

深度探索 嵌入式操作系统

从零开始设计、架构和开发

彭东 著

Inside Embedded Operation System
Design, Structure and Develop from Scratch

- 作者是计算机领域“鬼才”，独立编写出基于x86_64平台的系统内核和基于ARM平台的嵌入式系统内核，可运行于真实物理机
- 从硬件和软件两个维度系统、深度阐述嵌入式操作系统的构成、原理和实现方法，真正教读者从零开始设计、架构和开发一个完整的、可工作的嵌入式操作系统



机械工业出版社
China Machine Press

深度探索 嵌入式操作系统

从零开始设计、架构和开发

Inside Embedded Operation System
Design, Structure and Develop from Scratch

彭东 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深度探索嵌入式操作系统：从零开始设计、架构和开发 / 彭东著. —北京：机械工业出版社，2015.9

(Linux/Unix 技术丛书)

ISBN 978-7-111-51487-9

I. 深… II. 彭… III. 实时操作系统 IV. TP316.2

中国版本图书馆 CIP 数据核字 (2015) 第 216003 号

深度探索嵌入式操作系统 从零开始设计、架构和开发

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：高婧雅

责任校对：殷虹

印刷：北京市荣盛彩色印刷有限公司

版次：2015 年 10 月第 1 版第 1 次印刷

开本：186mm × 240mm 1/16

印张：33.75

书号：ISBN 978-7-111-51487-9

定价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

为什么写这本书

随着移动互联网技术的成熟，物联网也开始崭露头角，由此产生了各种小型、低功耗的智能硬件，这些智能硬件被嵌入到移动电话、手表、冰箱、空调、电视机、洗衣机等常用电子产品中，使这些常用电子产品功能更加强大、更加智能，而且它们可以连接到网络，便于用户远程操控，从而大大改善了人们的生活。

因此，嵌入式行业才变得如此火热，那些被嵌入到电子产品中的智能硬件，也需要一些小巧、特殊的操作系统软件才能正常工作，这类小巧、特殊的操作系统软件，称为嵌入式系统。2013年下半年，我开始学习嵌入式系统，并编写了一个嵌入式操作系统——LMOSEM。在互联网上也认识了不少研究嵌入式的朋友，在他们的要求和规劝下，我终于有勇气把我研究出来的东西归纳、整理成册，也算是我学习的笔记，于是就有了这本书。虽然有很多的顾虑，怕贻笑方家、怕误导同道……但是我的每行代码、每个点子，都在实机上测试过并证明了其正确性，所以也就心下一片坦然了。如果这本书能够被后来者借鉴一二，或者解决他们的一些疑惑，我自然欣慰万分。

关于 LMOSEM

关于 LMOSEM，这得从 LMOS 开始说起。2010年下半年，我开始准备要写个操作系统内核，没有其他目的，只是出于学习，出于兴趣。由于是自己独立从零开始设计、编写的，我觉得自己这种行为有点疯狂，索性用 LMOS (liberty madness operating system) 命名了我的操作系统。LMOS 经过这几年的独立开发，现在已经发布了 6 个测试版本。先后从 32 位单 CPU 架构发展到 64 位多 CPU 架构，现在的 LMOS 已经是多进程、多线程、多 CPU、支持虚拟内存的 x86_64 体系下的操作系统内核。LMOS 的这些特性，非常适合通用计算机领域，如 PC、工作站、小型服务器。这些特性导致 LMOS 代码量庞大，一些组件不够小巧，削剪起来非常复杂，很难保证削剪后的组件是否健壮，因此 LMOS 不适合于嵌入式领域，所以笔

者才重新开发了 LMOSEM——适合嵌入式领域的操作系统。

LMOSEM 依然删除了很多代码，因为写书要做到简单，便于理解。即便如此，LMOSEM 依然包含了现代操作系统的大部分重要组件，如内存管理、进程管理、驱动模型、文件系统等。这些组件的实现过程在本书中都会有详细的介绍。LMOSEM 不支持实时性功能，嵌入式操作系统也不一定要是实时性的操作系统，何况我们是出于学习的目的。为了代码的清晰、简单，我们暂不考虑安全性和性能方面的问题。等到明白了操作系统原理，我们再去不断修正、优化，使之功能变得更多，性能变得更强。笔者开发的 LMOSEM 操作系统项目，是在 Linux 操作系统下开发的，用到了 Linux 操作系统的很多工具。笔者不会和读者讨论为什么不用常用的 Windows 系统，也不会说谁好、谁不好。如果读者非常喜欢 Windows 系统，那么也可以尝试着把这个项目迁移到 Windows 系统下。但是笔者书中演示的环境还是 Linux 系统。关于如何搭建开发环境，本书后面的章节有详细的介绍。在那里读者会发现用 Linux 系统开发 LMOSEM 内核有很多方便之处，如会用到的 MAKE、GCC、LD 等，这些工具在 Linux 系统下都很容易得到，在 Windows 系统下虽然也能做到，但相对麻烦一点。何况今天的 Linux 系统已经很好用了。

读者对象

- ❑ 如果读者是一位纯粹的操作系统爱好者，对其有着浓厚的兴趣，那么本书将非常适合。
- ❑ 如果读者是嵌入式领域的从业者或者学生，也可以从本书中获得很多帮助。
- ❑ 如果读者是一位普通的应用软件开发者，业余时间也可以翻翻此书，书中的一些设计方法和编程手段，或许可以借鉴一二。
- ❑ 如果你只是想了解一些计算机硬件系统和软件系统的常识，那么本书同样会让你获益。

如何阅读本书

为了能更轻松地阅读这本书，笔者建议先了解 C 语言这门编程语言，对数据结构有所了解就更好了。除这些外，笔者假定读者没有其他任何技能。除了需要的上述技能，读者还需要对操作系统有强大的兴趣和求知欲，要有坚强的意志、永远不放弃的精神。开发操作系统内核本身就不是件容易的事，必然会有很多问题在等着我们，但是遇到问题不要害怕，静下心来从容面对，只要我们不放弃，问题最终会解决。

本书很简单，没有拐弯抹角，没有反复修饰，但是必要的细节从不漏掉。宁可在细节上啰嗦一点，也不在不相关的地方多写一句。

本书的最终目的是构建一个用于学习的嵌入式操作系统内核，并工作在真正的物理机上。为了达到这一目的，本书大体上分为三部分：综述、硬件部分和软件部分。

综述部分 (第 1 章)。

第 1 章, 先说明操作系统的概念、功能和演进历史, 最后得出现代操作系统的模型, 使我们了解操作系统的轮廓。

硬件部分 (第 2 ~ 3 章)。

第 2 章, 从选择硬件平台开始, 首先概述硬件平台的整体情况, 接着了解编写操作系统内核必需的一些平台上的组件, 如实时时钟、定时器、串口、中断控制器、内存芯片、Flash 芯片、CPU、MMU 等。让读者有初步的印象, 在写代码用到某个组件时再详述其内部编程细节。

第 3 章, 详细介绍处理器, 重点介绍处理器的结构和特性、处理器的地址空间、处理器的状态和工作模式、处理器的寄存器和指令集。最后介绍处理器中的 MMU 和 Cache, 对于 MMU, 主要介绍 MMU 的作用和它对操作系统内核开发的影响、如何对 MMU 编程、MMU 的几种地址映射方式。而对于 Cache, 重点介绍 Cache 的作用、Cache 的类型、Cache 的使用。

软件部分 (第 4 ~ 12 章)。

第 4 章, 介绍操作系统内核设计、操作系统内核的开发环境、开发操作系统内核的工具: GCC、LD、MAKE, 以及它们的使用方法, 最后介绍硬件平台的安装与测试。

第 5 章, 首先介绍 C 语言使用寄存器的约定, 以及它是如何处理参数、返回值的。接着介绍 C 语言基本数据类型的位宽及占用内存的大小, 并用它们构建后面将要用到的一些基本的数据结构, 如 `list_h_t`、`spinlock_t`、`kwlst_t`、`sem_t`。然后介绍 C 语言的数据结构在内存中存在的形式、对齐方式。最后介绍 GCC 独有的嵌入汇编代码的方式。

第 6 章, 开始介绍 LMOSEM 的初始化, 从第一行汇编代码开始、初始化 MMU 和中断向量、初始化串口设备、初始化内存管理数据结构和中断相应的数据结构, 最后对一些数据结构进行测试。

第 7 章, 开始介绍 LMOSEM 的内存管理组件。LMOSEM 的内存管理组件分为三层: 块级内存管理、页级内存管理、字级内存管理, 这三个层分别应对不同的内存分配请求。本章将结合实际代码介绍如何一步步实现这三大内存管理层。

第 8 章, 介绍 LMOSEM 的中断管理组件, 内容包括中断控制器的细节、中断管理组件的结构、如何处理中断、安装中断处理的回调函数等。

第 9 章, 介绍 LMOSEM 的设备驱动模型, 其中介绍 LMOSEM 如何管理众多的硬件设备、LMOSEM 支持的设备类型、驱动模型的数据结构和基础性代码, 最后用两个驱动程序实例介绍如何在 LMOSEM 的驱动模型下编写规范的设备驱动程序。

第 10 章, 介绍 LMOSEM 的进程管理组件, 包括进程的由来、进程相关的数据结构、系统空转进程的建立与运行、进程调度、新建进程、进程的睡眠与唤醒、进程测试等相关内容。

第 11 章, 介绍 LMOSEM 的文件系统组件, 包括文件系统的设计、文件系统的建立、文件系统的基础操作、文件本身的操作, 如文件的打开、新建、读写、删除等。最后对文件系

统组件进行严格的测试。

第 12 章，介绍 LMOSEM 的接口，包括许多 LMOSEM 的 API 和库函数的实现细节，主要包括时间、进程、内存、文件与设备、标准输入 / 输出等方面的 API 和库函数。

勘误和支持

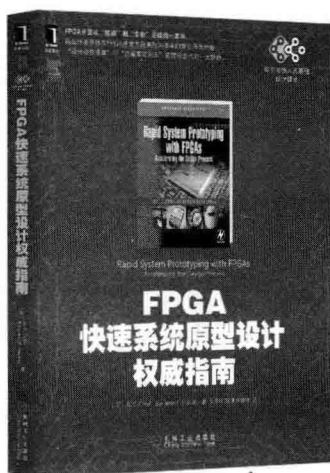
由于笔者水平有限，加之编写时间仓促，书中难免会出现一些不准确的地方，恳请读者批评指正，在技术之路上共勉。我的 CU 博客地址是：<http://blog.chinaunix.net/uid/28032128.html>。本书源代码已制作成光盘镜像文件，并上传到华章网站（www.hzbook.com），需要的读者可自行下载。

致谢

我，自幼患病，读书不多，计算机成了唯一的兴趣爱好，没有父母的长期支持，连生活都尚且不能自理，更别说完成此书了，他们对我的帮助和关爱，纵使千万言语也难表一二。由于经常在物理机上测试内核，要拆装一些设备和器件，这多亏了我的小弟，因为他一有时间就帮我做这部分工作。当然还有帮助过我的朋友，有一些是身边的，有些是网络中的。对父母、所有的亲人、朋友，我也只有常怀感恩之心，说声谢谢，谢谢他们一直的支持、帮助，谢谢他们一直对我那满满的关爱！

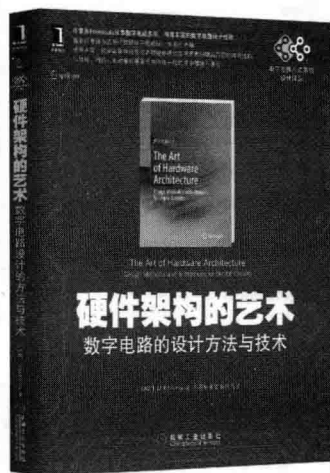
让笔者和你一起带着未知，带着好奇，带着兴奋，踏上操作系统的旅程吧！

推荐阅读



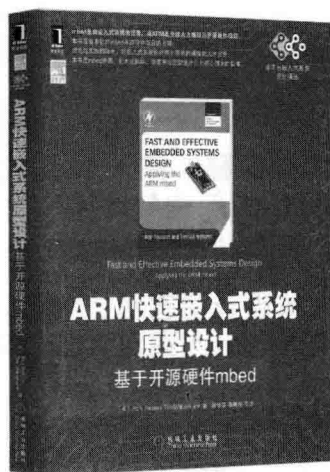
FPGA快速系统原型设计权威指南

作者: R.C. Cofer 等 ISBN: 978-7-111-44851-8 定价: 69.00元



硬件架构的艺术: 数字电路的设计方法与技术

作者: Mohit Arora ISBN: 978-7-111-44939-3 定价: 59.00元



ARM快速嵌入式系统原型设计: 基于开源硬件mbed

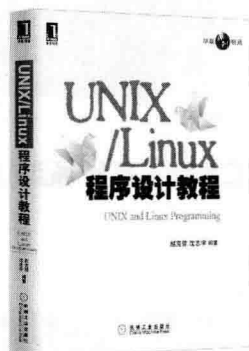
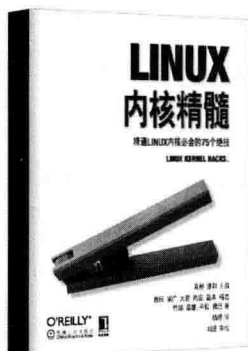
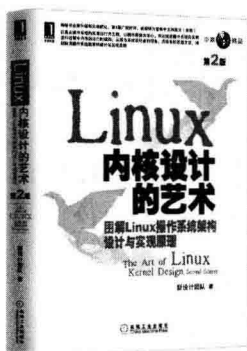
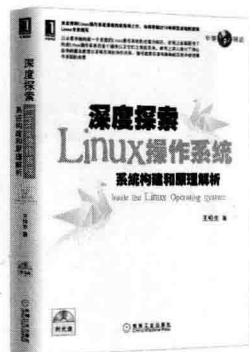
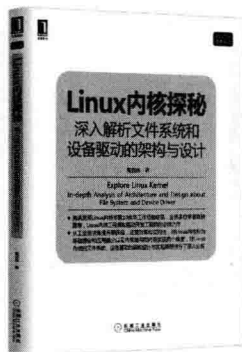
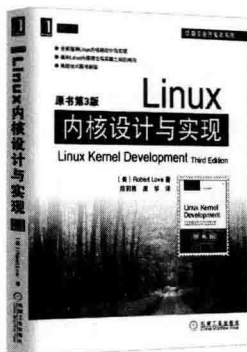
作者: Rob Toulson 等 ISBN: 978-7-111-46019-0 定价: 69.00元



嵌入式软件开发精解

作者: Colin Walls ISBN: 978-7-111-44952-2 定价: 79.00元

推荐阅读



Linux内核设计与实现（原书第3版）

Linux内核开发人员Robert Love的力作，畅销多年的经典著作

深度探索Linux操作系统：系统构建和原理解析

百度核心系统部门资深专家力作，Linux操作系统领域的里程碑作品

Linux内核精髓：精通Linux内核必会的75个绝技

日本多位一线内核技术专家的经验 and 智慧结晶

Linux内核探秘：深入解析文件系统和设备驱动的架构与设计

腾讯顶级Linux系统专家和存储系统专家10年经验结晶

Linux内核设计的艺术：图解Linux操作系统架构设计与实现原理（第2版）

中国首部将版权输出到美国的计算机图书，中美两国取得骄人成绩

UNIX/Linux程序设计教程

UNIX/Linux权威著作，多所高校选定为教材

Contents 目 录

前言

第 1 章 操作系统的功能及为什么需要它

- 1.1 从 hello world 开始
- 1.2 操作系统功能演进
- 1.3 小结

第 2 章 硬件平台

- 2.1 选择平台
 - 2.1.1 mini2440
 - 2.1.2 mini2440 平台的信息
- 2.2 必须要关注的硬件
 - 2.2.1 原因
 - 2.2.2 RTC
 - 2.2.3 定时器
 - 2.2.4 串口
 - 2.2.5 中断控制器
 - 2.2.6 SDRAM
 - 2.2.7 Norflash
 - 2.2.8 Nandflash
- 2.3 小结

第 3 章 处理器

- 3.1 ARM 公司与其处理器

- 3.2 ARM920T 的结构与特性
 - 3.2.1 ARM920T CPU 结构
 - 3.2.2 ARM920T CPU 特性
- 3.3 ARM920T 存储体系
 - 3.3.1 ARM920T 地址空间
 - 3.3.2 ARM920T 存储器格式
 - 3.3.3 ARM920T 存储地址对齐
- 3.4 ARM920T 状态
 - 3.4.1 ARM 状态
 - 3.4.2 Thumb 状态
- 3.5 ARM920T 处理器的 7 种工作模式
- 3.6 寄存器
- 3.7 异常和中断
 - 3.7.1 什么是异常和中断
 - 3.7.2 异常中断向量
- 3.8 ARM920T 指令集
 - 3.8.1 指令及其编码格式
 - 3.8.2 分支跳转指令
 - 3.8.3 数据处理指令
 - 3.8.4 装载和存储指令
 - 3.8.5 程序状态寄存器操作指令
 - 3.8.6 协处理器操作指令
 - 3.8.7 异常中断产生指令
- 3.9 MMU
 - 3.9.1 MMU 概述

3.9.2	为什么要有 MMU	82	5.1.1	寄存器别名	157
3.9.3	ARM920T CP15 协处理器	85	5.1.2	参数传递与返回值	157
3.9.4	MMU 页表	98	5.2	基本数据结构	159
3.9.5	MMU 页面访问权限的控制	113	5.2.1	C 语言的基本数据结构	160
3.9.6	MMU 的快表 TLB	113	5.2.2	list_h_t 数据结构	161
3.9.7	MMU 的编程接口	114	5.2.3	spinlock_t 数据结构	164
3.10	Cache	115	5.2.4	kwlst_t 数据结构	165
3.10.1	ARM920T 的 Cache	115	5.2.5	sem_t 数据结构	166
3.10.2	Cache 的原理	116	5.3	数据结构存在于内存中的形式	168
3.10.3	Cache 的类型及要注意的 问题	117	5.4	C 与汇编的混用	170
3.10.4	ARM920T Cache 的编程接口	119	5.5	小结	174
3.11	小结	120	第 6 章	内核初始化	175
第 4 章	操作系统内核的设计与构建	122	6.1	开始	175
4.1	操作系统内核的设计	122	6.1.1	第一行汇编代码	175
4.1.1	内核要完成的功能	123	6.1.2	第一个 C 函数	178
4.1.2	内核的架构	124	6.2	MMU 和中断向量的初始化	181
4.1.3	分离硬件的相关性	126	6.2.1	初始化 MMU	181
4.1.4	我们的选择	127	6.2.2	复制中断向量	186
4.2	开发环境及相关工具	129	6.3	串口初始化	190
4.2.1	Linux 环境	129	6.3.1	串口硬件	190
4.2.2	文本编辑器	132	6.3.2	内核的 printf	196
4.2.3	GCC	134	6.4	机器数据结构	201
4.2.4	LD	136	6.4.1	设计数据结构	201
4.2.5	make	139	6.4.2	确定一些重要数据结构与内核 的地址	203
4.3	LMOSEM 的构建系统	142	6.5	初级内存管理初始化	205
4.3.1	LMOSEM 的 makefile	142	6.5.1	设计一些数据结构	205
4.3.2	LMOSEM 的链接脚本	147	6.5.2	初始化 mmapdsc_t 结构数组	209
4.4	开发板的安装	150	6.5.3	建立起内存分配数据结构	212
4.5	小结	154	6.6	中断初始化	215
第 5 章	语言间调用约定与基本数据 结构	156	6.6.1	设计一些数据结构	215
5.1	寄存器使用约定	156	6.6.2	初始中断源描述符	220
			6.7	初始化测试	222
			6.8	小结	225

第 7 章 内存管理	226	7.5.4 分配时查找 <code>mplhead_t</code>	278
7.1 内核功能层入口	226	7.5.5 分配时新建字级内存池	279
7.2 内存管理组件的设计.....	228	7.5.6 分配时操作 <code>mplhead_t</code>	282
7.3 块级内存管理.....	229	7.5.7 分配代码写得对吗	283
7.3.1 块级内存管理数据结构视图	229	7.5.8 释放主函数	285
7.3.2 块级内存管理接口	230	7.5.9 释放时查找 <code>mplhead_t</code>	285
7.3.3 主分配函数	232	7.5.10 释放时操作 <code>mplhead_t</code>	287
7.3.4 分配时查找 <code>alcfrlst_t</code>	234	7.5.11 释放时删除字级内存池.....	288
7.3.5 分配时查找和操作 <code>mmapdsc_t</code>	236	7.5.12 测试字级内存管理层	289
7.3.6 分配代码写得对吗	239	7.6 小结.....	292
7.3.7 主释放函数	240	第 8 章 中断管理	293
7.3.8 释放时查找 <code>alcfrlst_t</code>	241	8.1 中断与中断控制器.....	293
7.3.9 释放时查找和操作 <code>mmapdsc_t</code>	242	8.1.1 什么是中断	293
7.3.10 测试块级内存管理层	246	8.1.2 S3C2440A 中断控制器.....	294
7.4 页级内存管理.....	248	8.2 中断管理的架构与相关数据 结构	298
7.4.1 页级内存管理接口及调用 流程.....	248	8.2.1 中断管理的架构	298
7.4.2 相关的数据结构	251	8.2.2 设计数据结构 <code>intfltdsc_t</code> 和 <code>intserdsc_t</code>	298
7.4.3 页级内存管理初始化	254	8.3 中断处理.....	301
7.4.4 分配主函数	256	8.3.1 中断辅助例程	301
7.4.5 分配时查找 <code>mplhead_t</code>	257	8.3.2 从中断向量开始	305
7.4.6 分配时新建页级内存池	258	8.3.3 保存 CPU 上下文.....	306
7.4.7 分配时操作 <code>mplhead_t</code>	262	8.3.4 中断主分派例程	310
7.4.8 分配代码写得对吗	263	8.3.5 确定中断源	314
7.4.9 释放主函数	265	8.3.6 调用中断处理例程	317
7.4.10 释放时查找 <code>mplhead_t</code>	266	8.4 安装中断回调例程.....	319
7.4.11 释放时操作 <code>mplhead_t</code>	268	8.5 小结.....	322
7.4.12 释放时删除页级内存池	269	第 9 章 驱动模型	323
7.4.13 测试页级内存管理层	271	9.1 操作系统内核如何管理设备.....	323
7.5 字级内存管理.....	273	9.1.1 分权而治	323
7.5.1 字级内存接口及调用流程	274	9.1.2 设备类型	325
7.5.2 相关的数据结构	275	9.1.3 驱动程序	327
7.5.3 分配主函数	276		

9.2 相关数据结构	328	10.3.1 进程管理组件的初始化	395
9.2.1 驱动	329	10.3.2 建立空转进程	396
9.2.2 派发例程类型	329	10.3.3 空转进程运行	399
9.2.3 设备 ID	330	10.4 新建进程	404
9.2.4 设备	331	10.4.1 分配进程描述符	404
9.2.5 IO 包	332	10.4.2 分配内存空间	406
9.2.6 设备表	333	10.4.3 加入进程调度表	408
9.3 驱动模型的基础设施	335	10.5 进程调度	410
9.3.1 驱动程序从哪里执行	335	10.5.1 调度算法	410
9.3.2 新建与注册设备	340	10.5.2 处理进程时间片	411
9.3.3 注册回调函数	344	10.5.3 检查调度状态	414
9.3.4 发送 IO 包	345	10.5.4 选择进程	415
9.3.5 调用驱动程序函数	346	10.5.5 进程切换	418
9.3.6 等待服务	347	10.5.6 进程等待与唤醒	421
9.3.7 完成服务	350	10.5.7 进程测试	425
9.3.8 驱动模型辅助函数	352	10.6 小结	428
9.4 systick 驱动程序实例	356	第 11 章 文件系统	430
9.4.1 systick 硬件	356	11.1 文件系统设计	430
9.4.2 systick 驱动程序框架	360	11.1.1 文件系统只是一个设备	430
9.4.3 systick 驱动程序实现	362	11.1.2 数据格式与存储块	432
9.4.4 测试 systick 驱动程序	368	11.1.3 如何组织文件	433
9.5 RTC 驱动程序实例	370	11.1.4 关于我们文件系统的限制	434
9.5.1 RTC 硬件	370	11.2 相关的数据结构	434
9.5.2 RTC 驱动程序实现	375	11.2.1 超级块	435
9.6 小结	385	11.2.2 位图	435
第 10 章 进程	386	11.2.3 目录	437
10.1 应用程序的运行	386	11.2.4 文件管理头	438
10.1.1 程序运行需要什么资源	387	11.3 文件系统格式化	440
10.1.2 任何时刻资源都可用吗	388	11.3.1 建立超级块	440
10.1.3 提出多道程序模型	389	11.3.2 建立位图	445
10.2 相关的数据结构	390	11.3.3 建立根目录	448
10.2.1 设计进程的数据结构	391	11.4 文件系统基础操作	452
10.2.2 调度进程表	392	11.4.1 获取与释放根目录文件	453
10.3 LMOSEM 内核的第一个进程	394	11.4.2 字符串操作	455

11.4.3	分解路径名	457	12.1.2	传递系统调用参数	486
11.4.4	检查文件是否存在	459	12.1.3	系统调用分发器	488
11.5	文件操作	460	12.2	时间管理系统调用	489
11.5.1	新建文件	461	12.3	进程管理系统调用	492
11.5.2	删除文件	463	12.3.1	进程的运行与退出	492
11.5.3	打开文件	466	12.3.2	获取进程的 ID	494
11.5.4	读写文件	469	12.4	内存管理系统调用	496
11.5.5	关闭文件	472	12.5	设备与文件系统调用	498
11.5.6	驱动整合	473	12.5.1	设备与文件的打开	498
11.6	文件系统测试	475	12.5.2	设备与文件的关闭	506
11.6.1	格式化测试	475	12.5.3	设备与文件的读写	508
11.6.2	文件操作测试	479	12.5.4	设备与文件的控制	512
11.7	小结	482	12.6	应用程序库	514
第 12 章	系统调用与应用程序库	483	12.7	测试	520
12.1	系统调用机制	483	12.8	小结	526
12.1.1	软中断指令	484	后记		528

操作系统的功能及为什么需要它

你或许已经卷起了衣袖，或许在摩拳擦掌准备大干一场，打一场硬仗。年轻人嘛，行事总是风风火火的。但不是笔者扫你的兴，泼你冷水，在我们写代码之前还有很长一段路要走，要静下心来。如果写操作系统是一次旅行，那么千万不要错过沿途的风景……

1.1 从 hello world 开始

操作系统也是软件，也是由一大堆程序组成的，所以不要觉得它多么神秘。既然是程序，我们就要知道它是干什么的以及为什么需要，笔者想用大多数人写的第一个程序——“hello, world!!”来描述这个原因。当然这个程序是用 C 语言写的。大致过程如图 1-1 所示。

这个程序正常运行后，会在屏幕上输出 hello, world!! 这个字符串。显然这个程序中最重要的是 printf 函数，然而我们并没有实现它。如果你能发现这一点，并迫切地想知道它究竟在背后干了些什么，那说明你和笔者一样有着对问题追根溯源的性格。这个也是我们研究问题本相的原动力。

那么 printf 在哪儿呢？都干了些什么呢？

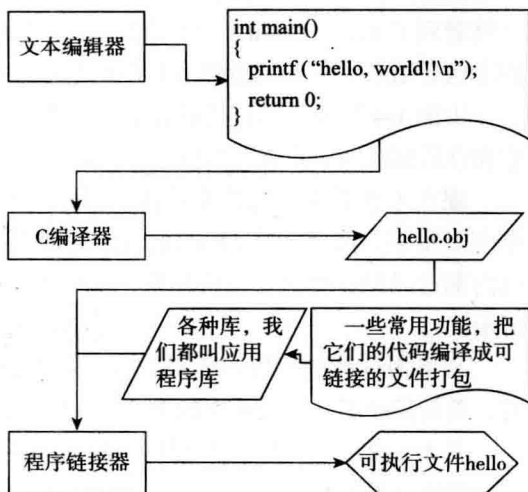


图 1-1 hello, world!! 程序的生成过程

它就在应用程序库中，代码虽然没有实现它，但库中实现了，程序链接器最终把它和代码链接装配在一起，这样程序也就能正确工作了，如图 1-2 所示。

实际中可能有差异，可能 printf 不是在单独的库中，也可能不只是链接这一个库就行了。但是道理都一样。

printf 函数把程序传给它的“hello, world!!”字符串数据复制到一个内存缓冲区中，然后调用操作系统提供的 API（应用程序编程接口）函数。可能不同的实现之间有差异，但是大致动作如图 1-3 所示。

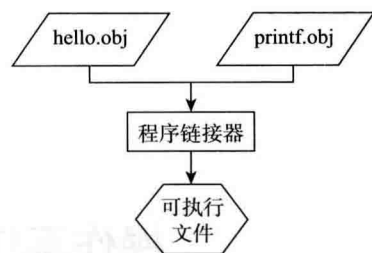


图 1-2 hello, world!! 程序的链接过程

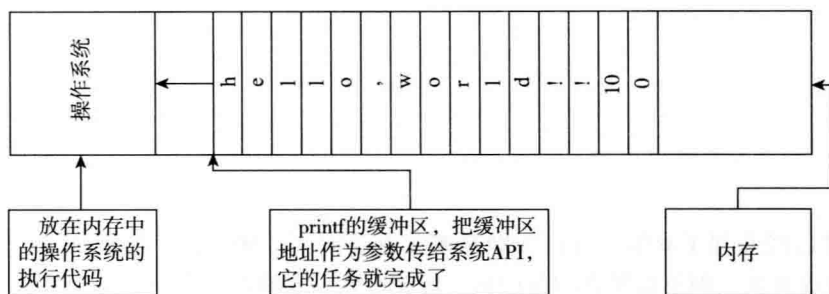


图 1-3 printf 函数的逻辑工作机理

一旦调用操作系统 API，代码的控制权就交给了操作系统，执行的也是操作系统的代码。至于实现这一机制的进程，后面的章节会详细讨论。操作系统会根据不同的 API 函数以及应用程序传递进来的参数完成不同的动作和功能。比如，程序中的 printf、调用的 API 和传递的参数，是要告诉操作系统把它缓冲区中的数据写入标准输出设备上。于是我们在屏幕上就看到了 hello, world!!，然后程序就逐层返回到 main 函数，最后 main 函数退出，这个小程序也就结束了。下面用图 1-4 来描述这一过程。

由图 1-4 可知，真正操控计算机硬件完成显示 hello, world!! 的是操作系统这个软件。是它在背后默默地工作着。当然它的功能并不只是操控计算机硬件，我们后面会慢慢讨论。

现在来看看没有操作系统能完成这个 hello, world!! 的程序吗？有！只是我们要做的工作多了很多，我们得亲自实现 printf 函数、完成控制计算机硬件的程序，并且还要求计算机内部这时只运行这一个应用程序，因为有些硬件同一时间只能被一个程序使用，如图 1-5 所示。

写上面这样的程序可能对程序员的要求有点高，因为必须要知道计算机硬件的每个细节，否则程序不会出现正常的结果。好吧，我们还是应该承认这个世界存在这种编程高手。

关于 hello, world!! 这个程序，我们就说到这里。如果没有操作系统，无论开发程序还是运行程序，难度都会很高。

现在去看看先辈是如何一步一步创造出操作系统的。

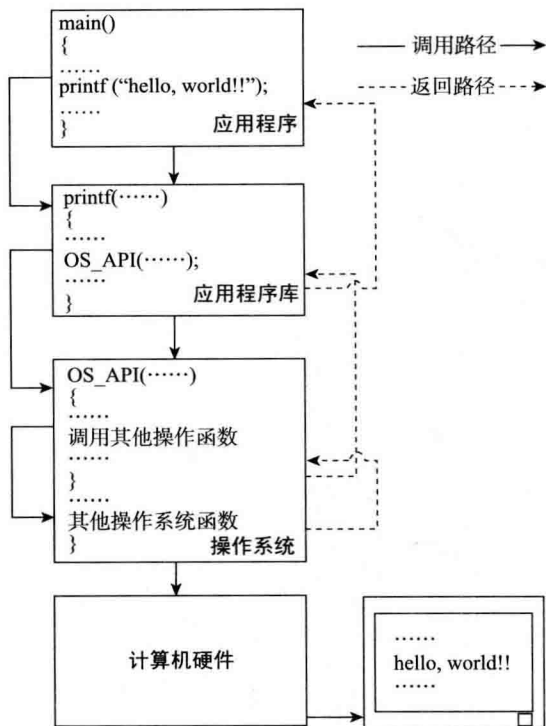


图 1-4 hello, world!! 程序调用操作系统 API 的过程

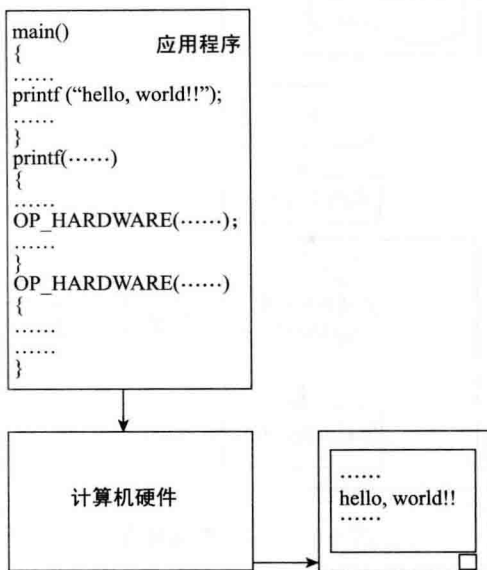


图 1-5 裸机 hello, world!! 程序

1.2 操作系统功能演进

在很久以前确实存在上述这种模式的程序，并且每次计算机中只存在一道这样的程序，运行完一道，手工装入第二道……你会发现这太慢了，也太麻烦了，对昂贵的计算机硬件的利用率也太低了，因为这样的程序是不可能让计算机内部所有的设备都工作起来的。

然而首先知道这个问题的不是我们，先辈早已看到这类问题，他们想了个方法，在计算机的内部放入一道监控程序，然后把要执行的所有程序放在一个固定的介质上，每次那个监控程序选择一道要运行的程序装入计算机内部去执行，这道程序执行完了，再装入下一道程序……直到运行完所有的程序。如图 1-6 所示。从全局看效率高了不少，但是从局部来看，它仍然没有让计算机的所有设备都工作起来。同时也还是没有降低程序员的难度，因为还是要操控程序要用到的硬件。

我们马上会想到，能不能把那个所谓的监控程序稍稍改进和扩展一下，把那些操作各种硬件的代码和实现重要功能的代码，放进监控程序里。然后我们对这些功能进行编号，比如，1 号功能是操作键盘的、2 号功能是打开硬盘上的文件……最后应用程序要完成什么功能，只要 call $\times\times$ 号功能，计算机的控制权就会交给监控程序，当监控程序中的代码完成这个功能号对应的服务时就返回到应用程序中。图 1-7 展示了这一过程。

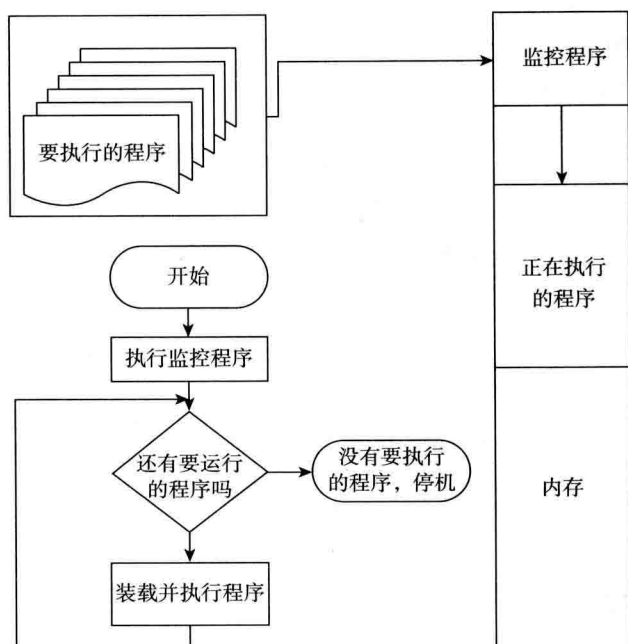


图 1-6 古老的监控程序及其执行过程

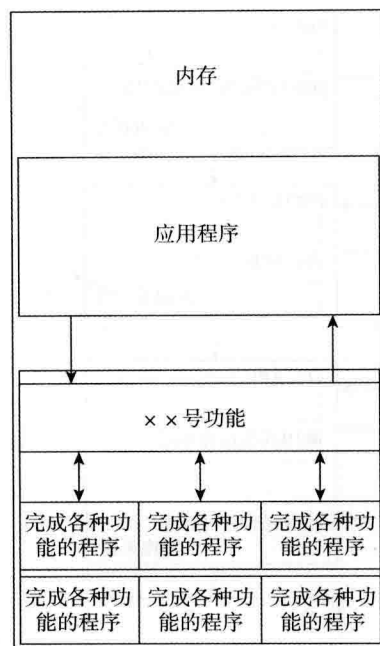


图 1-7 进一步扩展的监控程序

这样给应用程序开发人员带来的不只是一点点方便，对他们而言，完全是种解脱。方便是方便了不少，但这还是只运行了一个应用程序，你或许想到了 DOS 操作系统。是的，DOS 就是这样的操作系统。

我们继续思考一种情景：你正在用 QQ 软件和你的朋友聊天，QQ 软件等待你输入消息，然后把这个消息通过网络发送给你的朋友。QQ 软件等待你的输入，这个时间可能是几秒钟，甚至更久。这几秒钟对如此之快的计算机来说，无疑是巨大的浪费。如果这时内存中存在着另一个软件，这个几秒钟是否可以拿来运行另一个软件呢？那当然是可以的。比如，我们让这几秒钟切换到 KuGou 软件上，等到用户输入消息时，再换回来，于是我们就可以一边聊天，一边听音乐了，如图 1-8 所示。当然这是其中一种比较常见的情景，还有等待网卡上的网络数据包、等待磁盘上的数据、等待声卡缓冲区空闲、等待打印机打印完文档、等待扫描仪的输入等之所以会有以上这些“等待”，是因为计算机内部各种设备的速度不相同，如键盘的输入速度赶不上 CPU 的速度、磁盘的数据传输速度比不上 CPU 的数据处理速度……

还是上面的情景，我们继续进一步思考一些问题。QQ 和 KuGou 同时存在于内存中，那么就要占据不同的物理内存空间。作为操作系统当然就要知道哪些内存是空闲的，哪些内存是已经占用的，当然如果 QQ 程序退出了操作系统就要标识 QQ 占用的内存已经空闲了，从而可以被用于其他用途。QQ 和 KuGou 是两个不同的程序，操作系统要知道它们的存在，包括它们的一些信息，例如，它们各自占用多大内存、在内存中的什么地址，它们各自执行到哪里了，以及当前 CPU 上执行的是哪个程序。QQ 要记录你的聊天数据，KuGou 要读取

MP3 音乐数据，操作系统就要组织这些数据的存放、读取，以及控制这些数据的访问权限。最后 QQ 要发送消息，KuGou 要播放音乐，这最终都要用到计算机内部具体的硬件设备，QQ 要用到网卡和磁盘、KuGou 要用到磁盘和声卡，操作系统当然要知道计算机内部有多少设备，每个设备的状态、类型及访问操作方式，以便代替应用程序访问控制设备。

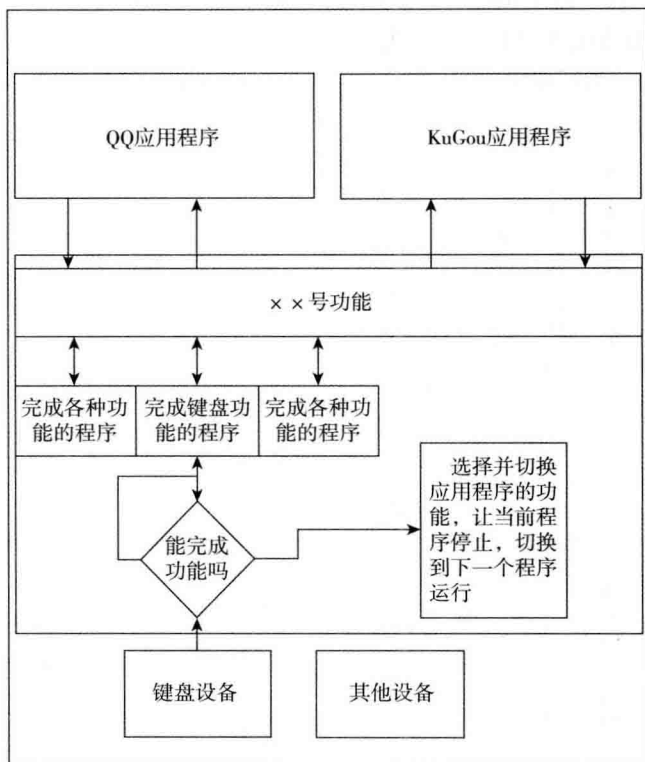


图 1-8 多道程序的设想

上述情景中，我们思考的问题，当然也是操作系统重点要解决的问题，也可以说是操作系统要完成的功能。先辈还对其进行规划并定义了如下专业的术语。

内存管理：最简单的原因是要在内存中放入不止一道程序。

进程管理：既然内存中有多道程序，首先要知道它们的存在、状态及与这道程序相关的重要信息。

文件系统：用户有那么多数据，如音乐、电影、文档、其他数据，如何组织它们，以何种形式查找访问它们，如何才能把它们安全地保存在计算机中。

设备管理：由上可知，操作系统必须要知道计算机内部有多少个设备、设备各自是什么类型，设备当前能否被访问，最重要的任务是对应用程序屏蔽设备细节，代替应用软件访问控制设备。

如果站在全局较高的角度看，就是这样的：把计算机内部的设备和设备中的数据统称为“资源”，操作系统即是众多资源的管理者。应用程序无非就是通过操作系统这层软件获取资源、使用资源、释放资源。而操作系统则一边维护众多资源的状态，一边通过众多资源状态，如该资源当前能否使用，或者该资源是共享访问的资源还是互斥访问的资源，从而调度安排众多应用程序有序、高效地运行。最后达到高效使用计算机的目的。

上面所有这些描述或许看上去很陌生，先不要急，我们后面慢慢地一步一步介绍。我们根据上述所有的描述，用一张图来大致表示现代整个计算机系统的结构，如图 1-9 所示。



图 1-9 现代操作系统模型

从图 1-9 不难看出，如此复杂的计算机，有如此多的功能，时刻影响并改变着人们的生活。然而它也是一层一层地建立起来的。或许我们还有很多疑问：如何实现设备管理、进程又是什么、如何实现系统调用……这正是我们后面要解决的问题，现在我们只要明白计算机中有一层叫操作系统的软件，它在背后默默地干了些十分重要的事情，如果没有它事情会变得很糟糕。

1.3 小结

这一章就要结束了，现在我们来回想一下：

- 1) 我们从 `hello, world!!` 开始，发现完成输出功能的是一个叫操作系统的软件；
- 2) 后来我们想不用操作系统，也能完成这个小程序，只是困难了很多，重要的是计算机的利用率很低；
- 3) 然后我们做了一个监控程序，让它负责运行这种类型的程序，并对它进行改进，降低了编程者的编程难度，计算机的利用率依然很低；
- 4) 再然后我们在内存中同时放入多个程序，让一个程序因为一些原因不能继续向下运行时，就切换到别的程序，让其开始运行；
- 5) 最后，我们基于第4点的想法，不断地改进和扩展，终于建造出现代操作系统的模型（见图1-9）。

现在休息一下，进行下一步探索，相信会更精彩。

硬件平台

这可能有点天马行空了。第 1 章粗略地介绍了操作系统的功能以及为什么需要它，接着就讲到了硬件平台。或许你还在想操作系统的第一行代码是什么、该从哪儿开始呢？不要急，现在所做的一切，都是为了最终的目标而努力着。做任何事情必然都有个过程，不能一蹴而就，心态平静很重要。本章将首先介绍为什么要选择 mini2440 这个硬件平台，最后介绍我们必须关注的这个硬件平台上的一些组件。

开始吧！

2.1 选择平台

读者可能没有听说过 mini2440，它是一种 ARM 开发板，这里先搞清楚如何选择一款硬件平台，为什么要选择 mini2440，然后介绍 mini2440 的相关信息。

2.1.1 mini2440

现代计算机系统大都可以从逻辑上分为三大层：硬件层、系统软件层、应用软件层，当然还可以继续细分，如图 2-1 所示。每层都隐藏自己相关的实现细节，向上层提供相关接口，根据情况决定是否调用下层提供的接口。就像第 1 章分析的 `printf` 函数一样，你不必思考它是怎么实现的，只要调用它就行了，具体怎么做是它的事。这就是为什么写一个应用程序，首先要去了解各种应用程序库和这个应用程序运行所在的操作系统平台的 API。因为我们要知道这个系统平台提供给应用程序的编程接口。

没有硬件的软件是空中楼阁，没有软件的硬件是一堆废物品，二者缺一不可。就像人一

样，不仅要有强健的身躯，还要有智慧的大脑。

我们的目的是写操作系统内核。那么好了，由图 2-1 可知，首先应该了解某一具体硬件平台提供给我们的“接口”。

现在市面上的计算机平台种类繁多，有各种型号的超级计算机、服务器、PC、嵌入式开发板，当然你可能不会选择超级计算机和服务器来开发操作系统内核，我们要从中选择一个适合我们的。

既然这样，就要选择一个简单、常用、便宜的计算平台作为开发平台。而满足前三个条件的就是 mini2440 开发板。嵌入式开发板，比通常用的 PC 简单、便宜，而大多数嵌入式开发者或者学习嵌入式的同学都是从 mini2440 开发板开始的。下面就开始仔细研究这个开发板。

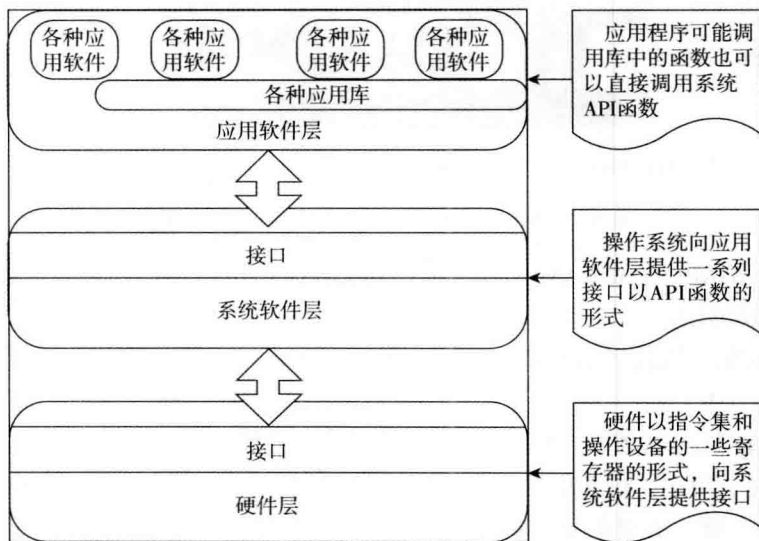


图 2-1 现代计算机系统的逻辑分层

2.1.2 mini2440 平台的信息

mini2440 开发板，是友善之臂公司基于三星公司的 S3C2440A 芯片，并结合了一系列的外围组件开发出来的一款嵌入式开发板。组件非常丰富，性能不错，也很常用，非常适合嵌入式系统的入门学习。

下面来看看 mini2440 开发板都有些什么，各自都有什么作用，即用来干什么的，mini2440 开发板的概况如图 2-2 所示。

哇，这看上去各种硬件接口、芯片……真不少。归纳如下。

1) Samsung S3C2440A CPU 处理器，主频 400MHz，最高 533MHz。

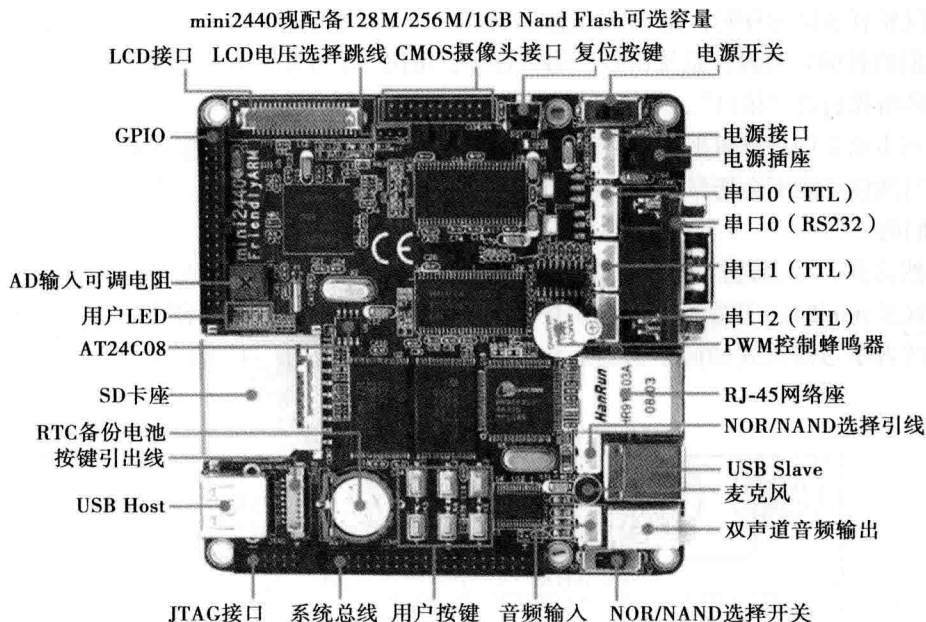


图 2-2 mini2440 物理视图 (源自友善之臂公司)

2) 由2片32MB的SDRAM芯片组成的64MB内存, 32位的内存数据总线, SDRAM时钟频率高达100MHz。

3) Flash存储, 板载, 256MB/1GB Nand Flash, 掉电非易失 (用户可定制 64MB/128MB/256MB/512MB/1GB)。

4) 2MB Nor Flash, 掉电非易失, 已经安装 superVIVI 引导程序, 相当于 PC 上的 BIOS。

5) LCD显示接口, 板上集成4线电阻式触摸屏接口, 可以直接连接四线电阻触摸屏, 支持黑白、4级灰度、16级灰度、256色、4096色的STN液晶屏, 尺寸从3.5寸~12.1寸, 屏幕分辨率可以达到1024×768像素。标准配置为3.5真彩LCD, 分辨率240×320, 带触摸屏。

6) 1个100M以太网RJ-45接口 (采用DM9000网络芯片)。

7) 3个串行口。

8) 1个USB Host。

9) 1个USB Slave B型接口。

10) 1个SD卡存储接口。

11) 1路立体声音频输出接口, 一路麦克风接口。

12) 1个2.0mm间距10针JTAG接口。

13) 4个USERLED灯。

14) 6个USER按键 (带引出座)。

15) 1个PWM控制蜂鸣器。

- 16) 1个可调电阻, 用于AD模数转换测试。
- 17) 1个I2C总线AT24C08芯片, 用于I2C总线测试。
- 18) 1个2.0mm间距, 20pin摄像头接口。
- 19) 1个板载实时时钟电池。
- 20) 电源接口(5V), 带电源开关和指示灯。
- 21) 系统时钟源, 12MHz无源晶振。
- 22) 内部实时时钟(带后备锂电池)。
- 23) 1个34pin, 2.0mm, GPIO接口。
- 24) 1个40pin, 2.0mm, 系统总线接口。

24条, 可真不少啊, 还好我们开发操作系统内核时, 并不需要关心这个开发板上全部的硬件资源。

现在把那些硬件提供的插口、小灯、按键、电池等都扔了, 搅在一起可能看不清楚。看图2-3或许会清楚许多, 心情也会愉悦不少。

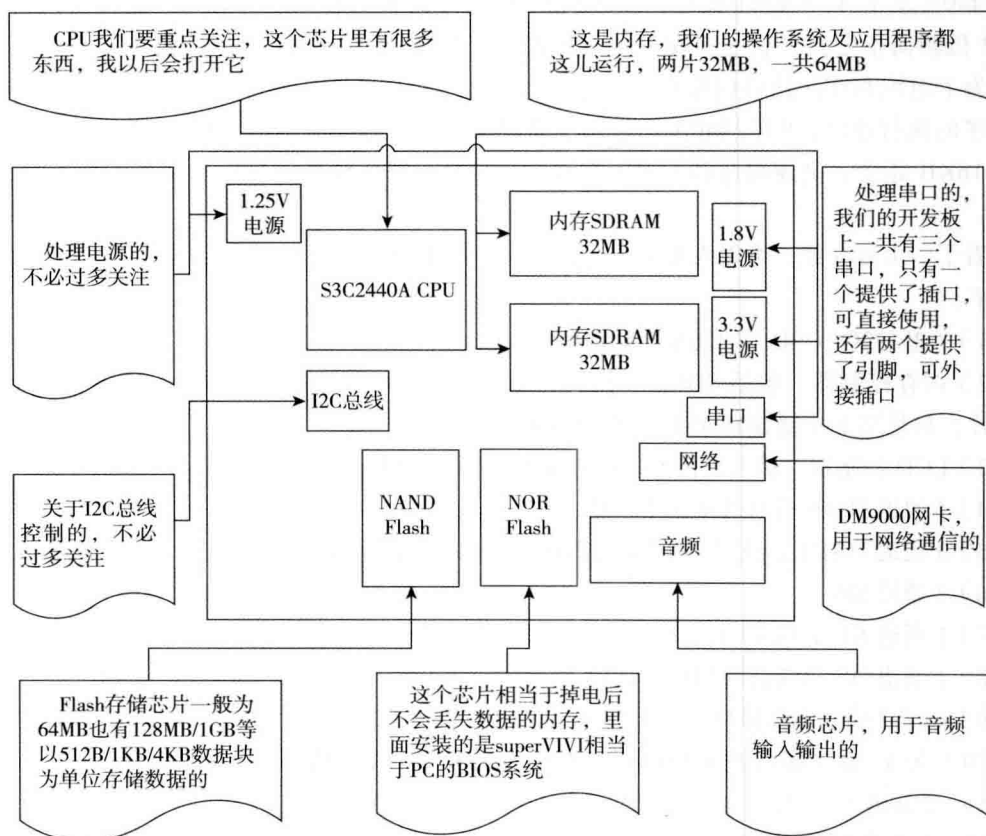


图 2-3 mini2440 逻辑视图

根据图 2-3 可以把它们分为这几类:

- 1) 处理器, 这是用来执行程序。
- 2) 存储器, 两个不同类型的 Flash, 当然它们也是存储器, 2 片 32MB SDRAM 的内存。
- 3) 关于开发板电源的, 不必过多关注。
- 4) 网络, 就是网卡。
- 5) 音频。
- 6) 串口, 注意图 2-3 上的串口不是用于处理串口发送、控制等逻辑的串口芯片。
- 7) I2C 总线, 这个暂时不必关注。

上面这 7 种类型的部件, 除第 1 类处理器外, 其他部件功能和用途都相对单一, 如 Flash 和 SDRAM 都是用于存储数据的、音频芯片就是用来处理声音的, 它们不做其他的事。但是 S3C2440A 处理器就不同了, 别看它只有指头大小, 完成的功能却不容小觑, 下面来掀开它神秘的面纱。

S3C2440A 是三星公司生产的微处理器, 这可不是通常我们认为的微处理器。S3C2440A 基于 ARM920T 核心, ARM920T 核心才是我们通常说的 CPU、处理器, 它支持 16/32 位精简指令集。它还实现了内存管理单元 MMU。S3C2440A 还提供了丰富的内部设备, 为了组织和连接这些内部设备, 它内部采用了先进的微控制总线构架 AMBA。为了提高程序的执行速度, S3C2440A 还采用了哈佛架构(高速缓冲体系结构), 这一结构具有独立的 16KB 指令, 高速缓存和 16KB 数据高速缓存, 每个都是由具有 8 字长的行(line)组成的。

看了一大堆介绍, 可能人都晕了, 有时又不能不介绍。下面把 S3C2440A 内部的构件列举如下。

- 1) ARM920T CPU, 即通常执行程序的 CPU。
- 2) 内存控制器, 包括 SDRAM 控制和地址空间的片选逻辑。但本身不包含 SDRAM 存储芯片, 只是用于控制 SDRAM 存储芯片的。
- 3) LCD 控制器, 最大支持 4K 色 STN 和 256K 色 TFT, 提供 1 通道 DMA 供 LCD 专用。
- 4) 4 通道 DMA 并有外部请求引脚。
- 5) 3 通道 UART, 支持 64B 发送 FIFO 和 64B 接收 FIFO。
- 6) 2 通道 SPI。
- 7) 1 通道 IIC 总线接口, 支持多主机。
- 8) 1 通道 IIS 总线音频编码器接口。
- 9) AC'97 编解码器接口。
- 10) 支持 SD 主接口协议 1.0 版和 MMC 卡协议 2.11 的兼容版。
- 11) 2 通道 USB 接口, 1 个主机通道, 1 个 USB 设备通道。
- 12) 4 通道 PWM 定时器和 1 通道内部定时器、看门狗定时器。
- 13) 8 通道 10 位 ADC 和触摸屏接口。

- 14) 实时时钟 RTC。
- 15) 摄像头接口，最大支持 4096×4096 像素输入。
- 16) 130 个通用 I/O 口和 24 通道外部中断源。
- 17) 电源管理，具有普通、慢速、空闲和掉电模式。

18) 时钟，具有 PLL 片上时钟发生器。

19) 中断控制器。

你一定在感叹人类的制造工艺，一定在努力地思考人类是怎么把这么多东西集成在一个指头大小的芯片内的，甚至还集成了总线，可以想象那得多复杂。的确，计算机系统是人类发展史上人类自己做过的最复杂的东西之一。

上面的文字也许不够形象，那么用一幅图来打开我们面前这个黑盒子——S3C2440A，如图 2-4 所示。

可以看到 S3C2440A 这个黑盒子里，还有很多黑盒子，我们不继续打开它们了，等到后面用到它们时，再一一打开。最后这个 S3C2440A 芯片提供了一大堆引脚，这些引脚各有各的功能，它们有的是输入信号的，有的是输出信号的。这个芯片就是这样和开发板上外围部件进行交互的，如从 SDRAM 内存芯片中读取程序指令执行、向 Flash 存储芯片中存入数据，控制音频芯片发出声音，用网卡访问网络等。

或许你还沉浸在漫无边际的思考中，在思考这么多设备和芯片都有些什么功能、又是怎么工作的、如何编程控制它们等，下面继续。

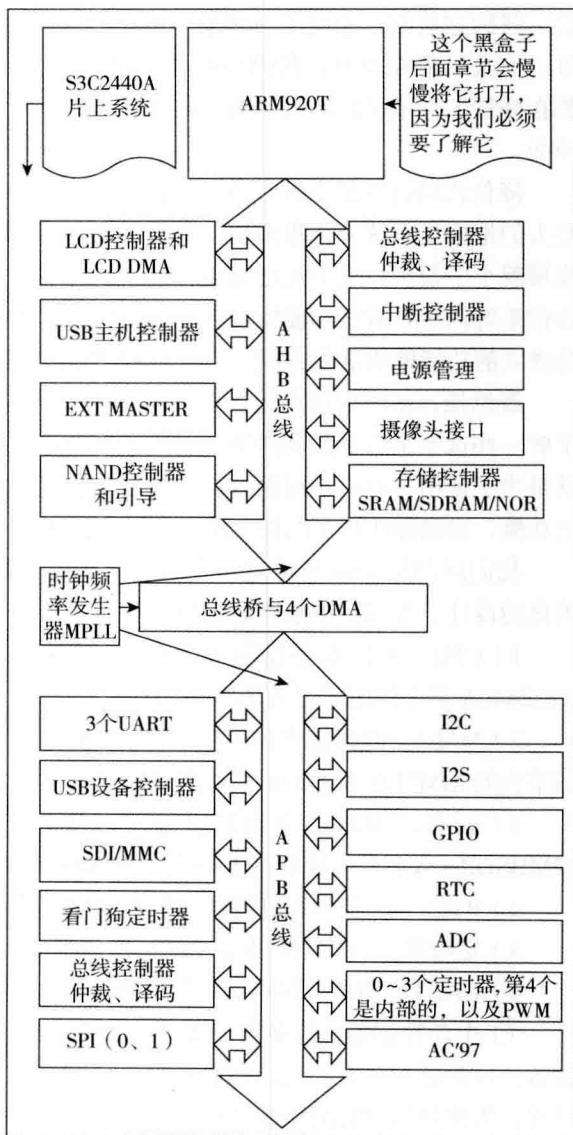


图 2-4 S3C2440A 逻辑结构视图

2.2 必须要关注的硬件

本节将先了解为什么操作系统内核必须要关注或者操作硬件平台中的一些设备，然后分别介绍必须要关注和操作的一些设备的简要信息。

2.2.1 原因

如果现在，在只有 5 行代码的程序中有一个错误，我们很快就能找到并解决它。如果 100 行代码的程序中有一个错误，我们也能容易地找到它。如果 10 000 行代码中有一个错误，找到它就有点难度了。如果 1000 万行代码中有一个错误，要找到它可真不是件容易的事。而且可能在 100 行代码中有 2 个错误，1000 行代码中有 8 个错误，甚至更多。还有可能在修复 1 个错误时产生了新的错误，等等，面对几千万行代码的操作系统，可能会让你疯掉的。

降低代码行数是个好主意，可是用 5 行代码去写个功能完善的操作系统是不可能的。于是人们把操作系统分为很多层，每个层又分为多个模块，这样每个分层里的每个模块的代码规模就小了很多。人们还把最重要的功能和一些机制放在一个独立的模块中，这个模块就是操作系统内核。有些功能模块在需要的时候才去装载，这样一个小而精致的内核最后配合那些独立的功能模块就能实现一个功能相当完善的操作系统。

既然操作系统内核是运行在这个硬件平台上的第一层软件，那么它是肯定要或多或少地了解一些这个平台上的硬件的。它究竟要知道多少，与它设计的规模有关，上面说了那么多就是为了证明一点：小而精致的内核更加稳定可靠。除非有必要，我们应该尽可能减少内核的功能，而把那些功能以独立模块的形式存在。

我们用的是 mini2440 这个平台。那么内核要关注这个平台的哪些硬件呢，后面再介绍内核的设计，先在此想象性地归纳一下：

- 1) CPU。内核也是程序要 CPU 去运行的，当然要知道它的细节，我们平台的 CPU 在 S3C2440A 那个黑盒子里叫 ARM920T。
- 2) MMU。内核是放在内存中的，CPU 要访问内存首先得经过 MMU 内存管理单元，我们平台的 MMU 在 S3C2440A 那个黑盒子里的一个叫 CP15 的黑盒子里。
- 3) 内存。内核本身和其他程序都是放在内存里面的，我们平台的内存是用两片 32MBSDRAM 芯片并接在一起做成的 64MB 的内存。
- 4) RTC。实时时间，内核当然要知道现在是什么时间了。
- 5) 定时器。内核中有很多功能的实现都依赖于定时器，如进程调度器的实现。
- 6) UART，串口。在调试内核时能够输出一些信息，表示内核是否工作正常。
- 7) 中断控制器。设备有时要通知 CPU，或者表示它的任务已经完成，而平台中那么多设备，这就必须要有专门的控制逻辑来控制它们，这就是中断控制器。内核要管理这些设备硬件，当然要知道它们什么时候需要 CPU 的关注，什么时候已经完成它们的任务。

对于一个小而精致的内核来说，知道这么多硬件已经够了。你可能会说，我们都还没设计内核呢，这只是想象中的。是的，万一后面不行了，再加入一些不就行了吗，没必要一步到位。

对上面这些硬件的介绍，除了 CPU 和 MMU 外，其他的都会在本章中以小节的形式先概述它们功能和用途，先让大家有个印象，做个铺垫。等到后面写操作系统内核真正用到它

们时，再详细讨论。CPU 和 MMU，这两个有点复杂，完成的功能很多，而且在开始之前又必须认真地了解它们。所以后面用专用的章节详细讨论。

2.2.2 RTC

RTC 是实时时钟，表示的是我们通常用的时间。比如，人们常说现在什么时间了，就是说的这个 RTC 时间。

说到自然时间，听到最多的可能就是“现在几点啦”。就再也没有往下思考，可以假想一下，物理世界没有了时间，是不是很有趣、也很可怕，可能世间万物会止步不前，可能我们的青春可以永驻。

在 mini2440 开发板上，只有一颗 RTC 备用电池，而完成 RTC 功能的部件在 S3C2440A 黑盒子里面。既然是自然界的时间，那么即使在开发板断电后也不能停止运行，这就是它要有备用电池的原因。

RTC 可以通过 8 位 BCD 数据和 CPU 通信。这些数据包括年、月、日、星期、时、分和秒的时间信息。RTC 单元工作在外部，32.768kHz 晶振，并且可以执行闹钟功能。

我们来看看 S3C2440A 里的 RTC 单元支持哪些功能，归纳如下：

- 1) 时间：年、月、日、星期、时、分和秒，支持 CPU 以 BCD 数据的方式读写。
- 2) 闰年发生器：支持实现和识别哪年是闰年。
- 3) 闹钟功能：能产生闹钟定时中断或者从省电模式下唤醒开发板。
- 4) 已经解决 2000 年是闰年的问题。
- 5) 支持独立电源引脚也就是用备用电池供电的功能。
- 6) 支持时钟节拍中断，如一秒钟内产生固定次数的中断。

S3C2440A 的 RTC 单元的闰年发生器能够基于天、月和年的 BCD 数据，从 28、29、30 或 31 中决定哪个是每月的最后日。此模块决定最后日时会考虑闰年因素。8 位计数器只能表示为 2 个 BCD 数字，因此其不能判决“00”年的问题，即最后两位数为 0 的年份是否为闰年。例如，不能判别 1900 年和 2000 年。请注意 1900 年不是闰年，而 2000 年是闰年。因此，S3C2440A 中 00 的两位数是表示 2000 年，而不是表示 1900 年。

RTC 还能在开发板省电模式中或正常工作模式中在指定的时间上产生一个闹钟信号。在正常工作模式中，只激活闹钟中断信号。在开发板省电模式中，除了激活闹钟中断信号，还激活电源管理唤醒信号。

RTC 时间节拍中断可以用于操作系统内核的时间节拍，也可用于内核时间的同步更新等。

虽然 RTC 单元是操作系统内核必须要关注的硬件之一，但并不在此详细介绍，只是先让大家有个印象，它就是处理自然时间的，还有闹钟等功能。等到写操作系统用到它时再详细地讨论。

2.2.3 定时器

前面说的 RTC 表示的是自然时间，就是我们常说的：“现在什么时间了”，而这次说的定时器是什么呢，它表示的是从现在开始到将来某一个时刻之间的时间，比如，从下午 3 点到下午 4 点之间过去了多少时间。

定时器单元也没有在 mini2440 开发板上，而是在 S3C2440A 这个黑盒子里。

S3C2440A 内部有 5 个 16 位定时器。其中定时器 0、1、2、3 具有脉宽调制功能。定时器 4 是一个无输出引脚的内部定时器。定时器 0 还包含用于大电流驱动的死区发生器。

定时器 0 和 1 共用一个 8 位预分频器，定时器 2、3、4 共用一个另外的 8 位预分频器。每个定时器都有一个可以生成 5 种不同分频信号的时钟分频器。每个定时器模块从相应 8 位预分频器得到时钟，然后再通过时钟分频器分频后得到其自己的时钟信号。8 位预分频器是可编程的。

每个定时器的计数缓冲寄存器中，包含了一个当使能了定时器时被加载到递减计数器中的初始值。定时比较缓冲寄存器中，包含了一个被加载到比较寄存器中与递减计数器相比较的初始值。这种双缓冲特征保证了改变频率时定时器产生稳定的输出。

每个定时器有它自己的由定时器时钟驱动的 16 位递减计数器。当递减计数器到达零时，产生定时器中断请求，通知 CPU 定时器操作已经完成。当定时器计数器到达零时，相应的计数值将自动被加载到递减计数器以继续下一次操作。

这 5 个定时器所有的特性如下：

- 1) 5 个 16 位定时器。
- 2) 2 个 8 位预分频器和两个 4 位分频器。
- 3) 可编程输出波形的占空比控制。
- 4) 自动重载模式或单稳态脉冲模式。
- 5) 死区发生。

定时器内部工作是靠时钟信号驱动的，事实上不只是定时器，CPU 也是，几乎计算平台所有的部件的运行都与时钟信号有关。比如，我们这里的定时器，来一个时钟信号就对自己内部计数器的值减 1，减到 0 时就发出中断通知 CPU。分频器就是对时钟频率进行处理，得到一个比当前更低或者更高的频率，当然定时器的分频器是得到更低的时钟频率，同时配合定时器中的计数值，达到精准的定时功能。定时器输入的时钟频率计算公式如下：

$$\begin{aligned} \text{定时器输入时钟频率} &= \text{PCLK} / \{ \text{预分频值} + 1 \} / \{ \text{分频值} \} \\ \{ \text{预分频值} \} &= 0 \sim 255 \\ \{ \text{分频值} \} &= 2, 4, 8, 16 \end{aligned}$$

PCLK 是 S3C2440A 内部 APB 总线的时钟频率。

这些定时器还可以自动重复定时或者一次性定时，自动重复定时是：比如，开始设定定时器每隔 2s 产生一次中断，中断产生后它会继续计时到下一个 2s 钟又产生中断，如此反复。

一次性定时是指，时间到达产生中断后就不继续定时了，直到下一次设定。

来看看定时器最常用的一个情景：我们让定时器每隔 1ms 产生一次中断，然后在中断处理程序中检查一个进程已经运行了多长时间，分配给这个进程的时间它是否已经用完，如果已经用完，就调度其他进程运行。如果没有定时器，我们在应用软件中写入一个死循环代码，操作系统立马就会被这个应用软件锁死，因为系统已经没有任何手段从这个应用程序手里夺回 CPU 的使用权了。想想吧，这有多可怕！

同样的，操作系统内核，也必须了解这个平台上的定时器，不然就会出现上述情况。但是也不会在这里详述其细节，依然等到用到它时再了解它的编程细节。

2.2.4 串口

一个计算平台有许多输入输出设备：键盘、屏幕、音频、串口甚至网络。当然各种计算平台可能不同，比如 mini2440 开发板就没有像 PC 那样的键盘。

我们选择一个最简单且开发板上又有的设备——串口来介绍。

mini2440 开发板上只有三个用于连接串口线的接口。真正串口功能单元在 S3C2440A 芯片内部，有三个功能相同的串口功能单元。

S3C2440A 中每个串口单元都是可以基于中断或基于 DMA 模式操作的。换句话说，串口可以通过产生中断或 DMA 请求来进行 CPU 和串口之间的数据传输。串口使用系统时钟可以支持最高 115.2Kbit/s 的比特率。如果是外部器件提供的外部时钟，则串口可以工作在更高的速度上。

S3C2440A 的串口包括了可编程波特率，红外发送 / 接收，插入 1 个或 2 个停止位，5 位、6 位、7 位或 8 位的数据宽度以及奇偶校验。

每个串口包含一个波特率发生器、发送器、接收器和一个控制单元。波特率发生器可以由总线时钟或外部输入时钟驱动。发送器和接收器包含了 64B 先进先出缓冲区和数据移位器。将数据写入先进先出缓冲区中接着在发送前复制到发送移位器中。随后将在发送数据引脚移出数据。与此同时从接收数据引脚移入收到的数据，接着从移位器复制到先进先出缓冲区中。

每个串口发送数据是可编程的。由 1 个起始位、5 ~ 8 个数据位、1 个可选奇偶校验位以及 1 ~ 2 个停止位组成的帧，是由行控制寄存器指定的。发送器也可以产生单帧发送期间强制串行输出为逻辑 0 状态的断点状态。

与发送类似，接收数据帧也是可编程的。由 1 个起始位、5 ~ 8 个数据位、1 个可选奇偶校验位以及 1 ~ 2 个停止位组成，也是由行控制寄存器指定的。接收器能够检测出溢出错误、奇偶校验错误、帧错误和断点状态。

这些由 1 个起始位、5 ~ 8 个数据位、1 个可选奇偶校验位以及 1 ~ 2 个停止位组成的帧。在串口单元中，是逐位在串口线上发送的，发送的速度与串口单元的时钟速度有关，时钟速度越快，发送数据的速度越快，如图 2-5 所示。

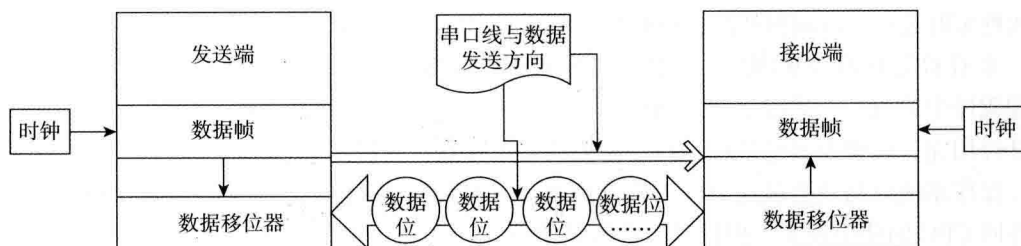


图 2-5 串口设备通信过程

这些数据可能是其他数据，也可能是字符串的 ASCII 码。通常情况下我们发送的就是字符串，这样我们的开发板上的串口线连接上 PC，在 PC 上就能看到开发板输出的信息了。

到时我们的操作系统内核就用 S3C2440A 中的第一个串口作为默认的输出端口。这样就能看到操作系统内核输出的信息了，如系统是否运行正常等，在此先初步认识一下，到我们写代码时再来看看怎么使用它。

2.2.5 中断控制器

先不说中断的概念，现在只要知道中断是各种设备通知 CPU 的一种方式。由于计算平台上不止一个设备，所以需要有一个专门的功能单元来管理这些通知信号（即中断）。让它来决定各设备中断信号的先后次序，以及是否允许各设备的中断信号到达 CPU，还有识别是哪个设备的中断信号。这个功能单元就是中断控制器，如图 2-6 所示。

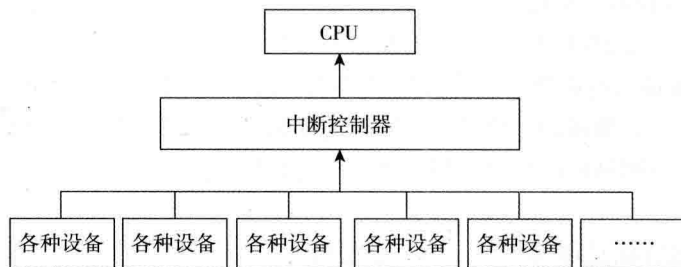


图 2-6 设备与中断控制器、CPU 的连接逻辑视图

mini2440 开发板上没有这个功能单元，它存在于 S3C2440A 这个芯片内部。

S3C2440A 中的中断控制器接收来自 60 个中断源请求。提供这些中断源的是内部外设或者外部设备，如 DMA 控制器、串口、IIC、GPIO 等。外部设备就是通过 GPIO 与 S3C2440A 内部连接起来的。GPIO 是什么，怎么连接，我们不在此处介绍，那也不是必须关心的。

当从内部外设和外部中断请求引脚收到多个中断请求时，中断控制器在仲裁先后次序后

才发送请求到 ARM920T 内核的 FIQ 或 IRQ 引脚, ARM920T 内核就是图 2-6 中的 CPU。这个后面会介绍。仲裁步骤由硬件优先级逻辑决定, 并且将结果写入中断挂起寄存器中, 从而通告用户是中断源中的哪一个发生了中断。

中断优先级仲裁, 就是当两个不同的设备同时产生中断时, 而 CPU 一次只能响应一个中断请求, 中断控制器要在两个或者多个中断信号中选择一个, 然后把那个中断请求发送给 CPU。和自然世界中一样, 事情有轻重缓急, 当然中断控制器最好选择一个要求 CPU 尽可能快地处理其中断请求的设备。

S3C2440A 中的中断控制器只能连接 32 根中断信号线, 那么它是怎么支持 60 个中断源的呢? 它采用了二级中断源, 就是在一级中断的几根中断信号线上连接几个设备的中断信号线。所以当中断发生时, 我们就不能单纯地认为是一个设备发生了中断, 而是要进一步根据次级中断的相关信息确定是哪个设备发生了中断, 如图 2-7 所示。

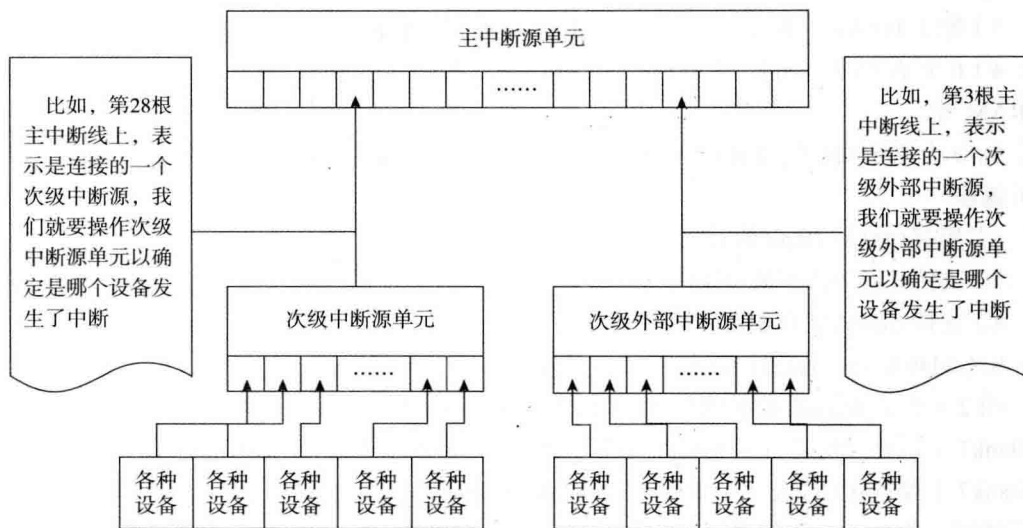


图 2-7 S3C2440A 的中断控制器与设备的连接

操作系统内核的重要功能就是要代替应用软件操控设备, 然而设备与 CPU 的通信大多数是由中断信号来驱动的。中断控制器就是它们之间的必经之路。所以操作系统内核必须要了解并控制中断控制器。在此有个初步印象即可, 后面介绍操作系统内核的中断管理时, 会详细介绍。

2.2.6 SDRAM

SDRAM 是同步动态随机存储器, 同步是指它工作需要同步时钟, 内部命令的发送与数据的传输都是以这个时钟为基准的。动态是指它内部的存储阵列需要不断地刷新来保证数据

不丢失。它是可以随机访问的，就是说可以自由地在指定的地址上进行数据读写。当然也不是绝对自由的，这个访问地址是由存储控制器和总线决定的。我们通常说的内存就是用这种芯片和技术实现的。

mini2440 开发板有 64MB 的 SDRAM，它是用两片 32MB 的 SDRAM 芯片组成的，这两片 32MB 的 SDRAM 芯片存在于开发板上，而它的存储控制功能单元则存在于 S3C2440A 芯片内部。通过 S3C2440A 芯片的引脚和外部的 SDRAM 芯片通信的。

我们先介绍 S3C2440A 芯片内部的存储控制功能单元，再来介绍 mini2440 开发板上的两片 SDRAM 芯片。

S3C2440A 芯片内的存储器控制器为访问外部存储器提供了存储器控制的信号。

S3C2440A 芯片内的存储器控制器包含以下特性：

- 1) 可通过软件选择大端、小端。
- 2) 总共 8 个存储器 Bank，每个 Bank 有 128MB，总共 1GB 地址空间。
- 3) 除了 Bank0 (16/32 位)，其他全部 Bank 都可编程访问宽度 (8/16/32 位)。
- 4) 6 个存储器 Bank 为 ROM、SRAM 等。其余 2 个存储器 Bank 为 ROM、SRAM、SDRAM 等。
- 5) 7 个固定的存储器 Bank 起始地址，1 个可变的存储器 Bank 起始地址，并且 Bank 大小可编程。
- 6) 所有存储器 Bank 的访问周期可编程。
- 7) 支持外部等待扩展总线周期。
- 8) 支持 SDRAM 自刷新和掉电模式。

S3C2440A 的存储地址空间如图 2-8 所示。

图 2-8 中的 SROM 表示的是这个 Bank 只能连接 SRAM 或者 ROM 类型的芯片。Bank6 和 Bank7 上所连接的芯片大小必须相等。比如，Bank6 上连接是 32MB 的 SDRAM，那么如果 Bank7 上要连接存储芯片，也必须是 32MB 的 SDRAM。图 2-8 也是 mini2440 开发板上的实际情况。除 Norflash 和两片 32MB 的 SDRAM 芯片外，其余的都在 S3C2440A 芯片内部，Norflash 和两片 32MB 的 SDRAM 芯片是在开发板上的。

上面简单说明了 S3C2440A 的存储控制器和存储系统的物理地址空间，接下来看看 SDRAM 芯片。数据和程序都是放在它里面运行的，当然操作系统内核也毫不例外，也是放在它里面的。

mini2440 开发板使用了两片外接的 32MB 的 SDRAM 芯片，型号为 HY57V561620 或者 MT48LC16M16A2，它们并接在一起形成 32 位的总线数据宽度，这样可以增加访问的速度，并且它们的物理起始地址为 0x30000000。可能不同时期出厂的开发板用 SDRAM 芯片型号不一样，但是功能是一样的。

SDRAM 内部是一个存储阵列。可以把它想象成一个表格。和表格的检索原理一样，先指定行，再指定列，就可以准确找到所需要的存储单元。这个表格称为逻辑 Bank。

目前的SDRAM 基本都是4个Bank。寻址的流程就是先指定Bank地址，再指定行地址，最后指定列地址。这就是SDRAM 的寻址原理。笔者的开发板是用的SDRAM 芯片是32M的HY57V561620，这是一个4Banks $\times$ 4MB $\times$ 16bit的SDRAM，也就是由4个逻辑块（即Logical Bank，简称L-Bank）组成的，每个L-Bank有4MB存储单元。2片总共空间为64MB，连接到S3C2440A的Bank6上，所以访问的物理地址空间为0x30000000~0x33FFFFFF。SDRAM 芯片结构如图2-9所示。图有点专业，暂时看不懂也没什么问题，有个印象即可。

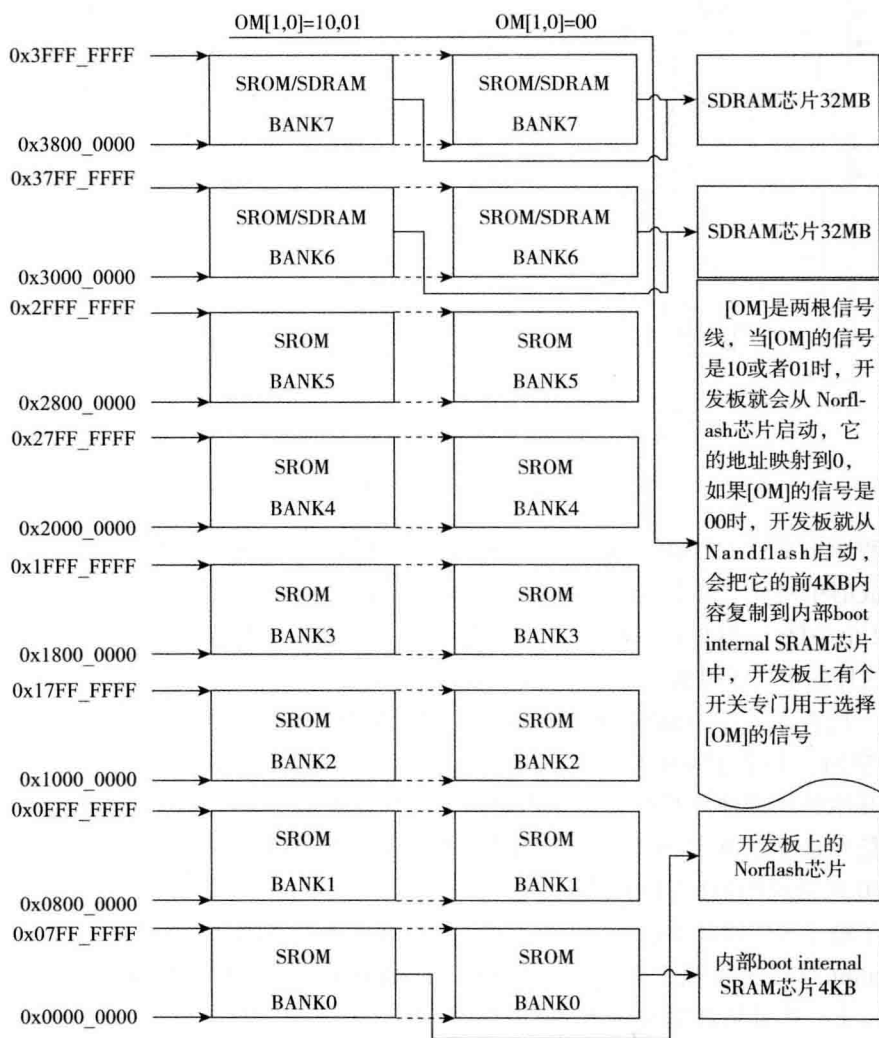


图 2-8 S3C2440A 的存储地址空间

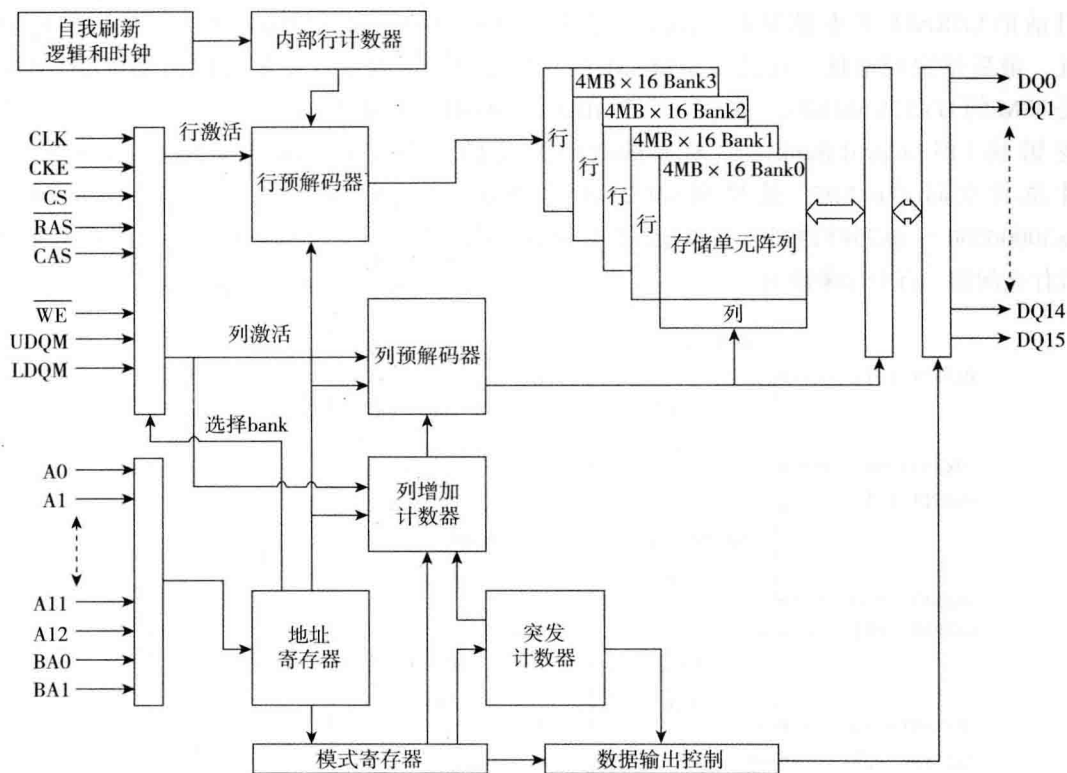


图 2-9 SDRAM 芯片结构

对于程序员来说，只要认为它就是内存并且是这样的：由 8 个可以存储 1 位二进制数据的空间组成的小格子，每个小格子有个地址，通过这个地址可以向小格子里读写 8 位二进制的数，2 个小格子就可以存放 16 位二进制数据、4 个小格子就可以存放 32 位二进制数据。最后许多这样的小格子组成一个很大的存储空间，如图 2-10 所示。

注意，内存芯片（即 SDRAM）并不决定内存的物理地址从哪里开始，它只是一个存放数据的空间，如果 32MB 大小的内存，每个字节一个地址，那么它就有 32M 个地址编码。内存开始的物理地址是存储控制器和内存芯片地址信号线决定的，比如，存储控制器地址译码信号是从 0x30000000 地址开始的，那么这个内存可以访问的有效地址空间就是从 0x30000000 开始到 0x31FFFFFF 结束。

我们了解了 S3C2440A 芯片的地址空间，又粗略地看了 SDRAM 芯片的内部结构，最后只要知道 mini2440 开发板上有两片 32MB 的 SDRAM 芯片，并且它们并接到 S3C2440A 芯片的 Bank6 上。因此地址空间是 0x30000000 ~ 0x33FFFFFF。操作系统内核、应用软件、数据，都是放在这个里面的，CPU 就能通过内存地址运行其中的程序和操作其中的数据。后面章节中还有更多关于内存的内容，到时会详细介绍的。

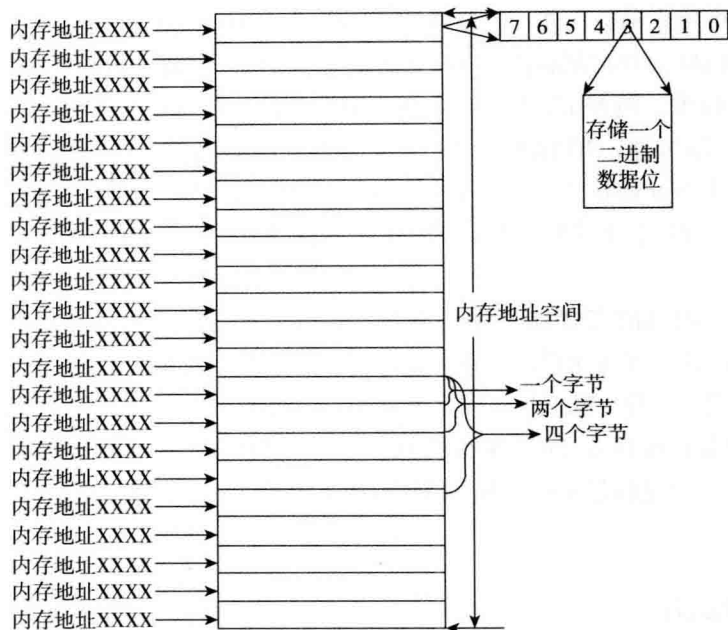


图 2-10 内存逻辑视图

2.2.7 Norflash

Flash 存储器称为闪存，它结合了 ROM 和 RAM 的特点，不仅具备电子可擦除、可编程的特点，还可以快速读写数据，而且数据不会因为断电而丢失。这种特性使它可以用于 U 盘、MP3、智能手机等众多移动智能设备上。

目前 Flash 闪存芯片主要有两种：Norflash 和 Nandflash。下面介绍 Norflash。

mini2440 开发板上有一片 2MB 大小的 Norflash 芯片，它连接在 S3C2440A 的 Bank0 上，并且可以通过跳线控制。

mini2440 开发板上的 Norflash 芯片特性如下：

- 1) 有 22 根地址信号线 和 16 根数据信号线。
- 2) 掉电不会丢失数据。
- 3) 支持“片上运行”。

由于它有 22 根地址信号线和 16 根数据信号线，所以它能索引 Norflash 芯片上每个存储字，一次可以传输 16 位数据，刚好是一个字。这就像 SDRAM 芯片一样，可以被 CPU 直接读写。掉电也不会丢失数据。这样写入其中的代码和数据就可以长久地保存。如果它里面包含可执行代码，那么就可以在不需要 SDRAM 芯片的情况下执行程序。什么情况下需要这样的存储芯片呢？例如，PC 上的 BIOS 系统，CPU 运行的第一个程序就是 BIOS 程序，那个时候连内存都还没初始化呢。再就是 mini2440 嵌入式开发板的上引导程序，也可以说是 BIOS，

它也完成了 BIOS 绝大部分工作，只是叫法不同而已。mini2440 开发板在上电时，S3C2440 芯片时钟和 SDRAM 芯片控制器都还没有初始化呢。所以要借助 Norflash 存储技术。

利用 JTAG 技术，向 Norflash 芯片中烧入引导程序，这个引导程序负责初始化开发板上的各种设备，如时钟、SDRAM、串口等。初始化完设备之后，就装载操作系统内核。mini2440 开发板上 Norflash 芯片中烧写的是 superVIVI 引导程序。这个引导程序功能非常多，我们要用到的是从 PC 上下载的操作系统内核到 mini2440 开发板上的 SDRAM 芯片中，这非常方便。

Norflash 芯片内部的实现细节我们不需要知道，我们只要知道 mini2440 开发板上有 2MB 的 Norflash 芯片，它就相当于掉电不丢失数据的内存，当然它的速度赶不上内存，它里面存放的是一个引导程序。我们还可以通过拨动 mini2440 开发板上一个开关从 Norflash 芯片开始启动，并且将它连接在 S3C2440A 的 Bank0 上，开始的物理地址是 0，S3C2440A 中的 ARM920TCPU，一上电就是从地址 0 开始执行第 1 条指令，这样刚好就运行了 Norflash 芯片中的引导程序。

2.2.8 Nandflash

Nandflash 也是 Flash 闪存的一种，其内部采用非线性单元设计模式，为固态大容量闪存的实现提供了廉价有效的解决方案。Nandflash 存储器具有容量较大、改写速度快等优点，适用于大量数据的存储，广泛应用于嵌入式产品中。

由于内部结构相对简单，Nandflash 芯片使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。所以不能像 Norflash、SDRAM 等芯片，可以直接和 CPU 等设备相连并且寻址到每个存储字节。因此 Nandflash 芯片需要为它设计专门的控制器才能和其他设备进行通信。

mini2440 开发板上有两种 Flash 闪存，一种是前面说过的 Norflash 闪存，大小为 2MB；另一种是 Nandflash，型号为 K9F1G08，大小为 128MB，旧版本为 K9F1208，大小为 64MB。Norflash 和 Nandflash 闪存的芯片都存在于 mini2440 开发板上，前面说过 Norflash 有足够的地址引脚和数据引脚，因此它可以直接提供类似 SRAM 接口和 CPU 相连、通信。而 Nandflash 闪存则不行，它需要专门的 Nandflash 控制器，这个 Nandflash 控制器就存在于 S3C2440A 这个芯片中。S3C2440A 支持这两种 Flash 启动系统，通过拨动跳线开关，可以选择从 Norflash 还是从 Nandflash 启动系统。实际的产品中使用一片 Nandflash 就够了，因为 mini2440 开发板是为了方便用户开发学习，所以还保留了 Norflash。

要操作 Nandflash 芯片，从中读取和写入数据。那么就要 Nandflash 控制器进行操作，对它进行编程，通过 Nandflash 控制器操作 Nandflash 芯片。对 S3C2440A 芯片上的 Nandflash 控制器的编程步骤如下：

写命令寄存器，向 Nandflash 控制器写入相关命令，对应于 Nandflash 芯片的命令周期。

写地址寄存器，向 Nandflash 控制器写入相关地址，比如上面输入的读写命令是需要读写地址的，对应于 Nandflash 芯片的地址周期。

读/写数据寄存器，读/写入数据到 Nandflash 存储器，根据上面相应的命令做相应的动作，可能是读也可能是写数据，对应于 Nandflash 芯片的读/写周期。

读 Nandflash 控制器的主 ECC 寄存器和备份 ECC 寄存器。对数据进行校验。

等用到 Nandflash 控制器时，再详细研究其细节。下面介绍 Nandflash 芯片的结构。

Nandflash 芯片的数据是以位的方式保存在存储单元中的，一般来说，一个存储单元中只能存储一个位。这些存储单元以 8 个或者 16 个为单位，连接成位行，这些位行会再组成页，这些页面会再组成块。Nandflash 芯片有多种结构，笔者使用的 Nandflash 芯片是 K9F1208，它是这样的：每页 512B，其中 512B 用于存放数据，还有 16B 存放这个页面的数据的校验信息，每 32 个页面形成一个块，块大小为 $32 \times 512\text{B}$ 。具体一片 Flash 上有多少个块是根据需要决定的。K9F1208 的 Nandflash 芯片具有 4096 个块，故总容量为 $4096 \times (32 \times 512\text{B})$ 一共是 66MB，但是其中的 2MB 是用来保存 ECC 校验码等额外数据的，故实际中可使用的空间为 64MB，如图 2-11 所示。

Nandflash 芯片以页为单位读写数据，所以一次读写最少为 512B 的数据，不能像 Norflash 芯片一样可以以字为单位读写数据。而 Nandflash 芯片是以块为单位擦除数据的。按照这样的组织方式可以形成所谓的三类地址：

- 1) 列地址。
- 2) 页地址。
- 3) 块地址。

对于 Nandflash 来说，地址和命令只能在 I/O[7:0] 引脚上传递，数据宽度是 8 位。

由于 Nandflash 芯片的工艺不能保证存储单元在其生命周期中保持性能的可靠，所以在 Nandflash 芯片的生产及使用过程中会产生坏块。为了检测其可靠性，在应用 Nandflash 芯片的系统中一般都会采用一定的坏区管理策略，而管理坏区的前提是能比较可靠地进行坏区检测。如果操作时序和电路稳定性不存在问题，Nandflash 芯片出错的时

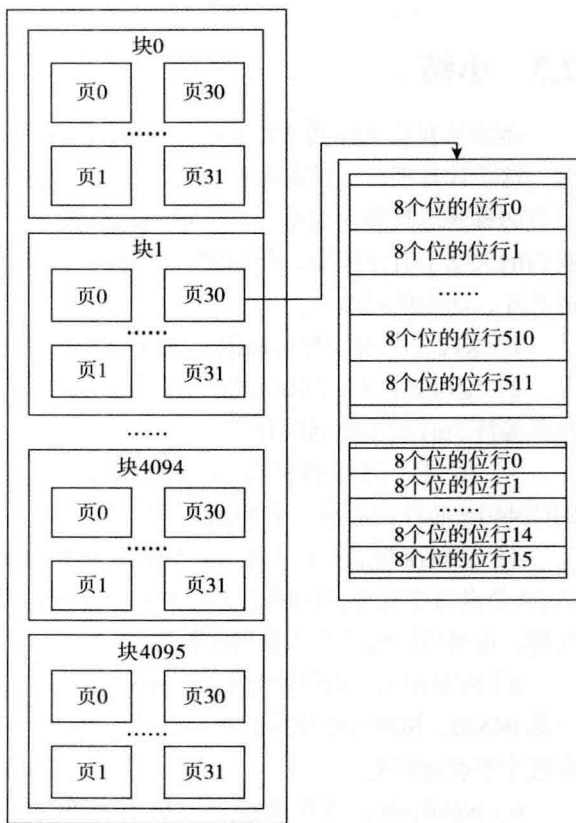


图 2-11 Nandflash 芯片逻辑视图

候一般不会造成整个块或页面不能读取或全部出错，而是整个页面（如 512B）中只有一位或几位出错。对数据的校验常用的有奇偶校验、CRC 校验等，而在 Nandflash 芯片处理中，一般使用一种比较专业的 ECC 校验。ECC 能纠正单比特错误和检测双比特错误，而且计算速度很快。

我们只要知道 Nandflash 芯片就和 PC 中硬盘一样，只能一次以页面为单位读取数据，最少是 512B。CPU 要访问 Nandflash 芯片需要通过专门的 Nandflash 控制器，其动作是对 Nandflash 控制器发送相关的命令。Nandflash 芯片就说到这里吧，大脑中有个印象即可。

2.3 小结

本章从我们选择的计算平台——mini2440 开发板开始，描述了为什么要选择这个计算平台，这个计算平台上都有些什么芯片，我们的操作系统内核要运行起来必须要关注这个计算平台的哪些芯片和功能单元，分别简单地概述了这些必须要关注的芯片、功能单元，以便于我们的大脑中有个印象，使用到它们时，知道它们是什么、有什么功能和作用。我们了解到的芯片、功能单元如下：

1) RTC：处理自然时间的，可以用备用电池供电维持时间。

2) 定时器：S3C2440A 芯片内部有 5 个定时器，可以用于处理间隔时间，比如，一个进程要等待 5ms 后去访问硬件等。

3) 串口：用于和其他设备通信或者传输数据，比如，连接 PC 输出信息，并且 S3C2440A 芯片内部有三个相同的串口硬件。

4) 中断控制器：存在于 S3C2440A 芯片内部，是设备和 CPU 通信的重要机制，支持多达 60 个设备中断源，同时也支持中断优先级，哪些设备的中断要尽快发送给 CPU 内核进行处理，也可以屏蔽某个设备的中断信号。

5) SDRAM：即常说的内存，mini2440 开发板用了两片 32MBSDRAM 芯片并接在一起，一共 64MB，SDRAM 开始的物理地址是 0x30000000。操作系统内核和各个应用软件都是放在这个里面运行的。

6) Norflash：就像是掉电不会丢失数据的内存，支持在其上运行程序。mini2440 开发板上有 2MB 的 Norflash 芯片，连接在 S3C2440A 芯片的存储控制器的 Bank0 上，并能通过跳线开关控制。这个芯片里烧写了一个叫 superVIVI 的开发板引导程序，相当于 PC 上的 BIOS 程序，它负责初始化开发板的时钟和 SDRAM 等其他设备，然后加载操作系统内核。

7) Nandflash：它需要专门设计的控制器来进行读写，一次读写最少是 1 页，1 页最少是 528B，512B 数据加 16B 的校验信息，32 页组成一个块。我们的开发板上这个芯片是 64MB。可以用于操作系统构建文件系统，在其上存放用户数据、应用软件、操作系统

等。它 also 支持系统引导，S3C2440A 芯片自动把它的前 4KB 空间的数据读到一个内部 4KB 的 SRAM 中，把那个 SRAM 的首地址映射为 0，S3C2440A 芯片内部的 CPU 内核就从内部 4KB 的 SRAM 中开始运行了。

讲到这里，已经对我们的计算平台有了初步的印象，了解组建一个计算平台需要些什么，同时初步知道了一个操作系统内核至少要实现什么功能，以及必须要与之打交道的硬件等。

处 理 器

一个计算平台最重要的部分，是要有一颗性能良好、功能强大的处理器，就是通常说的 CPU，它完成各种数据处理、计算、控制等任务。既然我们要写一个具体计算平台上的操作系统内核，就必须仔细了解它。我们的这个计算平台上的 CPU 是 ARM 公司设计的。那么，首先从这个公司开始，接着了解我们要关注的处理器 ARM920T 及其内部结构，知道它的内部结构并了解它的存储体系、运行状态、工作模式、有哪些寄存器，还有它的中断异常处理机制、有多少条指令以及 MMU 和 Cache 等。

本章将详细讨论处理器的细节。

3.1 ARM 公司与其处理器

ARM 即 Advanced RISC Machines，是一种基于精简指令集的处理器，通常也指这种处理器体系，同时也可代表开发这种处理器体系的公司。

1990 年，12 个富有远见和梦想的人，在英国剑桥大学的一个谷仓之中成立了 ARM 公司。ARM 是微处理器行业的一家知名企业，设计了大量高性能、廉价、耗能低的 RISC 处理器，以及相关的技术与软件。

ARM 的商业模式，主要是出售芯片设计技术的授权，而非生产和销售实际的半导体芯片。ARM 向所有合作伙伴授予 ARM 技术知识产权许可证。这些合作伙伴来自世界领先的半导体公司和各大 IT 企业。这些合作伙伴可利用 ARM 技术的授权设计创造和生产片上系统，但需要向 ARM 公司支付技术授权的许可费用并为生产的每块芯片交纳版税。除了处理器技术授权，ARM 还提供了一系列软件工具。正因为如此，基于 ARM 解决方案的芯片和软件体

系十分庞大。

采用 ARM 技术授权的微处理器，就是我们通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，如 MP3、数字机顶盒、智能手机、汽车多媒体和制动系统等，基于 ARM 技术的微处理器应用占据了 32 位 RISC 微处理器 90% 以上的市场份额，ARM 技术正在逐步渗入我们生活的各个方面。

ARM 处理器到今天已经有 8 个体系版本了。这些体系版本各有特点，比如，有的支持 32 位地址空间（最开始只有 26 位地址空间），有的支持乘法指令，有的支持调试接口，有的支持 Thumb 精简指令集等。

根据这些体系版本，各大生产商生产了不同的 ARM 微处理器系列，如下：

- ☐ ARM Securcore 系列。
- ☐ ARM7 系列。
- ☐ ARM9 系列。
- ☐ ARM11 系列。
- ☐ Cortex-M 系列。
- ☐ Cortex-R 系列。
- ☐ Cortex-A 系列。

关于 ARM 公司就说到这里，因为这个公司并不是我们关注的重点，我们要关注的是这个公司的微处理器。

3.2 ARM920T 的结构与特性

三星 S3C2440A 芯片自称处理器，但其实它是采用了 ARM 公司设计的 ARM920T CPU 加上一些其他组件，然后把它们封装在一起形成的一个多功能芯片。这里关注的是 ARM920T CPU，下面就去看一看它的内部结构。

3.2.1 ARM920T CPU 结构

mini2440 开发板采用了 S3C2440A 芯片，而 S3C2440A 芯片又采用了 ARM 公司设计的 ARM920T CPU。上次研究了计算平台——mini2440 开发板。发现上面有个叫 S3C2440A 的黑盒子，并打开了它，发现它里面又有许多黑盒子，并且对其中一些黑盒子做了简要的介绍，如图 3-1 所示。另外，为了便于阅读，把 2-4 的图又搬过来，笔者觉得虽然增加篇幅，但是节省了回去翻书的时间。

图 3-1 就是 S3C2440A 这个黑盒子内部结构图，其实早已看过，不过上次我们没有继续拆开它里面的黑盒子。其中有个非常重要的黑盒子——ARM920T，它才是这次要关注的重点，下面打开它，看看它里面又有什么，如图 3-2 所示。

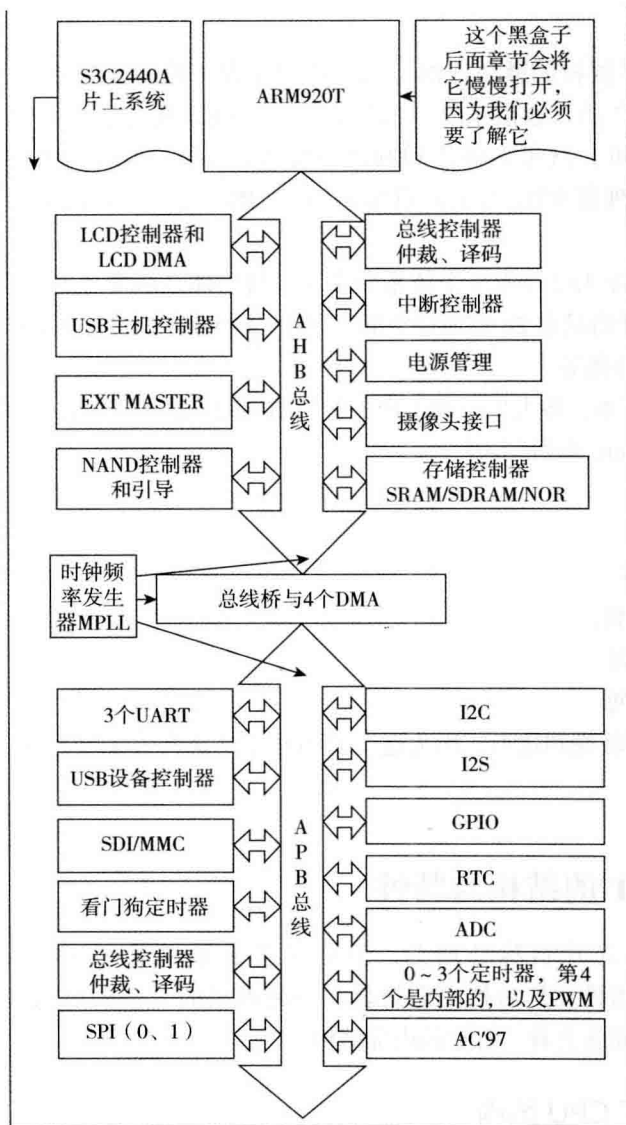


图 3-1 S3C2440A 逻辑结构视图

ARM920T 里面的黑盒子以及功能单元如下:

- 1) JTAG: 一种调试或者测试芯片的接口, 可以通过 JTAG 接口访问 CPU 的内部寄存器和挂在总线上的设备, 如 Flash、RAM 等, 我们不必过多关注。
- 2) ARM9TMDI: 它才是真正运行程序的。
- 3) 指令、数据 MMU: 包括两个 C13 一起处理内存地址映射的。
- 4) 指令、数据 Cache: 包括写回 PATAG RAM、写缓冲, 一起完成高速缓存的功能。

5) CP15：这里是用于控制 MMU 和 Cache 的，还可以控制别的器件，不同的系统配置下是不同的。

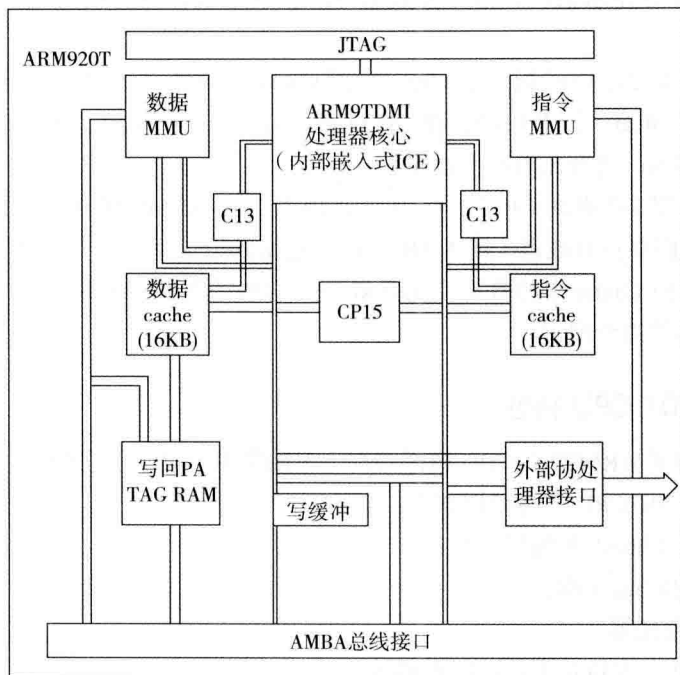


图 3-2 ARM920T 逻辑结构视图

6) 外部协处理器接口：ARM920T 可以加入多达 15 个协处理器，如增加一个浮点运算处理器等。

7) AMBA 总线接口：ARM920T 内部就是通过它和外部内存及其他设备通信的。

不过不需要把所有的东西都弄得一清二楚。我们主要关注 4 个部分：① CPU 内核：和 ARM920T 内部的其他部件一起工作，用来运行程序。包括操作系统内核和应用程序。② MMU：映射内存，主要用于实现虚拟内存，什么是虚拟内存后面会介绍的。③ Cache：缓存内存中的指令和数据，它比内存小很多但快很多，把内存中的一部分内容放在 Cache 里面，这样可以加快程序的运行速度，后面也会详细介绍的。④ CP15：它在 ARM920T 内部称为协处理器，在这里是专门用于控制 MMU 和 Cache 的。可以看到 ARM920T 还能通过外部协处理器接口增加其他协处理器。

CPU 内核对应于图 3-2 中的 ARM9TMDI。操作系统和应用程序都需要它执行，所以我们写操作系统内核之前，必须先仔细了解它，以及 ARM920T 的其他组件。后面会重点研究 ARM920T 的程序员模型，即指令集、工作模式、寄存器等。

MMU 对应于图 3-2 中的指令 MMU、数据 MMU。为什么有两个 MMU，通常情况下一

个 MMU 不就够了吗？是的，大部分 CPU 确实只有一个 MMU，然而这里有两个，这是因为 ARM920T 内部采用了哈佛架构。哈佛架构是一种将程序指令存储和数据存储分开的存储器架构。是一种并行体系架构，指令和数据可以独立存取，这样就大大提高了程序的运行性能。

Cache 对应于图 3-2 中的指令 Cache、数据 Cache，同样也是因为哈佛架构才会有两个 Cache。它们都是 16KB。关于 Cache 的工作机制，后面会专门介绍，现在只要知道它是用来缓存内存中的数据，速度比内存快很多。

CP15 对应于图 3-2 中的 CP15，称为协处理器。在 ARM920T 中，CP15 控制着 MMU 和 Cache，比如我们要打开或者关闭 MMU、改变地址映射关系、打开或者关闭 Cache、锁定 Cache 中的内容、使 Cache 中的数据变为无效等。这些功能都将通过操作 CP15 协处理器而达到，后面会专门详细介绍它。

3.2.2 ARM920T CPU 特性

我们大致了解了 ARM920T CPU 内部结构，下面简单了解一下它的特性。

首先，ARM920T CPU 支持特性如下：

- ❑ 支持 16 位 Thumb 压缩指令集。
- ❑ 支持片上 Debug（调试）。
- ❑ 内嵌硬件乘法器。
- ❑ 嵌入式 ICE，支持片上辅助断点调试。

随着后面的介绍，会逐步了解这些特性。

CPU 是 RISC 体系，即精简指令集体系。当然与之对应就是 CISC 体系，即复杂指令集体系，CISC 体系有个著名的实例——早期 X86 CPU，而现在的 X86 CPU 不完全是 CISC，它也结合了 RISC 优点。CISC 体系的指令多、指令长短不一、每条指令完成的功能相当多等。这里不过多讨论 CISC 体系。

ARM920T CPU 是 RISC 体系但也改进了 RISC 体系。RISC 体系特性及 ARM 改进的特性如下：

- ❑ 指令长度固定。
- ❑ 有大量的寄存器。
- ❑ 大多数操作都在寄存器中进行。
- ❑ 通过装载、存储两个操作，在寄存器和内存之间交换数据。
- ❑ 内存寻址方式简单。
- ❑ ARM 改进了，在一条数据处理指令中同时加入算术、逻辑、移位等操作。
- ❑ 使用地址自动增加或者减少技术，ARM 改进和优化了程序中的循环处理。
- ❑ ARM 改进了装载和存储指令，使之能一次装载或者存储多个内存字，由此增加了数据传送性能。

□ ARM 还让大部分指令能根据前一条指令执行的结果状态，决定是否执行当前这条指令。这大大提高了程序的运行效率。

上面这多的特性，可能一时半会还不能全明白。因为计算机的知识体系从来就没有顺序，它们有时是互相交错的，所以暂时不明白也没事儿，只要我们不放弃，走完这一段路，再回首，就会全明白的。

ARM920T 内部结构和它的一些简单的特性就介绍到这里，下面进行仔细研究，这个有点难，但是慢慢来，要相信自己。要知道，处理器从来就不是简单的。

3.3 ARM920T 存储体系

处理器主要是运行程序和处理数据的，程序和数据大多数情况下是放在存储器中的，处理器要运行程序和处理数据，首先要解决访问存储器的一系列问题。

ARM920T 存储体系有点复杂，因为它里面还有 MMU 和 Cache，这些玩意儿的关系很复杂。开始时我们还不能一竿子捅到底，得一步一步慢慢来，先把 MMU 和 Cache 从 ARM920T 中拿掉，相当于 ARM920T 中没有了 MMU 和 Cache 这些部件。本节都是假定 ARM920T 中没有 MMU 和 Cache。这两个部件，后面章节会详细介绍。到时再把它们联系起来。

3.3.1 ARM920T 地址空间

开始之前，我们先看看 ARM 体系中的字节、半字、字、双字，它们究竟表示能存放多少二进制位。可能其他 CPU 体系与此不同。

- 字节：能存放 8 个二进制位。
- 半字：能存放 16 个二进制位。
- 字：能存放 32 个二进制位。
- 双字：能存放 64 个二进制位。

ARM920T CPU 使用的是单一扁平的地址空间。由于 ARM920T CPU 是 32 位的，所以它使用了 $0 \sim 2^{32}-1$ 的无符号整数来表示一个地址值。其地址空间大小是 2^{32} 个，可以存放 8 个二进制位的字节。可以把它想象成线性的字节数组，用伪 C 语言描述是这样的：`unsigned char a[232]`。

ARM920T 的地址空间也可以看成是 2^{30} 个 32 位的字单元，用伪 C 语言描述是这样的：`unsigned int a[230]`。每个字单元的地址都能被 4 整除，即地址值的最低两个二进制位是 00。这个 32 位的数据如果从 0 地址开始存放，那么地址 0、地址 1、地址 2、地址 3 就包含这整个 32 位的数据。

ARM920T 的地址空间还可以看成是 2^{31} 个 16 位的半字单元，用伪 C 语言描述是这样的：`unsigned short int a[231]`。每个半字单元的地址都能被 2 整除，即地址值的最低 1 个二进

制位是 0。这个 16 位的数据如果从 0 地址开始存放，那么地址 0、地址 1 就包含这整个 16 位的数据。

3.3.2 ARM920T 存储器格式

要运行程序就得把程序指令和一些数据存放在存储器中，这个存储器大多数情况下是内存（即 SDRAM 芯片），下面我们就认识这里的存储器（即内存），如数据在内存中是怎么存放的，有些什么方式，这都是问题。

先来看看一个字节数据：“0xFF”怎么存放在内存中的。假如数据都是从内存地址 0 开始存放的。因为内存可以寻址的最小单位是 1 字节，而数据也只有 1 个字节，因此只有一种情况，如图 3-3 所示。

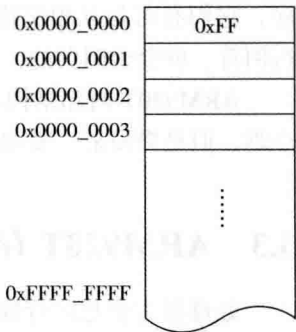


图 3-3 字节数据存储

然后来看一个 16 位的半字数据：“0xEEFF”怎么存放在内存中的。因为数据有了 2 个字节，放在内存中也要占用 2 个字节。因此有两种情况，如图 3-4 所示。

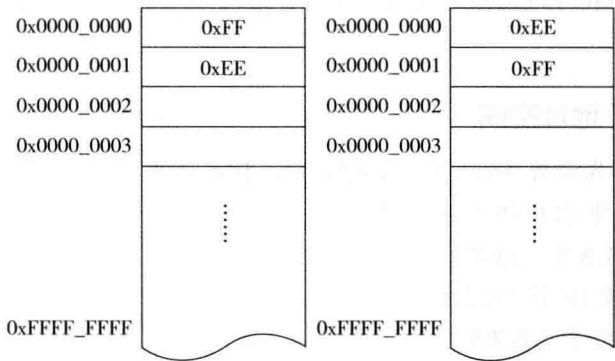


图 3-4 半字数据存储

图 3-4 中，一种情况是半字数据“0xEEFF”的最高有效位放在内存的高地址字节中。第二种情况是半字数据“0xEEFF”的最高有效位放在内存的低地址字节中。

最后来看一个 32 位的字数据：“0xCCDDEEFF”怎么存放在内存中的。因为数据有了 4 个字节，放在内存中也要占用 4 个字节。因此还是有至少两种情况，如图 3-5 所示。

和前述一样，还是有两种情况分别是数据的最高有效位放在内存的高地址字节中，或者数据最高有效位放在内存的低地址字节中。

人们常说，某某处理器是大端或者小端的。表示的就是上面的这些情况。

- ❑ 小端模式：数据的最高有效位放在内存的高地址字节中。
- ❑ 大端模式：数据的最高有效位放在内存的低地址字节中。

ARM920T CPU 既支持小端模式也支持大端模式。这是通过 CP15 协处理器控制的，后面会介绍怎么对它进行编程，达到这一目的。

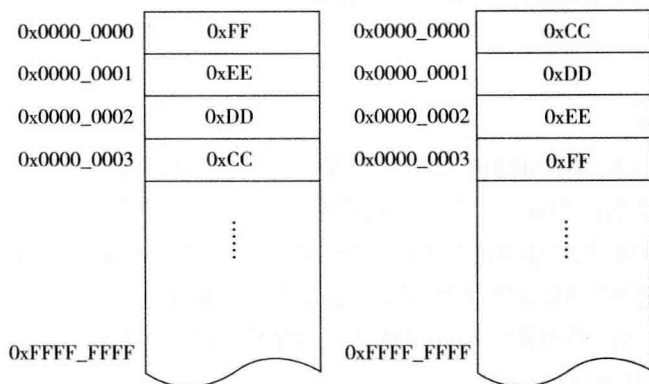


图 3-5 字数据存储

3.3.3 ARM920T 存储地址对齐

在 ARM920T 中访问存储器时通常希望的是：如果是字单元访问，那么这个访存地址就应该是字对齐的，即地址的最低两个二进制位是 00；如果是半字单元访问，那么这个访存地址就应该是半字对齐的，即地址的最低二进制位是 0；如果字单元访问或者半字单元访问的访存地址不是字对齐或者半字对齐，这就称为地址非对齐的访存操作。有很多处理器是非对齐的访存操作，最多影响处理器的性能，不会导致程序运行结果错误，而 ARM920T 是不允许非对齐的访存操作的，对它而言这会带来灾难性的后果。

当 ARM920T 处于 ARM 状态时（关于 ARM 状态和 Thumb 状态后面会有介绍），ARM920T 运行的是 ARM 指令集，每条指令长度都是 32 位，即一个字。这时写入 PC 寄存器（即程序指针寄存器中的值）如果不是字对齐的，要么程序的运行结果不可预知，要么地址的最低两个二进制位被硬件忽略。当 ARM920T 处于 Thumb 状态时，ARM920T 运行的是 Thumb 指令集，每条指令长度都是 16 位，即一个半字。这时写入 PC 寄存器中的值如果不是半字对齐的。也和 ARM 状态时一样，要么程序的运行结果不可预知，或者地址的最低二进制位被硬件忽略。

3.4 ARM920T 状态

ARM920T 处理器有两种状态：ARM 状态和 Thumb 状态。这两种状态之间可以通过软件或者硬件来回切换。ARM920T 处理器根据 CPSR 寄存器中的 T 位来分辨这两种状态。先不必管这是个什么寄存器，后面会有介绍。

□ CPSR 中 T 位为 0: 表示 ARM920T 运行在 ARM 状态。

□ CPSR 中 T 位为 1: 表示 ARM920T 运行在 Thumb 状态。

要注意的是, 这里说的处理器状态和后面要介绍的处理器的工作模式是两码事, 不可等同而论。

3.4.1 ARM 状态

处于 ARM 状态时, ARM920T 运行的都是 32 位的指令, 指令地址是字对齐的, 即指令地址的最低两位二进制位为 0。指令的运行效率非常高。同时也能访问系统中所有的资源和寄存器等。因为 ARM 状态时的指令都是 32 位的长度, 因此也增加了程序运行时的存储空间。在以下几种情况下, ARM920T 处理器会自动进入 ARM 状态。

□ 第一次上电时: 处理器完成自身硬件初始化后, 进入 ARM 状态从 0 地址开始读取第一条指令并开始运行。

□ 响应中断和异常时: 当处理器响应外部硬件中断和处理器异常时, ARM920T 会自动切换到 ARM 状态。

上面描述的两种情况, 都是处理器硬件自动完成的, 不需要程序员的任何操作。不管发生上述两种情况时处理器先前是什么状态, 都会自动切换到 ARM 状态。当然程序员可以手动切换处理器状态, 用处理器的 BX 指令可以做到, 这是个带状态切换的程序跳转指令, 实际上它是根据跳转的地址值的第一个二进制位来判断的, 如果该位为 1 就切换到 Thumb 状态, 如果该位为 0 就切换到 ARM 状态。

3.4.2 Thumb 状态

处于 Thumb 状态时, ARM920T 运行的都是 16 位的指令, 指令地址是半字对齐的, 即指令地址的最低一位二进制位为 0。这些指令都是对 ARM 的一些指令进行重新编码得到的一个 ARM 指令集的子集, 只能访问系统中有限的寄存器和资源。因为 Thumb 状态时的指令都是 16 位的长度, 所以也大大减小了程序运行时的存储空间, 同时这对 16 位的存储系统很有效。Thumb 指令并没有改变 ARM 底层的设计模型, 只是在高层增加了一些限制, 它的数据操作和程序寻址还是 32 位的。ARM920T 处理器不会从任何状态自动切换到 Thumb 状态。这需要程序员手动切换, 用 BX 指令, 可以从 ARM 状态切换到 Thumb 状态, 执行一段 Thumb 指令的程序, 然后从 Thumb 状态返回到 ARM 状态。如果在 Thumb 状态中运行 Thumb 程序时产生中断或者异常, 处理器自动进入 ARM 状态, 处理完中断或者异常, 然后可以从 ARM 状态中返回 Thumb 状态下继续运行。

不必过多关注 Thumb 状态和 Thumb 指令, 因为操作系统代码始终是运行在 ARM 状态下的。并且还要求操作系统下的应用程序也运行在 ARM 状态下, 这样就简单多了。

3.5 ARM920T 处理器的 7 种工作模式

ARM920T 处理器一共有 7 种工作模式，如下：

- 1) 系统管理模式。
- 2) 数据访问终止模式。
- 3) 未定义指令终止模式。
- 4) 外部中断模式。
- 5) 快速中断模式。
- 6) 系统模式。
- 7) 用户模式。

以上这 7 种工作模式除用户模式外，其他 6 种模式都可以称为特权模式。ARM920T 处理器用 CPSR 寄存器中几个二进制位来识别处理器的工作模式，等到介绍寄存器时，再详细讨论它及其中的二进制位。为什么要有这么多工作模式，并且还要分有没有特权？有一点很重要：就是为了保护系统资源，比如，一个行为不端的应用软件去访问硬件，但是由于它运行在用户模式下，并没有访问操作硬件的权力，处理器就能捕获这个错误。

(1) 系统管理模式

系统管理模式，是一种特权模式，在这种模式下软件程序可以自由访问所有硬件资源，还可以自由切换到其他工作模式。当 ARM920T 处理器第一次上电时，就会自动进入这种模式，如果用户模式下软件执行 SWI 指令（即软中断指令），处理器执行完指令后就会切换到系统管理模式。这种工作模式通常用于执行操作系统内核代码。

(2) 数据访问终止模式

当 ARM920T 处理器操作、访问数据时，数据的地址不存在，或者这个地址上的数据不允许被操作或者访问，这时处理器就会自动进入数据访问终止模式。它也是一种特权模式，在这种模式下程序可以自由访问所有硬件资源，也可以自由切换到其他工作模式。这种工作模式通常被操作系统用于虚拟内存管理。比如，对操作系统的内存空间进行保护，不允许应用程序访问系统代码数据的内存空间。

(3) 未定义指令终止模式

当 ARM920T 处理器执行程序时，执行到一条未定义的指令，即它不认识这条指令时，处理器就会自动进入未定义指令终止模式。此工作模式也是一种特权模式，在这种模式下程序可以自由访问所有硬件资源，同时可以自由切换到其他工作模式。这种工作模式通常被系统用于支持软件仿真硬件协处理器。比如，用软件仿真硬件浮点计算单元，假如我们在程序指令数据中放一个处理器不认识的数据充当未定义的指令，处理器就自动切换这个工作模式，最后我们在这种模式下用软件模拟完成浮点计算任务。

(4) 外部中断模式

当 ARM920T 处理器外部中断引脚有电子信号时，并且 ARM920T 处理器这时允许响应

外部中断时，处理器就会自动进入外部中断模式。这种工作模式也是一种特权模式。在这种模式下，程序可以自由访问所有硬件资源，同时可以自由切换到其他工作模式。这种工作模式通常被系统用于普通的外部设备中断，以便和外部设备进行通信。

（5）快速中断模式

当 ARM920T 处理器快速中断引脚有电子信号时，并且 ARM920T 处理器这时允许响应快速中断时，处理器就会自动进入快速中断模式。这种工作模式也是一种特权模式，在这种模式下程序可以自由访问所有硬件资源，同时可以自由切换到其他工作模式。这种工作模式通常被系统用于高速的外部设备中断或者要求实时性响应的设备，以便处理器可以快速、实时地和外部设备进行通信。

（6）系统模式

系统模式除了有特权外，其他都和用户模式一样，但要切换到这种模式下，一定要在其他特权模式下，通过程序手动切换。这种工作模式有特权，所以这种模式下程序可以自由访问所有硬件资源，同时还可以自由切换到其他工作模式。通常用于运行操作系统服务程序，这种程序像普通的应用软件，但是它需要自由访问硬件和系统资源。这种工作模式就刚好派上用场了。

（7）用户模式

用户模式没有特权，不能访问和操作硬件资源，同样也不能自由地切换到其他特权模式，但是若出现下列三种情况，ARM920T 处理器就会自动切换到其他特权模式。

1) 数据访问和未定义指令异常。在用户模式下处理器操作数据和执行程序若遇到这两种情况，处理器会分别切换到数据访问终止模式或者未定义指令终止模式。

2) 外部中断。在用户模式下处理器快速中断和外部中断引脚上有信号，并且处理器允许响应它们。处理器会分别切换到快速中断模式或者外部中断模式。

3) SWI 指令。在用户模式下处理器若执行 SWI 软中断指令，处理器会切换到系统管理模式。

用户模式最大的优点是能有效地保护系统资源，防止恶意软件的不良行为。所以专门用于运行应用软件，使构建现代意义的操作系统成为可能。

3.6 寄存器

处理器处理数据和执行程序都必须要用到寄存器。寄存器是包含在处理器内部的一种高速的暂存器，用于存放访存地址、临时数据和计算结果，大多数情况下其大小和处理器的字长相称，比如 ARM920T 的寄存器都是 4 字节 32 位的，下面就去看看 ARM920T 处理器到底有哪些寄存器。

ARM920T 处理器是 RISC 体系的，这种体系最明显的特点就是有大量的寄存器。ARM920T 处理器一共有多达 37 个寄存器。这 37 个寄存器大致可分为以下几类：

- ❑ 通用寄存器。
- ❑ 栈指针寄存器。
- ❑ 程序链接寄存器。
- ❑ 程序指针寄存器。
- ❑ 程序状态寄存器。

这 37 个寄存器都是 32 位的，虽然有些寄存器只用到其中的几位，如程序状态寄存器，但是它仍然是 32 位的。对程序代码来说，同一时刻最多能看到 18 个寄存器，这取决于 ARM920T 处理器的运行状态和工作模式，如图 3-6 所示。

R0					
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8					R8
R9					R9
R10					R10
R11					R11
R12					R12
R13	R13	R13	R13	R13	R13
R14	R14	R14	R14	R14	R14
R15					
CPSR					
		SPSR	SPSR	SPSR	SPSR
用户模式	系统模式	系统管理模式	数据访问终止模式	未定义指令终止模式	外部中断模式
					快速中断模式

图 3-6 ARM920T 的寄存器

为什么会这样？因为 ARM 公司采用了一种称为“备份寄存器”的技术，不同的工作模式下，虽然使用了相同的寄存器名称，但是操作的却不是同一个物理寄存器。比如，在快速中断模式下向 R13 寄存器中写入“0”值，但这并不会改变其他模式下 R13 寄存器中的值。这些寄存器的分布如下。

- 1) R0 ~ R7、R15、CPSR：这几个寄存器在所有的工作模式下都使用相同的物理寄存器。
- 2) R8 ~ R12：这些寄存器除快速中断模式外，其他工作模式都使用相同的物理寄存器。

3) R13、R14：这两个寄存器除系统模式和用户模式下使用相同的物理寄存器，其他工作模式都使用各自的备份寄存器，即不同的物理寄存器。

4) SPSR：在系统模式和用户模式下没有这个寄存器，其他工作模式都有各自的 SPSR 备份物理寄存器。比如，在快速中断模式下访问 SPSR，访问的是快速中断模式下的 SPSR 物理寄存器，在系统管理模式访问 SPSR，访问的是系统管理模式下的 SPSR 物理寄存器。

下面分别介绍这些寄存器。

(1) R0 ~ R7 寄存器

R0 ~ R7 这 8 个寄存器可以称为“通用寄存器”，也可以称为“低寄存器”。这 8 个寄存器没有用备份寄存器，也就是说 ARM920T 处理器不管运行在哪个工作模式下，对 R0 ~ R7 的操作都是访问同一个物理寄存器。如果 ARM920T 工作在 Thumb 状态下，那么就只能访问这 8 个通用寄存器。

(2) R8 ~ R12 寄存器

R8 ~ R12 这 5 个寄存器也是通用寄存器，同时也可以称为“高寄存器”。这 5 个寄存器除快速中断模式使用了备份寄存器，其他工作模式都使用相同的物理寄存器。也就是说，除快速中断模式外，ARM920T 处理器不管运行在哪个工作模式下，对 R8 ~ R12 的操作都是访问的同一个物理寄存器。如果 ARM920T 处理器工作在 Thumb 状态下，那么就不能访问这 5 个通用寄存器。快速中断模式用于处理高速设备的中断，中断处理过程需要首先保存大量的寄存器，即运行中断处理程序之前的 CPU 上下文。现在 ARM920T 处理器在响应快速中断时，就不需要保存 R8 ~ R12 这 5 个寄存器，因为在快速中断模式下 ARM920T 处理器有 R8 ~ R12 这 5 个备份物理寄存器，在快速中断模式下使用这 5 个寄存器，并不会影响其他模式下的 R8 ~ R12 寄存器，同时也加快了中断处理速度。

(3) R13 寄存器

R13 寄存器，又可以称为 SP 寄存器，这个寄存器除系统模式和用户模式共用相同的物理寄存器外，其他处理器工作模式都有各自的物理备份寄存器。这个寄存器常用作栈指针寄存器。栈是一块内存空间，它存放数据有一定的规则，必须后进先出即最后存放进去的数据要最先取出来，栈指针寄存器的值始终指向最后存放数据的内存地址。如果要用 C 语言写程序就必须要用到栈。因为函数调用和函数中的局部变量都会用到栈。在 ARM920T 处理器上从用户模式切换到系统管理模式，程序的栈也会随着切换，因为用户模式和系统管理模式的 R13 寄存器不是同一个物理寄存器。相应的其他模式也一样，除用户模式和系统模式之间切换不会切换栈，其他模式的切换，程序的栈也会随着切换。另外如果 ARM920T 工作在 Thumb 状态下也能用这个 R13 寄存器作为栈指针寄存器。

(4) R14 寄存器

R14 寄存器，又可以称为 LR 寄存器，这个寄存器除系统模式和用户模式共用相同的物理寄存器外，其他处理器工作模式都有各自的物理备份寄存器。这个寄存器常用作程序链接寄存器。我们的程序中有分支代码和函数调用，这就要使用跳转指令，对函数调用后要返

回, 当我们用带返回的跳转指令时, 这条指令就会把当前指令的下一条指令的地址放进处理器当前模式下的 R14 寄存器中, 函数返回时只要将 R14 寄存器的内容送给程序指针寄存器就行了。如果处理器响应中断、异常, 处理器会自动将当前模式下的程序指针寄存器的内容放在相应中断、异常模式下的 R14 寄存器中, 后面会详细介绍。另外, 如果 ARM920T 工作在 Thumb 状态下, 也能用这个 R14 寄存器作为程序链接寄存器。其作用也和上面说的一样。

(5) R15 寄存器

R15 寄存器, 又可以称为 PC 寄存器, 即程序指针寄存器。这个寄存器在所有的工作模式下都共用相同的物理寄存器, 并且在 Thumb 状态下也是使用这个寄存器作为程序指针寄存器。它总是指向下一条要读取代码指令的地址, 除跳转指令之外, 每读取一条代码指令, 处理器就会自动对这个 R15 寄存器加上一条指令的存储大小, 使其指向下一条要读取代码指令的地址。当处理器处于 ARM 状态时, 该寄存器中的值始终是字对齐的, 而处理器处于 Thumb 状态时该寄存器中的值始终是半字对齐的。这个寄存器还可以用于数据的相对寻址, 后面我们可以看到它神奇的作用。

(6) CPSR 寄存器

这个寄存器有点复杂, 我们慢慢看。CPSR 寄存器, 即当前程序状态寄存器。这个寄存器在所有的工作模式下都共用相同的物理寄存器, 并且在 Thumb 状态下也是使用这个寄存器作为程序的状态寄存器。CPSR 也是 32 位的, 但却没有完全用到 32 位, 也就是说有许多位是保留而留作后用的。它里面保存了如下信息, 如图 3-7 所示。

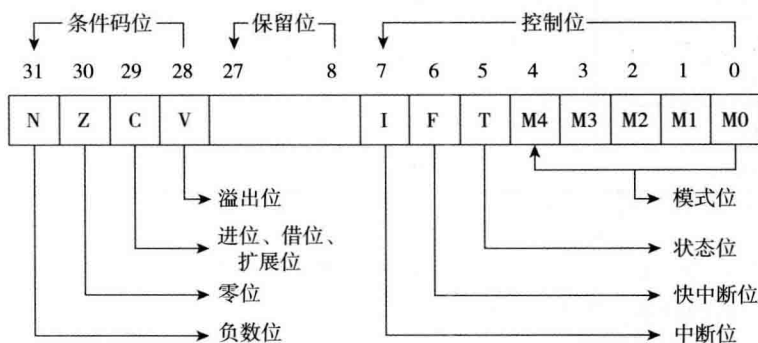


图 3-7 CPSR 寄存器

1) 模式位: 控制处理器的运行模式, 如表 3-1 所示。

表 3-1 CPSR 寄存器模式位

CPSR M[4:0] 位	ARM920T 处理器模式
0b10000	用户模式
0b10001	快速中断模式

(续)

CPSR M[4:0] 位	ARM920T 处理器模式
0b10010	外部中断模式
0b10011	系统管理模式
0b10111	数据访问终止模式
0b11011	未定义指令终止模式
0b11111	系统模式

注：0b 表示它后面的数字是二进制数。

2) CPSR T 位：控制处理器的运行状态，如表 3-2 所示。

表3-2 CPSR寄存器T位

CPSR T 位	ARM920T 处理器状态
0b0	ARM 状态
0b1	Thumb 状态

注：0b 表示它后面的数字是二进制数。

3) CPSR I、F 位：控制中断的使能与禁止，如表 3-3 所示。

表3-3 CPSR寄存器I、F位

CPSR [I、F]	禁止或者允许 ARM920T 外部中断	禁止或者允许 ARM920T 快速中断
0b00	允许	允许
0b01	允许	禁止
0b10	禁止	允许
0b11	禁止	禁止

注：0b 表示它后面的数字是二进制数。

4) CPSR [N、Z、C、V] 位：保存了最近处理器的运算部件执行的信息，即运行了某些数据运算指令后的状态，如表 3-4 所示。

表3-4 CPSR寄存器N、Z、C、V位

CPSR [N、Z、C、V] 位	说 明
N 位	有符号整数运算时，N=1 时表示结果为负数，N=0 时表示结果为正数或者是 0
Z 位	Z=1 时表示运算结果为 0，Z=0 时表示运算结果不为 0
C 位	①在加法指令中，结果产生了进位（即上溢出）时，C=1，其他情况下 C=0 ②在减法指令中，结果产生了借位（即下溢出）时，C=0，其他情况下 C=1 ③在移位指令中，C 中包含了最后一次被移位溢出的位的数值

(续)

CPSR [N、Z、C、V] 位	说 明
V 位	在加、减法指令中, 如果操作数和运算结果为二进制的补码表示的带符号数时, V=1 时表示符号位溢出

5) 保留位。在程序中最好不要使用。这些位可能被 ARM 公司在未来的 ARM 处理器另作他用。

(7) SPSR 寄存器

SPSR 寄存器, 即程序状态备份寄存器。除系统模式和用户模式下没有这个寄存器外, 其他处理器工作模式都有各自的物理备份寄存器, 并且在 Thumb 状态下也是使用这个寄存器作为程序状态备份寄存器。当处理器响应中断、异常时, 处理器会自动将 CPSR 寄存器的内容放在相应中断、异常模式下的 SPSR 寄存器中, 当中断、异常处理返回时, 就将该工作模式下 SPSR 寄存器的内容写入 CPSR 中。这其中有些步骤是 ARM920T 处理器硬件负责完成的, 后面会详细介绍的。

3.7 异常和中断

如果读者开发过应用程序, 或许听说过异常和中断。但是那种异常或者中断, 与这里的异常和中断还是有区别的。本节首先了解什么是异常和中断, 接着分别介绍 ARM920T 处理器上的异常和中断以及 ARM920T 处理器上的相应处理方式, 最后了解什么又是异常中断向量。

3.7.1 什么是异常和中断

处理器执行一条指令, 这条指令能不能运行成功, 要看它的运行条件能不能满足。比如, 一条访问存储器的指令, 如果访问存储器的地址不存在或者不允许访问, 那么这条指令就不能执行成功, 先不管失败后怎么处理, 反正最好的方式是执行另一段程序, 但是当前这段程序肯定要被打断的。另一种情况, 处理器正在运行程序, 这时某个设备急切地需要处理器的关注, 最好的方式也是去执行一段关注这个设备的代码, 这也是要打断当前正在运行的程序。打断当前程序执行流转而执行其他程序的方式示意图, 如图 3-8 所示。我们对打断的方式进行了分类, 如下:

1) **异常**: 同步的, 是由于程序指令本身的问题, 或者不能满足指令的执行条件而程序不能继续往下执行时, 又或者执行到专门产生异常的指令时, 所表现的一种状态。

2) **中断**: 异步的, 是由外部设备发起的, 处理器不知道什么时刻什么设备需要关注, 它只是每执行完一条指令就去检

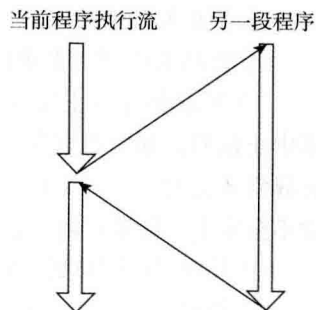


图 3-8 中断、异常的执行过程

测它的中断信号引脚，看看有无中断信号。如果没有就继续执行指令。

下面分别看看 ARM920T 是怎么处理这两种情况的。

1. 异常

异常，是因为不能满足程序指令的执行条件，或者违反了指令的执行规则，又或者执行了用于产生异常的指令。异常又是同步的，如果程序中有一条会产生异常的指令，你不修正这个程序，那么不管重新运行这个程序多少遍，都会在相同的程序地址上发生异常。先来看看 ARM920T 处理中规定了哪些异常。

❑ ARM920T 复位异常。

❑ 未定义指令异常。

❑ 预取指令终止异常。

❑ 数据访问终止异常。

❑ SWI 指令异常。

1) 复位异常：其实这个也不能完全划归为异常，只是它的行为和异常一样，它是当处理器的复位引脚收到一个信号时，就会马上中断当前执行的程序。处理器硬件执行如下操作并且这些操作不可中断：

① 把处理器设为 ARM 状态和系统管理模式。

② 系统管理模式下 R14、SPSR 寄存器中是不确定的数据，因为复位。

③ 禁止快速中断和外部中断。

④ 强制 R15 寄存器中的值为复位向量，这个复位向量后面会有介绍。

2) 未定义指令异常：当 ARM 处理器执行协处理器指令时，它必须等待任何一个外部协处理器应答后，才能真正执行这条指令，此时若没有协处理器回应，则产生未定义指令异常。如果尝试执行未定义的指令，也会产生未定义指令异常。处理器硬件执行如下操作并且这些操作不可中断：

① 处理器被切换到 ARM 状态和未定义指令终止模式。

② 把未定义指令后的下一条指令的地址放进处理器未定义指令终止模式下的 R14 寄存器中，把 CPSR 寄存器的内容放进未定义指令终止模式下的 SPSR 寄存器中。

③ 禁止外部中断。

④ 强制 R15 寄存器中的值为未定义指令终止异常向量。

3) 预取指令终止异常：当指令预取访问存储器失败时，存储器系统向处理器发出存储器中止信号，预取的指令被记为无效，但只有当处理器试图执行该无效指令时，指令预取终止异常才会发生，如果该指令未被执行，如在指令流水线中发生了跳转，则预取指令中止异常不会发生。如果预取终止异常发生，处理器硬件执行如下操作并且这些操作不可中断：

① 处理器被切换到 ARM 状态和数据访问终止模式。

② 把预取终止指令后的下一条指令的地址放进处理器数据访问终止模式下的 R14 寄存器中，把 CPSR 寄存器的内容放进数据访问终止模式下的 SPSR 寄存器中。

③ 禁止外部中断。

④ 强制 R15 寄存器中的值为预取指令终止异常向量。

4) 数据访问终止异常: 如果数据访问指令的目标地址不存在, 或者该地址不允许当前指令访问, 处理器会产生此异常。处理器硬件执行如下操作并且这些操作不可中断:

① 处理器被切换到 ARM 状态和数据访问终止模式。

② 把终止指令后的第二条指令的地址 (这是因为指令流水线), 放进处理器数据访问终止模式下的 R14 寄存器中, 把 CPSR 寄存器的内容放进数据访问终止模式下的 SPSR 寄存器中。

③ 禁止外部中断。

④ 强制 R15 寄存器中的值为数据访问终止异常向量。

5) SWI 指令异常: 其实它也不能称为异常, 只是它的行为和异常一样, 使用指令 SWI 会使处理器模式切换到系统管理模式运行特定的程序。这条指令通常用于作为系统调用的陷入指令。处理器硬件执行如下操作并且这些操作不可中断。

① 把处理器设为 ARM 状态和系统管理模式。

② 把 SWI 指令后的下一条指令的地址放进系统管理模式下的 R14 寄存器中、把 CPSR 寄存器的内容放进系统管理模式下的 SPSR 寄存器中。

③ 禁止外部中断。

④ 强制 R15 寄存器中的值为 SWI 向量。

2. 中断

处理器的速度比外部设备的速度快得多, 然而外部设备又需要处理器的关注, 于是这样高速的处理器就不得不等待低速的外部设备, 这对高速的处理器无疑是巨大的浪费。人们就想了个方法, 设备在需要处理器关注时就给处理器发送个信号, 于是这样处理在没收到信号之前就可以高速地做它自己的事情。处理器在每运行完一条指令后, 首先看有没有这个信号, 如果有这个信号, 还要看系统允不允许响应这个信号, 如果系统允许响应这个信号, 那么处理器就开始处理这个信号。由于处理器不知道什么设备会在什么时刻发送这个信号, 所以这种信号是异步的。这种信号就是中断信号, 由于各种外部设备有轻重缓急, 有的设备需要快速响应, 有的则不那么急。因此 ARM920T 处理器定义两种中断类型: 外部设备中断和快速外部设备中断。

1) 外部设备中断: 通过处理器上的 IRQ 输入引脚, 由外部设备产生 IRQ 中断, 若此时 CPSR 的 I 位为 0, 处理器会产生外部设备中断请求 IRQ 中断。此时, 处理器硬件执行如下操作并且这些操作不可中断。

① 把处理器设为 ARM 状态和外部中断模式。

② 把中断后将要执行的第二条指令的地址 (这也是因为指令流水线的原因), 放进处理器外部中断模式下的 R14 寄存器中, 把 CPSR 寄存器的内容放进外部中断模式下的 SPSR 寄存器中。

- ③ 禁止外部设备中断。
- ④ 强制 R15 寄存器中的值为外部中断向量。

2) 快速外部设备中断：通过处理器上的 FIQ 输入引脚，由外部设备产生 FIQ 中断，若此时 CPSR 的 F 位为 0，处理器会产生快速设备中断请求 FIQ 中断。此时，处理器硬件执行如下操作并且这些操作不可中断。

- ❑ 把处理器设为 ARM 状态和快速中断模式。
- ❑ 把中断后将要执行的第二条指令的地址（这也是因为指令流水线的原因），放进处理器快速中断模式下的 R14 寄存器中，把 CPSR 寄存器的内容放进快速中断模式下的 SPSR 寄存器中。
- ❑ 禁止快速设备中断和外部设备中断。
- ❑ 强制 R15 寄存器中的值为快速中断向量。

3.7.2 异常中断向量

所谓的“异常中断向量”，在 ARM920T 中实际上是一组存储地址，每个地址都是字对齐的，一个地址对应一个字大小的空间，这些字对齐的地址是连续的。根据它们的开始地址分为两种类型：

- ❑ 低地址向量；
- ❑ 高地址向量。

无论低地址向量还是高地址向量，它们的地址不是由软件随意安排的，而是从某个特定的地址开始的，地址如下。

- ❑ 0x00000000 地址：低地址向量，一般情况下 ARM920T 上电时都是从 0 地址开始的低地址向量。
- ❑ 0xFFFF0000 地址：高地址向量，可由软件通过操作 CP15 协处理器中的一个寄存器达到这一目的，后面会详细了解的。

ARM920T 有 7 个向量，外加一个保留向量，一共是 8 个向量。这些向量包含异常和中断，如表 3-5 所示。

表3-5 ARM920T处理器的中断向量、异常向量

低 / 高向量地址	向量名称	ARM920T 进入的状态和模式
0x00000000 / 0xFFFF0000	复位向量	ARM 状态，系统管理模式
0x00000004 / 0xFFFF0004	未定义指令终止向量	ARM 状态，未定义指令终止模式
0x00000008 / 0xFFFF0008	SWI 向量	ARM 状态，系统管理模式
0x0000000c / 0xFFFF000c	预取指令终止向量	ARM 状态，数据访问终止模式
0x00000010 / 0xFFFF0010	数据访问终止向量	ARM 状态，数据访问终止模式
0x00000014 / 0xFFFF0014	保留	保留

(续)

低 / 高向量地址	向 量 名 称	ARM920T 进入的状态和模式
0x00000018 / 0xFFFF0018	外部中断向量	ARM 状态, 外部中断模式
0x0000001c / 0xFFFF001C	快速中断向量	ARM 状态, 快速中断模式

由表 3-5 可知, 这些向量只有一个字的存储空间, 显然这是放不下整个异常或者中断的处理代码的。于是这个地址上就放了一条跳转指令, 就由这条跳转指令跳转到处理程序的地址开始运行。正因为如此, 快速中断向量放在向量表最后, 允许快速中断处理程序直接放在地址 0x0000001C 或 0xFFFF001C 开始的位置, 而不需要由跳转指令跳转到快速中断处理程序。这可以提高快速中断响应的速度。

有些情况下, 会在同一时刻发生多个异常或者中断, 而 ARM920T 处理器在同一时刻又只能处理一个异常或者中断。怎么办呢, ARM920T 处理器采用了一个固定的优先级来决定它们的处理顺序(如下)。数字越小的优先级越高, 优先级最高的最先被 ARM920T 处理。

- ① 复位。
- ② 数据访问终止。
- ③ 快速中断。
- ④ 外部中断。
- ⑤ 预取指令终止。
- ⑥ 未定义指令终止和 SWI。

注意, 并不是所有的异常和中断都可以在同一时刻发生! 比如, 未定义指令终止和 SWI, 因为 SWI 指令不是一条未定义指令, 它是软中断指令。所以在执行 SWI 指令发生软中断异常的時刻, 绝对不可能发生未定义指令终止异常。

正是因为有了异常和中断这种机制, 使得 ARM920T 处理器不仅在高速执行程序时能捕获程序中的问题并且进行相关问题的处理, 还能高效地和外部设备进行通信。同时还能让应用程序通过执行 SWI 指令调用操作系统服务。

3.8 ARM920T 指令集

ARM920T 处理器是精简指令集体系的, 指令不多, 每条指令都是必要的, 指令都是通过精心设计的, 同时指令都是等长的, 即指令占用的存储空间都是相同的。ARM920T 的所有指令都可以分为如下几类。

- ❑ 分支跳转指令。
- ❑ 数据处理指令。
- ❑ 装载和存储指令。

- ❑ 程序状态寄存器操作指令。
- ❑ 协处理器操作指令。
- ❑ 异常中断产生指令。

本节按这样顺序介绍 ARM920T 指令集：概要说明指令及编码格式，然后按上面的分类分别介绍指令，可能有些类型有很多条指令，有些类型只有一条指令。因为指令是程序最基本的组成元素，所以我们必须认真了解它。

3.8.1 指令及其编码格式

首先看 ARM920T 处理器都有哪些指令，如表 3-6 所示。

表3-6 ARM920T处理器的指令集

ARM 汇编程序 指令助记符号	指令相关说明	ARM 汇编程序 指令助记符号	指令相关说明
b	分支跳转	cmp	比较
bl	带链接的分支跳转	cmn	基于相反数比较
bx	带状态切换的分支跳转	teq	相等测试
blx	带链接和状态切换的分支跳转	tst	位测试
mov	数据传送	swp	寄存器与存储器内容交换
mvn	数据求反传送	ldr	加载字数据从存储器到寄存器
add	加法	str	存储字数据从寄存器到存储器
adc	带进位的加法	ldm	加载多个字数据从存储器到多寄存器
sub	减法	stm	存储多个字数据从多个寄存器到存储器
sbc	带借位的减法	mrs	传送程序状态寄存器到通用寄存器
rsb	逆向减法	msr	传送通用寄存器到程序状态寄存器
rsc	带借位的逆向减法	mrc	传送协处理器寄存器到 ARM920T 寄存器
mul	乘法	mcr	传送 ARM920T 寄存器到协处理器寄存器
mla	乘加	ldc	加载存储器字数据到协处理器寄存器
and	逻辑与	stc	存储协处理器寄存器到存储器
orr	逻辑或	cdp	协处理器数据操作指令
bic	位清零	swi	软中断异常
eor	逻辑异或		

35 条指令，当然有很多指令在不同的情况下有很多变形，后面会详细介绍。
虽然我们不是开发编译器的，但是我们还是粗略了解这些指令是怎么编码、怎么

变成二进制数据存储在存储器中的。这对开发操作系统内核是有很多好处的，如图 3-9 所示。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																																
条件码	0	0	1	opcode				S	Rn				Rd				operand 2													数据处理PSR传送		
条件码	0	0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				乘法		
条件码	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				长乘法			
条件码	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				单数据交换			
条件码	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				分支和状态切换				
条件码	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				半字传送寄存器偏移			
条件码	0	0	0	P	U	1	W	L	Rn				Rd				offset				1	S	H	1	offset				半字传送立即数偏移			
条件码	0	1	1	P	U	B	W	L	Rn				Rd				offset													单数据传送		
条件码	0	1	1	xxxxxxxxxxxxxxxxxx																		1	xxxxxxxxxxxxxxxx						未定义			
条件码	1	0	0	P	U	B	W	L	Rn				Register List																	块数据传送		
条件码	1	0	1	L	offset																											分支跳转
条件码	1	1	0	P	U	B	W	L	Rn				CRd				CP#				offset							协处理器数据传送				
条件码	1	1	1	0	CP opc				CRn				CRd				CP#				CP	0	CRm				协处理器数据操作					
条件码	1	1	1	0	CP opc				L	CRn				Rd				CP#				CP	1	CRm				协处理器寄存器传送				
条件码	1	1	1	1	处理器忽略																											软中断

图 3-9 ARM920T 指令编码格式

可以看出，ARM920T 处理器 ARM 状态下的指令都是等长的 32 位编码。每条指令最高 4 位都是一个条件码，还有许多我们不认识的位和符号。这个留到介绍指令时再介绍。这里研究条件码是干什么的。

ARM920T 处理器大部分指令都可以根据上条指令执行结果的状态，决定是否执行当前指令。如果条件满足就执行当前指令，否则不执行当前指令，相当于执行了一条空指令。设计人员为了达到这一目的，才给每条指令加上了一个条件码段，当然条件码是可选的，可以不写条件码的助记符，那么指令就是无条件执行的。一共有 16 种可能的条件，每种条件表示为在指令助记符后面附加两个字符后缀。例如，一条分支跳转指令“b”变成了“beq”，那么即为“如果相等则分支跳转”，这意味着只有 CPSR 寄存器中的 Z 标志位被置位了才会执行分支跳转。在实际应用当中，将会使用到 15 种不同的条件，保留第 16 种（1111）。这些条件码如表 3-7 所示。

表3-7 ARM920T处理器指令的条件码

条 件 码	助 记 符	CPSR 寄存器中的标志位	相 关 说 明
0b0000	EQ	Z 置位	相等
0b0001	NE	Z 清零	不相等
0b0010	CS	C 置位	无符号大于或等于
0b0011	CC	C 清零	无符号小于
0b0100	MI	N 置位	负数
0b0101	PL	N 清零	正数或零
0b0110	VS	V 置位	溢出
0b0111	VC	V 清零	未溢出
0b1000	HI	C 置位并且 Z 清零	无符号大于
0b1001	LS	C 清零或 Z 置位	无符号小于或等于
0b1010	GE	N 等于 V	大于或等于
0b1011	LT	N 不等于 V	小于
0b1100	GT	Z 清零和 (N 等于 V) 逻辑与	大于
0b1101	LE	Z 置位和 (N 不等于 V) 逻辑或	小于或等于
0b1110	AL	忽略	始终

注：0b 表示后面的数字是二进制数。

要注意的是，很多指令不会主动影响 CPSR 寄存器中的标志位。有时可以在指令助记符后面加入“s”符号，表示要求这条指令的执行结果将影响 CPSR 寄存器中的标志位。具体什么样的指令可以加“s”符号，得视具体情况而定。后面介绍每条指令时，会详细介绍。

本书不会过于详细地介绍 ARM920T 处理器所有指令以及每条指令所有的详细使用方式。因为，一是篇幅所限，二是本书不是介绍某一特定处理器的指令集的专著。但是也不用害怕，笔者会尽量做到写操作系统内核时足够使用。

3.8.2 分支跳转指令

如果一个处理器不能跳转执行指令，那么程序的判断、循环结构以及函数调用将不能实现。这样的处理器是不是糟透了，我想肯定是的。庆幸的是，大部分处理器都支持跳转执行指令，ARM920T 也不例外。

ARM920T 支持如下几条分支跳转指令，来实现程序的跳转执行。

- ❑ b。
- ❑ bl。
- ❑ bx。
- ❑ blx。

1. b、bl 指令

指令在汇编程序中的用法：

```
b{1}{cond} <label_offset>|<Rn>
```

- ❑ {} 中的表示可选项。
- ❑ <> 中的表示必需项。
- ❑ | 表示多选一。
- ❑ cond 表示条件码，如 EQ、NE、CS。
- ❑ 1 为链接位，如果在“b”后面写上“1”表示在执行跳转动作的同时，将原来的 PC 值写入当前模式下的 R14 寄存器中。
- ❑ label_offset 表示跳转标号即跳转地址。
- ❑ Rn 表示一个合法的寄存器，一般通用寄存器都行。

分支跳转指令中包含了一个有符号补码的 24 位偏移量。该值左移两位后并将其有符号位扩展到 32 位，因为有效位数为 25 位加 1 位符号位，所以最多能表示 32MB，然后加入 R15 中。因此该指令可以指定 $\pm 32\text{MB}$ 的分支跳转。分支跳转偏移还必须考虑预取操作，因为 PC 超前于当前指令的 2 个字。分支跳转地址如果超过了 $\pm 32\text{MB}$ ，必须将这个地址事先装载到某一通用寄存器中。例如，将这个地址先装入 R0 中，再执行“bl r0”。

