嵌入式 Linux 操作系统裁剪和定制研究

摘要

嵌入式 Linux 的研究之所以成为当今操作系统研究的热点，是因为它的应用蕴含着巨大的商业价值。嵌入式系统之间差别很大，掌上电脑（PDA）、机顶盒、手机、数码相机、数字电视、家用电器、工业控制设备，等等，都是典型的嵌入式应用。和桌面操作系统相比，由于嵌入式应用环境之间的差别很大，难于开发出适应于各种嵌入式应用环境的通用嵌入式操作系统。当前流行的各种嵌入式操作系统，仅仅在某些特定领域获得成功，其原因就在于此。嵌入式 Linux 操作系统也存在这方面的问题。因此，研究嵌入式操作系统的裁剪和定制技术就显得非常必要。

作者从致力于开发自己的嵌入式操作系统和商业应用的目的出发，进行了嵌入式 Linux 内核的裁剪与定制研究，力求创造出具有我国自主产权的操作系统。而任何的裁减都是基于对内核的准确理解之上的，作者首先系统介绍了一个完整的 Linux 内核的总体结构并就其主要子系统，如初始化、中断处理子系统、进程调度进行了详细分析；然后从嵌入式 Linux 现代设计特点出发，提出了一种基于调用图的裁剪 Linux 方法并对各部分的具体裁减从实例出发进行了说明。同时又从定制角度，讨论了内存管理子系统、文件子系统的优化和选择方法。最后，文章针对 DSP 应用的特点，分析了如何将 Linux 内核加以裁剪、改造，使其适合 DSP 应用，形成通用的 DSP 操作系统。移植后的 Linux 内核，大小可以控制在 200K 以下，减小了 DSP 平台上应用软件的开发难度，大大降低了 DSP 系统的开发成本。

设备驱动程序运行在核心态，是 Linux 内核重要组成部分。它出现的问题会直接影响嵌入式 Linux 的稳定，严重时会导致操作系统崩溃。文章利用虚拟字符设备来最大限度地封装硬件设备驱动的具体细节和其特定信息模式，以降低嵌入式 Linux 应用系统程序开发调试的难度，增强系统的可配置性。并总结了两种虚拟字符设备用以实际的嵌入式开发。

文章最后对嵌入式 Linux 系统开发尚需解决的问题提出了自己的看法。

关键字：嵌入式 Linux，操作系统,内核，裁剪，定制

Research on Tailoring and Customizing Embedded Linux Operating System

Abstract

LI Gemei LIU Fuyan

Embedded Linux possesses great application value and has been the hotspot of OS research, but the distinctions on the embedded systems is colossal, which include PDA, top box, mobile phone, digital camera, digital TV, industrial control device and other typical applications. In contrast with the versatile OS, as different environments, it is very difficult to develop the embedded operating system which adapt all applications. That results in the consequence, the current popular system only succeed in some special domain. The embedded Linux also embraces this problem, thus the research on tailoring and customizing operating system is apparently very important.

To develop Embedded OS for commercial purpose, the author applies to the research of cutting and customizing Linux kernel, thus can create our own IP (Intellectual Property) embedded system. At first, this paper presents the concrete structure of Linux kernel, to understand essence of kernel accurately for tailoring, and analyze the brief parts in detail for initialization, the interrupt handling subsystem, process schedule. In light of the characteristics of the design for the embedded Linux system, this paper propose an approach, which is techniques of cutting for embedded Linux based on call graph and the practical cuttings for individual parts are explained, and while from another angle, discuss the choosing and optimization for memory management subsystem and file subsystem. At last, the author propounds how to reduce Linux kernel and build a general DSP RTOS for DSP applications. Using this ported Linux kernel on DSP platform can make your work more easily and reduce the cost for DSP development.

Device driver is one of the most important components of Linux kernel. It runs in kernel status, and any problem will affect the stability or even lead to crash. This paper use the virtual character device to envelop, in the ultimate degree, the concrete hardware driving details and their individual information patterns, thus reduce the difficulties of debug and development on the embedded Linux system and enhance the system configurable. Also summarize two modes of virtual characters for the factual applications.

Eventually, this paper brings forward viewpoint for the remaining problem on the development for embedded Linux.

key words: embedded Linux, operating system, kernel, tailor, customize,

本人声明

我声明:本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

姓名：李革梅

签字：李革梅

日期：2005 年 3 月 1 日

1 绪论

1.1 研究的目的与意义

随着芯片集成度的提高，限制嵌入式系统发展的瓶颈突出表现在软件方面。从八十年代起，国际上就有一些 IT 组织、公司，开始进行商用嵌入式系统和专用操作系统的研究与开发，这其中著名的嵌入式操作系统有：Windows CE、VxWorks、pSOS、QNX、Palm OS、OS-9、LynxOS 等[1,2]。由于 Linux 所具有的内核小、效率高、源代码开放和强大的网络支持等的一系列独特优势，业界已经逐渐达成共识：嵌入式 Linux 将在未来的嵌入式领域中占最大的份额，其巨大的市场潜力已经吸引了众多的厂商进入这一领域。目前，国外的著名厂商，例如 IBM、DELL 等，几乎都在嵌入式 Linux 操作系统的研究、开发和推广方面投入了巨资。国内有一些公司，例如华恒、共创、中软、红旗、万禾等，也开始积极从事嵌入式 Linux 操作系统的研究、开发和推广[3]。

嵌入式操作系统是内装于专用系统或设备中，具有通用操作系统的基本功能，执行计算和数据处理功能的操作系统。与通用操作系统相比，嵌入式操作系统具有配置专一、结构紧凑、坚固可靠等特点。近年来，嵌入式操作系统得到了飞速的发展，从仅支持单一品种的微处理器到支持多品种的微处理器；从仅包含任务调度、内存管理、进程间通讯等基本功能到不仅提供功能完备、可以与通用操作系统相比的功能外，还提供其他如文件系统、TCP/IP 网络协议、窗口系统等功能。随着半导体工业的飞速发展，加上嵌入式应用的多样化需求，要求嵌入式系统具有合理的体系结构和良好的可用性、可伸缩性、易于移植等特点。嵌入式系统的多样化特征，使得对操作系统进行面向应用的定制比通用系统更为必要。因此，研究嵌入式操作系统裁剪定制技术，增加操作系统的通用性就显得非常必要。

新世纪之初，信息化浪潮席卷全球。不论是发达国家、新兴工业化国家，还是发展中国家，在全球进入以网络计算为核心的信息时代都不可避免地被卷入到这股信息化的世纪风暴之中，嵌入技术是电子信息技术应用最广泛的应用领域之一，嵌入技术及其产品广泛应用于智能家用电器、智能建筑、仪器仪表、通讯产品、工业控制、数控机床、

掌上型电脑、各种智能 IC 卡的应用，如家庭电表、水表、煤气表的 IC 卡计费系统、电子钱包、第二代身份证、公共交通收费系统、电子病历及挂号系统、以及军事领域的应用等，大量的嵌入技术的应用不胜枚举，大多数老百姓感受到信息技术的进步及电子信息技术产品的小巧、便捷和高效，都直接或间接源于嵌入技术，嵌入技术的研发和广泛应用已经成为我国信息化进程的重要课题之一，受到社会各界的广泛关注。

嵌入式技术和设备进入到我国以后，呈星火燎原之势迅速蔓延开来，其应用已涉及到生产、工作、生活各个方面。从家电中的冰箱、洗衣机、电视、微波炉到 MP3、DVD；从轿车中控到火车、飞机的安全防范；从手机电话到 PDA；从医院的 B 超、CT 到核磁共振器；从机械加工中心到生产线上的机器人、机械手；从航天飞机、载人飞船，到水下核潜艇，到处都有嵌入式计算机、嵌入式控制器、嵌入式实时多任务操作系统，嵌入式应用软件。可以说嵌入式技术无所不在，嵌入式技术和设备的应用在我国国民经济和国防建设的各个方面存在着广泛的应用领域，有着巨大的市场。可以说它是信息技术的一个新的发展，是信息产业的一个新的亮点[4]。计算机迈入了另一个充满机遇的阶段---后 PC 时代。形式多样的数字化产品已经成为信息处理的一大主要工具，并且正在逐步形成一个充满商机的巨大产业。如果说 PC 机的发展带动了整个桌面软件的发展，那么数字化产品的广泛普及将为嵌入式软件的蓬勃发展提供无穷的推动力。

863 智能计算机首席专家高文教授说:所谓后 PC 时代,是英文 pervasive computing 的中文意译，pervasive 的原意是普遍的、蔓延的、渗透的，所以 pervasive computing 这个词组直接的翻译应该是渗透到各个方面的计算。因而我们可以认为，所谓后 PC 时代是指：计算机无所不在，它渗透到我们工作和生活的方方面面。当然，这样的无所不在的计算机也绝不都是象今天的 PC 一样摆在桌子上或放在书包里，后 PC 时代的绝大多数计算机是以非计算机的形式出现的，例如作为随身物品出现的电话、遥控开关、电子戒指、电子手杖等，再例如作为家庭网络组成部分的电视机、电冰箱、空调等等。这些设备的核心部分都有计算机，但大多是以嵌入式系统的形式存在，而不是以整机的形象出现。所以，我们也可以说后 PC 时代的特点是计算机无处不在。

嵌入于信息家电（或其他设备）中的嵌入式 Linux,是国际软件界的一个新宠，女娲 Hopen OS 的开发成功给我们以提示，Linux 可以提供完成嵌入功能的的基本内核和你所需要的所有用户界面，它本身就是多面的。它能处理嵌入式任务和用户界面。将 Linux

看成是连续的统一体，从一个具有内存管理、任务切换和实践服务及其他分拆的微内核到完整的服务器，完美实现支持所有的文件服务和网络服务。同时 Linux 作为一个自由软件，也得到了极大的发展，Linux 自身具备一整套工具链，容易自行建立嵌入式系统的开发环境和交叉运行环境，并且可以跨越嵌入式系统开发中的仿真工具（ICE）的障碍[5]。更值得一提的是，Linux 内核的模块化特点将接口和其实现分离开来，理论上能够保证一个模块可以在不影响其他模块的情况下进行改变，也可以将模块之间的依赖关系仅仅限定于接口。所以国际上一些科研机构提出可以根据自己的具体需要选择和裁减模块，利用高度模块化的构件方法实现可定制的嵌入式操作系统。同时嵌入式操作系统也常常要求通用的功能，为了避免重复劳动，这些功能的实现运用了许多现成的程序和驱动程序，它们可以用于公共外设和应用。 然而，因为还在起步阶段，目前的嵌入式 Linux 版本还不是一套非常简洁的系统。如何优化系统至适合嵌入式应用成为最主要的目标。

可见对嵌入式 Linux 的软件开发主要集中在嵌入式 Linux 内核。对于嵌入式 Linux 的主要挑战是把系统资源的需求减少，以适应于诸如内存、固态电子盘容量、处理器速度、以及节能的限制[6]。嵌入式操作系统需要从一个芯片级盘片或者闪存式电子盘启动，或者启动并运行于没有显示及键盘的环境中，或从一个远程的设备上通过以太网连接来加载应用程序。现在已经有一些可参考的小型 Linux 的来源，其中又发展出大量的 Linux 配置及发行版本用来满足特别的需求，诸如路由器、防火墙、Internet 以及网络应用、网络服务器、网关等等。我们可以有选择地生成我们需要的嵌入式 Linux，从一个标准发行版中开始裁减不需要的模块。甚至，我们可以从别人配置过的版本开始开发，因为他人的嵌入式版本也是开放源代码的，最突出的优点是在他人的工作的基础上建立自己的系统，这在 Linux 开发群体中不仅是合法的，而且是受到鼓励的和支持的。我们可以根据应用系统设计的具体需要，从硬件环境的现有配置包括内部体系结构、内存限制、输入输出接口来认真选择相应的数据结构以及系统调用函数，并可以在完全了解内核工作原理的基础上更改、替换内核源代码中已经定义过的数据结构的数据域和内核线程函数，最后对 Linux 内核源代码进行重新编译链接，生成各种不同的嵌入式操作系统。系统典型的实现步骤为：

(1) 重新编译 Linux 内核(kernel)，去掉内核中不需要的模块，诸如 PCMCIA 之类的外

设支持模块等。

(2) 编写 Boot Loader，制作 Boot ROM 用于加载嵌入式 Linux 内核到内存

(3) 重新设计以太网驱动程序以及串 / 并口驱动程序。

(4) 设计嵌入式 Linux 应用程序，管理打印服务的应用。

(5) 嵌入式 Linux 系统执行流程如图 1.1 所示。

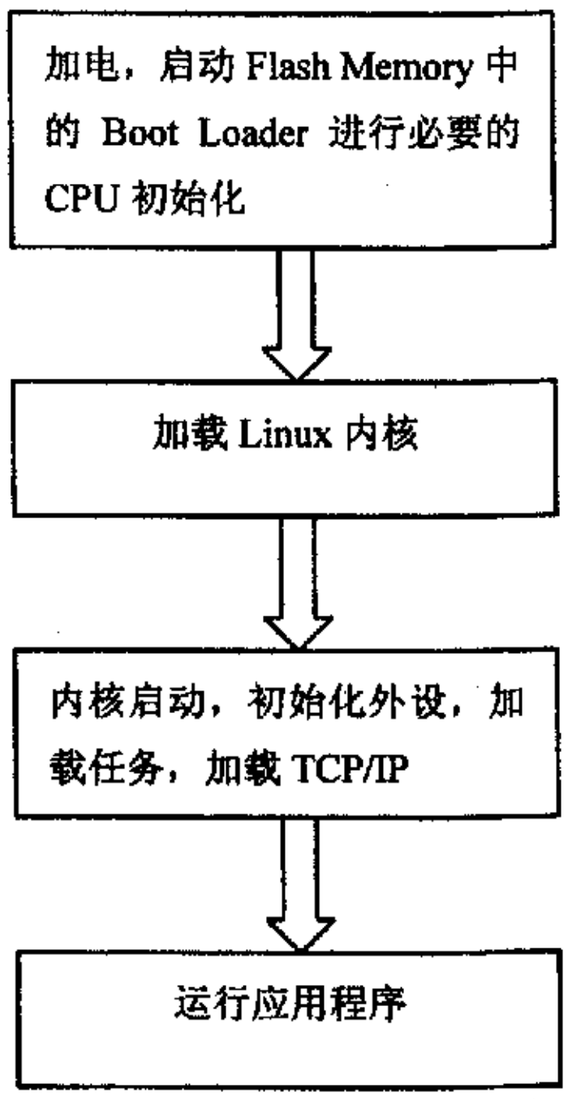

图 1.1　系统执行流程图

由于各种嵌入式系统产品中包含各不相同的特殊需求，导致这类系统中使用的专用 Linux 嵌入式操作系统不下几百种，而且至今仍有一半的开发者使用自行开发的嵌入式操作系统；因为开发成功的嵌入式产品时都要针对具体的工作做出巨大的调整，这是解决嵌入式 Linux 软件开发的一个技术难点。我们知道 Linux 非常地灵活，适用于不同的处理器和不同的硬件平台；而嵌入式系统总的来说却不灵活；而且它们完全是为最有效实现预定功能而严格设计的。这样软件开发的大部分时间将集中在对于不同应用的具体内核裁剪和配置。我们希望找到这个统一的最小内核，然后再根据具体的内部结构定制出自己的内核，从而大大缩短软件开发的周期。

本论文研究的目的，就是解决如何使嵌入式 Linux 操作系统适应尽量多的应用领域的问题。目前这方面的研究还很少，是一个需要研究还没有充分研究的问题。对于嵌入

式系统软件的开发与应用，具有重要的研究意义和应用价值。

进入 21 世纪，数字化产品欣欣向荣；由于硬件平台种类各异，因此嵌入式软件市场呈现出一种高度细分的格局，国外产品进入后也很难垄断整个市场，尽管绝大部分的嵌入式系统的硬件平台仍掌握在外国大公司的手中，国产的嵌入式操作系统在技术含量、兼容性、市场运作模式等方面还有相当长的一段距离。但由于 Linux 自由操作系统的出现，特别是将嵌入式系统和 Linux 有机结合起来的嵌入式 Linux，给我们提供跟踪国外嵌入式操作系统最新应用技术难得的机遇，可以在机顶盒、掌上电脑或 PDA、手机和寻呼机上网、车载盒、工业控制等方面充分开发自己的嵌入式 Linux 产品，从而推动我国软件业的发展。

开发中国自主产权的嵌人式处理器和嵌人式操作系统，有十分重要的战略意义。谁掌握了 OS,谁也就掌握了嵌入式系统软件开发的主动权。由于 Linux 具有免费、源代码开放、支持多种 CPU 等优点，使用 Linux 作为底层操作系统，对它进行裁剪和定制，并在其基础上搭建嵌入式系统平台，成为日益流行的嵌入式操作系统的解决方案。

1.2 国内外发展状况

嵌入式系统，就是用于控制设备的计算机。它们最初于二十世纪六十年代晚期在通讯中被用于控制机电电话交换机。在过去的十多年里，计算机产业不断朝着更小的系统方向发展，嵌入式系统也与之一起为这些小型机器提供了更多的功能。渐渐地，就需要把这些嵌入式系统连接到某种网络上，这就提高了系统的复杂程度并要求更多的存储器和接口，因而也就产生了对操作系统服务的要求。二十世纪七十年代晚期出现了用作嵌入式系统的现成的操作系统。近年来，随着电子技术的不断进步，嵌入式系统开发已成为热点，它将先进的计算机技术、半导体技术和电子技术和一些著名的公司更选中了 Linux 操作系统作为开发嵌入式产品的工具。现在国外基于嵌入式 Linux 系统的产品已问世的有：韩国三星公司的 Linux PDA、可联网的 Linux 照相机，美国 Transmeta 公司的 Linux 手机、NetGem 的机顶盒、Qubit Technology 公司推出的基于 Linux 的书写板 Qubit(Tablet)、Screen Media 公司开发的基于 Linux 的手持设备 FreePad 等[7]。

各个行业的具体应用相结合的产物，这一特点就决定了它必然是一个技术密集、资

金密集、高度分散、不断创新的知识集成系统。嵌入式操作系统是支持嵌入式系统应用的操作系统软件，它是嵌入式系统极为重要的组成部分，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。与通用操作系统相比较，嵌入式操作系统在系统实时高效性、硬件的依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点。嵌入式操作系统的出现,将大大提高嵌入式系统开发的效率，改变以往嵌入式软件设计只能针对具体的应用从头做起的局面。在嵌入式操作系统之上开发嵌入系统将减少系统开发的工作量，增强嵌入式应用软件的可移植性，使嵌入式系统的开发方法更具科学性。

自 1989 年芬兰赫尔辛基大学学生 Linus Torvald 发布了一个新的 Unix 变种---Linux。到今天各种 Linux 系统已经成功应用于服务器，作为操作系统的一个重要分支又为嵌入式系统的技术发展带来了新的契机，嵌入式系统和 Linux 的有机结合，成为后 PC 时代计算机最普遍的应用形式,Linux 作为嵌入式系统的新选择,是非常有潜力的。

国内的嵌入式 Linux 厂商队伍正在逐渐壮大，开始形成一个百家争鸣的局面。市场上的嵌入式 Linux 厂商主要有中软、红旗、博利思、蓝点、网虎科技和共创软件联盟等等，它们各自均有自己的发展特点和技术特色。如：中软股份公司开发的中软嵌入式 Linux 操作系统，具有微秒级的强实时功能，已经在数控领域得到很好的应用，并在最近举办的 Linux World China 2001 的展示会上，受到国内外厂家和用户的广泛关注和好评。而且中软股份公司还具备了开发网络终端、信息家电、手持设备等嵌入式产品的技术储备和项目承接能力；中科红旗的嵌入式 Linux 在机顶盒、彩票机等也做了不少工作。但国内厂商们正在设计的嵌入式产品形态，实际上还都普遍处于概念产品的阶段，除了实时数控领域已经涌现大量明确需求以外，其它嵌入式领域仍需要一段市场的培育期，以及一个根据市场反馈不断修正产品形态的过程。

目前，国际上有数以百计的嵌入式 Linux 开发计划,许多大型跨国企业，已经瞄准了后 PC 时代的下一代计算设备——嵌入式计算设备，其中嵌入式 Linux 技术的关键：

