

深入研究嵌入式操作系统的绝佳教材

一步一步写
嵌入式操作系统
—ARM编程的方法与实践

李无言 著

内 容 简 介

本书是一本介绍怎样去实际编写一款嵌入式操作系统的书，所涉及的内容包括操作系统基本原理以及这些原理在嵌入式平台中的实现方法。全书共分九章，从最基本的嵌入式编程方法开始，逐渐深入到中断管理、内存管理、设备管理、文件系统管理以及进程管理等操作系统核心部分，为读者系统地呈现了一个操作系统的全貌。

另外，本书遵循理论联系实际的基本原则，在阐述基本原理的同时，还给出非常详尽的示例代码，以及对这些代码的讲解。读者研读这些代码，不但可以进一步巩固对操作系统理论知识的理解，更可以以此为基础，去实现一个属于自己的嵌入式操作系统。也许写操作系统是很多人的梦想，或者对有些人来说根本不敢想象，希望通过学习本书，您可以超越您的想象，实现您的梦想！

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

一步一步写嵌入式操作系统：ARM 编程的方法与实践 / 李无言著. —北京：电子工业出版社，2011.1
ISBN 978-7-121-12240-8

I. ①…… II. ①李… III. ①微处理器，ARM—系统设计 IV. ①TP332

中国版本图书馆 CIP 数据核字(2010)第 218180 号

策划编辑：袁金敏

责任编辑：贾 莉

印 刷：北京机工印刷厂

装 订：三河市胜利装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编 100036

开 本：720×1000 1/16 印张：17.25

字数：274 千字

印 次：2011 年 1 月第 1 次印刷

印 数：4000 册 定价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：(010) 68279077；邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

前　　言

本书作者多年的嵌入式领域研发、管理和教学经验，促成了本书的诞生。

本书的目的

操作系统是一个古老的话题，它的出现和发展，对于计算机技术来说，意义非凡。这种发展不仅仅体现在传统的计算机中，在强调精简、高效率和实时性的嵌入式领域，操作系统也发挥着不可替代的作用。

熟悉嵌入式技术的读者朋友多少都能列举出一些典型的嵌入式操作系统，如ECOS、UCOS、Linux、VxWorks，等等，这些操作系统凭借自身的优势，在嵌入式领域的各种应用中安营扎寨，各占据了一席之地，国内基于这些嵌入式操作系统的书籍也是琳琅满目、层出不穷。

本书写的一个根本目的是能够体现出嵌入式操作系统的基本原理和结构特点，于是，本书力求从嵌入式操作系统的实现方法这一角度出发，去讲解嵌入式系统的基本原理，为大家展示一个嵌入式操作系统的全貌，不拘泥于某款具体的操作系统，不局限在操作系统上的应用程序这一范畴。

一次跟Android领军人物高焕堂先生聊天时，他的一番话让我感触颇深。他说，国外先进的开发工具、平台和操作系统就好比是武器，而中国人喜欢拿着别人给的先进武器去打仗（做应用层开发），一旦有一天我们跟外国人打起来，人家拿走我们的武器，我们就真的是一筹莫展了。

这句话很有道理，中国计算机技术整体水平的提高需要以大量自主研发的开发工具、平台架构以及操作系统为基础。不过，目前我们离这样的一个目标还相去甚远。

本书强调实践，力求能够帮助读者编写出属于自己的嵌入式操作系统。如果读者以本书为基础（或者哪怕从中得到了一丝灵感）开发出一些优秀的嵌入式操作系统，那将会是非常令人高兴的事情！

本书的特点

目前，市面上与操作系统理论相关的书有很多，与这些书相比，本书特点十分鲜明。

第一，本书的内容立足于嵌入式技术，以目前最流行的ARM体系结构为基础，为您展示出嵌入式环境下操作系统的基本原理和实现方法。这是一本学习嵌入式技术，尤其是系统级技术的首选教材。

第二，本书是以实践的方式讲述全书内容的，重视理论联系实际。操作系统涉及的每一个角落，如进程、内存、中断、文件系统、驱动程序，都有若干段代码供读者实践。实际上，读者只需要将书中的代码拼接起来，就可以构成一个结构完整的操作系统的内核。

第三，涉猎广泛。为了让读者能够全方位地理解操作系统的理论和实现方法，书中涉及了高级C语言编程、汇编语言、算法、ARM体系结构等诸多领域，分析研究了包括freeRTOS、uCOSII、u-boot、Linux、Minix等在内的操作系统和引导程序的源代码。这些必备的知识和概念都将成为您深入研究任何一款操作系统的绝佳入门内容。

第四，本书语言通俗易懂。书中在描述各种操作系统概念或原理时，力求使用通俗的语言浅显地说明问题，这与一些学术性质的操作系统读物有明显的不同。

谁适合读这本书

本书是写给那些想了解操作系统原理的人，也适合那些想要学习ARM技术的开发者。当然，对于那些致力于开发属于自己的嵌入式操作系统的读者们来说，本书是一部绝佳的入门指南。

当然，我们也希望本书的读者至少具备一些基本的C语言编程基础。除了C语言之外，如果您对微机原理之类的知识也略知一二，那么在阅读本书



的时候，将会觉得更加轻松。本书不需要读者了解嵌入式技术，因为这些知识在书中都会涉及，但如果您也知道一些关于ARM的事，那么无论是重写书中的代码还是深入理解书中所讲述的操作系统原理，都会水到渠成。

学习本书的方法

本书在知识的宽度和篇幅上做了平衡。希望展示给读者一个操作系统的全貌，保证读者能够自己动手完成操作系统的编写，而这些都需要我们在开发的过程中不断学习书中所涉猎的知识点。毫无疑问，这些内容是相当庞大的。

为了避免在书中罗列各种技术手册的细节，在本书的编写过程中，我们遵循了只对用到的知识深入讲解的原则。这样做的弊端是，读者会对很多知识点有深入的理解，但却不全面。

因此，我们推荐阅读本书的最佳方法是，以本书为主线，以其他本书中未提到的相关知识、参考资料为辅助。当然，如果读者的确是时间有限，那么单凭书中所涉猎的知识，也足以完成读者学习和实践的过程了。

另外，这里还想强调的是，一定要多多实践，实践将成为您快速掌握一门新技术的不二选择。

为了方便读者获取书中涉及的相关资料和工具，我们开设了一个网站，读者朋友可以去访问www.leeos.org以获取帮助。

最后，希望所有的读者在读完本书之后，都能够有所收获，希望每位读者都能编写出属于自己的嵌入式操作系统！

目 录

第 1 章 搭建工作环境	1
1.1 选择合适的开发环境	1
1.1.1 准备 Cygwin 开发环境	2
1.1.2 使用 Linux 开发环境	7
1.2 开发工具的使用	8
1.2.1 编译器的选择和安装	8
1.2.2 编辑器的选择和使用	10
1.3 虚拟硬件的安装和使用	13
1.3.1 SkyEye 的安装	15
1.3.2 SkyEye 的使用	15
1.4 总结	17
第 2 章 基础知识	18
2.1 使用 C 语言写第一段程序	18
2.2 用脚本链接目标文件	22
2.3 用汇编语言编写程序	25
2.4 汇编和 C 的混合编程	29
2.4.1 过程调用标准	30
2.4.2 混合编程的例子	31
2.5 Makefile	33
2.6 总结	35
第 3 章 操作系统的启动	36
3.1 启动流程	36
3.1.1 ARM 的启动过程	38
3.1.2 ARM 操作系统解读	39
3.1.3 正式开始写操作系统	46
3.1.4 让启动代码运行起来	52
3.2 MMU	56
3.2.1 页表	58
3.2.2 页权限	64
3.2.3 cache 和 write buffer	66

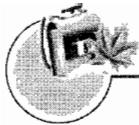
3.2.4 激活 MMU	69
3.3 GCC 内联汇编	71
3.4 总结	78
第 4 章 打印函数	79
4.1 打印函数实例	80
4.1.1 变参函数是如何工作的	81
4.1.2 亲自实现一个可变参数函数	83
4.1.3 实现打印函数中的格式转换	87
4.2 实现自己的打印函数	91
4.3 总结	99
第 5 章 中断处理	100
5.1 ARM 的中断	101
5.1.1 统一的异常和中断处理	102
5.1.2 独立的中断处理	114
5.2 简单的中断处理实例	118
5.2.1 解决异常向量表的问题	118
5.2.2 简单的中断处理代码	120
5.2.3 S3C2410 中的定时器	122
5.2.4 让中断处理程序运行起来	124
5.3 复杂的中断处理实例	126
5.3.1 提高中断的效率	126
5.3.2 中断嵌套的实现	130
5.4 更优秀的中断嵌套方法	134
5.5 总结	139
第 6 章 动态内存管理	140
6.1 伙伴算法	141
6.1.1 伙伴算法的原理	143
6.1.2 伙伴算法的实现	146
6.2 slab	169
6.2.1 使用 slab 的前提条件	170
6.2.2 slab 的组成	171
6.2.3 通过 slab 进行内存分配	176
6.2.4 内存空间的释放	177
6.2.5 slab 的销毁	178
6.3 kmalloc 函数	179





6.4 总结	183
第 7 章 框架	184
7.1 驱动程序框架	184
7.1.1 基于存储设备的实例	186
7.1.2 运行存储设备实例	192
7.2 文件系统框架	194
7.2.1 文件系统的原理	195
7.2.2 文件系统框架的实现	197
7.2.3 romfs 文件系统类型	200
7.2.4 实现 romfs 文件系统	204
7.2.5 让代码运行起来	210
7.3 总结	215
第 8 章 运行用户程序	217
8.1 二进制程序的运行方法	218
8.2 可执行文件格式	222
8.2.1 ELF 格式的组成结构	223
8.2.2 操作 ELF 格式文件的方法	226
8.2.3 运行 ELF 格式的应用程序	230
8.3 系统调用	232
8.3.1 用户和内核的运行空间	232
8.3.2 实现一个系统调用	235
8.3.3 运行系统调用程序	244
8.4 总结	246
第 9 章 进程	247
9.1 进程的实现原理	247
9.2 进程的实现	252
9.2.1 改写中断处理程序	252
9.2.2 抽象调度函数	256
9.2.3 新进程的产生	258
9.2.4 多个进程同时运行	262
9.3 总结	265
结束语	266
参考资料	267





第1章

搭建工作环境



“工欲善其事，必先利其器”。在我们开始写操作系统代码之前，花些时间来学习工具的使用其实是非常必要的。从过程上看，操作系统的开发与普通应用程序的开发并没有太大的区别。正像开发应用程序那样，要开始一个操作系统的开发，首先也必须要解决开发环境搭建的问题。

搭建开发环境归根结底是要解决四个问题：

- 在什么样的系统环境下开发？
- 使用什么样的编辑工具？
- 怎样编译程序？
- 程序如何运行？

为了能够帮助读者学习本书的内容，在本章中我们将会围绕上述四个问题展开深入的讨论，以帮助读者顺利地编写出属于自己的嵌入式操作系统。

1.1 选择合适的开发环境

每一个计算机爱好者都有自己心仪的的操作系统。有些开源软件和嵌入式技术的爱好者也许偏好 Linux，一些追求时尚和个性的朋友或许钟爱 Mac OS，但对于绝大多数读者来说，Windows 操作系统应该还是最熟悉的。



本来使用什么样的操作系统，不应该成为限制开发的理由。但这里需要说明的是，本书的所有示例代码都是在一个叫做 Gentoo 的 Linux 发行版中开发的。书中所使用的绝大多数工具在 Linux 下都有原生的支持。因此如果有可能，还是建议大家使用 Linux 操作系统来学习和编写本书的示例。

然而，要求那些不熟悉 Linux 操作系统的开发者在短时间内学会使用 Linux 也并不现实。好在 Windows 操作系统在兼容性和应用程序多样性等方面是无人能敌的，于是在 Windows 下，我们同样也找到了一套解决方案来编译和运行本书的代码，它就是 Cygwin。

Cygwin 是一个在 Windows 平台上运行的 UNIX 模拟环境，是 Cygnus Solutions 公司开发的自由软件。它对于学习 UNIX/Linux 操作环境、从 UNIX 到 Windows 的应用程序移植，或者进行某些特殊的开发工作（尤其是使用 gnu 工具集在 Windows 上进行嵌入式系统开发）都非常有用。

下面我们就来介绍一下 Cygwin 环境的安装和使用方法。如果您是 Linux 用户，就可以跳过这部分，继续阅读下一小节的内容。

~~1.1.1 准备 Cygwin 开发环境~~

首先，我们需要从 www.Cygwin.org/Cygwin/setup.exe 处下载最新版的 Cygwin 安装程序。

接下来运行这个安装程序，如图 1-1 所示。

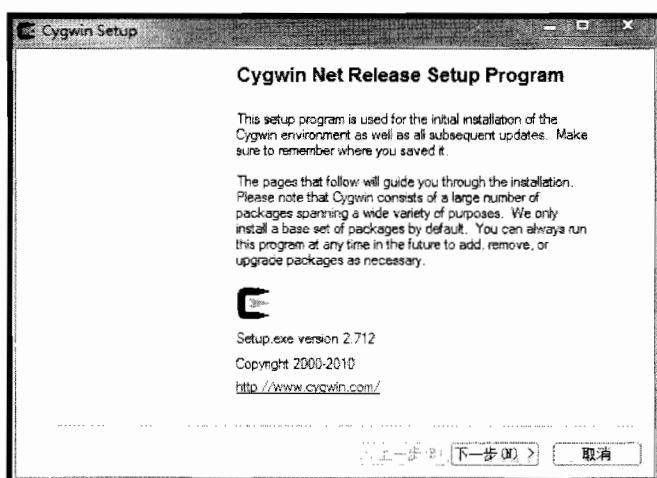


图 1-1 Cygwin 的安装

在图 1-1 的对话框中单击“下一步”按钮，进入安装类型选择页面，如图 1-2 所示。

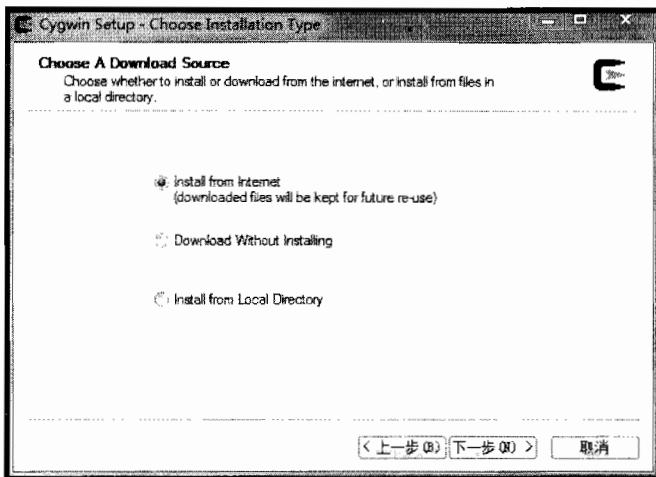


图 1-2 Cygwin 安装类型选择页面

此处如果没有特殊的要求，可以采用默认的“Install from Internet”单选项，通过网络安装 Cygwin 系统。当然，这要求读者的电脑能够上网才可以。

继续单击“下一步”按钮，进入安装路径选择页面，如图 1-3 所示。在这里，读者可以根据自己系统的情况选择某一安装路径。

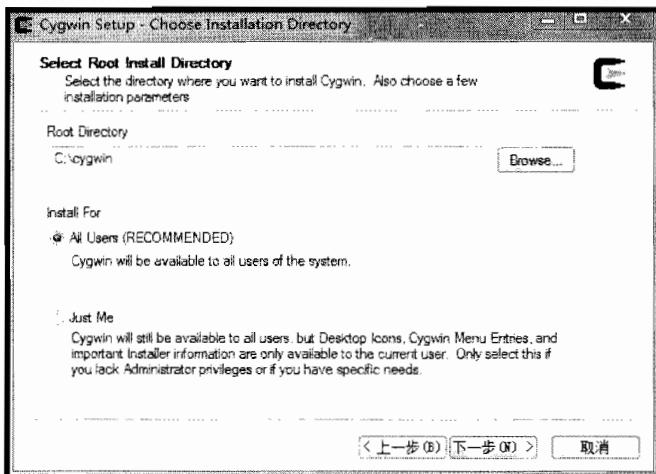


图 1-3 Cygwin 安装目录选择页面

再次单击“下一步”按钮，程序进入软件包下载目录选择页面。之前我



们已经选择了通过网络安装的方法来进行安装，那么在这个页面中，我们就需要指定一个本地文件夹，保存从网上自动下载的各种应用软件包，如图 1-4 所示。

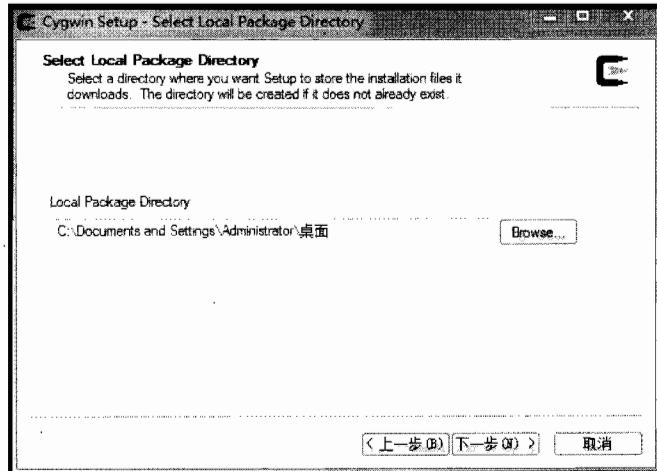


图 1-4 Cygwin 软件包下载目录选择页面

之后程序将打开连接方式选择页面，如果没有特殊要求，这一步选中默认的“Direct Connection”单选项即可，如图 1-5 所示。

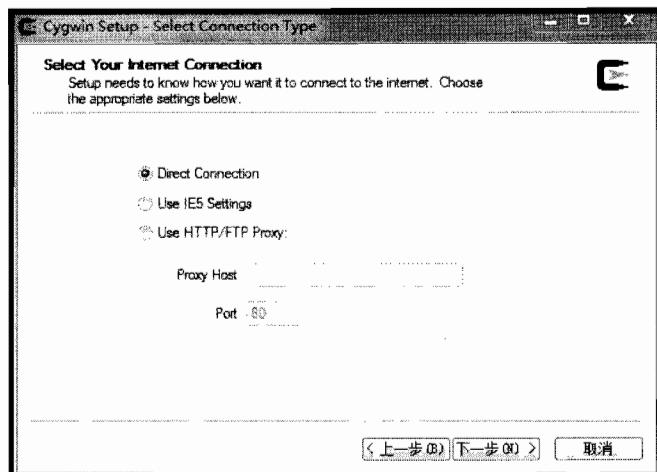


图 1-5 Cygwin 连接方式选择页面

单击“下一步”按钮，进入下载站点选择页面。通常我们会选择 <http://www.cygwin.cn/> 作为下载源，如图 1-6 所示，对于国内的用户来说，这个站点的连接速度最快。

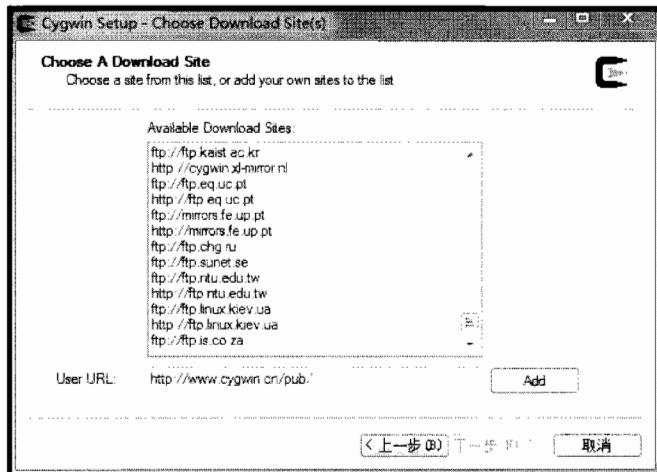


图 1-6 Cygwin 下载站点选择页面

如果是首次运行 Cygwin 安装程序，则需要在“User URL”文本框中添加“<http://www.cygwin.cn/pub/>”这一地址，单击“Add”按钮。在其他情况下，只需在“Available download Sites:”列表框中选择“<http://www.cygwin.cn>”站点即可。下载源选择完成后，单击“下一步”按钮，进入软件包选择页面，如图 1-7 所示。

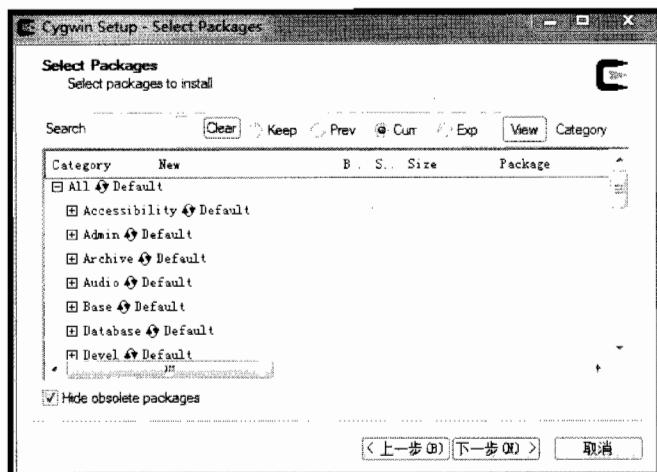


图 1-7 Cygwin 软件包选择页面

在该页面中，我们可以选择需要安装的软件包。由于 Cygwin 默认安装的软件包不能完全满足我们自己的操作系统开发，因此需要至少选择两个额外的软件包进行安装。单击“Devel”类前面的“+”，将该分类展开，从中选择“make”和“gcc4”两组软件包，而其他软件包按照默认方式处理即

可，完成操作后，单击“下一步”按钮，程序进入安装过程，如图 1-8 所示，显示安装进度。

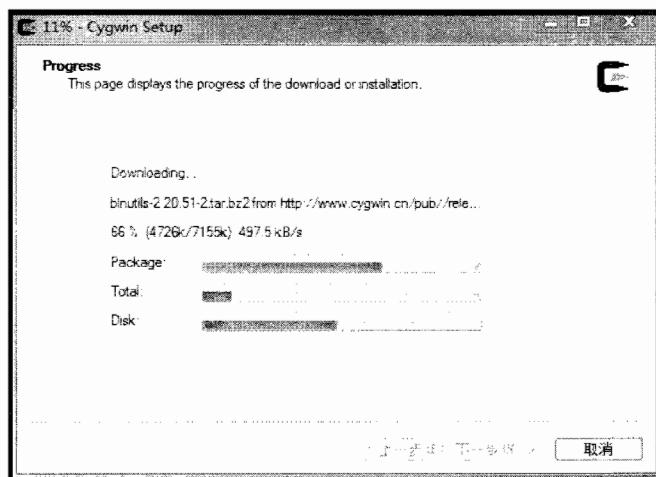


图 1-8 Cygwin 安装过程

片刻之后，Cygwin 的安装就完成了，如图 1-9 所示。

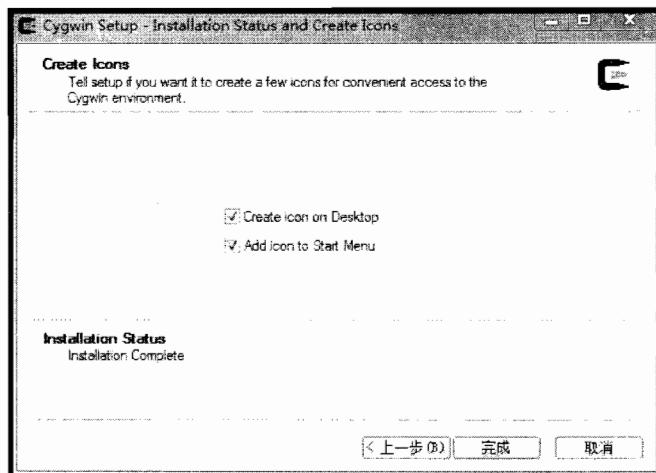


图 1-9 Cygwin 完成安装

单击“完成”按钮，退出 Cygwin 的安装过程。

Cygwin 的使用方法非常简单，只需要单击桌面上的 Cygwin 图标，就会弹出一个类似于 Windows 命令行的页面，如图 1-10 所示。

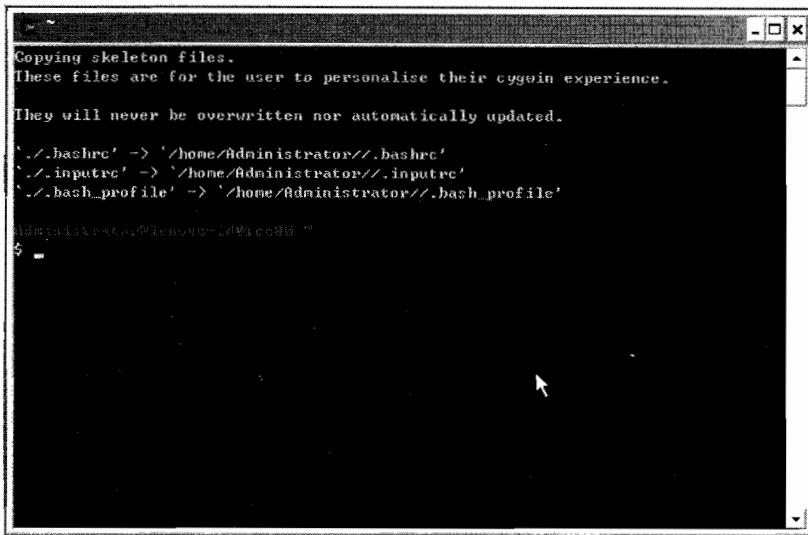


图 1-10 Cygwin Shell 页面

简单地讲，如图 1-10 所示的操作页面实际上是在 Windows 操作系统中模拟出来的类 UNIX 系统中的 Shell 页面，我们可以输入一些 UNIX 下常用的命令，如 ls、cd 等，这些命令的使用方法和运行结果与 UNIX 系统是一致的。对于那些使用 Windows 操作系统的朋友们来说，属于我们自己的操作系统也将在这样一个环境中诞生！

~~1.1.2 使用 Linux 开发环境~~

如果读者朋友已经准备使用 Linux 来开发属于自己的操作系统，又没有把握在电脑中直接安装某个 Linux 发行版，那么一个折中的办法就是使用虚拟机，就好像我们又多了一台电脑一样。

目前流行的虚拟机软件五花八门，各有优劣。使用什么样的虚拟机安装 Linux 系统，其实并没有太大的差别。在这些虚拟机中安装 Linux 操作系统的大致步骤如下。

(1) 选择读者喜欢的 Linux 发行版，主流的如 ubuntu、redhat，个性的如 gentoo、archLinux，极端的如 LFS。

(2) 从各发行版的网站上下载安装光盘镜像，这些光盘镜像有的可能很小，需要通过网络安装的方式进行，有的也许很大，直接一张光盘就可以完

成一个基本系统的安装。

(3) 将下载下来的光盘镜像插入到虚拟机中，根据各 Linux 发行版的相关文档或软件提示进行 Linux 的安装。

出于篇幅的原因，本书不对虚拟机下安装 Linux 发行版的详细步骤进行介绍。因为如果介绍详细步骤，一连串的关于 Linux 原理、基本操作方法等问题都将被牵扯出来，本书很有可能就会变成一本 Linux 入门级书籍。为避免喧宾夺主，我们建议只了解 Windows 操作系统的朋友使用 Cygwin 进行开发。对于那些了解 Linux 的朋友们来说，对于 Linux 如何安装自然会非常清楚，因此也无须多言。

1.2 开发工具的使用

自己动手写操作系统的第二个重要工作是选择合适的编译器和编辑器。

1.2.1 编译器的选择和安装

由于我们的操作系统需要运行在 ARM 体系结构中。因此，在琳琅满目的 ARM 编译器中选择出一款，用来编译操作系统源程序，就成为了必须首先要解决的问题。于是我们选择了 GCC。

GCC 的大名，相信读者应该都会如雷贯耳。简单地说，GCC 就是由 GNU 基金会组织开发的一套开源的编译器。根据相关许可，我们可以自由地下载和使用这套编译器，可以随意地修改和重新发布它的源代码，整个过程不需要付一分钱或承担任何责任。这就是我们选择 GCC 作为开发工具的一个最主要的原因。

GCC 的免费获得和使用并不等于质量低劣或功能欠缺。它的功能不逊于任何一款商用 ARM 编译器。这也是我们选择 GCC 作为操作系统开发工具的第二个原因。

接下来我们就来介绍一下 GCC 编译器的安装过程。

GCC 是源码开放的，所以理论上，我们完全可以在网站上下载 GCC 源代码，然后在本地进行编译，生成一个能够编译 ARM 体系结构下程序

的工具。

这个过程听起来似乎轻而易举，但做起来却没有那么容易。况且也没有太大的必要自己生成编译器。因此，本书不会对这一部分内容进行介绍。如果读者确实对自己构建编译器很感兴趣，可以去“www.leeos.org”上翻阅相关文档。

其实，这个问题远远没有那么复杂。编译器也只不过是一套软件而已，别人编译好的，拿到我们的系统中一样可以使用。

通常我们会看到网上有许多名为“arm-Linux-gcc”或“arm-elf-gcc”的编译工具。使用这些工具来编译属于我们自己的操作系统，基本上都是可以的。

读者也许会提出这样的疑问，为什么这些编译器的名字有些与众不同呢？其实，这些编译器的命名方法有些约定俗成的规矩，例如，开头的“arm”关键字表示该编译器将生成 ARM 体系结构机器码，而中间的“Linux”或“elf”则表示应用的目标平台。一般说来，“Linux”关键字意味着该编译器在编译时针对 Linux 系统做了特定的优化，生成的代码将更好地运行在 Linux 系统之中，而“elf”关键字则更适合生成通常的可执行程序代码。对于我们自己的操作系统来说，使用“arm-elf-gcc”这套工具显然更合适一些。与传统的 PC 程序开发不同，嵌入式的开发过程通常都是在 PC 中对源代码进行编译，再将编译生成的可执行程序放到嵌入式平台中去运行，因此有人把这种编译过程称作交叉编译或跨平台编译，而将一整套用于交叉编译的工具称为交叉编译工具链。

为了方便读者对本书代码进行实践，我们制作好了一套专门用于这套操作系统的编译器，大家可以去“www.leeos.org”网站上下载使用。

需要注意的是，这样的一套编译工具只能用来编译我们自己写的操作系统，不能保证编译其他 ARM 应用程序的正确性，包括 Linux 下的程序和内核。这是因为在制作这套工具的过程中，为了能让编译器尽可能小一些，我们没有添加任何标准 C 函数库和其他一些 GCC 扩展工具，对于我们自己的操作系统来说，这样做并无问题。也就是说，通常情况下，一个操作系统的核心代码并不需要第三方函数库，但这个假设对于绝大多数应用程序来说却并不成立。

当下载到某一个版本的交叉编译工具之后，第一步要做的就是将其解压。

以我们所提供的交叉编译工具链为例，Windows 用户可以运行如下命令。

命令 1-1

```
tar zxvf leeos_tools_for_Cygwin.tar.gz
```

当然这需要首先将下载的编译工具复制到某一目录中，如果读者将 Cygwin 安装到了 C 盘的话，这个目录可以是“C:\Cygwin\usr”，然后打开 Cygwin 命令行，运行下面的命令。

命令 1-2

```
cd /usr
```

这样，我们就可以使用命令 1-1，将安装工具解压到“usr”目录下。此时虽然解压过程已经完成，但并不表示编译器已经可以使用，我们还需运行如下命令才能将编译器彻底地安装到系统中。

命令 1-3

```
echo "PATH=$PATH:/usr/le eos_tools_for_Cygwin/bin" >>/etc/profile
```

对于 Linux 用户，步骤基本是一样的。但需要注意的是这些命令需要以超级用户的身份去运行。使用命令 1-4 可以进行超级用户的切换。

命令 1-4

```
su
```

使用 su 命令后，终端会提示输入超级用户密码。用户切换完成后，就可以使用 cp 命令将下载的编译工具复制到“/usr”目录中。

命令 1-5

```
cp /where/your/compiler/locate/le eos_tools_for_Linux.tar.gz /usr
```

接下来运行命令 1-2，然后是命令 1-1 和命令 1-3。注意，要将“le eos_tools_for_Cygwin”改成“le eos_tools_for_Linux”。重启系统后，交叉编译工具就可以使用了。

1.2.2 编辑器的选择和使用

编译器已经有了，我们需要使用什么工具来编写操作系统代码呢？

其实，只要自己觉得合适，无论是使用简单的（诸如记事本之类的）工



具还是复杂的（诸如进行程序开发的专用 IDE），都不会有问题，每个人心中应该都有自己最理想的编辑工具。在程序的开发过程中，我们只需要打开一款编辑器，将程序源代码敲进去，然后将这些内容保存成源文件，那么剩下的工作就是使用编译器对这些代码进行编译。

接下来，让我们通过 Windows 下的“记事本”这一工具来演示一下在 Cygwin 环境中编写代码的一般方法，其他代码编辑器的使用与该方法类似。

首先请打开文本编辑器，向里面输入一个空的 main 函数，如图 1-11 所示。



图 1-11 使用编辑器编写代码

接下来我们尝试对这段代码进行编译。将这些内容保存成文件，取名为“test.c”。在 Cygwin 的安装目录下，有一个“home”文件夹，进入这个文件夹，可以看到以用户名命名的一个文件夹，该文件夹代表 Cygwin 用户的家目录。所有个人私有的文件和数据存放到这个目录中是比较合适的。

例如，某个人将 Cygwin 安装到了 C 盘根目录，同时他又是以 Administrator 用户登录的话，那么这个文件夹就应该是 C:\Cygwin\home\Administrator。

保存完成后，运行 Cygwin，并在弹出的命令行中输入“ls”这条命令，

可以看到 test.c 文件出现在该目录下，如图 1-12 所示。



图 1-12 在 Cygwin 命令行运行“ls”命令

紧接着就在这个命令行下，运行编译工具，将“test.c”文件编译成可执行程序，命令如下。

命令 1-6

```
arm-elf-gcc -nostdlib test.c
```

最终在该目录下，会出现一个名为“a.out”的文件，这个文件正是由 GCC 编译 test.c 文件后生成的。这就表示我们已经使用编辑器编写了代码，并用编译器成功编译了第一个程序。

对 GCC 比较熟悉的读者可能会觉得命令 1-6 这种编译方法比较奇怪。如果您使用的是我们提供的专用编译器，那么“-nostdlib”参数是必需的。它表示编译时不去链接标准函数库。相关参数的具体含义，我们会在接下来的章节中详细阐述。

如果是 Linux 用户，那么程序的编写和编译过程会更简单些。因为在 Linux 下，存在有一些基于命令行的文本编辑器，如 vi。使用这样的一些编辑器，程序的编写、代码的保存和编译等过程都可以在命令行下实现，这样，开发效率会较高。

如果一些 Linux 初学者觉得 vi 较难掌握，也可以尝试使用 nano。nano 也是一个在命令行下就能运行的文本编辑器，它的用法像 Windows 下的“记



事本”一样简单，图 1-13 是一个在 Linux 下运行 nano 的页面。



图 1-13 nano 运行页面

在确保系统中安装了 nano 工具的前提下，想要使用 nano，只需要首先通过 cd 命令切换到某一路径下，然后运行下面的命令。

命令 1-7

```
nano test.c
```

这样就可以打开一个 test.c 文件，并进入文本编辑的页面中了。在写入适当的内容之后，按下“Ctrl+O”组合键可以保存文件，按下“Ctrl+X”组合键可以退出页面。

如果有些朋友还是觉得使用图形化的工具更加方便，那么，在 Linux 下，也有很多图形化的文本编辑器可供选择，如 gedit、gvim、mousepad，等等。这些工具的使用方法与 Windows 下的编辑工具完全一致。最终我们还是要通过命令行来编译源程序。

1.3 虚拟硬件的安装和使用

现在虽然我们已经编译生成了一个应用程序，但却没有办法运行它。其中的道理很简单，我们使用 ARM 编译器交叉编译生成的可执行程序，自然



是只有在 ARM 硬件环境下才能够使用。

一个最直接的方法，就是使用 ARM 开发平台。市面上基于 ARM 的开发板多如牛毛，无论选择哪一款，都可以帮助我们将操作系统运行起来。但很显然，这种方法并不具备可操作性。其中的一个原因是这样将会无端地增加我们的学习成本。另一个原因是，由于各种开发板的芯片选型、电路设计都不尽相同，使用这些开发平台进行开发，我们将会陷入无尽的硬件细节当中，将不能够从宏观的角度去理解操作系统的原理。

为了解决这个问题，我们需要借助一种特殊的软件来运行编译器生成的 ARM 程序，这就是虚拟机。

虚拟机是运行在 PC 中的一种软件，能够模拟出 ARM 硬件环境。这样一来，我们既不必在操作系统的编写和学习过程中多花一分钱，又能拥有一个统一的硬件平台，不至于牵扯过多的硬件细节，迷失到森林之中。

有很多免费的 ARM 虚拟机可供选择，SkyEye 正是其中之一。

SkyEye 是一个开源软件项目，其目标是在通用的 Linux 和 Windows 平台上实现一个纯软件集成开发环境，模拟常见的嵌入式计算机系统。我们可在 SkyEye 上运行 μCLinux 以及 μC/OS-II、Linux 等多种嵌入式操作系统和各种系统软件，并可对它们进行源码级的分析和测试。SkyEye 是一个指令级模拟器，可以模拟多种嵌入式开发板，可支持多种 CPU 指令集，并支持网络、Flash 等大量硬件。在 SkyEye 上运行的操作系统意识不到它是在一个虚拟的环境中运行的。值得一提的是，SkyEye 项目是由清华大学的博士后陈先生发起的，是属于我们中国人的优秀软件项目。

与 SkyEye 类似的虚拟机还有 QEMU。从某种程度上说，QEMU 较之于 SkyEye 更加优秀。但是本书最终选择了 SkyEye 而放弃了 QEMU，主要有以下两个原因：

第一，SkyEye 易于配置，方便使用。 我们可以通过修改配置文件，轻松地搭建出具有任何硬件特性的虚拟平台。

第二，SkyEye 对时下比较流行的几款 ARM 芯片支持良好， 如 SAMSUNG 的 s3c 系列及 ATMEL 的 AT91 系列等。拥有 ARM 知识基础的读者对这两个系列的芯片应该都有所了解。以这些常用芯片为基础进行操作系统的开发，将有利于为那些只是略懂硬件的读者拨开硬件迷雾，更清晰地看清楚操作系统的全貌。



ARM虚拟机既已选定，接着我们就来了解一下SkyEye的安装与使用。

1.3.1 SkyEye的安装

SkyEye的安装不需要多说，只需要从官方网站上下载SkyEye源代码，然后复制到本地文件夹中，并在命令行中输入相应命令即可。如果以1.2.6_rc1版本为例，那么安装命令如下。

命令 1-8

```
tar jxvf skyeye-1.2.6_rc1.tar.bz2  
cd skyeye-1.2.6_rc1  
.configure  
make  
make install
```

如果是Linux用户，在运行make install命令前可能需要暂时切换到超级用户下，这可以通过命令1-4实现。

由于每个人的系统环境可能会略有差别，我们提供的这种安装方法不能保证一定会成功。即使如此，读者也不用担心，我们已经预先编译好了SkyEye可执行程序，读者也可以去“www.leeos.org”下载。

对于Windows用户，可以将下载下来的“skyeye.exe”文件复制到Cygwin安装目录中的“usr/bin”文件夹下。而对于Linux用户，可以下载skyeye这个文件并复制到“/usr/bin”目录中。这样，ARM虚拟机就安装完成了。

1.3.2 SkyEye的使用

想要使用SkyEye，就必须要了解它的配置方法。skyeye.conf文件是SkyEye的默认配置文件。通过编写合适的skyeye.conf文件，我们可以配置出任何SkyEye支持的硬件环境。

代码1-9是一个比较典型的skyeye.conf配置文件的范例，针对的是s3c2410x这款芯片。

代码 1-9

```
cpu: arm920t
```

```
mach: s3c2410x  
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,  
file=./le eos.bin, boot=yes  
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

第一行中的关键字 `cpu` 记录的是芯片系列，第二行中的 `mach` 关键字记录的是芯片的具体型号。

接下来的 `mem_bank` 关键字描述的是芯片内存空间特性。其中，map=M 代表该段内存空间是一段内存，而如果是 map=I，则代表该内存空间对应的是外设端口。type=RW 表示的则是该内存空间具备可读写属性。之后的 addr=0x30000000 和 size=0x00800000 分别代表内存空间的起始地址和大小，这里，我们将虚拟开发平台配置为拥有 8M 内存并开始于 0x30000000 处。后面的 file=./le eos.bin 表示的是预先要被加载到这段内存空间的映像文件。而 boot=yes 则表示默认从此处启动。有了这两个属性，在操作系统调试或运行时就不必关心程序的引导，而可以直截了当地去运行了。需要注意的是，mem_bank 关键字之后的内容需要出现在一行里，由于版面的原因，这里被分成了两行。

关于 SkyEye 配置文件的写法，我们已经了解，下面介绍它的运行方法。

首先我们需要将代码 1-9 保存成文件，名为“skyeye.conf”，并放到某一个文件夹下。其次，需要将编译完成的二进制映像文件放到同一个目录下，这个二进制文件的名字应该与 `skyeye.conf` 文件中记录的文件名相一致。最后，只需要开启一个命令行，输入 `skyeye` 命令，程序就可以运行了。整个过程如图 1-14 所示。

```
#ls  
le eos.bin skyeye.conf  
#skyeye  
  
***** WARNING *****  
If you want to run ELF image, you should use -e option to indicate  
your elf-format image filename. Or you only want to run binary image,  
you need to set the filename of the image and its entry in skyeye.conf.  
*****  
  
Your elf file is little endian.  
arch: arm  
cpu info: armv4, arm920t, 41009200, ff00ffff, 2  
mach info: name s3c2410x, mach_init addr 0x426c78  
uart_mod:0, desc_in:, desc_out:, converter:  
SKYEYE: use arm920t mmu ops  
Loaded RAM ./le eos.bin  
start addr is set to 0x30000000 by exec file.
```

图 1-14 SkyEye 的使用方法

最后还要强调，本书所有的示例代码都是运行在由 SkyEye 模拟的 s3c2410 这一硬件平台，原因就像前面介绍的，s3c2410 在国内比较流行，了解的人较多。同时，SkyEye 对这一平台的支持也非常成熟且资源丰富，这些对操作系统的开发都非常有利。如果读者对其他平台有所了解，将本书的例子代码移植过去也不是什么难事。

1.4 总结

本章我们主要介绍了与开发嵌入式操作系统相关的一系列工具及其使用方法。这些工具方法都具备免费获取、使用简单等特点。它们有可能不是最优秀的，但恰恰都是最适合的。

其实一件作品的完成，关键并不在于工具，而是在于使用工具的人。美国人可以在计算机技术尚不发达的情况下就让航天飞机上天，中国人亦可在经济条件极度困难的情况下爆破原子弹。倘若胸怀愚公之志，纵使手挑肩担，也能将大山夷为平地。

不过这个问题似乎说远了，其实自己一步一步地去写一个操作系统，远没有那么大的难度。不信？那就接着读下去吧！



第2章

基础知识



现在我们就可以动手打造这款属于我们自己的操作系统了。当然，在这之前我们还需要将开发操作系统所需的必备基础知识说清楚。众所周知，操作系统是一门偏难的综合性学科。尤其在亲自实践的过程中，对于编程语言、程序算法、硬件原理等相关知识，都需要开发者有相当深入的了解。

因此，无论读者从事的是基于 PC 的程序开发还是基于应用层的嵌入式开发，我们的传统知识结构可能都不足以实现一个完整的操作系统，这可能将会成为我们开发属于自己的操作系统的一个障碍。为了能够扫除这些障碍，我们有必要对一些知识和技巧进行一定程度的补充。

本章正是以此为目的，虽不能对开发操作系统所必备的所有知识进行深入并且全面的讲解，但至少会保证将一些重点或易被忽略的地方拿出来聊一聊。这些内容涵盖了链接和库、基于 ARM 的汇编程序设计、汇编与 C 语言间的混合编程、过程调用标准等。

这些话题都跟程序的编写直接相关，在后续章节的学习过程中都会反复讲到。我们会介绍几个实际的编程例子，确保读者在读完本章之后，至少能够编写并运行一些最基本的程序，为操作系统的开发打下基础。

如果读者对这些知识已有所了解，可以跳过本章，直接学习后边的内容。

2.1 使用 C 语言写第一段程序

我们不如从每个程序员都曾写过的“hello world”切入，来尝试编写能

在虚拟环境中运行的第一段代码，看看操作系统的开发过程究竟是怎样的。

首先请打开文本编辑器，输入以下内容。

代码 2-1

```
#define UFCON0 ((volatile unsigned int *)(0x50000020))
void helloworld(void){
    const char *p= "helloworld\n";
    while(*p){
        *UFCON0=*p++;
    };
    while(1);
}
```

程序简单到无须介绍，这里只需要说明一点，物理地址 0x50000020 代表的是 s3c2410 的串口 FIFO 寄存器地址，简单地说，就是写向该地址的数据都将会通过串口发送给另一端。这段程序只是串行的将字符串“helloworld”依次送给串口 FIFO 寄存器。我们将以上内容保存成文件，命名为“helloworld.c”。

接下来让我们尝试编译这段程序，在终端里运行如下命令。

命令 2-1

```
arm-elf-gcc -O2 -c helloworld.c
```

编译过程中，我们使用了一个优化参数-O，这样最终生成的代码将由编译器视情况优化，而数字 2 则代表了一个优化级别，数字越高，优化程度越深，但同时带来的不确定性也越多。

目前我们只能将这段代码首先编译生成目标文件，而不能直接生成可执行文件，原因很简单，因为我们的程序当中没有 main 函数。也就是说，编译器找不到程序的运行入口，在链接过程中便会报错。那么是不是说凡是 C 代码，想要生成可执行程序并且成功运行，就必须要有 main 函数呢？

不是。这个答案和我们理解的传统 C 语言程序有些出入，初学者也许不能接受。但事实是 main 函数和普通的函数并没有任何差别，它只不过是由标准所规定的程序入口而已。我们完全可以使用一些手段迫使程序从我们指定的任意函数处开始运行，这一方法可以通过添加适当的编译参数或使用特定的链接脚本来实现。这些内容稍后我们就会讨论。

在命令 2-1 运行之后，一个名为“helloworld.o”的文件便生成了。接下

来需要运行链接工具，用该目标文件来生成一个 ELF 格式的可执行程序。

命令 2-2

```
arm-elf-ld -e helloworld -Ttext 0x0 helloworld.o -o helloworld
```

链接器将目标文件“helloworld.o”链接生成可执行程序 helloworld。参数-e 的作用是指定程序的运行入口。也就是说，可执行程序将会从代码 2-1 中的 helloworld 函数开始执行，取代了默认使用的 main 函数。这样我们就实现了从自定义的函数入口运行程序。参数-Ttext 的作用是指定该程序的运行基地址，此处的值是内存零地址。也就是说，链接工具将目标文件的 helloworld 函数链接到内存 0x0 的位置上，并且从此处执行。

那为什么是内存零地址呢？这是因为在 ARM 体系结构中，程序必须从内存零地址处开始运行。将目标文件链接到内存 0x0 处，我们的 helloworld 理论上就具备了运行条件。

此时，helloworld 程序虽然已经生成，但它仍然不能拿来直接加载到硬件环境里去运行，我们还必须使用如下命令生成一个只包含程序机器码的二进制文件。

命令 2-3

```
arm-elf-objcopy -O binary helloworld helloworld.bin
```

这里的 arm-elf-objcopy 命令将 ELF 格式文件中的二进制机器码抽离出来，生成“helloworld.bin”文件，该文件除了运行时所必要的数据和代码之外，不含有任何其他信息，是纯粹的可执行机器码镜像。而我们原来所理解的可执行文件格式，除了实际代码之外，还会包含 ELF 格式文件头、段头或节头等附加信息。这样程序在运行时，就要首先解析这些附加信息，根据 ELF 格式信息构造出程序运行时所需要的环境，然后才能运行。

接下来我们需要根据程序的编译情况来编写一个 SkyEye 的配置文件，其内容如下。

```
cpu: arm920t
mach: s3c2410x
#physical memory
mem_bank: map = M, type = RW, addr = 0x00000000, size =
0x00800000, file=./helloworld.bin
#all peripherals I/O mapping area
```



```
mem_bank: map = I, type = RW, addr = 0x48000000, size = 0x20000000
```

在该配置文件中，我们将“helloworld.bin”文件加载到硬件平台 0x0 地址处，与之前编译时程序的链接环境一致。同时，还要保证 0x50000020 处的地址是作为外设寄存器被映射出来的。

将上述内容保存成文件，名为“skyeye.conf”，与“helloworld.bin”文件放置在同一个目录下。在终端运行 skyeye 命令，就会看到如下结果。

```
.....
Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM ./helloworld.bin
helloworld.
```

可以看出，helloworld 字符串从 SkyEye 虚拟机中打印了出来。我们的第一段 ARM 程序正在运行中！

在上述程序的执行过程中，有一点可能会令读者感到费解。我们的程序明明是将字符串 helloworld 发送给了串口，但为什么最终会在 SkyEye 的页面中出现？

通常在嵌入式产品的开发过程中，串口多是作为设备终端，用于进行产品的调试和操作的。于是 SkyEye 默认地将串口映射到命令行中，以方便虚拟硬件的模拟。当然这也仅仅是虚拟环境所提供的一种方便而已，在实际的串口设备通信中，由于会涉及到传输速率、数据格式等诸多问题，这段程序能够成功运行的可能性并不大。

简要总结一下上面这段程序：我们首先使用编译器将一段代码编译成目标文件，同时又使用链接器将目标文件链接成了可执行程序，并在链接的过程中做了点小动作，使得程序没有使用 main 函数也可以成功编译，又将程序的入口设置在了内存 0x0 处。最后，我们又使用了目标文件复制工具将 ELF 格式的可执行程序转换成只包含代码和数据的原始格式，并通过 SkyEye 成功运行。

在整个过程中，我们涉及了一些编译工具的基本用法。下面我们就来

了解一下链接工具的一种高级用法，那就是通过链接脚本来链接目标文件。

2.2 用脚本链接目标文件

什么是链接脚本？链接脚本就是程序链接时的参考文件，其目的是描述输入文件中各段应该怎样被映射到输出文件，以及程序运行时的内存布局，等等。绝大多数情况下，链接程序在链接的过程中都使用到了链接脚本。

有的读者可能会疑惑，似乎我们在利用 GCC 编译程序的时候并没有使用到什么链接脚本，程序依然会被成功链接并且正常执行。其实，链接工具 arm-elf-ld 在未被指定使用哪一个链接脚本的时候，会使用内嵌到链接工具内部的默认脚本，我们可以使用 -verbose 参数来查看该链接脚本。

当应用程序处在操作系统之上时，往往不需要显式指定链接脚本，因为自己编写的链接脚本可能和操作系统默认环境不符，导致运行出错，所以通常使用链接命令内置的脚本，这样可以保证程序运行环境和操作系统默认环境相一致，保证了程序的正常运行。但是，如果我们的程序是运行于操作系统之下或程序就是操作系统本身，那么链接脚本就显得十分重要了。要根据硬件平台的实际环境去编写合适的链接脚本，代码或数据才有可能出现在正确的位置上并正常运行。

本小节我们将讨论链接脚本的具体细节，使用的方式并不是简单地罗列参考手册的内容，而会针对可能用到的知识做深入的研究，将那些根本不可能用到的知识扔到垃圾桶里。

让我们首先来看一下一个典型的链接脚本是什么样子的。

代码 2-2

SECTIONS

{

```
. = 0x1000;  
.text : { *(.text) } → DUT  
. = 0x8000000;  
.data : { *(.data) }
```



```
.bss : { *(.bss) }
}
```

代码 2-2 就是一个最简单的链接脚本，其中使用了一个非常重要的命令——SECTIONS，这个命令是用来描述输出文件的内存布局的。SECTIONS 命令的标准格式如下。

代码 2-3

```
SECTIONS
{
    sections-command
    sections-command
    .....
}
```

一个 SECTIONS 命令由若干个 sections-command 组成。通过该命令，我们可以设计出程序运行时各段在内存中的分布情况。例如，在代码 2-2 中，代码段被安排到内存的 0x1000 处，数据段则分布在内存的 0x8000000 位置，而 bss 段从形式上紧挨着数据段，整个脚本看起来很容易理解。

SECTIONS 命令的具体用法如下。

(1) 点号 (.): 点号在 SECTIONS 命令里被称为位置计数器。通俗地讲，它代表了当前位置，你可以对位置计数器赋值，例如，第三行中我们对点号赋予了 0x1000 这个值，其结果是代码段的起始位置从 0x1000 开始。我们也可以将位置计数器的值赋给某个变量，而这个变量可以在源程序中使用。明确定某段的起始地址并不是必需的，就如上例中，在 bss 段前面就没有对位置计数器赋值，在这种情况下，位置计数器会采用一个默认值，而这个默认值也会随着各段的大小动态地增加。例如，代码 2-2 中，在数据之前，位置计数器的值为 0x8000000，而在数据段之后，位置计数器并没有被显性赋值，其默认值是 0x8000000 加上数据段的长度。所以，可以说在内存中 bss 段形式上是紧挨着数据段分布的。当然，实际情况可能并非如此，我们稍后会做讨论。如果 SECTIONS 命令一开始就没有对位置计数器赋值，则其默认值为零。

(2) 输出段定义：代码 2-2 中，各关键字代表了输出段的段名。.text 关键字定义了该输出位置为代码段，花括号内部定义了代码段的具体内容，其中，星号 (*) 代表所有文件，*(.text) 的意思是所有目标文件的代码段都



将被链接到这一区域，我们也可以特别地指定某个目标文件出现在代码段的最前面。.data 关键字定义了该输出位置为数据段，其格式和用法与.text 关键字相同。.bss 关键字定义了输出位置是 bss 段，格式和用法与上面相同。这种写法很可能给我们造成一种错觉，即对输出段的定义必须以 text、data 或 bss 关键字命名。其实输出段完全可以任意定义，因为输出段的实际内容与输出段的命名无关，而只与花括号内的具体内容有关。

(3) ALIGN(N): 在代码 2-2 中并没有 ALIGN 关键字。但是在实际应用中，我们经常需要使用 ALIGN 产生对齐的代码或数据。很多体系结构对对齐的代码或数据有严格的要求，还有一些体系结构在处理对齐的数据或代码时效率更高，这都体现了 ALIGN 关键字的重要性。我们可以用某一数值取代 ALIGN()括号中的 N，同时对位置计数器赋值，如.=ALIGN(4)就表示位置计数器会向高地址方向取最近的 4 字节的整数倍。

链接脚本除了 SECTIONS 命令之外，还有一个比较常用的命令是 ENTRY 命令，该命令等同于 arm-elf-ld 命令的参数-e，即显式地指定哪一个函数为程序的入口。

现在我们可以应用上面提到的知识，针对前面的 helloworld 程序写一个链接脚本，其内容如下。

代码 2-4

```
ENTRY(helloworld)
SECTIONS
{
    . = 0x00000000;
    .text : {
        *(.text)
    }
    . = ALIGN(32);
    .data : {
        *(.data)
    }
    . = ALIGN(32);
    .bss : {
        *(.bss)
    }
}
```

结合前面的描述，上面这段链接脚本的作用是将程序的代码段链接到地址 0x0 处，保证了系统上电时就能从该地址中获取第一条指令并运行。程序的数据段在代码段之后，而 bss 段则被安排到数据段之后。这两个段的起始地址都是经过对齐的。最后，链接脚本还明确规定将名为 helloworld 的函数作为整个程序的入口。这也就是说，helloworld 函数就是出现在内存 0x0 位置上的那个函数。

好了，链接脚本写完了，那我们应该如何使用该脚本呢？这就要在运行链接命令时借助于-T 参数。使用-T 参数可以让链接器从脚本中读取并解析相应命令，然后按照命令的要求链接生成可执行程序。

如果针对前面的例子，它的用法会像下面这样。

命令 2-4

```
arm-elf-ld -T helloworld.lds helloworld.o -o helloworld
```

这样我们就得到与前一小节同样的结果，但使用的方法却更加专业。

2.3 用汇编语言编写程序

前面能够运行在虚拟硬件环境中的第一段程序是使用 C 语言写成的。理论上我们可以完全使用 C 语言来编写整个操作系统。但在实际应用中，完全使用 C 语言编写的操作系统却寥寥无几。汇编语言虽然有很多的缺点，但在操作系统底层开发中，有时却能发挥出不可替代的作用，这一点相信读者会在今后的学习中有深入的体会。正因为如此，我们还需要利用一节的篇幅，说一说如何使用汇编语言进行 ARM 程序开发。

我们仍将采用边做边讲、在遇到问题的时候再去说明的方式，避免因为枯燥地将知识点罗列出来而影响读者的兴趣。

下面仍从一个例子开始。

代码 2-5

```
.arch armv4
.global helloworld

.equ REG_FIFO, 0x50000020
```



```
.text
.align 2

helloworld:
    ldr r1,=REG_FIFO
    adr r0,.L0
.L2:
    ldrb r2,[r0],#0x1
    str r2,[r1]
    cmp r2,#0x0
    bne .L2
.L1:
    b .L1
.align 2
.L0:
.ascii "helloworld\n\0"
```

代码 2-5 就是 helloworld 程序的汇编版本。想要让读者理解这段程序，我们并不需要对 ARM 的汇编程序设计进行太过系统的介绍，我们只需要说清楚两点，一是寄存器，二是指令集。

寄存器是 CPU 进行运算时所必需的存储设备。ARM 上的寄存器很丰富，能够参与普通运算的从 R0 到 R12，就有 13 个之多。如果没有特殊规定，程序在运算时可以随意选择 R0~R12 的任意一个寄存器参与运算。除了这些寄存器之外，在 ARM 体系结构中还包括负责保存堆栈地址的 R13 寄存器，负责保存程序返回值的 R14 寄存器以及负责记录程序地址的 R15 寄存器。这些寄存器由于拥有专门的功能，所以有的时候也使用别名来称呼它们，分别叫做 SP、LR 和 PC。另外，ARM 还有一个非常特殊的寄存器专门用来保存程序的运行状态，叫做程序状态寄存器，使用 CPSR 来表示。CPSR 中既包含描述程序在运算过程中是否产生了溢出、负数等状态的相应位，也包含用于描述当前处理模式的模式位。关于处理器模式，因为代码 2-5 中没有涉及，所以我们在它们出现的时候进行详细讲述。

说完了寄存器，我们再来聊一聊指令。在汇编程序设计过程中，指令是我们构成汇编程序的基本单元。

指令包括指令助记符和伪指令。每一条指令助记符都代表 CPU 提供的某一条指令，也可以说指令助记符都唯一地对应了 CPU 的一条机器码。因



为机器码本身就是无规则的二进制数，本身不容易被记忆和使用，所以用助记符来帮助记忆，这其实也是“助记符”三个字的真正意思。

伪指令不像指令助记符那样是在程序运行期间由 CPU 来执行的，通常伪指令只会作用在编译过程中，对最终生成的文件造成不同程度的影响。有些伪指令在编译的时候并不生成代码，还有些伪指令则会扩展成一条或多条实际的指令。

下面我们就对代码 2-5 中所使用的指令和伪指令做深入的介绍。

(1) .arch 伪指令：该伪指令的作用是选择目标体系结构。例如，在代码 2-5 中，.arch 表示该段汇编代码将会被编译生成符合 armv4 体系结构的代码，从而实现了为特定平台生成特定的代码。如果不想使用该伪指令，那么在程序编译的时候，使用-march 参数也能达到同样的效果。

记忆会用
 (2) .global 伪指令：该伪指令的含义是让 global 过的符号对链接器可见，也就是说，一个函数或变量，通常情况下只在本文件内有效，当需要在外部引用该文件里的某一个函数或变量时，必须首先将该函数或变量使用.global 伪指令进行声明。在代码 2-5 中，helloworld 函数作为我们程序的入口函数，必须在链接时用-e 参数来指定，或者在链接脚本中用 ENTRY 命令来做显示声明。因此.global 伪指令在这里就显得至关重要了。

(3) .equ 伪指令：该伪指令其实很简单，相当于 C 中的宏定义。在代码 2-5 中，正是使用了.equ 伪指令将 0x50000020 用宏 REG_FIFO 来代替。

(4) .text 伪指令表示从当前位置开始的内容被归并到代码段中。

(5) .align 伪指令：我们并不是第一次遇到 align 这个词，回想一下链接脚本一节，关键字 ALIGN 的作用是在链接时，迫使被修饰的内容对齐。这里的.align 与 ALIGN 关键字意义相同，也能够更新位置计数器的值，使代码对齐到某一边界。但此处需要强调的是，该伪指令针对 ARM 汇编的用法与其他体系结构稍有不同。例如，在 m68k、sparc 以及运行有 ELF 文件格式的 x86 结构中，.align 后边的数字直接代表了要对齐的字节数，比如.align 8 表示此处代码会按照 8 字节边界对齐，而在 ARM 体系结构下，.align 后边的数是以幂的形式出现的，正如代码 2-5 中那样，.align 2 表示此处是以 4 字节对齐的。

(6) .ascii 伪指令：该伪指令用于在内存中定义字符串，我们要输出到串口中的字符串“helloworld\n”就是由它定义的。



(7) ldr 伪指令：很多朋友在看了前面的介绍后可能会得到一个错误结论，即所有伪指令都是以点开头的。其实不然，ldr 就是一个反例。ldr 被称为常量装载伪指令，其作用是将一个常量装载到寄存器中。因为 ARM 指令等宽指令格式的限制，不能保证所有的常数都可以通过一条指令装载到寄存器中。程序编译时，如果能够将 ldr 指令展开成一条常量装载指令，则编译器就会用该指令代替 ldr，否则编译器会首先开辟一段空间存储被装载的常量，然后使用一条存储器读取指令将该常量读入到寄存器当中。

(8) adr 伪指令：adr 伪指令被称做地址装载伪指令，与 ldr 类似，adr 伪指令能够将一个相对地址写入寄存器中。

至此，代码 2-5 中使用的全部伪指令我们就介绍完了。这里我们想强调的是，伪指令都是与编译器有关的，换句话说，即使是同一种硬件平台，GCC 中的伪指令与其他编译器中的伪指令也并不完全兼容。

我们前面说过，伪指令要么作用在编译过程中对最终生成的文件造成影响，要么会扩展成一条或多条实际的指令，这些都是在程序编译过程中由编译器决定的，因此，不同的编译器的伪指令集可能不同。而指令是与硬件平台有关的，即使不同的编译器对同一指令使用的助记符不一致，它们生成的机器码仍然是一致的，并且不同编译器所使用的指令助记符集几乎都彼此相同的，因为指令助记符都是由芯片生产厂商定义的，编译器大多会遵守这种定义。

说完了伪指令，我们来介绍一下代码 2-5 中的指令。

(1) 内存装载指令 ldr：ldr 如果作为实际指令出现，表示从内存读取数据到寄存器中，而代码中使用了一个 b 作为后缀，表示读取的只是一个字节的数据。相信读者能够举一反三，明白 ldrh 指令的含义。在该语句中，[r0] 表示地址值，而后边的#0x1 是常数 1，代表寄存器 R0 的增量，这种寻址方式被称为立即数后变址寻址。整条语句的意思是，首先从寄存器 R0 所指向的地址中读取一个字节的数据存储在 R2 中，然后 R0 中的值自加 1。因为在这之前 R0 的值已经指向了 helloworld 字符串的首地址，所以这条语句第一次执行，就会把字符 ‘h’ 存储在 R2 中。

(2) 内存存储指令 str：str 能够将寄存器中的值存储到另一个寄存器所指向的内存地址中。代码 2-5 正是使用了 str 指令将刚刚读取到的 R2 中的值存储到 R1 地址处，而该地址正是串口 FIFO 寄存器的地址。因此，字符

‘h’ 就会被显示到屏幕当中。

(3) 比较指令 cmp：比较指令是将待比较的两个数相减，然后去影响标志位，所以它实际上是一条不返回运算结果的减法指令。比较指令不保存结果，但是会使当前程序状态寄存器 CPSR 中零标志位置位或清除，进而影响条件判断语句的执行。

(4) 跳转指令 b：接下来我们看到的指令 bne 其实就是跳转指令 b，后缀 ne 是条件码，表示执行条件为“不相等”。整个语句可以解释为如果不相等，则跳转到符号.L2 处。这里的不相等指的正是上一条指令比较产生的结果。因为比较指令只影响标志位，所以我们可以说条件码只是根据相应标志位的值来有选择地执行。

程序在每次向串口 FIFO 寄存器写入数据后，都要与零进行比较。当比较的结果不相等，表示字符串还没有结尾，则继续将下一个字符写入到串口 FIFO 寄存器中；如果字符与零相等，证明读到了字符串结束符，然后程序进入死循环，这就是程序 2-5 的执行过程。

最后我们可以使用与代码 2-1 同样的编译方法来编译这段汇编版的 helloworld。

总结一下，现在我们已经学习了 ldr、str、cmp 和 b 这四条汇编指令，也学习了 arch、global、equ、text、align、ascii、ldr 和 adr 这八条伪指令。这些指令不但常用，而且很重要。在操作系统代码的编写过程中，也会被经常使用。

2.4 汇编和 C 的混合编程

很少有一款操作系统会完全使用汇编语言来编写，也几乎没有一个操作系统是完全使用 C 语言写成的。汇编语言负责编写启动、堆栈初始化、系统运行段划分等关键部分的代码，而 C 语言则被用在涉及到算法、逻辑等与硬件关联不大的地方，解决复杂的问题。因此需要将两者完美的结合，才能发挥出各自的优势。

这一节我们就来探讨一下 ARM 的混合编程问题，为我们以后正式编写

操作系统打下基础。

通俗地讲，无论是汇编语言、C 或是其他类型的语言，都是人能够理解的语言，都有特定的语法规则支持各种执行逻辑。从这一点出发，所有的语言（包括人类互相交流的语言），彼此之间都没有什么不同。而编译生成的机器码只有机器才能理解，并且不同的硬件平台彼此又语言不通，这样，问题似乎就来了，人能理解的语言机器理解不了，机器能理解的语言人又很难掌握，这个问题该怎样解决呢？

编译器能够帮助我们解决这个问题，编译器的作用就是将人能够理解的语言翻译成机器能够理解的语言，如图 2-1 所示。从这个角度看，把编译器叫做翻译器，多少也有些道理。

既然人能理解的语言五花八门，而机器能理解的语言对于特定平台又是唯一的，因而不同语言间的混合编程，就要从硬件的角度去理解才会有意义。

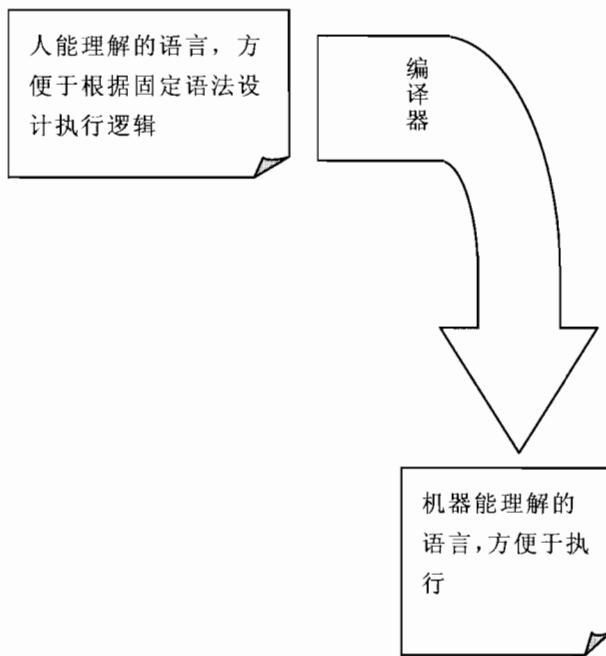


图 2-1 编译器的作用

2.4.1 过程调用标准

要想具体实现汇编语言和 C 语言之间的混合编程，就必须制定出一套

统一的标准并约束双方共同遵守，我们将这套共同遵守的标准称为过程调用标准（Procedure Call Standard），简称为 PCS。

过程调用标准其实就是一套规定，任何编程语言在应用的时候，只要去遵守这种约定，就能够与同样遵守该约定的其他编程语言和平共处。因此，想要让 C 语言和汇编语言能够混合编程并且正确执行，对过程调用标准的严格遵守就显得相当重要了。

其实，过程调用标准并没有具体针对任何语言，也就是说，无论什么编程语言，只要遵循这种标准，就可以直接调用其他同样遵守该标准的编程语言写出来的程序，而不一定非要限制在 C 语言或汇编语言上。ARM 的 PCS 有很多版本，如 ARM 过程调用标准 APCS、Thumb 过程调用标准 TPCS，以及 ARM-Thumb 之间互相调用时所要遵守的过程调用标准 ATPCS。目前“最新”的一套标准名为 ARM 体系结构过程调用标准（Procedure Call Standard for the ARM Architecture），简称 AAPCS，该标准定义了 ARM 下的若干个子例程如何才能实现独立编写、独立编译、独立汇编，却可以共同运行。

我们依旧会借助一个实际例子来介绍一下一些基本的过程调用规则，看看如何实现 C 语言和汇编语言的混合编程。

2.4.2 混合编程的例子

代码 2-6

