook);
        KernelThreadManager.SetThreadHook(THREAD_HOOK_TYPE_BEGIN SCHEDULE,BeginScheduleHook);
        KernelThreadManager.SetThreadHook(THREAD_HOOK_TYPE_ENDSC HEDULE,EndScheduleHook);
        KernelThreadManager.SetThreadHook(THREAD_HOOK_TYPE_TERMI NAL,TerminalHook);

        //Initialize the StatCpuObject.
        StatCpuObject.IdleThreadStatObj.lpNext
        &StatCpuObject.IdleThreadStatObj;
        StatCpuObject.IdleThreadStatObj.lpPrev
        &StatCpuObject.IdleThreadStatObj;

        //Save current CPU cycle counter.
        __GetTsc(&lpStatObj->PreviousTsc);

        return TRUE;
    }
```

其中，调用 `KernelThreadManager.SetThreadHook` 函数，设定了四个回调函数。这四个回调函数被操作系统调用的时机如下表：

函数名称	回调时机
CreateHook	在一个核心线程被创建的时候调用
BeginScheduleHook	在核心线程被调入 CPU 开始执行的时候调用

EndScheduleHook	在核心线程被调出 CPU，结束当前执行时间片的时候调用
TerminalHook	核心线程结束的时候调用

正是这四个回调函数，完成了系统中核心线程的 CPU 占用信息的记录功能（这四个函数操作核心线程的统计对象，把每个核心线程的 CPU 占用信息，保存在统计对象里）。详细信息，请参考本章“核心线程统计对象”一节。

### 核心线程 CPU 占用率统计

其中，DoStat 函数被周期性的调用，完成了一个特定统计周期内的 CPU 占用率的计算。该函数被统计线程周期性的调用。其实现代码如下，由于该函数相对复杂，我们分段进行解释：

```
static VOID DoStat()
{
    __STAT_CPU_OBJECT*    lpStatObj        = &StatCpuObject;
    __THREAD_STAT_OBJECT* lpThreadStatObj = NULL;
    __U64                  currTsc;
    __U64                  temp;
    __U64                  remainder;
    WORD                   ratio;
```

下面的代码，读取了当前 CPU 的 TSC 计数器，存储在 currTsc 变量里面。并跟 CPU 统计对象保存的上一次 TSC 计数器相减，取一个差值，这个差值就是本次统计周期内，CPU 运行的节拍数。

```
    __GetTsc(&currTsc);
    u64Sub(&currTsc, &lpStatObj->PreviousTsc, &currTsc); //Get
CPU cycle difference.
```

下面的代码，把当前 CPU 的 TSC 计数器存储在 CPU 统计对象的 PreviousTsc

变量里面，作为下一个统计周期的起始 TSC。然后，把 CPU 统计对象的当前统计周期的节拍数，设置为刚才计算的差值。同时，也把当前统计周期内 CPU 运行的节拍数，累加到 TotalCpuCycle 变量里面。

```

    __GetTsc(&lpStatObj->PreviousTsc);    //Save current time
stamp counter.

    lpStatObj->CurrPeriodCycle.dwHighPart    =
currtsk.dwHighPart;

    lpStatObj->CurrPeriodCycle.dwLowPart    =
currtsk.dwLowPart;

    //Accumulate total CPU cycle since startup.
    u64Add(&lpStatObj->TotalCpuCycle,&currtsk,&lpStatObj->To
talCpuCycle);

```

上面代码执行完毕之后，计算核心线程 CPU 占用率的一些变量就初始化好了。其中最重要的一个，就是 currtsk 变量，该变量记录了当前统计周期内，CPU 运行的节拍数。该变量是作为核心线程 CPU 占用率计算的分母部分的。

在目前版本的 Hello China 中，尚未实现浮点运算支持。但在计算 CPU 占用率的时候，却需要详细到小数点后一位的精确度。而且在计算 CPU 占用率的时候，分子是单核心线程的 CPU 节拍数，而分母则是该统计周期内 CPU 运行的所有节拍数，显然计算结果小于 1，这样必须要有浮点运算的支持。

为了解决这个问题，采用下列方式进行模拟浮点运算：

- 1、假设分子为 M，分母为 N，其中  $M < N$ ，计算结果为一个百分比，假设为 Q.P%，其中 Q 为整数部分，P 为小数部分；
- 2、首先计算  $N / 1000$ ，这可以通过整数计算来模拟，无需浮点运算，误差小于 1，是可以接受的。假设计算结果为 N1；
- 3、再计算  $M / N1$ ，假设计算结果为 M1；
- 4、则 Q 就等于  $M1 / 10$ （取整数部分），P 就等于  $M1 \% 10$ 。

通过上述方式，就避免了浮点运算。下面的计算核心线程 CPU 占用率的代码，就是通过上述模拟算法进行的。

首先，remainder 是定义的一个 64 位整数，初始化为 1000，然后调用 u64Div 函数，计算 currtsk / remainder，其中计算结果的整数部分，保存在 currtsk 里面，余数部分，则保存在 remainder 里面。这样 u64Div 执行完毕后，currtsk

和 remainder 都将被改变:

```
//Calculate each thread's CPU occupancy.
INT i,j;
remainder.dwHighPart = 0;
remainder.dwLowPart  = 1000; //Divisor used to shrink
currtscc later.
u64Div(&currtscc,&remainder,&currtscc,&remainder);
//currtscc = currtscc / 1000.
```

接下来就是遍历 CPU 统计对象维护的线程统计对象列表, 针对每个核心线程, 计算其 CPU 占用率。需要注意的是, 针对每个核心线程, 实际上计算了三个时间间隔内的 CPU 占用率:

- 1、当前统计周期内的 CPU 占用率;
- 2、最近 1 分钟内的 CPU 占用率;
- 3、最近 5 分钟内的 CPU 占用率。

下面的代码, 首先取得核心线程的当前统计周期内运行的 CPU 节拍数 (CurrPeriodCycle 变量), 然后采用该变量除以 currtscc, 并把结果保存在线程统计对象的 wCurrPeriodRatio 里面。实际上这个计算结果还不是真正的占用率, 而应该是真实占用率的 1000 倍。在输出结果的时候, 只要把该数值除以 10, 取其整数部分, 就是 CPU 占用率百分比的整数部分, 然后跟 10 取模, 就是 CPU 占用率百分比的小数部分。在计算当前周期的 CPU 占用率后, 会把统计对象的 CurrPeriodCycle 清零, 这样就可重新记录下一个统计周期内该线程的 CPU 占用节拍数。

```
lpThreadStatObj = &lpStatObj->IdleThreadStatObj;
do{
    temp = lpThreadStatObj->CurrPeriodCycle;
    u64Div(&temp,&currtscc,&temp,&remainder); //temp = temp
/ currtscc.
    ratio = (WORD)(temp.dwLowPart);
    //Clear this period's counter of the kernel thread.
    lpThreadStatObj->CurrPeriodCycle.dwHighPart = 0;
```

```
lpThreadStatObj->CurrPeriodCycle.dwLowPart = 0;
```

下面代码，把当前统计周期的计算结果，保存在 wCurrPeriodRatio 变量里面，同时也保存在占用率队列里面。其中占用率队列记录了当前核心线程自运行以来，最近 MAX\_STAT\_PERIOD 个统计周期的 CPU 占用率（当前为 300）。该队列用于计算核心线程最近 1 分钟的 CPU 占用率和最近 5 分钟的 CPU 占用率。

```
//Save calculation result to kernel thread's statistics
object.
```

```
lpThreadStatObj->RatioQueue[lpThreadStatObj->wQueueHdr]    =
ratio;
lpThreadStatObj->wCurrPeriodRatio                            =
ratio;
```

下面的代码，更新了队列指针。占用率队列是一个 300 个元素的数组，只用于记录最近 300 个统计周期内的 CPU 占用率。若超过了 300 个统计周期，则最远的占用率统计数据会被覆盖掉。

```
//Update ratio queue's pointers.
lpThreadStatObj->wQueueHdr += 1;
if(lpThreadStatObj->wQueueHdr == MAX_STAT_PERIOD)
//Exceed queue length.
{
    lpThreadStatObj->wQueueHdr = 0;
}
if(lpThreadStatObj->wQueueHdr ==
lpThreadStatObj->wQueueTail)
{
    lpThreadStatObj->wQueueTail += 1;
    if(MAX_STAT_PERIOD == lpThreadStatObj->wQueueTail)
    {
        lpThreadStatObj->wQueueTail = 0;
    }
}
```

下面的代码分别计算了最近 1 分钟和最近 5 分钟的CPU占用率。计算方法采用了本章“[统计周期和统计算法](#)”一节中介绍的方法，即把最近 60 个统计周期（针对一分钟）或 300 个统计周期（针对 5 分钟）的CPU占用率进行累加，然后再除以对应的周期数。其中MAX\_STAT\_PERIOD和ONE\_MINUTE\_PERIOD是预先定义的两个常数，MAX\_STAT\_PERIOD定义为 300，ONE\_MINUTE\_PERIOD定义为 60：

```
//Calculate CPU occupancy ration in last one minute.
lpThreadStatObj->wOneMinuteRatio  = 0;
lpThreadStatObj->wMaxStatRatio    = 0;
for(i  =  0,j  =  lpThreadStatObj->wQueueHdr;i  <
MAX_STAT_PERIOD;i ++,j --)
{
    if(j <= 0)
    {
        j = MAX_STAT_PERIOD;
    }

    if(i < ONE_MINUTE_PERIOD)        //Accumulate last
ONE_MINUTE_PERIOD result.
    {
        lpThreadStatObj->wOneMinuteRatio      +=
lpThreadStatObj->RatioQueue[j - 1];
    }
    lpThreadStatObj->wMaxStatRatio      +=
lpThreadStatObj->RatioQueue[j - 1];
}
lpThreadStatObj->wOneMinuteRatio /= ONE_MINUTE_PERIOD;
lpThreadStatObj->wMaxStatRatio  /= MAX_STAT_PERIOD;
//Process next thread statistics object.
lpThreadStatObj = lpThreadStatObj->lpNext;
}while(lpThreadStatObj != &lpStatObj->IdleThreadStatObj);
}
```

到此为止，一个统计周期内所有核心线程的 CPU 占用率就计算完了。可以看出，DoStat 算法依次遍历每个核心线程对象，对每个核心线程对象的 CPU 占用率进行计算。因此该算法的复杂度跟系统中核心线程的数量有关。

另外，DoStat 函数是在统计周期到达时，被统计线程调用的。下面对统计线程的实现进行描述。

## CPU 占用率统计线程

Hello China V1.5 的实现中，对 CPU 占用率的统计，是通过一个单独线程实现的。通过设定一个定时器，来模拟统计周期。下面的代码，是统计线程的入口函数，在操作系统初始化的时候，创建该统计线程。需要注意的是，该核心线程的创建，是在 CPU 统计对象初始化之后完成的。因为 CPU 统计对象初始化之后，相应的回调函数（Hook 函数）才会被安装到系统中，这样再调用 CreateKernelThread 函数创建统计线程（之前还有 Idle 线程），统计线程本身的 CPU 占用率，才会被统计。

下面是统计核心线程的入口代码：

```
DWORD StatThreadRoutine(LPVOID lpData)
{
    __TIMER_OBJECT*          lpTimer = NULL;
    __KERNEL_THREAD_MESSAGE  msg;

    //Set a timer,to calculate statistics information periodic.
    lpTimer =
    (__TIMER_OBJECT*)System.SetTimer((__COMMON_OBJECT*)&System,
        KernelThreadManager.lpCurrentKernelThread,
        1024,
        1000,
        NULL,
        NULL,
        TIMER_FLAGS_ALWAYS);

    while(TRUE)
    {
        if(GetMessage(&msg)) //Get message to process.
```

```

    {
        switch(msg.wCommand)
        {
            case KERNEL_MESSAGE_TIMER:
                StatCpuObject.DoStat(); //Do statistics.
                break;
            case STAT_MSG_SHOWSTATINFO:
                ShowStatInfo();
                break;
            case STAT_MSG_LISTDEV: //List the device
information.
                ShowDevList();
                break;
            case STAT_MSG_TERMINAL: //Should exit.
                goto __EXIT;
            default:
                break;
        }
    }
}

__EXIT:
    //Cancel the timer first.
    System.CancelTimer((__COMMON_OBJECT*)&System,
        (__COMMON_OBJECT*)lpTimer);
    return 0L;
}

```

该函数的功能十分简单，首先调用 `SetTimer`（由 `System` 系统对象提供）函数，设定一个定时器，用于模拟统计周期。该定时器设定为 `1000ms`，一旦定时器到时，该统计线程会收到一个 `KERNEL_MESSAGE_TIMER` 消息。这样统计线程在处理该消息的时候，调用了 `CPU` 统计对象的 `DoStat` 函数，从而完成对系统中所有核心线程 `CPU` 占用率的统计。

另外，该线程还可处理 `STAT_MSG_SHOWSTATINFO` 消息，该消息是统计线程

自定义的一个消息，用于在屏幕上打印出系统中所有核心线程的 CPU 占用率信息。在处理该消息的时候，调用了 `ShowStatInfo` 函数，该函数是一个静态函数，用于打印出系统中所有核心线程的 CPU 占用率信息。代码如下：

```
static VOID ShowStatInfo()    //Display the statistics
information.
{
    __THREAD_STAT_OBJECT*    lpStatObj    =
&StatCpuObject.IdleThreadStatObj;
    char Buff[MAX_BUFFER_SIZE];

    //Print table header.
    PrintLine("      Thread Name  Thread ID  1s usage  60s usage
5m usage");
    PrintLine("      -----      -----      -----
-----      -----");
    //For each kernel thread,print out statistics information.
    do{
        sprintf(Buff,"      %13s  %9X  %6d.%d  %7d.%d  %6d.%d",
            lpStatObj->lpKernelThread->KernelThreadName,
            lpStatObj->lpKernelThread->dwThreadID,
            lpStatObj->wCurrPeriodRatio / 10,
            lpStatObj->wCurrPeriodRatio % 10,
            lpStatObj->wOneMinuteRatio / 10,
            lpStatObj->wOneMinuteRatio % 10,
            lpStatObj->wMaxStatRatio / 10,
            lpStatObj->wMaxStatRatio % 10
            //lpStatObj->TotalCpuCycle.dwHighPart,
            //lpStatObj->TotalCpuCycle.dwLowPart
        );
        PrintLine(Buff);

        lpStatObj = lpStatObj->lpNext;
    }while(lpStatObj != &StatCpuObject.IdleThreadStatObj);
```

```
}
```

该函数通过遍历 CPU 统计对象维护的线程统计对象，依次打印出每个核心线程的 CPU 占用率信息，包括最近一个统计周期内的 CPU 占用率、最近 1 分钟的 CPU 占用率，以及最近 5 分钟的 CPU 占用率。需要注意的是，在计算 CPU 占用率的时候，保存的结果是实际结果的 1000 倍（为了避免浮点运算），因此在输出的时候，把保存结果除以 10，取整数部分，就是百分比的整数部分，保存结果对 10 取模，就是百分比的小数部分。这样实际上根本没有任何浮点运算，但输出的结果，却是精确到小数点后一位的。

Hello China V1.5 的实现中，实现了一个诊断程序 `sysdiag`。操作系统加载完成，进入命令行提示界面后，输入 `sysdiag`，按回车键，就可进入 `sysdiag` 操作界面。在 `sysdiag` 操作界面下，输入 `cpuload`，然后按回车，就可显示出系统中当前运行的所有核心线程的 CPU 占用率统计信息。

需要注意的是，若把系统中所有核心线程的 CPU 占用率累加一下，其结果并不是 1，而是 98% 左右。这是因为系统除了运行核心线程之外，还处理中断等工作，对于线程切换等系统动作，也是没有计算在内的。2% 的 CPU 占用率，可以认为是操作系统本身开销。按照统计，操作系统本身占用 2% 的 CPU 资源，是可以忍受的。

## 进程和多 CPU 情况下的考虑

### 进程的用户态和核心态执行时间统计

虽然在嵌入式开发领域，嵌入式操作系统很少引入进程的概念，但在通用操作系统领域，比如面向个人计算机的操作系统，大多数都引入进程的概念。而且为了确保整个系统的可靠性，还把 CPU 的运行模式，分成了至少两个级别：用户态和核心态。一般情况下，只有操作系统的核心代码和核心服务、设备中断程序运行在核心态，所有进程的用户代码，都运行在用户态。这样即使用户态的代码出现异常，也不会导致整个系统崩溃。

这种模型下，在对进程的执行时间进行统计的时候，就可以做得更精确一些，对一个进程的运行时间，可分开统计，分为核心态运行时间和用户态运行时间。这样做的好处是，可以让用户或程序员很精确的判断一个进程的组成，这在优化应用程序代码或排除问题的时候非常有用。比如，若一个进程的 CPU 占用率非常高，则可进一步判断是核心态执行时间多，还是用户态执行时间多。若是核心态执行时间较多，则说

明该应用程序调用了较多的系统调用，若用户态执行时间多，则说明程序的功能代码或算法可能效率不高。从而有针对性的对应用程序进行优化。

这样就需要为每个进程设置两个变量，一个变量记录了用户态的执行时间，假设为  $T1$ ，另外一个变量  $T2$  记录了应用程序在核心态的执行时间，假设为  $T2$ 。对于进程运行时间的统计，需要在四个时刻进行：

- 1、系统调用入口点处。在该位置，CPU 的执行状态从用户态转移到核心态，这时候，需要停止用户态时间  $T1$  的记录，开始核心态时间  $T2$  的记录；
- 2、系统调用出口处。在该位置，CPU 的执行状态从核心态转移到用户态。这时候，需要停止核心态运行时间  $T2$  的记录，并开始用户态运行时间  $T1$  的记录；
- 3、中断处理程序入口处。在该位置，需要进一步判断当前进程的执行状态是用户态，还是核心态。若是用户态，则停止用户态运行时间  $T1$  的记录，若是核心态，则停止核心态运行时间  $T2$  记录；
- 4、中断处理程序出口处。在该位置，需要进一步判断待恢复的进程的运行状态。若是核心态，则重新开始核心态运行时间  $T2$  的记录，若是用户态，则重新开始用户态运行时间  $T1$  的记录。

需要注意的是，上述假设没有发生进程切换。一般情况下，在中断处理程序结束（出口处）或系统调用的结束，会发生进程切换。这时候，就需要对重新运行的进程进行判断，判断其停止运行前（即被打断前）的状态（核心态还是用户态）。根据不同的状态，重新开始不同的运行时间记录（ $T1$  或  $T2$ ）。

在多线程的进程模型下，对于进程的 CPU 占用率，应该是属于该进程的所有线程的 CPU 占用率之和。

## 多 CPU 环境下的考虑

在多 CPU 的环境下（比如 SMP），线程或进程真正并行的运行在各不同的 CPU 上。在这种环境下，统计线程或进程的 CPU 占用率的时候，跟在单 CPU 环境下差别不大。因为统计算法很单纯，就是在线程或进程被调度进入 CPU 的时候，开始统计，在被换出 CPU 的时候，结束统计时间，把所有的运行时间累加起来，就是线程或进程的真正 CPU 占用时间。但在多 CPU 环境下，单一的线程或进程，可能会分布在不同的 CPU 上进行处理。这虽然对 CPU 占用率统计来说没什么影响，但在计算整体的 CPU 占用率的时候，分子是线程的 CPU 占用时间，分母则是所有 CPU 的运行时间。比如，在一个统计周期内（假设为  $T$ ），有  $N$  个 CPU，假设有一个线程或进程，统计的运行时间为  $T1$ ，则该线程或进程的 CPU 占用率应该为：

$$T1/(N \times T) \times 100\%$$

在多 CPU 环境下，占用率的统计，也可详细到每个 CPU 的粒度，即针对同一个线程，统计出该线程在每个 CPU 上的运行时间（从而可计算出该线程或进程在每个 CPU 上的占用率）。比如，假设有 N 个 CPU，M 个线程或进程，则需要维护每个线程或进程对每个 CPU 的占用率，这是一个二维的表格，如下所示：

线 程 / 进程	CPU1	CPU2	CPU3	... ..	CPU <sub>n</sub>
线程 1					
线程 2					
线程 3					
... ..					
线程 m					

这样在实现的时候，每个线程需要针对每个统计周期，定义 N 个变量，对应 N 个 CPU，来分别记录每个 CPU 上运行的时间。

# 附录 C 系统核心 HOOK 机制的实现

## — The implementation of Hook

**by GarryXin**  
**2006/10/09**

## Hook 概述

Hook 俗称钩子，实际上是一种函数回调机制。通过 Hook，可不修改系统框架，而实现用户定制功能。一个比较典型的例子，是 Windows 操作系统提供的列表控件（List Control），该控件提供了列表项目排序功能，可通过比较每个列表项目的特定属性值，来升序或降序排列。但每个列表项目及其属性值，都是用户通过编程方式添加的，系统无法提前知晓。这样为了实现排序功能，列表框控件提供了一个比较回调函数，用户可按照 Windows 定义的原型，编写特定的比较函数，然后把这个比较函数挂接到 List Control 即可。这样用户在单击列表框的表头的时候，列表框就可调用用户定义的比较函数，对每个列表项目进行比较并排序显示。

Hook 机制得益于 C 语言强大的函数指针功能。在实现上，实际上是定义了一个函数指针，并实现一个接口（函数），来设置或取消该指针。当特定的时机到达，或特定的事件发生的时候，实现 Hook 机制的代码会检查该函数指针，如果不为空，则调用该函数。

在操作系统的设计和实现中，Hook 机制得到了大量的应用。通过 Hook 机制，可在不改变整体系统构架和代码的情况下，很容易的增加一些辅助功能。在 Hello China V1.5 的实现中，实现了一种很有用的 Hook 机制：线程 Hook。通过线程 Hook 机制，用户可方便的增加系统相关的功能，比如线程的 CPU 占用率统计、浮点运算支持等。在本章中，就对 Hello China V1.5 的线程 Hook 机制的实现进行详细描述，并对其应用进行简单解释。在 V1.5 版本的实现中，也采用 Hook 机制实现了核心线程 CPU 占用率统计功能，可参考“CPU 占用率及其统计”一章，获取详细信息。

## 线程 Hook 的实现

核心线程被创建、调换上 CPU、调入 CPU、运行结束的四个时刻，对核心线程来说都是很关键的时刻。在这些时刻，可对线程的运行状态、运行时间等进行统计，也可采取特定的动作，改变核心线程的行为。在 Hello China V1.5 的实现中，在这四个时刻，分别安插了一个回调函数指针，若该函数指针不为空，则系统会调用该函数。用户可通过设定这些函数指针，来实现定制的功能。这就是 Hello China 的线程 Hook。

## 线程 Hook 的实现概述

在 V1.5 的实现中，在下列四个时机，会调用用户设定的回调函数（Hook 函数）：

- 1、核心线程创建的时候；
- 2、核心线程得到调度，获得 CPU 资源前；
- 3、当前运行的线程，被切换出去的时候；
- 4、线程结束的时候。

回调函数的原型如下：