例如：

```
main:                ; 标号，汇编器和程序链接器最后会把它转换成地址
    mov r0,#2        ; r0=2
    mov r1,#3        ; r1=3
    bl addfunc1      ; r14=pc; pc=pc+offset(addfunc1)。保存pc至r14并跳转到addfunc1
                    ; 地址上运行。注意offset(addfunc1)是编译工具处理的，表示当前pc和addfunc1
                    ; 的绝对地址之间的差值
lable:              ; 标号
    b lable          ; PC=offset(lable)死循环。注意offset(lable)是编译工具处理的，
                    ; 表示当前pc和lable的绝对地址之间的差值
addfunc1:          ; 标号
    add r0,r0,r1      ; r0=r0+r1
    b r14            ; pc=r14实现子程序返回
```

这段程序非常简单，看代码注释就能明白它是干什么的，但我们主要是为了解 b、bl 指令的使用方式，当然没有写条件码，所以默认情况下指令总是会执行。

2. bx 指令

bx 指令是将一个寄存器中的内容（通常这个内容是一个程序的地址），放进 PC 寄存器中，导致程序的跳转执行并且它还改变处理器的状态：ARM 状态到 Thumb 状态或者 Thumb 状态到 ARM 状态。bx 指令用 Rn 寄存器中的数据和 0xFFFFF0 进行与操作，把结果写到 PC 寄存器中，然后根据 Rn 寄存器第 0 位的值决定处理器的运行状态，即用这个值设置 CPSR 寄存器的 T 位，也就是处理器的状态位。

地址并切换处理器
b 状态之间互相切

3.8.3 数据处理指令

处理器的大部分功能是为了完成数据的处理,所以必然要提供很多数据处理指令,首先来看看 ARM920T 有哪些数据处理指令,如表 3-8 所示。

表3-8 ARM920T处理器的数据处理指令

数据传送	加法运算	减法运算	乘法运算	逻辑运算	比 较	测 试	交 换
mov	add	sub	mul	and	cmp	teq	swp
mvn	adc	sbc	mla	orr	cmn	tst	
		rsb		bic			
		rsc		eor			

1. mov 指令

mov 指令是将一个常数或者寄存器中的数据传送到目标寄存器中。

在汇编中的用法:

```
mov{cond}{s} <Rd>,<operand>
```

- ☐ {} 表示可选项。
- ☐ <> 表示必需项。
- ☐ cond 表示条件码。
- ☐ Rd 表示目标寄存器。
- ☐ operand 表示常数或者是可用的寄存器。
- ☐ s 表示指令执行是否影响 CPSR 中的标志位。还有另外一种情况:如果 Rd 是 PC, s 表示将数据写进 PC 的同时还将当前模式下的 SPSR 寄存器复制到 CPSR 中;如果当前代码运行在用户模式或者系统模式,这两种模式下没有 SPSR,这种方式会导致结果不可预知。

例如:

```
main:
    mov r0,#0          ; r0=0, 把0放进r0中
    movs r1,r0          ; r1=r0, 有s所以根据结果设置CPSR标志位
    moveq r2,#0x40      ; 上步中r1=0, 条件满足所以会执行这条指令, r2=0x40
    movs pc,r2          ; 目标寄存器为pc, 又有s所以CPSR=SPSR, pc=r2, 当然假定这个程序执行除
                        ; 系统和用户模式之外的模式下。最后处理器会跳到0x40地址开始执行
```

2. mvn 指令

mvn 指令是将一个常数或者寄存器中的数据按位取反后传送到目标寄存器中。

在汇编中的用法:

```
mvn{cond}{s} <Rd>,<operand>
```

- ☐ {} 中的表示可选项。

- <> 中的表示必需项。
- cond 表示条件码。
- Rd 表示目标寄存器。
- operand 表示常数或者是可用的寄存器。
- s 表示指令执行是否影响 CPSR 中的标志位，还有另外一种情况，参见 mov 指令。

例如：

```
main:
    mov r0,#0          ; r0=0, 把0放进r0中
    mvn r1,r0          ; r1=r0, 执行后, r1中是0xFFFFFFFF, r0中是0, 因为没有s所以不影响CPSR中的标志位
```

3. add 指令

add 指令完成第一个源寄存器和第二个常数或者第二个源寄存器相加，并把结果放进目标寄存器中，也可以影响 CPSR 中的标志位。

在汇编中的用法：

```
add{cond}{s} <Rd>,<Rn>,<operand>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。
- Rd 表示目标寄存器，和 mov 指令用法一样，也有 PC 的特殊情况。
- Rn 表示第一个源寄存器，只要是通用寄存器即可。
- operand 表示常数或者是可用的寄存器。
- s 和 mov 指令用法情况一样，也有 PC 的特殊情况。

例如：

```
main:
    mov r0,#5          ; r0=5
    mov r1,#5          ; r1=5
    add r0,r0,#5        ; r0=r0+5, 执行后r0中为10
    add r0,r0,r1        ; r0=r0+r1, 执行后r0中为15
```

4. adc 指令

adc 指令完成第一个源寄存器和第二个常数或者第二个源寄存器的相加，再加上 CPSR 中的 C 标志位，并把结果放进目标寄存器中。结果也可以影响 CPSR 中的标志位。

在汇编中的用法：

```
adc{cond}{s} <Rd>,<Rn>,<operand>
```

- {} 中的表示可选项。

- ❑ <> 中的表示必需项。
- ❑ cond 表示条件码。
- ❑ Rd 表示目标寄存器, 和 add 指令相同。
- ❑ Rn 表示第一个源寄存器, 只要是通用寄存器即可。
- ❑ operand 表示常数或者是可用的寄存器。
- ❑ s 和 add 指令情况相同。

例如:

```
main:
    adds r7,r3,r4      ; r7=r3+r4, 有s所以指令的执行会设置CPSR中的标志位
    adc  r8,r5,r6      ; r8=r5+r6+CPSR中C位
                        ; 如果r3、r4中分别放了两个64位数据的低32位, 而r5、r6分别放了这
                        ; 两个64位数据的高32位, 那么这两条指令完成了两个64位数据的加法,
                        ; 结果的低32位放在r7中, 结果的高32位放在r8中
```

5. sub 指令

sub 指令完成第一个源寄存器和第二个常数或者第二个源寄存器相减, 并把结果放进目标寄存器中, 也可以影响 CPSR 中的标志位。需要注意的是减法若发生借位, CPSR 中的 C 位会设置成 0。

在汇编中的用法:

```
sub{cond}{s} <Rd>,<Rn>,<operand>
```

- ❑ {} 中的表示可选项。
- ❑ <> 中的表示必需项。
- ❑ cond 表示条件码。
- ❑ Rd 表示目标寄存器, 和 add 指令情况相同。
- ❑ Rn 表示第一个源寄存器, 只要是通用寄存器即可。
- ❑ operand 表示常数或者是可用的寄存器。
- ❑ s 和 add 指令情况相同。

例如:

```
main:
    mov r0,#10          ; r0=10
    mov r1,#5           ; r1=5
    sub r0,r0,#5        ; r0=r0-5, 执行后r0中为5
    subs r0,r0,r1       ; r0=r0-5, 执行后r0中为0, 有s所以会影响CPSR标志位
    subne r5,r7,#0      ; r5=r7-0, 然而因为上条指令的执行结果导致这条指令的运行条件不满足, 因此这
                        ; 条指令不会执行
```

6. sbc 指令

sbc 指令完成第一个源寄存器和第二个常数或者第二个源寄存器相减, 再减去 CPSR 中的 C 标志位的反码, 并把结果放进目标寄存器中。结果也可以影响 CPSR 中的标志位。

在汇编中的用法：

```
sbc{cond}{s} <Rd>, <Rn>, <operand>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。
- Rd 表示目标寄存器，和 add 指令情况相同。
- Rn 表示第一个源寄存器，只要是通用寄存器即可。
- operand 表示常数或者是可用的寄存器。
- s 和 add 指令情况相同。

例如：

```
main:
subs r7,r3,r4      ; r7=r3-r4, 有s所以指令的执行会设置CPSR中的标志位
sbc r8,r5,r6        ; r8=r5-r6-CPSR中C位反码
                    ; 同adc的思想一样，这两条指令完成了两个64位数的减法
```

7. rsb 指令

rsb 指令完成第二个常数或者第二个源寄存器和第一个源寄存器相减，并把结果放进目标寄存器中。正因为这样，它也称为“逆向减法指令”，也可以影响 CPSR 中的标志位。需要注意的是减法若发生借位，CPSR 中的 C 位会设置成 0。

在汇编中的用法：

```
rsb{cond}{s} <Rd>, <Rn>, <operand>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。
- Rd 表示目标寄存器，和 add 指令情况相同。
- Rn 表示第二个源寄存器，只要是通用寄存器即可。
- operand 表示第一个常数或者是可用的寄存器。
- s 和 add 指令情况相同。

例如：

```
main:
    mov r0,#10      ; r0=10
    mov r1,#5       ; r1=5
    rsb r0,r0,#15    ; r0=15-r0, 执行后r0中为5
    rsbs r0,r0,r1     ; r0=r1-r0, 执行后r0中为0, 有s所以会影响CPSR标志位
    rsbne r5,r7,#0   ; r5=0-r7, 然而因为上条指令的执行结果导致这条指令的运行
                    ; 条件不满足，因此这条指令不会执行
```

8. rsc 指令

rsc 指令完成第二个常数或者第二个源寄存器和第一个源寄存器相减，再减去 CPSR 中的 C 标志位的反码，并把结果放进目标寄存器中。正因为这样，它也称为带借位的逆向减法指令，也可以影响 CPSR 中的标志位。

在汇编中的用法：

```
rsc{cond}{s} <Rd>, <Rn>, <operand>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rd 表示目标寄存器，和 add 指令情况相同。
- ☐ Rn 表示第二个源寄存器，只要是通用寄存器即可。
- ☐ operand 表示第一个常数或者是可用的寄存器。
- ☐ s 和 add 指令情况相同。

例如：

```
main:
    rsbs r4,r0,#0      ; r4=0-r0, 有s所以会影响CPSR标志位
    rsc r5,r7,#0       ; r5=0-r7, 这两条指令完成了求一个64位数据的负数的操作，两个
                        ; 寄存器存放64位数据，两个寄存器存放操作后的64位结果
```

9. mul 指令

mul 指令实现两个 32 位数的乘法运算，把结果放在目标寄存器中。这两个数可以是无符号整数也可以是有符号整数，并且这两个数要放在两个寄存器中。当然结果可能影响 CPSR 中的标志位。由于两个 32 位数的乘积是 64 位数，但是 mul 指令仅仅保存了这个乘积的低 32 位。

在汇编中的用法：

```
mul{cond}{s} <Rd>, <Rn1>, <Rn2>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ s 表示是否根据结果影响 CPSR 中的标志位。
- ☐ Rd 表示目标寄存器，用于存放结果（即乘积）。
- ☐ Rn1 表示存放第一个 32 位数据的寄存器。
- ☐ Rn2 表示存放第二个 32 位数据的寄存器。

例如：

```
main:
```

```

mov r0, #10          ; r0=10
mov r7, #20          ; r7=20
mul r2, r0, r7       ; r2=r0×r7, 执行r2中等于200, 没有s所以不会影响CPSR标志位

```

10. mla 指令

mla 指令实现两个 32 位数的乘法运算, 再加上第三个 32 位数, 把结果放在目标寄存器中。第一、二个数可以是无符号整数也可以是有符号整数, 这三个数据必须放在 3 个寄存器中。当然结果可能影响 CPSR 中的标志位。同样, mla 指令也是只保存前两个数据乘积的低 32 位。

在汇编中的用法:

```
mla{cond}{s} <Rd>, <Rn1>, <Rn2>, <Rn3>
```

- ❑ {} 中的表示可选项。
- ❑ <> 中的表示必需项。
- ❑ cond 表示条件码。
- ❑ s 表示是否根据结果影响 CPSR 中的标志位。
- ❑ Rd 表示目标寄存器, 用于存放结果即前两个数的乘积加上第三个数的结果。
- ❑ Rn1 表示存放第一个 32 位数据的寄存器。
- ❑ Rn2 表示存放第二个 32 位数据的寄存器。
- ❑ Rn3 表示存放第三个 32 位数据的寄存器。

例如:

```

main:
    mov r0, #10          ; r0=10
    mov r7, #20          ; r7=20
    mov r3, #50          ; r3=50
    mlas r2, r0, r7, r3   ; r2=r0×r7+r3, 执行后r2中等于250
                          ; 有s所以会影响CPSR标志位

```

11. and 指令

and 指令完成第一个源寄存器与第二个常数或者第二个源寄存器, 按位进行逻辑与操作, 把结果放进目标寄存器中, 同时根据结果可能影响 CPSR 中的标志位。逻辑与操作是参加与运算的两个数据都为“1”时, 结果才为“1”。例如, 0xF 与 0xF 的结果还是为 0xF; 0xF 与 0 的结果为 0; 0 与 0xF 的结果也为 0; 0x1 与上 0xF 的结果为 0x1。

在汇编中的用法:

```
and{cond}{s} <Rd>, <Rn>, <operand>
```

- ❑ {} 中的表示可选项。
- ❑ <> 中的表示必需项。
- ❑ cond 表示条件码。

□ Rd 表示目标寄存器，和 mov 指令用法情况一样，也有 PC 的特殊情况。

□ Rn 表示第一个源寄存器，只要是通用寄存器即可。

□ operand 表示常数或者是可用的寄存器。

□ s 和 mov 指令用法情况一样，也有 PC 的特殊情况。

例如：

```
main:
    mov r0,#0           ; r0=0
    mov r1,#0xf         ; r1=0xf
    ands r2,r0,#0xf     ; r2=r0 and 0xf, 执行后r2中为0, 有s所以影响CPSR标志位
    andeq r3,r0,r1      ; 由于上条指令的执行, 导致这条指令运行条件满足, r3=r0 and
                        ; r1, 执行后r3中为0
    mov r0,#1           ; r0=1
    and r4,r1,r0        ; r4=r1 AND r0, 执行后r4中为1
```

12. orr 指令

orr 指令完成第一个源寄存器与第二个常数或者第二个源寄存器的按位逻辑或操作，把结果放进目标寄存器中，结果可能影响 CPSR 中的标志位。逻辑或操作是参加或运算的两个数据有一个为“1”时，结果就为“1”。例如，0xF 或 0xF 的结果还是为 0xF；0xF 或 0 的结果为 0xF；0 或 0xF 的结果为 0xF；0x1 或 0xF 的结果为 0xF。

在汇编中的用法：

```
orr{cond}{s} <Rd>,<Rn>,<operand>
```

□ {} 中的表示可选项。

□ <> 中的表示必需项。

□ cond 表示条件码。

□ Rd 表示目标寄存器，和 and 指令用法情况一样。

□ Rn 表示第一个源寄存器，只要是通用寄存器即可。

□ operand 表示常数或者是可用的寄存器。

□ s 和 and 指令用法情况一样。

例如：

```
main:
    mov r0,#0           ; r0=0
    mov r1,#0x1         ; r1=0x1
    orrs r2,r0,#0xf     ; r2=r0 orr 0xf, 执行后r2中为0xf, 有s所以影响CPSR标志位
    orrne r3,r0,r2      ; 上条指令的执行, 导致这条指令运行条件满足, r3=r0 orr
                        ; r2, 执行后r3中为0xf
    mov r0,#1           ; r0=1
    orr r4,r1,r0        ; r4=r1 ORR r0, 执行后r4中为1
```


13. bic 指令

bic 指令完成第一个源寄存器与第二个常数或者第二个源寄存器的反码操作，把结果放进目标寄存器中，同时结果可能影响 CPSR 中的标志位。这条指令经常用于数据位清零操作。

在汇编中的用法：

```
bic{cond}{s} <Rd>, <Rn>, <operand>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rd 表示目标寄存器，和 and 指令用法情况一样。
- ☐ Rn 表示第一个源寄存器，只要是通用寄存器即可。
- ☐ operand 表示常数或者是可用的寄存器。
- ☐ s 和 and 指令用法情况一样。

例如：

```
main:
    mov r0, #0xf          ; r0=0xf
    mov r1, #0xf          ; r1=0xf
    mov r4, #0            ; r4=0
    bics r2, r0, #0xf      ; r2=r0 bic 0xf, 执行后r2中为0, 有s所以影响CPSR标志位
    biceq r3, r0, r4       ; 上条指令的执行, 导致这条指令运行条件满足, r3=r0 bic
                          ; r4, 执行后r3中为0xf
```

14. eor 指令

eor 指令完成第一个源寄存器与第二个常数或者第二个源寄存器的按位逻辑异或操作，把结果放进目标寄存器中，同时结果可能影响 CPSR 中的标志位。逻辑异或操作是参加异或运算的两个数据位相同时，结果就为“0”，不同时为“1”。例如，0xF 异或 0xF 的结果为 0；0xF 异或 0 的结果为 0xF；0 异或 0xF 的结果为 0xF；0x0 异或 0x0 的结果为 0x0。

在汇编中的用法：

```
eor{cond}{s} <Rd>, <Rn>, <operand>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rd 表示目标寄存器，和 and 指令用法情况一样。
- ☐ Rn 表示第一个源寄存器，只要是通用寄存器即可。
- ☐ operand 表示常数或者是可用的寄存器。
- ☐ s 和 and 指令用法情况一样。

例如:

```
main:
    mov r0,#0           ; r0=0
    mov r1,#0x1         ; r1=0x1
    eors r2,r0,#0xf      ; r2=r0 eor 0xf, 执行后r2中为0xf, 有s所以影响CPSR标志位
    orrne r3,r1,r2       ; 上条指令的执行, 导致这条指令运行条件满足, r3=r0 orr r2,
                        ; 执行后r3中为0xe
```

15. cmp 指令

cmp 指令用第一个源寄存器减去第二个常数或者第二个源寄存器, 总是根据结果影响 CPSR 中的标志位, 但是不回写结果。

在汇编中的用法:

```
cmp{cond} <Rn>,<operand>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rn 表示第一个源寄存器, 只要是通用寄存器即可。
- ☐ operand 表示常数或者可用的寄存器。

例如:

```
main:
    mov r0,#0           ; r0=0
    mov r1,#0x1         ; r1=0x1
    mov r2,#0x1         ; r2=0x1
    cmp r0,r1           ; r0-r1, 执行后r0、r1中数据不变, 会影响CPSR标志位, 会根据
                        ; 结果是否为0、正数、负数, 决定设置CPSR哪些标志位
    cmpne r1,#1          ; 上条指令的执行, 导致这条指令的运行条件满足,
                        ; 会根据结果是否为0、正数、负数, 决定设置CPSR哪些标志位
```

16. cmn 指令

cmn 指令用第一个源寄存器加上第二个常数或者第二个源寄存器, 总是根据结果影响 CPSR 中的标志位, 但是不回写结果。

在汇编中的用法:

```
cmn{cond} <Rn>,<operand>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rn 表示第一个源寄存器, 只要是通用寄存器即可。
- ☐ operand 表示常数或者可用的寄存器。

例如：

```
main:
    cmp r0,#0          ; r0-0, CPSR中C标志位等于0
    cmn r0,#0          ; r0+0, CPSR中C标志位等于1
```

17. tst 指令

tst 指令完成第一个源寄存器与第二个常数或者第二个源寄存器按位逻辑与操作。总是根据结果影响 CPSR 中的标志位，C 位、V 位通常是不受影响的。同样也不回写结果。

在汇编中的用法：

```
tst{cond} <Rn>,<operand>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。
- Rn 表示第一个源寄存器，只要是通用寄存器即可。
- operand 表示常数或者是可用的寄存器。

例如：

```
main:
    mov r0,#0          ; r0=0
    mov r1,#0x1        ; r1=0x1
    tst r0,#0          ; r0 AND 0, 执行后, CPSR中的N标志位=0、Z标志位=1
    mov r2,#1          ; r2=1
    tst r1,r2          ; r1 AND r0, 执行后, CPSR中的N标志位=0、Z标志位=0
```

18. teq 指令

teq 指令完成第一个源寄存器与第二个常数或者第二个源寄存器按位逻辑异或操作。总是根据结果影响 CPSR 中的标志位，C 位、V 位通常是不受影响的。同样也不回写结果。

在汇编中的用法：

```
teq{cond} <Rn>,<operand>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。
- Rn 表示第一个源寄存器，只要是通用寄存器即可。
- operand 表示常数或者可用的寄存器。

例如：

```
main:
    mov r0,#0          ; r0=0
    mov r1,#0x1        ; r1=0x1
```

```

teq r0,#0      ; r0 EOR 0执行后, CPSR中N标志位=0、Z标志位=1
mov r2,#1      ; r2=1
teq r1,r2      ; r1 EOR r2执行后, CPSR中N标志位=0、Z标志位=1

```

19. swp 指令

swp 指令完成从一个内存字单元中读取一个数据到目标寄存器, 这个内存字的地址放在第二个源寄存器中, 以此同时把第一个源寄存器中的内容写入这个内存字单元中。因此这条指令完成内存字和寄存器之间数据的原子交换。还有一条指令, 其功能、用法都和 swp 指令差不多的指令——swpb 指令。它唯一和 swp 指令不同的是: 它只是交换一个内存字节单元的数据。执行后, swpb 自动把目标寄存器的高 24 位清零, 只保留低 8 位, 即一个字节。

在汇编中的用法:

```
swp{cond} <Rd>, <Rn1>, [<Rn2>]
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。
- ☐ Rd 表示目标寄存器。
- ☐ Rn1 表示第一个源寄存器。
- ☐ Rn2 第二个源寄存器, 存放内存字单元的地址。
- ☐ [] 表示用它里面的寄存器的内容作为访问内存的地址去访问内存。

例如:

```

main:
    mov r0,#0      ; r0=0
    mov r1,#0x40   ; r1=0x40
    swp r0,r0,[r1] ; 执行后, r0等于内存地址0x40开始的一个32位数据, 内存地址0x40
                    ; 开始的一个字单元中的数据等于0, 并且这两个动作是原子的, 即在
                    ; 不可分割的状态下执行的

```

3.8.4 装载和存储指令

ARM920T 处理器是 RISC 体系的, 它有大量的寄存器, 并且大部分数据的操作都是在寄存器中完成的。这点我们从前面的诸多指令可以得到证明。这么有个问题, 这么多程序和数据, 不太可能通过某种机制一次性全部放进寄存器中, 何况对于庞大的程序和数据, 寄存器实在少得可怜。所以大部分程序和数据还是要放在内存里的。那么就需要有一种指令用于装载内存数据到寄存器中和把寄存器中的数据存储到内存中。于是 ARM 设计人员设计两种类型的装载、存储指令。一种用于装载、存储单个数据, 一种用于装载、存储数据块。指令如下。

- ☐ ldr, 装载单个数据指令。
- ☐ str, 存储单个数据指令。

- ❑ ldm, 装载数据块指令。
- ❑ stm, 存储数据块指令。

先把 ldr、str 指令放在一起介绍, 然后把 ldm、stm 指令放在一起介绍。这里要仔细一点, 这几条指令是最常用的, 也是在对它们的理解上最容易出问题的。

1. ldr、str 指令

ldr 指令完成从指定地址的内存单元中装载数据到目标寄存器中, 而 str 指令将一个寄存器中的数据存储到指定地址的内存单元中。然而指定地址的形式很烦琐, 装载或者存储数据的大小也有不同。所以下面要仔细分析。

在汇编程序中这两条指令的用法:

```
<ins>{<cond>}{h|b} Rd,<addr_mode>
```

- ❑ {} 中的表示可选项。
- ❑ <> 中的表示必需项。
- ❑ | 表示多选一。
- ❑ ins 表示指令助记符。
 - ldr 表示从 <addr_mode> 表示的地址的内存单元中装载数据到 Rd 表示的目标寄存器中。
 - str 表示把 Rd 表示的目标寄存器中的数据存储到 <addr_mode> 表示的地址的内存单元中。
- ❑ cond 表示条件码。条件满足的情况下才会执行这条指令。
- ❑ h|b 表示装载、存储数据的宽度。
 - h 表示装载、存储半字数据 (即 16 位), 并且 ldr 自动清除 Rd 寄存器的高 16 位。
 - b 表示装载、存储字节数据 (即 8 位), 并且 ldr 自动清除 Rd 寄存器的高 24 位。
- ❑ Rd 表示目标寄存器。
- ❑ addr_mode 表示内存地址。它有如下几种情形。
 - [Rn] 表示用 Rn 寄存器中的值, 作为装载、存储内存单元的地址。
 - [Rn, +/-Rm] 表示用 Rn 寄存器的值加上或者减去 Rm 寄存器的值的结果, 作为装载、存储内存单元的地址。
 - [Rn, +/-Rm]! 表示用 Rn 寄存器的值加上或者减去 Rm 寄存器的值的结果, 作为装载、存储内存单元的地址。然后把 Rn 寄存器的值加上或者减去 Rm 寄存器的值的结果回写到 Rn 寄存器中。
 - [Rn], +/-Rm 表示用 Rn 寄存器的值, 作为装载、存储内存单元的地址。然后把 Rn 寄存器的值加上或者减去 Rm 寄存器的值的结果回写到 Rn 寄存器中。
 - [Rn, +/-#offset_8] 表示用 Rn 寄存器的值加上或者减去 8 位偏移量的结果, 作为装载、存储内存单元的地址。

- `[Rn, +/- #offset_8]!` 表示用 `Rn` 寄存器的值加上或者减去 8 位偏移量的结果, 作为装载、存储内存单元的地址。然后把用 `Rn` 寄存器的值加上或者减去 8 位偏移量的值的结果回写到 `Rn` 寄存器中。
- `[Rn], +/- #offset_8` 表示用 `Rn` 寄存器的值作为装载、存储内存单元的地址。然后把用 `Rn` 寄存器的值加上或者减去 8 位偏移量的值的结果回写到 `Rn` 寄存器中。

例如:

```
main:
    mov r0, #0x40          ; r0=0x40
    ldr r1, [r0]            ; 把地址0x40的内存字单元装载到r1寄存器中
    ldrh r2, [r0, #4]       ; 把地址0x44 (r0+4) 的内存半字单元装载到r2寄存器中并且其
                            ; 高16位清零
    ldrb r3, [r0, #-#8]     ; 把地址0x38 (r0-8) 的内核字节单元装载到r3寄存器中并且其
                            ; 高24位清零
    str r1, [r0]            ; 把r1寄存器的数据存储到地址0x40的内存字单元中
    strh r2, [r0, #4]       ; 把r2寄存器的低16位数据存储到地址0x44 (r0+4) 的内存半字单元中
    strb r3, [r0, -#8]      ; 把r3寄存器的低8位数据存储到地址0x38 (r0-8) 的内存字节单元中
    mov r0, #0x20          ; r0=0x20
    mov r1, #0x10          ; r1=0x10
    ldr r2, [r0, r1]        ; 把地址0x30 (r0+r1) 的内存字单元装载到r2寄存器中
    ldrh r3, [r0], r1       ; 把地址0x20的内存字半单元装载到r3寄存器中。然后r0=r0+r1
                            ; 所以r0=0x30
    ldrb r4, [r0, r1]!      ; 把地址0x40 (r0+r1) 的内存字节单元装载到r4寄存器中。然后
                            ; r0=r0+r1, 所以r0=0x40
    strb r4, [r0, ]-r1      ; 把r4寄存器低8位数据存储到地址0x40的内存字节单元中。
                            ; 然后r0=r0-r1, 所以r0=0x30
    strh r3, [r0, #8]!      ; 把r3寄存器低16位的数据存储到地址0x38 (r0+8) 的内存半字
                            ; 单元中。然后r0=r0+8, 所以r0=0x38
    str r2, [r0], #0        ; 把r2寄存器的数据存储到地址0x38的内存字单元中。然后
                            ; r0=r0+0, 所以r0=0x38
```

2. ldm、stm 指令

`ldm` 指令从一个内存地址开始的连续内存单元中, 装载多个字到多个寄存器中。`stm` 指令把多个寄存器存放到一个内存地址开始的连续内存空间中。

在汇编程序中这两条指令的用法:

```
<ins>{<cond>}<addr_mode><Rn>{!}, <{><regs_list><}>{^}
```

- ☐ `{}` 中的表示可选项。
- ☐ `<>` 中的表示必需项。
- ☐ `ins` 表示指令。

- `ldm` 表示从 `Rn` 里的地址开始装载多个内存字到一组寄存器中, 低地址的内存字装载进低寄存器 (即编号小的寄存器) 中, 高地址的内存字装载进高寄存器 (即编号大的寄存器) 中。

- `stm` 表示从 `Rn` 里的地址开始存储一组寄存器到多个内存字单元中，小编号寄存器存储在低地址的内存字单元中，大编号寄存器存储在高地址的内存字单元中。
- ❑ `cond` 表示条件码，条件满足时才执行指令。
- ❑ `addr_mode` 表示内存地址形势，有如下几种。
 - `ia` 表示事后递增方式。每次传送数据后，地址加上一个内存字单元大小。
 - `ib` 表示事先递增方式。每次传送数据前，地址加上一个内存字单元大小。
 - `da` 表示事后递减方式。每次传送数据后，地址减去一个内存字单元大小。
 - `db` 表示事先递减方式。每次传送数据前，地址减去一个内存字单元大小。
 - `fd` 表示满递减的栈操作。
 - `ed` 表示空递减的栈操作。
 - `fa` 表示满递增的栈操作。
 - `ea` 表示空递增的栈操作。
- ❑ `Rn` 表示用于存放内存空间的开始地址的寄存器。
- ❑ `!` 表示这条指令执行后，是否将已经变化的地址回写到 `Rn` 寄存器中。回写的地址是，用 `Rn` 里的值加上或者减去 `regs_list` 里的寄存器个数乘以 4 计算的。加上或者减去取决于 `addr_mode` 的设置。
- ❑ `regs_list` 表示寄存器列表，可以是 `R0 ~ R15` 这 16 个寄存器，可以写成：“`r0-r15`”、“`r0-r1,r3-r4,r7-r15`”、“`r0,r2,r3,r10,r14`”等。当然不是一定要包含所有寄存器，可以是一个寄存器，那么它就只读取或者存储一个内存字单元，如果是两个寄存器，它就读取或者存储两个内存字单元。
- ❑ `^` 根据 `regs_list` 里的不同寄存器，设置这个“`^`”符号有如下两种情况。
 - `ldm` 指令的 `regs_list` 里面有 `PC` 即 `R15` 寄存器时，表示在装载 `regs_list` 里面的寄存器的同时将当前模式下的 `SPSR` 装载到 `CPSR` 中。当然这个当前模式不能是系统模式和用户模式。
 - `ldm`、`stm` 指令的 `regs_list` 里面没有 `R15` 寄存器时，表示当处理器工作在非用户模式下时，装载和存储的 `regs_list` 里的寄存器是用户模式下相应的寄存器。

例如：

```
main:
    mov r0,#1           ; r0=1
    mov r1,#2           ; r1=2
    mov r2,#3           ; r2=3
    mov r3,#4           ; r3=4
    mov r4,#5           ; r4=5
    mov r5,#6           ; r5=6
    mov r6,#7           ; r6=7
    mov,r7,#8           ; r7=8
    mov,r8,#9           ; r8=9
```

```

mov r9, #10           ; r9=10
mov r10, #11          ; r10=11
mov r11, #12          ; r11=12
mov r12, #13          ; r12=13
mov r13, #0x40        ; r13=0x40
mov r14, #0x80        ; r14=0x80

stmia r13, {r0-12, r14}
stmib r13, {r0-12, r14}
stmda r13, {r0-12, r14}
stmdb r13, {r0-12, r14}
ldmia r13, {r0-12, r14}
ldmib r13, {r0-12, r14}
ldmda r13, {r0-12, r14}

ldmdb r13, {r0-12, r14}
stmfd r13!, {r0-r12, r14}
ldmfd r13!, {r0-r12, r14}
stmfd r13!, {r0-r12, r14}^
ldmfd r13!, {r0-r12, pc}^
stmfd r13!, {r0, r5, r7, r14}
ldmfd r13!, {r0, r5, r7, r14}^

```

图 3-10 为 `stmia`、`stmib` 指令的执行过程。`stmia r13, {r0-12, r14}` 把 R0 ~ R12 和 R14 这 14 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 执行过程如图 3-10 (a) 所示; `stmib r13, {r0-12, r14}` 把 R0 ~ R12 和 R14 这 14 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 执行过程如图 3-10 (b) 所示。

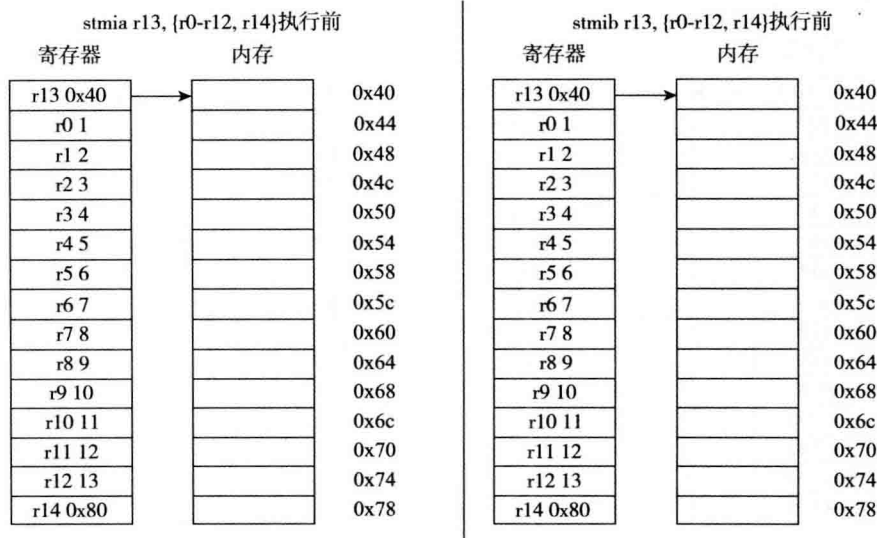


图 3-10 `stmia`、`stmib` 指令的执行过程

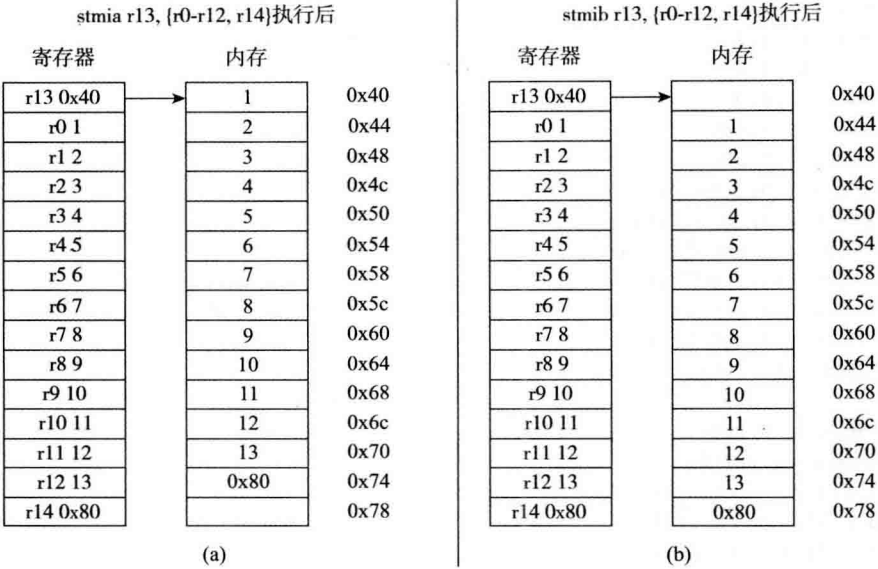


图 3-10 (续)

图 3-11 为 stmda、stmdb 指令的执行过程。stmda r13,{r0-12,r14} 把 R0 ~ R12 和 R14 这 14 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 执行过程如图 3-11 (a) 所示; stmdb r13,{r0-12,r14} 把 R0 ~ R12 和 r14 这 14 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 执行过程如图 3-11 (b) 所示。

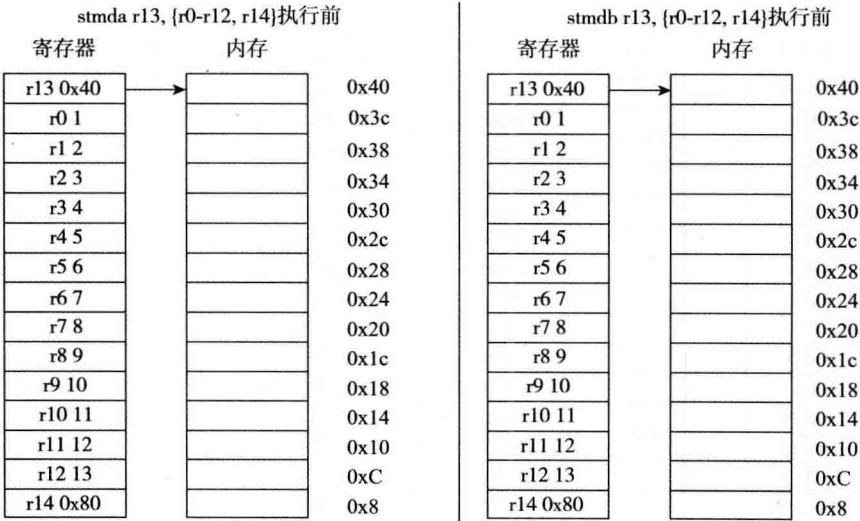


图 3-11 stmda、stmdb 指令的执行过程

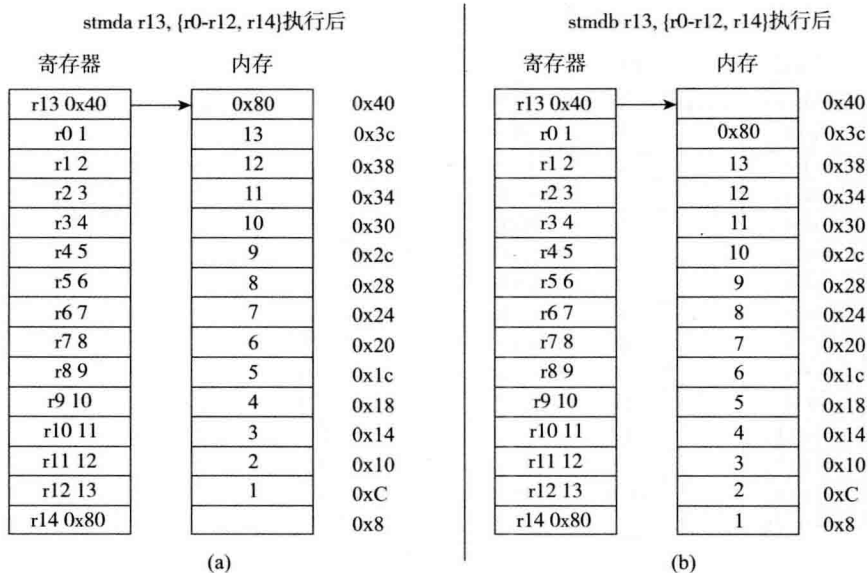


图 3-11 (续)

图 3-12 为 ldmia、ldmib 指令的执行过程。ldmia r13,{r0-12,r14} 从以 R13 寄存器值为开始地址的内存空间, 装载 14 个内存字到 R0 ~ R12 和 R14 这 14 个寄存器中, 执行过程如图 3-12 (a) 所示; ldmib r13,{r0-12,r14} 从以 R13 寄存器值为开始地址的内存空间, 装载 14 个内存字到 R0 ~ R12 和 R14 这 14 个寄存器中, 执行过程如图 3-12 (b) 所示。

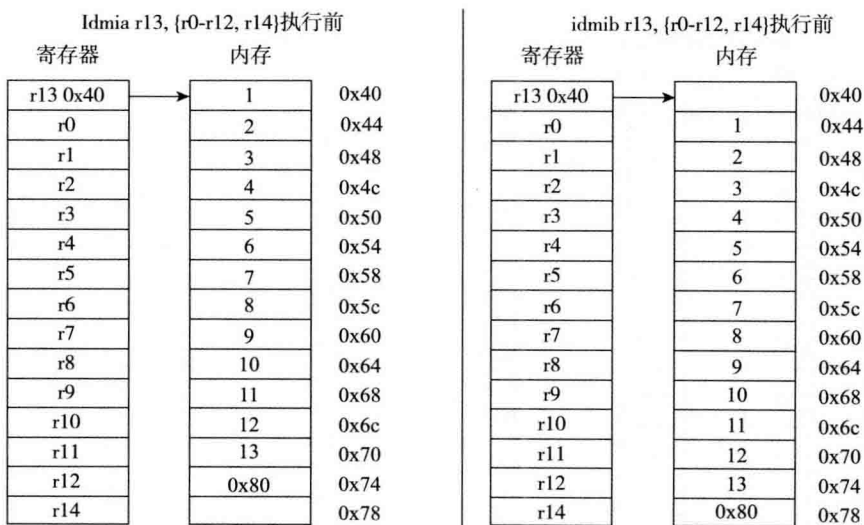


图 3-12 ldmia、ldmib 指令的执行过程

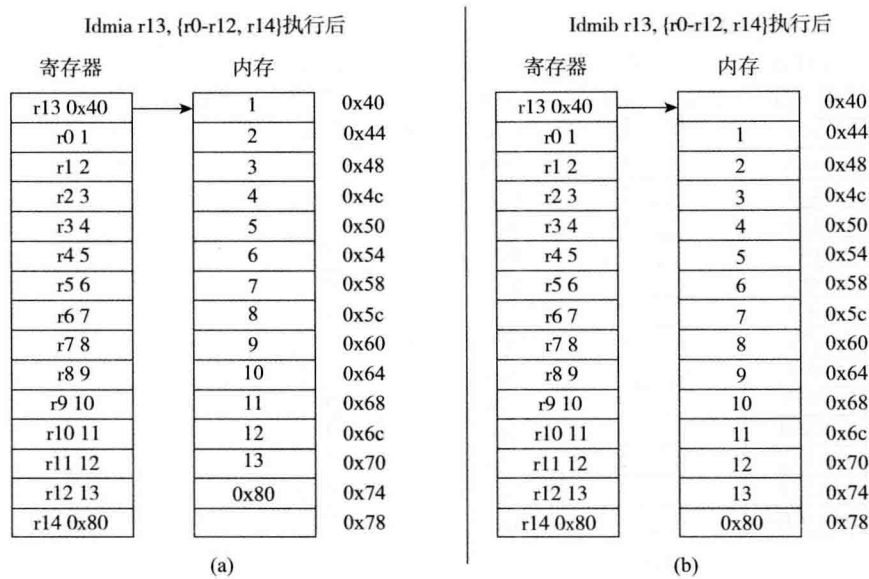


图 3-12 (续)

图 3-13 为 ldmda、ldmdb 指令的执行过程。ldmda r13,{r0-12,r14} 从以 R13 寄存器值为开始地址的内存空间，装载 14 个内存字到 R0 ~ R12 和 R14 这 14 个寄存器中，执行过程如图 3-13 (a) 所示；ldmdb r13,{r0-12,r14} 从以 R13 寄存器值为开始地址的内存空间，装载 14 个内存字到 R0 ~ R12 和 R14 这 14 个寄存器中，执行过程如图 3-13 (b) 所示。

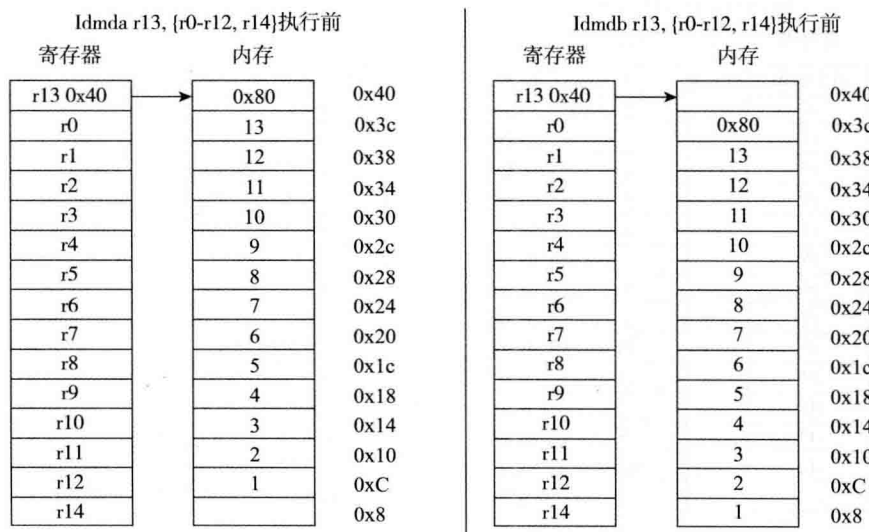


图 3-13 ldmda、ldmdb 指令的执行过程

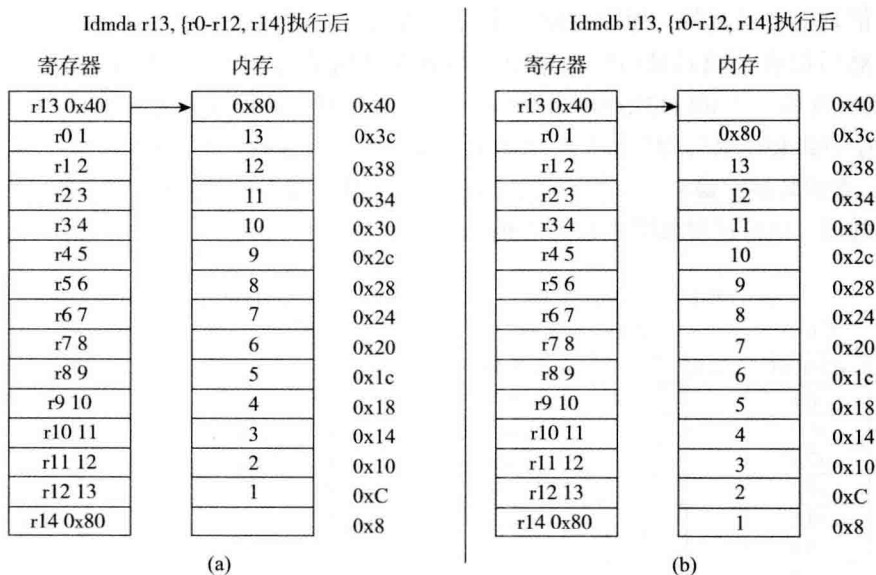


图 3-13 (续)

图 3-14 为 `stmfd`、`ldmfd` 指令的执行过程。`stmfd r13!,{r0-r12,r14}` 把 R0 ~ R12 和 R14 这 14 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 注意这条指令 R13 里的地址值是向下递减的, 每次存储一个寄存器之前 R13 就先减 4, 直到存储完最后一个寄存器。然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值, 执行过程如图 3-14 (a) 所示。`ldmfd r13!,{r0-r12,r14}` 从以 R13 寄存器值为开始地址的内存空间, 装载 14 个内存字到 R0 ~ R12 和 R14 这 14 个寄存器中, 每装载一个内存字之后, R13 寄存器的值加 4, 直到装载完最后一个寄存器, 然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值, 执行过程如图 3-14 (b) 所示。

图 3-15 为 `stmfd`、`ldmfd` 指令的执行过程。`stmfd r13!,{r0-r12,r14}^` 把 R0 ~ R12 和 R14 这 14 个用户模式的寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 注意这条指令中 R13 里的地址值是向下递减的, 每次存储一个寄存器之前 R13 就先减 4, 直到存储完最后一个寄存器。然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值, 执行过程如图 3-15 (a) 所示。`ldmfd r13!,{r0-r12,pc}^` 从以 R13 寄存器值为开始地址的内存空间, 装载 14 个内存字到 R0 ~ R12 和 PC 这 14 个寄存器中, 每装载一个内存字之后, R13 寄存器的值加 4, 直到装载完最后一个寄存器, 然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值。注意这有 PC 寄存器且有 “^” 符号, 所以该指令最后还会把某个特定模式下的 SPSR 复制到 CPSR 中。执行过程如图 3-15 (b) 所示。

图 3-16 为 `stmfd`、`ldmfd` 指令的执行过程。`stmfd r13!,{r0,r5,r7,r14}` 把 R0、R5、R7 和 R14 这 4 个寄存器存储在以 R13 寄存器值为开始地址的内存空间中, 注意这条指令中 R13

里的地址值是向下递减的，每次存储一个寄存器之前 R13 就先减 4，直到存储完最后一个寄存器。然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值，执行过程如图 3-16 (a) 所示。ldmfd r13!,{r0,r5,r7,r14}^ 从以 R13 寄存器值为开始地址的内存空间，装载 4 个内存字到 R0、R5、R7 和 R14 这 4 个寄存器中，每装载一个内存字之后，R13 寄存器的值加 4，直到装载完最后一个寄存器，然后把这个减到最后的地址值回写到 R13 寄存器中改变 R13 的值，执行过程如图 3-16 (b) 所示。

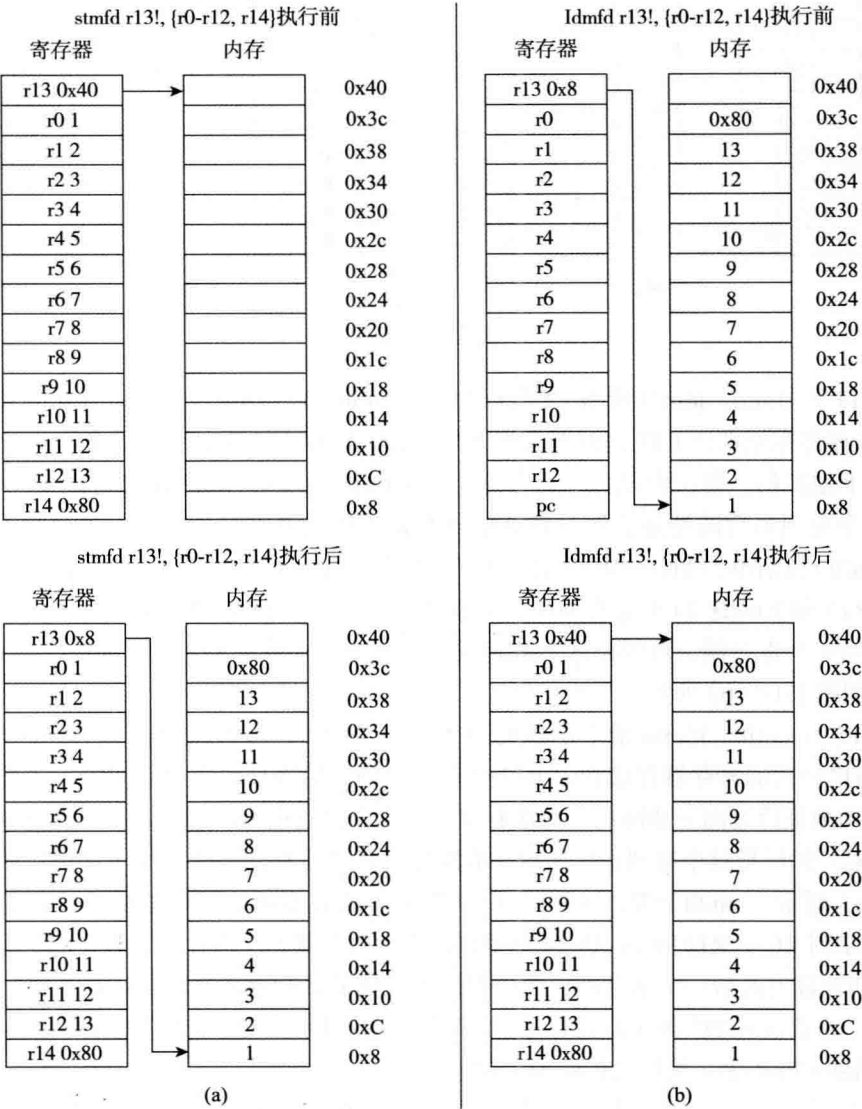


图 3-14 stmfd、ldmfd 指令的执行过程 (R13 寄存器后带 !)

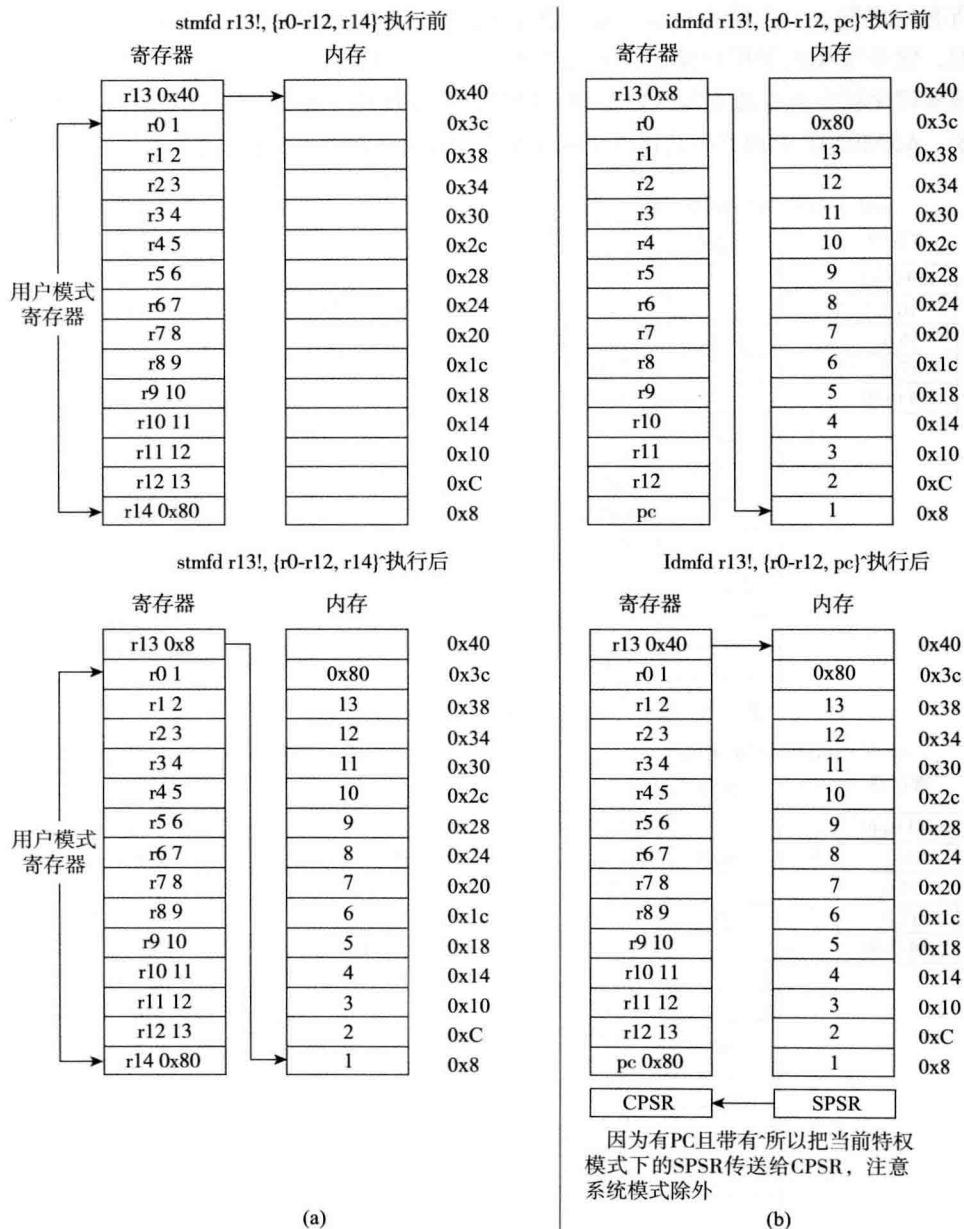


图 3-15 stmfd、ldmfd 指令的执行过程 (R13 寄存器后带 ! 且带有 ^ 和包含 PC 的情况)

3.8.5 程序状态寄存器操作指令

ARM920T 处理器有一个 CPSR 寄存器和 5 个 SPSR 寄存器。CPSR 是当前程序状态寄存

器，所有处理器 b 工作模式和状态 d 下都使用同一个 CPSR 寄存器。SPSR 是程序备份状态寄存器，除系统模式和用户模式没有这个寄存器外，其他 5 种模式都各自有一个 SPSR。然而向这些程序状态寄存器中写入数据和读取数据都要使用专用的指令，不能简单地使用 mov 等指令。ARM920T 处理器一共提供了两条指令来操作 CPSR、SPSR 寄存器。

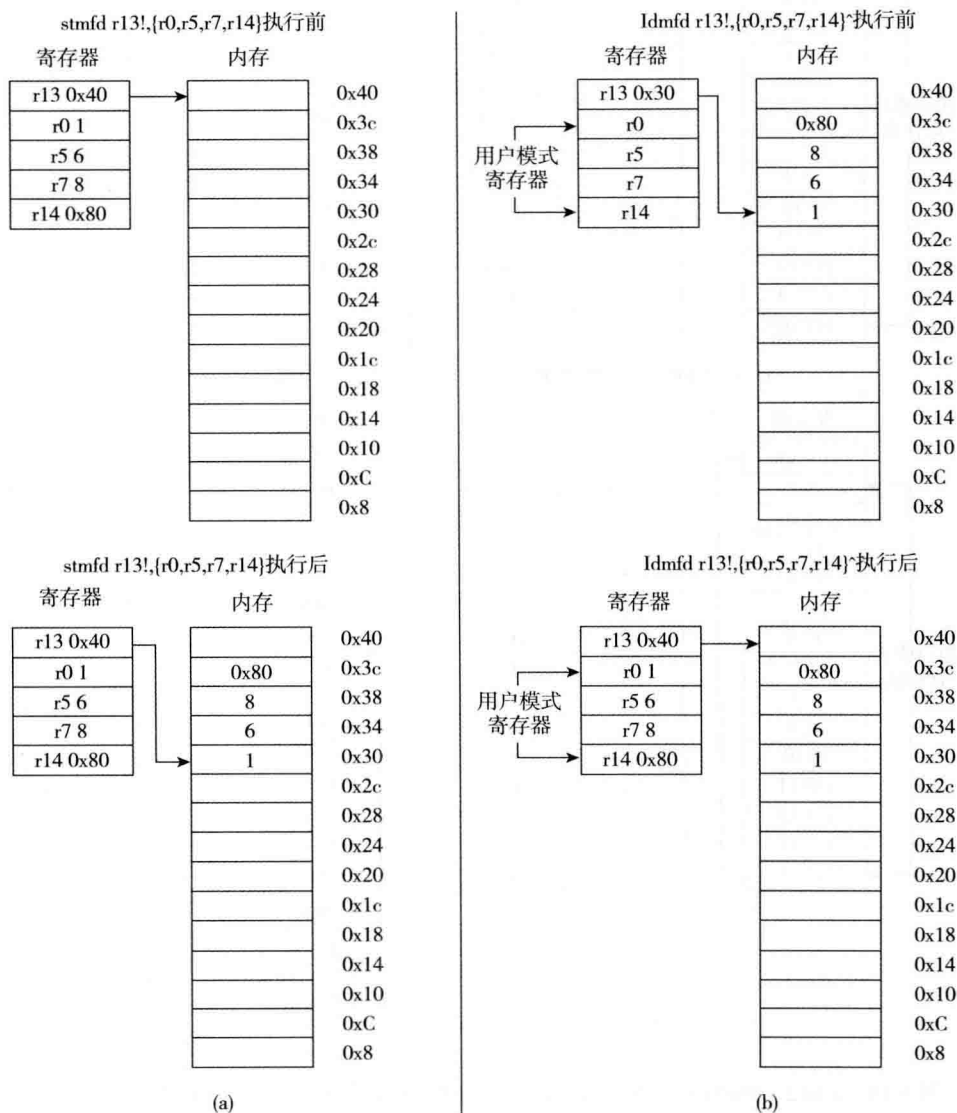


图 3-16 stmfd、ldmfd 指令的执行过程 (R13 寄存器后带 !) 且 (b) 中的指令最后带有 “^”

- ❑ mrs 指令，把程序状态寄存器中的数据读取到通用寄存器中。
- ❑ msr 指令，把通用寄存器中的数据写入程序状态寄存器中。

1. mrs 指令

mrs 指令将 CPSR 或者 SPSR 寄存器中的数据读取到通用寄存器中，例如，读取到 R0 ~ R12 等各个寄存器中。

在汇编中的用法：

```
mrs{cond} <Rd>,<SR>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。条件满足时执行，忽略条件码时表示无条件执行。
- Rd 表示目标寄存器，只要是通用寄存器即可，如 R0、R1、R2、R6 等。
- SR 表示程序状态寄存器。有如下两种选择。
 - CPSR，表示将 CPSR 寄存器中的数据读取到 Rd 表示的目标寄存器中。
 - SPSR，表示将特定模式下的 SPSR 寄存器中的数据读取到 Rd 表示的目标寄存器中。

例如：

```
main:
    mov r8,#0xd3          ; r8=0xd3
    mov r9,#0             ; r9=0
    mrs r9,cpsr           ; r9=cpsr。把CPSR读取到r9
    mrs r7,spsr           ; r7=spsr。把一个特定模式的SPSR读取到r7，这个模式可
                        ; 能是系统管理模式也可能是IRQ模式等
```

2. msr 指令

msr 指令将一个立即数或者一个通用寄存器中的内容，写入 CPSR 寄存器或者某个特定模式下的 SPSR 寄存器中。

在汇编中的用法：

```
msr{cond} <SR>{<_><fds>},<Rn|immed>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- | 表示多选一。
- cond 表示条件码。条件满足时执行，忽略条件码时表示无条件执行。
- SR 表示程序状态寄存器，有如下两种选择。
 - CPSR，表示将立即数或者通用寄存器中的值写入 CPSR 寄存器中。
 - SPSR，表示将立即数或者通用寄存器中的值写入 SPSR 寄存器中。
- _ 表示连接符。
- fds 表示 4 个不同的位段，有如下 4 种选择。执行指令时表示将改写哪个位段。
 - f，表示改写状态寄存器中的 24 ~ 31 位。

- s, 表示改写状态寄存器中的 16 ~ 23 位。
- x, 表示改写状态寄存器中的 8 ~ 15 位。
- c, 表示改写状态寄存器中的 0 ~ 7 位。

❑ Rn 表示通用寄存器。

❑ immmed 表示立即数。

例如:

```
main:
    mov r8, #0xd3          ; r8=0xd3
    mov r9, #0xd3          ; r9=0xd3
    msr cpsr, r9           ; CPSR=r9. 把r9写入CPSR, 将改写CPSR所有位
    msr spsr, r9           ; SPSR=r9. 把r9写入SPSR, 将改写SPSR所有位
    msr cpsr, #0xd3        ; CPSR=0xd3. 把0xd3写入CPSR, 将改写CPSR所有位
    msr spsr, #0xd3        ; SPSR=0xd3. 把0xd3写入SPSR, 将改写SPSR所有位
    msr cpsr_f, #0x10       ; 把0x10写入CPSR寄存器的24~31位
    msr spsr_f, #0x10       ; 把0x10写入SPSR寄存器的24~31位
    msr cpsr_s, #0x10       ; 把0x10写入CPSR寄存器的16~23位
    msr spsr_s, #0x10       ; 把0x10写入SPSR寄存器的16~23位
    msr cpsr_x, #0          ; 把0写入CPSR寄存器的8~15位
    msr spsr_x, #0          ; 把0写入SPSR寄存器的8~15位
    msr cpsr_c, #0xd3       ; 把0xd3写入CPSR寄存器的0~7位
    msr spsr_c, #0xd3       ; 把0xd3写入SPSR寄存器的0~7位
    msr cpsr_c, r8          ; 把r8低8位写入CPSR寄存器的0~7位。这导致处理器
                           ; 切换到系统管理模式、ARM状态、关掉IRQ、IFQ中断
    msr spsr_c, r8          ; 把r8低8位写入SPSR寄存器的0~7位
```

3.8.6 协处理器操作指令

ARM920T 处理器为了能扩展系统功能, 支持最多加入 15 个协处理器, 如内存管理单元、浮点运算器、图像协处理器、通信调试协处理器等。每种协处理器都有各自特定的功能, 并且有些协处理器还能执行自己特定的指令。要控制并和这些协处理器进行通信, 当然需要一些特定的指令。ARM920T 处理器一共提供了 5 条指令, 如下。

- ❑ mrc 指令。
- ❑ mcr 指令。
- ❑ ldc 指令。
- ❑ stc 指令。
- ❑ cdp 指令。

1. mrc 指令

mrc 指令将协处理器寄存器的内容传送到 ARM920T 处理器的寄存器中, 如果协处理器不能完成这一操作, ARM920T 处理器将产生未定义指令的异常。

在汇编中的用法:

`mrc{cond} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, opcode_2}`

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ `cond` 表示条件码。条件满足则执行，不写条件码则无条件执行。
- ☐ `coproc` 表示协处理器编号。
- ☐ `opcode_1` 表示协处理器的执行操作码。
- ☐ `Rd` 表示目标寄存器，即 ARM920T 处理器的寄存器。
- ☐ `CRn` 表示协处理器的寄存器，存放第一个源操作数。
- ☐ `CRm` 表示附加的协处理器寄存器或者附加操作数的寄存器。
- ☐ `opcode_2` 表示协处理器的执行操作码 2，有时可能需要几个操作码。

例如：

```
main:
    mov r0, #0           ; r0=0
    mov r4, #0           ; r4=0
    mrc p15, 0, r0, c5, c0, 0    ; 把cp15协处理器的c5寄存器，传送到ARM920T处理器r0寄存器
                                ; 中，两个操作码都是0
    mrc p15, 0, r4, c6, c0, 0    ; 把cp15协处理器的c6寄存器，传送到ARM920T处理器r4寄存器
                                ; 中，两个操作码都是0
```

2. mcr 指令

`mcr` 指令将 ARM920T 处理器的寄存器中的内容传送到协处理器的寄存器中。如果协处理器不能完成这一操作，ARM920T 处理器将产生未定义指令的异常。

在汇编中的用法：

`mcr{cond} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, opcode_2}`

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ `cond` 表示条件码。条件满足则执行，不写条件码则无条件执行。
- ☐ `coproc` 表示协处理器编号。
- ☐ `opcode_1` 表示协处理器的执行操作码。
- ☐ `Rd` 表示 ARM920T 处理器的寄存器。
- ☐ `CRn` 表示协处理器的寄存器。
- ☐ `CRm` 表示附加的协处理器寄存器或者附加操作数的寄存器。
- ☐ `opcode_2` 表示协处理器的执行操作码 2，有时可能需要几个操作码。

例如：

```
main:
    mov r0, #0           ; r0=0
    mov r4, #0           ; r4=0
```

```

mcr p15,0,r0,c1,c0,0      ; 把ARM920T处理器r0寄存器,传送到cp15协处理器的c1寄存
                           ; 器中,两个操作码都是0
mcr p15,0,r4,c2,c0,0      ; 把ARM920T处理器r4寄存器,传送到cp15协处理器的c2寄存
                           ; 器,两个操作码都是0

```

3. ldc 指令

ldc 指令从一系列连续的内存单元中装载数据到协处理器的寄存器中。如果协处理器不能完成这一操作,ARM920T 处理器将产生未定义指令的异常。

在汇编中的用法:

```
ldc{cond}{l} <coproc>,<CRd>,<addr_mode>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。条件满足则执行,不写条件码则无条件执行。
- ☐ l 表示读取长数据,如双精度数据。
- ☐ coproc 表示协处理器编号。
- ☐ CRd 表示协处理器的寄存器,用于作为目标寄存器。
- ☐ addr_mode 表示内存地址,参见 ldr、str 指令。

例如:

```

main:
    mov r0,#0x80           ; r0=0x80
    mov r4,#0x40           ; r4=0x40
    ldc p6,c4,[r4,#4]      ; 把ARM920T处理器r4寄存器值加4的地址的内存单元的值装载
                           ; 到p6协处理器的c4寄存器中
    ldc p3,c4,[r0]         ; 把ARM920T处理器r0寄存器的值为地址的内存单元的值装载到
                           ; p3协处理器的c4寄存器中

```

4. stc 指令

stc 指令将协处理器的寄存器存储到一系列连续的内存单元中。如果协处理器不能完成这一操作,ARM920T 处理器将产生未定义指令的异常。

在汇编中的用法:

```
stc{cond}{l} <coproc>,<CRd>,<addr_mode>
```

- ☐ {} 中的表示可选项。
- ☐ <> 中的表示必需项。
- ☐ cond 表示条件码。条件满足则执行,不写条件码则无条件执行。
- ☐ l 表示存储长数据,如双精度数据。
- ☐ coproc 表示协处理器编号。
- ☐ CRd 表示协处理器的寄存器,用于作为目标寄存器。
- ☐ addr_mode 表示内存地址,参见 ldr、str 指令。

例如:

```
main:
    mov r0,#0x80          ; r0=0x80
    mov r4,#0x40          ; r4=0x40
    stc p8,c8,[r4]        ; 把p8协处理器的c8寄存器存储到以ARM920T处理
                          ; 器r4寄存器的值为地址的内存单元中
    stc p3,c4,[r0]        ; 把p3协处理器的c4寄存器存储到以ARM920T处理
                          ; 器r0寄存器的值为地址的内存单元中
```

5. cdp 指令

cdp 指令用于 ARM920T 处理器通知相关协处理器执行特定的操作。如果协处理器不能成功完成特定的操作, ARM920T 处理器将产生未定义指令的异常。

在汇编中的用法:

```
cdp{cond} <coproc>,<opcode_1>,<CRd>,<CRn>,<CRm>,<opcode_2>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。条件满足则执行, 不写条件码则无条件执行。
- coproc 表示协处理器编号。
- opcode_1 表示协处理器的执行操作码。
- CRd 表示协处理器的寄存器, 作为目标寄存器。
- CRn 表示协处理器的寄存器, 存放第一个操作数。
- CRm 表示附加的协处理器寄存器或者附加操作数的寄存器。
- opcode_2 表示协处理器的执行操作码 2, 有时可能要几个操作码。

例如:

```
main:
    mov r0,#0              ; r0=0
    mov r4,#0              ; r4=0
    cdp p5,2,c12,c10,c3,4 ; 该指令完成协处理器p5的初始化
```

3.8.7 异常中断产生指令

几乎所有现代处理器都提供至少一条异常中断产生指令, 如 x86 处理器上的 int 指令。为什么要有这样的指令, 因为现代处理器要安全地隔离操作系统内核和应用软件, 使应用软件不能随便运行特权指令和随意访问操作系统内核的内存空间, 然而应用软件又必须和操作系统内核通信、调用操作系统服务。于是开发处理器的工程师就想了个方法: 设计一条指令, 处理器一旦执行这条指令, 处理器就跳转到一个固定的地址开始运行, 而这个地址上放的就是操作系统内核的代码。这样, 处理器的控制权就稳定地从应用软件手里转移到操作系统内核里。若应用软件以其他方式访问内核内存空间或者运行特权指令, 都会被

处理器认为是“非法操作”，处理器捕获到“非法操作”，最终处理器的控制权也会转移到操作系统内核代码里。详情可以参阅异常和中断的介绍。ARM920T 也提供一条这种类型的指令如下。

swi 指令的运行将导致 ARM920T 处理器产生软件中断异常，执行这条指令时会使处理器硬件执行一些操作：① 把处理器设为 ARM 状态和系统管理模式；② 把 swi 指令后的下一条指令的地址放进系统管理模式下的 R14 寄存器中，把 CPSR 寄存器的内容放进系统管理模式下的 SPSR 寄存器中；③ 禁止外部中断；④ 强制 PC 寄存器中的值为 swi 向量，这个向量的地址根据系统配置的不同而不同，即低地址向量是 0x00000008，高地址向量是 0xFFFF0008。这条指令通常用于作为系统调用的陷入指令。

在汇编中的用法：

```
swi{cond} <immed_24>
```

- {} 中的表示可选项。
- <> 中的表示必需项。
- cond 表示条件码。条件满足则执行，不写条件码则无条件执行。
- immed_24 表示一个 24 位的立即数。处理器运行这条指令时忽略这条指令的低 24 位，它不对这低 24 位数据进行检查。即不管在这低 24 个二进制位里写入任何数据，处理器始终认为这是一条 swi 指令。很多操作系统用这低 24 位存放系统调用号。不过我们不用这种方式。

例如：

```
mian:
    swi #0          ; ARM920T处理器产生软中断异常，swi指令的低24位存放的是0
    swi #5          ; ARM920T处理器产生软中断异常，swi指令的低24位存放的是5
    mov r0,#10      ; r0=10，用r0寄存器传递系统调用号
    swi #0          ; ARM920T处理器产生软中断异常，swi指令的低24位存放的是0
```

这可真是一个痛苦而又漫长的过程，还好我们已经完成了。一共了解了 35 条指令，后面用这些指令就能写出操作系统内核关键部分的代码了。

3.9 MMU

MMU 是一种硬件实现的用于内存管理的器件。本节先概述 MMU 是什么，在 ARM920T 中是什么角色，有什么作用，接着了解为什么需要 MMU，然后看一看 ARM920T 中完成 MMU 功能的设备，最后详细了解 MMU 用于管理和控制内存的页表。

3.9.1 MMU 概述

MMU 即内存管理单元，它是 ARM920T 处理器存储系统的重要组成部分。它对处理器

发出的访存地址进行映射和检查,可以让处理器发出的访存地址去访问不同的物理内存单元。例如,处理器发出的地址是0,通过MMU后则访问的可能是以4096为物理地址的内存单元。同时它还检查这个地址是不是存在以及有没有访问权限。例如,我们可以通过对MMU的设置,让应用软件不能访问操作系统内核的存储空间。

前面介绍的内容都假定系统中没有MMU或者MMU处于关闭状态,如图3-17所示。

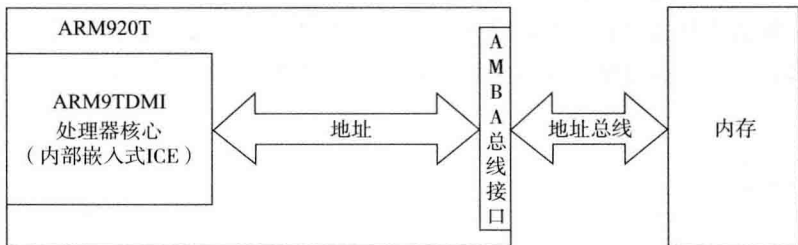


图 3-17 ARM920T 关闭 MMU 的情况

在图3-17中,我们看到在没有MMU或者MMU关闭状态下,ARM920T的内核即ARM9TDMI发出的地址,直接被送到总线上作为物理地址去访问内存。有很多简单的硬件系统确实和图3-17一样没有MMU,或者有个比MMU更简单的器件叫MPU即内存保护单元,没有MMU的地址映射功能。

当然,ARM920T处理器中采用的是功能相当强大的MMU器件。这些功能在后面会有所介绍。当开启ARM920T处理器上的MMU时,情况如图3-18所示。

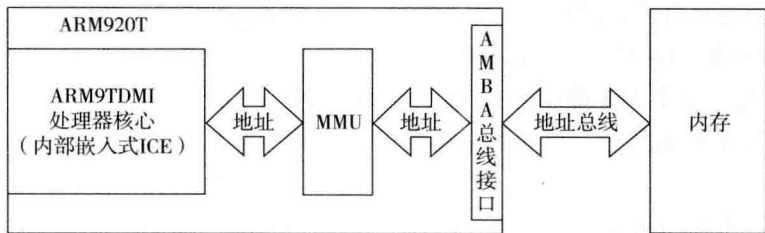


图 3-18 ARM920T 开启 MMU 的情况

图3-18所示的是启用ARM920T处理器的MMU时的情况。当然这是个简易图,ARM920T处理器的MMU比图3-18中的MMU复杂多了。我们可以暂时把这个图上的MMU想象成一个黑盒子。ARM920T处理器的内核即ARM9TDMI发出的地址,首先要通过这个黑盒子才能被送到总线上去访问内存。当ARM920T处理器的内核ARM9TDMI发出的地址到达MMU时,MMU首先检查这个地址是否能被访问,然后用一套机制对这个地址进行转换,生成一个新的地址,最后把这个新生成的地址放到总线上去访问内存。人们对这两种地址用了两个专业的术语,从处理器内核ARM9TDMI到MMU的地址,叫虚拟地址,后

面我们会看到 ARM920T 比这个情况更复杂，因为它还有一种地址。把 MMU 放到总线上的地址称为物理地址。

后面介绍相关内容时，都假定是图 3-18 所示的情况。下面就去仔细看看这个 MMU。

3.9.2 为什么要有 MMU

3.9.1 节中大概了解了 MMU 是干什么的，不外乎就是两点：一是内存地址映射；二是内存空间保护。那么为什么要有 MMU 这个硬件呢，或者说它给操作系统及应用软件的设计与开发带来什么方便以及影响呢。我们先从一个小程序的例子开始，程序代码如下：

```
main:
    mov r0,#1
    mov r1,#2
    ldr r2,add_func_l    ; 这是条伪指令，把add_func_l标号地址的内容装入r2寄存器
    bl r2
die:
    b die
add_func:
    add r0,r0,r1
    bx lr
.align 4
add_func_l: .word add_func ; 定义一个内存字空间存放标号add_func的地址
```

来看看汇编器和链接器对这段代码的处理结果，我们同时指定链接器从 0 地址开始链接和生成纯指令的二进制文件。装入内存后的情况如图 3-19 所示。

为了便于阅读，笔者画上了标号和物理地址，并且在内存字单元中还是用的汇编指令符号。注意内存芯片并不认识标号和这些符号，它只能在每个内存字单元中存放它自己都不认识的 32 个二进制的数数据位。笔者这样做仅仅是为了便于阅读。ARM920T 处理器的 ARM 状态的指令都是一个字大小即 32 位。在汇编语言中，ldr 指令是条伪指令，它具体被汇编器编译成什么样的机器指令，是由汇编器根据具体情况决定的。这里它会编译成如图 3-19 所示。

我们只关注上面两指令：

1) ldr r2,[pc,#20]，这条指令执行后 R2 中为 0x14，当前 PC 加 20 的地址正好是 0x1C，这里的 20 和 0x14 是由编译器和链接器写入的。

2) bl r2，保存该条指令的下一条指令的地址到 R14，然后跳转到 0x14 地址上执行。

可以发现，即便再简单的程序，也会和内存地址打交道。因为内存芯片只能根据地址来读取或者存放指令和数据。那么编译器和链接器在把我们的代码转换成二进制可执行文件时，自然也要在程序指令数据

标号	物理内存	物理地址
main	mov r0,#1	0x0
	mov r1,#2	0x4
	ldr r2,[pc,#20]	0x8
	bl r2	0xc
die	b die	0x10
add_func	add r0,r0,r1	0x14
	bx lr	0x18
add_func_l	0x14	0x1c
		0x20

图 3-19 程序指令装入内存中的情况

中写入相应的地址值，程序才能正常工作。

由上面的例子我们发现如下两个问题。

1) 程序只能装载到 0 地址开始内存中运行，因为“bl r2”指令执行时它会跳转到 0x14 地址，一旦移动这个程序的位置，0x14 地址的内存单元中不再是这个程序的指令。

2) 如果另一个程序，也是从 0 地址开始链接并且也要装载到 0 地址开始的内存中运行，这样就无法同时运行多个程序了，因为第二个程序会覆盖第一个程序的内存空间。当然让每个程序从不同的地址开始链接也是可以的。如果你的操作系统有数以万计的应用，并且这些应用来自不同的厂商，你会发现这种情况有多么糟糕。

我们期望结果是下面这样的。

1) 每个应用软件或者说进程，都能看到一个完整且连续的地址空间。例如，ARM920T 处理器的地址可以为 0 ~ 0xFFFFFFFF，有 4GB 的地址空间。

2) 对每个应用软件或者说进程，这 4GB 的地址空间是独立的，其他程序是不能占有的。

3) 由于 1)、2) 两点，编译器和链接器总是可以从这个连续的地址空间的某个地址开始编译链接我们的程序，而且每种应用软件都一样。

然而这只是一个美好的想法，我们十分清楚地知道，一个程序要运行就必须装进物理内存中。那么有没有办法可以解决这种问题呢。那些敢想敢做的人，经过不懈的努力终于开发出 MMU 这种器件。

人们在 CPU 和内存之间加入了 MMU，如图 3-18 所示。让 CPU 在运行指令时发出的访问地址先通过 MMU 的某种机制转换之后再去访问内存。还是前面的那个例子，如图 3-20 所示。

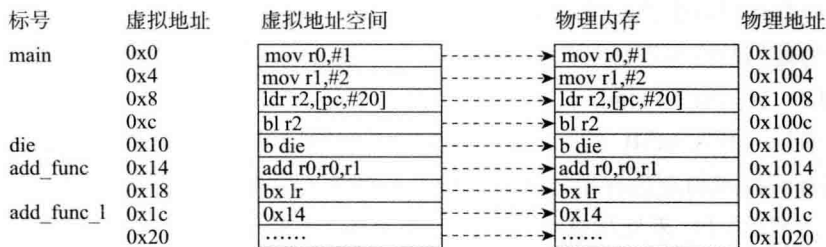


图 3-20 加入 MMU 后访问内存中指令的情况

由图 3-20 可知，MMU 把从 0 开始的虚拟地址空间映射到了从 0x1000 开始的物理内存空间。当处理器从 0 地址开始读取程序指令时，实则是读取到是 0x1000 地址内存单元中的指令。这时我们发现改变 MMU 的映射规则，就能把程序装载进以任意地址开始的物理内存空间中，只要这个空间可以装载且装得下这个程序就行。

再来看看多道程序的情况，我们假定每个应用程序都是从 0 地址开始链接的，并且它们拥有 4GB 的虚拟地址空间。如图 3-21 所示。

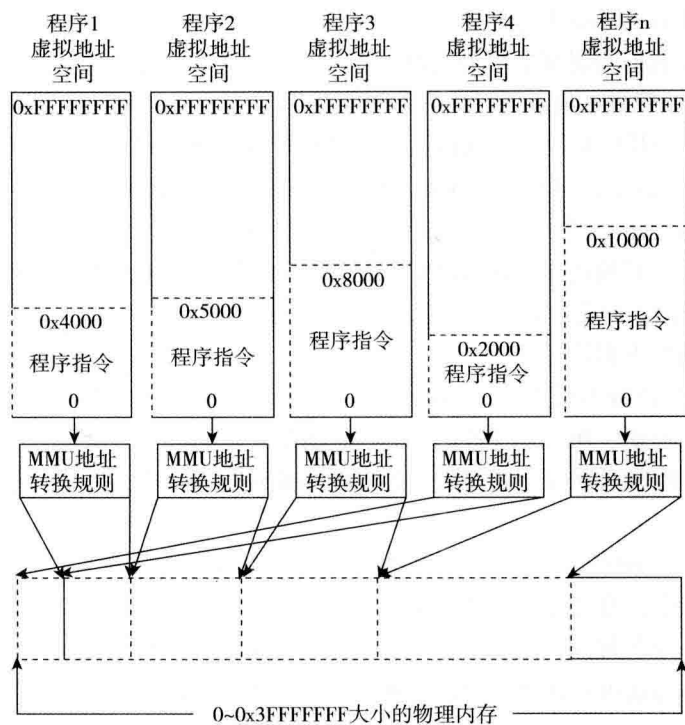


图 3-21 多道程序的虚拟地址空间到物理地址空间的映射

通过观察图 3-21 不难发现，操作系统内核只要牢牢地控制每个应用程序的 MMU 的地址转换规则，并且调度另一个应用程序运行时就让 MMU 使用另一个应用程序的地址转换规则。操作系统就可以自由地装载应用程序到任意物理内存空间中。应用软件开发商也少了不少麻烦，因为他们始终可以认为他们拥有一个非常大而且连续的内存空间，并且这个内存空间就是他们的，神圣而不可侵犯。

进一步来说，开发 MMU 的人，还让 MMU 能够通过具体的地址转换规则判断当前这个地址能不能被转换成物理内存地址去访问内存。如果不能被转换，那么 MMU 就向 CPU 发送一个信号，请求 CPU 来处理这个问题。由此操作系统就能利用这种机制。例如，对内存实施保护，还可以实现这样的机制：只装载应用软件的一部分，当应用程序运行到下一部分时，由于 MMU 发现地址不能转换，所以向 CPU 发出地址转换失败的信号，CPU 运行操作系统的处理程序：再次装载应用程序的下一部分，并填写好地址转换规则。由于这样，操作系统就可以充分利用有限的物理内存，使之同时运行足够多的应用软件。

MMU 的地址转换规则，后面会详细介绍，这里我们不必知道得那么清楚。

到这里应该明白了，为什么要有 MMU。MMU 给我们开发软件带来了巨大的方便，而且还能让我们的程序运行得更加稳定、可靠。下面我们就去了解 ARM920T 处理器中的 MMU 器件的具体实现。

3.9.3 ARM920T CP15 协处理器

从前面我们得知，ARM920T 处理器可以通过协处理器接口增加多达 16 个协处理器。CP15 协处理器就是其中之一。

CP15 是专门负责 ARM920T 处理器的存储系统的，如 MMU、Cache、MPU 的控制等。不同版本的 ARM 处理器的实现都有所不同。有的 ARM 处理器没有 Cache，有的 ARM 处理器没有 MMU，有的 ARM 处理器只有 MPU 即内存保护单元。CP15 结合其他技术共同实现了 ARM 处理器的存储系统。

ARM920T 处理器实现了 MMU 和 Cache，但是没有 MPU，因为 MMU 已经包含了 MPU 的功能，如图 3-22 所示。这个图可能在前面就见过。我们重新把它搬到这里。

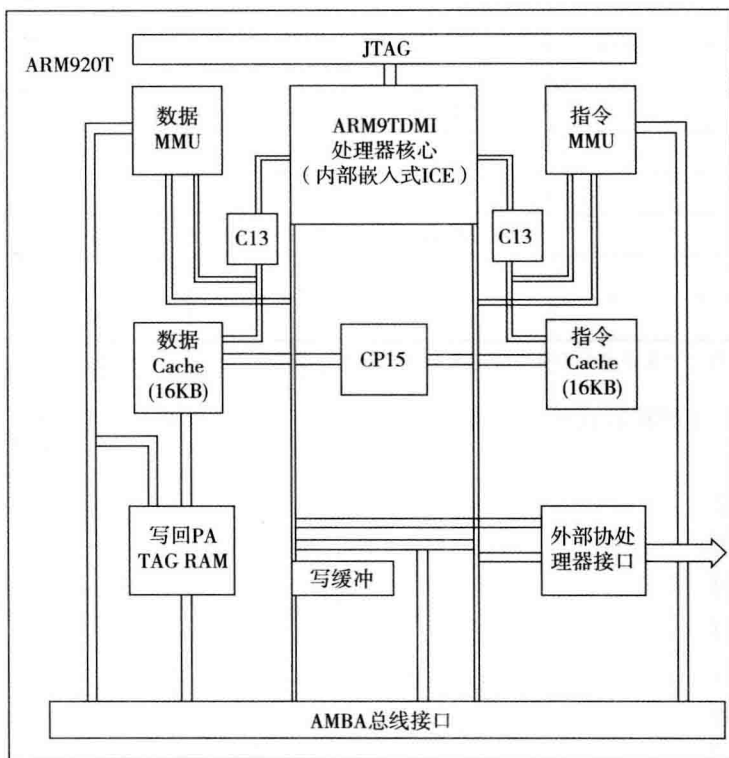


图 3-22 ARM920T 详细的逻辑结构

从图 3-22 不难发现 MMU、Cache 都直接或者间接地连接在 CP15 上。程序通过操作 CP15 就能达到控制 MMU、Cache 的目的。当然，CP15 提供给程序的接口就是一大堆寄存器。

CP15 一共有 16 个 32 位的寄存器，如表 3-9 所示。

表3-9 CP15协处理器的寄存器

寄 存 器	MMU 中的作用	MPU 中的作用
c0	ID 编码和 Cache 类型	
c1	各种控制位	
c2	页表基地址	Cachability 的控制位
c3	域访问控制位	Bufferability 控制位
c4	保留	保留
c5	内存访问失败的状态	访问权限控制位
c6	内存访问失败的地址	保护区域控制
c7	高速缓存和写缓存控制	
c8	TLB 控制	保留
c9	高速缓存锁定	
c10	TLB 锁定	保留
c11		
c12		
c13	进程标识符	
c14		
c15	因不同设计而异	因不同设计而异

注：表中空白的地方表示定义不明确，不用担心也不用关心这些。

下面我们来分别看看这些寄存器，注意我们不会关注 MPU 下的情况。我们要关注的 CP15 寄存器如下。

- ☐ c0 寄存器。
- ☐ c1 寄存器。
- ☐ c2 寄存器。
- ☐ c3 寄存器。
- ☐ c5 寄存器。
- ☐ c6 寄存器。
- ☐ c7 寄存器。
- ☐ c8 寄存器。
- ☐ c9 寄存器。
- ☐ c10 寄存器。
- ☐ c13 寄存器。

1. c0 寄存器

c0 寄存器稍微复杂一些，它对应两个不同的物理寄存器，这两个不同的物理寄存器

分别存放 ARM 处理器相关信息及 ID 和 Cache 类型。操作时用 mrc 指令的第二个操作码 opcode_2 区分。关于访问协处理器及其指令，请参阅指令集的相关章节。opcode_2 编码如表 3-10 所示。

表3-10 mrc指令第二个操作码opcode_2

opcode_2 编码	对应的寄存器
0b000	主 ID 标识寄存器
0b001	Cache 类型寄存器
其他	保留

注：0b 表示紧跟其后的是二进制数据

表 3-10 中两个寄存器都是 32 位的，先来看看主 ID 标识寄存器的格式，如图 3-23 所示。

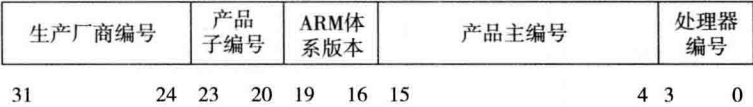


图 3-23 主 ID 标识寄存器

主 ID 标识寄存器的各位段如表 3-11 所示。

表3-11 主ID标识寄存器各位段

寄存器中的位段	相 关 说 明
位 [3:0]	生产商定义的处理器版本号
位 [15:4]	生产商定义的产品主编号，其中最高 4 位即位 [15:12] 可能的取值为 0 ~ 7 但不能是 0 或 7
位 [19:16]	ARM 体系的版本号，可能的取值如下： ☐ 0x1，ARM 体系版本 4 ☐ 0x2，ARM 体系版本 4T ☐ 0x3，ARM 体系版本 5 ☐ 0x4，ARM 体系版本 5T ☐ 0x5，ARM 体系版本 5TE ☐ 其他，由 ARM 公司保留将来使用
位 [23:20]	生产商定义的产品子编号，当产品主编号相同时，使用子编号来区分不同的产品子类，如产品中不同的高速缓存的大小等
位 [31:24]	生产厂商的编号，现在已经定义的有以下值： ☐ 0x41=A 表示 ARM 公司 ☐ 0x44=D 表示 Digital Equipment 公司 ☐ 0x69=I 表示 Intel 公司

根据表 3-10 可知，当 mrc 指令的 opcode_2 编码为 0b001 时，访问的就是 Cache 类型寄存器。这个的寄存器格式如图 3-24 所示。

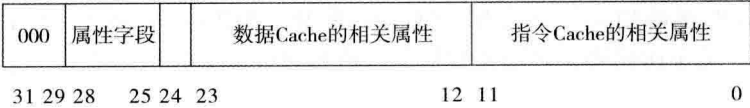


图 3-24 Cache 类型寄存器

Cache 类型寄存器各位段如表 3-12 所示。

表3-12 Cache类型寄存器各位段

寄存器中的位段

相关说明

定义指令 Cache 的相关属性，如下

位段	相关说明	
位 [1:0]	定义块大小，其编码如下	
	编码	Cache 块大小
	0b00	8B
	0b01	16B
	0b10	32B
	0b11	64B

位 [2:2]

取值 0/1，将影响其他位段

位 [5:3]	定义 Cache 的相联特性，其编码如下		
	编码	位 [2:2]=0 时的情况	位 [2:2]=1 时的情况
	0b000	1 路相联	没有 Cache
	0b001	2 路相联	3 路相联
	0b010	4 路相联	6 路相联
	0b011	8 路相联	12 路相联
	0b100	16 路相联	24 路相联
	0b101	32 路相联	48 路相联
	0b110	64 路相联	96 路相联
	0b111	128 路相联	192 路相联

位 [8:6]

	定义 Cache 的容量，其编码如下		
	编码	位 [2:2]=0 时的情况	位 [2:2]=1 时的情况
	0b000	0.5KB	0.75KB
	0b001	1KB	1.5KB
	0b010	2KB	3KB
	0b011	4KB	6KB
	0b100	8KB	12KB

位 [11:0]

(续)

寄存器中的位段	相 关 说 明			
位 [11:0]	位 [8:6]	0b101	16KB	24KB
		0b110	32KB	48KB
		0b111	64KB	96KB
	位 [11:9]	保留, 值为 0b000		
位 [23:12]	定义数据 Cache 的相关属性, 如下:			
	位段	相关说明		
	位 [13:12]	定义块大小, 其编码如下		
		编码	Cache 块大小	
		0b00	8 字节	
		0b01	16 字节	
		0b10	32 字节	
		0b11	64 字节	
	位 [14:14]	取值 0/1, 将影响其他位段		
	位 [17:15]	定义数据 Cache 的相联特性, 其编码如下		
		编码	位 [14:14]=0 时的情况	位 [14:14]=1 时的情况
		0b000	1 路相联	没有 Cache
		0b001	2 路相联	3 路相联
		0b010	4 路相联	6 路相联
		0b011	8 路相联	12 路相联
		0b100	16 路相联	24 路相联
		0b101	32 路相联	48 路相联
		0b110	64 路相联	96 路相联
位 [20:18]	定义数据 Cache 的容量, 其编码如下			
	编码	位 [14:14]=0 时的情况	位 [14:14]=1 时的情况	
	0b000	0.5KB	0.75KB	
	0b001	1KB	1.5KB	
	0b010	2KB	3KB	
	0b011	4KB	6KB	
0b100	8KB	12KB		

(续)

寄存器中的位段	相 关 说 明			
位 [23:12]	位 [20:18]	定义数据 Cache 的容量，其编码如下		
		0b101	16KB	24KB
		0b110	32KB	48KB
		0b111	64KB	96KB
	位 [23:21]	保留，值为 0b000		
位 [24]	为 0 时系统使用统一的 Cache，为 1 时系统分别使用指令 Cache 和数据 Cache。为 0 时，位 [23:12]、位 [11:0] 都定义同一个 Cache 的相关属性			
位 [28:25]	定义 Cache 的其他属性，如下			
	编码值	Cache 类型	Cache 内容清除方法	Cache 内容锁定方法
	0b0000	写通类型	不需要内容清除	不支持内容锁定
	0b0001	写回类型	数据块读取	不支持内容锁定
	0b0010	写回类型	由 c7 寄存器定义	不支持内容锁定
	0b0011	写回类型	由 c7 寄存器定义	支持格式 A，后面会介绍
	0b0100	写回类型	由 c7 寄存器定义	支持格式 B，后面会介绍
位 [31:29]	保留 0b000			

注：0b 表示紧跟其后的是二进制数据。

2. c1 寄存器

c1 寄存器是个控制寄存器，控制 ARM920T 处理器的存储系统。例如，打开、关闭 MMU，打开、关闭 Cache，设置中断异常向量地址即高地址向量，控制存储器访问对齐等。

c1 寄存器的格式如图 3-25 所示。

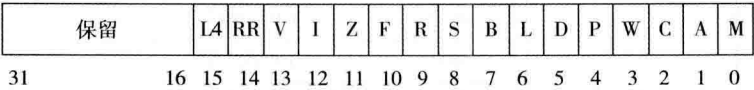


图 3-25 c1 寄存器

c1 寄存器各位段如表 3-13 所示。

表3-13 c1寄存器各位段

寄存器中的位段	相 关 说 明
M 位 [0:0]	0：禁止 MMU 或者 MPU；1：使能 MMU 或者 MPU。如果没有 MMU 或者 MPU，读取该位时返回 1，写入时忽略该位
A 位 [1:1]	0：禁止地址对齐检查；1：使能地址对齐检查

(续)

寄存器中的位段	相 关 说 明
C 位 [2:2]	0: 禁止数据 / 整个 Cache; 1: 使能数据 / 整个 Cache。如果没有 Cache, 读取该位时返回 0, 写入时忽略该位。如果不能禁止 Cache, 读取该位时返回 1, 写入时忽略该位
W 位 [3:3]	0: 禁止写缓冲; 1: 使能写缓冲。如果没有写缓冲, 读取该位时返回 0, 写入时忽略该位。如果不能禁止写缓冲, 读取该位时返回 1, 写入时忽略该位
P 位 [4:4]	0: 异常中断处理程序进入 32 位地址模式; 1: 异常中断处理程序进入 26 位地址模式。由于早期 ARM 处理器支持 26 位地址, 如果不向前兼容 26 位地址, 读取该位时返回 1, 写入时忽略该位
D 位 [5:5]	0: 禁止 26 位地址异常检查; 1: 使能 26 位地址异常检查。如果不向前兼容 26 位地址, 读取该位时返回 1, 写入时忽略该位
L 位 [6:6]	0: 选择早期中止模型; 1: 选择后期中止模型。对于 ARM 版本 3 以后的处理器, 读取该位时返回 1, 写入时忽略该位
B 位 [7:7]	0: 小端存储格式; 1: 大端存储格式。对于只支持小端存储格式的系统, 读取该位时返回 0, 写入时忽略该位; 对于只支持大端存储格式的系统, 读取该位时返回 1, 写入时忽略该位
S 位 [8:8]	在基于 MMU 的存储系统中, 本位用作系统保护。后面会详细介绍
R 位 [9:9]	在基于 MMU 的存储系统中, 本位用作 ROM 保护。后面会详细介绍
F 位 [10:10]	由具体生产商定义
Z 位 [11:11]	0: 禁止跳转预测功能; 1: 使能跳转预测功能。如果不支持跳转预测, 读取该位时返回 0, 写入时忽略该位
I 位 [12:12]	0: 禁止指令 Cache; 1: 使能指令 Cache。如果使用统一的 Cache 或者没有 Cache, 读取该位时返回 0, 写入时忽略该位; 如果不能禁止指令 Cache, 读取该位时返回 1, 写入时忽略该位
V 位 [13:13]	0: 选择低端异常中断向量 0x00000000 ~ 0x0000001c; 1: 选择高端异常中断向量 0xFFFF0000 ~ 0xFFFF001C。如果不支持高端向量, 读取该位时返回 0, 写入时忽略该位
RR 位 [14:14]	0: 常规的 Cache 淘汰算法, 如随机淘汰; 1: 预测性淘汰算法, 如 round-robin 淘汰算法
L4 位 [15:15]	0: 保持 ARMv5 以上版本的正常功能; 1: 将 ARMv5 以上版本与以前版本处理器兼容, 不根据跳转地址的 bit[0] 进行 ARM 状态和 Thumb 状态切换。对于 ARM920T 处理器该位保留
位 [31:16]	这些位保留将来使用

3. c2 寄存器

c2 寄存器保存的是 32 位的一级页表基地址。注意这个基地址是物理地址, 并且这个物理地址是 16KB 对齐的, 即这个地址的低 14 位是 0。关于什么是页表, 后面会详细介绍。c2 寄存器的格式如图 3-26 所示。

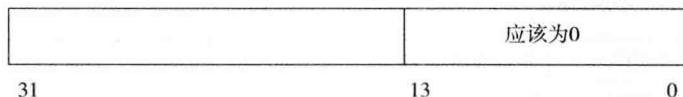


图 3-26 c2 寄存器

4. c3 寄存器

c3 寄存器定义了 ARM920T 处理器的 16 个域的访问权限。域就是一些页面的集合，关于页面后面会详细介绍。这个寄存器也是 32 位的，每两位表示一个域，故能表示 16 个域。其格式如图 3-27 所示。

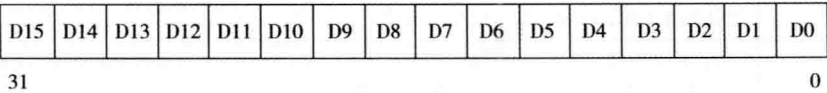


图 3-27 c3 寄存器

每个域的编码及含义如表 3-14 所示。

表3-14 c3寄存器中每个域的编码

域 编 码	访 问 类 型	相 关 说 明
0b00	没有访问权限	这时访问该域，将产生存储器访问失败
0b01	用户类型权限	根据页表条目（后面会介绍）中的相关控制位，决定是否允许特定的存储器访问
0b10	保留	该值将产生不可预料的结果
0b11	管理类型权限	不考虑页表条目中的相关控制位，这种情况下不会产生存储器访问失败

注：0b 表示紧跟其后的是二进制数据。

5. c5、c6 寄存器

当访问存储器失败时，c5 寄存器存放存储器访问失败时的状态，访问存储器失败的地址则存放在 c6 寄存器中。MMU 在地址不对齐、地址转换失败、域控制失败、访问权限不够这几种情况时，会产生存储器访问失败。c5、c6 寄存器的编码格式如图 3-28 所示。

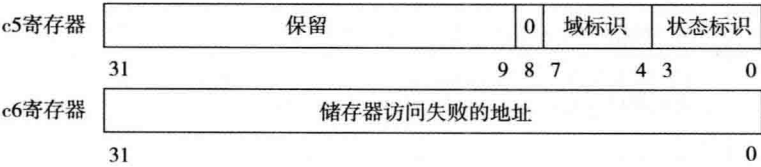


图 3-28 c5、c6 寄存器

c5 寄存器各位段如表 3-15 所示。

表3-15 c5寄存器各位段

寄存器中的位段	相 关 说 明			
位 [3:0]	存放引起存储器访问失败的访存类型。各类型值和相关说明如下			
	类型值	存储访问失败的原因	域标识有 / 无效	c6 失败地址寄存器有 / 无效
	0b0010	极端异常	无效	由生产商定义

(续)

寄存器中的位段	相 关 说 明			
位 [3:0]	0b0000	中断向量访问异常	无效	有效
	0b00x1*	地址对齐	无效	有效
	0b1100	一级页表访问失败	无效	有效
	0b1110	二级页表访问失败	有效	有效
	0b0101	基于段的地址变换失败	无效	有效
	0b0111	基于页的地址变换失败	有效	有效
	0b1001	基于段的存储器访问中域控制失败	有效	有效
	0b1011	基于页的存储器访问中域控制失败	有效	有效
	0b1101	基于段的存储器访问中访问权限控制失败	有效	有效
	0b1111	基于页的存储器访问中访问权限控制失败	有效	有效
	0b0100	基于段的 Cache 预取时外部存储系统失败	有效	有效
	0b0110	基于页的 Cache 预取时外部存储系统失败	有效	有效
	0b1000	基于段的非 Cache 预取时外部存储系统失败	有效	有效
	0b1010	基于页的非 Cache 预取时外部存储系统失败	有效	有效
位 [7:4]	存放了存储器访问失败时所在的域			
位 [8:8]	这个位为 0			
位 [31:9]	保留将来使用			

注：0b 表示紧跟其后的是二进制数据。0b00x1* 中的 “x” 表示未定义，可以是 0 也可以是 1。关于段、页表，后面会详细介绍。

6. c7 寄存器

c7 寄存器是一个只写寄存器，读操作将产生不可预知的后果。用来控制 Cache 和写缓存。这个寄存器有点复杂，寄存器编码格式如图 3-29 所示。

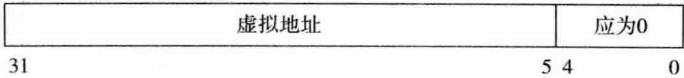


图 3-29 c7 寄存器

访问 c7 寄存器的指令格式如下：

```
mcr p15,0,<rd>,<c7>,crm,<opcode_2>
```

<rd>、<crm> 和 <opcode_2> 取值的不同组合将实现不同功能。如表 3-16 所示。这个表里可能有些东西不是很明白，没事，读者也许还不知道什么是 Cache，我们后面会专门介绍。

表3-16 访问c7寄存器时mcr指令的<rd>、<crm>和<opcode_2>取值

<rd> (表示写入的数据)	<crm>	<opcode_2>	相 关 说 明
0	c0	4	等待中断激活，这是因为 ARM 处理器进入省电状态时，需要外部中断激活
0	c5	0	将整个指令 Cache 的所有内容标识为无效。访问时将不命中，Cache 会从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
虚拟地址	c5	1	将指令 Cache 中的某一块标识为无效。访问到这块时，将从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
Cache 组号 / 组内序号	c5	2	将指令 Cache 中的某一块标识为无效。访问到这块时，将从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
0	c5	4	清空预取缓冲区，预取缓冲区是生产商定义的，预取缓冲区的内容会被回写到存储器中
0	c5	6	清空整个跳转目标 Cache，跳转目标 Cache 是生产商定义的，跳转目标 Cache 的内容会被回写到存储器中
生产商定义	c5	7	清空整个跳转目标 Cache 中的某块
0	c6	0	将整个数据 Cache 的所有内容标识为无效。访问时将不命中，Cache 会从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
虚拟地址	c6	1	将数据 Cache 中的某一块标识为无效。访问到这块时，将从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
Cache 组号 / 组内序号	c6	2	将数据 Cache 中的某一块标识为无效。访问到这块时，将从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
0	c7	0	将统一 Cache 或者数据 Cache 和指令 Cache 的所有内容标识为无效。访问时将不命中，Cache 会从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中
虚拟地址	c7	1	将统一 Cache 中的某一块标识为无效。访问到这块时，将从存储器中加载数据。对于写回类型的 Cache，并不将标识无效的内容回写到存储器中

(续)

<rd> (表示写入的数据)	<crm>	<opcode_2>	相 关 说 明
Cache 组号 / 组内序号	c7	2	将统一 Cache 中的某一块标识为无效。访问到这块时, 将从存储器中加载数据。对于写回类型的 Cache, 并不将标识无效的内容回写到存储器中
0	c8	2	等待中断激活, 这是因为 ARM 处理器进入省电状态时, 需要外部中断激活
虚拟地址	c10	1	清空数据 Cache 中的某一块。将该块内容回写到存储器中
Cache 组号 / 组内序号	c10	2	清空数据 Cache 中的某一块。将该块内容回写到存储器中
0	c10	4	清空写缓冲区。将写缓冲区中的内容全部回写到存储器中
虚拟地址	c11	1	清空统一 Cache 中的某一块。将该块内容回写到存储器中
Cache 组号 / 组内序号	c11	2	清空统一 Cache 中的某一块。将该块内容回写到存储器中
虚拟地址	c13	1	清空预取指令 Cache 中的某一块。将该块内容回写到存储器中
虚拟地址	c14	1	清空并标识无效数据 Cache 中的某一块
Cache 组号 / 组内序号	c14	2	清空并标识无效数据 Cache 中的某一块
虚拟地址	c15	1	清空并标识无效统一 Cache 中的某一块
Cache 组号 / 组内序号	c15	2	清空并标识无效统一 Cache 中的某一块

7. c8 寄存器

c8 寄存器用来控制清除 TLB 的内容, 是只写寄存器, 读操作将产生不可预知的后果。TLB 是 MMU 用于加速查询地址转换规则的一种快表。一些地址转换之后就存放在这个表中, 下次用到这个地址时就不用从存储器中读取转换规则了。寄存器编码格式如图 3-30 所示。

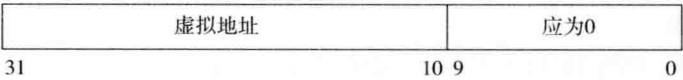


图 3-30 c8 寄存器

访问 c8 寄存器的指令格式如下:

```
mcr pl15,0,<rd>,<c8>,crm,<opcode_2>
```

<rd>、<crm> 和 <opcode_2> 的不同取值组合实现不同功能, 如表 3-17 所示。

表3-17 访问c8寄存器时mcr指令的<rd>、<crm>和<opcode_2>取值

<rd> (表示写入的数据)	<crm>	<opcode_2>	相关说明 (对 Cache 的影响)
0	c7	0	将统一 Cache 或者数据 Cache 和指令 Cache 标识为无效
虚拟地址	c7	1	将统一 Cache 中的单个地址转换规则标识为无效

(续)

<rd> (表示写入的数据)	<crm>	<opcode_2>	相关说明 (对 Cache 的影响)
0	c5	0	将整个指令 Cache 标识为无效
虚拟地址	c5	1	将整个指令 Cache 中的单个地址转换规则标识为无效
0	c6	0	将整个数据 Cache 标识为无效
虚拟地址	c6	1	将整个数据 Cache 中的单个地址转换规则标识为无效

8. c9 寄存器

c9 寄存器用于控制 Cache 内容锁定。寄存器编码格式如图 3-31 所示。

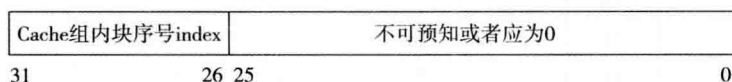


图 3-31 c9 寄存器

访问 c9 寄存器的指令格式如下：

```
mcr p15,0,<rd>,<c9>,c0,<opcode_2>
mrc p15,0,<rd>,<c9>,c0,<opcode_2>
```

如果系统中包含独立的指令 Cache 和数据 Cache，那么对应于数据 Cache 和指令 Cache 分别有一个独立的 Cache 内容锁定寄存器，<opcode_2> 用来选择其中的某个寄存器：

- ❑ <opcode_2>=1，选择指令 Cache 的内容锁定寄存器；
- ❑ <opcode_2>=0，选择数据 Cache 的内容锁定寄存器。

其中，index 表示当下一发生 Cache 未命中时，将预取的存储块存入 Cache 中该块对应的组中序号为 index 的 Cache 块中。此时序号为 0 ~ index-1 的 Cache 块被锁定，当发生 Cache 块替换时，从序号为 index 到其他关联性的块中选择被替换的块。

9. c10 寄存器

c10 寄存器用于控制 TLB 内容锁定。寄存器编码格式如图 3-32 所示。

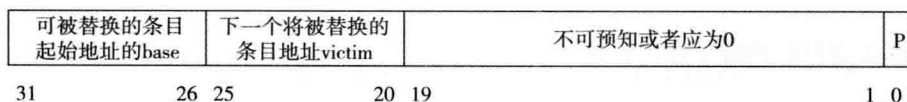


图 3-32 c10 寄存器

c10 寄存器各位段如表 3-18 所示。

表3-18 c10寄存器各位段

寄存器位段	相关说明
P 位 [0:0]	1：写入 TLB 的地址转换规则条目不会受到使整个 TLB 标识无效操作的影响，一直保持有效；0：写入 TLB 的地址转换规则条目将会受到使整个 TLB 标识无效操作的影响

(续)

寄存器位段	相 关 说 明
位 [19:1]	读取时不可预知，写入时应为 0
victim 位 [25:20]	指定下一次 TLB 没有命中（所需的地址转换规则条目没有包含在 TLB 中）时，从内存页表中读取所需的地址转换规则条目，并把该地址转换规则条目保存在 TLB 中的地址 victim 处
base 位 [31:26]	指定 TLB 替换时，所使用的地址范围，从 base 到（TLB 中条目数 -1）；字段 victim 的值应该包含在该范围内

访问 c10 寄存器的指令格式如下：

```
mcr p15,0,<rd>,<c10>,c0,<opcode_2>
mrc p15,0,<rd>,<c10>,c0,<opcode_2>
```

如果系统中包含独立的指令 TLB 和数据 TLB，那么对应于数据 TLB 和指令 TLB 分别有一个独立的 TLB 内容锁定寄存器，<opcode_2> 用来选择其中的某个寄存器：

- ☐ <opcode_2>=1，选择指令 TLB 的内容锁定寄存器；
- ☐ <opcode_2>=0，选择数据 TLB 的内容锁定寄存器。

10. c13 寄存器

c13 寄存器用于快速上下文切换。这个寄存器说简单也简单说复杂也复杂，因为它关系到快速上下文切换技术。我们慢慢看，首先 c13 寄存器的编码格式如图 3-33 所示。

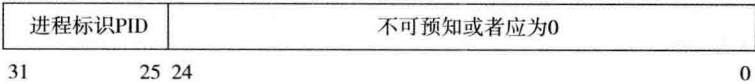


图 3-33 c13 寄存器

访问 c13 寄存器的指令格式如下：

```
mcr p15,0,<rd>,<c13>,c0,0
mrc p15,0,<rd>,<c13>,c0,0
```

图 3-33 中，PID 表示当前进程所在的进程空间块的编号，即当前进程的进程标识符，取值为 0 ~ 127。若 PID 为 0，MVA（变换后的虚拟地址）= VA（虚拟地址），即禁止快速上下文切换技术，系统复位后 PID=0；若非 0，即开启快速上下文切换技术。

快速上下文切换位于 CPU 和 MMU 之间，可以参见图 3-22 的 c13。如果两个进程使用了同样的虚拟地址空间，则对 CPU 而言，两个进程就使用了相同的虚拟地址空间。快速上下文切换机构对各个进程的虚拟地址再次进行变换。这样，系统中除了 CPU 之外的部分看到的是已经经过快速上下文切换机构变换的虚拟地址。如图 3-34 所示。

在 ARM 系统中，4GB 的虚拟空间被分成了 128 个进程空间块，每一个进程空间块大小为 32MB。该块中包含一个进程，该进程可以使用的虚拟地址空间为 0x00000000 ~ 0x01FFFFFF。这个地址范围也就是 CPU 看到的一个进程的虚拟地址空间。系统中有 128 个进程空间块编号，

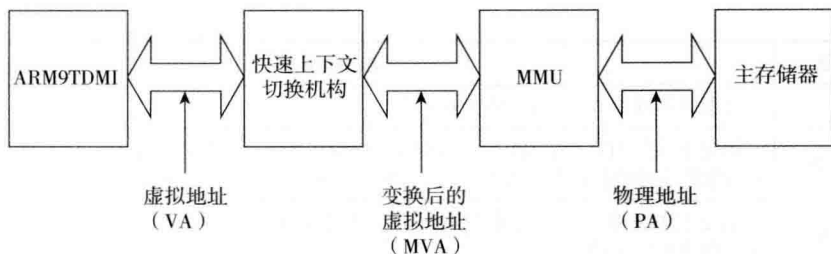


图 3-34 ARM920T 的快速上下文切换机构

即 $0 \sim 127$ ，标号为 X 的进程空间块中的进程实际使用的虚拟地址空间为 $X \times 0x02000000 \sim X \times 0x02000000 + 0x01FFFFFF$ ，这个地址空间是系统中除 CPU 外的其他部分看到的该进程所占用的虚拟地址空间。如图 3-35 所示。

到这里我们关注的所有寄存器就介绍完了。这些寄存器中我们肯定还有一些不明白的，不要怕，等到后面介绍相关内容时，再回过头来看这里就全明白了。一时的不懂，不要觉得失败。

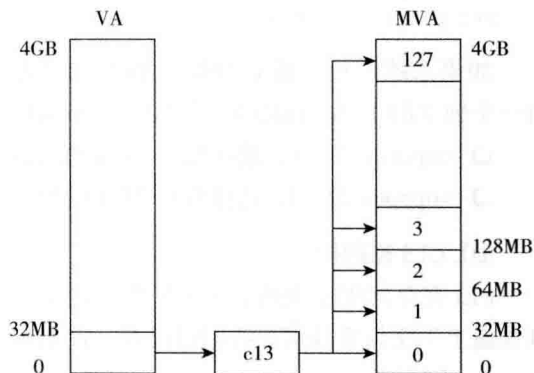


图 3-35 开启快速上下文切换结构后的虚拟地址空间

3.9.4 MMU 页表

前面我们说到 MMU 把虚拟地址转换成对应的物理地址，需要一些转换规则，实现这一转换规则的东西，实际上有个专业的术语叫页表。页表中的每个条目表示一个地址转换规则。

MMU 地址映射时是按照地址空间块映射的，即一块连续的虚拟地址空间对应一块连续的物理地址空间。这个地址空间块叫“页面”，如图 3-36 所示。这样说可能一下子不能明白，但看了后面的页表之后就明白了。

当然图 3-36 中的地址空间块的大小是可变的。ARM920T 处理器中的 MMU 支持 4 种不同大小的地址空间块，分别如下：

- ❑ 1MB，称为段；
- ❑ 64KB，称为大页；
- ❑ 4KB，称为小页；
- ❑ 1KB，称为极小页。

下面我们分别仔细介绍这 4 种不同大小的页面。在这之前先来看看什么是页表，页表在哪里，它里面是什么数据。页表是存在于内存中的一块数据，这块数据中有很多条目，在 ARM920T 的 MMU 下，每个条目是一个存储字大小即 32 位。相当一个整形数组，用 C 语言

可以这样描述：int pagetbl[xxxx]。具体有多少条目要视情况而定，后面会有介绍。如图 3-37 所示，这些条目的位段，有一定的格式。根据这些条目中不同的数据，MMU 转换后的物理地址和地址空间块即页面的大小也会有所不同。

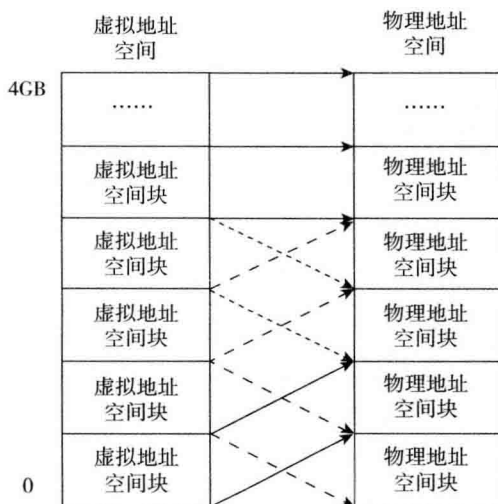


图 3-36 虚拟地址空间块到物理地址空间块的映射

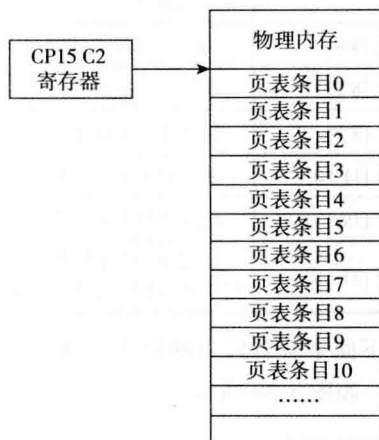


图 3-37 页表条目

1. 段

段映射下，ARM920T 的 MMU 把虚拟地址空间和物理地址空间分成 1MB 大小的地址空间块，即每个页面大小为 1MB。由于是 1MB 大小的页面，所以只需要一级页表。这个页表中有 4096 个条目。每个条目映射 1MB 大小的空间，正好对应于 ARM920T 4GB 的虚拟地址空间。段映射下的页表条目格式如图 3-38 所示。

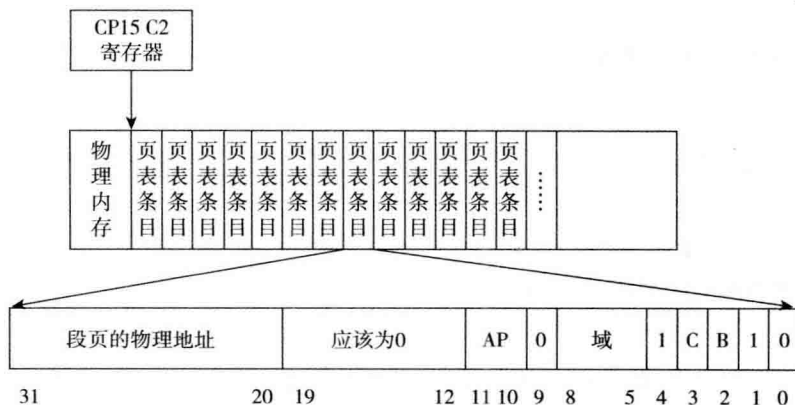


图 3-38 段映射下的页表条目格式

图 3-38 中的段页表条目各位段如表 3-19 所示。

表3-19 段页表条目各位段

条目位段	相关说明
位 [1:0]	页表条目类型标识，对于段页必须是二进制 10。00 表示无效映射，MMU 会发出访存失败 的信号
位 [3:2]	表示这个条目映射的内存空间，是否为回写缓存、写通缓存、非缓存、缓冲，或者无缓存 并且无缓冲
位 [4:4]	为 1、0 都行，但为了向后兼容应该为 1
位 [8:5]	表示这个条目映射的内存空间属于哪个域，参见 c3 寄存器
位 [9:9]	段页条目下总是为 0
位 [11:10]	表示这个条目映射的内存空间的访问权限
位 [19:12]	段页条目下总是为 0
位 [31:20]	表示这个条目映射的内存空间的物理地址，由于段页是 1MB 大小，所以地址的低 20 位始 终为 0，正因为如此，这个条目的低 20 位才可以用于访问属性控制等

下面来看看段页映射下，MMU 是如何利用内存中的页表实现虚拟地址到物理地址的转换的，如图 3-39 所示。

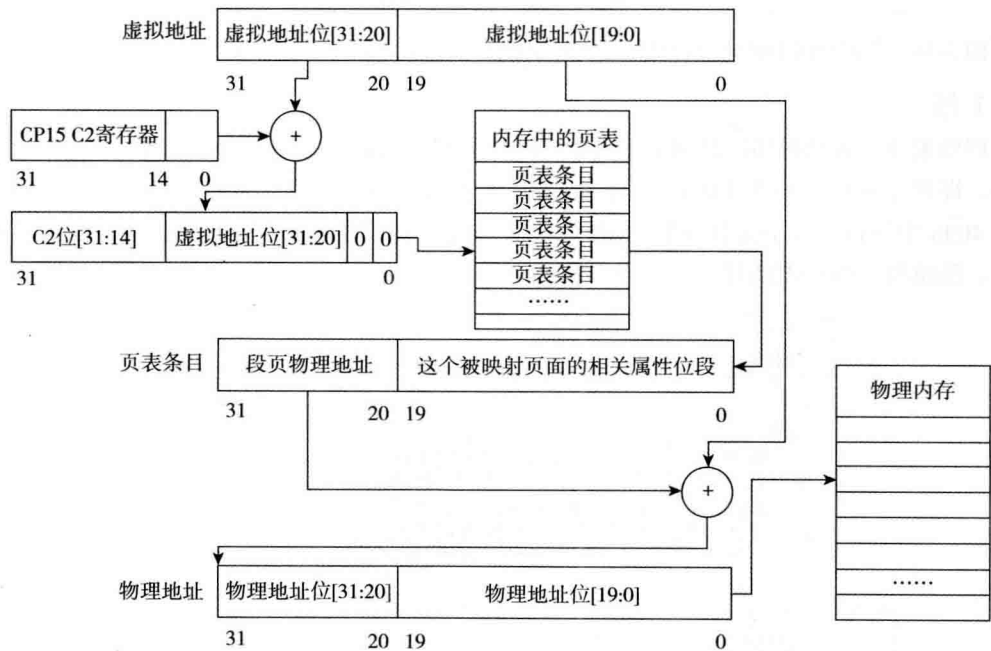


图 3-39 段映射下虚拟地址到物理地址的转换

图 3-39，MMU 的执行步骤如下。

第 1 步：MMU 拿到虚拟地址。

第2步: MMU 将 CP15 C2 寄存器中的高位 [31:14] 和虚拟地址的高位 [31:20] 并且外加两个为 0 的二进制位, 一起组成一个 32 位的地址, 这个地址就是指向页表条目的地址。

第3步: 用第2步中合成的地址读取一个页表条目。

第4步: 用页表条目中的高位 [31:20] 与虚拟地址的低位 [19:0] 一起组成 32 位的物理地址。

第5步: 用第4步中合成的物理地址去访问物理内存。

我们来看一个段映射下虚拟地址到物理地址的转换实例, 如图 3-40 所示。

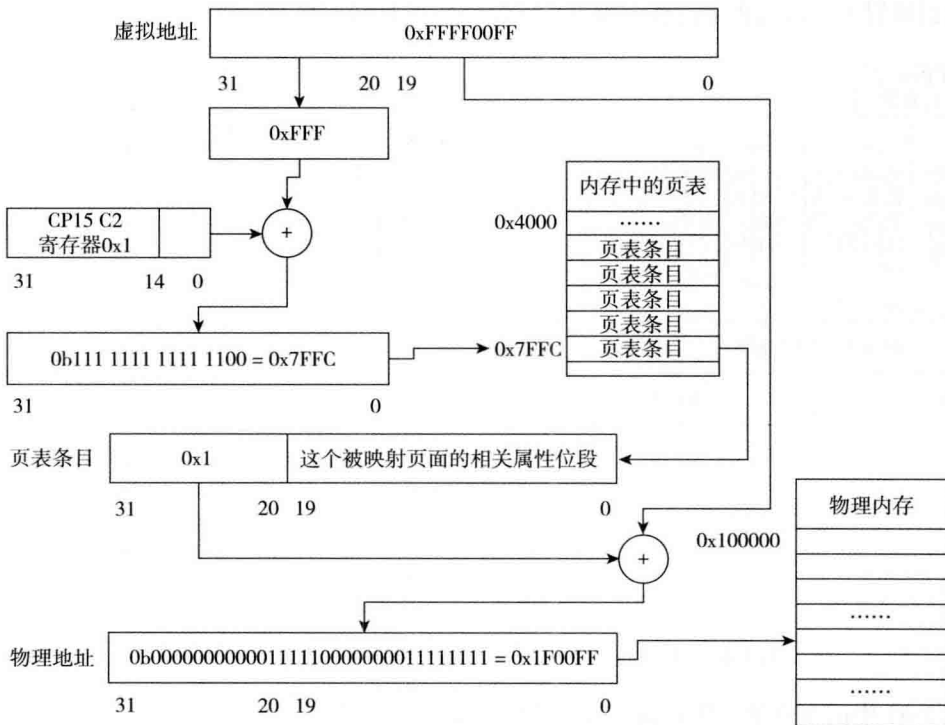


图 3-40 段映射下虚拟地址到物理地址的转换实例

这个例子演示了从虚拟地址 0xFFFF00FF 到物理地址 0x1F00FF 的映射过程。这个过程如下。

第1步: MMU 拿到虚拟地址: 0xFFFF00FF。

第2步: MMU 将 CP15 C2 寄存器中的高位 [31:14] (0x1) 和虚拟地址的高位 [31:20] (0xFFF) 并且外加两个为 0 的二进制位, 一起组成一个 32 位的地址 0x7FFC, 这个地址就是指向页表条目的地址。

第3步: 用第2步中合成的地址 0x7FFC, 读取一个页表条目。

第4步: 用页表条目中的高位 [31:20] (0x1) 与虚拟地址的低位 [19:0] (0xF00FF) 一起组成 32 位的物理地址 0x1F00FF。

第 5 步：用第 4 步中合成的物理地址 0x1FF0FF 去访问物理内存。

2. 大页

大页映射下，ARM920T 的 MMU 把虚拟地址空间和物理地址空间分成 64KB 大小的地址空间块，即每个页面大小为 64KB。由于是 64KB 大小的页面，所以需要两级页表。一级页表中有 4096 个条目。每个条目映射二级页表的空间，二级页表中有 256 个条目。在只有大页的情况下，只用到了前 16 个条目。每个二级页表的条目可以映射 64KB 大小的地址空间，大页映射下一级页表条目和二级页表条目的格式如图 3-41 所示。

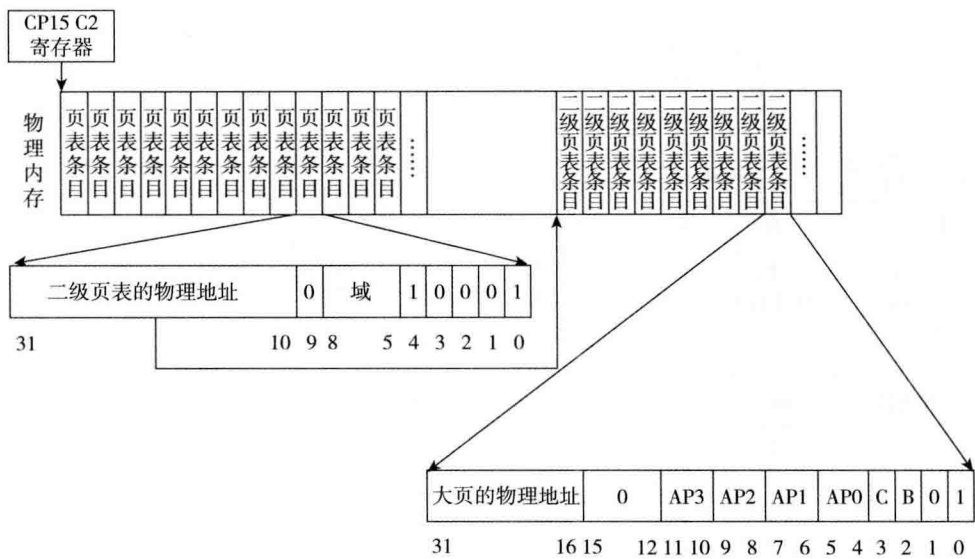


图 3-41 大页映射下一级页表条目和二级页表条目的格式

图 3-41 中的大页的一级页表条目的各位段如表 3-20 所示。

表3-20 大页的一级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识，对于大页必须是二进制 01
位 [3:2]	应该为 0
位 [4:4]	为 1、0 都行，但为了向后兼容应该为 1
位 [8:5]	表示这个条目映射的内存空间属于哪个域，参见 c3 寄存器
位 [9:9]	应该为 0
位 [31:10]	指向二级页表的物理地址，二级页表地址是 1KB 对齐的，所以地址的低 10 位始终为 0，正因为如此，这个条目的低 10 位才可以用于访问属性控制等

图 3-41 中的大页的二级页表条目的各位段如表 3-21 所示。

表3-21 大页的二级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识，对于大页必须是二进制 01
位 [3:2]	表示这个条目映射的内存空间，是否为回写缓存、写通缓存、非缓存、缓冲，或者无缓存并且无缓冲
位 [11:4]	一个大页可分为 4 个子页，每个子页 16KB AP0：控制第 1 个子页的访问权限 AP1：控制第 2 个子页的访问权限 AP2：控制第 3 个子页的访问权限 AP3：控制第 4 个子页的访问权限
位 [15:12]	应该为 0
位 [31:16]	表示这个条目映射的内存空间的物理地址，由于大页是 64KB 大小，所以地址的低 16 位始终为 0，正因为如此，这个条目的低 16 位才可以用于访问属性控制等

下面我们来看大页映射下 MMU 是如何用内存中的两级页表实现虚拟地址到物理地址的转换的，如图 3-42 所示。

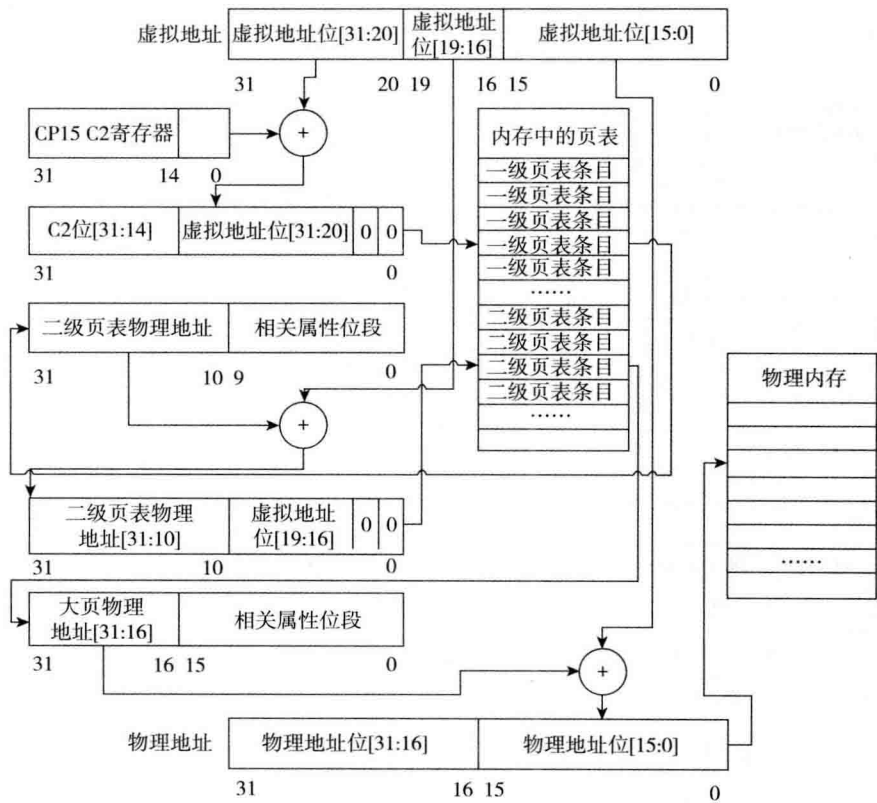


图 3-42 大页映射下虚拟地址到物理地址的转换

图 3-42 中，MMU 的执行步骤如下。

第 1 步：MMU 拿到虚拟地址。

第 2 步：MMU 将 CP15 C2 寄存器中的高位 [31:14] 和虚拟地址的高位 [31:20] 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址。这个地址就是指向一级页表条目的地址。

第 3 步：用第 2 步中合成的地址读取一个一级页表条目。

第 4 步：用一级页表条目中的高位 [31:10] 与虚拟地址的低位 [19:16] 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址。这个地址就是指向二级页表条目的地址。

第 5 步：用第 4 步中合成的地址读取一个二级页表条目。

第 6 步：用二级页表条目中的高位 [31:16] 与虚拟地址的低位 [15:0] 一起组成 32 位的物理地址。

第 7 步：最后用第 6 步中组成 32 位的物理地址去访问内存。

我们来看一个例子，如图 3-43 所示。

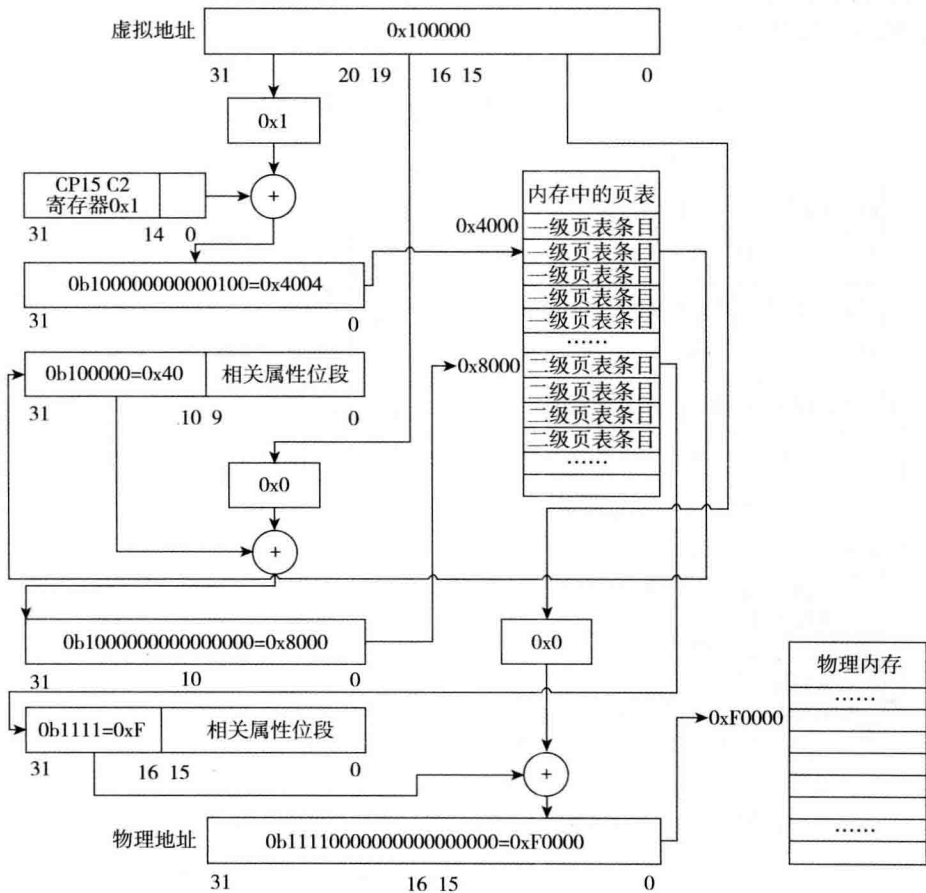


图 3-43 大页映射下虚拟地址到物理地址的转换实例

这个例子演示了从虚拟地址 0x100000 到物理地址 0xF0000 的映射过程。这个过程如下。

第 1 步：MMU 拿到虚拟地址 0x100000。

第 2 步：MMU 将 CP15 C2 寄存器中的高位 [31:14] (0x1) 和虚拟地址的高位 [31:20] (0x1) 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址 0x4004。这个地址就是指向一级页表条目的地址。

第 3 步：用第 2 步中合成的地址读取一个一级页表条目。

第 4 步：用一级页表条目中的高位 [31:10] (0x40) 与虚拟地址的低位 [19:16] (0x0) 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址 0x8000。这个地址就是指向二级页表条目的地址。

第 5 步：用第 4 步中合成的地址读取一个二级页表条目。

第 6 步：用二级页表条目中的高位 [31:16] (0xF) 与虚拟地址的低位 [15:0] (0x0) 一起组成 32 位的物理地址 0xF0000。

第 7 步：最后用第 6 步中组成 32 位的物理地址 0xF0000 去访问内存。

3. 小页

小页映射下，ARM920T 的 MMU 把虚拟地址空间和物理地址空间分成 4KB 大小的地址空间块，即每个页面大小为 4KB。由于是 4KB 大小的页面，所以需要两级页表。一级页表中有 4096 个条目。每个条目映射二级页表的空间，二级页表中有 256 个条目，每个二级页表条目可以映射 4KB 大小的地址空间，小页映射下一级页表条目和二级条目的格式如图 3-44 所示。

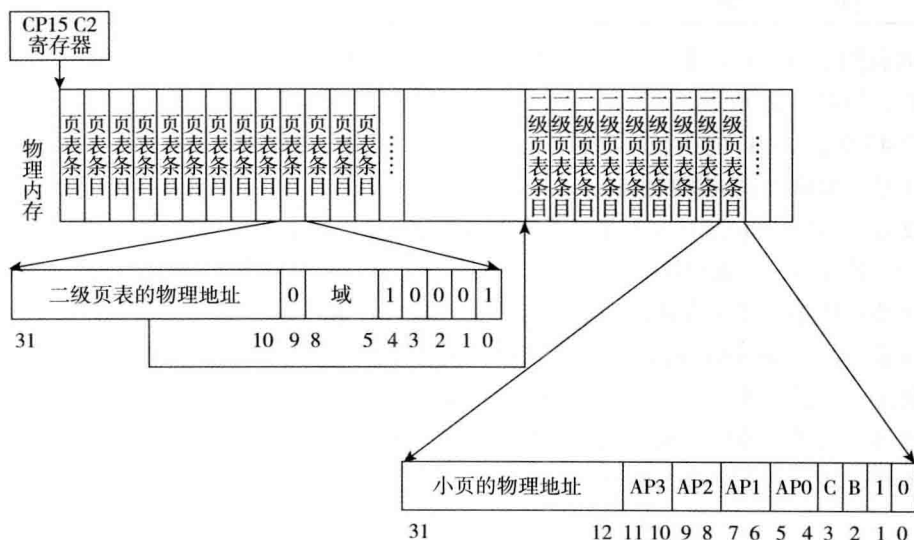


图 3-44 小页映射下一级页表条目和二级条目的格式

图 3-44 中的小页的一级页表条目中的各位段如表 3-22 所示。

表3-22 小页的一级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识, 对于小页可以是二进制 01 也可以是 11。11 下情况有所不同, 这里以 01 为准
位 [3:2]	应该为 0
位 [4:4]	为 1、0 都行, 但为了向后兼容应该为 1
位 [8:5]	表示这个条目映射的内存空间, 属于哪个域, 参见 c3 寄存器
位 [9:9]	应该为 0
位 [31:10]	指向二级页表的物理地址, 二级页表地址是 1KB 对齐的, 所以地址的低 10 位始终为 0, 正因为如此, 这个条目的低 10 位才可以用于访问属性控制等

图 3-44 中的小页的二级页表条目中的各位段如表 3-23 所示。

表3-23 小页的二级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识, 对于小页必须是二进制 10
位 [3:2]	表示这个条目映射的内存空间, 是否为回写缓存、写通缓存、非缓存、缓冲, 或者无缓存并且无缓冲
位 [11:4]	一个小页可以分为 4 个子页, 每个子页 1KB。AP0: 控制第 1 个子页的访问权限; AP1: 控制第 2 个子页的访问权限; AP2: 控制第 3 个子页的访问权限; AP3: 控制第 4 个子页的访问权限
位 [31:12]	表示这个条目映射的内存空间的物理地址, 由于小页是 4KB 大小, 所以地址的低 12 位始终为 0, 正因为如此, 这个条目的低 12 位才可以用于访问属性控制等

下面我们来看小页映射下, MMU 是如何用内存中的两级页表实现虚拟地址到物理地址的转换的, 如图 3-45 所示。

图 3-45 中, MMU 的执行步骤如下。

第 1 步: MMU 拿到虚拟地址。

第 2 步: MMU 将 CP15 C2 寄存器中的高位 [31:14] 和虚拟地址的高位 [31:20], 外加两个为 0 的二进制位, 一起组成一个 32 位的地址。这个地址就是指向一级页表条目的地址。

第 3 步: 用第 2 步中合成的地址读取一个一级页表条目。

第 4 步: 用一级页表条目中的高位 [31:10] 与虚拟地址的低位 [19:12] 并且外加两个为 0 的二进制位, 一起组成一个 32 位的地址。这个地址就是指向二级页表条目的地址。

第 5 步: 用第 4 步中合成的地址读取一个二级页表条目。

第 6 步: 用二级页表条目中的高位 [31:12] 与虚拟地址的低位 [11:0] 一起组成 32 位的物理地址。

第 7 步: 最后用第 6 步中组成 32 位的物理地址去访问内存。

我们来看一个例子, 如图 3-46 所示。

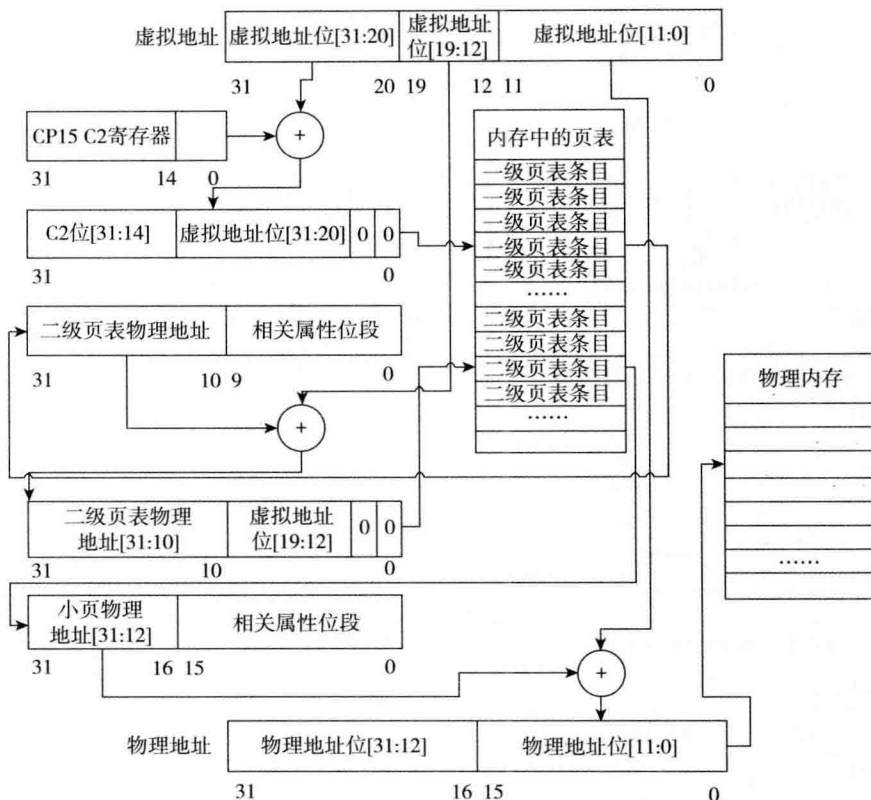


图 3-45 小页映射下虚拟地址到物理地址的转换

这个例子演示了从虚拟地址 0x100040 到物理地址：0x40 的映射过程。这个过程如下。

第 1 步：MMU 拿到虚拟地址 0x100040。

第 2 步：MMU 将 CP15 C2 寄存器中的高位 [31:14] (0x1) 和虚拟地址中的高位 [31:20] (0x1) 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址 0x4004。这个地址就是指向一级页表条目的地址。

第 3 步：用第 2 步中合成的地址 0x4004 读取一个一级页表条目。

第 4 步：用一级页表条目中的高位 [31:10] (0x40) 与虚拟地址中的低位 [19:12] (0x0) 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址 0x8000。这个地址就是指向二级页表条目的地址。

第 5 步：用第 4 步中合成的地址 0x8000 读取一个二级页表条目。

第 6 步：用二级页表条目中的高位 [31:12] (0x0) 与虚拟地址位 [11:0] (0x40) 一起组成 32 位的物理地址 0x40。

第 7 步：最后用第 6 步中组成 32 位的物理地址 0x40 去访问内存。

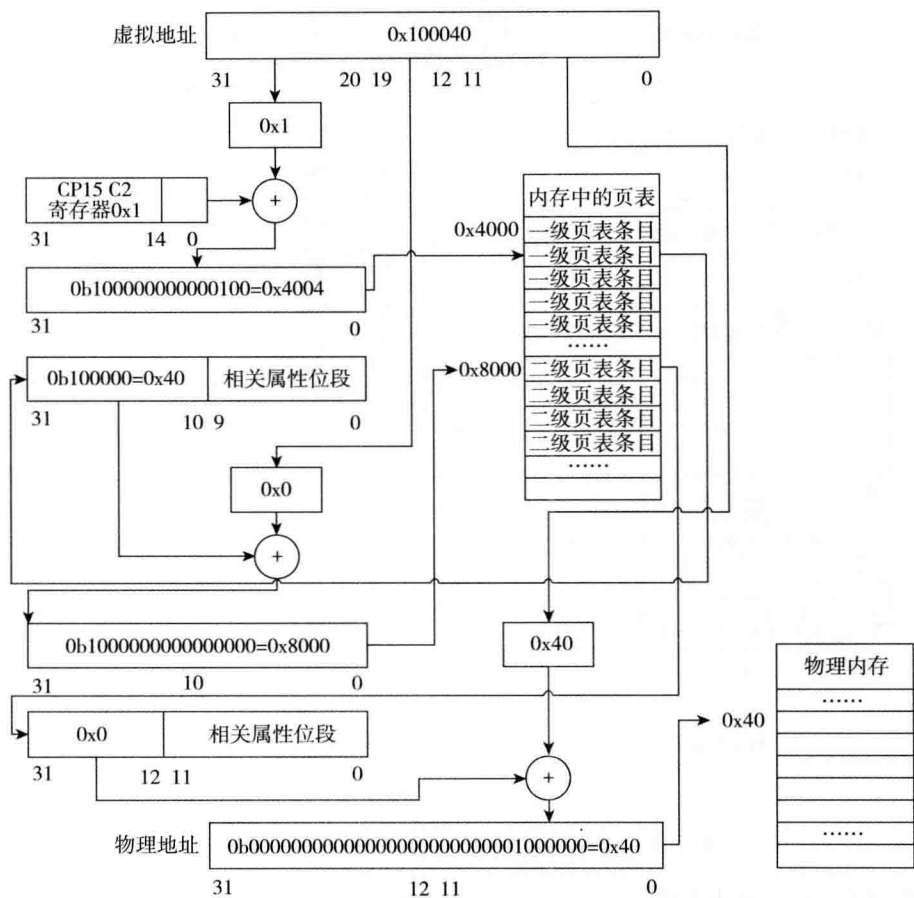


图 3-46 小页映射下虚拟地址到物理地址的转换实例

4. 极小页

极小页映射下，ARM920T 的 MMU 把虚拟地址空间和物理地址空间分成 1KB 大小的地址空间块，即每个页面大小为 1KB。由于是 1KB 大小的页面，所以需要两级页表。一级页表中有 4096 个条目。每个条目映射二级页表的空间，二级页表中有 1024 个条目，每个二级页表条目可以映射 1KB 大小的地址空间。极小页映射下一级页表条目和二级条目的格式如图 3-47 所示。

图 3-47 中的极小页的一级页表条目中的各位段如表 3-24 所示。

表3-24 极小页的一级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识，对于极小页是二进制 11
位 [3:2]	应该为 0

(续)

条 目 位 段	相 关 说 明
位 [4:4]	为 1、0 都行，但为了向后兼容应该为 1
位 [8:5]	表示这个条目映射的内存空间，属于哪个域，参见 c3 寄存器
位 [11:9]	应该为 0
位 [31:12]	指向二级页表的物理地址，极小页的二级页表地址是 4KB 对齐的，所以地址的低 12 位始终为 0，正因为如此，这个条目的低 12 位才可以用于访问属性控制等

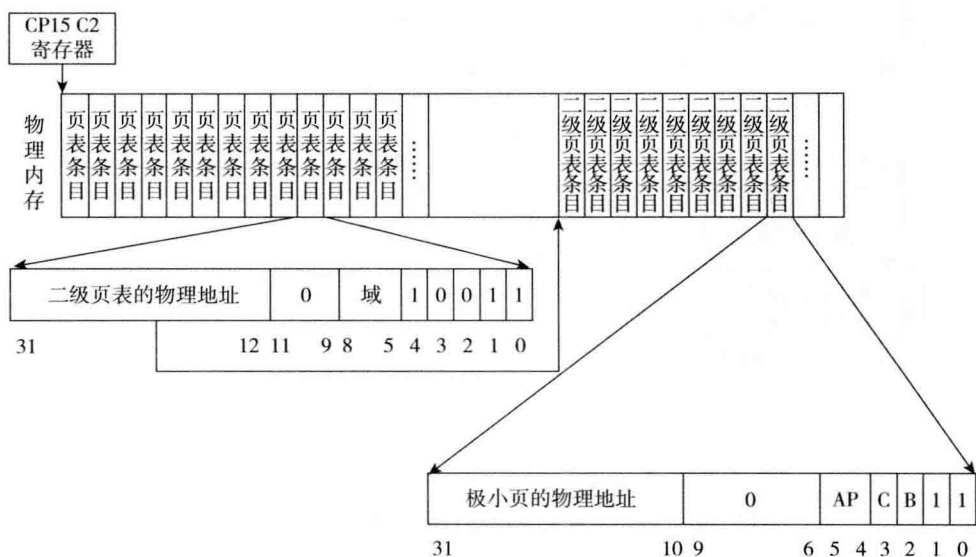


图 3-47 极小页映射下一级页表条目和二级条目的格式

图 3-47 中的极小页的二级页表条目中的各位段如表 3-25 所示。

表 3-25 极小页的二级页表条目各位段

条 目 位 段	相 关 说 明
位 [1:0]	页表条目类型标识，对于小页必须是二进制 11
位 [3:2]	表示这个条目映射的内存空间，是否为回写缓存、写通缓存、非缓存、缓冲，或者无缓存并且无缓冲
位 [5:4]	AP：控制这个页的访问权限。极小页没有子页
位 [9:6]	应该为 0
位 [31:10]	表示这个条目映射的内存空间的物理地址，由于极小页是 1KB 大小，所以地址的低 10 位始终为 0，正因为如此，这个条目的低 10 位才可以用于访问属性控制等

下面我们来看极小页映射下 MMU 是如何用内存中的两级页表实现虚拟地址到物理地址的转换的，如图 3-48 所示。

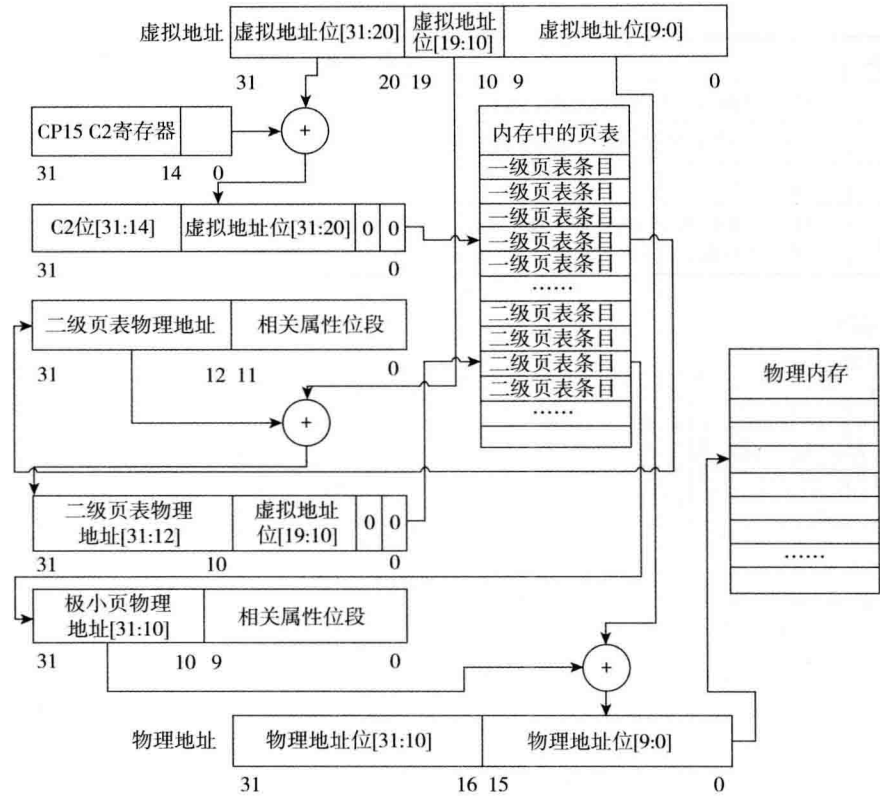


图 3-48 极小页映射下虚拟地址到物理地址的转换

图 3-48 中，MMU 的执行步骤如下。

第 1 步：MMU 拿到虚拟地址。

第 2 步：MMU 将 CP15 C2 寄存器中的高位 [31:14] 和虚拟地址中的高位 [31:20] 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址。这个地址就是指向一级页表条目的地址。

第 3 步：用第 2 步中合成的地址读取一个一级页表条目。

第 4 步：用一级页表条目中的高位 [31:12] 与虚拟地址的低位 [19:10] 并且外加两个为 0 的二进制位，一起组成一个 32 位的地址。这个地址就是指向二级页表条目的地址。

第 5 步：用第 4 步中合成的地址读取一个二级页表条目。

第 6 步：用二级页表条目中的高位 [31:10] 与虚拟地址的低位 [9:0] 一起组成 32 位的物理地址。

第 7 步：最后用第 6 步中组成 32 位的物理地址去访问内存。

我们来看一个例子，如图 3-49 所示。

这个例子演示了从虚拟地址 0x400800 到物理地址 0xFFFFFC00 的映射过程。这个过程如下。

第 7 步：最后用第 6 步中组成 32 位的物理地址 0xFFFFFC00 去访问内存。

前面介绍的 MMU 的 4 种分页方式，在同一时刻都是使用的同一种分页方式。例如，要么全部使用大页，要么全部使用小页，或者全部使用段，其实 ARM920T 的 MMU 设计得非常优秀，它不仅可以同时使用一种分页方式，还可以同时把 4 种分页方式混合起来一起使用，如图 3-50 所示。

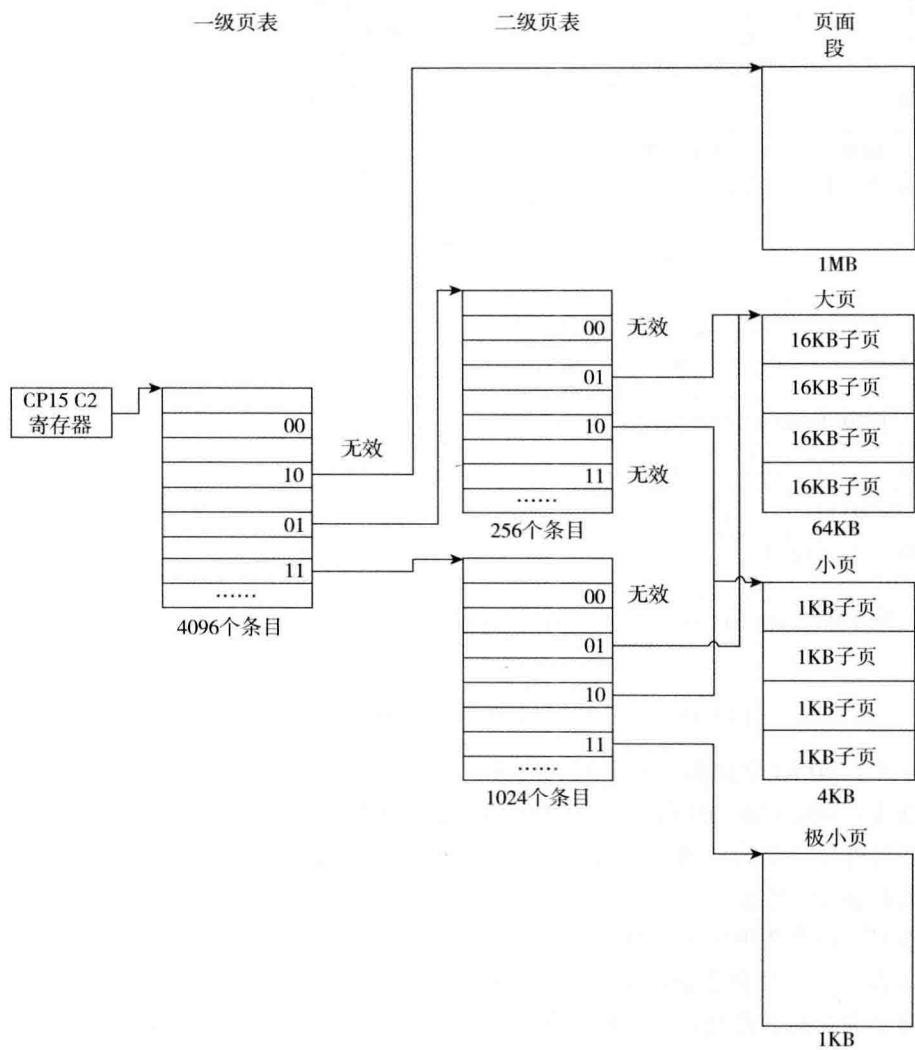


图 3-50 4 种分页方式混用

我们看到，MMU 正是用页表条目的最低两位来区分当前使用的是哪种分页方式。注意：混合使用时要注意虚拟地址空间与分页的大小以及 MMU 所要求的空间块基地址的对齐标准。

ARM920T 处理器的 MMU 支持的 4 种分页方式介绍完了，下面去看看 MMU 是如何控制物理页面的访问权限的。

3.9.5 MMU 页面访问权限的控制

ARM920T 处理器的 MMU 页面访问权限控制非常简单：它是用 CP15 的 c1 寄存器中的 S、R 位，联合页表条目的 AP 位一起工作的。它是根据 AP、S、R 中的数据进行判断的，如表 3-26 所示。