```
.arch armv4
.global _start

.equ REG_FIFO, 0x50000020

.text
.align 2

_start:
    ldr r0,=REG_FIFO
    adr r1,.L0
    bl helloworld
.L1:
```



```
b .L1  
.align 2  
.L0:  
.ascii "helloworld\n\0"
```

代码 2-6 是一段汇编程序，它与代码 2-5 很类似。二者的区别在于代码 2-6 不是直接将 helloworld 字符串送给串口 FIFO 寄存器，而是通过 bl 指令调用 helloworld 函数，由该函数负责数据写入。

bl 指令我们没有接触过，它的功能与 b 指令几乎一致。唯一的不同是 bl 指令在运行时，能够自动地保存程序的返回地址。这样，在成功调用子程序并且运行结束时，就能够返回原先的位置继续运行。

helloworld 函数是由 C 语言写成的。根据 AAPCS 的规定，当函数发生调用的时候，函数的参数会保存在 ARM 核心寄存器 R0~R3 中，如果参数多于 4 个，则剩下的参数会保存在堆栈中。因此，我们也会把寄存器 R0~R3 称为参数寄存器 (argument register)，用别名 a1~a4 代替。在代码 2-6 中，调用 helloworld 函数之前，我们分别对 R0、R1 两个寄存器赋值，这两个值就是 helloworld 函数的参数。

下面就让我们来了解一下这个函数具体是怎样实现的。

代码 2-7

```
int helloworld(unsigned int *addr,const char *p){  
    while(*p){  
        *addr=*p++;  
    }  
    return 0;  
}
```

代码 2-7 类似于之前用 C 语言实现的 helloworld 函数。不同的是该函数多了两个参数，分别是内存地址和要写向该地址的字符串的首地址。

整个函数的功能是将指针 p 所指向的字符串写向由 addr 代表的内存地址中。当函数 helloworld 执行完毕后，程序会返回到代码 2-6 的.L1 地址处继续运行。helloworld 函数带有一个 int 型返回值，该返回值会保存到核心寄存器 R0 中，如果返回的是一个 64 位的值，则该值会由 R0、R1 同时保存。很显然这也是由 AAPCS 所规定的。

这两段程序在编译的时候会稍有不同，首先我们必须分别编译这两段程

序，产生两个目标文件。

将代码 2-6 命名为“start.s”、代码 2-7 命名为“helloworld.c”。接着我们要使用类似于命令 2-1 的命令来生成两个目标文件。然后，使用命令 2-5 来编译生成 ELF 格式的可执行文件。

命令 2-5

```
arm-elf-ld -e start -Ttext 0x0 start.o helloworld.o -o helloworld
```

在终端运行 skyeye 命令，我们可以得到与代码 2-1 相同的结果。这表示我们已经成功实现了 C 和汇编的混合编程。

2.5 Makefile

在这一小节中，我们将介绍一种有效的编译手段来提高代码的编译效率。

我们看到，在前面的所有例子中，程序都是采用一种“单打独斗”的编译方法来编译的。程序在编写完成后，需要程序员手动运行命令去编译链接，最终生成可以运行的程序。当然，如果只是编译一两个文件，这似乎并不是什么问题。但一旦程序的规模扩大，需要编译的文件增多，程序编译过程就将会给我们带来极大的负担。

因此，这样的编译方式很显然不合适对操作系统进行。幸好 make 工具可以帮助我们解决所有的问题。

make 工具能够自动地按照我们的要求依次编译指定的源程序，在项目工程中是一个不可或缺的工具。make 工具功能非常强大，而在使用 make 工具的同时，我们还需要写一个 Makefile 文件。顾名思义，Makefile 文件就是专门供 make 命令使用的文件，里面记录的都是编译源代码的具体细节。这里我们给出了相对通用的 Makefile 文件模板，内容如下。

代码 2-8

```
CC=arm-elf-gcc
LD=arm-elf-ld
OBJCOPY=arm-elf-objcopy
```

```
CFLAGS= -O2 -g  
ASFLAGS= -O2 -g  
LDFLAGS=-Tleeos.lds -Ttext 30000000  
  
OBJS=start.o helloworld.o  
  
.c.o:  
    $(CC) $(CFLAGS) -c $<  
.s.o:  
    $(CC) $(ASFLAGS) -c $<  
  
leeos:$(OBJS)  
    $(CC) -static -nostartfiles -nostdlib $(LDFLAGS) $? -o $@  
lgcc  
    $(OBJCOPY) -O binary $@ leeos.bin  
  
clean:  
    rm *.o leeos leeos.bin -f
```

对于不熟悉 Linux 或 Makefile 的读者来说，这段代码就像天书一样。这里我们不会深入研究 Makefile 的写法，只需要知道其用法即可。毕竟这些知识离操作系统的原理还是比较远的。

代码 2-8 可以作为一个模板，只需要稍做修改就能使用。具体的方法是，如果我们要添加并编译新的源文件，只需要在变量 OBJ 中增加与源文件同名的目标文件即可，仅此而已，这个文件就可以用来编译我们自己的操作系统了，其他参数无须修改。

如果读者想要在其他场合下使用该模板，那么当需要修改编译选项时，可以将其添加到变量 CFLAGS 或 ASFLAGS 中，LDFLAGS 变量代表链接选项，同样可以根据实际情况更改。CC、LD、OBJCOPY 则代表编译工具，也可以自行确定。

这里需要强调的是，在编译的过程中，除-nostdlib 之外，我们使用了另一个不太常用的 GCC 编译选项，那就是-nostartfiles。简单地说，这个选项的作用就是不允许编译器使用默认的启动文件。因为我们要写的是一个操作系统，系统的启动是自行处理的，不需要编译器帮我们添加。同时，在程序中我们通过-lgcc 在编译时链接 libgcc.a 这一函数库，这样，ARM 中的一些基本运算（如除法）就不需要我们在操作系统中亲自实现了。

将代码 2-8 保存到名为 Makefile 的文件中。此时我们便可以进行编译了，确保所有的文件都在同一个目录下，打开一个终端，切换到源代码所在目录，同时运行如下命令。

命令 2-6

```
make
```

这样就会得到与上一节同样的结果，所付出的仅仅是适当修改 Makefile 文件中的 OBJS 变量的内容，并在终端敲四个字母。

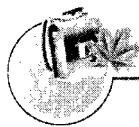
如果想要同时删除所有编译生成的文件，只保留程序代码，可以通过如下命令来实现。

命令 2-7

```
make clean
```

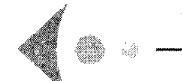
2.6 总结

目前，我们完成的工作并不多，但至少为整个系统的完成提供了一个结构或者说一种解决的方法。在以后的工作中，我们无非也是利用类似的方法和技巧，结合我们对操作系统的理解，一步一步地实现属于我们自己的嵌入式操作系统！



第3章

操作系统的启动



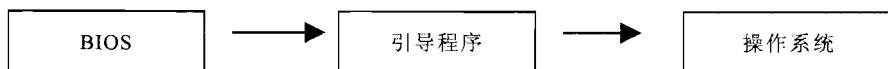
操作系统的启动与硬件体系结构密不可分。一个操作系统究竟应该按照什么样的方式启动并没有一套统一的标准。因此我们可以说，操作系统的启动过程是一个与操作系统自身无关的过程，但却是一个与硬件体系结构相关的过程。

这句话听起来不好理解，却能够恰当表达操作系统启动的本质。举个例子来说，同样是 Linux 操作系统，在 x86 体系结构下与在 ARM 体系结构下启动是完全不同的，这就是所谓的“启动过程与操作系统本身无关”，而 Windows 和 Linux 虽然分属不同的操作系统，但如果两个操作系统都在 x86 下运行，其启动过程从结构上来讲仍然是类似的，这就是所谓的“启动过程与硬件体系结构相关”。

但是，这里似乎忽略了一个很重要的概念——操作系统应该怎样界定，或者说操作系统是从上电那一刻开始算起呢还是从其他时刻开始算起？想要说清楚这个问题，我们首先来讨论一下操作系统的启动流程。

3.1 启动流程

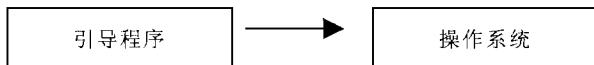
让我们先从计算机说起，计算机的启动流程分为如下三个阶段：



BIOS (Basic Input/Output System) 程序是指固化到主板上 BIOS 芯片内的一段初始启动程序，是系统启动时运行的真正意义上的第一段程序。在计算机中，BIOS 程序主要负责硬件自检和初始化、中断处理程序的基本设置，以及从存储器中加载零道零扇区的代码到内存并从该位置开始运行。而存储在外存零道零扇区的代码，通常就是所谓的引导程序了。

简而言之，引导程序就是用来引导操作系统的。由于多数操作系统都要存储在外存中而运行于内存。因此，需要一段小程序能够将操作系统从外存搬运到内存之中并交出 CPU 的控制权，这就是引导程序存在的意义。引导程序与操作系统有很大的关系，有些特定的引导程序只能引导特定的系统，但他们仍然不是操作系统本身。引导程序的作用就是给被引导的操作系统提供运行时所需的环境和条件，然后从外存中把真正的操作系统加载到内存中运行。

以上就是一个普通计算机的启动过程，但这样的过程并不一定适合于嵌入式系统。在很多嵌入式平台中，人们会将计算机的三个启动阶段简化为两个：

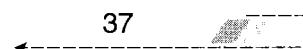


这里，引导程序通常会固化到 Flash 或 ROM 存储器中，当系统上电时会首先被运行，起到初始化基本硬件和引导操作系统的双重作用。

还有一些更加简单的嵌入式操作系统，不需要额外的引导程序就可以直接运行对硬件的初始化等工作，作为该操作系统的一部分而发挥作用。

以上是有关操作系统启动的最简要的介绍，有些读者可能会产生如下疑问，如果 BIOS 或引导程序已经对板载硬件完成了初始化，操作系统在被引导以后是否需要重复执行这个工作？例如，引导程序已经完成了串口设备的初始化工作，在操作系统运行时，是否需要重新初始化串口？

这个问题的答案不言而喻，操作系统是有必要对所有硬件重新进行初始化的。究其原因，可以总结为一点：我们不能让操作系统只适应于特定的引导程序，或者说我们希望任何引导程序都能引导我们的操作系统。但引导程序五花八门，有的简单，有的复杂，初始化过程又各不相同，如果将启动时需要进行的初始化任务全权交给引导程序负责，势必会降低操作系统的



通用性，容易产生错误。因此比较保险的做法是，无论引导程序是如何初始化的，操作系统自身的初始化过程都要重新进行。

既然如此，那么就可以按照上述原则对操作系统从何处开始这一问题做界定了。我们的操作系统开始于一个基础代码可以运行的环境，处在板载硬件未被初始化的状态，也就是说，我们默认已有一个引导程序保证了我们的操作系统能够出现在内存中并且正确运行，但该引导程序不负责环境的任何初始化工作。

3.1.1 ARM 的启动过程

接下来我们就来聊聊 ARM 的启动。

ARM 的启动其实再简单不过了。在 CPU 上电的时候，ARM 就会首先从地址零处开始运行，第 2 章的几个例子程序都说明了这一点。因为是精简指令集，同时硬件结构较新，所以不需要考虑让人挠墙的历史兼容问题。这些都使程序在 ARM 体系结构下的启动变得简洁而自然。但是作为操作系统，我们不能简简单单地让程序能够运行就了事了。操作系统的根本目的之一是维护和保障其他程序能够依托本系统顺利工作。因此，区别于简单的裸机程序，操作系统在启动时应至少关心如下两个方面。

1. 程序运行栈的初始化。程序的运行离不开栈，程序运行时所需的临时变量、数据、函数间调用的参数传递等都需要栈的支持。如果是在操作系统之上进行编程，堆栈的问题往往不需要程序员关心，但对于操作系统本身而言，需要对用户应用程序提供统一的运行环境和资源，堆栈的处理就不可避免了。因此操作系统在启动时，必须首先解决好堆栈的初始化问题。

2. 处理器及外设的初始化。一个操作系统能够正常运行，处理器的正确配置和外设的正常工作也是必不可少的先决条件。例如，中断处理程序能够提供给操作系统非顺序的、突发的执行逻辑。时钟设备则可以产生固定频率的时钟驱动整个系统运行。所以在操作系统各功能模块得以发挥作用之前，处理器及外设也必须被初始化。

上述两条原则只是理论上的，不同的操作系统运行在不同的平台上，实现方法可能各不相同，但无论如何这两条原则都是要遵守的。

3.1.2 ARM 操作系统解读

接下来我们来看一段例子，研究一下 ARM 体系结构下的操作系统在初始化的过程当中，应该如何实现。

代码 3-1

ARM 指令

```

.section .startup, "ax"
.code 32
.align 0

    b    _start      /* reset - _start          */
    ldr pc, _undf   /* undefined - _undf        */
    ldr pc, _swi    /* SWI - _swi              */
    ldr pc, _pabt   /* program abort - _pabt   */
    ldr pc, _dabt   /* data abort - _dabt     */
    nop             /* reserved                */
    ldr pc, [pc,#-0xFF0] /* IRQ - read the VIC      */
    ldr pc, _fiq    /* FIQ - _fiq              */

    .undf: .word _undf      /* undefined                */
    .swi:  .word vPortYieldProcessor /* SWI                      */
    .pabt: .word _pabt      /* program abort            */
    .dabt: .word _dabt      /* data abort               */
    .fiq:  .word _fiq       /* FIQ                     */

    _undf: b  _undf      /* undefined                */
    _pabt: b  _pabt      /* program abort            */
    _dabt: b  _dabt      /* data abort               */
    _fiq:  b  _fiq       /* FIQ                     */

```

这一段代码节选自开源嵌入式操作系统 freeRTOS 的初始化部分，与该段代码类似的初始化方式广泛应用于支持 ARM 的各种类型的操作系统和引导程序中，因此非常具有代表性。

代码一开始使用了三条伪指令，其中，.align 的含义和用法我们已经介绍过了（参见第 2 章）。

.section 的作用是声明接下来的代码属于该段，其格式为.section name[, " flags "]或.section name[, subsection]。代码 3-1 中采用的是第一种形式，将整段代码编译到 startup 段中，该段也是整个操作系统的入口。

而方括号内的内容是可选的，代码中该标志被定义为 `ax`，其中的“`a`”代表可重定位（`section is allocatable`），而“`x`”的意思是可执行（`section is executable`）。

另一个伪指令`.code 32`的作用是编译生成 32 位指令集的代码。`.code` 的另一种形式是`.code 16`，其作用是生成 16 位指令集的代码。伪指令`.arm` 以及`.thumb` 分别与`.code 32` 和`.code 16` 效果相同。我们知道，在传统的 ARM 体系结构中既可以使用 32 位的指令，又可以使用 16 位的指令，此处正好与 ARM 的这套指令系统相统一。

接下来，代码 3-1 将执行该程序的第一条指令——跳转指令，通过该指令程序将直接跳转至`_start` 处继续运行。既然如此，紧接着跳转指令的其他指令在什么情况下才能运行呢？熟悉 ARM 体系结构的读者可能清楚，包括第一条指令在内，这段代码正是异常向量表。也就是说，所有其他指令都会在异常模式发生时才会运行。

对 ARM 不了解的读者可能会问，异常模式是何物？简单地说，异常模式就是程序由于某种原因执行不正常时，系统所处于的一种运行模式。当异常模式出现时，预先指定的代码将会被运行，从而可以尽可能地解决异常并返回到正常模式中。

在 ARM v6 及以前的体系中定义了七种模式，分别是管理模式（SVC）、快速中断模式（FIQ）、中断模式（IRQ）、未定义模式（UND）、终止模式（ABT）、用户模式（USR）以及系统模式（SYS）。其中，前五种就是所谓的异常模式。每一种异常模式都拥有私有的堆栈指针寄存器 R13、返回地址寄存器 R14 以及模式备份寄存器 SPSR。一段代码只有处在该异常模式下时才有权力访问属于该异常模式的这三个私有寄存器。当某种异常发生时，CPU 会立刻停止当前正在执行的动作，转而去运行预先指定的代码，同时，系统会切换到特定模式中，这是一种被动的模式切换方法。除此之外，程序员还可以通过编程主动进行模式切换，使用的方法是更改 CPSR 寄存器中的模式位。有关异常模式的细节我们将会在第 5 章详细阐述。

既然程序在启动时，就立刻跳转到`_start` 处运行，那么在`_start` 位置上 freeRTOS 又做了哪些工作呢？我们接着来看`_start` 处的代码。

代码 3-2

```

_start:
    ldr r0, .LC6
    msr CPSR, #MODE_UND|I_BIT|F_BIT
    mov sp, r0
    ...
    sub r0, r0, #SVC_STACK_SIZE r0 = r0 - size
    msr CPSR, #MODE_SYS|I_BIT|F_BIT /* System Mode */
    mov sp, r0

    msr CPSR, #MODE_SVC|I_BIT|F_BIT

    /* Clear BSS. */

    mov a2, #0      /* Fill value */
    mov r12, a2     /* Null frame pointer */
    mov r7, a2      /* Null frame pointer for Thumb */

    ldr r1, .LC1    /* Start of memory block */
    ldr r3, .LC2    /* End of memory block */
    subs r3, r3, r1 /* Length of block */
    beq .end_clear_loop
    mov r2, #0

.clear_loop:
    strb r2, [r1], #1
    subs r3, r3, #1
    bgt .clear_loop
.end_clear_loop:

    mov r0, #0      /* no arguments */
    mov r1, #0      /* no argv either */

    bl main
.LC1:
.word bss_beg
.LC2:
.word bss_end

.LC6:
.word stack_end

```

标志与 M RF

中止权限

r0 = r0 - size

BSS_start

bss_end

代码 3-2 从_start 处开始，一步步进行栈指针和 bss 段的初始化工作，最终通过一条 bl 指令跳转到主程序处继续运行。示例代码删减了相对重复和不具有代表性的内容，以使代码结构更清晰。

程序一开始，将局部变量.LC6 的值加载到寄存器中，也就是`_stack_end_`这一变量的值。从名字上我们可以判断出，该值应该是程序堆栈的顶端。问题似乎随之而来，从理论上讲，任何 CPU 都必须使用一个专有寄存器来指向程序当前的堆栈地址。在 ARM 中寄存器 R13 充当了这一角色，在 x86 下，我们可以通过 ESP 寄存器实现类似的功能。无论什么体系结构，在处理堆栈指针寄存器时都必须面对两个问题：堆栈如何增长？堆栈为满堆栈还是空堆栈？

图 3-1 直观地描述了这两个问题的四种组合情况，所谓堆栈如何增长，指的是堆栈指针更新时，是向着高地址方向还是低地址方向，而所谓满堆栈和空堆栈，指的是当程序利用堆栈指针向堆栈中存储数据或从堆栈中提取数据时，是先将数据存入或读出再更新堆栈指针还是先更新堆栈指针，然后再存取数据，也就是说，当前堆栈指针是否指向了有效数据。因此，堆栈的处理方法可以总结为四种，分别为空递增堆栈、满递增堆栈、空递减堆栈、满递减堆栈。读者可以将这四种处理方法在图 3-1 中一一对号入座。

其实无论堆栈的处理有多少种方法，只要保证所有的函数在保存局部变量或相互调用过程中始终使用同一种处理方法，程序的运行就不会出问题。因为堆栈的使用在汇编语言下是可控的，也就是说，数据向堆栈的存取和堆栈指针寄存器的更新是需要程序员自行处理的。因此，如果所有的程序都使用汇编写成，我们可以直接选择一种处理方式来处理堆栈，就不会出问题了。但如果使用其他语言编程，堆栈的处理已经被编译器屏蔽，程序员并不知道编译器选择了哪种方式，这样编译出来的程序就会存在隐患。例如，使用 C 语言编程，并且编译器是按照满递减的方式处理源代码的，但我们却错误地在操作系统初始化时将堆栈指针寄存器设置成内存的最低地址，一旦该程序使用堆栈，便会出现数据异常的情况。

那这个问题该如何解决呢？答案是，通过过程调用标准来解决。

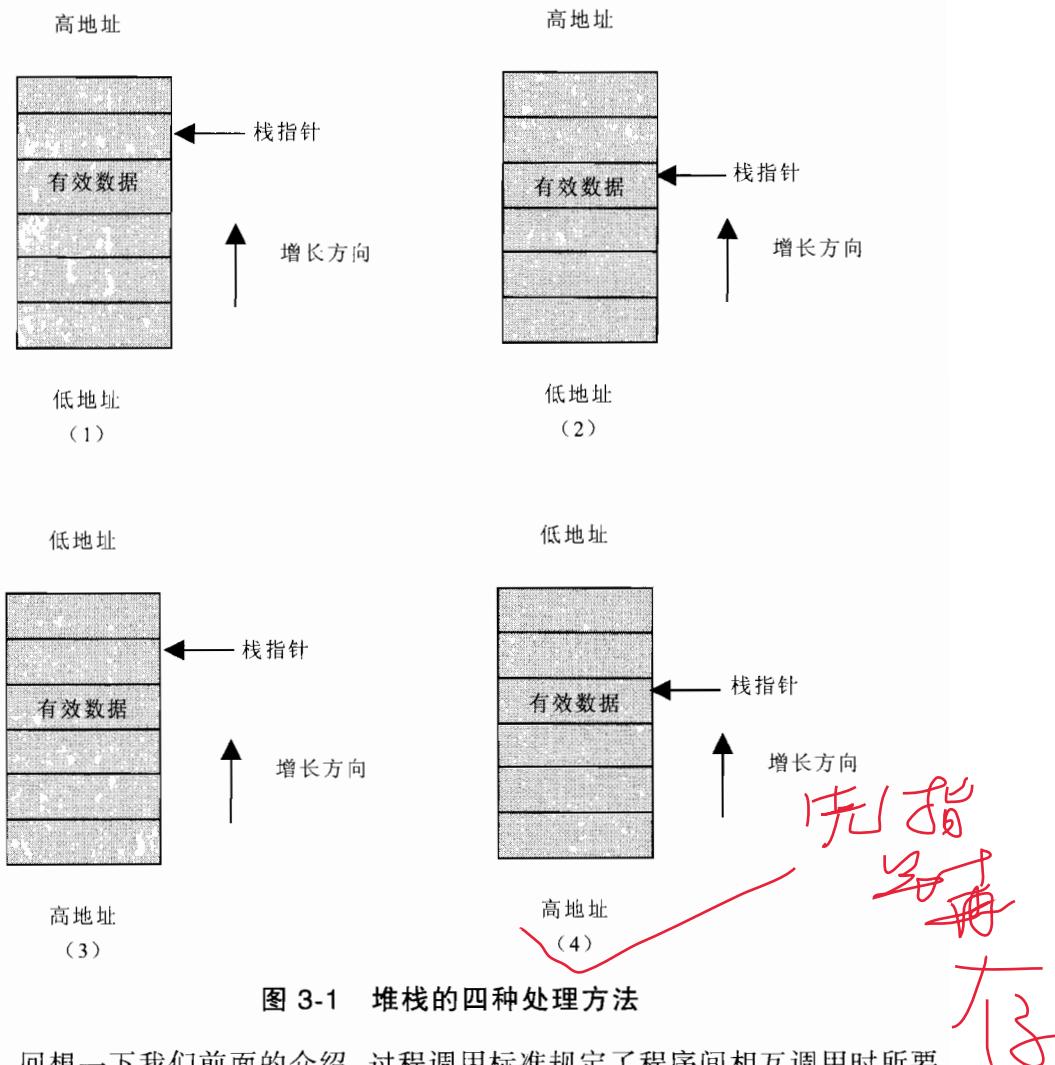


图 3-1 堆栈的四种处理方法

回想一下我们前面的介绍，过程调用标准规定了程序间相互调用时所要遵守的规则，无论程序来源于哪、使用什么语言写成，只要共同遵守某一规则，程序间就可以相安无事、各司其职。也就是说，堆栈工作方式在过程调用标准当中会被明确地规定。AAPCS 中明确指出使用满递减堆栈方式处理堆栈。这表示操作系统在堆栈初始化阶段，必须将堆栈指针寄存器赋予一个高端的内存地址值，在代码 3-2 中，该值是内存的最高地址值减 4。

回到代码 3-2 中，R0 被赋值之后，紧接着程序通过一条 msr 指令更改当前处理器模式为未定义指令模式。在 ARM 的 7 种处理器模式中，除用户模式和系统模式共用一个 R13 寄存器外，其他异常模式都有独立的 R13 寄

存器，这表示各模式的堆栈是彼此独立的。同时，因为不同模式之间的私有寄存器不能够互相访问，因此，想要对不同模式的 R13 寄存器分别进行初始化，必须首先切换到相应模式中，然后再对 R13 寄存器赋值。而 msr 指令便能实现模式切换，其完整格式如下。

```
msr{cond} psr_fields,#immed_8r
```

或

```
msr{cond} psr_fields,Rm
```

显然，代码 3-2 采用的是第一种格式，将一个立即数赋值给 CPSR 寄存器的控制域，其结果是将处理器模式由启动时默认的 SVC 模式切换到未定义模式。在未定义模式中，程序将寄存器 R0 的值赋给堆栈指针寄存器 SP，从而完成了一次异常模式的堆栈初始化工作。接下来，类似的代码被重复执行，最终完成处理器各个模式的堆栈初始化。

在完成堆栈初始化后，代码 3-2 进入下一阶段，清除掉 bss 段的数据。bss 段通常是用来存放程序中未初始化或初始值为零的全局变量或静态变量的一块内存区域。与 data 段不同，bss 段内的所有数据都没有初始值或初始值为零。也就是说，在编译程序的过程中，没有必要将 bss 段的内容生成在可执行程序中，这意味着如果我们将无所谓初始值的变量都空下来不赋值或赋值为零，将会大大减少可执行程序的尺寸，这一点对于对文件大小敏感的程序来说很有意义。当然，有利必有弊，虽然存储空间被节省了，可是这些变量的初始值就不得不在运行时赋值了。换句话说，程序的启动速度会受到一定程度的影响。

有些读者可能会有这样的疑问，对于那些没有初始值的数据，是不是有必要仍然将其初始化成零？其实，这是 C 语言标准所规定的，所有的变量都应该被初始化。具体地说，所有的指针都应该被初始化为空指针，所有的算术类型数据，都应该被初始化成零。正因为如此，即使程序在整个运行期间不会使用零这个值，在操作系统启动的过程中，却仍然需要对 bss 段进行初始化。

于是在代码 3-2 里，程序将 bss 段起始地址和结束地址读出，并算出 bss 段的大小，然后在.clear_loop 循环内，逐个对 bss 段的内容赋值。其中用到的一些汇编指令是前面没有提过的，下面具体介绍一下。

(1) sub 指令是减法指令，几乎所有的体系结构都支持这种指令并使用

同样的指令助记符。我们在这里看到的减法指令稍有变化，是在 sub 后边又多加了一个 s，意思是根据该指令运行的结果来更新 CPSR 寄存器的标志位。减法指令的标准格式是：sub{<cond>}{s} Rd, Rn, N。它可以将寄存器 Rn 减 N 的结果保存在寄存器 Rd 中，并且根据是否带有后缀 s 来选择更改标志位。举例来说，“subs r3,r3,#1” 将会使 R3 寄存器的值减一，当减到零时，CPSR 中的零标志位会被置 1。

(2) b 指令是我们早已熟悉的了，但是 b 后边的 gt 后缀我们却是头一次接触。这里的 gt 充当了条件助记符的作用，表示如果“有符号数大于”这一条件发生时便执行跳转。这里所谓的“有符号数大于”针对的是上一条减法指令。无论 Rn 和 N 是正是负，当二者相减的结果大于零时，该条件就会发生。而在代码 3-2 中，这一条件其实是表示 R3 的值并未递减到零，因此需要跳转回 .clear_loop 处继续将下一个位置清零，直至 bss 段末尾。

代码 3-2 中有一个细节需要注意，那就是.word 伪指令。这个伪指令可以用来在内存中分配一个 4 字节的空间，并可以同时对其初始化。例如，代码中的“.LC6: .word __stack_end__”，就是在定义空间的同时，又向该空间赋予了 __stack_end__ 这一变量的值，这样就可以直接在程序中引用.LC6 了。

在所有初始化过程完成后，程序跳转到 main 函数处继续接下来的工作。

对于 freeRTOS 来说，程序在进入 main 函数后立即执行 prvSetupHardware 函数，完成特定硬件平台的初始化工作，这些初始化工作因为与平台相关，因此程序本身不具有代表性，但程序执行的基本流程仍然是确定的，总结起来有如下两个方面。

(1) 时钟初始化。时钟的频率直接决定了 CPU 的性能，因此如果条件允许，我们都会尽量提高 CPU 的时钟频率，从而提高效率。但是这样，问题也就来了，一般 CPU 内部时钟都是可编程的，时钟的本源往往来源于外部晶振，很多 CPU 允许外接的晶振频率是可选的，而我们必须根据晶振的频率计算出相应的值来产生正确的 CPU 时钟。因此，CPU 通常不是一上电就立刻工作在一个较高的频率下的，而是首先以一个相对较低的频率运行，之后通过编程的方法进行配置，才可以使用高频时钟。除此之外，CPU 内部集成的外围器件或控制器也需要不同频率的时钟，这些时钟频率远远低于 CPU 的工作频率，因此需要通过分频的方法获得。无论怎样，时钟就是机

器的心跳，是不可或缺的。对时钟的初始化几乎是任何一款操作系统初始化阶段的必要步骤。

(2) 内存初始化。多数 CPU 与内存的耦合程度并不是特别紧密。通俗地讲，多数 CPU 不需要特别复杂的硬件辅助就可以直接与内存连接。但是，这不能表示已经连接上的内存处于一个可用或者好用的状态。不同的内存芯片、厂商、型号类型在不同的板子上都是不同的，通常我们需要根据实际情况，专门对内存进行配置以适应这种环境。由于内存没有正确配置之前很可能不能正常使用，或者仅能以很低的效率被使用。因此，初始化内存就成了操作系统初始化时又一必不可少的工作。

除了上述必须完成的工作之外，一款成熟的操作系统还可能包含一些可选的初始化步骤，比如输入输出设备、存储设备的初始化等。只有操作系统赖以生存的环境被正确并且完整的配置后，操作系统才能发挥出它真正的威力。

3.1.3 正式开始写操作系统

前面我们分析了一段基于 ARM 的典型初始化程序并总结了系统初始化的一般步骤，做到了知其然并知其所以然。接下来就利用我们学到的知识和原理，正式开始编写属于我们自己的操作系统。请打开一款代码编辑器，并输入下面这段代码。

代码 3-3

```
.section .startup
.code 32
.align 0
.global _start
.extern __vector_reset
.extern __vector_undefined
.extern __vector_swí
.extern __vector_prefetch_abort
.extern __vector_data_abort
.extern __vector_reserved
.extern __vector_irq
.extern __vector_fiq
```

```

_start:

ldr pc, _vector_reset
ldr pc, _vector_undefined
ldr pc, _vector_sw
ldr pc, _vector_prefetch_abort
ldr pc, _vector_data_abort
ldr pc, _vector_reserved
ldr pc, _vector_irq
ldr pc, _vector_fiq

.align 4
:

_vector_reset: .word _vector_reset
_vector_undefined: .word _vector_undefined
_vector_sw: .word _vector_sw
_vector_prefetch_abort: .word _vector_prefetch_abort
_vector_data_abort: .word _vector_data_abort
_vector_reserved: .word _vector_reserved
_vector_irq: .word _vector_irq
_vector_fiq: .word _vector_fiq

```

有些读者可能已经意识到，以前从未接触过或很少接触的汇编程序，现在看起来似乎已非常简单。这说明我们已经在实践中掌握了一定的 ARM 汇编知识。

这段代码我们无须解释，但需要辨析一下不同的跳转指令之间的区别。在 ARM 汇编程序中，要实现程序间跳转有三种方法可供使用：

(1) 使用跳转指令 b、bl、bx 等。使用这一系列指令的优点是执行速度快，只需要一个指令周期即可完成跳转。但该系列指令有一个明显的缺点，那就是它们都不能实现对任意地址的跳转。比如说程序在地址 0x0 处运行了跳转指令，但是该指令不能跳转到 0xc0000000 处去运行，这是由 ARM 指令等宽特性决定的。ARM 指令集是 32 位等长的，所以，所有的指令（包括指令的参数在内）都必须在 4 字节的范围内完成。这样，当一条指令需要附带一个立即数或一个地址值作为参数时，该立即数或地址值必然要小于 32 位。事实上，跳转指令 b 所能跳转的最大范围是当前指令地址前后的 32M。
(x-16, x+16)

(2) 使用内存装载指令，将存储在内存的某一地址装载到程序计数器 PC 中。例如，若我们将跳转的目的地址存储在高于当前地址 24 字节处，就



可以使用“ldr pc, [pc,#24]”这条指令实现跳转。当然，在实际编码中，我们并不一定要亲自做这种偏移量的计算，而可以采用代码 3-3 的方法，用一条 ldr 伪指令来实现。这样，编译器会根据情况将该指令展开成 b 指令或 ldr 指令。ldr 指令能够实现任意地址的跳转，但因为需要读写存储器，所以在执行速度上与 b 指令相比就稍逊一筹了。

(3) 第三种跳转方法是通过 mov 指令来实现的。mov 指令是最简单的移动指令，几乎所有的处理器都包含这条指令。通过 mov 指令将已经保存到某一寄存器中的地址值直接赋值给程序计数器 PC，也可以实现程序跳转。使用该方法仍然需要额外的指令或别的手段，将跳转地址预先保存到寄存器中。因此这种方法多用在函数返回时的跳转中。

了解了 ARM 中的跳转方法之后，相信读者已能在编程过程中灵活应用它们了。但是，代码 3-3 是只有当异常模式发生时才会运行的。正常情况下，程序一开始就会马上跳转到 `_vector_reset` 处运行。

下面就让我们跟随程序的执行过程，继续完善我们的操作系统。

代码 3-4

```
.text
.code 32
.global _vector_reset

.extern plat_boot
.extern __bss_start__
.extern __bss_end__

_vector_reset:
    msr cpsr_c, #(DISABLE_IRQ|DISABLE_FIQ|SVC_MOD)
    ldr sp,=_SVC_STACK

    msr cpsr_c, #(DISABLE_IRQ|DISABLE_FIQ|IRQ_MOD)
    ldr sp,=_IRQ_STACK

    msr cpsr_c, #(DISABLE_IRQ|DISABLE_FIQ|FIQ_MOD)
    ldr sp,=_FIQ_STACK

    msr cpsr_c, #(DISABLE_IRQ|DISABLE_FIQ|ABT_MOD)
    ldr sp,=_ABT_STACK
```

```

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|UND_MOD)
    ldr sp,=_UND_STACK

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|SYS_MOD)
    ldr sp,=_SYS_STACK

    _clear_bss:
        ldr r1,_bss_start_
        ldr r3,_bss_end_
        mov r2,#0x0
    l1:
        cmp r1,r3
        beq _main
        str r2,[r1],#0x4
        b l1

    _main:
        b plat_boot

    _bss_start_:.word  _bss_start_
    _bss_end_:.word  _bss_end_
    .end

```

在正常模式下，程序会进入 `_vector_reset` 并运行，而后分别进行堆栈寄存器的初始化和 bss 段的清除工作。这里有一个细节需要强调，那就是对 `_bss_start_` 和 `_bss_end_` 两个变量的定义，在代码 3-4 的开始部分我们用 `extern` 关键字声明了这两个变量，这表明这些变量来自于别处。但问题是 bss 段的开始和结束的位置是动态的，程序稍有修改就会产生变化，因此我们很难确定一个固定值。那么这两个值到底是如何确定的呢？

答案是，由编译器确定。回想一下第 2 章关于链接脚本的内容，bss 段、data 段和 text 段的具体位置都是在链接脚本中指定的，也就是说，编译器在编译的时候会计算每段程序的代码、未初始化的全局变量和已经初始化的全局变量，并根据链接脚本安排各段的大小和位置。

读者也许还是会问，就算能够确定 bss 段的起始位置和结束位置，那这两个值又是如何传递的呢？

这个问题的答案还是在于链接脚本，看看下面这段代码，读者朋友们就会明白了。

代码 3-5

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
    .text :
    {
        *(.startup)
        *(.text)
    }
    . = ALIGN(32);

    .data :
    {
        *(.data)
    }

    . = ALIGN(32);
    __bss_start__ = .;
    .bss :
    {
        *(.bss)
    }
    __bss_end__ = .;
}
```

读者可能已经在代码 3-5 中发现了 `__bss_start__` 和 `__bss_end__` 两个变量。在这段链接脚本中，我们将 `__bss_start__` 变量赋予了当前位置值，紧接着的就是 bss 段。因此，`__bss_start__` 变量恰好代表了 bss 段的起始地址，而 `__bss_end__` 变量也是通过同样的方式赋值的，最终这两个变量在代码 3-4 中被引用。

回到代码 3-4 中，在对 bss 段的清零工作完成后，代码通过一条跳转指令跳转到函数 `plat_boot` 处执行，开始外围硬件的初始化工作。

由于操作系统在设计之初就兼顾了自身的可移植性，因此，板载硬件的初始化工作对于不同的平台，虽然毫无相同之处，但却必须要追求一种统一。这种不同平台的统一其实是一种抽象，在程序设计当中，好的抽象能够以最高的效率、通过最少的代码修改实现最大程度的兼容，可以说软件设计能力其实就是一种抽象能力。可能有些读者会觉得我的这段话也很抽象，那就让

我们直接看代码，体会一下抽象软件设计的优势吧。

代码 3-6

```
static init_func init[] = {
    arm920t_init_mmu,
    s3c2410_init_clock,
    s3c2410_init_memory,
    s3c2410_init_irq,
    s3c2410_init_io,
    NULL
};

void plat_boot(void) {
    load_init_boot(init);
}
```

这段代码其实很简单，首先在程序中定义了一个名为 `init`、类型为 `init_func` 的数组。`init_func` 类型其实是一种函数类型，其定义如下：

代码 3-7

```
typedef void (*init_func)(void);
```

从定义中我们可以看出，`init_func` 类型的函数既没有参数也没有返回值。初始化函数不需要参数，是因为它们只需要被调用一次，并不需要通过参数的动态改变来执行内容，也不需要返回值，因为这些函数都不允许执行失败，即便能够返回执行失败的结果，在初始化阶段也没有能力补救。但不管怎样，正是 `init` 数组中的各个成员函数完成了对各种硬件的具体的初始化工作。在 `plat_boot` 函数中，将 `init` 数组作为参数传递给了 `load_init_boot` 函数，而该函数便是这个数组成员函数的实施者，让我们来看一下这个函数的具体实现：

代码 3-8

```
void load_init_boot(init_func *init) {
    int i;
    for(i=0;init[i];i++) {
        init[i]();
    }
    boot_start();
};
```





函数通过 for 循环依次运行 init 数组中的成员函数，最终调用 boot_start() 函数，至此，操作系统的初始化过程结束。

这段程序之所以这样设计主要是从体系结构相关性这个角度考虑的。代码 3-8 中的函数就是一个与体现结构无关的函数，无论我们的代码运行在什么硬件平台下，load_init_boot 都可以被重复使用。而 init_func 数组是与体系结构相关的，因此当我们需要支持新的体系结构，或者更改原有体系结构的代码时，只需要重写或修改 init_func 数组实现其内部相关函数即可。我们所说的最基本的抽象指的就是这个。将各硬件平台都要实施的部分提炼出来，供所有体系结构复用，并将各平台独有的部分交给平台相关代码随意实现，从而在一定程度上解决了平台间移植的问题。

3.1.4 让启动代码运行起来

目前我们已经完成了操作系统初始化的第一阶段。现在不如让我们暂时停下来，整理并运行一下前边学习到的代码。

首先需要做的是将代码 3-3 的内容保存成文件，命名为 start.s。

为了节省篇幅，在代码 3-4 中我们只列出了核心程序，许多宏定义并没有列出来。为了使这段代码能够运行，我们需要在代码 3-4 最开头的地方添加如下内容：

代码 3-9

```
.equ DISABLE_IRQ,      0x80
.equ DISABLE_FIQ,      0x40
.equ SYS_MOD,          0x1f
.equ IRQ_MOD,          0x12
.equ FIQ_MOD,          0x11
.equ SVC_MOD,          0x13
.equ ABT_MOD,          0x17
.equ UND_MOD,          0x1b

.equ MEM_SIZE,          0x800000
.equ TEXT_BASE,         0x30000000
.equ _SVC_STACK,        (TEXT_BASE+MEM_SIZE-4)
.equ _IRQ_STACK,        (_SVC_STACK-0x400)
.equ _FIQ_STACK,        (_IRQ_STACK-0x400)
```

```
.equ _ABT_STACK,          (_FIQ_STACK-0x400)
.equ _UND_STACK,          (_ABT_STACK-0x400)
.equ _SYS_STACK,          (_UND_STACK-0x400)
```

在代码 3-9 中，程序主要定义了两类宏。

一类是与处理器模式和中断有关的，我们可以使用这些宏来方便地在不同模式间切换，同时使能或禁止中断。这些细节我们会在后续的学习中详细讲解。

另一类宏定义则是与内存有关的。根据 2410 的芯片手册，外部 sram 可以接在起始地址为 0x30000000 的位置上，我们虚拟的内存便是从这一地址开始的，并假设内存总容量为 8M。这样，我们选取 SVC 模式堆栈的最顶端为内存的最后一个字节（即 $0x30000000+0x800000-4$ ），并给其分配 1K 的堆栈空间供该模式使用，其他模式的堆栈空间大小同样为 1K，位置依次排列。我们将代码 3-9 与代码 3-4 的内容保存成文件，命名为“init.s”。

在代码 3-3 中，程序通过 `extern` 关键字声明了很多外部变量，但这些变量并未定义。于是我们需要新建一个文件并将这些变量的定义写进去，其内容如下：

代码 3-10

```
.global __vector_undefined
.global __vector_swí
.global __vector_prefetch_abort
.global __vector_data_abort
.global __vector_reserved
.global __vector_irq
.global __vector_fiq

.text
.code 32

__vector_undefined:
    nop
__vector_swí:
    nop
__vector_prefetch_abort:
    nop
__vector_data_abort:
    nop
```

```
__vector_reserved:  
    nop  
__vector_irq:  
    nop  
__vector_fiq:  
    nop
```

这里我们只需要将这段代码复制到文件中，命名为“abnormal.s”。代码 3-10 还有一条指令，那就是 `nop` 指令。其实这条指令在这里根本没有什么作用，只是在执行的时候能够消耗掉一个指令周期。此时我们尚未涉及到异常模式的处理，所以使用 `nop` 指令，权当占个位置，在讲到异常模式的时候，我们会进一步改写这些异常处理函数。

为了能够运行这部分程序，我们需要临时将代码 3-6、代码 3-7 和代码 3-8 改写一下，并使用我们在第 2 章中编写的 `helloworld` 函数来验证操作系统初始化部分的运行进度。

代码 3-11

```
typedef void (*init_func)(void);  
#define UFCNO ((volatile unsigned int *)(0x50000020))  
  
void helloworld(void){  
    const char *p="helloworld\n";  
    while(*p){  
        *UFCNO=*p++;  
    };  
}  
  
static init_func init[]={  
    helloworld,  
    0,  
};  
  
void plat_boot(void){  
    int i;  
    for(i=0;init[i];i++){  
        init[i]();  
    }  
    while(1);  
}
```

在代码 3-11 中，原本是应该填写启动函数的 init 数组，现在只填写了一个 helloworld 函数，目的仅是看到操作系统启动时的运行现象。我们将代码 3-11 保存到文件中，命名为“boot.c”。

接着将代码 3-5 的内容保存到文件中，将其命名为“le eos.lds”。

编译过程非常简单，我们需要写一个 Makefile 文件，将代码 3-12 的内容保存成文件，与源代码放在同一目录下，命名为 Makefile。

代码 3-12

```
CC=arm-elf-gcc
LD=arm-elf-ld
OBJCOPY=arm-elf-objcopy

CFLAGS= -O2 -g
ASFLAGS= -O2 -g
LDFLAGS=-Tle eos.lds -Ttext 30000000

OBJS=init.o start.o boot.o abnormal.o

.c.o:
    $(CC) $(CFLAGS) -c $<
.s.o:
    $(CC) $(ASFLAGS) -c $<

le eos:$ (OBJS)
    $(CC) -static -nostartfiles -nostdlib $(LDFLAGS) $? -o $@ -lgcc
    $(OBJCOPY) -O binary $@ le eos.bin

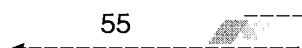
clean:
    rm *.o le eos le eos.bin -f
```

在确认所有文件都处在同一目录下后，只需要在终端上运行 make 这条命令，最终将会生成二进制文件“le eos.bin”。

当然，如果需要在 skyeye 中虚拟运行，还有一个文件是我们需要的，那就是“skyeye.conf”文件，不同于第 2 章的配置文件，这里的 skyeye.conf 的内容稍有变化，如下所示。

```
cpu: arm920t
mach: s3c2410x

#physical memory
```



```
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,  
file=./le eos.bin, boot=yes  
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

因为在代码中，我们已经假设内存起始于地址 0x30000000 处，大小为 0x00800000。为保证代码能够成功运行，我们必须虚拟出一块内存，起始位置和大小恰与代码一致。

现在万事俱备了，在终端运行 skyeye 命令，将可以看到 helloworld 字符串从 skyeye 中打印出来：

```
.....  
Loaded RAM ./le eos.bin  
start addr is set to 0x30000000 by exec file.  
helloworld
```

看到这个结果，不能不让人兴奋，虽然该结果与第 2 章相比没有什么变化。但从代码实现上看，二者有本质区别，其意义完全不同。我们现在正在一步步地将我们所学的知识转化为实践，逐步打造属于自己的嵌入式操作系统！

当然，现在就沾沾自喜还为时过早。接下来我们就来啃一块硬骨头——MMU。这一部分内容，相对来说是比较具有挑战性的。如果您的 ARM 基础相对薄弱，或者读过前面的内容之后仍然是一知半解，不妨暂时略过这一节，继续阅读接下来的内容。

3.2 MMU

什么是 MMU 呢？MMU 是 Memory Management Unit 的缩写，它代表集成在 CPU 内部的一个硬件逻辑单元，主要作用是给 CPU 提供从虚拟地址向物理地址转换的功能，从硬件上给软件提供一种内存保护的机制。

举个例子来说，若一个 CPU 没有 MMU，我们想向内存地址 0x0 处写一个整型值 0xffff，就只能使用如下的方法：

```
(int *)0x0=0xffff;
```

但如果一个 CPU 带有 MMU，并且通过软件对 MMU 进行了配置，将地

址 0xc0000000 映射到地址 0x0 处，并激活 MMU。那么当我们想要向内存地址 0x0 处写入一个 0xffff 时，代码应该并且只能是这样的：

```
(int *)0xc0000000=0xffff;
```

这里，我们将映射之前的地址 0x0 称为物理地址，物理地址即硬件自己定义的地址往往是不可更改的，而将映射之后的地址 0xc0000000 称为虚拟地址，这个地址在真实的硬件中并不存在，只不过我们可以通过访问虚拟地址而间接实现对物理地址的访问，并且只能使用这种方法进行访问，如图 3-2 所示。当然，我们也可以实时调整这种映射关系，让别的地址也映射到物理地址 0x0 上，而不一定非得是 0xc0000000。这就是说，这种映射关系允许动态调整，允许多对一的映射。

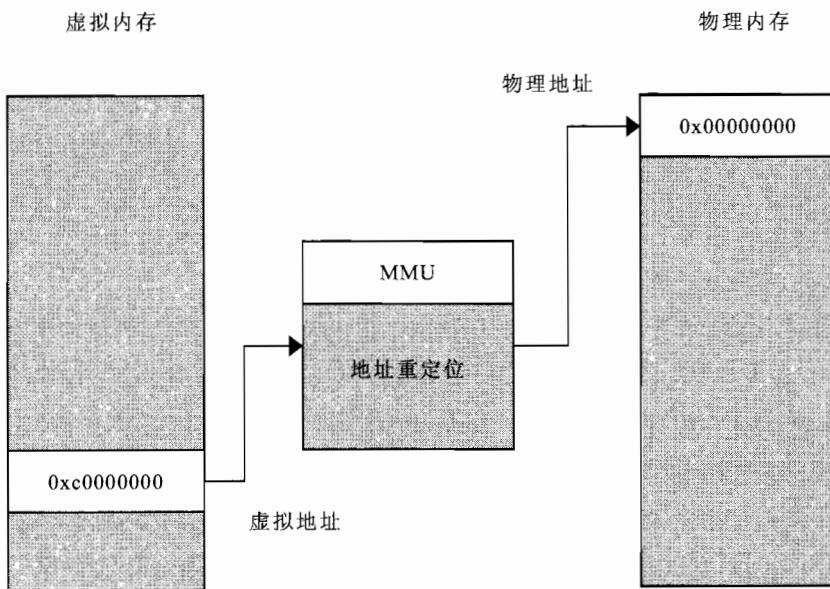


图 3-2 MMU 工作原理

刚刚接触虚拟地址映射的读者朋友们一定都会有这样的疑问，我们为什么要多此一举，非要使用 MMU 进行地址映射呢？

这个问题说来话长，在操作系统还是以单任务的方式运行的年代，MMU 并没有它存在的意义，当操作系统开始支持多任务后，从宏观上看，多个进程并行执行，并且各进程之间的资源都是相互独立的，通过什么手段可以有效地保护各进程的资源不会被其他进程破坏呢？当然，我们可以通过使用软



件划分好各进程的地址空间，但因为各进程仍然有跨界操作的权限，即便在正常执行时进程间彼此不会互相干扰，也无法避免错误执行时的后果。因此就需要从硬件上提供一种机制，彻底限制某个进程对其他进程资源的访问权限，于是内存保护单元应运而生，MMU能够很好地提供内存保护作用，这就是需要 MMU 的第一个原因。

随着计算机技术的发展，操作系统愈加复杂，应用环境也呈现多样化，比如我们经常需要将同一个应用程序同时运行多次，这样，问题就会随之而来。我们知道，一个应用程序在编译完成后，其运行地址也就确定了。假设该程序不可以被重定位，那么编译时所确定的地址就成了程序能够运行的唯一地址，当同一个应用程序被两次运行时，必然造成同一个地址既存储了一个进程的指令或数据又存储了另一个进程的指令或数据，这一矛盾的解决仍然要靠 MMU。例如，一个应用程序在虚拟地址 0x10000000 处存储了一个计数值，当该程序第一次运行时，MMU 将该地址映射到 0x30000000 处，而同一个程序再次被运行时，只需要将 0x10000000 这一地址映射到别的地方，比如 0x31000000，这样，前后两个进程虽然虚拟地址是相同的，但他们却独立运行在不同的物理地址中，因此能够在同一时刻同时运行。给各进程提供独立的地址空间是 MMU 存在的第二个原因。

虽然在高级操作系统中，使用 MMU 优势明显，但同时也存在着很多争议。其中最大的争议是关于地址映射的执行效率问题。尽管整个地址翻译的过程都是由硬件实现的，相比于直接访问地址，时间上的浪费微乎其微，但大量寻址的累积势必会造成系统效率降低。因此很多对实时性具有高要求的操作系统并不倾向于使用 MMU。然而，在一些非实时或对实时性要求并不严格的系统中，MMU 却能够给上层应用程序提供极大的便利。

不管怎样，地址映射这种内存管理方式对于高级操作系统来说仍然是一种普遍采用的方法。所以接下来，就让我们以 ARM 为例，深入研究一下 MMU 的工作原理。

3.2.1 页表

首先让我们来介绍一个概念——页表（page table）。页表就是存储在内存中的一张表，表中记录了将虚拟地址转换成物理地址的关键信息。MMU 正是通过对页表进行查询，实现了地址之间的转换。也就是说，MMU 每次



工作的时候都要去查这张表，从中找出与虚拟地址相对应的物理地址，然后再进行数据存取。页表的作用如图 3-3 所示。

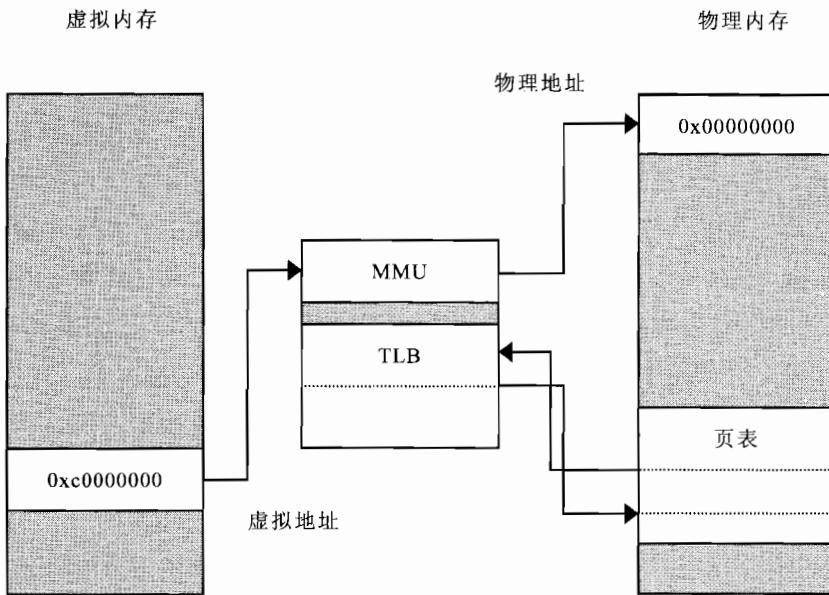


图 3-3 页表的作用

页表中的条目被称为页表项 (page table entry)，一个页表项负责记录一段虚拟地址到物理地址的映射关系，稍后我们会详细介绍。

既然页表是存储在内存中的，那么程序每次完成一次内存读取时都至少会访问内存两次，相比于不使用 MMU 时的一次内存访问，效率被大大降低了，如果所使用的内存的性能比较差的话，这种效率的降低将会更明显。因此，如何在发挥 MMU 优势的同时使系统消耗尽量减小，就成为了一个亟待解决的问题。

于是，TLB 产生了。TLB 是什么呢？我们叫它转换旁路缓冲器，它实际上是 MMU 中临时存放转换数据的一组重定位寄存器。既然 TLB 本质上是一组寄存器，那么不难理解，相比于访问内存中的页表，访问 TLB 的速度要快很多。因此如果页表的内容全部存放于 TLB 中，就可以解决访问效率的问题了。

然而，由于制造成本等诸多限制，所有页表都存储在 TLB 中几乎是不可能的。这样一来，我们只能通过在有限容量的 TLB 中存储一部分最常用



的页表，从而在一定程度上提高 MMU 的工作效率。

这一方法能够产生效果的理论依据叫做存储器访问的局部性原理。它的意思是说，程序在执行过程中访问与当前位置临近的代码的概率更高一些。因此，从理论上我们可以说，TLB 中存储了当前时间段需要使用的大多数页表项，所以可以在很大程度上提高 MMU 的运行效率。

让我们接着聊页表。页表是由页表项组成的，每一个页表项都能够将一段虚拟地址空间映射到一段物理地址空间中。这里所谓的这段虚拟地址空间，更专业地讲，应该叫页，一个页对应了页表中的一项，页的大小通常是可以选的。在 ARM 中，一个页可以被配置成 1K、4K、64K 或 1M 大小（ARM v6 体系以后，不再支持 1K 大小的页），分别叫做微页、小页、大页和段页。页的大小决定了映射的粒度，是根据实际应用有选择地配置的。以 1M 为例，按照我们前面的描述，假设系统中将有 64M 内存需要被映射，那么我们一共需要 $64M/1M$ 个页表项，而每个页表项需要占据 4 个字节，也就是说，有 256 字节的内存要专门负责地址映射，不能用于其他用途。

对于 1K、4K 和 64K 大小的页，MMU 采用二级查表的方法，即首先由虚拟地址索引出第一张表的某一段内容，然后再根据这段内容搜索第二张表，最后才能确定物理地址。这里的第 1 张表，我们叫它一级页表，第二张表被称为是二级页表。采用二级查表法的主要目的是减小页表自身占据的内存空间，但缺点是进一步降低了内存的寻址效率。不同大小的页对查表方法的支持程度如表 3-1 所示。

表 3-1 不同大小页的查表方法

查表方法	页大小			
	1K	4K	64K	1M
一级查表	不支持	不支持	不支持	支持
二级查表	支持	支持	支持	不支持

下面，首先来研究一下相对简单的一级查表。

3.2.1.1 一级查表

一级查表只支持大小为 1M 的页。准确地讲，这里所谓的 1M 大小的页，

应称为段 (section)。此时，一级页表也被称为段页表。图 3-4 描述了段页表的内存分布情况和段页表项的具体格式。

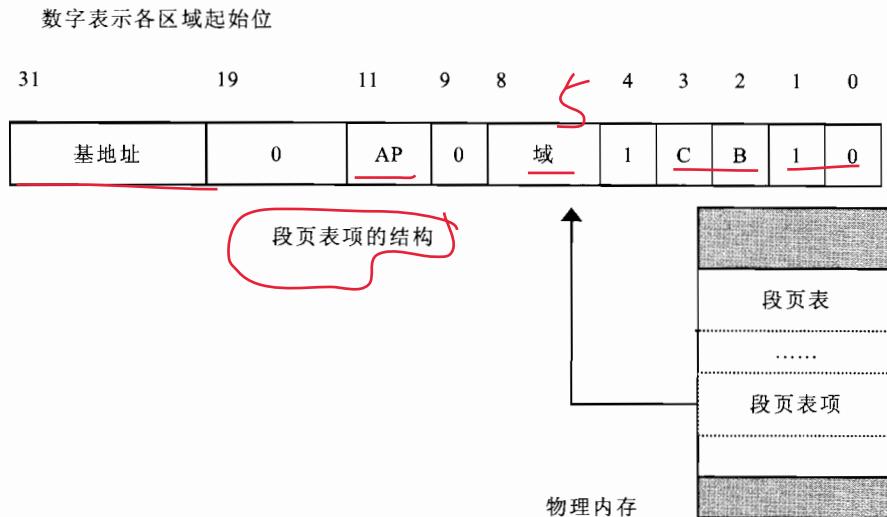


图 3-4 段页表项的结构

段页表中的每一项都类似于图 3-4 中的形式，其中：

(1) 31~20 位段表示物理地址的基地址，一共 12 位，也就是说，如果我们通过虚拟地址找到某一个段页表项，那么就可以确定这段虚拟地址所对应的物理地址的高 12 位了。因为段页表项后 20 位正好可以描述 1M 的内存，因此基地址每增加一个单位，物理地址就会增加 1M 空间。所以，我们也可以说明该基地址表示了虚拟地址属于哪 1M 范围的物理地址。由此可知，使用段页表进行地址映射时，每一页能够描述 1M 的物理地址空间。进一步讲，段页表最多支持 1024 个页，最多占用系统 4K 字节的内存来存放页表。

(2) 11~10 位是 AP 位，区分了用户模式和特权模式对同一个页的不同访问权限。例如，当 AP 位为“11”时，表示任何模式下都可以对该空间进行读写，“10”则表示特权模式可读写该页，而用户模式只能读取该页。对 AP 位详细的描述请参考页权限一节。

(3) 8~5 位代表该页所属的域。在 ARM 体系结构中，系统中规定了 16 个域，因此使用 4 个位就能表示该页属于哪个域，而每一个域又有各自独立的访问权限，从而实现了初级的存储器保护。



(4) 3位和2位分别代表cache和write buffer。相应的位为1则表示被映射的物理地址将使用cache或write buffer。关于cache和write buffer的有关内容，我们稍后会详述。

(5) 1~0位。这两位用来区分页表类型，对于段页表，这两位的值总是为“10”。

总的来说，段页表项的内容虽然复杂，但归结起来无非就是两个问题，一是，如何解决某一虚拟地址属于哪段物理地址，二是，如何确定这段地址的访问权限。使用其他形式的页表，本质上也是要解决这两个问题。

现在，假设段页表已经被成功地添加到内存之中了，那么接下来的问题是我们应该怎样通过一个虚拟地址找到与之对应的段页表项呢？找到页表项之后，又是怎样找到对应的物理地址的呢？

很显然，虚拟地址本身就可以解决上述问题。

如图3-5所示，首先，我们要让MMU知道段页表在内存中的首地址，也就是图中所说的页表基址，因为段页表是我们通过程序确定的，存储在什么位置程序员自然清楚。然后，在CPU需要寻址的时候，MMU就可以自动地利用页表将虚拟地址映射为物理地址并寻址物理地址，其步骤如下：

(1) MMU取出虚拟地址的前12位作为页表项的偏移，结合页表基址，找到对应的页表项。具体来说，就是将这12位数取出，然后左移两位和页表基址相与，就能得到相应页表项的地址了。例如，页表基址是0x10000000，如果虚拟地址是0x00101000，则前12位数是0x001，那么根据上述步骤将其左移两位，结果为0x004，那么页表项地址就应该是 $0x10000000|0x004=0x10000004$ ，从该地址中读取的32位的数据即是该虚拟地址所对应的页表项。

(2) 找到与虚拟地址对应的页表项之后，就可以从该页表项中读出一个重要信息，如图3-4所示，页表项的前12位定位了该虚拟地址所对应的物理地址在哪个范围内。例如，从0x10000004中读出的页表项的内容为0x30000c12，那么我们就已经知道了，虚拟地址0x00101000对应的物理地址在0x30000000与0x30100000之间。既然地址范围已经清楚了，那么虚拟地址又该定位到该范围的哪个位置呢？这就要靠虚拟地址的后20位了。



虚拟地址

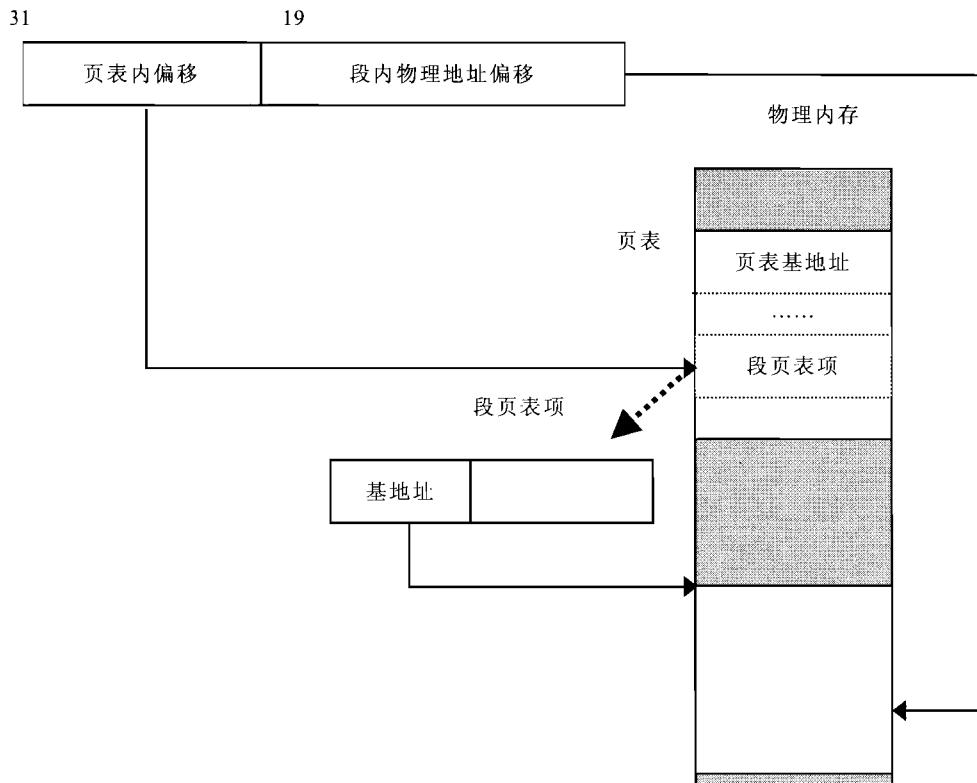


图 3-5 段页表映射过程

(3) MMU 将虚拟地址后 20 位和页表项内容清除掉后 20 位之后的结果做与的操作，就得到了与虚拟地址对应的实际物理地址了。例如，页表项的内容为 0x30000c12，清除掉后 20 位的结果为 0x30000000，虚拟地址的后 20 位为 0x01000，将其和 0x30000000 相与，结果为 0x30001000，这便是最终的物理地址。

当然，MMU 在进行地址映射期间，还要进行访问权限的检查，方法是读出页表项的权限位，按照既定规则去检查，如果允许对该地址进行访问则正常访问，如果不允许访问，则抛出异常，通过程序将其捕获并处理。这便是使用段页表时 MMU 的地址映射过程。



3.2.1.2 二级查表

我们也可以选择使用二级查表的方式去实现地址映射。从原理上讲，一级查表和二级查表其实并没有太大的差别。使用二级查表法，经过一级页表得到的数据不再是记录了物理地址信息的数据了，而是二级页表项的索引信息。而这些信息除了相应位和标志与一级页表项略有不同之外，与一级查表并无不同。

由于在我们的操作系统中没有使用到二级查表法，这部分内容我们就不介绍了，读者如果感兴趣，可以参考相关文档，自己去实现。

3.2.2 页权限

前面我们已经介绍过，内存管理单元不仅能够给进程提供独立的地址空间，而且可以设置进程对某段地址空间的访问权限。总结起来，页的权限主要跟两个方面有关——域（domain）和访问权限（access permission）。

域的权限是以 1M 为单位的，ARM 最多允许定义 16 个不同的域，并允许配置每 1M 的地址空间分属于 16 个域中的任意一个，从而可以通过使用改变域权限的方法来批量控制整块地址空间的访问权限。举个例子来说，假设我们配置系统中的前 4M 内存，使其由第一个域来控制，而将系统中的后 4M 内存进行配置，让第二个域去控制。那么当环境要求让系统中的前 4M 内存可以被自由访问、后 4M 内存只能被特定任务访问时，就可以简单地修改系统中这两个域的权限，迅速地实现要求。在 ARM 920T 体系结构中，这 16 个域的权限是由一个叫做 CP15 的协处理器中的 C3 寄存器来控制的，如图 3-6 所示。

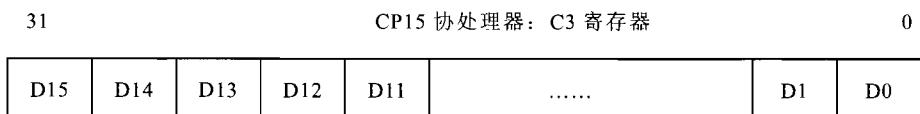


图 3-6 C3 寄存器

在图 3-6 中，在 32 位的寄存器 C3 中，每两位代表一个域权限，一共是 16 个域。而每个域只有 4 种取值，分别是 0b00、0b01、0b10、0b11，这 4



个值分别代表了该域的不同权限类型，如表 3-2 所示。

表 3-2 域权限设置

访问	域权限值	描述
管理者	0b11	该域不受来自页表的权限限制，总可以被访问
保留	0b10	结果不可预料
用户	0b01	访问受页表中的权限限制
禁止访问	0b00	将产生域错误

因此，只要配置好某块内存对应域的权限，就可以实时地改变这块内存区的访问类型。接下来的问题是如何知道哪块内存属于哪个域呢？读者只要回头看一下我们前面讲的段页表项的内容就会明白了。在段页表项中，第 8 位~第 5 位的值用来表示该段页表项属于 16 个域中的哪一个。MMU 在进行地址映射时，需要读出这些位，在 CP15: C3 中找到对应域的权限，根据表 3-2 中的表述确定该段内存的访问权限，然后进行访问，域就是通过这种方式实现了一种粗粒度的页权限。

另一种更有效的权限控制方法是访问权限 (access permission)，段页表项中 11 位和 10 位代表该段页表项的 AP 位，其可能的取值也只有 4 种，与之对应的读写权限如表 3-3 所示。

表 3-3 AP 权限设置

AP 位	S 位	R 位	特权模式	用户模式
0b11	忽略	忽略	可读写	可读写
0b10	忽略	忽略	可读写	只读
0b01	忽略	忽略	可读写	不可访问
0b00	0	0	不可访问	不可访问
0b00	0	1	只读	只读
0b00	1	0	只读	不可访问
0b00	1	1	结果不可预料	结果不可预料

表 3-3 中的 S 位和 R 位指的是协处理器 CP15: C1 寄存器中的两个位，专门负责全局修改存储器访问权限，正确地配置好这两个位能够提高系统访问大块内存时的速度。设定 S 位能够使用户任务无法对页进行访问，同时又允许了特权模式的读权限，而改变 R 位则可以允许特权模式和用户模式都对页有只读的权限。当然，R 位和 S 位能够发挥作用的前提是页表项中的 AP 位必须为零值。当页表项中的 AP 位为非零值时，表示该页表项所对应的内存权限由 AP 位的值来决定，当 AP 为零，则该页表项所代表的内存的

读写权限由 R 位和 S 位决定。修改 R 位和 S 位要比一项一项地去修改页表项更迅速，因此适用于对大块内存进行权限控制。

3.2.3 cache 和 write buffer

如果有人问，从 CPU 的角度来看哪种存储器件的访问速度最快？答案应该是寄存器，因为寄存器是寄生在 CPU 芯片内部的，与 CPU 处理单元距离最近，速度最快。比寄存器稍慢一些的呢？那就应该是各种形式的内存了。有些 CPU 内部会内置小容量的静态内存，但多数内存是工作在板卡的层次上的，也就是说，多数情况下，内存与 CPU 需要通过主板进行连接，再加上廉价的内存本身就有读写速度的瓶颈，这样一来，站在 CPU 的角度上看，内存也就变成一个慢速设备了。因此，我们需要一种有效手段来提高内存的访问效率，解决的方法就是 cache 和 write buffer。

简单地说，cache 和 write buffer 都是内置于 CPU 内部的一小段高速存储器，cache 中保存着最近一段时间被 CPU 使用过的内存数据，而 write buffer 则是用来应对内存的写操作的，将原本要写向内存的数据暂写到 write buffer 中，等到 CPU 空闲的时候，数据才会慢慢地被搬移到内存里。

由于成本的问题，cache 和 write buffer 的容量都不会很大，因此，里边只能保存局部的数据信息。但这也不会影响 cache 和 write buffer 发挥其提高系统性能的作用。还记得前面提到的存储器访问的局部性原理吗？程序在一段时间内对数据的访问总是局部的。所以大多数情况下，cache 和 write buffer 会对程序的运行速度有很大的帮助。在这里我们只需要介绍一下 cache 和 write buffer 在页表项中的配置方法，而这足以使我们建立一个段页表了。

在段页表项中，第 3 位和第 2 位分别用来设置 cache 和 write buffer。简单地说，当这两个位为 1 时，表示该段页表项所描述的内存将使用 cache 和 write buffer，如果将这两个位设置为 0，则表示禁用 cache 和 write buffer。

当我们按照上述方法完成了段页表的初始化工作后，就可以使能 MMU 了，这主要是通过配置 ARM 的 CP15 协处理器实现的。

可能有些读者看着前面冗长的描述已经厌倦了，此时迫不及待地想实践一下。那我们就先看一段代码，体会一下从程序上如何初始化段页表。

代码 3-13