```
DWORD __CALLBACK (*__THREAD_HOOK_ROUTINE)(
    __KERNEL_THREAD_OBJECT lpKernelThread,
    DWORD*                    lpdwUserData);
```

其中，lpKernelThread 的含义，在上述不同的时机，有不同的含义，如下：

- 1、在核心线程创建的时候，是刚刚创建的核心线程对象的指针；
- 2、在核心线程得到调度的时候，是即将运行的核心线程的指针；
- 3、在核心线程被切换出去的时候，是当前线程（即即将被切换出去的线程）的指针；
- 4、在线程结束的时候，是当前线程的指针。

这样站在一个线程的角度上看，就是该线程被创建的时候、该线程被调度运行的时候、该线程被暂停运行的时候、该线程结束的时候等四个时机，上述四个回调函数会被调用。这样可很容易的实现一种功能，就是对线程执行时间进行测量。

相应地，lpdwUserData 则是 lpKernelThread 的一个成员 dwUserData，该成员是 V1.5 的实现中增加的，如下：

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    INHERIT_FROM_COMMON_SYNCHRONIZATION_OBJECT
    __KERNEL_THREAD_CONTEXT          KernelThreadContext;
    __KERNEL_THREAD_CONTEXT*         lpKernelThreadContext;
    ... ..
    DWORD                            dwUserData;
    ... ..
    DWORD                            dwLastError;
END_DEFINE_OBJECT()
```

需要注意的是，虽然回调函数的参数是线程相关（即随着当前线程和即将投入运行

的线程的变化而变化)的,但这四个回调函数却是系统级的,即不论对哪个线程,这四个函数都相同。在 V1.5 的实现中,对于\_\_KERNEL\_THREAD\_MANAGER 对象,增加了四个函数指针,用于保存这四个回调函数,并增加了一个函数,用于设置或取消特定的回调函数。如下:

```
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_MANAGER)
    DWORD                                dwCurrentIRQL;
    __KERNEL_THREAD_OBJECT*              lpCurrentKernelThread;
//Current kernel thread.

    __PRIORITY_QUEUE*                    lpRunningQueue;
    //__PRIORITY_QUEUE*                  lpReadyQueue;
    __PRIORITY_QUEUE*                    lpSuspendedQueue;
    __PRIORITY_QUEUE*                    lpSleepingQueue;
    __PRIORITY_QUEUE*                    lpTerminalQueue;
    __PRIORITY_QUEUE* ReadyQueue[MAX_KERNEL_THREAD_PRIORITY + 1];

    __THREAD_HOOK_ROUTINE                lpCreateHook;
    __THREAD_HOOK_ROUTINE                lpBeginScheduleHook;
    __THREAD_HOOK_ROUTINE                lpEndScheduleHook;
    __THREAD_HOOK_ROUTINE                lpTerminalHook;

    __THREAD_HOOK_ROUTINE                SetThreadHook(DWORD dwHookType,
        __THREAD_HOOK_ROUTINE lpRoutine);
    VOID                                CallThreadHook(DWORD dwHookType,
        __KERNEL_THREAD_OBJECT* lpPrev,
        __KERNEL_THREAD_OBJECT* lpNext);

    ... ..

    END_DEFINE_OBJECT()                //End of the kernel thread manager's
definition.
```

其中,SetThreadHook 函数用于设置四个回调函数,或者取消设置的回调函数。该函数的两个参数中,dwHookType 指明了设置(或取消)的 HOOK 函数的类型,取值如下:

```
#define THREAD_HOOK_TYPE_CREATE        0x00000001
#define THREAD_HOOK_TYPE_BEGINSCHEDULE 0x00000002
#define THREAD_HOOK_TYPE_ENDSCHEDULE  0x00000004
```

```
#define THREAD_HOOK_TYPE_TERMINAL 0x00000008
```

而 `lpRoutine` 则是设置的回调函数指针，若该参数为 `NULL`，则会取消先前的设置。该函数返回先前设置的回调函数指针。

`CallThreadHook` 则是一个辅助函数，用来完成实际回调函数的调用，这样操作系统核心在实现的时候，就无需挨个调用每个回调函数，只需要采用合适的参数，调用该函数即可。该函数的实现，在本文后续部分会进行详细描述。

## 线程调度前后的回调机制

`BEGIN_SCHEDULE` 和 `END_SCHEDULE` 两个回调函数，在 V1.5 的实现中，会在下列几个时机内调用：

- 1、`ScheduleFromProc` 函数内，该函数保存当前线程的上下文，并切换到新的线程，因此在该函数内，会以当前线程对象指针为参数，调用 `END_SCHEDULE` 回调函数，并用新投入运行的线程对象指针，调用 `BEGIN_SCHEDULE` 回调函数；
- 2、`ScheduleFromInt` 函数，该函数在中断处理程序结束时调用，用于恢复中断前线程上下文，或切换到新的线程。这时候，该函数应该以当前线程对象指针，或新选择的核心理程对象指针为参数，调用 `BEGIN_SCHEDULE` 回调函数；
- 3、在中断发生后，这时候当前执行的线程被打断，需要在中断上下文中以当前线程对象指针调用 `END_SCHEDULE` 函数。

这样针对每个核心线程，`BEGIN_SCHEDULE` 和 `END_SCHEDULE` 两个回调函数会成对出现，符合逻辑。

### ScheduleFromProc 函数中的回调

`ScheduleFromProc` 函数用于在非中断上下文中完成线程的重调度。该函数十分关键，是 Hello China 核心理程调度机制的关键函数之一。在 V1.5 的实现中，对 `ScheduleFromProc` 函数进行了扩展。其中，线程 Hook 机制的实现，是 V1.5 扩展的重要内容之一。下面代码中的黑体标注部分，是与线程 Hook 实现相关的，如下：

```
static VOID ScheduleFromProc(__KERNEL_THREAD_CONTEXT* lpContext)
{
    __KERNEL_THREAD_OBJECT*      lpKernelThread    = NULL;
    __KERNEL_THREAD_OBJECT*      lpCurrent          = NULL;
    __KERNEL_THREAD_OBJECT*      lpNew              = NULL;
    __KERNEL_THREAD_CONTEXT**     lppOldContext      = NULL;
```

```

__KERNEL_THREAD_CONTEXT**      lppNewContext      = NULL;
DWORD                           dwFlags            = 0L;

__ENTER_CRITICAL_SECTION(NULL,dwFlags);
lpCurrent = KernelThreadManager.lpCurrentKernelThread;
switch(lpCurrent->dwThreadStatus)    //Do different actions
according to status.
{
case KERNEL_THREAD_STATUS_RUNNING:
    lpNew = KernelThreadManager.GetScheduleKernelThread(
        (__COMMON_OBJECT*)&KernelThreadManager,
        lpCurrent->dwThreadPriority);    //Get a ready kernel
thread.
    if(NULL == lpNew) //Current one is most priority whose status
is READY.
    {
        lpCurrent->dwTotalRunTime += SYSTEM_TIME_SLICE;
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return; //Let current kernel thread continue to run.
    }
    else //Should swap out current kernel thread and run next ready
one.
    {
        lpCurrent->dwThreadStatus = KERNEL_THREAD_STATUS_READY;
        KernelThreadManager.AddReadyKernelThread(
            (__COMMON_OBJECT*)&KernelThreadManager,
            lpCurrent); //Insert into ready queue.
        lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
        lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
        KernelThreadManager.lpCurrentKernelThread = lpNew;
        CallThreadHook(THREAD_HOOK_TYPE_ENDSCHEDULE
            | THREAD_HOOK_TYPE_BEGINSCHEDULE,lpCurrent,lpNew);
        __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
            &lpNew->lpKernelThreadContext); //Switch.
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return;
    }
    break;
case KERNEL_THREAD_STATUS_READY:
    lpNew = KernelThreadManager.GetScheduleKernelThread(
        (__COMMON_OBJECT*)&KernelThreadManager,

```

```

        lpCurrent->dwThreadPriority); //Get a ready thread.

if(NULL == lpNew) //Should not occur.
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    BUG();
    return;
}
if(lpNew == lpCurrent) //The same one.
{
    lpCurrent->dwTotalRunTime += SYSTEM_TIME_SLICE;
    lpCurrent->dwThreadStatus
    =
    KERNEL_THREAD_STATUS_RUNNING;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return;
}
else //Not the same one.
{
    lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
    lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
    KernelThreadManager.lpCurrentKernelThread = lpNew;
    CallThreadHook(THREAD_HOOK_TYPE_ENDSCHEDULE
        | THREAD_HOOK_TYPE_BEGINSCHEDULE,lpCurrent,lpNew);
    __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
        &lpNew->lpKernelThreadContext);
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return;
}
break;
case KERNEL_THREAD_STATUS_BLOCKED:
case KERNEL_THREAD_STATUS_SUSPENDED:
case KERNEL_THREAD_STATUS_TERMINAL:
    lpNew = KernelThreadManager.GetScheduleKernelThread(
        (__COMMON_OBJECT*)&KernelThreadManager,
        0L); //Get a ready thread to run.

if(NULL == lpNew)
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    BUG();
    PrintLine("Fatal error: in ScheduleFromProc,lpNew ==

```

```

NULL.");

        return;
    }
    lpNew->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
    lpNew->dwTotalRunTime += SYSTEM_TIME_SLICE;
    KernelThreadManager.lpCurrentKernelThread = lpNew;
    CallThreadHook(THREAD_HOOK_TYPE_ENDSCHEDULE |
        THREAD_HOOK_TYPE_BEGINSCHEDULE, lpCurrent, lpNew);
    __SaveAndSwitch(&lpCurrent->lpKernelThreadContext,
        &lpNew->lpKernelThreadContext);
    __LEAVE_CRITICAL_SECTION(NULL, dwFlags);
    return;
    break;
default:
    BUG();
    break;
}
}

```

只要发生核心线程的切换（由当前核心线程切换到另外一个不同的核心线程），就会调用 `CallThreadHook` 函数，该函数继而根据参数，调用对应的回调函数。需要注意的是，如果 `ScheduleFromProc` 函数经过检查后，发现当前核心线程应该继续执行（比如，当前核心线程的优先级最高），则不会发生线程切换，从而不会导致 `CallThreadHook` 的调用。

## ScheduleFromInt 函数中的回调

`ScheduleFromInt` 函数在设备中断处理程序结束的时候被调用，用于完成核心线程的重新调度。该函数实现代码如下，其中跟线程 `Hook` 有关的代码，用黑体做了标注：

```

static VOID ScheduleFromInt(__COMMON_OBJECT* lpThis, LPVOID lpESP)
{
    __KERNEL_THREAD_OBJECT*      lpNextThread    = NULL;
    __KERNEL_THREAD_OBJECT*      lpCurrentThread = NULL;
    __KERNEL_THREAD_MANAGER*      lpMgr           = NULL;

    if((NULL == lpThis) || (NULL == lpESP))    //Parameters check.
        return;

    lpMgr = (__KERNEL_THREAD_MANAGER*)lpThis;

```

```

        if(NULL == lpMgr->lpCurrentKernelThread)    //The routine is called
first time.
    {
        lpNextThread = KernelThreadManager.GetScheduleKernelThread(
            (__COMMON_OBJECT*)&KernelThreadManager,
            0L);
        if(NULL == lpNextThread)                    //If this case is
occurs,the system is crash.
    {
        BUG();
        return;
    }
        KernelThreadManager.lpCurrentKernelThread = lpNextThread;
        lpNextThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
        lpNextThread->dwTotalRunTime += SYSTEM_TIME_SLICE;

CallThreadHook(THREAD_HOOK_TYPE_BEGINSCHEDULE,NULL,lpNextThread);
        __SwitchTo(lpNextThread->lpKernelThreadContext);    //Switch
to this thread.
    }
    else //Not the first time be called.
    {
        lpCurrentThread = KernelThreadManager.lpCurrentKernelThread;
        //This code line saves the context of current kernel thread.
        lpCurrentThread->lpKernelThreadContext
        =
        (__KERNEL_THREAD_CONTEXT*)lpESP;

        switch(lpCurrentThread->dwThreadStatus)
        {
        case KERNEL_THREAD_STATUS_BLOCKED:
        case KERNEL_THREAD_STATUS_TERMINAL:
        case KERNEL_THREAD_STATUS_SLEEPING:
        case KERNEL_THREAD_STATUS_SUSPENDED:
        case KERNEL_THREAD_STATUS_READY:
            lpCurrentThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
            CallThreadHook(THREAD_HOOK_TYPE_BEGINSCHEDULE,
                NULL,lpCurrentThread);
            __SwitchTo(lpCurrentThread->lpKernelThreadContext);
            break;

```

```

        case KERNEL_THREAD_STATUS_RUNNING:
            lpNextThread
KernelThreadManager.GetScheduleKernelThread(
                (__COMMON_OBJECT*)&KernelThreadManager,
                lpCurrentThread->dwThreadPriority);
            if(NULL == lpNextThread) //Current is most priority.
            {
                lpCurrentThread->dwTotalRunTime
SYSTEM_TIME_SLICE;
                CallThreadHook(THREAD_HOOK_TYPE_BEGINSCHEDULE,
                    NULL,lpCurrentThread);
                __SwitchTo(lpCurrentThread->lpKernelThreadContext);
                return;
            }
            else
            {
                lpCurrentThread->dwThreadStatus
KERNEL_THREAD_STATUS_READY;
                KernelThreadManager.AddReadyKernelThread(
                    (__COMMON_OBJECT*)&KernelThreadManager,
                    lpCurrentThread); //Add to ready queue.

                lpNextThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
                lpNextThread->dwThreadStatus
KERNEL_THREAD_STATUS_RUNNING;
                lpMgr->lpCurrentKernelThread = lpNextThread;
                CallThreadHook(THREAD_HOOK_TYPE_BEGINSCHEDULE,NULL,
                    lpNextThread);
                __SwitchTo(lpNextThread->lpKernelThreadContext);
                return;
            }
        default:
            BUG();
            break;
    }
}
}

```

需要注意的是，在该函数中调用 `CallThreadHook` 的时候，只指定了 `BEGINSCHEDULE` 标记，这样 `CallThreadHook` 函数只会调用核心线程结束调度的 hook

函数。这与 `ScheduleFromProc` 不同，在 `ScheduleFromProc` 的实现中，调用 `CallThreadHook` 的时候，指定的调用标记是 `BEGINSCHEDULE` 和 `ENDSCHEDULE`。这是因为，在 `ScheduleFromProc` 函数中，若发生线程切换，总是伴随着当前核心线程被切换出 CPU，所以应该以当前核心线程为参数，调用 `ENDSCHEDULE` 回调函数。但在中断上下文中，在中断调度程序入口处，已经调用了 `ENDSCHEDULE` 回调函数了（以当前核心线程为参数），这是因为一旦中断发生，当前核心线程实际上已经被换出 CPU 了。所以在 `ScheduleFromInt` 的实现中，只会调用 `BEGINSCHEDULE` 回调函数。

## 线程创建和结束的回调机制

在线程创建函数 `CreateKernelThread` 函数执行到最后的时候，会调用 `CallThreadHook`，调用线程创建回调函数，实现如下：

```
static __KERNEL_THREAD_OBJECT *CreateKernelThread(
    __COMMON_OBJECT*          lpThis,
    DWORD                      dwStackSize,
    DWORD                      dwStatus,
    DWORD                      dwPriority,
    __KERNEL_THREAD_ROUTINE
lpStartRoutine,
    LPVOID                    lpRoutineParam,
    LPVOID                    lpReserved)
{
    ... ..

lpMgr->CallThreadHook(THREAD_HOOK_TYPE_CREATE,lpKernelThread,NULL);
    bSuccess = TRUE;

__TERMINAL:
    ... ..
    return lpKernelThread;
}
```

同样地，在 `DestroyKernelThread` 函数中，`TERMINAL HOOK` 会被调用，代码如下：

```
static VOID DestroyKernelThread(__COMMON_OBJECT*
lpThis,__COMMON_OBJECT* lpKernel)
{
```

```

    __KERNEL_THREAD_OBJECT*    lpKernelThread    = NULL;
    __KERNEL_THREAD_MANAGER*    lpMgr             = NULL;
    __PRIORITY_QUEUE*          lpTerminalQueue    = NULL;
    LPVOID                      lpStack           = NULL;

    if((NULL == lpThis) || (NULL == lpKernel))    //Parameter check.
        return;

    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpKernel;
    lpMgr          = (__KERNEL_THREAD_MANAGER*)lpThis;

    if(KERNEL_THREAD_STATUS_TERMINAL !=
lpKernelThread->dwThreadStatus)
        return;

    lpTerminalQueue = lpMgr->lpTerminalQueue;
    lpTerminalQueue->DeleteFromQueue((__COMMON_OBJECT*)lpTerminalQueue,
                                     (__COMMON_OBJECT*)lpKernelThread);

    lpMgr->CallThreadHook(THREAD_HOOK_TYPE_TERMINAL,lpKernelThread,
        NULL);
    lpStack = lpKernelThread->lpInitStackPointer;
    lpStack = (LPVOID)((DWORD)lpStack - lpKernelThread->dwStackSize);
    KMemFree(lpStack,KMEM_SIZE_TYPE_ANY,0L);
    ObjectManager.DestroyObject(&ObjectManager,
        (__COMMON_OBJECT*)lpKernelThread);
}

```

这样创建回调函数和结束回调函数就都会被调用，从而在逻辑上保持一致。一般情况下，创建回调函数一般用来分配资源，比如记录核心线程运行状态信息的内存等，而结束回调函数则用来释放分配的资源。

## CallThreadHook 例程的实现

CallThreadHook 实际上是一个封装函数，在该函数内，实现了实际的回调函数(HOOK 函数)的调用。之所以实现这样一个函数，是为了实现上的简便，以及减少代码量。

该函数实现如下：

```

VOID CallThreadHook(DWORD dwHookType, __KERNEL_THREAD_OBJECT* lpPrev,
                    __KERNEL_THREAD_OBJECT* lpNext)
{
    if(dwHookType & THREAD_HOOK_TYPE_CREATE) //Should call create
hook.
    {
        if(NULL == lpPrev)
        {
            BUG(); //Should not occur.
            return;
        }
        if(NULL == KernelThreadManager.lpCreateHook) //Not set yet.
        {
            return;
        }
        //Call the create hook routine.

KernelThreadManager.lpCreateHook(lpPrev,&lpPrev->dwUserData);
    }
    if(dwHookType & THREAD_HOOK_TYPE_ENDSCHEDULE) //Should call end
hook.
    {
        if(NULL == lpPrev)
        {
            BUG();
            return;
        }
        if(NULL == KernelThreadManager.lpEndScheduleHook)
        {
            return;
        }
        //Call end schedule hook now.

KernelThreadManager.lpEndScheduleHook(lpPrev,&lpPrev->dwUserData);
    }
    if(dwHookType & THREAD_HOOK_TYPE_BEGINSCHEDULE) //Should call
begin hook.
    {
        if(NULL == lpNext)
        {
            BUG();

```

```

        return;
    }
    if(NULL == KernelThreadManager.lpBeginScheduleHook)
    {
        return;
    }
    //Call begin schedule hook now.
    KernelThreadManager.lpBeginScheduleHook(
        lpNext,&lpNext->dwUserData);
}
if(dwHookType & THREAD_HOOK_TYPE_TERMINAL) //Should cal terminal
hook.
{
    if(NULL == lpPrev)
    {
        BUG();
        return;
    }
    if(NULL == KernelThreadManager.lpTerminalHook)
    {
        return;
    }
    //Call terminal hook now.

    KernelThreadManager.lpTerminalHook(lpPrev,&lpPrev->dwUserData);
}
}

```

该函数十分简单，只是根据 `dwHookType` 的不同取值，来调用特定的回调函数。需要注意的是，可以一次设定 `dwHookType` 为 `BEGINSCHEDULE`、`ENDSCHEDULE`、`CREATE`、`TERMINAL` 等的组合，但实际的情况中，却不会出现这种情况，因为不可能出现核心线程被创建即结束的情形。

## 线程 Hook 的应用

通过核心线程的 Hook 机制，可实现许多系统级的功能。比如，在 `Hello China V1.5` 的实现中，就通过 Hook 机制，实现了一个线程 CPU 占用率统计的功能。详细信息请参考“CPU 占用率及其统计”章节。

另外，通过线程的 Hook 机制，可实现浮点运算支持功能。虽然 `V1.5` 的实现中，没

有对 CPU 的浮点运算部件 (FPU) 进行初始化, 不能支持浮点运算功能, 但可通过 **Hello China V1.5** 提供的核心线程 Hook 机制, 很容易的增加该功能。比如, 可按照下列思路实现线程级别的 FPU 支持功能:

- 1、在核心线程创建的时候, 通过创建回调函数, 来初始化 FPU。这时候, 需要申请一块专用内存, 用于保存 FPU 的寄存器;
- 2、在核心线程被调度运行的时候, 通过 **BEGINSCHEDULE** 回调函数, 可恢复核心线程的 FPU 寄存器;
- 3、在核心线程被调度出 CPU 的时候, 通过 **ENDSCHEDULE** 回调函数, 保存当前核心线程的 FPU 寄存器信息 (保存到步骤 1 申请的内存中), 这样再下次核心线程被调入运行的时候, 就可恢复这些信息;
- 4、在核心线程结束 (**TERMINAL**) 回调函数中, 销毁申请的用于保存 FPU 寄存器的内存。

这样就可确保 FPU 对每个核心线程来说, 状态都是连续的, 因为 FPU 寄存器的值, 是针对每个核心线程进行保存和恢复的。

一些其它的功能, 也可通过线程的 Hook 机制实现, 在此不作赘述。

## 附录 D 如何搭建一个基于 Windows 的操作系统开发平台

## 总体概述

通常情况下，操作系统的开发都是基于 Linux 平台的，个人总结起来，主要有下列原因：

- 1、一些开发工具，比如 NASM、GCC 等，都是源代码开放的，跟 Linux 的源代码开放思想一致，而且在 Linux 平台下，这些工具可以得到更好的发挥；
- 2、操作系统开发者一般情况下都是 Linux 爱好者，而基于 Windows 平台的程序员，一般热衷于框架、组件、数据库等领域的程序设计，对操作系统层面的程序设计，涉及得相对少一些。

因此，虽然目前有很多团体和个人都在开发自己的操作系统，或源于兴趣，或源于应用，但搭建的开发平台大多数是基于 Linux 操作系统以及相应工具的，而基于 Windows 操作系统，以及 Windows 下编译工具的操作系统的开发环境，目前来说还很少。

实际上，基于 Windows 平台，以及 Windows 下开发工具的操作系统的开发环境，也是能够胜任操作系统开发任务的，而且如果能够熟练应用，其产出效率可能比 Linux 平台下的更高，因为相对 GCC 等编译工具，Microsoft Visual C++ 提供了更好的编程和编译界面，比如大家十分熟悉的联想功能（只要敲完函数名和左边的括号，函数原形就会联想出来），给程序编辑者带来了大大的方便，另外，基于 Windows 操作系统的编译工具，不但把程序编辑、调试的功能集成了起来，而且把端到端的程序设计理念，也集成到了里面，包括需求分析、概要设计、模块设计甚至最后的单元测试等。而且有的情况下，把一些团队管理的思想也集成到了集成开发环境里面，充分利用这些功能，对 OS 的开发有很大帮助。