（1）对 Linux 的裁减达到小型化的目的，并移植应用程序；

（2）对不同嵌入式微处理器的 Linux 内核代码移植主要依赖于不同的体系结构(ucLinux, xinhua_rong)，驱动程序的研究(老铁)；

（3）图形接口 GUI 以及微型浏览器的研究。Cool_bird & sky_yin

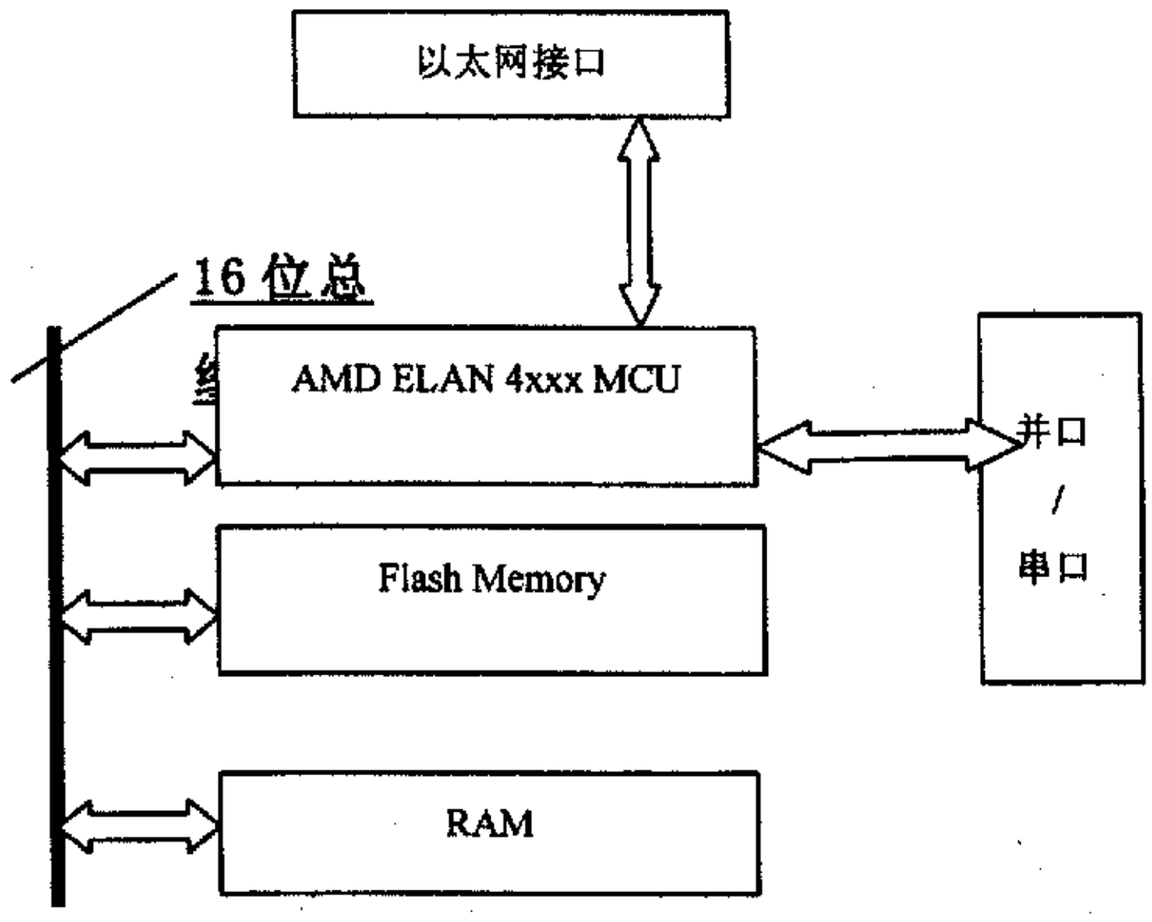

图 1.2　嵌入式 Linux 硬件平台框图

可见国内外对嵌入式 Linux 操作系统研究，主要集中下列几个方面：

（1）提高实时性

Linux 操作系统的实时性不是很好，而嵌入式环境往往对 Linux 操作系统的实时性要求很高。提高 Linux 的实时性是嵌入式 Linux 操作系统研究的一个研究热点[8,9]，也取得了很多有价值的研究成果。

（2）对特殊硬件的支持

嵌入式环境往往使用一些特殊的硬件，如何在嵌入式Linux操作系统支持特殊硬件，也是一个研究热点。例如，flash 存储的使用问题，把 flash 存储是作为存储器使用？作为文件系统使用？还是直接作为外设使用？哪种方法最合理，就是一个需要研究的问题。

（3）嵌入式 Linux 操作系统的裁剪和定制

Linux 操作系统内核中的实现算法，有些不适应嵌入式环境，有些在嵌入式环境中不需要。如何修改内核中某些部件的实现算法，一方面降低系统规模（嵌入式环境往往要求系统的规模较小），另一方面也使其更加适应嵌入式环境，也是一个研究热点。

当前在 Linux 操作系统的裁剪和定制技术方面的研究，主要集中于研究具体的裁剪

实现算法和裁剪实现技术，如何分离出 Linux 操作系统内核中必要部分和可裁剪部分，根据具体的应用，实现对各个可裁剪部分的定制，这方面的研究刚刚开始。本论文就是在这方面做了具体地研究和开发工作。

目前，世界上已经开始走向面向应用特定操作系统的研制工作（Application Specific Operating System,ASOS），从 ASOS 到各类的嵌入式系统，我认为这反映了电子和信息系统已经开始了“机械化”的进程。同时，ASOS 的成熟和嵌入式系统的广泛应用，必然加速系统程序设计的模块框图化和开发环境的集成智能化的趋势。Linux 会在这一过程中起重要作用，而对于其内核的裁剪和定制的研究必将是最核心的任务，这也是嵌入式 Linux 系统对软件产业的最大贡献。

1.3 本论文的主要内容

根据课题研究的目的及目前存在的问题，本论文主要从以下几个方面展开工作：

（1）系统研究 Linux 内核的各个版本，从 Linux0.11 到 Linux2.4 详细分析各个内核的特点和其所实现的功能有什么不同，一些在早期版本上未实现的功能是如何整合到后期版本中的。而操作系统的某一部分,就其实现来说,看起来比较简单。它的难点是如何将大量的功能集成出一个 kernerl。从这些工作中我们发现一个完整可靠的内核需要哪些机制和接口，在此基础上力争全面理解 Linux 内核。

（2）逐步修改内核，其中提出各种设想，在实验基础上，自己去尝试实现各种机制，发现自己不足的过程中，做到深刻理解内核复杂性和其各部分相关关系，并掌握内核各种重要数据结构。为今后的工作做好准备。

（3）了解嵌入式 Linux 的设计特点，提出裁减 Linux 内核的科学方法，用它可以实现在各应用平台上分离出 Linux 操作系统内核源代码中的必要部分和可裁减部分，针对各个可裁减部分，研究相应的裁减技术。

（4）具体分析内核各组成部分的特性,找到其相应的裁减和定制原则与方法,其各子系统的虚拟管理机制、缓冲管理机制在嵌入式系统里已成为负担，详细说明如何完全将这些代码从内核中裁去，并分析裁去的可行性。

（5）嵌入式是微处理器、单片机和 DSP 处理器的应用技术，Linux 不支持 DSP 平台，这将制约其在嵌入式领域的发展；在了解了内核所有源代码工作特性的基础上，致力于

将 Linux 内核移植到 DSP 平台的研究。

（6）以嵌入式环境为应用背景，讨论如何降低驱动程序开发难度，并以虚拟字符设备抽象硬件体系特征，减少开发人员的代码工作量。

2 Linux 内核具体结构分析

概括而言，操作系统主要具有两个功能：其一是管理硬件资源，其二是屏蔽具体硬件差异并为应用程序提供虚拟机[10]。于是，操作系统必定由进程控制、内存管理、设备驱动、文件系统等子系统构成。进程控制、内存管理等核心部分与目标计算机的体系结构密切相关，必须针对目标计算机单独开发；而设备驱动、文件系统和网络部分只涉及具体的外设，与处理器结构无关。

内核是操作系统的内部核心程序，它向外部提供了对计算机设备的核心管理调用。我们将操作系统的代码分成两部分。内核所在的地址空间称作内核空间。而在内核以外，剩下的程序统称为外部管理程序，它们大部分是对外围设备的管理和界面操作。外部管理程序与用户进程所占据的地址空间称为外部空间。通常，一个程序会跨越两个空间。当执行到内核空间的一段代码时，我们称程序处于内核态，而当程序执行到外部空间代码时，我们称程序处于用户态。

从 UNIX 内核起，人们开始用高级语言编写内核代码，使得内核具有良好的扩展性。单一内核是当时操作系统的主流，操作系统中所有的系统相关功能都被封装在内核中，它们与外部程序处在不同的内存地址空间中，并通过各种方式防止外部程序直接访问内核中的数据结构。程序只有通过一套称作系统调用的界面访问内核结构。近些年来，微内核结构逐渐流行起来。成为操作系统的主要潮流。1986 年，Tanenbaum 提出 Mach kernel，之后，他的 minix 和 GNU 的 Hurd 操作系统更是微内核的典范。在微内核结构中，操作系统的内核只需要提供最基本、最核心的一部分操作（比如创建和删除任务、内存管理、中断管理等）。而 Linux 内核并不完全局限于单内核特性，有其独特的结构体系。嵌入式 Linux 内核一般由标准 Linux 内核裁剪而来。用户可在完全了解内核源代码的基础上，根据需求配置系统，剔除不需要的服务功能、文件系统和设备驱动[11]，来使内核适用于各种嵌入式设备。

2.1 Linux 内核的系统体系结构

Linux 内核主要由5 个模块构成，它们分别是：进程调度模块、内存管理模块、文

件系统模块、进程间通信模块和网络接口模块。进程调度模块用来负责控制进程对CPU资源的使用。所采取的调度策略是各进程能够公平合理而有效地访问CPU，同时保证内核能及时地执行硬件操作。内存管理模块用于确保所有进程能够安全地共享机器主内存区，同时，内存管理模块还支持虚拟内存管理方式，使得Linux 支持进程使用比实际内存空间更多更大的内存容量。并可以利用文件系统把暂时不用的内存数据块交换到外部存储设备上去，当需要时再交换回来。文件系统模块用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备的不同细节。从而提供并支持与其它操作系统兼容的多种文件系统格式。进程间通信模块子系统用于支持多种进程间的信息交换方式。网络接口模块提供对多种网络通信标准的访问并支持许多网络硬件。

这几个模块之间的依赖关系见图 2.1 所示。其中的连线代表它们之间的依赖关系，虚线和虚框部分表示 Linux 后期版本中实现的的模块和关联部分。

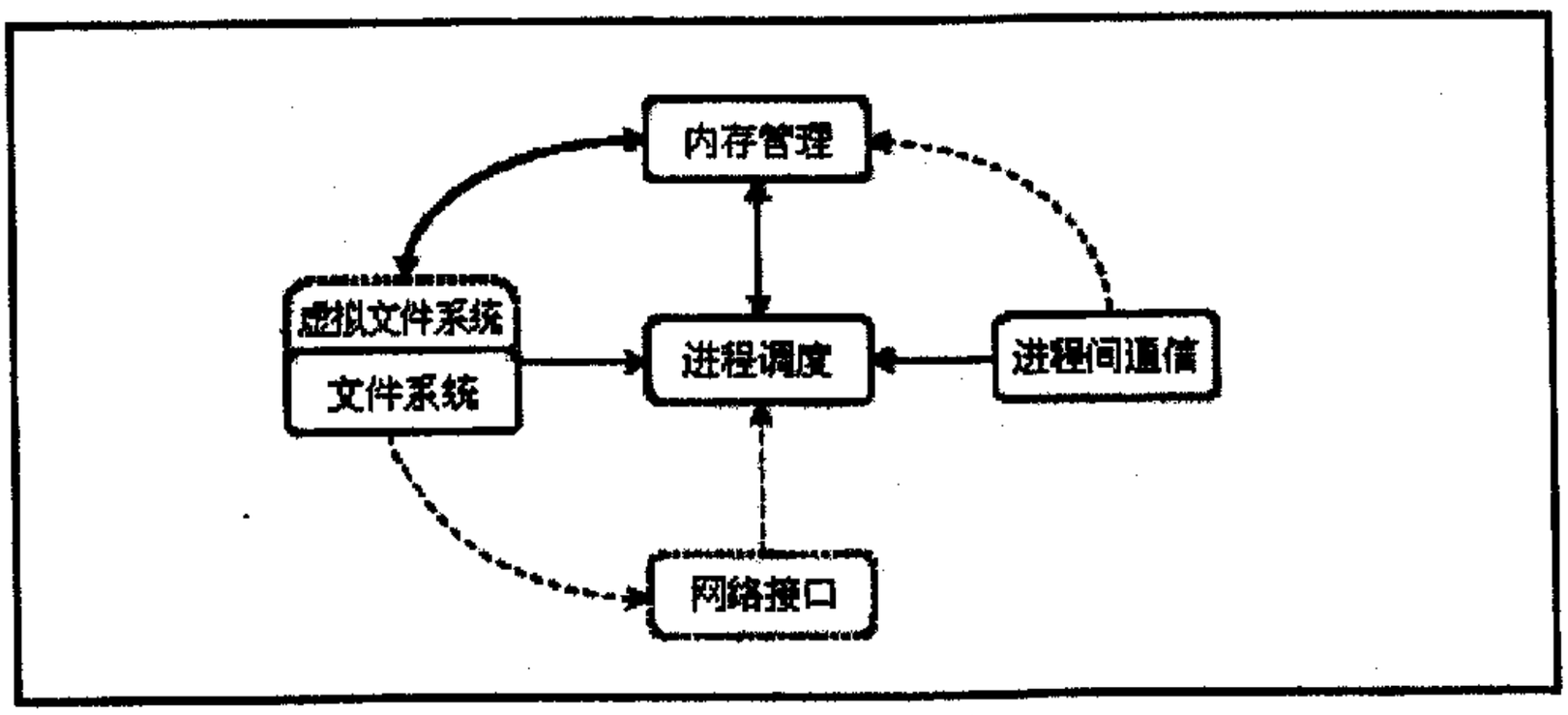

图2.1　Linux内核系统模块结构及相互依赖关系

由图可以看出，所有的模块都与进程调度模块存在依赖关系。因为它们都需要依靠进程调度程序来挂起（暂停）或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作完成后才可继续运行。例如，当一个进程试图将一数据块写到软盘上去时，软盘驱动程序就可能在启动软盘旋转期间将该进程置为挂起等待状态，而在软盘进入到正常转速后再使得该进程能继续运行。另外3个模块也是由于类似的原因而与进程调度模块存在依赖关系。其它几个依赖关系有些不太明显，但同样也很重要。进程调度子系统需要使用内存管理器来调整一个特定进程所使用的物理内存空间。进程

间通信子系统则需要依靠内存管理器来支持共享内存通信机制。这种通信机制允许两个进程访问内存的同一个区域以进行进程间信息的交换。虚拟文件系统也会使用网络接口来支持网络文件系统(NFS),同样也能使用内存管理子系统来提供内存虚拟盘(ramdisk)设备。而内存管理子系统也会使用文件系统来支持内存数据块的交换操作。

其中进程管理、内存管理和文件系统是其最基本的3个子系统。可将这三部分构成是Linux内核的单内核模式结构模型[12]。用户进程可直接通过系统调用或者函数库来访问内核资源。正因为Linux内核具有这样的结构，因此修改内核时必须注意各个子系统之间的协调。

2.2 Linux 内核部分分析

根据功能不同，Linux内核的模块可基本划分为进程管理、内存管理、网络系统、文件系统与进程通信几部分。这种划分是从用户程序的角度为出发点的。从内核开发本身的角度看，Linux内核应包括初始化、中断处理、进程调度、内存管理几个基本部分。在此基础之上引入文件系统、网络系统、总线管理系统等。而与之相关的设备驱动程序的管理也是这些子系统的功能。本节将从内核的角度，详细介绍Linux内核中的系统初始化、中断处理、进程调度等部分。

2.2.1 初始化

Linux是一个跨平台的系统，其内核代码中与平台无关的部分是从init/main.c文件中的start_kernel(void)函数开始执行的。在此之前所执行的程序主要完成内核运行环境的建立，包括查找处理器类型、查找机器结构、建立系统运行所需的最小内存页表、激活MMU、设置系统初始化任务栈、保存处理器ID与机器类型等参数供C代码部分使用。

从start_kernel()开始，内核将完成其余的初始化工作。下面重点分析从start_kernel()开始的系统初始化工作。

start_kernel()先调用lock_kernel（），由于嵌入式环境中，一般不会用到Linux的SMP支持，与SMP相关的一些函数一般定义为空循环，没有实质意义。

初始化过程然后调用setup_arch()。该函数完成于特定机器相关的一些数据初始

化。这些数据包括：

- ROOT_DEV，这些参数决定了系统在装载根文件系统时使用哪一个设备上的文件系统；
- Proc_info，该结构记录了CPU名称与 CPU制造商名称；
- System_utsname，该结构记录了系统名称、机器名称、系统版本号、域名等；
- Init_mem,该结构记录了内核中代码的起始位置、数据起始位置等存储器信息。
- Meminfo该结构中记录了系统中存储器的分布情况，包括系统有几个存储器bank，每个bank大小。

除了初始化上述数据结构外，setup_arch() 还初始化了系统的页表，对传递给系统的命令行参数进行整理，并进行部分处理，总的来说，系统运行所需要的基本数据都在这里初始化[13]。setup_arch()是与体系结构紧密相关的。

初始化过程接着调用parse_options()函数对命令行参数进行分析处理，将命令行参数中的各个环境变量分割出来保存到 envp_init数据中，将命令行参数的内核选项分离并保存到 argv_init 数组中。这两个数组将用于后续的初始化过程。

分析命令参数后，调用trap_init()对系统中断向量表进行初始化。中断向量初始化时系统与体系结构相关的接口函数是_trap_init()。在ARM结构中，该函数在文件arch\arm\kernel\entry_armv.S中实现。该函数将中断向量拷到位于虚地址0x00000000的内存中，同时将可重定位的中断处理程序头部（stub）拷贝到虚地址 0x00000200的内存中，从而完成中断向量表的初始化过程。

中断向量表初始化后，调用init_IRQ()函数对中断处理分配表irq_desc数组进行初始化，为各个中断分配缺省处理函数。中断处理是操作系统中一个重要的部分，他与CPU结合得比较紧密，不同CPU的中断处理过程往往相差较大，因此Linux中与中断处理相关代码大部分是在与体系结构相关的，其接口也相对较粗。

中断分配表初始化后，调用 sched_init() 对进程调度所用到的数据结构进行初始化。sched_init()所初始化的数据中包括：

- pidhash 数组，这是一个 task_struct结构的指针数组。该数组初始化为空指针。pidhash 数组用于根据pid（进程标识号）来快速查找其对应的task_struct结构体。它所用的hash函数为((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))。由

于linux可同时支持多个进程，通过hash表来查找pid所对应的任务结构可大大加速查找的速度。

- tv1、tv2、tv3、tv4、tv5。tv1- tv5 时时间向量，用于处理系统的计时器功能。它们的初始化是通过调用init_timervecs()进行的。
- bh_base 数组。bh_base 是一个函数指针，用于系统的中断下半部(bottom half)处理。sched_init()初始化了bh_base 数组的三个成员：TIMER_BH、TQUEUE_BH、IMMEDIATE_BH。这是通过调用init_bh(int nr,void(*routine)(void)) 函数完成的。

调用sched_init()之后，调用time_init()函数对系统时钟变量xtime初始化，然后安装系统时钟中断处理函数。

初始化过程接着调用 softirq_init()函数进行软中断处理初始化。它所涉及的数据结构bh_task_vec数组、softirq _vec数组。这些数据用于实现系统的bh机制与tasklet机制。

调用 console_init() 进行控制台初始化。Console_init之后，内核中的printk()语句就可以产生打印输出了。控制台初始化之后，系统调用mm/slab.c中的函数kmem_cache_init()，该函数初始化Linux 的slab分配器中的基本数据结构cache_cache、cache_chain_sem、cache_chain。这几个数据结构用于slab 分配器中的cache管理。

初始化过程接着调用mem_init() 函数，该函数根据meminfo结构的信息初始化系统基于页的内存管理系统的所有数据结构，标识系统中所有空闲内存的情况，并输出系统最初的内存使用情况。mem_init()之后，系统就可以使用gfp（即页面分配与释放）功能了。

mem_init()之后系统调用kmem_cache_sizes_init()函数，它的作用是建立系统内部以及通用的cache.。

系统随后调用fork_init(mempages) 函数，它的作用是根据系统中的空闲内存的多少（即 mempages值）确定系统所允许的线程数量，其确定标准为线程结构最多可占用一半的系统内存。

之后调用proc_caches_init()函数创建进程管理子系统用到的几个cache，它们包

括：signal_act 、files_cache、fs_cache 、vm_area_struct、mm_struct等。

随后调用vfs_caches_init()创建虚拟文件系统的几个cache，它们包括：buffer_head、 names_cache、filp、dquot、dentry_cache等，并初始化dentry 哈希表dentry_hashtable。

初始化过程接着调用buffer_init()函数初始化buffer哈希表。关于虚拟文件系统所用到的这些数据结构

系统接着调用page_cache_init() 初始化page哈希表page_hash_table，该数据用于系统中文件内存映射功能。

继page_cache_init()之后，初始化过程将分别调用 kiobuf_setup()、signal_init()、bdev_init() 、 inode_init()、 ipc_init() 函数。其中：kiobuf_setup() 函数建立kiobuf 类型的cache，用于抽象的内核空间 io 缓冲；signal_init()函数建立sigqueue 类型的cache，用于系统的信号处理队列管理；bdev_init() 函数初始化块设备系统，建立block_device结构类型的cache；inode_init() 函数初始化inode哈希表，并建立inode结构类型的cache；ipc_init() 函数则初始化信号、共享内存与消息队列三种通信进程机制。

系统初始化过程执行到此时，系统最基本的部分已经初始化完毕，在接下来的初始化过程中将初始化其他的子系统，如网络子系统、各子文件系统、PCMCIA/PCI等总线子系统，并初始化内核中的设备驱动程序，释放内核初始化部分的代码区域，最后加载根文件系统，并开始执行用户级的初始化程序，至此内核的初始化过程结束。

2.2.2 中断处理

中断处理是OS的一个重要部分，它是 CPU与外设通信的重要手段。本节介绍Linux的中断处理过程以及Linux为处理中断所提供的接口。

不同的CPU其中断方式有所不同，下面以目前32位、64位嵌入式微处理器中应用最广泛的ARM系列CPU[14]为例介绍Linux的中断处理过程。

①ARM的中断类型

ARM Linux的中断处理用到的核心数据结构是irq_desc[NR_IRQS]。这是个struct irqdesc类型的结构。NR_IRQS是系统所允许的最多中断数，下面是其定义：