```

#define PAGE_TABLE_L1_BASE_ADDR_MASK          (0xfffffc000)
#define VIRT_TO_PTE_L1_INDEX(addr)    ((addr)& \
                                     0xffff000000) >> 18
#define PTE_L1_SECTION_NO_CACHE_AND_WB      (0x0<<2)
#define PTE_L1_SECTION_DOMAIN_DEFAULT       (0x0<<5)
#define PTE_ALL_AP_L1_SECTION_DEFAULT      (0x1<<10)
#define PTE_L1_SECTION_PADDR_BASE_MASK     (0xffff000000)
#define PTE_BITS_L1_SECTION                (0x2)
#define L1_PTR_BASE_ADDR                  0x30700000
#define PHYSICAL_MEM_ADDR                0x30000000
#define VIRTUAL_MEM_ADDR                 0x30000000
#define MEM_MAP_SIZE                     0x800000
#define PHYSICAL_IO_ADDR                 0x48000000
#define VIRTUAL_IO_ADDR                  0xc8000000
#define IO_MAP_SIZE                      0x18000000

unsigned int gen_l1_pte(unsigned int paddr){
    return (paddr&PTE_L1_SECTION_PADDR_BASE_MASK) | \
           PTE_BITS_L1_SECTION;
}

unsigned int gen_l1_pte_addr(unsigned int baddr,\n                           unsigned int vaddr) {
    return (baddr&PAGE_TABLE_L1_BASE_ADDR_MASK) | \
           VIRT_TO_PTE_L1_INDEX(vaddr);
}

void init_sys_mmu(void) {
    unsigned int pte;
    unsigned int pte_addr;
    int j;

    for(j=0;j<MEM_MAP_SIZE>>20;j++) {
        pte=gen_l1_pte(PHYSICAL_MEM_ADDR+(j<<20));
        pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;
        pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;
        pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;
        pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\n                               VIRTUAL_MEM_ADDR+(j<<20));
        *(volatile unsigned int *)pte_addr=pte;
    }
}

```

```
for(j=0;j<IO_MAP_SIZE>>20;j++) {  
    pte=gen_l1_pte(PHYSICAL_IO_ADDR+(j<<20));  
    pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;  
    pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;  
    pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;  
    pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\  
                             VIRTUAL_IO_ADDR+(j<<20));  
    *(volatile unsigned int *)pte_addr=pte;  
}
```

代码 3-13 中使用的两个核心函数分别是 `gen_l1_pte` 和 `gen_l1_pte_addr`, `gen_l1_pte` 函数以物理地址为参数, 能够返回与该物理地址相对应的段页表项的内容, 函数直接返回了下面这个表达式的值:

```
(paddr&PTE_L1_SECTION_PADDR_BASE_MASK) |  
PTE_BITS_L1_SECTION
```

`PTE_L1_SECTION_PADDR_BASE_MASK` 的值是 `0xffff0000`, 和任意一个物理地址相与, 所得到的即是该物理地址的段地址。换句话说, 这个结果恰好代表了该物理地址属于哪一段内存范围, 而这正是段页表项的内容之一。`PTE_BITS_L1_SECTION` 被定义为 `0x2`, 根据我们前面的描述, 页表项的后两位如果是 `0b10` 则表示该页表项为段页表项。因此, 该函数返回的正是一个基本的段页表项内容。而函数 `gen_l1_pte_addr` 则是通过段页表的基址与虚拟地址产生该段页表项的地址, 其表达式如下:

```
(baddr&PAGE_TABLE_L1_BASE_ADDR_MASK) |  
VIRT_TO_PTE_L1_INDEX(vaddr);
```

`PAGE_TABLE_L1_BASE_ADDR_MASK` 被定义为 `0xfffffc000`, 与页表基地址相与, 确保了当我们传递不正确的页表基地址时, 迫使该页表基地址的后 14 位是零。这样做的原因何在? 回顾一下虚拟地址到物理地址的映射过程, MMU 需要将虚拟地址的前 12 位作为段页表项地址的偏移量, 结合段页表基地址得到真正的段页表项的地址, 而又因为段页表项为 4 个字节, 需要两位表示, 因此一共需要 14 位来表示段页表项的偏移, 那么段页表基地址的后 14 位就应该是零了。当计算某一页表项地址时, 该地址的前 18 位即为页表基地址, 14 位~2 位应该是虚拟地址的前 12 位, 而最后两位总是零。宏 `VIRT_TO_PTE_L1_INDEX` 的作用正是计算偏移量, 再和段页表基地址做

与操作，就能得到该段页表项的物理地址了。

有了这两个函数，再加上设置域、AP 以及 cache 和 write buffer 的宏，即可实现段页表项的初始化。函数 init_sys_mmu 通过一个循环操作将物理地址 0x30000000~0x30800000 映射到虚拟地址的 0x30000000~0x30800000 处，这样一来，就正好覆盖了我们虚拟出来的 8M 内存。采用这种映射方法，很大程度上简化了代码的复杂度。

既然操作系统已经使用了 MMU，那么在编译系统源代码时就必须使用虚拟地址链接。倘若虚拟地址和物理地址不一致，那么操作系统开始运行的第一个工作就是通过虚拟地址计算物理地址，从中取出页表项并迅速激活 MMU，其复杂程度可想而知。

某些操作系统，比如 Linux，因为有跨平台支持的需求，很多概念都被抽象化了，Linux 系统就规定内核默认使用虚拟地址空间 3~4G 的范围，而很少会有某种 CPU 会将物理内存映射到 3G 之后的地址。因此，对于 Linux 来说，物理内存地址与虚拟内存地址的不一致就在所难免了。

我们的操作系统在设计时，也在一定程度上考虑到了跨平台兼容的问题。当然，这并不意味着我们必须要把物理内存地址和虚拟内存地址分开。关于操作系统的设计问题，我们不急于在这里探讨。读者只需要认识到，将内存所在的物理地址对等地映射给虚拟地址是比较好的选择。当然，除了内存所在地址之外的其他地址应该如何被映射就无所谓了。

为了说明并验证 MMU 的工作原理，在代码 3-13 中，我们将 0x48000000~0x60000000 的物理地址范围映射到虚拟地址 0xc8000000~0xe0000000 处，熟悉 s3c2410 的读者朋友应该知道，s3c2410 的外设地址空间就处在这一范围内。因此，当 MMU 被激活以后，我们就不能通过将数据写入到 0x50000020 来显示数据了，而必须要将显示的数据写入到 0xd0000020 才可以。

现在，只要激活 MMU，我们的系统就可以在虚拟地址中工作了。

3.2.4 激活 MMU

MMU 的配置和控制都是通过操作 CP15 协处理器实现的。ARM 支持 16 个协处理器，当然，并不是所有的 ARM 系统结构都全部包含这些协处理器。在 ARM920T 系列处理器中，协处理器只有两个，一个是负责系统调试

的 CP14，另一个就是负责系统控制的 CP15。系统中的 cache、write buffer、MMU、时钟模式等都可以通过操作 CP15 来完成。

访问 CP15 协处理器不能使用常规指令，ARM 提供了一组专门操作协处理器的指令，这些指令与 ARM 自身的指令格式有很大的不同。

协处理器也有属于它自己的寄存器，比如 CP15 协处理器内部就有 16 个寄存器，通常使用 C_n 来表示，这里的 n 代表协处理器寄存器的序号。想要使能 MMU，我们所要做的便是正确地配置 CP15 的相关寄存器。

让我们来看一下这段代码：

代码 3-14

```
void start_mmu(void){  
    unsigned int ttb=L1_PTR_BASE_ADDR;  
    asm(  
        " mcr p15,0,%0,c2,c0,0\n"  
        " mvn r0,#0\n"  
        " mcr p15,0,r0,c3,c0,0\n"  
        " mov r0,#0x1\n"  
        " mcr p15,0,r0,c1,c0,0\n"  
        " mov r0,r0\n"  
        " mov r0,r0\n"  
        " mov r0,r0\n"  
        :  
        : "r" (ttb)  
        : "r0"  
    );  
}
```

有些读者读了上述代码，可能会觉得很奇怪，这明明是一个 C 函数，为什么其中又包含了汇编代码呢？其实，这里我们使用的是 GCC 内联汇编的技术。使用 C 代码内联汇编的编程方式在某些情况下可以发挥出强大的作用，如手动优化软件关键部分的代码、直接调用特殊的处理器指令实现特定功能等。

在针对 ARM 体系结构的编程中，我们很难直接使用 C 语言产生操作协处理器的相关代码，因此使用汇编语言来实现就成为了唯一的选择。但由于对 MMU 的操作也涉及相对复杂的逻辑，如果完全通过汇编代码实现，又会过于复杂、难以调试。这样看来，C 语言内嵌汇编的方式倒是一个不错的选择。

然而，使用内联汇编的一个主要问题是，内联汇编的语法格式与使用的编译器直接相关，也就是说，使用不同的 C 编译器内联汇编代码时，它们的写法是各不相同的。为了让读者读懂代码 3-14 的内容，我们首先需要介绍一下在 ARM 体系结构下 GCC 的内联汇编方法。

3.3 GCC 内联汇编

首先，让我们来共同了解一下 GCC 内联汇编的一般格式：

```
asm(
    代码列表
    : 输出运算符列表
    : 输入运算符列表
    : 被更改资源列表
);
```

在 C 代码中嵌入汇编需要使用 `asm` 关键字，在 `asm` 的修饰下，代码列表、输出运算符列表、输入运算符列表和被更改的资源列表这 4 个部分被 3 个 “`:`” 分隔。这种格式虽然简单，但并不足以说明问题，我们还是通过例子来理解它吧。

代码 3-15

```
void test(void){
    .....
    asm(
        " mov r1,#1\n"
        : 0
        :
        : " r1" C
    );
    .....
}
```

在代码 3-15 中，函数 `test` 中内嵌了一条汇编指令实现将立即数 1 赋值给寄存器 R1 的操作。由于没有任何形式的输出和输入，因此输出和输入列表的位置上什么都没有填写。但是，我们发现在被更改资源列表中，出现了

寄存器 R1 的身影，这表示我们通知了编译器，在汇编代码执行过程中 R1 寄存器会被修改。

寄存器被修改这种现象发生的频率还是比较高的。例如，在调用某段汇编程序之前，寄存器 R1 可能已经保存了某个重要数据，当汇编指令被调用之后，R1 寄存器被赋予了新的值，原来的值就会被修改，所以，需要将会被修改的寄存器放入到被更改资源列表中，这样编译器会自动帮助我们解决这个问题。也可以说，出现在被更改资源列表中的资源会在调用汇编代码一开始就首先保存起来，然后在汇编代码结束时释放出去。所以，代码 3-15 与如下代码从语义上来说是等价的。

代码 3-16

```
void test(void){  
    ....  
    asm(  
        " stmfd sp!,{r1}\n"  
        " mov r1,#1\n"  
        " ldmfd sp!,{r1}\n"  
    );  
    ....  
}
```

代码 3-16 中的内联汇编既无输出又无输入，也没有资源被更改，只留下了汇编代码的部分。由于程序在修改 R1 之前已经将寄存器 R1 的值压入了堆栈，在使用完之后，又将 R1 的值从堆栈中弹出，所以，通过被更改资源列表来临时保存 R1 的值就没什么必要了。

在以上两段代码中，汇编指令都是独立运行的。但更多的时候，C 和内联汇编之间会存在一种交互。C 程序需要把某些值传递给内联汇编运算，内联汇编也会把运算结果输出给 C 代码。此时就可以通过将适当的值列在输入运算符列表和输出运算符列表中来实现这一要求。请看下面的代码：

代码 3-17

```
void test(void){  
    int tmp=5;  
    asm(  
        " mov r4,%0\n"  
    );
```

```

    : "r" (tmp)
    : "r4"
};

}

```

代码 3-17 中有一条 mov 指令，该指令将%0 赋值给 R4。这里，符号%0 代表出现在输入运算符列表和输出运算符列表中的第一个值。如果%1 存在的话，那么它就代表出现在列表中的第二个值，依此类推。所以，在该段代码中，%0 代表的就是 " r " (tmp)这个表达式的值了。

那么这个新的表达式又该怎样解释呢？原来，在 " r " (tmp)这个表达式中，tmp 代表的正是 C 语言向内联汇编输入的变量，操作符 " r " 则代表 tmp 的值会通过某一个寄存器来传递。在 GCC4 中与之相类似的操作符还包括 " m " 、 " I " ，等等，其含义见表 3-4。

表 3-4 嵌入汇编操作符节选

操作符	含义
r	通用寄存器 R0~R15
m	一个有效内存地址
I	数据处理指令中的立即数
x	被修饰的操作符只能作为输出

从结构上讲，代码 3-17 与代码 3-14 已经很类似了。与输入运算符列表的应用方法一致，当 C 语言需要利用内联汇编输出结果时，可以使用输出运算符列表来实现，其格式应该是下面这样的。

代码 3-18

```

void test(void){
    int tmp;
    asm(
        " mov %0,%1\n"
        : " =r" (tmp)
    );
}

```

在代码 3-18 中，原本应出现在输入运算符列表中的运算符，现在出现在了输出运算符列表中，同时变量 tmp 将会存储内联汇编的输出结果。这

里有一点可能已经引起大家的注意了，代码 3-18 中操作符 `r` 的前面多了一个“`=`”。这个等号被称为约束修饰符，其作用是对内联汇编的操作符进行修饰。几种修饰符的含义如表 3-5 所示。

表 3-5 GCC4 中嵌入汇编修饰符

修饰符	说明
无	被修饰的操作符是只读的
<code>=</code>	被修饰的操作符只写
<code>+</code>	被修饰的操作符具有可读写的属性
<code>&</code>	被修饰的操作符只能作为输出

当一个操作符没有修饰符对其进行修饰时，代表这个操作符是只读的，这正好符合代码 3-17 的要求。但是在代码 3-18 中，我们需要将内联汇编的结果输出出来，那么至少要保证该操作符是可写的。因此，“`=`”或者“`+`”也就必不可少了。

至此，本书对 ARM 体系结构下 GCC 内联汇编方法的讲解就算完成了。当然，出于篇幅的限制，GCC 内联汇编还有很多细节我们没有介绍。这些细节或者不常用，或者不重要，总之，在掌握了前面介绍的基本知识的前提下，我们可以在需要这些细节的时候查阅相关参考手册，以便进一步解决更复杂的问题。

现在让我们再次回到代码 3-14，你会发现，代码 3-14 中原本晦涩难懂的内联汇编代码，现在看起来就非常简单了。接下来我们就来分析一下这段汇编代码，看看 ARM 的协处理器是如何操作的。

首先，我们要学习两条指令，分别是 `mcr` 和 `mrc`，这两条指令都是跟协处理器的操作有关的，它们能够实现协处理器寄存器与通用寄存器之间的数据传递，其格式如下：

~~<mrc|mrc>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}~~

其中，`mrc` 指令能够将协处理器寄存器中的内容传输到 ARM 通用寄存器中，而 `mcr` 则实现了相反的操作，将通用寄存器中的值传递给协处理器中的寄存器，“`{}`”内的表达式是可选的。`cp` 代表协处理器，不同的协处理器的作用各不相同，如 CP15 恰与系统控制有关；`opcode1` 被称为第一操作数；`Rd` 是 ARM 通用寄存器；`Cn`、`Cm` 代表的是协处理器的寄存器；`opcode2` 是第二操作数。整个指令看起来有些复杂，这么多的操作数和寄存器，在实

际使用中应该如何选择呢？这一点读者不需要担心，在实际操作中应该选择什么样的操作数，使用哪些协处理器寄存器，这些都是由协处理器所规定的，我们只需要依照其格式使用即可。

好了，结合这两条协处理器指令以及 GCC 下内联汇编的技术，读懂代码 3-14 也就轻而易举了。下面让我们具体来分析一下这段代码。

```
mcr p15,0,%0,c2,c0,0
```

这是在代码 3-14 中出现的第一条汇编指令，其中 %0 的值为 L1_PTR_BASE_ADDR。当然这个值最终会通过寄存器传递，所以上面的指令等效于如下两条指令：

```
mov r0,#L1_PTR_BASE_ADDR  
mcr p15,0,r0,c2,c0,0
```

很显然，这将会把 L1_PTR_BASE_ADDR 这个值传递给 CP15 协处理器中的 C2 寄存器，而 C2 寄存器正是负责保存页表基地址的。回想一下，前面提到的地址映射过程，MMU 在进行地址映射时，首先需要拿到页表的基址，然后通过虚拟地址的部分位找到以页表基地址为基础的页偏移地址，读出页表项，然后定位到物理地址的某个区段，最后再结合虚拟地址剩下的位找到对应的物理地址。这个页表基地址就保存在 CP15 协处理器的 C2 寄存器中，也就是说激活 MMU 的第一步工作首先应该是初始化页表基地址。

在完成对页表基地址的初始化工作之后，第二步的工作就是处理好页权限的问题。

```
mvn r0,#0  
mcr p15,0,r0,c3,c0,0
```

回忆一下页权限一节，页表的正确设置只是成功使用 MMU 的一个方面，另一方面，我们必须保证该段内存具有足够的读写权限，而读写权限首先是由域来控制的，如果域权限允许页表项的权限发挥作用，则由页表项中的权限位来决定页的权限。

在这里为了简化步骤，我们把所有 16 个域权限都设置成 0b11，允许所属的页自由访问。而这 16 个域权限恰是由 CP15 协处理器中的 C3 寄存器负责的。细心的读者可能注意到了，代码中我们使用了一条新的指令 mvn，

光论长相，这条指令与 mov 类似，其实，二者的功能也是近似的。mvn 指令首先将 原的操作数取反，然后再赋值给 原操作数，所以， mvn r0,#0 的结果是将 0xffffffff 赋值给 R0。最后， mcr p15,0,r0,c3,c0,0 使得 C3 寄存器的值也为 0xffffffff。这样，系统中的 16 个域权限就全部设置成了 0b11。

到这里，我们已经做好了激活 MMU 前的所有准备工作，已经可以正式激活 MMU 了。

在代码 3-14 中，MMU 的激活是通过下面这两条指令来实现的。

```
mov r0, #0x1  
mcr p15,0,r0,c1,c0,0
```

这两条指令的目的是将 CP15 协处理器的 C1 寄存器中的第一位置为 1，而这一位，掌管了 MMU 激活的大权。一旦 C1 寄存器的第一位被设置成 1，MMU 便被激活了，从这一时刻开始，物理地址便与世隔绝。

最后我们还需要几条空操作，目的是将激活 MMU 之前流水线上的指令清空。关于 ARM 流水线的故事，我们会在后续章节中相继介绍。

读到这里，有的读者可能会产生这样一种感受，我们似乎在对 ARM 还是一知半解的情况下就进行了相当多的程序开发。程序员在涉足一个全新的领域时，其开发过程大多也是这样的，我们几乎没有足够的时间成本，在完全掌握一个新领域之后再去进行开发。从另一个角度来讲，如果我们将有关 ARM 的所有内容都先介绍完之后再介绍操作系统的实现，那这本书可能就会变成 ARM 体系结构与应用之类的书了。这样一来，枯燥不说，也会大大削减大家的学习热情。而本书这种精心设计的内容布局会让读者在不知不觉中就彻底地掌握了原本枯燥无味的知识。不管怎样，有效的学习方法是可以让学习事半功倍的。

我们回来再看一下代码，你会发现针对 MMU 的操作已经彻底完成了。

现在，让我们以之前的启动代码为基础来运行一下这段代码。

首先将代码 3-13 与代码 3-14 的内容保存成文件，命名为“mmu.c”。然后修改原有的 Makefile 文件，将下面的这条代码：

```
OBJS=init.o start.o boot.o abnormal.o
```

更改为：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o
```

修改原有的“boot.c”文件，修改其中的 plat_boot 函数如下：

代码 3-19

```
void plat_boot(void) {
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    while(1);
}
```

代码 3-19 中新添加了三个函数，其中，init_sys_mmu 和 start_mmu 两个函数的具体实现我们已经介绍过了。在 start_mmu 函数运行之后，MMU 被激活，然后我们可以利用函数 test_mmu 来验证一下关于 MMU 这部分代码的正确性，具体实现如下：

代码 3-20

```
void test_mmu(void){
    const char *p="test_mmu\n";
    while(*p){
        *(volatile unsigned int *)0xd0000020=*p++;
    }
}
```

该函数的实现很简单，只是将字符串 "test_mmu\n" 依次写入地址 0xd0000020 处。请回头看一下代码 3-13，我们将物理地址从 0x48000000 开始的一段线性区域映射到虚拟地址 0xc8000000 处。也就是说，代码 3-20 中的虚拟地址 0xd0000020 实际上对应着物理地址 0x50000020，而这个地址正是 2410 的串口 FIFO 寄存器。

从前面的例子中我们已经知道，在 skyeye 虚拟环境下串口是作为默认终端存在的，写向串口中的数据都将输出到标准输出中去，也就是说，字符串 "test_mmu\n" 最终会打印到终端当中去。而与原来的方法不同的是，我们这里是在 MMU 被激活的前提下使用虚拟地址进行操作的，从而验证了前面关于 MMU 这部分代码的正确性。

好了，现在将代码 3-20 中的内容添加到文件“boot.c”中。在终端运行



make 命令，如果一切正常的话，新的“le eos.bin”文件就会生成。在终端运行 skyeye 命令，你将看到如下输出结果：

```
....  
Loaded RAM ./le eos.bin  
start addr is set to 0x30000000 by exec file.  
helloworld  
test_mmu
```

字符串 test_mmu 被打印到终端中，MMU 被成功激活了！

3.4 总结

本章我们主要讨论了操作系统启动时的初始化问题。从理论上探讨了操作系统启动的一般步骤。不过理论终归是理论，一切理论只有应用到实际当中才能真正发挥作用，理论联系实际也是本书秉承的原则，因此，本章中包含有大量的练习代码，这些代码密切结合理论，同时突出重点、通俗易懂，并配有详细的解释。另外，本章也展开了对堆栈理论、系统设计模式等内容的描述。

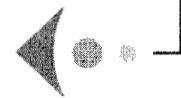
本章还包含了一个操作系统的难点问题，即 MMU 的问题。在这部分内容当中，我们使用了大量的篇幅描述了 MMU 的原理和作用，然后讲解了在 ARM 中虚拟地址到物理地址映射的过程，并就 MMU 与系统控制的重点问题进行了介绍。期间还讲解了 ARM 协处理器的操作方法、GCC 内联汇编技术等重要内容。这些内容五花八门，涉及到了计算机原理的方方面面，讲述的内容虽然重要却很容易被忽视。本章结合大量实例对上述内容做了系统的阐释，使原本晦涩难懂的知识点变得易于理解，当然，对于庞大的操作系统理论来说，这些知识也仅仅是冰山一角，接下来还有更多的内容等待我们去学习。





第4章

打印函数



目前，我们的操作系统已经完成了初始化的工作，现在可以正式进入系统运行阶段了。当然，在这之前我们必须首先解决一个基础性的问题——打印函数的问题。

一个功能强大的打印函数对于操作系统来说是至关重要的。操作系统作为底层软件，调试的时候往往要依赖于硬件调试工具，这本身就加大了操作系统调试的复杂度。对于那些高级操作系统来说，因为涉及到的问题非常复杂，即使有专业的调试工具，也很难实时捕获系统运行时的完整信息，因此，解决好调试的问题就成为了我们的操作系统编码过程中的首要问题。当然，最简单的往往最有效。一个强大的打印函数在大多数情况下足以承担操作系统的调试任务。所以，这里我们利用一章的篇幅来详细地分析一下打印函数的实现原理和方法，并在掌握了这些原理和方法之后亲自实现一个功能完整的打印函数。

在我们接触到的打印函数中，当属 `printf` 这个函数最为大家所熟悉，与之同系列的函数还包括 `fprintf`、`sprintf`、`snprintf`，等等。这一类函数通常被叫做格式化输出函数，是标准 C 库中不可或缺的一部分。这些函数的作用是都能够将“给定的内容”按照“指定格式”输出到“指定目标”内，比如 `fprintf` 和 `sprintf` 这两个函数都有类似的功能，而当指定的目标特指系统标准输出时就可以使用 `printf` 函数了。

明白了这一点，我们也就掌握了编写打印函数的关键点，那就是要处理





好如下三个问题。

- “给定的内容”
- “指定的格式”
- “指定目标”

以下面的这段代码为例：

```
int foo=1;
printf("%d", foo);
```

所谓“给定的内容”，指的是整型变量 `foo`，而“指定的格式”则是由 `%d` 来描述的，那么特定的目标呢？对于 `printf` 函数来说，标准输出就代表了特定目标。

处理好这三个问题，实现一个标准的打印函数便轻而易举了。然而，话虽如此，但如果现在就要求我们立刻给自己的操作系统实现一个标准输出函数，似乎还为时过早。我们不如先来学习一个打印函数的例子，掌握了基本原理和方法之后再动手不迟。

4.1 打印函数实例

代码 4-1 就是一个非常好的例子。

代码 4-1

```
void printf (const char *fmt, ...){
    va_list args;
    uint i;
    char printbuffer[CFG_PBSIZE];
    va_start (args, fmt);

    i = vsprintf (printbuffer, fmt, args);
    va_end (args);

    puts (printbuffer);
}
```

代码 4-1 选自 u-boot，u-boot 是高级嵌入式系统中比较常用的引导加载程序，可应用于多种硬件平台，支持许多嵌入式操作系统。尽管 u-boot 不是操作系统，不具有高级操作系统的基本结构，但是，作为一款优秀的引导程序，u-boot 具有功能强大、实现简单、结构清晰等诸多特点，非常适合学习和借鉴。于是，我们选用 u-boot 来进行标准输出函数的研究。

代码的第一行看起来就有些与众不同。printf 函数首先定义了一个只读的字符指针 const char *fmt，而剩下的参数则使用“...”来表示。我们都知道，这里的“...”表示 printf 函数的参数是可变的。该函数也正是使用了可变参数的处理方法才拥有了非常强大的功能。所以在学习 printf 函数之前，我们应该首先学习一下可变参数的原理和使用方法。

4.1.1 变参函数是如何工作的

我们知道，函数调用的参数传递需要依靠寄存器和堆栈来实现。C 语言默认的函数调用规范是所有参数从右到左依次入栈。这样，这些函数参数在弹出堆栈时，便能够获得一个从左到右的顺序。对于那些固定参数的函数，编译器清楚地知道要将多少个参数压入堆栈，在函数运行时依次从堆栈中弹出各个参数值然后使用。

然而，当传递的参数个数不确定时，编译器便不会知道参数的个数，也不知道应该将多少个参数从堆栈中弹出。

因此，使用可变参数的函数，其参数不能仅包含可变参数，还至少需要一个参数来表示可变参数的个数。

举个例子来说，像下边这种函数形式是不允许的。此时，虽然函数参数可变，但当函数被调用时，程序却没有办法知道需要将多少个参数从堆栈中弹出才合适。因此，这样的代码是不能编译通过的。

```
void foo(...);
```

想要使用可变参数，其格式至少应该类似于如下格式。

```
void foo(int n,...);
```

这里，第一个参数的作用就是直接或间接地通知 foo 函数对 foo 函数的调用传递的参数有多少个，这样，程序至少在编译时不会报错。

若对函数 foo 的调用写成了如下形式：

```
foo(2,arg1,arg2);
```

这表示此次调用 foo 函数将传递两个参数，而写成下边的形式则代表传递 5 个参数：

```
foo(5,arg1,arg2,arg3,arg4,arg5);
```

当然，这里 foo 函数的第一个参数直接充当了传递参数个数的角色。很多时候，可变参数的个数并不是直接传递的，如 printf 函数。在代码 4-1 中，printf 函数的第一个参数是一个字符串，被打印的信息、打印格式以及可变参数个数都包含在这个字符串中。

定参函数和变参函数的调用过程如图 4-1 和图 4-2 所示。

了解完定参函数和变参函数的工作原理，接下来我们来学习一下如何在程序中使用变参函数。

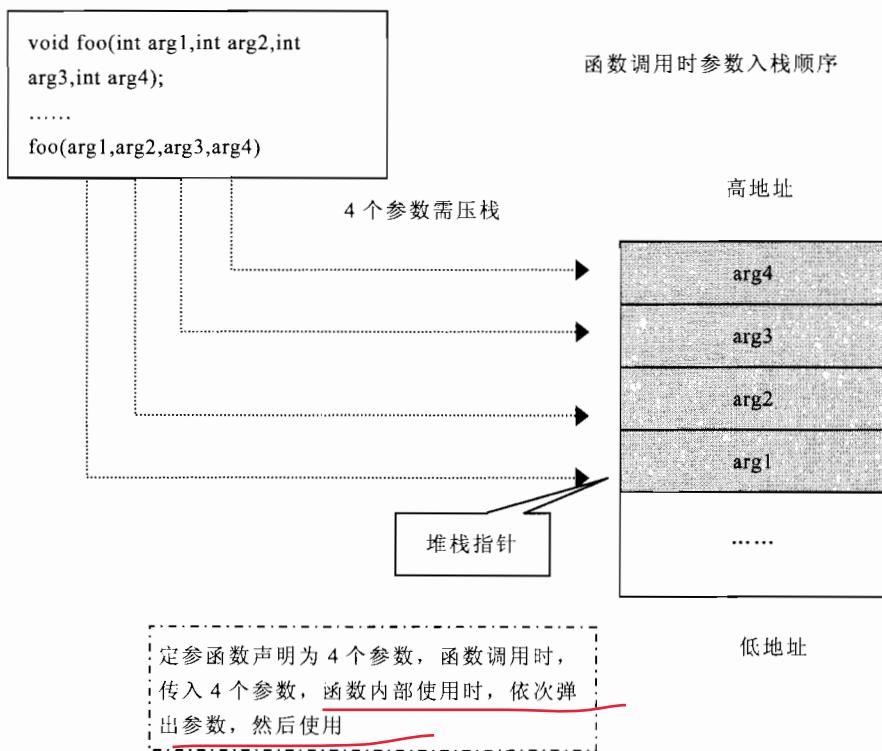


图 4-1 定参函数的调用过程

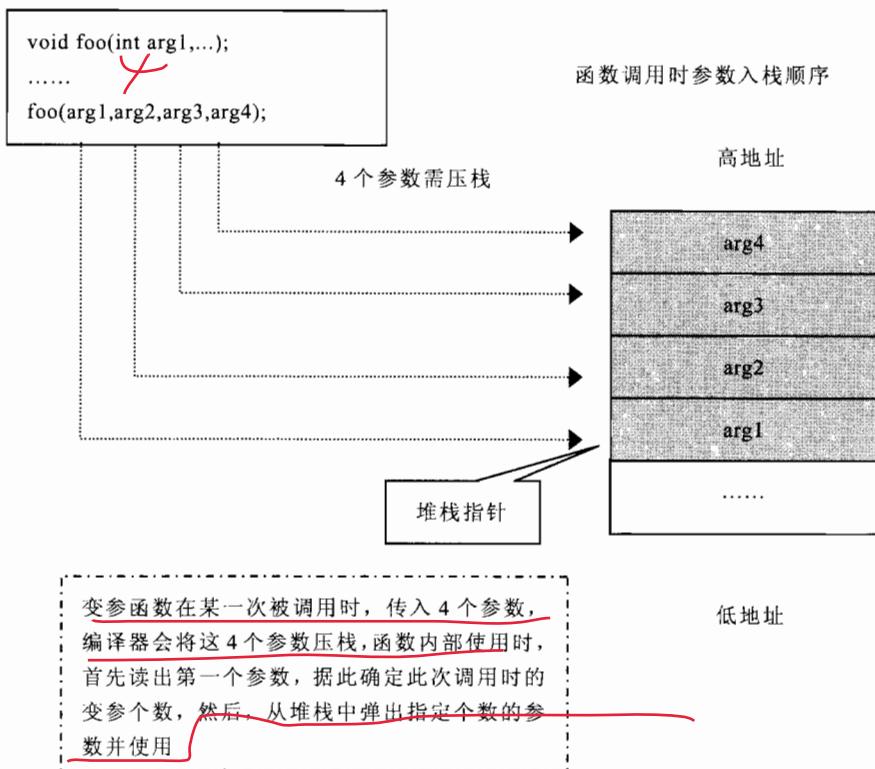


图 4-2 变参函数的调用过程

4.1.2 亲自实现一个可变参数函数

首先，关于变参函数声明方法，使用“...”来表示参数的可变部分，这在前面我们已经介绍过了。

接下来，就是使用一系列方法从变参函数里依次取出函数参数。在 C 语言中，我们有一组变量类型和宏专门用于处理变参函数，它们是 va_list、va_start、va_arg 和 va_end。对于这些变量类型和宏，典型定义如下：

代码 4-2

```
typedef char * va_list;
#define _INTSIZEOF(n) ((sizeof(n)+sizeof(int)-1)&~(sizeof(int)-1))
#define va_start(ap,v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap,t) (* (t*) ((ap+_INTSIZEOF(t)) - _INTSIZEOF(t)))
```

```
#define va_end(ap)      ( ap = (va_list)0 )
```

当然，代码 4-2 并不是唯一的定义方法。在不同的系统中，使用不同的编译器，包含有不同的标准库文件，其定义方法都各不相同。这里我们不急于分析每个宏的作用，首先通过一段实例来了解一下这些类型和宏的使用方法。

代码 4-3

```
void test_num(int num){  
    *(char *)0xd0000020=num+'0';  
}  
  
void test_vparameter(int i,...){  
    int c;  
    va_list argv;  
    va_start(argv,i);  
    while(i--){  
        c=va_arg(argv,int);  
        test_num(c);  
    }  
    va_end(argv);  
}
```

代码 4-3 是以我们已经完成的代码为基础的，结合第 3 章的程序，这段代码也可以正常运行。在程序中我们首先定义了一个变参函数 `test_vparameter`，该函数的第一个参数是一个整型值，表示在某次调用中传递给该函数的变参数数，而且要求所有变参的类型必须一致，这里我们人为地规定可变参数的类型都为 `int`。

在函数 `test_vparameter` 中，首先定义了一个 `va_list` 类型的变量 `argv`。

然后调用宏 `va_start`，这个宏定义的作用是得到可变参数列表中第一个参数的确切地址，并赋值给 `argv`。此时，指针变量 `argv` 就成功指向了函数 `test_vparameter` 参数堆栈的栈顶了。

接下来，函数在每次循环中读出一个参数，并将这些参数传递给 `test_num` 函数进行显示。`test_num` 函数只能简单地打印 0~9 中的数字作为测试函数，这已经足够了。循环控制依靠的是函数 `test_vparameter` 中的第一个参数。当然，这里最关键的一条语句是 `va_arg`，大家可以从代码 4-2 中了解到它的定义。该宏以 `va_list` 变量作为第一个参数，以需要返回的变量的

类型作为第二个参数，这个宏的返回值就是可变参数列表中下一个待处理的参数。例如，在我们的例子里，已经规定了所有的可变参数的类型均为 int，所以在代码 4-3 中，其调用方法总是不变的，而每次宏 va_arg 返回时都将得到可变参数列表中的下一个参数，直至循环结束。

最后，函数 test_vparameter 调用了 va_end 宏结束对 va_list 变量的操作。

代码 4-3 中的例子描述了在 C 语言中操作变参函数的典型用法，由于去除了多余的代码，因此能够清晰地说明变参函数的基本结构。这里，我们把这种结构提取出来总结成表 4-1。

表 4-1 变参函数的使用结构

步骤	说明
va_list argv	定义一个变参变量 argv
va_start(argv,i)	初始化 argv
c=va_arg(argv,int)	在已知变量类型的情况下，获得下一个变参变量
va_end(argv)	结束变参变量操作

既然已经了解到表 4-1 中各个宏定义和变量类型的用法，那么我们不妨结合代码 4-2，深入研究一下这些宏定义究竟是怎样在变参函数中发挥作用的。

先来了解一下宏 _INTSIZEOF(n) 的作用，_INTSIZEOF(n) 被定义成了如下形式：

```
((sizeof(n)+sizeof(int)-1)&~(sizeof(int)-1))
```

假设系统的 sizeof(int) 的值为 4，那么经该宏计算后，会得到一个比 n 大的能整除 4 的数中的最小的那个数。例如，当 sizeof(n)=1、2、3 或者 4 时，该宏的展开结果都为 4，而 sizeof(n) 为 5、6、7 或 8 时，结果为 8。当然，这里 n 并不会随意取值，_INTSIZEOF(n) 是用来计算各种数据类型按照 4 字节对齐后的结果的。也就是说，对于 char 和 short 类型的数据，因为不满一个 int 类型的内存空间，所以要按照 int 类型对齐，也就是 4 个字节，而对于类似于 long long 这种数据类型，则需要 8 个字节。

在宏 _INTSIZEOF(n) 的帮助下，我们可以根据一个变量的类型来计算该变量在内存中会占据多少个字节，从而正确定位内存中的参数位置。va_start(ap,v) 就是一个非常典型的例子，它的定义如下：

```
( ap = (va_list)&v + _INTSIZEOF(v) )
```

va_start(ap,v)实际上就是在对变量 ap 进行赋值,其值为变量 v 的地址加上 v 的类型按照 4 字节对齐后的偏移。因为参数在内存中的分布是连续的,又因为传递给宏 va_start 的参数 v 都是变参函数里可变参数列表中的前一个参数。所以,变量 ap 最终的结果是一个指向变参列表首地址的指针。

紧接着在 va_start(ap,v)运行完成后,我们就可以使用宏 va_arg(ap,t)来依次提取变参列表中的参数,其定义如下:

```
( * (t *) ((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
```

这里,宏 va_arg(ap,t)中的参数 t 只能输入变量类型,该宏恰恰也是根据变量的类型来正确提取参数的。首先,宏 va_arg 将变量 ap 的值在原有的基础上加上一个对齐后的偏移,此时,ap 已经指向变参到表中的下一个变量了,然后表达式在减去一个同样的值后被强行转成 t 类型,此时我们正好取出了变量 ap 所指向的那个参数,同时更新变量 ap,让它指向参数列表中的下一个参数以便下次使用。

最后,宏 va_end(ap)将变量 ap 赋值为零,从此不再使用。

这便是 C 语言中,变参函数的原理和基本操作方法。

本着在实践中学习的原则,接下来我们来运行一下代码 4-3。

首先,请在原有代码的基础上新建一个新的文件,命名为“print.c”。然后将代码 4-2 和代码 4-3 的内容复制到“print.c”文件中。接下来修改 Makefile 文件,将原来的:

```
OBJS=init.o start.o boot.o abnormal.o mmu.o
```

改为:

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
```

最后,将“boot.c”文件中的 plat_boot 函数修改为如下形式:

代码 4-4

```
void plat_boot(void){  
    extern void test_vparameter(int,...);  
    int i;  
    for(i=0;init[i];i++){  
        init[i]();  
    }  
    init_sys_mmu();
```



```

    start_mmu();
    test_mmu();
    test_vparameter(3,9,8,7);
    test_vparameter(2,6,5);
    while(1);
}

```

编译源代码，并运行 skyeye 命令，在终端你将会看到如下结果：

```

.....
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:, desc_out:, converter:
SKYEye: use arm920t mmu ops
Loaded RAM ./le eos.bin
start_addr is set to 0x30000000 by exec file.
helloworld
test_mmu
98765

```

该结果正是我们两次调用 test_vparameter 函数的输出结果。

4.1.3 实现打印函数中的格式转换

在掌握了变参函数的用法之后，代码 4-1 的原理就不难理解了。在代码 4-1 中，printf 函数首先为变参函数的使用做了适当的准备，然后调用函数 vsprintf 将变参列表 args 中的变量按照 fmt 中规定的格式保存到临时缓存 printbuffer 中，最后再调用 puts(printbuffer) 函数将临时缓存中的字符串数据打印到终端中去。

在这个过程中，vsprintf 函数无疑起到了重要的作用，是整个 printf 函数的核心，下面就让我们来共同学习一下 vsprintf 函数的实现方法。

代码 4-5

```

int vsprintf(char *buf, const char *fmt, va_list args){
    unsigned NUM_TYPE num;
    int base;
    char *str;
    int flags;
    int field_width;

```

```
int precision;
int qualifier;
str = buf;

for (; *fmt ; ++fmt) {
    if (*fmt != '%') {
        *str++ = *fmt;
        continue;
    }

    /* process flags */
    flags = 0;
    repeat:
        ++fmt; /* this also skips first '%' */
        switch (*fmt) {
            case '-': flags |= LEFT; goto repeat;
            case '+': flags |= PLUS; goto repeat;
            ....
        }
        ....
}

base = 10;

switch (*fmt) {
case 'c':
    ....
    *str++ = (unsigned char) va_arg(args, int);
    ....
    continue;

case 's':
    str = string(str, va_arg(args, char *), \
                 field_width, precision, flags);
    continue;
    ....
case 'x':
    base = 16;
    break;

case 'd':
case 'i':
    flags |= SIGN;
case 'u':
```



```

        break;

default:
    *str++ = '%';
    if (*fmt)
        *str++ = *fmt;
    else
        --fmt;
    continue;
}

str = number(str, num, base, field_width, precision, flags);
}
*str = '\0';
return str - buf;
}

```

代码 4-5 有些长，出于篇幅限制，我们删除了许多内容，只留下主干部分，以使结构更清晰。在讲解这段代码之前我们需要首先复习一下 printf 函数的格式符。

每一个懂 C 语言的人，都会了解表 4-2 的含义，每一本讲 C 语言的书都会涉及表 4-2 的内容。当然，printf 函数的输出格式符还远不止这些，这里我们也没有必要一一列举。

表 4-2 printf 函数格式符节选

常用格式符	说明
x	以 16 进制形式输出
u	以无符号整型形式输出
c	以字符形式输出
d	以有符号整型形式输出
s	以字符串的形式输出
o	以 8 进制形式输出

有的时候，打印函数并不需要支持所有的格式符，比如，在 Linux 内核当中，打印函数 printk 就不具备对浮点数的数据进行输出的能力。当然，表 4-2 中的这些格式符还可以进行组合变化。例如，在格式符前面加上“-”，表示字符串靠左，右侧补空格。如果在格式符前面出现“l”，表示以长类型输出，等等。

从学习的角度来讲，数据仍然是原来的数据，只不过我们需要根据不同的格式要求对同样的数据做不同的解析。而这种解析方法，针对数字、字符

或字符串，又完全不同。所以，我们只需要讲解几种格式的转换方法，就可以触类旁通。

`vsprintf` 函数的第一个参数指向了一段缓冲区，该缓冲区中的字符将会输出到终端上去。所以我们要做的就是，根据格式要求将需要转换的字符进行转换后放到这段缓冲区里，而将不需要转换的字符原样复制到缓冲区中。

我们知道，`printf` 函数中的格式符都需要以“%”开头，所以，代码 4-5 首先要将“%”（包括“%”之前的内容）剥离，得到待解析的字符。于是，程序需要在循环中依次读出缓存中的每一个字符，判断是否为“%”，如果是，就把它后边的字符作为解析格式，如果不是“%”，则将该字符原封不动地复制到缓冲区中，然后继续判断下一个字符。这些操作只需通过一个简单的 if 判断就能实现。

在读出“%”后边的字符之后，我们要判断该格式符是否是可变化的形式，比如前面是否带有“+”、“l”，等等。在代码 4-5 中，这是通过一个 `switch…case` 结构实现的。

在处理完所有的变化形式后，代码 4-5 就真正进入了格式解析的过程。该过程也是由一个 `switch…case` 结构实现的。`printf` 函数的格式符虽然很多，但基本上可以被分成三大类：数字、字符、字符串。其中，对字符的解析最简单，所以我们先来看一下字符解析。

解析字符之所以简单，是因为我们无须做转换。在代码 4-5 中，如果格式符为“c”，则将可变参数列表中的数据直接提取出来，强制转换成 `char` 类型后，直接放到缓冲区中。

如果待转换的数据类型是字符串，则从可变参数列表中拿到的将是指向该字符串的指针。那么我们需要做的工作就是，将该指针指向的数据复制到缓冲区中，所有这些工作都可以交给 `string` 这个函数来实现。

如果格式符是任何一种数字形式，那么我们就需要将可变参数列表中的数字根据不同的进制要求转换成字符串的形式，然后再进行复制。因此，在转换的过程中，我们只需要弄清楚两件事——进制和符号，就可以完成数字到字符串的转换了。在代码 4-5 里，进制数由一个名为 `base` 的变量保存，而符号则有 `flag` 标志保存。搞清楚这两个问题后，就可以调用 `number` 函数对数字做统一的处理了。

当可变参数列表中的所有参数都处理完毕后，`vsprintf` 函数也就完成了

它的任务，此时缓冲区中的数据已经转换完成，只需要将这些数据输出到终端即可。在代码 4-1 中，这最后的收尾工作是交给 puts 函数来完成的。对终端的定义，在不同的环境下是完全不同的，很多嵌入式系统并没有负责显示的设备，对于这样的系统，往往依靠串口来打印信息。u-boot 作为一个典型的嵌入式系统引导加载程序，串口也是作为默认终端出现的。因此，这里的 puts 函数其实就是将缓冲区中的数据顺序发送到串口 FIFO 寄存器中。

至此，u-boot 中的 printf 函数就介绍完了。

读者也可以读一下其他程序的打印函数，你会发现，不同环境下的打印函数从结构上看非常相似。这种现象并不是偶然的，打印函数无论在哪种环境里，其功能都是一致的，而像 u-boot 中的这种函数结构，又非常有利于功能的实现。所以，打印函数的这种结构也几乎成为了该类函数的一个标准实现方法。

4.2 实现自己的打印函数

在我们自己的操作系统里，也可以仿造 u-boot 中的打印函数来实现数据的格式化输出。这里，我们不妨根据打印函数的原理实现一个属于自己的版本，完整的代码如下。

代码 4-6

```

typedef char * va_list;
#define _INTSIZEOF(n) ((sizeof(n)+sizeof(int)-1)&~(sizeof(int)-1))
#define va_start(ap,v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap,t) (*((t *) (ap += _INTSIZEOF(t)) - _INTSIZEOF(t)))
#define va_end(ap) (ap = (va_list)0)

const char *digits="0123456789abcdef";
char numbers[68];

static char print_buf[1024];

#define FORMAT_TYPE_MASK 0xff00
#define FORMAT_TYPE_SIGN_BIT 0x0100

```

```
#define FORMAT_TYPE_NONE      0x000
#define FORMAT_TYPE_CHAR       0x100
#define FORMAT_TYPE_UCHAR      0x200
#define FORMAT_TYPE_SHORT      0x300
#define FORMAT_TYPE USHORT     0x400
#define FORMAT_TYPE_INT        0x500
#define FORMAT_TYPE_UINT       0x600
#define FORMAT_TYPE_LONG       0x700
#define FORMAT_TYPE ULONG      0x800
#define FORMAT_TYPE_STR         0xd00
#define FORMAT_TYPE_PTR        0x900
#define FORMAT_TYPE_SIZE_T     0xb00

#define FORMAT_TYPE(x)          ((x)&FORMAT_TYPE_MASK)
#define SET_FORMAT_TYPE(x,t)    do{\
    (x)&=~FORMAT_TYPE_MASK;(x)|=(t);\
}while(0)
#define FORMAT_SIGNED(x)        ((x)&FORMAT_TYPE_SIGN_BIT)

#define FORMAT_FLAG_MASK        0xfffff0000
#define FORMAT_FLAG_SPACE       0x10000
#define FORMAT_FLAG_ZEROPAD    0x20000
#define FORMAT_FLAG_LEFT        0x40000
#define FORMAT_FLAG_WIDTH      0x100000

#define FORMAT_FLAG(x)          ((x)&FORMAT_FLAG_MASK)
#define SET_FORMAT_FLAG(x,f)   ((x)|=(f))

#define FORMAT_BASE_MASK        0xff
#define FORMAT_BASE_O           0x08
#define FORMAT_BASE_X           0x10
#define FORMAT_BASE_D           0x0a
#define FORMAT_BASE_B           0x02

#define FORMAT_BASE(x)          (FORMAT_BASE_MASK&(x))
#define SET_FORMAT_BASE(x,t)    do{ (x)&=~FORMAT_BASE_MASK;(x)|=(t); }while(0)

#define do_div(n,base) ({ \
    int __res; \
    __res = ((unsigned int) n) % (unsigned int) base; \
    n = ((unsigned int) n) / (unsigned int) base; \
    __res; })


```

```

void __put_char(char *p,int num){
    while(*p&&num--){
        *(volatile unsigned int *)0xd0000020=*p++;
    }
}

void * memcpy(void * dest,const void *src,unsigned int count)
{
    char *tmp = (char *) dest, *s = (char *) src;
    while (count--)
        *tmp++ = *s++;
    return dest;
}

char *number(char *str, int num,int base,unsigned int flags){
    int i=0;
    int sign=0;

    if(FORMAT_SIGNED(flags)&&(signed int)num<0){
        sign=1;
        num=~num+1;
    }

    do{
        numbers[i++]=digits[do_div(num,base)];
    }while(num!=0);

    if(FORMAT_BASE(flags)==FORMAT_BASE_O){
        numbers[i++]='0';
    }else if(FORMAT_BASE(flags)==FORMAT_BASE_X){
        numbers[i++]='x';
        numbers[i++]='0';
    }else if(FORMAT_BASE(flags)==FORMAT_BASE_B){
        numbers[i++]='b';
        numbers[i++]='0';
    }

    if(sign)
        numbers[i++]='';

    while (i-- > 0)
        *str++ = numbers[i];
}

```



```
    return str;
}

int format_decode(const char *fmt,unsigned int *flags){
    const char *start = fmt;

    *flags &= ~FORMAT_TYPE_MASK;
    *flags |= FORMAT_TYPE_NONE;
    for (; *fmt ; ++fmt) {
        if (*fmt == '%')
            break;
    }

    if (fmt != start || !*fmt)
        return fmt - start;

    do{
        fmt++;
        switch(*fmt) {
            case 'l' :
                SET_FORMAT_FLAG(*flags,FORMAT_FLAG_WIDTH);
                break;
            default:
                break;
        }
    }while(0);

    SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
    switch (*fmt) {
        case 'c':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_CHAR);
            break;

        case 's':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_STR);
            break;

        case 'o':
            SET_FORMAT_BASE(*flags,FORMAT_BASE_O);
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
            break;

        case 'x':
            break;
    }
}
```

```
case 'X':
    SET_FORMAT_BASE(*flags,FORMAT_BASE_X);
    SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
    break;

case 'd':
case 'i':
    SET_FORMAT_TYPE(*flags,FORMAT_TYPE_INT);
    SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
    break;
case 'u':
    SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
    SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
    break;

default:
    break;
}
return ++fmt-start;
}

int vsnprintf(char *buf, int size, const char *fmt, va_list args){
    int num;
    char *str, *end, c,*s;
    int read;
    unsigned int spec=0;

    str = buf;
    end = buf + size;

    if (end < buf) {
        end = ((void *)-1);
        size = end - buf;
    }

    while (*fmt) {
        const char *old_fmt = fmt;
        read = format_decode(fmt, &spec);
        fmt += read;

        if((FORMAT_TYPE(spec))==FORMAT_TYPE_NONE) {
            int copy = read;
```

```
    if (str < end) {
        if (copy > end - str)
            copy = end - str;
        memcpy(str, old_fmt, copy);
    }
    str += read;

} else if(format_flag_width) {
    //do nothing
} else if(format_type(spec)==FORMAT_TYPE_CHAR) {
    c = (unsigned char) va_arg(args, int);
    if (str < end)
        *str = c;
    ++str;
} else if(format_type(spec)==FORMAT_TYPE_STR) {
    s = (char *) va_arg(args, char *);
    while(str<end&&*s!='\0'){
        *str++=*s++;
    }
} else{
    if(format_type(spec)==FORMAT_TYPE_INT){
        num = va_arg(args, int);
    } else if(format_type(spec)==FORMAT_TYPE ULONG){
        num = va_arg(args, unsigned long);
    } else if(format_type(spec)==FORMAT_TYPE_LONG){
        num = va_arg(args, long);
    } else if(format_type(spec)==FORMAT_TYPE_SIZE_T){
        num = va_arg(args, int);
    } else if(format_type(spec)==FORMAT_TYPE USHORT){
        num = (unsigned short) va_arg(args, int);
    } else if(format_type(spec)==FORMAT_TYPE_SHORT){
        num = (short) va_arg(args, int);
    } else{
        num = va_arg(args, unsigned int);
    }
    str=number(str,num,spec&FORMAT_BASE_MASK,spec);
}
}

if (size > 0) {
    if (str < end)
        *str = '\0';
    else
```

```

        end[-1] = '\0';
    }
    return str_buf;
}

void printk(const char *fmt, ...)
{
    va_list args;
    unsigned int i;

    va_start(args, fmt); 10) 4
    i = vsnprintf(print_buf, sizeof(print_buf), fmt, args);
    va_end(args); | 0

    __put_char(print_buf, i);
}

void test_printk(void){
    char *p= "this is %s test";
    char c='H';
    int d=256;
    int k=0;
    printk("testing printk\n");
    printk("test string :::: %s\n");
    printk("char :::: %c\n");
    printk("digit :::: %d\n");
    printk("test X :::: %x\n");
    printk("unsigned :::: %u\n");
    printk("test zero :::: %d\n", p, c, d, d, d, k);
}

```

代码 4-6 是一套相对完整的打印函数，我们按照操作系统源代码的一般原则，效仿 Linux 给它取名为 printk。这段打印函数功能相对较全，代码的组织与 uboot 中的 printf 函数稍有不同，但结构是一致的。因为前面我们已经对打印函数各个技术要点做了深入的讲解，所以这里就不对代码 4-6 进行具体分析了，相信读者读懂这段代码是没有问题的。

要运行这段代码，我们需要将其保存成文件，名字为“print.c”，用来替换原来的文件。

然后修改“boot.c”中的 plat_boot 函数，如下：

代码 4-7

```

void plat_boot(void){
    int i;
}

```



```
for(i=0;init[i];i++){
    init[i]();
}
init_sys_mmu();
start_mmu();
test_mmu();
test_printf();
while(1);
}
```

为了验证打印函数的正确性，在代码 4-7 中我们使用了一个 test_printf 函数，将 printf 函数以各种形式进行数据输出。

在所有代码都整理完成后，在终端运行 make 命令进行编译，如果成功地生成了二进制文件，则运行 skyeye 命令，你将会看到如下结果：

