

信息科学与技术丛书

辛庆祥 编著

操作系统 实现之路

- 内容完整** 涉及Hello China操作系统实现的方方面面
- 基于实例** 通过简洁深刻的实例描述问题
- 操作性强** 像修改应用程序一样修改操作系统内核
- 面向未来** 探讨了操作系统的发展趋势、物联网操作系统等内容



机械工业出版社
CHINA MACHINE PRESS

信息科学与技术丛书

操作系统实现之路

辛庆祥 编著



机械工业出版社

本书以 Hello China 操作系统为例,详细讲解了操作系统的内核、文件系统、图形界面、设备驱动程序、SDK 和系统调用等主要功能模块的实现原理。讲解过程中不仅陈述概念,还配以详细的实现源代码对概念进行说明,达到理论联系实际的目的。书中穿插了大量的案例,读者可通过亲手操作这些案例来更加深入地理解操作系统原理。此外,本书还对操作系统发展趋势和商业模式进行了探讨。

本书可供程序员和计算机相关专业师生阅读。

相关的源代码可在 blog.csdn.net/hellochina15 下载。

图书在版编目(CIP)数据

操作系统实现之路 / 辛庆祥编著. —北京:机械工业出版社, 2013.2
(信息科学与技术丛书)

ISBN 978-7-111-41444-5

I. ①操… II. ①辛… III. ①操作系统—程序设计 IV. ①TP316

中国版本图书馆 CIP 数据核字(2013)第 026062 号

机械工业出版社(北京市百万庄大街 22 号 邮政编码 100037)

策划编辑:车 忱

责任编辑:车 忱

责任印制:邓 博

保定市 中画美凯印刷有限公司印刷

2013 年 4 月第 1 版·第 1 次印刷

184mm×260mm·31 印张·769 千字

0001—3500 册

标准书号:ISBN 978-7-111-41444-5

定价:79.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

电话服务

网络服务

社 服 务 中 心 : (010) 88361066

教 材 网 : <http://www.cmpedu.com>

销 售 一 部 : (010) 68326294

机工官网 : <http://www.cmpbook.com>

销 售 二 部 : (010) 88379649

机工官博 : <http://weibo.com/cmp1952>

读者购书热线 : (010) 88379203

封面无防伪标均为盗版

出版说明

随着信息科学技术的迅速发展，人类每时每刻都会面对层出不穷的新技术和新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术不断地受到挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机和互联网在社会生活中日益普及，掌握计算机网络技术和理论已成为大众的文化需求。由于信息科学技术在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中已经得到充分、广泛的应用，所以这些专业领域中的研究人员和工程技术人员越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们了解和掌握新知识、新技能的热切期待，以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络和工程应用等内容，注重理论与实践的结合，内容实用、层次分明、语言流畅，是信息科学与技术领域专业人员不可或缺的参考书。

目前，信息科学技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设作出贡献。

机械工业出版社

前 言

欢迎阅读本书！首先说一下阅读本书所需要的一些基础知识。本书聚焦于操作系统的实现，对实现所需要的工具，比如 C 语言、汇编语言等，并没有做深入介绍。因此需要读者首先具备这些编程语言的基本知识和使用技能。在此基础上，建议读者先熟悉一下 Intel x86 CPU 的架构和工作原理，尤其是保护模式的工作原理，会对阅读本书有很大帮助。如果读者没有这些基础，则建议先不要阅读本书，否则可能会产生挫折感。我个人认为，挫折感会打击学习热情，对学习效果造成重大影响。

再说一下哪些人士适合阅读本书。我认为任何计算机专业的人士，都可通过阅读本书而获益。如果您本身就是操作系统方面的专家，则可通过阅读本书了解一些独特的操作系统设计理念，虽然这些理念不一定多么先进和高明，但至少是独一无二的。独一无二的东西往往是最有价值的。如果您是应用软件编程人员，则可通过阅读本书深入洞悉操作系统的工作原理，这会对应用软件的开发有很大帮助，毕竟操作系统是所有软件的基础。如果您是一名系统架构师，那么这本书就更适合您了。操作系统设计最核心的内容，就是其架构设计。可以通过本书了解一些重要的架构设计思想。当然，如果您的架构水平很高，也可以从专业的角度提出进一步的优化建议。我认为书籍的本质就是一种交流工具，读者和作者通过这个工具交流思想，相互学习，共同提升。

现在简单说明一下本书的特点。这虽然有一些自夸的成分，但会帮助您做出选择，到底是否阅读本书。随着我国系统软件水平的整体提升，操作系统原理和实现方面的书越来越多，且大都质量不错。如何根据这些书籍的特点选出最适合您自身的呢？这是一个问题，毕竟您是独一无二的，适合别人的书不一定适合您。本书的第一个特点是理论联系实际，通过深入剖析笔者开发的 Hello China 操作系统来说明操作系统的原理。这很容易理解，无非是通过例题来说明原理。第二个特点是内容完善，包含操作系统的内核、图形用户界面、文件系统、设备驱动程序、SDK、用户 shell 等方方面面的实现说明，希望通过一本书，让读者了解整个操作系统，而不仅仅是内核。另外一个特点是，本书除介绍操作系统实现的技术细节外，还探讨了当前 IT 环境下，操作系统应该如何发展和演进的问题。当然，这只是作者的个人理解，主要目的是同业界同仁进行探讨。众所周知，操作系统实现的技术壁垒已不存在，制约操作系统发展的是商业模式。

接下来简单介绍一下 Hello China 操作系统。这是作者利用业余时间开发的智能终端操作系统，具备鲜明的特点（详情请参考本书第 1 章内容），本书以 V1.75 版本为例来讲解操作系统的实现原理。这个版本功能全面通用，又不过度复杂，且直接运行在个人计算机上，非常适合作为实例讲解。对于这个操作系统，作者将持续开发下去，并欢迎有兴趣的朋友一起参与开发。我认为操作系统会向按行业或应用场景细分的方向发展，某一操作系统的应用领域将会局限在某个专业的范围之内。这样可使操作系统本身聚焦某个行业，成为行业发展的内在引擎，产生的总体经济效益远大于通用操作系统模式。Hello China 后续版本聚焦于物联网领域，希望做成面向物联网应用的软件平台，来支撑物联网的发展。

最后我想说明一下，这不仅仅是一本书，随之一起提供给您还有后续的学习和沟通服务。这包括问题解答、后续的资料共享、Hello China 操作系统最新功能的介绍等。只要您选择了本书，作者就有义务让您完全理解书中的内容。当然，这需要您加入作者创建的 QQ 群，或者关注作者的 blog。详细的联系方式以及更进一步的信息，请访问作者的 blog：

<http://blog.csdn.net/hellochina15>

本书相关的源代码，也需要通过这个链接下载。

受作者水平限制，书中错误或不当之处在所难免。希望读者朋友能多多提出批评意见，以期共同进步。还是那句话，书是一种交流的工具，希望以此为纽带，促成读者和作者、读者之间的交流，并使每个参与者从交流中获益。本书写作过程中得到了很多人的支持，包括家人、朋友、Hello China 操作系统爱好者、机械工业出版社等，在此一并感谢。

祝您阅读愉快！

作 者

目 录

出版说明

前言

第 1 章 操作系统概述	1	2.2.2 Hello China 在 Virtual PC 上 的安装过程	28
1.1 操作系统的基本概念	1	2.3 Hello China 在物理计算机上 的安装	31
1.1.1 操作系统的功能	1	2.3.1 安装注意事项	31
1.1.2 操作系统的分类	2	2.3.2 在 Windows XP 操作系统上 的安装	31
1.1.3 操作系统的发展趋势	2	2.3.3 在 Windows 7 操作系统上 的安装	32
1.1.4 操作系统的基本概念	3	2.4 Hello China 的卸载	34
1.2 嵌入式系统和嵌入式操作系统	6	2.5 Hello China 的使用	34
1.2.1 嵌入式系统概述	6	2.6 内核的编译和生成	35
1.2.2 嵌入式操作系统概述	7	2.6.1 Hello China 内核的开发环境	35
1.2.3 嵌入式操作系统的特点	7	2.6.2 开发环境的搭建	36
1.2.4 嵌入式操作系统与通用操作 系统的区别	8	2.6.3 内核映像文件的生成	38
1.2.5 嵌入式实时操作系统	9	第 3 章 Hello China 的引导和 初始化	41
1.3 Hello China 操作系统概述	10	3.1 概述	41
1.3.1 Hello China 的主要功能	10	3.2 个人计算机的引导和 初始化	41
1.3.2 Hello China 的架构	12	3.2.1 BIOS 的引导工作	41
1.3.3 Hello China 的主要特点	12	3.2.2 硬盘逻辑结构及引导扇区 的功能	42
1.3.4 Hello China 的应用场景	14	3.2.3 操作系统引导扇区的功能 和局限	44
1.3.5 面向对象思想的模拟	17	3.2.4 预置引导法概述	45
1.4 实例：一个简单的 IP 路由器 的实现	20	3.2.5 预置引导法在 FAT32 文件 系统上的实现	46
1.4.1 概述	20	3.2.6 预置引导法在 NTFS 文件系统 上的实现	47
1.4.2 路由器的硬件结构	21	3.2.7 通过软盘启动 Hello China	49
1.4.3 路由器的软件功能	22		
1.4.4 各任务的实现	23		
第 2 章 Hello China 的安装和使用	26		
2.1 Hello China 安装概述	26		
2.2 Hello China 在 Virtual PC 上 的安装	26		
2.2.1 Hello China 在 Virtual PC 上的 启动过程	26		

3.3 嵌入式操作系统的引导和初始化	53	第 5 章 内存管理机制	129
3.3.1 典型嵌入式系统内存映射布局	53	5.1 内存管理机制概述	129
3.3.2 嵌入式系统的启动概述	54	5.2 IA32 CPU 内存管理机制	129
3.3.3 常见嵌入式操作系统的加载方式	55	5.2.1 IA32 CPU 内存管理机制概述	129
3.3.4 嵌入式系统软件的写入	59	5.2.2 几个重要的概念	131
3.4 Hello China 的初始化	61	5.2.3 分段机制的应用	132
3.4.1 实地址模式下的初始化	61	5.2.4 分页机制的应用	135
3.4.2 保护模式下的初始化	65	5.3 Power PC CPU 的内存管理机制	142
3.4.3 操作系统核心功能的初始化	68	5.4 Hello China 内存管理模型	144
3.4.4 Hello China 的内存布局图	73	5.4.1 Hello China 的内存管理模型	144
3.5 Hello China 的字符 shell	74	5.4.2 Hello China 的内存布局	146
3.5.1 字符 shell 的概述和启动	74	5.4.3 核心内存池的管理	147
3.5.2 shell 的消息处理过程	75	5.4.4 页框管理对象	149
3.5.3 实例：增加一个字符 shell 内置命令	78	5.4.5 页面索引对象	154
3.6 从保护模式切换回实模式	79	5.4.6 虚拟内存管理对象	158
3.6.1 模式切换概述	79	5.5 线程本地堆	175
3.6.2 实模式服务调用举例	81	5.5.1 线程本地堆概述	175
3.6.3 保护模式切换到实模式	82	5.5.2 堆的功能需求定义	175
3.7 引导和初始化总结	87	5.5.3 堆的实现概要	177
第 4 章 Hello China 线程的实现	88	5.5.4 堆的详细实现	181
4.1 进程、线程和任务	88	5.6 Hello China 的内存管理机制总结	193
4.2 Hello China 的线程实现	89	第 6 章 系统调用的原理与实现	194
4.2.1 核心线程管理对象	89	6.1 系统调用概述	194
4.2.2 线程的状态及其切换	95	6.2 Hello China 系统调用的实现	197
4.2.3 核心线程对象	96	6.3 系统调用时的参数传递	199
4.2.4 线程的上下文	99	第 7 章 线程互斥和同步机制的实现	204
4.2.5 线程的创建和初始化	101	7.1 互斥和同步概述	204
4.2.6 线程的结束	107	7.2 关键区段概述	204
4.2.7 线程的消息队列	108	7.3 关键区段产生的原因	205
4.2.8 线程的切换——中断上下文	111	7.3.1 多个线程之间的竞争	205
4.2.9 线程的切换——系统调用上下文	118	7.3.2 中断服务程序与线程之间的竞争	206
4.2.10 上下文保存和切换的底层函数	123	7.3.3 多个 CPU 之间的竞争	206
4.2.11 线程的睡眠与唤醒	128	7.4 单 CPU 下关键区段的实现	207
4.2.12 核心线程实现总结	128		



7.5 多 CPU 下关键区段的实现	209	8.7.3 ResetTimer 函数的调用	243
7.5.1 多 CPU 环境下的实现方式	209	8.8 设置定时器操作	243
7.5.2 Hello China 的未来实现	210	8.9 定时器超时处理	245
7.6 Power PC 下关键区段的实现	211	8.10 定时器取消处理	247
7.6.1 Power PC 提供的互斥访问机制	211	8.11 定时器复位	248
7.6.2 多 CPU 环境下的互斥机制	212	8.12 定时器注意事项	249
7.7 关键区段使用注意事项	213	第 9 章 系统总线和设备管理	251
7.8 Semaphore 概述	214	9.1 系统总线概述	251
7.9 Semaphore 对象的定义	214	9.1.1 系统总线	251
7.10 Semaphore 对象的实现	215	9.1.2 总线管理模型	251
7.10.1 Initialize 和 Uninitialize 的实现	216	9.1.3 设备标识符	255
7.10.2 WaitForThisObject 的实现	217	9.2 系统资源管理	255
7.10.3 WaitForThisObjectEx 的实现	218	9.2.1 资源描述对象	256
7.10.4 ReleaseSemaphore 的实现	222	9.2.2 IO 端口资源管理	257
7.11 互斥和同步机制总结	223	9.3 驱动程序接口	257
第 8 章 中断和定时处理机制	224	9.3.1 GetResource	258
的实现	224	9.3.2 GetDevice	258
8.1 中断和异常概述	224	9.3.3 CheckPortRegion	258
8.2 硬件相关部分处理	225	9.3.4 ReservePortRegion	258
8.2.1 IA32 中断处理过程	225	9.3.5 ReleasePortRegion	259
8.2.2 IDT 初始化	226	9.3.6 AppendDevice	259
8.3 硬件无关部分处理	230	9.3.7 DeleteDevice	259
8.3.1 系统对象和中断对象	230	9.4 PCI 总线驱动程序概述	260
8.3.2 中断调度过程	232	9.4.1 PCI 总线概述	260
8.3.3 缺省中断处理函数	233	9.4.2 PCI 设备的配置空间	260
8.4 对外服务接口	234	9.4.3 配置空间关键字段的说明	262
8.5 系统时钟中断	235	9.4.4 PCI 配置空间的读取与设置	269
8.5.1 系统时钟中断概述	235	9.5 PCI 总线驱动程序的实现	270
8.5.2 系统时钟中断的初始化	236	9.5.1 探测 PCI 总线是否存在	270
8.5.3 系统时钟中断处理函数的		9.5.2 对普通 PCI 设备进行枚举	271
主要工作	237	9.5.3 配置 PCI 桥接设备	277
8.5.4 时钟中断周期对系统实时性		第 10 章 设备驱动程序管理	279
的影响分析	239	10.1 设备管理框架	279
8.6 注意事项	241	10.1.1 概述	279
8.7 定时器概述	241	10.1.2 通用的操作系统设备管理机制	281
8.7.1 SetTimer 函数的调用	242	10.1.3 设备管理框架的实现	285
8.7.2 CancelTimer 函数的调用	243	10.1.4 IO 管理器	287

10.2 设备驱动程序.....	295	11.6.3 二层窗口剪切域的实现.....	358
10.2.1 设备请求控制块.....	295	11.7 Hello China 窗口机制的实现.....	360
10.2.2 设备驱动程序对象的定义.....	298	11.7.1 窗口管理器与窗口对象.....	361
10.2.3 设备驱动程序的物理结构.....	299	11.7.2 窗口函数与窗口消息.....	363
10.2.4 设备驱动程序的功能函数.....	300	11.7.3 窗口归属线程.....	366
10.2.5 DriverEntry 的实现.....	302	11.7.4 窗口树.....	366
10.2.6 UnloadEntry 的实现.....	303	11.8 用户输入处理和消息传递.....	369
10.3 设备对象.....	304	11.8.1 用户输入和消息传递过程简介.....	369
10.3.1 设备对象的定义.....	304	11.8.2 设备驱动程序的工作.....	372
10.3.2 设备对象的命名.....	304	11.8.3 设备输入管理器.....	379
10.3.3 设备对象的类型.....	305	11.8.4 GUI 原始输入线程 ——GUIRAWIT.....	383
10.3.4 设备对象的设备扩展.....	306	11.8.5 消息循环的本质.....	388
10.3.5 设备的打开操作.....	306	11.8.6 应用线程之间的窗口消息交互.....	392
10.3.6 设备命名策略.....	307	11.9 Hello China 的 GUI Shell.....	393
10.4 设备的中断管理.....	308	11.9.1 GUI Shell 概述.....	393
10.5 设备管理实例：串口 通信程序.....	309	11.9.2 GUI Shell 的启动和初始化.....	395
10.5.1 串行通信接口概述.....	309	11.9.3 加载一个应用程序.....	399
10.5.2 串行通信编程方式.....	312	11.9.4 GUI Shell 的退出.....	400
10.6 设备驱动程序管理总结.....	334	11.10 GUI 模块的开发方法.....	402
第 11 章 图形用户界面.....	336	第 12 章 文件系统及其实现.....	404
11.1 图形用户界面概述.....	336	12.1 文件系统概述.....	404
11.2 符合 VESA 标准的显卡 操作方法.....	337	12.1.1 文件系统的基本概念.....	404
11.2.1 判断显卡是否支持 VBE 标准.....	338	12.1.2 文件系统的操作——fs 程序.....	405
11.2.2 切换到 0x118 工作模式.....	339	12.2 FAT32 文件系统原理.....	406
11.3 对显示设备的封装——Video 对象.....	342	12.2.1 FAT32 卷的布局.....	406
11.3.1 GUI 模块的分层架构.....	342	12.2.2 引导扇区和 BPB.....	407
11.3.2 Video 对象.....	343	12.2.3 文件分配表.....	408
11.3.3 通用绘制层简介.....	349	12.2.4 文件目录项.....	410
11.4 鼠标指针的实现.....	351	12.2.5 文件的查找.....	411
11.5 窗口消息传递机制概述.....	354	12.3 Hello China 文件系统的实现.....	412
11.6 Hello China 的窗口机制.....	355	12.3.1 IO 管理器.....	412
11.6.1 父窗口要完全包含子窗口.....	355	12.3.2 文件系统的加载和初始化.....	418
11.6.2 二层窗口剪切域.....	357	12.3.3 存储设备驱动程序.....	419
		12.3.4 分区的识别和安装.....	426
		12.3.5 文件的打开操作.....	431
		12.3.6 文件的读取操作.....	434



12.4 文件系统 API 的使用举例.....	437	14.2.1 MS-DOS 头和 DOS stub 程序	454
12.5 文件系统实现总结.....	439	14.2.2 PE 文件头	455
第 13 章 应用程序开发方法	441	14.2.3 PE 文件中的节.....	458
13.1 概述	441	14.3 开发辅助工具的实现和使用	460
13.2 HCX 文件的结构和 加载过程.....	441	14.3.1 process 工具.....	460
13.2.1 HCX 文件的格式	441	14.3.2 hcxbuild 工具	461
13.2.2 HCX 文件的生成方式.....	442	14.3.3 append 工具	463
13.2.3 HCX 文件的加载和执行	443	14.3.4 vfmaker 工具	464
13.3 Hello China 应用程序 开发步骤	444	14.3.5 dumpf32 和 mkntfsbs 工具	464
13.3.1 建立应用程序开发环境.....	444	附录	465
13.3.2 启动 VS 2008, 建立一个新的 应用程序.....	444	附录 A 关于操作系统开发的 两篇博文.....	465
13.3.3 在应用程序中添加源代码	444	A.1 操作系统开发过程应遵循的 一些原则	465
13.3.4 对新建的应用程序进行设置	446	A.2 对操作系统开发的一些 相关问题的思考.....	469
13.3.5 编写应用程序代码, 并进行 编译链接	448	附录 B 源代码组织结构说明	473
13.3.6 对生成的 DLL 进行处理, 形成 HCX 文件	450	附录 C 内核开发环境的搭建	476
13.3.7 运行生成的 HCX 文件.....	451	C.1 概述	476
13.4 应用程序开发总结.....	452	C.2 Microsoft Visual C++ 的设置	479
第 14 章 开发辅助工具	453	C.3 操作系统开发中常用的 Microsoft Visual C++ 特性	479
14.1 开发辅助工具概述.....	453	C.4 搭建操作系统开发环境	480
14.2 PE 文件格式简介	453	C.5 操作系统核心模块开发示例.....	482
		参考文献	485

第 1 章 操作系统概述

1.1 操作系统的基本概念

本书主要介绍操作系统的实现，囿于篇幅，不会对操作系统的基本原理做太多的阐述。但首先还是简单介绍一下与操作系统相关的关键概念，帮助读者建立操作系统的整体图景，为后续内容的阅读做好铺垫。

1.1.1 操作系统的功能

操作系统最核心的功能是管理计算机资源。计算机资源可进一步分为硬件资源和软件资源两大类。硬件资源指的是组成计算机系统的硬件设备，如中央处理器、主存储器、磁盘存储器、打印机、显示器、键盘输入设备和鼠标等。软件资源指的是存放于计算机内的各种数据和文件。操作系统位于底层硬件与用户应用程序之间，是两者沟通的桥梁，也是计算机系统的第一层软件（其下层就是计算机硬件）。用户可以通过操作系统的用户界面来操作计算机。

以现代观点而言，一个完善的操作系统应该至少提供以下的功能：

（1）进程管理（Process Management）。本质上是对计算机的 CPU 的管理。操作系统需要有效协调计算机系统内的一个或多个 CPU，为用户程序提供效率最高的服务。为了提升 CPU 的利用效率，现代操作系统会引入线程、进程等概念，把用户程序划分成逻辑上独立的执行线索，然后按照一定的算法或规则，给这些执行线索分配 CPU，完成计算任务。

（2）内存管理（Memory Management）。即计算机的随机访问存储器（RAM）的管理。操作系统需要通过某种算法，动态管理和监控内存的分配，并按照应用程序的需要来分配内存。管理的原则是：尽量保证能够满足应用程序的内存需求，同时也要确保内存的使用效率。

（3）文件系统（File System）。本质上是外部存储器，比如硬盘、光驱、磁带、各种存储卡等的管理。按照预先定义的规则，把这些存储设备分片（比如分为扇区、磁道等存储单位），然后对每个分片的使用情况进行跟踪，确保外部存储设备能够有效使用，同时确保存储在上面的数据是准确的、可恢复的。

（4）用户界面（User Interface）。提供计算机系统与用户的接口，方便用户操作计算机。字符模式的命令行界面和图形模式的 GUI（图形用户界面）是最常见的两种用户接口呈现形式。其本质是对显示设备和键盘、触摸屏、鼠标等输入设备的有效管理。

（5）设备管理（Device Drivers）。对计算机中除上述描述的物理设备外的物理设备的管理，比如对声卡的管理、打印机的管理等。

（6）网络协议（Network Protocol）。现代社会，网络无处不在。从家庭宽带网络、企业



办公网络，到如火如荼的移动互联网，甚至到将来的泛在网（无处不在的网络），随时随地接入网络是计算机的最根本要求。这就要求计算机操作系统能够提供多种多样的网络接口方式（Ethernet/光纤等固定接入方式，WiFi/GPRS/3G/LTE 等无线接入方式），同时能够提供符合国际标准的网络协议栈（比如 IP 协议）。这些支持都是操作系统的任务。

1.1.2 操作系统的分类

目前的操作系统种类繁多，很难进行统一分类。下面只是根据一个单一的应用维度，来对操作系统做一个简单的分类，目的是进一步加深读者对操作系统功能的理解。

（1）根据应用领域来划分，可分为桌面操作系统、服务器操作系统、嵌入式操作系统等。

（2）根据所支持的用户数目，可分为单用户操作系统（比如 MSDOS、OS/2、早期的 Windows）、多用户操作系统（比如 UNIX 系列）等。

（3）根据源码开放程度，可分为开源操作系统（比如 Linux、Chrome OS）和不开源操作系统（比如 Windows）。

（4）根据操作系统对作业处理的方式，可分为批处理系统（比如 MSDOS）、分时操作系统（比如 Linux、UNIX、Windows 等）、实时操作系统（比如 VxWorks、μC/OS、RTOS 等）。

（5）根据 CPU 指令的长度，可分为 8bit、16bit、32bit、64bit 的操作系统。

值得一提的是，随着移动互联网和物联网的发展，又出现了专门针对终端设备的操作系统。这类操作系统又可分为智能终端操作系统和非智能终端操作系统等。所谓智能终端操作系统，是指提供了明确的开发接口和丰富的操作系统特性，能够通过开发应用程序来扩展本身功能的操作系统。非智能终端操作系统，则是一个封闭的、与硬件结合紧密的嵌入式操作系统，这类操作系统不提供面向公众的开发接口，因此无法通过开发应用程序扩充其功能。典型的智能终端操作系统是 Android 和 Apple iOS，但是这两者都是应用于手机、平板电脑等个人消费设备上的。还有应用于其他设备的智能终端操作系统，比如本书介绍的 Hello China，就是面向物联网终端设备的智能终端操作系统。

1.1.3 操作系统的发展趋势

这里说的操作系统发展趋势，特指操作系统的应用发展趋势。只要计算机的结构没有本质变化（目前大部分都是冯·诺依曼体系结构），操作系统的架构就不会有太大的变化。但操作系统的应用场景却在不断演变，作者认为，操作系统正朝着按应用场景细分的方向发展。即针对每种应用场景，或某个特定的用户群，会有一个或多个与之相应的操作系统。比如，以前的操作系统，大致可分为桌面操作系统、服务器操作系统和嵌入式操作系统等三大类。Windows、Linux 是桌面操作系统的典型代表，UNIX 系列操作系统在服务器（或大型机）领域一家独大，嵌入式领域，则存在 pSOS、VxWorks、μC/OS 等操作系统。而到了当前的移动互联网时代，智能移动终端这个应用场景出现后，又催生了广泛应用于智能终端上的 Android 操作系统、Apple iOS 操作系统等。随着云计算的兴起，云操作系统又有流行的趋势。可以看出，操作系统的类别（或种类）并不是一成不变的，而是随着应用的不断变化和演进，会有全新的操作系统被开发出来，以适应这些应用。总体呈现出一种按照应用场景进行细分的趋势。

随着移动互联网的不断发展成熟，会逐渐催生出更多的应用场景，比如家庭网络、物联

网等。由于体系结构的限制，传统的操作系统很可能无法适应这些新兴场景的需求，因此又会催生出一批更新的操作系统。

1.1.4 操作系统的基本概念

为了便于后续内容的讲解，先简单介绍与操作系统相关的几个概念。

1. 微内核与大内核

微内核与大内核是操作系统设计中不同的两种思想，这与 CPU 的 RISC（精简指令集）和 CISC（复杂指令集）架构类似。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进行实现，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、内存管理、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。由于效率是嵌入式操作系统追求的最主要目标，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。

图 1-1 示意了微内核操作系统的设计思路。

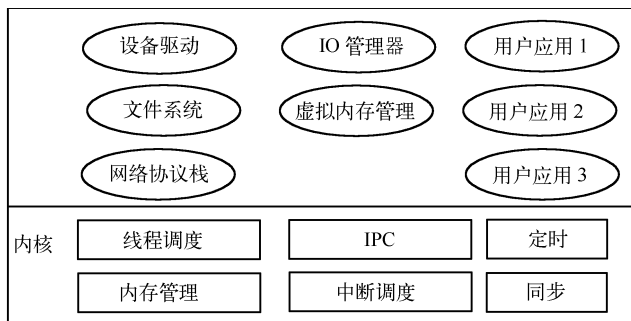


图 1-1 微内核操作系统的设计思路

大内核的思路相反，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。所以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，最明显的问题就是内核过于庞大，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 1-2 示意了大内核的开发思路。

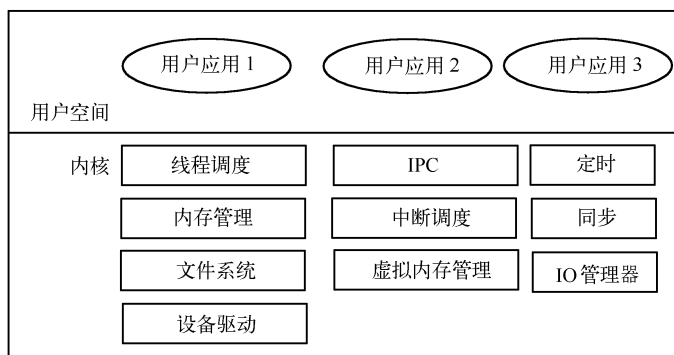


图 1-2 大内核操作系统的设计思路

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 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，线程之间共享同一进程的全局变量，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元。另外一个说法就是，任务往往是一个无限循环，操作系统启动时，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，而是与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，因此也用“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”。

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，操作系统不会强行剥夺其运行权，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。

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 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 `WaitForObject` 等待该信号量对象时， N 就减 1，如果 N 小于 0，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，而对于 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非 Terminal 状态的内核线程对象，那么将一直阻塞，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，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。

1.2 嵌入式系统和嵌入式操作系统

1.2.1 嵌入式系统概述

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各类电子产品，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。这些说法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。但笔者认为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、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/电转换装置等组成的，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，软件仅起到辅助作用。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，因为软件比重高的系统中，大部分功能是由软件完成的，通过添加更多的软件，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进行支撑，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 1-3 所示。

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，除了性能和实现细节的差异，功能部分往往是相同的。本书介绍的就是一个嵌入式操作系统的功能及其功能的实现细节。

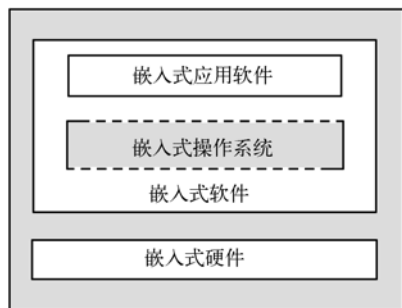


图 1-3 嵌入式系统软、硬件之间的关系

1.2.2 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。

1.2.3 嵌入式操作系统的特点

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，对整个操作系统的功能进行细致划分，每个功能模块尽量以独立模块的形式来实现。

具体的裁剪方式有两种。一种方式是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，与应用代码一起联编，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. 与应用代码一起链接

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，链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，它有自己的二进制映像，可以自行启动计算机，应用程序单独编译链接，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

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，仅 CPU 可能就会有十几款。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

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 功能或 HTTP 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。

1.2.4 嵌入式操作系统与通用操作系统的区别

1. 地址空间的区别

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，每个进程都有自己的 4GB 地址空间。这样每个进程相互独立，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 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（详细信息可参考第 5 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) 嵌入式系统通常没有本地存储介质，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，嵌入式操作系统一般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. 应用方式的区别

通用的操作系统在使用之前必须先进行安装，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。而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，因而也不存在安装的过程。

嵌入式操作系统还有一些其他特点，在此不再详述，有兴趣的读者可参阅相关资料。

1.2.5 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）那样对事件的响应有一个严格的底限，但又与普通操作系统对外部事件响应的不确定性有所区别。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 1-4 所示。

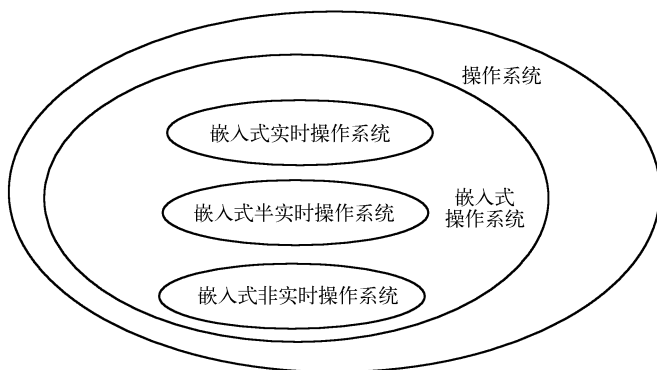


图 1-4 操作系统之间的关系

1.3 Hello China 操作系统概述

Hello China 是作者开发的一个面向智能终端设备的智能终端操作系统，本质上是一个嵌入式操作系统，但是提供了与通用操作系统基本一样的机制和服务，同时提供了基于 PC 的版本，方便读者学习和使用。本书以 Hello China V1.75 为例进行讲解，这个版本比较稳定，功能完备，同时又不过于复杂，非常适合讲解使用。Hello China 具备操作系统应该具备的所有核心功能，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、字符/图形界面、文件系统、设备管理、网络支持等。

本节对 Hello China V1.75 的功能特点做一个初步介绍，本书的后续部分，将对这些功能和机制的实现进行详细说明。

1.3.1 Hello China 的主要功能

Hello China V1.75 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）可抢占式调度。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, 基于硬件 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。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把设备内存映射到 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 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 等基于单中断向量机制的 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接口函数, 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) 文件系统支持。当前版本的 Hello China 可支持 FAT32 文件系统的高效读写功能, 支持 NTFS 文件系统的读取功能 (因为 NTFS 文件系统规范不公开, 所以无法实现其写入功能)。同时提供了一组供文件系统调用的接口函数, 很容易在系统中增加其他文件系统的支持。

(12) 实现了完善的字符命令行用户接口和图形用户接口功能。使用字符命令行接口, 可以方便地遍历文件系统、查看 CPU 占用率、诊断故障、操纵设备等。而 GUI 则提供了基本的界面元素和 API, 应用程序可调用这些 API, 实现更加丰富的界面元素。

(13) 实现了可执行文件的动态加载功能。使操作系统核心模块与外围应用程序完全分离。可通过开发应用程序扩展操作系统功能。

(14) 实现了系统调用功能。这是隔离应用程序和操作系统核心模块的基础机制。

(15) 提供了较为完整的应用开发工具包 (SDK), 用户可通过 Visual C++、Visual Studio 等开发环境, 方便地开发应用程序, 而无需对操作系统进行任何修改。

(16) Hello China V1.75 的 PC 版, 支持 Virtual PC、VMware 等虚拟机。

此外, Hello China 还提供了其他操作系统相关的服务, 并实现了基于 TCP/IP 的网络协

议栈（本书不涉及这部分内容）。

1.3.2 Hello China 的架构

Hello China V1.75 操作系统的大致逻辑架构如图 1-5 所示。

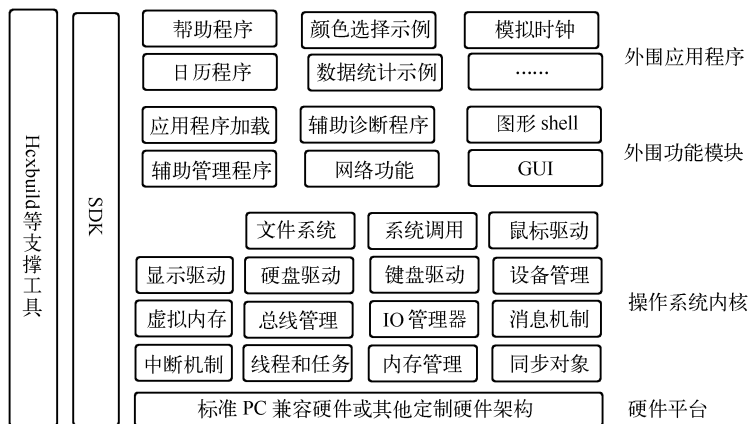


图 1-5 Hello China V1.75 的逻辑架构

总体上说，整个操作系统由内到外，可分为四个逻辑层次。

（1）硬件平台，这是支撑操作系统运行的硬件系统，比如 IBM PC 兼容机、专用的嵌入式硬件平台等。严格地说，这个层次不能算作操作系统的功能。

（2）操作系统内核，主要由中断处理机制、线程模型、内存管理、线程同步、消息机制、虚拟内存、文件系统等操作系统核心机制以及键盘驱动、硬盘驱动、鼠标/显示设备驱动等基本驱动程序组成。这是操作系统的内核，实现最基本的操作系统功能。

（3）外围功能模块，指的是与操作系统内核独立的其他系统功能模块，比如 GUI、网络、图形 shell、应用程序加载器等辅助功能模块。这些模块调用操作系统内核提供的服务，同时向更上层的应用程序提供服务。

（4）外围应用程序，指的是与操作系统完全独立开发、调用操作系统功能实现的应用程序。比如，Hello China V1.75 版本内置了模拟时钟、日程序、各类 GUI 元素示例程序等几个示例程序。当然，用户可以调用 Hello China 提供的 API，进一步开发功能更强大的程序。

同时，Hello China 还提供了 SDK、专用的可执行文件格式（HCX，Hello China eXecutable）以及对应的生成工具等。这些辅助功能贯穿整个操作系统的开发和应用程序的开发，因此把它们纵向排放，并贯穿上述四个层次。

1.3.3 Hello China 的主要特点

Hello China 定位为智能终端操作系统，具备如下特点。

1. 伸缩性强

在最初设计的时候，就把 Hello China 操作系统的可伸缩性定为最重要的设计准则。所谓可伸缩性，指的是操作系统的功能和尺寸能够根据不同的需求灵活变动。从提供最

基本的内核服务，到包含图形界面、文件系统、网络功能、各类应用程序的复杂系统，都可通过 Hello China 的定制机制做到。目前，Hello China 实现了三级模块定制机制，如图 1-6 所示。

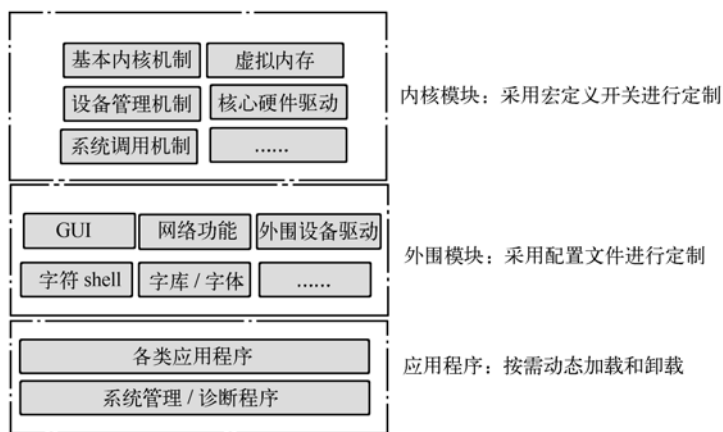


图 1-6 Hello China 的三级扩展机制

第一层定制机制是针对操作系统内核模块的源代码级定制。可通过设定一些宏定义开关，对源代码功能进行裁剪。这个层面的定制需要重新编译内核，但无需对源代码做任何修改，也非常方便。这个层级的定制，可以使 Hello China 的内核只包含线程调度、内存管理（无虚拟内存）、线程同步等最基本的机制，编译后的尺寸可以达到十几 KB，可满足功能受限的低端嵌入式系统的需求。

第二层定制机制是对外围模块的配置文件定制机制。Hello China V1.75 版本支持 GUI、网络、各类非核心的设备驱动程序等外围模块。系统定义了一个模块配置文件（MODCFG.INI），凡是在这个文件内的模块，操作系统就会加载，否则不会加载。这样可以通过合理配置模块配置文件，使得操作系统只加载必要的外围模块，而无需加载全部模块，以节约资源。比如，可以配置 Hello China，使之只加载网络功能模块，而无需加载 GUI 等外围模块，以满足网络设备的需要。当然，要实现这个层面的定制，操作系统内核必须支持文件系统，因为模块配置文件的读取和模块本身的读取都是通过文件系统完成的。

第三层定制机制是针对各类应用程序的动态加载和卸载机制。可以根据需要编写特定功能的应用程序，然后根据需要动态加载或卸载。这与通用操作系统类似，应用程序作为存放在外部存储介质上的可执行模块，可动态加载和卸载。

通过充分利用上述三个层面的定制机制，可以把 Hello China 的尺寸控制在十几 KB 到几 MB 的范围内，从而满足多种应用场景的需要。

2. 外部事件的快速响应

Hello China 采用基于优先级的线程调度算法，支持多时机（如中断程序结束后、系统调用结束后等）的线程切换机制，可对外部事件做出快速响应。Hello China 的中断机制支持基于优先级的中断调度方式，使关键外部中断能够优先处理。Hello China 的内存管理机制可支持多种内存分配算法，比如可支持分配时间固定的内存分配算法，这使得内存分配操作的时



间可预期，满足实时系统的需要。同时，Hello China 虽然可选择启用 MMU 机制，但不提供内存换出（换出到硬盘，需要时再调回内存）功能，这可充分保证内存分配的速度。

总之，通过这些机制的综合支持，Hello China 基本可以达到实时操作系统的标准要求。即使采用缺省的基于空闲链表的内存分配算法，禁止中断嵌套功能，其外部实时性也是可预期的。

3. 兼容性设计

这里的兼容性，指的是兼容已有标准或已有功能的能力。在设计的时候，Hello China 充分遵从相关领域的已有标准，尽量降低用户或程序员的学习成本。比如对文件系统的支持，Hello China 缺省支持 FAT32 和 NTFS（只读）等文件系统，这可使得已有的存储卡、硬盘等存储设备上的数据，很容易地共享到 Hello China 上。再如提供的 API 函数，尽量模拟 Windows 操作系统的 API 函数，这样可使得程序员的学习成本最低，无需重新学习一套全新的 API。又如，通过预置引导法等独创的引导机制，Hello China 可以与已有操作系统（如 Windows、Linux 等）和谐共存，在对现有操作系统无任何影响的情况下，即可在 PC 上安装 Hello China。

总之，做到最大可能的兼容性，可大大降低系统的使用成本和开发成本。

此外，Hello China 还有其他一些特征，比如独创的内核回调机制、系统调用注册机制、可扩展的文件系统管理框架、扩展性很强的驱动程序管理框架等，后续章节将做详细描述。

1.3.4 Hello China 的应用场景

Hello China 可以应用到各类智能终端设备中，比如支持智能控制的各类仪器仪表、各类交互式专用终端等。当然，也可以把 Hello China 应用于智能手机，但至少目前来说，其在这个领域的竞争力是比不上 Android 和 iOS 的。

另外一个重点应用领域是发展得如火如荼的物联网领域。Hello China 定位于物联网的控制网关、智能传感节点等设备的应用上，这也是正在规划的 Hello China V1.90 的主要应用场景。下面相对详细地介绍一下这个应用场景，使读者对 Hello China 的特点有更深入的认识。首先引入物联网操作系统的概念。

1. 物联网操作系统

物联网的概念和特征，在这里就不详细说明了，有兴趣的读者可以到网上搜索了解。我们大致提一下物联网的整体结构，物联网大致可分为感知层、网络层（即用于信息传输的基础网络）、设备管理层、应用层等四个层次。其中最能体现物联网特征的，就是物联网的感知层。感知层由各种各样的传感器、协议转换网关、通信网关、智能终端、刷卡机（POS 机）、智能卡等终端设备组成。这些终端大部分都是具备计算能力的微型计算机。运行在这些终端上的最重要的系统软件——操作系统，就是所谓的物联网操作系统。

与传统的个人计算机或个人智能终端（智能手机、平板电脑等）上的操作系统不同，物联网操作系统有自己的特征。这些特征是为了更好地服务物联网应用而存在的，运行物联网操作系统的终端设备，能够与物联网的其他层次结合得更加紧密，数据共享更加顺畅，能够大大提升物联网的生产力。

2. 物联网操作系统的架构

物联网操作系统由内核、通信组件支持、辅助外围模块（文件系统、图形用户界面、事件管理、XML 解析、Java 虚拟机等）、集成开发环境等组成，基于此，可衍生出一系列面向行业的特定应用。图 1-7 展示了这个概念。



图 1-7 物联网操作系统的大致架构

3. 物联网操作系统的特点

物联网操作系统与传统的个人计算机操作系统和智能手机类操作系统不同，它具备物联网应用领域内的一些独特特点，现说明如下。

(1) 内核尺寸伸缩性强，能够适应不同配置的硬件平台。比如，一个极端的情况下，内核尺寸必须维持在 10KB 以内，以支撑内存和 CPU 性能都很受限的传感器，这时内核具备基本的任务调度和通信功能即可。在另外一个极端的情况下，内核必须具备完善的线程调度、内存管理、本地存储、复杂的网络协议、图形用户界面等功能，以满足高配置的智能物联网终端的要求。这时的内核尺寸必然大大大增加，可以达到几百 KB 甚至 MB 级。这种内核尺寸的伸缩性，可以通过两个层面的措施来实现：重新编译和二进制模块选择加载。重新编译措施很简单，只需要根据不同的应用目标，选择所需的功能模块，然后对内核进行重新编译即可。这个措施应用于内核定制非常深入的情况，比如要求内核的尺寸小于 10KB 的场合。而二进制模块选择加载，则用在对内核定制不是很深入的情况。这时候维持一个操作系统配置文件，文件里列举了操作系统需要加载的所有二进制模块。在内核初始化完成后，会根据配置文件，加载所需的二进制模块。这需要终端设备有外部存储器（如硬盘、Flash 等），以存储要加载的二进制模块。

(2) 内核的实时性必须足够强，以满足关键应用的需要。大多数物联网设备要求操作系统内核具备实时性，因为很多关键动作必须在有限的时间内完成，否则将失去意义。内核的实时性包含很多层面的意思，首先是中断响应的实时性，一旦外部中断发生，操作系统必须在足够短的时间内响应中断并做出处理。其次是线程或任务调度的实时性，一旦任务或线程所需的资源或进一步运行的条件准备就绪，必须能够马上得到调度。显然，基于非抢占式调

度方式的内核很难满足这些实时性要求。

(3) 内核架构可扩展性强。物联网操作系统的内核，应该设计成一个框架，这个框架定义了一些接口和规范，只要遵循这些接口和规范，就很容易在操作系统内核上增加新的功能和新硬件支持。因为物联网的应用环境具备广谱特性，要求操作系统能够扩展以适应新的应用环境。内核应该有一个基于总线或树结构的设备管理机制，可以动态加载设备驱动程序或其他核心模块。同时内核应该具备外部二进制模块或应用程序的动态加载功能，这些应用程序存储在外部介质上，这样就无需修改内核，只需要开发新的应用程序，就可满足特定的行业需求。

(4) 内核应足够安全和可靠。可靠性就不用说了，物联网应用环境具备自动化程度高、人为干预少的特点，这要求内核必须足够可靠，以支撑长时间的独立运行。安全对物联网来说更加关键，甚至关系到国家命脉。比如一个不安全的内核应用到国家电网控制当中，一旦被外部侵入，造成的损失将无法估量。为了加强安全性，内核应支持内存保护（VMM 等）、异常管理等机制，以在必要时隔离错误的代码。另外一个安全策略，就是不开放源代码，或者不开放关键部分的内核源代码。不公开源代码只是一种安全策略，并不代表不能免费使用内核。

(5) 节能省电，以支持足够的电源续航能力。操作系统内核应该在 CPU 空闲的时候，降低 CPU 运行频率，或干脆关闭 CPU。对于周边设备，也应该实时判断其运行状态，一旦进入空闲状态，则切换到省电模式。同时，操作系统内核应最大程度地降低中断发生频率，比如在不影响实时性的情况下，把系统的时钟频率调到最低，以节约电源。

(6) 支持操作系统核心、设备驱动程序或应用程序等的远程升级。远程升级是物联网操作系统的最基本特征，这个特性可大大降低维护成本。远程升级完成后，原有的设备配置和数据能够继续使用。即使在升级失败的情况下，操作系统也应该能够恢复原有的运行状态。远程升级和维护是支持物联网操作系统大规模部署的主要措施之一。

(7) 支持常用的文件系统和外部存储，比如支持 FAT32/NTFS/DCFS 等文件系统，支持硬盘、U 盘、Flash、ROM 等常用存储设备。在网络连接中断的情况下，外部存储功能会发挥重要作用。比如可以临时存储采集到的数据，在网络恢复后再上传到数据中心。但文件系统和存储驱动的代码，要与操作系统核心代码有效分离，能够做到容易裁剪。

(8) 支持远程配置、远程诊断、远程管理等维护功能。这里不仅包含常见的远程操作特性，如远程修改设备参数、远程查看运行信息等，还应该包含更深层面的远程操作，比如可以远程查看操作系统内核的状态，远程调试线程或任务，异常时的远程 dump 内核状态等功能。这些功能不仅需要外围应用的支持，更需要内核的天然支持。

(9) 支持完善的网络功能。物联网操作系统必须支持完善的 TCP/IP 协议栈，包括对 IPv4 和 IPv6 的同时支持。这个协议栈要具备灵活的伸缩性，以适应裁剪需要。比如可以通过裁剪，使得协议栈只支持 IP/UDP 等协议功能，以减小代码尺寸。同时也支持丰富的 IP 协议族，比如 Telnet/FTP/IPSec/SCTP 等协议，以适应智能终端和安全性要求高的应用场合。

(10) 对物联网常用的无线通信功能要内置支持。比如支持 GPRS/3G/HSPA/4G 等公共网络的无线通信功能，同时要支持 Zigbee/NFC/RFID 等近场通信功能，支持 WLAN/Ethernet 等桌面网络接口功能。这些协议要能够相互转换，能够把从一种协议获取的数据报文，转换为另外一种协议的报文并发送出去。除此之外，还应支持短信息的接收和发送、语音通

信、视频通信等功能。

(11) 内置支持 XML 文件解析功能。物联网时代, 不同行业之间, 甚至相同行业的不同领域之间, 会存在严重的信息共享壁垒。而 XML 格式的数据共享可以打破这个壁垒, 因此 XML 标准在物联网领域会得到更广泛的应用。物联网操作系统要内置对 XML 解析的支持, 所有操作系统的配置数据, 统一用 XML 格式进行存储。同时也可对行业自行定义的 XML 格式进行解析, 以完成行业转换功能。

(12) 支持完善的 GUI 功能。图形用户界面一般应用于物联网的智能终端中, 完成用户和设备的交互。GUI 应该定义一个完整的框架, 以方便图形功能的扩展。同时应该实现常用的用户界面元素, 比如文本框、按钮、列表等。另外, GUI 模块应该与操作系统核心分离, 最好支持二进制的动态加载功能, 即操作系统核心根据应用程序需要, 动态加载或卸载 GUI 模块。GUI 模块的效率要足够高, 从用户输入确认, 到具体的动作开始执行之间的时间 (可以叫做 click-launch 时间) 要足够短, 不能出现用户点击了确定按钮但任务的执行却等待很长时间的情况。

(13) 支持从外部存储介质中动态加载应用程序。物联网操作系统应提供一组 API, 供不同应用程序调用, 而且这组 API 应该根据操作系统所加载的外围模块实时变化。比如在加载了 GUI 模块的情况下, 需要提供 GUI 操作的系统调用, 但是在没有 GUI 模块的情况下, 就不应该提供 GUI 功能调用。同时操作系统、GUI 等外围模块、应用程序模块应该二进制分离, 操作系统能够动态地从外部存储介质上按需加载应用程序。这样的一种结构, 就使得整个操作系统具备强大的扩展能力。操作系统内核和外围模块 (GUI、网络等) 提供基础支持, 而各种各样的行业应用通过应用程序来实现。最后在软件发布的时候, 只发布操作系统内核、所需的外围模块、应用程序模块即可

后续版本的 Hello China, 将根据上述特性需求做进一步的针对性开发, 以打造出一个高效通用的物联网操作系统。

1.3.5 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剪性。

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, 统一管理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做对象, 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++ 中，`struct` 关键字也用来定义对象类型）。比如在 C++ 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++ 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。

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。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，还适用于对象枚举、对象统计等具体功能。比如，为了列举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 实例：一个简单的 IP 路由器的实现

1.4.1 概述

下面以一个 IP 路由器的实现为例，来说明如何在 Hello China 操作系统环境下开发一个具体的应用。下面出现的一些概念，在后续章节中才涉及，因此如果读者阅读下列内容感到吃力，那么不要紧，可以读完本书后再返回阅读。之所以在这里举这个例子，是为了让读者对 Hello China 的整体架构有一个概要的理解。实际上，如果读者对线程、共享数据的访问等操作系统概念有认识的话，下列内容应该比较容易理解。

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，与另外一个 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 地址同掩码进行与运算，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，路由器会就接收这个 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。

1.4.2 路由器的硬件结构

在这个实例中，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 1-8 所示。

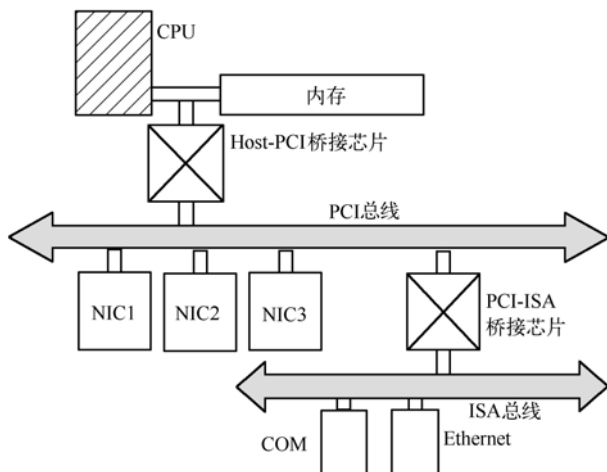


图 1-8 一个简单的路由器硬件架构

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 就可以很容易地与 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，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）。熟悉个人计算机（PC）硬件架构的读者，会发现该硬件架构与 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。

1.4.3 路由器的软件功能

在这种硬件架构下，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 发起一个中断请求。

(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）复制到内存中。

(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），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，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。

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录到路由器，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，若路由器应用在电信、大型企业等环境中，则还需要支持 SNMP（简单网络管理协议）等管理协议，以便通过网管中心进行维护。在我们的实例中，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 6 个任务组成，如图 1-9 所示。

图中，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。

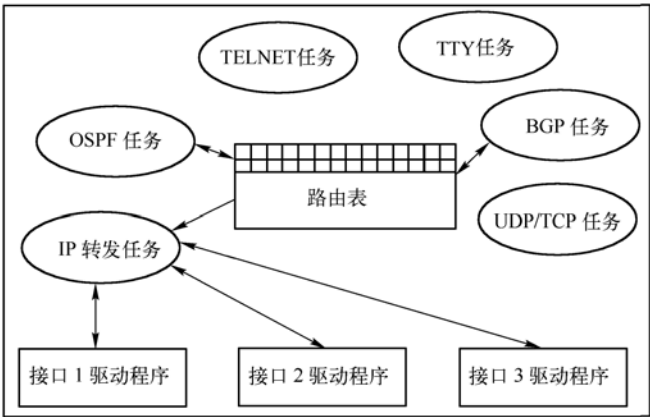


图 1-9 一个简单的路由器软件架构

1.4.4 各任务的实现

路由器的软件功能被分解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，代表实现路由器功能的 6 个功能模块。

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

Hello China 在完成自身初始化后，就应该启动上述 6 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) //创建线程失败
```

```
        goto __TERMINAL;
    //
    //创建剩余的 5 个核心线程
    //
    ... ..
}
```

这样，当 Hello China 完成自身的初始化后，上述路由器功能的 6 个线程就会被启动，目标系统就具备路由器的功能了。

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，就会把该 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);
    if(NULL == lpHasPacket)    //不能创建事件对象
        goto __TERMINAL;

    while(TRUE)
    {
        lpHasPacket->WaitForThisObject((__COMMON_OBJECT*) lpHasPacket);
        while(queue is not empty)
        {
            get an 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);
//修改路由表
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，然后进行修改，修改完毕，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 2 章 Hello China 的安装和使用

2.1 Hello China 安装概述

作为一个功能相对完善的操作系统，Hello China 可安装在虚拟机和物理计算机上。为了开发方便，建议读者在虚拟机上安装。但是如果应用到实际的生产环境中，则需要安装在物理计算机上。本部分内容将详细讲解如何在虚拟机和物理计算机上安装 Hello China。不论是在虚拟机上安装，还是在物理计算机上安装，都比较简单，而且是可完全卸载的。

对于在物理计算机上的安装，本来是可以做一个程序自动完成安装的。但是考虑到在安装过程中需要修改原有 Windows 操作系统的启动文件，这存在一定的风险，因此还是把安装步骤写出来，让读者自行修改。这样整个安装过程就处于一种完全可控的方式下，安全性大大提高。

2.2 Hello China 在 Virtual PC 上的安装

下面介绍如何在 Windows 7 操作系统和 Virtual PC 2007 虚拟机上安装 Hello China V1.75。对于 Windows XP 等非 Windows 7 操作系统，由于不能直接支持虚拟硬盘，因此不能按照本文介绍的方法安装 Hello China 的 GUI 功能，但是可以安装内核和基于字符界面的 shell。需要说明的是，其他虚拟机，比如 VMware，安装原理与 Virtual PC 一样，读者可仿照 Virtual PC 上的安装方式来完成 VMware 上的安装。

2.2.1 Hello China 在 Virtual PC 上的启动过程

首先介绍一下 Hello China V1.75 在 Virtual PC 上的启动过程。为了最大可能地降低安装和使用的复杂性，V1.75 版本在 Virtual PC 上是通过虚拟软盘启动的。Hello China 的内核和核心驱动程序（比如键盘驱动、鼠标驱动、IDE 接口硬盘驱动、文件系统等）等文件都集成在了虚拟软盘中。这样通过虚拟软盘启动计算机，操作系统的核心模块就直接从虚拟软盘中加载到内存并执行。内核初始化完成之后，Hello China 会进入字符 shell 模式，这时候用户就可以运行字符模式命令了。

在字符模式下，用户输入 `gui` 命令，即可进入图形模式的 shell。一旦用户输入 `gui` 命令，Hello China 会在硬盘的第一个分区（用 C: 标识，与 Windows 类似）的 PTHOUSE 目录下，寻找 `hcngui.bin` 文件，这个文件即是 Hello China 图形模式模块的可执行二进制文件。一旦找到这个文件，Hello China 内核就会把它读入内存，然后运行该模块。因此要在虚拟机上支持图形模式，则必须创建一个虚拟硬盘，并分区和格式化。完成后在其第一个分区上创建 PTHOUSE 目录，把 `hcngui.bin` 等文件复制到该目录上就可以了。

GUI 模块创建图形 shell 线程和其他图像模式的核心线程，然后初始化安装在 Hello China 上的应用程序。所谓初始化应用程序，指的是 GUI 模块读取所有应用程序的特征数据，比如应用程序名字、应用程序的图标等，然后把所有这些应用程序显示在图形界面的主窗口上。最终的启动结果如图 2-1 所示。



图 2-1 Hello China 图形模式启动结果

其中左边部分就是图形界面主窗口，里面列举了所有已安装在 Hello China 上的应用程序。一旦用户用鼠标点击某个应用程序，该程序就会运行。

在 Hello China V1.75 的实现中，所有应用程序都是安装（实际上就是复制）在硬盘第一个分区（C：分区）的 HCGUIAPP 目录下。在 GUI 模块的初始化过程中，会读取该目录下的所有文件，一旦发现一个合法的应用程序，就会进一步读取其特性数据，比如名称（上图中的显示名称）、图标等，然后加载到主窗口中。因此，要运行 Hello China V1.75 的应用程序，还必须在硬盘的第一个分区上创建一个 HCGUIAPP 目录，然后把所有应用程序文件复制到该目录即可。

Hello China V1.75 应用程序就是一个扩展名是 HCX（Hello China eXecutable）的文件，该文件包含了应用程序的可执行二进制代码、应用程序的图标、应用程序的名称和版本等信息。HCX 文件是由一个叫做 hcxbuild（随 Hello China V1.75 的 SDK 一起发行）的程序构建的。

总结起来，在 Virtual PC 上安装 Hello China，需完成下列工作。

（1）创建虚拟软盘文件，用于引导 Hello China。

（2）创建一个虚拟硬盘，并至少创建一个分区，格式化该分区（建议格式化为 NTFS）。Hello China 会自动把分区的分区标识设置为 C：。

（3）在 C:分区上创建 PTHOUSE 和 HCGUIAPP 两个目录，把 Hello China 的外围模块（比如 GUI 模块）复制到 PTHOUSE 目录下，把应用程序（HCX 文件）复制到 HCGUIAPP 目录下。

这样即可启动 Hello China 了。

Windows 7 操作系统添加了对虚拟硬盘的支持，这就使上述过程变得非常简单。

2.2.2 Hello China 在 Virtual PC 上的安装过程

Hello China 安装包中包含了安装需要的所有文件和辅助工具，主要有：

- 虚拟软盘文件（vfloppy.vfd）。
- Hello China 的内核模块。
- Hello China 的外围模块，比如 GUI 模块等。
- 一些应用程序文件。
- 相关辅助工具。

针对 Virtual PC 的安装包，放在 bin/VirtualPC 目录下。下面在介绍安装步骤的时候，会对上述文件的使用进行说明。

步骤一：下载和安装 Virtual PC 2007

Virtual PC 2007（更高版本也可）可从 Microsoft 的官方网站上下载。可用 Google 搜索 Virtual PC 2007，找到 Microsoft 网站上的具体链接，点击下载即可。该软件完全免费，也无需注册。而且根据作者的经验，下载速度也非常快。

下载后双击即可安装，无需做特殊设置，采用其默认安装设置即可。

步骤二：创建一个虚拟机

这个步骤也简单，运行步骤一中安装的 Virtual PC，在 Virtual PC 的控制台中点击“New...”菜单，即可启动一个虚拟机的创建过程。假设创建的虚拟机名字是“Hello China”，无需做特殊设置，采用默认设置即可。但是在创建虚拟硬盘时，要选择“A new hard disk”，如图 2-2 所示。

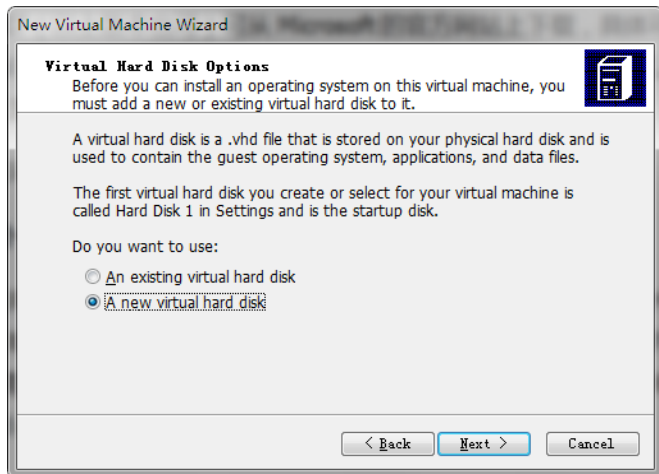


图 2-2 选择创建新的虚拟硬盘

在下一步中，指定虚拟硬盘的存放位置（可采用默认位置）即可。对于虚拟硬盘的空间，可以直接使用默认大小，也可以指定一个数值。若指定数值，建议至少在 256MB 以上。

最后点击 Finish 按钮，即可完成虚拟机的创建。

对于非 Windows 7 操作系统, 由于不能直接支持虚拟硬盘的创建和操作, 因此下列介绍的步骤三、四、五不能适用。读者可直接跳到步骤六, 完成基于字符界面的 Hello China 的安装。

步骤三: 对虚拟硬盘进行分区并格式化

Windows 7 已经支持虚拟硬盘功能。在“我的电脑”上单击右键, 点击“管理”菜单, 启动 Windows 计算机管理工具。在计算机管理工具上, 点击左边导航树中“磁盘管理”选项, 出现磁盘管理界面。点击右面面板的“更多操作”菜单, 可出现虚拟硬盘操作菜单。进一步选择“附加 VHD”菜单, 在显示的对话框中, 选择步骤二中创建的虚拟硬盘文件, 点击“确定”按钮即可。这时 Windows 的磁盘管理器就会把这个虚拟硬盘挂接到磁盘管理界面上了, 如图 2-3 所示。

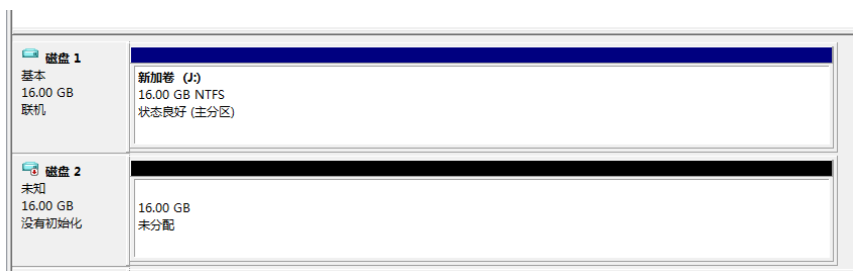


图 2-3 新创建的虚拟硬盘

其中“磁盘 2”就是刚刚挂接的虚拟硬盘。

在上图中“没有初始化”位置处单击右键, 在弹出的菜单中选择“初始化硬盘”, 采用默认设置初始化该硬盘即可。所谓初始化, 指的是操作系统会建立这个硬盘的 MBR 扇区, 并写入相关数据。需要注意的是, 虚拟硬盘刚刚创建完成时, 是没有任何数据的。要使这个虚拟硬盘可用, 必须对之进行初始化。

初始化完成之后, 只是写入了 MBR 扇区, 但是具体分区操作还没有完成。这时候可在初始化后的磁盘上, 单击鼠标右键, 在出现的菜单中, 选择“新建简单卷...”, 采用默认设置, 即可完成虚拟硬盘的分区和格式化工作。

完成格式化后, 虚拟硬盘就可用了。可以看到, 系统会增加一个新的盘符, 这时就可以像使用普通硬盘一样使用虚拟硬盘了。

假设 Windows 7 为新增加的虚拟硬盘分配的盘符是 J:, 本书后续部分将会以 J: 为标识引用该虚拟硬盘。

步骤四: 准备 Hello China V1.75 安装目录

把 Hello China 软件包中 bin 目录下的 VirtualPC 目录, 复制到任意一个本地硬盘 (注意, 这里不是虚拟硬盘) 上, 然后再把 VirtualPC 目录下的 install 目录复制到虚拟硬盘上 (即 J: 盘)。

步骤五: 在虚拟硬盘上安装 Hello China

以 DOS 命令行方式进入 J: 盘的 install 目录, 运行 batch.bat 文件, 即可完成 Hello China 在虚拟硬盘 J: 上的安装。Batch.bat 文件主要是创建了 HCGUIAPP 和 PTHOUSE 两个目录, 然后把相关文件复制到了这两个目录下, 同时也复制了一些相关文件到 J: 盘的根目录下。其中 PTHOUSE 目录中存放的是 Hello China 的外围功能模块, 比如 GUI 模块、网络模块等。



而 HCGUIAPP 目录中存放的是基于 GUI 模块的 Hello China 应用程序，即 HCX 文件。

步骤六：将虚拟软驱与虚拟机进行关联

由于 Hello China 是使用虚拟软盘启动虚拟机的，最后一步就是把虚拟软盘文件（vfloppy.vfd）与虚拟机进行关联，该文件在 VirtualPC 目录下。在 Virtual PC 的控制台中，双击启动步骤二中创建的虚拟机。由于没有可启动的设备，虚拟机无法正常启动。此时用鼠标把 vfloppy.vfd 文件拖到虚拟机窗口下面的软驱图标上，就实现了虚拟软驱与虚拟机的关联。

重新启动虚拟机，正常情况下应该可进入 Hello China 的字符模式了。

步骤七：验证 Hello China 是否成功安装

完成上述步骤之后，虚拟机应该可以正常启动，并进入 Hello China 的字符操作界面了。在字符界面下执行 help 命令，可看到一些帮助信息。运行 version 命令，可看到 Hello China 的版本信息，如图 2-4 所示。

在字符模式下，运行 fs 命令，进入文件系统操作程序。执行 fslist，应该可以看到 C: 分区，如图 2-5 所示。

```

Action Edit CD Floppy Help
[system-view]help
The following commands are available currently:
version      : Print out the version information.
memory       : Print out current version's memory layout.
sysinfo      : Print out the system context.
sysname      : Change the system host name.
help         : Print out this screen.
support      : Print out technical support information.
runtime      : Display the total run time since last reboot.
memview      : View a block memory's content.
kinfo        : View all the kernel threads' information.
ioctrl       : Start IO control application.
sysdiag      : System or hardware diag application.
fs           : File system operating application.
fdisk        : Hard disk operating application.
loadapp      : Load application module and execute it.
gui          : Load GUI module and enter GUI mode.
reboot       : Reboot the system.
cls          : Clear the whole screen.
[system-view]version
Hello China (Version 1.750, build in 2011/12/13, by Garry.Xin)
[system-view]
```

图 2-4 图形模式操作界面

```

[system-view]fs
#fslist
fs_id      fs_label      fs_type
-----
C:         NTFS_UOL      0
#_
```

图 2-5 默认的文件分区

输入“exit”，退出 fs 程序，然后输入 gui 命令，即可进入图形模式，显示所有图形应用程序。点击任何一个应用程序即可运行。比如点击“CPI 统计”程序，运行结果如图 2-6 所示。



图 2-6 图形模式下 CPI 统计程序运行结果

在图形模式下，输入“Ctrl+Alt+Del”组合键（虚拟机下，可点击 Action 菜单，选择该组合键），即可退到字符模式。

2.3 Hello China 在物理计算机上的安装

作为一个功能比较完善的操作系统，Hello China V1.75 可以直接安装在物理计算机上，并能够通过各种程序操作和管理物理计算机。安装方式有两种：

（1）自行启动安装方式：由 Hello China 自行格式化硬盘，并把操作系统相关文件安装到计算机的硬盘上。这时候硬盘上的原有数据会丢失。

（2）借助 Windows 的引导程序进行启动的安装方式：这时候计算机的硬盘上必须成功安装 Windows 操作系统（Windows 2000 以上版本），Hello China 的安装程序只是复制操作系统核心文件到硬盘上，与普通的文件类似。同时需要修改 Windows 操作系统的引导配置文件，以引导 Hello China。这种方式对原有操作系统和硬盘数据不会造成影响。

大多数情况下，建议选择第二种方式进行安装。这种方式非常简单且风险很小，与安装一个普通的 Windows 应用程序类似。但需要在安装前，正确识别所安装硬盘的分区格式（FAT32 或 NTFS），并根据分区格式选择合适的安装包。同时，由于 Windows Vista 以上版本的操作系统与 Windows XP 等版本的操作系统的启动管理方式有所不同，因此还需要针对不同的 Windows 版本做不同的引导配置。

下面以第二种方式为例，详细介绍 Hello China 在不同版本 Windows 和不同硬盘分区类型上的安装。

2.3.1 安装注意事项

在安装前，请注意以下事项，以确保顺利安装。

（1）确保计算机的 BIOS 支持 INT 13H 扩展，Hello China 在启动时会通过这个扩展调用加载核心模块。且其文件系统在访问硬盘时，也是通过这个扩展实现的。需要说明的是，目前大部分计算机的 BIOS 会支持这个扩展。

（2）确保系统分区上没有 BOOTSECT.DOS 文件。如果用户的计算机上安装了多个操作系统（比如 Windows XP 和 DOS），则可能存在 BOOTSECT.DOS，这时候千万不要安装 Hello China。因为 Hello China 会覆盖 BOOTSECT.DOS 文件，导致原来的 DOS 系统无法启动。当然，可以把原有的 BOOTSECT.DOS 文件修改为其他名字再安装，这样在需要的时候，很容易恢复。

（3）为确保安全，Hello China V1.75 版本禁止了硬盘的写入功能，只提供了硬盘的读取功能。使用 FS 程序，可浏览计算机上的文件系统，但是不能修改或格式化硬盘。

（4）Hello China V1.75 提供了图形界面功能（执行 gui 命令，可进入 GUI 界面），但是目前尚不支持 USB 接口鼠标。如果用户使用的是 USB 接口的鼠标，则不能操作其 GUI 界面，这时可按下“Ctrl + Alt + Del”组合键退出 GUI 界面。

2.3.2 在 Windows XP 操作系统上的安装

（1）首先判断待安装硬盘的分区格式。如果是 FAT32，则使用 bin/FAT32 目录下的安装



包，如果是 NTFS 文件系统，则使用 bin/NTFS 目录下的安装包。

(2) 把安装包下的文件复制到 C: 盘（严格说应该是 Windows 的系统分区，大多数情况下是 C: 盘，但也有可能是其他分区，这时候就要改变安装分区）的一个任意目录下，比如 hcninst 目录。

(3) 进入 DOS 命令行模式，并定位到上述目录，执行 batch 即可。batch 是一个批处理文件，该文件直接调用了安装目录下的相关工具生成内核，并复制到根目录下。同时在 C: 盘上创建了 PTHOUSE 目录，用于存放 Hello China V1.75 版本的二进制系统模块和二进制应用模块。如果在安装中遇到问题，比如不能安装成功，可再次运行 batch 程序，把结果定向到一个文本文件中（比如“batch > result.txt”，则输出结果将存放在 result.txt 文件中），然后发给本书作者帮用户诊断。

(4) 修改根目录下的 boot.ini 文件，增加下列一行：

```
C:\BOOTSECT.DOS="Hello China V1.75"
```

同时确保启动等待时间（boot.ini 中的 timeout 值）足够长，比如 30s。

比如，原始的 boot.ini 文件可能如下：

```
[boot loader]
timeout=0
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=opt in
/fastdetect
```

修改后的 BOOT.INI 文件如下：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=opt in
/fastdetect
C:\BOOTSECT.DOS="Hello China V1.75"
```

注意黑体部分是修改或增加的内容。

如果 boot.ini 文件不允许修改（因为该文件是系统文件，增加了只读保护），可通过执行 attrib 命令去掉只读属性：

```
attrib -R boot.ini
```

然后再进行修改，修改完成后重新启动计算机，即可看到操作系统引导列表里出现了“Hello China V1.75”的选项，选择并按下回车键，正常情况下即可进入 Hello China 的字符操作界面。

2.3.3 在 Windows 7 操作系统上的安装

Windows 7 操作系统的启动管理程序配置方式与 Windows XP 等不同。XP 等操作系统是由一个叫做 boot.ini 的文本文件对可引导的操作系统进行管理的，而 Windows 7 等则是通过

一个命令程序 bcdedit 对引导程序进行管理。下面讲解 Windows 7 上的安装过程。

(1) 首先判断待安装硬盘的分区格式。如果是 FAT32, 则使用 bin/FAT32 目录下的安装包, 如果是 NTFS 文件系统, 则使用 bin/NTFS 目录下的安装包。

(2) 把安装包下的文件复制到 C: (严格说应该是 Windows 的系统分区, 大多数情况下是 C: 盘, 但也有可能是其他分区, 这时候要就要改变安装分区) 盘的一个任意目录下, 比如 hcninst 目录。

(3) 进入 DOS 命令行模式, 并进入上述目录, 执行 batch 即可。batch 是一个批处理文件, 该文件直接调用了安装目录下的相关工具生成内核, 并复制到根目录下。同时在 C: 盘上创建了 PTHOUSE 和 HCGUIAPP 目录, 用于存放 Hello China V1.75 版本的二进制系统模块和二进制应用模块。如果在安装中遇到问题, 比如不能安装成功, 可再次运行 batch 程序, 把结果定向到一个文本文件中 (比如 “batch > result.txt”, 则输出结果将存放在 result.txt 文件中), 然后发给本书作者帮用户诊断。

(4) 使用 bcdedit 命令, 对 Vista 或 Windows 7 的系统加载器进行配置, 具体过程如下。

1) 运行 cmd, 进入命令行界面。

2) 运行命令: bcdedit /create /d "Hello China V1.75" /application bootsector, 完成后会生成一个 GUID, 其中 “Hello China V1.75” 可以修改为任意内容, 如图 2-7 所示。

```
C:\>bcdedit /create /d "Hello China V1.75" /application bootsector
项 {9cebaca8-4dc0-11df-8cf7-d93e49e38653} 成功创建。
```

图 2-7 创建一个引导项

生成的 GUID 内容 (即图 2-7 中大括号内的十六进制数字串) 会不同, 但只要提示成功即可。注意, 这个生成的 GUID 在后续命令中会用到, 因此要记录或复制下来。

(5) 运行命令: bcdedit /set {9cebaca7-4dc0-11df-8cf7-d93e49e38653} device partition=C:, 注意大括号中的内容就是上面/create 命令生成的 GUID。这条命令告诉 Windows 7, 启动扇区位于 C: 盘上, 如图 2-8 所示。

```
C:\>bcdedit /set {9cebaca7-4dc0-11df-8cf7-d93e49e38653} device partition=C:
操作成功完成。
```

图 2-8 设置启动分区

(6) 执行命令: bcdedit /set {9cebaca7-4dc0-11df-8cf7-d93e49e38653} path \bootsect.dos, 大括号中的内容仍然是上述 GUID。该命令告诉 Windows 7, 引导扇区文件名字是 bootsect.dos; 如图 2-9 所示。

```
C:\>bcdedit /set {9cebaca7-4dc0-11df-8cf7-d93e49e38653} path \bootsect.dos
操作成功完成。
```

图 2-9 设置引导扇区文件

(7) 执行命令: bcdedit /displayorder {9cebaca7-4dc0-11df-8cf7-d93e49e38653} /addlast, 告诉 Windows 7, 把新增加的项添加到启动列表的最后。如图 2-10 所示。

```
C:\>bcdedit /displayorder {9cebaca7-4dc0-11df-8cf7-d93e49e38653} /addlast
操作成功完成。
```

图 2-10 把新增加的启动项添加到列表最后

上述步骤执行完之后，重新启动计算机，就可以看到新增加的引导项了。这时候选择该引导项，并按下回车键，即可引导 Hello China。

2.4 Hello China 的卸载

如果采用方式二（借助 Windows 的引导程序启动 Hello China）在物理计算机上安装 Hello China，则很容易卸载。下面简单描述其卸载步骤。

- (1) 删除启动文件（boot.ini）或启动管理器（bcdedit 命令）中 Hello China 的对应项目。
- (2) 删除安装分区根目录上的 bootsect.dos 和 hcnimg.e.bin 文件。
- (3) 删除安装分区根目录上的 hcnguiapp 目录和 pthouse 目录。

如果用户采用的是第一种方式（完全格式化硬盘安装），则无法删除了。这与在硬盘上直接安装 Windows 操作系统一样，必须重新安装其他的操作系统覆盖之。

2.5 Hello China 的使用

安装成功之后，Hello China 的操作就比较简单了，在字符模式下，运行 help 命令，即可查看支持的所有命令，如图 2-11 所示。

```
[system-view]help
The following commands are available currently:
version      : Print out the version information.
memory       : Print out current version's memory layout.
sysinfo      : Print out the system context.
sysname      : Change the system host name.
help         : Print out this screen.
support      : Print out technical support information.
runtime      : Display the total run time since last reboot.
menview      : View a block memory's content.
ktview       : View all the kernel threads' information.
ioctrl       : Start IO control application.
sysdiag      : System or hardware diag application.
fs           : File system operating application.
fdisk        : Hard disk operating application.
loadapp      : Load application module and execute it.
gui          : Load GUI module and enter GUI mode.
reboot       : Reboot the system.
cls          : Clear the whole screen.
[system-view]version
Hello China [Version 1.750, build in 2011/12/13, by Garry.Xin]
[system-view]
```

图 2-11 字符模式 help 命令运行结果

表 2-1 列出了 V1.75 版本提供的主要命令及其功能。

表 2-1 字符模式主要命令

命令或程序	用 途
help	Shell 或任意程序提示符下，输出帮助信息
cls	Shell 模式下，清除屏幕内容
version	Shell 模式下，输出当前版本

(续)

命令或程序	用 途
support	输出技术支持信息
ioctl	输入/输出端口和设备寄存器控制应用，主要用于调试
sysdiag	系统诊断程序，查看系统运行状态信息
fdisk	硬盘分区和格式化程序，完成硬盘分区和格式化功能
fs	文件系统操作程序，完成文件系统的常规操作
gui	进入图形用户接口（GUI）模式
reboot	重新启动计算机，在字符模式下输入“Alt + Ctrl + Del”组合键，也可重新启动计算机
loadapp	从应用程序目录中加载一个应用程序
runtime	显示操作系统自启动以来的运行时间
hypertrm	超级终端模拟程序，实现超级终端功能
hedit	Hello China 内置的文本编辑器

需要注意的是，有些程序提供了进一步的子命令，比如 `fs`、`sysdiag` 等。这些程序进入后，会改变默认的提示符。在程序提示符下，输入 `help` 命令，即可看到程序提供的功能子命令及其功能描述。

2.6 内核的编译和生成

本节对 Hello China 内核模块的开发环境进行简要描述。这里的内核模块，主要是指 Hello China V1.75 的内核和 GUI 两个模块，分别对应源代码的[/kernel]目录和[/gui]目录。需要注意的是，随本书一起发布的 Hello China V1.75 版本的源代码，所有内核模块都包含了已经设置好的 Visual C++ 工程文件。读者只需直接打开工程文件，即可加载内核工程，无需单独进行设置。

2.6.1 Hello China 内核的开发环境

操作系统的开发涉及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：

（1）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。

（2）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。

（3）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++ 编写了开发辅助工具，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。详细内容见本书第 14 章。

Hello China 在开发过程中涉及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。

2.6.2 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、链接。其中, NASM 完成汇编语言实现的模块的编译和链接, 其他由 C 语言完成的模块, 则由 VC 编译和链接。

1. 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。

所有[/kernel/arch/sysinit]目录下的汇编语言文件, 都是按照上述方式编译的。

2. 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 作为编译链接工具。下列步骤描述了 Hello China 开发过程中开发环境的搭建。

步骤一: 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作 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, 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++中创建一个 DLL 工程, 如图 2-12 所示。

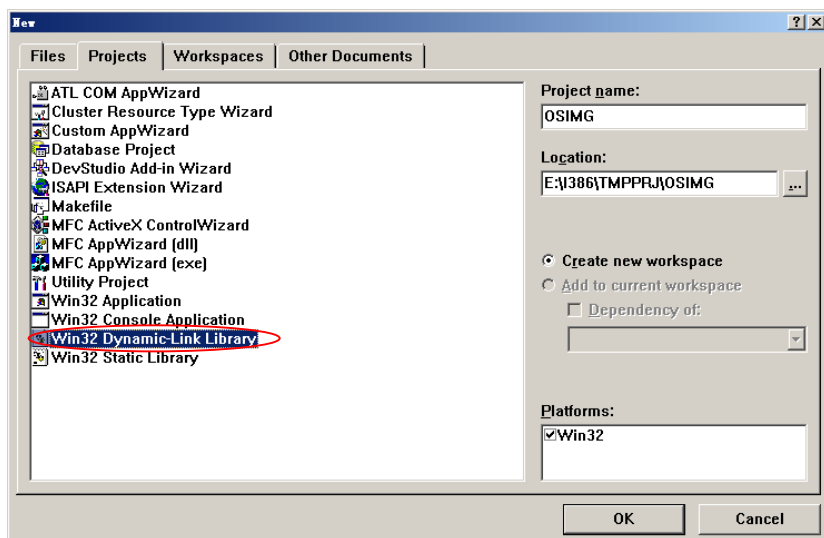


图 2-12 创建一个 DLL 工程

步骤二：设置项目编译与链接选项

一般情况下，需要对创建的工程设定如下编译链接选项。

(1) 对齐方式，在项目选项中，添加/ALIGN:XXXX 选项，告诉链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，需要设置为与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，则在链接工程选项里面添加/BASE:0x00100000 选项。针对 master 模块，其加载地址是 0x00110000，因此 base 应该配置为 0x00110000。

(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，采用缺省的入口地址，有时候不能正确地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，请参考附录。

上述所有的设置，如图 2-13 所示。

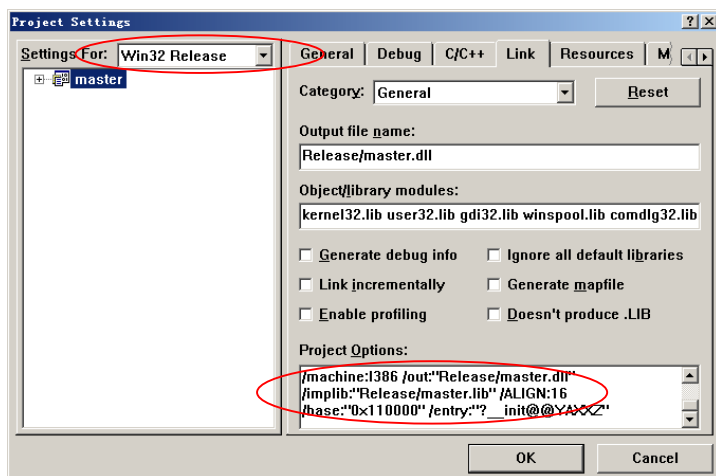


图 2-13 设置编译、链接选项

在 Visual C++ 6.0 中，上述对话框可以从“project→settings...”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（图中左上角椭圆中注明的地方）。

步骤三：编辑源文件，编译链接

上述步骤完成之后，集成开发环境就设置好了，剩下的工作就是直接在该工程中添加 C 源代码文件，完成编码工作。编码完成之后，即可编译链接该项目了。需要注意的是，在编译的时候，要使用 Release 方式编译，即选择菜单 build→Batch build，在弹出的对话框中选择 Release 选项，如图 2-14 所示。

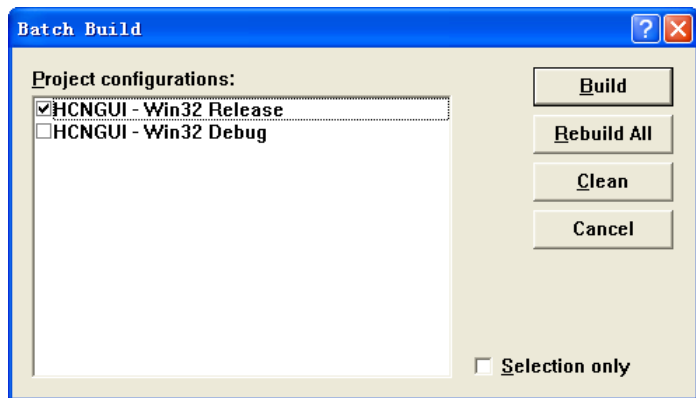


图 2-14 使用 Release 方式构建项目

然后单击 Rebuild All 按钮即可。

3. 编译后模块的进一步处理

上述开发环境搭建完毕，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预处理。预处理的目的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++ 编译链接而成的目标文件（Release 版本的 DLL 文件）复制到与 process.exe 相同的目录下，其中，process.exe 就是预处理程序。

(2) 运行 process.exe 程序修改上述 DLL 文件，比如：process -i master.dll -o master.bin，即对 master.dll 文件进行修改处理，处理结果即为 master.bin 文件，其中 master.dll 文件内容保持不变。修改之后，master.bin 模块就可以直接加载到内存中运行了。

2.6.3 内核映像文件的生成

Hello China V1.75 PC 版的操作系统内核映像（即二进制可执行文件），是由下列几个文件组成的。

(1) 引导扇区，针对不同的文件系统或存储设备，分别有不同的文件。针对 NTFS 文件系统，引导扇区由 [kernel/arch/sysinit/ntfsbs.asm] 文件编译而成，命名为 bootsect.dos，针对 FAT32 文件系统，引导扇区由相同目录下的 hdbbs.asm 编译而成，也是命名为 bootsect.dos。但是针对软盘，则是由同一目录下的 bootsect.asm 文件编译而成的。根据安装的目标存储设备的不同，Hello China 安装程序会选择合适的引导扇区文件。需要注意的是，引导扇区是平台特定的，也是唯一需要根据平台选择的模块。所有其他模块都是相同的。

(2) 实模式初始化模块 realinit.bin，这是由 [kernel/arch/sysinit/realinit.asm] 文件编译而成的，用于完成实模式下的初始化工作。

(3) 微小内核 miniker.bin，由相同目录下的 miniker.asm 文件编译而成。这是个历史遗留模块，在 V1.75 版本中，大多数功能已被剥离，但是仍有一些初始化工作，比如中断描述符表的初始化工作，是在这个模块内完成的。

(4) 内核文件 `master.bin`, 这个模块完全是由 C 语言编写, Visual C++ 集成开发环境编译, 所有操作系统相关的核心机制, 都是在这个模块内完成的。

上述模块是支撑 Hello China 正常启动的基础。只有上述模块全部加载并正常运行后, 才能进一步加载其他的模块, 比如 GUI、网络、各类应用程序等。为了方便, Hello China 通过工具, 把上述几个模块链接在一起, 形成一个物理的文件。这样操作系统在启动的时候, 只需要加载这个链接在一起的物理文件即可, 无需加载四个文件。下面是生成这个操作系统内核的批处理命令:

```
[/bin/ntfs/batch.bat]
process -i master.dll -o master.bin -----(1)
append -s realinit.bin -a miniker.bin -b 2000 -o image_1.bin -----(2)
append -s image_1.bin -a master.bin -b 12000 -o image_2.bin -----(3)
ren image_2.bin hcnimge.bin -----(4)
del image_1.bin -----(5)
copy hcnimge.bin c:\ -----(6)
mkntfsbs -----(7)
copy bootsect.dos c:\ -----(8)
.....
```

下面分别说明以上步骤。需要注意的是, 各个工具的原理和使用方式, 请参考第 14 章。

(1) 调用 `process` 工具, 对编译后的 `master.dll` 文件进行处理, 形成可直接加载和跳转运行的模块。

(2) 第 (2) 和第 (3) 条命令, 使用 `append` 工具把 `realinit.bin`、`miniker.bin` 和 `master.bin` 链接在一起, 生成一个统一的内核文件。由于 `append` 工具的限制, 需要生成一些中间过程文件, 比如 `image_1.bin`、`image_2.bin` 等。

(3) 与第 (2) 相同。

(4) 把最终合并形成的二进制模块, 命名为 `hcnimge.bin`, 这是操作系统的内核文件。引导扇区在加载操作系统的时候, 也是搜索这个文件并加载的。

(5) 删除中间过程文件。

(6) 把操作系统内核文件复制到根目录。

(7) 调用 `mkntfsbs` 工具, 生成针对 NTFS 文件系统的引导扇区, 即修改 `bootsect.dos` 文件。

(8) 把 `bootsect.dos` 文件复制到 C: 盘。

当然, 这个批处理文件 (`batch.bat`) 还完成了其他一些安装工作, 比如创建了系统目录 (`pthouse` 目录), 复制了 GUI 等外围模块和自带的应用程序等。

把 `bootsect.dos` 和 `hcnimge.bin` 两个文件复制到根目录下, 并修改 Windows 的 `boot.ini` 文件 (针对 Vista 以上版本, 需要使用 `bcdedit` 修改加载管理器)。这样 Windows 操作系统的引导程序, 即可把 `bootsect.dos` 读入内存, 然后 `bootsect.dos` 加载 `hcnimge.bin` 文件, 从而完成系统的加载。

需要注意的是, 上述批处理程序和相关工具, 都已默认放在 `[/bin/ntfs]` 目录下。如果读者希望修改内核, 则只需修改相关的源文件, 并编译生成二进制文件, 然后复制到该目录下, 直接运行 `batch` 即可。



但是在 Virtual PC 上, Hello China V1.75 是用虚拟软盘启动的。与安装在物理计算机上不同, 有另外一个工具 `vfmaker`, 可以直接把几个核心模块 (包含引导扇区 `bootsect.bin` 模块) 打包成一个虚拟软盘文件 (`vfloppy.vfd`)。因此如果读者希望在 Virtual PC 上开发, 则只需要把修改后的二进制模块复制到 `[bin/VirtualPC]` 目录下, 然后运行 `vfmaker` 工具, 即可生成启动虚拟机的虚拟软盘文件。

如果读者希望自行修改 Hello China 的内核, 建议在虚拟机下进行调试。具体方法如下。

(1) 对核心模块进行修改和编译, 如果是对 `realinit.bin` 和 `miniker.bin` 进行修改, 则使用 NASM 编译, 如果是希望对 `master.bin` 模块进行修改, 则使用 VC 集成开发环境。

(2) 如果是对 `master.bin` 模块进行了修改, 则需要使用 `process` 工具对编译后的 DLL 文件进行预处理。其他模块则无需预处理。

(3) 把最终的二进制模块复制到一个目录下, 同时把 `vfmaker` 工具也复制到该目录下, 然后运行 `vfmaker` 工具, 即可生成虚拟软盘文件 `vfloppy.vfd`。

(4) 使用最新生成的 `vfloppy.vfd` 文件, 重新启动虚拟机, 即可看到修改后的结果。

这样在虚拟机上进行调试, 可大大减轻开发工作量, 因为无需重复启动物理计算机。在虚拟机上调试通过后, 再用实际的物理计算机做进一步验证。

第 3 章 Hello China 的引导和初始化

3.1 概述

本章将详细介绍操作系统的引导初始化过程，包括通用操作系统（比如 Windows 和 Linux 等）的引导和初始化过程、嵌入式操作系统的引导和初始化过程。这些内容都是通用的，并不针对某个具体的操作系统。

之后介绍 Hello China 操作系统的引导和初始化。希望读者阅读本章后，能够以 Hello China 为实例，举一反三，理解各类操作系统的初始化和加载过程。引导和初始化过程，可以看作整个操作系统运行生命周期的童年阶段。所谓“三岁看大，七岁看老”，通过对操作系统初始化阶段的分析，即可大概知道操作系统本身的特点和不足。而且初始化过程是操作系统整个运行周期中最容易出问题的地方，理解了初始化过程，对计算机专业人员来说，可以更好地排除系统级故障。

接下来正式进入主题。首先看一下通用操作系统的引导和初始化过程。

3.2 个人计算机的引导和初始化

3.2.1 BIOS 的引导工作

BIOS（Basic Input and Output System，基本输入/输出系统）的功能和作用，在这里就不赘述了，这是阅读本书最基本的铺垫内容，比掌握 C 语言还要基本。如果读者不理解 BIOS 的功能和作用，那么建议先不要阅读后续内容，先补习一下 BIOS 相关的知识，否则将无法阅读。

按照个人计算机（PC）的硬件标准，引导环节发生在计算机的硬件系统检测完成之后。具体的引导工作，是由 BIOS 完成的。BIOS 维持一个可用于引导计算机的硬件设备列表，比如本地硬盘、本地光驱、网络、USB 接口设备等，然后做一个排序。BIOS 会试图从整个序列的第一个设备开始，检查其状态和引导能力。比如针对光驱，首先会判断光驱中是否存在光盘，如果不存在，则跳过光驱设备，进入下一个设备的检测过程。如果发现光盘存在，则试图读取光盘的第一个扇区，并检查这是否是一个可引导扇区（通过检查扇区的最后两个字节是不是 0x55AA）。如果发现不是一个可引导扇区，则也跳过光盘，再检查引导序列中的下一个设备，直到发现一个可引导的设备为止。如果遍历完整个引导设备列表，未找到任何可引导设备，则引导过程失败，BIOS 会提示无法找到可启动设备。如果在这个过程中能够找到一个可引导扇区，则 BIOS 会把该扇区的内容加载到内存，并跳转到该扇区，执行引导代码。这个跳转指令，就是 BIOS 程序在计算机启动过程中执行的最后一条指令，至



此，BIOS 的工作结束。后续工作将由引导扇区代码完成。

由此可见，BIOS 在计算机引导过程中的角色是非常重要且复杂的。但是由于其独立于操作系统的实现，不是本书重点分析的对象。本章重点分析引导扇区被加载到内存，并开始执行后的过程。这个过程属于操作系统的作用范畴，而且是操作系统生命周期的最开始部分。本章将以硬盘为例，首先分析硬盘的逻辑结构和主引导记录（MBR）的作用，然后再分析不同的文件系统下，操作系统引导扇区的功能差异和实现差异，以及每种实现的功能缺陷。最后给出一种克服了这些缺陷的引导程序制作方法——预置引导法，并以 Hello China 操作系统的引导程序为实例，说明这种方法的应用。

3.2.2 硬盘逻辑结构及引导扇区的功能

可以在逻辑上把硬盘理解为一个线性数组，这个数组的元素就是扇区。每个扇区按照其在硬盘上的位置进行编号，第一个扇区就是大名鼎鼎的 MBR（Master Boot Record，主引导区）。

为了方便管理，一个硬盘可以划分为若干个逻辑分区，每个分区占据了硬盘上的一部分连续的存储空间。一般情况下，分区在硬盘上的位置和大小，可用其在硬盘上的起始扇区号、扇区数量进行描述。目前的实现是，一个物理硬盘，最多可以分成四个分区，分区的位置、大小、属性等信息，记录在一个只有四个元素的线性表里，这个表就是分区表。

分区表就存放在 MBR 中，且在 MBR 中的固定位置处。因为一般情况下，硬盘的每个扇区的容量是 512B，如果分区表在 MBR 中的位置不固定，就无法明确读出分区表的内容。但分区表却不是 MBR 的唯一内容，MBR 还存放了引导扇区代码。BIOS 在发现一个可引导的硬盘之后，会把 MBR 的代码读入内存，然后跳转到开始处执行。需要说明的是 MBR 的代码一般是与操作系统无关的。这可能会引起一些疑惑，下面稍作解释。

操作系统是安装在硬盘分区上的，至少到目前为止，尚未见有哪个操作系统，不对硬盘进行分区（或者说把整个硬盘看做一个分区），而直接安装。而操作系统本身特定的引导代码，是放在其所在分区的第一个扇区上的。比如，一个硬盘被划分成了两个分区，一个分区的起始扇区号是 2（MBR 的扇区号是 1），扇区数量是 M。另外一个分区的起始扇区号是 M+2，扇区数量是 N。假设第一个分区上安装的是 Windows 操作系统，第二个分区上安装的是 Linux 操作系统。这样，Windows 操作系统的引导扇区，是第 2 个物理扇区（第一个分区的第一个扇区），而 Linux 的引导扇区，则是第 M+2 个物理扇区。显然，这两个（第 2 个和第 M+2 个）引导扇区是与操作系统强相关的。

但 BIOS 最初读入的是 MBR。一旦跳入 MBR 执行，BIOS 就撒手不管了。这时候 MBR 上的代码，必须能够找到 Windows 或 Linux 的引导扇区，并把它们读入内存中，完成相应操作系统的引导。因此 MBR 的功能代码本身需要完成两个问题的决策：

（1）在硬盘上有多个分区，每个分区都有可能安装操作系统的情况下，如何选择一个分区进行继续引导？

（2）如果确定了一个分区，如何得到这个分区的引导扇区的物理位置（即扇区在整个物理磁盘上的编号）？

显然，在第一个问题确定的情况下，第二个问题很容易解决。因为这时候分区编号已经

确定，只要读取分区表，并找到对应的记录，就可读出该分区的第一个扇区的编号，这就是引导扇区。对于第一个问题，是通过在分区表中设置一个活动标志解决的。每个分区表项（对应一个分区）中，都有一个字节，叫做活动标志，且四个分区表项中，必须只有一个分区的活动标志被设置，这个分区就是活动分区。安装在这个分区上的操作系统会被引导。具体是由谁来设置这个标志呢？答案是操作系统。在安装操作系统的时候，会让用户选择要安装的分区（比如 Windows，会列出硬盘的分区情况，用 C:、D: 等盘符表示）。一旦选定一个分区，Windows 安装程序就会把该分区对应的分区表项的活动标志设置为 1，同时清除其他分区表项的活动标志。

按照这种规则，最后安装的操作系统，往往会“压制”以前安装的操作系统，使它们无法引导。但只要知道了这个过程，就可以利用一些工具，改变这种情况。比如用户最后安装的是 Windows 操作系统，在启动 Windows 后，可以通过运行在 Windows 上的工具，把 Linux 操作系统所在分区修改为活动分区。这样在下次启动的时候，Linux 就会被引导。但是为了安全起见，操作系统都提供了对 MBR 的保护功能，不能直接写入 MBR。这样就很麻烦了，一旦成功安装了 Windows，就意味着原有的 Linux（与新安装的 Windows 在不同分区）不能直接引导了，除非借助于 Windows 的引导程序进行引导。但这也不是绝对的，比如可以通过光盘引导的操作系统，来修改 MBR，或者可以在 Windows 分区上安装 Hello China，由 Hello China 帮用户完成修改活动分区标志的工作。

再回到原来的话题，MBR 根据活动标志选择一个分区，并根据分区表中的分区起始位置确定操作系统引导扇区，然后把该扇区读入内存内的一块连续空间内，并跳转到该空间的起始位置处继续执行。由此可见，MBR 本身，是与特定操作系统无任何关系的。实际上，MBR 扇区的可执行代码，自从 DOS 开始，就一直没有改变过。

一个物理硬盘最多有四个分区，某些软件公司，比如微软，认为这是不够的。于是又发明了一种技术，叫做扩展分区。扩展分区本质上就是一个硬盘分区（四个分区之一），但是在此基础上，又进行进一步扩展，把一个分区再进行细分，分为更小的分区（这些更小的分区叫逻辑分区），即分区套分区的结构。与硬盘的 MBR 一样，逻辑分区的相关信息，被记录在扩展分区的第一个扇区上。但是与普通分区不一样的是，扩展分区上的逻辑分区是不能引导操作系统的，即操作系统不能安装在逻辑分区上。既然操作系统不能安装在逻辑分区上，那么肯定也不能安装在扩展分区上了。因为扩展分区是逻辑分区的“容器”，其本身没有空间存储操作系统文件。因此要安装操作系统，必须安装在普通的磁盘分区上。为显示这种“容纳”操作系统的特性，普通分区又被冠以“主要分区”的称号。

至此，下列一些概念或原理，读者应该清楚：

- MBR 和分区表。
- MBR 上的代码与操作系统无关。
- 操作系统特定的引导扇区，是其所安装分区的第一个扇区。
- 分区活动标志。
- 主要分区、扩展分区、逻辑分区。
- MBR 与操作系统引导扇区的关系。



3.2.3 操作系统引导扇区的功能和局限

接下来我们把目光转移到操作系统引导扇区上，即操作系统所在分区的第一个扇区。操作系统引导扇区与操作系统密切相关，其主要功能就是，在操作系统分区上，找到引导操作系统内核相关的文件，完成操作系统的加载。这里说的“引导操作系统相关的文件”，既可能是操作系统核心模块，也可能是为进一步引导操作系统核心模块而作准备的一些可执行代码。毕竟现代操作系统十分复杂，核心模块很大，无法直接完成引导。这样就可能有一些辅助的引导模块，毕竟引导扇区代码的功能是十分有限的。

但不论如何，引导扇区的代码必须在操作系统所在分区的文件系统里找到一个模块（实际上是一个文件），并加载到内存。因此下列两项功能是引导扇区代码的核心：

(1) 在分区的文件系统上，找到一个特定文件。

(2) 把这个文件装入内存，并跳转执行。

第二项工作比较容易，一般情况下，引导扇区还是运行在 CPU 的实模式下（以 PC 为例），可以调用 BIOS 提供的磁盘读写服务，很容易把文件读入内存。关键是第一项工作，如何在一个文件系统里搜索到一个特定的文件。实际上“搜索到一个文件”也不是关键，关键是如何以“一个扇区”的代码、在一个复杂的文件系统里找到一个需要的文件。这里的两个因素，形成一对矛盾：

(1) 引导扇区代码尺寸有限，比如只有 512B，无法适应复杂的处理要求。

(2) 文件系统结构复杂，访问文件系统所需要的代码量很大，一个扇区无法容纳。

引导扇区需要很好地平衡这对矛盾，以便引导过程能够继续。一般情况下，有下列几种方式解决这个问题。

(1) 扩展引导扇区大小。一般认为，操作系统引导扇区是 512B，操作系统的初始引导完全是由这 512B 代码完成的。实际上不然，很多操作系统，比如 Windows，已经大大扩展了引导扇区尺寸。既然整个分区都是操作系统的地盘，那么每个扇区怎么使用，就完全由操作系统决定了。引导扇区是分区的第一个扇区，这个不能变。那么完全可以把第二个、第三个……第 N 个扇区也作为引导扇区。第一个扇区只是作为跳板，MBR 把第一个扇区读入内存并运行后，第一个扇区再把后面连续的一片扇区读入内存，这些所有的扇区共同组成引导模块。这样无论文件系统多么复杂，只要多分配几个扇区，就可以容纳访问文件系统的代码了。显然，这种方式很有效，且被广泛采用。比如 Windows 操作系统，在 NTFS 文件系统上的引导扇区，就有 16 个（0~15 号扇区），有 8KB 的代码空间。这对分析 NTFS 文件系统并读取引导文件（比如 NTLDR），就足够了。

(2) 引导扇区维持一个不变，但是固定操作系统核心或相关文件在磁盘上的位置。操作系统可以把引导相关的文件固定在磁盘的一个特定位置上，比如 1024 号扇区位置处。这样引导扇区就无需分析文件系统了，直接从 1024 号扇区处读取操作系统核心就行了。这种策略在 DOS 时代似乎被用过。记得我在上大学的时候，制作了一张 DOS 启动软盘，上面有 IO.SYS 等文件。有一次把 IO.SYS 文件删除了，重新复制了一个进去，结果就不能启动 DOS 了。据此推测，DOS 可能把 IO.SYS 等固定在了软盘的某个位置。一旦删除再复制，其位置变化了，就导致不能引导。

上述两种解决方案都有局限。第一种方案，在一个分区上只安装一个操作系统，或者安

装相同软件厂商的不同操作系统时是有效的，但如果安装两个不同厂商的操作系统，则可能就会有问题。假设操作系统 A 和 B 都使用的是第一种策略，A 先安装到分区上。在第二个操作系统 B 安装的时候，为了能同时引导 A 系统，B 会把 A 的引导扇区（第一个扇区）备份到一个文件里（比如 Windows 系统下的 BOOTSECT.DOS），然后把自己的引导扇区（可能连续几个）写入分区的开始处。这样 A 操作系统的第一个以外的引导扇区，都被 B 覆盖了。显然，这时候是无法再次引导 A 系统的。但是如果 A 和 B 是同一个软件公司的产品，比如 Windows 98 系列和 Windows 2000/XP 系列，由于都是微软开发，对引导扇区的结构和数量都是已知的，这样在 Windows 98 上安装 XP 或其他更高版本的操作系统时，更高版本的系统（比如 XP）就会把原有系统的所有引导扇区都统一打包到一个文件里，这样就不会出现覆盖问题。如果这种打包仍然有问题（比如第一个扇区会调用 BIOS 服务读取后续扇区，即使打包了，也无法改变这种动作），XP 甚至会修改打包后的引导扇区文件。这里的关键就是，最新版本的操作系统，对原有版本的操作系统能够识别，并作出有效处理。

第二种方案的缺点是，需要文件系统的良好支持。比如用户固定了某个文件的位置，而且要求该位置不能变动（除非文件被删除），这样就要求文件系统不能随便改变文件位置（或者针对操作系统文件做单独处理），否则这种方法就会失效。这显然是苛刻的，结果就是，由于文件系统不能改变已有文件的位置，会导致大量的磁盘碎片产生。常用的文件系统，比如 FAT、NTFS、EXT 等，都不提供这种功能。因此第二种方案的应用范围非常有限。只有早期的一些操作系统会采用。目前大部分操作系统都采用第一种方案进行设计。

至此，下列一些概念或原理，读者应该已经清楚。

- (1) 操作系统的引导扇区可能不止一个。
- (2) 对于 FAT32/NTFS 等复杂的文件系统，需要多个引导扇区才能完全容纳引导代码。
- (3) 可通过固定操作系统核心文件在磁盘上的位置，来确保引导代码只占一个扇区。
- (4) 不同厂商操作系统很难共存于同一个分区的原因。

3.2.4 预置引导法概述

那么，是否就意味着一定不能在同一个分区上安装不同生产厂家提供的操作系统呢？笔者认为答案是否定的。我们可以通过一些设计，来有效协调不同厂商的操作系统，在同一个分区上和谐共存。在 Hello China 操作系统的设计中，就采用了一种称为“预置引导法”的策略，有效规避了上述两个问题。

预置引导法的整体思路是，在操作系统安装的时候，根据实际硬盘的文件系统情况，预先读取操作系统核心模块在磁盘上的物理位置，并直接写入引导扇区。这样引导扇区在引导操作系统的时候，就无需再自行分析文件系统、确定操作系统核心文件的位置了，而只要从引导扇区预先设定的位置中，把文件在磁盘上的物理位置找出来，加载进内存即可。可见，预置引导法有一个前提：一旦操作系统核心文件被写入硬盘，其位置也不能变动。因为核心文件在磁盘上的位置，会被写入引导扇区。如果文件的位置改变，则仍然无法引导。这显然也是苛刻的，与固定操作系统在磁盘上的位置的策略有同样限制（但这两者有根本的不同：预置引导法要求操作系统核心文件在磁盘上的位置不变即可，其位置不固定，具体位置是文件被复制到磁盘上时确定的。而固定操作系统核心模块在磁盘特定位置的做法，则是要求系统文件一定要位于磁盘的某个固定位置，比如 1024 扇区开始）。



我们可以通过一些策略，来改进预置引导法，使得该方法能够避免上述局限。总体改进策略就是，在操作系统安装的时候动态生成（或动态配置）引导扇区内容，把能够固定的数据，事先写入引导扇区，避免引导扇区自行计算这些参数，从而降低引导扇区的代码量。比如，大部分引导扇区都需要计算一个 `cluster` 的大小，具体计算方法是根据每个扇区的字节数，乘以每个 `cluster` 的扇区数。显然，在操作系统安装的时候，`cluster` 的大小就固定了。因此可直接在引导扇区中设置一个变量（`cluster` 的尺寸），并写入 `cluster` 的尺寸值。这样引导扇区就无需计算 `cluster` 的大小，而直接引用即可。这种方法可大大减少引导扇区代码量。

下面以 Hello China 操作系统的引导程序为例，针对不同的文件系统，来说明预置引导法的设计思想。

3.2.5 预置引导法在 FAT32 文件系统上的实现

显然，FAT32 文件系统是一个相对复杂的文件系统，一个引导扇区的空间，很难装载完整的引导代码，因为即使操作系统核心文件放在根目录下，也需要搜索整个根目录，找到操作系统文件，并加载该文件。这个过程需要两个扇区左右的代码量。但是通过预置一些变量到引导扇区，可以大大减少 FAT32 文件系统的引导扇区尺寸，使得代码能够被容纳在一个引导扇区中。比如，下面是标准的 FAT32 文件系统引导扇区的布局：

字段名称	字段偏移	字段长度	含义
BS_jumpBoot	0	3	跳转代码
BS_OEMName	3	8	OEM 名字
BPB_BytsPerSec	11	2	每扇区长度（字节数）
BPB_SecPerClus	13	1	每 cluster 的扇区数
BPB_RsvdSecCnt	14	2	保留扇区数
BPB_NumFATs	16	1	FAT16 的数量
BPB_RootEntCnt	17	2	有多少个根目录
BPB_TotSec16	19	2	FAT32 已废弃
BPB_Media	21	1	媒体类型
BPB_FATSz16	22	2	FAT16 文件分配表长度
BPB_SecPerTrk	24	2	每磁道扇区数
BPB_NumHeads	26	2	磁头数
BPB_HiddSec	28	4	隐藏扇区数
BPB_TotSec32	32	4	分区总扇区数

.....