ARM支持7种类型的中断，对应每种中断类型有一种优先级处理状态。ARM支持的7种中断类型是：

- 复位中断：复位中断是由处理器的复位信号引起的中断，系统加电时也进入复位中断处理向量。复位中断时系统进入SVC模式，MMU失效，系统从物理位置0x0处开始执行程序，中断与快中断均关闭。
- 无定义指令中断：当系统执行了非法指令时进入该中断，中断矢量位于0x4，处理器进入UNDEF模式，中断关闭，快中断状态不变。在Linux中，该中断用于在没有浮点支持的CPU中模拟浮点处理命令。
- 软件中断：系统执行SWI指令后进入软件中断，中断矢量位于0x8，处理器进入SVC状态，中断关闭，快中断状态不变。该中断用于实现Linux中的系统调用。
- 预取指令失败中断：当系统从内存中预取到的指令无效时，系统进入该中断处理，中断矢量位于0xc，处理器进入ABORT状态。
- 预取数据失败中断：当系统从内存中预取到的指令无效时，系统进入该中断处理，中断矢量位于0x10，处理器进入ABORT状态。预取指令失败中断与预取数据失败中断在Linux中用于系统的页面失效处理。
- 中断请求中断：当处理器的IRQ输入信号有效时，系统进入中断请求处理，中断矢量位于0x18，处理器进入IRQ状态。中断请求中断在Linux中用于一般外设中断处理，它也是Linux中使用最多的中断。
- 快中断请求中断：当处理器的FIQ输入信号有效时，系统进入快中断请求处理，中断矢量位于0x1c，处理器进入FIQ状态。这种类型的中断在Linux中没有很好的支持。

②Linux的中断处理要求与接口

Linux只定义了两种类型的运行状态即用户态与内核态，Linux的中断处理是在内核态下执行的。对于ARM系统CPU来说，系统存在User、ABORT、SVC、IRQ 、FIQ、UNDEF等执行状态，并且在这些状态下可见的寄存器也不一样，因此要求在进入到正规的中断处理程序之前先进入正确的CPU执行状态。

在ARM Linux中，用户态对应ARM的User状态，而内核态对应ARM的SVC状态。在ARM处理器发生中断时，CPU进入相应的运行状态，在这些状态下最初执行的几条指

令需要保存这些状态下运行环境，然后进入SVC状态，以便

Linux执行正常的中断处理程序。

Linux的中断处理部分为内核其他部分提供了标准的接口，这些接口可分为三类：

● 中断的申请与释放：

中断的申请：

int request_irq(unsighed int irq,void(*handler)(int.void*.struct pt_regs*). unsighed long irq_flags,const char*devname.void*dev_id);

其中irq为申请的中断号，handler未中断处理函数，irq_flags为申请的中断处理属性，dev_id用于标识共享中断的设备号。通常，中断处理函数的返回值为0表示申请成功，返回值为负时表示申请失败的错误码。

中断的释放：

void free_irq(unsighed int irq, void*dev_id);

其中irq为申请释放的中断号，dev_id为共享中断时相应的设备号，即注销与dev_id相应的中断处理程序。

● 中断的打开与关闭：

void enable_irq(unsighed int irq);

void disable_irq(unsighed int irq);

这两个函数的作用分别为打开与关闭与irq相关的中断。

● 中断的下半部处理：

中断的下半部分（bh）处理负责完成中断处理过程中比较耗时的那部分工作任务。中断程序可以调用函数

mark_bh(int nr);

来激活nr所对应的中断下半部分处理，驱动程序一般使用IMMEDIATE_BH进行中断的下半部分处理，而通过调用函数

queue_task(&short_task,&tq_immediate);

来将中断下半部分处理函数所在的tq_struct结构short_task挂到tq_immediate处理队列中，short_task.routine即为处理中断下半部分的函数，而short_task.data是传递给该函数的参数，用以标识下半部分处理函数。

③ARM Linux的中断处理

Linux的中断处理除下半部分（bh）处理以外都是与CPU紧密相关的。由于中断的打开与关闭以及中断的申请与释放只是对相关的数据结构进行了操作，或是调用了对中断控制硬件进行操作的函数，因此在这里就不加以详细分析，下面重点分析ARM Linux中的中断处理过程。

ARM Linux的中断处理过程主要是在do_IRQ()函数中进行的。中断处理在进入do_IRQ()函数之前所作的工作主要是中断现场的保护，并从相应的处理机运行状态转到SVC状态下运行。而从do_IRQ()函数返回之后中断处理程序所做的工作主要是根据进入中断处理前系统的运行状态来进行相应的中断退出处理；如果进入中断处理前系统运行在用户态下，即ARM的user状态，则恢复被中断的程序的执行环境，但并不恢复它的运行，而是调用schedule()调度其他合适的进行运行；如果进入中断处理程序之前系统运行在核心态，即ARM的SVC状态，则恢复被中断的程序的执行环境，并恢复该程序的运行。下面主要介绍do_IRQ()。

do_IRQ()函数原型为asmlinkage void do_IRQ(int irq.struct pt_regs);宏asmlinkage说明该函数可以从汇编程序直接调用，两个参数irq和regs分别表示引起中断的中断号与中断现场的寄存器值。

do_IRQ()函数主要完成两个功能：

首先，它根据irq的值，从数组irq_desc[]中取出元素irq_desc[irq],调用irq_desc[irq]。Action链表中的所有中断处理函数。

Irq的所有中断处理函数调用完毕后，do_IRQ检查系统中是否有软中断（包括bh处理、tasklet等）,如果有则调用do_softirq()执行软中断处理。

ARM Linux的中断处理用到的核心数据结构是irq_desc[NR_IRQS] 。这是个struct irqdesc类型的结构，NR_IRQS是系统所允许的最多中断数，下面是其定义：

```
Struct irqdesc{
    Unsigned int nomask   :1 ;     /* 中断处理时不屏蔽该中断 */
    Unsigned int enabled  :1 ;     /* 该中断当前允许 */
    Unsigned int triggered  :1 ;   /* 本中断已经发生 */
    Unsigned int probing  :1 ;     /* 本中断正在检测 */
```

```
Unsigned int probe-ok   :1 ;      /* 本中断可以被检测 */
Unsigned int valid   :1 ;         /* 本中断可以被申明 */
Unsigned int noautoenable   :1 ;  /* 不自动允许本中断 */
Unsigned int unused   :25 ;
Void(*mask_ack)(unsigned int irq)   /* 本中断的屏蔽且应答函数 */
Void(*mask)(unsigned int irq) ;     /* 本中断的屏蔽函数 */
Void(*unmask)(unsigned int irq) ;   /* 中断允许函数 */
struct irqaction *action ;          /* 中断处理函数队列 */
/*
*中断锁检测;
*/
Unsigned int lck_cnt ;
Unsigned int lck_pc ;
Unsigned int lck_jif ;
    };
```

struct irqaction定义为:

```
struct irqaction{
    void(*handler)(int,void*,struct pt_regs*);
    unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
    };
```

2.2.3 进程调度

进程调度是OS的基本组成部分。Linux中的进程运行在用户态时可以被抢占调度，但当进程运行在内核态时是以非抢占的方式进行调度的。关于Linux的进程管理、进程

在内核中的表示方式、进程的创建、进程的调度策略等已有很多文献作了详细的分析，在此，我们将结合 ARM 处理器分析 Linux 的进程调度过程。

Linux 的进程调度可以用图 2.2 来形象地表示：

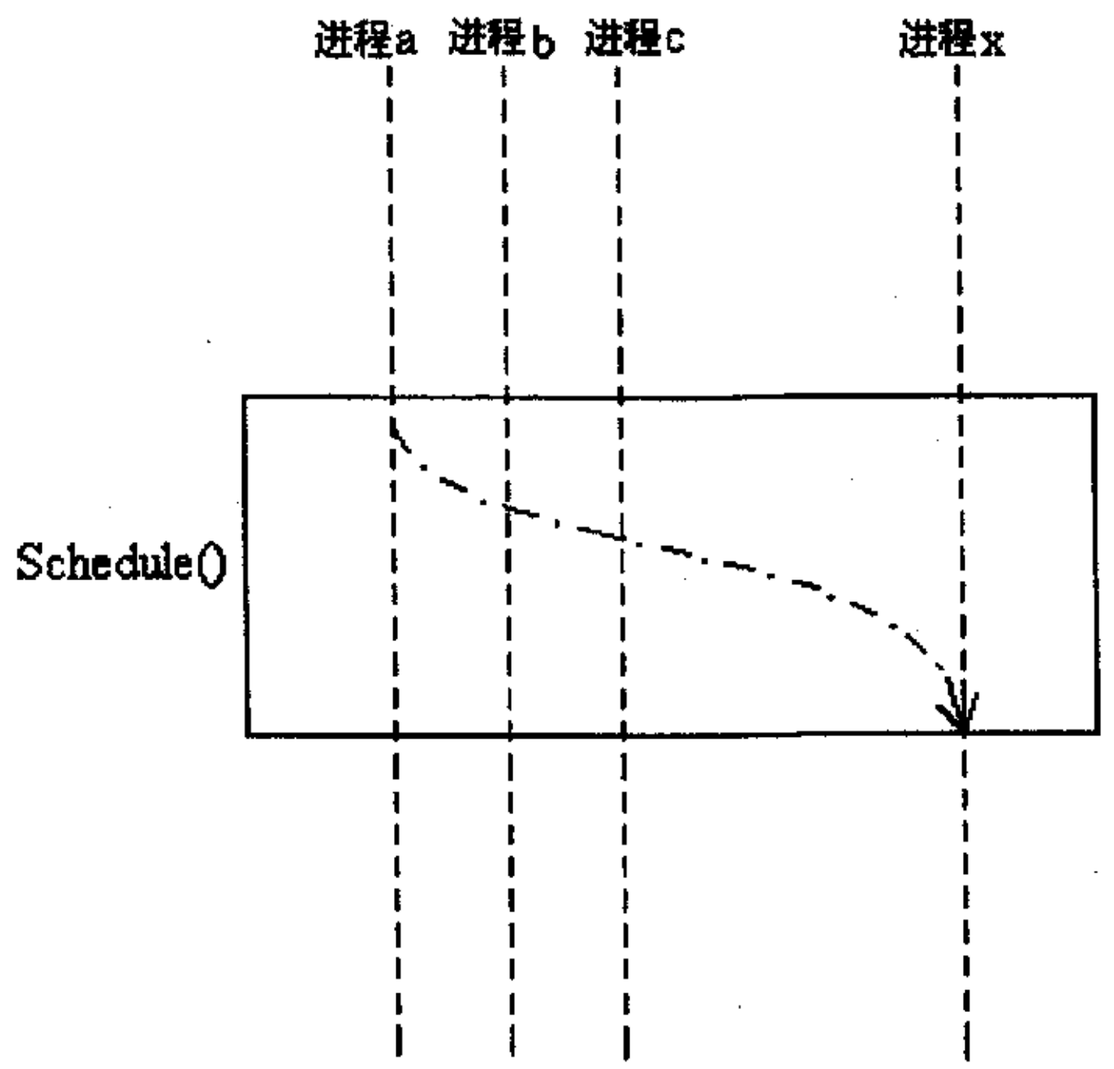

图 2.2　进程调度过程示意图

图中虚线部分表示单个进程，实线部分表示当前时刻正在执行的进程。当前正在执行的进程当发生系统调用而进入核心态执行，相应系统调用服务程序执行完毕后将调用 schedule()；或者，当前进程运行在用户态时系统发生中断，相应中断处理程序执行完毕，用户态程序运行环境恢复之后，从中断处理返回之前也将调用 schedule()。进入 schedule 程序之后，内核将从就绪进程队列中选取一个进程作为当前进程，schedule 程序退出之后即恢复该进程的运行。

Linux 的进程调度包括调度策略与上下文切换两个部分，其中调度策略是与 CPU 无关的，是 Linux 进程调度的核心部分，而上下文切换则是与 CPU 紧密相关的，它涉及许多硬件管理的细节，是进程调度程序的基础。

① 进程调度策略

SCHED_FIFO 意味着这是一个实时进程，对于该进程的调度需要遵守 POSIX.lb

标准的 FIFO 调度方式。该进程会一直进行，直到它发生 I/O 阻塞而释放 CPU,或者有一个实时优先级更高的实时进程在等待运行。

SCHED_RR 表示该进程是一个实时进程，对它的调度遵守 POSIX.Ib 的 RR 调度规则。该进程有一个时间片，时间片用完之后就使用相同的实时优先级将该进程放到 SCHED_FIFO 与 SCHED_RR 队列的末尾。

schedule()函数是在文件 kernel/sched.c 中实现的，其函数流程图如图 2.3 所示。

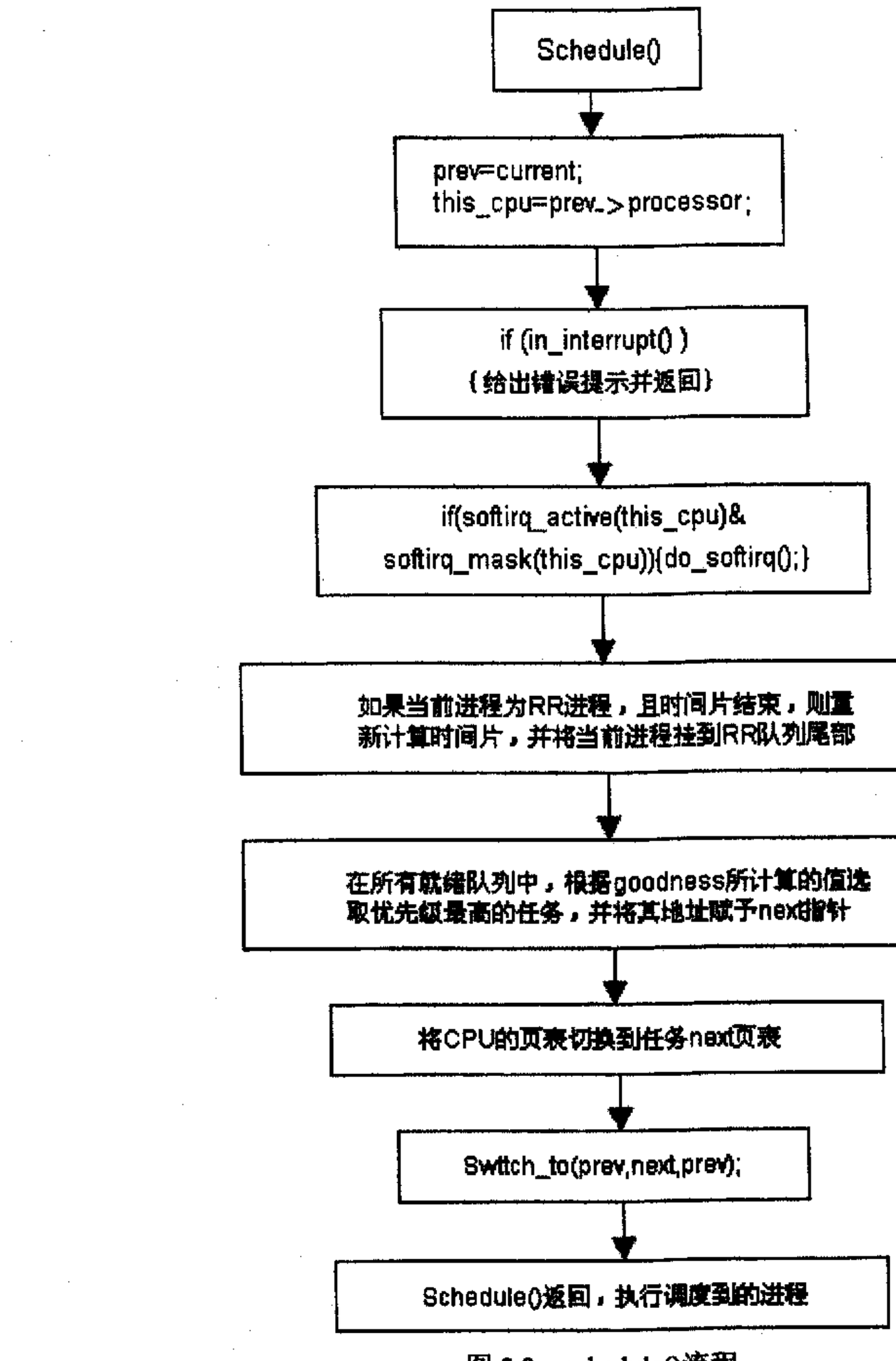

图 2.3　schedule()流程

schedule()函数首先确认当前任务的 active_mm 指针非空，然后将 prev 指针置为当前任务，并取得当前任务运行的 CPU 号。

调度函数然后判断当前指令是否在中断处理函数中运行，如果是则立即返回，如果不是则继续以下处理。

调度程序随后释放全局内核锁与全局中断锁，并判断系统中是否存在软件中断，如果有软件中断发生则先处理软件中断。

调度程序接着从就绪队列中选取一个最合适的任务，即 goodness()返回值最高的任务，并将 next 指针指向该任务，随后调用 switchto()函数完成进程的上下文切换。Goodness()函数的计算充分考虑了上述进程调度策略。

② Switch_to 函数实现

在 ARM Linux 的实现中，switch_to 的定义为：

```
#define swith_to(prev,next,last)        \
        do{                             \
        last=__switch_to(prev,next);\
        mb();                           \
          }while(0)
```

其中的关键函数为__switch_to(prev,next); 该函数是用汇编语言实现的：

```
ENTRY(__switch_to)
1:   stmfd   sp!,{r4-sl,fp,lr}
2:   mrs ip,cpsr
3:   str ip ,[sp,#-4]!
4:   str sp,[r0,#TSS_SAVE]
5:   ldr sp,[r1,#TSS_SAVE]
6:   ldr r2,[r1,#TSS_DOMAN]
7:   ldr ip,[sp],#4
8:   mcr pl5,0,r2,c3,c0
9:   msr spsr,ip
10:  ldmfd   sp!,{r4-sl,fp,pc}^
```

上面第1行程序将大部分寄存器保存在栈中，3保存在SVC状态的机器状态字cpsr,4保存SVC状态的栈指针，5恢复保存的SVC状态的栈指针，8设置域寄存器，9将保存CPSR的恢复到SPSR中，10恢复以前保存的其它寄存器。

以上函数是ARM系列CPU进程上下文切换的过程，在切换过程中将当前进程的任务环境保存到任务结构中，然后从需要切换到进程的任务结构中恢复任务环境。在ARM中，CPU的任务环境包括寄存器R4~R15,CPSR,SPSR等。

在操作系统术语中，有进程和线程这一对概念。一般进程拥有自己的虚拟地址空间（堆栈、数据、程序代码）和系统资源（例如打开文件）；而线程通常只有程序计数器、堆栈和寄存器，与其他线程共享地址空间和系统资源。线程比一般进程小，因而创建线程的速度快，操作系统在线程之间的切换不需要付出很高的成本。并且由于多个线程可以共享资源和变量，因而可以进行快速而有效的通信[15]。

以上完成了内核最基本部分的概述，并从实例出发，对各部分的接口、运行机制和数据结构进行了详细说明，是我们进行内核裁剪定制的基础。

3 内核裁剪研究

嵌入式 Linux(Embedded Linux)是指对标准 Linux 经过小型化裁剪后，应用于特定嵌入式场合的专用 Linux 操作系统。与其他嵌入式操作系统相比，Linux 的源代码是开放的，不存在黑箱技术[16]。Linux 作为一种可裁剪的软件平台系统，是发展未来嵌入式设备产品的绝佳资源。Ljnux 具有更小、更稳定、价格更具竞争力等优势，正是嵌入式操作系统的特殊要求，为 Linux 在嵌入式系统中的发展提供了广阔空间，使得 Linux 成为嵌入式操作系统中的新贵。在应用上，嵌入式 Linux 可应用于信息家电、多媒体手机、工业、商业控制、电子商务平台等领域。如何对标准 L1nux 进行裁剪，以适应特定的嵌入式应用就成为一个关键性的问题．通常在裁剪 Linux 的过程中还存在以下一些问题：①Linux 是结构相当复杂的大型操作系统，要手动配置 Linux 是十分繁琐的，目前还没有一个正式有效的方案来解决这一问题；②很难保证配置后内核的完整性；③尽管 Linux 为用户提供了动态添加或删除模块的功能[17]，但是实现这一功能的程序代码仍然占用一定的存储空间，这对于资源有限的嵌入式系统是以提高成本为代价的。本文将提出一种新的调用图方法来裁剪 Linux，成为面向具体应用的嵌入式 Linux。

3.1 嵌入式 Linux 的设计

嵌入式系统是一种使用专门设计的硬件平台，并且运行专有软件而达到特殊应用目的的系统。相对于传统的设备而言，嵌入式系统提供了可编程性和可操作性；相对于计算机而言，嵌入式系统又需要削减体积，减少一些不必要的功能，增强一些特殊性能。总的来说，嵌入式系统主要有如下一些需求：

- 低成本和短的进入市场时间
- 响应时间和实时性
- 轻量级
- 安全和可靠性

传统的嵌入式系统设计开发主要是软件和硬件的协同设计。首先是硬件和软件的功能划分，确定每项功能由硬件完成还是软件完成；然后是软件和硬件的并行设计实现，

并且分别测试；最后是软件和硬件的整合和测试。

随着 IC 集成度的提高和生产规模的扩大，嵌入式系统的硬件环境已经有了很大的进步。高性能嵌入式微处理器已经得到了广泛的使用．硬件集成度的增强简化了嵌入式系统硬件环境的建立过程．尤其是在 SoC 出现之后，硬件环境的搭建变得更加容易了。SoC 是一种结合了多个功能模块的电路系统，将需要的功能实现在一个芯片上，从而可以借助集成电路的集成度来缩小整个系统所占的体积。SoC 的产品设计一般都是以知识产权(intellectual property，IP)的方式提供．更新换代 SoC 产品时，只需要将原来的 IP 转移到新的嵌入式微处理器设计中，并修改部分电路，就可以满足新的要求。IP 的重复利用，减少了嵌入式硬件设计的复杂性，使得传统的嵌入式系统设计从硬件和软件的并行设计转移到以软件为中心的角度上来。所以，为了满足嵌入式应用的需求和提高开发速度，采用 SoC，减少硬件开发周期和复杂度，配以合适的嵌入式软件，成为了现在嵌入式系统开发的重要途径。目前大量使用的 SoC 主要有 ARM、StrongARM、MIPS、xScale，等等[18]。

使用 Linux 作为嵌入式操作系统开发的基础，可以适应基于 SoC 的硬件环境，提高软件方面的开发速度。Linux 是一种开放源代码形式的广泛使用的操作系统，它的内核支持多达几十种处理器体系结构，包括上面提供的那些 SoC 嵌入式平台。可以修改 Linux 内核中的各个模块以达到满足嵌入式应用的需求。