但由于各种原因，基于 Windows 操作系统的快速开发工具（RAD）却不能直接应用于 OS 的开发，因为这些开发工具都是基于 Windows 平台的，与 Windows 平台绑定太紧密，而我们的目标则是一个跟 Windows 无任何关系的 OS，因此，如何充分剥离这些编译工具跟 Windows 的绑定关系，是正常利用这些 RAD 开发 OS 的关键所在。

在本文中，笔者详细分析 Windows 平台下各种开发工具的特点以及与 Windows 的绑定关系，并给出如何利用这些 Windows RAD 开发操作系统的基本原则，然后以 Microsoft Visual C++ 为例，搭建起一个具体的 OS 开发环境。

另外，再提醒一下读者，任何 OS 的开发，都离不开汇编语言的帮助，尤其是编写一个引导程序、操作底层的硬件等。即使在 Windows 开发环境里，使用内嵌的汇编代码也是必不可少的，因此，请读者一定掌握好汇编语言。

## Windows RAD 开发工具概述

Windows 下的 RAD 工具多重多样，但并不是每个 RAD 开发系统都适合 OS 的开发，比如，所有的 Java 开发环境都不适合于 OS 的开发，因为这些开发环境编译生成的代码不是真正的机器代码，而是一种中间层次的，与具体机器无关的字节码，这种字节码在一种成为虚拟机的软件支持下，才能正常运行。而操作系统的开发工具则要求，能够直接编译出机器代码。下面列举一些常用的 Windows 环境下的 RAD 开发工具，并分析这些工具是否适合 OS 的开发。

### 常用的 Windows RAD

#### Microsoft Visual C++

这是 Microsoft 公司开发的一款 Windows 平台下的编译工具。该工具不但提供了集成的代码编辑、编译、调试、运行等功能，而且提供了一个应用程序框架 MFC (Microsoft Foundation Class)，通过直接应用这个应用框架，Windows 程序员可以省却很多的劳动量，直接编写出需要的应用程序。

这个开发工具以 C/C++ 语言为目标语言，可以直接编译出机器代码，因此这个开发工具可以适合 OS 的开发。

#### Borland Delphi

这是 Borland 公司的一款 RAD 开发工具，这款工具以面向对象的 Pascal 语言为目标语言，可以直接编译出机器代码。

这款开发工具不但提供了一体化的代码编辑、调试、运行等基础程序设计功能，而且也提供了一个很优秀的框架 VCL，应用程序员基于 VCL，可以很容易的编译出自己想要的应用程序。另外，这款工具的编译部件经过专门优化，生成的代码（机器码）的运行效率并不比 C 编译器差，而且这款工具支持的 Object Oriented Pascal 语言，对原有的 Pascal 语言进行了很大的扩展，可以支持面向对象机制，也可以支持内嵌汇编功能。

总体来说，这款工具适合 OS 的开发，而且按照 Borland 的宣称，VCL 是一种跨平台的框架，这就说明，有可能把 VCL 进行改良，直接应用于 OS 的开发。如果这样的举措成为可能，那么 OS 的开发效率将会大大提高。

## **JBuilder 等 Java 开发工具**

Java 开发工具不会直接生成机器代码，而是生成一种中间字节码，这种字节码需要在虚拟机（一个软件模块或进程）的支持下运行。因此，不能直接使用 Java 作为开发语言来开发 OS，但 Java 的出现，给应用程序的移植带来了很大的方便。比如，新开发一个 OS，然后在这个 OS 上开发一个 Java 语言虚拟机，那么从理论上说，这个 OS 就可以支持所有的 Java 程序了。

## **Borland C++**

Borland 公司推出的 C++ 开发工具曾经风靡一时，而且 Borland 系列 C++ 开发工具可以产生比 Microsoft C++ 开发工具效率高得多的机器代码，但由于历史原因，目前 Borland 的 C++ 开发工具应用得已经不是很广泛了。

从理论上说，Borland 的 C++ 开发工具十分适合于 OS 的开发，甚至比 VC 更适合。

## **Windows RAD 工具总结**

总体来说，能够胜任 OS 开发的 Windows RAD 开发工具很多，但目前最常用的有下列这些：

- 1、Microsoft Visual C++，6.0 以上版本最佳；
- 2、Borland Delphi，2.0 版本以上均可；
- 3、Borland C++，3.0 版本以上均可。

在本文中，我们以 Microsoft Visual C++ 为例，讲解如何搭建一个 OS 开发环境。基于其它 RAD 开发工具的 OS 开发环境，跟 VC 类似，不过需要根据具体的编译工具进行适当变化。

## **Windows RAD 工具不能直接用于 OS 开发的原因**

Windows RAD 开发工具之所以不能直接应用于 OS 的开发，主要原因就是，这些 RAD 工具与 Windows 绑定得太紧密，生成的最终目标文件只能在 Windows 的支持下运行，无法直接用于其它的非 Windows 操作系统。

通过分析这些开发工具与 Windows 之间的紧密绑定关系，可以寻找合适的策略，来解除这种绑定，这样就可以采用这些 RAD 工具开发我们自己的操作系统了。

### 缺省情况下，生成的目标文件的入口地址固定

一般情况下，Windows RAD 开发工具都是以 WinMain 或 main 函数为入口的，应用程序模块被 OS 加载以后，会直接跳转到这个入口点开始执行。如果编译器严格按照这种规则指定入口点，那么对于 OS 的开发是合适的，但是通常情况下，编译器却不是这样做的，而是在调用 WinMain 或 main 函数前，先调用其它的一些初始化函数，比如 C 运行库的初始化函数、C++ 对象的构造函数等，等这些初始化工作准备好了，再调用 WinMain 或 main 函数。这样就不适合于我们的 OS 开发了，因为 OS 的开发过程中，需要严格的知道每个模块的入口点是什么，这样才能控制程序的行为。

为了解决这个问题，可以通过编译工具提供的编译选项，来手工指定模块的入口点。在后面的叙述中，我们以 Microsoft Visual C++ 为例，来说明如何改变模块的缺省入口地址。

### 缺省情况下，生成的目标文件的加载地址固定

Windows 的 RAD 开发工具，生成的目标文件一般是基于 PE 格式的可执行文件，或基于 DLL 的动态链接库，这两种格式的文件，在连接的时候，都指定了一个缺省的加载地址，比如，基于 PE 格式的可执行文件，缺省加载地址是 4M。由于 Windows 操作系统使用了虚拟内存技术，每个进程都独占 4Gbyte 的线形空间，因此 PE 格式的文件，缺省加载地址不论是多少，操作系统在加载这些 PE 模块的时候，都可以不做任何修改的加载到指定的地址。而我们的操作系统开发过程中，对每个模块的加载地址，都是有严格限制的，比如，一个模块，在我们自己开发的 OS 中，加载地址应该是 1M，而这个模块如果按照 RAD 工具的缺省设置，则可能按照 4M 加载地址进行连接，这样就会产生问题。

为了进一步说明这个问题，举一个简单的例子，下面是一段简单的 C 语言代码：

```
unsigned long ulOsVersion = 0;
```

```
BOOL InitializeVersion()
```

```
{  
    ulOsVersion = 5;  
    return TRUE;  
}
```

如果按照 4M 加载地址，翻译成汇编语言以后，是如下格式：

```
push ebp
mov ebp,esp
mov dword ptr [0x00400000],5
mov eax,0xFFFFFFFF
leave
ret
```

可以看出，对 `ulOsVersion` 的一个赋值操作，引用的地址是 4M（假设编译器把全局变量 `ulOsVersion` 放到了开始处）。

而如果我们的操作系统要求，这段代码所在的模块需要被加载到 1M 开始处，那么对 `ulOsVersion` 的引用，应该是下面的样子：

```
mov dword ptr [0x00100000],5
```

可以看出，与编译器缺省情况下的不一致，这在实际的系统中，是无法正常工作的。

为了解决这个问题，也可以通过设置编译器的编译连接选项，来消除这个矛盾。在本文的后面，将以 Microsoft Visual C++ 为例，来说明如何消除这种矛盾。

## RAD 生成的目标文件，增加了一个特定文件头

Windows 平台下的 RAD 开发工具，根据特定的目标，增加了一个特定的文件头，比如，针对 PE 格式的文件（可执行文件或 DLL），编译工具增加了一个 PE 头，而这个头长度是可变的，在这个头中的特定偏移处指定了这个模块的入口地址。Windows 加载器在加载这些模块的时候，根据 PE 头来找到入口地址，然后跳转到入口地址去执行。

而在我们的 OS 开发中，如果再进行这样的处理，可能就比较麻烦，有的情况下，甚至是不可能的，我们 OS 开发的要求是，直接定位到模块的入口地址，通过一条跳转指令直接跳转到该地址开始执行。

为了解决这个问题，我们可以通过对目标文件进行修改的方式来解决。比如，单独形成一个工具软件，这个工具软件读入目标文件的头，找出目标文件的入口所在位置，然后在文件的开始处加上一条跳转指令，跳转到入口处即可。这样处理之后，在我们的 OS 程序中，只要直接跳转到这个模块的开始处执行即可。

为了进一步说明这个过程，考虑如下的示例：

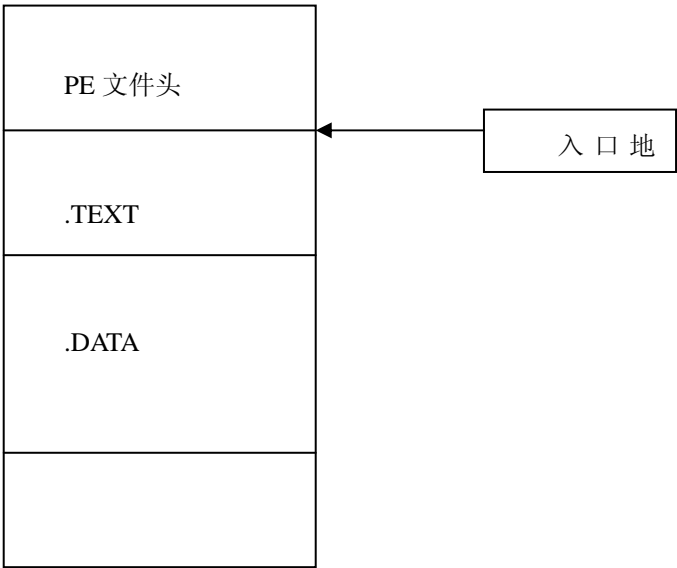


图 附 1-1 一个二进制目标模块

这是一个目标模块的示例，其中文件头的长度是可变的，在文件头中的一个字段中，指明了入口地址的位置（相对于文件头的偏移量）。

在我们的 OS 开发中，理想的目标是，直接跳转到入口地址处开始执行。但由于文件头长度是可变的无法确定入口地址的位置，因此，在这种情况下，我们可以通过软件的手段，把目标文件的开头 8 个字节修改成下列形式：

```
90 90 90 e9 xx xx
```

上述几个字节，对应的汇编代码就是：

```
nop
nop
nop
```

jmp xx xx

其中 xx xx 就是入口地址的偏移量（准确的说，应该是目标地址的偏移量减 8，因为文件头距离 jmp 后一条指令的偏移是 8），入下所示：

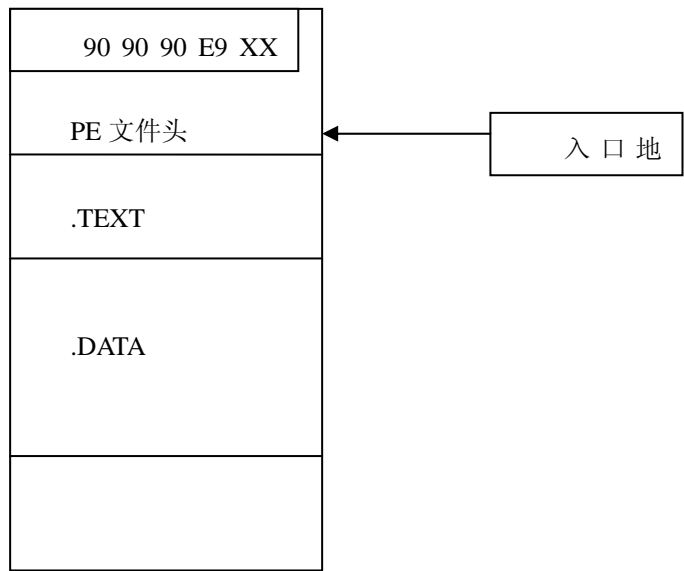


图 附 1-2      处理后的二进制可执行模块

这样，在我们的代码中，把这个目标模块直接加载到内存，然后跳转到模块的第一个字节执行即可。由于模块的开头部分被我们修改，因此会间接的跳转到入口地址处开始执行。

在本文中，我们给出如何修改 PE 文件头的代码，并附加一个修改工具。

目标模块加载到内存后，需要经过处理才能运行

Windows 下的 RAD 开发工具，一般生成 PE 格式的目标文件，而 PE 格式的目标文件，是按照节来组织的，比如，对模块中的代码，组织到 TEXT 节中，对于初始化的全局变量，组织到 DATA 节中，对于只读变量，组织到 RDATA 节中，按照节组织好以后，

然后节与节联合起来，就组成了整个文件，在 PE 文件头中，附加了一个节描述结构，用来描述每个节的位置、大小等属性。

问题的关键在于，在磁盘上存储目标模块的时候，节与节之间的间隔一般很小，比如按 16 字节对齐，但加载到内存之后，节与节之间却以 4K 为边界对齐。这样就产生了一个问题：把文件直接读入内存，是无法直接运行的，需要根据 PE 文件头，来适当的调整节与节之间在内存中的对其关系，然后才能投入运行。

在 OS 的开发当中，我们要求文件在内存中的映象，应该跟在磁盘上的映象一致，这样只要把磁盘上的文件加载到内存即可，不需要经过处理。尤其是在操作系统开发初期，还没有一个成型的加载器的情况下。为了避免这个矛盾，可以通过编译器选项来告诉编译器修改这种默认行为。

在 Microsoft Visual C++ 中，提供了一个连接选项/align，这个选项告诉编译器，节与节之间的对齐方式（在内存中的对齐方式），我们把这个选项设置为 16 (/ALIGN 16)，就可以把内存中的对齐方式，跟硬盘中的对齐方式协调一致，方便目标模块的直接运行。

另外，对于源程序中出现的未初始化的全局变量，连接程序把它们组织在.BSS 节中，而这个节在存储设备中，是不分配空间的，只有当加载到内存之后，才根据 PE 头中的信息，为这个节分配空间。比如，下面是一个 PE 格式的文件，在磁盘中的存储格式和在内存中的映象关系：

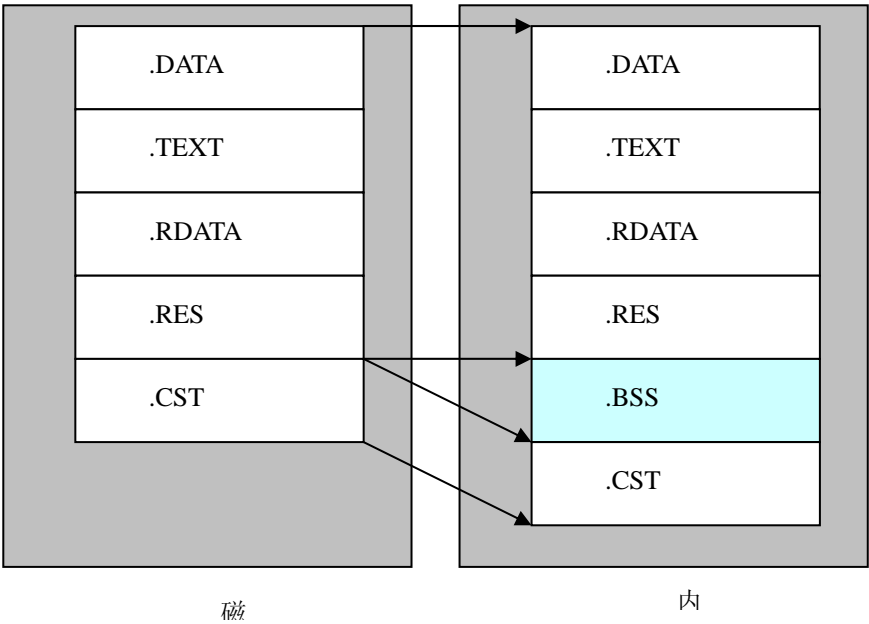


图 附 1-3 PE 文件的存储布局和内存布局

可以看出，在内存中，比在磁盘介质中的格式，多出一个节（.BSS）。  
在我们的 OS 开发中，尤其是开发初期，没有一个完善的加载器的情况下，是无法处理这种情况的，为了避免这种情况的出现，建议在程序编码的时候，对于全局变量，都进行初始化处理，这样就可以避免.BSS 节的出现。

总之，由于以上原因，Windows 下的 RAD 开发工具不适合直接用于 OS 的开发，但经过一些相应的对应策略，对上述限制进行修正之后，却很容易的应用于 OS 的开发。

## RAD 开发工具开发 OS 注意事项

RAD 开发工具是建立在 Windows 操作系统之上的，为 Windows 操作系统开发应用程序的工具，因此，很大程度上的一些功能、特性的实现，都是以 Windows 操作系统为

基础的，Windows 操作系统为这些功能和特性大的实现，提供了支持代码。

在我们自己开发操作系统的时候，却不能借助于 Windows 操作系统提供的功能，因为我们的操作系统是一个独立的 OS，直接使用硬件提供的功能，不能使用更底层的软件功能（除了 BIOS）。因此，在使用 RAD 开发 OS 的时候，下列事项需要格外注意：

## 避免调用任何 Windows 操作系统提供的系统调用

由于系统调用调用的是 Windows 的实现代码，这些代码在我们开发的 OS 中没有实现，或者实现的机制也跟 Windows 不一样，因此，在使用 RAD 工具编写 OS 代码的时候，一定不要调用 Windows 提供的系统调用（API）。

这意味着 RAD 提供的基于 Windows 的编程框架也不能使用（比如 VCL、MFC 等），因为这些框架在实现的时候，也调用了 Windows 的 API 函数。

但是一些跟 Windows 独立的框架或类库，比如 STL 等，是可以直接调用的。

## 避免任何 C 运行期函数调用

诸如 printf、malloc 等 C 运行期函数，也不能直接调用，因为这些函数也是调用 Windows API 来实现的。

当然，如果实现这些函数的代码是开源的，那么可以通过借鉴这些代码的一部分实现，来实现自己的 C 运行期函数，这样就可以随便调用了。

## 避免使用 C 或 C++提供的异常处理机制

C 或 C++的异常机制，不同的 RAD 开发工具有不同的实现方式，比如，在 Microsoft 的编译器中，就是采用结构化异常处理（SHE）来实现了 C++的异常处理，而结构化异常处理，则需要操作系统的支持，因此，在我们新开发的 OS 中，避免调用这些异常机制。

在 Microsoft Visual C++中，下列关键字应该避免使用：

```
__try  
__except  
__finally  
try  
throw  
catch
```

## 使用 Microsoft Visual C++ 搭建一个 OS 开发环境

Microsoft Visual C++ 是 Microsoft 公司开发的一个快速开发环境，该开发环境以 C/C++ 语言为目标语言，提供了集成的编辑、编译、调试、运行、开发项目管理等功能。使用这个开发工具编译连接的目标文件，是最终可以直接执行的机器代码，而且，这个开发工具提供了众多的编译和连接选项，而且提供了内置的函数修饰、编译修饰等选择项，十分适合于 OS 映像文件的开发。

在本文中，我们以 Microsoft Visual C++ 为开发环境，Microsoft Windows XP 为操作系统平台，建立一个操作系统开发环境。

首先，简单介绍一些在 OS 开发中，用到的 VC 特性。

### 操作系统开发中常用的 Microsoft Visual C++ 特性

在操作系统的开发中，下列 VC 特性经常用到：

#### 内嵌汇编代码

在操作系统开发中，在 C 或 C++ 语言中，嵌入汇编代码直接操作硬件，是一件非常频繁的事情，比如，进程或线程的切换，端口输入，各类系统表格的建立等，都需要使用内嵌的汇编代码来完成，因此，一个编译环境，如果不支持内嵌汇编代码支持，则不适合于 OS 的开发。

Microsoft Visual C++ 实现了对内嵌汇编代码的支持，在 C 源代码中，可以使用 `__asm` 关键字，来嵌入汇编代码，比如：

```
__asm mov eax,dword ptr [ebp + 0x08]
__asm mov ebx,dword ptr [eax]
__asm push dword ptr [ebx]
```

在每个汇编语句的前面，都需要增加关键字 `__asm`。

在汇编语句比较多的情况下，可以使用一个汇编语言块的格式，把这些汇编语句组织起来：

```

__asm{
    fld qword ptr [ebp + 0x08]
    fld qword ptr [ebp + 0x0C]
    fadd
    fstp qword ptr [ebp + 0x10]
}

```

一般情况下，内嵌的汇编代码一般用于操作底层的硬件，以及用于 CPU 硬件表格的初始化等工作。

### **\_\_declspec(naked)函数修饰**

缺省情况下，Microsoft Visual C++编译器对函数进行编译的时候，根据实际需要，插入一些辅助的汇编代码，比如，下面这个 C 语言函数：

```

unsigned long GetMax(unsigned long ulFirst,unsigned long ulSecond)
{
    return ulFirst > ulSecond ? ulFirst : ulSecond;
}

```

编译器编译成汇编代码后，可能会是下面的样子：

```

push ebp
mov ebp,esp
mov eax,dword ptr [ebp + 0x08]
cmp eax,dword ptr [ebp + 0x0C]
ja __END
mov eax,dword ptr [ebp + 0x0C]
leave
ret
__END:
leave
ret

```

其中，黑色标记部分代码是编译器自动插入的，插入这样的代码后，堆栈框架就发生了变化，而且有的时候变得难以预料。而在 OS 的开发中，有时候则要求每个函数在被调用的时候，堆栈框架保持一定的结构，比如线程切换函数。这个时候，就需要通过一种特定的机制，来告诉编译器不要生成额外的代码，而 `__declspec(naked)` 可以达到这个目的。

当使用 `__declspec(naked)` 对上述函数修饰后，编译器将不增加任何代码，这样所有的控制，都需要由程序设计者使用汇编语言完成。

有的时候，使用 `__declspec(naked)` 修饰函数是必须的，在 OS 的开发中，下列情况下需要这种修饰：

- 1、系统调用涉及到线程切换的时候；
- 2、中断处理函数；
- 3、在操作硬件系统表格，比如 IA32 的 GDT/LDT/IDT 等的时候。

## 如何搭建一个 OS 映象文件开发环境

在充分理解上述叙述的情况下，通过下列步骤，可以很容易的搭建一个 OS 映象文件开发环境：

### 创建一个 Windows DLL 工程

一般情况下，VC 可以生成 PE 格式的可执行文件、DLL 文件等文件类型，但可执行文件不太适合 OS 映象，因为编译器在编译的时候，自动在映象文件中加入一些其它代码，比如 C 运行期库的初始化代码等，这样会导致映象文件的体积变大。而 DLL 格式的文件则不会有这个问题，因此，建议从 DLL 开始，来建立 OS 映象。