此后就是可执行的引导代码。

显然，其中很多信息是无用的，尤其是在 FAT32 文件系统上。比如 `BPB_TotSec16` 等变量。在配置启动扇区的时候，可省略这些变量，从而腾出更多的空间来安排引导代码，使 FAT32 文件系统的引导代码能够放到一个扇区内。下面是经过“预置”处理后的引导扇区结构：

```
; 3 字节的跳转指令
JMP SHORT _BOOT_CODE ; 跳转到真正的引导代码
NOP      ; 空指令以保证字节数为 3
SectorsPerCluster DB 00 ; 每个簇的扇区数 ( 可以为 1 2 4 8 16 32 64 128 )
ReservedSectors  DW 00 ; 从卷的第一个扇区开始的保留扇区数目;
NumberOfFATs     DB 00; 卷上 FAT 数据结构的数目, 该值通常为 2
HiddenSectors    DD 00; 包含该 FAT 卷的分区之前的隐藏扇区数
SectorsPerFAT32  DD 00; 对于 FAT32, 该字段包含一个 FAT 的大小
RootDirectoryStart DD 00; 根目录的起始簇号, 通常为 2;
DriveNumber      DB 00; 用于 INT 0x13 的驱动器号, 0x00 为软盘, 0x80 为硬盘
```

此后跟着真正的可执行代码。

这样预置后, 不但节约了大约 20B 的空间, 而且很多变量已经被预置 (这些被预置的变量, 称为预置变量), 无需计算 (比如 cluster 大小)。而如果按照传统的引导扇区, 则需要安排专门代码计算这些变量值。这样综合下来, 可节约大约 60B 的空间。不要小看这 60B, 在汇编语言实现的引导扇区中, 60B 可以实现分析 FAT32 根目录项的功能, 而这是 FAT32 引导扇区的核心功能。

同时, 可以优化启动扇区的代码, 省略一些不必要的判断。比如 int13 号调用功能, 目前多数 BIOS 都提供这项功能, 就无需判断 BIOS 是否支持 int13 调用了。同时, 由于操作系统是运行在 32 位 CPU 上, 可通过使用 CPU 的 32 位寄存器 (比如 EAX、EBX 等) 和 32 位指令扩大数据处理范围, 以节约代码量。

具体的实现过程是, 在 Hello China 的安装过程中, 提供一个磁盘分析工具, 用于读取或计算 cluster 的尺寸等预置变量, 然后写入引导扇区。

采用上述措施后, Hello China 实现了在一个引导扇区内, 引导 FAT32 文件系统上的操作系统的功能。而通常情况下, 这些功能是需要两个以上的扇区内实现的。

上述实现方法, 是不需要操作系统核心文件在硬盘上固定位置的。虽然对某些变量做了预置, 但是引导过程并没有改变, 引导扇区还是首先读入根目录, 在根目录中搜索操作系统核心文件。找到后再查询 FAT 表, 找到操作系统核心文件在磁盘上的具体位置, 然后依次读入。因此, 即使操作系统核心文件的位置不断变化, 也不会影响操作系统的加载。显然, 这种“预置引导法”既克服了操作系统引导扇区过大的问题, 也规避了固定操作系统核心文件位置的不利之处。

3.2.6 预置引导法在 NTFS 文件系统上的实现

相比 FAT32, NTFS 是一个更加复杂的文件系统。用预置引导法可以勉强将 FAT32 的引导代码塞到一个扇区里, 但是对 NTFS 来说, 这是绝对不可能的。在 Windows 系列操作系统中, NTFS 分区的引导扇区一共用了 16 个 (也就是说, 要引导 NTFS 上的操作系统文件, 需要 8KB 的代码空间)。因此只能更彻底地使用预置引导法。

预置引导法最彻底的用法是, 把操作系统核心文件在磁盘上的位置及分布情况, 全部写入引导扇区。引导扇区不做任何文件系统相关的分析代码, 直接根据写入的数据读取磁盘分区即可。比如, 操作系统核心文件存储在磁盘上的布局如下:

起始扇区号	扇区数量



2048	512
4096	512
8000	256

即操作系统核心文件在磁盘上分成了三部分存储，第一和第二部分分别占用连续的 512 个扇区，第三部分占用连续的 256 个扇区。可以算出，操作系统核心文件的大小是 640KB。这时候可以把上述布局中的数据直接写入引导扇区。比如，引导扇区开始部分的代码如下：

```
JMP SHORT _BOOT_CODE ; 跳转到真正的引导代码
NOP      ; 空指令以保证字节数为 3
startSectNum1 DD 00
sectorCount1 DD 00
startSectNum2 DD 00
sectorCount2 DD 00
startSectNum3 DD 00
sectorCount3 DD 00
```

此后紧跟真正的引导代码。

一旦操作系统在 NTFS 磁盘分区上安装完成，操作系统核心文件的位置就固定了（即上述布局中的描述）。这时候可以通过一个工具软件，读取操作系统核心文件的位置信息，然后写入引导扇区中连续的三个 startSectNum 和 sectorCount 的位置处。这样真正的引导扇区代码，无需做任何 NTFS 文件系统分析工作，只需要根据 startSectNum 和 sectorCount 处的信息，把相应扇区读入内存即可。显然，这样的预置变量和磁盘读取代码加在一起，绝不可能超过 512B。

但这样处理的问题也很明显，就是要求操作系统核心文件在磁盘上的位置一定要固定。一旦文件系统挪动了操作系统核心文件的位置，就无法引导。显然，对 NTFS 文件系统来说，这个要求几乎是不可能满足的。因为 NTFS 为了充分避免碎片，可能会定期对文件系统进行扫描和碎片整理，通过移动文件在磁盘上的具体位置，把不连续的磁盘空间链接起来。

为避免由于文件位置的变动而导致的无法引导问题，我们需要更进一步，把位置更加稳定的内容写入引导扇区，而不是文件在磁盘上的具体位置和大小。为了找出位置更加稳定的内容，我们首先从分析 NTFS 文件系统原理开始。需要说明的是，NTFS 文件系统内容非常庞大，在这里只对几个关键概念进行描述。详细的 NTFS 文件系统信息，请参考 NTFS 文件规范等相关资料。

NTFS 文件系统中，分区上任何文件的具体位置，都是记录在 MFT（主文件表）中。而 MFT 本身的起始位置，则存储在引导扇区中。操作系统在处理 NTFS 文件系统的时候，首先从 NTFS 分区的引导扇区中获取 MFT 的位置，然后读取 MFT 的相关内容。对任何文件的查找，NTFS 也是根据文件名等关键字段搜索 MFT（实际上是从根目录开始逐级搜索），找到对应的文件记录。文件记录是 NTFS 的一个核心数据结构，每个文件都有至少一个文件记录，里面记录了文件的大小、位置等属性信息。找到文件记录之后，即可获取文件在磁盘上的存储位置和大小。一旦一个文件被创建或被复制到磁盘上，NTFS 文件系统代码就会在 MFT 中分配一个记录，记录该文件的相关信息。对我们来说，文件的位置和大小等信息是最关键的。一般情况下，文件记录在 MFT 中的位置是固定的。同时，MFT 本身的位置也是固定的（虽然从理论上讲，MFT 本身位置也可以不固定，但在一般 NTFS 文件系统的实现中，MFT 的位置都是

固定的)。即使文件本身被移动, NTFS 文件系统代码也只是修改文件的 MFT 记录, 更确切地说, 是文件记录中的数据运行属性 (data run)。所谓数据运行, 是分区上的一片位置连续的扇区, 由扇区的起始位置和扇区数量表示。文件内容就存储在数据运行上, 一个文件可能存储在多个数据运行上, 但这些数据运行信息都记录在 MFT 中的文件记录内。

因此, 从上面的分析来看, 只要我们把操作系统核心文件所对应的 MFT 记录所在的扇区预先置入引导扇区, 就可保证不出问题。假设操作系统核心文件被移动位置, 这时候 NTFS 文件系统代码会修改对应的 MFT 记录的内容, 而 MFT 记录本身所在位置不会变化。

基于这个原理, 在 NTFS 文件系统上的预置引导法的实现, 预先置入引导扇区的是操作系统核心文件所对应的 MFT 记录的位置。这样引导扇区只要根据这个预置位置, 读取操作系统文件的 MFT 记录, 然后从记录中读出文件的数据运行 (即位置信息), 再分析数据运行、加载操作系统文件即可。

Hello China 操作系统在 NTFS 文件系统上的引导程序, 就是这样实现的。在 Hello China 的安装过程中, 安装程序首先把操作系统核心文件 (HCNIMGE.BIN) 复制到 NTFS 分区的根目录下, 然后运行一个 NTFS 文件系统分析工具, 读出 HCNIMGE.BIN 文件所对应的 MFT 记录在磁盘上的扇区号, 再把这个扇区号写入引导扇区的预置变量中。这样引导扇区在引导操作系统的时候, 只需要根据预置的磁盘扇区编号, 读入 HCNIMGE.BIN 对应的文件记录, 然后根据文件记录找到数据运行, 分析数据运行, 并读取即可。

即使如此, 在 512B 的引导扇区中实现 NTFS 文件系统的引导程序, 也有些困难。主要是数据运行的解码程序有点复杂, 占用了大部分的代码空间。

3.2.7 通过软盘启动 Hello China

Hello China 的 V1.75 版本, 可以支持从 FAT32 和 NTFS 格式的硬盘分区上启动, 也支持从软盘启动。而且在虚拟机上使用时, 使用虚拟软盘启动最为简便。本节将以软盘启动过程为例, 详细讲解 Hello China 的加载过程。之所以以软盘引导方式为例讲解, 是因为软盘引导方式相对简单直观, 而且无需涉及 FAT32 和 NTFS 等文件系统的内容。当然我们也可以以硬盘引导方式为例来解释其加载过程, 但这需要至少几十页的篇幅。考虑到系统加载不是本书核心内容 (本书核心内容为操作系统内核机制的实现), 因此我们从简处理。如果读者对硬盘启动的方式感兴趣, 则可以查阅[kernel/arch/sysinit]目录下的 hdbas.asm 和 ntfsbs.asm 两个文件, 它们分别是 FAT32 分区和 NTFS 分区的引导扇区源代码文件。当然, 如果读者对 FAT32 文件系统和 NTFS 文件系统不熟悉, 阅读这两个文件会有一些困难, 可先阅读本书第 12 章, 或者通过其他途径对 FAT32 和 NTFS 熟悉后, 再阅读这两个文件。

目前版本的 Hello China 操作系统核心由四个二进制模块组成, 见表 3-1。

表 3-1 Hello China 各组成模块

名 称	文 件 尺 寸	用 途
BOOTSECT.BIN	512B	引导扇区, 用汇编语言编写
REALINIT.BIN	4KB	实模式下的初始化代码, 用汇编语言编写
MINIKER.BIN	48KB	保护模式下的初始化代码和基本的输入/输出驱动程序
MASTER.BIN	128~560KB	操作系统核心模块, 用 C 语言编写



上述模块被一个程序 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）。如果是虚拟机，则由 VFMaker 程序根据上述四个文件，创建一个虚拟软盘文件（/bin/virtualpc/vfloppy.vfd），通过这个虚拟软盘文件引导虚拟机。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。

一张大小为 1.44MB 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 512B。Hello China 的每个模块在软盘上的位置（被 FMTLDRF.COM 写入）见表 3-2。

表 3-2 各组成模块在引导盘上的布局

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	512B（1 扇区）
REALINIT.BIN	0 面 0 道 3 扇区	0 面 0 道 10 扇区	4KB（8 扇区）
MINIKER.BIN	0 面 0 道 11 扇区	0 面 7 道 4 扇区	64KB（128 扇区）
MASTER.BIN	0 面 7 道 5 扇区	0 面 79 道 18 扇区	560KB（剩余扇区）

之所以把每个模块在软盘上的位置固定，完全是为了引导的方便，这也是预置引导法的一个具体应用。在 Hello China 的实现中，软盘完全被操作系统模块独占，没有文件系统的概念。

BIOS 会把 BOOTSECT.BIN（软盘的第一个扇区）文件读入内存，然后执行该文件。BOOTSECT.BIN 再根据上述布局，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。这三个模块在内存中连续分布，其起始地址为 0x1000（即 4KB 偏移处）。下面是 BOOTSECT.BIN 模块中的相关代码（用汇编语言编写，NASM 编译）：

```
[kernel/arch/sysinit/bootsect.asm]
gl_start:
    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，其功能是把自身（BOOTSECT.BIN 模块）从内存的 0x07C0 偏移处搬移到 0x9F000 处（即 636KB 处），以腾出空间加载其余三个核心模块。搬迁完成后，跳转到 0x9F000 处继续执行。其中，DEF_ORG_START 是预定义的一个宏，定义为 0x07C0，即引导扇区被 BIOS 加载到内存后的地址，而 DEF_BOOT_START 则被定义为 0x9F00，是 BOOTSECT.BIN 被重新搬移到的位置。要理解上述汇编代码，需要知道实模式下 CPU 的寻址方式。在实模式下，CPU 是通过段地址加上段内偏移地址，形成最终的物理地址。在与段内偏移地址相加的时候，段地址需要向左移动 4 比特。反之亦然，在设置段地址寄存器的时候，需要把物理地址向右移动 4 比特。

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，完成操作系统剩余模块（即除 BOOTSECT.BIN 之外的三个模块）的加载。代码如下：

```
gl_bootbgn:
    call np_printmsg
    call np_load
    jmp DEF_RINIT_START / 16 : 0 ;;Jump to the real mode initialization code.
```

加载完毕，使用一个远跳转指令，跳转到 REALINIT.BIN 模块处开始执行。下面是加载函数 np_load 的相关代码：

```
np_load:
    push es
    mov ax,0x0000
    mov es,ax
    mov bx,DEF_RINIT_START
    xor cx,cx
.ll_start:
    mov ah,0x02
    mov al,0x02
    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
    je .ll_inc_es
    add bx,1024
```



```
        jmp .ll_continue1
.ll_inc_es
        mov bx,es
        add bx,4*1024
        mov es,bx
        xor bx,bx
.ll_continue1:
        inc byte [curr_sector]
        inc byte [curr_sector]
        cmp byte [curr_sector],DEF_SECT_PER_TRACK
        jae .ll_inc_track
        jmp .ll_start
.ll_inc_track:
        mov bp,es
        mov word [tmp_word],bp
        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 shuts 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。代码之所以较长，是因为这三个模块分布在软盘的一个整面上，跨越了多个磁道和多个扇区，加载过程中必须判断是否跨越磁道和盘面，如果是则需要递增磁道计数器。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。我总感觉这个加载过程有点啰嗦，但考虑到这部分代码的利用率比较低，只是操作系统加载时会用一次，加载完成之后便不用了，因此没有做进一步优化。

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 48KB，而写到磁盘上时，却占用了 64KB 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。一旦跳转到 `0x1000` 处，加载过程就完成了。剩下的过程是初始化过程，在本章的最后一部分中进行叙述。需要说明的是，不论是从软驱启动还是从硬盘启动，不同的是加载过程。一旦加载完成，都是跳转到 `0x1000` 处开始初始化的，即初始化过程是与加载过程完全无关的。

3.3 嵌入式操作系统的引导和初始化

3.3.1 典型嵌入式系统内存映射布局

介绍完个人计算机的操作系统引导过程后，再看一下嵌入式操作系统的引导和初始化过程。`Hello China` 定位为嵌入式智能操作系统，相比通用个人计算机领域，嵌入式领域的应用更加关键。

首先从分析嵌入式系统的硬件开始。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 1MB），用于存放嵌入式系统加电后的初始化代码。在 PC 上，用于完成加电后检测（POST 功能）的 BIOS，功能与此类似。

（2）Flash，是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。这类介质的容量一般比 Boot ROM 要大，比如，可以在 1MB 到 64MB 之间变化。

（3）SRAM/DRAM，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，如图 3-1 所示。

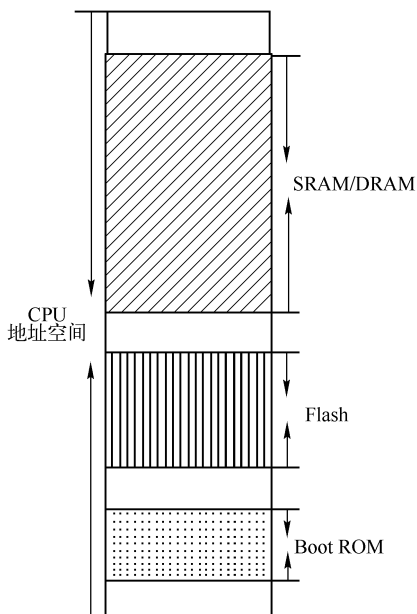


图 3-1 各类存储设备的内存布局

这样，只需要采用 CPU 的内存读写机制，就可以很方便地完成对这些设备的读写操作，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，这些存储介质往往是作为存储设备配置数据的介质而存在的，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。

3.3.2 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，并开始执行。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在地址空间的开始处（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 0xFFFF FFF0 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，存放嵌入式系统的初始化代码，启动向量所在的跳转指令，其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，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（如 COM 接口）的初始化等。其余设备的初始化代码与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（在嵌入式系统中，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。

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 的地址空间中的，因此，使用 CPU 的访问内存指令，就可以直接完成对 Flash 的操作，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，来支撑对这种形态的 Flash 的访问。由于这时候操作系统还没有加载，Flash 的驱动程序只能存放在 Boot ROM 中。

3.3.3 常见嵌入式操作系统的加载方式

本节对嵌入式操作系统的加载方式进行简单的描述。之所以对嵌入式操作系统的加载方式进行描述，是为了让读者更好地了解常见的嵌入式系统的启动过程，以便根据实际需要，把 Hello China 移植到特定的目标系统上。下面介绍三种方式。

1. 从 Flash 直接加载

这种加载方式下，嵌入式操作系统映像和应用程序映像，都存放在 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，包括 TEXT 节、DATA 节、BSS 节等。不同的节存放的内容不同，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图 3-2 所示。

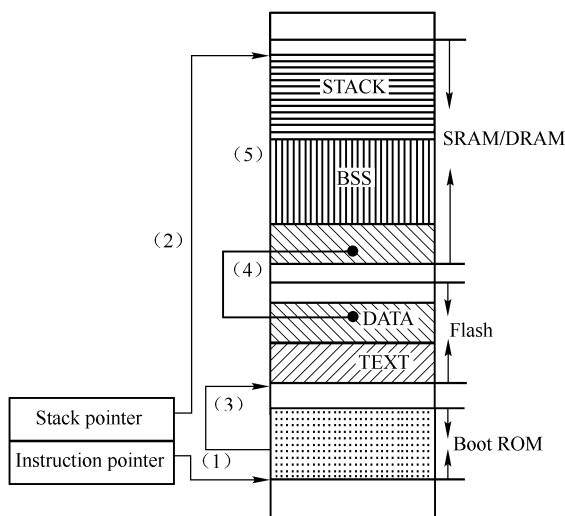


图 3-2 从 Flash 直接加载嵌入式系统

详细步骤为（每个步骤对应图中的一个数字标号）：

（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，执行必需的硬件初始化工作。

（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，以及其他硬件的初始化。

（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）把 Flash 中 DATA 节代码复制到 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，留给未初始化变量使用。

上述功能完成之后，嵌入式系统的执行环境已经准备完毕，进入操作系统初始化阶段。

需要注意的是，DATA 节的搬迁和 BSS 节的预留工作也可能由 Boot ROM 完成，即 Boot ROM 中的硬件初始化代码执行完后，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，Boot ROM 中的代码需要知道 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，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。若 CPU 本身携带了较大数量的代码 Cache，则这个问题会得到一定程度的缓解。

（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、DATA 节和 BSS 节的起始地址，这个地址应该与它在 Flash 和 RAM 中的最终位置相同。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，被大量地采用。Internet 发展初期的一些低端路由器经常采用这种方式。

2. 从 RAM 中加载

与从 Flash 直接加载方式不同的是，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，加载过程如图 3-3 所示。

详细步骤为（每个步骤对应图中的一个数字标号）：

（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，执行必需的硬件初始化

工作。

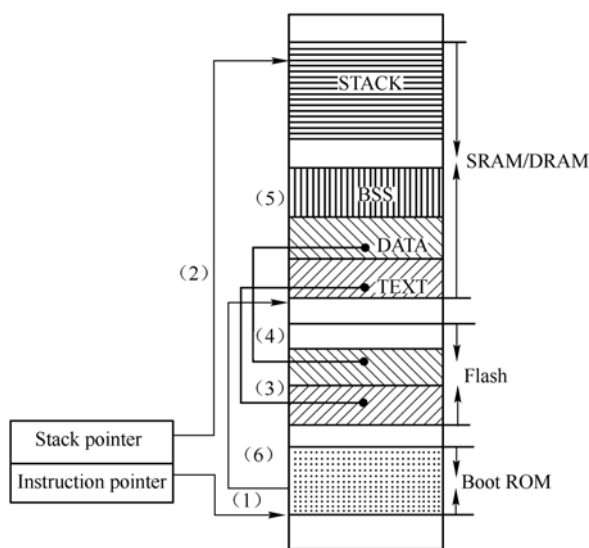


图 3-3 RAM 加载过程

(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，以及其他硬件的初始化。

(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像的开始处（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像的 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直接在 RAM 中执行。

3. 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像存放在 Flash 当中，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，这在操作系统映像很大的情况下，矛盾尤其突出。而下面介绍的方式是从外部存储器加载操作系统和应用程序映像的，可以解决该问题。

这种方式与从 Flash 加载方式的区别是，操作系统映像和应用程序映像存放在外部存储器（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图 3-4 所示。

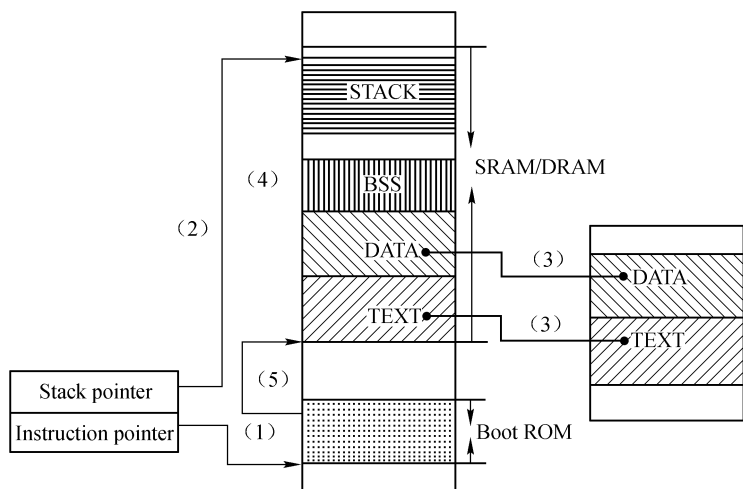


图 3-4 从文件系统加载运行过程

详细步骤为（每个步骤对应图中的一个数字标号）：

(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，执行必需的硬件初始化工作。

(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，以及其他硬件的初始化。

(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。这个过程中，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），跳转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（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。

这种设计方式的另一个优点就是可以在线升级（系统运行过程中，对操作系统和应用程序软件进行升级）。实际上，从 RAM 中运行代码的方式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，默认情况下加载哪个模块（或版本）。

实际上，PC 上的操作系统就是这样一种系统。PC 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，对于嵌入式开发入门者来说，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（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。

除了上述几种加载方式，在嵌入式开发领域中，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。

3.3.4 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地写入到 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，默认情况下所有存储设备，包括 Boot ROM、Flash、外部存储器等都是空的。因此，要想使硬件运行，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，把这些系统软件加载到 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。

1. 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，常用的写入方式有两种。

（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 芯片插到印制电路板上。一般情况下，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。

(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, 通过转换头与 PC 的串口或并口进行连接, 然后通过 PC 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。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。

2. 操作系统和应用程序映像的写入

在完成 Boot ROM 软件的写入后, 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, 也可以采用与 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, 操作系统可方便地写入 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, 则会进入系统软件更新界面。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 的终端模拟软件 (比如 Windows 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...
```

如果用户在 10s 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, 进入默认的启动过程):

```
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
```

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。在选择使用以太网接口更新软件之前, 需要首先配置嵌入式系统的以太网接口, 包括 IP 地址、网络掩码、默认网关等参数, 配置完成之后, 才能通过 TFTP, 把软件下载到目标系统上。

上面列出的用户交互内容, 都是由位于 Boot ROM 内的软件完成的, 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, 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, 还可能在 Boot ROM 中集成软件调试功能。

对于 Boot ROM 代码的编写，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，有时甚至比操作系统和应用软件本身更加复杂。

3.4 Hello China 的初始化

3.4.1 实地址模式下的初始化

Hello China 引导完成，即引导扇区把所有操作系统核心模块成功加载到内存之后，内存布局如图 3-5 所示。

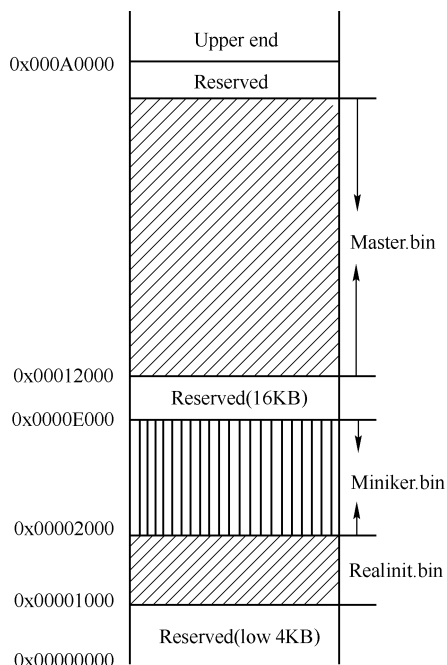


图 3-5 引导完成后的内存布局

该图是按照内存地址从小到大的顺序，从下往上画的。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（0x1000）开始执行。下面是 Realinit.bin 开始部分的代码，采用 NASM 编译，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：

```
[kernel/arch/sysinit/realinit.asm]
bits 16                                ;;The real mode code.
org 0x0000
%define DEF_RINIT_START 0x01000
```

```
%define DEF_MINI_START 0x02000
```

上述代码，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，第一个宏 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，即在 realinit.bin 执行完后，进一步跳转的目标地址。

```
gl_initstart:
    mov ax,DEF_RINIT_START
    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 就不用考虑自己在内存中的位置，直接从 0 偏移开始执行。对于堆栈寄存器的值，设置为 0x0FFF，即相对于段寄存器，偏移约 4KB，如图 3-6 所示。

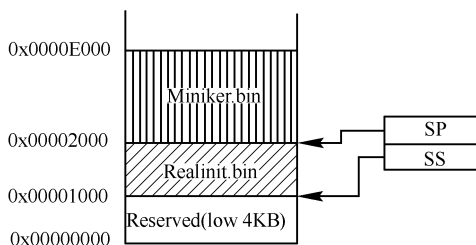


图 3-6 SP 和 SS 寄存器的初始化

需要记住的是，在代码执行过程中，堆栈指针是向下递减的。按照目前的实现，为 realinit.bin 预留了 4KB 的空间，但实际上，该模块的大小尚不超过 2KB。因此，把堆栈指针 sp 设置为 0x0FFF，意味着有 2KB 左右的堆栈空间使用，这是足够的。

```
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，提示操作进度及结果。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 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）、时钟等，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 移植到其他非 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。在 Hello China V1.75 版针对 PC 的实现中，这些代码大部分都比较简单，尽量保留了 BIOS 的初始设置。

```
call np_act20addr         ;;激活 A20 地址线
```

上面这个过程调用用来激活 A20 地址线。这在 PC 上十分关键，因为只有激活了 A20 地址线，才能确保 CPU 在保护模式下，可以访问到所有的 32 位物理地址。其中的原因在很多资料上都有描述，在此不再详述。

```
xor eax,eax
mov ax,ds
shl eax,0x04
add eax,gl_gdt_content    ;;这时 EAX 内存放了 GDT 的物理地址
mov dword [gl_gdt_base],eax
lgdt [gl_gdt_ldr]        ;;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。在初始化全局描述表寄存器的时候，实际上是用全局描述表的物理地址填入了该寄存器。因此首先要计算出 GDT 的物理地址，计算方式就是由当前数据段地址左移 4 位，再加上 GDT 在段内的偏移。

gl_gdt_content 是 GDT 的定义，这个定义如下：

```
gl_gdt_content:
    dd 0x00000000        ;;空描述符
    dd 0x00000000
    dd 0x0000ffff        ;;代码段描述符
    dd 0x00cf9b00
    dd 0x0000ffff        ;;数据段描述符
    dd 0x00cf9300
```

按照 Intel 的定义，每个段描述表项占用 8B，其结构如图 3-7 所示。

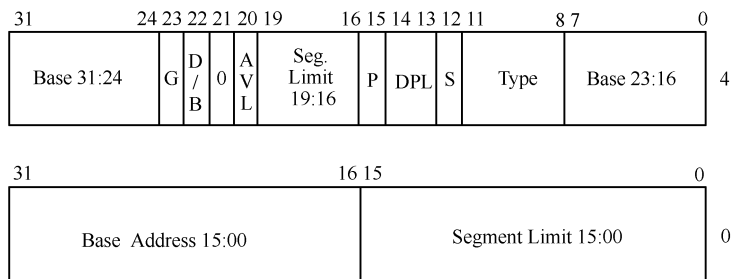


图 3-7 段描述符的结构

其中下面的方框是低字节部分，上面的方框是高字节部分。里面的几个关键字段含义如下：

(1) Segment Limit: 段界限，即这个段的最大尺寸，以 4KB 为单位。在 Hello China 的实现中，我们一般设置为 4GB，即整个 CPU 的寻址空间。

(2) Base Address: 段基址，这个段在内存中的起始地址。

(3) Type: 段类型，如代码段、数据段等。

(4) DPL: 描述符权限级别，在 Hello China 的实现中，统一设置为 0。

更详细的信息，请参考本书参考文献[1]。同时按照 Intel 的定义，全局描述表中的第一个表项必须是空表项（全为 0），然后才是实际的描述表项。Hello China 的实现中，所有的段的基址都是 0，界限都是 4GB，优先级都是 0（即系统级）。不同的只是段类型，有的是代码段，有的是数据段。按照这样的填写方式，形成了上面 `gl_gdt_content` 处的 GDT 定义。填写 GDT，并完成 GDT 寄存器的加载，是为转移到保护模式做准备。GDT 寄存器成功加载之后，就可以转移到保护模式了：

```
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，即上述 `gl_gdt_content` 标号处的定义）中的偏移，而 `DEF_MINI_START` 则指明了 `miniker.bin` 模块在代码段内的偏移地址。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 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。

接下来 CPU 就正式进入了保护模式，之后的初始化工作，就完全运行在保护模式下了。

3.4.2 保护模式下的初始化

Minker.bin 模块是完全运行在保护模式之下的模块。严格来说，这个模块是一个历史遗留模块。当初在设计 Hello China 的时候，是希望把键盘驱动程序、字符显示驱动程序等内容放到这个模块里实现，同时在这个模块内实现一个最简单的 shell，完成用户交互。通过 Miniker 即可独立操作计算机，无需进一步加载 master 等模块，这也是 miniker (Mini-Kernel) 名称的由来。但随着 Hello China 结构的转变，miniker 中的大部分功能都被独立出来单独实现。比如键盘驱动程序，用 C 语言做了更加完善的实现。这样 Miniker 的功能就退化为初始化中断描述符表 (IDT) 和全局描述符表 (GDT) 的功能。这两个表项的初始化代码不多，后续会移植到 realinit 中实现 (实际上 realinit 已经初始化了 GDT，但是比较简单，miniker 又做了更加完整的初始化)，这样 miniker 会慢慢消失。但在 V1.75 版本里，这个模块还是存在的。

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。

```
[kernel/arch/sysinit/miniker.asm]
bits 32
org 0x00100000
mov ax,0x010
mov ds,ax
mov es,ax
jmp gl_sysredirect
```

上述代码明确地指示编译器编译成 32 位指令代码，即保护模式下的指令，且偏移地址设定为 1MB 开始处。MINIKER.BIN 模块也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。需要注意的是，对于 DS/ES 等寄存器的初始化，使用的是全局描述符表中的第三个表项 (偏移 0x10 处)。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。

```
gl_sysredirect:
    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
```

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 1MB 内存范围内）重新搬移到 1MB 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，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，内存的布局如图 3-8 所示。

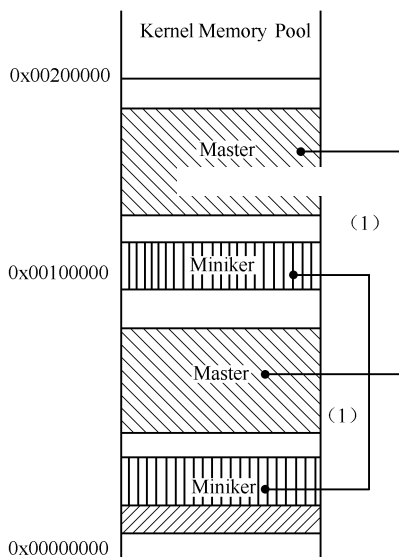


图 3-8 模块搬移完成后的内存布局

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 1MB 开始处，因此，后续的执行也必须相应地调整到新的 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，即以当前位置为基础，在 EIP 寄存器上加上当前位置到标号的偏移量，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:
    lgdt [gl_gdtr_ldr]
```

```
mov ax,0x010
mov ds,ax
mov ax,0x018
mov ss,ax
mov esp,DEF_INIT_ESP

mov ax,0x020
mov es,ax
mov fs,ax
mov ax,0x020
mov gs,ax
jmp dword 0x08 : gl_sysinit
```

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函数调用（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。详细信息可参考第 5 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 ESP 寄存器，这个值目前定义为 0x13FFFFFF，即物理内存 20MB 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:
    mov eax,gl_trap_int_handler
    push eax
    call np_fill_idt           ;;Initialize the IDT table.
    pop eax
    lidt [gl_idtr_idtr]       ;;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 8259 中断控制寄存器。这样做是因为在实模式下，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。对于中断描述符表（IDT）的初始化，在第 8 章中有详细介绍，在此不作赘述。



完成上述功能后，该过程使能中断（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，这部分也是与特定硬件平台相关的。后续所有功能，都是采用 C 语言实现的与机器无关部分功能。

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 BSP，因为在这两个模块中，完成了对特定硬件平台（PC）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。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，操作系统启动一个 shell 线程，用来完成与用户的交互，到达这一步后，操作系统才算成功启动完毕。

3.4.3 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（process 工具，采用 C 语言编写的 PE 文件处理工具），把 PE 文件的开头部分替换为可执行的指令，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，跳转到此处执行。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 00B8000000 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。

下面便是 MASTER.BIN 入口函数的实现代码。为了便于阅读，我们分段解释。

```
[kernel/osentry/os_entry.cpp]
```

```
void __init()
{
    __KERNEL_THREAD_OBJECT*    lpIdleThread    = NULL;
    __KERNEL_THREAD_OBJECT*    lpShellThread   = NULL;
    __KERNEL_THREAD_OBJECT*    lpKeeperThread  = NULL;
    DWORD                      dwKThreadId    = 0;
    DisableInterrupt();
```

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();
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);
```

SetKeyHandler 函数用于设置键盘中断处理程序，在 Hello China 最初的实现中，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，设计了一个回调机制，即在 MASTER.BIN 中实现一个处理函数（该函数就是 _KeyHandler），将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，MASTER.BIN 就可以接收到这个按键事件，从而做进一步处理。但是在 Hello China V1.75 版的实现中，键盘驱动程序功能被剥离了出来，由单独的驱动程序实现，这样上述处理过程就无意义了。但为了兼容，还是保留了上述代码。

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

上述代码完成页索引对象的初始化工作，详细信息可参考第 5 章。

```
#ifdef __ENABLE_VIRTUAL_MEMORY
    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，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。Hello China 目前尚未实现进程机制，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，则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(
    (__COMMON_OBJECT*)&KernelThreadManager,
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_LOWEST,
    SystemIdle,
    (LPVOID>(&dwIdleCounter),
    NULL);
if(NULL == lpIdleThread)
{
    __ERROR_HANDLER(ERROR_LEVEL_FATAL, 0L, NULL);
    goto __TERMINAL;
}
lpShellThread = KernelThreadManager.CreateKernelThread(
    (__COMMON_OBJECT*)&KernelThreadManager,
    0L,
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_NORMAL,
```

```
SystemShell,
NULL,
NULL);
if(NULL == lpShellThread)
{
    __ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,NULL);
    goto __TERMINAL;
}
g_lpShellThread = lpShellThread;
if(!DeviceInputManager.Initialize((__COMMON_OBJECT*)&DeviceInputManager,
                                NULL,
                                (__COMMON_OBJECT*)lpShellThread))
{
    __ERROR_HANDLER(ERROR_LEVEL_FATAL,0L,NULL);
    goto __TERMINAL;
}
```

上述代码完成了两项初始化功能：

(1) 创建了空闲线程 (Idle Thread) 和用户交互线程 (Shell Thread)。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 到其他硬件环境，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会导致系统停止启动，进入死循环。表 3-3 列举了上述初始化的全局对象，以及这些对象的功能。

表 3-3 全局对象

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

这些对象的详细功能及其实现方式等，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，采用其他线程来替代 shell 线程，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，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
__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，分页机制的使能工作。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 20MB 依然映射到物理内存的前 20MB，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，是通过定义宏 __ENABLE_VIRTUAL_MEMORY 来进行控制的。

```
SetTimerHandler(GeneralIntHandler);
```

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 GeneralIntHandler 作为入口的，然后 GeneralIntHandler 再调用 System 对象的相应函数，完成中断的进一步分发（详细信息请参考第 8 章）。在 Hello China 的当前实现中，GeneralIntHandler 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 GeneralIntHandler 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处理程序的设置，其实不然，该函数完成了通用中断处理函数 GeneralInitHandler 和所有 IDT 之间的连接。之所以用 SetTimerHandler 作为函数名，是由于历史原因造成的。关于中断的详细信息，请参考第 8 章。

```
StrCpy("[system-view],&HostName[0]);
EnableInterrupt();
DeadLoop();
__TERMINAL:
ChangeLine();
GotoHome();
__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，开始正式调用线程。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毕，这时候，如果不进入一个死循环，则__init 函数返回后，可能会使 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（Idle 或 shell），重新投入运行，这样初始化线程就算正式结束了。更详细的内容，在第 4 章中有详细介绍。

最后部分的代码是出错处理部分。在初始化过程中，遇到任何一个错误都可能导致初始化失败，__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。

3.4.4 Hello China 的内存布局图

最后，我们再看一下 Hello China V1.75 初始化完成、进入稳定运行阶段后的内存布局图，如图 3-9 所示。这个布局图包含了所有核心模块，有些模块是在其他章节中进行讲解的，目前不理解没有关系。

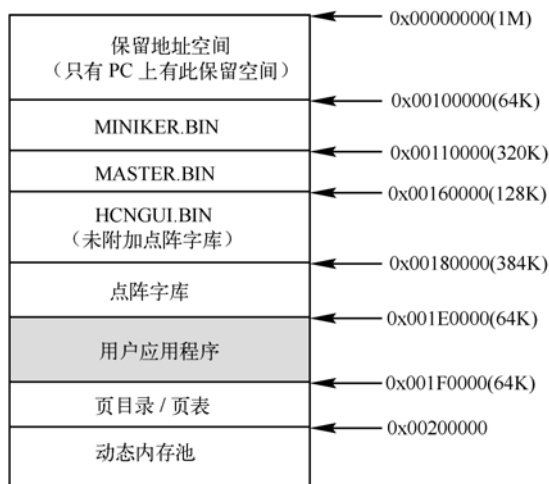


图 3-9 启动完成后的内存布局

在 PC 版的实现上，最低的 1MB 内存是预留的。看起来或许有点浪费，但实际上这样的操作机制带来了非常大的便利。比如在后续开发中，需要从保护模式重新切回实模式，并调用 BIOS 的功能。正是保留了这 1MB 的预留空间，才使得这个过程变为可能。

REALINIT.BIN 模块是实模式的初始化模块，一旦完成实模式的初始化，这个模块就没有用处了。因此在 Hello China 稳定之后的内存布局中，是没有这个模块的。从 1MB 开始，首先是 miniker.bin 模块，占用 64KB 空间，紧接着是 master.bin 模块，占用了 320KB 的空间。当前版本的 master.bin 模块没有这么大，预留 320KB 的空间是为了将来扩展该模块使用



的。master 模块之后是 GUI 模块，包含 GUI 的功能代码模块、汉字库、ASCII 字符库等。然后就是应用程序的加载空间，所有 Hello China V1.75 版本的应用程序，都被加载到这 64KB 的空间中运行。需要注意的是，这个 64KB 的空间，只是应用程序的固定代码和全局变量所占空间。应用程序可以调用操作系统提供的 API 函数，从动态内存池中申请更多的空间使用。

部分页目录和页表被固定在 0x001F0000 处，这只有在起用了 VMM（虚拟内存管理功能）之后才有效。从 2MB 开始，就是可供内核和应用程序使用的动态内存空间了。

3.5 Hello China 的字符 shell

3.5.1 字符 shell 的概述和启动

作为用户与计算机的接口，shell 的地位非常重要。但是 shell 的实现原理却并不复杂，它本身是操作系统的一个程序，用于接受用户输入，然后调用对应的程序完成处理，并反馈结果。同时 shell 也是大多数操作系统初始化之后，运行的第一个程序。本节简单介绍一下 Hello China 的字符界面 shell 的实现。图形界面的 shell，在第 11 章中做详细介绍。

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（详细内容请参考 3.4.3 节）。线程的入口点，即线程的功能函数，就是 SystemShell。该函数是一个封装函数，直接调用 EntryPoint 例程。在 EntryPoint 例程中，事先定义了线程的消息处理功能，代码如下：

```
[kernel/shell/shell.cpp]
DWORD EntryPoint()
{
    __KERNEL_THREAD_MESSAGE KernelThreadMessage;

    PrintPrompt();
    while(TRUE)
    {
        if(GetMessage(&KernelThreadMessage))
        {
            if(KTMSG_THREAD_TERMINAL == KernelThreadMessage.wCommand)
                goto __TERMINAL;
            DispatchMessage(&KernelThreadMessage,EventHandler);
        }
    }
    __TERMINAL:
    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;
    WORD        wParam;
    DWORD       dwParam;
};
```

EntryPoint 函数调用 GetMessage 函数，从 shell 线程的消息队列中获取消息并调用 DispatchMessage 函数进行处理。需要注意的是，在调用 DispatchMessage 处理之前，首先判断是不是一个系统结束消息（如果用户按下了“CTRL+ALT+DEL”组合键，则系统会向 shell 线程发送一个系统结束消息（KTMSG_THREAD_TERMINAL 消息）。如果发现是系统结束消息，则函数返回，从而导致 shell 线程结束。

3.5.2 Shell 的消息处理过程

缺省情况下，一个线程在创建的同时会创建一个消息队列，该消息队列与线程的控制结构（核心线程对象）进行绑定。其他的线程可以调用 SendMessage 函数向该线程发送消息，发送的消息类型是 __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。若该函数成功地从消息队列中获取了一个消息，则返回 TRUE，KernelThreadMessage 结构体里面存储了所获得的消息的内容；若当前线程消息队列为空，则该函数会阻塞当前线程的运行，等待消息的到来。

线程之间的消息有许多种类型，比如键盘消息、鼠标消息以及用户自己定义的消息。在 EntryPoint 的实现中，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 函数，从自己的线程队列中获取消息进行处理。

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，其实现十分简单，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 函数，该函数的实现代码如下：

```
[kernel/shell/shell.cpp]
BOOL EventHandler(WORD wCommand,WORD wParam,DWORD dwParam)
{
    WORD wr = 0x0700;
    BYTE bt = 0x00;
    BYTE Buffer[12];

    switch(wCommand)
    {
    case MSG_KEY_DOWN: //键被按下
        bt = LOBYTE(LOWORD(dwParam)); //bt 中存放了被按下键的 ASCII 码
        if(VK_RETURN == bt) //按下的键是回车键
        {
            if(BufferPtr)
                DoCommand();
        }
    }
```

```

        PrintPrompt(); //处理完命令后，再打印出提示符
        break;
    }
    if(VK_BACKSPACE == bt) //如果是一个删除键
    {
        if(0 != BufferPtr)
        {
            GotoPrev(); //光标回退一格，并删除当前字符
            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 函数只处理键盘消息。其中，KernelThreadMessage 的 wCommand 变量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 函数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 函数就可以根据 dwParam 的最低一个字节确定按下的是哪个键。根据按键的不同，分三种情况进行处理。

(1) 若按下的键是回车键（VK_RETURN），则 EventHandler 函数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串 “[system-view]”，然后返回。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 函数会调用 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 (GotoPrev 函数), 最终的表现就是用户按 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 函数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, 则不做任何处理, 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 函数实际上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 函数会调用 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。系统中维护了一个命令字符串与对应的处理函数之间的映射表, DoCommand 函数无非是用命令缓冲区中的字符串, 逐个匹配映射表中的项目。若匹配到一个命令, 则调用对应的处理函数。在调用处理函数的时候, 是以命令缓冲区 (CmdBuffer) 为参数的。这样命令处理函数可以把用户输入作为参数进行更进一步处理。比如用户输入了字符串 “ktview -i 10” 后, 按下了回车键。这时候 DoCommand 会使用字符串中的 “ktview” 来匹配映射表, 找到其处理函数 KtViewHandler 后, 把整个字符串传递给这个函数。下面是命令字符串与处理函数之间的映射表:

```
[kernel/shell/shell.cpp]
#define CMD_OBJ_NUM 21
__CMD_OBJ CmdObj[CMD_OBJ_NUM] = {
    {"version" , VerHandler},
    {"memory" , MemHandler},
    {"sysinfo" , SysInfoHandler},
    {"sysname" , SysNameHandler},
    {"help" , HlpHandler},
    {"cpuinfo" , CpuHandler},
    {"support" , SptHandler},
    {"runtime" , RunTimeHandler},
    {"test" , TestHandler},
    {"untest" , UnTestHandler},
    {"memview" , MemViewHandler},
    {"ktview" , KtViewHandler},
    {"ioctl" , IoCtrlApp},
    {"sysdiag" , SysDiagApp},
    {"loadapp" , LoadappHandler},
    {"gui" , GUIHandler},
    {"reboot" , Reboot},
    {"poff" , Poweroff},
    {"cls" , ClsHandler}
```

```
};
```

其中 `_CMD_OBJ` 是预先定义的一个结构体，把一个字符串和一个处理函数进行了关联。在 `shell` 成功启动后，用户即可看到 `[system-view]` 提示符。这时候输入 `help` 命令，即可显示出系统中所有可用的命令（及其帮助信息）。如图 3-10 所示。

```
[system-view]help
The following commands are available currently:
version      : Print out the version information.
memory       : Print out current version's memory layout.
sysinfo      : Print out the system context.
sysname      : Change the system host name.
help         : Print out this screen.
support      : Print out technical support information.
runtime      : Display the total run time since last reboot.
menview      : View a block memory's content.
ktview       : View all the kernal threads' information.
ioctl        : Start IO control application.
sysdiag      : System or hardware diag application.
fs           : File system operating application.
fdisk        : Hard disk operating application.
loadapp      : Load application module and execute it.
gui          : Load GUI module and enter GUI mode.
reboot       : Reboot the system.
cls          : Clear the whole screen.
```

图 3-10 运行截图

需要注意的是，`help` 的输出结果与上述命令行与处理函数映射表之间不是一一对应的关系，在映射表中的一些命令，并没有显示在 `help` 列表中。这些未显示的命令是内部测试命令，不对外呈现。

3.5.3 实例：增加一个字符 shell 内置命令

采用映射表的方式，使系统功能的添加和删除变得比较容易。比如要添加一个功能，只需要写一个处理函数，然后在映射表内添加一个项目即可。下面通过一个示例说明如何向系统中添加新的功能。

第一步：假设新增的功能命令为 `mycommand`，首先编写一个功能函数，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言的一个入门示例。

第二步：把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。代码如下（黑体部分是修改内容）：

```
///#define CMD_OBJ_NUM 21
```

```
#define CMD_OBJ_NUM 22
__CMD_OBJ CmdObj[CMD_OBJ_NUM] = {
    {"version" ,    VerHandler},
    {"memory" ,    MemHandler},
    {"sysinfo" ,    SysInfoHandler},
    {"sysname" ,    SysNameHandler},
    {"help" ,       HlpHandler},
    {"cpuinfo" ,    CpuHandler},
    {"support" ,    SptHandler},
    {"runtime" ,    RunTimeHandler},
    {"test" ,       TestHandler},
    {"untest" ,     UnTestHandler},
    {"memview" ,    MemViewHandler},
    {"ktview" ,     KtViewHandler},
    {"ioctl" ,      IoCtrlApp},
    {"sysdiag" ,    SysDiagApp},
    {"loadapp" ,    LoadappHandler},
    {"gui" ,        GUIHandler},
    {"reboot" ,     Reboot},
    {"poff" ,       Poweroff},
    {"cls" ,        ClsHandler},
    {"mycommand",   mycommand}
};
```

第三步：重新编译链接（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。

3.6 从保护模式切换回实模式

3.6.1 模式切换概述

在操作系统初始化的过程中，完成硬件初始化等工作之后，CPU 即切换到保护模式。从实模式切换到保护模式相对简单，首先定义 GDT 表，针对代码段、数据段等都要定义对应的表项，并把 GDT 表的起始物理地址加载到 GDTR 寄存器，接下来把 CR0 寄存器的第一个比特设置为 1，最后采用一条段间跳转指令，即可切入保护模式。在 Hello China 的最初版本中，切换到保护模式之后，就彻底告别了实模式。但是在 Hello China V1.75 的实现中，为了读写磁盘、设置显示器工作模式等工作，还需要从保护模式切换回实模式，通过 BIOS 调用完成这些工作。下面大致介绍一下如何从保护模式切换回实模式，供读者参考。

按照 Intel IA32 编程指导手册上的描述，从保护模式切换到实模式，需要遵循下列步骤：

- （1）关闭中断，即执行 CLI 指令清除 CPU Eflags 寄存器的中断允许位。
- （2）如果启用了分页功能（Hello China 缺省启用了分页功能），则需执行下列步骤。

1) 跳转到一段线性地址和物理地址相同的代码, 即经过分页机制映射后的线性地址与实际物理地址相同的代码段。由于 Hello China 系统空间的映射地址与物理地址相同, 因此该步骤可省略。分页机制及相关概念, 可参考第 5 章获取更多信息。

2) 确保 IDT 和 GDT 在物理地址和虚拟地址相同的页面中, 这很容易做到。

3) 清除 CR0 寄存器的 PG 比特 (第 32 比特), 即关闭分页机制。上述两个步骤的要求, 就是为关闭分页机制奠定基础。

4) 对 CR3 寄存器清零, 以刷新 TLB。

(3) 跳转到一个长度是 64KB 的代码段, 这可确保 CS 寄存器保持与实模式相同的段长度限制。这样就必须在 GDT 中增加一个段描述符, 这个段描述符的段长度是 64KB。

(4) 向 DS/SS/FS/ES/GS 等段寄存器装载一个满足下列条件的选择符, 以确保切换到实模式时, 这些寄存器能够与实模式要求吻合。

1) 段长度限制是 64KB。

2) 以 BYTE 为粒度, 而不是以缺省的 4KB 为长度粒度。

3) 向上扩展。

4) 可写入。

5) 基地址可以是任何值。

(5) 执行 LIDT 指令, 确保中断寄存器指向一个实模式中中断表。在 PC 启动时, 中断描述表位于内存的开始处, 长度为 1024B。由于在 Hello China 启动过程中这部分内存没有被破坏, 因此在这里可继续使用。

(6) 清除 CR0 的 PE 比特 (第 0 比特), 正式进入实模式。

(7) 执行一个远跳转指令, 跳入实模式中的一段代码继续执行。这个跳转指令刷新指令预取队列, 并加载 CS 寄存器。这时就正式进入了实模式, 寻址方式按照“段基址向左移动 4 位, 加上段偏移”的方式进行。

(8) 按照实模式要求, 装载 DS/ES/FS/GS/SS 等寄存器。如果有寄存器不使用, 可装入 0。

(9) 执行 STI 指令, 启用中断功能。

最后, 要确保上述代码位于同一个内存页面中, 而且页面的虚拟地址与物理地址相同。在 Intel 32 位的 CPU 中, 一个页面的缺省大小是 4KB, 这对上述代码来说是足够的。

在 Hello China 的实现中, 从保护模式向实模式切换的代码, 都是在 realinit.asm 文件中实现的。在编译的时候, 这段模式切换代码被放在了 realinit.bin 模块内的 2KB 偏移处。再回忆一下 Hello China 的加载过程, realinit.bin 模块被加载到物理内存 4KB 开始处的位置, 共占用 4KB 的空间。这样模式切换部分代码即位于内存的 6KB 偏移处。在进行模式切换的时候, 只需要跳转到这个位置, 即可执行模式切换代码。实模式代码执行完毕, 会重新返回保护模式。可使用通用寄存器在保护模式和实模式之间传递参数和返回值, 下面是各个寄存器的分配情况:

(1) EAX 寄存器给出了需要调用的 BIOS 服务代码, 这个代码是 Hello China 自己定义的。实模式代码根据这个号码, 决定调用哪种 BIOS 服务。调用返回的时候, EAX 返回实模式执行结果是否成功的标志, 0 表示执行失败, 否则表示成功。

(2) ECX、EDX、ESI、EDI、EBP 分别存储了调用 BIOS 服务的参数, 最多五个参数。后续如需要, 可通过共享内存方式解决参数不足问题。

需要说明的是，从保护模式切换回实模式，比从实模式切换到保护模式要复杂得多。但是建议读者不要对这个过程太过关注，因为这个过程与操作系统的核心机制关系不大，对操作系统关键原理的理解也没有太多帮助，还是建议读者把更多的精力放在理解操作系统核心的实现机制上。

3.6.2 实模式服务调用举例

在正式介绍保护模式切换回实模式的实现之前，先举例说明如何在保护模式下调用实模式的功能。下面的代码用于实现硬盘扇区的读取功能，这个函数把待读取硬盘号（0 表示第一个硬盘、1 表示第二个硬盘）、起始扇区、扇区个数等作为输入参数。成功读取之后，扇区数据被复制到 pBuffer 缓冲区中。代码如下：

```
[/kernel/kernel/arch/bios.cpp]
BOOL BIOSReadSector(int nHdNum,DWORD nStartSector,DWORD nSectorNum,BYTE* pBuffer)
{
    __asm{
        push ecx
        push edx
        push esi
        push ebx
        mov eax,BIOS_SERVICE_READSECTOR //BIOS 扇区读取功能
        mov ecx,nHdNum
        mov edx,nStartSector
        mov esi,nSectorNum
        mov ebx,BIOS_ENTRY_ADDR //模式切换代码所在位置，即内存 6KB 处
        call ebx //调用 6KB 处的代码
        pop ebx
        pop esi
        pop edx
        pop ecx
        cmp eax,0x00
        jz __BIOS_FAILED
        jmp __BIOS_SUCCESS
    }
    __BIOS_FAILED: //BIOS 调用失败
    return FALSE;
    __BIOS_SUCCESS: //BIOS 调用成功
    for(int i = 0;i < 512 * nSectorNum;i++) //把读取的数据复制到 pBuffer 缓冲区
    {
        pBuffer[i] = ((BYTE*)BIOS_HD_BUFFER)[i];
    }
    return TRUE;
}
```

这个函数大部分是使用内嵌汇编语言实现的。首先保存几个通用寄存器，然后把待读取



硬盘号、起始扇区、扇区数量等参数，存放到几个通用寄存器中，以传递给实模式代码。接下来使用 CALL 指令，调用物理内存 6KB 处的代码。这里的代码即是从保护模式切换到实模式，并通过 BIOS 调用读取硬盘扇区的实现代码。BIOS 调用成功完成之后，会把读取的扇区放到预定义的 BIOS_HD_BUFFER 位置。这个宏定义为 8KB，即物理内存 8KB 处。回忆一下操作系统的初始化过程，最初 8KB 处加载的是 miniker.bin 模块。但是一旦切换到保护模式，这个模块的代码被搬到 1MB 开始处，因此 8KB 开始处的内存就空闲了。我们把从 8KB 开始到 640KB 结束的内存空间，作为保护模式和实模式的数据交换空间，即共享内存。

EAX 寄存器存放了一个功能号，这个功能号告诉实模式代码完成什么样的具体功能。毕竟所有的实模式 BIOS 调用都是通过一个统一的入口完成的。实模式 BIOS 调用执行完后重新返回保护模式。这时候 EAX 寄存器存放了执行结果是否成功的标志。

根据 EAX 寄存器的值，决定下一步的动作。在上述函数中，如果 EAX 寄存器的值为 0，说明执行失败，于是跳转到 __BIOS_FAILED 标号处，函数失败返回。否则跳转到 __BIOS_SUCCESS 标号处继续执行。接下来的代码，即把磁盘扇区的内容，由共享内存（物理内存 8KB 处）复制到 pBuffer 缓冲区，然后返回。

其他调用 BIOS 功能的函数也是按照这个思路实现的，不同的是 EAX 存放的功能号和函数参数不同。

知晓了保护模式下调用实模式的功能方法之后，让我们详细解释如何从保护模式切换到实模式。

3.6.3 保护模式切换到实模式

按照 IA32 编程手册的描述，从保护模式跳转到实模式需要两个段选择符：一个用于刷新 DS/SS/ES 等寄存器缓存，称作规范描述符，另一个用于刷新 CS 寄存器，称为 16 位代码描述符，分别定义如下：

1) NormalDesc: FFFF0000 00009200

2) Code16Desc: FFFF0000 00009800

上述两个描述符都是在 miniker.asm 中定义的，其索引（相对 GDT 表头的偏移）分别为 0x30 和 0x38。这两个段描述符与保护模式 CS、DS 等段描述符，位于同一个 GDT 表中。定义好切换需要的段描述符后，看一下 6KB 处的汇编代码：

```
[/kernel/kernel/arch/sysinit/realinit.asm]
align 4 ;4 字节对齐
bits 32 ;跳转到这个地方时，仍然是保护模式、32 位指令代码
jmp __32CODE_BEGIN
;接下来的几个全局变量，用于临时保存保护模式的 IDTR 和 CR3 寄存器值
align 4
__P_IDTR:
    dw 00    ;IDT 表的长度
    dd 00    ;IDT 表的起始地址
__CR3 dd 0x00 ;保存 CR3 寄存器
__32CODE_BEGIN:
    push ebx
```

```

push ecx
push edx
push esi
push edi
push ebp
cli;
sidt [__P_IDTR + 4096];把 IDT 保存到上述 __P_IDTR 处。注调整内存地址
push eax
mov eax,cr3
mov dword [__CR3 + 4096],eax ;保存 CR3 寄存器
mov eax,cr0
and eax,0x7FFFFFFF
mov cr0,eax ;清除 PG 比特，即禁止分页功能
xor eax,eax
mov cr3,eax ;Flush TLB.
pop eax
jmp 0x38 : 4096 + __16BIT_ENTRY ;跳转到 16 位代码段

```

上述代码保存了保护模式下的几个寄存器的值，主要是中断描述符表寄存器 IDTR，CR3 寄存器等。在接下来的代码中，这些寄存器会被修改。这样在重新返回保护模式时，直接从保存的全局变量中恢复即可。接下来的代码是过渡代码，用于初始化实模式下的中断描述符表寄存器，清除 DS/SS 等寄存器值，为正式进入实模式做准备：

```

align 4
bits 16 ;16 比特代码
__16BIT_ENTRY:
jmp __16CODE_BEGIN
align 4
__R_IDTR: ;实模式下的中断描述符表的长度和起始地址，用于初始化实模式的 IDTR
dw 1024
dd 0x00
__16CODE_BEGIN:
mov bx,0x30 ;使用规范段描述符初始化 DS/SS/ES/FS/GS 等段寄存器值
mov ds,bx
mov ss,bx
mov es,bx
mov fs,bx
mov gs,bx
lidt [__R_IDTR + 4096] ;初始化实模式下的中断描述符表寄存器
mov ebx,cr0
and bl,0xFE ;清除 PE 比特
mov cr0,ebx ;正式转入保护模式
jmp 0x100 : __REAL_MODE_ENTRY ;跨段跳转，转移到实模式代码

```

上述 jmp 指令正式跳转到了实模式下的代码处。接下来是实模式的实现代码：

```

__REAL_MODE_ENTRY:
jmp __REALCODE_BEGIN

```



```
align 4
__ESP dd 0x00 ;保存保护模式的 ESP 寄存器
__REALCODE_BEGIN:
    mov bx,cs
    mov ds,bx
    mov ss,bx
    mov es,bx
    mov fs,bx
    mov gs,bx
    mov dword [__ESP],esp ;保存保护模式下的 ESP 寄存器
    mov sp,0xff0
    call __REINIT_8259_EX ;初始化 8259 中断控制器，使之适应实模式下的要求
    sti
    ;实模式执行环境建立完毕，可执行具体的 BIOS 功能调用了
__BIOS_BEGIN:
    mov bx,ax
    shl bx,0x01 ;根据 EAX 寄存器中的功能代码，计算功能函数在功能表中的偏移
    add bx,__BIOS_JMP_TABLE ;__BIOS_JMP_TABLE 是实模式功能表
    mov ax,word [bx]
    call ax ;调用具体的实模式功能函数
    jmp __BACK_TO_PROTECT ;完成后，重新返回保护模式
```

上述实模式功能表实际上是一些代码标号的数组，每个代码标号标注了一段实现特定功能的代码。上述__BIOS_BEGIN 处的代码，根据 EAX 寄存器（目前只有几个功能号，因此使用 ax 就足够了）中存放的功能号，计算出功能函数（标号）在功能表中的偏移，然后调用即可。下面是功能表的样子：

```
align 4
bits 16
__BIOS_JMP_TABLE:
    dw __REBOOT ;重启动
    dw __POWEROFF ;关闭电源
    dw __READSECT ;读磁盘扇区
    dw __DELAY_TEST ;延迟
    dw 0x00 ;以 0 作为功能表结尾
```

具体的功能函数的实现就不详细介绍了，读者可直接阅读 realinit.asm 中的代码，都比较简单。调用 BIOS 功能完成后，接着就需要返回保护模式。下面是从实模式重新返回保护模式的代码：

```
align 4
bits 16
__BACK_TO_PROTECT:
    mov esp,dword [__ESP] ;恢复先前保存的保护模式堆栈寄存器
    cli
    mov ebx,cr0
    or bl,0x01 ;设置 PE 比特
```



```

mov cr0,ebx
jmp 0x08 : __PM_BEGIN + 4096 ;跳转到真正的保护模式代码处

;下面的代码段位于保护模式，32 位
align 4
bits 32
__PM_BEGIN: ;保护模式代码，首先初始化各段寄存器，与初次进入保护模式一样
mov bx,0x10
mov ds,bx
mov bx,0x18
mov ss,bx
mov bx,0x20
mov es,bx
mov fs,bx
mov gs,bx
lidt [__P_IDTR + 4096] ;恢复原先保存的保护模式中断描述符寄存器
mov ebx,dword [__CR3 + 4096] ;恢复原先保存的 CR3 寄存器
mov cr3,ebx
mov ebx,cr0
or ebx,0x80000000 ;使能分页功能
mov cr0,ebx
mov ebx,np_init8259 ;重新设置 8259 的工作模式，使之满足保护模式需要
add ebx,4096
call ebx ;使用绝对地址方式，调用 np_init8259 函数
sti
;恢复保存在保护模式堆栈内的通用寄存器
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
ret ;正式返回保护模式

```

上面的代码比较简单，指令后面的注释足够说明代码的作用了。有两个地方需要进一步说明一下。

1) 在引用全局变量，或者使用绝对地址跳转的时候，都是在标号基础上再加上 4096，比如在恢复 CR3 寄存器的时候，使用的指令是 `mov ebx,dword [__CR3 + 4096]`。这里 `__CR3` 标号是一个全局变量，保存了保护模式下的 CR3 寄存器。之所以增加 4096，是因为 `__CR3` 标号的偏移值是相对于 `realinit.bin` 模块的，而 `realinit.bin` 模块被加载到了物理内存的 4K 位置处。因此在引用绝对地址的时候，需要在标号基础上加上 4K。

2) 在切换到实模式时，由于中断控制器 8259 原先工作在保护模式下，其第一个中断向量号是 32，这不符合实模式的需要。在实模式下，外部中断是从 8 开始的，因此必须对

8259 重新进行初始化。同样的道理，在切换回保护模式后，又需要重新对其进行设置，使其第一个中断向量号变为 32。由于在前面没有讲解 8259 芯片的初始化方法，再次补充一下，下面是把 8259 从保护模式转换为实模式的代码：

```
__REINIT_8259_EX:
    push ax
    mov al,0x11
    out 0x20,al
    out 0xa0,al
    mov al,0x08
    out 0x21,al
    mov al,0x70
    out 0xa1,al
    mov al,0x04
    out 0x21,al
    mov al,0x02
    out 0xa1,al
    mov al,0x01
    out 0x21,al
    out 0xa1,al
    mov al,0xb8
    out 0x21,al
    mov al,0x8f
    out 0xa1,al
    pop ax
    ret
```

代码比较繁琐，但是逻辑比较简单，就是通过向 8259 芯片的控制字寄存器和命令字寄存器中写入一些值，使之满足实模式的需要。各行代码的具体含义，在这里就不详细讲了，读者可到网络上搜索相关资料进行对比分析。关于 8259 中断控制器的编程，不论是在互联网上还是在计算机接口课程中，都可以找到非常多的资料。

好了，从保护模式切换到实模式的过程介绍完了。虽然这个过程对整个操作系统的核心机制来说不是很重要，但是比较复杂。作者在实现这个切换过程的时候，遇到了非常多的问题，足足用了一个星期，才初步搞定。之所以说是初步搞定，是因为上述代码在大多数情况下都是正常工作的，但是在一些个别情况下，仍然会引发异常。具体原因到现在也没有彻底搞清楚。好在从保护模式切换回实模式只是为了试验目的和解决临时问题而存在的，在正式的商用场合，不会采用这种方式，因此问题还不算非常严重。

另外需要说明的是，从实模式切换到保护模式的资料非常多，但是从保护模式切换到实模式的资料却非常少，且在已有的少量资料和实例中，大多是针对 CPU 本身工作模式的切换。须知在实际系统中，仅仅 CPU 切换回来是不够的，与之配套的系统辅助部件的工作模式也必须对应切换，比如 8259 中断控制器的切换、中断描述符表的切换等。作者在这里描述的切换过程，完全是经过独立分析之后得出的方法，没有任何实现作为参考，可以说是纯粹的原创。

3.7 引导和初始化总结

本章对通用计算机和嵌入式系统的初始化过程做了简要描述，在此基础上分析了 Hello China 操作系统的加载和初始化过程。最后简要介绍了字符模式 shell 的实现机制和从保护模式到实模式的切换。希望通过本章内容的介绍，读者对计算机操作系统的加载和初始化过程有一个比较深入的理解。计算机启动过程相关的技术资料非常多，但是与具体操作系统相结合来介绍的资料并不是很多，希望这种理论和实际结合的方式，能够取得更好的展示效果。

当然，囿于篇幅，很多基本的内容没有深入讲解，只是做了简略介绍，比如 Intel CPU 的实模式寻址方式、保护模式工作原理等。而这些内容又是理解本章内容的基础。因此如果阅读本章内容感到吃力，建议读者先通过网络等方式补习一下这些基础知识，再来阅读本章，会有不一样的收获。之所以不重点介绍这些内容的另一个原因，是因为这些内容的学习资料很容易获得，随便搜索一下，便可在互联网上找到大量浅显易懂的文章。

第 4 章 Hello China 线程的实现

4.1 进程、线程和任务

线程是 Hello China 操作系统的任务模型，本章将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。

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，然后按回车键），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，因为不同进程的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，只引入了线程的概念，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，然后开始新一轮循环。

4.2 Hello China 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（为了描述方便，删除了部分注释）：

```

[kernel/include/ktmgr.h]
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*          lpTerminalQueue;
    __PRIORITY_QUEUE*          ReadyQueue[16];
    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          (*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,
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()

```

这是一个比较大的对象，其中，四个优先级队列和一个优先级队列数组用于对系统中的线程进行管理，每个队列的用途见表 4-1。

表 4-1 Hello China 的线程队列

队列或队列数组	用 途	对应的线程状态
lpRunningQueue	所有当前运行的线程对象，存储在该队列 ^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
ReadyQueue[16]	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

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，有与系统中 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。

这些线程的管理队列比较容易理解，处于某一状态的核心线程，会被放入对应状态的队列。线程在不同状态之间的转换，实际上就是不同队列之间的转移操作。比如，线程状态从睡眠转到就绪，则核心线程对象会被从睡眠队列中删除，然后加入就绪队列。需要注意的是，就绪队列不是一个单一的队列，而是一个包含 16 个优先队列对象的队列数组，数组中的每个元素，对应相应优先级的核心线程队列。图 4-1 说明了这种关系。

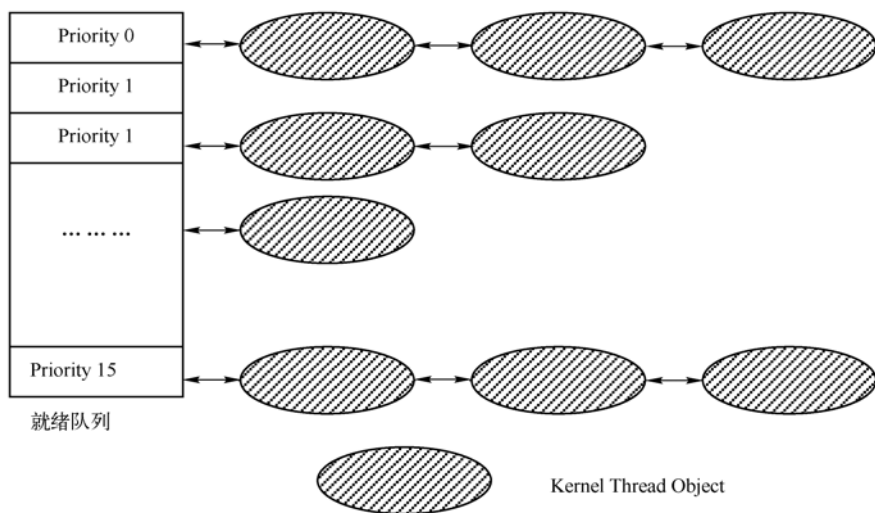


图 4-1 Hello China 的核心线程队列

缺省情况下，Hello China 采用的是严格基于线程优先级的调度方式。优先级为 0 的核心线程的调度优先级最高，如果优先级为 0 的就绪队列中存在就绪的核心线程，那么优先级是 1 或者大于 1 的核心线程，将没有机会得到调度。相同优先级的核心线程，按照轮转方式调度，即首先调度队列中的第一个核心线程，运行一个时间片后，第一个线程会被重新插入就绪队列的尾部，然后再调度队列中的第二个核心线程，这样依此类推。这种调度方式符合大部分应用场景下的需求，尤其是嵌入式系统的需求。但存在低优先级线程饿死的问题。不过可通过修改调度算法，来避免这个问题。

在 Hello China V1.75 的实现中，线程调度算法被封装到两个函数中——GetScheduleKernelThread 和 AddReadyKernelThread。第一个函数的功能是从就绪队列数组中，提取一个优先级最高的就绪状态的核心线程。操作系统的调度程序会调用这个函数，选择一个就绪状态的核心线程去执行。而 AddReadyKernelThread 的作用则相反，用于向就绪队列数组中增加一个状态为 READY 的核心线程。在核心线程状态转换的时候，比如从睡眠状态转换到就绪状态，转换代码会调用该函数，把就绪线程加入就绪队列数组中。在当前的实现中，这两个函数都是按照严格优先级的调度方式实现的。下面看一下其代码：

```
[kernel/kernel/ktmgr.cpp]
__KERNEL_THREAD_OBJECT* GetScheduleKernelThread(
__KERNEL_THREAD_MANAGER* lpThis,
DWORD dwPriority)
{
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    for(DWORD i = dwPriority; i <= MAX_KERNEL_THREAD_PRIORITY; i++)
    {
        lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpMgr->
ReadyQueue[MAX_KERNEL_THREAD_PRIORITY - i]->GetHeaderElement(
    (__COMMON_OBJECT*)ReadyQueue[i - 1],
    NULL); //获取队列中的第 1 个元素
    }
```



```
        if(lpKernelThread) //Get one.
        {
            return lpKernelThread;
        }
    }
    return lpKernelThread; //Can not find.
}
```

函数代码比较简单，无非是按照从低到高的顺序，依次检查就绪队列数组，试图从优先级最高的就绪队列中提取一个核心线程。如果提取成功（GetHeaderElement 返回核心线程的指针），则直接返回这个核心线程对象，否则继续检查下一个优先级的就绪队列。如果所有就绪队列中都没有核心线程对象，则返回 NULL。

下面是 AddReadyKernelThread 的实现代码：

```
[kernel/kernel/ktmgr.cpp]
static void AddReadyKernelThread(__COMMON_OBJECT* lpThis,
__KERNEL_THREAD_OBJECT* lpKernelThread)
{
    if(lpKernelThread->dwThreadPriority > MAX_KERNEL_THREAD_PRIORITY)
    {
        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。

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（被其他线程占用），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
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他线程中断）
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，是操作系统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，示意代码如下：

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

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

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

并假设这三个线程对 Sleep 函数的调用发生在同一个时间片内，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，然后与当前的系统 tick 数（System 对象维护，详细信息请参考第 8 章）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 与当前的时钟 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，插入就绪队列数组。

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，其参数列表也与其他对象一样，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。当共享资源被其他线程释放的时候，会重新修改当前等待该共享资源的线程（Blocked 线程），把其状态修改为 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，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，调度程序只选择就绪队列中的线程投入运行。

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。

从图中可以看出，Running 状态只能从 Ready 状态转换过来，Blocked 状态和 Suspended 状态只能从 Running 状态转换过来。

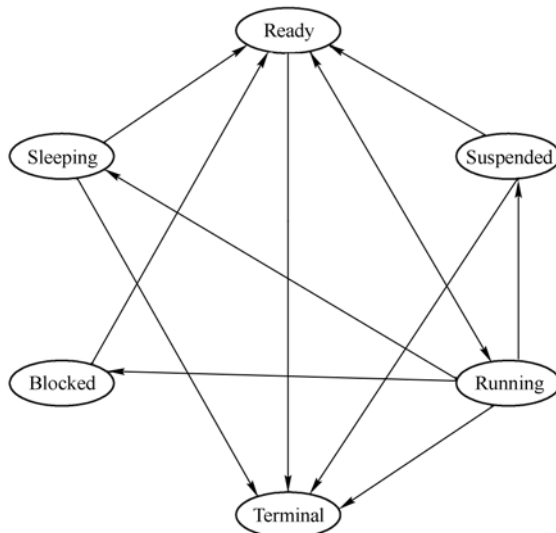


图 4-2 核心线程的状态及其之间的转换

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 其他线程调用	TerminalKernelThread 其他线程调用	运行完	——	TerminalKernelThread 其他线程调用	——

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 操作系统中的一个线程，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，核心线程管理对象是一个全局对象，记录了系统中的全局信息，并提供了操作线程的函数。而核心线程对象则是针对一个具体的核心线程的，一个核心线程有一个与之对应的核心线程对象。下面是核心线程对象的定义：

```

[kernel/include/ktmgr.h]
BEGIN_DEFINE_OBJECT(_KERNEL_THREAD_OBJECT)
    INHERIT_FROM_COMMON_OBJECT
    INHERIT_FROM_COMMON_SYNCHRONIZATION_OBJECT
    __KERNEL_THREAD_CONTEXT*      lpKernelThreadContext;
    DWORD                        dwThreadId;
    DWORD                        dwThreadStatus;
    __PRIORITY_QUEUE*            lpWaitingQueue;
    DWORD                        dwThreadPriority;
    DWORD                        dwReturnValue;
    DWORD                        dwTotalRunTime;
    DWORD                        dwTotalMemSize;
    LPVOID                       lpHeapObject;
    LPVOID                       lpDefaultHeap;
    BOOL                         bUsedMath;
    DWORD                        dwStackSize;
    LPVOID                       lpInitStackPointer;
    DWORD                        (*KernelThreadRoutine)(LPVOID);
    LPVOID                       lpRoutineParam;

    __KERNEL_THREAD_MESSAGE      KernelThreadMsg[MAX_KTHREAD_MSG_NUM];
    UCHAR                        ucMsgQueueHeader;
    UCHAR                        ucMsgQueueTail;
    UCHAR                        ucCurrentMsgNum;
    UCHAR                        ucAlign;
    __PRIORITY_QUEUE*            lpMsgWaitingQueue;
    __EVENT*                     lpMsgEvent;
    DWORD                        dwLastError;
END_DEFINE_OBJECT()

```

上述代码为了版面清晰，省略了相关注释。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 相关的硬件信息	lpKernelThreadContext	一个指向硬件上下文的指针，硬件上下文的定义与特定 CPU 相关
线程属性信息	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

(续)

类 别	类 别 含 义	对 应 的 变 量	变 量 含 义
消息队列信息	完成线程的消息队列控制功能	ucMsgQueueTrial	消息尾索引
		ucCurrentMsgNum	当前消息队列中的消息数
		lpMsgWaitingQueue	等待队列，当线程等待消息到达时，将被加入该队列
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
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
线程函数信息	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内容。

(1) 硬件上下文：这是一个指向 `_KERNEL_THREAD_CONTEXT` 类型的指针，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他状态（如就绪状态）时，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，这个结构的定义与具体的 CPU 有关，即不同的 CPU，该结构的定义也不一样。详细情况请参考 4.2.4 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。

(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 `ucMsgQueueTrial` 两个变量记录了环形队列的头和尾，`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：与其他同步对象（比如 `Event`、`Mutex` 等）一样，核心线程对象也是一个同步对象，不同的是，核心线程对象的状态不能人为地通过 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），所有其他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) 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。

(7) 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 `_KTHREAD_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。本书重点关注 Intel IA32 构架的 CPU。在这种硬件平台下，该结构体的定义如下（为了解释简便，删除了部分注释）：

```
[kernel/include/ktmgr.h]
BEGIN_DEFINE_OBJECT(_KTHREAD_CONTEXT)
    DWORD        dwEFlags;
    WORD         wCS;
    WORD         wReserved;
    DWORD        dwEIP;
    DWORD        dwEAX;
    DWORD        dwEBX;
    DWORD        dwECX;
    DWORD        dwEDX;
    DWORD        dwESI;
    DWORD        dwEDI;
    DWORD        dwEBP;
END_DEFINE_OBJECT()
```

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
ESI	DwESI	0
EDI	dwEDI	0
EBP	dwEBP	0

按照 IA32 的体系结构，EFLAGS 寄存器的内容如图 4-3 所示。

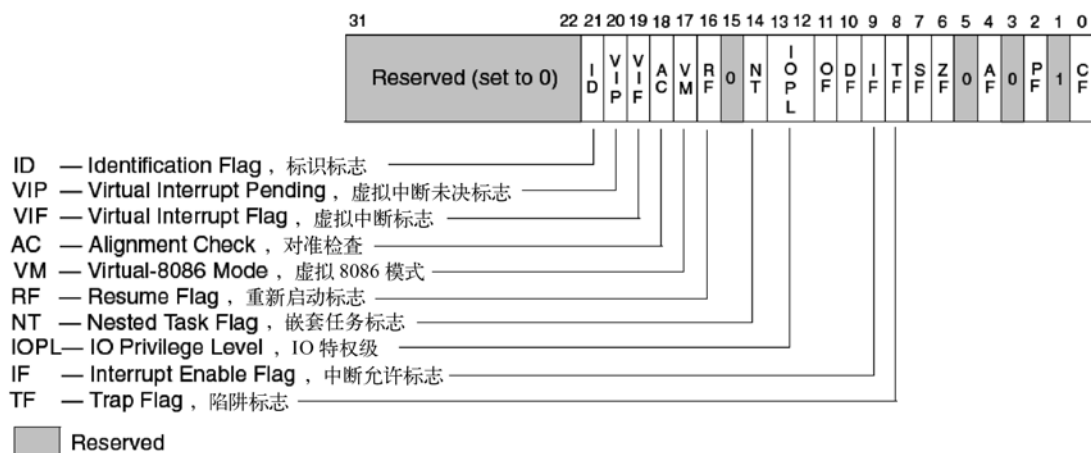


图 4-3 EFLAGS 寄存器各位的含义

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，也就是在上述寄存器标志位中，只把 IF (中断允许标志) 设置为 1，这样允许中断。所有其他标志均设置为 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 (代码段在 GDT 中的索引值)。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，堆栈寄存器 ESP 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，详细信息可参考 4.2.5 节)。

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他的工作，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：

```
[kernel/kernel/ktmgr.cpp]
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                        dwRetVal              = 0L;
    DWORD                        dwFlags               = 0L;

    ... ..

    if(NULL == lpKernelThread->KernelThreadRoutine) //如果未指定线程的函数功能
        goto __TERMINAL;

    dwRetVal =
```



```
lpKernelThread->KernelThreadRoutine(lpKernelThread->lpRoutineParam);
```

```
__ENTER_CRITICAL_SECTION(NULL,dwFlags);
lpKernelThread->dwReturnValue    = dwRetVal;
lpKernelThread->dwThreadStatus  = KERNEL_THREAD_STATUS_TERMINAL;
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
.....
}
```

上述代码中的黑体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 4.2.5 节）。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`），唤醒等待该线程的其他核心线程等。

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。

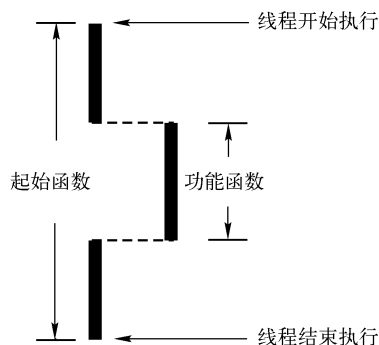


图 4-4 核心线程的生命周期

4.2.5 线程的创建和初始化

`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,
    LPSTR lpszName)
{
    __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))
    goto __TERMINAL;

```

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，所有以其他值调用该函数的尝试都会失败。

```

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*)lpKernelThread))
    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，将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)
    dwStackSize = DEFAULT_STACK_SIZE;
else
{
    if(dwStackSize < KMEM_MIN_ALLOCATE_BLOCK)
        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。

```

lpKernelThread->dwThreadId      = lpKernelThread->dwObjectID;
lpKernelThread->dwThreadStatus   = dwStatus;
lpKernelThread->dwThreadPriority = dwPriority;
lpKernelThread->dwScheduleCounter = dwPriority;
lpKernelThread->dwReturnValue    = 0L;
lpKernelThread->dwTotalRunTime   = 0L;
lpKernelThread->dwTotalMemSize   = 0L;
lpKernelThread->lpCurrentDirectory = NULL;
lpKernelThread->lpRootDirectory  = NULL;
lpKernelThread->lpModuleDirectory = NULL;

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;

```

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。

```

if(lpszName)
{
    for(i = 0; i < MAX_THREAD_NAME - 1; i++)
    {
        if(lpszName[i] == 0) //End.
        {
            break;
        }
        lpKernelThread->KernelThreadName[i] = lpszName[i];
    }
    lpKernelThread->KernelThreadName[i] = 0;
}

```

上面这段代码设置了核心线程的线程名。在当前的实现中，核心线程的线程名字最大不能超过 16B，因此为了安全起见，不使用诸如 `strcpy` 等函数，而是采取最“笨拙”也是最安全的逐个字节复制的方式来实现。

```
InitKernelThreadContext(lpKernelThread, KernelThreadWrapper);
```

`InitKernelThreadContext` 函数初始化了核心线程对象的硬件上下文，具体的初始化方法，本章后续会介绍。

```
if(KERNEL_THREAD_STATUS_READY == dwStatus)
{
    lpMgr->AddReadyKernelThread((__COMMON_OBJECT*)lpMgr,
        lpKernelThread);
}
else
{
    if(!lpMgr->lpSuspendedQueue->InsertIntoQueue((__COMMON_OBJECT*)lpMgr->
lpSuspendedQueue,
    (__COMMON_OBJECT*)lpKernelThread,dwPriority))
        goto __TERMINAL;
}
```

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 函数会把线程加入就绪队列数组。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。

```
lpMgr->CallThreadHook(THREAD_HOOK_TYPE_CREATE,lpKernelThread,
    NULL);
bSuccess = TRUE;
__TERMINAL:
if(!bSuccess)
{
    //失败处理代码
}
else
{
    return lpKernelThread;
}
```

上述代码中最主要的一个地方，就是调用了 CallThreadHook 函数。在 Hello China V1.75 的实现中，采用了一种回调机制，用户可自行创建一个函数，然后注册到系统（实际上是核心线程管理器对象）中。当核心线程被创建、被销毁、被换出 CPU、被调入 CPU 时，将会调用这个回调函数，从而完成一些系统级的任务。最典型的一个应用就是线程运行时间的统计功能。可创建一个回调函数，在线程被调度到 CPU 开始执行的时候，记录下当时的系统时间戳。当线程被调度出 CPU 的时候，再次记录当时的时间戳。这样对比两个时间戳，就可得到线程本次被调度的执行时间。这样累加起来，就可以得到线程的完整执行时间。具体的回调机制的实现，在本书后面将会讲到。

最后，CreateKernelThread 如果执行失败，则必须进行一些资源清除工作，比如释放线程的堆栈、释放创建的核心线程对象等。如果执行成功，则直接返回创建成功的核心线程对

象指针即可。

相信读者对核心线程的创建过程已经有一个基本的了解了。在上面的描述中，曾提到 `CreateKernelThread` 函数通过调用 `InitKernelThreadContext` 函数，完成了核心线程上下文的初始化。核心线程上下文是与 CPU 紧密相关的一些硬件信息的集合，下面详细讲解 `InitKernelThreadContext` 函数的实现。先看代码（为了描述简便，代码有所精简）：

```
[kernel/arch/arch_x86.cpp]
VOID InitKernelThreadContext(__KERNEL_THREAD_OBJECT* lpKernelThread,
                             __KERNEL_THREAD_WRAPPER lpStartAddr)
{
    DWORD*      lpStackPtr = NULL;
    DWORD       dwStackSize = 0;

    lpStackPtr = (DWORD*)lpKernelThread->lpInitStackPointer;
    __PUSH(lpStackPtr,lpKernelThread);
    __PUSH(lpStackPtr,NULL);
    __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，代码如下：

```
#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，建立了如图 4-5 所示的堆栈框架。

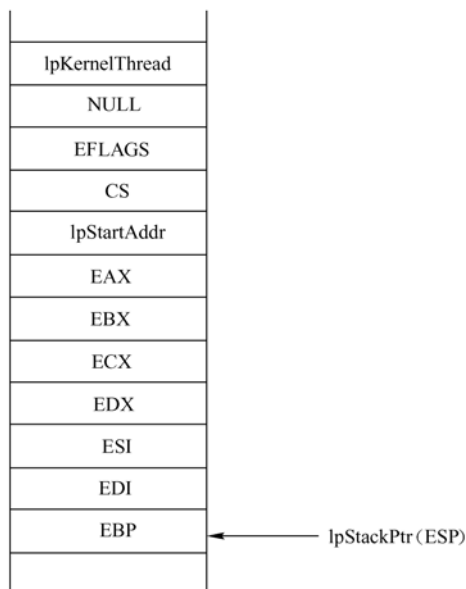


图 4-5 新创建的核心线程的堆栈框架

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，InitKernelThreadContext 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后，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）。KernelThreadWrapper 函数的原型如下：

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

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。如果读者对 iretd 和 call 指令的执行过程不是很清楚，那么可能会感到迷惑。在此再进一步解释一下这个过程。

InitKernelThreadContext 函数把 lpStackPtr 赋值给核心线程对象的上下文指针之后，核心线程创建的工作就结束了。在核心线程得到调度的时候，调度程序会恢复线程的硬件上下文信息。恢复的方法就是把 CPU 的 ESP 寄存器（堆栈指针）替换为核心线程对象的硬件上下文指针（lpKernelThreadContext），然后依次恢复 EAX 到 EBP 等几个通用寄存器，执行 iretd 指令。iretd 指令的执行过程是，从堆栈中依次恢复 EIP 寄存器、EFlags 寄存器和 CS 寄存器，然后接着执行。按照上述过程，在核心线程完成创建，第一次被调度执行的时候，是从 KernelThreadWrapper 函数处开始执行的，即 iretd 指令执行完成后，CPU 的执行线索就跳转到了 KernelThreadWrapper 处。注意，这里不是通过 call 指令跳到这个函数处的，而是通过 iretd 指令。但 KernelThreadWrapper 函数却不知道这些，它依然会按照传统的方式（即认为是被 call 指令调用）去访问函数的参数，即核心线程对象指针（lpKernelThread）。

而传统方式对函数参数的访问方法，则认为堆栈框架是通过 `call` 指令构建的，即首先压入函数参数，然后压入函数调用者调用 `KernelThreadWrapper` 时的指令偏移（返回地址）。因此我们必须模拟这种堆栈框架，以适应 `KernelThreadWrapper` 函数本身的要求。模拟方法就是，首先把该函数的参数（`lpKernelThread`）压入堆栈，然后压入 4 个空字节模拟返回地址。这样在 `KernelThreadWrapper` 函数看来，就是一个通过 `call` 指令建立起来的堆栈框架，于是就可正常访问 `lpKernelThread` 参数。

需要注意的是，在堆栈中压入一个“空”双字（4 字节），并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 4.2.6 节。

4.2.6 线程的结束

线程的结束有两种方式，一种是线程执行完功能函数，自然结束。另一种是被其他线程调用 `TerminalKernelThread` 函数强行终止。

其中，第一种结束情况属正常情况，这种方式不会发生资源泄漏等情况，而采用第二种方式，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 4.2.4 节中有介绍，本节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。

```
[kernel/kernel/ktmgr.cpp]
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;

    //正式执行线程功能函数前的准备工作。
    ... ..
    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(NULL,dwFlags);
```



```
lpKernelThread->dwReturnValue = dwRetVal;  
lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_TERMINAL;  
__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)。

```
lpWaitingQueue = lpKernelThread->lpWaitingQueue;  
lpReadyQueue = KernelThreadManager.lpReadyQueue;  
lpWaitingThread = (__KERNEL_THREAD_OBJECT*)lpWaitingQueue->GetHeaderElement((__COMMON_OBJECT*)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他线程。核心线程对象本身也是一个同步对象，其他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他线程将被激活（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 4.2.3 节。

```
__TERMINAL:  
    KernelThreadManager.lpTerminalQueue->  
>InsertIntoQueue((__COMMON_OBJECT*)KernelThreadManager.lpTerminalQueue,  
        (__COMMON_OBJECT*)lpKernelThread,  
        0L);  
    KernelThreadManager.ScheduleFromProc(NULL);  
    return;  
}
```

上述代码把当前线程核心对象插入终止队列 (lpTerminalQueue)，然后调用 ScheduleFromProc 函数，引发一个重新调度操作。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，在合适的时机会被系统删除。

4.2.7 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[32];
UCHAR                        ucMsgQueueHeader;
UCHAR                        ucMsgQueueTail;
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（在实际代码中，以宏定义 MAX_KTHREAD_MSG_NUM 替代硬编码的 32）。ucAligment 是为了实现数据对齐（32bit 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 4-6 所示。

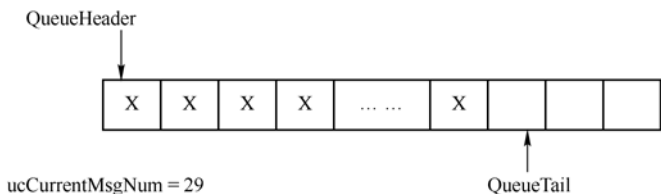


图 4-6 线程的消息队列

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 4-7 所示。

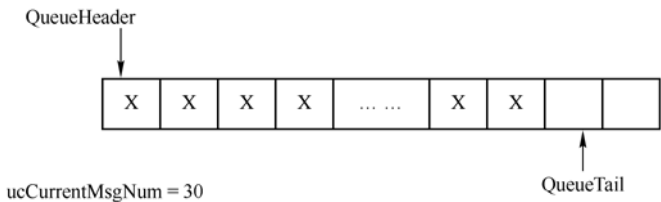


图 4-7 线程消息队列的添加操作

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），直到有其他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 4-8 所示。

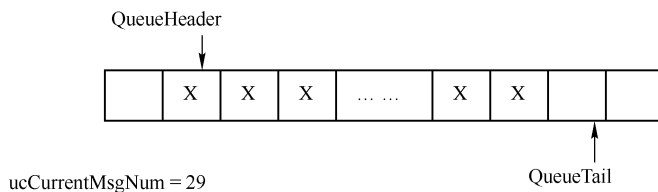


图 4-8 线程消息队列的删除操作

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户程序；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 GetMessage(__COMMON_OBJECT*
lpThread, __KERNEL_THREAD_MESSAGE* lpMsg)
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread = NULL;
    DWORD                       dwFlags       = 0L;

    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpThread;
    if(MsgQueueEmpty(lpThread))
    {
        lpKernelThread->lpMsgEvent->WaitForThisObject(
            (__COMMON_OBJECT*)(lpKernelThread->lpMsgEvent));
    }
    ....
    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(MsgQueueFull(lpThread))                //If the queue is full.
        return bResult;
    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpThread;
    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，此时就会被唤醒。

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;  
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` 是两个与 `wCommand` 关联的参数，比如，与“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。

4.2.8 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，即任何一个中断发生后，中断处理程序处理结束后，都会重新检查线程的就绪队列数组，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。

这样就涉及一个问题：在中断上下文中，如何保存当前线程的上下文状态，并选择另外一个线程，恢复其上下文，并投入运行？本节对这个问题进行详细描述。

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（没有考虑不同优先级之间的转换，比如用户态和核心态，而只考虑在核心态保护模式下的情况）：

（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）。

（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，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，中断发生后，CPU 跳转到中断处理程序前，当前线程堆栈的堆栈框架如图 4-9 所示。

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。

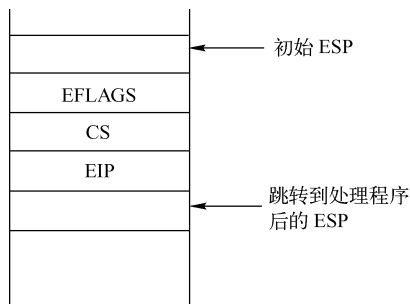


图 4-9 中断发生后的堆栈框架

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 线程的其他相关寄存器（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）。执行完毕，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成，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。

线程切换的机制清楚后，再来看 Hello China 实现在中断上下文中切换线程的细节部分。在当前的 Hello China 的实现中，中断处理程序被分成两部分实现。