嵌入式 Linux 是根据嵌入式系统的不同需求，逐渐修改和更新 Linux 内核代码形成的。嵌入式 Linux 和一般的 Linux 区别集中在 Linux 内核上：前者的内核为嵌入式目标平台将通用 Linux 做了专门设计和修改，后者的内核应用于通用 PC 平台。

在嵌入式系统的历史中，使用通用操作系统作为嵌入式系统原型的实例并不只有这一例，如 Vx—Works 是基于 4.3BSD 操作系统改造的，NT Em—bedded 是基于 Windows NT 操作系统改造的[19]。改造的途径可以使得原来在通用操作系统上运行的应用可以快速方便地移植到嵌入式版本的系统中来。Linux 也是一样，广泛开放源代码的 Linux 应用已经被移植到嵌入式的运行环境中。因此，使用 Linux 开发嵌入式操作系统，可以加快嵌入式系统的开发速度，缩短产品进入市场的时间。

嵌入式 Linux 的设计要点主要包括 bootloader 和操作系统的内存管理、进程管理等部分。

(一) bootloader 固件

固件(firmware)是指在硬件的 ROM 或者 Flash 中驻留运行的程序，配合主要程序完成系统任务。bootloader 固件固化在目标板的 ROM 中，用于启动存放在 RAM 中或者 ROM 中的嵌入式软件的程序。如果 CPU 支持，在 bootloader 开发过程中也可以将它放在 RAM 中运行。

在嵌入式 Linux 的开发中，首先要根据硬件目标板的特性开发出 bootloader 程序，以完成下面的任务：

(1) 初始化处理器

使用 bootloader 初始化处理器中的一些配置寄存器。比如需要使用 ARM720T 处理器的 MMU 的话，就应当在 bootloader 中使用控制命令打开 MMU。

(2)初始化必备的硬件

使用 bootloader 初始化和设置板上的必备硬件，比如初始化内存、Flash ROM 和中断控制器等。从主机下载系统映像到硬件板上的接口设备也是由它完成初始化的。比如，有些硬件板使用以太网传输嵌入式系统映像文件，那么在 bootloader 中会使用以太网卡驱动程序初始化硬件，随后与客户端的 bootloader 客户端程序通讯，并完成下载工作。

(3)下载系统映像

系统映像的下载只能由bootloader提供.通过bootloader提供的命令行或者交互Shell界面可以指定内核映像和文件系统映像的下载位置，也可以检查目标板上内存地址中的内容．在目标端的 bootloader 程序中提供了接收映像的服务端程序，在主机端的程序提供了发送数据包动作——可以通过串口，也可以通过以太网卡等其他方式发送．发送系统映像结束之后，如果硬件允许，bootloader 还可以提供命令将下载成功的映像写入到 Flash ROM 中。一般 bootloader 都提供对 Flash 的驱动、支持擦写命令，为写入 Flash 和检查 Flash 带来了很大的便利。

(4)初始化操作系统并准备执行

使用 bootloader 可以启动已经下载好的系统。可以指定 bootloader 启动在 RAM 中或者 Flash 中的系统，也可以指定具体的启动地址。

(二) 内存管理

操作系统的内存管理功能用于向操作系统提供一致的地址映射功能和内存页面的

申请、释放操作。如果没有操作系统，这些工作由嵌入式应用程序本身完成，也就是说，每个应用程序需要管理好自己的内存空间，通过嵌入式系统程序员在编程过程中保证内存访问不会越界，从而来保证程序运行的安全性。比如每个任务自己只能申请局部的静态内存空间，在运行的过程中别的任务都不可能得到，也就不存在越界的问题．但是静态的方式加剧了内存资源使用的浪费程度。如果有操作系统，嵌入式应用程序可以根据需要向操作系统申请内存空间，由操作系统统一管理。操作系统可以根据申请和释放时应用程序提供的参数，申请不同需求的内存，从而简化了嵌入式应用程序的开发，并保证了运行的安全稳定。

㈢ 任务管理

使用操作系统可以实现多任务调度。系统设计人员只需要设计任务，然后通过操作系统本身的调度方法便可完成多任务调度。通用操作系统的任务以进程(process)或者线程(thread)的方式实现，一般情况下不允许改变操作系统的调度方式．但是嵌入式操作系统以任务为核心，而任务本身就可以要求系统提供给自己合适的调度方式。比如说很多嵌入式操作系统都可以提供实时任务的调度方式，同时也可以提供非实时任务的调度方式。

普通 Linux 使用的调度算法是不可抢占式的分时调度算法。Linux 还可以通过任务描述字区分任务类型，分别使用不同的任务调度算法。通过上一章的分析我们可以知道，普通 Linux 中可以支持的三种任务类型及对应的调度算法：

可以通过增加策略标志以及对应的任务类型和调度算法的方式实现新的 Linux 任务调度算法。

将 Linux 改造成为支持实时任务的嵌入式操作系统，主要有两个方面的问题需要考虑：

1)中断处理

对外部中断的响应能力决定了系统的实时性能．在操作系统内核中，有一部分的操作需要在关闭中断情况下运行，这必然会延迟对外部中断的响应。可以通过双内核的模式，在 Linux 内核和硬件中断之间增加一个实时内核，而 Linux 内核本身作为该实时内核的一个任务进行调度。实时内核运行时，不会发生关闭中断的操作，只要是需要实时响应的中断，都会触发实时内核运行实时任务[20]，抢占 Linux 内核的运行；其他中断则

被传送到普通 Linux 内核处理。采用这种实时机制的内核有 RT-Linux、DI-APM-RTAI 等。

2) 进程抢占

Linux 是作为一种通用操作系统设计的，吞吐量是它设计过程中的重要因素。尤其在单处理器模式下，可抢占的调度算法会使得在抢占切换过程中需要许多操作完成临界区的保护，产生额外的开销，对系统吞吐量产生严重的影响．因此通用 Linux 是不支持抢占式调度的．但是用于实时环境下的 Linux 不能将实时任务和一般任务一视同仁，而应该根据任务优先级，让实时任务抢占一般的低优先级任务，才能获得良好的实时性能。

有两种方法修改 Linux 内核以提高实时任务抢占非实时任务的能力：第一种是抢占点方法。在内核运行路径中设置一系列的抢占点，检查是否需要运行实时任务。这种情况下两个抢占点之间最长的路径就是最高优先级的实时任务被调度的最长时间。在 Red Hat 公司的 Ingo Molnar 提供的内核 patch 中实现了这种方法[21]。第二种方法是直接将内核改造成可抢占式内核．当高优先级的实时任务被唤醒时，只要当前任务不在临界区内，就直接抢占当前任务。在 Monta Vista 公司的 HardHat Linux 中实现了这种抢占式内核。

通常，一个编译好的内核主要是由以下几个部分组成的。

① 初始化程序段(init)。由内核中所有初始化程序的代码组成这部分代码是用__init 定义的，在内核初始化结束后，它们所占的存储空间就会被释放以节省 RAM 资源。一般有 32K 左右。

② 数据段(data)。由内核中所有含初始化值的全局变量数据结构组成。因为这些数据都是在内核运行的过程中被读写的变量，所以这个段必须放在 RAM 中。一般有 50-IOOK 左右

③ 未初始化数据段(bss)。由内核中所有未被初始化的全局变量·数据结构组成。这个段也必须放在 RAM 中，但这个段中的变量只有地址和大小没有初始值，在最后生成的二进制下载映像文件中并不占用空间。这个段所对应的 RAM 区域通常在内核启动前用“0”值来初始化·一般在 RAM 中将会占用 IOOK-150K 左右。

④ 代码段(text)。由内核中所有非初始化程序的代码组成。这个段是内核代码的主体部分。为了节省 RAM 空间，可以把它放到 ROM 中执行。

⑤ 文件系统(romfs)。 就拿 romfs 来说，它是 uCIinux 缺省使用的一种文件系统

类型， 比较简单实用，支持 romfs 文件系统的代码占用的 RAM 也比 Ext2 的小很多，非常适合嵌入式应用的需求，romfs 可以放在 RAM 中，也可以放在 ROM 中.它的大小要视添加应用程序的多少而定。比较典型的一个包含 init 和 sh 程序的 romfs 大小为 80K 左右。

而内核编译链接后得到的代码，主要由 4 个段(segment)组成。其中包括 init （初始化段）、data（数据段）、bss（未初始化数据段)和 text(代码段)。这 4 个段中，前面 3 个都需要有写权限，因此只能放在 RAM 中;而最后的代码段只需要可读就行了，因此可以放在 Flash 中，以节省 RAM 空间。这些代码中究竟应该包含内核的哪些功能，完全可由设计需求来定，也就是内核的哪些代码必须被裁去，而哪些又必须保留下来，可以由应用所需实现的功能决定。

3.2 一种基于调用图的裁剪 Linux 方法

Linux 是一个庞大、高效且复杂的操作系统，虽然它的开发起始于 Linus Torvalds 一个人，但随着时间的推移，越来越多的人加入了 Linux 的开发和对它不断完善，这使得 Linux 的内部结构更加复杂[22]，因此，裁剪 Linux 的第一个难点就是准确理解它的内部结构。调用图(Call Graph)就是解决这一难题的有效方法，调用图通常用来表示程序中过程之间的调用关系，根据调用图，可以提取出软件系统的可重用组件。因此，调用图通常应用于软件的重构和维护上。

利用调用图来裁剪 Linux 的方法分为六步.

1)构造应用程序调用图　目前,Linux 上运行的应用程序几乎都是用 C 语言编写的，因此很容易构造出此应用程序的调用图。

程序的调用图定义为有向图 C＝(v，R)，其中 v 表示程序中所有过程的集合，每个过程是调用图中的一个顶点；R 表示过程之间调用关系的集合，即

$R=\{(r_1, r_2) \mid r_1, r_2 \in v$ 且 r_1 调用 r_2 一次以上$\}$.

由此可见，调用图反映了程序的静态结构. 定义

S(Main)＝{P | P∈v 且从顶点 main()到顶点 p 存在一条路径}，显然 S(Main)≦v.

因此，如果存在一个过程 Q∉S(Main)，那么 Q 就是此应用程序不需要的过程，应该将过程 Q 从该程序中删除。尽管过程 Q 不会在 Linux 中引起任何错误，但也无法肯定可

安全将其移植到嵌入式系统上，所以最好将其删除．如应用程序形式化算法如下

```
main( )                          d( )
  {a( ); b( ); c( );}              {…….}
a( )                             e( )
  {…….}                            {f( );}
b( )                             f( )
  {d ( );}                         {…….}
c( )
  {d ( );}
```

由此生成的应用程序调用图如图 3.1。从图中可知过程 e()和 f()是可以删除的。

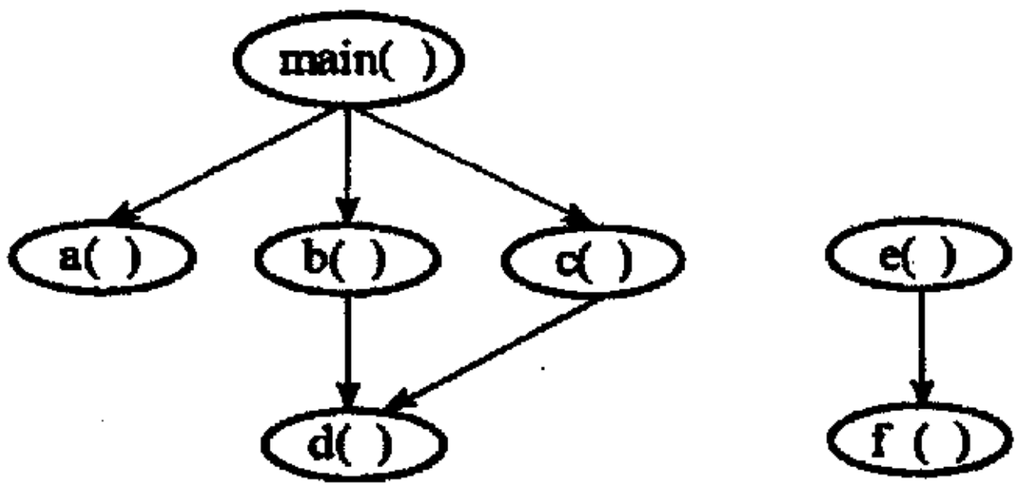

图 3.1　根据应用程序代码生成的应用程序调用图

2)构造共享库调用图　共享库对于 Linux 非常重要，几乎所有的应用程序都要调用共享库。共享库包含了大量的基本函数供程序使用，如打开文件，在屏幕上打印信息，从使用者得到反馈等等。然而，共享库文件通常有几兆字节大，必须对其进行裁剪，才能满足嵌入式系统的要求。通过分析共享库调用图，解决这一问题将不再困难。如图 3.2 是一个简化的共享库调用图，它可以表示共享库的调用结构，但是其中没有像应用程序调用图的 main()那样的初始顶点，应用程序提供了多个入口。

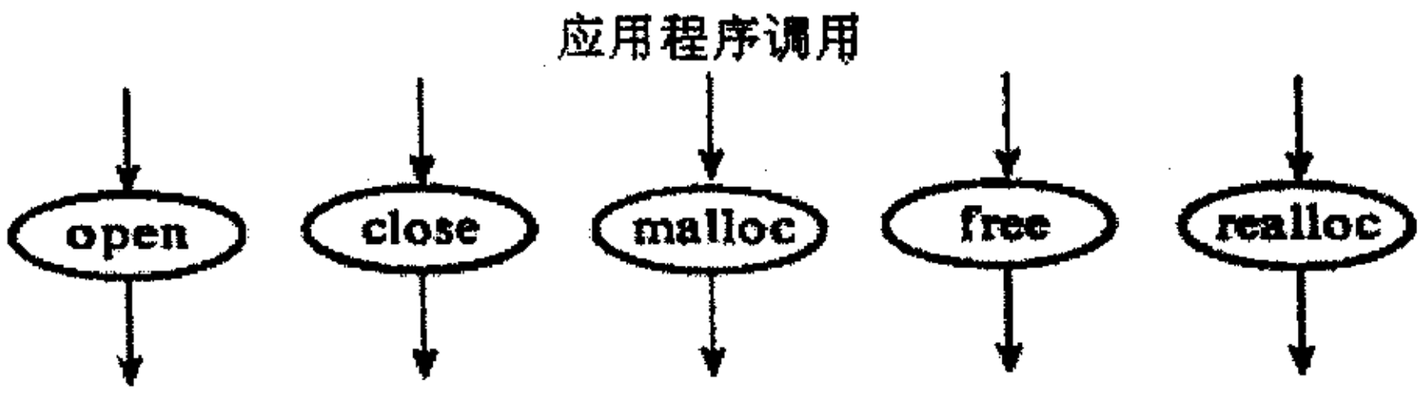

图 3.2 简化的共享库调用图

3)构造内核调用图　内核是 linux 的核心部分，它控制并协调所有的指令操作，由于内核的重要性及其庞大的结构，多数嵌入式 Linux 都没有改写它，主要是考虑到确保内核的安全执行。因此，仍然有许多不必要的代码保留在内核中。

构造内核调用图的方法与构造共享库调用图的方法类似，Linux 是中断驱动系统，当有事件通过中断或异常向 CPU 提出服务请求时，内核就启动相应的处理程序。因此，要构造内核调用图必须首先弄清内核函数将在何时执行。通常，内核函数会在下列 3 种情况下执行：①应用程序调用系统调用；②发生异常；③发生中断。

前两种情况是针对应用程序的，第三种情况是针对硬件设备的。为了找出不必要的内核函数，可以从上述 3 种情况的内核调用图入手，对于前两种情况，可以再结合应用程序调用图和共享库调用图，这样将产生一个 S(Main)，不在这个集合中的过程就可以删除。

4)删除不必要的硬件驱动程序　标准 Linux 支持多种硬件设备，但嵌入式系统是面向特定硬件平台的，有许多硬件设备是不需要的。因此，在裁剪过程中必须找出嵌入式系统不需要的硬件驱动程序的相关代码，并将其删除。

5)提取库函数和内核函数　综合应用程序调用图、共享库调用图和内核调用图，提取出应用程序所需要的库函数和内核函数。

如上所述，共享库调用图没有初始顶点，因此，把应用程序调用图与共享库调用图合并。这样，应用程序的初始节点 main()就成了整个混合调用图的入口。根据 Smain()就能够提取出不必要的库函数。

由于共享库是连接应用程序和 Linux 内核的通道，所以合并内核调用图和共享库调用图，就能够找出哪些内核函数与库函数相互作用。这样，被调用的内核函数就被添加

到S main()中，从而，提取出不必要的内核函数。

6)测试新内核找出不必要的过程后，在这里就可以将其从Linux中删除，但是必须确定新内核的正确性，如果新内核有错误，必须核对调用图并生成正确的调用图。

确定了内核中哪些部分要被裁去后，下面就是进行内核裁剪工作了。但要具体问题具体分析，首先每一个部分的裁剪具有不同的方法；而具体到每个部分的代码的裁剪方法也是各不相同的，因此必须对其采取相应的裁剪方法和步骤。下面就以核linux-2.4.x各个部分的具体裁剪为例说明。

3.3 去除多余的驱动程序

标准的Linux 内核是整体式的结构，所有的驱动程序都在内核空间运行，所以在内核的源代码中，驱动程序占据超过80%的体积。我们应该首先在内核中去除那些不需要的驱动程序。

㈠ 首先，在 drivers/目录中的 Makefile 文件中，把不需要的子目录配置删除掉。

例如： subdir-$(CONFIG_SCSI)+= scsi

接着， 在 arch/arm 目录中的 config.in 目录中把下面的代码删除，即可：

```
mainmenu_option next_comment
comment 'SCSI support'
tristate 'SCSI support' CONFIG_SCSI
if [ "$CONFIG_SCSI" != "n" ]; then
   source drivers/scsi/Config.in
fi
endmenu
```

还有包括 pnp 等的一些驱动程序目录都可以从此处删除掉。

㈡ 另外，还有些驱动程序不能一次性的从内核源代码中把目录去除掉的。对于这些程序我们只能够在驱动程序所在的目录的 Config.in 或 config.in 文件中去掉相应的选项，把它的代码从Linux 的源代码树中拿掉。例如，有一种PCMCIA 的网卡是3Com公司出产的类型是574，我们想把这个驱动程序从内核中摘除掉，可以这样制作： 首先

从 drivers/net/pcmcia/ 中的 config.in 文件中把

dep_tristate '3Com 3c574 PCMCIA support' CONFIG_PCMCIA_3C574 $CONFIG_PCMCIA

这行删除掉，然后再从

```
if{ "$CONFIG_PCMCIA_3C574" = "y" -o "$CONFIG_PCMCIA_3C574" = "y" -o\
    $CONFIG_PCMCIA_FMVJ18X" = "y" -o "$CONFIG_PCMCIA_PCNET" = "y" -o\
$CONFIG_PCMCIA_NMCLAN" = "y" -o "$CONFIG_PCMCIA_SMC91C92" = "y" -o\
    $CONFIG_PCMCIA_XIRC2PS" = "y" -o "$CONFIG_PCMCIA_RAYCS" = "y" -o\
$CONFIG_PCMCIA_NETWAVE" = "y" -o "$CONFIG_PCMCIA_WAVELAN" = "y" -o\
    $CONFIG_PCMCIA_XIRTULIP" = "y" };then
        define_boot CONFIG_PCMCIA_NETCARD y
    fi
```

中把 $CONFIG_PCMCIA_3C574" = "y" -o "$CONFIG_PCMCIA_3C574" = "y" -o\这行拿掉。然后在 drivers/net/pcmcia/中的 Makefile 文件中去掉这行

```
obj-$(CONFIG_PCMCIA_3C574) += 3c574_cs.o
```

就可以从目录 drivers/net/pcmcia/中把 3c574_cs.c 这个驱动程序文件从内核的源代码树中删掉。

㈢ 更复杂的是，有些设备驱动在链接过程去掉时，编译中会报告一些链接错误，这就需要手工修改或者注释部分内核源码。例如：在 arm 核下修改 linux-2.4.x/makefile 去掉链接时无用的设备驱动 drivers/net/net.o、drivers/media/media.o、net/network.o 等；就需要修改和网络数据结构 sock 相关的内核代码，具体涉及到下面的文件：

linux-2.4.x/init/main.c

linux-2.4.x/arch/arm/kemel/calls.s

linux-2.4.x/fs/fcntl.c

无疑，这是在裁去驱动部分代码中最难、最繁琐的。在具体的工作中我们更能体会到裁剪内核往往是动一毫而会牵一面，大到文件、函数，小到某一具体数据结构、成员，在具体修改中必须做好记录，并且不断总结内核编译失败的原因。驱动代码只是内核的外围部分，在内核各子系统的裁减中是相对较简单的，但它已经牵连到内核内部甚至是

底层的代码，因此，裁减的过程也是我们理解内核内部工作机制和其预留的应用接口的最佳时机。

3.4 裁剪有关体系结构的代码

我们开发的嵌入式系统都是基于某一种体系结构的，而内核的源代码树中，支持包括 alpha 、i386 等多种处理器结构，所以其它体系结构的代码完全可以从内核中删除。例如：如果我们的系统是面向 Strongarm SA-1110 处理器，只需要从源代码的根目录中的.config 文件中把下列代码删去：

```
#System Type
#CONFIG_ARCH_ARCA5K is not set
#CONFIG_ARCH_CLPS7500 is not set
#CONFIG_ARCH_C0285 is not set
#CONFIG_ARCH_EBSA110 is not set
#CONFIG_ARCH_FOOTBRIDGE is not set
#CONFIG_ARCH_INTEGRATOR is not set
#CONFIG_ARCH_RPC is not set
#CONFIG_ARCH_CLPS711X is not set
```