```
.....
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff0, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:., desc_out:., converter:
SKYEYE: use arm920t mmu ops
Loaded RAM ./le eos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printf
test string :::: this is %s test
test char :::: H
test digit :::: -256
test X :::: 0xffffffff00
test unsigned :::: 4294967040
test zero :::: 0
```

至此，我们已经拥有了一个功能强大的打印函数，为我们的操作系统的后续实现进一步扫清了障碍。

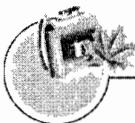
4.3 总结

打个比方说，打印函数就像是计算机的一张嘴，从中可以将它的想法、情绪讲给我们听，这是计算机与人的一种最基本的交流手段。

相信读者都应该知道巴别塔的故事，在《圣经·旧约·创世记》第十一章中说，当时人类联合起来想要兴建希望能通往天堂的高塔，为了阻止人类的计划，上帝让人类说不同的语言，由于人类相互之间不能沟通，计划因此失败。从这个故事中我们也可以看出，交流在人与人之间是多么重要。

然而，人与计算机呢？毫无疑问，一个成功的交流方式会给人与计算机之间的关系带来巨大的变化。想一想计算机的发展历史吧，从最早的纸带输入到中期的文本模式的输出，再到底现在的图形化窗口，精美的页面、炫酷的特效、语音、字符、图像等多种输入方式，人们在追求人与计算机之间的畅快交流的道路上是永无止境的。

由于硬件设施与本书篇幅的限制，我们不能够在我们的操作系统中实现人与机器间多元的沟通方式。然而，打印函数作为机器对人最原始最基本的交流手段，却是一个里程碑。从这一刻起，我们已经为属于我们自己的操作系统插上了翅膀，能否让它飞翔起来，就看您是否敢想敢做了！



第5章

中断处理



什么是中断？通俗地讲，中断就是你正在干的这件事没干完，突然要停下来去干另一件更加紧急的事。所以，中断就是相对于当前事情的那件更加紧急的事情。

换个角度讲，工作无非分成两种：普通的事和紧急的事。这两种事各有各的特点，所谓普通的事，有的时候也许会很简单，而有的时候却并不简单，一些非常复杂的工作也可以属于普通的事，这些事不是一朝一夕就能完成的，因此，即便耽搁几天也无大碍，而紧急的事，特点就是很急，通常要求你尽快做完。大多数情况下，紧急的事并不复杂，三下五除二就可以完成，但却要求你尽快完成，要不然就会有麻烦。在生活中，这些都是再明白不过的道理了，可要是把这些道理放到操作系统当中去理解，结果却大不一样。

在操作系统当中，我们可以把那些普通的事理解成进程，而把那些紧急的事理解成中断。无论是进程还是中断，对于 CPU 来讲，都是它份内的工作。只不过这两种工作各有各的特点，所以应当区别对待。本章中我们将会涉及操作系统的底层核心之一——中断的处理。

从纯技术的角度出发，每一种 CPU 都有自己独立的中断处理方法。而操作系统又是建立在硬件的基础之上，所以要深究操作系统中的中断处理方法，就不得不提 CPU。另一方面，成熟的操作系统都会尽量摆脱硬件的束缚，为自身或上层应用层提供一种与硬件无关的抽象方法，就像标准库那样，无论在什么硬件平台、什么操作系统下，其编程方法都是统一的，这就是一种很成功的抽象。操作系统中也是处处离不开抽象，中断的处理过程自然也不例外。



在本章，我们需要解决操作系统的中断处理问题，毫无疑问，首先需要介绍 ARM 这种时下最流行的处理器，从它的中断处理方法说起。

5.1 ARM 的中断

对于 ARM 体系结构来说，对中断的处理大致认为可以分成两个阶段：体系统一的处理过程和体系独立的处理过程。所谓体系统一的处理过程指的是同一系列芯片的中断处理方法是一致的，而体系独立的处理过程则是指不同款芯片的处理方法完全不同，我们来举个例子。

以 s3c2410 为例，当中断发生时，芯片首先要保存当前运算环境，比如标志位的值、返回地址，等等。然后，程序跳转到一个固定的地址去执行。上述过程对于整个 ARM9 系列的芯片来说，都是统一的，也就是说无论是 s3c2410 还是 ep9312，或是其他什么芯片，这个过程是完全一致的。CPU 都需要进行返回地址的保存、模式的切换、PC 的跳转等操作。

但是，中断处理就这样完成了吗？显然没有。在 s3c2410 中，如果我们要处理串口中断，就需要将相应寄存器的 pending 位清除，这样中断才不会重复触发，对于复杂的中断处理过程，我们可能需要局部地禁止某些中断，而同时又允许其他设备产生中断，这些都需要有一个专门的中断控制器来参与工作。而中断控制器属于哪段地址空间、应该怎样控制等问题，都是各芯片的生产商自定义的。例如，在 s3c2410 中，中断控制器位于 0x4A000000 处，而对于 at91rm92 系列芯片，中断控制器则位于 0xFFFFF000 处。所以，同种硬件的中断处理程序在不同的芯片中的处理方法完全不同，也就是说，中断处理程序的这部分工作的处理方法是彼此独立的。

在本章中，中断的这两部分工作我们都会介绍，但显然，体系结构统一的处理方法将会是本章的重点。另外，如不明确指出，本章所涉及到的一些对 ARM 体系结构的描述针对的都是 ARM v7 之前的版本。新版的结构变化较之于传统版本不能说变化不大，不过无论怎样，这对我们能否深入理解嵌入式操作系统的原理不会有太大影响。





5.1.1 统一的异常和中断处理

前面我们大量提到了中断这个概念。但是，如果要深入到技术细节，把 ARM 的中断处理过程彻底说清楚的话，我们就不得不介绍异常这个概念。

5.1.1.1 ARM 的异常

所谓异常，指的就是中止了程序正常执行的过程而不得不去完成的一些特殊工作，如芯片复位、取址失败、指令未定义，等等，具体的异常类型如表 5-1 所示。

表 5-1 ARM 的异常状态

异常类型	所属模式	说明
芯片复位	SVC	当芯片复位发生时产生
未定义指令	UND	指令不能被芯片识别时产生
软中断	SVC	软件调用 swi 指令时产生
预取址中止	ABT	没有权限访问存储器时产生
数据中止	ABT	没有权限访问存储器时产生
中断	IRQ	硬件进行中断请求时发生
快速中断	FIQ	硬件进行快速中断请求时发生

我们通常所说的中断其实也是一种异常，这里的中断包括外部硬件产生的外部中断和由芯片内部硬件产生的内部中断。由中断产生的异常和其他异常，从处理方法的角度来看并没有任何区别，所以我们可以把这些异常统一起来研究。

5.1.1.2 模式与寄存器

首先，我们要从 ARM 的寄存器开始说起。关于 ARM 寄存器的一些知识，在前面的章节中已经有过介绍了。在这里，我们会将重点放在各个模式与寄存器的关系中。

我们知道，ARM 一共有 37 个通用寄存器。当然这并不意味着在写程序的时候，可以同时使用这 37 个寄存器中的任何一个。因为这些寄存器都是根据模式分组的，所以每个模式都有属于各自模式的私有寄存器，程序工作

在某种处理器模式时，只能访问属于该模式的私有寄存器和公有寄存器。这里我们还需要详细说一说另外一个概念，那就是处理器模式。

正如前文中介绍的那样，ARM一共有7种处理器模式，分别是中止模式（ABT）、中断模式（IRQ）、快速中断模式（FIQ）、管理模式（SVC）、系统模式（SYS）、未定义模式（UND）以及用户模式（USR）。其中，除用户模式和系统模式之外的5种处理器模式又被称为异常模式，同时，把除用户模式之外的其他6种处理器模式称为特权模式。

处理器之所以被设计出支持多种模式是为了能够更好地处理各种异常。

在正常情况下，一个普通程序可能会运行在用户模式或系统模式下，而当中断发生时，ARM就会自动切换到中断模式去处理中断，处理完成后，又会回到用户模式或系统模式下继续之前的工作。因为每一种模式都包含有相应的私有资源，因此可以保证在处理中断时，原来的程序环境不会被新的环境破坏，从而保证了系统的正常运行。

因为异常有很多种，所以ARM处理器在设计时，就定义了5种异常模式，分别处理可能出现的异常。当然，将各种异常分得太细，势必会使程序的设计变得复杂，这也是ARM异常模式的一个主要缺点。所以在最新的ARM v7的体系结构中，人们便打破了这种模式设计的思路，使异常的处理更加简化。当然这并不是我们需要讨论的问题，感兴趣的读者朋友可以进一步查阅相关文档。

对ARM寄存器和处理器模式的完整描述如图5-1所示。图5-1列举了在7种处理器模式下，ARM允许访问的全部寄存器。图中标注有底纹的寄存器代表该模式下的私有寄存器，而未标注底纹的代表共有寄存器。从图中我们可以看出，除用户模式和系统模式之外的处理器模式都有各自的私有寄存器，而用户模式和系统模式自身则共同使用同一组寄存器。

共有寄存器只有一组，或者可以这样解释，所有模式下访问的共有寄存器都是同一个寄存器。也就是说，系统中的R0、R1、R2、R3、R4、R5、R6、R7、R8、R15以及CPSR各寄存器只有唯一一个。所以，如果在中断模式下和管理模式下都去访问R0的话，其实是在访问同一个寄存器。

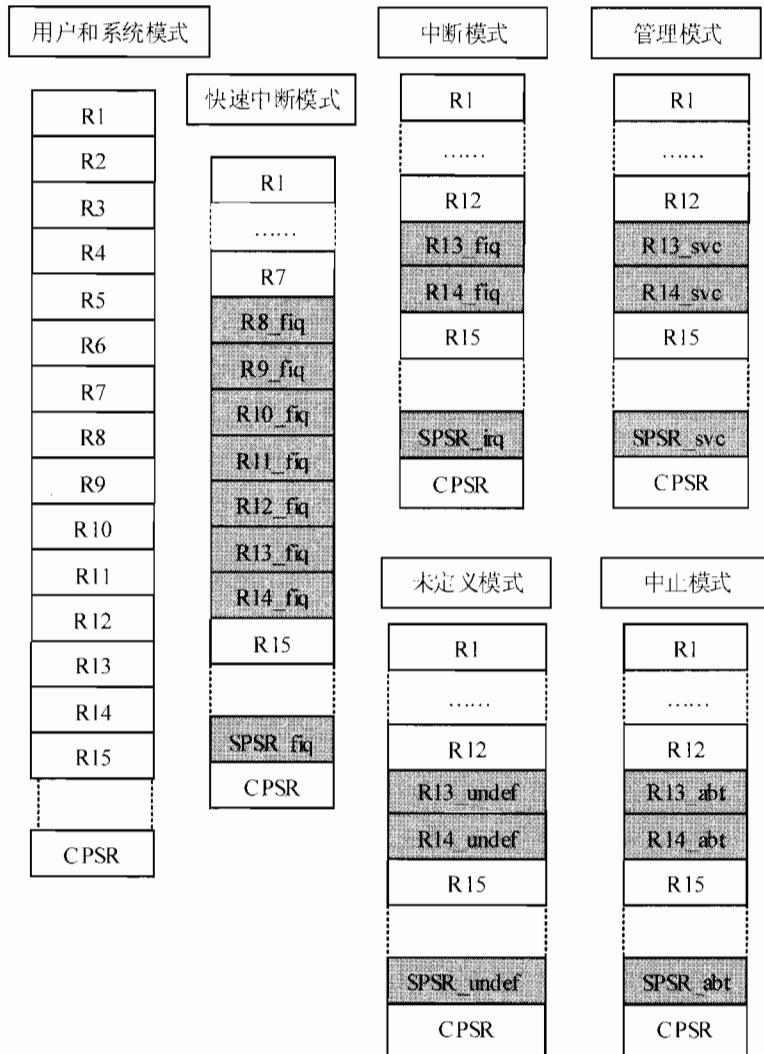


图 5-1 ARM 模式及其寄存器

那么这是不是就意味着，在中断模式或管理模式下，就不能访问 R13 或 R14 了呢？这个问题应该这样理解，单从程序的角度来看，我们可以在中断模式下使用如下代码：

```
mov r14, r2
```

但这里的 R14 指的并不是共有寄存器 R14，而是指中断模式下的私有寄存器 R14_irq。所以上面这条指令的真正意思是：

```
mov r14_irq, r2
```

再比如说，如果有一条指令：

```
mov r0,r13
```

那么在中断模式下，这条指令其实代表了：

```
mov r0,r13_irq
```

而在管理模式里，意思又变成了：

```
mov r0,r13_svc
```

可见没有一种直接的办法能够保证在带有私有寄存器的特权模式下，访问与私有寄存器相对应的共有寄存器的内容。这就为模式切换时对程序的保护提供了一个天然的屏障。

如果从寄存器的应用这个角度来看，除 R15、CPSR 以及各个 SPSR 寄存器以外，所有寄存器并不区分三六九等。但诸如寄存器 R13、R14 寄存器等，还有特殊的作用。这一点在前面的章节中也曾提到过，这里我们再来更详细地介绍一下。

寄存器 R13 用来保存堆栈指针，而寄存器 R14 则保存着上一段程序的返回地址。结合图 5-1，读者朋友们就能理解为什么多数特权模式都含有 R13 和 R14 这两个私有寄存器了。给这些特权模式都赋予一个堆栈指针寄存器，就能保证在这些特权模式下堆栈空间的独立性，避免了互相干扰。而 R14 因为保存有返回值信息，所以如果某种特权模式被动发生了，那么 CPU 会将上一个模式的 R15 寄存器的值保存在相应特权模式下的私有寄存器的 R14 中，从而保证了程序在模式切换时能够正确执行。正是因为寄存器 R13 和 R14 的这两个作用，所以它们都有自己的别名，分别是 SP 和 LR，这样的别称也是可以用在程序里的。

快速中断相对于其他特权模式又稍有不同，除了 R13 和 R14 外，快速中断模式又额外多出了 5 个私有寄存器，这些私有寄存器是专门提供给快速中断模式去迅速地执行中断处理程序的。

通过前面的描述我们知道，共有寄存器是在各个模式下都允许访问的，这在模式切换时会带来一些问题，比如，程序原本运行在系统模式下，寄存器 R0~R12 的值都保存了系统模式下的数据，此时突然发生了中断，程序跳转到中断模式下工作，能够使用的私有寄存器只有 R13 和 R14 以及



SPSR_irq。所以在中断模式下，如果需要使用某些共有寄存器，那就必须先将这些寄存器的值保存到中断模式的堆栈中，这一点我们在前面的章节中也是提到过的。要知道把寄存器压入堆栈是需要花时间的。如果运行在中断模式下的程序需要使用从 R0~R12 的所有寄存器，就需要将这些寄存器全部都压入堆栈。而如果是在快速中断模式下处理中断，又会有怎样的效果呢？如果是在快速中断模式下，最多只需要将寄存器 R0~R7 的内容压入堆栈，从而提高了中断处理的速度。

除了 R13 和 R14 这两个寄存器外，寄存器 R15 也比较特殊。简单地说，它的值代表了当前程序的运行位置。例如，一段程序从内存中 0x0 处开始，至 0x200 处结束，那么当芯片上电时，R15 寄存器的值被初始化为零。CPU 从该寄存器所指向的位置读出第一条指令运行，同时 R15 的值自加 4 个字节。当然，由于多级流水线的设计，R15 的真实情况可能比这要复杂一些，关于这一点我们稍后再做讨论。同样的道理，我们也可以直接改变 R15 的值来实现跳转，如下面这条指令：

```
mov r15, lr
```

该指令可以将保存在 LR 寄存器中的返回值赋给 R15，实现程序的返回。正因为 R15 负担着记录程序代码地址的重任，所以 R15 寄存器也被称为程序计数器，别名是 PC。

另外，CPSR 寄存器以及各模式下的 SPSR 寄存器可以说更加特殊。与 PC 寄存器相同，这些寄存器也不能存储运算结果。CPSR 全名叫做当前程序状态寄存器，其中记录的都是程序当前的运行状态，如当前正在运行的程序属于哪种状态、当前程序的运行结果是否有溢出，等等。不同系列的 ARM 核，CPSR 寄存器的结构可能会有略微的差别，如图 5-2 所示，给出了 ARM 920t 系列中 CPSR 的基本格式。

在图 5-2 中，N、Z、C、V 这 4 个位为条件标志位，某些指令的执行结果会影响这些位，如 cmp。cmp 指令用来比较两个操作数的值是否相等，当二者相等时，CPSR 当中的 Z 位就会变成 1，表示本次运算的结果为 0。有些指令可以在后边加上一个“S”后缀，强行更新标志位，如 subs 指令，就是在做减法的同时，根据结果更新标志位。



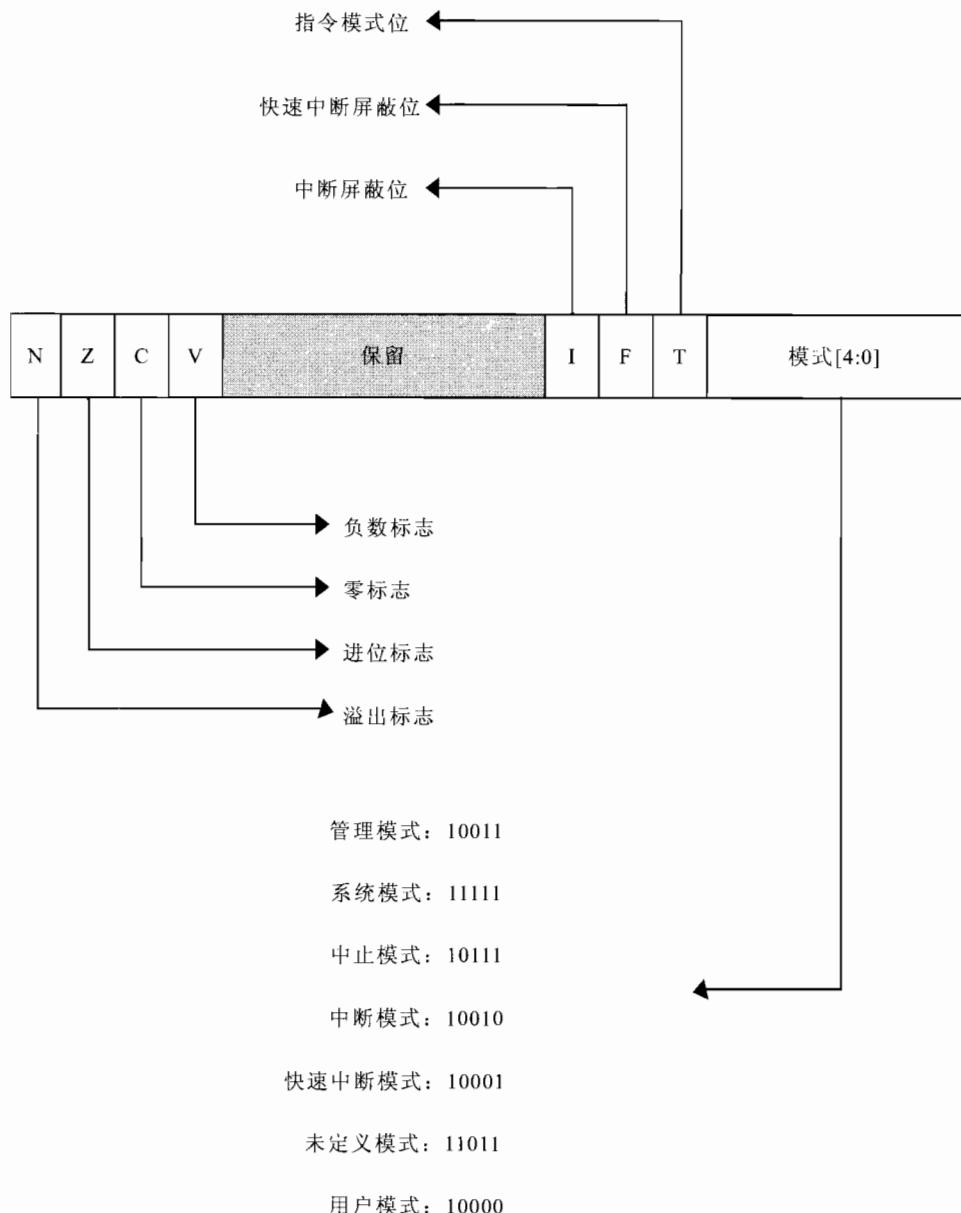


图 5-2 ARM 的 CPSR 寄存器

CPSR 中的 I 位和 F 位分别表示是否允许产生中断和快速中断。相应位的值为 1 代表禁止相应中断，为 0 则代表允许相应中断。在 ARM v6 以前的

体系结构中，并没有提供专门的开关中断指令。因此通常的做法是首先读出 CPSR 当前值，使用 BIC 或 ORR 指令更改相应的位，然后再将新的结果写回 CPSR 中。

CPSR 寄存器的后 5 位是模式位。当异常模式被触发时，硬件会自动地更新这些位的值，以变换到相应的异常模式。当然，我们也可以手动更改这几个位的值，通过程序实现模式的主动切换。采用的方法同样是“读——修改——写”的经典执行逻辑。

CPSR 保证了程序的正确执行，在一个程序正在执行时，无故改变 CPSR 是非常危险的。但当某个异常发生时，CPU 必须中止一段程序的执行，跳转到别处执行另一段程序。如此一来，当异常处理结束，程序跳转回原处时，CPSR 的值很有可能就被破坏了。

想要解决这个问题就必须依靠各异常模式的 SPSR 寄存器。当异常发生时，硬件首先自动地将上一个状态的 CPSR 寄存器值保存到异常模式下的 SPSR 寄存器中。而当程序从异常模式返回时，再将 SPSR 寄存器中的值写进 CPSR 寄存器。因此，每一种异常模式下都会有一个私有的 SPSR 寄存器，就像在图 5-1 中所看到的那样。

以上就是 ARM 体系结构中有关寄存器、异常以及二者之间的关系的描述。在充分掌握了这些 ARM 体系结构的基本知识后，我们就可以运用这些知识来学习某一异常发生时 CPU 会做的工作。

5.1.1.3 异常发生时的处理器动作

当一种异常发生时，硬件就会自动执行如下动作：

- (1) 将 CPSR 保存到相应异常模式下的 SPSR 中。
- (2) 把 PC 寄存器保存到相应异常模式下的 LR 中。
- (3) 将 CPSR 设置成相应的异常模式。
- (4) 设置 PC 寄存器的值为相应处理程序的入口地址。

以上 4 个步骤完成后，接下来程序就会进入软件控制部分。剩下的工作就全权交给程序员了。图 5-3 是对上述 4 个步骤的总结。

正如前面描述的，当某一异常发生时，硬件自动完成图 5-3 中所示的 4 个步骤的工作，软件会从相应异常的入口地址处运行第一条指令。然而，按照我们前面的描述，PC 寄存器的值不是应该指向正在运行的指令吗？为什



么图 5-3 中却指向了正在提取的指令，这里的“正在提取”又有什么含义呢？

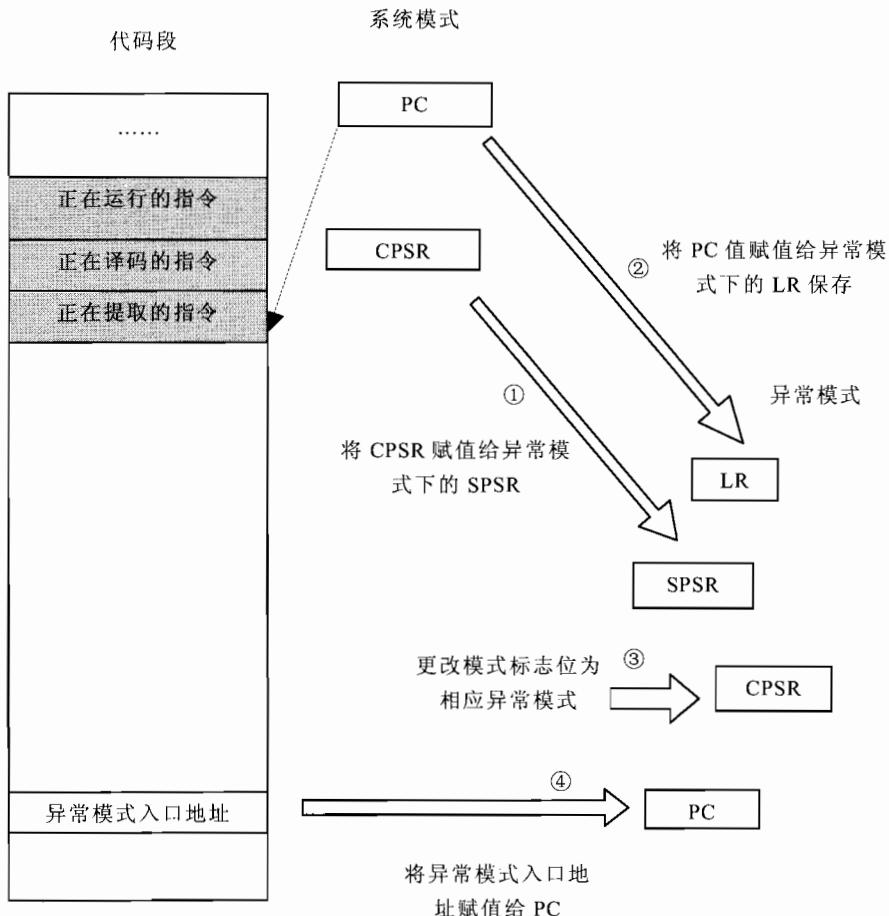


图 5-3 异常发生时，CPU 的动作

这里有一个问题需要说明，因为多级流水线的问题，程序计数器 PC 并不指向当前正在执行的指令，因此读者朋友们可能不能完全理解图 5-3。

要搞清楚这个问题，我们首先需要弄清楚的是一条指令是怎样被执行的。

5.1.1.4 指令执行过程

图 5-4 描述了一条指令的执行过程。

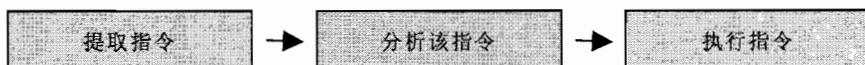


图 5-4 指令的基本执行过程

简单来说，执行某条指令至少要通过取指、译码、执行三个步骤。就好像盲人在吃饭，第一步是用筷子夹出要吃的东西（从内存中取出指令），第二步是把吃的东西举到鼻子底下闻一下看看是否能吃（分析该指令），第三步是放到嘴里吃（执行指令）。

我们假设该盲人又只有一只手，而每一个步骤又要一秒钟的时间，那么这位盲人至少要三秒钟才能吃到一样东西，很显然这种吃饭的方法效率太低。所以，如果 CPU 也采取同样的方法，像图 5-4 那样去执行一条指令，那就意味着 CPU 要消耗掉 3 个指令周期才能完成一个动作，可见其运行效率的低下。

为了弥补这个问题，ARM 采用了一种多级流水线的指令执行方式。例如，在 ARM9 中就采用了三级流水线的处理方法，过程如图 5-5 所示。

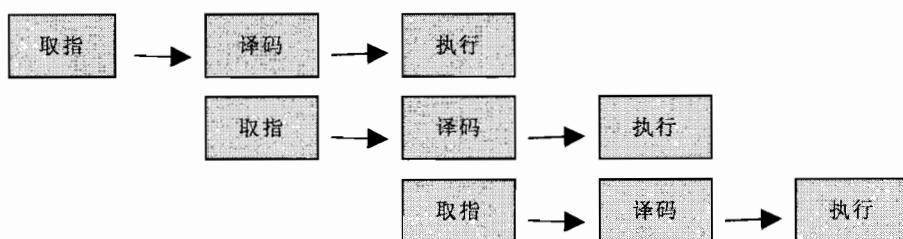


图 5-5 ARM9 中的三级流水线

就像图 5-5 那样，CPU 采用流水线作业的方式，在大多数情况下，是利用三个时钟周期的时间去执行三条指令，从而大大提高了代码运行的效率。

这就好像有一位乐于助人的科学家，知道了盲人吃饭的故事之后，给这位盲人制作了两只机械手，现在盲人已经有三只手了，那么他会怎样吃饭呢？当他的第一只手把吃的送到嘴里吃的时候（执行指令），第二只手已经将另外的食物凑到鼻子底下闻了（分析指令），而第三只手此时正在从盘子里夹第三样东西呢。从此，盲人吃饭的效率就提高了三倍。

现在读者已经理解三级流水的工作原理了，剩下的事情就不言自明了。

程序计数器 PC 的值指向的永远都是待取出的那条指令，也就是说，PC 寄存器的值应是当前正在执行的指令地址加上 8 字节的偏移。到这里，图

5-3 当中的这点疑问也被完全解开了。

此时，如果读者朋友们追问，既然异常模式发生时，最终会转到异常模式入口地址处去运行那里的代码，那么这一入口地址究竟在哪里呢？别着急，这恰恰是我们要向大家介绍的有关异常处理的最后一个知识点——异常向量表。

5.1.1.5 异常向量表

ARM 一共有 5 种异常模式，按道理，每一种异常模式都应该有一个唯一的人口地址。这些入口地址彼此相邻，我们一般称之为异常向量表。

通常情况下，异常向量表是从物理地址 0x0 处开始的，如图 5-6 所示。但这并不是绝对的，我们也可以通过配置 CP15 协处理器，将异常向量表映射到地址的最末尾，即 0xfffff000 处。

内存	地址	模式
复位	0x0	管理模式
未定义指令	0x4	未定义模式
软件中断	0x8	管理模式
预取指中止	0xc	中止模式
数据中止	0x10	中止模式
保留	0x14	
中断	0x18	中断模式
快速中断	0x1c	快速中断模式

图 5-6 ARM 异常向量表



一个典型的实例是 Linux，在 ARM+Linux 的综合系统中，中断向量表就位于这个位置上。Linux 这么做当然是有道理的，就像我们自己的操作系统那样，Linux 也需要建立虚拟地址到物理地址的映射。但是，Linux 还有另外一条规定，那就是将虚拟地址 0x0~0xbfffffff 的 3G 大小的空间划归为用户空间，而将 0xc0000000~0xffffffff 的 1G 大小的空间定义为内核空间。显然，异常向量表作为系统的一个重要结构，理应归属于内核空间。可是，如果异常向量表按照默认的位置出现在了 0x0 处，就变成了用户空间的资源了。因此，Linux 才需要将中断向量表映射到地址末尾。

异常向量表就是一段出现在固定位置的内存空间，当某一异常发生时，程序最终会到达相应的异常入口去执行存放在那里的指令。但请注意，异常向量表中的每一个入口地址只有 4 个字节大小的空间，因此只能存放一条 ARM 指令。

很显然，这一条指令必须是能够实现跳转功能的指令。实现程序跳转有很多种方法，在启动一章我们也有过介绍。

这样，ARM 与异常向量表有关的代码通常会与下面的代码类似。

代码 5-1

```
_start:
ldr pc, _vector_reset
ldr pc, _vector_undefined
ldr pc, _vector_sw
ldr pc, _vector_prefetch_abort
ldr pc, _vector_data_abort
ldr pc, _vector_reserved
ldr pc, _vector_irq
ldr pc, _vector_fiq

.align 4

_vector_reset: .word _vector_reset
_vector_undefined: .word _vector_undefined
_vector_sw: .word _vector_sw
_vector_prefetch_abort: .word _vector_prefetch_abort
_vector_data_abort: .word _vector_data_abort
_vector_reserved: .word _vector_reserved
_vector_irq: .word _vector_irq
_vector_fiq: .word _vector_fiq
```

这个时候，我们就要利用之前学到的链接脚本将这段代码链接到内存的 0x0 位置处。这样算来，从_start 开始的 8 条跳转指令刚好可以落在图 5-6 描述的异常向量表正确的位置上。因此，这些跳转指令就是异常向量表的真正内容了。

那么，在代码 5-1 中，在类似于 __vector_irq 等这样的变量地址处，存放的正是对相应异常具体的处理方法，就像下面这段代码一样。

代码 5-2

```
__vector_irq:
sub r14,r14,#4
stmfd r13!,{r0-r3,r14}
...
ldmfd r13!,{r0-r3,pc}^
```

stmfd 和 ldmfd 这两条指令，大家应该并不觉得陌生。代码 5-2 首先将 R0、R1、R2、R3 和 R14 这 5 个寄存器的值保存到堆栈中。中断处理的工作完成后，再将各自寄存器的值恢复。有意思的是，当寄存器 R14 从堆栈中恢复的时候，程序并不是直接将堆栈中的数据加载到 R14 中，而是将这个值直接复制给 PC 寄存器。因为 R14 存放的恰是上一段程序的返回值，因此，代码 5-2 的最后一条指令将会在寄存器 R0~R3 的内容被恢复后立刻返回。当然，代码 5-2 作为一段简单的中断处理程序，R14 保存的应该是被中断的那条指令的下一条指令，这也是为什么我们一定要将 R14 的值减去 4 的原因。

中断发生时的 PC 寄存器如图 5-7 所示。

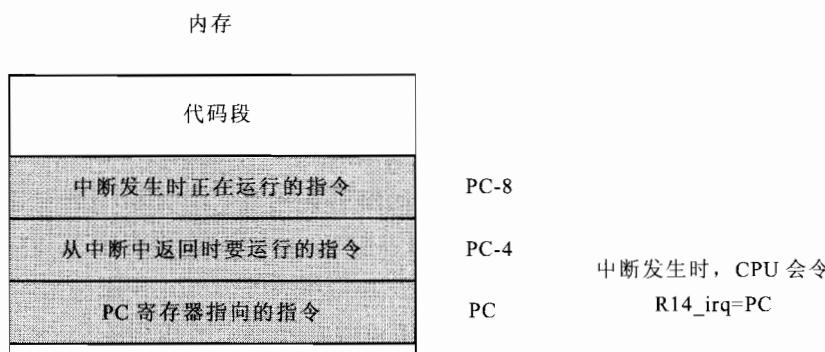


图 5-7 中断发生时的 PC 寄存器

如图 5-7 的描述，中断发生时，CPU 会自动将 PC 寄存器的值保存到中断模式下的私有寄存器 R14 中。中断一旦被执行完，就需要跳转回被中断指令的下一条指令处去执行，这条指令的地址正是 PC-4。因此，我们需要将中断模式下 R14 寄存器的值调整为 R14-4，才是中断结束后真正的程序返回值。

同样的道理也适用于其他异常情况，表 5-2 列出了各种异常模式返回时的调整值。

表 5-2 异常模式

异常	地址	说明
芯片复位	无	未定义芯片复位时的 R14
未定义指令	R14	指向未定义指令的下一条指令
软中断	R14	指向 SWI 指令的下一条指令
预取址中止	R14-4	指向导致预取址中止异常的指令
数据中止	R14-8	指向导致数据中止的那条指令
中断	R14-4	指向被中断的指令的下一条指令
快速中断	R14-4	指向被中断的指令的下一条指令

现在，还剩下一个细节需要说明。在中断模式下，将 R14 中的值传递给 R15，就跳转回了被中断之前的模式，但此时 CPSR 寄存器中的相应位也需要更改过来，在代码 5-2 中，`stmfd` 这条指令后边的“^”符号恰恰完成了这个工作。使用这个符号可以在从堆栈中恢复寄存器的同时，将相应模式下的 SPSR 寄存器中的值恢复到 CPSR 中。

此时，ARM 体系结构中的统一的中断处理过程，我们就介绍完了。

5.1.2 独立的中断处理

正像我们前面提到的那样，将 ARM 中断处理过程分为统一的处理过程和独立的处理过程，有助于帮助我们更好地掌握 ARM 下的中断处理方法。虽然在前面的代码中，我们实现了中断的跳转和返回。但此时仍然不能使用中断来解决具体的问题。这一节，我们会以 2410 为例来讲讲与具体硬件相关的中断处理过程。

图 5-8 描述了 s3c2410 的中断产生过程。

要知道，中断产生于外设硬件，也就是说，外设由于某种原因产生了一

一个由高电平到低电平或者是由低电平到高电平, 又或者是持续的高电平或低电平的信号, 而这些信号就会产生一个中断请求。

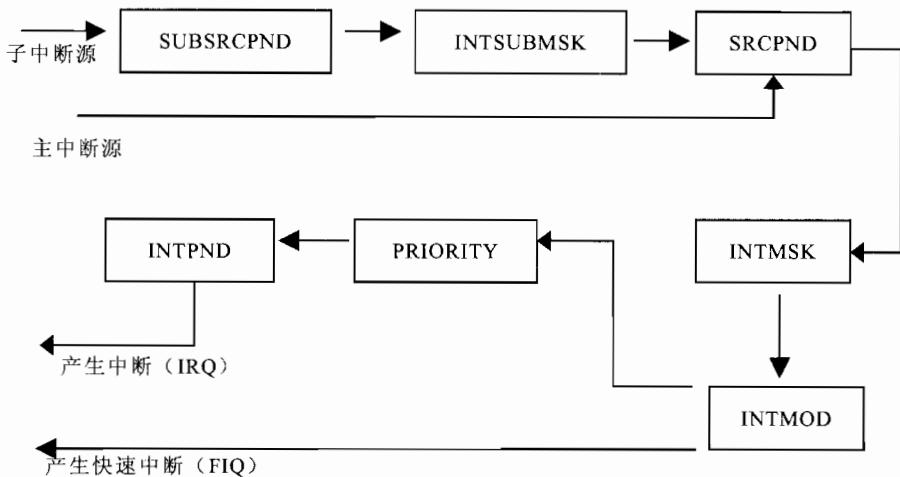


图 5-8 s3c2410 的中断产生过程

当然, 请求终归是请求, 请求是否真的能被处理还要取决两个因素, 一是请求是否被正常传递, 二是请求是否被正常处理。

从另一个角度来看, 中断本身也可以被分成两种——主中断和子中断。所谓主中断, 是指占用一条中断信号线的中断, 如 s3c2410 中的时钟中断, 该中断独立地占用一条 CPU 中断信号线, 即只要 CPU 检测到该中断线有信号, 就必然说明时钟产生了一个中断请求, 它们之间的关系是一对一的。有的时候, 中断和中断之间共同使用同一条信号线, 如 s3c2410 中的串口控制器, 当串口发送缓冲为零、串口接收缓冲有数据、串口收发错误时, 都会在同一条中断信号线上产生请求, 即 CPU 只能检测到与串口对应的这条中断信号线上有中断请求, 而无法直接就判断出这一中断请求到底是因为哪个原因才产生的。这样, 我们就可以将能够产生串口中断的每一种原因叫做子中断了。

说清楚了这两个问题, 我们再回头来看一下图 5-8。

在 s3c2410 中, 中断都是由中断控制器所控制的, 当某一个主中断产生的时候, 其结果会保存到中断控制器的 SRCPND 寄存器中。

例如, 4 号定时器中断 (TIMER4) 发生时, SRCPND 寄存器中的第 14

位就会被置 1。从另一个角度讲，CPU 通过判断 SRCPND 寄存器中哪一位被置 1 了就能够知道哪个硬件产生了中断。如果我们有办法人为地将 SRCPND 某一位置 1（当然，这不可能实现），就可以欺骗 CPU，伪装成某个硬件产生中断了。

SRCPND 并不是排他的，也就是说这个寄存器允许同时有多位被置 1，这代表着可以同时有多个硬件产生了中断。

当一个或多个中断同时产生时，中断控制器会将 SRCPND 的内容送去做中断屏蔽检测。对中断屏蔽的控制由 s3c2410 中断控制器中的 INTMSK 寄存器完成。当 INTMSK 寄存器中的某一位为 0 时，这一位代表的硬件所产生的中断就会被传递到下一级，若该位的值是 1，那么该硬件产生的中断就会被屏蔽掉，中断信号就此消失了。

中断屏蔽寄存器 INTMSK 可以筛选掉一些硬件产生的中断，而那些通过了屏蔽寄存器的中断请求将会被送到 INTMOD 寄存器中做模式判断。

INTMOD 寄存器中每一位的默认值都是 0，这表示系统默认会产生 IRQ 中断请求，如果 INTMOD 寄存器中的某一位为 1，则最终会产生 FIQ 中断。FIQ 中断作为一种快速的中断处理，必须要保证它的唯一性。因为如果所有的中断都是快速中断的话，那就相当于没有快速中断了。因此，INTMOD 寄存器在同一时刻有且只能有一位置 1。这就保证了在所有能够产生中断请求的硬件中只允许一个快速中断请求。

当某个中断请求经过 INTMOD 寄存器被确定为快速中断请求时，CPU 将立刻产生一个快速中断，体系结构统一的中断处理过程就会发生。如果经过 INTMOD 寄存器确认为 FIQ 中断请求，则该请求会被送往下一级，做中断优先级的检测，这个过程是由 PRIORITY 寄存器负责的。

很多时候，都会有多个中断请求同时到达 PRIORITY 寄存器。但是，各种各样的硬件中断请求的优先级却各不相同，CPU 会在此阶段选择一个优先级最高的中断请求送往下一级。当然，各种中断请求的优先级是可以根据软件的需求动态调配的，我们可以通过改写 PRIORITY 寄存器的值进行配置。中断请求被送往的下一级是 INTPND 寄存器。

INTPND 寄存器中的每一位代表一个已经通过优先级仲裁的中断请求。因为同一时刻只能有唯一一个中断请求通过 PRIORITY 寄存器，所以，INTPND 寄存器只会有一个位被置 1，我们可以读出这个位来判断究竟是哪

一个硬件产生的中断请求。与此同时，CPU 会立刻产生一个中断异常，体系结构统一的中断处理过程就会发生。

至此，从一个硬件产生中断请求到 CPU 触发一个中断异常的全部过程，我们就已经描述清楚了。对于子中断的处理与主中断类似，只是多了两个步骤而已。

在这个过程中，从头到尾都是硬件在做工作。那么从软件的角度来讲，我们又应该做些什么才能保证整个中断处理过程顺利完成呢？

我们知道，当 CPU 接收硬件中断请求并抛出中断异常后，程序会跳转到中断向量表中对应的地址继续运行，最终也会跳转到某段代码处，针对具体的硬件去完成它的工作。在这个过程中，软件至少要完成如下两步才能最基本地保证中断处理的正确性。

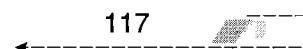
(1) 我们必须要知道的是，究竟哪个硬件产生了中断请求。一个快速的方法是读取 INTOFFSET 寄存器。该寄存器中的值是位于 0~31 的一个整数，每一个数字代表一个硬件，如 28 代表零号串口。而这个数字与 INTMSK、SRCPND 等寄存器中代表硬件的位序号又是一致的。也就是说，INTPND 寄存器中的第 28 位正是代表了零号串口，该位为 1 则表示这个串口产生了中断。

(2) 必须清除掉 SRCPND 和 INTPND 这两个寄存器中代表某一中断请求的相应位。为什么要这样做呢？如果不将相应的中断位置 0，SRCPND 和 INTPND 等寄存器中代表中断请求的对应位将永远是 1，这样，同一个中断请求就会反复产生，而实际上硬件仅仅请求了一次中断，这样一来，错误就产生了。而清除 SRCPND 和 INTPND 两个寄存器的某一位的方法其实很简单，只需要向该位写 1 即可。

在程序中实现以上两点，是保证一个中断被正确处理的关键。表 5-3 列出了 s3c2410 中断控制器的相应物理地址。表 5-4 描述了 s3c2410 中诸如 INTPND 这样的寄存器的各个位所代表的中断源。这些都是在写代码之前需要了解的。

表 5-3 s3c2410 中断控制器

寄存器	地址	默认值
SRCPND	0x4A000000	0x00000000
INTMOD	0x4A000004	0x00000000
INTMSK	0x4A000008	0xFFFFFFFF



续表

寄存器	地址	默认值
PRIORITY	0x4A00000C	0x7F
INTPND	0x4A000010	0x00000000
INTOFFSET	0x4A000014	0x00000000
SUBSRCPND	0x4A000018	0x00000000
INTSUBMSK	0x4A00001C	0x7FF

表 5-4 偏移量代表的中断源

中断源	偏移值	中断源	偏移值
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT	9
Reserved	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	Reserved	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

5.2 简单的中断处理实例

接下来让我们亲自运行一段最简单的中断代码，以巩固我们学到的知识。

5.2.1 解决异常向量表的问题

在真正写代码之前，我们还需要完成一件棘手的事情。

回忆一下前面的内容，我们虚拟出来的硬件平台，具有 8M 的 SDRAM，

被挂接到了 s3c2410 的 0x30000000 这个物理地址。之前运行的程序都是以这一假设为基础的。

但如果要运行中断处理程序，异常向量表的问题就不太容易解决了。我们知道异常向量表必须位于内存地址 0x0 处，但是，在我们的虚拟硬件平台中，地址 0x0 根本就不存在任何存储设备。

在实际的应用中，无论是 NAND FLASH 还是 NOR FLASH 都有办法映射到内存地址 0x0 的位置上。因此，将异常向量放在 FLASH 的起始位置，让它们在异常产生时参与程序跳转是一个非常有效的方法。

另一种方法与之类似，我们可以在程序启动时就将已经准备好的向量表搬运到地址 0x0 处。当然，这同样需要有可写的存储器存在。

以上两种方法实现起来并不困难。我们也可以修改虚拟硬件平台的配置来满足这些条件。

```
cpu: arm920t
mach: s3c2410x

#physical memory
mem_bank: map=M, type=RW, addr=0x00000000, size=0x00100000
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,
file=./le eos.bin,boot=yes
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

这样一来，我们就可以将代码的向量表部分搬运到新的存储器中，或者干脆将整个代码段放置于此处，从而实现异常向量表的初始化。

然而，解决异常向量表的方法并不仅限于以上两种。通过 MMU 进行地址重映射，更加灵活方便，本书中就将会使用这种方法来实现中断处理程序。

为了保证异常向量表出现在内存 0x0 的位置上，就必须要修改 MMU 相关的代码，将内存中的中断向量表映射到地址 0x0 处。于是我们需要修改原有代码中的“mmu.c”这个文件，在#define IO_MAP_SIZE 下边一行添加如下内容。

代码 5-3

```
#define VIRTUAL_VECTOR_ADDR      0x0
#define PHYSICAL_VECTOR_ADDR      0x30000000
```

完成相关宏定义后，就需要修改 init_sys_mmu 函数了。请朋友们在该

函数的两个 for 循环之前添加如下一段代码。

代码 5-4

```
for(j=0;j<MEM_MAP_SIZE>>20;j++) {  
    pte=gen_l1_pte(PHYSICAL_VECTOR_ADDR+(j<<20));  
    pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;  
    pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;  
    pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;  
    pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\  
        VIRTUAL_VECTOR_ADDR+(j<<20));  
    *(volatile unsigned int *)pte_addr=pte;  
}
```

以上两段程序负责将内存地址 0x30000000 处开始的 1M 内存映射到地址 0x0 处。这样一来，当中断产生的时候，硬件跳转到中断入口地址 0x18 处执行，而实际运行的则是 0x30000018 处的代码。异常向量表的问题就成功地解决了。

5.2.2 简单的中断处理代码

接下来我们要改写一下启动代码的内容。打开原有代码中的“abnormal.s”文件，原来的代码如下：

代码 5-5

```
_vector_irq:  
    nop
```

将其修改为：

代码 5-6

```
_vector_irq:  
    sub r14,r14,#4  
    stmdfd r13!,{r0-r3,r14}  
    bl common_irq_handler  
    ldmfd r13!,{r0-r3,pc}^
```

在中断模式下，代码最终会跳转到 _vector_irq 处运行。此处，CPU 首先会在修正返回地址的同时，保存相关寄存器的数据，然后跳转到函数 common_irq_handler 继续执行。细心的读者可能已经发现了，代码 5-6 其实就是代码 5-2 的具体化。

接下来我们需要新建一个文件，名为“interrupt.c”，添加如下内容。

代码 5-7

```
#define INT_BASE      (0xca000000)
#define INTMSK       (INT_BASE+0x8)
#define INTOFFSET    (INT_BASE+0x14)
#define INTPND       (INT_BASE+0x10)
#define SRCPND       (INT_BASE+0x0)

void enable_irq(void){
    asm volatile (
        "mrs r4,cpsr\n\t"
        "bic r4,r4,#0x80\n\t"
        "msr cpsr,r4\n\t"
        :::"r4"
    );
}

void umask_int(unsigned int offset){
    *(volatile unsigned int *)INTMSK&=~(1<<offset);
}

void common_irq_handler(void){
    unsigned int tmp=(1<<(*(volatile unsigned int *)INTOFFSET));
    printk(" %d\n",*(volatile unsigned int *)INTOFFSET);
    *(volatile unsigned int *)SRCPND|=tmp;
    *(volatile unsigned int *)INTPND|=tmp;
    printk(" timer interrupt occurred\n");
}
```

代码 5-7 包含了三个函数，其中，common_irq_handler 就是代码 5-6 中被调用的那个函数。

在该函数中，程序首先读出了寄存器 INTOFFSET 的值，然后转换成为序号并保存到临时变量 tmp 中。紧接着，又将其写入寄存器 SRCPND 和 INTPND 中，目的是清除寄存器 SRCPND 和 INTPND 中的相应位，这一点我们在上一节的最后部分有详细的描述。最后，函数 common_irq_handler 又打印了一个字符串，表示中断处理被正确地运行。

umask_int 函数负责将相应中断的屏蔽位清除，该函数是通过写 INTMSK 寄存器实现的。

函数 enable_irq 则通过内联汇编的方式清除 cpsr 中的 I 位，全局地使能了中断。

关于代码 5-7，我们还有一点需要强调，因为程序此时已经使能了 MMU，所以代码中使用的寄存器地址就不再是芯片手册中的地址了，而应该是经过映射后的虚拟地址，其基值定义成了宏 INT_BASE。

最后，想要成功编译“interrupt.c”文件，我们还需要将 Makefile 中 OJBS 一行修改为如下形式。

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o interrupt.o
```

到这里，中断处理部分的程序就正式完成了。

下面我们需要一种方法，能够在我们的虚拟设备中产生一些有效的中断，比如每按一次键盘，就会有一个中断在虚拟平台中产生，只有这样，我们才能够看到中断处理程序的运行效果。

当然，键盘中断是一个很好的例子。但最终我们还是选择了定时器中断。究其原因，不仅仅是因为它的配置和使用都相对简单，还因为这个内部硬件通常是作为操作系统时钟周期来驱动进程、定时器等系统核心模块，保证操作系统的正确运行的。

接下来我们就来说一说 s3c2410 中的定时器。

5.2.3 S3C2410 中的定时器

s3c2410 一共集成了五个定时器，TIMER0~TIMER3 四个定时器都有外部引脚的输出，专门为外设提供同步时钟，而第五个定时器 TIMER4 则是一个内部定时器。这五个定时器的工作原理都是一致的，如图 5-9 所示。

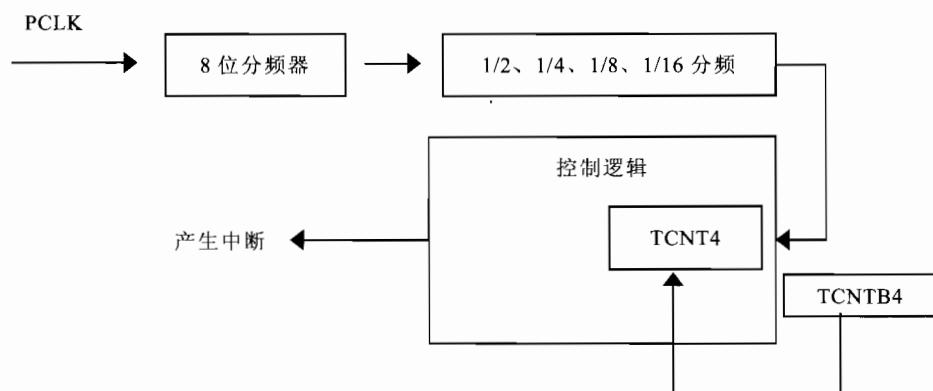


图 5-9 s3c2410 定时器工作原理

首先，定时器所需要的基础时钟由系统的 PCLK 提供。PCLK 是一个由 CPU 核产生的专门供给低速外围设备的基本时钟。通常情况下，程序会选择将 PCLK 的典型值配置成 50M。当然别的值也是可以的，这要根据需求而定。除了 PCLK 以外，s3c2410 还有专门提供给 CPU 内部的时钟 FCLK，提供给高速外设的基本时钟 HCLK 以及专供 USB 设备使用的 UCLK 时钟。FCLK 可以由内部锁相环将外接晶振时钟倍频得到，而 HCLK 和 PCLK 又可以由 FCLK 分频得到。s3c2410 外接晶振的典型频率是 12M，正常工作时，FCLK 的时钟频率往往是 200M 左右，而 FCLK 与 HCLK 和 PCLK 之比一般设置成 1 : 2 : 4。所以，PCLK 就接近 50M 了，以上过程都可以通过将相应的值写入相关寄存器来实现。

虽然 PCLK 负责驱动慢速外设，但 50M 左右的时钟频率仍然很高。于是，不同的慢速硬件需要根据实际情况对 PCLK 进行再分频，定时器也是如此。s3c2410 的 TIMER 控制器首先将 PCLK 预分频，这是通过将 PCLK 的时钟频率值除以一个 0~255 范围内的某一个整数实现的。

然后，控制器紧接着又对这个结果再次分频，这次分频是将上一步骤的结果除以 2、4、8 或 16 中的某一个数。除得的结果将分别送给五个定时器供它们使用。当然，在这一步中，我们也可以选择使用外接的时钟作为五个定时器的输入时钟，但这种用法并不多见。当二次分频完成时，其结果会送到各定时器的控制逻辑，控制逻辑主要负责计数和比较的工作。

当然，不同定时器的控制逻辑，结构会稍有不同。图 5-9 表示的是 TIMER4 的控制逻辑结构图，也是最简单的一个。在 TIMER4 的控制逻辑内部有一个寄存器 TCNT4。TCNT4 寄存器其实是一个减数器，每当上一个逻辑部件传来一个时钟脉冲时，TCNT4 中的数值就减 1。当减到 0 时，TIMER4 就会产生一个中断请求。然而，TCNT4 这个寄存器是不可见的，也就是说，我们不能直接读写这个寄存器的内容。想要改写 TCNT4 寄存器的初始值，必须通过改写外部寄存器 TCNTB4 来实现，当我们把某一个数写入到 TCNTB4 寄存器时，该值并不会立刻被复制到内部寄存器 TCNT4 中，而是要等到 TCNT4 中的值减到零时才会复制。

以上就是 TIMER4 定时器的硬件执行流程。从软件的角度来讲，我们也只需要依据上述过程对相关的寄存器进行配置，就可以完成对定时器的操作。表 5-5 列出了与 TIMER4 有关的寄存器及其使用方法。

表 5-5 s3c2410 中与定时器有关的寄存器

寄存器	地址	说明
TCFG0	0x51000000	15:8 位代表 TIMER4 一级分频器的值
TCFG1	0x51000004	23:20 位可以选择使用 DMA 方式还是中断方式， 19:16 位代表二级分频器的值
TCON	0x51000008	22:20 位用来配置 TIMER4
TCNTB4	0x5100003C	代表 TIMER4 减数寄存器的数值

我们将具体的步骤总结如下。

(1) 根据 PCLK 的值确定一级分频器和二级分频器的具体值，改写 TCFG0 和 TCFG1 两个寄存器。TCFG0 只需要向 15:8 写一个 8 位数即可，TCFG1 的默认值是中断方式工作、1/2 分频，这个可以改写，也可以使用默认值。

(2) 根据需要配置 TCNTB4，调整减数器的值。

(3) 配置 TCON 寄存器。以 TIMER4 为例，TCON 寄存器的第 22 位代表用来配置是否使用 TIMER4 的 autoreload 功能。所谓 autoreload，是指控制逻辑中的内部寄存器 TCNT4 在减数到零时，是否自动地从 TCNTB4 寄存器中将新的值自动地复制到 TCNT4 里。想要定时器持续有效，必须开启 autoreload 功能，也就是将 TCON 的第 22 位置 1。将 TCON 的第 21 位置 1，会使 TCNTB4 中的内容加载到 TCNT4 中，在首次使用 TIMER4 时，这一步是需要的。在 autoreload 模式下，如果已经将 TCNTB4 中的内容复制到 TCNT4 了，就可以把 TCON 的第 21 位清零了。TCON 的第 20 位用来控制 TIMER4 的开始和停止，可以通过将该位置 1 来启动 TIMER4。

现在，所有关于定时器的原理和操作方法我们就介绍完了。

5.2.4 让中断处理程序运行起来

事不宜迟，让我们来实现一下这段代码吧！

代码 5-8

```
#define TIMER_BASE (0xd1000000)
#define TCFG0 ((volatile unsigned int *) (TIMER_BASE+0x0))
#define TCFG1 ((volatile unsigned int *) (TIMER_BASE+0x4))
#define TCON ((volatile unsigned int *) (TIMER_BASE+0x8))
#define TCONB4 ((volatile unsigned int *) (TIMER_BASE+0x3C))
```



```

void timer_init(void) {
    *TCFG0 |= 0x800;
    *TCON &= (~ (7 << 20));
    *TCON |= (1 << 22);
    *TCON |= (1 << 21);
    *TCONB4 = 10000;
    *TCON |= (1 << 20);
    *TCON &= ~ (1 << 21);

    umask_int(14);
    enable_irq();
}

```

函数 timer_init 首先对与 TIMER4 相关的寄存器进行了适当的配置。然后，利用 umask_int 函数清除掉中断屏蔽寄存器中代表 TIMER4 的那一位。最后，使用 enable_irq 函数使能全局中断。整个过程完全是按照前面描述的方法依次进行的。

同样的，这里与 TIMER4 寄存器有关的地址使用的仍然是虚拟地址。因此，寄存器地址偏移值应该是 0xd1000000，而不是表 5-5 中描述的 0x51000000。

我们可以将代码 5-8 的内容保存到“boot.c”中的 plat_boot 函数前面，然后修改 plat_boot 函数为如下形式：

代码 5-9

```

void plat_boot(void) {
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printf();
    timer_init();
    while(1);
}

```

所有代码都修改完毕了，此时可以在终端运行 make 命令，如果成功编译，运行 skyeye 命令，你将看到如下输出：

```
helloworld
test_mmu
testing printf
test string :::: this is %s test
test char :::: H
test digit :::: -256
test X :::: 0xffffffff00
test unsigned :::: 4294967040
test zero :::: 0
14 timer interrupt occurred
....
```

终端会不停地输出字符串“timer interrupt occurred”，表示 TIMER4 中断被定时触发。至此，在我们自己的操作系统中也能实现基本的中断了。

5.3 复杂的中断处理实例

在前面的描述中，为了突出重点，我们忽略了很多细节。在本节中，我们将进一步优化这段程序代码，这需要先从分析被我们忽略的那些细节开始了解这些细节所带来的问题，然后再想办法解决它们。

5.3.1 提高中断的效率

首先有一点必须要强调，能够产生中断请求的硬件有很多，这些硬件什么时候产生中断请求，我们并不清楚。也就是说，当我们正在处理某个硬件产生的中断请求时，另外的硬件可能也产生了中断请求。那么当这种情况发生时，使用我们前面介绍的简单的中断处理方法会带来怎样的问题呢？我们先来分析一下。

中断的简单执行过程如图 5-10 所示。

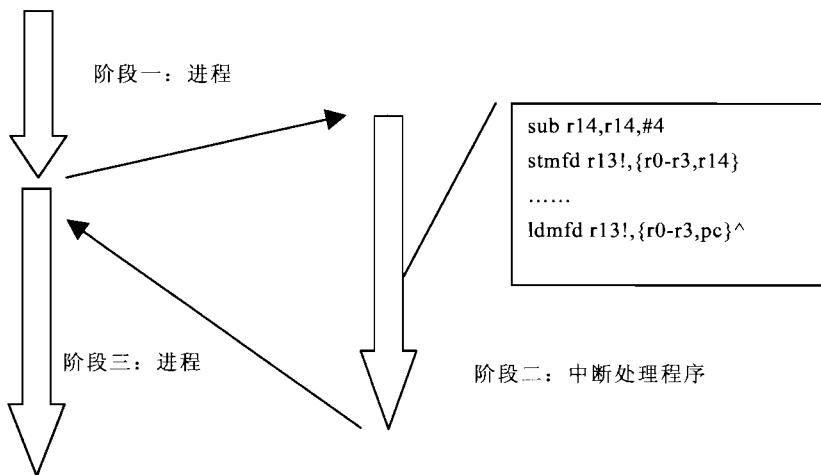


图 5-10 中断的简单执行过程

如图 5-10 所示，可以将中断的简单执行过程划分为三个阶段。某个进程在第一阶段运行时，被一个中断中止，于是系统进入第二阶段，进行中断的处理，处理完成后，再回到进程被中止的位置继续执行，运行第三阶段的代码。

正常情况下，程序运行在第一阶段和第三阶段时，是允许被其他的中断中止的。但当第二阶段发生时，CPU 会自动地将 CPSR 寄存器的 I 位和 F 位置 1。也就是说，在第二阶段运行时，中断默认是被禁止的。此时若有其他硬件产生了新的中断请求，则该中断请求将不会被立刻处理，而只能等到退出第二阶段后再去处理。

这样，问题就来了。如果第二阶段程序运行时间较长，并且同时有很多的硬件产生了中断请求的话，就会出现中断的延迟，导致某些中断等待了过长的时间才被处理，违背了中断处理程序的初衷。在实际的运行环境下，这种情况还是会发生的。

上面我们对中断处理过程中可能遇到的普遍问题进行了描述，这个问题即是中断延迟。所谓中断延迟，指的是从硬件产生中断请求开始到运行中断处理程序中的第一条指令之间的间隔时间。中断处理的基本原则是能够在产生请求时，第一时间对中断进行处理。而中断延迟恰恰延长了这段时间，其结果是使系统响应速度变慢，严重时甚至会引起系统瘫痪。因此，一个以应



用为目的的操作系统必须要想办法最小化中断延迟,来提高系统的运行效率和稳定性。

解决这个问题的方法通常有两个,一是允许中断嵌套,二是引入中断优先级,优先处理高优先级的中断。

使用中断嵌套来处理中断延迟的方法多见于一些通用操作系统之中。中断嵌套的基本思想是,中断请求不分高地贵贱,生来平等,没有谁重要谁不重要之分,只优先处理最新发生的中断请求。因此,通过中断嵌套的方法处理中断延迟问题能够在一定程度上解决中断延迟问题。

图 5-11 是中断嵌套的示意图。

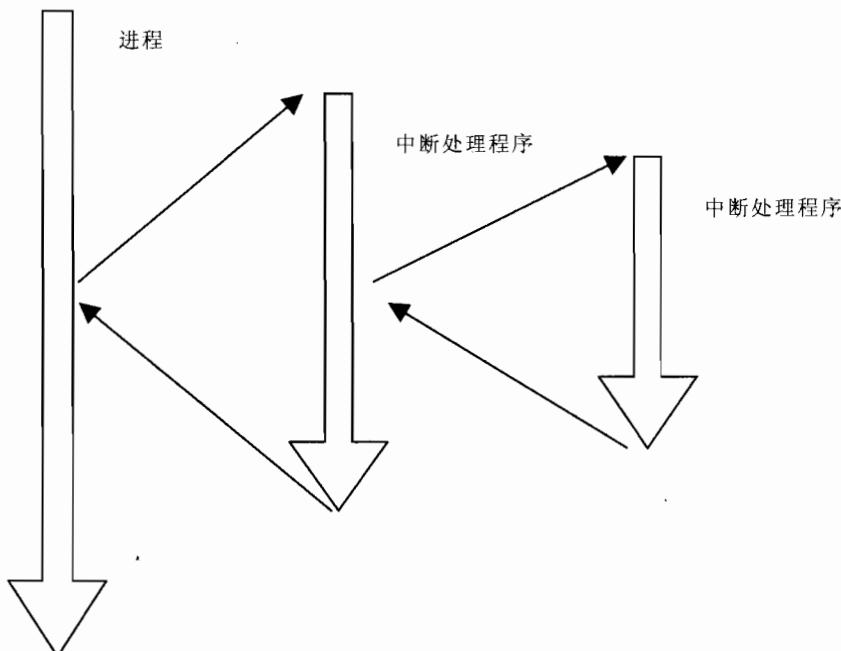


图 5-11 中断嵌套示意图

如图 5-11 所示,当第一个中断处理程序还没有执行完毕的时候,第二个中断请求就已经产生了,于是 CPU 去执行第二个中断处理程序,执行完毕后,又回到第一个中断处理程序中继续执行,这就是一个最简单的中断嵌套模型。

使用这种方法,第二个中断请求的延迟就完全没有了,而第一个中断请求虽然执行的时间被拖长了,但毕竟也执行了一部分,从而使整体的中断运行效率提高。



图 5-11 看似简单，但是在真正执行的时候，却没有那么容易。首先我们要解决的是，怎样在一个中断正在被处理的过程中允许另外一个中断发生。有的读者朋友可能会觉得这很容易，用 msr 和 mrs 清除掉 CPSR 寄存器的 I 标志就可以了。事实当然不会那么简单，读者朋友们可以结合图 5-3 看一下在中断模式下，正在执行中断处理程序时又发生了中断，会有什么严重后果。

(1) 在中断模式下，寄存器 R14_irq 作为私有寄存器，保存着被中止的进程的返回地址。我们将它保存到堆栈当中去，目的是当中断模式发生函数调用时，该寄存器可以用来保存函数返回值。但如果此时另一个中断恰好发生，R14_irq 就又要保存新的中断处理程序返回时的地址，原来的返回值便会丢失。

(2) 同样地，在中断模式下，SPSR_irq 寄存器保存了上一个模式的 CPSR 值，但是如果在中断模式下再次发生中断，那么 SPSR_irq 又要保存新中断的 CPSR，从而将原来的 CPSR 丢掉。

图 5-12 描述的就是以上两个问题。我们可以看到，如果中断模式下又发生了中断，相应的寄存器所保存的值就会被更新，从而造成数据丢失。即便我们可以将 SPSR_irq 寄存器的值保存到堆栈中，避免了该寄存器被篡改，也无法通过类似手段解决寄存器 R14_irq 的问题。

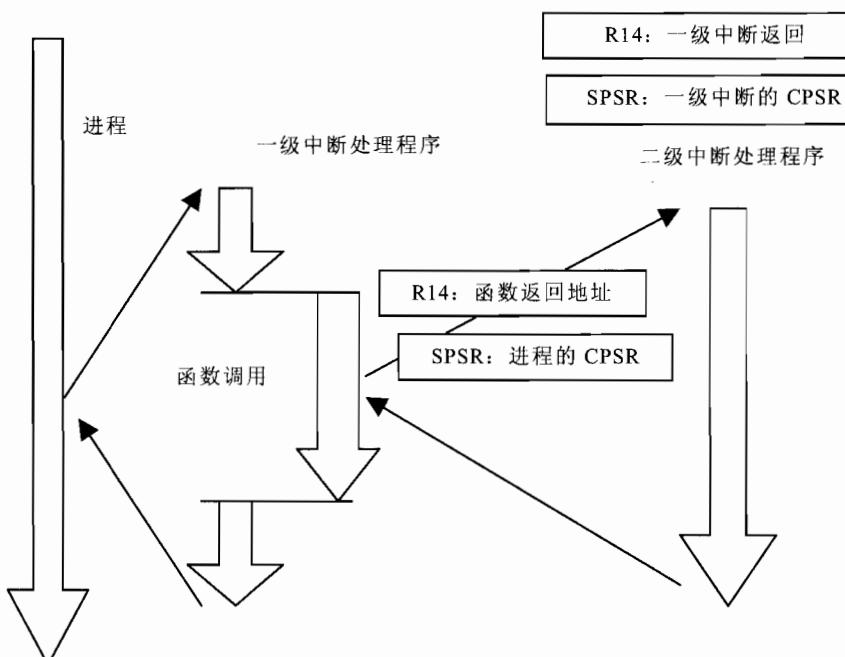


图 5-12 中断嵌套时，寄存器的问题

当然，难解决并不代表不能解决，只不过我们需要换个思路，不在中断模式下，而是切换到其他模式进行中断处理。

当中断处理程序发生在其他模式（如 SVC 模式）时，函数调用的返回值会保存在寄存器 R14_svc 中，即使此时中断再次发生，CPU 也会自动进入中断模式，将被中断的程序的返回值保存在 R14_irq 中，二者不再冲突。这就是实现中断嵌套的普遍方法。

5.3.2 中断嵌套的实现

让我们再一次通过例子来深入理解一下中断嵌套。我们需要将文件“abnormal.s”中的内容修改一下，如下所示：

代码 5-10