在 Microsoft Visual C++ 中，按照如下向导创建一个 DLL 工程：

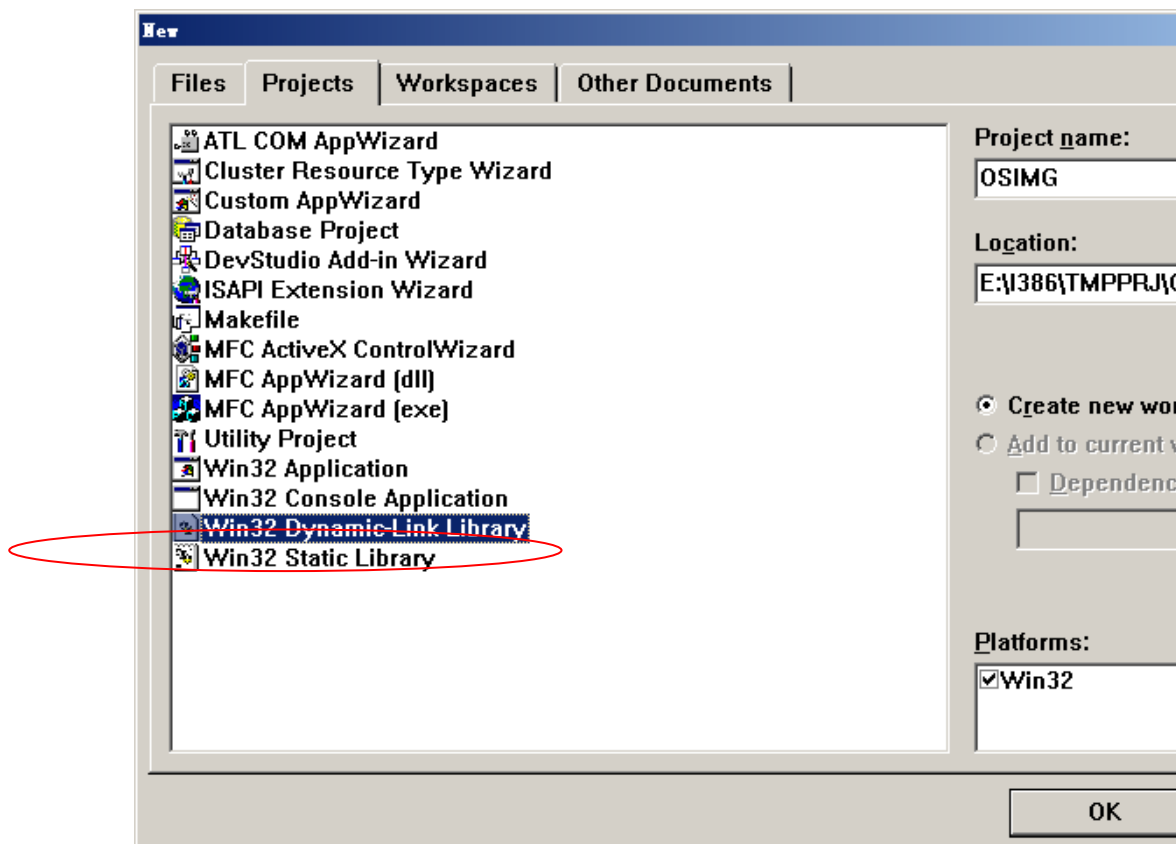


图 附 1-4 创建一个动态链接库工程

### 设置项目编译与连接选项

一般情况下，需要对创建的工程设定如下编译连接选项：

- 1、对齐方式，在项目选项中，添加/ALIGN:XXXX 选项，告诉连接器，如何处理目标文件映象在内存中的对齐方式，一般情况下，需要设置为跟目标文件在磁盘存储时的对齐方式一致，根据经验，设置为 16，一般是可以正常工作的；
- 2、设置基址选项，修改默认情况下的加载地址，比如，目标文件在我们自己的操作系统中，从 0x00100000（1M）处开始加载，则在连接工程选项里面添加 /BASE:0x00100000 选项；
- 3、设置入口地址，一般情况下，如果不设置入口地址，编译器会选择缺省的函数作为入口，比如，针对可执行文件，是 WinMain 或 main，针对动态链接库，是 DllMain，或 EntryPoint，等等，采用缺省的入口地址，有时候不能正确的控制映象文件的行为，

而且还可能导致映像文件尺寸变大，因为编译器可能在映像文件中插入了一些其它的代码。因此，建议手工设置入口地址，比如，假设我们的操作系统映像的入口地址是 `__init` 函数，则需要设定如下选项：`/entry:?__init@@YAXXZ`，其中，`?__init@@YAXXZ` 是 `__init` 函数被处理后的内部标号，因为 Visual C++ 采用了 C++ 的名字处理模式，而 C++ 支持重载机制，因此编译器可能把原始的函数名进行变换，变换成内部唯一的标号表示形式，关于如何确定一个函数的内部标号表示，请参考本文的附录。

上述所有的设置，请参考下图：

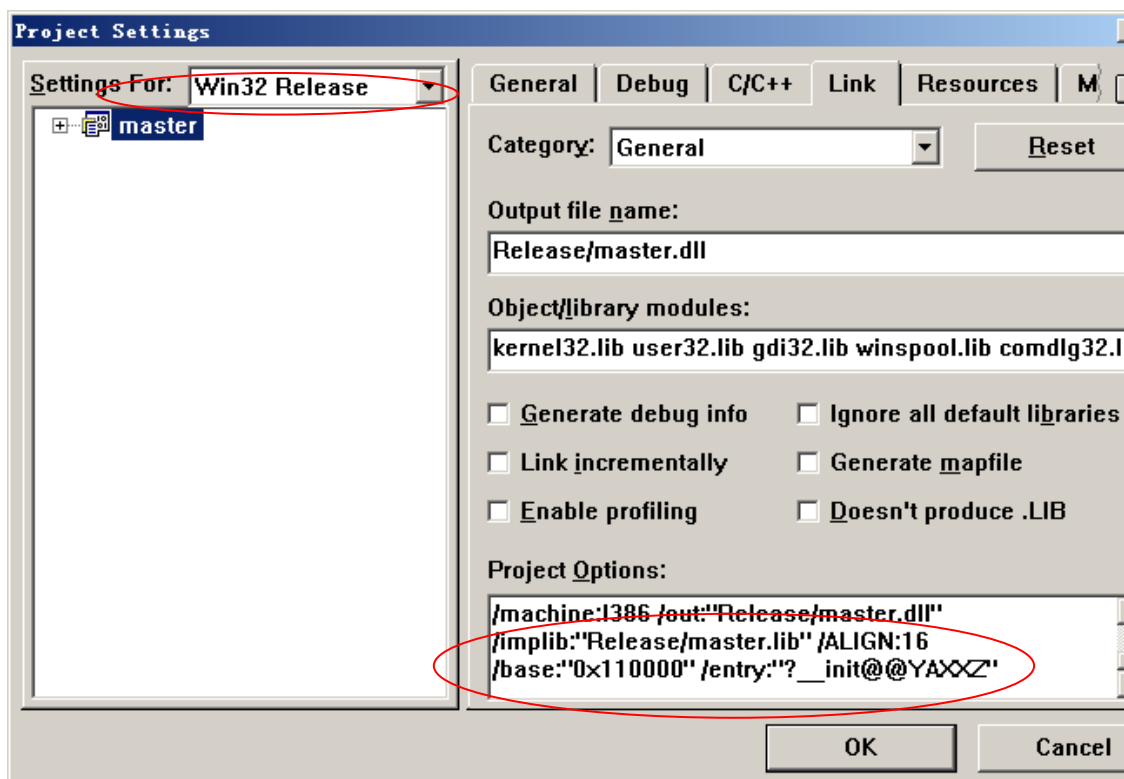


图 附 1-5 设置编译选项

在 Visual C++ 6.0 中，上述对话框可以从“project->setting...”打开，需要注意的是，打开的时候，是针对 DEBUG 版本设定的，请一定选择 Release 版本进行设定（图中红色圆圈中注明的地方）。

### 对目标文件进行处理

完成上述设置，编译连接好目标文件以后，就可以对目标文件进行了。

下面的代码是一个控制台程序的源代码，在 VC 环境下，创建一个控制台程序，把下列代码拷贝到源代码文件中，编译，连接后，就可以直接对目标文件进行了。需要注意的是，在编译的时候，需要根据实际情况，修改目标文件（待处理文件）所在的路径和文件名，这个程序运行以后，目标文件将被改写（**如果想保留目标文件，一定要在运行该程序前，对目标文件进行备份**）。

```
#include "stdio.h"
#include "windows.h"
#include "stdlib.h"

DWORD g_dwFileHeader[1024] = {0};    //The PE file's header will be read
into this buffer.

typedef struct __tagFILE_HEADER{
    unsigned char ucNop[4];
    DWORD        dwJumpAddr;
}__FILL_HEADER;

__FILL_HEADER    g_FillHeader    =    {0x90,0x90,0x90,0xe9,0x00000000};
//This structure will be

//written to target file.

char*    g_lpszTargetPath    =    "d:\\ospath\\osimg\\release\\osimg.dll";    //Target
file's path and name.

void main()
{
    IMAGE_DOS_HEADER*        ImageDosHeader = NULL;
    IMAGE_NT_HEADERS*        ImageNtHeader = NULL;
```

```
    IMAGE_OPTIONAL_HEADER* ImageOptionalHeader = NULL;
    HANDLE                  hFile = INVALID_HANDLE_VALUE;
    DWORD                  dwReadBytes = 0L;
    BOOL                   bResult = FALSE;
    DWORD                  dwActualBytes = 0L;
    DWORD                  dwOffset = 0L;
    UCHAR*                 lpucSource = NULL;
    UCHAR*                 lpucDes    = NULL;
    DWORD                  dwLoop     = 0;

    hFile = CreateFile(                                //Open the target file.
        g_lpszTargetPath,
        GENERIC_READ | GENERIC_WRITE,
        0L,
        NULL,
        OPEN_ALWAYS,
        0L,
        NULL);
    if(INVALID_HANDLE_VALUE == hFile)
    {
        printf("Can not open the target file to read.");
        goto __TERMINAL;
    }

    dwReadBytes = 4096;                                //Read 4k bytes from target file.
    bResult =
    ReadFile(hFile,g_dwFileHeader,dwReadBytes,&dwActualBytes,NULL);
    if(!bResult)
        goto __TERMINAL;

    CloseHandle(hFile);
    hFile = INVALID_HANDLE_VALUE;

    //
```

```

//The following code locates the entry point of the PE file,and modifies it.
//
ImageDosHeader = (IMAGE_DOS_HEADER*)&g_dwFileHeader[0];
dwOffset = ImageDosHeader->e_lfanew;

ImageNtHeader
=
(IMAGE_NT_HEADERS*)((UCHAR*)&g_dwFileHeader[0] + dwOffset);
ImageOptionalHeader = &(ImageNtHeader->OptionalHeader);

g_FillHeader.dwJumpAddr = ImageOptionalHeader->AddressOfEntryPoint;
printf("                Entry        Point        :
%d\r\n",ImageOptionalHeader->AddressOfEntryPoint);
g_FillHeader.dwJumpAddr -= sizeof(__FILL_HEADER);    //Calculate the
target address will

//jump to.
//Because we
have added some nop instruc-
//tions in front
of the target file,so
//we must
adjust it.

lpucSource = (UCHAR*)&g_FillHeader.ucNop[0];
lpucDes     = (UCHAR*)&g_dwFileHeader[0];

for(dwLoop = 0;dwLoop < sizeof(__FILL_HEADER);dwLoop ++) //Modify
the target file's header.
{
    *lpucDes = *lpucSource;
    lpucDes ++;
    lpucSource ++;
}

hFile = CreateFile(                //Open the target file to write.

```

```
        g_lpszTargetPath,
        GENERIC_READ | GENERIC_WRITE,
        0L,
        NULL,
        OPEN_ALWAYS,
        0L,
        NULL);
if(INVALID_HANDLE_VALUE == hFile)
{
    printf("Can not open the target file to write.");
    goto __TERMINAL;
}

WriteFile(hFile,(LPVOID)&g_dwFileHeader[0],sizeof(__FILL_HEADER),&dwActualBytes,
        NULL);

printf("SectionAlignent : %d\r\n",ImageOptionalHeader->SectionAlignent);
printf("    FileAlignent : %d\r\n",ImageOptionalHeader->FileAlignent);

__TERMINAL:
if(INVALID_HANDLE_VALUE != hFile)
    CloseHandle(hFile);
}
```

## 映象文件的加载与运行

通过上述步骤（编译、连接、处理等），可以最终得到一个可执行的 OS 映象，只要把这个 OS 映象加载到内存中，跳转到该映象的开始处，就可以运行了。但最后一个问题是，如何把操作系统映象加载到内存中？

这个问题解决起来，相对复杂一些，需要编写一个引导程序，还需要另外一个工具，创建一个引导磁盘，这些工作，可能是开发操作系统的门槛，一般情况下，完成这些初始化的准备工作，需要一个礼拜，甚至更多的时间，而且需要开发者详细了解汇编语言、BIOS 等 PC 计算机的底层协议和标准。

为了给操作系统开发者提供便利，笔者编写了一个简单的引导程序，并附带了一个引导磁盘创建工具，操作系统开发者可以直接把使用 VC 开发的操作系统映象，跟这些引导代码集成到一张磁盘上，直接引导计算机并运行，无需自己开发引导程序。

笔者创建的这个引导程序和磁盘创建工具，由四个文件组成：

- 1、bootsect.bin，这是一个引导扇区的代码，512byte 大小；
- 2、realinit.bin，这是一个实模式下初始化代码，4K 大小，这个映象文件的任务是，初始化键盘、显示系统、定时器等底层的硬件，并激活 A20 地址线；
- 3、miniker.bin，在这个操作系统映象中，集成了一个键盘驱动程序和一个字符模式显示驱动程序，读者可以利用这个系统映象提供的功能，也可以不用；
- 4、fmtldrf.com，这是一个引导磁盘创建工具，这个工具把 bootsect.bin、realinit.bin、miniker.bin 以及读者创建的操作系统映象（名字一定为 master.bin）写到软盘中，创建一张引导盘。

因此，读者只要把自己创建的操作系统映象（更名为 master.bin）以及上述四个文件，拷贝到同一个目录下，在 DOS 窗口下运行 fmtldrf.com 程序，就可以创建一张引导盘（需要一张新格式化的软盘），使用这个新创建的引导盘，引导 PC 机，就可以运行自己的操作系统了。

需要说明的是，上述几个文件只为读者提供了硬件初始化功能和引导功能，没有提供任何的操作系统功能，所有跟操作系统相关的功能（比如，进程管理、存储管理、文件/资源管理以及设备管理等）都需要读者在自己的操作系统映象中完成。

## 简单 OS 开发示例

在这一部分中，我们按照上面介绍的方法，创建一个简单的操作系统映象。这个操作系统映象非常简单，引导计算机后，清屏，然后在屏幕的顶端打印出“ABCDE.....WXY”，然后进入死循环。

### 创建一个名字为 OSIMG\_1 的 DLL 工程

如下图所示：

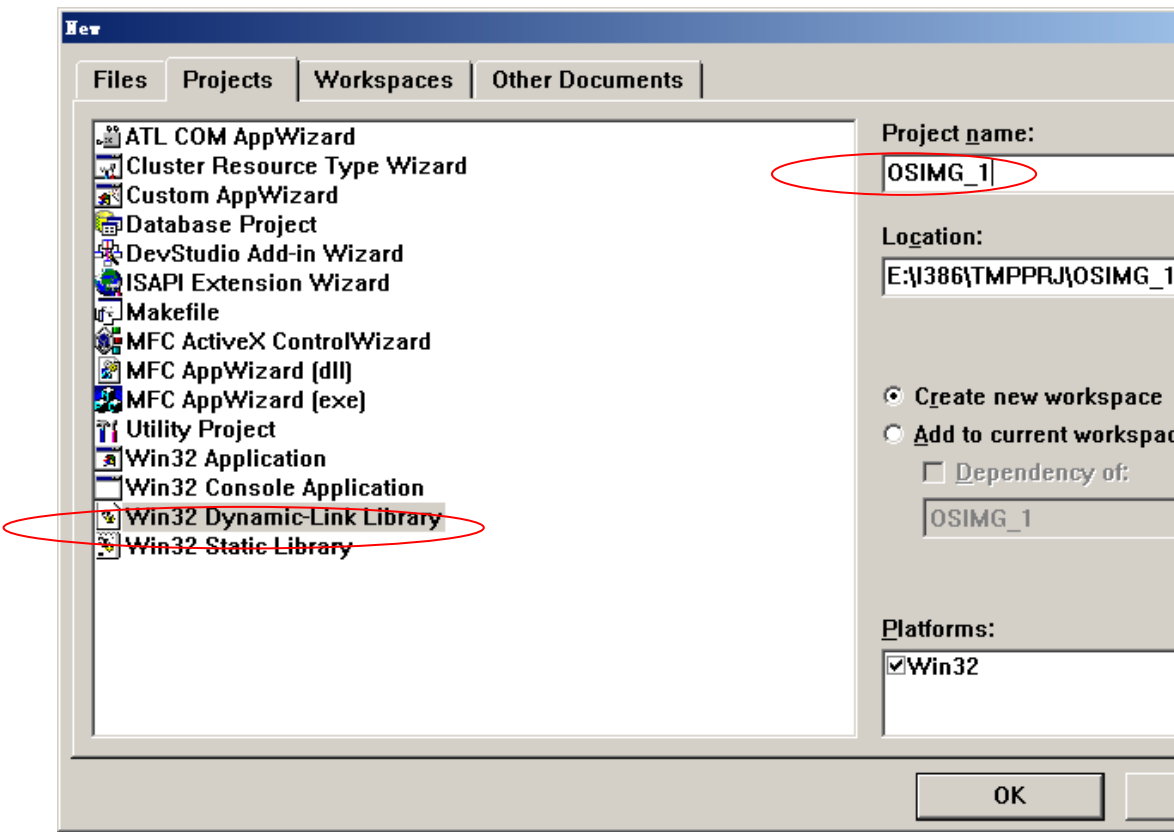


图 附 1-6 创建一个新的工程

添加一个源程序文件，并编辑实现代码

入下图所示：

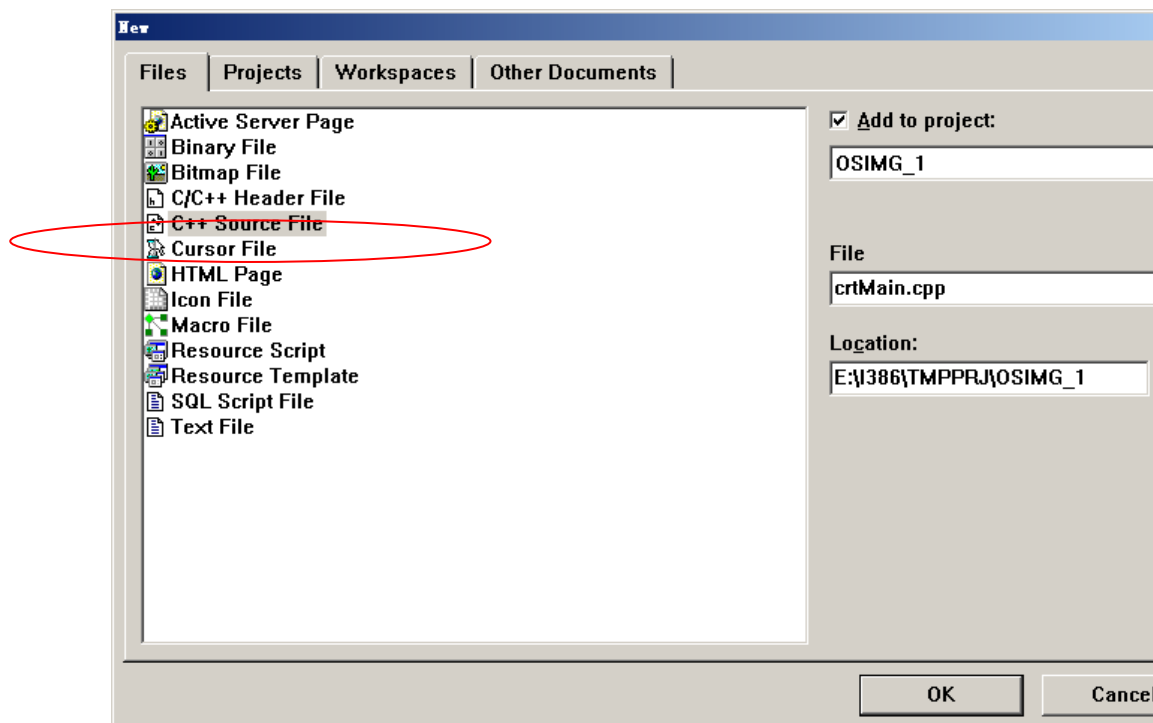


图 附 1-7 在工程中添加一个源代码文件

在增加的程序文件中，键入一下操作系统实现代码：

```
void ClearScreen()
{
    unsigned long ulBase = 0xb8000;
    unsigned long i      = 0;

    while(i < 80*25)
    {
        *(char*)ulBase = ' ';
        ulBase ++;
        *(char*)ulBase = 0x07;
```

```
        ulBase++;
        i++;
    }
}

void __init()
{
    char          uc          = 'A';
    unsigned long ulVgaBase = 0xb8000;

    ClearScreen();    //Clear screen.

    for(uc = 'A';uc < 'Z';uc++)
    {
        *(char*)ulVgaBase = uc;
        ulVgaBase++;
        *(char*)ulVgaBase = 0x07;
        ulVgaBase++;
    }

    while(1)    //Dead loop.
    {
    }
}

void main()
{
}
```

这段代码的功能是，调用 `ClearScreen` 函数，清屏，然后打印出“ABCD.....WXY”，完毕后，进入死循环。

设置编译连接选项，并进行编译连接

根据上面的叙述，设置下列编译和连接选项：

- 1、入口点，设置为“?\_\_init@YAXXZ”（\_\_init 函数编译后的内部标号）；
- 2、设置加载地址：/BASE:0x00110000，即该程序断从 1M+64K 开始加载，并运行（笔者提供的引导程序，把操作系统映象加载到该地址，并运行，这在大多数情况下是满足要求的）；
- 3、设置对齐选项：/ALIGN:16，这样可以导致文件内的节对齐方式，和内存中的节对齐方式一致。