然后在/arch 目录中把相应的目录删除，再在/include 目录中把相应的包含文件目录删除即可。

另外还有很多细致的工作要做；由于 arm 系列处理器有很多的类型，我们可以去除掉如 ebsa110 等的支持代码，这些代码存在于 arch/arm/mach-ebsa110 这样的目录中，对于这些代码的删除也只需要在 config.in 和 Makefile 中删除相应的选项即可拿掉代码。

还有，在 Strongarm SA-1110 处理器这部分代码中，还有一些支持其他的类型的开发平台的，这部分的代码分在不同的程序文件之中，我们要从程序文件中直接删除。

例如，在 arch/arm/mach-sa1100/目录中又一个为 yopy.c 的文件时处理 GPIO 的边沿检查的程序，在这个程序中存在下列的代码 ：

```
static unsigned long yopy_egpio =
```

```
    GPIO_MASK(GPIO_CF_RESET) |
    GPIO_MASK(GPIO_CLKDIV_CLR1) | GPIO_MASK(GPIO_CLKDIV_CLR2) |
    GPIO_MASK(GPIO_SPEAKER_MUTE) | GPIO_MASK(GPIO_AUDIO_OPAMP_POWER);
static int __init yopy_hw_init(void)
{
    if (machine_is_yopy()) {
        YOPY_EGPIO = yopy_egpio;
        /* Enable Output */
        PPDR |= PPC_L_BIAS;
        PSDR &= ~PPC_L_BIAS;
        PPSR |= PPC_L_BIAS;
        YOPY_EGPIO = yopy_egpio;
    }
    return 0;
}
int yopy_gpio_test(unsigned int gpio)
{
    return ((yopy_egpio & (1 << gpio)) != 0);
}
void yopy_gpio_set(unsigned int gpio, int level)
{
    unsigned long flags, mask;
    mask = 1 << gpio;
    spin_lock_irqsave(&egpio_lock, flags);
    if (level)
        yopy_egpio |= mask;
    else
        yopy_egpio &= ~mask;
```

```
    YOPY_EGPIO = yopy_egpio;
    spin_unlock_irqrestore(&egpio_lock, flags);
}
EXPORT_SYMBOL(yopy_gpio_test);
EXPORT_SYMBOL(yopy_gpio_set);
```

这部分代码是针对 Samsung 的掌上电脑 yopy 来设置 GPIO 的读写掩码和读写控制的，这样的代码在程序中有很多，都可以从程序中拿掉。

另外在设备驱动程序中，有一些也包含了类似的其他体系结构的代码，也可以裁减掉。例如在 drivers/char/目录中的 sa1110 的触摸屏的驱动程序中，就包含有 freebird 的按键的驱动程序，也是需要我们裁去的。

内核其他部分的裁剪，与上述讲的裁剪过程有相似之处，如支持的多种文件系统部分就可以采用与裁去驱动的类似过程把多余的代码去除。而要去除内核中多余的工作机制等，就需要通过修改内核源代码来完成，这会在以后的章节中介绍。

综上所述，进行内核裁剪，就是力求将内核裁到最小，从某种程度上来说，即将内核中各模块的相关制约因素降到最小，从而在加载模块时将需要考虑修改的部分减到最小，无形中也加速了嵌入式软件的开发历程。但是，裁剪内核的难度主要在于完全了解内核的复杂的相关关系，每当改变内核的某一部分就不可避免地影响到其它部分的改变，如：进程调度子系统和其它内核子系统的相关关系，进程调度子系统在管理一个进程被调度时，需要内存管理子系统建立内存管理映像；而内存管理子系统又可以直接被进程调度子系统、文件系统、进程间通讯子系统以及网络子系统直接调用。可以看出这种关系是相当复杂的，就像一个错综复杂的交通网络，而利用本节所介绍的方法就可解决以上难题。同时，也可分离出 Linux 操作系统内核中必要部分和可裁剪部分，找到一个最小的内核：由线程管理加简单的寄存器管理和中断服务组成。它能够完成整个内核的初始化，由于这个系统内核没有外部设备驱动，所以只能封闭地运行，没有实用价值。我们要让它能做实际的工作，还需要在这个基本框架上进行内存管理和文件系统的定制，并最终附上设备驱动环节。

4 Linux 嵌入式系统下内存管理的优化

内存管理是操作系统中非常重要的子模块。如同普通操作系统一样。在嵌入式系统中，内存管理实现的好坏对系统性能有决定性的作用。Linux 内核的内存管理子模块性能优越，目前大多数使用的嵌入式 Linux 解决方案中都直接使用了标准 Linux 内存管理机制，或者对 Linux 内核的内存管理子模块作了很小的改动，例如去掉高端应用而特地引入复杂的内存管理机制，像基于 zone 区的内存分配机制等。对目前能够方便得到源代码的主流嵌入式 Linux 进行分析就可以发现，像 Hard HatLinux 使用的 Linux 内核中的内存管理子模块没有进行任何修改[23]，而像 ccLinux 这样的嵌入式 Linux 系统，干脆就直接使用了没有做过任何修改的标准 Linux 内核。从这两个例子可以看出，目前嵌入式 Linux 领域的裁减还未涉及到内存子模块。除了内核裁减以外，对硬件平台的支持也不构成嵌入式系统开发人员修改内存管理子模块的原因。因为目前 Linux 内核已经被移植到大量的非 x86 平台上，包括：ARM， M68K，PPC，Alpha，Sparc 等。这就为在这样体系结构的微处理器上运行 Linux 内核打下基础。

话虽如此，嵌入式 Linux 的内存管理仍然是值得关心的问题。目前，由于体积限制或者处于降低成本的考虑，嵌入式系统中所使用的微处理器大多缺少 MMU。这些硬件平台包括数字信号处理器（DSP），SOC(System On Chip)，以及那些不具有 MMU 的微处理器，例如 ARM7TDMI，Motorola 的 M68K 龙珠系列等。这些没有 MMU 的微处理器为传统 Linux 的应用造成了一个障碍，它们要求使用 flat 内存模型—— 即没有虚拟内存（分页/页面交换）、内存地址转换（分段）和内存保护的内存管理机制。

4.1 Linux 操作系统内存管理模型研究

4.1.1 Linux 内存管理模型

操作系统的内存管理模型可以分成如下 3 种：

1. 单一程序模型

这是没有硬件地址转换的内存管理模型。应用程序始终在物理内存的同一地址运行。一个时刻只有一个应用程序被加载运行，程序加载器把应用程序加载到内存低端，

将操作系统加载到高端。一个应用程序可以对所有的物理内存地址进行寻址。早期的操作系统受限于硬件能力多采用这种硬件模型，现代操作系统已经不见这种内存管理方式了。

2. 多程序模型

这也是没有硬件地址转换的内存管理模型。即使没有硬件地址转换功能，多个程序也可以共享相同的物理地址。在程序被加载到内存的时候，改变程序中的寻址指令（load，store，jump）所使用的地址值为当前被加载到的位置。

3. 具有地址转换硬件的内存管理模型

应用程序使用的是虚拟地址，CPU实际执行程序所使用的是物理地址，从虚拟地址到物理地址的转换需要操作系统和MMU硬件的参与。

在以上三种内存管理模型模型中，标准Linux操作系统使用了上述第三种模型，即使用了具有地址转换硬件的内存管理模型。

Linux内核作为操作系统的核心，必须能够克服物理内存的局限，使用户进程在透明方式下，拥有比实际物理内存大得多的内存。其策略之一就是使用虚拟内存。Linux成功地实现了以虚拟内存为核心的内存管理策略，强大的分页机制，公平的交换方式，各类有效的高速缓存，以及以页保护为主的保护措施等。内存管理的目的是要尽可能地方便用户，同时Linux系统通过对用户进程虚存的有效管理，作到了虚存对一般用户和Linux程序员的透明。 内存管理是十分复杂的工作。如果把所有的工作都集中在一个模块中完成，必然会使内存管理模块变得十分复杂。因此Linux将内存管理的工作划分开，实现几个不同的内存管理器。这些内存管理器各司其职、互相配合，共同完成对内存的管理工作。Linux共实现了5个内存管理器，它们是：

1 物理内存管理器 物理内存管理器负责物理内存的分配、释放与回收，它以页为单位实施管理，目的是提高性能，减少碎块。

2 内核内存管理器 由于内核经常需要小内存用于建立各种管理机构，而物理内存管理器过于粗放，所以Linux提供了内核内存管理器，专门负责内核中小内存的分配和回收，Linux的内核内存管理器采用slab算法。

3 虚拟内存管理器 虚拟内存管理器在物理内存管理器的基础上，通过页目录、页表及交换机制，为系统中每个进程模拟了一个大小为4GB的虚拟地址空间。

4 内核虚拟内存管理器 如果内核需要较大的内存，物理内存管理器和内核内存管理器都不能很好的工作，原因是它们只能分配有限大小的、物理上连续的内存。为了满足内核对大内存的需求，Linux利用虚拟内存管理的思想，在内核虚拟地址空间（3GB以上），实现了内核虚拟内存管理器。

5 用户空间内存管理器 该管理器负责进程用户态虚拟内存的动态分配和回收（如C的malloc, free）它管理的内存在进程的堆中。用户空间内存管理器一般在库中（libc）实现，不属于内核，但内核对其提供有相应的支持。

毫无疑问，物理内存管理器是基础，虚拟内存管理器是核心，而各个内存管理器有都离不开内核内存管理器的支持。

虚拟内存是一种对RAM和磁盘进行无缝混合访问技术。所有的虚拟内存对于应用程序来说好像真的存在一样，应用程序在加载的时候并不需要关心是否会超过系统中实际的物理RAM的大小[24]，内核主要通过页目录和页表的地址转换功能将应用程序的虚拟地址转换成物理地址，这个过程中可能将应用程序中使用的超过了实际物理内存大小的虚拟地址映射到适当地物理地址，从而使应用程序可以随心所欲地使用巨大的虚拟存储空间。

但是只通过地址映射还不能解决有限的物理地址内存被虚拟地址空间所使用的问题，操作系统还必须使用页面交换机制将那些暂时不再使用的内存空间交换到外存中以使其他程序能够使用物理内存。Linux没有将整个进程所使用的空间交换到外存中，而是对部分不再使用的大小为4KB的页面进行交换，这样就获得了更多的灵活性。

应用程序在标准Linux中的加载使用了“按需调页”的策略，也就是说，应用程序在开始被加载的时候并没有一次被全部装载到内存中，只有那些现在必须使用的代码和数据在开始进行了加载。这可以从引导Linux 核心自身和应用程序在Linux上获得执行的唯一方式，即进程载入— exec 系统调用看出。

Linux 是用 comand-loading 算法，所以 do_execve() 实际所做的工作并不多，没有把所要执行的程序真正的读入内存，而仅仅把所要的代码头设置好，并把进程的状态设置为“就绪”，等待系统的调度，以运行此进程。真正的把代码读入内存，要涉及到较复杂的算法，如在首先要内存中寻找一块足够大的空间，然后再把程序从外存读入此空间中。

为此，exec()所做的和内存有关的工作有以下这些：

(1) 内存空间的获取：

申请1页用于可执行头文件。

申请 1 页以上用于数据堆栈。 (MAX_ARG_PAGES)

(2) 将页表结构清零clear_page_tables()，除去旧的页表项。

(3) 更改局部描述符表change_ldt() 。

(4) 设置局部描述符代码段ldt[1] = code base=0x00, limit=TASK_SIZE

(5) 设置局部描述符数据段ldt[2] = data base=0x00, limit=TASK_SIZE

设段特权级为3, P=1, S=1, G=1. type=1 (code) or 2 (data)

(6) 设值指令寄存器值eip = ex.a_entry

(7) 设置新创建的中断调用点堆栈指针，而这些值在调用恢复时被弹出。

(8) 更新存储空间界限值包括：用户代码空间、用户数据空间、堆空间。

end_code = ex.a_text

end_data = end_code + ex.a_data

brk = end_data + ex.a_bss

Interrupts and traps 是在当前任务的环境中被处理的。特别地，地址的转换是用当前任务的页目录进行的。然而，段却是用的内核的。这样，所有的线性地址都指向 kernel memory。在程序执行的过程中，如果遇到了不在内存中的程序部分将产生页面错误，操作系统在处理这个错误中断的时候回到外存中找到相应的应用程序部分进行加载。这种设计是基于计算机科学中著名的90-10规则的：90%的程序执行时间花费在整个程序10%的代码上。所以只要保持我们用到的程序在内存中，就既可以满足程序的执行速度又节约物理内存空间。

可见，为了解决无穷大问题和快速访问问题，linux 引入了虚拟存储系统、缓存和交换等；这些算法设计也非常精巧，各个内存管理器的划分使得操作更灵活。在桌面应用领域满足了许多应用的需求，更加体现了Linux操作系统的完整性、可靠性和优越性。在嵌入式应用领域里，从因特网设备到专用控制系统，Linux操作系统前景都很光明，但Linux虚拟内存管理机制对于一个嵌入式系统来讲有时往往是多余的，它不仅限制了Linux在传统嵌入式领域的拓展，更重要的是影响了嵌入式系统的一个重要特性 —— 实时性，这也使原来的嵌入式内存解决方案受到很大冲击。

4.1.2 Linux VMM 机制用于嵌入式实时系统的问题

一个实时系统经常要对多个周期性事务进行响应，依照每一事件进行处理所需要的时间不同，实时系统有可能不能对它们全部进行处理，例如，存在m个周期事件，每一个事件的周期对应为P_i，所需要的CPU时间对应为C_i，当且仅当满足条件式：

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1 \tag{4.1}$$

时，一个实时系统才能处理全部的事务，满足这一条件的实时系统被称为可调度的（Schedulable）实时系统[25,26]。从这个公式中，用于系统管理的周期事件越多，系统的可调度外部事件越少；事务的周期P_i越小，C_i越大，可被实时调度的进程就越少。

根据上面的定理，我们可以分析出虚拟内存管理机制对嵌入式系统实时性存在很大影响，原因如下：

（1）页面调度是程序执行延时的主要原因：等待队列中的进程其页面有可能被换出，当所需要的资源获得的时候，就会产生页面中断，从交换区换入相关页面，如果这时候内存页面不足，还要淘汰内存中别的页面再换出来，才可以最终执行。这是一个复杂的过程，更何况内存同外部存储器交换数据非常费时间[27]。这个过程给实时进程带来了不可预知的时延，严重的影响了系统响应的可确定性，这在 实时系统中是不允许的。同时CPU在各种进程和环境中切换过多，无疑增加了进程执行的CPU时间C。可调度性降低。

（2）页面调度的kwapd是典型的实时周期事件，而且周期P短，每10ms调度一次，成为系统中最为频繁的系统调用之一，每一次调度都要察看内存页面情况，做出反应，极大的影响了别的进程响应。这种短周期的进程越多，实时系统的可调度性就越差。

（3）其它的系统调用一般都和内存相关，而这些系统调用是系统中最基本的工作并被频繁的使用，同样会因为虚拟内存的管理机制而浪费时间。

所以桌面Linux版本不能用于实时性要求强的嵌入式环境。而且同时运行的进程越多，每个等待运行的进程所分配的CPU时间片就越受到限制。而CPU又不得不响应这些系统调用，CPU的控制权会在各种进程上频繁切换。操作系统的相关理论证明CPU控制权的切换越快，资源的利用率就越低。对于一个实时的任务而言，CPU具有的响应越快就越能够在需求的时间里面完成任务。

所以，剔除了虚拟内存机制可以减少很多相关的系统调用、减少很多的系统调用时间、减少频繁的缺页中断从而使得CPU响应速度加快。对于具体进程而言，去除了由于页面调度带来的不确定性。

另一方面，嵌入式系统往往没有硬盘，只有体积很小的电子盘，交换也失去了其意义，因而虚拟内存机制也是多余的。

基于以上这些问题，也促使我们无法绕过这个难题，也就是说，除了根据需要配置Linux内核，我们还必须针对嵌入式应用的特点对Linux内存管理实现进行修改，从更低层次使Linux适应嵌入式系统应用。桌面Linux系统设计的目标是在内外存丰富的条件下运行多任务，并能满足各种应用对运行性能的要求。但是绝大多数嵌入式系统应用的运行情况单一，桌面Linux系统中为了提高运行性能采取的措施在嵌入式系统应用中可能会失去作用，甚至成为负担。我们希望借鉴上述的第二种内存管理模型：多程序模型，真正从源代码级上对内存管理机制做一个彻底地改造，即考虑将虚存机制和其相关代码从内核中剔除。下面的内容先对虚拟内存管理机制相关的函数和数据结构进行细致系统分析。

4.2 Linux 虚存机制在嵌入式系统背景下的分析

4.2.1 虚存机制在硬件平台上的实现

Linux 支持很多硬件运行平台，常用的有：Intel X86，Alpha，Sparc 等。对于不能够通用的一些功能，Linux 必须依据硬件平台的特点来具体实现。工作在保护模式时，进程使用的 48bits 逻辑地址（Logical address）。逻辑地址的高 16bits 为段选择符，低 32bits 是段内的偏移量。通过段选择符在 GDT 或 LDT 中索引相应的段描述符（得到该段的基地址），再加上偏移量得到逻辑地址对应的线性地址（Linear Address）。如果没有采用分页管理，线性地址是直接映射物理地址（Physical Address），于是可以直接用线性地址访问内存；否则，还要通过 X86 的分页转换，将线性地址转换为物理地址。Linux 在 X86 上采用最低限度的分段机制，其目的是为了避开复杂的分段机制，提高 Linux 在其他不支持分段机制的硬件平台的可移植性，同时又充分利用 X86 的分段机制来隔离用户代码和内核代码。因此，在 Linux 上，逻辑地址和线性地址具有相同的值。

VM 机制是现代操作系统的共性，标准 Linux 系统采用 3 层页式存储管理策略和“请求调页”的调页策略，为了实现跨平台运行，Linux 提供了一系列转换宏使得核心可以访问特定进程的页表。这样核心无需知道页表入口的结构以及它们的排列方式。这种策略相当成功，无论在具有三级页表结构的 Alpha、AXP 还是两级页表的 Intel X86 处理器中，Linux 总是使用相同的页表操纵代码。

Linux 把虚存和物理内存都划分为大小相同、尺寸固定的块/页面，通过地址转换机制将虚存地址转换为物理内存地址。在 x86 处理器的 Linux 中，页面尺寸为 4KB。Linux 虚存管理的页表结构为 3 级，依次为页目录(PGD，Pagedirectory)、中间页目录(PMA，Page Middle Directory)和页表(PTE page Table)，因此，线性地址在转换为物理地址的过程中，线性地址就被解释为四个部分（不是 X86 所认识的三个部分），增加了页中间目录中的索引[28]。当运行在 X86 平台上时，Linux 通过将中间页目录最大的页目录项个数定义为 1，并提供一组相关的宏（这些宏将中间页目录用页目录来替换）将三级页面结构分解过程完美的转换为 X86 使用的二级页面分解。这样，无需改动内核中页面解释的主要代码（这些代码都是认为线性地址由四个部分组成）。Linux 采用“请求调页”策略，当进程运行时，发现需要的页面不在内存中，便产生缺页中断，然后把需要的页面从外存调入内存，如果物理内存分配完，则淘汰最近最久不用的页。Linux 通过 kswaPd 守护进程每 10ms 检查系统空闲页面，如空闲页面太少，则按某种途径释放内存，其中一种就是使用 LRU 算法选择进程将其部分页面换出(swap out)到交换区，在页面需要又将该进程页面从交换区换入到内存(swaPin)。进程在等待状态下都可能被换出内存而被写入交换区。进程状态与交换区关系如图 4.1 所示。

同时，内核态虚拟空间从 3GB 到 3GB+4MB 的一段[29]（对应进程页目录第 768 项指引的页表），被映射到物理地址 0x0～0x3FFFFF（4MB）。因此，进程处于内核态时，只要通过访问 3GB 到 3GB+4MB 就可访问物理内存的低 4MB 空间。所有进程从 3GB 到 4GB 的线性空间都是一样的，由同样的页目录项，同样的页表，映射到相同的物理内存段。Linux 以这种方式让内核态进程共享代码和数据。