（1）中断处理程序入口，采用汇编语言实现，该部分保存当前线程的寄存器（通用寄存器）信息，并把中断向量号压入堆栈，然后调用采用 C 语言实现的中断处理程序。

（2）C 语言实现的中断处理程序，根据压入的堆栈号，再调用特定的中断处理函数（详细的中断处理过程请参考第 8 章）。

采用汇编语言实现的中断处理入口程序，对所有的中断和异常都是类似的，代码如下（为了解释方便，代码做了精简）：

```
np_int20:
    push eax
    cmp dword [gl_general_int_handler],0x00000000
    jz .ll_continue
    push ebx
    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
    pop eax
    mov esp,eax
    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx
```

```

.ll_continue:
    mov al,0x20
    out 0x20,al
    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 模块没有加载，`gl_general_int_handler` 实际上是定义了一个函数指针，由 master 模块在初始化的时候，把这个指针填写为 `GeneralIntHandler` 函数），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（`GeneralIntHandler` 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 4-10 所示。

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 寄存器的值是压入 EBP 后 ESP 的值。之所以保存该值，是因为 `gl_general_int_handler` 函数可以通过该值访问堆栈框架。

`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，`GeneralIntHandler` 函数就会调用对应的例程。

所有在中断上下文下的线程调度工作，是通过一个函数 `ScheduleFromInt` 来实现的。中断处理程序在处理完所有其他任务后，调用该函数。这个函数实现了核心线程的重新调度，即选择优先级最高的就绪线程，然后恢复执行。下面是该函数的实现代码，为了阅读方便起见，分段进行解释（同时代码做了精简）。

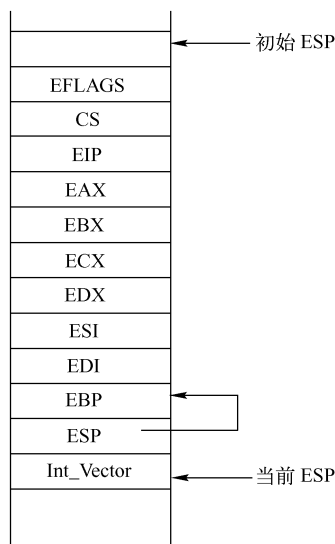


图 4-10 当前线程的各寄存器在堆栈中的布局

```
[kernel/kernel/ktmgr.cpp]
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;

    lpMgr = (__KERNEL_THREAD_MANAGER*)lpThis;
    if(NULL == lpMgr->lpCurrentKernelThread)
    {
        lpNextThread =
            (__KERNEL_THREAD_OBJECT*)KernelThreadManager.lpReadyQueue->
            GetHeaderElement(
                (__COMMON_OBJECT*)KernelThreadManager.lpReadyQueue,
                NULL);
        if(NULL == lpNextThread)
        {
            BUG();
        }
        KernelThreadManager.lpCurrentKernelThread = lpNextThread;
        lpNextThread->dwThreadStatus = KERNEL_THREAD_STATUS_RUNNING;
        lpThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
        __SwitchTo(lpThread->lpKernelThreadContext);
        return;
    }
}
```

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（ScheduleFromInt）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，在后面会详细描述，现在只要知道，一旦以目标线程的硬件上下文信息调用了__SwitchTo 函数，就会切换到目标线程开始运行。

```
else
{
    lpCurrentThread->lpKernelThreadContext =
        (__KERNEL_THREAD_CONTEXT*)lpEsp;
    switch(lpCurrentThread->dwThreadStatus)
    {
```

```

//处于下列状态的核心线程，会被允许继续执行
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)
    {
        lpCurrentThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
        __SwitchTo((__KERNEL_THREAD_CONTEXT*)lpEsp);
        return; //Should not reach here.
    }
    else //调度优先级更高的线程
    {
        lpCurrentThread->dwThreadStatus =
            KERNEL_THREAD_STATUS_READY; //Change status.
        lpMgr->AddReadyKernelThread((__COMMON_OBJECT*)lpMgr,
            lpCurrentThread);
        lpThread->dwThreadStatus =
            KERNEL_THREAD_STATUS_RUNNING;
        lpThread->dwTotalRunTime += SYSTEM_TIME_SLICE;
        lpMgr->lpCurrentKernelThread = lpThread;
        __SwitchTo(lpThread->lpKernelThreadContext);
        return; //Should not reach here.
    }
}
default:
{
    BUG();
}
}
}
}

```

上面这一段代码相对比较复杂，而且内部关系紧密，不容易拆开解释，因此放在下面统一解释，希望读者能够真正理解这段代码的含义。这段代码可以说是整个 Hello China 操作

系统调度程序的核心。

代码首先把核心线程的上下文（即 `lpEsp` 指针，由 `GeneralIntHandler` 传递过来，指向线程的堆栈框架）保存起来，然后根据线程的不同状态，做不同的处理。首先，对于下列几种状态，调度程序是“直接放行”的，即允许当前线程（实际上是中断发生时的线程）继续执行。因为下面这些状态都是“过渡状态”，继续执行很短一段时间后，会马上让出 CPU。

(1) `KERNEL_THREAD_STATUS_SUSPENDED`

在核心线程被挂起的时候，会处于这种状态。之所以产生这种状态的核心线程，是因为当前核心线程在调用 `SuspendKernelThread` 函数时，把自己指定为待挂起线程。该函数首先设置当前运行核心线程的状态为 `KERNEL_THREAD_STATUS_SUSPENDED`，并放入挂起队列，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`。这是一种临时状态，会在很短的时间内被切换出 CPU。因此，发生中断的时候，若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。

(2) `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。

(3) `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。

(4) `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，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。

处于上述状态的核心线程，说明已经打算主动让出 CPU 了。既然人家态度很明确，已经想走了，调度程序就表现得很“大度”，礼貌性地“挽留”一下，或者至少不要主动“驱逐”人家。因此对于上述几种状态的核心线程，调度程序采取“直接放行”的策略。但在放

行之前，也会增加其执行时间（增加一个时间片），然后调用 `__SwitchTo` 函数，重新切换到当前核心线程继续执行。这就是上述代码中 `switch` 语句开始处的几个连续 `case` 的含义。

接下来，如果核心线程的状态是 `RUNNING`，则说明线程被中断打断的时候，处于正常执行状态，这时候就必须切换处理了。既然你没有退出的意思，而且你的时间到了，那么不客气，调度程序就只能下逐客令了。但这时候又要分两种情况进行处理，第一种情况是，当前核心线程的优先级足够高，就绪队列数组中没有比它更优先的线程。这种情况下，调度程序不得不“厚着脸皮”，重新把当前线程“请回来”继续执行。这就是 `case KERNEL_THREAD_STATUS_RUNNING` 语句后前半部分的处理动作。具体的实现方式是，使用当前核心线程的优先级调用 `GetScheduleKernelThread`，试图从就绪队列中找到一个比当前核心线程优先级更高的就绪线程。如果返回 `NULL`，说明就绪队列中没有比当前核心线程优先级更高的线程，于是恢复当前核心线程的上下文，使之继续执行。但在恢复其执行之前，需要增加其运行时间。

另外一种情况是，调用 `GetScheduleKernelThread` 函数成功，返回一个核心线程对象。这时候调度程序就毫不含糊了，决然放弃当前核心线程的执行，选择返回的线程恢复执行。这个过程也很简单，首先处理当前线程的收尾工作，主要是把其状态修改为 `READY`，并加入就绪队列。然后启动新线程的恢复工作，先把其状态修改为 `RUNNING`，增加其运行时间片，同时把当前核心线程指针（`KernelThreadManager` 维护的一个全局变量，`lpCurrentKernelThread`）保存起来，然后调用 `__SwitchTo` 函数，切换到目标线程继续执行。

所有不属于上述情况（即上述 `switch` 块中的 `default` 语句）的线程状态，都会是一种异常状态，即内核 BUG。这时候会调用 `BUG` 函数，打印出 BUG 发生时的相关诊断信息，然后整个系统停止运行。这与 Windows 操作系统的蓝屏异常情况类似，一旦遇到这种情况，说明系统核心数据结构已不连续，操作系统必须停止运行。当然，这种情况很少发生，除非用户程序直接修改（不是通过调用 API 修改）线程核心对象的状态。

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）所有硬件中断处理结束后，线程都会被重新调度。这一步是由 `ScheduleFromInt` 函数完成的。

（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 `__SwitchTo` 函数切换到目标线程。

（6）对于状态是 `SUSPENDED`、`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。

底层函数 `__SwitchTo` 实现了核心线程的切换，其实现是与特定 CPU 的架构相关的。在移植 Hello China 操作系统的核心时，对调度程序的移植，只需要修改这个函数即可，`ScheduleFromInt` 等函数可不作任何修改，这样就大大提升了系统内核的可移植性。这个函数

的具体实现，在本章的后续部分会有详细介绍，目前只要记住，以核心线程的硬件上下文指针调用该函数，即可切换到对应的核心线程。

4.2.9 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，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，称为“系统调用上下文中的切换”。在 `Hello China` 的实现中，任何一个系统调用结束后，都会执行核心线程的重调度工作，这样可确保最高优先级的线程总能得到及时的调度。

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，唯一的不同是，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，堆栈框架是由 `CALL` 指令建立的。系统调用上下文中的线程调度，是通过调用 `ScheduleFromProc` 函数来实现的。这个函数的参数是当前核心线程的上下文指针，但也可以以 `NULL` 为参数调用该函数，这时候该函数会通过 `KernelThreadManager` 维护的全局变量 `lpCurrentKernelThread`，来获取到当前核心线程的硬件上下文。下面通过一个比较典型的系统调用 `WaitForEventObject`（事件对象等待函数，对用户来讲，统一用 `WaitForThisObject` 来呈现）的实现，来说明 `ScheduleFromProc` 函数的使用方式：

```
[kernel/kernel/synobj.cpp]
static DWORD WaitForEventObject(__COMMON_OBJECT* lpThis)
{
    __EVENT*                lpEvent                = NULL;
    __KERNEL_THREAD_OBJECT* lpKernelThread          = NULL;
    __KERNEL_THREAD_CONTEXT* lpContext              = NULL;
    DWORD                   dwFlags                 = 0L;

    lpEvent = (__EVENT*)lpThis;
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(EVENT_STATUS_FREE == lpEvent->dwEventStatus)
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return OBJECT_WAIT_RESOURCE;
    }
    else
    {
        lpKernelThread = KernelThreadManager.lpCurrentKernelThread;
        lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_BLOCKED;
        __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`，然后插入事件对象的等待队列。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` 函数，来引发一个重新调度。

下面重点考察 `ScheduleFromProc` 函数的实现。明白了这个函数，系统调用上下文的线程重调度机制就清楚了。但与 `ScheduleFromInt` 一样，这个函数还是有些复杂的。倒不是逻辑有多复杂，而是这个函数是一个整体，不像其他函数一样，可以比较清楚地分成几个部分单独讲解。但如果读者对 `ScheduleFromInt` 函数理解清楚了，那么对 `ScheduleFromProc` 函数的理解应该也不会困难，这两个函数的实现逻辑，大致上是类似的。下面先考察函数的源代码，为了解释方便，我们对源代码做了简化：

```

[kernel/kernel/ktmgr.cpp]
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 has the topest priority.
        {
            lpCurrent->dwTotalRunTime += SYSTEM_TIME_SLICE;
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            return; //Allow current thread to continue 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,
            &lpNew->lpKernelThreadContext);
        __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_STATUS_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，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。但与 ScheduleFromInt 不同，ScheduleFromInt 函数对于非 RUNNING 状态的临时状态，采取的是放行的策略，因为处于这种临时状态的核心线程，将很快自动放弃执行（实际上，处于这些临时状态的核心线程，正是通过调用 ScheduleFromProc 完成了 CPU 的让出工作）。而 ScheduleFromProc 则不同，它着重处理的就是这些非 RUNNING 状态的核心线程，因为一旦核心线程把自己设置为 BLOCKED 等非 RUNNING 状态，就表示核心线程希望退出运行，退出的方式，就是通过调用 ScheduleFromProc 实现的。如果这个函数也“谦让”一下，继续让这些非 RUNNING 状态的线程继续执行，就会产生死循环。

ScheduleFromProc 函数的代码，也是以 switch-case 语句为骨架的，根据当前核心线程的线程状态分别处理。因此分析每个 case 语句的处理动作，是解释这个函数的最好方式。下面就分别解释：

(1) 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 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, 这样 `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, 以使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。

(2) `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, 但也可以看做是一个初始化核心线程。一旦初始化完成, 切换到其他的核心线程, 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与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, 以使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。

(3) 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。

(4) KERNEL_THREAD_STATUS_SLEEPING

当前核心线程调用 `Sleep` 函数，试图进入睡眠状态的时候，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。

(5) 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 `KERNEL_THREAD_STATUS_SUSPENDED` 的处理机制一样。

与 `ScheduleFromInt` 一样，`ScheduleFromProc` 也是通过调用 `GetScheduleKernelThread` 函数来获取一个最高优先级的就绪线程。在切换到目标线程的时候，也需要递增其运行时间片。与 `ScheduleFromInt` 不同的是，这个函数调用了 `__SaveAndSwitch` 函数实现了线程的切换。这个函数与 `__SwitchTo` 不同，除切换到目标核心线程外，还保存了当前核心线程的硬件上下文。在中断上下文的线程调度中，CPU 和汇编语言代码自动保存了线程的硬件上下文，因此对于当前线程的上下文无需保存。但是在 `ScheduleFromProc` 中，由于没有中断发生，必须手工建立起当前核心线程的硬件上下文，然后才能切换到目标线程。在 4.2.10 节将详细解释 `__SwitchTo` 和 `__SaveAndSwitch` 这两个底层的硬件上下文操作函数。

4.2.10 上下文保存和切换的底层函数

前面提到了两个完成线程切换和上下文保护的底层函数 `__SwitchTo` 和 `__SaveAndSwitch`。本节对这两个函数的实现进行描述。

首先回顾一下核心线程硬件上下文的定义：

```
[kernel/include/ktmgr.h]
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_CONTEXT)
    DWORD        dwEFlags;
    WORD         wCS;
    WORD         wReserved;
    DWORD        dwEIP;
    DWORD        dwEAX;
    DWORD        dwEBX;
    DWORD        dwECX;
    DWORD        dwEDX;
    DWORD        dwESI;
    DWORD        dwEDI;
    DWORD        dwEBP;
END_DEFINE_OBJECT()
```

上述定义中各变量的含义已经做了讲解。前面也讲到，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，会建立如图 4-11 所示的堆栈框架（为了方便查阅，我们把这个图再放到这里。这可能是整个线程切换部分中最重要的图示了）。

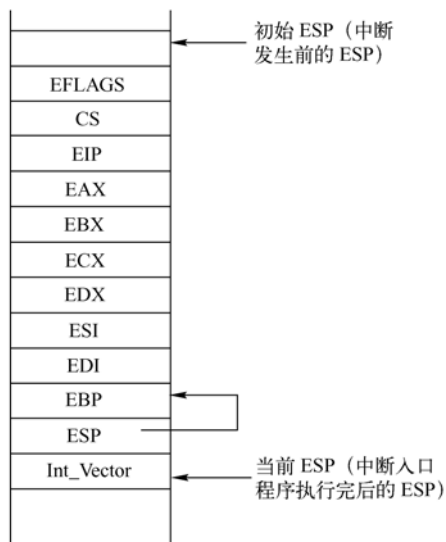


图 4-11 中断发生后的堆栈框架

建立上述框架后，调用 `GeneralIntHandler` 函数，该函数接受两个参数，一个是中断向量，另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，这个堆栈框架与核心线程上下文的定义（`__KERNEL_THREAD_CONTEXT`）是吻合的，因此，只要在 `GeneralIntHandler` 里把传递过来的 `ESP` 指针保存到核心线程对象的 `lpKernelThreadContext` 里就可以了。

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。`__SwitchTo` 函数就是这样实现的。下面是其实现代码。为了移植方便，`__SwitchTo` 函数放在了 `ARCH` 目录

下的源代码中，在移植 Hello China 到其他硬件平台的时候，只需要移植该目录下的相关代码即可，其他目录下的代码，都是与 CPU 无关的（当然，这只是理论情况，在移植的时候还是需要对每行代码都检查一遍的）。

```
[kernel/arch/arch_x86.cpp]
__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
    }
}
```

该函数使用 `__declspec(naked)` 进行修饰，这个修饰的含义是，不在函数的前面增加任何附加的汇编代码（即函数是 `naked` 的，“裸体”的）。在微软的编译器中，缺省情况下，编译器会在每个函数的开始处，插入一些汇编代码。这些汇编代码保存了 `EBP` 寄存器，然后把 `ESP` 寄存器的内容复制到 `EBP` 寄存器里面。这样使用 `EBP` 寄存器，即可访问函数的参数了。这个过程与上述代码中的前两条汇编指令所达到的效果是一样的，实际上据作者观察，大多数情况下，微软的编译器就是在函数的开始处，插入“`push ebp`”和“`mov ebp,esp`”两条指令。

但是线程的切换过程需要对 CPU 的每个动作都非常清楚，因此为了保险，作者还是让编译器不要插入辅助代码，而由作者自行插入，虽然这两者所达到的效果是一样的。在把 `ESP` 寄存器的值复制到 `EBP` 之后，就可通过 `EBP` 寄存器访问函数的参数列表了。上述代码中的第三条指令，就是把 `__SwitchTo` 函数的参数，实际上就是切换目标线程的硬件上下文，保存到了 `ESP` 指针处。我们知道，核心线程的硬件上下文，就是其被中断时的堆栈框架。这样一旦把硬件上下文恢复到 `ESP` 寄存器，本质上就是恢复了目标上下文的线程堆栈。这样再依次恢复目标线程的通用寄存器（线程被中断打断的时候，这些通用寄存器被中断处理程序的入口汇编代码保存，现在必须按照相反的顺序恢复它们），并执行 `iretd` 指令，即可切换到目标线程了。

需要注意的是，`__SwitchTo` 函数是在中断上下文中调用的，因此在调用 `iretd` 指令恢复线程执行前，必须解除 8259 中断控制器的中断请求。在操作系统初始化的时候，8259 中断

控制器是被初始化为应答工作方式的，即中断发生后，8259 在收到 CPU 的中断应答前，将不会发起任何其他中断。上述 `iretd` 前面的代码，就是对 8259 控制器做了应答。

但在系统调用上下文中切换的时候，却有些麻烦，因为这时候是没有中断发生的，图 4-11 中的堆栈框架无法建立。这时候必须手工建立上述堆栈框架，以对当前核心线程的执行上下文进行保存，然后恢复目标线程的上下文。这就是 `__SaveAndSwitch` 函数的实现了。下面是该函数的原型：

```
[kernel/arch/arch.h]
__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（通过 `CALL` 指令），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，如图 4-12 所示。

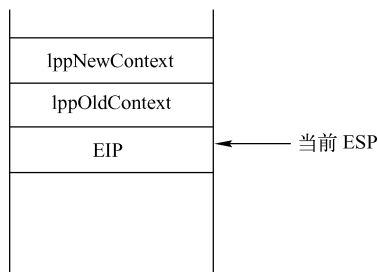


图 4-12 `__SaveAndSwitch` 调用后的堆栈框架

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。代码如下：

```
[kernel/arch/arch_x86.cpp]
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

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）。

该函数的一个难点在于，如何建立与中断发生时完全一样的堆栈框架。在中断发生的时候，CPU 自动把 CPU 的标志寄存器（EFLAGS）、CS 段寄存器和返回的指令压入堆栈中。但 `CALL` 指令执行后，堆栈中只保存了函数执行完毕后返回的指令，没有压入 CS 和 EFLAGS。而且中断发生的时候，CS 和 EFLAGS 是位于返回指令之前的。因此为了建立这样的堆栈框架，首先需要把返回的指令地址保存起来，然后依次压入 EFLAGS 和 CS，再压入刚才保存的返回指令地址。这样就建立了与中断发生时相通的堆栈框架。然后再保存当前核心线程的通用寄存器。之后，再把堆栈指针保存到当前核心线程的硬件上下文指针（`lpKernelThreadContext` 变量，由 `lppOldContext` 指向）中。

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。通过连续几条 POP 指令，进行新核心线程的上下文恢复，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。

在上述保存和恢复线程堆栈框架过程中，发现仅仅通过 CPU 提供的几个寄存器已经不能解决问题，于是定义了几个静态全局变量，用于数据的交换工作。

4.2.11 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，一段时间之后，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)，则该线程的实际睡眠时间应该为

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

其中， t/T 为 t 除以 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。

4.2.12 核心线程实现总结

至此，我们把 Hello China 线程的实现机制做了详细的解释。这是操作系统实现中最核心、最关键的内容，但是在大多数操作系统书籍上，却找不到这些内容，能够找到的都是一些线程调度算法等理论上的东西。即使有很多 Linux 相关的书籍，对线程的切换做了很细致的解释，但是由于其代码的庞大和复杂，除非对 Linux 有很长时间的研究，否则读者很难在短时间内真正明白线程的切换过程。Hello China 目前版本的代码还不是太庞大和复杂，因此希望通过 Hello China 的实现，向读者展示一个相对浅显但又真实的线程切换图景，让读者对线程和进程不再感到神秘。但这个过程仍然是比较复杂的。

第 5 章 内存管理机制

5.1 内存管理机制概述

当前版本 Hello China 的内存管理机制的实现分为两部分。

(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（本书中称为 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。

本章首先讨论 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，其他类型或厂家的 CPU 的内存管理机制与 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，就可以很容易地通过阅读特定 CPU 的技术资料，掌握其他类型 CPU 的内存管理机制。为了进行比较，本章对 Power PC 的内存管理机制也进行了简要的介绍。

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，每个任务都有自己特定的段，这样每个任务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（只加载应用程序的部分代码到内存中，即可以开始执行）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、与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，且应用程序之间也互不影响，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。

图 5-1 清楚地表示了这种内存管理机制。

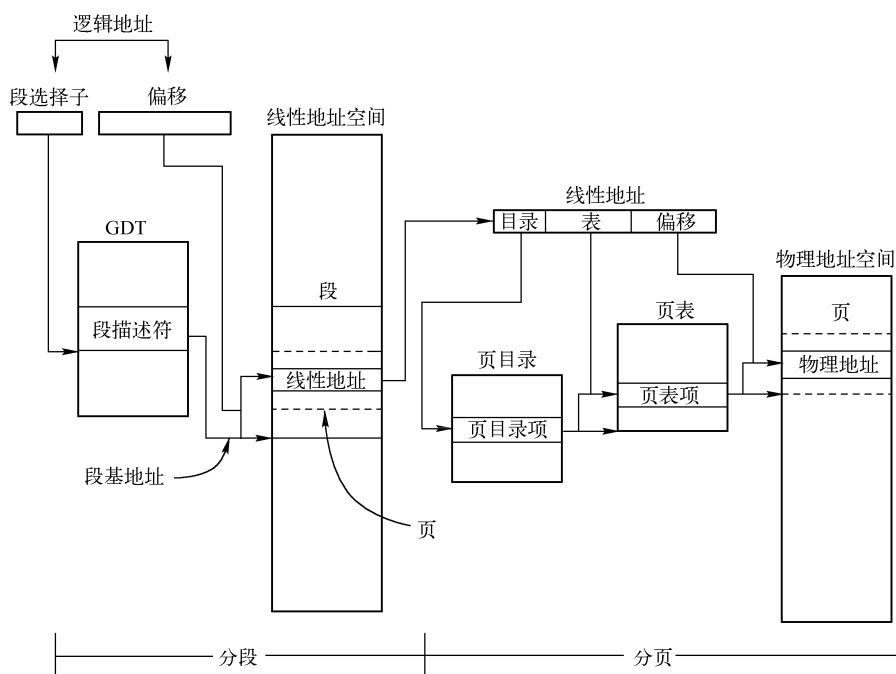


图 5-1 Intel CPU (IA32) 内存管理机制

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。要描述每个段，就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全局描述符表（GDT）中，全局描述符表的每一项（称为段描述符）描述了一个特定的段。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（用于确定段的基地址）和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 查找到全局描述符表对应的物理内存起始地址，然后再根据描述符在全局描述符表中的索引，定位到具体的段描述符的物理地址。段描述符在全局描述符表中的索引，称为段选择子。对于所有的段，由于 GDTR 是固定的，所以，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）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。图 5-2 示意了逻辑地址和线性地址的关系。

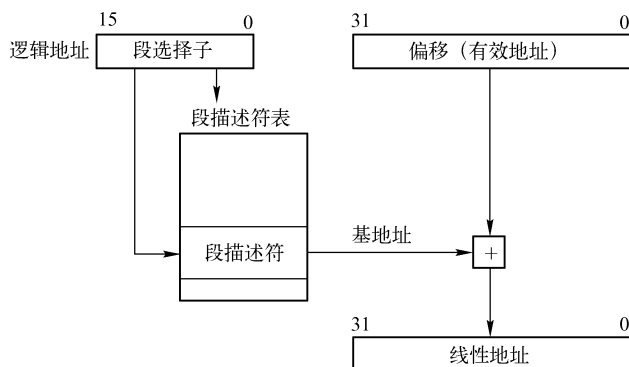


图 5-2 逻辑地址和线性地址的关系

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），然后把基地址与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，这样势必影响效率。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，访问具体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，这样在访问段内数据时，就不用再从物理内存中获取段描述符，然后再获得段基地址了，而是直接从影子寄存器内获得段基地址。在这样的机制下，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图 5-3 所示。

可见部分	隐藏部分	
段选择子	段基地址、段界限、访问属性等	CS 寄存器
段选择子	段基地址、段界限、访问属性等	SS 寄存器
段选择子	段基地址、段界限、访问属性等	DS 寄存器
段选择子	段基地址、段界限、访问属性等	ES 寄存器
段选择子	段基地址、段界限、访问属性等	FS 寄存器
段选择子	段基地址、段界限、访问属性等	GS 寄存器

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，因此在本节中再次强调一下：

- 逻辑地址：段选择子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的，比如，内存中的代码，其逻辑地址是由 CS 寄存器存储的代码段选择符和 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时，首先通过 CS 寄存器的影子部分获得段基地址，然后与 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：段选择子与段内偏移共同组成了逻辑地址，由段选择子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，存储了该段的基地址，线性地址就是段基地址与段偏移相加形成的地址。线性地址是 32 位的，线性地址与逻辑地址的关系并不是一对一的，一个逻辑地址对应唯一的一个线性地址，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，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，因此，在不启用分页机制的情况下，线性地址与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。

图 5-4 示意了上述各地址之间的关系。

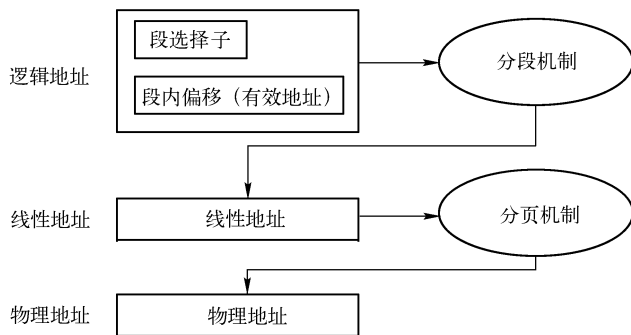


图 5-4 各地址概念之间的关系

还存在一种“虚拟地址”的概念，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，本书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。本节将对不同的段模式进行介绍。

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，并且每个段都与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。图 5-5 示意了这种基本的平展段模式。

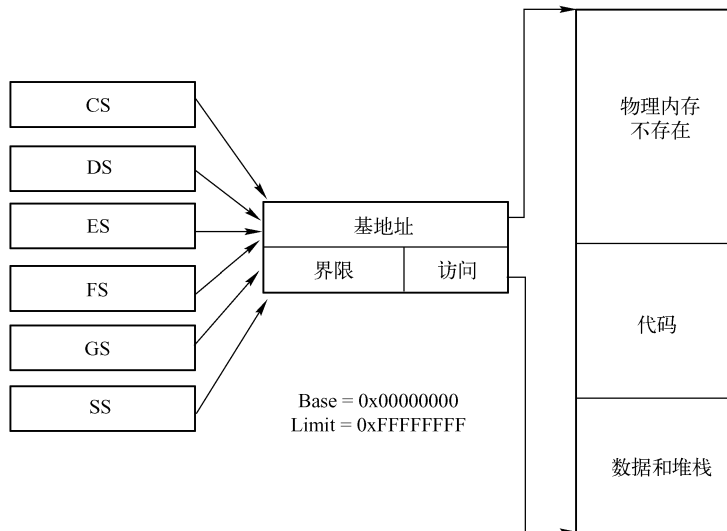


图 5-5 基本的平展段模式

这种模式下，至少要创建两个段描述符：一个为代码段描述符，该描述符的选择子存放到了 CS 寄存器；另外一个为数据段描述符，其选择子存放到了 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、基地址和界限都相同，唯一不同的是标志字段。

平展段模式是一种最基本的段模式，又是一种最通用的方式，按照这种方式实现的操作系统，可以很容易地移植到其他 CPU，甚至是一些没有实现段机制的 CPU。当前版本的 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。

所有段的基地址和界限，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。

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）时，也仍然是可行的，不会引起异常。而保护平展段模式则不同，这种模式下，根据需要把段的界限和基地址设置为合适的值，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）。图 5-6 示意了这种结构。

这种段模型也有两个段：

- 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，界限设置为数据和堆栈的长度的和。

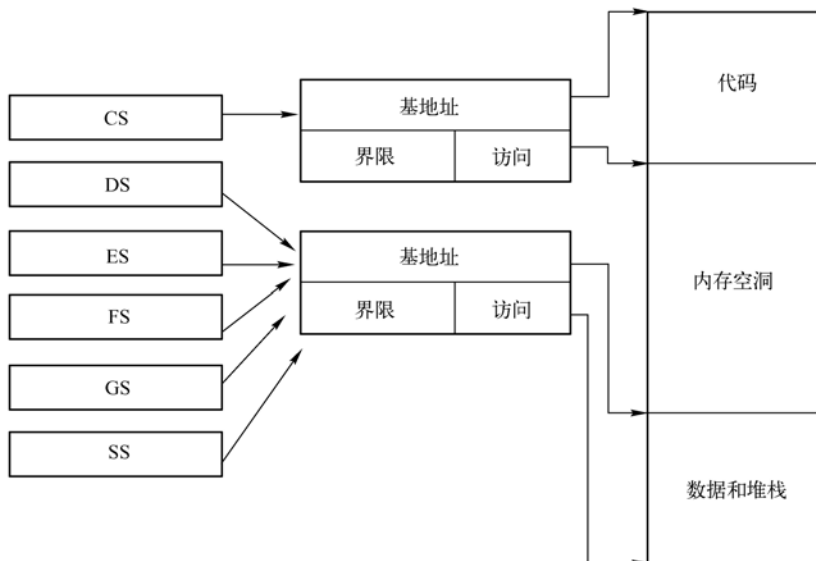


图 5-6 保护平展段应用模式

这样就可以避免以下两种错误：

- (1) 内存访问越界：若对数据或代码的访问超出了实际配置的物理内存，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。

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。这种模式完整地应用了 IA32 CPU 提供的段机制，这样可以很好地应用一些基于段机制的保护措施，如图 5-7 所示。

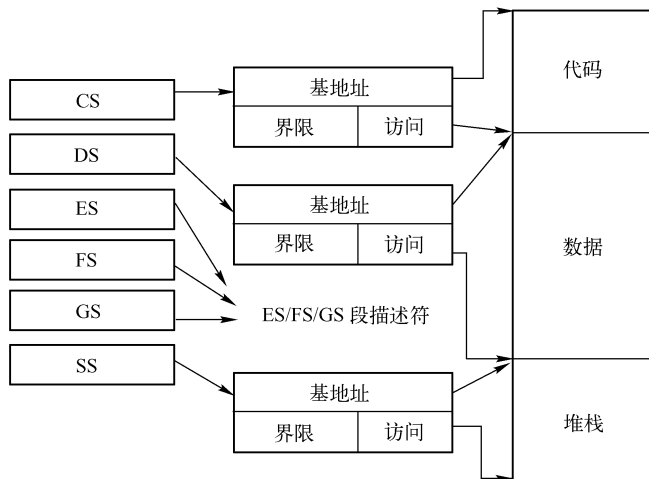


图 5-7 多段模式

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

基地址和界限，分别与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 提供的段保护机制，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，就会引发异常。但这种结构也有一个缺点：CPU 依赖性太强，按照这种模型实现的操作系统，很难移植到其他与 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 的最基本特征。只有基于分页机制，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 的操作系统）中，而在实时嵌入式操作系统中，一般很少使用。

IA32 CPU 实现了完善的分页机制，在 Hello China 中也实现了基于 IA32 CPU 的分页功能，提供了简单的内存保护、高速缓存控制等功能。不过，当前版本 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来取消分页功能。

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 的分页机制进行了描述，并列举了几个典型的应用。这些应用，都是现代通用操作系统中实现的最基本功能。

1. 分页机制概述

IA32 CPU 根据逻辑地址计算出线性地址，此时，若没有启用分页机制（通过 CR0 寄存器中的一个位判断），则直接把线性地址映射到物理地址，即直接把线性地址送到物理地址总线上完成一个内存访问动作。若启用了分页机制，CPU 就不会直接根据线性地址访问物理地址了，而是根据线性地址查找页目录和页表，最终获得物理地址。图 5-8 示意了根据线性地址查找物理地址的过程（算法）。

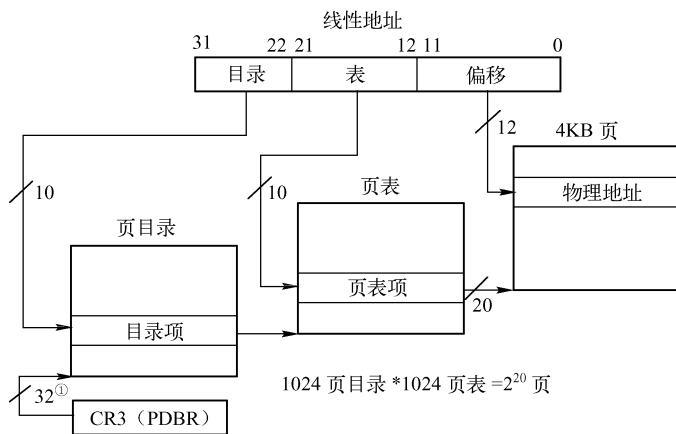


图 5-8 地址转换算法

① 32 位对齐至 4KB 页边界

(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（22~31 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（12~21 比特），形成页表内索引，根据这个索引查找步骤（2）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（0~11 比特）为偏移，以步骤（3）获取的物理地址为基地址，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和两次查表操作，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，查找时，首先从 TLB 中查找，若查找失败，再启动内存查找，并更新 TLB。

图 5-9 是 IA32 CPU 的页目录、页表和页框的结构。

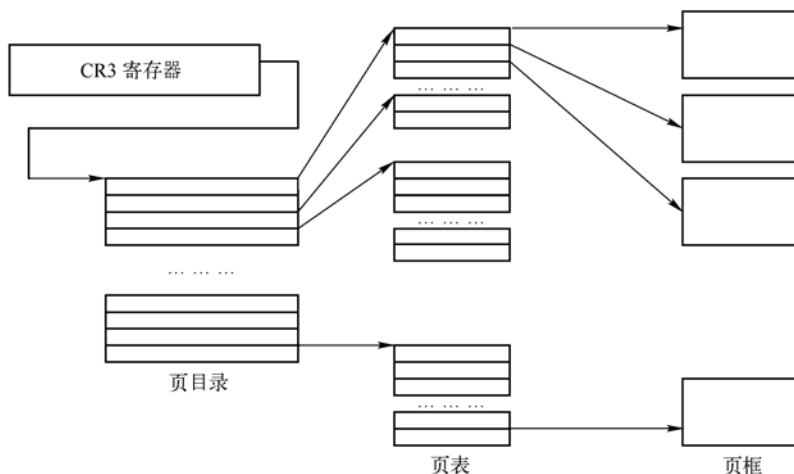


图 5-9 页目录、页表和页框的关系

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，如图 5-10 所示。

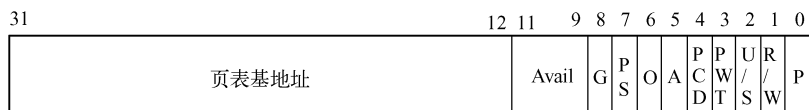


图 5-10 页目录项的组成

其中，页表基地址部分指出了与该页目录项对应的页表的基地址，其他比特的含义见表 5-1。详细的含义可参考 Intel CPU 的用户编程手册。

表 5-1 页目录项各比特的含义

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 4KB, 否则为 4MB
G	全局标志
Avail	供应用程序使用

线性地址的前 10 比特是页目录的索引, 因此最大可以确定 1024 个页目录项, 每个页目录项的大小是 4B, 整个页目录的大小是 4KB。

同样, 页表也是由页表项组成的, 页表项的结构如图 5-11 所示。

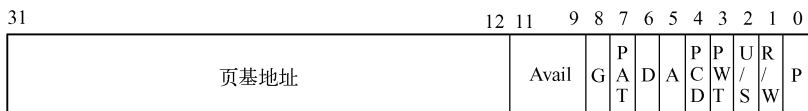


图 5-11 页表项的组成

其中, 页基地址给出了页框的物理地址 (基地址), 该字段是 20bit, 因此, 每个页框是 4KB 对齐的, 且其大小是 4KB, 其他标志的含义见表 5-2。

表 5-2 页表项各比特的含义

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 4KB, 否则为 4MB
G	全局标志
Avail	供应用程序使用

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 4KB (1024 个页表项, 每个页表项 4B)。32 比特线性地址空间采用这种页目录和页表结构完整表示, 需要内存的数量为

$$4KB (\text{页目录}) + 1024 \times 4KB = 4100KB$$

其中，第一个 4KB 是页目录所占用的空间， $1024 \times 4\text{KB}$ 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，表示其中的一部分就可以满足应用需要了，因此，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与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，可以很灵活地实现操作系统特性。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，还有一些其他的组合情况也会引发异常。表 5-3 列举了会引发异常的情况以及访问方式。

表 5-3 特权模式和访问模式之间的组合

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
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

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）。另外，按照 Intel 的软件编程手册中的描述，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。但在本书的描述中，这些特殊情况不会造成影响。

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，下面简单介绍几个比较典型的机制。

2. 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，导致整个系统故障。假设一个操作系统采用保护平展段模式使用 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），基地址为 0x00000000，界限为 4GB，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 3GB 线性地址空间供应用程序使用。这样，操作系统需要为这 1GB 空间建立页表（只为实际装入代码和数据的内存建立部分页表项即可），建立页表时，把页表的访问特权级设置为 Supervisor。每创建一个进程（加载一个应用程序），操作系统就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 3GB 空间中，并为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），如图 5-12 所示。

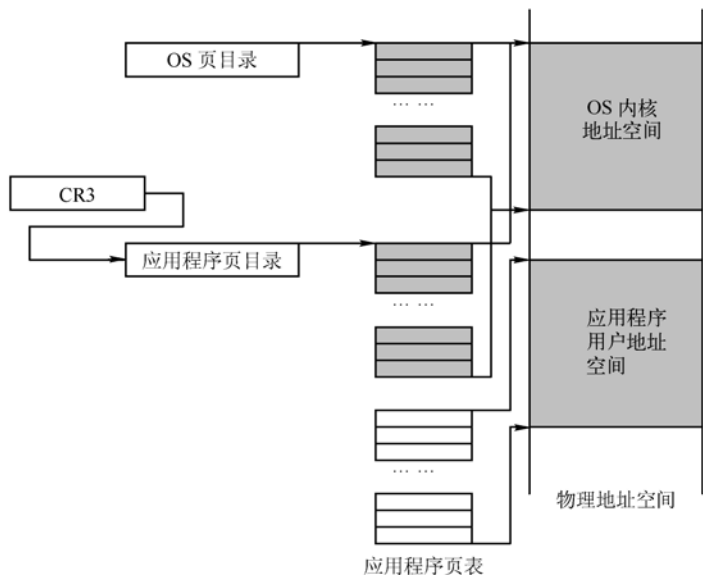


图 5-12 应用程序和操作系统地址空间

图中灰色的页表项是操作系统空间所对应的页表项。在创建应用程序的时候，把这部分页表项追加到应用程序的页表项当中。这样应用程序的页表项即包含操作系统的页表项，但应用程序用户空间的页表项的访问权限是用户级，而操作系统内核地址空间所对应的页表项的访问权限是系统级。

正常情况下，应用程序的运行只限制在其用户地址空间内，以特权级 3（用户级）来运行。由于操作系统页表访问特权级是 0（系统级），而目前的特权级是 3，因此，一旦应用程

序试图访问操作系统占用的内核空间，就会引发越权访问异常。

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），进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地址空间中的相同位置。进程切换时，只需把 CR3 寄存器（页目录基地址寄存器）切换为新的进程，就实现了进程地址空间的切换。目前许多流行的操作系统，比如 Linux、Windows 都是按照这种方式实现的，或者与此方式类似。

3. 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，物理内存大大增加了，使得计算机系统能够运行实际大小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，就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，从永久性存储介质中重新装入内存，并更改 P 标志为 1，然后从异常处理程序中返回。返回后，原先引起内存访问异常的指令会被再次执行。这时候由于对应的页框已经被切换回内存中，且 P 标志被修改成了 1，因此不会再次引发缺页异常。

图 5-13 简单地说明了这种情况。

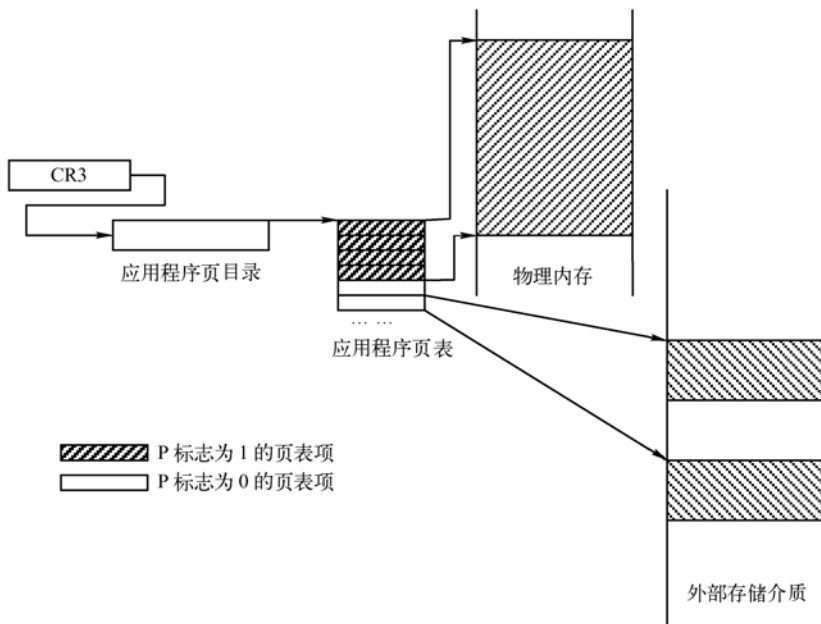


图 5-13 虚拟内存的实现机制

应用程序的页表项中有的 P 比特为 1, 该页表项与唯一的物理内存页框对应; 而有的 P 比特为 0, 表明与该页表项对应的物理内存页框不在内存中, 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。在 P 比特为 0 的情况下, 页表项的前 31 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以页长度为单位的偏移), 若应用程序 (代码或数据) 访问了该页框, 就会引发缺页异常。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回到这个物理页框, 然后更改对应的页表项 (若需要, 还需更改页目录项), 所有这些操作完成之后, 操作系统从异常处理程序中返回, 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, 将不会再次引起异常。

需要补充的是, 在页表项中, 若 P 标志为 0, 则 CPU 对页表项的其他 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 有时并不代表该页面被切换出了内存, 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)。可以将剩余的 31 比特全部设置为 0 来表示这种情况。为了区分这两种情况 (按需内存分配和虚拟内存), 规定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, 这样的后果就是页面文件的第一个页面大小的块将不被使用。

4. 按需内存分配

按需内存分配是分页机制的另外一个应用, 其作用是尽可能地把应用程序对内存的需求延迟到必须的时候才分配, 以尽可能地提高内存利用率, 尤其是应用程序申请内存的数量较大的时候。比如一个应用程序申请了 1MB 的内存, 一般情况下, 应用程序不可能一下子把 1MB 内存都使用完, 为了提高内存使用效率, 操作系统会给应用程序返回一个分配成功的信息, 但实际上只为应用程序分配了有限数量的内存 (比如几个页框), 而应用程序请求的内存所对应的页表项, 都已经创建, 除了已经分配的几个页框, 其他页表项的 P 比特都设置为 0。

这样若应用程序访问 P 标志为 0 的页表项会导致一个缺页异常。在缺页异常处理程序中, 操作系统会重新为应用程序分配内存, 并更新页表项 (需要的时候进一步更新页目录项)。这个过程与虚拟内存类似。为了区别这两种情况, 采用了一个特殊的标识方法——页表项除了 P 标志之外的 31 比特全部为零时表示页表项对应一个未分配页框, 否则表示页表项对应虚拟内存的情况。

5. 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一份副本, 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这一点)。

6. 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, 用来运行实际大小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时, 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, 其他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，其他的设置为 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，就会引发一个缺页异常，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，这样可使应用程序继续执行。

5.3 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。

图 5-14 示意了 PPC (Power PC) CPU 的内存管理机制。

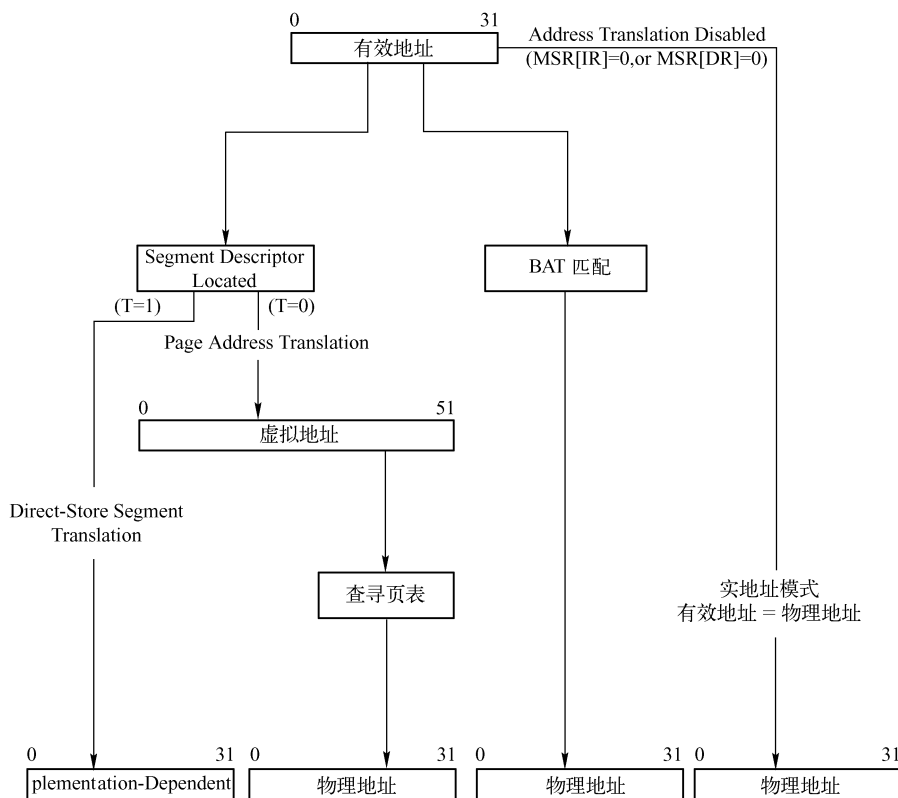


图 5-14 Power PC 的内存转换机制

在 PPC 的指令系统中，最初的地址叫做有效地址。在 32 位 CPU 中，有效地址是 32 位的。有效地址与 IA32 内存管理机制中的线性地址类似，在 PPC 中所有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。

PPC 的实现，可以把内存分成 4KB 大小的页面，以分页机制进行管理；也可以把物理内存分成长度可变化（但最小长度不能低于 128KB）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大小

为 256MB 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。进行内存寻址时，CPU 会对一个虚拟地址同时进行 BAT 匹配和段匹配 (以虚拟地址的高 4 比特来索引一个段描述符)，其中 BAT 优先级更高，即若虚拟地址匹配 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。

下面重点介绍一下 PPC 的段页管理机制。CPU 在寻址的过程中，根据虚拟地址的高 4 比特定位到一个段描述符之后，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程序，根据段描述符以及虚拟地址的剩余比特形成一个虚拟地址 (虚拟地址的长度是 52bit)，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 CPU 上的一种加快设备访问的机制，在新的 PPC CPU 中将会被淘汰，因此不必太关注。图 5-15 示意了 PPC CPU 根据虚拟地址获取物理地址的过程。

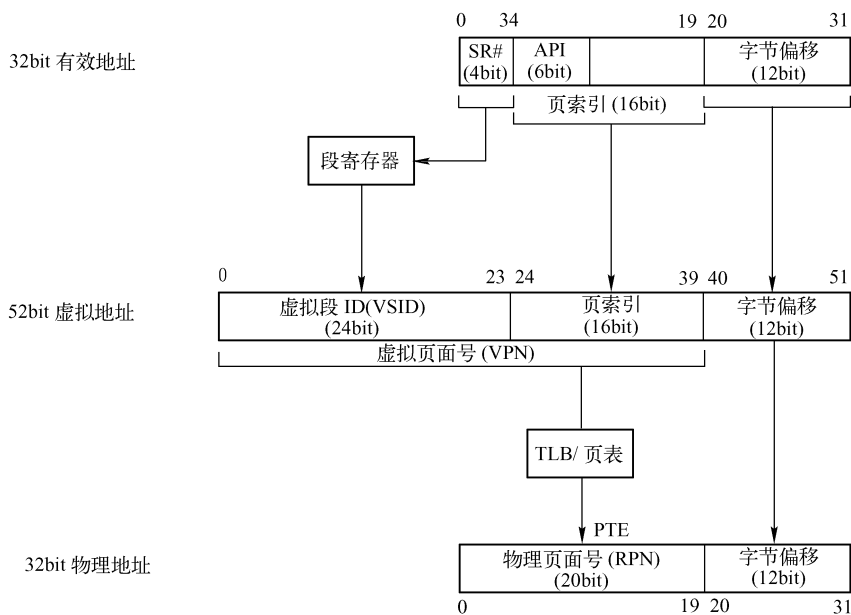


图 5-15 Power PC 的段页管理机制

在查找段表的过程中，会把 52bit 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。通过合理地设计页表组织结构和大小 (由操作系统完成)，可以大大提高命中率。

需要注意的是，与 IA32 CPU 一样，PPC CPU 也有一种实地址模式 (Real Address



Mode)。在这种模式下，PPC 的有效地址直接映射为物理地址，但与 IA32 的实地址模式不同的是，IA32 的实地址模式是 16 位模式。

5.4 Hello China 内存管理模型

5.4.1 Hello China 的内存管理模型

当前在 Intel 32 位 CPU 上实现的 Hello China 版本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，定义了 Hello China PC 版的实现中，全局描述表（GDT）的结构。全局描述表包含了代码段、数据段和堆栈段，需要注意的是，按照 Intel 的定义，GDT 中的第一个描述符必须为空，即全部填写为 0。

```
[kernel/arch/sysinit/miniker.asm]
gl_sysgdt:                ;;GDT 起始地址
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
                           db 0x00
                           dw 0xCF93
                           db 0x00
```

这样在初始化 GDT 寄存器（lgdtr 指令）的时候，只需要把 gl_sysgdt 标号装入 GDT 寄存器即可。结合 IA32 CPU 的段描述符的定义，可以看出，目前 Hello China 实现的各个段的属性见表 5-4。

表 5-4 Hello China 的段属性

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

这样的实现实际上是一种最简单、最通用的实现，相当于忽略了 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，这种模型符合 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問題。比如下列两个函数：

```
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引用 lpdwResult 时，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，导致执行结果不正确，严重的话还会引起系统崩溃。之所以产生这个问题，是因为一般的编译器在传递指针参数时只传递段偏移部分，而不传递段选择子。

在当前的实现中，Hello China 没有实现进程，只实现了线程，而且实现时，所用的线程和操作系统核心代码以及数据共享同一线性空间。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，就是保护功能稍微弱一些，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 的分页机制来完成内存保护功能。通过分页功能可以很容易地把线性地址空间内的内存地址映射到物理内存中，而且还可以实现按需内存分配功能，保证了内存资源的充分利用。

下面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 4KB 区（以 4KB 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，在介绍上面两个区域的管理方式后，将详细介绍 Hello China 的虚拟内存实现方法。在当前版本中，虚拟内存的实现是建立在 CPU 的分页机制上的。

5.4.2 Hello China 的内存布局

按照目前的实现，Hello China 启动完成后的内存布局如图 5-16 所示。

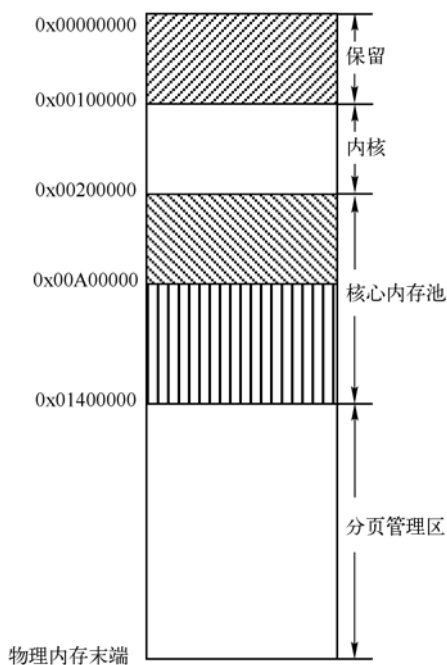


图 5-16 Hello China 的内存布局

对图 5-16 的内存布局，描述如表 5-5 所示。

表 5-5 Hello China 的内存布局

范 围	大小	用 途
0x00000000~0x000FFFFF	1MB	预留
0x00100000~0x001FFFFF	1MB	操作系统核心代码、静态数据等
0x00200000~0x009FFFFF	8MB	操作系统核心内存池，以 4KB 为单位进行分配
0x00A00000~0x013FFFFF	10MB	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

其中，核心内存池供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），都从核心内存池中分配内存。核心内存池又进一步分成两部分：一部分以 4KB（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；另一部分以任意大小（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（4KB）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。

当然，上述内存布局在实现的时候，是通过宏定义来定义各个区域的。可以修改宏定义，来调整各个区域的大小，或者修改宏定义，取消某个区域。

5.4.3 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）4KB 区域，以 4KB 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使用，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（在当前的实现中，最小的分配单位是 16B），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。开始时，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时，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（或等于）请求的内存。

（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 16B），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 16B），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，回收函数判断与该空闲块相邻的下一块是否可以同当前块合并（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 示意了任意尺寸内存池的逻辑结构。

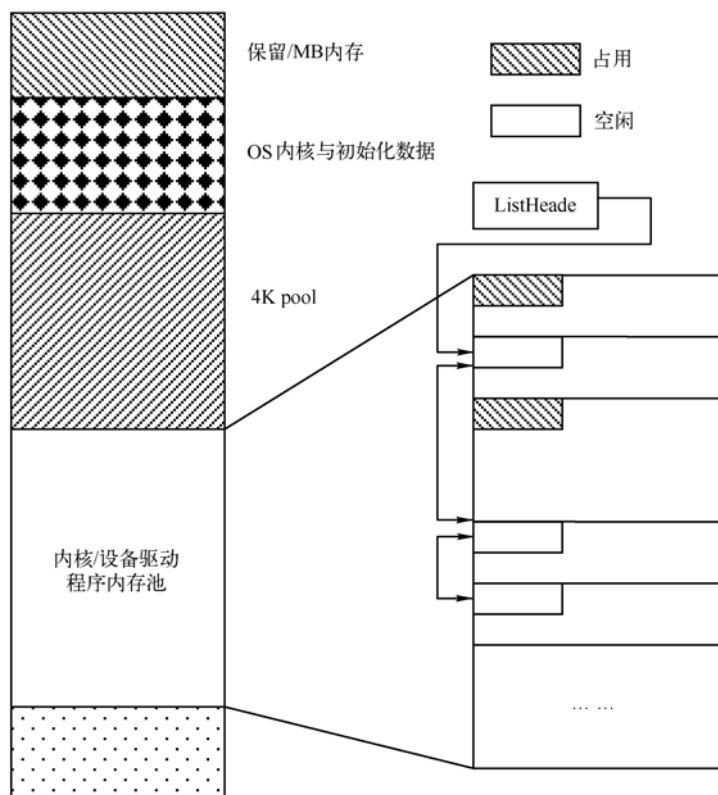


图 5-17 Hello China 的任意尺寸内存池

为了维护空闲块，必须为每块空闲块分配一个控制结构，然后由这个控制结构指定特定的空闲内存块。分配和回收时，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地索引到其控制块，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，如果此时用户请求一块大的内存，往往会以失败告终。但这些缺点仅仅是理论上的，实验表明，首次适应算法能很好地满足实际需求。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，运行效果也十分理想。

以任意尺寸（`KMEM_SIZE_TYPE_ANY`）为参数调用内存分配函数 `KmemAlloc`，是操作系统开发过程中使用最频繁的操作。

对于 4KB 区域, 采用位图算法进行管理, 即把整个 4KB 区域以 4KB 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 (4KB) 已经分配, 若为 0, 则说明该区域尚未分配。如图 5-18 所示。

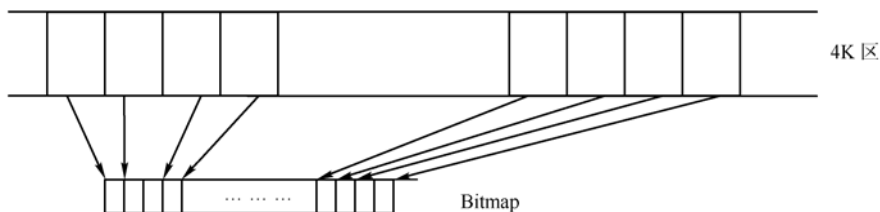


图 5-18 Hello China 的位图算法示意

位图实际上是一个静态定义的全局数组, 操作系统初始化时, 会对位图数据进行适当的初始化。内存分配时, 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, 并把首地址返回给申请程序, 否则返回 NULL。内存释放时, 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 参数指明了是从 4KB 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 4KB 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)。4KB 区域内的内存块释放时, 需要指定尺寸, 即最后一个参数 dwSize。

根据作者的经验, 不论是在操作系统核心的开发中, 还是在应用程序的开发中, 内存分配函数 (KmemAlloc 和 KmemFree) 都是使用最频繁的函数。因此一个好的内存分配算法非常重要, 会大大提升程序的整体效率。

5.4.4 页框管理对象

页框管理对象 (PageFrameManager) 用于对物理内存的分页区域 (0x01400000 到物理内存的末端) 进行管理。为了管理方便, 对物理内存进行分页管理, 每页的大小为 PAGE_FRAME_SIZE, 可以根据不同的 CPU 类型确定定义为 4KB 或 8KB 等。为了管理每个页框, 使用一个页框对象来描述, 代码如下定义:

```
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()

```

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 4KB），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，数组中的每个元素，对应一个实际页框，整体结构如图 5-19 所示。

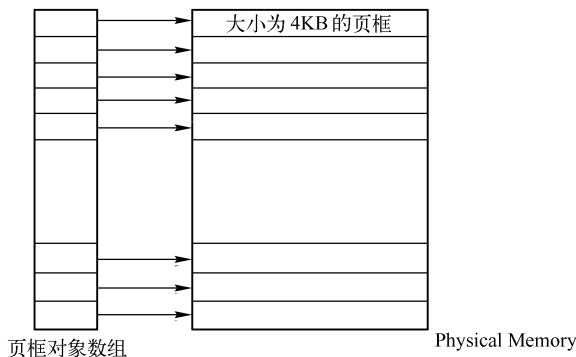


图 5-19 Hello China 的页框管理

需要注意的是，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，所以无法事先确定页框数组的大小。操作系统初始化时，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之间的一一对应关系。因此，给定一个页框对象的索引就可以唯一地确定一块物理内存（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：

$$lpPageFrameAddr = lpStartAddr + PAGE\_FRAME\_SIZE \times N;$$

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：

$$dwIndex = (lpPageFrameAddr - lpStartAddr) / PAGE\_FRAME\_SIZE;$$

其中，`lpStartAddr` 为物理内存的起始地址（物理地址）。

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，为了更有效地利用内存，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。伙伴算法的核心思想就是，通过尽量合并小的块来形成大的块，避免内存浪费。有关伙

伴算法的具体流程，请参考数据结构书籍。

在伙伴算法中，把数量不同的连续的物理页框组合成块，进行分配时，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大小的内存块：

- 4KB
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1024KB
- 2048KB
- 4096KB
- 8192KB

可以看出，申请的内存块的尺寸是 4KB 乘以 2 的整数次幂。

在系统核心数据区维护了下面一个数据结构（参考图 5-20）。

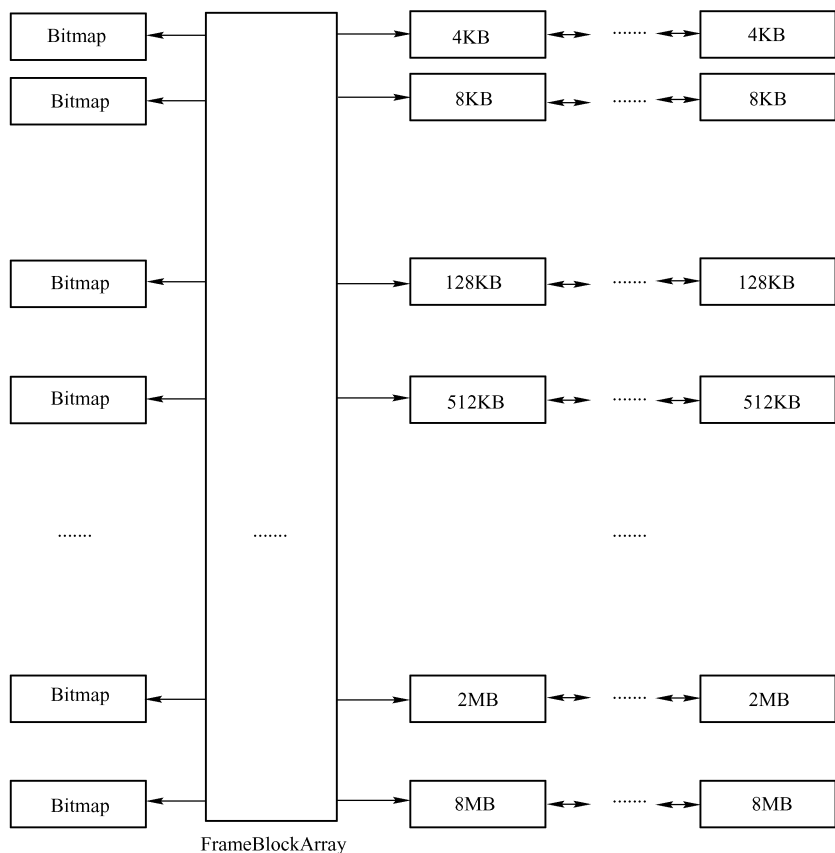


图 5-20 Hello China 的物理页框管理结构



有一个单独的管理对象——页框管理器（PageFrameManager）管理所有的页框和页块，当一个线程请求一块页块时，页框管理器进行下列动作：

（1）判断申请的块是否超出了最大范围（MAX_CHUNK_SIZE，当前版本定义为 8MB），如果是，则返回一个空指针。

（2）如果没有超出，则从第一个页块控制数组（FrameBlockArray）开始搜索，判断哪个尺寸的页块适合请求者的要求，比如请求者请求了 500KB 的页块，那么 PageFrameManager 会认为 512KB 的页块符合要求。

（3）判断 512KB 页块的空闲列表是否为空，如果不为空，则从中删除一块，返回请求者对应页块的指针。

（4）如果为空，则依次往下（页块尺寸大的方向）搜索，直到找到一块更大的页块，或者失败（搜索到最后一级，仍然没有空闲块）为止。

（5）如果失败，返回用户一个空指针。

（6）否则，依次对分找到的更大的页块，并插到上一级页块空闲列表中，直到找到 512KB 页块为止，然后返回用户经过对分最后剩下的页块的指针。

每个页块空闲列表控制结构都对应一个位图，系统通过这个位图来判断对应块空闲或被使用，这样释放页块时，可以根据位图把连续的页块（伙伴块）组装成更大的页块。

下面是页框管理对象的定义代码：

```
BEGIN_DEFINE_OBJECT(__PAGE_FRAME_MANAGER)
    __PAGE_FRAME*      lpPageFrameArray;
    __PAGE_FRAME_BLOCK  FrameBlockArray[PAGE_FRAME_BLOCK_NUM];
    DWORD               dwTotalFrameNum;
    DWORD               dwFreeFrameNum;
    LPVOID               lpStartAddress;
    BOOL                (*Initialize)(__COMMON_OBJECT* lpThis,
                                     LPVOID lpStartAddr,
                                     LPVOID lpEndAddr);
    LPVOID              (*FrameAlloc)(__COMMON_OBJECT* lpThis,
                                     DWORD dwSize,
                                     DWORD dwFrameFlag);
    VOID                (*FrameFree)(__COMMON_OBJECT* lpThis,
                                     LPVOID lpStartAddr,
                                     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 4KB、8KB 等总共 12 种，所以该数组的大小（PAGE_FRAME_BLOCK_NUM 的大小）为 12。lpPageFrameArray 是一个页框对象类型的数组，该数组由操作系统启动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（除去 OS 核心占用的内存）被分割成以 PAGE_FRAME_SIZE 为大小的页框块，每块物理内存对应一个页框对象（__PAGE_FRAME），假设系统中物理内存的大小为 32MB，其中 OS 核心占用了 20MB，系统中剩下的 12MB 内存以 4KB 为单位分割成 12MB/4KB = 3K 块，因此，lpPageFrameArray

数组的大小就是 $3\text{KB} \times \text{sizeof}(\_\text{PAGE\_FRAME})$ 。

`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` 参数和 `lpEndAddr` 参数是系统中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）。在目前 `Hello China` 的实现中，需要分页管理的物理内存的起始地址定义为 `0x01400000`（20MB 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 64MB 的物理内存，则操作系统初始化时，这样调用该函数：

```
PageFrameManager.Initialize(&PageFrameManager,0x01400000,0x03FFFFFF);
```

`FrameAlloc` 和 `FrameFree` 两个函数分别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（`FrameAlloc` 和 `FrameFree`），一般情况下应用程序不要直接调用页框管理对象提供的这两个函数，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来完成物理内存的分配。但这两个函数只能完成以 4KB 大小为粒度的物理内存的分配，无法满足更小粒度的物理内存的分配。`Hello China` 目前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，下面是一种可行的思路：

（1）应用程序初始化时，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 32KB）。

（2）把申请的上述 32KB 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 4KB 的内存时，就调用应用程序开发者自己编写的 `malloc` 函数进行分配。

（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。表 5-6 对物理内存管理机制的要点进行了总结。



表 5-6 Hello China 的内存访问服务接口

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	4KB 池	位图算法	4KB 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	4KB 倍数	FrameAlloc	FrameFree

5.4.5 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，而对于像 PowerPC 等 RISC 结构的 CPU，则采用散列算法，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）。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，该对象定义代码如下：

```
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);
    __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。

1. 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 20MB 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），把页目录固定地放置在物理内存 PD_START 开始的内存处（PD_START 定义为 0x00200000 -

0x00010000), 占用 4KB 的物理地址空间, 然后把 20MB 物理地址空间所占用的页表 (共 5 个页表框, 20KB 内存) 项紧接着页目录存放, 并进行填充。如图 5-21 所示。

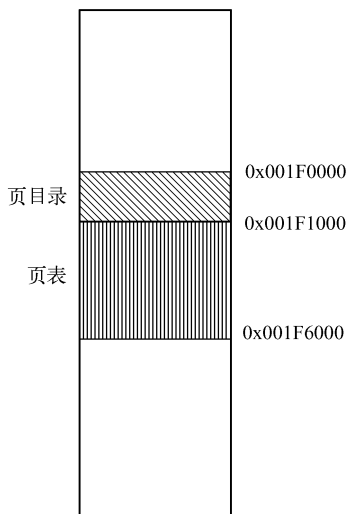


图 5-21 Hello China 的核心页目录和页表在内存中的位置

因为前 20MB 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, 对于页目录中没有占用的目录项, 都使用 `EMPTY_PDE_ENTRY` 填充, 即都填充成空。前 20MB 物理内存与线性地址空间的映射关系采用的是“照实映射”, 即映射到线性地址空间内的相同的位置 (线性地址空间的前 20MB 处)。这样做的好处是分页机制可以透明地显示给操作系统核心代码, 在编写、编译操作系统核心代码时, 无需考虑分页机制。

(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 (这时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)。

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 (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 20MB 内存空间映射到新建进程的虚拟地址空间中, 以使得这前 20MB 地址空间, 可以被任何进程访问)。

这样一个问题就出现了, 即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时调用。这个问题可以通过下列办法解决:

在第一次调用时, 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不是一个合



法的目录项，或者是一个空目录项（系统初始化时填充成空目录项），如果是空，则是第一次调用，直接初始化即可，否则，需要调用 `KMemAlloc` 函数分配页索引对象空间。

2. 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，只要对比该地址是不是 `PD_START` 就可以判断是不是针对进程调用。

3. 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，通过适当地分割虚拟地址，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；最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。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。

4. 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。该函数的任务就是通过在页索引对象中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（4KB），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 4KB 边界对齐的，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 4KB 边界对齐的要求）。

5. 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 4KB 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 4KB 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`。

6. 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。

7. 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来完成线性地址空间内的页面与物理地址空间内的页面之间的映射。

5.4.6 虚拟内存管理对象

1. 虚拟区域

可以采用虚拟区域（Virtual Area）来表示线性内存空间中的一个区域。需要注意的是，这个区域仅仅是线性内存空间的一部分，不一定与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，由于没有与物理内存对应，所以会导致访问异常。因此，在使用虚拟内存区域前，一定要通过 API 调用来完成虚拟内存到物理内存的映射。

但虚拟内存区域不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。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。

Hello China 目前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
DWORD
__VIRTUAL_AREA_DESCRIPTOR*
__VIRTUAL_AREA_DESCRIPTOR*
UCHAR
__FILE*
DWORD
__FILE_OPERATIONS*
END_DEFINE_OBJECT()

```

Reference;
dwTreeHeight;
lpLeft;
lpRight;
strName[MAX_VA_NAME_LEN];
lpMappedFile;
dwOffset;
lpOperations;

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)，形成图 5-22 所示的结构。

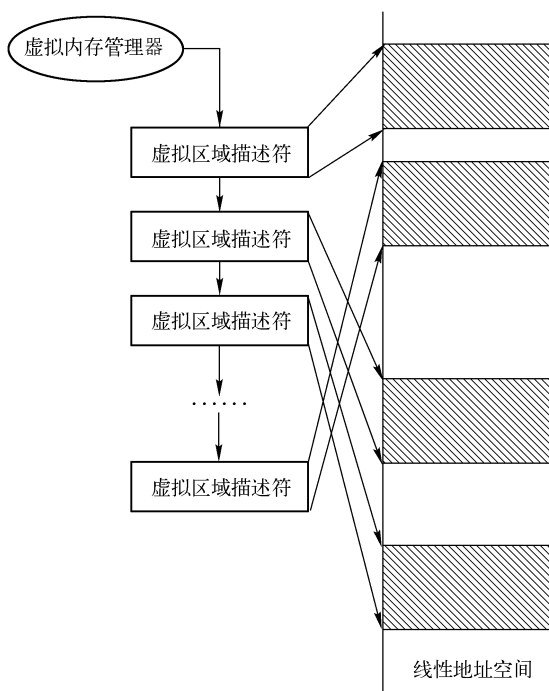


图 5-22 Hello China 的虚拟区域管理结构

之所以对虚拟区域（线性地址空间）进行统一管理，是因为线性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，都需要一块线性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，就可能会导致冲突，即两个实体占用了同一块线性地址空间区域。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，用以申请线性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性地址空间的分配情况（通过遍历虚拟区域描述符链表来实现），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（通过页索引对象来实现）。在 32 位线性地址空间中，整个线性地址空间的大小为 4GB，这是一个庞大的空间，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事情，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。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都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，所以只有一个全局的虚拟内存管理器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，进而共享整个虚拟内存空间。

lpStartAddr 和 lpEndAddr 分别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 则分别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时指定（一般由调用者指定）。

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与物理内存进行关联，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_CACHE_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 策略是禁用系统 CACHE，并禁用随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在不断变化，若采用 cache 缓存的读策略，则可能出现数据不一致的情况，所以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。

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，所以，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，而且这些存储空间还不能采用默认的内存缓冲策略。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剪的，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 的实现中也不作考虑。

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，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（将来的版本中会出现）。

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的（参考 VirtualAlloc 的定义）。

2. 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，提供应用程序（或设备驱动程序）可以直接调用的接口完成虚拟内存（线性地址空间）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。当前版本没有实现进程模型，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（Virtual Memory Manager）对象。

下面就是虚拟内存管理器对象的定义代码：

```
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）。在系统核心中，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，整个系统只有一个页面管理对象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，不考虑多处理器、多内存池的情况），而一个线性地址空间，对应一个虚拟内存管理对象和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），所以，整个系统中，这三种对象都只有一个。将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，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图 5-23 所示。

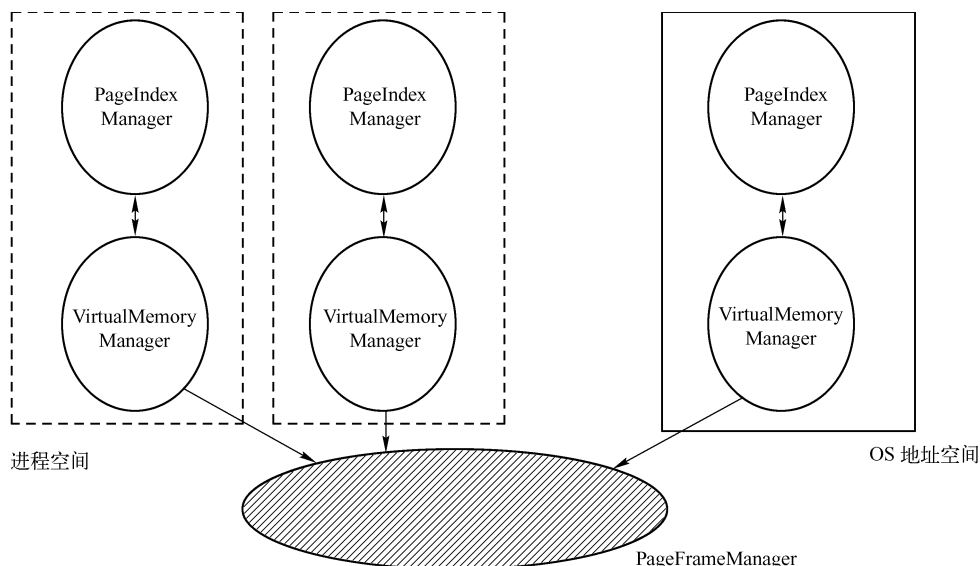


图 5-23 页框管理器、虚拟内存管理器和页索引管理器

由于当前没有实现多线程模型，所以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，虚线矩形表示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, 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, 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。

3. 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 函数实现), 起始地址为 0, 终止地址为 0x013FFFFFFF, 长度为 20MB (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。

操作系统初始化时, 调用该函数 (Initialize), 如果该函数失败 (返回 FALSE), 将直接导致操作系统初始化不成功。

4. 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。

5. 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 是应用程序的希望地址, 即应用程序希望能够得到从



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性内存空间，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，此时，对这一块虚拟内存的访问将会引起异常。
- **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，此时访问对应的虚拟内存就不会引起异常了。
- **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，预留的线性地址空间的地址与采用页索引结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）。也就是说，**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，用户调用时，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。

(1) **VIRTUAL_AREA_ALLOCATE_IO**

设置了这个标志，说明分配者希望分配到一块内存映射 IO 区域，访问 IO 设备。一般情况下，用户指定了 **lpDesiredAddr** 参数是希望系统能够在 **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，其访问对象根据 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时设置的数值，也可能是由 VirtualAlloc 重新分配的数值），以指示调用成功。

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 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(VIRTUAL_AREA_ALLOCATE_IO != dwAllocFlags)    //Invalidate flags.
        return NULL;
    lpIndexMgr = lpMemMgr->lpPageIndexMgr;
    if(NULL == lpIndexMgr)    //Validate.
        return NULL;

```

上述代码完成了局部变量的定义、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));
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：第一种情况是，能够找到一个满足用户需求大小的虚拟区域，但其起始地址与用户提供的期望的起始地址不一致，此时 `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；第二种情况是，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；第三种情况是，未能找到满足用户要求的结果（即系统线性空间中沒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.
```

与代码中的注释指明的那样，上述代码完成了页表的预留，然后使用与线性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，每次循环都预留一个页表项，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*)&PageFrame Manager,
            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，上述凡涉及共享变量的操作，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。

(2) VIRTUAL_AREA_ALLOCATE_RESERVE

当调用者使用该标志调用 VirtualAlloc 时，说明调用者仅想预留一部分虚拟地址空间以备将来使用。

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 重新寻找一块满足上述长度的连续区域，如果能够找到，则创建一个虚拟区域描述对象，并初始化，如果没找到，则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，递增 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，不再赘述。不同的是，VIRTUAL_AREA_ALLOCATE_IO 预留了页表项，而当以该标志调用 VirtualAlloc 时，却没有预留页表项，仅返回预留的虚拟区域的首地址。这时若引用这个地址，会引发内存访问异常。

(3) VIRTUAL_AREA_ALLOCATE_COMMIT

当使用该参数调用 VirtualAlloc 时，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。这时 lpDesiredAddr 绝不能为 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，则说明该地址没有被预留，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，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，这样由于调用一次 FrameAlloc 函



数，最多可以分配的物理内存是 8MB（目前的实现），所以，若采用本标志调用 VirtualAlloc 函数，目标虚拟区域的大小不能大于 8MB，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，可实现一种称为“按需分配”的内存分配策略，即开始时为调用者分配部分内存，比如一个物理内存页，此时，如果用户访问没有分配物理内存的虚拟内存地址空间，就会引发一个访问异常，系统的页面异常处理程序会被调用。异常处理程序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。这样就不会因频繁的内存访问异常导致系统效率大大降低。

(4) VIRTUAL_AREA_ALLOCATE_ALL

当设定 VIRTUAL_AREA_ALLOCATE_ALL 标志调用 VirtualAlloc 时，其效果与用 VIRTUAL_AREA_ALLOCATE_RESERVE 和 VIRTUAL_AREA_ALLOCATE_COMMIT 联合调用效果相同。

相关操作流程如下：

1) 向下舍入 lpDesiredAddr 到 FRAME_PAGE_SIZE 边界，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，创建并初始化虚拟区域描述对象。

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，根据 dwVirtualAreaNum 决定具体插入哪个数据结构。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 变量和 lpPhysicalAddr 变量（因为 ReservePage 每次填充一个页面），并递减 dwSize，直到 dwSize 为 0 为止。如果分配的物理内存的尺寸小于 dwSize（只有 FRAME_PAGE_SIZE）大小，则只在第一次调用 ReservePage 时，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，以指示内存尚未分配，这个循环，也是直到 dwSize 递减到 0（FRAME_PAGE_SIZE 为递减单位为止）。

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），这时直接给出严重警告并停机。

下面给出上述功能的实现代码。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(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 = (__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.
{
    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`。

```
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，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，则选择另外一个大小满足要求但不是用户期望地址的虚拟区域，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。

```
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之间的映射，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*)&PageFrame Manager,  
        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`。

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，然后在异常处理程序中继续为没有分配到物理内存的虚拟内存分配物理页面。

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。用户应用程序可以尝试改变请求的大小，再次调用 `VirtualAlloc` 函数。

6. 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，则说明该区域不存在，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，然后直接返回。

(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 为止) `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。

7. GetPdeAddress

该函数返回页目录的物理地址, 该地址用于设置 CPU 的特定寄存器, 比如针对 Intel 的 IA32 构架 CPU, 需要使用该地址设置 CR3 寄存器, 这时候就需要知道页目录的物理地址。

另外, 该函数直接读取 `PageIndexMgr` 的页目录物理地址, 并返回给调用者。

5.5 线程本地堆

5.5.1 线程本地堆概述

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` 函数时, 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与物理内存对应起来, 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称作“堆”(heap)。下面就对堆的实现进行详细描述。

5.5.2 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, 即软件工程中的所谓“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), 就不再改变。这样在该系统的实现过程中, 可避免频繁修改功能需求, 导致大量反复工作。在堆的实现中, 本书充分考虑用户程序的方便, 并向标准的 C 运行库靠拢, 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, 用堆的“内存释放”和“内存分配”函数实现 `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, 完全可以共用一个堆, 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 (如 16KB), 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, 堆对象会对内存池进行清理, 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, 而不能从其他应用程序 (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。另外在中断处理程序中, 也不能调用堆功能函数从堆中分配内存, 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 (把其他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转代理，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 4KB 大小），直接调用 `VirtualAlloc` 函数申请，因为通过堆管理器申请，堆管理器也可能调用 `VirtualAlloc`，这样经过了堆的中转，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。

5.5.3 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：

```
[kernel/include/heap.h]
BEGIN_DEFINE_OBJECT(__HEAP_MANAGER)
    __HEAP_OBJECT*      (*CreateHeap)(DWORD dwInitSize);
    VOID                 (*DestroyHeap)(__HEAP_OBJECT*);
    VOID                 (*DestroyAllHeap)();
    LPVOID               (*HeapAlloc)(__HEAP_OBJECT*,DWORD);
    VOID                 (*HeapFree)(LPVOID,__HEAP_OBJECT*);
END_DEFINE_OBJECT()
```

其中，`__HEAP_OBJECT` 是一个堆对象，该对象定义如下：

```
[kernel/include/heap.h]
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，它们除了接受一个 `DWORD` 类型的参数，用于指明需要申请的内存数量，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，作为一个统一的空间空闲内存池，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重新插入空闲链表。在操作系统核心内存池的管理中，也使用了空闲链表算法对任意尺寸内存池进行管理。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，而且这块区域已经与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向链表，每个链表元素是一个 `__VIRTUAL_AREA_NODE` 结构，该结构定义如下：

```
[kernel/include/heap.h]
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（单向），如图 5-24 所示。

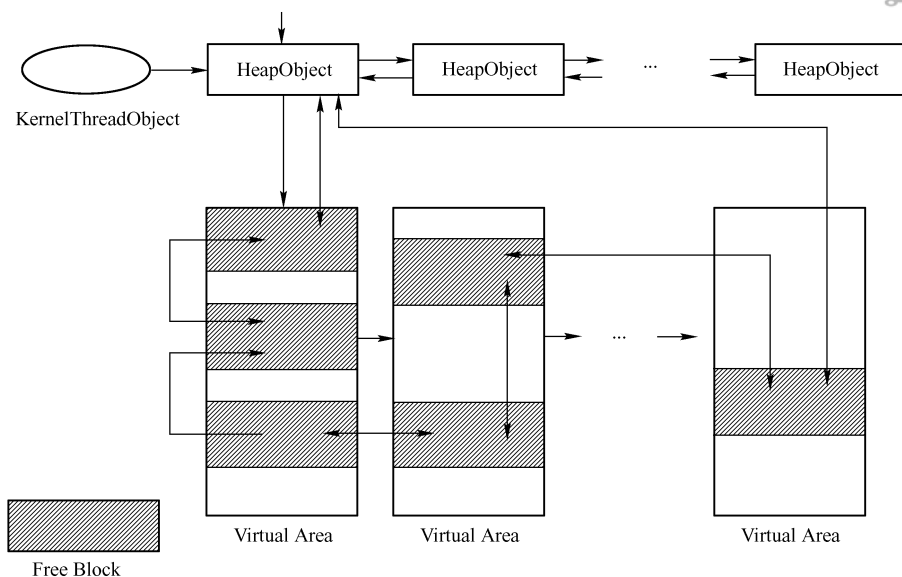


图 5-24 线程本地堆的整体数据结构

需要注意的是，空闲链表中的空闲块，与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。

空闲链表算法比较简单，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，该算法做如下操作：

(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 16B）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）来实现。

(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），如图 5-25 所示。

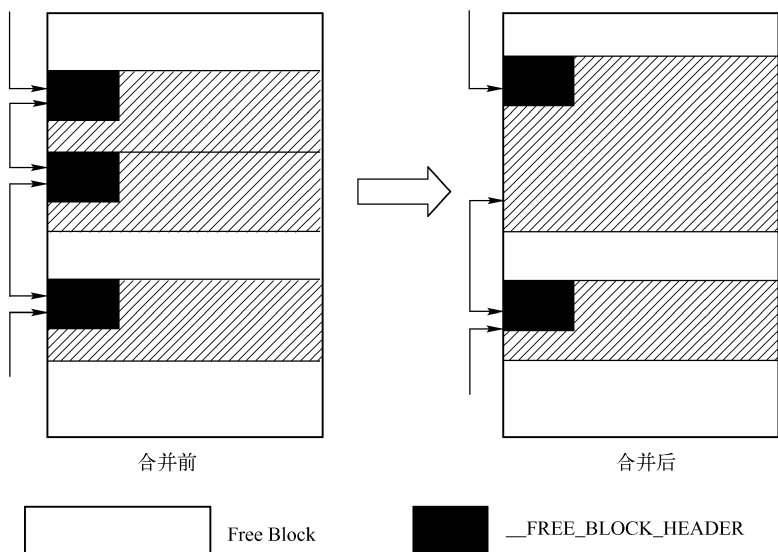


图 5-25 空闲内存块的合并操作

(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)。

将对空闲块的合并操作限制在一个虚拟区域内部, 可以提高效率, 因为无需遍历整个空闲链表。

从上述描述中可以看出, 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”很多, 则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, 然后在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:

```
[kernel/include/heap.h]
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 16B, 作为控制头, 如图 5-26 所示。

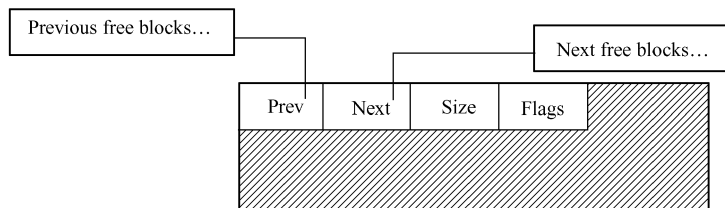


图 5-26 空闲内存块的控制头结构及位置

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 16B, 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。

5.5.4 堆的详细实现

下面对当前版本 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）。

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：

```
[kernel/kernel/heap.cpp]
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;

    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);
```

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（16B），因为要考虑空闲块控制头所占用的大小。

```

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.
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;
}

```

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，不存在与其他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。这种事务式的处理方式，在 Hello China 的实现中很常见。

```

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;
}

```

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。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占用的内存释放，即虚拟区域节点占用内存。
- （4）最后，释放堆对象本身。

代码比较简单，摘录如下（为了看起来方便，删除了一些无关紧要的代码，比如注释、参数安全检查等）：

```

[kernel/kernel/heap.cpp]
static VOID DestroyHeap(__HEAP_OBJECT* lpHeapObject)
{
    __VIRTUAL_AREA_NODE*    lpVirtualArea  = NULL;
    __VIRTUAL_AREA_NODE*    lpVirtualTmp   = NULL;
    LPVOID                  lpVirtualAddr  = NULL;
    DWORD                   dwFlags        = 0L;

    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);
}
RELEASE_KERNEL_MEMORY((LPVOID)lpHeapObject);
return;
}
```

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，以寻找一块大于用户请求大小的空闲内存块。

(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。

实现代码如下：

[kernel/kernel/heap.cpp]



```
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(lpHeapObject->lpKernelThread != CURRENT_KERNEL_THREAD)
    {
        return lpResult;
    }

    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 16B），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。之所以在 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。

```
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));
            lpTmpHeader->dwFlags      = BLOCK_FLAGS_FREE;
            lpTmpHeader->dwBlockSize = lpFreeBlock->dwBlockSize - dwSize
                - sizeof(__FREE_BLOCK_HEADER);
            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;
            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.
    {
        lpFreeBlock->lpNext->lpPrev = lpFreeBlock->lpPrev;
        lpFreeBlock->lpPrev->lpNext = lpFreeBlock->lpNext;
        lpFreeBlock->dwFlags        |= BLOCK_FLAGS_USED;
        lpFreeBlock->dwFlags        &= ~BLOCK_FLAGS_FREE;
        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，就可以很清楚地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 16B（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）。

如果无法从空闲链表中找到满足的空闲块，则需要扩充堆的内存池了，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堆的虚拟区域链表。代码如下：

```

    lpVirtualArea = (__VIRTUAL_AREA_NODE*)GET_KERNEL_MEMORY(sizeof(__VIRTUAL_
AREA_NODE));
    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;
}

lpVirtualArea->lpNext      = lpHeapObject->lpVirtualArea;
lpHeapObject->lpVirtualArea = lpVirtualArea;
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，然后把计算结果返回给用户。

```

lpResult = HeapAlloc(lpHeapObject,dwSize);
__TERMINAL:
return lpResult;
}

```

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不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，否则函数就直接返回。

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，则可能出现当前线程申请了大量的虚拟区域却闲置不用，而其他线程申请内存失败的情况。

HeapFree 函数代码如下：

```

[kernel/kernel/heap.cpp]
static VOID HeapFree(LPVOID lpStartAddr, __HEAP_OBJECT* lpHeapObj)
{

```



```

__FREE_BLOCK_HEADER*   lpFreeHeader   = NULL;
__VIRTUAL_AREA_NODE*   lpVirtualArea = NULL;
__VIRTUAL_AREA_NODE*   lpVirtualTmp  = NULL;

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。

```

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)
{
    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，目的是为了完成一个合并操作。若待释放内存无法与一个虚拟区域对应，则可能是一个不正常的操作，因此直接返回。

```

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，插入当前堆的空闲链表中，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。

```

lpFreeHeader = (__FREE_BLOCK_HEADER*)(lpVirtualArea->lpStartAddress);
if((lpFreeHeader->dwFlags & BLOCK_FLAGS_FREE) &&
    (lpFreeHeader->dwBlockSize + sizeof(__FREE_BLOCK_HEADER)
    == lpVirtualArea->dwAreaSize))
{

```

```

lpFreeHeader->lpPrev->lpNext = lpFreeHeader->lpNext;
lpFreeHeader->lpNext->lpPrev = lpFreeHeader->lpPrev;
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.
}

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本身是不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不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个空闲块，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个内存块，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：

```

[kernel/kernel/heap.cpp]
static VOID CombineBlock(__VIRTUAL_AREA_NODE* lpVirtualArea, __HEAP_OBJECT* lpHeapObj)
{
    __FREE_BLOCK_HEADER*    lpFirstBlock    = NULL;
    __FREE_BLOCK_HEADER*    lpSecondBlock   = NULL;
    LPVOID                  lpEndAddr       = NULL;

    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 指向第二个内存块，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);
        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 的标志字段是否为空闲（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）更新 lpFirstBlock 为 lpSecondBlock，更新 lpSecondBlock 为 lpSecondBlock 的相邻块，继续下一轮迭代。

5. malloc 和 free 的实现

malloc 和 free 是标准 C 运行时库函数，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 函数可以很容易地模拟这两个函数，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：

```
[kernel/kernel/heap.cpp]
LPVOID malloc(DWORD dwSize)
{
    DWORD          dwFlags    = NULL;
    LPVOID          lpResult   = NULL;
    __HEAP_OBJECT*  lpHeapObj  = NULL;
    DWORD           dwHeapSize = 0L;

    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;
        CURRENT_KERNEL_THREAD->lpDefaultHeap = (LPVOID)lpHeapObj;
    }
    else
    {
        lpHeapObj = (__HEAP_OBJECT*)CURRENT_KERNEL_THREAD->lpDefaultHeap;
    }
    return HeapAlloc(lpHeapObj,dwSize);
}
```

首先，该函数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。

free 函数的实现更简单：

```
[kernel/kernel/heap.cpp]
VOID free(LPVOID lpMemory)
{
    __HEAP_OBJECT*  lpHeapObj = NULL;
    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。

5.6 Hello China 的内存管理机制总结

内存管理机制是操作系统最复杂最核心的机制, 对任何操作系统来说, 内存管理机制的好坏, 将直接影响操作系统的整体效率。作为面向智能领域的操作系统, Hello China 实现了比较灵活的内存管理机制, 把整个内存空间分成了几个区域, 采取不同的算法对其进行管理:

(1) 操作系统保留区域和操作系统核心代码加载区域, 即内存的起始 2MB 内存空间 (起始的 1MB 内存空间保留未用)。这 2MB 的内存空间不能分配, 操作系统加载的时候自动占用。

(2) 核心内存池, 进一步又分为以 4KB 为单位进行分配的内存池和以任意尺寸进行分配的内存池。其中以 4KB 为单位分配的内存池, 是供驱动程序、文件系统等需要大量内存作为缓冲的功能实体使用的。而以任意尺寸分配的内存池, 则是操作系统核心使用的, 用于分配尺寸大小变化很大的核心数据结构。这两个内存池通过统一的接口——`KmemAlloc` 和 `KmemFree` 两个函数——进行操作。同时通过宏定义, 可灵活调整核心内存池的起始地址和大小。

(3) 分页内存池, 除去上述 (1) 和 (2) 之外的内存区域。这个区域的内存, 以 CPU 的页尺寸为大小 (比如 4KB) 进行管理和分配, 一个页框管理对象对所有处于这个范围的物理内存进行统一管理。这部分内存可供应用程序使用, 也可以供操作系统加载应用程序使用。操作系统内核可通过调用页框管理对象提供的函数, 来分配和释放这个范围内的内存 (以页面为单位)。但应用程序不能直接调用页框管理器提供的服务, 而只能调用 `VirtualAlloc` 来分配这个范围内的内存。由于 `VirtualAlloc` 的局限 (只能分配尺寸为页面大小整数倍的内存), Hello China 又实现了线程本地堆机制, 应用程序 (线程) 可通过 `malloc` 和 `free` 等符合 ANSI C 函数库标准的接口, 来灵活分配和释放这个区域的内存。

(4) 如果启用了虚拟内存功能, 操作系统提供 `VirtualAlloc` 和 `VirtualFree` 等函数, 来动态地在整个虚拟内存空间 (对 32 位 CPU 来说, 是 4GB) 范围内预留和释放内存区域。在预留和释放内存区域的时候, 可以设置特定的标志位, 控制 `VirtualAlloc` 等函数的行为。比如可以只预留虚拟内存空间而不分配实际的物理内存 (设备驱动程序经常使用该功能), 或者在预留虚拟内存空间时分配对应大小的物理内存, 等等。`VirtualAlloc` 通过调用页框管理器提供的服务, 来分配物理内存。页索引对象实现了虚拟内存到物理内存的映射管理。

之所以按区域划分内存, 并通过不同的机制来分别进行管理, 是为了能够满足各种场景下的需要。当然, 内存划分模式是由源代码中的宏定义来控制的, 读者可以通过修改这些宏定义, 来控制每个区域的大小和起始位置, 甚至取消某个或几个区域。这样的实现方式, 为用户提供了最大可能的灵活性。

第 6 章 系统调用的原理与实现

6.1 系统调用概述

系统调用（System Call）是实现应用程序与操作系统核心模块物理分离的基础。如果没有系统调用，那么应用程序必须与操作系统核心模块进行静态链接。这样在开发应用程序的时候，还必须把操作系统源代码（或编译后的二进制模块）纳入应用程序的项目范围。完成应用程序源代码的编译后，必须与操作系统进行静态链接，最终形成一个二进制模块。这样操作系统的可扩展性就大打折扣了。当然，在嵌入式开发领域，这种静态链接的模式特别常见，但是在通用操作系统或智能操作系统领域，操作系统核心模块与应用程序完全分离是一种常用的做法。

实现二进制模块完全分离的解决办法不止系统调用一种，动态链接方式也是一种解决办法。所谓动态链接，即应用程序和操作系统（或应用程序的不同模块）分开编译和链接，编译完成的二进制模块也完全独立。在运行的时候，再把应用程序和操作系统，或者应用程序之间的不同二进制模块链接在一起。比较典型的实现就是 Windows 操作系统的动态链接机制（DLL 库）。但这种方式需要有一种协议或规范，来定义模块之间的动态链接方式，同时也需要编译器和链接器的支持，有较大的局限。

系统调用是应用程序与操作系统之间实现模块物理分离的最重要机制。实际上，大多数系统调用在实现的时候，都是借助 CPU 的硬件机制来完成的。以 Intel 的 x86 系列 CPU 为例，该 CPU 提供了系统调用门机制。一般应用程序的代码运行在 CPU 的用户态，而操作系统代码则运行在核心态。用户态的代码，需要通过系统调用门“切入”核心态，完成系统功能的调用。这种实现方式比较简便，可充分利用 CPU 的硬件机制。但是也有其局限性，那就是对 CPU 硬件的依赖比较强，会增加 CPU 相关的汇编代码，源代码的移植性会降低。因此更加普遍的一种实现方式是，利用 CPU 的中断处理机制来实现系统调用。

在 DOS 时代，通过中断调用方式实现系统调用表现得最直接。只要是 DOS 程序员，对诸如 `int 0x21` 之类的中断调用一定不会陌生。通过这个中断，应用程序可调用 DOS 操作系统提供的服务。在 Windows 时代，实际上也是通过中断方式实现的系统调用，但由于 Windows 操作系统提供了基于 C 语言的 API，掩盖了底层的系统调用实现方式。实际上，Windows 程序员在调用 Windows 的 API 函数的时候，也是通过中断方式陷入到核心态，来调用操作系统核心服务的。比如，用户调用通过 `CreateThread` 函数创建一个线程，实际上 `CreateThread` 函数只是一个简单的封装，它又进一步调用了 `0x80`（Windows 的系统调用号）中断，陷入到核心代码后，才调用了真正的线程创建代码。下面的伪码，示意了 `CreateThread` 函数的用户态实现：

```
HANDLE CreateThread(LPVOID pStartAddr)
{
    __asm push pStartAddr
    __asm push _CREATE_THREAD
    __asm int 0x80
    __asm pop eax
    __asm pop eax
}
```

上述代码首先把函数的参数压入堆栈，再压入一个系统调用号（系统调用号需要与操作系统核心保持一致），然后使用 `int` 指令引发一个中断。`int` 指令被执行后，运行路径已切入到了核心态。按照 Intel CPU 的实现，CPU 会在全局描述符表（GDT）中，索引到第 0x80 个表项，从中找到一个函数的入口地址，然后跳转到这个入口处继续执行。这个入口函数，再根据堆栈内保存的功能号（`_CREATE_THREAD`，是一个宏定义常数）调用对应的功能。下面的伪码大致描述了这个入口函数的处理方式：

```
VOID EntryOfSysCall(int FunctionNum)
{
    switch(FunctionNum)
    {
        case _CREATE_THREAD:
            _CreateThread(...);
            break;
        .....
    }
}
```

需要注意的是，`_CreateThread` 函数是真正的内核代码，正是这个函数创建了线程。

后面我们把 `CreateThread` 函数在用户态的封装，叫做系统调用的代理，把 `EntryOfSysCall` 叫做系统调用在核心态的存根。系统调用的代理，一般被编译到一个静态库中，在开发应用程序的时候，必须包含此静态库，同时在源代码中包含定义系统调用代理函数的头文件（以 C 语言为例），就可实现系统调用代理与用户代码的静态链接。注意，这里的静态链接，只是系统调用代理函数与用户代码链接到了一起，真正的操作系统功能函数（比如 `_CreateThread`），是在操作系统核心模块中实现的，不需要与用户代码链接。而系统调用的存根，则是操作系统本身实现的，与应用程序没有关系。系统调用存根代码，与真正的操作系统功能函数（比如 `_CreateThread`）是静态链接的。图 6-1 说明了整个关系。

上图中的用户空间和系统空间，实际上都是 4GB（32 位 CPU）大小的连续内存空间。所谓用户空间，是 4GB 内存空间中，用户态代码可访问的部分内存区域。而系统空间，则是必须切换到内核态才能访问的内存区域。一般来说，用户空间大小要小于系统空间，系统空间包含用户空间，因为用户空间的内容，处于内核态的代码也可以访问。在 Windows 的实现中，大部分版本的用户空间为 2GB，而系统空间则为完整的 4GB 内存空间。还有一种说法是，系统空间是内核态代码和数据所占用的空间，这样在 Windows 的实现中，系统空间就特指 4GB 空间中，除去用户空间后所剩余的内存区域，一般是从 0x80000000 到 0xFFFFFFFF 之间的 2GB 空间。但不论怎么说，用户空间和系统空间的最根本划分依据，是

代码的执行权限。处于用户态的代码，只能直接访问用户空间，而处于核心态的代码，则可以访问整个 4GB 内存空间。处于用户态的代码要访问系统空间（严格来说，应该是系统空间中除去用户空间的内存区域），必须通过系统调用，陷入到内核态才能访问，否则会引发异常。这时候，系统调用除用于物理分离操作系统核心模块和应用程序模块外，还具备权限转换的功能。

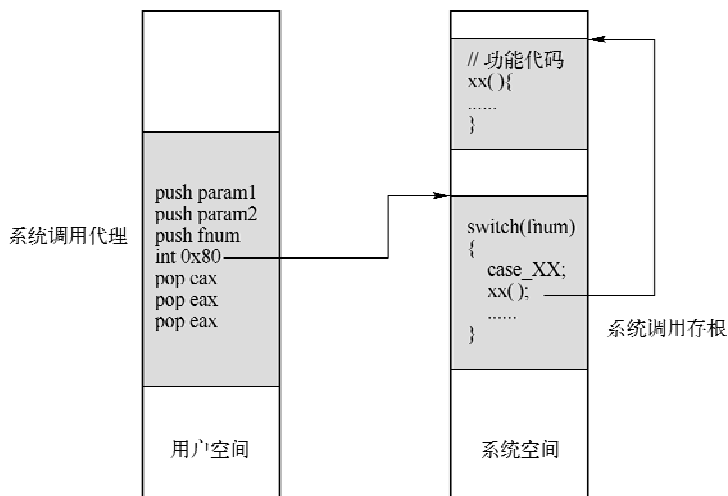


图 6-1 系统调用与功能函数间的关系

在 Hello China V1.75 的实现中，由于没有实现进程的概念，整个系统只有一个逻辑的内存空间，操作系统核心代码和应用程序都运行在这个空间内，且都以 CPU 的内核态来运行，因此没有用户空间和系统空间的概念。理论上说，用户应用程序的代码，可直接访问操作系统核心数据结构，但很显然，这样做是不合法的，可能导致整个系统崩溃。比较规范的做法，是与实现了进程概念的操作系统一样，使用系统调用来请求操作系统内核服务和操作内核数据结构。

这里顺便提一下，采用“空间”的概念，来描述整个 CPU 的内存寻址范围，是勉强能说得过去的。但是用于描述用户态和核心态能够访问的内存区域，严格来说是有问题的。因为从数学意义上讲，空间是一个集合，有严格的数学定义，其中最基本的一条就是，对加法运算是封闭的。比如，有两个内存地址 A 和 B，都属于同一个地址空间，那么地址 A+B，也必须在相同的地址空间内。显然，CPU 的整个内存寻址范围可以认为是一个地址空间，因为对于地址的加法是封闭的（地址相加并不是直接加，而是相加之后，对整个地址范围取模）。但用户空间的说法就不合适了，假设 A 和 B 两个内存地址都在用户空间内，但是 A+B 可能就不在用户空间内。但大多数操作系统书籍都采用了用户空间、系统空间、内核空间（系统空间的另一种说法）等概念，且得到广泛认同，所以读者就不必去钻“这些说法不符合空间的数学定义”这个牛角尖了。毕竟在这个世界上这种情况比比皆是，或许正是因为这些矛盾的存在，才使得这个世界丰富多彩。