表3-26 c1寄存器S、R位及页表条目AP位的取值

c1 S 位	c1 R 位	AP 位	处于特权级时的访问权限	处于用户级时的访问权限
0b0	0b0	0b00	没有访问权限，进行访问将导致 MMU 发出访存失败的信号	没有访问权限，进行访问将导致 MMU 发出访存失败的信号
0b1	0b0	0b00	只读，但不可写，写入将导致 MMU 发出访存失败的信号	没有访问权限，进行访问将导致 MMU 发出访存失败的信号
0b0	0b1	0b00	只读，但不可写，写入将导致 MMU 发出访存失败的信号	只读，但不可写，写入将导致 MMU 发出访存失败的信号
0b1	0b1	0b00	保留	保留
0bx	0bx	0b01	可读 / 写	没有访问权限，进行访问将导致 MMU 发出访存失败的信号
0bx	0bx	0b10	可读 / 写	只读，但不可写，写入将导致 MMU 发出访存失败的信号
0bx	0bx	0b11	可读 / 写	可读 / 写

注：0b 表示紧跟其后的是二进制数据。“x”表示未定义，可以是 0 也可以是 1。

正是因为 MMU 可以通过表 3-26 对访问存储器的地址进行检查，我们才可以利用这种特性，对操作系统内核的内存空间或者各个进程的内存空间进行保护，以防止其他程序的恶意访问。也正是因为这样，才使构建现代健壮的操作系统成为可能。

3.9.6 MMU 的快表 TLB

从前面关于页表的介绍，我们看到了一个虚拟地址到物理地址的转换过程，其实这个过程就是查询页表的过程。

我们的页表是放在内存里的，查询一次页表 MMU 最少要访问一次内存，有时要访问多次。可是人类今天的技术、成本，导致内存的速度和 CPU 的速度相比还是很慢。本来查询页表就是为了访问内存的，可是查询页表就需要巨大的时间代价。

怎么解决上述问题呢，人们基于这样的观察：程序的运行大多数总是在访问当前地址附近的内存地址，程序中的循环代码更是体现了这一点。于是人们把当前页表附近的页表装进 MMU 内部的一个小且高速的存储器中，这个寄存器就叫 TLB，也叫快表，它的速度和

CPU 中的寄存器差不多, 大小有几十 KB。并且把页表装载进 TLB 的过程是 MMU 自动完成的, 不需要人为编程操作。基于这种情况, 在特定的一段时间内, MMU 总是可以从 TLB 中获取页表内容, 这样地址的转换速度在那一段时间内就上去了, 进而程序的执行速度也上去了。

从前面的描述中, 看到 MMU 中的 TLB 确实是个好东西, 然而需要操作系统开发人员时刻注意的问题出现了。那就是当我们修改内存中的页表内容以达到改变地址映射规则时, 假如我们修改的这部分内容在修改之前, 已经被 MMU 装进 TLB 中了, 这时 TLB 中和内存中的内容就不一致了, 从而不能达到改变地址映射规则的目的。这个问题非常危险。所以我们要能通知 MMU, 让它把 TLB 中的内容或者一部分内容变得无效, 重新从内存中装载最新的内容。ARM920T 的 MMU 当然提供了这个操作接口, 我们只要操作 CP15 的 c8 寄存器就行了, 操作细节, 请参见 c8 寄存器的介绍。

相反, 有时为了达到系统的最高性能, 把一些经常访问的数据或者经常运行的程序的地址对应页表的内容, 锁定在 MMU 的 TLB 中, 使得这些页表的内容长期存在于 MMU 的 TLB 中, 这样 MMU 就能总是在 TLB 中找到这些页表的内容, 从而最大限度地加快地址的转换速度。ARM920T 的 MMU 当然也提供了这个操作接口, 我们只要操作 CP15 的 c10 寄存器就行了, 操作细节, 请参 c10 寄存器的介绍。

对于 MMU 的 TLB, 只要明白它是什么、干什么的、要注意什么问题, 以及能利用它的什么特性, 达到提高程序运行效能的目的就行了。

3.9.7 MMU 的编程接口

MMU 的编程接口很简单, 不过就是操作 CP15 协处理器中的几个寄存器而已, 其实硬件提供给软件的接口就是一堆寄存器。先在此简单提一下, 等到我们操作系统初始化时, 要操作 MMU 时再用代码加于实现。在这里只是用文字大致描述编程 MMU 的过程。

大致过程和要操作的寄存器如下。

第 1 步: 选择一块开始地址以 16KB 对齐的物理内存, 在其中初始化一级页表或者根据需要可能还要初始化二级页表, 填写好每个页表条目的内容, 如访问权限、映射的物理地址等。

第 2 步: 把第 1 步中的一级页表的开始物理地址写入 CP15 的 c2 寄存器中。根据需要可能要操控 CP15 的 c3 寄存器、CP15 的 c13 寄存器。

第 3 步: 把 CP15 的 c1 寄存器的 M 位即第 0 位, 置为 1, 即打开了 ARM920T 的 MMU。

注意, 上述操作有一定的顺序, 不能在还没有页表时就打开了 MMU。如果这样, 系统要么崩溃, 要么执行结果不可预知。

确实很简单, MMU 的编程接口就先说到这里吧, 不是很清楚也没关系, 等到写代码时, 就自然明白了。

到这里我们了解的内容大致如下。

- ❑ MMU 是什么，没有它会有什么问题。
 - ❑ 接着介绍 ARM920T 处理器中的 CP15 协处理器。并详细了解了它的一些寄存器，还知道它就是用来控制 ARM920T 中的 MMU 和 Cache 的。
 - ❑ 我们又介绍了 ARM920T 中 MMU 的地址转换机理以及 MMU 的页表。
 - ❑ 然后了解了 ARM920T 中 MMU 如何利用 CP15 的 c1 寄存器中的 S、R 位和页表条目中的 AP 位，控制物理页面的访问权限。
 - ❑ 粗略了解了 MMU 中的 TLB。
 - ❑ 最后大致了解了如何开启 MMU 和开启 MMU 的前提条件。
- 对于 MMU 我们知道了这些就够了。

3.10 Cache

Cache 是一种比常见的内存 (RAM) 更快的一种存储器。但它一般情况下比内存小很多，所以被称为高速缓冲存储器。下面首先概述 ARM920T 中的 Cache，接着了解 Cache 的原理，然后看一看 Cache 的类型以及要注意的问题，最后了解 Cache 的编程接口。

3.10.1 ARM920T 的 Cache

Cache 即高速缓冲存储器，也是 ARM920T 存储系统中的重要组成部分，它用来缓存一小部分内存中的内容。

在前面的介绍中，我们都假定系统中没有 Cache 或者 Cache 处于关闭状态。有些简单的嵌入式系统中确实没有 Cache。但我们用的 ARM920T 处理器中提供了 Cache 组件。下面来看看加入或者打开 Cache 后，ARM920T 的存储系统是什么样的，如图 3-51 所示。当然笔者画的图可能与实际芯片中有差异。因为这里只需要知道 ARM920T 处理器的存储系统是由哪些部件构成即可。

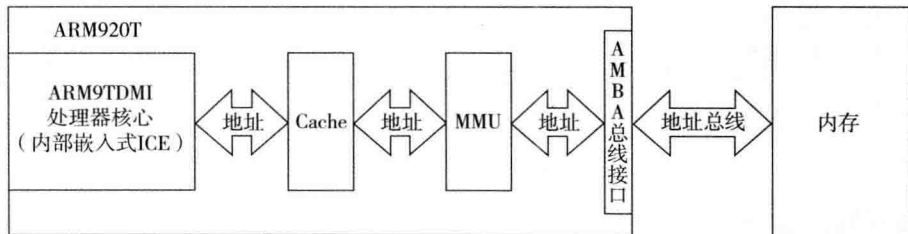


图 3-51 ARM920T 开启 Cache 和 MMU 的情况

ARM920T 处理器没有使用统一的 Cache，它使用了指令 Cache 和数据 Cache 分开的形势，图 3-51 中没有画出来，如果有需要请参见图 3-22。下面先来看看 Cache 的工作机理。

果 CPU 是写内存操作, 根据 Cache 的类型不同 Cache 的动作也不同。

第 5 步: 如果第 3 步中没有在 Cache 内找到对应的 Cache 块, 即表示未命中。

第 6 步: 如果 Cache 未命中, Cache 首先查找 Cache 内部有没有空闲块。

第 7 步: 如果第 6 步中 Cache 找到一个空闲块, 就在该块中装入 CPU 访问内存地址对应的内存块, 同时, 如果 CPU 是读内存操作就把这个地址对应的数据返回给 CPU。如果 CPU 是写内存操作, 根据 Cache 的类型不同 Cache 的动作也不同, 后面会介绍的。

第 8 步: 如果第 6 步中 Cache 没有找到一个空闲块, 就让 Cache 块替换逻辑, 找出 Cache 中最值得替换出去的块。并且有相关的替换算法, 只是这种算法是硬件实现的。如果 CPU 是读内存操作, 那么根据替换块的块号和状态, Cache 会决定是否把这个块回写到内存中, 或者直接废除, 最后在该替换出去的块中装入 CPU 访问内存地址对应的内存块, 同时把这个地址对应的数据返回给 CPU。如果 CPU 是写内存操作, 根据 Cache 的类型不同 Cache 的动作也不同。

由于前面介绍的程序局部性原理, CPU 在某一特定的时间段内会对 Cache 保持很高的命中次数。这样, 在那一段时间内, CPU 就可以直接从 Cache 中获取指令或者数据, 而 Cache 的速度远远高于内存, 所以这样就用合理的成本, 达到了很高的系统性能。

3.10.3 Cache 的类型及要注意的问题

根据 Cache 的工作机制, 可以把 Cache 分成很多类型, 我们主要看看下面两种类型, 在 ARM920T 中知道这两种就够了。当然了解这两种类型是为了要说明后面要注意的问题, “要注意的问题”才是重要的。

□ 回写式 Cache。

□ 写通式 Cache。

1. 回写式 Cache

当 CPU 执行写数据操作时, 回写式 Cache 只把该数据写入其数据地址对应的 Cache 块中, 不直接写入内存。仅当该 Cache 块需要替换时, 才把该 Cache 块回写入内存中。回写式 Cache, 每个 Cache 块中都有个对应的修改位。只要该 Cache 块中的任何单元被修改, 该位就为 1, 否则该位为 0。当该 Cache 块需要替换时, 如果其修改位为 1, 那么就必须先将该 Cache 块写入内存, 如果其修改位为 0, 就可以直接用新的内存块覆盖该 Cache 块即可。回写式 Cache 的稳定性不太高, 因为它不能实时地保证内存中有 Cache 中的数据最新副本。回写式 Cache 和内存的通信量很少, 因为 Cache 在特定的时间段内有很高的命中率, CPU 访问的是 Cache, 只有在发生 Cache 块替换时才访问内存。回写式 Cache 硬件的实现较为简单。

2. 写通式 Cache

当 CPU 执行写数据操作时, 写通式 Cache 必须同时把该数据写入其数据地址对应的 Cache 块和内存中。写通式 Cache 的每个 Cache 块中不需要有对应的修改位。当该 Cache 块

需要替换时,也不必把该 Cache 块写入内存中。新的内存块可以直接覆盖该 Cache 块。因为写通式 Cache 能始终保持 Cache 中的数据 and 内存中的数据是一致的。写通式 Cache 的稳定性很高,因为它始终实时的保证内存中有 Cache 中的数据 of 数据的最新副本。这同时也增加了 Cache 和内存的通量。写通式 Cache 硬件的实现也复杂很多。

采用 Cache 后要注意什么问题呢?说到底就是数据的一致性问题。

如果硬件系统中有多核心的 CPU,每个 CPU 核心内部都有独立的 Cache。这时如果内存中有个全局数据 A 且 $A=0$ 。如果两个 CPU 核都对其执行加 1 操作。如果按照下面的执行顺序,就会出现问。而事实上如果不加控制我们无法对这种硬件的执行顺序进行估计,很有可能就发生下面的执行顺序。

1) CPU0 读取内存中 A 的数据到 R0 寄存器,并且缓存 A 的数据到 CPU0 的 Cache 中。CPU0 的 R0 寄存器中为 0, CPU0 的 Cache 中 A 数据对应的 Cache 块中的单元为 0。

2) CPU0 对它的 R0 寄存器执行加 1 操作,并且把结果写入 CPU0 的 Cache 中,假定这个 Cache 是回写式 Cache, CPU0 的 Cache 中 A 数据对应的 Cache 块中的单元为 1。CPU0 接着执行下一条指令。

3) CPU1 读取内存中 A 的数据到 R1 寄存器,并且缓存 A 的数据到 CPU1 的 Cache 中。由于第 2) 步中 CPU0 的 Cache 中的 A 数据没有回写到内存中,内存中 A 数据没有改变。所以 CPU1 的 R1 寄存器中为 0, CPU1 的 Cache 中 A 数据对应的 Cache 块中的单元为 0。

4) CPU1 对它的 R1 寄存器执行加 1 操作,并且把结果写入 CPU1 的 Cache 中,假定这个 Cache 是回写式 Cache, CPU1 的 Cache 中 A 数据对应的 Cache 块中的单元为 1。CPU1 接着执行下一条指令。

这时不管哪个 CPU 的 Cache 中 A 数据对应的 Cache 块发生了替换,内存中数据 A 为 1。

我们看到,这个执行顺序下,明明对 A 数据 ($A=0$) 执行了两次加 1 动作,可是结果还是为 1。如果这个 A 数据是个局部数据,或许这不是个严重的问题。可是这个 A 数据是个全局数据,更有可能 A 数据是一个信号量数据结构中的信号值变量,你会发现这个问题会有多么重要。对于开发操作系统的人员,这种问题太可怕了,要时刻注意避免类似问题的发生。

上面这个例子只是从逻辑上说明了 Cache 的数据一致性问题是多么重要,然而 ARM920T 不是多核心的 CPU。现在来看 ARM920T 系统中一个非常实际的问题。

S3C2440A 处理器提供了丰富的设备,如 RTC、串口等。这些设备提供了一系列的寄存器作为编程接口。要操作控制这些设备,就是读写这些设备的相关寄存器。但是要访问这些寄存器就要知道这些寄存器的地址。ARM920T 没有提供用于专门访问设备寄存器的指令。硬件设计人员想了个办法,他们把这些设备的寄存器映射到存储器地址空间中。访问特定的存储器单元就是访问某个设备的某个特定的寄存器。例如,访问“0x57000040”这个地址的存储单元就是访问 RTC 的控制寄存器,访问“0x50000004”这个地址的存储单元就是访问

第一个串口设备的控制寄存器等。

在打开 Cache 的情况下，如果读取 RTC 寄存器中的状态数据时，很有可能读取的不是 RTC 寄存器的最新数据而是读取缓存在 Cache 中的数据。这会让我们获取的数据不准确，有时会产生致命的错误。如果写入数据到 RTC 寄存器中对 RTC 设备进行控制。这个数据会首先写入 Cache 中（如果是回写式 Cache）。我们满心以为 RTC 设备已经得到了控制，然而我们错了，这时控制数据也许还在 Cache 中，还没有到 RTC 设备控制寄存器中去。当然其他设备也是一样。这种问题非常危险，而且这种类型的错误根本就不是由程序代码本身的问题导致的，查起来也不是一般的困难。

解决上述问题的最好方法是，对映射设备寄存器的存储器地址空间，不执行 Cache 缓存。在 MMU 的章节中，了解到每个页表条目中有 C、B 两个位。这两个位就是控制这个页表条目映射的地址空间，是否被 Cache 部件（C 位）或者写缓冲部件（B 位，写缓冲也是一种小型的 Cache，但是不详细研究它）缓存。只要把设备寄存器所在的页面的页表条目中的 C、B 位设置为 0，就解决上述问题了。

Cache 技术带来了系统性能的提升，同时也带来了不小的麻烦，这就像世间其他事物一样，有优点就有不足。

3.10.4 ARM920T Cache 的编程接口

Cache 实现的功能大部分对程序员是透明的，因为它的功能逻辑几乎全部是用硬件实现的。所以程序员只需要少量的控制操作即可。

ARM920T 处理器 Cache 的操作很简单，综合起来无非就是以下三点。

- ☐ 打开或者关闭 Cache。
- ☐ 清空 Cache。
- ☐ 锁定 Cache。

1. 打开或者关闭 Cache

ARM920T 处理器使用了独立的数据 Cache 和指令 Cache，它们可以独立关闭或者开启。把 CP15 的 c2 寄存器的 C 位设置为 1 就开启了 ARM920T 的数据 Cache，否则即为关闭 ARM920T 的数据 Cache；把 CP15 的 c2 寄存器的 I 位设置为 1 就开启了 ARM920T 的指令 Cache，否则即为关闭 ARM920T 的指令 Cache。打开 Cache 后，还可以通过设置 CP15 的 c2 寄存器的 RR 位为 1，从而改变 Cache 替换 Cache 块的算法。具体操作请参阅 AM920T CP15 协处理器相关的章节。

2. 清空 Cache

清空 Cache，实际上是强制 Cache 把其中已修改的数据块回写到内存中去。基于前面所述 Cache 的问题，有时修改了重要数据之后必须要手动清空 Cache，使内存中的数据得到更新。ARM920T 处理器中可以分别清空指令 Cache 或者数据 Cache。可以向 CP15

的 c7 寄存器中写入虚拟地址或者组号（许多块组成一个组），以清空特定的 Cache 块，也可以清空整个 Cache。操作时相关的参数和数据请参阅 AM920T CP15 协处理器相关的章节。

3. 锁定 Cache

锁定 Cache 就是让 Cache 在替换 Cache 块时，不把被锁定的 Cache 块替换出去。例如，把一些运行密度非常高的指令或者数据所在的内存块，锁定到 Cache 块中，这样就可以大大提升系统的性能。ARM920T 可以分别锁定指令或者数据 Cache 中的块。可以向 CP15 的 c9 寄存器中写入组内序号，以锁定特定的 Cache 块，操作时相关的参数和数据请参阅 AM920T CP15 协处理器相关的章节。

通过上述 Cache 的编程接口，合理利用 Cache 的特性，既可以避免 Cache 带来的问题又可以使系统性能提高。

3.11 小结

本章就要结束了，这意味着我们又走了一大步，同时也考验了我们的耐性。现在回顾一下这章我们了解到了什么。

- ❑ ARM920T 处理器，它是由一个名叫 ARM 的公司设计的，广泛应用于嵌入式领域。
- ❑ 在 ARM920T 存储体系中，了解了 ARM920T 处理器的地址空间、存储器格式及存储地址对齐方式。
- ❑ ARM920T 处理器的运行状态，包括 ARM 状态和 Thumb 状态。
- ❑ ARM920T 处理器的 7 种工作模式，并且这 7 种工作模式可以分为特权模式和用户模式。
- ❑ ARM920T 处理器共有 37 个寄存器。这些寄存器按照不同的工作模式分成了几组，并且有些寄存器在不同工作模式下有不同的物理寄存器。
- ❑ ARM920T 处理器的中断和异常。这些中断和异常有固定的向量地址。
- ❑ ARM920T 处理器的指令集。一共了解了 35 条常用的指令及使用方式，这些指令可以分类为分支跳转指令、数据处理指令、装载和存储指令、程序状态寄存器操作指令、协处理器操作指令、异常中断产生指令。
- ❑ ARM920T 处理器内部的 MMU，了解了为什么要有 MMU、用于控制 MMU 的 CP15 协处理器、MMU 实施地址转换的页表、MMU 如何控制内存的访问权限、MMU 用于加快地址转换速度的 TLB 以及如何开启 MMU。
- ❑ ARM920T 处理器内部的 Cache，了解了 Cache 的工作原理，利用它可以大大提高系统的性能，同时也因为它的特性给程序的正确运行带来了很严重的问题，但是可以通过 Cache 的一些编程接口对其进行控制。

或许这内容不少，但这是为了今后我们写操作系统内核的第一行代码以及操作系统内核在这个平台上稳定高效地运行而准备的，所以要勇敢了解这个平台上硬件的一些细节。有了这些硬件基础，后面就可以一边写代码一边关注操作系统要用到的某些具体的硬件设备。

操作系统内核的设计与构建

前面了解了一大堆硬件平台相关的东西，说实话那些东西很枯燥、很无趣，但是为了更加轻松地写操作系统内核，又不得不了解那些东西。

是不是走到这里就可以开始写操作系统内核了，当然不是的。看了这句话，你的兴趣可能掉了一大半，别急，相信花点时间了解下面这些东西一定是值得的。

操作系统内核是一个十分巨大的软件工程项目，了解如何设计、管理、编译、测试这个项目，十分重要。

这里将从操作系统内核需要完成什么功能开始，分析各种操作系统内核的架构，然后在这个架构上组建各种功能模块，选择适合我们自己操作系统内核的设计，最后研究如何配置系统开发环境、各种工具的使用，以至于最终能高度自动化编译、测试操作系统内核。

4.1 操作系统内核的设计

操作系统内核的设计包括安全、性能、稳定、可移植、可扩展、可维护、BUG 的排查、各功能组件的测试、开发人员的协作等。要详细讨论这些问题，可能一整本书都不够，所以得简单点，因为我们不是操作系统内核架构师，我们只是想做出自己的操作系统内核，过多的理论概念，我们并不喜欢。

首先从内核需要完成的功能开始，了解哪些功能应该由内核实现，哪些功能又不应该由内核实现。接着研究两种成熟的内核架构，主要研究这两种内核架构的优缺点。然后讨论软件分层带来的好处。最后看看我们自己选择设计的架构。

4.1.1 内核要完成的功能

一个操作系统内核必须要有什么功能，没有一个统一的标准来定义，有的系统内核功能很多，如 Windows 系统的 NT 内核，有的系统内核功能非常少，如 minix 系统的内核。

第 1 章通过分析一个小程序执行的过程细节，大概了解了一个现代意义的操作系统内核需要些什么功能，如下：

- 内存管理，因为程序运行要占用内存。
- 进程管理，表示一个运行中的程序。
- 设备管理，系统里有多个设备及它们的状态。
- 文件管理，管理存放用户的数据。

以上是从全局来看需要这 4 大块功能，然而这也只是在第 1 章中，观察现有操作系统上的一个小程序的执行过程大致得出的一个结论。但这不是标准，没有人规定操作系统内核一定要实现这 4 大块功能。

一个软件的功能越多，复杂程度就越大，出现问题的可能性就越高，并且查找问题就越困难。内核是操作系统的灵魂，它一旦有什么问题，整个操作系统必将崩溃。由此可以想到越简单的内核也就越可靠。所以必须将最少的、最重要的、非内核不可的功能交由内核实现，除此之外的，都不应该由内核实现。虽然有许多模块具有内核一样的特权，然而也应该把它们和内核分开。例如，许多驱动程序或者一些功能模块，虽然它们有内核一样的权力，可以操作硬件或者完成特殊的功能，但是它们是以独立的模块存在的。

现在来思考一下哪些功能是需要由内核来实现的，如下：

- 时间管理：时间是自然界最重要的量，对计算机系统也是如此，内核理当实现它。
- 中断管理：中断是内核得以正常工作的重要机制，所以无论软中断还是硬中断，都应该由内核管理和控制。
- 内存管理：内存是所有程序运行的基础，包括内核本身，能否高效分配、释放内存，是系统性能的关键。当然内存管理又分为物理内存管理和虚拟内存管理。
- 进程管理：操作系统都是多任务的，必须要管理多个应用程序，把每个应用程序分配到一个 CPU 上运行。即把每个应用程序抽象成一个进程，并对其行为进行管理和控制。
- 设备管理：内核必须要管理计算机内的设备，并且要提供一个统一的设备模型。方便加载设备驱动程序，同时向应用程序提供访问操作设备的接口。当然内核并不实现设备驱动程序本身。
- 文件系统：主要管理、组织用户的数据，但是由于设备管理的存在，文件系统又可以作为一个虚拟设备加载到系统中运行，并且还可以支持各种不同的文件系统。因此文件系统可以不和内核一起实现。

以上这些功能就是内核要实现的，但是这些功能都还没有细化。例如，进程管理包括怎

样表示一个进程、进程的创建、进程的调度、进程的退出等。

先暂且不要考虑如何细化这些功能，只要在心中知道一个内核需要这么多功能，知道一个操作系统内核大致需要这么多功能。接下来就去研究一下，用什么结构或者架构来组织这么多功能。

4.1.2 内核的架构

操作系统经过 40 多年的发展，人们开发了很多不同的系统架构，它们有的不够成熟，有的非常成熟并且广泛应用于商业操作系统之上。例如，宏内核架构、微内核架构、混合内核架构、外内核架构等。

这里主要介绍两种架构：宏内核架构和微内核架构，混合内核架构也只是宏内核架构的一个简单的变形。其实宏内核架构和微内核架构，业界一直在争论不休，那么它们到底各有什么优势呢？

1. 宏内核

操作系统内核主要是给应用程序提供服务的，如内存管理、I/O 操作等。从前面知道，现代操作系统至少要提供：进程管理、内存管理、设备管理、文件管理等服务功能，宏内核就是把进程管理代码、内存管理代码、设备管理代码、文件管理代码、各种设备驱动程序代码以及其他功能模块的代码经过编译，最后链接在一起，形成一个大的可执行程序。这个大程序里有实现支持这些功能的所有代码，向用户应用软件提供一些接口，这些接口就是常说的系统 API 函数。这个大程序运行在处理器的特权模式下，这个模式通常被称为内核模式。宏内核架构图如图 4-1 所示。

再来看一个例子。宏内核提供内存分配功能的服务过程，如下：

- 1) 应用程序调用内存分配的 API 函数。
- 2) 处理器切换到特权模式，开始运行内核代码。
- 3) 内核里的内存管理代码按照特定的算法，分配一块内存。
- 4) 把分配的内存块的首地址返回给内存分配的 API 函数。
- 5) 内存分配的 API 函数返回，处理器开始运行用户模式下的应用程序，应用程序就得到了一块内存的首地址并可以使用这块内存了。

上面的这个过程和一个实际的操作系统

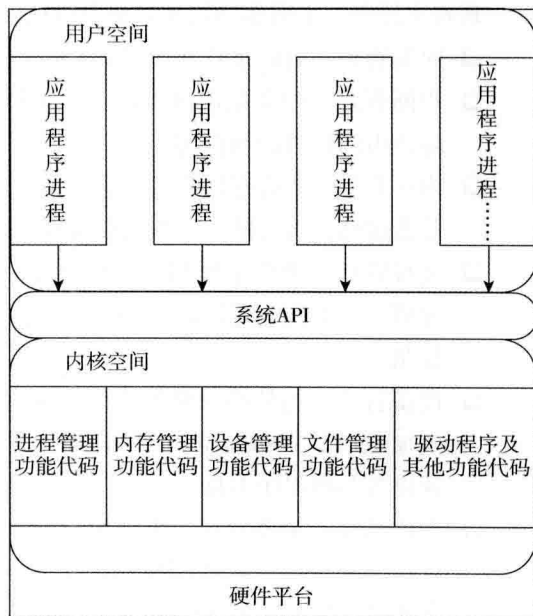


图 4-1 一般性宏内核架构

中的运行过程,可能有差异,但大同小异。当然,系统 API 和应用程序之间可能还有库函数,也可能只是分配了一个虚拟地址空间,但是我们关注的只是这个过程。

从图 4-1 和以上服务过程不难发现,宏内核的代码耦合度非常高,甚至内核内部的功能组件的代码可以互相调用,如文件系统的代码可以直接调用内存管理的代码。但是,由于宏内核代码的高耦合度,若其中有一行代码有问题,那么整个内核将崩溃。经长期证明,只要开发者对宏内核进行精密的实现与测试,宏内核是非常稳定高效的。同时还不难发现,如果要给这个内核增加一个功能,就不得不把整个内核代码全部编译、链接,然后重启计算机启用新版本的内核。

宏内核的经典代表作,就是早期的 UNIX 和 Linux 系统,注意是早期的。现在的 UNIX 和 Linux 微内核架构正好与宏内核架构相反,它提倡内核功能尽可能少:仅仅只有进程调度、处理中断、内存空间映射、进程间通信等功能。这样的内核是不能完成什么实际功能的,开发者把实际的进程管理、内存管理、设备管理、文件管理等服务功能,做成一个个服务进程,和用户应用进程一样,只是它们很特殊,它们是为了专门完成传统宏内核提供的功能的。微内核定义了一种良好的进程间通信的机制——消息。应用程序要请求相关服务,就向微内核发送一条与此服务对应的消息,微内核再把这条消息转发给相关的服务进程,服务进程完成相关的服务,如图 4-2 所示。服务进程的编程模型就是循环处理来自其他进程的消息,完成相关的服务功能。

2. 微内核

微内核提供内存分配功能的服务过程如下:

1) 应用程序发送内存分配的消息,这个发送消息的函数是微内核提供的,相当于系统 API。微内核的 API 相当少,极端情况下仅需要两个,一个接受消息的 API 和一个发送消息的 API。

2) 处理器切换到特权模式,开始运行内核代码。

3) 微内核代码让当前进程停止运行并根据消息包中的数据,确定发送给谁,分配内存的消息当然是发送给内存管理服务进程。

4) 内存管理服务进程收到消息,分配一块内存。

5) 内存管理服务进程,也通过消息的形式返回分配内存块的地址给内核。继

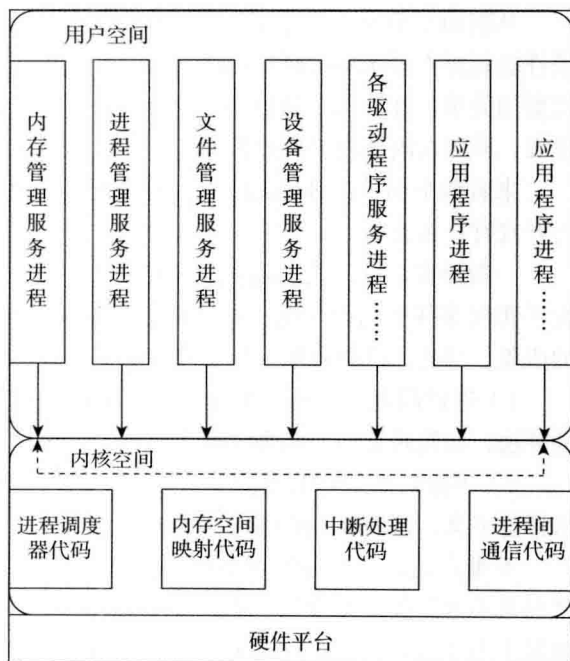


图 4-2 一般性微内核架构

续等待下一条消息。

6) 微内核把包含内存块地址的消息返回给发送内存分配消息的应用程序。

7) 处理器开始运行用户模式下的应用程序, 应用程序就得到了一块内存的首地址并可以使用这块内存了。

微内核的架构实现虽然不同, 但是大致过程和前面一样。同样是分配内存, 在微内核下拐了几个弯, 一来一去的消息带来了非常大的开销, 当然各个服务进程的切换开销也不小。这样, 系统性能就大打折扣。但是微内核有很多优点, 首先, 系统结构相当清晰利于协作开发。其次, 系统有良好的移植性, 微内核代码量非常少, 就算重写整个内核也不是难事。最后, 微内核有相当好的伸缩性、扩展性, 因为那些系统功能只是一个进程, 可以随时拿掉一个服务进程以减少系统功能, 或者增加几个服务进程以增强系统功能。

微内核的代表作有 mach、minix、L4 系统, 这些系统都是微内核, 但是它们不是商业级的系统, 商业级的系统不采用微内核的原因主要还是因为性能。

好了, 粗略地了解了宏内核和微内核两大系统内核架构的优、缺点, 以后设计我们自己的系统内核时, 心里也就有了底, 到时就可以扬长避短了。

4.1.3 分离硬件的相关性

经常听说, Windows 内核有 hal 层、Linux 内核有 arch 层。这些 xx 层就是 Windows 和 Linux 内核设计者, 给他们的系统内核分的第一个层。

从前面章节中不难知道, 今天如此庞杂的计算机, 不过也是一层一层地构建起来的, 从硬件层到操作系统层再到应用软件层, 分层的主要目的和好处在于屏蔽底层细节, 使上层开发更加简单。计算机领域的一个基本方法是增加一个抽象层, 从而使抽象层的上下两层独立发展, 所以在内核内部再分若干层也不足为怪。

来看两个例子: 进程调度、内存分配。通过这两个例子, 来看看分层对系统内核的设计与开发有什么影响。

一般操作系统理论书籍花大量篇幅去写进程相关的概念, 说到底进程是操作系统开发者为了实现多任务而提出的, 并让每个进程在 CPU 上运行一小段时间以实现多任务同时运行的假象。当然, 这种假象十分奏效。要实现这种假象, 就要实现如下两种机制:

1) 进程调度。它的目的是要从众多进程中选择一个将要运行的进程, 当然有各种选择的算法, 如轮转算法、优先级算法等。

2) 进程切换。它的目的是停止当前进程, 运行新的进程, 主要动作是保存当前进程的机器上下文, 装载新进程的机器上下文。

不难发现, 不管是在 ARM 硬件平台上还是在 x86 硬件平台上, 选择一个进程的算法和代码是不容易发生改变的, 需要改变的代码是进程切换的相关代码, 因为不同的硬件平台的机器上下文是不同的。这时最好是将进程切换的代码放在一个独立的层中实现, 如硬件平台相关层, 当操作系统要运行在不同的硬件平台上时, 就只是需要修改硬件平台相关层中的相

关代码。这样，操作系统的移植性就大大增强了。

内存分配也是一样，主要动作如下：

1) 分配一块空闲内存，用相关的内存管理算法，如伙伴系统算法、工作集算法分配一块空闲的物理内存。

2) 进行地址映射，操作硬件平台上的 MMU 和页表，以映射新分配的内存，返回其对应的虚拟地址。

同样，实现伙伴系统算法、工作集算法的代码在不同的硬件平台上是不会发生改变的，当然不同的硬件平台，机器代码是不同的，但是编译器会解决这个问题。要想操作系统运行在不同的硬件平台上，会发生改变的代码是地址映射的相关代码，因为不同的硬件平台上的 MMU 和页表都不尽相同。如果把这段代码放在独立的层中，系统移植起来就会容易许多。

如果把所有硬件平台相关的代码，都抽离出来，放在一个独立硬件相关层中实现并且定义好相关的调用接口。再在这个层之上开发内核的其他功能代码，就会方便很多，结构也会清晰很多，操作系统的移植性也会大大增强。移植到不同的硬件平台就构造开发一个与之对应的硬件相关层。这就是分离硬件相关性的好处。

4.1.4 我们的选择

从前面内容中知道了内核必须要完成的功能，宏内核架构和微内核架构各自的优、缺点，最后分析了分离硬件相关层的重要性，其实说了这么多，就是为了设计我们自己的操作系统内核。虽然前面的内容对操作系统设计这个领域是远远不够的，但是对于我们自己从零开始的操作系统内核已经够了。

首先大致将我们的操作系统内核分为如下三个大层：

- 内核接口层。
- 内核功能层。
- 内核硬件层。

内核接口层定义了一系列接口，主要有如下两点内容：

1) 定义了一套 UNIX 接口的子集，我们出于学习和研究的目的，用 UNIX 接口的子集，一则是接口少，只有几个，并且这几个接口又能大致定义出操作系统的功能；二则用 UNIX 接口也有一定的参考价值。

2) 这套接口的代码，就是检查其参数是否合法，如果参数有问题就返回相关的错误，接着调用下层完成功能的核心代码。

内核功能层，主要完成各种实际功能，这些功能按照其类别可分为各种模块，当然这些功能模块最终会用具体的算法、数据结构、代码去实现它，内核功能层的模块如下：

1) 进程管理，主要是实现进程的创建、销毁、调度进程，这当然要设计几套数据结构用于表示进程和组织进程，还要实现一个简单的进程调度算法。

2) 内存管理，在内核功能层中只有内存池管理，分两种内存池：页面内存池和任意大

小的内存池。可能不明白什么是内存池，后面研究它时会有介绍。

3) 中断管理，这个在内核功能层中非常简单，就是把一个中断回调函数安插到相关的数据结构中，一旦发生相关的中断就会调用这个函数。

4) 设备管理，是最难的，需要用一系列的数据结构表示驱动程序模块、驱动程序本身、驱动程序创建的设备，最后把它们组织在一起，还要实现创建设备、销毁设备、访问设备的代码，这些代码最终会调用设备驱动程序，达到操作设备的目的。

内核硬件层，主要包括一个具体硬件平台相关的代码，如下：

1) 初始化，初始化代码是内核被加载到内存中最先运行的代码，如初始化少量的设备、CPU、内存、中断的控制、内核用于管理的数据结构等。

2) CPU 控制，提供 CPU 模式设定、开、关中断、读写 CPU 特定寄存器等功能的代码。

3) 中断处理，保存中断时机器的上下文，调用中断回调函数，操作中断控制器等。

4) 物理内存管理，提供分配、释放大块内存，内存空间映射，操作 MMU、Cache 等。

5) 平台其他相关的功能，有些硬件平台上有些特殊的功能，需要额外处理一下。

如果上述文字让你看得头晕，我们来画幅图，可能就会好很多，如图 4-3 所示，当然没有画出用户空间的应用进程，API 接口以下的为内核空间，这才是设计、开发内核的重点。

从上述文字和图，可以发现，我们的操作系统内核没有任何设备驱动程序，甚至没有文件系统和网络组件，内核所实现的功能很少，这吸取了微内核的优势。内核小，出问题的可能性就少，扩展性就强，同时把文件系统、网络组件、其他功能组件作为虚拟设备交由设备管理，例如，需要文件系统时就写一个文件系统虚拟设备的驱动，完成文件系统的功能，需要网络时就开发一个网络虚拟设备的驱动，完成网络功能。这些驱动一旦被装载就是内核的一部分了，并不是像微内核一样作为服务进程运行。

这又吸取了宏内核的优势，代码高度耦合，性能强劲。这样的内核架构既不是宏内核架构也不是微内核架构，而是这两种架构综合的结果，可以说是混合内核架构，也可以说这是我们自己的内核架构。

我们设计了内核，确定了内核的功能并且设计了一种内核架构用来组织这些功能，这离完成我们自己的操作系统内核又近了一步。

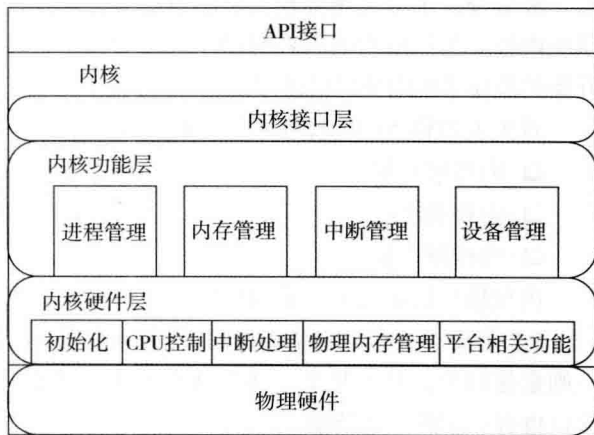


图 4-3 我们的操作系统架构

4.2 开发环境及相关工具

工欲善其事，必先利其器。开发操作系统内核，无非就是编写各个功能模块的代码，接着用编译器编译这些功能模块，然后用链接器把这些功能模块链接成可执行文件即内核，最后在硬件平台上安装运行我们的操作系统内核。这和开发普通的应用软件并无本质上的差别。

从上述开发过程得知，我们最起码需要一个文本编辑器来编写代码，然后需要编译器等工具把代码转换成可执行文件，然而这些工具不过是某一个成熟操作系统下的应用软件而已，所以还需要一个可以运行这些工具的操作系统环境。幸运的是，我们并不是这个世界上第一个开发操作系统和这些工具的人，否则麻烦和困难要大几个数量级。这个世界上已经有 Linux、Windows 并且它们上面已经有了这些工具，我们只需要拿来用就好了。

本节从 Linux 环境开始详细介绍其上开发操作系统内核可能用到的一些工具，并且本书始终用的是 Linux 环境，并不是说这些工作在 Windows 下无法完成，也不是说 Linux 一定比 Windows 优秀，如果硬是要询问原因，笔者只能说是喜好而已，如果你是一个 Windows 狂热者，那么请见谅。

4.2.1 Linux 环境

Linux 是一款开源的类 UNIX 的操作系统内核，1991 年由 Linus Benedict Torvalds 发起的，由于他的理念是开放、自由、平等，迅速引来了世界各地的内核黑客加入 Linux 内核的开发中。这使得 Linux 快速成为一个安全、性能、稳定、功能集一身的成熟的操作系统内核，并且能运行在不同的硬件体系下，从嵌入式到 PC 再到服务器、超级计算机，Linux 内核都可以在上面流畅地运行。关于 Linux 内核就到这里，如果读者想了解更多关于 Linux 的信息，可自行搜索。

可是 Linux 只是一个操作系统内核，它还需要配备一些工具软件、用户应用程序、图形系统等相关组件才能组成一个完整的操作系统。如果你有能力，完全可以下载 Linux 内核代码并选择自己喜欢的应用软件、相关组件，构建一个属于自己唯一的 Linux 操作系统。当然世界上有很多公司已经帮我们做了这个工作，叫作 Linux 发行版，目前世界上已经有上百种 Linux 发行版。其中有非常著名的 Red Hat Linux、Fedora、Debian 等，由 Debian 又派生出了 Ubuntu，这个系统应用得非常广泛。由 Ubuntu 又派生出了 Mint、Deepin 等，由此可见 Linux 的世界非常繁荣。这也是 Linus Benedict Torvalds 始终坚持开放、自由、平等的理念造就的。

笔者所使用的 Linux 是国内的 Deepin，这个发行版出现没有几年，然而已经位居世界 Linux 发行版的前列。这个发行版有一系列的自主原创应用软件，如深度软件中心、深度音乐、深度影音、深度截图、深度终端、深度翻译等。最重要的是他们自主研发了一个非常漂亮、非常完善、交互体验非常棒的桌面环境，如图 4-4 所示。



图 4-4 DeepinLinux 操作系统桌面

如果你的计算机上没有 Linux 发行版，而且又想知道怎么安装笔者的这款 Linux 发行版，请参照笔者的博客（<http://blog.chinaunix.net/uid-28032128-id-4017703.html>）。由于限于篇幅，本书不会介绍如何安装 Linux，如果你不会在物理 PC 上或者在现有的虚拟机中安装 Ubuntu 或者 DeepinLinux，请参阅其他资料。因为你的计算机上必须要有 Linux 操作系统，最好是 Ubuntu 或者 DeepinLinux，才能运行后面那些开发我们自己操作系统内核的工具。并且笔者强烈建议使用 DeepinLinux，因为本书后面的描述全部是基于 DeepinLinux 的。

Linux 操作系统以前是以向操作系统输入命令为主要交互方式的操作系统，要想完成什么功能，就要向系统输入功能对应的命令。即使是今天 Linux 系统有了非常漂亮的图形操作环境，但有些功能还是依赖有关命令行程序，可是在图形环境下怎么输入命令运行程序呢，Linux 操作系统的图形环境都会提供一个名叫“终端”的软件，相当于 Windows 的命令提示符窗口。当然 Ubuntu 下就叫“终端”，DeepinLinux 下叫“深度终端”。打开深度终端，如图 4-5 所示。

下面通过一个例子，来熟悉一下 Linux 环境。首先在 DeepinLinux 上安装两个软件，这两个软件也是后面必须要用到的软件。

首先来安装第一个软件：minicom。minicom 是一个串口通信工具，就像 Windows 下的超级终端。可用来与串口设备通信，到时我们的 mini2440 开发板就是用串口和 PC 通信的，通过它就可以观察到开发板的运行状态了。如何安装 minicom 呢，步骤如下：

第 1 步：在 DeepinLinux 下打开“深度终端”软件。

第 2 步：在“深度终端”中输入：`sudo apt-get install minicom` 后回车即可，如图 4-6 所示。

再来安装一个库 `libusb-dev`。这个库是一个操作 USB 设备的库，后面就是用它来在 PC 上操作开发板，把开发板作为一个 USB 设备，下载我们的操作系统内核到开发板上运行，以

便测试我们的操作系统内核。安装 libusb-dev 的步骤如下：



图 4-5 深度终端

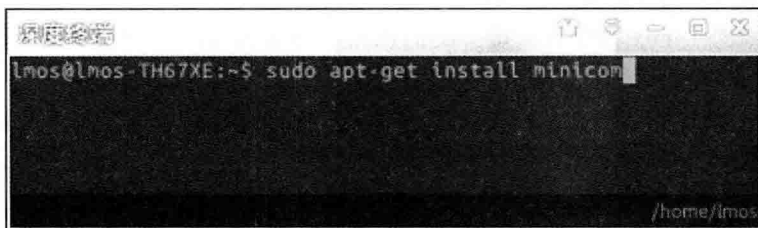


图 4-6 安装 minicom

第 1 步：在 DeepinLinux 下打开“深度终端”。

第 2 步：在“深度终端”中输入：`sudo apt-get install libusb-dev` 后回车即可，如图 4-7 所示。

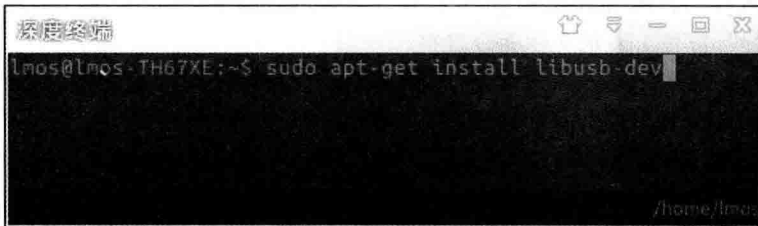


图 4-7 安装 libusb-dev

上述安装软件的操作，要确保你的计算机处于联网状态，因为 apt-get 软件包管理器是通过专门的软件源服务器，下载对应的软件包并进行安装的。

到此为止，如果你觉得意犹未尽，那么请参阅其他相关专著，本书不是介绍 Linux 操作

系统的专著，但是上述两个软件必须要安装。如果你真得完全不会使用 Linux 操作系统，笔者只能感到有点小遗憾，不过也不用害怕，在介绍后面的内容时，也会时不时地写到如何使用 Linux 操作系统。

4.2.2 文本编辑器

首先如果读者是一个十分了解 Linux 系统的人或者只是对我们操作系统内核的实现感兴趣，大可略过本小节。

操作系统内核最终是由一行行代码构成的，一行行代码构成一个个以“.c”(C 程序文件)或者“.s”(汇编程序文件)为后缀名的文件。这两种文件就是普通的文本文件，也就是说任何文本编辑器软件都可以用来写 C 程序或者汇编程序代码。现在就来看看 Linux 操作系统下应该选择什么样的文本编辑器。

Linux 操作系统下有很多针对程序员写代码的利器，如 emacs、vim、sublime text 2 等。这些“高大上”的工具对于绝大多数非专业人员来说，首先玩转它们本身难度是极大的，所以得选择几个简单的，如 gedit、qt creator。

下面就用 gedit、qt creator 这两软件分别来写我们最熟悉的 C 程序：“hello, world!!”为了让读者更加熟悉 Linux 命令行环境，我们依然用终端输入相关命令来完成这个例子。

先用 gedit 试试，步骤如下：

第 1 步：首先新建一个文件夹“test”，在“深度终端”中输入 `mkdir test` 回车即可。`mkdir` 命令就是 Linux 下建立文件夹的命令。

第 2 步：切换到 test 文件夹下，在“深度终端”中输入 `cd test` 回车即可。`cd` 命令就是 Linux 下切换文件夹的命令。

第 3 步：建立一个“helloworld.c”文件，在“深度终端”中输入 `touch helloworld.c` 回车即可。`touch` 命令就是 Linux 下建立一个空文件的命令。

第 4 步：用 `gedit ~ /test/helloworld.c` 命令打开这个文件，就会出现 gedit 编辑器的图形编辑界面，输入代码，保存并关闭 gedit 软件，如图 4-8 所示。“~”符号，表示当前用户的工作文件夹。“/”符号，表示文件夹分界符。

第 5 步：用 `gcc helloworld.c -o helloworld` 命令编译这个 C 程序，在 test 文件夹下会出现一个可执行文件 helloworld。

第 6 步：用 `./helloworld` 命令运行这个程序，“.”符号，表示当前文件夹。

上述过程如图 4-9 所示。当然也可以用图形环境完成上述 1) ~ 4) 步的过程，和在 Windows 下的操作并没有多大差别。

qt creator 是最近新出来的一款 IDE，可以用来编写 C/C++ 代码，当然也可以用来编写其他文本文件，笔者经常使用它编写内核代码。功能非常强大，如代码补全、自动保存、代码函数跟踪等。当然如果你的 Ubuntu 或者 DeepinLinux 上没有 qt creator，可以在终端中输入 `sudo apt-get install qt creator` 命令安装 qt creator。还是上面那个例子，用 qt creator 打开

helloworld.c, 如图 4-10 所示。

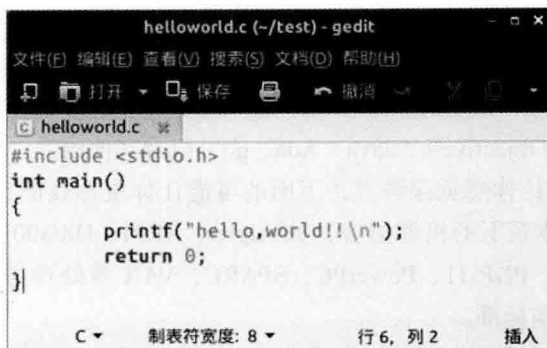


图 4-8 编辑 hello, world!!

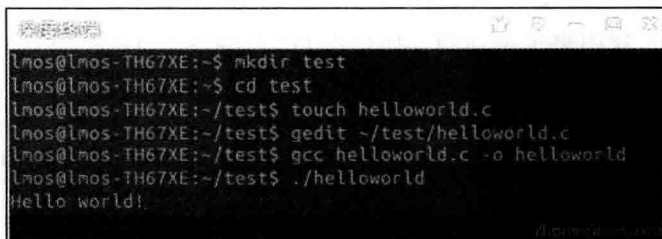


图 4-9 建立、编辑、编译、运行 hello, world!!



图 4-10 qt creator 打开 helloworld.c

以上这些都不是重点, 关键是读者能挑选一款自己用得顺手的文本编辑器编写代码即可。至于什么文本编辑器并不重要, 如果读者能配置好 emacs、vim, 能亲自领略一下这两大神器的强大, 自然也是再好不过的。

4.2.3 GCC

GCC 即 GNU Compiler Collection, GNU 编译器套装, 是一套由 GNU 开发的编译器。它是一套以 GPL 及 LGPL 许可证所发布的自由软件, 是 Linux 操作系统上的标准编译器, 当然也可以运行在 Windows、UNIX、Mac OS 上。GCC 不仅是 C 语言编译器, 它还可以编译 C++、Fortran、Pascal、Objective-c、Java、Ada、go, 以及其他语言。如果上面 GCC 跨语言、跨操作系统的特性已经让你感觉很强大, 下面的可能让你更加震撼, GCC 还能将上述这些语言编译成不同 CPU 体系下的机器语言, 如 Alpha、ARM、H8/300、x86、x86-64、IA-64、Motorola 68000、MIPS、PDP-11、PowerPC、SPARC、VAX 等处理器。因此 GCC 也常被认为是跨平台编译器的事实标准。

前面用 Linux 系统中自带的 GCC 编译了 helloworld 程序, 这个 GCC 是 x86 体系下的 GCC。因为我们的 Linux 系统安装在 PC 上, PC 都是采用 x86 体系的 CPU, 编译出的程序也只是在 x86 CPU 上能执行。而我们要开发的是 mini2440 平台上的操作系统内核, 并且是在 PC 上开发, 这引来了个问题, 如何在 PC 上编译出 ARM CPU (因为 mini2440 平台上用的 ARM CPU) 上的程序, 前面已经知道 GCC 是可以编译出 ARM 体系下的程序的, 而且在一种平台上 (PC 平台) 编译另一个平台上 (mini2440 平台) 的程序的方式叫交叉编译。

交叉编译, 首先要安装 ARM 的 GCC 交叉编译器, 对于 mini2440 平台的 GCC 安装步骤如下:

第 1 步: 打开 DeepinLinux 下的网络浏览器 chrome, 输入网址: <http://arm9download.cncn.cn/mini2440/linux/arm-linux-gcc-4.4.3-20100728.tar.gz> 下载 arm-linux-gcc-4.4.3 到相关目录, 也可以从本书光盘中得到这款 GCC。

第 2 步: 用 `tar -xf arm-linux-gcc-4.4.3-20100728.tar.gz` 命令解压刚刚下载的 tar.gz 文件包。

第 3 步: 把刚刚解压出的文件夹复制到 /opt/ 目录下, 可以用命令 `sudo cp -r ~ /opt/` 实现, sudo 是因为 Linux 是多用户系统, 有的目录只有 root 用户 (超级用户) 权限才能访问。

第 4 步: 用 `sudo gedit /etc/profile` 命令打开 /etc 下的 profile 文件, 把 `export PATH="/opt/opt/FriendlyARM/toolschain/4.4.3/bin:$PATH"` 增加到这个文件的末尾, 然后执行 `source /etc/profile` 命令, 即把 arm-linux-gcc 加入到环境变量中, 以便在当前工作目录下可以使用 arm-linux-gcc 编译器。

第 5 步: 在深度终端中, 输入 `arm-linux-gcc -v` 测试一下, 这时终端中会出现 GCC 版本信息。

上述过程如图 4-11 所示。

交叉编译器已经安装好了, 下面来看看 arm-linux-gcc 的一些编译选项, arm-linux-gcc 的编译选项太多了, 不必每个都弄得一清二楚, 这里只是关注对于编译操作系统内核需要关注的选项, 如表 4-1 所示。

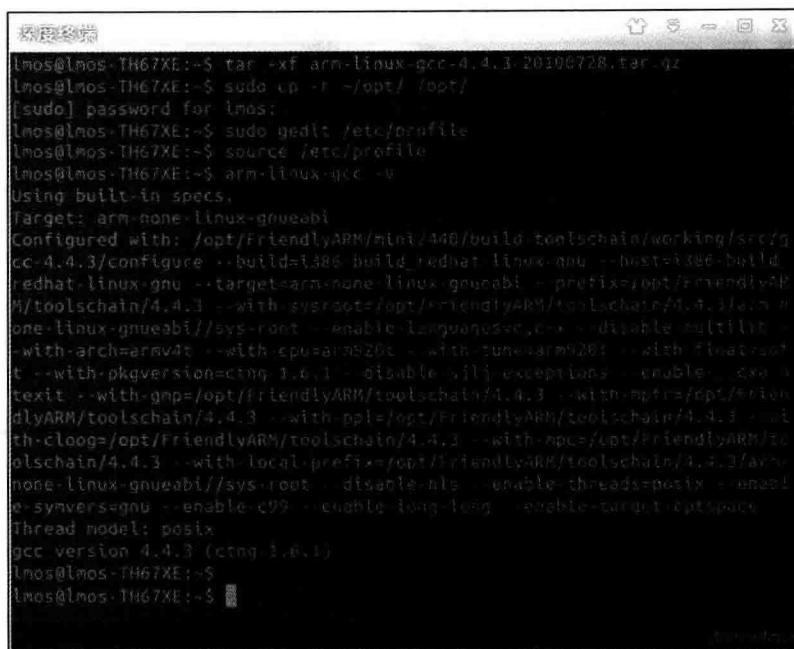


图 4-11 安装 arm-linux-gcc

表4-1 arm-linux-gcc编译选项

相 关 选 项	说 明
-c	编译生成可链接目标文件
-I dir	指定头文件目录, dir 是头文件所在的目录
-E	只做预处理而不编译
-P	禁止 GCC 在预处理后的文件中插入任何额外的信息
-Ox	“x” 可以是 0、1、2、3、s, 表示 GCC 对代码的优化级别, 0 表示不优化, 3 表示最优化, s 表示优化程序大小
-std	表示 GCC 编译时所用程序语言的标准, -std=C99 表示采用 C 语言的 C99 标准
-S	只将 C 语言程序编译生成相关机器的汇编代码
-mtune	指定某一型号 CPU 的选项, 如我们的是 -mtune=arm7tdmi
-march	march 指定的是当前 CPU 的架构, -march=armv4 表示生成 ARM 体系 4 的可执行文件
-save-temps	保存所有 GCC 编译时生成的临时文件
-fno-ident	禁止 GCC 在可执行文件末尾生成编译器信息
-fno-builtin	除非利用前缀 __builtin_ 进行引用, 否则不使用 GCC 所有的内建函数, 编译内核时最好不要用编译器的内建函数。GCC 为了提高程序性能, 把很多常用函数都做好了, 如 strcpy

(续)

相关选项	说 明
-ffreestanding	按独立环境编译, 该环境可以没有标准库, 且对 main() 函数没有要求。最典型的例子就是操作系统内核
-fno-stack-protector	禁用栈保护, 有些应用利用栈溢出进行攻击, GCC 实现了一种保护机制, 但对内核没有用, 而且会增加无用的保护栈的代码
-fomit-frame-pointer	对于不需要栈指针的函数就不在特定寄存器中保存栈指针, 因此可以忽略存储和检索地址的代码, 同时对许多函数可多提供一个额外的寄存器
-Wunused-variable	用来警告存在一个定义了却未使用的局部变量或者非常量的 static 变量
-Wno-unused-parameter	禁用警告一个函数的参数在函数的实现中并未被用到
-Wno-sign-conversion	禁用警告符号和无符号整数之间的转换

了解了 GCC 的一些常用的选项, 再来看看 GCC 编译一个 C 程序的过程, 当然以上的选项和下面的编译过程对 arm-linux-gcc 是等价的。如图 4-12 所示, 可以发现 GCC 编译器并不是一个程序, 而是由多个程序: 预处理程序、C 编译器、汇编器一起完成编译工作的。

安装了交叉编译器 GCC, 知道了它的许多选项, 又看了它全部的工作流程, 后面就可以熟练地应用 GCC 编译我们的操作系统内核了, 并且可以利用 GCC 的一些特性, 达到一些编译之外的特殊功能。

4.2.4 LD

LD 是一款由 GNU 开发的程序链接器。是以 GPL 及 LGPL 许可证所发布的自由软件, 也是 Linux 操作系统上的标准链接器, 其实它是 GCC 编译器套装中一个组件, 通常是和 GCC 一起工作的, 当然也是跨操作系统平台、跨硬件平台的, 前面安装了 GCC 交叉编译器, 当然 LD 也就包含在其中了。

LD 其实就是把诸多 GCC 生成的可链接的目标文件合并成一个大的可执行文件, 主要完成如下功能:

1) 把每个目标文件中各个相同的段, 合并成各个相同的大段, 一个目标文件中通常默认情况下至少有三个段: .text 段即代码段, .data 段即已初始化的数据段, .bss 段即未初始化或者初始化为 0 的段。后面会了解到怎么让编译器生成我们自定义的段。

2) 重定位程序中的地址, 完成装载程序运行时的地址绑定, 一般程序都必须从某一固

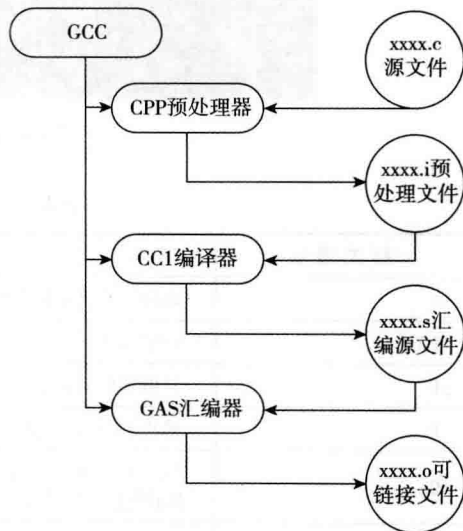


图 4-12 GCC 编译 C 程序的过程

定的地址开始运行。

3) 解决不同目标文件之间符号引用的关系, 例如, A 目标文件中的一个函数调用了 B 目标文件中一个函数。

上述过程如图 4-13 所示。

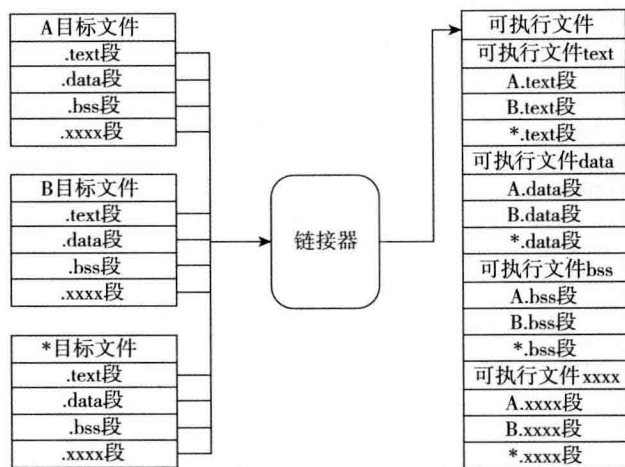


图 4-13 LD 链接目标文件的过程

看了 LD 链接器的工作过程, 现在来看一下 LD 链接器的一些选项, 如表 4-2 所示。

表 4-2 LD 链接器选项

LD 选项	相 关 说 明
-c	指定程序的入口点, 一般是从程序中的“_start”标号开始的
-T	-T 后面跟着 LD 要打开的文件, 一般是 LD 链接脚本文件
-Map	指示 LD 输出可执行文件的 Map 文件, 这个文件里有这个可执行文件所有的段、符号以及符号所在的地址信息
-Bstatic	禁止链接共享库, 操作系统内核是不需要共享库的
-nostdlib	禁止搜索标准库, 内核也是不需要标准库的
-s	删除所有可执行文件中的符号信息
-Ttext	后面跟一个地址值, 表示 LD 输出的可执行文件 text 段运行时的地址
-Tdata	后面跟一个地址值, 表示 LD 输出的可执行文件 data 段运行时的地址
-Tbss	后面跟一个地址值, 表示 LD 输出的可执行文件 bss 段运行时的地址

关于 LD 的选项就介绍到这里, 后面所用到的也就这么多。现在去研究 LD 的链接脚本, 这个链接脚本才是 LD 真正强大的地方。

什么是链接脚本呢, 简单地说就是一个文本文件, 它里面存放了一些信息, 这些信息有一定的规则, 相当于一种编程语言, LD 读取这个文件里的信息, 然后根据这些信息完成相

关的动作。

先来看一看一个非常简单的 LD 链接脚本：

```
INPUT(file.o file1.o)
OUTPUT(binfile.elf)
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x30008000;
    __begin_text = .;
    .text ALIGN(4) : { *(.text) }
    __end_text = .;
    __begin_data = .;
    .data ALIGN(4) : { *(.data) }
    __end_data = .;
    __begin_bss = .;
    .bss ALIGN(4) : { *(.bss) }
    __end_bss = .;
}
```

上面 LD 链接脚本中的内容，我们要关注的内容如下：

- 1) INPUT(), 表示需要链接的目标文件。
- 2) OUTPUT(), 表示链接后输出可执行文件的名称，名称随意。
- 3) OUTPUT_FORMAT(), 表示输出可执行文件的格式，如 “elf32-littlearm”。
- 4) OUTPUT_ARCH(), 表示输出可执行文件的机器体系，如 ARM。
- 5) ENTRY(), 表示程序中某一标号为可执行文件的入口点，即表示程序从这里开始运行，如 _start 标号。

6) SECTIONS, 表示定义一个输出段，内部按照一定的规则组织合并所有可链接目标文件内的所有段，最后形成一个输出的大段。以一对 “{}” 结束。

7) “.” 表示当前地址计数器，LD 创建程序加载运行地址时，就是以这个基准的，最开始应该给初始值，LD 就是以那个初始值为地址开始链接程序的。

8) __begin_text、__end_text、__begin_data、__end_data、__begin_bss、__end_bss，诸如这些标号不是必须的，标号可由英文字符串组成，也可以没有这些标号，它们是属于自定义的，如果定义了这些标号，就可以在程序代码中访问它们，它们的值表现为地址，例如，上面的 __begin_text 的值为 0x30008000，关于访问它们的方法后面会有介绍，上面的 “=” 相当于 C 语言里的赋值号，“. = 0x30008000;”、“__begin_text = .;” 相当于两个表达式，分别要以 “;” 结束。

9) “.text ALIGN(4) : { *(.text) }”、“.data ALIGN(4) : { *(.data) }”、“.bss ALIGN(4) : { *(.bss) }”，其中的 ALIGN(4) 都表示每个大段链接时的地址按 4B 对齐，ALIGN(4) 不是必须的。ALIGN(4) 前面的 .text、.data、.bss，表示 LD 合成可执行文件时将会输出大段的段名，大段

名后面要以“:”开始,紧跟其后的是“{}”,“{}”里面是链接规则。例如,“{*(.text)}”表示将所有目标文件中的.text段(用“()”括起来)组成一个大的.text段,其中的“*”表示通配符,当然可以在此写上所有目标文件的名称。其他.data、.bss段是也一样的道理。可以仔细观察一下图4-13。

关于LD链接脚本就介绍到这里了,在后面的介绍中,还会有一个用于我们操作系统内核的链接脚本,有了这里的基础一定可以看懂那个链接脚本。也正是因为操作系统内核的特殊性,我们不能用LD提供的默认的链接脚本,那个脚本是用于普通应用的。

正是因为有了LD链接器的存在,使得编译器在编译程序时不必考虑程序运行时的地址,使得一个大程序可以分解成许多小模块以便软件的维护,使得程序模块之间可以互相调用,以便程序的开发可以分工协作化,还使得可以把常用的功能做成库以供其他软件使用,增加代码的可重用性。这当然也是所有软件开发人员所希望的。

了解了LD是做什么的,也看了它今后要用到的一些选项,探索了指挥它的链接脚本,基本上也就能使用它为我们开发操作系统内核而服务了,至此也结束对LD链接器的探讨。

4.2.5 make

前面了解了GCC、LD,并且以后操作系统内核的代码就是用GCC编译的,一个最简单操作系统内核至少也由几十个C程序文件和几个汇编程序文件构成的。如果一个个程序文件都要在终端中输入GCC及相关命令去编译,那完全对我们的身心是种巨大的折磨。怎么办呢,还好开发操作系统的前辈,开发了一个软件叫make。

在软件开发中,make是一个工具程序,它读取一个叫makefile的文件,也是一种文本文件,这个文件中写好了构建软件的规则,它能根据这些规则自动化构建软件。

makefile文件中规则是这样的,首先有一个或者许多构建目标称为target;目标后面紧跟着用于构建该目标所需要的文件,目标下面是构建该目标所需要的命令及参数。与此同时,它也检查文件的依赖关系,如果需要,它会调用一些外部软件来完成任务。第一次构建目标后,下一次执行make时,它会根据该目标所需的文件是否更新了,如果没有更新且该目标又存在,那么它便不会构建该目标。这种特性非常有利于编译程序源代码。

任何一个Linux发行版中都默认自带这个make程序,所以不需要额外的安装工作,可以直接使用即可。

还是来看看make程序的选项,make常用的选项没有几个,如表4-3所示。

表4-3 make常用选项

选 项	相 关 说 明
-B	认为所有的目标都需要更新
-C	指定读取 makefile 文件的目录
-f	指定需要执行的 makefile。makefile 是可以重命名的

(续)

选 项	相 关 说 明
-k	出错也不停止运行
-n	仅输出执行过程中的命令序列，但并不执行命令

下面来通过一个简单的 makefile 实例，来看看怎么写好一个 makefile，如代码清单 4-1 所示。

代码清单4-1 makefile实例1

helloworld : helloworld.c gcc -o \$@ \$<

这就是一个最简单的 makefile。其中，构建目标为 helloworld，目标所依赖的文件是 helloworld.c，它们之间用“:” 隔开。要执行的命令是 gcc -o \$@ \$<，\$@ \$< 是 make 程序内部使用的两个变量，分别代表目标及目标所依赖的文件，也就是说 make 程序最终会把 gcc -o helloworld helloworld.c 送到终端里去执行。注意 make 规定要执行的命令的前面必有一个 tab 符。

再来看一个有点复杂的 makefile，如代码清单 4-2 所示。

代码清单4-2 makefile实例2

CC = gcc	#定义一个宏CC 等于gcc
CFLAGS = -c	#定义一个宏 CFLAGS等于-c
OBJS_FILE = file.c file1.c file2.c file3.c file4.c	#定义一个宏
.PHONY : all everything	#定义两个伪目标all、everything
all:everything	#伪目标all依赖伪目标everything
everything :\$(OBJS_FILE)	#伪目标everything依赖OBJS_FILE，而OBJS_FILE是宏会被替换成file.c file1.c file2.c file3.c file4.c
%.o : %.c	
\$(CC) \$(CFLAGS) -o \$@ \$<	

make 规定“#” 后面为注释,make 处理 makefile 时会自动丢弃。makefile 中可以定义宏，方法是在一个字符串后跟一个“=” 或者“:=” 符号，引用宏时要用“\$(宏名)”，宏最终会在宏出现的地方替换成相应的字符串，例如，\$(CC) 会被替换成 gcc，\$(OBJS_FILE) 会被替换成 file.c file1.c file2.c file3.c file4.c。 .PHONY 在 makefile 中表示定义伪目标，所谓伪目标，就是它不代表一个真正的文件名，在执行 make 时可以指定这个目标来执行其所在规则定义的命令。但是伪目标可以依赖另一个伪目标或者文件，例如，all 依赖 everything，everything 最终依赖 file.c file1.c file2.c file3.c file4.c。然而发现 everything 下面并没有相关的执行命令，但是下面有个通用规则：“%.o: %.c”。其中的“%” 表示通配符，表示所有以“.o” 结尾的文件依赖所有以“.c” 结尾的文件，例如，file.c file1.c file2.c file3.c file4.c 通过这个通用规则会自动转换为 file.o:file.c、file1.o: file1.c、file2.o: file2.c、file3.o:file3.c、file4.o: file4.c，然后针对这 5 个规则，分别会执行 \$(CC) \$(CFLAGS) -o \$@ \$< 命令，当然最终转换为 gcc -c -

O XXXX.O XXXX.C, “XXXX”表示一个具体的文件名。

现在来把上面的那个 makefile 稍稍改进一下, 如果细心观察不难发现上面的 makefile 一共有 4 大部分: 构建时用到的命令、依赖文件、构建目标、构建规则。

首先建立一个文件 cmd.mh, 文件名随意。用于存放相关的命令, 文件内容如代码清单 4-3 所示。

代码清单4-3 包含相关命令的makefile

```
CC = gcc                #定义一个宏 CC 等于gcc
CFLAGS = -c             #定义一个宏 CFLAGS 等于-c
```

实际中或许有更多的命令, 这里为了简单起见, 只有一个命令和一个命令选项。

再建立一个文件 objs.mh, 文件名随意。用于存放需要编译的文件即依赖文件, 文件内容如代码清单 4-4 所示。

代码清单4-4 包含相关依赖文件的makefile

```
OBJS_FILE = file.c file1.c file2.c file3.c file4.c #定义一个宏
#当然可以有更多的文件
```

然后建立一个文件 rule.mh, 文件名随意。存放用于构建目标的通用规则, 文件内容如代码清单 4-5 所示。

代码清单4-5 包含通用规则的makefile

```
%O : %.C
$(CC) $(CFLAGS) -o $@ $<
#也许有更多的规则例如:
#%.O : %.S
# $(AS) $(ASFLAGS) -o $@ $<
```

最后在任何情况下, 我们可能就只需建立这样的 makefile, 如代码清单 4-6 所示。

代码清单4-6 makefile实例3

```
include cmd.mh
include objs.mh
.PHONY : all everything
all:everything
everything :$( OBJS_FILE)
include rule.mh
```

代码清单 4-6 中的 include 和 C 语言中的 include 是一样的工作机制, 同样也是包含一个文本文件, 只是 make 不需要 “<” 和 “””。并且 make 会在 include 的地方开始展开包含的这个文件。也就是说上面这个 makefile 最终会变成和前面代码清单 4-2 中的那个 makefile 一样。这样写起 makefile 文件就容易多了, 有时甚至只需要改变 everything: 后面的这个宏就可以了, 这对于管理构建操作内核模块文件非常有效。

通过三个简单的例子, 大致了解了 make 的工作机制, 无非就是层层替换, 然后根据目

标的依赖文件是否已更新，决定是否重新构建目标，最后执行构建目标的命令。后面会有一个用于构建我们操作系统内核的 makefile 例子。也许那时就会对 make 明白许多。

4.3 LMOSEM 的构建系统

兴趣、自由、自主、外加有点小疯狂，是我们研究开发操作系统内核的核心理念，因此用 LMOSEM (Liberty Madness Operating System Embedded) 来命名我们的操作系统内核。后面就可以直接用 LMOSEM 来代指我们的操作系统内核了。

前面了解了 make 完全可以用来管理 LMOSEM 这个软件工程，下面就去研究一下怎么用众多的 makefile 一起协作生成相关的文件，如生成要编译的对象列表文件、用于 LD 链接器的链接脚本，以至于最后能自动化编译 LMOSEM。

4.3.1 LMOSEM 的 makefile

开发操作系统内核和开发普通应用并无太大区别，就是写好代码然后编译代码。不要说你会把整个操作系统写成一个源代码文件，就是 LMOSEM 这种用于学习的操作系统内核也有六七十个源代码文件。这么多源代码文件，怎么自动化地编译它们，就成了非常重要的问题。

make 是个好东西且历史悠久，最初开发它的人就是为了自动化编译像操作系统这样的大型软件，然而几十个源代码文件用一个 makefile 文件去处理，一是没有结构可言，二是容易变得混乱，三是不好控制。

LMOSEM 的编译一共是由如下 11 个 makefile 文件互相协同完成的：

- 1) makefile，是顶层全局的，它负责调用 make 执行所有下层的 makefile。
- 2) krnlbuidcmd.mh，这个 makefile 中包括了编译的常用命令，如 GCC、LD 等。其他以“.mk”结尾的 makefile 文件中都要用“include”包含它。
- 3) krnlbuidrule.mh，这个 makefile 中包括了编译的通用规则，其他以“.mk”结尾的 makefile 文件中都要用“include”包含它。
- 4) pretreatment.mk，该 makefile 是最开始执行的，它调用 GCC 预处理功能，用 krnlobjs.S、lmemknllink.S 两文件生成 krnlobjs.mkh 和 lmemknllink.lids。
- 5) krnlobjs.mkh，该文件由 krnlobjs.S 生成，里面主要是一些要编译的源文件列表。其他以“.mk”结尾的 makefile 文件中都要用“include”包含它。
- 6) lmosemhal.mk，该 makefile 主要用于编译生成硬件相关层的各个模块。
- 7) lmosemkrl.mk，该 makefile 主要用于编译生成内核层（包括内核功能层和内核接口层）的各个模块。
- 8) lmosemdrv.mk，该 makefile 主要用于编译生成一些驱动程序的各个模块。
- 9) lmosemllib.mk，该 makefile 主要用于编译生成库的各个模块。

10) lmoosemtask.mk, 该 makefile 主要用于编译生成应用程序。

11) lmoosemlink.mk, 该 makefile 首先根据前面生成的 lmemknl.link.lds 链接脚本文件, 然后把上面编译生成的各个模块链接在一起, 生成最后的 LMOSEM 可执行文件。

下面再来看一幅图可能更加形象, 如图 4-14 所示。

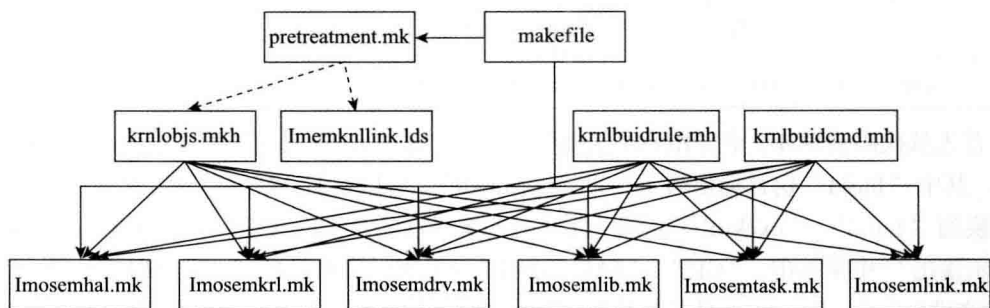


图 4-14 用于构建 LMOSEM 的各个 makefile

再来看看这一整个编译过程, 首先从顶层全局的 makefile 开始分析, 如代码清单 4-7 所示, 由于代码很多, 全部列举出来会占用大量的篇幅, 不利于阅读, 省略的部分会用“……”代替。代码文件的路径如“XXXX/lmosem/build/lmoosemhal.mk”中的“XXXX”表示 LMOSEM 工程目录所在的路径。

代码清单4-7 LMOSEM的全局makefile

```
.....
CD = cd
RM = rm
MAKE = make
.....
PREMENTMFLGS = -C $(BUILD_PATH) -f pretreatment.mk
ARMHALYMFLGS = -C $(BUILD_PATH) -f lmoosemhal.mk
ARMKRNLNFLGS = -C $(BUILD_PATH) -f lmoosemkrl.mk
ARMDRIVMFLGS = -C $(BUILD_PATH) -f lmoosemdrv.mk
ARMLIBSMFLGS = -C $(BUILD_PATH) -f lmoosemlib.mk
ARMTASKMFLGS = -C $(BUILD_PATH) -f lmoosemtask.mk
ARMLINKMFLGS = -C $(BUILD_PATH) -f lmoosemlink.mk
.....
BUILD_PATH = ./build
EXKNL_PATH = ./exckrnl
.PHONY : build print clean all cpkrnl knlexc writekrl downkrnl
build: clean print all
all:
    $(MAKE) $(PREMENTMFLGS)
    $(MAKE) $(ARMHALYMFLGS)
    $(MAKE) $(ARMKRNLNFLGS)
    $(MAKE) $(ARMDRIVMFLGS)
    $(MAKE) $(ARMLIBSMFLGS)
```

```

$(MAKE) $(ARMTASKMFLGS)
$(MAKE) $(ARMLINKMFLGS)
@echo '恭喜我，系统编译构建完成！ ^_^'

clean:
$(CD) $(BUILD_PATH); $(RM) -f *.mkh *.lds *.o *.bin *.i *.elf *.krnl *.s
*.map *.lib
$(CD) $(EXKNL_PATH); $(RM) -f *.mkh *.lds *.o *.bin *.i *.elf *.krnl *.s
*.map *.lib @echo '清理全部已构建文件... ^_^'

print:
@echo '*****正在开始编译构建系统*****'

```

首先从代码清单 4-7 中 .PHONY 开始定义一些伪目标，我们主要关注“build print clean all”。其中“build”伪目标依赖于“clean print all”，所以它们会依次执行，“clean”伪目标会切换到“build”、“exckrnl”目录下删除一些编译器的产生的文件，“@echo”在 makefile 中表示输出‘ ’中字符串。“CD”、“RM”分别是两个宏，会替换成“cd”（切换目录）、“rm”（删除文件）命令。“clean”伪目标执行完后会执行“print”伪目标，它是输出字符串的。最后会执行“all”伪目标，“all”伪目标会执行 7 条 make 命令。我们选择其中两条来分析一下。

第一条 \$(MAKE) \$(PREMENTMFLGS)，因为它会产生两个重要文件，这两个文件是所有其他 makefile 所依赖的，所以它要最先被执行，它被替换后就是 make -C ./build -f pretreatment.mk。也就是说 make 会到 build 目录下读取 pretreatment.mk 文件执行。看一看 pretreatment.mk 文件中是什么，如代码清单 4-8 所示。

代码清单 4-8 用于预处理的 make 文件 pretreatment.mk

```

.....
KERNELCE_PATH = ../script/
HEADFILE_PATH = -I ../include/script/ -I ../include/ -I ../include/bastypeinc -I
../include/halinc
CCBUILDPATH = $(KERNELCE_PATH)
include krnlbuidcmd.mh
PREMENT_OBJS = krnlobjs.mkh lmemknllink.lds
.PHONY : all everything build_kernel
all: build_kernel
build_kernel:everything
everything : $(PREMENT_OBJS)
include krnlbuidrule.mh

```

代码清单 4-8 中有两个 include，分别包含了 krnlbuidcmd.mh、krnlbuidrule.mh 两个文件，我们把它展开来看看，如代码清单 4-9 所示。

代码清单 4-9 pretreatment.mk 展开后的情形

```

.....
KERNELCE_PATH = ../script/
HEADFILE_PATH = -I ../include/script/ -I ../include/ -I ../include/bastypeinc -I
../include/halinc

```

```
CCBUILDPATH = $(KERNELCE_PATH)
CC = arm-linux-gcc
.....
CPPFLGSLDS = $(HEADFILE_PATH) -E -P
PREMENT_OBJS = krnlobjs.mkh lmemknllink.lds
.PHONY : all everything build_kernel
all: build_kernel
build_kernel:everything
everything : $(PREMENT_OBJS)
.....
%.lds : $(CCBUILDPATH)%.S
    $(CC) $(CPPFLGSLDS) -o $@ $<
    $(PRINTCSTR)
%.mkh : $(CCBUILDPATH)%.S
    $(CC) $(CPPFLGSLDS) -o $@ $<
    $(PRINTCSTR)
```

在代码清单 4-9 中, `KERNELCE_PATH` 表示要编译的源代码文件的目录, `HEADFILE_PATH` 表示 GCC 查询头文件的目录列表。目标就是生成 `krnlobjs.mkh`、`lmemknllink.lds` 两个文件, 然而 `script` 目录下确实有 `krnlobjs.S`、`lmemknllink.S` 正好匹配下面两条规则, 会分别执行 `arm-linux-gcc-I ../include/script/..... -E -P-o lmemknllink.lds ../script/ lmemknllink.S`、`arm-linux-gcc-I ../include/script/..... -E -P-o krnlobjs.mkh ../script/krnlobjs.S`。这两条命令是预处理命令。

这里主要分析 `krnlobjs.S` 文件, 如代码清单 4-10 所示。`lmemknllink.lds` 文件以后再分析。

代码清单4-10 生产编译对象的代码

```
.....
#include "buildfile.h"
BUILD_MK_BOOT_OBJS :=
BUILD_MK_HALY_OBJS :=
.....
BUILD_MK_BOOT_OBJS +=BUILD_BOOT_OBJS
BUILD_MK_HALY_OBJS +=BUILD_HALY_OBJS
.....
```

代码清单 4-10 中不是汇编语言代码, 也不是 C 程序代码, 所以不能被编译只能作预处理操作, 这个文件中包含了 `buildfile.h` 文件, 其内容如代码清单 4-11 所示。

代码清单4-11 包含编译对象名称的头文件

```
#ifndef BUILDFILE_H
#define BUILDFILE_H
#define BUILD_BOOT_OBJS
#define BUILD_HALY_OBJS init.o lmoemhal_start.o interrupt.o intabtdistr.o.....
.....
#endif
```

看到这里，会发现 `krnlobjs.S` 文件中的 `BUILD_BOOT_OBJS`、`BUILD_HALY_OBJS` 不过是 `buildfile.h` 文件中定义的两个宏，所谓对 `krnlobjs.S` 文件进行预处理，也不过就是进行宏替换操作。到这里你也许会恍然大悟，最终生成的 `krnlobjs.mkh` 文件如代码清单 4-12 所示。

代码清单4-12 生成后的krnlobjs.mkh文件

```

BUILD_MK_BOOT_OBJS :=
BUILD_MK_HALY_OBJS :=
.....
BUILD_MK_BOOT_OBJS +=
BUILD_MK_HALY_OBJS +=init.o lmosemhal_start.o interrupt.o intabtdistr.o.....
.....

```

到这里我们看到了将 `krnlobjs.S` 文件转换成其他 `makefile` 文件都要包含的编译对象列表文件 `krnlobjs.mkh` 的整个过程，并且只需要通过改变一个 `buildfile.h` 文件就能决定哪些代码文件需要编译。

分析了生成 `krnlobjs.mkh` 文件的过程，并且它是其他 `LMOSEM` 组件编译的基础。再来看看怎么编译 `LMOSEM` 硬件相关层的代码模块，由代码清单 4-7 知道，执行完 `make -C ./build -f pretreatment.mk` 命令，接着会执行 `make -C ./build -f lmosemhal.mk` 命令。`lmosemhal.mk` 中的内容如代码清单 4-13 所示。

代码清单4-13 用于生成LMOSEM的hal层中的各代码模块

```

.....
KERNELCE_PATH = ../hal/
HEADFILE_PATH = -I ../include -I ../include/bastypeinc -I ../include/halinc.....
CCBUILDPATH = $(KERNELCE_PATH)
include krnlbuidcmd.mh
include krnlobjs.mkh
.PHONY : all everything build_kernel
all: build_kernel
build_kernel:everything
everything :$(BUILD_MK_HALY_OBJS)
include krnlbuidrule.mh

```

其实所有的 `lmosem***.mk` 文件都和代码清单 4-13 是一样的，唯一不同的是代码中的 `KERNELCE_PATH`、`HEADFILE_PATH` 两个宏后面的字符串，分别表示要编译源代码文件的目录和 `GCC` 查询头文件的目录列表，还有 `everything` 伪目标所依赖的文件对象也不同，这里它是依赖 `krnlobjs.mkh` 文件中 `BUILD_MK_HALY_OBJS`。可以看到 `lmosemhal.mk` 中包含了三个文件：`krnlbuidcmd.mh`、`krnlobjs.mkh`、`krnlbuidrule.mh`。同样地，也把它们在包含的地方展开，以便看得更清楚，如代码清单 4-14 所示。

代码清单4-14 lmoosemhal.mk展开后的情形

```

.....
KERNELCE_PATH = ../hal/
HEADFILE_PATH = -I ../include -I ../include/bastypeinc -I ../include/halinc.....
CCBUILDPATH = $(KERNELCE_PATH)
CC = arm-linux-gcc
CFLAGS = $(HEADFILE_PATH) -c -O2 -fno-builtin -ffreestanding -std=c99.....
.....
BUILD_MK_BOOT_OBJS :=
BUILD_MK_HALY_OBJS :=
.....
BUILD_MK_BOOT_OBJS +=
BUILD_MK_HALY_OBJS +=init.o lmoosemhal_start.o interrupt.o intabtdistr.o.....
.....
.PHONY : all everything build_kernel
all: build_kernel
build_kernel:everything
everything :$(BUILD_MK_HALY_OBJS)
CCSTR = 'CC -[M] 正在构建... '$<
PRINTCSTR = @echo $(CCSTR)
%.o : $(CCBUILDPATH)%.c
    $(CC) $(CFLAGS) -o $@ $<
    $(PRINTCSTR)
%.o : $(CCBUILDPATH)%.S
    $(CC) $(CFLAGS) -o $@ $<
    $(PRINTCSTR)
.....

```

看到代码清单 4-14，一个完整的用于编译 LMOSEM 硬件相关层各代码模块的 makefile 文件就这样完成了，其中有了编译目录、编译命令、编译命令的选项、编译文件对象的列表，还有用于编译特定代码文件的规则。其他用于编译 LMOSEM 的其他组件的 makefile 也是同样的原理。由于篇幅所限，编译 LMOSEM 的内核层、编译 LMOSEM 的驱动程序等，就不再详细介绍了，读者能看懂这个，也应该能看懂其他的。

通过对 LMOSEM 的众多 makefile 文件的了解，并且看到了如何应用这许多的 makefile 文件组成一个结构良好的自动化编译环境，同时也加深了对 make 的认识，感受到了 make 的强大。但 make 远远不止如此，这不过是对 make 一点小小的应用，然而这已经够了，我们还有其他问题要探索，所以要对 make 说再见了。

4.3.2 LMOSEM 的链接脚本

编译器编译出众多的 LMOSEM 模块，这些模块是不能直接运行的，因为代码指令中的一些地址没有绑定，模块与模块之间可能存在互相调用，这就需要链接器来处理这些问题。这也是构建 LMOSEM 的最后一步——链接。

然而链接器的工作需要一个链接脚本，这个链接脚本也使得我们有机会告诉链接器，怎么链接这些目标文件，我们只需要按照要求，写好这个链接脚本就行了。

下面就来看看 LMOSEM 的链接脚本，前面分析 pretreatment.mk 文件时，就知道 LMOSEM 的链接脚本就是它调用 GCC 对 lmemknllink.S 文件预处理后产生的，首先粗略看一看 lmemknllink.S 文件是什么样的，如代码清单 4-15 所示。

代码清单4-15 生成LMOSEM链接脚本的lmemknllink.S

```
#include "buildfile.h"
INPUT(LINKR_IPUT_FILE)
OUTPUT(LINKR_OPOT_FILE)
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x30008000;
    __begin_kernel = .;
    __begin_lmosem_hal_head_text = .;
    .head.text ALIGN(4) : { *(.head.text) }
    __end_lmosem_hal_head_text = .;
    __begin_lmosem_hal_head_data = .;
    .head.data ALIGN(4) : { *(.head.data) }
    __end_lmosem_hal_head_data = .;
    .....
    __begin_lmosem_hal_intvect = .;
    .lmosem_hal.intvect ALIGN(4) : { *(.lmosem_hal.intvect) }
    __end_lmosem_hal_intvect = .;
    __begin_text = .;
    .text ALIGN(4) : { *(.text) }
    __end_text = .;
    __begin_data = .;
    .data ALIGN(4) : { *(.data) }
    __end_data = .;
    .....
    __begin_bss = .;
    .bss ALIGN(4) : { *(.bss) }
    __end_bss = .;
    __end_kernel = .;
    .....
}
```

代码清单 4-15 中只是链接脚本的几个片段，其中唯一要经过 GCC 预处理的只有 INPUT(LINKR_IPUT_FILE)、OUTPUT(LINKR_OPOT_FILE) 两行代码，INPUT、OUTPUT 中的两个宏定义在 buildfile.h 文件中，如代码清单 4-16 所示。

代码清单4-16 LINKR_IPUT_FILE、LINKR_OPOT_FILE宏

```
.....
#define BUILD_LINK_OBJS BUILD_BOOT_OBJS BUILD_HALY_OBJS\
    BUILD_KRNL_OBJS BUILD_MEMY_OBJS\
    BUILD_FSYS_OBJS BUILD_DRIV_OBJS\
```

```

BUILD_LIBS_OBJS BUILD_TASK_OBJS
#define LINKR_IPUT_FILE BUILD_LINK_OBJS
#define LINKR_OPUT_FILE lmosemkrnl.elf
.....

```

这就非常清楚了，LINKR_IPUT_FILE 会替换成 BUILD_LINK_OBJS，最终会替换成 LMOSEM 中所有被 GCC 编译的目标文件，而 LINKR_OPUT_FILE 会替换成 lmosemkrnl.elf。于是 LMOSEM 的链接脚本就变成如代码清单 4-17 所示。

代码清单4-17 LMOSEM链接脚本

```

INPUT( init.o lmosemhal_start.o interrupt.o intabtdistr.o halglobal.o halinit.o
.....)
OUTPUT(lmosemkrnl.elf)
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x30008000;
    __begin_kernel = .;
    __begin_lmosem_hal_head_text = .;
    .head.text ALIGN(4) : { *(.head.text) }
    __end_lmosem_hal_head_text = .;
    __begin_lmosem_hal_head_data = .;
    .head.data ALIGN(4) : { *(.head.data) }
    __end_lmosem_hal_head_data = .;
    __begin_lmosem_hal_init = .;
    .lmosem_hal.init ALIGN(4) : { *(.lmosem_hal.init) }
    __end_lmosem_hal_init = .;
    __begin_lmosem_hal_vector = .;
    .lmosem_hal.vector ALIGN(4) : { *(.lmosem_hal.vector) }
    __end_lmosem_hal_vector = .;
    __begin_lmosem_hal_intvect = .;
    .lmosem_hal.intvect ALIGN(4) : { *(.lmosem_hal.intvect) }
    __end_lmosem_hal_intvect = .;
    __begin_text = .;
    .text ALIGN(4) : { *(.text) }
    __end_text = .;
    __begin_data = .;
    .data ALIGN(4) : { *(.data) }
    __end_data = .;
    __begin_rodata = .;
    .rodata ALIGN(4) : { *(.rodata) *(.rodata.*) }
    __end_rodata = .;
    __begin_kstrtab = .;
    .kstrtab ALIGN(4) : { *(.kstrtab) }
    __end_kstrtab = .;
    __begin_bss = .;
    .bss ALIGN(4) : { *(.bss) }
}

```

```

        __end_bss = .;
        __end_kernel = .;
.....
}

```

从代码清单 4-17 中的链接脚本不难看出，生成 LMOSEM 的可执行文件是 ARM 小端 32 位的 ELF 格式，运行地址是从 0x30008000 开始的。LMOSEM 可执行文件中有如下各个段：

- 1) .head.text 段，存放最开始运行的指令。
- 2) .head.data 段，存放最开始运行的数据结构。
- 3) .lmosem_hal.init 段，包含 LMOSEM 硬件相关层的初始化指令。
- 4) .lmosem_hal.vector 段，包含 ARM 平台的中断向量。
- 5) .lmosem_hal.intvect 段，包含中断处理程序的指令。
- 6) .text 段，包含所有目标文件的默认指令段。
- 7) .data 段，包含所有目标文件的默认数据段。
- 8) .rodata 段，包含所有目标文件的默认只读数据段。
- 9) .kstrtab 段，包含所有目标文件的默认字符串表段。

10) .bss 段，包含所有目标文件的默认未初始化或者初始化为 0 的数据段，它不占用可执行文件的空间。

LD 链接器也一定会按照上述段的顺序合成 LMOSEM 可执行文件，如 LMOSEM 可执行文件最开始的地方放的是 .head.text 段，然后放的是 .head.data 段，依次类推。诸如 __begin_XXXX、__end_XXXX，这些符号对链接脚本来说不是必须的，但是后面会用到它们，以确定内核各段的大小，后面在使用它们的地方会有介绍。

关于 LMOSEM 的链接脚本就介绍到这里，非常简单，如果不能一下子全明白可以结合前面 LD 链接器一起看，应该不难。当然也可以参阅 LD 链接器的官方手册。不过这些内容对于 LMOSEM 已经是足够了。

4.4 开发板的安装

操作系统内核编译好了，然后就是测试，以证明操作系统内核的健壮性。操作系统内核的测试必须需要一套计算平台。

读者可能想到了虚拟机，虚拟机用于测试操作系统内核固然好，可是虚拟机并不能仿真硬件的每个特性，如时钟硬件，虚拟机能仿真，但不一定像真实时钟一样精确，并且虚拟机可能有 Bug，但是虚拟机确实能完美地运行现有的操作系统，因为虚拟机的开发者大部分情况下总是用现有的操作系统测试他们的虚拟机，现有的操作系统可能并没有用到计算平台上特定硬件的所有特性，并且我们的操作系统内核的代码绝对不可能和现有的操作系统的代码完全相同。所以用虚拟机测试我们的操作系统内核，很有可能出现未知的问题，这个问题直接导致你会怀疑你的代码，事实上那是虚拟机的问题。

由上可知,我们应该用真实的物理硬件去运行我们的操作系统内核。前面就写到,是用 mini2440 开发板作为开发平台的,下面就去探讨怎么安装和使用它。

前面章节中就了解了 mini2440,现在来看看怎么安装它并且怎么和 PC 进行连接,以便下载操作系统内核到 mini2440 中运行,达到测试操作系统内核的目的。

安装 mini2440 其实很简单,无非就是连接上电源、连接上串口线,连接上 USB 下载线。

先来看看 mini2440 提供上述连接的接口,如图 4-15 所示。

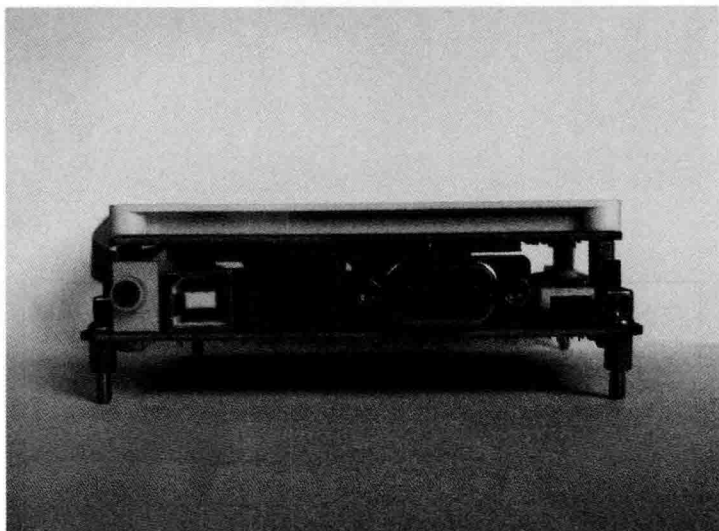


图 4-15 mini2440 的接口

图 4-15 中从左至右依次是音频接口、USB 接口、网络接口、串口、电源接口。其中音频接口和网络接口我们并不关心。

由于现代 PC 桌面机和笔记本电脑上可能都已经没有串口硬件了,所以需要一个额外的部件:串口转 USB 接口部件,如图 4-16 所示。该部件接受串口数据然后通过 USB 接口发送到 PC,PC 的操作系统中有专门处理这个设备和其中相关数据的驱动程序。

依次连接上电源线、USB 下载线、串口线及串口转 USB 接口部件,如图 4-17 所示。

按照图 4-17 中连接好后,把 USB 下载线和串口转 USB 接口部件分别插入 PC 上两个不同的 USB 端口,然后把电源插入交流电源的插座中就行了。

上面配置了硬件,下面就要去配置软件了,在 Linux 环境下需要如下软件:

- 1) minicom, 主要显示串口数据的,前面已经安装了。
- 2) dnw2, LMOSEM 代码中自带的一个工具,用于下载内核到 mini2440 中的。
- 3) libusb-dev, 一个 USB 库 dnw2 需要通过它识别 mini2440。同样在前面也已经安装了。

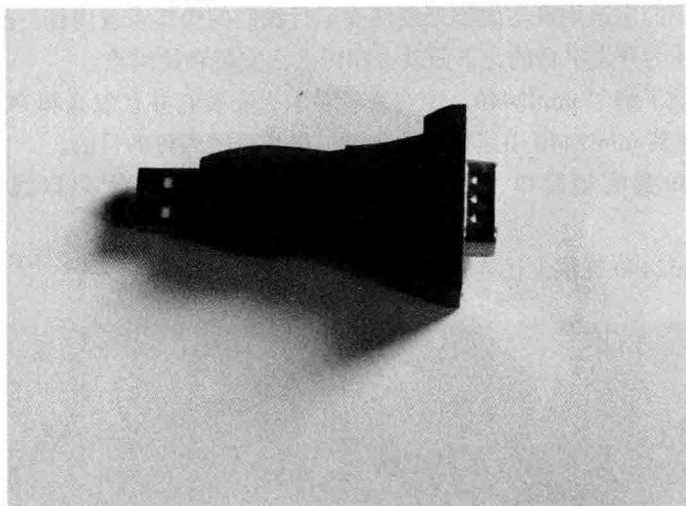


图 4-16 串口转 USB 接口部件

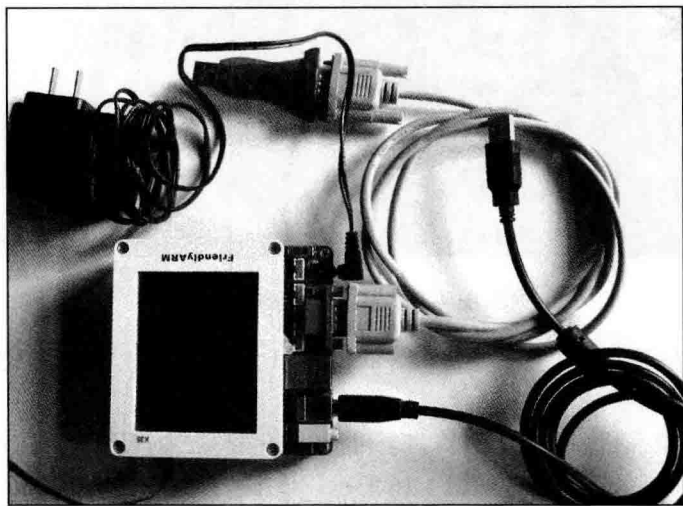


图 4-17 连接好各类线缆的 mini2440

4) USB 转串口设备驱动, 用于获取 USB 转串口设备的数据, 见图 4-16。大部分情况下 Linux 内核中都自带这个驱动, 不需要额外安装。

minicom 这个软件需要配置一下, 因为它必须要知道 USB 转串口的设备文件和设置一下串口的通信速率。Linux 内核把设备管理纳入文件系统了, 所有设备一旦插入计算机并装载该设备驱动后, 就会在文件系统的“/dev/”目录下创建一个设备文件, 拔掉该设备就会删掉这个设备文件。USB 转串口设备的文件名一般是由“ttyUSB”开始后面接一个数字命名的, 读者的计算机中可能有多个 USB 转串口设备, 请多拔插几下 USB 转串口设备, 观察

“/dev/”目录下文件的变化,以确定 USB 转串口设备的设备文件,确定之后常用这个 USB 插口就行了。笔者机器上的 USB 转串口设备的设备文件名是 ttyUSB0。

minicom 这个软件的配置很简单,步骤如下:

第 1 步:首先在终端中输入命令: `sudo minicom -s` 回车。

第 2 步:选择 Serial port setup, 如图 4-18 所示。

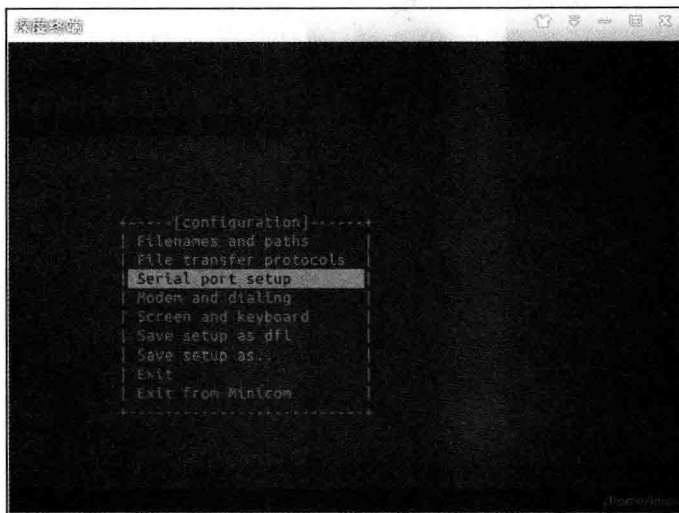


图 4-18 配置 minicom 步骤二

第 3 步:输入 a 配置 USB 转串口设备的设备文件名,输入 /dev/ttyUSB0, 如图 4-19 所示。

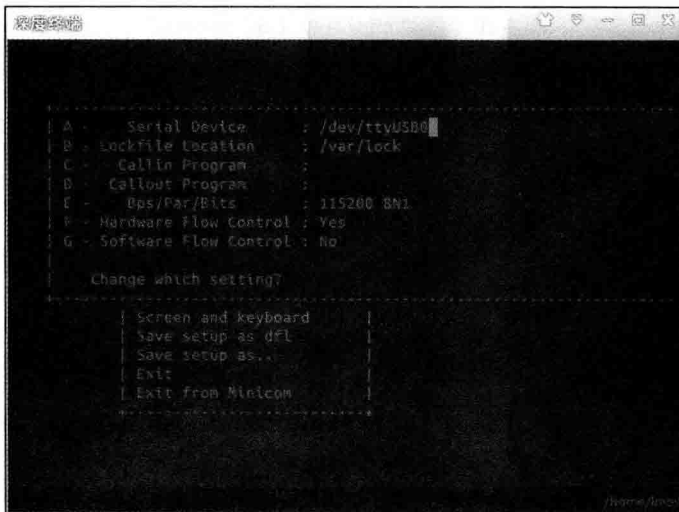


图 4-19 配置 minicom 步骤三

第4步：输入两次回车后，返回 minicom 的主菜单选择 Save setup as dfi，然后回车。

第5步：选择 Exit 退出 minicom 的配置主菜单。

由上述第3)步中，发现 minicom 默认情况下，串口数据通信速率就是 115200，这正好与我们开发板上的串口通信速率是吻合的。

上面配置了 minicom，现在就可以启动 mini2440 开发板了，开发板上一共有两个开关，一个是电源接口这边的，这个是电源开关。一个是音频接口这边的，这个是启动方式的跳线开关。先把启动方式的跳线开关拨向 NOR 端，这是启动 mini2440 开发板上的引导程序，相当于 PC 的 BIOS。然后打开电源开关，mini2440 开发板就启动了，如图 4-20 所示。

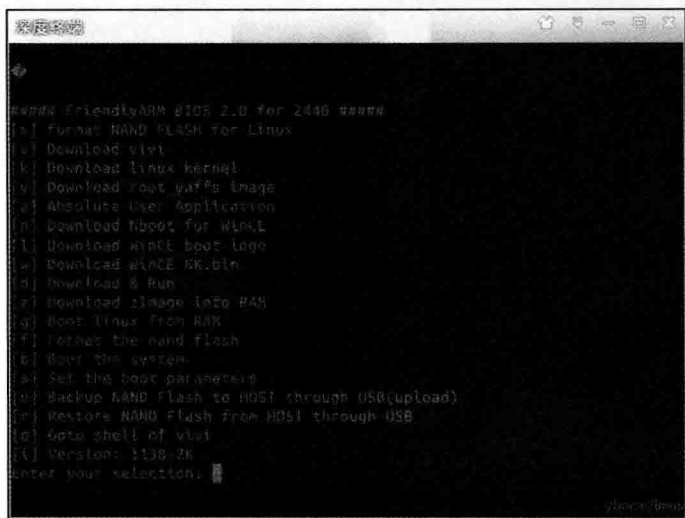


图 4-20 启动 mini2440 上的 supervivi

mini2440 开发板的安装和配置，非常简单，不必过多在此浪费时间，不过要相信，现在所做的一切，都是为了后面我们开发 LMOSEM 时尽可能少地遇到麻烦，这并不是徒劳无功的行为。

4.5 小结

本章就要结束了，这是让人兴奋的事情。这章主要了解了操作系统内核设计、操作系统内核开发环境和开发工具相关的内容。

1) 操作系统内核的设计，介绍了内核需要什么功能；关于内核的一些架构，介绍了两种内核架构的优点和缺点；考虑了内核将来的可移植性，这主要通过分离硬件相关性而达到；最后利用这些知识点设计了我们的操作系统内核。

2) 开发环境及相关工具，从 Linux 环境开始，介绍了一些开发中要用到的工具，如

GCC、LD、make。

3) LMOSEM 的构建系统, 主要介绍了两个例子 LMOSEM 的 makefile 和 LMOSEM 的链接脚本, 通过它们组成了 LMOSEM 的构建系统。

4) 开发板的安装, 安装并测试了 mini2440 开发板, 用它作为 LMOSEM 的运行平台。

我们设计了操作系统内核, 了解了开发工具, 后面的内容就轻松多了, 我们正一步一步地完成这些开发前的准备工作。

语言间调用约定与基本数据结构

“为什么还没有写到操作系统内核？”笔者仿佛看到你火冒三丈地跳了起来，不过也别太着急了，这章内容不多。

开发操作系统内核一般情况下会用到两种编程语言：汇编和 C 语言。汇编有很多优势是其他高级编程语言不具备的，如能操控底层硬件、代码运行效率高等。C 语言则提供良好的可移植性、高效的编程效率，并且拥有许多数据类型，但 C 语言无法操作底层的硬件。所以让这两种语言结合起来，无疑是非常好的选择。

既然两种不同的语言结合起来开发，它们之间必然会互相调用对方的子程序和访问对方的数据。要互相调用、互相访问，就得有一定的规则，如寄存器的使用约定、参数如何传递、数据结构在内存中存在的形式等，这些是首先要搞清楚的东西。另外，得了解一些操作系统内核中最常用的基本数据结构，以及如何用它们构建更为复杂的数据结构。最后了解如何在 C 函数中直接嵌入汇编，因为这种方法非常方便实用，在开发操作系统内核过程中经常用到。

了解以上内容对写操作系统内核至关重要，这也是本章存在的意义。

5.1 寄存器使用约定

处理器都会提供一定数量的寄存器，用于完成各种计算任务。然而汇编和 C 语言结合，那么就要明白 C 语言中对某一特定处理器中的一些关键寄存器的使用和命名，以及 C 语言编译一个个 C 函数时，用什么方式传递参数和如何传递返回值。了解了这些内容才能让汇编和 C 语言的代码互相调用。

5.1.1 寄存器别名

寄存器在机器指令中仅是一个编码而已，处理器手册上的 R0、R1、R2 等也仅是说明这是第几个寄存器，这对软件开发人员帮助有限，因为软件开发人员最希望的是见其名知其意。

综上所述，为了见其名知其意，所以在汇编代码中出现了许多寄存器别名，如在我们所熟悉的 ARM 汇编代码中，R13 有时会写成 SP、R14 有时会写成 LR、R15 有时会写成 PC 等。有时它们也可以混合使用，如在此处写的是 SP，下一处写的可能是 R13，这完全取决于编程者或者产生汇编代码的其他语言的编译器。

来看看 ARM 体系的汇编代码中的寄存器别名表，如表 5-1 所示。该表中表示的只是通用寄存器，一些特殊寄存器是没有别名的，如 CPSR、SPSR 等。

表 5-1 寄存器别名

寄 存 器	别 名	寄 存 器	别 名
R0	A1	R8	V5
R1	A2	R9	V6、sb
R2	A3	R10	V7、sl
R3	A4	R11	V8
R4	V1	R12	IP
R5	V2	R13	SP
R6	V3	R14	LR
R7	V4、wr	R15	PC

表 5-1 中，有些寄存器有多个别名，事实上它们表示的是同一个寄存器，例如，在汇编代码中既可以写 R10，也可以写 V7 或者写 sl，这都是能够被汇编器所识别的。之所以会有这些别名，是为了让开发人员更加容易阅读代码，这对编译器，对机器是没有任何意义的。这里写到寄存器别名，只是为了在后面的代码中如果用到了一些寄存器的别名，我们不至于感到奇怪。

5.1.2 参数传递与返回值

写 C 语言程序，都会定义并实现一些函数。在 C 语言中所谓的函数不过就是一段代码，这段代码一般有三个部分：① 保存栈指针和一些寄存器的值，在栈中分配函数中的局部变量所占用的存储空间以及获取函数的参数等；② 用户完成功能的实际代码；③ 处理函数返回值，恢复一些寄存器中的值并平衡栈空间等，最后加上函数返回指令。即使写一个 C 语言的空函数也会有第一部分和第三部分，这是由 C 语言编译器自动完成的。

因此不难发现，C 语言函数的参数怎么传递以及参数放在哪里，最后又是怎么传回返回

值的。编译器的开发者可能有很多解决方法，但是一种编程语言要和其他编程语言协作，那么就得有一套标准来处理函数的参数和返回值，当然这仅只是一个方面，它还包括一些可链接二进制文件方面的规定。目前只需要关注函数的参数和返回值方面的相关规定。

不同的硬件平台上，这个标准也是不同的，在 ARM 平台上这个标准是 ATPCS。下面来看看表 5-2，它总结了 ATPCS 规定的参数传递与返回值的相关标准。

表5-2 ATPCS标准下相关寄存器的使用说明

寄存器	作用	相关说明
R0 ~ R3	传递参数	函数通过寄存器 R0 ~ R3 来传递参数；这些寄存器可以记作 A1 ~ A4，被调用的函数在返回前不需要恢复寄存器 R0 ~ R3 的值。当函数的参数个数超过 4 个时，多余的参数就用栈来传递
R4 ~ R11	局部变量	在函数中，使用 R4 ~ R11 寄存器来保存局部变量。也可以记作 V1 ~ V8。如果在函数中使用到 V1 ~ V8 中的某些寄存器，函数进入时必须保存这些寄存器的值，在返回前必须恢复它们的值，没用到的寄存器则不必执行任何操作
R12	函数内部调用	函数内部调用其他函数会用到。这个不必深究。后面也不会涉及它，C 语言编译器会自动处理
R13	栈指针	其中保存栈顶地址
R14	程序返回地址	其中保存函数的返回地址，如果函数用其他方法保存了返回地址，则这个寄存器可以用于其他用途
R15	程序指针	R15 中是处理器当前要执行指令的地址，所以不得用于其他用途
R0	返回值	返回值为一个 32 位的整数时，可以通过寄存器 R0 返回。返回值为一个 64 位整数时，可以通过 R0 和 R1 返回，R0 中保存低 2 位，R1 中保存高 32 位

下面来看一个实例，用 C 语言写个函数，如代码清单 5-1 所示。

代码清单5-1 验证ATPCS的C代码

```
int addfunc(int v11, int v12, int v13, int v14, int v15,int v16)
{
    return ((v11+v12)+(v13+v14))+(v15+v16);
}
int test()
{
    return addfunc(1,2,3,4,5,6);
}
```

用命令 arm-linux-gcc -S -O1 -fno-builtin -ffreestanding -std=c99 -march=armv4 -fno-stack-protector -fomit-frame-pointer -fno-ident -mtune=arm7tdmi -o addfunc.s addfunc.c 编译代码清单 5-1。该命令是让 GCC 生成该代码的汇编代码，而不是直接产生二进制文件，其结果如代码清单 5-2 所示。

代码清单5-2 验证ATPCS的汇编代码

```
.....
addfunc:
```

```

add    r1, r1, r0
add    r1, r1, r2
add    r1, r1, r3
ldr    r2, [sp, #0]      @从栈中取出第五个参数, 值为5
add    r0, r1, r2
ldr    r2, [sp, #4]      @从栈中取出第六个参数, 值为6
add    r0, r0, r2        @最终结果被加入到R0寄存器中, 作为返回值
bx     lr

test:
str     lr, [sp, #-4]!    @把lr保存到SP-4的位置, 即栈顶减4
sub     sp, sp, #12      @SP=SP-12, 在栈中分配一块空间
mov     r3, #5
str     r3, [sp, #0]     @把5放入当前SP指向的位置
mov     r3, #6
str     r3, [sp, #4]     @把6放入当前SP+4指向的位置。这当然是为了处理参数
mov     r0, #1           @把1、2、3、4分别放入R0~R4寄存器中, 也是为给addfunc函数传
                        递参数

mov     r1, #2
mov     r2, #3
mov     r3, #4
bl      addfunc          @参数处理好了, 调用addfunc函数。函数执行后R0中保存着返回结果
add     sp, sp, #12      @平衡栈空间
ldr     lr, [sp], #4     @读出原来LR寄存器中的值, 放入LR中
bx      lr               @test函数返回

```

代码清单 5-2 是 GCC 编译器把两个 C 语言函数编译成的 ARM 平台的汇编语言代码。回到我们关注的问题：程序中是如何传递参数和处理返回值的。从代码清单 5-2 中，不难看出 test 函数是怎么给 addfunc 函数传递参数的。它首先发现 addfunc 函数有 6 个参数，用 4 个寄存器传递不够了，于是它把后两个参数按从右至左的顺序依次压入栈中，然后把前 4 个参数按从左至右的顺序依次放入 R0 ~ R3 这 4 个寄存器中。代码清单 5-2 中的注释一看便知。

上面用 GCC 编译出代码清单 5-1 的 ARM 平台的汇编语言代码清单 5-2，正好证明了 ATPCS 规定的函数参数传递与处理函数返回值的标准。了解了这个标准，C 语言和汇编语言的函数互相调用就容易多了，以后就可以按此标准，用汇编语言写可以被 C 语言调用的函数，也可以从汇编语言中调用带参数的 C 函数，一切都变得很轻松。

5.2 基本数据结构

Niklaus Wirth 说过：“程序=算法+数据结构”。因为这句话他获得了 1984 年的图灵奖。也许 C 的教材会教会你 C 语言，但这离用 C 语言写出像 UNIX 那么伟大的操作系统还差得很远。所以不要奇怪为什么学会了某种编程语言却写不出有用的程序。因为重要的是数据结构和算法，尤其是对于操作系统内核这样的程序而言。

笔者在写了两个操作系统内核雏形后，有个感受，开发操作系统内核首先根据需要设计

不同的数据结构，然后写代码来操作这些数据结构，最后将这些代码和数据结构合理地结合在一起。所以明白 C 语言的各种基本类型在不同情况下占用的位宽、内存中对齐的方式，以及如何用它们构建操作系统中最基本的数据结构：链表、自旋锁、等待链、信号量，到最后能用这些基本的数据结构构建更为复杂的数据结构，这对开发操作系统内核非常重要。

5.2.1 C 语言的基本数据结构

大可以这样认为：C 语言中的基本数据类型就是一种数据结构，是最原始的数据结构。C 语言提供了丰富的数据类型，但是开发操作系统内核却只用到了几个常用的数据类型。然而在不同的 C 编译器、不同的编译选项、不同的主操作系统（运行编译器的操作系统）环境下，这些常用的数据类型占用内存的大小、位宽，都会不同。这对应用程序的开发，或许没有什么问题，但是对于操作系统内核的开发，在某些情况下却是致命的。

我们用的 Linux 系统上的 ARM 交叉 C 编译器（ARM-GCC）的数据类型的属性，如表 5-3 所示。该表只显示了开发操作系统内核所用到的数据类型。

表5-3 ARM交叉C编译器的数据类型及其属性

基本类型	位宽	占用内存大小	该类型的指针位宽及大小	用该类型的指针(*p)一次能读取数据的大小	有/无符号及最大/最小能表示数据的范围
unsigned char	8 位	1 字节	32 位 4 字节	1 字节	无、0 ~ 2 ⁸ -1
unsigned short	16 位	2 字节	32 位 4 字节	2 字节	无、0 ~ 2 ¹⁶ -1
unsigned int	32 位	4 字节	32 位 4 字节	4 字节	无、0 ~ 2 ³² -1
unsigned long	32 位	4 字节	32 位 4 字节	4 字节	无、0 ~ 2 ³² -1
unsigned long long	64 位	8 字节	32 位 4 字节	8 字节	无、0 ~ 2 ⁶⁴ -1
char	8 位	1 字节	32 位 4 字节	1 字节	有、-2 ⁷ ~ 2 ⁷ -1
short	16 位	2 字节	32 位 4 字节	2 字节	有、-2 ¹⁵ ~ 2 ¹⁵ -1
int	32 位	4 字节	32 位 4 字节	4 字节	有、-2 ³¹ ~ 2 ³¹ -1
long	32 位	4 字节	32 位 4 字节	4 字节	有、-2 ³¹ ~ 2 ³¹ -1
long long	64 位	8 字节	32 位 4 字节	8 字节	有、-2 ⁶³ ~ 2 ⁶³ -1

在很多情况下，操作系统内核中有许多数据结构对数据成员的位宽有严格的要求，又或者操作系统内核代码需要在不同的 C 编译器上编译。不同的 C 编译器也许对基本的数据类型的位宽及大小处理不同，如果在内核代码中直接使用这些基本类型是非常可怕的，于是我们这样处理，如代码清单 5-3 所示。

代码清单5-3 定义数据类型别名

```
typedef unsigned char u8_t;
typedef unsigned short u16_t;
```

```

typedef unsigned int u32_t;
typedef unsigned long long u64_t;
typedef char s8_t;
typedef short s16_t;
typedef int s32_t;
typedef long long s64_t;
typedef long sint_t;
typedef unsigned long uint_t;
typedef u32_t cpuflg_t;
typedef unsigned long adr_t;
typedef sint_t bool_t;
typedef u32_t dev_t;
typedef const char* str_t;
typedef char char_t;
typedef sint_t drvstus_t;
typedef sint_t sysstus_t;
typedef sint_t hand_t;
typedef void* buf_t;
typedef unsigned long size_t;
typedef u32_t reg_t;

```

代码清单 5-3 中, typedef 关键字的作用是定义类型的别名, 例如, 定义 int 的别名为 s32_t, 注意类型别名和普通的宏替换有本质的区别, 这里并不深究有何区别。例如, 用 reg_t 定义用于存放 CPU 寄存器值的变量比用 int 定义更加直观, 因为我们所用的处理器上的寄存器是 32 位的, 如果哪天处理器升级了, 寄存器变成 64 位了, 那么就只需要在此处改一行代码即可。用一种数据类型定义一个变量, 就是为了存放相应的数据的。如果在代码中, 有的数据对数据的位宽有严格的要求, 就可以使用带数字的这些类型, 如 u32_t、u16_t、u64_t、s8_t、s32_t 等。还有一些数据对位宽没有要求, 但是需要随着编译操作系统内核选项的变化而变化的, 如把操作系统内核编译成 64 位时它们就要变成 64 位的, 编译成 32 位时它们就要变成 32 位的。这种情况我们就用 sint_t、uint_t, 因为 long 类型对于 GCC 编译器生成 64 位代码时, 它就是 64 位的, 生成 32 位代码时它就是 32 位的。

经过上面的处理, 不难发现 LMOSEM 的代码更加直观了, 也更加容易在不同的编译器上编译了, 同时读者也会看到 LMOSEM 的可移植性、可扩展性、可维护性就是从这些不起眼的小细节开始的。

5.2.2 list_h_t 数据结构

链表是一种非常常用的数据结构, 特别是在动态创建相应数据结构的情况下更是如此, 然而在操作系统内核中, 动态创建相应的数据结构尤为频繁。

但是下面看到的这种链表, 不同于数据结构课程里讲的链表。下面这种链表是通用的, 在任何情况下, 只要需要链表的数据结构包含它就行了, 当然这种技术不是笔者开发出来的, 笔者参考了 Linux 内核中链表的实现原理, 有些基本的数据结构在所有的操作系统内核

下都是相通的，这不能算是抄袭。

来看看这个链表数据结构，如代码清单 5-4[xxxx/lmosem/include/bastypeinc/list_t.h] 所示。可以很清楚地看到数据结构中的两个指针都指向其自身。

代码清单5-4 链表数据结构

```
Typedef struct s_LIST_H {
    struct s_LIST_H *prev,*next;
}list_h_t;
```

怎么使用这种链表呢，只要在需要链表的数据结构中包含它就行了。看个简单的例子就明白了，如代码清单 5-5 所示。

代码清单5-5 dastuc_t数据结构

```
typedef struct s_DASTUC {
    int ds_stus;
    int ds_type;
    list_h_t ds_list;
    int ds_count;
}dastuc_t;
```

在代码清单 5-5 中，dastuc_t 数据结构中包含了一个链表数据结构类型的成员变量 ds_list。它就是起中间链接的作用，在中间像一根链子把众多的 dastuc_t 数据结构串起来，如图 5-1 所示。

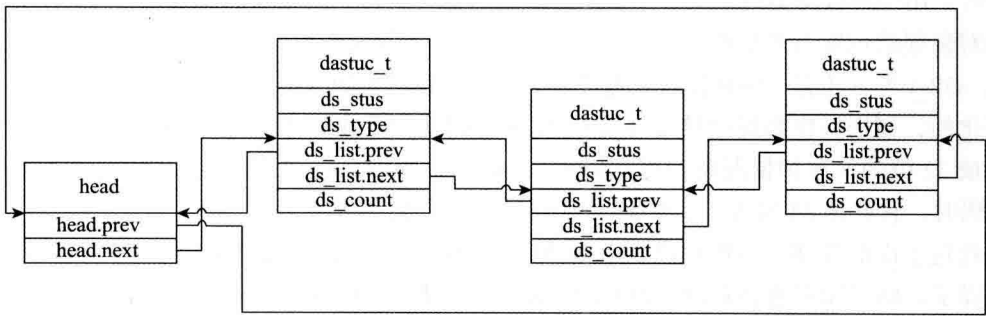


图 5-1 list_h_t 链表的使用

从图 5-1 中发现，list_h_t 是个循环双向链表。还可以发现只需要写一套通用的代码就可以操作这个链表，并且在操作系统内核的所有代码中，这套操作链表的代码都是通用的。如代码清单 5-6 [xxxx/lmosem/include/bastypeinc/list.h] 所示，其中有初始化链表、添加链表、删除链表、遍历链表等相应的链表操作。

代码清单5-6 操作链表的通用代码

```
//初始化链表list
//KLINE是个宏，会替换为static inline，这是告之GCC把该函数处理为内连函数
```

```

KLINE void list_init(list_h_t* list)
.....
//把链表new添加到链表头head上
KLINE void list_add(list_h_t *new, list_h_t *head)
.....
//把链表entry删除
KLINE void list_del(list_h_t* entry)
.....
//这是一个宏，用于遍历链表头head上所有的链表即list_h_t
#define list_for_each(pos, head) for (pos = (head)->next; pos != (head); pos =
pos->next)

```

到这里，读者可能会思考，这些代码仅仅是操作链表本身，链表里的指针指向的也是链表本身，而我们关注的重点应该是包含这个链表（list_h_t）的数据结构。不难发现，只要通过链表的地址获取链表所在的数据结构的地址就行了，在C语言中我们需要采用额外的手段来解决这个问题，如代码清单5-7[xxxx/lmosem/include/bastypeinc/list.h]所示。

代码清单5-7 获取链表所在的数据结构的地址

```

#define list_entry(ptr, type, member) \
((type *)((char *) (ptr) - (unsigned long) (&((type *)0)->member)))

```

list_entry 是一个宏，怎么使用呢，还是用先前那个 dastuc_t 作为实例，如要获取挂在链表头上的 dastuc_t 的首地址，如下所示。

```
dastuc_t* dsp=list_entry(head.next,dastuc_t,ds_list);
```

代码经过C编译器预处理后如下所示。

```
dastuc_t* dsp=((dastuc_t*)((char*)(head.next)-( unsigned long) (&((dastuc_t*)0)->ds_list)));
```

在上面的代码中，(&((dastuc_t*)0)->ds_list)) 语句是最关键的，也是最难理解的，它是让GCC以0地址为基准计算ds_list在dastuc_t数据结构中的地址，也可以说是偏移量。上述情况中，从0地址开始计算ds_list在dastuc_t中的偏移量为8。然后把这数据强制为无符号的long类型，再把head.next，即指向dastuc_t中ds_list的地址强制成char*类型，最后用它减去那个偏移量，再把结果强制成dastuc_t*类型，这个结果就是dastuc_t数据结构的首地址。

看个图可能比上面的文字更容易理解，如图5-2所示。

图5-2中，head.next等于0x1008，经过list_entry宏就能精准地计算出dastuc_t数据结构的首地址为0x1000。任何一个数据结构定义好了之后，其中list_h_t在这个结构中的偏移量在程序的运行过程中是不会改变的。

一个小小的链表，竟然如此精妙，同时也看到了Linux内核的开发者是非常有实力的。对于我们，唯有学习之。

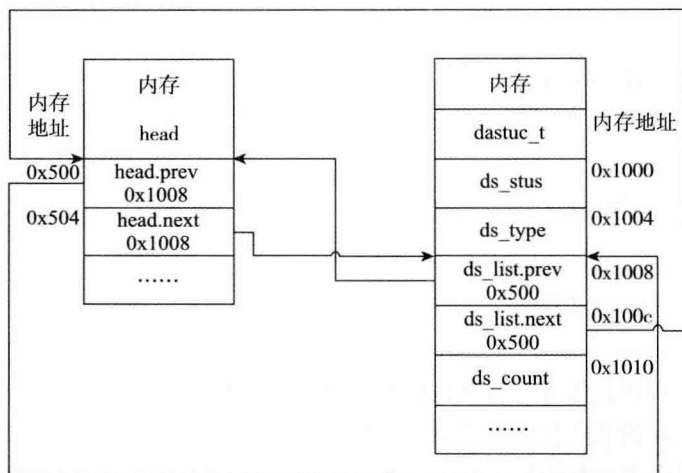


图 5-2 list_entry 宏的原理

5.2.3 spinlock_t 数据结构

操作系统内核中有许多数据是全局的，有些全局的数据是所有的进程都可以访问和修改的，例如，表示内存剩余量的数据，每次分配内存后都会修改它，以表示当前剩余内存量。然而问题来了，如内存还有最后 1KB，一个进程调用内存分配接口，分配了这最后 1KB 内存，当分配函数还没有改写表示内存剩余量的数据时，一个外部事件发生了，操作系统内核可能调度另一个进程运转，另一个进程也需要分配内存，它读取表示内存剩余量的数据，发现还有最后 1KB 内存（其实已经没有了，已经被先前的进程分配了，只是没有来得及更新表示内存剩余量的数据）。显然这里出现了错误，并且这种错误终将引发灾难性的后果。

如果上述问题在单 CPU 下，解决方法很简单——关掉外部事件源（通常为中断）。如果在多 CPU 的硬件平台上，解决方法就复杂了，因为每个 CPU 都可以运行进程。关掉一个 CPU 上的中断，另几个 CPU 上的中断依然是开启的，并且其他 CPU 可能还运行着其他进程，它们依然有可能访问全局数据。我们的 LMOSEM 是支持多进程、多 CPU 的操作系统内核，所以那就得解决这个问题。

在多 CPU 下解决这类问题的方法是用自旋锁 `spinlock_t`，自旋锁的原理很简单，即每次访问全局数据时就执行如下步骤。

1) 首先要加锁。加锁的动作如下。① 检测锁值是否为 0，同时把锁值设为 1。这是个交换的动作，这个动作是原子执行的，即这个动作要么执行完成，要么不执行。几乎每种 CPU 都有完成这种功能的指令。② 如果第一步中交换回的值是 1，说明已经加锁了，这时回到第一步继续执行那个交换动作，直到有一次换回的值是 0。这也是叫自旋锁的原因。

2) 一旦加锁成功，就开始访问和操作数据。

3) 最后解锁。解锁的动作如下：把锁值置为 0。

明白了自旋锁的原理，下面来看看 LMOSEM 的自旋锁数据结构的设计。如代码清单 5-8 [xxxx/lmosem/include/bastypeinc/spinlock_t.h] 所示。

代码清单5-8 自旋锁数据结构

```
typedef struct
{
    volatile u32_t lock;
} spinlock_t;
```

所谓自旋锁不过就是一个 32 位的整型变量，前面的 `volatile` 关键字是告诉 GCC 不要对这个变量进行优化，在编译时不要把它分配到 CPU 寄存器中，每次访问它时都必须从内存中读取。lock 成员的值初始时应该为 0。

一般情况下使用自旋锁是这样的：直接把 `spinlock_t` 这个数据结构包含在需要自旋锁保护的其他数据结构中。例如，代码清单 5-9 中。在相应的代码中每次访问 `mem_t` 中的其他成员时，都要首先对 `m_lock` 自旋锁执行加锁操作，然后才能访问，加锁操作不成功的情况下则原地自旋等待其他代码对 `m_lock` 自旋锁执行解锁操作，最后访问完 `mem_t` 中的其他成员时，要对 `m_lock` 自旋锁执行解锁操作。

代码清单5-9 自旋锁的运用

```
typedef struct s_MEM
{
    spinlock_t m_lock;
    uint_t m_start;
    uint_t m_end;
    size_t m_size;
    .....
}mem_t;
```

LMOSEM 中操作自旋锁的接口如代码清单 5-10 [xxxx/lmos_em/include/halinc/cpuctrl.h] 所示。

代码清单5-10 操作自旋锁的接口函数

```
void hal_spinlock_init(spinlock_t* lock); //自旋锁初始化
void hal_spinlock_lock(spinlock_t* lock); //自旋锁加锁
void hal_spinlock_unlock(spinlock_t* lock); //自旋锁解锁
```

操作系统内核中不只是前面提到的分配内存的例子，其实操作系统内核中大部分数据结构都存在并发访问的问题，所以在设计其他数据结构时，大部分情况下都会包含一个 `spinlock_t` 数据结构，以起到保护作用，所以在此首先了解 `spinlock_t`，因为它是保证其他数据结构正确性和一致性的基石。

5.2.4 kwlst_t 数据结构

`kwlst_t` 这个数据结构非常简单，一般情况下也不会单独使用，它总是和其他数据结构一

起工作。

一个非常简单的情境。一个进程要获取键盘上的输入信息，然而这时用户在一段时间内没有按下任何按键，那么这个进程该怎么办呢？这时操作系统内核会让该进程进入等待状态，调度其他进程运行，等用户按下键盘上的按键时重新激活该进程。事实上，这种“等待”在系统中随处可见：等待磁盘、等待网络，等待一切需要等待的资源。所以进程在系统内部总是“走走停停”。既然进程要等待，那么当等待的事件或者资源满足时就需要重新激活它，这就带来了一个最基本的问题：等待的进程放在哪里，当等待的事件或者资源满足时如何找到它，其他进程也等待同一事件或资源该怎么办。所以需要设计一个数据结构，如代码清单 5-11 [xxxx/lmosem/include/bastypeinc/sem_t.h] 所示。

代码清单5-11 进程等待数据结构

```
typedef struct s_KWLST
{
    spinlock_t wl_lock;           //支持多CPU的OS内核下的数据结构需要有自旋锁保护
    uint_t     wl_tdnr;           //记录有多少个等待的进程
    list_h_t   wl_list;           //把所有等待的进程挂入这个链表
}kwlst_t;
```

这个数据结构非常简单，其中的 `wl_lock` 是为了保护 `kwlst_t` 其自身。该数据结构相当一个容器，把所有等待相同事件或资源的进程都归纳到这个容器中，保证以后操作系统内核代码能够从这里找到等待的进程，以便能够重新激活它们。

到这里，了解了 `kwlst_t` 数据结构，同时也发现了它就是由 `spinlock_t`、`list_h_t` 外加一个 `uint_t` 整型变量构建出来的，这时我们就去看看如何使用它构建更为高级的数据结构。

5.2.5 sem_t 数据结构

看过操作系统理论书籍的读者应该都知道信号量，信号量是 1965 年荷兰学者 Edsger Dijkstra 提出的，是一种用于资源互斥或者进程间同步的机制。这里就来看看实现这一机制的数据结构。

还是先前那个情境：一个进程要获取键盘上的输入信息。下面就去看看操作系统内核完成这个功能一般方法。当然具体实现中可能不同，但是原理是相通的，如下：

- ❑ 一块内存，相当于缓冲区，用于保存键盘的按键码。
- ❑ 需要一套控制机制，如进程读取这个缓冲区，而该缓冲区为空时怎么处理；该缓冲区中有了按键码却没有进程来读取，又该怎么处理。

我们期望是这样的：

- ❑ 进程获取键盘输入信息时，发现键盘缓冲区中是空的，就进入等待状态。
- ❑ 同一时刻，只能有一个进程操作键盘缓冲区。
- ❑ 当用户按下键盘时，我们有能力把按键码写入缓冲区中，并且能看一看有没有等待该缓冲区的进程，如果有就重新激活它们，让它们重新竞争读取键盘缓冲区，竞争

失败的进程依然进入等待状态。

其实以上所述无非有三个问题：等待、互斥、唤醒即重新激活等待的进程。这就需要一种全新的数据结构来解决这些问题。根据上面的问题，这个数据结构至少需要一个变量来表示互斥，如大于0则进程可以继续运行，等于0则让进程进入等待状态。还需要一个等待链，用于保存等待的进程。该数据结构如代码清单5-12 [xxxx/lmosem/include/bastypeinc/sem_t.h]所示。

代码清单5-12 信号量数据结构

```
typedef struct s_SEM
{
    spinlock_t sem_lock;    //支持多CPU的OS内核下的数据结构需要有自旋锁保护
    uint_t sem_flg;         //该信号量的一些标志和类型
    sint_t sem_count;       //该信号量的信号值
    kwlst_t sem_waitlst;    //该信号量的等待链，用于保存等待的进程
}sem_t;
```

再看看信号量的一般用法，注意信号量在使用之前需要先进行初始化。这里假定信号量数据结构中的 `sem_count` 初始化为1、`sem_waitlst` 等待链初始化为空。使用信号量的步骤如下：

1) 获取信号量。①首先对用于保护信号量自身的自旋锁 `sem_lock` 进行加锁。②对信号值 `sem_count` 执行“减1”操作，并检查其值是否小于0。③上步中检查 `sem_count` 如果小于0，就让进程进入等待状态并且将其挂入 `sem_waitlst` 中，然后调度其他进程运行。否则表示获取信号量成功。当然最后别忘了对自旋锁 `sem_lock` 进行解锁。

2) 进程开始执行相关操作，如读取键盘缓冲区。

3) 释放信号量。①首先对用于保护信号量自身的自旋锁 `sem_lock` 进行加锁。②对信号值 `sem_count` 执行“加1”操作，并检查其值是否大于0。③上步中检查 `sem_count` 值如果大于0，就执行唤醒 `sem_waitlst` 中进程的操作，并且需要调度进程时就执行进程调度操作，不管 `sem_count` 是否大于0（通常会大于0），都标记信号量释放成功。当然最后别忘了对自旋锁 `sem_lock` 进行解锁。

在 LMOSEM 中信号量的接口，如代码清单5-13 [xxxx/lmosem/include/knlinc/krlsem.h] 所示。

代码清单5-13 操作信号量的接口函数

```
void krlsem_t_init(sem_t* initp);    //初始化信号量
void krlsem_set_sem(sem_t* setsem, uint_t flg, sint_t conut); //有些情况需要改变信号量
void krlsem_down(sem_t* sem);        //获取信号量
void krlsem_up(sem_t* sem);          //释放信号量
```

这里只是信号量的一种常用的形式，称为二值信号量，是用于控制对资源进行互斥访问的，后面开发 LMOSEM 时还会见到记数型的信号量。通常 `sem_t` 数据结构也不是以独立的方式使用的，它也是和 `list_h_t`、`spinlock_t` 一样，被嵌入到更大的数据结构中一起运作的。

我们了解了信号量的运行机制，也了解了如何用 C 语言的基本类型、list_h_t、spinlock_t、kwlist_t，来构建出 sem_t 数据结构，然而 sem_t 也是操作系统内核中最基本的数据结构。在后面的开发中，我们势必还会用这些基本的数据结构，构建更为庞大和高级的数据结构。

5.3 数据结构存在于内存中的形式

一个数据结构设计好了，在使用之前，必须将其实例化，也就是定义一个这种数据结构类型的变量。无论这种类型的变量是静态地存在于程序的二进制文件中，还是由程序运行时动态创建，程序在执行时，它都必须占有一定的内存。一个数据结构的实例，在内存中是怎么存放的，是否占用连续的内存，数据结构成员之间的关系，以及它们之间内存地址对齐的方式，这些都是要弄清楚的问题。

上述这些问题难道不该由编译器来处理吗，是的，对于开发应用程序完全不必了解这些问题，但是开发操作系统内核则不同，它的许多数据结构和硬件有直接关系，有些硬件对数据结构的形式有特定的要求，然而这只是一方面，另一方面，开发操作系统内核往往不只是一用一种编程语言，例如，有很多情况下要用汇编语言去操作 C 语言定义的数据结构。

可以发现，这些问题不是操作系统理论所能涉及的，但是要实现可运行的操作系统，就必须了解它们，即使它们会让你在白昼里抓狂，深夜里徘徊。

C 数据结构到内存映像

操作系统内核主要是用 C 语言编写的，暂时只要搞清楚 C 语言是怎么把一个特定数据结构映射到内存中的即可，其他高级语言也和 C 语言是差不多的。

下面用一个实例来看一个 C 语言的数据结构存放在内存中的形式，如代码清单 5-14 所示。

代码清单5-14 stutinem_t数据结构及其初始化函数

```
.....
//用于测试的数据结构
typedef struct s_STUTINMEM
{
    u8_t sm_u8;
    u16_t sm_u16;
    u32_t sm_u32;
    u64_t sm_u64;
    sint_t sm_sint;
    uint_t sm_uint;
    spinlock_t sm_lock;
    list_h_t sm_list;
}stutinem_t;
//初始化这个数据结构的函数
void stutinem_t_init(stutinem_t* initp)
```

```

{
    initp->sm_u8=1;
    initp->sm_u16=2;
    initp->sm_u32=3;
    initp->sm_u64=4;
    initp->sm_sint=5;
    initp->sm_uint=6;
    initp->sm_lock.lock=0;
    initp->sm_list.prev=&initp->sm_list;
    initp->sm_list.next=&initp->sm_list;
    return;
}

```

用命令 `arm-linux-gcc -S -O1 -fno-builtin -ffreestanding -std=c99 -march=armv4 -fno-stack-protector -fomit-frame-pointer -fno-ident -mtune=arm7tdmi -o stinmem.s stinmem.c` 编译代码清单 5-14, 就会产生如代码清单 5-15 所示的汇编代码。

代码清单5-15 stutinmem_t_init函数的汇编代码

```

stinmem_t_init: @r0存放这个数据结构类型的指针initp, 参阅前面C函数的参数传递
    str r4, [sp, #-4]!
    mov r3, #1
    strb r3, [r0, #0] @strb指令是存储字节数据的, 相当于initp->sm_u8=1;
    mov r3, #2
    strh r3, [r0, #2] @strh指令是存储半字数据的, 即双字节的, 相当于initp->sm_u16=2;
    mov r3, #3
    str r3, [r0, #4] @str指令是存储字数据的, 即4字节的, 相当于initp->sm_u32=3;
    mov r3, #4
    mov r4, #0
    str r3, [r0, #8] @相当于initp->sm_u64的低4字节=4;
    str r4, [r0, #12] @相当于initp->sm_u64的高4字节=0;
    mov r3, #5
    str r3, [r0, #16] @相当于initp->sm_sint=5;
    mov r3, #6
    str r3, [r0, #20] @相当于initp->sm_uint=6;
    mov r3, #0
    str r3, [r0, #24] @相当于initp->sm_lock.lock=0;
    add r3, r0, #28 @相当于r3=initp+28;
    str r3, [r0, #28] @相当于initp->sm_list.prev= initp+28;
    str r3, [r0, #32] @相当于initp->sm_list.next= initp+28;
    ldmfd sp!, {r4}
    bx lr

```

代码清单 5-15 中的注释已经给出了玄机, 可以假定一个值给 `initp`, 如 `0x1000`, 那么 `r0` 中的值也就是 `0x1000`, 可以看到 `stinmem_t` 的成员 `sm_u8` 正好在 `0x1000 (r0+0)` 地址上, `stinmem_t` 的成员 `sm_u16` 正好在 `0x1002 (r0+2)` 地址上, `stinmem_t` 的成员 `sm_u32` 正好在 `0x1004 (r0+4)` 地址上。照此逻辑画个图就更清楚了, 如图 5-3 所示。我们看到一个数据结构在内存中占用一段连续的内存空间, 除了由于 CPU 要求内存地址对齐问题而留下的

少量不会用到的内存空间，内存空间是按数据结构内部成员的顺序而增量分配的，如果成员变量也是一个数据结构，则 C 编译器会计算该数据结构的大小，以便顺序分配相应的内存空间。

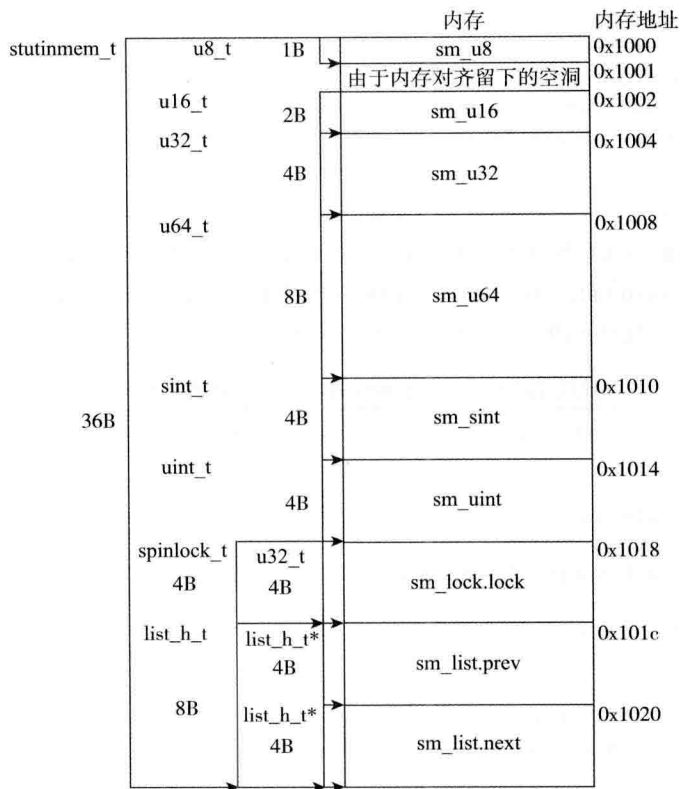


图 5-3 数据结构在内存中的存储方式

图 5-3 再一次验证了各种 C 语言基本数据类型在内存中占用的空间大小，同时也完整地展示了一个数据结构在内存中的视图，并且这个视图是通过 C 编译器编译出的汇编代码反推出来的，所以以后我们完全可以通过观察 C 语言数据结构，从而手工写出操作它们的汇编代码，以实现 C 语言无法完成的一些功能。

5.4 C 与汇编的混用

诸如开、关 CPU 中断，读取 CPU 的一些特殊寄存器，设置 CPU 模式等这些功能无法用 C 语言实现，但如果把这些功能都用汇编语言实现，势必增加汇编代码量，使代码不够直观，写汇编代码的函数还要遵循 C 语言调用约定，而且我们都不是计算机大神，不能完全保证自己写的汇编代码没有问题。

事实上,实现上述功能,一般不会超过5行汇编代码。为了这5行代码要写出符合C语言调用规范的汇编代码函数,这也不太划算。其实在C语言的扩展功能中,支持在C语言代码中嵌入汇编代码,这既能实现特殊功能,又能让C语言帮忙处理参数传递、返回值等。下面就去看看GCC支持嵌入汇编代码的方式。

GCC 中嵌入汇编代码的方式

在GCC中,支持嵌入汇编代码的方式与其他C编译器支持嵌入汇编代码的方式有些不同。GCC为了优化用户代码,设计了一种特有的嵌入方式,后面会通过一些实例来学习这种方式。

首先来看看GCC支持嵌入汇编代码的模板,它规定了汇编代码嵌入的形式和嵌入汇编代码需要由哪几个部分组成,如下所示。

```
__asm__ __volatile__ (代码部分:输出部分列表:输入部分列表:损坏部分列表);
```

可以看到,代码模板从__asm__开始(当然也可以是asm),紧跟着__volatile__(当然也可以是volatile,它也可以省略,它是防止GCC过度优化代码,从而带来问题,因为操作系统内核不同于应用),然后是跟着一对括号以分号结束。括号里大致分为如下4个部分:

- 1) 汇编代码部分,这里是实际嵌入的汇编代码。
- 2) 输出列表部分,让GCC能够处理C语言左值表达式,以实现与汇编代码的结合。
- 3) 输入列表部分,也是让GCC能够处理C语言表达式、变量、常量,让它们能够输入到汇编代码中去。
- 4) 损坏列表部分,告诉GCC汇编代码中用到了哪些寄存器,以便GCC在汇编代码运行前,生成保存它们的代码和生成汇编代码运行后恢复它们的代码。

它们之间用冒号隔开,如果只有汇编代码部分,后面的冒号可以省略,但是有输入列表部分而没有输出列表部分,那么输出列表部分的冒号必须要写,否则GCC没办法判断,同样的道理对于其他部分也一样。

来看一个最简单、最常用的例子,如代码清单5-16所示。

代码清单5-16 C语言中嵌入汇编代码的实例

```
__asm__ __volatile__ (: : : "memory");
//告诉GCC内存中的数据可能发生了改变,要回写寄存器了,这是为了达到防止GCC过
//度优化带来问题的目的
```

我们再来用这种代码形式,写一个加法函数,如代码清单5-17所示。

代码清单5-17 C语言中嵌入汇编代码的加法函数

```
int add(int a1,int a2)
{
    int sum;
    __asm__ __volatile__(
```

```

    "add %[svl],%[av1],%[av2] \n\t"
    : [svl] "=r" (sum)
    : [av1] "r" (a1), [av2] "r" (a2)
    : "r4", "cc", "memory"
);
return sum;
}

```

先来看输入列表部分，有两个输入条目，每个输入条目，都由一对 [] 括起的字符串构成的条目名称、一对引号里的限定符、一对括号括起的 C 语言表达式组成，输入条目之间用逗号隔开。条目名称可以在指令中引用，但前面要加上 % 号，如 %[svl]、 %[av1]。限定符是为了让 GCC 处理 C 语言表达式的，如 “r” (a1) 表示让 GCC 给 a1 分配一个通用寄存器，“r” (a2) 也是一样。其实限定符还可以是 “m”：表示后面的 C 表达式是内存地址、“I” 表示后面的 C 表达式是常数。当然还有很多限定符，有兴趣的可以查看 GCC 手册。我们的代码中只会用到这些。

再来看看输出列表部分，其中输出条目的格式和输入条目一样，唯一不同的是限定符的前面要加上等号，相当于 C 语言的赋值表达式，即相应 C 表达式的值必须是左值，例如 “=r”(sum) 表示让 GCC 给 sum 分配一个寄存器，只不过它是用于输出的，这里就相当于 sum=av1+av2。

然后来看看损坏列表部分，损坏部分的条目，就是一些用 “” 括起的寄存器，还有内存。例如，“r4” 寄存器、“cc” 表示 CPU 标志寄存器、“memory” 表示内存，这些东西就是为了告诉 GCC 这些寄存器需要保存和恢复，还有内存中的数据可能发生了改变。当然这里并没有在指令部分显示的用到 r4 寄存器、CPU 标志寄存器，所示这里损坏部分不会有什么作用，这主要是为了演示。

最后来看看指令部分，指令部分是由引号包含的，“\n\t” 即换行符和制表符，完全是为了让 GCC 格式化输出汇编代码的。add %[svl], %[av1], %[av2]，为什么不直接用寄存器呢，当然也可以直接使用寄存器，而使用条目名主要是让 GCC 能动态分配寄存器、优化代码，能让 C 语言表达式、变量自动和汇编代码结合。例如，代码 add %[svl], %[av1], %[av2]，一般情况下会被 GCC 处理成代码 add r0, r0, r1，这是不是比直接写纯汇编代码智能得多。

有了上面的说明，再去看看几个实例就更加熟悉了，先来看看读取 CPU 标志寄存器的实例，如代码清单 5-18 [xxxx/lmos_em/hal/cpuctrl.c] 所示。

代码清单5-18 读取CPU标志寄存器

```

cpuflg_t hal_read_cpuflg()
{
    cpuflg_t cpuflg;
    __asm__ __volatile__ (
        "mrs %[retcpr], cpsr \n\t"
        : [retcpr] "=r" (cpuflg)
        :
    );
}

```

```

        : "cc", "memory"
    );
    return cpuflg;
}

```

代码清单 5-18 中嵌入的汇编代码，没有输入条目，但有一个输出条目，它关联一个 `cpuflg_t` 类型的变量 `cpuflg`，并且要求 GCC 分配一个通用寄存器给它，然后使用 `mrs` 指令直接把 CPSR 寄存器的值读入分配给 `cpuflg` 变量的寄存器中，理想情况下这行代码 `"mrs %[retcpr],cpsr \n\t"` 会被 GCC 处理成 `"mrs r0,cpsr \n\t"`，而 `r0` 寄存器正是 GCC 用来存放函数返回值的，最后嵌入汇编代码模板的损坏部分告诉 GCC，CPU 的标志寄存器可能损坏，内存中的值可能发生改变。

下面来看看开、关 ARM920T CPU 中断的函数，如代码清单 5-19 [xxxx/lmos_em/hal/cpuctrl.c] 所示。

代码清单5-19 开、关ARM920T CPU中断的函数

```

void hal_disable_irqfiq()
{
    __asm__ __volatile__(
        "mrs r0,cpsr \n\t"
        "orr r0,r0,%[closeifiq]\n\t"
        "msr cpsr,r0 \n\t"
        :
        : [closeifiq] "I" (CIRQFIQ)
        : "r0", "cc", "memory"
    );
    return;
}

void hal_enable_irqfiq()
{
    __asm__ __volatile__(
        "mrs r0,cpsr \n\t"
        "bic r0,r0,%[openifiq]\n\t"
        "msr cpsr,r0 \n\t"
        :
        : [openifiq] "I" (CIRQFIQ)
        : "r0", "cc", "memory"
    );
    return;
}

```

ARM920T CPU 上开、关中断是设置 CPU 上的 CPSR 寄存器中的相关位来实现的，代码清单 5-19 中的 CIRQFIQ 是个宏，会被 GCC 预处理时，替换成常数 `0xc0`。从上面代码的输入条目中它前面的限定符“`I`”可以告诉 GCC，CIRQFIQ 是一个常数。GCC 会根据这个常数的大小决定把这个常数分配在寄存器中或者直接放在指令编码数据中。并且会在 `__asm__ __volatile__()` 中的指令执行之前，生成相关代码处理好这个问题。由于 CPSR 寄存器要求以

读 - 改 - 写的方式操作，所以上面代码中的指令部分先把 CPSR 寄存器中的值读取到 r0 寄存器中，然后改写 r0 寄存器中的值，最后把 r0 寄存器中的值写入 CPSR 寄存器中，操作中用到了 r0、CPSR 寄存器，所以要在代码模板的损坏部分写上 r0，让 GCC 在 `__asm__ __volatile__()` 中的指令执行之前，生成保存 r0 寄存器中的值的相关代码，以便在其后恢复到 r0 寄存器中。

几个例子过后，想必读者已经对 GCC 扩展的嵌入汇编代码方式的强大，心领神会了，不完全清楚的也不用害怕，后面还会对其有所提及，现在可以去看部电影，或者听首钢琴曲，轻松一下了。

5.5 小结

本章的内容很少，所以这么快就要结束了。回顾这章，主要内容如下：

1) 学习了 C 语言使用寄存器的约定，以及它是如何处理参数、返回值的。了解这些主要是为了能用汇编语言写出能被 C 语言调用的函数，或者在汇编代码中调用 C 函数应该做些什么。

2) 探讨了 C 语言基本数据类型的位宽及占用内存的大小，并用基本类型一步步构建了后面将要用到的一些基本的数据结构，如 `list_h_t`、`spinlock_t`、`kwlst_t`、`sem_t`。

3) 通过研究 C 语言编译器生成用于操作 C 语言数据结构的汇编代码，证明了 C 语言的数据结构在内存中存在的形式、对齐方式，这是因为有些情况下要用汇编代码去操作 C 语言定义的数据结构。

4) 最后学习了 GCC 独有的嵌入汇编代码的方式，可以让开发者自由地在 C 语言函数中嵌入汇编代码并且能自动处理汇编代码和 C 语言表达式结合的问题，这种方式对开发操作系统内核很有利。

直到这里，我们做好了一切准备，具备了操作系统内核开发的绝大部分基础，学习了这么多东西，就是为了能实现真正可运行的操作系统内核，所以路才刚刚开始，让我们一起带着愉快的心情踏上操作系统内核之旅吧。

内核初始化

对于大部分看过各种标准操作系统教材的读者，具体操作系统的第一行代码是什么样子的，代码该从哪里开始写起，恐怕永远是一段距离、一种幻想。

前面介绍了一些硬件、操作系统内核设计、开发环境相关的资料，其实那些资料可以作为一个独立的部分。如果读者对那些资料有深入的理解，大可跳过，从这里开始。

操作系统内核就像一个服务者，随时处理应用进程的请求。但是在这之前，操作系统内核要完成自身的初始化，一步步带着计算平台和其自身到一个已知的状态，这包括初始化 CPU 的工作模式、启动 MMU、启动中断控制器、建立内存管理的数据结构和一些其他重要的数据结构等。这些内容也是硬件平台相关的，所以也可以称为硬件平台相关的初始化。

6.1 开始

话不多说，纸上谈兵的事情，我们就不干了，这事儿都让赵括和那些操作系统理论教材的作者给干了，并且他们干得漂亮。

从第一行代码为什么是汇编代码开始，了解 CPU 从哪个地址开始运行代码，然后在代码的引导下设定 CPU 的工作模式，排除一些可能干扰代码正常运行的因素，如中断，在初始化阶段还没有响应中断的能力，最后设置 C 语言的工作环境，让第一个 C 语言函数运行起来。有了 C 语言，后面的代码写起来就方便多了。

6.1.1 第一行汇编代码

大部分现代 CPU，或多或少都有几条跳转指令或者调用指令用于改变程序的控制路径，

使得程序可以分支、循环执行。想象一下，如果 CPU 只能顺序执行程序，那是不是很可怕，至少很多算法都无法实现。不管 CPU 支不支持分支执行程序，在其上电之后，它一定要从某个具体的地址开始执行，然后才能在指令的引导下改变执行路径。

对于我们所用的 ARMCPU，它规定必须从“复位向量”开始执行，并且在 CPU 上电时向量地址为 0，即它是从 0 地址开始执行的。从前面章节中知道 ARMCPU 有 8 个向量，并且每个向量只有 4 字节大小，不能存储所有相关代码，所以这里必然存放的是一条跳转指令。事实上除最后一个向量可以在其后存放相关指令，其他向量都存放的是一条跳转指令，以跳转另一个地址上执行相关指令。

从前面知道 mini2440 上已经安装了 supervivi，相当于 PC 的 BIOS，它有初始化 mini2440 和加载操作系统内核的功能。mini2440 也有两种启动方式，不同的启动方式，0 地址空间上映射的存储器是不同的，在 NOR 启动方式下，ARMCPU 首次开始运行的代码是 NORFlash 中的 supervivi。当然也可以写一个比 supervivi 更简单的引导程序，只是需要频繁烧写开发板，这个过程非常麻烦，也不利于开发和调试操作系统内核。所以不如借助 supervivi，它能把操作系统内核文件加载到指定地址的内存空间中，并且让 CPU 跳转到那个地址上执行。

mini2440 上物理内存地址空间是 0x30000000 ~ 0x33FFFFFF，这就是我们 LMOSEM 内核的领地。从 LMOSEM 内核的链接脚本中，不难看出 LMOSEM 内核是从 0x30008000 地址开始运行的，如代码清单 6-1 [xxxx/lmosem/script/lmemknllink.S] 所示。

代码清单6-1 生成LMOSEM链接脚本的lmemknllink.S

```
.....
ENTRY(_start)
SECTIONS
{
    . = 0x30008000;
    __begin_kernel = .;
    __begin_lmosem_hal_head_text = .;
    .head.text ALIGN(4) : { *(.head.text) }
    __end_lmosem_hal_head_text = .;
    .....
}
```

好了，知道了这些内容就可以开始写代码了，在 lmosem/hal 目录下新建一个文件 init.S，在其中写上如代码清单 6-2 [xxxx/lmosem/hal/init.S] 所示的汇编代码，并在 buildfile.h 文件中的 #define BUILD_HALY_OBJS 后面加入 init.o，最后 make 一下，就会自动编译链接了。如果不明白其中的原理，请参阅前面相关的章节。

代码清单6-2 LMOSEM初始化的汇编代码

```
.equ WTCN, 0x53000000
.equ INTMASK, 0x4a000008
.equ INTSUBMASK, 0x4a00001c
.section .head.text @让GCC把下面指令数据放在一个名叫.head.text的节中，以便让链接器处理
```

```

.code 32 @让GCC生成32位的ARMCPU指令
.extern lmosemhal_start @该符号是外部的, 在lmosemhal_start.c文件中
.global _start @导出_start符号让链接器能看到
.align 4
_start:
    @关闭irq和fiq中断, 并且切换到SVE模式
    msr cpsr , #0xc0|0x13
    @关闭看门狗
    ldr r0 , =WTCON
    mov r1 , #0
    str r1 , [r0]
    @屏蔽所有中断
    ldr r0 , =INTMASK
    ldr r1 , =0xffffffff
    str r1 , [r0]
    @屏蔽所有子中断
    ldr r0 , =INTSUBMASK
    ldr r1 , =0x7fff
    str r1 , [r0]
    ldr sp,=INIT_HEAD_STACK_ADDR
    bl lmosemhal_start