请参考下图：

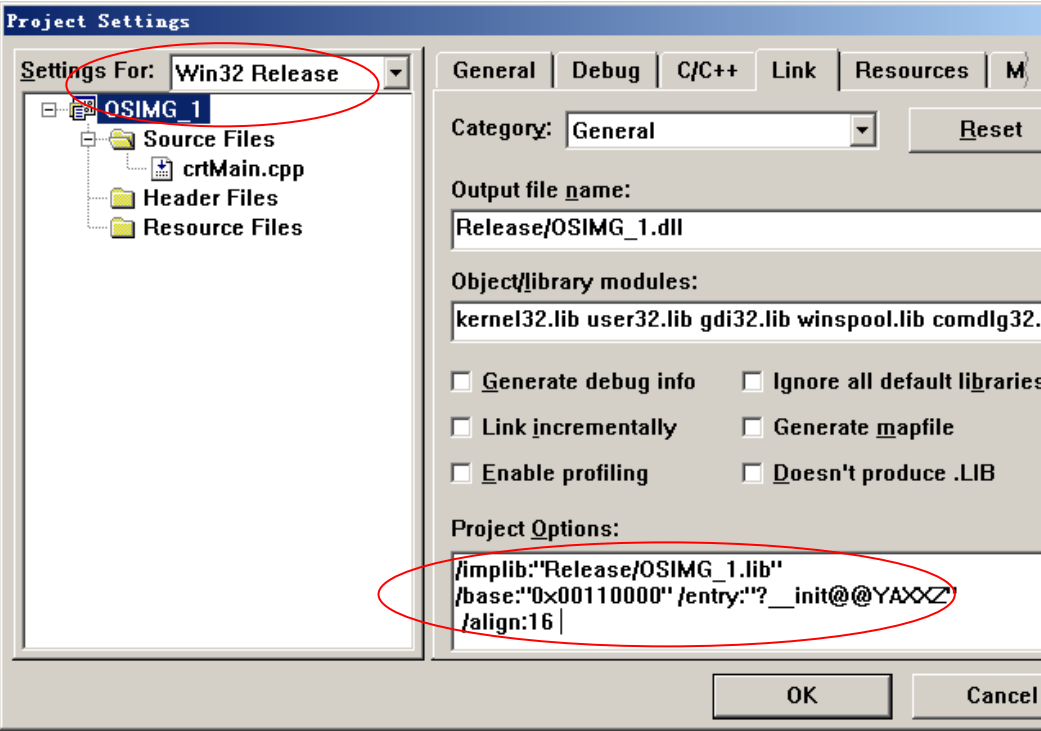


图 附 1-8 设置工程的编译连接选项

设置完后，选择“build->Batch Build...”菜单，出现下列对话框：

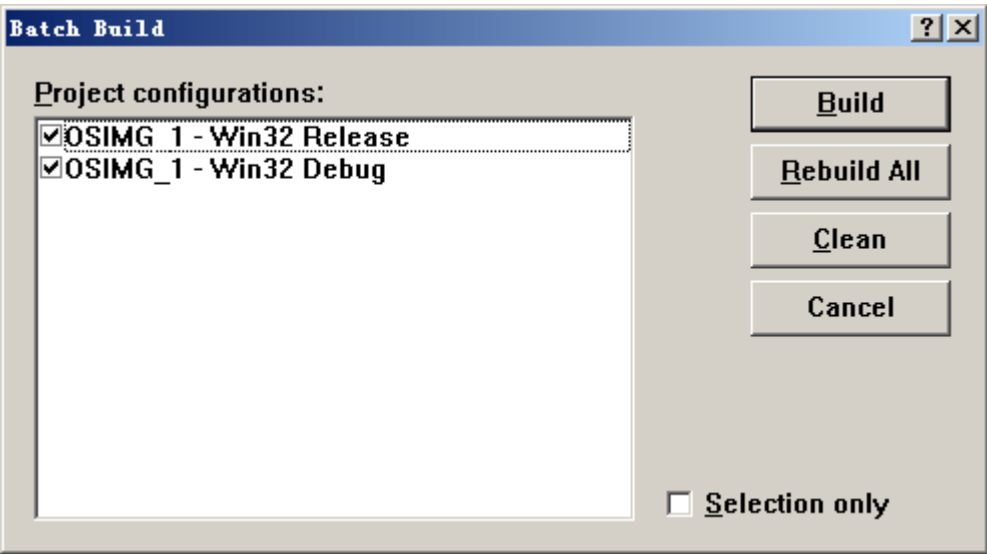


图 附 1-9 选择 Release 模式对工程进行编译

点击“Rebuild All”，编译这 DLL 工程。

**处理目标文件**

另外创建一个控制台项目，把本文中附带的代码拷贝到新建项目中，并根据目标文件路径（Release 版本所在的路径）修改其中的变量（目标文件路径和文件名），编译，运行，这样就可以把新创建的 DLL 文件进行处理，适合直接加载并运行了。

**创建引导磁盘**

经过上述步骤之后，读者就已经创建了自己的操作系统映象了，剩下的任务，就是创建引导磁盘了，把下列四个文件连同读者创建的操作系统映象文件，拷贝到同一个目录下，然后把读者创建的操作系统映象改名为 master.bin，如下：



图 附 1-10 准备创建一张引导盘

其中，master.bin 就是读者自己编译的操作系统映象文件（经过处理的 DLL 文件，改名之后的形式）。

然后，插入一张新格式化的软盘，并在该目录下，运行 fmtldr.com，便可创建一张引导盘，所有这些完成以后，就可以使用新创建的引导盘引导计算机系统了。

如果最终的结果是，屏幕最上方显示“ABCDEFGHIJKLMNOPQRSTUVWXYZ”，那么，恭喜您，您已经成功创建了一个自己的最简单的操作系统内核了。需要说明的是，创建这个简单的 OS 核心，读者没有编写一行汇编代码！

## 如何快速掌握汇编语言

由于各种原因，许多程序员，甚至是很资深的程序员，对汇编语言都有一种“望而生畏”的感觉，觉得汇编语言高深莫测，不好掌握，在平时的工作中，对于一些最底层的硬件操作，宁愿牺牲一些效率，使用高级语言（比如，C）完成，也不愿使用汇编语言。

造成这种现象的根本原因，就是因为汇编语言的应用不是很广泛，操作系统和编译程序为程序员提供了足够的抽象，把一些底层的硬件操作封装起来，提供高层语言接口。这样程序员在大部分情况下，不用汇编语言，就可以完成汇编语言完成的功能，对汇编语言的依赖不是很强烈，因此对汇编语言也逐渐疏远。

其实，在各种程序设计语言中，汇编语言应该是最简单的，没有任何编程思想，也没有任何条条框框，只要一条指令一条指令的堆积起来就可以了，不想其它编程语言一样，有很深刻和严密的编程思想，比如，C++/Java 等面向对象的语言，需要很深入的理解，并有很长的实践经验，才能很好的掌握并使用。

虽然汇编语言相对简单，应用比较少，但有些情况下的功能实现，必须由汇编语言来完成，其它语言是无法替代的，比如，在编写操作系统的时候，就需要由汇编语言来完成硬件初始化、任务切换等功能，还有一些情况下，比如多媒体程序设计、网络程序设计等，有时候也必须使用汇编语言。因此，在目前情况下，掌握汇编语言是一项很有必要，并且很有意义的事情。

在这部分中，我们首先分析一下掌握汇编语言可以带来的好处，然后分析一下学习汇编语言的难点，最后给出一些简单的试验方法，使用这些试验方法可以快速的掌握汇编语言。

### 掌握汇编语言的益处

掌握汇编语言，有下列好处：

#### 可以充分利用在操作系统等系统软件开发中

在操作系统的开发中，引导代码、硬件初始化和检测代码、设备驱动程序、线程/任务切换、CPU 初始化等任务，都需要汇编语言来完成，掌握汇编语言，是成功编写出操作系统的最最必要的条件。

另外，掌握汇编语言，可以充分理解操作系统实现中的一些机制，比如系统调用、任务切换等，对于操作系统学习有很大帮助。

### 可以充分理解高级语言特性

如果不掌握汇编语言，对于一些高级语言提供的特性，可能无法真正理解，比如，针对 C++ 提供的异常处理机制，如果按照通常的函数调用/执行/返回的流程理解，是无法理解清楚的，但如果有汇编语言的知识，对函数的调用/返回等机制很熟悉，并熟悉堆栈框架结构，那么就很容易理解这些异常处理机制，从而能够更好的利用这些机制。

掌握汇编语言，可以充分理解高级语言的一些其它特性的在实现，从而编写出更高效的程序代码。比如，针对 C 语言的 case 语句，一般认为需要进行一系列比较，代价很大，但实际情况下，如果能很好的控制 case 变量，使得 case 变量足够接近，那么执行效率会大大提高，跟 case 语句的多少没有关系。

### 可以帮助程序员进行程序调试，寻找程序中的 bug

掌握好汇编语言后，对于程序的调试，就可以在汇编级进行，这样更容易理清楚程序的执行线路和步骤，对于一些使用普通调试方法无法捕获的错误，使用汇编语言级的调试可能更有效果。

总之，掌握好汇编语言的益处有很多很多，这些仅仅是一些典型的示例而已。

## 熟练掌握汇编语言的困难所在

根据个人经验，掌握汇编语言有下列困难：

### 汇编语言没有建立起方便的输入/输出方法

不像 C/C++ 等高级语言一样，汇编语言没有建立起方便的输入/输出方法，也就是没有方便的输入/输出函数调用。在 C 中，一个很方便的函数 printf 可以用来进行输出，另外一个函数 scanf 可以用来输入，而且这两个函数非常容易使用，因而使用 C 语言编程，其输入和输出问题很容易解决，入门比较容易。

在汇编语言中就不是这样了，对于所有的输入/输出引用，都是使用操作系统调用或 BIOS 调用的方式来实现的，每个 I/O 调用，需要很复杂的输入参数（比如，设置各种寄存器），而且这些参数都是数字，晦涩难懂，不易使用，从而制约了汇编语言的学习。

## 汇编语言编译/连接复杂，开发工具难以使用

汇编语言的开发工具，相对 C/C++ 语言的集成开发工具，也非常难以使用，一般情况下，汇编语言的开发工具都是基于命令行格式的，在编辑完汇编代码以后，需要对源文件进行保存，然后引用汇编器对源代码进行编译，再执行连接命令，对编译结果进行连接。这样非常复杂，不易使用。而高级语言提供的集成开发环境，则只要一个菜单选项，就可以把所有这一切搞定，非常容易。

## 一种快速掌握汇编语言的方法

一般的 C/C++ 语言集成开发环境，都内置了内嵌汇编语言的支持功能，比如，在 Microsoft Visual C++ 中，使用关键字 `_asm` 可以嵌入汇编语言，因此，我们可以充分使用这些集成开发环境的汇编语言嵌入功能，来打开学习汇编语言的大门。

使用集成开发环境的内嵌汇编功能，可以彻底解决汇编语言学习过程中的两大难题（I/O 和编译连接），为汇编语言的学习，铺设了一条相对平坦的道路，可以起到入门的目的。当然，随着汇编语言学习的深入，在充分掌握一定数量的指令，并熟悉汇编语言编程结构以后，还是建议采用专用的汇编语言编译工具进行系统、专业的学习。

本文我们使用 Visual C++ 为例，介绍一下如何在代码中嵌入汇编语言，来学习汇编语言的使用。

首先创建一个 console 工程（控制台工程，控制台工程可以充分使用 C 语言运行期库），添加一个源文件，然后对这个源文件进行编辑，在实际的代码中添加汇编代码。

比如，可以使用汇编语言计算两个整数的和，然后把计算结果打印出来：

```
#include <stdio.h>

unsigned long Add(unsigned long ulNum1,unsigned long ulNum2)
{
    __asm{
        push ebx
        mov ebx,dword ptr [ebp + 0x08]
        move ax,dword ptr [ebp + 0x0C]
        add eax,ebx
    }
```

```

        pop ebx
    }
}

void main()
{
    printf("%d\r\n",Add(100,200));
}

```

在这个示例中，函数 `Add` 的实现就是用汇编代码实现的，首先，保存实现中使用的一个寄存器（一般情况下，如果编程过程中修改了寄存器的值，建议首先保存该寄存器）`ebx`，然后把函数的两个参数加载到 `ebx` 和 `eax` 中（参数保留在堆栈中，使用 `ebp` 可以引用堆栈框架），把这两个寄存器相加，最后恢复 `ebx`，需要注意的是，函数返回值是放在 `eax` 寄存器里面的（如果是浮点返回值，则放在 `st(0)` 寄存器里），因此我们的汇编代码不需要返回。

读者可能奇快，为什么我们直接使用 `ebp` 来引用堆栈框架，而没有做任何初始化，其实，编译器已经为我们做好了，如果把上述 `Add` 函数翻译成汇编语言，将是下面的样子：

```

__Add:
    push ebp
    mov ebp,esp
    push ebx
    mov ebx,dword ptr [ebp + 0x08]
    mov eax,dword ptr [ebp + 0x0C]
    add eax,ebx
    pop ebx
    leave
    ret

```

其中，黑色标出部分，是编译器为我们自动添加上去的，以便于引用堆栈。

如果我们使用 `__declspec(naked)` 对 `Add` 函数进行限定，则编译器将不添加任何代码。

读者可以在这个示例的基础上，进行进一步的修改，比如，可以计算两个整数的乘法，可以计算两个向量的加法，如果对浮点运算熟悉，则可以计算诸如浮点矩阵的乘法等复杂运算。比如，下面的代码，用于计算两个浮点向量的积：

```
__declspec(naked) double Vector_Mul(double* lpx,double* lpy,unsigned short
i)
{
    __asm{
        push ebp
        mov ebp,esp
        push ebx
        push ecx
        push edx
        push esi
        mov ebx,dword ptr [ebp + 0x08]    //Load the base address of x.
        mov edx,dword ptr [ebp + 0x0C]    //Load the base address of y.
        mov cx,word ptr [ebp + 0x10]      //Load i.
        movzx ecx,cx
        xor esi,esi
        fldz                               //Set st(0) to zero.
    __START:
        fld qword ptr [ebx + esi*8]        //Load the i-th element of the first
vector.
        fld qword ptr [edx + esi*8]        //Load the i-th element of the second
vector.
        fmul                               //Multiple the two elements.
        fadd
        inc esi                             //Adjust the index register.
        loop __START
        pop esi
        pop edx
        pop ecx
        pop ebx
        leave
    }
```

```
        ret  
    }  
}
```

从这个例子中也可以看出，由于我们使用了 `__declspec(naked)` 对该函数进行修饰，故编译器没有添加任何附加代码（参考 `Add` 函数的情况），所以 `ebp` 寄存器的初始化、堆栈框架的恢复（`leave` 指令）、函数的返回等，都需要我们手工添加进去。

等读者使用这种方便的方法，对一些常用指令熟悉，并掌握了一定的汇编语言编程技巧之后，便可以很容易的使用专业的汇编语言开发工具（比如，**TASM/MASM/NASM** 等）进行汇编程序的开发了。



# 附录 E 一种代码执行时间测量方法的实现

—— **An implementation for a mechanism used to  
measure code executing time**

## 概述

本文描述了一种衡量一段核心代码的精确执行时间的机制。在操作系统核心的实现中，精确的确定一段代码（比如，一个中断处理程序）的执行时间，是非常重要的，尤其是在实时操作系统中，这种需求尤为重要。

## 执行时间计算方法

采用普通的记录系统时钟的方式来粗略的计算执行时间，对于计算一个核心线程所占用的 CPU 资源很有意义，但由于系统时钟的粒度比较粗，一般是毫秒级，因此，对于执行时间停留在微妙级的代码，是没有意义的，这时候，必须采用其它的机制来完成。

一般情况下，CPU 可以提供特定的机制，来实现这个需求。比如，在 Intel 32 位 CPU 中，实时的记录了 CPU 的每个时钟周期（跟系统时钟不同，该 CPU 是在该时钟的驱动下工作的，而系统时钟则是一种外部定时机制），因此，可以通过读取 CPU 的时钟周期数，并取差值来记录一段代码的执行时间。这种机制十分精确，可以用于计算任意数量的指令的执行时间。比如，假设 CPU 的主频率（执行频率）是  $X$  MHz，为了衡量一段代码的执行时间，在代码开始执行的时候，记录 CPU 的时钟周期数（通过读取 CPU 内部寄存器获取），在代码执行结束的时候，再次读取 CPU 时钟周期数，并取前后的差值（假设为  $N$ ），这样可以按照如下方式，计算这段代码的执行时间：

$$Et = N / (X * 1000 * 1000)$$

比如，在一个主频率是 166MHz 的 CPU 中，一段代码的执行周期数是 10000，则该段代码的执行时间为： $Et = 10000 / (166 * 1000 * 1000) = 0.06ms$ 。

## 实现方式

### 对外接口

在当前版本的 Hello China 实现中，定义了如下数据结构，来记录一段代码的执行时间：

```
BEGIN_DEFINE_OBJECT(__PERF_RECORDER)
    __U64                u64Start;
    __U64                u64End;
    __U64                u64Result;
    __U64                u64Max;
END_DEFINE_OBJECT()
```

并以全局函数的方式，定义了下列接口函数，供需要计算执行时间的代码调用，完成代码执行时间的记录：

```
VOID PerfBeginRecord(__PERF_RECORDER*);
VOID PerfEndRecord(__PERF_RECORDER*);
VOID PerfCommit(__PERF_RECORDER*);
```

其中，第一个函数记录下当前的 CPU 时钟周期数，然后进入待计算的代码，在待计算代码的后面，调用 `PerfEndRecord` 函数，该函数记录下当前 CPU 的时钟周期数，这样就获得了待测试代码执行前及执行后的 CPU 时钟周期。

而 `PerfCommit` 函数则用来计算当前先后两次测试结果的差值，保存在 `u64Result` 中，并跟 `u64Max` 比较，如果大于 `u64Max`，则重新设置 `u64Max` 的值为 `u64Result`，这样就可以保存测试过程中，出现的最大测试结果。

需要注意的是，`__PERF_RECORDER` 对象保存的是 CPU 时钟周期数，而不是具体的时间，因此，测试结果理论上是跟 CPU 的主频率无关的（但实际上，由于不同频率的 CPU，其指令实现、指令周期可能略有不同，进而导致 CPU 时钟频率的差异，但在本文中，不考虑这种相对细微的误差）。若要获得具体的时间数值，则需要按照上面介绍的换算方式，根据测试的 CPU 时钟频率个数和 CPU 的主频率，得到具体的时间。

## IA32 硬件平台下的实现

在 Hello China 的当前版本中，只做了 IA32 构架下的实现。IA32 硬件平台提供了一个指令，RDTSC，用来读取 CPU 的时钟周期数，该指令执行后，CPU 的时钟周期数就会被记录在 EDX : EAX 两个寄存器里面，通过读取这两个寄存器的值，就可以得到当前 CPU 的时钟周期。因此，PerfBeginRecord 函数的实现如下：

```
VOID PerfBeginRecord(__PERF_RECORDER* lpPr)
{
    __U64*          lpStart = NULL;

    if(NULL == lpPr) //Parameter check.
        Return;
    lpStart = &lpPr->u64Start;

#ifdef __I386
    __asm{
        push eax
        push ebx
        push edx
        mov ebx,lpStart
        rdtsc                //Read time stamp counter.
        mov dword ptr [ebx], eax    //Save low part
        mov dword ptr [ebx + 4],edx //Save high part.
        pop edx
        pop ebx
        pop  eax
    }
#else
#endif
}
```

在上述实现中，使用 RDTSC 指令，读取当前 CPU 的时钟周期，并保存在 \_\_PERF\_RECORDER 变量的 u64Start 部分中。

对于 PerfEndRecord 的实现，与 PerfBeginRecord 类似：

```

VOID PerfEndRecord(__PERF_RECORDER* lpPr)
{
    __U64*          lpEnd = NULL;

    if(NULL == lpPr) //Parameter check.
        Return;
    lpStart = &lpPr->u64End;

#ifdef __I386
    __asm{
        push eax
        push ebx
        push edx
        mov ebx,lpStart
        rdtsc                //Read time stamp counter.
        mov dword ptr [ebx], eax    //Save low part
        mov dword ptr [ebx + 4],edx //Save high part.
        pop edx
        pop ebx
        pop  eax
    }
#else
#endif
}

```

## 误差分析

需要注意的是，在上述实现中，会引入误差，因为 CPU 的当前时钟周期数，在 rdtsc 指令被执行的时候就已经被读取，但在该指令执行完成后，还执行了保存时钟周期数、

弹出使用的寄存器、调用函数返回等相关的指令操作，然后才进入被测量代码，因此，在 `rdtsc` 指令执行后，到实际的待测量代码执行前，这段时间也被进行了计算，假设这段时间为  $T_1$ 。

另，当调用 `PerfEndRecord` 函数，以记录测试代码执行完毕后的时钟周期数的时候，也额外执行了一些指令，那就是从 `PerfEndRecord` 函数被调用开始（可能是一条 `push ebp` 指令）的指令，到 `rdtsc` 指令之间被执行的指令。假设这些指令执行时间为  $T_2$ 。因此，实际测量过程中，实际上引入了  $T_1 + T_2$  时间的误差。

假设测量时间为  $T$ ，则实际的代码执行时间应该为：

$$T_r = T - T_1 - T_2$$

一般情况下， $T_1$  占用的时间，维持在 50 — 100 个 CPU 时钟周期内， $T_2$  占用的时间，维持在 80 — 100 个时钟周期内，因此，在实际测试代码执行时间很大（大于 10000 个 CPU 周期）的情况下，这些误差可以忽略不计，但在实际测试代码执行周期很小的情况下，则上述原因引入的误差，就不能不考虑。这种情况下，可以取  $T_1$  为 80（时钟周期数，需要换算成实际执行时间）， $T_2$  为 90，然后把实际测试的时间  $T$ ，减去  $T_1$  和  $T_2$ ，来大致的修正测试结果。

## Hello China 系统时钟中断测试结果

通过上述方法，对当前版本的 Hello China 操作系统的时钟中断处理程序性能进行了测试，测试平台是一台 Intel P3 处理器，主频率是 2.26GHz（实际上，测试结果是 CPU 时钟周期数，与 CPU 的工作频率关系不大，但跟 CPU 的型号有关），测试结果如下：

一次定时器个数	CPU 周期数	所需时间（ms）
0	965	0.0004
1	2093	0.0009
4	4689	0.0020
16	14769	0.0062
22	20669	0.0087
32	32425	0.0137
50	54302	0.0229
64	75299	0.0318

100	145615	0.0614
128	216260	0.0913
144	263238	0.1111
256	720209	0.3039
296	946675	0.3995
512	4087064	1.7247
1024	20878308	8.8102
2048	90241108	38.0799

表 附 2-1 一个测试结构

为了把时钟中断处理时间限制在 0.5ms 以内，因此要求在 2.26G 以上的 CPU 上，建议设定的一次定时器个数，不要超过 512 个。

实际上，当前版本的 Hello China 中，处理一次定时器是最耗时的操作，因为在当前版本中，对于一次定时器，需要在时钟中断上下文中完成下列事情：

- 1、调用定时器设定线程的一个定时器函数（如果定时器设定线程指定了一个函数），或者给定时器设定线程发送一个定时器消息；
- 2、删除一次定时器对象，这包括删除一次定时器对象所占用的内存等。