好了，言归正传。在理解了系统调用代理、系统调用存根、用户地址空间、系统地址空间等概念后，接下来将详细讲解 Hello China 的系统调用实现机制。

6.2 Hello China 系统调用的实现

在 Intel x86 CPU 上, Hello China 系统调用利用了 CPU 提供的中断处理机制, 采用第 128 号软件中断 (异常) 作为系统调用的入口点。但在 Hello China 初始化的时候, 把从第 32 号开始的所有中断描述表 (IDT) 表项, 都设置成了中断调用门。中断调用门与异常门不同的是, 在中断调用门中, CPU 在调用中断处理程序前, 会自动清除 CPU 的中断允许标志。这样的处理是适合硬件中断的处理要求的, 但对于系统调用来说, 就不太合适了。因此, 需要修改第 128 号中断 (异常) 的入口处理程序, 通过软件方式打开中断标志。下面是第 128 号中断的处理程序:

```
[kernel/arch/sysinit/miniker.asm]
gl_syscall:                ;;System service call entry point.
    sti
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ebp
    mov eax,esp
    push eax
    mov eax,0x80
    push eax
    call dword [gl_general_int_handler]
    pop eax
    pop eax
    mov esp,eax
    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx
.ll_end:
    pop eax
    iret
```

该入口程序与所有硬件中断处理程序一样, 首先保存通用寄存器, 把异常或中断号压入堆栈, 然后再调用通用中断处理程序 (`gl_general_int_handler` 标号处的函数)。与普通中断处理程序的唯一不同是, 在开始处, 就打开了 CPU 的中断标志位 (`sti` 指令)。`gl_syscall` 作为第 128 号中断的处理程序, 在初始化中断描述符表 (IDT) 的时候, 就被填写到了中断描述符表的第 128 号表项中。这样一旦用户应用程序代码执行了 `int 0x80` 指令, CPU 就根据中断描述符表寄存器 (`idtr`) 的内容, 找到 IDT 的首地址, 然后再定位到第 128 号表项, 从中找

到中断处理程序的入口地址（即 `gl_syscall` 标号），跳转到该处执行。因此 `gl_syscall` 标号处的代码，就是系统调用存根的一部分，注意不是全部，因为 `gl_syscall` 又进一步调用了通用中断处理程序（`gl_general_int_handler`），而通用中断处理程序是用 C 语言编写的一个通用函数，这个函数根据中断编号，又调用了系统调用分发函数，系统调用分发函数才根据系统调用功能号，调用了真正的系统功能函数。由此可见，系统调用的存根，可能是由多个函数组成的，可以说，从 128 号中断处理程序（`gl_syscall` 标号的代码）开始，一直到真正的功能函数被调用之前的所有代码，都叫做系统调用存根。

通用中断处理程序采用 C 语言实现，该处理程序首先判断当前发生的是异常还是中断。若是异常，则调用 `DispatchException` 函数，从而根据异常向量调用对应的异常处理函数。若是中断，则调用 `DispatchInterrupt` 函数，该函数根据中断向量号调用对应的处理程序。不论中断还是异常，处理方式都是相通的，就是根据中断或异常向量号，索引一个全局数组，这个全局数组记录了处理对应中断或异常的中断对象（`InterruptObject`），在中断对象中记录了处理函数的入口地址。对于异常来说，这个处理函数是 `SyscallHandler`，下面是其实现代码（为了便于描述，做了适当简化）：

```
[kernel/kernel/syscall.cpp]
BOOL SyscallHandler(LPVOID lpEsp, LPVOID)
{
    __SYSCALL_PARAM_BLOCK* pspb = (__SYSCALL_PARAM_BLOCK*)lpEsp;

#define PARAM(i) (pspb->lpParams[i]) //To simplify the programming.
    switch(pspb->dwSyscallNum)
    {
        case SYSCALL_CREATEKERNELTHREAD:
            pspb->lpRetVal = (LPVOID)CreateKernelThread(
                (DWORD)PARAM(0),
                (DWORD)PARAM(1),
                (DWORD)PARAM(2),
                (__KERNEL_THREAD_ROUTINE)PARAM(3),
                (LPVOID)PARAM(4),
                (LPVOID)PARAM(5),
                (LPSTR)PARAM(6));
            break;
        .....
        default:
            if(!DispatchToModule(lpEsp, NULL))
            {
                PrintLine(" SyscallHandler: Unknown system call is requested.");
            }
            break;
    }
    return TRUE;
}
```

显然，这个函数无非是根据系统调用的功能号和传递过来的参数，调用了操作系统内核

提供的功能函数。__SYSCALL_PARAM_BLOCK 是定义的一个结构体，用于传递系统调用相关的所有信息，包括系统调用功能号、对应的参数，调用完成后的返回值，也是存放在这个结构体中。这个结构体的具体定义和用法，将在下一节中做详细介绍。

6.3 系统调用时的参数传递

前面讲过，系统调用涉及几个概念：代理函数、存根代码（可能包含多个函数）和服务例程，其中代理函数与用户应用程序链接在一起，形成一个可执行的二进制模块。而存根代码和服务例程则是由操作系统核心实现的。代理函数是用户态的系统调用服务函数，该函数只建立合适的堆栈框架，然后发起一个软件中断，从而使得当前执行环境切换到内核中。系统调用存根代码，则是核心态的系统调用功能支撑代码，这些代码提取相关的函数参数，并根据系统调用功能编号，调用对应的服务例程。而服务例程则是运行在内核中的实际功能函数，用于完成实际的系统功能，然后把结果返回给发起系统调用的应用程序。

以 CreateKernelThread 函数为例，该函数的系统调用代理函数代码如下（为便于解释，代码做了修改，主要是展开了宏定义）：

```
[kernel/kapi/kapi.cpp]
HANDLE CreateKernelThread(DWORD dwStackSize,
    DWORD dwStatus,
    DWORD dwPriority,
    __KERNEL_THREAD_ROUTINE lpStartRoutine,
    LPVOID lpParam,
    LPVOID lpReserved,
    LPSTR lpzName)
{
    __asm{
        push lpzName
        push lpReserved
        push lpParam,
        push lpStartRoutine,
        push dwPriority,
        push dwStatus,
        push dwStackSize,
        push 0
        push SYSCALL_CREATEKERNELTHREAD
        int 0x80
        pop eax
        pop eax
        pop dwStackSize
        pop dwStatus
        pop dwPriority
        pop lpStartRoutine
        pop lpParam
        pop lpReserved
```

```
    pop lp.szName  
}  
}
```

上述汇编代码中，`int 0x80` 处是一个分界点。前面部分代码，是把函数的参数压入堆栈，同时压入一个空的双字，以存储函数的返回值，然后压入系统调用编号，以指示内核调用哪个服务例程。这些汇编指令执行完毕，堆栈框架如图 6-2 所示。

在压入系统调用编号前，压入一个 0，以预留返回值空间。在内核完成系统服务函数的执行后，会把服务函数的返回值存储在这个地方。因此在 `int 0x80` 指令后的两条 `pop` 指令中，就把该返回值存入了 `EAX` 寄存器，从而可以直接返回给调用程序。

在 `int 0x80` 汇编指令执行后，对应的堆栈框架如图 6-3 所示。

SYSCALL_KERNEL...
0
dwStackSize
dwStatus
dwPriority
lpStartRoutine
lpParam
lpReserved
lp.szName

图 6-2 堆栈框架

EIP
CS
EFlags
SYSCALL_KERNEL...
0
dwStackSize
dwStatus
dwPriority
lpStartRoutine
lpParam
lpReserved
lp.szName

图 6-3 `int 0x80` 指令执行后的堆栈框架

这就是一个典型的中断发生后的堆栈框架。在中断处理程序（`gl_syscall` 标号处的汇编代码，参见前一节内容）入口处，汇编语言入口函数又把通用寄存器保存在了堆栈里面，这样在调用通用中断处理函数（`GeneralIntHandler`，该函数被写入汇编标号 `gl_general_int_handler` 处）时，堆栈框架如图 6-4 所示。

而通用中断处理函数 `GeneralIntHandler` 的原型如下：

```
[kernel/include/system.h]  
VOID GeneralIntHandler(DWORD dwVector, LPVOID lpEsp);
```

于是 `lpEsp` 参数就指向了上述堆栈框架的箭头处（注意，`ESP` 寄存器的值是通过先存入 `EAX`，再压入堆栈的。这样在堆栈内的 `ESP` 值，实际上是指向 `EBP` 处的）。要理解这个框架，需要读者非常清楚每条指令执行后引起的堆栈指针的变化。如果在这个地方感到迷茫，也不要紧，可参考本书第 8 章，或者参考 Intel 的 CPU 指令用户手册。或者只要记住 `lpEsp` 参数指向图 6-4 中的箭头处就可以了，至于是否理解指向这个位置的原因，并不影

响后续阅读。

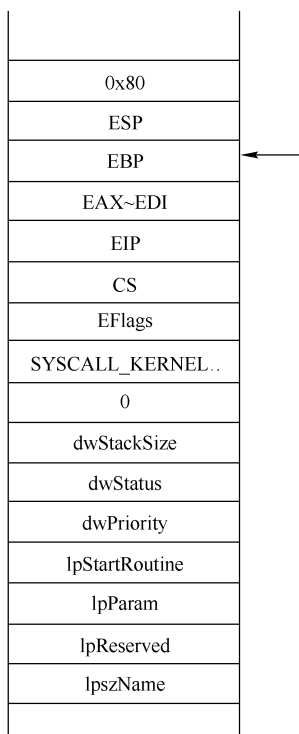


图 6-4 调用通用中断处理函数时的堆栈框架

GeneralIntHandler 根据中断号码，判断是异常还是中断。如果是中断，则调用 **DispatchInterrupt** 函数，该函数进一步会调用 **SyscallHandler** 函数，而 **SyscallHandler** 函数会根据系统调用功能号，调用真正的功能例程：

```
[kernel/kernel/syscall.cpp]
BOOL SyscallHandler(LPVOID lpEsp, LPVOID)
{
    __SYSCALL_PARAM_BLOCK* pspb = (__SYSCALL_PARAM_BLOCK*)lpEsp;

#define PARAM(i) (pspb->lpParams[i]) //To simplify the programming.
    switch(pspb->dwSyscallNum)
    {
    case SYSCALL_CREATEKERNELTHREAD:
        pspb->lpRetVal = (LPVOID)CreateKernelThread(
            (DWORD)PARAM(0),
            (DWORD)PARAM(1),
            (DWORD)PARAM(2),
            (__KERNEL_THREAD_ROUTINE)PARAM(3),
            (LPVOID)PARAM(4),
            (LPVOID)PARAM(5),
            (LPSTR)PARAM(6));
```

```
        break;
    ...
    default:
        if(!DispatchToModule(lpEsp,NULL))
        {
            PrintLine(" SyscallHandler: Unknown system call is requested.");
        }
        break;
    }
    return TRUE;
}
```

注意，SyscallHandler 的 lpEsp 参数，就是 GeneralIntHandler 函数传递过去的，依然指向堆栈框架的 EBP 处。

__SYSCALL_PARAM_BLOCK 是预先定义的一个结构体，如下：

```
[kernel/include/syscall.h]
typedef struct SYSCALL_PARAM_BLOCK{
    DWORD          ebp;
    DWORD          edi;
    DWORD          esi;
    DWORD          edx;
    DWORD          ecx;
    DWORD          ebx;
    DWORD          eax;
    DWORD          eip;
    DWORD          cs;
    DWORD          eflags;
    DWORD          dwSyscallNum;
    LPVOID          lpRetValue;
    LPVOID          lpParams[1];
}__SYSCALL_PARAM_BLOCK;
```

可以看出，__SYSCALL_PARAM_BLOCK 实际上就是反映了系统调用过程中建立起来的堆栈框架结构。通过这个结构，系统调用存根代码可以访问到任何系统调用相关的信息，包括功能号、相关参数等。对于功能例程的返回值，也是填写到这个结构体中（实际上就是填写到了堆栈内，在系统调用代理函数返回时，可直接返回给用户程序）。

SyscallHandler 根据系统调用编号，调用对应的系统调用服务例程，对于有返回值的系统调用服务例程，则把返回值强制转换为 LPVOID，并存储在系统调用参数块对象中。这实际上是存储在了堆栈中的返回值位置处。这样在系统调用的代理函数中，就会把返回值返回给用户程序。需要注意的是，在 __SYSCALL_PARAM_BLOCK 的定义中，只定义了一个用户参数——lpParams[1]，实际上这是一个可扩展数组，可以通过增加索引（比如 lpParam[2]）的方式，来访问更多的用户参数。

系统调用实现后，基于 Hello China 的应用程序开发就非常简单了。只需在源文件中包

含一个系统调用头文件，而无需把所有 Hello China 操作系统的核心代码都包含进来。这样可大大减小软件工程的源代码规模，且使操作系统和应用程序完全独立，有利于以模块化方式实现大型软件系统的开发。

系统调用的实现机制讲完了，希望读者能够对系统调用的实现有一个清晰的理解。通用操作系统，比如 Windows 和 Linux 的系统调用实现与此类似，不同的是这些通用操作系统需要处理用户态堆栈和核心态堆栈之间的数据转换，这实际上就是一些内存复制操作，比较简单。如果读者对这部分内容仍然存有疑问，可到网络上搜索相关资料做进一步了解。据作者了解，在互联网上，系统调用实现机制相关的文章非常多，而且大多数都讲得比较深入浅出。

第 7 章 线程互斥和同步机制的实现

7.1 互斥和同步概述

在操作系统的设计中，尤其是在引入多任务的情况下，需要考虑下列几种可能的“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。

Hello China 当前版本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几种竞争关系。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有余地，便于今后向多 CPU 系统平滑过渡。

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，则用来实现事件对象（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。本章仅对信号量对象的实现做详细描述，其他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

Critical Section（关键区段）是一段可执行的代码。这段代码在执行过程中，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，如果在修改的过程中被打断，可能会导致这些全局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，执行过程中不会受到任何干扰。一旦代码执行完毕，就调用 `__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，在本书后续部分介绍）；`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，修改时，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)
```

在接下来的几节中将详细介绍关键区段产生的原因以及当前版本 `Hello China` 关键区段的实现方式。本章还将对 `Power PC` 环境下的互斥机制进行简单的描述，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）的最根本目的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。操作系统中常见的竞争修改发生原因如下所示。

7.3.1 多个线程之间的竞争

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来反映这种情况（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，可能是下面这样：

```
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，由于 `nKernelObject` 的值还没有被修改，所以仍然是 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，B 修改

了 `nKernelObject`，使得 `nKernelObject` 变成了 101，而不是原来的 100。然后线程 A 又被唤醒，继续运行，由于线程 A 被挂起的时候，刚刚执行完 `inc eax` 指令，所以线程 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，因此当前 `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` 变量时，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原因，是线程与中断处理程序之间的竞争。例如，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，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 1（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 `mov nKernelObject, eax` 指令。由于线程 A 被打断的时候，`eax` 的值是 101，所以，`nKernelObject` 被 `eax` 覆盖后，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（对称多处理器）环境下，情形还要复杂，因为不仅有多线程之间的竞争，还存在多个 CPU 之间的竞争。仍然以上述假设为例，`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，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，如果出现表 7-1 所示运行序列就会产生不一致现象。

表 7-1 一个产生访问冲突的指令序列

时 刻	线程 A 执行的指令	线程 B 执行的指令	<code>nKernelObject</code> 的值
N	...	<code>mov eax, nKernelObject</code>	100
N + 1	<code>mov eax, nKernelObject</code>	<code>inc eax</code>	100
N + 2	<code>inc eax</code>	<code>mov nKernelObject, eax</code>	101
N + 3	<code>mov nKernelObject, eax</code>	...	101
N + 4	...		

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，且因为线程被打断而产生的不一致状态也仍然存在。

7.4 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，即当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在试图修改同一变量。

因此，在单 CPU 环境下，只要在修改关键数据结构的代码段的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）的，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后重新开启中断，代码如下：

```
__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地重新打开了中断（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，因此调用这两个函数的时候，首先禁止了中断。但 `IncreaseGlobal` 函数，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，中断已经打开，此时 `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地关闭或打开了中断，而没有考虑调用自己的更上级函数的实际情况。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合理的。一个合理的实现是，首先在关键区段的开始处保存 `EFLAGS` 寄存器的值，然后再关闭中断，在关键代码段的结束处恢复先前保存的 `EFLAGS` 的值，这样就可以确保不影响原始 `EFLAGS` 寄存器的值了。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。代码如下：

```

VOID IncreaseGlobal()
{
    __asm{
        pushfd    //Save EFLAGS register.
        cli
    }
    nKernelObject++;
    __asm{
        popfd     //Restore EFLAGS register.
    }
}

```

从 `IncreaseGlobal` 函数返回后，原来设置的中断标志得以保存，从而使得调用函数的关键区段使之连续。

`Hello China` 的实现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，代码如下：

```
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`。

7.5 多 CPU 下关键区段的实现

7.5.1 多 CPU 环境下的实现方式

在多 CPU（比如对称多处理 SMP）环境下，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，对这些全局数据进行修改时，首先检测该保护标志，如果该保护标志没有被设置（说明没有其他线程正在修改数据），再设置该保护标志，并进入修改数据的关键代码段，完成修改（即在关键代码段的最后）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）为止。上述过程，可以采用伪代码进行描述：

```
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，由于此时没有其他线程在修改该全局变量，所以保护标志处于清除（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。

此时，线程 B 也试图修改全局数据，并执行与 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，并已经设置了保

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，恢复保护标志为 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地完成多个线程之间的同步。

细心的读者会发现，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 也认为没有其他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，而不是作为一个整体的操作。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` 中的第 `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`），在该指令执行的时候锁定总线，这样在该指令执行的过程中，其他 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。

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相关 bit 即可。在此不再详述。

7.5.2 Hello China 的未来实现

需要说明的是，Hello China 当前版本的实现，是针对单 CPU 环境的，没有实现支持多 CPU 的版本。但为实现支持多 CPU 的版本预留了扩展接口。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);  
//修改 KernelThread 对象的互斥代码  
__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。

7.6 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，通过清除 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 CPU 环境下的同步，Power PC 提供了与 IA32 不同的机制来完成。本节简单介绍这种机制，并给出几个实例。

7.6.1 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 CPU 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（可能是其他的处理器），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（由该指令的操作数寄存器指定内存位置和指定字，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，则该指令不做任何其他操作，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。

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他的用途，在此不再详述。



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 r5 寄存器，并设置 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 0，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，成功地把 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，则表 7-2 所示的情况可能发生。

表 7-2 一个产生访问冲突的指令序列（双 CPU）

CPU1 指令序列	CPU2 指令序列
任意指令	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	任意指令
Bne- loop	

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，所以 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 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 设置成功，其他 CPU 都没有设置成功，从而实现了共享资源的同步和保护。

7.6.2 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，若获取成功，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，从而导致该例程又一次被调用，直到获得自旋锁为止。

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。

7.7 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）关键代码段不能太长，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，那么它必须等待。如果正在修改的线程长时间地占有保护锁，则会导致其他线程长时间等待，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，这样如果长时间地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（CPU 资源）浪费，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。

7.8 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, 则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。需要注意的是, 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, 则 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)。

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), 则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 演变为一个简单的 mutex 对象(互斥体对象), 可以保护临界区的并发访问。之所以用简单的 mutex 对象来形容 N 为 1 的情况, 是因为 semaphore 对象不支持 mutex 对象支持的部分功能(比如递归调用等), 而仅仅支持 mutex 对象功能的一个子集。

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。

7.9 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 CPU)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 7.8 节中的 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，可以调用 SetSemaphoreCounter 函数来设置新的 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（函数中没有指定超时时间的参数），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 或 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，使用时应该谨慎。

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。

7.10 Semaphore 对象的实现

本节对 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）。

7.10.1 Initialize 和 Uninitialize 的实现

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 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 函数的实现代码。

```
static BOOL SemInitialize(__COMMON_OBJECT* lpObject)
{
    __SEMAPHORE*      lpSem          = (__SEMAPHORE*)lpObject;
    __PRIORITY_QUEUE* lpWaitingQueue = NULL;
    BOOL               bResult        = 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))
        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，不再赘述。

7.10.2 WaitForThisObject 的实现

WaitForThisObject 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地继续运行。

(2) 如果 nCounter 递减后的结果小于 0，则说明当前已经没有资源可供使用，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，选择其他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代码。

```
static DWORD WaitForSemObject(__COMMON_OBJECT* lpObject)
{
    __SEMAPHORE*      lpSem          = lpObject;
    __KERNEL_THREAD_OBJECT* lpKernelThread = NULL;
    DWORD              dwFlags        = 0L;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    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,
    (__COMMON_OBJECT*)lpKernelThread,
    0L);
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    KernelThreadManager.ScheduleFromProc(&lpKernelThread->Context);
    return 0L;
}
```

需要注意的是，上述实现中，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，即这些操作在执行过程中不会被中断，这样可以充分确保所有这些操作能够原子地完成。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。

7.10.3 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，资源如果能够在时间参数超时前（从调用开始计时）可用，则返回资源可用指示，否则，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，以便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。

上述描述，提到了一个定时器超时的时候调用的回调函数，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，回调函数就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，而 TA 设定的定时器还未超时。

这种情况下，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（一次性定时器在超时后，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;

    lpKernelThread = lpSemParam->lpKernelThread;
    lpSem          = lpSemParam->lpSemaphore;
    if(!lpSem->lpWaitingQueue->DeleteFromQueue((__COMMON_OBJECT*) lpSem->lpWaitingQueue,
```

```

(__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，该函数首先试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`）。

下面是 `WaitForThisObjectEx` 函数的实现代码。

```

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。

需要注意的是，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。如果 SemCallback 被调用，则 dwWakeupReason 将被修改为 TIMEOUT，否则，会一直保持 SEMAPHORE_WAIT_RESOURCE。在当前线程被唤醒之后（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。

7.10.4 ReleaseSemaphore 的实现

ReleaseSemaphore 函数的实现相对简单。在这个函数中，首先对 nCounter 进行递增操作，然后判断递增后的结果是否大于 0。如果大于 0，则说明当前没有线程等待资源，于是直接返回，否则，说明有线程等待资源，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，然后返回。实现代码如下。

```

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)
{
    __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，该函数没有进行相应的安全检查，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。在使用该对象的时候，一定要注意这一点。

7.11 互斥和同步机制总结

本章简单介绍了互斥和同步机制产生的原因以及实现机理，并以信号量（Semaphore）的实现为例，介绍了互斥和同步对象的具体实现。实际上，互斥和同步机制是操作系统内核设计中最复杂的问题（注意，这里没有“之一”）。如果不考虑资源共享冲突，操作系统的多任务模型是很简单的，但共享资源是客观存在的，有效协调共享资源的访问，使操作系统的任务模型变得复杂。互斥和同步机制设计的好坏将直接决定操作系统的整体性能。这好比高速公路，如果没有交叉路口，没有收费站，只有划分明晰的一条条车道，则会非常快速通畅。一旦有了共享的收费站、交叉口等，整体通行效率就会受到极大影响。但是如果这些共享资源的访问机制设计得非常好，则效率会得到大大提升。

同时，互斥和同步机制有许多复杂问题需要考虑，比如多对象同时等待、互斥对象的安全删除、优先级翻转等。在 `Hello China` 的实现中，这些问题都做了仔细考虑，并提供了完整的实现。但本章没有提到这些内容。这主要是为了限制篇幅、降低内容的复杂度。但是读者在掌握本章所讲内容的基础上，通过自行阅读代码，很容易理解这些更加复杂机制的实现机理。可以说，本章内容是互斥和同步机制的基础。

第 8 章 中断和定时处理机制的实现

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，就会引发一个异常。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（比如已经被暂时置换到硬盘上的可执行代码），就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，CPU 并不是在中断发生后马上跳转到相应的处理程序，而只是在指令的执行边界检查是否有中断发生。如果没有，CPU 继续执行下一条指令。一旦有中断发生，会引起 CPU 设置内部的特定寄存器位，然后继续执行当前指令（中断发生时，正在执行的指令），只有当前指令执行完毕，相应的中断才有机会得到处理。

不同的 CPU 类型，其异常或中断的处理机制是不同的，比如 Intel CPU 对系统中可能产生的异常进行了编号，一旦异常发生，CPU 就根据异常类型找到对应的编号，然后根据异常编号，查找中断描述符表（IDT），找到对应的处理程序，再跳转到具体的处理程序。对于中断，也是采取类似的方式，只不过中断的编号（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，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，采用了中断描述符表与中断处理链方式结合的实现方式，即首先有一个中断描述符表，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，定位到一个中断描述符表项，在每个中断描述符表的表项中，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。

本章以 Intel CPU（IA32 架构）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。

8.2 硬件相关部分处理

8.2.1 IA32 中断处理过程

硬件相关部分处理指的是针对不同的 CPU 平台采用不同的中断处理方法。比如针对 Intel IA32 架构的 CPU，需要建立 IDT（中断描述符表），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。下面针对 IA32 架构的 CPU 进行描述，需要说明的是，硬件相关处理仅仅是为了适应不同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，主要功能就是完成硬件部分的处理，并完成与 Hello China 本身中断处理机制之间的连接工作。

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 64bit 的数据结构，其每个比特的内容及含义都是由硬件严格定义的。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，使用 `iret` 指令，从中断处理程序中返回（该指令的含义可参考本书第 4 章）。
- （5）CPU 继续执行中断处理前的任务。

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，最小化地利用了 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，在 Hello China 的实现中，上述简单的处理过程描述已经足够。

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。
- 第（2）步非常简单，只需要如下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。

接下来将对 IDT 的填写进行详细描述。

8.2.2 IDT 初始化

图 8-1 是 IA32 体系架构定义的中断描述符 (Interrupt Descriptor) 的结构。

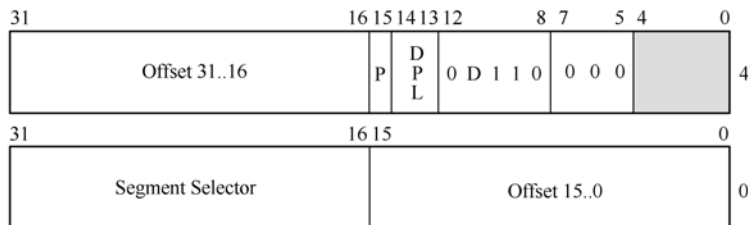


图 8-1 中断描述符的结构

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 32bit 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段内, 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 **DPL** 字段的地方, 都设置为 0, **D** 字段设置为 1 (32 位操作系统), **P** 字段也设置为 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。为方便起见, 在当前版本的实现中为中断描述符中前 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 上只有 16 个外部中断, 分别对应中断描述符表中的第 32~47 个中断描述符。目前, 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 (采用 C 语言实现这个处理程序)。
- (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                ;;保存通用寄存器
    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                 ;;恢复通用寄存器
    pop eax
    mov esp,eax
    pop ebp
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx
.ll_continue:

    mov al,0x20            ;;解除中断
    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_handler` 是在 `[kernel/arch/sysinit/MINIKER.ASM]` 文件中定义的一个 32 比特的全局变量，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图 8-2 所示。

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` 之前形成如图 8-3 所示的堆栈框架。

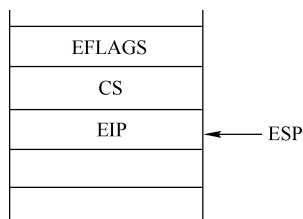


图 8-2 中断发生后的堆栈框架

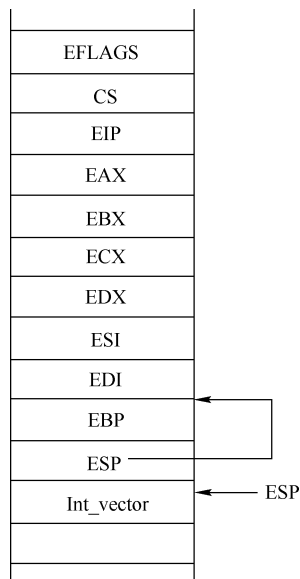


图 8-3 执行中断处理程序前建立的堆栈框架

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 8-3 中的折线示意位置）。之所以保存 ESP 的值，是要把 ESP 作为通用中断和异常处理程序的一个参数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)` 函数。`GeneralIntHandler` 函数就可以通过 `dwVector` 和 `lpEsp` 直接访问中断向量号和堆栈框架了。

对异常的处理与中断处理类似，唯一不同的是，对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，对异常的处理，IA32 CPU 会根据异常类型的不同有选择地向堆栈中压入一个异常错误号，这样就导致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。

各个中断和异常处理程序编写完成之后，只需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 处即可。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，不再赘述。

最后说明一下 gl_general_int_handler 和 GeneralIntHandler 之间的关系。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，与其他全局变量不同的是，gl_general_int_handler 被声明在固定的位置，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 48KB，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 1MB 开始的地方，而 `MASTER.BIN` 则被加载到物理内存 `0x00110000` 位置处（1MB+64KB），如图 8-4 所示。

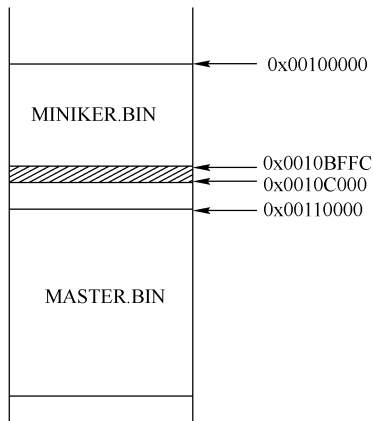


图 8-4 Hello China 相关模块在内存中的位置

其中阴影部分是在 `MINIKER.ASM` 中声明的 `gl_general_int_handler` 变量的位置。`MASTER.BIN` 初始化时，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，为了链接 `MINIKER.BIN` 和 `MASTER.BIN` 两个模块，大量地采用了这种方式。

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 大量地采用了面向对象的思想进行设计，把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。

下面代码是 `System` 对象的定义：

```
BEGIN_DEFINE_OBJECT(__SYSTEM)
    __INTERUPT_OBJECT*      lpInterruptVector[MAX_INTERRUPT_VECTOR];
    __PRIORITY_QUEUE*       lpTimerQueue;
    DWORD                   dwClockTickCounter;
    DWORD                   dwNextTimerTick;
    DWORD                   dwPhysicalMemorySize;
    BOOL                    (*Initialize)(__COMMON_OBJECT* lpThis);
    DWORD                   (*GetClockTickCounter)(__COMMON_OBJECT* lpThis);
    DWORD                   (*GetPhysicalMemorySize)(__COMMON_OBJECT* lpThis);
    VOID                    (*DispatchInterrupt)(__COMMON_OBJECT* lpThis,
```

```

                                LPVOID          lpEsp,
                                UCHAR            ucVector);

__COMMON_OBJECT* (*ConnectInterrupt)(__COMMON_OBJECT* lpThis,
__INTERRUPT_HANDLER InterruptHandler,
LPVOID          lpHandlerParam,
UCHAR           ucVector,
UCHAR           ucReserved1,
UCHAR           ucReserved2,
UCHAR           ucInterruptMode,
BOOL            blfShared,
DWORD           dwCPUMask
);

VOID (*DisconnectInterrupt)(__COMMON_OBJECT* lpThis,
__COMMON_OBJECT* lpIntObj);

__COMMON_OBJECT* (*SetTimer)(__COMMON_OBJECT* lpThis,
__KERNEL_THREAD_OBJECT* 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。

先介绍 lpInterruptVector。这是一个中断对象数组。所谓中断对象是 Hello China 定义的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，所以遵循 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 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（或者其他实体）如果想连接一个中断，需调用 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` 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` 变量为索引找到 `lpInterruptVector` 数组的一个特定元素（`lpInterruptVector[ucVector]`），这个元素是一个指向中断对象链表的指针，`ConnectInterrupt` 会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图 8-5 所示形式组织。

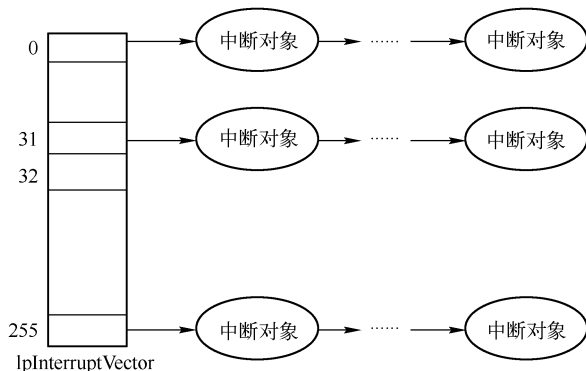


图 8-5 Hello China 中断对象的组织

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都按照中断向量号（`ucVector`）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。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，该函数用如下代码实现：

```
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，仅仅是调用了 `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 为止。

代码如下：

```
static VOID DispatchInterrupt(__COMMON_OBJECT* lpThis,
                             LPVOID          lpEsp,
                             UCHAR ucVector)
{
    __INTERRUPT_OBJECT* lpIntObject = NULL;
    __SYSTEM*          lpSystem     = NULL;

    lpSystem = (__SYSTEM*)lpThis;
    lpIntObject = lpSystem->lpInterruptVector[ucVector];
    if(NULL == lpIntObject)    //无中断对象与该中断向量对应
    {
        DefaultIntHandler(lpEsp,ucVector);
        return;
    }
    while(lpIntObject)        //遍历整个链表
    {
        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。

下面是中断处理函数的原型：

```
BOOL    InterruptHandler(LPVOID lpESP,LPVOID lpParam);
```

其中，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，该函数一旦被调用，需要尽快判断自己是不是对应的中断处理程序（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返回 FALSE。

8.3.3 缺省中断处理函数

从 DispatchInterrupt 函数的处理过程可以看出，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 函数。目前该函数实现下列功能：



- (1) 打印 “Unhandled interrupt or Exception!” 提示信息。
- (2) 打印相应的中断向量号。
- (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, 可以进一步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;
}
```

将来的版本中, 可以针对未处理异常或中断进行进一步改进。

8.4 对外服务接口

设备驱动程序 (或其他实体) 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。声明代码如下所示：

```
__COMMON_OBJECT* (*ConnectInterrupt)(__COMMON_OBJECT* lpThis,
__INTERRUPT_HANDLER InterruptHandler,
LPVOID lpHandlerParam,
UCHAR ucVector,
UCHAR ucReserved1,
UCHAR ucReserved2,
UCHAR ucInterruptMode,
BOOL bIfShared,
DWORD dwCPUMask
);
```

目前版本的实现中，驱动程序只需要给出 InterruptHandler、lpHandlerParam、ucVector 即可，其他参数都是为了将来扩展而保留的，目前都设置为 NULL。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，检索 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。该函数将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。该函数原型如以下代码所示：

```
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删除 lpIntObj。
- (2) 销毁相应的中断对象。

一般情况下，设备驱动程序被卸载或者中断向量被修改需要重新连接时，调用该函数。

8.5 系统时钟中断

8.5.1 系统时钟中断概述

系统时钟中断 (简称时钟中断) 是操作系统最重要的中断，是分时机制实现的基础，操作系统内核依靠时钟中断完成时间片计算和分配、定时等管理工作。可以说，没有时钟中断，操作系统就无法正常运行。时钟中断由专门的时钟芯片产生，如 PC 上的 8253 芯片。大多数的操作系统实现，时钟中断周期会维持在 10~100ms 之间，如 Windows 操作系统，其时钟中断周期一般为 10ms 或者 20ms。Hello China 的实现中，时钟中断周期是一个预定义的



宏，可通过修改这个宏，然后重新编译内核，来调整系统时钟中断的周期。当然，一旦系统时钟中断的宏定义被修改，对应硬件平台上的时钟芯片初始化代码也要做相应修改，使硬件产生的中断周期与操作系统内核定义的周期匹配。在 Hello China PC 版本的实现中，时钟中断采用了 BIOS 初始化的时钟中断周期，即大约 55ms 左右。

在接下来的内容中，将详细介绍 Hello China PC 版的时钟中断初始化方法和中断处理函数的主要工作。时钟中断处理动作是操作系统最核心的系统功能之一，可以说，理解了系统中断的处理机制，对操作系统核心的工作原理，就理解了至少一半。

8.5.2 系统时钟中断的初始化

在 Hello China V1.75 版本的实现中，并未对计算机的系统定时芯片（8253 芯片）做特殊初始化，而是直接采用了 BIOS 的初始化工作方式，即周期定时方式，定时周期为 55ms。这样的实现方式比较简单，省略了硬件初始化代码（实际上即使重新初始化，也不会超过 10 行汇编代码），且至今尚未发现不妥的地方。可能有的读者会认为系统中断周期太长，对操作系统的性能和实时性产生影响。作者对这个问题进行过比较长时间的思考，发现并非如此。具体分析过程请参考 8.5.4 节。

时钟中断的硬件中断向量（即系统时钟芯片连接的中断控制器 8259 的引脚号）为 0，即中断控制器的第一个中断输入引脚。但在切换到保护模式后，x86 CPU 把从 0 开始的 32 个连续中断向量保留为系统异常使用，硬件中断向量号是从 32 开始的。因此在切换到保护模式后，时钟中断的中断向量是 32。即 CPU 会从 IDT 的第 32 个表项中，找到中断处理程序的入口地址，然后调用该中断处理程序。在初始化 IDT 的时候，第 32 个中断描述符的中断处理程序被填写为 np_int20，这是一个汇编函数的入口标号，下面是该汇编函数的源代码（为了描述方便，源代码做了简单修改，主要是删除了部分注释及参数安全检查代码）：

```
[kernel/arch/sysinit/miniker.asm]
np_int20:
    push eax
    .....
    push ebx
    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
out 0x20,al
out 0xa0,al
pop eax
iret
```

这就是时钟中断处理程序的汇编语言处理部分。这几行汇编语言的功能比较简单明了，无非是保存通用寄存器，然后调用通用中断处理函数（`gl_general_int_handler`）。需要注意的是，在调用通用中断处理函数前，首先把中断向量（这里为 `0x20`，即十进制的 32）压入堆栈，告诉通用中断处理函数，当前的中断向量是 32，即系统时钟中断。

本章曾解释过，`gl_general_int_handler` 实际上是定义的一个 4 字节全局变量，并不是真正的通用中断处理函数本身。这个全局变量存储了通用中断处理函数的函数地址，这是在操作系统初始化的时候，由操作系统初始化代码填写进去的。通用中断处理函数是用 C 语言实现的，其代码比较简单，无非是进一步调用了 `DispatchInterrupt` 函数。`DispatchInterrupt` 函数根据中断向量，索引全局中断对象表（参考 8.3.1 节），找到系统时钟中断对象，然后调用中断处理函数。

8.5.3 系统时钟中断处理函数的主要工作

时钟中断的所有处理工作，都是在时钟中断处理函数内完成的。目前版本的实现中，时钟中断函数主要完成下列两项主要工作：

1) 处理定时器队列。中断处理函数会检查系统中的所有定时器对象，一旦发现有定时器对象超时，则会处理该超时的定时器对象。处理方式包括调用定时器超时函数、向定时器的归属线程发送消息等。更详细的信息，请参考 8.9 节。

2) 唤醒睡眠状态的核心线程。核心线程通过调用 `Sleep` 函数进入睡眠状态，`Sleep` 函数会指定一个睡眠时间。一旦该睡眠时间到达，线程就会被重新唤醒。唤醒线程的操作，也是在时钟中断处理函数内完成的。

定时器队列的处理，将在 8.9 节中做详细介绍。在本节中，我们查看唤醒核心线程的操作代码。

下面是时钟中断处理函数中唤醒睡眠线程的实现代码，为了解释方便，删除了一些无关代码，比如参数安全检查代码、代码注释、安全检查的条件分支代码等。这些代码的删除，不会影响我们对这个函数的解释。

```
[kernel/kernel/system.cpp]
static BOOL TimerInterruptHandler(LPVOID lpEsp,LPVOID)
{
    DWORD dwPriority = 0L;
```

```

__TIMER_OBJECT*      lpTimerObject      = 0L;
__KERNEL_THREAD_MESSAGE Msg              ;
__PRIORITY_QUEUE*     lpTimerQueue       = NULL;
__PRIORITY_QUEUE*     lpSleepingQueue    = NULL;
__KERNEL_THREAD_OBJECT* lpKernelThread   = NULL;
DWORD                 dwFlags            = 0L;

```

```
/*
```

```
// 定时器队列处理代码，本部分代码将在 8.9 节中做详细介绍
```

```
// .....
```

```
*/
```

/* dwClockTickCounter 就是当前的时钟中断计数，即系统的时钟 tick 计数，dwNextWakeupTick 则是最近待唤醒的核心线程的唤醒时机。核心线程通过调用 Sleep 函数进入睡眠状态。Sleep 函数会根据睡眠的时间（Sleep 函数的参数，以毫秒计），计算出线程需要被唤醒的时钟周期数（tick 数），然后把这个数存放在 KernelThreadManager 对象维持的一个全局变量中。在每个时钟中断的处理函数中，都会检查这个待唤醒的时钟 tick，是否与当前时钟 tick 相同。如果相同，则说明有线程需要被唤醒。*/

```

if(System.dwClockTickCounter == KernelThreadManager.dwNextWakeupTick)
{
    lpSleepingQueue = KernelThreadManager.lpSleepingQueue;
    lpKernelThread = lpSleepingQueue->GetHeaderElement(
        (__COMMON_OBJECT*)lpSleepingQueue,
        &dwPriority);
    while(lpKernelThread)
    {
        dwPriority = MAX_DWORD_VALUE - dwPriority;
        if(dwPriority > System.dwClockTickCounter)
            break; //This kernel thread should not be wake up.
        lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_READY;
        KernelThreadManager.AddReadyKernelThread(
            (__COMMON_OBJECT*)&KernelThreadManager,
            lpKernelThread); //Insert the waked up kernel thread into ready queue.

        lpKernelThread = lpSleepingQueue->GetHeaderElement(
            (__COMMON_OBJECT*)lpSleepingQueue,
            &dwPriority); //Check next kernel thread in sleeping queue.
    }
}

```

/* 上面这个循环的功能是，依次从睡眠队列中获取核心线程，并再次检查该线程是否应该被唤醒。需要注意的是，睡眠队列中有很多核心线程，这些核心线程按照被唤醒的先后顺序排序，处于队头的线程，应该首先被唤醒。如果发现核心线程需要被唤醒，则时钟中断函数就会修改线程的状态为 READY，然后插入就绪队列。这样在下一个调度时机，核心线程就可能被调度执行。

如果发现一个线程的唤醒时机尚未到来，则结束这个循环。睡眠队列是由 KernelThreadManager 对象维护的一个全局队列，存放了所有处于睡眠状态的核心线程。*/

/* 下面的代码，更新了下一个需要唤醒的核心线程的唤醒时机。在上面的代码中，如果发现从睡眠队列中提取的一个核心线程的唤醒时机尚未到来，则仅仅是退出上面的循环。由于睡眠状态的核心线程是按照被唤醒的时机进行先后排序的，因此导致上述循环退出的核心线程（睡眠队列为空的情况不做考虑），将是下一个被唤醒的核心线程。这时候用这个线程的唤醒时机（时钟 tick）更新 dwNextWakeupTick 变量。

当然，如果睡眠队列中的所有核心线程都被唤醒（队列为空），则复位 dwNextWakeupTick 变量。*/

```
if(NULL == lpKernelThread)
{
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    KernelThreadManager.dwNextWakeupTick = 0L;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
}
else
{
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    KernelThreadManager.dwNextWakeupTick = dwPriority;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    dwPriority = MAX_DWORD_VALUE - dwPriority;
    lpSleepingQueue->InsertIntoQueue((__COMMON_OBJECT*)lpSleepingQueue,
        (__COMMON_OBJECT*)lpKernelThread,
        dwPriority);
}
}
}
.....
return TRUE;
}
```

上述代码主要是对睡眠状态的核心线程进行处理，虽然代码相对比较多，但是整体逻辑比较简单，无非是判断线程是否需要被唤醒，如果是，则修改其状态，然后加入就绪队列。

同时也可以得到结论：线程的睡眠时间（即 Sleep 函数的参数），必须是时钟周期的整数倍，因为睡眠状态的核心线程，只有在时钟中断处理程序中，才会被唤醒。当然，在调用 Sleep 函数的时候，可以指定任何参数，但 Sleep 函数会自动把参数向上舍入到系统时钟周期的整数倍。

8.5.4 时钟中断周期对系统实时性的影响分析

表面上看，似乎时钟中断周期越短，系统的实时性越好，因为进程或线程的运行时间片会被控制得越精确，优先级高的进程或线程会优先运行。但仔细分析，会发现并非如此，时钟中断周期的大小与系统整体实时性关系并不十分紧密。

可用两个指标来衡量操作系统的实时性：一个是中断响应时间，即从外部中断发生，到得到操作系统处理之间的时间；另外一个为任务切入时间，即一个高优先级的线程运行所需的资源就绪，到得到调度所需的时间。时钟中断周期的大小与这两个指标并无直接关联。

首先看中断响应时间，这个时间与硬件系统关联紧密。中断一般由外部设备引发，外部设备的控制电路连接到计算机的中断控制器上（比如 PC 的 8259A 芯片）。一旦外部设备发生中断，设备会通过一条中断引脚通知中断控制器，中断控制器根据输入引脚的状态（比如是否禁止引发中断）、输入引脚的优先级、连接到片上的其他中断引脚的情况，综合判断是否需要对该中断进行处理。如果判断结果为进一步处理，则通过 CPU 的中断输入（比如 x86 CPU 的 INTR 信号）通知 CPU。CPU 并不会马上处理中断，而是有一套比较复杂的判断机制，判断是否对刚刚输入的中断进行响应。最重要的决定因素有两个：一个是 CPU 只会在

一条指令执行完毕才会检查中断状态，如果 CPU 正处于一条指令的执行过程中（需要注意的是，CPU 执行一条指令，有时候会需要很多个 CPU 节拍），中断是不会被处理的。另外一个因素是中断使能标志是否被清除（比如 x86 CPU 标志寄存器中的中断使能位）。如果使能标志被清除，则 CPU 也不会响应中断。只有 CPU 的中断使能标志位被设置、同时 CPU 处于指令完毕状态时，外部中断才会得到处理。一旦 CPU 响应外部中断，它会根据中断号（在 x86CPU 上，需要从中断控制器中读入），从中断向量表中选择一个中断向量，然后调用该中断向量对应的中断处理函数，这时候才正式进入操作系统的处理范围。

可见，在这个复杂的判定处理过程中，并未涉及时钟中断周期，即时钟中断周期与中断响应时间无关。影响中断响应时间的，是中断控制器本身处理、CPU 的最大指令处理时间（处理一条指令的最大时间）、操作系统是否频繁清除中断使能标志等因素。为了尽可能缩短中断响应时间，操作系统在实现的时候，需要尽可能避免清除中断使能标志。即使无法避免，也应尽可能缩短中断关闭时间。

再考察任务切入时间。响应时间快的操作系统，在一个高优先级的线程或任务的运行资源准备就绪后，应该马上调度该线程进入运行状态，以快速处理高优先级的事件，除非系统中有更高优先级的线程或任务在运行。线程或任务的运行资源，一般是由外部中断控制的，比如接收到一个外部输入事件，这个输入事件会被调度到一个专用的处理线程进行处理。这时候，一旦外部中断处理结束，CPU 会重新调度系统中的所有线程或任务。这样处理外部事件的高优先级线程会立即得到调度，而不用等时钟中断的发生。因此这种情况下，时钟中断周期的大小，对切入时间是没有影响的。另外一种可能是，线程运行所需要的资源，是由另外的线程产生的。比如一个高优先级的线程等待一个事件对象，而该事件对象由另外的核心线程所控制。这种情况下，一旦另外的核心线程释放事件对象，等待该事件的高优先级线程也会马上得到调度，因为释放事件对象的过程是一个系统调用，在系统调用结束后，一般的操作系统也会重新调度系统中的所有线程。当然，这里的前提是在系统调用结束后，操作系统会立即重新调度所有核心线程，而不用等到时钟中断发生。有的操作系统的实现机制是，在系统调用结束后，并不会马上重新调度核心线程，而必须等到时钟中断发生后才调度。这种机制下，时钟中断的周期大小，就与整体响应时间有非常紧密的关系了。

从上面的分析来看，在实现了抢占式调度机制，同时实现了系统调用结束后即可重新调度所有线程的操作系统，其时钟中断周期与线程切入时间也是无关的。

系统时钟中断周期短的一个好处是，可以把定时器精度做得很高，比如时钟中断周期为 10ms，则可以实现 10ms 级别的定时器。而如果系统时钟中断周期为 100ms，则如果不借助于其他外部定时机制，定时器功能最多能实现 100ms 的精度范围。但在大多数情况下，定时器功能的精度要求并不需太高，因为定时器功能往往用在诸如网络协议等的实现中，这些功能并不需要精度非常高的定时机制。在精度要求非常高的场合，定时器是不建议使用的，因为定时器要借助于系统时钟控制芯片，在中断被关闭的情况下，可能会造成较大的定时误差，比如误差达到毫秒级，即使时钟中断周期非常短。这在实时性要求非常严格的系统中，是无法接受的。

综合上述分析可见，系统时钟中断周期的长短，与操作系统的实时性无必然联系。时钟中断周期长的操作系统，其实时性不一定会差，只要它尽可能减少中断关闭时间，同时在系统调用和外部中断结束后立即重新调度线程。相反，时钟中断周期短的操作系统的实时性，如果没有

实现抢占式调度机制，其实时性也可能很差。虽然在时钟中断周期短的情况下，定时功能会精确一些，但现实意义也不是非常大。而且在电源电量受限的嵌入式应用场景下，时钟中断太频繁，还会导致电量消耗严重。这时候为了省电，还不如把系统时钟中断周期设置得较长，以降低整体资源消耗。

8.6 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，可以很方便地相互移植。
- (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，另外有其他的中断发生，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，才会被响应。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。

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得知），如果不是自己的，需要尽快返回 `FALSE`。
- (4) 在中断处理过程中，建议不要显式地打开中断，即显式地插入“`sti`”等指令，或调用包含“`sti`”指令的函数。

8.7 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。当前版本的 Hello China 提供了三个应用编程接口：

- (1) `SetTimer`，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。

8.7.1 SetTimer 函数的调用

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，如果用户设定定时器时，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 则是该函数的参量；如果用户设定定时器时，没有指定该参数，则当定时器超时的时候，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 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，则设置了一个一次定时器，如果 dwTimerFlags 设置为 TIMER_FLAGS_ALWAYS，则设定了一个永久定时器。

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；另外一种方式是调用 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；如果用户调用 SetTimer 的时候，指定 DirectHandler 为 NULL，则超时，直接给用户线程发送一个消息。下面代码是 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)。

8.7.2 CancelTimer 函数的调用

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。但有时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。

8.7.3 ResetTimer 函数的调用

该函数用来复位尚未超时的定时器对象, 原型如下:

```
VOID ResetTimer(_COMMON_OBJECT* lpTimerObject);
```

其中, lpTimerObject 是待复位的定时器对象的指针 (由 SetTimer 调用返回), 复位定时器对象后, 定时器对象将重新开始计时。这项功能在网络协议的处理中十分有用, 比如一个协议实体等待接收一个网络报文, 为了实现容错 (考虑网络中断的情况) 设定一个定时器, 如果在定时器超时后, 仍然没有收到报文, 则会进行进一步的动作, 如果在超时前收到了报文, 则需要重新复位定时器, 从头开始计时。

需要注意的是, 该函数只能用来操作永久定时器, 如果用来操作一次定时器, 会存在风险, 因为定时器对象可能已经被删除 (超时后)。

8.8 设置定时器操作

在当前版本 Hello China 的实现中, 定时器服务接口集成在 System 对象里面 (另外与中断有关的操作等, 也是作为 System 对象接口来提供), 这样主要是考虑到实现上的方便。在 System 的定义中, 定义了一个定时器对象队列用来维护定时器对象, 并定义了三个定时器对象操作接口。下面是 System 对象定义的相关部分代码。

```
BEGIN_DEFINE_OBJECT(_SYSTEM)
... ..
__PRIORITY_QUEUE* lpTimeQueue;
... ..
__COMMON_OBJECT* (*SetTimer)(_COMMON_OBJECT* lpSystem,
                              __COMMON_OBJECT* lpKernelThread,
```



```
        DWORD          dwTimerID,  
        DWORD          dwTimeSpan,  
        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，省略前两个参数 lpSystem 和 lpKernelThread，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，调用该函数的当前线程为第二个参数）。

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说明一下。

第一个问题是，如何确定该定时器对象的超时时刻。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 dwClockTickCounter 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 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（以毫秒为单位，当前定义为 55）。上述处理中，首先把 dwTimeSpan（定时器设定的等待时间，单位为毫秒）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，再与 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，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，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），所以，为了把最先超时的定时器对象插入优先队列的最前面，可采用下列方式：

```
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）。

8.9 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 8.7.1 节），然后与当前 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：

判断回调函数（DirectHandler）是否为空，如果不为空，则以 lpHandlerParam 为参数调用该回调函数）。

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 函数操作相同）。

（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。

当定时器对象的超时时刻（以 tick 计）不等于 dwNextTimerTick 时，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 8.7.1 节），并判断提取的定时器对象的超时时刻是否与 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);
}
}

```

8.10 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。

代码如下所示。

```

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->lpTimerQueue,
    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 when 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;
}
```

上面的处理首先从定时器对象队列中删除要取消的定时器对象，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，然后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`），则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。

8.11 定时器复位

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。

8.12 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时，需要遵循下列规则。

(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。图 8-6 是在一个 Pentium 4 2.5GHz 处理器上做的测试结果。

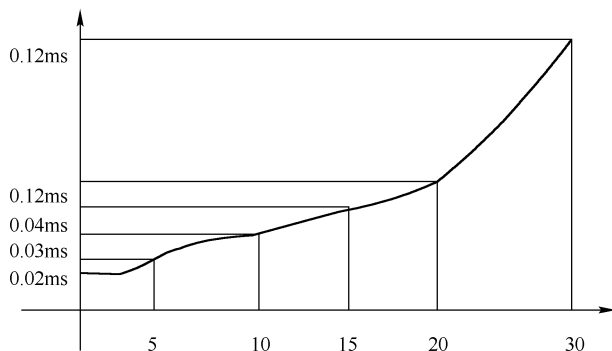


图 8-6 一个测试结果

图中，横坐标是一次定时器个数，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。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) 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 T_1 时刻超时，则在 T_1 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 T_2 ，如图 8-7 所示。

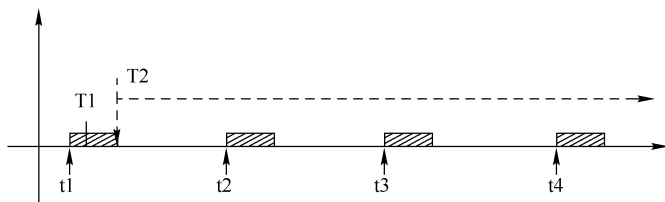


图 8-7 定时器消息的处理延时



图中， t_N ($t_1, t_2, t_3, \dots$) 是时钟中断发生时刻，且 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，系统会选择线程就绪队列中优先级最高的线程投入运行。此时如果设定定时器的线程是优先级最高的线程，那么会马上投入运行，定时器超时消息会被处理，因此，图中的 T_2 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，其他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。所以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 T_1 时刻，回调函数就可以被调用，即超时消息马上被处理。因此，在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 9 章 系统总线和设备管理

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，所有外部设备都连接在系统总线上，甚至可以把 CPU、内存等部件也看作系统总线上的“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物理设备的管理十分简便，也使驱动程序的编写和调试十分简便，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系统无法扩展，无法支持更多的物理设备，也使设备驱动程序编写工作十分繁杂，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地容纳新的设备，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。本章对 Hello China 的总线管理框架（模型）进行描述，并详细介绍 PCI 总线驱动程序的实现。

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，系统中只存在一个，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，用来描述任意系统总线（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，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，把总线也当做设备来看待。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 `ISA` 和 `PCI` 两种总线的驱动程序，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，在这种情况下，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）。

各种总线驱动程序所完成的工作主要检测总线是否存在，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一个抽象的数据结构 `_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 `lpHomeBridge`，用来表明该总线连接到的桥设备。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，将建立如图 9-1 所示的数据结构。

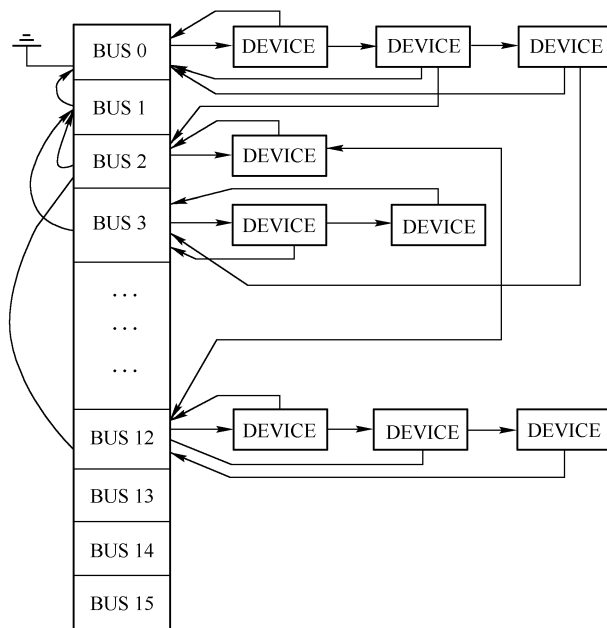


图 9-1 Hello China 的系统总线管理结构

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, 第一个桥接设备下一级总线 (lpChildBus) 是总线 2 (BUS2), 而第二个桥接设备下一级总线是总线 3。在总线 2 (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) 静态地添加到系统总线上。

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, 代码如下:

```
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 PowerPC, 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 0~65 535 (16bit 范围内), 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（Virtual MemoryMgr）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，则该函数会返回 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联在一起。

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联在一个双向链表中。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，这个数组的元素个数固定为

MAX_RESOURCE_NUM (当前定义为 7)，这样如果设备所占用的资源少于该数值，剩余 Resource 数组的元素的 dwResType 就设置为 RESOURCE_TYPE_EMPTY。比如，一个物理设备仅占用了端口范围，一个中断向量号，则 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 5 个元素的 dwResType 标志都设置为 RESOURCE_TYPE_EMPTY。

9.2.2 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 则把当前空闲的资源连接成一个双向链表，如图 9-2 所示。

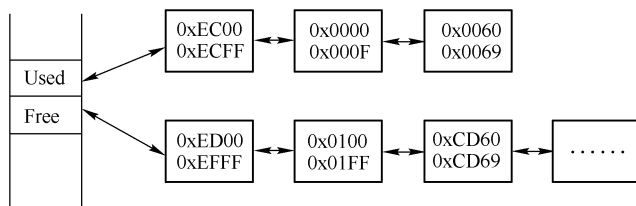


图 9-2 Hello China 的端口资源管理

UsedPortResource 链表起始为空，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 0x0000~0x000F、0x0010~0x001F 和 0x0020~0x01FF 三部分，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 驱动程序接口

DeviceManager 对象提供了下列接口为驱动程序调用 (这里的驱动程序，是普通的设备驱动程序，不是总线驱动程序。在 Hello China 当前版本的实现中，总线驱动程序作为内核的一部分实现，其结构不遵循普通的设备驱动程序的体系结构)。



9.3.1 GetResource

GetResource 函数供设备驱动程序调用，用来请求设备驱动程序所服务的目标设备所占用的资源。一般情况下，对于 PCI、USB 等总线，总线驱动程序事先完成了总线上所有设备的枚举以及资源分配，这样当这类接口设备的设备驱动程序加载后，设备驱动程序不需要自己分配资源，只需要根据设备标识（__IDENTIFIER）向 DeviceManager 请求对应的资源即可。

该函数原型如下：

```
__RESOURCE* GetResource(__DEVICE_MANAGER* lpThis,
                        DWORD dwBusType,
                        DWORD dwResType,
                        __IDENTIFIER* lpIdentifier);
```

各参数的含义十分明确，最后一个参数 lpIdentifier 是物理设备的标识符。该函数返回相应设备的资源数组（__PHYSICAL_DEVICE 对象的 Resource 数组）的地址（第一个元素的地址）。

9.3.2 GetDevice

与 GetResource 类似，该函数被设备驱动程序调用，用来获得相应的物理设备对象。原型如下：

```
__PHYSICAL_DEVICE* (*GetDevice)(__DEVICE_MANAGER*,
                                DWORD dwBusType,
                                __IDENTIFIER* lpIdentifier,
                                __PHYSICAL_DEVICE* lpStart);
```

9.3.3 CheckPortRegion

CheckPortRegion 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。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，ReservePortRegion 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。

另外, 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或本地变量中), 必须调用 `KMemFree` 函数销毁该对象。

9.3.5 ReleasePortRegion

`ReleasePortRegion` 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, 需要设备驱动程序静态地检测设备存在, 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时必须调用该函数, 向 `DeviceManager` 注册相应的设备。这样做的目的是使 `DeviceManager` 能够统一地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并初始化的一个物理设备对象。在设备驱动程序存在的时间内 (设备驱动程序被卸载前), 必须始终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, 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

`DeleteDevice` 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。图 9-3 是一个典型的 PCI 系统框图。

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 IO 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，俗称“北桥”，而 PCI 总线上连接 ISA 总线的桥路相应地称为“南桥”。

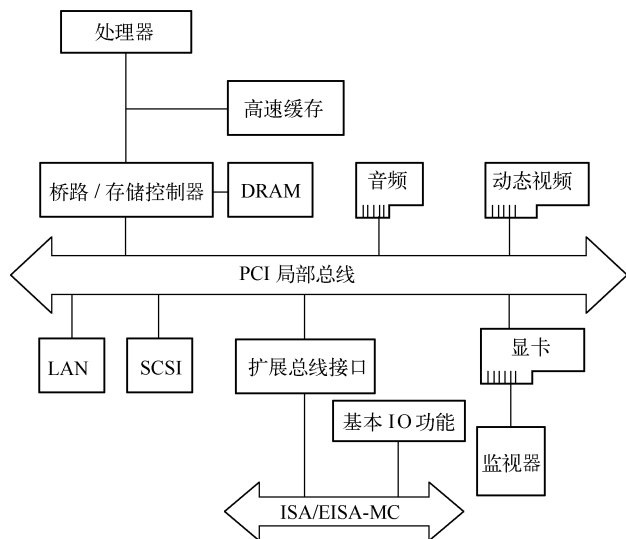


图 9-3 一个典型的基于 PCI 总线的计算机结构

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 256B 的配置空间，这个空间分为 64B 的预定义首部和 192B 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 64B 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 64B 首部结构如图 9-4 所示。

31		16		15		0		
Device ID				Vendor ID				00H
Status				Command				04H
Class Code						Revision ID		08H
BIST		Header Type		Latency Timer		Cache Line Sz		0CH
Base Address Register 1								10H
Base Address Register 2								14H
Base Address Register 3								18H
Base Address Register 4								1CH
Base Address Register 5								20H
Base Address Register 6								24H
CardBus CIS Pointer								28H
System ID				Subsystem ID				2CH
Expansion ROM base address								30H
						Capabilities ptr		34H
								38H
Max_Lat		Min_Gnt		Interrupt Pin		Interrupt Line		3CH

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），预定义的首部结构如图 9-5 所示。

31				16		15		0	
Device ID				Vendor ID				00H	
Status				Command				04H	
Class Code						Revision ID		08H	
BIST		Header Type						0CH	
Base Address Register 1								10H	
Base Address Register 2								14H	
Latency		Subordinate		Secondary		Primary		18H	
Secondary status				IO Limit		IO Base		1CH	
Memory Limit				Memory Base				20H	
Prefetch memory Limit				Prefetch memory Base				24H	
Prefetch memory base upper 32								28H	
Prefetch memory limit upper 32								2CH	
IO limit upper 16				IO Base upper 16				30H	
						Capabilities ptr		34H	
								38H	
Max_Lat		Min_Gnt		Interrupt Pin		Interrupt Line		3CH	

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，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，以及本书没有提到的字段含义，请参考 PCI 总线规范。

1. Vendor ID 和 Device ID

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。例如，假设总线类型为 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（dwClass 成员，实际上只有高 24bit 有效，最低的 8bit 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，例如，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，其他字段不需要给出。因此，ucMask 字段指明了 PCI_Identifier 结构中哪个字段有效。例如，假设 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。

2. Class code

Class code 是设备类代码，用来描述设备的功能。这个字段长 24bit，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, 粗略地描述了一个 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。

3. Header type

Header type 是头部类型。根据 PCI 规范的定义, 对所有 PCI 设备保留了 256B 的配置空间, 其中前 64B 是预定义的, 后续 192B 则根据设备的具体情况具体实现。对于预先定义的 64B 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书中已经给出。

(2) PCI-PCI 桥接设备 (PCI-PCI 桥), 这类设备的头部组织在本书中已经给出。

(3) PCI-CardBus 桥接设备 (PCI-CardBus 桥), 这类设备的组织本书中没有给出。

上述三种情形预定义头部开始的 16B 的组织结构都是一样的, 从 16B 往后 (17~64B), 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 7 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)。0~6bit 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。

4. Base Address Register

PCI 接口的硬件设备有一个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, 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 总线的设备在安装的时候必须静态地完成 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 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, 这样对设备的后续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 Address Register, 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，此时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 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 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 必须符合如图 9-6 所示的结构。

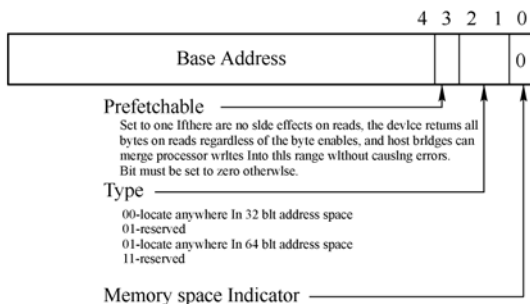


Figure 6-5: Base Address Register for Memory

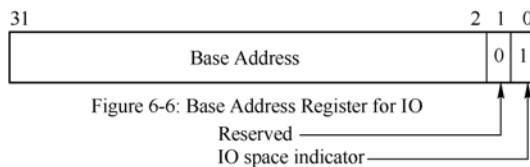


Figure 6-6: Base Address Register for IO

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。表 9-1 是这三个比特的取值，以及对应的含义。

表 9-1 三个控制比特的含义

取 值	含 义
000	该寄存器映射到 32bit 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 64bit 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 32bit 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 64bit 的内存空间内，而且映射的空间范围是可预取的
111	保留

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 512KB 到 2MB 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），则就节省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（如 4KB）读入 CPU 的本地 cache，这样后续 CPU 如果再读取同样的内存单元，或者与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为其他的值。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，与在时刻 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 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与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，详细信息请参考第 5 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 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，其他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 32bit 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 16bit (16~31bit)。

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, 这样对设备的后续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, 一个用来指定 IO 端口范围, 另外一个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建议尽量采用内存区域对设备进行控制。

5. Interrupt Line 和 Interrupt Pin

Interrupt Line 和 Interrupt Pin 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 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 字段就指明了当前的 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 上, 这个字段的值可以是 0~15 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 和 INTD, PCI 设备的中断输入可以与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 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，则建议调用 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。

6. 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二级总线（Secondary BUS），这样 Primary 字段和 Secondary 字段就是主总线号和二级总线号。

但是 PCI-PCI 桥设备的二级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……这样不断连接，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，其二级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。图 9-7 是一个由两个 PCI-PCI 桥设备组成的典型的硬件配置结构，总共有三条 PCI 总线。

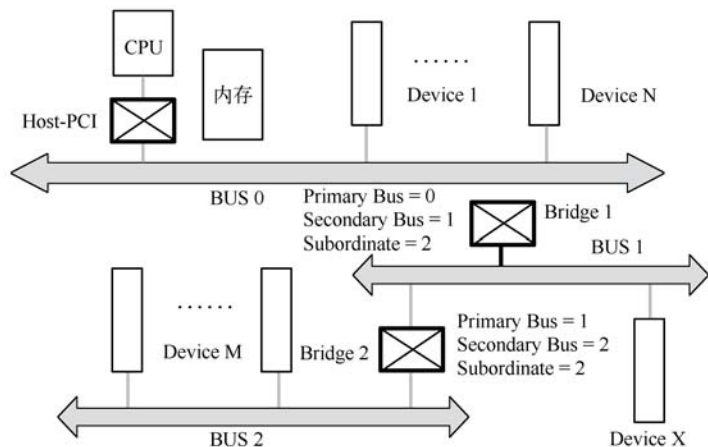


图 9-7 一个由三条 PCI 总线组成的总线结构

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，其二级总线号（Secondary 字段）就是 1，同样，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，就是其二级总线（Bridge 2 的 Secondary BUS 是 2）。由于 Bridge 2 的二级总线上没有连接其他的桥接设备，因此 Bridge 2 的 Subordinate 字段就是 2，同样，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 桥设备就把访问请求的目标总线号与 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 桥就不做任何处理。

这些字段都是在 PCI 总线初始化的时候填写的，在 PC 上，这个初始化操作由 BIOS 软件完成。为可靠起见，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 9.5 节。

7. IO Base 和 IO Limit

IO Base 和 IO Limit 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 256B 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 0x00。同样地，IO Limit 也是以 256B 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 256B。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 16bit 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 32bit 的，因此，1 型头部中的 IO Base Upper 16 字段和 IO Limit Upper 16 字段就作为 32bit IO 端口范围的扩展，与 IO Base 和 IO Limit 结合起来共同确定一个 32bit 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 32bit 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 32bit 端口范围的版本，会十分容易。

8. Memory Base 和 Memory Limit

Memory Base 和 Memory Limit 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。

9. Prefetch memory base 和 Prefetch memory limit

Prefetch memory base 和 Prefetch memory limit 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。

9.4.4 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 设备的配置空间时分两步进行。

(1) 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 32bit 的双字, 如图 9-8 所示。

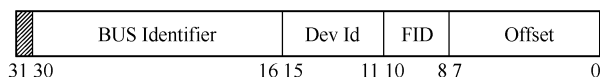


图 9-8 PCI 总线操作命令字结构

其中, 第 31bit 固定为 1, 16~30bit 是总线标识符, 11~15bit 则是设备号, 8~10bit 是功能号 (一个 PCI 设备, 最多可以支持 8 种不同的功能), 最后 8bit 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 (4B), 则首先需要向 CF8 端口输出 80014000 (十六进制)。

(2) 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, 可以这样做 (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, 可以这样进行:

```
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 32bit 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 4B（上面 Interrupt Line 只有 8bit），则首先读出 4B，然后修改要写入的部分，再把 4B 整体写入配置空间。

9.5 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，指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他总线类型，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，与操作系统一起加载。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。为了便于实现，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 对象是该函数的参数（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。

9.5.1 探测 PCI 总线是否存在

对于 PCI 总线的探测，目前做得十分简单。

- (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 Vendor 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，上述判断可以很好地工作，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 探测做得很简单的主要原因。

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;
}
```

9.5.2 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 设备进行枚举，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) 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，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;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    for(dwLoop = 0;dwLoop < MAX_BUS_NUM;dwLoop ++)
    {
```

```

        if(lpDeviceMgr->SystemBus[dwLoop].dwBusType == BUS_TYPE_NULL)
            break;
    }
    if(MAX_BUS_NUM == dwLoop)
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return MAX_DWORD_VALUE;
    }
    lpDeviceMgr->SystemBus[dwLoop].dwBusType = BUS_TYPE_PCI;
    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，定义了下列四个取值。

```

#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 设备进行枚举时，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），初始化之后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;

    lpPhyDev = KMemAlloc(KMEM_SIZE_TYPE_ANY, sizeof(__PHYSICAL_DEVICE));
    if(NULL == lpPhyDev)
        goto __TERMINAL;
    lpDevInfo = KMemAlloc(KMEM_SIZE_TYPE_ANY, sizeof(__PCI_DEVICE_INFO));
    if(NULL == lpDevInfo)
        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 &= 0xFFFFF00
    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 &= 0xFFFFF00;
    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 &= 0FFFFFFF0;
        PciFillDevResource(dwConfigReg,lpPhyDev);
        bResult = TRUE;
        break;
    case 1:    //The header type is 1.
        lpDevInfo->dwDeviceType = PCI_DEVICE_TYPE_BRIDGE;
        dwConfigReg& = 0FFFFFFF0;
        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），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 &= 0xFFFFF00;    //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 &= 0xFFFFF00;    //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 &= 0xFFFFF00;
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 读取一遍（按照本章概述中介绍的方法），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 设备大致相同，在此不作赘述，读者可通过阅读代码做深入了解。

对 PCI 总线上的设备枚举完毕，系统就建立了一棵设备树，所有 PCI 总线上的普通设备（非总线桥设备）都是这棵树的叶子节点，总线桥设备（比如 PCI-PCI 桥等）则是这棵树的分支。Hello China 提供了一个系统诊断程序 sysdiag，里面实现了一个 pcilist 命令，可以查看系统设备树（目前仅显示 PCI 总线和 PCI 设备，包含 PCI-PCI 桥设备）。图 9-9 是 pcilist 命令在 Virtual PC 上的运行截图。

```

#
#pcilist
Device ID/Vendor ID      Bus Number      Description
00000009/00001011        0                Ethernet controller.
00008811/00005333        0                VGA controller.
00007113/00008086        0                Other bridge device.
00000000/00000000        0                Old devices(no-PCI device).
00007111/0000B086        0                IDE controller.
00007110/0000B086        0                ISA bridge.
00007192/0000B086        0                PCI-Host bridge.
#

```

图 9-9 pcilist 命令在 Virtual PC 上的运行截图

可见，Virtual PC 模拟的计算机，只有一条 PCI 总线（Bus Number 为 0），连接了以太网卡、VGA 控制器、IDE 控制器等设备。

9.5.3 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，这几个字段如下。



(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，需要遍历当前总线上的所有 PCI 设备，并计算它们的并集。

(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，就已经做了配置（参考 PciScanBus 函数的实现）。

第 10 章 设备驱动程序管理

10.1 设备管理框架

10.1.1 概述

设备管理是操作系统最核心的管理任务之一。在一个典型的成熟操作系统中，设备管理部分的实现代码（含设备驱动程序和设备管理框架）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和重要。

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，即把特定设备的设备驱动程序（Driver）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。而具体设备的驱动程序，则由生产这种设备的厂商（或熟悉设备工作原理的第三方开发人员）实现。这样就实现了操作系统核心代码与设备驱动程序的逻辑分离，操作系统无需了解所有设备的工作原理，设备驱动程序只需按照操作系统的规范，实现相关接口函数即可，无需了解操作系统内核的实现机制。这种设计思想，最大可能地确保了操作系统的可扩展性。

因此，一般来说，操作系统面对的是设备驱动程序，由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对具体的硬件设备。要实现这个功能，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与操作系统交互，操作系统也根据自己提供的这个接口调用设备驱动程序的一些功能函数，来简单地操纵硬件。

另外，在设备管理过程当中，还有一些问题要解决，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，如果无法正常地命名设备，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与设备之间的桥梁。

（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（也有很多情况是采用轮询方式的，即操作系统主动查询设备的状态），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。大多数情况下，中断处理程序是在设备驱动程序中实现的，而中断的调度（根据中断向量号调用对应的中断处理程序）则是由操作系统内核完成的。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，即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。这种方式的优点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。其二，动态分配，操作系统核心在加载的时候，通过某种总线协议，动态地检测总线上的设备，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，第二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，比如 PCI 系列总线。但有些情况下，系统总线是不支持自动发现设备的，比如比较老的 ISA 总线。这时候就需要通过手工的方式，静态分配系统资源。

(4) 用户线程对设备的访问问题。用户如何通过一种统一的接口调用各种设备驱动程序提供的服务，也是操作系统的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来访问所有的设备。

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，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，对于设备管理部件的设计，也充分考虑了上述问题，并采取了合适手段对上述问题进行了解决。本章将对 Hello China 的设备管理模块进行详细描述，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。

本书把 Hello China 实现的设备管理体系称为“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由用户在设备驱动程序中实现的。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。

这个设备管理框架也是按照面向对象的思想设计的，概括来说，主要由以下几个全局对象组成：

1) System 对象。前面已经介绍过，这个对象包含了很多操作系统核心功能，比如定时服务、异常处理等。与设备驱动程序相关的，是中断调度部分。该对象提供了两个重要的函数——ConnectInterrupt 和 DisconnectInterrupt，管理中断向量和中断处理程序之间的关联。其中第一个函数把一个中断向量和一个中断处理程序关联起来。这样一旦对应的中断发生，中断处理程序即可被调用。这个函数一般在设备驱动程序初始化的时候调用。后者则取消这种关联，在设备驱动程序被卸载的时候调用。这个对象在前面的章节中已有详细介绍，在此不作进一步讲解。在本章后面的一个示例中，读者将看到这两个函数的使用方法。

2) DeviceManager 对象。这个对象实现了系统总线的管理、物理设备的硬件资源管理等功能。所有硬件资源的分配，包括动态配置和静态分配，都是由这个对象完成的。设备驱动程序可根据设备 ID 来调用 GetDevice 等函数，来获取特定设备的硬件资源配置信息，包括分配的中断向量号、IO 端口资源、内存映射区域资源等。当然，前提是设备所在的总线能够支持自动配置功能。这个对象在第 9 章中做了详细介绍，不再赘述。

3) IOManager 对象，即输入/输出管理器对象。这个对象是整个设备管理框架的最核心的对象，它实现了设备驱动程序的管理、设备对象的管理、文件系统的管理、上层应用程序的接口实现等功能。本章将重点介绍这个对象的设备驱动程序和设备对象管理功能，而文件

系统等功能，将在第 12 章中详细介绍。

这几个对象相互协调，共同实现了 Hello China 操作系统的设备管理框架。接下来将重点介绍 IOManager 对象。

10.1.2 通用的操作系统设备管理机制

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，从而可以更好地理解 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（前面讲过，在 Hello China 的实现中，这个数据库是由 DeviceManager 全局对象管理和维护的）。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，是在操作系统启动过程中进行的。在操作系统启动的过程中，首先从系统总线开始探测，比如针对 PCI 总线，操作系统引导代码读取适当的端口（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 总线上所有连接设备的枚举和检测。当然，对于其他支持自动配置的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。再以 PCI 总线为例，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，请参考第 9 章），然后针对每个系统中存在的设备，在硬件信息数据库中创建相应的对象（数据结构），并使用读取的配置信息填充这个对象。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，操作系统收集了所有的系统硬件信息，并存放到一个统一的数据库中进行管理。比如，图 10-1 表示了一个典型的操作系统的硬件信息数据库。

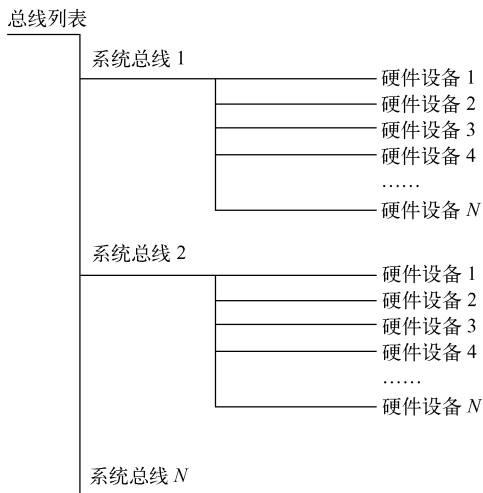


图 10-1 操作系统维护的硬件信息数据库



这样，如果要标识一个物理设备，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 ID 来标识具体的设备。

这个硬件信息数据库会根据操作系统实现的不同而位于不同的位置。例如，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，如中断向量号、内存映射空间、IO 端口地址、DMA 通道等，这一步称为设备配置。由于这些资源信息都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，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，而具体的端口号则尚未确定。因此，操作系统需要为该设备分配一段连续的 IO 端口资源，并写入设备的配置寄存器，这就是设备配置的工作。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 为硬件分配的资源会不会出现冲突。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，进入正式使用步骤之前，操作系统必须加载对应的设备驱动程序。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，就可以得到物理设备的设备 ID（针对不同的总线类型，设备 ID 的标识方式也不一样），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。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提供给操作系统核心。

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 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。这种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，给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与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，下面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设备的时候，就可以通过设备字符串很容易地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，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调用 SendPacket 函数。

图 10-2 示例了设备标识字符串数据库的格式。

上述数据库（链表）中的每个元素（结构）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。为了解决这个问题，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，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/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，只需要提供有限的与抽象操作对应的函数即可。比如，提供给用户一个 Read 和一个 Write 函数，这两个函数与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普通文件的读/写操作，也可以完成设备的读/写操作。实际上，在物理设备上（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。

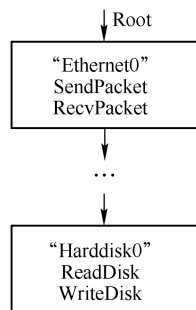


图 10-2 设备标识字符串数据库格式

到此为止，用户就可以很容易地访问硬件设备了，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。

但细心的读者可能发现，任何针对设备的操作，如果按照上述形式，则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，或者其他可以快速检索到设备对象的数据，后续操作（比如 `Read`、`Write` 等）则不必提供设备标识字符串，只需直接使用 `Open` 函数返回的句柄即可。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了一定了解，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函数。图 10-3 是设备对象数据库的一个更详尽的示例。

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

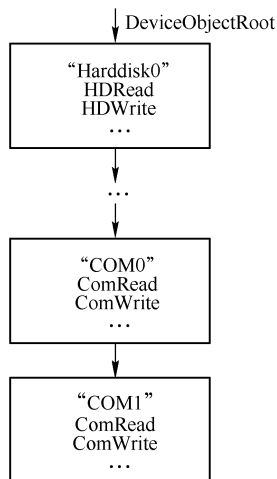


图 10-3 一个设备对象数据库的例子

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定义的一致; 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的, 用于维护系统中的硬件配置信息以及硬件物理参数。再强调一下, 在 Hello China 的实现中, 这个硬件设备的配置信息数据库是由设备管理器 (DeviceManager) 对象维护的。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与其操作方法 (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。这个设备对象数据库 (或设备对象链表), 在 Hello China 的实现中, 是在 IOManager 中实现的。这样读者就可进一步理解 DeviceManager 和 IOManager 的区别了。

下面对 IOManager 做一个简单的介绍, 其详细实现, 本章后续内容会陆续介绍。

IO 管理器用于管理设备对象数据库 (设备对象列表) 和文件系统, 并提供一组统一的函数 (方法) 呈现给上层应用程序, 供上层应用程序访问设备对象列表和文件系统。对设备对象的管理, 主要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 完成。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 每加载一个设备驱动程序, 都需要创建一个驱动程序对象 (__DRIVER_OBJECT) 并初始化, 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), 针对自己支持的每个设备, 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。

另外, IO 管理器对象也提供了应用程序调用的标准接口, 比如 Read、Write 等, 这些函数与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。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。这样就实现了用户调用到具体设备驱动程序的透明传递。可见, 操作系统是物理设备和用户应用程序之间的“桥梁”。

10.1.3 设备管理框架的实现

在 Hello China 的实现中, 设备的管理是按照下列策略来实现的:

(1) 采用一个统一的 IO 管理器对象 (IOManager) 来集中管理所有的设备对象、设备驱动程序等, 实现设备驱动程序的加载和卸载。

(2) 在 IOManager 对象和用户线程之间也定义了一个良好的交互接口, 用户线程直接调用 IOManager 的用户侧接口, 请求设备提供的服务。

(3) 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。

(4) 文件系统在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。

(5) 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的功能函数 (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。也需要充分考虑可能面临多个线程同时访问的问题, 通过操作系统核心提供的同步或互斥机制来保护设备相关数据, 充分保证代码的可重入性。

(6) 实现设备的动态发现和枚举, 比如, 针对 PCI 总线, 总线驱动程序可以动态地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), 并可动态加载这些设备的驱动程序。

(7) 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, 操作系统会及时做出响应, 比如分配资源、加载驱动等, 对于从总线上实时拆离的设备, 操作系统也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。

图 10-4 示意了 Hello China 设备管理框架的大致架构。

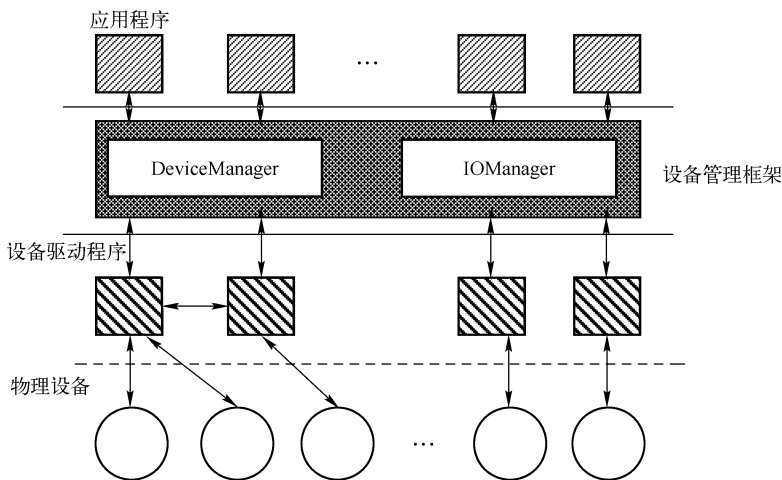


图 10-4 Hello China 的设备管理框架

可以看出, 在整个设备管理框架中, DeviceManager 和 IOManager 是核心部件。IOManager 提供了两个规范的接口, 对于用户核心线程的接口称为用户接口 (或上行接口), 对于设备驱动程序称为设备接口 (或下行接口), 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调用 IOManager 提供的设备管

理服务，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。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，对实际的物理设备进行操作。对文件系统的实现，可参考第 12 章。

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，可以以中断向量号和对应的中断处理程序（一个函数指针）为参数调用 ConnectInterrupt 函数（该函数由操作系统核心提供），注册自己的中断处理函数。当然，在设备卸载的时候，驱动程序也需要调用 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）。另外之所以把设备和设备驱动程序之间的交互关系表示为双向，是因为设备可能通过中断的方式主动与设备驱动程序交互。

下面将详细介绍该管理框架中涉及的模块以及模块之间的接口。

10.1.4 IO 管理器

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。还提供了面向设备驱动程序的接口，供设备驱动程序调用，完成诸如创建设备对象、销毁设备对象等操作。

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，可以认为该对象是设备管理框架的核心对象。

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，系统都为之创建了一个驱动程序对象，并调用驱动程序的 DriverEntry 函数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指针，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。设备的所有操作都是由驱动程序对象提供的函数完成的，通过指向驱动程序对象的指针，即可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，下面举一个磁盘驱动程序的例子。

（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。其中驱动程序的加载工作，是由 IOManager 完成的。

（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)对系统中的硬盘设备进行检测。比如检测硬盘的数量、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。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)。在创建设备对象的时候, 驱动程序需要为每个设备对象取一个字符串名字。在 Hello China 的实现中, 第一个物理硬盘的名字为“PhysicalDisk0”, 第二个为“PhysicalDisk1”, 依此类推。对于分区设备, 第一个分区的名字为“Partition0”, 第二个为“Partition1”, 依此类推。

(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。因为上述几个设备对象已被 IOManager 插入设备链表。一旦插入设备链表, 用户应用程序(用户线程)即可调用 Open 函数(实际上是 IOManager 的 CreateFile 函数)来打开设备并访问了。

比如, 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 10.2.1 节)并初始化, 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(以设备名称为参数, 调用 CreateFile 函数)。在打开设备的过程中, IOManager 会根据设备名称查询整个设备链表, 返回匹配的设备对象指针(在 Windows 操作系统中, 这个返回的值叫做句柄。后续描述中, 有时候也把设备对象指针叫做句柄)。

2. IOManager 对设备对象和设备驱动程序的管理

在 Hello China 当前版本的实现中, 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。

整体架构可参考图 10-5。

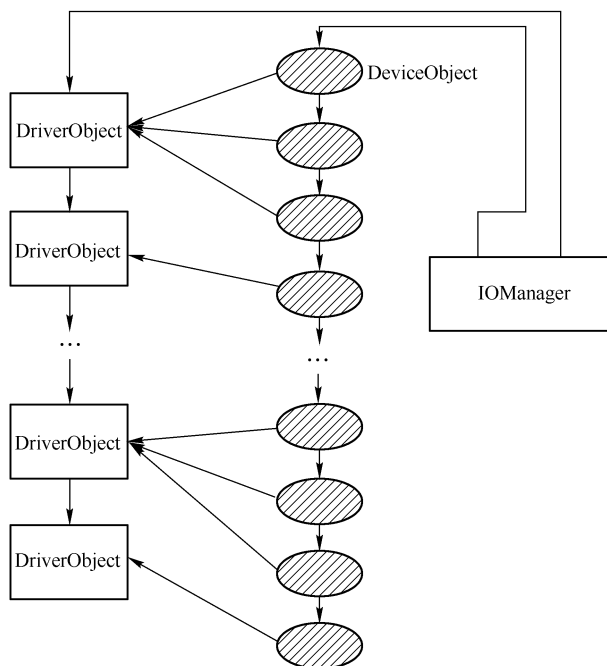


图 10-5 Hello China 的设备对象和设备驱动程序对象

3. IOManager 对象的实现

下面正式介绍 IOManager 对象的实现代码。首先看该对象的定义（为了方便，删除了部分注释和无关内容）：

```
[kernel/include/iomgr.h]
BEGIN_DEFINE_OBJECT(_IO_MANAGER)
//全局变量，包含设备链表、设备驱动程序链表、文件系统、文件系统驱动程序数组等。
//文件系统相关变量将在第 12 章中详细介绍。
    __DEVICE_OBJECT*          lpDeviceRoot;
    __DRIVER_OBJECT*          lpDriverRoot;
    __FS_ARRAY_ELEMENT        FsArray[FILE_SYSTEM_NUM];
    __COMMON_OBJECT*          FsCtrlArray[FS_CTRL_NUM];

    //面向应用程序的服务接口。
    BOOL                      (*Initialize)(__COMMON_OBJECT* lpThis);
    __COMMON_OBJECT*          (*CreateFile)(__COMMON_OBJECT* lpThis,
        LPSTR                  lpFileName,
        DWORD                  dwAccessMode,
        DWORD                  dwShareMode,
        LPVOID                  lpReserved);
    BOOL                      (*ReadFile)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT*      lpFileObject,
        DWORD                  dwByteSize,
        LPVOID                  lpBuffer,
        DWORD*                  lpReadSize);
```



```
BOOL          (*WriteFile)(__COMMON_OBJECT* lpThis,
                          __COMMON_OBJECT* lpFileObject,
                          DWORD              dwWriteSize,
                          LPVOID            lpBuffer,
                          DWORD*             lpWrittenSize);

VOID          (*CloseFile)(__COMMON_OBJECT* lpThis,
                          __COMMON_OBJECT* lpFileObject);

BOOL          (*CreateDirectory)(__COMMON_OBJECT* lpThis,
                                LPCTSTR lpszFileName,
                                LPVOID lpReserved);

BOOL          (*DeleteFile)(__COMMON_OBJECT* lpThis,
                            LPCTSTR lpszFileName);

BOOL          (*FindClose)(__COMMON_OBJECT* lpThis,
                          LPCTSTR lpszFileName,
                          __COMMON_OBJECT* FindHandle);

__COMMON_OBJECT* (*FindFirstFile)(__COMMON_OBJECT* lpThis,
                                LPCTSTR lpszFileName,
                                FS_FIND_DATA* pFindData);

BOOL          (*FindNextFile)(__COMMON_OBJECT* lpThis,
                              LPCTSTR lpszFileName,
                              __COMMON_OBJECT* FindHandle,
                              FS_FIND_DATA* pFindData);

DWORD         (*GetFileAttributes)(__COMMON_OBJECT* lpThis,
                                   LPCTSTR lpszFileName);

DWORD         (*GetFileSize)(__COMMON_OBJECT* lpThis,
                            __COMMON_OBJECT* FileHandle,
                            DWORD* lpdwSizeHigh);

BOOL          (*RemoveDirectory)(__COMMON_OBJECT* lpThis,
                                LPCTSTR lpszFileName);

BOOL          (*SetEndOfFile)(__COMMON_OBJECT* lpThis,
                              __COMMON_OBJECT* FileHandle);

BOOL          (*IOControl)(__COMMON_OBJECT* lpThis,
                          __COMMON_OBJECT* lpFileObject,
                          DWORD            dwCommand,
                          DWORD            dwInputLen,
                          LPVOID          lpInputBuffer,
                          DWORD            dwOutputLen,
                          LPVOID          lpOutputBuffer,
                          DWORD*           lpdwOutFilled);

BOOL          (*SetFilePointer)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpFileObject,
                                DWORD*           pdwDistLow,
                                DWORD*           pdwDistHigh,
                                DWORD            dwWhereBegin);

BOOL          (*FlushFileBuffers)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpFileObject);
```

```

//面向设备驱动程序的服务接口。
__DEVICE_OBJECT*      (*CreateDevice)(__COMMON_OBJECT* lpThis,
                                     LPSTR          lpDevName,
                                     DWORD           dwAttribute,
                                     DWORD           dwBlockSize,
                                     DWORD           dwMaxReadSize,
                                     DWORD           dwMaxWriteSize,
                                     LPVOID          lpDevExtension,
                                     __DRIVER_OBJECT* lpDrvObject);
VOID                   (*DestroyDevice)(__COMMON_OBJECT* lpThis,
                                     __DEVICE_OBJECT* lpDevObj);
BOOL                   (*LoadDriver)(__DRIVER_ENTRY DrvEntry);
//专门提供给文件系统驱动程序的接口。
BOOL                   (*AddFileSystem)(__COMMON_OBJECT* lpThis,
                                     __COMMON_OBJECT* lpFileSystem,
                                     DWORD           dwAttribute,
                                     BYTE*           pVolumeLbl);
BOOL                   (*RegisterFileSystem)(__COMMON_OBJECT* lpThis,
                                     __COMMON_OBJECT* lpFileSystem);
END_DEFINE_OBJECT()    //End of __IO_MANAGER.

```

可以看出, IOManager 的定义比较复杂, 涉及很多函数, 但这些对外函数 (或接口) 总体上可以分为四类:

- (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, 由用户调用来访问设备。如果读者对 Windows 的 API 比较熟悉, 那么对 IOManager 定义的这些函数的功能应该不会陌生。实际上这些函数的语义, 与 Windows 大致相同。
- (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。
- (4) 文件系统驱动程序专用的几个函数, 主要用于文件系统驱动程序向操作系统核心注册文件系统等功能, 在第 12 章中将做详细介绍。

在下面的部分中, 分别对初始化函数和设备驱动程序服务函数进行描述, 文件系统相关的全局变量和函数将在第 12 章中做详细介绍。

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。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。Hello China 当前版本的设计目标为嵌入式操作系统, 这样就不需要动态地加载设备驱动程序, 设备驱动程序事先已经同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与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) 其他相关工作。

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，对于与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 定义了一个数据结构：

```
BEGIN_DEFINE_OBJECT(__DRIVER_ENTRY_MAP)
    LPSTR                lpzDriverName;
    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，Initialize 函数在实现时，就会遍历这个数组，为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].lpzDriverName)
    {
        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，对于每个需要静态联编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}）是该数组结束的标志。

`Hello China` 目前没有实现动态设备驱动程序的加载功能，但如果将来需要，则可以按照下列思路实现这一功能。

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，并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。

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）：

（1）`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。

（2）`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。

（3）`WriteFile`，向设备或文件写入数据，实现机制与 `ReadFile` 类似。

（4）`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。

（5）`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。

(6) SetFilePointer，移动文件的当前指针。

(7) 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地使用了缓冲机制，即对文件的写操作先在内存中完成，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地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地打开一个文件，也可以在打开文件的时候，指定另外的打开标志，只允许其他应用程序只读地打开文件，等等。

这些函数的实现以及使用方法在第 12 章中有详细介绍，在此不作赘述。

6. IOManager 对设备驱动程序的接口

下列函数供设备驱动程序调用。

(1) 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。

(2) 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（调用 DeviceManager 对象提供的 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，但也必须显式地通过 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，则初始化结束，设备可以使用。

7. 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, 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。

8. 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掉, 以释放内存, 供其他应用程序使用。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 设备驱动程序

在上面的介绍中, 多次提到设备驱动程序对象和设备对象, 但一直没有给出设备驱动程序对象的详细定义和详细实现机制, 虽然也多次提到, 设备驱动程序对象实际就是设备操作函数的集合。本节将正式揭开设备驱动程序的神秘面纱。在此之前, 先介绍设备请求控制块的概念。

10.2.1 设备请求控制块

设备请求控制块 (Device Request Control Block, DRCB) 是 Hello China 的 IO 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。设备驱动程序根据 DRCB 里面的参数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。DRCB 是贯穿 Hello China 设备管理框架的核心数据结构。

该对象 (数据结构) 的定义如下。

```
[kernel/include/iomgr.h]
BEGIN_DEFINE_OBJECT(__DRCB)
    INHERIT_FROM_COMMON_OBJECT
    __EVENT*                lpSynObject;           //同步对象
    __KERNEL_THREAD_OBJECT* lpKernelThread;        //归属线程
    DWORD                   dwDrcbFlag;            //标志
    DWORD                   dwStatus;              //状态
    DWORD                   dwRequestMode;         //操作, 比如 read/write 等
```

```
DWORD                                dwCtrlCommand;

//操作结果的输出缓冲区
DWORD                                dwOffset;
DWORD                                dwOutputLen;
LPVOID                               lpOutputBuffer;

//输入信息
DWORD                                dwInputLen;
LPVOID                               lpInputBuffer;

//双向指针，用于把多个 DRCB 串接成一个队列（链表）
__DRCB*                              lpNext;
__DRCB*                              lpPrev;

DRCB_WAITING_ROUTINE                 WaitForCompletion;
DRCB_COMPLETION_ROUTINE               OnCompletion;
DRCB_CANCEL_ROUTINE                   OnCancel;
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。

- WaitForCompletion
- OnCompletion
- OnCancel

它们都是函数指针，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，设备驱动程序不能一直忙等待操作完成，因为这样会浪费很多 CPU 资源，而是采用中断的方式来等待完成结果。即设备操作完成之后，设备通过中断通知设备驱动程序，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（中断发生）。显然，这样做的好处是大大节约了 CPU 的资源。

第二个函数是与第一个函数对应的，这个函数由设备驱动程序在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函数中调用的。这个函数执行后，就会唤醒原来等待 IO 操作完成的线程（即调用 WaitForCompletion 函数进入阻塞状态的用户线程），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，那么该线程就可以被调度执行。

为了帮助读者进一步理解上述过程和配合关系，下面举一个硬盘读/写的例子。假设用户线程想读取一个文件，于是发起了一个 ReadFile 的函数调用，后续执行过程如下。

（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，然后再根据文件对象句柄，找到合适的文件对象（其实是一个设备对象），并调用对应的驱

动程序（文件系统驱动程序）提供的 DeviceRead 函数（以创建的 DRCB 对象为参数）。

(2) DeviceRead 函数根据传递过来的 DRCB 对象以及设备对象，找到实际的硬盘设备，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 对象为参数调用硬盘设备对象的 DeviceRead 函数。注意，这里的 DeviceRead 函数是由硬盘设备驱动程序提供的，而前一个 DeviceRead 函数是由文件系统驱动程序提供的。

(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 对象放到硬盘驱动程序的等待队列中，然后调用 DRCB 对象中的 WaitForCompletion 函数。这时用户线程（即调用 ReadFile 的线程）会进入阻塞状态。

(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），中断处理函数从硬盘控制器读取数据，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 函数唤醒等待的用户线程（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，省略了数据尺寸不匹配的情况（比如，用户请求 4KB 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 512B，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），因此实际情况可能比上述情况更加复杂。

最后一个函数（OnCancel）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），这时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 Hello China 的当前版本实现中，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，都是其所在的 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），所以这些函数很容易实现线程的阻塞、唤醒等操作。

另外，为了标识 DRCB 对象的状态，定义了 dwDrcbStatus 变量，这个变量可以取下列值：

(1) DRCB_STATUS_INITIALIZED: DRCB 对象已经被初始化，但尚未被任何线程应用。一个 DRCB 对象刚被创建后，就被设置为该状态。

(2) 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，但还没有收到最终响应，这个时

候, dwDrcbStatus 被设置为该值。

(3) DRCB_STATUS_COMPLETED: DRCB 对象跟踪的设备请求操作已经成功完成。

(4) 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。

(5) 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, 该变量指明了该 DRCB 跟踪的请求类型, 可以取下列值。

(1) 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。

(2) 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。

(3) 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。

(4) DRCB_REQUEST_MODE_SEEK: 在调用 DeviceSeek 函数的时候, 设定该值。

(5) DRCB_REQUEST_MODE_FLUSH: 在调用 DeviceFlush 函数的时候, 设定该值。

总之, DRCB 是从最初的用户请求, 到最终的物理设备操作的核心对象。在文件系统的实现中, 将会进一步介绍该对象的应用。

10.2.2 设备驱动程序对象的定义

在了解了 DRCB 对象的作用后, 再来具体看设备驱动程序的定义(为了描述简便, 删除了部分注释):

```
[kernel/include/iomgr.h]
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              (*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();
```

驱动程序对象是操作系统核心对象之一，与其他核心对象一样，这个对象也是从通用对象（Common object）继承来的。这样 common object 对象的一些机制，该对象也可直接继承使用。lpNext 和 lpPrev 两个指针，把系统中所有设备驱动程序对象连接在一起，形成一个双向链表。IOManager 对象中的 lpDriverRoot 变量即指向这个双向链表。

其他的都是一些函数指针，这些函数指针构成了设备驱动程序的标准功能集合，无论对于何种物理设备，其设备驱动程序必须实现上述部分或全部功能，才能被操作系统核心（具体说，是 IOManager）进行管理。在后续的介绍中，这些函数被称为功能函数。功能函数的具体使用方法，在下面的章节中会详细描述。

10.2.3 设备驱动程序的物理结构

可以说，设备驱动程序对象（__DRIVER_OBJECT）定义了设备驱动程序的逻辑结构。但设备驱动程序在物理上，必须以文件的形式存在，文件的组织形式，就是驱动程序的物理结构。下面对设备驱动程序的物理结构做简要说明。

设备驱动程序一般是由开发环境生成的。开发环境（包含编译器和链接器等）对源代码编译后，必须按照某种特定的格式对其进行链接。不同的操作系统，会对设备驱动程序的物理结构有不同要求。比如 Windows 操作系统，要求其设备驱动程序符合 VxD 规范。Hello China 的 V1.75 版本则采用 DLL 文件格式（实际上是 PE 文件格式）作为其设备驱动程序的物理结构。这是为了兼容性考虑，因为 Hello China 操作系统的内核模块，都是基于 DLL 结构的。而且 DLL 格式的文件能够得到大多数编译器的支持。图 10-6 大致展示了 Hello China 驱动程序文件组织结构（实际

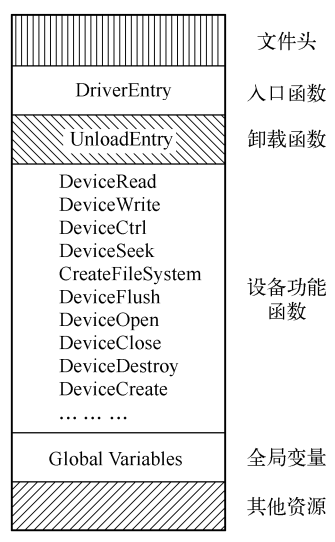


图 10-6 设备驱动程序文件在磁盘上的存储结构



上就是 PE 文件格式)。

其中, `DriverEntry` 和 `UnloadEntry` 用于设备驱动程序的初始化和设备驱动程序的卸载, 其他的函数和全局变量用于实现该设备驱动程序的特定功能。相应的, `DriverEntry` 必须是 DLL 文件的入口函数, 其在驱动程序文件内的具体位置, 由编译器填写到 DLL 文件头中。而其他函数则无需固定位置, 只需在 `DriverEntry` 中用这些功能函数初始化驱动程序对象的对应成员函数即可。

10.2.4 设备驱动程序的功能函数

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组标准的功能函数集合, 这些功能函数由 `IOManager` 调用 (设备驱动程序在初始化时, `DriverEntry` 函数要把这些函数的地址填写到驱动程序对象对应的函数指针内), 设备驱动程序对实际设备的操作就是在这些功能函数中实现的。

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, 对应功能函数中的 `DeviceRead/DeviceWrite`、`DeviceOpen/DeviceClose` 等函数, 但也有一些其他的操作, 比如定位当前设备位置 (`DeviceSeek`)、特殊的控制命令 (`DeviceCtrl`) 等。其中 `DeviceCtrl` 函数最为灵活, 这个函数为特殊设备的特殊功能 (不能抽象为 `Read/Write` 等操作的功能) 提供了支持。

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, 初始化并注册到操作系统 (确切地说是 `IOManger`) 维护的设备对象链表中; 对于关闭操作, 设备驱动程序释放对应的系统资源, 并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), 本节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, 作为实现具体设备驱动程序时的参考。

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` 函数来实现, 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, 然后根据用户提供的设备对象的地址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驱动程序对象的后向指针), 调用设备驱动程序对象的 `DeviceRead` 函数。到此为止, 所有的操作都是由操作系统核心完成的 (确切地说是 `IOManager`), 后续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 DRCB 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。于是会根据设备访问方式的不同（中断或轮询），采取不同的处理动作。

（4）对于中断方式，DeviceRead 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）。该函数的调用会阻塞当前线程。

（5）对于轮询方式，则当前线程会持续检查队列的状态，直到队列为空。这时候才发起设备读取操作。显然，这是一个忙等待的过程，非常消耗 CPU 资源。设备操作成功完成或失败后，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）。

（6）在中断方式下，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，操作系统会调用对应的中断处理程序。中断处理程序从 DRCB 队列中摘取一个 DRCB 对象，根据设备的操作结果填充该对象，然后调用该 DRCB 对象的 OnCompletion 函数，该函数唤醒等待的线程。然后进一步检查 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。图 10-7 反映了上述调用关系。

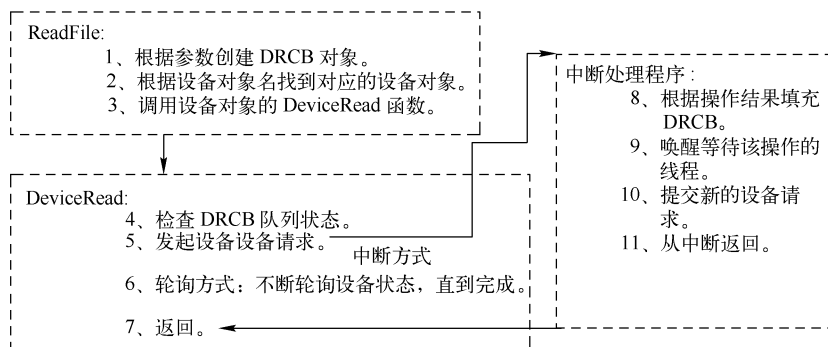


图 10-7 设备读取操作的步骤

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，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。

2. 写操作（DeviceWrite）的实现

写操作的过程与读操作基本一致。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。

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地读写，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地读写。在 UNIX 的实现中，对这两种类型的设备分别做了处理（对应于 UNIX 的字符设备和块设备）。而在 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）。至于如何获取每次操作的字节数量，DeviceRead/DeviceWrite 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 1B，针对磁盘可以是 512B。

(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地实现该功能，如果不能实现，则简单地返回失败结果。

(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”。

其他的功能，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。

其他的功能函数，比如 DeviceSeek、DeviceFlush 等，功能比较简单，在第 12 章中会有比较详细的描述。

10.2.5 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。一般情况下，该函数可以做下列工作。

- (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功能函数。
- (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功能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：

```
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）。对物理设备的资源（IO 端口、中断向量、内存映射范围等）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，而自己硬性地给设备分配资源，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，该空间内存储了与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。设备扩展的大小和具体内容是与特定物理设备相关联的，只有设备驱动程序自己知道，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其初始化的例子。

```
_DEVICE_OBJECT*  lpIdeHardDisk  = NULL;
lpIdeHardDisk = CreateDevice("IDE Hard Disk 0", //Device name.
                             lpDriverObject,   //Driver object.
                             NULL,             //Resource descriptor.
                             lpIDEEExt,        //Device extension.
                             DEVICE_TYPE_STORAGE,
                             512);             //Device block size.