```
.equ DISABLE_IRQ,0x80
.equ DISABLE_FIQ,0x40
.equ SYS_MOD,0x1f
.equ IRQ_MOD,0x12
.equ FIQ_MOD,0x11
.equ SVC_MOD,0x13
.equ ABT_MOD,0x17
.equ UND_MOD,0x1b
.equ MOD_MASK,0x1f

.macro CHANGE_TO_SVC
    msr cpsr_c, #(DISABLE_FIQ|DISABLE_IRQ|SVC_MOD)
.endm

.macro CHANGE_TO_IRQ
    msr cpsr_c, #(DISABLE_FIQ|DISABLE_IRQ|IRQ_MOD)
.endm

.global __vector_undefined
.global __vector_swsi
.global __vector_prefetch_abort
.global __vector_data_abort
.global __vector_reserved
.global __vector_irq
.global __vector_fiq
```

```

.text
.code 32

__vector_undefined:
    nop
__vector_swi:
    nop
__vector_prefetch_abort:
    nop
__vector_data_abort:
    nop
__vector_reserved:
    nop
__vector_irq:
    sub r14,r14,#4
    stmfd r13!,{r14}
    mrs r14,spsr
    stmfd r13!,{r14}
    CHANGE_TO_SVC
    stmfd r13!,{r0-r3}
    bl common_irq_handler
    ldmfd r13!,{r0-r3}
    CHANGE_TO_IRQ
    ldmfd r13!,{r14}
    msr spsr,r14
    ldmfd r13!,{pc}^

__vector_fiq:
    nop

```

代码 5-10 是在原有内容的基础上修改了函数 __vector_irq 的部分而得到的。将原来的简单中断处理方式修改成现在的嵌套中断处理方式，其原理正是我们前面所介绍的更换处理模式的方法。

首先，我们需要正确地计算程序上一个状态的返回值并及时地将该返回值压入堆栈。这是通过 sub r14,r14,#4 和 stmfd r13!,{r14} 这两条指令实现的。

然后，mrs R14,spsr 这条指令负责将 SPSR_irq 寄存器的值保存到 R14 中，因为 R14 寄存器的值已经被压入堆栈了，因此这里利用它作为一个普通寄存器来使用是安全的。此时，R14_irq 保存的就是中断状态下的 SPSR 寄存器的值，同时也是上一个被中断的状态的 CPSR 寄存器的值，此时也将

该值压入中断模式堆栈中。这样，在保护好上一个状态的 CPSR 和返回值后，即使中断再次发生，这两个值也不会遭到破坏，从而保证了程序执行的正确性。

在相关值保存完毕后，我们就可以跳转到 SVC 模式了。这么做的目的是为了保证在函数调用时，返回值可以保存在 R14_svc 中，而不是原先的 R14_irq 中。这样，一旦中断恰在此时发生，R14_irq 就可以用来保存上一状态的返回值，避免函数空间被破坏。

此处，我们是利用宏定义的方式实现模式间的切换的。关于汇编模式下的宏定义方法，下面介绍一下。

GNU 汇编下的宏定义的基本格式如下：

代码 5-11

```
.macro THE_NAME_OF_YOUR_MACRO (PARAMETER)
    assemble code
.endm
```

如代码 5-11 所示，宏是以 .macro 关键字开始的。THE_NAME_OF_YOUR_MACRO 代表了宏定义名字，如果宏有参数，参数要跟在宏名的后边。紧接着若干行是实际的汇编代码。最后一行以 .endm 关键字结尾，代表宏定义到此结束。

代码 5-12

```
.macro CHANGE_TO_SVC
    msr cpsr_c, #(DISABLE_FIQ|DISABLE_IRQ|SVC_MOD)
.endm
```

代码 5-12 就是一个宏定义的最简单例子。在该例子中，CHANGE_TO_SVC 就是宏定义的名字，msr 指令一行就是该宏定义的具体内容。这个宏非常简单，只有一行汇编代码。

如果想在宏定义中使用参数，可以参考代码 5-13。

代码 5-13

```
.macro enable_fiq      reg
    mrs \reg,      cpsr
    bic \reg,      \reg, #DISABLE_FIQ
    msr cpsr,     \reg
.endm
```

在代码 5-13 中，reg 是宏 enable_fiq 的参数，而在汇编代码中，对该参

数的引用是通过将该参数名字前面加上一个“\”符号来实现的。这样，在使用这个宏定义时，就可以通过某个寄存器名来替换 reg 参数，例如：

```
enable_fiq r0
```

这样一来，关于 GNU 汇编下宏定义的使用方法我们就介绍完了。此时再回到代码 5-10 中，`_vector_irq` 函数正是通过宏 `CHANGE_TO_SVC` 实现了从中断模式向管理模式的切换，同时禁止中断和快速中断发生。

之后，寄存器 R0~R3 的值被保存到管理模式下的堆栈之中。在做好了所有准备工作之后，程序调用 `common_irq_handler` 函数进行具体的中断处理。

函数 `common_irq_handler` 与代码 5-7 中的函数也稍有不同，其定义如下：

代码 5-14

```
void common_irq_handler(void) {
    unsigned int tmp=(l<<(*(volatile unsigned int *)INTOFFSET));
    printk(" %d\t", *(volatile unsigned int *)INTOFFSET);
    *(volatile unsigned int *)SRCPND|=tmp;
    *(volatile unsigned int *)INTPND|=tmp;
    enable_irq();
    printk(" interrupt occurred\n");
    disable_irq();
}
```

在代码 5-14 中，由 `printk(" interrupt occurred\n")` 一行所代表的具体硬件的中断处理代码前后各添加了一条新的代码，分别是使能和禁止全局中断。也就是说，在函数 `enable_irq()` 和 `disable_irq()` 之间，虽然与某一硬件有关的中断程序正在处理，但同时也允许其他嵌套中断发生。

在调用完 `common_irq_handler` 函数之后，需要将 R0~R3 四个寄存器的值从堆栈中弹出。之后程序调用 `CHANGE_TO_IRQ` 宏，再次返回到中断模式。在中断模式下弹出堆栈中保存的 SPSR 值之后，返回到中断前的状态中。

这样，一个可嵌套的中断处理过程就正式完成了。

运行这段代码的方法其实也很简单。读者用代码 5-10 和 5-14 的内容替换掉原有的代码，之后编译并运行，结果如下：

```
helloworld
test_mmu
testing printk
```

```
test string :::: this is %s test
test char :::: H
test digit :::: -256
test X :::: 0xfffffff00
test unsigned :::: 4294967040
test zero :::: 0
14 interrupt occurred
....
```

5.4 更优秀的中断嵌套方法

代码 5-10 中使用的中断嵌套方法虽然能够很好地解决中断延时的问题，但仍然有不足之处。其中一点就是中断被使能的时间偏少。

我们看到，在代码 5-10 中，使能中断发生在 common_irq_handler 函数中，清除相应中断标志之后，被使能了的中断很快就又被禁止了，在调用 common_irq_handler 函数之前和调用之后，中断都是被禁止的。

在中断处理过程中，使能中断时间的长短决定了系统实时性的优劣。因此，优秀的操作系统都要求可嵌套中断尽量延长使能中断的时间。本节中我们介绍一种被广泛采用的方法，可以尽可能地延长中断使能时间，保证系统的强实时性。

仔细分析一下代码 5-10，在处理中断请求时，程序首先从中断模式切换到管理模式，在管理模式下使能全局中断，在中断处理完成后，又需要先禁止全局中断，然后再切换回中断模式，最后返回到上一状态中。

结合以上分析以及前面我们对可嵌套中断所产生的问题的描述，读者会发现，中断嵌套的一个原则是不允许在中断模式下使能全局中断。因为这样会导致中断模式下的函数返回值寄存器 R14_irq 有可能被改写。所以，想要延长中断处理过程中使能中断的时间，就意味着要尽量减少中断处理程序在中断模式下的工作时间。

如果我们能够让中断处理程序在管理模式下直接返回到被中断之前的模式，而不是像代码 5-10 那样从中断模式返回，就可以在一定程度上延长中断使能的时间，从而提高系统实时性。

代码 5-10 在中断时的模式切换过程如图 5-13 所示。

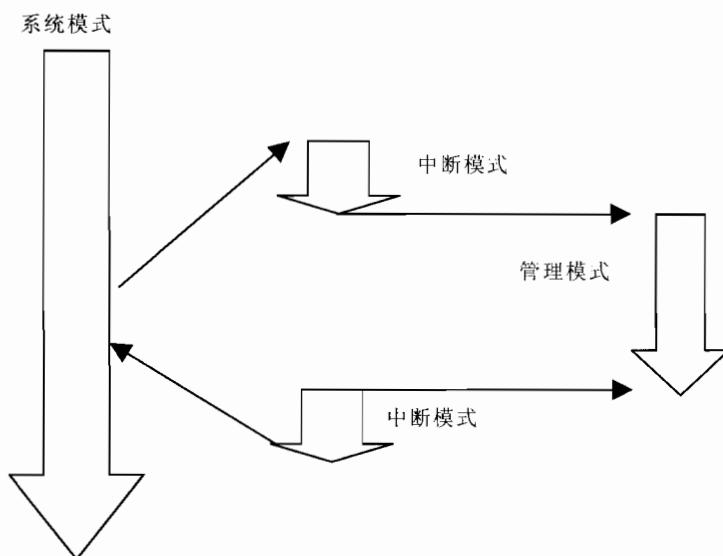


图 5-13 简单的中断嵌套方法

与图 5-13 相比，图 5-14 中的中断模式出现的时间更短。因此，在中断处理过程中，中断会有更多的时间处于使能状态，这样就延长了中断嵌套的时间，从而提高了系统运行效率。

实现该功能的关键一点是将上一个状态的相关寄存器直接保存到管理模式的堆栈中，而不是像代码 5-10 那样在中断模式下保存。处理好这个问题后，其他的就很简单了。

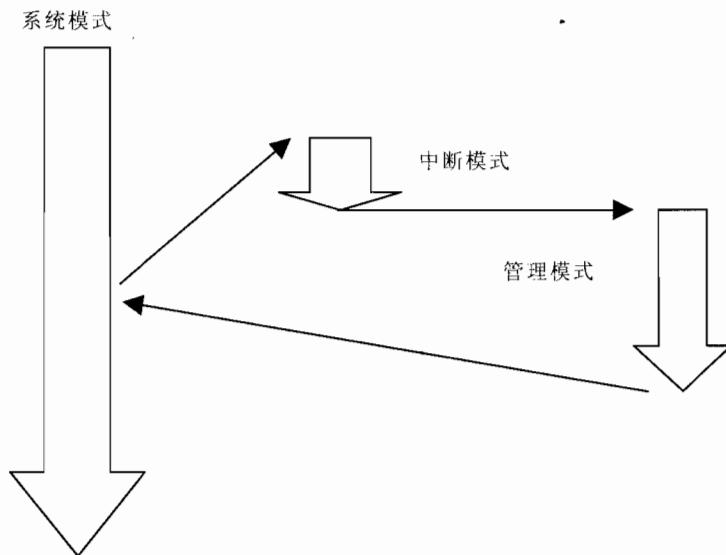


图 5-14 快速实现中断嵌套的方法

接下来，就让我们看一下按照图 5-14 的流程改写的中断处理程序代码。

代码 5-15

```
__vector_irq:
    sub r14,r14,#4
    str r14,[r13,#-0x4]
    mrs r14,spsr
    str r14,[r13,#-0x8]
    str r0,[r13,#-0xc]
    mov r0,r13
    CHANGE_TO_SVC
    str r14,[r13,#-0x8]!
    ldr r14,[r0,#-0x4]
    str r14,[r13,#-0x4]
    ldr r14,[r0,#-0x8]
    ldr r0,[r0,#-0xc]
    stmdb r13!,{r0-r3,r14}
    bl common_irq_handler
    ldmia r13!,{r0-r3,r14}
    msr spsr,r14
    ldmfd r13!,{r14,pc}^
....
```

在代码 5-15 中，第一步的工作仍然是计算返回地址，对于中断程序，返回地址的处理方法都是一致的。

紧接着，程序再将该返回地址保存到一段内存当中。返回地址被保存到哪块内存并不重要，重要的是该内存地址不能被别的进程破坏，并且同时允许管理模式访问，这样程序才能在管理模式中把这个返回地址读出来。

我们当然可以像确定异常堆栈那样，预先在系统初始化时就划分出块独立的空间用来保存。不过，更好的办法是像代码 5-15 那样，将该值暂时地保存到中断模式堆栈的低地址处。因为堆栈指针都是向下生长的，类似于 R13_irq-#0x4 这样的地址，都是没有被使用过的地址，因此可以用来保存临时数据。

同样的方法也适用于 SPSR 寄存器，也就是上一个状态的 CPSR 寄存器的值。代码 5-15 首先将这个值赋给 R14_irq，然后保存到 R13_irq-#0x8 处，因为 R14_irq 的值已经备份过了，因此这里可以将它当成是一个普通寄存器使用。

然后，我们还需要将 R0 的值保存到 R13_irq-#0xC 处，这是因为 R0 这个寄存器需要保存 R13_irq 的值。通过 R0 寄存器可以将中断模式下的堆栈指针传递给管理模式。

当前面这些工作完成后，调用 CHANGE_TO_SVC 切换到管理模式。在管理模式下，代码 5-15 首要的任务是恢复寄存器，要恢复的寄存器包括：

- 被中断的状态的返回值
- 被中断的状态的 CPSR 值
- R0 寄存器

它们分别被保存在 R13_irq-#0x4、R13_irq-#0x8 和 R13_irq-#0xC 三个地址中。因为在中断模式下，R13_irq 已经被保存到 R0 中，这样，在管理模式里，我们可以通过 R0-#0x4、R0-#0x8 和 R0-#0xC 来读取它们。读出这些值后，还需要按照一定的要求对它们正确地进行压栈。压栈的顺序自顶向下分别是被中断状态的返回值、管理模式原来的 R14 值、被中断状态的 CPSR 值，还有 R0~R3 寄存器的值。

当压栈完成后，程序立刻调用函数 common_irq_handler 进行具体的中断处理工作。之后，程序依次恢复各个寄存器，并从管理模式直接返回到中

断之前的运行状态。

代码 5-15 实现了图 5-14 的执行流程。这样当程序切换到管理模式后，就可以直接开启全局中断，直至最后返回，全局中断都不需要被禁止，从而能够更大限度地提高中断嵌套的效率。

当然，为了配合代码 5-15，对 common_irq_handler 函数要稍做修改。

代码 5-16

```
void common_irq_handler(void){  
    unsigned int tmp=(1<<(*(volatile unsigned int *)INTOFFSET));  
    printk(" %dt",*(volatile unsigned int *)INTOFFSET);  
    *(volatile unsigned int *)SRCPND|=tmp;  
    *(volatile unsigned int *)INTPND|=tmp;  
    enable_irq();  
    printk(" interrupt occurred\n");  
}
```

在代码 5-16 中，从 enable_irq() 函数开始直至整个中断处理过程的结束，全局中断都是被允许的，程序中不需要调用 disable_irq() 函数。

使用同样的方法运行代码 5-15 和代码 5-16，我们将会看到与前面一致的结果。

```
helloworld  
test_mmu  
testing printk  
test string :::: this is %s test  
test char :::: H  
test digit :::: -256  
test X :::: 0xffffffff00  
test unsigned :::: 4294967040  
test zero :::: 0  
14 interrupt occurred  
.....
```

一个更加优秀的中断处理方法就大功告成了！

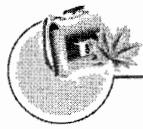
5.5 总结

至今还记得一篇小学课本上的文章，是由世界著名数学大师华罗庚写的《统筹方法》，要解决的问题没有变，但是经过对操作步骤和结构的适当调整后，工作效率就被大大提高了。正如文章中所说的那样，统筹方法是一种安排工作进程的数学方法，它的实用范围极其广泛，在操作系统的设计中也不例外。

正是因为这种基本思想的出现，在芯片级，人们便设计了两种执行逻辑，就是本章一开始提到的进程和中断。这两种执行逻辑共同努力、各司其职，从而整体地提高了系统的运行效率。

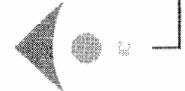
同样，在进程或中断内部也无时无刻不充斥着科学的设计思想。结合本章的例子，我们也可以清楚地看到，同为中断处理程序，依托于某种硬件平台，采用不同的处理方法，最终能够达到的效果也不尽相同。

计算机科学不是局限在程序、语言上的一种小技巧，而是从抽象的算法分析、形式化语法到软硬件等的一个系统的科学门类。



第 6 章

动态内存管理



程序要运行，就必须加载到内存中。早期的程序员，无论是否是嵌入式领域里的，都会想尽各种办法让自己写的程序尽可能少地占用内存。因为那时候内存非常得小，随便申请一块内存，可能就会超出系统最大内存值。也正是由于内存的限制，那时的程序功能并不强大。当然，随着技术的发展，内存越来越廉价，容量也越来越大。所以，现在的程序员似乎已经不大关心内存的占用问题了。

理想的内存模型具备两个条件：无限容量和无限速度。不幸的是，这两个条件在实际应用中都不可能满足。因此操作系统作为管理者，必须要肩负起管理内存的重任，让有限的内存发挥出无限的功能。

实际上，一个操作系统对内存的处理有两个基本原则。

一是怎样可以让内存的消耗最小。这一点的意义不言自明，内存消耗小则运行该操作系统的硬件成本就低，最终产品化后，企业就拥有了更低的价格优势。这里，我们绝对不是泛泛而谈，这种例子屡见不鲜。有一些芯片，如 AT91SAM7 系列，其内部集成了静态 RAM，这样，使用这种芯片的产品就可以省去外接 RAM 所带来的成本。如果一个操作系统精于内存节约之道，仅使用内置 RAM 就能够实现必要的功能，那么这样的产品无疑是具有市场价格竞争力的。

二是要尽可能地满足更多程序对内存的需求。节省归节省，但如果进程需要使用内存时，操作系统也不能吝惜。进程的运行是否足够自由是检验一个操作系统是否强大的重要指标。于是，在一个进程没有恶意使用内存的前

提下,尽最大努力满足进程对内存的需求就成为了操作系统处理内存的又一核心问题了。

上述两点只是原则上的,如果要在实际应用中去满足这两个原则,通常可以使用以下两种手段。

1) 内存映射。将物理内存映射到虚拟内存中,变有限为无限。

2) 有效管理。使用各种内存管理算法减少内存浪费,将一分钱掰成两半花。

对于第一种方法,我们在操作系统的启动一章中已有描述。理论上,经地址映射之后,用户程序可以使用的内存将会由原来的有限大小变成4G内存空间(对于32位CPU来说),这使得进程不会为系统可用内存的多少而纠缠不清,从而使进程能够去任意发挥。

然而,内存映射却不能解决所有的问题。这种方法只是看起来将内存扩大了,而实际的可用内存并没有丝毫增加。于是,如何对内存进行合理规划来减少内存的浪费,提高内存利用率,就成为了一个操作系统必须要考虑的问题。

当然,并不是所有的嵌入式操作系统都同时使用以上两种方法。很多嵌入式操作系统,比如UCOS,都被设计成可以运行在一些没有内存管理单元的CPU中,所以不具备内存映射的能力。对于这些操作系统来说,使用适当的内存管理算法是解决内存问题的唯一途径。而对于那些高端嵌入式操作系统,在完成内存映射操作的基础之上也会双管齐下,对映射完成后的虚拟内存进行有效的管理。

于是,本章将会把重点放到内存管理的方法上,让我们自己的操作系统也拥有最适合进程生存的土壤。

6.1 伙伴算法

通常情况下,一个高级操作系统必须要给进程提供基本的、能够在任意时刻申请和释放任意大小内存的功能,就像malloc函数那样。然而,实现malloc函数并不简单。由于进程申请内存的大小是任意的,如果操作系统对



malloc 函数的实现方法不当，将直接导致一个不可避免的问题，那就是内存碎片。

通俗地讲，内存碎片就是内存被分割成很小很小的一些块，这些块虽然是空闲的，但是却小到无法使用。随着申请和释放次数的增加，内存将变得越来越不连续。最后，放眼望去，整个内存将只剩下碎片，没有有效的内存可用。所以减少内存浪费的核心是尽量避免产生内存碎片。

针对这样的问题，有很多种解决方法，伙伴算法就是其中之一。伙伴算法被证明是非常行之有效的一套内存管理算法，因此也被相当多的优秀操作系统所采用。本节中，我们就来介绍一下伙伴算法的原理，然后再结合实例深入理解一下操作系统中伙伴算法的具体实现。

什么是伙伴算法？简而言之，伙伴算法就是将内存分成若干小块，然后尽可能以最适合的方式满足程序内存需求的一种内存管理算法。伙伴算法的一大优势是它能完全避免外部碎片的产生。

那么什么又是外部碎片呢？我们来举个例子，假设系统可提供 1M 的内存空间供进程使用，由于算法设计等诸多原因，系统将这 1M 内存分成 4 个区域，让每个区域的默认大小为 300K。这样的结果是最后一个内存区域将不足 300K，此时，这个不足 300K 的内存区虽然理论上可以被使用，但由于容量偏小，将这段内存分配出去也许不能够满足程序的正常需要，同时操作系统也不得不专门去处理这段唯一的、不足默认 300K 大小的内存区域，这也为程序的设计带来了难度。这最后一段内存区域就像鸡肋一样，弃之不舍，食之无味，它就代表了我们所说的外部碎片。

当然，这里默认内存大小为 300K 的假设过于灵活了。很多关于操作系统原理的书中都会将这些概念与硬件分页机制相结合。这样一来，默认内存大小将会被限制为页的大小，而外部碎片就很难再被利用了。

不过，既然我们的目的不是对内存管理算法做理论性研究，很多概念上的东西也不需要解释得太精确。通过一个形象的例子来理解问题，反而是一件好事。与外部碎片相对应的另一个概念是内部碎片。内部碎片是指当某一个程序申请了一块内存，而此时系统中却没有大小恰好合适的内存提供给该程序，于是系统只能给该程序分配一块稍大一点的内存，而这块内存中会有一小部分根本不会被使用，这一部分内存就叫做内部碎片。

还拿前面的那个例子来说明问题。如果一个程序只想向系统申请 298K



的内存，但操作系统怎么会有恰好合适的内存分配给它呢？唯一的办法就是直接分配给这个进程一个 300K 的内存。这样一来，进程只利用了其中的 298K，而余下的 2K 就变成内部碎片浪费掉了。

伙伴算法虽然能够避免外部碎片的产生，但这恰恰是以产生内部碎片为代价的。

6.1.1 伙伴算法的原理

伙伴算法是如何工作的呢？

想要使用伙伴算法来进行内存管理。首先，系统需要将一块平坦的内存划分成若干块，可以用“buddy”这个词来描述。当然，这些内存块并不是任意大小的。根据伙伴算法的经典理论，这些内存块的大小必须是 2 的整数次方。例如，一块 1M 的内存可以按照 64K、128K、256K 和 512K 等的数值分成若干块。其实按照什么数值分块还是有一定原则的。通常我们选择比可用内存数小的 2 的整数次方的最大值。例如，对于一个 8M 的内存，最大的内存块可以是 4M，而余下的 4M 内存用来划分小于 4M 的块。这样可以最大限度地保证程序能够从系统中申请到尽可能多的内存。

确定了最大块的数目之后，我们还需要确定最小内存块的大小。通常最小内存块大小的取值应该适中。如果取值过大，比如 1M，当一个程序需要申请 100K 内存时，操作系统也只能将 1M 的内存分配给该程序，从而使余下的 924K 内存变成碎片浪费掉。但如果最小内存块数值过小，则会加重操作系统搜索内存块的负荷。

在内存块划分完成后，系统就可以对内存进行分配和释放了。在分配内存时，首先从空闲的内存中搜索比申请的内存大的最小的内存块。如果这样的内存块存在，则将这块内存标记为“已用”，同时将该内存分配给应用程序。如果这样的内存不存在，则操作系统将寻找更大块的空闲内存，然后将这块内存平分成两部分，一部分返回给程序使用，另一部分作为空闲的内存块等待下一次被分配。

当程序释放内存时，操作系统首先将该内存回收，然后检查与该内存相邻的内存是否是同样大小并且同样处于空闲的状态。如果是，则将这两块内存合并，然后程序递归进行同样的检查。

以上就是伙伴算法的实现过程。不过，单纯的文字描述真的是非常枯燥、难以理解。下面我们通过一个例子，来更深入地理解一下伙伴算法的真正内涵。

假设系统中有 1M 大小的内存需要动态管理，按照伙伴算法的要求，需要将这 1M 的内存进行划分。这里，我们将这 1M 的内存划分成 64K、64K、128K、256K 和 512K 共五个部分，如图 6-1 (a) 所示。

此时，如果有一个程序 A 想要申请一块 45K 大小的内存，则系统会将第一块 64K 内存块分配给该程序，如图 6-1 (b) 所示。

然后程序 B 向系统申请一块 68K 大小的内存，系统会将 128K 内存块分配给该程序，如图 6-1 (c) 所示。

接下来，程序 C 要申请一块大小为 35K 的内存，系统将空闲的 64K 内存分配给该程序，如图 6-1 (d) 所示。

之后程序 D 需要一块大小为 90K 的内存。当程序 D 提出申请时，系统本该分配给程序 D 一块 128K 大小的内存，但此时内存中已经没有空闲的 128K 内存块了，于是根据伙伴算法的原理，系统会将 256K 大小的内存块平分，将其中一块分配给程序 D，另一块作为一个空闲内存块保留，等待以后使用，如图 6-1 (e) 所示。

紧接着，程序 C 释放了它申请的 64K 内存。在内存释放的同时，系统还负责检查与之相邻并且同样大小的内存是否也空闲，由于此时程序 A 并没有释放它的内存，所以系统只会将程序 C 的 64K 内存回收，如图 6-1 (f) 所示。

然后程序 A 也释放掉由它申请的内存，系统随即发现与之相邻且大小相同的一段内存块恰好也处于空闲状态。于是，将二者合并，如图 6-1 (g) 那样。

之后程序 B 释放掉它的 128K 内存，系统将这块内存与相邻的 128K 内存合并成 256K 的空闲内存，如图 6-1 (h) 所示。

最后程序 D 也释放掉了它的内存，经过三次合并后，系统得到了一块 1024K 的完整内存，如图 6-1 (i) 所示。

结合上面的例子，原本枯燥无味的算法原理现在看起来是不是很简单了呢？下面我们趁热打铁，尝试在自己的操作系统中实现伙伴算法。

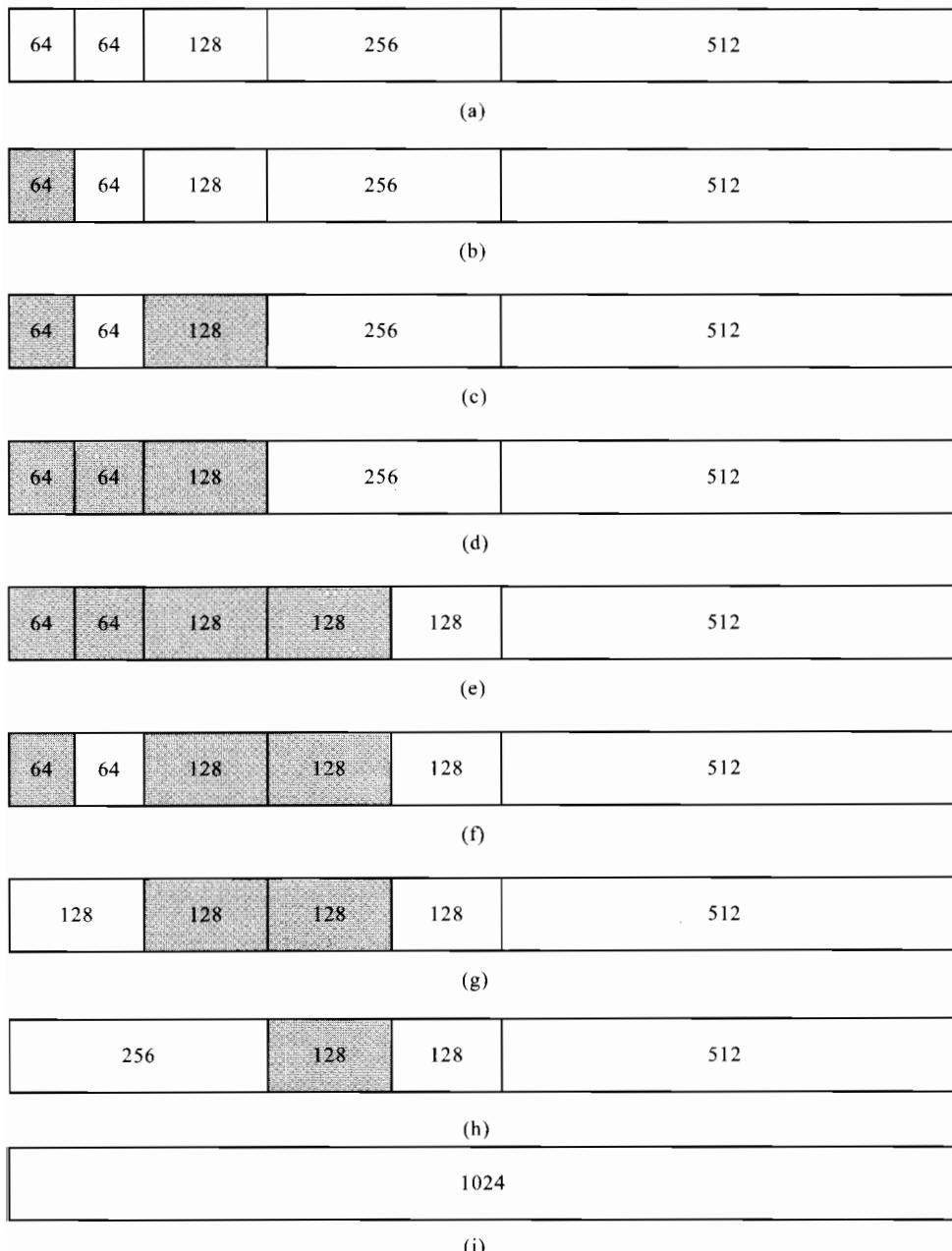


图 6-1 伙伴算法示例

6.1.2 伙伴算法的实现

对于一个操作系统来说，所承担的内存管理角色较之于用户程序有着明显的不同。此时由于内存使用上受到了限制，操作系统还不具备动态数据管理的有效方法。于是，我们不得不只采用静态的办法来管理算法中需要的动态数据。在实现伙伴算法之前，对要管理的内存进行合理的规划是非常必要的。

6.1.2.1 内存规划

目前在我们的操作系统所使用的 8M 内存中，开头的部分是用来存储内核代码本身的，而结尾则用来存储各模式的运行堆栈、页表等系统关键数据。所以，能够进行动态管理的内存区域仅是中间的那一部分。

内存分区情况如图 6-2 所示。

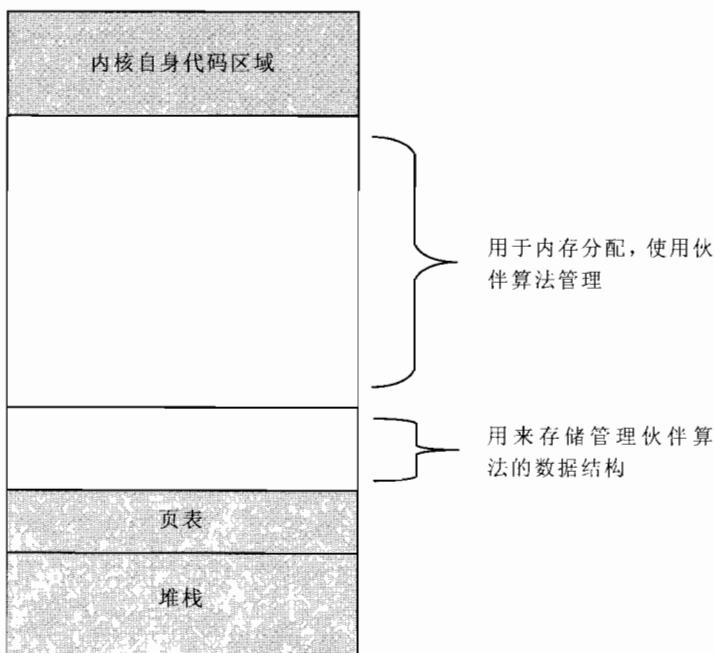


图 6-2 内存分区情况

在图 6-2 中，虽然整个可用内存中间的绝大部分都可以作为待分配的内

存进行管理，但同时，我们也必须留出一段特殊的内存空间用来存储管理伙伴算法所必需的数据结构。参考其他操作系统的命名方法，我们不如将用来管理伙伴算法的这一数据结构命名为 page。

代码 6-1

```
struct page {
    unsigned int vaddr;
    unsigned int flags;
    int order;
    struct kmem_cache *cachep;
    struct list_head list;
};
```

回忆一下我们对伙伴算法原理的描述。伙伴算法要求将内存分割成若干块，这些块最小不能低于一个数值，前面的例子中，这个值是 64K。任何一个大于这个值的内存块都必须是这一数值的 2 的 n 次方倍，也就是我们前面例子中的 128K、256K、512K，等等。

于是，我们将这个最小的内存区域叫做一个“页”，也就是前文所说的 64K 大小的内存区域。这里需要先强调一下，因为 page 翻译成中文也叫页，所以为了避免混淆，以后凡是提到 page 这个概念时，我们指的就是代码 6-1 这个数据结构，而当我们需要描述被分配的内存块时，我们就用中文“页”这个词来表达。

读者朋友可能会觉得把被分配的内存块叫做 block 或 area，似乎概念上会更清晰些。我们这样命名其实包含了一个非常重要的信息，有些朋友可能已经猜到了，没错，一个 page 是用来描述一个“页”的，反过来也是一样，一个“页”只能由一个 page 来描述，它们之间是一对一的映射关系。

既然一个 page 结构是用来描述一块最基本的内存区，该结构体中必然含有描述内存区域基本信息的成员，如起始地址、“页”的大小，等等。根据伙伴算法的描述，一个“页”的大小必然是一个固定的值。这里我们会选择与 MMU 的内存管理最小值相一致的值，在我们前面的例子中这个值是 4K。

但请注意，4K 这个值绝不是我们一拍脑门想出来的，里面也有些玄机。通过这种方法，系统可以在使用伙伴算法分配内存的同时将这一内存区映射到其他地址之中。当系统在执行用户应用程序时，这一点就显得相当必要了，

待分配的内存大小和待映射的内存大小相一致可以保证我们在为用户申请了一个“页”的内存空间的同时，以最小的代价实现对该页的映射，提高了系统的运行效率。

既然一个“页”的大小是固定不变的，那么就不需要在 struct page 结构体中体现出来了，我们可以非常简单地使用一个宏定义来描述“页”的大小。这样一来，在 struct page 结构体当中就仅需要一个描述内存区域起始位置的成员了。在代码 6-1 中，成员 vaddr 就扮演了这一重要角色；flags 成员则是一个标志，负责对 page 结构体和“页”进行控制；order 成员专门用于伙伴算法的管理，这个值可以是任意正数、0 或 -1；成员 counter 是一个计数器，用来描述这块内存区被使用了多少次，如果 counter 值为零，代表没有任何进程使用该“页”，那么就可以将它当成空闲“页”处理；cachep 成员是一个 kmem_cache 类型结构体的指针，关于这个指针的内容，我们稍后会详细介绍。

最后一个成员是 struct list_head 结构体，该结构体是一个通用双向链表，其作用就是实现一个双向链表，将一系列同样大小的空闲内存连接起来。接下来我们会花些时间介绍一下这个通用链表结构的原理和使用方法。这么做的原因，一方面是由于该链表结构功能强大，被广泛地应用在各种程序之中，另一方面是因为它具有一定的抽象性，非常通用。掌握了它就能够在今后的编程过程中轻松运用，并发挥出巨大的作用了。

另外，我们这里的通用链表结构其实就来源于 Linux，因此，如果读者对 Linux 内核比较了解的话，也可以略过这部分内容。

6.1.2.2 通用链表结构

直接来看代码：

代码 6-2

```
struct list_head {  
    struct list_head *next, *prev;  
};  
  
static inline void INIT_LIST_HEAD ( struct list_head *list )  
{  
    list->next = list;  
    list->prev = list;
```



```

}

static inline void __list_add( struct list_head *new_lst,
    struct list_head *prev,
    struct list_head *next )
{
    next->prev = new_lst;
    new_lst->next = next;
    new_lst->prev = prev;
    prev->next = new_lst;
}

static inline void list_add( struct list_head *new_lst, struct
list_head *head )
{
    __list_add( new_lst, head, head->next );
}

static inline void list_add_tail( struct list_head *new_lst,
struct list_head *head )
{
    __list_add( new_lst, head->prev, head );
}

static inline void __list_del( struct list_head * prev, struct
list_head * next )
{
    next->prev = prev;
    prev->next = next;
}

static inline void list_del( struct list_head * entry )
{
    __list_del( entry->prev, entry->next );
}

static inline void list_remove_chain( struct list_head
*ch, struct list_head *ct ) {
    ch->prev->next=ct->next;
    ct->next->prev=ch->prev;
}

static inline void list_add_chain( struct list_head *ch, struct

```

```

list_head *ct,struct list_head *head) {
    ch->prev=head;
    ct->next=head->next;
    head->next->prev=ct;
    head->next=ch;
}

static inline void list_add_chain_tail ( struct list_head
*ch,struct list_head *ct,struct list_head *head) {
    ch->prev=head->prev;
    head->prev->next=ch;
    head->prev=ct;
    ct->next=head;
}

static inline int list_empty ( const struct list_head *head)
{
    return head->next == head;
}

#define offsetof( TYPE, MEMBER ) (( unsigned int ) &(( TYPE * )
0 ) ->MEMBER )

#define container_of( ptr, type, member ) ( {
    const typeof( ((type *) 0 )->member ) * __mptr = (ptr); \
    (type *)( ( char *)__mptr-offsetof(type,member) ); } )

#define list_entry( ptr,type,member ) \
    container_of( ptr, type, member )

#define list_for_each( pos, head ) \
    for ( pos = ( head ) ->next; pos != ( head ); pos = pos->next )

```

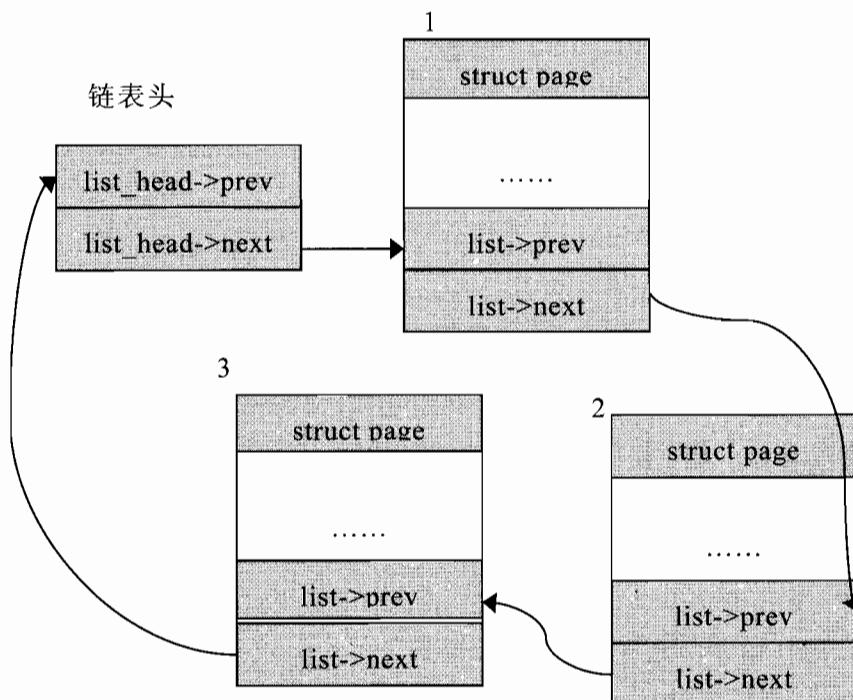
代码 6-2 中包含了对 struct list_head 结构的基本定义和操作方法。这其中，结构体的定义、初始化、添加、删除等内容其实都非常得常规。完全是指向双向链表的通用方法实现的，相信读者都看得明白。此时有的读者朋友可能会问，这样一个普普通通的链表结构又是怎样实现抽象化，发挥出通用性的优势的呢？这一切都源于一个构思精巧的宏——list_entry。

简单说来，如果一个结构体中包含一个 struct list_head 成员，那么 list_entry 这个宏能够从一个指向 list_head 成员的指针中计算出指向该结构

体的指针。

假设目前有三个 struct page 结构体，通过 list_head 连接在一起，如图 6-3 所示。

现在假设这三个 struct page 结构体所描述的“页”都为空闲页。那么，如果某个进程向操作系统申请使用一个页，就必然要从这三个“页”中分配。图 6-3 中的这三个 struct page 并不是通过某个类型为 struct page 的指针连接在一起的，而是使用了 struct list_head 结构体。



注：图中的 prev 指针示意省略未画

图 6-3 通用链表结构

内存分配的时候，系统首先会从链表头开始去查找它的 next 成员所指向的结构体。根据通用链表的定义，这个指针指向另一个 struct list_head 结构体。于是我们就得到了一个指向 struct list_head 结构体的指针，而这个指针恰好又是一个空闲的 struct page 结构体的成员了。

接下来又要如何通过 struct list_head 指针得到 struct page 指针呢？这其

实很简单。看一下代码 6-1，`struct page` 一共包含 5 个成员，前 4 个成员都是整型，默认大小为 4 个字节。因此，在已知第 5 个成员（`struct list_head`）地址的情况下，想要求出结构体（`struct page`）首地址，只需要减去 `sizeof(int) * 4` 个字节的偏移即可。

前面说的 `list_entry` 这个宏就是专门用来获取结构体首地址的，只不过它的实现方法更加“智能”，我们来具体分析一下。

首先来看看该宏的第一行表达式：

```
const typeof( ((type *)0)->member ) *__mptr = (ptr)
```

这里的 `type` 其实是宏的一个参数，使用时应该传递包含有 `list_head` 结构体的类型，`ptr` 是另外一个宏参数，代表指向 `list_head` 结构体的指针。`member` 也是 `list_entry` 的一个参数，它表示包含 `list_head` 结构体的 `list_head` 类型成员的名字。

根据前面的例子，`struct page` 中 `list_head` 类型成员名为 `list`。假设指针 `struct list_head *p` 恰好指向了 `struct page` 结构体的 `list` 成员，那么展开之后，上面的表达式就会像下面这样：

```
const typeof( ((struct page *)0)->list ) *__mptr = (p)
```

这样看来，这个表达式清晰很多，注意，其中的 `typeof` 是 GCC 针对 C 语言的一个扩展关键字，它的作用是得出其参数的具体类型。例如，有一个指针定义为 `int foo`，那么 `typeof(foo)` 的最终结果就应该是 `int`。

将上述所有内容综合起来，整个表达式的含义就一目了然了。

宏 `list_entry` 首先将地址 0 强制转成 `struct page` 类型的指针，取其中的 `list` 成员，然后通过 `typeof` 提取它的类型，也就是 `struct list_head`。这就相当于定义了一个 `const struct list_head` 类型的指针数据 `__mptr`，并让它的值等于已知的 `struct list_head` 指针 `p`。

然后，`list_entry` 宏又通过第二行的表达式返回一个指向 `struct page` 的指针，其形式如下：

```
(type *)( (char *)__mptr + offsetof(type,member) )
```

展开之后，该表达式等价于如下形式：

```
(struct page *)( (char *)__mptr - offsetof(struct_page,list) )
```

宏 offsetof 能求出结构体首地址与它的某一个成员地址的偏移。其定义如下：

```
((unsigned int) &((TYPE *)0) ->MEMBER)
```

展开之后，表达式变成：

```
((unsigned int) &((struct page *)0) ->list)
```

这里的意思是，假设有一个 struct page 结构体从内存 0 位置开始，那么 list 成员的地址值就等于 list 成员地址与结构体首地址的偏移值。

宏 list_entry 最后返回的结果就是一个指向 struct page 结构体的指针。这样，针对 struct list_head 结构体的最重要的一种操作方法，我们就介绍完了。

如果有的读者此时正在思考下面这个问题的话，那您也一定闻到了这阵四溢的茶香了。

其实，list_head 宏其实还蕴藏着一处妙笔，list_entry 这个宏完全可以写成如下形式：

```
((struct page *)((char *)ptr) - offsetof(struct_page, list))
```

乍看之下，这个表达式相比代码 6-2 中 list_entry 的定义更加简单，那为什么在代码 6-2 中，非要定义一个临时变量 __mptr 并赋值为 ptr，然后代替 ptr 来计算地址值呢？

其实，这里使用 const typeof(((type *)0) ->member) *__mptr 这种表达方法有一个好处，那就是可以强迫宏进行类型检查。

我们知道，宏定义本身并不会进行类型检查，而这种表达方法会迫使编译器检查 type 类型是否含有 member 成员。同时，如果程序在编译的时候打开了优化选项，那么 __mptr 这个变量也会被优化掉，最终生成的代码与 ((struct page *)((char *)ptr) - offsetof(struct_page, list)) 表达式生成的代码也没有什么不同。这样一来，使用代码 6-2 中 list_entry 的实现方法，既能够进行类型检查，提高了安全性，同时在效率上又没有丝毫的浪费。

正是由于有 list_entry 这个宏，当我们想将任何一种结构体类型连接成双向链表时，便不再需要专门给这个结构体写一套操作方法，而只需要将 struct list_head 作为这个结构体的成员，使用代码 6-2 中的一系列方法操作 struct list_head 链表。当需要得到结构体的指针时，只需要调用 list_entry 宏就可以了。



除了 `list_entry` 宏定义外，另一个比较常用的就是 `list_for_each` 宏。该宏能够遍历由 `list_head` 连接而成的整个链表。实际上，这个宏是通过一个 `struct list_head` 结构体的指针从链表头开始，每次都将当前操作链表的 `next` 值赋给该指针，直到该指针的值再次等于链表头的地址值为止，从而将整个链表循环了一遍。

6.1.2.3 与内存管理相关的宏定义

在掌握了一套构建链表的方法后，就可以开始进行伙伴算法的编码了。

首先我们需要参考图 6-2，规划一下内存边界。其原因正如前文所说，我们将内存按照“页”的大小进行划分的同时，还需要分配出同样多个用来存储 `struct page` 结构体的空间。

举例来说，假设一个“页”有 4K 大小，并且系统中共有 6M 空间可以利用，那么系统最多可以被划分出如下这么多个“页”。

$6M / (4K + \text{sizeof}(\text{struct page}))$

我们需要一些宏定义辅助我们在程序中实现内存划分。

代码 6-3

```
#define _MEM_END    0x30700000
#define _MEM_START   0x300f0000

#define PAGE_SHIFT   (12)
#define PAGE_SIZE    (1 << PAGE_SHIFT)
#define PAGE_MASK    (~(PAGE_SIZE - 1))

#define KERNEL_MEM_END  (_MEM_END)

#define KERNEL_PAGING_START \
({ _MEM_START + (~PAGE_MASK) & ((PAGE_MASK)) })
#define KERNEL_PAGING_END  (((KERNEL_MEM_END - \
                           KERNEL_PAGING_START) / (PAGE_SIZE + \
                           sizeof(struct page))) * (PAGE_SIZE) + \
                           KERNEL_PAGING_START)

#define KERNEL_PAGE_NUM   ((KERNEL_PAGING_END - \
                           KERNEL_PAGING_START) / PAGE_SIZE)

#define KERNEL_PAGE_END _MEM_END
```

```
#define KERNEL_PAGE_START  (KERNEL_PAGE_END-\n    KERNEL_PAGE_NUM*sizeof(struct page))
```

代码 6-3 通过一系列宏计算出了内存中各区域的大小。

`_MEM_START` 的值被定义为 `0x3000f000`，因为我们的虚拟硬件平台的默认物理内存地址是从 `0x30000000` 开始的，并且操作系统自身的代码也被加载到 `0x30000000` 处。那么，这里就简单地认为操作系统在运行时的大小将小于 `0xf000`。于是，从 `0x3000f000` 处开始向高地址延伸的内存就可以用来进行分配了。

同样地，`_MEM_END` 宏代表了可用内存的结束地址，即 `0x30700000`。在前面的例子中，我们都假设虚拟硬件平台共有 `8M` 物理内存。这其中，`0x30700000~0x30800000` 的内存区域被各模式的堆栈以及页表所占用，因此不能随意被修改。

这样一来，从`_MEM_START` 开始到`_MEM_END` 结束的内存空间，就都可以用来参与伙伴算法的分配了。如图 6-2 那样，这其中既包含了待分配的“页”，又包含了管理这些“页”的 `struct page` 结构体。

按照前文的描述，一个“页”的大小被设计成 `4K`，也就是代码 6-3 中 `PAGE_SIZE` 的值。而宏定义 `KERNEL_PAGING_START` 代表的则是 `_MEM_START` 的值按照 `PAGE_SIZE` 对齐的地址。这里的对齐指的是能不能被整除。如果 `_MEM_START` 本身就是按照“页”对齐的，那么这里的 `KERNEL_PAGING_START` 的值就应该等于 `_MEM_START`，而如果 `_MEM_START` 的值不是按页对齐的，如 `0x3000f010` 这样的值，那么 `KERNEL_PAGING_START` 的值就会向下取整，其结果最终会是 `0x30010000` 这个值。

`KERNEL_PAGING_END` 值的计算就稍微复杂了点。首先需要通过 `_MEM_END-KERNEL_PAGING_START` 来计算可用内存的大小，然后再用这个值除以 `PAGE_SIZE+sizeof(struct page)` 来计算该可用内存区一共能容纳多少个页，接着再用这个值乘以 `PAGE_SIZE`，之后再加上 `KERNEL_PAGING_START`，最终得到的就是内存中被分配的“页”的结束地址了。

知道了页的起始地址和结束地址之后，我们便可以通过下面这个表达式来计算系统中一共包含多少个“页”。

```
(KERNEL_PAGING_END-KERNEL_PAGING_START)/PAGE_SIZE
```

这个值被定义成了 KERNEL_PAGE_NUM 宏。当然，有多少个“页”，就会有多少个 struct page 结构体。于是，如果用于存放 struct page 结构体的内存区域到 _MEM_END 结束的话，那么这一区域的起始地址就应该是：

```
(KERNEL_PAGE_END-KERNEL_PAGE_NUM*sizeof(struct page))
```

也就是代码中 KERNEL_PAGE_START 这个值。

这样，我们便完成了操作系统的内存划分。当所有准备工作都已完成，接下来我们就一起来看看伙伴算法的核心实现。下面首先介绍伙伴算法是如何初始化的。

6.1.2.4 buddy 的初始化

初始化部分的代码如下：

代码 6-4

```
#define PAGE_AVAILABLE      0x00
#define PAGE_DIRTY          0x01
#define PAGE_PROTECT         0x02
#define PAGE_BUDDY_BUSY      0x04
#define PAGE_IN_CACHE        0x08

#define MAX_BUDDY_PAGE_NUM   (9)

#define AVERAGE_PAGE_NUM_PER_BUDDY \
    (KERNEL_PAGE_NUM/MAX_BUDDY_PAGE_NUM)
#define PAGE_NUM_FOR_MAX_BUDDY \
    ((1<<MAX_BUDDY_PAGE_NUM)-1)

struct list_head page_buddy[MAX_BUDDY_PAGE_NUM];

void init_page_buddy(void) {
    int i;
    for (i=0;i<MAX_BUDDY_PAGE_NUM;i++) {
        INIT_LIST_HEAD(&page_buddy[i]);
    }
}

void init_page_map(void) {
    int i;
    struct page *pg=(struct page *)KERNEL_PAGE_START;
    init_page_buddy();
}
```

```
for ( i=0; i<( KERNEL_PAGE_NUM ) ; pg++, i++ ) {  
    pg->vaddr=KERNEL_PAGING_START+i*PAGE_SIZE;  
    pg->flags=PAGE_AVAILABLE;  
    INIT_LIST_HEAD ( & ( pg->list ) );  
  
    if ( i<( KERNEL_PAGE_NUM&\n        (~PAGE_NUM_FOR_MAX_BUDDY) ) ) {  
        if (( i&PAGE_NUM_FOR_MAX_BUDDY ) ==0 ) {  
            pg->order=MAX_BUDDY_PAGE_NUM-1;  
        } else{  
            pg->order=-1;  
        }  
        list_add_tail ( & ( pg->list ) ,\n            &page_buddy[MAX_BUDDY_PAGE_NUM-1] );  
    } else{  
        pg->order=0;  
        list_add_tail ( & ( pg->list ) ,&page_buddy[0] );  
    }  
}
```

代码 6-4 似乎长了一些，我们慢慢来看。首先，程序定义了一个宏 MAX_BUDDY_PAGE_NUM，这个值与系统分配的最大内存的阶数有关。回忆一下前文中对伙伴算法原理的描述，在伙伴算法中 buddy 代表了一个可供分配的内存区。一个 buddy 的大小不是任意的，而必须是最小内存区大小的 2 的整数次方倍，如 4K、8K、16K、32K…。那么 MAX_BUDDY_PAGE_NUM 宏描述的就是所有不同大小的 buddy 的个数。

为了简化程序设计，此处我们选择了一个定值 9，表示从 2 的 0 次方一直到 2 的 8 次方共有 9 组 buddy 可以分配。换句话说，伙伴算法允许至少申请一个“页”，也就是 4K 大小的内存，同时也允许我们最大申请 256 个页，即 1M 内存。相对于系统不足 7M 的可用内存空间来说，这个范围还算合理。

参考伙伴算法的原理，为了能够更好地管理各组不同大小的 buddy，我们需要将同组的 buddy 连接成双向链表。结合通用链表一节中的描述，这需要系统中定义 MAX_BUDDY_PAGE_NUM 个链表头。

于是在代码 6-4 中，程序通过下面这行代码实现了链表头的定义：

```
struct list_head page_buddy[MAX_BUDDY_PAGE_NUM]
```

读者也可以通过索引数组成员来找到对应的链表头，而数组的索引值正是 buddy 的阶数，如 `page_buddy[5]` 这个链表头所代表的双向链表都是大小为 2 的 5 次方的 buddy。

对这些链表头的初始化是通过 `init_page_buddy` 函数来实现的，该函数循环调用了 `INIT_LIST_HEAD` 函数，将每一个链表头的成员指针初始化为 `NULL`。`INIT_LIST_HEAD` 函数的实现方法见代码 6-2。

对各个 buddy 的初始化最终是由 `init_page_map` 函数实现的，这个函数首先调用 `init_page_buddy` 初始化链表头，然后通过循环分配 buddy，原则是尽可能分出空间最大的 buddy，当剩余空间不足以进行最大 buddy 的分配时，则该空间就按照最小的 buddy 进行分配。最终初始化的结果是系统中只包含有最大和最小的两种 buddy，如图 6-4 所示。

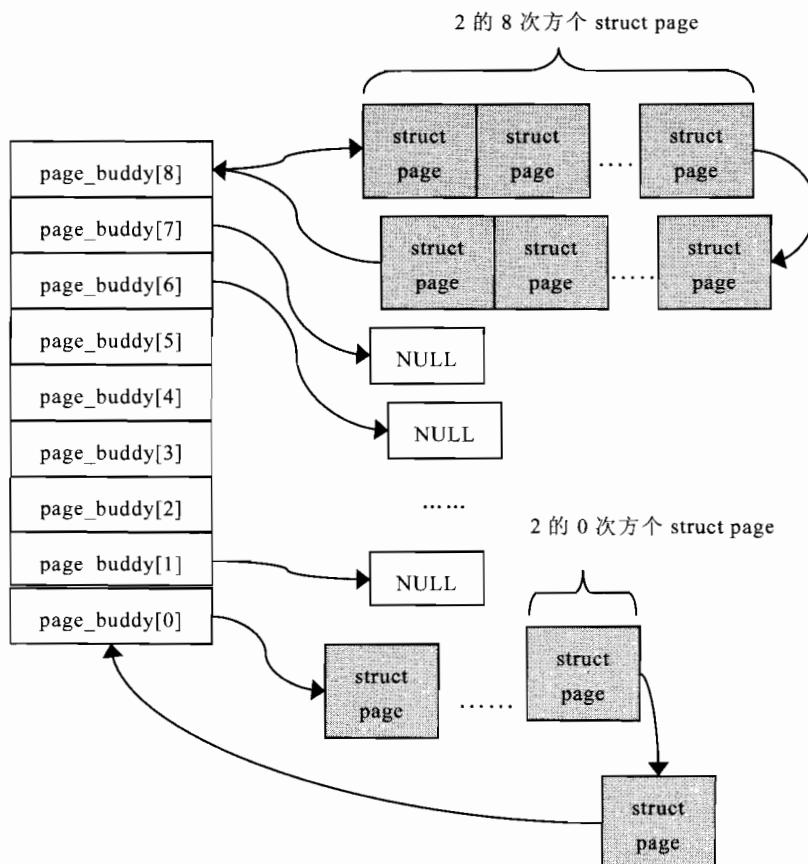


图 6-4 伙伴算法的初始化过程

读者朋友们此时也许正琢磨着，为什么要采用这样一种方式来初始化 buddy。根据伙伴算法，当系统中没有大小刚好合适的 buddy 时，系统可以将一个更高质量级的 buddy 拆成小块，再进行分配。但如果系统中初始化时就没有大块的 buddy，那么在首次分配时，有可能会出现没有可用 buddy 的情况。因此初始化 buddy 时，遵循大块优先的原则是比较合理的。

另一方面，统计结果说明系统中使用频率最高的内存绝大多数都小于 4K。因此，如果系统中不存在单“页”的 buddy 供程序频繁地申请，就必须从更高质量级的 buddy 中去分配。因此，预留一些单“页”的 buddy 供系统频繁使用也是提高效率的一种有效策略。

搞清楚了 buddy 的分配策略后，接下来我们来了解一下处理 struct page 结构的细节问题。对于任何一个 buddy 来说，其中都包含了一个或者多个“页”的空间，而每个页又都唯一对应着一个 struct page 结构体。于是，怎样来描述一个 buddy 就成了我们首先要考虑的问题。

这里我们使用的方法是，利用整个 buddy 中开始位置的 struct page 结构体来代表整个 buddy。于是，就需要一种手段区分出开头的 struct page 结构和普通的 struct page 结构，order 成员恰好可以解决这个问题。

在 buddy 初始化的过程中，我们首先要找到描述整个 buddy 最开头的那个“页”的 struct page 结构体，然后将这个结构体的 order 成员赋值成相应 buddy 的阶数，同时将那些普通的 struct page 结构体的 order 成员都置成 -1，如图 6-5 所示。

这样一来，order 这个成员就具有了双重含义，一方面我们可以通过读取 order 的值直接得到当前 page 结构体属于哪个 buddy，另一方面，又可以利用它来判断这个 page 结构体是否是所属 buddy 的上边界。struct page 结构体的 order 成员在 buddy 的分配和释放过程中都会起到至关重要的作用。

在完成了系统中所有 struct page 结构体的初始化工作后，程序通过 list_add_tail 函数将所有 struct page 结构通过 list_head 结构体串联起来，添加到各自 buddy 链表头的尾部。

至此，伙伴算法初始化的部分就完成了。

该 buddy 包含 2 的 3 次方个 struct page

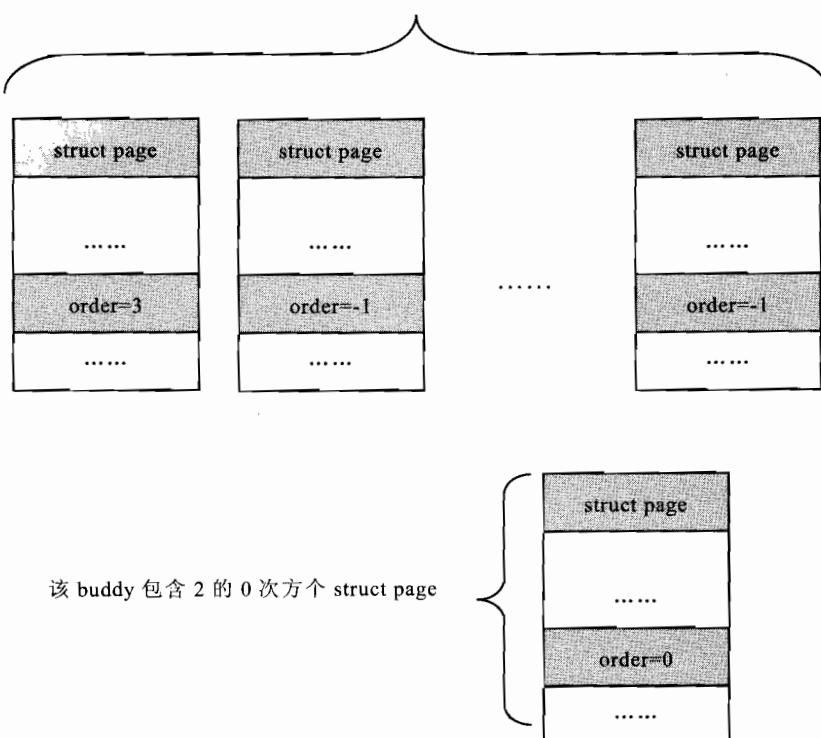


图 6-5 buddy 与 struct page 结构体的关系

6.1.2.5 buddy 的申请和释放

应用伙伴算法进行内存分配时，无非就是两种情况，其中一种情况是相对应的链表头下面恰好有空闲的 buddy，此时，我们只需要将一个 buddy 从链表中提出来即可，这种情况最为简单，通过下面几行代码就可以实现。

代码 6-5

```
.....
#define BUDDY_END(x,order) ((x)+(1<<(order))-1)

.....
pg=list_entry(page_buddy[neworder].next,struct page,list);
tlist=&BUDDY_END(pg,neworder)->list;
tlist->next->prev=&page_buddy[neworder];
.....
```

```
page_buddy[neworder].next=tlist->next;
```

举例来说,如果我们要从系统中申请大小为 2 的 2 次方个“页”的 buddy,那么就需要首先找到阶数为 2 的链表头,该链表头的 next 成员所指向的地址其实就代表了一个空闲的 buddy,只不过这个地址的内容是一个 struct list_head 结构体。于是我们需要通过它得到 struct page 结构体的指针,而 list_entry 宏定义正好可以满足要求,这些都是我们前面介绍过的。

这个动作通过下面这行代码就可以实现:

```
pg=list_entry(page_buddy[order].next,struct page,list)
```

此时我们得到了内存中连续的 4 个“页”,但请注意,这里返回的并不是这块连续内存区的首地址,而是与这 4 个“页”中第一个“页”相对应的 struct page 结构体的地址。根据前面初始化部分的描述,这个 struct page 结构体的 order 成员的值应该是 2,而与另外 3 个“页”相对应的 struct page,它们的 order 成员都为 -1。,虽然这个结构体能够代表整个 buddy,但想得到 buddy 首地址,还需要做一个小小的动作,即取出 struct page 结构体的 vaddr 成员。

在得到了连续的 buddy 之后,程序就必须把这个新的 buddy 从链表中拆下来,以免被再次分配。这就需要将相应 buddy 的链表头的 next 指针指向下一段空闲的 buddy。于是,我们需要得到标记下一段空闲 buddy 开始的 struct page 结构体取出它的 list 成员,赋值给 page_buddy[order]->next,等待下次分配。

申请 buddy 的示例如图 6-6 所示。

在代码 6-5 中, list_entry 函数通过 page_buddy[n] 的 next 成员得到一个指向 struct page 的指针 pg,这里,n 就是该 buddy 的阶数。之后,宏 BUDDY_END 通过 pg 指针得到整个 buddy 结尾的那个 struct page。这样一来,buddy 所包含的起始和结束的 struct page 指针我们就都得到了,因此,可以将整个 buddy 从 page_buddy[n] 链表中删除并返回给用户。此时,在 page_buddy[n] 链表不为空的情况下,一个 buddy 就分配完成了。

buddy 的分配还有另外一种更复杂的情况,即当 page_buddy[n] 链表为空时,就没有恰好空闲的 buddy 可用了。这样,根据伙伴算法的原理,程序只能去搜索 page_buddy[n+1] 链表并对新的 buddy 进行拆分。

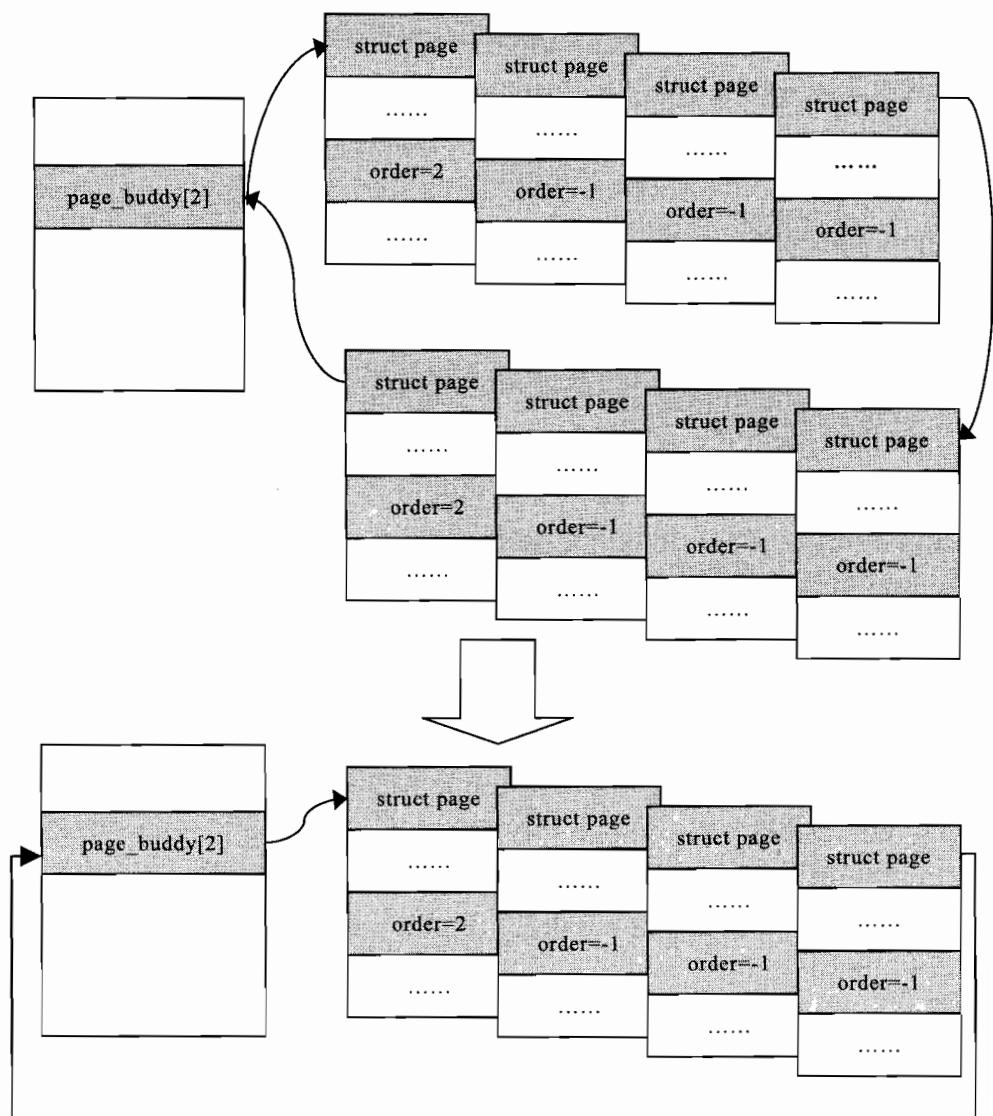


图 6-6 buddy 的申请

代码 6-6

```
#define PAGE_AVAILABLE      0x00  
#define PAGE_DIRTY         0x01  
#define PAGE_PROTECT        0x02  
#define PAGE_BUDDY_BUSY     0x04  
#define PAGE_IN_CACHE       0x08
```

```

#define BUDDY_END(x,order)          ((x)+(1<<(order))-1)
#define NEXT_BUDDY_START(x,order)   ((x)+(1<<(order)))
#define PREV_BUDDY_START(x,order)   ((x)-(1<<(order)))

struct page *get_pages_from_list(int order){
    unsigned int vaddr;
    int neworder=order;
    struct page *pg,*ret;
    struct list_head *tlst,*tlst1;
    for(;neworder<MAX_BUDDY_PAGE_NUM;neworder++){
        if(list_empty(&page_buddy[neworder])){
            continue;
        }else{
            pg=list_entry(page_buddy[neworder].next,struct
page,list);
            tlst=&(BUDDY_END(pg,neworder)->list);
            tlst->next->prev=&page_buddy[neworder];
            page_buddy[neworder].next=tlst->next;
            goto OUT_OK;
        }
    }
    return NULL;
}

OUT_OK:
    for(neworder--;neworder>=order;neworder--){
        tlst1=&(BUDDY_END(pg,neworder)->list);
        tlst=&(pg->list);
        pg=NEXT_BUDDY_START(pg,neworder);
        list_entry(tlst,struct page,list)->order=neworder;
        list_add_chain_tail(tlst,tlst1,&page_buddy[neworder]);
    }
    pg->flags|=PAGE_BUDDY_BUSY;
    pg->order=order;
    return pg;
}

```

代码 6-6 其实就是 buddy 分配函数的完整实现。在代码中，首先要经过从用户请求的 order 数到 MAX_BUDDY_PAGE_NUM 数的一个循环。在循环里，如果通过 list_empty 检查到 page_buddy[n] 不为空，那就表示恰好有合适的 buddy 供分配，于是就可以采用与代码 6-5 一样的方法进行分配了。



但如果 `page_buddy[n]` 为空，那就必须循环向上搜索，直到找到一个非空的 `page_buddy` 链表头为止，然后，还是使用相同的方法将整个 `buddy` 从 `page_buddy[n]` 链表中拆下来。

这个时候，因为系统分配给用户的 `buddy` 阶数大于用户请求的 `buddy` 阶数。根据伙伴算法的描述，我们需要将多余的内存空间作为空闲的 `buddy` 挂到相应的 `page_buddy` 链表上。在代码 6-6 中，标签 `OUT_OK` 之后的 `for` 循环就实现了这个功能。

在这里，程序从实际分配给用户的 `buddy` 阶数减去 1 作为循环的开始，直到递减到用户请求的 `buddy` 阶数时就终止循环，在循环中实现 `buddy` 的拆分。例如，用户申请了一个阶数为 3 的 `buddy`，但系统只能分配一个阶数为 5 的 `buddy`。那么程序就需要从 4 开始循环，到 3 终止。这样，程序先拿出一个阶数为 4 的 `buddy`，再拿出一个阶数为 3 的 `buddy`，分别挂到相应的链表中。余下的阶数为 3 的 `buddy` 正好可以分配给用户。

在链表挂载的过程中，因为已知 `buddy` 的起始 `struct page` 结构，再通过 `BUDDY_END` 宏找到 `buddy` 中最后一个 `struct page`，就可以调用 `list_add_chain_tail` 把 `buddy` 挂到新的链表中去了。同时也要将新 `buddy` 的 `order` 数修改为相应的值。

最后，将得到的代表该 `buddy` 的 `struct page` 结构体中 `flags` 成员设置为 `PAGE_BUDDY_BUSY`，表示整个 `buddy` 正在被使用。

至此，伙伴算法中 `buddy` 的分配部分就完成了。

`buddy` 的释放正好与分配过程相反。这里我们也可以分两种情况来讨论。第一种情况是，与要释放的 `buddy` 前后相连的内存空间不能被合并，这种情况下，我们只需要在计算出 `buddy` 首尾的 `struct page` 结构体后，调用 `list_add_chain` 函数将其挂载到相应链表中即可。另外一种情况是，如果与要释放的 `buddy` 相连的内存空间可以合并，那就需要首先进行检查，然后再调用 `list_add_chain` 函数。其代码实现如下：

代码 6-7

```
void put_pages_to_list ( struct page *pg,int order ) {  
    struct page *tprev,*tnext;  
    if ( !( pg->flags&PAGE_BUDDY_BUSY ) ) {  
        return;  
    }  
}
```

```

pg->flags&=~( PAGE_BUDDY_BUSY );
for ( ;order<MAX_BUDDY_PAGE_NUM;order++ ) {
    tnext=NEXT_BUDDY_START ( pg,order );
    tprev=PREV_BUDDY_START ( pg,order );
    if (( ! ( tnext->flags&PAGE_BUDDY_BUSY )) \
        && ( tnext->order==order )) {
        pg->order++;
        tnext->order=-1;
        list_remove_chain ( &( tnext->list ), \
            &( BUDDY_END ( tnext,order )->list ) );
        BUDDY_END ( pg,order )->list.next=&( tnext->list );
        tnext->list.prev=&( BUDDY_END ( pg,order )->list );
        continue;
    }else if (( ! ( tprev->flags&PAGE_BUDDY_BUSY )) \
        && ( tprev->order==order )) {
        pg->order=-1;
        list_remove_chain ( &( pg->list ), \
            &( BUDDY_END ( pg,order )->list ) );
        BUDDY_END ( tprev,order )->list.next=&( pg->list );
        pg->list.prev=&( BUDDY_END ( tprev,order )->list );
        pg=tprev;
        pg->order++;
        continue;
    }else{
        break;
    }
}

list_add_chain( &( pg->list ), &( ( tnext-1 )->list ), &page_buddy[order] );
}

```

程序首先通过 PREV_BUDDY_START 和 NEXT_BUDDY_START 这两个宏，得到与要释放的 buddy 相邻的前后两个 buddy 的起始 struct page 地址。然后判断这两个 struct page 是否设置了 PAGE_BUDDY_BUSY，同时还要判断 order 是否与要释放的 buddy 的 order 值相等。如果两个条件都具备，那么就调用 list_remove_chain，将这个 buddy 从原来的链表中拆下来，再修改链表指针将两个 buddy 连起来并更新 buddy 的 order 值。

put_pages_to_list 和 get_pages_to_list 两个函数都是底层函数，作为用户应用的直接函数接口，其实并不成熟。原因很简单，用户在申请内存时，希望得到的应该是内存首地址，而并不是所谓的 struct page 结构体。同时，当

一个 buddy 被成功申请下来，怎样去管理 buddy 中的每一个“页”，这两个函数都没能提供一个好的解决方法。

于是，针对上述问题，我们封装了这两个函数，提供给用户一组基本可用的内存分配功能。

代码 6-8

```
struct page *virt_to_page ( unsigned int addr ) {
    unsigned int i;
    i= ((addr)-KERNEL_PAGING_START) >> PAGE_SHIFT;
    if ( i>KERNEL_PAGE_NUM )
        return NULL;
    return ( struct page * ) KERNEL_PAGE_START+i;
}

void *page_address ( struct page *pg ) {
    return ( void * )( pg->vaddr );
}

struct page *alloc_pages ( unsigned int flag,int order ) {
    struct page *pg;
    int i;
    pg=get_pages_from_list ( order );
    if ( pg==NULL )
        return NULL;
    for ( i=0;i<( 1<<order ) ;i++ ) {
        ( pg+i ) ->flags|=PAGE_DIRTY;
    }
    return pg;
}

void free_pages ( struct page *pg,int order ) {
    int i;
    for ( i=0;i<( 1<<order ) ;i++ ) {
        ( pg+i ) ->flags&=~PAGE_DIRTY;
    }
    put_pages_to_list ( pg,order );
}

void *get_free_pages ( unsigned int flag,int order ) {
    struct page * page;
    page = alloc_pages ( flag, order );
```

```

if (!page)
    return NULL;
return page_address(page);
}

void put_free_pages(void *addr, int order) {
    free_pages(virt_to_page((unsigned int)addr), order);
}

```

`alloc_pages` 首先调用 `get_pages_from_list` 进行“页”的分配，然后将分配得到的每一个 `struct page` 结构体的 `flags` 成员设置为 `PAGE_DIRTY`。这就表示程序可以利用该标志来判断每一个页的使用情况。

`get_free_pages` 函数在调用了 `alloc_pages` 函数之后，通过 `page_address` 将 `struct page` 结构体转换成已分配的内存块首地址，因此更有益于用户使用。

这里有一点要稍加说明，`get_free_pages` 和 `alloc_pages` 这两个函数都有一个 `flag` 参数。这个参数在这段例子代码中并没有什么作用，读者完全可以忽略它，但在实际应用中，`flag` 标志可以帮助我们对每一个要分配的“页”进行更高级的控制，如让某个“页”处在被保护的状态、禁止其他进程访问，等等。于是在代码 6-8 中，该标志便被保留了下来，以方便读者利用这段代码进行扩展。

与内存分配相对应，`free_pages` 和 `put_free_pages` 这两个函数分别在不同的层次中负责内存的释放工作。其中也调用了 `virt_to_page` 函数进行内存地址和 `struct page` 结构体之间的转换。具体的实现非常简单，我们就不再进行讨论了。

至此，在我们的操作系统中，一套基于伙伴算法的内存管理机制就正式建立起来了。让我们来运行这段代码，享受一下成功的喜悦吧！

6.1.2.6 算法的运行

要运行这段程序，首先需要在原来代码的基础上，将代码 6-8、6-7、6-6、6-4 的内容保存到名为“mem.c”的文件中。

然后修改 Makfile 中 OBJS 的内容为如下形式：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
interrupt.o mem.o
```

接下来修改“boot.c”中的 plat_boot 函数，以便能调用内存分配的测试程序来检验一下伙伴算法的正确性。

代码 6-9

```
void plat_boot( void ) {
    int i;
    for( i=0; init[i]; i++ ) {
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test	printk();
    //timer_init();

    init_page_map();
    char *p1,*p2,*p3,*p4;
    p1=(char *)get_free_pages(0,6);
    printk( "the return address of get_free_pages %x\n",p1 );
    p2=(char *)get_free_pages(0,6);
    printk( "the return address of get_free_pages %x\n",p2 );
    put_free_pages(p2,6);
    put_free_pages(p1,6);
    p3=(char *)get_free_pages(0,7);
    printk( "the return address of get_free_pages %x\n",p3 );
    p4=(char *)get_free_pages(0,7);
    printk( "the return address of get_free_pages %x\n",p4 );
    while(1);
}
```

为了使结果更加清楚，我们禁用了 timer_init 函数，以避免中断持续产生，影响输出结果。在 init_page_map 函数之后，程序调用了两个 get_free_pages 函数，分配两个 order 为 6 的 buddy，返回它们的内存首地址。然后，又连续调用两个 put_free_pages 函数将其释放。

此时，由于两个连续的 buddy 都处于空闲的状态，系统会将二者合并。紧接着，程序调用 put_free_pages 函数再次分配两个 order 为 7 的 buddy。打印出所有被分配内存的地址值供检验。

编译运行这些代码，运行结果如下：

```
arch: arm
```

```

cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init_addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM ./le eos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printf
test string ::| this is %s test
test char ::| H
test digit ::| -256
test X ::| 0xffffffff00
test unsigned ::| 4294967040
test zero ::| 0
the return address of get_free_pages 0x301b0000
the return address of get_free_pages 0x30170000
the return address of get_free_pages 0x300f0000
the return address of get_free_pages 0x30170000

```

从上面的结果中我们可以看出内存被反复申请和释放的一系列过程。相信读者此时已经领略到了伙伴算法的奥义了。

6.2 slab

优秀的内存管理方法对于一个操作系统来说是非常重要的。伙伴算法虽然解决了内存分配的外部碎片问题，但对于一个功能强大的操作系统来说，这还远远不够。正像我们前面提到过的，用户应用程序对内存的需求是频繁的和任意的。而伙伴算法作为一个基础内存管理算法，并不具备提供这种任意性的条件。

因此，我们还需要以伙伴算法为基础，实现另外的内存管理机制，为用户提供申请任意大小内存的可能。出于这样的目的，我们也有必要在这里重点介绍一下 slab 这个概念。

熟悉 Linux 内核的朋友一定会知道 slab。通俗地讲，slab 就是专门为某



一模块预先一次性申请一定数量的内存备用，当这个模块想要使用内存的时候，就不再需要从系统中分配内存了（因为从系统中申请内存的时间开销相对来说比较大），而是直接从预申请的内存中拿出一部分来使用，这样就提高了这个模块的内存申请速度。

6.2.1 使用 slab 的前提条件

这听起来似乎很不错，但 slab 也并不是万能的，要在合适的场合下使用它才能发挥作用。使用 slab 通常需要以下两个条件。

第一个条件是，当某一子系统需要频繁地申请和释放内存时，使用 slab 才会合理一些。如果某段程序申请和释放内存的频率不高，就没有必要预先申请一块很大的内存备用，然后再从这段私有空间中分配内存了。因为这样就意味着系统将会一次性损失过多内存，而由于内存请求的频率不高，也不会对系统性能有多大的提升。所以，对于频繁使用内存的程序来说，使用 slab 才有意义。

使用 slab 的另外一个条件是，利用 slab 申请的内存必须是大小固定的。只有固定内存大小才有可能实现内存的高速申请和释放。

使用 slab 的一个典型的例子是进程结构体。到目前为止，本书还没有涉及有关进程的问题。但可想而知，每一个进程都需要一个专门的结构体来描述它。每当一个新的进程产生的时候，系统都需要为该进程分配一段内存空间，用来存储该结构体。可想而知，这个结构体的分配具备了 slab 的两个条件，即第一，因为进程的产生和终止是一个频繁的过程，因此内存的申请和释放的频率就非常高，第二，因为结构体的大小是固定的，所以待分配的内存空间就固定了。因此，很多操作系统都将 slab 这种机制应用于进程结构体的分配中。

这里我们要强调一下，slab 这种机制并不是源于 Linux 的。Linux 所使用的 slab 分配器的基础是 Jeff Bonwick 为 SunOS 操作系统引入的一种算法。Jeff 的分配器是围绕对象缓存进行的。在内核中会为有限的对象分配大量内存，如文件描述符、进程结构体，等等。Jeff 发现对内核中普通对象进行初始化所需的时间超过了对其进行分配和释放所需的时间，因此他的结论是，不应该将内存释放回一个全局的内存池，而应保存在一个私有的内



存中。正是由于他的这种思想，最终产生了 slab 这种内存管理机制。在 Linux 中，除了 slab 之外，还有所谓的 slob 和 slub，这些内存分配机制名字不同，内部实现也大相径庭，不过它们的工作原理和根本目的都是一致的。这些不同的分配器各有优劣，分别适合于不同的应用场合。读者感兴趣的话，也可以查阅相关的文章，做进一步的了解。

既然对于一个操作系统来说，slab 机制如此地重要。那么接下来，不妨也在我们自己的操作系统中实现一个 slab。

6.2.2 slab 的组成

首先，我们要搞清楚 slab 的组成问题。从本质上讲，slab 也是一段内存空间，只不过这段内存空间只能包含那些阶数相同的 buddy，如图 6-7 所示。

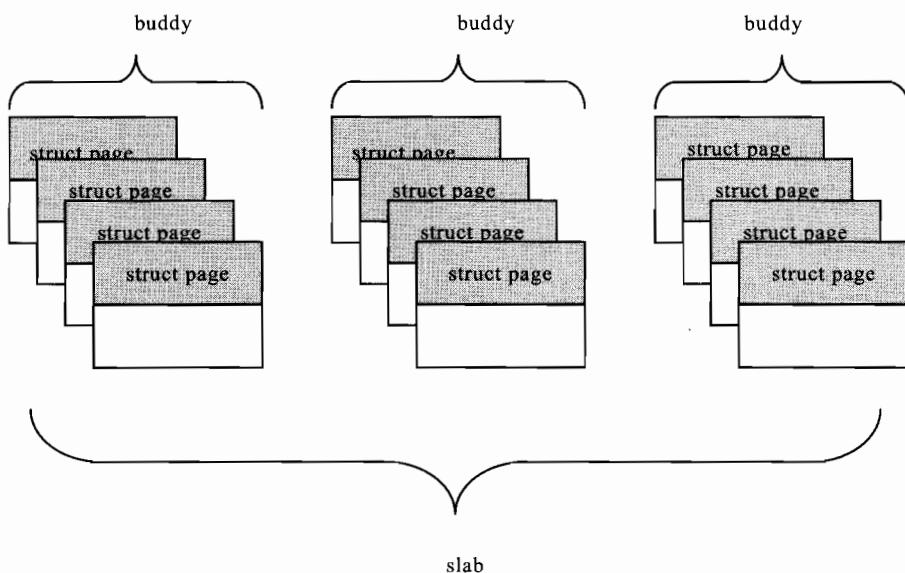


图 6-7 slab 的组成结构

下面的代码 6-10 定义了 `kmem_cache` 结构体，这是用来描述 slab 的一个基本结构体。此处我们取了一个与 Linux 内核相同的名字。当然，叫什么名字其实无关紧要，但这样做有一个好处，以后读者在接触 Linux 内核时，至少可以对其中的 `kmem_cache` 结构体有一个感性的认识。

代码 6-10

```
struct kmem_cache{
    unsigned int obj_size;
    unsigned int obj_nr;
    unsigned int page_order;
    unsigned int flags;
    struct page *head_page;
    struct page *end_page;
    void *nf_block;
};
```

在 `kmem_cache` 诸多成员中, `obj_size` 代表了该 slab 中子内存块的大小; `obj_nr` 表示 slab 中子内存块的数目; 因为 slab 需要由一组 `order` 相同的 buddy 组成, 那么这个 `order` 值就可以存储在 `page_order` 成员中; `flags` 成员用于对该 slab 进行控制, 为了简化程序, 这里我们同样省略掉对 `flags` 的操作; `head_page` 和 `end_page` 两个成员作为 `struct page` 类型的指针, 指向了整个 slab 内存中开始和结束的那两个“页”所对应的 `struct page` 结构体; `nf_block` 成员最重要, 它记录了该 slab 中下一个可用的子内存块的首地址。因此, 当我们需要从 slab 中分配一个内存块时, 直接返回 `nf_block` 即可。

也是因为同样的原因, 我们也将采用与 Linux 相似的 slab 函数接口来操作 slab。

于是在使用 slab 进行内存分配时, 就要首先调用 `kmem_cache_create` 函数来初始化一个 `kmem_cache` 结构体, 然后再调用 `kmem_cache_alloc` 函数进行内存的分配, 而释放已分配的内存则要依靠 `kmem_cache_free` 函数, 当我们不再需要 slab 时, 调用 `kmem_cache_destroy` 将这个 slab 内存区域彻底释放。于是, 使用 slab 进行内存分配的步骤如图 6-8 所示。

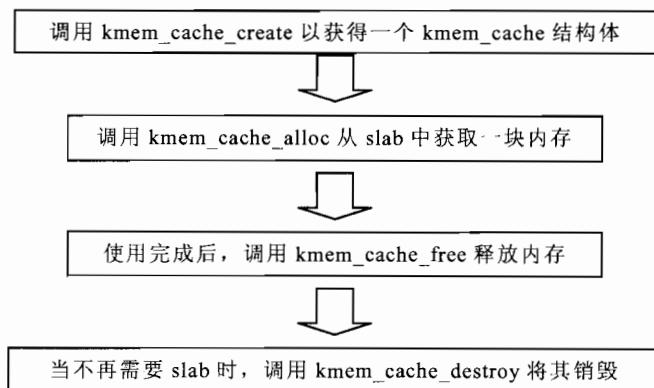


图 6-8 slab 使用流程

`kmem_cache_create` 函数能够初始化一个 `struct kmem_cache` 结构体。从代码 6-11 中我们可以看到函数 `kmem_cache_create` 共有三个参数, 其中一个是 `struct kmem_cache` 结构体, 另外两个是 slab 中子内存块的大小和控制标志。在该函数中, 程序首先定义了一个指向 `kmem_cache` 中 `nf_block` 成员的指针 `nf_blk`, 因为该成员也是指针, 所以 `nf_blk` 其实是一个二级指针。然后, `find_right_order` 函数根据 slab 中子内存块的大小分析出究竟该 slab 需要阶数为几的 buddy, 并把该阶数返回。

代码 6-11

```
#define KMEM_CACHE_DEFAULT_ORDER (0)
#define KMEM_CACHE_MAX_ORDER      (5)
#define KMEM_CACHE_SAVE_RATE      (0x5a)
#define KMEM_CACHE_PERCENT        (0x64)
#define KMEM_CACHE_MAX_WAST      (PAGE_SIZE-\n                                KMEM_CACHE_SAVE_RATE*\n                                PAGE_SIZE/KMEM_CACHE_PERCENT)

int find_right_order(unsigned int size){
    int order;
    for(order=0;order<=KMEM_CACHE_MAX_ORDER;order++){
        if(size<=(KMEM_CACHE_MAX_WAST)*(1<<order)){
            return order;
        }
    }
    if(size>(1<<order)){
        return -1;
    }
    return order;
}

int kmem_cache_line_object(void *head,unsigned int size,int
order){
    void **pl;
    char *p;
    pl=(void **)head;
    p=(char *)head+size;
    int i,s=PAGE_SIZE*(1<<order);
    for(i=0;s>size;i++,s-=size){
        *pl=(void *)p;
        pl=(void **)p;
        p=p+size;
    }
}
```



```
    }

    if(s==size)
        i++;
    return i;
}

struct kmem_cache *kmem_cache_create(struct kmem_cache *cache,\nunsigned int size,unsigned int flags){\n    void **nf_blk=&(cache->nf_block);\n    int order=find_right_order(size);\n    if(order==-1)\n        return NULL;\n    if((cache->head_page=alloc_pages(0,order))==NULL)\n        return NULL;\n    *nf_blk=page_address(cache->head_page);\n\n    cache->obj_nr=kmem_cache_line_object(*nf_blk,size,order);\n    cache->obj_size=size;\n    cache->page_order=order;\n    cache->flags=flags;\n    cache->end_page=BUDDY_END(cache->head_page,order);\n    cache->end_page->list.next=NULL;\n\n    return cache;\n}
```

find_right_order 函数就是在从 0 开始到 KMEM_CACHE_MAX_ORDER 的范围内查找当 order 等于多少的时候，能够容纳指定个数的子内存块。这里使用的是依照百分比的计算方法查看一个 size 是否占整个 buddy 的百分比小于某个特定值，然后把那个值返回。找到了正确的 order 值，就可以调用 alloc_pages 分配内存了。alloc_pages 函数返回的是 struct page 结构体的指针。于是我们将该指针赋值给 cache->head_page，为以后对 slab 的控制打下基础。同时，通过 page_address 获取 slab 内存首地址的指针赋值给 nf_blk 指针指向的内容，简单地说，就是赋值给了 cache->nf_block 这个变量。在一切准备就绪后，程序调用了 kmem_cache_line_object 初始化已分配的内存区域，然后在对 cache 的其他成员进行必要的初始化之后将指向 cache 的指针返回。

函数 kmem_cache_line_object 的主要目的就是将已分配好的 slab 内存连

接成如图 6-9 所示的样子。将 slab 内存分配成若干子块，每一个子块的大小由 kmem_cache 结构体的 obj_size 成员决定。子块分配完成后，再让每一个子块最开始的内存中存储下一个子块的起始地址。最后一个子块则存储 NULL 值，表示 slab 中已经没有子块可以提供使用。

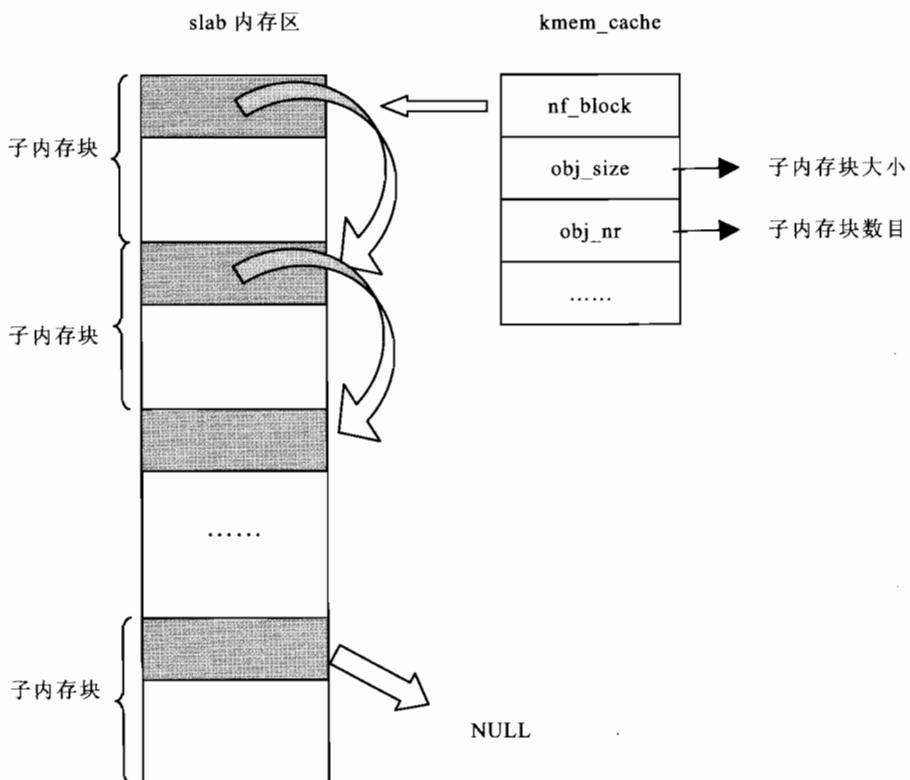


图 6-9 slab 的初始化

在初始化的时候，因为整个 slab 内存区都是空闲的，因此我们可以让 slab 中前后子块之间连续分布，一旦 slab 空间参与了分配，那么各子块之间就不会连续了，不过连续与否并不影响 slab 的正常使用。当然，一个 slab 内存区域可能并不能容纳整数倍的子内存块，余下最后一点内存区域，因为不足整个子块大小，就被浪费掉了。

6.2.3 通过 slab 进行内存分配

一旦 slab 内存区域初始化完成，我们便可以调用 slab 的分配函数从 slab 中获取内存了。

代码 6-12

```
void *kmem_cache_alloc ( struct kmem_cache *cache, unsigned int flag ) {
    void *p;
    struct page *pg;
    if ( cache==NULL )
        return NULL;
    void **nf_block=& ( cache->nf_block );
    unsigned int *nr=& ( cache->obj_nr );
    int order=cache->page_order;

    if ( !*nr ) {
        if ( ( pg=alloc_pages ( 0,order ) ) ==NULL )
            return NULL;
        *nf_block=page_address ( pg );
        cache->end_page->list.next=&pg->list;
        cache->end_page=BUDDY_END ( pg,order );
        cache->end_page->list.next=NULL;
        *nr+=kmem_cache_line_object (*nf_block,cache->obj_size,order);
    }

    (*nr)--;
    p=*nf_block;
    *nf_block=*( void ** ) p;
    pg=virt_to_page ( ( unsigned int ) p );
    pg->cachep=cache;
    return p;
}
```

函数 kmem_cache_alloc 能够从 slab 内存区中分配一个子内存块，并将这个子内存块的首地址返回。这里，我们也需要分两种情况来看。

第一种情况是，当 slab 中含有可分配的子块。此时，内存的分配相当简单，只需要将 kmem_cache 结构体的 nf_block 成员指向下一个空闲子内存块即可，这可以通过读取 nf_block 的内容来实现，将 obj_nr 成员的值自减 1，

然后将原来的 nf_block 值返回。

另一种情况略复杂一些。当经过了一段时间的分配和释放后，`kmem_cache` 的 `obj_nr` 成员为 0，`slab` 中已经没有可供分配的子内存块了。此时我们必须重新调用 `alloc_page` 函数分配一组新 `buddy`，同时调整 `kmem_cache` 相关成员，如 `end_page` 和 `obj_nr` 等。

`kmem_cache_alloc` 函数完整的实现方法就如代码 6-12 那样，需要注意的是函数结尾的部分有一行赋值语句：

```
pg->cachep=cache
```

这为 `slab` 子内存块的释放和进一步处理打下了基础。因为这样一来就可以确保每个 `struct page` 结构只属于一个 `kmem_cache`，同时，我们既可以通过虚拟地址找到所属的 `struct page` 结构，又可以通过 `struct page` 结构找到对应的 `kmem_cache` 结构体，从而保证了各个内存管理结构之间的相互转化。

6.2.4 内存空间的释放

相对于 `slab` 分配函数而言，`slab` 子内存块的释放函数就相对简单了。在这里，为使程序更加简化，我们并不承担内存回收的重任。也就是说，当内存不足的时候，我们不负责扫描空闲的内存块并将其释放，而是简简单单地返回 `NULL`。这样，无论 `slab` 占用多少内存，释放 `slab` 内存区这个动作只会发生在 `slab` 被销毁的时候。因此，对 `slab` 子内存块的释放不需要几行代码就可以实现了。

代码 6-13

```
void kmem_cache_free ( struct kmem_cache *cache, void *objp ) {
    * ( void ** ) objp = cache->nf_block;
    cache->nf_block = objp;
    cache->obj_nr++;
}
```

释放 `slab` 子内存块，无非就是更新 `kmem_cache` 结构体的 `nf_block` 成员，让它指向新被释放的内存地址，同时更新该地址的内容，指向原有的第一个子内存块首地址，这样，下次内存申请的时候，就可以将这个地址分配出去了。

6.2.5 slab 的销毁

最后，当这个程序不再需要 slab 进行内存分配时，就可以调用 `kmem_cache_destroy` 函数将这个内存空间销毁。而这里所谓的销毁，无非就是将组成 slab 内存区的各个小 buddy 依次释放掉而已，实现过程如下：

代码 6-14

```
void kmem_cache_destroy ( struct kmem_cache *cache ) {
    int order=cache->page_order;
    struct page *pg=cache->head_page;
    struct list_head *list;
    while (1) {
        list=BUDDY_END ( pg,order ) ->list.next;
        free_pages ( pg,order );
        if ( list ) {
            pg=list_entry ( list,struct page,list );
        } else {
            return;
        }
    }
}
```

看一下代码 6-14，在得到一个指向 `struct page` 结构体的指针和所属 buddy 的 order 值之后，就可以调用 `free_pages` 函数将这一个 buddy 释放。当然，此时函数还不能退出，因为一个 slab 内存区可能包含了多个相同阶数的 buddy。所以，程序需要通过循环来依次释放这些 buddy 直到结束。

至此，我们也在自己的操作系统中实现了一个简单的 slab 管理层。从结构上看，slab 的这种实现是能够处理内存高速请求的问题的。但这并不表示这段代码能够应付所有操作系统可能遇到的极端环境。

对于一些简单的操作系统来说，这段代码已经足够使用了。但如果我们的目的是设计出一套精简高效的通用操作系统，那么一些极端的情况也是必须要考虑进去的，其中之一就是内存的耗尽问题。一些典型的嵌入式操作系统，如 UCOS 等，因为应用程序和操作系统内核是一体的，所以程序和系统之间彼此完全了解。在这样的系统中，应用程序会充分了解系统内存的大小和使用情况，从而在设计时就避免了内存的过渡使用。然而，对于功能强

大的通用操作系统来说，应用程序和操作系统内核之间是完全分离的。应用程序并不充分了解系统的状况，也不太可能保证不去过度消耗内存。这样一来，系统内存耗尽时要如何应对就成了通用操作系统必须要考虑的问题。

想要解决内存耗尽问题其实也不是难事。当系统中可用内存耗尽时，只需要去扫描已用的内存，看看是否有一些内存虽然被申请了但却没有被使用。

毫无疑问，slab 应该是首当其冲的。对本章内容理解相对透彻的朋友应该清楚，slab 就是预先申请一大块内存，然后留给某段私有程序慢慢用。因此，slab 中必然包含了许多空闲内存，这些内存不能被其他应用程序使用，同时也暂时没有被 slab 的拥有者所利用。

因此，当操作系统发现系统剩余内存不足以分配给某段程序时，就应该要求各个 slab “提前还款”了。不过，由于这部分内容已经违背了本书的编写目的，因此我们就不再详细讲述了。毕竟即使没有内存回收策略，我们的操作系统结构仍然是完整了，应用程序依然可以正常运行的。

slab 究竟有多重要，想一想一个操作系统中，究竟有多少类似于文件描述符这样的结构，就应该很清楚了。slab 仍然有它自己的适用范围，毕竟我们不能要求程序去适应操作系统，而应该想尽一切办法让操作系统尽可能满足程序的所有要求。

然而，读者朋友们也许在读完本节之后仍然没有掌握解决任意内存申请的方法。别急，slab 正是实现这种方法的一个基本手段。接下来的一节，我们将以此为基础，在操作系统中实现一个能够申请任意内存的函数。借鉴 Linux 的命名规则，这里我们也将这个函数命名为 kmalloc。

6.3 kmalloc 函数

kmalloc 函数应该怎样实现呢？

试想，我们预先定义若干个 kmem_cache 结构体，用来代表不同大小子内存块的 slab。这样，当某一个应用程序想要申请一段内存空间时，只需要就近找到一个合适的 slab 并从中分配一块内存即可。



例如，我们预先申请一组子内存块大小为 32 字节~128K 的 slab，每个 slab 之间能够管理的子内存块大小间隔 32 字节。当某个程序想申请 78 字节的内存，就可以从子内存块为 96B 的 slab 中拿出一块内存分配给它，其他大小亦是如此。这样，理论上每一次申请所浪费的内存数最多是 32 字节，通常这是可以接受的。如果想减少这种内存的浪费，可以将不同内存块的步长设置成 16 字节，甚至是 8 字节。

这些都是 slab 的典型应用，让我们一起来看一下这段代码：

代码 6-15

```
#define KMALLOC_BIAS_SHIFT      (5)
#define KMALLOC_MAX_SIZE        (4096)
#define KMALLOC_MINIMAL_SIZE_BIAS \
    (1<<(KMALLOC_BIAS_SHIFT))
#define KMALLOC_CACHE_SIZE \
    (KMALLOC_MAX_SIZE/KMALLOC_MINIMAL_SIZE_BIAS)
struct kmem_cache kmalloc_cache[KMALLOC_CACHE_SIZE]= \
    {{0,0,0,0,NULL,NULL,NULL},};

#define kmalloc_cache_size_to_index(size) \
    (((size))>>(KMALLOC_BIAS_SHIFT))

init_kmalloc_init(void) {
    int i;
    for (i=0;i<KMALLOC_CACHE_SIZE;i++) {
        if (kmem_cache_create(&kmalloc_cache[i], \
            (i+1)*KMALLOC_MINIMAL_SIZE_BIAS,0)==NULL)
            return -1;
    }
    return 0;
}

void *kmalloc(unsigned int size) {
    int index=kmalloc_cache_size_to_index(size);
    if (index>=KMALLOC_CACHE_SIZE)
        return NULL;
    return kmem_cache_alloc(&kmalloc_cache[index],0);
}

void kfree(void *addr) {
    struct page *pg;
    pg=virt_to_page((unsigned int)addr);
```

```

    kmem_cache_free( pg->cachep, addr ) ;
}
}

```

首先在代码 6-15 中, 我们定义了一个全局的 `kmem_cache` 类型结构体数组, 名为 `kmalloc_cache`。该数组的大小为 `KMALLOC_CACHE_SIZE`。当然根据不同的应用情况, 这个值是可以调整的。

在其他模块使用 `kmalloc` 函数分配内存之前, 必须首先调用 `kmalloc_init` 函数将 `kmalloc_chache` 数组中每一个成员都初始化。从程序中我们可以看出, `kmalloc_init` 函数只不过是循环调用了 `kmem_cache_create` 函数而已。

在初始化完成之后, 就可以使用 `kmalloc` 函数了, 而 `kmalloc` 的参数只有一个, 那就是想要获取的内存的大小。`kmalloc` 函数无非是调用 `kmem_cache_alloc` 从相应的 `kmem_cache` 结构体中分配内存而已。当然在调用 `kmem_cache_alloc` 函数之前, 我们需要首先得到一个 `kmalloc_cache` 数组的索引。这很简单, 只需要调用 `kmalloc_cache_size_to_index` 这个宏就可以了。

当 `kmalloc` 申请的内存不再使用后, `kfree` 函数就可以将它释放。`kfree` 函数的实现就更加简单了, 首先从虚拟地址得到与之对应的 `struct page` 结构体, 这样就可以通过 `struct page` 结构体的 `cachep` 成员得到所属的 `kmem_cache` 结构, 然后以这个结构体作为参数调用 `kmem_cache_free` 释放内存。

最后, 我们来运行一下 slab 和 `kmalloc` 的代码。

首先我们需要将代码 6-10 之后的全部代码保存到文件 “mem.c” 中。然后为了验证这段代码的正确性, 我们需要修改一下 “boot.c” 的内容。

代码 6-16

```

void *kmalloc( unsigned int size );
void plat_boot( void ) {
    int i;
    for ( i=0; init[i]; i++ ) {
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printf();
//    timer_init();
    init_page_map();
}

```

```
kmalloc_init();
char *p1,*p2,*p3,*p4;
p1=kmalloc(127);
printf("the first allocoed address is %x\n",p1);
p2=kmalloc(124);
printf("the second allocoed address is %x\n",p2);
kfree(p1);
kfree(p2);
p3=kmalloc(119);
printf("the third allocoed address is %x\n",p3);
p4=kmalloc(512);
printf("the forth allocoed address is %x\n",p4);
while(1);
}
```

这里可以随意选取几个值，使用 kmalloc 来分配内存空间并调用 kfree 将某些空间释放。读者可以比较一下最终的运行结果，根据各函数的运行原理查看一下是否正确。另外为了避免编译警告，代码 6-16 中也对 kmalloc 函数进行了声明。

最终的运行结果如下所示：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printf
test string :::: this is %s test
test char :::: H
test digit :::: -256
test X :::: 0xffffffff00
test unsigned :::: 4294967040
test zero :::: 0
the first allocoed address is 0x306f3000
the second allocoed address is 0x306f3080
the third allocoed address is 0x306f3080
the forth allocoed address is 0x301e0000
```

6.4 总结

一个平台下的内存容量越大、数据访问速度越快，那么这台计算机就越“聪明”。但是很多时候我们不能向计算机提出要求，就好像一个人天生愚钝，但是却没有能力改变这种既成事实，于是唯一的希望就寄托在后天的努力上。可想而知，操作系统对于内存的处理方法就成为了计算机改变自己命运的唯一手段。从这个角度看，无论我们对内存有多么重视，都是不过分的。

再回到技术细节中，纵观本章的内容，我们实现了不同层次，不同应用的几种内存管理方法。图 6-10 总结了这些内存管理算法各层次之间的关系。

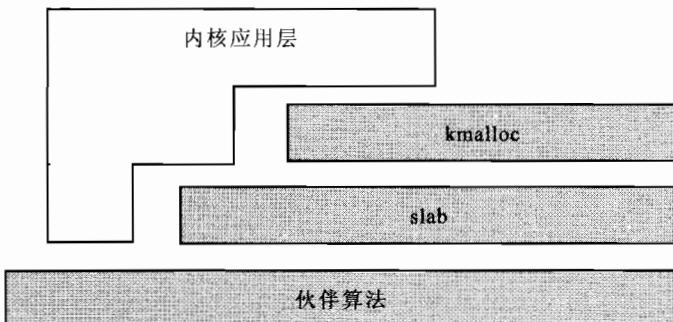
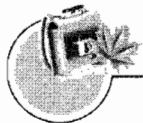


图 6-10 操作系统中的内存层次结构

如图 6-10 中所示的那样，`kmalloc` 函数是建立在 `slab` 的基础上的，`slab` 又是以伙伴算法为基础的，从而构成了一个全方位、立体的内存管理机制。而对于存在于操作系统内核中的其他程序，这些机制也都是公开的，所以，程序可以根据实际情况选择合适的方法获取内存，从而大大增强了内核程序的灵活性和适应能力。

当然在本章中，我们只是给出了一套自己的实现方法。程序是人写的，代码的实现方法也并不唯一。只要背后的算法原理没有变化，不同程序的实现方面没有太多优劣之分。因此，本章的代码也仅仅代表了一种参考方法，能够为读者抛砖引玉足矣。



第7章

框 架



本章主要讨论的是有关驱动程序和文件系统的问题。

标题之所以这样取，是因为本章的侧重点不是去实现某一种硬件的驱动程序或支持某一个优秀的文件系统，而是从一个更抽象的角度去实现两个框架。任何驱动程序和文件系统都可以依据这种框架去实现它们的功能。这样，操作系统在与驱动程序和文件系统交互时，不再需要了解具体的驱动代码或文件系统结构，而只需要调用框架中的相关函数，从而实现了代码结构的抽象。同时，由于驱动程序五花八门、文件系统多种多样，旧的代码会被淘汰，新的功能会被添加，因此保持文件系统和驱动程序框架的相对稳定性，允许框架下的代码实时变化才是一个优秀操作系统应具备的。这与前面我们反复强调的模块化的思想如出一辙。

当然，这些经验都是前人工作的结晶。因此，许多通用的操作系统都继承着这种结构，包括一些嵌入式操作系统以及几乎所有的非嵌入式操作系统。在本章中就让我们在学习其他操作系统优秀思想的同时，实现一下属于自己的框架。

7.1 驱动程序框架

在类 UNIX 操作系统中，根据传统的驱动程序框架，驱动程序被分成两大类：字符设备和块设备（因为网络设备稍微有些特殊，这里我们避开网络设备不谈）。

字符设备是指每次数据传输时将字符作为最基本单位的设备，如键盘、声卡等。它们通常不支持随机存取数据，因此对字符设备的处理非常直观简单，字符设备在实现时也大多不需要缓存，系统直接从设备读取 / 写入每一个字符。

而块设备则是与字符设备相对的另一个概念。它是指以批量方式进行数据传输的一类设备，如硬盘、CD-ROM 等。块设备通常支持随机存取和寻址并使用缓存。通常，操作系统会为块设备的输入或输出过程分配缓存，以存储临时读出或写入的数据。当缓存被填满时，会采取适当的操作把数据传走，而后系统清空缓存。这种针对块设备的缓存机制其实是广泛存在的，目的就是使访问速度较慢的块设备，能够适当地提高读写效率。

类 UNIX 系统将设备分成这样两大类，其实是对所有外设高度抽象化的结果。任何外设都可以被归类为按照字符读写的设备，或者被归类为按照块读写的设备。这样，在类 UNIX 操作系统中，内核只需要实现一个字符设备框架和一个块设备框架，就可以管理系统中所有设备了。

以上就是对类 UNIX 操作系统中设备管理的简单介绍。然而，想要在我们的操作系统中实现自己的驱动程序框架，却不能简简单单地照搬 UNIX 系统。

虽然类 UNIX 系统实现了设备的高度抽象，却在一定程度上牺牲了系统的整体效率。拿字符设备来说，类 UNIX 操作系统的驱动程序与文件系统共用同一种处理方法，也就是说，程序员只需要通过文件操作函数接口就可以直接操作设备了。这虽然统一了程序接口，但却是以降低程序运行效率为代价的。同时，为了迎合程序框架的通用结构，许多设备独有的特性不得不由驱动程序编写者去完成，因而提高了程序编写的复杂性。

就拿 Linux 系统下的摄像头为例，根据字符设备的分类原则，摄像头应该属于字符设备，于是在编写驱动时，就要根据字符设备框架的规定编写 `file_operations` 结构体，实现 `open`、`write`、`ioctl` 等函数（`file_operations` 是实现 Linux 字符设备的一个通用结构）。这样一来，针对摄像头这类设备的特有属性，如对颜色、格式等视频属性的处理，就不得不由驱动程序自己去完成，而每一个写驱动的人又几乎都要完成一些内容相似的东西，无论从效率上还是从稳定性上来看都不理想。

这就是传统设备驱动程序框架所带来的弊端。为了弥补这种不足，现在



这些系统通常都在原有传统设备框架的基础之上封装了各种各样针对不同设备的子框架，并让这种趋势渐成主流。

例如，Linux 中目前就有专门针对摄像头等媒体设备的 v4l 框架，有专门针对声卡设备的 alsa 框架等。这些驱动程序框架都是在原有的基于文件系统的框架基础上发展而来的，同时又更深入地解决了不同设备之间的差异问题。

那么在我们自己的操作系统中，不妨总结并发展类 UNIX 操作系统针对设备驱动的处理方法，一方面可以摒弃基于文件系统的设备框架，保证了效率，另一方面又为针对设备类型的框架提供了支持。因此，我们可以将设备分为若干类，每一类设备都有一套子框架对它做支持，同时保证以直接系统调用或类似的方式提供给用户，从而确保了内核结构的清晰和简单。

这便是我们设计驱动程序框架的思路。

7.1.1 基于存储设备的实例

下面我们以存储设备为例来学习驱动程序框架的应用。之所以选择存储设备，还有另外一个目的，那就是为后续章节的实践做好铺垫。

但在真正开始之前，我们还要解决一个非常棘手的问题，那就是设备。由于我们的操作系统一直都是在虚拟机中运行的，因此就不得不利用虚拟机虚拟出一块存储设备来。

出于多种原因，我们决定选用 RAM 盘作为例子来讲解设备框架。所谓 RAM 盘，其实就是将一块内存当成是诸如硬盘之类的非易失存储设备进行操作。从程序的角度来讲，RAM 盘是最简单的存储设备了，因为数据与存储器之间的交换可以通过原生的内存操作方法来实现，不像硬盘、存储卡等存储设备需要经过复杂的执行逻辑才可以将整块数据存入或读出存储器，这样就可以将重点放在程序结构上，而不是深陷到细节当中。

然而，框架的设计不同于写代码，需要对具体实例进行高度抽象。对于任意一种设备，我们都可以将针对它的操作方法抽象为五种类型：

- 设备的初始化操作
- 设备的释放操作

- 对设备的控制操作
- 数据写向设备的操作
- 数据从设备中读出的操作

存储设备作为设备的一个子类，自然也符合这五种基本的操作方法。就拿硬盘为例，系统开始运行时对硬盘控制器的配置就属于设备的初始化操作，向硬盘中复制数据就属于数据写向设备的操作，而更改硬盘的读写模式等，就可被归类为对设备的控制操作。

有的时候，有些设备可能不需要或没必要进行初始化和释放，对于这样的设备，设备的初始化和释放操作就可以被省略掉。同样，有些设备可能是只读的或只写的，比如一些简单的传感器，只需要从中读出数据即可，不需要进行数据的写入、初始化或控制。

在完成了对设备的抽象化过程后，我们需要做的就是将这种抽象实施到代码之中。此时我们已经将设备视为一种对象，并将以面向对象的方法去实践。

代码 7-1

```
#ifndef __STORAGE_H__
#define __STORAGE_H__

#define MAX_STORAGE_DEVICE (2)
#define RAMDISK 0

typedef unsigned int size_t;
struct storage_device{
    unsigned int start_pos;
    size_t sector_size;
    size_t storage_size;
    int (*dout)(struct storage_device *sd,void *dest,\n
                unsigned int bias,size_t size);
    int (*din)(struct storage_device *sd,void *dest,\n
               unsigned int bias,size_t size);
};

extern struct storage_device *storage[MAX_STORAGE_DEVICE];
extern int register_storage_device(struct storage_device *sd,\n
                                    unsigned int num);
```



```
#endif
```

在代码 7-1 中定义了 `struct storage_device` 结构体，该结构体专门用来描述一个通用的存储设备。

`start_pos` 成员用于描述这个存储设备的绝对位置，这个成员主要是为了解决对存储设备的分区问题而专门定义的；`sector_size` 成员用于描述存储设备的最小存储块，对存储设备的读写操作必须以这个值为基本单位，对于硬盘来说，这个值就是一个扇区的字节数。正因为存储设备是以块而不是以字节作为基本存储单位的，才使得存储设备具备了对大量数据进行存储的能力。`storage_size` 代表了这一个存储设备的总容量，函数指针 `dout` 和 `din` 分别指向了数据写入设备的函数和数据从设备中读出的函数。这两个函数的参数是相同的，都包含了写入的目的位置信息、数据源位置信息、待写入数据的大小，以及一个指向 `struct storage_device` 结构体的指针。其他诸如设备的初始化或控制的函数指针，这里省略没有列出。

举例来说，一个 NOR FLASH 设备一共有 4M 存储空间，如果在该设备上分出两个分区，分别为 1M 和 3M，那么就需要同时定义两个 `struct storage_device` 结构体：第一个结构体的 `start_pos` 应该为 0，`storage_size` 成员的值为 1M；第二个结构体的 `start_pos` 值为 1M，`storage_size` 的值就应该为 3M。由于 NOR FLASH 本身可以像内存一样，是不受一次读写数据大小的限制的，因此我们可以将这两个 `struct storage_device` 结构体的 `sector_size` 成员定义成 512 字节、1K 或者 2K 等常规数值。

代码 7-1 是使用 C 语言进行面向对象编程的典型方法，C 语言并非不适合进行以对象为基本思想的编程，它只是不够严谨和直接，真正起作用的是思想而不是语言本身。

既然已经有了描述抽象设备的基本类型，接下来我们就可以针对存储设备对这个类型进行实例化。

代码 7-2

```
#include "storage.h"

#define RAMDISK_SECTOR_SIZE    512
#define RAMDISK_SECTOR_MASK   (~(RAMDISK_SECTOR_SIZE-1))
#define RAMDISK_SECTOR_OFFSET ((RAMDISK_SECTOR_SIZE-1))
```



```

extern void *memcpy(void * dest,const void *src,unsigned int
count);

int ramdisk_dout(struct storage_device *sd,void *dest,\n
                  unsigned int addr,size_t size){
    memcpy(dest,(char *) (addr+sd->start_pos),size);
    return 0;
}

struct storage_device ramdisk_storage_device={
    .dout=ramdisk_dout,
    .sector_size=RAMDISK_SECTOR_SIZE,
    .storage_size=2*1024*1024,
    .start_pos=0x40800000,
};

int ramdisk_driver_init(void){
    int ret;
    remap_11(0x30800000,0x40800000,2*1024*1024);
    ret=register_storage_device(&ramdisk_storage_device,RAMDISK);
    return ret;
}

```

代码 7-2 是一个超级简化版的 RAM 盘驱动程序。这段代码虽然简单，却足以说明驱动程序和框架之间是如何结合的。

写一个存储设备驱动程序，首先要做的是准备一个 struct storage_device 结构体。在对该结构体进行必要的初始化之后，调用 register_storage_device 将它注册到系统之中。

在代码中定义的 struct storage_device 结构体名为 ramdisk_storage_device，该结构体的 start_pos、sector_size 和 storage_size 分别被定义成了 0x40800000、512 以及 2M。这表示这一存储设备的最小读写块为 512 字节，共有 2M 大小的空间，并且其起始的绝对地址是 0x40800000。请注意，这里的地址应该是内存虚拟地址，不要忘了我们的代码早已激活了 MMU 的功能。这也意味着在内存的物理地址某处，有 2M 大小的内存空间专门被当做 RAM 盘使用，并在使用这段空间时，需要将该物理地址映射到虚拟地址 0x40800000 处，这个动作可以依靠代码 7-3 中的 remap_11 函数完成。

对于 ramdisk_storage_device 结构体来说，最关键的参数当属 dout 成员了。在代码 7-2 中，这一成员指向了函数 ramdisk_dout，其目的就是实现数据从 RAM 盘设备中的读取操作。根据代码 7-1 中的定义，该函数应有 4 个



参数，其中，`dest` 代表了数据将要被复制到的内存地址，`size` 表示复制数据的大小，可以是任意值，不必受限于存储设备的 `sector_size` 成员，而 `addr` 则代表了存储设备内部的偏移，它的取值范围为 `0~storage_size`。

我们要操作的硬件是一块 RAM 盘，因此实现数据的读取操作并不困难。函数 `ramdisk_out` 就是使用了 `memcpy` 来实现数据的读取。这个函数之前是定义在“`print.c`”文件里的，所以这里我们根本不需要实现，只需声明一下即可。

如果是其他存储设备，比如硬盘、NAND FLASH、SD 卡等呢？由于每一种存储设备都有自己的特点和操作方法，因此，`dout` 函数实现起来将会完全不同。但不管怎样，它的接口必须要符合 `storage_device` 中 `dout` 成员的定义。

代码中只实现了对存储设备的读的操作，写的操作与之类似，如果需要的话，读者朋友们可以自行完成。

当一切必要的成员都实现之后，就可以在 `ramdisk_driver_init` 函数中进行 RAM 盘设备的初始化了。

首先，我们要进行必要的内存映射，将物理地址某处的一块内存区映射到虚拟地址空间当中充当 RAM 盘进行使用。在前面几章的例子中我们都假设虚拟平台中有 8M 的内存空间，为了不改变前面的代码，保证代码的前后兼容性，本章中我们假设虚拟硬件中共有 10M 内存，其中前 8M 内存的划分方法与前面的代码相同，后 2M 内存则专门用来做 RAM 盘，这样一来，之前的程序就不需要修改了。

根据这样的假设，在我们的虚拟平台中 RAM 盘所处的物理内存地址就应该是 `0x30800000~0x30a00000`，紧接在系统原有的 8M 内存之后。此时我们就可以调用 `remap_11` 函数进行地址映射了，该函数的实现方法如下：

代码 7-3

```
void remap_11 ( unsigned int paddr,unsigned int vaddr,int size ) {  
    unsigned int pte;  
    unsigned int pte_addr;  
    for (;size>0;size-=1<<20) {  
        pte=gen_11_pte ( paddr );  
        pte_addr=gen_11_pte_addr ( L1_PTR_BASE_ADDR,vaddr );  
        *( volatile unsigned int * ) pte_addr=pte;
```



```
    }
}
```

如果读者已经完全理解了 MMU 的原理和操作方法，看这段代码就没有什么难度了。代码只是在循环里一次次地求出与物理地址以及要映射到的虚拟地址所对应的页表项和页表项的地址，再将页表项存入地址中去。

函数 `ramdisk_driver_init` 在完成了地址映射后，就可以调用 `register_storage_device` 函数将存储设备注册到系统中了。

代码 7-4

```
#include "storage.h"

struct storage_device *storage[MAX_STORAGE_DEVICE];

int register_storage_device(struct storage_device *sd,unsigned int num) {
    if (storage[num]) {
        return -1;
    } else {
        storage[num]=sd;
    }
    return 0;
};
```

代码 7-4 只是示例性地定义了一个全局 `storage_device` 类型的数组，这个数组用来存储系统中所有的 `storage_device` 结构体，因为在一个硬件平台中很多时候并不会只有一种存储设备，当系统中既包含 NOR FLASH，又包含硬盘，同时还支持 RAM 盘时，将指向这些结构体类型的指针都保存到 `storage` 数组当中是一种非常简单的选择。而 `register_storage_device` 函数无非就是将某一个 `storage_device` 结构体赋值到 `storage` 数组相应的位置上。这样，我们就必须为每一种硬件都分配一个序号，就像代码 7-1 那样，`RAMDISK` 宏代表了 RAM 盘的硬件序号。注册的时候，首先检查 `storage` 数组的 `RAMDISK` 位置是不是 `NULL`，如果是，则表示有别的 RAM 盘驱动已经被注册了，如果没有注册，则将参数指针赋值给结构体数组的 `RAMDISK` 成员。

这样我们就利用了存储设备的驱动程序框架，编写出了一个简单的 RAM 盘驱动程序。作为本书的示例代码，我们尽量保证程序的精简和直观。

不过无论简单还是复杂，框架结构始终是统一的，本质的东西并没有什么变化。

7.1.2 运行存储设备实例

运行这段代码会稍微麻烦一些。首先我们必须修改“skyeye.conf”配置文件，新添加一块内存空间作为 RAM 盘。新的“skyeye.conf”内容如下：

代码 7-5

```
cpu: arm920t  
mach: s3c2410x  
  
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,  
file=./le eos.bin, boot=yes  
mem_bank: map=M, type=RW, addr=0x30800000, size=0x00200000,  
file=./ram.img  
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

这里我们定义了一个新的空间，从 0x30800000 开始，大小为 0x200000。在分配新空间的同时，又将一个名为“ram.img”的文件加载到这段空间内。“ram.img”文件的大小可以是 2M 或小于 2M，这其实是表示“ram.img”文件中的内容恰好代表了这段内存空间的内容。在实际的操作系统中，可以通过引导加载程序将相同的内容预先复制到该内存处去实现。

一个小问题随之而来，“ram.img”文件应该如何得到呢？其实这样的方法还是有很多的，无论在 Linux 环境还是 Cygwin 环境下，我们都可以借助于 dd 命令，其运行方法如下：

```
dd if=/dev/zero of=ram.img bs=1M count=2
```

这样，就恰好可以得到大小为 2M、内容全部为 0 的文件了。这条命令的意思是将设备文件 zero 中的内容复制出两块，每一块为 1M 大小，存储为“ram.img”文件。因为 zero 设备文件的内容全都是 0，因此“ram.img”的内容就全部是 0。

文件的内容可以是 0 或者是别的。我们使用“ram.img”文件的根本目的是在程序中读出从 0x30800000 开始，大小为 0x200000 的空间的任何一段内容，这其实就是文件本身的内容。然后将读出来的部分与文件做比较，以

此证明程序运行的正确性。

在完成上述工作后，将代码 7-4 的内容保存成名为“driver.c”的文件，再将代码 7-3 的内容添加到文件“mmu.c”的末尾，然后把代码 7-2 保存成文件，名为“ramdisk.c”，最后要保存代码 7-1 的内容，将文件命名为“storage.h”。

在所有的准备工作都完成后，为了验证这段代码的正确性，我们需要修改一下“boot.c”文件中的内容。

代码 7-6

```
void plat_boot ( void ) {
    int i;
    for ( i=0; init[i]; i++ ) {
        init[i] ( );
    }
    init_sys_mmu ( );
    start_mmu ( );
    // timer_init ( );

    init_page_map ( );
    kmalloc_init ( );
    char buf[128];
    ramdisk_driver_init ( );
    storage[RAMDISK]->dout ( storage[RAMDISK],buf,0,sizeof ( buf ) );

    for ( i=0;i<sizeof ( buf );i++ ) {
        printk ( "%d ",buf[i] );
    }
    printk ( "\n " );
    while ( 1 );
}
```

为了避免代码过长，我们删掉了以前的代码中各模块测试的相关部分，只保留了对块设备的测试代码。程序首先调用了 ramdisk_driver_init 函数进行 RAM 盘设备的初始化，然后通过 ramdisk_storage_device 结构体的 dout 成员将 RAM 盘内部 0 位置处的 128 个数据读出，保存到 buf 数组中，最后通过循环将数组中的数据读出。

修改“boot.c”文件的同时还要记得在该文件的开头处加上“storage.h”

头文件。

最后，不要忘记更改 Makefile 的 OBJS 变量，如下：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o  
interrupt.o mem.o driver.o ramdisk.o
```

好了，现在编译运行程序，运行结果如下：

```
arch: arm  
cpu info: armv4, arm920t, 41009200, ff00ffff, 2  
mach info: name s3c2410x, mach_init addr 0x426c70  
uart_mod:0, desc_in:, desc_out:, converter:  
SKYEye: use arm920t mmu ops  
Loaded RAM ./le eos.bin  
Loaded RAM ./ram.img  
start addr is set to 0x30000000 by exec file.  
helloworld  
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 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

最终打印出来的值都是 0，恰好是“ram.img”的内容。读者也可以使用其他文件替换“ram.img”，来验证一下结果的正确性。

这样，一个最简单的存储设备驱动程序就完成了。当然在实际的系统应用中，我们必须还要考虑至少如下两个问题：第一个问题是，在极端情况下，操作系统可能同时会有七八个存储设备存在，那样的话，程序必须在初始化时，既调用 ramdisk_driver_init() 又调用 nandflash_driver_init()，还要调用其他的 driver_init 函数。这样不但复杂，而且难以管理。因此，比较理想的方法是让驱动程序自行注册到系统中，而只在初始化的时候调用一次 all_driver_init() 函数。另一个问题是存储设备分区的问题，通常的方法是定义结构体去描述每一个分区，而对这些分区的操作，则共享一组操作函数即可。

7.2 文件系统框架

现在，我们的操作系统已经具备了利用外部介质存储数据的条件，但这

并不意味着操作系统的数据管理部分就此完成了。

数据管理是操作系统的另一个重要核心。一个应用程序几乎不可能不需要对数据做处理，那么应用程序应该怎样去处理数据呢？这个问题乍看起来非常简单。在计算机的世界里，我们可以使用一组二进制数来描述所有的对象，数据自然也不例外。于是，在一个操作系统已经可以操作存储设备的前提下，将表示数据的一组二进制数顺序地写入存储器中，不就实现了数据的存储了吗？

没错，很多低端的操作系统的确是将数据直接存储在存储设备当中，但这样一点都不明智。

一方面，因为直接存储在存储设备上的数据很难管理，比如需要把个人的联系方式作为通讯录存储到 flash 中，如果数据不加处理直接存储，那么当我们想添加新的联系人、删除某个人的数据或者查找某个人的时候，就会变得非常困难。

另一方面，如果数据过于频繁地被删除和存储，就会很容易产生存储器碎片，前面章节中介绍的内存就是一个很典型的例子。当系统中碎片过多时，就会导致存储器利用率过低，并会降低数据读写的速度。还有一点是既然数据是直接存储在存储设备上的，这就要求用户至少在一定程度上了解存储设备的结构和原理，而对于一个存储设备众多的平台来说，就必须要求用户既要懂得 flash 的工作原理，又要懂硬盘、光盘的工作原理……很显然，这是非常不切实际的。

因此，高级操作系统都会想办法解决这个问题，不直接使用二进制数的形式去存储数据，而是将数据适当进行加工，使加工后的数据可以有效地克服上面几个缺点，然后把新的数据存储到设备当中。这种数据加工的格式和方法，我们就称它为文件系统。

7.2.1 文件系统的原理

文件系统其实是操作系统的一种抽象，它可以让存储设备的操作变得容易，用户不再需要了解存储器原理，而只需要知道文件系统的基本概念，如文件、文件夹等，就可以直接进行数据读写了。另外，由于要存储的数据是被重新加工后再被存储的，因此我们可以将数据的存储位置、大小等信息也

存到存储器中，这样，再进行数据查找、插入或删除时，就会变得相当容易了，就是因为这种抽象性，文件系统才不能代表一个具体的存储载体，而必须依托某一个具体的存储设备才会有意义。