而对于同样处于时钟中断上下文中被处理的多次定时器和线程切换，相对一次定时器来说，消耗的 CPU 资源相对少一些，因此，上述测试结果是一个比较保守的测试结果，在实际中，Hello China 的表现会优于上述结果。

另，当前版本的 Hello China，一般建议运行在 Intel 256M 以上频率的 CPU 上，假设目标 CPU 是 256M，那么同样上述测试，花费的时间大致估计如下（不考虑指令集的差异）：

一次定时器个数	CPU 周期数	所需时间（ms）
0	965	0.0040
1	2093	0.0090
4	4689	0.0200
16	14769	0.0620
22	20669	0.0870

32	32425	0.1370
50	54302	0.2290
64	75299	0.3180
100	145615	0.6140
128	216260	0.9130
144	263238	1.1110
256	720209	3.0390
296	946675	3.9950
512	4087064	17.2470
1024	20878308	88.1020
2048	90241108	380.7990

表 附 2-2    一个测试结构

可以看出，如果是在 256MHz 主频率的 Intel CPU 上，如果要限定时钟中断处理时间不大于 0.5ms，则需要限制一次定时器个数不能大雨 64 个（保守估计），如果要限制时钟中断处理时间不大于 1ms，则需要限制一次定时器个数不能大于 128 个。

## 附录 F 64bit 整型数据类型的实现

—— The implementation of 64 signed/unsigned  
bit integer data type

## 概述

目前系统版本是 32 位的，缺省情况下，最大的整型数据长度是 32bit，这在一些特殊的场合，可能不能满足需要，比如，在记录系统时钟的时候，或者在文件系统的实现中。这个时候，需要一种取值范围更大的数据类型，来满足这种要求，\_\_U64 和 \_\_I64 就是 Hello China 定义并实现的两种数据类型。

\_\_U64 是 64bit 的无符号整数类型，其最大取值可以为 0xFFFFFFFFFFFFFFFF，而 \_\_I64 则是 64bit 的有符号整型数，其最大取值则可以为 0x8FFFFFFFFFFFFFFFFF。在当前版本的实现中，对于上述两种数据类型，定义分别如下：

```
BEGIN_DEFINE_OBJECT(__U64)
DWORD                dwLowPart;
DWORD                dwHighPart;
END_DEFINE_OBJECT()
```

```
BEGIN_DEFINE_OBJECT(__I64)
DWORD                dwLowPart;
DWORD                dwHighPart;
END_DEFINE_OBJECT()
```

在当前版本的实现中，以函数调用（内联函数）的形式，实现了下列运算：

## 加法

加法实现的定义如下：

```
VOID u64Add(__U64*,__U64*,__U64*);
VOID i64Add(__I64*,__I64*,__I64*);
```

其中，第一和第二个参数是加数，上述函数把第一个操作数和第二个操作数相加，结果存放在第三个操作数内。

具体的实现，是平台相关的，在本文中，将描述在 Intel 32 位 CPU 硬件平台下，相关函数的实现。

## \_\_U64 的加法

在 IA32 平台下，实现 64bit 的加法，使用了 IA32 平台的一个重要指令：ADC。该指令可以完成进位加法，比如：

ADC dest,src //dest 和 src 是两个 32bit 的操作数，可以是寄存器，也可以是一个寄存器和一个内存引用。

则执行的效果为：

`dest = dest + src + carry`

其中，carry 为 EFLAGS 寄存器的 C 比特（进位比特）。

这种情况下，对于\_\_U64 的实现，相关代码为：

```
VOID u64Add(__U64* lpu64_1,__U64* lpu64_2,__U64* lpu64_result)
{
#ifdef __I386__          //The implementation for Intel's IA32 platform.
    __asm{
        push eax
        push ebx
        push ecx
        push edx
        mov eax,lpu64_1
        mov ebx,lpu64_2
        mov ecx,lpu64_result
        mov edx,dword ptr [eax]    //Load low part of first operand to edx.
        add edx,dword ptr [ebx]    //Add low part of two operands together.
        mov dword ptr [ecx],edx    //Save low part to result
        mov edx,dword ptr [eax + 4] //Load high part of first operand to edx
        adc edx,dword ptr [ebx + 4] //Add high part and carry bit together.
        mov dword ptr [ecx + 4],edx //Save high part to result.
        pop edx
        pop ecx
        pop ebx
        pop eax
    }
}
```

```
#else  
#endif  
}
```

可以看出，上述实现中，对于 64bit 整数的低 32bit 的加法，按照正常加法进行，如果产生进位，则会设置 carry 比特，对于高 32bit 的加法，采用 ADC 指令进行，该指令把进位标志、两个加数的高 32 位加在一起，然后存储在结果数据的高 32 位部分。

## \_\_I64 的加法

\_\_I64 加法的实现，与 \_\_U64 类似。

## 减法

64bit 长整型的加法运算，采用下列函数定义：

```
VOID u64Sub(__U64* lpu64_1,__U64* lpu64_2,__U64* lpu64_result);
```

```
VOID i64Sub(__I64* __I64*,__I64*,__I64*);
```

其中，第一个操作数是被减数，第二个操作数是减数，第三个操作数存储运算结果，即：

```
lpu64_result = lpu64_1 - lpu64_2
```

## \_\_U64 的减法

对于 64 位无符号整数的减法运算，使用了 IA32 平台的 SBB 指令。与 ADC 指令类似，该指令在执行减法的时候，减去减数的同时，也减去 carry 比特。比如，如下指令：

```
SBB dest,src
```

执行过程为：

```
dest = dest - src - carry
```

其中，carry 是 EFLAGS 标志寄存器里面的进位标志（C 比特）。

在当前版本的实现中，无符号 64 位整数的减法实现如下：

```

VOID u64Sub(__U64* lpu64_1,__U64* lpu64_2,__U64* lpu64_result)
{
#ifdef __I386__
    __asm{
        push eax
        push ebx
        push ecx
        push edx
        mov eax,lpu64_1
        mov ebx,lpu64_2
        mov ecx,lpu64_3
        mov edx,dword ptr [eax]
        sub edx,dword ptr [ebx]
        mov dword ptr [ecx],edx    //Save low part.
    mov edx,dword ptr [eax + 4]
        sbb edx,dword ptr [ebx + 4]
        mov dword ptr [ecx],edx    //Save high part.
        pop edx
        pop ecx
        pop ebx
        pop eax
    }
#else
#endif
}

```

可以看出，在这个实现中，对于低 32 位部分，直接采用 SUB 指令进行减法运算，对于高 32 位部分，则采用 SBB 指令完成计算。

## \_\_I64 的减法

\_\_I64 的减法运算，与 \_\_U64 类似。

## 比较

64bit 整型数比较运算接口如下:

```
BOOL EqualTo(__U64* lpu64_1,__U64* lpu64_2); // lpu64_1 == lpu64_2.  
BOOL EqualTo(__I64* lpi64_1,__I64* lpi64_2);
```

```
BOOL LessThan(__U64* lpu64_1,__U64* lpu64_2); //lpu64_1 < lpu64_2.  
BOOL LessThan(__I64* lpi64_1,__I64* lpi64_2);
```

```
BOOL MoreThan(__U64* lpu64_1,__U64* lpu64_2); //lpu64_1 > lpu64_2.  
BOOL MoreThan(__I64* lpi64_1,__I64* lpi64_2);
```

```
BOOL LessEqual(__U64* lpu64_1,__U64* lpu64_2); //lpu64_1 <= lpu64_2  
BOOL LessEqual(__I64* lpi64_1,__I64* lpi64_2);
```

```
BOOL MoreEqual(__U64* lpu64_1,__U64* lpu64_2); //lpu64_1 >= lpu64_2  
BOOL MoreEqual(__I64* lpi64_1,__I64* lpi64_2);
```

按照当前的定义, 对于 64 比特长度整型数的比较, 不需要采用汇编语言实现, 而可以直接采用 C 语言实现。下面给出了有符号 64 位整数的比较和无符号 64 位整数的比较的相关实现。

### \_\_U64 的比较

对于 64bit 无符号整数的比较, 只需要比较其 dwLowPart 和 dwHighPart 就可以了。比如, 对于 LessThan 函数, 实现可以如下:

```
BOOL u64LessThan(__U64* lpu64_1,__U64* lpu64_2)  
{  
  
    return (lpu64_1->dwHighPart < lpu64_2->dwHighPart) ||  
           (lpu64_1->dwHighPart == lpu64_2->dwHighPart) &&  
           (lpu64_1->dwLowPart < lpu64_2->dwLowPart);  
}
```

即在比较的时候，一旦 `dwHighPart` 小于对方，则说明整个整数小于对方，于是直接返回 `TRUE`，或者如果 `dwHighPart` 相等，`dwLowPart` 小于对方，也可以满足比较条件。

对于相等操作，其充分必要条件是 `dwHighPart` 和 `dwLowPart` 分别相等。

## \_\_I64 的比较

`__I64` 的比较操作，与 `__U64` 相同。

## 移位

对于 64bit 整数的移位操作，定义下列接口（函数）：

```
VOID u64RotateLeft(__U64* lpu64_1,DWORD dwTimes);
```

```
VOID i64RotateLeft(__I64* lpi64_1,DWORD dwTimes);
```

```
VOID u64RotateRight(__U64* lpu64_1,DWORD dwTimes);
```

```
VOID i64RotateRight(__I64* lpi64_1,DWORD dwTimes);
```

其中，第一组函数实现左移操作，`lpu64` 是操作数，而 `dwTimes` 是移动的位数。第二组函数实现右移操作。

需要注意的是，上述函数定义的都是逻辑移动，即被操作数的符号位会随其它位一起移动，不保留符号位。对于算术移动（保留符号位的移动），在后续版本中，根据需要实现。

需要注意的是，对于移位操作，可能会不正确的影响 `EFLAGS` 寄存器里的特定比特。当前版本下，移动一个 64bit 整数，分两步进行：第一步，移动 64 位整数的 `low part`，然后移动 `high part`，这样一旦在移动 `high part` 的时候，结果为 0，则会导致 `EFLAGS` 寄存器的 `Z` 比特设置为 1，虽然整个 64bit 整数的移位结果不为 0（因为 `low part` 移位后，可能不为 0）。因此，在进行 64bit 整数的移位后，如果判断移位结果是否为 0，则需要采取另外的方式（比如，采用 64 比特整数的比较函数），而不能直接判断。

## \_\_U64 的移位

在 IA32 平台的实现中，移位操作充分利用了该平台提供的几个移位指令。比如，对于左移操作，可以采用 `shl` 和 `rcl` 两个指令实现。其中，`shl` 指令可以把一个机器字长度的整数，向左方向移动若干位，其中，最高位被移动到 `carry` 比特（`EFLAGS` 寄存器），而 `rcl` 指令，则完成如下操作：

- 1、把 `carry` 比特移动到操作数的最低位（`bit 0`）；
- 2、把操作数向左移动一位；
- 3、把操作数的最高位移动到 `carry` 比特。

因此，联合使用 `shl` 和 `rcl`，可以实现 64 比特整数的移位操作。比如，64 位无符号整数的移位操作，可以实现如下：

```
VOID u64RotateLeft(__U64* lpu64_1,DWORD dwTimes)
{
#ifdef __I386__
    __asm{
        push eax
        push ebx
        push ecx
        mov eax,lpu64_1
        mov ecx,dwTimes
    __BEGIN:
        shl dword ptr [eax];    //Shift low part.
        rcl dword ptr [eax + 4]; //Shift high part,including carry bit.
        loop __BEGIN
        pop ecx
        pop ebx
        pop eax
    }
#else
#endif
}
```

对于向右方向的移动，则使用了 IA32 平台提供的 `shr` 和 `rcr` 指令，这两个指令的含

义，与 `shl` 和 `rc1` 类似。在实现 64bit 的向右移位中，需要从整数的高部分开始，实现如下：

```

VOID u64RotateRight(__U64* lpu64_1,DWORD dwTimes)
{
#ifdef __I386__
    __asm{
        push eax
        push ecx
        mov eax,lpu64_1
        mov ecx,dwTimes
    __BEGIN:
        shr dword ptr [eax + 4]    //Shift high part first.
        rcr dword ptr [eax]        //Shift left part then.
        loop __BEGIN
        pop ecx
        pop eax
    }
#else
#endif
}

```

## \_\_I64 的移位

\_\_I64 的移位方式，与 \_\_U64 类似。

## 后续支持

对于 64bit 长度整型数的乘法和除法运算、循环移位运算等，由于目前的实现还没有应用需求，因此，为了简便期间，没有实现这些运算，在未来需要的时候，再实现这些运算。或者如果应用程序编译环境支持内建的 64 位整数，则应用程序可以直接使用编译器提供的内嵌支持。

从技术上来说，实现 64 位整数的乘法和除法运算，与其加法和减法运算一样简单。

## 附录 G IOCTRL 控制程序使用介绍及 实例

——IOCTRL control application: usage and example

# 概述

IOCTRL 是 Hello China 提供的一个底层控制程序，用于完成端口的读写功能，对于使用内存映射方式工作的设备，也可以通过该程序，从设备映射到的内存空间，读取或写入数据。在进行计算机硬件驱动程序的编写过程中，该程序起到了很关键的调试作用，比如，在编写网络接口卡驱动程序的时候，首先通过该程序，熟悉了网络接口卡的寄存器结构，并通过该程序，通过手工的方式完成了网络接口卡的发送报文、接收报文处理。在编写 PCI 总线驱动程序的时候，通过该程序，采用手工的方式，完成了 PCI 总线设备的枚举、PCI 设备配置空间的读取和写入等工作，为 PCI 总线驱动程序的编写，铺平了道路。

因此，对于设备驱动程序开发者来说，在进行实际的设备驱动程序开发前，可以通过该软件，以手工的方式，熟悉相关硬件的寄存器组织、工作模式，对于深入理解硬件设备的工作方式，以及对于驱动程序的调试，将是十分有用的。

# 使用方式

该程序的使用方式十分简单，在命令行界面下，输入 `ioctl`，并回车，就进入该程序的控制界面，如下：

```
[system-view]ioctl
#
```

“#” 是该程序的提示符，可以输入 “help” 命令，来查看该程序提供的命令：

```
#help
inputb io_port           : Input a byte from IO port.
inputw io_port           : Input a word from IO port.
inputd io_port           : Input a dword from IO port.
inputsb io_port          : Input a byte string from IO port.
inputsw io_port          : Input a word string from IO port.
outputb io_port val      : Output a byte to IO port.
```

<code>outputw io_port val</code>	: Output a word to IO port.
<code>outputd io_port val</code>	: Output a dword to IO port.
<code>outputsb io_port size addr</code>	: Output a byte string to IO port.
<code>outputsw io_port size addr</code>	: Output a word string to IO port.
<code>memwb addr val</code>	: Write a byte to memory location.
<code>memww addr val</code>	: Write a word to memory location.
<code>memwd addr val</code>	: Write a dword to memory location.
<code>memrb addr</code>	: Read a byte from memory.
<code>memrw addr</code>	: Read a word from memory.
<code>memrd addr</code>	: Read a dword from memory.
<code>memalloc</code>	: Allocate a block of memory.
<code>memrels addr</code>	: Release the memory allocated by memalloc.
<code>help</code>	: Print out this screen.
<code>exit</code>	: Exit from this application.

输出的格式为:

命令字 参数 1 参数 2 参数 3 : 帮助信息

比如, 对于 `outputb` 命令, 含有两个参数: `io_port` 和 `val`, 其中 `io_port` 是输出数据的端口号, 而 `val` 则是输出的数值, 后面的帮助信息说明, 该命令用于向一个端口输出一个字节 (byte)。需要注意的是, 所有参数都是 16 进制的。

下面对几个常用命令的使用方式, 进行描述。

## Inputb

从计算机端口输入一个字节, 格式如下:

```
inputb io_port
```

其中, `io_port` 是目标端口号, 一个 2 字节的 16 进制数字。比如:

```
#inputb 60
000000ff
```

000000ff 是输出结果, 以 32 比特显示, 但是对于该命令, 只有最后一个字节有效。

## Inputw

从计算机端口输入两个字节，格式如下：

```
inputw io_port
```

其中，io\_port 是目标端口号，一个 2 字节的 16 进制数字。比如：

```
#inputb 60  
0000ffff
```

0000ffff 是输出结果，以 32 比特显示，但是对于该命令，只有最后两个字节有效。

## Inputd

从计算机端口输入四个字节（长字），格式如下：

```
inputd io_port
```

其中，io\_port 是目标端口号，一个 2 字节的 16 进制数字。比如：

```
#inputb 60  
ffffffff
```

ffffffff 是输出结果，以 32 比特显示。

## Outputb

向外部端口输出一个字节，格式如下：

```
outputb io_port val
```

其中，io\_port 是待输出的端口，val 是输出值，都是以 16 进制表示，其中，io\_port 是一个 2 字节的 16 进制数字。

比如：

```
#outputb 60 ff  
#
```

如果命令执行成功，则没有任何提示。

## Outputw

向外部端口输出两个字节，格式如下：

```
outputw io_port val
```

其中，io\_port 是待输出的端口，val 是输出值，都是以 16 进制表示，其中，io\_port 是一个 2 字节的 16 进制数字。

比如：

```
#outputd 60 ffff
```

```
#
```

如果命令执行成功，则没有任何提示。

## Outputd

向外部端口输出四个字节，格式如下：

```
outputd io_port val
```

其中，io\_port 是待输出的端口，val 是输出值，都是以 16 进制表示，其中，io\_port 是一个 2 字节的 16 进制数字。

比如：

```
#outputd 60 ffffffff
```

```
#
```

如果命令执行成功，则没有任何提示。

## Memwb

写一个字节到内存中，一般是设备映射内存，格式如下：

```
memwb mem_addr val
```

其中，mem\_addr 是待写入的内存地址，val 则是具体的数值。比如：

```
#memwb ec000000 ff
```

#

写入后，不作任何提示。需要注意的是，对应的内存地址，必须是已经经过预留的地址（通过 `VirtualAlloc` 调用），否则会产生内存异常。

## Memww

写两个字节到内存中，一般是设备映射内存，格式如下：

```
memww mem_addr val
```

其中，`mem_addr` 是待写入的内存地址，`val` 则是具体的数值。比如：

```
#memww ec000000 ffff
```

#

写入后，不作任何提示。需要注意的是，对应的内存地址，必须是已经经过预留的地址（通过 `VirtualAlloc` 调用），否则会产生内存异常。

## Memwd

写四个字节到内存中，一般是设备映射内存，格式如下：

```
memwd mem_addr val
```

其中，`mem_addr` 是待写入的内存地址，`val` 则是具体的数值。比如：

```
#memwd ec000000 ffffffff
```

#

写入后，不作任何提示。需要注意的是，对应的内存地址，必须是已经经过预留的地址（通过 `VirtualAlloc` 调用），否则会产生内存异常。

## Memrb

从内存地址读取一个字节数据，一般是设备映射内存，格式如下：

```
memrb mem_addr
```

其中，mem\_addr 是待读入数据的内存地址，比如：

```
#memrb ec000000  
000000ff
```

其中，000000ff 是读入的结果数据，以 32 比特显示，但对于该命令，只有最有一个字节有效。需要注意的是，对于内存地址，必须是经过预留的（通过 VirtualAlloc 函数调用）内存地址，即已经在 CPU 的页表中，有了对应的页表项，否则会产生内存访问异常。

## Memrw

从内存地址读取两个字节数据，一般是设备映射内存，格式如下：

```
memrw mem_addr
```

其中，mem\_addr 是待读入数据的内存地址，比如：

```
#memrw ec000000  
0000ffff
```

其中，0000ffff 是读入的结果数据，以 32 比特显示，但对于该命令，只有最有两个字节有效。需要注意的是，对于内存地址，必须是经过预留的（通过 VirtualAlloc 函数调用）内存地址，即已经在 CPU 的页表中，有了对应的页表项，否则会产生内存访问异常。

## Memrd

从内存地址读取四个字节数据，一般是设备映射内存，格式如下：

```
memrd mem_addr
```

其中，mem\_addr 是待读入数据的内存地址，比如：

```
#memrd ec000000
ffffff
```

其中，ffffff 是读入的结果数据，以 32 比特显示。需要注意的是，对于内存地址，必须是经过预留的（通过 VirtualAlloc 函数调用）内存地址，即已经在 CPU 的页表中，有了对应的页表项，否则会产生内存访问异常。

### Help 和 exit

Help 命令用于打印出帮助信息，即本节开始部分，exit 用于推出该程序，返回到命令行界面。

## 一个读取 PCI 设备配置空间的例子

在此，我们列举一个使用 IOCTL，读取基于 PCI 总线的硬件设备的配置空间的例子，来说明该工具的使用方法。

### PCI 配置空间读取方法

支持 PCI 总线的设备，必须实现一个 256 字节长度的配置空间，在该配置空间内，保存了设备相关的信息（比如，厂家标识号、设备标识号等），以及设备使用的系统资源信息（中断向量号、输入/输出端口号、内存映射地址等），这个配置空间，是实现对 PCI 总线设备进行管理、PCI 总线设备动态配置等的基础。

在基于 PC 构架的计算机上，可以通过 IO 指令来读取设备空间。一般情况下，按照下列步骤来完成配置空间的读取：

- 3、
- 首先向预留端口 CF8（十六进制），输出要读取的 PCI 设备标识信息（包括 PCI 总线号、设备号、功能号等），以及要读取的配置空间的偏移，所有这些信息，组织成一个 32 比特的长字，如下：

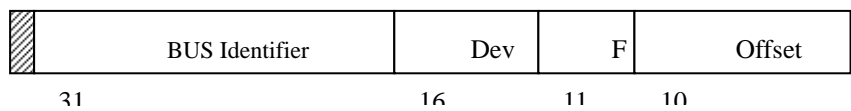


图 附 4-1 PCI 配置空间的访问命令字

其中，第 31 比特固定为 1，16 到 30 比特是总线标识符，11 到 15 比特则是设备号，8 到 10 比特是功能号（一个 PCI 设备，最多可以支持 8 种不同的功能），最后 8 比特则是要读取的配置空间数据的偏移。比如，一块 PCI 接口的网络接口卡，位于 PCI 总线 1 上，设备 ID 是 8，功能 ID 为 0（只有一种功能），要读取配置空间的第一个双字（4byte），则首先需要向 CF8 端口输出 80014000（十六进制）；

4、然后读取 CFC 端口，以双字的方式，就可以获得上述设置的配置空间内相应的内容。

可以看出，可以通过 Hello China 的 IOCTL 程序，来完成配置空间的读取（当然，直接使用汇编语言编程，也可以完成读取操作）。比如，要读取 PCI 总线 1，设备 8，功能 0 的配置空间中的第一个双字，则可以输入如下命令完成：

```
#outputd cf8 80014000
#inputd cfc
813910EC
```

inputd 命令后的输出，就是配置空间内，开始的第一个长字的值。

## PCI 配置空间布局

下面是 PCI 总线规范 2.2 修订版（PCI Local BUS Specification, Revision 2.2），定义的 PCI 设备配置空间的布局格式：