If(NULL == lpIdeHardDisk)    //Failed to create device.
    Return FALSE;
InitializeIdeHardDisk(lpIdeHardDisk); //Initialize it.
... ..
```

CreateDevice 函数由 IOManager 实现，该函数创建一个设备对象（_DEVICE_OBJECT 对象），并使用函数指定的参数对其初始化，然后插入到系统的设备对象链表中。设备对象的定义和机制，请参考本章的后续内容。

10.2.6 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。

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地加载或卸载，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（如内存等），但在卸载的时候没有释放，那么会造成资源泄漏。

10.3 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。CreateFile（IOManager 提供的用户侧接口调用）函数操作成功后，返回值即为打开的设备对象的地址。

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 就是靠这个名字来检索设备对象列表，找到具体的设备的。

10.3.1 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：

```
[kernel/include/iomgr.h]
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                dwEndPort;
    DWORD                dwDmaChannel;
    DWORD                dwInterrupt;
    LPVOID               lpMemoryStartAddr;
    DWORD                dwRefCounter;
    DWORD                dwBlockSize;
    DWORD                dwMaxReadSize;
    DWORD                dwMaxWriteSize;
    LPVOID               lpDevExtension;
END_DEFINE_OBJECT();
```

该对象也是从通用对象（COMMON OBJECT）继承来的。lpNext 和 lpPrev 是两个前后向指针，把系统中的所有设备对象连接成一个双向链表。IOManager 中的 lpDeviceObject 则指向这个双向链表的头节点。在下面的几节中，将对该定义中的其他几个重要字段（变量或成员）进行说明。

10.3.2 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，这些设备不对应具体的物理外设，但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，就可以间接地访问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，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。为简单起见，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，对于服务器 shanghai 上的一个共享文件 shanghai.map，命名结果为\\.shanghai\shanghai.map。

设备的命名机制是操作系统对设备进行管理的基础，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，也可以成功加载和初始化，但可能会引起混乱。比如一个物理设备，把自己的名字命名为 D:\HOWAREYOU，则可能会被操作系统当作一个文件对待。

10.3.3 设备对象的类型

为了方便管理，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，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设备对象。

(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。

当设备驱动程序加载完毕, `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` 会尝试使用系统中已安装的文件系统来初始化这个设备。比如, 针对硬盘 (严格来说, 应该是硬盘的每个分区), 操作系统会调用已安装文件系统的 `CheckPartition` 函数, 让文件系统驱动程序去检查该存储设备是否被格式化成了对应的文件系统。如果是, 则文件系统驱动程序 (`CheckPartition` 函数) 会调用 `IOManager` 提供的 `AddFileSystem` 函数, 向系统中添加一个文件系统。具体的内容, 在第 12 章中会有详细介绍。

10.3.4 设备对象的设备扩展

设备扩展是设备对象定义中的最后一个变量 (`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特征数据上。例如, 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/10M 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, 该指针由设备驱动程序初始化, 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, 则由设备驱动程序进行管理, 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。

10.3.5 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)。下面描述了打开一个物理设备的过程:

(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, 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\\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，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。否则即可保存设备对象句柄，由后续其他操作（比如 `ReadFile` 等）使用。

可以看出，上述操作的关键是遍历整个系统设备对象链表。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，可能发生冲突，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。

10.3.6 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，而不要随意地命名。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，可以让用户很容易地得到设备描述信息。

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）。

1. 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 32B 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，就可以确保不会产生冲突。

2. 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由统一的组织管理并分配，可以确保全球唯一，该地址由 48bit（6B）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
```

10.4 设备的中断管理

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与相应的中断向量进行连接。

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 lpESP,LPVOID lpParam);
```

其中，第二个参数为传递给中断处理程序的特定参数，第一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判断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`），来判断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刚

刚调用的中断处理程序来处理，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不是针对自己的呢？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，或者通过设备相关的一些特定操作来确定。例如，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，就进行进一步处理，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不是自己的中断。因为正常情况下，必须由设备驱动程序发起请求，比如一个读取操作，当操作结束时，设备才会发生中断，所以，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，读取适当的状态寄存器，来判断是不是 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特征来考虑。

10.5 设备管理实例：串口通信程序

下面通过一个串口通信程序，进一步说明 Hello China 的设备管理机制。严格来说，这不是一个设备驱动程序，而是一个应用程序。但是这个应用程序通过调用 Hello China 提供的设备管理服务，实现了硬件（串口）的管理，同时也使用了 Hello China 提供的线程同步机制实现了用户输入与设备输入之间的同步。这些技术都是在驱动程序开发过程中经常使用的，希望通过这个实际程序的介绍，让读者对设备管理机制有更进一步的了解。

更重要的是，本书曾多次提到物理设备的两种最主要操作模式——轮询方式和中断方式，但都没有深入介绍。在下面这个实例的介绍中，将详细介绍这两种设备操作方式。可以说，这两种操作方式的机理是操作系统设备管理的最核心内容之一。

10.5.1 串行通信接口概述

一般情况下，IBM PC 兼容机的串口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 的编程实际上是对内部寄存器执行读写操作。因此可将 UART 看作一组寄存器集合，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表 10-1。其中端口 0x3F8-0x3FE 用于 PC 上 COM1 串行口，0x2F8-0x2FE 对应 COM2 端口。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。

表 10-1 给出了 PC 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。



表 10-1 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 的含义
0x3F8/0x2F8	R	0	读接收缓存寄存器。含有收到的字符
	W	0	写发送保持寄存器。含有将发送的字符
	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	读中断标识寄存器。中断处理程序用以判断此次中断是 4 种中的哪一种。 位 7-3 全 0 (不用); 位 2=1 确定中断的优先级; =11 接收状态有错中断, 优先级最高; =10 已接收到数据中断, 优先级第 2; =01 发送保持寄存器空中断, 优先级第 3; =00 modem 状态改变中断, 优先级第 4。 位 0=0 有待处理中断; =1 无中断
0x3FB/0x2FB	W	X	写线路控制寄存器。 位 7=1 除数锁存访问位(DLAB)。 0 接收器, 发送保持或中断允许寄存器访问; 位 6=1 允许间断; 位 5=1 保持奇偶位; 位 4=1 偶校验; =0 奇校验; 位 3=1 允许奇偶校验; =0 无奇偶校验; 位 2=1 1 位停止位; =0 无停止位; 位 1-0 数据位长度: =00 5 位数据位; =01 6 位数据位; =10 7 位数据位; =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 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)

下面简要介绍表中比较重要的几个寄存器比特。

(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, 用于设置或读取其他寄存器的值。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, 串行接口有可能与其他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中断是不是串行接口发生的。若是, 则进行处理, 否则把处理过程转让给其他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, 可通过图 10-8 所示对话框设置 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 10-8 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, 按照表 10-2 来设置串口的波特率因子。



表 10-2 波特率与波特率因子之间的对应关系

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

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时，才会访问到波特率因子寄存器。

10.5.2 串行通信编程方式

了解了上述内容之后，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。

1. 串口初始化

在发送或接收数据前，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); //把 DLAB 位设为 1。
    __outb(0x0C,0x3F8); //设置波特率分量的低字节。
    __outb(0x0,0x3F9); //设置波特率分量的高字节。
    __outb(0x07,0x3FB); //设置 DLAB 为 0，8 位数据，1 位停止，无奇偶校验。
    __outb(0x0D,0x3F9); //使能所有中断。
    __inb(0x3FB); //从数据端口读取一个字节，以复位数据端口。
}
```

2. 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地检查外设的状态，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用的时间内，发送或接收程序也不停地运行，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。

下面介绍这两种方式下的串口数据传输。

(1) 基于轮询方式的数据发送

轮询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。下面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)
    {
        while(nCounter1-- > 0)
        {
            if(__inb(wPort + 5) & 32)//发送寄存器空，开始发送
            {
                __outb(wPort,bt); //发送一个字节
                return TRUE;
            }
        }
        nCounter1 = 1024;
        __MicroDelay(1000); //延迟 1ms.
    }
    return FALSE; //超时，失败返回
}
```

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。

(2) 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，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;
        }
    }
    //等待一小段时间后，若发送保持寄存器仍然被占用，
    //则进入睡眠状态等待更长时间
    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，需重点关注的中断类型是“发送保持寄存器为空”中断（ISR 的第二个比特为 1）。在 COM 接口的发送寄存器为空时，会引发中断。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。

3. 数据接收

(1) 基于轮询方式的数据接收

基于轮询方式的数据接收，与基于轮询方式的数据发送程序类似。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。

（2）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。

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 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。

4. 串口交互程序的实现

有了上述知识作为铺垫之后，下面正式介绍 Hello China 附带的串口通信程序的实现。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 启动完成进入字符界面后，输入 `hypertrm` 或 `hyptm2` 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 `hyptm2` 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，然而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 设备管理机制的使用方法以及步骤，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。

5. 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：

(1) 通过 PC 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，如图 10-9 所示。

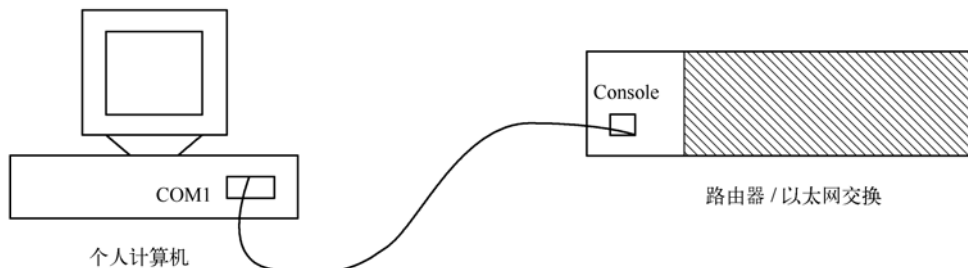


图 10-9 通过 PC 的串口控制网络设备

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 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 `hyptrm2`，就可启用串口交互程序。这时候，按下回车键或其他键，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 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、1 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 `hyptrm2`，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，退出一个命令行程序。但为了实现上的简便，暂且以这种方式结束程序的运行。读者可通过简单修改代码，使得这个超级终端模拟程序能够支持 `CTRL+C` 等组合键的退出。

（2）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 PC 上分别启用 `hypertrm` 或 `hyptrm2` 程序，在 PC1 上输入的数据，就可显示在 PC2 上，反之亦然。这实际上是一个点对点的通信程序。这种情形如图 10-10 所示。

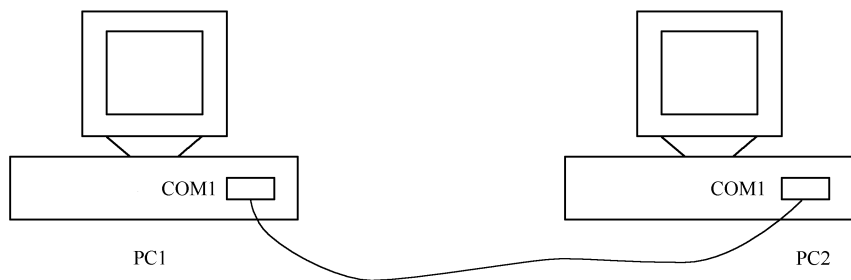


图 10-10 通过串口实现两台计算机的通信

连接两台 PC 的串口线，可自行制作，也可到电子市场上购买。若自行制作，可到互联网上搜索串口线的制作指南，在此不作赘述。

6. 轮询模式的串口交互程序实现

轮询模式的 IO 交互程序，由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` 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不是 `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，1 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，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 `ComSendByte`，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，与基于轮询方式的数据发送类似，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;

    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他形式的键，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，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（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) 获得串口的字符后，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）。若是其他字符，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。函数应该遵循 `Hello China` 定义的核心线程入口函数原型（以 `LPVOID` 为参数，返回 `DWORD`）。该主入口函数实现了

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 = CreateKernelThread(
        (__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 = CreateKernelThread(
    (__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,对用户的任何输入(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` 应用程序的启动。

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。另外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。

7. 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,即仅当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，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，不论是接收还是发送，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，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，应用程序可先启动一个发送操作，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。

但本书实现的 `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，仍然采用与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。但一些实现细节和数据结构则有较大不同。`hypertrm` 没有采用任何复杂的数据结构，只是简单地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，因为涉及中断处理程序和接收线程的数据交互和同步，所以采用了环形缓冲区（ring buffer）对象。

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), 所有其他中断, 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, 而且功能也不会受太大影响。若需要补充其他类型的中断, 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, 即使发生了可引发中断的事件。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。

在中断模式下的发送线程 `hyptm2` 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;

    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);
                    return 0L;
                }
                __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` (即 'z' 字符), 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他字符, 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; //指示结束的事件对象
    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)
{
    __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 `hyptrm2` 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) 若是其他字符，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。

与轮询方式实现的 `hyptrm` 最大的不同，就是在 `hyptrm2` 的实现中，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;
    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的数据，然后调用 `AddRingBuffElement`，添加到环形队列中。`AddRingBuffElement` 函数会唤醒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) 在没有中断要处理的情况下中断程序的返回值。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，于是会继续调用其他的中断处理程序（这些中断处理程序对应的设备，与 `COM` 接口共享同一中断输入）。若返回 `TRUE`，则表明当前中断处理程序已成功地处理了中断，于是中断调度程序不会再调用其他设备的中断处理程序了。

(2) 之所以会出现中断程序被调用，但中断状态寄存器却表明没有中断发生（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与 `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由 `COM` 接口芯片引发，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，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：

```
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;

    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_O
BJECT*)hSendThread);

//Wait for receiving and sending kernel threads to terminate.
WaitForThisObject(hSendThread);
WaitForThisObject(hRecvThread);

__TERMINAL:
if(NULL != hIntHandler)
{
    DisconnectInterrupt(hIntHandler);
}
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，但比较简单，主要由三部分组成：

(1) 初始化部分代码。在这部分代码中，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。完成这些工作后, 才调用 InitComPort2, 初始化串口芯片。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, 就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。

8. 串行通信编程总结

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 不停地去查询设备的状态, 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地等待即可, 一旦设备有输入, 就会引发中断, 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, 从而延误关键事件的处理。更糟糕的是, 若系统外设硬件故障, 导致外设不断引发中断, 这样可能会使系统一直忙于处理不重要的外部中断, 无法响应其他的系统事件。

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, 适当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, 也不会对系统中其他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。这样可提高系统的整体鲁棒性。

10.6 设备驱动程序管理总结

至此, 设备驱动程序管理就讲解完了。设备驱动程序管理是操作系统最核心的功能之一, 也是最复杂的功能。在 Hello China 的实现中, 由三个功能相互独立的全局对象, 组成了驱动程序管理的核心: DeviceManager、IOManager、System。

其中 DeviceManager 对象完成对物理设备的硬件资源的管理, 包括 IO 端口资源、中断

向量资源、内存映射资源等。IOManager 则完成了设备对象和设备驱动程序对象的管理，同时提供了统一的用户访问接口。文件系统的管理也是在 IOManager 对象中实现的，这是第 12 章的重点内容。System 对象提供了中断的管理功能。

设备驱动程序对象（__DRIVER_OBJECT）和设备对象（__DEVICE_OBJECT）是操作系统设备管理框架中的另外两个核心对象，分别与设备驱动程序和硬件设备对应。IOManager 维护两个全局链表，把系统中所有的设备驱动程序对象和设备对象连接到了一起。其中设备对象维护了指向其对应设备驱动程序对象的指针。而 DRCB（设备请求控制块）对象则是贯穿整个 IO 过程的核心数据结构，这个数据结构完成函数之间的参数传递和操作结果记录功能。

最后，我们通过一个示例程序讲解了设备驱动程序管理机制的应用，同时通过这个示例，详细讲解了轮询方式和中断方式这两种最常见的设备访问方法。

希望通过本章的内容，使读者对操作系统的设备管理功能有一个比较深入的认识。操作系统的设备管理是一个内容非常庞杂的主题，先不说内容本身，单是如何有序、清晰地组织这部分内容就是一个挑战。作者不是文学家，无法把这些相对分散的内容组织成散文一样行云流水的形式，既让读者理解内容，又让读者产生美感。在本章的描述中，作者只是根据大致的功能划分，按照从整体到局部的顺序，介绍了操作系统设备管理的内容。如果读者读完本章感觉凌乱和迷惑，不要紧，再读一遍，相信会有不同的感觉。

第 11 章 图形用户界面

11.1 图形用户界面概述

图形用户界面（GUI，Graphical User Interface）对操作系统的重要性是不言而喻的，个人计算机的普及，最重要的驱动因素就是 GUI。正是由于有了直观、易用的 GUI，才缩短了计算机与人的距离。基于命令行的复杂用户界面，对于普通的个人来说，毕竟还是太专业了，那只是计算机专业人士的玩具。如果没有 GUI 的出现和发展，计算机这个现代化工具可能依然停留在科学实验室内。

与 GUI 对应的是 CLI（命令行接口，Command Line Interface），最典型的的就是 DOS/UNIX 等操作系统的命令行界面。虽然对普通个人用户来说，命令行界面是过于专业了，但是对于熟练的专业人士来说，CLI 的效率远远超过 GUI。从逻辑上说，命令行界面是一维的、串行执行的，这与计算机本身的逻辑结构是吻合的。操作人员的视线也只要集中在光标处即可，无需像 GUI 那样，在整个屏幕上寻找输入热点。实际上，在一些大型计算机上，命令行操作模式几乎是唯一的选择。这里之所以提一下 CLI，是为了说明作为传统的人机交互方式，命令行界面仍然有其独特优势。很难从总体上说究竟是 GUI 更有优势还是 CLI 更有优势，只能说这两者有不同的适用对象，GUI 更适合普通的计算机用户，当然，对图像处理用户来说，GUI 是唯一选择。而 CLI 则更适合系统管理员、程序员等专业人士。

本章将聚焦 GUI。这是一个庞大的课题，其理论基础是计算机图形学。对计算机专业的学生来说，计算机图形学是比较复杂的课程之一，因为这门课程要用到比较复杂的数学变换和推导。如果从计算机图形学的层次开始，来完整说明 GUI 的工作原理，可能至少需要三本书的规模。在本书中，为了限制规模，同时考虑到作者自身的技能范围，我们把注意力集中到 GUI 的实现框架上，尽量不涉及底层的图形学理论。有时为了说明一些关键问题，图形学理论是无法绕过的，这时本书也会局限在文字描述上，不会涉及数学公式和数学推导。即使如此，在几十页的篇幅内，完全说清楚 GUI 的每一个方面也很困难。本书只选择 GUI 中的关键问题和实现方案进行说明，读者在理解这些基本原理和概念的基础上，可通过阅读代码，进一步了解其他相关方面的实现机制。

本章关注下列几个主题：

- （1）符合 VESA 标准的图形显示卡的操作方法。
- （2）Hello China 对显示设备的抽象。
- （3）基于 GUI 的应用程序编程模型。
- （4）用户输入（键盘、鼠标等）如何传递到一个具体的窗口上。
- （5）窗口之间如何协同一致工作，比如窗口重绘、关闭等。
- （6）与窗口关联的设备上下文（Device Context，DC），这是在窗口上绘制图形的基础对象。

(7) 基于 GUI 的 shell。

与其他章节一样，本章仍然以 Hello China V1.75 版本的代码为例，来解释 GUI 的实现。虽然以 Hello China 操作系统来说明，但其原理和概念是相通的，这些原理和概念可以应用到任何实现了 GUI 的操作系统中。需要说明的是，Hello China V1.75 实现的 GUI 功能还不是很完善，只具备了一个可用的框架和基础功能。一些高级功能，比如字体、动画、颜色渐变等尚未实现，但对本部分不会造成影响。OK，让我们正式进入主题吧。

11.2 符合 VESA 标准的显卡操作方法

VESA 是国际视频电子标准学会（Video Electronics Standards Association）的缩写，这是一个专门制定计算机显示标准的组织，由它制定的标准，就叫做 VESA 标准，目前已发展到 VESA 3.0 版本。凡是符合 VESA 标准的显示卡或显示设备，都可以通过一组既定的方式来进行操作。当然，为了提升竞争力，一般的显示卡都支持 VESA 标准，就像一般的计算机，都能够支持 Windows 操作系统一样。否则会没有市场，除非完全是为了研究或者好玩。

VESA 标准体系有很多子标准，包含显示相关的方方面面。但与本章的主题关系最紧密的，是一个叫做 VBE（Vesa BIOS Extension）的标准。顾名思义，就是在标准 BIOS 功能基础上进行扩展，使得只要通过 BIOS 调用，就可操作显卡的标准。在进一步介绍之前，先引入显示模式的概念，即显卡的工作模式。比如显卡可以工作在 800×600 像素模式，每个像素可以有 256 色，等等。VBE 标准对各种显示模式进行了统一的编号，只要通过 BIOS 调用，告诉显示卡的显示模式编号，显示卡就可切换到指定的模式下进行工作，前提是该显卡支持设定的工作模式。一些常用工作模式见表 11-1。

表 11-1 常用工作模式

显示模式编号	分辨率	每像素颜色数
0x100	640×400	256
0x101	640×480	256
0x102	800×600	16
0x103	800×600	256
0x104	1024×768	16
0x105	1024×768	256
0x106	1280×1024	16
0x107	1280×1024	256
0x10D	300×200	5:5:5
0x10E	320×200	5:6:5
0x10F	320×200	8:8:8
0x110	640×480	5:5:5
0x111	640×480	5:6:5
0x112	640×480	8:8:8
0x113	800×600	5:5:5



(续)

显示模式编号	分辨率	每像素颜色数
0x114	800×600	5:6:5
0x115	800×600	8:8:8
0x116	1024×768	5:5:5
0x117	1024×768	5:6:5
0x118	1024×768	8:8:8
0x119	1280×1024	5:5:5
0x11A	1280×1024	5:6:5
0x11B	1280×1024	8:8:8

在这个表格中，每像素颜色数有两种表示方式，第一种是纯数字，比如 16、256 等，指的是一个像素最多可以有 16 色，或者 256 色。具体显示哪个颜色，由一个调色板来控制。另外一种表示形式是诸如 5:6:5、8:8:8 等，这指明一个像素颜色的三基色（RGB，红/绿/蓝）分别占用的比特数。比如 8:8:8，说明一个像素的颜色值需要用 24 个比特表示，其中 R/G/B 各占 8 个比特。这样一个像素总共可以达到 2^{24} 种颜色，这就是所谓的真彩色。

一般的显示卡，只要不是太古老，上表中的所有显示模式都是支持的。在 Hello China V1.75 的实现中，作者选择 0x118 号工作模式作为 GUI 模块的标准工作模式。一旦试图切换到 GUI 模式（在字符界面下执行 GUI 命令），Hello China 会首先检查显卡是否支持这种模式。如果不支持，则不能切入 GUI 模式。从作者测试的情况来看，大部分计算机的显示卡都是支持的。

现在就面临两个问题：一是如何判断显示卡是否支持 0x118 模式。虽然大部分是支持的，但是不排除存在不支持的情况，因此必须做出判断。二是如何切入 0x118 号模式。显然，VBE 定义了明确的 BIOS 功能扩展，来完成这两个功能。下面分别说明。

11.2.1 判断显示卡是否支持 VBE 标准

在判断显示卡是否支持 0x118 号显示模式之前，有必要首先判断 BIOS 是否支持 VBE 标准。如果连 VBE 标准都不支持，那么后续工作就可省略了。因为所有后续设置，包括测试是否支持 0x118 模式，设置该显示模式等，这些工作都依赖于 VBE 标准。需要补充的是，有可能显示卡是支持 0x118 显示模式的，但 BIOS 却不支持 VBE 标准。这不奇怪，在 BIOS 版本比较老而显示卡比较新的情况下，可能会遇到这种情况。通用操作系统，比如 Windows 等，是可以支持这种情况的，因为在 Windows 启动后，会加载独立的显卡驱动程序，不依赖于 BIOS 的功能。但 Hello China 却不支持这种情况。

我们知道，BIOS 0x10 号调用是专门针对显示设备的功能调用，VBE 对这个功能调用做了扩展。通过 VBE 标准可知，给 ax 寄存器传入 0x4F00 参数，调用 0x10 号中断服务，可判断是否支持 VBE 标准。如果支持，则 al 寄存器中会返回 0x4f，否则会返回其他值。下面是一段示例代码：

```
ll_testvbe:
    mov di,DEF_VBE_INFO
    mov ax,0x4f00
```

```
int 0x10
cmp al,0x4f
jnz .ll_failed      ;Can not support VBE mode.
ll_setmodebgn:
;Can support VBE mode,then try to set the desired display mode.
ll_failed:
;Can not support VBE mode,handle error here.
```

代码比较简单, DEF_VBE_INFO 是一个预定义的常数, 该常数是一段可用内存的起始地址。在 int 0x10 调用成功的情况下, 会把 VBE 相关的信息存储在该地址处。由于我们不会用到这些 VBE 相关的信息, 因此不必理会具体的返回内容是什么。我们关注的是 BIOS 是否支持 VBE, 因此在完成 0x10 调用后, 直接检查 al 是否为 0x4F。如果是, 说明调用成功, 于是会执行 ll_setmodebgn 标号处的代码。如果 al 的值不是 0x4F, 说明调用失败, BIOS 不支持 VBE 模式, 于是跳转到 ll_failed 标号处继续执行。这时候就需要在 ll_failed 标号处进行错误处理了。

一般情况下, BIOS 都是支持 VBE 标准的, 因此上面的探测会成功, 会执行标号 ll_setmodebgn 后的代码。这里就是设置显示模式的地方了。

11.2.2 切换到 0x118 工作模式

从理论上说, 在设置显示卡工作模式为 0x118 之前, 首先应该检查一下显示卡是否支持该模式。因此应该有两段代码, 一段是检测是否支持该模式, 一段是在支持的情况下, 设置显示卡工作在 0x118 模式下。但是根据 VBE 的标准, 这两个过程是合一的。即直接尝试设置 0x118 模式即可。如果显示卡支持, 则会设置成功, 如果不支持, 则设置失败。按照 VBE 的标准, 在 ax 寄存器中传入 0x4F02, bx 寄存器中存入待设置的显示模式代码, 然后调用 0x10 号中断, 即可设置显示模式为 bx 寄存器中指定的值。如果设置成功, 则 ah 寄存器返回 0。如果 ah 寄存器的返回值不为 0, 则说明设置失败。下面是设置工作模式为 0x118 的汇编代码:

```
mov bx,0x4118
mov ax,0x4f02
int 0x10
cmp ah,0x00
jnz ll_failed
ll_setmodesucc:
;Set display mode 0x118 success
ll_failed:
;Set display mode 0x118 failed.
```

代码比较简单, 如果设置成功, 说明支持 0x118 模式, 于是会继续执行 ll_setmodesucc 后面的代码。否则跳转到 ll_failed 标号处, 做失败处理。

在上面的代码中, bx 寄存器的值被设置为 0x4118, 而不是 0x118, 这里需要做进一步解释。按照 VBE 的标准, 针对每种工作模式, 有两种显存访问方式: 平直访问方式 (flat display memory mode) 和非平直访问方式。在平直访问方式下, 显示器显示内容与显存有直

接对应关系。如果按照显示器从左到右、从上到下的顺序，为每个像素编号的话，那么每个像素的颜色值，就存放在以显存地址为基址、以像素编号为索引的位置处。假设显示卡工作在 0x118 模式下，则屏幕上共有 $1024 \times 768 = 786432$ 个像素点，每个像素点占用 4 个字节的存储空间来存储颜色值（R/G/B 各一个字节，再加上一个 alpha 字节，共四个字节），则需要的显示存储空间为 $786432 \times 4 = 3\text{MB}$ 。相反，在显存的对应地址上写入四个字节的颜色数据，对应的颜色就可显示在屏幕上与之对应的像素点上。

这种工作模式非常简单直观，也是大多数显示卡都支持的显示模式。在 Hello China 的实现中，就使用了这种平直显存访问方式。按照 VBE 的标准，0x4118 代表的就是 0x118 模式的平直内存工作模式。于是我们在 bx 寄存器中装入 0x4118，来设置这种模式。

这时候另外一个问题又出来了，就是如何确定平直显存的起始地址。这个地址是变动的，应该是由 BIOS 对显示卡进行配置后的结果。答案是显然的，VBE 标准会提供接口，让程序员获得该地址。如果 VBE 不提供这样的接口，则这个标准就不是完备的。作为一个国际标准，VBE 不会犯如此低级的错误。阅读 VBE 标准可知，当 ax 的值为 0x4F01 的时候，在 cx 中存入显示模式号，调用 0x10 中断，可获得对应显示号的详细数据。如果调用成功，ah 中会返回 0，详细的显示模式信息，会被 BIOS 存放在 di 寄存器指定的位置处。下面是获取 0x118 显示模式详细信息的汇编代码：

```
mov di,DEF_VBE_INFO
mov cx,0x118
mov ax,0x4f01
int 0x10
cmp ah,0x00
jnz ll_failed
ll_ok:
;Retrieve display mode OK.
ll_failed:
;Retrieve display mode failed,error handling here.
```

代码比较简单，如果调用成功，则模式详细信息会被存放到 DEF_VBE_INFO 位置处，我们可以读取该位置的内存内容，来获取详细的模式信息。如果调用失败，则跳转到 ll_failed 标注的代码处做错误处理。需要注意的是，这时候的模式号（存入 cx 寄存器的值），就不是 0x4118 了，而是 0x118。作者曾在这个地方出过问题，把 0x4118 存入了 cx，结果调用失败。

调用成功后，就需要进一步分析模式信息，获取平直显存起始地址了。按照 VBE 的标准描述，模式信息应该是下面定义的这样：

```
struct __VBE_MODE_INFO{
    // Mandatory information for all VBE revision
    WORD modeattributes;    // Mode attributes
    BYTE winaattributes;    // Window A attributes
    BYTE winbattributes;    // Window B attributes
    WORD wingranularity;    // Window granularity
    WORD winsize;           // Window size
```



```

WORD  winasegment;      // Window A start segment
WORD  winbsegment;      // Window B start segment
DWORD winfuncptr;       // pointer to window function
WORD  bytesperscanline;  // Bytes per scan line

// Mandatory information for VBE 1.2 and above
WORD  xresolution;       // Horizontal resolution in pixel or chars
WORD  yresolution;       // Vertical resolution in pixel or chars
BYTE  xcharsize;         // Character cell width in pixel
BYTE  ycharsize;         // Character cell height in pixel
BYTE  numberofplanes;    // Number of memory planes
BYTE  bitsperpixel;      // Bits per pixel
BYTE  numberofbanks;     // Number of banks
BYTE  memorymodel;       // Memory model type
BYTE  banksize;          // Bank size in KB
BYTE  numberofimagepages; // Number of images
BYTE  reserved1;         // Reserved for page function

// Direct Color fields (required for direct/6 and YUV/7 memory models)
BYTE  redmasksize;       // Size of direct color red mask in bits
BYTE  redfieldposition;  // Bit position of lsb of red bask
BYTE  greenmasksize;     // Size of direct color green mask in bits
BYTE  greenfieldposition; // Bit position of lsb of green bask
BYTE  bluemarksize;      // Size of direct color blue mask in bits
BYTE  bluefieldposition; // Bit position of lsb of blue bask
BYTE  rsvdmasksize;      // Size of direct color reserved mask in bits
BYTE  rsvdfieldposition; // Bit position of lsb of reserved bask
BYTE  directcolormodeinfo; // Direct color mode attributes

// Mandatory information for VBE 2.0 and above
DWORD physbaseptr;      // Physical address for flat frame buffer
DWORD offscreenmemoffset;  // Pointer to start of off screen memory
WORD  offscreenmemsize;    // Amount of off screen memory in 1Kb units
char  reserved2[206];     // Remainder of Mode
}

```

内容比较多且复杂，我们只关注 **physbaseptr**（黑体标注）这个参数，这就是 flat memory 的起始地址。获得这个地址后，就可以通过直接向这个地址写入颜色值，来操作显示器了。

到此为止，简单地介绍了通过直接写显存来操作显卡的工作原理，为进一步的介绍奠定基础。直接写显存是最基本、最简单的显卡操作方法，也是比较通用的一种方法。其最大的优点就是简单直观，不需要了解显卡的具体工作原理。但实际上，显卡是一块非常复杂的集成电路芯片，复杂程度甚至可以跟 CPU 媲美。原因是在显卡上，集成了非常多的附加功能。从比较基本的功能如矩形填充、贝塞尔曲线、二维动画功能等，到比较复杂的 3D 图形渲染、复杂的数学变换（比如傅里叶变换、矩阵变换等），都可以集成在显卡上。这样操作系统在运行的时候，就可以直接把这些消耗 CPU 的工作交给显卡完成，把 CPU 从复

杂的图形处理中解脱出来，专注于功能运算。但是要使用这些附加功能，必须有显示卡驱动程序的支持。操作系统定义好一个功能集合，显示卡驱动程序有选择地实现全部或部分预定义的功能集合，然后通知操作系统。这样操作系统在用到特定功能的时候，首先判断显示卡（显示驱动程序）是否支持。如果支持，则交给显示卡驱动程序完成（驱动程序最终驱动硬件完成），如果不支持，则由 CPU 计算完成。

Hello China 也定义了一组基本的功能集合，作为与显示驱动程序的接口。这组功能集合包括画线、画椭圆、矩形填充、区域渲染等，后续可根据需要添加。但支持 Hello China 的显卡毕竟有限，因此大部分情况下，Hello China 还是通过最基本的直接写显存方式，实现 GUI 绘图功能。这显然无法充分发挥显卡的所有潜能，同时也浪费了大量 CPU 的计算能力。但只要不涉及复杂的图形处理和大量的图形变换（比如游戏），这种最简单的处理方式也足够了。况且，Hello China 的定位是面向嵌入式应用的终端类操作系统，这种应用场合下，硬件配置都是已知的，且需要针对硬件做功能裁剪和定制。在这种与硬件有紧密结合的情况下，可完全避免在 PC 上出现的硬件资源利用不充分的问题。

11.3 对显示设备的封装——Video 对象

11.3.1 GUI 模块的分层架构

作为操作系统的关键组成部分，GUI 必须按硬件无关性设计，以便能够支持多种多样的显示硬件。如果不做任何抽象，按照 VBE 标准取得显卡的显存地址，直接进行显存写操作，这样的可移植性显然不高。毕竟 VBE 是基于 PC 的标准，其他的非 PC 硬件平台一般不支持这个标准。

把一个功能进行层次化划分，定义层次之间的接口，是实现可移植性的比较好的选择。但仅仅用层次化的方式隔离硬件实现和软件实现，也存在一些局限，比如不能同时支持多种硬件等。因此 Hello China 的 GUI 模块在设计的时候，综合利用了层次化的划分方法和面向对象的抽象方法，定义了一个名为 Video 的抽象对象，来实现对硬件的管理，同时实现软硬件的有效隔离。图 11-1 说明了这个架构。

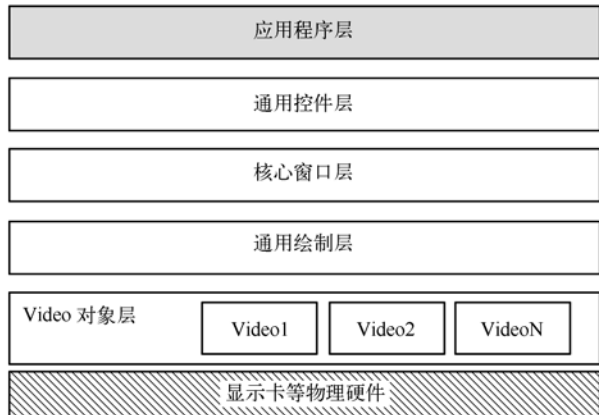


图 11-1 GUI 模块的分层架构

每个层次的含义如下：

(1) 显示卡等物理硬件层：这是纯粹的物理硬件设备，用于提供实际的显示和绘制服务。

(2) Video 对象层：系统中的每个显示设备，比如显示卡等，都会被抽象成一个 Video 对象，这样就可以适配系统中有多个显示设备的场景。由 Video 对象来实际操作对应的硬件，并向上层呈现统一的功能接口。Video 对象层提供多个 Video 的管理功能，比如管理哪个 Video 对象是当前的默认输出对象，在有多个 Video 对象的时候，是采用镜像输出机制（每个 Video 对象输出相同的内容），还是采用拼接输出机制（每个 Video 输出部分内容，所有 Video 的输出，组成完整画面）。在 V1.75 的实现中，只能支持一个 Video 对象的输出，但后续版本可扩展到多个 Video 对象的同时输出和管理。

(3) 通用绘制层，该层是一个封装层，封装 Video 对象层的各个对象提供的功能，为核心窗口层提供统一的绘图服务。之所以增加这个层次，是因为在 Video 对象层中，可能存在许多个 Video 对象，而这些不同的 Video 对象所提供的服务可能不同。这样为了协调一致，对更上层呈现一个统一的调用界面，所以增加了这一层进行适配。

(4) 核心窗口层，实现最核心的窗口机制。比如窗口的创建、绘制、销毁等操作，窗口消息的缺省处理、窗口的刷新等功能。这个层次是整个 GUI 部分的核心。

(5) 通用控件层，即实现通用控件功能的层次。所谓的通用控件，就是组成 GUI 界面的按钮、菜单、编辑框、对话框等界面要素。这个层次基于核心窗口层功能，实现了很多预定义的通用控件，通过更加简洁的接口，提供给上层的应用使用。

(6) 应用程序层，这就是具体的用户应用程序所在的层次。这个层次的代码，可以调用通用控件层提供的功能函数，来构筑基于应用的用户界面。这个层面完全是由应用程序实现的，严格来说，不算是操作系统 GUI 模块的一部分。

每个层次都是按照面向对象的思想，把相关功能划分为一个一个的对象来分别实现。比如核心窗口层，就通过一个 WindowManager 的核心对象来实现，统筹管理所有最底层的窗口功能。

接下来重点介绍 Video 对象层的功能和实现，其他层次会陆续展开描述。

11.3.2 Video 对象

Video 对象层中的主要对象就是 Video 对象，系统中的每个显示设备，对应一个 Video 对象。一个典型的例子就是，在个人计算机中，会有显示器设备、打印机设备、投影仪设备等。每个这样的设备，对应一个显示对象。显示对象的具体实现是由设备的驱动程序完成的，在实现 Video 对象的时候，具体的功能代码，与硬件关系紧密，不同的 Video 对象会存在较大差异。但是每个 Video 对象的实现，必须按照预先定义的接口进行，否则无法被更上层（通用绘制层）调用。

对程序员来说，解释概念的最直观方法，就是展示代码。下面是 Video 对象的预定义代码：

```
[gui/include/video.h]
struct __VIDEO{
    DWORD          dwScreenWidth;    //Screen width.
```

```

DWORD      dwScreenHeight;    //Screen height.
DWORD      BitsPerPixel;      //Color bits for one pixel.
LPVOID      pBaseAddress;      //Base address of display memory.
DWORD      dwMemLength;        //Length of display memory.

BOOL        (*Initialize)(__VIDEO* pVideo);
VOID        (*Uninitialize)(__VIDEO* pVideo);

VOID (*DrawPixel)(__VIDEO* pVideo,int x,int y,__COLOR color);
__COLOR (*GetPixel)(__VIDEO* pVideo,int x,int y);
VOID (*DrawLine)(__VIDEO* pVideo,int x1,int y1,int x2,int y2,__COLOR color);
VOID (*DrawRectangle)(__VIDEO* pVideo,int x1,int y1,int x2,int y2,
                      __COLOR lineclr,BOOL bFill,__COLOR fillclr);
VOID (*DrawEllipse)(__VIDEO* pVide,int x1,int y1,int x2,int y2,__COLOR color,
                   BOOL bFill,__COLOR fillclr);
VOID (*DrawCircle)(__VIDEO* pVideo,int xc,int yc,int r,__COLOR color,BOOL bFill);
VOID (*MouseToScreen)(__VIDEO* pVideo,int x,int y,int* px,int* py);
};

```

当前的定义比较简单，基本上是按照 flat display memory 的模型来定制的。但是可以对其进行扩展，添加更多的功能。重点关注下列几个预定义函数：

(1) Initialize/UnInitialize 函数，其中第一个是在 GUI 模块初始化的时候被调用，第二个则是在 Video 对象被卸载的时候调用。对显示卡来说，应该在 Initialize 函数中，对硬件进行初始化，获得硬件的相关信息，并把关键的信息填到 Video 的几个变量中。最主要的是 pBaseAddress、BitsPerPixel 等。这些变量的含义都是自解释的。

(2) DrawPixel、DrawLine 等函数。这些函数实现了最基本的绘制功能，上层模块通过调用这些功能，实现更复杂的绘制操作。DrawPixel 和 GetPixel 是必须实现的，其他诸如画线、画椭圆等，可选择实现。如果实现了，GUI 的通用绘制层会直接调用，如果没有实现，通用绘制层会通过调用 DrawPixel 函数来自行实现，这时候的效率，可能不如 Video 对象实现的高。因为 Video 对象在实现的时候，可以调过自身的硬件机制来实现绘制功能，而通用绘制层的实现，则完全是基于软件的。通过这里的描述，读者会更进一步理解显示卡对 CPU 在图形操作上的处理分担 (offload) 功能。

(3) MouseToScreen 函数，这是必须实现的。这个函数把鼠标的坐标数值，换算成屏幕的坐标数值。因为通常情况下，鼠标的横纵坐标最大为 255，而屏幕的分辨率则是变化的。我们在操作窗口元素的时候，是以屏幕坐标为基础的，而用户输入，则是以鼠标坐标为基础。这样就必须实现这两者之间的转换。

在 Hello China V1.75 的实现中，只有一个 Video 对象——全局 Video 对象。该对象对应屏幕显示设备。这个对象直接定义在源代码中，这样其他层次（比如通用绘制层）就可直接调用它提供的功能方法了：

```

[gui/video/video.cpp]
__VIDEO Video = {
    1024,          //dwScreenWidth.

```

```

768,          //dwScreenHeight.
32,           //BitsPerPixel.
NULL,         //pBaseAddress.
0,            //dwMemLength.
Initialize, //Initialize routine.
Uninitialize, //Uninitialize.
DrawPixel,    //DrawPixel.
GetPixel,     //GetPixel.
DrawLine,     //DrawLine.
DrawRectangle, //DrawRectangle.
NULL,         //DrawEllipse.
DrawCircle,   //DrawCircle.
MouseToScreen, //MouseToScreen.
};

```

上面的定义，初始化了 Initialize 等函数，但是对于 pBaseAddress 等，并没有设置。这些变量的设置，是在 Initialize 函数中实现的。下面详细解释 Initialize 函数的实现。

Initialize 函数的实现代码如下：

```

[gui/video/video.cpp]
static BOOL Initialize(__VIDEO* pVideo)
{
    __VBE_INFO* pVbeInfo      = (__VBE_INFO*)VBE_INFO_START;
    LPVOID      pBaseAddr      = NULL;
    LPVOID      pPhysAddr      = NULL;
    BOOL         bResult        = FALSE;

    if(!SwitchToGraphic()) //Can not switch to graphic mode.
    {
        PrintLine(" Can not switch to graphic mode.");
        PrintLine(" Please make sure the following graphic display mode is available.");
        PrintLine(" Demension: 1024 * 768,32 bits true color.");
        goto __TERMINAL;
    } //------(1)

    pPhysAddr = (LPVOID)pVbeInfo->ModeInfo.physbaseptr;
    pBaseAddr = VirtualAlloc(pPhysAddr,
        DISPLAY_MEMORY_LENGTH,
        VIRTUAL_AREA_ALLOCATE_IO, //Allocate flags.
        VIRTUAL_AREA_ACCESS_RW, //Access flats.
        "VIDEO");
    if(pBaseAddr != (LPVOID)pVbeInfo->ModeInfo.physbaseptr)
    {
        goto __TERMINAL;
    } //----- (2)
}

```

```
pVideo->pBaseAddress    = pBaseAddr;
pVideo->dwMemLength      = DISPLAY_MEMORY_LENGTH; //-----(3)
//Now clear the screen.
ClearScreen(pVideo);
bResult = TRUE;

__TERMINAL:
if(!bResult) //Failed to initialize video object,release all resources.
{
    if(pBaseAddr)
    {
        VirtualFree(pBaseAddr);
    }
}
return bResult;
}
```

Video 对象初始化的过程比较简单，主要有三个关键点，代码中分别用（1）、（2）、（3）表示。分别介绍如下：

（1）首先调用 SwitchToGraphic 函数，试图切入图形模式的 0x118 显示模式。该函数首先切换到实模式，然后通过 0x10 号中断调用，设置显示卡工作模式为 0x118，并获取对应模式的相关信息，存储到 VBE_INFO_START 定义的内存处，然后再切换回保护模式。如果一切操作都是成功的，则返回 TRUE，任何一个环节失败就返回 FALSE。如果这个函数失败，则直接导致 Video 对象的 Initialize 函数失败，从而导致 GUI 模块加载失败。这里需要注意的是，Hello China 在加载 GUI 模块的时候，就已经进入保护模式了。这时若要调用 0x10 BIOS 中断设置显示模式，则必须重新返回实模式。

（2）SwitchToGraphic 函数成功后，说明已经切换到 0x118 号的图形模式，该模式的相关信息被保存在 VBE_INFO_START 定义的内存地址处。该处存放了一个 VBE_MODE_INFO 结构的信息块，其中的 physbaseptr 变量，就是我们关注的 flat display memory 的起始地址。需要注意的是，这个地址是显卡显存的物理地址，这个地址由 BIOS 在初始化显卡时设置。缺省情况下 Hello China 启用了 VMM（虚拟内存管理，详情可参考第 5 章）功能，任何一块可用的内存空间，必须经 VMM 管理器建立对应的页目录和页表才能访问，否则会引发异常。因此在获得该地址后，必须调用 VirtualAlloc 函数，来为显存建立页目录和页表。关于 VirtualAlloc 函数的使用，在第 5 章有详细介绍。正常情况下应该不会失败，因为显存地址不会被占用。但是也有可能出现显存地址被预先占用的情况，这时候就会导致 VirtualAlloc 失败，从而导致 Video 初始化失败。如果读者对 VirtualAlloc 的使用方法不了解，也无需在这里花费过多时间，可以先认为这一步不存在，以免影响对 GUI 部分的理解。

（3）另外一个需要设置的是显存的大小。在 0x118 模式下，显存的大小是 3MB。正常情况下，显存的大小也应该是从 VBE 信息块中获取的。但是为了实现上的方便，直接用预定义的常数 DISPLAY_MEMORY_LENGTH 来初始化显存大小。一般情况下，这是没有问题的。

初始化完成之后，Video 对象就可以被更上层的应用代码直接使用了。在当前版本的实现中，Video 对象的功能比较简单，只实现了画点、画线、画圆等基础函数。下面是画点函数的实现，非常简单。

```
[gui/video/video.cpp]
VOID DrawPixel(__VIDEO* pVideo,int x,int y,__COLOR color)
{
    CHAR*      pAddress = NULL;
    pAddress = (CHAR*)pVideo->pBaseAddress;
    pAddress += ((y * pVideo->dwScreenWidth) + x) * pVideo->BitsPerPixel / 8;
    *(DWORD*)pAddress = color;
}
```

该函数的功能是在屏幕坐标 (x,y) 处，画一个颜色是 color 的点。函数首先计算出这个坐标所对应的显存地址，然后把颜色代码写入显存即可。这样显卡硬件就会在 (x,y) 处点亮对应的颜色。

其他函数，比如画线函数，则是使用了一些成熟的计算机图形学算法，调用 DrawPixel 来实现画图功能。如何快速有效地画出一条直线，甚至不使用浮点运算功能，是计算机图形学的重要课题之一。比如比较有名的画线算法，是 Bresenham 算法，这个算法快速且效果好，又不用浮点数支持，非常高效。Hello China V1.75 GUI 的画线和画圆算法，就使用了该算法。这个算法的详细思想，可参考相关资料，或者到网上搜索。

到此为止，GUI 部分与硬件相关的内容介绍完毕，相信读者们已经建立起一个清晰的脉络。此后的所有内容，基本都是硬件无关的，它们只会调用 Video 对象提供的服务，硬件操作完全由 Video 对象屏蔽。为了进一步加深读者理解，我们简单描述一下 GUI 模块加载过程的开始部分，这样会把 Video 对象的初始化等动作所在的位置说明得更加清楚。

Hello China V1.75 的 GUI 模块是在字符 shell 模式下加载的。在字符操作模式下，用户输入 gui 命令并回车后，字符 shell 会在 C:\PTHOUSE 目录下寻找 hcngui.bin 模块。如果找不到，则加载失败，重新回到字符 shell。如果能够找到 hcngui.bin 模块，则字符 shell 会读取该模块到内存，然后进行合法性检查，主要是检查该模块的开始部分是不是一个合法的 Hello China 外围模块。如果检查失败，则放弃加载，返回字符 shell。如果检查顺利通过，则字符 shell 会以 hcngui.bin 的起始地址为入口点，创建一个名字为“GUI”的核心线程，然后等待该线程运行结束（通过调用 WaitForThisObject，等待 GUI 线程运行结束）。需要注意的是，字符 shell 本身就是一个核心线程，在完成 GUI 线程的创建后，系统中至少存在三个核心线程：字符 shell 线程、GUI 线程、空闲 idle 线程。

GUI 线程会马上被调度运行，这时候正式进入 GUI 模块的初始化过程。下面是初始化的部分代码：

```
[gui/guientry.cpp]
DWORD __init(LPVOID)
{
    HANDLE      hRawInputThread = NULL; //RAW input thread handle.
```



```
HANDLE    hLastFocusThread  = NULL;  //Hold last focusable thread.
HANDLE    hGUIShell         = NULL;  //Shell thread in GUI mode.
BOOL      bResult           = FALSE;
BOOL      bVideoOK          = FALSE;

//Initialize the Video object.
if(!Video.Initialize(&Video))
{
    bVideoOK = FALSE;
    goto __TERMINAL;
} //----- (1)
bVideoOK = TRUE;
//Initialize the GlobalParams object,system level variables are held by this object.
if(!GlobalParams.Initialize(&GlobalParams,&Video))
{
    goto __TERMINAL;
}
//Initialize the WindowManager object.
if(!WindowManager.Initialize(&WindowManager))
{
    goto __TERMINAL;
}

//Register system calls for GUI module.
if(!RegisterSystemCall(SYSCALL_GUI_BEGIN,SYSCALL_GUI_END,SyscallHandler))
{
    goto __TERMINAL;
}

//Now create RAWIT thread to receive and dispatch all events(input) in GUI mode.
hRawInputThread = CreateKernelThread(
    0, //Use default stack size.
    KERNEL_THREAD_STATUS_READY,
    PRIORITY_LEVEL_HIGH,
    RAWIT,
    NULL,
    NULL,
    "GUIRAWIT");
if(NULL == hRawInputThread) //Can not create the RAW input thread.
{
    goto __TERMINAL;
}

... ..
```


上述 Init 函数即是 GUI 模块的初始化函数，也是 GUI 模块被加载后调用的第一个函数。该函数首先初始化 Video 对象（在上述代码位置（1）处），如果初始化失败，则直接进入出错处理过程，取消 GUI 的进一步初始化。代码中 Video 对象的 Initialize 函数，就是前面部分讲解的 Video 初始化函数。在这个函数内，完成切换到图形模式、初始化显存地址、为显存分配页目录和页表等工作。

如果 Video 对象初始化成功，则继续初始化系统中的其他对象，比如窗口管理器对象等。这些对象的详细初始化过程，在后续章节中会一一展开。全局对象初始化完毕，GUI 模块会创建几个核心线程如 GUIRAWIT 线程、GUISHELL 线程等，这些线程的用途，会在后面详细介绍，现在不必理会。核心线程创建完毕，GUI 线程就会进入等待状态（等待 GUIRAWIT 等线程运行结束），正式的 GUI 功能由 GUIRAWIT、GUISHELL 等线程完成。一旦 GUIRAWIT 等核心线程运行结束（由用户驱动，比如用户在图形模式下按了“CTRL+ALT+DEL”组合键，就会导致 GUIRAWIT 线程自然结束），GUI 线程会重新被调度运行，于是就进行资源清除工作，然后正式退出图形模式。

这里提到的一些线程，可能会让读者糊涂，图 11-2 清晰地说明了这些线程的等待关系。因为这些线程非常重要，所以读者必须搞清楚它们之间的关系，否则理解 GUI 的后续内容将存在一定困难。

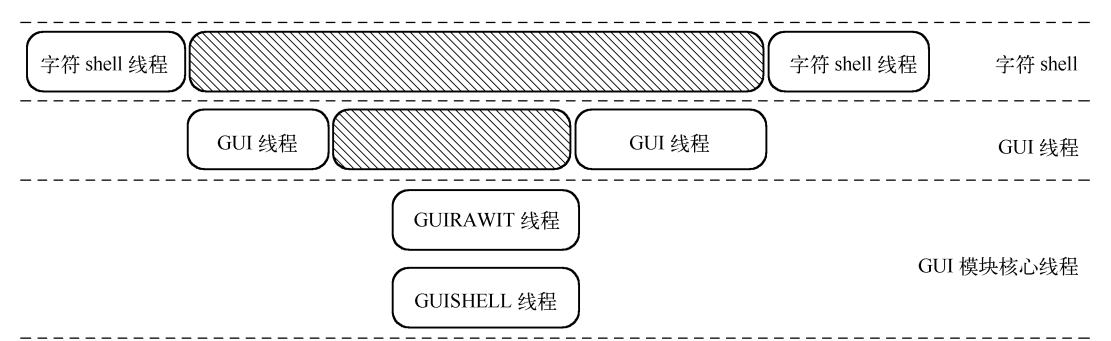


图 11-2 GUI 关键线程生命周期示意

图中阴影部分指的是线程处于阻塞等待状态，非阴影部分指的是线程处于正常运行状态。这种以核心线程为基础的设计方式，可有效地把 GUI 这个复杂的功能模块，分解为相互独立又相互协作的独立功能模块，具备很强的伸缩性和灵活性。

11.3.3 通用绘制层简介

通用绘制层是为了弥补 Video 对象功能不一而设置的。设想，有的 Video 对象自己实现了画椭圆算法，而有的 Video 对象则没有。这时候就需要通用绘制层来弥补这两者之间的差异了。如果 Video 对象实现了画椭圆功能，则通用绘制层会直接调用 Video 的画椭圆功能来完成椭圆绘制。否则，通用绘制层会使用自己的绘制算法，但是在绘制每个点的时候，仍然会调用 Video 对象的 DrawPixel 函数。当然，画点函数 DrawPixel 和 MouseToScreen 等功能，是每个 Video 对象都必须实现的。

通用绘制层为更上层功能（如核心窗口层）提供统一的绘制服务。表 11-2 是通用绘制



层的一些最基本绘制函数。

表 11-2 通用绘制层的基本绘制函数

函数名称	原型	作用
DrawPixel	VOID DrawPixel(_VIDEO* pVideo, int x, int y, _COLOR color);	在指定的位置 (X,Y) 处, 画一个颜色是 color 的点。该函数直接调用了 Video 的画点函数来实现
GetPixel	_COLOR GetPixel(_VIDEO* pVideo, int x,int y);	获取指定点 (X,Y) 处的颜色。该函数实际上就是读取显存, 获得指定点的当前颜色
DrawLine	VOID DrawLine(_VIDEO* pVideo, int x1,int y1,int x2,int y2,_COLOR c);	画一条从(x1,y1)开始, 到(x2,y2)结束, 颜色是 c 的线段。该函数首先判断 pVideo 对象是否实现了画线功能。如果实现了, 则直接调用, 否则使用通用绘制层的算法来画线
DrawRectangle	VOID DrawRectangle(_VIDEO* pVideo, int x1,int y1,int x2,int y2, _COLOR lineclr,BOOL bFill,_COLOR fillclr);	画一个左上角坐标为(x1,y1), 右下角为(x2,y2)的矩形, 矩形的边框颜色是 lineclr, 如果填充 (bFill 为 TRUE), 则使用 fillclr 填充矩形内部
DrawEllipse	OID DrawEllipse(_VIDEO* pVide,int x1,int y1,int x2,int y2,_COLOR color, BOOL bFill, _COLOR fillclr);	在矩形(x1,y1)、(x2,y2)内画一个椭圆, 椭圆边框的颜色为 color, 如果要填充, 则使用 fillclr 填充椭圆内部

通过这些简单的函数, 读者可进一步理解通用绘制层为上层提供的绘制服务, 同时也可以更深入地理解通用绘制层如何调用 Video 对象层的功能, 来实现绘制功能。

需要说明的是, 通用绘制层功能的当前实现, 是存在缺陷的, 就是没有考虑剪切域的问题。通用绘制层当前的实现, 是可以在屏幕上任意位置进行输出的, 只要你指定一个具体位置。但在很多情况下, 是需要把输出限制在一个特定区域内的, 比如一个固定窗口内。超出窗口的输出, 将会被剪切掉。目前为了实现方便, 没有在通用绘制层考虑剪切的问题, 后续版本中会增加对剪切域的支持。即使没有剪切域的支持, 目前的绘制功能也是比较完备的, 可以满足大多数情况的需要。

GUI 模块的绘制机制已解释完毕, 相信读者应该有一个比较清晰的脉络了。在此做一下总结:

(1) 通过直接写显存的方式, 实现对显卡的操作。使用 VESA 的 VBE 标准, 对显卡进行预设置 (设置显示模式为 0x118), 同时获取其显存的地址。

(2) 通过 Video 对象来封装硬件, 隔离硬件和软件。系统中的每个显示设备对应一个 Video 对象, 每个 Video 对象可实现部分或全部预先定义的绘制功能, 但 DrawPixel 和 MouseToScreen 这两个函数是必须实现的。

(3) 为了弥补不同 Video 对象的能力差异, 引入了通用绘制层来适配。通用绘制层调用 Video 对象来实现绘制功能, 同时向更上层提供一致的绘制接口。

(4) 更上层的代码, 比如窗口管理代码、通用控件代码、用户应用程序代码等, 都是直接或间接调用通用绘制层的功能完成屏幕视频输出的, 不会直接接触硬件。当然, 如果应用程序希望直接操作硬件, Hello China 操作系统也不会阻止。

图 11-3 是 Hello China V1.75 版自带的一个小程序 CPI 统计程序的运行结果。这个程序就是综合运用了通用绘制层功能, 实现的一个演示程序。

下面进入另外一个主题: 用户输入消息在窗口之间的传递, 这也是 GUI 功能的最核心机制。

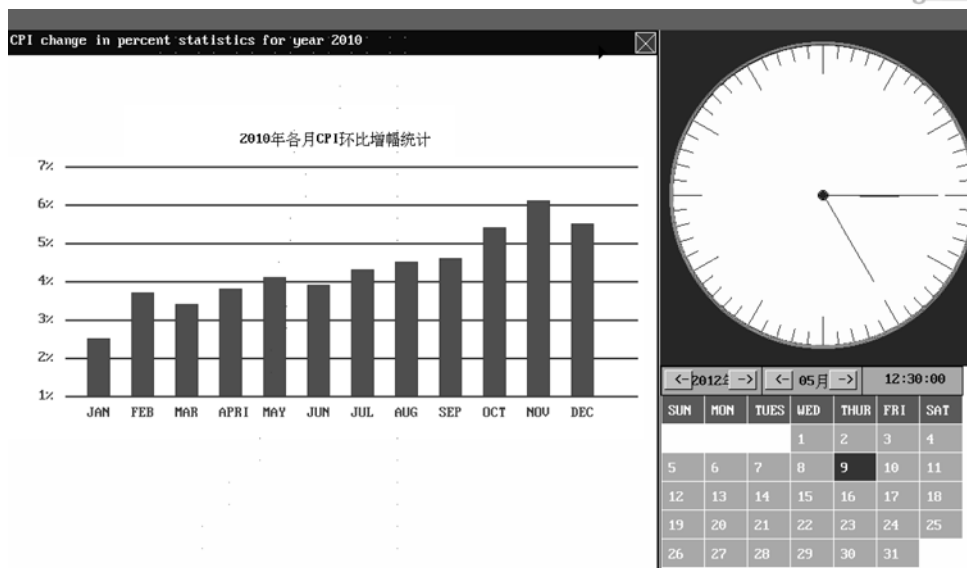


图 11-3 CPI 统计程序的运行结果

11.4 鼠标指针的实现

在介绍消息传递机制之前，先简单介绍一下 GUI 模块的另外一个重要内容——绘制鼠标指针。鼠标指针的绘制，不但与主动绘制机制有关，而且还与消息传递机制有关联，而消息传递机制是接下来的重点内容。但鼠标绘制的大部分工作，还是绘制本身，因此我们放在这里进行讲解。其中涉及的消息传递机制及 RAWIT 等线程，暂时不用理会，阅读本章后续内容后，读者自然会明白。

在图形系统中，需要通过鼠标指针动态跟踪鼠标的位置，来实现鼠标的点击输入。在 Hello China 当前版本的实现中，是通过下列简单的方式实现的：

- (1) 在 RAWIT 线程中，实时处理鼠标移动事件 (Mouse Move)，在鼠标移动事件中，画出鼠标指针。
- (2) 在画鼠标指针前，首先保存被鼠标图标覆盖的当前屏幕区域，使鼠标在更换位置的时候，能够恢复窗口原有位置的信息。
- (3) 保存鼠标覆盖区域后，再通过画点的方式，把鼠标图形画到屏幕上。
- (4) 在鼠标移出当前位置的时候，恢复原来保存的屏幕信息。

鼠标指针的处理，都是在 RAWIT 线程中实现的。这样可实现如下功能：

- (1) 鼠标指针的处理，跟实际应用程序无关，是系统级别的处理。这样即使应用程序出现故障，也不会影像鼠标在屏幕上的移动。

(2) 鼠标指针是通过 Video 对象的画点函数 (DrawPixel) 来实现的。而所有使用 GUI 功能的应用程序，也都是调用 Video 对象提供的函数来画出窗口，因此可处理动态屏幕更新的情况：鼠标覆盖住的屏幕位置是动态变化的。比如，鼠标覆盖了一个不断变化的计数器，在鼠标覆盖期间，计数器的数值已经变化。若鼠标移开以后，仍然按照鼠标进入的内容恢复

屏幕，则会出现内部不一致的问题。解决这个问题的办法，就是通过 Video 对象，实时更新鼠标位置。每次接收到屏幕更新需求，Video 对象首先判断该更新对象是否在鼠标覆盖位置。若不是，则直接更新，否则，需要在更新数据后，同时保存屏幕更新的数据，并再次画出鼠标指针（鼠标指针永远位于所有屏幕内容之上）。

图 11-4 是 Hello China V1.75 版本的 GUI 模块中，鼠标的指针形状。

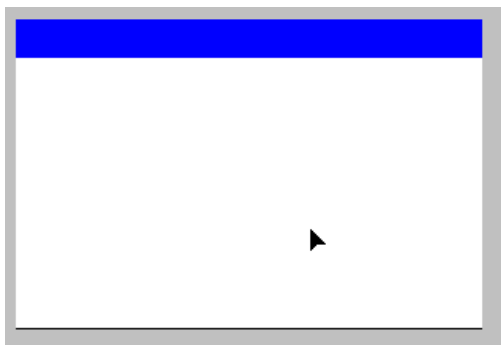


图 11-4 GUI 模块的鼠标指针形状

鼠标实际上是一个 16×16 像素的图标，这 16×16 个像素，并不是都要画出来的，而只是画出需要的一些像素，反映出一个箭头形状即可。因此，采用一个 bit 数组，来指明这个 16×16 像素的方块中，哪些像素需要画出。如下：

```
static int MouseMap[16][16] = {
    {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0},
    {1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0},
    {1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
};
```

在上面这个二维数组中，只有标志为 1 的像素，需要画出。画鼠标指针的代码如下所示：

```
static VOID DrawMouse(__VIDEO* pVideo,int x,int y)
{
```

```
int i,j;
//Save the rect occupied by mouse first.
SaveMouseRect(pVideo,x,y);
for(i = 0;i < 16;i ++)
{
    if(y + i >= (int)pVideo->dwScreenHeight)
    {
        break;
    }
    for(j = 0;j < 16;j ++)
    {
        if(MouseMap[i][j]) //Should draw.
        {
            if(x + j >= (int)pVideo->dwScreenWidth)
            {
                break;
            }
            DrawPixel(pVideo,x + j,y + i,COLOR_BLACK);
        }
    }
}
}
```

在上述代码中，x 和 y 是当前鼠标指针的位置，以 x、y 坐标为鼠标指针的左上角，画出鼠标指针。该函数在执行前，首先调用 SaveMouseRect 函数，保存了鼠标矩形所覆盖的屏幕信息，以便后续恢复。SaveMouseRect 的代码如下：

```
static VOID SaveMouseRect(__VIDEO* pVideo,int x,int y)
{
    int i,j;
    for(i = 0;i < 16;i ++)
    {
        if(y + i >= (int)pVideo->dwScreenHeight)
        {
            break;
        }
        for(j = 0;j < 16;j ++)
        {
            if(x + j >= (int)pVideo->dwScreenWidth)
            {
                break;
            }
            if(MouseMap[i][j])
            {
                MouseRect[i][j] = GetPixel(pVideo,x + j,y + i);
            }
        }
    }
}
```

```
}  
}
```

上述代码与 DrawMouse 代码类似，就是根据鼠标位图（MouseMap），来保存特定的屏幕信息，而不是保存整个 16×16 像素大小的矩形。

这样就很容易实现鼠标指针的移动了：

```
static VOID DoMouseMove(int x,int y) //x and y is the coordinate of mouse.  
{  
    static int xppos = 0; //Previous position of x.  
    static int yppos = 0; //Previous position of y.  
    int xpos,ypos;  
  
    MouseToScreen(&Video,x,y,&xpos,&ypos);  
    RestoreMouseRect(&Video,xppos,yppos); //Restore previous screen rectangle.  
    DrawMouse(&Video,xpos,ypos); //Draw mouse in the new location.  
    xppos = xpos;  
    yppos = ypos;  
}
```

上述代码中，首先把鼠标位置转换为屏幕位置（因为屏幕分辨率是可以变化的，而鼠标的坐标范围却一直固定），然后调用 RestoreMouseRect 函数，恢复被鼠标覆盖的矩形。最后再调用 DrawMouse 函数，在新的位置上画出鼠标指针。

11.5 窗口消息传递机制概述

一个完整的 GUI 给人的初步印象是：具备漂亮的颜色搭配，具备功能繁多的窗口控件，具备各种各样的字体，等等。貌似这些漂亮的外观，就是 GUI 的最主要最核心内容。其实不然，不能否认这些漂亮的外观要素是 GUI 的重要组成部分，但这些功能只占据了 GUI 三个字母中的一个“G”，即 Graphic。另外两个，UI，即 User Interface，才是 GUI 的本质。所谓 User Interface（用户接口），本质上是连接计算机和人的一种途径和手段，完成人与计算机的交互，其本质是交流和沟通。既然是一种沟通，要做到有效，必须能够“相互理解”。无法相互理解的沟通是无任何意义的。“相互理解”体现出两层意思：

（1）人要理解计算机的输出，这需要通过人的学习完成，不是我们讨论的目标。我们的目标是计算机操作系统，不是人的思想，这比计算机操作系统复杂多了。

（2）同样，计算机要正确理解人的意图。人通过鼠标、触摸屏、键盘等手段把自己的意图告诉计算机，计算机要能够正确理解这些输入，并把输入传递给适当的程序，由程序做出处理。这是我们讨论的重点。

正确地理解人的意图，根据人的意图做出正确的回馈，是计算机的核心工作理念。这里进一步又含有两个意思：

（1）正确理解人通过输入所表达的意图，然后把这个意图传递给正确的程序进行处理。

（2）计算机程序根据人的输入，准确地做出处理，并把结果反馈给人。

第二条任务，是由应用程序完成的，在处理过程中，会调用操作系统的功能，尤其是反

馈处理结果的时候，会调用操作系统的 GUI 等绘制功能，输出绘制结果。这些绘制等输出机制，在前面的章节中已有介绍，这里不做重复。

这里重点讲解第一步工作的完成机理。这一步完全是由操作系统完成的，把第一步再分解一下，也包含两个意思：

(1) 正确地理解人通过输入所表达的意图。翻译成更加技术化的语言，就是正确地识别人通过触摸屏、鼠标、键盘等的输入动作，识别出是双击还是单击操作，双击或单击的位置是什么地方，是要打开一个程序，输入一串数字，还是要关闭一个窗口。

(2) 正确识别人的输入意图后，需要把这个意图传递给正确的应用程序。比如用户点击了一个窗口的关闭按钮，操作系统正确地识别出这是一个关闭请求后，需要把这个关闭请求发送给窗口所属的进程（应用程序）。这就是把输入传递给应用程序的过程。

上述两项工作，表面看起来非常简单直观，但真正实现起来，却并不简单。且这些工作是操作系统 GUI 模块的最核心内容。下面将以 Hello China V1.75 的 GUI 模块为例，详细进行说明。虽然是以 Hello China 为例的，但这些原理和机制都是通用的，可以扩展到任何一个实现了 GUI 的操作系统。

11.6 Hello China 的窗口机制

窗口几乎是任何一个 GUI 的核心元素，Hello China 也不例外。在介绍更深入的内容之前，有必要简单介绍一下 Hello China 的窗口机制。为了实现上的简单和方便，Hello China 采用了比 Windows 等通用操作系统更加简化的窗口机制，这种简化的机制，在智能手机操作系统（iOS 和 Android 等）上有广泛应用。具体来说，主要有以下简化内容。

11.6.1 父窗口要完全包含子窗口

在 Windows 等通用操作系统中，一个窗口的子窗口，是可以位于该窗口覆盖区域之外的，如图 11-5 所示。

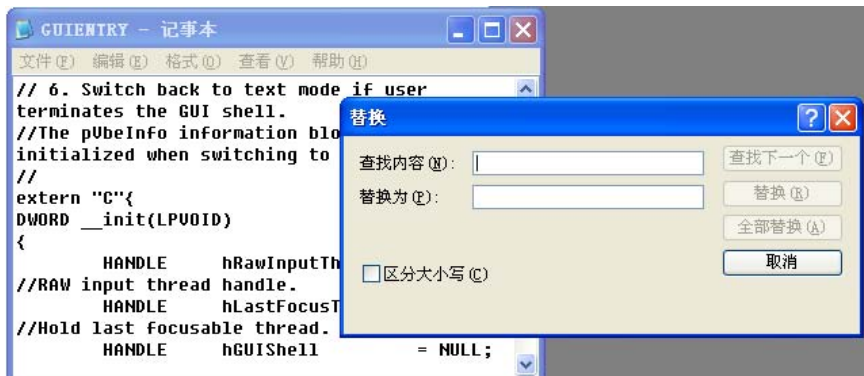


图 11-5 覆盖区域之外的子窗口

其中替换窗口是记事本窗口的子窗口，但是替换窗口的覆盖区域可以超过记事本窗口的覆盖范围。

但为了实现上的简便，Hello China V1.75 的实现中，要求子窗口不能位于父窗口之外。比如，图 11-6 所示窗口关系是合法的。

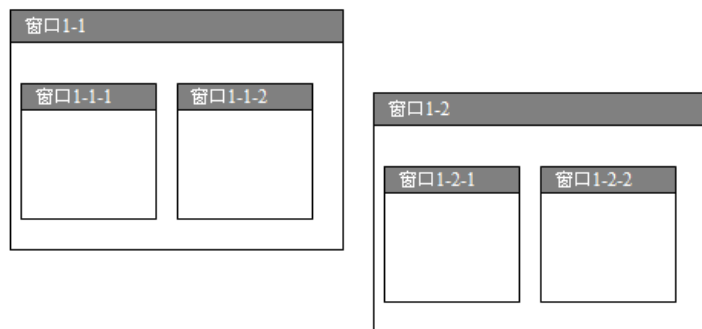


图 11-6 合法的窗口覆盖关系

窗口 1-1-1 和窗口 1-1-2、窗口 1-2-1 和窗口 1-2-2 分别是窗口 1-1 和窗口 1-2 的子窗口，这四个子窗口都位于其父窗口覆盖范围之内。

图 11-7 所示的窗口关系就不符合 Hello China 的要求。

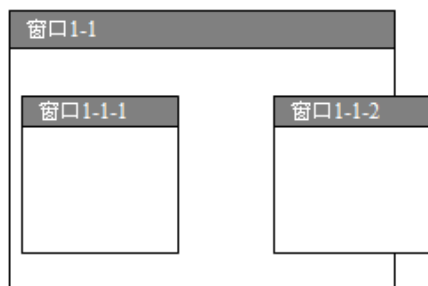


图 11-7 一个非法的窗口覆盖关系

其中子窗口 1-1-2 覆盖了父窗口 1-1 之外的区域。

同时，兄弟窗口之间也不能出现重叠，比如图 11-8 所示窗口关系，就是非法的，因为两个兄弟窗口窗口 1-1 和窗口 1-2 之间出现了重叠。

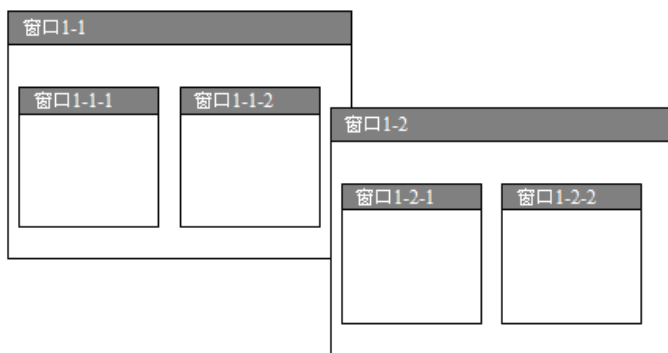


图 11-8 另一个非法的窗口覆盖关系

通过这种窗口关系的简化,可以使很多处理过程得到简化,最主要的是:

- (1) 整个系统只需要一棵窗口树就可管理全部窗口,无需再维护一个窗口 Z 序的列表。
- (2) 窗口变化时的刷新机制非常简单,只需要考虑父窗口的更新即可,无需考虑系统中的所有窗口。
- (3) 避免了复杂窗口剪切域的实现,使得窗口剪切基于简单的“二层窗口剪切域”机制(见 11.6.2 节)即可。

正是因为有这样的简单性,这个限制得到了比较广泛的使用。比如很多智能手机操作系统,就是这样处理窗口的。而且从实际应用上,这样也是足够的。层层叠叠、杂乱无序的窗口关系,不但不会提升计算机使用效率,反而会增加操作的复杂性,导致错误率增大。因此即使是微软,也似乎极力推广这种窗口模型。比如其 Office 软件,缺省都是占据整个屏幕,每当打开一个文档,Office 便会为其创建一个单独的窗口,而不是以前广泛使用的一个框架窗口管理多个子窗口的机制(MDI 机制)。

11.6.2 二层窗口剪切域

窗口剪切域在基于窗口的图形界面中,是最核心的机制之一。要实现窗口剪切域,往往涉及比较繁琐的矩形剪裁和合并算法,会消耗很多的内存和 CPU 资源。为避免通用窗口剪切域带来的弊端,简化实现方式,以适应嵌入式系统的特点,Hello China 目前 GUI 模块的实现,采用了一种简化窗口剪切域的方案:二层剪切域。

所谓二层剪切域,指的是一个窗口在变动(创建、移动、撤销等)的时候,只对其父窗口的剪切域进行更新,而不像通用操作系统一样,对所有窗口的剪切域更新。这样可大大降低处理器的消耗和内存的消耗,同时又不会降低 GUI 模块的效果。之所以能够采用二层窗口剪切域,是因为 Hello China 操作系统的窗口机制是基于“父窗口完全涵盖子窗口”的原则的,这样一个窗口的变动,只会对其父窗口造成影响,不会对其他窗口造成影响。

实际上,Windows 操作系统的一些主流应用程序(比如 Microsoft Office 等),也在遵循这种简单的剪切域关系。这些通用的软件,已经彻底摒弃了原有的 MDI 多层次窗口概念,同时程序的工具条默认处于固定位置,这样事实上就是一种二层剪切域的理念。用户可以改变这种缺省的设置,比如建立多个子窗口,随便移动工具条等。但这样做会消耗很多的 CPU 和内存资源。

采用二层剪切域机制,需要遵循下列限制(或规则):

- (1) 兄弟窗口(具备相同父窗口的窗口)之间不能重叠。因为一个窗口的变动,只会更新父窗口的剪切域,不会改变兄弟窗口的剪切域。若两个兄弟窗口重叠,在一个窗口移动或销毁的时候,会导致另外一个兄弟窗口不能完整显示。
- (2) 窗口之间不能“隔代”覆盖。即一个窗口,不能覆盖在其父窗口的父窗口(即祖父窗口)之上。原因也很明显,这个窗口的显示或变化,只会使得其父窗口更新剪切域,而不会使其祖父窗口更新剪切域。这样在窗口变化的时候,会影响其祖父窗口的外观。
- (3) 一个窗口也不能覆盖在其“叔叔”窗口或“伯伯”窗口之上,原因同上。

虽然这些规则会限制窗口系统的使用范围,但在大多数情况下,这种实现是可以满足功能需求的。尤其是在嵌入式系统情形下,各种硬件资源受到限制,显示屏幕尺寸不会太大。即使实现了完整的窗口机制,也不会得到全部的功能发挥。

11.6.3 二层窗口剪切域的实现

窗口剪切域可看作是一系列不相交的矩形的集合，这些集合限制了窗口可以输出的范围。本质上说，剪切域是一系列矩形的集合，凡是位于这个集合内的矩形区域，都是可以输出的。但是不在这个集合内的区域，则不能输出，必须被裁剪掉，否则会影响到其他应用程序的外观。在一个窗口被创建的时候，其剪切域初始化为窗口矩形，客户区的剪切域也初始化为窗口矩形（需要注意，这里不是客户区矩形，窗口客户区的剪切域与窗口的剪切域是完全一致的）。与整个窗口不同的是，客户区的上下文，其限定范围（通过 `x`, `y`, `width` 和 `height` 四个参数限制的矩形）为窗口的客户区，而不是整个窗口。

`__WINDOW_CLIP_ZONE` 对象，定义了窗口剪切域中的一个矩形元素（后面把这个对象称为窗口剪切域元素），如下：

```
[gui/include/clipzone.h]
struct __WINDOW_CLIP_ZONE{
    int x;
    int y;
    int width;
    int height;
    __WINDOW_CLIP_ZONE* pPrev;
    __WINDOW_CLIP_ZONE* pNext;
};
```

这个对象实际上定义了一个矩形。与矩形对象（`RECT`）不同的是，该对象还维护了两个指针，这两个指针把窗口剪切域元素连接到一个双向链表中，这个双向链表构成了窗口剪切域。另外，其名字之所以是 `ZONE` 而不是 `RECT`，是为将来扩展的考虑。剪切域最简单的形态是一系列矩形的集合，但是不排除包含其他非矩形形状，比如椭圆形、三角形等。以后可通过扩展剪切域的定义，来支持其他几何形状。

为了方便实现窗口剪切域元素的合并，在上述双向链表中，剪切域元素是按照 `x` 的大小进行排序的。这样在向窗口剪切域中增加一个元素（即扩大了窗口剪切域的大小）的时候，首先会把该元素按照顺序插入到链表中，然后从插入位置的上一个位置开始，尝试执行一个合并操作。

窗口刚开始被创建的时候，其剪切域对象初始化为窗口的整个矩形。需要注意的是，即使窗口的状态是不显示的，该窗口也是有剪切域的，而且剪切域的状态，与可显示窗口是一样的。对于不可显示窗口的输出禁止操作，是通过设置窗口上下文对象（整个窗口的上下文和客户区上下文）的 `width` 和 `height` 为 0，来达到禁止显示的目的。

一个窗口，只有在下列情形下，才会更新其剪切域：

- (1) 该窗口的子窗口被创建。
- (2) 该窗口的子窗口被销毁（或关闭）。
- (3) 该窗口的子窗口被隐藏（不显示）。
- (4) 该窗口的子窗口被移动。
- (5) 该窗口的子窗口被改变大小。
- (6) 窗口本身被改变大小。

(7) 窗口本身被移动。

上述任何一种情形，都会导致系统做出如下的函数调用，以更新对应窗口的剪切域：

```
BOOL UpdateWndClipZone(HANDLE hWnd,RECT newRect,DWORD dwUpdateMode);
```

其中 `hWnd` 是待更新的剪切域所归属的窗口句柄，而 `newRect`，则是一个新的屏幕矩形。最后一个参数 `dwUpdateMode`，指明窗口剪切域的更新方式，目前定义的几种见表 11-3。

表 11-3 剪切域更新模式及含义

剪切域更新模式	值	含义及使用场合
CZU_MODE_ADD	1	向剪切域中增加一个矩形，在窗口的子窗口被撤销时调用
CZU_MODE_MINUS	2	向窗口剪切域中删除一个矩形，用于有子窗口被创建时
CZU_MODE_MOVE	3	更新窗口剪切域中所有矩形的起始位置，用于窗口被移动位置时
CZU_MODE_SIZE	4	更新窗口剪切域的大小，用于窗口被改变大小时

其中宏定义的前面三个字母（CZU），是 Clip Zone Update 的缩写，用于指明上面表格中定义的几个模式，只用于窗口剪切域的更新例程中。

其中在第三种情形中（CZU_MODE_MOVE），窗口的大小没有改变，改变的只是窗口的位置。这时候，`newRect` 对象指明了新的窗口位置和大小，只不过此时的窗口大小（`width` 和 `height`）与原有窗口一致而已。这种情况下的处理是最简单的。Update WndClipZone 例程只需要遍历整个剪切域元素列表，更新每个元素的起始位置（`x` 和 `y`）即可。

对于窗口大小被改变的情形（CZU_MODE_SIZE），实际上可分解为两个 CZU_MODE_ADD 操作。改变大小后的窗口矩形，减去原有的窗口矩形，可得到两个矩形，如图 11-9 所示。

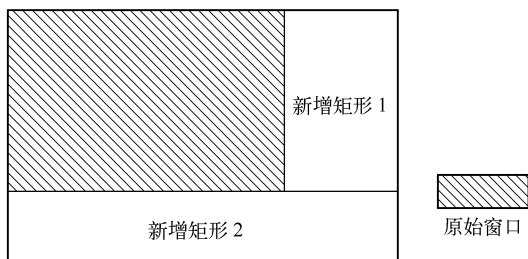


图 11-9 窗口大小改变的剪切操作

这时候只需在窗口剪切域中，执行两次 ADD 操作（加入上图中两个新增的矩形）即可。

因此，对窗口剪切域的更新，最终会抽象成两种操作：ADD（加法）和 MINUS（减法）。

窗口剪切域的加法操作如下：

在向窗口剪切域中加入剪切域元素的时候，除了把元素加入列表中，还需要考虑剪切域的合并操作。下列两种情形下，需要对窗口剪切域元素进行合并，以减少窗口剪切域中的元素数量，降低内存使用率并缩短遍历整个链表的时间，如图 11-10 所示。

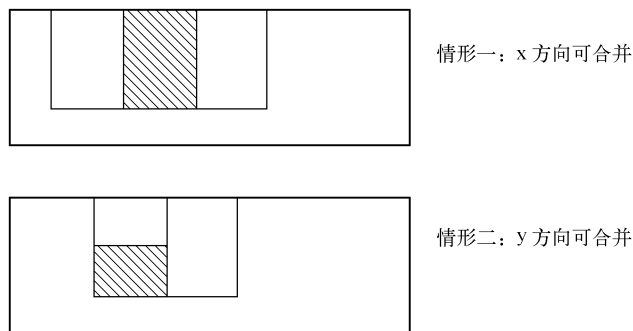


图 11-10 剪切域的加法操作

上述任何一种情形，都需要对连续的两个矩形进行合并。合并操作可抽象为下列两步：

- (1) 从链表中把两个矩形删除。
- (2) 把两个矩形合并后的新矩形，重新插入到链表中。

在重新插入到链表中的时候，有可能又会引发一次合并，比如在图 11-10 中，一个新的矩形插入在两个等高的矩形中间，且刚好占据中间的位置。这样就必须进行另一次合并操作。这实际上是一个递归过程，因此在实现的时候，也是通过递归函数来实现的。

窗口剪切域的减法操作如下：

窗口剪切域的减法操作，实际上也是可以转化为窗口剪切域的加法操作来实现的。如图 11-11 所示。

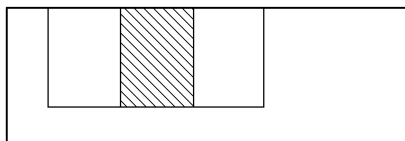


图 11-11 剪切域的减法操作

在图 11-11 中，待减去的矩形（阴影表示）把原有的矩形分成了四个部分（待减去部分、左右各一个小的矩形、下面的一个矩形）。这样在操作的时候，可抽象为下列几步：

- (1) 从剪切域中删除与待减去矩形有交集的矩形。
- (2) 从删除的矩形中，减去目标矩形，这可能会得到另外的多个矩形。
- (3) 把执行步骤 2 中所得到的几个矩形，增加到窗口剪切域中（加法操作）。

11.7 Hello China 窗口机制的实现

窗口机制是任何一个图形用户接口（GUI）所必备的基础机制，Hell China 的 GUI 也不例外，窗口是其最基本的界面元素。通过窗口，可以把多个应用程序的输出同时呈现给用户，用户可以通过切换窗口，实现与不同应用程序的同时交互。

窗口机制的核心，并不是窗口的形状、窗口的颜色等外在特征，而是操作系统如何有效地管理全部窗口，如何准确无误地把用户输入传递给正确的窗口，实现“所见即所得”的目的。

11.7.1 窗口管理器与窗口对象

在 Hello China V1.75 GUI 模块的实现中, 通过一个全局对象——窗口管理器, 对系统中的所有窗口进行管理。窗口管理器本质上是一些全局变量和全局函数的集合, 这些全局变量和全局函数对系统中的窗口进行操作, 比如创建一个窗口、销毁一个窗口等。同时实时记录窗口的状态, 比如窗口的当前位置、当前状态 (是不是焦点窗口) 等信息。下面列举窗口管理器提供的几个常用操作函数, 让读者对窗口管理器的功能有一个总体的认识。下面这些函数的具体应用和注意事项, 暂时先不用深入追究。如果读者对 Windows 的 API 非常熟悉, 就会发现这些函数有“似曾相识”的感觉。当然, 它们的功能, 与 Windows 的 API 还是有差别的。见表 11-4。

表 11-4 窗口管理器函数作用表

函数名称	原型	作用
SetWindowTitle	BOOL SetWindowTitle(HANDLE hWnd, WCHAR* pszTitle);	设置窗口的标题
SetWindowLong	LONG SetWindowLong(HANDLE hWnd, DWORD dwSetFlags; LONG newValue);	设置一个窗口的特定属性, 比如修改窗口的 DefWindowProc 等
ShowWindow	DWORD ShowWindow(HANDLE hWnd, DWORD showFlag);	显示或隐藏窗口, 返回先前的窗口显示或隐藏状态
GetWindowDC	HDC GetWindowDC(HANDLE hWnd, DWORD dcFlag);	获取窗口的上下文句柄, 以对窗口进行输出操作。dcFlag 指出了要获取窗口的哪个区域所对应的 DC
GetWindowStyle	DWORD GetWindowStyle(HANDLE hWnd);	获得窗口的风格信息
SetTimer	HANDLE SetTimer(HANDLE hWnd, DWORD dwTimerId, DWORD dwMillionSeconds, DWORD timerFlags);	设置一个窗口定时器, 返回窗口定时器句柄
CancelTimer	VOID CancelTimer(HANDLE hWnd, HANDLE hTimer);	取消一个窗口定时器
HitTest	DWORD HitTest(HANDLE hWnd, int x,int y);	针对一个屏幕坐标, 判断该屏幕坐标所对应的窗口区域。在处理鼠标点击消息时, 可通过该函数, 判断鼠标的点击位置
DefWindowProc	DWORD DefWindowProc(HANDLE hWnd, UINT msg, WORD wParam, DWORD lParam);	缺省窗口过程。任何未处理的窗口消息, 都需要调用该过程进行处理
GetWindowDC	HANDLE GetWindowDC(HANDLE hWnd);	获得窗口的设备上下文 (DC) 对象, 以便对窗口进行 GUI 输出操作
GetWindowClientDC	HANDLE GetWindowClientDC(HANDLE hWnd);	获取窗口客户区的上下文对象, 以便对窗口的客户区进行 GUI 输出

窗口管理器管理的一个最重要的全局数据结构, 就是窗口树。窗口树是系统中所有窗口对象, 按照其父子或兄弟等逻辑关系, 组合到一起形成的一个内部数据结构。窗口树是窗口机制实现的基础, 也是本章中最重要的内容。系统中的每一个窗口, 无论是由哪个线程创造的, 都会被放在窗口树中的适当位置。任何一个窗口, 都对应一个窗口对象。窗口对象就是一个数据结构, 里面记录了与窗口有关的所有属性。我们通过分析窗口对象的定义, 来解释

窗口树的组织结构。在此之前，先讲解窗口对象中的其他一些相关变量。

窗口的定义如下：

```
[gui/include/wndmgr.h]
struct __WINDOW{
    TCHAR        WndTitle[WND_TITLE_LEN];
    DWORD        dwWndStyle;
    DWORD        dwWndStatus;
    int           x;                //Start position of window.
    int           y;
    int           cx;                //Width of this window.
    int           cy;
    int           xclient;          //Start position of client area.
    int           yclient;
    int           cxclient;         //Client area's width.
    int           cyclient;
    int           xcb;              //x coordinate of close button.
    int           ycb;              //y coordinate of close button.
    int           cxcb;             //width of close button.
    int           cycb;             //height of close button.
    HANDLE        hFocusChild;      //Child window in focus status.
    HANDLE        hCursor;          //Window cursor.
    HANDLE        hIcon;           //ICON of this window.
    HANDLE        hWindowDC;        //Device context of this window.
    HANDLE        hClientDC;        //Device context of this window's client area.
    __COLOR       clrBackground;    //Background color.
    LPVOID        lpWndExtension;    //Window extension pointer.
    __WINDOW_PROC WndProc;          //Base address of window procedure.
    HANDLE        hOwnThread;       //The thread handle owns this window.
    __REGION*     pRegion;          //Clip zone of this window.

    __WINDOW*     pPrevSibling;     //Previous sibling of this window.
    __WINDOW*     pNextSibling;     //Next sibling.
    __WINDOW*     pParent;          //Parent;
    __WINDOW*     pChild;           //Child list header.
    DWORD         dwSignature;      //Signature of window object.
};
```

WndTitle 就是窗口的标题，显示在窗口标题栏中。同时显示在窗口标题栏中的，还有一个关闭按钮。由于 Hello China 采用的是窗口固定的设计方式，即窗口一旦被创建，其大小和位置都不能变化，因此不像 Windows 窗口那样，在窗口标题栏中还有一个最大化和最小化按钮。图 11-12 给出了 Hello China 窗口的相关概念。

窗口对象中的大多数变量，都是用于记录窗口布局的。表 11-5 对窗口对象的部分变量进行了描述。之所以用表格的方式说明，是因为这些变量的含义比较简单，一两句话就可以说清楚。对于含义比较复杂的变量，后面会逐个介绍。



图 11-12 窗口的相关概念

表 11-5 窗口对象的部分变量含义

变 量 名 称	变 量 含 义
WndTitle	窗口标题，最大 64B
dwWndStyle	窗口的风格，比如窗口是否有标题等。可以创建没有标题栏的窗口
dwWndStatus	窗口的状态。比如窗口是否被关闭等
x	窗口左上角在屏幕上的 x 坐标
y	窗口左上角在屏幕上的 y 坐标
cx	窗口的宽度
cy	窗口的高度
xclient	窗口客户区在屏幕上的 x 坐标。客户区指标题以外的区域
yclient	窗口客户区在屏幕上的 y 坐标。客户区指标题以外的区域
cxclient	窗口客户区的宽度，一般与 cx 相同
cyclient	窗口客户区的高度，一般与 cy 相同
xcb	关闭按钮左上角在屏幕上的坐标
ycb	关闭按钮左上角在屏幕上的坐标
cxcb	关闭按钮的宽度
cycb	关闭按钮的高度
hFocusChild	窗口的所有孩子中，处于焦点状态的孩子
hCursor	与窗口关联的鼠标形状。一旦鼠标移入窗口区，就会以该形状显示
hIcon	窗口的图标，实际上是应用程序的图标
clrBackground	窗口背景颜色，用这个颜色填充窗口
lpWndExtension	窗口扩展，可以设置一些应用程序私有的数据，与窗口关联起来
dwSignature	窗口对象签名，用于检查是不是一个合法的窗口对象

11.7.2 窗口函数与窗口消息

窗口函数是本书要介绍的重要窗口属性之一，即窗口对象定义中的 WndProc 变量。这是一个函数指针，定义如下：

```
[gui/include/wndmgr.h]
//Window procedure definition.
typedef DWORD (*__WINDOW_PROC)(HANDLE hWnd,UINT msg,WORD wParam,DWORD lParam);
```

这个函数由应用程序编写人员实现，并在窗口创建的时候，传递给窗口对象。下面是窗口的创建函数：

```
[gui/include/wndmgr.h]
HANDLE CreateWindow(DWORD dwWndStyle,TCHAR* pszWndTitle,int x,
                    int y,int cx,int cy,__WINDOW_PROC WndProc,
                    HANDLE hParent,HANDLE hMenu,__COLOR clrbackground,
                    LPVOID lpReserved); //Create one window.
```

黑体标注的就是窗口函数指针。

窗口函数是实现图形界面用户功能代码的主要位置，所有图形界面相关的操作，比如显示计算结果、接收用户输入等，都是在窗口函数内完成的。这与 Windows 的窗口机制是一样的。下面列举一个窗口函数代码示例，帮助读者加深印象。当然，如果读者是经验丰富的 Windows 程序员，相信这个示例非常简单：

```
[app/hcnhello/hcnhello/hcnmain.cpp]
static DWORD HelloWndProc(HANDLE hWnd,UINT message,WORD wParam,DWORD lParam)
{
    static HANDLE hDC = GetClientDC(hWnd);
    __RECT rect;
    switch(message)
    {
        case WM_CREATE:
            break;
        case WM_TIMER:
            break;
        case WM_DRAW:
            TextOut(hDC,0,0,"Hello,world!");
            break;
        case WM_CLOSE:
            PostQuitMessage(0); //Exit the application.
            break;
        default:
            break;
    }
    return DefWindowProc(hWnd,message,wParam,lParam);
}
```

这个函数处理了 WM_DRAW 消息，其他消息虽然列出来了，但是没有实质性的功能代码。WM_DRAW 消息是系统预定义消息，在窗口需要绘制的时候，系统会给窗口发送该消息，从而导致窗口函数被调用，实现窗口重画的目的。

窗口函数的最后，一定要调用 DefWindowProc 函数，即缺省的窗口函数。许多窗口相

关的处理都是在这个窗口函数内完成的。DefWindowProc 本身是一个窗口函数，其实现代码位于[gui/window/defwproc.cpp]文件中，读者可自行阅读该函数代码。

所谓“给窗口发送消息”，无非就是采用某个特殊的消息（即 message 参数）值来调用窗口函数而已。比如给窗口发送 WM_CREATE 消息，实际上就是以下列形式调用窗口函数：

```
HelloWndProc(hWnd,WM_CREATE,wParam,lParam);
```

这样窗口函数与 WM_CREATE 匹配的 case 语句就会被执行。

Hello China V1.75 的 GUI 模块实现中，表 11-6 所示消息会被发送给窗口。如果读者对某个消息感兴趣，只需在窗口函数的 switch 分支中，添加对应分支，并编写代码即可。当然，所有不在用户定义的窗口函数中处理的消息，都将被 DefWindowProc 函数处理。

表 11-6 窗口消息

消 息 名 称	数 值	含 义
WM_CREATE	1	窗口对象被创建时发送
WM_DESTROY	2	窗口对象被销毁时发送
WM_CLOSE	3	窗口关闭时发送
WM_LBUTTONDOWN	4	鼠标左键被按下
WM_RBUTTONDOWN	5	鼠标右键被按下
WM_LBUTTONDBLCLK	6	鼠标左键双击
WM_RBUTTONDBLCLK	7	鼠标右键双击
WM_SHOW	8	窗口被显示时发送
WM_HIDE	9	窗口被隐藏时发送
WM_SIZE	10	窗口改变大小时发送，当前未实现
WM_VIRTUALKEY	11	有虚拟键（比如 F1 等）被按下时发送
WM_CHAR	12	ASCII 键被按下时发送
WM_KEYDOWN	13	任何一个键盘键被按下时发送
WM_KEYUP	14	任何一个键盘键被释放时发送
WM_MOUSEMOVE	15	鼠标移动时发送
WM_LBUTTONUP	16	鼠标左键抬起时发送
WM_RBUTTONUP	17	鼠标右键抬起时发送
WM_TIMER	18	定时器到时后发送
WM_COMMAND	19	用户点击窗口菜单后引发
WM_NOTIFY	25	窗口的子窗口向父窗口发送的通知消息
WM_DRAW	26	窗口需要重绘时发送
WM_UPDATE	27	窗口更新时发送
WM_SCROLLUP	28	垂直滚动条向上移动时发送
WM_SCROLLDOWN	29	垂直滚动条向下移动时发送
WM_SCROLLLEFT	30	水平滚动条向左移动时发送
WM_SCROLLRIGHT	31	水平滚动条向右移动时发送
WM_CHILDCREATED	33	当有子窗口被创建时发送
WM_CHILDCLOSE	40	当子窗口被关闭时发送

上述窗口消息是一些最基本的窗口消息，不同窗口之间通过相互发送这些消息（实质上是相互调用窗口函数），可实现协调统一的窗口机制。当然，还可以对窗口消息进行进一步扩展，以实现更加丰富的窗口功能。后续版本的 Hello China GUI 实现中，主要通过扩展窗口消息，来添加更多的功能。窗口基础架构更改的必要性不是很大。