我们刚才说实际要存储的数据在存储之前先要按照某种格式加工一下，再存储到存储器中，这种格式就叫做文件系统类型。读者一定会对 FAT 和 NTFS 这两种文件系统类型非常熟悉，这些都是在 Windows 系统下使用的默认文件系统类型。不同文件系统类型下的文件内容是一致的，只不过组织的方法不同罢了，而将某个存储设备组织成某种文件系统类型的过程就叫做格式化。文件系统在内核中的结构如图 7-1 所示。

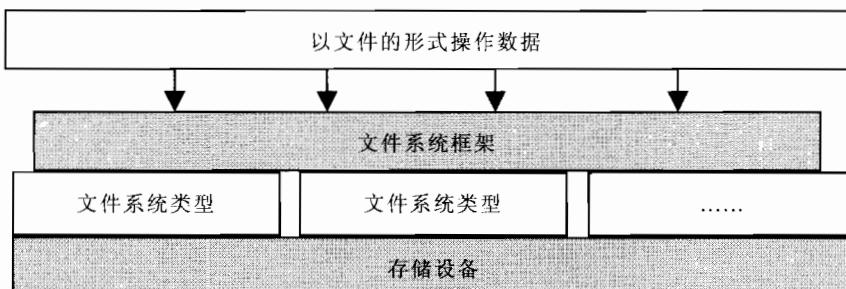


图 7-1 文件系统在内核中的层次结构

与驱动程序一致，本书中我们也不会将重点放在实现某种文件系统类型上，而是会从一个更高的层次出发去实现一个文件系统的框架。这样一来，任何一种文件系统都可以在这个框架下轻易地实现，从而从一个更高的角度去完成操作系统的设计。

现在，我们就正式进入文件系统框架的研究当中。在设计这个文件系统框架的时候，我们首先会想到的也许就是定义一个结构，用来描述文件系统上的一个对象。这里的对象就是我们平时所说的文件。

在很多操作系统中都用索引节点这个概念来描述这样一个结构，比如，在 minix 中，将索引节点称做 inode，在其他类 UNIX 操作系统中的命名方法也与此类似。索引节点列出了文件的属性等信息以及文件中各个数据块在磁盘中的相应位置。因此，我们可以得出如下结论：每个存储在存储器中的文件都会有一个索引节点与之对应。也可以说，索引节点就是存储在存储器中的数据的抽象。当我们想从设备中读取一段数据时，应该首先找到代表该

数据的索引节点，再调用与索引节点相配套的数据操作函数，通过索引节点中记录的数据信息正确地读出数据。这个函数在传统的 UNIX 操作系统中被称为 namei。使用 namei 能够得到对应文件的索引节点，而索引节点里又记录了必要的数据信息，于是我们就可以调用上一小节实现的存储设备驱动程序，将数据从索引节点中记录的实际数据存储位置中读出来，从而实现对数据的读取。

索引节点的意义如图 7-2 所示。

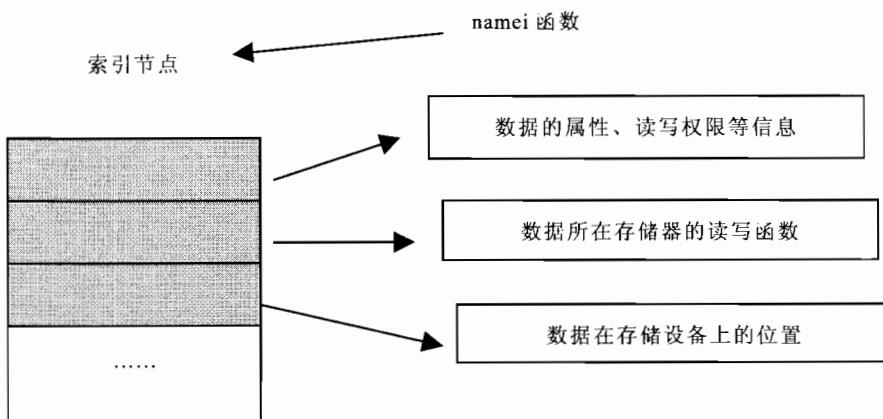


图 7-2 索引节点的意义

这样，我们的操作系统只需要根据某种文件系统类型的组织方式写出一个相应的 namei 函数，就可以支持该文件系统类型。当需要读出某个文件时，首先要知道这个文件是以哪种类型进行格式化的，还要知道数据被存储到了哪个存储器中，最后调用该文件系统类型的 namei 函数得到数据地址等信息，再调用相应的存储器读写函数进行实际的数据读写。

可以看出在实现了文件系统框架之后，无论文件系统类型怎样变化，是支持最新的文件系统类型还是剔除掉已经淘汰了的文件系统类型，都会变得很灵活。文件系统的框架是不变的，可变的只是与文件系统类型有关的代码。

7.2.2 文件系统框架的实现

好了，文件系统框架的原理我们已经搞清楚了。接下来让我们一起来实现一个最简单的文件系统框架吧！



代码 7-7

```
#include "storage.h"

#define MAX_SUPER_BLOCK    (8)
#define ROMFS   0

struct super_block;
struct inode{
    char *name;
    unsigned int flags;
    size_t dsize;
    unsigned int daddr;
    struct super_block *super;
};

struct super_block{
    struct inode *(*namei)(struct super_block *super, char *p);
    unsigned int (*get_daddr)(struct inode *);
    struct storage_device *device;
    char *name;
};

extern struct super_block *fs_type[];
```

代码 7-7 中包含了两个结构体，其中，`struct inode` 就是前文中提到的索引节点。

在该结构体中，成员 `name` 指向了该索引节点所代表的文件名，`flags` 成员用来实现对文件的控制。`dsize` 和 `daddr` 分别代表了文件的大小和它在存储器中的位置。从图 7-2 中索引节点的例子中可以看出，一个文件既包含了与文件系统相关的头信息又包含了真正的数据。在这里，成员 `dsize` 用来描述真实数据的大小，而 `daddr` 描述的则是整个文件在存储器中的实际位置，包含实际数据和相关的头信息。最后一个成员是一个结构体，名为 `super_block`，该结构体用来描述针对某个文件系统类型的控制数据和操作方法。每个文件系统类型都有一套这样的方法。这里，我们采用的都是与类 UNIX 操作系统中类似的命名方式。

`struct super_block` 结构体包含了前面提到过的 `namei` 函数指针，该指针指向的函数能够通过一个文件的文件名得到代表该文件的 `inode` 结构体。函数指针 `get_addr` 指向的函数能够计算头信息的大小，返回实际数据所在的位置。

置。成员 name 代表了该文件系统类型的名字，成员 device 是一个 storage_device 类型的结构体，它代表了这个文件系统类型所依存的存储设备。

代码 7-7 中还声明了一个 struct super_block 类型的数组，这个数组代表操作系统中所支持的文件系统列表。为了简化设计，我们采用数组的方式静态地管理每一个注册到该文件系统框架中的文件系统类型，与之相关的代码如下：

代码 7-8

```
#include "string.h"

#define MAX_SUPER_BLOCK 8
#define NULL (void *)0

struct super_block *fs_type[MAX_SUPER_BLOCK];

int register_file_system(struct super_block *type,unsigned int id) {
    if (fs_type[id]==NULL) {
        fs_type[id]=type;
        return 0;
    }
    return -1;
}

void unregister_file_system(struct super_block *type,unsigned int id) {
    fs_type[id]=NULL;
}
```

代码 7-8 主要定义了两个函数，它们是 register_file_system 和 unregister_file_system。这两个函数的作用分别是将一个新的文件系统注册到文件系统框架中，以及从文件系统框架中删除一个已有的文件系统。这里的实现方法相当简单，无非就是操作 fs_type 这个数组的成员指向注册了的文件系统类型。系统还会给所支持的每一个文件系统类型都分配一个唯一的 ID，这个 ID 就代表了 fs_type 数组的索引，保证数组之间各成员不会冲突。

这样，一个基本文件系统框架就算完成了。与驱动程序框架类似，这段示例代码虽然功能简单，但是也足以说明文件系统框架的结构了。

接下来，我们就结合某一个文件系统类型来说明一下这样一个简单的文件系统框架是如何工作的。

7.2.3 romfs 文件系统类型

下面以某一个文件系统类型为例实现一个文件系统框架及其功能。

文件系统类型的种类非常多，除了前面提到过的 NTFS 等类型，比较知名的还包括 ext 系列、hfs 和 zfs，以及嵌入式环境下常常被提到的 jffs 系列和 cramfs，等等。甚至，我们自己也可以根据实际需要设计出自己的文件系统类型。

然而，综合比较各种文件系统类型的特点，本书最终决定以 romfs 为例来演示文件系统框架的作用，其原因主要包括如下两点。

第一，romfs 应该是功能完整的文件系统类型中最简单的一个了。romfs 文件系统广泛地被 Linux 以及 uClinux 所支持，其功能可谓完整，而且实现方法也相对简单，因为 romfs 是只读的文件系统，没有写的动作。在 Linux 中核心代码只有六七百行，相对于其他文件系统类型来说，这个尺寸简直太“苗条”了。因为我们的重点是文件系统框架，因此文件系统类型选择得越简单，就越能说明问题。

第二，romfs 在嵌入式领域里的应用也是非常广泛的。本书虽然展现了一个通用操作系统的完整结构，但重点仍然要侧重于嵌入式环境。因此可以说，romfs 文件系统类型非常合适。

既然选择了 romfs，那么在写代码之前，我们就必须要首先搞懂 romfs 的具体格式是怎样的，如图 7-3 所示。

首先，每一个 romfs 映像都包含一个文件系统头信息，用来记录整个文件系统类型的基本情况，比如，该 romfs 映像叫什么名字，一共有多少字节，等等。在 romfs 的文件系统头信息当中，前 8 个字节分别用来存储 '-'、'r'、'o'、'm'、'l'、'f'、's'、'-' 这 8 个字符，它们代表了 romfs 文件系统类型的标识符。

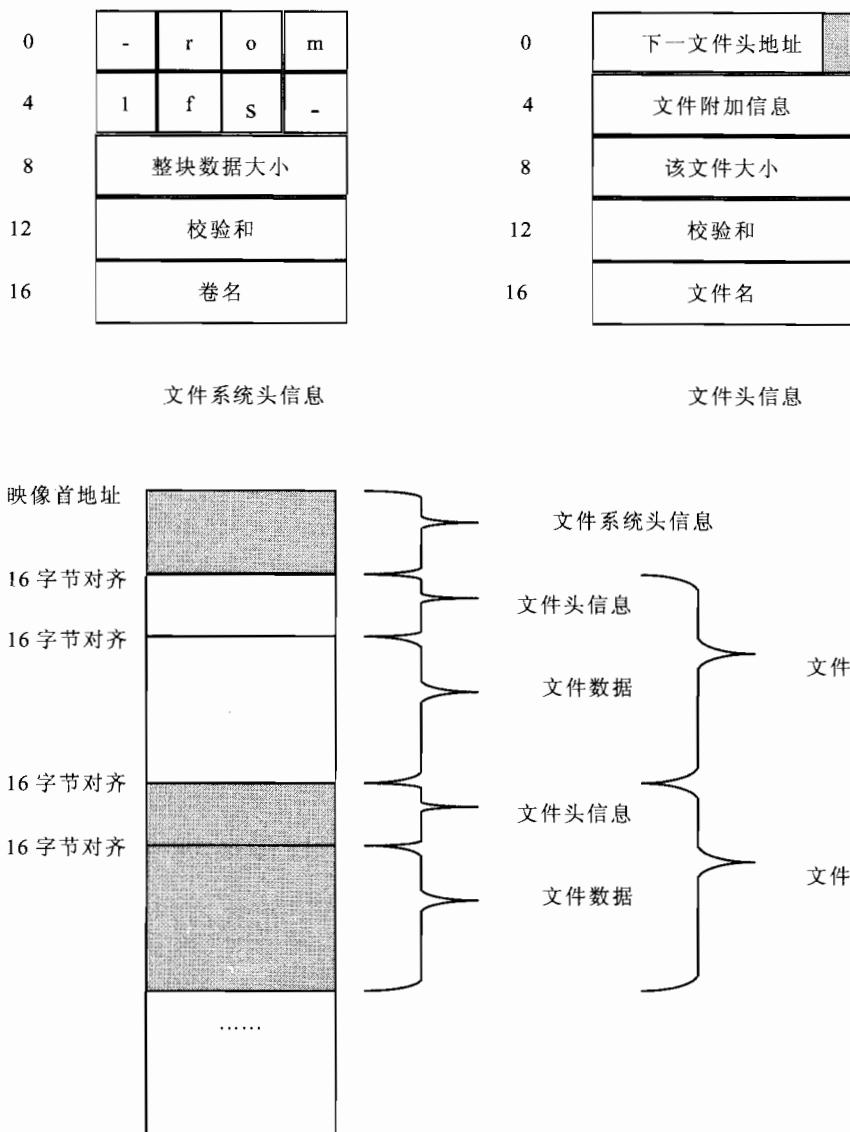


图 7-3 romfs 文件系统类型

在文件系统头信息结束的位置向下偏移 16 个字节处，就是该 romfs 映像中存储的第一个文件了。当然，首先出现的并不是数据本身，而是代表该段数据的文件头，其中记录了这个文件的名字、文件的数据部分有多大、文件是什么类型的文件等信息。这里，描述文件类型的信息由文件头信息最开始 4 个字节中的最后 3 个位 bit[0-2] 来表示，也就是图中小方块的部分，而



bit[3]则代表了这个文件的执行属性。这些特点都是为了兼容 POSIX 标准而专门设计的。同时，文件头信息中的第 4~第 7 个字节说明了不同文件的附加信息，完整的说明如表 7-1 所示。

表 7-1 romfs 文件附加信息

数值（二进制）	含义	附加信息
000	硬链接	目的文件
001	目录	目录内首文件地址
010	普通文件	—
011	软链接	—
100	块设备文件	主、次设备号
101	字符设备	主、次设备号
110	socket 文件	—
111	fifo 文件	—

紧接着头信息后边的就是文件的实际数据了，当然这些数据的起始位置也需要按照 16 个字节向下对齐。

最后我们还要强调，romfs 中使用的所有数据都是大尾端的，这就要求在读取数据的时候，必须将其转换成 cpu 当前默认的端格式。

尾端这个概念我们前面从未提过，可能有些读者不清楚什么是端。通俗地讲，尾端就是在内存中，是用 0x01000000 来表示数字 1 还是用 0x00000001 来表示数字 1 的问题。

图 7-4 是大小尾端的数据存储示意图。

内存以字节为基本存储单位，那么当系统需要在内存中存储一个字节时，无论怎样都不会出现问题，但如果要存储的数据有两个字节呢？比如 0x0A0B 这个数，应该在低地址处存储 0x0A、高地址处存储 0x0B 还是相反呢？

很显然，存储的方式无所谓哪种正确，关键是要让所有数据都保持一致。这两种存储方式都是正确的，将低字节数存储在低端内存的位置上，就是小尾端格式，而将高字节数存储在低端内存上，就是大尾端格式。

读者也许紧接着就要问了，什么因素会影响尾端呢？

其实只要用心想一想就会知道，尾端跟 CPU 和人有关。在我们没有通知 CPU 采用什么尾端格式存储数据的前提下，CPU 会以本平台默认的方式完成数组存储。但对于某些平台来说，程序员也可以在编译程序时，人为地修改 CPU 的默认尾端方式。



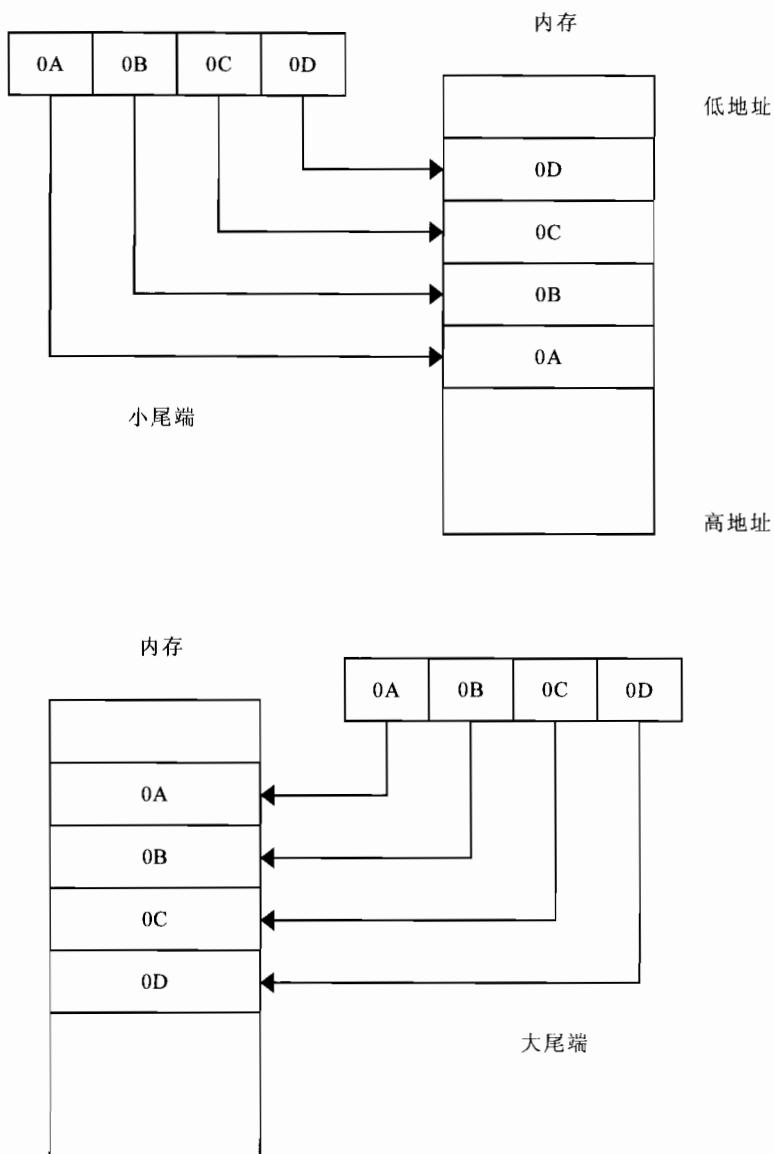


图 7-4 大小尾端示意图

现在一切都清楚了。ARM 体系结构在默认情况下采用小尾端格式存储数据，而我们也没有人为地更改这种默认格式，因此我们的操作系统使用的就是小尾端格式。

这样看来，将 romfs 文件系统类型从大尾端转换成小尾端的操作过程就

不可避免了。结合图 7-4 的描述，我们能够得出尾端转换的方法，只要将高位数据与低位数据互换即可。

7.2.4 实现 romfs 文件系统

此刻，在了解了 romfs 的存储格式后，我们就可以得出操作 romfs 文件系统类型的一般步骤了，代码的具体实现方法如下：

代码 7-9

```
#include "fs.h"
#include "storage.h"
#include "string.h"

#define NULL (void *) 0

#define be32_to_le32(x) \
    ((unsigned int)( \
        (((unsigned int)(x) & (unsigned int)0x000000ffUL) << 24) | \
        (((unsigned int)(x) & (unsigned int)0x0000ff00UL) << 8) | \
        (((unsigned int)(x) & (unsigned int)0x0ff00000UL) >> 8) | \
        (((unsigned int)(x) & (unsigned int)0xff000000UL) >> 24) \
    ))

struct romfs_super_block {
    unsigned int word0;
    unsigned int word1;
    unsigned int size;
    unsigned int checksum;
    char name[0];
};

struct romfs_inode {
    unsigned int next;
    unsigned int spec;
    unsigned int size;
    unsigned int checksum;
    char name[0];
};

struct super_block romfs_super_block;
```

```

#define ROMFS_MAX_FILE_NAME (128)
#define ROMFS_NAME_ALIGN_SIZE (16)
#define ROMFS_SUPER_UP_MARGIN (16)
#define ROMFS_NAME_MASK \
    (~ (ROMFS_NAME_ALIGN_SIZE - 1))
#define ROMFS_NEXT_MASK 0xffffffff0

#define romfs_get_first_file_header(p) \
    (((strlen(((struct romfs_inode *) (p))->name)) \ 
    + ROMFS_NAME_ALIGN_SIZE + \
    ROMFS_SUPER_UP_MARGIN)) & \
    ROMFS_NAME_MASK) << 24)

#define romfs_get_file_data_offset(p, num) \
    (((((num) + ROMFS_NAME_ALIGN_SIZE) & \
    ROMFS_NAME_MASK) + \
    ROMFS_SUPER_UP_MARGIN) + (p)))

```

代码 7-9 定义了实现 romfs 文件系统类型所需要的数据结构、变量和宏。

结构体 struct romfs_super_block 用于描述 romfs 文件系统头信息，而 struct romfs_inode 则用于描述 romfs 文件头信息。这两个结构体的作用和各成员的含义可以参考图 7-3 的描述。

宏 ROMFS_MAX_FILE_NAME、ROMFS_NAME_ALIGN_SIZE 等都是为了对 romfs 文件系统进行操作定义，是根据 romfs 文件系统类型格式而设计的。稍后我们可以看到这些宏在代码中的应用。

宏定义 romfs_get_first_file_header 专门负责找到整块磁盘中存储的第一个文件的位置。它的依据是 romfs 文件系统类型的 16 字节对齐的原则。如图 7-3 所示，只要知道了卷名的长度，再加上 struct romfs_super_block 结构体大小的偏移量，将这个值按照 16 字节对齐，就得到了第一个文件的位置了。

romfs_get_file_data_offset 宏可以通过文件头信息在存储器中的位置以及文件名的长度求出文件中实际数据所在位置，其工作方式与前面的宏类似。代码 7-9 中还有一个宏也是需要反复使用的，那就是 be32_to_le32，这个宏能将大尾端的数据转换成小尾端的数据。

介绍完了 romfs 文件系统类型相关的数据结构，我们再来研究一下文件系统框架中最最重要的一个函数——namei 函数在 romfs 中的实现方法。



代码 7-10

```
static char *bmap( char *tmp, char *dir) {
    unsigned int n;
    char *p=strchr( dir, '/');
    if (!p) {
        strcpy( tmp, dir);
        return NULL;
    }
    n=p-dir;
    n= (n>ROMFS_MAX_FILE_NAME) ? \
        ROMFS_MAX_FILE_NAME:n;
    strncpy( tmp, dir, n);
    return p+1;
}

static char *get_the_file_name( char *p, char *name) {
    char *tmp=p;
    int index;
    for (index=0; *tmp; tmp++) {
        if (*tmp=='/') {
            index=0;
            continue;
        } else{
            name[index]=*tmp;
            index++;
        }
    }
    name[index]='\0';
    return name;
}

struct inode *simple_romfs_namei( struct super_block *block, char *dir) {
    struct inode *inode;
    struct romfs_inode *p;
    unsigned int tmp, next, num;
    char name[ROMFS_MAX_FILE_NAME], \
        fname[ROMFS_MAX_FILE_NAME];
    unsigned int max_p_size=(ROMFS_MAX_FILE_NAME+\
        sizeof( struct romfs_inode));
    max_p_size=max_p_size>(block->device->sector_size) \
        ?max_p_size:(block->device->sector_size);
    get_the_file_name( dir, fname);
```

```

    if ((p = (struct romfs_inode *) kmalloc (max_p_size, 0)) == NULL) {
        goto ERR_OUT_NULL;
    }
    dir=bmap ( name,dir );
    if (block->device->dout (block->device,p,0,\n
        block->device->sector_size))
        goto ERR_OUT_KMALLOC;
    next=romfs_get_first_file_header (p);

    while (1) {
        tmp=(be32_to_le32 (next)) &ROMFS_NEXT_MASK;
        if (tmp>=block->device->storage_size)
            goto ERR_OUT_KMALLOC;
        if (tmp!=0) {
            if (block->device->dout (block->device,p,\n
                tmp,block->device->sector_size)) {
                goto ERR_OUT_KMALLOC;
            }
            if (!strcmp (p->name, name)) {
                if (!strcmp (name, fname)) {
                    goto FOUND;
                } else{
                    dir=bmap ( name,dir );
                    next=p->spec;
                    if (dir==NULL) {
                        goto FOUNDDIR;
                    }
                }
            } else{
                next=p->next;
            }
        } else{
            goto ERR_OUT_KMALLOC;
        }
    }

    FOUNDDIR:
    while (1) {
        tmp=(be32_to_le32 (next)) &ROMFS_NEXT_MASK;
        if (tmp!=0) {
            if (block->device->dout (block->device,p,tmp,\n
                block->device->sector_size)) {
                goto ERR_OUT_KMALLOC;
            }
        }
    }
}

```

```
    }
    if (!strcmp ( p->name, name )) {
        goto FOUND;
    } else {
        next=p->next;
    }
} else{
    goto ERR_OUT_KMALLOC;
}
}

FOUND:
if ((inode = ( struct inode * ) kmalloc ( sizeof ( struct inode ), 0 )) \
==NULL ) {
    goto ERR_OUT_KMALLOC;
}
num=strlen ( p->name );
if (( inode->name= ( char * ) kmalloc ( num, 0 )) ==NULL ) {
    goto ERR_OUT_KMEM_CACHE_ALLOC;
}
strcpy ( inode->name,p->name );
inode->dsize=be32_to_le32 ( p->size );
inode->daddr=tmp;
inode->super=&romfs_super_block;
kfree ( p );
return inode;

ERR_OUT_KMEM_CACHE_ALLOC:
kfree ( inode );
ERR_OUT_KMALLOC:
kfree ( p );
ERR_OUT_NULL:
return NULL;
}
```

simple_romfs_namei这个函数就是romfs中namei函数的具体实现方法。这个函数比较复杂，我们对它的介绍会相对宏观一些。

首先，想要让simple_romfs_namei函数能够正常工作，就得依赖两个辅助函数。其中一个函数是get_the_file_name，它能够除去文件的路径，得到文件名，另一个函数是bmap，函数bmap每次运行的时候，都可以将文件名字符串中最顶层的目录名去掉，然后返回余下的目录名。例如，一个文件

的完整路径名保存到下面这个变量中：

```
char *name= "firstdir/seconddir/filename"
```

那么，当我们调用 bmap 函数时：

```
char *newname=bmap ( buf, name );
```

函数返回值 newname 就应该是 seconddir/filename 字符串，而 buf 内存区存储的字符串就应该是 firstdir 了。simple_romfs_namei 函数正是反复调用了 bmap 函数，一次次剥掉路径名后，找到了文件的具体位置。

在 simple_romfs_namei 函数的一开始，程序首先定义了一些变量，获取最终文件名、分配内存空间。其中，临时变量 p 代表新申请的内存空间的首地址。通过 get_the_file_name 函数，数组 fname 保存了不含路径的文件名。字符数组 name 专门给 bmap 函数使用，保存每次得到的文件路径。

紧接着，程序通过 super_block 中 device 成员的 dout 成员函数指针读取存储器中最开始的一块数据。成功读出数据之后，调用代码 7-9 中的 romfs_get_first_file_header 宏，就可以得到第一个文件在存储器中的位置了。

然后，通过两个大循环，结合 bmap 函数，一层层地查找文件的各级路径。在第一个循环中，临时变量 tmp 保存了下一个要读取的文件在存储器中的位置，根据 romfs 文件系统的规则，每一个文件头信息的 next 成员都指向了下一个文件的具体位置。程序读取这个值，然后通过 be32_to_le32 转换成小尾端，赋值给 tmp 变量。之后再次调用 dout 函数，读出新的文件内容并判断这个新的文件是不是我们要查找的文件，或者至少是我们要查找的文件的某个父目录，如果没有找到，则进行下次循环，如果找到的是文件的所属目录，则跳转到 FOUNDDIR 标签处，查找最终的文件。

如果文件最终被找到，则跳转到 FOUND 标签，在此处，程序首先分配一个 struct inode 空间，然后根据找到文件的相关头信息填充 struct inode 的相应成员。最后释放掉不使用的空间，将 struct inode 指针返回。

函数 simple_romfs_namei 成功得到了文件系统框架下的 struct inode 结构体。此时，我们再通过另外一个函数就可以使 romfs 这一文件系统类型在框架下发挥作用了。

代码 7-11

```
unsigned int romfs_get_daddr ( struct inode *node ) {
```

```

    int name_size=strlen ( node->name );
    return romfs_get_file_data_offset ( node->daddr, name_size );
}

struct super_block romfs_super_block={
    .namei=simple_romfs_namei,
    .get_daddr=romfs_get_daddr,
    .name="romfs",
};

int romfs_init ( void ) {
    int ret;
    ret=register_file_system ( &romfs_super_block,ROMFS );
    romfs_super_block.device=storage[RAMDISK];
    return ret;
}

```

这个函数就是 `romfs_get_daddr`，它能够从一个标准的 `struct inode` 结构体中得到文件实际数据的存储位置。这个函数的实现非常简单，只需要计算文件名所占的字节数，然后调用 `romfs_get_file_data_offset` 宏就可以了，这个宏被定义在了代码 7-9 中。

现在，我们需要定义一个文件系统框架下的 `super_block` 结构体为 `romfs_super_block`，然后将其中的 `namei` 和 `get_daddr` 成员分别赋值。另外，我们还需要有一个针对 `romfs` 的初始化函数 `romfs_init`，在该函数中，首先需要将 `romfs_super_block` 的 `device` 成员赋值，因为文件系统类型是一个抽象的格式，而它最终必须寄生在某个具体的存储设备中，才能发挥作用。因此，我们必须在 `romfs` 初始化的时候，就为它选定哪个设备使用了该文件系统类型去管理文件。在 Linux 中，这一工作是通过 `mount` 命令完成的，在这里我们只是简单地将这个信息直接写到代码当中去。

最后，`romfs_init` 函数还需要调用框架中的 `register_file_system` 函数，将 `romfs_super_block` 注册到系统中去。

这样，基于 `romfs` 的文件系统类型就可以在我们的操作系统中发挥作用了。

7.2.5 让代码运行起来

读者朋友们一定迫不及待地想要看一下，怎样在 `romfs` 文件系统类型中

读取文件吧？接下来我们就借助 RAM 盘，在其中存储一些实际的文件，然后利用文件系统框架下的方法来读取这些文件的内容。

首先，我们要将前面的代码添加到我们自己的操作系统中进行编译。

请将代码 7-7 的内容保存成名为“fs.h”的文件，将代码 7-8 命名为“fs.c”，这两个都是与文件系统框架有关的。然后，将代码 7-9、代码 7-10 和代码 7-11 的内容合并成一个文件，取名为“romfs.c”。所有与 romfs 文件系统类型有关的代码都保存在该文件中。

此时，我们还需要另外一个文件来提供一些必要的函数。因为在我们写操作系统的一开始，就没有使用标准库函数，所以，前面程序中使用的诸如 `memcpy` 之类的函数就必须由我们自己实现了。好在实现这些函数并不是什么难事，代码 7-12 恰好能完成这些工作。

代码 7-12

```
#ifndef __STRING_H__
#define __STRING_H__

static inline int strcmp (const char * cs,const char * ct) {
    register signed char __res;
    while (1) {
        if ((__res = *cs - *ct++) != 0 || !*cs++)
            break;
    }
    return __res;
}

static inline void * memset (void * s,int c,unsigned int count)
{
    char *xs = (char *) s;
    while (count--)
        *xs++ = c;
    return s;
}

static inline char * strcpy (char * dest,const char *src)
{
    char *tmp = dest;
    while ((*dest++ = *src++) != '\0');
    return tmp;
}
```

```
static inline char * strcpy( char * dest,const char *src,\n    unsigned int count) {\n    char *tmp = dest;\n    while (count-- && (*dest++ = *src++) != '\0') ;\n    return tmp;\n}\n\nstatic inline unsigned int strlen( const char * s) {\n    const char *sc;\n    for (sc = s; *sc != '\0'; ++sc);\n    return sc - s;\n}\n\nstatic inline char * strchr( const char * s, int c) {\n    for( ; *s != (char) c; ++s)\n        if (*s == '\0')\n            return (void *) 0;\n    return (char *) s;\n}\n#endif
```

这些函数的具体实现我们就不多解释了，相信读者一定看得明白。这里我们使用了一个小技巧，那就是将所有的函数都定义成 `inline`。`inline` 函数代表着凡是使用 `inline` 函数的地方都要将函数展开，而不是去调用这个函数，同时在编译的时候，也对函数参数进行了类型检查。所以，使用 `inline` 函数要比使用宏定义稍安全一些，而它的缺点就是会增大最后生成的代码的尺寸。

普通的函数调用只需要知道函数的入口地址就可以了，因此在将程序编译成目标文件时，并不需要知道函数是怎样实现的。而 `inline` 函数则不同，因为需要原地展开，所以程序在使用 `inline` 函数之前，必须知道函数是怎样实现的。因此，一个 `inline` 函数通常被定义在头文件中，然后让别的程序包含这个头文件。但这样会带来另外一个问题，当两段程序包含了同一个头文件时，编译时就会出现同一个函数被定义两次的情况，这在 C 语言中是不允许的。所以在 `inline` 函数定义时，还要加上 `static` 关键字，表示该函数是私有的。

我们将代码 7-12 保存成文件，命名为“`string.h`”。然后修改 `Makefile`

中的 OBJS 变量为如下形式：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
interrupt.o mem.o driver.o ramdisk.o romfs.o fs.o
```

此时，文件系统框架和 romfs 文件系统类型的代码就基本添加完成了。在运行这段代码之前，我们还需要做一个额外的工作。不要忘记，现在系统中的“ram.img”文件镜像中并没有被格式化成 romfs 的文件系统类型，这个过程需要一些工具的辅助。

为了使代码结构更清晰，我们需要新建一个目录，将格式化 romfs 的工具保存到这个目录当中。无论在 Linux 系统中还是 Cygwin 模拟环境下，都可以使用 mkdir 命令。于是，请先使用 cd 命令切换到代码的根目录下，我们需要在程序根目录下运行如下命令：

命令 7-1

```
mkdir tools
```

然后，进入该文件夹中。

命令 7-2

```
cd tools
```

读者需要从 romfs 的网站或者网站 www.leeos.org 上下载一个名为“genromfs.c”的源程序到当前文件夹，使用下面这条命令，就可以生成一个能够格式化 romfs 文件系统类型的工具了。

命令 7-3

```
gcc genromfs.c
```

这样会生成一个名为“a.out”的文件。

同时，我们需要在 RAM 盘中存储一个文件。请在 tools 目录中新建另一个文件夹，取名为“filesystem”。

命令 7-4

```
mkdir filesystem
```

文件夹“filesystem”里的内容，就代表了存储在 RAM 盘根目录中的内容。于是我们可以使用 echo 命令，在 RAM 盘中生成一个文件，里面存储 0~9 共 10 个数字。

命令 7-5

```
echo "0123456789" > filesystem/number.txt
```

然后，使用刚才编译成的 a.out 命令生成一个 romfs 文件系统格式的文件镜像，取名为“romfs.img”。

命令 7-6

```
./a.out -d filesystem -f romfs.img
```

最后，使用 dd 命令将“romfs.img”的内容写到我们之前使用过的“ram.img”文件当中。如果读者就是按照本书的方法进行操作的话，这个文件就应该在父目录中。于是，运行如下命令：

命令 7-7

```
dd if=./romfs.img of=../ram.img
```

这样会让“ram.img”文件变小，不过这不会影响到程序的正常运行。

好了，现在是时候使用程序读取 RAM 盘中的文件了。我们需要修改“boot.c”文件。

代码 7-13

```
void plat_boot( void ) {
    int i;
    for ( i=0; init[i]; i++ ) {
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    // timer_init();

    init_page_map();
    kmalloc_init();
    ramdisk_driver_init();
    romfs_init();

    struct inode *node;
    char buf[128];

    node=fs_type[ROMFS]->namei( fs_type[ROMFS], " number.txt " );
    fs_type[ROMFS]->device->dout( fs_type[ROMFS]->device,\
```

```

        buf, fs_type[ROMFS]->get_daddr ( node ), \
        node->dsize);

    for ( i=0;i<sizeof ( buf );i++ ) {
        printk ( " %c ",buf[i] );
    }

    while ( 1 );
}
}

```

在代码 7-13 中，程序首先使用了 romfs_init 函数进行 romfs 文件系统的初始化。然后调用 fs_type[ROMFS] 的 namei 函数，从 RAM 盘中读取“number.txt”文件，得到代表该文件的 inode 结构体。接下来，程序调用 device 成员的 dout 函数，从该结构体中读取实际数据到 buf 中。最后，通过循环将 buf 中的值以字符的形式打印出来，不出意外，打印的结果应该是 0~9。

最后，不要忘记在“boot.c”文件的最开始处添加“fs.h”头文件。

一切都准备好后，就可以编译运行程序，运行的结果如下：

```

Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEye: use arm920t mmu ops
Loaded RAM ./le eos .bin
Loaded RAM ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
1 2 3 4 5 6 7 8 9 0

```

7.3 总结

相信读者已经理解了框架的真正作用和意义。

从程序员的角度来看，我们必须承认一个现实，那就是——需求的变化



是项目过程中唯一不变的东西。在日常的工作过程中，相信很多读者都会对这句话深有感触。

如果读者朋友们能够承认这一点，就不会去不遗余力地追求那些不可能实现的完美程序，或者要求别人一开始就提出确定不变的完整需求，而是会努力地让自己的程序尽最大可能适应一切变化，能够在变化中投入的资源最少却收益最大。想要实现这一点，就需要把握一条最基本的设计原则，那就是将相对不变的对象独立出来，而让勤于变化的对象去随意改变，同时尽可能地降低二者之间的耦合。

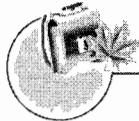
框架正是因此而诞生的。有了框架，就拥有了一套相对稳定的结构，此时，它就成为了相对不变的对象，在需求的变化过程中，框架就会相对静止，而那些依存框架的部分也就相对静止了。将那些依托于框架的实现安排在框架之外，让这一部分因时而动，并保持二者之间的低耦合。

这样一来，我们便拥有了主动应对需求变化的致命武器。

在我们的操作系统中，文件系统和驱动程序框架都是为此而设计的。虽然代码短小、功能单一，但其背后的思想却是强大的。因为框架结构设计的相对不变，所以依存于框架之上的应用程序就可以保持相对稳定，当我们需要在操作系统中实现新的文件系统或驱动程序时，只需要实现框架之下的接口，而框架之上的一切程序都无须改变。

用这样的设计思路和方法来写程序，我们的程序就能够实现真正的“永生”。





第8章

运行用户程序



经过了不懈的努力，现在我们的操作系统已然有了一点大家风范。

然而，无论我们如何强调操作系统的重要性，如何想尽办法来提高操作系统的运行效率和稳定性，都不得不承认一个事实，那就是我们的操作系统目前似乎什么有用的事都做不了。它不能用来听歌、看电影，不能玩游戏、上网……

事实上，一个操作系统的作用不是发挥在功能上，而是发挥在机制上。就像 Windows一样，仅仅装上了 Windows 操作系统，基本上是用不了的，你要额外装一个 QQ 工具才能聊天，要装一个客户端，才可以玩网络游戏，所以，操作系统并不能让你直接去做什么，而是在你想做什么的时候，尽最大可能地为你提供足够的支持，当然操作系统的作用还包括了资源的适当分配和系统监控，保证了玩游戏的时候不会因为程序抢占了所有的资源而导致你无法聊天。

与之对应的，应用程序负责的正是功能上的实现。在强大的操作系统支持下，应用程序能够更自由地实现各种复杂功能。这些程序虽然可以通过与操作系统相类似的方法编写和编译并被操作系统所调用，但是很多时候，它们不能享受与操作系统相同的待遇，并且要受到来自于操作系统的限制和管理。

从这个角度上看，操作系统就像一位司令官，自己并不会亲自上战场打仗，而是分配和调动手下的士兵，尽可能为他们提供充足的枪支弹药去赢得战场上的胜利。而这些冲锋陷阵的沙场战士们就是用户应用程序。





我们的操作系统现在已经成长为了一名合格的司令官。于是，本章的重点内容将会是如何让我们的操作系统有效地运行应用程序。

8.1 二进制程序的运行方法

其实，让我们的操作系统运行一个应用程序是非常简单的。不要忘了，应用程序也是程序，操作系统也是程序，让一个程序去调用另一个程序，又会难到哪儿去呢？

在 ARM 体系结构中，实现程序的跳转有很多种方法，这些内容在前面我们都有介绍。

但有一点我们需要强调，我们这里的应用程序指的都是独立程序。因为有的嵌入式操作系统将应用程序定义成实现具体功能的程序，在这样的操作系统中，应用程序往往会与操作系统内核统一编程以实现程序的简化。

既然程序是独立的，那么一些简单的程序跳转方法就不能实现应用程序的运行了。此时我们可以获取应用程序的入口地址，使用 mov 指令来实现。

应用程序的入口地址应该怎样获取呢？目前没有什么好办法，只能让操作系统事先跟应用程序商量好，比如让一个应用程序在编译时就链接到某地址处，然后操作系统要把应用程序复制到链接时的内存位置，最后调用 mov 指令将 PC 寄存器的值赋值成该地址，实现程序运行。

下面我们通过一个小小例子，看看最简单的应用程序调用过程是如何实现的。

代码 8-1

```
int main() {
    const char *p= "this is a test application\n";
    while (*p) {
        *(volatile unsigned int *) 0xd0000020 = *p++;
    }
}
```

代码 8-1 就是这样一个应用程序。为了保证代码的简单，程序仅仅打印出 " this is a test application\n " 这个字符串，以证明应用程序的正常运行。



然后，我们需要将代码 8-1 保存到文件之中，命名为“main.c”，把这个文件保存到代码根目录的“tools”文件夹里，然后切换到这个文件夹并使用如下命令编译这段程序。

命令 8-1

```
arm-elf-gcc -e main -nostartfiles -nostdlib -Ttext 0x30100000 -o  
main main.c
```

这些编译选项的具体含义前面都已经介绍过了。虽然这里编译的是用户应用程序，但我们却不能像通常编写程序那样使用标准库函数和启动程序。广义上讲，标准库函数也属于应用程序的范畴，但此时这些函数库尚未建立。因此，目前的应用程序会显得非常简陋，我们既不能在程序中使用标准库中的函数和头文件，也不能在编译程序的过程中链接它们。

与此同时，为了保证程序能够正常运行，我们必须在编译时指定程序的运行地址，让它恰好落在有效的内存中。于是程序选定了 0x30100000 这个地址。回忆一下前面的内容，我们在 MMU 一节中将实际的物理地址映射到了虚拟地址中相应的位置。因此在虚拟地址空间中，0x30000000 之后 8M 的内容都是有效的，留出操作系统自身所占的内存之后，0x30100000 这个位置就非常合适了。

既然代码已经编译完成了，那么我们的应用程序是不是就可以运行了？

不，现在还不是时候。要知道，GCC 默认只会生成 ELF 格式的文件，这是一种非常常用的可执行程序的文件格式，但因为这样一个可执行的文件包含了除代码和数据之外的附加信息，所以不能直接拿来运行。有关 ELF 文件格式的详细内容，我们稍后会有介绍。

这样一来，想要运行这段应用程序，必须首先将由 GCC 编译生成的文件转换成二进制的形式，也就是仅由代码和数据所组成的文件，使用的命令如下：

命令 8-2

```
arm-elf-objcopy -O binary main main.bin
```

此时我们需要将“main.bin”文件复制到“filesystem”文件夹中：

命令 8-3

```
cp main.bin filesystem
```

不要忘了，现在在我们的操作系统中已经能够支持文件系统了。也就是说，程序完全可以以文件的形式来读取数据，不再需要使用存储器的原始操作方法了。

于是我们便可以使用上一章提到的方法，将 filesystem 文件夹中的内容制作成 romfs 格式的映像文件，并将这个映像文件复制到源代码根目录的“ram.img”文件中。

这样，我们的文件系统之中便包含了一个可用的用户应用程序，接下来我们就来运行它。

代码 8-2

```
int exec( unsigned int start ) {
    asm volatile (
        " mov pc,r0\n\t"
    );
    return 0;
}
```

想要在操作系统的源代码中调用一个外部应用程序，首先应该有一个能够运行外部应用程序的函数，代码 8-2 就是这样一个函数。

前面我们已经提到过，在 ARM 体系结构中运行外部应用程序，`mov` 指令是一个不错的选择。使用这条指令将程序计数器 PC 的值赋值为外部应用程序的入口地址。结合过程调用标准，函数的第一个参数会保存到寄存器 R0 中。所以在代码 8-2 的 `exec` 函数里，程序简单地将 R0 的值赋值给 PC，目的是去运行地址 R0 处的代码。而函数 `exec` 的使用方法就是把外部的用户应用程序在内存中的地址作为参数传递给 `exec` 函数。

现在，请将代码 8-2 的内容保存成文件，名为“`exec.c`”。然后，修改 `Makefile` 中的 `OBJS` 变量来编译这个文件。接下来，我们还需要将文件“`boot.c`”稍做修改，其内容如下：

代码 8-3

```
void plat_boot( void ) {
    int i;
    for( i=0;init[i];i++ ) {
        init[i]( );
    }
    init_sys_mmu( );
```

```

    start_mmu( );
// timer_init( );

    init_page_map( );
kmalloc_init( );
ramdisk_driver_init( );
romfs_init( );

    struct inode *node;
char *buf=(char *) 0x30100000;

if((node=fs_type[ROMFS]->namei(fs_type[ROMFS], \
        "main.bin"))==(void *)0) {
    printk("inode read error\n");
    goto HALT;
}

if(fs_type[ROMFS]->device->dout(fs_type[ROMFS]->device, \
        buf,fs_type[ROMFS]->get_daddr(node), \
        node->dsize)) {
    printk("dout error\n");
    goto HALT;
}

exec(buf);
HALT:
    while(1);
}

```

在代码 8-3 中，程序首先通过 romfs 文件系统类型的 namei 函数得到一个 struct inode 结构体，并赋值给 node 指针，然后再调用该文件系统类型所寄生的存储设备的 dout 方法，将该 inode 表示的文件的实际内容读到内存 buf 处。

这里我们使用了一个非常不规范，甚至是危险的做法，将 buf 的指针强行指向 0x30100000 内存处，因为这段代码只是示例，同时，操作系统中目前并没有使用 0x30100000 这段内存空间，所以此处姑且可以这样用，并不会产生错误的结果。

那为什么一定要将 buf 指向 0x30100000 呢，别的内存地址不可以吗？别忘了，用户的外部应用程序是从 0x30100000 处开始运行的，这在程序编



译的时候就已经确定了。因此，程序只有被加载到该处才能够正常运行。

现在，请编译并运行程序，不出意外的话，程序的运行结果将会如下：

```
arch: arm  
cpu info: armv4, arm920t, 41009200, ff00ffff, 2  
mach info: name s3c2410x, mach_init addr 0x426c70  
uart_mod:0, desc_in:, desc_out:, converter:  
SKYEye: use arm920t mmu ops  
Loaded RAM ./le eos.bin  
Loaded RAM ./ram.img  
start addr is set to 0x30000000 by exec file.  
helloworld  
this is a test application
```

可以看出，“this is a test application”字符串被成功打印出来了，这表明用户应用程序运行得正确无误。

上面我们实现了一个在自己的操作系统中调用外部程序的简单方法。就好像是在 Windows 系统中安装了一个应用程序，并成功运行了它。从此以后，凡是功能上的需求，都可以交给用户应用程序去完成。虽然我们的实现方法很简单，但至少给应用程序的正确运行铺平了道路。接下来我们就来慢慢地完善这段代码，让应用程序的调用过程更加正规和不受限制。

8.2 可执行文件格式

在前面的例子中，用户应用程序已经可以正常运行了。但是程序的运行却受到了限制，其中一点就是，操作系统必须将应用程序加载到指定的位置去运行。这样一来，如何能够更方便地知道一个应用程序的运行地址而不是简单地让开发者去约定，就成了一个必须要解决的问题。

解决这个问题的思路是，由应用程序根据操作系统的指导去选择自己的运行地址，然后应用程序想办法将该地址通知操作系统，操作系统利用各种手段构建出能够运行应用程序的环境，整个过程都是自动实现的，不需要程序员人为参与。

其中，应用程序选择运行地址和操作系统运行应用程序这两个步骤都没有



什么好说的，重点是应用程序如何将自身运行的必要信息通知给操作系统。

其实，让一个可执行的用户程序不仅仅包含代码和数据，还包含一些额外的信息即可。操作系统在运行程序时，首先读取并分析这些信息，检查这些信息的正确性，然后根据这些信息构建运行环境，还要将真正的代码从文件中提取出来，放到正确的位置上再去运行。

在这些信息中，最重要的一条就是程序的运行地址。其他信息还包括程序的版本，程序是使用什么样的编译器编译的，程序是运行在哪种硬件平台、哪个操作系统下的，等等。这些其他信息，有的时候能够避免应用程序由于版本不匹配、格式不正确或系统不兼容等问题而造成的运行错误。

既然应用程序包含了真正的代码和相关信息，那么这些信息采用什么样的格式进行存储就是我们需要考虑的问题了。其实格式并不重要，我们完全可以自定义一种格式，然后让我们的操作系统去支持它。当然，这是不明智的。对于可执行文件格式，更好的做法是选择一种主流标准并去实现它，于是我们选择了 ELF。

8.2.1 ELF 格式的组成结构

ELF 指的是 Executable and Linkable Format。最初是由 UNIX 系统实验室作为应用程序二进制接口开发和发行的，后来逐渐发展成为了可执行文件的格式标准，在很多操作系统和非操作系统环境中都有非常广泛的应用。完整的 ELF 格式标准涉及了三个方面的内容。在这里我们只需要关心一个方面，那就是一个 ELF 格式可执行程序的组成结构。

一个 ELF 可执行文件格式如图 8-1 所示。

像图 8-1 那样，一个 ELF 可执行文件包含了一个描述全局信息的 ELF 文件头、若干个 Program 头、若干个 Segment 以及若干个可有可无的 Section 头。在 Segment 中保存的正是程序的运行代码，而 Program 头描述了各个 Segment 和其他必要信息，如链接库信息、文件辅助信息等。在代码真正运行时，我们往往只关心实际的代码部分，而与运行无关的其他信息则可以忽略掉。

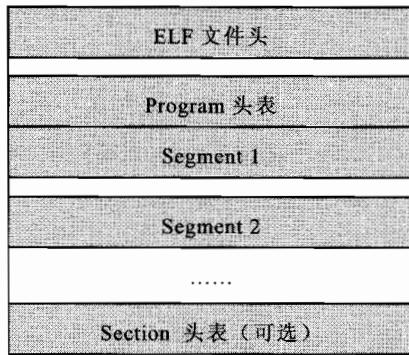


图 8-1 ELF 文件结构

当操作系统需要执行一个 ELF 格式的文件时，只需要分析 ELF 文件头，然后找到 Program 头的位置，依次分析这些 Program 头，找到代表代码和数据的 Segment，最后将这些 Segment 复制到指定的地址处，便可以运行程序了。

一个 ELF 文件头由如下部分组成：

代码 8-4

```
typedef unsigned int elf32_addr;
typedef unsigned int elf32_word;
typedef signed int elf32_sword;
typedef unsigned short elf32_half;
typedef unsigned int elf32_off;

struct elf32_ehdr{
    unsigned char e_ident[16];
    elf32_half e_type;
    elf32_half e_machine;
    elf32_word e_version;
    elf32_addr e_entry;
    elf32_off e_phoff;
    elf32_off e_shoff;
    elf32_word e_flags;
    elf32_half e_ehsize;
    elf32_half e_phentsize;
    elf32_half e_phnum;
    elf32_half e_shentsize;
    elf32_half e_shnum;
```

```
elf32_half e_shstrndx;
};
```

因为 ELF 可执行程序格式可以支持不同位数的处理器，所以 ELF 标准中自定义了一些专有的数据类型。无论是 8 位的处理器还是 32 位的处理器，这些数据类型的大小都是一致的，从而保证了文件与处理器格式的无关性。这些数据类型的大小和含义如表 8-1 所示。

表 8-1 ELF 格式中的数据规定

数据类型	大小	对齐	含义
elf32_addr	4	4	用于描述程序运行地址
elf32_word	4	4	描述无符号大整数
elf32_sword	4	4	描述有符号大整数
elf32_half	2	2	描述无符号中等大小的整数
elf32_off	4	4	描述无符号的文件偏移量

让我们结合表 8-1 的描述，逐个分析一下 ELF 文件中各部分的含义。

一个 ELF 文件头总是出现在文件的最开始处。其中，第一个成员是一个 16 个字节数组，里边记录了文件的标识、版本、编码格式等信息。

之后的两个字节是 `e_type` 成员，记录了目标文件属于 ELF 格式标准下的哪种类型，比如是可执行的文件还是可重定位文件，等等。根据 ELF 格式标准，这个值是 2，代表了这个文件是一个可执行的文件，这也正是我们需要的。

接下来的两个字节 `e_machine` 描述的是该程序运行的硬件平台。在我们的例子中，这个值必须是 40，表示这个应用程序是在 ARM 中运行的。

然后，4 个字节的空间 `e_version` 用于描述应用程序的版本，通常可以是 1。

接下来的 4 个字节就相当重要了，结构体成员 `e_entry` 记录了程序运行时的入口地址，也就是说，应用程序的第一条指令就应该出现在这个地址处。

`e_phoff` 成员记录了第一个 Program 头在文件内的偏移。`e_phentsize` 成员则代表了每一个 Program 头大小，再结合能够描述文件中共有多少个 Program 头的 `e_phnum` 成员，我们就可以遍历每一个 Program 头，并可以从中找到代码和数据在文件中的位置。

代码 8-4 中的其他成员与程序的执行关系不大，这里就不多介绍了。读者朋友们如果还对这些内容感兴趣，可以去查阅相关文档。

另外，还有一个问题需要解决。在遍历每一个 Program 头时，如何才能知道这个头信息所描述的 Segment 就是数据或代码，而不是与运行程序无关的其他信息呢？我们在 Program 头结构体中可以找到答案。

代码 8-5

```
struct elf32_phdr{  
    elf32_word p_type;  
    elf32_off p_offset;  
    elf32_addr p_vaddr;  
    elf32_addr p_paddr;  
    elf32_word p_filesz;  
    elf32_word p_memsz;  
    elf32_word p_flags;  
    elf32_word p_align;  
};
```

代码 8-5 定义了 `elf32_phdr` 结构体用于描述 Program 头信息。在该结构体中，与段类型直接相关的是 `p_type` 成员，只有当该成员的值为 1 时，才表示该 Segment 是要运行的代码或数据，需要在执行时加载到内存中去。

一旦确定需要进行内存加载，之后要做的就是读取 `p_offset` 成员的值，它表示要加载到内存中的这个 Segment 在文件中的偏移量，再结合描述 Segment 大小的成员 `p_filesz`，就可以精确地定位程序的代码和数据了。

最后，程序还要读取 `p_vaddr` 成员，它表示这个 Segment 应该出现在内存的哪个位置。这样就可以将该 Segment 从文件中复制到正确的内存地址处了。

8.2.2 操作 ELF 格式文件的方法

综合以上的描述，总结执行 ELF 格式文件的方法，步骤如下：

- (1) 从文件起始位置读取一个 `struct elf32_ehdr` 结构体，验证文件的正确性以及文件与操作系统是否匹配。
- (2) 找到该结构体中 `e_entry` 成员，从系统中获得这个值所指向的内存地址。
- (3) 读出 `struct elf32_ehdr` 结构体中的 `e_phoff`、`e_phentsize` 以及 `e_phnum` 三个成员。根据这三个值，利用 `struct elf32_phdr` 结构体遍历文件中每一个



Program 头。

(4) 在遍历的过程中，检查 struct elf32_phdr 结构体中的 p_type 成员，如果为 1，则调用存储设备的相关函数，将文件内偏移为 p_offset、大小为 p_filesz 的一段数据从存储器中读取到 p_vaddr 所指向的内存位置。

(5) 调用执行函数，使程序从 e_entry 内存处开始执行。

这样，一个 ELF 格式文件的执行过程就可以顺利完成了，将上述步骤用程序来实现，如下：

代码 8-6

```
void plat_boot ( void ) {
    int i;
    for ( i=0; init[i]; i++ ) {
        init[i] ( );
    }
    init_sys_mmu ( );
    start_mmu ( );
//    timer_init ( );
    init_page_map ( );
    kmalloc_init ( );
    ramdisk_driver_init ( );
    romfs_init ( );

    struct inode *node;
    struct elf32_phdr *phdr;
    struct elf32_ehdr *ehdr;
    int phnum, pos, dpos;
    char *buf;

    if (( buf=kmalloc ( 1024 ) ) == ( void * ) 0 ) {
        printk ( " get free pages error\n" );
        goto HALT;
    }

    if (( node=fs_type[ROMFS]->namei ( fs_type[ROMFS], " main " \
)) == ( void * ) 0 ) {
        printk ( " inode read error\n" );
        goto HALT;
    }

    if ( fs_type[ROMFS]->device->dout ( fs_type[ROMFS]->device, buf, \

```



```
    fs_type[ROMFS]->get_daddr( node ), node->dsize ) ) {  
    printk( "dout error\n" );  
    goto HALT;  
}  
  
ehdr= ( struct elf32_ehdr * ) buf;  
phdr= ( struct elf32_phdr * )( ( char * ) buf+ehdr->e_phoff );  
  
for ( i=0; i<ehdr->e_phnum; i++ ) {  
    if ( CHECK_PT_TYPE_LOAD( phdr ) ) {  
        if ( fs_type[ROMFS]->device->dout( fs_type[ROMFS]->device, \  
            ( char * ) phdr->p_vaddr, \  
            fs_type[ROMFS]->get_daddr( node ) + \  
            phdr->p_offset, phdr->p_filesz ) <0 ) {  
            printk( "dout error\n" );  
            goto HALT;  
        }  
    }  
    phdr++;  
}  
  
exec( ehdr->e_entry );  
HALT:  
    while( 1 );  
}
```

在代码 8-6 中，程序首先通过 romfs 文件系统的 namei 函数读取 RAM 盘上的 main 文件，得到代表该文件的 inode 结构体。不同于代码 8-3 中的“main.bin”文件，这里的 main 文件是直接由编译器编译生成的 ELF 格式文件。于是，我们可以依据执行 ELF 程序的一般步骤对它进行处理。

首先要做的就是通过存储设备的 dout 函数从文件系统中读出 main 文件的内容，并保存到 buf 中。为了保证程序简单直观，这里我们没有验证缓冲区的大小是否足够装下 ELF 和 Program 头信息，程序仅仅通过 kmalloc 函数申请了一个 1K 大小的内存来存储文件开头的部分。这样做，运行本书的例子至少是没有问题的。

接下来，我们需要找到描述 ELF 头信息的结构体 struct elf32_ehdr 的位置，并通过读取它的 e_phoff 成员找到第一个 struct elf32_phdr 结构体。这两个结构体的地址，分别被保存在变量 ehdr 和 phdr 中。

然后，程序从 `phdr` 变量开始，通过循环读取每一个 `Program` 头信息，依次判断 `struct elf32_phdr` 结构体的 `p_type` 成员是否为 1。如果该成员为 1，则表示此 `Program` 头所描述的正是代码段或数据段。此时需要再次调用 `dout` 函数，将用户应用程序的代码或数据从存储设备中复制到内存里。这两个信息分别记录在了 `struct elf32_phdr` 结构体的 `p_offset` 和 `p_vaddr` 结构体的成员中。

当所有的代码和数据最终都被加载到内存的正确位置后，就可以调用 `exec` 来运行用户应用程序了。用户程序的入口地址可从 `struct elf32_ehdr` 的 `e_entry` 成员中得到。

有了这样的方法，操作系统在运行用户程序时，便不再需要了解用户程序的细节。而仅需从应用程序中读出与程序运行有关的信息，根据这些信息的提示准备好程序运行的环境。这样，理论上就实现了运行任意二进制应用程序的可能。

然而从另一个角度来看，这种运行应用程序的方法还存在一个致命的缺陷，那就是我们无法保证用户应用程序的运行地址恰好是有效的。

这需要分两种情况去分析。第一种情况是，用户应用程序的运行地址已经超出了物理内存的范围，例如，一个 ELF 格式的用户应用程序需要运行在内存为 `0x40000000` 的地址中，但我们的虚拟系统，不计算 RAM 盘所占用的内存空间，一共只有 8M 的可用内存，`0x40000000` 这个地址远远超过了硬件原有内存的范围。第二种情况是，用户应用程序的运行地址虽然落在了物理内存的范围内，但这个地址却恰好被别的应用程序提前占用了。无论哪一种情况发生，都会使我们的操作系统将一个原本可以正常运行的用户应用程序拒之门外。

想要解决这个问题，可以通过虚拟内存映射将原本无效的内存地址映射到有效空间中。于是，我们只需要将应用程序的代码和数据保存到任意一个没有被使用的有效内存中，然后修改页表，将这个内存地址映射到用户应用程序规定的地址。由此，读者也可以深入地体会一下虚拟地址映射对于一个功能强大的操作系统来说是多么的重要。

当然，想要实现这样的功能，我们的操作系统代码需要进行很大的调整，这其中至少包含内存管理部分和 MMU 部分两个子结构。为了不占用过多的篇幅，这段代码我们就不去具体实现了。读者如果感兴趣的话，可以尝试自

行实现。

下面我们来实践一下这段代码。

8.2.3 运行 ELF 格式的应用程序

首先我们需要提供一些与 ELF 格式有关的宏定义。

代码 8-7

```
#define ELFCLASSNONE 0
#define ELFCLASS32    1
#define ELFCLASS64    2
#define CHECK_ELF_CLASS (p)          ((p)->e_ident[4])
#define CHECK_ELF_CLASS_ELFCLASS32 (p) \
(CHECK_ELF_CLASS (p) ==ELFCLASS32)

/*definition of elf data*/
#define ELFDATANONE      0
#define ELFDATA2LSB       1
#define ELFDATA2MSB       2
#define CHECK_ELF_DATA (p)           ((p)->e_ident[5])
#define CHECK_ELF_DATA_LSB (p) \
(CHECK_ELF_DATA (p) ==ELFDATA2LSB)

/*elf type*/
#define ET_NONE           0
#define ET_REL            1
#define ET_EXEC           2
#define ET_DYN            3
#define ET_CORE           4
#define ET_LOPROC         0xff00
#define ET_HIPROC         0xfffff
#define CHECK_ELF_TYPE (p)        ((p)->e_type)
#define CHECK_ELF_TYPE_EXEC (p) \
(CHECK_ELF_TYPE (p) ==ET_EXEC)

/*elf machine*/
#define EM_NONE           0
#define EM_M32            1
#define EM_SPARC          2
#define EM_386             3
```

```

#define EM_68k      4
#define EM_88k      5
#define EM_860      7
#define EM_MIPS     8
#define EM_ARM      40
#define CHECK_ELF_MACHINE (p)      ((p)->e_machine)
#define CHECK_ELF_MACHINE_ARM (p) \
    (CHECK_ELF_MACHINE (p) == EM_ARM)

/*elf version*/
#define EV_NONE      0
#define EV_CURRENT   1
#define CHECK_ELF_VERSION (p)          ((p)->e_ident[6])
#define CHECK_ELF_VERSION_CURRENT (p) \
    (CHECK_ELF_VERSION (p) == EV_CURRENT)

#define ELF_FILE_CHECK (hdr) (((((hdr)->e_ident[0]) == 0x7f) && \
                           (((hdr)->e_ident[1]) == 'E') && \
                           (((hdr)->e_ident[2]) == 'L') && \
                           (((hdr)->e_ident[3]) == 'F')))

#define PT_NULL      0
#define PT_LOAD      1
#define PT_DYNAMIC   2
#define PT_INTERP    3
#define PT_NOTE      4
#define PT_SHLIB     5
#define PT_PHDR      6
#define PT_LOPROC    0x70000000
#define PT_HIPROC    0x7fffffff
#define CHECK_PT_TYPE (p)      ((p)->p_type)
#define CHECK_PT_TYPE_LOAD (p)  (CHECK_PT_TYPE (p) \
                                == PT_LOAD)

```

代码 8-7 中，有一部分内容我们曾经使用过，但大部分宏在本书的代码中都未曾用到。这些宏都是根据 ELF 文件格式标准定义的，具有一定的通用性。朋友们也可以将这些宏用到自己的代码之中。

现在，读者朋友们可以将代码 8-7 与代码 8-5、代码 8-4 的内容共同保存到一个文件之中，命名为“elf.h”。

然后修改“boot.c”文件，先在文件的开始处将“elf.h”文件包含进来，

再把 `plat_boot` 函数修改为代码 8-6 那样。

这样，操作系统部分的代码就算完成了，读者可以尝试编译一下程序，确保没有语法错误。

接下来，我们还需要重新制作一个 RAM 盘存储设备映像。进入 `tools` 目录，将编译生成的测试应用程序 `main` 复制到 `tools` 中的 `filesystem` 目录。使用工具制作一个新的 `romfs` 文件系统类型文件，并将这个文件复制到上级目录的“`ram.img`”文件中。

现在回到我们的操作系统根目录，启动虚拟机，运行操作系统。不出意外的话，您将看到与上一节相同的结果。

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEye: use arm920t mmu ops
Loaded RAM ./leeos.bin
Loaded RAM ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is a test application
```

8.3 系统调用

现在，我们的操作系统已经拥有了运行和管理用户应用程序的能力，那些五花八门的功能和产品需求都可以交给用户应用程序去完成。此刻，我们的操作系统距离最终的无所不能，只差最后一步了，而这一步就是系统调用。

8.3.1 用户和内核的运行空间

想要讲清楚系统调用的来龙去脉，我们需要首先来说一说另外一对概念，那就是用户空间与内核空间。

这里的空间，可以简单地理解成运行环境。所以，我们可以简单地将用

户空间解释成为用户应用程序的运行环境，而内核空间则代表了操作系统所运行的环境。

通常对于单内核结构的操作系统来说，操作系统自身将会运行在内核空间，它拥有对整个硬件平台的绝对控制权。因此，当一段程序运行在内核空间中，既可以说明该程序应属于操作系统内核的一部分，又说明这段程序可以毫无限制地对计算机为所欲为。