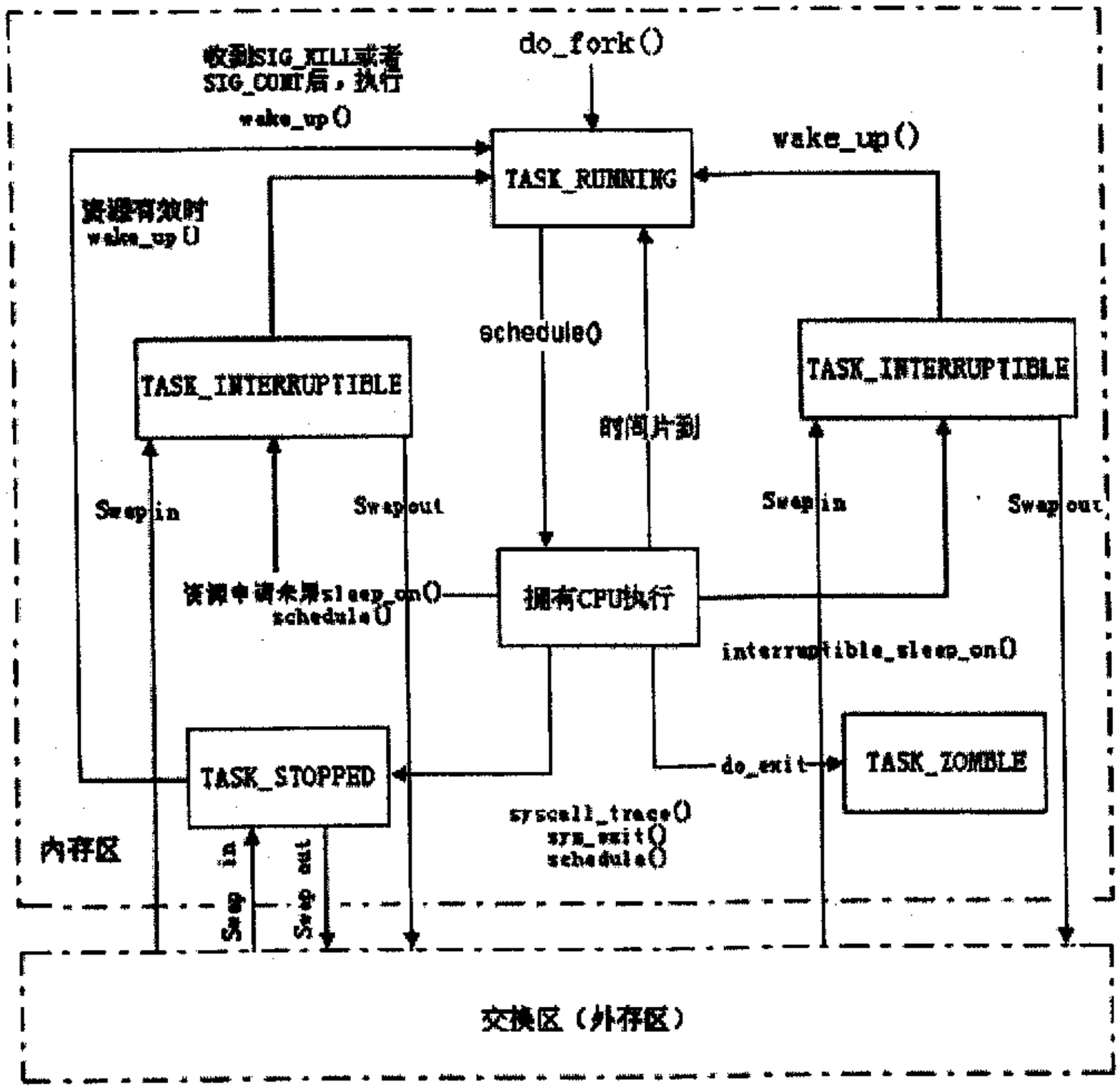

图 4.1　Linux 进程状态与交换区的关系

4.2.2　Linux 源码对虚拟内存的实现构架

Linux 虚存管理主要由 4 个模块组成，其源码大部分在/mm 目录下。

• 内存映射模块(mmap)

把磁盘文件的逻辑地址映射到虚拟地址，以及把虚拟地址映射到物理地址。源程序分别是:①mmap.c 文件中主要函数 do_map 的功能是把文件中的逻辑地址映射成虚存线性地址;②Rnremap.c 文件中主要函数 sys_ mremap 的功能是扩张或缩小现存的虚拟内存空间;③Filemap，c 文件中主要函数功能是处理内存映射和处理页高速缓存器，即把线性地址映射到内存且修改页高速缓存。

• 交换模块(swap)

控制内存内容换入和换出，使得在物理内存的页(RAM)中保留有效的逻辑页。源程

序分别是:①page_io.c主要函数的功能是读写交换文件;②Swap_state.c主要函数的功能是修改交换高速缓存 swap cache;③Swap_file.c 主要函数的功能是完成换入换出系统调用(sys_swapin, sys_swapon);④Swap.c 主要函数的功能是定义交换使用的数据结构和常量;⑤Kswapd 是一个周期性处理内外存交换的守护进程。

• 核心内存管理模块(core)

负责核心内存管理功能，即对页的管理，这此功能将被别的内核子系统使用。源程序分别是:①page_allocc 的主要函数功能是处理页释放，回收和分配:②Memory.c 请页的相关函数;有关用于请页处理的函数代码。

• 特定结构的模块

给各种硬件平台提供通用接口,这个模块通过执行命令来改变硬件 MMU(内存管理单元)的虚拟地址映射，并在发生页错误时，提供公用的方法来通知别的内存子系统。这个模块是实现虚存管理的基础。该模块实现的源程序分别是:①Arch/386/mm/fault.c 处理页异常;②Arch/386/mm/init.c 内存初始化的有关函数。

以上这些模块可以用图 4.2 表示:

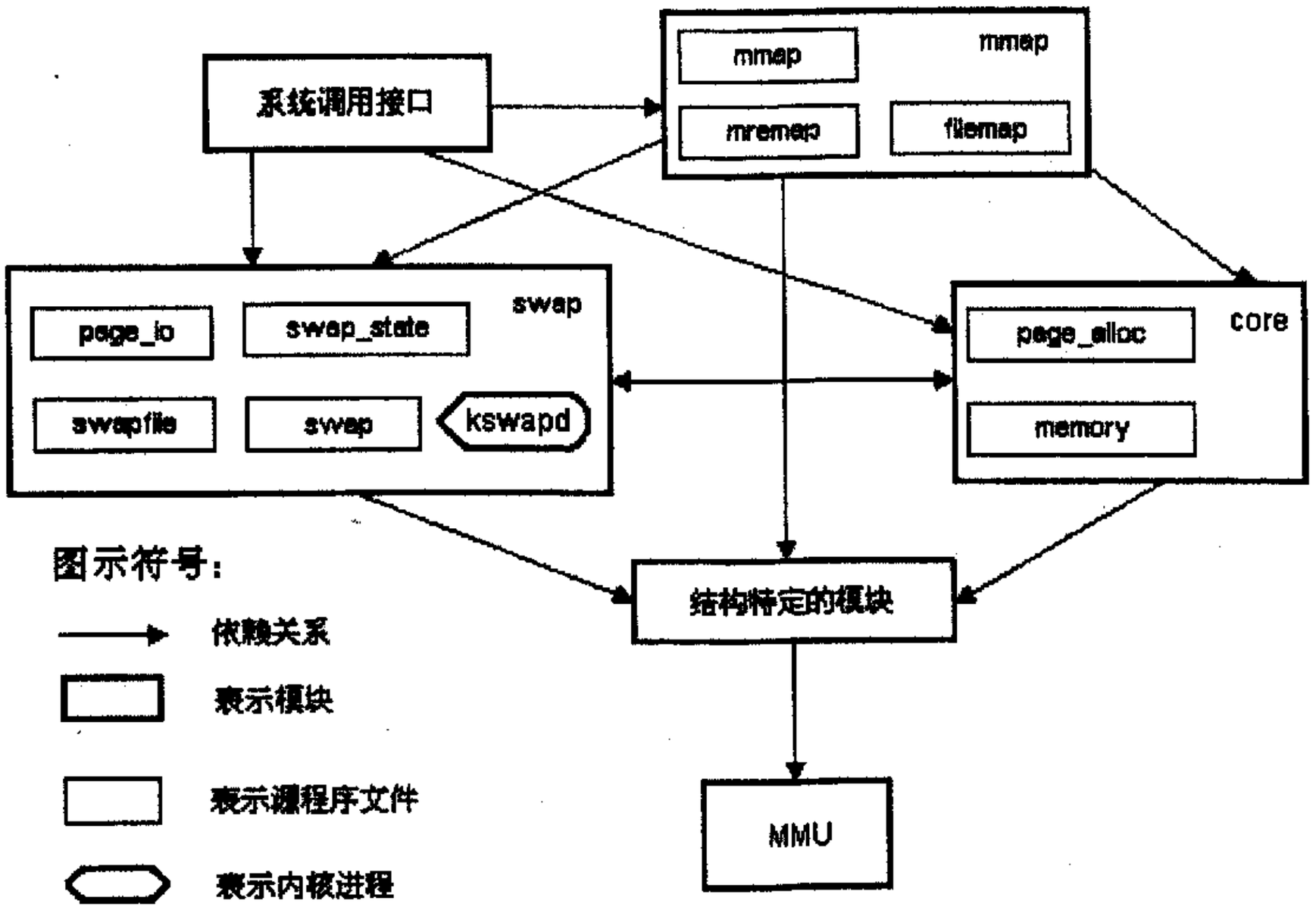

图 4.2 Linux 内存管理模块关系图

4.2.3 虚拟内存管理的主要数据结构

在整个 linux 内核中用于虚存管理的数据结构是最复杂的。虚存空间的管理不像物理空间的管理那样有一个总的物理页面仓库，而是以进程为基础的，每个进程都有各自的虚存空间。每个进程的虚拟内存用一个 mm_struct 来表示，在内核代码中，用于这个数据结构（指针）的变量名通常是 mm。这是比下面要提及的 vm_area_struct 结构更高层次上使用的数据结构，mm_struct 数据结构是整个进程用户空间的抽象，也是总的控制结构[30]。它描述了一个进程的页目录，有关进程的上下文信息，以及数据、代码、堆栈的起始、结束地址，还有虚拟存储取的数目以及调度存储用的链表指针。此数据结构具体定义如下(见 include/Linux/mm.h)：

```
stuct mm struct{
    int count;/*使用该 mm 结构的个数，如果是多处理机，count>1*/
    pgd_t *pgd;/*进程页目录的起始地址*/
    unsigned long context;
    unsigned Iong start_code, end_code, start_data, end_data;
    /*进程代码段、数据段起始、结束地址*/
unsigned long start_brk, start_stack, start _mmap;
unsigned long arg_start, arg_end, env_start, env_end;
    /*调用参数区、进程环境区的起始、结束地址*/
unsigned long rss, total_vm, locked_vm;/*rss 进程驻留
    物理内存页面数;totalvm 占用虚拟页面数*/
unsigned long def_flags;
 struct vm_area_struct*mmap ;  /*mmap 以双向链表组
 成的指向 vma 模块的首指针*/
 struct vm_area_struct*mmap_ avl: /*以 avl 树结构组
 成的虚拟空间的首指针*/
 struct semaphore mmap_sem; /*对 mmap 操作的互斥
 信号量*/
```

```
};
```

进程的虚拟是由 vma 块组成的，Linux 中关于存储管理的绝大部分功能，都是利用 vm area struct 结构实现的。每个 vm_area_struct 数据结构描叙了虚拟内存的起始与结束位置，进程对此内存区域的存取权限以及一组内存操作函数等。数据结构如下(同见 include/Linux/mm.h)：

```
struct vm area struct{
        struct mm_struct * vm_mm;/*VM 地址参数*/
        unsigned long vm_start, vm_end;/*该 vma 块代表的使
         用虚拟空间的起始和结束地址*/
        pgprot_t vm_page_prot;    /*保护位*/
        unsigned long vm_flags;   /*标志位*/
        /*AVL tree of VM areas Pertask, sortedbyaddress*/
        short vm_avl_height;/*avl 树高度*/
        struct vm_area_struct  *vm_avl_left, *vm_avl_right;/*
         vma 块的左、右子节点*/
        struct vm_area_struct  * vm_next:/*按地址排序的下
         一个指针*/
       struct vm_area_struct  *vm_next_share, * vm_prev_share;/*
        只用在共享内存和索引节点的情况下*/
        struct vm_operations_struct * vm_ops; /*对 vma 块操作
        的函数集*/
        unsigned long vm_offset:
        struct inode*vm inode:/*该 vma 块对应的文件*/
        unsigned long vm_pte:/*shared mem*/
        };
```

可执行映象映射到进程虚拟地址时将产生一组相应的 vm_area_struct 数据结构。每个 vm_area_struct 数据结构表示可执行映象的一部分：可执行代码、初始化数据(变量)、未初始化数据等等。Linux 支持许多标准的虚拟内存操作函数，创建 vm_area_struct

数据结构时有一组相应的虚拟内存操作函数与之对应。当进程启动时，Linux 要为它分配一个 task_struct 结构，每个进程（根据需要）建立新页目录（mm 成员 pgd_t *pgd）当内核调度一个进程进入运行时，就将这个指针转换成物理地址，并写入控制寄存器 CR3。当一个进程运行时,他的代码段和数据段将都会被调入内存。如果它使用了共享库，共享库的内容也将被调入内存。系统首先分配一个 vm_area_struct 给进程，并将这个进程连结到虚拟内存的链表中去，这时根据进程的可执行映像中的信息把数据段和可执行代码分配内存，新分配的内存必须和进程已有的内存连结起来才能应用。这样就会出现页故障，系统利用了请页机制来避免对物理内存的过分使用，但进程访问的虚存不在当前的物理内存时,这时系统会将需要的页调入内存，同时修改进程的页表，用来标志虚拟页是否在物理内存中。

因此,系统用了较复杂的数据结构来跟踪进程的虚拟地址，在 task_struct 中包含一个指向 mm_struct 结构的指针，进程的 mm_struct 中则包含了进程可执行映像的页目录指针 pgd.还包含了指向 vm_area_struct 的几个指针,每个 vm_area_struct 包含一个进程的虚拟地址区域。一个进程有多个 vm_area_stuct 结构，linux 要经常对进程分配或调整 vm_area_struct ，这样对 vm_area_stuct 的查找效率.对系统很有影响.所以在这里将所有的 vm_area_struct 形成了一个查找效率较高的平衡二叉树结构。

从上面数据结构可以看出，一个进程都有占用的 vma 页面是由 mmap 和 mmap_avl 指针来链接的，整体显示如图 4.3 所示，采用 avl 树的优点显而易见就是可以快速找到相关地址所在的 vma 块。同时，同一个任务的 vma 块按前后顺序排成一个线性链表。Linux 为维护这些虚拟页面提供了很多操作，如查询、插入、删除等。

从图中可见,mm_struct 结构及其下属的各个 vm_area_struct 结构只是表明了对虚存的需求。一个虚拟地址有相应的虚存区间存在[31]，并不保证该地址所在的页面已经映射到某一物理页面，更不保证该页面就再内存中。当一个未经映射的页面受到访问时，就会产生一个异常，那时候，Page Fault 异常的服务程序就会来处理这个问题。所以从这个意义上，mm_struct 结构、vm_area_struct 说明了对页面的需求；相对应的物理页面 page 结构则说明了对页面的供应；而代表 MMU 记忆单元的页目录、中间页目录及页面表则是二者中间的桥梁,这些都是我们考虑要除去的。

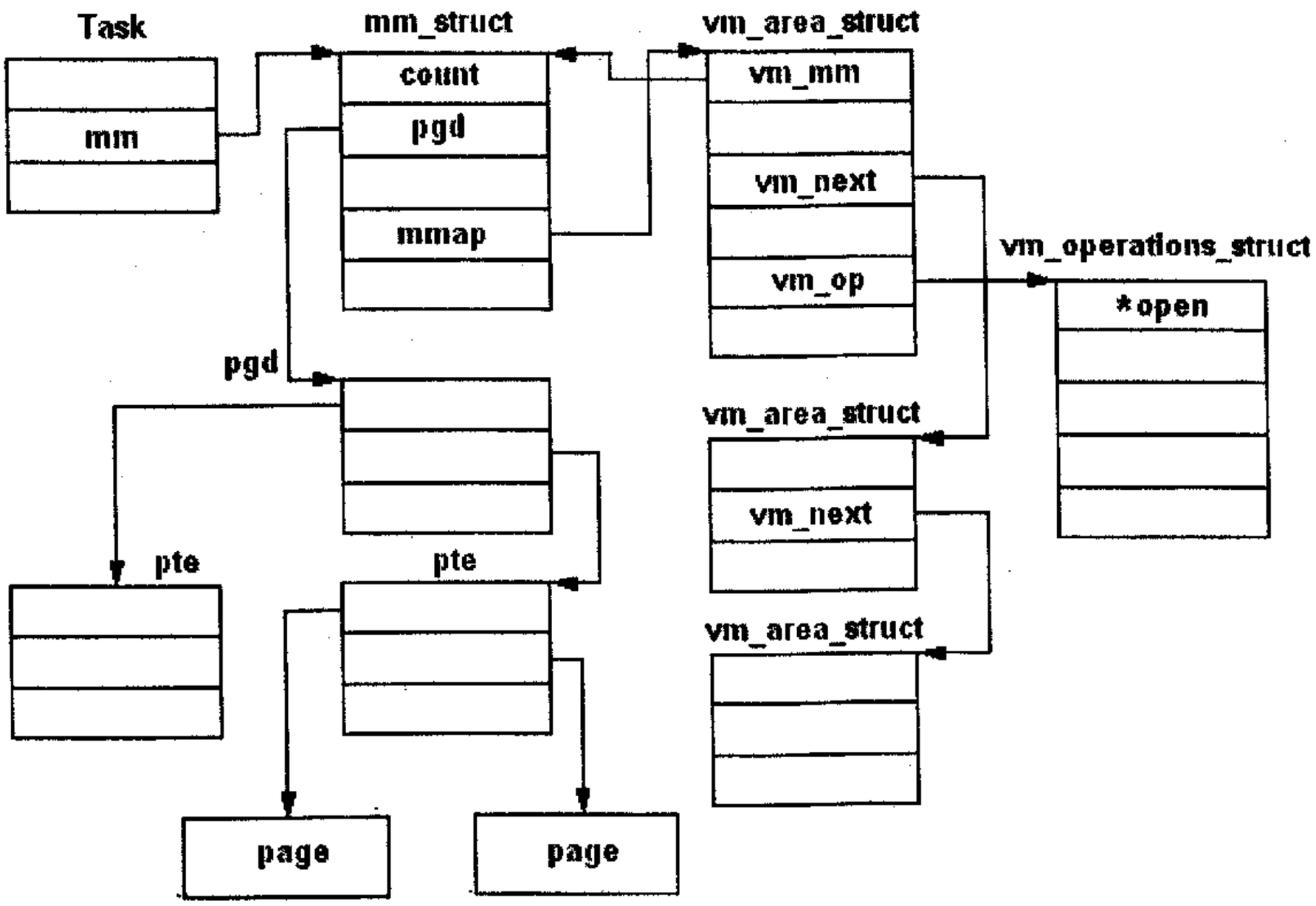

图 4.3　进程的虚存管理数据结构

4.3 修改源码，剔除 Linux 虚存管理机制

4.3.1 修改源码及具体的改进部分

Linux 用 vm_area_struct 及由它构成的链表来表示一个进程真正占用的虚拟空间。它维护 3 种关于 vm_area_struct 的链表，即由 mmap、vm_ext 表示 vma 段单向链表，由 mmap_avl、vm_avl_left、vm_avl_right 表示的 AVL 树，以及由 attaches、vm_next_share、vm_prev_share 表示的共享内存链表。前 2 种每个进程都有一条，第 3 种每个共享进程拥有一条。剔除虚存机制就意味着不再需要一些链表和对这些链表的相关操作。在该改进的时候先要确定自己的数据结构，与内存管理关键的数据结构是 mm_struct，剔除了有关虚拟的存储页表后，要建立新的数据结构以管理好真实的物理页面。

关于 mm_struct 结构的修改如下：

```
struct mm_rblock_struct{
```

```
  int size;/*一次所分配的单位物理块的大小*/
  int refcount;/*访问该块的进程数*/
  Void*kblock;/*数据指针*/
  };
struct mm_tblock struct{/*物理块的链表*/
  struct mm_rblock struct *rblock;
  struct mm_tblock struct *next;
  };
stuct mm struct{/*没有改动的部分省略*/
  unsigned long def_flags;
  struct wait_queue*vforkwait;/*在等待队列*/
  struct inode*executable:
  struct mm tblock struct tblock;/*进程所占用的物理块的链表*/
  };
```

改动后的数据结构维护的是真实内存链，这样，进程运行中就不会出现缺页中断，因为所有页面都已经装入了内存。在需要内存的时候，注意使用 kmalloc 分配真正的物理内存，如：

```
tblock=(structmm_tblock_struct*)kmalloc(slzeof(struct
        mm_tblock_struct ), GFP_KERNEL);
tblock->rblock=(structmm_rblock_struct*)kmalloc(sizeof(struct
        mm_rblock_struct), GFP_KERNEL);
```

也就是进程可以向核心申请使用物理内存。这些内存块来自于 free_area 数组，由 blocksize 表、size 表、page_descripor 结构和 block_header 结构共同管理。而过去的 vmalloc（） 和 vfree（）由于是从虚拟空间 3GB 以上的虚拟空间的最高端分配内存，所以现在对它们的实现只是简单的调用 kmalloc（）和 kfree（），实际上分配的也是从空闲物理页面链表中获得的页面。

对于 struct vm area struct 结构修改如下：

```
struct vm_area_struct{
```

```
    unsigned long vm_start;
    unsigned long vm_end:
    unsigned long vm_flags;
    unsigned long vm_offset;
};/*vm area struct 结构保留它只为了传递数据，实际上已不是一个真正的虚
    拟地址结构*/
```

操作系统的虚存机制是 Linux 内存管理的核心机制，和文件系统进程调度、进程通讯都有关系，修改时要了解内存管理程序的结构和各子系统之间的依赖关系。由于去掉了管理虚存的数据结构，该数据结构相关的代码都要改动。那些对虚存结构操作的函数，即 struct vm_operatings_struct 结构中包含的 open、close、unmap、swapout、swapin 都不再需要。无虚存管理也就不需要支持交换操作，故 swap.h 中定义的用于交换的结构和函数也需要屏蔽。/mm 目录是内存管理的内核目录，在该目录下的文件中体现虚存机制的部分都要做相关改动，最明显的，对内存管理部分的改进，主要制止实时进程的页面交换。根据上面的虚拟内存模块组成可知，页面交换主要在交换模块中实现，这也是改动最大的，有的函数干脆从内核中全部裁去，这在后面的图形中可以看出。还有关于共享虚拟内存的部分和进程调度的部分等都要改 如在 ipc/shm.c 回收共享内存函数 shm_swap()、kernel/sched.c 等。

其次，根据 Linux 采用"按需调页"的原则来创建一个进程时页面分配的整个过程，依次裁去：进程控制块 mm 成员；内存态堆栈；以及页目录、页表等结构。

最后，还要修改 Linux 系统主函数 main.c，剔除虚拟内存机制，就没有缺页中断和请求调页，也就没有交换的任务，故用于这一任务的守护进程 kswapd 是多余的，也就不需要它的设置函数 kswapd_setup。具体做法是屏蔽声明和调用。同时，在标准 Linux 的启动过程中要进行许多与内存管理相关的功能模块的初始化工作，诸如页目录和页表映射的建立等。在把它改造成不使用 MMU 的系统的过程中必然要对这些功能进行修改。arch/*/kernel/entry.S 是一个汇编文件，它包含了一个 kernel_ entry 的定义，这是整个内核的进入点。在完成某些平台相关的初始化工作之后，执行流程跳转到 start_kernel(void)处，在其中与内存模块相关的函数调用流程如下：

```
    setup_arch(&command_line);
```

```
kmem_cache_init();
paging_init();
free_area_init()
mem_init();
kmem_cache_sizes_init();
pgtable_cache_init();
```