11.7.3 窗口归属线程

窗口归属线程，即创建该窗口的线程。在窗口创建的时候（CreateWindow 函数），是无需指定线程对象（或线程句柄）的。但是 CreateWindow 函数会获取当前核心线程的句柄，并存放在窗口对象的 hOwnThread 变量中。

归属线程是实现消息有效传递的基础。用户用鼠标点击了某个窗口的某个位置，这个点击操作首先传递给操作系统。操作系统这时候是不知道用户点击了哪个窗口的，因此它必须搜索窗口树，找到鼠标消息落入的窗口即目标窗口。找到目标窗口后，操作系统还必须确定该窗口到底属于哪个核心线程。只有确定了所属的核心线程，才能向这个线程发送消息。这就是消息传递机制的大概描述，详细内容请参考本章后续相关内容。

下面是 CreateWindow 函数的代码片段，说明了窗口归属线程是如何被设置的：

```
HANDLE CreateWindow(DWORD dwWndStyle, TCHAR* pszWndTitle, int x,
                    int y, int cx, int cy, __WINDOW__PROC WndProc,
                    HANDLE hParent, HANDLE hMenu, __COLOR clrbackground,
                    LPVOID lpReserved)
{
    ... ..
    pWindow->clrBackground = clrbackground;
    pWindow->dwWndStatus    = WST_NORMAL;
    pWindow->dwWndStyle     = dwWndStyle;
    pWindow->hCursor        = NULL;
    pWindow->hFocusChild    = NULL;
    pWindow->hIcon          = NULL;
    pWindow->hOwnThread     = GetCurrentThread();
    pWindow->WndProc        = (NULL == WndProc) ? DefWindowProc : WndProc;
    pWindow->lpWndExtension = NULL; //Very important, this member maybe re-initialized by common
    ctrls.
    pWindow->dwSignature    = WINDOW_SIGNATURE;
    ... ..
}
```

上面黑体代码，即是设置窗口归属线程的代码。GetCurrentThread 是一个系统调用，用于返回当前线程（执行 CreateWindow 函数的线程）的线程句柄。

11.7.4 窗口树

窗口树是窗口机制中核心的数据结构，系统中的所有窗口，不论其归属核心线程是否相同，都会被连接到这一棵全局的树结构中。窗口对象中的四个指针（parent、child、next sibling 和 previous sibling），把窗口组织成一个树形结构。pParent 和 pChild，组成窗口树的层次结构，而 pPrevSibling 和 pNextSibling，则组成相同级别的链表结构，即所有具有兄弟

关系（父窗口相同）的窗口，组成一个双向链表，如图 11-13 所示。

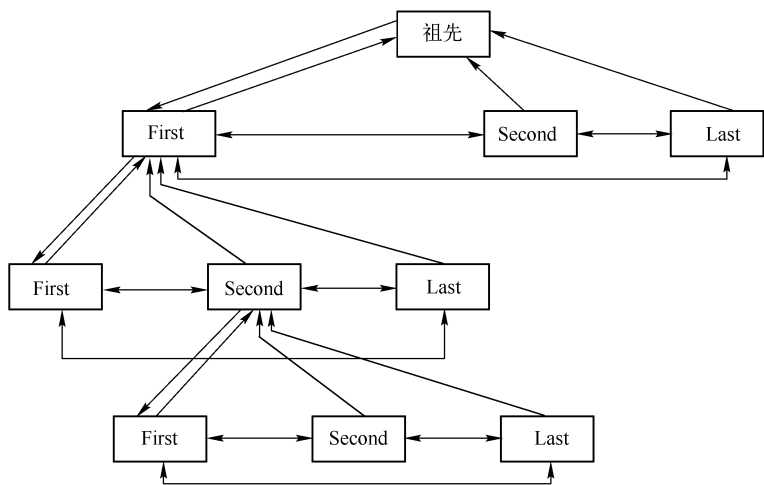


图 11-13 窗口树

在调用 `CreateWindow` 创建窗口的时候，需要指定一个父窗口对象句柄（下列代码中的黑体变量），这个句柄就是所创建窗口的父窗口：

```
[gui/include/wndmgr.h]
HANDLE CreateWindow(DWORD dwWndStyle, TCHAR* pszWndTitle, int x,
                    int y, int cx, int cy, __WINDOW_PROC WndProc,
                    HANDLE hParent, HANDLE hMenu, __COLOR clrbackground,
                    LPVOID lpReserved); //Create one window.
```

如果在创建窗口的时候，把 `hParent` 指定为 `NULL`，则系统会把祖先窗口（GUI 模块初始化过程中创建的第一个窗口）作为其父窗口。除了祖先窗口，系统中的任何一个窗口都有一个父窗口。

窗口树的最重要的用途，就是用于实现消息的有效传递。消息传递的详细过程，在本章 11.8 节中将详细介绍，这里只介绍消息传递过程中，如何通过窗口树找到消息的归属窗口。在搜索消息的归属窗口时，是按照消息位置（比如鼠标点击消息，其位置就是在屏幕上的坐标）检索整个窗口树的。在检索的时候，是按照深度优先的顺序进行的，即一个窗口的子窗口优先被检查。这很容易理解，比如在图 11-14 所示的窗口层叠关系中，鼠标点击消息应该传递给子窗口，而不是父窗口。



图 11-14 消息应该被传递给子窗口

在窗口树中，子窗口位于父窗口的下面，即更深的位置，因此采用深度优先的搜索算法，可确保消息被传递给子窗口。下面是以深度优先算法搜索窗口树的部分代码：

```
[gui/kthread/rawit.cpp]
static HANDLE GetFallWindowFromTree(HANDLE hWnd, int x, int y)
{
```

```
__WINDOW* pWindow = (__WINDOW*)hWnd;
__WINDOW* pChild = NULL;
HANDLE hResult = NULL;
//Check children first.
pChild = pWindow->pChild;
do{
    if(NULL == pChild)
    {
        break;
    }
    hResult = GetFallWindowFromTree((HANDLE)pChild,x,y);
    if(hResult){
        return hResult;
    }
    pChild = pChild->pNextSibling;
}while(pChild != pWindow->pChild);
if(PtInRegion(pWindow->pRegion,x,y))
{
    return (HANDLE)pWindow;
}
return NULL;
}
```

显然，这是个递归函数，这个函数的执行过程是：首先检查子窗口，如果消息落在子窗口内，则返回子窗口的对象句柄，否则认为落在了父窗口内。同时对于相同层级的兄弟窗口，则是链表前面的窗口被优先匹配。相信这个函数的代码，会比文字描述更加清楚。但这需要读者对递归函数的调用过程非常清楚，否则可能会对这一段代码感到迷惑。不过不要紧，即使递归调用对读者来说是个障碍，也无需沮丧，只要记住，在搜索消息的目标窗口时是深度优先的，即消息如果既落入了父窗口的覆盖范围，又落入了子窗口的覆盖范围，则优先传递给子窗口。

上面是对窗口树的纵向分析，以深度优先为原则。在窗口树的横向处理上，即兄弟窗口之间，也必须维持一种优先关系，否则也会产生混乱。比如图 11-15 所示这个窗口关系。

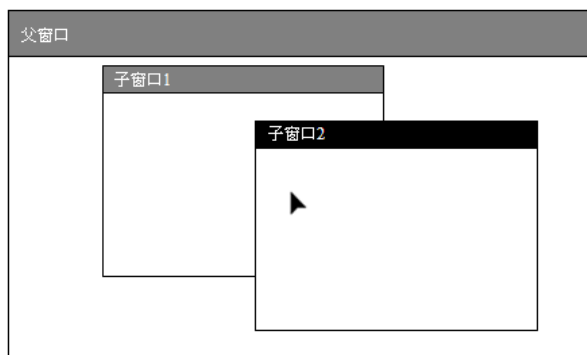


图 11-15 窗口消息的横向传递

子窗口 1 和子窗口 2 是兄弟关系，但是在兄弟链表中，子窗口 1 却位于子窗口 2 之前。这样如果在上图所示位置点击了鼠标，则消息可能会被错误地传递到子窗口 1，而不是子窗口 2。要解决这个问题，必须确保子窗口 2 在兄弟列表中，位于子窗口 1 之前。

要达到这个目的，只需要定义一条规则即可：在兄弟链表中，处于焦点状态（图中子窗口 2）的窗口，总是位于链表的第一个位置。这样就确保任何消息，都会被传递到正确的窗口。要满足这个原则，只需要在窗口的焦点状态改变时，同时变更其在兄弟链表中的位置即可。图 11-16 说明了这个过程。

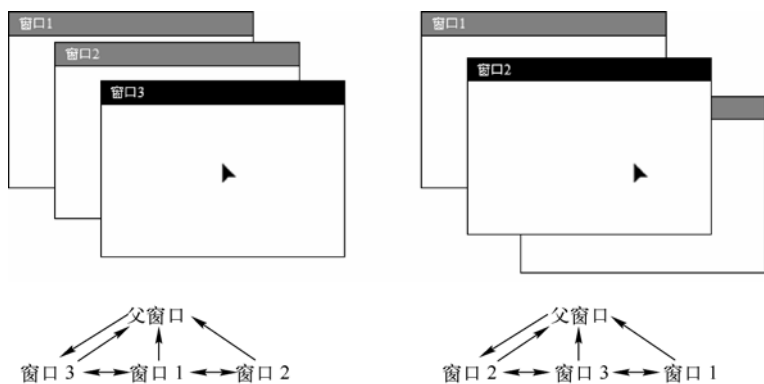


图 11-16 窗口焦点状态与在兄弟链表中的位置

开始的时候，窗口 3 是焦点窗口，于是在兄弟链表中，窗口 3 排在第一的位置。这时候用户用鼠标点击了窗口 2，于是窗口 2 变成了焦点窗口。这样必须同时修改兄弟链表，使窗口 2 位于链表的第一位置。

通过这样的一种顺序调整，就可确保我们设定的原则的达成。这样结合窗口树的深度优先原则，就可实现消息的准确传递。

11.8 用户输入处理和消息传递

11.8.1 用户输入和消息传递过程简介

图 11-17 详细展示了从用户输入开始，到最终消息被处理为止，涉及的所有中间实体，包括核心对象、核心线程、处理消息的窗口函数等。

下面对图 11-17 所示流程进行详细说明。

在 Hello China V1.75 GUI 模块的实现中，一个独立运行的线程 GUIRAWIT——GUI Raw Input Thread（后文简称为 RAWIT 或 GUIRAWIT），完成用户输入消息的第一层处理和分发。该线程是整个消息处理机制的核心。下面是一个典型的消息处理过程。

(1) 用户的输入（鼠标、触摸屏、键盘等），通过键盘驱动程序捕获。设备驱动程序，调用 DIM（Device Input Manager）对象提供的接口，把这些输入消息转换为特定的消息格式，传递给当前输入焦点线程。

(2) 在 GUI 模块初始化的时候，RAWIT 被作为当前输入焦点线程来安装，这样步骤 1 中的所有输入消息，首先到达 RAWIT。

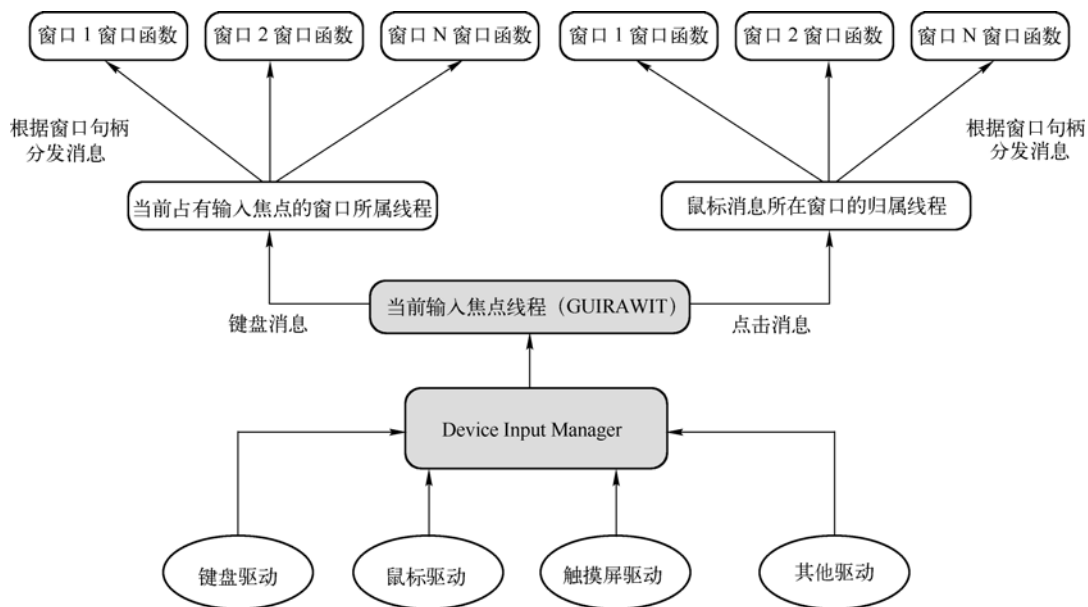


图 11-17 用户输入的消息传递过程

(3) RAWIT 判断消息的类型，完成系统级别的处理。比如，对于当前进程的切换（类似 Windows 操作系统的 Alt+Tab 组合键）、对于系统键（比如 Windows 操作系统的 Windows 徽标键）的输入等，都是在这个线程中处理。另外一个比较典型的系统级别的处理，就是 Alt+Ctrl+Del 组合键。一旦用户按下 Alt+Ctrl+Del 组合键，RAWIT 线程会立即启动 GUI 模块的退出操作，包括给其他核心线程发送 TERMINATE 消息、终止自己的运行等。

(4) 对于非系统级别的消息，需要调度到对应的应用程序进行处理。这时候，需要分不同的消息类别进行处理。

1) 对于键盘消息，则 RAWIT 直接调度到当前界面的输入焦点线程进行处理。这里的当前界面输入焦点，与整个系统的输入焦点线程不同。RAWIT 是整个系统的输入焦点线程，而当前界面输入焦点线程，则是接受当前用户输入的应用线程，一般为当前活动窗口所在线程。

2) 对于鼠标输入的消息，因为与输入位置有关，所以 RAWIT 需要根据鼠标的输入位置，获得该消息所对应的窗口，从而进一步获得该消息的目标线程，并把该消息调度到目标线程进行处理。这个过程在后面的部分中有详细描述。

3) 对于其他消息，比如 timer 消息等，则直接根据消息对应的窗口句柄，调度给对应的窗口所在的线程进行处理。

上述过程结束后，用户输入消息已经被调度到了一个特定的用户程序（线程）。下面的过程，是用户线程的消息处理过程：

(1) 消息被 RAWIT 发送到线程的消息队列，线程通过调用 GetMessage 函数，来获得该消息，然后做进一步的分析。

(2) 若该输入消息不是一个窗口消息，比如定时器消息、终止消息（TERMINATE 消

息)等,则该线程直接处理。

(3) 若输入消息是窗口消息(即消息需要与窗口进行关联,比如键盘消息、鼠标消息、定时器消息等),则线程会做如下处理:

1) 若输入消息是键盘消息,则直接传递给当前输入焦点窗口。因此,一个应用线程,应该实时跟踪当前焦点窗口的状态。

2) 若输入消息是一个鼠标消息,则 RAWIT 在分发该窗口消息的时候,已经把与消息关联的窗口句柄存储在了窗口消息中。这样在大多数情况下,当前线程直接根据窗口句柄,调用该窗口对应的窗口函数即可。只有一种情况例外,就是一个核心线程在处理诸如菜单、下拉列表窗口、模式对话框等 GUI 对象的时候。这些 GUI 元素,需要捕获所有鼠标的输入消息,即使鼠标的输入位置没有落在该窗口内。这种情况下,当前线程需要做一个转换,即把已经由 RAWIT 分配好处理窗口的窗口消息,再进行一次修改,修改当前处理窗口为捕获所有消息的当前窗口。需要注意的是,鼠标消息的位置,在 RAWIT 级别已经确定,因此在核心线程层面,无需做进一步判断和确定。

下面再说明一下 RAWIT 如何把一个鼠标点击消息(触摸屏点击消息遵循相同处理机制),与一个特定的窗口关联起来。在 Hello China 当前版本 GUI 模块的实现中,把系统中所有的窗口对象组织成了一棵树。系统中的第一个窗口对象(由 GUI SHELL 线程创建),是系统中所有其他窗口对象的祖先窗口。每创建一个窗口,就需要为创建的窗口指定一个父窗口,这样新创建的窗口就是指定窗口的子窗口。若在调用 CreateWindow 函数时不指定父窗口,则系统缺省把祖先窗口作为创建窗口的父窗口。

对于有相同父窗口的窗口,称为兄弟窗口。归属于同一个父窗口的所有兄弟窗口,组成一个链表,在窗口树中处于相同的层级。而具备父子关系的窗口,则在窗口树中表现为上下级的关系。

在理解了窗口的组织结构之后,窗口消息的传递机制就很明确了:在 RAWIT 接收到鼠标输入之后,会按照深度优先的原则遍历窗口树,根据窗口的位置和大小,判断鼠标消息位置是否落入了窗口之中。若是,则申请一个窗口消息对象,将该消息传递给窗口对应的线程即可。这种传递过程,是通过调用 SendMessage 来完成的。

从上述过程可以看出,对于鼠标类的位置消息,遵循子窗口优先的原则,即首先由子窗口进行处理,在子窗口无法处理的情况下,才会由父窗口进行处理。之所以遵循这样的原则,是因为一般情况下,子窗口都是覆盖在主框架窗口之上的。若采取相反的顺序,可能会导致任何消息都被主框架窗口处理掉,子窗口没有处理消息的机会。

总结上述过程可以看出,RAWIT 线程是用户输入消息的系统级分发者,为了完成消息的分发,该线程必须维护如下信息:

(1) 系统中的所有窗口实例(窗口对象)。因为 RAWIT 在处理位置相关消息时,需要根据输入位置定位到一个窗口,从而进一步定位到处理线程。在我们的实现中,是通过窗口树来管理系统中的所有窗口的。同时遵循这样的原则:父窗口总是位于子窗口的后面,具有相同父窗口的当前活动窗口,总是排在非活动兄弟窗口的前面。这样可有效处理窗口重叠的情况。

(2) 当前消息输入焦点线程的句柄。RAWIT 是系统中所有消息的输入焦点,但在用户程序层面,在多个用户程序并存的情况下,需要有一个当前消息输入焦点,RAWIT 会把与

位置无关的消息（比如键盘消息）直接调度给当前消息输入焦点线程。

(3) 所有窗口实例的当前状态。比如窗口是处于最大化还是最小化状态等。

11.8.2 设备驱动程序的工作

详细的设备驱动程序实现机制可参考第 10 章，此处不做详细说明。下面以鼠标驱动程序为例，对驱动程序中与消息输入有关的部分进行解释，希望读者能够建立一个大概的逻辑概念。

鼠标驱动程序被加载后，操作系统核心会调用其 `DriverEntry` 函数，该函数主要完成注册鼠标中断、初始化鼠标硬件、创建鼠标设备对象等工作。其中最重要的是第一步：注册鼠标中断。下面是相关代码：

```
[kernel/drivers/mouse.cpp]
BOOL MouseDrvEntry(__DRIVER_OBJECT* lpDriverObject)
{
    __DEVICE_OBJECT* lpDevObject = NULL;
    BOOL bResult = FALSE;

    g_hIntHandler = ConnectInterrupt(MouseIntHandler,
        NULL,
        MOUSE_INT_VECTOR);
    if(NULL == g_hIntHandler) //Can not connect interrupt.
    {
        goto __TERMINAL;
    } //------(1)
    //Initialize the key board.
    if(!InitMouse())
    {
        goto __TERMINAL;
    }
    //Create driver object for key board.
    lpDevObject = IOManager.CreateDevice((__COMMON_OBJECT*)&IOManager,
        "MOUSE",
        0,
        0,
        0,
        0,
        NULL,
        lpDriverObject);
    if(NULL == lpDevObject) //Failed to create device object.
    {
        PrintLine("Mouse Driver: Failed to create device object for MOUSE.");
        goto __TERMINAL;
    }
}
```

上面代码片段中，(1) 处标注的是注册中断函数的代码。这个函数（`ConnectInterrupt`）

的用途，是把一个中断处理函数与一个中断向量连接起来。这里的中断向量 0x2C (MOUSE_INT_VECTOR) 就是 PC 的缺省 PS/2 鼠标设备中断。需要注意的是，0x2C 是调整后的鼠标中断向量号。在切换到保护模式时，向量表的前 32 个，是系统保留给异常使用的，硬件中断从 32 号开始。因此虽然按照 IBM 兼容机的标准，PS/2 鼠标的中断是 12，在切换到保护模式时，也要加上 32，得到十六进制的 0x2C。其他硬件的中断号，也是这样得到的。

上述函数执行完毕，一旦鼠标被移动或被按下，鼠标的硬件就会通过该中断向量，中断 CPU 请求服务。因此鼠标中断处理函数 `MouseIntHandler` 是整个鼠标驱动程序的核心，后面将以该函数的实现为主要内容，讲解鼠标驱动程序是如何识别用户按下的鼠标键，以及如何把鼠标消息发送给操作系统核心的。为了便于理解下列代码，首先简单介绍一下机械鼠标的工作原理。

当前一般存在两种类型的鼠标，一类就是所谓的 2D（二维）鼠标，它就是我们平常用的那种没有滚轮的鼠标，由于这种鼠标在位移上只有 X 与 Y 两个方向，所以称之为 2D（二维）鼠标；还有一类就是现在比较常见的 3D（三维）鼠标，它们有一个滚轮，而这个滚轮会产生一个额外的 Z 位移量，因此，它在位移上有 X、Y、Z 三个方向，所以又称之为 3D（三维）鼠标。为了简单，我们以二维鼠标为例，讲解其工作机理。一旦用户移动鼠标、按下鼠标、释放鼠标，不管是左键还是右键，鼠标控制器（在 PC 上是 i8042 芯片，该芯片同时还控制键盘）会发起三次中断，通过三次中断向主机发送三个字节的数：第一个字节是控制字节，说明了鼠标当前的状态，比如鼠标左右键的状态（按下/释放），鼠标移动的偏移量的符号（正向移动还是负向移动），等等。接下来的两个字节，一个是 X 方向的偏移量，另外一个 Y 方向的偏移量。注意这里是偏移量，即是针对鼠标最后一次的位置的位置变化，因此会有正负之分（正、负号在第一个字节里面）。如果 X 或 Y 是负数，则是以补码表示的，需要进行特殊处理。下面是第一个字节各比特的含义：

- 位 0：左键按下标志位，为 1 表示左键被按下。
- 位 1：右键按下标志位，为 1 表示右键被按下。
- 位 2：中键按下标志位，为 1 表示中键被按下。
- 位 3：保留位，总为 1。
- 位 4：X 符号标志位，为 1 表示 X 位移量为负。
- 位 5：Y 符号标志位，为 1 表示 Y 位移量为负。
- 位 6：X 溢出标志位，为 1 表示 X 位移量溢出了。
- 位 7：Y 溢出标志位，为 1 表示 Y 位移量溢出了。

因此，针对任何一个鼠标消息，鼠标中断处理函数需要被调用三次：第一次输入控制字节，第二次输入 X 偏移分量，第三次输入 Y 偏移分量。好了，有了这些铺垫之后，我们正式进入鼠标中断处理程序。代码比较长，我们分段进行解释：

```
[kernel/drivers/mouse.cpp]
static BOOL MouseIntHandler(LPVOID,LPVOID)
{
    static BYTE MsgCount    = 0;
    static WORD x           = 0;
```

```

static BOOL xpostive    = TRUE;    //True if x scale is postive.
static WORD y           = 0;
static BOOL ypostive    = TRUE;    //True if y scale is postive.
static BOOL bLDPprev    = FALSE;    //Left button previous status,TRUE if down.
static BOOL bRDPprev    = FALSE;    //Right button previous status,TRUE if down.
static BOOL bLDCurr     = FALSE;    //Current left button status,TRUE if down.
static BOOL bRDCurr     = FALSE;    //Current Right button status,TRUE if down.
static BOOL bHasLDown   = FALSE;    //Left button has down before this down status.
static BOOL bHasRDown   = FALSE;
static DWORD dwTickCount = 0;
__DEVICE_MESSAGE dmsg;
UCHAR  data;

```

上面代码定义了中断函数要用到的一些变量，大部分是静态变量，这是因为一个鼠标消息，要至少调用三次中断处理函数，而且这三次之间还是有关联的（后面 X/Y 分量字节，受第一个控制字节的控制），因此上述静态变量用于记录之前的鼠标状态。

MsgCount 记录了三次中断的次序，第一次中断发生时，该值为 0，于是中断程序会认为是控制字节，会按照控制字节进行处理，然后增加 MsgCount。第二次中断的时候，中断程序会认为是 X 分量的偏移，于是处理 X 分量，然后再增加 MsgCount。第三次的时候自然是 Y 分量，处理完毕，重置 MsgCount 为 0。这样就实现了三次中断处理一个鼠标消息的目的。x 和 y 两个变量记录了鼠标的位置。

```

data = __inb(MOUSE_DATA_PORT);    //Read the input data.
MsgCount++;
switch(MsgCount)
{
case 1:    //The first byte of one mouse event.
    bLDCurr    = data & 0x01;    //If left button pressed.
    bRDCurr    = data & 0x02;    //If right button pressed.
    xpostive    = !(data & 0x10); //If x is postive.
    ypostive    = !(data & 0x20); //If y is postive.
    break;

```

中断函数第一次被调用，于是记录左键是否被按下、右键是否被按下，同时记录 X/Y 偏移的符号，然后结束中断处理函数，等待第二次中断。

```

case 2:    //The second byte of one mouse event.
    if(xpostive)
    {
        x += data;
        if(x >= MAX_X_SCALE) //Exceed the max X range.
        {
            x = MAX_X_SCALE;
        }
    }
    else
    {

```

```
data = 255 - data;    //CAUTION HERE!
if(x >= (WORD)data)
{
    x -= (WORD)data;
}
else
{
    x = 0;
}
}
break;
```

第二次中断的时候，根据 X 分量的符号值，来对 X 分量分别进行处理。如果是正数，则在当前鼠标 X 分量（x 值）基础上，增加相应的偏移。如果增加后超过了鼠标 X 分量的最大值（255），则会把鼠标的当前 x 坐标设置为最大。这时候鼠标在屏幕上的表现就是，鼠标到达屏幕的最右边，不能继续往右移动。如果 X 分量偏移是负数，则调整一下（原始数据是偏移的补码），在当前 X 分量上减去偏移。当然，X 分量不能为负数，最小为 0。如果为 0，则鼠标在屏幕上表现为到达最左端，不能继续左移。

```
case 3:    //The third byte of one mouse event.
    if(!ypostive) //For Y scale,down as postive.
    {
        data = 255 - data;    //CAUTION HERE!
        y += data;
        if(y >= MAX_Y_SCALE) //Exceed the max X range.
        {
            y = MAX_Y_SCALE;
        }
    }
    else
    {
        if(y >= (WORD)data)
        {
            y -= (WORD)data;
        }
        else
        {
            y = 0;
        }
    }
}
```

第三次中断的时候，首先处理 Y 分量，处理方式与 X 分量相同。

处理完毕，就形成一条完整的鼠标消息了。这时候首先要确定消息类型。XOR 是异或操作，用于判断两个 BOOL 值是否相同。如果两个 BOOL 值的异或结果为 TRUE，说明这两个值是不同的，否则相同。可以用 XOR 运算来确认前后两次鼠标消息的按键状态是否有变化，代码如下：

```

#define XOR(a,b) ((a && (!b)) || (b && (!a)))
if(XOR(bLDPprev,bLDCurr)) //Left button status changed.
{
    if(bLDPprev)
    {
        dmsg.wDevMsgType = KERNEL_MESSAGE_LBUTTONUP;
    }
    else
    {
        dmsg.wDevMsgType = KERNEL_MESSAGE_LBUTTONDOWN;
        if(bHasLDown) //A left button event has occurred before this one.
        {
            if(System.dwClockTickCounter - dwTickCount <= 3)
            {
                dmsg.wDevMsgType = KERNEL_MESSAGE_LBUTTONDBLCLK;
                bHasLDown = FALSE;
            }
            else
            {
                dwTickCount = System.dwClockTickCounter;
            }
        }
        else
        {
            bHasLDown = TRUE;
            dwTickCount = System.dwClockTickCounter;
        }
    }
}
}

```

上面代码判断鼠标左键是否有变化。如果有变化，则进一步判断鼠标键是被按下，还是抬起。由于 `bLDPprev` 记录了先前一次鼠标消息的左键是否被按下，因此如果 `bLDPprev` 是 `TRUE`，说明本次鼠标消息是按键抬起（本次消息和前一次消息一定相反）。这时候设置消息类型是 `KERNEL_MESSAGE_LBUTTONUP`。如果 `bLDPprev` 是 `FALSE`，则说明本次消息是按下鼠标键的消息，因此记录鼠标消息类型为 `LBUTTONDOWN`。这里还有一种特殊情况，就是双击。鼠标双击本质上是两次间隔很短的单击消息，因此我们要进一步判断是不是双击消息。这里的判断方式是，使用另外一个变量，`bHasLDown` 记录了先前一次消息是不是鼠标按下。如果是，且本次与前一次之间的时间间隔很小（小于 3 个系统时钟 tick），则认为是一次双击鼠标消息，于是重新设置消息的类型为左键双击（`LBUTTONDBLCLK`）。这时候还要把 `bHasLDown` 设置为 `FALSE`，因为本次是一个双击消息。

好了，下面的代码处理鼠标右键消息，处理方法与左键相同，不做详细解释。

```

else
{

```

```

        if(XOR(bRDPprev,bRDCurr)) //Right button status changed.
        {
            if(bRDPprev)
            {
                dmsg.wDevMsgType = KERNEL_MESSAGE_RBUTTONUP;
            }
            else
            {
                dmsg.wDevMsgType = KERNEL_MESSAGE_RBUTTONDOWN;
                if(bHasRDown)
                {
                    if(System.dwClockTickCounter - dwTickCount <= 3)
                    {
                        dmsg.wDevMsgType = KERNEL_MESSAGE_
RBUTTONDBCLK;

                        bHasRDown = FALSE;
                    }
                    else
                    {
                        dwTickCount = System.dwClockTickCounter;
                    }
                }
                else
                {
                    bHasRDown = TRUE;
                    dwTickCount = System.dwClockTickCounter;
                }
            }
        }
        else //Now button status is changed.
        {
            dmsg.wDevMsgType = KERNEL_MESSAGE_MOUSEMOVE;
        }
    }
}

```

经过上面的处理之后，就确定了鼠标消息类型，共有七个：左键被按下、左键抬起、左键双击、右键按下、右键抬起、右键双击、鼠标按键状态不变的鼠标移动。在操作系统核心中，鼠标消息也只有这七个。

确定消息类型后，还需要确定鼠标的当前位置。这很简单，在前面已经说过了，鼠标的当前位置记录在 *x* 和 *y* 变量里。鼠标消息类型和鼠标位置确定后，鼠标消息就完整了，这时候只需要把这个消息发送给操作系统核心即可：

```

dmsg.dwDevMsgParam = (DWORD)y;
dmsg.dwDevMsgParam <<= 16;
dmsg.dwDevMsgParam += (DWORD)x;
DeviceInputManager.SendDeviceMessage(

```

```
        (__COMMON_OBJECT*)&DeviceInputManager,
        &dmsg,
        NULL);

    //Change status variables.
    bLDPprev = bLDCurr;
    bRDPprev = bRDCurr;
    MsgCount = 0; //Reset.
    break;
}
return TRUE;
}

        (__COMMON_OBJECT*)&DeviceInputManager,
        &dmsg,
        NULL);

    //Change status variables.
    bLDPprev = bLDCurr;
    bRDPprev = bRDCurr;
    MsgCount = 0; //Reset.
    break;
}
return TRUE;
}
```

上述代码中，`dmsg` 是核心消息对象，把鼠标消息的相关参数记录到里面，然后调用 `SendMessage` 函数，把消息递交给 DIM（`DeviceInputManager`）对象即可。

`SendMessage` 函数比较简单，只是把设备消息发送给 DIM 对象，由 DIM 对象再进一步放到当前输入焦点线程的消息队列。

鼠标中断处理函数的最后，一定要恢复 `MsgCount` 的值为 0，这样可保证下一次鼠标消息（三个连续中断）被有效处理。

至此，我们以鼠标驱动程序为例，展示了如何把鼠标的硬件输入传递到操作系统内核（DIM 对象）。需要注意的是，设备驱动程序只是把硬件输入消息传递到内核了事，不会进一步跟踪消息的去向。即这个消息最终会被传递到哪个线程（或进程），在设备驱动层面是不知道的，这是操作系统内核的工作，在后面的章节中会有详细介绍。

对鼠标硬件的具体操作方法，无非是使用 `in` 或 `out` 指令，对硬件端口进行操作，这非常简单，因此本书没有详细介绍，感兴趣的读者可以查看本书的参考文献，或者到互联网上去搜索。键盘、触摸屏等硬件设备的输入机制是相同的，不同的是硬件相关的操作。

这个过程理解了，GUI 部分的消息传递机制就算是理解了至少三分之一，很简单，是不是？后面部分会更简单。同时，Windows 等操作系统基本也是这样处理的，是不是感觉到 Windows 也不是那么神秘莫测了？如果读者有这种感觉，那么本章一半的目的就算达到了，但愿如此。

11.8.3 设备输入管理器

下面把焦点转移到 DIM (Device Input Manager, 设备输入管理器) 对象。在前面关于鼠标驱动程序描述中, 已经初步接触到 DeviceInputManager 对象, 就是鼠标驱动程序通过综合三次连续中断, 形成一个合法的用户输入消息后, 调用 SendDeviceMessage, 把这个消息提交给操作系统内核。SendDeviceMessage 就是 DeviceInputManager (后文简称为 DIM) 提供的一个最重要的函数。对程序员来说, 阅读代码或原型定义, 是理解算法或对象最好的方法, 下面就是 DIM 对象的定义, 非常简单:

```
[kernel/include/dim.h]
BEGIN_DEFINE_OBJECT(__DEVICE_INPUT_MANAGER)
    __KERNEL_THREAD_OBJECT*    lpFocusKernelThread;
    __KERNEL_THREAD_OBJECT*    lpShellKernelThread;

    DWORD                      (*SendDeviceMessage)(__COMMON_OBJECT* lpThis,
                                                    __DEVICE_MESSAGE* lpDevMsg,
                                                    __COMMON_OBJECT* lpTarget);

    __COMMON_OBJECT* (*SetFocusThread)(__COMMON_OBJECT* lpThis,
                                       __COMMON_OBJECT* lpFocusThread);

    __COMMON_OBJECT* (*SetShellThread)(__COMMON_OBJECT* lpThis,
                                       __COMMON_OBJECT* lpShellThread);

    BOOL                  (*Initialize)(__COMMON_OBJECT* lpThis,
                                       __COMMON_OBJECT* lpFocusThread,
                                       __COMMON_OBJECT* lpShellThread);
END_DEFINE_OBJECT()
```

该对象最重要的两个变量, 就是 lpFocusKernelThread 和 lpShellKernelThread。其中第一个就是当前输入焦点线程, 而 lpShellKernelThread 则是缺省的 shell 线程。道理很简单, DIM 对象在接收到设备驱动程序送过来的消息后 (通过 SendDeviceMessage 函数), 会首先判断 lpFocusKernelThread 是否为空。如果为空, 说明还未设置当前输入焦点线程, 于是会把消息提交给 shell 线程 (lpShellKernelThread)。如果不为空, 则把消息提交给当前输入焦点线程。从这里可以看出, 当前输入焦点线程是可有可无的, 但 shell 线程是一定要有的。shell 线程是一个兜底线程, 当没有输入焦点接收消息时, DIM 就会把消息送给 shell 线程。再强调一下, 一定要把当前输入焦点线程与 GUI 界面下的焦点窗口所属线程区分开。焦点窗口所属线程不一定是当前输入焦点线程 (在 GUI 模式下, 当前输入焦点线程是 RAWIT 线程), 当前输入焦点线程也不一定是焦点窗口所属线程。

另外两个函数, SetFocusThread 和 SetShellThread, 就是用于设置这两个变量的。在 Hello China 初始化过程中, 创建完字符 shell 线程之后, 马上调用 SetShellThread, 把当前 shell 线程设置为字符 shell。这样任何用户按键数据都可以传递到字符 shell 线程了, 由 shell 线程做进一步的处理。而对于 GUI 模式, 则稍有不同。在 GUI 模块初始化过程中,



创建完 RAWIT 线程后，会调用 SetFocusThread，把当前焦点线程设置为 RAWIT。这样任何用户主动输入都会被 DIM 线程送到 RAWIT 进行处理，RAWIT 再进一步把消息分发到合适的用户线程，进而被递送到窗口进行处理。这里之所以把 RAWIT 当作当前输入焦点线程，而不是作为当前 shell 线程进行处理，是因为 GUI 模式是可退出的，退出后，就会重新进入字符 shell 模式。这样如果把字符模式的 shell 替换为 RAWIT 线程，就不能退出到字符 shell 了。

下面是 GUI 模块初始化的部分代码，摘录在此，以便进一步加深读者印象：

```
[gui/guientry.cpp]
DWORD __init(LPVOID)
{
    ... ..
    hRawInputThread = CreateKernelThread(
        0, //Use default stack size.
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_HIGH,
        RAWIT,
        NULL,
        NULL,
        "GUIRAWIT");
    if(NULL == hRawInputThread) //Can not create the RAW input thread.
    {
        goto __TERMINAL;
    }
    //Set the RAW input thread as current input focus thread,so all external input,such as
    //keyboard,mouse,will be dispatched to RAW input thread.
    hLastFocusThread = SetFocusThread(hRawInputThread);
    ... ..
}
```

这段代码的含义比较简单，主要是创建了 RAWIT 线程，然后把这个线程设置为当前输入焦点。从这以后，所有键盘输入、鼠标输入、触摸屏输入等主动输入消息，都将会被传递到 RAWIT 线程。RAWIT 线程再把消息传递给合适的用户线程。

大概的过程似乎已经说清楚了，只要调用 SetFocusThread，把一个线程设置为当前输入线程，硬件设备输入的消息就会被传递到该线程。但这只是比较泛泛的描述，具体是怎么传递的呢？所有细节都在 SendDeviceMessage 函数的实现里。仍然采用老方法，阅读代码。Linux 操作系统的作者 Linus Torvalds 曾经说过，阅读源代码是最好的理解系统原理的方法。下面就是 SendDeviceMessage 的相关代码，为了节约篇幅，我们省略了许多不相关的注释和安全检查代码：

```
[kernel/kernel/dim.cpp]
static DWORD SendDeviceMessage(__COMMON_OBJECT* lpThis,
                                __DEVICE_MESSAGE* lpDevMsg,
                                __COMMON_OBJECT* lpTarget)
```



```

{
    __DEVICE_INPUT_MANAGER*   lpInputMgr    = NULL;
    __KERNEL_THREAD_MESSAGE*   lpThreadMsg   = NULL;
    DWORD                      dwFlags       = 0L;

    lpInputMgr  = (__DEVICE_INPUT_MANAGER*)lpThis;
    lpThreadMsg = (__KERNEL_THREAD_MESSAGE*)lpDevMsg;
    if(lpTarget != NULL)
    {
        SendMessage(lpTarget,lpThreadMsg);
        return DEVICE_MANAGER_SUCCESS;
    } //------(1)

    if(lpInputMgr->lpFocusKernelThread != NULL)
    {
        if(KERNEL_THREAD_STATUS_TERMINAL == lpInputMgr->lpFocusKernelThread->dw
ThreadStatus)
        {
            __ENTER_CRITICAL_SECTION(NULL,dwFlags);
            lpInputMgr->lpFocusKernelThread = NULL;
            __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
            if(NULL != lpInputMgr->lpShellKernelThread)
            {
                SendMessage((__COMMON_OBJECT*)(lpInputMgr->lpShellKernel Thread),
                    lpThreadMsg);
                return DEVICE_MANAGER_SUCCESS;
            }
            else    //The current shell kernel thread is not exists.
            {
                return DEVICE_MANAGER_NO_SHELL_THREAD;
            }
        }    //The current status of the focus kernel thread is not TERMINAL.
    }
    else
    {
        SendMessage((__COMMON_OBJECT*)lpInputMgr->lpFocusKernelThread,
            lpThreadMsg);
        return DEVICE_MANAGER_SUCCESS;
    }
}
else    //The current focus kernel thread is not exists.
{
    if(NULL != lpInputMgr->lpShellKernelThread)
    {
        SendMessage((__COMMON_OBJECT*)lpInputMgr->lpShellKernelThread,
            lpThreadMsg);
        return DEVICE_MANAGER_SUCCESS;
    }
}

```

```
else
{
    return DEVICE_MANAGER_NO_SHELL_THREAD;
}
return DEVICE_MANAGER_SUCCESS;
}
```

这段代码比较简单，但是分支比较多。标号（1）部分代码是个特殊处理，这里首先判断设备是不是指定了消息的目标线程。很多情况下，有些设备是与特定应用程序相关联的。比如一个定制的游戏操控设备，可能只会与一个特定的游戏线程相关联。这时候游戏操控设备的输入，就无需经过内核（DIM 对象），直接传送到游戏程序即可。这种情况下，操控设备以游戏程序的线程对象为参数（lpTarget 参数），调用 `SendMessage` 函数，即可直接把消息传递到游戏程序。因此（1）处的代码，就是处理这种情况的。

但大多数的设备，是不指定目标线程的（lpTarget 为 NULL），具体送给哪个线程，是由 DIM 来确定的。这就是后续代码的目的。（1）后面的代码不复杂，但是分支比较多，大致步骤如下：

（1）首先判断当前输入焦点线程是否为 NULL，如果为 NULL，则转至步骤（5）处理，否则转至步骤（2）处理。

（2）判断当前输入焦点线程的状态是否为终止状态。有些情况下，输入焦点线程运行结束了，但是还未被取消作为输入焦点，会出现这种情况。这种情况下，把消息送过去是无意义的，因此要特殊处理。如果不为终止状态，则转至步骤（4）处理，否则转至步骤（3）处理。

（3）如果当前输入焦点为终止状态，则首先把当前输入焦点线程设置为 NULL，这样后续消息就会直接被送到 shell 线程处理了。然后把消息送给当前 shell 线程处理。如果 shell 线程也不存在，则失败返回。

（4）当前输入焦点线程状态正常（不是终止状态），则直接把消息送给当前输入焦点线程，然后返回。

（5）如果当前输入焦点线程为 NULL，则消息会被送到 shell 线程。在送到 shell 线程之前，也需要做一下判断，看 shell 是否也为 NULL。如果 shell 也为 NULL，则以失败返回，否则会把消息送给 shell 线程

至此，消息就会被送到当前输入焦点线程或 shell 线程的消息队列，等待输入焦点线程或 shell 线程做进一步处理。需要注意的有三个地方：

（1）`SendMessage` 一般是在设备驱动程序内调用的，其运行上下文为中断上下文，不属于线程上下文。在中断返回的时候，运行上下文会切换为线程上下文，优先级最高且状态为 `READY` 的核心线程会被调度运行。这样就确保了系统的响应时间，这在嵌入式领域非常重要。

（2）`SendMessage` 是调用 `SendMessage` 函数，把消息传递给特定线程的。`SendMessage` 函数是一个重要的系统函数，该函数把消息放到线程的消息队列，然后检查线程的状态，如果线程处于等待消息的阻塞状态，则会唤醒线程。这样在下一个调度时机，线

程就可能被调度执行（只要其优先级足够高），于是消息得以处理。SendMessage 函数的详细机制，可参考第 4 章。

(3) `__DEVICE_MESSAGE` 和 `__KERNEL_THREAD_MESSAGE` 是两个定义完全一致的结构体，用于承载消息信息。其中 `__DEVICE_MESSAGE` 是面向设备驱动程序使用的，而后者则是面向应用程序使用的，因此使用了不同的名字。为了方便读者理解，再把消息对象的定义摘录下来：

```
[kernel/include/ktmgr.h]
BEGIN_DEFINE_OBJECT(__KERNEL_THREAD_MESSAGE)
    WORD        wCommand;
    WORD        wParam;
    DWORD       dwParam;
END_DEFINE_OBJECT()
```

其中 `wCommand` 是消息的类型，而 `wParam` 和 `dwParam` 分别是与消息关联的两个变量，用于承载更进一步的消息信息，比如鼠标的坐标等。

至此，DIM 的工作机制解释完了。至于 DIM 的初始化（Initialize 函数）、如何设置当前输入焦点线程、当前 shell 线程，可参考相关代码（位于 `[kernel/kernel/dim.cpp]` 文件中），这些内容比较简单，就不单独讲述了。

到此为止，消息已经经过了设备驱动程序的处理，经过了 DIM 对象的处理，被传递到了当前输入焦点线程（在 GUI 模块中即是 RAWIT 线程）。后续部分将进一步讲解当前输入焦点线程是如何进一步把消息传递给应用线程的。

11.8.4 GUI 原始输入线程——GUIRAWIT

消息被 DIM 对象送到 RAWIT 线程之后，操作系统内核对消息的处理就算结束了，接下来 RAWIT 线程要大显身手了。RAWIT (Raw Input Thread)，翻译过来就是原始输入线程，这里“原始”的意思，是指消息还未被操作系统或应用程序处理过，是由设备驱动程序“原汁原味”地发过来的。同时还有另外一层意思，那就是消息的归属还不明确，到底归属哪个线程、哪个应用程序，还未被最终确定，因此用 Raw 来形容这个线程。

在前面关于 DIM 的描述中，RAWIT 线程是在 GUI 模块初始化过程中被创建的，创建完毕，被设置成当前输入焦点线程。这样只要从 RAWIT 线程的入口代码开始分析，逐层深入，就很容易把消息在 RAWIT 线程内的处理机制搞清楚。RAWIT 线程处理很多主动输入消息，比如鼠标的七个消息、键盘的若干个消息、其他类似输入设备的输入消息等，但对每个消息的处理机制是相同的，因此为了简便起见，以鼠标左键被按下（LBUTTONDOWN）这个消息为例，来说明 RAWIT 对消息的处理。

先从 RAWIT 线程的入口函数开始，下列代码是该线程的入口函数（即该线程被调度执行的起始点）：

```
[gui/kthread/rawit.cpp]
//Entry routine of RAWIT.
DWORD RAWIT(LPVOID)
{
```



```
MSG Msg;
WORD x = 0;    //Mouse x scale.
WORD y = 0;    //Mouse y scale.
MSG  msg;

while(TRUE)
{
    if(GetMessage(&Msg))
    {
        switch(Msg.wCommand)
        {
            case KERNEL_MESSAGE_LBUTTONDOWN:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoLButtonDown(x,y);
                break;
            case KERNEL_MESSAGE_LBUTTONUP:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoLButtonUp(x,y);
                break;
            case KERNEL_MESSAGE_RBUTTONDOWN:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoRButtonDown(x,y);
                break;
            case KERNEL_MESSAGE_RBUTTONUP:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoRButtonUp(x,y);
                break;
            case KERNEL_MESSAGE_MOUSEMOVE:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoMouseMove(x,y);
                break;
            case KERNEL_MESSAGE_LBUTTONDBCLK:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoLButtonDbClk(x,y);
                break;
            case KERNEL_MESSAGE_RBUTTONDBCLK:
                x = (WORD)Msg.dwParam;
                y = (WORD)(Msg.dwParam >> 16);
                DoRButtonDbClk(x,y);
                break;
        }
    }
}
```

```
case KERNEL_MESSAGE_TIMER:
    break;
case KERNEL_MESSAGE_AKUP:    //ASCII key up.
    OnAkUp(&Msg);
    break;
case KERNEL_MESSAGE_AKDOWN:  //ASCII key down.
    OnAkDown(&Msg);
    break;
case KERNEL_MESSAGE_VKUP:    //Virtual key up.
    OnVkUp(&Msg);
    break;
case KERNEL_MESSAGE_VKDOWN:  //Virtual key down.
    OnVkDown(&Msg);
    break;
case KERNEL_MESSAGE_TERMINAL: //System terminal message.
    if(GlobalParams.hGUIShell)
    {
        msg.wCommand = KERNEL_MESSAGE_TERMINAL;
        SendMessage(GlobalParams.hGUIShell,
            &msg); //Send terminate message to GUI shell thread.
    }
    goto __TERMINAL; //End this thread.
    break;
default:
    break;
}

}

__TERMINAL:
return 0;
}
```

代码比较简单，就是一个无限循环，调用 `GetMessage` 从其线程队列内获取消息。注意 `GetMessage` 函数是阻塞操作，如果线程的消息队列内无任何消息，则该线程进入阻塞等待状态，等待消息的来临。一旦有消息被送入消息队列（通过 `SendMessage` 函数），则线程会被唤醒，进而对消息进行处理。如果线程消息队列内有消息，则 `GetMessage` 会把线程消息队列内的第一个消息（队列头消息），复制到你参数 `Msg` 内，然后从消息队列中删除该消息。

接下来就是分析 `Msg` 的消息类型了。Hello China V1.75 版本定义的消息类型见表 11-7。

表 11-7 Hello China 定义的消息类型

消 息 名 称	类型值	消 息 含 义
KERNEL_MESSAGE_AKDOWN	1	键盘的 ASCII 键被按下
KERNEL_MESSAGE_AKUP	2	键盘的 ASCII 键抬起
KERNEL_MESSAGE_VKDOWN	203	键盘的虚拟键（比如 F1 等）被按下



(续)

消 息 名 称	类型值	消 息 含 义
KERNEL_MESSAGE_VKUP	204	键盘的虚拟键（比如 F1 等）抬起
KERNEL_MESSAGE_TERMINAL	5	终止消息，用户按下 Ctrl+Del+Alt 组合键，键盘驱动程序会向操作系统核心发送该消息
KERNEL_MESSAGE_TIMER	6	定时器消息
KERNEL_MESSAGE_LBUTTONDOWN	301	鼠标左键被按下
KERNEL_MESSAGE_LBUTTONUP	302	鼠标左键抬起
KERNEL_MESSAGE_RBUTTONDOWN	303	鼠标右键被按下
KERNEL_MESSAGE_RBUTTONUP	304	鼠标右键抬起
KERNEL_MESSAGE_LBUTTONDBCLK	305	鼠标左键双击
KERNEL_MESSAGE_RBUTTONDBCLK	306	鼠标右键双击
KERNEL_MESSAGE_MOUSEMOVE	307	鼠标移动

针对每个消息类型，RAWIT 会进一步调用不同的函数进行处理。这里看 LBUTTONDOWN 消息，在收到该消息后，RAWIT 线程会调用 DoLButtonDown 函数，对其进一步进行处理。需要记住的是，鼠标驱动程序使用消息对象的 dwParam 记录了鼠标消息的位置（低 16 比特记录了 x 分量的位置，高 16 比特记录了 y 分量的位置），因此在进一步调用 DoLButtonDown 函数前，需要把鼠标位置的 x 和 y 分量分离出来。

```
[gui/kthread/rawit.cpp]
//LEFT MOUSE BUTTON DOWN event handler.
static VOID DoLButtonDown(int x,int y)
{
    __WINDOW_MESSAGE* pWmsg = NULL;
    MSG                msg;
    int xpos,ypos;
    HANDLE hTargetWnd = NULL;
    pWmsg = (__WINDOW_MESSAGE*)KMemAlloc(sizeof(__WINDOW_MESSAGE), KMEM_
SIZE_TYPE_ANY);
    if(NULL == pWmsg)
    {
        return;
    } //------(1)
    MouseToScreen(&Video,x,y,&xpos,&ypos); //------(2)
    hTargetWnd = GetFallWindow(xpos,ypos);
    if((HANDLE)WindowManager.pCurrWindow != hTargetWnd)
    {
        UnfocusWindow((HANDLE)WindowManager.pCurrWindow);
        FocusWindow(hTargetWnd);
    } //------(3)
    pWmsg->message = WM_LBUTTONDOWN;
    pWmsg->hWnd     = hTargetWnd;
    pWmsg->wParam   = 0;
```

```

pWmsg->lParam = xpos;
pWmsg->lParam <=& 16;
pWmsg->lParam += ypos;
msg.dwParam = (DWORD)pWmsg;
msg.wCommand = KERNEL_MESSAGE_WINDOW;
SendMessage((HANDLE)WindowManager.pCurrWindow->hOwnThread,&msg); //------(4)
}

```

在介绍上述代码之前，先引入一个概念：窗口消息。之前的设备消息和内核消息（这两者的定义是一样的），都是与图形界面的窗口无关的。但到了 GUI 之后，一个消息需要与窗口关联起来，比如一个鼠标点击，最终是落在某个窗口之内的，这样这个点击消息就与所在窗口建立了关联。因此这些与窗口进行关联的鼠标点击等消息，就叫做窗口消息。RAWIT 线程的重要工作之一，就是把原始的内核（或设备）消息，转换为窗口消息。

与设备消息和内核消息不同的是，窗口消息的定义中增加了一个窗口句柄，下面是其定义：

```

[gui/include/wndmgr.h]
struct __WINDOW_MESSAGE{
    HANDLE hWnd;
    UINT   message;
    WORD   wParam;
    WORD   wReserved;
    DWORD  lParam;
};

```

这里的 hWnd，就是消息归属窗口的窗口句柄，message 则是消息类型，与内核消息的 wCommand 含义一样。后面的 wParam 和 lParam，分别用于存放与 message 相关的进一步信息。在 GUI 模式下，鼠标消息、键盘消息、触摸屏消息等，都是窗口消息。

下面正式分析 DoLButtonDown 函数，针对代码中标注的（1）、（2）、（3）、（4），分别介绍如下：

（1）显然，鼠标左键按下消息是一个窗口消息，RAWIT 需要把原始消息转换为窗口消息，然后把窗口消息发送给目标应用程序。因为窗口消息对象是发送给另外的核心线程的，因此窗口消息不能从堆栈中分配内存，必须从核心内存池中分配。（1）部分代码，就是从核心内存池中申请一个窗口消息对象。需要注意的是，RAWIT 只负责创建窗口消息，然后把消息发送给目标线程。窗口消息的销毁，是由目标线程完成的。目标线程处理完窗口消息后，必须调用 KMemFree 函数释放对应内存。这个过程无需应用程序编程人员考虑，操作系统相关调用已完成了内存的释放工作。

（2）从设备输入管理器（DIM）送到 RAWIT 线程的鼠标消息，其坐标位置是原始的，范围在 0~255 之间。但切换到图形模式后，屏幕分辨率是变化的（V1.75 的缺省分辨率是 1024×768 像素）。这样必须把鼠标的原始位置，转换为在屏幕上的具体位置，这就是 MouseToScreen 函数的工作。这个函数通过读取 Video 对象的相关参数，获得屏幕分辨率，然后把原始鼠标坐标转换为屏幕坐标，并通过 xpos 和 ypos 返回。

（3）找到鼠标在屏幕上的坐标后，接下来就是 RAWIT 线程最核心、最重要的工作了：确

定鼠标消息的归属窗口。要知道在图形模式下，屏幕上是有多个窗口的，而 `MouseToScreen` 函数返回的是鼠标消息在整个屏幕上的位置，而不是某个具体窗口的位置。因此，`RAWIT` 线程就需要确定窗口左键按下消息，到底是落在哪个窗口内。找到归属窗口后，归属线程就好找了，因为在窗口对象中记录了窗口的归属线程。显然，`GetFallWindow` 就是用户获取鼠标消息的归属窗口的。该函数查询窗口树，按照某个优先级对系统中的所有窗口进行匹配，最终找到目标窗口。需要注意的是，找到一个归属窗口后，首先判断该窗口是不是当前焦点窗口（注意与当前输入焦点线程的区别）。对于当前焦点窗口，窗口管理器会单独记录（`WindowManager.pCurrWindow`），这是为了处理键盘消息的目的。所有键盘消息（除了系统组合键），都会被发送到当前焦点窗口所在线程，这是操作系统 GUI 功能的通用处理方式。如果 `RAWIT` 发现鼠标左键按下的目标窗口不是当前焦点窗口，则需要修改当前焦点窗口为左击消息落入的窗口。在这之前，需要通知原焦点窗口已失去输入焦点，这就是 `UnfocusWindow` 函数的作用。同样地，也要通知鼠标消息落入窗口已经获得输入焦点。这就是 `FocusWindow` 的用途。

（4）找到鼠标左键按下消息落入的窗口对象后，剩下的事情就好办了，就是把鼠标在屏幕上的位置保存在窗口消息对象的 `lParam` 参数内（最低两个字节保存 `x` 坐标，最高两个字节保存 `y` 坐标），同时也把目标窗口的句柄保存在窗口消息内，然后把这个窗口消息发送给目标窗口所在线程。注意，这里在发送的时候，由于消息目标还是一个线程，因此必须以核心线程消息对象进行发送。于是，上述代码首先把窗口消息的地址，保存在内核消息的 `dwParam` 内，然后设置内核消息的消息类型为 `KERNEL_MESSAGE_WINDOW`，调用 `SendMessage`，把消息发送给目标线程。这里实际上是用一个内核消息携带了窗口消息。

好了，这个鼠标左键按下的消息就处理完了。其他消息，比如鼠标双击、鼠标右键按下等，与此处理方式类似，读者可通过阅读代码做进一步理解。相关代码，都在 `[gui/kthread/rawit.cpp]` 文件内。

接下来，就需要应用程序来完成消息的最终处理了。后面的过程，对 Windows 应用程序员来说，应该是驾轻就熟。但本书还是要详细讲解一下。因为这部分内容是消息驱动机制的精华所在，而且本书在讲解的时候，不会太注重应用程序层面的处理，而是深入内核机制，考察一下消息循环到底是怎么工作的。

11.8.5 消息循环的本质

`RAWIT` 线程调用 `SendMessage`，把消息送入目标线程的消息队列后，`RAWIT` 线程的工作就完成了，剩下的工作，就是应用程序自己的事了。下面就详细讲解一下基于消息驱动的应用程序编程模型。

对 Windows 程序员来说，消息驱动机制不应该是陌生的内容。在 Windows 程序中，首先要有一个消息循环，这个消息循环不断地从消息队列中获取消息，一旦获得消息，就调用 `SendMessage`（注意这里是 Windows 的 `SendMessage`，与 Hello China 的 `SendMessage` 功能不同。前者的功能是把消息送给窗口进行处理，而后者则是把消息送给另外一个线程进行处理）或 `DispatchMessage`，对消息进行分发处理。Windows 的 `SendMessage` 或 `DispatchMessage` 进一步调用窗口对象的窗口函数，窗口函数就是真正的用户功能代码所在地。

Hello China 的消息驱动机制与此类似，与 Windows 不同的是，Hello China 使用

SendMessage 替代了 Windows 的 SendMessage 函数，这主要是因为 SendMessage 函数已经被内核提前“征用”了。图 11-18 说明了这个过程。

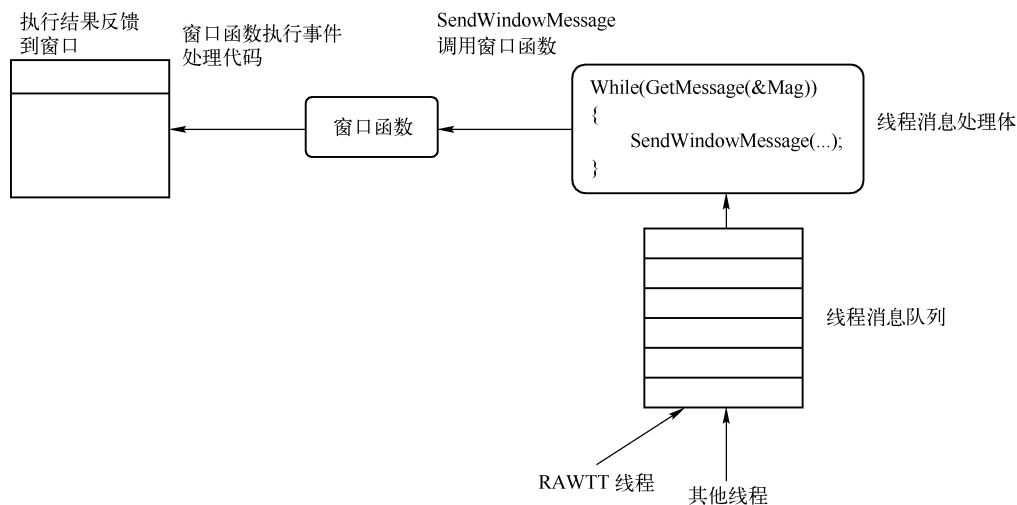


图 11-18 消息在核心线程内的调度过程

这里最关键的是两个函数——GetMessage 和 SendMessage。把这两个函数讲清楚了，这部分的内容就讲完了。先看 SendMessage，这个函数的功能相对简单，只是调用了窗口消息的窗口函数。下面是其代码（为了简便，省略了参数安全检查等部分代码）：

```
[gui/window/wndmgr.cpp]
BOOL SendMessage(HANDLE hWnd, __WINDOW_MESSAGE* pWndMsg)
{
    __WINDOW* pWnd = (__WINDOW*)hWnd;
    if(pWnd->dwWndStatus & WST_CLOSED)
    {
        if((pWndMsg->message == WM_DRAW) || (pWndMsg->message == WM_PAINT))
        {
            return FALSE;
        }
    }
    return pWnd->WndProc(hWnd,
        pWndMsg->message,
        pWndMsg->wParam,
        pWndMsg->lParam);
}
```

最关键的部分，就是黑体标注部分。这行代码调用了窗口对象的窗口函数。应用程序的功能，就是在窗口函数内实现的，因此这实际上就是调用了应用程序的功能代码。Windows 操作系统的 SendMessage 函数，大致实现也是如此，就是调用了目标窗口的窗口函数。需要注意的是，上述 SendMessage 函数中的 hWnd 参数，与窗口消息 pWndMsg 中的窗口

对象句柄是同一个，这个窗口对象句柄是由 RAWIT 确定并填写在窗口消息里面的。但为什么 `SendMessage` 函数不直接利用 `pWndMsg` 里面的窗口句柄，而是额外设计第一个参数呢？当初这样设计的时候，是考虑到 `pWndMsg` 参数内的窗口句柄可以设置为 `NULL`，但后来发现这个设计有点多余。或许在后续的版本中，会更改这个设计。

对于“窗口函数就是应用程序的功能代码”这个结论，可能会有很多人不认同，尤其是没有编写过原始 Windows 程序（直接调用 Windows API 编程）的程序员。比如 MFC 程序员，他们会认为 `OnDraw` 或 `OnPaint` 等虚拟函数，才是真正的用户功能代码。但 MFC 不过是对 Windows API 的一个封装。它通过充分利用 C++ 语言的虚拟函数机制，进一步抽象了 Windows 的编程模型，向程序员隐藏了充满 `case` 语句的窗口函数实现过程。但两者本质是一样的，无非是通过层层封装，简化了程序员的编程工作而已。窗口函数是一个分水岭，在窗口函数之前的所有消息处理，都是操作系统的工作。窗口函数之后的处理，则完全是应用程序的事情了。当然，如果窗口函数对消息不做特殊处理，而直接调用 `DefWindowProc`，效果是一样的。

`SendMessage` 函数的前半部分，是根据窗口的状态做了一个特殊处理。在窗口处于关闭状态（窗口已被用户关闭，不显示在屏幕上，同时窗口对象可能即将被销毁）时，对窗口绘制消息（`WM_DRAW/WM_PAINT`）进行了拦截，不会发送给窗口函数。这是因为在窗口处于关闭状态时，向其发送绘制消息是无意义的，这是一种高效率的处理方式。

到此为止，实际上消息传递机制已经讲完了。一条消息，从最初的硬件（鼠标、键盘等）输入，到最终的应用程序功能代码（窗口函数），经历了万水千山。下面对整个过程做一个简单总结。

（1）首先，用户按下键盘上的键，或者移动鼠标、点击鼠标，硬件驱动程序代码会被调用。驱动程序完成原始的处理后，调用 `SendDeviceMessage` 函数，把硬件消息传递到内核。

（2）内核的 DIM 对象把这个消息传递给当前输入焦点线程，或者在当前输入焦点线程不存在的情况下，传递给字符 shell 线程。对于 GUI 模式，当前输入焦点线程就是 RAWIT。

（3）RAWIT 对消息做进一步处理，包括硬件坐标和屏幕坐标的映射、找到消息落入的窗口对象等，然后通过调用 `SendMessage`，把消息递交给应用线程。

（4）应用线程调用 `GetMessage`，从自身的消息队列中获得消息，然后再调用窗口函数（实际上是通过 `SendMessage` 间接调用了窗口函数），对消息进行处理。

（5）应用程序的窗口函数对消息进行处理，并把处理结果显示在屏幕上。

按既定的内容范围来说，本章内容到此就结束了。因为本章内容聚焦于消息的传递机制，把消息传递的每个环节都解释了。

但我还不想就这样结束本章，我还要介绍一下 `GetMessage` 函数。这个函数与 `SendMessage` 一起，组成了消息驱动机制的核心。这部分内容相当重要，而且与消息传递机制关联紧密，放到这里讲是比较合适的。

我想强调的最核心的一点，就是 `GetMessage` 是基于阻塞操作的。应用程序在一个无限循环内调用 `GetMessage` 函数，并不会导致应用程序一直占用 CPU。`GetMessage` 函数会检查线程的消息队列，如果消息队列为空，没有任何消息，则线程会进入等待状态，这时候是不会被调度的。只有另外的线程（或设备驱动程序等）把消息发送到线程的消息队列，线程才会被重新唤醒，进而对消息进行处理。

还是通过阅读代码，来分析 GetMessage 的实现。代码如下（做了删节）：

```
[kernel/kernel/ktmgr.cpp]
static BOOL MgrGetMessage(__COMMON_OBJECT* lpThread, __KERNEL_THREAD_MESSAGE*
lpMsg)
{
    __KERNEL_THREAD_OBJECT*    lpKernelThread = NULL;
    DWORD                      dwFlags      = 0L;
    lpKernelThread = (__KERNEL_THREAD_OBJECT*)lpThread;
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    if(MsgQueueEmpty(lpThread)) //Current message queue is empty,should waiting.
    {
        lpKernelThread->dwThreadStatus = KERNEL_THREAD_STATUS_BLOCKED;
        lpKernelThread->lpMsgWaitingQueue->InsertIntoQueue(
            (__COMMON_OBJECT*)lpKernelThread->lpMsgWaitingQueue,
            (__COMMON_OBJECT*)lpKernelThread,
            0L);
        KernelThreadManager.ScheduleFromProc(NULL); //Re-schedule.
    }
    lpMsg->wCommand      = lpKernelThread->KernelThreadMsg[lpKernelThread->ucMsgQueue
Header].wCommand;
    lpMsg->wParam         = lpKernelThread->KernelThreadMsg[lpKernelThread->ucMsgQueue
Header].wParam;
    lpMsg->dwParam        = lpKernelThread->KernelThreadMsg[lpKernelThread->ucMsgQueue
Header].dwParam;

    lpKernelThread->ucMsgQueueHeader ++;
    if(MAX_KTHREAD_MSG_NUM == lpKernelThread->ucMsgQueueHeader)
        lpKernelThread->ucMsgQueueHeader = 0x0000;
    lpKernelThread->ucCurrentMsgNum --;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return TRUE;
}
```

重点注意上面黑体标注的两行代码。第一行中，判断线程的当前消息队列是否为空。如果为空，则把当前线程的状态设置为 **BLOCKED**，然后插入消息等待队列，并引发一个调度。**KernelThreadManager.ScheduleFromProc** 函数调用后，实际上当前线程就不再运行了，**ScheduleFromProc** 函数会选择另外一个状态是就绪的线程投入运行。

从 **ScheduleFromProc** 返回后，说明线程的消息队列里面已经有消息了。这时候线程会获取消息，然后返回即可。这里的难点在于，**ScheduleFromProc** 函数实际上是一个“跨越时空”的函数。一旦被调用，该函数会检查操作系统的就绪线程队列，从中选择一个优先级最高的投入运行。这样调用它的当前线程，实际上就不再占用 CPU 了。但是 **ScheduleFromProc** 函数会保存当前线程的上下文，等待线程被另外的线程唤醒。另外的线程必须调用 **SendMessage**，才能唤醒一个等待消息的线程。

一旦另外的线程（或驱动程序）调用 **SendMessage**，在当前线程的消息队列里面放入一

个消息，则当前线程同时会被唤醒。这里的唤醒，实际上就是 `SendMessage` 函数，把当前线程的状态修改为 `READY`，然后加入就绪队列。这样下一次调度时机来临的时候，如果当前线程的优先级足够高，就会被调度执行。

`SendMessage` 函数的代码刚好与 `GetMessage` 相反，它向一个线程的消息队列里面放入一条消息，然后检查目标线程是否处于等待消息状态。需要注意的是，一个线程的消息队列中可能有多个消息，因此如果消息队列不为空，线程是不会处于等待消息状态的。如果 `SendMessage` 函数发现目标线程不处于消息等待状态，则直接返回。否则，会试图“唤醒”目标线程。这里的唤醒，无非是修改一下目标线程的状态，然后把它重新放入就绪队列。

这样 `GetMessage` 和 `SendMessage` 两个函数你来我往，密切配合，就形成了大名鼎鼎的消息驱动机制。很多人可能认为消息驱动机制不适合实时操作系统，因为线程之间的通信，只是发送消息，不能确保消息被及时有效处理。其实，消息传递机制与其他线程同步机制是一样的，都是线程的阻塞/唤醒等轮换操作。不同的是，一般的线程同步机制，大多是通过全局变量、共享对象等传递信息，然后通过互斥体、信号量等完成同步和资源保护。但消息机制则是通过消息来实现信息传递的，这不需要全局变量或全局对象，代码逻辑反而更加清晰。在实时性方面，消息传递机制与其他线程同步机制是一样的。不论是 `SendMessage` 函数还是其他的同步函数，本质上都是修改线程状态，然后再加入就绪队列。只要线程的优先级足够高，在下次调度时机到来时，就会被调度执行。并不会出现消息积压的情况。

如果上面的分析还是不能说服读者，没有问题，Hello China 等操作系统也实现了传统的线程同步和资源保护机制，比如事件、互斥体、信号量等机制，读者完全可以用这些机制来替代消息驱动机制。但是如果读者认同消息驱动机制，那么建议还是充分使用消息驱动机制完成程序的开发。因为消息驱动机制有很强的扩展性和灵活性，能够模拟非常复杂的交互关系，且代码逻辑关系清晰，便于维护。

11.8.6 应用线程之间的窗口消息交互

在 GUI 模块中，对于相同核心线程内的窗口消息传递，通过 `SendWindowMessage` 函数完成。该函数实际上直接调用了目标窗口的窗口过程。但对于不同线程的窗口消息传递，比如线程 A 发送一个窗口消息给线程 B，则需要经过操作系统核心提供的 `SendMessage` 函数来实现。下面的代码片断，是从 `FocusWindow` 函数中截取的：

```
[gui/window/wndmgr.cpp]
VOID FocusWindow(HANDLE hWnd)
{
    __WINDOW_MESSAGE* pWmsg = NULL;
    MSG                msg;
    __WINDOW*         pWnd  = (__WINDOW*)hWnd;
    __WINDOW*         pParent = NULL;
    DWORD              dwFlags = 0;
    ... ..
    pWmsg = (__WINDOW_MESSAGE*)KMemAlloc(sizeof(__WINDOW_MESSAGE), KMEM_
    SIZE_TYPE_ANY);
```

```
... ..
//Send On focus message to target thread.
pWmsg->hWnd      = hWnd;
pWmsg->message     = WM_ONFOCUS;
pWmsg->wParam      = 0;
pWmsg->lParam      = 0;

msg.wCommand      = KERNEL_MESSAGE_WINDOW;
msg.wParam         = 0;
msg.dwParam        = (DWORD)pWmsg;
SendMessage(((__WINDOW*)hWnd)->hOwnThread,&msg);
}
```

这段代码示例了向不同线程的窗口对象发送消息的过程：

- (1) 首先创建一个窗口消息对象，并初始化。
- (2) 把创建的窗口消息对象，附加在核心线程消息对象（msg）上。
- (3) 调用 SendMessage，把核心线程消息发送给目标线程。

这样目标核心线程在得到调度的时候，就可以从自己的消息队列中获得对应的核心线程消息，然后做进一步判断，若发现是窗口消息，则会调用 SendWindowMessage 函数，向指定的窗口分发窗口消息。

需要注意的是，在 SendWindowMessage 函数执行完毕前，需要释放窗口消息对象占用的内存。因为这片内存是在发送消息的核心线程中申请的。

11.9 Hello China 的 GUI Shell

11.9.1 GUI Shell 概述

Shell 是计算机操作系统呈现给用户的第一层交互接口，实际上就是一个应用程序，这个应用程序由操作系统提供，用户通过这个应用程序来启动其他应用程序。一般情况下，存在图形模式的 shell（GUI shell）和字符模式的 shell（CUI shell）两种 shell 表现形式。对于字符模式的 shell，大多数人都应该很熟悉，最有名的就是 DOS 操作系统的命令行界面和 UNIX/Linux 的各种 shell，如 C Shell、K Shell 等。

对于 GUI 的 shell，或许有的读者感到迷惑，GUI 模式下会有 shell 吗？答案是肯定的，以最典型的 Windows 操作系统为例，开机启动完成后，只要看到桌面，实际上就进入了它的 shell 程序。Shell 程序（在 Windows 操作系统里是 explorer 进程）把系统已安装的应用程序统一呈现给用户，由用户根据需要启动特定的程序，以完成特定功能。除此之外，shell 还有其他的功能，比如修改显示外观等。

Hello China 操作系统对两种 shell——图形模式 shell 和字符模式 shell 都支持。缺省情况下，Hello China 启动完成后，进入的是字符模式的 shell。在字符模式下，用户运行 gui 命令，即可进入图形模式，从而切换到图形 shell。在图形 shell 下，展现给用户的是系统中已安装的所有图形模式的应用程序，同时也呈现给用户一些辅助信息，比如 V1.75 版的实现

中，呈现给用户的是时间和日历信息。图 11-19 是 GUI Shell 的运行截图。

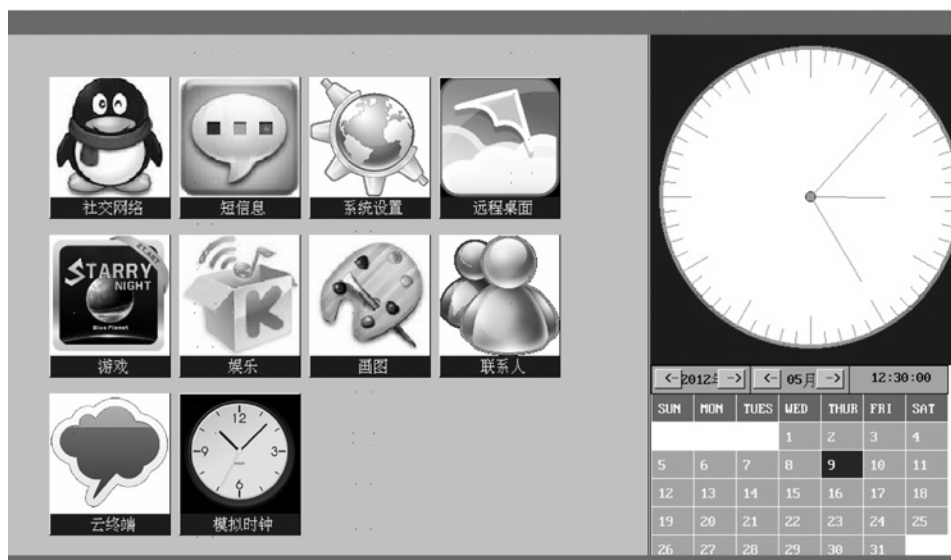


图 11-19 GUI Shell 运行截图

整个屏幕被分成了四个部分，最上面和最下面的条形区域，是系统信息显示区，用于直观显示一些系统信息。操作系统提供编程接口，供应用程序直接调用，来显示特定信息。右面的区域，是用于显示辅助信息的地方，V1.75 的实现中，在这个区域内放了一个模拟时钟和一个日程序，这样用户就可实时查看时间和日历信息。位于窗口左面的大部分区域，就是应用程序呈现区。系统中所有成功安装的应用程序，都被操作系统枚举，并显示在这个地方。用户只需用鼠标或触摸屏按一下相应图标，应用程序即可启动。在 Hello China V1.75 的实现中，应用程序窗口会占据整个应用程序显示区。这样的结果是，用户在同一时间，只能运行一个基于图形用户界面的应用程序。这是充分考虑了嵌入式系统的特点而做的实现策略，因为在嵌入式领域，应用都是由单一的应用程序来完成，而且屏幕相对较小，若罗列多个 GUI 应用程序会显得拥挤，而且实现起来也非常困难。因此采取了这种“一个应用程序覆盖整个显示客户区”的做法。流行的智能终端操作系统，比如 Android 和 iOS，也采用了这种实现方式。

Hello China V1.75 版本的 GUI 实现还是比较基础的，只实现了最简单的窗口机制和绘图机制。但对于很多功能相对单一、用户交互不是很密集的应用程序来说，已经足够使用了。在后续版本的实现中，将根据需要增加其他的图形功能。

在后面的内容中，我们将简单介绍 GUI Shell 的实现，以便读者对操作系统图形 shell 有一个大概的认识。还是那句话，虽然是以 Hello China 的图形 shell 为例来讲，但基本原理是通用的。通过 Hello China GUI Shell 这片树叶，读者可感受到通用操作系统 GUI Shell 的整个森林。

虽然 V1.75 版 GUI Shell 的实现比较简单，但在几页或十几页的篇幅内分析全部代码，也是不可能的。我们采用“情景化”的分析方法，分析 GUI Shell 的初始化、应用程序的加载、GUI Shell 的退出等三个场景，通过这三个场景的分析，达到以点带面的效果。

11.9.2 GUI Shell 的启动和初始化

由于 GUI Shell 是用户启动其他 GUI 应用程序的基础, 因此在 GUI 模块初始化的时候, 就必须启动 shell, 否则用户将无法与计算机交互。前面介绍过, GUI 模块被加载到内存后, 操作系统会创建一个叫做“GUI”(注意不是 GUIShell)的线程, 然后等待这个线程的结束。GUI 线程被调度执行后, 就会以__init 函数作为入口函数开始 GUI 功能的初始化。为了加深读者印象, 再把__init 函数的部分相关代码摘录如下:

```
[gui/guientry.cpp]
DWORD __init(LPVOID)
{
    .....

    //Now create RAWIT thread to receive and dispatch all events(input) in GUI mode.
    hRawInputThread = CreateKernelThread(
        0, //Use default stack size.
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_HIGH,
        RAWIT,
        NULL,
        NULL,
        "GUIRAWIT");
    if(NULL == hRawInputThread) //Can not create the RAW input thread.
    {
        goto __TERMINAL;
    }
    hLastFocusThread = SetFocusThread(hRawInputThread);
    GlobalParams.hRawInputThread = hRawInputThread;

    hGUIShell = CreateKernelThread(
        0, //Use default stack size,it's 16K.
        KERNEL_THREAD_STATUS_READY,
        PRIORITY_LEVEL_NORMAL, //Normal priority.
        GuiShellEntry,
        NULL,
        NULL,
        "GUIShell");
    if(NULL == hGUIShell)
    {
        goto __TERMINAL;
    }
    .....
}
```

__init 函数首先初始化显示硬件, 初始化全局对象等, 这在前面已讲解, 不再赘述。接着, 初始化函数会创建原始输入线程(RAWIT), 并设置为当前输入焦点线程。自此开始, 所有用户主动输入, 都会被送到 GUI 模块。但这时候由于系统中还没有创建另外的 GUI 线程, 任

何消息都会被 RAWIT 线程丢弃。严格来说，是 RAWIT 线程试图搜索窗口树，由于这时候的窗口树是一棵空树，因此不会搜索到任何目标线程，从而导致消息被丢弃。

接下来，初始化函数创建了 GUIShell 线程，也就是图形用户 shell 线程。这个线程的入口点是 GuiShellEntry 函数。创建完毕，GUI 线程就进入等待状态，等待 RAWIT 和 GUIShell 运行完毕。需要注意的是，这时候的 RAWIT 和 GUIShell 线程仅仅是被创建完成了，还不一定会被调度运行。因为若系统中存在更高优先级的核心线程且状态为 READY 的话，那么这两个线程会一直处于就绪状态，不会得到调度。

GUIShell 线程一旦得到调度，就会从 GuiShellEntry 函数处开始执行。下面就是 GuiShellEntry 的入口代码：

```
[gui/kthread/guishell.cpp]
DWORD GuiShellEntry(LPVOID)
{
    MSG Msg;
    WORD x = 0;    //Mouse x scale.
    WORD y = 0;    //Mouse y scale.
    __WINDOW_MESSAGE wmsg;

    if(!InitGuiShell())
    {
        return 0;
    }
    while(TRUE)
    {
        if(GetMessage(&Msg))
        {
            switch(Msg.wCommand)
            {
                case KERNEL_MESSAGE_TIMER:
                    wmsg.hWnd = (HANDLE)Msg.dwParam;
                    wmsg.message = WM_TIMER;
                    wmsg.wParam = 0;
                    wmsg.lParam = 0;
                    SendWindowMessage(wmsg.hWnd,&wmsg);
                    break;
                case KERNEL_MESSAGE_WINDOW:
                    DispatchWindowMessage((__WINDOW_MESSAGE*)Msg.dwParam);
                    break;
                case KERNEL_MESSAGE_TERMINAL:
                    goto __TERMINAL;
                default:
                    break;
            }
        }
    }
}
```



```
__TERMINAL:
return 0;
}
```

这段代码看起来是不是很熟悉？这是一个典型的基于消息驱动的线程入口函数。该函数首先调用 `InitGuiShell` 函数，做了一些初始化工作，然后进入一个消息循环。在这个消息循环中，处理了三个典型消息：定时器消息、窗口消息和终止消息。与字符模式的 `shell` 不同，GUI Shell 是可退出的，在图形模式下，用户按下 `Ctrl+Alt+Del` 组合键，即可退出图形模式。终止消息（`KERNEL_MESSAGE_TERMINAL`）就是用于通知 GUI Shell，用户已经发起了退出操作。这个消息是键盘驱动程序根据用户输入自动生成的，即如果用户连续按下了 `Ctrl`、`Alt` 和 `Del` 组合键，键盘驱动程序不但会给操作系统内核发送三个键盘按下消息，分别对应 `Ctrl`、`Alt` 和 `Del`，还会额外发送一个 `TERMINAL` 消息。DIM 不会对这个消息做进一步处理，只会把这个消息透传给当前输入焦点线程。此时的当前焦点线程是 `RAWIT`，于是 `RAWIT` 会通知 GUI Shell 启动退出操作。详细的传递机制，在本章后续章节中会有详细介绍。

定时器消息和窗口消息的处理机制，在前面相关章节中已介绍过，不再赘述。总之，GUI Shell 本质上是一个基于 GUI 接口的用户线程，这个线程受消息驱动。与普通应用程序不同的是，GUI Shell 是 GUI 模式下的第一个用户线程。

所有 GUI Shell 的初始化工作，是在 `InitGuiShell` 函数内完成的。这个函数创建了 GUI Shell 的所有窗口（应用程序显示窗口、系统信息显示窗口、辅助信息显示窗口等）。为了简便，我们摘取该函数的部分代码如下：

```
[gui/kthread/guishell.cpp]
static BOOL InitGuiShell()
{
    .....

    GuiGlobal.hMainFrame = CreateWindow(
        0,
        NULL,
        0,
        INDICATEBAND_HEIGHT + 1,
        cxScreen - APPLICATIONBAND_WIDTH - 3, //Reserve space for separate line.
        cyScreen - INDICATEBAND_HEIGHT - TASKBAND_HEIGHT - 2,
        MainFrameProc,
        NULL,
        NULL,
        COLOR_APPLAUNCHER,
        NULL);
    if(NULL == GuiGlobal.hMainFrame)
    {
        return FALSE;
    }
    .....
}
```

```
return TRUE;
}
```

上述代码片断创建了管理应用程序的窗口，即 GUI 模式下屏幕左边用于呈现所有已安装应用程序的窗口。其他窗口，比如辅助信息窗口（呈现了时钟和日历）、系统信息显示窗口等，都是在该函数内完成创建的。

应用程序管理窗口的窗口函数就是 `MainFrameProc`，这是要重点介绍的窗口函数。在窗口创建完成时，系统会给应用程序管理窗口发送 `WM_CREATE` 消息，即以 `WM_CREATE` 作为参数，调用 `MainFrameProc` 函数。在 `MainFrameProc` 函数的 `WM_CREATE` 消息处理代码中，加载了系统中的所有应用程序。下面是其代码：

```
[gui/kthread/guiwproc.cpp]
DWORD MainFrameProc(HANDLE hWnd,UINT message,WORD wParam,DWORD lParam)
{
    switch(message)
    {
        case WM_CREATE:
            LoadAppProfile(hWnd);
            break;
        case WM_DRAW:
            break;
        case WM_COMMAND:
            LaunchApplication(wParam);
            break;
        case WM_CHILDCLOSE:
            DestroyWindow((HANDLE)lParam);
            break;
        case WM_LBUTTONDOWN:
            MessageBox(hWnd,"刚才您用鼠标双击了屏幕。","Notification",MB_OKCANCEL);
            break;
        default:
            break;
    }
    return DefWindowProc(hWnd,message,wParam,lParam);
}
```

这个函数结构比较简单，是一个典型的窗口函数的结构。针对每个窗口消息，该函数又调用了其他的函数来完成具体功能。比如针对 `WM_CREATE` 消息，该函数调用 `LoadAppProfile`，来完成所有应用程序的枚举工作。这里“应用程序的枚举”，实际上是遍历 `HCGUIAPP` 目录，把该目录下所有扩展名是 `HCX` 的文件检查一遍，如果确认是一个合法的可执行文件，则提取其相关信息，形成一个 `application profile`，然后把这个 `profile` 显示在屏幕上。这里所说的应用程序 `profile`，指的是能够描述应用程序基本信息的一个数据结构，包含了应用程序文件名、应用程序可视化名字、应用程序文件大小、应用程序图标等。这些信息用于加载一个特定的应用程序。`LoadAppProfile` 的实现代码比较繁琐，在此就不列举了，

只把该函数的大致执行过程描述一下，读者可根据下面的描述自行阅读该函数的代码（位于 [gui/kthread/launch.cpp]）：

（1）该函数首先检查应用程序目录下（C:\HCGUIAPP）所有已安装的 GUI 应用程序。在检查的时候，该函数调用了 FindFirstFile 和 FindNextFile 等目录枚举函数。针对每个扩展名是.HCX（或对应的小写形式）的文件，该函数读取该文件的开始部分信息，然后做进一步检查。如果发现是一个合法的 HCX 应用程序，则会进一步读取应用程序文件，提取应用程序图片、可视化名字等信息，然后保存在一个列表中。

（2）在枚举完应用程序信息后，就形成了一个应用程序 profile 列表（代码中的 pAppProfile 变量），这个列表包含了所有安装在应用程序目录下的合法应用程序。针对每一个应用程序 profile，该函数调用 CreateBitmapButton，在窗口内创建一个图形按钮。这个图形按钮显示应用程序图标和应用程序可视化名字两个信息。

（3）所有应用程序 profile 的图形按钮创建完毕，该函数就执行完了。

图形按钮（bitmap button）是 V1.75 实现的一个简单的用户交互控件。与普通的按钮控件不同的是，该控件可以显示一个 bitmap 格式的位图信息。一旦用户单击了按钮，按钮会向其父窗口发送 WM_COMMAND 消息。WM_COMMAND 消息的 wParam 参数携带了一个 ID，这个 ID 标明了图形按钮的一个标识。回过去看 MainFrameProc 函数的代码，该函数处理了 WM_COMMAND 消息，这个消息就是由图形按钮发送过去的。

在处理图形按钮通知的 WM_COMMAND 消息时，调用了 LaunchApplication 函数。顾名思义，这个函数就是根据图形按钮的索引 ID，启动其对应的应用程序。这是本章 11.9.3 节的内容。

至此，GUI Shell 初始化这个场景就介绍完了。从本质上说，GUI Shell 是一个用户线程，其特别之处在于，它是 GUI 模块初始化后的第一个用户线程。与其他基于 GUI 的线程相通，它以消息为驱动机制，根据消息调用不同的窗口函数，从而实现不同的功能。以线程机制为基础，把一个复杂的功能，划分为一个个独立的模块（线程）分别实现，并以消息作为模块（线程）之间的交互机制，是一种非常灵活且富有弹性的软件开发方式。

11.9.3 加载一个应用程序

应用程序的加载是 GUI Shell 最主要的工作。一旦用户选择了一个图形按钮，该图形按钮就会给管理应用程序的窗口发送一个 WM_COMMAND 消息，消息的 wParam 参数携带了图形按钮的 ID。这个 ID 也是对应的应用程序在 application profile 列表中的位置。这样 GUI 模块即可根据该 ID，定位到应用程序的 profile，从而找到应用程序可执行文件（HCX 文件）的文件名和路径，把该应用程序加载到内存并执行。

下面是 LaunchApplication 的关键代码片断，该函数是应用程序管理窗口在处理 WM_CREATE 消息时调用的：

```
[gui/kthread/launch.cpp]
void LaunchApplication(DWORD dwButtonID)
{
    __APP_PROFILE* pProfile = pAppProfileList;
    while(pProfile->dwButtonID != dwButtonID)
```

```
{  
    pProfile = pProfile->pNext;  
    if(NULL == pProfile) //End of list.  
    {  
        break;  
    }  
}  
.....  
//Find the appropriate application profile,launch it.  
LoadHCX(pProfile);  
return;  
}
```

代码首先根据图形按钮的 ID，搜索整个应用程序 profile 链表。需要注意的是，应用程序 profile 链表（pAppProfileList）是一个全局变量，因此在这里直接引用即可。当然，这种实现方式有些低效，尤其是链表中的元素非常多的时候。后续可通过 hash 表等方式进行优化。

在找到应用程序的 profile 之后，就调用 LoadHCX 函数，加载这个应用程序。加载过程比较简单，无非是把文件读入内存，做一番检查，如果确认是一个合法的 HCX 程序，则创建一个线程，执行该程序即可。这里有两个地方需要重点说明一下：

（1）与 PE 等其他可执行文件类似，HCX 文件也有一个入口地点，不过目前的定义是其入口地点，就是 HCX 文件的开始处。这样做的目的是为了使得加载过程简单，而且也似乎没有明显不足。

（2）LoadHCX 函数会单独创建一个用户线程，用于执行应用程序。与字符 shell 的执行方式不同，GUI Shell 在创建完用户线程之后，本身仍然会继续运行，而不像字符 shell 一样等待应用程序结束。也就是说，GUI Shell 本身与它创建的任何用户线程没有本质区别，除了优先级可能会有不同。

11.9.4 GUI Shell 的退出

与 Windows 等完全基于 GUI 的操作系统不同，Hello China 在缺省情况下是基于字符 shell 的。用户通过 gui 命令，进入图形界面，这时候 GUI Shell 线程才会被创建。在图形模式下，用户可以按下 Ctrl+Alt+Del 组合键，退出 GUI 模式，返回到字符界面。这时候由于操作系统尚在运行，因此 GUI 模式的退出，必须像任何应用程序一样，完成扫尾工作。这包括释放相关资源，关闭需要 GUI 功能支撑的应用程序，等等。而如果是完全基于 GUI Shell 的操作系统，比如 Windows 等操作系统，在 GUI Shell 退出的时候，实际上操作系统已结束运行，这时候的退出操作就无需考虑资源的释放等工作。

在当前的实现方式中，用户一旦按下这个组合键，键盘驱动程序会截获该组合键，然后给操作系统核心发送一个 TERMINAL 消息。由于 GUI 模块在初始化时，会设定 RAWIT 线程为当前输入焦点线程，所以 TERMINAL 消息会首先被发送到 RAWIT 线程。这时候 RAWIT 线程会执行退出操作，下面是 RAWIT 线程处理 TERMINAL 消息的代码：

```
DWORD RAWIT(LPVOID)
{
    .....
    while(TRUE)
    {
        if(GetMessage(&Msg))
        {
            switch(Msg.wCommand)
            {
                .....
                case KERNEL_MESSAGE_TERMINAL: //System terminal message.
                    if(GlobalParams.hGUIShell)
                    {
                        msg.wCommand = KERNEL_MESSAGE_TERMINAL;
                        SendMessage(GlobalParams.hGUIShell,
                                    &msg); //Send terminate message to GUI shell thread.
                    }
                    goto __TERMINAL; //End this thread.
                    break;
                default:
                    break;
            }
        }
    }
    __TERMINAL:
    return 0;
}
```

上述代码中的黑体部分，是处理 `TERMINAL` 消息的代码。`RAWIT` 线程首先判断 `GUI Shell` 线程是否存在。一旦 `GUI` 线程被创建，该线程的句柄会被存储在 `GlobalParams.hGUIShell` 中。因此判断该句柄是否为 `NULL`，即可确定 `GUI` 线程是否被创建。之所以做一个判断，是因为 `GUI` 模块在初始化的时候，是首先创建 `RAWIT` 线程的，然后再创建 `GUI Shell` 线程。这样有可能出现 `GUI Shell` 线程尚未被创建，用户就按下了 `Ctrl+Alt+Del` 组合键的情况。这样 `RAWIT` 线程就必须退出了，但这时候 `GUI Shell` 线程还未创建。因此在这里必须检查一下。如果 `GUI Shell` 被创建，则需给 `GUI Shell` 也发送一个结束消息，然后 `RAWIT` 线程也退出。

在 `GUI Shell` 收到 `TERMINAL` 消息后，也会做一些收尾工作，比如销毁创建的几个窗口对象，销毁应用程序 `profile` 列表，等等。然后即可结束运行。

`RAWIT` 和 `GUI Shell` 两个线程都结束运行之后，`GUI` 线程会恢复执行。这时候 `GUI` 线程执行的就是 `GUI` 模块的退出清理操作。为了加深读者印象，再把 `GUI` 模块的 `__init` 函数相关代码摘录如下：

```
[gui/guientry.cpp]
DWORD __init(LPVOID)
```

```
{
    .....
    WaitForThisObject(hRawInputThread);
    WaitForThisObject(hGUIShell);
    //GUI Shell 运行结束，返回字符模式，
    //并恢复最后保存的线程为当前焦点线程
    SetFocusThread(hLastFocusThread);
    bResult = TRUE;

    __TERMINAL:
    //Destroy all kernel threads in GUI mode.
    if(hRawInputThread)
    {
        DestroyKernelThread(hRawInputThread);
    }
    if(hGUIShell)
    {
        DestroyKernelThread(hGUIShell);
    }
    Video.Uninitialize(&Video);
    WindowManager.Uninitialize(&WindowManager);
    //Switch back to text mode.
    SwitchToText();
    return bResult;
}
```

上述黑色字体部分代码，就是 GUI 线程在创建完 RAWIT 和 GUI Shell 线程之后，进入等待状态的两行代码。在 RAWIT 和 GUI Shell 两个线程不结束的情况下，GUI 线程一直处于等待状态。而一旦这两个线程执行完毕，GUI 线程会恢复执行。这时候黑体部分后面的代码将得到执行。这部分的代码，主要是销毁了 RAWIT 和 GUI Shell 这两个核心对象，释放了 Video 对象的相关资源（Video.Uninitialize 函数），释放了 Windows 管理器的相关资源（WindowManager.Uninitialize 函数），切换到字符模式，然后即可退出。

一旦 GUI 线程结束运行，字符模式的 shell 会被唤醒（字符 shell 会等待 GUI Shell 运行结束），然后重新进入字符 shell。

11.10 GUI 模块的开发方法

在本章的最后，对 GUI 模块的开发方法做一简单介绍，目的是让读者能够修改 GUI 模块的功能。与 master 等核心模块一样，GUI 模块也是在 Visual C++ 6.0 开发环境下开发和编译的。对编译环境的设置，与其他核心模块一样，请参考附录 C。下面简要描述一下修改 GUI 模块的方法：

（1）GUI 模块的源代码包含在[/gui]目录下。这也是一个 VC 工程项目文件夹，项目相关的控制文件和配置文件，也包含在这个文件夹下。双击 hcngui.dsw，即可把所有 GUI 有关的

文件和资源加载到集成开发环境。在集成开发环境中对 GUI 模块进行修改。

(2) 修改完成后, 即可选择 build→batch build 菜单, 然后点击 Rebuild all, 即可完成所有模块的重新构建。

(3) 完成之后形成的 DLL 文件需要经过 process 工具进行处理, 具体的方法请参考第 14 章。

(4) 处理完成之后, 需要把 GUI 相关的支撑资源, 比如汉字库等, 一起链接到 GUI 模块中。这是通过 append 工具完成的。

这样完成上述处理后, 即可形成 HCNGUI.BIN 模块。把这个模块复制在 Hello China 的系统目录 (PTHOUSE 目录) 下即可。进入字符界面后, 运行 gui 命令, 操作系统就会在 pthouse 目录下寻找 hcngui.bin 模块, 然后加载运行。

如果读者希望对 GUI 模块进行修改, 建议直接使用 VC 6.0 打开 GUI 工程进行修改, 修改完成之后, 把编译的 hcngui.dll 文件 (位于 release 目录下) 复制到[/gui/guimaker]目录下, 然后直接运行 guimaker 批处理命令, 即可生成 hcngui.bin 模块。guimaker 是一个 DOS 批处理文件, 调用了该目录下的相关工具来生成 GUI 模块。下面是其内容:

```
:/gui/guimaker/guimaker.bat
del hcngui.bin
process -i hcngui.dll -o hcngui.bin
append -s hcngui.bin -a ASC16 -b 20000
append -s hcngui.bin -a HZK16 -b 30000
```

首先删除原有的 hcngui.bin 模块, 然后调用 process 工具, 对新复制的 hcngui.dll 进行处理, 处理后的文件为 hcngui.bin。然后连续两次调用 append 工具, 把 ASCII 码点阵字库和汉字点阵字库追加到 hcngui.bin 模块上, 最终生成 hcngui.bin 模块。

按照 Hello China V1.75 的实现, hcngui.bin (处理前) 被加载到内存后的地址是 0x160000, 而 ASCII 点阵字库和汉字点阵字库则分别被加载到 0x180000 和 0x190000 位置处。为了使读者对各个模块的加载位置有更清楚的了解, 我们再把 Hello China 进入保护模式后的内存布局图放在这里, 如图 11-20 所示。

注意上述布局图是按照内存从小到大的顺序编排的, 右面的十六进制数字代表各个模块的起始地址, 后面括号内的数字代表这个模块的实际大小。



图 11-20 Hello China 稳定后的内存布局

第 12 章 文件系统及其实现

12.1 文件系统概述

文件系统是操作系统最核心的功能之一。任何一个相对完善的操作系统，必须能够对外部存储设备进行管理。而对存储设备有效管理的最通用的方法就是文件系统。作为一个功能相对完善的操作系统，Hello China V1.75 版也实现了一个文件系统管理框架，并实现了基本的 FAT32 文件系统功能和 NTFS 文件系统的只读功能。在本章中，以 FAT32 文件系统为例，详细介绍 Hello China 的文件系统实现原理。还是那句话：虽然我们是以 Hello China 为例的，但是其中的大部分概念都是相通的，适用于任何操作系统。

先看一下文件系统相关的基本概念。

12.1.1 文件系统的基本概念

首先界定一下文件系统的概念。严格来说，文件系统应该叫做文件管理系统，本质上是一种规则或算法，用于对存储介质进行管理。比较常见的文件管理系统有 FAT32、NTFS、EXT、CDFS 等，每种文件管理系统都有其特定的适应场合和适应对象。文件管理系统是以卷（volume）为单位对存储介质进行管理的，卷进一步被文件管理系统分为目录和文件。注意卷与磁盘分区的区别。一个磁盘分区是一段连续的磁盘存储空间，一个物理磁盘可以分为一个或多个逻辑分区。在个人计算机上，一个卷对应于一个磁盘分区是最常见的情况。但是一个卷也可以包含多个磁盘分区，比如在存储领域应用广泛的 RAID 协议，就是把一个磁盘上的几个分区，或者跨越多个物理磁盘上的多个分区，组合成一个逻辑的卷。一个卷由单一的文件管理系统进行管理，在管理之前，首先进行格式化。所谓格式化，就是按照文件管理系统的要求，在卷上创建特定的数据结构，比如扇区使用情况跟踪表、用于管理目录和文件的原始数据等。在后续的描述中，文件系统统一指文件系统管理系统，比如 NTFS、FAT32 等。而对于一个采用某种文件系统格式化的分区，称之为“文件卷”。

再看一下文件系统的命名规则。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（:）来标识一个文件卷，一个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对应的卷标识为 C:，第二个硬盘分区对应的卷标识为 D:，第三个为 E:，等等。

对于卷上的文件，分两类对待：一类是目录，这类文件可以理解为一个容器，里面进一步包含了目录和普通文件。目录的内容即是其包含的下级目录（子目录）和普通文件的相关信息。这里的相关信息，指的是子目录或普通文件的名称、大小、创建/修改时间，等等。用不同的文件系统格式化的卷，目录内容也不同。另外一类就是普通的文件，所有用户数据

都存放在普通文件中。普通文件必须位于某个特定的目录下。虽然在逻辑上可以分为目录和普通文件，但是大多数文件系统都把目录看作一种特殊类型的普通文件。与普通文件不同的是，目录的内容是其子目录和文件的属性信息。

目录是可以嵌套的，比如一个目录文件（假设为 Directory1），进一步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（相应的标识符为 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符。当前情况下，下列字符不能出现在文件的命名中：

```
/\=*?
```

12.1.2 文件系统的操作——fs 程序

为了使读者对 Hello China 文件系统有一个大致的印象，先看一下操作文件系统的内置程序——fs。这个程序运行在字符模式下，可以完成文件卷枚举（列出系统中所有的文件卷）、浏览目录中的文件、显示文件内容、创建目录、创建文件并写入等。需要说明的是，为了安全考虑，缺省情况下禁止了硬盘的写入操作。这样虽然 fs 程序提供了文件卷的写入操作，但不能使用。如果希望使用文件系统的写入功能，则需要启用硬盘写入功能。

启动到字符 shell 之后，输入 fs 并按 Enter 键，即可进入 fs 程序的操作界面。图 12-1 是一个运行截图。

```
[system-view]fs
#fslist
fs_id      fs_label    fs_type
-----
C:         NTFS_VOL    0
#help
fslist    : Show all available file systems.
dir       : Show current directory's file list.
cd        : Show or change current directory.
vl        : Change current fs's volume label.
md        : Create a new directory.
mkdir     : Alias command of md.
del       : Delete one file from current directory.
rd        : Delete one sub-directory from current directory.
ren       : Change file or directory's name.
type      : Show a specified file's content.
copy      : Copy file to other location, or reverse.
use       : Set current file system.
exit      : Exit the application.
help      : Print out this screen.
#_
```