与之相对的就是普通应用程序了。操作系统的作用是为应用程序提供必要的服务和支持，同时也会进行适当的监督和控制。而应用程序的行为对于操作系统来说是未知的，也就是说操作系统并不知道应用程序将做什么事。这样，当一个应用程序企图破坏整个系统的正常秩序、恶意操纵和控制其他应用程序时，就会产生极其严重的后果。

引入用户空间这种思想正是为了解决这一问题。一个用户应用程序从诞生那天开始就只能工作在用户空间，而处在用户空间的程序只有在自己的一亩三分地里才可以作主，决不允许多管别人的闲事。

这样，操作系统将自己运行在内核空间，而将应用程序限制在用户空间。同时，操作系统还将 CPU、内存、外设等几乎所有的资源都划归自己直接去管辖，不给用户空间直接访问的权力。此时，一个普通的用户应用程序即使有天大的本领，也不可能惹出什么乱子来，从而保证了各应用程序之间公平有序地运行，更好地去实现其功能。

但如此一来，当某个应用程序确实需要合理地使用系统中某些资源时，就会变得很困难。

如果这种情况真的发生，应用程序就不得不首先向操作系统做出申请。操作系统在确定应用程序是出于正当目的后，就会尽最大努力满足用户的要求，也只有这样，才能实现系统安全和运行效率之间的平衡。

于是，我们可以将用户应用程序向操作系统提出申请的这一动作叫做系统调用。

系统调用的实施过程可以通过图 8-2 进行描述。

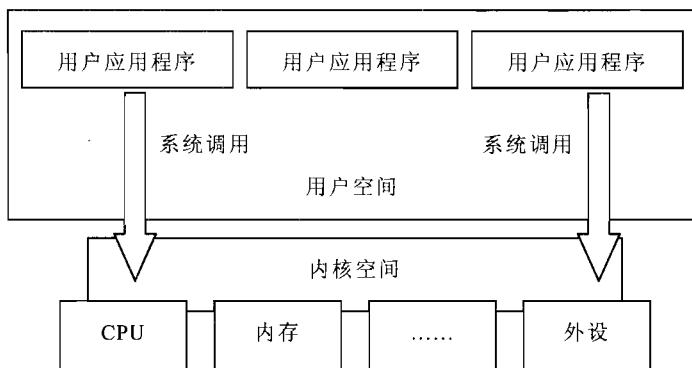


图 8-2 系统调用示意图

这里我们还需补充的是，图 8-2 并不能代表所有高端操作系统的系统调用结构。有些操作系统，如 Windows，甚至连内核自身也不具备控制所有资源的能力。可以说，图 8-2 的描述只适用于单一内核的高端操作系统。

有的读者可能会思考这样一个问题，操作系统怎样能够保证运行于用户空间的代码就一定不具备访问系统资源的能力呢？或者可以这样说，当某个应用程序硬是要强行访问系统或别的应用程序的资源，我们拦得住吗？

这个问题的答案是，拦不住。无论操作系统程序想什么样的办法，都没有能力彻底限制用户程序去做它想做的事。因为从 CPU 的角度上看，操作系统程序与用户应用程序同为代码，本没有任何区别。

正是因为只有 CPU 才有权力去限制程序的运行，所以这种用户空间和内核空间的划分，本质上讲是由 CPU 来完成的。CPU 只是让操作系统默认出生在了具有特权的内核空间，然后授权操作系统，让操作系统去管理用户程序，使用户程序生长在没有特权的用户空间，这样就自然而然地限制了运行在用户空间的应用程序。

回忆一下 ARM 体系结构的 7 种模式。在这 7 种模式中，用户模式与其他 6 种模式相比，不具备访问系统资源的能力，如全局中断控制等。因此，用户模式就成为了限制应用程序越权的天然屏障。再比如 x86 系列 CPU，保护模式下特权级拥有 4 个 Level，其中运行在 Level 0 中的代码享有至高无上的权力，适合作为内核空间运行操作系统程序，而运行于 Level 3 中的代码，其能力却极为有限，适合作为用户空间来运行用户应用程序。

既然用户模式总是工作在低特权的环境中，那么一旦一个应用程序确实需要使用系统资源时，就不得不想办法从低特权模式跳跃到高特权模式，然后才能去实施。这也成为了系统调用的另外一个特点。

因为低特权级的程序没有办法直接调用高特权级的程序，所以系统调用的实现不能简单依靠函数间调用，而必须使用别的方法。在多数体系结构中，操作系统都是通过软件中断来实现系统调用的。

8.3.2 实现一个系统调用

接下来，我们不如也来尝试实现一个系统调用接口。一旦系统调用能够轻松实现，那么在用户空间封装标准库函数就不再困难，一个结构完整的操作系统也将指日可待。无论从哪个角度看，这都是非常有意义的。

8.3.2.1 软中断的使用

为了实现一个系统调用，我们需要给目前的操作系统内核动手术。本质上讲，系统调用的内核实现其实就是软中断的处理问题。想要使系统调用得以运行，首先就要激活软中断。于是，我们首先要做的就是修改与异常向量有关的代码。一个最简单的软中断向量表的形式如下：

代码 8-8

```
_vector_SWI:
    stmfd r13!, {r0-r3,r14}
    ....
    <handler code>
    ....
    ldmfd r13!, {r0-r3,pc}^
```

代码 8-8 是软中断的一个简单结构。当软中断发生时，经过异常向量表的索引，程序会跳转到标签 `_vector_SWI` 处运行。在此处程序首先将参数寄存器 R0~R3 的值保存到堆栈之中，同时将上一个状态的返回值也压入堆栈。

回忆一下前面我们讲过的内容，不同异常发生时，返回值的修正方法是不同的。对于软中断来说，寄存器 R14 保存的恰是上一个状态的返回值，因此不需要修正。当软中断的代码操作完成后，程序依次将堆栈中的值恢复到寄存器中，返回到原状态继续运行。



这段软中断的实现方法虽然简单，但是用在一个功能强大的操作系统当中却并不合适，原因主要有下面两点。

第一，由于异常模式在切换时默认是关闭中断的。因此当系统调用发生时，程序在内核中运行，将不能够被其他进程，甚至是中断处理程序所中断。这样一来，程序就失去了其实用价值。

第二，软中断默认工作在管理模式，而我们的操作系统却是运行在系统模式下的。这就意味着程序不得不首先切换到系统模式，然后才能执行用户请求的相关动作，这势必会影响系统调用的执行效率。

因此针对 ARM 体系结构，使用一些更高级的方法实现软中断对于系统调用来说更加有效。

下面这段代码给出了一种实现方法。

代码 8-9

```
vector_SWI:  
    str r14, [r13, #-0xc]  
    mrs r14, spsr  
    str r14, [r13, #-0x8]  
    str r0, [r13, #-0x4]  
    mov r0, r13  
    CHANGE_TO_SYS  
    str r14, [r13, #-8]  
    ldr r14, [r0, #-0xc]  
    str r14, [r13, #4]  
    ldr r14, [r0, #-0x8]  
    ldr r0, [r0, #-0x4]  
    stmfd r13!, {r0-r3, r14}  
    ldr r3, [r13, #24]  
    ldr r0, [r3, #-4]  
    bic r0, r0, #0xffff000000  
    ldr r1, [r13, #32]  
    ldr r2, [r13, #36]  
    bl sys_call_schedule  
    str r0, [r13, #28]  
    ldmfd r13!, {r0-r3}  
    ldmfd r13!, {r14}  
    msr cpsr, r14  
    ldmfd r13!, {r14, pc}
```

在代码 8-9 中，一开始程序运行在管理模式，为了能够正确处理软中断，

程序必须想办法在第一时间切换到系统模式中去。因此，我们需要将用户模式下的状态和返回地址保存到管理模式下的堆栈中。

之后我们还需要将管理模式下的堆栈指针保存到 R0 寄存器中，以便在系统模式下也可以访问到管理模式堆栈中的内容。因为一旦程序切换到系统模式后，管理模式的堆栈指针寄存器 R13 将不可访问。

当一切准备工作都完成后，程序通过一个叫做 CHANGE_TO_SYS 的宏切换到系统模式，该宏的实现方法如下：

代码 8-10

```
.macro CHANGE_TO_SYS
    msr cpsr_c, # (DISABLE_FIQ|DISABLE_IRQ|SYS_MOD)
.endm
```

一旦程序切换到系统模式，首先需要做的就是保存系统模式下的 R14 寄存器，以防止子程序调用时子程序的返回值将原来存储在 R14 寄存器中的值改写。

紧接着，程序利用 R0 寄存器依次读出用户模式的返回值和状态信息，它们也将保存到系统模式中。这样一来，程序就可以从系统模式中直接返回到调用之前的状态，而不需要再度切换回管理模式了。

于是这一段代码运行完之后，系统模式下的堆栈数据就如图 8-3 (a) 所示（图 8-3 (b) 为完成 8.3.2.2 节的过程后的程序堆栈）。

代码 8-9 接下来的工作就比较关键了。程序先将当前堆栈向高地址偏移 24 个字节处的数据读到寄存器 R3 中。然后再以这个值为地址，取该地址前面的 4 个字节的数据保存到 R0 中。读者可以再看一下图 8-3 (a)。系统模式下的 R13 向高地址偏移 24 个字节，此处存储的正是用户模式的返回地址。结合前面章节的描述，这用户模式的返回值，记录的不正好是软中断指令的下一条指令地址吗？那么如果将这个地址向低地址处偏移的 4 个字节内容读出来，那不正好就是软中断指令自身吗？于是，上述动作完成后，R0 中保存的就是软中断指令的机器码。

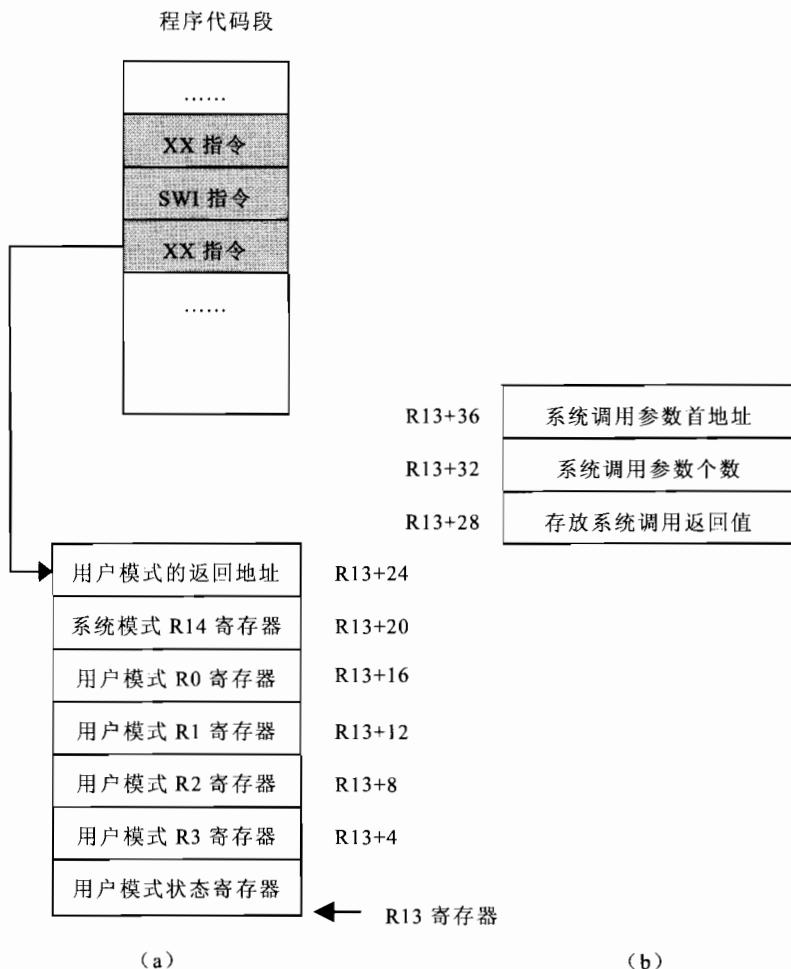


图 8-3 系统调用过程中的堆栈设计

但问题接着就来了，我们费这么大劲儿得到软中断的机器码又有什么意义呢？

这就需要从 ARM 体系结构中的软中断指令结构说起了。

众所周知，ARM 指令集中的指令都是 32 位等宽的，软中断指令自然也不例外。ARM 的软中断指令 SWI 由两部分组成，其机器码格式为 0xEFXXXXXX。由此可知，SWI 这条指令的高 8 位为 0xEF，其值是固定的，低 24 位则可以是任意数值，不会影响该指令的运行。而这个 24 位数，通常会被当成是软中断的参数在软中断发生时传递给处理程序。

因此，我们可以利用前面所描述的方法将这 24 位数值读出来，然后提取出软中断调用参数，并根据这个参数去完成不同的功能。在系统调用过程中，这个参数也就直接或间接地成为了标识系统调用号的重要手段。

系统调用号就是系统调用的代号，绝大多数高级操作系统都不只有一个系统调用。这么多的系统调用应该如何管理？很显然数字编号是最简单的一种方式。同时，由于多数操作系统会在系统调用的基础之上再封装一些函数接口供用户使用，这样一来，系统调用号所带来的使用上的不便也就不明显了。

现在回到程序当中。当系统能够正确得到系统调用号之后，程序又依次将 R13 寄存器向高地址偏移 32 个字节和 36 个字节的这两个数据从内存中加载到 R1 和 R2 两个寄存器中。

之后，程序调用了一个叫做 `sys_call_schedule` 的函数。毫无疑问，这里的 R1、R2 和前面的 R0 寄存器其实都是传递给 `sys_call_schedule` 函数的参数。其中，该函数的第一个参数就是系统调用号，而第二个和第三个参数是预先被存储到内存中的，我们稍后会解释。

`sys_call_schedule` 函数返回后，程序将寄存器 R0 的值保存到了 R13 向高地址偏移 28 个字节处。根据过程调用标准，函数返回后会将返回值保存到寄存器 R0 中。这也就说明保存到这段内存中的数据，恰好就是 `sys_call_schedule` 函数的返回值了。

最后，当寄存器 R0~R3、状态寄存器、系统模式下的 R14 寄存器以及用户模式的返回地址依次从堆栈中弹出时，系统调用过程结束，程序返回到用户空间当中。

8.3.2.2 为用户程序提供系统调用接口

系统调用的具体实现都是在内核空间中完成的，那么用户空间的应用程序又该做哪些工作才能成功触发某个系统调用呢？

正是通过代码 8-11 中 `SYSCALL` 这个宏，用户应用程序的系统调用申请才得以实现，下面我们就来简要分析一下这个宏的实现方法。

代码 8-11

```
#define __NR_SYSCALL_BASE 0x0
```



```
#define __NR_test          ( __NR_SYSCALL_BASE+ 0 )
#define __NR_SYS_CALL        ( __NR_SYSCALL_BASE+1 )

typedef int (*syscall_fn)( int num,int *args );

#define SYSCALL( num,pnum,parray,ret ) do{      \
    asm volatile( \
        "stmfd r13!,{%3}\n" \
        "stmfd r13!,{%2}\n" \
        "sub r13,r13,#4\n" \
        "SWI %1\n" \
        "ldmfd r13!,{%0}\n" \
        "add r13,r13,#8\n" \
        : "=r" ( ret ) \
        : "i" ( num ), "r" ( pnum ), "r" ( parray ) \
        : "r2", "r3" \
    ); \
}while( 0 )
```

我们先来聊一聊这个宏的实现结构。有的读者可能会对这种 `do{}while(0)` 的结构非常好奇，其实，这是一种保证宏定义能够在各种应用条件下都能成功编译的有效方法。假如，我们想定义一个由多个代码块组成的宏 MACRO，其形式如下：

```
#define MACRO first_step;second_step;third_step
```

那么如果某个程序员是下面这样的方式使用这个宏，就会造成编译错误。

```
if( condition )
    MACRO;
else
    do_something_else;
```

很显然，我们不可能强制要求用户在使用 MACRO 宏时必须在 if 结构中加上 {}。所以，为了尽可能减少类似问题造成的编译错误，就需要在定义宏的时候使用一个看似无用的 `do{}while(0)` 结构了。在这个例子中，如果宏 MACRO 被定义成这样：

```
#define MACRO do{first_step;second_step;third_step;}while( 0 )
```

那么经过预处理后，程序将展开成如下形式：

```

if ( condition )
    do{first_step;second_step;third_step;}while ( 0 );
else
    do_something_else;

```

这样一来，`do{}while (0)` 中包含的代码片段就会被编译器认为是一个完整结构，从而保证了程序的正确编译。

SYSCALL 宏最终是通过 C 语言内嵌汇编的方法实现了系统调用的请求的。有关在 GNU 环境下 C 语言内嵌汇编的具体方法，我们在前面的章节已经介绍过了，如果读者忘记了，可以翻看前面的内容。虽然 SYSCALL 宏是用汇编语言实现的，但却不难理解。程序是按照“压入堆栈，预留空间，触发系统调用，提取返回值，修正堆栈指针”的顺序完成系统调用申请的。

在整个过程中，最重要的应该是 SWI 指令。为了保证能够使用同一个系统调用接口调用操作系统内核中不同的函数，程序在使用 SWI 指令的同时，又将系统调用号传递到内核中去作为操作系统内核中不同函数的一个索引。

但即使这样，系统调用仍不足以发挥它的威力。举一个 UNIX 中最常用的例子，在类 UNIX 操作系统中打开一个文件是通过系统调用 `open` 来实现的。在 minix 下，`open` 的系统调用对应的号码为 5。我们可以假定，用户最终会通过类似于 SWI 5 这样的指令产生一个软中断，并且在内核中利用 5 这个软中断号最终索引到某一个 `open` 函数，但因为 `open` 系统调用实现的是打开文件的功能，所以用户在产生系统调用时，除了要使用到系统调用号之外，还必须想办法将一些参数传递给内核，如要打开的文件名以及要以什么样的方式打开，等等。

那么用户空间的程序应该通过什么方法，才能在系统调用时将参数传递给内核空间呢？

也许读者朋友们首先想到的就是过程调用标准。按照标准中的规定，我们可以将系统调用的参数通过寄存器传递到内核空间。然而，这种做法将会带来一个问题，那就是当系统调用参数多于可用寄存器个数时，程序的执行就会遇到麻烦。

所以如果有一种方法能够适应任意个数的参数传递，那么程序将变得更加易用。读者可能首先想到的就是通过变参函数的方法来实现。

的确变参函数完全可以解决这个问题，但使用变参函数的实现方法，程序运行的开销却比较大，而系统调用又是一个频繁的动作。这样，变参函数的处理器开销会因此被放大，进而影响系统整体的运行效率。

所以这里我们选取了一个折中的方法，将所有需要传递给内核的参数都强制转换成 32 位整型数，保存到一个数组当中。当用户调用 SYSCALL 时，只需将数组首地址和数组成员的个数分别传递给内核，内核一次性从数组中读出参数并进行处理。不同于变参函数可以支持任意类型的特点，我们的方法只能使用整型数据，从而在保证了传递任意参数的同时也简化了参数的传递开销。

于是在代码 8-11 中，程序首先将参数数组的首地址 parray 压入堆栈。紧接着，将系统调用的参数个数 pnum 也压入堆栈。请注意，SYSCALL 这个宏是内核专门给用户封装的，使用户能够方便地申请一个系统调用。因此，SYSCALL 中所有的代码都将运行在用户空间。这就表示 parray 和 pnum 这两个值会被压入用户模式下的堆栈中。

事实的确如此，但是我们却不需担心，因为用户模式和系统模式共享同一个堆栈。这样，操作系统就可以无缝地访问用户程序的任何数据了。

再次回到代码当中，接下来程序通过将堆栈指针寄存器的值减 1，在堆栈中预留了 4 个字节的内存空间，这段空间是专门用于保存内核系统调用实施函数的返回值的。

以上所有过程完成后，程序堆栈就变成了如图 8-3 (b) 所示的那样了。

之后程序使用 SWI 指令产生一个软中断，然后将跳转到代码 8-9 的 _vector_SWI 处去运行。

当控制权再次从内核中移交到用户空间之后，程序将内核系统调用函数的返回值从堆栈中读出，赋值给 ret，在对堆栈指针寄存器进行修正后，程序结束退出。

8.3.2.3 通用的系统调用函数

现在，代码 8-11 和代码 8-9 的功能都已经清楚了。系统调用也只剩下最后一个面纱，那就是代码 8-9 中的 sys_scall_schedule 函数。它代表了一个通用的系统调用函数。

代码 8-12

```
#include "syscall.h"

void test_syscall_args( int index,int *array ) {
    printk( "this following message is from kernel printed by
test_syscall_args\n" );
    int i;
    for( i=0;i<index;i++ ) {
        printk( "the %d arg is %x\n",i,array[i] );
    }
}

syscall_fn __syscall_test( int index,int *array ) {
    test_syscall_args( index,array );
    return 0;
}

syscall_fn syscall_table[ __NR_SYS_CALL ]={
    (syscall_fn) __syscall_test,
};

int sys_call_schedule( unsigned int index,int num,int *args ) {
    if( syscall_table[index] ){
        return (syscall_table[index])( num,args );
    }
    return -1;
}
```

函数 `sys_call_schedule` 共有三个参数，分别是代表系统调用号的 `index`，代表系统调用参数个数的 `num`，以及参数数组的首地址 `args`。在代码 8-9 中调用该函数之前，这些参数已经被压入到堆栈里了。

于是，`sys_call_schedule` 函数首先要读取 `syscall_table` 数组的具体值，用来判断是否为空，若不为空，就表示在系统调用号 `index` 处已经注册了一个系统调用函数。然后执行这个函数，返回该函数的运行结果，如果 `syscall_table` 数组的 `index` 索引处的相应函数指针为 `NULL`，那就说明该系统调用并没有注册，于是返回-1。

每一个系统调用都是 `syscall_fn` 类型的函数。该类型定义在了代码 8-11 中，系统调用号和最大系统调用号也在该处定义。对于一个成熟的操作系统来说，一系列精心设计的系统调用函数是必不可少的。在这里我们仅想通过

一个小例子来说明问题，不会对系统调用函数的设计做深入的研究。

程序中我们只实现了一个调用号为 0 的系统调用，它所对应的函数原形为 `_syscall_test`，这个函数仅仅实现了依次打印用户传递给内核的参数的功能以证明代码的正确性。

最后，程序还需要将所有的系统调用函数通过静态的方法保存到 `syscall_table` 数组中。

8.3.3 运行系统调用程序

一个系统调用的整个过程已经完成了，当然我们还需要亲自运行一下这段代码，才能更深入地理解系统调用的内涵。

首先，将代码 8-9 和代码 8-10 的内容添加到文件“abnormal.s”中，并将该文件中如下内容删除：



vector swi:
nop

然后将代码 8-11 的内容保存成文件，命名为“`syscall.h`”。同时将代码 8-12 的内容保存成文件，取名“`syscall.c`”。

最后，别忘记修改 `Makefile` 文件中的 `OBJS` 变量，添加 `syscall.o` 目标文件。

至此，操作系统内核中的系统调用代码就顺利完成了。由于没有用户的触发，这样的代码目前尚不具备运行的条件。我们还需要写一个用户应用程序，并在用户程序中使用系统调用接口触发内核中的 0 号系统调用函数。程序的实现过程如下：

代码 8-13

```
#include "../syscall.h"

int main() {
    const char *p="this is a test application\n";
    while(*p) {
        *(volatile unsigned int *)0xd0000020=*p++;
    }

    int test_array[2],ret;
```

```

test_array[0]=0xf0;
test_array[1]=0x0f;

SYSCALL( __NR_test, 2, test_array, ret );
}

```

不同于前面几小节的用户应用程序，在代码 8-13 中，程序使用了操作系统内核为我们封装好的 SYSCALL 宏，这个宏能够通过软中断指令将系统从用户模式切换到系统模式中，进而执行内核中的系统调用函数。

`__NR_test` 宏展开后的值是 0，表示调用内核中 0 号系统函数。接下来传递的是该系统函数所需要的参数个数和参数列表。最后，程序还传入了一个未初始化的变量 `ret`，用来存储内核系统调用函数的返回值。

这样，用户空间的系统调用代码就完成了。需要注意的是，在这段用户应用程序中，`SYSCALL` 和 `__NR_test` 等宏定义来源于操作系统源代码头文件“`syscall.h`”，我们可以在代码 8-13 中包含这个头文件，来保证用户使用的系统调用方法与内核代码中的定义方法一致。

现在请编译这段程序，并将生成的“`ram.img`”映像文件保存到我们的操作系统根目录中，然后切换到操作系统根目录下，编译操作系统代码，最后运行 `skyeye` 命令，如果一切顺利的话，您将会看到如下运行结果：

```

Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEye: use arm920t mmu ops
Loaded RAM ./le eos .bin
Loaded RAM ./ram .img
start addr is set to 0x30000000 by exec file.
helloworld
this is a test application
this following message is from kernel printed by
test_syscall_args
the 0 arg is 0xf0
the 1 arg is 0xf

```

这表示系统调用被正确运行了。



8.4 总结

总的说来，本章解决的是如何在操作系统中自由地运行外部应用程序的问题。这个问题很重要，但是却经常被忽视。很多嵌入式操作系统不重视或限制用户应用程序自由地发挥。这样造成的结果就是大量的应用不得不由操作系统开发人员亲自操刀完成，从长远的角度来看，这样的系统没有生命力。

因此，一个更好的结构是让别人帮助你去完成系统的应用。就像 Windows一样，操作系统的内核是由 Microsoft 开发的。但是，在 Windows 操作系统中的那些五花八门的应用程序则交给全世界的程序员去完成。同时，Microsoft 也在尽最大努力，通过提供编译器、丰富的文档、程序接口等手段，保证这些程序在 Windows 操作系统中能够可靠运行。

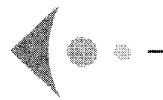
另一个例子是 Android。Android 其实就相当于一个 Linux 针对手机的发行版。既然是与 Linux 有关的开发，并且系统又是运行在手机这种处理器性能相对较低的硬件平台中，使用 C 语言开发应用程序应该是理所当然的事。但是 Android 却大量提供了以 Java 为基础的运行环境，力挺 Java 作为上层应用的开发语言，这么做的目的之一，就是让尽可能多的程序员去使用一种相对简单的编程语言，更加容易地为 Android 编写应用程序。

把应用程序交给第三方开发人员去完成，操作系统的开发者就可以更加专注于系统自身的提升，而不会被各种各样的应用需求搞得焦头烂额。更重要的一点是，依存于某款操作系统进行应用程序开发的人越多，对这种操作系统的依赖程度就越大，操作系统本身就越越来越不可替代。例如，在 PC 操作系统这个领域里，没有人能够取代 Windows 系列操作系统的地位，因为我们所熟悉的所有软件、操作方法、概念、规则标准等，统统都是基于 Windows 操作系统的。除非有一天 PC 不复存在了，否则 Microsoft 将会永远财源滚滚。要说这种运营战略有多高明倒不见得，这只不过是一种共赢的思想罢了，只不过鸡鸭鱼肉要进自己的肚子里，而那些剩菜剩饭，就分给应用程序的开发者好了。可悲的是，现在有相当多的一群人，还在为自己能够写出一些应用程序，赚点小钱而沾沾自喜呢！



第9章

进 程



一路走来辛辛苦苦，现在终于到进程这一步了。

进程管理、内存管理和文件系统管理是操作系统的三个最重要的功能。这其中，进程的重要程度非同一般。关于内存管理和文件系统管理两部分内容，前面章节中已介绍过了，本章我们将会揭开进程的面纱，一个操作系统的全貌将会最终呈现。

9.1 进程的实现原理

什么是进程呢？这是一个常识性的问题。通俗地讲，进程就是一个执行中的程序。一个应用程序放在存储设备中不用，它只能叫做程序。而当我们把这个程序读取到内存中并运行时，一个进程就产生了。

应用程序往往不是一个挨着一个去运行的，例如，一个人在浏览网页的同时，也在欣赏着美妙的音乐，可能还要时不时地跟在线的朋友聊上两句，无论是聊天程序、音乐播放程序还是网页浏览器，只要它们都处在运行的状态，那么这些进程就都是同步的。

除了并行执行外，进程通常还具有另外一个重要特点，那就是每个进程的地址空间都是独立的，比如，一个聊天的程序没有办法随意修改网页浏览器的变量，除非它们通过某种方法预先进行了沟通。



以上两个方面描述的其实是进程模型的两个基本属性，总结成一句话，就是从执行逻辑上看，进程是并行执行的，从内存空间上看，进程则是彼此独立的。

话又说回来，进程模型的这两个基本属性只是针对多任务操作系统来说的。对于那些单任务的操作系统，情况可就不一样了。DOS 是一个典型的单任务操作系统。想要在 DOS 系统中运行多个程序，只能是先运行一个，等到这个程序执行完毕，才能运行另一个。在这样的操作系统中，进程不存在并行的问题。

在了解了进程的一般特点后，我们就不得不去研究一个具体的问题了。这涉及到我们的操作系统进程部分的顺利实现。要知道一切硬件平台的 CPU 永远都是有限多个，而操作系统理论上却能够并行运行无限多个进程。怎样做到让有限个 CPU 同时执行无限多个程序呢？

这个问题其实只有一个解决办法，那就是宏观并行、微观串行。我们让每一个要执行的进程都只执行一段很短的时间，如 1 毫秒。在这段时间用完之后，不管这个进程是否执行完成，CPU 都将放弃执行这个进程，而选择别的进程去运行。这样，虽然从微观上看我们的进程依然是顺序执行的，但从宏观上审视运行的各个进程时，进程就变成并行执行的了。

这种思想是实现进程并行执行的最基本的方法。其中，让每一进程都执行一小段的这段时间，专业术语叫做“时间片”。而当一个进程的时间片耗尽，CPU 选择下一个进程去执行的这个动作，就叫做进程切换，整个过程可以用图 9-1 来描述。

接下来我们需要考虑的问题就是，一个进程怎么就可以在没有执行完的情况下，中途切换到别的进程去执行呢？寄希望于进程本身并不现实。这其实是指让系统中每一个进程在运行时，都要定时检查自己的运行时间是否超过了某个时间片，如果超过了运行时间片，就立刻放弃执行，这是绝对不可能的。

既然进程自身不可能定时地去检测超时，那么我们就让操作系统去做这个工作。不要忘了，几乎是所有的 CPU 都会通过内部或外部的方式提供某种定时机制。当某一特定时间到来时，硬件会触发一个中断，那么在中断处理程序中，系统将迫使 CPU 放弃执行一个进程，选择一个新的进程去运行。于是我们就得出了实现进程的一个非常重要的结论：依靠硬件时钟中断在中断处理程序中实现进程切换。



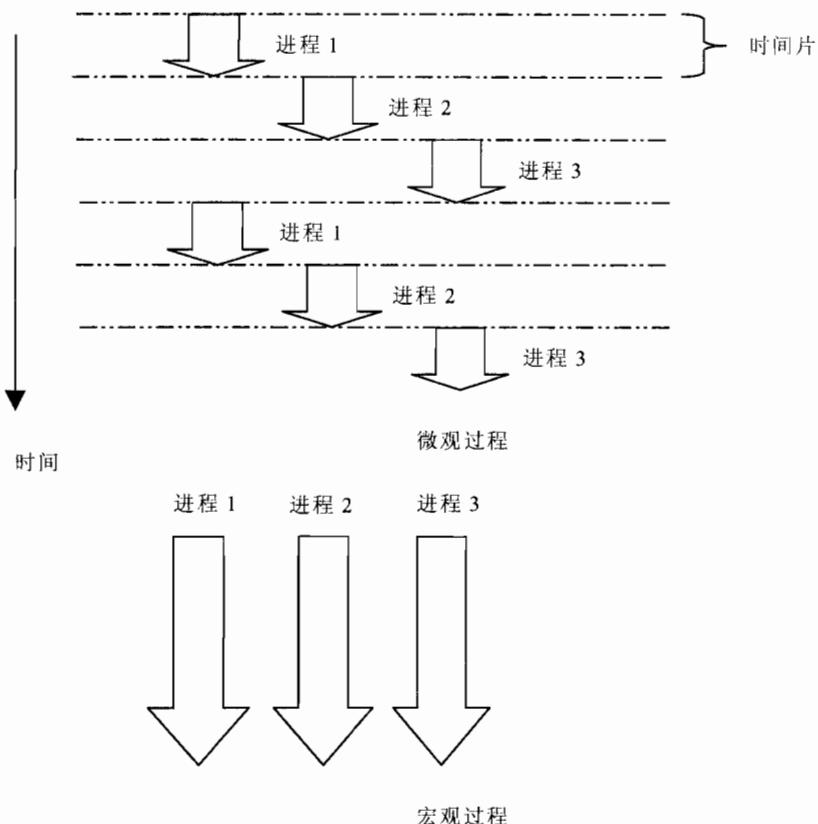


图 9-1 进程调度过程

在时钟中断处理程序中，系统需要做哪些工作才能保证进程顺利切换呢？读者朋友们应该都会很自然地想到，只要记录上一个进程切换时正在运行的那条代码的地址，同时将下一个要运行的进程所保存的地址读出来，再跳转过去运行，不就可以了吗？

当然就是这个意思，但过程却没有这么简单。进程在切换时，被终止的那条指令地址当然是必须保存的，除此之外，还包括所有寄存器、程序状态、程序堆栈、进程的地址空间……这个过程，叫做保护现场。于是，我们需要为每一个进程预留出一块内存空间，专门用于保存进程切换时的现场。很显然，这块内存区应该尽可能独立，不能让别的进程轻易地访问到。

另一个棘手的问题是，既然进程与进程之间需要完全隔离开，那么这就意味着每个进程所使用的堆栈必须彼此独立，这样才能保证进程在执行时，



彼此之间没有干扰。

还有，进程本身非常复杂、属性很多，需要使用自定义的数据类型来描述。在 C 语言中，结构体就是专门用来描述自定义类型的。于是通常的做法是定义一个结构体来描述每一个进程。当一个新的进程产生时，系统就需要为该进程分配一个结构体。

以上三个问题，本质上来讲，还是内存分配的问题。当一个新的进程产生时，系统就给它分配三块独有的内存区，一块用于存储进程的运行环境，另一块作为进程堆栈，还有一块用来存储描述进程的结构体。

其实我们完全可以在进程产生时，只给它分配一块空间，而让上述三种内存需求都在这一块内存空间中完成。这既是对程序编码的一种简化，也是对进程执行效率的一种提升。这种对进程的处理方法，一定程度上是源自 Linux 的。我们不妨先来了解一下 Linux 是怎么解决这个问题的。

在 Linux 系统中，当一个进程产生时，内核就会给该进程分配一个专有内存区。将这个专有内存区的一端作为堆栈首地址，而将描述进程的结构体保存到内存的另一端中。这段内存空间的大小在 Linux 中通常是 8K。当然，这个值绝不是 Linux 一拍脑门想到的。选择 8K 的原因是它恰好等于两个页的大小。这样，在内存分配时，系统就可以直接给进程分配一个 order 为 1 的两个页的空间，从而可以迅速获得内存。当然，这里也可以根据实际内存情况适当地选择 4K、16K 或其他尺寸。

同时，系统对这段内存空间还有一个硬性要求，那就是内存必须按照一个页的整数被对齐。举个例子来说，如果一个内存空间位于 0x30211000~0x30212000，那么，对于进程来说就是可用的，但如果内存的位置是 0x30325020~0x30326020，则该内存不能被进程使用。这是 Linux 的一个巧妙设计，原因我们稍后会详述。

Linux 将用于描述进程信息的结构体命名为 `thread_info`。于是，对于一个堆栈向下生长的体系结构来说，Linux 下一个进程的内存空间类似于图 9-2 所描述的那样。



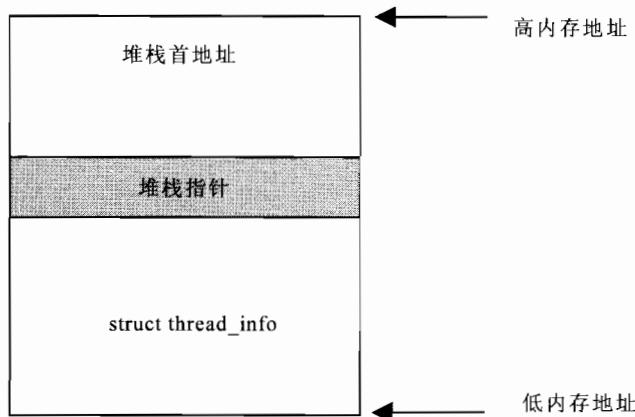


图 9-2 Linux 的进程内存

有的读者朋友可能会觉得这种设计似乎没有什么特别之处。但如果我们要获得描述当前进程的结构体时，这可能就是最快的办法了。不要忘了我们前面说过，分配给进程的这个 8K 内存空间是按照页的整数倍对齐的。这样，无论当前堆栈指针指向何处，我们都可以让它和 0xffffe000 做“与”操作，得到的结果将必然指向这段内存的低地址，也就是结构体 struct thread_info 的地址。

在 x86 体系下，这段代码的实现如下：

代码 9-1

```
movl $-8192,%eax
andl %esp,%eax
```

而在 ARM 体系结构中，获取结构体的地址可以使用下面这样的方式来完成：

代码 9-2

```
struct task_info *current_task_info ( void ) {
    register unsigned long sp asm ( " sp " );
    return ( struct task_info * )( sp&~( 8192-1 ) );
}
```

从这两段代码中我们可以看出，取出堆栈指针寄存器的值并通过“与”的操作来得到进程结构体的首地址，无论在哪种硬件平台下都是很方便的。

这样，在操作系统中实现进程的所有理论问题，我们都解释清楚了。事

实证明这些理论并不难理解。对操作系统原理稍懂一些的读者都知道进程切换要靠时钟中断来驱动，也知道进程在切换时需要保护现场。但落实到代码中，恐怕绝大多数人是会手足无措的。毕竟进程调度的实现逻辑不同于普通程序，是需要很多技巧的。

事不宜迟，下面就让我们从实践的角度出发，让进程也能在我们的操作系统中运行。

9.2 进程的实现

首先我们需要改写一下系统的中断处理方式，让原有的时钟中断能够负担起进程调度的责任。

9.2.1 改写中断处理程序

为了让程序看起来更清晰，我们删除了原来的中断处理程序，只保留了与进程切换有关的部分。

代码 9-3

```
—vector_irq:  
    sub r14,r14,#4  
    stmfd r13!,{r0}  
    stmfd r13!,{r1-r3}  
  
    mov r2,#0xca000000  
    add r1,r2,#0x10  
    ldr r0,[r1]  
    ldr r3,[r2]  
    orr r3,r3,r1  
    str r3,[r2]  
    str r0,[r1]  
  
    ldmfd r13!,{r1-r3}  
    mov r0,r14  
    CHANGE_TO_SYS
```

```

    stmfd r13!, {r0}
    stmfd r13!, {r14}
    CHANGE_TO_IRQ
    ldmfd r13!, {r0}
    ldr r14, =__asm_schedule
    stmfd r13!, {r14}
    ldmfd r13!, {pc}^

__asm_schedule:
    stmfd r13!, {r0-r12}
    mrs r1, cpsr
    stmfd r13!, {r1}

    mov r1, sp
    bic r1, #0xff0
    bic r1, #0xf
    mov r0, sp
    str r0, [r1]

    bl __common_schedule
    ldr sp, [r0]
    ldmfd r13!, {r1}
    msr cpsr,r1
    ldmfd r13!, {r0-r12,r14,pc}
}

```

代码 9-3 就是这样一段程序。首先，程序通过一条 sub 指令修正了返回时的地址。然后，将寄存器 R0 ~ R3 压入中断模式下的堆栈。虽然这段程序是针对进程切换而设计的，但我们同样可以看到，这些处理方法与通用的中断处理程序没有什么差别。

在接下来的一段代码中，我们需要做的就是清除掉 s3c2410 中断控制器与时钟有关的寄存器，避免中断不停产生。中断控制器的物理地址本应是从 0x4a000000 开始的，但由于我们的操作系统已经开启了 MMU，于是程序必须通过虚拟地址 0xca000000 作为基地址对控制器进行访问。如果读者对这段代码依旧似懂非懂，可以翻看前面中断一章的内容，那里有非常详细的讲解。

之后的这段代码便与进程调度有关了。程序首先把寄存器 R1 ~ R3 从堆栈中恢复出来。这里没有将寄存器 R0 弹出堆栈的原因是，我们需要通过 R0 来实现不同异常模式间的参数传递。于是在中断模式下，程序通过 mov



指令将寄存器 R14 的值保存到 R0 里。这也就意味着，上一个进程的返回地址此时被保存到了寄存器 R0 中。紧接着，程序通过 CHANGE_TO_SYS 宏切换到系统模式下，又将寄存器 R0 和系统模式下的 R14 压入了系统模式的堆栈之中。

读者可以仔细品味一下这种操作方法的真正用意，R0 是中断发生时上一个状态的返回值，但也可以说成是上一个进程被切换时将要执行的指令地址，而寄存器 R14 保存的则是上一个进程在某个函数调用时的返回地址。将上一个进程的这两个值压入的这块堆栈区，除了可以理解成是系统模式下的堆栈外，还可以理解成是上一个进程的堆栈，这只需要通过改写 R13 寄存器的值就可以实现。

于是我们得到了一个非常重要的结论：当时钟中断发生时，中断处理程序负责进程调度，首先要将上一个进程的要执行的那条指令的地址和函数返回值地址压入上一个进程的私有堆栈中，如图 9-3 所示。

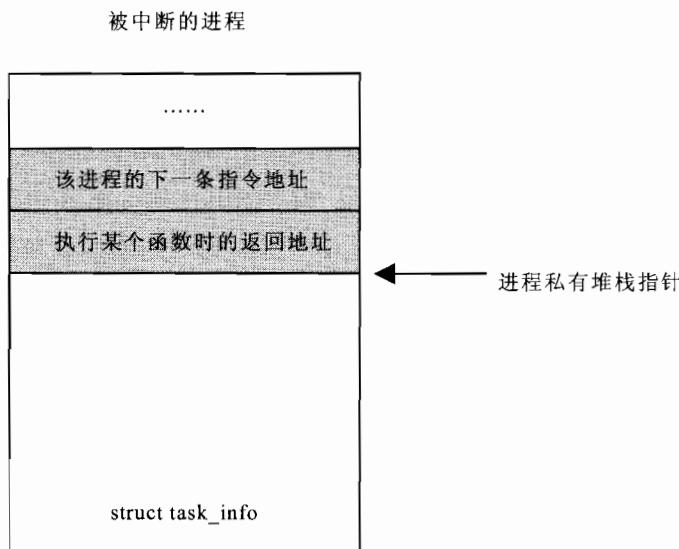


图 9-3 进程内存空间

在完成了上述操作之后，程序通过 CHANGE_TO_IRQ 宏返回到中断模式，然后从中断模式下的堆栈中恢复寄存器 R0 的值。到这一步为止，程序好像是什么都没有发生一样，又恢复到了刚刚发生中断时的样子。殊不知这



是“明修栈道，暗渡陈仓”，程序悄悄地从中断模式切换到系统模式，将两个关键值压栈，又悄悄地返回到了中断模式中。

之后的动作就有些巧妙了，函数`_asm_schedule`的地址被赋值给了R14寄存器，然后将该寄存器压入堆栈，紧接着又将它弹出堆栈。所不同的是，此时堆栈中的值不是恢复到寄存器R14里，而是直接恢复到程序计数器PC中，同时也将寄存器SPSR的值恢复到CPSR中去。

这个过程从本质上讲就是一个程序的跳转，只不过跳转的同时要伴有模式的切换。为什么会选择寄存器R14来实现这个过程呢？原因是中断模式下的R14是唯一一个空闲不用的寄存器。使用这种独特的跳转方法的另一个目的是，这样可以实现在程序跳转的同时更改运行模式。

于是，程序恢复到了中断之前的模式中，或者更贴切地说，是上一个进程所处的模式之中，与此同时，程序跳转到了函数`_asm_schedule`处开始运行。

这样，一段原本要从一个进程切换到另一个进程的程序似乎又回到了原来的进程之中了。

这种观点当然是错误的，此时虽然程序回归到原来的进程当中，但根本目的不是继续原来的进程运行，而是在原进程环境下保护进程现场。`_asm_schedule`函数的核心功能就在于此。

在`_asm_schedule`函数中，程序首先将原进程的R0~R12寄存器都压入到了该进程的堆栈之中，之后又借助寄存器R1将程序状态寄存器的值也压入堆栈。接下来，我们又将堆栈指针寄存器SP赋值给了R1，然后通过两次BIC指令清除掉R1寄存器的后12位。

朋友们是否已经意识到了这段程序的真正用意了？没错，此时R1寄存器的值恰好是进程所属内存区的低地址。回头看一下图9-2，大家就会完全明白了。最终，寄存器R1的值等于该进程结构体的地址。只不过我们没有效仿Linux，将这个结构体命名为`thread_info`，而是称之为`task_info`。该进程的当前堆栈指针就保存在这个地址中。如果读者还在疑惑，为什么一个结构体的地址可以用来保存堆栈指针的值，那么就请看一下`struct task_info`结构体的定义吧。

代码 9-4

```
struct task_info{
    unsigned int sp;
```

```
    struct task_info *next;  
};
```

通过代码 9-4 我们知道，`struct task_info` 结构体的第一个成员是一个无符号整型变量，名字叫做 `sp`，这个变量就是专门用来保存进程切换时的堆栈指针的。

这样，进程保护现场的工作就成功完成了，完成之后的进程堆栈如图 9-4 所示。

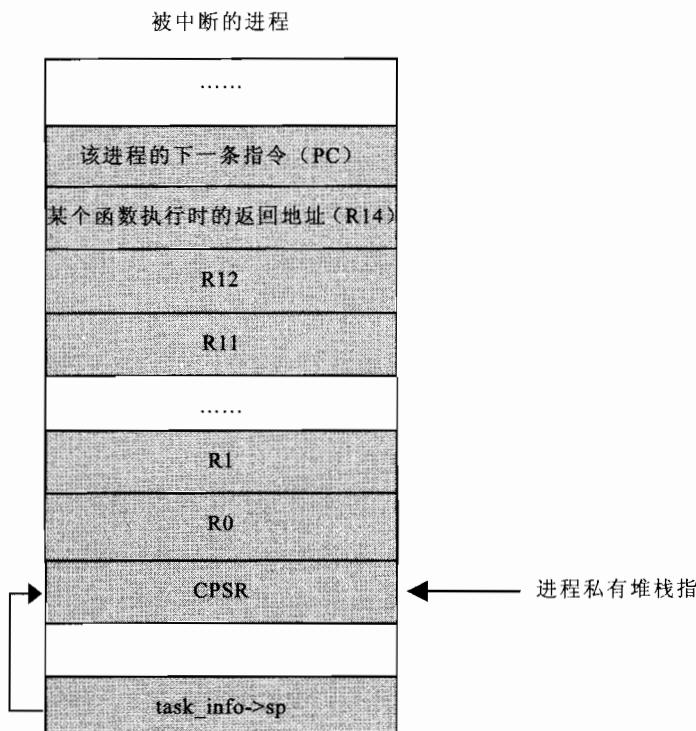


图 9-4 进程私有堆栈空间

9.2.2 抽象调度函数

虽然进程保护现场的过程结束了，但进程切换过程远没有结束。所以在代码 9-3 中，程序紧接着调用了一个名为 `_common_schedule` 的函数。这个函数与硬件平台无关，它并不需要在指定的硬件平台下运行。因此我们将该函数定义到了别处，并使用 C 语言去实现，这就是所谓的抽象调度函数。

抽象调度函数能够返回下一个进程的 struct task_info 结构体首地址，也就是下一个进程所属的内存空间的低地址。不要忘了，进程在切换时会将所有资源都保存到这段空间之中。于是，在得到这段内存空间的基址之后，我们就可以得到相应的进程资源，并将它们全部恢复。这样，进程切换就算完成了。

通过上面的描述，读者应该可以体会到这种抽象的真正含义，第一，这个函数本身与硬件无关，理论上独立于任何硬件体系结构之上；第二，只要这个函数的返回值不变，无论其实现方法是怎样的，都能够确保进程切换的正确性。

正是凭借这种抽象性，使我们能在今后的操作系统优化过程中更加容易地实现各种不同的进程调度算法，从这个角度来看，该函数的意义非常重大。

进程调度算法对于操作系统来说是非常重要的，简单说来，进程调度算法是用来决定某个进程是运行的时间长一点还是短一点的算法，它会直接影响到操作系统的运行效率。然而，本书将不会对进程调度算法进行深入研究，因为进程的问题一旦深入，就会引出源源不断的新话题，这不是本书的初衷。因此，在这里我们会采用一种最简单的进程调度算法来实现进程的切换，那就是基于时间片的轮询调度算法。

说白了，这种方法就是让所有待运行的进程挨个排队，每个进程运行一段固定的时间，循环往复直至程序结束。

轮询调度算法简单到什么程度，看一下 __common_schedule 函数的具体实现就清楚了。

代码 9-5

```
void * __common_schedule ( void ) {
    return ( void * )( current->next );
}
```

代码 9-5 仅仅将当前 current 的 next 成员返回。这里，current 代表的正是当前正在运行的进程，具体的定义方法我们稍后会介绍。我们知道，系统中的所有进程会在初始化时被连接成一个环形链表，每次取得当前进程的 next 成员就意味着程序在循环遍历链表中的所有进程，从而实现对进程的轮询调度。

一旦 __common_schedule 函数运行完成，寄存器 R0 的值就是下一个进



程的 struct task_info 结构体的地址了。并且根据前面的描述我们知道，这个进程在切换之前，也会将该进程的私有堆栈指针保存到 struct task_info 结构体的 sp 成员处。于是在代码 9-3 中，程序通过一条 ldr 指令成功地恢复了 R13 寄存器。

接下来，程序又将保存在堆栈中的 CPSR 寄存器的值恢复。最后将除 R13 寄存器外的所有其他寄存器的值依次恢复。当然，这其中也包括程序计数器 PC，而恢复给 PC 寄存器的正是进程被切换时恰要运行的那条指令。

至此，进程切换的所有工作就完成了。

9.2.3 新进程的产生

代码 9-3 为我们展现了多进程同时运行的一种工作方法。现在的问题是，系统从启动开始就是以单进程的形式出现的，我们应该怎样产生一个新进程并与原有的进程并行运行呢？

一些简单的操作系统在实现多进程时，会采用静态或半静态的方式。这种方法的实现思路是，要在这种系统中运行的进程，其功能和数量都是固定的，也就是说，用户进程的信息需要被写入到操作系统源代码中，每次当我们需要更改用户进程时，就必须重新编译系统。在这样的操作系统中，开新进程的方法虽然简单，但同时也带来一个问题，即操作系统与应用程序之间的耦合太紧密了，系统过分限制了用户，没能留给用户一个自由的环境去任意发挥。

所以在多数高级操作系统中，开新进程的工作往往会由函数来实现。这样，当一个用户想要以多进程的方式运行程序时，只需要调用系统提供的函数就可以了。熟悉 Linux 的朋友应该都会想到，Linux 下的 fork 系统调用就是这样一个函数。

考虑到操作系统的初设计，在我们的操作系统中，不妨也将开新进程的动作封装到一个函数之中来实现动态的进程管理。

如果读者理解了代码 9-3 的工作原理，那么开一个新进程的工作将不难实现。图 9-5 描述了生成一个新进程的四个基本步骤。



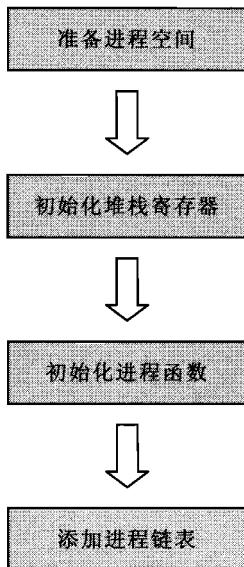


图 9-5 新进程产生的步骤

首先，我们需要为新进程开辟出一块内存空间，用来保存进程运行时需要的所有数据。结合图 9-3 我们知道，`struct task_info` 结构体就位于该空间的低地址端。

然后，我们要初始化新进程的堆栈寄存器。在这里，初始化寄存器指的并不是要真地去改写堆栈寄存器的值，而只是要初始化结构体 `struct task_info` 中 `sp` 成员的值。这样，一旦这个进程被调度，系统就会读取该值并赋值给寄存器 `SP`，就像代码 9-3 那样。那这个值应该是多少才正确呢？读者只需要参考一下图 9-4，就会知道答案。

接下来的工作是初始化进程函数。进程是一段执行着的程序，本质上还是一段代码。那么，当一个新的进程产生时，如何让它去执行既定代码？恐怕最简单的方法莫过于函数。我们把新进程将要完成的动作都写到一个函数中，然后将这个进程的入口地址压入进程堆栈中，等到进程切换时，系统将该地址恢复，于是该函数将以一个全新进程的方式去运行。

最后我们还需要将与新进程有关的数据信息保存起来。这样，当进程切换时，`__common_schedule` 函数就能够通过某种方法找到并返回该进程的 `struct task_info` 结构体，完成进程切换了。通常，系统都会选择链表这种方



式来保存进程信息，理论上这样可以支持任意多的进程同时运行。

将上述过程转换成代码，我们会得到：

代码 9-6

```
#define disable_schedule(x) disable_irq()
#define enable_schedule(x) enable_irq()

int task_stack_base=0x30300000;
struct task_info *copy_task_info( struct task_info *tsk ) {
    struct task_info *tmp=( struct task_info * )task_stack_base;
    task_stack_base+=TASK_SIZE;
    return tmp;
}

#define DO_INIT_SP(sp,fn,args,lr,cpsr,pt_base) \
do{ \
    (sp)=(sp)-4; /*r15*/ \
    *( volatile unsigned int * )( sp ) = ( unsigned int ) \
    ( fn ); /*r15*/ \
    (sp)=(sp)-4; /*r14*/ \
    *( volatile unsigned int * )( sp ) = ( unsigned int )( lr ); /*r14*/ \
    (sp)=(sp)-4*13; /*r12,r11,r10,r9,r8,r7,r6,r5,r4,r3,r2,r1,r0*/ \
    *( volatile unsigned int * )( sp ) = ( unsigned int )( args ); \
    (sp)=(sp)-4; /*cpsr*/ \
    *( volatile unsigned int * )( sp ) = ( unsigned int )( cpsr ); \
}while(0)

unsigned int get_cpsr( void ) {
    unsigned int p;
    asm volatile( \
        "mrs %0,cpsr\n" \
        : "=r" ( p ) \
        : \
        );
    return p;
}

int do_fork( int (*f)(void *),void *args ) {
    struct task_info *tsk,*tmp;
    if(( tsk=copy_task_info( current )) == ( void * ) 0 )
        return -1;
```

```

tsk->sp=({ unsigned int })( tsk ) +TASK_SIZE ;
DO_INIT_SP( tsk->sp,f,args,0,0x1f&get_cpsr( ),0 );

disable_schedule( );
tmp=current->next;
current->next=tsk;
tsk->next=tmp;
enable_schedule( );

return 0;
}

```

在代码 9-6 中，`do_fork` 就是一个进程产生函数，它是完全依照图 9-5 的描述实现的。通过调用这个函数，程序可以轻轻松松地动态实现一个新进程，我们现在就来分析一下。

函数 `copy_task_info` 负责分配进程空间，为简化程序考虑，我们仅仅是从 0x30300000 地址开始，依次分配 4K 大小的区域来实现这段内存空间分配函数的。对于我们这个示例操作系统来说不会造成什么影响，但却更加直观。而在实际的操作系统中，也只需使用页分配函数对此进行替换即可。

然后，函数通过宏定义 `DO_INIT_SP` 来实现堆栈寄存器的初始化以及进程函数的压栈工作。该宏压栈的顺序从高地址向低地址分别是进程入口地址、进程返回地址、R0 ~ R12 寄存器、函数参数、进程 CPSR，这些都是参考图 9-4 实现的。

对于一个新产生的进程来说，我们并不关心 R1 ~ R12 寄存器的具体值，但是也必须在堆栈中为这些寄存器留出空间来。而寄存器 R0 则负责为进程函数传递参数，因此程序需要将参数保存到 R0 寄存器所在的堆栈位置中。为了保证新进程的进程模式与调用 `do_fork` 函数的进程模式相同，我们通过 `get_cpsr` 函数获取上一进程的状态后，提取出后 5 位作为新进程的运行状态。

最后，为了能够让 `_common_schedule` 函数成功返回新进程的 `struct task_info` 结构体，我们需要将系统内的所有进程结构体连接成链表。于是，`do_fork` 函数接下来的工作就是将新进程的 `struct task_info` 结构体添加到当前进程之后。在我们的操作系统中，我们设计了一套单向链表来连接进程。

链表本身属于基本的数据结构范畴，其操作方法非常简单。但不寻常的地方在于，在操作链表之前，我们调用 `disable_schedule` 宏临时关掉了进程

切换，而完成链表操作后，又调用 `enable_schedule` 宏恢复进程切换。这其实是一个与进程并发和同步有关的话题，不在本书的讲授内容之中。简单地说，这样做的目的是防止进程链表操作过程正在进行时，进程发生切换带来未知的后果。

这样，函数 `do_fork` 就完成了产生新进程的所有工作。读者朋友们应该还会对 `current` 这个变量心存疑惑。这里为了代码编写方便，我们将 `current` 定义成代码 9-2 的 `current_task_info` 函数，这种方法来源于 Linux。完整的代码如下所示：

代码 9-7

```
#define TASK_SIZE 4096

struct task_info *current_task_info ( void ) {
    register unsigned long sp asm ( "sp" );
    return ( struct task_info * )( sp&~( TASK_SIZE-1 ) );
}

#define current current_task_info ( )
```

9.2.4 多个进程同时运行

能够让多个进程同时运行是一件非常激动人心的事情，我们马上就可以实现。

首先，我们需要用代码 9-3 的内容来改写“abnormal.s”文件的内容。

然后，将代码 9-4、9-5、9-6、9-7 以及下面这段代码 9-8 的内容保存成一个新的文件，名为“proc.c”。

代码 9-8

```
int task_init ( void ) {
    current->next=current;
    return 0;
}
```

之后我们还要修改“boot.c”文件，删除掉 `plat_boot` 函数，并将代码 9-9 的内容添加进去。



代码 9-9

```

int do_fork( int (*f)(void *),void *args);

void delay( void ) {
    volatile unsigned int time=0xffff;
    while( time-- );
}

int test_process( void *p ) {
    while( 1 ) {
        delay();
        printk( "test process %dth\n", ( int ) p );
    }
    return 0;
}

void plat_boot( void ) {
    int i;
    for( i=0;init[i];i++ ) {
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    task_init();
    timer_init();

    init_page_map();
    kmalloc_init();

    ramdisk_driver_init();
    romfs_init();

    i=do_fork( test_process,( void * ) 0x1 );
    i=do_fork( test_process,( void * ) 0x2 );

    while( 1 ) {
        delay();
        printk( "this is the original process\n" );
    };
}

```

在代码 9-9 中，程序首先调用了 task_init 函数。这个函数的功能只有一

个，那就是初始化原始进程结构体的链表结构，让 `next` 成员指向自身。这样，调用 `do_fork` 函数时，新的进程结构才能成功被添加。而后，我们两次调用 `do_fork` 函数，开辟两个新进程。这两个进程将会执行同一个函数 `test_process`，该函数仅仅是循环以十进制整数的形式打印进程函数参数。为了能在运行时对这两个进程做区分，我们分别在调用时传递了两个不同的参数供其打印。此时，原始进程仍在运行，我们让它循环打印 “`this is the original process\n`” 这段字符串，可以看出，这段代码就是对前面封装的进程函数的一系列简单调用。

最后，修改 `Makefile` 中的 `OBJS` 变量，将 “`proc.o`” 添加到目标文件列表中。

这样，我们的操作系统就可以编译运行了。不出意外的话，这段程序的运行结果将会是如下情况：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00ffff, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEye: use arm920t mmu ops
Loaded RAM ./le eos.bin
Loaded RAM ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is the original process
test process 2th
test process 1th
this is the original process
test process 2th
test process 1th
this is the original process
test process 2th
test process 1th
....
```

可以看出，三个进程依次被调度的情况出现了，我们的操作系统正在以多进程的方式运行！

9.3 总结

正如本章开头所讲的，进程永远都是操作系统的核心，这就是很多关于操作系统原理的书中都把进程作为开头的重点内容介绍的原因。

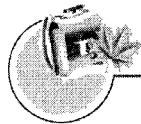
然而，本书的编排却恰恰相反，将进程的内容放在了最末尾。其中一个原因是，本书强调理论通俗易懂、重视实践易于操作。如果将进程的话题放到最前面，很多读者就有可能会在硬件体系结构、操作系统原理等知识都一知半解的情况下编写大量的代码，实现一个最终可能都不能完全理解的进程调度程序。当我们一步一步、循序渐进地掌握了从硬件到算法等各种基础知识后，再来实现多进程，也就水到渠成了。

一个操作系统是否就此就结束了呢？远远还没有。

往操作系统核心看去，一旦程序以并行的方式去运行，那么进程调度算法的问题、进程之间的互斥和通信问题、受进程影响的内存和文件系统等问题都会蜂拥而至，即便再用这么长的篇幅，也很难一一说清楚。

这些内容再结合本书整整九章的描述，就构成了一套完整的操作系统理论。从这个角度来看，进程可以成为本书的结束，也可以成为新的知识的开始。





结束语



至此，本书的内容就全部结束了，感谢读者的支持，能够跟随书中的脚步一点点地走到这里！

然而，本书虽然读完了，但这并不代表对操作系统的学习和研究就此终结。本书的结束标志着一个新的旅程的开始。

一方面，本书针对操作系统的结构虽然完整，但有很多细节我们并未涉及。这包含了嵌入式操作系统实时性问题、进程调度算法问题，进程间互斥和通信问题、多核问题、移植性问题，等等。本书只是为读者打开了一扇前往操作系统世界的大门。如果有可能，我也希望能够在另一本书里与大家再次见面，共同探讨那些未曾涉及的操作系统话题。

另一方面，读者虽然已经完成了属于自己的嵌入式操作系统，但这却仅仅是个开始。如果读者还有热情和精力，希望能够在如下两个方面尝试远行：
尝试继续完善自己的操作系统，完善核心代码、增加操作系统健壮性、移植函数库、实现用户程序的标准化等；尝试将已有的操作系统代码移植到实际的硬件平台中。

最后，欢迎读者访问 www.leeos.org，就书里书外的一切话题与本人进行交流！



参考资料



- [1]陈渝, 谌卫军. 操作系统设计与实现 [M]. [美]Andrew S.Tanenbaum, Albert S. Woodhull. 第 3 版. 北京: 电子工业出版社, 2008: 上册
- [2]邵贝贝. 嵌入式实时操作系统 uC/OS II [M]. [美]Jean J.Labrosse. 第 2 版. 北京: 北京航空航天大学出版社, 2003
- [3] 陈莉君, 康华, 张波. Linux 内核设计与实现 [M]. [美] Robert Love. 第 2 版. 北京: 机械工业出版社, 2006
- [4] 陈葆珏, 王旭, 柳纯录, 冯雪山. UNIX 操作系统设计 [M]. [美] Maurice J. Bach. 北京: 机械工业出版社, 2004
- [5] 沈建华. ARM 嵌入式系统开发——软件设计与优化 [M]. [美] Andrew N.Sloss, [英] Dominic Symes, [美] Chris Wright. 北京: 北京航空航天大学出版社, 2005
- [6] 杜春雷. ARM 体系结构与编程 [M]. 北京: 清华大学出版社, 2003
- [7] 邹恒明. 计算机的心智: 操作系统之哲学原理 [M]. 北京: 机械工业出版社, 2009

一步步写嵌入式 —ARM编



01762282

一次跟Android领军人物高焕堂先生聊天时，他的国外先进的开发工具、平台和操作系统就好比是武器，进武器去打仗（做应用层开发），一旦有一天我们跟外国人打起来，人家拿走我们的武器，我们就真的是一筹莫展了。

这句话很有道理，中国计算机技术整体水平的提高需要以大量自主研发的开发工具、平台架构以及操作系统为基础。不过，目前我们离这样的一个目标还相去甚远。

本书强调实践，力求能够帮助读者编写出属于自己的嵌入式操作系统。如果读者以本书为基础（或者哪怕从中得到了一丝灵感）开发出一些优秀的嵌入式操作系统，那将会是非常令人高兴的事情！

本书涵盖内容：

● 搭建工作环境

选择合适的开发环境
开发工具的使用
.....

简单的中断处理实例
.....

● 基础知识

使用C语言写第一段程序
用脚本链接目标文件
.....

● 动态内存管理

伙伴算法
slab
.....

● 操作系统的启动

启动流程
MMU
.....

● 框架

驱动程序框架
文件系统框架
.....

● 运行用户程序

二进制程序的运行方法
可执行文件格式
.....

● 打印函数

打印函数实例
实现自己的打印函数

● 进程

进程的实现原理
进程的实现
.....

● 中断处理

ARM的中断

上架建议：计算机>嵌入式/ARM

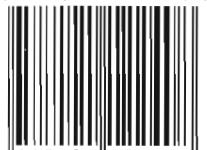


策划编辑：袁金敏
责任编辑：贾莉
封面设计：李玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。



ISBN 978-7-121-12240-8



9 787121 122408 >

定价：39.00元