```

看其中的代码注释已经很清楚了, 0xc0|0x13=(11010011b)正好改变了CPSR寄存器中的0~4位(模式位)和6~7位(irq和fiq中断屏蔽位), 所以达到了切换CPU工作模式和关闭CPU响应中断的目的。前面章节中有对CPSR寄存器的介绍。INTMASK和INTSUBMASK这两个内存地址映射的是中断控制器上的主中断和子中断的中断源屏蔽寄存器, 把它们的每个位都设置为1, 所有的中断源就被屏蔽了。因为LMOSEM才刚开始运行, 还没有处理中断的能力。关于中断控制器后面还有专门介绍, 这里只要知道这些就够了。

剩下的就是看门狗了。看门狗是什么呢, 它其实就是定期产生中断和复位信号的一个设备, 它里面有个计数器, 每来一个时钟信号就对其中的值减一, 当计数器中的值减到0时, 就产生中断和复位信号(根据不同的设置, 可以同时产生这两种信号, 也可只产生其中一种)。这是为了防止程序卡死和内部错误的一种机制, 如果一段时间内不向计数器中写入新的数据, 即喂狗, 开发板可能就要重启了。这种机制在嵌入式应用领域中很有用, 因为有些设备是无人看管的。WTCON这个内存地址映射的就是看门狗设备的控制寄存器, 把这个寄存器中的第0位(复位信号)和第2位(中断信号)设置为0就行了, 当然把所有的位都置为0也行。因为在LMOSEM初始化时, 我们可不想时刻提防着狗饿了。

最后, 把INIT_HEAD_STACK_ADDR(在platform_t.h文件中定义: #define INIT_HEAD_STACK_ADDR 0x34000000)写入SP寄存器中, 这是初始化栈指针, 该地址值正好是物理内存的顶端, 然后调用C函数lmosemhal_start(在lmosemhal_start.c文件中)。因为C语言需要栈才能工作。

设置了CPU的工作模式, 干掉了所有中断, 也宰了看门狗, 这排除了所有干扰LMOSEM代码运行的因素, 最后设置了栈指针, 到这里可以用C语言了, 我们不妨马上去写一个C函数试试。

6.1.2 第一个C函数

init.S 文件中的汇编代码，已经初始化了栈，可以调用 C 语言写的函数了，也就是说可以用 C 语言写 LMOSEM 内核的代码了。

首先在 hal 目录下新建一个 C 代码文件 lmosemhal_start.c。新建好了文件，可以写上代码，如代码清单 6-3[xxxx/lmosem/hal/lmosemhal_start.c] 所示。

代码清单6-3 LMOSEM的第一个C函数

```
#include "lmosemtypes.h"
#include "lmosemctrl.h"
LKHEAD_T void lmosemhal_start()
{
    for(;;);
    return;
}
```

代码清单 6-3 非常简单，lmosemhal_start 函数中是一个死循环，让 CPU 原地踏步，函数自然也不会返回，可能不清楚的就是 LKHEAD_T，它在 xxxx/lmosem/include/bastypeinc/bastype_t.h 中：#define LKHEAD_T __attribute__((section(".head.text")))，它是一个宏，__attribute__((section(".head.text"))) 是 GCC 的扩展功能，可以指定函数代码编译后放在哪个节中，相当于 init.S 文件中汇编代码里的 .section .head.text，它们是等效的。

在编译之前，先来看看 LMOSEM 内核项目是怎么处理 C 语言的头文件的，LMOSEM 内核的头文件非常多，不仔细管理，后面的开发就会非常麻烦，不仅仅要在 C 语言代码文件的前面包含一大堆 C 语言头文件，有时还存在交叉引用数据结构类型的错误。为了解决这个问题，我们每建立一个 xxx.c 文件就建立一个 xxx_t.h 文件和一个 xxx.h 文件，其中 xxx_t.h 文件用于存放定义的数据结构类型，xxx.h 文件用于存放一些声明的函数原型。先来看看 lmosemtypes.h 文件中是什么，如代码清单 6-4[xxxx/lmosem/include/lmosemtypes.h] 所示。

代码清单6-4 LMOSEM代码中顶层数据类型头文件

```
#ifndef _LMOSEMTYPES_H
#define _LMOSEMTYPES_H
#include "config.h"
#include "bastype.h"
#include "haltypes.h"
#include "krltypes.h"
#include "drvtypes.h"
#endif
```

lmosemtypes.h 文件中又包含了一些头文件，其中有配置 (config.h)、基本类型 (bastype.h)、硬件相关层 (haltypes.h)、内核功能层 (krltypes.h)、驱动层 (drvtypes.h) 头文件。并且它们都再次各自包含各自层中的 xxx_t.h 文件，即各自层中所有定义数据结构的头文件如

haltypes.h, 如代码清单 6-5[xxxx/lmosem/include/halinc/haltypes.h] 所示。

代码清单6-5 LMOSEM硬件相关层的数据类型头文件

```
#ifndef _HALTYPES_H
#define _HALTYPES_H
#include "platform_t.h"
#include "cpu_t.h"
#include "cpuctrl_t.h"
#include "halmmu_t.h"
.....
#endif
```

其他如 krltypes.h、drvtypes.h 和 haltypes.h 文件中, 也是一样的形势, 不必多说。

看看 lmosemmctrl.h 文件, 如代码清单 6-6[xxxx/lmosem/include/lmosemmctrl.h] 所示。

代码清单6-6 LMOSEM代码中顶层头文件

```
#ifndef _LMOSEMMCTRL_H
#define _LMOSEMMCTRL_H
#include "halheads.h"
#include "krlheads.h"
#include "drvheads.h"
#include "libheads.h"
#endif
```

其实, 它和 lmosemtypes.h 文件的道理相同, 内核的每个层定义一个头文件, 如 halheads.h, 它再次包含各自层中的 xxx.h 文件, 只不过这些 xxx.h 文件中是函数原型的声明。由于篇幅所限, 就不再往下看了, 读者可自行参阅代码。

编译 lmosemhal_start.c 文件和编译 init.S 文件的方法是一样的, 也是在 buildfile.h 文件中的 #define BUILD_HALY_OBJS 后面加入 lmosemhal_start.o。只不过现在有了 init.o 和 lmosemhal_start.o 两个编译目标文件, 以后还会有更多, 我们只要往后面添加就行了, 注意它们之间要有空格。最后 make 一下, 就会自动编译链接了。

按照前面章节中的方法连接好开发板, 配置好 minicom, 在 Linux 下打开终端输入: sudo minicom 如图 6-1 所示。



图 6-1 运行 minicom

然后打开开发板电源, 在打开电源之前, 确定启动方式是 NOR 启动方式, 这时就会启动开发板上 NORFlash 中的 supervivi, 启动成功时会看到如图 6-2 所示的界面。

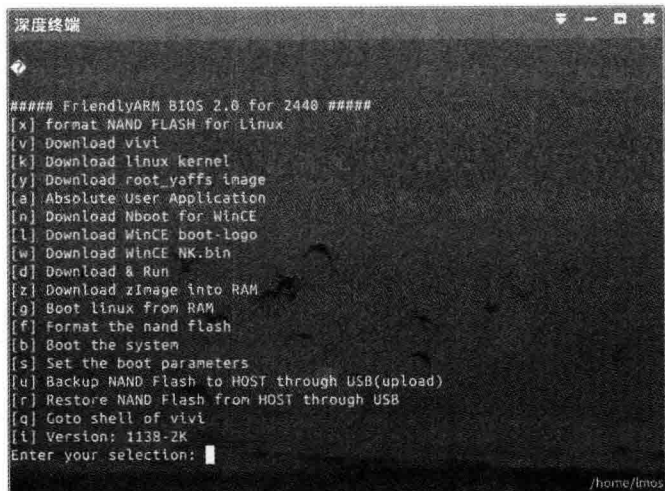


图 6-2 启动 mini2440 中的 supervivi

这里 supervivi 给出了很多选项：直接下载 Linux 内核、启动系统、格式化 NANDFlash 等。我们要用的是 d 选项，这个选项是加载二进制文件到指定地址的内存空间中去的。所以用来加载 LMOSEM 内核非常方便。当输入 d 后，正常情况如图 6-3 所示，表示 supervivi 正在等待 USB 数据下载器 dnw2 通过 USB 线向开发板中写入数据。打开另一个终端，切换到 lmosem 目录下，make 一下，然后执行 `sudo make knlexc`，make 就会调用 dnw2，把编译生成的 lmosemkrnl.bin 文件写入开发板上以 0x30008000 为开始地址的内存空间中，然后在 supervivi 的控制下，CPU 会跳转到这个地址上开始运行。注意笔者所用的 dnw2 和网上流传的 dnw 有些不同，笔者修改了 dnw，如图 6-4 所示。

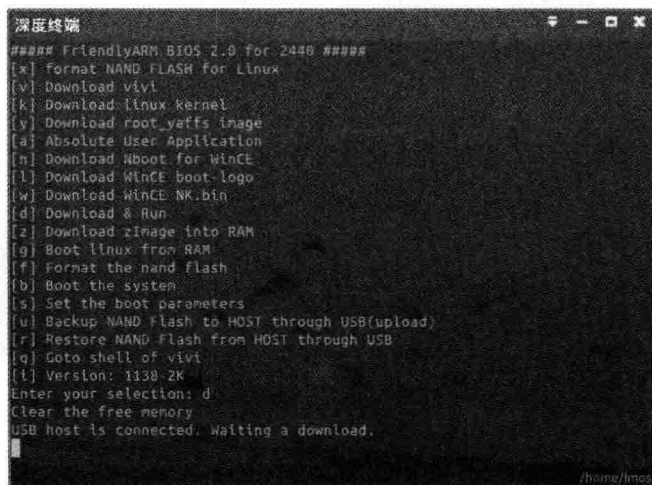


图 6-3 等待下载相应的程序

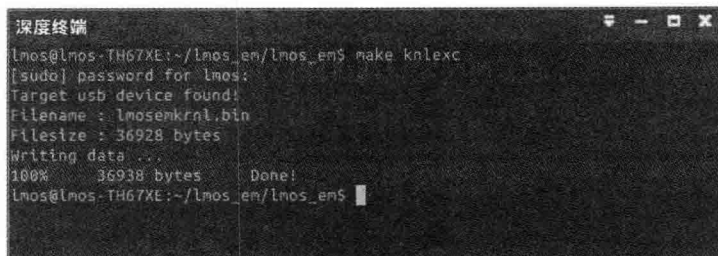


图 6-4 下载 lmosemkrnl.bin 到 mini2440 中

两个终端上没有任何显示，开发板也没有任何异常。这是当下最正常的表现，因为在 lmosemhal_start 函数中的代码是一行死循环语句，开发板上的 CPU 就在那里不停地原地打转，没有做其他任何事。程序确实是正常运行的。

到这里好像根本没有干什么实质性的工作，但是能够体会到，在裸机上要运行一行代码都是不容易的。然而已经可以用 C 语言写 LMOSEM 内核了，这是一个不小的飞跃，也算是推开了 LMOSEM 内核初始化的大门。入了门，我们就可以慢慢向前探索了。

6.2 MMU 和中断向量的初始化

MMU 是映射和保护内存的，中断向量是 CPU 规定的。这是 LMOSEM 内核其他部件运行的基础，所以越早初始化越好。但是有很多现有的嵌入式操作系统是不用 MMU 的，因为有很多嵌入式操作系统是强实时的操作系统，使用 MMU 会带来额外的操作，从而影响实时性能，但是它们自始至终只是运行几个特定的应用，并且这些应用是和内核连接在一起的。所以不使用 MMU 反而更好。

我们是探索操作系统内核机制的，所以还是要用到 MMU，它是很多高级操作系统内核必须要支持的东西，如虚拟内存的实现。

下面就去看看 LMOSEM 是如何初始化 MMU 和中断向量的。

6.2.1 初始化 MMU

MMU 是对内存空间块进行映射和保护的设备，所以对它的初始化非常关键，出不得错，否则 LMOSEM 内核本身就无法运行了，但也不用怕，我们一步一步来。

先得清楚 MMU 初始化的步骤，如下。

- 1) 找一块内存空间存放页表，ARM920T 的 MMU 对这块内存空间的首地址有要求：必须是 16KB 对齐的。
- 2) 建立页表数据。
- 3) 把页表的首地址，写到 MMU 一个特定的寄存器中。
- 4) 打开 MMU。

开始之前要规划一下，上次写到了 `lmosemhal_start` 函数。这次也是要从这里开始，但从前面的章节中知道，**LMOSEM** 是分层的，所以这里要再写一个函数 `init_hal`，同时新建一个文件 `halinit.c`，并且还要建立一些头文件，最后把 `halinit.o` 加入编译头文件 `buildfile.h` 中，这应该要和 6.1 节描述的一样。让所有硬件相关层的初始化函数都归纳到 `init_hal` 函数中，于是 `lmosemhal_start` 函数变成了如代码清单 6-7[xxxx/lmosem/hal/lmosemhal_start.c] 所示。

代码清单6-7 LMOSEM初始化

```
LKHEAD_T void lmosemhal_start()
{
    init_hal();
    return;
}
```

我们发现有些硬件平台上并没有 MMU 器件，所以再新建一个文件 `platform.c`，当然也要建立相关的头文件，把 MMU、中断向量的初始化代码都归纳到其中。首先写一个 `init_paltform` 函数，如代码清单 6-8[xxxx/lmosem/hal/platform.c] 所示。然后把 `init_paltform` 函数归纳到 `init_hal` 函数中，如代码清单 6-9[xxxx/lmosem/hal/halinit.c] 所示。

代码清单6-8 平台初始化

```
void init_paltform()
{
    s3c2440mmu_init();      //初始化S3C2440A的MMU
    s3c2440vector_init();   //初始化S3C2440A的中断向量
    return;
}
```

代码清单6-9 硬件相关层初始化

```
LKHEAD_T void init_hal()
{
    init_paltform();
    return;
}
```

好了，框架也准备好了，下面就在 `s3c2440mmu_init` 函数中实现初始化 MMU 的步骤。

从前面的内容得知，mini2440 上的物理内存地址空间是 `0x30000000 ~ 0x33FFFFFF`，所以用 `0x30004000` 地址开始的内存空间存放页表目录。因为我们为了尽量简单，采用了 ARM920T 上 MMU 支持最大的页面——1MB 页，所以只需要一级页表即可。这种分页方式下，每个页表条目是 4 字节大小，可以映射 1MB 大小的内存空间，因此有 4096 个页表项，刚好映射 ARM920T 上 4GB 的虚拟地址空间。

确定了页表目录的首地址，就可以写代码初始化页表了，如代码清单 6-10[xxxx/lmosem/hal/platform.c] 所示。

代码清单6-10 MMU初始化

```

LKHEAD_T void s3c2440mmu_init()
{
    uint_t paddr=0;
    u32_t pgdsc=0;
    u32_t* pgdiradr=(u32_t*)PAGE_TLB_DIR; // 0x30004000
    for(uint_t i=0;i<PAGE_TLB_SIZE;i++) // PAGE_TLB_SIZE 4096
    {
        pgdsc=paddr|PTE_SECT_AP|PTE_SECT_DOMAIN|PTE_SECT_NOCW|PTE_SECT_BIT;
        pgdiradr[i]=pgdsc;
        paddr+=0x100000;
    }
    pgdiradr[0]=SDRAM_MAPVECTPHY_ADDR|PTE_SECT_AP|PTE_SECT_DOMAIN|PTE_SECT_NOCW|PTE_
    SECT_BIT;
    .....
    return;
}

```

代码清单 6-10 中用一个循环来初始化 4096 个页表项，paddr 从 0 开始，每循环一次就加 0x100000，PTE_SECT_AP、PTE_SECT_DOMAIN、PTE_SECT_NOCW、PTE_SECT_BIT 定义在 platform_t.h 文件中，如代码清单 6-11[xxxx/lmosem/include/halinc/platform_t.h] 所示。如果对代码清单 6-11[xxxx/lmosem/include/halinc/platform_t.h] 中的宏定义不清楚，可以参阅 MMU 相关章节。

代码清单6-11 平台相关的一些宏

```

#ifndef _PLATFORM_T_H
#define _PLATFORM_T_H
#ifdef CFG_S3C2440_PLATFORM
#define CPUCORE_MAX 1
#define SDRAM_MAPVECTPHY_ADDR 0x30000000
.....
#define PAGE_TLB_DIR 0x30004000
#define PAGE_TLB_SIZE 4096
.....
#define PTE_SECT_AP (3<<10) //AP位11b
#define PTE_SECT_DOMAIN (0<<5) //页面所在的域为0
#define PTE_SECT_NOCW (0<<2) //页面不支持缓存回写
#define PTE_SECT_BIT (2) //1MB大小的页面
.....
#endif
#endif // PLATFORM_S3C2440_T_H

```

可以看到，代码清单 6-11 中对虚拟地址和物理地址的映射是一一对应的，即 0x30000000 虚拟地址经过 MMU 按其页表转换后的物理地址还是 0x30000000。但那个循环结束后，代码突然改变了第 0 个页表项，使其虚拟地址 0 映射到物理地址 0x30000000，后面会看到这么做的作用。

建立了页表，接着就是应该把页表的首地址送给 MMU，其实是把页表的首地址送给 CP15 的 C2 寄存器，只要把 0x30004000 这个值写入这个寄存器中就行了。但是操作这个寄存器，需要特定的汇编代码，因此我们来写个函数，在这个 C 语言函数中嵌入几行汇编代码来达到此目的，如代码清单 6-12[xxxx/lmosem/hal/platform.c] 所示。

代码清单6-12 设置MMU的页表首地址

```
LKHEAD_T void s3c2440mmu_set_tblbass(u32_t tblbphyadr)
{
    __asm__ __volatile__(
        //把[tbass]关联的寄存器中的值写入CP15的C2寄存器中
        "mcr p15,0,%[tbass],c2,c0,0 \n\t"
        :
        //分配一个通用寄存器，在其中写入tblbphyadr的内容，该动作是由GCC完成的
        : [tbass]"r"(tblbphyadr)
        : "cc","memory"
    );
    return;
}
```

代码清单 6-12 中的注释，已经写得很清楚了，自然不必多说。但是在打开 ARM920T 的 MMU 之前，需要做两件额外的工作：设置域寄存器和使 Cache 中的指令和数据无效，如代码清单 6-13[xxxx/lmosem/hal/platform.c] 所示。若有疑问，可参见前面章节里的 CP15 的 C3 寄存器和 Cache。

代码清单6-13 设备域寄存器、使Cache中的指令和数据无效

```
LKHEAD_T void s3c2440mmu_set_domain(u32_t domain)
{
    __asm__ __volatile__(
        //把[domval]关联的寄存器中的值写入CP15的C3寄存器中
        "mcr p15,0,%[domval],c3,c0,0 \n\t"
        : : [domval]"r"(domain)
        : "cc","memory"
    );
    return;
}

LKHEAD_T void s3c2440mmu_invalid_dicache()
{
    __asm__ __volatile__(
        "mov r0, #0 \n\t"
        "mcr p15, 0, r0, c7, c7, 0 \n\t" //使指令Cache和数据Cache中的数据无效
        "mcr p15, 0, r0, c7, c10, 4 \n\t"
        "mcr p15, 0, r0, c8, c7, 0 \n\t" //使指令TLB、数据TLB中的数据无效
        : : "cc","memory","r0" //用到了r0寄存器，所示要告诉GCC
    );
    return;
}
```

后面会看到，表示 16 个存储域的域寄存器的所有位都被设置成 1，这表示不考虑页表条目中相关的控制位。事实上只需要设置域寄存器中的第 0 个域就好了，因为在前面所有的页表条目都已经设置为属于第 0 个域的了。

最后就是打开 MMU，一旦打开它，CPU 发送的地址通过 MMU 转换后才会发送到具体的硬件设备。打开 MMU 是通过设置 CP15 的 C1 寄存器中的第 0 位为 1 做到的，如代码清单 6-14[xxxx/lmosem/hal/platform.c] 所示。代码中，那么多 nop 指令是为了填充 ARM920T CPU 的流水线。

代码清单6-14 打开MMU

```
LKHEAD_T void s3c2440mmu_enable()
{
    __asm__ __volatile__(
        "mrc p15,0,r0,c1,c0,0 \n\t" //把CP15的C1寄存器中的值读取到r0寄存器中
        "orr r0 ,#1 \n\t" //改变r0寄存器中的第0位的数值为1
        "mcr p15,0,r0,c1,c0,0 \n\t" //把r0寄存器中的值写入CP15的C1寄存器中
        "nop \n\t"
        "nop \n\t"
        "nop \n\t"
        "nop \n\t"
        "nop \n\t"
        "nop \n\t"
        "nop \n\t"
        : : "r0", "cc", "memory" //告诉GCC使用了r0寄存器
    );
    return;
}
```

下面把这些代码组合在一起，就形成了初始化 MMU 的整个流程，见代码清单 6-15 [xxxx/lmosem/hal/platform.c]。

代码清单6-15 MMU初始化

```
LKHEAD_T void s3c2440mmu_init()
{
    uint_t paddr=0;
    u32_t pgdsc=0;
    u32_t* pgdiradr=(u32_t*)PAGE_TLB_DIR;
    for(uint_t i=0;i<PAGE_TLB_SIZE;i++)
    {
        pgdsc=paddr|PTE_SECT_AP|PTE_SECT_DOMAIN|PTE_SECT_NOCW|PTE_SECT_BIT;
        pgdiradr[i]=pgdsc;
        paddr+=0x100000;
    }
    pgdiradr[0]=SDRAM_MAPVECTPHY_ADDR|PTE_SECT_AP|PTE_SECT_DOMAIN|PTE_SECT_NOCW|PTE_SECT_BIT;
    s3c2440mmu_set_tblbass(PAGE_TLB_DIR); //把页表的首地址写入CP15的C2中
    s3c2440mmu_set_domain(~0); //设置域寄存器的所有位为1
    s3c2440mmu_invalid_dicache(); //使指令Cache和数据Cache中的数据无效
}
```

```
s3c2440mmu_enable();//打开MMU
return;
}
```

MMU 的初始化完成了，除了为 0 的虚拟地址外，其他虚拟地址和物理地址都是一一对应的，映射了完整的 4GB 地址空间。这看上去好像和没有使用 MMU 一样，看不出 MMU 的优势在哪里。不过，我们既然能写出让虚拟地址和物理地址对等映射的代码，利用 MMU 完成许多高级操作系统中请求调页的机制自然也就不远了。

6.2.2 复制中断向量

从前面内容得知，ARM920T CPU 在访问数据或者读取指令失败时、IRQ 引脚或者 FRQ 引脚上有信号时、执行 SWI 指令时，CPU 就会跳转到一些固定的地址上去执行指令。这些地址上存放的指令数据称为中断向量。CPU 就是通过这些向量地址上的指令再次跳转到具体的中断处理代码上去的。

不妨再次把前面章节中的中断向量表，拿到这里看看，如表 6-1 所示。ARM920T CPU 一共有 8 个中断向量。向量地址可以是地址 (0x0 ~ 0x1F)，也可以是地址 (0xFFFF0000 ~ 0xFFFF001F)。

表6-1 ARM920T的中断、异常向量

低 / 高向量地址	向量名称	ARM920T 进入的状态和模式
0x00000000 / 0xFFFF0000	复位向量	ARM 状态，系统管理模式
0x00000004 / 0xFFFF0004	未定义指令终止向量	ARM 状态，未定义指令终止模式
0x00000008 / 0xFFFF0008	SWI 向量	ARM 状态，系统管理模式
0x0000000c / 0xFFFF000c	预取指令终止向量	ARM 状态，数据访问终止模式
0x00000010 / 0xFFFF0010	数据访问终止向量	ARM 状态，数据访问终止模式
0x00000014 / 0xFFFF0014	保留	保留
0x00000018 / 0xFFFF0018	外部中断向量	ARM 状态，外部中断模式
0x0000001c / 0xFFFF001C	快速中断向量	ARM 状态，快速中断模式

下面先来写好向量代码，而中断处理的代码是以后要做的事，从表 6-1 中也可以看出，每个向量只有 4 字节大小，最多只能存放一条指令，无法存放整个中断处理的代码，所以必须先执行一条跳转指令，再跳转到中断处理代码那里执行。先新建一个文件 vector.s，在其中写上如代码清单 6-16[xxxx/lmosem/hal/vector.s] 所示的汇编代码。当然别忘了把它加入编译头文件 buildfile.h 中的 #define BUILD_HALY_OBJS 后面。

代码清单6-16 ARM920T的中断、异常向量代码

```
.section .lmosem_hal.vector,"ax"
.code 32
```

```

.align 0
.extern hal_undefins
.extern hal_swi
.extern hal_prefabt
.extern hal_dataabt
.extern hal_bugdie
.extern hal_irq
.extern hal_frq
.extern _start
.global vector
vector:
    ldr pc,lmosem_restart_l
    ldr pc,lmosem_undefins_l
    ldr pc,lmosem_swi_l
    ldr pc,lmosem_prefabt_l
    ldr pc,lmosem_dataabt_l
    ldr pc,lmosem_bugdie_l
    ldr pc,lmosem_irq_l
    ldr pc,lmosem_frq_l @把hal_frq的地址装载进PC完成跳转，上面的代码也一样
.align 4
lmosem_restart_l: .word _start @在init.S文件中
lmosem_undefins_l: .word hal_undefins
lmosem_swi_l:      .word hal_swi
lmosem_prefabt_l: .word hal_prefabt
lmosem_dataabt_l: .word hal_dataabt
lmosem_bugdie_l:  .word hal_bugdie
lmosem_irq_l:     .word hal_irq
lmosem_frq_l:     .word hal_frq @由链接器填入实际的地址，上面的代码也一样
.end

```

hal_undefins、hal_swi、hal_prefabt、hal_dataabt、hal_bugdie、hal_irq、hal_frq 这些标号，是外部其他代码文件中的，也是真正实际中断处理代码的地址，LD 链接器在链接时会自动把它们地址填上去。再来看看向量代码，正好是 8 条 ldr 指令，从前面的章节中了解到 ldr 其实是条伪指令，在这里它会变成 ldr pc,[pc,xxxx]，即表示以当前 PC 中的值加上一个 xxxx 标号的偏移量（由编译器计算），然后把这个内存地址上的数据装载到 PC 中，从而完成了跳转，这里当然就会跳转到相关的中断处理代码上去。

由表 6-1 可知，中断向量的开始地址必须是 0 或者 0xFFFF0000，而 mini2440 上这两个地址上都没有内存，即 SDRAM 芯片，并且 LMOSEM 内核文件也是加载到以 0x30008000 为开始地址的内存中的，不过还好有 MMU，它可以把 0 或者 0xFFFF0000 地址映射到相应的物理内存地址上，事实上，前面初始化 MMU 时，我们已经把 0 虚拟地址映射到 0x30000000 物理地址上了，所以只要把编译过的 vector.s 文件中代码的二进制数据，复制到以 0x30000000 为开始地址的物理内存中就行了。

下面就来写好复制中断向量的代码，如代码清单 6-17[xxxx/lmosem/hal/platform.c] 所示。

代码清单6-17 复制ARM920T的中断、异常向量

```
LKHEAD_T void s3c2440vector_copy()
{
    u32_t* src_t = (u32_t*) (&__end_lmosem_hal_vector);
    u32_t* src = (u32_t*) (&vector);
    u32_t* det = (u32_t*) (CPU_VECTOR_VIRADR);
    for(uint_t i=0;i<4096;i++)
    {
        det[i]=0;
    }
    for(;src<src_t;src++,det++)
    {
        *det=*src;
    }
    return;
}
```

代码清单 6-17 中 CPU_VECTOR_VIRADR 赋值为 0，vector 就是代码清单 6-16 中的标号，C 语言中访问汇编语言中标号的地址要加 & 符号，并在相应的头文件中申明为 extern 类型。这里，C 语言看到的 vector 是在 platform.h 文件中申明的。__end_lmosem_hal_vector 是在 link.h 文件中申明的：extern char __end_lmosem_hal_vector; 可是找不到它是在哪里定义的，其实它是在链接脚本里定义的，链接器最终会把它的地址自动填写到用到它的地方，如代码清单 6-18 所示。

代码清单6-18 LMOSEM的链接脚本

```
.....
SECTIONS
{
    . = 0x30008000;
    .....
    __begin_lmosem_hal_init = .;
    .lmosem_hal.init ALIGN(4) : { *(.lmosem_hal.init) }
    __end_lmosem_hal_init = .;
    __begin_lmosem_hal_vector = .;
    .lmosem_hal.vector ALIGN(4) : { *(.lmosem_hal.vector) }
    __end_lmosem_hal_vector = .;
    __begin_lmosem_hal_intvect = .;
    .lmosem_hal.intvect ALIGN(4) : { *(.lmosem_hal.intvect) }
    __end_lmosem_hal_intvect = .;
    .....
}
```

代码清单 6-17 中，有两道循环，第一道循环是为了清零那块内存，det 为 0，因为 MMU 已经映射了，访问虚拟地址 0 就会访问到物理地址 0x30000000 的内存空间了，第二道循环才是复制中断向量数据的。

如果现在就进行编译链接，LD 链接器一定会报错，因为中断处理代码还没有写，即没有 `hal_bugdie`、`hal_irq`、`hal_frq` 等符号。为了能编译链接成功，我们来写几个超级简单的中断处理代码，新建一个汇编代码文件 `interrupt.S`，在其中写上如代码清单 6-19[xxxx/lmosem/hal/interrupt.S] 所示的汇编代码。

代码清单6-19 ARM920T的中断、异常处理代码

```
.section .lmosem_hal.intvect,"ax"
.code 32
.align 4
.global hal_undefins
.global hal_swi
.global hal_pfabt
.global hal_dataabt
.global hal_bugdie
.global hal_irq
.global hal_frq
hal_undefins:
b .
hal_swi:
b .
hal_pfabt:
b .
hal_dataabt:
b .
hal_bugdie:
b .
hal_irq:
b .
hal_frq:
b .
.end
```

这确实够简单了，每个中断处理程序都是一个死循环，不做任何实质性的工作，然后让 `interrupt.S` 文件也参与一起编译链接就行了。

最后把初始化 MMU 的函数和复制中断向量的函数，整合进 `init_paltform` 函数中，如代码清单 6-20 [xxxx/lmosem/hal/platform.c] 所示。

代码清单6-20 平台初始化

```
void init_paltform()
{
    s3c2440mmu_init();
    s3c2440vector_init();
    return;
}
```

到这里，初始化了 MMU，也写好了中断向量代码，并将其复制到指定的内存地址上了，即最基本的初始化已经搞定了。至此，`init_paltform` 函数结束，后面将会初始化更多的硬件和相关的数据结构。

6.3 串口初始化

任何硬件在使用它之前，必须要先进行初始化，我们主要使用 S3C2440A 的第一个串口和 PC 通信，一是为了了解 s3c2440a 串口输出数据的过程，二是以此来验证 LMOSEM 内核运行的是否正常。

6.3.1 串口硬件

直到现在，LMOSEM 内核工作后，不能看到任何外在的表现，那是因为它没有输出任何数据，所以无法直观地看到它是否正常运行，其实它确实是正常运行的。为了能输出一些数据让我们直观观察到 LMOSEM 内核的运行状态，所以决定先用简单的串口设备，因为它可以直接输出 ASCII 字符数据，能够方便地输出一些英文字符串，这样在 PC 上就可以看到 LMOSEM 内核输出的信息了。

要使用串口设备输出数据，必须先初始化它。S3C2440A 提供了三个串口，它们都可以通过查询、中断和 DMA 的方式传输数据，而且每个串口都分别有一个 64 个字节的接收 FIFO 和一个 64 个字节的发送 FIFO。目前我们只使用非 FIFO 模式。

在开始之前，先来看一看要注意的地方，从 S3C2440A 的原理图不难发现，串口 0 的数据引脚有一点特殊，如图 6-5 所示。它们被连接到 S3C2440A 的 GPH2 和 GPH3 引脚上了，而 GPH2 和 GPH3 引脚是 S3C2440A 的复用引脚，因此要对 GPH2 和 GPH3 引脚对应的寄存器进行设置，其对应寄存器的地址为 GPHCON (0x56000070)。具体的细节可以参阅 S3C2440A 手册的 GPIO 部分，现在我们需要操作这个寄存器，因为 supervivi 已经帮我们配置好了这些引脚。我们只需要配置串口设备本身就行了。

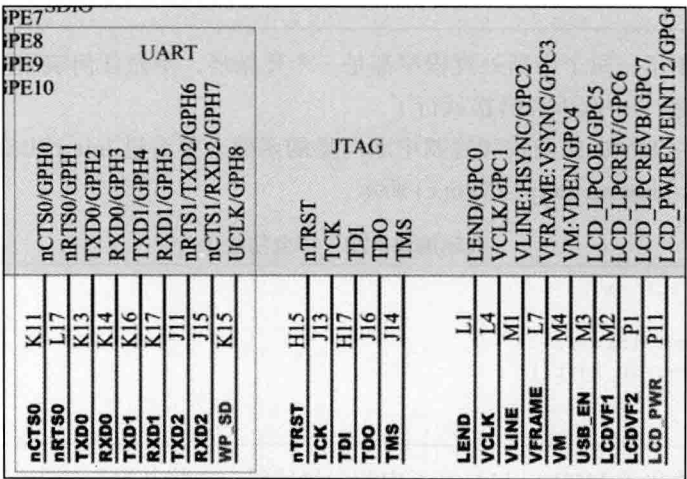


图 6-5 串口 0 的引脚连接图

配置串口设备的步骤一般如下。

第1步：设置串口设备的通信模式、发送、接收数据的格式，以及数据的校验方式。

第2步：设置串口设备的时钟源、发送、接收数据的方式等。

第3步：设置串口设备的FIFO、流控制等。

第4步：设置串口设备的波特率。

如果对以上步骤不太懂，可以看看前面章节的介绍，或者参阅S3C2440A手册。

操作硬件，就是操作与之对应的一些寄存器，先来看看第1步中需要关注的寄存器，我们只用到串口0，所以只需要关注串口0所对应的寄存器，如表6-2所示。

表6-2 串口0对应的寄存器

寄存器相关	地 址	读 写 方 式	描 述	复 位 值
ULCON0	0x50000000	R/W	串口0串行控制寄存器	0
ULCON1	0x50004000	R/W	串口1串行控制寄存器	0
ULCON2	0x50008000	R/W	串口2串行控制寄存器	0
ULCONn	位	描述		复位值
保留	[7]	无		0
红外模式	[6]	决定是否使用红外模式：0=普通模式；1=红外Tx/Rx模式		0
校验模式	[5:3]	指定在串口发送和接收操作期间奇偶校验产生和检查的类型：0xx=无奇偶校验；100=奇校验；101=偶校验；110=固定/检查奇偶校验为1；111=固定/检查奇偶校验为0		0
停止位	[2]	指定用于结束帧信号的停止位的个数：0=每帧1个停止位；1=每帧2个停止位		0
数据位	[1:0]	指定每帧用于发送或接收的数据位的个数：00=5位；01=6位；10=7位；11=8位		0

我们选择一般通信模式，无校验位，1个停止位，8个数据位的数据通信方式。因此ULCON0 = 0x03。

接着设置串口0的时钟源、发送、接收数据的方式。这需要操作串口0的控制寄存器，串口0的控制寄存器如表6-3所示。

表6-3 串口0的控制寄存器

寄存器相关	地 址	读 写 方 式	描 述	复 位 值
UCON0	0x50000004	R/W	串口0控制寄存器	0
UCON0	位	描述		复位值
FCLK分频器	[15:12]	当串口0选择FCLK作为时钟源时，设置其FCLK的分频因子，串口0工作时钟频率=FCLK/FCLK分频因子+6		0

(续)

寄存器相关	地 址	读 写 方 式	描 述	复 位 值
时钟选择	[11:10]		选择串口0的工作时钟 PCLK, UEXTCLK, FCLK/n : 00,10=PCLK; 01=UEXTCLK; 11=FCLK/n; 当选择 FCLK/n 作为串口0工作时钟时, 还要做其他设置, 具体请读者自行查看手册 $UBRDIVn = (\text{int})(\text{被选时钟} / (\text{波特率} \times 16)) - 1$	0
发送数据中断产生类型	[9]		设置串口0中断请求类型, 在非 FIFO 传输模式下, 一旦发送数据缓冲区为空, 立即产生中断信号, 在 FIFO 传输模式下达到发送数据触发条件时立即产生中断信号: 0=脉冲触发; 1=电平触发	0
接收数据中断产生类型	[8]		设置串口0中断请求类型, 在非 FIFO 传输模式下, 一旦接收到数据, 立即产生中断信号, 在 FIFO 传输模式下达到接收数据触发条件时立即产生中断信号: 0=脉冲触发; 1=电平触发	0
接收数据超时	[7]		设置当接收数据时, 如果数据超时, 是否产生中断: 0=不开启超时中断; 1=开启超时中断	0
接收数据错误中断	[6]		设置当接收数据时, 如果产生异常, 如传输中止, 帧错误, 校验错误时, 是否产生中断信号: 0=不产生错误状态中断; 1=产生错误状态中断	0
回送模式	[5]		设置该位时串口0会进入回送模式, 该模式仅用于测试: 0=正常模式; 1=回送模式	0
发送终止信号	[4]		设置该位时, 串口0会发送一个帧长度的终止信号, 发送完毕后, 该位自动恢复为0: 0=正常传输; 1=发送终止信号	0
发送模式	[3:2]		设置采用哪个方式执行数据写入发送缓冲区: 00=无效; 01=中断请求或查询模式; 10=DMA0 请求模式	0
接收模式	[1:0]		设置采用哪个方式执行数据写入接收缓冲区: 00=无效; 01=中断请求或查询模式; 10=DMA0 请求模式	0

这里为了简单, 串口0采用 PCLK 作为输入工作时钟, 采用简单的轮询方式进行数据的接收和发送, 不开启数据接收超时, 数据产生错误时不产生错误状态中断, 因此 UCON0 = 0x05。

再来看看, 设置串口0的 FIFO、流控制的两个寄存器, FIFO 控制寄存器 UFCON0 (0x50000008) 和 MODEM 控制寄存器 UMCN0 (0x5000000C)。但由于我们不使用 FIFO 和流控制, 因此这两个寄存器中都写入0值。

最后来看看设置串口0的波特率, 即设置波特率除数寄存器 UBRDIV0, 如表 6-4 所示。

表6-4 波特率除数寄存器

寄存器相关	地 址	读 写 方 式	描 述	复 位 值
UBRDIV0	0x50000028	R/W	串口0波特率除数寄存器	未知

(续)

寄存器相关	地 址	读 写 方 式	描 述	复 位 值
UBRDIV0	位		描述	复位值
波特率除数	[15:0]		设置波特率除数(大于0),而使用外部输入时钟时可以置0	未知

UBRDIV_n的取值范围应该为 $1 \sim 2^{16}-1$, 我们设置串口0的波特率为 115200bit/s, 采用 PCLK 时钟为其工作时钟, supervivi 初始化 PCLK 为 50MHz, 因此 UBRDIV_n 为 $\text{UBRDIV}_n = (\text{int})(50\text{M} / (115200 \times 16)) - 1 = 26$ 。

了解了这些寄存器, 就可以开始写代码初始化串口设备了, 首先新建一个文件 `haluart.c` 和相关的头文件, 并把它加入编译头文件中相关宏的后面, 如代码清单 6-21[xxxx/lmosem/hal/haluart.c] 所示。

代码清单6-21 串口设备初始化

```
void init_haluart()
{
    init_uart0();
    return;
}
void init_uart0()
{
    hal_io32_write(ULCON0_R, 3); //设置串口0的通信模式、发送、接收格式、校验方式
    hal_io32_write(UCON0_R, 5); //设置串口0的时钟源、发送、接收数据的方式等
    hal_io32_write(UFCON0_R, 0); //不采用FIFO模式
    hal_io32_write(UMCON0_R, 0); //不使用控制流
    hal_io32_write(UBRDIV0_R, 26); //设置串口0的波特率为115200bit/s
    return;
}
```

ULCON0_R、UCON0_R、UBRDIV0_R 等寄存器地址的宏定义在 `haluart_t.h` 文件中, 读者可自行查看。

S3C2440A 所有外设寄存器的地址都映射在内存地址空间中, 用 C 语言指针就可以直接操作它们, 然而, 为了和操作内存地址有所区别, 我们定义了一些专门读写设备寄存器的函数, 例如, `hal_io32_write`、`hal_io32_read`、`hal_io16_write`、`hal_io16_read`、`hal_io8_write`、`hal_io8_read`, 知所以会有这么多个函数, 是因为有许多寄存器是 16 位的或者 8 位的。为此要新建一个文件 `cpuctrl.c` 和一些头文件。在其中写上如代码清单 6-22[xxxx/lmosem/hal/cpuctrl.c] 所示的代码。并且这些函数都是在 C 语言中, 嵌入相应的汇编代码而实现的。

代码清单6-22 读写设备寄存器的函数

```
u8_t hal_io8_read(uint_t ioadr)
{
    u8_t retdata;
    __asm__ __volatile__(
        "ldrb %[retdata], [%[ioaddr]] \n\t"
```

```

        : [retdata] "=r"(retdata) : [ioaddr] "r" (ioadr) : "memory"
    );
    return retdata;
}
u16_t hal_io16_read(uint_t ioadr)
{
    u16_t retdata;
    __asm__ __volatile__(
        "ldrh %[retdata],[@[ioaddr]] \n\t"
        : [retdata] "=r"(retdata) : [ioaddr] "r" (ioadr) : "memory"
    );
    return retdata;
}
u32_t hal_io32_read(uint_t ioadr)
{
    u32_t retdata;
    __asm__ __volatile__(
        "ldr %[retdata],[@[ioaddr]] \n\t"
        : [retdata] "=r"(retdata) : [ioaddr] "r" (ioadr) : "memory"
    );
    return retdata;
}
void hal_io8_write(uint_t ioadr,u8_t val)
{
    __asm__ __volatile__(
        "strb %[valdata],[@[ioaddr]] \n\t"
        : : [valdata] "r"(val),[ioaddr] "r" (ioadr) : "memory"
    );
    return;
}
void hal_io16_write(uint_t ioadr,u16_t val)
{
    __asm__ __volatile__(
        "strh %[valdata],[@[ioaddr]] \n\t"
        : : [valdata] "r"(val),[ioaddr] "r" (ioadr) : "memory"
    );
    return;
}
void hal_io32_write(uint_t ioadr,u32_t val)
{
    __asm__ __volatile__(
        "str %[valdata],[@[ioaddr]] \n\t"
        : : [valdata] "r"(val),[ioaddr] "r" (ioadr) : "memory"
    );
    return;
}

```

串口0就初始化好了，但是在用串口0发送数据之前，我们还要了解几个和发送数据相关的寄存器。先来看看串口0的数据发送和数据接收的寄存器，如表6-5所示。

表6-5 串口0的数据发送、接收寄存器

寄存器名	地 址	是否读写	描 述	复 位 值
UTXH0	0x50000020 (L 小端模式) 0x50000023 (B 大端模式)	W	串口 0 数据发送寄存器	未知
URXH0	0x50000024 (L 小端模式) 0x50000027 (B 大端模式)	R	串口 0 数据接收寄存器	未知

有了这些寄存器，就可以用串口 0 发送或者接收数据了，但是发送数据或者接收数据之前，首先要检测串口 0 是否空闲或者其中有无数据，我们没有使用串口设备的中断模式，所以只有通过串口设备的状态寄存器来获取相关的信息，串口 0 的状态寄存器如表 6-6 所示。

表6-6 串口0的状态寄存器

寄存器相关	地 址	是否读写	描 述	复位值
UTRSTAT0	0x50000010	R/W	串口 0 的状态寄存器	6
发送器为空		[2]	当发送缓存寄存器中没有数据要发送且发送移位寄存器为空时，自动置 1。0=非空；1=发送器为空且移位寄存器也为空	1
发送缓存寄存器为空		[1]	当发送缓存寄存器为空时，自动置 1。0=发送缓存寄存器非空；1=发送缓存寄存器为空	1
接收缓存寄存器为空		[0]	当接收缓存寄存器有数据到达时，自动置 1。0=接收缓存寄存器为空；1=缓存寄存器中有数据到达	0

到这里，对于串口 0 设备，有了数据发送寄存器和状态寄存器，就可以写发送数据的代码了，先写一个简单的函数，如代码清单 6-23[xxxx/lmosem/hal/haluart.c] 所示。

代码清单6-23 向串口0发送一个8位数据

```
void hal_uart0_putc(char_t c)
{
    //读状态寄存器，检测串口0发送器是否为空，不为空就循环检测，直到为空
    while(!(hal_io32_read(UTRSTAT0_R)&4));
    //把数据写到串口0的数据发送寄存器中
    hal_io32_write(UTXH0_R,c);
    return;
}
```

读者也可以现在测试一下，在 init_haluart 函数中加入 hal_uart0_putc(‘a’)，如代码清单 6-24[xxxx/lmosem/hal/haluart.c] 所示。最后别忘了，把 init_haluart 函数加入硬件相关层初始化的 init_hal 主函数中。然后编译运行，一定会在 Linux 的终端中看到“a”字符。

代码清单6-24 测试串口0的数据发送

```
void init_haluart()
{
    init_uart0();
}
```



```
    hal_uart0_putc( 'a' );  
    return;  
}
```

费了这么多工夫，终于初始化了串口 0 设备，还了解了串口 0 设备的许多寄存器，最重要的是以后我们的 LMOSEM 内核可以向 PC 上 Linux 的终端中输出相关的信息了。调试 LMOSEM 内核也就方便多了，这可是个大跃进。

6.3.2 内核的 printf

经常在开发应用程序的过程中用到 printf 函数，它功能强大且神奇。神奇之处是它的参数个数是可变的，但是应用程序调用的是 C 语言标准库里面的 printf 函数。我们的 LMOSEM 内核也要输出相关的信息，因此也需要用到此类函数。不防在此实现 LMOSEM 内核的 printf 函数。其实前面已经实现了串口输出功能，可以以此为根基，让 LMOSEM 内核的 printf 函数调用串口输出功能来显示相关的信息。

先来看看 LMOSEM 内核 printf 函数的申明，如下所示。

```
void printfk(const char_t* fmt,...);
```

因为是 LMOSEM 内核所用的，所以名称为 printfk，函数第一个参数后面是“...”，它表示函数参数的个数不确定，让 C 编译器不对其参数的个数进行审查。C 语言调用约定，规定调用者为被调用者准备参数，这是这个函数得以实现的基础。

一般使用 printf 函数都是如下形式。

```
.....  
int val1,val2;  
printfk( "val1 is %d val2 is %d",val1,val2);  
.....
```

从 printfk 函数的申明可以看到，printfk 函数最多仅能够获取“val1 is %d val2 is %d”这一个参数，而 val1 和 val2 这两个参数，printfk 函数是不知道的，那么 printfk 函数是如何获取它们的呢，这得从 C 语言如何处理参数开始说起。一般情况下，C 语言用栈传递参数时，始终保持函数最右边的参数先入栈，以上情况如图 6-6 所示。

看了图 6-6，可能明白了许多，我们只要获取参数 fmt 的地址，那么获取其他参数也就顺理成章了，只要把参数 fmt 的地址加上一个 int 类型大小的偏移量就可以获取下一个参数了。问题来了，从前面章节中知道，ARM 平台上 C 语言调用约定，规定用 CPU 的 4 个寄存器传递前 4 个参数，而不是把所有的参数都放进栈中，这使得 printfk 函数无法实现了。不过不用怕，GCC 编译器对其提供了支持，我们只要使用几个 GCC 编译器内建的函数，GCC 编译器就可以自

栈	地址
.....	0x100C
val2	0x1008
val1	0x1004
fmt	0x1000
.....	0xFFC

图 6-6 调用 printfk 函数时栈中的情形

动识别这种情况了，它会把用于存放参数的4个寄存器也压入栈中，如代码清单6-25[xxxx/lmosem/include/halinc/printfk_t.h]所示。

代码清单6-25 printfk函数中相关的宏

```
typedef __builtin_va_list va_list;
#define va_start(ap,np) __builtin_va_start(ap,np)
#define va_end(ap) __builtin_va_end(ap)
#define va_arg(ap,ty) __builtin_va_arg(ap,ty)
```

其中，__builtin_va_start(ap,np)就是把fmt的地址放入ap中的，ap是__builtin_va_list类型的变量，这个类型相当于char*类型，是C语言中的指针类型。va_end(ap)是把ap的值清0，防止下次代码访问ap时出现错误，没有什么其他重要作用。

来看看这些宏在printfk函数中是如何使用的，如代码清单6-26所示。

代码清单6-26 va_start、va_arg等宏在printfk函数中的使用

```
void printfk(const char_t* fmt,...)
{
    va_list ap;
    va_start(ap,fmt);
    //下面进行获取各个参数并执行格式化操作之后，执行输出
    .....
    va_end(ap); //ap指针清0，表示操作结束
    return;
}
```

printfk函数还有一个强大的功能，就是它能按照要求格式化输出参数的数据，例如，按十进制(%d)输出数据、按十六进制(%x)输出数据、输出一个字符串(%s)数据等，因此还要写几个函数，如代码清单6-27 [xxxx/lmosem/hal/printfk.c]所示。

代码清单6-27 格式化输出的数据

```
void vsprintfk(char_t* buf,const char_t* fmt,va_list args)
{
    char_t* p =buf;
    va_list next_arg=args;
    while(*fmt)
    {
        if(*fmt != '%')
        {
            *p++ = *fmt++;
            continue;
        }
        fmt++;
        switch(*fmt)
        {
            case 'x':
                p=numberk(p,va_arg(next_arg,uint_t),16);
                fmt++;
```

```

        break;
    case 'd':
        p=numberk(p,va_arg(next_arg,uint_t),10);
        fmt++;
        break;
    case 's':
        p=strcpyk(p,(char_t*)va_arg(next_arg,uint_t));
        fmt++;
        break;
    default:
        break;
    }
}
*p=0;
return;
}
char_t* numberk(char_t* str,uint_t n, sint_t base)
{
    register char_t *p;
    char_t strbuf[36];
    p = &strbuf[36];
    *--p = 0;
    if(n == 0)
    {
        *--p = '0';
    }
    else
    {
        do
        {
            *--p = "0123456789abcdef"[n % base];
        }while( n /= base);
    }
    while(*p!=0)
    {
        *str++=*p++;
    }
    return str;
}
char_t* strcpyk(char_t* buf,char_t* str_s)
{
    while(*str_s)
    {
        *buf=*str_s;
        buf++;
        str_s++;
    }
    return buf;
}

```

代码有点长，我们慢慢看，首先看 vsprintfk 函数，它的代码主要是个大循环，在循环中

一边把格式化字符串（如“val1 is %d val2 is %d”）复制到缓冲区 buf 中（p 表示 buf 的当前位置），一边寻找格式化前缀“%”符，找到“%”符后，根据相应的格式符（如 d, x, s 等）分别进行处理。我们知道数据在内存中是二进制的，是不能直接被显示的，必须要转换成相应的 ASCII 码才行，这个工作是由 numberk 函数来做的。然而，在转换之前首先要获取参数的数据，方法是用 va_arg(next_arg, uint_t)，其中 next_arg 表示参数的地址，uint_t 表示参数的类型，并且它能够自动操作 next_arg，使之指向下一个参数。numberk 函数主要是对数据进行取模运算，然后转换成 ASCII 码，并复制到 p 指向的当前 buf 的空间中。如果是“s”格式，对应的参数就是字符串的地址，则直接使用 strcpyk 函数把其指向的字符串数据复制到 p 指向当前 buf 的空间中就行了。其实还可以实现更多的格式，然而 LMOSEM 内核中有了这三种格式就够了。

现在来让 printfk 函数和 vsprintfk 函数连起来一起工作，如代码清单 6-28[xxxx/lmosem/hal/printfk.c] 所示。

代码清单6-28 打印输出相关信息的函数printfk

```
void printfk(const char_t* fmt,...)
{
    char_t buf[512];
    va_list ap;
    va_start(ap,fmt);
    vsprintfk(buf,fmt,ap);
    hal_uart_write(0,buf,0);    //把buf缓冲区中的数据写入串口
    va_end(ap);
    return;
}
```

可以发现，buf 和 ap 都是在 printfk 函数中定义的，buf 数组也只能存放 512 个字符，但这对 LMOSEM 内核来说已经够了，ap 中存放的是 fmt 的地址，buf、fmt、ap 都要传递给 vsprintfk 函数，当 vsprintfk 函数处理好后，buf 中就存放了格式化好了以 0 结尾的字符串，最后调用 hal_uart_write 函数把 buf 中的数据发送给串口硬件，如代码清单 6-29[xxxx/lmosem/hal/haluart.c] 所示。

代码清单6-29 向串口设备发送数据

```
drvstus_t hal_uart_write(uint_t uart,void* buf,uint_t len)
{
    char_t* p=buf;
    while(*p)
    {
        if(uart_send_char(uart,*p)==DFCERRSTUS)
        {
            hal_sysdie("uart err");
            return DFCERRSTUS;
        }
        p++;
    }
}
```

```

    }
    return DFCOKSTUS;
}
drvstus_t uart_send_char(uint_t uart, char_t ch)
{
    uint_t time=0;
    if(uart!=0)
    {
        return DFCERRSTUS;
    }
    while(!(hal_io32_read(UTRSTAT0_R)&4))
    {
        if(time>0x100000)
        {
            return DFCERRSTUS;
        }
        time++;
    }
    hal_io32_write(UTXH0_R,ch);
    return DFCOKSTUS;
}

```

代码清单 6-29 中，`drvstus_t` 类型实际上就是 `int` 类型，而 `DFCOKSTUS`、`DFCERRSTUS` 的值就是 1 和 -1。`hal_uart_write` 函数再次调用 `uart_send_char` 函数完成数据的发送，而 `uart_send_char` 函数中的循环代码主要是防止串口硬件出现问题。我们认为循环检测多于 0x100000 次时，如果串口发送器还是不为空，那么就认为串口硬件有问题了。

如果读者现在就编译这些代码，在链接时一定会出错，LD 会提示找不到一些符号，可能读者从来没见过这些符号，也没有直接在哪里使用过它们。这是什么原因呢？读者可能还记得，在 `numberk` 函数中，用到了取模运算和除法运算，而我们的 ARM920T CPU 是不支持除法或者取模指令的。原因就在这里，因为在这种情况下，GCC 会自动调用软件实现的除法函数和取模函数，LD 提示的符号就是这些函数的名字。既然找到了原因，我们就从 `bootloader`（一款 ARM 平台的引导器，类似于 `supervivi`）代码文件中复制 4 个文件 `_umodsi3.S`、`_udivsi3.S`、`_modsi3.S`、`_divsi3.S` 到 `LMOSEM` 内核的 `hal` 目录下，并且把它们加入到编译头文件中相关宏的后面，一起参与 `LMOSEM` 内核的编译。事实上，`bootloader` 也是复制 Linux 内核的。由于篇幅所限，这里就不分析这些文件中的代码了，有兴趣的读者可以自行查看。

下面来试一试我们亲自写的 `printfk` 函数，在 `init_haluart` 函数中写下如代码清单 6-30 [xxxx/lmosem/hal/haluart.c] 所示的代码。

代码清单6-30 测试printfk函数

```

void init_haluart()
{
    init_uart0();
    uint_t vall = 12;
    char_t* str = "test printf";
}

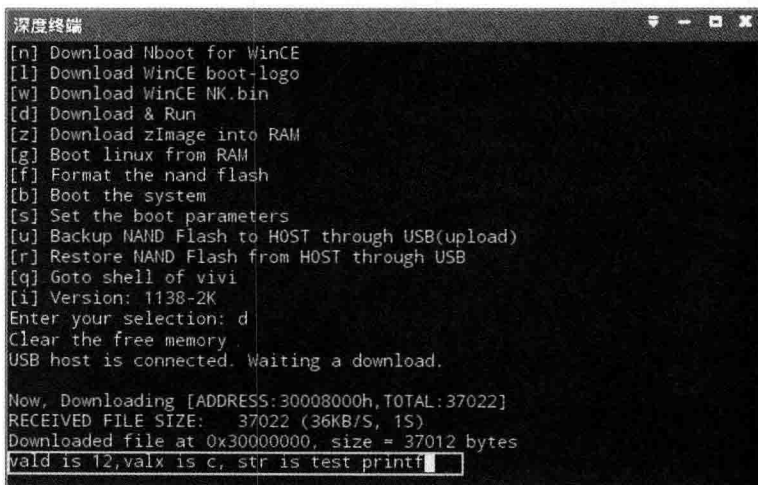
```

```

printfk("vald is %d, valx is %x, str is %s", vall, vall, str);
for (;;)
return;
}

```

按照前面的编译方法，编译运行 LMOSEM 内核。正常情况下会出现如图 6-7 所示的界面。



```

深度终端
[n] Download Nboot for WinCE
[l] Download WinCE boot-logo
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1138-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30008000h, TOTAL:37022]
RECEIVED FILE SIZE: 37022 (36KB/S, 1S)
Downloaded file at 0x30000000, size = 37012 bytes
vald is 12, valx is c, str is test printf

```

图 6-7 测试 printfk 函数

又一次不小的尝试，这一尝试让我们有了 printfk 函数，感受了变参函数的精妙之处，也感受了 C 语言设计者精湛的设计，更重要的是 LMOSEM 内核在今后的开发过程中，可以随意在 Linux 的终端上输出相关的信息了，这至少能告诉我们 LMOSEM 内核运行到哪了，出 bug 了也能打印一些信息，这对于我们开发 LMOSEM 内核的帮助无疑是巨大的。

6.4 机器数据结构

机器数据结构这个名称很奇怪，这里读者只需要认为它是一个很重要的数据结构就行了，它的内部存放了一些非常重要的数据。LMOSEM 内核在初始化其他内部组件时，需要以它为基础。

6.4.1 设计数据结构

先来设计这个重要的数据结构，注意这个数据结构中有很多个域，可能现在读者不明白，这里只是需要熟悉一下这个数据结构及用它定义的实例变量放在哪里，不必清楚它有什么作用。作用在初始化其他 LMOSEM 内部组件时才会显现。

下面来看一看这个数据结构，如代码清单 6-31[xxxx/lmosem/include/halinc/mach_t.h] 所示。当然这需要建立一些头文件。这个前面已经说的很多了，不再详述。

代码清单6-31 机器数据结构

```
typedef struct s_MACH
{
    spinlock_t mh_lock;           //保护数据结构自身的自旋锁
    list_h_t   mh_list;           //链表
    adr_t      mh_kerinramstart;   //内核在内存中的开始地址
    adr_t      mh_kerinramend;     //内核在内存中的结束地址
    mmapdsc_t* mh_mmapdscadr;      //内存位图在内存中的开始地址
    uint_t     mh_mmapdscnr;        //内存位图的个数
    uint_t     mh_spedsdscnr;       //物理地址空间描述符的个数
    phyadrspce_t* mh_spedsdsc;     //物理地址空间描述符的地址
    ilnedsc_t* mh_ilnedsc;
    uint_t     mh_ilnedscnr;
    intfltdsc_t* mh_intfltdsc;      //中断源描述符地址
    uint_t     mh_intfltnr;         //中断源描述符个数
}mach_t;
```

对于这个数据结构本身，代码清单 6-31 中的注释已经说得很清楚了，至于它们都是干什么的，我们先不必深究。

可是有了数据结构类型，还需要用这种类型定义一个实例变量，这个数据结构是全局性的，在许多代码模块中都要使用它，其实，LMOSEM 内核代码中还有许多全局性的数据结构，然而它们必须包含在 LMOSEM 内核的可执行文件之中，所以我们用一个专门的 C 语言源文件 (halglobal.c) 来定义这些全局性的数据结构的实例变量，如代码清单 6-32[xxxx/lmosem/hal/halglobal.c] 所示。

代码清单6-32 定义全局的mach_t类型的变量

```
.....
HAL_DEFGLOB_VARIABLE(mach_t,osmach);
.....
```

HAL_DEFGLOB_VARIABLE 是一个宏，定义在 halglobal.h 文件中，这里它会处理成如代码清单 6-33 所示。__attribute__((section(".data"))) 是 GCC 的扩展功能，表示把 osmach 变量分配在 LMOSEM 内核的可执行文件的 .data 段中。

代码清单6-33 定义全局变量的宏

```
#define HAL_DEFGLOB_VARIABLE(vartype,varname) \
EXTERN __attribute__((section(".data"))) vartype varname
.....
//经过上面的宏处理后，halglobal.c中的HAL_DEFGLOB_VARIABLE(mach_t,osmach);就变成下面这样
EXTERN __attribute__((section(".data"))) mach_t osmach;
.....
```

EXTERN 也是一个宏，它在 `bastype_t.h` 文件中定义成 `#define EXTERN extern`。可能奇怪了，如果这样，那么在 `halglobal.c` 文件中的 `osmach` 变量不过是个外部变量，GCC 不会为这个外部变量分配具体的存储空间，链接器最终会报错，提示找不到 `osmach` 符号，怎么回事呢？我们再去写一下 `halglobal.h` 文件中的代码就明白了，在 `halglobal.h` 文件中写下如代码清单 6-34[xxxx/lmosem/include/halinc/halglobal.h] 所示的代码。

代码清单6-34 EXTERN宏的实现机理

```
#ifndef _HALGLOBAL_H
#define _HALGLOBAL_H
#ifdef HALGOBAL_HEAD //如果定义HALGOBAL_HEAD
#undef EXTERN //就不定义EXTERN宏
#define EXTERN //否则定义EXTERN宏
#endif
HAL_DEFGLOB_VARIABLE(mach_t,osmach);
.....
#endif
```

代码清单 6-34 中的注释，已经非常明显了，只要 `halglobal.c` 文件包含了 `halglobal.h` 文件并且在包含 `halglobal.h` 文件的前面定义 `HALGOBAL_HEAD` 宏，问题就解决了，见代码清单 6-35[xxxx/lmosem/hal/halglobal.c]。这样就使在 `halglobal.c` 文件中，是不会定义 `EXTERN` 宏的，而在其他源代码文件中，就会定义 `EXTERN` 宏，因此在其他源代码文件中看到的就是 `extern__attribute__((section(".data"))) mach_t osmach`；所以它们就认为 `osmach` 是一个外部变量。这种优雅的做法并不是笔者发明的，而是 MINIX 的作者。

代码清单6-35 定义HALGOBAL_HEAD宏

```
#define HALGOBAL_HEAD //定义HALGOBAL_HEAD
#include "lmosemtypes.h"
#include "lmosemmctrl.h"
HAL_DEFGLOB_VARIABLE(mach_t,osmach);
.....
```

到这里没有操作这个数据结构，我们只是设计并且定义了这个数据结构的类型的变量，对于本小节，明白这些就够了。

6.4.2 确定一些重要数据结构与内核的地址

前面已经设计好了 `mach_t` 数据结构，本小节就是要把一些关键的数据，放入前面设计的 `mach_t` 数据结构中，以供后面的 `LMOSEM` 内核代码使用。

前面已经定义好了一个 `mach_t` 数据结构类型的变量 `osmach`，这里只要写代码操作它就行了，先新建一个文件 `halmach.c` 和相关的头文件，在其中写上如代码清单 6-36[xxxx/lmosem/hal/halmach.c] 所示的代码。

代码清单6-36 初始化mach_t数据结构

```

void init_halmach()
{
    mach_t_init(&osmach);
    return;
}

void mach_t_init(mach_t* initp)
{
    hal_spinlock_init(&initp->mh_lock);
    list_init(&initp->mh_list);
    initp->mh_kerintramstart=KRNL_INRAM_START;
    initp->mh_kerintramend=(adr_t)(&__end_kernel);
    initp->mh_mmapdscadr=(mmapdsc_t*)(ALIGN(((uint_t)(&__end_kernel)),4096));
    initp->mh_mmapdscnr=0;
    .....
    return;
}

```

这里需要注意一下，mach_t 数据结构是全局性的，会被不同的 LMOSEM 内核组件进行修改，也就是说不到最后，osmach 中的数据是无法确定下来的，事实上，LMOSEM 内核中的一些组件都是互相依存的，很多情况下，没办法做到先来后到，而是要把它们一起做。虽然笔者从大体上尽量按照先来后到的顺序写书，但还是有些部分是无法按照顺序来写的。mach_t 数据结构中有很多域，但是这里我们仅需要关注它的前 6 个域，而剩下的是后面该做的。

代码清单 6-36 中链表的初始化非常简单，前面章节中已经看过了。hal_spinlock_init 函数倒是第一次看到，不妨先把它搞明白。该函数是在 cpuctrl.c 文件中定义的，如代码清单 6-37[xxxx/lmosem/hal/cpuctrl.c] 所示。

代码清单6-37 初始化自旋锁的接口函数

```

void hal_spinlock_init(spinlock_t* lock)
{
    __asm__ __volatile__(
        "swp %[lockval],[%[lockval]],[%[lockadr]] \n\t"
        : : [lockval] "r"(0),[lockadr] "r" (&lock->lock) : "memory"
    );
    return;
}

```

又是用在 C 语言中嵌入汇编代码实现的，C 语言无法实现这种函数，因为它需要保证该操作是原子的。其实自旋锁的初始化，也不一定非要是原子操作的，只是这样做保险一些。其中，swp 指令是 ARM920T CPU 中的原子交换指令，它至少需要两个寄存器，前两个寄存器可以是相同的，它的操作是：把第二个寄存器中的值写到以第三寄存器中的值为地址的内存中，与此同时，将该内存中的值写到第一个寄存器中，这个交换动作是原子的，不可打断的。

代码清单 6-36 中的 `mach_t_init` 函数接下来处理内核在内存中的开始地址、内核在内存中的结束地址、内存位图在内存中的开始地址、内存位图的个数。`adr_t` 类型其实就是无符号的整型，在 `platform_t.h` 文件中有这样一行代码 `#define KRNL_INRAM_START 0x30000000`，为什么是 `0x30000000` 而不是 `0x30008000`，因为地址 `0x30000000 ~ 0x30008000` 这块内存中，存放的是中断向量和 MMU 页表，理应算作 LMOSEM 内核本身的东西。`__end_kernel` 是在链接脚本文件中定义的，此种类型的变量在前面已经遇到过多次了，不再详述。总之链接器会在这里填上 LMOSEM 内核的结束地址。

再来看看内存位图在内存中的开始地址。LMOSEM 内核的内存位图简单来说，是一个数据结构类型的数组（这里并不急于关注该数据结构类型），这个数组的大小和内存的容量相关。所以不能像 `osmach` 一样静态地定义在 LMOSEM 内核的可执行文件中。因此我们要在获取内存容量大小之后，动态地建立这个数据结构类型数组，要动态建立这个数组，就必须要有该数组的首地址，即从这个地址开始存放这个数组。正如代码清单 6-36 中的那样，我们把这个数组存放在 LMOSEM 内核后面的空闲内存中，不过这个地址是 LMOSEM 内核结尾地址处的第一个页面（4096 个字节）。`ALIGN` 是个宏，它在 `bastype_t.h` 文件中定义成 `#define ALIGN(x, a) (((x) + (a) - 1) & ~((a) - 1))`，它能返回 `x` 地址对齐到 `a` 值后的地址。而内存位图的个数（即这个数组元素的个数）现在还不能确定，因为还没有初始化到内存位图，故而写 0，到时内存位图初始化的代码，会重新将 `osmach` 的 `mh_mmapdscnr` 域中写入正确的值。

有兴趣的读者可以在 `init_halmach` 函数的后面加入 `printfk` 函数输出 `osmach` 中的一些值，看看是不是预期的，最后别忘了把 `init_halmach` 函数加入 `init_hal` 函数中。由于这个非常简单，笔者就不在此演示了。

好了，虽然 `osmach` 变量中的一些域还没有初始化完，不过那是后面的事情了，本小节到这里就结束了，在后面的过程中会继续看到 `osmach` 的身影。

6.5 初级内存管理初始化

现代计算机中，大部分程序都是放在内存中运行的，大部分数据也是首先暂存于内存中再存储到别的设备中去的。所以合理的管理内存是操作系统内核的重要任务，随着我们的操作系统内核的开发越往后继续，读者会越觉得内存管理的重要性，比理论书籍中反复强调的进程调度重要得多。

当然，要想管理内存首先要初始化内存管理的相关数据结构才行。

6.5.1 设计一些数据结构

内存管理最核心的部分，简单来说无非就是一块内存空间的分配与释放。要用代码实现内存空间的分配与释放，首先得知道哪些内存空间是空闲的，哪些内存空间是被已经占用

的，也就是说得设计出一些数据结构来表示内存空间的状态。

首先来设计用于表示内存空间状态的数据结构，当然先要划分内存，即把整个内存空间分为大小相等的内存空间块，每个这样的数据结构对应一块内存空间块，我们的 mini2440 上有 64MB 的内存，可分为 16 块，每块 4MB。下面来看看这个数据结构的设计，如代码清单 6-38[xxxx/lmosem/include/halinc/halmm_t.h] 所示。

代码清单6-38 表示内存空间状态的数据结构

```
typedef struct s_MMAPDSC
{
    list_h_t map_list;
    spinlock_t map_lock;
    adr_t      map_phyadr;
    adr_t      map_phyadrend;
    u32_t      map_allcount;
    u32_t      map_flg;
    /*它管理4MB大小的内存块，在实际分配时，它所描述的4MB
    *可以被分割成128KB、256KB、512KB、1MB、2MB、4MB，只能
    *分成同一大小的等份*/
}mmapdsc_t;
```

该数据结构很重要，但数据结构中的注释不够清楚和详细，还是列个表更清楚一点，如表 6-7 所示。

表6-7 mmapdsc_t数据结构中的相关域

域	说 明			
map_list	链表			
map_lock	保护其自身的自旋锁			
map_phyadr/ map_phyadrend	该内存块的开始和结束地址，其实这两个域可以不用的，只不过在分配内存时要计算一下地址，有了它就不用计算了，这样做是为了提高性能			
map_allcount	该域的含义是根据 map_flg 域来决定的			
map_flg	这个域是 32 位的，各位段说明如下：			
	位段	说明		
	[31:8]	当前没有使用，保留以后代码中使用		
	[7:4]	指定 mmapdsc_t 所表示的内存块的内部再次分为多大的小块，各值如下		
		值	说明	map_allcount 域
		1	把大内存块的内部再分成 128KB 小块，共 32 小块	map_allcount 的 [31:0] 位，每个位对应小块的分配状态，为 0 表示空闲，为 1 表示已分配

(续)

域	说 明		
map_flg	[7:4]	值	说明
		2	把大内存块的内部再分成 256KB 小块, 共 16 小块
		3	把大内存块的内部再分成 512KB 小块, 共 8 小块
		4	把大内存块的内部再分成 1MB 小块, 共 4 小块
		5	把大内存块的内部再分成 2MB 小块, 共 2 小块
		6	把大内存块的内部再分成 4MB 小块, 共 1 小块
	map_allcount 域	map_allcount 的 [15:0] 位, 每个位对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
		map_allcount 的 [7:0] 位, 每个位对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
		map_allcount 的 [3:0] 位, 每个位对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
		map_allcount 的 [1:0] 位, 每个位对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
		map_allcount 的 [0] 位, 对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
		map_allcount 的 [0] 位, 对应小块的分配状态, 为 0 表示空闲, 为 1 表示已分配	
map_flg	[3:0]	指定 mmapdsc_t 所表示的内存块大小, 如下:	
		值	说明
		0	没有指定大小
		1	表示 16KB 大小的内存块
		2	表示 32KB 大小的内存块
		8	表示 4MB 大小的内存块, 一般使用这个

看了表 6-7, 才会感觉到 `mmapdsc_t` 数据结构的复杂, 这也是为了适应以后的变化。不过不用怕, 读者会发现后面的实现中, 并不会用到这里所有的情形。

其实前面的章节中已经说过 `mmapdsc_t` 数据结构的实例在内存中是一个数组, 该数组是内存管理初始化代码建立的, 数组的首地址就放在 `osmach` 数据结构的 `mh_mmapdscadr` 域中。

这时不妨想象一下, 这个数据结构数组已经存在了, 现在要写分配内存的代码, 该怎么写呢? 最简单的方法就是遍历该数组, 依次检测数组中的每个元素即 `mmapdsc_t` 结构, 看哪个 `mmapdsc_t` 结构中有空闲的内存块, 就返回该空闲内存块的首地址, 如果找到最后一个 `mmapdsc_t` 结构也没有发现空闲的内存块, 就宣告内存分配失败。若在 `mini2440` 上这种算法是最好的, 因为 `mini2440` 上最多只有 16 个 `mmapdsc_t` 结构 (64MB 内存, 4MB 大小的内存块)。可是我们的 `LMOSEM` 内核将来可能运行在其他计算平台上, 其上可能有上

10GB 内存，那么 `mmapdsc_t` 数组可能非常大，这种算法可能非常低效。我们需要一种折中的方法。

为了加快对多 `mmapdsc_t` 结构的搜寻速度，最直接的方法就是对众多 `mmapdsc_t` 结构进行分类，首先把表示不同大小内存块的 `mmapdsc_t` 结构分为不同的类，然后把同类大小内存块的 `mmapdsc_t` 结构，再次分类为全部未分配的（空闲的）、已经分配一部分的、已经分配完的，所以一种数据结构可能不行，还得多设计几个其他数据结构，如代码清单 6-39[xxxx/lmosem/include/halinc/halmm_t.h] 所示。

代码清单6-39 `alcfrelst_t`、`phymem_t`数据结构

```
typedef struct s_ALCFRELST
{
    spinlock_t afl_lock;        //保护其自身的自旋锁
    size_t     afl_sz;          //表示该类的大小
    list_h_t   afl_fulllsth;     //同类已经分配完的mmapdsc_t结构链表头
    list_h_t   afl_emtplsth;     //同类全部未分配的（空闲的）mmapdsc_t结构链表头
    list_h_t   afl_fuemplsth;    //同类已经分配一部分的mmapdsc_t结构链表头
    /*用于挂载同一种内存块大小的mmapdsc_t结构队列，如分割成128KB大小的
    *并对其进行分类，分配完的、分配一部分的、未分配的*/
}alcfrelst_t;

typedef struct s_PHYMEM
{
    list_h_t pmm_list;
    spinlock_t pmm_lock;
    uint_t freeblks;
    uint_t allcblks;
    alcfrelst_t pmm_sz_lsth[BLKSZ_HEAD_MAX];
    /*形成了128KB~4MB大小的alcfrelst_t队列数组 [0]128KB
    * [1]256KB [2]512KB [3]1MB [4]2MB [5]4MB*/
}phymem_t;
```

代码清单 6-39 中设计了两个数据结构 `alcfrelst_t`、`phymem_t`。可以看到，`alcfrelst_t` 数据结构只是一个容器，它就像一个箱子，只不过 `alcfrelst_t` 这个箱子中装的是 `mmapdsc_t` 结构。而 `phymem_t` 数据结构看上去像是一个更大的箱子，它里面装下了 128KB ~ 4MB 这几种分类大小的所有 `alcfrelst_t` 数据结构。

关于 `phymem_t` 数据结构定义的实例变量，和前面的 `osmach` 变量一样，是全局性的并且其占用空间分配在 `LMOSEM` 内核的可执行文件中，如代码清单 6-40[xxxx/lmosem/hal/halglobal.c] 所示。

代码清单6-40 定义全局的`phymem_t`类型的变量

```
.....
HAL_DEFGLOB_VARIABLE(phymem_t,osphymem);
.....
```

从上面几个数据结构的设计不难看出，最小分配的内存块也有 128KB，即使只需要 1

字节的空间也要分配 128KB 大小。这是不是太浪费了？不过这里我们暂且就这样，因为 LMOSEM 内核的内存管理组件是层次结构的，这才是第一层，所以不怕，其他上层中会有解决办法。好了，关于这一层次的相关数据结构就设计好了，下面去写代码初始化这些数据结构。

6.5.2 初始化 mmapdsc_t 结构数组

前面不止一次提到过 mmapdsc_t 结构在内存中是一个数组。这个数组是 LMOSEM 内核启动时根据内存大小按需要动态建立的，现在我们就去建立这个数组。

建立 mmapdsc_t 结构数组之前，需要有一些信息，如物理内存空间从哪里开始，哪里结束，有几段这样的地址空间。因为物理内存地址空间可能不是连续的，还有一些地址空间映射的是设备的寄存器，而不是 SDRAM。对于 mini2440 上的 SDRAM，它映射在一个 64MB 大小的连续地址空间中，其他地址空间有的是空洞，有的是映射的设备寄存器。为了以后能灵活地移植 LMOSEM 内核，我们不把代码写死了，所以需要额外建立一个地址空间数组，这个数组的类型也是一个数据结构，如代码清单 6-41[xxxx/lmosem/include/halinc/halmm_t.h] 所示。

代码清单6-41 地址空间数据结构类型及相关宏

```
#define DEV_TYPE_NOT 0xffffffff
#define ADRSPCE_NOT 0
#define ADRSPCE_IO 1
#define ADRSPCE_SDRAM 2
#define ADRSPCE_RAM 3
#define ADRSPCE_ROM 4
#define ADRSPCE_NORFLASH 5
#define ADRSPCE_NANDFLASH 6
typedef struct s_PHYADRSPCE
{
    u32_t ap_flg;    //地址空间的类型如ADRSPCE_IO、ADRSPCE_SDRAM等
    u32_t ap_devtype; //如果地址空间的类型是ADRSPCE_IO，则指示其设备类型。这里不必关心，这是为后面扩展所用的
    adr_t ap_adrstart; //地址空间的开始地址
    adr_t ap_adrend;   //地址空间的结束地址
}phyadrspace_t;
```

这个类型的数组定义在哪里呢，同样是和 osmach 一样，如代码清单 6-42[xxxx/lmosem/hal/halglobal.c] 所示。不过这个数组在定义时就被初始化了，ap_devtype 域全部初始化为 0，其中的地址部分请参阅 S3C2440A 手册，数组最后一个元素仅是个标记，表示数组到此就结束了，其实这里我们只需要关注 {ADRSPCE_SDRAM,0,0x30000000,0x33ffff}，这一个数组元素就行了。

代码清单6-42 定义全局的地址空间数组

```

HAL_DEFGLOB_VARIABLE(phyadrspce_t, machadrspce) [PLFM_ADRSPCE_NR] = {
{ADRSPCE_NORFLASH, 0, 0, 0x001fffff},
.....
{ADRSPCE_IO, 0, 0x29000000, 0x29000007},
{ADRSPCE_SDRAM, 0, 0x30000000, 0x33ffffff},
.....
{ADRSPCE_IO, 0, 0x50000000, 0x50008028},
{ADRSPCE_IO, 0, 0x51000000, 0x51000040},
{ADRSPCE_IO, 0, 0x52000000, 0x5200026f},
{ADRSPCE_IO, 0, 0x53000000, 0x53000008},
.....
{ADRSPCE_NOT, DEV_TYPE_NOT, 0, 0}};

```

还记得前面的 `osmach` 结构变量的初始化代码吗，在那里我们要把这个数组的首地址和元素个数写到 `osmach` 结构的相关域中，如代码清单 6-43[xxxx/lmosem/hal/halmach.c] 所示。

代码清单6-43 把地址空间数组的首地址、个数写到osmach结构中

```

void mach_t_init(mach_t* initp)
{
    .....
    initp->mh_mmapdscadr = (mmapdsc_t*) (ALIGN(((uint_t) (&__end_kernel)), 4096));
    initp->mh_mmapdscnr = 0;
    initp->mh_spedschnr = PLFM_ADRSPCE_NR;
    initp->mh_spedschnr = machadrspce;
    .....
    return;
}

```

读者可能感觉这不是自找麻烦吗，直接用 `machadrspce` 不就行了吗，还要回去写 `osmach` 的相关域？这是为了后面的代码简单一点。

有了地址空间数据结构，就有了相关的地址空间信息，也就可以写初始化 `mmapdsc_t` 结构数组的代码了，先在 `halmm.c` 文件中写一个函数，函数有一个 `mach_t*` 的参数。下面就会看到，这个函数正是通过 `osmach` 中相关的数据，获取 `mmapdsc_t` 结构数组的首地址和 `machadrspce` 结构数组的首地址，如代码清单 6-44[xxxx/lmosem/hal/halmm.c] 所示。

代码清单6-44 初始化mmapdsc_t结构数组

```

void init_mmapdsc(mach_t* mahp)
{
    phyadrspce_t* adsp = mahp->mh_spedschnr;
    uint_t spnr = mahp->mh_spedschnr;
    uint_t mindx = 0;
    for(uint_t i = 0; i < spnr; i++)
    {
        if(adsp[i].ap_flg == ADRSPCE_SDRAM)
        {
            mindx = init_core_mmapdsc(adsp[i].ap_adrstart, adsp[i].ap_adrend, mahp->mh_

```

```

        mmapdscadr, mindx);
    }
}
mahp->mh_mmapdscnr=mindx;
mahp->mh_kerinramend=(adr_t)((uint_t)mahp->mh_mmapdscadr+((sizeof(mmapdsc_
t))*mindx));
return;
}

```

代码清单 6-44 主要是由一个 for 循环驱动的，它主要是扫描所有具备 ADRSPCE_SDRAM 类型的物理地址空间数组的元素，找到一个就调用 init_core_mmapdsc 函数。循环结束后，把 mindx 的值写入 osmach 结构中的 mh_mmapdscnr 域中，并重新调整了 LMOSEM 内核在内存中的结束地址，因为它要包含 mmapdsc_t 结构数组的那段空间。

接着来写 init_core_mmapdsc 函数，如代码清单 6-45[xxxx/lmosem/hal/halmm.c] 所示。它主要是根据一段物理内存地址空间，初始化一些 mmapdsc_t 结构数组中的元素，究竟有多少个元素是根据物理内存地址空间的大小来决定的，这个函数的参数也是地址空间的开始地址和结束地址，还有 mmapdsc_t 结构数组的首地址和当前 mmapdsc_t 结构数组元素的计数，首次执行时计数变量当然是 0。返回值是下一次将要初始化其数组元素的计数。

代码清单 6-45 初始化 mmapdsc_t 结构数组的核心函数

```

uint_t init_core_mmapdsc(adr_t adrs, adr_t adre, mmapdsc_t* gmmp, uint_t curmindx)
{
    uint_t mindx=curmindx;
    adr_t tmpadr=adre;
    for(; adrs<adre; adrs+=MAPONE_SIZE, mindx++)
    {
        if((adrs+MAPONE_SIZE)<adre)
        {
            tmpadr=(adrs+MAPONE_SIZE)-1;
        }else
        {
            tmpadr=adre;
        }
        mmapdsc_t_init(&gmmp[mindx], adrs, tmpadr, 0, MAP_FLAGS_VAL(0, MAPF_ACSZ_4MB, MAPF_
SZ_4MB));
    }
    return mindx;
}

```

这个函数的内部，同样是由一个循环驱动的，MAPONE_SIZE 的定义 #define MAPONE_SIZE (0x400000)，循环中前面的检查，是为了防止内存块结束地址超过地址空间的结束地址。每循环一次就调用 mmapdsc_t_init 函数初始化一个 mmapdsc_t 结构。mmapdsc_t_init 函数的代码，如代码清单 6-46[xxxx/lmosem/hal/halmm.c] 所示。MAP_FLAGS_VAL 也是一个宏：#define MAP_FLAGS_VAL(RV, SALLOCSZ, MSZ) (RV|SALLOCSZ|MSZ)，它是用于合

成 `mmapdsc_t` 结构中的 `map_flg` 域的。这里我们看到，初始化时，一个 `mmapdsc_t` 结构表示 4MB 大小的内存块，而当进行一次内存分配时的内存块也是 4MB，即 `MAP_FLAGS_VAL(0,MAPF_ACSZ_4MB,MAPF_SZ_4MB)`。

代码清单6-46 初始化一个 `mmapdsc_t` 结构

```
void mmapdsc_t_init(mmapdsc_t* mmp, adr_t phyadr, adr_t phyadre, u32_t allcount, u32_t flgs)
{
    list_init(&mmp->map_list);
    hal_spinlock_init(&mmp->map_lock);
    mmp->map_phyadr=phyadr; // 内存块的开始地址
    mmp->map_phyadrend=phyadre; // 内存块的结束地址
    mmp->map_allcount=allcount; // 分配计数，初始化时为0
    mmp->map_flg=flgs; // 相关的标志
    return;
}
```

代码清单 6-46 中的注释已经很明显了，不用多加说明。

有了 `mmapdsc_t` 结构数组还不够，前面已经说明了，我们不能去遍历这个 `mmapdsc_t` 结构数组，所以我们设计了一些容器来对 `mmapdsc_t` 结构进行归类，下面就去把它们都归类，便于以后分配内存时查找它们。

6.5.3 建立起内存分配数据结构

有了 `mmapdsc_t` 结构数组，似乎就可以分配内存了。但是为了灵活和以后便于提高性能，不能直接通过扫描 `mmapdsc_t` 结构数组获取空闲的内存块，因为当 `mmapdsc_t` 结构数组非常大时，这种方法就不行了。由前面章节可知，我们设计了一些容器来对 `mmapdsc_t` 结构进行归类，以提高搜索性能。可是这些容器（数据结构）还没初始化，这里我们就来初始化这些容器（数据结构）。

可以发现，全部容器（`alcfrlst_t`）都是放在 `phymem_t` 数据结构中的，首先在 `halmm.c` 文件中来写一个 `init_phymem` 函数，如代码清单 6-47[xxxx/lmosem/hal/halmm.c] 所示。该函数的代码首先是初始化 `osphymem`（即 `phymem_t` 类型的结构变量）本身，下一步才把 `mmapdsc_t` 结构放进那些容器中。

代码清单6-47 初始化物理内存管理

```
void init_phymem()
{
    phymem_t_init(&osphymem);
    pmmlist_init(&osmach, &osphymem);
    return;
}
```

接着写几个函数初始化 `osphymem` 结构变量，前面已经定义了 `osphymem` 结构变量，这里直接写代码，如代码清单 6-48[xxxx/lmosem/hal/halmm.c] 所示。

代码清单6-48 初始化osphymem结构变量

```

void alcfrlst_t_init(alcfrlst_t* initp, size_t bsiz)
{
    hal_spinlock_init(&initp->afl_lock);
    initp->afl_sz=bsiz;
    list_init(&initp->afl_fulllst);
    list_init(&initp->afl_emtplst);
    list_init(&initp->afl_fuemlst);
    return;
}

void phymem_t_init(phymem_t* initp)
{
    list_init(&initp->pmm_list);
    hal_spinlock_init(&initp->pmm_lock);
    alcfrlst_t_init(&initp->pmm_sz_lsth[0], BLK128KB_SIZE);
    alcfrlst_t_init(&initp->pmm_sz_lsth[1], BLK256KB_SIZE);
    alcfrlst_t_init(&initp->pmm_sz_lsth[2], BLK512KB_SIZE);
    alcfrlst_t_init(&initp->pmm_sz_lsth[3], BLK1MB_SIZE);
    alcfrlst_t_init(&initp->pmm_sz_lsth[4], BLK2MB_SIZE);
    alcfrlst_t_init(&initp->pmm_sz_lsth[5], BLK4MB_SIZE);
    return;
}

```

代码清单 6-48 非常简单，phymem_t_init 函数中分别调用 alcfrlst_t_init 函数初始化了 6 个 alcfrlst_t 结构（容器）。这 6 个 alcfrlst_t 结构只是表示装 mmapdsc_t 结构的大小不同而已（因为 LMOSEM 运行后，mmapdsc_t 结构所对应的内存块会分成大小不同的小块），其他部分都一样，链表一定要初始化后才能使用，参阅前面章节中的 list_h_t 数据结构。

接下来，就是把 mmapdsc_t 结构按要求分别装入那些 alcfrlst_t 结构（容器）中。怎么装呢，如代码清单 6-49[xxxx/lmosem/hal/halmm.c] 所示。这个过程不是太难，第一步是找到一个 mmapdsc_t 结构，然后调用 mapdsc_addto_memlst 函数始终把它装入 4MB 大小的 alcfrlst_t 结构（容器）中，因为初始化时，所有 mmapdsc_t 结构都被初始化成 4MB 大小的内存块，直到找完所有的 mmapdsc_t 结构，list_add_tail 函数是在 list.h 文件中定义的。这里再次用到了 osmach 结构变量。

代码清单6-49 将mmapdsc_t结构加入到相应的alcfrlst_t结构中

```

void mapdsc_addto_memlst(alcfrlst_t* aflp, mmapdsc_t* mapp, uint_t atflg)
{
    switch(atflg)
    {
        case ADDT_EMTP_FLG:
            list_add_tail(&mapp->map_list, &aflp->afl_emtplst);
            break;
        case ADDT_FUEM_FLG:
            list_add_tail(&mapp->map_list, &aflp->afl_fuemlst);
            break;
        case ADDT_FULL_FLG:

```

```

        list_add_tail(&mapp->map_list,&aflp->afl_fulllsth);
        break;
    default:
        break;
    }
    return;
}

void pmmllist_init(mach_t* mahp,phymem_t* pmp)
{
    mmapdsc_t* mapp=mahp->mh_mmapdscadr;
    uint_t mapnr=mahp->mh_mmapdscnr;
    for(uint_t mr=1;mr<mapnr;mr++)
    {
        mapdsc_addto_memlst(&pmp->pmm_sz_lsth[BLKSZ_HEAD_MAX-1],&mapp[mr],ADDT_
            EMTP_FLG);
    }
    return;
}

```

还有最后一个问题，我们初始化 `mmapdsc_t` 结构数组时，都是假定内存空间是完全空闲的，但是事实不是这样的，因为 `LMOSEM` 内核包括 `mmapdsc_t` 结构数组，本身就存在于内存中，所以要把 `LMOSEM` 内核所占用的内存对应的那个 `mmapdsc_t` 结构拿出来处理一下，以免内存分配时，分配到 `LMOSEM` 内核占用的那块内存，那可就是灾难性的错误。在 `mini2440` 上可以做如下假定：`LMOSEM` 内核占用第一个 `4MB` 内存块的前 `128KB`，这对 `LMOSEM` 内核在 `mini2440` 上是可行的，所以我们草草地写下一个 `onmmmapdsc_inkrlram` 函数，如代码清单 6-50[xxxx/lmosem/hal/halmm.c] 所示。这个函数有严重的局限性，请读者在测试时要格外注意。

代码清单6-50 处理LMOSEM内核和相关数据占用的内存

```

void onmmmapdsc_inkrlram(mach_t* mahp,phymem_t* pmp)
{
    mmapdsc_t* mapp=mahp->mh_mmapdscadr;
    adr_t ker_r_s=mahp->mh_kerinramstart,ker_r_e=mahp->mh_kerinramend;
    if((ker_r_e-ker_r_s)>BLK128KB_SIZE||ker_r_s<mapp[0].map_phyadr)
    {
        hal_sysdie("onmmmapdsc_inkrlram err");//死机
    }
    u32_t cut=1;
    u32_t flg=MAPF_FLAGS_VAL(0,MAPF_ACSZ_128KB,MAPF_SZ_4MB);
    mapp[0].map_allcount=cut;
    mapp[0].map_flg=flg;
    mapdsc_addto_memlst(&pmp->pmm_sz_lsth[0],&mapp[0],ADDT_FUEM_FLG);
    return;
}

```

尽管这个 `onmmmapdsc_inkrlram` 函数非常草率，但是我们还是作了严格的审查，如果 `LMOSEM` 内核的大小大于 `128KB`，或者 `LMOSEM` 内核不在第一个 `4MB` 大小的内存块中，

就调用 `hal_sysdie` 函数输出: "onmmappedsc_inkrlram err", 然后死机, 这个函数是在 `cpuctrl.c` 文件中定义的。接下来, 就是改变标志: 指定第一个 `mmapdsc_t` 结构对应的是 4MB 大小的内存块, 但其内部分成了 128KB 大小的小块, 修改 `mapp[0].map_allcount` 为 1 指定其内部的第一个 128KB 小块已经分配, 最后调用 `mapdsc_addto_memlst` 函数把第一个 `mmapdsc_t` 结构挂入 128KB 大小的 `alcfrlst_t` 结构中的 `afl_fuemlsth` 链表中。这也是 `pmmlist_init` 函数中的循环变量 `mr` 从 1 开始的原因, 因为第一个 (数组下标为 0) `mmapdsc_t` 结构是 LMOSEM 内核所占用的。

最后把本小节的初始化函数归纳进 `init_halmm` 函数中, 并把 `init_halmm` 函数加入 LMOSEM 内核 `hal` 层的 `init_hal` 函数中, 如代码清单 6-51 [xxxx/lmosem/hal/halmm.c] 所示。

代码清单6-51 物理内存管理的顶层初始化函数

```
void init_halmm()
{
    .....
    init_mmapdsc(&osmach);
    init_phymem();
    //下面这个函数要格外注意
    onmmappedsc_inkrlram(&osmach, &osphymem);
    return;
}
```

如果读者有兴趣, 可以在 `init_halmm` 函数的最后写一个小的循环语句, 在其中加入 `printfk` 函数输出一些 `mmapdsc_t` 结构中的地址值, 验证一下是不是预期的那样, 必然每一步都进行验证, 才能写出稳定的代码。由于篇幅所限, 笔者就只能走到这里了。

6.6 中断初始化

前面章节中, 复制了中断向量, 这是 CPU 所需要的。CPU 在允许响应中断的情况下, 一旦中断信号到来, 它就去执行相关的中断向量, 至此就再也没有它的事了。它接着执行中断向量处的代码, 而 LMOSEM 内核或者设备驱动程序需要确定中断源, 即哪个设备产生了中断, 接着进行相关的中断处理。因为不止一个中断源, 所以需要设计一些数据结构来管理众多的中断源。下面就去设计并初始化这些数据结构。

6.6.1 设计一些数据结构

前面章节中对 S3C2440A 的中断控制器已经有所了解, 它最多能接收 60 个中断源, 而且这 60 个中断源是按层次关系组织的, 不是并列存在的。它们分为主中断源、次级中断源、外部中断源 (也可认为是次级中断源)。图 2-7 很好地展示了这种关系。为了更好地反映它们之间的对应关系, 来看一个表, 如表 6-8 所示。

表6-8 S3C2440A的中断源

主 中 断 源	主中断源挂起寄存器 (SRCPND_R 0x4A000000) 中的相关位	描 述		
EINT0	0	外部中断 0		
EINT1	1	外部中断 1		
EINT2	2	外部中断 2		
EINT3	3	外部中断 3		
EINT4_7	4	外部中断 4 ~ 7, 该主中断源上包含如下 4 个外部中断:		
		外部中断源	外部中断源挂起寄存器 (EINTPEND_R 0x560000A8) 中的相关位	描述
		保留	[0:3]	
		EINT4	4	外部中断 4
		EINT5	5	外部中断 5
		EINT6	6	外部中断 6
		EINT7	7	外部中断 7
EINT8_23	5	外部中断 8 ~ 23, 该主中断源上包含如下 16 个外部中断:		
		外部中断源	外部中断源挂起寄存器 (EINTPEND_R 0x560000A8) 中的相关位	描述
		EINT8	8	外部中断 8
		EINT9	9	外部中断 9
		EINT10	10	外部中断 10
		EINT11	11	外部中断 11
		EINT12	12	外部中断 12
		EINT13	13	外部中断 13
		EINT14	14	外部中断 14
		EINT15	15	外部中断 15
		EINT16	16	外部中断 16
		EINT17	17	外部中断 17
		EINT18	18	外部中断 18
		EINT19	19	外部中断 19
		EINT20	20	外部中断 20

(续)

主中断源	主中断源挂起寄存器 (SRCPND_R 0x4A000000) 中的相关位	描 述		
EINT8_23	5	EINT21	21	外部中断 21
		EINT22	22	外部中断 22
		EINT23	23	外部中断 23
INT_CAM	6	摄像头接口中断, 该主中断源上包含如下 2 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_CAM_C	11	摄像头接口中 C 端口捕获中断
		INT_CAM_P	12	摄像头接口中 P 端口捕获中断
nBATT_FLT	7	电池故障中断		
INT_TICK	8	RTC 时钟滴答中断		
INT_WDT_AC97	9	看门狗定时器中断, 该主中断源上包含如下 2 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_WDT	13	看门狗中断
		INT_AC97	14	AC97 中断
		保留	[15:31]	
INT_TIMER0	10	定时器 0 中断		
INT_TIMER1	11	定时器 1 中断		
INT_TIMER2	12	定时器 2 中断		
INT_TIMER3	13	定时器 3 中断		
INT_TIMER4	14	定时器 4 中断		
INT_UART2	15	串口设备 2 的中断, 该主中断源上包含如下 3 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_RXD2	6	串口设备 2 的数据接收中断
		INT_TXD2	7	串口设备 2 的数据发送中断

(续)

主 中 断 源	主中断源挂起寄存器 (SRCPND_R 0x4A000000) 中的相关位	描 述		
INT_UART2	15	INT_ERR2	8	串口设备 2 的错误中断
INT_LCD	16	LCD 中断		
INT_DMA0	17	DMA 通道 0 中断		
INT_DMA1	18	DMA 通道 1 中断		
INT_DMA2	19	DMA 通道 2 中断		
INT_DMA3	20	DMA 通道 3 中断		
INT_SDI	21	SDI 中断		
INT_SPI0	22	SPI0 中断		
INT_UART1	23	串口设备 1 的中断, 该主中断源上包含如下 3 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_RXD1	3	串口设备 1 的数据接收中断
		INT_TXD1	4	串口设备 1 的数据发送中断
		INT_ERR1	5	串口设备 1 的错误中断
INT_NFCON	24	Nand Flash 控制中断		
INT_USBD	25	USB 设备中断		
INT_USBH	26	USB 主机中断		
INT_IIC	27	IIC 中断		
INT_UART0	28	串口设备 0 的中断, 该主中断源上包含如下 3 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_RXD0	0	串口设备 0 的数据接收中断
		INT_TXD0	1	串口设备 0 的数据发送中断
		INT_ERR0	2	串口设备 0 的错误中断
INT_SPI1	29	SPI1 中断		
INT_RTC	30	RTC 闹钟中断		

(续)

主中断源	主中断源挂起寄存器 (SRCPND_R 0x4A000000) 中的相关位	描 述		
INT_ADC	31	ADC EOC 和触屏中断, 该主中断源上包含如下 2 个次级中断:		
		次级中断源	次级中断源挂起寄存器 (SUBSRCPND_R 0x4A000018) 中的相关位	描述
		INT_TC	9	触摸屏中断 (笔起 / 落)
		INT_ADC_S	10	ADC 中断

表 6-8 展示了外部中断源、次级中断源和主中断源的关系。而当一个设备的中断发生了, S3C2440A 的中断控制器所能提供的信息只是哪个主中断源上来了中断, 例如, 28 号主中断源上来了中断, 还需要进一步确定到底是哪一个次级中断源上 (INT_RXD0、INT_TXD0、INT_ERR0) 来了中断。这给 LMOSEM 内核的中断处理程序带来了麻烦。

下面开始设计相关的数据结构。为什么要设计这些数据结构, 是因为中断通常是由设备产生的, 而中断处理程序, 则分为通用部分和设备驱动程序的回调函数部分, 也就是说一个设备发生了中断, LMOSEM 内核最终会调用具体设备驱动程序的中断处理代码来处理具体的中断。如代码清单 6-52 [xxxx/lmosem/include/halinc/halintupt_t.h] 所示, 我们就用这样一个数据结构来表示一个中断源。

代码清单6-52 中断源数据结构

```
typedef struct s_INTFLTDSC
{
    spinlock_t    i_lock;           //保护其自身的自旋锁
    list_h_t      i_serlist;        //设备中断服务程序链
    uint_t        i_sernr;          //设备中断服务程序链个数
    u32_t         i_flg;            //相关标志位
    u32_t         i_stus;           //相关状态位
    u32_t         i_pndbitnr;       //中断源挂起寄存器中的位序
    uint_t        i_irqnr;          //中断号
    uint_t        i_deep;           //保留
    uint_t        i_indx;           //产生中断的次数
}intfltdsc_t;
```

代码清单 6-52 中的注释已经清楚地说明了一切, 但是它只是表示一个中断源, 而上面提到的设备驱动程序的回调函数, 则是在另外的数据结构之中, 然后将其挂载到设备中断服务程序链表 (i_serlist) 中。为什么要这样, 是因为很多计算平台中许多设备会共用同一个中断源。当然, 这种情况在 mini2440 上是不存在的, 但是设计 LMOSEM 内核, 不能只为 mini2440 平台而设计, 也许有天 LMOSEM 内核会移植到更为强大的计算平台之上。

有了数据结构类型，就可以去定义实例变量了，如代码清单 6-53[xxxx/lmosem/hal/halglobal.c] 所示。

代码清单6-53 定义全局的intfltdsc_t结构类型的数组

```
.....
HAL_DEFGLOB_VARIABLE(intfltdsc_t,machintflt)[IRQ_MAX];
.....
```

又是一个全局性的 intfltdsc_t 结构数组，因为有多个中断源，自然用数组表示最方便直观，并且这个数组包含在 LMOSEM 内核的可执行文件中。IRQ_MAX 定义为 71。为什么是 71，不是只有 60 个中断源吗？因为我们为了直接用数组下标寻找 intfltdsc_t 结构，采用了平级映射的方法，例如，28 号主中断源发生了中断，发现它其中包含有 3 个次级中断源，就用 28 加上一个次级偏移量：SINT_OFFSET (32)，找到相应的 intfltdsc_t 结构。再如，5 号主中断源发生了中断，发现它其中包含有 16 个外部中断源，就用 5 加上一个外部次级偏移量 EINT_OFFSET (47)，找到相应的 intfltdsc_t 结构。不难发现，数组下标 0 ~ 31 对应的是主中断源，数组下标 32 ~ 46 对应的是次级中断源，数组下标 47 ~ 70 对应的是外部中断源。

最后应该把 intfltdsc_t 结构数组的首地址和数组元素的个数，归纳到 osmach 结构相关的域中，如代码清单 6-54[xxxx/lmosem/hal/halmach.c] 所示。

代码清单6-54 把中断源数组的首地址、个数写入osmach中

```
void mach_t_init(mach_t* initp)
{
    .....
    initp->mh_intfltdsc=machintflt;
    initp->mh_intfltnr=IRQ_MAX;
    return;
}
```

了解了 S3C2440A 所有的中断源，intfltdsc_t 数据结构已经设计好了，也定义好了该数据结构类型的数组变量，可是还没有对其初始化呢，现在就去初始化 intfltdsc_t 结构数组。

6.6.2 初始中断源描述符

在操作系统内核中任何数据结构的实例，在使用之前都要先对其初始化，否则数据结构实例中的数据就是不确定的，6.6.1 节中只是设计并定义了 intfltdsc_t 结构数组，还没有对其初始化。这里只是初始化 intfltdsc_t 结构数组，不做管理中断相关的事情。

下面来写初始化 intfltdsc_t 结构数组的代码，在 LMOSEM 内核工程目录下的 hal 目录下新建一个 C 语言程序源文件 halintupt.c，在其中写下几个函数，如代码清单 6-55[xxxx/lmosem/hal/halintupt.c] 所示。

代码清单6-55 初始化intfltdsc_t结构数组

```

void init_halintupt()
{
    init_intfltdsc(&osmach);
    return;
}

void init_intfltdsc(mach_t* machp)
{
    intfltdsc_t* ifdp=machp->mh_intfltdsc;
    uint_t ifdnr=machp->mh_intfltnr;
    .....
    for(uint_t i=MINTNR_START; i<=MINTNR_END;i++)
    {
        intfltdsc_t_init(&ifdp[i],MINT_FLG,0,i-MINT_OFFSET,i);
    }
    for(uint_t i=SINTNR_START; i<=SINTNR_END;i++)
    {
        intfltdsc_t_init(&ifdp[i],SINT_FLG,0,i-SINT_OFFSET,i);
    }
    for(uint_t i=EINTNR_START; i<=EINTNR_END;i++)
    {
        intfltdsc_t_init(&ifdp[i],EINT_FLG,0,i-EINT_OFFSET,i);
    }
    return;
}

void intfltdsc_t_init(intfltdsc_t* initp,u32_t flg,u32_t stus,u32_t pbitnr,uint_t irqnr)
{
    hal_spinlock_init(&initp->i_lock);
    list_init(&initp->i_serlist);
    initp->i_sernr=0;
    initp->i_flg=flg;
    initp->i_stus=stus;
    initp->i_pndbitnr=pbitnr;
    initp->i_irqnr=irqnr;
    initp->i_deep=0;
    initp->i_indx=0;
    return;
}

```

代码清单 6-55 中写了一些宏，它们定义在 halintupt_t.h 文件中，如代码清单 6-56[xxxxx/Imosem/include/halinc/halintupt_t.h] 所示。

代码清单6-56 intfltdsc_t结构相关的宏

```

.....
#define MINT_OFFSET 0
#define SINT_OFFSET 32
#define EINT_OFFSET 47
#define IRQ_MAX 71
#define MINTNR_START (0+MINT_OFFSET)

```

```

#define MINTNR_END (31+MINT_OFFSET)
#define SINTNR_START (0+SINT_OFFSET)
#define SINTNR_END (14+SINT_OFFSET)
#define EINTNR_START (0+EINT_OFFSET)
#define EINTNR_END (23+EINT_OFFSET)
#define MINT_FLG 1
#define SINT_FLG 2
#define EINT_FLG 3
.....

```

首先代码还是从 `osmach` 结构中获取 `intfltdsc_t` 结构数组的首地址和其数组元素的个数，接着是三个循环语句驱动的，分别初始化主中断源描述符、次级中断源描述符、外部中断源描述符，在循环中分别调用 `intfltdsc_t_init` 函数初始化具体的 `intfltdsc_t` 结构数组中的元素，`intfltdsc_t` 结构中的 `i_flg` 域表示了 `intfltdsc_t` 结构是什么类型的。代码非常直观，不用过多说明。只是最后别忘了把 `init_halintupt` 函数加入到 `LMOSEM` 内核的 `hal` 层的主初始化函数 `init_hal` 中。

作为一个用于学习、探索其原理的操作系统内核，`LMOSEM` 内核的硬件相关层的初始化工作，就到此为止了。下面我们需要边休息边测试我们的代码是否正确。

6.7 初始化测试

一步步走来，写了不少代码，好像从来没有质疑过代码的质量，代码在运行的过程中是否像我们预期的那样。这些代码都是最基础的代码，因此它们需要绝对正确，证明代码是否正确，测试一下就知道了。

下面我们来测试一些数据结构。

调试过应用软件的读者可能知道，有各种调试软件，它们能对一款应用软件进行单步调试，它们可以查看内存中的内容，也可以查看 CPU 寄存器中的值，但对于调试操作系统内核就没那么好运了，因为调试软件本身就需要操作系统内核的支持才行。可想而知，对操作系统内核的调试只能我们自己想办法。

前面只是初始化了一些重要的数据结构，只需要把它们其中的数据输出，看看是不是我们预期的就行了，因为有些数据结构的变量实例是定义在 `LMOSEM` 内核二进制文件中的数组，它们不太容易出错，这里主要输出两个数据结构中的数据 `mach_t` 结构和 `mmapdsc_t` 结构数组。

先来输出 `mach_t` 数据结构实例中的数据，在 `halmach.c` 文件中写一个 `print_mach` 函数来完成相应数据的输出，记住这个函数要放在 `init_hal` 函数的最后调用，如代码清单 6-57 所示。

代码清单6-57 输出mach_t数据结构实例中的数据

```

void print_mach(mach_t* mchp)
{
    printfk("mach.mh_kerinramstart:%x\n\r", (uint_t)mchp->mh_kerinramstart);
}

```

```

printfk("mach.mh_keriramend:%x\n\r", (uint_t)mchp->mh_keriramend);
printfk("mach.mh_mmapdscadr:%x\n\r", (uint_t)mchp->mh_mmapdscadr);
printfk("mach.mh_mmapdscnr:%x\n\r", (uint_t)mchp->mh_mmapdscnr);
printfk("mach.mh_intfltdsc:%x\n\r", (uint_t)mchp->mh_intfltdsc);
printfk("mach.mh_intfltnr:%x\n\r", (uint_t)mchp->mh_intfltnr);
return;
}
LKHEAD_T void init_hal()
{
    .....
    print_mach(&osmach);
    for(;;);
    return;
}

```

按照前面章节的方法编译好 LMOSEM 内核，连接好开发板，将 LMOSEM 内核下载到开发板中运行，正常情况下应该会得到和笔者一样的效果，如图 6-8 所示。

```

深度终端
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1138-2K
Enter your selection:

##### FriendlyARM BIOS 2.0 for 2440 #####
[x] format NAND FLASH for Linux
[v] Download vivi
[k] Download linux kernel
[y] Download root_yaffs image
[a] Absolute User Application
[n] Download Nboot for WinCE
[l] Download WinCE boot-logo
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1138-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30008000h, TOTAL:37202]
RECEIVED FILE SIZE: 37202 (36KB/S, 15)
Downloaded file at 0x30000000, size = 37192 bytes
mach.mh_keriramstart:30000000
mach.mh_keriramend:300121c0
mach.mh_mmapdscadr:30012000
mach.mh_mmapdscnr:10
mach.mh_intfltdsc:3000f73c
mach.mh_intfltnr:47

CTRL-A Z for help | 115200 BN1 | NOR | Minicom 2.7 | VT102 | 0

```

图 6-8 输出 mach_t 数据结构中的一些数据

不妨接着在 halmm.c 文件中写一个 print_mmapdsc 函数，用于输出每个 mmapdsc_t 结构数组元素所管理的内存块的地址值。同样，该函数也应该放在 init_hal 函数的最后调用，如代码清单 6-58 所示。

代码清单6-58 输出每个mmapdsc_t结构数组元素的相关信息

```

void print_mmapdsc(mach_t *mchp)
{
    mmapdsc_t* map=mchp->mh_mmapdscadr;
    uint_t mnr=mchp->mh_mmapdscnr;
    for(uint_t i=0;i<mnr;i++)
    {
        printfk("mmapdsc[%x].map_phyadr:%x,map_phyadrend:%x\n\r",i,(uint_t)map[i].
            map_phyadr,(uint_t)map[i].map_phyadrend);
    }
    return;
}

LKHEAD_T void init_hal()
{
    .....
    print_mmapdsc(&osmach);
    for(;;);
    return;
}

```

再次编译好 LMOSEM 内核并将其下载到开发板中运行。如果读者用的是 mini2440 开发板, 正常情况下也应该会得到和笔者一样的效果, 如图 6-9 所示。

```

深度终端
[k] Download linux kernel
[y] Download root_yaffs image
[a] Absolute User Application
[n] Download Nboot for WinCE
[l] Download WinCE boot-logo
[w] Download WinCE NK bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the hard flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1130-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30000000h,TOTAL:37322]
RECEIVED FILE SIZE: 37322 (36KB/S: 1S)
Downloaded file at 0x30000000, size = 37312 bytes
mmapdsc[0].map_phyadr:30000000,map_phyadrend:303fffff
mmapdsc[1].map_phyadr:30400000,map_phyadrend:307fffff
mmapdsc[2].map_phyadr:30800000,map_phyadrend:30bfffff
mmapdsc[3].map_phyadr:30c00000,map_phyadrend:30fffff
mmapdsc[4].map_phyadr:31000000,map_phyadrend:313fffff
mmapdsc[5].map_phyadr:31400000,map_phyadrend:317fffff
mmapdsc[6].map_phyadr:31800000,map_phyadrend:31bfffff
mmapdsc[7].map_phyadr:31c00000,map_phyadrend:31fffff
mmapdsc[8].map_phyadr:32000000,map_phyadrend:323fffff
mmapdsc[9].map_phyadr:32400000,map_phyadrend:327fffff
mmapdsc[a].map_phyadr:32800000,map_phyadrend:32bfffff
mmapdsc[b].map_phyadr:32c00000,map_phyadrend:32fffff
mmapdsc[c].map_phyadr:33000000,map_phyadrend:333fffff
mmapdsc[d].map_phyadr:33400000,map_phyadrend:337fffff
mmapdsc[e].map_phyadr:33800000,map_phyadrend:33bfffff
mmapdsc[f].map_phyadr:33c00000,map_phyadrend:33fffff

CTRL-A Z for help | 115200 BPS | NOR | Minicom 2.7 | VT102 | 0

```

图 6-9 输出每个 mmapdsc_t 数据结构中的一些数据

图 6-9 中的结果表示，我们前面的代码正确无误，前面几步走得非常稳健，后面可以放心地继续开发了。

6.8 小结

本章虽然没有让 LMOSEM 内核有任何实际的功能，但是起码让 mini2440 开发板在 LMOSEM 内核的控制下运行起来了。LMOSEM 内核为此初始化了 MMU，复制了中断向量表，初始化了串口设备，有了 printfk 函数可以输出信息了，初始化了内存管理的数据结构、中断源描述符，最后还确定了我们所做的一切都是正确无误的。

下面用一幅图来结束本章，如图 6-10 所示。

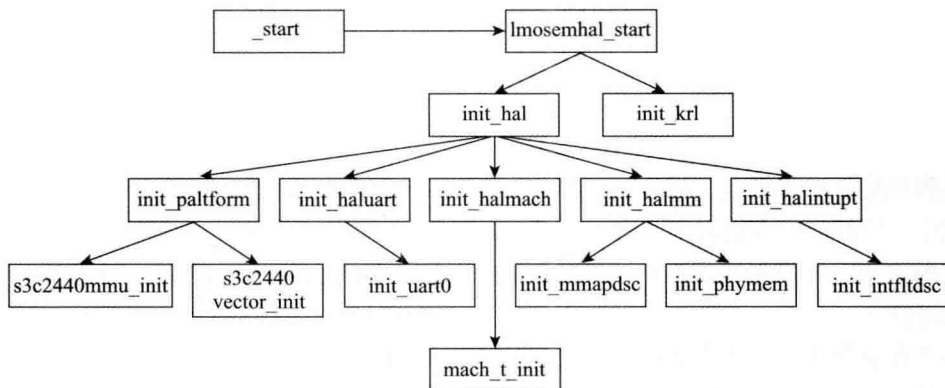


图 6-10 LMOSEM 内核初始化函数调用图

图 6-10 是本章我们编写的主要函数及调用图，当然没有画出所有函数，只画了一些主要的函数。这幅图从上到下，从左至右，完整地展示了 LMOSEM 内核代码的执行轨迹。

LMOSEM 内核最基本的初始化已经完成，接着即将进入下一段艰难而又富有挑战的开发旅程，同样也更加精彩。

内存管理

进程创建、中断管理、驱动程序加载、读写文件数据，都需要使用内存，所以在操作系统内核中，良好的内存管理是非常重要的。几乎 LMOSEM 内核中的其他组件都需要内存管理组件的支持，所以先实现 LMOSEM 内核中的内存管理组件，才能实现 LMOSEM 内核中的其他组件。

不要在意操作系统理论书籍上那些“先进”的内存管理算法，那些都是“学术派”干的事，甚至有许多内存管理算法，在现有商业操作系统中从未实现过。作为一个学习原理的操作系统内核，读者只要明白，内存管理就是能按照需求高效地分配、释放一块内存，这就是最终的目的。为此，本章将首先让 LMOSEM 内核的代码，从 LMOSEM 内核的硬件相关层运行到 LMOSEM 内核的功能层，在 LMOSEM 内核的功能层中从底向上实现：块级内存管理、页级内存管理、字级内存管理，最后让它们一起组成 LMOSEM 内核的内存管理组件。

下面就去一步一步地实现以上目标。

7.1 内核功能层入口

在第 6 章中，LMOSEM 内核虽然没有实现什么功能，但总算是让计算平台运行起来了，并且运行完了 `init_hal` 函数，其实到了这里，整个 LMOSEM 内核的初始化之路才刚刚走完了一半。在前面操作系统内核设计的相关章节中，发现 LMOSEM 内核分为了三个层：硬件相关层（`hal`）、内核功能层（`kernel`）、内核接口层。下面的 `init_krl` 函数就是其内核功能层的入口函数，也是其内核功能层的主初始化函数。那么在前面初始化的相关章节中，为什么不直接继续完成 LMOSEM 内核功能层的初始化呢，那是因为有些在 LMOSEM 内核功能层中组

件的初始化，本身需要内存管理组件的支持。下面就去写好 `init_krl` 函数。

前面提到的 `init_krl` 函数是 LMOSEM 内核功能层的主初始化函数。进入 LMOSEM 内核功能层的开发，首先就是要写好这个函数。

首先在 LMOSEM 工程目录下新建一个目录 `kernel`，表示我们的开发进入了 LMOSEM 内核功能层，在 `include` 目录下建立一个 `knlinc` 目录，用于存放 LMOSEM 内核功能层相关的头文件，还要另外建立两个头文件 `krltypes.h`、`krlheads.h`。为什么需要这样，前面章节中已经说明了。然后在 `kernel` 目录下建立一个文件 `krlinit.c`，同时在 `krlinc` 目录下建立两个头文件 `krlinit.h`、`krlinit_t.h`，分别加入 `krltypes.h`、`krlheads.h` 头文件中。相关原因，前面章节中已经说明了，不再详述。最后在 `krlinit.c` 文件中写一个 `init_krl` 函数，如代码清单 7-1[xxxx/lmosem/kernel/krlinit.c] 所示。

代码清单7-1 LMOSEM内核功能层初始化

```
#include "lmosemtypes.h"
#include "lmosemmctrl.h"
void init_krl()
{
    hal_sysdie("LMOSEM RUN!!");
    return;
}
```

在代码清单 7-1 中，`init_krl` 函数只是调用了个死机函数，这是为了暂时防止 `init_krl` 函数返回并告诉我们代码运行到哪里了，以免发生未知的错误。

最后还要让 `lmosemhal_start` 函数调用 `init_krl` 函数，不然代码的控制权不会传递到 `init_krl` 函数中来，如代码清单 7-2[xxxx/lmosem/hal/lmosemhal_start.c] 所示。

代码清单7-2 LMOSEM初始化

```
LKHEAD_T void lmosemhal_start()
{
    init_hal();
    init_krl();
    return;
}
```

如果现在就编译 LMOSEM 内核，LD 链接器一定会报错：无法找到 `init_krl` 符号，那是因为 `krlinit.c` 文件还没有参与编译。现在就将其加入 `buildfile.h` 编译头文件中，如代码清单 7-3[xxxx/lmosem/include/script/buildfile.h] 所示。

代码清单7-3 LMOSEM内核编译对象头文件

```
#ifndef BUILDFILE_H
#define BUILDFILE_H
#define BUILD_HALY_OBJS init.o lmosemhal_start.o interrupt.o\
.....
#define BUILD_KRNL_OBJS krlinit.o
```

```
.....  
#endif
```

因为 `krlnit.c` 文件已经不是 LMOSEM 内核硬件相关层的代码文件了, 所以不能再加入 `BUILD_HALY_OBJS` 宏的后面了, 而是要加入 `BUILD_KRNL_OBJS` 宏的后面了, 最后 `make` 一下, 就能自动编译链接这个文件了。

有了 `init_krl` 函数, 以后所有在 LMOSEM 内核功能层中的内核组件的初始化函数, 就放在 `init_krl` 函数中。有了它, 就可以进行下面的工作了。

7.2 内存管理组件的设计

如果单纯从设计的角度来思考, 关于内存管理组件设计的问题就多了, 如性能、分配算法、分配粒度等。只是这三个方面的问题已经让很多初学者不知所措了。所以暂且先把前两个方面的问题扔掉, 如果有可能, 希望把最后一个方面的问题也扔掉, 当然这是不行的。

通常分配内存空间时有一个重要的参数, 就是需要告诉内存管理组件, 这一次需要分配多大的内存空间。而分配粒度就是指内存管理组件一次最多、最少能分配多大的内存空间, 相应的释放内存空间时也是对应的, 开始分配多大的内存空间, 后面就应该释放多大的内存空间。如果这个问题不考虑好, 可能就会造成不必要的内存空间的浪费。当然还要考虑内存碎片的问题, 因为有时内存地址空间必须是要连续的等。下面就单从内存分配粒度的角度来简单地设计 LMOSEM 内核的内存管理组件。

内存管理组件的结构

内存分配请求, 可能是分配数兆字节的大内存空间, 也可能是分配几个字节的小内存空间, 更有可能要求该空间的物理地址是连续的。为什么要求该空间的物理地址是连续的呢, 因为一些计算平台上, 能访问内存的设备不只是 CPU, 如 DMA, 它们发出的访存地址不通过 MMU, 也就是说它们使用的是物理地址空间, 很有可能它们为了完成某些功能而需要一大块内存空间且需要该空间的物理地址是连续的。

从内存分配粒度的角度, 应该把内存管理组件分为如下三个层。

- ❑ 字级内存管理: 该层支持分配或者释放 32 ~ 2048B 大小的内存空间, 因为很多程序, 包括操作系统内核本身, 它们大多数时候会频繁地分配几个字节或者几十个字节到几百个字节。如果这时也给它们分配一大块固定的内存空间, 显然是种巨大的浪费。所以对于这种内存分配请求就交给字级内存管理。
- ❑ 页级内存管理: 该层支持分配或者释放 4 ~ 128KB 大小的内存空间, 对于这种内存分配请求就交给页级内存管理。
- ❑ 块级内存管理: 该层支持分配或者释放 128KB ~ 4MB 大小的内存空间。有些情况下, 程序需要非常大的连续内存空间, 如图形、图像处理软件, 对于这种内存分配

请求就交给块级内存管理。

上面内存管理组件的三个层，字级内存管理和页级内存管理在 LMOSEM 内核的功能层中，块级内存管理则是放在 LMOSEM 内核的硬件相关层中的，如图 7-1 所示。

内存管理组件已经设计好了，下面就一步一步地用代码去实现内存管理组件吧。

7.3 块级内存管理

在 7.2 节中，内存管理组件的结构图是从上往下画的，最上层是字级内存管理，然而实现时却不能从最上层开始而是需要从最底层开始的。因为上层的实现总是依赖下一层的实现，或者是依赖更下层的实现。

本节将专注于块级内存的分配和释放，从块级内存管理数据结构视图开始，到块级内存管理接口，再到块级内存分配，最后到块级内存释放。下面将详述这一实现过程。需要注意的是，块级内存管理的数据结构，在先前的初始化章节中已经初始化了。这里所谓的接口也只是块级内存管理这个层的接口，并不是整个内存管理组件的接口。

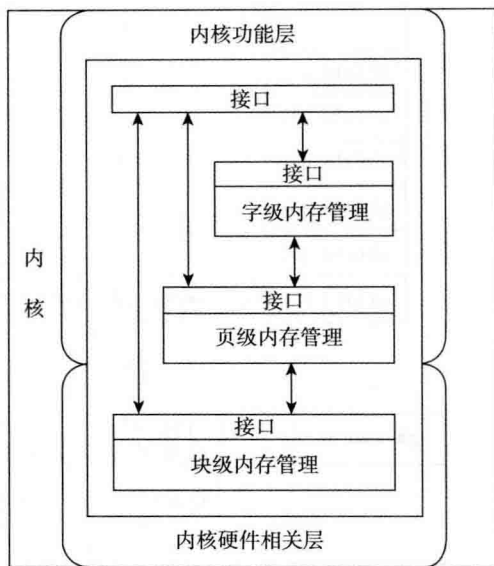


图 7-1 LMOSEM 内存管理组件的结构

7.3.1 块级内存管理数据结构视图

块级内存管理数据结构，在初始化章节中已经初始化了，所以这里会轻松许多。在这里再次提到块级内存管理的数据结构，主要是为了搞清楚它们之间的关系，这样在后面写代码实现分配或者释放内存空间时，才能心中有数，才能真正明白其中的原理。

与块级内存管理相关的数据结构，主要有三个 `phymem_t`、`alcfrlst_t`、`mmapdsc_t`。前面我们对这几个数据结构已经是再熟悉不过了，现在来用一幅图表示它们之间的关系。图 7-2 所表示的是这些数据结构初始化后的状态。

下面再来看当 LMOSEM 内核运行且分配一些不同大小的内存空间块以后，某一时刻下的状态，如图 7-3 所示。

从图 7-2 和图 7-3 中不难发现，块级内存管理最大能分配或释放 4MB 大小的内存块，最小能分配或释放 128KB 大小的内存块。根据把 `mmapdsc_t` 结构再次分成不同大小的小内存块（如把对应 4MB 大小的 `mmapdsc_t` 结构再次分成 32 个 128KB 大小的小内存块），分别把 `mmapdsc_t` 结构挂入不同的 `alcfrlst_t` 结构中。

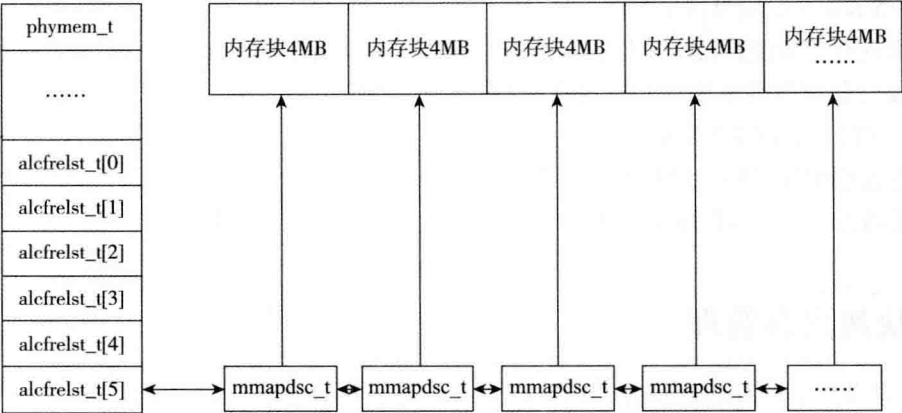


图 7-2 块级内存管理数据结构视图

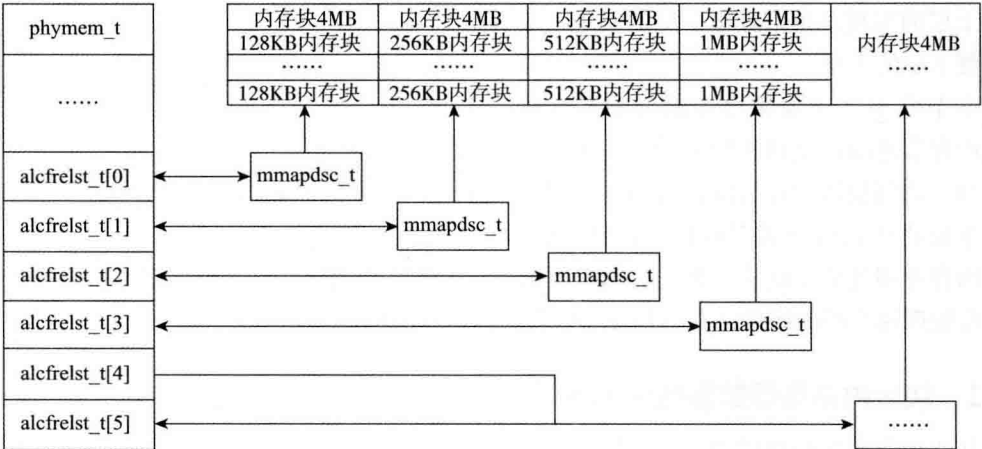


图 7-3 块级内存管理数据结构某一运行时刻下的状态

从图 7-2 和图 7-3 中大概了解了分配或者释放内存空间的步骤了，只等用代码实现它们了。

7.3.2 块级内存管理接口

前面已经说明，块级内存管理对于内存管理组件是一个独立的层，理应提供一套接口。对于块级内存管理，其接口只有两个：内存空间块分配接口和内存空间块释放接口。下面首先来实现这两个接口函数。由于块级内存管理是在 LMOSEM 内核的 hal 层中实现的，所以相应的代码应该写在 halmm.c 文件中，如代码清单 7-4[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-4 块级内存管理接口

```
adr_t hal_memallocblks(size_t blkksz)
```

```

{
    if(blksz<BLK128KB_SIZE||blksz>BLK4MB_SIZE)
    {
        return NULL;
    }
    return hal_memallocblks_core(blksz);
}
bool_t hal_memfreeblks(adr_t frebadr,size_t blksz)
{
    if(frebadr==NULL||blksz<BLK128KB_SIZE||blksz>BLK4MB_SIZE)
    {
        return FALSE;
    }
    return hal_memfreeblks_core(frebadr,blksz);
}

```

首先，排除 `hal_memallocblks_core`、`hal_memfreeblks_core` 这两个函数，这是我们后面该实现的，那么其他代码都比较简单了。都是首先检查其参数合不合规要求，若不合乎要求就立即返回一个表示错误的值。同时也不难发现，块级内存管理的分配、释放内存空间块的粒度都是在 128KB ~ 4MB，然后就是一些奇怪的数据类型，下面来介绍它们，如代码清单 7-5[xxxx/lmosem/include/bastypeinc/bastype_t.h] 所示。

代码清单7-5 LMOSEM内核代码使用的基本数据类型

```

#ifndef _BASTYPE_T_H
#define _BASTYPE_T_H
.....
typedef long sint_t;
typedef unsigned long uint_t;
typedef unsigned long adr_t;
typedef sint_t bool_t;
typedef unsigned long size_t;
.....
#define NULL      0
#define TRUE      1
#define FALSE     0
.....
#endif

```

从代码清单 7-5 来看已经非常明显了，`adr_t` 类型就是无符号的长整型，`size_t` 类型也是无符号的长整型，而 `bool_t` 类型则是有符号的长整型，之所以定义这些类型主要是为了让代码更易读、更易懂，如在写 `bool_t` 的地方直接写上 `long` 完全是可行的。

好了，块级内存管理的接口函数实现了，内存空间块的分配接口函数需要的参数，只有分配内存空间的大小，而内存空间块的释放接口函数的参数，则同时需要释放内存空间的地址和其对应的大小。

7.3.3 主分配函数

明白了前面的块级内存管理数据结构视图，想必读者已经能够大概分析出，分配内存空间块的过程如下。

第1步：根据分配内存空间块的大小，在 `phymem_t` 结构中找到合适的 `alcfrelst_t` 结构。

第2步：接着在 `alcfrelst_t` 结构中找到合适的 `mmapdsc_t` 结构。

第3步：然后操作 `mmapdsc_t` 结构，计算分配的内存空间块的地址。

第4步：最后根据情况把 `mmapdsc_t` 结构移动到合适的 `alcfrelst_t` 结构中，返回已分配的内存空间块的地址。

显然，上述4大步骤不能放在接口函数中进行，也不适合放在一个函数中实现，最好是一个步骤对应一个函数，然后有一个主函数来调用它们。下面就去写一个这样的主函数，如代码清单7-6[xxxx/lmosem/hal/halmm.c]所示。

代码清单7-6 块级内存管理的主分配函数

```

adr_t hal_memallocblks_core(size_t blksz)
{
    phymem_t* memp=&osphymem;
    size_t retbsz=0;
    cpuflg_t cpufg;
    adr_t retadr=NULL;
    alcfrelst_t* allcflp=NULL;
    alcfrelst_t* aftp=NULL;
    hal_spinlock_saveflg_cli(&memp->pmm_lock,&cpufg);
    .....
    return_step:
        hal_spinunlock_restflg_sti(&memp->pmm_lock,&cpufg);
        return retadr;
}

```

先写好这个主函数框架，后面再把完成那些步骤的函数填上去。这个函数里首先定义并初始化了一些重要的变量，暂且不管它们。先搞清楚 `hal_spinlock_saveflg_cli`、`hal_spinunlock_restflg_sti` 这两个函数。观其名就知道这是自旋锁函数，一个是加锁函数，另一个是解锁函数。那么分配内存空间为什么要加锁呢。我们知道，`phymem_t`、`alcfrelst_t`、`mmapdsc_t` 都是全局性的数据结构，LMOSEM 内核是支持多进程的，如果让两个进程同时分配内存空间（即同时操作这些数据结构），显然会出现各种不可预料的问题，可能两个进程会分配到同一块内存空间等。在操作系统内核中，任何一个细小的错误，导致的结果都是灾难性的，所以这就需要用到互斥机制了，即在同一时刻只让一个进程分配内存空间，就不会有问题了，而自旋锁只是互斥机制中的一种。既然要用到自旋锁，那么先得实现这两个函数，如代码清单7-7[xxxx/lmosem/hal/cpuctrl.c]所示。

代码清单7-7 自旋锁的加锁、解锁函数

```

void hal_spinlock_saveflg_cli(spinlock_t* lock, cpuflg_t* cpuflg)

```

```

{
    cpuflg_t tmpcpsr;
    __asm__ __volatile__(
        "mrs r7,cpsr \n\t"
        "mov %[tmpcpr],r7 \n\t" //保存CPSR寄存器的值到tmpcpsr变量中
        "orr r7,r7,%[closefirq] \n\t"
        "msr cpsr,r7 \n\t"//关掉IRQ、FRQ中断
        : [tmpcpr] "=r" (tmpcpsr) : [closefirq] "I" (CIRQFIQ) : "r7","cc","memory"
    );
    *cpuflg=tmpcpsr;//把原来的CPSR寄存器的值保存到cpuflg指向的内存单元中
    __asm__ __volatile__(
        "mov r8,#1 \n\t"
        "1:swp r7,r8,[%[lockadr]] \n\t" //原子交换自旋锁的锁值
        "cmp r7,#0 \n\t" //比较换回的锁值是否为0
        "bne 1b \n\t" //如果换回的锁值为0,则返回,否则跳回到标号1处继续交换
        : : [lockadr] "r" (&lock->lock) : "r7","r8","cc","memory"
    );
    return;
}

void hal_spinunlock_restflg_sti(spinlock_t* lock, cpuflg_t* cpuflg)
{
    __asm__ __volatile__(
        "swp %[lockval],%[lockval],[%[lockadr]] \n\t" //把0值换回自旋转锁中,表示解锁
        :: [lockval] "r"(0),[lockadr] "r" (&lock->lock) : "memory"
    );
    __asm__ __volatile__(
        "msr cpsr,%[ordcpr] \n\t" //恢复保存在cpuflg地址处的值到CPSR寄存器中
        :: [ordcpr] "r" (*cpuflg) : "cc","memory"
    );
    return;
}

```

在C语言中嵌入汇编代码,我们已经写过不少了,不必多说,代码中的注释已经说明了一切。上面的cpuflg_t类型就是一个32位的无符号的整型,用来保存CPSR寄存器的值(ARM920T CPU的CPSR寄存器就是32位的)。回到代码清单7-6中看看hal_spinlock_saveflg_cli函数的调用情形就明白了,代码中正是将cpuflg变量的地址传给hal_spinlock_saveflg_cli函数的。

通常,只要自旋锁加锁即可,这里为什么还要关掉中断呢,因为在中断处理中也可能有分配内存空间的需要。例如,一个进程正在分配内存空间时刚执行完加锁操作。中断发生了,在中断处理中也要分配内存空间,于是分配内存空间的函数再次对同一把锁执行加锁操作。你会发现这时永远不会加锁成功,CPU从此就在这里不停的测试锁值。对于mini2440平台来说,这是徒劳的,因为系统中只有一颗CPU,所以永远没有另一颗CPU去执行那个要分配内存空间的进程,从而导致最后该进程的内存空间分配完成而解锁。其实mini2440平台之上,只要关掉中断就行了,不需要自旋锁的,这样做是因为以后LMOSEM内核可能会运行在多CPU的计算平台上,这样就会降低代码的移植难度。

对于关闭、开启中断只要操作 CPSR 寄存器就行了，为什么要先保存它的值，然后操作它，最后恢复它先前的值呢？来看一个情景。如果在调用 `hal_memallocblks_core` 函数之前已经有其他函数关闭中断了，那么在 `hal_memallocblks_core` 函数中直接操作 CPSR 寄存器关闭中断是没有问题的，可是最后操作 CPSR 寄存器开启中断时，问题就来了，因为在 `hal_memallocblks_core` 函数返回到上一个函数时，那个函数依然认为中断是关闭的（其实在 `hal_memallocblks_core` 函数中已经开启中断了），而且那个函数需要中断处于关闭状态，于是 bug 就这样出现了。

本节，我们就写了一个简单的框架函数：`hal_memallocblks_core`，然后就去写自旋锁函数了，好像是文不对题吧，因为这里必须要用到自旋锁了，所以就在这里就地解决它，在后面的章节中也是一样。

7.3.4 分配时查找 `alcfrelst_t`

在 7.3.3 节中写到，分配内存空间的第一步就是要根据分配内存空间的大小查找相应的 `alcfrelst_t` 结构，本小节就来编写这个函数，使之能被 `hal_memallocblks_core` 主分配函数调用。

在编写这个函数之前，首先要思考一些问题，查找 `alcfrelst_t` 结构需要一些什么条件，首先是要有 `phymem_t` 结构，因为 `alcfrelst_t` 结构包含在 `phymem_t` 结构中，还有分配内存空间的大小，最后能返回查找到的 `alcfrelst_t` 结构的地址。下面来写代码，如代码清单 7-8[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-8 查找相应的`alcfrelst_t`

```
alcfrelst_t* hal_onblksz_findalcfrelst(alcfrelst_t** retalcfrl, size_t* retalcpsz, size_t blksz)
{
    phymem_t* memp=&osphymem;
    alcfrelst_t* aftp=NULL, *aft4mbp=&memp->pmm_sz_lsth[BLKSZ_HEAD_MAX-1];
    for(uint_t bj=0;bj<BLKSZ_HEAD_MAX;bj++)
    {
        if(memp->pmm_sz_lsth[bj].afl_sz==blksz)
        {
            aftp=&memp->pmm_sz_lsth[bj];
            *retalcfrl=aftp;
            *retalcpsz=memp->pmm_sz_lsth[bj].afl_sz;
            goto next_step;
        }
    }
    aftp=NULL;
    *retalcfrl=NULL;
    *retalcpsz=0;
next_step:
    if(aftp==NULL)
    {
```

```

        return NULL;
    }
    if(list_is_empty_careful(&aftp->afl_emtplsth)==FALSE)
    {
        return aftp;
    }
    if(list_is_empty_careful(&aftp->afl_fuemlstst)==FALSE)
    {
        return aftp;
    }
    aftp=aft4mbp;
    if(list_is_empty_careful(&aftp->afl_emtplsth)==FALSE)
    {
        return aftp;
    }
    return NULL;
}

```

代码清单 7-8 有点长，但大体上分为三段，第一段是循环比较每个 `alcfrlst_t` 结构，第二段是处理找到的 `alcfrlst_t` 结构，检查其两个链表是否为空，如果为空就进入到第三段，即把最后一个 `alcfrlst_t` 结构返回。

可是，`hal_onblksz_findalcfrlst` 函数中的前两个参数是干什么的，细心的读者会发现它们是用来返回结果的，在前面的那个循环中，一旦找到一个 `alcfrlst_t` 结构中的大小和要分配内存空间的大小（`blksz`）相等时，不管该 `alcfrlst_t` 结构中有没有 `mmapdsc_t` 结构，就将此 `alcfrlst_t` 结构的地址和其大小写回 `retalcfrl` 和 `retalcsz` 指向的内存单元中。为什么要这么做呢。例如，我们分配 128KB 大小的内存空间块，这时如果第 1 个 `alcfrlst_t` 结构中没有 `mmapdsc_t` 结构，那么就得到最后一个 `alcfrlst_t` 结构中拿一个 4MB 大小的 `mmapdsc_t` 结构，然后把它分成 32 个 128KB 大小的小内存空间块，最后还要把这个 `mmapdsc_t` 结构放在第一个 `alcfrlst_t` 结构中，方便下一次查找。在这个函数中做这个动作是为后面的操作做铺垫的，反正是查找 `alcfrlst_t` 结构不如一下子全做了。如果 `retalcfrl` 返回的值和函数返回的值相同，就不需要移动 `mmapdsc_t` 结构。

我们看到，如果循环结束后还没找到合适的 `alcfrlst_t` 结构，其返回结果都是 `NULL`，表示查找失败。如果找到合适的 `alcfrlst_t` 结构，就会跳转到 `next_step` 标号处继续检查找到的 `alcfrlst_t` 结构中有没有 `mmapdsc_t` 结构，如果没有就把最后一个 `alcfrlst_t` 结构返回，返回之前同样要检查最后一个 `alcfrlst_t` 结构中是否还有 `mmapdsc_t` 结构，`list_is_empty_careful` 函数是在 `list.h` 文件中定义的，就是为了检查一个 `list_h_t` 链表结构中是否还有后续链表。`phymem_t` 结构在主函数中已经加锁了，所以这里不能再加锁了。

找到了 `alcfrlst_t` 结构，离成功分配内存空间块就近了一大步，但是这一步走得并不困难，后面的就困难多了。下面就去解决这个问题，无论它有多么困难。

7.3.5 分配时查找和操作 mmapdsc_t

在 7.3.4 节中，按照分配内存空间块的大小找到了 `alcfrelst_t` 结构并且确定了其中有没有 `mmapdsc_t` 结构，但是并没有真正确定一个 `mmapdsc_t` 结构是不是合适，这里将在已经确定的 `alcfrelst_t` 结构中，查找合适的 `mmapdsc_t` 结构并对其操作，以完成最终的内存空间块的分配。

首先，由于查找和操作 `mmapdsc_t` 结构都和同一个 `alcfrelst_t` 结构有关，于是我们再写一个函数，在这个函数里去查找和操作 `mmapdsc_t` 结构。其实这个函数也是一个框架函数，它是根据不同的内存空间块的大小分别调用另一个函数来处理的。还是先写好这个函数吧，如代码清单 7-9[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-9 区分内存空间块的大小

```

adr_t hal_onalcfrel_allocblks(alcfrelst_t* allclh, alcfrelst_t* relaclh, size_t relalcsz)
{
    adr_t retadr=NULL;
    switch(relalcsz)
    {
        case BLK128KB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_128KB, BLK128KB_MASK, BLK128KB_
                BITL, allclh, relaclh);
            break;
        case BLK256KB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_256KB, BLK256KB_MASK, BLK256KB_
                BITL, allclh, relaclh);
            break;
        case BLK512KB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_512KB, BLK512KB_MASK, BLK512KB_
                BITL, allclh, relaclh);
            break;
        case BLK1MB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_1MB, BLK1MB_MASK, BLK1MB_
                BITL, allclh, relaclh);
            break;
        case BLK2MB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_2MB, BLK2MB_MASK, BLK2MB_
                BITL, allclh, relaclh);
            break;
        case BLK4MB_SIZE:
            retadr=hal_onmapdsc_allcblks(MAPF_ACSZ_4MB, BLK4MB_MASK, BLK4MB_
                BITL, allclh, relaclh);
            break;
        default:
            retadr=NULL;
            break;
    }
    return retadr;
}

```

不难发现这个函数有三个参数，第一个参数是用于查找 `mmapdsc_t` 结构的 `alcfrelst_t` 结构的指针，第二个参数是操作完 `mmapdsc_t` 结构后，将要把 `mmapdsc_t` 结构移动到哪个 `alcfrelst_t` 结构的指针，原因在前面已经说明，`MAPF_ACSZ_4MB`、`BLK4MB_MASK`、`BLK4MB_BITL` 这些宏是在 `halmm_t.h` 文件中定义的，若不明白其含义，可参阅前面章节中的 `mmapdsc_t` 数据结构。下面来写 `hal_onmapdsc_allcblks` 函数，如代码清单 7-10[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-10 查找并操作mmapdsc_t

```

mmapdsc_t* hal_onafl_findmapdsc(alcfrelst_t* allclh)
{
    mmapdsc_t* mp=NULL;
    //先非满队列 (allclh->afl_fuemlsth) 中看有没有mmapdsc_t结构，如果有就是第一个
    if(list_is_empty_careful(&allclh->afl_fuemlsth)==FALSE)
    {
        mp=list_entry(allclh->afl_fuemlsth.next,mmapdsc_t,map_list);
        return mp;
    }
    //然后在空队列 (allclh->afl_emtplsth) 中看有没有mmapdsc_t结构，如果有就是第一个
    if(list_is_empty_careful(&allclh->afl_emtplsth)==FALSE)
    {
        mp=list_entry(allclh->afl_emtplsth.next,mmapdsc_t,map_list);
        return mp;
    }
    //都没有的话就返回NULL，不可能在allclh->afl_fulllsth中，因为它里面装的是没有空闲内存块的
    //mmapdsc_t结构
    return NULL;
}

adr_t hal_onmapdsc_allcblks(u32_t mflg, u32_t mask,u32_t bitls,alcfrelst_t* aflp,alcfrelst_t* mvtoaflp)
{
    adr_t retadr=NULL;
    mmapdsc_t* map=NULL;
    uint_t bi=0;
    u32_t fg=0xffffffff;
    if(bitls>32)
    {
        return NULL;
    }
    map=hal_onafl_findmapdsc(aflp);
    //没有找到mmapdsc_t结构，当然不能继续执行了，只能返回NULL
    if(map==NULL)
    {
        return NULL;
    }
    //循环扫描map->map_allcount中的位到bitls位数结束，记录哪个位为0
    for(;bi<bitls;bi++)
    {
        if((map->map_allcount>>bi)&1)==0)

```

```

        {
            goto next_step;
        }
    }
    //如果循环结束后map->map_allcount中没有为0的位, 则bi=0xffffffff
    bi=0xffffffff;
    next_step:
    if (bi==0xffffffff)
    {
        return NULL;
    }
    //计算返回地址, 并且对计算出的地址进行合理化检查
    retadr=map->map_phyadr+(mvtoaflp->afl_sz*bi);
    if (retadr<map->map_phyadr||retadr>=map->map_phyadrend)
    {
        return NULL;
    }
    //将map->map_allcount中的第bi位置为1, 表示已分配
    map->map_allcount|=(1<<bi);
    //清除mmapdsc_t结构中的相关标志位
    map->map_flg&=fg;
    //设置新的标志位, 该标志位正是表示其中小内存块大小的位
    map->map_flg|=mflg;
    //如果分配后的map->map_allcount中的位和mask码相等, 表示其中没有空闲内存块了, 所以要移动到
    //mvtoaflp->afl_fulllsth (满) 队列中去
    if ((map->map_allcount&mask)==mask)
    {
        list_move_tail(&map->map_list, &mvtoaflp->afl_fulllsth);
        return retadr;
    }
    //否则移动到mvtoaflp->afl_fuemlsth (非满) 队列中去
    list_move_tail(&map->map_list, &mvtoaflp->afl_fuemlsth);
    return retadr;
}

```

这个 `hal_onmmapdsc_allcblks` 函数的代码很长, 不过注释也很多, 看注释就能明白, 其主要动作就是调用 `hal_onafl_findmmapdsc` 函数, 从 `alcflrst_t` 结构中获取一个 `mmapdsc_t` 结构, 然后根据 `mmapdsc_t` 结构中的信息计算分配内存空间块的首地址, 改写 `mmapdsc_t` 结构的标志和 `map_allcount` 域, 标识已经分配走一块内存空间块, 最后根据 `mmapdsc_t` 结构中是否还有空闲的内存空间块, 决定把该 `mmapdsc_t` 结构移动到哪个 `alcflrst_t` 结构的某个队列中。由于不同的内存空间分配大小, 它所需要的标志位、mask、扫描位的宽度都不相同, 所以 `hal_onmmapdsc_allcblks` 函数需要有 `mflg`、`mask`、`bitls` 这三个参数。

到这里, 关于块级内存分配的函数包括主分配函数, 我们一共写了 5 个函数, 这 5 个函数各自完成自己的工作, 但是只有让它们结合起来才能完成内存空间块的分配, 现在就把它们装配在一起, 如代码清单 7-11[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-11 块级内存管理的主分配函数

```

adr_t hal_memallocblks_core(size_t blksz)
{
    phymem_t* memp=&osphymem;
    size_t retbsz=0;
    cpuflg_t cpufg;
    adr_t retadr=NULL;
    alcfrelst_t* allcflp=NULL;
    alcfrelst_t* aftp=NULL;
    hal_spinlock_saveflg_cli(&memp->pmm_lock,&cpufg);
    aftp=hal_onblksz_findalcfrelst(&allcflp,&retbsz,blksz);
    if(aftp==NULL||allcflp==NULL||retbsz==0)
    {
        retadr=NULL;
        goto return_step;
    }
    if(aftp->afl_sz!=retbsz&&aftp->afl_sz!=BLK4MB_SIZE)
    {
        retadr=NULL;
        goto return_step;
    }
    retadr=hal_onalcfrel_allocblks(aftp,allcflp,retbsz);
return_step:
    hal_spinunlock_restflg_sti(&memp->pmm_lock,&cpufg);
    return retadr;
}

```

代码清单 7-11 已经很简单了，不用多说。块级内存的分配到这里就完成了，只有分配是不行的，还要能释放。不过在这之前，首先得确定块级内存的分配代码是不是正确的。下面我们就去确定一下。

7.3.6 分配代码写得对吗

关于块级内存分配，前面章节中，我们写了不少代码，究竟这些代码实现得对不对，凭感觉是不能确定的，得上开发板试一试。何况它处于内存管理组件的最底层，更应严格保证它的健壮性。

证明分配代码对与不对，写个 `testblkalloc` 函数去调用块级内存分配接口，把分配内存空间块的首地址打印出来看一看就知道了。下面就来写一写这个函数，如代码清单 7-12[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-12 测试块级内存分配

```

void testblkalloc()
{
    size_t ablksz=0;
    adr_t retadr=NULL;
    for(uint_t bli=0;bli<6;bli++)

```

```

{
    ablkksz=BLK128KB_SIZE<<bli;
    for(uint_t i=0;i<2;i++)
    {
        retadr=hal_memallocblks(ablokksz);
        if(retadr==NULL)
        {
            hal_sysdie("NOT MEMBLK;return NULL");
        }
        printfk("allocblksz:%x return adrr:%x \n\r",ablokksz,retadr);
    }
}
return;
}

```

testblkalloc 函数中，主要有两重循环驱动，外层循环控制分配大小，内层循环控制分配次数。不难看出，每种内存空间分配大小都分配两次。接着把 testblkalloc 函数加入到 init_hal 函数的最后让其调用。按照前面的方法编译好 LMOSEM 内核并连接好开发板，把编译好的 LMOSEM 内核下载到开发板中运行，正常情况下会出现如图 7-4 所示的情形。

```

深度终端
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30008000h,TOTAL:37534]
RECEIVED FILE SIZE: 37534 (36KB/S, 1S)
Downloaded file at 0x30000000, size = 37524 bytes
allocblksz:20000 return adrr:30020000
allocblksz:20000 return adrr:30040000
allocblksz:40000 return adrr:30400000
allocblksz:40000 return adrr:30440000
allocblksz:80000 return adrr:30800000
allocblksz:80000 return adrr:30880000
allocblksz:100000 return adrr:30c00000
allocblksz:100000 return adrr:30d00000
allocblksz:200000 return adrr:31000000
allocblksz:200000 return adrr:31200000
allocblksz:400000 return adrr:31400000
allocblksz:400000 return adrr:31800000
LMOSEM SYSTEM IS DIE LMOSEM RUN!