31		16 15		0
Device ID		Vendor ID		
Status		Command		
Class Code			Revision ID	
BIST	Header Type	Latency	Cache Line	
Base Address Register 1				
Base Address Register 2				
Base Address Register 3				
Base Address Register 4				
Base Address Register 5				
Base Address Register 6				
CardBus CIS Pointer				
System ID		Subsystem ID		
Expansion ROM base address				
			Capabilities	
Max Lat	Min Gnt	Interrupt Pin	Interrupt	

图 附 4-2 PCI 配置空间的布局

其中，阴影部分是保留区域（Reserved），各字段的具体含义，参考 PCI 标准规范文本（或者在 Hello China 设备管理框架相关文档中，也有部分描述），在此，重点介绍一下几个字段：

- 1、**Base Address Register 1—6**: 这六个 32 比特的字段, 保存了 PCI 设备使用的内存区域 (内存映射区域) 或 IO 端口区域, 每个 PCI 设备, 最多可有六个连续的内存或 IO 端口空间;
- 2、**Interrupt Pin**: 该 PCI 设备连接到了 PCI 总线的那条中断引脚上。PCI 总线有四条中断引脚 (INTA、INTB、INTC、INTD), 每个 PCI 设备可以连接到任何一条中断引脚上 (不过 PCI 规范建议, 对于多功能设备, 针对每种功能, 可以连接到上述四条中断引脚上, 对于单功能的 PCI 设备, 则强烈建议连接到 INTA 上);
- 3、**Interrupt Line**: 该设备被逻辑连接到中断控制器 (比如, Intel 8259 芯片) 的那条中断输入上。

在对设备进行初始化的时候, 需要为每个 PCI 设备, 分别分配上述资源, 一般情况下, 这种资源分配, 是系统 BIOS 完成的, 但为了可靠期间, 操作系统可用进行进一步的检查 (以确保资源没有冲突), 甚至可以重新分配资源。

## PCI 总线设备位置获取方法

对于安装在 PCI 总线上的设备 (支持 PCI 接口规范的设备), 其位置信息可以通过遍历 PCI 总线的方式获取 (参考相关的文档), 在 Hello China 的当前版本下, 也提供了一个程序—`pcilist`, 用来列举出系统中所有的 PCI 设备, 及其资源分配情况。

除此之外, 还有一种在 Windows 操作系统下, 获取 PCI 设备位置的方式, 即通过 Windows 操作系统携带的设备管理器。打开设备管理器, 可以枚举出系统中所有的硬件设备, 如下:

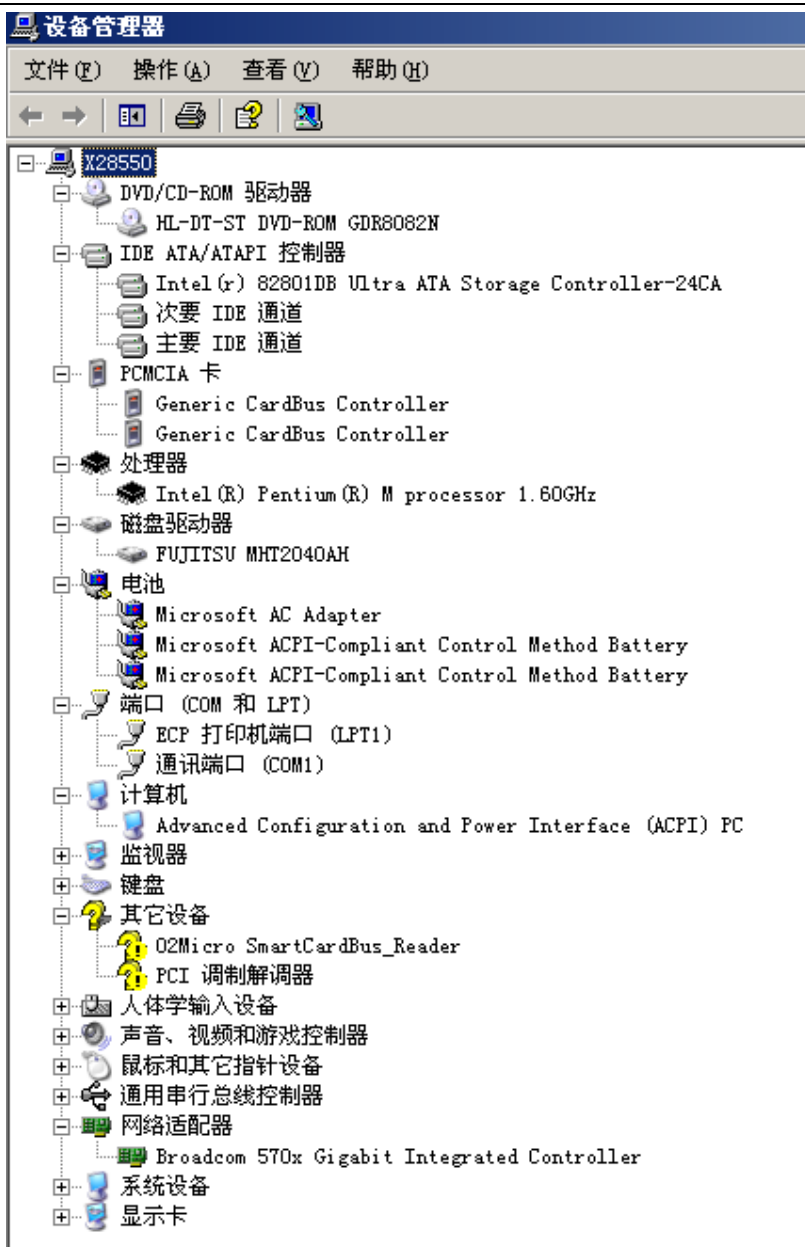


图 附 4-3 通过 Windows 的设备管理器查看设备信息

比如，要想获得网络适配器的 PCI 总线位置，可以双击图中“Broadcom 570x Gigabit Integrated Controller”，得到如下显示画面：

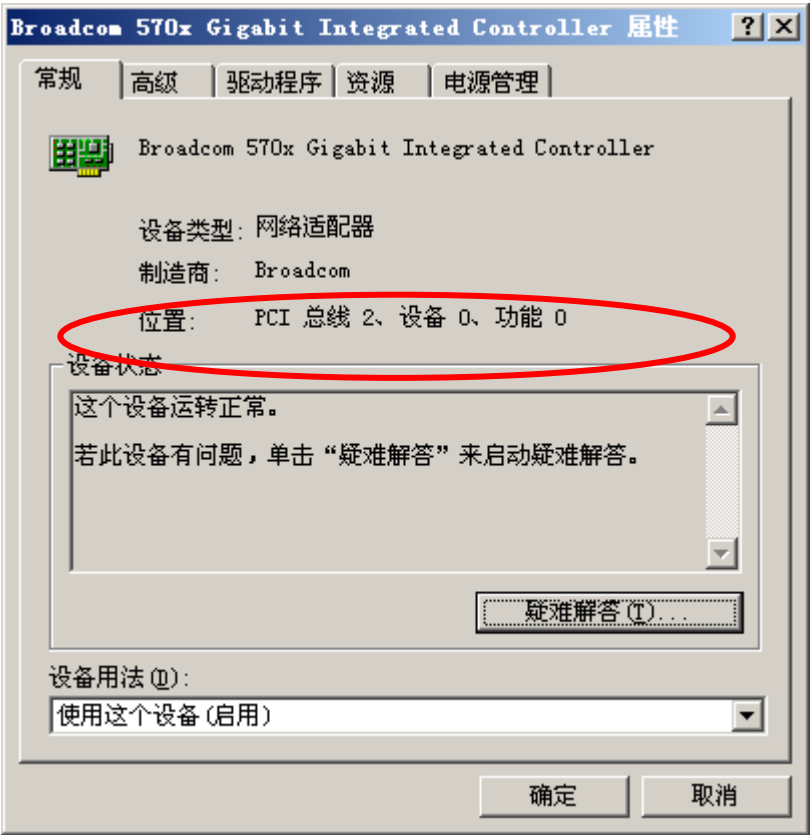


图 附 4-4 通过 Windows 的设备管理器查看设备信息

其中，画面中的位置信息，就是该设备在系统中的物理位置（总线 2、设备 0、功能 0）。

一个读取 PCI 配置信息的例子

有了上述描述的背景信息之后，我们列举一个使用 IOCTL 程序，读取 PCI 接口的网络接口卡的配置信息的例子。在一台计算机上，安装了一块 PCI 接口的网络接口卡，该网络接口卡的位置信息为：PCI 总线 1，设备 8，功能 0。

假设计算机启动并完成配置后（BIOS 对硬件的配置），引导 Hello China 操作系统，我们可以使用 IOCTRL 查看 BIOS 为该 NIC 分配的资源：

```
#outputd cf8 80014000
#inputd cfc
      813910EC
#outputd cf8 80014010
#inputd cfc
      0000d001
#outputd cf8 80014014
#inputd cfc
      EC000001
#outputd cf8 8001403c
#inputd cfc
      98AC001B
```

其中，第一个 inputd 输入的，是 PCI 设备 ID 和厂家 ID，第二个 inputd 输入的，是设备第一个 base address register 的值，该数值说明，为该 PCI 设备分配了输入/输出端口 D000—D0FF（端口的范围可以通过对该寄存器进行读取和写入操作完成，详细信息请参考 PCI 规范文本），第三个 inputd 输入的，是 BIOS 为该 PCI 设备分配的内存映射资源，即内存映射区域，最后一个 inputd，输入了系统配置空间的最后四个字节，其中，最后四个字节中，第一个字节（倒数第四个字节）是系统所占用的中断向量号，可以看出，这块 NIC 所占用的中断向量号是 11（B）。

## 附录 H 优先队列（Priority Queue）和 环形缓冲区（RING BUFFER）的实现

—— The implementation of Priority Queue and  
Ring Buffer

## 优先队列概述

在操作系统的实现当中，到处会用到队列数据结构，比如核心线程队列、核心对象的等待队列等。这样若把这些对队列的功能要求进行抽象，实现一个适应度相对广泛的队列对象，会大大减少代码量。

在 Hello China V1.0 的实现中，引入了一个队列对象 `__PRIORITY_QUEUE`，所有核心对象（从 `__COMMON_OBJECT` 继承）都可以存储到该队列中。虽然从名字上看，该队列对象是一个优先队列，但若以相同的优先级插入元素，则该队列会演变成一个先入先出队列（FIFO），因此，该队列的使用面是很广的，在 Hello China V1.0 的实现中，几乎所有用到队列的地方，都是使用该优先队列来实现的。

但在 Hello China V1.0 的实现中，`__PRIORITY_QUEUE` 对象的实现算法不是很完善，效率相对较低，且不支持多 CPU（SMP）的情况。因此，在 Hello China V1.5 的实现中，专门对该优先队列对象进行了优化，在大大提升其操作性能的同时，也插入了多 CPU 支持的代码，虽然 V1.5 仍然不支持 SMP 结构，但至少在 V1.5 中优化的 `__PRIORITY_QUEUE` 对象，可以在后续版本里，无需经过优化而直接支持 SMP 结构。

## 优先队列对象的定义和对外接口

下列是 `__PRIORITY_QUEUE` 对象在 Hello China V1.5 中的实现定义：

```
BEGIN_DEFINE_OBJECT(__PRIORITY_QUEUE)
    INHERIT_FROM_COMMON_OBJECT    //Inherit from __COMMON_OBJECT.
    __PRIORITY_QUEUE_ELEMENT      ElementHeader;
    volatile DWORD                 dwCurrElementNum;
    BOOL                           (*InsertIntoQueue)(
                                    __COMMON_OBJECT* lpThis,
                                    __COMMON_OBJECT* lpObject,
                                    DWORD              dwPriority
                                );
    BOOL                           (*DeleteFromQueue)(
                                    __COMMON_OBJECT* lpThis,
                                    __COMMON_OBJECT* lpObject
                                );
```

```

__COMMON_OBJECT*          (*GetHeaderElement)(
__COMMON_OBJECT* lpThis,
        DWORD*          lpPriority
        );

END_DEFINE_OBJECT( )

```

该队列对象是从\_\_COMMON\_OBJECT 继承而来的，因此其可以通过 ObjectManager 对象提供的方法(CreateObject) 进行创建和销毁，这样符合 Hello China 采用的对象框架。ElementHeader 是该队列元素链表的头元素，即所有插入到该队列中的元素，都被以双向链表的形式连接在一起。在 Hello China V1.0 的实现中，该元素则是一个指针（指向\_\_QUEUE\_ELEMENT）结构，这个改进，是 V1.5 对优先队列所做的最大改进。

dwCurrElementNum 是当前队列中元素的个数，每当向优先队列中插入一个元素，该数值会被增加，每当从链表中删除一个元素（从队列中提取一个元素），则该数值会相应的递减。显然，若该元素为 0，则当前优先队列是空的，对于队列是否为空的判断，V1.5 的实现中，就是通过判断该数值是否为 0 来实现的。需要注意的是，该数值通过 volatile 关键字进行限制，以确保编译器不对其进行主观的优化，这在 SMP 结构中是十分关键的。

InsertIntoQueue、DeleteFromQueue 和 GetHeaderElement 三个函数，是对优先队列进行操作的接口函数，在本文后续部分会进行详细描述。

其中，\_\_PRIORITY\_QUEUE\_ELEMENT 的定义如下：

```

BEGIN_DEFINE_OBJECT(__PRIORITY_QUEUE_ELEMENT)
__COMMON_OBJECT*          lpObject;
        DWORD          dwPriority;
__PRIORITY_QUEUE_ELEMENT* lpNextElement;
__PRIORITY_QUEUE_ELEMENT* lpPrevElement;
END_DEFINE_OBJECT( )

```

其中，lpObject 是指向对象的指针，dwPriority 是该元素的优先级，lpNextElement 和 lpPrevElement 是两个指针，分别指向该元素在队列链表中的前一个元素和后一个元素。这样每当向队列中插入一个元素的时候，InsertIntoQueue 首先创建一个该对象，然后把待插入核心对象的指针和优先级赋给 lpObject 和 dwPriority，最后把该队列元素插入队列链表当中。这样，队列中所有元素组成如下的数据结构：

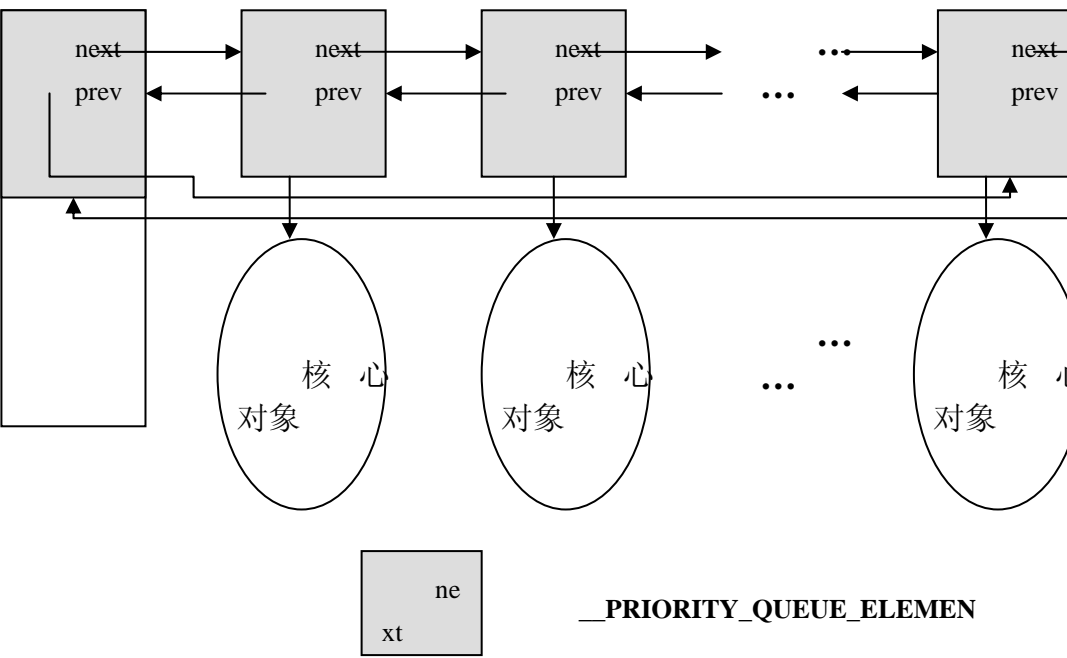


图1. 优先队列组织结构

这样内嵌在\_\_PRIORITY\_QUEUE 定义中的队列元素，只作为链表头使用，不算入队列元素个数当中。在初始化的时候，链表头元素的 lpNextElement 和 lpPrevElement 都指向自己。

## 中断安全和多 CPU 支持

在 Hello China V1.0 中对优先队列的实现，是中断安全的，即一个优先队列对象，可以被核心线程对象和中断处理程序同时进行引用，在实现的代码中，每当对可能产生共享冲突的数据结构进行修改（比如，插入一个队列元素）的时候，都调用 \_\_ENTER\_CRITICAL\_SECTION（）宏，确保在修改过程中不被中断，修改完毕后，再调用 \_\_LEAVE\_CRITICAL\_SECTION 宏，退出关键区段。由于 \_\_ENTER\_CRITICAL\_SECTION 以及对应的离开关键区段两个宏，是可以支持多 CPU 的（为 SMP 的支持预留了空间，详细信息请参考 V1.0 相关文档），因此从理论上说，V1.0 的实现已经是中断安全和支持 SMP 的。但实际上，不同的 CPU 有不同的内存冲突避免

机制。比如，在 IA32 构架的 CPU 上（不是全部，某些特定的型号），实现了内存窥探的机制，来确保不同 CPU 之间 cache 的同步，这种情况下，就无需考虑不同 CPU 之间 cache 的同步情况，只需考虑原子操作即可，但有的 CPU 则不具备这种能力，比如 ARM 体系构架的 CPU，这样就需要软件来确保不同 CPU 之间的数据同步。为了适应这种情况，每当对优先队列的共享数据结构（元素链表）进行修改完毕，就调用一个预先定义的宏 `__BARRIER()`，来把被当前 CPU 修改的数据，刷新到内存当中，这样内存中的内容，对位于同一系统上的其它 CPU 就是可见的。但若 CPU 不支持内存窥探机制，这样做还是不够的，假设在另外一个 CPU 上，保存了相同数据的副本（保存在数据 cache 中），这样内存中内容的修改，并不会被另外的 CPU 马上感知，这样若另外的 CPU 访问同样的数据，实际获得的还是未经修改的陈旧数据。因此，为了应对这种情况，在对共享数据访问前，再调用一次 `__BARRIER()` 宏，该宏可以完成不同构架 CPU（针对不同构架 CPU 分别定义）上 cache 和内存数据的同步，因此调用了该宏后，再对共享数据进行访问，获取的必然是最新的版本。

总之，与 Hello China V1.0 的实现不同的是，在 V1.0 实现的基础上，又增加了 `__BARRIER()` 宏的调用，这样可完全确保 SMP 情况下的数据一致性，从而可以安全的支持 SMP。

## 优先队列的实现

在这一部分中，我们对 `__PRIORITY_QUEUE` 的实现进行详细描述。

### InsertIntoQueue 的实现

该接口函数用于向队列中插入一个元素。该函数的动作如下：

- 1、判断参数是否合法，若调用者提供了非法的参数，则该函数返回 `FALSE`；
- 2、调用 `KMemAlloc` 函数，申请一块内存，作为队列元素 (`__PRIORITY_QUEUE_ELEMENT`)，然后初始化该队列元素；
- 3、在一个原子操作内（关键区段），把该队列元素插入队列元素列表，并更新队列元素计数。
- 4、所有上述操作成功后，返回 `TRUE`。

代码如下：

---

```

static BOOL InsertIntoQueue(__COMMON_OBJECT* lpThis,__COMMON_OBJECT*
lpObject,DWORD dwPriority)
{
    if((NULL == lpThis) || (NULL == lpObject)) //Invalid parameters.
    {
        return FALSE;
    }
    __PRIORITY_QUEUE_ELEMENT* lpElement = NULL;
    __PRIORITY_QUEUE_ELEMENT* lpTmpElement = NULL;
    __PRIORITY_QUEUE*          lpQueue   = (__PRIORITY_QUEUE*)lpThis;
    DWORD                      dwFlags   = 0L;

    //Allocate a queue element,and initialize it.
    lpElement =
    KMemAlloc(sizeof(__PRIORITY_QUEUE_ELEMENT),KMEM_SIZE_TYPE_ANY);
    if(NULL == lpElement) //Can not allocate the memory.
    {
        return FALSE;
    }
    lpElement->lpObject      = lpObject;
    lpElement->dwPriority     = dwPriority;
    //Now,insert the element into queue list.
    __ENTER_CRITICAL_SECTION(NULL,dwFlags); //Atomic operation.
    __BARRIER(NULL); //Barrier operation.
    lpQueue->dwCurrElementNum ++; //Increment element number.
    lpTmpElement = lpQueue->ElementHeader.lpPrevElement;
    //Find the appropriate position according to priority to insert.
    while((lpTmpElement->dwPriority < dwPriority) &&
        (lpTmpElement != &lpQueue->ElementHeader))
    {
        lpTmpElement = lpTmpElement->lpPrevElement;
    }
    //Insert the element into list.
    lpElement->lpNextElement = lpTmpElement->lpNextElement;
    lpElement->lpPrevElement = lpTmpElement;
    lpTmpElement->lpNextElement->lpPrevElement = lpElement;
    lpTmpElement->lpNextElement = lpElement;
    __BARRIER(NULL); //Flush CPU's internal content to memory.
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);

    return TRUE;
}

```

```
}

```

需要注意的是，在上述代码的关键区段开始和结束处，都调用了 `__BARRIER` 宏，用于完成多 CPU 之间的数据同步。

## DeleteFromQueue 的实现

对一个队列的操作，一般情况下只有入队列、出队列操作是符合逻辑的，但出队列操作，是取出队列的队列头元素，无法对队列中间的元素进行删除操作。而有的时候，却需要从队列中删除一个元素，比如在同步对象的超时等待操作中。队列删除操作，可以理解为排队过程中的放弃操作，比如排队等待某项服务，在尚未到达队头（为得到服务）的情况下，主动离去。

`DeleteFromQueue` 就是这种操作，该函数删除队列中的指定元素。被删除的目标元素，是由其指针（地址）指定的，因此若队列中存在相同的多个核心对象的情况（`InsertIntoQueue` 允许多次插入一个对象的情况），则该操作只删除离队头元素最近的那一个。

该函数动作如下：

- 1、判断参数的合法性；
- 2、从队列头开始，寻找符合删除条件的队列元素；
- 3、找到后从列表中删除，并递减队列元素计数，然后返回 `TRUE`；
- 4、若无法找到，则返回 `FALSE`。