则关闭掉不再使用的初始化部分并对留下函数的各自功能进行改造，特别是setup_arch()存储了可用物理内存起始地址，记录了内核中代码的起始位置、数据起始位置、系统中存储器的分布情况等存储器信息；这些都要做具体的改动，以为 ld 目标文件提供正确相应的定位信息。另外，也包括进程管理的初始化代码如proc_caches_init() 函数创建进程管理子系统用到的几个 cache，其中的vm_area_struct、mm_struct 结构也必须作出修改。

总之涉及的文件包括 schedh、fs.h、mm.h、Proc_fs、binfmts.h、swap.h、memory.c、swap.c、mmap.c、vmalloc.c、mprotect.c、filemap.c、mlock.c、page_io.c、vmscan.c、remap.c、page_alloc.c、swapfile.c、swap_state.c、mem.c、fbmem,.c、buffer.c、inode.c、open.c、exec.c、super.c、nameic、binfmt_script.c、locks.c，sched.c、exitc、fork.c、sysctl.c、shm.c 以及 main.c 等文件。

这些文件是内核程序，改动中遇到了很多困难。修改代码最好用#i fdef 和#ifndef 将原代码和新改动代码分开，并将改动过的文件做好记录。创建新的 Nommu 目录，把改动后的原内存管理目录下的文件加入其中，而其他目录下的文件改动利用条件编译的形式编译，修改 config.in 文件，以创建新的无 MMU 的内存管理模块，用于今后的定制。

4.3.2 改进后的效果分析

裁去虚拟内存系统后，虽然不再使用标准内核中的分页管理机制，没有了页表和页目录对线性地址的映射及按需调页机制，但我们还是将整个物理内存划分成固定大小的页面。由数据结构 page 管理，有多少页面就有多少 page 结构，他们有作为元素组成一个数组 mem_map[]。物理页面可以作为进程的代码、数据、堆栈的一部分，还可以存储装入的文件，也可以当作缓冲区；而且仍用标准 Linux 的变型 Buddy Sytem 机制来管理

空闲的物理页面，这就是 bitmap 表和 free_area 数组，以及相关的函数/宏 _get_free_pages()、free_pages()、_get_free_page()。 修改后的系统多任务管理更加简单，因为没有进程的页表和保护机制要处理，内核的调度器不需要修改，唯一需要完成的工作就是进行程序上下文的正确保护和恢复。

这样，简化了内存管理体系，如图 4.4 所示，所有和交换文件相关的依赖关系不再存在；进一步地由于进程所有的虚拟空间都不能被置换到辅存，kswapd 守护进程也完全不再需要了。这也就意味着 kmem_cache_reap()、shrink_mmap()、shm_swap()、swap_out()、shrink_dcache_memory()这些内核函数与有关数据结构都会从内核中去除，则简化了整个页面管理模块，减小了内核尺寸。

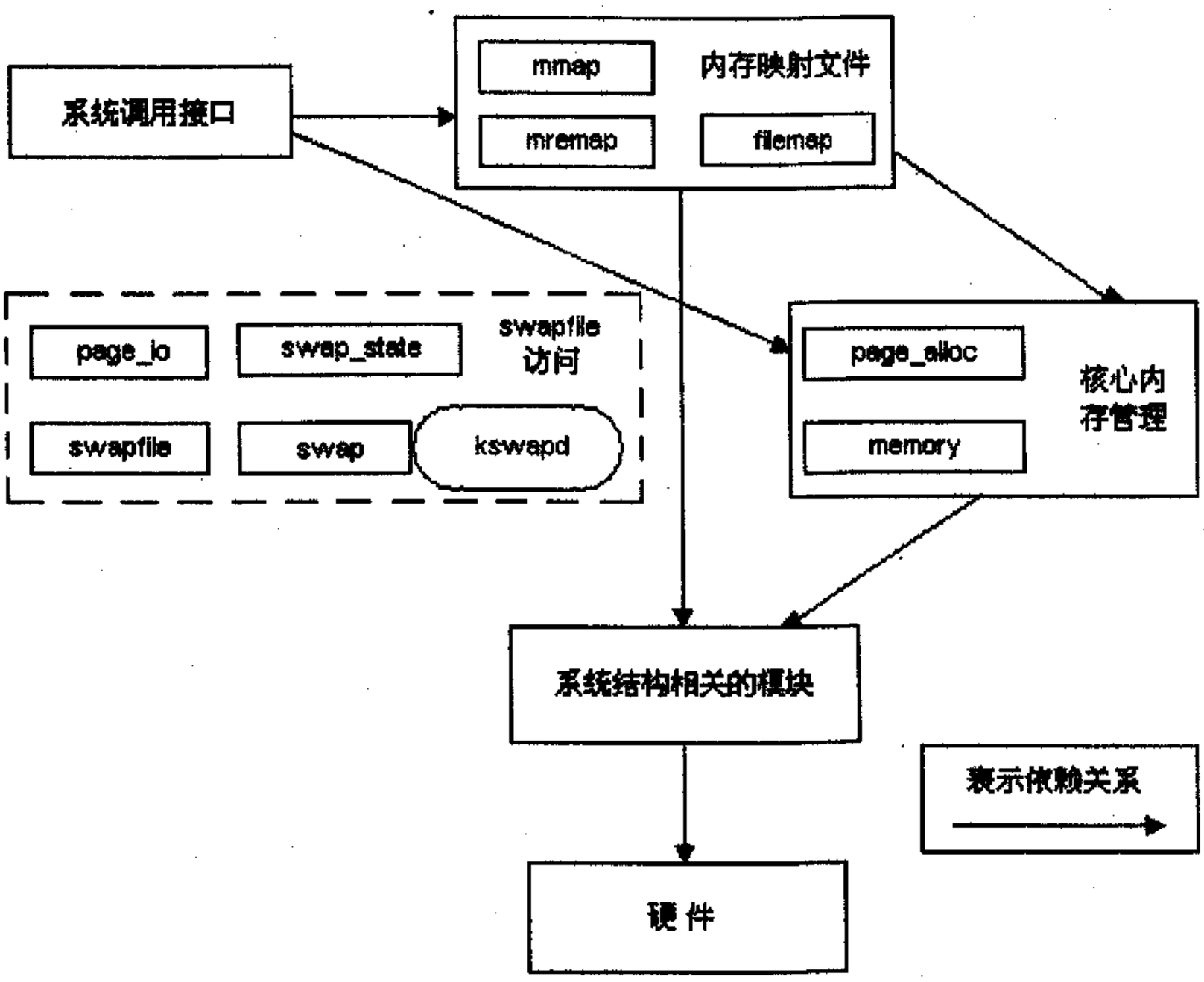

图 4.4 无 VM 管理机制的内存管理程序结构

去掉虚拟内存机制后各个进程状态和交换区域的关系如图 4.5 所示。 图 4.1 交换区在图 4.5 已不存在，被加粗的 swapin 和 swapout 线都被去掉，进程状态管理的代码进一步得到简化。同时，去除了大量的系统调用，如 brk()的完全取代，系统调用 swapon

和 swapoff 等；还有一些功能简化了的系统调用，包括不再实现共享页面的 do_mmap()、精简的进程载入调用 sys_exece、进程链接拆除调用 sys_exit、进程创建调用用 fork 即用 vfork()代替等。以上这些都将大大减少内核代码的体积，提高内存管理效率。

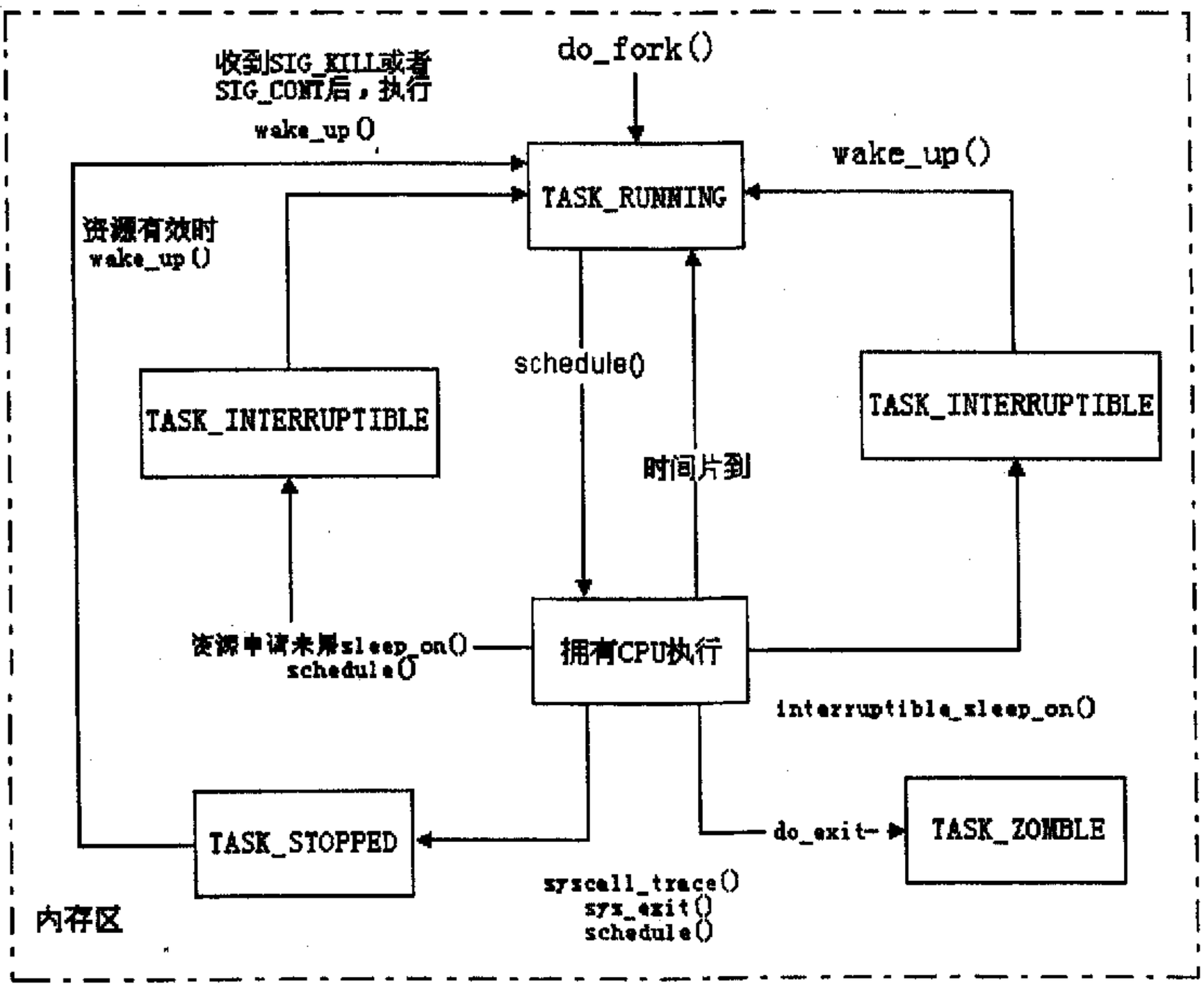

图 4.5　无 VM 管理机制的 Linux 进程状态图

同样就内存和文件系统等关系而言，也得到了简化。Linux 为了充分利用处理器资源，在内存管理，虚拟文件系统，网络支持等方面使用了许多消耗大量内存资源的缓存机制和预分配机制，而这些利用存储空间来换取处理速度的部分也将被删除，从而实现了在对系统性能不产生大影响的情况下，大幅减小系统运行的内存开销与 CPU 开销，提高了任务的时间反应能力。

修改后的操作系统还不是硬实时，仍属软实时，但系统性能还是显著提高。在我们所研究的一个 Linux 嵌入式系统项目中，按上述方法，去掉 VM 机制的操作系统与原系统对比，稳定和实时性明显好转。如要使系统的实时性和稳定性更好，须从系统的调度和中断处理方式入手；相应地，C 库、编译器都需要为嵌入式系统提供标准相同的 API，

以及系统的启动代码包括 setup.S、head.S 也要进行修改。这些都是该项目后期需完成的工作，也是项艰巨的工作。

总之，衡量内存管理算法好坏的主要标准是效率和利用率。但是两者往往矛盾：内存利用率高的算法其效率不一定高；反之，效率高的算法其内存利用率有可能很低。Linux 采用一个折衷的办法：由物理内存管理器以页为单位使用伙伴算法，高效地分配与回收物理内存，并可避免内存碎块的产生；由内核内存管理器在物理内存管理器的基础上向用户、内核提供各种大小不等的物理内存，从而提高物理内存的利用率；也就是说，Linux 内核内存管理分为了两个层次：基于页面的内存管理与基于 cache 的内存管理，基于页面的内存管理是内核内存管理的基础，它为系统其他部分提供了页面分配和释放界面，即所谓的 GFP 函数；而基于 cache 的内存管理则为其他部分提供了方便与统一的存储功能界面，并且有很高的性能[32]。这里采用现代 UNIX 操作系统常用的内存管理技术，即块分配器（slab allocator）。我们可利用调用图的方式，将快分配器代码进一步裁去，实现完全基于页的内存管理。

5 文件系统的选择与定制

在现代计算机系统中，文件系统是操作系统的重要组成部分之一。对于用户而言，文件系统也是操作系统中最直接可见的部分。它负责管理外存上的文件，并为操作系统和用户提供文件的存取、共享和保护等功能。文件系统设计的好坏对系统安全和性能有着非常大的影响。

Linux 是一个迅速发展的、性能卓越的、具有巨大发展潜力的操作系统。它的重要特征之一就是虚拟文件系统抽象机制。Linux 文件系统的总体结构可分为三个部分：

第一部分是Virtual File System Switch(VFS) ,这是Linux 文件系统对外的接口,任何要使用文件系统的程序都必须经由这层接口；

第二部分是Cache , Linux 通过这一层缓冲机制来加快速度；

第三部分就是实际的文件系统,像Ext2 ,VFAT 等等。

为避免混淆,通常将 VFS 及缓冲机制统称为 VFS(Virtual File System) 。VFS 是用户程序空间与实际文件系统之间的一层接口软件,它将不同的文件系统的具体实现统一起来,并向上提供一层统一的系统调用接口[33]。VFS 只存在于内存空间,它在系统启动时建立,在系统关闭时消亡。Linux 就是以这样的结构得以支持多达几十种文件系统。

linux 的文件系统是一个结构清晰的文件系统。在用户进程对文件系统提出操作请求后，虚拟文件系统将内存的数据结构与具体的文件系统的数据结构关联起来，同时将调用具体的文件系统的操作函数，启动设备的输入/输出操作，实现设备上文件的读取、写回、查找、更改、更新等操作。当然，由虚拟文件系统提供的内存节点缓冲区、内存目录项缓冲区、数据块缓冲区提供了内存中操作节点、目录、数据块的手段，实现文件系统尽量在内存中处理文件，减少读取外设的操作次数。在操作完成之后，文件系统在适当的时机将调用虚拟文件系统的更新例程，将改变的数据从内存中全部写回外部设备。

5.1 虚拟文件系统的接口

虚拟文件系统必须管理任何时间安装的所有不同的文件系统。为此它管理整个文件

系统（虚拟）和各个真实的、安装的文件系统的数据结构。VFS 的主要实现思路：在内核中提供一个文件系统框架，包括接口函数集、管理用的数据结构、以及各种缓存机制。

虚拟文件系统的所有数据结构都是在系统运行后才建立的，在系统卸载后删除。它不是一个真正的文件系统，所以在磁盘上没有虚拟文件系统的数据结构。要想使系统工作，必须具有像 ext2、minix 这样的逻辑文件系统并且让虚拟文件系统和逻辑文件系统之间建立逻辑的连接。VFS 为各个不同的逻辑文件与内核通信提供了一致的接口。

如果用户开发自己的文件系统，只要符合这个标准接口就可以了。实际上，所有的文件系统不管是自己开发的还是 linux 本身所支持的文件系统都是通过 VFS 与 linux 内核交互信息的。 linux 中各文件系统的关系如图 5.1：

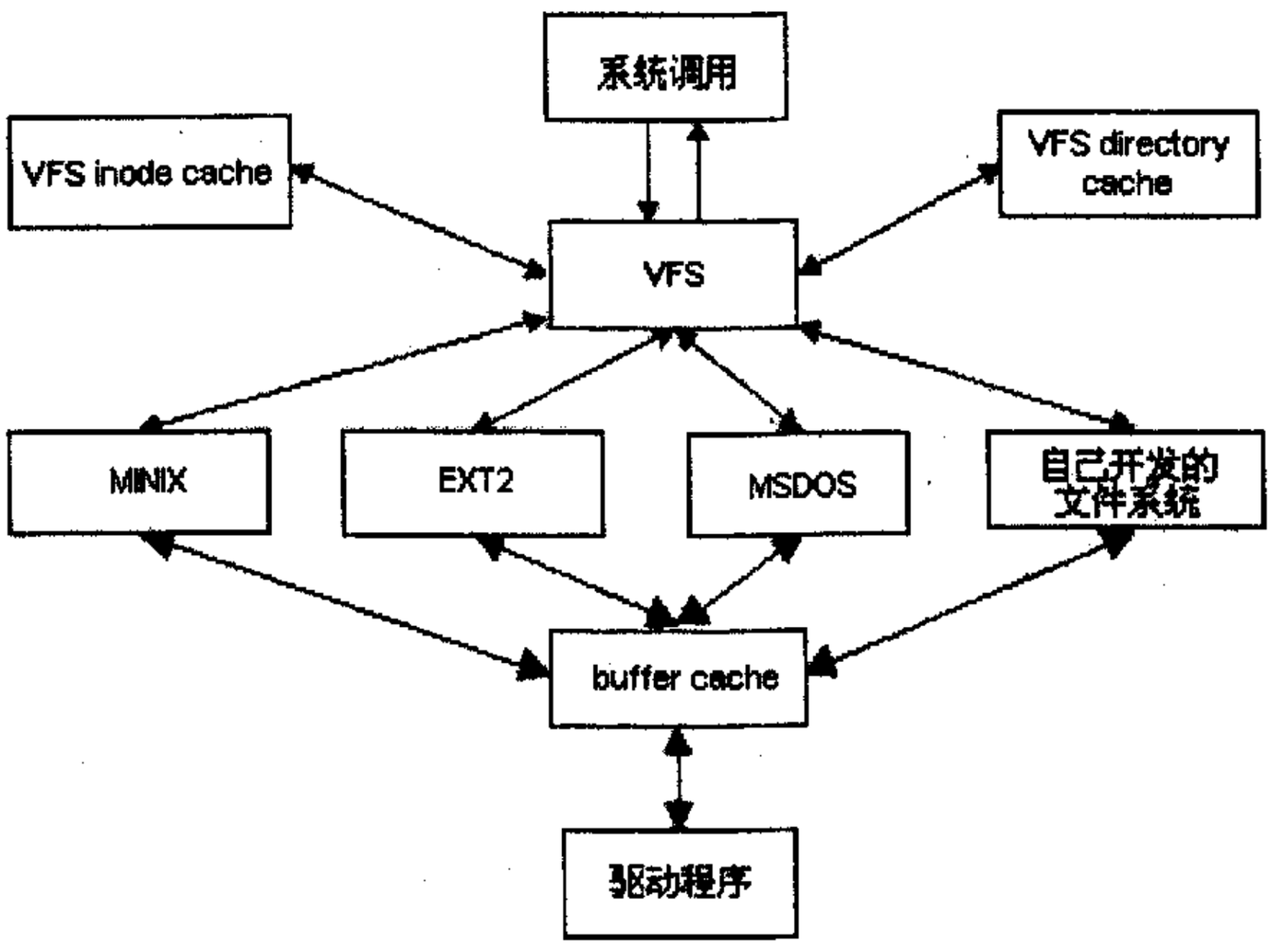

图 5.1　Linux 中文件系统的关系

所有的 Linux 文件系统使用一个共同的 buffer cache 来缓存底层设备的数据缓冲区，这样可以加速对于存放文件系统的物理设备的访问，从而加快对文件系统的访问。这个 buffer cache 独立于文件系统，集成在 Linux 内核分配、读和写数据缓冲区的机制中。

让 Linux 文件系统独立于底层的介质和支撑的设备驱动程序有特殊的好处。所有的块结构的设备向 linux 内核登记，并表现为一个统一的，以块为基础的，通常是异步的接口。甚至相对复杂的块设备比如 SCSI 设备也是这样。当真实的文件系统从底层的物

理磁盘读取数据的，引起块设备驱动程序从它们控制的设备上读取物理块。在这个块设备接口中集成了 buffer cache。当文件系统读取块的时候，它们被存放到了所有的文件系统和 Linux 内核共享的全局的 buffer cache 中。其中的 buffer 用他们的块编号和被读取设备的一个唯一的标识符来标记。所以，如果相同的数据经常需要，他会从 buffer cache 中读取，而不是从磁盘读取（会花费更多时间）。

操作系统引入 VFS 对所有文件系统进行了抽象，使得不同的文件系统在 Linux 内核和系统中对其它进程来说都是相同的。它有如下的功能：

记录可用的文件系统的类型；

将设备同对应的文件系统联系起来；

处理面向文件的通用的操作；

涉及到针对文件系统的操作时，VFS 将他们映射到与控制文件、目录以及 INODE 相关的逻辑文件系统中。

当进程发布一个面向文件系统的调用的时候，内核调用 VFS 中相应的函数，这个函数处理一些与物理结构无关的操作，并且把他重定向为真实文件系统中相应的函数调用，这些函数调用则用来处理那些与物理结构相关的操作。