```

图 7-4 测试块级内存管理的分配代码

图 7-4 中，出现的分配内存空间的地址值，表明我们的代码没有任何问题，同时，这些地址值也告诉我们，移动 mmapdsc_t 结构的正确性。最后的“……LMOSEM RUN”字符串表明代码的控制路径已经进入到 init_krl 函数中了。

块级内存的分配代码测试正确了，下面可以放心地去实现块级内存的释放代码了。

7.3.7 主释放函数

对于已经分配的内存空间块若不使用了，就应该释放，否则不管有多少内存空间都会有

耗尽的时候，现在就来实现块级内存的释放代码。

释放内存空间块就是分配内存空间块的逆向过程，步骤如下。

第1步：根据释放内存空间块的大小，在 `phymem_t` 结构中找到合适的 `alcfrelst_t` 结构。

第2步：根据释放内存空间块的地址，在 `alcfrelst_t` 结构中找到合适的 `mmapdsc_t` 结构。

第3步：操作找到的 `mmapdsc_t` 结构，表明那块内存已经释放。

第4步：根据当前 `mmapdsc_t` 结构的状态，把它移动到合适的 `alcfrelst_t` 结构中。

关于块级内存释放的接口函数，前面已经写过了，它只是对传递过来的参数做合理化检查，这里主要是实现块级内存的主释放函数，如代码清单 7-13[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-13 块级内存管理的主释放函数

```
bool_t hal_memfreeblks_core(adr_t frebadr, size_t blksz)
{
    phymem_t* memp=&osphymem;
    bool_t retstus=FALSE;
    cpuflg_t cpufg;
    alcfrelst_t*
    aftp=NULL, *mvtoaftp=&memp->pmm_sz_lsth[BLKSZ_HEAD_MAX-1];
    hal_spinlock_saveflg_cli(&memp->pmm_lock, &cpufg);
    .....
    return_step:
        hal_spinunlock_restflg_sti(&memp->pmm_lock, &cpufg);
        return retstus;
}
```

在代码清单 7-13 中，`hal_memfreeblks_core` 函数和块级内存的主分配函数的架构差不多，都是开始初始化了一些变量，接着对 `phymem_t` 结构加锁，然后做了一些操作，最后解锁返回一个状态值表示释放操作是否成功。不过该函数的参数多了一个释放内存空间块的地址，返回值也不一样了。

上面对释放内存空间块的步骤已经说明了，而且块级内存的主释放函数也已经写好了，下面就去用代码实现那些释放内存空间块的步骤。

7.3.8 释放时查找 `alcfrelst_t`

按照 7.3.7 节中的内存空间块的释放步骤，首先是根据内存空间块的释放大小，在 `phymem_t` 结构中找到合适的 `alcfrelst_t` 结构，因为与释放内存空间块相关的 `mmapdsc_t` 结构，就挂载在某个 `alcfrelst_t` 结构中。

这个查找过程将比内存空间块分配时的查找过程要容易很多，因为内存空间块分配时还要返回 `mmapdsc_t` 结构将要移动到的那个 `alcfrelst_t` 结构的指针。虽然释放内存空间块时也

要移动 `mmapdsc_t` 结构，但那仅是当 `mmapdsc_t` 结构中所有的小块内存空间都为空闲时，即 `mmapdsc_t` 结构所对应的是一块 4MB 的大块空闲内存时，才将该 `mmapdsc_t` 结构移动到最后一个 `alcfrelst_t` 结构中，即挂载 4MB 大小的 `mmapdsc_t` 结构的地方。所以这个过程很简单，我们就把查找 `alcfrelst_t` 结构的过程直接放在 `hal_memfreeblks_core` 函数内进行，下面来写这个查找过程的代码，如代码清单 7-14[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-14 查找相应的`alcfrelst_t`

```
bool_t hal_memfreeblks_core(adr_t frebadr, size_t blksz)
{
    phymem_t* memp=&osphymem;
    bool_t retstus=FALSE;
    cpuflg_t cpufg;
    alcfrelst_t* aftp=NULL, *mvtoaftp=&memp->pmm_sz_lsth[BLKSZ_HEAD_MAX-1];
    hal_spinlock_saveflg_cli(&memp->pmm_lock, &cpufg);
    for(uint_t fi=0; fi<BLKSZ_HEAD_MAX; fi++)
    {
        if(memp->pmm_sz_lsth[fi].afl_sz==blksz)
        {
            aftp=&memp->pmm_sz_lsth[fi];
            goto next_step;
        }
    }
    aftp=NULL;
next_step:
    .....
return_step:
    hal_spinunlock_restflg_sti(&memp->pmm_lock, &cpufg);
    return retstus;
}
```

在代码清单 7-14 中，`mvtoaftp` 变量已经表明了，如果要移动 `mmapdsc_t` 结构，就移动到最后一个 `alcfrelst_t` 结构中。查找过程是一个循环，在循环中用内存空间块的释放大 小 比 对 所 有 的 `alcfrelst_t` 结 构 的 `afl_sz` 域，一旦它们相等就跳到 `next_step` 标号处继续相关的操作。

以上的查找过程非常简单，不必多说，如果没有找到相应的 `alcfrelst_t` 结构，`aftp` 就会等于 `NULL`，接着后面就会返回 `FALSE`，如果找到了 `alcfrelst_t` 结构，就会继续在其中查找并操作相应的 `mmapdsc_t` 结构，这也是下一节我们该做的。

7.3.9 释放时查找和操作 `mmapdsc_t`

在 7.3.8 节中查找 `alcfrelst_t` 结构，就是为了能进一步找到和释放内存空间块相关的 `mmapdsc_t` 结构，然后操作 `mmapdsc_t` 结构中的相关域，以表示该内存空间块已经释放了，最后移动 `mmapdsc_t` 结构。

上述过程，有点复杂，不能在一个函数中完成，更不能在主释放函数中完成，否则代码

将很难写下去，首先来写一个 `hal_onalcfrel_freeblks` 函数，它根据不同的释放内存空间块的大小，分别调用 `hal_onmapdsc_freeblks` 函数来做具体的处理，如代码清单 7-15[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-15 区分内存空间块的大小

```
bool_t hal_onalcfrel_freeblks(adr_t frebadr,alcfrelst_t* allclh,alcfrelst_t* mvaflh,size_t
    relalcsz)
{
    bool_t retstus=FALSE;
    switch(relalcsz)
    {
        case BLK128KB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK128KB_
                MASK,allclh,mvaflh);
            break;
        case BLK256KB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK256KB_
                MASK,allclh,mvaflh);
            break;
        case BLK512KB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK512KB_
                MASK,allclh,mvaflh);
            break;
        case BLK1MB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK1MB_
                MASK,allclh,mvaflh);
            break;
        case BLK2MB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK2MB_
                MASK,allclh,mvaflh);
            break;
        case BLK4MB_SIZE:
            retstus=hal_onmapdsc_freeblks(frebadr,MAPF_ACSZ_4MB,BLK4MB_
                MASK,allclh,mvaflh);
            break;
        default:
            retstus=FALSE;
            break;
    }
    return retstus;
}
```

接着来实现 `hal_onmapdsc_freeblks` 函数，如代码清单 7-16[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-16 操作mmapdsc_t

```
bool_t hal_onmapdsc_freeblks(adr_t frebadr,u32_t mflg, u32_t mask,alcfrelst_t*
    aflp,alcfrelst_t* mvtoaflp)
{
    mmapdsc_t* map=NULL;
```

```

uint_t bitnr;
u32_t fg=0xfffffffff;
map=hal_free_findmapdsc(frebadr,aflp)
//根据释放内存空间块的地址和所在的alcfrelst_t结构, 查找mmapdsc_t结构
if(map==NULL) ;
{
    return FALSE;
}
//根据内存空间块的释放地址计算该内存空间块, 在map_allcount中对应的位, 如果不
//懂, 请思考前面分配函数中是怎么通过map_allcount中的位, 计算出返回地址的
bitnr=(uint_t) (frebadr-map->map_phyadr) / (uint_t) (aflp->afl_sz);
if(bitnr>31)
{
    return FALSE;
}
//检查一下计算出的map_allcount中的对应位是否为0, 为0就表示出错了
if(((map->map_allcount>>bitnr)&1)!=1)
{
    return FALSE;
}
//把释放内存空间块在map_allcount中对应的位清0, 表示已经释放
map->map_allcount&=(~(1<<bitnr));
//如果map_allcount中表示内存空间块状态相应的位都是0, 即它已经是一整块空
//闲内存时, 就设置相应的标志, 然后移动mmapdsc_t结构到最后一个alcfrelst_t结构中
if((map->map_allcount&mask)==0)
{
    map->map_flg&=fg;
    map->map_flg|=mflg;
    list_move(&map->map_list,&mvtoaflp->afl_emtplsth);
}
return TRUE;
}

```

hal_onmmapdsc_freeblks 函数中, 最后两个参数: aflp 是用于查找 mmapdsc_t 结构的, mvtoaflp 是用于移动 mmapdsc_t 结构的。mflg、mask 是用于设置和判断 mmapdsc_t 结构中相关标志的。其他部分, 代码中的注释已经很明显了。

可是还没有写 hal_free_findmapdsc 函数呢, 少了它怎么能行呢。下面就来实现它, 如代码清单 7-17[xxxx\lmosem\hal\halmm.c] 所示。

代码清单7-17 查找mmapdsc_t

```

mmapdsc_t* hal_free_findmapdsc(adr_t frebadr,alcfrelst_t* allclh)
{
    mmapdsc_t* mp=NULL;
    list_h_t* tmlst=NULL;
    if(list_is_empty_careful(&allclh->afl_fuemlsth)==FALSE)
    {
        list_for_each(tmlst,&allclh->afl_fuemlsth)
        {

```

```

        mp=list_entry(tmplst,mmapdsc_t,map_list);
        if(frebadr<mp->map_phyadrend&&frebadr>=mp->map_phyadr)
        {
            return mp;
        }
    }
}
if(list_is_empty_careful(&allclh->afl_fulllsth)==FALSE)
{
    list_for_each(tmplst,&allclh->afl_fulllsth)
    {
        mp=list_entry(tmplst,mmapdsc_t,map_list);
        if(frebadr<mp->map_phyadrend&&frebadr>=mp->map_phyadr)
        {
            return mp;
        }
    }
}
return NULL;
}

```

代码清单 7-17 中，两个 `list_is_empty_careful` 函数就是为了判断两个链表是否为空，不为空才进行查找。由于要释放的内存空间块对应的 `mmapdsc_t` 结构只可能出现在 `allclh->afl_fuemlsth` 链表和 `allclh->afl_fulllsth` 链表中，所以只在这两个链表中查找。不太明白的可能就是 `list_for_each`，它又不像是函数，又不像是 C 程序的语句。其实它是一个宏，定义在 `list.h` 文件中，如代码清单 7-18[xxxx/lmosem/include/bastypeinc/list.h] 所示。

代码清单7-18 遍历相应的链表

```

#define list_for_each(pos, head) for (pos = (head)->next; pos != (head); pos = pos->next)

```

原来就是一条 `for` 循环语句，如代码 `list_for_each(tmplst,&allclh->afl_fulllsth)`，经过 C 编译器预处理之后，就会变成如代码清单 7-19 所示。

代码清单7-19 遍历相应的mmapdsc_t

```

for(tmplst=(&allclh->afl_fulllsth)->next; tmplst!=( &allclh->afl_fulllsth);
    tmplst= tmplst->next)
{
    .....
}

```

而代码清单 7-17 中的 `list_entry` 宏，正是根据链表地址 (`tmplst`) 获取一个具体 `mmapdsc_t` 结构的指针的，紧接着就是一个比较操作，看看释放的内存空间块的地址是不是落在这个 `mmapdsc_t` 结构所管理的内存空间块中。

本小节共写了 3 个函数，我们需要把它们结合起来，才能完成释放内存空间块的目的。这个结合操作非常简单，只要在主释放函数 `hal_memfreeblks_core` 中调用 `hal_onalcfrel_`

freeblks 函数即可，如代码清单 7-20[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-20 块级内存管理的主释放函数

```
bool_t hal_memfreeblks_core(adr_t frebadr, size_t blksz)
{
    phymem_t* memp=&osphymem;
    bool_t retstus=FALSE;
    cpuflg_t cpufg;
    alcfrelst_t* aftp=NULL, *mvtoaftp=&memp->pmm_sz_lsth[BLKSZ_HEAD_MAX-1];
    hal_spinlock_saveflg_cli(&memp->pmm_lock, &cpufg);
    for(uint_t fi=0; fi<BLKSZ_HEAD_MAX; fi++)
    {
        if(memp->pmm_sz_lsth[fi].afl_sz==blksz)
        {
            aftp=&memp->pmm_sz_lsth[fi];
            goto next_step;
        }
    }
    aftp=NULL;
next_step:
    if(aftp==NULL)
    {
        retstus=FALSE;
        goto return_step;
    }
    retstus=hal_onalcfrel_freeblks(frebadr, aftp, mvtoaftp, blksz);
return_step:
    hal_spinunlock_restflg_sti(&memp->pmm_lock, &cpufg);
    return retstus;
}
```

好了，释放内存空间块的全过程就到此结束了。下面就去测试一下，看看这些代码是不是能正常工作，必然块级内存管理是其他内存管理层的基础。

7.3.10 测试块级内存管理层

再简单的代码也得放在开发板上测试检验，才能知道代码是否有问题，凭感觉认为代码很正确，这是很危险的。

下面就来写一写测试代码，该代码能同时测试块级内存管理的分配、释放操作。但目前所能做的无非就是分别调用分配、释放内存空间块的接口，然后打印出结果，看一看是不是预期的，如代码清单 7-21[xxxx/lmosem/hal/halmm.c] 所示。

代码清单7-21 测试块级内存管理

```
void test_blkmm()
{
    adr_t retadra[14], retadr=0, retadroid=0, retadrend=0;
```

```

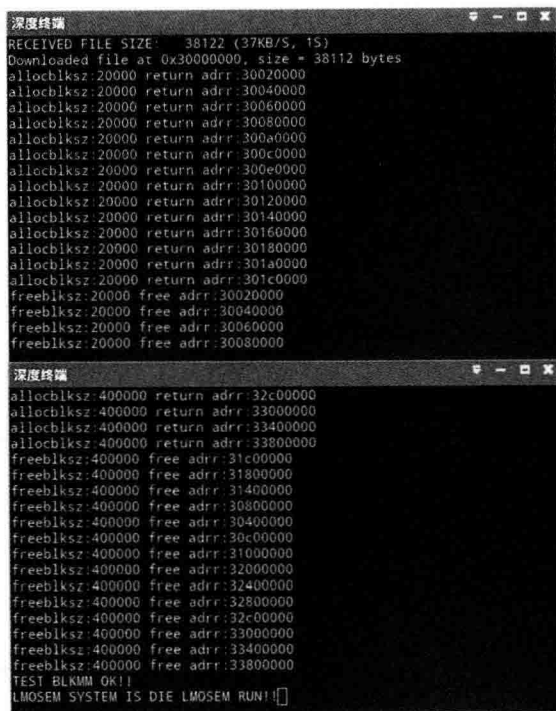
size_t ablksz=0;bool_t retstus=FALSE;
retadrolld=hal_memallocblks(BLK128KB_SIZE);
if(retadrolld==NULL)
    hal_sysdie("NOT MEMALLOCBLKold;return NULL");
if(hal_memfreeblks(retadrolld,BLK128KB_SIZE)==FALSE)
    hal_sysdie("NOT MEMFREEBLKold;return FALSE");
for(uint_t bli=0;bli<6;bli++)
{
    ablksz=BLK128KB_SIZE<<bli;
    for(uint_t i=0;i<14;i++)
    {
        retadr=hal_memallocblks(ablksz);
        if(retadr==NULL)
            hal_sysdie("NOT MEMALLOCBLK;return NULL");
        printfk("allocblksz:%x return adrr:%x \n\r",ablksz,retadr);
        retadra[i]=retadr;
    }
    for(uint_t j=0;j<14;j++)
    {
        retstus=hal_memfreeblks(retadra[j],ablksz);
        if(retstus==FALSE)
            hal_sysdie("NOT MEMFREEBLK;return FALSE");
        printfk("freeblksz:%x free adrr:%x \n\r",ablksz,retadra[j]);
    }
}
retadrend=hal_memallocblks(BLK128KB_SIZE);
if(retadrend==NULL)
    hal_sysdie("NOT MEMALLOCBLKend;return NULL");
if(retadrend==retadrolld)
printfk("TEST BLKMM OK!!\n\r");
return;
}

```

代码清单 7-21 的 `test_blkmm` 函数中，开始分配了一个 128KB 大小的内存空间块，接着又将其释放了，而返回的地址值却保存在 `retadrolld` 变量中。然后是每种大小的内存空间块分配 15 次后又全部释放了。最后又分配了一个 128KB 大小的内存空间块并且和首次分配返回的地址值（`retadrolld`）进行比较，预期中它们应该是相等的，因为代码清单 7-21 中的分配、释放操作是对等的，也就是说最终内存空间应该是还原了，所以 `retadrolld` 变量和 `retadrend` 变量中的值，应该是相等的。

读者可以把 `test_blkmm` 函数，加入 `init_hal` 函数中的最后并替换掉 `testblkalloc` 函数，然后按照前面章节中的说明，编译好 LMOSEM 内核，连接好 mini2440 开发板，把 LMOSEM 内核下载到开发板中运行，正常情况下会出现如图 7-5 所示的情况。

好了，块级内存管理层可以正常工作了，后面就可以以块级内存管理层为基础，开发更高级别的内存管理层了。



```
深度终端
RECEIVED FILE SIZE: 38122 (37KB/S, 15)
Downloaded file at 0x30000000, size = 38112 bytes
allocblksz:20000 return addr:30020000
allocblksz:20000 return addr:30040000
allocblksz:20000 return addr:30060000
allocblksz:20000 return addr:30080000
allocblksz:20000 return addr:300a0000
allocblksz:20000 return addr:300c0000
allocblksz:20000 return addr:300e0000
allocblksz:20000 return addr:30100000
allocblksz:20000 return addr:30120000
allocblksz:20000 return addr:30140000
allocblksz:20000 return addr:30160000
allocblksz:20000 return addr:30180000
allocblksz:20000 return addr:301a0000
allocblksz:20000 return addr:301c0000
freeblksz:20000 free addr:30020000
freeblksz:20000 free addr:30040000
freeblksz:20000 free addr:30060000
freeblksz:20000 free addr:30080000
allocblksz:400000 return addr:32c00000
allocblksz:400000 return addr:33000000
allocblksz:400000 return addr:33400000
allocblksz:400000 return addr:33800000
freeblksz:400000 free addr:31c00000
freeblksz:400000 free addr:31800000
freeblksz:400000 free addr:31400000
freeblksz:400000 free addr:30800000
freeblksz:400000 free addr:30400000
freeblksz:400000 free addr:30c00000
freeblksz:400000 free addr:31000000
freeblksz:400000 free addr:32000000
freeblksz:400000 free addr:32400000
freeblksz:400000 free addr:32800000
freeblksz:400000 free addr:32c00000
freeblksz:400000 free addr:33000000
freeblksz:400000 free addr:33400000
freeblksz:400000 free addr:33800000
TEST BLKMEM OK!!
LMOSEM SYSTEM IS DIE LMOSEM RUN!!
```

图 7-5 测试块级内存管理层

7.4 页级内存管理

7.3 节实现了块级内存管理层，然而它分配一块内存空间的大小最少也有 128KB。这对于小块内存空间的分配请求，无疑是种浪费。所以页级内存管理层的主要任务，就是把一块内存空间划分成 4KB 大小的页，然后返回给请求者。注意，后面的页或者页面，即表示 4KB 大小的内存空间块。

怎么实现这一机制呢，我们用了一种称为内存池的思想。什么是内存池，当我们实现之后，自然就会明白的。Linux 及其他操作系统也实现了这种技术，只是实现的方式各不相同。当然，LMOSEM 内核的实现方式也不相同，但是都是源于这种思想。

本节将从设计页级内存管理层的接口及相关的数据结构开始，然后实现分配和释放内存页面，最后测试页级内存管理层。下面就用代码去实现它们。

7.4.1 页级内存管理接口及调用流程

同样，还是需要从接口函数开始做起，对于页级内存管理层的接口，也无非就是内存页的分配和释放，不过在这里有点复杂。由于 LMOSEM 内核的页级内存管理层和后面的字级内存管理层会融合在一起，虽然结构很清晰，然而需要在接口函数中，根据内存页的分配或

者释放大小进行相应的判断和选择。

由于页级内存管理层是在 LMOSEM 内核功能层中实现的, 所以首先要在 “xxxx/lmosem/kernel” 目录下建立一个 C 语言代码文件 krlpagempol.c, 当然还要在 “xxxx/lmosem/include/knlinc” 目录下建立两个头文件 krlpagempol.h、krlpagempol_t.h, 最后把 krlpagempol.o 加入到 buildfile.h 文件中的 BUILD_KRNL_OBJS 宏的后面, 这个步骤前面已经多次提到了, 不再详述。

下面在这个 krlpagempol.c 文件中写上两函数, 如代码清单 7-22[xxxx/lmosem/kernel/krlpagempol.c] 所示。这两个函数都是首先对参数进行合理性检查, 然后调用另一个函数。

代码清单7-22 内存管理接口

```

adr_t kmempool_new(size_t msiz)
{
    if (msiz < KMEMPALCSZ_MIN || msiz > KMEMPALCSZ_MAX)
    {
        return NULL;
    }
    return kmempool_onsize_new(msiz);
}

bool_t kmempool_delete(adr_t fradr, size_t frsz)
{
    if (fradr == NULL || frsz < KMEMPALCSZ_MIN || frsz > KMEMPALCSZ_MAX)
    {
        return FALSE;
    }
    return kmempool_onsize_delete(fradr, frsz);
}

```

代码清单 7-22 的两个函数中最后都是调用了一个 kmempool_onsize_xxxx 函数, 可是这个函数是页级内存管理层的主分配函数或者主释放函数吗。当然不是的, 下面分别画了两幅图来表示它们到主分配函数和主释放函数的调用路径。先来看看内存页的分配函数的情况, 如图 7-6 所示。

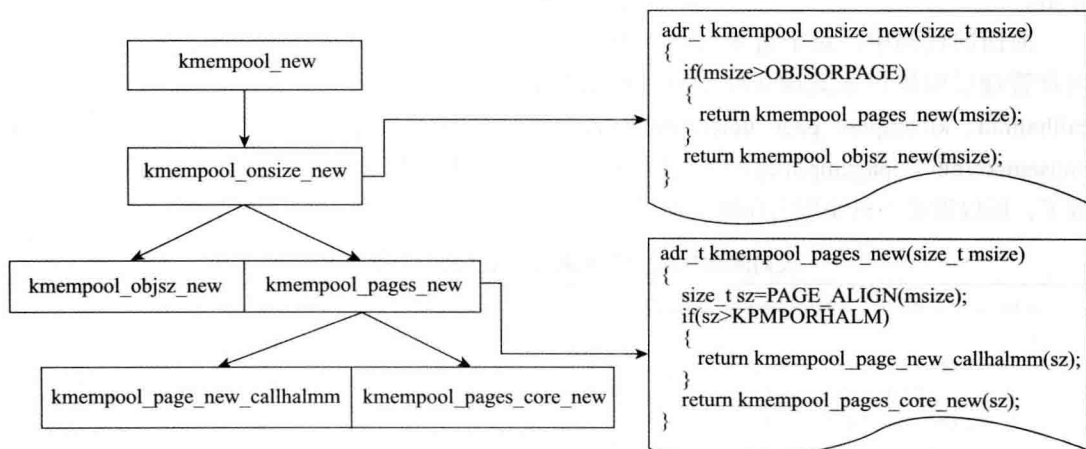


图 7-6 页级内存管理层主分配函数的调用路径

图 7-6 中出现了两条分水岭，一条是因为 OBJORPAGE (#define OBJSORPAGE 2048)，只有分配大小大于 2048 时，才调用页级内存管理层的分配函数 kmempool_pages_new，否则调用字节内存管理层的分配函数：kmempool_objsz_new（这是后面该实现的）。另一条是因为 KMPORNALM (#define KMPORHALM (PAGE_SIZE*31), #define PAGE_SIZE 4096)，如果分配大小大于 31 个页面，就直接调用块级内存管理层的分配函数，分配一个内存块。否则调用页级内存管理层的分配函数 kmempool_pages_core_new。

下面来看看页级内存管理层的释放接口，如图 7-7 所示。

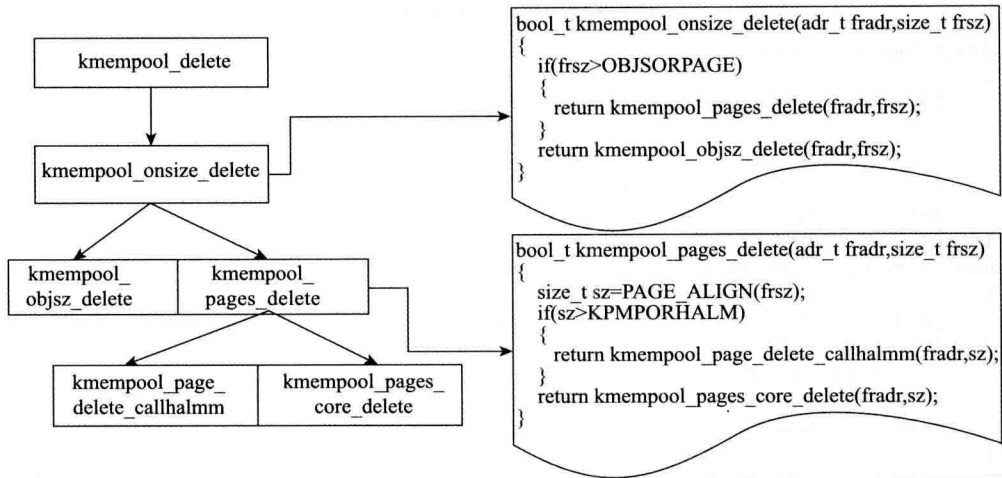


图 7-7 页级内存管理层主释放函数的调用路径

与分配接口一样，也有两条分水岭，只不过对于页级内存管理层的主释放函数 kmempool_pages_core_delete，它所执行的动作正是前面分配接口函数的逆过程，后面会有详细的介绍。

前面的代码中，对于请求大小大于 31 个页面时，无论分配还是释放，都会调用块级内存管理层的接口来完成实际的分配或者释放操作。上面两个函数 kmempool_page_new_callhalmm、kmempool_page_delete_callhalmm 就是做这个事情的，如代码清单 7-23[xxxx/lmosem/kernel/kr/pagempol.c] 所示。因为块级内存管理层已经支持至少 32 个页面的分配、释放了，所以没必要再使用内存池技术了。

代码清单7-23 区分分配、释放的内存空间大小

```
adr_t kmempool_page_new_callhalmm(size_t msize)
{
    if(msize<=BLK128KB_SIZE)
        return hal_memallocblks(BLK128KB_SIZE);
    if(msize<=BLK256KB_SIZE)
        return hal_memallocblks(BLK256KB_SIZE);
    if(msize<=BLK512KB_SIZE)
```

```

        return hal_memallocblks(BLK512KB_SIZE);
    if(msize<=BLK1MB_SIZE)
        return hal_memallocblks(BLK1MB_SIZE);
    if(msize<=BLK2MB_SIZE)
        return hal_memallocblks(BLK2MB_SIZE);
    if(msize<=BLK4MB_SIZE)
        return hal_memallocblks(BLK4MB_SIZE);
    return NULL;
}

bool_t kmempool_page_delete_callhalmm(adr_t fradr,size_t frze)
{
    if(frze<=BLK128KB_SIZE)
        return hal_memfreeblks(fradr,BLK128KB_SIZE);
    if(frze<=BLK256KB_SIZE)
        return hal_memfreeblks(fradr,BLK256KB_SIZE);
    if(frze<=BLK512KB_SIZE)
        return hal_memfreeblks(fradr,BLK512KB_SIZE);
    if(frze<=BLK1MB_SIZE)
        return hal_memfreeblks(fradr,BLK1MB_SIZE);
    if(frze<=BLK2MB_SIZE)
        return hal_memfreeblks(fradr,BLK2MB_SIZE);
    if(frze<=BLK4MB_SIZE)
        return hal_memfreeblks(fradr,BLK4MB_SIZE);
    return FALSE;
}

```

代码清单 7-23 中的两个函数非常简单，不用多说，无非就是看分配、释放的大小落在哪种大小的区间中，例如，分配、释放的大小比 128KB 多一个字节，那么就会按照 256KB 大小进行分配或者释放。

到这里，关于页级内存管理层接口函数的设计就完成了。图 7-6 和图 7-7 也展示了它们的调用流程。下面就得去设计页级内存管理层相关的数据结构了。

7.4.2 相关的数据结构

想要实现基于内存池思想的页级内存管理层，首先就得设计构造相关的一套数据结构，否则代码无从写起。暂时可以理解内存池其实就是一大块内存空间，然后按需要把它分成许多小块的内存空间。例如，把一块 128KB 大小的内存空间块（相当于内存池）分成 32 个 4KB 大小的页面，然后把另一块 128KB 大小的内存空间块（相当于内存池）分成 16 个 8KB 大小的内存空间块等。最后，要把这些内存池合理地组织起来进行管理。

基于内存池思想的页级内存管理层的实现，无非需要以下数据结构。

- ❑ 管理这个内存池（一块内存空间）的数据结构，即需要一个内存池管理头数据结构。
- ❑ 表示这个内存池（一块内存空间）中的一个小块（页）空间的状态，如它是已经分配了还是空闲的。
- ❑ 内存池肯定不止一个，所以需要有一个数据结构来存放组织那些内存池，方便 LMOSEM

内核查找。

以上这些数据结构，对于基于内存池思想的页级内存管理层是足够了，下面就去依次构造以上这些数据结构。

首先来构造一个内存池的管理头数据结构。该数据结构中需要的信息可能有链表、自旋锁、整个内存池空间的大小、内存池中小块（页）的个数、内存池中小块（页）的大小、已经分配的小块（页）个数、描述小块（页）状态数据结构的指针等。下面就把这些信息变成数据结构，如代码清单 7-24[xxxx/lmosem/include/knlinc/krlpagempol_t.h] 所示。

代码清单7-24 内存池管理头

```
typedef struct s_MPLHEAD
{
    spinlock_t      mh_lock;        //保护其自身的自旋锁
    list_h_t        mh_list;        //链表
    uint_t          mh_hedty;       //内存池管理头的类型
    adr_t           mh_start;       //内存池的开始地址
    adr_t           mh_end;         //内存池的结束地址
    adr_t           mh_firtfreadr;
    pglmap_t*       mh_firtpmap;    //指向第一个空闲页状态描述结构的指针
    uint_t          mh_objnr;       //池中对象（页）的个数
    uint_t          mh_aliobsz;
    uint_t          mh_objsz;      //池中对象（页）的大小
    uint_t          mh_nxtpsz;
    uint_t          mh_afindx;      //已经分配对象（页）的个数
    uint_t          mh_pmnr;        //页状态描述结构的个数
    pglmap_t*       mh_pmap;        //页状态描述结构的存放地址
}mplhead_t;
```

代码中的注释已经很清楚了，要注意的是 `mplhead_t` 结构被设计成多用途的，它既可以用于页级内存池又可以用于字级内存池，使用其中的 `mh_hadty` 域加于区分。所以其中有很多域是用于字级内存管理层的，在这里我们不必过多关注。

下面就来设计页级内存池中的页状态描述数据结构，它描述内存池中的页面是否已经分配或者是空闲的。它需要一个页面的首地址和指向自己的指针就行了，如代码清单 7-25 [xxxx/lmosem/include/knlinc/krlpagempol_t.h] 所示。

代码清单7-25 页状态描述符

```
typedef struct s_PGLMAP
{
    adr_t           pgl_start;      //保存内存池中一个页面的首地址
    struct s_PGLMAP* pgl_next;     //指向自身的一个指针
}pglmap_t;
```

`mplhead_t` 结构中的 `mh_firtpmap` 域始终指向下一个空闲的 `pglmap_t` 结构，所以分配时，始终是通过 `mh_firtpmap` 域获取一个 `pglmap_t` 结构，返回其中 `pgl_start` 域中的地址值，同时把该 `pglmap_t` 结构中的 `pgl_next` 域送给 `mh_firtpmap` 域，使它能指向下一个空闲的

pglmap_t 结构。

最后还要把众多的 mplhead_t 结构组织起来。因为随着 LMOSEM 内核的运行，肯定不只是建立一个内存池，再说它们本身也是动态建立的，更有可能它们不是同一类大小的内存池，有的可能是页级内存池，有的可能是字级内存池。所以要设计一种数据结构来组织它们，以便于查找。如代码清单 7-26[xxxx/lmosem/include/knlinc/krlpagempol_t.h] 所示。

代码清单7-26 内存池容器

```
typedef struct s_KMEMPOOL
{
    spinlock_t      mp_lock;           //保护其自身的自旋锁
    list_h_t        mp_list;          //链表
    uint_t          mp_stus;          //状态
    uint_t          mp_flg;           //标志
    spinlock_t      mp_pglock;        //保护页级内存池的自旋锁
    spinlock_t      mp_oblock;        //保护字级内存池的自旋锁
    uint_t          mp_pgmplnr;       //页级内存池的个数
    uint_t          mp_obmplnr;       //字级内存池的个数
    list_h_t        mp_pgmpmheadl;    //挂载页级内存池的链表头
    list_h_t        mp_obmpmheadl;    //挂载字级内存池的链表头
    mplhead_t*      mp_pgmpmhcach;    //保存当前操作过的页级mplhead_t结构
    mplhead_t*      mp_obmpmhcach;    //保存当前操作过的字级mplhead_t结构
} kmempool_t;
```

这个 kmempool_t 结构的实例变量包含在 LMOSEM 内核的可执行文件中，同样也是全局变量，如代码清单 7-27[xxxx/lmosem/kernel/krlglobal.c] 所示。和前面章节中的 halglobal.c 文件一样，在 LMOSEM 内核功能层中也有一个专门定义全局变量的 C 语言代码文件 krlglobal.c，其中的机理和前面章节中描述的一样。

代码清单7-27 定义全局的kmempool_t类型变量oskmempool

```
#define KRLGLOBAL_HEAD
#include "lmosemtypes.h"
#include "lmosemctrl.h"
KRL_DEFGLOB_VARIABLE(kmempool_t,oskmempool);
.....
```

如果现在你对 mplhead_t、pglmap_t 这两套数据结构的实例变量放在哪里，感到困惑的话，说明你一直在思考最核心的问题，这种问题还是画幅图比较容易弄清楚，如图 7-8 所示。

图 7-8 中，我们不难发现，mplhead_t、pglmap_t 这两套数据结构的实例变量都是动态建立的，建立内存池的代码会找块级内存管理层要一块内存，把这块内存的前 4KB（一个页面）的空间拿出来，存放 mplhead_t、pglmap_t 这两套数据结构的实例变量，mplhead_t 结构的空间在该 4KB 空间的最前面，这是固定的，就在 mplhead_t 结构其后存放 pglmap_t 结构，例如，要的是 128KB 的内存空间块，就要在 mplhead_t 结构其后建立 32 个 pglmap_t 结

构（对于 256KB 的内存空间块那就是 64 个 `pglmap_t` 结构，如果分成 8KB 的页面那就是 32 个 `pglmap_t` 结构，依此类推），最后形成了一个 `pglmap_t` 结构数组。你可能会发现用第一个 4KB 空间存放这些数据结构，有点浪费内存了。是的，这些数据结构有时只有 100 多个字节，如果读者有能力或者兴趣，也可以在最开始时要一块内存空间，专门用于存放这两套数据结构，即形成用于存放内存池管理数据结构的内存池。不过这将增加代码的复杂程度，不利于我们初学者探讨操作系统原理。

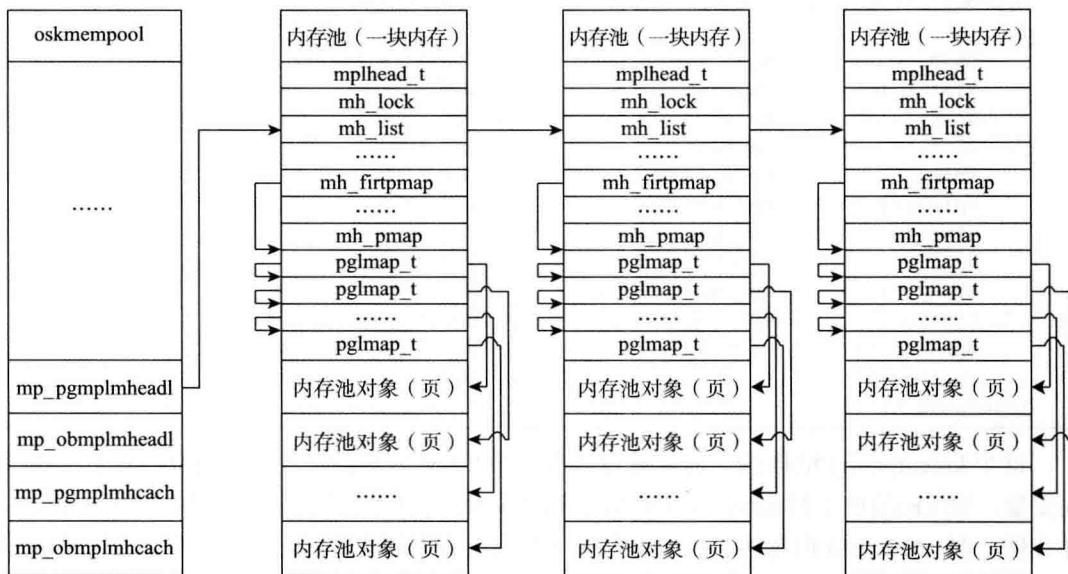


图 7-8 页级内存池管理数据结构视图

相应的数据结构也设计好了，并且也知道它们在内存中的组织形势了，下面就去从页级内存管理层的初始化代码开始写起，一步步实现页级内存管理层。

7.4.3 页级内存管理初始化

任何 LMOSEM 内核组件，或者其中一个小部件，都要从初始化开始，页级内存管理只是内存管理组件中的一个层，它也需要初始化，因为它使用了相关的数据结构。对于 LMOSEM 内核代码使用的数据结构，无论动态建立的数据结构实例变量还是静态定义的数据结构实例变量，在初始化之前是不能使用的。

页级内存池的 `mplhead_t`、`pglmap_t` 这两套数据结构的实例变量都是动态建立的。我们等建立它们时再进行初始化也不迟，但是 `kmempool_t` 结构的实例变量却是静态定义的，并且建立内存池之前要依赖它，所以在 LMOSEM 内核启动时就要初始化它。下面就去写两个函数，一个是页级内存管理层的主初始化函数 `init_krlpagempol`，另一个是初始化 `kmempool_t`

结构实例变量的函数 `kmempool_t_init`。如代码清单 7-28[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-28 页级内存管理初始化

```
void init_krlpagempol()
{
    kmempool_t_init(&oskmempool);
    return;
}

void kmempool_t_init(kmempool_t* initp)
{
    hal_spinlock_init(&initp->mp_lock);
    list_init(&initp->mp_list);
    initp->mp_stus=0;
    initp->mp_flg=0;
    hal_spinlock_init(&initp->mp_pglock);
    hal_spinlock_init(&initp->mp_oblock);
    initp->mp_pgemplnr=0;
    initp->mp_obmplnr=0;
    list_init(&initp->mp_pgemplheadl);
    list_init(&initp->mp_obmplheadl);
    initp->mp_pgemplmhcach=NULL;
    initp->mp_obmplmhcach=NULL;
    return;
}
```

代码清单 7-28 中的代码形式我们并不陌生，无非就是把一些自旋锁、链表结构实例初始化，再把结构中的一些域初始化为 0，这不必多说。

但是 `init_krlpagempol` 函数是谁调用的呢。其实对于 LMOSEM 内核而言，内存管理这个大组件才是它所感知的，而这个组件中的一个管理层的初始化，是由该组件的初始化函数调用的，内存管理组件的主初始化函数是 `init_krlmm`。它在 `krlmm.c` 文件中，如代码清单 7-29[xxxx/lmosem/kernel/krlmm.c] 所示。

代码清单7-29 内存管理组件初始化

```
void init_krlmm()
{
    init_krlpagempol();
    return;
}
```

`init_krlmm` 函数又是谁调用的呢，当然是 LMOSEM 内核功能层的入口函数 `init_krl`，如代码清单 7-30[xxxx/lmosem/kernel/krlinit.c] 所示。并且它最开始调用的就是内存管理组件的主初始化函数。

代码清单7-30 LMOSEM内核功能层初始化

```
void init_krl()
```

```

{
    init_krlmm();
    .....
    hal_sysdie("LMOSEM RUN!!");
    return;
}

```

页级内存管理层的初始化函数就写好了，只初始化了一个数据结构的实例变量，非常简单，不必过多在此浪费时间，我们还有更多的事要做呢。

7.4.4 分配主函数

关于页级内存管理层的接口，7.4.3 小节中已经实现了，也设计好了相应的数据结构，并且做了相关的初始化工作。现在不妨思考一下，基于页级内存池的页面分配步骤是什么。事实上，看懂前面那些数据结构的读者，已经明白了，大致如下。

1) 根据内存空间分配大小在 `kmempool_t` 结构中查找合适的 `mplhead_t` 结构，因为一个 `mplhead_t` 结构管理一个内存池。

2) 如果第一步中没有找到合适的 `mplhead_t` 结构，那就要新建一个合适的 `mplhead_t` 结构，也就是建立一个新的页级内存池。

3) 操作查找到的或者新建的 `mplhead_t` 结构，以致最后成功分配内存空间，即页面。

同样，我们也和前面章节中一样，把以上这些步骤写成一个一个的函数，并且让这些函数在一个主函数中被调用。下面就来写这个主函数 `kmempool_pages_core_new`，如代码清单 7-31[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-31 页级内存池的主分配函数

```

adr_t kmempool_pages_core_new(size_t msize)
{
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    adr_t retadr=NULL;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_pglock,&cpufg);
    .....
    return_step:
        hal_spinunlock_restflg_sti(&kmplp->mp_pglock,&cpufg);
        return retadr;
}

```

是不是有似曾相识的感觉？和块级内存主分配函数的形式是一样的，都是先初始化一些变量，然后加锁，在加锁状态中干一些事情，最后解锁返回。

先在此写好这个框架函数。下面我们就去一步步实现上面那些分配步骤的函数，然后让它们一起协同工作，就达到目的了。

7.4.5 分配时查找 `mplhead_t`

在 7.4.4 节中，已经写好了主分配函数，但它只是一个框架函数，好像没做什么实际的工作。本节就要开始干点实事了。

前面提到，页级内存管理层分配内存空间的第一步，就是根据其分配大小在 `kmempool_t` 结构中查找合适的 `mplhead_t` 结构，因为它管理一个内存池。这里就去写这样一个查找 `mplhead_t` 结构的函数 `kmemplpg_retn_mplhead`，如代码清单 7-32[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-32 查找相应的`mplhead_t`

```
mplhead_t* pagenew_mplhead_isok(mplhead_t* mhp, size_t msize)
{
    //mh_firtmap是否为NULL，为NULL即表示页面已经分配完了
    if (mhp->mh_firtmap==NULL)
        return NULL;
    //是否为页面内存池
    if (mhp->mh_hedty!=MPLHTY_PAGE)
        return NULL;
    //实际对象（页面）大小是否与请求大小相等
    if (mhp->mh_aliobsz!=msize)
        return NULL;
    //分配计数是否大于等于对象（页面）个数
    if (mhp->mh_afindx>=mhp->mh_objnr)
        return NULL;
    //以上测试通过就返回mhp，表示一个页面内存池是合乎分配要求的
    return mhp;
}

mplhead_t* kmemplpg_retn_mplhead(kmempool_t* kmplockp, size_t msize)
{
    mplhead_t* retmhp;
    list_h_t* list;
    if (kmplockp->mp_pgimplmhcach!=NULL)
    {
        retmhp=kmplockp->mp_pgimplmhcach;
        retmhp=pagenew_mplhead_isok(retmhp,msize);
        if (retmhp!=NULL)
        {
            return retmhp;
        }
    }
    list_for_each(list,&kmplockp->mp_pgimplmheadl)
    {
        retmhp=list_entry(list,mplhead_t,mh_list);
        retmhp=pagenew_mplhead_isok(retmhp,msize);
        if (retmhp!=NULL)
        {
            kmplockp->mp_pgimplmhcach=retmhp;
            return retmhp;
        }
    }
}
```

```

    }
}
return NULL;
}

```

代码清单 7-32 中有两个函数。先来看重要的 `kmemplpg_retn_mplhead` 函数。它首先检查 `kmempool_t` 结构中的 `mp_pgmplmhcach` 域，因为它里面保存的正是上次刚刚操作过的 `mplhead_t` 结构，检查它是否符合分配要求，如果它符合分配要求，就不用遍历其中的 `mp_pgmplmheadl` 链表了。如果 `mp_pgmplmhcach` 域中指向的 `mplhead_t` 结构中的内存空间对象已经分配完了或者与请求大小不符，就要用 `list_for_each` 宏遍历整个 `mp_pgmplmheadl` 链表了，每找到一个 `mplhead_t` 结构，就调用 `pagenew_mplhead_isok` 函数检查其 `mplhead_t` 结构是否符合要求。如果符合要求就在返回这个 `mplhead_t` 结构指针之前更新 `mp_pgmplmhcach` 域，以便下次分配时减少查找时间，否则继续遍历链表。如果遍历完了也没找到合适的 `mplhead_t` 结构，则返回 `NULL` 表示查找失败。`pagenew_mplhead_isok` 函数的工作主要是检查一个 `mplhead_t` 结构是否合适，代码中的注释已经很明显了，不必多说。

对于页级内存分配时，根据其分配大小在 `kmempool_t` 结构中查找合适的 `mplhead_t` 结构的工作，就做完了，非常简单。不妨紧接着进入下一节，看看如果没有找到合适的 `mplhead_t` 结构，即 `kmemplpg_retn_mplhead` 函数返回 `NULL` 时，该怎么办。

7.4.6 分配时新建页级内存池

如果是第一次请求页级内存分配，那么 7.4.5 节中的 `kmemplpg_retn_mplhead` 函数肯定返回 `NULL`，因为这时 `kmempool_t` 结构中没有任何内存池，即 `mplhead_t` 结构。如果这时就宣告分配失败，当然是不行的。这时需要做的是根据请求大小，新建一个内存池（`mplhead_t`），并把它挂入到 `kmempool_t` 结构中。

建立一个新内存池的步骤，无非就如下两步。

- ❑ 向块级内存管理层要一块合适的内存空间块。
- ❑ 在该块内存空间中建立相应的 `mplhead_t`、`pglmap_t` 数据结构。

下面将在一个主函数中，执行向块级内存管理层要一块合适的内存空间块的动作，然后在另一个函数中，执行建立相应的 `mplhead_t`、`pglmap_t` 数据结构的动作。

首先来写好这个主函数 `new_page_mpool`，如代码清单 7-33[xxxx/lmosem/kernel/krlpage-mpol.c] 所示。

代码清单7-33 新建页级内存池

```

mplhead_t* new_page_mpool(kmempool_t* kmplockp, size_t msize)
{
    mplhead_t* mphdp=NULL; size_t pgnr=msize>>12; adr_t blkadr=NULL;
    if(pgnr<1)
        return NULL;
    if(pgnr<=2)

```

```

{
    blkadr=hal_memallocblks(BLK128KB_SIZE);
    if(blkadr==NULL)
        return NULL;
    mphdp=page_mpool_init(kmplockp, (mplhead_t*)blkadr,msize,blkadr, (blkadr+B
        LK128KB_SIZE-1));
    if(mphdp==NULL)
        hal_sysdie("new pg mpool err 1");
    return mphdp;
}
if(pgnr<=4)
{
    blkadr=hal_memallocblks(BLK256KB_SIZE);
    if(blkadr==NULL)
        return NULL;
    mphdp=page_mpool_init(kmplockp, (mplhead_t*)blkadr,msize,blkadr, (blkadr+B
        LK256KB_SIZE-1));
    if(mphdp==NULL)
        hal_sysdie("new pg mpool err 2");
    return mphdp;
}
if(pgnr<=8)
{
    blkadr=hal_memallocblks(BLK256KB_SIZE);
    if(blkadr==NULL)
        return NULL;
    mphdp=page_mpool_init(kmplockp, (mplhead_t*)blkadr,msize,blkadr, (blkadr+B
        LK256KB_SIZE-1));
    if(mphdp==NULL)
        hal_sysdie("new pg mpool err 3");
    return mphdp;
}
if(pgnr<=16)
{
    blkadr=hal_memallocblks(BLK256KB_SIZE);
    if(blkadr==NULL)
        return NULL;
    mphdp=page_mpool_init(kmplockp, (mplhead_t*)blkadr,msize,blkadr, (blkadr+B
        LK256KB_SIZE-1));
    if(mphdp==NULL)
        hal_sysdie("new pg mpool err 4");
    return mphdp;
}
if(pgnr<=31)
{
    blkadr=hal_memallocblks(BLK128KB_SIZE);
    if(blkadr==NULL)
        return NULL;
    mphdp=page_mpool_init(kmplockp, (mplhead_t*)blkadr,msize,blkadr, (blkadr+B
        LK128KB_SIZE-1));
    if(mphdp==NULL)

```



```

        hal_sysdie("new pg mpool err 5");
    return mphdp;
}
return NULL;
}

```

代码清单 7-33 虽然很长，但是很简单，首先通过 `size_t pgnr=msize>>12`，把请求分配的大小转换成页面个数（每个页面大小为 4096B，所以右移 12 位）。然后根据页面个数调用块级内存管理层的接口函数，分配不同大小的内存空间块，如 1～2 个页面就分配 128KB 大小的内存空间块，3～8 个页面就分配 256KB 大小的内存空间块等。最后调用 `page_mpool_init` 函数，初始化并建立相关的数据结构。其中的死机函数 `hal_sysdie`，可以帮助我们捕捉错误。

分配了一块空闲的内存空间块，就要在这块内存空间中建立相应的 `mplhead_t`、`pglmap_t` 数据结构，`page_mpool_init` 函数就是完成这些事情的，如代码清单 7-34[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-34 初始化一个新的页级内存池

```

void mplhead_t_init(mplhead_t* initp)
{
    //初始化自旋锁
    hal_spinlock_init(&initp->mh_lock);
    //初始化链表
    list_init(&initp->mh_list);
    initp->mh_hedty=0;
    initp->mh_start=0;
    initp->mh_end=0;
    initp->mh_firtfreadr=NULL;
    initp->mh_firtpmap=NULL;
    initp->mh_objnr=0;
    initp->mh_alioobsz=0;
    initp->mh_objsz=0;
    initp->mh_nxtpsz=0;
    initp->mh_afindx=0;
    initp->mh_pmnr=0;
    //可以看出pglmap_t结构的开始地址正是紧跟mplhead_t结构其后的
    initp->mh_pmap=(pglmap_t*)((uint_t)initp+sizeof(mplhead_t));
    return;
}

mplhead_t* page_mpool_init(kmempool_t* kmplockp,mplhead_t* initp,size_t msize,adr_
    t start,adr_t end)
{
    if(((start&0xffff)!=0)||((end-start)<((PAGE_SIZE*2)-1)))
        return NULL;
    //因为内存空间块的第一页要存放mplhead_t、pglmap_t数据结构
    adr_t sadr=start+PAGE_SIZE;
    uint_t pi=0,pnr=0;
    //和其他数据结构初始化函数一样，是把数据结构本身初始化为默认状态

```

```

    mplhead_t_init(initp);
//把这个内存池的类型标记为页级内存池类型
    initp->mh_hedty=MPLHTY_PAGE;
//设置内存块（池）的开始地址
    initp->mh_start=start;
//设置内存块（池）的结束地址
    initp->mh_end=end;
//设置内存池实际分配对象的大小
    initp->mh_aliobsz=msize;
//内存池对象的大小，对于页级内存池，它和mh_aliobsz的值相同
    initp->mh_objsz=msize;
//每循环一次，就初始化一个pglmap_t数据结构
    for(;;)
    {
//如果(sadr+msize-1)大于end，则跳出循环
        if((sadr+msize-1)>end)
            break;
        initp->mh_pmap[pi].pgl_start=sadr;
        initp->mh_pmap[pi].pgl_next=&(initp->mh_pmap[pi+1]);
//sadr=sadr+msize循环一次就加上一个分配大小
        sadr+=msize;
        pi++;
    }
    if(pi>0)
    {
//设置最后一个pglmap_t结构的pgl_next域为NULL
        initp->mh_pmap[pi-1].pgl_next=NULL;
        pnr=pi;
//设置mh_firtmap指向第一个pglmap_t结构
        initp->mh_firtmap=&(initp->mh_pmap[0]);
        goto add_step;
    }
    initp->mh_pmap[pi].pgl_next=NULL;
    initp->mh_pmap[pi].pgl_start=NULL;
    pnr=pi;
add_step:
//一共有几个页面对象
    initp->mh_objnr=pnr;
//一共有几个pglmap_t数据结构
    initp->mh_pmr=pnr;
//把新内存池（mplhead_t）加入到kmempool_t结构的链表中
    list_add(&initp->mh_list,&kmplockp->mp_pgmptheadl);
//表示又建立了一个页面内存池
    kmplockp->mp_pgmplnr++;
    return initp;
}

```

page_mpool_init 函数的代码有点长，但大致可分为三段。第一段是初始化并建立 mplhead_t 结构。第二段是初始化并建立 pglmap_t 结构，然后把最后一个 pglmap_t 结构中的

pgl_next 域设为 NULL。第三段是根据初始化 pglmap_t 结构的情况填写一些 mplhead_t 结构中的信息，最后把这个内存池（mplhead_t 结构）加入到 kmempool_t 结构的 mp_pgimplheadl 链表中。其他代码，加上其中的注释应该很容易明白，不必多言。

新建一个页面内存池，我们用了两个函数就实现了，代码虽长，但并不复杂。好了，有了内存池，下面就要基于它分配实际的内存页面了。

7.4.7 分配时操作 mplhead_t

页级内存分配的第三步，就是操作查找到的或者新建的 mplhead_t 结构，以致最后成功分配到内存页面。

下面来写一个函数完成这个操作，由于我们的数据结构设计得很简单，所以这个函数写起来很容易，如代码清单 7-35[xxxx/lmosem/kernel/krtpagempol.c] 所示。

代码清单7-35 操作相应的mplhead_t

```

adr_t page_new_on_mplhead(mplhead_t* mplhdp)
{
    pglmap_t* map;
    if(mplhdp->mh_afindx>=mplhdp->mh_objnr)
        return NULL;
    if(mplhdp->mh_firtpmap!=NULL)
    {
        //把当前空闲的pglmap_t结构保存到map中
        map=mplhdp->mh_firtpmap;
        //指向下一个空闲的pglmap_t结构
        mplhdp->mh_firtpmap=map->pgl_next;
        //把当前空闲pglmap_t结构的pgl_next域置为NULL，以断掉链接
        map->pgl_next=NULL;
        //增加分配计数
        mplhdp->mh_afindx++;
        //返回当前空闲pglmap_t结构中pgl_start域的值，这正是空闲页面的首地址
        return map->pgl_start;
    }
    return NULL;
}

```

page_new_on_mplhead 函数首先检查 mplhead_t 结构中的分配计数，是不是大于等于 mplhead_t 结构中的对象个数。接着又检查 mplhead_t 结构中指向第一个空闲 pglmap_t 结构的指针 mh_firtpmap 域是不是 NULL。通过这两次判断就能确定 mplhead_t 结构中有没有空闲的页面对象。而分配的步骤就是从 mh_firtpmap 域中拿出空闲的 pglmap_t 结构指针，同时使 mh_firtpmap 域指向下一个空闲的 pglmap_t 结构，然后增加分配计数，最后返回空闲页面的首地址。

现在就去把前两步的代码和上面完成的代码组装在一起，完成整个页级内存管理的分配流程。既然那些函数已经写好了，把它们放在主分配函数中调用即可。如代码清单 7-36

[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-36 页级内存池的主分配函数

```

adr_t kmempool_pages_core_new(size_t msize)
{
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    adr_t retadr=NULL;
    kmempool_t* kmplp=&oskmempool;
    //对kmempool_t结构加锁
    hal_spinlock_saveflg_cli(&kmplp->mp_pglock,&cpufg);
    //查找相应的mplhead_t结构(内存池)
    mplhdp=kmemplpg_retn_mplhead(kmplp,msize);
    if(mplhdp==NULL)
    {
        //新建相应的mplhead_t结构(内存池)
        mplhdp=new_page_mpool(kmplp,msize);
        if(mplhdp==NULL)
        {
            retadr=NULL;
            goto return_step;
        }
    }
    //基于相应的mplhead_t结构(内存池)完成内存页面分配
    retadr=page_new_on_mplhead(mplhdp);
return_step:
    //对kmempool_t结构解锁
    hal_spinunlock_restflg_sti(&kmplp->mp_pglock,&cpufg);
    return retadr;
}

```

基于内存池思想的内存页面分配的整个过程,就到这里了。代码结构很清晰,只是新增三个数据结构。要注意的是页级内存管理在 LMOSEM 内核的功能层中。可是我们的代码写的对吗。现在就去做一个小小的测试,确定这部分代码没有问题之后,再继续做后面的事情。

7.4.8 分配代码写得对吗

页级内存管理的释放代码,我们还没有写,但是已经迫不及待地想要知道分配代码写得对不对。下面就去写相应的测试代码测试一下。

要确定页级内存管理的分配代码写得对不对,我们写一个用于测试分配内存页面的函数,然后进行内存页面分配,最后看一看分配的结果就可以知道了。下面就来写这个函数,如代码清单 7-37[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-37 测试页级内存池的分配

```

typedef struct s_adrsz
{

```

```

    adr_t adr; //存放分配内存空间的地址
    size_t sz; //存放分配内存空间的大小
} adrsz_t;
void testpgmpool()
{
    adrsz_t adsz[10];
    size_t alcsz=0x1000;
    adsz[0].sz=alcsz;
    adsz[0].adr=kmempool_new(alcsz);
    adsz[1].sz=alcsz;
    adsz[1].adr=kmempool_new(alcsz);
    alcsz=0x1500;
    adsz[2].sz=alcsz;
    adsz[2].adr=kmempool_new(alcsz);
    adsz[3].sz=alcsz;
    adsz[3].adr=kmempool_new(alcsz);
    alcsz=0x3000;
    adsz[4].sz=alcsz;
    adsz[4].adr=kmempool_new(alcsz);
    adsz[5].sz=alcsz;
    adsz[5].adr=kmempool_new(alcsz);
    alcsz=0x3200;
    adsz[6].sz=alcsz;
    adsz[6].adr=kmempool_new(alcsz);
    adsz[7].sz=alcsz;
    adsz[7].adr=kmempool_new(alcsz);
    alcsz=0x7000;
    adsz[8].sz=alcsz;
    adsz[8].adr=kmempool_new(alcsz);
    adsz[9].sz=alcsz;
    adsz[9].adr=kmempool_new(alcsz);
    for(int i=0;i<10;i++)
    {
        printfk("adsz[%x] sz:%x adr:%x\n\r",i,adsz[i].sz,adsz[i].adr);
    }
    mplhead_t* retmhp;
    list_h_t* list;
    list_for_each(list,&oskmempool.mp_pgmplmheadl)
    {
        retmhp=list_entry(list,mplhead_t,mh_list);
        printfk("mph_t_adr:%x mph_t.mh_end:%x mph_t.mh_objsz:%x mph_t.mh_objnr:%x\n\r", retmhp,retmhp->mh_end,retmhp->mh_objsz,retmhp->mh_objnr);
    }
    return;
}

```

代码很长，却很简单，开始是每种分配大小都分配两次，因为要观察分配结果是否正确，例如，比较两个返回地址之间的区间是否合理。接着，打印了所有分配的结果。最后，遍历了 kmempool_t 结构的 mp_pgmplmheadl 链表中的所有 mplhead_t 结构，每遍历到一个

mplhead_t 结构就打印出它的一些信息, 让我们可以观察到它们是否合理。

下面把 testpgmpool 函数加入到 init_krlpagempool 函数的最后, 注意要把先前测试块级内存的 test_blkmm 函数从 init_hal 函数中注释掉, 把 init_krlmm 函数加入到 LMOSEM 内核功能层的入口函数 init_krl 的开始处。最后我们打开 Linux 终端, 切换到 lmosem 目录下 make 一下, 连接好开发板, 打开另一个终端输入 sudo minicom, 打开开发板的电源, 在刚才的终端中输入 d, 接着在先前的终端中输入 make knlexc, 就会将编译好的 LMOSEM 内核文件下载到开发板中运行。正常情况下, 会出现如图 7-9 所示的情况。

```

深度终端
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30008000h,TOTAL:38574]
RECEIVED FILE SIZE: 38574 (37KB/S, 1S)
Downloaded file at 0x30000000, size = 38564 bytes
adsz[0] sz:1000 adr:30021000
adsz[1] sz:1000 adr:30022000
adsz[2] sz:1500 adr:30041000
adsz[3] sz:1500 adr:30043000
adsz[4] sz:3000 adr:30401000
adsz[5] sz:3000 adr:30404000
adsz[6] sz:3200 adr:30441000
adsz[7] sz:3200 adr:30445000
adsz[8] sz:7000 adr:30481000
adsz[9] sz:7000 adr:30488000
mph_t_adr:30480000 mph_t_mh_end:304bffff mph_t_mh_objsz:7000 mph_t_mh_objnr:9
mph_t_adr:30440000 mph_t_mh_end:3047ffff mph_t_mh_objsz:4000 mph_t_mh_objnr:f
mph_t_adr:30400000 mph_t_mh_end:3043ffff mph_t_mh_objsz:3000 mph_t_mh_objnr:15
mph_t_adr:30040000 mph_t_mh_end:3005ffff mph_t_mh_objsz:2000 mph_t_mh_objnr:f
mph_t_adr:30020000 mph_t_mh_end:3003ffff mph_t_mh_objsz:1000 mph_t_mh_objnr:1f
LMOSEM SYSTEM IS DIE LMOSEM RUN!!
  
```

图 7-9 测试页级内存管理的分配代码

图 7-9 中显示的数据, 证明我们的代码没有问题, 可能发现上面的分配大小和 mplhead_t 结构的 mh_objsz 域中的值不同, 例如, 分配大小为 0x1500, 而在 mplhead_t 结构的 mh_objsz 域中的值却为 0x2000。这是因为开始分配之前就进行了页面对齐操作。

页级内存管理层的分配代码就写好了, 下面就去实现页级内存管理层的释放代码吧。

7.4.9 释放主函数

页级内存管理层的释放接口函数, 在前面已经实现了, 可是释放接口函数就是对传递进来的参数进行合理化检查, 不能达到真正释放内存页面的目的。真正释放内存页面的工作才正式开始。而基于内存池思想的内存页面的释放操作过程如下。

- ❑ 根据相应内存页面的释放地址和释放大小, 在 kmempool_t 结构中查找相应的 mplhead_t 结构。
- ❑ 操作第一步中查找到的 mplhead_t 结构, 达到释放内存页面的目的。
- ❑ 如果当前操作的 mplhead_t 结构管理的页面已经全部空闲了, 就要删除这个 mpl-

head_t 结构，最后通知块级内存管理层彻底释放对应的物理内存空间块。

这里同样把上面这三大步骤写成三个函数，然后由一个主框架函数来调用它们，现在就来实现这个主框架函数，如代码清单 7-38[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-38 页级内存池的主释放函数

```
bool_t kmempool_pages_core_delete(adr_t fradr, size_t frsz)
{
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    bool_t rets=FALSE;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_pglock, &cpufg);
    .....
    return_step:
        hal_spinunlock_restflg_sti(&kmplp->mp_pglock, &cpufg);
        return rets;
}
```

kmempool_pages_core_delete 函数的形式，和前面的一些主框架函数的形式如出一辙。都是先定义了一些变量，然后在加锁的状态中做了一些事情，最后解锁返回一个状态值。这不必多说。

好了，主框架函数的代码就写好了。下面就要去实现上述内存页面释放的三大步骤了。

7.4.10 释放时查找 mplhead_t

7.4.9 节说得很清楚，内存页面释放的第一步，就是根据内存页面的释放地址和释放大小，去 kmempool_t 结构中查找相应的 mplhead_t 结构。因为要释放的某个内存页面就存在某个 mplhead_t 结构所管理的内存块中。

查找相应的 mplhead_t 结构，无非就是遍历 kmempool_t 结构中相应的链表，取出里面相应的 mplhead_t 结构，看看它里面的对象（页面）大小和将要释放页面的大小是否相同，相同时就比较页面释放地址是不是落在 mplhead_t 结构管理的内存块的区间中。明白了方法，就来写一写这个函数，如代码清单 7-39[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-39 查找相应的mplhead_t

```
mplhead_t* pagedel_mplhead_isok(mplhead_t* mhp, adr_t fradr, size_t msiz)
{
    //是否为页级内存池
    if(mhp->mh_hedty!=MPLHTY_PAGE)
        return NULL;
    //实际对象大小是否和页面释放大小相同
    if(mhp->mh_aliobsz!=msiz)
        return NULL;
    //内存池中的页面是否已经分配过了，为0即表示没分配过，没有分配当然不能释放
```

```

    if (mhp->mh_afindx==0)
        return NULL;
    //释放地址是不是落在这个内存池中了
    if (fradr < (mhp->mh_start+PAGE_SIZE) || (fradr+msize-1) > mhp->mh_end)
        return NULL;
    //以上都判断无误后就返回这个mplhead_t结构的指针
    return mhp;
}

mplhead_t* kmempldelpg_retn_mplhead(kmempool_t* kmpplockp, adr_t fradr, size_t
    msize)
{
    mplhead_t* retmhp;
    list_h_t* list;
    if (kmpplockp->mp_pgimplmhcach!=NULL)
    {
        retmhp=kmpplockp->mp_pgimplmhcach;
        retmhp=pagedel_mplhead_isok(retmhp, fradr, msize);
        if (retmhp!=NULL)
        {
            return retmhp;
        }
    }
    list_for_each(list, &kmpplockp->mp_pgimplmheadl)
    {
        retmhp=list_entry(list, mplhead_t, mh_list);
        retmhp=pagedel_mplhead_isok(retmhp, fradr, msize);
        if (retmhp!=NULL)
        {
            return retmhp;
        }
    }
    return NULL;
}

```

函数 `kmempldelpg_retn_mplhead` 中依然是首先判断 `kmempool_t` 结构中的 `mp_pgimplmhcach` 域，因为它里面保存的是最近查找并操作过的 `mplhead_t` 结构，所以很可能它就是要找的。如果不是才去遍历 `mp_pgimplmheadl` 链表。每遍历一个 `mplhead_t` 结构就调用 `pagedel_mplhead_isok` 函数来确认这个 `mplhead_t` 结构是不是要找的。`pagedel_mplhead_isok` 函数中的注释已经写得很明白了，不必多言。

与内存页面的分配操作不同，`kmempldelpg_retn_mplhead` 函数如果没有找到相应的 `mplhead_t` 结构，那肯定是内存页面的释放地址和释放大小有问题了，所以只能中止内存页面的释放操作。

如果找到了相应的 `mplhead_t` 结构，下面的工作，就是操作相应的 `mplhead_t` 结构以至于最后释放内存页面。

7.4.11 释放时操作 `mplhead_t`

假如在 7.4.10 节中，找到了 `mplhead_t` 结构（通常情况下会找到），那么根据 7.4.9 节的基于内存池的内存页面的释放步骤：操作第一步中查找到的 `mplhead_t` 结构，达到释放内存页面的目的。现在就来操作这个找到的 `mplhead_t` 结构。

怎么操作这个 `mplhead_t` 结构才能达到释放内存页面的目的呢。当然是首先进行一些例行检查，如果发现有不对的地方，则立马返回错误。接着是搜索 `mplhead_t` 结构中的 `pglmap_t` 结构，看看内存页面的释放地址属于哪个 `pglmap_t` 结构。最后把这个找到的 `pglmap_t` 结构加入以 `mplhead_t` 结构中的 `mh_firtppmap` 域为首的 `pglmap_t` 结构链表中，减少 `mplhead_t` 结构的分配计数。下面就去实现这个函数。如代码清单 7-40[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-40 操作相应的`mplhead_t`

```
bool_t page_delete_on_mplhead(mplhead_t* mplhdp, adr_t fradr)
{
    pglmap_t * map;
    //进行例行检查, mh_afindx<1表示没有分配过, 所以更谈不上释放了
    //mh_pmn<1则表示没有pglmap_t结构
    if (mplhdp->mh_afindx<1 || mplhdp->mh_pmn<1)
    {
        return FALSE;
    }
    //搜索所有的pglmap_t结构, 看看哪个pglmap_t结构的pgl_start域中的值
    //等于释放地址, 同时让其pgl_next域等于NULL
    //然后把这个pglmap_t结构的指针保存在map中
    for (uint_t i=0; i<mplhdp->mh_pmn; i++)
    {
        if (fradr==mplhdp->mh_pmap[i].pgl_start)
        {
            map=&mplhdp->mh_pmap[i];
            if (map->pgl_next!=NULL)
            {
                return FALSE;
            }
            goto del_step;
        }
    }
    //如果没找到这个pglmap_t结构, 返回错误
    return FALSE;
del_step:
    //把mplhdp->mh_firtppmap域中的指针放进map->pgl_next中
    map->pgl_next=mplhdp->mh_firtppmap;
    //然后把map放进mplhdp->mh_firtppmap中, “入链”
    mplhdp->mh_firtppmap=map;
    //减少分配计数, 表示又释放了一个对象(内存页面)
    mplhdp->mh_afindx--;
```

```

    return TRUE;
}

```

代码清单 7-40 中的注释已经很明确了，不过这里需要搜索所有的 `pglmap_t` 结构，按照我们的设计，一般情况下最多也就三十几个 `pglmap_t` 结构，性能不是最好的，但也坏不到哪儿去。何况以我们现在的实力，先得把原理搞清楚，然后才能提升性能。

是不是内存页面的释放流程到这里就结束了呢，当然不是，因为这是对内存池的操作，也就是说只是在相应的内存池中释放了一个对象（页面）而已，还要看看整个内存池是不是空闲了，如果它空闲了，连它也要删除。下面就去写一写删除相应内存池的代码吧。

7.4.12 释放时删除页级内存池

如果一个页级内存池里面所有的对象（页面）都已经释放了，就说明这个内存池空闲了，没有其他程序使用了，但它又不能用于其他用途。这就会导致有很多空闲的内存池，但物理内存空间却越来越少了。因为物理内存空间都被一个个空闲的内存池占用着。

怎么解决这种问题呢，当然是发现一个内存池空闲后就删除它，把它占有的那一块内存空间也释放掉。所以释放一个页面后，还没完，还要看看这个内存池是不是空闲了，如果空闲了就要把它也删除。有些读者可能会说，这里删除了下次分配时又要新建，所以性能会下降，能不能等到 LMOSEM 内核发现内存空间紧缺时，再来删除那些内存池。是的，这种想法很好，可是这个方案要留给读者以后实现了，而现在就只能实现这种简单的方案，如代码清单 7-41[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-41 删除页级内存池

```

bool_t del_page_mpool(kmempool_t* kmplockp, mplhead_t* mphdp)
{
    //内存池非空闲，所以不能继续删除这个内存池了
    if(mphdp->mh_aindx>0)
    {
        return TRUE;
    }
    //计算内存池所占用的这块内存空间的大小
    size_t frsz=mphdp->mh_end-mphdp->mh_start+1;
    //把这个内存池的开始地址，作为释放地址
    adr_t fradr=mphdp->mh_start;
    //把这个内存池从kmempool_t结构中删除，即“断链”
    list_del(&mphdp->mh_list);
    //检查页级内存池的个数，如果小于1，那么内核代码肯定出了大问题，只能死机
    if(kmplockp->mp_pgemplnr<1)
    {
        hal_sysdie("del_page_mpool kmplockp->mp_pgemplnr<1");
    }
    //减少页级内存池的个数，因为已经要删除一个了
    kmplockp->mp_pgemplnr--;
    if(kmplockp->mp_pgemplmhcach==mphdp)

```

```

    {
        kmplockp->mp_pgmplmhcach=NULL;
    }
    //如果块级内存释放失败,也只能死机
    if(hal_memfreeblks(fradr,frsz)==FALSE)
    {
        hal_sysdie("del_page_mpool hal_memfreeblks err");
    }
    return TRUE;
}

```

代码清单 7-41 中,开始时对相应的内存池是否空闲进行了检查。不是空闲时要返回真,表示函数运行成功,但执行相关操作失败时就死机。然后就是该函数代码中间的部分,如果发现 kmempool_t 结构的 mp_pgmplmhcach 域中的值和正在删除的 mplhead_t 结构的地址相同,一定要把 mp_pgmplmhcach 域置为 NULL,看看前面的查找 mplhead_t 结构的代码就明白了。

好了,我们写完了所有释放内存页面步骤的函数,现在就要把它们一个个加入主框架函数中,使之协同完成整个释放内存页面的过程,如代码清单 7-42[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-42 页级内存池的主释放函数

```

bool_t kmempool_pages_core_delete(adr_t fradr,size_t frsz)
{
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    bool_t rets=FALSE;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_pglock,&cpufg);
    mplhdp=kmempldelpg_retn_mplhead(kmplp,fradr,frsz);
    //没有找到相应的mplhead_t结构时,则返回假表示释放失败
    if(mplhdp==NULL)
    {
        rets=FALSE;
        goto return_step;
    }
    //释放内存页面失败时,返回假
    if(page_delete_on_mplhead(mplhdp,fradr)==FALSE)
    {
        rets=FALSE;
        goto return_step;
    }
    //删除内存池失败时,也返回假
    if(del_page_mpool(kmplp,mplhdp)==FALSE)
    {
        rets=FALSE;
        goto return_step;
    }
}

```

```

    //如果代码安全运行到这里,就表示释放成功
    rets=TRUE;
return_step:
    hal_spinunlock_restflg_sti(&kmplp->mp_pglock,&cpufg);
    return rets;
}

```

到这里,页级内存管理层的分配和释放代码就写完了。但是没有结束,还要对页级内存管理层做全面的测试,看看相应的代码有没有问题。下面就去测试一下!

7.4.13 测试页级内存管理层

到这里还有最后一步,就是要确定我们的页级内存管理层是不是实现得正确无误,虽然前面已经测试了相关的分配代码,但是释放代码写好了之后,就可以做一个整体测试了。这次测试比前面的更为严格,所以测试代码也多了许多。

下面就来完成这几个测试函数,如代码清单 7-43[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-43 测试页级内存管理

```

void cmp_adrsz(adrsz_t* assp,uint_t nr)
{
    for(uint_t i=0;i<nr;i++)
    {
        for(uint_t j=0;j<nr;j++)
        {
            if(i!=j)
            {
                //比较每个分配地址是否和其他分配地址相等
                if(assp[i].adr==assp[j].adr)
                {
                    hal_sysdie("cmp adr start err");
                }
            }
        }
    }
    for(uint_t k=0;k<nr;k++)
    {
        for(uint_t h=0;h<nr;h++)
        {
            if(k!=h)
            {
                //比较每个地址区间是否和其他地址区间相等
                if((assp[k].adr+assp[k].sz)==(assp[h].adr+assp[h].sz))
                {
                    hal_sysdie("cmp adr end err");
                }
            }
        }
    }
}

```

```

    for(uint_t l=0;l<nr;l++)
    {
        for(uint_t m=0;m<nr;m++)
        {
            if(l!=m)
            {
                //比较每个地址区间是否和其他地址区间相交
                if((assp[l].adr>=(assp[m].adr)) && ((assp[l].adr+assp[l].sz)<=(assp[m].adr+assp[m].sz)))
                {
                    hal_sysdie("cmp adr in err");
                }
            }
        }
    }
    return;
}

void testpagemgr()
{
    adrsz_t adsz[30];
    size_t alcsz=0x1000;
    for(;alcsz<0x19000;alcsz+=0x1000)
    {
        //同一种分配大小, 分配30次
        for(int i=0;i<30;i++)
        {
            adsz[i].sz=alcsz;
            adsz[i].adr=kmempool_new(alcsz);
            if(adsz[i].adr==NULL)
            {
                hal_sysdie("testpagemgr kmempool_new err");
            }
            printfk("adsz[%x] sz:%x adr:%x\n\r",i,adsz[i].sz,adsz[i].adr);
        }
        //比较adsz中的分配地址是否正确
        cmp_adrsz(adsz,30);
        //依次释放所有已经分配的页面
        for(int j=0;j<30;j++)
        {
            if(kmempool_delete(adsz[j].adr,adsz[j].sz)==FALSE)
            {
                hal_sysdie("testpagemgr kmempool_delete err");
            }
            printfk("delete adsz[%x] sz:%x adr:%x\n\r",j,adsz[j].sz,adsz[j].adr);
        }
    }
    printfk("oskmempool.mp_pgmplnr:%x\n\r",oskmempool.mp_pgmplnr);
    return;
}

```

代码虽长,但是很好理解,首先是 testpagemgr 函数,它其中的外层循环控制分配大小,

循环内部的两个内层循环分别控制分配和释放操作，中间调用 `cmp_adrsz` 函数校验分配的所有页面是否正确。校验方法在 `cmp_adrsz` 函数中的注释已经说明了。如果 `cmp_adrsz` 函数成功返回，则说明正确无误。最后去看看 `oskmempool.mp_pgmplnr` 中的值是不是为 0，因为分配次数和释放次数都是对等的，所以它的值应该回到初始状态，也就是应该为 0。

现在连接好开发板，编译好 LMOSEM 内核，注意要用 `testpagemgr` 函数替换掉 `init_krlpagempol` 函数中的 `testpgmpool` 函数，因为现在已经不需要它了。下载 LMOSEM 内核文件到开发板中运行，方法前面已经说过多次了。正常情况下应该会出现和笔者相同的情况，如图 7-10 所示。

```

深度终端
delete adsz[9] sz:18000 adr:30141000
delete adsz[a] sz:18000 adr:30161000
delete adsz[b] sz:18000 adr:30181000
delete adsz[c] sz:18000 adr:301a1000
delete adsz[d] sz:18000 adr:301c1000
delete adsz[e] sz:18000 adr:301e1000
delete adsz[f] sz:18000 adr:30201000
delete adsz[10] sz:18000 adr:30221000
delete adsz[11] sz:18000 adr:30241000
delete adsz[12] sz:18000 adr:30261000
delete adsz[13] sz:18000 adr:30281000
delete adsz[14] sz:18000 adr:302a1000
delete adsz[15] sz:18000 adr:302c1000
delete adsz[16] sz:18000 adr:302e1000
delete adsz[17] sz:18000 adr:30301000
delete adsz[18] sz:18000 adr:30321000
delete adsz[19] sz:18000 adr:30341000
delete adsz[1a] sz:18000 adr:30361000
delete adsz[1b] sz:18000 adr:30381000
delete adsz[1c] sz:18000 adr:303a1000
delete adsz[1d] sz:18000 adr:303c1000
oskmempool.mp_pgmplnr:0
LMOSEM SYSTEM IS DIE LMOSEM RUN!

```

图 7-10 测试页级内存管理层

页级内存管理层终于实现了，但是对于 LMOSEM 内核的整个内存管理组件还没有完，不过也只剩下最后一层了。不用着急，可以休息一下，然后继续探索。

7.5 字级内存管理

字级内存管理位于内存管理组件的最高层。在前面的内存管理结构视图中已经展示了这一点，它依赖于它下面的页级内存管理层和块级内存管理层。

后面就会发现，LMOSEM 内核自身许多的数据结构的实例变量都是动态建立的，例如，进程数据结构的实例变量，这些数据结构的实例变量在内存中一般占用 30 ~ 100 个字节的空间。如果为此分配一个页面（4KB）那就太过浪费了，所以要想办法解决这种操作系统内核自身频繁使用小块内存空间的问题。

字级内存管理层就是应对这些小于 4KB 大小的内存空间分配请求的，字级内存管理层还是同样基于内存池思想，只不过其中的对象不再是页面，而是比页面更小的内存空间对象。下面依然从字级内存管理层的接口开始，然后是相关的数据结构，到最后的字级内存分

配、释放的整个流程。

7.5.1 字级内存接口及调用流程

一切还是从接口开始吧，前面已经说了，字级内存管理接口和页级内存管理接口相同，但它们从哪里开始不同呢，答案就在 `kmempool_onsize_new` 和 `kmempool_onsize_delete` 函数中。这两个函数中都将内存空间的分配、释放大小和 `OBJSORPAGE` 进行比较，`OBJSORPAGE` 这个宏定义在 `krlpagempol.t.h` 文件中：`#define OBJSORPAGE 2048`。只要内存空间的分配、释放大小小于这个值，就会调用字级内存管理的相关函数。

下面来看看代码，实际上在前面的小节中已经写好了。如代码清单 7-44[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-44 内存管理接口

```

adr_t kmempool_onsize_new(size_t msize)
{
    if(msize>OBJSORPAGE)
        return kmempool_pages_new(msize);
    return kmempool_objsz_new(msize);
}
bool_t kmempool_onsize_delete(adr_t fradr,size_t frsz)
{
    if(frsz>OBJSORPAGE)
        return kmempool_pages_delete(fradr,frsz);
    return kmempool_objsz_delete(fradr,frsz);
}
adr_t kmempool_new(size_t msize)
{
    if(msize<KMEMPALCSZ_MIN||msize>KMEMPALCSZ_MAX)
        return NULL;
    return kmempool_onsize_new(msize);
}
bool_t kmempool_delete(adr_t fradr,size_t frsz)
{
    if(fradr==NULL||frsz<KMEMPALCSZ_MIN||frsz>KMEMPALCSZ_MAX)
        return FALSE;
    return kmempool_onsize_delete(fradr,frsz);
}

```

代码很简单，要注意的是，如果内存空间的分配、释放大小大于 2048 字节，就会转而调用页级内存管理的相关函数。但是上面的代码只是接口，我们还要搞清楚，它们的代码控制路径是怎么传递到字级内存管理的核心函数的。看一幅图就明白了，如图 7-11 所示。

图 7-11 中清楚地展示了字级内存管理层的接口到其核心函数的调用流程，也清楚地看到 `kmempool_objsz_core_new`、`kmempool_objsz_core_delete` 函数，就是字级内存管理层的分配、释放操作的主核心函数。

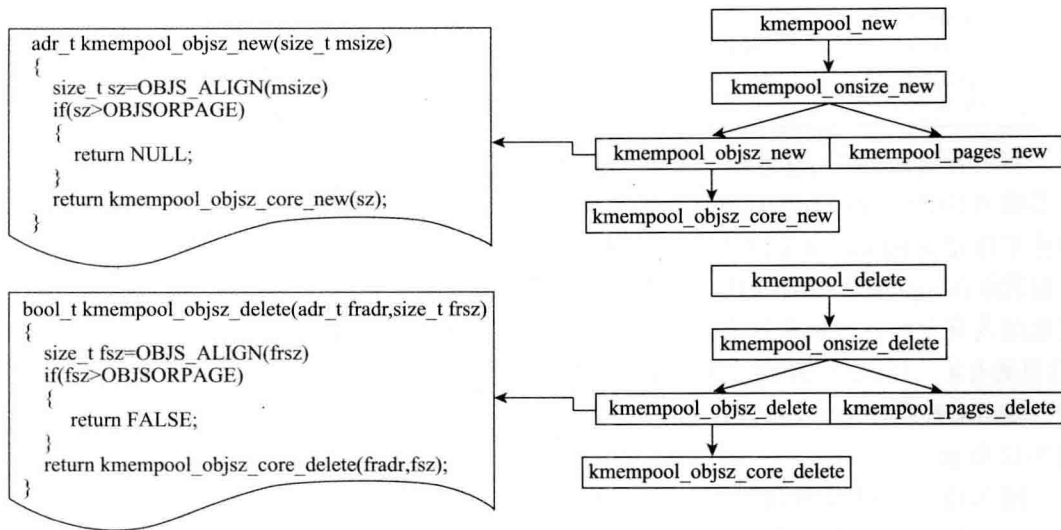


图 7-11 字级内存管理层主分配、释放函数的调用路径

清楚了字级内存管理层的分配接口和释放接口到其核心函数的调用流程，下面就去研究相关的数据结构。

7.5.2 相关的数据结构

操作系统内核里的大部分机制的实现，都需要有相关的数据结构。这里的字级内存管理层也是一样。

字级内存管理层和页级内存管理层一样，都是基于内存池的思想，所以它也要有链表、自旋锁、整个内存池空间的大小、对象个数、已分配对象的个数、描述对象状态的指针等。这不是和前面的页级内存池一样吗，所以我们使用了同一个数据结构 `mplhead_t`，但是使用其中不同的域。如代码清单 7-45[xxxx/lmosem/include/knlinc/krlpagempol_t.h] 所示。

代码清单7-45 内存池管理头

```

typedef struct s_MPLHEAD
{
    spinlock_t      mh_lock;           //保护其自身的自旋锁
    list_h_t        mh_list;           //链表
    uint_t          mh_hedty;          //内存池的类型，这里是MPLHTY_OBJS，值为2
    adr_t           mh_start;          //内存池的开始地址
    adr_t           mh_end;            //内存池的结束地址
    adr_t           mh_firtfreadr;     //指向第一个空闲对象的地址
    pglmap_t*       mh_firtpmap;       //字级内存池中不使用
    uint_t          mh_objnr;          //对象个数
    uint_t          mh_aliobsz;        //实际分配的对象大小，有时分配空间比请求的大
    uint_t          mh_objsz;          //对象大小
    uint_t          mh_nxtpsz;         //确定下一个对象指针的偏移

```



```

uint_t      mh_afindx;      //已经分配的对象个数
uint_t      mh_pmnrr;      //字级内存池中不使用
pglmap_t*   mh_pmap;       //字级内存池中不使用
}mplhead_t;

```

可以看到，字级内存池和页级内存池的数据结构一样，它是通过其中的 `mh_hedty` 域来区分的。而创建字级内存池时也不用建立 `pglmap_t` 数据结构的实例变量了，内存空间对象就放在 `mplhead_t` 结构的后面。同样也是首先分配几个页面的内存空间，然后在这个内存空间中存放具体的对象（这里的对象，指 32 ~ 2048 字节大小的内存空间）。下面通过一幅图来看看对象在字级内存池中到底是怎么存放的，如图 7-12 所示。

图 7-12 中，可以看到每个内存池中的对象（内存空间）的大小，可能不同，但是同一个内存池中的对象大小都是相同的。不难发现，在每个对象空间的后面还有一个指针空间，称为下一个对象的指针，就是通过这个指针把所有的对象链接起来的，最后一个对象的指针是 NULL，而 `mplhead_t` 结构中的 `mh_firtfreadr` 域始终指向第一个空闲的对象。

在 `mplhead_t` 结构中，为什么有两个作用差不多的域 `mh_alioobsz` 和 `mh_objsz`。那是因为要处理内存地址对齐的问题，有许多计算平台（特别是 RISC 体系）对内存访问有强制要求：访问内存地址要 4 字节对齐，或者 8 字节对齐。例如，请求分配 61 字节大小的内存空间，那么字级内存管理层会实际分配 64 字节，64 就会保存在 `mh_alioobsz` 域中，而 61 会保存在 `mh_objsz` 域中。

不同的内存空间分配大小需要建立不同的内存池（`mplhead_t` 结构），那么这些 `mplhead_t` 结构是如何组织的呢。当然还是放在 `kmempool_t` 结构中的，不过它和页级内存池不是放在一起的，它则是挂在 `mp_obmplmheadl` 链表中的，如图 7-13 所示。

明白了图 7-12 和图 7-13，就可以明白整个字级内存池的精髓，剩下的只是写代码实现了。下面就去编码吧。

7.5.3 分配主函数

这次不用初始化，因为我们沿用了和页级内存管理层一样的数据结构，所以这里就可以

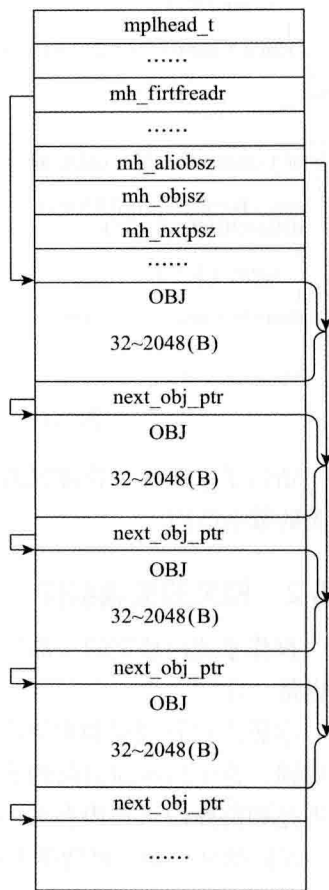


图 7-12 字级内存池中对象的存放


```

    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_oblock,&cpufg);
    .....
return_step:
    hal_spinunlock_restflg_sti(&kmplp->mp_oblock,&cpufg);
    return retadr;
}

```

和前面代码的形式一样，都是先定义好一些变量，然后加锁，在加锁中做了一些事，最后解锁返回。不过要注意的是，这里加的是 `kmempool_t` 结构中的 `mp_oblock` 锁。这把锁是专门管理字级内存池的。

下面就去编写实现上述三大步骤的函数吧。

7.5.4 分配时查找 `mplhead_t`

7.5.3 节中字级内存分配的第一步，就是根据内存空间的分配大小，在 `kmempool_t` 结构的 `mp_obmplmheadl` 链表中，查找合适的 `mplhead_t` 结构。因为分配的内存空间（内存池对象）是放在字级内存池中的，所以只有找到合适的 `mplhead_t` 结构，才能继续下面的工作。

下面来实现这个查找函数，如代码清单 7-47[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-47 查找相应的 `mplhead_t`

```

mplhead_t* objsnew_mplhead_isok(mplhead_t* mhp,size_t msize)
{
    //有没有内存池对象（即内存空间）
    if(mhp->mh_firtfreadr==NULL)
        return NULL;
    //是否为字级内存池类型
    if(mhp->mh_hedty!=MPLHTY_OBJS)
        return NULL;
    //请求分配的大小是否与对象大小相同
    if(mhp->mh_aliobsz!=msize)
        return NULL;
    //内存池中的对象是否已经分配完
    if(mhp->mh_afindx>=mhp->mh_objnr)
        return NULL;
    //若上面的检测都通过，就返回这个mplhead_t结构的指针
    return mhp;
}

mplhead_t* kmemplob_retn_mplhead(kmempool_t* kmplockp,size_t msize)
{
    mplhead_t* retmhp;
    list_h_t* list;
    if(kmplockp->mp_obmplmhead!=NULL)

```

```

{
    retmhp=kmplockp->mp_obmplmhcach;
    retmhp=objsnew_mplhead_isok(retmhp,msize);
    if(retmhp!=NULL)
    {
        return retmhp;
    }
}
list_for_each(list,&kmplockp->mp_obmplmheadl)
{
    retmhp=list_entry(list,mplhead_t,mh_list);
    retmhp=objsnew_mplhead_isok(retmhp,msize);
    if(retmhp!=NULL)
    {
        kmplockp->mp_obmplmhcach=retmhp;
        return retmhp;
    }
}
return NULL;
}

```

代码清单 7-47 中的 `kmemplob_retn_mplhead` 函数首先检查 `kmempool_t` 结构中的 `mp_obmplmhcach` 域，因为它里面保存的是上次刚刚操作过的 `mplhead_t` 结构的指针，如果它符合要求，就不用遍历 `mp_obmplmheadl` 链表了。否则，就要用 `list_for_each` 宏去遍历整个 `mp_obmplmheadl` 链表了，遍历过程中每找到一个 `mplhead_t` 结构，就调用 `objsnew_mplhead_isok` 函数检查其 `mplhead_t` 结构是否符合要求。如果符合要求就在返回这个 `mplhead_t` 结构指针之前更新 `mp_obmplmhcach` 域，以便下次分配时减少其查找时间。如果遍历完该链表也没找到符合要求的 `mplhead_t` 结构，就返回 `NULL` 表示查找失败。

当 `kmemplob_retn_mplhead` 函数返回 `NULL` 之后，该怎么办呢？下面我们继续。

7.5.5 分配时新建字级内存池

当一个字级内存池中所有的内存空间已经分配完时，又或者第一次通过字级内存管理层分配内存空间时，`kmemplob_retn_mplhead` 函数肯定会返回 `NULL`，这时就需要新建一个字级内存池，而不是宣告内存空间分配失败。

现在就来实现这个新建字级内存池的函数。该函数同样是先找其下级内存管理层，获取一块内存空间，然后在这块内存空间中建立内存池对象（空间块）和相关的数据结构。如代码清单 7-48[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-48 新建一个字级内存池

```

mplhead_t* new_objs_mpool(kmempool_t* kmplockp,size_t msize)
{
    mplhead_t* mphdp=NULL;
    adr_t pgadr=kmempool_pages_new(0x4000);

```

```

    if (pgadr==NULL)
    {
        return NULL;
    }
    mphdp=(mplhead_t*)pgadr;
    mphdp=objs_mpool_init(kmplockp,mphdp,msize,pgadr,(pgadr+0x4000-1));
    if (mphdp==NULL)
    {
        hal_sysdie("new objs mpool err");
    }
    return mphdp;
}

```

代码清单 7-48 中的 `new_objs_mpool` 函数只是找页级内存管理层要了 4 个页面大小的连续内存空间。如果要不到，就立即返回 `NULL`，不继续往下运行了。而真正建立字级内存池的函数是 `objs_mpool_init`，如代码清单 7-49[xxxx/lmosem/kernel/kr1pagempol.c] 所示。

代码清单7-49 初始化字级内存池

```

mplhead_t* objs_mpool_init(kmempool_t* kmplockp,mplhead_t* initp,size_t msize,adr_t
start,adr_t end)
{
    //对参数进行合理化检查
    if(((start&0xfff)!=0)||((end-start)<(PAGE_SIZE-1)))
    {
        return NULL;
    }
    //计算第一个对象的开始地址
    adr_t sadr=start+sizeof(mplhead_t);
    //如果对象的开始地址没按4字节对齐，就停机。一般不会发生这种情况
    if((sadr&3)!=0)
    {
        hal_sysdie("objs_mpool_init:sadr not align dword");
        return NULL;
    }
    uint_t objnr=0;
    //对mplhead_t数据结构本身进行初始化，该函数前面已经写过了
    mplhead_t_init(initp);
    //设置内存池的类型为MPLHTY_OBJS（字级内存池）
    initp->mh_hedty=MPLHTY_OBJS;
    //设置内存池的开始、结束地址
    initp->mh_start=start;
    initp->mh_end=end;
    //设置第一个内存池对象的地址
    initp->mh_firtfreadr=sadr;
    //设置内存池对象的大小
    initp->mh_aliobsz=msize;
    initp->mh_objsz=msize;
    //设置包含下一个对象指针的偏移量
    initp->mh_nxtpsz=msize+sizeof(adr_t);
}

```

```

    adr_t* nexp=NULL;
    adr_t adrsz=msize;
    adr_t npsz=initp->mh_nxtpsz;
    //如果包含下一个对象指针的偏移量没按4字节对齐,就停机
    if((initp->mh_nxtpsz&3)!=0)
    {
        hal_sysdie("objs_mpool_init:initp->mh_nxtpsz not align dword");
        return NULL;
    }
    for(;;)
    {
        //如果(sadr+npsz-1)大于内存池的结束地址就跳出循环
        if((sadr+npsz-1)>end)
        {
            break;
        }
        //计算存放下一个对象指针的地址
        nexp=(adr_t*)(sadr+adrsz);
        //把下一个对象指针的值放入这个地址的内存单元中
        *nexp=(sadr+npsz);
        //运行完这行代码,sadr中的值就是下一个对象的开始地址
        sadr+=npsz;
        //增加对象计数
        objnr++;
    }
    //如果对象计数为0,说明没初始化内存池对象,所以停机
    if(objnr==0)
    {
        hal_sysdie("objs_mpool_init:objnr err");
        return NULL;
    }
    sadr-=npsz;
    nexp=(adr_t*)(sadr+adrsz);
    //把最后一个对象后的下一个对象指针设为NULL,表示其后再无对象
    *nexp=NULL;
    //设置内存池对象的个数
    initp->mh_objnr=objnr;
    //把内存池加入到kmempool_t结构中(入链)并增加其计数
    list_add(&initp->mh_list,&kmplockp->mp_obmplmheadl);
    kmplockp->mp_obmplnr++;
    //返回
    return initp;
}

```

objs_mpool_init 函数的代码,也可以分为三个部分。第一部分是对参数进行合理性检查,然后初始化 mplhead_t 数据结构本身。第二部分是一个循环,在循环中建立一个对象并填写好下一个对象的指针。第三部分是把字级内存池(mplhead_t 结构)加入到 kmempool_t 结构的 mp_obmplmheadl 链表中。可是这里的 mh_aliobsz、mh_objsz 都是 msize 的值,这是因为在 kmempool_objsz_new 函数中,已经对 msize 的值,调用了 OBJS_ALIGN(msize) 宏进

行了对齐调整, OBJS_ALIGN 宏是这样定义的: #define OBJS_ALIGN(x) ALIGN(x,4), 它总是把参数 x, 按 4 个字节对齐的, 因为 ARMCPU 访问数据时必须要是按 4 字节对齐的地址。

不管是查找到的还是新建的 mplhead_t 结构, 总之有了一个 mplhead_t 结构, 有了 mplhead_t 结构就有了相应的内存池, 下面就去写好真正分配字级内存池对象的代码吧。

7.5.6 分配时操作 mplhead_t

字级内存池上的分配内存空间的最后一步, 就是前面提到的操作查找到的或者新建的 mplhead_t 结构, 以至于最后成功分配内存空间。现在就去用代码实现这最后一步操作。

注意, 字级内存池中 (mplhead_t 结构) 没有用到 pglmap_t 数据结构, 所以在具体操作的细节上还是和页级内存池上的操作有些不同。下面就来写好这个函数, 如代码清单 7-50 [xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单 7-50 操作相应的 mplhead_t

```

adr_t objs_new_on_mplhead(mplhead_t* mplhdp)
{
    //如果这里任何一个条件为真, 即表示这个内存池中没有一个可分配的对象了
    if(mplhdp->mh_afindx>=mplhdp->mh_objnr||mplhdp->mh_firtfreadr==NULL)
    {
        return NULL;
    }
    //实际对象的大小
    adr_t retadr=NULL,*nextp=NULL,adrsz=(adr_t)(mplhdp->mh_aliobsz);
    //把mh_firtfreadr的值(当前空闲对象的地址)赋给retadr
    retadr=mplhdp->mh_firtfreadr;
    //(retadr+adrsz)就是当前对象后的下一个对象的指针
    nextp=(adr_t*)(retadr+adrsz);
    //让mh_firtfreadr指向下一个空闲对象
    mplhdp->mh_firtfreadr=*nextp;
    //把当前对象的下一个对象指针域设为NULL(断开链表)
    *nextp=NULL;
    //增加对象分配计数
    mplhdp->mh_afindx++;
    //返回分配内存空间的地址
    return retadr;
}

```

objs_new_on_mplhead 函数很简单, 无非就是把内存池 (mplhead_t 结构) 中的一个空闲对象从链表中取出, 让 mplhead_t 结构中的 mh_firtfreadr 域指向下一个空闲对象, 关键是要理解 nextp 指针是怎么计算出来的, 对照前面的数据结构理解这个应该不难。其他部分, 代码中的注释已经说明了一切。

下面还有一件重要的事要做, 就是把前面写好的几个函数, 放进主框架函数中让它们被调用, 以便完成整个内存空间的分配流程, 如代码清单 7-51 [xxxx/lmosem/kernel/

krlpagempol.c] 所示。

代码清单7-51 字级内存池的主分配函数

```

adr_t kmempool_objsz_core_new(size_t msize)
{
    adr_t retadr=NULL;
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_oblock,&cpufg);
    mplhdp=kmemplob_retn_mplhead(kmplp,msize);
    //没有找到相应的mplhead_t结构,就建立一个新的mplhead_t结构(字级内存池)
    if(mplhdp==NULL)
    {
        mplhdp=new_objsz_mpool(kmplp,msize);
        //如果建立mplhead_t结构(字级内存池)失败,就宣告内存空间分配失败
        if(mplhdp==NULL)
        {
            retadr=NULL;
            goto return_step;
        }
    }
    //操作mplhead_t结构,完成内存空间的分配
    retadr=objs_new_on_mplhead(mplhdp);
return_step:
    hal_spinunlock_restflg_sti(&kmplp->mp_oblock,&cpufg);
    return retadr;
}

```

kmempool_objsz_core_new 这个函数的结束,标志着字级内存管理层的分配流程已经完成。接着就要去测试一下,以确定这些代码正确无误,然后才能去写字级内存管理层的释放内存空间的代码。

7.5.7 分配代码写得对吗

字级内存管理层的内存空间分配代码就完成了。不过依然要写一个函数来测试这些分配代码写得对不对,只有一步一测试,才能写出稳定的代码。

下面就来写好这个测试函数,如代码清单 7-52[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-52 测试字级内存池分配

```

void testobjsmgr()
{
    adrsz_t adsz[64];
    size_t alcsz=0x20;
    uint_t i=0;
    for(,alcsz<0x800;alcsz+=0x20,i++)
    {

```

```

    adsz[i].sz=alcsz;
    adsz[i].adr=kmempool_new(alcsz);
    if(adsz[i].adr==NULL)
    {
        hal_sysdie("testobjsmgr kmempool_new err");
    }
    printfk("objssadsz[%x] objssz:%x objssadr:%x\n\r",i,adsz[i].sz,adsz[i].adr);
}
printfk("oskmempool.mp_obmplnr:%x\n\r",oskmempool.mp_obmplnr);
return;
}

```

代码清单 7-52 中，内存空间的分配大小从 32 字节开始，每次分配之后就加上 32 字节，直到 2048 字节。并且可以发现最多能分配 63 次，而且由于每次内存空间的分配大小不同，所以每次都会建立一个新的字级内存池，最后我们输出字级内存池的个数，查看一下正不正常就行了。

好了，编译好 LMOSEM 内核，注意用 testobjsmgr 函数替换掉原来 init_kmempool 函数中的 testpagemgr 函数。连接上开发板，按照前面的方法，把 LMOSEM 内核下载到开发板中运行。正常情况下会出现如图 7-14 所示的情况。

```

深度终端
objssadsz[2a] objssz:560 objssadr:304b103c
objssadsz[2b] objssz:580 objssadr:304b503c
objssadsz[2c] objssz:580 objssadr:304b903c
objssadsz[2d] objssz:5c0 objssadr:304c103c
objssadsz[2e] objssz:5e0 objssadr:304c503c
objssadsz[2f] objssz:600 objssadr:304c903c
objssadsz[30] objssz:620 objssadr:304cd03c
objssadsz[31] objssz:640 objssadr:304d103c
objssadsz[32] objssz:660 objssadr:304d503c
objssadsz[33] objssz:680 objssadr:304d903c
objssadsz[34] objssz:6a0 objssadr:304dd03c
objssadsz[35] objssz:6c0 objssadr:304e103c
objssadsz[36] objssz:6e0 objssadr:304e503c
objssadsz[37] objssz:700 objssadr:304e903c
objssadsz[38] objssz:720 objssadr:304ed03c
objssadsz[39] objssz:740 objssadr:304f103c
objssadsz[3a] objssz:760 objssadr:304f503c
objssadsz[3b] objssz:780 objssadr:304f903c
objssadsz[3c] objssz:7a0 objssadr:3050103c
objssadsz[3d] objssz:7c0 objssadr:3050503c
objssadsz[3e] objssz:7e0 objssadr:3050903c
oskmempool.mp_obmplnr: 3f
LMOSEM SYSTEM IS DIE LMOSEM RUN!!

```

图 7-14 测试字级内存池的分配代码

正好和前面分析的一样，是 63 个字级内存池。为什么返回的内存空间的地址后面都是 3c，这是由于每次分配都是分配了一个内存池中的第一个对象，想想 mplhead_t 结构本身的大小就明白了。如果读者不放心，可以遍历所有的字级 mplhead_t 结构，打印出它们的信息看一看对不对。

验证了字级内存管理层的分配代码，现在就可以去实现字级内存管理层的释放代码了。

7.5.8 释放主函数

前面写好了字级内存管理层的分配代码，只能分配内存空间而不能释放内存空间，这是不行的。现在就要来实现字级内存管理层的释放代码了。

回忆一下字级内存管理层的内存空间分配过程，而它释放内存空间的过程就是其逆过程，所以字级内存管理层的内存空间释放步骤，如下所示。

- ❑ 根据内存空间的释放地址和释放大小，在 `kmempool_t` 结构中查找相应的 `mplhead_t` 结构。
- ❑ 操作第一步中找到的 `mplhead_t` 结构，达到释放内存空间的目的。
- ❑ 最后还要删除已经空闲的内存池（`mplhead_t` 结构），真正释放物理内存空间。

同样，把以上三步写成一个个函数，然后再由一个主框架函数去调用它们。下面就来写好这个主框架函数，如代码清单 7-53[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-53 字级内存池的主释放函数

```
bool_t kmempool_objsz_core_delete(adr_t fradr, size_t frsz)
{
    bool_t rets=FALSE;
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    kmempool_t* kmplp=&oskmempool;
    hal_spinlock_saveflg_cli(&kmplp->mp_oblock, &cpufg);
    .....
    return_step:
        hal_spinunlock_restflg_sti(&kmplp->mp_oblock, &cpufg);
        return rets;
}
```

这种类型的代码，太熟悉了，已然不用多说什么。不过，代码依然要在加锁中执行整个内存空间的释放过程，`kmempool_t` 结构中的 `mp_oblock` 自旋锁，就是用于保护字级内存池的。

好了，下面就要去分别实现完成上述具体步骤的函数了。

7.5.9 释放时查找 `mplhead_t`

根据前面的字级内存管理层的释放步骤，第一步就是根据内存空间的释放地址和释放大小，在 `kmempool_t` 结构中查找相应的 `mplhead_t` 结构。因为 `mplhead_t` 结构是管理一个内存池的，我们要释放内存池中的一块内存空间，当然首先得找到这个 `mplhead_t` 结构。

下面就要实现这个查找函数了，查找方法还是遍历 `kmempool_t` 结构中挂载字级内存池（`mplhead_t` 结构）的链表头。查找条件当然也是内存空间的释放地址和释放大小，如代码清单 7-54[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-54 查找相应的mplhead_t

```

mplhead_t* objsdel_mplhead_isok(mplhead_t* mhp, adr_t fradr, size_t msize)
{
    //检查是不是字级内存池
    if(mhp->mh_hedty!=MPLHTY_OBJS)
        return NULL;
    //内存空间的释放大小是不是和内存池中的对象大小相等
    if(mhp->mh_aliobsz!=msize)
        return NULL;
    //该内存池中是不是已经分配内存空间对象了
    if(mhp->mh_afindx==0)
        return NULL;
    //内存空间的释放地址是不是落在内存池区间之中了
    if(fradr<(mhp->mh_start+sizeof(mplhead_t))||(fradr+msize-1)>mhp->mh_end)
        return NULL;
    return mhp;
}

mplhead_t* kmempldelob_retn_mplhead(kmempool_t* kmplockp, adr_t fradr, size_t
    msize)
{
    mplhead_t* retmhp;
    list_h_t* list;
    if(kmplockp->mp_obmplmhcach!=NULL)
    {
        retmhp=kmplockp->mp_obmplmhcach;
        retmhp=objsdel_mplhead_isok(retmhp, fradr, msize);
        if(retmhp!=NULL)
        {
            return retmhp;
        }
    }
    list_for_each(list, &kmplockp->mp_obmplmheadl)
    {
        retmhp=list_entry(list, mplhead_t, mh_list);
        retmhp=objsdel_mplhead_isok(retmhp, fradr, msize);
        if(retmhp!=NULL)
        {
            return retmhp;
        }
    }
    return NULL;
}

```

kmempldelob_retn_mplhead 函数依然是首先检查 kmempool_t 结构的 mp_obmplmhcach 域中指向的 mplhead_t 结构是不是要找的, mp_obmplmhcach 域中保存的是最近操作过的 mplhead_t 结构的指针。如果它不是要找的, 那么就要遍历整个 mp_obmplmheadl 链表了。遍历过程中每找到一个 mplhead_t 结构, 就调用 objsdel_mplhead_isok 函数检查它是不是要查找的 mplhead_t 结构。如果没有找到相应的 mplhead_t 结构, 那肯定是释放内存空间的参数出

问题了，所以整个释放过程也就只能宣告失败了。

好了，有了具体的 `mplhead_t` 结构，下面就要操作它以释放内存空间了。现在就去写操作它的代码。

7.5.10 释放时操作 `mplhead_t`

到了这里，我们一定是查找到了一个相应的 `mplhead_t` 结构，并且需要释放的内存空间，就在这个 `mplhead_t` 结构所管理的内存池中。现在就要操作 `mplhead_t` 结构，以便达到释放内存空间的目的。

下面来实现在字级内存管理层下，释放内存空间时操作 `mplhead_t` 结构的函数，如代码清单 7-55[xxxx/lmosem/kernel/kr/pagempol.c] 所示。

代码清单7-55 操作相应的`mplhead_t`

```
bool_t objs_delete_on_mplhead(mplhead_t* mplhdp, adr_t fradr)
{
    //做合理性检查，该内存池中的内存空间（内存池对象）是否已分配，释放地址是否落在这个内存池的区间中
    if (mplhdp->mh_afindx<1 || fradr<(mplhdp->mh_start+sizeof(mplhead_t)) ||
        (fradr+mplhdp->mh_aliobsz)>=mplhdp->mh_end)
    {
        return FALSE;
    }
    adr_t retadr=NULL,*nextp=NULL, adrsz=(adr_t) (mplhdp->mh_aliobsz);
    //计算释放对象（内存空间）的下一个对象指针域的地址
    nextp=(adr_t*)(fradr+adrsz);
    //释放对象（内存空间）的下一个对象指针域是否为NULL
    if(*nextp!=NULL)
    {
        return FALSE;
    }
    //将mh_firtfreadr域的值赋给retadr
    retadr=mplhdp->mh_firtfreadr;
    //将释放对象的地址：fradr赋给mh_firtfreadr域
    mplhdp->mh_firtfreadr=fradr;
    //将以前的mh_firtfreadr域的值赋给释放对象的下一个对象指针域（挂入链表）
    *nextp=retadr;
    //减少分配计数，因为释放了一个内存池对象（内存空间）
    mplhdp->mh_afindx--;
    return TRUE;
}
```

熟悉前面分配代码的读者，看懂这段代码应该不难，代码清单 7-55 中的注释已经写得很明白了。如果不懂请回到数据结构的内容介绍，搞清楚字级内存池的对象是怎么存放的，就明白了。

释放操作到这里并没有完，因为还有字级内存池本身占用的物理内存空间也要释放。当然前提是该内存池中的内存空间完全空闲。下面就去实现相关的代码。

7.5.11 释放时删除字级内存池

释放了字级内存池中的内存空间块，如果内存池中所有的内存空间块都空闲了，那么就要删除这个内存池，释放其占用的物理内存空间。所以释放操作的最后一步就是删除空闲的 `mplhead_t` 结构（内存池），真正释放其占用的物理内存空间。前面也提到过，等物理内存空间紧缺了，再来删除相应的内存池也不迟，但是我们为了代码简单，并没有实现此机制。

下面来写一个函数完成上述动作，如代码清单 7-56[xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-56 删除一个字级内存池

```
bool_t del_objs_mpool(kmempool_t* kmplockp, mplhead_t* mphdp)
{
    //如果字级内存池中还有对象没有释放，立刻返回真
    if (mphdp->mh_afindx>0)
    {
        return TRUE;
    }
    //释放字级内存池占用的大小
    size_t frsz=mphdp->mh_end-mphdp->mh_start+1;
    //释放字级内存池的开始地址
    adr_t fradr=mphdp->mh_start;
    //从kmempool_t结构中脱链
    list_del(&mphdp->mh_list);
    //对kmempool_t结构的mp_obmplnr域做合理性检查，一般不会发生这种情况
    if (kmplockp->mp_obmplnr<1)
    {
        hal_sysdie("del_objs_mpool kmplockp->mp_obmplnr<1");
    }
    //减少字级内存池计数
    kmplockp->mp_obmplnr--;
    //如果相等，则清除kmempool_t中的缓存
    if (kmplockp->mp_obmplmhcach==mphdp)
    {
        kmplockp->mp_obmplmhcach=NULL;
    }
    //释放字级内存池占用的页面
    if (kmempool_pages_delete(fradr,frsz)==FALSE)
    {
        hal_sysdie("del_objs_mpool page delete err");
    }
    return TRUE;
}
```

代码清单 7-56 中，首先检查这个内存池中的所有内存空间块是否都是空闲的。如果不是就立刻返回真。否则让其从 `kmempool_t` 结构中相应的链表上删除、减少字级内存池计数等，最后释放字级内存池所占用的物理内存空间。

下面就来把已经写好的那些完成字级内存空间释放的函数，装配到释放字级内存空间的主框架函数中，使它们结合起来一起完成释放字级内存空间的任務，如代码清单 7-57[lmosem/kernel/krlpagempol.c] 所示。

代码清单7-57 字级内存池的主释放函数

```
bool_t kmempool_objsz_core_delete(adr_t fradr, size_t frsz)
{
    bool_t rets=FALSE;
    cpuflg_t cpufg;
    mplhead_t* mplhdp;
    kmempool_t* kmplp=&oskmempool;
    //加锁
    hal_spinlock_saveflg_cli(&kmplp->mp_oblock, &cpufg);
    //查找相应的内存池 (mplhead_t结构)
    mplhdp=kmempldelob_retn_mplhead(kmplp, fradr, frsz);
    if (mplhdp==NULL)
    {
        rets=FALSE;
        goto return_step;
    }
    //释放内存池中的内存空间块
    if (objsz_delete_on_mplhead(mplhdp, fradr)==FALSE)
    {
        rets=FALSE;
        goto return_step;
    }
    //释放内存池自身占用的物理内存空间
    if (del_objsz_mpool(kmplp, mplhdp)==FALSE)
    {
        rets=FALSE;
        goto return_step;
    }
    rets=TRUE;
return_step:
    //解锁
    hal_spinunlock_restflg_sti(&kmplp->mp_oblock, &cpufg);
    return rets;
}
```

好了，代码清单 7-57 这种代码形式，前面已经看过很多了，加上其中的注释，非常简单，不必多说。

字级内存空间块的释放流程结束了，意味着整个字级内存管理层也结束了，但整个字级内存管理层的代码写得对不对，还得测试一下。

7.5.12 测试字级内存管理层

字级内存管理层的分配、释放内存空间块的代码都有了，所以这次是针对字级内存管理

层的整体测试，因此测试代码也就复杂了许多。

下面就去写一写测试代码，其实我们只需要修改两个函数的代码即可，如代码清单 7-58 [xxxx/lmosem/kernel/krlpagempol.c] 所示。

代码清单7-58 测试字级内存管理

```

void cmp_adrsz(adrsz_t* assp, uint_t nr)
{
    //前面那些对地址、大小的比对，依然不能丢
    .....
    //在cmp_adrsz函数的尾部加上以下代码，对分配的内存空间中的值进行比对
    u8_t* bytp=NULL, bytv=0;
    for(uint_t n=0; n<nr; n++)
    {
        //指向内存空间的开始地址
        bytp=(u8_t*)(assp[n].adr);
        //把内存空间的开始地址的低8位的值赋给bytv
        bytv=(u8_t)(assp[n].adr&0xff);
        for(uint_t o=0; o<assp[n].sz; o++)
        {
            //对该空间的每个字节的值进行依次比对
            if(bytp[o]!=bytv)
            {
                hal_sysdie("cmp val err");
            }
        }
    }
    return;
}

void testobjsmgr()
{
    adrsz_t adsz[64];
    size_t alcsz=0x20;
    u8_t* adrbytp=NULL, bytval=0;
    for(; alcsz<0x800; alcsz+=0x10)
    {
        for(int i=0; i<64; i++)
        {
            adsz[i].sz=alcsz;
            adsz[i].adr=kmempool_new(alcsz);
            if(adsz[i].adr==NULL)
            {
                hal_sysdie("testobjsmgr kmempool_new err");
            }
            printfk("objs alloc adsz[%x] sz:%x adr:%x\n\r", i, adsz[i].sz, adsz[i].adr);
            adrbytp=(u8_t*)adsz[i].adr;
            //把该空间的开始地址的低8位的值赋给bytval
            bytval=(u8_t)(adsz[i].adr&0xff);
            //把bytval的值写进该空间的每个字节中
            for(int k=0; k<adsz[i].sz; k++)

```

```

    {
        adrbytp[k]=bytval;
    }
}
//进行校验
cmp_adrsz(adsz,64);
//释放已经分配的内存空间块
for(int j=0;j<64;j++)
{
    if(kmempool_delete(adsz[j].adr,adsz[j].sz)==FALSE)
    {
        hal_sysdie("testobjsmgr kmempool_delete err");
    }
    printfk("objs delete adsz[%x] sz:%x adr:%x\n\r",j,adsz[j].sz,adsz[j].adr);
}
printfk("oskmempool.mp_obmplnr:%x\n\r",oskmempool.mp_obmplnr);
return;
}

```

还是每种内存空间大小，分配一定的次数后，进行校验，最后释放。不同的是，这次每分配一块内存空间，就把这块内存空间的开始地址的低 8 位，写进这块内存空间中，同时也修改了 `cmp_adrsz` 函数，加入了一些代码，用于比对内存空间中的值。由于分配了又释放，其相应的管理数据应该还原，所以最后输出 `kmempool_t` 结构中 `mp_obmplnr` 域的值，以便观察。

编译好 LMOSEM 内核，连接上开发板，最后把 LMOSEM 内核下载到开发板中运行，正常情况下会出现如图 7-15 所示的情况。

```

深度终端
objs delete adsz[2b] sz:7f0 adr:30416918
objs delete adsz[2c] sz:7f0 adr:3041700c
objs delete adsz[2d] sz:7f0 adr:30417800
objs delete adsz[2e] sz:7f0 adr:30417ff4
objs delete adsz[2f] sz:7f0 adr:304187e8
objs delete adsz[30] sz:7f0 adr:3041903c
objs delete adsz[31] sz:7f0 adr:30419830
objs delete adsz[32] sz:7f0 adr:3041a024
objs delete adsz[33] sz:7f0 adr:3041a818
objs delete adsz[34] sz:7f0 adr:3041b00c
objs delete adsz[35] sz:7f0 adr:3041b800
objs delete adsz[36] sz:7f0 adr:3041bfff4
objs delete adsz[37] sz:7f0 adr:3041c7e8
objs delete adsz[38] sz:7f0 adr:3041d03c
objs delete adsz[39] sz:7f0 adr:3041d830
objs delete adsz[3a] sz:7f0 adr:3041e024
objs delete adsz[3b] sz:7f0 adr:3041e818
objs delete adsz[3c] sz:7f0 adr:3041f00c
objs delete adsz[3d] sz:7f0 adr:3041f800
objs delete adsz[3e] sz:7f0 adr:3041fff4
objs delete adsz[3f] sz:7f0 adr:304207e8
oskmempool.mp_obmplnr:0
LMOSEM SYSTEM IS DIE LMOSEM RUN!

```

图 7-15 测试字级内存管理层

看到这个结果，证明我们的代码正确无误，字级内存管理层就此结束，以后开发 LMOSEM 内核的其他组件就方便多了，因为有了动态分配内存空间的能力。动态，这可是

一个不小的飞跃，因为在 LMOSEM 内核中，有很多数据结构的数量在开发时是不确定的，只有到运行状态下根据情况才能确定需不需要，需要时就新建一个，LMOSEM 内核的扩展性就是从这里开始的。

7.6 小结

“小结”一个另人兴奋的字眼，它标志着一个部分的结束，意味着又向前跃进了一步，我们 LMOSEM 内核的整个内存管理组件就做好了。

从整体上看，LMOSEM 内核的内存管理组件分为如下三层。

1) 块级内存管理层：它是最开始实现的，位于 LMOSEM 内核的硬件相关层，实现了 128KB、256KB、512KB、1MB、2MB、4MB，6 种大小的内存空间块的分配。

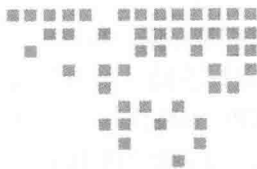
2) 页级内存管理层：它位于 LMOSEM 内核的功能层中，基于内存池的思想，依赖于块级内存管理层，能够提供 1 个页面（4KB）到 31 个页面的分配粒度。

3) 字级内存管理层：它也位于 LMOSEM 内核的功能层中，也同样是基于内存池的思想，依赖于页级内存管理层，分配粒度为 32 ~ 2048 字节，如果分配粒度小于 32 字节，就直接分配 32 字节大小的内存空间，并且它是作为 LMOSEM 内核自身使用的。

虽然 LMOSEM 内核的内存管理组件的性能不太可观，但总算能满足各种大小的内存空间分配请求。

笔者要开始说“但是”了，但是 LMOSEM 内核只是实现了物理内存的管理，并不具备像 Linux、Windows 有强大的虚拟内存管理。虚拟内存管理要用到 MMU 器件，它最终会把虚拟内存空间映射到物理内存空间，所以物理内存管理是虚拟内存管理得以实现的基础。作为一个探索原理的操作系统内核，不应该太复杂，也不应该太简单。如果有一天，我们觉得 LMOSEM 内核的内存管理组件不够完美，再来实现虚拟内存管理也不迟。

啊！LMOSEM 内核终于有了内存管理组件，有了它，实现其他 LMOSEM 内核组件也就容易多了，让我们接着探索 LMOSEM 内核里的其他东西吧。



中断管理

除了初始化之外，操作系统对应用提供服务。当然，这些服务有些是操作系统中的一些软件完成的，而有时要借助于设备完成，但不管怎么样，它们都是从中断开始的，后面读者慢慢就会认同这句话。

从上面的言语中，好像感觉中断就是操作系统之灵魂。我们暂且先带着这样的感觉，看看什么是中断、了解中断控制器、如何处理中断、怎么添加中断回调函数。下面就要脚踏实地，一步一步地来。

8.1 中断与中断控制器

中断到底是什么，是干什么的，有什么作用，能否被控制，用什么方法控制。下面就去解决这些问题。

8.1.1 什么是中断

先来看三种情景：“你在开车中，突然汽车引擎坏了，你需要修复它才能继续驾驶汽车……”，“你在外旅游，你女朋友突然来电话了，你可以选择接电话或者不接电话，当然不接电话的后果很严重（笑）……”，“在老鼠必经的路上放上老鼠夹，只要老鼠走上这条路踩上老鼠夹，就必然被老鼠夹夹住，如果不解开老鼠夹，老鼠无法继续前行……”

在以上三种情景中，虽然不十分恰当，但都是在做一件事时，因为一些原因从而要做另一件事。例如，开车中汽车引擎坏了、在外旅游女朋友来电话了、老鼠踩上老鼠夹。

计算机中的 CPU 也是一样的，在做一件事时，因为一些原因从而要做另一件事，于是

中断产生了。

因为原因的类型不同，所以中断被分为如下三类：

1) 异常：是同步的，是因为错误和故障，如汽车引擎坏了。不修复错误就不能继续运行，所以这时 CPU 会跳到这种错误的处理代码那里开始运行，运行完了会返回。如果不修改程序中的错误，下次运行程序到这里同样会发生异常，所以它是同步的。

2) 中断：是异步的，我们通常说的中断就是这种类型，它是因为外部事件而产生的，例如，旅游时女朋友来电话了。通常是设备需要 CPU 关注时，给 CPU 发送一个中断信号，所以这时 CPU 会跳到处理这种事件的代码那里开始运行，运行完了会返回。由于不知道何时设备何时发出这种中断信号，所以它是异步的。

3) 陷入：也是同步的，例如，老鼠只要踩上老鼠夹，就必然被老鼠夹夹住，CPU 中也有专门制造老鼠夹的指令——swi 指令，只要一运行这条 swi 指令，CPU 就会跳到专门安排的代码处运行，运行完了会返回。只要一运行这条指令，CPU 就会发生这种行为，所以我们说它也是同步的。

中断，一句话：打断当前 CPU 代码的执行流，转而执行另一段代码的行为。

或许上面的例子还不能让你明白中断有什么用，下面就来看看几个中断的应用场景。

场景一：如果当前程序中有条指令 CPU 不认识，而 CPU 又没有中断异常机制，CPU 该何去何从，可能机器重启是最好的方法，但是这是不是太粗暴了。如果 CPU 有中断异常机制，CPU 就能在这种情况下，跳转到相应的处理代码处开始运行并修复错误。

场景二：CPU 向设备发送一条命令以读取设备中的数据，可是设备的数据还没有准备好，而 CPU 又没有中断异常机制，CPU 只能在那里循环等待设备，不能做任何其他事，这是不是效率太低了。如果 CPU 有中断异常机制，CPU 就能在这种情况下去做别的事情，设备准备好相关的数据后给 CPU 发送一个相应的中断信号，然后 CPU 在中断处理代码中就能读取设备的数据了。

场景三：操作系统代码是为应用程序服务的。如果 CPU 没有陷入机制，应用程序只能直接调用操作系统的代码，这种情况下根本就不受控制，系统代码也不能被保护，系统的安全、可靠性无从谈起。如果 CPU 陷入机制后，系统代码就可以被保护起来，应用程序要得到服务，只有执行陷入指令，一旦执行陷入指令，就会被操作系统的代码接管。

总之，中断综合起来就是，通过打断当前 CPU 代码的执行流，转而执行另一段代码的方式：处理异常事件、解决 CPU 与设备之间的通信效率、保证内核代码安全地为应用程序提供服务。中断本身是简单的，因为它而产生的一切功能却不简单。这里了解了中断是什么就行了，下面我们继续。

8.1.2 S3C2440A 中断控制器

中断带来的作用是巨大的，可是中断要不要被适当控制，答案是肯定的。如果不可被控制，那是相当可怕的。

前面发现中断有三种类型，其中异常和陷入是同步的，本身就是可控的，只要代码中没有错误或者包含陷入指令，CPU 就不会产生异常和陷入。可是设备中断就不同了，它是异步的，我们不知道它何时向 CPU 发出中断信号。例如，有时临界区中代码的运行是绝对不能产生中断的。

由上可知，对于设备产生的异步中断一定要被控制，因为不仅仅是允不允许 CPU 响应中断，还要确定中断的来源（因为系统中不止一个设备），对每个中断源分别控制，允不允许它们各自产生中断，它们产生一个中断后要等待 CPU 处理，不然它们不能产生下一次中断等。由于这些控制很复杂，所以产生了专门的控制芯片——中断控制器。

关于 S3C2440A 中断控制器的介绍，请参阅前面章节，这里我们直接进入系统编程需要的东西。S3C2440A 中断控制器在 S3C2440A 芯片内部，最多支持 60 个中断源。这 60 个中断源分两层级连的，我们只能直接获取顶层的 32 个中断源，也就是说顶层中的有些中断源，包含了几个下级中断源，如图 8-1 所示。

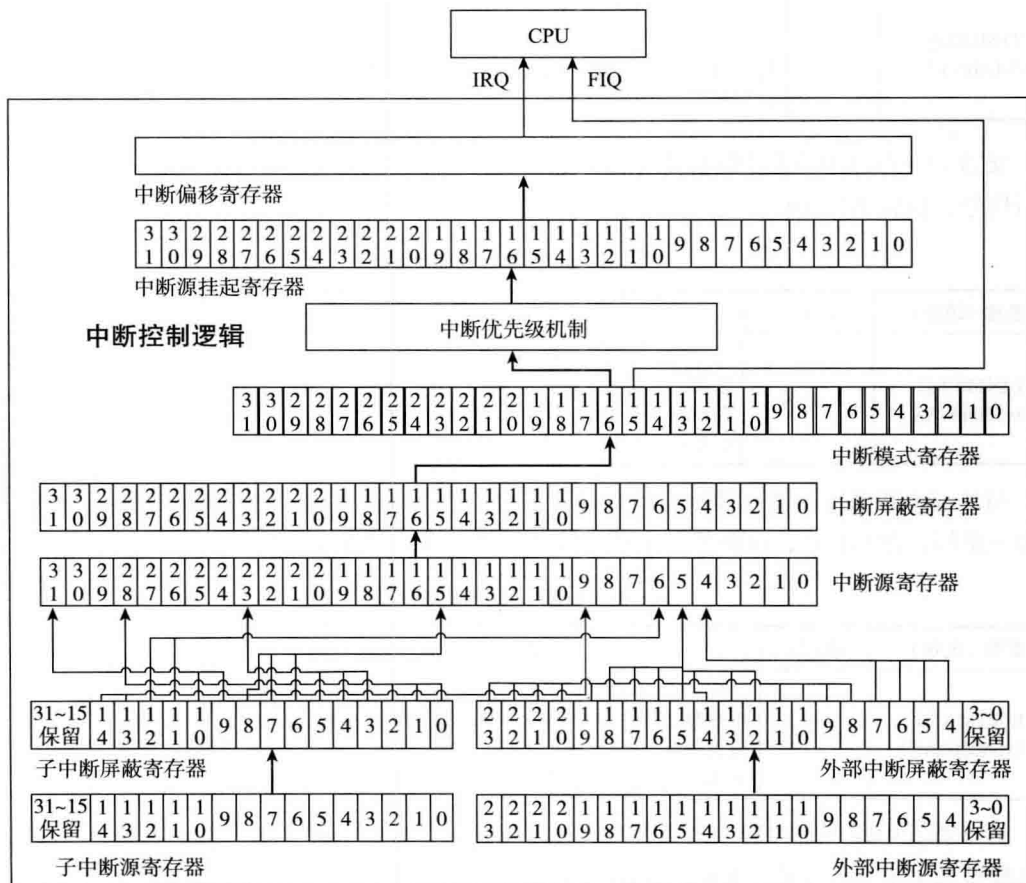


图 8-1 S3C2440A 的中断控制逻辑

图 8-1 中也画出了一些重要的寄存器，其实硬件提供给软件的编程接口就是这一个个的寄存器，所以作为编程人员，了解这些寄存器就行了。下面分别介绍这些寄存器。

图 8-1 中从下向上看，首先是子中断源寄存器，如表 8-1 所示。

表8-1 子中断源寄存器

名称 (地址)	读 / 写	说 明	中 断 源
SUBSRCPND (0X4A000018)	读 / 写	15 ~ 31 位保留。0 ~ 14 位，每位对应一个中断源，其位为 1 时表示有中断请求，为 0 时表示无中断请求。初始化时为 0	参见 6.6.1 节

与子中断源寄存器直接相连的是子中断屏蔽寄存器。子中断屏蔽寄存器就像一扇门，它不打开，这些子中断信号是不能通过的，如表 8-2 所示。

表8-2 子中断屏蔽寄存器

名称 (地址)	读 / 写	说 明	备 注
INTSUBMSK (0X4A00001C)	读 / 写	15 ~ 31 位保留为 0。0 ~ 14 位，每位对应一个子中断源，其位为 1 时表示屏蔽该位对应的中断，为 0 时表示放行该位对应的中断。初始化时，0 ~ 14 位为 1，15 ~ 31 位为 0	该寄存器位与子中断源寄存器位位对应

接着看看右边的外部中断源寄存器，它和子中断源寄存器是平级的，它负责外部器件的中断信号，如表 8-3 所示。

表8-3 外部中断源寄存器

名称 (地址)	读 / 写	说 明	中 断 源
EINTPEND 0x560000A8	读 / 写	0 ~ 3 位保留为 0。4 ~ 23 位，每位对应一个外部中断源，其位为 1 时表示有中断请求，为 0 时表示无中断请求。初始化时为 0。注意 4 ~ 23 位，各位要清 0，必须向其位写入 1 才行	参见 6.6.1 节

与外部中断源寄存器直接相连的也是外部中断屏蔽寄存器。同样，外部中断屏蔽寄存器也像一扇门，它不打开，这些外部中断信号也不能通过，如表 8-4 所示。

表8-4 外部中断屏蔽寄存器

名称 (地址)	读 / 写	说 明	备 注
EINTMASK (0x560000A4)	读 / 写	0 ~ 3 位保留为 1。4 ~ 23 位，每位对应一个外部中断源，其位为 1 时表示屏蔽该位对应的中断，为 0 时表示放行该位对应的中断。初始化时，4 ~ 19 位为 1，20 ~ 23 位为 0	该寄存器位也是与外部中断源寄存器位位对应的

再来看看中断源寄存器，其中有些位连接着多个子中断源和多个外部中断源，所以如果它有些位为 1 了，还要进一步确定是哪个子中断源或者外部中断源，如表 8-5 所示。

表8-5 中断源寄存器

名称 (地址)	读 / 写	说 明	中 断 源
SRCPND (0X4A000000)	读 / 写	0 ~ 31 位, 每位对应一个中断源, 其位为 1 时表示有中断请求, 为 0 时表示无中断请求。初始化为 0	参见 6.6.1 节

同样, 中断源寄存器上面也有一扇门, 就是中断屏蔽寄存器, 它其中相关的位不打开, 中断源信号是不能通过的, 如表 8-6 所示。

表8-6 中断屏蔽寄存器

名称 (地址)	读 / 写	说 明	备 注
INTMSK (0X4A000008)	读 / 写	0 ~ 31 位, 每位对应一个中断源, 其位为 1 时表示屏蔽该位对应的中断, 为 0 时表示放行该位对应的中断。初始化时所有位为 1	该寄存器位与中断源寄存器位对应

由于 CPU 支持两种中断方式即 IRQ 和 FIQ 中断方式, 所以得有个模式寄存器, 如表 8-7 所示。注意, FIQ 模式的中断并不经过中断优先级机制仲裁, 直接就到达 CPU 的 FIQ 引脚上去了。中断优先级机制, 其实就是选择一个需要最先响应的中断源送给 CPU 处理, 这里不介绍其细节。

表8-7 中断模式寄存器

名称 (地址)	读 / 写	说 明	备 注
INTMOD (0X4A000004)	读 / 写	0 ~ 31 位, 每位对应一个中断源。如果某个指定位被设置为 1, 则在 FIQ (快中断) 模式中处理相应中断。否则在 IRQ 模式中处理。初始化时所有位为 0	该寄存器位与中断源寄存器位对应

如果多个 IRQ 模式的中断源同时发生了中断, 会首先通过中断优先级机制仲裁出一个最先需要响应的中断源, 并在中断源挂起寄存器相应位中标记, 同一时刻其中只会有一位被置 1, 如表 8-8 所示。

表8-8 中断源挂起寄存器

名称 (地址)	读 / 写	说 明	备 注
INTPND (0X4A000010)	读 / 写	0 ~ 31 位, 每位对应一个中断源。如果某个指定位被设置为 1, 即表明了相应未屏蔽并且正在等待中断服务的中断源请求, 具有最高的优先级。所以只有 1 位可以设置为 1, 并且产生 IRQ 模式中断请求给 CPU。中断服务程序读取此寄存器来决定是哪个中断源发生了中断。初始化为 0	该寄存器位与中断源寄存器位对应

最后来看看中断偏移寄存器, 这个寄存器也是 32 位的, 它里面保存着当前中断源挂起寄存器中为 1 的位号, 即 0 ~ 31 个中断源, 哪个中断源产生了中断, 如表 8-9 所示。

表8-9 中断偏移寄存器

名称 (地址)	读 / 写	说 明	备 注
INTOFFSET (0x4A000014)	只读	0 ~ 31 位, 只读的, 寄存器的值从 0 ~ 31, 中断服务程序读取此寄存器来决定哪个中断源发生了中断。初始化时为 0	无

中断控制器的一些关键寄存器就介绍到这里了。图 8-1 中画得可能与具体芯片的实现不一致, 不过这也没什么问题, 因为我们了解这 9 个寄存器就可以了。

通过前面几章和这里的介绍, 对 S3C2440A 的中断控制器的理解, 已经了然于胸了, 这仅仅是硬件, 下面就要做软件了, 写代码合理地管理和控制这些中断了。

8.2 中断管理的架构与相关数据结构

前面已经熟悉了中断控制器, 操作系统内核只要和中断控制器打交道就行了。在开始写代码之前, 我们要做到对中断管理架构有所了解, 例如, 它在内核中是处于哪个层次的, 又是怎么组织的。当然了, 要合理管理这么多中断还需要设计相应的数据结构才行。

话不多说, 开始吧。

8.2.1 中断管理的架构

如同内存管理组件一样, 中断管理也需要设计相应的架构, 这样才能便于内核代码以后的移植和维护, 所以对内核中的每个组件设计良好的架构是必要的。

LMOSEM 内核整体上有三层, 这个前面已经说过了。但是几乎每个计算平台的中断控制器都不一样, 所以操作中断控制器本身的代码在不同的计算平台下肯定是要改变的, 那么操作中断控制器的代码和函数, 就应该放在内核的 hal 层 (硬件平台相关层) 中。而操作中断控制器的接口和一些通用功能的代码, 应该放在内核功能层中, 因为这些代码不太可能发生改变。由上所述, 中断管理组件可分为两层, 上层是中断管理接口和针对中断管理的通用功能, 下层是中断管理的核心。

再来看看中断管理的下层中有些什么, 不外乎就是中断向量、保存 CPU 上下文的代码、中断分发器、中断控制器相关的代码等。

所以 LMOSEM 内核的中断管理组件的架构, 如图 8-2 所示。

一图解千言, 图 8-2 中清晰地展示了中断管理组件的架构。但是除了中断向量外, 其他部分都好像不太明白, 不要紧张, 因为这正是我们后面要实现的。这里只要清楚这个架构就行了。

设计好了中断管理组件, 接着就要编码实现了, 我们继续。

8.2.2 设计数据结构 intfltdsc_t 和 intserdsc_t

中断管理组件的框架已经设计好了, 下面就要开始写代码具体实现了, 不过首先要设计相应的数据结构。

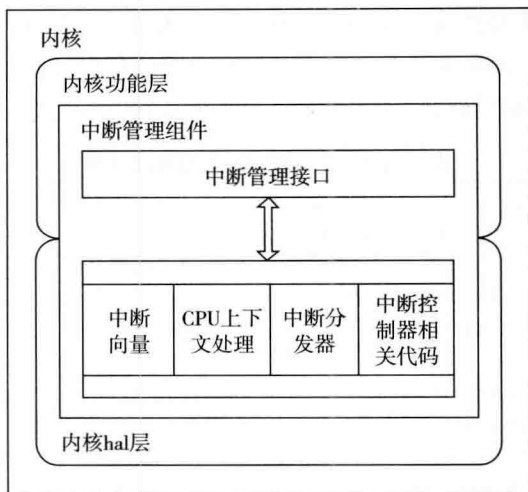


图 8-2 LMOSEM 中断管理组件的结构

中断管理组件重要的数据结构，无非就是以下两个：

- ❑ 中断源描述符，一个中断源描述符管理一个中断源。这个数据结构在初始化章节中已经看过了。
- ❑ 中断回调函数描述符，中断管理的重点，是要让设备驱动程序能处理设备的中断。后面会发现，中断处理分为公共部分和特定的设备部分。

先来看看中断源描述符，这个数据结构在前面章节中就说过了，并且我们用这种数据结构类型的数组，平级映射了 mini2440 上的外部中断源、子中断源、主中断源。在这里仍然贴出它的代码，如代码清单 8-1[xxxx/lmosem/include/halinc/halintupt_t.h] 所示。

代码清单8-1 中断源数据结构

```

typedef struct s_INTFLTDSC
{
    spinlock_t    i_lock;           //保护其自身
    list_h_t      i_serlist;        //设备中断服务程序链，后面会介绍
    uint_t        i_sernr;          //设备中断服务程序链的个数
    u32_t         i_flg;            //标志位
    u32_t         i_stus;           //状态位
    u32_t         i_pndbitnr;       //中断源寄存器中的位序
    uint_t        i_irqnr;          //中断号
    uint_t        i_deep;           //保留
    uint_t        i_indx;           //产生中断的次数
}intfltdsc_t;

```

intfltdsc_t 已然不必多说了，下面来看看 intserdsc_t 数据结构，它可以称为中断回调函数描述符，也可以称为中断服务函数描述符。总之，该数据结构中存放着具体设备中断处理代码的地址和相关信息。为什么不直接放在 intfltdsc_t 中呢，因为在有些计算平台上，多

个设备的中断信号线可能是连接在同一个中断源上。既然可以有多个设备，那么就不可能只包含一个中断回调函数，还需要包含一些设备信息。因此我们需要设计另一个数据结构，即 `intserdsc_t`，如代码清单 8-2 [xxxx\lmosem\include\halinc\halintupt_t.h] 所示。

代码清单8-2 中断回调函数描述符

```
typedef sint_t drvstus_t; //设备返回值类型
typedef drvstus_t (*intflthandle_t)(uint_t ift_nr,void* device,void* sframe); //中
断回调函数的指针类型：该函数类型返回值为drvstus_t，参数类型为uint_t//ift_nr（中断号），void*
device（设备指针），void* sframe（中断栈指针）
typedef struct s_INTSERDSC
{
    list_h_t      s_list;           //该链表是挂在intfltdsc_t中的i_serlist上的
    list_h_t      s_indevlst;       //该链表是挂在设备描述符中的相关链表上的
    u32_t         s_flg;           //一些标志
    intfltdsc_t*  s_intfltp;        //指向它所在的intfltdsc_t，通过这个可快速找到intfltdsc_t
    void*         s_device;         //指向其设备描述符，可以通过它找到这是哪个设备的中断
    uint_t        s_indx;          //该中断回调函数的运行计数
    intflthandle_t s_handle;        //回调函数的指针，保存回调函数的地址
}intserdsc_t;
```

注意，`intfltdsc_t` 是静态定义的，但是 `intserdsc_t` 不是，它是由驱动程序动态分配建立的。后面就会发现，驱动程序动态分配建立一个 `intserdsc_t` 数据结构后，就把它挂在 `intfltdsc_t` 中的 `i_serlist` 上。

还是用一幅图来展示 `intfltdsc_t` 和 `intserdsc_t` 数据结构之间的关系，心中有图，理解起来就容易多了，如图 8-3 所示。

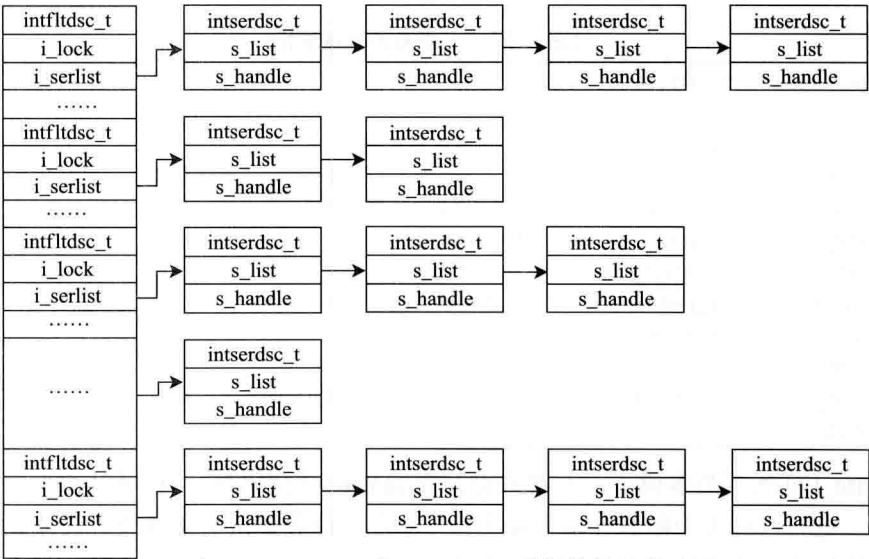


图 8-3 intfltdsc_t 与 intserdsc_t 数据结构的关系

图 8-3 中，只画出了数据结构中的关键域，实际中也可能没有这么多的 `intserdsc_t`，这不过是可能会出现的一种情况。

设计好了数据结构，并清楚了它们之间的关系，下面就去编写相关的代码了。

8.3 中断处理

一个硬件中断的到来，接下来的动作当然是中断处理，这也是中断管理组件的核心部分。本节首先会写几个中断辅助例程，然后从中断向量开始，一步步到最后调用中断处理例程结束整个中断处理过程。

前面已经了解了中断控制器，也知道了中断管理组件的架构，同时设计好了相关的数据结构，下面就去用代码实现中断处理的整个过程。

8.3.1 中断辅助例程

在中断处理开始之前，首先要实现一些处理中断中要用到的辅助功能函数，如下：

- ❑ 获取中断源描述符。
- ❑ 确定中断源。
- ❑ 清除中断源挂起寄存器。
- ❑ 打开和关闭中断源。

先来看看怎么获取中断源描述符，现在就来写这个函数，如代码清单 8-3[xxxx/lmosem/hal/halintupt.c] 所示。

代码清单8-3 获取中断源描述符

```
intfltdsc_t* hal_retn_intfltdsc(uint_t ifdnr)
{
    if(ifdnr>=osmach.mh_intfltnr)
    {
        return NULL;
    }
    return &osmach.mh_intfltdsc[ifdnr];
}
```

代码清单 8-3 中首先对参数 `ifdnr` 进行检查，看它是否大于等于 `osmach.mh_intfltnr`，`osmach` 这个数据结构在前面已经写过了，并且我们知道 `intfltdsc_t` 类型数组的个数就是放在 `osmach.mh_intfltnr` 中的，而 `intfltdsc_t` 类型数组的首地址则放在 `osmach.mh_intfltdsc` 中。

中断来了要获取中断源，即是哪个中断源发生了中断，如何获取中断源呢，前面知道我们所用的计算平台上的中断控制器中有个中断偏移寄存器，它里面的数据正是表示哪个中断源发生了中断，中断发生后读取它就行了，如代码清单 8-4[xxxx/lmosem/hal/halintupt.c] 所示。

代码清单8-4 读取中断偏移寄存器

```
uint_t hal_retn_intnr()
{
    return (uint_t)hal_io32_read(INTOFFSET_R);
}
```

代码清单 8-4 中的 `hal_io32_read` 函数在前面章节中已经写过了，`INTOFFSET_R` 宏正是中断偏移寄存器的地址 `0x4A000014`。代码很简单不必多说。

中断处理完了，必须要清除这个中断源在中断挂起寄存器中的相关位，它才能响应下一次中断。所以要写一个清除中断源挂起寄存器的函数，如代码清单 8-5[xxxx/lmosem/hal/halintupt.c] 所示。

代码清单8-5 清除中断源挂起寄存器

```
drvstus_t hal_clear_srcpnd(uint_t ifdnr)
{
    u32_t inttmp;
    uint_t flg;
    uint_t phylinenr;
    intfltdsc_t* ifdp=hal_retn_intfltdsc(ifdnr);
    if(ifdp==NULL)
    {
        return DFCERRSTUS;
    }
    flg=ifdp->i_flg&0xff;
    phylinenr=ifdp->i_pndbitnr;
    if(flq==MINT_FLG)
    {
        inttmp=(1<<phylinenr);
        hal_io32_write(SRCPND_R,inttmp);
        return DFCOKSTUS;
    }
    if(flq==SINT_FLG)
    {
        inttmp=(1<<phylinenr);
        hal_io32_write(SUBSRCPND_R,inttmp);
        return DFCOKSTUS;
    }
    if(flq==EINT_FLG)
    {
        inttmp=(1<<phylinenr);
        hal_io32_write(EINTPEND_R,inttmp);
        return DFCOKSTUS;
    }
    return DFCERRSTUS;
}
```

代码清单 8-5 虽然有点长,但是很简单,首先根据参数 `ifdnr` 获取中断源描述符 `intfltdsc_t`,接着取出这个中断源描述符对应的物理中断线在寄存器中的位号,这个位号就是保存在 `intfltdsc_t` 的 `i_pndbitnr` 域中的。然后根据中断源描述符的类型,分别进行操作,因为一个中断源描述符管理的可能是中断源、子中断源、外部中断源的其中之一。这些中断源可能对应不同的中断源挂起寄存器,最后调用 `hal_io32_write` 把相关中断源挂起寄存器的特定位写上 1 就清除该特定位了。执行成功就返回 `DFCOKSTUS` (即为 1),失败则返回 `DFCERRSTUS` (即为 -1)。

有时需要开启或者关闭某个设备的中断,这就需要通过操作中断控制器相应的屏蔽中断寄存器而达到。所以还要写两个函数,即开启和屏蔽中断的函数,如代码清单 8-6[xxxx/lmosem/hal/halintupt.c] 所示。

代码清单8-6 开启和关闭设备中断

```

drvstus_t hal_enable_intline(uint_t ifdnr)
{
    u32_t inttmp;
    uint_t flg;
    uint_t phylinenr;
    intfltdsc_t* ifdp=hal_retn_intfltdsc(ifdnr);
    if(ifdp==NULL)
    {
        return DFCERRSTUS;
    }
    flg=ifdp->i_flg&0xff;
    phylinenr=ifdp->i_pndbitnr;
    if(flq==MINT_FLG)
    {
        inttmp=hal_io32_read(INTMSK_R);
        inttmp&=(~(1<<phylinenr));
        hal_io32_write(INTMSK_R,inttmp);
        return DFCOKSTUS;
    }
    if(flq==SINT_FLG)
    {
        inttmp=hal_io32_read(INTSUBMSK_R);
        inttmp&=(~(1<<phylinenr));
        hal_io32_write(INTSUBMSK_R,inttmp);
        return DFCOKSTUS;
    }
    if(flq==EINT_FLG)
    {
        inttmp=hal_io32_read(EINTMASK_R);
        inttmp&=(~(1<<phylinenr));
        hal_io32_write(EINTMASK_R,inttmp);
    }
}

```

```

        return DFCOKSTUS;
    }
    return DFCERRSTUS;
}
drvstus_t hal_disable_intline(uint_t ifdnr)
{
    u32_t inttmp;
    uint_t flg;
    uint_t phylinenr;
    intfltdsc_t* ifdp=hal_retn_intfltdsc(ifdnr);
    if(ifdp==NULL)
    {
        return DFCERRSTUS;
    }
    flg=ifdp->i_flg&0xff;
    phylinenr=ifdp->i_pndbitnr;
    if(flg==MINT_FLG)
    {
        inttmp=hal_io32_read(INTMSK_R);
        inttmp|=(1<<phylinenr);
        hal_io32_write(INTMSK_R,inttmp);
        return DFCOKSTUS;
    }
    if(flg==SINT_FLG)
    {
        inttmp=hal_io32_read(INTSUBMSK_R);
        inttmp|=(1<<phylinenr);
        hal_io32_write(INTSUBMSK_R,inttmp);
        return DFCOKSTUS;
    }
    if(flg==EINT_FLG)
    {
        inttmp=hal_io32_read(EINTMASK_R);
        inttmp|=(1<<phylinenr);
        hal_io32_write(EINTMASK_R,inttmp);
        return DFCOKSTUS;
    }
    return DFCERRSTUS;
}

```

因为中断屏蔽寄存器和中断源挂起寄存器是位位对应的，所以在代码清单 8-6 中还是用到了中断源描述符中的 `i_pndbitnr` 域。中断屏蔽寄存器是把相应的位：置 1 表示禁止中断，置 0 表示开启中断，所以操作中断屏蔽寄存器时一定要用读 - 改 - 写的方式，例如，代码清单 8-6 中先调用 `hal_io32_read` 读取，然后改变值，最后调用 `hal_io32_write` 写入新值到寄存器。因为中断源、子中断源、外部中断源的原因，导致相应的中断屏蔽寄存器也不同，所以

要像 `hal_clear_srcpnd` 函数一样，要对中断源描述符进行判断。

好了，中断辅助例程也写好了，一共 5 个简单的函数，有了这些函数，在中断处理的过程中就可以进行相应的控制了。下面就正式开始中断的处理过程了。

8.3.2 从中断向量开始

在什么是中断的章节中，因为打断 CPU 当前代码执行流的原因不同，中断被分为三类，分别是异常、中断、陷入，其实它们都是广义上的中断，只是处理上稍有区别。

异常、中断、陷入的处理，是一个硬件和软件协作的过程。异常、中断、陷入发生时，首先是硬件做了一部分工作之后，才把控制权交给软件。由于操作系统的内核是软件，作为一个内核开发者，所以我们认为异常、中断、陷入的处理是从中断向量开始的。

关于异常、中断、陷入发生时，硬件所做的动作，在第 3 章已经说明，关于中断向量在第 6 章初始化时也已经说明了，在这里仍然列出相应的代码回顾一下，以便后面的代码更有连贯性，如代码清单 8-7 [xxxx/lmosem/hal/vector.s] 所示。

代码清单8-7 ARM920T处理器的中断、异常向量

```
.section .lmosem_hal.vector, "ax"
.code 32
.align 0
.extern hal_undefins
.extern hal_swi
.extern hal_prefabt
.extern hal_dataabt
.extern hal_bugdie
.extern hal_irq
.extern hal_frq
.extern _start
.global vector
vector:
    ldr pc, lmosem_restart_l
    ldr pc, lmosem_undefins_l
    ldr pc, lmosem_swi_l
    ldr pc, lmosem_prefabt_l
    ldr pc, lmosem_dataabt_l
    ldr pc, lmosem_bugdie_l
    ldr pc, lmosem_irq_l
    ldr pc, lmosem_frq_l @把hal_frq的地址装载进pc寄存器中完成跳转
.align 4
lmosem_restart_l: .word _start @init.S中，初始化向量
lmosem_undefins_l: .word hal_undefins @异常向量
lmosem_swi_l: .word hal_swi @陷入向量
lmosem_prefabt_l: .word hal_prefabt @异常向量
lmosem_dataabt_l: .word hal_dataabt @异常向量
```

```

lmosem_bugdie_l:    .word hal_bugdie      @保留向量
lmosem_irq_l:       .word hal_irq        @中断向量
lmosem_frq_l:       .word hal_frq        @快速中断向量由链接器填入hal_frq的实际地址
.end

```

代码清单 8-7 编译后，会被放到什么内存地址上，在第 6 章已经说明了，硬件要求这些向量要在固定的地址上，“ldr pc, xxxx”指令的原理前面章节中也已经说明了。hal_bugdie 是保留向量，在实际中没什么作用，因为硬件对向量地址有要求，所以它一定要占一个存储字的空间。可能有多个异常向量，它们的中断类型也都是异常，hal_frq 是快速中断向量，而 hal_swi 则是陷入向量。

关于中断向量，就说到这里，因为前面章节中已经写得很明白了。下面就去实现 hal_irq、hal_swi 等这些标号处的代码。

8.3.3 保存 CPU 上下文

保存 CPU 上下文是中断处理的首要任务，因为异常、中断、陷入的开始执行，也意味着内核的代码开始运行，内核代码的执行势必会用到 CPU 的寄存器，可是这一时刻的 CPU 寄存器中的值是当前进程的，为了处理完中断能正确返回或者下一次能让这个进程继续运行，就必须保存好当前 CPU 寄存器中的值，即 CPU 上下文。

开始写代码之前，先在 hal 目录中另建一个 interrupt.s 汇编文件。该文件是中断处理的入口代码文件。建立好文件后，还要清楚下面两件事：

1) CPU 的那套寄存器是保存在内核栈空间中的，内核栈可能现在还不明白，等到了后面的进程章节就明白了，现在只要明白它和普通栈一样，就是一块内存空间。

2) ARMCPU 在进入不同的中断向量时会自动进入不同的 CPU 工作模式，例如，进入 irq 中断向量时，ARMCPU 切换至 IRQ 模式，进入 swi 陷入向量时会自动进入特权管理模式，关于 CPU 的工作模式请参阅第 3 章。而 LMOSEM 内核的设计只使用两种 CPU 工作模式：用户模式和特权管理模式，当 CPU 进入别的中断向量时，我们首先切换到特权管理模式，再进行中断处理。

ARM920T 有 8 个向量，除了复位向量和保留向量外，还有 6 个向量，所以要写 6 段代码，如代码清单 8-8[xxxx/lmosem/hal/interrupt.s] 所示。

代码清单8-8 保存和恢复CPU的上下文并调用相应的C函数

```

.section .lmosem_hal.intvect,"ax" @定义一个叫.lmosem_hal.intvect的节
.code 32 @表明这是32位的ARM状态代码
.align 4 @按字对齐
.....

.equ ALLREGS_STKSZ,64
.equ STKOFFSET_PC,60
.equ NPCREGS_STKSZ,60
.equ CPUMODE_MASK,31

```

```

.equ CPUSVE_MODE,19
.equ MSTK_OFF,12
.equ NINT_SVEMODE,0xd3
.equ NINT_IRQMODE,0xd2
.equ NINT_FRQMODE,0xd1
.equ NINT_ABTMODE,0xd7
.equ NINT_UNDMODE,0xdb
hal_undefins:
    msr    cpsr_c,#NINT_SVEMODE
    sub    sp,sp,#ALLREGS_STKSZ
    stmia  sp,{r0-lr}^
    msr    cpsr_c,#NINT_UNDMODE
    sub    r1,lr,#0
    mrs    r0,spsr
    msr    cpsr_c,#NINT_SVEMODE
    str    r1,[sp,#STKOFFSET_PC]
    stmfd  sp!,{r0,lr}
    mov    r0,sp
    bl     hal_undefins_distr
    ldmfd  sp!,{r0,lr}
    mrs    spsr,r0
    ldmia  sp,{r0-lr}^
    add    sp,sp,#NPCREGS_STKSZ
    ldmfd  sp!,{pc}^
hal_swi:
    sub    sp,sp,#ALLREGS_STKSZ
    stmia  sp,{r1-lr}^
    mrs    r1,spsr
    str    lr,[sp,#STKOFFSET_PC]
    stmfd  sp!,{r1,lr}
    mov    r1,sp
    bl     hal_swi_distr
    ldmfd  sp!,{r1,lr}
    mrs    spsr,r1
    ldmia  sp,{r1-lr}^
    add    sp,sp,#NPCREGS_STKSZ
    ldmfd  sp!,{pc}^
hal_pfabt:
    .....
hal_dataabt:
    .....
hal_bugdie:
    b . @死循环
hal_irq:
    msr    cpsr_c,#NINT_SVEMODE
    sub    sp,sp,#ALLREGS_STKSZ

```



```

stmia sp,{r0-lr}^
msr cpsr_c,#NINT_IRQMODE
sub r1,lr,#4
mrs r0,spsr
msr cpsr_c,#NINT_SVEMODE
str r1,[sp,#STKOFFSET_PC]
stmfd sp!,{r0,lr}
mov r0,sp
bl hal_irq_distr
ldmfd sp!,{r0,lr}
msr spsr,r0
ldmia sp,{r0-lr}^
add sp,sp,#NPCREGS_STKSZ
ldmfd sp!,{pc}^
hal_frq:
.....