图 12-1 fs 程序的运行结果

fslist 命令可列出系统中所有的文件卷。在 fs 的提示符下运行 help 命令, 即可显示所有支持的文件系统命令。

运行 dir 命令, 即可列出当前目录下的所有文件和目录, 如图 12-2 所示。

```
[system-view]fs
#cd pthouse
#dir
  File_Name      FileSize      F_D
    APPEND.exe    155706        FILE
      ASC16       4096         FILE
    batch.bat     342           FILE
    hcngui.bin    464224        FILE
      hzk16       267616        FILE
    master.dll    135648        FILE
    modcfg.ini    168           FILE
    network.exe   736           FILE
    process.exe   28672         FILE
#_
```

图 12-2 dir 命令执行结果

cd 命令可改变当前目录。进入 fs 程序后, 缺省的当前目录是 C:分区的根目录, 输入 cd pthouse 命令后, 即可进入 Hello China 的系统目录。

在当前目录下, 运行 type 命令 (后跟一个文件名), 即可显示文件内容。当前, 这个命令可以显示任何文件, 但人可识别的只有文本文件。

fs 程序完全是通过调用 Hello China 提供的 API 接口函数实现的。读者可以通过调用 API 函数, 实现更复杂的文件操作程序。在对 Hello China 的文件系统有一个感性的认识之后, 我们正式进入主题。首先大致了解一下 FAT32 文件系统的基本原理, 后续将以 FAT32 文件系统的实现为例进行讲解。之所以选择 FAT32 而不是 NTFS, 是因为 FAT32 有 Microsoft 正式发布的规范可以参考。而 NTFS 的规范则一直未发布, 对于其运行机理, 都是通过大量的猜测和试验得到的, 并非 Microsoft 公司标准。因此大部分 NTFS 文件系统的实现, 都只实现了只读功能, 当然, 微软的实现除外。

12.2 FAT32 文件系统原理

在本节中, 我们对 FAT32 文件系统的原理做一简要说明, 为后续内容的阅读奠定基础。当然, 如果读者对 FAT32 文件系统的规范已经非常熟悉, 可以略过本节。本节内容的主要参考资料, 就是 Microsoft 公司发布的 FAT32 文件系统规范 (FAT32 File System Specification)。读者也可以直接阅读该规范来了解 FAT32 文件系统的原理。

12.2.1 FAT32 卷的布局

一个被格式化为 FAT (注意不仅仅是 FAT32, 还有可能是 FAT12 和 FAT16) 格式的文件分区 (文件卷), 其总体布局如下:

(1) 保留区, 这是从分区第一个扇区开始的连续几个扇区, 其中第一个扇区内存放了文件卷有关的关键信息, FAT32 文件系统正是靠第一个扇区内的一些信息, 来进一步定位到下列几个区域。

(2) FAT 区, 即文件分配表所在区域。这也是一片连续的磁盘扇区, 在格式化的时候就已确定。一般情况下, FAT 区包含两个文件分配表, 这两个相互备份, 以最大可能地降低数据丢失风险。

(3) 根目录区, 这个区在 FAT32 中不存在。

(4) 文件和目录数据区。这是真正的存储文件和目录的地方。

下面的内容, 就是对这几个区域的内容和作用进行说明。

12.2.2 引导扇区和 BPB

引导扇区即是分区的第一个扇区, 这个扇区对文件卷来说至关重要, 因为所有与文件系统有关的信息, 都存储在这个扇区上, FAT32 也不例外。FAT32 文件系统驱动程序在试图分析一个分区是不是 FAT32 文件卷的时候, 首先读取这个扇区, 然后根据这个扇区内的数据做进一步判断。需要注意的是, 这个引导扇区并不一定是真正的引导扇区, 因为里面可能没有引导代码。如果对应的分区不是活动分区, 则是否存在引导代码, 对系统来说是不重要的。但是文件系统相关的信息一定要存在, 否则文件系统驱动程序将无法识别这个分区。

引导扇区一般为 512B, 其中对文件系统有用的数据, 是从偏移 0x0B (即十进制的 11) 开始的。从 0x0B 开始的 53 个字节, 有一个专业的名字, 叫做 BPB (BIOS Parameter Block, BIOS 参数块)。至于为什么叫这个名字, 我们无需理会, 只需知道这部分数据对所有 FAT 文件系统, 包括 FAT12/16 和 FAT32, 都是通用的。接下来从偏移 0x40 开始的 26 个字节, 叫做扩展 BPB, 只有 FAT32 卷中才存在。下面我们详细解释引导扇区的每个字段, 了解了这些字段的含义, 对 FAT 文件系统的工作原理也就有一个大致了解了。为了解释方便, 我们以表格的形式呈现, 如表 12-1 所示。

表 12-1 引导扇区各字段的含义

偏 移	长度/Byte	字 段 含 义
0x00	3	三个字节的跳转指令, 跳转到真正的引导代码
0x03	8	厂商特定的数据, 比如厂商的标志字符串、版本号等
0x0B	2	每扇区字节数 (Bytes Per Sector), 即硬件扇区的大小。本字段合法的十进制值有 512、1024、2048 和 4096。对大多数磁盘来说, 本字段的值为 512
0x0D	1	每簇扇区数 (Sectors Per Cluster), 一簇中的扇区数。由于 FAT32 文件系统只能跟踪有限个簇 (最多为 4 294 967 296 个), 因此, 通过增加每簇扇区数, 可以使 FAT32 文件系统支持更大的分区尺寸。一个分区缺省的簇大小取决于该分区的大小。本字段的合法十进制值有 1、2、4、8、16、32、64 和 128。Windows 2000 的 FAT32 实现只能创建最大为 32GB 的分区。但是, Windows 2000 能够访问由其他操作系统 (Windows 95、OSR2 及其以后的版本) 所创建的更大的分区
0x0e	2	保留扇区数 (Reserved Sector), 即分区保留区域, 也就是第一个 FAT 表开始之前的扇区数, 包括引导扇区。本字段一般为 32
0x10	1	分区上的 FAT 表个数 (Number of FAT), 一般为 2 个, 包含一个主用 FAT 和一个备份 FAT
0x11	2	根目录项数 (Root Entries), 只有 FAT12/FAT16 使用此字段。对 FAT32 分区而言, 本字段必须设置为 0
0x13	2	小扇区数 (Small Sector), 只有 FAT12/FAT16 使用此字段, 对 FAT32 分区而言, 本字段必须设置为 0
0x15	1	媒体描述符 (Media Descriptor), 提供分区所在硬件设备的信息。值 0xF8 表示硬盘, 0xF0 表示高密度的 3.5 寸软盘, 等等
0x16	2	每 FAT 扇区数 (Sectors Per FAT), 只供 FAT12/FAT16 使用, 对 FAT32 分区而言, 本字段必须设置为 0
0x18	2	每道扇区数 (Sectors Per Track), 即磁盘每个磁道上的扇区数量



(续)

偏 移	长 度	字 段 含 义
0x1A	2	磁头数 (Number of Head), 磁盘的磁头数。例如, 在一张 1.44MB 3.5 英寸的软盘上, 本字段的值为 2
0x1C	4	隐藏扇区数 (Hidden Sector), 该分区上引导扇区之前的扇区数。一般用于一个物理硬盘被分为多个分区的情况。这个数据记录了本分区之前存在的扇区数量
0x20	4	总扇区数 (Large Sector), 本字段包含 FAT32 分区中总的扇区数
0x24	4	每 FAT 扇区数 (Sectors Per FAT), 该字段只被 FAT32 使用, 用于记录分区中每个 FAT 表所占的扇区数。计算机利用这个数和 FAT 表数以及隐藏扇区数 (本表中所描述的) 来决定根目录从哪里开始
0x28	2	扩展标志 (Extended Flag), 该字段只被 FAT32 使用。这两个字节中各位的值及含义如下: 位 0-3: 活动 FAT 数 (从 0 开始计数, 而不是 1), 只有在不使用镜像时才有有效 位 4-6: 保留 位 7: 0 值意味着在运行时所有的 FAT 都是活动的 1 值表示只有一个 FAT 是活动的 位 8-15: 保留
0x2A	2	文件系统版本 (File system Version), 只供 FAT32 使用。高字节是主要的修订号, 而低字节是次要的修订号。本字段支持将来对该 FAT32 媒体类型进行扩展。如果本字段非零, 以前的 Windows 版本将不支持这样的分区
0x2C	4	根目录簇号 (Root Cluster Number), 只供 FAT32 使用, 用于记录根目录第一簇的簇号
0x30	2	文件系统信息扇区号 (File System Information Sector Number), 只供 FAT32 使用。在 FAT32 分区中, 有一个记录文件系统信息的扇区, 这个扇区中存放了 FSINFO (File System Information) 结构, 记录了文件系统相关的信息。这个字段即存放了 FSINFO 结构所在的扇区号, 其值一般为 1
0x34	2	备份引导扇区号, 这个字段只供 FAT32 使用, 一般为一个非零值, 这个非零值表示该分区保存引导扇区的副本的保留区中的扇区号。为了保险起见, 在分区的保留区域中, 预留了一个扇区, 用于保存引导扇区 (即分区的第一个扇区)。本字段的值一般为 6, 建议不要使用其他值
0x36	12	保留, 只供 FAT32 使用, 供以后扩充使用的保留空间。本字段的值总为 0
0x40	1	物理驱动器号 (Physical Drive Number), 与 BIOS 物理驱动器号有关。软盘驱动器被标识为 0x00, 物理硬盘被标识为 0x80, 而与物理磁盘驱动器无关
0x41	1	保留 (Reserved) FAT32 分区总是将本字段的值设置为 0
0x42	1	扩展引导标签 (Extended Boot Signature), 似乎只对 Microsoft 的操作系统有效
0x43	4	分区序号 (Volume Serial Number), 在格式化磁盘时所产生的一个随机序号, 它有助于区分磁盘
0x47	11	分区的卷标 (Volume Label), 用来保存卷标号。现在, 卷标被作为一个特殊文件保存在根目录中, 占用根目录中的一个文件目录项。即在实现的时候, 卷标的值是以根目录中的目录项为准的
0x52	8	系统 ID (System ID), FAT32 文件系统中一般取为 "FAT32", 但这只是一个约定, 并不能据此判断分区就是一个 FAT32 文件卷。对一个分区是否为 FAT32 文件卷的判断, Microsoft 给出了一套标准算法。这套算法综合考虑上述各个字段的取值, 然后根据计算结果的范围, 来确定分区到底是 FAT12、FAT16, 还是 FAT32。且给出了标准的 C 语言代码。在 Hello China 的实现中, 只是简单地使用了这段代码, 至于这段代码的准确含义, 说实话, 我现在也没有搞清楚。不过不要紧, 能够工作就行

从 0x5A 往后, 就是真正的引导代码了, 如果存在的话。引导扇区的最后两个字节是 0x55AA, 用于标志一个合法的引导扇区的结束。这样通过分析上述各个变量, FAT 文件卷的全局信息就可以把握了。实际上, 文件系统的实现代码, 也是通过综合分析上述各个字段, 来判断出文件系统的类型 (是 FAT12/16 还是 FAT32), 进而得到 FAT 表的开始位置和长度、根目录的开始位置等信息。这样一个可用的文件卷就建立起来。

12.2.3 文件分配表

文件分配表 (File Allocation Table, FAT) 是文件管理系统用来记录每个文件所分配的磁盘物理空间的表格, 它告诉操作系统或者用户, 文件的具体内容存放在磁盘分区的什么地方。

FAT 在分区上是紧接保留区域之后的。一般情况下，在一个分区上共有 FAT 表的两个副本，一个是基本 FAT 表，另一个是 FAT 表的备份。两者在磁盘上前后紧排在一起，其大小根据分区的大小不同而变化，一个原则就是，所有分区的簇，都必须在分区表中有对应的表项。在 FAT12 或者 FAT16 中，FAT 表之后紧接着是根目录，根目录之后是数据区。而在 FAT32 文件系统中，FAT 表之后即是文件数据区。

在分区上的每一个可用簇，都对应 FAT 表中的一个登记项。通过在对应簇号的登记项内填入特定值，来表明数据区中的该簇是否占用、空闲或者已损坏。损坏的簇是在格式化的过程中，通过 FORMAT 命令发现的。在一个簇中，只要有一个扇区有问题，该簇就被作为坏簇处理，不能够使用。

FAT 文件系统是以簇为单位给文件分配存储空间的，每个簇在 FAT 表中占用一个表项。所以，在 FAT 表中，簇编号即对应 FAT 表项的编号。每一个 FAT 表项，都作为一个簇的标志信息而存在。针对 FAT12、FAT16 和 FAT32 等不同的 FAT 版本，对应的 FAT 表项的长度分别为 1.5B、2B 和 4B。表 12-2 给出了不同 FAT 表项值的含义，虽然我们重点介绍 FAT32，但是也顺便列出 FAT12 和 FAT16 的相关内容，供读者参考，如图 12-3 所示。

表 12-2 特殊 FAT 表项的含义

FAT12 表项值	FAT16 表项值	FAT32 表项值	含义
000H	0000H	00000000H	标明对应的簇尚未分配
001H--FEFH	000H--FFEFH	00000001--FFFFFFEFH	对应的簇已占用，同时这个表项指向属于同一个文件的下一个簇号
FF0H--FF6H	FFF0--FFF6H	FFFFFFF0--FFFFFFF6H	保留
FF7H	FFF7H	FFFFFFF7H	坏簇
FF8H--FFFH	FFF8H--FFFFH	FFFFFFF8--FFFFFFFHH	文件结束标志

下面举一个例子，进一步说明文件分配表的工作原理。假设有一个文件，共占用了 4 个簇，如图 12-3 所示。

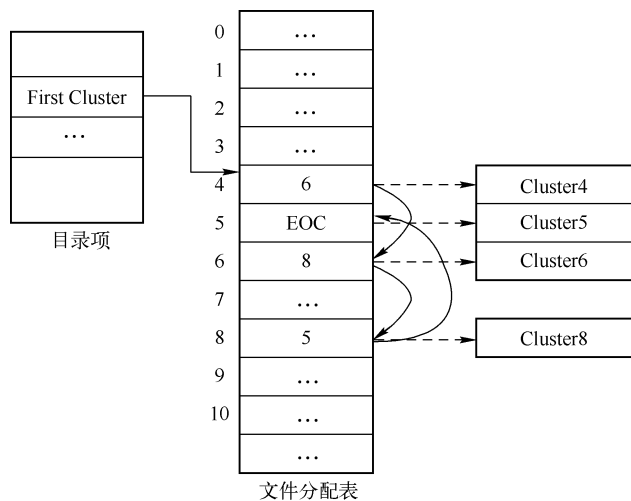


图 12-3 文件分配表的结构



在文件的目录项中，存放了文件内容的起始簇号。在这个例子中，文件的起始簇号是 4，因此分区上的第 4 个簇被这个文件占用。但是 4 号簇所对应的文件分配表中的表项值是 6，则说明该文件占用的下一个簇是 6。6 号簇所对应的 FAT 表项值是 8，则 8 号簇也分配给了该文件。8 号簇的 FAT 表项值是 5，说明 5 号簇也属于该文件。最后，5 号簇对应的 FAT 表项值是 EOC (0xFFFFFFFF8 到 0xFFFFFFFF 之间的任意值)，说明 5 号簇已是该文件的最后一个簇。

相信读者已经把 FAT 表的工作机理弄明白了。本质上，针对每个文件，都会有一个与之对应的簇链，这个簇链的第一个簇存放在文件目录项中，链中的每个簇的索引，都存放在其前一个簇的 FAT 表项中。不论对普通文件还是目录文件，都是以这种相同的方式来存储其内容。

这样在读取一个文件的时候，只要找到文件对应的目录项，从目录项中提取出其起始簇号，然后就顺藤摸瓜，按照簇链依次读取对应的磁盘簇即可。对于文件的写操作，则相对麻烦。因为要考虑是追加写——从文件的最后开始写，还是插入写——从文件的起始或中间开始写。如果是对文件进行追加写，则只需分配一个或一些空闲簇，把要写的内容写入这些簇，然后把这簇连接到文件簇链的末尾即可。对于插入写，需要分三步进行：

(1) 首先申请一个或几个空闲簇，然后连接到文件簇链的末尾。

(2) 把从待写入位置开始、到文件结尾（写入前）的内容，向后移动与待写入内容长度相同的字节数。假设待写入位置为偏移 100 处，写入内容长度为 20B，原始文件长度为 300B。则需要把从偏移 100 处、长度为 200B 的原始文件数据，向后移动 20B，空出待写入位置。

(3) 把待写入数据写入待写入位置即可。

总之，文件分配表是 FAT 文件卷的最重要数据结构。一旦 FAT 表损坏，可能会导致大量的数据丢失，因此 FAT 文件系统规范定义了 FAT 表项的保护机制，即在同一个分区上，要有两个 FAT 表，一个作为主用，另外一个为主用的备份。在读取的时候，只从主用 FAT 表上读取相关内容即可，除非主用 FAT 表损坏。在写入的时候，要同时把 cluster 的使用情况写入主用和备用 FAT 表。FAT 表也是访问频率最高的数据结构，为了提升访问速率，一般的 FAT 文件系统在实现的时候，都会为 FAT 表单独申请一些缓存，然后把整个 FAT 表或部分 FAT 表项读入缓冲区。这样在访问 FAT 表的时候，直接从缓冲区读取即可。

12.2.4 文件目录项

FAT 目录文件的主要内容就是文件目录项，每个目录项对应一个文件或子目录。当然，其他的信息，比如文件卷的卷标、长文件名等，也以目录项的形式存在。每个目录项的长度是固定的，都是 32B。表 12-3 是文件目录项的详细内容。

表 12-3 FAT 目录项详细内容

字节偏移（十六进制）	长度/Byte	含 义	
0x0~0x7	8	文件名，长度为 8B	
0x8~0xA	3	扩展名，长度为 3B	
0xB	1	属性字节，不同的值对应不同的属性，可以是几个值的组合	00000000（读/写）

(续)

字节偏移 (十六进制)	长度	含 义	
0xB	1	属性字节, 不同的值对应不同的属性, 可以是几个值的组合	00000001 (只读)
			00000010 (隐藏)
			00000100 (系统)
			00001000 (卷标)
			00010000 (子目录)
			00100000 (归档)
0xC	1	系统保留	
0xD	1	创建时间的 10 毫秒位	
0xE~0xF	2	文件创建时间	
0x10~0x11	2	文件创建日期	
0x12~0x13	2	文件最后访问日期	
0x14~0x15	2	文件起始簇号的高 16 位, 与起始簇的低 16 位一起, 形成文件第一个簇的簇号	
0x16~0x17	2	文件的最近修改时间	
0x18~0x19	2	文件的最近修改日期	
0x1A~0x1B	2	文件起始簇号的低 16 位	
0x1C~0x1F	4	表示文件的长度, 以字节为单位	

在目录中根据文件名查找一个特定文件或子目录的时候, 就是把目录文件的内容读入内存, 然后分析每个文件目录项, 看其中的文件名是否匹配待查找的文件名。如果匹配则查找成功, 否则继续下一个文件目录项的匹配。总之, 文件目录项的含义明确之后, 目录文件的本质就清楚了。目录文件无非是内容为文件目录项、长度为 32B 的整数倍的一个特殊文件。

12.2.5 文件的查找

在了解了上面一些概念后, FAT32 文件卷上的文件查找过程就很简单了。下面以一个具体的例子, 来说明查找过程。假设查找的文件路径和文件名为 “C:\HelloChina\Application\info.txt”, 则查找的过程如下:

(1) 文件系统代码首先逐个检查根目录内的目录项, 试图找到第一级目录项 “HelloChina”。注意, 根目录的起始扇区号, 是在 BPB 当中存放的, 文件系统可直接得到。任何目录内的目录项, 都符合上面介绍的 FAT32 目录项结构。

(2) 找到 HelloChina 目录项后, 在该目录项内存放了该目录的起始簇号、目录尺寸等内容。这时候文件系统代码会把 HelloChina 目录读入内存, 逐个匹配该目录下的目录项, 试图查找 “Application” 目录。当然, 如果不存在这个目录, 则查找失败, 我们假设这个目录是存在的。

(3) 在 HelloChina 目录下找到 Application 目录项后, 文件系统会进一步得到该目录的起始簇号和大小等数据, 然后再以 “info.txt” 为关键字搜索 Application 目录。如果搜索整个目录后找不到 info.txt 文件, 则查找失败。如果能够匹配到合法的目录项, 则查找成功。一旦找到对应的目录项, 文件的基础信息, 比如存放的起始 cluster 号等, 即可得到。



查找完成之后，目标文件的信息就得到了，于是可以进行进一步的读取、写入等操作。在上述过程中，任何目录的读取操作，如果涉及多个 cluster，则需要访问文件分配表 (FAT)，来找到属于该目录的 cluster，然后读取其中的内容。

12.3 Hello China 文件系统的实现

在实现 Hello China 操作系统的文件系统功能的时候，设定了下列几条原则：

(1) 支持的文件系统类型要能够扩展。当前版本 (V1.75) 实现了 FAT32 和 NTFS 两种文件系统，后续可通过增加文件系统驱动程序，很容易地支持其他类型的文件系统。这就要求文件系统的具体实现 (文件系统驱动程序) 能够与操作系统核心代码分离。

(2) 能够处理存储设备动态添加和删除功能。比如一旦一个 USB 存储设备被插入系统，文件系统就应该能够自动识别新增加存储设备的文件系统类型，并能够自动添加到系统中。如果存储设备被拔出，则需要自动删除对应的卷。

(3) 为应用程序提供与普通设备一致的访问接口，即应用程序可通过一组标准且一致的接口，来访问文件和物理设备。

当然，文件系统的实现代码要安全可靠，同时确保执行效率，这是最基本的要求。

在 Hello China 的实现中，把文件系统也纳入驱动程序管理框架范围之内，作为一种特殊的设备驱动程序来看待。在设备对象 (`_DEVICE_OBJECT`，见第 10 章) 的定义中，有一个变量是 `dwDevType`，即设备类型，这个变量指出了设备对象的大致类型。如果这个值是 `DEVICE_TYPE_FILE_SYSTEM`，那么这个设备对象就是一个文件系统设备对象。与普通设备对象一样，文件系统设备对象也是由 `IOManager` 对象管理的。我们还是从 `IOManager` 对象说起，在本章中，我们重点考察 `IOManager` 对象的文件管理相关内容。

12.3.1 IO 管理器

虽然 `IOManager` 对象的定义比较复杂，而且第 10 章已经介绍了 `IOManager` 对象的定义，但是我们还是把这个对象的定义再次呈现在这里，以方便阅读和说明：

```
[kernel/include/iomgr.h]
BEGIN_DEFINE_OBJECT(_IO_MANAGER)
    //全局变量，包含设备链表、设备驱动程序链表、文件卷数组、文件系统驱动程序数组等。
    _DEVICE_OBJECT*          lpDeviceRoot;
    _DRIVER_OBJECT*          lpDriverRoot;
    _FS_ARRAY_ELEMENT         FsArray[FILE_SYSTEM_NUM];
    _COMMON_OBJECT*          FsCtrlArray[FS_CTRL_NUM];
    //面向应用程序的服务接口。
    BOOL                      (*Initialize)(_COMMON_OBJECT* lpThis);
    _COMMON_OBJECT*          (*CreateFile)(_COMMON_OBJECT* lpThis,
        LPSTR                  lpFileName,
        DWORD                  dwAccessMode,
        DWORD                  dwShareMode,
        LPVOID                 lpReserved);
    BOOL                      (*ReadFile)(_COMMON_OBJECT* lpThis,
```



```

        __COMMON_OBJECT*  lpFileObject,
        DWORD              dwByteSize,
        LPVOID              lpBuffer,
        DWORD*              lpReadSize);
BOOL (*WriteFile)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* lpFileObject,
        DWORD              dwWriteSize,
        LPVOID              lpBuffer,
        DWORD*              lpWrittenSize);
VOID (*CloseFile)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* lpFileObject);
BOOL (*CreateDirectory)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName,
        LPVOID lpReserved);
BOOL (*DeleteFile)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName);
BOOL (*FindClose)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName,
        __COMMON_OBJECT* FindHandle);
__COMMON_OBJECT* (*FindFirstFile)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName,
        FS_FIND_DATA* pFindData);
BOOL (*FindNextFile)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName,
        __COMMON_OBJECT* FindHandle,
        FS_FIND_DATA* pFindData);
DWORD (*GetFileAttributes)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName);
DWORD (*GetFileSize)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* FileHandle,
        DWORD* lpdwSizeHigh);
BOOL (*RemoveDirectory)(__COMMON_OBJECT* lpThis,
        LPCTSTR lpszFileName);
BOOL (*SetEndOfFile)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* FileHandle);
BOOL (*IOControl)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* lpFileObject,
        DWORD              dwCommand,
        DWORD              dwInputLen,
        LPVOID              lpInputBuffer,
        DWORD              dwOutputLen,
        LPVOID              lpOutputBuffer,
        DWORD*              lpdwOutFilled);
BOOL (*SetFilePointer)(__COMMON_OBJECT* lpThis,
        __COMMON_OBJECT* lpFileObject,
        DWORD*              pdwDistLow,

```

```

                                DWORD*      pdwDistHigh,
                                DWORD      dwWhereBegin);
BOOL      (*FlushFileBuffers)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpFileObject);
    //面向设备驱动程序的服务接口。
__DEVICE_OBJECT* (*CreateDevice)(__COMMON_OBJECT* lpThis,
                                LPSTR      lpDevName,
                                DWORD      dwAttribute,
                                DWORD      dwBlockSize,
                                DWORD      dwMaxReadSize,
                                DWORD      dwMaxWriteSize,
                                LPVOID     lpDevExtension,
                                __DRIVER_OBJECT* lpDrvObject);
VOID      (*DestroyDevice)(__COMMON_OBJECT* lpThis,
                                __DEVICE_OBJECT* lpDevObj);
BOOL      (*LoadDriver)(__DRIVER_ENTRY DrvEntry);
    //专门提供给文件系统驱动程序的服务接口。
BOOL      (*AddFileSystem)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpFileSystem,
                                DWORD      dwAttribute,
                                BYTE*      pVolumeLbl);
BOOL      (*RegisterFileSystem)(__COMMON_OBJECT* lpThis,
                                __COMMON_OBJECT* lpFileSystem);

END_DEFINE_OBJECT()    //End of __IO_MANAGER.

```

这个全局对象提供了应用程序的服务接口，应用程序可以调用诸如 CreateFile/ReadFile 等函数，完成文件和设备的访问。同时还提供了面向设备驱动程序的服务接口，设备驱动程序可以调用 CreateDevice 等函数，完成设备对象的创建。这些内容在第 10 章中已做了详细介绍。接下来重点看 IOManager 对象提供的面向文件系统的变量和服务接口。

首先看 FsArray 数组，这是个类型为 __FS_ARRAY_ELEMENT、长度为 16 的数组，这个数组用于记录系统中的所有文件卷。下面是 __FS_ARRAY_ELEMENT 的定义：

```

[kernel/include/iomgr.h]
typedef struct{
    BYTE      FileSystemIdentifier;    //Such as C:,D:,etc.
    __COMMON_OBJECT* pFileSystemObject;
    DWORD      dwAttribute;           //File attribute.
    BYTE      VolumeLbl[VOLUME_LBL_LEN]; //Volumne ID.
} __FS_ARRAY_ELEMENT;

```

这个对象的第一个变量 FileSystemIdentifier 就是文件卷标识符，比如 C:、D: 等。在打开一个文件的时候，系统会在给出的文件全名（比如 C:\WINDOWS\DATA.TXT）中提取出卷标识符，然后逐个匹配这个数组。一旦卷标识符能够匹配，就可找到对应的 pFileSystemObject 指针，这个指针指向了具体的卷设备对象。卷设备对象是由文件系统驱动

程序针对卷来创建的。每当文件系统检测到一个属于自己管理的卷，则文件系统驱动程序会创建一个对应的卷设备对象，并调用 `AddFileSystem` 添加到系统中。卷设备对象对应的驱动程序，即是文件系统驱动程序。

找到卷设备对象之后，即可调用设备对象所对应驱动程序的 `DeviceOpen` 函数，来打开文件。`DeviceOpen` 是设备驱动程序对象提供的标准函数之一，用于打开一个设备。在文件系统中，这个函数用于打开一个指定的文件。

`dwAttribute` 则是文件卷的一些属性，比如是否可移动、是否为系统卷等。最后一个变量 `VolumeLbl` 则是卷标，定义为 13B 的长度，包含结尾的 0 字符。

在 `IOManager` 初始化的时候，`FsArray` 数组被初始化为全 0。在文件系统初始化的时候，会根据对存储设备或分区的检测情况，调用 `AddFileSystem` 函数，在这个数组中增加对应的文件卷。显然，`AddFileSystem` 函数的参数是与 `_FS_ARRAY_ELEMENT` 结构体中的成员变量一一对应的。这个函数的实现也很简单，无非是从头开始依次检查每个数组元素，试图找到一个未用（`FileSystemIdentifier` 为 0）的元素。如果能够找到一个，则把对应的数据添加到里面，否则返回 `FALSE`。下面是其实现代码，为了简便，我们省略了相关注释和无关紧要的代码：

```
[/kernel/kernel/iomgr.cpp]
static BOOL AddFileSystem(__COMMON_OBJECT* lpThis,
                        __COMMON_OBJECT* lpDevObj,
                        DWORD          dwAttribute,
                        BYTE*          pVolumeLbl)
{
    BOOL          bResult      = FALSE;
    __IO_MANAGER* pMgr        = (__IO_MANAGER*)lpThis;
    __DEVICE_OBJECT* pFileSystem = (__DEVICE_OBJECT*)lpDevObj;
    BYTE          FsIdentifier = 'C'; //First file system identifier.
    DWORD          dwFlags;
    int            i,j,k;

    //寻找一个元素，并占用它
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    for(i = 0; i < FILE_SYSTEM_NUM; i++)
    {
        if(0 == pMgr->FsArray[i].FileSystemIdentifier) //Empty slot.
        {
            break;
        }
    }
    if(FILE_SYSTEM_NUM == i) //数组全部被占用，没有空闲元素
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        goto __TERMINAL;
    }
    /* 找到一个空闲的数组元素，下面占用这个数组元素。首先计算一个可用的文件系统标识符。假
```

```

设数组中所有文件标识符中最大的是 X:，则计算出的文件标识符为 X+1*/
for(j = 0; j < FILE_SYSTEM_NUM; j++)
{
    if(pMgr->FsArray[j].FileSystemIdentifier >= FsIdentifier)
    {
        FsIdentifier += 1; //Use next one.
    }
}
pMgr->FsArray[i].FileSystemIdentifier = FsIdentifier;
pMgr->FsArray[i].pFileSystemObject = (__COMMON_OBJECT*)pFileSystem;
pMgr->FsArray[i].dwAttribute = dwAttribute;
//设置卷标
k = 0;
for(j = 0; j < VOLUME_LBL_LEN - 1; j++)
{
    if(' ' == pVolumeLbl[j]) //Skip space.
    {
        continue;
    }
    pMgr->FsArray[i].VolumeLbl[k] = pVolumeLbl[j];
    k += 1;
}
pMgr->FsArray[i].VolumeLbl[k] = 0; //Set terminator.
__LEAVE_CRITICAL_SECTION(NULL, dwFlags);
bResult = TRUE;
__TERMINAL:
return bResult;
}

```

代码比较简单，但是考虑到这段代码很可能是在设备驱动程序加载和初始化的时候被调用的，而 `FsArray` 是一个全局变量，因此使用关键区段对其进行保护，以免发生数据不一致的情况。另外，对于卷标的复制，也是采用了最简单的逐个字符复制的方式，这是为了最大限度地提高代码的安全性，毕竟 `strcpy` 等函数存在一定的风险。虽然这样做效率比较低，但是考虑到这个函数被调用的次数不会太多，而且卷标也不是太长，这个牺牲效率换安全的做法还是值得的。在 `Hello China` 的内核代码中，很多情况都是采用了这种策略，比如创建核心线程的时候，线程名字的复制也是采用了这种安全的方法。

下面再看 `FSCtrlArray`，这个数组就是文件系统设备对象数组。针对 `Hello China` 支持的每种文件管理系统，比如 NTFS、FAT32、CDFS 等，在这个数组中都会有一个文件系统设备对象与之对应。`IOManager` 初始化的时候，这个数组被清零。在文件系统驱动程序加载过程中，每个文件管理系统都会创建一个文件系统设备对象，并调用 `RegisterFileSystem` 添加到系统中。下面是 `RegisterFileSystem` 的实现代码（做了删减）：

```

[/kernel/kernel/iomgr.cpp]
static BOOL RegisterFileSystem(__COMMON_OBJECT* lpThis,
                               __COMMON_OBJECT* pFileSystem)

```

```

{
    __IO_MANAGER*      pManager = (__IO_MANAGER*)lpThis;
    DWORD              dwFlags;

    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    for(int i = 0;i < FS_CTRL_NUM;i ++)
    {
        if(NULL == pManager->FsCtrlArray[i]) //Find a empty slot.
        {
            break;
        }
    }
    if(FS_CTRL_NUM == i) //Can not find a empty slot.
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return FALSE;
    }
    //Insert the file system object into this slot.
    pManager->FsCtrlArray[i] = pFileSystem;
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return TRUE;
}

```

比较简单，就是在 `FsCtrlArray` 中寻找一个空闲的元素，然后把参数给出的文件系统设备对象复制到这个元素中。如果所有数组元素都不为 `NULL`，则失败返回。`FsCtrlArray` 的长度预定义为 4，即最大可以支持 4 个文件系统，这对目前的应用来说已经足够了。

图 12-4 进一步示意了卷设备对象数组 `FsArray` 和文件系统设备对象数组 `FsCtrlArray` 之间的关系：

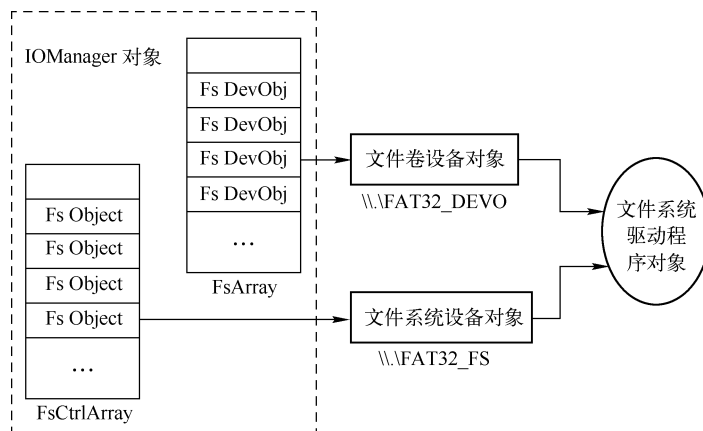


图 12-4 卷设备对象和文件系统设备对象的关系

针对系统中的每个文件卷，都会创建一个文件卷对象，并加入 `FsArray` 数组。针对系统中的每个文件管理系统，都会有一个文件系统设备对象与之对应。类型相同的文件卷和文件管理系统，都指向一个相同的文件系统驱动程序对象。



AddFileSystem 和 RegisterFileSystem 两个函数，以及对应的 FsArray 和 FsCtrlArray 两个数组，是文件系统功能可扩展、可支持存储设备的动态插拔的基础。下面我们以 FAT32 文件系统驱动程序为例，具体看一下文件系统的初始化过程。在初始化过程中会调用上述两个函数。

12.3.2 文件系统的加载和初始化

在 Hello China 的实现中，文件系统是作为设备驱动程序实现的，因此其遵循设备驱动程序的逻辑结构，需要实现一个 DriverEntry 函数。在文件驱动程序加载的时候，这个函数会被 IOManager 调用。

在当前版本的实现中，文件系统是作为操作系统核心模块实现的，与内核编译链接到一起。文件系统驱动程序的 DriverEntry 函数是一个内核全局函数，IOManager 初始化的时候会直接调用这个函数，完成文件系统的加载。需要注意的是，文件系统驱动程序的加载时机，必须位于存储设备驱动程序之前。比如在 PC 上的实现是，首先加载 FAT32 和 NTFS 文件系统，然后加载硬盘驱动程序。这样存储设备才有机会被文件系统处理。下面是 FAT32 文件系统的 DriverEntry 实现代码：

```
[/kernel/fs/fat32.cpp]
BOOL FatDriverEntry(__DRIVER_OBJECT* lpDriverObject)
{
    __DEVICE_OBJECT* pFatObject = NULL;

    lpDriverObject->DeviceClose = FatDeviceClose;
    lpDriverObject->DeviceCtrl = FatDeviceCtrl;
    lpDriverObject->DeviceFlush = FatDeviceFlush;
    lpDriverObject->DeviceOpen = FatDeviceOpen;
    lpDriverObject->DeviceRead = FatDeviceRead;
    lpDriverObject->DeviceSeek = FatDeviceSeek;
    lpDriverObject->DeviceWrite = FatDeviceWrite;
    lpDriverObject->DeviceCreate = FatDeviceCreate;

    //Create FAT file system driver object now.
    pFatObject = IOManager.CreateDevice((__COMMON_OBJECT*)&IOManager,
        FAT32_DRIVER_DEVICE_NAME,
        DEVICE_TYPE_FSDRIVER, //Attribute, this is a file system driver object.
        DEVICE_BLOCK_SIZE_INVALID, //File system driver object can not be read or written.
        DEVICE_BLOCK_SIZE_INVALID,
        DEVICE_BLOCK_SIZE_INVALID,
        NULL, //With out any extension.
        lpDriverObject);
    if(NULL == pFatObject) //Can not create file system object.
    {
        return FALSE;
    }
    //Register file system now.
```

```
if(!IOManager.RegisterFileSystem((__COMMON_OBJECT*)&IOManager,
    (__COMMON_OBJECT*)pFatObject))
{
    IOManager.DestroyDevice((__COMMON_OBJECT*)&IOManager,
        pFatObject);
    return FALSE;
}
//Record the driver and device objects.
g_Fat32Driver = lpDriverObject;
g_Fat32Object = pFatObject;
return TRUE;
}
```

驱动程序对象是由 IOManager 创建的，作为参数传递到 DriverEntry 函数中。DriverEntry 函数首先用本地实现的函数，初始化驱动程序对象中的标准函数指针，然后创建了一个设备对象（FAT32 文件系统设备对象），这个设备对象的驱动程序对象，即是刚刚初始化的设备驱动程序对象。并调用 RegisterFileSystem 函数，把这个文件系统设备对象注册到系统中（即注册到 IOManager 的 FsCtrlArray 数组中）。需要注意的是，这个设备对象的设备扩展为空，因为这个文件系统设备对象的作用比较有限，唯一的用途就是检查最新加载的存储设备是不是其管理的设备分区。

DriverEntry 函数执行完毕，FAT32 文件系统就挂接到操作系统中了。这样一旦有存储设备对象被创建，则操作系统就会调用 FAT32 文件系统的特定函数，对新增加的存储设备进行检查。如果检查结果是一个 FAT32 文件卷，则该卷即会被加载到操作系统核心中。

对于 NTFS 文件系统的实现，遵循同样的逻辑。在 Hello China V1.75 的实现中，这两个文件系统管理程序都会被加载到内核中，因此正常情况下，IOManager 的 FsCtrlArray 数组中会存在两个文件系统设备对象——FAT32 文件系统设备对象和 NTFS 文件系统设备对象。需要注意的是，当文件系统驱动程序刚刚被加载完毕，FsArray 数组是没有任何内容的，因为这时候还没有文件卷被加载到系统中。

12.3.3 存储设备驱动程序

1. 存储设备驱动程序的加载

文件系统驱动程序加载完毕，IOManager 会进一步加载存储设备驱动程序。下面以硬盘驱动程序为例说明存储设备的加载过程。与任何设备驱动程序一样，存储设备驱动程序必须输出一个 DriverEntry 函数，这个函数被 IOManager 调用。当然，在调用该函数之前，IOManager 首先应该创建一个设备驱动程序对象，然后以该对象为参数，调用硬盘驱动程序的 DriverEntry 函数。下面是硬盘驱动程序的 DriverEntry 函数源代码：

```
[/kernel/drivers/idehd.cpp]
BOOL IDEHdDriverEntry(__DRIVER_OBJECT* lpDrvObj)
{
    __DEVICE_OBJECT* lpDevObject = NULL;
    __PARTITION_EXTENSION *pPe = NULL;
```



```
//初始化设备驱动程序对象
lpDrvObj->DeviceRead    = DeviceRead;
lpDrvObj->DeviceWrite    = DeviceWrite;
lpDrvObj->DeviceCtrl     = DeviceCtrl;

/*
//连接中断处理程序。最开始的时候是采用中断方式、直接读取硬盘寄存器来操作硬盘的。后来
//修改为使用 BIOS 调用方式读取硬盘数据，因此这段代码注释掉了。
ConnectInterrupt(IDEIntHandler,
    NULL,
    0x2D);
*/
//初始化硬盘控制器
if(!IdeInitialize())
{
    PrintLine("Can not initialize the IDE controller,you may not access HD directly.");
    return FALSE;
}
else
{
    PrintLine("Initialize IDE controller successfully.");
}

//识别硬盘，通过执行 IDE 接口的 IDENTIFY 命令，获取硬盘的相关数据，比如硬盘的大
//小、硬盘扇区的个数等。硬盘的属性数据会存放在 Buff 数组中。
if(!Identify(0,(BYTE*)&Buff[0]))
{
    PrintLine("Can not identify the hard disk device,you can not access IDE directly.");
    return FALSE;
}
else
{
    //计算出磁盘的扇区数，当前只支持 LBA 方式访问磁盘。
    dwLba = ((DWORD)Buff[123] << 24) + ((DWORD)Buff[122] << 16)
        + ((DWORD)Buff[121] << 8) + (DWORD)Buff[120];
    //创建磁盘扩展。磁盘扩展对象用于存放磁盘特定的属性数据。
    pPe = (__PARTITION_EXTENSION*)CREATE_OBJECT(__PARTITION_EXTENSION);
    if(NULL == pPe)
    {
        PrintLine("Can not create RAW partition extension.");
        return FALSE;
    }
    //初始化磁盘扩展
    pPe->BootIndicator    = 0x00;
    pPe->PartitionType    = PARTITION_TYPE_RAW;
    pPe->dwStartSector    = 0;
```



```

pPe->dwSectorNum    = dwLba; //dwLba 即是磁盘的扇区数
pPe->nDiskNum        = 0;
pPe->dwCurrPos       = 0;

//创建磁盘设备对象。
lpDevObject = IOManager.CreateDevice((__COMMON_OBJECT*)&IOManager,
    "\\\\.\\PHYSICALHARDDISK0",
    DEVICE_TYPE_STORAGE | DEVICE_TYPE_HARDDISK,
    512,
    2048,
    2048,
    pPe,
    lpDrvObj);
if(NULL == lpDevObject) //Failed to create device object.
{
    PrintLine("WINHD Driver: Failed to create device object for WINHD.");
    return FALSE;
}
}

//到此为止，磁盘设备对象（整个磁盘，没有考虑分区）成功创建，磁盘算是加载成功了。接
//下来进一步分析磁盘的分区情况，针对每个分区创建一个对应的分区对象。分析分区情况之
//前，首先读取磁盘的 MBR，然后以 MBR 的内容作为参数，调用 InitPartitions 函数。这个函
//数完成了磁盘分区的分析工作。
if(!ReadSector(0,0,1,(BYTE*)&Buff[0]))
{
    PrintLine("Can not read the MBR,all file system in this host may unavailable.");
    return FALSE;
}
InitPartitions(0,(BYTE*)&Buff[0],lpDrvObj);
return TRUE;
}

```

在上述代码中，调用了 `CreateDevice` 函数，创建了物理磁盘设备对象。在创建这个设备对象时，指定了设备的属性为 `DEVICE_TYPE_STORAGE` 和 `DEVICE_TYPE_HARDDISK`。其中第一个属性（`DEVICE_TYPE_STORAGE`）非常重要，这会导致 `IOManager` 调用文件系统设备对象的 `CheckPartition` 函数，对这个存储设备进行检查，以确定其是否为对应的文件系统能够识别的文件卷。

`InitPartitions` 函数对磁盘的分区进行分析。这个函数遍历磁盘 MBR 中的分区表，对每个可识别的分区，创建一个分区设备对象。这个函数比较长，我们只把创建分区设备对象相关的代码看一下，因为这与文件系统关系最密切：

```

[kernel/drivers/idehd.cpp]
static int InitPartitions(int nHdNum,BYTE* pSector0, __DRIVER_OBJECT* lpDrvObject)
{
    static int nPartNum = 0;

```

```

__DEVICE_OBJECT* pDevObject = NULL;
__PARTITION_EXTENSION* pPe = NULL;
BYTE* pStart = NULL;
.....
pStart = pSector0 + 0x1be; //pStart 指向分区表的开始处。
//按照规范，每个物理硬盘上最多有四个主要分区，依次检查四个分区表项。
for(i = 0; i < 4; i++) //Analyze each partition table entry.
{
    .....
    //创建分区扩展对象并初始化。分区扩展是存放分区设备对象特定信息的地方，主要的信息
    //包括分区在硬盘上的起始扇区号、分区长度、分区类型等。
    pPe = (__PARTITION_EXTENSION*)CREATE_OBJECT(__PARTITION_EXTENSION);
    if(NULL == pPe)
    {
        break;
    }
    pPe->dwCurrPos = 0;
    pPe->BootIndicator = *pStart;
    pStart += 4;
    pPe->PartitionType = *pStart;
    pStart += 4;
    pPe->dwStartSector = *(DWORD*)pStart;
    pStart += 4;
    pPe->dwSectorNum = *(DWORD*)pStart;
    pStart += 4; //Pointing to next partition table entry.
    switch(pPe->PartitionType)
    {
        case 0x0B: //FAT32.
        case 0x0C: //Also FAT32.
        case 0x0E: //FAT32 also.
            dwAttributes |= DEVICE_TYPE_FAT32;
            break;
        case 0x07:
            dwAttributes |= DEVICE_TYPE_NTFS;
            break;
        default:
            break;
    }

    //初始化分区设备的设备扩展对象之后，就开始创建分区设备对象。
    .....
    pDevObject = IOManager.CreateDevice(
        (__COMMON_OBJECT*)&IOManager,
        strDevName,
        dwAttributes, //dwAttributes 包含 DEVICE_TYPE_PARTITION 属性
        512,

```

```

        16384,
        16384,
        pPe,
        lpDrvObject);

    .....
}
return nPartNum;
}

```

InitPartitions 函数依次检查四个分区表项，对任何记录了合法分区的表项，该函数会创建一个分区设备对象（设备对象属性中包含 DEVICE_TYPE_PARTITION 属性）。当然，在创建设备对象之前，首先创建设备对象的设备扩展，用于记录分区特定的信息。需要注意的是，这个函数支持扩展分区的处理。

至此，硬盘的设备驱动程序加载过程介绍完了。设备驱动程序加载过程中的最关键动作，就是分析硬盘本身以及硬盘分区情况，并分别创建对应的设备对象。在创建设备对象的时候，需要指定对象的属性（DEVICE_TYPE_STORAGE 或 DEVICE_TYPE_PARTITION）。这非常重要，会直接驱动 IOManager 做进一步的检查。

为了帮助读者进一步理解这个过程，我们举一个简单例子。假设一个硬盘，上面有两个分区，一个分区为 FAT32，另外一个分区类型为 NTFS。这样在硬盘驱动程序加载完成之后，会创建下列三个设备对象：

- (1) 物理硬盘设备对象，名字为“\\.\PHYSICALHARDDISK0”。
- (2) 第一个分区的设备对象，名字为“\\.\PARTITION0”。
- (3) 第二个分区的设备对象，名字为“\\.\PARTITION1”。

这三个设备对象，都指向相同的设备驱动程序对象。此时，我们即可调用 CreateFile 函数，指定上述任何一个名字作为对象，来打开对应的设备进行操作。比如可通过调用 ReadFile，直接读取物理硬盘或分区上的内容。这时候的读取，是不考虑文件系统的存在的，只是把整个硬盘或者分区当作顺序存放的字节数组来处理。

在 Hello China V1.75 的实现中，devlist 程序（字符模式下，输入 devlist 并按 Enter 键）可显示系统中所有的存储设备对象和分区对象。图 12-5 显示了系统中所有的物理设备对象。

```
#devlist
```

DeviceName	Attribute	BlockSize	RdSize/WrSize	
\\.\PARTITION0	18	512	16384/	16384
\\.\FAT32_DEVO	10	0	0/	0
\\.\PHYSICALHARDDISK0	82	512	2048/	2048
\\.\FS_FAT32	40	0	0/	0
KEYBRD	0	0	0/	0

图 12-5 物理设备对象及其属性

其中的“\\.\PHYSICALHARDDISK0”即是硬盘设备对象。这个硬盘只有一个分区，对应于“\\.\PARTITION0”设备对象。

2. 存储设备的读/写

上面说过，通过调用 `ReadFile` 直接读取磁盘或分区设备是可以的。但是这个函数只能按照顺序对磁盘进行读/写。比如第一次调用 `ReadFile`，读取 1KB 的内容。当第二次调用的时候，读取的内容是从 1KB 处开始的。这样显然缺乏灵活性。在文件系统的实现中，需要通过一种灵活的方式，在磁盘或分区上“跳跃”读取或写入，`ReadFile/WriteFile` 是难以胜任的，我们需要另外的途径解决这个需求。

第 10 章讲到，设备驱动程序的 `DeviceCtrl` 函数是一个非常灵活的函数，通过这个函数可以实现常规操作无法胜任的功能。在存储设备驱动程序中，我们也是通过这个函数，实现了磁盘扇区的随机读取和写入功能。我们先通过一段代码，来理解一下如何调用扇区的随机访问功能。下面这个函数，用于从一个分区设备（用分区设备对象标识）上读取一个或几个扇区，这个函数被文件系统实现代码调用（为了解释方便，做了删减）：

```
[/kernel/fs/fatmgr2.cpp]
BOOL ReadDeviceSector(__COMMON_OBJECT* pPartition, //分区设备对象
                     DWORD          dwStartSector, //起始扇区号
                     DWORD          dwSectorNum,   //待读取扇区数量
                     BYTE*          pBuffer) //输出缓冲区指针
{
    BOOL          bResult          = FALSE;
    __DRIVER_OBJECT* pDrvObject    = NULL;
    __DEVICE_OBJECT* pDevObject    = (__DEVICE_OBJECT*)pPartition;
    __DRCB*        pDrcb          = NULL;
    pDrvObject = pDevObject->lpDriverObject;
    //首先创建一个 DRCB 对象，用于传递参数和返回结果。
    pDrcb = (__DRCB*)CREATE_OBJECT(__DRCB);
    if(NULL == pDrcb)
    {
        goto __TERMINAL;
    }
    //Initialize the DRCB object.
    pDrcb->dwStatus          = DRCB_STATUS_INITIALIZED;
    pDrcb->dwRequestMode     = DRCB_REQUEST_MODE_IOCTL;
    pDrcb->dwCtrlCommand     = IOCONTROL_READ_SECTOR; //随机扇区读取命令
    //起始扇区号、读取扇区数量等参数，通过 DRCB 对象的成员变量进行传递，驱动程序也必须按
    //照这个参数传递协议，对 DRCB 的参数进行解释。
    pDrcb->dwInputLen        = sizeof(DWORD);
    pDrcb->lpInputBuffer     = (LPVOID)&dwStartSector; //Input buffer stores the start position
    pointer.
    pDrcb->dwOutputLen        = dwSectorNum * (pDevObject->dwBlockSize);
    pDrcb->lpOutputBuffer     = pBuffer;
    //调用 DeviceCtrl 函数，发起随机读取请求
    if(0 == pDrvObject->DeviceCtrl((__COMMON_OBJECT*)pDrvObject,
                                   (__COMMON_OBJECT*)pDevObject,
                                   pDrcb))
```

```

{
    goto __TERMINAL;
}
bResult = TRUE; //Indicate read successfully.
__TERMINAL:
if(pDrcb) //Should release it.
{
    RELEASE_OBJECT(pDrcb);
}
return bResult;
}

```

上述函数比较简单，首先申请一个 DRCB 对象，然后初始化这个对象。需要注意的是，把 dwCtrlCommand 初始化为 IOCONTROL_READ_SECTOR，这是一个预定义的常量，表示要调用磁盘驱动程序的扇区随机读取功能。

准备好 DRCB 对象之后，然后就以分区设备对象、DRCB 等为参数，调用对应驱动程序对象的 DeviceCtrl 函数。函数成功返回之后，pBuffer 里面就存放了读取的磁盘数据。

DeviceCtrl 是磁盘驱动程序实现的，这个函数分析 DRCB 对象的参数，然后根据参数进一步调用扇区读取函数 __CtrlSectorRead。这是该函数的实现代码：

```

[/kernel/drivers/idehd.cpp]
static DWORD __CtrlSectorRead(__COMMON_OBJECT* lpDrv,
                              __COMMON_OBJECT* lpDev,
                              __DRCB* lpDrcb)
{
    __PARTITION_EXTENSION* pPe = NULL;
    __DEVICE_OBJECT* pDevice = (__DEVICE_OBJECT*)lpDev;
    DWORD dwStartSector = 0;
    DWORD dwSectorNum = 0;
    int nDiskNum = 0;
    DWORD i;
    DWORD dwFlags;

    /*需要修改或读取设备对象的设备扩展信息。由于设备对象是全局数据，可能被多个线程或中断
    程序访问，因此需要使用关键代码段进行保护。*/
    __ENTER_CRITICAL_SECTION(NULL,dwFlags);
    pPe = (__PARTITION_EXTENSION*)pDevice->lpDevExtension; //获取分区扩展对象
    dwStartSector = *(DWORD*)(lpDrcb->lpInputBuffer); //获取起始扇区号
    if(lpDrcb->dwOutputLen % pDevice->dwBlockSize) //读取数据尺寸必须是扇区长度的整数倍
    {
        __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
        return 0L;
    }
    dwSectorNum = lpDrcb->dwOutputLen / pDevice->dwBlockSize; //获取读取扇区数
    //检查是否超过了分区大小。如果起始扇区数加上待读取扇区数大于分区扇区总数，则越界。
    if((dwStartSector + dwSectorNum) > pPe->dwSectorNum)

```

```
{
    __LEAVE_CRITICAL_SECTION(NULL,dwFlags);
    return 0L;
}
dwStartSector += pPe->dwStartSector;
nDiskNum      = pPe->nDiskNum; //磁盘号
__LEAVE_CRITICAL_SECTION(NULL,dwFlags);
//获得硬盘相关的物理参数后, 调用 ReadSector 函数, 读取扇区内容。
for(i = 0; i < dwSectorNum; i++)
{
    if(!ReadSector(nDiskNum,dwStartSector + i,1,((BYTE*)lpDrcb->lpOutputBuffer) + 512*i))
    {
        return FALSE;
    }
}
return TRUE;
}
```

在当前的实现中, ReadSector 函数是通过 BIOS 调用实现的。这个函数首先切换回实模式, 然后调用 int 0x13 中断调用, 完成磁盘读取, 然后返回到保护模式。当然, 也可以使用直接硬盘控制器读/写的方法来实现 ReadSector 函数 (在最初版本中, 就是直接实现的), 但是在很多非 IDE 接口的硬盘上会出问题, 所以使用 BIOS 调用方式更加保险。这个函数的实现非常简单, 其代码位于[kernel/arch/bios.cpp]文件中, 读者可自行阅读。

对分区的随机写入功能, 实现方法与此类似, 在此不作赘述。分区的随机读取和写入功能, 是实现文件系统的基础。在文件系统的实现代码中, 把这两项功能分别封装为 ReadDeviceSector 和 WriteDeviceSector 两个函数, 以方便程序编写。

12.3.4 分区的识别和安装

前面分别介绍了文件系统驱动程序和存储设备驱动程序的加载过程。从介绍的内容来看, 这两者似乎是完全独立的, 没有任何关联。但是要实现文件操作功能, 必须让这两者建立关联, 否则无法建立正确的文件卷。在介绍文件系统驱动程序的时候我们提到, 文件系统驱动程序的加载, 一定要先于存储设备驱动程序。之所以有这样的约束条件, 正是为了在加载存储设备驱动程序的时候, 能够让存储设备有机会“接触”到文件系统, 提供一个“展示”自己的机会, 以让文件系统能够识别自己。用一个不恰当的比喻, 这是一个“寻找组织”的过程。文件系统就好比是组织, 而存储设备则是脱离组织的“成员”, 存储设备需要找到正确的组织, 才能实现自身的有效管理。

这个“寻找组织”的过程, 是由 IOManager 作为中介来“协助”完成的。文件系统不知道存储设备的存在, 同时存储设备也不知道文件系统在哪里。这样 IOManager 就必须为两者搭一座桥, 让它们能够有效沟通。细心的读者会发现, 是 IOManager 的 CreateDevice 函数完成了这个“搭桥”的工作。

文件系统驱动程序首先加载。在加载的时候, 会创建一个文件系统设备对象, 并把这个对象的指针存放在 FsCtrlArray 数组中 (通过调用 RegisterFileSystem 函数)。接下来加载存储

设备驱动程序，存储设备驱动程序会针对每个设备，比如物理硬盘、硬盘分区等，都会调用 `CreateDevice` 函数创建对应的存储设备对象。在创建存储设备对象的时候，需要指定设备的属性，比如可以是 `DEVICE_TYPE_STORAGE`、`DEVICE_TYPE_PARTITION` 等值。这样 `CreateDevice` 就可以根据这些属性，来做不同处理了。对于属性是 `DEVICE_TYPE_PARTITION` 的设备，`CreateDevice` 函数会给所有已加载的文件系统一个机会，让文件系统尝试识别这些存储设备。

具体实现时，文件系统驱动程序需要实现一个函数——`CheckPartition`，这个函数被 `CreateDevice` 调用，用于检查存储设备是不是一个当前文件系统能够识别的分区。如果是，则 `CheckPartition` 需要加载这个分区，否则会继续检查系统中存在的下一个文件系统驱动程序。

与存储设备驱动程序的随机扇区读写功能一样，这个函数也是通过 `DeviceCtrl` 作为入口进行提供的。下面先看一下 FAT32 文件系统的 `CheckPartition` 函数的实现。为了解释方便，删除了部分无关代码：

```
[/kernel/fs/fat32.cpp]
static BOOL CheckPartition(__COMMON_OBJECT* lpThis, //FAT32 文件系统设备对象
                          __COMMON_OBJECT* pPartitionObject) //分区设备对象
{
    __DEVICE_OBJECT*      pPartition = (__DEVICE_OBJECT*)pPartitionObject;
    __FAT32_FS*            pFatObject = NULL;
    BOOL                   bResult    = FALSE;
    __DEVICE_OBJECT*      pFatDevice = NULL;
    CHAR                   DevName[64];
    int                    nIndex;
    static CHAR            nNameIndex = 0 ;

    pFatObject = InitFat32(pPartitionObject);
    if(!pFatObject)
    {
        goto __TERMINAL;
    }
    /* FAT32 分区初始化成功，下面需要创建文件卷对象。在创建之前，首先形成合适的文件卷对象
    名称。FAT32_DEVICE_NAME_BASE 是预定义的字符串“\\FAT32_DEV0”，针对系统中的每个
    FAT32 文件卷对象，其名字前面部分固定，变动的是后面的数字。比如第一个 FAT32 文件卷
    的名字是 FAT32_DEV0，第二个则是 FAT32_DEV1，依此类推。*/
    StrCpy(FAT32_DEVICE_NAME_BASE, DevName);
    nIndex = StrLen(FAT32_DEVICE_NAME_BASE);
    DevName[nIndex - 1] += nNameIndex;
    nNameIndex += 1;
    pFatDevice = IOManager.CreateDevice((__COMMON_OBJECT*)&IOManager,
        DevName,
        DEVICE_TYPE_FAT32,
        DEVICE_BLOCK_SIZE_INVALID, //FAT device object can not be accessed directly.
        DEVICE_BLOCK_SIZE_INVALID,
```

```
    DEVICE_BLOCK_SIZE_INVALID,  
    (LPVOID)pFatObject,  
    ((__DEVICE_OBJECT*)lpThis)->lpDriverObject);  
if(NULL == pFatDevice) //Failed to create device.  
{  
    goto __TERMINAL;  
}  
  
    IOManager.AddFileSystem((__COMMON_OBJECT*)&IOManager,  
        (__COMMON_OBJECT*)pFatDevice,  
        pFatObject->dwAttribute,  
        pFatObject->VolumeLabel);  
bResult = TRUE;  
__TERMINAL:  
    return bResult;  
}
```

这个函数有三个关键点，分别对应上述代码中用黑体标注的三个函数：

(1) **InitFat32** 函数。这个函数试图识别一个分区对象（通过其参数传递）是不是一个合法的 FAT32 文件分区。如果是，则创建一个 FAT32 分区对象，然后返回。否则返回 NULL。这个函数在实现的时候，就是根据 FAT32 文件系统规范，对分区的第一个扇区进行检查。在检查之前，首先调用 **DeviceReadSector**（随机扇区读取，详细内容请参考本章 12.3.5 节）读取分区的第一个扇区，然后对第一个扇区进行分析。具体的分析过程不是很复杂，但是比较冗长，这里就不详细说明了。感兴趣的读者可以直接阅读源代码（位于 `[kernel/fs/fat32.cpp]` 文件中），只要对 FAT32 文件系统规范熟悉了，阅读整个函数的代码会非常简单。返回的 FAT32 分区对象，是一个用于管理 FAT32 分区的数据结构，里面包含了分区的卷标、根目录的起始 cluster 编号、每个 cluster 的扇区数量、文件分配表的个数和起始 cluster 号等。具体的定义，位于文件 `[kernel/fs/fat32.h]` 中。

(2) **CreateDevice** 函数。若 **InitFat32** 成功地识别了一个 FAT32 分区，则 **CheckPartition** 需要创建一个文件卷对象，并把这个卷对象注册到系统中。需要注意的是，这个文件卷对象的设备扩展就是 **InitFat32** 返回的 FAT32 分区对象。

(3) **AddFileSystem** 函数。成功创建 FAT32 文件卷对象之后，需要把文件卷对象注册到系统中，这样这个文件卷即可对系统用户可见。**AddFileSystem** 函数即是把新创建的文件卷对象加入到 **IOManager** 的 **FsArray** 数组。

上述几个步骤完成之后，被 FAT32 文件系统识别的 FAT32 文件卷，即可呈现在系统中了。

接下来再看 **CheckPartition** 函数是怎么被调用的。前面说过，这个函数是由 **CreateDevice** 函数调用的。**CreateDevice** 针对每个属性是 **DEVICE_TYPE_PARTITION** 的设备，都会依次调用系统中已经注册的文件系统设备对象的 **CheckPartition** 函数。下面是 **CreateDevice** 函数的相关代码（**CreateDevice** 函数比较长，我们只摘录与 **CheckPartition** 调用有关的代码进行说明）：


```

[/kernel/kernel/iomgr.cpp]
static __DEVICE_OBJECT* CreateDevice(__COMMON_OBJECT* lpThis,
                                     LPSTR          lpszDevName,
                                     DWORD           dwAttribute,
                                     DWORD           dwBlockSize,
                                     DWORD           dwMaxReadSize,
                                     DWORD           dwMaxWriteSize,
                                     LPVOID          lpDevExtension,
                                     __DRIVER_OBJECT* lpDrvObject)
{
    __DEVICE_OBJECT* lpDevObject      = NULL;
    __DEVICE_OBJECT* lpFsDriver       = NULL;
    __DRCB*          lpDrcb           = NULL;
    __IO_MANAGER*    lpIoManager      = (__IO_MANAGER*)lpThis;
    DWORD            dwFlags           = 0L;
    DWORD            dwCtrlRet         = 0;
    int              i;
    .....
    if(!(dwAttribute & DEVICE_TYPE_PARTITION)) //若不是分区设备，则略过下面代码。
    {
        goto __CONTINUE;
    }
    //如果是一个分区设备，则依次调用系统中的文件系统设备对象的 CheckPartition 函数。
    for(i = 0; i < FS_CTRL_NUM; i++)
    {
        if(lpIoManager->FsCtrlArray[i]) //注册了文件系统设备对象。
        {
            lpFsDriver = (__DEVICE_OBJECT*)lpIoManager->FsCtrlArray[i];
            if(lpFsDriver->dwSignature != DEVICE_OBJECT_SIGNATURE) //检查签名
            {
                break;
            }
            //创建 DRCB 对象并初始化，用于发起 CheckPartition 请求。
            lpDrcb = (__DRCB*)ObjectManager.CreateObject(&ObjectManager,
                                                         NULL,
                                                         OBJECT_TYPE_DRCB);
            if(NULL == lpDrcb) //创建 DRCB 对象失败，很少发生。
                goto __CONTINUE;
            if(!lpDrcb->Initialize((__COMMON_OBJECT*)lpDrcb)) //初始化 DRCB 对象。
            {
                ObjectManager.DestroyObject(&ObjectManager,
                                             (__COMMON_OBJECT*)lpDrcb);
                goto __CONTINUE;
            }
            //初始化 DRCB 对象的各个参数，注意 dwCtrlCommand 参数。
            lpDrcb->dwStatus      = DRCB_STATUS_INITIALIZED;

```

```

lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_IOCTL;
lpDrcb->dwCtrlCommand = IOCONTROL_FS_CHECKPARTITION;
lpDrcb->dwInputLen = sizeof(__DEVICE_OBJECT*);
lpDrcb->lpInputBuffer = (LPVOID)lpDevObject;
//发起 CheckPartition 调用。
dwCtrlRet = lpFsDriver->lpDriverObject->DeviceCtrl(
    (__COMMON_OBJECT*)lpFsDriver->lpDriverObject,
    (__COMMON_OBJECT*)lpFsDriver,
    lpDrcb);
//调用完毕，释放 DRCB 对象。
ObjectManager.DestroyObject(&ObjectManager,
    (__COMMON_OBJECT*)lpDrcb);
if(dwCtrlRet) //判断分区是否被正确识别。
{
    break;
}
}
}

__CONTINUE:
.....
}

```

CreateDevice 函数针对设备的类型进行特殊处理。如果发现设备类型是一个分区对象，则会依次调用系统中已安装的文件系统驱动程序的 CheckPartition 函数，对分区进行识别。如果能够正确识别（即 DeviceCtrl 函数，实际上是 CheckPartition 函数，返回非 0 值），则结束循环，无需进一步检查。最后再强调一点，CheckPartition 函数的说法可能不太合适，更严格说，应该是 check partition 命令。CheckPartition 函数是 FAT32 文件系统（也是其他文件系统）驱动程序实现的一个分区检查函数，这个函数完全可以用其他的名字，只要实现分区检查和安装功能即可。这个函数被 DeviceCtrl 进一步封装，真正暴露给 CreateDevice 函数的，还是标准的驱动程序接口函数 DeviceCtrl。

好了，对文件系统、存储设备等之间的相互协调，以及分区的识别和加载过程，相信读者已经搞明白了。在此进一步总结一下：

（1）文件系统驱动程序首先被加载。

（2）存储设备驱动程序继而被加载。在加载存储设备驱动程序的时候，会调用 CreateDevice 函数创建分区设备对象。

（3）CreateDevice 根据设备属性（DEVICE_TYPE_PARTITION），调用系统中已经注册的文件系统的 CheckPartition 函数，对分区进行检查。

（4）文件系统的 CheckPartition 函数如果能够识别分区，则创建文件卷对象，并调用 AddFileSystem 注册到系统中。至此文件卷可见。

这不是一个复杂的过程，但具备比较好的扩展性，比如，这个框架可以很好地适应移动存储设备的即插即用功能。假设有一个 USB 接口的存储设备连接到了计算机，USB 总线驱动程序就会感知这个事件，通知操作系统加载对应的驱动程序。在 USB 存储设备驱动程序

的加载过程中, 调用 `CreateDevice` 函数, 创建 USB 存储设备对象。这时候一定要指定设备对象的属性为分区设备。这样系统中所有已安装的文件系统, 比如 FAT32、NTFS 等的 `CheckPartition` 函数都将有机会被调用, 试图识别 USB 设备的文件格式。一旦识别, 就会被自动安装到系统中, 从而对用户可见。

12.3.5 文件的打开操作

在 Hello China 当前版本的实现中, 把文件跟普通的设备同等对待。即每打开一个文件, 在操作系统内部都会增加一个设备对象 (文件设备对象); 对于文件的读写等操作, 直接调用设备对应的驱动程序提供的 `DeviceRead` 和 `DeviceWrite` 等服务函数。

`CreateFile` 是操作系统提供给用户的通用设备打开接口函数, 这个函数可用于打开普通的设备进行操作, 也可用于打开文件系统中的任意文件。该函数原型如下:

<code>_COMMON_OBJECT*</code>	<code>(*CreateFile)(_COMMON_OBJECT*</code>	<code>lpThis,</code>
	<code>LPSTR</code>	<code>lpzFileName,</code>
	<code>DWORD</code>	<code>dwAccessMode,</code>
	<code>DWORD</code>	<code>dwOperationMode,</code>
	<code>LPVOID</code>	<code>lpReserved);</code>

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` 两个参数用于控制打开的文件, 不适用于目标对象是物理设备的情形; 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 “文件卷标识符+文件路径+文件名” 的格式, 比如 “C:\HCN\DATA1.BIN”。

用户通过调用 `CreateFile` 函数来打开一个文件, 该函数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。

(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。这个过程将遍历设备对象链表, 根据名字匹配设备对象。详细过程请参考第 10 章。

(3) 如果判断结果是文件, 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, 并增加引用计数。如果不允许重复打开, 则返回失败标志。

(4) 如果遍历完设备对象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卷标识符 (比如, C:,D:,E:等)。

(5) 根据文件卷标识符遍历 `IOManager` 的 `FsArray` 数组, 这个数组存放了系统中所有正确识别的文件卷。一旦找到一个数组元素, 即可从数组元素中进一步得到文件卷设备对象。

(6) 如果找不到对应的文件卷设备对象, 则说明给出的文件名有问题。比如当前系统中



只有 C:、D: 两个文件卷，但是用户却尝试打开 “E:\DATA.TXT” 文件，则会出现这种情况。显然，这只能以失败返回。

(7) 如果可以找到对应的文件卷设备对象，则 IOManager 调用文件卷设备对象的 DeviceOpen 函数（以文件名或 DRCB 为参数），尝试打开该文件。

(8) 如果 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）。

(9) CreateFile 根据 DeviceOpen 函数的返回结果，给初始调用返回适当的数值。

文件驱动程序的 DeviceOpen 函数，会从分区的根目录开始，根据文件名进行逐级搜索。一旦定位到文件所在目录，即可读取目录文件信息，获得文件的相关信息，如文件名、文件起始 cluster 号、文件大小等。然后创建一个文件扩展对象（__FAT32_FILE），把文件相关的信息存放在这个对象中。接下来需要调用 CreateDevice 函数，创建一个文件设备对象，并把文件扩展对象作为文件设备对象的设备扩展。

下面是 FAT32 文件系统的实现中，文件扩展对象（__FAT32_FILE）的定义：

```
[/kernel/fs/fat32.h]
typedef struct FAT32_FILE{
    BYTE      FileName[13];           //以 0 结束的文件名
    BYTE      FileExtension[4];       //以 0 结束的文件扩展名
    BYTE      Attributes;             //文件属性，比如只读、存档、系统等；
    DWORD     dwStartClusNum;         //文件开始簇号
    DWORD     dwCurrClusNum;         //文件当前簇号
    DWORD     dwCurrPos;              //文件指针当前位置（相对文件开始处的偏移量）
    DWORD     dwClusOffset;           //文件指针在当前簇内的偏移
    DWORD     dwFileSize;             //文件尺寸
    DWORD     dwOpenMode;             //打开模式，比如只读等
    DWORD     dwShareMode;           //共享模式
    BOOL      bInRoot;               //是否位于根目录中
    DWORD     dwParentClus;           //父目录的起始 cluster 号
    DWORD     dwParentOffset;         //文件目录项在父目录中的偏移
    FAT32_FS* pFileSystem;            //指向文件卷设备对象扩展
    _COMMON_OBJECT* pFileCache;       //文件缓存对象，当前未用
    _COMMON_OBJECT* pPartition;       //文件所属分区对象
    FAT32_FILE* pNext;
    FAT32_FILE* pPrev;
} __FAT32_FILE;
```

显然，这是与特定文件相关的信息。最后两个指针 pNext 和 pPrev，把文件扩展对象链接成一个双向链表。需要注意的是，这个双向链表与文件设备对象所在的设备链表不同。pPartition 对象指针指向了文件所在的分区。而 pFileSystem 指针指向该文件所在的文件系统设备扩展对象。文件系统设备扩展对象记录了文件卷相关的特定信息，这是文件卷设备对象的设备扩展。而系统中的所有文件卷对象都统一存放在 IOManager 的 FsArray 数组中。图 12-6 示意了这些对象之间的逻辑关系。

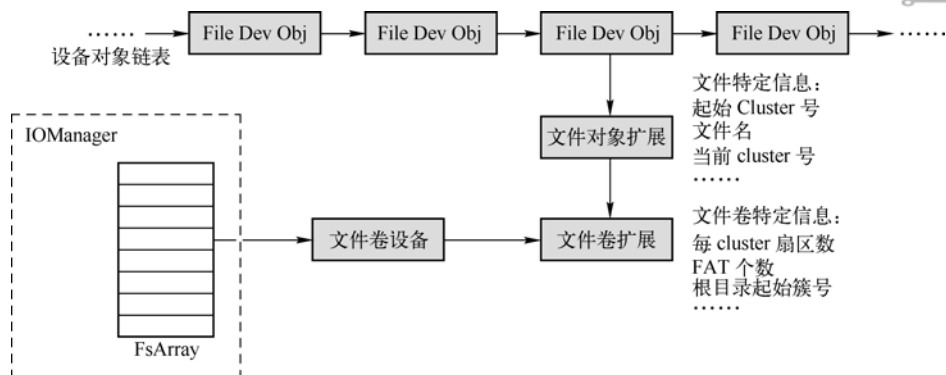


图 12-6 文件系统核心对象之间的关系

在我们的实现中，每打开一个文件，DeviceOpen 函数便会创建一个文件设备对象。这个对象与普通的设备对象一样，被连接到系统的设备对象链表中（IOManager 的 lpDevice Object 成员指向这个链表）。针对系统中每个成功识别的分区，都会有一个与之对应的文件卷设备对象。需要注意的是，文件卷设备对象也是一个设备对象，也会被链入设备链表中，这在图中没有体现。文件卷设备对象是与一个特定分区连接在一起的，其设备扩展——文件卷扩展，则包含了分区（文件卷）特定的信息。下面是文件卷扩展对象的定义：

```

[/kernel/fs/fat32.h]
typedef struct FAT32_FS{
    __COMMON_OBJECT*    pPartition;           //文件卷所属分区
    DWORD                dwAttribute;         //文件系统属性
    BYTE                SectorPerClus;        //每 cluster 扇区数
    BYTE                VolumeLabel[13];      //卷标
    BYTE                FatNum;               //FAT 表数量
    BYTE                Reserved;            //保留
    WORD                wReservedSector;      //保留
    WORD                wFatInfoSector;       //FAT 信息扇区号
    DWORD               dwBytePerSector;      //每扇区字节数，一般为 512
    DWORD               dwClusterSize;        //每 cluster 字节数
    DWORD               dwDataSectorStart;    //数据起始扇区号
    DWORD               dwRootDirClusStart;   //根目录起始扇区号
    DWORD               dwFatBeginSector;     //FAT 起始扇区号
    DWORD               dwFatSectorNum;       //每个 FAT 所占扇区数量
    DWORD               FatCache[FAT_CACHE_LENGTH]; //FAT 缓存
    __FAT32_FILE*       pFileList;          //文件设备扩展链表，即指向__FAT32_FS 对象链表
}__FAT32_FS;
  
```

这个对象记录了文件卷相关的信息。pPartition 指向该卷所在的分区对象，这个对象与文件对象扩展中指向的分区对象是同一个对象。在读取文件卷中的数据的时候，最终都是转化为以 pPartition 对象为参数调用 DeviceReadSector 函数的。DeviceReadSector 函数进一步调用了 pPartition 对象驱动程序提供的 DeviceCtrl 函数，这个函数提供了扇区随机读写功能。

需要注意的是，不论是文件设备对象，还是文件卷设备对象，都指向相同的驱动程序对象。但是在调用 DeviceOpen 函数的时候，是以文件卷设备对象及其指向的驱动程序对象为参数的。因为这时候具体的文件还未打开。而在调用 DeviceRead/DeviceWrite 等函数时，则是以具体的文件设备对象和其指向的设备驱动程序对象为参数的，这时文件已经打开。

12.3.6 文件的读取操作

文件打开之后，即可调用 ReadFile、WriteFile 等函数，对文件进行读写等操作了。下面以文件读取为例，详细介绍这个过程。写文件的过程与此类似，不再赘述。

首先看 ReadFile 函数的源代码，这个函数比 CreateFile 简单得多：

```
[/kernel/kernel/iomgr2.cpp]
BOOL ReadFile(__COMMON_OBJECT* lpThis,           //IOManager 对象指针
              __COMMON_OBJECT* lpFileObject,     //待读取文件对象（句柄）
              DWORD dwByteSize,                  //待读取字节数
              LPVOID lpBuffer,                   //缓冲区
              DWORD* lpReadSize)                 //实际读取字节数
{
    BOOL bResult = FALSE;
    __DEVICE_OBJECT* lpFile = (__DEVICE_OBJECT*)lpFileObject;
    __DRIVER_OBJECT* lpDriver = NULL;
    __DRCB* lpDrcb = NULL;
    LPVOID lpTmpBuff = NULL;
    DWORD dwToRead = 0;
    DWORD dwRead = 0;
    DWORD dwTotalRead = 0;
    DWORD dwPartRead = 0;
    BYTE* pPartBuff = NULL;

    //创建并初始化一个 DRCB 对象，以跟踪这个读取事务
    lpDrcb = (__DRCB*)ObjectManager.CreateObject(&ObjectManager,
        NULL,
        OBJECT_TYPE_DRCB);
    if(NULL == lpDrcb) //Failed to create DRCB object.
    {
        goto __TERMINAL;
    }
    if(!lpDrcb->Initialize((__COMMON_OBJECT*)lpDrcb)) //Failed to initialize.
    {
        goto __TERMINAL;
    }
    //接下来获得文件设备对象的设备驱动程序对象，然后调用其 DeviceRead 函数。
    lpDriver = lpFile->lpDriverObject;
    lpTmpBuff = lpBuffer; //在读取过程中，需移动缓冲区指针，因此用 lpTmpBuff 替代原始值
    do{
```

```

lpDrcb->dwRequestMode = DRCB_REQUEST_MODE_READ; //调用 DeviceRead 函数
lpDrcb->dwStatus      = DRCB_STATUS_INITIALIZED;
//如果读取的字节数大于文件允许的最大读取字节数，则分开多次读取
if(dwByteSize >= lpFile->dwMaxReadSize)
{
    dwToRead      = lpFile->dwMaxReadSize; //每次读取文件允许的最大读取尺寸
    lpDrcb->lpOutputBuffer = lpTmpBuff;      //设置输出缓冲区
    lpTmpBuff      = (BYTE*)lpTmpBuff + lpFile->dwMaxReadSize; //调整缓冲区
    dwByteSize     = dwByteSize - dwToRead;   //计算剩余的待读取尺寸
    lpDrcb->dwOutputLen = dwToRead;           //设置待读取尺寸到 DRCB 对象
}
else //如果读取的尺寸小于设备要求的最大读取尺寸，则读取一次即可
{
    if(0 == dwByteSize) //读取完毕
    {
        break;
    }
    dwToRead      = dwByteSize;
    dwPartRead     = dwToRead;
    //保留原始读取值，因为接下来 dwToRead 可能要被修改。
    //每次读取的数据尺寸，必须是设备块的整数值。如果不是，需要向上舍入到块边界。
    //需要注意的是，文件允许的最大读取尺寸，一定是文件块的整数值。
    if(dwToRead % lpFile->dwBlockSize)
    {
        dwToRead += (lpFile->dwBlockSize - (dwToRead % lpFile->dwBlockSize));
    }
    //上面的向上舍入操作后，可能会使得实际读取尺寸变大。而原始缓冲区大小是按照实
    //际读取尺寸给出的，因此这里需要另外申请一块缓冲区，以装载实际读取内容。返回
    //后，只复制所需大小的尺寸到原始缓冲区即可。比如原始读取尺寸是 2KB，而文件的块
    //尺寸是 4KB。这样就需要把 dwToRead 舍入为 4KB。而原始缓冲区是按照 2KB 设置的，不
    //能容纳 4KB 数据。因此需要重新申请一块 4KB 的缓冲区。读取完成之后，只复制开头的
    //2KB 数据给用户即可。
    pPartBuff = (BYTE*)KMemAlloc(dwToRead, KMEM_SIZE_TYPE_ANY);
    if(NULL == pPartBuff) //Can not allocate buffer, give up.
    {
        break;
    }
    //设置最新申请的缓冲区为输出缓冲区
    lpDrcb->lpOutputBuffer = pPartBuff;
    lpDrcb->dwOutputLen = dwToRead; //要求读取的尺寸
    dwByteSize = 0;                //设置 dwByteSize 为 0，指示所有读取已完成
}
//调用 DeviceRead 函数，向设备驱动程序发起读取请求
dwRead = lpDriver->DeviceRead(
    (__COMMON_OBJECT*)lpDriver,
    (__COMMON_OBJECT*)lpFile,

```

```

        lpDrcb);
        dwTotalRead += dwRead; //增加实际读取量, DeviceRead 函数返回实际读取的字节数
//实际读取字节数小于请求的字节数, 可能是到达文件末端导致
        if(dwRead < dwToRead)
        {
            dwPartRead = dwRead;
            break;
        }
        if(0 == dwByteSize)    //读取完毕
        {
            break;
        }
    }while(dwToRead >= lpFile->dwMaxReadSize);
//如果该缓冲区不为空, 说明有残余读取 (即上述向上舍入到读取块边界的读取操作) 存在, 需
//要把残余的读取量也复制到用户缓冲区。
    if(pPartBuff)
    {
        memcpy(lpTmpBuff,pPartBuff,dwPartRead); //Append the partition data to buffer.
        dwTotalRead -= dwRead;
        dwTotalRead += dwPartRead;
    }
//如果用户给定了实际读取尺寸的指针, 则返回实际读取尺寸。ReadFile 只是返回读取结果
// (TRUE 或 FALSE), 而实际读取的尺寸, 是通过 lpReadSize 返回的。这个指针如果被设
//置为 NULL, 则实际读取尺寸不能返回, 但 ReadFile 函数却可以成功执行。这与 Windows
//的 ReadFile 不一样。如果 Windows 的 ReadFile 函数不设置实际读取尺寸, 则会执行失败。
//作者编写过一些 Windows 程序, 经常因为这个原因导致程序出问题, 于是修改成这样的实现
//方式。
    if(NULL != lpReadSize)
    {
        *lpReadSize = dwTotalRead;
    }
//根据 DRCB 的处理结果状态, 返回读取结果。需要注意的是, 即使成功读取了部分内容, 但是
//由于某些原因导致 DRCB 对象处理状态为 FAIL, 则 ReadFile 函数仍然返回 FALSE。
    if(DRCB_STATUS_FAIL == lpDrcb->dwStatus)
    {
        bResult = FALSE;
    }
    else
    {
        bResult = TRUE;
    }
__TERMINAL:
    if(lpDrcb)
    {
        ObjectManager.DestroyObject(&ObjectManager,

```



```
        (__COMMON_OBJECT*)lpDrCb);  
    }  
    if(pPartBuff)  
    {  
        KMemFree(pPartBuff,KMEM_SIZE_TYPE_ANY,0);  
    }  
    return bResult;  
}
```

ReadFile 的实现思路很简单,就是进一步调用文件系统驱动程序的 DeviceRead 函数。但复杂的地方在于,ReadFile 需要做一下适配。设备驱动程序在实现的时候,可以设定一个最大读取尺寸和一个块尺寸。最大读取尺寸是一次 DeviceRead 调用能够读取的最大尺寸,而块尺寸则要求每次读取的时候,读取尺寸值必须是块尺寸的整数倍。这样可方便驱动程序的实现,比如针对文件来说,可以设置最大读取尺寸为 16KB,块尺寸为 4KB。这样在实现的时候,只需要有一个 16KB 的全局缓冲区和一个 4KB 的块缓冲区即可。无需考虑用户读取尺寸变化的情形。

而 ReadFile 则需要考虑用户读取尺寸变化的情形,把任何尺寸的读取请求,转换为按照最大读取尺寸和块尺寸要求的读取操作。针对大于最大读取尺寸的情形,ReadFile 需要分开多次进行读取。而如果读取的尺寸不是块尺寸的整数倍,ReadFile 也要以块尺寸为单位调用 DeviceRead 函数,调用完成后,只复制相关的内容到用户缓冲区。

接下来就是文件驱动程序的工作了。文件驱动程序需要实现 DeviceRead 函数,完成对实际文件的读取。在实现的时候,必须遵循设备驱动程序实现规范。实际上,DeviceRead 函数的大部分工作,是按照文件系统的规范,对文件进行操作。比如对 FAT32 来说,DeviceRead 函数需要通过读取文件分配表,获得待读取内容所在的 cluster 号,然后调用 DeviceReadSector 函数,发起真正的读取操作。具体的实现过程比较复杂,在此就不详细陈述代码了。在读者对 FAT32 文件系统规范熟悉的情况下,代码很容易阅读。

12.4 文件系统 API 的使用举例

接下来给出一个调用文件系统 API 函数对文件进行读写的例子。这个例子的源代码,是对 fs 程序的 type 子命令(显示文件内容)的实现进行修改后得到的。对文件的操作,一般遵循下列流程:

- (1) 调用 CreateFile, 打开一个文件。
- (2) 调用 ReadFile 或 WriteFile, 对文件进行读写操作。读写的时候,需要判断函数执行结果。
- (3) 操作完毕,调用 CloseFile 函数,关闭打开的文件。

下面看一段代码,这段代码用于在屏幕上显示一个文件的内容,其参数指明了要显示的文件名和详细路径:

```
DWORD ShowFileCoontent(CHAR* FullName)  
{
```



```
HANDLE    hFile = NULL;
CHAR      Buffer[128];
DWORD     dwReadSize    = 0;
DWORD     dwTotalRead   = 0;
DWORD     i;
WORD      ch = 0x0700;
//调用 CreateFile 打开文件
hFile = IOManager.CreateFile((__COMMON_OBJECT*)&IOManager,
    FullName, //文件名
    FILE_ACCESS_READ, //打开方式, 只读
    0,
    NULL);
if(NULL == hFile)
{
    PrintLine(" Please specify a valid and present file name.");
    goto __TERMINAL;
}
//读取文件内容并显示
ChangeLine();           //屏幕换行
GotoHome();             //屏幕光标回归到最左边
do{
    if(!IOManager.ReadFile((__COMMON_OBJECT*)&IOManager,
        hFile,
        128,                //每次读取 128 字节
        Buffer,              //读取到 Buffer 缓冲区中
        &dwReadSize))       //用于返回实际读取尺寸
    {
        PrintLine(" Can not read the target file."); //读取失败
        goto __TERMINAL;
    }
    for(i = 0; i < dwReadSize; i++) //把读取的内容, 一个字符一个字符地显示到屏幕上
    {
        if('\r' == Buffer[i]) //如果是回车, 则光标回到屏幕最左边
        {
            GotoHome();
            continue;
        }
        if('\n' == Buffer[i]) //如果是换行, 则屏幕光标换到下一行
        {
            ChangeLine();
            continue;
        }
    }
    //输出字符, 需要注意的是, ch 的低字节是实际字符, 高字节是显示属性, 比如显示颜
    //色、亮度等。
    ch += Buffer[i];
    PrintCh(ch);
}
```

```
        ch = 0x0700;
    }
    dwTotalRead += dwReadSize;
} while(dwReadSize == 128);
//最后, 重新起一行, 显示出总共读取的字节数。
ChangeLine();
GotoHome();
sprintf(Buffer, "%d byte(s) read.", dwTotalRead);
PrintLine(Buffer);
__TERMINAL:
if(NULL != hFile)
{
    IOManager.CloseFile((__COMMON_OBJECT*)&IOManager,
        hFile); //关闭文件
}
return FS_CMD_SUCCESS;
}
```

需要注意的是, 上述代码直接调用了 `IOManager` 提供的接口函数。还有一种实现方式是, 调用 `CreateFile`、`ReadFile` 等函数的系统调用版, 这样就无需指定第一个参数 (`&IOManager`) 了。但是系统调用版需要包含 `KAPI.H` 头文件, 并按照独立应用程序的方式进行编译连接和加载。详细内容可参考本书第 13 章。

12.5 文件系统实现总结

本章以 FAT32 文件系统的原理和实现为例, 对操作系统文件系统的实现进行了详细的讲解。虽然大部分代码和实现原理都是来自 `Hello China` 的文件系统实现, 但这些原理和思想, 对任何文件系统都是相同的。读者如果理解了本章内容, 那么对任何操作系统的文件系统的理解, 将不再困难。

文件系统是操作系统的关键功能, 尤其是对通用操作系统和输入/输出频繁的应用场合, 文件系统是决定整个系统性能的关键。本章重点关注了文件系统的实现框架和流程, 对于文件系统性能方面的优化没有做过多描述。读者可在理解本章内容基础上, 对现有代码做进一步优化, 使其性能能够得到更进一步的提升。当前版本的代码已预留了性能提升的扩展余地, 比如针对文件设备对象扩展 (`__FAT32_FILE`) 的 `pFileCache` 指针, 针对文件卷设备对象扩展 (`__FAT32_FS`) 的 `FAT` 表缓冲等。通过在这些指针上挂接缓冲对象和对应的操作方法, 即可对整体性能进行优化。

文件系统同时也是一个非常复杂的课题。虽然我们做了最大可能的简化, 但在几十页的篇幅内, 把文件系统说清楚也是非常困难的, 需要读者自行阅读代码做进一步的理解。阅读代码所用到的一些关键概念和机理, 本章都介绍了。

最后贴一张 MS-DOS 早期版本的 `dir` 命令输出图片 (如图 12-7 所示), 作为本章的结束。相信任何一个计算机专业的读者, 对这个命令都不会陌生。对于像作者这样 2000 年之前上大学的人来说, `dir` 命令基本等于计算机本身。看到下面这张图, 是否会勾起您对大学

生活的回忆？

```
IMG0000H.JPG          29,980  09-23-08  21:01
 ¼00~1.JPG            16,483  09-23-08  21:06
CMBC                   <DIR>      09-24-08  11:30
WINDBM~1.EXE          249,856  09-24-08  15:21
JAVATO~1              <DIR>      09-25-08  13:30
APPLIC~1.XML           8,475  09-25-08  14:47
DB2JCC.JAR            2,833,724  09-17-07  21:23
SSPLAT~1              <DIR>      09-26-08  17:29
7512V14               <DIR>      09-26-08  18:04
METADA~1              <DIR>      09-26-08  19:14
JRE15~1.0_1           <DIR>      09-26-08  19:18
TCNCLI~1              <DIR>      09-26-08  19:50
DJA VAT~1             <DIR>      09-26-08  19:54
      19 file(s)         14,409,637 bytes
      20 dir(s)          637,599,744 bytes free

A:\>cd 7512v14_
```

图 12-7 MS-DOS 早期版本的 dir 命令输出

第 13 章 应用程序开发方法

13.1 概述

一个完整的操作系统，必须能够提供一套完整的工具和方法支持应用程序的开发。一般情况下，操作系统提供一组系统调用接口（API 接口），程序员可以通过这一组 API 接口访问操作系统提供的服务。同时提供一个开发环境，程序员在这个开发环境中完成应用程序的开发和编译工作。

作为一个面向智能设备的嵌入式操作系统，Hello China 已经发展到 1.75 版。该版本提供了一组相对完整的 API 函数，供应用程序开发使用。这一组 API 函数包括线程/进程管理、内存管理、设备访问、文件管理、图形界面等方方面面，API 函数数量超过了 100 个。这组 API 通过系统调用的方式，实现了操作系统内核和应用程序代码的完全分离。即应用程序和操作系统内核都是独立的文件，完全分离，无需连接在一起，这大大增强了系统的可伸缩性。

同时 Hello China V1.75 版本提供了一组辅助开发工具，可辅助程序员快速、方便地完成基于 Hello China 操作系统的应用开发。但是当前版本的 Hello China 并未提供集成开发环境，需要程序员借助已有的集成开发环境来完成代码的编译和链接工作，比如可以使用 Microsoft 的 Visual Studio 系列开发环境。

本文首先介绍了 Hello China 应用程序的体系结构和加载原理，然后以 Visual Studio 2008 为例，详细介绍了应用程序的开发步骤和注意事项。

13.2 HCX 文件的结构和加载过程

HCX 是 Hello China 可执行文件（Hello China eXecutable）的缩写，是 Hello China 定义的一个文件结构。所有 Hello China 可执行程序，都必须以该文件定义的格式进行存储，否则不能被 Hello China 加载。

13.2.1 HCX 文件的格式

HCX 文件包含了可执行代码、初始化的全局数据、应用程序版本信息、应用程序图标等信息。HCX 文件会被 GUI shell 读取，并以图标的方式显示在应用程序列表内。一旦用户点击应用程序图标，则应用程序就会被加载并执行。

当前 HCX 文件格式版本是 V1.0，主要为 Hello China V1.75 版本定制。后续版本的 HCX 文件格式可能会有变动，但是会保持向前兼容性。下面是 HCX V1.0 文件的格式描述。

首先是一个 HCX 文件格式头，这个文件头定义如下：

```
[gui/include/launch.h]
struct _HCX_HEADER{
    DWORD    dwHcxSignature;        //HCX 文件签名, x86 CPU 上为 0xE9909090
    DWORD    dwEntryOffset;         //入口函数地址相对于文件头的偏移
    DWORD    dwBmpOffset;           //程序图标相对于文件头的偏移
    DWORD    dwBmpWidth;            //程序图标的宽度, 以像素表示
    DWORD    dwBmpHeight;           //程序图标的高度, 以像素表示
    DWORD    dwColorBits;           //图标颜色深度
    CHAR     AppName[16];           //应用程序的可视化名字
    CHAR     MajorVersion;          //应用程序主版本号
    CHAR     MinorVersion;          //应用程序次版本号
    CHAR     OsMajorVersion;        //可运行该程序的操作系统的主版本号
    CHAR     OsMinorVersion;        //可运行该程序的操作系统的次版本号
};
```

其中 HCX 签名用于标识一个合法的 HCX 文件。操作系统在试图执行一个 HCX 文件时, 首先把该文件读入内存, 然后就检查该文件的签名是不是 0xE9909090。如果是, 则认为是一个合法的 HCX 文件, 并试图执行。否则认为不是一个合法的 HCX 文件, 放弃进一步的执行。

在以 x86 CPU 为处理器的平台上, HCX 签名实际上是三个 `nop` 指令加一个跳转指令(跳转指令的目标地址, 就是 `dwEntryOffset`)。这样安排的目的, 是操作系统完成 HCX 文件的合法性检查之后, 可直接跳转到文件的开头去执行, 无需进一步检索入口点。

另外一个需要解释的是 `AppName`, 这是应用程序的可视化名字, 即显示在 GUI shell 中的应用程序列表中的名字。这个名字与 HCX 文件名可以不一样。建议取一个能够直接显示应用程序本身功能的名字。

应用程序主版本号和次版本号构成了一个管理应用程序版本的机制。在升级应用程序的时候, Hello China 会判断新应用程序的版本是否高于当前应用程序。如果是, 则允许升级, 否则会给出提示。

最后的目标操作系统版本号, 指明了能够运行当前 HCX 文件的最低操作系统版本。如果当前操作系统的版本低于该版本号, 则会拒绝 HCX 文件的执行。因为 HCX 文件有可能应用了高版本操作系统的一些特性。

紧跟 HCX 头, 就是可执行二进制代码和初始化的全局数据。这是 HCX 文件的主要内容, 所有可执行指令和相关数据, 都存放在这个部分中。这部分内容是由编译器生成的。

最后是一个应用程序图标, 以 RGB 方式存放, 当前只支持 128×128 像素的 BMP 文件, 且颜色深度必须是 24 位。GUI shell 会在应用程序列表中显示这个图标(以及应用程序的可视化名字)。

13.2.2 HCX 文件的生成方式

HCX 文件是由一个叫做 `hcxbuild` 的程序生成的。首先程序员使用某种编译工具(比如 VS 2008), 编译生成一个二进制可执行文件(比如 DLL 文件), 然后准备一个 BMP 格式的图标文件, 把这两个文件作为 `hcxbuild` 的输入。

Hcxbuild 程序会根据上述两个文件（DLL 和 BMP 文件）的信息以及用户输入信息（比如可视化名字等），自动生成一个 HCX 头，然后提取 DLL 文件中的可执行代码和全局数据，再分析 BMP 文件，提取图片数据，最后把上述信息整合起来，形成一个 hcx 文件。这个 hcx 文件即是 Hello China 操作系统可加载运行的应用程序。

图 13-1 说明了上述过程。

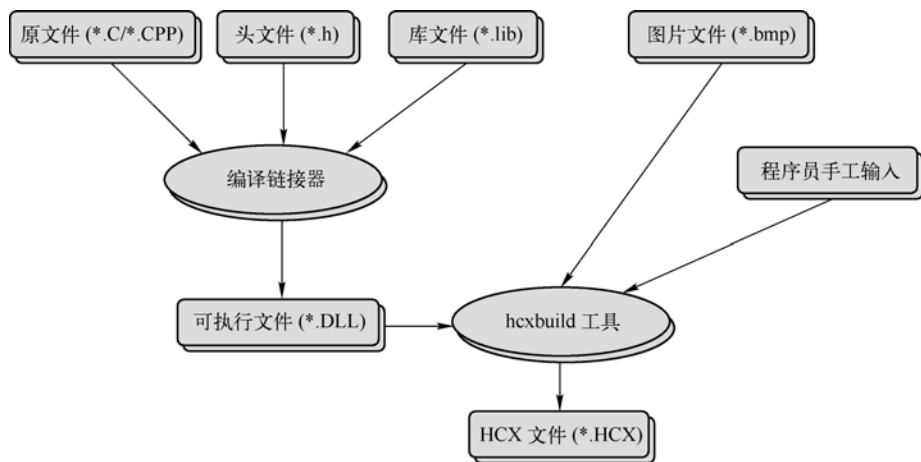


图 13-1 HCX 文件的生成方式

13.2.3 HCX 文件的加载和执行

GUI Shell 启动后，会在 C:\HCGUIAPP 目录下检索所有后缀是.HCX（或小写的 hcx）的文件。一旦发现一个 HCX 文件，GUI Shell 就会读取该文件的相关信息（主要是应用程序图标和可视化名字），然后把这个应用程序显示在程序列表中。

一旦用户点击一个应用程序，该程序对应的 HCX 文件就会被 Hello China 读入内存，并做检查。当前版本的 Hello China 主要检查 HCX 的签名是否正确，HCX 的尺寸是否符合要求，操作系统的版本信息和 HCX 目标操作系统版本信息是否匹配等。如果检查通过，则操作系统会创建一个核心线程，以 HCX 文件的起始位置作为线程的入口点，启动应用程序的运行。

需要说明的是，Hello China V1.75 版本的应用程序加载功能比较简单，未实现复杂的应用程序加载功能（比如代码重新定位、加载资源等）。但由于 Hello China 定位于嵌入式应用，这种加载功能在大部分应用场景下足够应用了。下面是 V1.75 版本应用程序加载器的一些限制：

（1）未实现可执行代码的重新定位，而是把所有应用程序都加载到内存地址 0x1E0000 开始处。这就要求在编译应用程序的时候，必须设置其链接基地址为 0x1E0000，否则会运行失败。

（2）限制应用程序的可执行部分（包括二进制代码和全局数据，不包括图标等辅助数据）不能大于 64KB。这是由于在 0x1E0000 开始处，只有 64KB 的空闲空间可以使用。实际表明，这个限制也不是大问题，64KB 的纯代码空间已经非常大，可以容纳数万行 C 语言代



码的编译结果。如果应用程序的大小超过了 64KB，可以通过修改内核的加载地址来解决该问题。Hello China 的后续版本将在保持兼容的情况下解决该问题。

13.3 Hello China 应用程序开发步骤

下面以 Visual Studio 2008 为例，介绍如何开发一个基于 HelloChina 的应用程序。对于其他版本的 Visual Studio 开发环境，开发步骤基本一致。

13.3.1 建立应用程序开发环境

把 Hello China SDK（应用程序开发套件）中的 sdklib.lib 文件复制到 VS 2008 的 lib 目录下，把 kapi.h 文件复制到 VS 2008 的 include 目录下。如果采用缺省安装，VS 2008 的 lib 和 include 目录分别为：C:\Program Files\Microsoft Visual Studio9.0\VC\lib 和 C:\ProgramFiles\Microsoft Visual Studio 9.0\VC\include。

这样在编写应用程序的时候，直接在源代码文件中包含 KAPI.H 文件即可，无需指明路径。同样地，在链接程序的时候，也无需指明 sdklib.lib 文件，VS 2008 会在 LIB 目录中自动搜索。

KAPI.H 文件包含了 Hello China 应用开发相关的所有类型定义、数据结构定义、函数接口定义等，类似于 Windows 操作系统的 windows.h 文件。任何应用程序源代码文件，必须包含该头文件。

而 sdklib.lib 则是编译后的二进制代码库。所有 Hello China 提供的 API 函数的实现，都被提前编译到该文件中。这样就无需把 API 源文件包含到项目中，链接器会自动到 sdklib.lib 文件中复制二进制代码。

13.3.2 启动 VS 2008，建立一个新的应用程序

选择 VS 2008 集成开发环境的“文件→新建→项目”菜单，在弹出的对话框中，项目类型选择 Visual C++→Win 32，在模版窗口中选择“Win32 项目”，输入一个应用程序名称和解决方案名称（比如“hcnhello”，这两者缺省是相同的，也可以分别指定不同的名字，但似乎没有必要这样做），指定应用程序的存放路径，如图 13-2 所示。

点击“确定”按钮，即可进入 Win 32 应用程序创建向导。点击“下一步...”按钮，在出现的对话框中，选择“应用程序类型”为“DLL”，勾选“空项目”选项，如图 13-3 所示。

单击“确定”按钮，即可建立一个全新的解决方案。

13.3.3 在应用程序中添加源代码

在新建的应用程序中添加源代码文件（.CPP 文件），使得应用程序至少包含一个 C/C++ 语言源代码文件。这是因为如果不包含源代码文件，VS 2008 将不显示编译器选项，从而无法对应用程序进行设置（即第四步工作）。