代码如下：

```
static BOOL DeleteFromQueue(__COMMON_OBJECT* lpThis, __COMMON_OBJECT*
lpObject)
{
    if((NULL == lpThis) || (NULL == lpObject)) //Invalid parameters.
    {
        return FALSE;
    }

    __PRIORITY_QUEUE*    lpQueue = (__PRIORITY_QUEUE*)lpThis;
    __PRIORITY_QUEUE_ELEMENT* lpElement = NULL;
    DWORD dwFlags;

    lpElement = lpQueue->ElementHeader.lpNextElement;
```

```

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    __BARRIER(NULL);
    while((lpElement->lpObject != lpObject) && (lpElement !=
&lpQueue->ElementHeader))
    {
        lpElement = lpElement->lpNextElement;
    }
    if(lpObject == lpElement->lpObject) //Found,delete it.
    {
        lpQueue->dwCurrElementNum --;
        lpElement->lpNextElement->lpPrevElement =
lpElement->lpPrevElement;
        lpElement->lpPrevElement->lpNextElement =
lpElement->lpNextElement;
        __BARRIER(NULL); //Commit the change.
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        KMemFree(lpElement,KMEM_SIZE_TYPE_ANY,0L); //Free memory.
        return TRUE;
    }
    //Not found the target object to delete.
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags); //No need barrier.
    return FALSE;
}

```

需要注意的是，\_\_BARRIER 和 \_\_LEAVE\_CRITICAL\_SECTION 宏在 KMemFree 函数前调用，为了尽可能少的缩短原子操作的范围，这样可使系统的相应时间得到保证。

## GetHeaderElement 的实现

顾名思义，GetHeaderElement 函数获取队头元素，并从队列中删除之。该函数动作如下：

- 1、判断参数的合法性；
- 2、判断当前队列是否为空，若是，则返回 NULL；
- 3、若当前队列不为空，则从链表中删除第一个元素，释放所占用的内存，并返回对象指针；
- 4、若 lpdwPriority 参数不为空，则返回该队列元素的优先级。

代码如下：

```

__COMMON_OBJECT*   GetHeaderElement(__COMMON_OBJECT*   lpThis,DWORD*
lpdwPriority)
{
    if(NULL == lpThis)
    {
        return FALSE;
    }
    __PRIORITY_QUEUE*   lpQueue = (__PRIORITY_QUEUE*)lpThis;
    __PRIORITY_QUEUE_ELEMENT* lpElement = NULL;
    __COMMON_OBJECT* lpCommObject = NULL;
    DWORD dwFlags;

    lpElement = lpQueue->ElementHeader.lpNextElement;
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    __BARRIER(NULL);
    if(lpElement == &lpQueue->ElementHeader) //Queue empty.
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return NULL;
    }
    //Queue not empty,delete header element.
    lpQueue->dwCurrElementNum -= 1;
    lpElement->lpNextElement->lpPrevElement =
lpElement->lpPrevElement;
    lpElement->lpPrevElement->lpNextElement =
lpElement->lpNextElement;
    __BARRIER(NULL);
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    lpCommObject = lpElement->lpObject;
    if(lpdwPriority) //Should return the priority value.
    {
        *lpdwPriority = lpElement->dwPriority;
    }
    KMemFree(lpElement,KMEM_SIZE_TYPE_ANY,0L); //Release the
element.
    return lpCommObject;
}

```

需要注意的是，关键区段只包涵了从链表中删除队列元素的操作，对于队列元素的销毁（内存释放）、返回值的设定等工作，都是在关键区段外完成的，这样可确保关键区

段的范围尽可能小，以提高系统的相应时间。

另外，一定要在销毁队头元素（KMemFree 函数）前，保存 lpObject，并设置返回的队列优先级，否则一旦队列元素对象被删除，则系统无法保证这些数据依然会合法。

## 初始化和销毁函数的实现

由于\_\_PRIORITY\_QUEUE 是按照核心对象来实现的，这样就必须提供两个函数 Initialize 和 Uninitialize，用以完成对象被创建时的初始化，以及对象销毁后的资源释放工作。在 V1.5 的实现中，Initialize 函数实现如下：

```
BOOL PriQueueInitialize(__COMMON_OBJECT* lpThis)
{
    __PRIORITY_QUEUE*          lpPriorityQueue    = NULL;

    if(NULL == lpThis)          //Parameter check.
    {
        return FALSE;
    }

    //Initialize the Priority Queue.
    lpPriorityQueue = (__PRIORITY_QUEUE*)lpThis;
    lpPriorityQueue->ElementHeader.lpObject = NULL;
    lpPriorityQueue->ElementHeader.dwPriority = 0L;
    lpPriorityQueue->ElementHeader.lpNextElement =
        &lpPriorityQueue->ElementHeader;
    lpPriorityQueue->ElementHeader.lpPrevElement =
        &lpPriorityQueue->ElementHeader;
    lpPriorityQueue->dwCurrElementNum = 0L;
    lpPriorityQueue->InsertIntoQueue = InsertIntoQueue;
    lpPriorityQueue->DeleteFromQueue = DeleteFromQueue;
    lpPriorityQueue->GetHeaderElement = GetHeaderElement;

    return TRUE;
}
```

可见，该函数完成了优先队列数据成员的初始化工作，并初始化了操作函数列表。

对于 Uninitialize 函数，实现如下：

```
VOID PriQueueUninitialize(__COMMON_OBJECT* lpThis)
{
    __PRIORITY_QUEUE_ELEMENT*    lpElement    = NULL;
    __PRIORITY_QUEUE_ELEMENT*    lpTmpElement = NULL;
    __PRIORITY_QUEUE*            lpPriorityQueue = NULL;

    if(NULL == lpThis)
    {
        return;
    }

    lpPriorityQueue = (__PRIORITY_QUEUE*)lpThis;
    lpElement = lpPriorityQueue->ElementHeader.lpNextElement;
    while(lpElement != &lpPriorityQueue->ElementHeader)
    {
        lpTmpElement = lpElement;
        lpElement = lpElement->lpNextElement;
        KMemFree(lpTmpElement, KMEM_SIZE_TYPE_ANY, 0L);    //Release
        element.
    }
}
```

可以看出，该函数主要是把队列元素列表清空，并删除每个队列元素。

队列接口函数的复杂度分析

本文开始的时候曾经提到，虽然\_\_PRIORITY\_QUEUE 对象的名字是一个优先队列，但若忽略其优先特性，以相同的优先级（比如 0）把元素插入队列，则队列会演变成一个先进先出队列（FIFO）。因此，在这一部分，我们按照两种情形—优先队列和 FIFO 队列，对每个操作函数的复杂度进行分析，如下表：

接口函数\队列形态	优先队列 (PriorityQueue)	先进先出队列(FIFO)
InsertIntoQueue	O(n)	O(1)
DeleteFromQueue	O(n)	O(n)
GetHeaderElement	O(1)	O(1)

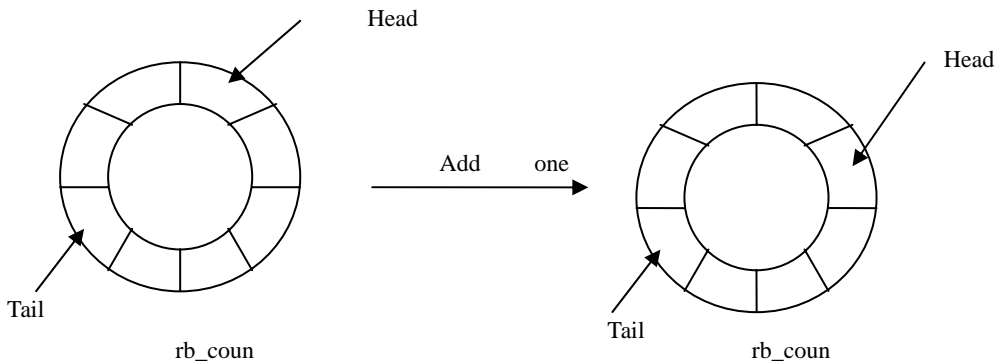
可见，若队列按 **FIFO** 形态使用，最常用的两个操作（入队列和出队列）的时间复杂度都是  $O(1)$ ，也就是说不随队列长度的变化而变化，这可满足许多要求苛刻的场合。虽然从队列中删除元素的时间复杂度是线性的，但这个操作用到的场合比较少。

但若按照优先队列来使用，则队列的插入操作也变成线性的。因此，在实际使用该队列对象的时候，尽量按照 **FIFO** 形态来使用，在必须使用优先队列的情况下（比如，线程的就绪队列），则建议限制队列的长度，比如，限制队列长度为 64 个队列元素。

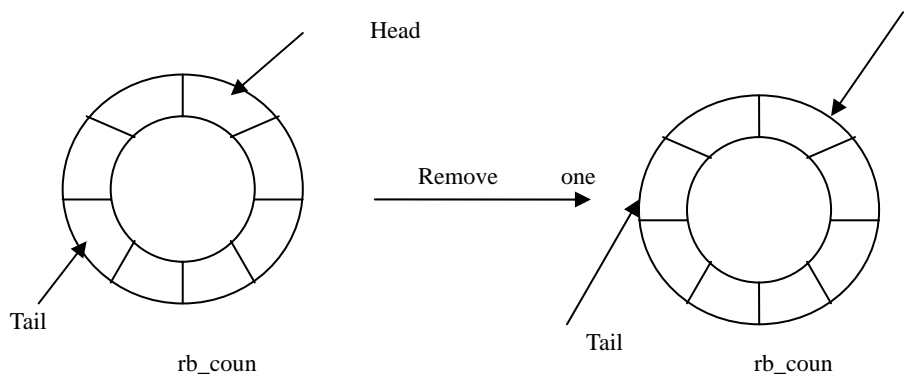
## 环形缓冲区概述

在设备驱动程序等的开发过程中，需要使用到环形缓冲区（Ring Buffer）。环形缓冲区是一种用于缓存特定数据的特殊缓冲区，采用两个指针跟踪缓冲区的头和尾：每当在缓冲区中加入一个元素，头指针往后移动一个元素，指向下一个空位置；每当从缓冲区中取出一个元素，则尾指针前移，指向下一个可移出的元素。采用一个计数器跟踪缓冲区中的元素个数。一旦往缓冲区中增加一个元素，则计数器增加 1。相应地，从缓冲区中取出一个元素，则计数器减少 1。当计数器为 0 的时候，说明缓冲区为空。当计数器增加到缓冲区的最大大小的时候，说明缓冲区已满。此时若再往缓冲区中增加元素，则会覆盖掉最先加入缓冲区中的元素。

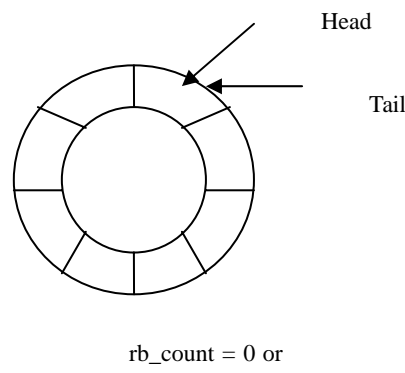
下图说明了往缓冲区中增加一个元素的过程：



下图则显示了从缓冲区中取出一个元素的情形：



而在缓冲区空或满的时候，头指针和尾指针是重合的，如下图：



唤醒缓冲区的另外一种实现形式，是额外增加一个元素位置，这个元素位置不存储任何实际的数据，而只是用于标志环形缓冲区的空或满状态。这种实现跟采用计数器跟踪缓冲区空或满的实现类似，在此不作赘述。

在当前版本的 Hello China 中，环形缓冲区中的元素，是定义为 DWORD 类型的，即一个无符号的 32 位整数。这样可存储任何形式的元素。比如，可通过类型强制转换，把 UCHAR、BYTE、WORD 等存储到缓冲区，也可通过把指针强制转换为 DWORD 的形式，把应用程序自定义的数据存储到缓冲区里面。

另外，在当前版本实现的环形缓冲区对象，是多线程安全的，也是中断安全的。即一个环形缓冲区对象，可以被多个线程所共享，也可以被中断处理程序和线程共享。因

为环形缓冲区一般应用在设备驱动程序的编写当中，中断安全是必须的。

## 环形缓冲区的实现

下面是环形缓冲区对象的定义：

```
BEGIN_DEFINE_OBJECT(__RING_BUFFER)
    INHERIT_FROM_COMMON_OBJECT
    DWORD*      lpBuffer;
    DWORD        dwCount;          //Current element count.
    DWORD        dwBuffLength;    //Buffer's length.
    __EVENT      eventWait;
    DWORD        dwHeader;
    DWORD        dwTail;

    //BOOL        (*Initialize)(__COMMON_OBJECT* lpThis); //Initialize
routine.
    BOOL        (*SetBufferLength)(__COMMON_OBJECT* lpThis,
                                   DWORD dwNewLength);    //Reset  buffer
length.
    BOOL        (*GetElement)(__COMMON_OBJECT* lpThis,
                              DWORD*      lpElement,
                              DWORD        dwMillionSecond);
    BOOL        (*AddElement)(__COMMON_OBJECT* lpThis,
                              DWORD        dwElement);
END_DEFINE_OBJECT()
```

该对象是从\_\_COMMON\_OBJECT 继承的，拥有了\_\_COMMON\_OBJECT 的所有功能，可以通过 OBJECT MANAGER 对象创建。

### 初始化函数的实现

由于环形缓冲区对象从\_\_COMMON\_OBJECT 对象继承，因此直接继承了\_\_COMMON\_OBJECT 的 Initialize 和 Uninitialize 两个函数。其中 Initialize 用于完成初始化功能，而 Uninitialize 则在对象被销毁的时候调用。初始化函数实现如下：

```

BOOL RbInitialize(__COMMON_OBJECT* lpThis)
{
    __RING_BUFFER *lprb = (__RING_BUFFER*)lpThis;

    if(NULL == lprb) //Invalid parameter.
    {
        return FALSE;
    }
    lprb->dwBuffLength    = DEFAULT_RING_BUFFER_LENGTH;
    lprb->lpBuffer        = (DWORD*)KMemAlloc(KMEM_SIZE_TYPE_ANY,
                                                lprb->dwCount,
                                                0L);

    if(NULL == lprb) //Can not allocate memory.
    {
        return FALSE;
    }
    lprb->dwCount        = 0L;
    lprb->dwHeader       = 0L;
    lprb->dwTail         = 0L;

    lprb->eventWait = (__EVENT*)
        ObjectManager.CreateObject(&ObjectManager
        ,
        NULL,
        OBJECT_TYPE_EVENT);
    if(NULL == lprb) //Can not create event object.
    {
        goto __TERMINAL;
    }
    if(!lprb->eventWait->Initialize((__COMMON_OBJECT*)lprb->eventWait
))

```

```
{
    goto __TERMINAL;
}
lprb->eventWait->ResetEvent((__COMMON_OBJECT*)lprb->eventWait)
);
lprb->SetBuffLength    = SetBuffLength;
lprb->GetElement        = GetElement;
lprb->AddElement        = AddElement;
return TRUE;

__TERMINAL:    //If any error,release all resource and return FALSE.
if(NULL != lprb->eventWait)
{
    ObjectManager.DestroyObject(&ObjectManager,
(__COMMON_OBJECT*)lprb->eventWait);
}
if(NULL != lprb->lpBuffer)
{
    KMemFree(KMEM_SIZE_TYPE_ANY,lprb->lpBuffer,0);
}
return FALSE;
}
```

该函数比较简单，主要完成了缓冲区存储空间的创建，各成员变量的初始化，以及事件对象的创建和初始化等。事件对象是用于完成缓冲区同步的。

对应地，Uninitialize 函数释放 Initialize 函数所申请的资源，包括缓冲区内存、创建的事件对象等。Uninitialize 函数非常简单，在此不做赘述。

## GetElement 函数的实现

GetElement 从环形缓冲区中获取一个元素。该函数支持阻塞操作，即若缓冲区中没有元素，调用该函数的核心线程可进入阻塞状态。其中阻塞可以是超时阻塞（阻塞一段时间后后被唤醒），也可以是永久阻塞。一旦有元素被放入缓冲区，则阻塞的核心线程会

被重新唤醒。

该函数实现如下：

```

BOOL GetElement(__COMMON_OBJECT* lpThis,DWORD* lpdwElement,
                DWORD dwMillionSecond)
{
    __RING_BUFFER      lprb = (__RING_BUFFER*)lpThis;
    DWORD              dwFlags;

    if((NULL == lprb) || (NULL == lpdwElement)) //Invalid parameter.
    {
        return FALSE;
    }
    __REPEAT:
    //The following is critical section code.
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(lprb->dwCount == 0) //Buffer is empty.
    {
        if(0 == dwMillionSecond) //Should return immediately.
        {
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            return FALSE;
        }
        if(MAX_DWORD_VALUE == dwMillionSecond) //Should infinite
wait.
        {
            lprb->eventWait->WaitForThisObject((__COMMON_OBJECT)*
            lprb->eventWait); //Wait for ever.
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            goto __REPEAT; //Once be waken up,try again.
        }
        else //Should submit a timeout wait.
        {
            switch(lprb->eventWait->WaitForThisObjectEx(

```

```
        (__COMMON_OBJECT*)lprb->eventWait,
        dwMillionSecond))
    {
        case OBJECT_WAIT_RESOURCE: //Element has available.
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            goto __REPEAT; //Try again.
        case OBJECT_WAIT_TIMEOUT: //Time out.
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            return FALSE;
    }
}
}
else //Buffer is not empty.
{
    lprb->dwCount --;
    if(0 == lprb->dwCount) //Empty again.
    {
        lprb->eventWait->ResetEvent((__COMMON_OBJECT*)
            lprb->eventWait); //Reset the event object.
    }
    *lpdwElement = lprb->lpBuffer[lprb->dwTail]; //Get element.
    lprb->dwTail ++;
    if(lprb->dwTail == lprb->dwBuffLength) //Reach buffer's boundry.
    {
        lprb->dwTail = 0;
    }
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return TRUE;
}
return FALSE; //Will never reach here.
}
```

上述代码有些复杂。但主要是考虑了两种情况：缓冲区中有元素的情况（dwCount 不为 0）和缓冲区中无数据元素的情况（对应 dwCount 为 0）：

对于缓冲区中无数据元素的情况，根据 `dwMillionSecond` 参数的不同，有三种应对措施：

- 1、 若 `dwMillionSecond` 为 0，则不做任何等待措施，直接返回 `FALSE`。这种情况是一种非阻塞操作，调用 `GetElement` 函数的核心线程（或中断例程）不会进入阻塞状态，而是直接返回；
- 2、 若 `dwMillionSecond` 为 `MAX_DWORD_VALUE`，则进入一个无限期等待，直到缓冲区中有元素（由其它核心线程或中断驱动程序放入）。待缓冲区中有元素后，等待的核心线程会被唤醒，这时候再重新做一个相同的检查；
- 3、 若 `dwMillionSecond` 为其它数值，则进入一个超时等待过程。超时时长设置为 `dwMillionSecond`。一旦等待超时，则会返回 `FALSE`。在没有超时被唤醒（缓冲区中被放入了元素）的情况下，说明缓冲区中已有元素可使用，因此会再试图从缓冲区中获取元素。

若缓冲区中有数据元素存在，则直接把缓冲区中的数据取走，与此同时更新 `dwCount` 和缓冲区的尾指针。这时候又有两种特殊情况要处理：

- 1、 `dwCount` 重新变为 0（即当前只有一个元素，已被取走），这时候需要重新设置事件对象为无信号状态（`ResetEvent`），以使得后续试图获取元素的核心线程，能够阻塞在该事件对象上；
- 2、 尾指针（`dwTail`）达到缓冲区的最大元素（即上边界）。这时候需要设置 `dwTail` 的值为 0，以便迂回到缓冲区的开始部分。这样就形成了环形缓冲区。

需要注意的是，上述所有操作，都是在一个关键区段内完成的。

## AddElement 函数的实现

`AddElement` 函数完成添加一个数据元素到环形缓冲区的操作。若缓冲区满，则该函数会覆盖掉环形缓冲区尾部的数据元素。

函数实现代码如下：

```

BOOL AddElement(__COMMON_OBJECT* lpThis,
                DWORD dwElement)
{
    __RING_BUFFER*    lprb = (__RING_BUFFER*)lpThis;
    DWORD              dwFlags;

    if(NULL == lprb) //Invalid parameter.

```

```

    {
        return FALSE;
    }
    //The following is critical section code.
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(0 == lprb->dwCount)    //Buffer is empty.
    {
        lprb->lpBuffer[lprb->dwHeader] = dwElement;
        lprb->dwHeader ++;
        if(lprb->dwHeader == lprb->dwBufferLength) //Reach up
boundary.
        {
            lprb->dwHeader = 0;
        }
        lprb->dwCount ++; //Increment element counter.
        lprb->eventWait->SetEvent(
            (__COMMON_OBJECT*)lprb->eventWait); //Wakeup waiting
threads.
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return TRUE;
    }
    if(lprb->dwBufferLength == lprb->dwCount) //Buffer is full.
    {
        lprb->lpBuffer[lprb->dwHeader] = dwElement;
        lprb->dwHeader ++;
        lprb->dwTail ++; //Omit the element has been replaced.
        if(lprb->dwHeader == lprb->dwBufferLength) //Reach up
boundary
        {
            lprb->dwHeader = 0;
            lprb->dwTail = 0; //In case of full,tail must equal header.
        }
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return FALSE; //Indicate one element has been replaced.
    }

```

```

    }
    //Normal case,not full and not empty.
    lprb->lpBuffer[lprb->dwHeader] = dwElement;
    lprb->dwHeader ++;
    if(lprb->dwHeader == lprb->dwBufferLength) //Reach up
boundary.
    {
        lprb->dwHeader = 0;
    }
    lprb->dwCount ++; //Increment element counter.
    //lprb->eventWait->SetEvent(__COMMON_OBJECT*)
    //lprb->eventWait); //Wakeup waiting threads.
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return TRUE;
}

```

AddElement 函数考虑了三种情况，每种情况及处理过程如下：

- 1、缓冲区中元素个数为 0（即环形缓冲区为空）。这种情况下，直接把待插入元素放在 dwHeader 位置即可。但这种情况下，需要复位 Event 对象，以使得等待的核心线程能够被唤醒；
- 2、缓冲区中元素个数为缓冲区大小的情况（即缓冲区满）。这种情况下，需要覆盖掉缓冲区尾部的元素。需要注意的是，这种情况下，缓冲区的头指针和尾指针必定是重合的，因此在更新头指针的时候，尾指针也需要前移一个元素（因为原来的尾元素已经被覆盖）。同时，若两个指针到达了缓冲区的上边界，则需要对两个指针同时复位（设置为 0）；
- 3、缓冲区不空也不满的情况。这种情况最简单，只需把元素放在缓冲区的内存中，然后更新缓冲区的头指针即可。这种情况下也需要考虑缓冲区指针超过缓冲区上边界的情况。若超过，需要复位头指针。