```

由于是汇编代码，所以感觉代码很长，事实上有些支持 ARM 的 C 编译器，能自动生成类似的代码，但是没有直接用汇编代码灵活。这里选择三个代码段作为例子来看一看，分别是 `hal_undefins`（未定义指令异常）、`hal_swi`（swi 指令陷入）、`hal_irq`（外部设备中断）。

先来看看第一段代码 `hal_undefins`（未定义指令异常）。当 CPU 遇到不认识的指令时，就会跳转这段代码上运行。这段代码分为 4 个部分：保存用户模式下的 R0 ~ R14 寄存器、保存发生异常前的 PC 和 CPSR 寄存器、调用一个 C 函数、恢复先前保存的那些寄存器以达到异常处理的返回。保存用户模式下的 R0 ~ R14 寄存器的部分，如代码清单 8-9[xxxx/lmosem/hal/interrupt.s] 所示

代码清单8-9 保存用户模式下的R0~R14寄存器

```

msr cpsr_c,#NINT_SVEMODE @切换至特权管理模式并关中断，有_c后缀所以只是改变CPSR寄存器的@
低8位，同时由于CPSR模式位的特定编码导致这一行代码可以不用读-改-写，也不必占用其他寄存器
sub sp,sp,#ALLREGS_STKSZ@把管理模式下的SP（即内核栈寄存器）减去64字节
stmia sp,{r0-lr}^ @对于stmia指令有^后缀表示操作的是用户模式的r0~r14这15个寄存器

```

如果不明白 CPU 模式切换代码，可参阅前面章节 CPU 的 CPSR 寄存器。由于 CPU 发生未定义指令异常时，会自动进入未定义指令异常模式，同时将当前 CPU 的 PC 和 CPSR 寄存器的值，放入未定义指令异常模式下的 LR 和 SPSR 寄存器中，所以要切换回去，把两个寄存器的值取出来，然后又回到特权管理模式下，把这两个寄存器中的值保存进内核栈，如代码清单 8-10[xxxx/lmosem/hal/interrupt.s] 所示。

代码清单8-10 切换CPU模式并保存PC和CPSR寄存器

```

msr cpsr_c,#NINT_UNDMODE @切换至未定义指令异常模式
sub r1,lr,#0 @把未定义指令异常模式的LR读取到r1中，即异常发生时的PC
mrs r0,spsr @把未定义指令异常模式的SPSR放进r0中，即异常发生时的CPSR

```

```

msr    cpsr_c, #NINT_SVEMODE @切换回特权管理模式
str    r1, [sp, #STKOFFSET_PC] @把r1写入sp+STKOFFSET_PC地址的内核栈中
stmfd  sp!, {r0, lr} @把以前的CPSR和特权管理模式的LR压栈, 并更新SP内核栈指针

```

如果不明白, 请参阅前面相关章节。接下来是调用一个 C 函数, 先不管这个 C 函数是干什么的, 如代码清单 8-11[xxxx/lmosem/hal/interrupt.s] 所示。

代码清单8-11 调用hal_undefins_distr函数

```

mov    r0, sp @把当前SP写进r0, 相当于给C函数传递一个参数
bl     hal_undefins_distr @bl调用指令会把返回地址写入LR中, 这就是上面代码中要把LR压栈的原因

```

当 C 函数返回时就要恢复先前保存的寄存器以达到异常处理的返回, 如代码清单 8-12[xxxx/lmosem/hal/interrupt.s] 所示。

代码清单8-12 恢复先前保存的寄存器

```

ldmfd  sp!, {r0, lr} @从内核栈中弹出两个字到r0、LR中, r0中的值就是先前压入的前CPSR
msr     spsr, r0 @把前CPSR放入当前CPU工作模式下的SPSR中
ldmia  sp, {r0-lr}^ @从SP指向的内核栈中加载15个字到用户模式的r0~r14寄存器中
add     sp, sp, #NPCREGS_STKSZ @把内核栈的指针加上60, 使之指向前PC存放的地址
ldmfd  sp!, {pc}^ @从内核栈中弹出一个字到PC中同时将SPSR复制到CPSR中, 并且同时更新内核栈
@SP使之平衡。因为更新了CPSR所以会导致CPU切换回发生异常前的工作模式, 所以实现了异常处理后的返回

```

再来看看第二段代码 hal_swi (swi 指令陷入), CPU 一旦运行了, swi 指令就会跳转到 hal_swi 代码处运行, 如代码清单 8-13[xxxx/lmosem/hal/interrupt.s] 所示。但是由于 CPU 运行了 swi 指令后会直接进入 CPU 的特权管理模式, 当然, 运行 swi 指令后的 PC 和 CPSR 也直接自动保存在 CPU 特权管理模式下的 LR 和 SPSR 中, 所以这次就不需要来回切换模式的那部分指令了。

代码清单8-13 保存和恢复CPU的上下文并调用相应的C函数

```

hal_swi:
sub     sp, sp, #ALLREGS_STKSZ @把特权管理模式下的SP (即内核栈指针) 减去64字节
stmia  sp, {r1-lr}^ @对于stmia有^后缀表示操作的是用户模式下r1~r14这14个寄存器
mrs    r1, spsr @把SPSR (前CPSR) 写入r1中
str    lr, [sp, #STKOFFSET_PC]^ @把LR (前PC) 写入sp+ STKOFFSET_PC为地址的内核栈中
stmfd  sp!, {r1, lr}
mov    r1, sp
bl     hal_swi_distr @调用C函数hal_swi_distr
ldmfd  sp!, {r1, lr}
msr    spsr, r1
ldmia  sp, {r1-lr}^ @恢复用户模式下的寄存器
add    sp, sp, #NPCREGS_STKSZ @同样让sp指向保存前PC的地址上
ldmfd  sp!, {pc}^ @带模式切换的返回, 并且会平衡内核栈

```

代码清单 8-13 中, 没有保存 r0 寄存器, 因为 swi 指令的运行是同步的, r0 中保存的是系统服务号, 每个系统服务程序都会用 r0 返回一个状态码, 所以 r0 不被保存和恢复。

来看最后一段代码 `hal_irq` (外部设备中断)。当设备发生了中断, 中断控制器会给 CPU 的 `irq` 引脚一个电子信号, CPU 检测到 `irq` 引脚上有电子信号时, 并且当前 CPSR 中的 `i` 位没有置 1, 这时 CPU 就进入 IRQ 模式并同时把 PC 和 CPSR 存放到 IRQ 模式中的 LR 和 SPSR 中, 然后跳转到 `irq` 向量进而转到 `hal_irq` 代码处开始执行, 如代码清单 8-14[xxxx/lmosem/hal/interrupt.s] 所示。由于 CPU 会自动切换到 IRQ 模式, 所以这里也要和未定义指令异常一样, 需要来回切换 CPU 的工作模式, 只不过是 IRQ 模式和特权管理模式间来回切换。

代码清单8-14 保存和恢复CPU的上下文并调用相应的C函数

```
hal_irq:
    msr    cpsr_c, #NINT_SVEMODE
    sub    sp, sp, #ALLREGS_STKSZ
    stmia  sp, {r0-lr}^
    msr    cpsr_c, #NINT_IRQMODE
    sub    r1, lr, #4@由于处理器流水线的原因, 所以前PC的值减4才是正确的返回地址
    mrs    r0, spsr
    msr    cpsr_c, #NINT_SVEMODE
    str    r1, [sp, #STKOFFSET_PC] @保存调整好了的前PC到内核栈中
    stmfd  sp!, {r0, lr}
    mov    r0, sp
    bl     hal_irq_distr @调用hal_irq_distr C函数
    ldmfd  sp!, {r0, lr}
    msr    spsr, r0
    ldmia  sp, {r0-lr}^ @恢复用户模式下的寄存器
    add    sp, sp, #NPCREGS_STKSZ @同样让SP指向保存前PC的地址上
    ldmfd  sp!, {pc}^@带模式切换的返回, 并且会平衡内核栈
```

其他中断处理保存 CPU 上下文的代码也是一样的道理。相信有了以上三个例子, 理解起来并不困难, 保存 CPU 上下文是为了以后能恢复, 所以在这里同时把恢复 CPU 上下文的代码也写好了, 不难发现每段代码中都调用了 C 函数, 下面就去实现这些 C 函数。

8.3.4 中断主分派例程

前面, 从中断向量到一个代码段, 该代码段保存好 CPU 当前工作模式下的所有寄存器中的值后, 都会调用一个 C 函数, 这个 C 函数是干什么的呢, 其实它不干什么实际的工作, 它只是一个框架函数, 而在它的内部才调用实际的中断处理函数, 所以称它为中断主分派例程。

在开始写中断主分派例程之前, 先在 LMOSEM 的工程目录下的 `hal` 目录中新建一个 C 程序文件 `intabtdistr.c` 和相关的头文件, 按照前面章节所述的方法把相关的头文件组织好, 并让 `intabtdistr.c` 加入到编译头文件中相关的编译宏中。

好了, 开始写代码, 如代码清单 8-15[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-15 中断、异常主分派函数

```

void hal_undefins_distr(void* sframe)
{
    //发生未定义指令异常，就死机，代码都是我们写的
    hal_sysdie("undefins runing!! ");
    return;
}

sysstus_t hal_swi_distr(uint_t swinr, void* sframe)
{
    return krlservice(swinr, sframe); //调用内核相关的服务代码，这里不必清楚这个函数
}

void hal_prefabt_distr(void* sframe)
{
    printfk("CP15 c5:%x c6:%x\n\r",
        hal_read_cp15regs(CP15_C5R), hal_read_cp15regs(CP15_C6R));
    hal_dbug_print_reg((intstkregs_t*)sframe);
    //打印所有寄存器后，死机，为了简单不做具体处理
    hal_sysdie("pref abt runing!! ");
    return;
}

void hal_dataabt_distr(void* sframe)
{
    printfk("CP15 c5:%x c6:%x\n\r",
        hal_read_cp15regs(CP15_C5R), hal_read_cp15regs(CP15_C6R));
    hal_dbug_print_reg((intstkregs_t*)sframe);
    //打印所有寄存器后，死机，为了简单不做具体处理
    hal_sysdie("data abt runing!! ");
    return;
}

void hal_bugdie_distr(void* sframe)
{
    //对于保留中断当然死机，事实上不会执行到该函数
    hal_sysdie("bug die runing!! ");
    return;
}

void hal_irq_distr(void* sframe)
{
    uint_t intoset = hal_retn_intnr();
    switch(intoset) //这是为了处理子中断源和外部中断源的，在这里不必清楚细节
    {
        case EINT4_7:
            hal_eint_distr(sframe, intoset, EI4_7_PNDBTS, EI4_7_PNDBTE);
            break;
        case EINT8_23:
            hal_eint_distr(sframe, intoset, EI8_23_PNDBTS, EI8_23_PNDBTE);
            break;
    }
}

```

```

    case INT_CAM:
        hal_sint_distr(sframe, intoiset, ICAM_PNDBTS, ICAM_PNDBTE);
        break;
    case INT_WDT_AC97:
        hal_sint_distr(sframe, intoiset, IACWDT_PNDBTS, IACWDT_PNDBTE);
        break;
    case INT_UART2:
        hal_sint_distr(sframe, intoiset, IUART2_PNDBTS, IUART2_PNDBTE);
        break;
    case INT_UART1:
        hal_sint_distr(sframe, intoiset, IUART1_PNDBTS, IUART1_PNDBTE);
        break;
    case INT_UART0:
        hal_sint_distr(sframe, intoiset, IUART0_PNDBTS, IUART0_PNDBTE);
        break;
    case INT_ADC:
        hal_sint_distr(sframe, intoiset, IADC_PNDBTS, IADC_PNDBTE);
        break;
    default:
        hal_int_distr(sframe, intoiset);
        break;
}

hal_clear_intpnd(intoiset); //清除主中断源挂起寄存器中的相关位
//检查是否需要进程调度, 这里不必清楚该函数的细节
krlsched_chkneed_pmptsched();
return;
}

void hal_frq_distr(void* sframe)
{
    hal_sysdie("frq runing!! "); //LMOSEM不使用fiq中断, 所以死机
    return;
}

```

代码清单 8-15 中, `hal_undefins_distr`、`hal_frq_distr`、`hal_bugdie_distr` 三个函数的运行直接会导致 LMOSEM 死机。第一, 所有的 LMOSEM 代码都是我们自己写的, 不可能发生未定义指令异常, 发生了我们也可以在代码中修正。第二, LMOSEM 会让所有设备的中断源全部以 `irq` 的方式触发中断, 不可能发生 `fiq` 中断。第三, 保留中断向量是不会执行的, 这样做是为了让代码更加规范化。最后我们是探索操作系统内核原理的, 当然是越简单越好。

`hal_prefabt_distr`、`hal_dataabt_distr` 一个是预取异常分配器, 另一个是数据异常分配器。在所有代码都是我们自己写的情况下, 这两种异常也不太可能发生, 除非代码有问题。这两个函数除了调用死机函数外, 还调用 `hal_dbug_print_reg` 函数完成了打印发生异常前的所有寄存器中的值, 这有助于我们调试内核代码, 如代码清单 8-16[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-16 打印发生异常前的所有寄存器的值

```

void hal_dbug_print_reg(intstkrege_t* intstkp)
{
    printfk("USR_REG r0:%x\n\r",intstkp->r0);
    printfk("USR_REG r1:%x\n\r",intstkp->r1);
    printfk("USR_REG r2:%x\n\r",intstkp->r2);
    printfk("USR_REG r3:%x\n\r",intstkp->r3);
    printfk("USR_REG r4:%x\n\r",intstkp->r4);
    printfk("USR_REG r5:%x\n\r",intstkp->r5);
    printfk("USR_REG r6:%x\n\r",intstkp->r6);
    printfk("USR_REG r7:%x\n\r",intstkp->r7);
    printfk("USR_REG r8:%x\n\r",intstkp->r8);
    printfk("USR_REG r9:%x\n\r",intstkp->r9);
    printfk("USR_REG r10:%x\n\r",intstkp->r10);
    printfk("USR_REG r11:%x\n\r",intstkp->r11);
    printfk("USR_REG r12:%x\n\r",intstkp->r12);
    printfk("USR_REG r13:%x\n\r",intstkp->r13);
    printfk("USR_REG r14:%x\n\r",intstkp->r14);
    printfk("SVE_REG lr:%x\n\r",intstkp->s_lr);
    printfk("SVE_REG spsr:%x\n\r",intstkp->s_spsr);
    //下面打印了当前进程的一些信息,先不管它,只看上面打印寄存器值的代码就行了
    printfk("CSP_REG sp:%x INTPND:%x\n\r",
        hal_read_currmodesp(),hal_io32_read(INTPND_R));
    printfk("CCR_REG cpsr:%x INTOFST:%x\n\r",hal_read_cpufg(),hal_retn_intnr());
    thread_t* prev=krlsched_retn_currthread();
    printfk("CURR_THREAD:%x CURR_THREAD_KSTKTOP:%x\n\r",
        (uint_t)prev,(uint_t)prev->td_krlstktp);
    return;
}

```

代码清单 8-16 中,主要调用 printfk 函数完成打印的,这个函数不必多说,关键是要理解 intstkrege_t 数据结构。intstkrege_t 数据结构定义在 cpu_t.h 文件中,如代码清单 8-17[lmosem/include/halinc/cpu_t.h] 所示。

代码清单8-17 intstkrege_t数据结构

```

typedef struct s_INTSTKREGS
{
    reg_t s_spsr;//相当于发生异常前的CPSR
    reg_t c_lr;//当前CPU工作模式下的LR
    reg_t r0;
    reg_t r1;
    reg_t r2;
    reg_t r3;
    reg_t r4;
    reg_t r5;

```

```

reg_t r6;
reg_t r7;
reg_t r8;
reg_t r9;
reg_t r10;
reg_t r11;
reg_t r12;
reg_t r13;
reg_t r14;
reg_t s_lr; //相当于发生异常前的PC
}intstkregs_t;

```

代码清单 8-17 中的 `reg_t` 类型就是 `u32_t` 类型。`hal_dbug_print_reg` 函数的参数是调用它的异常分配器函数的参数 `sframe` 直接传递进来的，那么 `sframe` 又是谁传递的呢，返回 8.3.3 节保存 CPU 寄存器值的代码中，它在调用一些分配器函数之前都把内核栈指针寄存器 `SP`，赋给了 `r0` 或者 `r1` 寄存器，从前面的章节中知道 ARM 体系下的 C 语言调用约定，是用 `r0 ~ r3` 四个寄存器传递前四个参数的，自然 `SP` 中的值就成了 `intstkregs_t` 类型的指针，那么自然栈中数据的格式，也就是和 `intstkregs_t` 数据结构是一致的。

再来看看 `hal_irq_distr` 函数，它是设备中断分配器函数，设备中断的处理就是从这个函数开始的，所以必须要写好它。它首先调用 `hal_retn_intnr` 函数获取中断源，即是哪个设备发生了中断，然后用 `switch` 语句，依次判断该中断源下是否有子中断源或者外部中断源，如果有就进一步调用相关的函数来处理。最后检查系统是否需要执行进程调度。这里先不深入到其内部，具体的中断处理后面再探索，这里理解框架即可。

从前面的代码看，中断处理才刚刚开始，下面就去确定具体的中断源。

8.3.5 确定中断源

从前面的章节中了解到，S3C2440A 中的中断控制器上有一些中断源，是采用级联方式连接的，即主中断源上可能连接着几个子中断源或者外部中断源。而中断控制器中的中断偏移寄存器只能呈现是哪个主中断源发生了中断，所以这给中断处理程序带来了麻烦。

在 8.3.4 节中，`hal_irq_distr` 函数已经调用 `hal_retn_intnr` 函数获取了主中断源，可是有了主中断源后，并不能立即进行中断的处理，还要看看这个主中断源下有没有子中断源或者外部中断源，如果有，就要进一步确定是哪个子中断源或者是哪个外部中断源之后才能进行中断处理。

下面就去实现这部分代码。在这里依然列出 IRQ 中断分配器中的框架代码，便于观察，如代码清单 8-18[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单 8-18 确定中断源

```

void hal_irq_distr(void* sframe)
{

```

```

uint_t intoset=hal_retn_intnr();//获取主中断源
switch(intoset)//依次判断和处理,子中断源和外部中断源
{
    case EINT4_7:
        hal_eint_distr(sframe,intoset,EI4_7_PNDBTS,EI4_7_PNDBTE);
        break;
    case EINT8_23:
        hal_eint_distr(sframe,intoset,EI8_23_PNDBTS,EI8_23_PNDBTE);
        break;
    case INT_CAM:
        hal_sint_distr(sframe,intoset,ICAM_PNDBTS,ICAM_PNDBTE);
        break;
    case INT_WDT_AC97:
        hal_sint_distr(sframe,intoset,IACWDT_PNDBTS,IACWDT_PNDBTE);
        break;
    case INT_UART2:
        hal_sint_distr(sframe,intoset,IUART2_PNDBTS,IUART2_PNDBTE);
        break;
    case INT_UART1:
        hal_sint_distr(sframe,intoset,IUART1_PNDBTS,IUART1_PNDBTE);
        break;
    case INT_UART0:
        hal_sint_distr(sframe,intoset,IUART0_PNDBTS,IUART0_PNDBTE);
        break;
    case INT_ADC:
        hal_sint_distr(sframe,intoset,IADC_PNDBTS,IADC_PNDBTE);
        break;
    default:
        hal_int_distr(sframe,intoset);
        break;
}
hal_clear_intpnd(intoset);    //清除主中断源挂起寄存器中的相关位
krlsched_chkneed_pmptsched();//检查是否需要进程调度,这里不必清楚该函数
return;
}

```

代码清单 8-18 中 `hal_irq_distr` 函数首先获取主中断源, `hal_retn_intnr` 函数就是读取中断偏移寄存器的。因为主中断源上只有几个中断源,级联的有外部中断源和子中断源,所以可以用 `switch` 语句。如果在当前主中断源下,级联的是外部中断源就调用 `hal_eint_distr` 函数来处理,如果级联的是子中断源就调用 `hal_sint_distr` 函数来处理,否则就调用 `hal_int_distr` 函数来处理。下面我们分别写好这三个函数。

先来看看遇到外部中断源后,该如何处理。当 `intoset` 等于 `EINT4_7` 或者 `EINT8_23` 时,就调用 `hal_eint_distr` 函数来处理。我们来写好这个函数,如代码清单 8-19[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-19 处理外部中断源

```

void hal_eint_distr(void* sframe,uint_t mintnr,uint_t pndbts,uint_t pndbte)
{
    u32_t pnd=hal_io32_read(EINTPEND_R); //读取外部中断源挂起寄存器
    pnd&=EINTPEND_BITS_MASK;           //处理掉无关的位, 参阅外部中断源挂起寄存器
    for(uint_t bi=pndbts;bi<pndbte;bi++)
    {
        if(((pnd>>bi)&1)==1) //确定一个外部中断源上有无中断, 一旦有了中断, 相应的外部中断源
            //挂起寄存器中的位就会置1
        {
            hal_run_intflthandle(EINT_IFDNR(bi),sframe);
            hal_clear_srcpnd(EINT_IFDNR(bi));
        }
    }
    return;
}

```

hal_eint_distr 函数首先读取外部中断源挂起寄存器并进行处理, 所谓处理也就是把 pnd 中无关的位清 0, 因为外部中断源挂起寄存器中有许多位是保留的。处理好了之后, 就开始运行一个循环, 循环 pndbts 到 pndbte 次, 每循环一次, 让 pnd 右移 bi 位后与上 1, 并且判断其结果是否为 1。说白了, 这个循环就是扫描 pnd 中 pndbts 到 pndbte 位段中所有为 1 的位。pndbts 和 pndbte 是作为函数的参数传递进来的, 对于 EINT4_7, 传递进来的是 EI4_7_PNDBTS (值为 4) 和 EI4_7_PNDBTE (值为 8)。当我们回过头去看看外部中断源挂起寄存器时, 一切都明白了。而扫描 pnd 相关位段就是扫描外部中断源挂起寄存器中的相关位段。为什么要这样, 这是因为中断控制器, 不能唯一确定地给出是哪一个具体的外部中断源发生了中断, 所以, 只能通过扫描当前主中断源, 对应的外部中断源挂起寄存器中, 所有的相关位段, 可是, 又为什么要扫描与此对应的所有位段呢, 因为同一时刻可能有多个已经挂起的中断源, 所以不能漏掉, 要依次调用它们的中断处理程序。先不用管 if 语句内部的函数是干什么的, 这里只要明白这个循环和右移语句的含意就行了。

处理好了外部中断源, 还有子中断源呢, 例如, 代码清单 8-18 中, case 到 INT_UART0 后就调用 hal_sint_distr 函数来处理, 下面就来写好这个函数, 如代码清单 8-20[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-20 处理子中断源

```

void hal_sint_distr(void* sframe,uint_t mintnr,uint_t pndbts,uint_t pndbte)
{
    u32_t pnd=hal_io32_read(SUBSRCPND_R); //读取子中断源挂起寄存器
    pnd&=SUBSRCPND_BITS_MASK;           //处理掉无关的位, 参阅外部中断源挂起寄存器
    for(uint_t bi=pndbts;bi<pndbte;bi++)
    {
        if(((pnd>>bi)&1)==1) //确定一个子中断源上有无中断, 一旦有了中断, 相应的子中断源挂
            //起寄存器中的位就会置1
    }
}

```

```

    {
        hal_run_intflthandle(SINT_IFDNR(bi),sframe);
        hal_clear_srcpnd(SINT_IFDNR(bi));
    }
}
return;
}

```

和 `hal_eint_distr` 函数的原理一样，也是读取相应的中断源挂起寄存器，然后用循环扫描其位段，来确定这些中断源上是否发生了中断。只不过这里读取的是子中断源挂起寄存器，还有相应的位段也不同。对于 `INT_UART0`，所对应的子中断源挂起寄存器中的位段是 `IUART0_PNDBTS`（值为 0）和 `IUART0_PNDBTE`（值为 3）。如果不明白，回到前面看看子中断挂起寄存器中，第一个串口的三个中断源对应的位段就明白了。

如果既不是子中断源也不是外部中断源，那就是主中断源。如果是主中断源，就调用 `hal_int_distr` 函数来处理，下面就去写好它，如代码清单 8-21[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-21 处理主中断源

```

void hal_int_distr(void* sframe,uint_t mintnr)
{
    hal_run_intflthandle(MINT_IFDNR(mintnr),sframe);
    hal_clear_srcpnd(MINT_IFDNR(mintnr));
    return;
}

```

`hal_int_distr` 这个函数非常简单，直接用 `hal_irq_distr` 函数传递进来的 `intoset` 参数，因为这个参数就表示是哪个主中断源，直接使用就行了。

到这里，从硬件的角度来说，只是确定了实际发生中断的中断源，并未调用实际的中断处理程序，但同时也发现，在 `hal_eint_distr`、`hal_sint_distr`、`hal_int_distr` 函数中，都调用了 `hal_run_intflthandle` 函数，就是在这个函数中调用具体的中断处理程序的，运行完中断处理程序后，会调用 `hal_clear_srcpnd` 函数，清除相应中断源挂起寄存器中的相关位，使之能响应下一次中断。

至于 `hal_run_intflthandle` 函数，是如何调用具体的中断处理程序的，下面就去实现。

8.3.6 调用中断处理例程

确定了产生中断的中断源之后，就要调用与此对应的中断处理程序。中断处理程序在哪里呢，读者可能对 `intfltdsc_t` 和 `intserdsc_t` 这两个数据结构还记忆犹新。中断处理程序，实际上是个函数，这个函数的地址就保存在 `intserdsc_t` 中。而 `intserdsc_t` 是挂在 `intfltdsc_t` 中的。一个 `intfltdsc_t` 中可能有多个 `intserdsc_t`，因为多个设备可能共享同一根中断信号线。

根据以上描述，调用实际中断处理函数的过程如下：

第1步：根据中断源找到中断源描述符 `intfltdsc_t` 数据结构。

第2步：在找到的 `intfltdsc_t` 中查找 `intserdsc_t`，找到一个 `intserdsc_t` 就调用其中的中断处理函数。

第3步：回到第2步，继续在刚才的 `intfltdsc_t` 中查找下一个 `intserdsc_t` 并执行其中的中断处理函数，直到查找完所有的 `intserdsc_t`。

有了上面的步骤，写代码就容易多了。下面就去实现这个 `hal_run_intflthandle` 函数，如代码清单 8-22[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单8-22 调用实际的中断处理函数

```
void hal_run_intflthandle(uint_t ifdnr, void* sframe)
{
    intserdsc_t* isdscp;
    list_h_t* lst;
    intfltdsc_t* ifdscp = hal_retn_intfltdsc(ifdnr);
    if(ifdscp == NULL)
    {
        //如果没有中断源描述符就死机
        hal_sysdie("hal_run_intfdsc err");
        return;
    }
    list_for_each(lst, &ifdscp->i_serlist)           //遍历intfltdsc_t中的i_serlist链表
    {
        isdscp = list_entry(lst, intserdsc_t, s_list); //获取具体的intserdsc_t的指针
        isdscp->s_handle(ifdnr, isdscp->s_device, sframe); //调用具体的中断处理函数
    }
    return;
}
```

代码清单 8-22 中，首先调用 `hal_retn_intfltdsc` 返回一个中断源描述符，然后遍历 `intfltdsc_t` 中的 `i_serlist` 链表，获取其中的 `intserdsc_t` 的指针，最后调用 `intserdsc_t` 的 `s_handle` 中的中断处理函数。可是 `ifdnr` 是哪里来的，在 8.3.5 节中我们看到，它是由中断源挂起寄存器中的位号再经过几个宏转换后，传递给 `hal_run_intflthandle` 函数的。这几个宏如代码清单 8-23[xxxx/lmosem/include/halinc/halintupt_t.h] 所示。

代码清单8-23 中断源相关的宏

```
#define MINT_OFFSET 0
#define SINT_OFFSET 32
#define EINT_OFFSET 47
.....

#define MINT_IFDNR(x) (x+MINT_OFFSET)           //转换主中断源描述符在其数组中的下标
#define SINT_IFDNR(x) (x+SINT_OFFSET)           //转换子中断源描述符在其数组中的下标
#define EINT_IFDNR(x) (x+EINT_OFFSET)          //转换外部中断源描述符在其数组中的下标
```

对于代码清单 8-23, 回到初始化中断章节那里, 看看是如何把外部中断源、子中断源、主中断源平级映射成一个 `intfltdsc_t` 类型数组的, 就明白了。

那么对于几个设备共享一条中断信号线的情况, 究竟是哪个设备产生了中断, 这应该交给设备中断处理程序去处理, 内核只需要依次调用所有挂在这个中断源描述符上的中断处理函数即可。

好了, 到这里, `hal_run_intflthandle` 函数的完成, 意味着 LMOSEM 内核的中断处理的部分已经完成, 具体的中断处理函数是设备驱动程序编写的, 操作系统内核不可能对每个设备都了如指掌。

我们发现, LMOSEM 内核的中断处理框架很好地解决了一个中断源对应多个中断 (因为中断源的级连) 的情况, 也解决了一个中断源对应多个设备的情况。但是好像少了点什么, 少了点什么呢? 下面继续……

8.4 安装中断回调例程

一个计算平台上有许多设备, 每个设备都不同, 操作系统内核开发者, 不可能了解计算平台上的所有设备的细节。例如, 不同设备发生了中断, 虽然对中断的响应是相同的, 但是对于一个设备发生中断后的处理, 各有不同。所以把中断的响应交给内核, 把具体一个设备的中断处理交给设备驱动程序。

既然中断处理函数是由驱动程序编写的, 那么就要给驱动程序提供一个接口, 让其可以安装自己设备的中断处理函数, 或者是安装中断回调例程。下面我们就去干这件事。

设备回调例程的安装接口

为什么要提供设备回调例程的安装接口, 刚刚说了是因为设备的种类千差万别, 同时为了操作系统内核更具有通用性和扩展性, 所以把具体设备的中断处理交给具体设备的驱动程序。这样如果计算平台更换了更先进的设备, 只需要加载新版本的驱动程序就行了。

前面章节中, 我们已经设计了 `intserdsc_t` 数据结构, 这个数据结构中有个域就是用来保存设备回调例程地址的。于是, 安装设备回调例程的方法如下:

1) 提供一些信息: 设备描述符 (后面会有介绍)、设备回调例程地址、设备中断源对应的中断源描述符数组的下标。之所以要提供设备描述符, 是因为一个设备中断总是与具体的设备有关的。

2) 分配并实例化一个 `intserdsc_t` 数据结构变量, 把以上等信息填入其中。

3) 把这个 `intserdsc_t` 数据结构变量, 挂载到对应的 `intfltdsc_t` 中断源描述符中。

有了以上方法, 写起代码时就容易多了, 但是别急着写代码, 对中断管理这个组件来说, 它分为两个部分: 一个部分在内核的硬件相关层 (`hal`) 中; 另一个部分在内核的功能

层中，对于设备回调例程的安装接口，我们应该把它放在内核功能层中。所以先在“xxxx/lmosem/kernel”目录下新建一个C程序文件krlintupt.c和相应的头文件，并按照前面章节中的方法组织好这些文件。

有了krlintupt.c文件，就可以开始写代码实现上述步骤了，如代码清单8-24[xxxx/lmosem/kernel/krlintupt.c]所示。

代码清单8-24 安装设备回调例程的接口

```
intserdsc_t* krladd_irqhandle(void* device,intflt_handle_t handle,uint_t phyline)
{
    if(device==NULL||handle==NULL)
    { //如果设备描述符或者中断回调函数地址其中一个为NULL，则返回NULL表示出错
        return NULL;
    }
    intflt_dsc_t* intp=hal_retn_intflt_dsc(phyline); //返回中断源描述符
    if(intp==NULL)
    {
        return NULL; //如果没有中断源描述符，则返回NULL表示出错
    }
    intserdsc_t* serdscp=(intserdsc_t*)krlnew(sizeof(intserdsc_t)); //分配一个intserdsc_t
                                                                    //数据结构变量的空间
    if(serdscp==NULL)
    {
        return NULL; //如果分配空间失败，则返回NULL表示出错
    }
    intserdsc_t_init(serdscp,0,intp,device,handle); //初始化该intserdsc_t数据结构变量
    if(hal_add_ihandle(intp,serdscp)==FALSE)
    { //如果intserdsc_t加入到intflt_dsc_t中失败，则释放intserdsc_t数据结构变量的空间
        if(krldelete((adr_t)serdscp,sizeof(intserdsc_t))==FALSE)
        { //内存空间释放失败时，就死机，不过这种情况不太可能发生
            hal_sysdie("krladd_irqhandle ERR");
        }
        return NULL;
    }
    return serdscp; //返回intserdsc_t变量的地址，表示成功
}
```

代码清单8-24中的注释已经够明白了，首先是对参数进行了检察，然后以phyline为参数返回一个中断源描述符，调用内存管理接口动态分配了一个intserdsc_t数据结构变量的空间，接着调用intserdsc_t_init函数对这个空间进行了初始化，因为动态分配的内存空间必须要初始化。而hal_add_ihandle才是使intserdsc_t和intflt_dsc_t结合在一起的函数。可是函数在哪儿呢，还没写呢，现在就去写好它们，如代码清单8-25[xxxx/lmosem/hal/halintupt.c]所示。

代码清单8-25 注册设备回调例程

```

void intserdsc_t_init(intserdsc_t* initp, u32_t flg, intfltdsc_t* intfltp, void*
device, intflthandle_t handle)
{
    list_init(&initp->s_list);
    list_init(&initp->s_indevlst); //初始化两个链表
    initp->s_flg=flg;
    initp->s_intfltp=intfltp;      //把它属于哪个intfltdsc_t的地址,放进去
    initp->s_indx=0;
    initp->s_device=device;        //把它属于哪个设备描述符的地址,放进去
    initp->s_handle=handle;        //把回调函数的地址,放进去
    return;
}

bool_t hal_add_ihandle(intfltdsc_t* intdscp, intserdsc_t* serdscp)
{
    if(intdscp==NULL||serdscp==NULL)
    { //对intfltdsc_t、intserdsc_t进行检索,有一个为NULL则返回FALSE表示失败
        return FALSE;
    }
    cpuflg_t cpuflg;
    hal_spinlock_saveflg_cli(&intdscp->i_lock, &cpuflg); //加锁并关中断
    list_add(&serdscp->s_list, &intdscp->i_serlist); //把intserdsc_t挂入i_serlist链表中
    intdscp->i_sernr++; //增加表示intserdsc_t个数的计数变量
    hal_spinunlock_restflg_sti(&intdscp->i_lock, &cpuflg); //开锁并恢复中断
    return TRUE; //返回TRUE表示成功
}

```

intserdsc_t_init、hal_add_ihandle 这两个函数非常简单,不过有一点需要注意,在把intserdsc_t挂入intfltdsc_t的i_serlist链表中时要加锁,还是前面多次提到的,可能多个设备共享同一根中断信号线,也许在同一时刻下,内核正在开始调用一个设备的中断处理函数,而另一个设备驱动程序正在向其中加入另一个设备的中断处理程序,读者可能会说目前情况下不可能,是的,可是如果LMOSEM内核将来要支持内核级可抢占或者多CPU时,问题就来了,所以不如在写每个函数时,多下点工夫。

为了以后让驱动程序最少化调用内核硬件相关层的函数,因为这里的函数在不同的计算平台下可能会改变,一旦改变,驱动程序就得重写了。读者可能会疑问,计算平台都改变了,旧的驱动程序还有什么用,因为不同的计算平台可能会用到相同的设备。

前面章节中,可以发现,中断控制器可以开启、关闭单个设备的中断信号通路,所以我们还要写两个函数:开、关具体设备中断信号通路的接口,如代码清单8-26[xxxx/lmosem/kernel/krlintupt.c]所示。

代码清单8-26 开、关具体设备中断信号通路的接口

```

drvstus_t krlenable_intline(uint_t ifdnr)

```

```

{
    return hal_enable_intline(ifdnr);
}
drvstus_t krldisable_intline(uint_t ifdnr)
{
    return hal_disable_intline(ifdnr);
}

```

代码清单 8-26 中的 `krlenable_intline`、`krldisable_intline` 两个函数，是直接对 LMOSEM 内核硬件相关层的 `hal_enable_intline`、`hal_disable_intline` 两个函数的封装，而这两个函数，在前面章节中已经介绍过了。

一直以来，我们都是在写中断管理的代码，从来没有测试代码的正确性，那是因为我们还没有驱动模型，不能装载设备驱动程序。从内核的角度来说，还没有设备呢，没有设备哪来的中断，没有中断怎么测试中断管理代码呢，等后面章节中完成了驱动模型，写好了第一个设备驱动程序，再一起来测试中断管理的代码。

到这里，对于驱动程序安装其设备中断处理的回调函数和用于开启或者关闭设备中断信号通路的接口就完成了，同时也宣告 LMOSEM 内核的中断管理组件完成了，或许还有许多不完善的地方，但是基本上可以工作了，对于探索操作系统内核的原理，足够了。

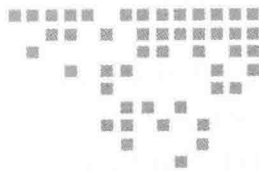
8.5 小结

又到了小结，意味着中断管理这一章节的结束，从此我们研究的操作系统内核 LMOSEM，可以响应和处理异常、中断、陷入了。

异常、中断、陷入，广义上都可以理解为“突发事件”。从前面章节中明白了，为了处理这种“突发事件”，CPU 和中断控制器，共同提供了一些基础的硬件机制。

但处理这种“突发事件”，硬件只是自动完成了最基础、最必要的动作。至于剩下的事情就是软件该做的，不管是异常还是中断或是陷入，软件首要的任务就是保存发生这种“突发事件”之前 CPU 寄存器中的值，由于 CPU 的实现和 LMOSEM 内核的设计，保存 CPU 寄存器中的值，我们是大费周折。接着是调用异常、中断、陷入的分发器函数，对于中断，还需要进一步地确定设备中断源，然后调用具体的设备中断处理函数。但由于设备之间差别很大，我们把响应中断的部分交给内核，把具体设备的中断处理，交给设备驱动程序，并给驱动程序，提供了安装中断处理函数的接口，同时为了操作中断控制器，还写了一些其他辅助函数。

以上就是本章中我们实现的功能，虽不多，但至关重要。LMOSEM 内核的又一个基础设施被实现了。虽然简单，但能工作，等有时间了再来完善它也不迟。现在休息一下，还有很多事情等着我们去做呢。



驱动模型

在实现了内存管理、中断管理之后，现在就要去实现一个新的 LMOSEM 内核组件——驱动模型。在理论教材中通常看到的设备管理、IO 系统等，就和这个有一点点关系。它们通常写了一堆让人发晕的概念，那些概念对于我们这种工程实践派来说，看上去好像是高大上，实则是一堆废话。

驱动模型是什么，先不定义。首先从操作系统内核如何管理设备开始，了解设备类型及设备驱动程序是干什么的。然后我们接着为管理设备和编写设备驱动程序设计一些数据结构。然后去看看，操作系统内核应该为设备驱动程序提供一什么基础设施，并实现它们。最后我们去亲自写几个设备的驱动程序。做完了这些，一定能够明白什么是驱动模型以及它是干什么的。

9.1 操作系统内核如何管理设备

操作系统内核是什么，从不同角度，有不同的答案。如果把计算机内部所有的设备和数据都描述成资源，操作系统内核无疑是这些资源的管理者。既然设备也是一种资源，如何高效管理它们，以便提供给应用进程使用和操作，是操作系统内核的重要任务。

9.1.1 分权而治

管理设备当然是为了更好地操作和使用设备。操作和使用设备就要对设备进行编程，对设备进行编程就要从设备在计算平台中的组织方式开始说起。

现代计算机早已不只是处理计算任务，还可以呈现图像、音频，远程计算机通信，存

储大量数据，以及和用户交互。所以计算机内部需要处理图像、音频、网络、存储、交互的设备。

现代计算机内部的设备都是用编程来控制的，即执行相应的程序指令来控制相应的设备，而执行相关程序指令的就是 CPU。相比以前的计算机设备，现代的计算机内部的一些设备大大减轻了 CPU 的负担，因为现代的计算机设备，都有为其而设计的、非常专业且智能的设备控制器。有些设备控制器本身就是一台小型的计算机。尽管如此，还是需要 CPU 给设备发送相关指令，通知设备什么时间开始工作及做什么工作。发送设备指令的负担对于 CPU 来说，虽然很小，但还是始终要做。因此我们可以画出如图 9-1 所示的计算机内部的设备结构图。

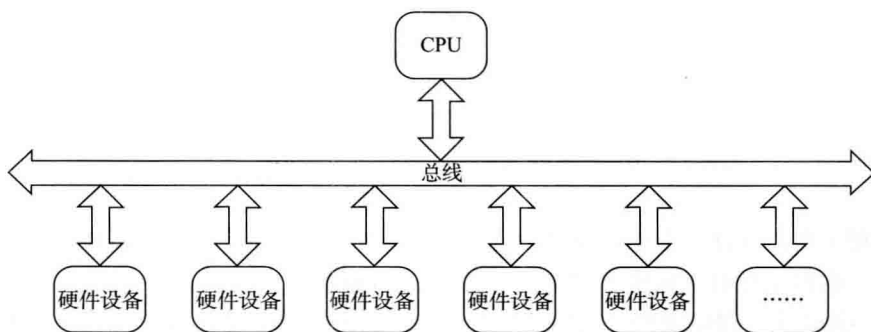


图 9-1 一般性现代计算机内部设备结构

图 9-1 画得非常简单，具体的物理计算平台可能与此不同，例如，我们的 mini2440 开发板，有很多设备和总线都是封装在一个芯片内部的，PC 上可能有多条总线，总线下面还有子总线，设备可以随时拔插。但不管在物理上存在的形式怎么样，设备在逻辑上的连接结构都和图 9-1 一样。

回到操作系统内核的角度，操作系统内核要控制这些设备，就要包含每个设备的控制代码。如果我们的操作系统内核被设计为通用、可移植的内核，那是相当可怕的。试想一下，这个世界上有这么多计算平台，其中有如此多样的设备，操作系统内核代码得多庞大，操作系统内核代码越庞大就越危险，因为其中有一行代码有问题，整个操作系统就崩溃了。可是只有这些问题吗，当然不是，具体如下：

1) 操作系统内核开发人员，不可能罗列世界上所有计算平台上的设备，并为其写一套控制代码。

2) 并不是所有的设备都是开放的。为了商业目的，有很多设备厂商并不愿意公开相关设备的编程细节。就算是操作系统内核开发人员想为其写控制代码也不行。

3) 如果设备更新换代，用户更新设备，就要重写设备的控制代码，然后重新编译操作系统内核。这样的话，操作系统内核开发人员和用户都可能受不了。

以上三点，足以证明，这种方案根本不可取。

既然操作系统内核无法包含所有的设备控制代码，那就索性不包含或者只包含最基本、最通用的设备控制代码。这样，操作系统内核就可以非常通用，非常精巧。但是要控制和操作设备就必须要有设备的相关控制代码才行。所以我们就想到把设备控制代码独立出来，与操作系统内核分开、独立开发，设备控制代码可由设备厂商或者其他开发人员开发。这样，每个设备就对应一个设备控制代码模块，操作系统内核要控制哪个设备，就加载哪个设备的控制代码模块，不使用该设备了，就可以删除这个设备控制代码模块。这种动态加载设备控制代码模块的方法，给操作系统内核带来了巨大的灵活性。设备厂商在发布新设备时，只要随着发布一个与此相关的设备控制代码模块就行了。

操作系统内核实现控制、操作设备的通用部分代码，而把具体的设备控制代码分离出来。使操作系统内核具有了灵活性和扩展性，同时也达到了对设备分权而治的目的。当然，这也带来了问题，如设备控制代码如何编写、设备控制代码应该提供一些什么功能、操作系统内核如何找到一个设备控制代码模块等。所以我们必须制定一套开发设备控制代码的规范，而这正是本章我们要解决的问题。

这里只要明白，因为上面种种原因，导致不能在操作系统内核里包含所有设备的控制代码，所以我们要把设备控制代码独立出来，需要时操作系统内核才会去加载。对设备分权而治，这是一个好方案，但是也带来了一些问题，这些问题我们在后面的章节中会一步步解决。

9.1.2 设备类型

现代计算机内部的设备很多，有图形设备、音频设备、网络设备、影像设备、存储设备、各种输入输出设备、各种传感器设备，还有连接这些设备的各种总线等，可谓是种类繁多，功能各异。

在开始对设备分类之前，不妨先思考一些问题：操作系统内核所感知的设备，一定要是物理存在的吗？一定要是与物理设备一一对应的吗？例如，存储设备，操作系统内核其实不必管它是机械硬盘，或是 SSD，还是 TF 卡。再如，一个设备控制代码模块，它向操作系统内核表明它是存储设备，但它完全有可能分配一块内存空间，在这块内存空间中存储数据，不必访问真正的存储设备。甚至可以为了完成某些特殊的功能，而编写一种虚拟的设备控制代码。所以操作系统内核所感知的设备，并不需要和物理设备一一对应，这完全取决于设备控制代码自身的行为。操作系统内核所定义的设备，可称为内核设备或者逻辑设备，只是对物理计算平台中几种类型设备的一种抽象。

我们依然以 mini2440 开发板为原型，对设备进行分类，所以分出的类型难免不够通用，这只是为了实现起来更简单，以后我们可以将其完善。下面以整数的形式来表示设备类型，即对不同类型的设备进行编码，如代码清单 9-1[xxxx/lmosem/include/knlinc/krldevice_t.h]所示。

代码清单9-1 设备类型

```

#define NOT_DEVICE 0           //不表示任何设备
#define AMBABUS_DEVICE 1       //AMB总线设备
#define AHBBUS_DEVICE 2       //AHB总线设备
#define APBBUS_DEVICE 3       //APB总线设备
#define BRIDGE_DEVICE 4       //总线桥接器设备
#define CPUCORE_DEVICE 5       //CPU设备, CPU也是设备
#define RAMCONTER_DEVICE 6     //内存控制器设备
#define RAM_DEVICE 7          //内存设备
#define USBHOSTCONTER_DEVICE 8 //USB主控制设备
#define INTUPTCONTER_DEVICE 9  //中断控制器设备
#define DMA_DEVICE 10          //DMA设备
#define CLOCKPOWER_DEVICE 11   //时钟电源设备
#define LCDCONTER_DEVICE 12    //LCD控制器设备
#define NANDFLASH_DEVICE 13    //Nandflash设备
#define CAMERA_DEVICE 14       //摄像头设备
#define UART_DEVICE 15         //串口设备
#define TIMER_DEVICE 16        //定时器设备
#define USB_DEVICE 17          //USB设备
#define WATCHDOG_DEVICE 18    //看门狗设备
#define IIC_DEVICE 19          //IIC设备
#define IIS_DEVICE 20          //IIS设备
#define GPIO_DEVICE 21         //通用输入输出设备
#define RTC_DEVICE 22          //实时时钟设备
#define A_DCONVER_DEVICE 23    //模转数设备, 信号转换
#define SPI_DEVICE 24          //SPI设备
#define SD_DEVICE 25           //SD卡设备
#define AUDIO_DEVICE 26        //音频设备
#define TOUCH_DEVICE 27        //触控设备
#define NETWORK_DEVICE 28      //网络设备
#define VIR_DEVICE 29          //虚拟设备
#define FILESYS_DEVICE 30      //文件系统设备
#define SYSTICK_DEVICE 31      //系统TICK设备
#define UNKNOWN_DEVICE 32      //未知设备, 也是设备

```

代码清单 9-1 中定义了 32 种设备, 但我们常见的设备却没有几个, 甚至连键盘设备都没有。不过这暂时对于我们研究学习操作系统来说是够了。另外, 记得要在“`xxxx/lmosem/include/knlincl/`”目录下建立 `krlddevice_t.h` 头文件, 并按照前面章节中的方法把这些文件组织好。

上面定义的这些类型的设备, 都是 LMOSEM 内核抽象出来的逻辑设备, 例如, `NETWORK_DEVICE` 网络设备, LMOSEM 内核不管它是有线网卡还是无线网卡, 或者是设备控制代码, 虚拟出来的虚拟网卡。

这里只要抽象出这几种类型的设备就可以了, 这是实现驱动模型的首要任务。

9.1.3 驱动程序

前面采用分权而治的方法，把操作每个设备的相关代码独立出来，这些代码称为设备控制代码，其实，操作、控制及支持每个设备的相关代码，在业界有一个更专业的名字——设备驱动程序。同时，下面的章节中，将不区分设备驱动程序和驱动程序。

有了驱动程序，给操作系统内核带了灵活性、可扩展性。可是也带来了新的问题，如下：

- ❑ 操作系统内核如何表示一个驱动程序的存在。
- ❑ 操作系统内核如何表示多个设备的存在。
- ❑ 操作系统内核如何组织多个设备和多个驱动程序。
- ❑ 操作系统内核应该让驱动程序提供一些什么支持。

以上是操作系统内核支持驱动程序开发的一些最基本的问题。如何解决呢，下面不妨设想一下。

对于前两个问题，LMOSEM 内核很好解决，我们定义描述驱动程序和设备的两个数据结构，描述驱动程序的数据结构中，保存了这个驱动程序的相关信息。而描述设备的数据结构中保存着一个设备的类型、ID 号、名字等重要的信息。LMOSEM 内核每加载一个驱动程序模块，就会自动分配一个驱动程序数据结构并且将其实例化。而 LMOSEM 内核在首次启动驱动程序时，会调用这个驱动程序的入口点函数，在这个函数中驱动程序会分配一个设备数据结构，并用相关的信息将其实例化，如填写正确的设备类型、设备 ID 号、设备名称等。LMOSEM 内核建立驱动数据结构，而驱动程序又建立设备数据结构，这一来二去就形成了一个驱动程序与 LMOSEM 内核“握手”的动作。对于第三个问题，LMOSEM 内核可以用一个称为设备表的数据结构来组织这些驱动程序数据结构和设备数据结构。

剩下最后一个问题，驱动程序究竟要为操作系统内核提供哪些最基本的功能支持。之所以写驱动程序就是为了操控相应的设备，所以这得看大多数设备能完成什么功能了。现代计算机的设备无非就是可以输入数据、处理数据、输出数据，然后完成一些特殊的功能。当然，现代计算机的设备很多，能耗是个严重的问题，所以操作系统内核应该能控制设备能耗。下面基于设备处理数据和控制设备能耗两个方面，归纳一下，看一看操作系统内核需要驱动程序提供哪些支持，如下：

- 1) 打开操作，告诉驱动程序，有一个进程在使用这个设备了。
- 2) 关闭操作，告诉驱动程序，一个进程不在使用这个设备了。
- 3) 读操作，从设备中读取相关的数据。
- 4) 写操作，写入相关的数据到设备中。

5) 控制操作，对设备进行控制，例如，声卡需要调节音量，显卡需要调整分辨率，改变设备访问权限等。

- 6) 调整数据读写位置操作，例如，硬盘需要改变读写扇区，以达到随机读写的目的。
- 7) 刷新缓存操作，有时设备驱动程序会在内存中建立数据缓存，以加快读写速度，但是有些情况下，数据必须即时写入设备。
- 8) 设置电源状态操作，有些设备可以待机，可以高性能运转，这可以通过设置电源状态来控制能耗。
- 9) 启动操作，加电启动设备。
- 10) 停止操作，停止供电，关闭设备。
- 11) 枚举操作，对于总线设备，它下面可能还有其他设备，它应该能枚举所有设备以通知操作系统内核。
- 12) 关机操作，操作系统关机时，通知设备驱动程序，计算机要关机了，该保存的数据就要保存了。

上面这 12 大操作，可以每一个操作对应一个函数，让驱动程序实现这些函数。当然有很多设备本身并不支持这么多操作，例如，时钟设备，驱动程序就可以不必实现相应的操作。

有了上面的设想和归纳，下面我们来看一看，一个标准的驱动程序的结构是什么样的，如图 9-2 所示。

图 9-2 表明了一个标准的驱动程序有三个部分。第一部分：驱动程序入口函数在驱动程序被 LMOSEM 内核加载时调用，执行驱动程序的初始化、向 LMOSEM 内核注册设备等操作，这个函数是必须要实现的。第二部分：对应上述 12 大操作的函数，当然有些操作函数，驱动程序可以选择不实现。第三部分：驱动程序的退出函数，该函数会在驱动程序退出时被调用，用于完成释放资源的操作，例如，驱动程序分配了内存空间，退出时就应该释放该内存空间，这个函数也是必须要实现的。

到这里，我们应该知道一个驱动程序是干什么的，一个驱动程序应该提供哪些功能，最后清楚了一个标准驱动程序的结构。下面就去设计实现驱动程序开发所需要的一些数据结构。

驱动程序
驱动程序入口函数
驱动程序对应内核操作设备的函数
驱动程序退出函数

图 9-2 驱动程序结构

9.2 相关数据结构

计算机不能理解驱动程序、设备、设备表、IO 包，这些东西，它只能认识或者存储数据，所以我们得把这些东西数据化，即把它们变成相应的数据结构。

对于驱动模型，数据结构有很多，所以得仔细一个一个地设计。

9.2.1 驱动

前面清楚了驱动程序和 LMOSEM 内核的“握手”过程，LMOSEM 内核一旦加载一个驱动程序，就会为此创建一个数据结构的实例变量，来表示这个驱动程序的存在。

仅仅表示一个驱动程序的存在是不够的，还要包含如驱动程序的状态、属性、名称、功能分发函数、安全访问相关的信息等。下面来设计一个数据结构用于存放这些信息，如代码清单 9-2[xxxx/lmosem/include/knlinc/krldevice_t.h] 所示。

代码清单9-2 驱动程序数据结构

```
typedef struct s_DRIVER
{
    spinlock_t drv_lock;           //保护驱动程序数据结构的自旋锁
    list_h_t drv_list;            //挂载驱动程序数据结构的链表
    uint_t drv_stuts;              //驱动程序的相关状态
    uint_t drv_flg;                //驱动程序的相关标志
    uint_t drv_id;                 //驱动程序ID
    uint_t drv_count;              //驱动程序的计数器
    sem_t drv_sem;                 //驱动程序的信号量
    void* drv_safedsc;             //驱动程序的安全体
    void* drv_attrb;               //LMOSEM内核要求的驱动程序属性体
    void* drv_privdata;            //驱动程序私有数据的指针
    drivcallfun_t drv_dipfun[IOIF_CODE_MAX]; //驱动程序功能派发函数指针数组
    list_h_t drv_alldevlist;       //挂载驱动程序所管理的所有设备的链表
    drventyexit_t drv_entry;       //驱动程序的入口函数指针
    drventyexit_t drv_exit;       //驱动程序的退出函数指针
    void* drv_userdata;            //用于将来扩展
    void* drv_extdata;             //用于将来扩展
    char_t* drv_name;              //驱动程序的名字
}driver_t;
```

代码清单 9-2 中的链表、自旋锁、信号量数据结构，在前面的章节中已经专门描述过了。driver_t 结构中有很多的指针，并且都是 void 类型的，这是为了方便以后的扩展和改变。同时，我们也考虑到了安全因素，所以有一个安全体的指针。由于一个驱动程序可能控制多个设备，所以需要有一个链表来挂载它们。drv_dipfun 是函数指针数组，这里保存着驱动程序提供的功能派发函数的指针。至于 drivcallfun_t、drventyexit_t 类型，9.2.2 节我们专门研究。

也许现在还不能完全明白驱动程序数据结构中所有域的意义，不过不用怕，等到我们写驱动程序时就会明白的。

9.2.2 派发例程类型

在驱动程序小节中，我们归纳了十几种驱动程序应该支持的操作，并且每个操作都应该对应一个函数，这里称这种函数为派发例程。

一个函数可以传递参数，但是对于这样的派发例程来说，究竟需要 LMOSEM 内核传递哪些参数呢，不同的派发例程有不同的参数，例如，读操作的派发例程，最少应该告诉它读多少数据，把数据读到什么地方等。而且还要保证派发例程的一致性，所以这是个严重的问题。因此我们把所有派发例程可能用到的参数，汇集在一起形成一个数据结构（即后面的 IO 包），然后把这个数据结构的地址传递给每个派发例程就可以了。当然，派发例程可能需要知道对应设备的信息，所以把设备数据结构的指针也传递给它。因此派发例程的类型，如代码清单 9-3[xxxx/lmosem/include/knlinc/krldevice_t.h] 所示。

代码清单9-3 派发例程类型

```
typedef drvstus_t (*drivcallfun_t)(device_t*,void*);
```

代码清单 9-3 中，可以发现派发例程函数的返回值类型为 `drvstus_t` 类型，而 `drvstus_t` 类型就是 `sint_t` 类型，只是返回一个有符号的整数，以表示派发例程函数执行是否成功。其中有一个参数的类型为 `void*` 类型，这是为了以后修改方便，因为 `void*` 类型可以存放所有其他类型的指针。

借此说一下驱动程序的入口函数和退出函数的类型，它们俩的类型是一样的，都是由 LMOSEM 内核调用的，因为它们与驱动程序有关，所以得把驱动程序数据结构的指针传递给它们，考虑到可能有附加数据，后面还跟了一个整数类型的参数和一个通用指针类型的参数，如代码清单 9-4[xxxx/lmosem/include/knlinc/krldevice_t.h] 所示。

代码清单9-4 驱动程序入口、退出函数类型

```
typedef drvstus_t (*drvtyexit_t)(struct s_DRIVER*,uint_t,void*);
```

派发例程类型非常简单，用两个指针类型作为参数，一个参数传递设备数据结构的指针，另一个参数传递参数包的指针，完美地解决了问题。所以不必过多在此浪费时间，我们继续做后面的事情。

9.2.3 设备 ID

一个计算平台内，有很多不同类型的设备，也许还有多个相同的设备，它们一起完成不同的功能。

如何索引一个具体的设备，除了设备名称外，我们还想有一个更准确的数字式的 ID 号，就像每个人都有身份证号一样。可是设备 ID 如何编码呢。前面已经说过，一个计算平台内有不同类型的设备，也可能有多个相同的设备，所以在设备类型的基础上加入设备子类型号和设备序号，因此这三个整数一起形成了一个设备 ID。为什么有设备子类型号呢，例如，硬盘和硬盘分区的关系。当然，也要为设备 ID 设计一个专用的数据结构，如代码清单 9-5[xxxx/lmosem/include/knlinc/krldevice_t.h] 所示。

代码清单9-5 设备ID数据结构

```
typedef struct s_DEVID
{
    uint_t    dev_mtype;    //设备类型号
    uint_t    dev_stype;    //设备子类型号
    uint_t    dev_nr;       //设备序号
}devid_t;
```

设备类型号、设备子类型号、设备序号都是 `uint_t` 类型的，并且都是 32 位的，这所能表示的设备数量应该足够多了。

`devid_t` 结构非常简单，它会嵌入到相应的设备数据结构中去。下面我们就去看一看怎么表示一个设备。

9.2.4 设备

设备是抽象的，而且前面 LMOSEM 内核定义的那套设备类型也只是逻辑上的，并不需要和物理设备一一对应。一个驱动程序运行之后，会向 LMOSEM 内核注册一个设备，这个注册设备的过程，就是驱动程序在一个数据结构中，填写相关的信息，然后把这个数据结构挂载到设备表数据结构之上，这样，LMOSEM 内核就能感知一个设备的存在了。

表示一个设备存在的，就是一个设备数据结构，而这个设备数据结构应该包含些什么信息呢，应该有设备名、设备 ID、设备标志和状态、设备属性等，所以设备数据结构，如代码清单 9-6[xxxx/lmosem/include/knlinc/krldevice_t.h] 所示。

代码清单9-6 设备数据结构

```
typedef struct s_DEVICE
{
    list_h_t    dev_list;           //设备链表
    list_h_t    dev_indrvlst;       //设备在驱动程序数据结构中对应的挂载链表
    list_h_t    dev_intbllst;       //设备在设备表数据结构中对应的挂载链表
    spinlock_t  dev_lock;           //设备自旋锁
    uint_t      dev_count;           //设备计数
    sem_t       dev_sem;            //设备信号量
    uint_t      dev_stus;           //设备状态
    uint_t      dev_flg;            //设备标志
    devid_t     dev_id;             //设备ID
    uint_t      dev_intlnenr;        //设备中断服务例程的个数
    list_h_t    dev_intserlst;       //设备中断服务例程的链表
    list_h_t    dev_rqlist;         //对设备的请求服务链表
    uint_t      dev_rqlnr;          //对设备的请求服务个数
    sem_t       dev_waitints;       //用于等待设备的信号量
    struct s_DRIVER* dev_drv;       //设备对应的驱动程序数据结构的指针
    void* dev_attrb;                //设备属性指针
    void* dev_privdata;             //设备私有数据指针
```



```

void* dev_userdata;           //将来扩展所用
void* dev_extdata;           //将来扩展所用
char_t* dev_name;            //设备名
}device_t;

```

device_t 结构中有很多链表，最开始三个链表是用于组织 device_t 结构的，例如，dev_indrvlst 链表是 device_t 结构在驱动程序数据结构中对应的挂载链表，因为一个驱动程序可能不止管理一个设备，所以它要用链表来挂载多个设备。而 dev_intbllst 链表是 device_t 结构在设备表数据结构中对应的挂载链表，这是为了方便查找所有的 device_t 结构。还记得前面中断管理中的 intserdsc_t 结构吗，它里面保存着设备的中断处理函数的指针，由于一个 intserdsc_t 结构总是与设备有关，所以这里的 dev_intserlst 链表就是为了挂载 intserdsc_t 结构，以便在删除设备时，同时查找并删除与这个设备相关的 intserdsc_t 结构。有很多设备请求，设备不能立即完成，它们是需要等待的，这时就把服务请求挂载在 dev_rqlist 链表中。

代码中的注释和上面的说明，应该对 device_t 数据结构很清楚了，我们后面写了驱动程序之后就会更加清楚的。

9.2.5 IO 包

LMOSEM 内核要操作相应的设备，而相应设备的操作是由设备驱动程序完成的，可是怎么让 LMOSEM 内核把相应的操作信息传递给设备驱动程序呢，LMOSEM 内核是这么做的，我们要求相应的设备完成什么操作就向相应的设备发送 IO 包。而 IO 包会包含一些重要的信息：设备对象、操作码、完成相关操作需要的参数等其他信息，其数据结构如代码清单 9-7[xxxx/lmosem/include/knlinc/krlobjnode_t.h] 所示。

代码清单9-7 IO包数据结构

```

#define OBJN_TY_DEV 1           //访问设备的类型
#define OBJN_TY_FIL 2           //访问文件的类型
#define OBJN_TY_NUL 0           //初始化类型
typedef struct s_OBJNODE
{
    spinlock_t  on_lock;         //IO包自旋锁
    list_h_t    on_list;         //IO包链表
    sem_t       on_complesem;    //IO包完成服务时用到的信号量
    uint_t      on_flg;         //IO包标志
    uint_t      on_stus;        //IO包状态
    sint_t      on_opcode;      //IO包操作码
    uint_t      on_objtype;     //IO包类型
    void*       on_objadr;      //对象指针
    uint_t      on_acsflgs;     //访问标志
    uint_t      on_acsstus;     //访问状态
    uint_t      on_currops;     //不同的操作，其意义不同
    uint_t      on_len;         //不同的操作，其意义不同
}

```

```

uint_t      on_ioctlrd;      //控制码
buf_t       on_buf;          //缓冲区地址
uint_t      on_bufcurops;    //不同的操作，其意义不同
size_t      on_bufsz;        //缓冲区大小
uint_t      on_count;        //不同的操作，其意义不同
void*       on_safedsc;      //IO包安全描述符
}objnode_t;

```

`objnode_t` 数据结构应该称为对象节点更准确，为什么这里叫 IO 包呢，因为它不只是用来操作设备的，访问文件也要用到它。`objnode_t` 结构中有些域在不同的情况下，有不同的含义，例如，对设备的读写操作只需要 `on_len`、`on_buf`、`on_bufcurops`、`on_bufsz` 这些域，而对文件的操作只需要 `on_count`、`on_buf`、`on_bufsz`、`on_currops`、`on_len` 这些域，还有一些操作不能立即返回，需要等待，这就要执行信号量（`on_complesem`）相关的操作了。还记得前面驱动程序中的分派例程数组吗，而 `on_opcode` 域最终会变成索引那个数组的下标。

同样，现在或许无法了解这个数据结构中所有成员的含义，但是到了后面的章节中，我们对设备发起相关的操作时就明白了，这里只要知道这个数据结构大致是干什么的就行了。下面我们去看一看最后一个数据结构。

9.2.6 设备表

LMOSEM 内核加载驱动程序时会创建 `driver_t` 数据结构的实例变量，而驱动程序运行后至少会创建一个 `device_t` 数据结构的实例变量。可是这些数据结构的实例变量该放在哪里，LMOSEM 内核怎么能够找到它们，这是一个问题。

`driver_t` 数据结构和 `device_t` 数据结构的实例变量，肯定是不能静态定义的，它们需要动态建立，为了解决查找它们的问题，就要定义一个全局的数据结构用于挂载它们，相当于容器。这个全局的数据结构就称为设备表，如代码清单 9-8[xxxx/lmosem/include/knlinc/krlobjnode_t.h] 所示。

代码清单9-8 设备表数据结构

```

#define DEVICE_MAX 33
typedef struct s_DEVTLST
{
    uint_t dtl_type;          //设备类型
    uint_t dtl_nr;            //设备计数
    list_h_t dtl_list;        //挂载设备device_t结构的链表
}devtlist_t;
typedef struct s_DEVTABLE
{
    list_h_t devt_list;        //设备表自身的链表
    spinlock_t devt_lock;     //设备表自旋锁
    list_h_t devt_devlist;    //全局设备链表
    list_h_t devt_drvlist;    //全局驱动程序链表，驱动程序不需要分类，一个链表就行
}

```

```
uint_t   devt_devnr;           //全局设备计数
uint_t   devt_drvnr;          //全局驱动程序计数
devtlst_t devt_devclsl[DEVICE_MAX]; //分类存放设备数据结构的devtlst_t结构数组
}devtable_t;
```

从代码清单 9-8 中的数据结构和其中的注释可以发现，它是按设备类型来组织设备数据结构的，而驱动程序的数据结构是根据设备数据结构中的信息来查找的，所以只要一个全局链表，用于挂载所有驱动程序的数据结构就可以了。

由于设备表对 LMOSEM 内核来说是全局的，所以应该把它定义成全局变量，因此和前面章节中定义全局变量一样，我们在 krlglobal.h 和 krlglobal.c 文件中加入相关的代码，代码如下所示。

```
KRL_DEFGLOB_VARIABLE(devtable_t,osdevtable);
```

对于驱动模型的这么多数据结构，当 LMOSEM 内核运行后，它们究竟是如何组织的，最好是心中有个图，现在就来画一画这幅图，如图 9-3 所示。

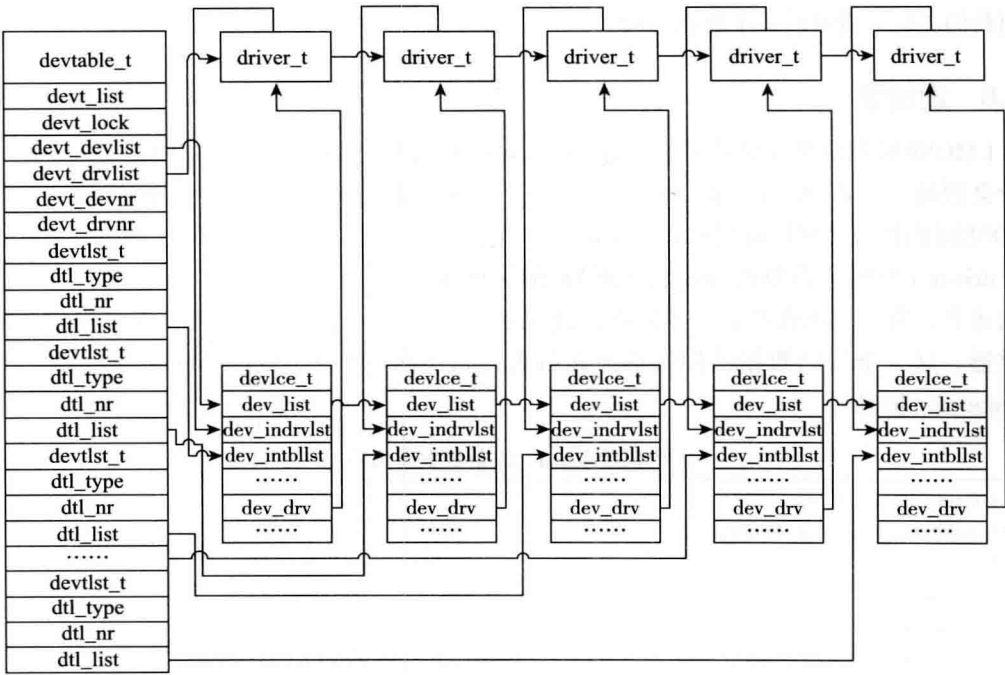


图 9-3 驱动模型相关数据结构的组织形式

好了，一个最简单的驱动模型所需要的所有数据结构就设计完成了，虽然简单，但五脏俱全，作为学习操作系统原理，这已经足够了。可是只有数据结构是不够的，还要有相应的代码，下面我们就去实现它们。

9.3 驱动模型的基础设施

从提出对设备分权而治的思想,到设计了一系列的数据结构,隐约能够感觉到,驱动模型像是操作系统内核开发者给设备驱动程序开发者制定的一套规范。既然是规范,那么除了操作设备的那一部分代码外,还有一些额外的事情要做,例如,驱动程序的加载、新建与注册设备、处理 IO 包、服务请求的同步等。这些东西本就是这套规范的重要组成部分,如果这些东西都要求驱动程序去实现,那么又何谈规范,何谈驱动模型呢。相反,正是通过这些规范来要求或协助设备厂商开发出正确稳定的驱动程序。

下面就去实现这套规范所必要的东西,这样也使驱动程序开发者只专注于设备相关代码的实现。

9.3.1 驱动程序从哪里执行

现代计算平台上,有很多的设备在使用之前,首先需要驱动程序对其进行初始化和分配相关的资源,否则设备将无法正常工作。

由于上面的问题,我们在设计驱动模型时,就要求驱动程序实现一个入口函数同时还有一个退出函数,于是驱动程序的运行过程就变成了如下过程:

1) 由操作系统内核加载驱动程序的二进制模块,这是大多数商用操作系统的通常做法,因为操作系统内核和驱动程序是用函数调用的形式“通信”的,所以操作系统内核首先要对其进行链接操作(就是链接器做的那部分工作),把操作系统内核函数的地址填写到驱动程序中调用操作系统内核函数的地方,还要按照驱动程序被加载到的内存地址,对其进行重定位。所以这使得操作系统内核必须支持动态链接功能。作为操作系统的初学者,我们并不希望一开始就搞得如此复杂。所以我们的驱动程序模块和 LMOSEM 内核在最后会链接到一起。但是下面的步骤依然会执行。

2) LMOSEM 内核动态分配一个 `driver_t` 结构的实例变量,并对其进行初级初始化。

3) LMOSEM 内核把第 2 步中的 `driver_t` 结构实例变量的指针作为参数,去调用驱动程序的入口函数。

4) 到这里驱动程序的入口函数开始运行,它首先填写正确的 `driver_t` 结构实例变量中的相关域,例如,把十几个功能派发函数指针,填入 `driver_t` 结构中的功能派发函数指针数组中。接着分配一些相关的资源,如内存缓冲区。最后初始化设备,分配一个 `device_t` 结构实例变量,在相关域中填写好正确的值,并向 LMOSEM 内核注册该设备。当驱动程序做好这一切,认为可以为 LMOSEM 内核提供服务了,就从驱动程序的入口函数中正确返回。

5) LMOSEM 内核检查驱动程序入口函数的返回值,然后检查 `driver_t` 结构实例变量中的值,最后检查系统中是否注册了新的设备。当这些检查通过之后, LMOSEM 内核就认为该驱动程序是正确的和符合驱动模型要求的。

第 1 步中,我们采用了所有驱动程序模块和 LMOSEM 内核链接在一起的方法,解决了

LMOSEM 内核加载驱动程序模块并对其进行链接的难题。可是问题又来了，虽然不用加载和链接驱动程序模块了，我们怎么找到每个驱动程序的入口函数呢。有编程经验的读者立马就会想到一个简单的方法：定义一个全局的驱动程序入口函数的指针数组，每加入一个驱动程序，就把其入口函数加入到这个数组中，LMOSEM 内核在启动时就去循环调用这个数组中的每个函数。

下面就用代码实现上面的想法，另外驱动模型也是需要初始化的，所以先写好驱动模型的初始化函数，如代码清单 9-9[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-9 初始化设备表

```
//初始化设备类型挂载表
void devtlst_t_init(devtlst_t* initp,uint_t dtype)
{
    initp->dtl_type=dtype;
    initp->dtl_nr=0;
    list_init(&initp->dtl_list);
    return;
}
//初始化设备表
void devtable_t_init(devtable_t* initp)
{
    //初始化链表
    list_init(&initp->devt_list);
    //初始化设备表自旋锁
    hal_spinlock_init(&initp->devt_lock);
    //初始化链表
    list_init(&initp->devt_devlist);
    //初始化链表
    list_init(&initp->devt_drvlist);
    initp->devt_devnr=0;
    initp->devt_drvnr=0;
    for(uint_t t=0;t<DEVICE_MAX;t++)
    {
        //依次初始化每个设备类型挂载表
        devtlst_t_init(&initp->devt_devclsl[t],t);
    }
    return;
}
void init_krldevice()
{
    //初始化设备表
    devtable_t_init(&osdevtable);
    return;
}
```

osdevtable 这个结构类型变量，定义在 krlglobal.c 文件中，所以代码清单 9-9 中的三个函数就是为了初始化设备表，init_krldevice 函数又是谁调用的呢，当然是 LMOSEM 内核功能层的主初始化函数 init_krl，如代码清单 9-10[xxxx/lmosem/kernel/krlinit.c] 所示。

代码清单9-10 LMOSEM内核功能层初始化

```
void init_krl()
{
    //初始化内存管理组件
    init_krlmm();
    .....
    //初始化设备表
    init_krldevice();
    //初始化驱动
    init_krldriver();
    .....
    hal_sysdie("LMOSEM RUN!!");
    return;
}
```

初始化了设备表，驱动程序还没有运行呢，而驱动程序的运行就是从驱动程序的入口函数开始的。可是驱动程序的入口函数在哪里调用的呢，当然就是在 init_krldriver 函数中调用的，如代码清单 9-11[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-11 执行驱动程序

```
void init_krldriver()
{
    for(uint_t ei=0;osdrvetytabl[ei]!=NULL;ei++)
    {
        if(krlrun_driverentry(osdrvetytabl[ei])==DFCERRSTUS)
        {
            hal_sysdie("init driver err");
        }
    }
    return;
}
```

代码清单 9-11 中以一个循环扫描 osdrvetytabl 数组，而所有驱动程序的入口函数就放在这个数组中。只要数组元素中的值不为 NULL 就调用 krlrun_driverentry 函数。这个函数的参数就是这个数组元素中的值，即驱动程序入口函数的地址。先来看看这个 osdrvetytabl 数组，它也定义在 krlglobal.c 文件中，如代码清单 9-12[xxxx/lmosem/kernel/krlglobal.c] 所示。

代码清单9-12 驱动程序入口函数指针数组

```
KRL_DEFGLOB_VARIABLE(drventyexit_t,osdrvetytabl)[]={
    systick_entry,uart_entry,rtc_entry,NULL};
```

osdrvetytbl 数组的最后一个元素一定要为 NULL，才能保证上面代码的循环得以结束，其他元素是笔者写好的三个驱动程序入口函数。接着重点看看 krlrun_driverentry 函数，其名字就已经告诉我们它的作用了，如代码清单 9-13[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-13 运行驱动程序入口函数

```
//所有功能操作的默认函数。返回一个错误状态就好了
drvstus_t drv_default_func(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//加载驱动程序数据结构到系统中
drvstus_t krldriver_add_system(driver_t* drv)
{
    cpufg_t cpufg;
    devtable_t* dtbp=&osdevtable;
//加锁
    hal_spinlock_saveflg_cli(&dtbp->devt_lock,&cpufg);
//把drv指向的driver_t数据结构挂载到dtbp->devt_drvlist链表中
    list_add(&drv->drv_list,&dtbp->devt_drvlist);
//增加驱动计数
    dtbp->devt_drvnr++;
//解锁
    hal_spinunlock_restflg_sti(&dtbp->devt_lock,&cpufg);
    return DFCOKSTUS;
}
//返回驱动程序ID
void krlretn_driverid(driver_t* dverp)
{
    //由于每个驱动程序数据结构的实例变量，都有唯一的内存空间，所以以它的地址作为ID具有唯一性
    dverp->drv_id=(uint_t)dverp;
    return;
}
//初始化驱动程序数据结构
void driver_t_init(driver_t* initp)
{
    hal_spinlock_init(&initp->drv_lock);
    list_init(&initp->drv_list);
    initp->drv_stuts=0;
    initp->drv_flg=0;
//获取驱动程序ID
    krlretn_driverid(initp);
    initp->drv_count=0;
    krlsem_t_init(&initp->drv_sem);
    initp->drv_safedsc=NULL;
    initp->drv_attrb=NULL;
```

```

    initp->drv_privdata=NULL;
    for(uint_t dsi=0; dsi<IOIF_CODE_MAX; dsi++)
    {
        //初始化驱动程序功能派发函数指针数组, drv_default_func是默认的派发函数
        initp->drv_dipfun[dsi]=drv_default_func;
    }
    list_init(&initp->drv_alldevlist);
    initp->drv_entry=NULL;
    initp->drv_exit=NULL;
    initp->drv_userdata=NULL;
    initp->drv_extdata=NULL;
    initp->drv_name=NULL;
    return;
}
//新建驱动程序数据结构的实例变量
driver_t* new_driver_dsc()
{
    //分配驱动程序数据结构的实例变量的空间
    driver_t* dp=(driver_t*)krlnew(sizeof(driver_t));
    if(dp==NULL)
    {
        return NULL;
    }
    //初始化驱动程序数据结构
    driver_t_init(dp);
    //返回驱动程序数据结构的实例变量的地址
    return dp;
}
drvstus_t krlrun_driverentry(drventyexit_t drventry)
{
    //新建驱动程序数据结构的实例变量
    driver_t* drvvp=new_driver_dsc();
    if(drvvp==NULL)
    {
        return DFCERRSTUS;
    }
    //把drvvp作为参数, 调用驱动程序的入口函数
    if(drventry(drvvp, 0, NULL)==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
    //到这里, 就表示驱动程序的入口函数正确返回了
    //加载驱动程序数据结构到系统中
    if(krldriver_add_system(drvvp)==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
}

```



```

    }
    return DFCOKSTUS;
}

```

我们一共实现了 6 个函数，实现了上面驱动程序运行过程的 2 ~ 5 这几个步骤。虽然代码很多，但总算很简单，看注释就可以明白。每个动态分配的数据结构的实例变量，都要进行初始后才能使用，同时看到 `driver_t_init` 函数中把 `driver_t` 结构中的功能派发函数指针数组的每个元素，都初始化为默认的 `drv_default_func` 函数的指针，这个函数只会返回一个错误状态，这是为了防止不良驱动程序不完全填写功能派发函数指针数组的所有元素，而 LMOSEM 内核又刚好调用到了那个功能派发函数，最后导致 LMOSEM 内核崩溃的情况。

到这里，我们写了这么多的代码，完成了先前的设想，应该很清楚在 LMOSEM 内核下，驱动程序是从哪里开始运行的，它又是如何开始运行的。好了，现在实现了驱动程序的首次运行的机制，还有很多机制等着我们实现呢。

9.3.2 新建与注册设备

驱动程序必须向 LMOSEM 内核注册一个设备，LMOSEM 内核才能感知一个设备的存在，才能访问设备以完成相应的操作，否则一个驱动程序除了占用内存空间外，不会有任何作用。因此驱动程序的入口函数运行后，首要任务是创建设备，其步骤如下：

第 1 步：分配一个 `device_t` 结构的实例变量，并认真填写好其中相关的域。

第 2 步：将这个 `device_t` 结构，加入到驱动程序数据结构自身管理设备的链表之中，因为一个驱动程序可能管理多个设备。

第 3 步：向 LMOSEM 内核注册这个设备，即这个 `device_t` 结构。

以上这些步骤，当然不应该由驱动程序来实现，而是由 LMOSEM 内核来实现的。

首先来看第一步，LMOSEM 内核应该提供一个函数，用于分配一个 `device_t` 结构，而填写其中相关域的工作，应该由驱动程序本身来做，当然分配一个 `device_t` 结构，还要对其进行初级的初始化工作，如代码清单 9-14[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-14 分配并初始化一个设备数据结构变量

```

//初始化设备ID
void devid_t_init(devid_t* initp,uint_t mty,uint_t sty,uint_t nr)
{
    initp->dev_mtype=mtty;
    initp->dev_stype=sty;
    initp->dev_nr=nr;
    return;
}
//初始化设备数据结构，即device_t结构
void device_t_init(device_t* initp)

```

```

{
    //初始化设备数据结构的三大链表
    list_init(&initp->dev_list);
    list_init(&initp->dev_indrvlst);
    list_init(&initp->dev_intbllst);
//初始化设备数据结构的自旋锁
    hal_spinlock_init(&initp->dev_lock);
    initp->dev_count=0;
//初始化设备数据结构的信号量
    krlsem_t_init(&initp->dev_sem);
    initp->dev_stus=0;
    initp->dev_flg=0;
//初始化设备ID
    devid_t_init(&initp->dev_id,0,0,0);
    initp->dev_intlnenr=0;
    list_init(&initp->dev_intserlst);
    list_init(&initp->dev_rqlist);
    initp->dev_rqlnr=0;
//初始化信号量
    krlsem_t_init(&initp->dev_waitints);
    initp->dev_drv=NULL;
    initp->dev_attr=NULL;
    initp->dev_privdata=NULL;
    initp->dev_userdata=NULL;
    initp->dev_extdata=NULL;
    initp->dev_name=NULL;
    return;
}
//分配设备数据结构的实例变量
device_t* new_device_dsc()
{
    //分配设备数据结构实例变量的内存空间
    device_t* dp=(device_t*)krlnew(sizeof(device_t));
    if(dp==NULL)
    {
        return NULL;
    }
//执行设备数据结构的初级初始化工作
    device_t_init(dp);
    return dp;
}

```

代码清单 9-14 中，一共实现了三个函数，但对于驱动程序开发者只需要调用 `new_device_dsc` 函数就可以了，另外两个函数都是为了初始化 `device_t` 结构的实例变量而编写的。`new_device_dsc` 函数返回创建 `device_t` 结构实例变量的指针，驱动程序开发者就可以通过这个指针填写好 `device_t` 结构中相关的域了。

驱动程序填写好 `device_t` 结构中相关的域之后，就要将这个 `device_t` 结构加入到驱动程序数据结构自身管理设备的链表之中。这个加入动作是 LMOSEM 内核的 `krldev_add_driver` 函数实现的，如代码清单 9-15[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-15 添加设备到驱动程序

```
//比较设备ID
bool_t krlcmp_devid(device_t* sdidp, device_t* cdidp)
{
//比较dev_mtype、dev_stype、dev_nr，只要其中一个不相等，就返回FALSE
    if(sdidp->dev_mtype!=cdidp->dev_mtype)
    {
        return FALSE;
    }
    if(sdidp->dev_stype!=cdidp->dev_stype)
    {
        return FALSE;
    }
    if(sdidp->dev_nr!=cdidp->dev_nr)
    {
        return FALSE;
    }
    return TRUE;
}

//把设备数据结构加入到相应的驱动程序数据结构之中
drvstatus_t krldev_add_driver(device_t* devp, driver_t* drv)
{
    list_h_t* lst;
    device_t* fdevp;
//首先检查这两个参数有没有问题
    if(devp==NULL||drv==NULL)
    {
        return DFCERRSTUS;
    }
//遍历driver_t 结构中的drv_alldevlist链表上所有的device_t结构，让其与将要加入的device_t
//结构对比，只要其设备ID相等，就返回DFCERRSTUS，表示加入操作失败
    list_for_each(lst, &drv->drv_alldevlist)
    {
        fdevp=list_entry(lst, device_t, dev_indrvlst);
        if(krlcmp_devid(&devp->dev_id, &fdevp->dev_id)==TRUE)
        {
            return DFCERRSTUS;
        }
    }
//如果所有设备ID都不相等，就把该device_t结构加入到driver_t结构中的drv_alldevlist链表上，
//同时将该device_t结构的dev_drv指向该driver_t 结构
```

```

list_add(&devp->dev_indrvlst, &drv->drv_alldevlist);
devp->dev_drv = drv;
return DFCOKSTUS;
}

```

krldev_add_driver 函数，首先会调用 krlcmp_devid 函数比较这个 driver_t 结构中所有存在的其他 device_t 结构的 ID。如果有相同的 ID 就返回错误，因为 LMOSEM 内核必须要求每个 device_t 结构的 ID 具有唯一性，就算有两个相同的物理设备，但是它们的序号应该不同。最后，调用 list_add 函数把 device_t 结构挂入 driver_t 结构的 drv_alldevlist 链表中，同时将该 device_t 结构的 dev_drv 指针指向该 driver_t 结构，这样就可以通过设备方便地找到与它对应的驱动程序了。

接下来，驱动程序就应该向 LMOSEM 内核注册设备了，下面就去实现这个向 LMOSEM 内核注册设备的函数，如代码清单 9-16 [xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-16 向LMOSEM内核注册设备

```

drvstatus_t krlnew_device(device_t* devp)
{
    device_t* findevp;
    //返回状态值的变量
    drvstatus_t rets=DFCERRSTUS;
    cpuflg_t cpufg;
    list_h_t* lstp;
    //获取LMOSEM内核设备表的地址
    devtable_t* dtbp=&osdevtable;
    //获取设备的主类型号
    uint_t devmty=devp->dev_id.dev_mtype;
    if(devp==NULL)
    {
        return DFCERRSTUS;
    }
    //检查设备是否有对应的驱动程序，没有则返回错误
    if(devp->dev_drv==NULL)
    {
        return DFCERRSTUS;
    }
    //检查设备主类型号，若大于最大设备类型号则返回错误
    if(devmty>=DEVICE_MAX)
    {
        return DFCERRSTUS;
    }
    //操作设备表要加锁
    hal_spinlock_saveflg_cli(&dtbp->devt_lock, &cpufg);
    //检查设备主类型是否和设备表元素中的类型相同，设备表的数组下标就是设备类型

```

```

    if(devmtty!=dtbp->devt_devclsl[devmtty].dtl_type)
    {
        rets=DFCERRSTUS;
        goto return_step;
    }
//遍历这个设备类型挂载链表上的所有设备，看看其中是否有相同的设备ID
    list_for_each(lstp,&dtbp->devt_devclsl[devmtty].dtl_list)
    {
        findevp=list_entry(lstp,device_t,dev_intbllst);
        if(krncmp_devid(&devp->dev_id,&findevp->dev_id)==TRUE)
        {
            rets=DFCERRSTUS;
            goto return_step;
        }
    }
//首先把设备挂入该设备类型的挂载链表中
    list_add(&devp->dev_intbllst,&dtbp->devt_devclsl[devmtty].dtl_list);
//然后把设备挂入全局链表中
    list_add(&devp->dev_list,&dtbp->devt_devlist);
//增加该设备类型挂载体中的计数
    dtbp->devt_devclsl[devmtty].dtl_nr++;
//增加全局设备计数
    dtbp->devt_devnr++;
//设置DFCOKSTUS返回值
    rets=DFCOKSTUS;
return_step:
    //解锁
    hal_spinunlock_restflg_sti(&dtbp->devt_lock,&cpufig);
    return rets;
}

```

这里关键是要理解前面那个设备表数据结构，它里面有个挂载设备的数组，那个数组的下标就是设备类型号。除了把设备加入到相应数组元素中的链接之中，同时也要把设备加入到全局设备链表中去。代码清单 9-16 中的注释已经很清楚了，不必再详述。

到这里一共写了 6 个函数，3 个函数是提供给驱动程序开发者的接口，刚好完成了驱动程序运行后必须要做的三大步骤。但驱动程序开发所需要的不仅仅是这些，还有其他的，下面就去实现它们。

9.3.3 注册回调函数

有很多设备都会产生中断，从前面中断管理的章节中，发现 LMOSEM 内核并不真正去处理一个具体设备的中断，而是提供一个可以安装具体设备中断处理函数的接口。这大大带来了灵活性，使得 LMOSEM 内核不需要了解一个具体设备中断处理的细节。

设备来了中断之后，当然是要处理的，那就得有中断处理函数，可是谁来实现这个函数呢，当然是最熟悉设备的驱动程序。可是我们希望驱动程序开发者，只专注于中断处理函数本身的实现，这个函数也可以称为中断回调函数，写好这个中断回调函数之后，只要向 LMOSEM 内核注册一下就行了。所以中断回调函数的注册函数得 LMOSEM 内核来实现。让驱动程序调用这个注册函数就行了，如代码清单 9-17[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-17 注册设备中断回调函数

```

drvstus_t krlnew_devhandle(device_t* devp,intflthandle_t handle,uint_t phyiline)
{
    intserdsc_t* sdp=krladd_irqhandle(devp,handle,phyiline);
    //如果krladd_irqhandle函数返回NULL，即表示注册失败，krladd_irqhandle函数正是中断管理组件
    //提供的注册中断回调函数的接口
    if(sdp==NULL)
    {
        return DFCERRSTUS;
    }
    cpuflg_t cpufg;
    //操作设备数据结构要加锁
    hal_spinlock_saveflg_cli(&devp->dev_lock,&cpufg);
    //把该intserdsc_t结构加入到device_t结构的dev_intserlst链表中，并增加device_t 结构的dev_
    //intlnenr计数
    list_add(&sdp->s_indevlst,&devp->dev_intserlst);
    devp->dev_intlnenr++;
    //解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufg);
    return DFCOKSTUS;
}

```

krlnew_devhandle 函数有三个参数：与此相关的设备指针、中断回调函数的指针、设备的物理中断信号线的线号（可参阅前面的中断控制器的相关信息）。这个函数首先会调用中断管理的注册中断回调函数的接口 krladd_irqhandle 函数，krladd_irqhandle 函数中会分配一个 intserdsc_t 结构的实例变量，并把中断回调函数的指针填入其中，如果一切正常，就会返回这个 intserdsc_t 结构的指针。krlnew_devhandle 函数接着就要把这个 intserdsc_t 结构加入到 device_t 结构的 dev_intserlst 链表中，并增加 device_t 结构的 dev_intlnenr 域中的值。

驱动程序所使用的注册中断回调函数的 krlnew_devhandle 函数就写好了，下面就去看一看还有什么要实现的。

9.3.4 发送 IO 包

从前面的设计中可以看出，应用进程要访问设备就会调用 LMOSEM 内核的 API 函数，LMOSEM 内核会为此创建一个 IO 包（即 objonde_t 结构），然后把这个 IO 包传递给相应的设备驱动程序。可是我们希望 LMOSEM 内核的上层代码只要创建一个 IO 包就行了，剩下的交

由驱动模型的代码去完成。

基于上面的构想，在驱动模型中实现一个以 IO 包的指针为参数的函数，LMOSEM 内核上层代码处理好 IO 包中的参数后，然后调用它就行了。下面就去实现这个函数，如代码清单 9-18[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-18 发送IO包

```

drvstus_t krldev_io(objnode_t* nodep)
{
    //提取要访问的设备数据结构的指针
    device_t* devp=(device_t*)(nodep->on_objadr);
    //首先检查objnode_t结构本身的类型是不是访问设备的类型，或者是否为NULL
    if (nodep->on_objtype!=OBJN_TY_DEV||nodep->on_objadr==NULL)
    {
        return DFCERRSTUS;
    }
    //检查IO操作码，是不是在0~ IOIF_CODE_MAX之间
    if (nodep->on_opercode<0||nodep->on_opercode>=IOIF_CODE_MAX)
    {
        return DFCERRSTUS;
    }
    //调用这个设备的驱动程序
    return krldev_call_driver(devp,nodep->on_opercode,0,0,NULL,nodep);
}

```

可以看到，krldev_io 函数只接受一个参数，就是 objnode_t 结构的指针，它其中的内容在这之前已经被 LMOSEM 内核上层代码处理好了。函数内部首先对 objnode_t 结构中相关的信息，做了合理性检查。然后把从 objnode_t 结构中获取的设备数据结构指针、IO 操作码、objnode_t 结构本身的指针，作为参数调用了 krldev_call_driver 函数，这个函数干了什么，这里先不研究。另外注意，krldev_io 函数不仅仅是 LMOSEM 内核上层代码可以调用，驱动程序内部也可以创建一个 IO 包，然后调用 krldev_io 函数，达到一个设备操作其他设备的目的。

krldev_io 函数已经写好了，下面就去看一看怎么实现调用驱动程序的函数。

9.3.5 调用驱动程序函数

krldev_io 函数中只是对 IO 包的相关信息进行了检查，取出了要访问的设备数据结构的指针和 IO 操作码。而驱动程序才是真正完成操作设备的代码，krldev_io 函数中并没有调用相应的驱动程序代码，只是调用了 krldev_call_driver 函数，事实上驱动程序中的功能派发函数就是在这个 krldev_call_driver 函数中调用的。

既然驱动程序的功能派发函数是在 krldev_call_driver 函数中调用的，所以我们得首先实现这个 krldev_call_driver 函数，如代码清单 9-19[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-19 调用驱动程序函数

```

drvstus_t krldev_call_driver(device_t* devp,uint_t iocode,uint_t vall,uint_
t val2,void* p1,void* p2)
{
    driver_t* drvp=NULL;
    //检查设备数据结构的指针和IO操作码
    if (devp==NULL||iocode>=IOIF_CODE_MAX)
    {
        return DFCERRSTUS;
    }
    //获取这个设备相关的驱动程序数据结构的指针
    drvp=devp->dev_drv;
    //检查驱动程序数据结构的指针,若为NULL就要返回一个错误状态值
    if (drv==NULL)
    {
        return DFCERRSTUS;
    }
    //调用驱动程序功能派发函数数组中的函数
    return drvp->drv_dipfun[iocode] (devp,p2);
}

```

krldev_call_driver 函数的参数一共有 6 个,目前只用到 devp、iocode、p2 这 3 个参数,其他参数是为了以后扩展所用。krldev_call_driver 函数首先检查 device_t 结构指针和 IO 操作码,然后通过该 device_t 结构的 dev_drv 域,获取该设备相应的 driver_t 结构,最后调用驱动程序中对应的功能派发函数,这个功能派发函数的指针就放在 driver_t 结构的 drv_dipfun 数组中,以 iocode 为索引获取其中一个元素,这个元素就是一个函数指针。在 C 语言中,函数指针加上括号和参数就变成了函数调用,看看前面定义的驱动程序派发函数的类型,就明白了,它刚好有两个参数:一个是设备指针,另一个是 IO 包的指针。

krldev_call_driver 函数的最后一行代码,就是驱动程序代码的执行,那是驱动程序开发者的事了。而我们驱动模型的 krldev_call_driver 函数就写好了。

9.3.6 等待服务

先来看几个情景。

情景一: 读取机械硬盘中的数据:操作系统内核上层代码,代表当前进程创建了一个 IO 包发送给机械硬盘驱动程序,机械硬盘驱动程序根据要求,给机械硬盘控制器发送命令,这时机械硬盘控制器不能立即返回数据,它要等待盘片转动,磁头寻道,这需要时间,当这些准备好了之后才能读取数据,然后以中断的形式通知 CPU。

情景二: 定时器操作,有时需要在下一个特定的时刻做特定的事,就需要操作定时器,让它在到达指定的时刻时产生中断通知 CPU。

以上两种情景都是需要等待一段时间后，才能完成任务的情况。如果操作系统内核在这段时间内就在这里“死等”，那么这个操作系统内核的多道程序的并发性能几乎为零。通常内核的做法是，当需要等待时，就把调度其他进程运行，细节在这里先不深究。

既然设备有时需要等待，那么就得起让驱动程序有机会通知操作系统内核：“我现在要等一等了，你可以做一做其他别的事情了。”通知的方法就是让驱动程序调用一个或者几个函数，可是为了让当前 IO 包不丢失，同时一个设备也可能面临多个 IO 包，因此必须要把 IO 包加入到当前设备的请求链表中。下面我们就去实现这两个函数，如代码清单 9-20 [xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-20 添加IO包到设备请求链表并执行等待操作

```

drvstus_t krldev_add_request(device_t *devp, objnode_t* request)
{
    cpufg_t cpufg;
    objnode_t* np=(objnode_t*)request;
    //加锁
    hal_spinlock_saveflg_cli(&devp->dev_lock,&cpufg);
    //把IO包加入到当前设备请求的链表中
    list_add_tail(&np->on_list,&devp->dev_rqlist);
    //增加设备请求IO包的计数
    devp->dev_rqlnr++;
    //解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufg);
    return DFCOKSTUS;
}

drvstus_t krldev_wait_request(device_t *devp, objnode_t* request)
{
    //对设备指针和IO包指针进行检查
    if(devp==NULL||request==NULL)
    {
        return DFCERRSTUS;
    }

    //操作IO包的on_complesem信号量
    krlsem_down(&request->on_complesem);
    return DFCOKSTUS;
}

```

krldev_add_request 函数非常简单，不用过多解释，驱动程序调用了 krldev_add_request 函数把 IO 包加入到设备的请求链表之后，如果接着调用 krldev_wait_request 函数，而 krldev_wait_request 函数最重要的就是调用了 krlsem_down 函数操作信号量，使当前进程可能进入等待状态。

虽然在这里写信号量的操作函数不太合适，但是我们的原则是，用到什么就写什么，所以这里要实现信号量相关的函数了。前面我们已经了解了信号量的数据结构，这里只要实现

它们的操作函数就可以了，如代码清单 9-21[xxxx/lmosem/kernel/krlsem.c] 所示。

代码清单9-21 信号量down、up操作的实现

```
void krlsem_down(sem_t* sem)
{
    cpuflg_t cpufg;
    start_step:
    // 加锁
    hal_spinlock_saveflg_cli(&sem->sem_lock, &cpufg);
    // 如果信号量值小于1，就让当前进程进入等待状态
    if(sem->sem_count < 1)
    {
        // 将当前进程挂入等待队列，后面章节中会有介绍
        krlwlst_wait(&sem->sem_waitlst);
        // 记得要解锁，否则其他代码就没办法操作信号量了
        hal_spinunlock_restflg_sti(&sem->sem_lock, &cpufg);
        // 调度并切换到新的进程运行，下次这个进程运行时将从下面的goto语句开始，后面章节中会有介绍
        krlschedul();
        // 要重新检测信号量，防止其他代码已经改变了信号量值
        goto start_step;
    }
    // 信号量值减1
    sem->sem_count--;
    // 解锁
    hal_spinunlock_restflg_sti(&sem->sem_lock, &cpufg);
    return;
}

void krlsem_up(sem_t* sem)
{
    cpuflg_t cpufg;
    // 加锁
    hal_spinlock_saveflg_cli(&sem->sem_lock, &cpufg);
    // 信号量值加1
    sem->sem_count++;
    // 如果加1了还小于1，说明信号量数据结构出了问题，所以只能死机
    if(sem->sem_count < 1)
    {
        // 解锁
        hal_spinunlock_restflg_sti(&sem->sem_lock, &cpufg);
        hal_sysdie("sem up err");
    }
    // 唤醒等待队列上所有的进程，后面章节中会有介绍
    krlwlst_allup(&sem->sem_waitlst);
    // 解锁
    hal_spinunlock_restflg_sti(&sem->sem_lock, &cpufg);
    // 设置需要调度的标志，后面章节中会有介绍
```

```

    krlsched_set_schedflgs();
    return;
}

```

如果拿开进程等待、唤醒、调度的函数，`krlsem_down` 函数、`krlsem_up` 函数就是对信号量值进行原子的减一、加一操作。然后通过信号量值判断一个进程是否需要进入等待状态。小于 1 则进入等待状态。其中有些函数是进程相关的章节该介绍的，这里先不介绍它们了。

到这里，我们看到了一个进程因为设备不能立即完成操作，而进入等待状态的过程，以及实现这一过程的代码。下面我们就去看看，设备过了一段时间后完成相应的操作了，如何让等待的进程活过来。

9.3.7 完成服务

紧接着 9.3.6 节，如果当设备完成了相应的服务请求，如何让请求这个服务的进程活过来。下面就去解决这个问题。

设备完成了相应的操作之后，大部分情况下都会产生一个中断，而驱动程序会编写这个设备的中断处理函数。而在该设备中断处理函数中就应该看一看，有没有未完成的服务请求。可是我们需要尽量减少驱动开发者的难度，所以把这些问题交由驱动模型的代码去处理，何况这里还有唤醒相应进程的任务呢。

事实上，下面我们要实现的函数，就是 9.3.6 节中两个函数的逆过程，无非就是把 IO 包从设备的请求链表上删除，然后把相应的进程唤醒。好了，现在就来实现这个函数，如代码清单 9-22[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-22 完成IO服务

```

drvstus_t krldev_retn_request(device_t *devp, uint_t iocode, objnode_t** retreq)
{
    //检查参数的合理性
    if(retreq==NULL||iocode>=IOIF_CODE_MAX)
    {
        return DFCERRSTUS;
    }
    cpuflg_t cpufg;
    objnode_t* np;
    list_h_t* list;
    drvstus_t rets=DFCERRSTUS;
    //加锁
    hal_spinlock_saveflg_cli(&devp->dev_lock, &cpufg);
    //遍历device_t 结构的dev_rqlist链表上所有的IO包 (objnode_t结构)
    list_for_each(list, &devp->dev_rqlist)
    {
        np=list_entry(list, objnode_t, on_list);
        //比较IO操作码是否相同，相同则通过retreq参数返回这个IO包 (objnode_t结构) 的指针
    }
}

```

```

        if(np->on_opcode==(sint_t)iocode)
        {
            *retreq=np;
            rets=DFCOKSTUS;
            goto return_step;
        }
    }
//到这里说明没有找到相应的IO包 (objnode_t结构) 所以要设置错误的状态值以便返回
    rets=DFCERRSTUS;
    *retreq=NULL;
    return_step:
    //解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufig);
    return rets;
}

drvstus_t krldev_complete_request(device_t *devp, objnode_t* request)
{
    //对参数进行检查
    if (devp==NULL||request==NULL)
    {
        return DFCERRSTUS;
    }
    //如果device_t 结构中的dev_rqlnr小于1说明没有相应的IO包, 那肯定出了问题, 所以死机
    if (devp->dev_rqlnr<1)
    {
        hal_sysdie("krldev_complete_request err devp->dev_rqlnr<1");
    }
    cpufig_t cpufig;
    //加锁
    hal_spinlock_saveflg_cli(&devp->dev_lock,&cpufig);
    //删除IO包
    list_del(&request->on_list);
    //减少其计数
    devp->dev_rqlnr--;
    //解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufig);
    //操作信号量, 同时唤醒相关的进程
    krlsem_up(&request->on_complesem);
    return DFCOKSTUS;
}

```

`krldev_retn_request` 函数主要是遍历 `device_t` 结构中的 `dev_rqlist` 链表上所有的 IO 包 (`objnode_t` 结构), 找到相应的 IO 包并且返回其指针, 事实上目前我们的驱动模型只支持等待一个 IO 包, 不支持多个 IO 包, 这样做是为了以后扩展。一般驱动程序调用了 `krldev_retn_request` 函数, 紧接着会调用 `krldev_complete_request` 函数, 在这个函数中才把刚刚找到

的 IO 包 (objnode_t 结构) 从 device_t 结构中的 dev_rqlist 链表上删除, 并减少其计数, 最后调用 krlsem_up 函数操作 IO 包的 on_complesem 信号量, 同时唤醒相关的进程。9.3.7 节中已经介绍过 krlsem_up 函数了, 关于唤醒进程的操作, 那是进程相关的章节要讨论的, 这里不必深究。

驱动模型提供给驱动程序开发者, 用于等待和完成服务的机制就实现了, 下面去实现驱动模型的一些辅助函数。

9.3.8 驱动模型辅助函数

前面写好了一个简单的驱动模型所必须的代码, 不过我们想让驱动开发者和 LMOSEM 内核上层代码开发者更加方便地操控设备, 所以还得去实现一些辅助性的函数。不过一切从简单出发, 作为一个操作系统初学者, 我们并不想让问题复杂化。

从下面三个简单的方面开始:

- 1) 增加或者减少设备相关的计数变量。
- 2) 获取 IO 包中的参数。
- 3) 查找一个设备。

设备可能同时被多个进程使用, 当一个进程打开一个设备时, 驱动程序可能要分配一些资源。当进程关闭设备时, 驱动程序应该释放那些已经分配的资源, 可是如果多个进程同时在使用一个设备, 这种做法就会带来问题。因为释放相关资源时, 可能设备还在使用它们为其他进程服务, 怎么处理这种情况呢, 最简单的方法是在设备数据结构中增加一个计数变量, 从前面相关章节中看到, 我们的 device_t 结构中确实有个计数变量。每打开一次设备就增加一次计数变量的值, 每关闭一次设备时就对其减少一次计数变量的值, 减至计数变量的值为 0 时就释放资源。下面就来实现这两个函数, 让驱动程序开发者只需要调用即可, 如代码清单 9-23[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-23 增加、减少设备相关的计数变量值

```

drvstus_t krldev_inc_devcount(device_t* devp)
{
    cpufg_t cpufg;
    //加锁
    hal_spinlock_saveflg_cli(&devp->dev_lock, &cpufg);
    //检查计数变量是否回绕
    if(devp->dev_count>=(~0UL))
    {
        //解锁
        hal_spinunlock_restflg_sti(&devp->dev_lock, &cpufg);
        return DFCERRSTUS;
    }
    //计数变量dev_count加1
    devp->dev_count++;

```

```

//解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufg);
    return DFCOKSTUS;
}
drvstus_t krldev_dec_devcount(device_t* devp)
{
    cpufg_t cpufg;
    hal_spinlock_saveflg_cli(&devp->dev_lock,&cpufg);
    //检查计数变量是否小于1,如果是,那肯定出错了,因为小于1就不能再减1了
    if(devp->dev_count<(1))
    {
        //解锁
        hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufg);
        return DFCERRSTUS;
    }
    //计数变量dev_count减1
    devp->dev_count--;
    //解锁
    hal_spinunlock_restflg_sti(&devp->dev_lock,&cpufg);
    return DFCOKSTUS;
}

```

代码清单 9-23 中的这两个函数非常简单,代码中的注释已经很明确了。不必详述。

接着来看看获取 IO 包中的参数。按照常理,驱动程序开发者有理由完全掌握 `objnode_t` 数据结构,所以它可以轻松获取其中任何一个参数。可是 LMOSEM 内核为什么还要写这样一个帮助驱动程序获取 IO 包中的参数的函数呢。因为随着以后 LMOSEM 内核的开发可能对 `objnode_t` 数据结构有所修改,为了最大可能地保证现有驱动程序的兼容性,所以我们让 LMOSEM 内核代码去做这个事情。那么就需要实现一个这样的函数,如代码清单 9-24[xxxx/lmosem/kernel/krldevice.c] 所示。

代码清单9-24 获取IO包中的参数

```

drvstus_t krldev_retn_rqueparm(void* request,buf_t* retbuf,uint_t* retcops,uint_
t* retlen,uint_t* retioclde,uint_t* retbufcops,size_t* retbufsz)
{
    objnode_t* ondep=(objnode_t*)request;
    if(ondep==NULL)
    {
        return DFCERRSTUS;
    }
    if(retbuf!=NULL)
    {
        *retbuf=ondep->on_buf;
    }
    if(retcops!=NULL)
    {

```

```

        *retcops=ondep->on_currops;
    }
    if(retlen!=NULL)
    {
        *retlen=ondep->on_len;
    }
    if(retiocldc!=NULL)
    {
        *retiocldc=ondep->on_ioctrd;
    }
    if(retbufcops!=NULL)
    {
        *retbufcops=ondep->on_bufcurops;
    }
    if(retbufsz!=NULL)
    {
        *retbufsz=ondep->on_bufsz;
    }
    return DFCOKSTATUS;
}

```

首先 `krldev_retn_rqueparm` 函数有 7 个参数，都是指针类型的，除了第一个参数外都是为了携带返回值的。只要携带相应返回值的参数不为 `NULL`，就要把相应 `objnode_t` 结构中的域赋给它。同时也告诉驱动程序开发者，如果不需要返回 `objnode_t` 结构中相关的信息，只要把相应的函数参数设置为 `NULL` 就行了。代码虽然很长，但实在太简单，不必详述。

最后我们来看看查找一个设备，`LMOSEM` 内核上层代码要访问一个设备，首先得找到这个设备。所以得实现一个查找设备的函数。经常看到 `Linux` 系统下以访问文件的方式访问设备。这是因为 `Linux` 系统把驱动模型纳入了文件系统，使得访问设备就像访问文件一样。虽然这样做很完美，但是为了简单，我们并不这样做，也许以后我们也会这样做，但是现在我们只想通过设备 ID 找到设备，所以这个设备查找函数，如代码清单 9-25[xxxx/lmosem/kernel/krlddevice.c] 所示。

代码清单9-25 查找相应的设备

```

device_t* krlonidfl_retn_device(void* dfname,uint_t flgs)
{
    device_t* findevp;
    cpuflg_t cpufg;
    list_h_t* lstp;
    //获取设备表指针
    devtable_t* dtbp=&osdevtable;
    //检查参数，如果flgs=DIDFIL_IDN就表示dfname指向的是devid_t数据结构，否则应该是Linux系统那种
    //方式，但是目前不支持，所以如果不符合要求就返回NULL

```

```

    if (dfname==NULL || flgs!=DIDFIL_IDN)
    {
        return NULL;
    }
    //获取devid_t数据结构的指针
    devid_t* didp=(devid_t*)dfname;
    //获取设备的主类型号
    uint_t devmty=didp->dev_mtype;
    //检查设备的主类型号是否大于等于DEVICE_MAX, 如果是就返回NULL
    if (devmty>=DEVICE_MAX)
    {
        return NULL;
    }
    //加锁
    hal_spinlock_saveflg_cli(&dtbp->devt_lock,&cpufg);
    //如果设备的主类型号不等于设备挂载数组中的设备主类型号也返回NULL
    if (devmty!=dtbp->devt_devclsl[devmty].dtl_type)
    {
        findevp=NULL;
        goto return_step;
    }
    //遍历当前设备类型的设备挂载链表上的所有设备, 每找到一个设备就比较其设备ID
    list_for_each(lstp,&dtbp->devt_devclsl[devmty].dtl_list)
    {
        findevp=list_entry(lstp,device_t,dev_intbllst);
        //调用krlcmp_devid函数比较设备ID, 返回TRUE就表明找到了, 然后返回该设备的指针
        if (krlcmp_devid(didp,&findevp->dev_id)==TRUE)
        {
            findevp=findevp;
            goto return_step;
        }
    }
    //如果到这里, 说明没有找到相应的设备
    findevp=NULL;
    return_step:
    //解锁
    hal_spinunlock_restflg_sti(&dtbp->devt_lock,&cpufg);
    //返回设备的指针
    return findevp;
}

```

`krlonidfl_retn_device` 函数有两个参数 `flgs` 参数完全是为了以后扩展所用, 也许有一天我们让 `flgs` 改变一个值, 同时让 `dfname` 指向一个字符串 (相当于 Linux 系统下的设备文件名), 就可以变成和 Linux 系统一样了。接着 `krlonidfl_retn_device` 函数获取设备表的指针, 并对参数和其他一些数值进行了检查。然后就遍历当前设备类型的设备挂载链表上的所有设备, 每找到一个设备就调用 `krlcmp_devid` 函数比较其设备 ID, 返回 `TRUE` 表明找到了设备。最后

返回找到的设备指针。

好了，一些辅助函数也写好了，我们这个简单的驱动模型的基础设施也差不多了。下面可以休息一下了，顺便回过头去，想一想我们实现了这么多机制都是为了什么。清楚了这个问题，驱动模型是什么也就清楚了。

9.4 systick 驱动程序实例

前面作为操作系统内核开发者，为设备驱动程序开发者制定了一套规范，并为实现这套规范而设计了很多数据结构和编写了大量代码。现在切换一下身份，去感受一下驱动程序开发者的痛苦与欢乐，去实现一个 systick 设备的驱动程序，亲自验证一下我们制定的那一套规范是否合理。

systick 设备是系统中一个特殊的设备，因为后面的进程调度会依赖于它。它实际上就是一个不间断地定时产生中断的设备。例如，每秒钟可以产生一千次中断。下面先从 systick 硬件开始，接着编写 systick 驱动程序的入口函数，然后按照驱动模型的要求编写 systick 驱动程序的功能派发函数，最后测试一下这个设备是否正常运转了。

9.4.1 systick 硬件

写 systick 设备驱动程序之前，首先要对 systick 硬件设备有所了解，systick 设备是一个不间断的定时产生中断的设备，所以得找找看计算平台上有没有这样的设备，答案是肯定的，S3C2440A 处理器内部的定时器就是这种设备。

关于定时器的简介请参阅前面的相关章节。S3C2440A 处理器内部一共有 5 个定时器，我们的 systick 设备只需要用到其中一个定时器，我们选择其中功能最简单的定时器 4，因为它没有其他额外功能，就只能不间断地定时产生中断，并且没有外部输出引脚。

S3C2440A 处理器内部的定时器系统很复杂，这里只是简单地画出定时器 4 的结构简化图，如图 9-4 所示。

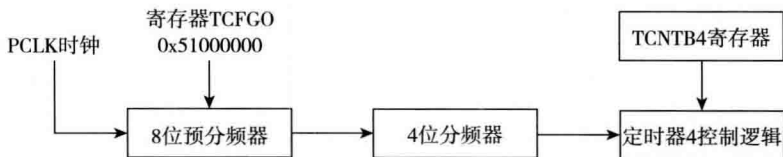


图 9-4 定时器 4 的结构

图 9-4 中，定时器的控制逻辑在每来一个时钟信号时，就对 TCNTB4 寄存器中的数值执行减 1 操作，当减到 0 时就发出中断信号。而定时器的时钟源来源于 PCLK 时钟源，经过一个 8 位的预分频器，再经过一个 4 位的分频器才到达定时器，而 PCLK 时钟源被开发板的引导程序设置为 50MHz，4 位的分频器被设置为 4。所以我们只要控制 8 位的预分频器和

TCNTB4 寄存器中的数值, 就能控制定时器每秒钟产生中断的次数。可是定时器的最终时钟源, 怎么计算呢。有一个相关的公式, 如下:

定时器的输入时钟源频率 = $PCLK / \{ \text{预分频值} + 1 \} / \{ \text{分频值} \}$

{ 预分频值 } = 0 ~ 255

{ 分频值 } = 2, 4, 8, 16

PCLK 是 S3C2440A 处理器内部的 APB 总线的时钟源。

上面的公式可能很抽象, 下面我们来看一个表就明白了, 如表 9-1 所示。

表9-1 定时器时钟源

4 位分频器设置	最小分辨率 (预分频器 = 0)	最大分辨率 (预分频器 = 255)	最大间隔时间 (TCNTB4=65535)
2 (PCLK = 50MHz)	0.0400 μ s (25.0000MHz)	10.2400 μ s (97.6562kHz)	0.6710s
4 (PCLK = 50MHz)	0.0800 μ s (25.0000MHz)	20.4800 μ s (97.6562kHz)	1.3421s
8 (PCLK = 50MHz)	0.1600 μ s (25.0000MHz)	40.9601 μ s (97.6562kHz)	2.6843s
16 (PCLK = 50MHz)	0.3200 μ s (25.0000MHz)	81.9188 μ s (97.6562kHz)	5.3686s

看了表 9-1 和上面的公式, 我们应该知道分频器和 TCNTB4 寄存器的值需要怎么设置了吧。

从前面操作硬件设备的经验, 发现操作硬件设备就是操作一些特定的寄存器。下面来看一看操作定时器 4 需要关注哪些寄存器。其列在下面:

- 1) 定时器配置寄存器 0 (TCFG0)。
- 2) 定时器配置寄存器 1 (TCFG1)。
- 3) 定时器控制寄存器 (TCON)。
- 4) 定时器 4 计数缓冲寄存器 (TCNTB4)。

下面分别介绍这些寄存器, 看一看它们有什么作用。

先来看一看定时器配置寄存器 0 (TCFG0), 预分频器的值, 就是在这个寄存器中设置的, 它的格式如表 9-2 所示。

表9-2 定时器配置寄存器0

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
TCFG0 (0x51000000)	32	读 / 写	配置两个 8 位预分频器	0
保留	[31:24]	读 / 写		0
死区长度	[23:16]	读 / 写	该 8 位决定了死区段。死区段持续为 1 的时间等于定时器 0 持续为 1 的时间	0
预分频器 1	[15:8]	读 / 写	该 8 位决定了定时器 2、3、4 的预分频值	0
预分频器 0	[7:0]	读 / 写	该 8 位决定了定时器 0、1 的预分频值	0

在时钟源进入定时器控制逻辑之前，还有一个类似于选通门的器件，定时器配置寄存器 1 (TCFG1) 就是控制它的，它的格式如表 9-3 所示。

表9-3 定时器配置寄存器1

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
TCFG1 (0x51000004)	32	读 / 写	多路选择器和 DMA 模式选择寄存器	0
保留	[31:24]	读 / 写		0
DMA 模式	[23:20]	读 / 写	选择 DMA 请求通道: 0000 = 未选择 (所有中断); 0001 = 定时器 0; 0010 = 定时器 1; 0011 = 定时器 2; 0100 = 定时器 3; 0101 = 定时器 4; 0110 保留	0
MUX4	[19:16]	读 / 写	选择定时器 4 的选通输入 0000 = 1/2; 0001 = 1/4; 0010 = 1/8; 0011 = 1/16; 01xx = 外部 TCLK1	0
MUX3	[15:12]	读 / 写	选择定时器 3 的选通输入 0000 = 1/2; 0001 = 1/4; 0010 = 1/8; 0011 = 1/16; 01xx = 外部 TCLK1	0
MUX2	[11:8]	读 / 写	选择定时器 2 的选通输入 0000 = 1/2; 0001 = 1/4; 0010 = 1/8; 0011 = 1/16; 01xx = 外部 TCLK1	0
MUX1	[7:4]	读 / 写	选择定时器 1 的选通输入 0000 = 1/2; 0001 = 1/4; 0010 = 1/8; 0011 = 1/16; 01xx = 外部 TCLK1	0
MUX0	[3:0]	读 / 写	选择定时器 0 的选通输入 0000 = 1/2; 0001 = 1/4; 0010 = 1/8; 0011 = 1/16; 01xx = 外部 TCLK1	0

定时器要启动或者停止，这就得依靠定时器的控制寄存器 (TCON)，它的格式如表 9-4 所示。

表9-4 定时器控制寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
TCON (0x51000008)	23	读 / 写	定时器控制寄存器	0
定时器 4 自动重载开 / 关	[22]	读 / 写	决定定时器 4 的自动重载开启或关闭: 0 = 单稳态; 1 = 间隙模式 (自动重载)	0
定时器 4 手动更新	[21]	读 / 写	决定定时器 4 的手动更新: 0 = 无操作; 1 = 更新 TCNTB4。注意此位必须在下次写操作时清零	0
定时器 4 启动 / 停止	[20]	读 / 写	决定定时器 4 的启动或停止: 0 = 停止定时器 4; 1 = 启动定时器 4	0
定时器 3 自动重载开 / 关	[19]	读 / 写	决定定时器 3 的自动重载开启或关闭: 0 = 单稳态; 1 = 间隙模式 (自动重载)	0

(续)

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
定时器 3 输出变向开 / 关	[18]	读 / 写	决定定时器 3 的输出变相开启或关闭: 0 = 关闭变相; 1 = TOUT3 (定时器 3 的输出引脚) 变换极性	0
定时器 3 手动更新	[17]	读 / 写	决定定时器 3 的手动更新: 0 = 无操作; 1 = 更新 TCNTB3 和 TCMPB3。注意此位必须在下次写操作时清零	0
定时器 3 启动 / 停止	[16]	读 / 写	决定定时器 3 的启动或停止: 0 = 停止定时器 3; 1 = 启动定时器 3	0
定时器 2 自动重载开 / 关	[15]	读 / 写	决定定时器 2 的自动重载开启或关闭: 0 = 单稳态; 1 = 间隙模式 (自动重载)	0
定时器 2 输出变向开 / 关	[14]	读 / 写	决定定时器 2 的输出变相开启或关闭: 0 = 关闭变相; 1 = TOUT2 (定时器 2 的输出引脚) 变换极性	0
定时器 2 手动更新	[13]	读 / 写	决定定时器 2 的手动更新: 0 = 无操作; 1 = 更新 TCNTB2 和 TCMPB2。注意此位必须在下次写操作时清零	0
定时器 2 启动 / 停止	[12]	读 / 写	决定定时器 2 的启动或停止: 0 = 停止定时器 2; 1 = 启动定时器 2	0
定时器 1 自动重载开 / 关	[11]	读 / 写	决定定时器 1 的自动重载开启或关闭: 0 = 单稳态; 1 = 间隙模式 (自动重载)	0
定时器 1 输出变向开 / 关	[10]	读 / 写	决定定时器 1 的输出变相开启或关闭: 0 = 关闭变相; 1 = TOUT1 (定时器 1 的输出引脚) 变换极性	0
定时器 1 手动更新	[9]	读 / 写	决定定时器 1 的手动更新: 0 = 无操作; 1 = 更新 TCNTB1 和 TCMPB1。注意此位必须在下次写操作时清零	0
定时器 1 启动 / 停止	[8]	读 / 写	决定定时器 1 的启动或停止: 0 = 停止定时器 1; 1 = 启动定时器 1	0
保留	[7:5]	读 / 写	保留	0
死区使能	[4]	读 / 写	决定死区操作: 0 = 禁止; 1 = 使能	0
定时器 0 自动重载开 / 关	[3]	读 / 写	决定定时器 0 的自动重载开启或关闭: 0 = 单稳态; 1 = 间隙模式 (自动重载)	0
定时器 0 输出变向开 / 关	[2]	读 / 写	决定定时器 0 的输出变相开启或关闭: 0 = 关闭变相; 1 = TOUT0 (定时器 0 的输出引脚) 变换极性	0
定时器 0 手动更新	[1]	读 / 写	决定定时器 0 的手动更新: 0 = 无操作; 1 = 更新 TCNTB0 和 TCMPB0。注意此位必须在下次写操作时清零	0
定时器 0 启动 / 停止	[0]	读 / 写	决定定时器 0 的启动或停止: 0 = 停止定时器 0; 1 = 启动定时器 0	0

最后来看一看定时器 4 的计数缓冲寄存器（TCNTB4），每当这个寄存器里的数值减到 0 时，定时器 4 就会发出中断信号。它的格式如表 9-5 所示。

表9-5 定时器4的计数缓冲寄存器

寄存器（地址）	位	是否可读 / 写	描 述	初始值
TCNTB4（0x5100003C）	16	读 / 写	定时器 4 的计数缓冲寄存器	0
定时器 4 计数缓冲寄存器	[15:0]	读 / 写	设置定时器 4 的计数缓冲寄存器的数值	0

因为我们只需要操作定时器 4，所以了解上述 4 个寄存器就足够了。

好了，我们了解了定时器 4 的结构图，接着又了解了与它相关的 4 个寄存器，下面就去设计驱动程序，编写相关代码操作这个定时器 4。

9.4.2 systick 驱动程序框架

前面了解了 systick 硬件设备，现在就要编写 systick 驱动程序了，必须按照前面 LMOSEM 内核的驱动模型规定的驱动程序结构来编写。

LMOSEM 内核驱动模型规定的驱动程序结构如下：

- 1）驱动程序必须要有入口函数和退出函数。
- 2）驱动程序功能派发函数。
- 3）如果硬件设备有中断，应该有中断回调函数。

下面，首先按照上面的规定写好这些函数，形成 systick 驱动程序框架，再根据需要具体化其中一些函数。注意这些函数的参数个数、参数类型、返回值类型都有相关规定，一定要按照那些规定来编写。不过在此之前，先在“xxxx/lmosem/”和“xxxx/lmosem/include/”目录下分别建立一个 drivers 目录和一个 drvinc 目录，在“xxxx/lmosem/drivers”目录下建立一个 C 语言源代码文件 drvtick.c，并在“xxxx/lmosem/include/drvinc”目录下建立 drvtick_t.h、drvtick.h 两个头文件，同时按照前面的方法组织好它们（注意，xxxx 即表示 LMOSEM 内核的工程目录）。还要改写 buildfile.h 文件中相关的宏，即 #define BUILD_DRIV_OBJS drvtick.o。最后在 drvtick.c 文件中写下如代码清单 9-26[xxxx/lmosem/drivers/drvtick.c] 所示的函数。

代码清单9-26 systick驱动程序框架

```
//systick设备驱动程序的入口函数
drvstus_t systick_entry(driver_t* drvvp,uint_t val,void* p)
{
    return DFCOKSTUS;
}
//systick设备驱动程序的退出函数
drvstus_t systick_exit(driver_t* drvvp,uint_t val,void* p)
{
    return DFCERRSTUS;
}
```

```

//systick设备驱动程序的中断回调函数
drvstus_t systick_handle(uint_t ift_nr,void* devp,void* sframe)
{
    return DFCOKSTUS;
}
//systick设备驱动程序的打开函数
drvstus_t systick_open(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的关闭函数
drvstus_t systick_close(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的读函数
drvstus_t systick_read(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的写函数
drvstus_t systick_write(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的调整读写位置函数
drvstus_t systick_lseek(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的控制函数
drvstus_t systick_ioctl(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的开始函数
drvstus_t systick_dev_start(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的停止函数
drvstus_t systick_dev_stop(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的设置电源函数

```

```

drvstus_t systick_set_powerstus(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的枚举函数
drvstus_t systick_enum_dev(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的刷新缓存函数
drvstus_t systick_flush(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}
//systick设备驱动程序的关机函数
drvstus_t systick_shutdown(device_t* devp,void* iopack)
{
    return DFCERRSTUS;
}

```

代码清单 9-26 告诉我们，目前一个符合 LMOSEM 内核驱动模型标准的驱动程序，至少需要 15 个函数。当然后面会看到，我们并不需要实现每个功能派发函数，而是按需要实现的。不需要实现的，让其返回一个错误状态值就行了，但是相应的功能派发函数还是要写上去的。

关于 systick 设备驱动程序框架就实现了。有了驱动程序框架，下面就要去具体实现驱动程序框架里的一些函数了。

9.4.3 systick 驱动程序实现

实现了驱动程序框架，接着就是按需要实现一些具体的函数了。按照驱动模型的要求，驱动程序首先执行的就是驱动程序的入口函数，那么驱动程序的入口函数到底要做些什么呢。

驱动程序的入口函数要做的事情，可分为如下两个方面。

(1) LMOSEM 内核驱动模型所需要的

- 1) 创建 device_t 结构的实例变量。
- 2) 填写 driver_t 结构实例变量与 device_t 结构实例变量中的相关域
- 3) 将 device_t 结构注册到驱动程序和 LMOSEM 内核中。

4) 注册设备中断回调函数

(2) 硬件设备所需要的

- 1) 初始化硬件设备。
- 2) 开启硬件设备中断响应。

看上去要做的事情确实不少，不过，实际上 LMOSEM 内核驱动模型提供了大量的接口函数，帮助我们完成了大部分工作，只要调用几个函数就解决了，唯一要仔细做的事情就是初始化硬件设备。

首先是 LMOSEM 内核驱动模型所需要的部分。下面将以代码片段的形式，一步步实现上面的要求。

第一，创建 `device_t` 结构的实例变量，这很简单，LMOSEM 内核驱动模型提供了相应的函数，不过要判断一下返回值，看一看是否创建成功了。所以写下如代码清单 9-27[xxxx/lmosem/drivers/drvtick.c] 所示的代码片段。

代码清单9-27 创建device_t结构的实例变量

```
.....
//创建device_t结构的实例变量
device_t* devp=new_device_dsc();
//判断是否创建成功，不成功则返回错误的状态值
if(devp==NULL)
{
    return DFCERRSTUS;
}
.....
```

第二，填写 `driver_t` 结构实例变量与 `device_t` 结构实例变量中的相关域，由于 `driver_t` 结构实例变量的指针是 LMOSEM 内核作为驱动程序入口函数的参数传递进来的，而 `device_t` 结构实例变量是由驱动程序创建的，但是它们都只进行了初级的初始化，其中具体的数据还是要由驱动程序来填写的。由于要填写的数据比较多，因此在驱动程序中写两个小函数来处理，如代码清单 9-28[xxxx/lmosem/drivers/drvtick.c] 所示。

代码清单9-28 填写driver_t、device_t实例变量中的相关域

```
//填写driver_t结构实例变量的相关域
void systick_set_driver(driver_t* drv)
{
    //把驱动程序的功能派发函数指针，填写到driver_t结构实例变量的drv_dipfun数组中
    drv->drv_dipfun[IOIF_CODE_OPEN]=systick_open;
    drv->drv_dipfun[IOIF_CODE_CLOSE]=systick_close;
    drv->drv_dipfun[IOIF_CODE_READ]=systick_read;
    drv->drv_dipfun[IOIF_CODE_WRITE]=systick_write;
    drv->drv_dipfun[IOIF_CODE_LSEEK]=systick_lseek;
    drv->drv_dipfun[IOIF_CODE_IOCTL]=systick_ioctl;
    drv->drv_dipfun[IOIF_CODE_DEV_START]=systick_dev_start;
    drv->drv_dipfun[IOIF_CODE_DEV_STOP]=systick_dev_stop;
    drv->drv_dipfun[IOIF_CODE_SET_POWERSTUS]=systick_set_powerstus;
    drv->drv_dipfun[IOIF_CODE_ENUM_DEV]=systick_enum_dev;
    drv->drv_dipfun[IOIF_CODE_FLUSH]=systick_flush;
    drv->drv_dipfun[IOIF_CODE_SHUTDOWN]=systick_shutdown;
```

```

//设置驱动程序名
drv->drv_name="systick0drv";
return;
}
//填写device_t结构实例变量的相关域
void systick_set_device(device_t* devp,driver_t* drv)
{
    //填写设备标志,为独占使用设备
    devp->dev_flg=DEVFLG_SHARE;
    //填写设备状态,为普通默认状态,并且设备是可用的
    devp->dev_stus=DEVSTS_NORML;
    //填写设备类型,为SYSTICK_DEVICE类型
    devp->dev_id.dev_mtype=SYSTICK_DEVICE;
    //填写设备子类型,为0
    devp->dev_id.dev_stype=0;
    //填写设备序号,为0
    devp->dev_id.dev_nr=0;
    //填写设备名,为systick0
    devp->dev_name="systick0";
    return;
}

```

第三,设备要想和LMOSEM内核、驱动程序联系起来,就要将device_t结构注册到驱动程序和LMOSEM内核中。将device_t结构注册到驱动程序和LMOSEM内核中。LMOSEM内核的驱动模型提供了两个函数,只要正确调用即可,如代码清单9-29[xxxx/lmosem/drivers/drvtick.c]所示。

代码清单9-29 向驱动和LMOSEM内核注册设备

```

.....
//检查设备注册到驱动程序是否成功,不成功就需要释放device_t结构实例变量
if(krldrv_add_driver(devp,drv)==DFCERRSTUS)
{
    //注意释放资源
    if(del_device_dsc(devp)==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
    //返回错误状态
    return DFCERRSTUS;
}
//检查设备注册到LMOSEM内核是否成功,不成功就需要释放device_t结构实例变量
if(krlnw_device(devp)==DFCERRSTUS)
{
    //注意释放资源
    if(del_device_dsc(devp)==DFCERRSTUS)

```

```

    {
        return DFCERRSTUS;
    }
    //返回错误状态
    return DFCERRSTUS;
}
.....

```

第四，如果硬件设备会产生中断，就必须要向 LMOSEM 内核注册一个该硬件设备对应的中断回调函数才可以，不然这个硬件设备的中断不会被处理。注册硬件设备中断回调函数，LMOSEM 内核驱动模型已经提供了相应的函数，我们直接调用就可以了，如代码清单 9-30[xxxx/lmosem/drivers/drvtick.c] 所示。

代码清单9-30 注册设备中断回调函数

```

.....
//对于systick设备的中断回调函数就是systick_handle，物理中断线号要经过MINT_IFDNR宏转换后才是
//LMOSEM内核所需要的，请参阅中断管理相应的章节，如果注册失败，就返回错误状态
if(krlnew_devhandle(devp,systick_handle,MINT_IFDNR(SYSTICK_PHYINTLINE))==DFCERRSTUS)
{
    //返回错误状态
    return DFCERRSTUS;
}
.....

```

对于一个驱动程序，LMOSEM 内核驱动模型所需要的东西就全部写好了。下面就要去写一写与硬件设备相关的东西了。例如，一个硬件设备在开始运转之前，驱动程序应该首先初始化它，有中断的要操作中断控制器，使 LMOSEM 内核能够响应该设备的中断等。

第一，初始化硬件设备。对于这个 systick 设备，实际上就是初始化 S3C2440A 处理器中的定时器 4。关于定时器 4 硬件，前面已经研究过了。这里直接写代码，如代码清单 9-31[xxxx/lmosem/drivers/drvtick.c] 所示。代码中的一些数值若有不明白的，请仔细思考前面关于定时器 4 硬件及其相关的 4 个寄存器。

代码清单9-31 初始化定时器4硬件

```

void init_s3c2440_timer4()
{
    //读取定时器的配置寄存器0
    u32_t cfgtmp=hal_io32_read(TCFG0_R);
    //清除TCFG0_R寄存器的8~15位
    cfgtmp&=0xffff00ff;
    cfgtmp|=0xff00;
    //设置第二个预分频器为255
    hal_io32_write(TCFG0_R,cfgtmp);
    //读取定时器的配置寄存器1

```

```

    cftgtmp=hal_io32_read(TCFG1_R);
//选择定时器4的选通输入为0000=1/2
    cftgtmp&=(0xfff0ffff);
//写入定时器的配置寄存器1
    hal_io32_write(TCFG1_R,cftgtmp);
//设置定时器4的TCNTB4_R寄存器为0x41
    hal_io32_write(TCNTB4_R,0x41);
    cftgtmp=hal_io32_read(TCON_R);
//这是为了清零TCON_R的20~22位
    cftgtmp&=0xff8fffff;
    hal_io32_write(TCON_R,cftgtmp);
    cftgtmp=hal_io32_read(TCON_R);
//这主要是为了设置TCON_R的20~22位为1,并清零了21位
    cftgtmp|=(0x700000);
    hal_io32_write(TCON_R,cftgtmp);
    cftgtmp=hal_io32_read(TCON_R);
//设置TCON_R的20位为1
    cftgtmp&=(~0x200000);
//写入TCON_R并启动定时器4
    hal_io32_write(TCON_R,cftgtmp);
    return;
}

```

第二,初始化并开启了定时器4,最后要做的就是解除中断控制器中对该设备的中断屏蔽位。在中断管理章节中,我们知道定时器4的中断源屏蔽位的位号,是在主中断源屏蔽寄存器中的第14位上。LMOSEM内核已经提供了打开中断源屏蔽位的函数,直接调用即可,如代码清单9-32[xxxx/lmosem/drivers/drvtick.c]所示。

代码清单9-32 开启定时器4的中断源屏蔽位

```

.....
//检查打开中断屏蔽位是否失败,其中SYSTICK_PHYINTLINE定义为14,物理中断屏蔽位号要经过MINT_
//IFDNR宏转换后才是LMOSEM内核所需要的,请参阅中断管理。
if(krlenable_intline(MINT_IFDNR(SYSTICK_PHYINTLINE))==DFCERRSTUS)
{
    return DFCERRSTUS;
}
.....

```

关于systick设备驱动程序的入口函数要做的事情就这么多。下面把上面的这些代码片段和许多函数,归纳整理进systick驱动程序的入口函数中,如代码清单9-33[xxxx/lmosem/drivers/drvtick.c]所示。

代码清单9-33 systick驱动程序的入口函数

```

drvstus_t systick_entry(driver_t* drvvp,uint_t val,void* p)
{

```

```

//检查LMOSEM内核传递进来的参数
if(drvp==NULL)
{
    return DFCERRSTUS;
}
//创建device_t结构的实例变量
device_t* devp=new_device_dsc();
if(devp==NULL)
{
    return DFCERRSTUS;
}
//设置驱动程序数据结构和设备数据结构中的相关域
systick_set_driver(drvp);
systick_set_device(devp,drvp);
//向驱动程序注册设备
if(krldev_add_driver(devp,drvp)==DFCERRSTUS)
{
//注意释放资源
if(del_device_dsc(devp)==DFCERRSTUS)
{
    return DFCERRSTUS;
}
return DFCERRSTUS;
}
//向LMOSEM内核注册设备
if(krlnew_device(devp)==DFCERRSTUS)
{
//注意释放资源
if(del_device_dsc(devp)==DFCERRSTUS)
{
    return DFCERRSTUS;
}
return DFCERRSTUS;
}
//注册设备中断回调函数
if(krlnew_devhandle(devp,systick_handle,MINT_IFDNR(SYSTICK_PHYINTLINE))==DFCERRSTUS)
{
//注意释放资源
return DFCERRSTUS;
}
//初始化定时器设备
init_s3c2440_timer4();
//解除中断控制器中对该设备的中断屏蔽位
if(krlenable_intline(MINT_IFDNR(SYSTICK_PHYINTLINE))==DFCERRSTUS)
{
    return DFCERRSTUS;
}

```

```

    }
    return DFCOKSTUS;
}

```

systick 设备驱动程序的人口函数总算是写好了。由于 systick 设备是系统内部的设备，它只是为了不间断地产生中断，以记录当前进程运行了多少时间。LMOSEM 内核上层代码不会访问这个设备的其他任何功能，所以那些功能派发函数，只要在其中返回一个错误状态值即可，不必实现实际功能。但是，LMOSEM 内核就是要让它产生中断的，那么它的中断回调函数一定要实现。下面就去实现这个设备的中断回调函数，如代码清单 9-34[xxxx/lmosem/drivers/drvtick.c] 所示。

代码清单9-34 systick设备的中断回调函数

```

drvstus_t systick_handle(uint_t ift_nr,void* devp,void* sframe)
{
    //打印字符串: systick_handle run!!
    printfk("systick_handle run!!\n\r");
    //返回正确的状态值
    return DFCOKSTUS;
}

```

其实到了后面，systick_handle 函数中会调用一个与进程相关的非常重要的函数，不过这里先打印一个字符串，证明该函数是否正常运行了。

一个符合驱动模型和功能要求的驱动程序就实现了。下面就去接上开发板测试一下我们的成果。这个测试顺便也会测试一下前面没有测试的驱动模型和中断管理。所以这次测试的成功将证明很多代码都是正确的。下面就去写好测试代码吧。

9.4.4 测试 systick 驱动程序

前面实现了 systick 驱动程序，但是究竟代码写得对不对，凭感觉是不得而知的。唯一证明代码正确性的方法就是大量、严谨的测试。

要测试 systick 驱动程序，先得把 systick 驱动程序的人口函数，加入到 krlglobal.c 文件中的 osdrvetytbl 数组中。这样，systick 驱动程序才会被加载，这也是前面设计好的。如代码清单 9-35[xxxx/lmosem/kernel/krlglobal.c] 所示。

代码清单9-35 osdrvetytbl数组

```

.....
KRL_DEFGLOB_VARIABLE(drvtyexit_t,osdrvetytbl)[]={
    systick_entry,NULL};

```

LMOSEM 内核代码写到这里，我们从来没有打开 CPU 响应中断的能力，那是因为 LMOSEM 内核还在初始化，不能响应任何中断，但是现在要测试驱动程序了。同时，中断管理和内存管理已经初始化完成了，所以打开 CPU 的 IRQ 中断没有问题了。可是 CPU 的

IRQ 中断在哪里打开呢，当然是 LMOSEM 内核主初始化函数的最后，如代码清单 9-36[xxxx/lmosem/kernel/krlinit.c] 所示。

代码清单9-36 打开CPU的IRQ中断

```
void init_krl()
{
    //初始化内存管理
    init_krlmm();
    //初始化设备管理
    init_krldevice();
    //初始化驱动程序，加载相应的驱动程序运行
    init_krldriver();
    //打开CPU的IRQ中断，该函数在cpuctrl.c文件中
    hal_enable_irq();
    //死循环
    for(;;);
    hal_sysdie("LMOSEM RUN!!");
    return;
}
```

好了，按照前面章节中的方法连接好开发板，配置好 minicom，把编译生成的 lmosem-kernel.bin 文件，即 LMOSEM 内核，下载到开发板中运行。

正常情况下，Linux 的终端中，应该不断地输出字符串：“systick_handle run!!”，这就表示定时器 4 在不断的产生中断并且 systick 驱动程序的中断回调函数 systick_handle 也正确运行了。如图 9-5 所示。

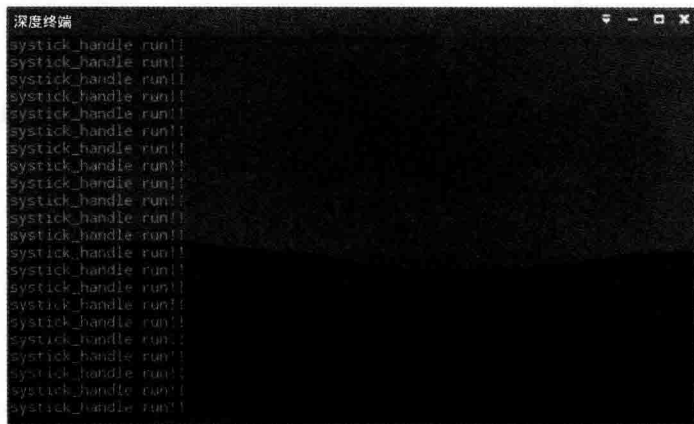


图 9-5 测试 systick 驱动程序

一切正常，这不仅仅表示 systick 驱动程序是正确的，而且驱动模型和中断管理相关的代码也是正确的。那么好，我们接着去实现一个更为复杂的驱动程序。

9.5 RTC 驱动程序实例

现代大部分计算平台都有专门处理时间的硬件，称为 RTC，即时实时钟，表示的是我们通常非常关心的那个时间，例如，现在几点了。要操作这个硬件就需要编写对应的驱动程序。前面开发了 systick 驱动程序，了解了一个驱动程序的开发过程，所以到了这里就简单多了。不过，依然从 RTC 硬件开始，然后直接就实现这个驱动程序了，省略了前面的那些驱动程序框架的东西。

9.5.1 RTC 硬件

编写 RTC 驱动程序，首先要了解 RTC 硬件。关于 RTC 硬件的简介，请参阅前面的相关章节。在 mini2440 开发板上，只有一颗 RTC 备用电池，而完成 RTC 功能的部件在 S3C2440A 处理器内部。既然是表示自然界的时间，那么即使在开发板断电后也不能停止运行，这就是它为什么要有备用电池的原因。

RTC 设备可以通过 8 位 BCD 数值和 CPU 通信。其中包括年、月、日、星期、时、分、秒的时间信息。RTC 设备通过外部 32.768kHz 晶振而工作，并且可以执行闹钟功能。

在了解 RTC 设备的细节之前，先来看看它的功能模块逻辑视图，如图 9-6 所示。

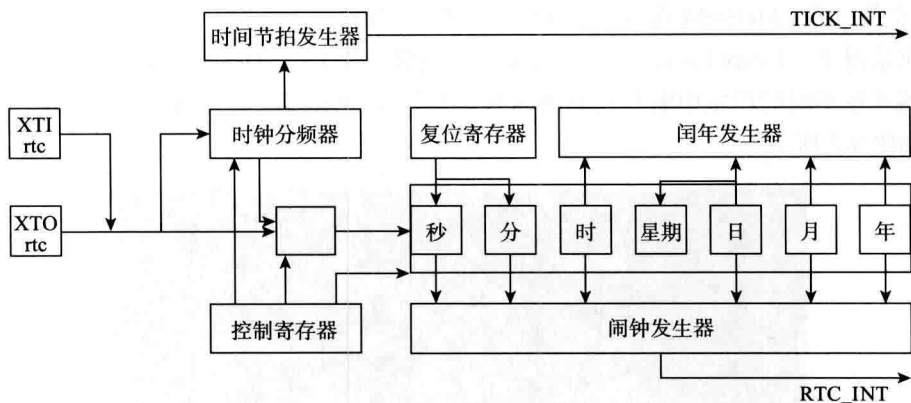


图 9-6 RTC 功能模块

RTC 设备的闰年发生器能够基于日、月、年数据，从 28、29、30 或 31 中决定哪个是每月的最后日。此模块决定最后日时会考虑闰年因素。8 位计数器只能够表示 2 个 BCD 数字，因此不能判断最后两位数为 0 的年份是否为闰年。RTC 设备能在掉电模式中或正常工作模式中的指定时间上产生一个闹钟信号。在正常工作模式中，只激活闹钟中断（RTC_INT）信号；在掉电模式中，除了 RTC_INT 信号之外，还激活电源管理唤醒（PMWKUP）信号。RTC 设备还能产生节拍中断，类似于前面定时器的不间断定时中断。

我们了解了 RTC 设备的结构，并不能让我们编程操控 RTC 设备，想要操控 RTC 设备就

得了解相应的寄存器，如下：

- 1) RTC 控制寄存器 (RTCCON)。
- 2) RTC 节拍时间的计数寄存器 (TICNT)。
- 3) RTC 闹钟的控制寄存器 (RTCALM)。
- 4) RTC 闹钟秒的数据寄存器 (ALMSEC)。
- 5) RTC 闹钟分的数据寄存器 (ALMMIN)。
- 6) RTC 闹钟时的数据寄存器 (ALMHOUR)。
- 7) RTC 闹钟日的数据寄存器 (ALMDATE)。
- 8) RTC 闹钟月的数据寄存器 (ALMMON)。
- 9) RTC 闹钟年的数据寄存器 (ALMYEAR)。
- 10) RTC 时钟的秒数据寄存器 (BCDSEC)。
- 11) RTC 时钟的分数据寄存器 (BCDMIN)。
- 12) RTC 时钟的时数据寄存器 (BCD HOUR)。
- 13) RTC 时钟的日数据寄存器 (BCDDATE)。
- 14) RTC 时钟的星期数据寄存器 (BCDDAY)。
- 15) RTC 时钟的月数据寄存器 (BCDMON)。
- 16) RTC 时钟的年数据寄存器 (BCDYEAR)。

一共 16 个寄存器，不过都很简单，下面就去一个一个把它们搞清楚。

要控制 RTC 设备，首先要了解 RTC 设备的控制寄存器 (RTCCON)。注意这个寄存器的地址是字节地址，如表 9-6 所示。

表9-6 RTC设备的控制寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
RTCCON (0x57000040)	4	读 / 写	RTC 控制寄存器	0
CLKRST	[3]	读 / 写	RTC 时钟计数复位: 0 = 不复位; 1 = 复位	0
CNTSEL	[2]	读 / 写	BCD 计数选择: 0 = 融入 BCD 计数器; 1 = 保留 (分离 BCD 计数器)	0
CLKSEL	[1]	读 / 写	时钟选择: 0 = XTAL1/215 分频时钟; 1 = 保留 (XTAL 时钟只用于测试)	0
RTCEN	[0]	读 / 写	RTC 控制使能; 0 = 禁止; 1 = 使能	0

有时需要 RTC 产生节拍中断，这就需要了解 RTC 节拍时间的计数寄存器 (TICNT)，如表 9-7 所示。

表9-7 RTC节拍时间的计数寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
TICNT (0x57000044)	8	读 / 写	RTC 节拍时间的计数寄存器	0

(续)

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
TICKINT 使能	[7]	读 / 写	节拍时间中断使能: 0 = 禁止; 1 = 使能	0
TICK 时间计数	[6:0]	读 / 写	节拍时间计数值 (1 ~ 127): 该位段的值, 在内部递减, 并且用户不能在工作中读取该值	0

RTC 设备有闹钟功能, 要想使用这个功能, 先得了解 RTC 闹钟的控制寄存器 (RTCALM)。闹钟时间到达时会产生 RTCINT 中断, 如表 9-8 所示。

表9-8 RTC闹钟的控制寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
RTCALM (0x57000050)	8	读 / 写	RTC 闹钟控制寄存器	0
保留	[7]	读 / 写	保留	0
ALMEN	[6]	读 / 写	全局闹钟使能: 0 = 禁止; 1 = 使能	0
YEAREN	[5]	读 / 写	年闹钟使能: 0 = 禁止; 1 = 使能	0
MONREN	[4]	读 / 写	月闹钟使能: 0 = 禁止; 1 = 使能	0
DATEEN	[3]	读 / 写	日闹钟使能: 0 = 禁止; 1 = 使能	0
HOUREN	[2]	读 / 写	时闹钟使能: 0 = 禁止; 1 = 使能	0
MINEN	[1]	读 / 写	分闹钟使能: 0 = 禁止; 1 = 使能	0
SECEN	[0]	读 / 写	秒闹钟使能: 0 = 禁止; 1 = 使能	0

RTC 闹钟秒的数据寄存器 (ALMSEC), 如表 9-9 所示。

表9-9 RTC闹钟秒的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMSEC (0x57000054)	8	读 / 写	RTC 闹钟秒的数据寄存器	0
保留	[7]		保留	0
SECDATA	[6:4]	读 / 写	闹钟秒 BCD 值 0 ~ 5	0
SECDATA	[3:0]	读 / 写	闹钟秒 BCD 值 0 ~ 9	0

RTC 闹钟分的数据寄存器 (ALMMIN), 如表 9-10 所示。

表9-10 RTC闹钟分的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMMIN (0x57000058)	8	读 / 写	RTC 闹钟分的数据寄存器	0
保留	[7]		保留	0
MINDATA	[6:4]	读 / 写	闹钟分 BCD 值 0 ~ 5	0
MINDATA	[3:0]	读 / 写	闹钟分 BCD 值 0 ~ 9	0

RTC 闹钟时的数据寄存器 (ALMHOUR)，如表 9-11 所示。

表9-11 RTC闹钟时的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMHOUR (0x5700005C)	8	读 / 写	RTC 闹钟时的数据寄存器	0
保留	[7:6]		保留	0
HOURLDATA	[5:4]	读 / 写	闹钟时 BCD 值 0 ~ 2	0
HOURLDATA	[3:0]	读 / 写	闹钟时 BCD 值 0 ~ 9	0

RTC 闹钟日的数据寄存器 (ALMDATE)，如表 9-12 所示。

表9-12 RTC闹钟日的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMDATE(0x57000060)	8	读 / 写	RTC 闹钟日的数据寄存器	1
保留	[7:6]		保留	0
DATEDATA	[5:4]	读 / 写	闹钟日 BCD 值 0 ~ 3	0
DATEDATA	[3:0]	读 / 写	闹钟日 BCD 值 0 ~ 9	1

RTC 闹钟月的数据寄存器 (ALMMON)，如表 9-13 所示。

表9-13 RTC闹钟月的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMMON (0x57000064)	8	读 / 写	RTC 闹钟月的数据寄存器	1
保留	[7:5]		保留	0
MONDATA	[4]	读 / 写	闹钟月 BCD 值 0 ~ 1	0
MONDATA	[3:0]	读 / 写	闹钟月 BCD 值 0 ~ 9	1

RTC 闹钟年的数据寄存器 (ALMYEAR)，如表 9-14 所示。

表9-14 RTC闹钟年的数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
ALMYEAR (0x57000068)	8	读 / 写	RTC 闹钟年的数据寄存器	0
YEARDATA	[7:0]	读 / 写	闹钟年 BCD 值 00 ~ 99	0

我们再来看看记录时间的寄存器。首先是 RTC 时钟的秒数据寄存器 (BCDSEC)，如表 9-15 所示。

表9-15 RTC时钟的秒数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDSEC (0x57000070)	8	读 / 写	RTC 时钟的秒数据寄存器	未知
保留	[7]		保留	未知
SECDATA	[6:4]	读 / 写	秒 BCD 值 0 ~ 5	未知
SECDATA	[3:0]	读 / 写	秒 BCD 值 0 ~ 9	未知

RTC 时钟的分数据寄存器 (BCDMIN), 如表 9-16 所示。

表9-16 RTC时钟的分数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDMIN(0x57000074)	8	读 / 写	RTC 时钟的分数据寄存器	未知
保留	[7]		保留	未知
MINDATA	[6:4]	读 / 写	分 BCD 值 0 ~ 5	未知
MINDATA	[3:0]	读 / 写	分 BCD 值 0 ~ 9	未知

RTC 时钟的时数据寄存器 (BCD HOUR), 如表 9-17 所示。

表9-17 RTC时钟的时数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCD HOUR (0x57000078)	8	读 / 写	RTC 时钟的时数据寄存器	未知
保留	[7:6]		保留	未知
HOURLDATA	[5:4]	读 / 写	时 BCD 值 0 ~ 2	未知
HOURLDATA	[3:0]	读 / 写	时 BCD 值 0 ~ 9	未知

RTC 时钟的日数据寄存器 (BCDDATE), 如表 9-18 所示。

表9-18 RTC时钟的日数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDDATE (0x5700007c)	8	读 / 写	RTC 时钟的日数据寄存器	未知
保留	[7:6]		保留	未知
DATEDATA	[5:4]	读 / 写	日 BCD 值 0 ~ 3	未知
DATEDATA	[3:0]	读 / 写	日 BCD 值 0 ~ 9	未知

RTC 时钟的星期数据寄存器 (BCDDAY), 如表 9-19 所示。

表9-19 RTC时钟的星期数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDDAY (0x57000080)	8	读 / 写	RTC 时钟的星期数据寄存器	未知
保留	[7:3]		保留	未知
DAYDATA	[2:0]	读 / 写	星期 BCD 值 1 ~ 7	未知

RTC 时钟的月数据寄存器 (BCDMON)，如表 9-20 所示。

表9-20 RTC时钟的月数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDMON (0x57000084)	8	读 / 写	RTC 时钟的月数据寄存器	未知
保留	[7:5]		保留	未知
MONDATA	[4]	读 / 写	月 BCD 值 1	未知
MONDATA	[3:0]	读 / 写	月 BCD 值 0 ~ 9	未知

RTC 时钟的年数据寄存器 (BCDYEAR)，如表 9-21 所示。

表9-21 RTC时钟的年数据寄存器

寄存器 (地址)	位	是否可读 / 写	描 述	初始值
BCDYEAR (0x57000088)	8	读 / 写	RTC 时钟的年数据寄存器	未知
YEARDATA	[7:0]	读 / 写	年 BCD 值 00 ~ 99	未知

至此，我们了解了 RTC 设备的 16 个寄存器的详细情况，了解了这些寄存器就等于了解了 RTC 设备的编程接口。需要注意的是，RTC 设备会产生两种中断：一种是 RTC_INT 闹钟中断；一种是 RTC_TICK_INT 节拍中断。

清楚了 RTC 设备的功能结构图，接着又了解了与它相关的 16 个寄存器，下面就去设计驱动程序，编写操作这个 RTC 设备的相关代码。

9.5.2 RTC 驱动程序实现

前面了解了 RTC 设备，下面就要按照 LMOSEM 内核驱动模型的要求，去实现 RTC 驱动程序了。对于驱动程序的编写要求和框架，请参阅 systick 驱动程序。为了节约篇幅，这里不再详述。

在开始之前，由于 RTC 是另一个设备，所以要在“xxx/lmosem/drivers”目录下新建一个 C 语言源代码文件：drvrtc.c，并在“xxx/lmosem/include/drvinc”目录下建立 drvrtc_t.h、drvrtc.h 两个头文件，同时按照前面的方法组织好它们。还要改写 buildfile.h 文件中的相关宏，即 #define BUILD_DRIV_OBJS drvtick.o drvrtc.o。

准备工作做好之后，先来看一看，对于 RTC 设备要完成哪些功能，其列于下面：

- ❑ 更新维护系统时间。
- ❑ 能读取时间。
- ❑ 能设置时间。
- ❑ 能设置闹钟。

了解了要实现的功能之后,下面先来写好 RTC 驱动程序的人口函数,不过这里只会列出人口函数相关的代码。其他函数的代码就不列出了,可以自行参阅源代码文件。RTC 驱动程序的人口函数,如代码清单 9-37 [xxxx/lmosem/drivers/drvrtc.c] 所示。

代码清单9-37 RTC驱动程序的入口函数

```
void rtc_set_device(device_t* devp, driver_t* drvvp)
{
    devp->dev_flg=DEVFLG_SHARE;
    devp->dev_stus=DEVSTS_NORML;
    //设置设备类型为RTC_DEVICE类型
    devp->dev_id.dev_mtype=RTC_DEVICE;
    devp->dev_id.dev_stype=0;
    devp->dev_id.dev_nr=0;
    //设置设备名为rtc
    devp->dev_name="rtc";
    return;
}

void init_rtc()
{
    //开启RTC设备
    hal_io8_write(RTCCON_R,1);
    //开启RTC节拍中断,并设置节拍时间计数值为127
    hal_io8_write(TICNT_R,RTCTICK_INT_EN|127);
    return;
}

drvstus_t rtc_entry(driver_t* drvvp, uint_t val, void* p)
{
    //检查LMOSEM内核传递进来的参数
    if(drvvp==NULL)
    {
        return DFCERRSTUS;
    }
    //创建device_t结构的实例变量
    device_t* devp=new_device_dsc();
    if(devp==NULL)
    {
        return DFCERRSTUS;
    }
    //设置驱动程序数据结构和设备数据结构中的相关域
    rtc_set_driver(drvvp);
```

```

rtc_set_device(devp, drv);
//向驱动程序注册设备
    if(krldrv_add_driver(devp, drv)==DFCERRSTUS)
    {
//注意释放资源
        if(del_device_dsc(devp)==DFCERRSTUS)
        {
            return DFCERRSTUS;
        }
        return DFCERRSTUS;
    }
//向LMOSEM内核注册设备
    if(krlnw_device(devp)==DFCERRSTUS)
    {
//注意释放资源
        if(del_device_dsc(devp)==DFCERRSTUS)
        {
            return DFCERRSTUS;
        }
        return DFCERRSTUS;
    }
//注册第一个设备中断回调函数
    if(krlnw_devhandle(devp, rtc_handle, MINT_IFDNR(RTC_INT_PHYLINE))==DFCERRSTUS)
    {
//注意释放资源
return DFCERRSTUS;
    }
//注册第二个设备中断回调函数，因为RTC设备有两个中断源
    if(krlnw_devhandle(devp, rtc_tick_handle, MINT_IFDNR(RTCTICK_INT_PHYLINE))==DFCERRSTUS)
    {
//注意释放资源
return DFCERRSTUS;
    }
//解除中断控制器中对该设备的中断屏蔽位一
    if(krlenable_intline(MINT_IFDNR(RTC_INT_PHYLINE))==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
//解除中断控制器中对该设备的中断屏蔽位二
    if(krlenable_intline(MINT_IFDNR(RTCTICK_INT_PHYLINE))==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
//初始化RTC设备本身
    init_rtc();
    return DFCOKSTUS;
}

```

代码清单 9-37 和 systick 驱动程序的入口函数是一样的,除了 RTC 设备有两个中断源,关于这两个中断源,可以参阅前面的中断管理章节或者 S3C2440A 手册。

写好了 RTC 驱动程序的入口函数,下面就要实现相应的功能了。首先来看一看第一个功能:更新维护系统时间。这里要先切入另一个主题:LMOSEM 内核是如何管理时间的。LMOSEM 内核管理时间其实相当简单,就是设计一个数据结构记录当前的时间,同时需要不断地更新这个时间。先来看一看这个数据结构,如代码清单 9-38[xxxx/lmosem/include/knlinc/ktime_t.h] 所示。

代码清单9-38 LMOSEM内核时间数据结构

```
typedef struct s_KTIME
{
    spinlock_t    kt_lock; //自旋锁
    uint_t        kt_year; //年
    uint_t        kt_mon;  //月
    uint_t        kt_day;  //星期
    uint_t        kt_date; //天
    uint_t        kt_hour; //时
    uint_t        kt_min;  //分
    uint_t        kt_sec;  //秒
    void*         kt_datap; //用于扩展
}ktime_t;
```

ktime_t 结构也是一个 LMOSEM 内核的全局性的数据结构,所以它的实例变量定义在 krlglobal.c 文件中,如代码清单 9-39[xxxx/lmosem/kernel/krlglobal.c] 所示。

代码清单9-39 定义全局的ktime_t结构类型的变量osktime

```
.....
KRL_DEFGLOB_VARIABLE(ktime_t,osktime);
.....
```

ktime_t 结构实例变量中的数据,一定要不断更新,才能保证时间的准确性。对于这个更新时间的长短,可以是一秒钟更新一次,甚至是更短的更新时间。所以要实现一个更新时间的函数才行,如代码清单 9-40[xxxx/lmosem/kernel/krltime.c] 所示。

代码清单9-40 更新LMOSEM内核时间数据

```
void krlupdate_times(uint_t year,uint_t mon,uint_t day,uint_t date,uint_t
hour,uint_t min,uint_t sec)
{
    ktime_t* initp=&osktime;
    cpuflg_t cpufg;
    hal_spinlock_saveflg_cli(&initp->kt_lock,&cpufg);
    initp->kt_year=year;
    initp->kt_mon=mon;
```

```

initp->kt_day=day;
initp->kt_date=date;
initp->kt_hour=hour;
initp->kt_min=min;
initp->kt_sec=sec;
hal_spinunlock_restflg_sti(&initp->kt_lock, &cpufig);
return;
}

```

`krlupdate_times` 函数非常简单，不必详述。

前面说了，要不断地更新时间才行，可是怎么不断地更新时间呢。读者可能已经想到了，用 RTC 设备的节拍中断，每发生一次 RTC 节拍中断，就把 RTC 中的时间写入到 `ktime_t` 结构实例变量的相关域中就可以了。这个动作当然是在 RTC 设备节拍中断回调函数中完成的。下面我们来写好 RTC 设备的节拍中断回调函数，如代码清单 9-41[xxxx/lmosem/drivers/drvrtc.c] 所示。

代码清单9-41 RTC设备的节拍中断回调函数

```

drvstus_t rtc_tick_handle(uint_t ift_nr, void *devp, void *sframe)
{
    //读取RTC中的年、月、星期、日、时、分、秒数据寄存器中的值
    uint_t year=hal_io8_read(BCDYER_R);
    uint_t mon=hal_io8_read(BCDMON_R);
    uint_t day=hal_io8_read(BCDDAY_R);
    uint_t date=hal_io8_read(BCDDAT_R);
    uint_t hour=hal_io8_read(BCDHOR_R);
    uint_t min=hal_io8_read(BCDMIN_R);
    uint_t sec=hal_io8_read(BCDSEC_R);
    //把BCD数据转换成二进制数据
    BCD_TO_BIN(year);
    BCD_TO_BIN(mon);
    BCD_TO_BIN(day);
    BCD_TO_BIN(date);
    BCD_TO_BIN(hour);
    BCD_TO_BIN(min);
    BCD_TO_BIN(sec);
    //因为RTC的年数据寄存器中的值是从2000年开始的，所以要加上2000
    year+=2000;
    //调用LMOSEM内核更新时间的函数
    krlupdate_times(year,mon,day,date,hour,min,sec);
    return DFCOKSTUS;
}

```

代码清单 9-41 中的 `BCD_TO_BIN` 是一个宏，定义在 `drvrtc_t.h` 文件中，用于把 BCD 数据转换成二进制数据。关于 BCD 数据，请自行参阅有关资料，它就是一种数据编码的方式，

这里不会详述。最后 `rtc_tick_handle` 函数会调用 `krlupdate_times` 函数更新 LMOSEM 内核的时间数据。

有时候应用程序希望直接从 RTC 设备读取时间，这时 LMOSEM 内核上层代码，会发送一个读操作的 IO 包到 RTC 设备驱动程序，所以要实现 RTC 驱动程序的读功能派发函数。不过，LMOSEM 内核驱动模型，对这种时间数据格式有一定的要求，所以 LMOSEM 内核驱动模型，为此定义了一个数据结构，告诉 RTC 驱动程序要把数据填入这个数据结构中，这个数据结构，如代码清单 9-42[xxxx/lmosem/include/knlinc/ktime_t.h] 所示。并且要求这个数据结构的实例变量，存放在 IO 包所提供的缓冲区中。

代码清单9-42 应用程序使用的时间数据结构

```
typedef struct s_TIME
{
    uint_t    year;
    uint_t    mon;
    uint_t    day;
    uint_t    date;
    uint_t    hour;
    uint_t    min;
    uint_t    sec;
}time_t;
```

下面来实现 RTC 驱动程序的读功能派发函数，如代码清单 9-43[xxxx/lmosem/drivers/drvrtc.c] 所示。

代码清单9-43 RTC驱动程序的读功能派发函数

```
drvstus_t rtc_read(device_t* devp,void* iopack)
{
    uint_t len;
    buf_t retbuf;
    //获取IO包中的缓冲区地址和缓冲区长度
    if(krldev_retn_rqueparm(iopack,&retbuf,NULL,&len,NULL,NULL,NULL)==DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
    //检查缓冲区地址和缓冲区长度是否合乎要求
    if(retbuf==NULL||len<sizeof(time_t))
    {
        return DFCERRSTUS;
    }
    //读取RTC中的年、月、星期、日、时、分、秒数据寄存器中的值
    uint_t year=hal_io8_read(BCDYER_R);
    uint_t mon=hal_io8_read(BCDMON_R);
    uint_t day=hal_io8_read(BCDDAY_R);
```

```

uint_t date=hal_io8_read(BCDDAT_R);
uint_t hour=hal_io8_read(BCDHOR_R);
uint_t min=hal_io8_read(BCDMIN_R);
uint_t sec=hal_io8_read(BCDSEC_R);
//把BCD数据转换成二进制数据
BCD_TO_BIN(year);
BCD_TO_BIN(mon);
BCD_TO_BIN(day);
BCD_TO_BIN(date);
BCD_TO_BIN(hour);
BCD_TO_BIN(min);
BCD_TO_BIN(sec);
//把缓冲区的地址强制成time_t结构实例变量的地址
time_t* times=(time_t*)retbuf;
//因为RTC的年数据寄存器中的值是从2000年开始的，所以要加上2000
//然后把数据写入到缓冲区中
times->year=2000+year;
times->mon=mon;
times->day=day;
times->date=date;
times->hour=hour;
times->min=min;
times->sec=sec;
//返回
return DFCOKSTUS;
}

```

RTC 驱动程序读功能派发函数的代码很简单。总之就是要把时间数据读取出来，转换成二进制数据，最后把转换好的数据，按一定的格式写入 IO 包所提供的缓冲区中。

可能因为一些特殊原因，导致 RTC 设备里的时间与自然界的标准时间不符合了，所以需要能重新设置时间。对于驱动程序来说，该功能既不是读功能也不是写功能，它应该是控制功能方面的，所以驱动程序要实现控制功能派发函数。当然控制功能就应该有相关的控制码。这个控制码是 LMOSEM 内核驱动模型提供并要求的，如代码清单 9-44[xxxx/lmosem/include/drvinc/drvrtc_t.h] 所示。

代码清单9-44 RTC驱动程序的控制码

```

.....
#define IOCTRCODE_SETTIME 1 //设置时间的
#define IOCTRCODE_SETUPFREQ 2 //设置RTC节拍中断次数的
#define IOCTRCODE_SET_SYNSECALM 3 //设置闹钟的

```

下面来实现 RTC 驱动程序的控制功能派发函数，如代码清单 9-45[xxxx/lmosem/drivers/drvrtc.c] 所示。由于设置时间，需要有正确的时间数据，这个时间数据就是通过 IO 包的缓冲区传递进来的。