当然，也可以在这一步中直接编写源代码，编写完成后再进入第四步（设置编译/链接选项）和第五步。

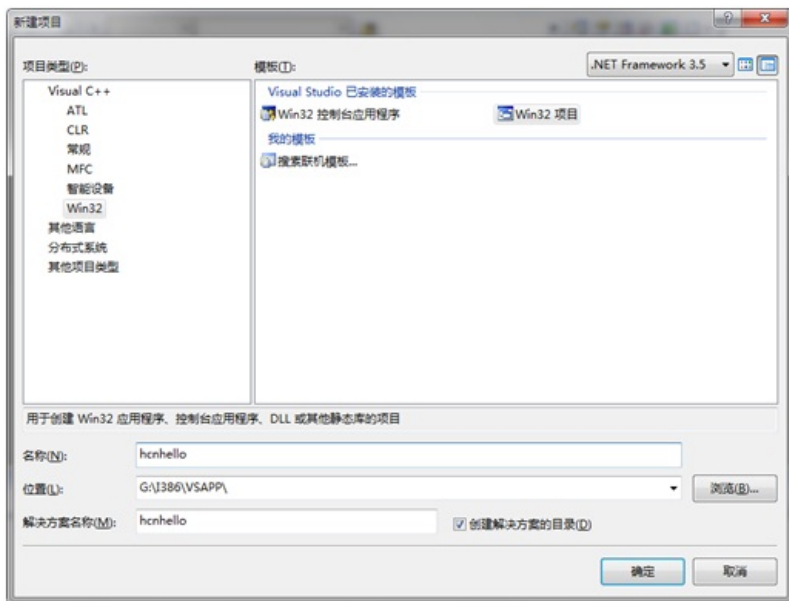


图 13-2 建立一个新的应用程序



图 13-3 选择应用程序类型为 DLL

添加源代码文件的步骤比较简单，选择“文件→新建→文件...”菜单，在出现的对话框中选择“C++文件”即可。输入代码后保存该文件，然后添加到新建的项目中。VS 2008 在缺省情况下不会把新建的文件加入项目，这与 VC 6.0 不同。个人感觉这样非常不方便，不知道 VS 2008 是基于何种考虑。

13.3.4 对新建的应用程序进行设置

新应用程序创建完成之后，VS 2008 会采用缺省设置对其进行配置。这些缺省设置都是针对 Windows 操作系统做出的，与 Windows 操作系统关联密切，但是与 Hello China 操作系统不兼容。为了使新建的应用程序适应 Hello China 操作系统的运行环境，必须对其进行合理设置。这也是 Hello China 开发中最重要的一个步骤。

第一个要做的设置，是编译器生成代码的方式。缺省情况下，VS 2008 会在代码中插入诸如“异常处理”、“缓冲区检查”等确保应用程序安全运行的代码。这些代码对程序员是透明的，即在程序的源文件中看不到这些代码，只有在编译的时候，才由编译器插入。这些代码的实现机制，往往依赖于 Windows 操作系统机制，因此必须禁止编译器插入这些代码。具体设置方式为：选择“项目→属性”菜单，在弹出的对话框中，选择“配置属性”，左上角的“配置”中，选择“Release”，如图 13-4 所示。

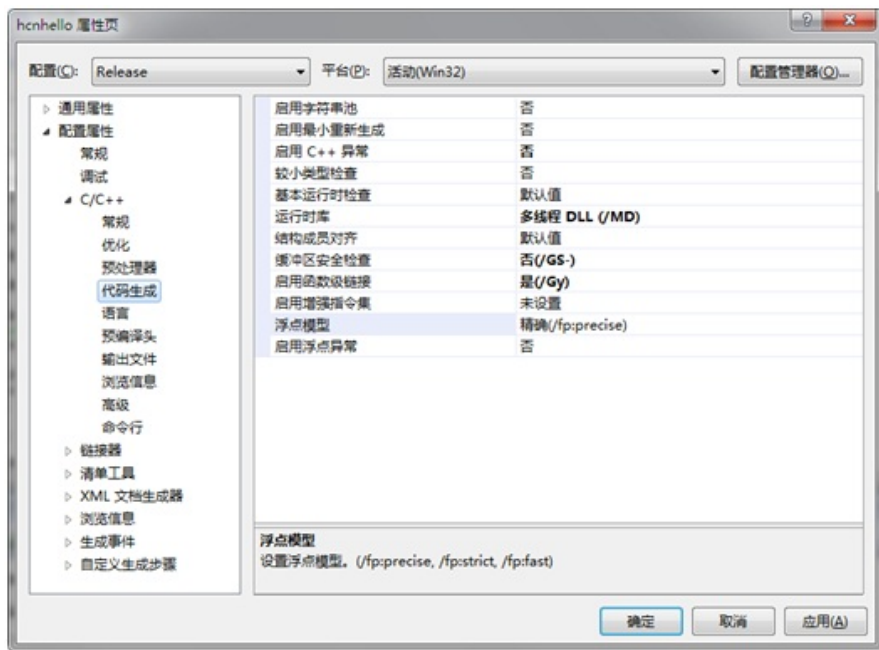


图 13-4 对项目进行设置

选择“C/C++→代码生成”，在右面的配置列表中，把“启用 C++异常”选项修改为“否”，“缓冲区安全检查”选项设置为“否 (/GS-)”，如图 13-4 所示。

再选择“链接器→高级”选项，在右面的选项列表中做如下配置：

入口点：输入“HCNMain”，即程序的入口函数名称。

基址：设置为“0x1E0000”，即应用程序的加载地址。

其他选项保持默认值即可，如图 13-5 所示。

再选择“链接器→命令行”配置项，在右面的“附件选项”编辑框中，输入如下附加选项：sdclib.lib /ALIGN:16。

其中 sdclib.lib 告诉链接器要到该文件中寻找相关函数的目标代码。所有 Hello China 操

作系统相关的功能函数的二进制代码，都在该文件中。而/ALIGN 选项，则是告诉链接器，在链接应用程序的时候，节（section）与节之间的间隔应该按照 16 字节对齐。缺省情况下，是按照 4K 字节对齐的，这不符合 Hello China 对应用程序的要求。

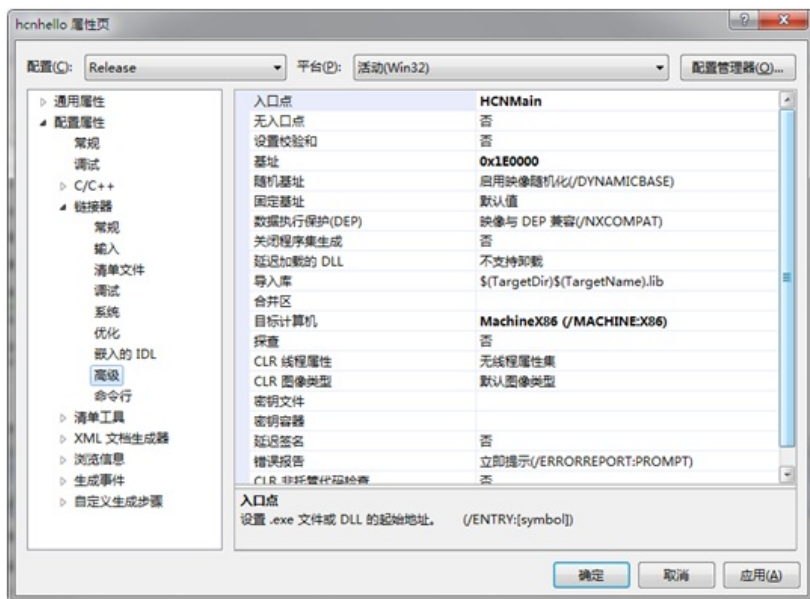


图 13-5 对链接选项进行设置

设置后的结果如图 13-6 所示。

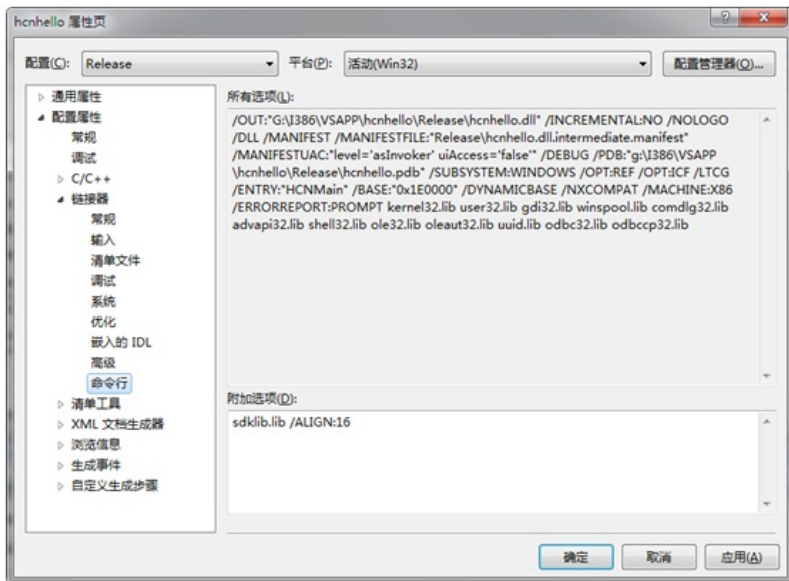


图 13-6 设置结果

至此编译链接选项设置完毕。

13.3.5 编写应用程序代码，并进行编译链接

编写代码是应用开发的最核心步骤，也是工作量最大的步骤。具体的代码与应用程序的功能相关，无法统一描述。但是 Hello China 应用程序的大致架构是相同的，每个应用程序必须包含下列关键要素：

- (1) 一个消息循环，循环从应用程序（线程）的消息队列中获取消息，并进行分发。
- (2) 至少一个窗口，用于显示所有用户界面元素，完成用户交互。
- (3) 至少一个窗口函数，对应上面的窗口，处理所有窗口相关的消息和用户动作。
- (4) 一个入口点（入口函数，即 HCNMain），操作系统加载应用程序，并从该点开始运行。

熟悉 Windows API 编程的读者很容易发现，这个逻辑结构与 Windows 应用程序结构类似。实际上，Hello China 也实现了基于消息的驱动机制，由外部消息（用户输入）来驱动程序的运行。下面是一个简单应用程序的代码，包含了上述各个要素：

```
#include <kapi.h>
//主窗口函数，处理各类窗口消息。
static DWORDHelloWndProc(HANDLE hWnd,UINT message,WORD wParam,DWORD lParam)
{
    staticHANDLE hDC = GetClientDC(hWnd);
    __RECT rect;
    switch(message)
    {
        caseWM_CREATE:    //Will receive this message when the window is created.
            break;
        caseWM_TIMER:    //Only one timer can be set for one window in current version.
            break;
        caseWM_DRAW:
            TextOut(hDC,0,0,"Hello,world!");
            break;
        caseWM_CLOSE:
            PostQuitMessage(0); //一旦用户关闭主窗口，则向应用程序的消息队列中发送结束消息。
            break;
        default:
            break;
    }
    returnDefWindowProc(hWnd,message,wParam,lParam); //必须调用缺省窗口函数。
}
//入口点函数。
extern "C"
{
    DWORD HCNMain(LPVOIDpData)
    {
        MSG msg;
        HANDLE hMainFrame = NULL;
        __WINDOW_MESSAGE wmsg;
```

```
//创建应用程序主窗口。
hMainFrame =CreateWindow(WS_WITHBORDER | WS_WITHCAPTION,
    "Mainwindow of the demonstration application",//窗口标题。
    150,//窗口在屏幕上的位置。
    150,
    600,
    400,
    HelloWndProc, //窗口函数
    NULL,
    NULL,
    0x00FFFFFF,    //窗口背景颜色，纯白色。
    NULL);
if(NULL== hMainFrame)
{
    MessageBox(NULL,"Can not create the main frame window.", "Error",MB_OK);
    goto __TERMINAL;
}
//消息循环，一个线程只有一个消息循环，从线程消息队列中获取消息。
while(TRUE)
{
    if(GetMessage(&msg))
    {
        switch(msg.wCommand)
        {
            case KERNEL_MESSAGE_WINDOW:
                DispatchWindowMessage((__WINDOW_MESSAGE*)msg.dwParam);
                break;
            case KERNEL_MESSAGE_TERMINAL: //Post byPostQuitMessage.
                goto __TERMINAL;
            default:
                break;
        }
    }
}

__TERMINAL:
if(hMainFrame)
{
    DestroyWindow(hMainFrame);
}
return 0;
}
```

上述代码只是创建了一个应用程序主窗口，并在窗口的左上角输出了“Hello,world!”字符串。

代码编写完毕，即可进行编译链接了。如果仅仅是编译一个 C/C++ 源文件，只要直接按“Ctrl +F7”组合键，即可完成编译。如果希望构建整个应用程序，形成最终的二进制模块并在 Hello China 上运行，则必须选择“生成→批生成...”菜单，在弹出的对话框中勾选“Release|Win32”，然后点击“重新生成”按钮即可。

这样生成的 DLL 文件，存放在项目的 Release 目录下。需要注意的是，这个生成的 DLL 是不能直接被 Hello China 执行的，必须对其进行处理，也就是下一步的工作。

13.3.6 对生成的 DLL 进行处理，形成 HCX 文件

VS 2008 编译链接生成的 DLL 文件不能被 Hello China 直接运行，而必须对其进行设置。SDK 中携带的 hcxbuild 工具，就是完成这个处理工作的。在处理之前，请准备一个大小是 128×128 像素、颜色深度是 24 位的位图文件，作为应用程序的图标。如果没有符合要求的位图文件，可以使用 Windows 自带的画图程序进行制作，非常方便。

准备好位图文件后，即可启动 hcxbuild 程序，来生成 HCX 文件了。图 13-7 是 hcxbuild 的运行截图。

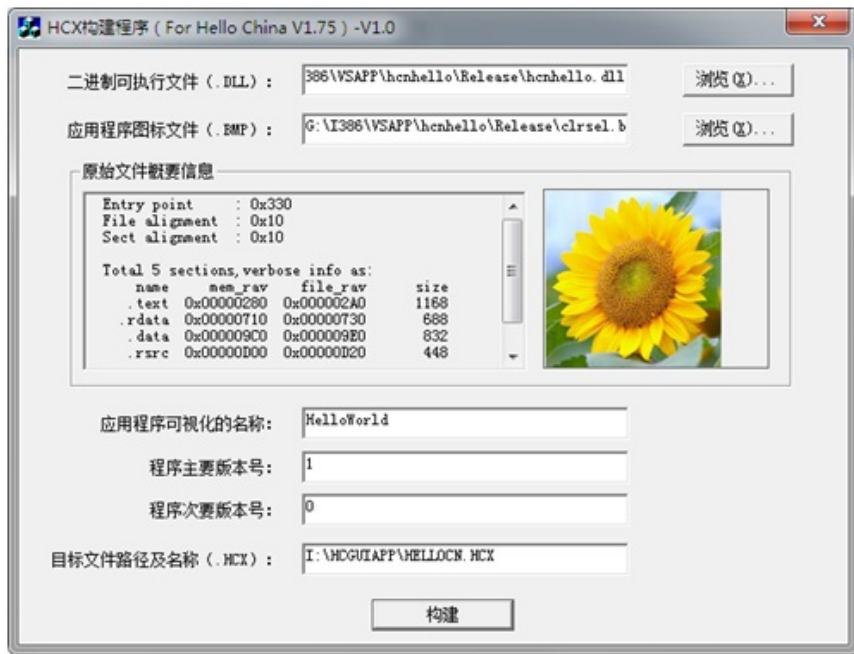


图 13-7 HCXBUILD 程序运行截图

其中“二进制可执行文件”就是刚刚编译完成的 DLL 文件，“应用程序图标文件”则是刚才准备好的位图文件。一旦这两个文件被选择，“原始文件概要信息”中就会输出这两个文件的相关信息，其中对 DLL 文件，会输出其入口地址、所有节的数量、每个节的文件开始地址和长度等信息。对于位图文件，则直接显示其内容。

“应用程序可视化的名称”，是该应用程序显示在 Hello China 操作系统的图形 shell 中的名称，这个名称与应用程序文件的名称不同。程序主要/次要版本两个参数，主要用于应用

程序版本控制。“而目标文件路径及名称”，则是待生成的 HCX 文件的存放位置和名称。注意，文件名称后缀一定要为全部大写的 HCX 或全部小写的 hcx，否则会提示错误。

所有信息输入完毕，点击“构建”按钮，即可生成 HCX 文件。如果提示错误，请根据提示修改相应参数。

生成的 HCX 文件就是 Hello China 可加载和运行的应用程序文件了。

13.3.7 运行生成的 HCX 文件

把上述步骤生成的 HCX 文件，复制到 Hello China 所在硬盘（或虚拟硬盘）的 HCGUIAPP 目录下，重新启动 Hello China，进入字符命令行模式后，再输入 gui 命令并回车，即可进入图形模式，在图形 shell 中就可看到新开发的应用程序了。点击应用程序图标，即可运行该应用程序。图 13-8 是 HelloWorld 应用程序被 GUI Shell 加载后的运行结果。



图 13-8 HelloWorld 应用程序被成功加载

单击“HelloWorld”图标后，运行结果如图 13-9 所示。

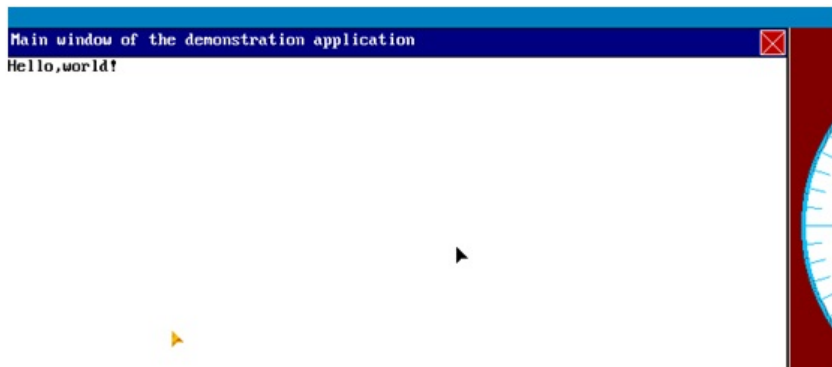


图 13-9 HelloWorld 程序的运行结果



13.4 应用程序开发总结

本章详细介绍了 Hello China 图形模式下的应用程序开发方法。首先介绍了 HCX 文件格式，然后以 Microsoft Visual Studio 2008 为例，详细介绍了开发一个图形应用程序的步骤。最后以一个 Hello World 实例作为结束。虽然是以 VS 2008 为开发环境进行介绍的，但是各种环境的工作原理却是相同的，读者可以按照本章中介绍的原理和原则，在其他开发环境中完成应用程序的开发，如 GCC、VC6.0 等。

作者认为，理解一个操作系统工作原理的最好方法，就是开发一些基于这个操作系统的应用程序。毕竟所有的操作系统都是为应用程序服务的，所有操作系统的机制和原理，都以 API 等形式提供给应用程序。通过应用程序的开发，可深入理解这些 API，进而可一窥操作系统本身的概貌。

当然，如果只是停留在应用程序的开发上，对操作系统核心的理解还是不够的。在熟练掌握应用程序的开发方法和关键 API 后，还是要阅读操作系统核心代码，来深入掌握操作系统的内部机理。作者也建议读者按照这种顺序完成对操作系统的理解，即首先尝试开发一些应用程序，在熟练掌握应用程序的开发方法和关键 API 之后，再阅读操作系统核心代码，将会得到事半功倍的效果。

第 14 章 开发辅助工具

14.1 开发辅助工具概述

在操作系统开发过程中，除用到编译器链接器、代码编辑器、调试器、文档管理工具等常规通用工具外，还会用到大量的辅助工具。所谓辅助工具，指的是专门为一个特定操作系统开发而制作的专用小型软件。

常规的开发工具生成的是通用的、格式固定的二进制可执行模块。大多数情况下，这些二进制可执行模块不能被直接加载到内存中运行，而必须由操作系统的加载器进行一些预处理才能运行。而操作系统核心模块在加载之前，是没有任何已有软件（比如加载器）进行支撑的，这时候如果把操作系统核心模块也编译成这种需要预处理的二进制格式，显然无法正常工作。因此需要一些辅助的工具，把编译器生成的二进制模块处理成可直接加载到内存中运行的二进制模块。

有的时候，为了加载方便，需要把几个二进制模块连接在一起，形成一个大的二进制模块，这时候也需要辅助工具的帮助。还有一种情况是，在启动虚拟机（比如 Virtual PC）的时候，需要有一个虚拟软盘文件。编译器是无法生成这个虚拟软盘文件的，必须由辅助工具完成。

在 Hello China 的开发过程中，同样也会用到一些辅助工具。本章就对 Hello China 开发过程中用到的一些辅助工具的原理和用法进行介绍，以使读者对操作系统的运行原理有更进一步的理解。虽然操作系统的核心机制和实现原理与这些辅助工具无关，但千万不要小看这些工具，它们在整个操作系统开发过程中起到黏合剂的作用。正是因为这些工具的存在，才使得操作系统核心的不同模块有效结合、统一运行。

Hello China V1.75 的 PC 实现版本，使用的是 Visual C++ 作为开发环境。缺省情况下，这个开发环境生成的二进制可执行文件都是 PE（Portable Executable）格式的。而 PE 格式文件显然不能直接作为操作系统核心模块使用，必须经过处理。本书引入的几个工具，最主要的工作就是对 PE 格式的文件进行处理，使之变成一种可直接加载运行的格式。在正式介绍辅助工具之前，先简单介绍一下 PE 文件格式，这是理解辅助工具的基础。

14.2 PE 文件格式简介

PE 文件格式内容比较多，如果读者希望对 PE 格式有进一步了解，可到互联网上去查阅相关资料。当然，理解了本部分内容，PE 文件格式的大致轮廓也就建立起来，再看进一步的内容时，也会比较容易。



图 14-1 示意了 PE 文件的主要组成。

下面对 PE 文件格式的每个部分做简略介绍。

14.2.1 MS-DOS 头和 DOS stub 程序

PE 文件格式的第一个组成部分是 MS-DOS 头。熟悉 DOS 操作系统的读者会清楚，DOS 头并非一个新概念，它与 MS-DOS 2.0 以来就有的 MS-DOS 头是完全一样的。保留这个相同结构的最主要原因是，当你尝试在低版本的 Windows（比如 16 位的 Windows）或者 DOS 操作系统下运行一个 PE 文件的时候，操作系统能够读有效地识别这个文件，并告诉用户这个文件不能在低版本的 Windows 操作系统下运行。如果 MS-DOS 头不是作为 PE 文件格式的第一部分的话，那么 DOS 操作系统将不能正确识别这个文件。作者认为，这也没有什么大碍，无非是不能运行而已。

下面是 DOS 头的格式定义，它占用了 64B：

```
typedef struct _IMAGE_DOS_HEADER {
    USHORT e_magic;    //魔术数字
    USHORT e_cblp;     //文件最后页的字节数
    USHORT e_cp;       //文件页数
    USHORT e_crlc;     //重定义元素个数
    USHORT e_cparhdr;  //头部尺寸，以段落为单位
    USHORT e_minalloc; //所需的最小附加段
    USHORT e_maxalloc; //所需的最大附加段
    USHORT e_ss;       //初始的 SS 值（相对偏移量）
    USHORT e_sp;       //初始的 SP 值
    USHORT e_csum;     //校验和
    USHORT e_ip;       //初始的 IP 值，即程序入口点
    USHORT e_cs;       //初始的 CS 值（相对偏移量）
    USHORT e_lfarlc;   //重分配表文件地址
    USHORT e_ovno;
    USHORT e_res[4];   //保留
    USHORT e_oemid;    //OEM 标识符
    USHORT e_oeminfo;  //OEM 信息
    USHORT e_res2[10]; //保留
    LONG e_lfanew;     //PE 头的地址
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

MS-DOS 头
DOS stub 程序
PE 文件头
节头表
节 1
节 2
.....
节 N

图 14-1 PE 文件格式的构成

这些字段都是用于 MS-DOS 处理可执行文件时使用的，因此大部分字段的含义不用深入追究。这里重点关注一下 `e_lfanew` 变量，即最后一个变量。这个变量指向了 PE 文件头的地址（PE 文件头在文件中的位置），我们就是通过这个字段，来获得 PE 文件头信息的。

前面我们提到，当你尝试在 MS-DOS 6.0 下运行一个 Windows NT 的可执行文件时，屏幕会显示这样一条消息：“This program cannot run in DOS mode”。这句话不是操作系统输出的，而是由 PE 文件头中的 DOS stub 程序输出的。即 DOS 在尝试运行一个 PE 文件的时候，

首先检查其文件头是不是一个合法的 MS-DOS 头。显然，编译器会放置一个合法的 MS-DOS 头在 PE 文件的开始处。于是 MS-DOS 会认为这是一个可执行的 DOS 程序，于是“很高兴地”按照 DOS 头部的相关信息，加载这个程序并开始运行。显然，编译器会把 MS-DOS 头中的 `e_ip` 和 `e_cs` 等变量设置为指向 DOS stub 程序的开始处。最终结果是，DOS 运行的只是一个 stub 程序，这个 stub 程序打印出上述提示信息，然后结束。DOS stub 程序是由编译器添加上去的，当然，你也可以告诉编译器，在 PE 文件头中添加一个更加复杂的程序。比如有很多可执行程序，不论是在 DOS 下，还是在 Windows 下，都能够正确运行，但显示不同的界面。这就是通过替换缺省的 stub 程序做到的。这个可执行程序需要准备两个版本——16 位的 DOS 版本和 32 位的 Windows 版本，然后把 16 位的 DOS 版本连接在 DOS stub 程序位置处。这样就实现了单一文件、双系统运行的效果。

14.2.2 PE 文件头

PE 文件头 (`IMAGE_NT_HEADER`) 才真正存放了 PE 文件的有用信息。PE 文件头包含三个数据成员：一个数字签名和两个文件头数据结构——`IMAGE_FILE_HEADER` 和 `IMAGE_OPTIONAL_HEADER`，这些数据结构都是在 `winnt.h` 头文件中定义的。我们看一下 `IMAGE_NT_HEADER` 结构的定义：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

其中数字签名 (Signature) 指出了这个 PE 文件适应的目标操作系统，需要注意的是，PE 格式的文件不仅适用于 Windows NT 系列操作系统，还被应用在 OS/2 等操作系统内。对于 NT 系列操作系统，这个签名的值为 `0x00004550`，其中 `0x4550` 即是“PE”的 ASCII 码。

`IMAGE_FILE_HEADER` 的定义如下：

```
typedef struct _IMAGE_FILE_HEADER {
    WORD Machine;
    WORD NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD SizeOfOptionalHeader;
    WORD Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

`IMAGE_FILE_HEADER` 结构中 with 后续内容关系比较密切的变量有三个：`Machine`、`NumberOfSections` 和 `SizeOfOptionalHeader`。`Machine` 可以用来判断目标平台，即运行的目标 CPU。操作系统在加载一个 PE 文件的时候，首先检查这个变量是否与当前的 CPU 类型吻合。如果吻合则继续运行，否则就放弃进一步运行。`SizeOfOptionalHeader` 指出 `IMAGE_OPTIONAL_HEADER` 结构，即 `IMAGE_NT_HEADERS` 中的第二个结构体的大小。显然，



IMAGE_OPTIONAL_HEADER 是一个大小可变的结构体，其大小由该变量确定。

对我们来说最重要的是 NumberOfSections 变量，这个变量记录了 PE 文件中节（section）的个数。可执行文件的相关数据，都以节的形式存放在 PE 文件中，如代码节（.text）、全局数据节（.data）、未初始化的数据节（.bss）等。节的具体内容，在 14.2.3 中讲解。

另外一个文件头——IMAGE_OPTIONAL_HEADER，是更重要的一个头，虽然其名字中包含 optional，但该头绝不是可选的，而是必须的。我们的开发辅助工具在对 PE 文件进行处理的时候，重点就是针对该结构所包含的信息，对文件进行处理。

下面是其定义：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    // 标准域
    USHORT Magic;
    UCHAR MajorLinkerVersion;
    UCHAR MinorLinkerVersion;
    ULONG SizeOfCode;
    ULONG SizeOfInitializedData;
    ULONG SizeOfUninitializedData;
    ULONG AddressOfEntryPoint;
    ULONG BaseOfCode;
    ULONG BaseOfData;
    // NT 附加域
    ULONG ImageBase;
    ULONG SectionAlignment;
    ULONG FileAlignment;
    USHORT MajorOperatingSystemVersion;
    USHORT MinorOperatingSystemVersion;
    USHORT MajorImageVersion;
    USHORT MinorImageVersion;
    USHORT MajorSubsystemVersion;
    USHORT MinorSubsystemVersion;
    ULONG Reserved1;
    ULONG SizeOfImage;
    ULONG SizeOfHeaders;
    ULONG CheckSum;
    USHORT Subsystem;
    USHORT DllCharacteristics;
    ULONG SizeOfStackReserve;
    ULONG SizeOfStackCommit;
    ULONG SizeOfHeapReserve;
    ULONG SizeOfHeapCommit;
    ULONG LoaderFlags;
    ULONG NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

这是一个庞大的数据结构，但大多数变量与我们没有关系，我们重点关注上述定义中用

黑体标注出来的四个变量，下面分别进行说明。

AddressOfEntryPoint，这个域表示应用程序入口点的位置，这个位置是程序被加载到内存，完成全部预处理后的位置，而不是 PE 文件在磁盘上的位置。需要注意的是，PE 文件在磁盘上的存储格式，与被操作系统加载到内存并处理后的格式，大多数情况下是不相同的。因此在 PE 格式的相关描述数据结构（比如我们介绍的这些 **HEADER** 结构）中，对于一个具体位置，一般会用两种方式来描述：虚拟相对位置和文件相对位置。所谓虚拟相对位置，是 PE 文件被加载到内存并预处理后的位置。由于 PE 文件加载到内存的起始地址是不固定的，因此这里的虚拟相对位置是一个“相对”值。其绝对值则是 PE 文件的加载地址加上虚拟相对位置。**AddressOfEntryPoint** 就是一个虚拟相对位置。比如一个 PE 文件，PE 头部中定义的 **AddressOfEntryPoint** 为 0x400，这是个虚拟相对位置。但是 PE 文件被加载到内存后，其加载地址是 0x400000，则该可执行文件入口点的真正地址是 0x400400。

另外一个描述位置的方法，是文件相对位置，即相对文件开始处的具体存储位置。理解虚拟相对位置和文件相对位置的前提，是理解 PE 文件在磁盘上的存储方式，与最终加载到内存后的布局是不一样的。我们前面讲过，PE 文件的主体部分是由许多节组成的。不同的节，在磁盘上可能是连续存放，也可能是以 16B 为单位进行对齐的。但是在加载到内存后，缺省情况下却是以 CPU 的页面尺寸为单位对齐的，比如以 4KB 为单位对齐。显然，文件被加载到内存后的布局“膨大”了。虚拟相对位置用的是文件被加载到内存后“膨大”的布局为基础进行描述的，而文件相对位置则是以文件在磁盘上的存储布局为基础进行描述的。还是以上述 **AddressOfEntryPoint** 为例，其虚拟相对位置是 0x400，但是其文件相对位置可能是 0x200。

理解虚拟相对位置和文件相对位置后，剩下的几个变量就容易理解了。我们先看 **SectionAlignment** 和 **FileAlignment**。**SectionAlignment** 是不同的节被加载到内存后的对齐大小。比如一个节的实际大小是 3KB，但是 **SectionAlignment** 是 4KB，则这个节必须占用 4KB 的空间，下一个节需要从另一个 4KB 的开始处进行加载。**FileAlignment** 与此类似，就是一个节在文件中的对齐大小。比如一个节的大小是 3KB，但是 **FileAlignment** 是 4KB，则该节存放在磁盘上的时候，也必须占用 4KB 空间。

ImageBase 变量的含义很简单，是 PE 模块被加载到内存后的基地址。需要注意的是，这个基地址只是对操作系统的一个建议，即建议以这个地址为基地址加载 PE 模块。操作系统可以以这个地址进行加载，但是如果这个地址被占用（加载 DLL 时，经常遇到），则操作系统需要找另外的地址进行加载。如果以 PE 头中的 **ImageBase** 进行加载，则无需对代码进行重定位操作，因为编译器缺省按照 **ImageBase** 为基地址对代码进行编译和链接。如果以不同于 **ImageBase** 的地址加载 PE 文件，则操作系统需要对加载后的 PE 文件重定位。这个加载地址可以使用编译器的 **/BASE** 选项进行修改，因为 Hello China 内核的加载地址是预先设计好的，因此我们在编译 Master 等操作系统核心模块的时候，就使用 **/BASE** 选项对编译器进行了配置，使得编译器以我们指定的基地址为依据对源代码进行编译和链接。

至此，我们把 PE 头，即 **IMAGE_NT_HEADER** 结构简略地讲完了，下面看一下如何访问这些结构体中的变量。假设 PE 文件被读入内存的 **pBinFile** 位置处，下面是一段显示 PE 文件头关键信息的代码：



```
void showPE(char* pBinFile) //pBinFile 指向 PE 文件在内存中的开始处。
{
    IMAGE_DOS_HEADER*      ImageDosHeader = NULL;
    IMAGE_NT_HEADERS*      ImageNtHeader = NULL;
    IMAGE_OPTIONAL_HEADER* ImageOptionalHeader = NULL;

    ImageDosHeader = (IMAGE_DOS_HEADER*)pBinFile;
    dwOffset = ImageDosHeader->e_lfanew; //找到 PE 头相对文件头的偏移
    ImageNtHeader = (IMAGE_NT_HEADERS*)(pBinFile + dwOffset); //找到 PE 头
    ImageOptionalHeader = &(ImageNtHeader->OptionalHeader); //找到 Optional 头
    printf("Section alignment = %d\r\n", ImageOptionalHeader->SectionAlignment ""); printf("File
alignment = %d\r\n", ImageOptionalHeader->FileAlignment);
    printf("Address of Entrypoint = %X\r\n", ImageOptionalHeader->AddressOfEntryPoint);
    return;
}
```

代码比较简单，无非是通过层层定位，找到对应的结构体的开始处，然后用结构体指针访问感兴趣的变量即可。

14.2.3 PE 文件中的节

节是 PE 文件的主体部分，代码、全局数据等，都是以节进行存放的。一个 PE 文件中具体包含的节的数量，是由 PE 文件头中的 `NumberOfSections` 变量决定的。通过读取这个变量，可确定文件中的节的数量。

每个节都有一个对应的节头对其进行描述，所有节的节头统一存放在 PE 文件中的节头表中，这个表紧跟着 PE 文件头。下面是节头的定义：

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress;
    ULONG SizeOfRawData;
    ULONG PointerToRawData;
    ULONG PointerToRelocations;
    ULONG PointerToLinenumbers;
    USHORT NumberOfRelocations;
    USHORT NumberOfLinenumbers;
    ULONG Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

下面对其中几个关键的字段进行描述：

(1) **Name:** 每个段都有一个 8 字符长的名称域，并且第一个字符必须是一个句点。

(2) **VirtualAddress**: 即节的虚拟相对地址, 实际的内存装入地址, 由这个域的值加上可选头部结构中的 **ImageBase** 虚拟地址得到。注意, 如果这个映像文件是一个 DLL, 那么这个 DLL 就不一定会装载到 **ImageBase** 要求的位置。一旦这个文件被装载进入了一个进程, 实际的 **ImageBase** 值应该通过使用 **GetModuleHandle** 来检验。

(3) **SizeOfRawData**: 这个域表示了节的实体尺寸。

(4) **PointerToRawData**: 节在文件中的具体位置, 是一个文件相对地址。通过这个地址以及 **SizeOfRawData** 变量, 即可从文件中找到节的具体位置和大小。

PE 文件中有几个节, 则节头表中会包含几个节头。节头后面则跟着具体的节数据。在文件中寻找一个具体的节的时候, 必须通过节的名字来查找。下面是一段根据节的名字, 来获得对应节头的代码:

```
BOOL GetSectionHeaderByName(LPVOID lpFile, IMAGE_SECTION_HEADER *sh, char
*szSectionName)
{
    PIMAGE_SECTION_HEADER psh;
    int nSections = NumOfSections(lpFile); //从 PE 文件头中获取节的数量。
    int i;
    if ((psh = GetSectionHeader(lpFile)) != NULL) //定位到文件的节头表位置处。
    {
        // 根据名称查找节
        for (i = 0; i < nSections; i++)
        {
            if (!strcmp(psh->Name, szSection))
            {
                //返回节头数据
                CopyData((LPVOID)sh, (LPVOID)psh,
                    sizeof(IMAGE_SECTION_HEADER));
                return TRUE;
            }
            else
                psh++;
        }
    }
    return FALSE;
}
```

对于节来说, 再重点说明几点:

(1) 节在文件内的位置, 与被加载到内存后的位置会不同。这样的—个结果就是, 不能直接把 PE 文件加载到内存, 不作处理地直接跳转到入口处运行。而必须根据 PE 头和节头的相关信息, 对节进行重新定位。在操作系统的开发中, 最开始的时候还没有 PE 文件加载程序, 因此必须对 PE 文件进行处理, 使得其在磁盘上的位置, 跟加载到内存后的布局保持一致。这样引导程序就可直接把 PE 文件读入内存, 直接跳转执行。完成这项工作的, 就是



即将介绍的辅助工具。

(2) 有些节在内存中是存在的，而在文件中却不一定存在。比如.BSS 节，这是未初始化的全局变量节。这个节在文件中存储的时候，只有一个节头，说明其大小，而无需存储具体的节。因为对于未初始化的全局变量，只需要在内存中预留空间即可，无需进行初始化。这时候为了直接加载和运行的目的，必须用工具把这些在文件中“不存在”的节，补充到文件中。这也是辅助工具的工作。

14.3 开发辅助工具的实现和使用

下面对 Hello China 开发过程中使用的几个辅助工具的工作原理以及大致的使用方法进行简要介绍。

14.3.1 process 工具

VC 生成的可执行二进制模块是 PE 格式的，前面我们讲过，文件的开始处是一个 PE 文件头，而这个头的长度是可变的。同时在这个头中的特定偏移处（即 IMAGE_OPTIONAL_HEADER 中的 AddressOfEntryPoint）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 开发的要求是直接找到模块的入口地址，通过一条跳转指令跳转到该地址开始执行。

为了解决这个问题，可以通过对目标文件进行修改的方式来解决。process 工具就是完成这项工作的。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 OS 核心模块的加载过程中，只要把这个模块读入内存，并直接跳转到开始处执行即可。

但是这样做的前提是，PE 文件在磁盘上的布局，与其被加载到内存后的布局保持一致。即前面介绍的两个概念——虚拟相对地址和文件相对地址，是一样的。在 Visual C++ 6.0 中，可通过设定/Alignment 和/Base 等几个选项，来确保这个条件能够满足。同时通过设置/Entry 编译选项，使得入口点指向一个特定的入口函数（比如__init 函数）。根据作者的经验，这在 VC 6.0 下是可以正常工作的，但是到了 VS 2003 以上的编译环境，就出现了问题，通过设置这些选项不能解决内存布局 and 文件布局一致的问题。具体解决方案，在 14.3.2 节中再进行介绍。

process 工具的工作原理非常简单，就是读取 PE 文件，从 PE 头中找到 AddressOfEntryPoint，然后把 AddressOfEntryPoint 的值填写到 PE 头中。这样处理的结果是，PE 头会被破坏，至少 MS-DOS 头已经被破坏。但这不要紧，Hello China 不关注 MS-DOS 头。

具体代码比较简单，我们就不看了。这个工具的使用方式如下：

```
process -i src_filename -o dst_filename
```

工具把 src_filename 指定的文件读入内存，修改其文件头部，然后把修改后的文件重新写入到 dst_filename 文件中。这样源文件的内容不会受到影响。

14.3.2 hcxbuild 工具

hcxbuild 工具用于构建 HCX (Hello China eExecutable) 文件, 即 Hello China 的可执行文件。应用程序经开发环境编译后, 生成的是 PE 格式的可执行文件。而 hcxbuild 则对 PE 文件进行进一步的修改, 使之符合 Hello China 的加载需要。具体来说, 这个工具对 PE 文件做了如下修改:

(1) 使用一个新定义的文件头 `__HCX_HEADER`, 覆盖 MS-DOS 头。

(2) 对 PE 格式的每个节进行分析, 根据它们在内存中的位置重新进行了定位, 确保其存储格式与加载格式一致。这样的结果就是, 操作系统只需要把文件读入内存即可直接运行, 无需再对各个节做重定位。

(3) 在 PE 文件的末尾处增加一个图标, 用于显示在操作系统的 GUI shell 主屏幕上。

hcxbuild 的运行结果: 如图 14-2 所示。



图 14-2 hcxbuild 的运行结果

其中二进制可执行文件, 就是待处理的 PE 文件, 而应用程序图标就是附加在 PE 文件之后的 $128 \times 128 \times 32$ 规格的位图。一旦选择好待处理的 PE 文件, 工具就会把该文件的相关信息, 比如入口点、文件和内存对齐方式、各个节的详细信息等, 都显示在“原始文件概要信息”栏内。一旦选择好图标文件, 文件内容就会被显示在信息栏右面的图标区域内。

应用程序的可视化名称, 是 Hello China 在枚举该应用程序时显示的名字。主要和次要两个版本号, 用于标记程序的版本。最后的目标文件路径及名称, 则是最终生成的 hcx 文件的路径和名称。注意, 最后生成的文件的后缀名, 一定要为 hcx (或大写的 HCX)。

使用该工具对 PE 文件进行处理后, 即可把生成的 HCX 文件复制到 Hello China 的应用程序目录下 (HCGUIAPP 目录), 这样启动 Hello China 的 GUI 功能, 即可看到应用程序。点击应用程序图标, 即可启动这个应用程序了。



hcxbuild 的工作原理比较简单,无非是生成一个 hcx 文件头,然后用这个头覆盖原有的 PE 文件 MS-DOS 头。hcx 文件头中包含了诸如操作系统版本号、入口地址等等信息。这个过程比较简单,我们重点看一下这个工具如何处理节。

在介绍 PE 格式的时候,我们指出,节在文件中的位置,可能与被加载到内存后的位置不一致。在 VC 6.0 中,通过设置/Align 等选项,可以确保节的文件相对地址和虚拟相对地址保持一致,但到了 VS 2003 以上的开发环境,即使这样设置,也不能确保两者一致。下面这个例子说明了这种情况。下面的信息是从 hcxbuild 工具的“原始信息概要”栏中复制的:

```
Entry point      : 0x660
File alignment   : 0x10
Sect alignment   : 0x10
```

Total 6 sections,verbose info as:

name	mem_rav	file_rav	size
.text	0x000002B0	0x000002D0	7216
.rdata	0x00001EE0	0x00001F00	880
.data	0x00002250	0x00002270	944
.CRT	0x00002600	0x00002620	16
.rsrc	0x00002610	0x00002630	448
.reloc	0x000027D0	0x000027F0	304

重点关注黑体标注的内容,其中第一列是节在内存中的加载地址(即虚拟相对地址,对应于节头中的 VirtualAddress 变量),而第二列则是文件相对地址,对应于节头中的 PointerToRawData 变量。显然这两列是不一样的,文件相对地址比虚拟相对地址要大 32 个字节(注意上面是以十六进制显示的)。

这意味着 PE 文件的文件尺寸,要大于被加载到内存后的实际内存占用量。我们需要修改这种情况,使得 PE 文件的内存布局与文件布局一样。具体修改方式也很简单,无非是把源文件中的节读入内存,然后按照其在内存中的虚拟相对地址,写入另外一个文件(即目标 HCX 文件)。现摘录部分代码,说明这个过程。为了便于说明,删除了代码中的不相关内容,比如注释、安全检查代码等:

```
[/tools/hcxbuild/hcxbuildDlg.cpp]
```

```
char* CHcxbuildDlg::InflateSections(char *pBinFile, char *pHCXFile) //pBinFile 是源文件在内存中的  
起始位置, pHCXFile 指向目标文件在内存中的位置
```

```
{  
    char*                psrcBuffer        = pBinFile;  
    char*                pdstBuffer        = pHCXFile;  
    IMAGE_DOS_HEADER*    pDOSHdr           = (IMAGE_DOS_HEADER*)pBinFile;  
    IMAGE_NT_HEADERS*    pNTHdrs           = NULL;  
    IMAGE_FILE_HEADER*   pFileHdr          = NULL;  
    IMAGE_OPTIONAL_HEADER* pOptionalHdr     = NULL;  
    IMAGE_SECTION_HEADER* pSectionHdr       = NULL;  
    DWORD                dwSectNum          = 0;  
    DWORD                dwTotalLength      = 0;
```

```

    BOOL                                bResult                = FALSE;
    DWORD                                i;
    pNTHdrs = (IMAGE_NT_HEADERS*)(pBinFile + pDOSHdr->e_lfanew);
    pFileHdr = &pNTHdrs->FileHeader;
    pOptionalHdr = &pNTHdrs->OptionalHeader; //得到 Optional 文件头
    pSectionHdr = (IMAGE_SECTION_HEADER*)(pBinFile + pDOSHdr->e_lfanew + sizeof
(IMAGE_NT_HEADERS)); //得到节头表的起始地址
    dwSectNum = pFileHdr->NumberOfSections; //获得 PE 文件中的节的数量

    for(i = 0; i < dwSectNum; i ++, pSectionHdr ++ ) //对每个节进行处理
    {
        psrcBuffer = pBinFile + pSectionHdr->PointerToRawData;
        pdstBuffer = pHCXFile + pSectionHdr->VirtualAddress;
        if(0 == pSectionHdr->PointerToRawData) //节内容为空, 可能是一个.BSS 节, 即
        未初始化变量节。我们只需要在目标文件中保留对应尺寸即可。
        {
            memset(pdstBuffer, 0, pSectionHdr->SizeOfRawData);
        }
        else //是一个常规节, 需要进行重定位操作
        {
            memcpy(pdstBuffer, psrcBuffer, pSectionHdr->SizeOfRawData);
        }
        dwTotalLength = pSectionHdr->VirtualAddress + pSectionHdr->SizeOfRawData;
    }
    dwTotalLength = ROUND_TO_16(dwTotalLength); //目标文件以 16 字节对齐
    pdstBuffer = pHCXFile + dwTotalLength;
    bResult = TRUE;

__TERMINAL:
.....
}

```

这段代码比较简单, 就是使用 PE 文件的几个头, 对源文件进行分析。然后针对每个节, 从源文件中读取其内容, 然后写入目标文件中 `VirtualAddress` 偏移处。这样的结果就是, 目标文件的布局, 与 PE 文件被加载到内存后的布局保持一致。

14.3.3 append 工具

`append` 工具是一个二进制模块合并工具。为了引导的方便, 我们把 Hello China PC 版的几个模块, 比如 `realinit.bin`、`minker.bin`、`master.bin` 等, 合并成一个内核映像文件。这个工具读取两个分离的模块, 然后把第二个模块追加到第一个模块后面。在追加的时候, 可以根据指定的对齐方式进行追加。这样一次只能合并两个模块, 如果有多个模块需要合并, 则必须多次使用这个工具。

下面是一个使用实例:

```
[/bin/ntfs/batch.bat]
```



```
append -s realinit.bin -a miniker.bin -b 2000 -o image_1.bin
append -s image_1.bin -a master.bin -b 12000 -o image_2.bin
ren image_2.bin hcimage.bin
del image_1.bin
```

第一个 `append` 命令，用于把 `realinit.bin` 和 `miniker.bin` 合并到一起，合并后的名字是 `image_1.bin`。在合并的时候，指明了第二个模块（`miniker.bin`）要放在相对第一个模块的 `0x2000` 开始处。

第二个命令，则是把 `master.bin` 模块追加在了 `image_1.bin` 上面，追加后的结果是 `image_2.bin`。第三条命令则把 `image_2.bin` 改名为 `hcimage.bin`，这就是内核映像文件。

第四条命令删除中间过程生成的 `image_1.bin` 文件。

`append` 这个工具比较简单，且只会在内核开发中用到，建议读者只了解其功能即可。如要用这个工具，也建议直接使用 `bin` 目录下提供的可执行文件。

14.3.4 vfmaker 工具

`vfmaker` 工具用于生成一个虚拟软盘文件。这个工具读取当前目录（即该工具所在目录）下的 `bootsect.bin`、`realinit.bin`、`miniker.bin` 和 `master.bin` 模块，按照一定的布局，把这些模块重新合并到一个文件中，这个文件即是虚拟软盘文件。

生成虚拟软盘文件后，即可用这个虚拟软盘启动虚拟机了。

如果要对内核的各个部分进行修改，则只需要把修改后的二进制模块（`.bin` 文件）放在这个目录下，然后直接运行 `vfmaker`，即可重新生成虚拟软盘文件。这时候使用最新的虚拟软盘文件启动虚拟机，即可看到修改后的运行结果。

再说明一下，`bootsect.bin`、`realinit.bin`、`miniker.bin` 等三个模块都是使用汇编语言编写的，因此只要使用 `NASM` 工具即可直接生成对应的二进制模块。但是对于 `master.bin` 模块，则需要使用 `process` 工具进行预处理。

14.3.5 dumpf32 和 mkntfsbs 工具

这两个工具的功能比较简单，就是采用预置引导法的思想，读取当前分区信息，写入引导扇区（`bootsect.dos` 文件）内。其中 `dumpf32` 是针对 `FAT32` 分区的，`mkntfsbs` 是针对 `NTFS` 分区的。但是其实现过程有点复杂，需要分析 `FAT32` 和 `NTFS` 文件系统的信息，涉及文件系统的内容。在实现 `mkntfsbs` 工具的时候，实际上是把 `Hello China` 的 `NTFS` 文件程序驱动源代码直接拿过来使用了。因此这两个工具的具体实现过程，可参考第 12 章。

这两个工具的使用也比较简单，首先根据分区类型选择使用 `dumpf32` 或 `mkntfsbs`，然后把 `bootsect.dos` 和工具放在同一个目录下，运行工具即可。需要注意的是，`bootsect.dos` 文件也是与文件系统相关的，由不同的源代码编译而成。这些源代码都位于 `[/kernel/arch/sysinit]` 目录下。

附录

附录 A 关于操作系统开发的两篇博文

A.1 操作系统开发过程应遵循的一些原则

如何衡量一个操作系统是否成功

在讨论如何衡量一个操作系统是否成功之前，首先必须明确，怎样的系统软件才算作是一个操作系统？并不是所有的系统软件都是操作系统，我认为，一个完整的操作系统，必须具备下列功能（或特征）：

1. 基于一种或多种硬件平台（或硬件体系架构），能够成功地启动这个硬件计算机平台，并能够对硬件平台的基本资源进行管理。这里的基本资源，至少包括 CPU、内存以及键盘和显示器等输入输出设备。

2. 提供一个人机接口，比如一个字符界面的 Shell 或一个图形交互界面，用户能够通过这个人机接口直接操作硬件设备。

3. 提供一个应用编程接口（API），程序员能够采用一种或多种计算机语言设计出针对该操作系统的软件程序，能够完成某些特定的功能。当然，软件开发所需的开发环境和开发语言，可以是基于其他操作系统的，不一定非得是本操作系统提供的环境。

通俗地讲，能够启动计算机，并能够对硬件资源进行管理和应用的系统软件，才能称为操作系统。按照这个定义，一些只能启动计算机、显示一些特定内容的程序片断，不能算作操作系统，因为它无法提供人机交互接口，也无法提供一个应用程序运行平台。比如第一个版本的 Linux（似乎是 Linux 0.0），其全部功能就是把 CPU 切换到保护模式，在屏幕上输出一连串的 A 和 B，然后进入死循环。按照上面的定义，这不算是操作系统。再进一步，一些软件片断除具备启动计算机的功能外，还提供了基本的键盘/显示器驱动代码，用户可以通过键盘输入一些字符，然后显示在屏幕上。这也不能算是一个操作系统，因为它没有提供一个应用编程接口，无法开发出满足特定需要的应用程序。

需要强调的是，这里的应用编程接口，不一定非得以中断或陷阱调用的方式提供。比如，操作系统以头文件的方式提供一些功能函数调用，应用程序部分直接与操作系统核心代码进行编译链接，组成一个整体的模块，这种实现机制也可认为是满足上述第三条的。而且在实际开发中，很多嵌入式操作系统就是采取这种方式实现系统调用的。这样的好处是，应用程序代码直接调用操作系统功能代码，无需经过描述符切换、上下文切换等额外工作，效率非常高。但缺点也很明显，就是应用程序部分代码与操作系统核心代码无法有机分离，必须统一编译链接。诚然，以中断或陷阱方式提供的 API 接口，可以使得应用程序与操作系统完全独立，充分实现模块化设计思想，是一种更好的方式。

然后，我们再讨论什么样的操作系统算是一个成功的操作系统。满足上述条件的操作系统很多很多，一个计算机专业的学生，在半年的时间内就可写出一个操作系统。但是大多数操作系统并不是成功的操作系统，这些不算成功的操作系统，其最主要意义，可能就是锻炼了操作系统开发者的编程能力，满足了操作系统开发者完成一个操作系统内核的心愿。但一个成功的操作系统的意义就远不止于此了，成功的操作系统会被广泛使用，产生巨大的经济效益。比如 Windows 操作系统，不论你喜欢还是不喜欢，它都大大地拉近了人与计算机的距离，使得计算机成为一种最重要的生产工具，产生的经济效益是难以估量的。并不是每个成功的操作系统，都必须像 Windows 那样有影响力，我认为，一个操作系统能够满足下列几条要求，就可称为一个成功的操作系统：

1. 提供一组清晰且功能完备的应用程序编程接口（API），并有一个与之配套的应用开发环境，可以完成设备驱动程序、应用程序的开发。开发难度维持在平均软件开发难度以下，比如，一个普通技能水平的程序员，就可以在这个开发环境下，开发出针对这个操作系统的设备驱动程序或应用程序。

2. 有一组与之配套的常规硬件驱动程序，能够完成大多数日常功能。比如具备常见的网卡驱动程序、音频/视频驱动程序、USB 总线驱动和 USB 存储设备驱动程序等。有了这些常规硬件驱动的支撑，该操作系统就可完成大多数的常用功能，比如上网、音频/视频播放等。

3. 有一些与之配套的常用应用程序，形成一个封闭的应用生态环境。比如浏览器、邮件客户端、通信录、即时消息客户端、文字处理软件等，用户可通过这些软件完成常规的任务。

上述可归纳为一个操作系统生态链，这个生态链包含硬件支持、应用程序支持、开发环境支持等。只有具备了一个相对完整的生态链的操作系统，才算作一个成功的操作系统。因为它已经具备了能够产生经济效益的基础条件。

再强调一点，这里讲的“成功的操作系统”，不一定是一个商业上成功的操作系统。很多满足上述条件的操作系统，根据这里的限定，算作是一个成功的操作系统，但是在商业上却不一定成功。但是反过来，一个商业上成功的操作系统，必然是一个满足上述限定条件的“成功操作系统”。本文中的成功，着重强调操作系统的生态环境成功。只要具备了一个完善的生态环境，在商业上成功的可能性就大大增加了。

那么，要开发出一个成功的操作系统，是否有一些基本原则可遵循呢？我认为有的，因为我们的目标非常简单和明确，那就是通过合理设计操作系统，使之能够以自己为核心，形成一个完整的操作系统生态链。任何事情，只要目标明确，原则和策略就好定了。当然，操作系统开发是一个复杂的系统工程，工作量巨大，其原则决不是一篇文章能够说清楚的。下面列举了一些我认为非常重要的原则，希望起到抛砖引玉的作用，供朋友们评判。

操作系统要有明确的定位和特色。选定某一场景或应用范围，在这个既定范围内深入耕耘。所有成功的操作系统，都有其明确的定位和特点。比如 Windows 操作系统，主要是面向大众应用、面向个人计算机，因此其在易用性和用户感知上有良好建树，这是其成功的最主要因素。Linux 则偏重于代码开源、高效率，因此其得到了更多的硬件平台的支持，同时在性能要求较高、成本较低的服务器领域得到广泛应用。由于其源代码开放的特点，Linux 还被移植到嵌入式领域，在嵌入式领域也建立了一个庞大完备的操作系统生态链。目前比较流行的 Android 操作系统，则明确定位于个人移动终端领域，针对这个领域的应用特点和需求，在用户交互（触摸屏输入、简洁图形输出等）、尺寸受限的屏幕显示、移动通信特性

(语音功能、短信功能、邮件功能等)等方面做得很优秀,因而得到广泛应用。要开发一个全新的操作系统,必须选定一个全新的应用场景,或者对已有应用场景进行进一步分析和抽象,做出更明显的特色,方能成功。如果特征不明显,或者与已有成功操作系统的应用领域完全重合又没有更吸引人的特色,则很难成功,因为现有操作系统已经“先入为主”了。

操作系统关键组件一定要与操作系统核心紧密耦合,核心模块尽量不要独立。这个原则可能与我们的常规印象有冲突。在我们接受的教育和日常秉承的开发理念中,软件模块尽量独立,尽量不依赖其他模块,即使模块之间有依赖关系,也必须定义一个良好的接口进行连接。但是在操作系统开发领域,我认为要想成功,操作系统关键模块与操作系统核心之间最好形成紧密绑定关系。这里的操作系统关键模块,指的是 GUI 模块、文件系统、网络功能模块、系统调用 API、支撑库等。这些模块之间不应形成紧密绑定关系,但是这些模块必须与操作系统核心紧密耦合。换言之,为某一特定操作系统开发的功能模块,不能被轻易地拿到别的操作系统之上。这样的原则,主要是为了充分聚集不同模块之间的合力,组成一个完整有机的独特操作系统。比如 Linux,虽然其秉承开源开放的原则,但是其文件系统功能、网络功能、系统调用 API 等,都是 Linux 本身特定的,不能被其他操作系统直接移植。正是因为这样的特点,使得整个操作系统不至于分散,达到“一荣共荣、一辱共辱”的目的。再举一个相反的例子,在低端嵌入式领域应用比较广泛的 ucOS,一直是一个操作系统核心,始终未形成一个完整的操作系统。这主要是因为,以其为基础的各个操作系统重要模块,比如 TCP/IP 协议栈、GUI、文件系统等,在设计的时候首先想到的是可移植,于是定义了尽量少的操作系统核心调用接口,尽量多的功能在模块内部实现,而不依赖于操作系统核心。这样这些功能模块很容易被移植到别的操作系统上,无法发挥“外围模块助力操作系统核心”的目的。这种朝秦暮楚的设计理念,大大阻碍了操作系统生态链的生成。最后再强调一下,这里的模块与操作系统核心的紧密耦合理念,应该仅仅局限于操作系统关键模块与操作系统核心之间。其他功能模块或实体之间,还是应该秉承独立、模块化原则。比如驱动程序与操作系统之间、应用程序与操作系统之间,必须以清晰独立的模块划分原则进行设计,并定义接口,否则会大大阻碍操作系统生态链的生成。

简洁明了的应用编程接口(API)非常重要。一个成功的操作系统,必须有一个完备的生态链与之配套。显然,打造这个生态链的最主要工具,就是操作系统提供的 API 接口。所有设备驱动程序、应用程序,都必须调用操作系统 API 接口完成特定功能。因此,设计良好、易于使用、功能强大的 API 接口,对操作系统生态链的建设非常关键。在设计 API 接口的时候,尽量保持接口的简洁,每一个 API 功能调用,完成一个单一的功能,确保其功能清晰简洁,不要多个功能使用同一个 API 函数。同时,对于 API 的参数个数和每个参数的取值范围,尽量简缩和明确。对于 API 函数的返回值,也应明确定义,不要产生歧义。同时,如果可能,尽量与现有流行操作系统提供的 API 函数语义保持一致,这样可大大降低程序员的学习成本。API 定义清楚之后,尽量不要改变,如果要改变,也要以扩展参数的形式进行改变,不要改变原有参数和返回值的含义。最后,一定要有一份清晰的文档,对 API 的原型和功能进行说明。这份文档非常关键,是程序员所需的最核心开发资料。

建立一个简便、高效的应用开发环境。定义和实现了 API 之后,必须建立一个与之配套的应用程序开发环境,供程序员开发与之配套的应用程序。需要说明的是,这个应用开发环境不一定要从头重新开发,完全可以借鉴现有的应用程序开发环境。比如,可以在现有开发



环境中增加插件，使得其支持该操作系统。Android 就是一个很好的例子，它没有从头开始建立一个开发环境，而是利用了被广泛应用的 Eclipse 开发环境。这样不但大大降低开发工作量，也可充分继承现有开发环境所积累的开发经验和开发资源。很多情况下，程序员是根据开发环境选择开发的目标操作系统的，而不是根据操作系统选择开发环境。比如同样是移动开发，熟悉 Java 和 Eclipse 的程序员，可能就直接选择了 Android 作为首选目标操作系统，而熟悉 C/C++ 语言的程序员，会首选 iOS 作为目标开发环境。虽然 iOS 使用 Objective C 语言作为其开发语言，但是该语言与 C/C++ 更加接近。因此，选择一个流行的、拥有广泛开发者的开发环境，对操作系统本身生态链的建设非常有价值。最后，开发环境（或者依赖于某个成熟开发环境的插件）必须与第一个操作系统版本一起发布，所谓“操作系统未动，开发环境先行”。而且一旦发布，尽量保持不变。这样做的目的，是使程序员尽快熟悉最新操作系统的开发方法，给程序员足够的时间去学习和掌握开发过程。

采用模块化设计，确保操作系统具有最大可能的扩展性。这里包含两个层面的意思：设备驱动程序层面和应用程序层面。在设备驱动程序层面，要提供一个明确定义、功能完备的设备驱动程序调用接口，设备驱动程序通过这一组接口，调用操作系统的特定服务。这样就可使得设备驱动程序作为独立于操作系统核心的独立模块，而无需与操作系统一起进行编译和链接。显然，这样的结构，可使得设备驱动程序的开发过程非常独立和容易。与之相反的做法是，操作系统不提供设备驱动程序调用接口，或者即使提供，也是以函数的形式提供，设备驱动程序开发的时候，必须包含对应的头文件和库文件，并与操作系统核心一起连接成一个大的系统模块。显然，这种设计方法把操作系统核心和设备驱动程序绑定在了一起，不利于设备驱动程序的单独设计。同样的，对于应用程序开发，也应该定义一个清晰的开发接口，供应用程序调用。同时定义和实现一个应用程序加载机制，使得应用程序在开发过程中无需考虑加载地址等因素，也无需考虑操作系统核心的实现机制，只需要专注于应用程序本身的功能即可。这样的结果是，任何一个应用程序，都是独立于操作系统的模块，这个模块可存放在外部存储介质上，供操作系统“按需加载”。主流的成功操作系统，大都是按照这样的原则进行设计。比如 Windows 和 Linux，都实现了可加载模块的功能，同时清晰定义了设备驱动程序和应用程序能够调用的接口函数。在移动领域流行的 Android 和 iOS 操作系统，也是按照这种方式设计和实现，取得了良好的成功。如果没有这种模块化的设计机制，Apple 应用商店和 Android 应用商店就无从谈起。最后强调一下，这里的模块化原则，与前面所说的操作系统关键模块之间紧密耦合原则不冲突。前面讲的紧密耦合，是操作系统核心模块之间的紧密耦合，这里讲的模块化分离原则，是操作系统核心部分与外围部分（驱动程序、应用程序等）之间的设计原则。

最大程度的包容设计，尽量与现有操作系统兼容和共存。所谓有容乃大，尽量与现有操作系统或现有的应用软件兼容，是操作系统开发的一个重要原则。这样做，可大大加快操作系统生态链的建设。或许有些不可思议，作为最贴近硬件的系统软件，操作系统之间能兼容吗？答案是肯定的，但是这种兼容可能不像应用程序兼容操作系统那样明显。举例来说，比如新开发的操作系统是针对 x86 平台的，则尽量支持 FAT32/NTFS 等文件系统，同时其启动过程，也建议充分利用 Windows 操作系统的启动机制，提供启动菜单让用户选择。这些动作，就是为兼容 Windows 操作系统而做的。这样做，可使得用户在安装或使用新操作系统的时候，无需重新格式化硬盘，也无需销毁已有的操作系统，而只要像安装普通软件那样安

装到已有硬盘上就可以了。另外一种层面的兼容，是 API 接口的兼容。开发出与现有操作系统完全一样的 API 接口，可能性比较小，而且这样做也可能侵犯知识产权。但是与现有操作系统的 API 接口尽量保持一致，比如保持语义、API 接口参数类型、错误处理机制等的一致，可显著降低程序员的学习成本，大大加快应用程序的开发速度。与现有操作系统的可执行文件格式保持一致，也是一种兼容行为。比如，新开发的操作系统支持 Windows 的 PE 格式执行文件，则可大大增加可用于应用程序开发的开发环境数量，有利于生态链的建设。兼容现有的可执行文件格式，某些基于已有操作系统比如 Windows 开发的应用程序，甚至可以在新操作系统上直接运行。

操作系统开发是一个复杂的系统工程，应该遵循的原则或策略，远不止上述内容。而且上述原则，也是作者在操作系统开发中的一些体会和理解，不一定适合所有操作系统的开发。在这里写出来，主要是希望起一个抛砖引玉的作用，希望能够以此为切入点，激发更多的讨论和共享，为操作系统开发贡献一份力量。

A.2 对操作系统开发的一些相关问题的思考

在操作系统开发过程中，与操作系统相关的一些问题尤其敏感。下面这些内容，都是一些典型的问题或观点，与朋友或同事有过讨论之后，形成的一些个人结论或想法。只是一家之言，不免有片面之处，欢迎朋友们批评讨论。

对未来操作系统发展趋势的思考

我认为，操作系统正朝着按应用场景细分的方向发展，即针对每种应用场景，或某个特定的用户群，会有一个或多个与之适应的操作系统。比如，以前的操作系统，大致可分为桌面操作系统、服务器操作系统和嵌入式操作系统等三个大类。Windows、Linux 是桌面操作系统的典型代表，UNIX 操作系统在服务器（或大型机）领域一家独大，嵌入式领域，则存在 pSOS、VxWorks、ucOS 等操作系统。而到了当前的移动互联网时代，智能移动终端这个应用场景出现后，又催生了广泛应用于智能终端上的 Android 操作系统、Apple iOS 操作系统等。随着云计算的兴起，云操作系统又有流行的趋势。可以看出，操作系统的类别（或种类）并不是一成不变的，而是随着应用的不断变化和演进，会有全新的操作系统开发出来，以适应这些应用，其总体呈现出一种按照应用场景进行细分的趋势。

随着移动互联网的不断发展成熟，会逐渐催生出更多的应用场景，比如家庭网络、物联网等。由于体系结构的限制，传统的操作系统很可能无法适应这些新兴场景的需求，因此又会催生出一批更新的操作系统。

本质上，这是由于人类的个性化需求不断增加决定的，与汽车等传统消费品的发展轨迹类似。最开始的时候，汽车型号单一，产量供不应求，人们的需求不存在个性化，只要有一辆汽车就行。这时候汽车制造企业的运营模式，是典型的存货型运营模式，即按照有限的几种型号，生产大量的汽车并库存，然后投放市场。随着汽车市场的饱和，人们的需求已不局限于有一辆汽车，而是要有一辆个性化的汽车，这样就催生了各种各样的汽车品类，比如 SUV、商务车、跑车等。为满足个性化需求，一种汽车型号（或平台）已经不能满足需要，汽车制造商不得不推出不同的基础汽车平台，这些不同平台之间已经不能相互兼容。计算机发展轨迹与此类似。以前，人们只要有一台计算机就可以了，没有太多个性化需求（当然，个性化的硬件配置和个性化的软件，不能算本质的个性化需求，因为这些个性化特征，都是



由统一的计算机平台满足的)。这个时候,计算机生产厂家只要生产一种计算机——IBM PC 兼容机,即可满足客户需求。相应的,只要有一种操作系统,理论上就可满足所有人的需求。随着计算机的普及和网络技术的发展,一台计算机已经不能满足个人需求,人们不但希望能够在家里或办公室里使用计算机,更希望在任何时候都能够使用计算机。这样就催生了 PDA、平板电脑、智能手机等设备的诞生。原来的操作系统已经不能适应这些新兴设备,于是新的操作系统应运而生。

因此,操作系统随应用细分,以适应人们的个性化需求,必然是一种趋势。个性化的极限情况是,每个人都有一台独特的、适应自己的计算机,对应一个独特的、专门满足这个个体的操作系统。即在极限情况下,操作系统的数量,应该与人的数量相同。这符合经济学原理,因为只有完全满足每个人的独特个性化需求,才能挖掘出全部的消费者剩余,从而使得计算机厂商的受益最大。

当前虽然已经有很多成熟的操作系统,但离真正的计算机个性化需求满足,还有非常大的距离。操作系统的数量,必然会以越来越快的速度增加。

在当前各类操作系统已相对成熟的环境下,开发操作系统是否有必要?

我认为非常有必要。根据上面的分析,操作系统会越来越呈现出应用场景细分的趋势,一个或几个通用的操作系统,已经不能覆盖所有的场景需求。这种情况下,新的操作系统需求就呈现出来。这时候如果能够提前发现这种新的应用场景,并及时开发出对应的操作系统,不论经济效益还是企业商机,都是非常大的。比如 Android,其开发人员就是看到了移动互联网时代智能手机会得到广泛应用,而传统的操作系统又无法适应这种应用,于是才决定投入开发的。当然,这其中可能有其他因素,但是选定应用场景,并持续投入开发,是其成功的主要因素之一。

但是不能盲目开发,一定要选择一个应用场景,针对场景的需求,做定制性质的开发。这里的难点是如何识别出应用场景,而不是操作系统开发本身。这就需要靠开发组织的业务嗅觉能力了。我个人认为,云计算终端可能是一个未来应用空间巨大的新场景。云终端的应用有其自身独特的地方,比如需要有很强的网络能力,能够支持各种网络接入技术,同时需要有较强的图形处理能力。而且其软件部分尺寸不宜过大,在必要时能够很快进行重新安装,而不影响客户使用。还要有很强的被管理能力,能够按照维护指令,做一些升级、打补丁等动作,甚至重新安装。而且还需要考虑用户认证、通信加密等功能。显然,已有的操作系统不能完全满足这种需要,开发一种最新操作系统的需求必然会出现。

再举一个例子,比如家庭网络应用中的家庭网关(HG, Home Gateway)。家庭网关需要支持多种多样的无线和有线接入技术,需要与各种各样的家用电器连接,比如电冰箱、空调、家庭电脑、电视机、微波炉等,有时候甚至需要与门铃、门锁、窗帘等完成连接。这需要非常复杂的数据处理能力和通信能力,同时要高度安全、高可靠、高效率。还有一些其他的需求,比如人脸识别、生物认证技术等。在已有操作系统上增加这些特性,其复杂度可能会比重新开发一个操作系统还要大。因此针对这种场景,开发一个专门针对家庭网关的操作系统,是非常有必要的。

还有很多其他的场景,在此不一一列举。总之,随着应用场景的细分,硬件的个性化,操作系统开发需求不但不会消失,而且会以越来越强劲的势头凸现出来。

什么样的公司适合开发自己的操作系统

我认为，直接面向终端用户的 IT 公司，都可以通过开发自己的操作系统来增强其竞争力。比如提供互联网服务的 ISP，提供通信服务的运营商（Operator），销售终端产品的终端供应商，甚至一些非 IT 公司，比如汽车制造商，也可以通过开发自己的操作系统来增强核心竞争力。一个原则就是，只要直接面向终端用户，为终端用户提供服务或产品，都有潜在的操作系统开发需求。主要是因为，操作系统是业务终端的最核心软件（也是最核心部件），只要控制了操作系统，就控制了业务终端，进而达到保持用户、增强用户忠诚度的目的。同时，以一个自有产权的操作系统为基础，可以派生出非常多的终端类型，来满足各种各样的业务需求。这样在竞争中，企业就有了主动性，通过不断的业务创新，使得企业永远位于产业链的前端，做行业的领导者而不是跟随者。

以苹果公司为例，正是由于其拥有完全自主知识产权的 iOS 操作系统，才使得其在产品推陈出新、更新换代的过程中始终保持领先地位，通过重复使用这个操作系统，开发出各种各样的新颖产品。假设其没有自主知识产权的操作系统，而利用第三方的操作系统，那就受限于操作系统本身更新换代的影响，很难及时推出有差异化的产品。

总之，在用户直接接触的终端领域，操作系统是最高的战略高地。只要占领了这个高地，就意味着建立了在整个领域内的王者地位。但是，操作系统的开发也不能盲目进行，必须找到一个符合自身整体战略规划的应用领域，针对这个领域进行开发。比如 Alibaba 公司，就针对云计算自主定制了自己的操作系统。我认为这是一个非常明智的举措。

最后再着重说明一下，开发自己的操作系统，并不意味着一定要自主开发，也可以选择在有开源操作系统基础上进行定制。比如，现在很多互联网服务提供商，就是在 Android 基础上开发定制自己的操作系统的。这种操作方式，不能算是完全的自主开发，可称为自主定制。相对自主开发，自主定制方便快捷，投入少，而且短期内的目标都可达到，不失为一种便捷的措施。但是从长远来看，这种方式的竞争力远远不如自主开发操作系统的竞争力强。因为自主定制的操作系统，在大部分功能上，很难赶上或超过其基础操作系统的开发进度和更新速度。即使定制厂商不跟随基础操作系统的主版本计划，自己完全建立一套从内核到应用的开发流水线，也会因为跟不上硬件平台的变动，而最终落伍。除非厂商有足够的实力，既能够跟上硬件平台的更新换代，又能够完全掌握基础操作系统代码。而在这种情况下，大部分厂商一开始就会选择自主开发，而不是自主定制。因为前者的竞争优势远远大于后者。

操作系统开发难度是否真的很大

纯粹从技术上说，相对二十世纪，当前操作系统开发的难度已大大降低，甚至低于很多应用软件开发难度。主要有以下一些原因。

首先，目前存在很多开源的操作系统，可供开发人员参考。虽然开发的目标操作系统的架构和核心功能与现有操作系统不一致，但是一些关键的机制，比如线程同步、内存管理算法等，很大程度上都是相通的，可以参考借鉴已有操作系统的实现思路。

其次，不存在人才壁垒。计算机行业中有非常多的系统软件开发人才，这些人才的水平和经验，足以支撑操作系统的有效开发。

再次，当前已有非常多的功能模块代码，可直接在操作系统开发过程中引用。比如图形库，当前有很多的开源图形库可供直接应用。如果说以前开发操作系统是从零开始的话，现在开发操作系统，则是站在巨人的肩膀上进行开发。除非你希望体验一下从无到有的整个过



程，否则没有必要完全重写所有模块。

最后，也是最重要的，操作系统过程中需要的技术和知识并不是非常复杂。比如操作系统核心模块的开发，几乎不会用到复杂的数学推导和运算，只要有最基本的数据结构知识和硬件知识就可以胜任。而在一些应用软件开发过程中，比如 GIS、图形处理软件，需要有非常丰富的数学知识，比如复杂的矩阵运算、高阶偏微分方程等，这对程序员的要求非常高。相比之下，操作系统的开发难度比复杂应用软件的开发生难度低得多。

总之，操作系统开发没有想象的那么难。人们之所以一听到操作系统开发，就认为非常难，甚至望而却步，我认为很大程度上是心理作用。因为我们从未成功开发出一个操作系统，不知道操作系统的开发难度如何，于是会产生一种对未知事物的畏惧。

打造出一个广泛使用的流行操作系统的难度，在于建立一个完善的操作系统生态环境。这包括操作系统本身，与操作系统配套使用的开发工具，与之配套使用的浏览器、多媒体播放器等软件，为其定制的各种硬件驱动程序，以及支持它的众多硬件平台。这不是一个开发团队或一个公司能够独立做到的，必须借助于整个行业的力量，包括硬件设备提供商、应用软件开发商、系统软件开发商等的通力合作才能完成。而且整个生态环境的成熟，需要很长时间，有时会超过十几年时间的培育。再强调一下，这里说的生态链，是针对一个通用操作系统来说的，比如 Linux，Android 等。

当然，这并不是说操作系统开发就没有任何机会了，相反，开发的机会还会越来越多。之所以这样说，就是基于先前论述的应用场景细分趋势。在操作系统应用场景细分的情况下，操作系统的生态链范围会大大缩小。一个优秀的公司，以一己之力就可以打造一个完整的生态环境。

怎样的操作系统才能算是独立开发的操作系统

这个问题可能比较敏感，而且见仁见智，这里只是说一下我个人的理解。我认为，一个独立开发的操作系统，下列各模块中，至少要有一个是完全独立开发的（即不重用任何现有模块，完全是独立编码）：

操作系统内核，这里的内核，包括基本的操作系统服务，比如进程/线程模型、内存管理机制、设备管理机制、核心设备驱动程序等。

图形用户接口（GUI），可以直接借用现有的图形库，但是 GUI 不仅仅是一个图形库，还包含了用户交互机制、图形资源管理等相对比较复杂的内容。这些内容需要自行编写。

全新的应用编程接口和开发工具。比如，在原有操作系统核心基础上，增加的 API 调用数量，要超过原有内核体系的 API 数量，这些 API 组合起来，提供一种全新的应用解决方案或应用场景。同时要开发一个全新的应用程序开发环境，并能与操作系统和最新的 API 有机集成，提供面向某个应用场景的整体解决方案。

比如 Android，虽然其操作系统核心是借用的 Linux 内核，但是其 GUI 却是完全重新编写，且提供了一个基于 Java 语言的全新开发环境，上述列举的模块中，有至少两个是全新编写的，因此 Android 属于一个独立开发的操作系统。当前市面上有很多号称是自主开发的操作系统，实际上只是把 Android 的 GUI 模块进行了部分修改，添加了一些个性化的东西，同时有针对性地增加了一些应用。我认为，这不应该算是自主开发的操作系统，而应看作自主定制的操作系统的。

这里并不是否定自主定制操作系统的行为，而是从技术上，试图界定自主开发和自主定

制。实际上，自主定制是一种非常明智的行为，可以使企业快速推出其个性化产品，满足市场需求，是企业积极适应市场需求、积极参与竞争的表现。相反，如果企业纯粹去追求自主开发，在没有明确市场需求和市场定位的情况下，不计成本地投入开发一种全新的操作系统，其实是一种愚蠢的行为。现有的东西能够满足需求，为什么不拿来直接用呢？模块重用原则一向是软件行业推崇的基本原则。

对 x86 平台在操作系统开发中的作用的考虑

我个人认为，x86 硬件平台是操作系统开发过程中无法绕过的一个平台，而且对一些相对通用（这里的“相对”通用，可以理解为至少支持两种以上硬件平台的操作系统）的操作系统来说，以 x86 为最初的开发目标平台，完成 x86 平台的开发后，再向其他硬件平台移植，或许是最有效且最省事的策略。主要有以下几点原因：

首先，可充分利用已有的大量的操作系统相关代码和文档。x86 硬件平台是一个高度标准化的计算机平台，不论是其初始化和加载过程，还是常用硬件的资源配置（端口号、内存映射等），都有明确定义。没有任何其他的计算机硬件平台能够像 x86 这样完善，虽然从纯技术角度讲，x86 CPU 的体系架构不一定是最优的。正是因为这样的特点，很多系统软件爱好者开发了大量的面向 x86 平台的操作系统代码片断，放到互联网上共享。这些代码片断包含了操作系统开发过程中的方方面面，是操作系统开发过程中最宝贵的资源。借鉴这些资源和文档，可大大加速操作系统开发过程，尤其是初期的开发过程。

其次，由于 x86 硬件平台的广泛应用，可大大加快新操作系统的推广和使用速度。一旦有一个操作系统雏形，能够成功引导计算机，并能够做一些基本的操作，只要你放到网上，肯定会有很多的操作系统爱好者下载使用。这无疑会大大提升新开发操作系统的推广范围和推广速度。同时可能会收到大量的对新系统的改进建议和 bug 报告，有助于操作系统软件质量的提升。

最后，对操作系统的进一步扩展开发有重大意义。比如，你完成了操作系统核心部分的开发，然后公布 API 接口、设备驱动程序开发接口甚至源代码，会吸引很多系统软件爱好者继续开发驱动程序和应用程序。这对整个操作系统生态链的构建是非常重要的。如果操作系统是直接面向 x86 硬件平台的，这个驱动程序和应用程序进一步开发的过程，就无需模拟器的支持，得到的响应必然会更多。

总而言之，如果不是针对一个固定硬件平台做的功能有限的操作系统开发，建议以 x86 为首要开发平台，以充分发挥已有优势。如果你开发的操作系统非常封闭，只适应于固定的一种硬件平台，则无需开发 x86 平台版本。但只要是一个功能相对丰富的操作系统，适应多种硬件平台是必然的，这时候，x86 就是绕不过的门槛了。

附录 B 源代码组织结构说明

1. 源代码的目录结构

据大致统计，Hello China V1.75 版本的源代码数量大约为 4.5 万行。除硬件相关部分采用汇编语言实现外，所有其他功能都是由 C 语言实现的。在源代码中，有效汇编代码大致为 500 行，其余都是 C 语言代码。操作系统核心模块的代码经过多次优化，不论是从执行正确性上说，还是执行效率上说，都已经算是比较完善了。

下面对源代码的组成结构做简要说明,以便读者查阅。随本书一起发布的源代码,是 V1.75 版本。所有源代码都打包在一个名字是 V175 的压缩文件中,解压后即可形成层次化的代码目录结构。需要说明的是,在代码目录结构中,不仅包含扩展名是 .h 或 .cpp 的源代码,还包含了开发环境生成的工程文件、项目管理文件、编译后的二进制模块等。这样读者可以用 Visual C++ 6.0 或者 Visual Studio 2003 以上版本的开发环境直接打开,无需单独创建工程项目。

下表列出了源代码根目录下的所有文件或目录。

名 字	文件或目录	主 要 内 容
app	目录	包含了所有随 V1.75 发布的 Hello China 应用程序的源代码和二进制模块。主要有颜色选择控件示例、帮助、CPI 统计、模拟时钟等几个 GUI 程序。
bin	目录	编译后的操作系统核心模块和外围模块,以及安装程序。该目录进一步分为 NTFS、FAT32、VirtualPC 等几个子目录,每个子目录下面分别存放了对应文件系统或虚拟机的安装程序。
gui	目录	GUI 模块的所有源代码和二进制文件
kernel	目录	操作系统内核的所有源代码和二进制文件
sdk	目录	应用程序开发工具和对应的库文件,该目录进一步分为 vc60 和 vs2005 两个目录,分别存放了针对 VC 6.0 和 VS 2005 的开发工具和库文件、头文件
tools	目录	包含所有开发辅助工具的源代码和可执行文件
hcntheme.jpg	文件	一个主题图

其中每个目录中,又根据功能划分成了多个子目录,后文将对几个重点目录中的内容做进一步说明。hcntheme 是个图片(图 B-1),是作者在印度出差期间拍摄的。图中的石匠非常专注、专业,以此作为主题图片,是希望自己能像石匠雕琢石料那样认真、细致、执着地开发 Hello China。



图 B-1 专注的石匠

2. kernel 目录的主要内容

Kernel 目录下包含了操作系统内核的所有源文件和开发控制文件(由开发工具生成的工程文件、项目管理文件等)。该目录实际上是一个 Visual C++ 6.0 的工程,双击 master.dll 文件,即可将该目录下的所有文件加载到集成开发环境中,前提是安装了 VC++ 6.0。

由于内核的源代码数量比较多,为了管理上的方便,又根据功能进一步划分成了下列几个子目录:

arch 子目录:包含了与硬件平台相关的代码,比如 BIOS 调用代码、x86 CPU 相关的汇编语言代码等。该目录进一步包含了 **sysinit** 目录,这个目录下存放了引导扇区、实模式初始化代码、**miniker** 等由汇编语言写的代码。在移植 **Hello China** 到其他硬件平台的时候,只需要修改该目录下的代码即可,其他目录下的代码可不作大的改动。

(1) **drivers** 子目录:包含了 IDE 接口硬盘、键盘、鼠标等硬件设备的驱动程序源代码。若需要增加其他硬件的支持,建议在这个目录下增加对应的驱动程序。

(2) **fs** 目录:文件系统实现代码,包含 NTFS、FAT32 等两个文件系统的实现代码。

(3) **include** 目录:包含操作系统内核相关的所有头文件。操作系统内核相关的对象和数据结构、全局函数等,都是在这个目录下定义的。把所有头文件放在一个目录中,可把这个目录增加到开发环境的包含目录列表中,这样在源代码中只需要使用尖括号包含需要的头文件即可(即`#include <headfile.h>`),无需指明整个头文件的目录。

(4) **kernel** 目录:操作系统内核的所有源代码。

(5) **kthread** 目录:内核线程的实现代码,这里的内核线程,是必须随内核一起运行的线程,目前主要是一个 **idle** 线程。这个线程的优先级最低,在系统中无任何其他线程需要调度的时候,**Hello China** 将调度该线程。电源管理代码、一些低优先级的系统级任务等,可在这个线程内实现。

(6) **lib** 目录:操作系统内核开发过程中需要的支撑库代码,主要是字符串操作代码、格式化输出代码、内存复制和清零代码等。

(7) **osentry** 目录:存放操作系统初始化代码。

(8) **shell** 目录:包含字符界面 **Shell** 的实现代码,以及内嵌在字符 **Shell** 中的应用程序代码。主要有磁盘格式化程序 **fdisk**、文件系统 **fs**、系统诊断程序 **sysdiag**、硬件诊断程序 **ioctl** 等几个字符界面程序的源代码。

其他几个由开发环境生成的目录,比如 **Debug** 和 **Release** 等。

3. gui 目录的主要内容

与 **kernel** 一样,gui 目录也是一个完整的 VC 6.0 工程,所有 GUI 相关的功能都在这个目录下实现。该目录下又进一步包含了下列子目录:

(1) **ctrl** 目录:GUI 控件所在目录,所有 GUI 的控件,比如按钮、图形按钮等,都是在这个目录下实现的。如果需要增加其他 GUI 空间,也建议把源代码放在这个目录下。

(2) **draw** 目录:所有绘制相关的代码,比如绘制和填充封闭图形、贝赛尔曲线、渲染等代码,都归属在整个目录下。但由于这些功能在 V1.75 版本中尚未得到支持,因此该目录暂时为空。

(3) **include** 目录:与 **kernel** 目录一样,所有 GUI 相关的头文件都归属在这个目录下。

(4) **kapi** 目录:GUI 模块可调用的系统调用代码,这个目录中是系统调用的代理代码,具体的系统调用的实现,是在 **kernel** 模块中完成的。

(5) **kthread** 目录:支撑 GUI 模块运行的所有核心线程的源代码,主要有原始输入线程(**RAWIT**)、GUI **Shell** 线程等。实现这些线程所需要的支撑功能实现代码,比如应用程序加载功能,也放在这个目录下。



(6) picture 目录：支撑 GUI 模块的所有图片文件。

(7) syscall 目录：GUI 模块输出的系统调用的存根代码。GUI 模块本身的功能，也是通过系统调用的方式封装的，以供用户应用程序调用。

(8) video 目录：video 对象的实现目录。所谓 video 对象，指的是显示器等可以实现图形输出的硬件设备。

(9) window 目录：GUI 的窗口机制实现代码。窗口机制是 GUI 的核心，所有窗口管理相关的代码，都放在整个目录下。

(10) word 目录：文字输出实现代码，比如汉字输出、ASCII 字符输出等。后续的字体等功能，也需要把代码放在这个目录下。

其他几个目录，比如 Debug/Release 等，是由开发环境自动生成的。

附录 C 内核开发环境的搭建

C.1 概述

Hello China V1.75 基于 PC 的版本的开发环境是 Microsoft Visual C++。之所以使用 Windows 操作系统和 VC 开发环境，是考虑到相比 Linux 和 GCC 等开发工具来说，这个组合有更加广泛的使用群体，可以让更多的人参与开发。但是如果不做一番特殊处理，Visual C++ 是不适合操作系统开发的，主要原因有以下几点：

1. 缺省情况下，VC 生成的目标文件的入口地址不固定

一般情况下，Windows 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，先调用其他的一些初始化函数，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。这样的结果就是，一个可执行模块的入口地址是不可见的，这不适合 OS 的开发，因为 OS 的开发过程中，需要严格地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，会说明如何改变模块的缺省入口地址。

2. 缺省情况下，生成的目标文件的缺省加载地址不符合要求

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动态链接库（DLL），这两种文件在链接的时候，都指定了一个缺省的加载地址，一般进程地址空间的 4MB 偏移处。由于 Windows 操作系统使用了虚拟内存技术，每个进程都独占 4GB 的线性空间，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地加载到指定的地址。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 1MB，而这个模块如果按照缺省设置，则可能按照 4MB 加载地址进行链接，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 4MB 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 4MB（假设编译器把全局变量 `ulOsVersion` 放到了模块的开始处）。

而如果我们操作系统要求这个模块被加载到 1MB 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结果不一致，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链接选项来消除这个矛盾。后面会说明如何消除这种矛盾。

3. VC 生成的目标文件，增加了一个 PE 文件头

VC 生成的可执行二进制模块，在文件的开始处增加了一个 PE 文件头，而这个头的长度是可变的，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 开发的要求是直接找到模块的入口地址，通过一条跳转指令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，单独开发一个工具软件，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 OS 核心模块的加载过程中，只要把这个模块读入内存，并直接跳转到开始处执行即可。

为了进一步说明这个过程，考虑图 C-1 所示的示例。

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，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（准确地说，应该是入口地址的偏移量减 8，因为文件头距离 `JMP` 指令后一条指令的偏移是 8），如图 C-2 所示。

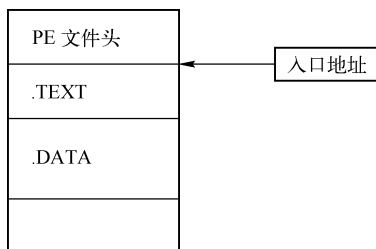


图 C-1 一个二进制目标模块

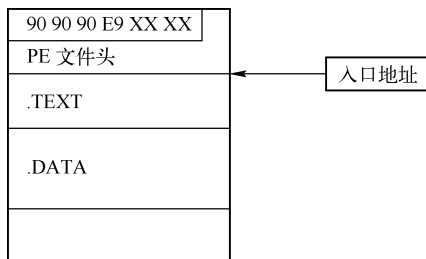


图 C-2 处理后的二进制可执行模块

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，因此会间接地跳转到入口地址处开始执行。`process` 工具就是为此而开发的，可以用这个工具来修改 PE 文件，使得 PE 文件可以直接被加载并执行。

4. 目标模块加载到内存后，需要经过处理才能运行

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 4KB 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 文件头来适当地调整节与节之间在内存中的对应关系，然后才能运行。

在 OS 的开发当中，我们要求文件在内存中的映像应该与在磁盘上的映像一致，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`)，就可以使内存中的对齐方式与硬盘中的对齐方式一致，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，

而这个节在硬盘中是不分配空间的，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，图 C-3 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像关系。

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，缺省的 Visual C++ 的编译选项是不适合操作系统开发的，必须进行修改。下面就详细讲解开发工具的设置方法。

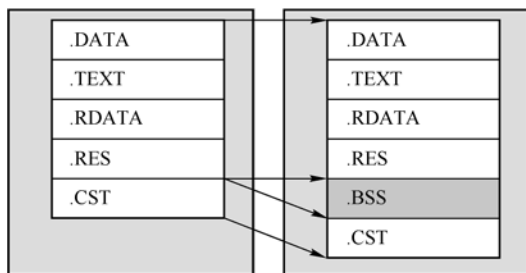


图 C-3 PE 文件的存储布局和内存布局

C.2 Microsoft Visual C++ 的设置

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链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链接选项，十分适合 OS 映像文件的开发。但其缺省配置是不适合操作系统开发的，必须进行修改。在介绍如何修改开发配置选项之前，先简单介绍一些在 OS 开发中用到的 VC 特性。

C.3 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。

1. 内嵌汇编代码

在开发操作系统时，经常在 C 或 C++ 语言中嵌入汇编代码直接操作硬件，比如，进程或线程的切换、端口输入、各类 CPU 相关的硬件系统表（比如 GDT、IDT 等）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，则不适合 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{
```



```
fild qword ptr [ebp + 0x08]
fild 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。同时为了代码的可移植性，尽量不要使用内嵌汇编语句。

2. __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，会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，有时候则要求函数在被调用的时候，堆栈框架保持明确的结构，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，下列情况需要这种修饰。

- (1) 系统调用涉及线程切换时。
- (2) 中断处理函数。
- (3) 在操作硬件系统表格，如 IA32 的 GDT/LDT/IDT 时。

C.4 搭建操作系统开发环境

在充分理解上述情况后，通过下列步骤，可以很容易地搭建一个 OS 映像文件开发

环境。

1. 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 映像，因为编译器在编译的时候，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 C-4 所示创建一个 DLL 工程。

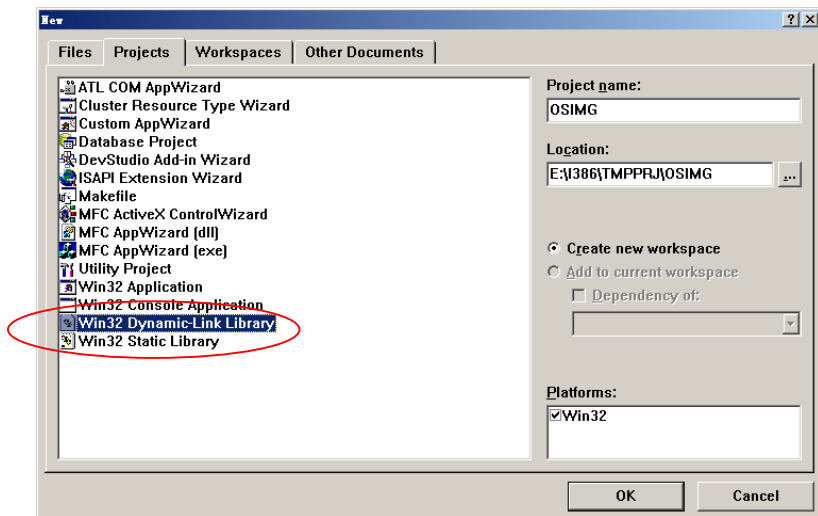


图 C-4 创建一个动态链接库工程

2. 设置项目编译与链接选项

一般情况下，需要对创建的工程设定如下编译链接选项。

(1) 对齐方式，在项目选项中，添加/ALIGN:XXXX 选项，告诉链接器，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，需要设置为与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 (1MB) 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 函数，针对动态链接库，是 DllMain 函数，或 EntryPoint 函数，等等。采用缺省的入口地址，有时候不能正确控制映像文件的行为，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，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。

上述所有的设置，请参考图 C-5。

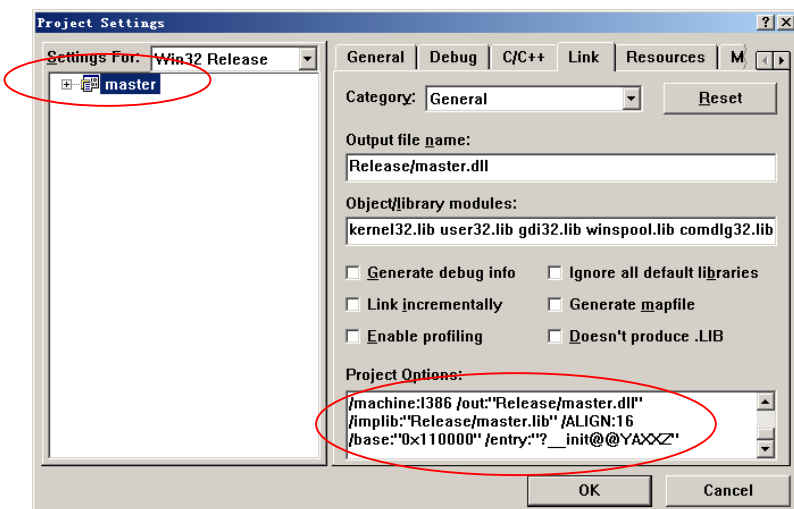


图 C-5 设置编译选项

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。

3. 对目标文件进行处理

完成上述设置，编译链接好目标文件以后，使用 process 工具对其进行处理，即可形成可直接加载的二进制模块。

4. 映像文件的加载与运行

通过上述步骤（编译、链接、处理等），可以最终得到一个可执行的 OS 映像，只要把这个 OS 映像加载到内存中，跳转到该映像的开始处就可以运行了。

C.5 操作系统核心模块开发示例

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像。这个操作系统映像非常简单，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。

1. 创建一个名字为 OSIMG_1 的 DLL 工程

如图 C-6 所示。

2. 添加一个源程序文件，并编辑实现代码

在新建的工程中，增加一个 C++源文件，并键入以下代码：

```
void ClearScreen()
{
    unsigned long ulBase = 0xb8000;
    unsigned long i      = 0;
    while(i < 80*25)
    {
        *(char*)ulBase = ' ';
        ulBase ++;
        *(char*)ulBase = 0x07;
```

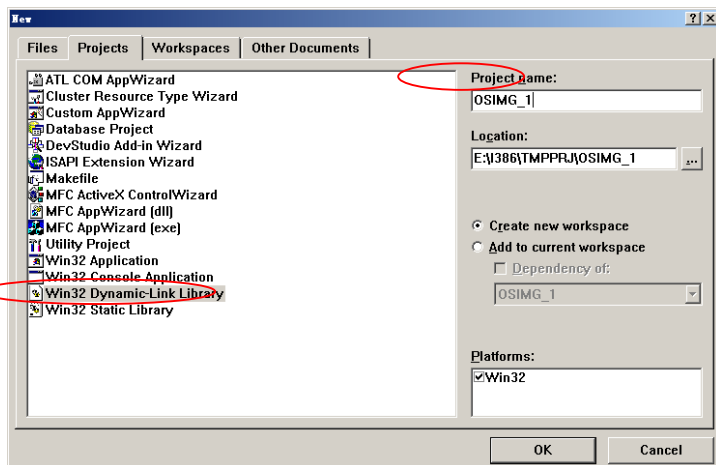


图 C-6 创建一个新的工程

```

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.
    {
    }
}

```

这段代码的功能是，调用 `ClearScreen` 函数清屏，然后打印出“ABCD.....WXY”，之后，进入死循环。在清屏幕和输出字符的时候，都是采用直接写显存的方法实现的。需要注意的是，在上述代码中，不要调用任何 C 语言库函数。

3. 设置编译链接选项，并进行编译链接

根据上面的叙述，设置下列编译和链接选项。

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`，即该程序段从 1MB+64KB 开始加载。1MB 开始的 64KB 代码，用于 `miniker.bin` 模块的加载；

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。如图 C-7 所示。

设置完后, 选择 “build->Batch Build...” 菜单, 出现如图 C-8 所示对话框。

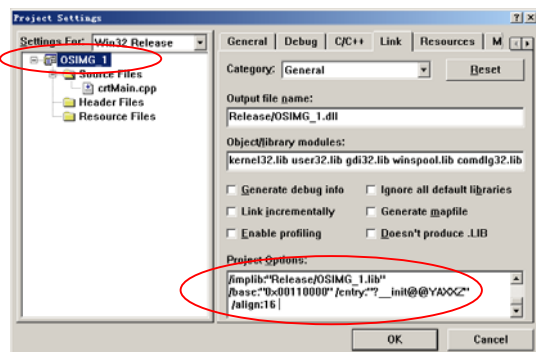


图 C-7 设置工程的编译链接选项

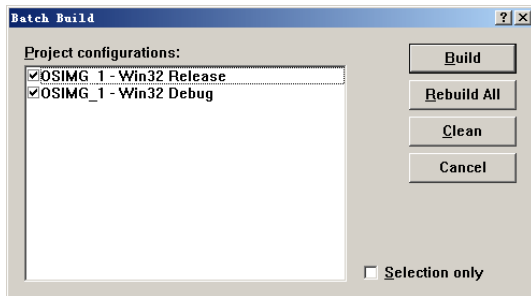


图 C-8 选择 Release 模式对工程进行编译

单击 “Rebuild All” 按钮, 编译这个 DLL 工程。

4. 处理目标文件

使用 process 工具处理生成的 DLL 文件, 并把文件重新命名为 master.bin:

```
process -i osimg_1.dll -o master.bin
```

5. 创建虚拟引导盘

经过上述步骤之后, 读者就已经创建了自己的操作系统映像了, 剩下的任务就是创建虚拟引导磁盘了。把 bootsect.bin、realinit.bin、miniker.bin 以及上述步骤生成的 master.bin, 复制到同一个目录下, 同时也把 vfmaker 工具复制到这个目录下, 运行 vfmaker, 即可生成一个 vfloppy.VFD 文件。这个文件就是虚拟引导软盘, 使用该文件启动虚拟机, 即可看到预期的运行结果。

最后需要说明的是, 采用上述方法开发出来的二进制模块, 是完全的 32 位可执行代码, 需要 CPU 运行在保护模式之下。这不是问题, 因为 realinit.bin 和 miniker.bin 等模块是首先被加载运行的, 这些模块会初始化硬件, 并把 CPU 切换到保护模式。

参 考 文 献

- [1] Intel Corporation. Intel® 64 and IA32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. 2012.
- [2] Microsoft Corporation. FAT32 File System Specification, Version 1.03. 2000.
- [3] PCI SIG. PCI Local Bus Specification, revision 2.0. 1993.
- [4] Video Electronics Standards Association. VESA BIOS Extension(VBE core) 3.0. 1998.
- [5] ANSI, NCITS. AT Attachment with Packet Interface – 7, 2005.
- [6] The Netwide Assembler development team. Online Documentation for NASM[OL]. <http://www.nasm.us/xdoc/2.10.01/html/nasmdoc0.html>.
- [7] Jeffrey Richter. Windows 核心编程[M]. 北京：机械工业出版社，2000.
- [8] MSDN, [http://msdn.microsoft.com/en-us/library/windows/desktop/hh447209\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/hh447209(v=vs.85).aspx).
- [9] Compaq Computer Corporation. Phoenix Technologies Ltd, Intel Corporation. BIOS Boot Specification, Version 1.01. 1996.
- [10] PCI special interest group. PCI BIOS Specification. 1994.
- [11] 谢煜波. 做一个支持图形界面的操作系统（上、下）[OL]. http://blog.csdn.net/xiaohan13916_830/article/details/108937. 2004.

在线互动交流平台

官方微博: <http://weibo.com/cmpjsj>

豆瓣网: <http://site.douban.com/139085/>

读者信箱: cmp_itbook@163.com

操作系统 实现之路

❧ 内容简介 ❧

本书以 Hello China 操作系统为例,详细讲解了操作系统的内核、文件系统、图形界面、设备驱动程序、SDK 和系统调用等主要功能模块的实现原理。讲解过程中不仅陈述概念,还配以详细的实现源代码对概念进行说明,达到理论联系实际的目的。书中穿插了大量的案例,读者可通过亲手操作这些案例来更加深入地理解操作系统原理。此外,本书还对操作系统的发展趋势和商业模式进行了探讨。

 读者可在 <http://blog.csdn.net/hellochina15> 下载 Hello China 源代码。

地址:北京市百万庄大街22号

邮政编码:100037

电话服务

社服务中心:010-88361066

销售一部:010-68326294

销售二部:010-88379649

读者购书热线:010-88379203

网络服务

教材网: <http://www.cmpedu.com>

机工官网: <http://www.cmpbook.com>

机工官博: <http://weibo.com/cmp1952>

封面无防伪标均为盗版

上架建议 计算机/操作系统

ISBN 978-7-111-41444-5

策划编辑 车 忱 / 封面设计



子时文化
Zishi Culture

ISBN 978-7-111-41444-5



9 787111 414445 >

定价: 79.00 元