代码清单9-45 RTC驱动程序的设置时间函数、控制功能派发函数

```

// RTC驱动程序的设置时间函数，会进一步调用该函数设置时间
drvstus_t rtc_settime(time_t* times)
{
    //开启RTC
    hal_io8_write(RTCCON_R,1);
    //获取要设置的时间值
    uint_t year=times->year;
    uint_t mon=times->mon;
    uint_t day=times->day;
    uint_t date=times->date;
    uint_t hour=times->hour;
    uint_t min=times->min;
    uint_t sec=times->sec;
    //都转换成BCD数据，因为RTC设备只认识BCD数据
    BIN_TO_BCD(year);
    BIN_TO_BCD(mon);
    BIN_TO_BCD(day);
    BIN_TO_BCD(date);
    BIN_TO_BCD(hour);
    BIN_TO_BCD(min);
    BIN_TO_BCD(sec);
    //依次把转换后的数据写入相应的寄存器中
    hal_io8_write(BCDYER_R, (u8_t)year);
    hal_io8_write(BCDMON_R, (u8_t)mon);
    hal_io8_write(BCDDAY_R, (u8_t)day);
    hal_io8_write(BCDDAT_R, (u8_t)date);
    hal_io8_write(BCDHOR_R, (u8_t)hour);
    hal_io8_write(BCDMIN_R, (u8_t)min);
    hal_io8_write(BCDSEC_R, (u8_t)sec);
    //返回
    return DFCOKSTUS;
}

drvstus_t rtc_ioctl(device_t* devp,void* iopack)
{
    uint_t retiocld=0;
    buf_t retbuf=NULL;
    //获取IO包中的缓冲区地址和控制码
    if(krldev_retn_rqueparm(iopack,&retbuf,NULL,NULL,&retiocld,NULL,NULL)==
    DFCERRSTUS)
    {
        return DFCERRSTUS;
    }
    //例行检查
    if(retbuf==NULL)
    {

```

```

    return DFCERRSTUS;
}
//设置闹钟的
if (retiocld==IOCTRCODE_SET_SYNSECALM)
{
    return rtc_set_synsecalm(devp,iopack,*((uint_t*)(retbuf)));
}
//设置时间的
if (retiocld==IOCTRCODE_SETTIME)
{
    return rtc_settime((time_t*)retbuf);
}
//设置RTC节拍中断次数的
if (retiocld==IOCTRCODE_SETUPFREQ)
{
    return rtc_upfreq(*((uint_t*)(retbuf)));
}
return DFCERRSTUS;//返回
}

```

有时应用程序需要睡眠一定的时间后继续运行，这就要用到闹钟功能，所以 RTC 驱动程序要让 RTC 设备能设置闹钟功能。而闹钟功能也应该放在驱动程序的控制功能派发函数中实现。正如代码清单 9-45 中所看到的那样，如果控制码是 IOCTRCODE_SET_SYNSECALM，就会调用 rtc_set_synsecalm 函数设置闹钟，如代码清单 9-46[xxxx/lmosem/drivers/drvrtc.c] 所示。

代码清单9-46 设置闹钟

```

drvstus_t rtc_set_synsecalm(device_t* devp,void* iopack, uint_t sec)
{
    //检查是否大于60s，这里为了简单，暂时只支持设置秒闹钟
    if (sec>=60)
    {
        return DFCERRSTUS;
    }
    //开启RTC
    hal_io8_write(RTCCON_R,1);
    //先读取当前时间
    uint_t year=hal_io8_read(BCDYER_R);
    uint_t mon=hal_io8_read(BCDMON_R);
    uint_t date=hal_io8_read(BCDDAT_R);
    uint_t hour=hal_io8_read(BCDHOR_R);
    uint_t min=hal_io8_read(BCDMIN_R);
    uint_t tmpsec=hal_io8_read(BCDSEC_R);
    //转换一下
    BCD_TO_BIN(tmpsec);

```

```

//当前秒数加上要设置的闹钟秒数，即当前时刻下过去多少秒后闹钟
tmpsec+=sec;
//再转换一下
BIN_TO_BCD(tmpsec);
//把时间写入RTC的闹钟寄存器中
    hal_io8_write(ALMYER_R, (u8_t)year);
    hal_io8_write(ALMMON_R, (u8_t)mon);
    hal_io8_write(ALMDAT_R, (u8_t)date);
    hal_io8_write(ALMHOR_R, (u8_t)hour);
    hal_io8_write(ALMMIN_R, (u8_t)min);
    hal_io8_write(ALMSEC_R, (u8_t)tmpsec);
//把IO包挂载到设备服务等待链表上，可能进程要睡眠
    if(krldrv_add_request(devp, (objnode_t*)iopack)==DFCERRSTUS)
    {
        hal_sysdie("rtc_setsynalm run");
        return DFCERRSTUS;
    }
//开启闹钟
hal_io8_write(RTCALM_R, RTCALM_EN|RTCSECALM_EN);
//让进程睡眠等待
krldrv_wait_request(devp, (objnode_t*)iopack);
//返回
    return DFCOKSTUS;
}

```

代码清单 9-46 中的注释已经很明显了，不必过多解释。当闹钟的时间到达时，RTC 会发出 RTC_INT 中断，所以还要写一个 RTC_INT 中断回调函数，在这个中断回调函数中完成闹钟功能服务并唤醒睡眠的进程，如代码清单 9-47[xxxx/lmosem/drivers/drvrtc.c] 所示。

代码清单9-47 RTC_INT中断回调函数

```

drvstus_t rtc_handle(uint_t ift_nr, void* devp, void* sframe)
{
    objnode_t* request;
    //先看一看设备中有没有等待的IO包
    if(((device_t*)devp)->dev_rqlnr>0)
    {
        //获取具体的IO包，如果出问题就死机
        if(krldrv_retn_request((device_t*)devp, IOIF_CODE_IOCTL, &request)==DFCERRSTUS)
        {
            hal_sysdie("rtc hand run err no retn request");
        }
        //完成服务，并唤醒睡眠的进程，如果出问题就死机
        if(krldrv_complete_request((device_t*)devp, request)==DFCERRSTUS)
        {
            hal_sysdie("rtc hand run err no complete_request");
        }
    }
}

```

```

    }
    return DFCOKSTUS;
}

```

虽然没有列出所有 RTC 驱动程序的代码，但是用于完成开始规定的四大功能的代码，已经一步步写好了。但是要记住，把 `rtc_entry` 函数，加入到 `osdrvetytbl` 数组中，不然 RTC 驱动程序不会被加载运行。

好了，一个简单且完整的 RTC 驱动程序就完成了，到这里我们已经完成了两个设备驱动程序，也运用且了解了驱动模型的大部分特性，另外还有一个串口驱动程序，也重新实现了，不过为了节约篇幅，加上前面 `printfk` 函数相关的章节中，已经了解过串口硬件，理解起来也很容易，所以我们不仔细研究了，读者有兴趣的话可以结合 `systick`、RTC 驱动程序的开发知识，自行阅读 `drvuart.c` 文件中的代码。

9.6 小结

LMOSEM 内核的驱动模型，到这里就要结束了，不妨回过头去看一看对于 LMOSEM 内核的驱动模型都做了一些什么。大致如下：

- 1) 由于计算平台之上的设备众多，我们提出了分权而治的想法。
- 2) 从逻辑上定义了一些 LMOSEM 内核设备类型，例如，串口、RTC、CPU、RAM、FS 等。
- 3) 操控设备需要一些代码，我们把这些代码规定成固定的结构，称为设备驱动程序。
- 4) 设计了一系列的数据结构，如驱动、设备、设备 ID、IO 包、设备表等。
- 5) 为了方便驱动程序开发和 LMOSEM 内核能有效地管理设备，由此，编写了很多驱动模型的基础代码，如新建与注册设备、注册中断回调函数、发送 IO 包、调用驱动程序功能派发函数等。
- 6) 编写了 `systick`、RTC 两个设备驱动程序实例。

看上去做的事确实不少，但是，这些对于一个完整的驱动模型来说，不过是冰山一角。我们没有对某个设备做具体的功能抽象，对于前面的 RTC 驱动程序，读者已经感觉到了，它不仅是要读取时间，还要能设置时间，设置闹钟。再如，存储设备，驱动模型应该规定一些控制码，以便返回其存储大小等。而键盘设备，驱动模型应该规定一些控制码，以便控制按键速率等。这些都是驱动模型该做的，但是为了简单，我们只是做了 LMOSEM 内核怎么组织、访问设备的部分，当然这是基础。

驱动模型还有很多东西没有完成。不过 LMOSEM 内核总算可以加载驱动程序、创建设备了，并且有了一些基本的设备驱动程序，其他东西可以留下来以后慢慢细化也不迟，但是现在我们实在忍不住要探索新的东西了。

进 程

进程是 20 世纪 60 年代由麻省理工学院提出的，是现代操作系统中非常重要的概念。因此，操作系统理论书籍，大多会从进程开始说起：进程是什么、进程状态、进程间通信、各种进程调度算法等。

没有进程概念的操作系统，就不能称为具有现代意义的操作系统，进程如此重要，到这里才开始探索，似乎有点姗姗来迟。最后我们会发现，进程的实现需要内存管理、中断、设备的支持，如果没有实现内存管理、中断管理、驱动模型，进程只能停留在宏伟的理论层面，看上去就像那在水一方的窈窕淑女，虽然美丽，却遥不可及。

本章先从应用程序运行的角度出发，看一看进程模型的提出，接着设计相关的数据结构，然后了解操作系统空转进程及空转进程的运行，最后解决创建一个新进程和进程调度的问题。

在了解了硬件平台，又做好了前面的那些工作之后，相信本章会让你觉得进程虽然重要，但是实现却并不困难。

10.1 应用程序的运行

静态的应用程序和进程似乎没有什么关系，确实如此。但是要搞清楚前辈在什么样的情况下提出进程概念的，又是为了解决什么问题，这就要从应用程序的运行开始。

先从应用程序运行需要什么资源开始，然后看看这些资源在任何时刻下是不是都可用，最后提出多道程序模型。其实最后我们会发现，正是因为一些资源在某些时刻不可用，才需要提出多道程序模型以解决这类问题，从而提高计算机效率。

在开始之前，希望读者在脑中建立这样一种视角，也是前面反复提到的：计算机内部有 CPU、内存、其他设备、各种数据，把这些东西统称为资源，而应用程序是资源的使用者，操作系统内核是资源的管理者或提供者。

10.1.1 程序运行需要什么资源

通常在 Windows 下单击鼠标就能打开一个应用程序，我们虽然清楚这个应用程序是干什么的，但好像不明白这个应用程序运行时，需要一些什么资源。当然，一个应用程序完成的功能不同，需要的资源可能也不同。

在观察应用程序运行时需要什么资源之前，先来看看一个静态的应用程序本身的结构是怎样的，一个应用程序无论它多么简单或者多么庞大复杂，在被编译器和链接器处理后，大多都是如图 10-1 所示的结构。

其实，一个静态的应用程序，在系统中是以文件的形式存在的，例如，我们经常可以从硬盘中找到一个应用程序的可执行文件。这个文件中包括用于存放应用程序相关信息的应用程序头、应用程序的代码部分和数据部分。当然代码部分和数据部分随着应用程序的规模不同而或大或小。数据部分可能包含很多类型的数据，如字符串、程序符号表等。

应用程序文件
应用程序头
代码部分
数据部分

图 10-1 静态应用程序结构

下面来简单归纳应用程序在运行时刻需要什么资源。如果细分的话，操作系统内的信号量、文件描述符等都是一种资源，那样就不利于探索问题本身了，所以这里分得非常粗略，只选择几种必要的资源，如下：

- 1) 内存。在前面看到，应用程序有代码部分和数据部分，都必须装载到内存中才能运行，而且运行时刻还需要分配栈空间，用于函数调用或者存放临时数据。
- 2) CPU。因为 CPU 就是用来执行应用程序的代码指令的，所以没有 CPU，应用程序将不能做任何事情。
- 3) 文件。如果应用程序要读写文件，那么它就需要文件这种资源。当然这还需要存储设备资源的支持。
- 4) 网络。如果应用程序要访问网络，那么它就需要网络这种资源，如 QQ 应用程序。
- 5) 音频。如果应用程序是播放音乐的，那么它就需要音频设备这种资源，如 KuGou 应用程序。
- 6) 视频设备。如果应用程序是播放视频的，那么它就需要视频设备这种资源，如 Windows Media Player 应用程序。

从上面的几种资源来看，对于一个应用程序的运行，有些资源是必要的，如内存和 CPU，没有它们应用程序根本不能运行，更别说完成相应的功能了。而有些资源是根据应用程序完成的功能来决定的。

到这里，只要知道，静态应用程序的结构和应用程序运行时刻需要一些什么资源就行了。下面就去看看上述这些资源，在任意时刻下都可以被应用程序使用吗？

10.1.2 任何时刻资源都可用吗

前面章节实现了内存管理、中断管理、驱动模型及设备管理，这也间接验证了上面所说的：操作系统内核是资源的管理者，应用程序是资源的使用者。从 10.1.1 小节已经知道应用程序的运行需要一些资源，那么在任何时刻下需要的这些资源都是可用的吗？带着这个疑问，我们继续。

既然操作系统内核是资源的管理者，应用程序是资源的使用者。首先就来搞清楚应用程序使用资源的模型。这种问题还是画一幅图比较直观，如图 10-2 所示。

一个计算平台内，远不止这几种硬件，为了简化，图 10-2 没有画出更多的硬件。例如，我们常用的键盘、鼠标，它们属于输入输出设备。这些硬件都称为资源，当然存在于磁盘中的文件也是一种资源。不难看出，应用程序正是通过操作系统来访问使用这些资源的。

众所周知，CPU 是目前绝大多数计算平台上最快的设备，并且它能执行访问设备的程序指令。于是就带来了问题：CPU 执行程序中访问硬盘的指令，可是硬盘准备好了吗？因为硬盘是机械设备，它需要旋转盘片和移动磁头，所以要花大量的时间做这些事情。在 CPU 执行程序中，访问键盘的指令，键盘被按下才能产生数据，如果没有按下键盘，CPU 访问键盘是有问题的。其实，网络、音频、显示设备都是一样的。所以应用程序通过操作系统访问的资源并不是任何时刻下都可用。

由上所述，操作系统内核必须要维护所有资源的状态：当前时刻下某一资源是否可以满足应用程序的请求。若不能满足，该怎么办？对于不能满足的情况，只有一种办法——等。等的方式有多种，一种是让 CPU 执行循环代码，在循环代码中检测资源状态，直到资源可以满足请求，就退出循环代码；另一种办法是让 CPU 停止执行代码转而睡眠，直到资源准备好后，就产生一个中断，让中断信号唤醒 CPU，从而继续执行代码。因此要改进应用程序请求使用资源的模型，如图 10-3 所示。

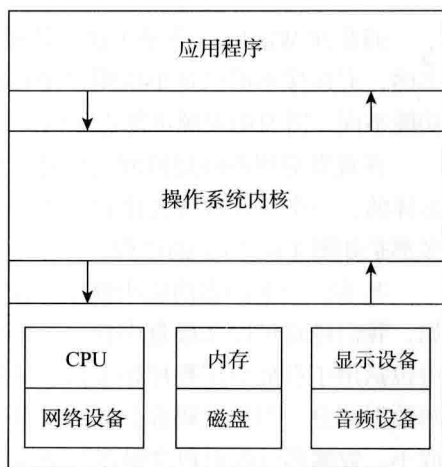


图 10-2 应用程序使用资源的模型

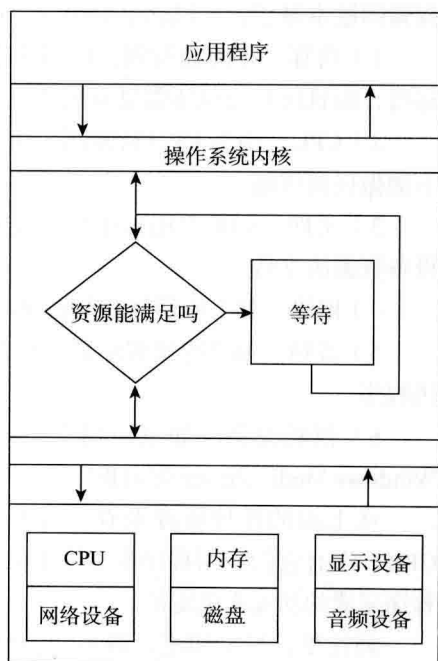


图 10-3 应用程序等待资源可用

图 10-3 中展示了应用程序通过操作系统访问资源，而资源又不能立即被使用的情况。在资源不能使用的情况下，操作系统只能让其进入等待，使其代码的执行流不能返回。不难发现，CPU 的速度越快，在等待上浪费的时间就越多，并且如果应用程序只是访问硬盘资源，那么就算其他设备资源已经准备就绪了，也没有任何作用。

如果永远是上述模型，我们会发现无论计算平台上的硬件性能多么强劲，这个计算平台的整体效率是上不去的，因为这种模型不能让计算平台内所有设备都忙起来。那么在资源不能满足应用程序请求时，应用程序就只能等待吗？先留个悬念，这个问题在后面我们继续探索。

10.1.3 提出多道程序模型

在 10.1.2 节的模型中，一道应用程序访问计算平台上的资源得不到满足时，只能进行等待，没有其他办法。并且如果当前访问的资源不能满足，其他资源就算可用，也无济于事。

真的只能这样吗？当然不是，我们改变一下想法，假设内存中装入了多道应用程序会是什么结果呢，你可能马上就会想到，如果一道应用程序访问资源得不到满足，就不用进入等待了，而是可以切换到另一道应用程序上运行，兴许另一道应用程序在做其他事情，使用其他资源，而其他资源是可能满足另一道应用程序。有了这个想法，就可以提出多道应用程序模型了，如图 10-4 所示。

为了说明原理，图 10-4 画得比较简单，为了更好地理解这一过程，下面来看一个情景：你正在用 QQ 软件和朋友聊天，QQ 软件等待你输入消息，然后把这条消息通过网络发送给你的朋友。QQ 软件等待你的输入，这个时间可能是几秒钟，甚至更久。如果这时内存中存在着另一个 KuGou 软件，这几秒钟就可以切换到 KuGou 软件上，等到用户完成消息输入时，再切换回来，于是就可以一边聊天，一边听音乐了。

通过上面的情景不难发现，QQ 和 KuGou 这两个软件，使计算平台内的键盘设备、网络设备、音频设备、显示设备都工作起来了，这大大提升了计算平台的整体效率。

在引入多道程序模型后，在计算平台中运行了多个应用程序，使得其中的众多资源都被利用起来了，还能让用户感觉到多个应用程序在并行运行。但是这仅仅是存在于我们大脑之中的一个美好的模型，还需要具体去实现它。

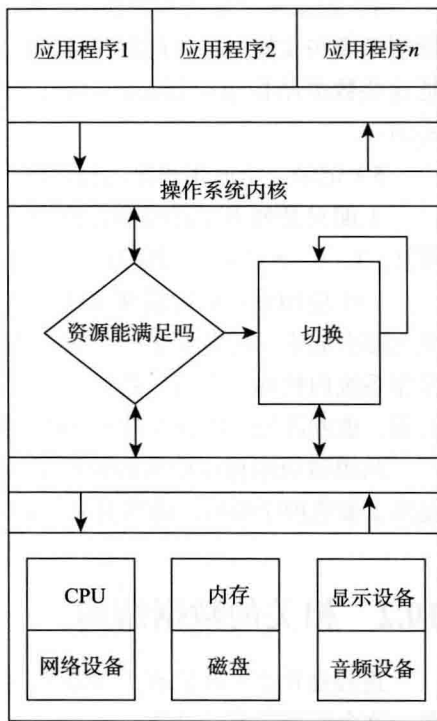


图 10-4 多道应用程序使用资源的模型

在实现多道程序模型之前，要思考一些问题，当然应用程序运行时需要的资源前面已经探讨了，但是还有一些问题，如下：

1) 操作系统内核要维护各种资源的状态，例如，应用程序要使用内存和设备资源，这时操作系统内核中的内存管理组件，就要分配内存空间资源，它维护着内存空间资源的状态：哪些内存空间资源是空闲的，哪些内存空间资源是已分配的。如果应用程序访问相应的设备，这时操作系统内核中的驱动模型组件，就要根据设备的状态进行相应的控制（因为它维护着设备的相关状态）：哪些设备是可以访问的，哪些设备是不能访问的。所幸的是，前面我们已经做好了内存管理和驱动模型组件。

2) 多道应用程序之间的切换，这个问题后面章节会仔细探讨。

3) 在操作系统内核中，如何唯一标识一个应用程序？针对这个问题，可以设计一个数据结构，其中保存应用程序的相关信息，每个应用程序对应一个这样的数据结构的实例变量。

4) 记录一个应用程序使用了哪些资源，例如，占用了多少内存空间，访问了哪些设备，读写了哪些文件。这个问题可以这样解决：我们可以定义描述这些资源的一套数据结构，并把这些数据结构组织起来，放在标识应用程序的数据结构中，因为这些都是与应用程序相关的。

5) 记录一个应用程序运行到哪里了，这个问题可以通过保存 CPU 的寄存器而做到。

上面只是列出了比较核心的问题及解决手段，这也是为了简单，方便了解原理。上面的第 2、3、4、5 这 4 个问题也是本章节要用代码解决的。

一个应用程序运行后在操作系统内核中的标识、使用的所有资源、运行到哪里了，这些概念综合起来，前辈给了一个响亮的名字——进程。这也是实现多道程序模型的核心。作为操作系统内核的开发者，我们认为进程或许是应用程序运行中的标识、容纳应用程序资源的容器，也或许是一堆令大脑发麻的数据结构。

从单道应用程序使用的资源开始，发现资源并不是在任何时刻下都可用，正因如此，才提出了多道程序模型，最终引出了进程。明白了进程是怎么回事，接着就得想办法实现它了。

10.2 相关的数据结构

进程是什么？我们在大多数情况下得到的答案是：进程是对某一特定程序运行过程的抽象。这个答案确实够抽象的。前面已经介绍了进程的由来，从操作系统内核实现者的角度来看：进程不过就是一套数据结构。

把一些概念、模型给数据结构化，这一直都是我们在做并且擅长的。下面将设计两个数据结构：进程和调度进程表。进程数据结构用于标识操作系统中正在运行的应用程序，以及该应用程序使用资源的集合、机器上下文等。而调度进程表就是把众多进程按照一定的规则组织起来。

10.2.1 设计进程的数据结构

通过前面内容已经感觉到了，从操作系统内核实现者的角度来看：进程不过就是一套数据结构。现在就来设计关于进程的数据结构。

从前面应用程序的运行，我们知道，一个应用程序的运行需要一些必要的资源，如内存、CPU、其他设备等。但是进程这个数据结构除了要统计应用程序所使用的资源外，还有一些操作系统内核用于进程切换的信息和应用程序本身的相关信息，列举如下：

- 1) 进程的 ID、标志位，用于标识一个应用程序及其属性。
- 2) 进程的状态，如新建、运行、睡眠等。
- 3) 进程的 CPUID，表示该进程运行在哪个 CPU 上，这是用于多 CPU 的计算平台的。
- 4) 进程的 TICK，表示进程的时间片，即进程运行多长时间就必须切换。
- 5) 进程的优先级，表示该进程现在是否要优先于其他进程运行。
- 6) 进程的运行模式，如进程在当前时刻下，是运行在内核模式还是用户模式下。
- 7) 进程的用户栈和内核栈，进程运行在用户模式下时，就使用用户栈，如果响应中断或者执行系统调用切换到内核模式下，就使用内核栈。
- 8) 进程 CPU 上下文，保存进程的运行进度，进程切换时必须保存进程执行到哪里了。
- 9) 进程使用的资源表，记录进程使用了哪些资源。

好了，马上就要写代码了，不过要先建立相关的源代码文件，先在“xxxx/lmosem/kernel/”目录下建立一个 krlthread.c 文件，然后在“xxxx/lmosem/include/knlinc/”目录下建立 krlthread_t.h、krlthread.h 两个头文件。按照前面章节中的方法组织好这些文件，并把 krlt-hread.o 加入到 LMOSEM 内核编译头文件（buildfile.h）中的 BUILD_KRNL_OBJS 宏的后面。

相关的文件已经建立好了，下面来设计好进程的数据结构，如代码清单 10-1[xxxx/lmosem/include/knlinc/krlthread_t.h] 所示。

代码清单 10-1 进程的数据结构及相关宏

```
//进程状态
#define TDSTUS_RUN 0
#define TDSTUS_SLEEP 3
#define TDSTUS_WAIT 4
#define TDSTUS_NEW 5
#define TDSTUS_ZOMB 6
//进程TICK
#define TDRUN_TICK 20
//进程最大能使用的资源数
#define TD_HAND_MAX 8
//保存CPU上下文的数据结构
typedef struct s_CONTEXT
{
```

```

    reg_t      ctx_usrsp;      //用户模式下的栈指针寄存器
    reg_t      ctx_svcs;      //内核模式下的栈指针寄存器
    reg_t      ctx_svcspsr;    //内核模式下的备份程序状态寄存器
    reg_t      ctx_cpsr;      //当前程序状态寄存器
    reg_t      ctx_lr;        //内核模式下的链接寄存器
}context_t;
//进程数据结构
typedef struct s_THREAD
{
    spinlock_t  td_lock;      //保护其自身的自旋锁
    list_h_t    td_list;      //链表
    uint_t      td_flg;      //进程标志
    uint_t      td_stus;      //进程状态
    uint_t      td_cpuid;     //进程CPUID
    uint_t      td_id;        //进程自身ID
    uint_t      td_tick;      //进程TICK
    uint_t      td_privilege;
    uint_t      td_priority;   //进程优先级
    uint_t      td_runmode;    //进程运行模式
    adr_t       td_krlstktp;
    adr_t       td_krlstkstart; //进程内核栈的开始地址与栈顶
    adr_t       td_usrstktp;
    adr_t       td_usrstkstart; //进程用户栈的开始地址与栈顶
    void*       td_mmdsc;      //进程内存描述符,以后扩展所用
    void*       td_resdsc;
    void*       td_privtep;
    void*       td_extdatap;    //以后扩展所用
    context_t    td_context;    //进程的CPU上下文
    objnode_t*   td_handtbl[TD_HAND_MAX]; //进程使用的资源数组
}thread_t;

```

我们实现的这个 LMOSEM 内核并不严格区分进程和线程,所以暂时就称为 `thread_t`。LMOSEM 内核每加载执行一个应用程序,就会创建一个 `thread_t` 数据结构的实例变量。因为 `thread_t` 结构表示一个应用程序的存在、执行进度、使用的资源。

一个简单的 `thread_t` 数据结构就设计好了。`thread_t` 数据结构是 LMOSEM 内核动态创建的。理论上只要内存空间和 CPU 寻址能力允许,就可以创建无限多个进程。但是怎么高效地查找这些进程呢?下面我们就去设计一个调度进程表,它就是用来组织众多进程的。

10.2.2 调度进程表

LMOSEM 内核每加载运行一个应用程序就会创建一个进程,而每创建一个进程就会创建一个 `thread_t` 结构的实例变量。那么,对于 LMOSEM 内核来说,查找一个应用程序就是查找其对应的 `thread_t` 结构。可是怎么高效地查找 `thread_t` 结构呢?下面就去解决这个问题。

查找 `thread_t` 结构, 先不考虑怎么高效, 先要考虑从哪里去查找, 所以先要建立一个容纳所有 `thread_t` 结构的数据结构。这个非常简单, 定义一个链表头, 把 `thread_t` 结构挂载到这个链表头上就解决了, 如代码清单 10-2[xxxx/lmosem/include/knlinc/krlsched_t.h] 所示。

代码清单10-2 进程链表头

```
typedef struct s_THRDLST
{
    list_h_t    tdl_lsth;           //链表头
    thread_t*   tdl_currntd;        //该链表头中正在运行的进程的指针
    uint_t      tdl_nr;             //计数器变量
}thrdlst_t;
```

可是把所有 `thread_t` 结构都放在这一个 `thrdlst_t` 结构中查找, 速度必然变慢。前面我们设计的 `thread_t` 结构中有一个表示优先级的域, 优先级实际上是一个整数, 数值越小优先级越高, 其最低优先级为 64。想想, 看能不能让每个优先级对应一个 `thrdlst_t` 结构, 最后形成一个 `thrdlst_t` 结构数组, 每个数组元素用于挂载不同优先级的 `thread_t` 结构。这当然是可行的, 如代码清单 10-3[xxxx/lmosem/include/knlinc/krlsched_t.h] 所示。

代码清单10-3 进程调度数据结构

```
typedef struct s_SCHDATA
{
    spinlock_t   sda_lock;           //自旋锁
    uint_t        sda_cpuid;          //CPU ID
    uint_t        sda_schedflgs;      //调度标志
    uint_t        sda_preemptidx;     //LMOSEM内核可抢占标志
    uint_t        sda_threadnr;       //进程数
    uint_t        sda_prityidx;       //当前动态优先级
    thread_t*     sda_cpuidle;        //当前CPU上的空闲进程
    thread_t*     sda_currtd;         //当前CPU上正在运行的进程
    thrdlst_t     sda_thdlst[PRITY_MAX]; //thrdlst_t结构数组
}schdata_t;
```

实际上, 查找进程通常是进程调度器的事, 所以这里将其称为 `schdata_t` 数据结构, `PRITY_MAX` 宏就是 64。我们接着思考, 如果计算平台中有多颗 CPU, 为了处理多颗 CPU 之间的同步问题, 最好每颗 CPU 对应一个 `schdata_t` 结构, 这样又形成了一个 `schdata_t` 结构数组, 如代码清单 10-4[xxxx/lmosem/include/knlinc/krlsched_t.h] 所示。

代码清单10-4 进程调度数据结构容器

```
typedef struct s_SCHEDCALSS
{
    spinlock_t    scl_s_lock;         //自旋锁
    uint_t         scl_s_cpunr;        //CPU数量, 在目前情况下只有一颗CPU
    uint_t         scl_s_threadnr;     //进程总数量
```

```

uint_t      scls_threadid_inc;          //保留以后使用
schdata_t   scls_schda[CPUCORE_MAX]; //schdata_t结构数组, CPUCORE_MAX为1
}schedclass_t;

```

schedclass_t 数据结构自然是一个全局的数据结构, 它的实例变量被定义在 krlglobal.c 文件中, 如代码清单 10-5[xxxx/lmosem/kernel/krlglobal.c] 所示。

代码清单10-5 定义全局的schedclass_t结构类型变量osschedcls

```

.....
KRL_DEFGLOB_VARIABLE(schedclass_t, osschedcls);
.....

```

综合前面的知识, 我们设计了一些数据结构。下面来看一看这些数据结构在 LMOSEM 内核运行后的组织形式, 如图 10-5 所示。

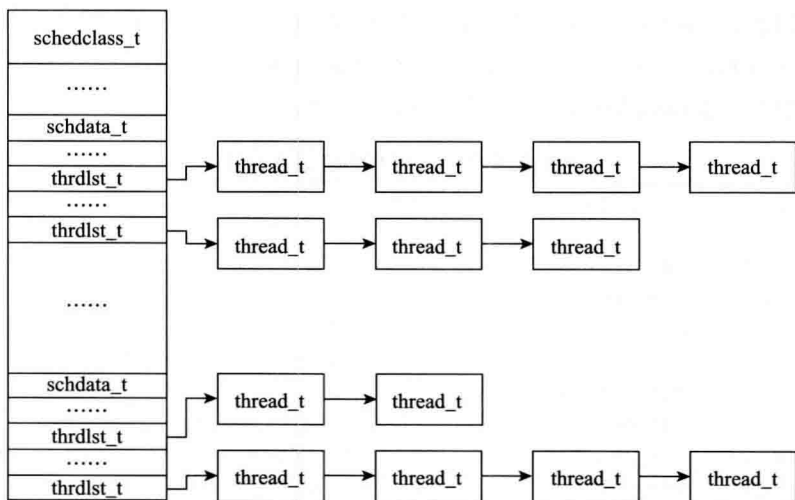


图 10-5 LMOSEM 运行时刻进程的组织结构

前面设计了 thread_t 数据结构, 这里为了组织 thread_t 数据结构又设计了 schedclass_t 数据结构, 让 thread_t 数据结构有了安放之处。或许对这些数据结构还有一些不懂, 不用怕, 后面用到时就明白了, 但是关于进程的数据结构就到这里了。

10.3 LMOSEM 内核的第一个进程

关于 LMOSEM 内核, 直到这里, 还没有执行过任何进程, 所以本节就要建立 LMOSEM 内核的第一个进程, 并且让它运转起来。当然, 在此之前一定要先初始化进程管理组件, 因为之前设计用于管理进程的数据结构, 一定要先初始化后才能使用。下面就去干好这几件事。

10.3.1 进程管理组件的初始化

LMOSEM 内核中的所有组件在开始工作之前，都必须先初始化，进程管理组件也不例外。这里就开始做进程管理组件的初始化工作，实际上就是初始化之前设计的那些数据结构。这并不陌生，前面其他初始化工作，已经做过很多次了。

与进程打交道最密切的是进程调度器，进程管理组件的初始化代码理所当然地放在进程调度器源代码模块中。进程调度器模块的源代码文件还没有建立呢，所以先建立 `krlsched.c` 文件和相关的头文件，并按照前面的规定组织好它们。

这里为了使逻辑清晰，依然采用每个数据结构对应一个初始化函数。在 `krlsched.c` 文件中编写相应的代码，如代码清单 10-6[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-6 进程调度相关数据结构的初始化

```
//初始化一个thrdlst_t结构
void thrdlst_t_init(thrdlst_t* initp)
{
    //初始化链表头
    list_init(&initp->tcl_lsth);
    //初始化时应该为0
    initp->tcl_currntd=NULL;
    initp->tcl_nr=0;
    return;
}
//初始化schdata_t结构
void schdata_t_init(schdata_t* initp)
{
    hal_spinlock_init(&initp->sda_lock);
    //目前LMOSEM内核中这个函数永远返回0
    initp->sda_cpuid=hal_retn_cpuid();
    //初始化时不需要调度
    initp->sda_schedflgs=NOTS_SCHED_FLGS;
    //初始化时都为0
    initp->sda_preemptidx=0;
    initp->sda_threadnr=0;
    initp->sda_prityidx=0;
    initp->sda_cpuidle=NULL;
    initp->sda_currtd=NULL;
    //初始化sda_thrdlst数组
    for(uint_t ti=0;ti<PRITY_MAX;ti++)
    {
        thrdlst_t_init(&initp->sda_thrdlst[ti]);
    }
    return;
}
//初始化schedclass_t结构
```



```

void schedclass_t_init(schedclass_t* initp)
{
    hal_spinlock_init(&initp->scls_lock);
    initp->scls_cpunr=CPUCORE_MAX;
    //初始化时都为0
    initp->scls_threadnr=0;
    initp->scls_threadid_inc=0;
    //初始化CPUCORE_MAX个schdata_t结构
    for(uint_t si=0;si<CPUCORE_MAX;si++)
    {
        schdata_t_init(&initp->scls_schda[si]);
    }
    return;
}
//进程管理组件初始化
void init_krlsched()
{
    schedclass_t_init(&osschedcls);
    return;
}

```

在代码清单 10-6 中，一共写了 4 个函数，这些函数最终初始化了进程相关的数据结构。这几个函数非常简单，看其中的注释就能明白。但是要记住，把 `init_krlsched` 函数加入到 `init_krl` 函数中，如代码清单 10-7[xxxx/lmosem/kernel/krlinit.c] 所示。这样，`init_krlsched` 函数就能被 LMOSEM 内核功能层的主初始化函数调用了。

代码清单10-7 LMOSEM内核功能层的初始化函数

```

void init_krl()
{
    init_krlmm();
    init_krldevice();
    init_krldriver();
    init_krlsched();
    .....
    hal_sysdie("LMOSEM RUN!!");
    return;
}

```

到这里，进程管理组件的数据结构就初始化好了，代码非常简单，下面就该去建立 LMOSEM 内核的第一个进程了。

10.3.2 建立空转进程

操作系统的第一个进程是什么，不同的操作系统各不相同，有的是 INIT 进程，有的是 SHELL 进程。对于我们的 LMOSEM 系统来说，是空转进程。

空转进程是什么呢？当 LMOSEM 系统中没有任何进程或所有进程都是等待状态时，必须有一个进程可以运行，这个进程就是空转进程，因为不能让 CPU 无处可去。那么空转进程是干什么的呢？在我们的 LMOSEM 系统中它只干一件事，那就是触发进程调度器重新调度进程，如果进程调度器发现没有可运行的进程，它又会调度空转进程，这就形成了一个循环。

好了，现在就来创建空转进程。创建空转进程与创建普通进程一样，首先要明白创建一个进程需要什么，如下：

1) 构成进程所必要的，如内核栈、用户栈、进程描述符 (thread_t 结构) 等。

2) 一个主函数，相当于我们写应用程序时的 main 函数，因为进程要有执行的代码，它就是从这个函数开始执行的。

这里先建立一个 krlcpuidle.c 源代码文件，在其中写上两个函数，如代码清单 10-8[xxxx/lmosem/kernel/krlcpuidle.c] 所示。

代码清单 10-8 创建空转进程

```

Void new_cpuidle()
{
    //调用new_cpuidle_thread函数创建空转进程
    thread_t* thp=new_cpuidle_thread();
    //返回NULL表示创建失败，死机
    if(thp==NULL)
    {
        hal_sysdie("newcpuidle err");
    }
    //输出空转进程描述符地址表示创建成功
    printfk("CPUIDLETASK: %x\n\r", (uint_t)thp);
    return;
}
//这是空转进程的主函数，但是它不是运行在用户态下的，而是运行在内核态下的，所以它可以直接调用进程
//调度器函数，但与别的LMOSEM内核函数不同的是，它有单独的thread_t结构和单独的内核栈
void krlcpuidle_main()
{
    //这个进程不会退出，所以是死循环
    for(;;)
    {
        printfk("cpuidle is run!!\n\r");
        //这是我们后面要实现的进程调度函数，这里先把它注释起来
        //krlschedul();
    }
    return;
}

```

代码清单 10-8 中的 new_cpuidle 函数，最终会调用 new_cpuidle_thread 函数创建空转进

程。下面就来写好 `new_cpuidle_thread` 函数，如代码清单 10-9[xxxx/lmosem/kernel/krlcpuidle.c] 所示。

代码清单10-9 创建空转进程的核心函数

```

thread_t* new_cpuidle_thread()
{
    thread_t* ret_td=NULL;
    bool_t acs=FALSE;
    size_t usrstksz=DAFT_TDUSRSTKSZ, krlstksz=DAFT_TDKRLSTKSZ;
    adr_t usrstkadr=NULL, krlstkadr=NULL;
    //分配用户栈
    usrstkadr=krlnew(usrstksz);
    //分配用户栈失败
    if(usrstkadr==NULL)
    {
        return NULL;
    }
    //分配内核栈
    krlstkadr=krlnew(krlstksz);
    //分配内核栈失败后，同时也要释放用户栈
    if(krlstkadr==NULL)
    {
        if(krldelete(usrstkadr,usrstksz)==FALSE)
        {
            return NULL;
        }
        return NULL;
    }
}
//分配进程描述符的函数，后面我们会看到的
ret_td=krlnew_thread_dsc();
if(ret_td==NULL)
{
    //释放内核栈和用户栈
    acs=krldelete(usrstkadr,usrstksz);
    acs=krldelete(krlstkadr,krlstksz);
    //acs为FALSE就说明内存空间释放出了问题，实际上在这里就应该死机
    if(ac==FALSE)
    {
        return NULL;
    }
    return NULL;
}
//用户特权，实际上我们没有实现进程权限检查
ret_td->td_privilege=PRILG_USR;
//最低优先级
ret_td->td_priority=PRITY_MAX;

```

```

//内核栈顶与内核栈的开始地址
ret_td->td_krlstktp=krlstkadr+(adr_t)(krlstksz-1);
ret_td->td_krlstkstart=krlstkadr;
//用户栈顶与用户栈的开始地址
ret_td->td_usrstktp=usrstkadr+(adr_t)(usrstksz-1);
ret_td->td_usrstkstart=usrstkadr;
krlthreadkrlsatck_init(ret_td, (void*)krlcpuidle_main, DAFT_CPSR, DAFT_CIDLESPSR);
//当前情况下, 该函数始终返回0
uint_t cpuid=hal_retn_cpuid();
//以cpuid为下标获取schdata_t结构指针
schdata_t* schdap=&osschedcls.scls_schda[cpuid];
//把空转进程描述符的地址放进schdata_t结构的sda_cpuidle域中
schdap->sda_cpuidle=ret_td;
//标识当前正在运行的进程是空转进程
schdap->sda_currtd=ret_td;
//返回空转进程描述符的指针
return ret_td;
}

```

代码清单 10-9 中的注释已经非常明确了, 但是有一个重要的 `krlthreadkrlsatck_init` 函数没有讲到, 它到底干了什么神奇的事, 后面再仔细研究。

好了, 到这里我们的空转进程就创建好了, 不过并没有让它运行。下面我们就让它运行起来。

10.3.3 空转进程运行

每个进程都必须要有有一套独立的 CPU 上下文, 也就是说要有一些全新的数据值。这些数据值对应 CPU 中的每个寄存器, 在进程要运行时, 就把这些数据值装入 CPU 的寄存器中。

如上所述, 进程要开始运行必须为 CPU 中所有的寄存器装入全新的值。在我们所使用的计算平台上, 什么方法可以做到呢? 还记得前面 CPU 指令集中的 `ldmfd`、`ldmai` 指令吗, 它们可以做到。在解决这个问题之前, 先来看一看 10.3.2 节中遗留的问题——`krlthreadkrlsatck_init` 函数。该函数是初始化进程内核栈的, 不过初始化进程的内核栈有一定的要求。我们还是来看代码吧, 如代码清单 10-10[xxxx/lmosem/kernel/krlthread.c] 所示。

代码清单10-10 初始化进程的内核栈

```

void krlthreadkrlsatck_init(thread_t* thdp, void* runadr, reg_t cpsr, reg_t spsr)
{
    //处理栈对齐
    thdp->td_krlstktp&=(~0xf);
    thdp->td_usrstktp&=(~0xf);
    //从栈顶减去一个armregs_t结构的大小, 并让armregs_t结构的指针指向这个地址, 相当于在进程内核栈
    //中让出一个armregs_t结构的空。下面就在这个空间内初始化CPU的r0~r14这15个寄存器, 高地址栈单
    //元放序号大的寄存器的值, r0~r12寄存器对应的值为0, r13寄存器对应的值为进程用户栈指针, r14寄存

```

```

//器对应的值为进程主函数的地址
armregs_t* arp=(armregs_t*)(thdp->td_krlstktop-sizeof(armregs_t));
arp->r0=0;
arp->r1=0;
arp->r2=0;
arp->r3=0;
arp->r4=0;
arp->r5=0;
arp->r6=0;
arp->r7=0;
arp->r8=0;
arp->r9=0;
arp->r10=0;
arp->r11=0;
arp->r12=0;
arp->r13=(reg_t)(thdp->td_usrstktop);
arp->r14=(reg_t)runadr;
//初始化进程的context_t结构
thdp->td_context.ctx_usrsp=(reg_t)(thdp->td_usrstktop);
//让context结构中的ctx_svcsp域等于arp地址
thdp->td_context.ctx_svcsp=(reg_t)(arp);
//让context结构中的ctx_svcpsr域等于spsr参数
thdp->td_context.ctx_svcpsr=spsr;
//让context结构中的ctx_cpsr域等于cpsr参数
thdp->td_context.ctx_cpsr=cpsr;
//让context结构中的ctx_lr域等于runadr参数
thdp->td_context.ctx_lr=(reg_t)runadr;
return;
}

```

代码清单 10-10 中的 `armregs_t` 数据结构在 `cpu_t.h` 文件中，如代码清单 10-11[xxxx/lmosem/include/halinc/cpu_t.h] 所示。

代码清单 10-11 `armregs_t` 数据结构

```

typedef struct s_ARMREGS
{
    reg_t r0;
    reg_t r1;
    reg_t r2;
    reg_t r3;
    reg_t r4;
    reg_t r5;
    reg_t r6;
    reg_t r7;
    reg_t r8;
    reg_t r9;

```

```

reg_t r10;
reg_t r11;
reg_t r12;
reg_t r13;
reg_t r14;
}armregs_t;

```

armregs_t 数据结构非常简单，不必多说。如果上面两段代码还是不能让你明白 krlthreadkrlsatck_init 函数到底干了什么，再看看下面画出的内核栈的逻辑视图，就明白了，如图 10-6 所示。

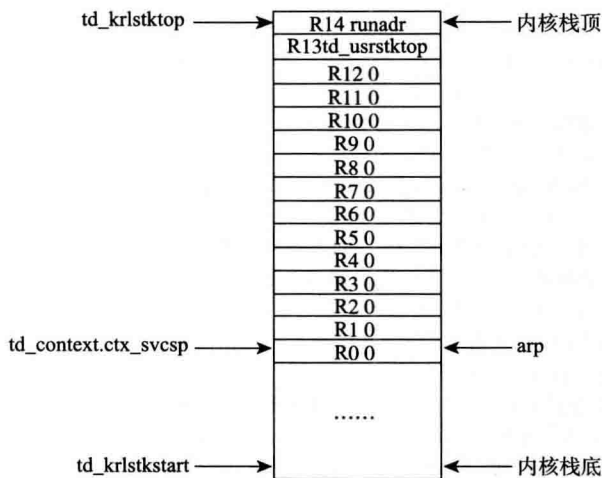


图 10-6 内核栈的逻辑视图

处理好了进程内核栈和进程 context_t 结构中的内容，下面就是“打响发令枪的时刻”——使空转进程开始运行了。这个“发令枪”由两个函数构成：krlcpuidle_start_retnfrom_first_sched，先来写好第一个函数，如代码清单 10-12[xxxx/lmosem/kernel/krlcpuidle.c] 所示。

代码清单10-12 运行空转进程

```

void krlcpuidle_start()
{
    //获取当前CPU的ID号，目前它会返回0
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //获取当前CPU的空转进程的thread_t结构指针
    thread_t* tdp=schdap->sda_cpuidle;
    //设置这个进程的状态为运行态
    tdp->td_stus=TDSTUS_RUN;
    //开始运行这个进程
}

```

```

    retnfrom_first_sched(tdp);
//事实上这个函数不会返回
    return;
}

```

在代码清单 10-12 中，最重要的就是 `retnfrom_first_sched` 函数，它才是真正的发令枪。下面就去实现它，如代码清单 10-13[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-13 装载进程的CPU上下文

```

void retnfrom_first_sched(thread_t* thrdp)
{
    __asm__ __volatile__ (
        //相当于把thrdp->td_context.ctx_svcpsr的值送到SPSR寄存器中
        "msr spsr,%[svcpsr]\n\t"
        //相当于把thrdp->td_context.ctx_svcsp的值送到SP寄存器中
        "mov sp,%[svcsp] \n\t"
        //相当于把thrdp->td_context.ctx_lr的值送到LR寄存器中
        "mov lr,%[usrlr] \n\t"
        //以上SPSR、SP、LR寄存器都是CPU特权管理模式下的寄存器
        //把SP寄存器指向的内存块中的15个字加载到CPU用户模式下的r0~r14寄存器中
        "ldmia sp,{r0-lr}^ \n\t"
        //把SP寄存器的值加上60，平衡内核栈，见图10-6
        "add sp,sp,#60 \n\t"
        //把LR寄存器的值送给PC寄存器，这样就开始运行krlcpuidle_main函数了，因为mov指令后
        //面有s，所以在更新PC寄存器的同时还会把当前模式下的SPSR寄存器的值复制到CPSR寄存器
        //中，改变了CPSR寄存器的值就会改变CPU的工作模式
        "movs pc,lr \n\t"
        :
        :[svcsp]"r"(thrdp->td_context.ctx_svcsp),[svcpsr]"r"(thrdp->td_context.ctx_svcpsr),
        [usrlr]"r"(thrdp->td_context.ctx_lr)
        : "cc","memory"
    );
}

```

`retnfrom_first_sched` 函数是在 C 语言代码中嵌入汇编代码实现的，那是因为我们用到一些特殊指令，还要操作 CPU 的寄存器，而 C 语言代码做不到这些。代码逻辑不算难：第一部分，就是把进程的 `context_t` 结构中的一些内容，复制到相应的寄存器中；第二部分，就是利用 `ldmia` 指令把先前在内核栈中初始化好的数据，加载到 CPU 用户模式下的 `r0 ~ r14` 寄存器中；第三部分，就是平衡内核栈后，用 `movs` 指令同时改变了 PC 和 CPSR 寄存器中的值。要注意的是，上面 `thrdp->td_context.ctx_svcpsr` 中的值并没有改变 CPU 模式（请参阅前面介绍的 CPSR 寄存器），CPU 还是特权管理模式，因此空转进程是运行在 CPU 特权管理模式下的一个内核态进程。

写好了上面这些函数，索性还写一个函数把 `new_cpuidle`、`krlcpuidle_start` 函数封装一下，

如代码清单 10-14[xxxx/lmosem/kernel/krlcpuidle.c] 所示。

代码清单10-14 初始化空转进程

```
void init_krlcpuidle()
{
    new_cpuidle();
    krlcpuidle_start();
    return;
}
```

上面这些代码写得对吗？编译一下 LMOSEM 内核，连接上开发板试试就知道了。不过在测试之前，还要把 init_krlcpuidle 函数加入到 LMOSEM 内核功能层的主初始化函数 init_krl 中，如代码清单 10-15[xxxx/lmosem/kernel/krlinit.c] 所示。

代码清单10-15 LMOSEM内核功能层的初始化函数

```
void init_krl()
{
    init_krlmm();
    init_krldevice();
    init_krldriver();
    init_krlsched();
    init_ktime();
    init_krlcpuidle();
    hal_sysdie("LMOSEM RUN!!");
    return;
}
```

好了，make 一下，连接上开发板测试一下吧。在正常情况下应该会出现如图 10-7 所示的情况。



图 10-7 空转进程的运行

从终端中我们应该看到，不断交替出现的 `systick_handle run !!` 和 `cpuidle is run !!` 这两个字符串，这说明我们的 `systick` 驱动程序和空转进程都成功运行了。

好了，到这里，可是一个不小的进步。从此以后，我们辛苦编写的操作系统终于可以执行进程了。但是这仅仅是第一个进程开始运行了，我们的操作系统还要支持多个进程并发运行呢。当然这里还有很多疑惑，例如，为什么要有内核栈和用户栈等，这些内容我们后面会详细介绍的，这里只是为了认识一个进程的创建和运行过程，所以很简单就过去了。为了解开这些疑惑，下面我们继续。

10.4 新建进程

现代操作系统一般都会提供新建进程的功能，前面通过空转进程了解进程的创建和运行过程，但那毕竟不是正式地建立一个常规进程。

下面将从分配进程描述符开始，到分配进程的内存空间，最后把新进程加入进程调度表中，详细介绍了 LMOSEM 内核新建进程的过程。

10.4.1 分配进程描述符

LMOSEM 内核感知一个进程的存在，就是通过进程描述符，即 `thread_t` 数据结构实现的。这个数据结构包含了一个应用程序运行时刻的绝大部分信息。创建一个进程的第一步就是分配一个进程描述符。

LMOSEM 内核的大部分数据结构的实例变量都是动态分配的，需要时才分配，不需要了就删除，`thread_t` 结构的实例变量也是一样。下面就去写好分配 `thread_t` 结构的实例变量的代码，如代码清单 10-16[xxxx/lmosem/kernel/krlthread.c] 所示。

代码清单10-16 分配进程描述符并初始化

```
//初始化context_t结构
void context_t_init(context_t* initp)
{
    initp->ctx_usrsp=0;
    initp->ctx_svcsp=0;
    initp->ctx_svcpsr=0;
    initp->ctx_cpsr=0;
    initp->ctx_lr=0;
    return;
}
uint_t krlretn_thread_id(thread_t* tdp)
{
    //返回进程ID，实际上就是thread_t结构的地址，它具有唯一性
    return (uint_t)tdp;
}
```

```

//初始化thread_t结构
void thread_t_init(thread_t* i nntp)
{
    hal_spinlock_init(&initp->td_lock);
    list_init(&initp->td_list);
    initp->td_flg=0;
    initp->td_stus=TDSTUS_NEW;
    initp->td_cpuid=hal_retn_cpuid();
    initp->td_id=krlretn_thread_id(initp);
    initp->td_tick=0;
//普通的用户进程
    initp->td_privilege=PRILG_USR;
//最高优先级
    initp->td_priority=PRITY_MIN;
    initp->td_runmode=0;
    initp->td_krlstktop=NULL;
    initp->td_krlstkstart=NULL;
    initp->td_usrstktop=NULL;
    initp->td_usrstkstart=NULL;
    initp->td_mmdsc=NULL;
    initp->td_resdsc=NULL;
    initp->td_privtep=NULL;
    initp->td_extdatap=NULL;
//初始化context_t结构
    context_t_init(&initp->td_context);
    for(uint_t hand=0;hand<TD_HAND_MAX;hand++)
    {
        initp->td_handtbl[hand]=NULL;
    }
    return;
}
//分配进程描述符
thread_t* krlnew_thread_dsc()
{
//分配thread_t结构的实例变量的内存空间
    thread_t* rettdp=(thread_t*)(krlnew((size_t)(sizeof(thread_t))));
//分配失败则返回NULL
    if(rettdp==NULL)
    {
        return NULL;
    }
//初始化thread_t结构的实例变量
    thread_t_init(rettdp);
    return rettdp;
}

```

代码清单 10-16 实在太简单了，前面也不止一次写过类似的代码。总之就是分配一个数据结构实例变量的内存空间，接着初始化这块内存空间。

到这里，创建进程的第一步，就这样轻松地完成了。下面我们去完成第二步，分配进程的内存空间。

10.4.2 分配内存空间

一个进程至少需要一定的内存空间来存储应用程序的代码和数据，同时需要两个栈空间来完成应用程序中的函数调用，这是一个应用程序得以运行的基础。所以，创建进程的一个重要步骤，就是分配一定的内存空间。

在写代码之前先要解决几个问题，我们的 LMOSEM 内核没有实现虚拟内存机制，多个应用程序的代码和数据，无法从一个固定的虚拟地址开始编译链接。在通常情况下，我们看到的 Linux 应用程序和 Windows 应用程序都是从一个固定的地址开始编译链接的，那个固定的地址，实际上就是虚拟地址，而应用程序的代码和数据，在运行时都是装载到不同的物理地址空间中，最后让虚拟地址映射到相应的物理地址上就行了。而我们只能让操作系统内核和应用程序链接在一起，运行在同一个物理地址空间中。事实上，很多小型的嵌入式操作系统就是这么干的。

可是一个进程为什么要有内核栈和用户栈呢？这得从进程中的代码执行开始说起。通常，进程中运行的代码是用户编写的代码，但是如果进程中的用户代码在运行中发生了中断或者执行了软中断指令，都会导致操作系统内核代码的运行，或者说就是中断导致这个进程进入操作系统内核的，但这个进程并没有停止或者切换，所以还是这个进程，但是执行的代码却不同了。从这个角度看，一个进程有两个部分，操作系统内核部分和用户编写的部分，只是操作系统内核的部分对所有进程都是共享的，而用户编写的部分各不相同且各自分开。因此，对于一个进程的地址空间中存放的有两个部分，如图 10-8 所示。

再来看看内核栈和用户栈。栈的作用不仅仅是保存临时变量和传递参数，还有一个重要的作用，就是保存函数的调用路径。什么是函数的调用路径？例如，func0 中调用了 func1，func1 中又调用了 func2，func2 中再调用了 func3 等，从 func0 到 func3 的调用过程就是函数的调用路径。当然最终 func3 能通过栈中的返回信息，层层返回到 func0 中。每个进程进入操作系统内核的方式不同，有的可能是因为中断，有的可能是因为执行软中断指令，这些都会导致进程在操作系统内核中的函数调用路径不同。必须保存完整的函数调用路径，才能在进程切换时保存一个进程运行到哪里了，所以每个进程需要一个内核栈。由于每个进程的用户代码都不同，为了和其他进程隔离，也为了进程中的用户代码和操作系统内核代码隔离，当然最好的做法是每个进程一个用户栈，因此形成了每个进程都有内核栈和用户栈。

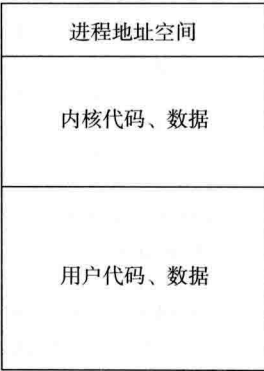


图 10-8 进程的地址空间

明白了原理,下面就编写代码分配上述要求的内存空间。对于一个进程的用户代码和数据占用的内存空间已经不需要分配了,因为它们已经和 LMOSEM 内核编译链接在一起了,这样做也是为了简单。剩下的就是分配内核栈和用户栈的内存空间了。下面我们来实现这些代码,如代码清单 10-17[xxxx/lmosem/kernel/krlthread.c] 所示。

代码清单10-17 新建进程的核心函数

```

thread_t* krlnew_thread_core(void*filerun,uint_t flg,uint_t prilg,uint_t prity,size_
t usrstksz,size_t krlstksz)
{
    thread_t* ret_td=NULL;
    bool_t acs=FALSE;
    adr_t usrstkadr=NULL,krlstkadr=NULL;
//先分配用户栈的内存空间
    usrstkadr=krlnew(usrstksz);
//分配失败则立即返回NULL,表示创建进程失败
    if(usrstkadr==NULL)
    {
        return NULL;
    }
//接着分配内核栈的内存空间
    krlstkadr=krlnew(krlstksz);
//如果分配失败,则先释放用户栈的内存空间并同时返回NULL
    if(krlstkadr==NULL)
    {
        if(krldelete(usrstkadr,usrstksz)==FALSE)
        {
            return NULL;
        }
        return NULL;
    }
//分配进程描述符
    ret_td=krlnew_thread_dsc();
//如果分配进程描述符失败,则释放用户栈和内核栈的内存空间,以免内存空间泄漏
    if(ret_td==NULL)
    {
        acs=krldelete(usrstkadr,usrstksz);
        acs=krldelete(krlstkadr,krlstksz);
        if(acs==FALSE)
        {
            return NULL;
        }
        return NULL;
    }
//设置进程的优先级和权限
    ret_td->td_privilege=prilg;

```

```

    ret_td->td_priority=prity;
//设置进程内核栈的开始地址和栈顶
    ret_td->td_krlstktp=krlstkadr+(adr_t)(krlstksz-1);
    ret_td->td_krlstkstart=krlstkadr;
//设置进程用户栈的开始地址和栈顶
    ret_td->td_usrstktp=usrstkadr+(adr_t)(usrstksz-1);
    ret_td->td_usrstkstart=usrstkadr;
//初始化进程的内核栈
    krlthreadkrlsatck_init(ret_td, filerun, DAFT_CPSR, DAFT_SPSR);
//把进程加入进程调度表中
    krlschdclass_add_thread(ret_td);
    return ret_td;
}

```

函数 `krlnew_thread_core` 有 6 个参数，其中 `prilg`、`prity`、`usrstksz` 和 `krlstksz` 分别是进程权限、进程优先级、进程的用户栈和内核栈的空间大小。我们允许自定义这两个栈空间的大小。`filerun` 是一个通用指针，可以是函数地址，也可以是文件名指针，这是为了方便以后改进。`flg` 是一个标志位，以备以后扩展所用。返回值是 `thread_t` 结构的指针。`krlnew_thread_core` 函数在分配栈空间后，把一些重要的信息写入进程描述符中，然后调用 `krlthreadkrlsatck_init` 函数初始化了进程的内核栈，最后一个 `krlschdclass_add_thread` 函数作用是把进程加入到进程调度表中，这个后面再看。

下面就去把新建进程的最后一步做完。

10.4.3 加入进程调度表

前面已经把进程创建好了，有了进程描述符、两个栈空间，并且初始化好了内核栈，可是还没有加入进程调度表呢。只有加入这个表，`LMOSEM` 内核的进程调度器才能找到这个进程。

在 10.4.2 节中，我们看到 `krlnew_thread_core` 函数最后调用了 `krlschdclass_add_thread` 函数，这个函数的作用就是把进程加入进程调度表中。下面来写好这个函数，如代码清单 10-18[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单 10-18 把进程加入进程调度表中

```

void krlschdclass_add_thread(thread_t* thdp)
{
    uint_t cpuid=hal_retn_cpuid();
//获取当前CPU对应的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    cpuflg_t cufg;
//加锁
    hal_spinlock_saveflg_cli(&schdap->sda_lock, &cufg);
//把进程加入对应进程优先级的thdlst_t结构的链表头中

```

```

    list_add(&thdp->td_list,&schdap->sda_thdlst[thdp->td_priority].tdl_lsth);
//增加这个进程优先级链表头的进程计数
    schdap->sda_thdlst[thdp->td_priority].tdl_nr++;
//增加当前CPU对应的schdata_t结构上的进程计数
    schdap->sda_threadnr++;
//解锁
    hal_spinunlock_restflg_sti(&schdap->sda_lock,&cufg);
//加锁
    hal_spinlock_saveflg_cli(&osschedcls.scls_lock,&cufg);
//增加全局进程计数
    osschedcls.scls_threadnr++;
//解锁
    hal_spinunlock_restflg_sti(&osschedcls.scls_lock,&cufg);
    return;
}

```

代码清单 10-18 中的注释已经很清楚了，不必再详述。

结合前面写的几个函数，它们都是为了创建进程。下面把它们归纳成一个创建进程的接口函数，如代码清单 10-19[xxxx/lmosem/kernel/krlthread.c] 所示。

代码清单10-19 创建进程的接口函数

```

thread_t* krlnew_thread(void* filerun,uint_t flg,uint_t prilg,uint_t prity,size_t
usrstksz,size_t krlstksz)
{
    size_t tustksz=0,tkstksz=0;
//首先对一些参数进行检查，若有不符合要求的，立即返回NULL，表示创建进程失败
    if((flg&0x1)!=0||filerun==NULL||usrstksz>DAFT_TDUSRSTKSZ||krlstksz>DAFT_TDKRLSTKSZ)
    {
        return NULL;
    }
//检查进程优先级和进程特权是否合理，如果不合理则返回NULL
    if((prilg!=PRILG_USR&&prilg!=PRILG_SYS)||prity>PRITY_MAX)
    {
        return NULL;
    }
//如果用户栈空间大小小于DAFT_TDUSRSTKSZ（8KB）就让tustksz=DAFT_TDUSRSTKSZ
    if(usrstksz<DAFT_TDUSRSTKSZ)
    {
        tustksz=DAFT_TDUSRSTKSZ;
    }
//如果内核栈空间大小小于DAFT_TDKRLSTKSZ（8KB）就让tkstksz=DAFT_TDKRLSTKSZ
    if(krlstksz<DAFT_TDKRLSTKSZ)
    {
        tkstksz=DAFT_TDKRLSTKSZ;
    }
}

```

```
return krlnew_thread_core(filerun, flg, prilg, prity, tustksz, tkstksz);  
}
```

`krlnew_thread` 函数的参数和 `krlnew_thread_core` 函数的参数相同，只是它只检查其参数的合法性，最后调用 `krlnew_thread_core` 函数来完成实际的工作。

创建进程的所有工作到这里就完成了。还需要一个进程调度器来调度多个进程，下面就去实现它。

10.5 进程调度

进程调度，简单点说就是当一个进程自愿放弃或者被操作系统内核剥夺运行之后，选择下一个进程运行的过程。其实对于我们这种工程实践派来说，最难的不是选择进程的过程，而是选择了之后如何让被选择的进程平滑地开始运行。

尽管进程调度算法很多，很复杂，但还是要从它开始并选择一种简单的算法，接着看一看怎么处理进程的时间片。然后也是最难的，就是怎么选择一个进程并切换。最后实现进程自愿进入等待和被 LMOSEM 内核唤醒的机制。

10.5.1 调度算法

看过操作系统理论书籍的人都知道进程调度算法非常多，例如，先到先服务、最短作业优先、最短剩余时间优先、时间片轮转、优先级等。这些进程调度算法又可分为几类：批处理系统进程调度、交互式系统进程调度、实时系统进程调度。

看到上面这段文字，很多人已经开始发晕了，其实操作系统初创阶段不必理会这么多，这些都是理论家折腾的东西。先自己设计一种简单的进程调度算法是比较好的选择。

在设计进程调度算法之前，先要清楚自己的操作系统类型，因为不同类型的操作系统调度进程的要求是不同的。对于交互式系统，它要求响应时间和均衡性，可以允许调度延时，例如，常用的 Windows 系统，在极高的系统负载下，可能会让你感觉播放视频、音频时产生卡顿现象。这最多会被计算机前的你骂几句，不会带来灾难性后果。对于硬实时系统就不同了，它要求满足截止时间，绝不允许有调度延时，例如，运行在火箭飞行控制计算机内的和核电站反应堆监测计算机内的硬实时操作系统，稍微慢一点点都会产生灾难性的后果，甚至毁掉一座城市。

事有轻重缓急之分。对于计算机也一样，例如，我们一边用音乐播放器放着音乐，一边用文本编辑器写着文章。对于操作系统，音乐播放器进程应该优先于文本编辑器进程被调度运行，这样播放音乐时才不会产生卡顿，进而不至于影响用户体验。但是我们不能永远只调度运行音乐播放器进程而把文本编辑器进程撇在一旁，这叫进程饥饿现象，这是不应该发生的。

综上所述，首先来设计一种简单的进程调度算法，该算法叫做动态优先级进程调度算法。这种进程调度算法用文字描述可能不太形象，下面我们来画一幅图来表示，如图 10-9 所示。

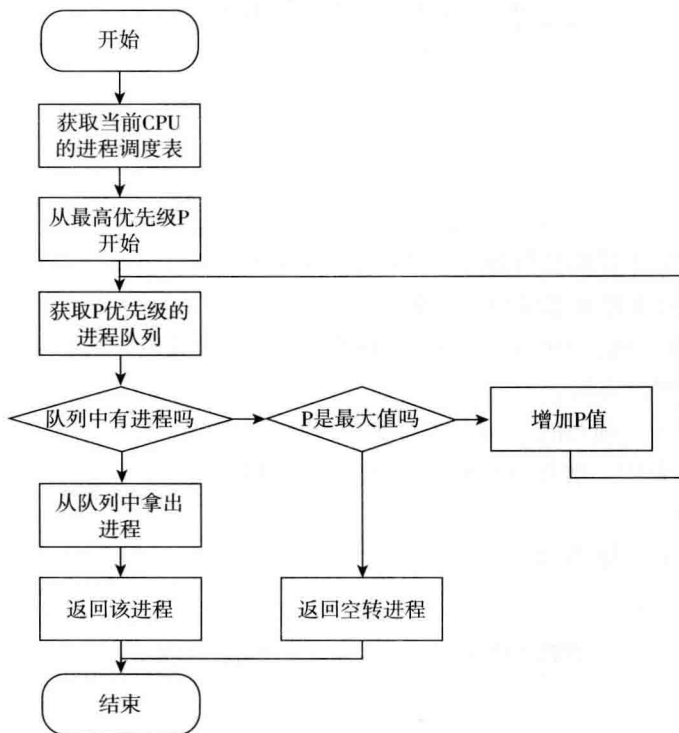


图 10-9 进程调度算法流程图

前面说了，进程调度算法被设计为动态优先级，但是图 10-9 中没有体现出动态优先级，更像一个遍历进程队列数组的算法。是的，为了代码的简单可读，笔者在代码中没有实现动态优先级。但是数据结构的设计是支持动态优先级的，正如后面所看到的那样，我们只要稍稍改动一下代码就能支持动态优先级。

写了一堆文字，对于我们这种工程实践派来说，其实只要看懂图 10-9 就行了。在操作系统初创阶段，一切以简单为上，只要多个进程能被调度运行就行了。下面我们去看一看怎么处理进程 TICK。

10.5.2 处理进程时间片

进程运行了多少个时间片（TICK），实际上就是运行了多长时间，这个时间在操作系统理论书籍中称为进程时间片。在开始运行进程之前，操作系统内核都要为其分配一个时间片。至于这个时间片有多长时间，各大操作系统各不相同，有的是几十毫秒，有的只有几

微秒。

时间片对于一个进程有什么作用，为什么要有时间片，我们先来观察一段应用程序代码，如代码清单 10-20 所示。

代码清单10-20 一段应用程序代码

```
int main()
{
    //死循环
    for(;;);
    return 0;
}
```

代码清单 10-20 不使用任何操作系统服务，但它也不会退出，会一直运行，直到“天荒地老”。由于它不使用操作系统服务，那么除了中断，操作系统内核没有任何手段让其停止并回收其 CPU 的使用权，因此 CPU 不会运行操作系统内核代码，而进程调度也就无从说起，最终它会锁死整个操作系统。

读者应该还记得前面写的 systick 设备驱动程序，它就是一个定时产生中断的设备，我们在它的中断回调函数中，增加当前运行进程的 TICK 值就行了，事实上，该设备正是为了完成此功能而设计的。

下面来写一个增加当前运行进程 TICK 值的函数，如代码清单 10-21[xxxx/lmosem/kernel/krlthread.c] 所示。

代码清单10-21 增加当前运行进程的TICK值

```
void krlthd_inc_tick(thread_t* thdp)
{
    cpuflg_t cpuflg;
    hal_spinlock_saveflg_cli(&thdp->td_lock,&cpuflg);
    //TICK值加1
    thdp->td_tick++;
    //检查进程的TICK值是否大于TDRUN_TICK
    if(thdp->td_tick>TDRUN_TICK)
    {
        //进程的TICK值复位为0
        thdp->td_tick=0;
        //设置进程调度标志，告诉LMOSEM内核要执行进程调度了
        krlsched_set_schedflgs();
    }
    hal_spinunlock_restflg_sti(&thdp->td_lock,&cpuflg);
    return;
}
```

代码清单 10-21 非常简单，就是在加锁状态下增加 thread_t 结构中 td_tick 域的值并做相应的检查。下面顺便写好同样简单的 krlsched_set_schedflgs 函数，如代码清单 10-22[xxxx/

lmosem/kernel/krlsched.c] 所示。

代码清单10-22 设置进程调度标志

```
void krlsched_set_schedflgs()
{
    cpuflg_t cpuflg;
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    hal_spinlock_saveflg_cli(&schdap->sda_lock,&cpuflg);
    //设置需要执行进程调度的标志，事实上NEED_SCHED_FLGS就是一个整数
    schdap->sda_schedflgs=NEED_SCHED_FLGS;
    hal_spinunlock_restflg_sti(&schdap->sda_lock,&cpuflg);
    return;
}
```

前面说过了，增加 TICK 值的一定是当前正在运行的进程，没有运行的进程是不增加 TICK 值的，那么怎么获取当前正在运行的进程呢？熟悉前面数据结构的读者马上就会想到，数据结构中有一个域，是专门保存当前正在运行的进程的，只要读取它就行了。下面还是来写好这个函数吧，如代码清单 10-23[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-23 获取当前正在运行的进程

```
thread_t* krlsched_retn_currthread()
{
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //sda_currtd如果等于NULL那肯定是出严重问题了，所以要死机
    if(schdap->sda_currtd==NULL)
    {
        hal_sysdie("schdap->sda_currtd NULL");
    }
    //返回当前正在运行的进程
    return schdap->sda_currtd;
}
```

最后我们把 krlthd_inc_tick 函数放入 systick 设备驱动程序的中断回调函数中，让它被调用，这套机制就完成了，如代码清单 10-24[xxxx/lmosem/drivers/drvtick.c] 所示。

代码清单10-24 systick设备的中断回调函数

```
drvstus_t systick_handle(uint_t ift_nr,void* devp,void* sframe)
{
    //以当前运行的进程为参数
    krlthd_inc_tick(krlsched_retn_currthread());
    //把这行用于测试的代码注释掉
```

```

    //printfk("systick_handle run!!\n\r");
    return DFCOKSTUS;
}

```

处理进程的时间片就完成了，非常简单。即使是如此简单的机制，如果没有中断管理、驱动模型的支持，也是不行的。这就是为什么我们没有像理论书籍那样一开始就大谈进程管理，因为我们知道没有这些底层机制，进程是玩不转的。下面我们继续。

10.5.3 检查调度状态

从 10.5.2 节中，知道当一个进程的时间片用完之后，`krllthd_inc_tick` 函数中只是调用 `krlsched_set_schedflgs` 函数设置了一个需要调度进程的标志，除此之外没有做任何其他事情，就返回了。

终归是要调度进程的，那么在哪里执行调度操作呢。我们很清楚地知道，无论中断、异常或是系统调用，这些地方都是进程进入 LMOSEM 内核的必经之地。当然，对于一个特定的进程，它从哪里进入就会从哪里返回。因此在进程返回的前一步，检查调度状态并执行调度操作，是个不错的办法。其实这个办法我们是跟 Linux 学的。

在调度进程之前，首先要检查调度状态，看看需不需要执行调度操作。下面来写一个函数完成这件事，如代码清单 10-25[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-25 检查调度状态

```

void krlsched_chkneed_pmptsched()
{
    cpuflg_t cpuflg;
    uint_t schd=0, cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    hal_spinlock_saveflg_cli(&schdap->sda_lock,&cpuflg);
    //查看调度进程标志和LMOSEM内核可抢占标志
    if(schdap->sda_schedflgs==NEED_SCHED_FLGS&&schdap->sda_preemptidx==PMPT_FLGS)
    {
        //清除需要调度进程的标志
        schdap->sda_schedflgs=NOTS_SCHED_FLGS;
        //让schd=1
        schd=1;
    }
    hal_spinunlock_restflg_sti(&schdap->sda_lock,&cpuflg);
    //如果schd等于1就执行进程调度操作，否则不执行任何动作
    if(schd==1)
    {
        krlschedul();
    }
}

```

```

return;
}

```

代码清单 10-25 其实非常简单，就是在加锁状态下，检查当前 CPU 的 `schdata_t` 结构中的 `sda_schedflgs`、`sda_preemptidx` 这两个域，它们分别表示需不需要调度进程和 LMOSEM 内核能不能被抢占。内核可抢占式特性在这里是关闭的，即对 `sda_preemptidx` 域的检查始终是通过的。什么是内核可抢占式特性，这里不介绍，也不会去实现，有兴趣的读者可自行查阅相关资料。`krlsched` 函数就是进程调度器的主函数，后面我们会实现。

前面已经说了，会在中断、异常、系统调用返回的前一步检查进程调度标志。现在，我们就像总装机器一样，把一些零件安装到特定的位置，这里的零件就是 `krlsched_chkneed_pmptsched` 函数，如代码清单 10-26[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单 10-26 IRQ 中断分发函数

```

.....
//IRQ中断分发函数
void hal_irq_distr(void* sframe)
{
    uint_t intoiset=hal_retn_intnr();
    .....
    hal_clear_intpnd(intoiset);
    //检查进程调度标志和LMOSEM内核可抢占标志
    krlsched_chkneed_pmptsched();
    return;
}

```

由于我们只使用 IRQ 中断，所以这里只在 `hal_irq_distr` 函数中加入了 `krlsched_chkneed_pmptsched` 函数。

这样检查进程调度标志的功能就实现了。如果需要进程调度，接着就会执行进程调度操作，下面就去实现进程调度操作。

10.5.4 选择进程

CPU 只有一颗，而进程有多个，为了让所有进程都得到执行，就需要进程调度。事实上，熟悉前面多道程序模型的读者会知道，即使有多个 CPU 和多个进程一一对应，还是会因为一些进程访问资源得不到满足或者进程间同步的问题，而需要进程调度。而进程调度的核心就是选择一个可运行的进程并切换该进程使之运行。

这里先实现选择一个可运行的进程，在实现这个之前，强烈建议读者回顾前面的调度算法和相关的数据结构，只有这样才能真正理解代码的原理。下面来写一个函数，在该函数内部执行选择一个进程的操作，然后如果找到一个可运行的进程就返回该进程，否则就返回该 CPU 上的空转进程，如代码清单 10-27[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-27 选择一个可运行的进程

```

thread_t* krlsched_select_thread()
{
    //实例化一些临时变量
    thread_t* retthd ,*tdtmp ;
    cpuflg_t cufg;
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //加锁
    hal_spinlock_saveflg_cli(&schdap->sda_lock,&cufg);
    //从0级优先级开始扫描,直到63级优先级,PRITY_MAX等于64
    for(uint_t pity=0;pity<PRITY_MAX;pity++)
    {
        //检查当前优先级的进程队列头中有没有进程
        if(schdap->sda_thdlst[pity].tdl_nr>0)
        {
            //进一步检查当前优先级的进程队列头的链表是否为空
            if(list_is_empty_careful(&(schdap->sda_thdlst[pity].tdl_lsth))==FALSE)
            {
                //提取进程队列头链表中的第一个进程并脱链
                tdtmp=list_entry(schdap->sda_thdlst[pity].tdl_lsth.next,thread_t,td_list);
                list_del(&tdtmp->td_list);
            }
            //如果当前进程队列头的tdl_currunttd域中有进程,就将其挂载到进程队列头链表的尾部
            if(schdap->sda_thdlst[pity].tdl_currunttd!=NULL)
            {
                list_add_tail(&(schdap->sda_thdlst[pity].tdl_currunttd->td_list),
                    &schdap->sda_
                    thdlst[pity].tdl_lsth);
            }
            //把tdtmp放入进程队列头的tdl_currunttd域中
            schdap->sda_thdlst[pity].tdl_currunttd=tdtmp;
            //最后将找到的进程放入retthd中以便返回
            retthd=tdtmp;
            //跳转到标号return_step处
            goto return_step;
        }
    }
    //如果进程队列头链表中没有进程,就看看进程队列头的tdl_currunttd域中有没有进程
    if(schdap->sda_thdlst[pity].tdl_currunttd!=NULL)
    {
        //如果有就放入retthd中返回
        retthd=schdap->sda_thdlst[pity].tdl_currunttd;
        //跳转到标号return_step处
        goto return_step;
    }
}

```

```

}
//将sda_prityidx 设置为0级优先级
schdap->sda_prityidx=PRITY_MIN;
//执行到这里还没有找到可运行的进程,所以只有返回空转进程
retthd=krlsched_retn_idlethread();
return_step:
//解锁
hal_spinunlock_restflg_sti(&schdap->sda_lock,&cufg);
//返回retthd中的进程描述符的指针
return retthd;
}

```

在代码清单 10-27 中,几乎每行代码都有注释,结合前面的数据结构和调度算法流程图,看懂该函数应该不是难事。先前就说过,我们并没有实现动态优先级,而是遍历 `schdata_t` 结构中的 `sda_thdlist` 数组,从最小下标开始,找到一个进程拿到 `thdlist_t` 结构的 `tdl_currntd` 域中后就立即返回,如果遍历完了 `sda_thdlist` 数组还是没有找到相应的进程,就返回空转进程。试想一下,代码清单 10-27 中的 `pity` 循环变量,如果开始不是等于 0,而是等于 `schdata_t` 结构中的 `sda_prityidx` 域的值,在找到一个进程后就对 `sda_prityidx` 域中的值加 1,当它加到 `PRITY_MAX` 时,就让它等于 `PRITY_MIN`,这样是不是就实现动态优先级了?

当代码清单 10-27 中的大循环没有找到一个可运行的进程时,就返回当前 CPU 上的空转进程。下面来写一写返回当前 CPU 上的空转进程的函数,如代码清单 10-28[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-28 返回当前CPU上的空转进程

```

thread_t* krlsched_retn_idlethread()
{
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //检查schdata_t结构中的sda_cpuidle域是否为NULL,为NULL就死机
    if(schdap->sda_cpuidle==NULL)
    {
        hal_sysdie("schdap->sda_cpuidle NULL");
    }
    //返回空转进程
    return schdap->sda_cpuidle;
}

```

从前面的数据结构得知,当前 CPU 上的空转进程,正是保存在当前 CPU 对应的 `schdata_t` 结构中的 `sda_cpuidle` 域中的进程。

这样,选择一个可运行进程的代码就写好了,但是这对进程调度还只完成了一半,下面我们就去实现进程切换相关的代码。

10.5.5 进程切换

如果已经选择好了一个进程，剩下的就是进程之间的切换了，即从当前运行的进程切换到选择的进程。

首先要明确，LMOSEM 内核代码是由一堆函数组成的，这堆函数是所有进程公用的。不管一个进程是从中断还是从系统调用进入 LMOSEM 内核的，但它进入之后，就开始执行 LMOSEM 内核的函数，并且在一个进程的内核栈中，都保存着完整的 LMOSEM 内核函数调用路径。

进程切换就是让一个进程停止运行，让另一个进程开始运行，即让 CPU 执行另一个进程的代码。我们的进程切换操作也是由一个函数完成的，那么这就意味着所有进程的开始或者停止都是在该函数中完成的，也就是说在一个进程停止和另一个进程开始的那一刹那，它们的 LMOSEM 内核函数调用路径都是在这个函数内的。当然，如果是新进程，第一次运行时会有不同，因为它的内核栈中是没有 LMOSEM 内核函数调用路径的。这种情况我们会单独处理。文字可能不够形象，画幅图就清楚了，如图 10-10 所示。

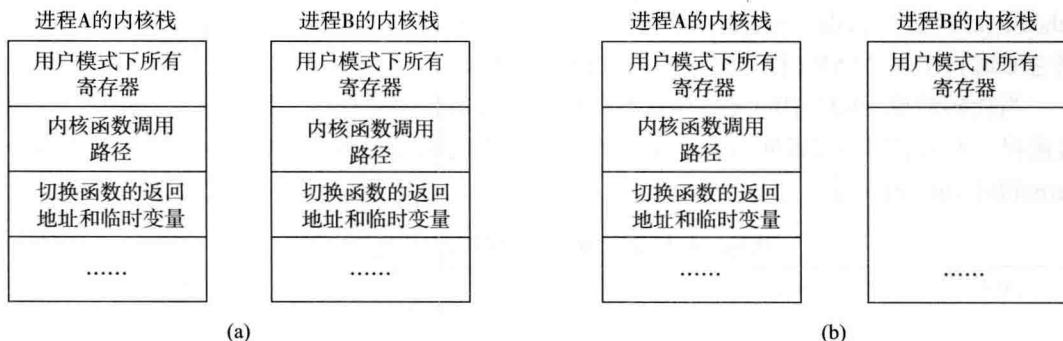


图 10-10 进程切换时其对应内核栈中的情形

在图 10-10 中，两种情况都是从进程 A 切换到进程 B，只是图 10-10 (b) 中的进程 B 是新建的，还没有运行过，它的内核栈中没有 LMOSEM 内核函数调用路径，因为它从未调用过任何 LMOSEM 内核函数。一定要明确，进程切换都是从当前正在运行的进程中切换到一个进程的。例如，在图 10-10 (b) 中，一旦进程 B 开始运行后再次发生进程切换，就会从它切换到别的进程，因此那时它内核栈的情况就会和图 10-10 中的进程 A 一样。图 10-10 (a) 中和 (b) 中的进程 A 的内核栈开始的地方，都保存着 CPU 用户模式下的所有寄存器。这些寄存器，都是由中断处理程序和系统调用处理程序压栈保存的，可以看看中断管理章节。

看到以上这种进程切换模型，你似乎已经明白了切换进程就是切换进程的内核栈，也就是把当前 CPU 的栈指针寄存器的值保存，然后加载下一个进程上次切换时保存的栈指针。其实不仅仅是做好这一点就行了，后面会有介绍的。操作 CPU 的栈指针寄存器需要用汇编代码，用 C 语言是做不到的，所以我们用在 C 语言中嵌入汇编代码来实现这个函数，如代码清单 10-29[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-29 进程切换

```

void save_to_new_context(thread_t* next,thread_t* prev)
{
    cpuflg_t cpuflg;
    //先保存CPSR寄存器中的值,然后关中断
    hal_disableirq_savecpuflg(&cpuflg);
    //保存CPU特权管理模式下的寄存器,因为下面将要使用一些寄存器
    __asm__ __volatile__ ("stmfd sp!,{r0-r12,lr} \n\t"
        ::: "memory"
    );
    __asm__ __volatile__ (
    //保存当前的SPSR寄存器的值到prev->td_context.ctx_svcpsr中
        "mrs lr,spsr\n\t"
        "str lr,[%[PREV_SPSR]] \n\t"
    //保存当前的CPSR寄存器的值到prev->td_context.ctx_cpsr中
        "mrs lr,cpsr\n\t"
        "str lr,[%[PREV_CPSR]] \n\t"
    //装载next->td_context.ctx_svcpsr到当前的SPSR寄存器中
        "msr spsr,%[NEXT_SPSR] \n\t"
    //装载next->td_context.ctx_cpsr到当前的CPSR寄存器中
        "msr cpsr,%[NEXT_CPSR] \n\t"
    //保存当前的SP寄存器的值到prev->td_context.ctx_svcsp中
        "str sp,[%[PREV_SVCSP]] \n\t"
    //装载next->td_context.ctx_svcsp到当前的SP寄存器中
        "ldr sp,[%[NEXT_SVCSP]] \n\t"
    //上面两行代码发生了进程PREV和进程NEXT的内核栈切换,即也发生了进程切换,下面的代码将使用进程
    //NEXT的内核栈运行
    //为下面调用__to_new_context函数准备参数
        "mov r0,%[NEXT_TD] \n\t"
        "mov r1,%[PREV_TD] \n\t"
        "bl __to_new_context \n\t"
        :
        : [PREV_TD] "r" (prev), [NEXT_TD] "r" (next),
        : [PREV_SVCSP] "r" (&prev->td_context.ctx_svcsp), [PREV_SPSR] "r" (&prev->td_context.ctx_svcpsr),
        : [PREV_CPSR] "r" (&prev->td_context.ctx_cpsr),
        : [NEXT_SVCSP] "r" (&next->td_context.ctx_svcsp), [NEXT_SPSR] "r" (next->td_context.ctx_svcpsr),
        : [NEXT_CPSR] "r" (next->td_context.ctx_cpsr)
        : "lr", "cc", "memory"
    );
    //恢复上次进程切换之前保存的寄存器组
    __asm__ __volatile__ (
        "ldmfd sp!,{r0-r12,lr} \n\t"
        ::: "memory"
    );
};

```



```

    hal_enableirq_restcpuflg(&cpuflg); //恢复CPSR寄存器
    return;
}

```

代码清单 10-29 有点难，不过分为了三个部分：第一部分是保存和恢复上次进程切换之前的寄存器组；中间部分是切换进程的内核栈；第三部分是调用 `__to_new_context` 函数，当 `save_to_new_context` 函数的返回指令执行时，由于已经切换了进程的内核栈，而返回指令是根据栈中的返回地址返回的，因此它会完全返回到该进程执行调度代码之前的调用路径上继续执行。这里关键是要理解进程内核栈中保存的信息，进程内核栈中保存的就是进程执行 LMOSEM 内核函数的路径信息及其他临时变量。当进程下一次被切换时，它的内核栈中保存的就是上一次它从 LMOSEM 内核的某个地方，到达 `save_to_new_context` 函数中的“`ldr sp,[%[NEXT_SVCSP]] \n\t`”指令之前的所有 LMOSEM 内核函数的调用路径信息和使用过的临时变量。

但是有个例外，新建进程没有执行任何 LMOSEM 内核代码，它的内核栈中没有 LMOSEM 内核函数的调用路径信息，所以新建进程就不能从 `save_to_new_context` 函数中返回。但是看过空转进程运行的读者知道，新建进程的内核栈中有我们精心安排的一组数据值，只要把它们弹出到 CPU 相应的寄存器中，新建进程就开始运行了。这部分我们放到 `__to_new_context` 函数中去处理，如代码清单 10-30[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-30 开始运行新建进程

```

void __to_new_context(thread_t* next,thread_t* prev)
{
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //让当前正在运行的进程改变为next进程
    schdap->sda_currtd=next;
    //检查next进程是否为新建进程
    if(next->td_stus==TDSTUS_NEW)
    {
        //设置新建进程的状态为运行状态
        next->td_stus=TDSTUS_RUN;
        //弹出新建进程内核栈中的数据到CPU相应的寄存器中，使之运行
        retnfrom_first_sched(next);
    }
    return;
}

```

`__to_new_context` 函数非常简单，就是检查进程是否为新建进程。如果是，就将其状态改为运行状态，并调用 `retnfrom_first_sched` 函数弹出其内核栈中的数据到 CPU 相应的寄存器中，使之运行。如果以后实现了虚拟内存机制，从而需要切换 MMU 页表时，也要放在这个

函数中完成。`retnfrom_first_sched` 函数的原理在前面已经了解了, 该函数不会返回了。

好了, 前面有了选择进程的代码, 这里又写好了切换进程的代码, 现在就来把它们封装一下, 让其变成一个进程调度器函数, 如代码清单 10-31[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-31 进程调度器函数

```
void krlsched()
{
    //返回当前正在运行的进程
    thread_t* prev=krlsched_retn_currthread(),
    //选择下一个可运行的进程
    *next=krlsched_select_thread();
    //进程切换
    save_to_new_context(next,prev);
    return;
}
```

注意, 进程切换一定是从当前正在运行的进程切换到新的可运行的进程, 所以 `krlsched` 函数调用了 3 个函数来完成整个进程调度过程。

进程调度虽难, 还好我们最终还是实现了, 下面就去实现进程的等待与唤醒。

10.5.6 进程等待与唤醒

进程在访问有关资源, 而有关资源又暂时不能满足请求时, 就需要进入等待状态。通常维护资源状态的数据结构都会在其中使用信号量来解决这个问题, 但是当相关资源可以满足请求时, 就应该唤醒等待该资源的进程。例如, 前面驱动模型章节中的 RTC 驱动程序的闹钟功能就使用了信号量机制。

实际上, 信号量是一个上层的数据结构, 它的底层数据结构是 LMOSEM 内核等待链: `kwlst_t` 数据结构, 虽然前面已经看过了, 在这里还是来回忆一下, 如代码清单 10-32[xxxx/lmosem/include/bastypeinc/sem_t.h] 所示。

代码清单10-32 等待链表数据结构

```
typedef struct s_KWLST
{
    spinlock_t wl_lock; //自旋锁
    uint_t    wl_tdnr; //进程计数
    list_h_t  wl_list; //挂载等待进程的链表头
}kwlst_t;
```

进程进入等待状态的方法, 就是把当前运行的进程从进程调度表中拿出来, 设置为等待状态, 然后把它挂载到相应的 `kwlst_t` 数据结构中。如果用一个函数来完成, 就如代码清单 10-33[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-33 让进程进入等待状态的函数

```

void krlsched_wait(kwlst_t* wlst)
{
    cpuflg_t cuflg,tcuflg;
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    //获取当前运行的进程
    thread_t* tdp=krlsched_retn_currthread();
    uint_t pity=tdp->td_priority;
    //检查进程的优先级和函数的参数是否合理
    if(pity>=PRITY_MAX||wlst==NULL)
    {
        goto err_step;
    }
    if(schdap->sda_thdlst[pity].tdl_nr<1)
    {
        goto err_step;
    }
    hal_spinlock_saveflg_cli(&schdap->sda_lock,&cuflg);
    hal_spinlock_saveflg_cli(&tdp->td_lock,&tcuflg);
    //设置进程状态为等待状态
    tdp->td_stus=TDSTUS_WAIT;
    //把进程从调度进程表中的相关链表上拿出来
    list_del(&tdp->td_list);
    hal_spinunlock_restflg_sti(&tdp->td_lock,&tcuflg);
    //如果拿出的进程和当前优先级链表上正在运行的进程相同,就让其为NULL
    if(schdap->sda_thdlst[pity].tdl_currntd==tdp)
    {
        schdap->sda_thdlst[pity].tdl_currntd=NULL;
    }
    //因为有进程被拿出了,所以要减少进程计数
    schdap->sda_thdlst[pity].tdl_nr--;
    hal_spinunlock_restflg_sti(&schdap->sda_lock,&cuflg);
    //把进程加入相关的kwlst_t结构中
    krlwlst_add_thread(wlst,tdp);
    //返回
    return;
err_step:
    //有错误就死机
    hal_sysdie("krlsched_wait err");
    return;
}

```

以上这种进程等待的方式是一种非剥夺式的方式,是进程访问相关资源得不到满足时不得不自愿放弃运行的行为,所以自然也就是当前正在运行的进程进入等待状态。因此,代码

清单 10-33 中调用 `krlsched_retn_currthread` 函数获取当前运行的进程，然后把它拿出来，调用 `krlwlst_add_thread` 函数把它加入到相关的 `kwlst_t` 结构中。下面来写好 `krlwlst_add_thread` 函数，如代码清单 10-34 所示。

代码清单10-34 把进程加入到相关的kwlst_t

```
void krlwlst_add_thread(kwlst_t* wlst, thread_t* tdp)
{
    cpuflg_t cufg;
    //加锁
    hal_spinlock_saveflg_cli(&wlst->wl_lock, &cufg);
    //把进程加入kwlst_t结构的链表中
    list_add(&tdp->td_list, &wlst->wl_list);
    //增加等待进程的计数
    wlst->wl_tdnr++;
    //解锁
    hal_spinunlock_restflg_sti(&wlst->wl_lock, &cufg);
    return;
}
```

`krlwlst_add_thread` 函数的代码实在太简单，可是我们好像还有一件重要的事没有做。当一个进程进入等待状态后，一定要调度其他进程运行才行，可是无论是 `krlwlst_add_thread` 函数还是 `krlsched_wait` 函数，都没有调用进程调度器函数，其实它是在调用 `krlsched_wait` 函数的上层函数中被调用的。

过了一段时间后，当资源满足请求时，就要唤醒等待中的进程，其方法正好相反，首先从相关的 `kwlst_t` 结构中拿出进程，然后设置进程的状态为运行状态，最后把这个进程加入到进程调度表中，如代码清单 10-35[xxxx/lmosem/kernel/krlsched.c] 所示。

代码清单10-35 唤醒等待中的进程

```
void krlsched_up(kwlst_t* wlst)
{
    cpuflg_t cufg, tcufg;
    uint_t cpuid=hal_retn_cpuid();
    //获取当前CPU的schdata_t结构的指针
    schdata_t* schdap=&osschedcls.scls_schda[cpuid];
    thread_t* tdp;
    uint_t pity;
    //检查函数的参数
    if(wlst==NULL)
    {
        goto err_step;
    }
    //返回kwlst_t结构中的进程
    tdp=krlwlst_del_thread(wlst);
```

```

    if(tdp==NULL)
    {
        goto err_step;
    }
    pity=tdp->td_priority;
//检查进程的优先级是否合理
    if(pity>=PRITY_MAX)
    {
        goto err_step;
    }
    hal_spinlock_saveflg_cli(&schdap->sda_lock,&cufg);
    hal_spinlock_saveflg_cli(&tdp->td_lock,&tcufg);
//设置进程的状态为运行状态
    tdp->td_stus=TDSTUS_RUN;
    hal_spinunlock_restflg_sti(&tdp->td_lock,&tcufg);
//把该进程加入调度进程表的相应优先级链表中
    list_add_tail(&tdp->td_list,&(schdap->sda_thdlst[pity].tdl_lsth));
//增加其计数
    schdap->sda_thdlst[pity].tdl_nr++;
    hal_spinunlock_restflg_sti(&schdap->sda_lock,&cufg);
    return;
err_step:
//若有错误就死机
    hal_sysdie("krlsched_up err");
    return;
}

```

在代码清单 10-35 中，用 `krlwlst_del_thread` 函数返回 `kwlst_t` 结构中的进程。来看看这个函数，如代码清单 10-36[xxxx/lmosem/kernel/krlwaitlist.c] 所示。

代码清单10-36 删除kwlst_t中相关的进程

```

thread_t* krlwlst_del_thread(kwlst_t *wlst)
{
    thread_t* tdp=NULL;
    cpuflg_t cufg;
    list_h_t* list;
    hal_spinlock_saveflg_cli(&wlst->wl_lock,&cufg);
//遍历wlst->wl_list链表，只要找到一个进程就立即返回
    list_for_each(list,&wlst->wl_list)
    {
        tdp=list_entry(list,thread_t,td_list);
        //检查结果和wlst->wl_tdnr是否合理
        if(tdp==NULL||wlst->wl_tdnr<1)
        {
            tdp=NULL;
            goto return_step;
        }
    }
}

```

```

    }
    //把该进程从wlst->wl_list链表上删除
    list_del(&tdp->td_list);
    //减少计数
    wlst->wl_tdnr--;
    //这样做只是为了让代码更清晰一点，不会有什么作用
    tdp=tdp;
    //返回
    goto return_step;
}

//到这里说明没有找到进程，所以就返回NULL
tdp=NULL;
return_step:
hal_spinunlock_restflg_sti(&wlst->wl_lock,&cufig);
return tdp;
}

```

代码清单 10-36 中的注释已经很明确了，不必多说。同样，在调用 `krlsched_up` 函数的上层函数中要调用设置进程调度标志的函数，告诉 LMOSEM 内核需要调度进程了。

进程的等待与唤醒就写到这里了。

10.5.7 进程测试

对任何 LMOSEM 内核管理组件的代码，不测试是不能判定其正确性的，尽管我们自我感觉代码已经写得很好了，但是还是要上机运行一下，真金不怕火炼。

在前面的章节中已经说过了，我们的应用程序代码和 LMOSEM 内核代码是链接到一起的，因此，应用程序代码和 LMOSEM 内核代码是运行在同一个物理地址空间中的。

要测试进程，得先有应用程序代码，下面我们来写好两个应用程序的主函数。先在“`xxxx/lmosem/utask/`”目录下建立一个 `utask.c` 文件，在这个文件中写好两个函数，如代码清单 10-37[`xxxx/lmosem/utask/utask.c`] 所示。

代码清单10-37 两个应用程序的代码

```

void task1_main()
{
    int i = 0;
    for (;;)
    {
        i++;
        printfk("task1 run %x \n\r", i);
    }
    return;
}

void task0_main()

```

```

{
    int i = 0;
    for (;;)
    {
        i++;
        printfk("task0 run %x \n\r", i);
    }
    return;
}

```

首先有两个问题：第一个问题是，为什么代码清单 10-37 中的两个函数名都不是 main，而是不同的名字，那是因为应用程序代码和 LMOSEM 内核代码链接到一起，函数名相同就会导致链接器发出符号重复的错误；第二个问题是，printfk 函数是 LMOSEM 内核的函数，为什么在应用程序代码中可以直接调用，也是因为应用程序代码和 LMOSEM 内核代码链接到一起的原因，况且我们还没有实现 LMOSEM 系统的 API 函数，这是唯一能看到进程正确运行的方法。等后面我们实现了 LMOSEM 系统的 API 函数，再重新实现应用程序调用的 printf 函数，这里先将就一下。

代码清单 10-37 只是两个应用程序的代码，一个应用程序要运行就必须先新建一个进程。在 Windows 下双击应用程序图标，Windows 的 shell 就会找到对应的应用程序并为此新建一个进程。我们的 LMOSEM 系统还不行，所以得想个办法，让 LMOSEM 系统在启动时就自动新建两个进程，如代码清单 10-38[xxxx/lmosem/kernel/task.c] 所示。

代码清单10-38 使两个应用程序运行

```

void task0()
{
    //调用krlnew_thread函数新建一个进程，如果失败就死机
    thread_t* thp= krlnew_thread((void*)task0_main,0,PRILG_USR,PRITY_MIN,0,0);
    if(thp==NULL)
    {
        hal_sysdie("task0 err");
    }
    return;
}

void task1()
{
    //调用krlnew_thread函数新建一个进程，如果失败就死机
    thread_t* thp= krlnew_thread((void*)task1_main,0,PRILG_USR,PRITY_MIN,0,0);
    if(thp==NULL)
    {
        hal_sysdie("task1 err");
    }
    return;
}

```

```

void init_task()
{
    //新建进程0
    task0();
    //新建进程1
    task1();
    return;
}

```

代码清单 10-38 非常简单，不必多说。下面还要把 `init_task` 函数加入到 LMOSEM 内核功能层的主初始化函数 `init_krl` 中，如代码清单 10-39[xxxx/lmosem/kernel/krlinit.c] 所示。

代码清单 10-39 LMOSEM 内核功能层的初始化函数

```

void init_krl()
{
    init_krlmm();
    init_krldevice();
    init_krldriver();
    init_krlsched();
    init_ktime();
    init_task();
    init_krlcpuidle();
    hal_sysdie("LMOSEM RUN!!");
    return;
}

```

一定要注意，`init_task` 函数一定要放在 `init_krlcpuidle` 函数之前，因为 `init_krlcpuidle` 函数一旦调用就不会返回了，它会建立空转进程，然后直接就运行空转进程了。注意，还需要修改一下 `krlcpuidle_main` 函数，因为它需要调用进程调度器函数，如代码清单 10-40[xxxx/lmosem/kernel/krlcpuidle.c] 所示。

代码清单 10-40 空转进程的主函数

```

void krlcpuidle_main()
{
    //这个进程不会退出，所以是死循环
    for(;;)
    {
        //为了便于观察，先把下面这行代码注释掉
        //printfk("cpuidle is run!!\n\r");
        //调用进程调度器函数
        krlschedul();
    }
    return;
}

```


按照前面章节中的方法编译一下 LMOSEM 内核，连接好 mini2440 开发板，把 LMOSEM 内核文件下载到 mini2440 开发板中运行，在正常情况下应该会看到如图 10-11 所示的情况。



```
深度终端
task1 run 2f74
task1 run 2f75
task1 run 2f76
task1 run 2f77
task1 run 2f78
task1 run 2f79
task1 run 2f7a
task1 run 2f7b
task1 run 2f7c
task0 run 2f75
task0 run 2f76
task0 run 2f77
task0 run 2f78
task0 run 2f79
task0 run 2f7a
task0 run 2f7b
task0 run 2f7c
task0 run 2f7d
```

图 10-11 多进程运行

在图 10-11 中，不断滚动交替出现的“task1 run……”和“task0 run……”的字符串，这说明了我们的两个进程正常运行了，并且间接地证明了 LMOSEM 内核的进程调度器代码也工作正常了。

关于进程的测试就到这里了，笔者只用了两个进程，读者还可以使用更多的进程进行测试看看行不行，当然那是绝对可行的。进程相关的内容介绍完了，但是 LMOSEM 内核还没介绍完，休息一下，我们继续探索。

10.6 小结

到这里，我们又实现了一个重要的 LMOSEM 内核组件——进程管理。这样，我们的 LMOSEM 内核可以运行、调度进程了。这是现代操作系统的重要概念。

现在来回顾一下本章中我们都干了些什么，如下：

- 1) 应用程序的运行，从应用程序的运行了解了它的运行需要一些什么资源，然后发现有时一些资源不可用，最后由此提出了多道程序模型。为了实现多道程序模型，提出了进程概念。
- 2) 为了实现进程这一概念，我们设计了一整套数据结构。
- 3) 从 LMOSEM 内核的第一个进程即空转进程开始，了解了进程的初始化，为什么要建立空转进程，以及如何让空转进程运行起来。
- 4) 操作系统都需要有新建进程的功能，所以我们实现了建立进程的一套接口函数。

5) 由于系统中可能有多个进程, 所以必须要实现进程调度, 由此了解了一个极为简单的进程调度算法, 以及如何选择进程并进行切换等, 最后进行了进程测试。

一个被那些理论家用大篇幅说得神乎其神的进程概念, 就这样被我们轻而易举地掌握了。不过, LMOSEM 内核中还有一些重要的东西没有完成, 先休息一下, 然后我们继续探索。

文件系统

经常在计算机上听 APE 音乐、看 4K 视频、阅读各种文档、浏览各种精美的网页，这些东西都是一些特定格式的数据，操作系统内称它们为文件。这些文件可能存储在 HD 机械硬盘、SSD 固态硬盘、TF 卡，甚至远程计算机上。

如何把许多文件存储在一种存储介质之上，并十分方便地读写、增加、删除，这也是操作系统的重要任务。为此，操作系统还分出一个子系统——文件系统，来专门处理这些问题。这里先从文件系统设计开始，接着设计并定义一些相关的数据结构，然后把这些数据结构写入到存储介质上完成文件系统的格式化并在实现文件系统的基础设施后实现文件的存储、删除、读写操作。

以上看上去好像有点复杂，是个技术活，但是别怕，毕竟我们不是干这事的第一批人，可以参考别人的设计与实现。好了，废话不多说，难不难做了就知道。

11.1 文件系统设计

又说到了设计，其实我们没资格谈设计的，只是惯用这个名词而已。下面首先从文件系统只是一个设备开始，了解文件系统在整个 LMOSEM 内核中的位置格局，接着了解文件数据的格式以及存储介质的最小单位，然后是如何组织越来越多的文件，最后了解我们的这个文件系统的相关限制。

11.1.1 文件系统只是一个设备

设备在大部分读者心中，是一个芯片，或者是一个具体的物理器件，而文件系统好像不

是一个具体的物理设备。是的，这里所说的文件系统，不过是为了解决如何管理、存储用户的文件数据，是属于软件部分的。可是为什么说它是一种设备，请继续往下看。

现在存储介质有很多，例如，HD 机械硬盘、SSD 固态硬盘、U 盘、各种 TF 卡等。在这些设备上，文件存储格式都不相同，甚至同一个硬盘上不同的分区的存储格式也不相同，这个存储格式就是相应文件系统在存储介质上组织存储文件的方式。例如，经常看到的 FAT32、NTFS、ext4、btrfs、zfs、hpfs 等，这些都是不同的文件系统建立的文件系统格式。

看到上面的情况之后，不难发现让文件系统成为 LMOSEM 内核中固定的一部分，是个非常愚蠢的想法。于是我们有了如下设想：

1) 文件系统组件是独立的，是和 LMOSEM 内核分开的，不是 LMOSEM 内核的一部分。

2) 文件系统组件是可以动态加载和删除的。

3) 可以动态加载和删除不同的文件系统组件。

以上三点，就可以适用于现在这种复杂的应用情况了，例如，硬盘上不同的分区有不同的文件系统格式，可拔插各种不同文件系统格式的 U 盘、TF 卡了，如 FAT32 格式的 TF 卡、ext4 格式的 U 盘等。

看到这里，可能想起前面 LMOSEM 内核的设备驱动模型的设计。如果文件系统也是 LMOSEM 内核下的一个设备，那就好办多了。因为不同的设备驱动程序可以动态加载，而且可以建立多个文件系统设备，而对各个文件系统设备驱动程序的实现，就是各个文件系统的实现。刚好前面的驱动模型中，定义了文件系统的设备类型。下面来画幅模型图就清楚了，如图 11-1 所示。

上面已经说过了，文件系统一定要有存储介质，这个存储介质可以是硬盘，也可以是 TF 卡。总之能存储数据的设备都是可以的，在我们所用的计算平台 mini2440 上，只有 Nandflash 设备，可以作为存储介质。可是这个设备操作起来非常复杂，要编写非常复杂的 Nandflash 设备驱动程序，为了减小程序的复杂程度，以便突出原理，我们使用一块 4MB 大小的内存空间来模拟存储介质，何况世界上又不是我们第一次建造内存文件系统（ramfs），只是我们做得更小。

在文件系统设备驱动程序的人口函数中，分配这个 4MB 大小的内存空间。相信即使如此，也能让我们清楚地看到文件系统的实现，只是为了让实现变得简单。等哪天有时间了，写好了 Nandflash 设备驱动程序，也可以让文件系统设备驱动程序处理好了数据，然后发送给 Nandflash 设备驱动程序，让其写入到 Nandflash 设备中去。这在我们设计的驱动模型中是完全允许的。

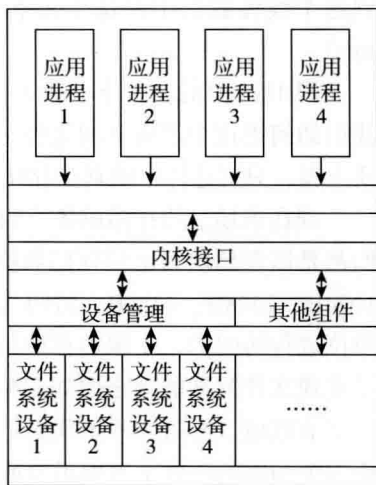


图 11-1 文件系统设备

可以看到，LMOSEM 内核并不要求一个设备与真正的物理设备对应，只是把相应的操作发送给设备驱动程序，然后设备驱动程序按照要求做好事就行了。这里只要了解文件系统在 LMOSEM 内核中，不过是一个可以动态加载的设备驱动程序，并且知道我们用了一块 4MB 大小的内存空间模拟了文件系统的存储介质就行了。

11.1.2 数据格式与存储块

通常说的文件，都是由一堆数据组成的，当我们把这堆数据组织成一个文件，存储在存储介质上时，就有了一问题：我们按什么格式，什么方式把这些数据存放在存储介质上。

以上说的格式是指文件系统存放文件数据的格式，不是指文件数据本身的格式，文件数据本身的格式，文件系统不该多管，如 MP3、Word 文档、各种视频数据的内部格式，各不相同。

关于文件系统存放文件数据的格式，类 UNIX 系统和 Windows 系统都采用了相同的方案，那就是逻辑上认为一个文件就是一个可以动态增加、减少的线性字节数组，即文件数据的每个字节都一一对应到这个线性数组中的每个元素。那么我们也和它们一样，如图 11-2 所示。

图 11-2 中的文件字节数组，终究是逻辑上的，所以问题又来了，我们如何把这个逻辑上的文件字节数组映射或者转换到具体的存储介质上呢，只有这样才能真正存储数据。

现在市场上的存储设备，如机械硬盘、SSD 固态硬盘、TF 卡，它们都是以存储块为单位存储数据的，一个存储块的大小可以是 512B、1024B、2048B、4096B，访问这些存储设备也是以最少一个存储块为单位进行访问的，不像内存设备可以最少访问一个字节。文件系统为了实现文件数据的动态增加和减少，把文件数据定义成一个动态的线性字节数组，可是一开始我们不知道这个数组是多大，需要分配多少个物理存储块，最好是把这个动态的线性字节数组分成一个个数据块，在需要改变数组大小时就按需要增加或者释放一个或多个数据块，可是问题还没有完，不同的存储介质的物理存储块的大小是不同的，如有的是 512B，而有的是 4096B。我们为了使文件系统能工作在不同的存储介质上，所以我们把这里的数据块定义为文件系统逻辑块，其大小为 4096B，然后把这个逻辑块映射到一个或多个物理存储块，如图 11-3 所示。

好了，我们已经看到了从文件这个抽象的概念到文件字节数组，再到文件逻辑块，最后到存储介质上的物理存储块，这一整个演变过程。了解了这个过程以及为什么要这样做，就可以了。

文件字节数组

文件字节
文件字节
文件字节
文件字节
文件字节
文件字节
文件字节
文件字节
文件字节
文件字节
.....
文件字节

图 11-2 文件数据格式

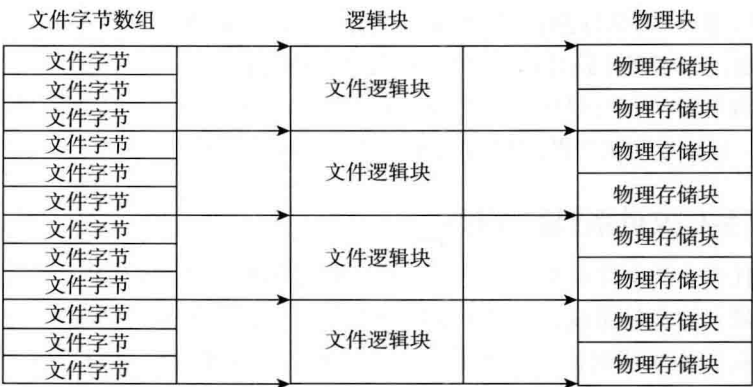


图 11-3 文件字节到物理存储块的映射

11.1.3 如何组织文件

现在，PC 上的文件数量都已经上十万的数量级了，网络服务器上更是不止。如果把十万个文件顺序地排列在一起，要找出其中一个文件，那是非常困难的，即使是计算机程序查找起来也是相当慢的，加上硬盘、TF 卡之类的存储介质比内存慢得多，因此会变得更慢。

我们的先辈发明了一个伟大的东西，叫文件目录或者叫文件夹。当然这里习惯将其称为目录。这样就可以用不同的目录来归纳不同的文件，目录可以自定义名称，例如，我们可以在 MP3 目录下存放音频文件，可以在 MP4 目录下存放视频文件。同时目录之下还可以创建目录。这样就建立了非常好的层次关系。

经常在 Linux 系统中看到如 “/ddd1/fff1，/ddd1/ddd2/ddd3/fff2” 之类的东西，其中 ddd1、ddd2、ddd3 就是目录，fff1、fff2 就是文件，“/” 符号就是文件路径分隔符，它们合起来就是文件路径名。可以看出，整个文件层次结构就像是一棵倒挂的树。画幅图就更清楚了，如图 11-4 所示。

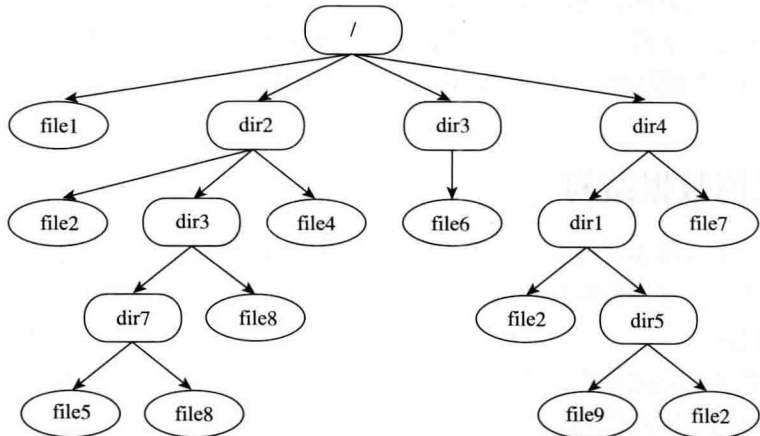


图 11-4 文件、目录的组织结构

图 11-4 中, file9 的文件路径是 /dir4/dir1/dir5/file9。file1 的文件路径是 /file1。开始的“/”表示根目录,一切文件路径都是从根目录那里开始的。

由于文件数量多了,所以出现了目录,后面我们的文件系统也会采用目录来组织文件。这里只要明白,目录是用来帮助用户组织或归纳文件的就行了。

11.1.4 关于我们文件系统的限制

一个完整且强大的文件系统的实现,需要复杂的算法和大量的代码,花大量的精力去实现一个完整且强大的文件系统,这对于我们这些学习文件系统原理的初学者来说,并不好,因为这样会让我们陷入代码的汪洋大海中,无法看清文件系统的全貌。当然也不能太过简单,太过简单了也不足以体现文件系统的原理。

为了解决上面的问题,我们采用了一个折中的方案,就是在设计文件系统相应的数据结构时,严格按照常规文件系统的需要来设计,但是在编写代码实现时,我们做出如下限制:

1) 不创建目录,所有文件都在根目录“/”下,即文件路径名都是这样的形式: /file、/file1、/file2 等。但是目录数据结构是可以允许建立目录的,不管多少层目录都可以。

2) 每个文件最多只能分配一个存储块(4KB 大小)。同样,数据结构是可以允许分配多个存储块的,并且一次可以分配多个连续的存储块作为一个大的存储块使用。

3) 一个文件的控制信息数据结构,并不是保存在固定的位置,而是保存在该文件的第一个存储块的头 512B 的空间中的。

4) 暂不支持文件的跳转随机读写,一旦发生读写操作,我们的文件系统会把一个文件的全部数据都返回给请求者,或者更新该文件的全部数据,因为前面的规定,一个文件最大也就只能允许(4096B - 512B)大小。

5) 目录也是一个文件,也有一个文件的控制信息数据结构,并且也是只占有一个存储块。

好了,有了上述限制,我们的文件系统实现起来也就简单多了。尽管有了很多限制,但是我们在设计上是完整的,只是代码实现的问题。哪天有时间了我们可以继续完善,现在就去设计相应的一些数据结构,一步步实现我们的文件系统。

11.2 相关的数据结构

一路走来,发现操作系统内核的任何组件的实现,都需要设计一套相应的数据结构,文件系统也不例外。这在开发应用程序中很少见,那是因为开发操作系统内核面对的是机器,除了机器什么也没有。

好了,那么文件系统需要些什么数据结构呢?第一是超级块,它保存着文件系统的重要信息,如文件系统对应的存储介质的空间大小、根目录位置、位图块位置等。接着就是位图数据结构,它用于分配回收文件存储块。然后是目录数据结构,它保存着文件名和文件位置

信息。最后是文件管理头数据结构，它保存着一个文件的大小、访问和创建时间、文件占用的文件存储块的信息等。

下面就去设计它们，把它们变成相应的数据结构。

11.2.1 超级块

一个文件系统有很多重要的信息，它们既要保存又要经常变化。为了能方便地操作它们，所以得把它们归纳成一个数据结构。

那么到底一个文件系统有哪些重要的信息呢，例如，文件系统标识、版本、状态，存储介质大小，文件系统逻辑存储块大小，位图所在的存储块，还有根目录等。因为这些信息很重要，没有它们就等于没有文件系统，所以称包含这些信息的数据结构为文件系统的超级块。

下面就来把上面这些信息，变成相应的数据结构，如代码清单 11-1[xxxx/lmosem/include/drvinc/drvrfs_t.h] 所示。

代码清单11-1 超级块数据结构

```
typedef struct s_RFSSUBLK
{
    spinlock_t rsb_lock;           //超级块在内存中使用的自旋锁
    uint_t rsb_mgic;               //文件系统标识
    uint_t rsb_vec;                //文件系统版本
    uint_t rsb_flg;                //标志
    uint_t rsb_stus;               //状态
    size_t rsb_sz;                 //该数据结构本身的大小
    size_t rsb_sblkisz;            //超级块大小
    size_t rsb_dblkisz;            //文件系统逻辑存储块大小，我们这里用的是4KB
    uint_t rsb_bmpbks;             //位图的开始逻辑存储块
    uint_t rsb_bmpbknr;            //位图占用多少个逻辑存储块
    uint_t rsb_fsyzallblk;         //文件系统有多少个逻辑存储块
    rfsdir_t rsb_rootdir;          //根目录，后面会看到这个数据结构的
}rfssublk_t;
```

我们文件系统的超级块，保存在存储介质的第一个逻辑存储块中，也就是说超级块占用存储介质的第一个 4KB 大小的空间，但是它本身的大小没有 4KB，后面多余的空间留作以后扩展所用。rfsdir_t 数据结构是一个目录数据结构，后面我们会有介绍的。当然把根目录数据结构直接放在超级块中，目前也是可行的，反正现在超级块中有多余的空间。

到这里文件系统的超级块数据结构就设计好了，下面就去看看位图数据结构。

11.2.2 位图

我们把一个存储介质分成一个个逻辑存储块（4KB）。当存储一个文件数据时，就按逻辑存储块进行分配。当然，一个文件可以占用一个逻辑存储块或者是多个逻辑存储块，这是

根据文件大小决定的。那么就有了一个问题，怎么来标识哪些逻辑存储块是空闲的，哪些逻辑存储块是已经分配占用的？

为了解决这个问题，先辈提出了位图。这里的位图，就是利用一块存储空间中所有位的状态，达到映射逻辑存储块状态（是否已分配）的目的。一个字节是 8 个位，那么 4KB 的存储空间中，就有 (4096×8) 个位，每个位映射到一个逻辑存储块，例如，第 1 个字节的 8 个位就映射存储介质的 1 ~ 8 个逻辑存储块，第 2 个字节的 8 个位就映射存储介质的 9 ~ 16 个逻辑存储块。其中，一个位的值为 0，就表示该位对应的逻辑存储块是空闲的，反之为 1 就表示对应的逻辑存储块是占用的。

位图并不需要定义实际的数据结构，在实际操作时，我们把位图这个存储块当成一个字节数组就行了。我们用了一块 4MB 的内存空间模拟存储介质，而且规定逻辑存储块为 4KB 大小，所以一共只有 1024 个逻辑存储块。远远小于 4096×8 ，甚至小于 4096，而我们的位图块最少也是一个逻辑存储块，所以，用不着把所有位都利用起来，操作一个个位，是非常麻烦的。在这种情况下，完全可以用一个字节表示一个逻辑存储块是否空闲还是占用。

上面的说明可能还是难以让人明白，画一幅图就清楚多了，如图 11-5 所示。

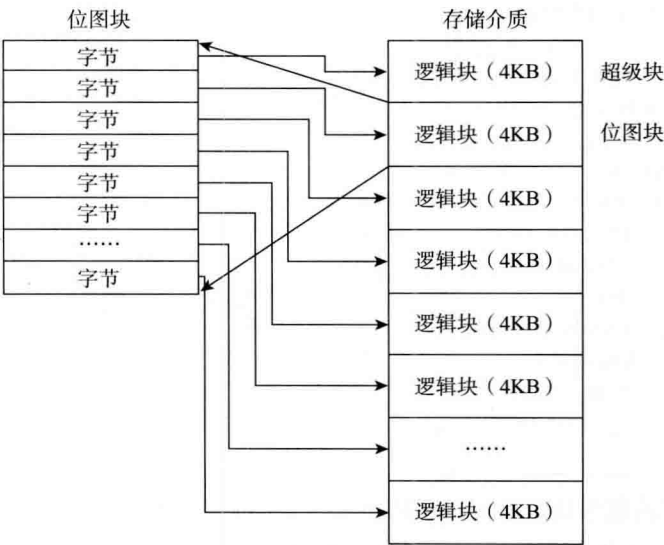


图 11-5 文件系统位图块

图 11-5 表明位图块中的每个字节对应一个逻辑存储块。字节中为 0 的表示该字节对应的逻辑存储块是空闲的，为 1 表示该字节对应的逻辑存储块已经分配占用。

位图实在是很简单，它主要是用于管理存储介质空间的，我们之所以可以用字节来管理存储介质空间，是因为我们的存储介质空间很小，如果上几百吉字节的空间，用字节的话你会发现，需要非常多的位图块，那就可以利用字节中的每个位了。下面我们接着去看看文件系统的目录数据结构。

11.2.3 目录

在前面的文件系统设计中, 为了方便计算机或者用户查找和归纳越来越多的文件, 从而提出了目录。

目录从本质上来说也是一种数据。这种数据中包含了目录类型、状态、指向文件数据管理头的块号、名称等信息。下面就来把这些信息整理成数据结构, 方便以后操作和使用, 如代码清单 11-2[xxxx/lmosem/include/drvinc/drvrfs_t.h] 所示。

代码清单 11-2 目录数据结构及相关宏

```
#define DR_NM_MAX (128-(sizeof(uint_t)*3))
#define RDR_NUL_TYPE 0
#define RDR_DIR_TYPE 1
#define RDR_FIL_TYPE 2
#define RDR_DEL_TYPE 5
.....
typedef struct s_RFSDIR
{
    uint_t rdr_stus;           // 目录状态
    uint_t rdr_type;          // 目录类型, 可以是空类型、目录类型、文件类型、已删除的类型
    uint_t rdr_blknr;         // 指向文件数据管理头的块号, 不像内存可以用指针, 只能按块访问
    char_t rdr_name[DR_NM_MAX]; // 名称数组, 大小为 DR_NM_MAX
}rfmdir_t;
```

从 DR_NM_MAX 宏, 可以看出 rfmdir_t 数据结构最多只有 128B 大小。而名称数组的大小就是 128 减去 3 个 4 字节。由于存储介质不能像内存一样, 用地址访问, 它只能一块一块地访问, 所以 rfmdir_t 结构中有个域, 指向文件数据管理头的块号。

那么为什么 rfmdir_t 结构中会有很多类型呢, 奇怪的是还有文件类型, 它不是目录吗? 这里要注意的是 rfmdir_t 结构并不表示目录的数据。前面已经说过, 目录也是一种特殊的文件, 它里面就是保存着一系列的 rfmdir_t 结构的实例变量, 这些 rfmdir_t 结构再次表明它代表的是一个文件, 还是一个目录, 而 rfmdir_t 结构本身是用于保存一个文件或者目录的名称和指向管理这个文件数据的文件数据管理头的块号, 如图 11-6 所示。

图 11-6 中可以看到, 超级块中的 rfmdir_t 结构只是保存了根目录的名称和指向管理根目

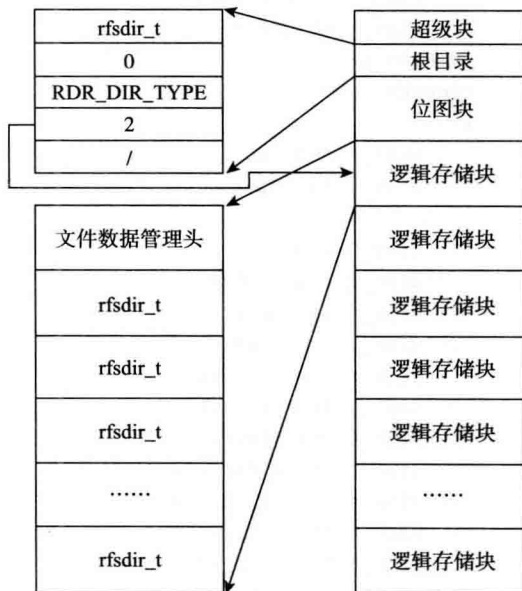


图 11-6 目录结构

录数据的文件数据管理头的块号。而实际的目录数据保存在逻辑存储块中。这正体现了前面所说的目录也是一种数据。当然这种数据就是一系列的 `rfmdir_t` 结构的实例变量。再通过一系列的 `rfmdir_t` 结构就能找到根目录下的其他文件和目录了。

目录数据结构非常简单，但是关键是要理解目录也是一种文件，这个文件也有对应的文件数据管理头，也要分配相应的逻辑存储块。图 11-6 已经说明了一切。现在就去看看什么是文件数据管理头。

11.2.4 文件管理头

一个文件的信息难道真的只有一个文件名称吗，显然不是的，它还有状态、类型、创建时间、访问时间、大小，更为重要的是要知道该文件使用了哪些逻辑存储块。

下面就来把上述所有的文件信息，归纳整理成一个数据结构，称为文件管理头，如代码清单 11-3[xxxx/lmosem/include/drvinc/drvrfs_t.h] 所示。

代码清单11-3 文件管理头数据结构及相关宏

```
#define FBLKS_MAX 32
.....

#define FMD_NUL_TYPE 0
#define FMD_DIR_TYPE 1
#define FMD_FIL_TYPE 2
#define FMD_DEL_TYPE 5                                //文件管理头，也需要表明它管理的是目录文件还是普通文件
typedef struct s_FILBLKS
{
    uint_t fb_blkstart;                                //开始的逻辑存储块号
    uint_t fb_blknr;                                    //逻辑存储块的块数，从blkstart开始的连续块数
}filblks_t;
typedef struct s_fimgrhd
{
    uint_t fmd_stus;                                    //文件状态
    uint_t fmd_type;                                    //文件类型：可以是目录文件、普通文件、空文件、
                                                        //已删除的文件
    uint_t fmd_flg;                                      //文件标志
    uint_t fmd_sfblk;                                    //文件管理头自身所在的逻辑存储块
    uint_t fmd_acss;                                    //文件访问权限
    uint_t fmd_newtime;                                //文件的创建时间，换算成秒
    uint_t fmd_acstime;                                //文件的访问时间，换算成秒
    uint_t fmd_fileallbk;                              //文件一共占用多少个逻辑存储块
    uint_t fmd_filesz;                                  //文件大小
    uint_t fmd_fileifstbkoff;                          //文件数据在第一块逻辑存储块中的偏移
    uint_t fmd_fileiendbkoff;                          //文件数据在最后一块逻辑存储块中的偏移
    uint_t fmd_curfwritebk;                            //文件数据当前将要写入的逻辑存储块
    uint_t fmd_curfinwbkoff;                          //文件数据当前将要写入的逻辑存储块中的偏移
    filblks_t fmd_fleblk[FBLKS_MAX];                  //文件占用逻辑存储块的数组，一共32个filblks_t结构
    uint_t fmd_linkpblk;                                //指向文件的上一个文件管理头的逻辑存储块
```

```
uint_t fmd_linknblk; //指向文件的下一个文件管理头的逻辑存储块
}fimgrhd_t;
```

fimgrhd_t 数据结构中，其他信息都比较容易懂，关键是要理解 fmd_fleblk 数组，它其中的每个元素都可以保存一片连续的逻辑存储块，例如，一个文件占用：4~8、10~15、30~40 的逻辑存储块，那么就在 fmd_fleblk[0] 中保存 4 和 4，在 fmd_fleblk[1] 中保存 10 和 5，在 fmd_fleblk[2] 中保存 30 和 10。细心的读者可以发现当文件特别大时，fmd_fleblk 数组元素可能就不够用了，但是我们想了一个办法，在 fmd_fleblk 数组元素用完时就再分配一个逻辑存储块，在里面再次存放同一个文件的 fimgrhd_t 结构，让该文件的上一个 fimgrhd_t 结构中的 fmd_linknblk 域指向这个逻辑存储块，让这个逻辑存储块中的 fimgrhd_t 结构中的 fmd_linkpblk 域指向该文件的上一个 fimgrhd_t 结构所在的逻辑存储块，如图 11-7 所示。

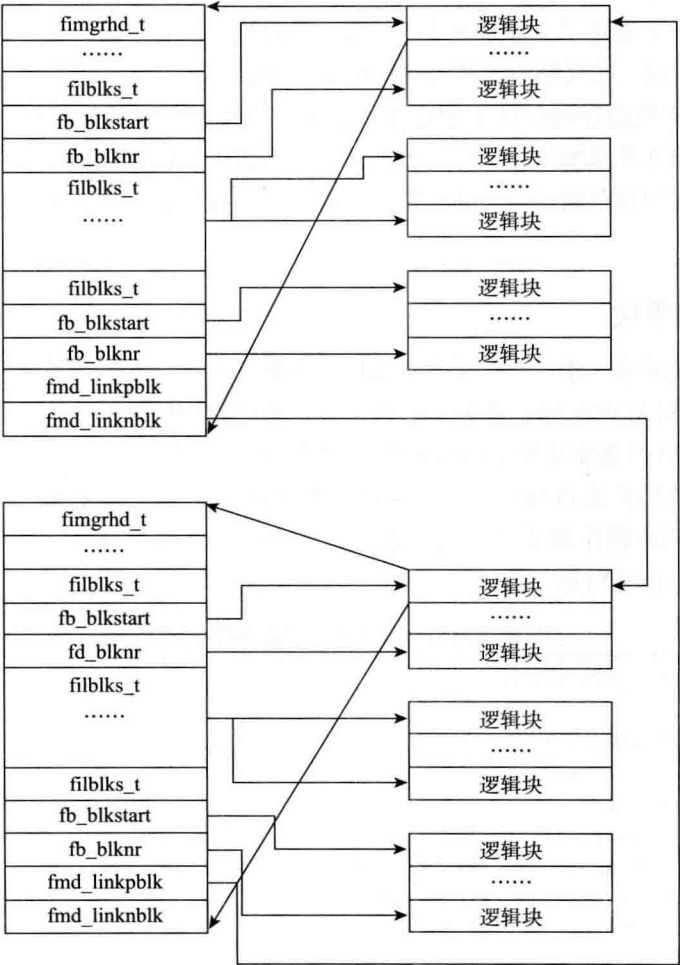


图 11-7 文件管理头结构

图 11-7 清楚地展示了 `fimgrhd_t` 结构如何管理一个文件占有的所有逻辑存储块，并且在 `fmd_fleblk` 数组元素用完时，可以通过类似链表的形式动态增加 `fimgrhd_t` 结构。事实上就是在动态增加文件的逻辑存储块。同时不难发现，文件的第一个逻辑存储块的首个 512B 空间中，存放的就是 `fimgrhd_t` 数据结构。

好了，一个简单的文件系统所需要的所有数据结构就设计完成了，剩下的就是怎么写代码实现这个文件系统了。

11.3 文件系统格式化

经常听说格式化硬盘、格式化 U 盘，并且同时也意味着被格式化的设备中的文件数据会被全部清空。事实上，格式化操作通常情况下，并不是把存储介质上所有的空间都清零，而是在这个存储介质上重建了文件系统用于管理文件的那一整套数据结构。这也是为什么格式化后的设备还能通过一些反删除软件找回一些文件的原因。

在一个未被使用的存储介质上创建文件系统，事实上也就是执行格式化操作。既然格式化操作就是重建文件系统管理文件的数据结构，那么这里也将从建立文件系统的超级块开始，然后建立用于管理存储介质空间的位图，最后建立根目录，以至于在存储介质之上完成文件系统的创建。

11.3.1 建立超级块

创建文件系统的第一步，当然是建立文件系统的超级块。不过，按照前面的设计，文件系统并不是 LMOSEM 内核的一部分，它只是 LMOSEM 内核下的一个设备。既然是设备那就要遵守 LMOSEM 内核驱动模型的相关要求，编写相应的设备驱动程序。

由上所述，我们首先得编写文件系统设备的驱动程序。由于前面已经写过几个设备的驱动程序了，对驱动程序框架已经很熟悉了，这里我们就简单点，如代码清单 11-4[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-4 文件系统驱动程序框架

```
typedef struct s_RFSDEVEXT
{
    spinlock_t rde_lock; //自旋锁
    list_h_t rde_list;   //链表
    uint_t rde_flg;      //标志
    uint_t rde_stus;     //状态
    void* rde_mstart;    //用于模拟存储介质的内存块的开始地址
    size_t rde_msize;    //内存块的大小
    void* rde_ext;       //扩展所用
}rfsdevext_t;

void rfsdevext_t_init(rfsdevext_t* initp)
```

```

.....
drvstus_t new_rfsdevext_mmblok(device_t* devp,size_t blksz)
{
    //分配模拟存储介质的内存空间,大小为4MB
    adr_t blkp= krlnew(blksz);
    //分配rfsdevext_t结构实例变量的内存空间
    rfsdevext_t* rfsexp=(rfsdevext_t*)krlnew(sizeof(rfsdevext_t));
    if(blkp==NULL||rfsexp==NULL)
    {
        return DFCERRSTUS;
    }
    //初始化rfsdevext_t结构
    rfsdevext_t_init(rfsexp);
    rfsexp->rde_mstart=(void*)blkp;
    rfsexp->rde_msize=blksz;
    //把rfsdevext_t结构的地址放入device_t 结构的dev_extdata域中,这里dev_extdata域就起作用了
    devp->dev_extdata=(void*)rfsexp;
    return DFCOKSTUS;
}

.....

void rfs_set_device(device_t* devp,driver_t* drvp)
{
    .....
    //设备类型为文件系统类型
    devp->dev_id.dev_mtype=FILESYS_DEVICE;
    devp->dev_id.dev_stype=0;
    devp->dev_id.dev_nr=0;
    //设备名称为rfs
    devp->dev_name="rfs";
    return;
}

drvstus_t rfs_entry(driver_t* drvp,uint_t val,void* p)
{
    .....
    //分配device_t结构并对其进行初级初始化
    device_t* devp=new_device_dsc();
    .....
    rfs_set_driver(drvp);
    rfs_set_device(devp,drvp);
    //分配模拟存储介质的内存空间
    if(new_rfsdevext_mmblok(devp,FSMM_BLK)==DFCERRSTUS)
    {
        .....
    }
    //把设备加入到驱动程序之中
    if(krldev_add_driver(devp,drvp)==DFCERRSTUS)

```

```

    {
        .....
    }
//向LMOSEM内核注册设备
    if(krlnew_device(devp)==DFCERRSTUS)
    {
        .....
    }
//初始化rfs文件系统, 这个函数我们后面会介绍
    init_rfs(devp);
    return DFCOKSTUS;
}
drvstus_t rfs_exit(driver_t* drv, uint_t val, void* p)
.....
drvstus_t rfs_open(device_t* devp, void* iopack)
.....
drvstus_t rfs_close(device_t* devp, void* iopack)
.....
drvstus_t rfs_read(device_t* devp, void* iopack)
.....
drvstus_t rfs_write(device_t* devp, void* iopack)
.....
drvstus_t rfs_lseek(device_t* devp, void* iopack)
.....
drvstus_t rfs_ioctl(device_t* devp, void* iopack)
.....
drvstus_t rfs_dev_start(device_t* devp, void* iopack)
.....
drvstus_t rfs_dev_stop(device_t* devp, void* iopack)
.....
drvstus_t rfs_set_powerstus(device_t* devp, void* iopack)
.....
drvstus_t rfs_enum_dev(device_t* devp, void* iopack)
.....
drvstus_t rfs_flush(device_t* devp, void* iopack)
.....
drvstus_t rfs_shutdown(device_t* devp, void* iopack)
.....

```

代码清单 11-4 中, 有很多函数的代码都省略了, 一是为了节约篇幅, 二是我们对这些框架性的代码早已了然于心。但是文件系统驱动程序与其他设备驱动程序不同的是, 它分配了一个扩展的数据结构, 将其地址放在 `device_t` 结构的扩展指针里, 该扩展数据结构里保存着用于模拟存储介质的内存空间的首地址和大小。该动作是 `new_rfsdevext_mmbk` 函数完成的。这里也可以看到, 我们驱动模型的设计给予了驱动程序的扩展性。最后不要忘了把 `rfs_entry` 函数加入到 `osdrvetytbl` 数组中。

编写好了文件系统设备驱动程序的框架代码，接下来就是要实现初始化文件系统的第一步，建立文件系统的超级块。建立超级块其实非常简单，就是初始化超级块的数据结构，然后把它写入到存储介质中的第一块逻辑存储块中就行了，如代码清单 11-5[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-5 创建文件系统超级块

```
//初始化rfsdir_t结构
void rfsdir_t_init(rfsdir_t* initp)
{
    initp->rdr_stus=0;
//初始化时，目录类型为空
    initp->rdr_type=RDR_NUL_TYPE;
//初始化时，块号为0
    initp->rdr_blknr=0;
//初始化时，名称数组中的每个元素都为0
    for(uint_t i=0;i<DR_NM_MAX;i++)
        initp->rdr_name[i]=0;
    return;
}
//初始化超级块数据结构
void rfssublk_t_init(rfssublk_t* initp)
{
    hal_spinlock_init(&initp->rsb_lock);
//标志就是一个数字而已，无其他意义
    initp->rsb_mgic=0x142422;
//文件系统版本为1
    initp->rsb_vec=1;
    initp->rsb_flg=0;
    initp->rsb_stus=0;
//超级块本身的大小
    initp->rsb_sz=sizeof(rfssublk_t);
//超级块占用多少个逻辑存储块
    initp->rsb_sblksz=1;
//逻辑存储块的大小为4KB
    initp->rsb_dblksz=FSYS_ALCBLKSZ;
//位图块从第1个逻辑存储块开始，超级块占用第0个逻辑存储块
    initp->rsb_bmpbks=1;
    initp->rsb_bmpbknr=0;
    initp->rsb_fsysallblk=0;
//初始化根目录
    rfsdir_t_init(&initp->rsb_rootdir);
    return;
}
//返回设备扩展数据结构
rfsdevext_t* ret_rfsdevext(device_t* devp)
```



```

{
    return (rfsdevext_t*)devp->dev_extdata;
}
//根据块号返回存储介质的块地址
void* ret_rfsdevblk(device_t* devp, uint_t blknr)
{
    rfsdevext_t* rfsexp=ret_rfsdevext(devp);
    //块号乘以块大小加上开始地址（用于模拟存储介质的内存空间的开始地址）
    void* blkp=rfsexp->rde_mstart+(blknr*FSYS_ALCBLKSZ);
    //如果该地址没有落在存储介质的空间中，就返回NULL表示出错
    if (blkp>=(void*)((size_t)rfsexp->rde_mstart+rfsexp->rde_msize))
        return NULL;
    //返回块地址
    return blkp;
}
//返回存储介质有多少个逻辑存储块
uint_t ret_rfsdevmaxblknr(device_t *devp)
{
    rfsdevext_t* rfsexp=ret_rfsdevext(devp);
    return (uint_t)((size_t)rfsexp->rde_msize)/FSYS_ALCBLKSZ;
}
//把4KB大小的缓冲区中的内容，写入到存储介质的某个逻辑存储块中
drvstus_t write_rfsdevblk(device_t* devp, void* weadr, uint_t blknr)
{
    //返回存储介质中第blknr块的逻辑存储块的地址
    void* p=ret_rfsdevblk(devp, blknr);
    if (p==NULL)
        return DFCERRSTUS;
    //复制数据到逻辑存储块中
    hal_memcpy(weadr, p, FSYS_ALCBLKSZ);
    return DFCOKSTUS;
}
//分配缓冲区
void* new_buf(size_t bufisz)
{
    //分配bufisz大小的内存空间
    return (void*)krlnew(bufisz);
}
//释放缓冲区
void del_buf(void* buf, size_t bufisz)
{
    //释放内存空间失败时就死机
    if (krldelete((adr_t)buf, bufisz)==FALSE)
        hal_sysdie("del buf err");
    return;
}

```

```

//创建超级块
bool_t create_superblk(device_t* devp)
{
    //分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
        return FALSE;
    //缓冲区清零
    hal_memset(buf,FSYS_ALCBLKSZ,0);
    //使rfssublk_t结构的指针指向缓冲区
    rfssublk_t* sbp=(rfssublk_t*)buf;
    //初始化超级块
    rfssublk_t_init(sbp);
    //获取存储介质的逻辑存储块数并保存到超级块中
    sbp->rsb_fsysallblk=ret_rfsdevmaxblknr(devp);
    //把缓冲区中的数据（即超级块）写入到存储介质的第0个逻辑存储块中
    if(write_rfsdevblk(devp,buf,0)==DFCERRSTUS)
        return FALSE;
    del_buf(buf,FSYS_ALCBLKSZ);
    return TRUE;
}

```

代码清单 11-5 中，创建文件系统超级块的函数是 `create_superblk`。这个函数本身很短，但它调用了大量的辅助函数，如分配、释放缓冲区，初始化超级块数据结构，获取设备扩展数据结构，以及返回存储介质的逻辑存储块数。还有根据块号返回逻辑存储块的地址，以及把缓冲区中的数据写入到具体逻辑存储块中的函数。有这些函数也是为了让用内存空间模拟存储介质变得更真实一些。其实，`rfssublk_t_init` 函数对 `rfssublk_t` 结构已经做了很多了，所以在 `create_superblk` 函数中只是改写了它的 `rsb_fsysallblk` 域，收集了存储介质的逻辑存储块的总数，最后把超级块写入到存储介质的第 0 个逻辑存储块中就完事了。

本小节写了不少代码，其实大部分是框架代码和辅助代码，并且都很简单，但是建立了超级块，并没有完成文件系统的创建。下面就去实现建立位图的代码。

11.3.2 建立位图

按照前面文件系统的设计，存储介质被分成了许多同等大小的逻辑存储块。可是如何管理这些逻辑存储块？位图就是为了解决这个问题。有了它就能准确地知道存储介质中哪些逻辑存储块是空闲的或者是被占用的。

按照前面的设计，我们使用一块存储空间中所有的字节来管理逻辑存储块的状态。建立位图无非就是把存储介质中的位图块清零。因为开始文件系统刚创建时，所有的逻辑存储块都是空闲的。如代码清单 11-6 [xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-6 建立文件系统位图

```

//把逻辑存储块中的数据,读取到4KB大小的缓冲区中
drvstus_t read_rfsdevblk(device_t* devp,void* rdadr,uint_t blknr)
{
    //获取逻辑存储块地址
    void* p=ret_rfsdevblk(devp,blknr);
    if(p==NULL)
        return DFCERRSTUS;
    //把逻辑存储块中的数据复制到缓冲区中
    hal_memcpy(p,rdadr,FSYS_ALCBLKSZ);
    return DFCOKSTUS;
}
//获取超级块
rfssublk_t* get_superblk(device_t* devp)
{
    //分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
        return NULL;
    //清零缓冲区
    hal_memset(buf,FSYS_ALCBLKSZ,0);
    //读取第0个逻辑存储块中的数据到缓冲区中,如果读取失败则释放缓冲区
    if(read_rfsdevblk(devp,buf,0)==DFCERRSTUS)
    {
        del_buf(buf,FSYS_ALCBLKSZ);
        return NULL;
    }
    //返回超级块数据结构的地址,即缓冲区的首地址
    return (rfssublk_t*)buf;
}
//释放超级块
void del_superblk(device_t* devp,rfssublk_t* sbp)
{
    //回写超级块,因为超级块中的数据可能已经发生了改变,如果出错则死机
    if(write_rfsdevblk(devp,(void*)sbp,0)==DFCERRSTUS)
    {
        hal_sysdie("del superblk err");
    }
    //释放先前分配的4KB大小的缓冲区
    del_buf((void*)sbp,FSYS_ALCBLKSZ);
    return;
}
//建立位图
bool_t create_bitmap(device_t* devp)
{
    bool_t rets=FALSE;

```

```

//获取超级块,失败则返回FALSE
    rfssubblk_t* sbp=get_superblk(devp);
    if(sbp==NULL)
        return FALSE;
    //分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
        return FALSE;
    //获取超级块中位图块的开始块号
    uint_t bitmapblk=sbp->rsb_bmpbks;
//获取超级块中存储介质的逻辑存储块总数
    uint_t devmaxblk=sbp->rsb_fsysallblk;
//如果逻辑存储块总数大于4096,就认为出错了
    if(devmaxblk>FSYS_ALCBLKSZ)
    {
        rets=FALSE;
        goto errlable;
    }
//把缓冲区中每个字节都置成1
    hal_memset(buf,FSYS_ALCBLKSZ,1);
    u8_t* bitmap=(u8_t*)buf;
//把缓冲区中的第3个字节到第devmaxblk个字节都置成0
    for(uint_t bi=2;bi<devmaxblk;bi++)
    {
        bitmap[bi]=0;
    }
//把缓冲区中的数据写入到存储介质中的第bitmapblk个逻辑存储块中,即位图块中
    if(write_rfsdevblk(devp,buf,bitmapblk)==DFCERRSTUS)
    {
        rets=FALSE;
        goto errlable;
    }
//设置返回状态
    rets=TRUE;
errlable:
    //释放超级块
    del_superblk(devp,sbp);
//释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
    return rets;
}

```

又多了几个辅助函数,因为位图块的块号和存储介质的逻辑存储块总数都保存在超级块中,所以要实现获取、释放超级块的函数,还需要一个读取逻辑存储块的函数。写入逻辑存储块的函数前面已经写过了。因为第0块是超级块,第1块是位图块本身,所以代码从缓冲区中的第3个字节开始清零,一直到 devmaxblk 个字节。devmaxblk 就是存储介质

的逻辑存储块总数。缓冲区中有 4096 个字节，但 devmaxblk 肯定是小于 4096 的，所以 devmaxblk 后面的字节全部为 1，就不会影响到后面分配逻辑存储块代码的正确性。最后把这个缓冲区中的数据写入到存储介质中的第 bitmapblk 个逻辑存储块中，就完成了位图的建立。

建立好了管理逻辑存储块状态的位图，下面就去接着建立根目录吧。

11.3.3 建立根目录

一切目录和文件都是存放在根目录下的，查询目录和文件也是从这里开始的，所以文件系统创建的最后一步就是创建根目录。

根目录也是一种文件，所以要为其分配相应的逻辑存储块，因为根目录下的文件和目录对应的 rfsdir_t 结构，就是保存在这个逻辑存储块中的，因为根目录是文件，所以要在这个逻辑存储块的首个 512B 空间中建立 fimgrhd_t 结构，即文件管理头数据结构，最后把这个逻辑存储块的块号存储在超级块中的 rfsdir_t 结构中，同时修改该 rfsdir_t 结构中的文件名称为“/”。下面就来写代码实现这些功能，如代码清单 11-7[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-7 建立文件系统根目录

```
//初始化filblks_t结构
void filblks_t_init(filblks_t* initp)
{
    //初始化时，应该为0
    initp->fb_blkstart=0;
    initp->fb_blknr=0;
    return;
}
//初始化fimgrhd_t结构
void fimgrhd_t_init(fimgrhd_t* initp)
{
    initp->fmd_stus=0;
    //空类型
    initp->fmd_type=FMD_NUL_TYPE;
    .....
    //从512B开始，前面的空间中保存着fimgrhd_t结构
    initp->fmd_fileifstbkoff=0x200;
    .....
    //初始化所有的filblks_t结构
    for(uint_t i=0;i<FBLKS_MAX;i++)
    {
        filblks_t_init(&initp->fmd_fleblk[i]);
    }
    //fimgrhd_t结构的链接域，都初始化为0
    initp->fmd_linkpblk=0;
    initp->fmd_linknblk=0;
```

```

        return;
    }
    //获取位图块
    u8_t* get_bitmapblk(device_t* devp)
    {
        //获取超级块
        rfssubblk_t* sbp=get_superblk(devp);
        if(sbp==NULL)
            return NULL;
        //分配4KB大小的缓冲区
        void* buf=new_buf(FSYS_ALCBLKSZ);
        if(buf==NULL)
            return NULL;
        //缓冲区清零
        hal_memset(buf,FSYS_ALCBLKSZ,0);
        //读取sbp->rsb_bmpbks块(位图块),到缓冲区中
        if(read_rfsdevblk(devp,buf,sbp->rsb_bmpbks)==DFCERRSTUS)
        {
            //如果读取失败,就要释放缓存区和超级块
            del_buf(buf,FSYS_ALCBLKSZ);
            del_superblk(devp,sbp);
            return NULL;
        }
        //释放超级块
        del_superblk(devp,sbp);
        //返回缓冲区的首地址
        return (u8_t*)buf;
    }
    //释放位图块
    void del_bitmapblk(device_t* devp,u8_t* bitmap)
    {
        //获取超级块
        rfssubblk_t* sbp=get_superblk(devp);
        if(sbp==NULL)
        {
            hal_sysdie("del superblk err");
            return;
        }
        //回写位图块,因为位图块中的数据可能已经发生改变
        if(write_rfsdevblk(devp,(void*)bitmap,sbp->rsb_bmpbks)==DFCERRSTUS)
        {
            del_superblk(devp,sbp);
            hal_sysdie("del superblk err1");
        }
        //释放超级块和存放位图块的缓冲区
        del_superblk(devp,sbp);
    }

```

```

    del_buf((void*)bitmap,FSYS_ALCBLKSZ);
    return;
}
//分配新的空闲逻辑存储块
uint_t rfs_new_blk(device_t* devp)
{
    uint_t retblk=0;
//获取位图块
    u8_t* bitmap=get_bitmapblk(devp);
    if(bitmap==NULL)
        return 0;
    for(uint_t blknr=2;blknr<FSYS_ALCBLKSZ;blknr++)
    {
        //找到一个为0的字节就置为1,并返回该字节对应的空闲块号
        if(bitmap[blknr]==0)
        {
            bitmap[blknr]=1;
            retblk=blknr;
            goto ret1;
        }
    }
//如果到这里就说明没有空闲块了,所以返回0
    retblk=0;
ret1:
    //释放位图块
    del_bitmapblk(devp,bitmap);
    return retblk;
}
//建立根目录
bool_t create_rootdir(device_t* devp)
{
    bool_t rets=FALSE;
//获取超级块
    rfssublk_t* sbp=get_superblk(devp);
    if(sbp==NULL)
    {
        return FALSE;
    }
//分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
    {
        rets=FALSE;
        goto errlable1;
    }
//缓冲区清零

```

```

    hal_memset(buf,FSYS_ALCBLKSZ,0);
//分配一个空闲的逻辑存储块
    uint_t blk=rfs_new_blk(devp);
    if(blk==0)
    {
        rets=FALSE;
        goto errlable;
    }
//设置超级块中的rfsdir_t结构中的名称为 "/"
    sbp->rsb_rootdir.rdr_name[0]='/';
//设置超级块中的rfsdir_t结构中的类型为目录类型
    sbp->rsb_rootdir.rdr_type=RDR_DIR_TYPE;
//设置超级块中的rfsdir_t结构中的块号为新分配的空闲逻辑存储块的块号
    sbp->rsb_rootdir.rdr_blknr=blk;
    fimgrhd_t* fmp=(fimgrhd_t*)buf;
//初始化fimgrhd_t结构
    fimgrhd_t_init(fmp);
//因为这是目录文件，所以fimgrhd_t结构的类型设置为目录类型
    fmp->fmd_type=FMD_DIR_TYPE;
//fimgrhd_t结构自身所在的块设置为新分配的空闲逻辑存储块
    fmp->fmd_sfbk=blk;
//fimgrhd_t结构中正在写入的块设置为新分配的空闲逻辑存储块
    fmp->fmd_curfwritebk=blk;
//fimgrhd_t结构中正在写入的块的偏移设置为512B
    fmp->fmd_curfinwbkoff=0x200;
//设置文件数据占有块数组的第0个元素
    fmp->fmd_fleblk[0].fb_blkstart=blk;
    fmp->fmd_fleblk[0].fb_blknr=1;
//把缓冲区中的数据写入到新分配的空闲逻辑存储块中，其中包含已经设置好的fimgrhd_t结构
    if(write_rfsdevblk(devp,buf,blk)==DFCERRSTUS)
    {
        rets=FALSE;
        goto errlable;
    }
    rets=TRUE;
errlable:
//释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
errlable1:
//释放超级块
    del_superblk(devp,sbp);
    return rets;
}

```

又多了许多辅助函数，因为根目录也是文件，需要为其分配空闲的逻辑存储块，所以有了 `rfs_new_blk` 函数，要分配一个空闲的逻辑存储块，就需要操作文件系统的位图块，所以

又有了 `get_bitmapblk`、`del_bitmapblk` 函数用于获取和释放位图块。还有用于初始化 `fimgrhd_t` 数据结构的 `fimgrhd_t_init` 函数。之所以花大量篇幅把代码都列出来，是为了方便读者了解整个过程的由来，加上代码中的注释，理解整个过程应该不是很难。需要明白的是每个文件都有一个 `fimgrhd_t` 结构，并且它是放在文件占用的第一个逻辑存储块的头 512B 空间中的，所以文件数据都是从文件占用的第一个逻辑存储块的 512B 处开始的。

最后我们还是来写一个 `rfs_fmat` 函数，把建立超级块、建立位图、建立根目录的三个操作包装起来，顺便把 `init_rfs` 函数也实现了，让它调用 `rfs_fmat` 函数，但是它本身会在 `rfs_entry` 函数的最后被调用，如代码清单 11-8[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-8 文件系统格式化

```
void init_rfs(device_t* devp)
{
    //文件系统格式化
    rfs_fmat(devp);
    return;
}

void rfs_fmat(device_t* devp)
{
    //建立超级块
    if(create_superblk(devp)==FALSE)
    {
        hal_sysdie("create superblk err");
    }
    //建立位图
    if(create_bitmap(devp)==FALSE)
    {
        hal_sysdie("create bitmap err");
    }
    //建立根目录
    if(create_rootdir(devp)==FALSE)
    {
        hal_sysdie("create rootdir err");
    }
    return;
}
```

到这里，我们实现了建立超级块、建立位图、建立根目录的代码，我们这个小型的文件系统的格式化操作就完成了。但对于整个文件系统来说，这才完成了一小部分，还需要做更多的事情，下面继续。

11.4 文件系统基础操作

从前面的文件系统格式化操作中不难发现，文件系统格式化操作的三个函数并不复杂，

但是它们需要大量的辅助函数，这使得在单一的问题下代码会很多，以至于无法突出重点。

同样，完成文件相关的操作也需要大量的辅助函数，这些辅助函数称为文件系统的基础操作函数。为了避免发生前面那种长篇代码的情况，这里将用一节来介绍这些文件系统的基础操作函数。文件操作，肯定是要读写根目录文件的，所以获取与释放根目录文件的函数是首先要实现的，接着写几个小的字符串操作函数，然后是分解文件路径名的函数，最后是检查一个文件是否存在于存储介质之上的函数。

实现了上面这些文件系统的基础操作函数后，再去实现上层的文件操作就容易多了。

11.4.1 获取与释放根目录文件

新建、删除、打开一个文件，首先都要找到与该文件对应的 `rfmdir_t` 结构。在我们的文件系统中，一个文件的 `rfmdir_t` 结构就存储在根目录文件中，所以要读取文件对应的 `rfmdir_t` 结构，首先就要获取根目录文件。

获取根目录文件的方法很简单，根据超级块中的 `rfmdir_t` 结构中的信息，读取根目录文件的逻辑存储块就行了。明白了方法就可以写代码了，如代码清单 11-9[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-9 获取根目录文件

```
//获取根目录文件的rfmdir_t结构
rfmdir_t* get_rootdir(device_t* devp)
{
    rfmdir_t* retptr=NULL;
//获取超级块
    rfssublk_t* sbp=get_superblk(devp);
    if(sbp==NULL)
    {
        return NULL;
    }
//分配一个rfmdir_t结构大小的缓冲区
    void* buf=new_buf(sizeof(rfmdir_t));
    if(buf==NULL)
    {
        retptr=NULL;
        goto err1;
    }
//把超级块中的rfmdir_t结构中的信息复制到缓冲区中
    hal_memcpy((void*)(&sbp->rsb_rootdir),buf,sizeof(rfmdir_t));
//设置缓冲区的首地址为返回地址
    retptr=(rfmdir_t*)buf;
err1:
    //释放超级块
    del_superblk(devp,sbp);
```

```

        return retptr;
    }
    //释放根目录文件的rfmdir_t结构
void del_rootdir(device_t* devp, rfmdir_t* rdir)
{
    //释放rfmdir_t结构大小的缓冲区
    del_buf((void*)rdir, sizeof(rfmdir_t));
    return;
}
//获取根目录文件
void* get_rootdirfile_blk(device_t* devp)
{
    void* retptr=NULL;
    //获取根目录文件的rfmdir_t结构
    rfmdir_t* rtdir=get_rootdir(devp);
    if(rtdir==NULL)
    {
        return NULL;
    }
    //分配4KB大小的缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
    {
        retptr=NULL;
        goto err1;
    }
    //清零缓冲区
    hal_memset(buf, FSYS_ALCBLKSZ, 0);
    //读取根目录文件的逻辑存储块到缓冲区中
    if(read_rfsdevblk(devp, buf, rtdir->rdr_blknr)==DFCERRSTUS)
    {
        retptr=NULL;
        goto err1;
    }
    //设置缓冲区的首地址为返回值
    retptr=buf;
    goto err1;
err1:
    del_buf(buf, FSYS_ALCBLKSZ);
err1:
    //释放根目录文件的rfmdir_t结构
    del_rootdir(devp, rtdir);
    return retptr;
}

```

get_rootdirfile_blk 函数非常简单，就是把根目录文件数据所在的逻辑存储块读取到一个

缓冲区中，然后返回该缓冲区的首地址。不过这个逻辑存储块的块号，保存在超级块中的 `rfsdir_t` 结构中，所以多了两个辅助函数，即 `get_rootdir` 和 `del_rootdir`，它们用于从超级块中获取和释放根目录文件的 `rfsdir_t` 结构，虽然直接获取超级块也能达到目的，但是不如这两个函数来得直观。

好了，获取根目录文件的函数就写好了，但是还需要有释放根目录文件的函数。这个释放的同时也意味着回写和删除其占用的缓冲区。因为根目录文件的数据可能已经发生了改变，所以一定要把它回写到存储介质中去，如代码清单 11-10[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-10 释放根目录文件

```
//释放根目录文件
void del_rootdirfile_blk(device_t* devp,void* blkp)
{
    //因为逻辑存储块的头512B的空间中，保存的就是fimg_rhd_t结构
    fimg_rhd_t* fmp=(fimg_rhd_t*)blkp;
    //把根目录文件回写到存储介质中去，块号为fimg_rhd_t结构自身所在的块号，因为限制每个文件只分配一个
    //逻辑存储块，所以这是可行的
    if (write_rfsdevblk(devp,blkp,fmp->fmd_sfbk) != DFCERRSTUS)
    {
        hal_sysdie("del_rootfile_blk err");
    }
}

//释放缓冲区
del_buf(blkp,FSYS_ALCBLKSZ);
return;
}
```

代码清单 11-10 很简单，这也是因为前面我们对文件系统的限制，即限制每个文件只能分配一个逻辑存储块，回写时才会如此轻松，其中的注释已经很明显了，不必多说。

获取与释放根目录文件的代码就写好了，究竟获取它之后要做什么，这里不用管，下面还要实现更多的基础操作函数呢。

11.4.2 字符串操作

一个文件都具有一个响亮的文件名，文件名其实是一个字符串，是字符串就免不了要对其进行比较、求长度、复制等相关操作。

在写这些操作字符串的函数之前，先来了解一下 C 语言编译器是怎么处理字符串的。C 语言编译器把代码中所有双引号中的一系列字符，都放在一个统一的地方，并且每个字符串的末尾加一个 0 表示该字符串的结束，然后我们使用时需要用一个字符指针去引用它，最后把所有可链接的二进制程序文件链接在一起形成可执行程序文件时，在这个可执行程序文件中就会有一个字符串节或者字符串段，它里面保存着所有程序中会引用到的字符串，并且每

个字符串的末尾都加一个 0。

下面就来写一写字符串的比较、求长度、复制这三个操作的函数，如代码清单 11-11[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-11 字符串操作的相关函数

```
//比较字符串
sint_t rfs_strcmp(char_t* str_s,char_t* str_d)
{
    for(;;)
    {
        //依次比较str_s和str_d所指向的字符串中的每个字符，若不相等就返回0
        if(*str_s!=*str_d)
        {
            return 0;
        }
        //如果字符串到达末尾了，就跳出循环
        if(*str_s==0)
        {
            break;
        }
        //同时向前移动指针
        str_s++;
        str_d++;
    }
    //返回1表示字符串相同
    return 1;
}

//获取字符串长度
sint_t rfs_strlen(char* str_s)
{
    sint_t chaidx=0;
    //不为0就循环，每循环一次指针前移，计数变量chaidx自增
    while(*str_s!=0)
    {
        str_s++;
        chaidx++;
    }
    //最后返回计数值，就是字符串的长度，也是字符串中字符的个数
    return chaidx;
}

//复制字符串，同时返回复制字符的个数
sint_t rfs_strcpy(char_t* str_s,char_t* str_d)
{
    sint_t chaidx=0;
    //不为0就循环
```

```

while(*str_s!=0)
{
    //复制一个字符
    *str_d=*str_s;
    //两个指针同时前移
    str_s++;
    str_d++;
    //计数变量自增
    chaidx++;
}
//把结束符(0)也复制进去
*str_d=*str_s;
//返回计数变量
return chaidx;
}

```

明白了 C 语言编译器如何处理字符串之后，加上代码清单 11-11 中的注释，理解上面三个函数应该不难，所以这里就不过多介绍了。

11.4.3 分解路径名

经常看到 “/lmosem/drivers/drvrfs.c” 这样的文件路径名。其中的 “/” 符号，除第一个 “/” 是根目录外，其他 “/” 只是一个路径分隔符。然而，在很多情况中经常需要把路径分隔符去除，提取其中的目录名称或者文件名称。由于前面文件系统的限制，我们的文件路径名，只会是 “/xxxx” 这种类型的，所以问题会简单许多。

下面就来实现去除路径分隔符提取文件名称的代码，如代码清单 11-12 [xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-12 分解文件路径名

```

//检查文件路径名
sint_t rfs_chkfilepath(char_t* fname)
{
    char_t* chp=fname;
    //检查文件路径名的第一个字符是否为 “/”，不是则返回2
    if(chp[0]!='/')
    {
        return 2;
    }
    for(uint_t i=1;;i++)
    {
        //检查除第1个字符外其他字符中还有没有为 “/” 的，有就返回3
        if(chp[i]=='/')
        {
            return 3;
        }
    }
}

```

```

    }
    //如果这里i大于等于文件名称的最大长度，就返回4
    if(i>=DR_NM_MAX)
    {
        return 4;
    }
    //到文件路径字符串的末尾就跳出循环
    if(chp[i]==0&& i>1)
    {
        break;
    }
}
//返回0表示正确
return 0;
}
//提取纯文件名
sint_t rfs_ret_fname(char_t* buf, char_t* fpath)
{
    //检查存放纯文件名的缓冲区指针和文件路径名指针是否合法
    if(buf==NULL||fpath==NULL)
    {
        return 6;
    }
    //检查文件路径名是不是“/xxxx”的形式
    sint_t stus=rfs_chkfilepath(fpath);
    //如果不为0就直接返回这个状态值表示错误
    if(stus!=0)
    {
        return stus;
    }
    //从路径名字符串的第2个字符开始复制字符到buf中
    rfs_strcpy(&fpath[1],buf);
    return 0;
}

```

可以看到，完成分解文件路径名的是 `rfs_ret_fname` 函数。这个函数把 `fpath` 指向的路径名中的文件名提取出来，放到 `buf` 指向的缓冲区中，提取动作是前面刚刚写过的 `rfs_strcpy` 函数。因为我们的文件路径名都是“/xxxx”的形式，才能用 `rfs_strcpy` 函数这么简单的方式，所以要在这之前先调用 `rfs_chkfilepath` 函数检查路径名是不是“/xxxx”的形式。这两个函数都比较简单，看注释就能明白。

提取文件名的函数也写好了，正是因为前面的限制才让代码变得简单，这样才有利于学习文件系统原理，接下来我们继续做后面的事情。

11.4.4 检查文件是否存在

在建立和删除文件时，第一件事就是应该检查新建或者删除的这个文件是不是已经存在于存储介质之上了。对于新建文件时，不能建立相同的文件，对于删除文件时，不能删除不存在的文件。

下面就来写代码完成这个功能。这个功能实现起来还是比较容易的，无非就是搜索一个目录文件中所有的 `rfsdir_t` 结构，比较其中的文件名。因为前面的限制，这里只要搜索根目录文件占用的第一块逻辑存储块就可以了，如代码清单 11-13[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-13 检查文件是否存在

```

sint_t rfs_chkfileisindex(device_t* devp, char_t* fname)
{
    sint_t rets=6;
    //获取文件名的长度，注意不是文件路径名
    sint_t ch=rfs_strlen(fname);
    //检查文件名的长度是不是合乎要求
    if(ch<1||ch>=(sint_t)DR_NM_MAX)
    {
        return 4;
    }
    //获取根目录文件，如果获取失败就返回2
    void* rdblkp=get_rootdirfile_blk(devp);
    if(rdblkp==NULL)
    {
        return 2;
    }
    fimgrhd_t* fmp=(fimgrhd_t*)rdblkp;
    //检查该fimgrhd_t结构的类型是不是FMD_DIR_TYPE，即这个文件是不是目录文件
    if(fmp->fmd_type!=FMD_DIR_TYPE)
    {
        rets=3;
        goto err;
    }
    //检查根目录文件是不是为空，即没有写入任何数据，所以返回0，表示根目录下没有对应的文件
    if(fmp->fmd_curfwritebk==fmp->fmd_fleblk[0].fb_blkstart&&fmp->fmd_curfinwbkoff==fmp->fmd_fileifstbkoff)
    {
        rets=0;
        goto err;
    }
    //指向根目录文件的第一个字节
    rfsdir_t* dirp=(rfsdir_t*)((uint_t)(fmp)+fmp->fmd_fileifstbkoff);
    //指向根目录文件的结束地址
    void* maxchkp=(void*)((uint_t)rdblkp+FSYS_ALCBLKSZ-1);
    //当前的rfsdir_t结构的指针比根目录文件的结束地址小，就继续循环

```



```

for(; (void*) dirp < maxchkp;)
{
    // 如果这个 rfsdir_t 结构的类型是 RDR_FIL_TYPE, 说明它对应的是文件而不是目录, 所以下面就继
    // 续比较其文件名
    if (dirp->rdr_type == RDR_FIL_TYPE)
    {
        // 比较其文件名
        if (rfs_strcmp(dirp->rdr_name, fname) == 1)
        {
            rets = 1;
            goto err;
        }
    }
    dirp++;
}
// 到了这里说明没有找到相同的文件
rets = 0;
err:
    // 释放根目录文件
    del_rootdirfile_blk(devp, rdblkp);
    return rets;
}

```

`rfs_chkfileisindex` 函数虽然有点长, 但代码逻辑还是很简单的。首先是检查文件名的长度, 接着获取了根目录文件, 然后就是搜索根目录文件中的所有 `rfsdir_t` 结构, 并比较其文件名是否相同, 有相同的就返回 1, 根目录文件为空或者没有相同的文件名就返回其他值, 最后释放了根目录文件。最难理解的可能就是 `dirp` 指针和 `maxchkp` 指针, 因为 `get_rootdirfile_blk` 函数已经把根目录文件读取到内存中了, 所以可以用指针操作其中的数据了。

好了, 检查一个文件是否存在于存储介质之上的函数就实现了。有了它, 建立和删除文件就方便多了。

11.5 文件操作

直到这里, 我们没有操作任何文件本身, 设计并实现文件系统, 就是为了让用户和应用程序更好地处理和存放文件的, 所以一个完整的文件系统必须要支持一些文件操作。

本节我们将依次实现新建文件、删除文件、打开文件、读写文件、关闭文件, 这几大文件操作, 这也是文件系统需要提供的最基本的功能, 对于文件本身来说, 这几大操作已经足够了。在实现这几大文件操作后, 还要看一看怎么将这些操作整合到文件系统设备驱动程序框架之中。

好了, 下面我们就来实现上述的文件操作。

11.5.1 新建文件

要操作文件，首先得有文件，在没有文件之前，任何对文件本身的操作都是徒劳无效的。这里首先就来实现新建文件这个功能。

在写代码之前，先来看一看建立一个文件的方法，如下：

- 1) 从文件路径名中提取出纯文件名。
- 2) 检查存储介质上是否已经存在这个文件。
- 3) 分配一个空闲的逻辑存储块。
- 4) 在根目录文件的末尾写入这个新建文件对应的 `rfsdir_t` 结构。
- 5) 在一个新的 4KB 大小的缓冲区中，初始化新建文件对应的 `fimg_rhd_t` 结构。
- 6) 最后将这个新的 4KB 大小的缓冲区中的数据，写入到先前分配的空闲逻辑存储块中。

下面用两个函数来实现上述 6 个步骤，如代码清单 11-14[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-14 新建文件

```
//新建文件的接口函数
drvstatus_t rfs_new_file(device_t* devp,char_t* fname,uint_t flg)
{
    //在栈中分配一个字符缓冲区
    char_t fne[DR_NM_MAX];
    //清零该缓冲区
    hal_memset((void*)fne,DR_NM_MAX,0);
    //从文件路径名中提取出纯文件名
    if(rfs_ret_fname(fne,fname)!=0)
    {
        return DFCERRSTUS;
    }
    //检查存储介质上是否已经存在这个新建的文件，如果是则返回错误
    if(rfs_chkfileisindex(devp,fne)!=0)
    {
        return DFCERRSTUS;
    }
    //调用实际建立文件的函数
    return rfs_new_dirfileblk(devp,fne,RDR_FIL_TYPE,0);
}

drvstatus_t rfs_new_dirfileblk(device_t* devp,char_t* fname,uint_t flgtype,uint_t val)
{
    drvstatus_t rets=DFCERRSTUS;
    //检查是不是建立文件
    if(flgtype!=RDR_FIL_TYPE)
    {
        return DFCERRSTUS;
    }
}
```

```

    }
//分配一个4KB大小的缓冲区
void* buf=new_buf(FSYS_ALCBLKSZ);
if (buf==NULL)
{
    return DFCERRSTUS;
}
//清零该缓冲区
hal_memset(buf,FSYS_ALCBLKSZ,0);
//分配一个新的空闲逻辑存储块
uint_t fblk=rfs_new_blk(devp);
if (fblk==0)
{
    rets=DFCERRSTUS;
    goto err1;
}
//获取根目录文件
void* rdirblk=get_rootdirfile_blk(devp);
if (rdirblk==NULL)
{
    rets=DFCERRSTUS;
    goto err1;
}
fimg_rhd_t* fmp=(fimg_rhd_t*)rdirblk;
//指向文件当前的写入地址, 因为根目录文件已经被读取到内存中了
rfsdir_t* wrdirp=(rfsdir_t*)((uint_t)rdirblk+fmp->fmd_curfinwbkoff);
//对文件当前的写入地址进行检查
if (((uint_t)wrdirp)>=((uint_t)rdirblk+FSYS_ALCBLKSZ))
{
    rets=DFCERRSTUS;
    goto err;
}
wrdirp->rdr_stus=0;
//设为文件类型
wrdirp->rdr_type=flgtype;
//设为刚刚分配的空闲逻辑存储块
wrdirp->rdr_blknr=fblk;
//把文件名复制到rfsdir_t结构中
rfs_strcpy(fname,wrdirp->rdr_name);
//增加根目录文件的大小
fmp->fmd_filesz+=(uint_t)(sizeof(rfsdir_t));
//增加根目录文件当前的写入地址, 保证下次不被覆盖, 因为写入的是rfsdir_t结构, 所以增加的是
//rfsdir_t结构的大小
fmp->fmd_curfinwbkoff+=(uint_t)(sizeof(rfsdir_t));
//指向新分配的缓冲区
fimg_rhd_t* ffmp=(fimg_rhd_t*)buf;

```

```

//调用fimg_rhd_t结构默认的初始化函数
    fimg_rhd_t_init(ffmp);
//因为建立的是文件，所以设为文件类型
    ffmp->fmd_type=FMD_FIL_TYPE;
//把自身所在的块，设为刚刚分配的空闲逻辑存储块
    ffmp->fmd_sfbk=fblk;
//把当前写入的块，设为刚刚分配的空闲逻辑存储块
    ffmp->fmd_curfwritebk=fblk;
//把当前写入块的写入偏移量设为512
    ffmp->fmd_curfinwbkoff=0x200;
//把文件存储块数组的第1个元素的开始块，设为刚刚分配的空闲逻辑存储块
    ffmp->fmd_fleblk[0].fb_blkstart=fblk;
//因为只分配了一个空闲的逻辑存储块，所以设为1
    ffmp->fmd_fleblk[0].fb_blknr=1;
//把缓冲区中的数据写入到刚刚分配的空闲逻辑存储块中
    if (write_rfsdevblk(devp,buf,fblk)==DFCERRSTUS)
    {
        rets=DFCERRSTUS;
        goto err;
    }
    rets=DFCOKSTUS;
err:
    //释放根目录文件
    del_rootdirfile_blk(devp,rdirblk);
err1:
    //释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
    return rets;
}

```

代码清单 11-14 很长，但是难点不多。首先在 `rfs_new_file` 函数中提取了文件名并检查了这个新建的文件是否已经存在于存储介质之上了。然后调用了 `rfs_new_dirfileblk` 函数。在这个函数中，关键有两点要理解，第一，前面反复提到的目录文件中存放的就是一系列的 `rfmdir_t` 结构；第二，就是 `fmp` 和 `ffmp` 这两个指针。`fmp` 指针指向的是根目录文件的 `fimg_rhd_t` 结构，因为要写入一个新的 `rfmdir_t` 结构，所以要获取并改写根目录文件的 `fimg_rhd_t` 结构中的数据。而 `ffmp` 指针指向的是新建文件的 `fimg_rhd_t` 结构，因为每个文件都必须有一个 `fimg_rhd_t` 结构，并且初始化了其中的一些数据。最后该函数把这个缓冲区中的数据写入到分配的空闲逻辑存储块中，同时释放了根目录文件和缓冲区。

至此，新建文件的操作就实现了，下面就去实现其他文件操作。

11.5.2 删除文件

当存储介质中的文件越来越多时，就需要把一些不需要或者不重要的文件删除，否则存

储介质的空间最终会耗尽，所以文件系统就必须提供删除文件的功能。

同样的先来看看删除一个文件的方法，如下：

1) 从文件路径名中提取纯文件名。

2) 获取根目录文件。

3) 从根目录文件中查找将要删除文件的 `rfsdir_t` 结构。

4) 释放该文件占用的逻辑存储块，并且初始化与该文件对应的 `rfsdir_t` 结构，然后设置对应的 `rfsdir_t` 结构的类型为 `RDR_DEL_TYPE`。

5) 释放根目录文件。

这次用三个函数来实现上述这些步骤，如代码清单 11-15[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-15 删除文件

```
//文件删除的接口函数
drvstatus_t rfs_del_file(device_t* devp,char_t* fname,uint_t flg)
{
    if (flg!=0)
    {
        return DFCERRSTUS;
    }
    drvstatus_t rets=rfs_del_dirfileblk(devp,fname,RDR_FIL_TYPE,0);
    return rets;
}
drvstatus_t rfs_del_dirfileblk(device_t* devp,char_t* fname,uint_t flgtype,uint_t val)
{
    if (flgtype!=RDR_FIL_TYPE||val!=0)
    {
        return DFCERRSTUS;
    }
    char_t fne[DR_NM_MAX];
    hal_memset((void*)fne,DR_NM_MAX,0);
    //提取纯文件名
    if (rfs_ret_fname(fne,fname)!=0)
    {
        return DFCERRSTUS;
    }
    //调用删除文件的核心函数
    if (del_dirfileblk_core(devp,fne)!=0)
    {
        return DFCERRSTUS;
    }
    return DFCOKSTUS;
}
//删除文件的核心函数
```

```

    sint_t del_dirfileblk_core(device_t* devp, char_t* fname)
{
    sint_t rets=6;
//获取根目录文件
    void* rblkp=get_rootdirfile_blk(devp);
    if(rblkp==NULL)
    {
        return 5;
    }
    fimgrhd_t* fmp=(fimgrhd_t*)rblkp;
//检查根目录文件的类型
    if(fmp->fmd_type!=FMD_DIR_TYPE)
    {
        rets=4;
        goto err;
    }
//检查根目录文件中有没有数据
    if(fmp->fmd_curfwritebk==fmp->fmd_fleblk[0].fb_blkstart&&fmp->fmd_curfinwbkoff==
        fmp->fmd_fileifstbkoff)
    {
        rets=3;
        goto err;
    }
    rfsdir_t* dirp=(rfsdir_t*)((uint_t)(fmp)+fmp->fmd_fileifstbkoff);
    void* maxchkp=(void*)((uint_t)rblkp+FSYS_ALCBLKSZ-1);
    for(; (void*)dirp<maxchkp;)
    {
//检查其类型是否为文件类型
        if(dirp->rdr_type==RDR_FIL_TYPE)
        {
            //如果文件名相同，就执行以下删除动作
            if(rfs_strcmp(dirp->rdr_name, fname)==1)
            {
                //释放rfsdir_t结构的rdr_blknr域中指向的逻辑存储块
                rfs_del_blk(devp, dirp->rdr_blknr);
                //初始化rfsdir_t结构，实际上是清除其中的数据
                rfsdir_t_init(dirp);
                //设置rfsdir_t结构的类型为删除类型，表示它已经删除
                dirp->rdr_type=RDR_DEL_TYPE;
                rets=0;
                goto err;
            }
        }
        dirp++; //下一个rfsdir_t
    }
    rets=1;
}

```

```

err:
    //释放根目录文件
    del_rootdirfile_blk(devp, rblkp);
    return rets;
}

```

代码清单 11-15 中有三个函数，但是做实际工作的是 `del_dirfileblk_core` 函数，它主要是用一个循环去扫描根目录文件中所有的 `rfmdir_t` 结构。每找到一个 `rfmdir_t` 结构就比较其文件名，看一看是不是和将要删除的文件的名称相同，相同时就释放该 `rfmdir_t` 结构的 `rdr_blknr` 域中指向的逻辑存储块，并清除该 `rfmdir_t` 结构中的数据，同时设置该 `rfmdir_t` 结构的类型为删除类型。而关于 `dirp` 和 `maxchkp` 指针的原理，我们前面已经讨论过了。

好了，删除一个文件，就是把这个文件对应的 `rfmdir_t` 结构中的数据清空，从而使之不能查找到该文件，同时也释放了该文件占用的逻辑存储块，删除文件的功能就实现了，下面继续实现其他的功能。

11.5.3 打开文件

对于一个已经存在于存储介质之上的文件，要对其进行读写操作，首先就应该打开这个文件。

在完成这个功能之前，先来回忆一下驱动模型中的 IO 包，即 `objnode_t` 数据结构。LMOSEM 内核上层组件调用设备驱动程序时，都需要建立一个相应的 IO 包，然后把这个 IO 包发送给相应的驱动程序，驱动程序就知道要做什么事情了，但是 IO 包不仅仅是用于驱动程序，它还用于表示进程使用了哪些资源，例如，打开了哪些设备或者文件，而每打开一个设备或者文件就建立一个 `objnode_t` 结构，放在特定进程的资源表中。为了适应文件系统设备驱动程序，需要把 `objnode_t` 结构中增加一些东西，如代码清单 11-16[xxxx/lmosem/include/knlinc/krlobjnode_t.h] 所示。

代码清单 11-16 IO 包数据结构

```

typedef struct s_OBJNODE
{
    spinlock_t    on_lock;
    list_h_t      on_list;
    sem_t         on_complesem;
    uint_t        on_flg;
    uint_t        on_stus;
    .....
    void*         on_fname;           //文件路径名指针
    void*         on_finode; //文件对应的fimg_rhd_t结构指针
    void*         on_extp;            //扩展所用
}objnode_t;
#define OBJN_TY_DEV 1                //设备类型

```

```
#define OBJN_TY_FIL 2           //文件类型
#define OBJN_TY_NUL 0          //默认类型
```

代码清单 11-16 给 `objnode_t` 结构增加了两个域，一个是指向文件路径名的指针，表示打开或者将要打开什么文件，另一个是指向已打开文件的 `fimgrhd_t` 结构的指针，因为要知道一个文件的所有信息，所以增加了对应文件的 `fimgrhd_t` 结构指针。

有了上面的回顾，现在来看看打开一个文件的方法，如下：

- 1) 检查 IO 包中的文件路径名。
- 2) 从文件路径名中提取纯文件名。
- 3) 获取根目录文件。
- 4) 在根目录文件中搜索该文件对应的 `rfsdir_t` 结构。
- 5) 分配一个 4KB 大小的缓冲区。
- 6) 读取该文件对应的 `rfsdir_t` 结构的 `rdr_blknr` 域中指向的逻辑存储块到缓冲区中。
- 7) 释放根目录文件。
- 8) 把缓冲区中的 `fimgrhd_t` 结构的地址保存到 IO 包的 `on_finode` 域中，然后返回。

下面将用两个函数来实现以上 8 个步骤，如代码清单 11-17[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-17 打开文件

```
//打开文件的接口函数
drvstus_t rfs_open_file(device_t* devp,void* iopack)
{
    objnode_t* obp=(objnode_t*)iopack;
    //检查IO包中的文件路径名
    if(obp->on_fname==NULL)
    {
        return DFCERRSTUS;
    }
    //调用打开文件的核心函数
    void* fmdp=rfs_openfileblk(devp,(char_t*)obp->on_fname);
    if(fmdp==NULL)
    {
        return DFCERRSTUS;
    }
    //把返回的fimgrhd_t结构的地址保存到IO包的on_finode域中
    obp->on_finode=fmdp;
    return DFCOKSTUS;
}
//打开文件的核心函数
void* rfs_openfileblk(device_t *devp, char_t* fname)
{
    char_t fne[DR_NM_MAX];
```



```

void* rets=NULL,*buf=NULL;
hal_memset((void*)fne,DR_NM_MAX,0);
//从文件路径名中提取纯文件名
if(rfs_ret_fname(fne,fname)!=0)
{
    return NULL;
}
//获取根目录文件
void* rblkp=get_rootdirfile_blk(devp);
if(rblkp==NULL)
{
    return NULL;
}
fimgrhd_t* fmp=(fimgrhd_t*)rblkp;
//判断根目录文件的类型是否合理
if(fmp->fmd_type!=FMD_DIR_TYPE)
{
    rets=NULL;
    goto err;
}
//判断根目录文件里有没有数据
if(fmp->fmd_curfwritebk==fmp->fmd_fleblk[0].fb_blkstart&&fmp->fmd_curfinwbkoff==
    fmp->fmd_fileifstbkoff)
{
    rets=NULL;
    goto err;
}
rfsdir_t* dirp=(rfsdir_t*)((uint_t)(fmp)+fmp->fmd_fileifstbkoff);
void* maxchkp=(void*)((uint_t)rblkp+FSYS_ALCBLKSZ-1);
//开始搜索该文件对应的rfsdir_t结构
for(;(void*)dirp<maxchkp;)
{
    if(dirp->rdr_type==RDR_FIL_TYPE)
    {
        //如果文件名相同就跳转到opfblk标号处运行
        if(rfs_strcmp(dirp->rdr_name,fne)==1)
        {
            goto opfblk;
        }
    }
    dirp++;
}
//如果到这里说明没有找到该文件对应的rfsdir_t结构,所以设置返回值为NULL
rets=NULL;
goto err;
opfblk:

```

```

//分配4KB大小的缓冲区
buf=new_buf(FSYS_ALCBLKSZ);
if (buf==NULL)
{
    rets=NULL;
    goto err;
}
//读取该文件占用的逻辑存储块
if (read_rfsdevblk(devp,buf,dirp->rdr_blknr)==DFCERRSTUS)
{
    rets=NULL;
    goto err1;
}
fimgrhd_t* ftmp=(fimgrhd_t*)buf;
//判断将要打开的文件是否合法
if (ftmp->fmd_type==FMD_NUL_TYPE||ftmp->fmd_fileifstbkoff!=0x200)
{
    rets=NULL;
    goto err1;
}
//设置缓冲区首地址为返回值
rets=buf;
goto err;
err1:
//上面的步骤若出现问题就要释放缓冲区
del_buf(buf,FSYS_ALCBLKSZ);
err:
//释放根目录文件
del_rootdirfile_blk(devp,rblkp);
return rets;
}

```

rfs_openfileblk 函数中的 for 循环，我们不只见过一次，它就是用来搜索将要打开的文件在根目录文件中对应的 rfsdir_t 结构的，然后把对应文件占用的逻辑存储块读取到缓冲区中，最后返回这个缓冲区的首地址，因为这个缓冲区开始的空间中，就存放着其文件对应的 fimgrhd_t 结构，所以在 rfs_open_file 函数的末尾，把 rfs_openfileblk 函数的返回值直接写入了 IO 包的 on_finode 域中，然后整个打开文件的过程就结束了。

好了，打开文件就是为了读写文件，下面我们就去实现文件的读写操作。

11.5.4 读写文件

打开一个存在的文件，就是为了读写该文件，其实对文件的读写包含两个操作，一是从存储介质中读取文件的数据，二是把文件的数据写入到存储介质中去。

下面先来看一看怎么读取已经打开的文件中的数据，其方法如下：

- 1) 检查 IO 包 (objnode_t 结构) 中用于存放文件数据的缓冲区及其大小。
- 2) 检查 fimgrhd_t 结构中文件相关的信息。
- 3) 把文件的数据读取到 IO 包中用于存放文件数据的缓冲区中。

下面仍然用两个函数来实现上述 3 大步骤, 如代码清单 11-18 [xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-18 读取文件中的数据

```
//读取文件数据的接口函数
drvstus_t rfs_read_file(device_t* devp,void* iopack)
{
    //IO包指针
    objnode_t* obp=(objnode_t*)iopack;
    //检查文件是否已经打开, 以及用于存放文件数据的缓冲区和它的大小是否合理
    if(obp->on_finode==NULL||obp->on_buf==NULL||obp->on_bufsz!=FSYS_ALCBLKSZ)
    {
        return DFCERRSTUS;
    }
    return rfs_readfileblk(devp, (fimgrhd_t*) obp->on_finode, obp->on_buf, obp->on_len);
}
//实际读取文件数据的函数
drvstus_t rfs_readfileblk(device_t* devp,fimgrhd_t* fmp,void* buf,uint_t len)
{
    //检查文件的相关信息是否合理
    if(fmp->fmd_sfblk!=fmp->fmd_curfwritebk||fmp->fmd_curfwritebk!=fmp->fmd_
    fleblk[0].fb_blkstart)
    {
        return DFCERRSTUS;
    }
    //检查读取文件数据的长度是否大于4096-512
    if(len>(FSYS_ALCBLKSZ-fmp->fmd_fileifstbkoff))
    {
        return DFCERRSTUS;
    }
    //指向文件数据的开始地址
    void* wrp=(void*)((uint_t)fmp+fmp->fmd_fileifstbkoff);
    //把文件开始处的数据复制len个字节到buf指向的缓冲区中
    hal_memcpy(wrp,buf,len);
    return DFCOKSTUS;
}
```

读取文件数据的函数很简单, 关键是要明白前面那个打开文件的函数, 因为在那里它已经把文件数据复制到一个缓冲区中了。rfs_readfileblk 函数中的参数 buf、len 都是上层函数从 IO 包中提取出来的, 也是因为限制一个文件只能分配一个逻辑存储块且其中的前 512B 的空间中存放 fimgrhd_t 结构, 所以要检查读取数据的长度 len 是否大于 4096 - 512。

好了，下面就来实现怎么向文件中写入数据，其方法如下：

- 1) 检查 IO 包 (objnode_t 结构) 中用于存放文件数据的缓冲区及其大小。
- 2) 检查 fimgrhd_t 结构中文件相关的信息。
- 3) 把将要写入的数据写入到打开文件时为其分配的缓冲区中。
- 4) 把打开文件时为其分配的缓冲区中的数据写入到相应的逻辑存储块中。

下面仍然用两个函数来实现上述这些步骤，注意执行上述这些步骤之前，一定要确保相应的文件已经打开了，如代码清单 11-19[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-19 向文件中写入数据

```
//写入文件数据的接口函数
drvstus_t rfs_write_file(device_t* devp,void* iopack)
{
    objnode_t* obp=(objnode_t*)iopack;
    //检查文件是否已经打开，以及用于存放文件数据的缓冲区和它的大小是否合理
    if(obp->on_finode==NULL||obp->on_buf==NULL||obp->on_bufsz!=FSYS_ALCBLKSZ)
    {
        return DFCERRSTUS;
    }
    return rfs_writefileblk(devp,(fimgrhd_t*)obp->on_finode,obp->on_buf,obp->on_
len);
}
//实际写入文件数据的函数
drvstus_t rfs_writefileblk(device_t* devp,fimgrhd_t* fmp,void* buf,uint_t len)
{
    //检查文件的相关信息是否合理
    if(fmp->fmd_sfblk!=fmp->fmd_curfwritebk||fmp->fmd_curfwritebk!=fmp->fmd_fleblk[0].
fb_blkstart)
    {
        return DFCERRSTUS;
    }
    //检查当前将要写入数据的偏移量加上写入数据的长度，是否大于等于4KB
    if((fmp->fmd_curfinwbkoff+len)>=FSYS_ALCBLKSZ)
    {
        return DFCERRSTUS;
    }
    //指向将要写入数据的内存空间
    void* wrp=(void*)((uint_t)fmp+fmp->fmd_curfinwbkoff);
    //把buf缓冲区中的数据复制len个字节到wrp指向的内存空间中去
    hal_memcpy(buf,wrp,len);
    //增加文件大小
    fmp->fmd_filesz+=len;
    //使fmd_curfinwbkoff指向下一次将要写入数据的位置
    fmp->fmd_curfinwbkoff+=len;
    //把文件数据写入到相应的逻辑存储块中，完成数据同步
```

```

write_rfsdevblk(devp, (void*) fmp, fmp->fmd_curfwritebk);
return DFCOKSTUS;
}

```

注意，`rfs_writefileblk` 函数永远都是从 `fimgrhd_t` 结构的 `fmd_curfinwbkoff` 域中的偏移量开始存放文件数据的，即追加写入数据的方式，例如，向空文件中写入 2 个字节后，那么 `fimgrhd_t` 结构的 `fmd_curfinwbkoff` 域中的值就是 2，因为第 0、1 个字节空间已经被占用了。`rfs_writefileblk` 函数最后调用 `write_rfsdevblk` 函数把文件数据写入到相应的逻辑存储块中，完成数据同步。

到这里，我们发现只要文件被打开了，对于文件读写功能的实现，还是很简单的，现在接着就去实现关闭文件的功能。

11.5.5 关闭文件

前面打开文件的操作中，我们看到打开一个文件，会为此分配一个缓冲区和一个 IO 包。这些都是系统资源，当打开一个文件对其进行读写操作之后，就需要一个操作来释放这些占用的资源，这个操作就是关闭文件的操作。

首先还是来看一看关闭文件的方法，如下：

- 1) 检查文件是否已经打开。
- 2) 把文件数据写入到对应的逻辑存储块中，完成数据的同步。
- 3) 释放文件数据占用的缓冲区。

我们依然用两个函数来完成上述三个步骤，如代码清单 11-20[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-20 关闭文件

```

//关闭文件的接口函数
drvstus_t rfs_close_file(device_t* devp,void* iopack)
{
    objnode_t* obp=(objnode_t*)iopack;
    //检查文件是否已经打开了
    if(obp->on_finode==NULL)
    {
        return DFCERRSTUS;
    }
    return rfs_closefileblk(devp,obp->on_finode);
}
//关闭文件的核心函数
drvstus_t rfs_closefileblk(device_t *devp, void* fblkp)
{
    //指向文件的fimgrhd_t结构
    fimgrhd_t* fmp=(fimgrhd_t*) fblkp;
    //完成文件数据的同步

```

```

    write_rfsdevblk(devp, fblkp, fmp->fmd_sfbk);
//释放缓冲区
    del_buf(fblkp, FSYS_ALCBLKSZ);
    return DFCOKSTUS;
}

```

关闭文件的操作是非常简单的。注意：在目前的情况下，`rfs_closefileblk` 函数中是不必要调用 `write_rfsdevblk` 函数的，因为前面限制了每个文件只能分配一个逻辑存储块，在写入文件数据的同时，就已经把文件的数据写入到逻辑存储块中去了。最后释放了先前打开文件时分配的缓冲区，而 IO 包不应该在此释放，它是由 LMOSEM 内核上层组件的代码进行释放的。

好了，到这里，我们的文件系统所有的文件操作全部已经实现了。下面还要把这些操作函数和文件系统驱动程序的框架代码联系起来，文件系统才能真正工作。

11.5.6 驱动整合

到目前为止，我们实现了文件系统所有的文件操作，并且提供了相关的接口函数，但是我们的文件系统是以设备的形式存在的。这些文件操作的接口函数，必须要整合到文件系统设备驱动程序之中，文件系统才能真正工作，下面我们就去整合文件系统设备的驱动程序。

先来整合文件系统的打开文件操作和新建文件操作，如代码 11-21[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-21 整合打开、新建文件的操作

```

drvstatus_t rfs_open(device_t* devp, void* iopack)
{
    objnode_t* obp=(objnode_t*)iopack;
//根据IO包的访问标志进行判断
    if(obp->on_acsflgs==FSDEV_OPENFLG_OPEFILE)
    {
        return rfs_open_file(devp, iopack);
    }
    if(obp->on_acsflgs==FSDEV_OPENFLG_NEWFILE)
    {
        return rfs_new_file(devp, obp->on_fname, 0);
    }
    return DFCERRSTUS;
}

```

`rfs_open` 函数对应于设备驱动程序的打开功能派发函数，而设备驱动程序没有相应的新建功能派发函数，于是我们就给 IO 包的访问标志域设置不同的编码，来进行判断。`FSDEV_OPENFLG_OPEFILE`、`FSDEV_OPENFLG_NEWFILE` 两个宏也是由驱动模型规定的，就是两个不同的整数。

接着来整合关闭文件的操作。这次简单一些，因为设备驱动程序有对应的关闭功能派发函数，直接调用关闭文件操作的接口函数就可以了，如代码清单 11-22[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-22 整合关闭文件的操作

```
drvstus_t rfs_close(device_t* devp,void* iopack)
{
    return rfs_close_file(devp,iopack);
}
```

然后是文件读写操作，设备驱动程序也有对应的读写功能派发函数，同样也是直接调用文件读写操作的接口函数即可，如代码清单 11-23[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-23 整合读写文件的操作

```
drvstus_t rfs_read(device_t* devp,void* iopack)
{
    //调用读文件操作的接口函数
    return rfs_read_file(devp,iopack);
}

drvstus_t rfs_write(device_t* devp,void* iopack)
{
    //调用写文件操作的接口函数
    return rfs_write_file(devp,iopack);
}
```

最后来看看稍微有点复杂的删除文件操作。这是因为设备驱动程序没有对应的功能派发函数，这就要用到设备驱动程序的控制功能派发函数了，如代码清单 11-24[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-24 整合删除文件的操作

```
drvstus_t rfs_ioctl(device_t* devp,void* iopack)
{
    objnode_t* obp=(objnode_t*)iopack;
    //根据IO包中的控制码进行判断
    if (obp->on_ioctlrd==FSDEV_IOCTLCD_DELFILE)
    {
        //调用删除文件操作的接口函数
        return rfs_del_file(devp,obp->on_fname,0);
    }
    return DFCERRSTUS;
}
```

LMOSEM 内核的驱动模型，会给文件系统设备分配一个 FSDEV_IOCTLCD_DELFILE（一个整数）控制码。LMOSEM 内核上层组件的代码就可以根据需要，设置 IO 包中的控制码

就能达到相应的目的了。

文件系统设备的驱动程序整合好了，这意味着我们的文件系统已经实现了。不过先别高兴得太早，这么多的代码到底有没有问题，还不知道，所以接下来就去做一些严格的测试。

11.6 文件系统测试

1000 多行代码，实现了一个超简单但也五脏俱全的文件系统。前面我们只是非常认真地写着代码，尽管非常认真，仍然不能凭感觉就认为那么多的代码都是正确的，代码到底正不正确，上物理计算平台测试一下就知道了。

下面将首先测试文件系统的格式化操作。这用于确定文件系统的超级块、位图、根目录这些用于组织文件的基本数据结构建立得对不对。然后就是测试关于文件本身的几大操作。好了，话不多说，开始吧！

11.6.1 格式化测试

前面已经看到了文件系统的格式化操作，就是在存储介质之上建立文件系统的超级块、位图、根目录，但是建立它们的代码到底有没有问题，现在还不知道，所以下面要进行一些测试。

先来测试一下建立文件系统超级块的代码。测试方法非常简单，我们只要把超级块读取到一个缓冲区中，然后把其中一些重要的数据，打印出来看一看就知道了，如代码清单 11-25[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-25 打印文件系统超级块中的相关数据

```
void test_rfs(device_t* devp)
{
    //获取超级块
    rfssublk_t* sbp=get_superblk(devp);
    printfk("rsb_mgic %x,vec %x\n\r",sbp->rsb_mgic,sbp->rsb_vec);
    printfk("rsb_sblksz %x,rsb_dblksz %x\n\r",sbp->rsb_sblksz,sbp->rsb_dblksz);
    printfk("rsb_bmpbks %x,rsb_bmpbknr %x,rsb_fsysallblk %x\n\r",sbp->rsb_
        bmpbks,sbp->rsb_bmpbknr,sbp->rsb_fsysallblk);
    printfk("rootdirblk:%x dirtytype:%x\n\r",sbp->rsb_rootdir.rdr_blknr,sbp->rsb_
        rootdir.rdr_type);
    printfk("rootdir:%s\n\r",sbp->rsb_rootdir.rdr_name);
    //释放超级块
    del_superblk(devp,sbp);
    //死机
    hal_sysdie("test rfs run");
    return;
}
//测试文件系统的主函数，这个函数将在rfs_fmat函数中被调用
```

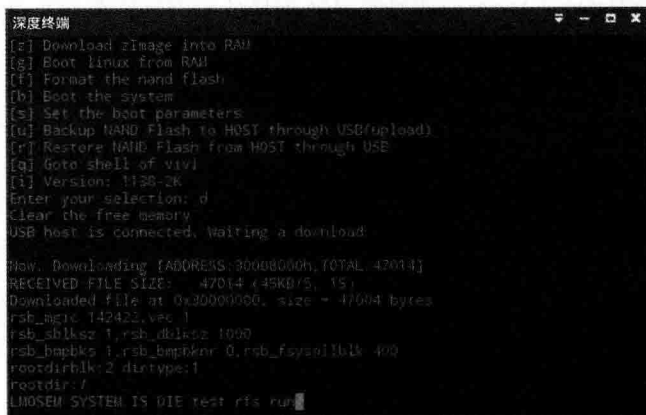


```

void test_file(device_t* devp)
{
    test_rfs(devp);
    .....
    return;
}

```

按照前面章节的方法，编译一下 LMOSEM 内核，连接好开发板把编译好的 LMOSEM 内核下载到开发板中运行。正常情况下，应该如图 11-8 所示。



```

深度终端
[?] Download zImage into RAM
[?] Boot linux from RAM
[?] Format the nand flash
[?] Boot the system
[?] Set the boot parameters
[?] Backup NAND Flash to HOST through USB(upload)
[?] Restore NAND Flash from HOST through USB
[?] Goto shell of vivi
[?] Version: 1120-28
Enter your selection: d
Clear the free memory
USB host is connected, waiting a download
Now, Downloading [ADDRESS:30080000,TOTAL 47014]
RECEIVED FILE SIZE: 47014 (45KB/5, 15)
Downloaded file at 0x30000000, size = 47004 bytes
rfs_nrc 142422 vec 1
rfs_shlks 1.rfs_dlnsz 1000
rfs_bmpbks 1.rfs_bmpbknr 0.rfs_fsysblk 400
vcoordirblk 2 dirtyp:1
rootdir /
LMOSEM SYSTEM IS DIE test rfs run

```

图 11-8 输出文件系统超级块的信息

图 11-8 中的数据都是十六进制的，可以看到文件系统的标识、版本和最初定义的是相同的，逻辑存储块的大小为 4KB，位图占用第 1 个逻辑存储块，因为第 0 个逻辑存储块被超级块占用了，同时也看到根目录文件的逻辑存储块为第 2 块，根目录名称为“/”，存储介质上共有 1024 个逻辑存储块。这些正确的数据证明了建立文件系统超级块的代码没有问题。

接着来看看怎么测试位图。其测试方法也很简单，先读取位图块到一个缓冲区中，然后循环扫描这个缓冲区，看看里面有多少个为 0 的字节，即表明存储介质上有多少个空闲的逻辑存储块，如代码清单 11-26[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-26 检查文件系统位图块

```

//chk_rfsbitmap函数将要在test_file函数中被调用
void chk_rfsbitmap(device_t* devp)
{
    //分配一个缓冲区
    void* buf=new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
    {
        hal_sysdie("chkbitmap err");
    }
    //清零缓冲区

```

```

    hal_memset(buf,FSYS_ALCBLKSZ,0);
//读取位图块
    if(read_rfsdevblk(devp,buf,1)==DFCERRSTUS)
    {
        hal_sysdie("chkbitmap err1");
    }
//指向缓冲区
    u8_t* bmp=(u8_t*)buf;
    uint_t b=0;
//循环扫描缓冲区
    for(uint_t i=0;i<FSYS_ALCBLKSZ;i++)
    {
        if(bmp[i]==0)
        {
            //为0则加1
            b++;
        }
    }
//输出b的值
    printfk("chk bmpp b:%x\n\r",b);
//释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
    return;
}

```

接着编译一下 LMOSEM 内核，把 LMOSEM 内核下载到开发板中运行一下。正常情况下。应该如图 11-9 所示。

```

深度终端
[In] Download Mboot for winCE
[1] Downloaded winCE boot-logo
[2] Download winCE HK.bin
[3] Download & Run
[4] Download zImage into RAM
[5] Boot linux from RAM
[6] Format the hard flash
[7] Boot the system
[8] Set the boot parameters
[9] Backup NAND Flash to HOST through USB(upload)
[10] Restore NAND Flash from HOST through USB
[11] Goto shell of vios
[12] Version: 1130-2F
Enter your selection: 4
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading (ADDR:0x0, TOTAL: 4096)
+512KIBD FILE SIZE: 13906 (450B N, 15)
Downloaded file at 0x00000000, size = 4096 bytes
chk bmpp b:3fd
LMOSEM SYSTEM IS FAT ext file run

```

图 11-9 检查文件系统位图块

图 11-9 中的 3fd 数据表示存储介质上，已经分配了 3 块逻辑存储块了。即我们的建立位

图的代码是没有问题的。

最后我们来看看根目录文件建立得对不对。测试方法就是先得到根目录文件的 `rfmdir_t` 结构, 然后读取其中指向的逻辑存储块到缓冲区中, 最后把它们的数据打印出来, 如代码清单 11-27[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单11-27 打印根目录文件的数据

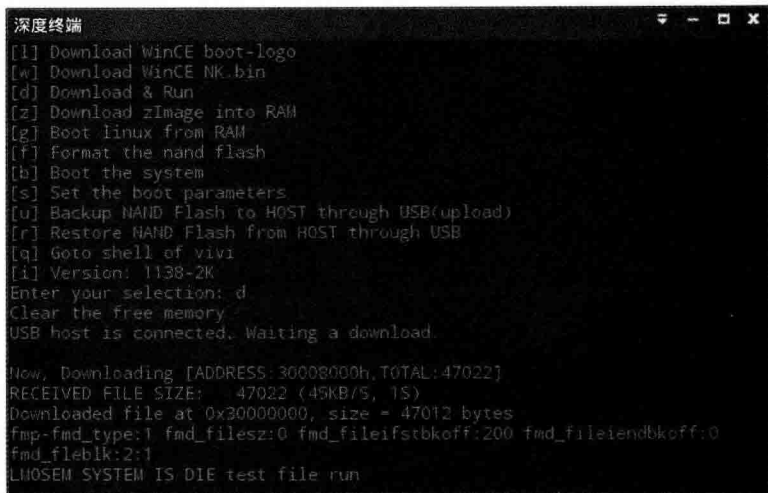
```
//test_dir函数将要在test_file函数中被调用
void test_dir(device_t* devp)
{
    //获取根目录文件的rfmdir_t结构
    rfmdir_t* dr=get_rootdir(devp);
    //分配缓冲区
    void* buf = new_buf(FSYS_ALCBLKSZ);
    if(buf==NULL)
    {
        hal_sysdie("testdir err");
    }
    hal_memset(buf,FSYS_ALCBLKSZ,0);
    //读取根目录文件的逻辑存储块到缓冲区中
    if(read_rfsdevblk(devp,buf,dr->rdr_blknr)==DFCERRSTUS)
    {
        hal_sysdie("testdir1 err1");
    }
    //指向缓冲区
    fimgrhd_t* fmp=(fimgrhd_t*)buf;
    printfk("fmp-fmd_type:%x fmd_filesz:%x fmd_fileifstbkoff:%x fmd_fileiendbkoff:%x\n\r",
        fmp->fmd_type,fmp->fmd_filesz,fmp->fmd_fileifstbkoff,fmp->fmd_fileiendbkoff);
    printfk("fmd_fleblk:%x:%x\n\r",fmp->fmd_fleblk[0].fb_blkstart,fmp->fmd_fleblk[0].fb_blknr);
    //释放缓冲区
    del_buf(buf,FSYS_ALCBLKSZ);
    //释放根目录文件的rfmdir_t结构
    del_rootdir(devp,dr);
    return;
}
```

再次编译 LMOSEM 内核, 然后把 LMOSEM 内核下载到开发板中运行。正常情况下, 应该如图 11-10 所示。

可以发现, 根目录文件的类型为目录文件类型, 文件大小为 0。因为才刚刚建立, 文件数据的存放位置, 从文件占用的第 1 块逻辑存储块的 512 (0x200) 字节处开始, 根目录文件占用的逻辑存储块, 从存储介质的第 2 块逻辑存储块开始且只占用 1 块, 因为第 0、1 块逻辑存储块已经被超级块和位图块占用了。

好了, 上面一系列的测试表明, 我们的文件系统格式化的代码正确无误。下面就去测试

文件本身相关的操作。



```

深度终端
[1] Download WinCE boot-logo
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1138-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30008000h,TOTAL:47022]
RECEIVED FILE SIZE: 47022 (45KB/S, 1S)
Downloaded file at 0x30000000, size = 47012 bytes
fmp-fmd_type:1 fmd_filesz:0 fmd_fileifstbkoff:200 fmd_fileiendbkoff:0
fmd_fileblk:2:1
LMOSEM SYSTEM IS DIE test file run
  
```

图 11-10 输出根目录文件的信息

11.6.2 文件操作测试

前面实现了文件系统的 6 大最常用的文件操作，并且已经整合到文件系统设备驱动程序框架代码中去了。可是这些代码究竟对不对，运行了才知道。

在开始之前，要注意一点，任何设备的驱动程序都必须要有 IO 包才能运行，所以在这里我们就手动建立一个 IO 包并设置好其中相关的域，模拟一下 LMOSEM 内核上层组件调用设备驱动程序的过程，如代码清单 11-28[xxxx/lmosem/drivers/drvrfs.c] 所示。

代码清单 11-28 测试基本的文件操作

```

void test_fsys(device_t* devp)
{
    //分配缓冲区
    void* rwbuf=new_buf(FSYS_ALCBLKSZ);
    if(rwbuf==NULL)
    {
        hal_sysdie("rwbuf is NULL");
    }
    //把缓冲区中的所有字节都置为0xff
    hal_memset(rwbuf,FSYS_ALCBLKSZ,0xff);
    //新建一个IO包
    objnode_t* ondp=krlnew_objnode();
    if(ondp==NULL)
    {
        hal_sysdie("ondp is NULL");
    }
  
```

```

    }
//设置新建文件标志
    ondp->on_acsflgs=FSDEV_OPENFL G_NEWFILE;
//设置文件名
    ondp->on_fname="/testfile";
//设置缓冲区
    ondp->on_buf=rwbuf;
//设置缓冲区大小
    ondp->on_bufsz=FSYS_ALCBLKSZ;
//设置读写多少字节
    ondp->on_len=512;
//设置控制码
    ondp->on_ioctrd=FSDEV_IOCTLCD_DELFILE;
//新建文件
    if(rfs_open(devp, ondp)==DFCERRSTUS)
    {
        hal_sysdie("new file is err");
    }
//设置打开文件标志
    ondp->on_acsflgs=FSDEV_OPENFLG_OPEFILE;
//打开文件
    if(rfs_open(devp, ondp)==DFCERRSTUS)
    {
        hal_sysdie("open file is err");
    }
//把数据写入文件
    if(rfs_write(devp, ondp)==DFCERRSTUS)
    {
        hal_sysdie("write file is err");
    }
//清零缓冲区
    hal_memset(rwbuf, FSYS_ALCBLKSZ, 0);
//读取文件数据
    if(rfs_read(devp, ondp)==DFCERRSTUS)
    {
        hal_sysdie("read file is err");
    }
//关闭文件
    if(rfs_close(devp, ondp)==DFCERRSTUS)
    {
        hal_sysdie("close file is err");
    }
//指向缓冲区
    char_t* cb=(char_t*)rwbuf;
//检查缓冲区空间中的头512B的数据, 是否为0xff
    for(uint_t i=0; i<512; i++)

```

```

{
    //如果不等于0xff就死机
    if(cb[i]!=0xff)
    {
        hal_sysdie("chek rwbuf err");
    }
    printfk("rwblk[%x]:%x\n\r",i,(uint_t)cb[i]);
}
//删除文件
if(rfs_ioctl(devp,ondp)==DFCERRSTUS)
{
    hal_sysdie("del file is err");
}
//重新设置打开文件标志
ondp->on_acsflgs=FSDEV_OPENFLG_OPEFILE;
//重新打开这个文件
if(rfs_open(devp,ondp)==DFCERRSTUS)
{
    hal_sysdie("2open2 file is err");
}
hal_sysdie("test_fsys ok");
return;
}

```

代码清单 11-28 虽然有点长，不过我们一下子测试了关于文件本身的 6 大操作，代码大致做了如下步骤：

- 1) 分配一个缓冲区，并向其中写入 0xFF 数值。
- 2) 建立一个 IO 包，并设置好其中的相关域。
- 3) 新建 “/testfile” 文件。
- 4) 打开 “/testfile” 文件。
- 5) 把缓冲区中的前 512B 的数据写入到 “/testfile” 文件中。
- 6) 清零缓冲区，为下一步读取 “/testfile” 文件数据做准备。
- 7) 读取 “/testfile” 文件的头 512B 的数据到缓冲区中。

8) 检查缓冲区中开始的 512B 的数据，看看是否都为 0xFF，因为写入时就是 0xFF，如果现在不是 0xFF 了，就说明出错了。

- 9) 删除 “/testfile” 文件。

10) 重新设置 IO 包的打开文件标志并打开 “/testfile” 文件，注意 “/testfile” 文件已经删除了，这时应该会打开失败的。

编译一下 LMOSEM 内核，然后把 LMOSEM 内核下载到开发板中运行，应该会出现如图 11-11 所示的情况。

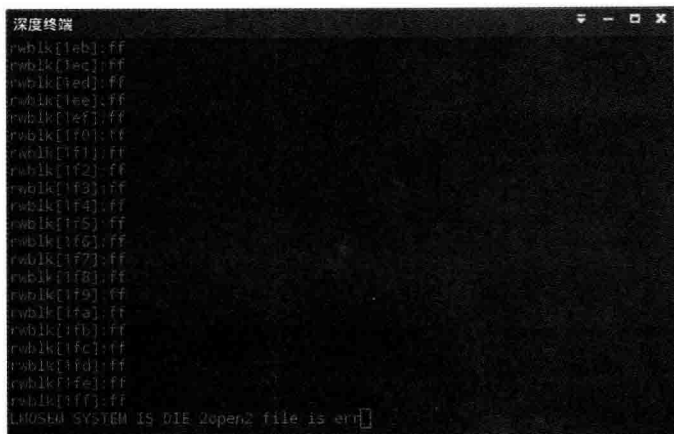


图 11-11 测试文件操作

从图 11-11 中，看到缓冲区中的数据 and 最后重新打开已经删除的“/testfile”文件时出现的错误：“2open2 file is err”，这说明了我们的代码是正确无误的。

到这里，测试了文件系统的格式化代码和关于文件本身的 6 大文件操作的代码，代码质量都是相当高的，都达到了我们的预期。一个简单、有诸多限制却五脏俱全的文件系统就实现了。

11.7 小结

文件系统虽然复杂，但我们做了很多限制，大大降低了实现的难度。虽然降低了实现的难度，但本章依然做了如下事情。

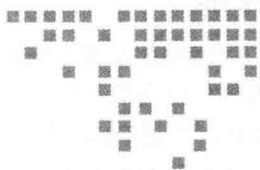
首先是文件系统的设计，在这里我们了解了文件系统只是 LMOSEM 内核下的一个设备，并用内存空间模拟了文件系统的存储介质，了解了文件的格式及组织文件的方式，但为了简化文件系统的实现，我们又对其做了一些限制。

接着是文件系统的格式化操作，格式化操作在存储介质上建立了文件系统的超级块、位图、根目录这一整套用于管理文件的数据结构。

然后是文件系统的基础操作，因为文件系统的复杂性，所以必须要实现一些诸如获取与释放根目录文件、字符串操作、分解文件路径名、检查文件是否存在等基础操作。

最后实现了文件系统必须要提供的 6 大文件操作：新建文件、删除文件、打开文件、读写文件、关闭文件，并且把这些文件操作全部整合到文件系统设备驱动程序框架之中。

又实现了一个 LMOSEM 内核里的基础组件，不过它是以设备的形式存在的。这样做是为了方便以后的扩展和移植，同时也证明了 LMOSEM 内核驱动模型的设计是优秀的。文件系统是实现了，但是一个最小的操作系统内核还没有完成。休息一下，继续探索。



系统调用与应用程序库

从计算平台的细节到第一行代码的运行再到这里，我们的 LMOSEM 内核已经具备了管理内存、运行多进程、读写文件、操控设备等功能。实现这些功能就是要以某种方式为应用程序服务。这种方式就是让应用程序需要什么系统服务，就调用对应的系统调用或者系统 API。

本章将从系统调用机制开始，然后实现时间管理、进程管理、内存管理、设备与文件管理等系统调用，最后实现一些相应的库，因为应用程序一般不直接使用系统调用。但是，为了迎合读者已有的知识体系，LMOSEM 系统调用的实现尽量接近于 UNIX 系统调用，因此也相应地实现了一个 UNIXAPI 子集。

系统调用属于 LMOSEM 系统的接口层，下面就去实现这个接口层。

12.1 系统调用机制

通常操作系统内核代码和应用程序代码运行在不同的内存地址空间中，并且应用程序代码不得访问操作系统内核的内存地址空间，这是为了保护操作系统内核代码不被应用程序破坏。可是应用程序要调用系统调用，从而使用操作系统内核提供的服务，就必须要和操作系统内核通信。

下面首先了解一下应用程序与操作系统内核的通信机制，然后看一看如何处理系统调用的参数，最后因为有很多的系统调用，所以得实现一个系统调用分发器。

以上都是系统调用的基础设施，下面就去实现它们。

12.1.1 软中断指令

上面说了，分离操作系统内核和应用程序是为了保护操作系统内核，但是应用程序又必须和操作系统内核进行通信，因此就必须实现一种机制，让应用程序在可控的状态下，调用系统调用从而使用操作系统内核提供的服务。

在前面关于中断管理章节中，看到设备和 CPU 通信的形式是设备向 CPU 发送一个中断信号，CPU 接收到这个信号后，在允许响应中断的情况下，就中断当前正在运行的程序，自动切换到相应的 CPU 工作模式并跳转到相应的固定地址上运行中断处理代码。当然，这里的中断处理代码就是操作系统内核的代码。就这样，CPU 的控制权就转到操作系统内核的手中了。

下面接着思考，应用软件是否也可以用设备和 CPU 通信的方式达到和操作系统内核进行通信的目的呢？当然是可行的，只是应用软件不能向 CPU 发送电子信号。现代 CPU 都设计有一条指令，一旦执行该指令，CPU 就中断当前正在运行的程序，自动跳转到相应的固定地址上运行代码。这条指令模拟了中断的电子信号，所以称为软中断指令。在 ARM CPU 上，这条指令是 swi 指令。关于 swi 指令，前面指令集章节中有详细的介绍，这里不再详述。

依然来回忆一下，在我们的计算平台上，执行 swi 指令后 CPU 执行 LMOSEM 内核代码的过程：CPU 在执行 swi 指令后会自动切换到系统管理模式并跳转到 0x8 地址上运行“ldr pc, lmosem_swi_1”指令，如代码清单 12-1[xxxx/lmosem/hal/vector.S] 所示。

代码清单12-1 ARM920T的中断、异常向量

```
.global vector
vector:
    ldr pc, lmosem_restart_1
    ldr pc, lmosem_undefins_1
    ldr pc, lmosem_swi_1
    ldr pc, lmosem_prefabt_1
    ldr pc, lmosem_dataabt_1
    .....
.align 4
lmosem_restart_1: .word _start
lmosem_undefins_1: .word hal_undefins
lmosem_swi_1:      .word hal_swi
lmosem_prefabt_1:  .word hal_prefabt
lmosem_dataabt_1:  .word hal_dataabt
    .....
```

代码清单 12-1 中的“ldr pc, lmosem_swi_1”代码会导致 CPU 继续跳转到 hal_swi 处运行，hal_swi 处的代码就是 LMOSEM 内核处理 swi 软中断的代码，如代码清单 12-2[xxxx/lmosem/hal/interrupt.S] 所示。

代码清单12-2 处理swi软中断

```

hal_swi:
    sub    sp,sp,#ALLREGS_STKSZ
    stmia  sp,{r1-lr}^
    mrs    r1,spsr
    str    lr,[sp,#STKOFFSET_PC]
    stmfd  sp!,{r1,lr}
    mov    r1,sp
    bl     hal_swi_distr
    ldmfd  sp!,{r1,lr}
    msr    spsr,r1
    ldmia  sp,{r1-lr}^
    add    sp,sp,#NPCREGS_STKSZ
    ldmfd  sp!,{pc}^

```

代码清单 12-2 中以“bl hal_swi_distr”这行代码为界限，上半部分是保存 CPU 用户模式下的除 r0 外所有寄存器中的值，下半部分就是恢复先前保存的值到相应的寄存器中并执行返回操作。hal_swi_distr 是一个 C 语言代码函数，如代码清单 12-3[xxxx/lmosem/hal/intabtdistr.c] 所示。

代码清单12-3 swi软中断派发函数

```

//按照ATPCS的规定，hal_swi_distr函数的两个参数应该由r0、r1寄存器来传递，由上述代码可知
//sframe就是SP中的值，即栈指针，而swinr是一个整数保存在r0中
sysstus_t hal_swi_distr(uint_t swinr,void* sframe)
{
    return krlservice(swinr,sframe);
}

```

在代码清单 12-3 中，hal_swi_distr 函数直接调用了 krlservice 函数，而且参数原封不动地传递给了 krlservice 函数。关于 krlservice 函数是干什么的，后面会详细介绍的。

我们看到了 CPU 执行 swi 指令后，从“ldr pc,lmosem_swi_l”代码到 krlservice 函数这一执行过程就是应用程序用 swi 指令完成调用系统 API 的过程。对于每个应用程序来说，调用 krlservice 函数之前的步骤都是相同的。另外，除了使用 swi 指令外，以其他形式访问 LMOSEM 内核的内存地址空间或者调用 LMOSEM 内核函数，都会被 CPU 视为违规操作，CPU 会发出相应的异常，最终也会转到 LMOSEM 内核的异常处理代码上去。

到这里，不难发现，正是因为有 swi 指令和 CPU 的保护机制，才使得应用程序既可以和 LMOSEM 内核通信（实则是调用 LMOSEM 内核函数），又能使 LMOSEM 内核不被不良应用程序破坏。有了 swi 指令就可以调用到 LMOSEM 内核函数了，但是调用函数还要传递参数，下面就去解决这个问题。

12.1.2 传递系统调用参数

前面的 swi 指令提供了应用程序调用操作系统内核函数的底层机制。既然是调用函数，那么肯定需要解决函数参数传递的问题。注意，下文不区分系统调用、系统调用函数和系统 API。

在通常情况下，按照 ATPCS 的规定，C 语言代码中的函数的前 4 个参数用 r0 ~ r3 这 4 个寄存器传递，多于 4 个的参数放在当前的栈中传递。对于系统调用就不能让 C 编译器自由安排了，因为对于一个应用程序，当它运行在用户模式下时，用的是用户栈；当它切换到内核特权模式下时，用的是内核栈。所以系统调用参数的传递，就需要硬性地规定一下，要么所有的参数都用寄存器传递，要么所有的参数都保存在用户栈中。显然第一种用寄存器传递所有参数的方法要简单得多，事实上有很多操作系统就是用寄存器传递系统调用参数的。

在 LMOSEM 系统中，使用 r1 ~ r7 这 7 个寄存器来传递系统调用的参数。事实上，一个系统调用函数不会超过 5 个参数，所以这是足够的，而 r0 寄存器中保存着一个整数，称为系统调用号。在 krlservice 函数中会根据这个系统调用号调用相应的函数，这是因为 swi 指令只能让 CPU 跳转到固定的地址上运行，而不是任意地址，而且应用程序也不可能知道每个用于实现系统调用的内核函数的地址。既然不用 C 编译器来处理参数，另外，C 编译器也不支持 swi 指令，所以要用汇编代码来处理这种问题，如代码清单 12-4 [xxxx/lmosem/include/libinc/lapinrentry.h] 所示。

代码清单12-4 处理系统调用参数的相关宏

```
#define API_ENTRY_PARE1(swinr,rets,pvall) __asm__ __volatile__(\
    "mov r0,%[snr] \n\t"\
    "mov r1,%[prv1] \n\t"\
    "swi 0 \n\t"\
    "mov %[retval],r0 \n\t"\
    :[retval] "=r" (rets)\
    :[snr] "r" (swinr),[prv1]"r" (pvall)\
    : "r0","r1","lr","cc","memory"\
)

#define API_ENTRY_PARE2(swinr,rets,pvall,pval2) __asm__ __volatile__(\
    "mov r0,%[snr] \n\t"\
    "mov r1,%[prv1] \n\t"\
    "mov r2,%[prv2] \n\t"\
    "swi 0 \n\t"\
    "mov %[retval],r0 \n\t"\
    :[retval] "=r" (rets) :[snr] "r" (swinr),[prv1]"r" (pvall),[prv2] "r" (pval2)\
    : "r0","r1","r2","lr","cc","memory"\
)

#define API_ENTRY_PARE3(swinr,rets,pvall,pval2,pval3) __asm__ __volatile__(\
    "mov r0,%[snr] \n\t"\
    "mov r1,%[prv1] \n\t"
```

```

"mov r2,%[prv2] \n\t"\
"mov r3,%[prv3] \n\t"\
"swi 0 \n\t"\
"mov %[retval],r0 \n\t"\
:[retval] "=r" (rets) :[snr] "r" (swinr),[prv1]"r" (pval1),\
[prv2] "r" (pval2),[prv3]"r" (pval3)\
:"r0","r1","r2","r3","lr","cc","memory"\
)
#define API_ENTRY_PARE4(swinr,rets,pval1,pval2,pval3,pval4) __asm__ __volatile__(\
"mov r0,%[snr] \n\t"\
"mov r1,%[prv1] \n\t"\
"mov r2,%[prv2] \n\t"\
"mov r3,%[prv3] \n\t"\
"mov r4,%[prv4] \n\t"\
"swi 0 \n\t"\
"mov %[retval],r0 \n\t"\
:[retval] "=r" (rets) :[snr] "r" (swinr),[prv1]"r" (pval1),\
[prv2] "r" (pval2),[prv3]"r" (pval3),\
[prv4] "r" (pval4)\
:"r0","r1","r2","r3","r4","lr","cc","memory"\
)
#define API_ENTRY_PARE5(swinr,rets,pval1,pval2,pval3,pval4,pval5) __asm__ __volatile__(\
"mov r0,%[snr] \n\t" //把系统调用号放在r0寄存器中
"mov r1,%[prv1] \n\t" //把第一个参数放在r1寄存器中
"mov r2,%[prv2] \n\t" //把第二个参数放在r2寄存器中
"mov r3,%[prv3] \n\t" //把第三个参数放在r3寄存器中
"mov r4,%[prv4] \n\t" //把第四个参数放在r4寄存器中
"mov r5,%[prv5] \n\t" //把第五个参数放在r5寄存器中
"swi 0 \n\t" //执行swi指令
"mov %[retval],r0 \n\t" //处理返回值,返回值固定存放在r0寄存器中
:[retval] "=r" (rets) :[snr] "r" (swinr),[prv1]"r" (pval1),\
[prv2] "r" (pval2),[prv3]"r" (pval3),\
[prv4] "r" (pval4),[prv5]"r" (pval4)\
:"r0","r1","r2","r3","r4","r5","lr","cc","memory" //告诉GCC这些寄存器中的数
//据,可能已经被改写,所以在操作之前应该先保存它们的数据
)

```

在代码清单 12-4 中,有 5 个带参数的宏,每个宏被展开后都是一段 GCC 支持的内连汇编代码,从其中 API_ENTRY 的字样可知,这些代码都是系统 API 的入口代码。由于不同的系统 API 需要参数的个数不同,所以定义了 5 个宏,分别对应需要从 1~5 个参数的系统 API。为了便于理解,在最复杂的一个宏中写好了注释,理解起来应该不难。

好了,明白了代码清单 12-4 中的这些宏,再来看看这些宏的使用场景。下面的代码也是 LMOSEM 的真实系统 API 的入口函数,如代码清单 12-5 所示。

代码清单12-5 系统API的入口函数

```

void* api_mallocblk(size_t blksz)
{
    void* retadr;
    API_ENTRY_PARE1(SNR_MM_ALLOC,retadr,blksz);
    return retadr;
}

sysstus_t api_read(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    sysstus_t rets;
    API_ENTRY_PARE4(SNR_FS_READ,rets,fhand,buf,len,flgs);
    return rets;
}

```

在代码清单 12-5 中，第一个函数是内存分配系统 API 的入口函数，第二个函数是设备与文件的读系统 API 的入口函数，SNR_MM_ALLOC、SNR_FS_READ 即是对应的系统调用号。参数也是从左至右依次传递的，最左边的是第一个参数且需要放在 r1 寄存器中，依次类推。有几个参数就使用对应参数个数的宏，这一点看看前面的宏就明白了。

好了，如果执行了相应系统 API 的入口函数，就会执行 swi 指令，从而导致 CPU 执行软中断的处理代码，进而开始运行 LMOSEM 内核代码，可是 LMOSEM 内核代码怎么获取传递进来的系统调用的参数，又怎么知道应用程序调用的是哪个系统调用？我们继续完成后面的代码。

12.1.3 系统调用分发器

在 12.1.2 小节中，之所以有一个系统调用号，就是因为操作系统内核不只是提供一个系统调用，而应用程序又不能直接调用一个具体的系统调用。

这里就来解决如何从一个整数（系统调用号）调用到具体的系统调用。这个问题说白了就是如何通过一个整数找到一个函数并调用。这可能让我们想到最简单的方法，就是用函数指针数组，即用系统调用号作为数组下标去索引数组中的函数指针然后调用。事实上我们正是这么做的，如代码清单 12-6 所示。

代码清单12-6 LMOSEM系统调用函数指针数组及相关数据结构

```

typedef struct s_STKPARAME
{
    reg_t rvs0; //因为stmfd sp!,{r1,lr}代码会让SP寄存器自增，所以这里有4B的空间
    reg_t rvs1;  //r0寄存器
    reg_t parmv1; //r1寄存器
    reg_t parmv2; //r2寄存器
    reg_t parmv3; //r3寄存器
    reg_t parmv4; //r4寄存器
    reg_t parmv5; //r5寄存器
}

```

```

    reg_t parmv6; //r6寄存器
    reg_t parmv7; //r7寄存器
}stkparam_t;
#define SWINR_MAX 0xf
typedef sysstus_t (*syscall_t)(uint_t swinr,stkparam_t* stkparm);
KRL_DEFGLOB_VARIABLE(syscall_t,osscalltab)[SWINR_MAX]={
    NULL,krlsvetabl_mallocblk,
    krlsvetabl_mfreeblk,krlsvetabl_exel_thread,
    krlsvetabl_exit_thread,
    krlsvetabl_retn_threadhand,
    krlsvetabl_retn_threadstats,
    krlsvetabl_set_threadstats,
    krlsvetabl_open,krlsvetabl_close,
    krlsvetabl_read,krlsvetabl_write,
    krlsvetabl_ioctl,krlsvetabl_lseek,
    krlsvetabl_time};

```

LMOSEM 内核代码正是通过 `stkparam_t` 结构的指针来获取应用程序传递进来的系统调用参数的。不明白的可以看一看 `hal_swi` 处的软中断处理代码。而在代码清单 12-6 中，最后定义了一个全局的函数指针数组并且数组的第一个元素保留为 `NULL`。

接着去看一看看在 `hal_swi_distr` 函数中被调用的 `krlservice` 函数，如代码清单 12-7 所示。

代码清单 12-7 系统调用分发函数

```

sysstus_t hal_swi_distr(uint_t swinr,void* sframe)
{
    return krlservice(swinr,sframe);
}
sysstus_t krlservice(uint_t swinr,void* sframe)
{
    //直接用swinr为数组下标，调用数组里的函数，并传递swinr,(stkparam_t*)sframe两个参数给调用
    //的函数
    return osscalltab[swinr](swinr,(stkparam_t*)sframe);
}

```

`krlservice` 函数在 `krlservice.c` 源代码文件中。要注意的是，它仍然是把 `sframe` 参数原样传递给具体的系统调用函数，因为需要哪些参数只有具体的系统调用函数才知道。

从软中断指令到系统调用参数的传递，再到系统调用分发器，整个系统调用机制就完成了。只要把具体的系统调用实现的接口函数，放在系统调用函数指针数组中就行了，下一步就可以继续实现一个个具体的系统调用了。

12.2 时间管理系统调用

时间在任何场景下都是重要的，操作系统内核必须向应用软件提供相应的时间管理系统

调用。对于应用程序来说，最常用的就是获取当前时间，所以本小节就来实现获取时间的系统调用。

一个应用程序在很多情况下都要获取当前时间，例如，常见的一个应用程序就是显示当前时间供用户查看。时间数据是由专门的时间芯片产生的，这就要求必须经过操作系统内核才能访问时间芯片以获取时间数据，所以操作系统内核必须要提供一个获取时间的系统调用。下面就来实现这个系统调用。

在目前的计算平台和 LMOSEM 内核下，获取时间数据的方法有两种：一是读取 LMOSEM 内核维护的时间数据结构中的数据，二是读取 RTC 设备中的时间数据。当然第一种方法更为简单，所以下面将采用第一种方法。

下面就来设计并实现获取时间数据的系统调用，实现一个系统调用有如下几个步骤。

第一步，首先在“xxxx/lmosem/lib/”目录下建立一个 lapitime.c 文件和建立相应的头文件，并修改编译头文件中相应的宏：“#define BUILD_LIBS_OBJS lapitime.o”。接着，在 lapitime.c 文件中写一个可以在应用程序、应用程序库中都能调用的系统调用入口函数，如代码清单 12-8[xxxx/lmosem/lib/lapitime.c] 所示。

代码清单 12-8 获取时间的系统调用入口函数

```
sysstus_t api_time(buf_t ttime)
{
    sysstus_t rets;
    API_ENTRY_PARE1(SNR_TIME, rets, ttime);
    return rets;
}
```

在代码清单 12-8 中，sysstus_t 类型就是 long 类型。函数名以 api 开头表示这是一个系统调用的入口函数，其中正是用到了先前定义的用来处理系统调用参数和执行软中断指令的宏。这个函数告诉编程人员，它会把一系列（时、分、秒）的时间数据，写入到 ttime 指向的缓冲区中，最后返回一个状态值表示该函数是否执行成功。

第二步，是在“xxxx/lmosem/kernel/”目录下建立一个 krltime.c 文件。不过这个文件中的代码是 LMOSEM 内核的代码，同时也要建立相应的头文件，并且要把它加入到相应的编译宏中：“#define BUILD_KRNL_OBJS krltime.o”。在 krltime.c 文件中，写上具体实现获取时间数据的系统调用的代码，一般用两到三个函数实现实际的功能。但是一个系统调用的第一个接口函数有一定的规定，因为它是存放在系统调用函数指针数组中的，所以它的参数形式和返回值都要遵守相应的规定，如代码清单 12-9[xxxx/lmosem/kernel/krltime.c] 所示。

代码清单 12-9 获取时间系统调用的实现

```
sysstus_t krlsvetabl_time(uint_t swinr, stkparame_t* stkparv)
{
    //判断系统调用号是不是 SNR_TIME，不是则返回 SYSSTUSERR 表示函数执行失败
    if (swinr != SNR_TIME)
```

```

    {
        return SYSSTUSER;
    }
//获取参数并调用具体完成功能的函数
    return krlsve_time((time_t*)stkparv->parmv1);
}
sysstatus_t krlsve_time(time_t* time)
{
//事实上, api_time函数中ttime指向的是一个time_t数据结构的内存空间, 即缓冲区
    if(time==NULL)
    {
        return SYSSTUSER;
    }
//指向系统时间数据结构
    ktime_t* initp=&osktime;
    cpufg_t cpufg;
//加锁
    hal_spinlock_saveflg_cli(&initp->kt_lock,&cpufg);
    time->year=initp->kt_year;
    time->mon=initp->kt_mon;
    time->day=initp->kt_day;
    time->date=initp->kt_date;
    time->hour=initp->kt_hour;
    time->min=initp->kt_min;
    time->sec=initp->kt_sec;
//解锁
    hal_spinunlock_restflg_sti(&initp->kt_lock,&cpufg);
//返回SYSSTUSOK表示函数执行成功
    return SYSSTUSOK;
}

```

krlsvetabl_time 函数以 krlsvetabl_ 开头, 表示它是要被放入系统调用的函数指针数组中的, 而 krlsve_time 函数以 krlsve_ 开头, 表示它是实现具体功能的函数。其中代码逻辑非常简单, 就是在加锁状态下, 把 LMOSEM 内核中的 ktime_t 结构中的数据, 写入到应用程序内存空间中的 time_t 结构中去。下面就来看一看这两个数据结构, 如代码清单 12-10[xxxx/lmosem/include/knlinc/krltime_t.h] 所示。

代码清单12-10 ktime_t、time_t两个时间数据结构

```

typedef struct s_KTIME
{
    spinlock_t    kt_lock;    //自旋锁
    uint_t        kt_year;    //年
    uint_t        kt_mon;     //月
    uint_t        kt_day;     //周
    uint_t        kt_date;    //天

```



```

uint_t      kt_hour;  //时
uint_t      kt_min;   //分
uint_t      kt_sec;   //秒
void*       kt_datap; //扩展所用
}ktime_t;
typedef struct s_TIME
{
    uint_t      year;    //年
    uint_t      mon;     //月
    uint_t      day;     //周
    uint_t      date;    //天
    uint_t      hour;    //时
    uint_t      min;     //分
    uint_t      sec;     //秒
}time_t;

```

可以看到，`ktime_t`、`time_t` 两个数据结构无太大区别，都是用来保存时间数据的，只不过 `ktime_t` 结构是 LMOSEM 内核自身使用的，有自旋锁和扩展域，而 `time_t` 结构则是为应用程序准备的，并且要让应用软件开发人员知道其细节。

好了，一个最简单的获取时间数据的系统调用，就轻松地实现了。以后实现系统调用就和这种代码结构是一样的：先在用户层中编写一个系统调用的入口函数，然后在 LMOSEM 内核层中编写一个系统调用的接口函数，再编写一个或者几个实现系统调用具体功能的函数。下面继续实现其他的系统调用吧！

12.3 进程管理系统调用

进程是现代操作系统中非常重要的概念，所以一款操作系统应该提供一些和进程相关的系统调用。例如，进程的运行与退出，获取进程的 ID 等。

下面就去实现上述功能的系统调用。

12.3.1 进程的运行与退出

在 Windows 下双击应用程序图标，就能运行对应的应用程序，在 Linux 下输入应用程序对应的命令，也能运行相应的应用程序。其实这个机制的背后，有一个专门的应用程序进程，它根据相应的信息（命令、图标），找到对应的应用程序的可执行文件，然后调用一个系统调用，这个应用程序就开始运行了。当这个应用程序的主函数返回时，也会调用另一个系统调用，这时操作系统就会停止这个应用程序并回收其资源，最后删除对应的进程描述符。

可是在 LMOSEM 系统上的情况就不同了，为了简化实现，我们让 LMOSEM 内核和应用程序链接在一起，这就导致上面说的两个系统调用目前无法真正实现，但是接口和框架代

码还是要写上的，以便于将来我们继续扩展。

先来写好运行应用程序的系统调用。按照获取时间的系统调用的实现步骤，首先要写一个系统调用的入口函数，当然同样需要建立一些相应的文件，如代码清单 12-11[xxxx/lmosem/lib/lapithread.c] 所示。

代码清单12-11 运行应用程序的系统调用接口函数

```
hand_t api_exel_thread(void* file,uint_t flgs)
{
    hand_t rethand;
    API_ENTRY_PARE2(SNR_TD_EXEL,rethand,file,flgs);
    return rethand;
}
```

api_exel_thread 函数的返回值类型是 hand_t，其实它就是 long 类型，而 file 参数是应用程序文件名的指针，flgs 是一个整数标志，以后扩展所用。因为 api_exel_thread 函数有两个参数，所以要用 API_ENTRY_PARE2 宏。

接着再来看看运行应用程序的系统调用在 LMOSEM 内核层中的实现，如代码清单 12-12[xxxx/lmosem/kernel/krlsvethread.c] 所示。

代码清单12-12 运行应用程序的系统调用实现

```
sysstus_t krlsvetabl_exel_thread(uint_t swinr,stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_TD_EXEL)
    {
        return SYSSTUSERR;
    }
    return krlsve_exel_thread((void*)stkparv->parmv1,(uint_t)stkparv->parmv2);
}
hand_t krlsve_exel_thread(void* file,uint_t flgs)
{
    //不做任何事，直接返回SYSSTUSERR表示该函数执行失败
    return SYSSTUSERR;
}
```

krlsvetabl_exel_thread 函数要被放在系统调用函数指针数组中对应的位置，而 krlsve_exel_thread 函数直接返回错误状态值，不做任何事。好了，运行应用程序的系统调用的框架代码就实现了。

有了运行应用程序的系统调用，还要有退出应用程序的系统调用，同样也只是实现其框架代码，还是要先实现该系统调用的入口函数，如代码清单 12-13[xxxx/lmosem/lib/lapithread.c] 所示。

代码清单12-13 退出应用程序的系统调用入口函数

```
void api_exit_thread()
{
    uint_t rs, pv1=0;
    API_ENTRY_PARE1(SNR_TD_EXIT, rs, pv1);
    return;
}
```

由于 `api_exit_thread` 函数都是应用程序自身调用的，所以不需要任何参数和返回值，当然这个函数也不会返回了。而其中的 `rs`、`pv1` 变量，只是为了填充 `API_ENTRY_PARE1` 宏的参数。

当然还有退出应用程序的系统调用，在 `LMOSEM` 内核层中的实现，如代码清单 12-14[xxxx/lmosem/kernel/krlsvethread.c] 所示。

代码清单12-14 退出应用程序的系统调用实现

```
sysstatus_t krlsvetabl_exit_thread(uint_t swinr, stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_TD_EXIT)
    {
        return SYSSTUSERR;
    }
    return krlsve_exit_thread();
}

sysstatus_t krlsve_exit_thread()
{
    //不做任何事，直接返回SYSSTUSERR表示该函数执行失败
    return SYSSTUSERR;
}
```

同样，`krlsvetabl_exit_thread` 函数要在系统调用函数指针数组中占用对应的位置，而 `krlsve_exit_thread` 函数依然不做任何实际的工作。

运行和退出应用程序的两个系统调用就实现了，虽然它们都没有完成实际的功能，但是以后可以慢慢扩展实现。下面接着去实现获取进程 ID 的系统调用吧。

12.3.2 获取进程的 ID

在通常情况下，应用程序需要获取自身进程或者其他应用程序进程的 ID 来进一步做其他工作。这里就来实现这个系统调用，不过仍然只是实现获取应用程序自身进程 ID 的系统调用。

获取应用程序自身进程 ID 的方法非常简单，就是首先得到正在运行的进程的描述符，因为调用这个系统调用的应用程序进程，就是正在运行的进程，然后从该进程描述符中读取

ID 值，最后返回这个 ID 值就行了。

先来实现获取进程 ID 这个系统调用的入口函数，如代码清单 12-15[xxxx/lmosem/lib/lapithread.c] 所示。

代码清单12-15 获取进程ID系统调用的入口函数

```
//参数为应用程序的名称，如果参数为NULL就返回应用程序自身的进程ID
hand_t api_retn_threadhand(void* tname)
{
    //进程ID只是一个32位的整数，函数的返回值类型当然可以是hand_t类型
    hand_t rethand;
    API_ENTRY_PARE1(SNR_TD_HAND, rethand, tname);
    return rethand;
}
```

接着再来看看获取进程 ID 这个系统调用。在 LMOSEM 内核层中的实现，如代码清单 12-16[xxxx/lmosem/kernel/krlsvethread.c] 所示。

代码清单12-16 获取进程ID系统调用的实现

```
sysstus_t krlsvetabl_retn_threadhand(uint_t swinr, stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_TD_HAND)
    {
        return SYSSTUSERR;
    }
    return (sysstus_t)krlsve_retn_threadhand((void*)stkparv->parmv1);
}
hand_t krlsve_retn_threadhand(void* tname)
{
    //检查应用程序的名称是否为NULL
    if(tname==NULL)
    {
        //获取当前正在运行的进程的描述符
        thread_t* tdp=krlsched_retn_currthread();
        //返回进程描述符中的ID值
        return (hand_t)(tdp->td_id);
    }
    //其他情况下返回SYSSTUSERR，表示该函数执行失败
    return SYSSTUSERR;
}
```

最后把 krlsvetabl_retn_threadhand 函数放入系统调用函数指针数组中相应的位置，这个系统调用就完成了。

获取进程 ID 的系统调用很简单，所以比较轻松就实现了。关于进程的系统调用就到这里了。

12.4 内存管理系统调用

虽然分配内存空间把应用程序装载进去运行,通常是由操作系统内核完成的。但是应用程序在运行过程中,免不了要动态使用内存空间,所以需要操作系统内核提供分配和释放内存空间的系统调用。下面就去实现这两个系统调用。

通常在比较成熟的操作系统上,分配和释放内存空间的系统调用实际上是操作这个应用程序进程的虚拟内存空间,并非立即分配或者释放物理内存空间,只有应用程序用到这个分配的虚拟内存空间才为其分配物理内存空间,然后通过 MMU 完成虚拟内存空间到物理内存空间的映射。而释放内存空间就是解除该虚拟内存空间和其对应的物理内存空间在 MMU 页表中的映射关系,最后释放对应的物理内存空间即可。这种机制对于应用程序来说是透明的,它使得应用程序不能访问未分配的物理内存空间,更不能访问其他应用程序的物理内存空间,正是因为有了虚拟内存才能保护各个应用程序的代码和数据的完整性,使之不能互相干扰。

可是我们的 LMOSEM 内核没有实现虚拟内存管理组件,所以只能实现分配和释放物理内存空间的系统调用,更不能硬性地保护各个应用程序的内存空间,只能依赖各个应用程序的良好行为,反正应用程序都是我们自己写的,这不会有什么问题,以后实现了虚拟内存管理组件,再来完善这两个系统调用也不迟。

先来实现分配内存空间的系统调用。方法很简单,就是根据分配内存空间的大小,调用 LMOSEM 内核的物理内存管理组件相应的接口函数就行了,依然要先写好该系统调用的入口函数,如代码清单 12-17[xxxx/lmosem/lib/lapimm.c] 所示。

代码清单12-17 分配内存的系统调用入口函数

```
void* api_mallocblk(size_t blksz)
{
    void* retadr;
    API_ENTRY_PARE1(SNR_MM_ALLOC,retadr,blksz);
    //返回分配内存空间的首地址
    return retadr;
}
```

再来看一看分配内存空间的系统调用在 LMOSEM 内核层中的实现,如代码清单 12-18[xxxx/lmosem/kernel/krlsvemm.c] 所示。

代码清单12-18 分配内存的系统调用实现

```
sysstus_t krlsvetabl_mallocblk(uint_t swinr,stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_MM_ALLOC)
    {
        return SYSSTUSERR;
    }
}
```

```

    }
    return (sysstus_t)krlsve_mallocblk((size_t)stkparv->parmvl);
}
void* krlsve_mallocblk(size_t blksz)
{
    //检查分配内存空间的大小
    if(blksz>=0x400000)
    {
        return NULL;
    }
    return krlsve_core_mallocblk(blksz);
}
void* krlsve_core_mallocblk(size_t blksz)
{
    //调用LMOSEM内核的物理内存管理组件的分配内存空间的接口函数
    return (void*)krlnew(blksz);
}

```

这次用了 3 个函数来实现一个系统调用，但是也非常简单，`krlsvetabl_mallocblk` 函数和 `krlsve_mallocblk` 函数不必过多解释。最后在 `krlsve_core_mallocblk` 函数中调用了物理内存管理组件的分配内存空间的接口函数，完成了内存空间的分配。需要注意的是，在 `krlsvetabl_mallocblk` 函数中，`void` 指针类型的数据被强制成 `sysstus_t` 类型的数据返回，其实它们都是 32 位的整数，其中的二进制数据并无区别，只是使用时的性质不一样罢了。

一块内存空间若不使用了，就要释放该内存空间，不然就会造成内存空间泄漏。下面就来实现释放内存空间的系统调用，由于我们在前面已经实现了好几个系统调用，已经非常了解系统调用的实现步骤了，所以这个系统调用我们试着一次性完成，如代码清单 12-19 所示。

代码清单 12-19 释放内存的系统调用

```

//释放内存空间的系统调用的入口函数
sysstus_t api_mfreeblk(void* fradr,size_t blksz)
{
    sysstus_t retstus;
    //释放内存空间时需要内存空间的首地址和其大小这两个参数
    API_ENTRY_PARE2(SNR_MM_FREE,retstus,fradr,blksz);
    return retstus;
}
sysstus_t krlsvetabl_mfreeblk(uint_t swinr,stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_MM_FREE)
    {
        return SYSSTUSERR;
    }
}

```

```

        return krlsve_mfreeblk((void*)stkparv->parmv1, (size_t)stkparv->parmv2);
    }
    sysstus_t krlsve_mfreeblk(void* fradr, size_t blksz)
    {
        //检查参数的合理性
        if(fradr==NULL||blksz>=0x400000)
        {
            return SYSSTUSERR;
        }
        return krlsve_core_mfreeblk(fradr, blksz);
    }
    sysstus_t krlsve_core_mfreeblk(void* fradr, size_t blksz)
    {
        //调用LMOSEM内核的物理内存管理组件的释放内存空间的接口函数
        if(krldelete((adr_t)fradr, blksz)==FALSE)
        {
            //释放内存空间是不允许失败的, 如果失败就死机
            hal_sysdie("krlsve_core_mfreeblk err");
        }
        //返回SYSSTUSOK表示释放内存空间成功
        return SYSSTUSOK;
    }
}

```

在代码清单 12-19 中, 完成释放内存空间的系统调用的核心函数是 `krlsve_core_mfreeblk` 函数, 其他都是框架性代码。

在目前的情况下, 关于内存管理方面, 实现上面两个系统调用就够了, 同时也意味着内存管理的系统调用也实现了。下面继续实现其他系统调用吧!

12.5 设备与文件系统调用

在类 UNIX 操作系统上, 编程操作文件都是先打开文件, 然后读写文件, 最后关闭文件的流程。这一流程不仅仅是可以操作文件, 还可以操作设备。其实这一流程是由一系列的系统调用完成的。本节我们就去实现完成这一流程所需要的系统调用。

12.5.1 设备与文件的打开

对于一个应用程序, 操作文件和设备的第一步, 就是打开对应的文件和设备。文件和设备是操作系统内核管理的重要资源, 所以操作系统内核需要提供打开设备与文件的系统调用, 使得应用程序可以打开相应的设备与文件, 进而使用它们完成相应的功能。

由于打开设备与文件的系统调用, 不仅仅需要打开设备和文件, 还需要有新建文件的功能, 这是一个非常复杂的过程, 因此首先要清楚 LMOSEM 内核执行该系统调用会做些什么,

步骤如下：

- 1) 获取调用该系统调用的应用程序进程描述符的指针，以备后用。
- 2) 根据系统调用的相关参数中的标志位，判断是打开设备还是打开文件。
- 3) 如果是打开设备，就建立 IO 包设置相应的参数，然后把该 IO 包加入到相应进程的资源数组中，最后把该 IO 包发送给相应设备的驱动程序。
- 4) 如果是打开文件，就需要进一步判断是新建文件还是打开文件。
- 5) 如果是新建文件，就建立 IO 包设置相应的参数，然后把该 IO 包发送给文件系统设备的驱动程序，最后直接删除该 IO 包。
- 6) 如果是打开文件，就建立 IO 包设置相应的参数，然后把该 IO 包加入到相应进程的资源数组中，最后把该 IO 包发送给文件系统设备的驱动程序。

为了便于后面的操作，先来实现建立、删除 IO 包和把 IO 包加入到相应进程的资源数组中的函数，如代码清单 12-20 所示。

代码清单12-20 处理IO包的函数

```
//初始化objnode_t结构（IO包）本身
void objnode_t_init(objnode_t* initp)
{
    hal_spinlock_init(&initp->on_lock);
    list_init(&initp->on_list);
    krlsem_t_init(&initp->on_complesem);
    .....
    initp->on_finode=NULL;
    initp->on_extp=NULL;
    return;
}

//建立新的objnode_t结构（IO包）
objnode_t* krlnew_objnode()
{
    //分配objnode_t结构实例变量的内存空间
    objnode_t* ondp=(objnode_t*)krlnew((size_t)sizeof(objnode_t));
    //如果失败就返回NULL
    if(ondp==NULL)
    {
        return NULL;
    }
    //初始化objnode_t结构（IO包）
    objnode_t_init(ondp);
    //返回objnode_t结构（IO包）的地址
    return ondp;
}

//删除objnode_t结构（IO包）
bool_t krldel_objnode(objnode_t* onodep)
```



```

{
    //如果释放objnode_t结构（IO包）实例变量的内存空间失败，就死机
    if(krldel((adr_t)onodep, (size_t)sizeof(objnode_t))==FALSE)
    {
        hal_sysdie("krldel_objnode err");
        return FALSE;
    }
//返回TRUE表示删除objnode_t结构（IO包）成功
    return TRUE;
}

//把objnode_t结构（IO包）加入到相应进程的资源数组中
hand_t krlthd_add_objnode(thread_t* thdp,objnode_t* ondp)
{
    cpuflg_t cpuflg;
//rethd=-1
    hand_t rethd=NO_HAND;
//加锁
    hal_spinlock_saveflg_cli(&thdp->td_lock,&cpuflg);
//扫描进程资源数组中的每一个元素
    for(uint_t hand=0;hand<TD_HAND_MAX;hand++)
    {
        //如果进程资源数组中的第hand个元素的值为NULL
        if(thdp->td_handtbl[hand]==NULL)
        {
            //就记录好该下标
            rethd=(hand_t)hand;
            //并跳转到next_step标号处，继续运行
            goto next_step;
        }
    }
//到这里说明进程资源数组中没有一个元素的值为NULL，所以让rethd=-1返回
    rethd=NO_HAND;
    goto retn_step;
next_step:
    //把objnode_t结构（IO包）的指针放入进程资源数组的第rethd个元素中
    thdp->td_handtbl[rethd]=ondp;
retn_step:
    //解锁
    hal_spinunlock_restflg_sti(&thdp->td_lock,&cpuflg);
//返回rethd
    return rethd;
}

//删除objnode_t结构（IO包）
hand_t krlthd_del_objnode(thread_t *thdp, hand_t hand)
{
    //如果参数不合法就返回NO_HAND

```

```

    if((hand>=TD_HAND_MAX) || (hand<=NO_HAND))
    {
        return NO_HAND;
    }
    cpuflg_t cpuflg;
    hand_t rethd=NO_HAND;
//加锁
    hal_spinlock_saveflg_cli(&thdp->td_lock,&cpuflg);
//如果进程资源数组的第hand个元素中的值为NULL,就说明出问题了
    if(thdp->td_handtbl[hand]==NULL)
    {
        rethd=NO_HAND;
        goto retn_step;
    }
//设置进程资源数组的第hand个元素中的值为NULL
    thdp->td_handtbl[hand]=NULL;
//设置返回值
    rethd=hand;
retn_step:
    //解锁返回
    hal_spinunlock_restflg_sti(&thdp->td_lock,&cpuflg);
    return rethd;
}

```

我们一下子就写好了 5 个基础函数，代码也很简单。有一点要说明，thread_t 结构中的 td_handtbl 数组就是一个 void 指针类型的数组，可以存放任何其他数据结构的指针。

好了，现在就来实现在 LMOSEM 内核层中，打开设备和文件的系统调用的主函数，如代码清单 12-21[xxxx/lmosem/kernel/krlsveopen.c] 所示。

代码清单12-21 打开设备和文件的系统调用的核心函数

```

hand_t krlsve_core_open(void* file,uint_t flgs,uint_t stus)
{
    devid_t devid;
    device_t* devp;
    objnode_t* ondp;
    hand_t rethd;
//获取调用该系统调用的应用程序进程描述符的指针，即当前正在运行的进程
    thread_t* tdp=krlsched_retn_currthread();
//检查flgs中相关的位段，如果是FILE_TY_DEV就跳转到op_dev_step标号处
    if(((flgs&FILE_TY_MASK)==FILE_TY_DEV))
    {
        goto op_dev_step;
    }
//检查flgs中相关的位段，如果是FILE_TY_FILE就跳转到op_fil_step标号处
    if(((flgs&FILE_TY_MASK)==FILE_TY_FILE))

```

```

    {
        goto op_fil_step;
    }
    //返回NO_HAND表示错误
    return NO_HAND;
op_dev_step:
    //打开设备的相关代码
    .....
op_fil_step:
    //打开文件的相关代码
    .....
res_step:
    //处理错误的相关代码
    .....
}

```

从代码清单 12-21 中看到，`krlsve_core_open` 函数的代码结构还是非常清晰的，只是代码比较长，所以要一步一步实现。这里已经实现了获取调用该系统调用的应用程序进程描述符的指针，以及根据系统调用的相关参数中的标志位，判断是打开设备还是打开文件。关于 `flgs` 的位段，后面会进行介绍的。

如果是打开设备，就建立 IO 包设置相应的参数，然后把该 IO 包加入到相应进程的资源数组中，最后把该 IO 包发送给相应设备的驱动程序。明白了方法，下面就用代码去实现。实现打开设备的代码，应该写在 `krlsve_core_open` 函数中的 `op_dev_step` 标号处，如代码清单 12-22[xxxx/lmosem/kernel/krlsveopen.c] 所示。

代码清单12-22 打开设备

```

.....
op_dev_step:
    //获取设备主类型号
    devid.dev_mtype=((devid_t*)file)->dev_mtype;
    //获取设备子类型号
    devid.dev_stype=((devid_t*)file)->dev_stype;
    //获取设备序号
    devid.dev_nr=((devid_t*)file)->dev_nr;
    //根据devid_t结构中的信息查找设备
    devp=krlonidfl_retn_device((void*)(&devid),DIDFIL_IDN);
    //如果没有找到相应的设备，就返回NO_HAND
    if(devp==NULL)
    {
        return NO_HAND;
    }
    //建立IO包(objnode_t结构)
    ondp=krlnew_objnode();
    //建立IO包(objnode_t结构)失败，就返回NO_HAND

```

```

    if(ondp==NULL)
    {
        return NO_HAND;
    }
//设置IO包中的操作码为打开操作
    ondp->on_opcode=IOIF_CODE_OPEN;
//打开类型为设备类型
    ondp->on_objtype=OBJN_TY_DEV;
//打开对象为查找到的设备
    ondp->on_objadr=devp;
//把IO包加入到相应进程的资源数组中，并返回存放IO包指针的数组元素的下标
    rethd=krlthd_add_objnode(tdp,ondp);
//如果加入IO包失败就跳转到res_step标号处
    if(rethd==NO_HAND)
    {
        goto res_step;
    }
//调用设备驱动程序，如果该函数返回SYSSTUSERR，就跳转到res_step标号处
    if(krlsve_open_device(ondp)==SYSSTUSERR)
    {
        goto res_step;
    }
//返回存放IO包指针的数组元素的下标
    return rethd;
.....
sysstus_t krlsve_open_device(objnode_t* ondep)
{
    if(ondep->on_objadr==NULL)
        return SYSSTUSERR;
//真正调用设备驱动程序
    if(krldev_io(ondep)==DFCERRSTUS)
        return SYSSTUSERR;
    return SYSSTUSOK;
}

```

在代码清单 12-22 中，要明白的是，如果应用程序要打开设备，那么该系统调用就要求应用程序准备一个 `devid_t` 结构，通过 `file` 参数传递进来，通过这个 `devid_t` 结构就能知道应用程序要打开的是哪个设备了。这点，LMOSEM 系统与类 UNIX 系统不同，类 UNIX 系统把设备的名字空间映射到文件系统之中了，这样只要打开特殊的文件就可以打开设备了。

打开文件的代码比打开设备的代码要更复杂一些，因为其中还有负责新建文件的代码，所以在代码中还要进一步判断是打开文件还是新建文件，如代码清单 12-23[xxxx/lmosem/kernel/krlsveopen.c] 所示。

代码清单12-23 打开或者新建文件

```

.....
op_fil_step:
    //设置用于查找文件系统设备的devid_t结构
    devid.dev_mtype=FILESYS_DEVICE;
    devid.dev_stype=0;
    devid.dev_nr=0;
//根据devid_t结构中的信息查找设备
    devp=krlonidfl_retn_device((void*)(&devid),DIDFIL_IDN);
//如果没有找到文件系统设备,就返回NO_HAND
    if(devp==NULL)
    {
        return NO_HAND;
    }
//建立IO包(objnode_t结构)
    ondp=krlnew_objnode();
//建立IO包(objnode_t结构)失败,就返回NO_HAND
    if(ondp==NULL)
    {
        return NO_HAND;
    }
//设置IO包中的操作码为打开操作
    ondp->on_opercode=IOIF_CODE_OPEN;
//打开类型为文件类型
    ondp->on_objtype=OBJN_TY_FIL;
//打开对象为查找到的设备,即文件系统设备
    ondp->on_objadr=devp;
//检查flgs中相关的位段,看一看是否为新建文件操作
    if((flgs>>RWO_FLG_BITS)&RWO_FLG_MASK)==NF_FLG)
    {
        //设置IO包中的访问标志为建立文件标志
        ondp->on_acsflgs=FSDEV_OPENFLG_NEWFILE;
        //设置IO包中的文件名为file
        ondp->on_fname=file;
        //调用文件系统设备驱动程序
        if(krlsve_open_device(ondp)==SYSSTUSERR)
        {
            goto res_step;
        }
        //删除IO包
        if(krldel_objnode(ondp)==FALSE)
        {
            hal_sysdie("NEWFILE api del objnode err");
        }
        //建立文件成功时返回0
        return 0;
    }

```

```

    }
// 否则就是打开文件操作
    else
    {
        // 设置IO包中的访问标志为打开文件标志
        ondp->on_acsflgs=FSDEV_OPENFLG_OPEFILE;
    }
// 设置IO包中的文件名为file
    ondp->on_fname=file;
// 把IO包加入到相应进程的资源数组中, 并返回存放IO包指针的数组元素的下标
    rethd=krlthd_add_objnode(tdp, ondp);
    if(rethd==NO_HAND)
    {
        goto res_step;
    }
// 调用文件系统设备驱动程序
    if(krlsve_open_device(ondp)==SYSSTUSERR)
    {
        goto res_step;
    }
// 打开文件操作成功时返回存放IO包指针的数组元素的下标
    return rethd;
res_step:
.....

```

因为文件系统也是一个设备, 所以这里要自己设置 `devid_t` 结构中的信息并查找文件系统设备。然后是检查 `flgs` 中相关的位段, 判断是新建文件操作还是打开文件操作。如果是新建文件操作, 则 IO 包不用加入到相应进程的资源数组中, 因为在新建文件操作完成之后, 就没有后续操作了, IO 包用完之后就没有用了, 所以最后要直接调用 `krlidel_objnode` 函数删除 IO 包。对于打开文件操作则不同, 因为打开文件就是为了后面对文件进行读写, 所以它就需要把 IO 包加入到相应进程的资源数组中, 并返回存放 IO 包指针的数组元素的下标。

这个打开设备和文件的系统调用的主函数终于完成了, 但是别忘了还有该系统调用在 LMOSEM 内核层中的接口函数和在应用程序中调用的入口函数, 如代码清单 12-24 所示。

代码清单12-24 打开设备、文件系统调用的入口函数及其在内核层的接口函数

```

hand_t api_open(void* file, uint_t flgs, uint_t stus)
{
    hand_t rethand;
    API_ENTRY_PARE3(SNR_FS_OPEN, rethand, file, flgs, stus);
    return rethand;
}

sysstus_t krlsvetabl_open(uint_t swinr, stkparame_t* stkparv)
{
    if(swinr!=SNR_FS_OPEN)

```

```

    {
        return SYSSTUSERR;
    }
    return (sysstus_t)krlsve_open((void*)stkparv->parmv1, (uint_t)stkparv->parmv2,
        (uint_t)stkparv->parmv3);
}
hand_t krlsve_open(void* file, uint_t flgs, uint_t stus)
{
    if (file==NULL)
    {
        return SYSSTUSERR;
    }
    return krlsve_core_open(file, flgs, stus);
}

```

到这里，打开设备和文件的系统调用就实现完成了。由于复杂度很高，导致我们花了大量的篇幅。不过这是所有系统调用中最复杂的一个，实现了这个系统调用，后面的就容易多了。下面就继续实现其他系统调用吧！

12.5.2 设备与文件的关闭

前面刚刚实现了最复杂的打开设备和文件的系统调用，这里就来实现关闭设备和文件的系统调用。

对于打开设备和文件的系统调用，它是分配 IO 包以及其他资源，然后调用设备驱动程序的打开函数，然而，设备驱动程序可能分配更多的资源，这些资源最终用 IO 包和一个应用程序进程联系起来。而关闭设备和文件的系统调用就是执行其逆过程，它执行的步骤如下：

- 1) 获取调用该系统调用的应用程序进程描述符的指针，以备后用。
- 2) 根据该系统调用的相关参数，获取相应进程资源数组中特定的 IO 包。
- 3) 调用设备驱动程序的关闭函数，让设备驱动程序释放相关的资源。
- 4) 如果设备驱动程序的关闭函数执行正确，就释放并删除 IO 包。

接下来，就用实际的代码来实现上述步骤，如代码清单 12-25[xxxx/lmosem/kernel/krlsveclose.c] 所示。

代码清单12-25 关闭设备和文件的系统调用的核心函数

```

sysstus_t krlsve_core_close(hand_t fhand)
{
    //获取调用该系统调用的应用程序进程描述符的指针
    thread_t* tdp=krlsched_retn_currthread();
    //根据fhand参数，获取相应进程资源数组中特定的IO包
    objnode_t* ondp=krlthd_retn_objnode(tdp, fhand);
    //获取IO包失败

```

```

    if(ondp==NULL)
    {
        return SYSSTUSERR;
    }
    //只要IO包的打开类型是文件类型或设备类型,就设置IO包中的操作码为关闭操作
    if(ondp->on_objtype==OBJN_TY_DEV||ondp->on_objtype==OBJN_TY_FIL)
    {
        ondp->on_opercode=IOIF_CODE_CLOSE;
        goto cl_dev_step;
    }
    return SYSSTUSERR;
cl_dev_step:
    //调用设备驱动程序
    if(krlsve_close_device(ondp)==SYSSTUSERR)
    {
        return SYSSTUSERR;
    }
    //释放进程资源数组中的IO包失败时,就死机
    if(krlthd_del_objnode(tdp,fhand)==NO_HAND)
    {
        hal_sysdie("close api td del objnode err");
    }
    //删除IO包失败时,就死机,这里是释放IO包本身占用的内存空间
    if(krldel_objnode(ondp)==FALSE)
    {
        hal_sysdie("close api del objnode err");
    }
    return SYSSTUSOK;
}
sysstus_t krlsve_close_device(objnode_t* ondep)
{
    if(ondep->on_objadr==NULL)
        return SYSSTUSERR;
    //调用相应的设备驱动程序
    if(krldev_io(ondep)==DFCERRSTUS)
        return SYSSTUSERR;
    return SYSSTUSOK;
}

```

可以看到, `krlsve_core_close` 函数唯一的参数就是 `fhand`, 其实它就是调用打开设备和文件的系统调用后的返回值。在 Windows 中它被称为句柄, 在 Linux 中它被称为文件描述符, 其实我们知道它就是进程资源数组的下标。注意一定要先调用设备驱动程序的关闭函数, 让其在内部先释放它所分配的资源, 最后才能释放和删除 IO 包。

最后来实现该系统调用在 LMOSEM 内核层中的接口函数和在应用程序中调用的入口函数, 如代码清单 12-26 所示。

代码清单12-26 关闭设备和文件系统调用的入口函数及其在内核层的接口函数

```

sysstus_t api_close(hand_t fhand)
{
    sysstus_t rets;
    API_ENTRY_PARE1(SNR_FS_CLOSE, rets, fhand);
    return rets;
}

sysstus_t krlsvetabl_close(uint_t swinr, stkparame_t* stkparv)
{
    if(swinr!=SNR_FS_CLOSE)
    {
        return SYSSTUSERR;
    }
    return krlsve_close((hand_t)stkparv->parmvl);
}

sysstus_t krlsve_close(hand_t fhand)
{
    if(fhand<0||fhand>=TD_HAND_MAX)
    {
        return SYSSTUSERR;
    }
    return krlsve_core_close(fhand);
}

```

好了，又实现了一个系统调用。下面就要去实现设备与文件的读写系统调用了。

12.5.3 设备与文件的读写

设备与文件的读写系统调用，其实是两个不同的系统调用，一个是对设备和文件的读系统调用，另一个是对设备和文件的写系统调用，这里将放在一起实现。

设备与文件的读写这两个系统调用，其实最后都是调用实际的设备驱动程序或者文件系统设备驱动程序的读写函数而工作的。设备和文件的读系统调用，就是把设备或文件中的数据读取到应用程序指定的缓冲区中。而设备和文件的写系统调用正好相反，它是把应用程序的缓冲区中的数据，写入到具体的设备或文件中去的。注意，在调用这两个系统调用之前，一定要确保相应的设备和文件已经打开了。

先来实现设备和文件的读系统调用，这里仍然先实现该系统调用在 LMOSEM 内核层中的核心函数，如代码清单 12-27[xxxx/lmosem/kernel/krlsveread.c] 所示。

代码清单12-27 读设备、文件的系统调用的核心函数

```

sysstus_t krlsve_core_read(hand_t fhand, buf_t buf, size_t len, uint_t flgs)
{
    //获取调用该系统调用的应用程序进程描述符的指针
    thread_t* currttd=krlsched_retn_currthread();

```

```

//根据fhand参数,获取相应进程资源数组中特定的IO包
objnode_t* onp=krlthd_retn_objnode(currtd,fhand);
//获取IO包失败,则返回SYSSTUSERR
if (onp==NULL)
{
    return SYSSTUSERR;
}
if (onp->on_objtype==OBJN_TY_DEV||onp->on_objtype==OBJN_TY_FIL)
{
//设置IO包中的操作码为读操作
onp->on_opercode=IOIF_CODE_READ;
//设置存放数据的缓冲区
onp->on_buf=buf;
//设置读取数据的长度
onp->on_len=len;
//设置缓冲区的大小,固定为4KB
onp->on_bufsz=0x1000;
//设置访问标志
onp->on_acsflgs=flgs;
//调用设备驱动程序
return krlsve_read_device(onp);
}
return SYSSTUSERR;
}
sysstus_t krlsve_read_device(objnode_t* ondep)
{
    if (ondep->on_objadr==NULL)
        return SYSSTUSERR;
//调用设备驱动程序
if (krldev_io(ondep)==DFCERRSTUS)
    return SYSSTUSERR;
    return SYSSTUSOK;
}

```

krlsve_core_read 函数依然是首先获取调用该系统调用的应用程序进程描述符的指针,接着获取相应进程资源数组中特定的 IO 包,然后设置 IO 包中的一些相关域,最后把这个 IO 包发送给相应的设备驱动程序,在相应的设备驱动程序中完成实际的读操作。注意,应用程序分配的缓冲区大小一定要是 4KB 大小,因为我们文件系统的实现,要求是 4KB 大小的缓冲区,加上这里为了简化实现,就强制写为 4KB。这个函数就是把设备或文件中的数据读取 len 个字节到 buf 指向的缓冲区中。

接着来实现该系统调用在 LMOSEM 内核层中的接口函数和在应用程序中调用的入口函数,如代码清单 12-28 所示。

代码清单12-28 读设备、文件系统调用的入口函数及其在内核层的接口函数

```

sysstatus_t api_read(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    sysstatus_t rets;
    //4个参数
    API_ENTRY_PARE4(SNR_FS_READ,rets,fhand,buf,len,flgs);
    return rets;
}
sysstatus_t krlsvetabl_read(uint_t swinr,stkparame_t* stkparv)
{
    if(swinr!=SNR_FS_READ)
    {
        return SYSSTUSERR;
    }
    return krlsve_read((hand_t)stkparv->parmv1,(buf_t)stkparv->parmv2,
        (size_t)stkparv->parmv3,(uint_t)stkparv->parmv4);
}
sysstatus_t krlsve_read(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    //检查参数是否合理
    if(fhand==NO_HAND||buf==NULL||len<1)
    {
        return SYSSTUSERR;
    }
    return krlsve_core_read(fhand,buf,len,flgs);
}

```

设备和文件的写系统调用与设备和文件的读系统调用，本质上没什么不同，只不过数据流向正好相反。设备和文件的写系统调用是把应用程序的缓冲区中的数据，写入到具体的设备或文件中去的，数据流向是由外向内的。下面就用代码去实现该系统调用的核心函数，如代码清单 12-29[xxxx/lmosem/kernel/krlsvewrite.c] 所示。

代码清单12-29 写设备、文件的系统调用的核心函数

```

sysstatus_t krlsve_core_write(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    //获取调用该系统调用的应用程序进程描述符的指针
    thread_t* currtid=krlsched_retn_currthread();
    //根据fhand参数，获取相应进程资源数组中特定的IO包
    objnode_t* onp=krlthd_retn_objnode(currtid,fhand);
    //获取IO包失败，则返回SYSSTUSERR
    if(onp==NULL)
    {
        return SYSSTUSERR;
    }
    if(onp->on_objtype==OBJN_TY_DEV||onp->on_objtype==OBJN_TY_FIL)

```

```

{
//设置IO包中的操作码为写操作
    onp->on_opcode=IOIF_CODE_WRITE;
//设置存放数据的缓冲区
    onp->on_buf=buf;
//设置读取数据的长度
    onp->on_len=len;
//设置缓冲区的大小, 固定为4KB
    onp->on_bufsz=0x1000;
//设置访问标志
    onp->on_acsflgs=flgs;
//调用设备驱动程序
    return krlsve_write_device(onp);
}
return SYSSTUSERR;
}
sysstus_t krlsve_write_device(objnode_t* ondep)
{
    if (ondep->on_objadr==NULL)
        return SYSSTUSERR;
//调用设备驱动程序
    if (krldev_io(ondep)==DFCERRSTUS)
        return SYSSTUSERR;
    return SYSSTUSOK;
}

```

krlsve_core_write 函数完成的工作就是把 buf 指向的缓冲区中的数据, 写 len 个字节到设备或文件中去。下面来实现该系统调用在 LMOSEM 内核层中的接口函数和在应用程序中调用的入口函数, 如代码清单 12-30 所示。

代码清单12-30 写设备、文件系统调用的入口函数及其在内核层的接口函数

```

sysstus_t api_write(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    sysstus_t rets;
//4个参数
    API_ENTRY_PARE4(SNR_FS_WRITE,rets,fhand,buf,len,flgs);
    return rets;
}
sysstus_t krlsvetabl_write(uint_t swinr,stkparame_t* stkparv)
{
    if (swinr!=SNR_FS_WRITE)
    {
        return SYSSTUSERR;
    }
    return krlsve_write((hand_t)stkparv->parmv1,(buf_t)stkparv->parmv2,
        (size_t)stkparv->parmv3,(uint_t)stkparv->parmv4);
}

```

```

}
sysstatus_t krlsve_write(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
//检查参数是否合理
    if(fhand<=NO_HAND||fhand>=TD_HAND_MAX||buf==NULL||len==0)
    {
        return SYSSTUSERERR;
    }
    return krlsve_core_write(fhand,buf,len,flgs);
}

```

本节到这里实现了设备与文件的读写这两个系统调用。下面就去实现最后一个系统调用。

12.5.4 设备与文件的控制

设备与文件除了打开之后进行读写操作之外，还有一些其他操作，例如，调整音频设备的音量，把一个文件删除等。这些操作归纳起来就称为设备与文件的控制操作，所以为此要专门实现一个用于控制设备和文件的系统调用。

用于控制设备和文件的系统调用，其实最后也是调用实际的设备驱动程序或者文件系统设备驱动程序的控制函数而工作的。由于设备和文件有很多其他操作，所以该系统调用需要一个控制码，把这个控制码传递给设备驱动程序或者文件系统设备驱动程序加于区分是哪个操作。下面就来实现该系统调用的核心函数，如代码清单 12-31[xxxx/lmosem/kernel/krlsveioctl.c] 所示。

代码清单12-31 控制设备、文件的系统调用的核心函数

```

sysstatus_t krlsve_core_ioctl(hand_t fhand,buf_t buf,uint_t iocode,uint_t flgs)
{
//获取调用该系统调用的应用程序进程描述符的指针
    thread_t* currttd=krlsched_retn_currthread();
//根据fhand参数，获取相应进程资源数组中特定的IO包
    objnode_t* onp=kralthd_retn_objnode(currttd,fhand);
//获取IO包失败，则返回SYSSTUSERERR
    if(onp==NULL)
    {
        return SYSSTUSERERR;
    }
    if(onp->on_objtype==OBJN_TY_DEV||onp->on_objtype==OBJN_TY_FILE)
    {
//设置IO包中的操作码为控制操作
        onp->on_opercode=IOIF_CODE_IOCTL;
//将设置存放数据的缓冲区
        onp->on_buf=buf;
//设置控制码

```

```

        onp->on_ioctrd=iocode;
        //设置访问标志
        onp->on_acsflgs=flgs;
//将调用设备驱动程序
        return krlsve_ioctl_device(onp);
    }
    return SYSSTUSERR;
}
sysstus_t krlsve_ioctl_device(objnode_t* ondep)
{
    if(ondep->on_objadr==NULL)
        return SYSSTUSERR;
//调用设备驱动程序
    if(krldev_io(ondep)==DFCERRSTUS)
        return SYSSTUSERR;
    return SYSSTUSOK;
}

```

krlsve_core_ioctl 函数，同样是首先获取调用该系统调用的应用程序进程描述符的指针，接着获取相应进程资源数组中特定的 IO 包，然后设置 IO 包中的一些相关域，最后把这个 IO 包发送给相应的设备驱动程序，在相应的设备驱动程序中完成实际的控制操作。然而对于控制操作，不仅仅是需要控制码，因为有些控制操作需要额外的数据，所以这里会传递一个缓冲区指针，需要相应的数据就放在缓冲区中，否则就把 buf 设为 NULL。所有的设备和文件系统的控制码和相应的数据结构，都要发布给应用程序开发者。这样应用程序开发者才能开发相应的应用程序。

接着还是不厌其烦地来实现该系统调用在 LMOSEM 内核层中的接口函数和在应用程序中调用的入口函数吧，如代码清单 12-32 所示。

代码清单12-32 控制设备、文件系统调用的入口函数及其在内核层的接口函数

```

sysstus_t api_ioctl(hand_t fhand,buf_t buf,uint_t iocode,uint_t flgs)
{
    sysstus_t rets;
//4个参数
    API_ENTRY_PARE4(SNR_FS_IOCTL,rets,fhand,buf,iocode,flgs);
    return rets;
}
sysstus_t krlsvetabl_ioctl(uint_t swinr,stkparame_t* stkparv)
{
    //检查系统调用号
    if(swinr!=SNR_FS_IOCTL)
    {
        return SYSSTUSERR;
    }
    return krlsve_ioctl((hand_t)stkparv->parmv1,(buf_t)stkparv->parmv2,

```

```

        (uint_t) stkpargv->parmv3, (uint_t) stkpargv->parmv4);
    }
    sysstatus_t krlsve_ioctl(hand_t fhand, buf_t buf, uint_t iocode, uint_t flgs)
    {
        //检查参数是否合理
        if(fhand==NO_HAND)
        {
            return SYSSTUSERR;
        }
        return krlsve_core_ioctl(fhand, buf, iocode, flgs);
    }

```

其实还有一个系统调用，是调整设备和文件当前数据的读写位置的，但是不管是我们的设备驱动程序还是文件系统设备驱动程序，都没有实现这个功能，所以在此就不过多介绍了，但是其框架代码还是实现了，请读者自行参阅代码。

这是我们实现的最后一个系统调用。至此，LMOSEM 内核的全部系统调用就实现完了，下面就要去实现应用程序库了。

12.6 应用程序库

通常情况下，应用程序中调用的函数都不是系统调用，而是一些库函数。库函数是对系统调用的封装。有的库函数是直接调用相应的系统调用，而有的库函数为了完成特定的功能，则调用了几个相应的系统调用，还有一些库函数完成的功能不需要调用相应的系统调用。应用程序库的形式有几种。一种是动态库，它不占用应用程序文件的大小，只有当应用程序运行时调用到它，它才被操作系统内核装入内存之中，然后和应用程序链接起来一起工作。我们的 LMOSEM 内核还不支持动态库。还有一种是静态库，它会和应用程序链接在一起。本节就去实现一些静态库。

我们的应用程序库分为时间库、进程库、内存库、标准输入/输出库这几种类型。其中大部分库函数是直接对相应系统调用的封装。好了，下面就去实现它们。

库函数

如上所述，下面将要实现时间库、进程库、内存库、标准输入输出库，并且我们实现的都是静态库，即最后要和应用程序链接在一起的库。

首先来实现标准输入输出库，这类库中主要包括文件与设备的打开与关闭、读写、控制等操作的函数，如代码清单 12-33 所示。

代码清单12-33 标准输入输出库函数

```

//打开设备与文件或者新建文件的库函数
hand_t open(void* file, uint_t flgs, uint_t stus)

```

```

{
    //调用相应系统调用的入口函数
    hand_t rethand=api_open(file,flgs,stus);
    return rethand;
}
//关闭设备与文件的库函数
sysstus_t close(hand_t fhand)
{
    //调用相应系统调用的入口函数
    sysstus_t rets=api_close(fhand);
    return rets;
}
//读设备与文件的库函数
sysstus_t read(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    //调用相应系统调用的入口函数
    sysstus_t rets=api_read(fhand,buf,len,flgs);
    return rets;
}
//写设备与文件的库函数
sysstus_t write(hand_t fhand,buf_t buf,size_t len,uint_t flgs)
{
    //调用相应系统调用的入口函数
    sysstus_t rets=api_write(fhand,buf,len,flgs);
    return rets;
}
//控制设备与文件的库函数
sysstus_t ioctl(hand_t fhand,buf_t buf,uint_t iocode,uint_t flgs)
{
    //调用相应系统调用的入口函数
    sysstus_t rets=api_ioctl(fhand,buf,iocode,flgs);
    return rets;
}

```

可以看到，代码清单 12-33 中的一些函数都有一个 flgs 参数，这个参数是一个 32 位的整数。其中的 0～3 位表示操作的是设备还是文件；4～15 位表示访问设备与文件的方式，是读写还是只读或者是新建；16～31 位保留为 0，留作以后使用。

接着来实现时间库，这个库中主要包括如获取系统时间、设置 RTC 时间、获取 RTC 时间等操作的函数。这些库函数将依赖前面所实现的标准输入输出库函数，如代码清单 12-34 所示。

代码清单12-34 时间库函数

```

//获取系统时间的库函数
sysstus_t time(times_t* ttime)
{

```



```

    //调用相应系统调用的入口函数
    sysstatus_t rets=api_time(ttime);
    return rets;
}
//设置RTC时间的库函数
sysstatus_t settime(times_t* ttime)
{
    //把ttime指向的times_t结构中的数据, 设置到RTC设备中, 同时更新时间
    if(ttime==NULL)
    {
        return SYSSTUSERR;
    }
    devid_t dev;
    dev.dev_mtype=RTC_DEVICE;
    dev.dev_stype=0;
    dev.dev_nr=0;
    //打开RTC设备
    hand_t fd=open(&dev, RW_FLG|FILE_TY_DEV, 0);
    if(fd==-1)
    {
        return SYSSTUSERR;
    }
    //调用控制设备与文件的库函数
    ioctl(fd, ttime, IOCTRCODE_SETTIME, 0);
    //关闭RTC设备
    close(fd);
    return SYSSTUSOK;
}
//获取RTC时间的库函数
sysstatus_t gettime(times_t* ttime)
{
    //把RTC设备中的时间数据, 读取到ttime指向的times_t结构中
    if(ttime==NULL)
    {
        return SYSSTUSERR;
    }
    devid_t dev;
    dev.dev_mtype=RTC_DEVICE;
    dev.dev_stype=0;
    dev.dev_nr=0;
    //打开RTC设备
    hand_t fd=open(&dev, RW_FLG|FILE_TY_DEV, 0);
    if(fd==-1)
    {
        return SYSSTUSERR;
    }

```

```
//读RTC设备
    read(fd, &times, sizeof(times_t), 0);
//关闭RTC设备
    close(fd);
    return SYSSTUSOK;
}
```

代码清单 12-34 中的三个函数都是在“xxxx/lmosem/lib/libtime.c”文件中实现的。代码中的 `times_t` 数据结构，是在 LMOSEM 内核的相关代码中定义的，但是必须发布给应用程序开发者，而 `IOCTRCODE_SETTIME`、`RTC_DEVICE` 宏和 `devid_t` 数据结构，是在驱动模型的相关代码中定义的，也必须发布给应用程序开发者。

再接下来就是内存库，内存库中包括两个函数，一个是内存空间的分配函数，另一个是内存空间的释放函数，如代码清单 12-35 所示。

代码清单12-35 内存库函数

```
//分配内存空间的库函数
void* mallocblk(size_t blksz)
{
    //调用相应系统调用的入口函数
    void* retadr=api_mallocblk(blksz);
    return retadr;
}
//释放内存空间的库函数
sysstus_t mfreeblk(void* fradr, size_t blksz)
{
    //调用相应系统调用的入口函数
    sysstus_t retstus=api_mfreeblk(fradr, blksz);
    return retstus;
}
```

内存库中的两个函数很简单，就是直接调用相应系统调用的入口函数。所以不必过多详述。

然后来实现关于进程方面的库函数，其中有进程的执行与退出的库函数。其实，这两个库函数是不会实际工作的，因为它们在 LMOSEM 内核中，对应的系统调用没有真正实现，当然还有获取进程 ID 的库函数，如代码清单 12-36 所示。

代码清单12-36 进程库函数

```
//执行进程的库函数
hand_t exel(void* file, uint_t flgs)
{
    //调用相应系统调用的入口函数
    hand_t rethand=api_exel_thread(file, flgs);
    return rethand;
}
```

```

}
//退出进程的库函数
void exit()
{
    //调用相应系统调用的入口函数
    api_exit_thread();
    return;
}
//获取进程ID的库函数
hand_t pid(void* tname)
{
    //调用相应系统调用的入口函数
    hand_t rethand=api_retn_threadhand(tname);
    return rethand;
}

```

最后让我们来完成一个编程初学者用得最多的函数，那就是 `printf` 函数。虽然我们在前面的 LMOSEM 内核代码中实现了 `printfk` 函数，它们的原理是相同的，不过这里实现的是库函数，在实现方式上有所不同。在 LMOSEM 内核中我们可以直接读写相关设备的寄存器，完成字符串的输出功能，然而在这里却不行，如代码清单 12-37 所示。

代码清单12-37 打印字符串库函数

```

int printf(const char* fmt,...)
{
    int rets=-1;
    va_list ap;
    //获取在栈中的fmt参数的地址
    va_start(ap,fmt);
    //分配4KB大小的内存空间
    char* buf=(char*)mallocblk(0x1000);
    if(buf==NULL)
    {
        return -1;
    }
    devid_t dev;
    dev.dev_mtype=UART_DEVICE;
    dev.dev_stype=0;
    dev.dev_nr=0;
    //打开串口设备
    hand_t fd=open(&dev,RW_FLG|FILE_TY_DEV,0);
    if(fd==-1)
    {
        rets=-1;
        goto res_step;
    }
}

```

//把要打印输出的数据进行格式化处理后, 保存到buf指向的内存空间中

```
vsprintf(buf,fmt,ap);
```

//把buf指向的内存空间中的数据写入到串口设备中去

```
if(write(fd,buf,strlen(buf),0)==SYSSTUSERR)
```

```
{
```

```
    rets=-1;
```

```
    goto res_step;
```

```
}
```

//关闭串口设备

```
close(fd);
```

```
rets=0;
```

```
res_step:
```

//释放先前分配的4KB内存空间

```
if(mfreeblk(buf,0x1000)==SYSSTUSERR)
```

```
{
```

```
    rets=-1;
```

```
}
```

```
va_end(ap);
```

```
return rets;
```

```
}
```

```
char* strcpy(char* buf,char* str_s)
```

```
{
```

```
    while(*str_s)
```

```
    {
```

```
        *buf=*str_s;
```

```
        buf++;
```

```
        str_s++;
```

```
    }
```

```
    return buf;
```

```
}
```

```
void vsprintf(char* buf,const char* fmt,va_list args)
```

```
{
```

```
    char* p =buf;
```

```
    va_list next_arg=args;
```

```
    while(*fmt)
```

```
    {
```

```
        if(*fmt != '%')
```

```
        {
```

```
            *p++ = *fmt++;
```

```
            continue;
```

```
        }
```

```
        .....
```

```
    }
```

```
    *p=0;
```

```
    return;
```

```
}
```

```

char* number(char* str, uint_t n, sint_t base)
{
    register char *p;
    char strbuf[36];
    p = &strbuf[36];
    .....
    return str;
}

```

为了节省篇幅，代码清单 12-37 中，有几个函数的代码省略了，因为前面已经对 printf 这种变参函数的实现原理非常了解了。这里不同的是，首先调用 mallocblk 函数分配了一个 4KB 大小的内存空间，作为数据的缓冲区，然后调用 vsprintf 函数把要打印显示的数据按照特定的格式，进行格式化处理后，保存到缓冲区中，最后调用 write 函数把这个缓冲区中的数据写入到串口设备中，就完成了打印显示相应数据的功能。

至此，所有的系统调用和所有的库函数就实现了。有的库函数是直接对系统调用的封装，而有的库函数调用了其他库函数或者调用了多个系统调用，既然都已经实现了，可是这些代码都正确无误吗？下面就去测试一下。

12.7 测试

LMOSEM 内核的接口层实现了，这意味着 LMOSEM 内核的开发终于接近尾声了，唯一担心的是，我们亲自用许多代码实现的那么多的功能机制是否工作正常。

当然，LMOSEM 内核的功能机制是否工作正常，最简单的方法就是测试。所不同的是，这里不仅仅是对本章的系统调用和应用程序库函数进行测试，还要对 LMOSEM 内核做全面的功能性测试。这里的测试，需要用应用程序来测试，所以先写两个应用程序来测试系统调用和库函数，然后用很多个应用程序来做全面的功能性测试。

前面刚刚实现了系统调用和应用程序库，首先要确定这两个部分的代码是否正确。代码是否正确，要上物理计算平台运行了之后才知道，这是我们一贯的作风，这里也不例外。要测试系统调用和应用程序库，我们只要让应用程序调用它们然后看结果就行了。

系统调用和应用程序库，它们属于 LMOSEM 内核的接口层，而调用它们的是应用程序，所以这里就去写两个应用程序，让 LMOSEM 内核建立两个进程去运行，如代码清单 12-38[xxxx/lmosem/task/utask.c] 所示。

代码清单12-38 两个应用程序主函数及读写文件的函数

```

//建立文件并写入相关数据
void usr_write_file()
{
    //分配4KB大小的缓冲区
    char* buf=(char*)mallocblk(0x1000);

```

```

    if(buf==NULL)
        goto err;
    //在缓冲区中写入512个0xff
    for(int i=0;i<512;i++)
        buf[i]=0xff;
    //新建文件file.bin
    if(open("/file.bin",NF_FLG|FILE_TY_FILE,0)==-1)
        goto err;
    //打开文件file.bin
    hand_t fd=open("/file.bin",RW_FLG|FILE_TY_FILE,0);
    if(fd==-1)
        goto err;
    //把缓冲区中的前512B写入到文件file.bin中
    if(write(fd,buf,512,0)==SYSSTUSERR)
        goto err;
    //关闭文件file.bin
    close(fd);
    //释放4KB大小的缓冲区
    if(mfreeblk(buf,0x1000)==SYSSTUSERR)
        goto err;
    return;
//出错处理
err:
    printf("usr_write_file err\n\r");
//死循环
    for(;;);
    return;
}
void usr_read_file()
{
    //分配4KB大小的缓冲区
    char* buf=(char*)mallocblk(0x1000);
    if(buf==NULL)
        goto err;
    //清零分配的缓冲区
    for(int i=0;i<0x1000;i++)
        buf[i]=0;
    //打开文件file.bin
    hand_t fd=open("/file.bin",RW_FLG|FILE_TY_FILE,0);
    if(fd==-1)
        goto err;
    //读取file.bin文件中的前512B的数据到缓冲区中
    if(read(fd,buf,512,0)==SYSSTUSERR)
        goto err;
    //检查缓冲区中的前512B的数据是不是都是0xff
    for(int j=0;j<512;j++)

```

```

    {
        if(buf[j]!=0xff)
            goto err;
    }
//关闭文件file.bin
    close(fd);
//释放4KB大小的缓冲区
    if(mfreeblk(buf,0x1000)==SYSSTUSERR)
        goto err;
    return;
//出错处理
err:
    printf("usr_read_file err\n\r");
    for(;;);
    return;
}
void task1_main()
{
    int i=0;
    times_t timet;
//这两个函数执行成功表示文件的建立、读写都是正常的，即那些库函数和系统调用的代码也是正确的
    usr_write_file();
    usr_read_file();
    for(;;)
    {
        i++;
        printf("task1 run %x task1ID:%x\n\r",i,pid(NULL));
        //获取系统时间
        time(&timet);
        printf("year:%d mon:%d date:%d hour:%d min:%d sec:%d\n\r",
            timet.year,timet.mon,timet.date,timet.hour,timet.min,timet.sec);
    }
    return;
}
void task0_main()
{
    times_t timet;
    int i=0;
    for(;;)
    {
        i++;
        printf("task0 run %x task0ID:%x \n\r",i,pid(NULL));
//获取RTC设备的时间
        gettimeofday(&timet);
        printf("year:%d mon:%d date:%d hour:%d min:%d sec:%d\n\r",
            timet.year,timet.mon,timet.date,timet.hour,timet.min,timet.sec);
    }
}

```

```

    }
    return;
}

```

为了测试文件的建立、读写，另外实现了两个函数 `usr_write_file` 和 `usr_read_file`。如果这两个函数执行成功并且返回，就表示文件的建立、读写都是正常的。当然，也间接证明了那些标准输入输出库函数、内存库函数和相应的系统调用都是正确的。而且在两个应用程序中，分别又测试了获取系统时间和获取 RTC 设备时间的库函数。`printf` 函数就不用多说了。并且这些函数的底层又互相调用其他库函数，所以这种测试是接近真实环境的。

按照前面的方法，编译并链接好 LMOSEM 内核，接着连接好我们的开发板，然后打开开发板的电源，最后把 LMOSEM 内核文件下载到开发板中运行。正常情况下，应该如图 12-1 所示。

```

深度终端
cpuidle is run!!
task1 run 1 taskID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task0 run 1 taskID:3041103c
year:2026 mon:1 date:22 hour:0 min:44 sec:2
task0 run 2 taskID:3041103c
year:2026 mon:1 date:22 hour:0 min:44 sec:2
task0 run 3 taskID:3041103c
year:2026 mon:1 date:22 hour:0 min:44 sec:2
task0 run 4 taskID:3041103c
year:2026 mon:1 date:22 hour:0 min:44 sec:2
task0 run 5 taskID:3041103c
year:2026 mon:1 date:22 hour:0 min:44 sec:2
task1 run 2 taskID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 3 taskID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 4 taskID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 5 taskID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 6 taskID:304110c0

```

图 12-1 两个进程测试文件读写和获取系统时间

从图 12-1 中，清楚地看到系统中第一个运行的进程是空转进程。接着进程 1、进程 0 也开始交替运行，并且进程 1 成功地执行了上述两个操作文件的函数，说明文件操作是成功的。就是时间好像不对，对于进程 1 中获取系统时间的数据都是 0，那是因为 RTC 设备还没有产生 `rtc_tick` 中断，所以 LMOSEM 内核的时间数据还没有被更新。那么进程 0 中是直接获取 RTC 设备的时间，为什么好像也有问题，那是因为笔者用的开发板上的 RTC 设备的电池没有电导致的。

接着我们来增加 4 个应用程序，看看更多任务下有没有问题。在新增的应用程序中，大部分都只是获取并输出自身对应的进程 ID。不过其中有一个应用程序设置了新的时间，代码在 `utask.c` 文件中，如代码清单 12-39[xxxx/lmosem/task/utask.c] 所示。

代码清单 12-39 多个应用程序

```

void task2_main()
{
    times_t timet;

```



```

    timet.year=14;
    timet.mon=9;
    timet.day=4;
    timet.date=11;
    timet.hour=12;
    timet.min=30;
    timet.sec=00;
//设置RTC设备的时间为2014年9月11日周四12时30分0秒
    settime(&timet);
    int i=0;
    for(;;)
    {
        i++;
        printf("task2 run %x task2ID:%x \n\r",i,pid(NULL));
//获取RTC设备的时间
        gettime(&timet);
        printf("year:%d mon:%d date:%d hour:%d min:%d sec:%d\n\r",
            timet.year,timet.mon,timet.date,timet.hour,timet.min,timet.sec);
    }
    return;
}

void task3_main()
{
    int i=0;
    for(;;)
    {
        i++;
        printf("task3 run %x task3ID:%x \n\r",i,pid(NULL));
    }
    return;
}

.....

void task5_main()
{
    int i=0;
    for(;;)
    {
        i++;
        printf("task5 run %x task5ID:%x \n\r",i,pid(NULL));
    }
    return;
}

```

记住别忘了增加 LMOSEM 内核的进程初始化函数中的新建进程的代码，因为又增加了 4 个应用程序，编译并链接好 LMOSEM 内核，打开开发板的电源，把 LMOSEM 内核文件下载到开发板中运行。正常情况下应该会出现如图 12-2 ~ 图 12-5 所示的情况。

```

深度终端
cpuidle is run!!
task5 run 1 task5ID:304112d9
task5 run 2 task5ID:304112d0
task5 run 3 task5ID:304112d0
task5 run 4 task5ID:304112d0
task5 run 5 task5ID:304112d0
task5 run 6 task5ID:304112d0
task5 run 7 task5ID:304112d0
task4 run 1 task4ID:3041124c
task4 run 2 task4ID:3041124c
task4 run 3 task4ID:3041124c
task4 run 4 task4ID:3041124c
task4 run 5 task4ID:3041124c
task4 run 6 task4ID:3041124c
task4 run 7 task4ID:3041124c
task4 run 8 task4ID:3041124c
task4 run 9 task4ID:3041124c
task4 run a task4ID:3041124c
task3 run 1 task3ID:304111c8
task3 run 2 task3ID:304111c8
task3 run 3 task3ID:304111c8
task3 run 4 task3ID:304111c8

```

图 12-2 多个进程运行 (一)

```

深度终端
task3 run 1 task3ID:304111c8
task3 run 2 task3ID:304111c8
task3 run 3 task3ID:304111c8
task3 run 4 task3ID:304111c8
task3 run 5 task3ID:304111c8
task3 run 6 task3ID:304111c8
task3 run 7 task3ID:304111c8
task3 run 8 task3ID:304111c8
task3 run 9 task3ID:304111c8
task3 run a task3ID:304111c8
task2 run 1 task2ID:30411144
year:2014 mon:9 date:11 hour:12 min:30 sec:0
task2 run 2 task2ID:30411144
year:2014 mon:9 date:11 hour:12 min:30 sec:0
task2 run 3 task2ID:30411144
year:2014 mon:9 date:11 hour:12 min:30 sec:0
task2 run 4 task2ID:30411144
year:2014 mon:9 date:11 hour:12 min:30 sec:0
task1 run 1 task1ID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 2 task1ID:304110c0
task0 run 1 task0ID:3041103c

```

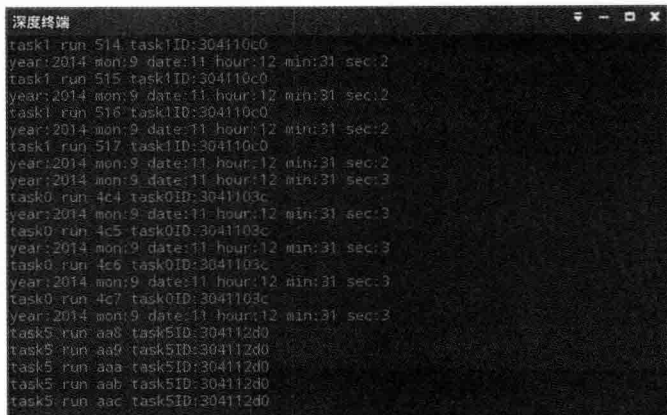
图 12-3 多个进程运行 (二)

```

深度终端
task1 run 1 task1ID:304110c0
year:0 mon:0 date:0 hour:0 min:0 sec:0
task1 run 2 task1ID:304110c0
task0 run 1 task0ID:3041103c
year:2014 mon:9 date:11 hour:12 min:30 sec:1
task0 run 2 task0ID:3041103c
year:2014 mon:9 date:11 hour:12 min:30 sec:1
task0 run 3 task0ID:3041103c
year:2014 mon:9 date:11 hour:12 min:30 sec:1
task0 run 4 task0ID:3041103c
year:2014 mon:9 date:11 hour:12 min:30 sec:1
task0 run 5 task0ID:3041103c
year:2014 mon:9 date:11 hour:12 min:30 sec:1
task5 run 6 task5ID:304112d0
task5 run 5 task5ID:304112d0
task5 run a task5ID:304112d0
task5 run b task5ID:304112d0
task5 run c task5ID:304112d0
task5 run d task5ID:304112d0
task5 run e task5ID:304112d0
task5 run f task5ID:304112d0
task5 run 10 task5ID:304112d0

```

图 12-4 多个进程运行 (三)



```
task1 run 514 taskID:304110c0
year:2014 mon:9 date:11 hour:12 min:31 sec:2
task1 run 515 taskID:304110c0
year:2014 mon:9 date:11 hour:12 min:31 sec:2
task1 run 516 taskID:304110c0
year:2014 mon:9 date:11 hour:12 min:31 sec:2
task1 run 517 taskID:304110c0
year:2014 mon:9 date:11 hour:12 min:31 sec:2
year:2014 mon:9 date:11 hour:12 min:31 sec:3
task0 run 4c4 taskID:3041103c
year:2014 mon:9 date:11 hour:12 min:31 sec:3
task0 run 4c5 taskID:3041103c
year:2014 mon:9 date:11 hour:12 min:31 sec:3
task0 run 4c6 taskID:3041103c
year:2014 mon:9 date:11 hour:12 min:31 sec:3
task0 run 4c7 taskID:3041103c
year:2014 mon:9 date:11 hour:12 min:31 sec:3
task5 run aa8 taskID:304112d0
task5 run aa9 taskID:304112d0
task5 run aaa taskID:304112d0
task5 run aab taskID:304112d0
task5 run aac taskID:304112d0
```

图 12-5 多个进程运行（四）

图 12-2 ~ 图 12-5 展示了 6 个进程交替运行的情况。这从输出的进程 ID 和运行循环的次数可以得到证明。其实，LMOSEM 内核中的进程 ID，就是进程描述符数据结构的实例变量在内存中的地址。由于这个地址具有唯一性，所以进程 ID 也具有唯一性。还可以看到应用程序开始获取的时间信息是错误的，但是过了一会儿有一个应用程序设置了相应的时间，RTC 设备就从设置的时间开始计时了，又过了几秒钟，RTC 设备的 `rtc_tick` 中断产生了，更新了 LMOSEM 内核的时间数据，有些获取系统时间的应用程序输出的时间也正常了。另外，我们的应用程序都是一个死循环，那是因为我们的应用程序不能退出，同时 LMOSEM 内核也没有实现一个应用程序进程退出的机制。

一开始，用两个应用程序进程测试了文件操作的库函数，这实际测试了大多数库函数和系统调用。接着又用 6 个应用程序进程，测试了在多任务环境下的情况，并且还设置了新的时间。看着交替出现的进程 ID 和连续的应用程序中的循环次数，就知道我们辛苦打造的操作系统内核虽然功能不多，但所有的功能机制的实现都是正确无误的。下面还是要去回顾本章到底做了些什么。

12.8 小结

本章主要实现了系统调用与应用程序库。有了它们，应用程序开发者就可以利用它们开发 LMOSEM 系统下的应用程序了。

下面来归纳系统调用与应用程序库到底涉及哪些内容，如下：

- 1) 从系统调用机制开始，了解了什么是软中断指令，以及如何用它来实现应用程序调用 LMOSEM 内核的函数，然后知道了如何在应用程序与 LMOSEM 内核之间传递系统调用的参数，最后实现了系统调用分发器，完成了多个系统调用的分发。

- 2) 关于系统调用，首先实现了时间管理的系统调用，用于应用程序获取、设置时间。

接着实现了进程方面的系统调用，例如，进程的运行和退出、获取进程 ID 等。然后实现了分配、释放内存空间的系统调用，让应用程序可以动态使用内存空间。最后实现了设备和文件的系统调用，主要用于设备和文件的打开、关闭、读写、控制方面的操作。

3) 为了方便应用程序开发者，我们实现了一些应用程序库供应用程序开发者使用。其中有一些库函数是直接对系统调用的封装，而有的库函数则调用了多个系统调用来完成特定的功能，并且它们都是静态库。

4) 最后我们写了多个应用程序，对整个 LMOSEM 内核的功能机制进行了一系列的测试，以保证其代码的正确性。

从第一行汇编代码开始，到 LMOSEM 内核的硬件相关层，到 LMOSEM 内核的功能组件层，再到这里的 LMOSEM 内核的接口层，一路走来，虽然经历了许多困难，不过也算是实现了我们最开始对 LMOSEM 内核的设计，现在终于可以闲下来自我欣赏我们自己建造的操作系统内核了。

后 记

书到此处，你亦能阅至此处，多半是出于兴趣，出于一种对操作系统的热爱，出于一种对事物本质发自内心的苛求。如果是这样，请永远保持这份心性，它会给你带来更多意想不到的结果。不过走到这里也该休息休息了。这里先停住前进的脚步，回忆一下这一路走来都做了些什么。

从编程初学者都会写的“hello, world!!”应用程序开始，我们知道了应用程序通常的运行过程，进而一步步了解了操作系统内核中的所有组件，在心中建立了一个现代操作系统内核的模型。但是并没有急于实现，因为我们知道操作系统内核必须要运行在具体的计算平台上。于是选择了一款具体的计算平台，了解了该计算平台上的各个组件，并且着重了解了其上的 CPU 和内存的细节，因为这两个器件是程序运行的基础。接着亲自设计了我们的操作系统内核，搭建了开发环境，了解了编译工具集。有了这些基础，我们从第一行汇编代码开始，逐步实现了内存管理、中断管理、驱动模型、文件系统、若干设备驱动程序、最后实现了系统调用和应用程序库。虽然为了简化实现，这些组件比成熟的操作系统内核中的组件简单得多，但是我们实现的都是最关键、最核心、最必要的功能机制。简洁而全面一直是我们的思想，这也正是作为一个操作系统初学者想要的。

这时，你或许在思索，我们亲自建造的操作系统，为什么没有 Linux 系统那么强大的文件系统和虚拟内存；为什么没有 Linux 系统高效的网络性能；为什么没有 Windows 系统那么精美而高性能的图形界面；为什么没有 UNIX 系统工业级的安全性？

如果你真的在思索、在好奇，如果你真的有这样一种兴趣，还想继续探索，笔者真诚希望你能再次阅读更多的书籍，或者打开那万能的互联网浏览器，去探索，去寻找。相信以这份好奇和兴趣为动力，必定会从一无所知，到知道一点点，再到知道一部分，慢慢积累，也许有一天你会惊奇地跳起来，用尽全身力气喊出来：“原来我也能了”，“我真的能了”！到了那一天，想必我们也已经有了全新的开始，那一定是真正具备创造性的开始。

也许有一天人们正用我们建造的操作系统，在计算机上听着杜比级别的音乐、看着 4K 画质的高清电影、玩着如梦如幻的 3D 游戏、用 4G 移动网络在和远方的恋人进行视频通话、进行超大规模的科学计算、处理着海量级的网络数据。但是别忘了，这仅仅是因为我们那一点点求知欲和兴趣。

相信很多同行和笔者以前一样，阅读过很多操作系统理论方面的书籍，感觉它们所讲述的理论大同小异且比较容易理解，但是真要我们自己去动手开发和实现操作系统时，我们可能连操作系统的第一行代码该怎么写都不知道。为什么会这样？

因为操作系统是硬件平台上的第一层软件，若不了解硬件平台的细节，是不能实现操作系统的，所以本书一开始就选择了一款常用的硬件平台，并对其进行了详细描述，重点介绍了硬件平台上的实时时钟、定时器、串口、中断控制器、内存芯片、flash芯片、CPU、MMU等，然后介绍操作系统架构设计和开发环境与开发工具，尤其开发工具，详细了解开发工具会大大方便后面的开发工作并解决一些困惑，最后介绍了操作系统的初始化、HAL、内存管理、中断管理、设备管理、进程管理、文件系统、系统调用接口、应用程序库。对这些组件的介绍都是先明确这个组件要完成什么功能，达到什么要求。接着详述这个组件必须要关注的硬件的细节，然后把我们的想法和设计归纳成数据结构。最后编写完成这些功能的代码。

这些内容将奠定本书在操作系统实践领域中里程碑式的地位。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/操作系统

ISBN 978-7-111-51487-9



9 787111 514879 >

定价: 99.00元

[General Information]

书名=Linux/Unix技术丛书 深度探索嵌入式操作系统 从零开始设计、架构和开发

作者=彭东著

页数=528

SS号=13869534

DX号=

出版日期=2015.09

出版社=北京机械工业出版社

封面

书名

版权

前言

目录

第1章 操作系统的功能及为什么需要它

1.1 从hello world开始

1.2 操作系统功能演进

1.3 小结

第2章 硬件平台

2.1 选择平台

2.1.1 mini2440

2.1.2 mini2440平台的信息

2.2 必须要关注的硬件

2.2.1 原因

2.2.2 RTC

2.2.3 定时器

2.2.4 串口

2.2.5 中断控制器

2.2.6 SDRAM

2.2.7 Norflash

2.2.8 Nandflash

2.3 小结

第3章 处理器

3.1 ARM公司与其处理器

3.2 ARM920T的结构与特性

3.2.1 ARM920T CPU结构

3.2.2 ARM920T CPU特性

3.3 ARM920T存储体系

3.3.1 ARM920T地址空间

3.3.2 ARM920T存储器格式

3.3.3 ARM920T存储地址对齐

3.4 ARM920T状态

3.4.1 ARM状态

3.4.2 Thumb状态

3.5 ARM920T处理器的7种工作模式

- 3.6 寄存器
 - 3.7 异常和中断
 - 3.7.1 什么是异常和中断
 - 3.7.2 异常中断向量
 - 3.8 ARM920T指令集
 - 3.8.1 指令及其编码格式
 - 3.8.2 分支跳转指令
 - 3.8.3 数据处理指令
 - 3.8.4 装载和存储指令
 - 3.8.5 程序状态寄存器操作指令
 - 3.8.6 协处理器操作指令
 - 3.8.7 异常中断产生指令
 - 3.9 MMU
 - 3.9.1 MMU概述
 - 3.9.2 为什么要有MMU
 - 3.9.3 ARM920T CP15协处理器
 - 3.9.4 MMU页表
 - 3.9.5 MMU页面访问权限的控制
 - 3.9.6 MMU的快表TLB
 - 3.9.7 MMU的编程接口
 - 3.10 Cache
 - 3.10.1 ARM920T的Cache
 - 3.10.2 Cache的原理
 - 3.10.3 Cache的类型及要注意的问题
 - 3.10.4 ARM920TCache的编程接口
 - 3.11 小结
- 第4章 操作系统内核的设计与构建
- 4.1 操作系统内核的设计
 - 4.1.1 内核要完成的功能
 - 4.1.2 内核的架构
 - 4.1.3 分离硬件的相关性
 - 4.1.4 我们的选择
 - 4.2 开发环境及相关工具
 - 4.2.1 Linux环境
 - 4.2.2 文本编辑器
 - 4.2.3 GCC

- 4.2.4 LD
 - 4.2.5 make
 - 4.3 LMOSEM的构建系统
 - 4.3.1 LMOSEM的makefile
 - 4.3.2 LMOSEM的链接脚本
 - 4.4 开发板的安装
 - 4.5 小结
- 第5章 语言间调用约定与基本数据结构
 - 5.1 寄存器使用约定
 - 5.1.1 寄存器别名
 - 5.1.2 参数传递与返回值
 - 5.2 基本数据结构
 - 5.2.1 C语言的基本数据结构
 - 5.2.2 list_h_t数据结构
 - 5.2.3 spinlock_t数据结构
 - 5.2.4 kwlst_t数据结构
 - 5.2.5 sem_t数据结构
 - 5.3 数据结构存在于内存中的形式
 - 5.4 C与汇编的混用
 - 5.5 小结
- 第6章 内核初始化
 - 6.1 开始
 - 6.1.1 第一行汇编代码
 - 6.1.2 第一个C函数
 - 6.2 MMU和中断向量的初始化
 - 6.2.1 初始化MMU
 - 6.2.2 复制中断向量
 - 6.3 串口初始化
 - 6.3.1 串口硬件
 - 6.3.2 内核的printf
 - 6.4 机器数据结构
 - 6.4.1 设计数据结构
 - 6.4.2 确定一些重要数据结构与内核的地址
 - 6.5 初级内存管理初始化
 - 6.5.1 设计一些数据结构
 - 6.5.2 初始化mmappedsc_t结构数组

- 6.5.3 建立起内存分配数据结构
- 6.6 中断初始化
 - 6.6.1 设计一些数据结构
 - 6.6.2 初始中断源描述符
- 6.7 初始化测试
- 6.8 小结
- 第7章 内存管理
 - 7.1 内核功能层入口
 - 7.2 内存管理组件的设计
 - 7.3 块级内存管理
 - 7.3.1 块级内存管理数据结构视图
 - 7.3.2 块级内存管理接口
 - 7.3.3 主分配函数
 - 7.3.4 分配时查找alcfrelst_t
 - 7.3.5 分配时查找和操作mmapdsc_t
 - 7.3.6 分配代码写得对吗
 - 7.3.7 主释放函数
 - 7.3.8 释放时查找alcfrelst_t
 - 7.3.9 释放时查找和操作mmapdsc_t
 - 7.3.10 测试块级内存管理层
 - 7.4 页级内存管理
 - 7.4.1 页级内存管理接口及调用流程
 - 7.4.2 相关的数据结构
 - 7.4.3 页级内存管理初始化
 - 7.4.4 分配主函数
 - 7.4.5 分配时查找mplhead_t
 - 7.4.6 分配时新建页级内存池
 - 7.4.7 分配时操作mplhead_t
 - 7.4.8 分配代码写得对吗
 - 7.4.9 释放主函数
 - 7.4.10 释放时查找mplhead_t
 - 7.4.11 释放时操作mplhead_t
 - 7.4.12 释放时删除页级内存池
 - 7.4.13 测试页级内存管理层
 - 7.5 字级内存管理
 - 7.5.1 字级内存接口及调用流程

- 7.5.2 相关的数据结构
- 7.5.3 分配主函数
- 7.5.4 分配时查找mplhead_t
- 7.5.5 分配时新建字级内存池
- 7.5.6 分配时操作mplhead_t
- 7.5.7 分配代码写得对吗
- 7.5.8 释放主函数
- 7.5.9 释放时查找mplhead_t
- 7.5.10 释放时操作mplhead_t
- 7.5.11 释放时删除字级内存池
- 7.5.12 测试字级内存管理层

7.6 小结

第8章 中断管理

8.1 中断与中断控制器

- 8.1.1 什么是中断
- 8.1.2 S3C2440A中断控制器

8.2 中断管理的架构与相关数据结构

- 8.2.1 中断管理的架构
- 8.2.2 设计数据结构intfltdsc_t和intserdsc_t

8.3 中断处理

- 8.3.1 中断辅助例程
- 8.3.2 从中断向量开始
- 8.3.3 保存CPU上下文
- 8.3.4 中断主分派例程
- 8.3.5 确定中断源
- 8.3.6 调用中断处理例程

8.4 安装中断回调例程

8.5 小结

第9章 驱动模型

9.1 操作系统内核如何管理设备

- 9.1.1 分权而治
- 9.1.2 设备类型
- 9.1.3 驱动程序

9.2 相关数据结构

- 9.2.1 驱动
- 9.2.2 派发例程类型

- 9.2.3 设备ID
 - 9.2.4 设备
 - 9.2.5 I/O包
 - 9.2.6 设备表
 - 9.3 驱动模型的基础设施
 - 9.3.1 驱动程序从哪里执行
 - 9.3.2 新建与注册设备
 - 9.3.3 注册回调函数
 - 9.3.4 发送I/O包
 - 9.3.5 调用驱动程序函数
 - 9.3.6 等待服务
 - 9.3.7 完成服务
 - 9.3.8 驱动模型辅助函数
 - 9.4 systick驱动程序实例
 - 9.4.1 systick硬件
 - 9.4.2 systick驱动程序框架
 - 9.4.3 systick驱动程序实现
 - 9.4.4 测试systick驱动程序
 - 9.5 RTC驱动程序实例
 - 9.5.1 RTC硬件
 - 9.5.2 RTC驱动程序实现
 - 9.6 小结
- 第10章 进程
- 10.1 应用程序的运行
 - 10.1.1 程序运行需要什么资源
 - 10.1.2 任何时刻资源都可用吗
 - 10.1.3 提出多道程序模型
 - 10.2 相关的数据结构
 - 10.2.1 设计进程的数据结构
 - 10.2.2 调度进程表
 - 10.3 LMOSEM内核的第一个进程
 - 10.3.1 进程管理组件的初始化
 - 10.3.2 建立空转进程
 - 10.3.3 空转进程运行
 - 10.4 新建进程
 - 10.4.1 分配进程描述符

- 10.4.2 分配内存空间
 - 10.4.3 加入进程调度表
 - 10.5 进程调度
 - 10.5.1 调度算法
 - 10.5.2 处理进程时间片
 - 10.5.3 检查调度状态
 - 10.5.4 选择进程
 - 10.5.5 进程切换
 - 10.5.6 进程等待与唤醒
 - 10.5.7 进程测试
 - 10.6 小结
- 第11章 文件系统
- 11.1 文件系统设计
 - 11.1.1 文件系统只是一个设备
 - 11.1.2 数据格式与存储块
 - 11.1.3 如何组织文件
 - 11.1.4 关于我们文件系统的限制
 - 11.2 相关的数据结构
 - 11.2.1 超级块
 - 11.2.2 位图
 - 11.2.3 目录
 - 11.2.4 文件管理头
 - 11.3 文件系统格式化
 - 11.3.1 建立超级块
 - 11.3.2 建立位图
 - 11.3.3 建立根目录
 - 11.4 文件系统基础操作
 - 11.4.1 获取与释放根目录文件
 - 11.4.2 字符串操作
 - 11.4.3 分解路径名
 - 11.4.4 检查文件是否存在
 - 11.5 文件操作
 - 11.5.1 新建文件
 - 11.5.2 删除文件
 - 11.5.3 打开文件
 - 11.5.4 读写文件

- 11.5.5 关闭文件
 - 11.5.6 驱动整合
- 11.6 文件系统测试
 - 11.6.1 格式化测试
 - 11.6.2 文件操作测试
- 11.7 小结
- 第12章 系统调用与应用程序库
 - 12.1 系统调用机制
 - 12.1.1 软中断指令
 - 12.1.2 传递系统调用参数
 - 12.1.3 系统调用分发器
 - 12.2 时间管理系统调用
 - 12.3 进程管理系统调用
 - 12.3.1 进程的运行与退出
 - 12.3.2 获取进程的ID
 - 12.4 内存管理系统调用
 - 12.5 设备与文件系统调用
 - 12.5.1 设备与文件的打开
 - 12.5.2 设备与文件的关闭
 - 12.5.3 设备与文件的读写
 - 12.5.4 设备与文件的控制
 - 12.6 应用程序库
 - 12.7 测试
 - 12.8 小结

后记

封底