可见，VFS 提供上下两方面的接口。VFS 的上层接口是提供给 I/O 系统的用户使用的，这些用户，包括应用程序和内核的其它管理模块，通过该接口使用 I/O 系统（文件、设备、网络等），如打开、关闭、读、写等。VFS 的下层接口是提供给真实文件系统的，VFS 支持的每个真实文件系统都要提供对这个接口的实现。换句话说，只要实现了这组接口，VFS 就认为它是一个真实的文件系统。VFS 本身利用它的下层接口实现它的上层接口，下层接口由真实文件系统实现。

为了实现上下层接口之间的转换，VFS 必须维护许多数据结构，如记录真实文件系统信息的 super_block 结构，记录文件或目录信息的 inode 结构 、描述每个打开文件或设备的 file 结构、描述文件组织关系的 denntry 结构、描述文件系统类型的 file_system_type 结构记录已安装文件系统信息的 vfsmount 结构、记录设备驱动程序的 chrdevs blkdevs 数组等。这些数据结构中的信息来源于真实文件系统或设备驱动程序。

在系统初始化时，VFS 支持的每个真实文件系统和外部设备都要向它注册，每个真

实文件系统在使用之前还要安装。在使用文件、目录、设备之前必须获得它的 VFS inode，、而后再将其打开。在注册、安装、获得、打开的过程中，VFS 建立起了上下层接口之间的映射关系。从而可以实现准确的操作转换。

5.2 VFS 实现机制

5.2.1 文件系统的初始化

在操作系统启动并进行初始化的过程中，系统调用 filesystem_setup（）例程进行文件系统初始化，它根据系统的配置参数调用各实际文件系统的初始化例程———init_fsname fs（），此过程向 VFS 进行注册 register_filesystem。

VFS 在内存中维护一个文件系统类型列表，全局指针变量为 file_systems，新注册的文件系统将一个描述该文件系统的数据结构 file_systemtype 传入 VFS，将它插入链表 file_systems 的结尾。文件系统的初始化执行过程如图 5.2 所示。

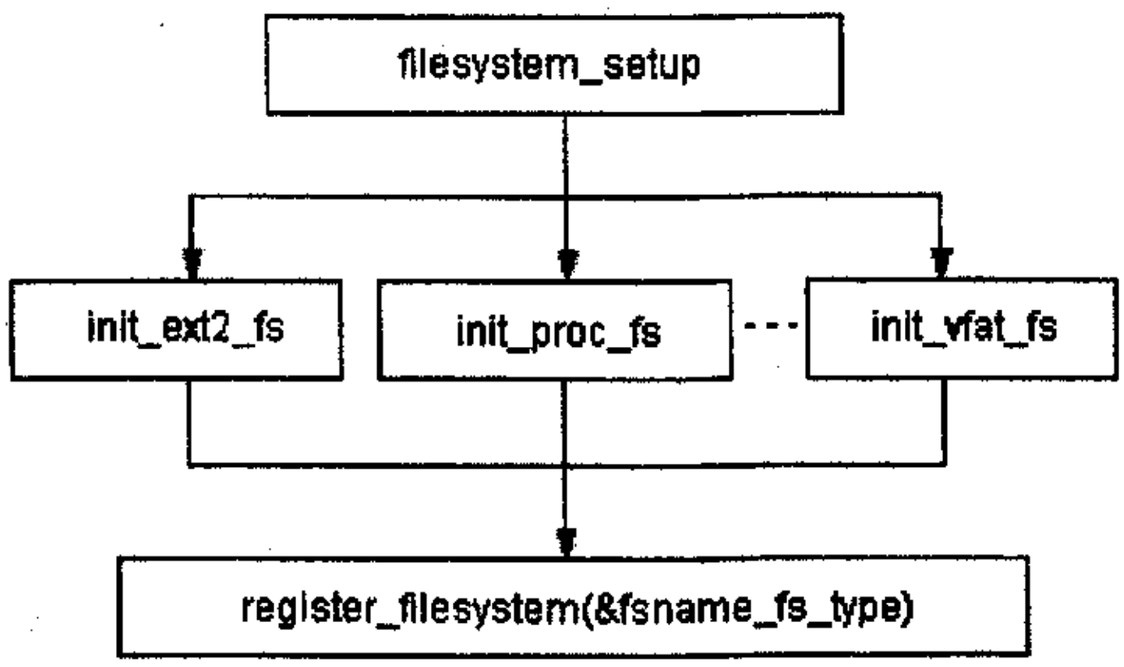

图 5.2 文件系统的初始化执行过程

这样，VFS 就可以遍历 file_systems 链表来决定内核支持哪些文件系统类型。

5.2.2 文件系统的装载

实际的文件系统是内建到系统内核中，也可作为可载入模块在需要时进行装载。系统在内存中维护一个已装载的文件系统列表，全局指针变量为 vfsmntlist 。另一个指针 vfsmnttail 指向表中最后一项，mru_vfsmnt 指针指向最近使用的文件系统[34]。装载例

程执行如下几个步骤:

- 搜索文件系统类型列表,若未找到则返回错误码。

```
fstype = get_fs_type ((char * )type) ;
if(!fstype) ret_ENODEV ;
```

- 检查物理设备是否存在且还没有安装,是则找到装载点的目录 i 节点,检查是否是目录类型且尚未装载其它的文件系统。
- 在 super_blocks 中搜索空闲的 super_block 结构,若没有搜索到,则分配一个超级块结构,将它初始化,然后加入到 super_blocks 向量中。
- 运行所装载文件系统的读超级块例程,这个读例程从物理设备读取信息,填充 VFS 超级块结构的各个域。fstype_> read_super(s ,data ,silent)
- 增加新的装载点。分配新的装载点结构,进行初始化设置,然后加入到 vfsmntlist 表中,并修改 vfsmnttail 指针。

5.2.3 VFS 的内部实现

Linux 文件系统的数据结构类似 EXT2 文件系统,即采用超级块与 inode 节点数据结构进行管理。超级块结构驻留在内存空间,存放了该文件系统的重要信息。从超级块中可以取到该文件系统中任何一个文件的 inode 节点,从 inode 节点则可对该文件进行读写等操作,这就实现了对磁盘中任一文件的控制。

5.2.3.1 超级块

super_block 结构存放本文件系统的重要信息,包括一些基本信息;一组 super block 结构的操作函数;一组与 quota 管理有关的函数;管理 super block 所属文件系统 inode 的信息;一些处理 superblock 的同步字段及各个文件系统本身所特有的信息。

1. super block 的同步机制管理

```
unsigned char s_lock ;
struct wait_queue *s_wait ;
```

上面这两个字段是用来处理 super block 的同步。s_lock 记录着目前 super block

是否被锁住,如果是,其值为 1 ;若不是则为 0 。s_wait 是一个 wait queue 的结构,被放到 queue 里的进程将会进入 sleep 的状态,直到被叫醒为止。并且,如果要改变 super block 的内容,需要先调用 lock_super () 锁住 super block ,以免产生竞争条件。操作完之后则要调用 unlock_super ( ) 将 lock 释放掉。

2. 文件系统本身特定信息的管理

super_block 结构是所有文件系统所共同使用的一个结构,但是,除了共同的部分之外,不同的文件系统之间也有相当的差异性,因此,在 super- block 结构有一个字段是专门用来存放各个文件系统所独有的信息。这些信息是在注册文件系统时调用 read_super () 所填入的。

```
union {
.
struct minix_sb_info minix_sb ;
struct ext2_sb_info ext2_sb ;
.
} u ;
```

因为每个 super- block 在同一时间内最多只会记录一个文件系统的资料,所以,这个字段是 union。例如 EXT2 文件系统 ext2_sb 就是专门存放 ext2 文件系统本身所额外需要的信息,由 ext2_read_super ( ) 函数填入的。

3. 超级块操作指针

```
struct super_operations {
.
void ( * put_super) (struct super_block 3 ) ;
void ( * write_super) (struct super_block 3 ) ;
int ( * statfs) (struct super_block 3 ,struct statfs 3 ,int) ;
.
} ;
```

Linux 在注册文件系统时,将操作指针添入,当进行超级块操作时,通过操作指针指向实际文件系统的具体执行函数,实现 VFS 向下一级文件系统的映射。

以系统调用 statfs 为例,该调用的执行过程如图 5.3 所示:

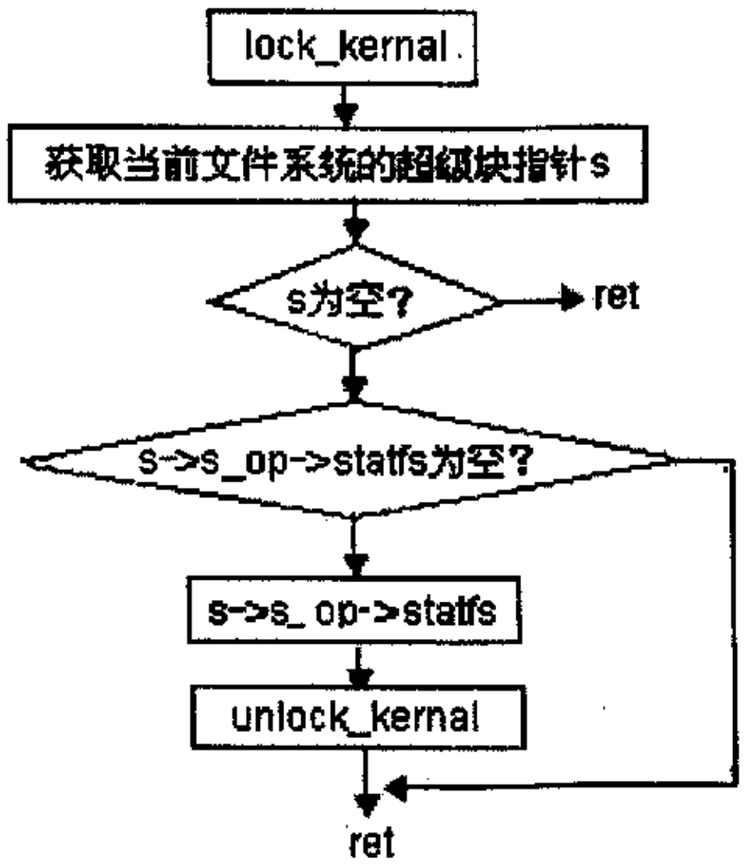

图 5.3 执行过程图

5.2.3.2 inode 节点

inode 节点唯一地表示文件系统中某一个目录或文件,是 Linux 管理文件系统的最基本单位。inode 结构包括 inode 节点链表管理字段、inode 基本信息、inode 同步管理字段、与内存管理有关的字段、Quota 管理字段、跟 file lock 有关的字段以及一组用来操作 inode 的函数。

inode 结构前三个字段是用来将 inode 链接起来的字段,分别是:

```
struct list_head i_hash ;
struct list_head i_list ;
struct list_head i_dentry ;
```

在 VFS 里,用几个链表来管理 inode。inode_unused 用来将目前未使用的 inode 链接在一起[35]。inode_in_use 用来将目前正在使用的 inode 链接在一起,当一个 inode 被使用时,它会从 inode_unused 中被取出来,接着它会利用 i_list 字段放到 inode_in_use 中。sb_> s -dirty 用来将标志为’脏’的 inode 链接在一起。这个链表的表头位于 super block 的 s_dirty 字段,使用 i_list

链接。另外,所有正在使用中的 inode 都可以经由 inode_in_use 链表找到,但是,因为系统的node 太多,该链可能会很长,查找的速度较慢,因此,Linux 构造了 inode 节

点哈希表 inode_hashtable，每个使用中的 inode 都会计算出其 hash value，并且放到 hash table 中。

5.2.3.3 文件结构

```
struct file {
struct file* f_next , ** f_pprev ;
struct dentry*f_dentry ;
struct file_operations *f_op ;
loff_t f_pos ;
.
} ;
```

i. file 结构也是用链式管理，f_next 指向下一个 file 结构，f_pprev 指向上一个 file 结构地址的地址，f -dentry 记录其 inode 的 dentry 地址，f_pos 记录目前文件的偏移量，每次读写都从 offset 记录的位置开始。

ii. 与超级块和 inode 节点相同，f_op 字段也提供一组操作文件的函数。这些函数提供了最终对文件的控制功能。

- llseek(file ,offset ,where)
- read(file ,buf ,buflen ,poffset)
- write (file ,buf ,buflen ,poffset)
- readdir (file ,dirent ,filldir)
- open(inode ,file)

5.2.3.4 文件描述符管理

Linux 通过几个间接的指针从文件描述符指向 inode 节点结构，例如，当一个进程执行 open 系统调用时，内核返回一个非负整数值，这个整数就是文件结构指针数组的索引值，而每一个文件结构指针通过 file-> f_dentry 指向 dentry 结构，dentry 结构又通过 dentry-> d_inode 指向 inode 节点结构。

5.3 文件系统的选择与改进

我们知道，在标准的 linux 系统中，对块设备的存取使用的是逻辑文件系统，一个逻辑文件系统被装配在虚拟文件系统的一个装配点上。Linux 支持几十种逻辑文件系统，但是在嵌入式系统中，这些功能不是必须的，这些是可以裁减的。

同时，因为嵌入式系统的应用范围非常广泛，所以不可能有一种文件系统在嵌入式 Linux 系统中一统天下，适用于大到嵌入式服务器、工控系统，小到嵌入式手表、PDA、机顶盒等的所有情况。真正有经验的开发者必须根据系统应用环境、目标、配置等信息来选择构建合适的文件系统。例如需要存储大量数据的场合，还是需要在系统中配置普通硬盘，搭建一个大容量、高吞吐率的文件系统。而在以 Flash 为主要存储设备的便携型嵌入式系统中，为了尽量节省空间，延长 Flash 使用寿命，则需要选择 cramfs 等文件系统。要想成功地开发完成一个嵌入式 Linux 系统应用，就必须在平时积累关于 Linux 各种文件系统的知识，熟悉其性能和操作。

过去，除了 ext2 等通用目的文件系统以外，没有专门针对嵌入式环境的文件系统，因此使用 Linux 作为嵌入式设备的操作系统时，有些情况下就需要自己修改文件系统代码，来满足特定目的的需要。随着 cramfs，JFFS2 等文件系统的成熟，在嵌入式环境下，也有了很多好用的 Linux 文件系统可供选用，降低了开发难度，减少了开发时间。

上面已经论述了在 linux 中，文件系统是如何挂上系统并且和系统进行通信的。Linux 既然通过 VFS 来和各种不同文件系统发生联系，不难考虑只要将 super_block 数据结构中的联合中的成员使用自己文件系统的数据结构去替代就可以了。同样，要把 VFS Inode 的联合使用自己文件系统的数据结构去替代。然后编写自己的系统调用，就可以了。

在建立自己的文件系统的时候，最好仿照 EXT2 文件系统的方法，组织自己的逻辑文件，组织自己的数据结构。EXT2 文件系统在组织文件上的优秀性是有目共睹的。比如支持长文件名、高扩展性等。但是对于一个嵌入式系统，这些功能和实现这些功能的数据结构反而显得多余，如 MINIX 文件系统，要求文件名不超过 14 个字符，这就能够满足嵌入式环境的需要了[36]。所以改进的较好的方法是在 EXT2 文件系统中去掉对嵌入式系统来说多余的、繁琐的信息，是切实可行的做法，也是一种节省工作量的做法。

然而修改一个文件系统，不是很容易的事情，因为涉及到很多方面的事情，所有关系到打开文件列表，进程创建、运行和结束等一系列的过程和文件系统相关联，相应的代码都需要修改。内存方面，凡是与交换文件有关的代码也需要改动。当然，也可以采用 MINIX 文件系统。

5.4 文件系统的简化

5.4.1 Ext2 文件系统缓冲区的预读取方法

Ext2 文件系统利用了缓冲区管理实现了文件的预读取方法。在一个数据块被读取时，内核将读取 I/O 设备的该数据块的相邻的多个数据块。这种方法很有效，因为 Ext2 文件系统采用了数据块分配的优化算法。该算法的原则是数据块局部性的原则，使用数据块通常是相邻或相近的。而数据块读取通常是多个数据块。故预读取的数据块经常是也要被读取的数据，这样减少了 I/O 操作。这种方法多用在对文件的连续读取和目录的读取上。

该方法被文件系统广泛地使用，实现文件系统缓冲区预读取方法的函数原型有：

linux/fs/ext2/namei.c

int ext2_lookup(struct inode *dir, struct dentry * t dentry)

linux/fs/ext2/dir.c

static int ext2_readdir(struct file *filp, void * dirent, filldir_t filldir)

linux/fs/ext2/filemap.c

ssize_t generic_file_read(struct file *filp, char *buf, ssize_t count, loff_t *ppos)

5.4.2 Ext2 文件系统精简实例

在本例中我们使用了两个 Ext2 文件系统分区，一个存放 Linux 系统，作为 root 分区，另一个主要存放应用程序和数据。Ext2 文件系统映像存放在 Flash 中。系统运行时，这两个文件系统映像被复制到内存中，并装载到 Linux 的虚拟文件系统上。所以这两个文件系统完全运行在内存内。

根据上面的分析，我们对 Ext2 文件系统进行了简化。主要针对 Ext2 文件系统的数据块预分配机制。Ext2 文件系统中的文件占用若干数据块。文件扩展时，首先向最后一个数据块的空闲部分写入数据。如果不够，则申请空闲数据块。Ext2 数据块预分配机制是在从文件系统分区分配数据块时，文件系统为该文件一次预分配若干连续的数据块；当文件被再次扩展，并要申请新数据块时，文件系统首先查看该文件是否有预分配的数据块。如果有，则直接使用其中第一个预分配数据块；如果没有，才真正从文件系统分区获取空闲数据块。为了避免文件系统碎片，提高访问效率，预分配的数据块尽量接近最后一个数据块。

预分配机制的主要目的是减少由于文件扩展造成的文件系统碎片，并且提高分配数据块的效率。使用预分配机制的必要性是：在低速外部存储设备上访问若干非连续数据块的速度比访问连续数据块慢很多。这在硬盘之类需要机械运动的存储设备上非常明显。使用预分配机制的可行性是：外部存储设备的存储资源非常丰富，不会因为预分配使得文件系统空闲数据块不足。

显然，对于我们使用的 Strong ARM 平台嵌入式 Linux 而言，情况正好相反：在内存中访问连续数据块与非访问连续数据块的速度差别不大；而 Ext2 文件系统分区却很小，由于整个 Ext2 文件系统都运行在内存中，碎片不会对文件访问效率造成影响；而预分配数据块却会提前耗尽系统空闲数据块。例如，用于存放用户应用数据的分区通常只有 128 kB～512kB，视用户应用程序和用户数据大小而定。而默认情况下， Ext2 文件系统的数据块大小是 1 kB，预分配机制每次预分配 8 个数据块，这样平均每个文件多占用 4kB。因为系统中大部分文件仅数十 kB，所以这部分开销很可观。数据块预分配机制对于我们使用的 Strong ARM 平台的 Linux 环境是不适用的。我们在 Ext2 文件系统实现中去除了数据块预分配机制。这样,；当文件扩展并要申请新数据块时，文件系统不再查找是否有预分配的数据块，而是直接在该文件最后一个数据块相邻区域内查找并分配且只分配一个空闲数据块。

实验表明，我们使用的 Strong ARM 平台上使用去除了数据块预分配机制的 Ext2 文件系统，文件操作的效率没有受到明显影响。以受预分配机制影响最大的文件写操作为例，我们采用每次写 1kB 的方式连续扩展文件，对采用和不采用预分配机制的 Ext2 文件系统进行比较，以执行文件操作的时间长度作为考察目标。

jiffies 是 Linux 内核的毫秒级计数器，我们将扩展写入操作前后记录的 jiffies 的差值作为文件写操作的时间开销，写 16 kB 以下的文件时，文件系统用时小于或接近 1 毫秒，计时存在较大相对误差。写 64kB 以上文件用时基本呈线性增长。但是无论哪一种大小，使用预分配机制的 Ext2 文件系统和不采用预分配机制的 Ext2 文件系统进行扩展写入操作的时间开销基本一致，效率没有明显差别。

Linux 是一个功能强大的操作系统，并且拥有大量的应用程序，将 Linux 移植到嵌入式系统，能够充分利用这些成熟的技术，提高开发效率和开发质量。为了使 Linux 成为真正的嵌入式操作系统，除了开发驱动程序，让 Linux 运行在嵌入式硬件平台上，还需要针对嵌入式系统应用的具体需求对 Linux 进行裁减和优化，为此，一方面我们利用 Linux 模块化的特点配置 Linux 内核，在我们使用的 Strong ARM 平台已经建立起一个功能类似桌面 Linux 的嵌入式 Linux 环境，而其体积和运行开销很小，能够满足手持移动设备的应用需求。另一方面，根据应用的特点对 Linux 实现进行优化。实验表明，我们对 Ext2 文件系统预分配机制修改适用于我们的基于 Strong ARM 平台嵌入式 Linux 环境，在对性能没有明显影响的情况下节省系统开销。在这个方面我们还有许多工作要做，例如为了进一步提高文件系统效率，需要对缓冲缓存机制进行优化，以及使用更加紧凑的文件系统。此外，进程管理、内存管理等模块也需要针对嵌入式应用的特点加以优化。而对于数据管理简单的应用，可以把文件系统以设备驱动的形式实现。下面将 Linux 内核经过裁剪、修改移植到 DSP 平台就使用了这一方式；而且通过这一实例也是对我们裁剪定制内核代码可行性的综合验证。

5.5 移植 Linux 内核至 DSP 平台方案研究

DSP技术已经广泛地应用于通信、控制、测试 / 测量、电子娱乐等领域。伴随其应用领域的扩大，应用的复杂性也在增加。采用对芯片直接编程的方法难以适应复杂应用的要求，开发DSP操作系统势在必行。现在的DSP操作系统按支持的平台范围可分为专用操作系统和通用操作系统。专用操作系统一般由DSP芯片的开发商自己开发，只支持该公司的芯片，如TI公司的DSP / BIOS。在这种系统上开发的软件，运行效率高，但移植性很差。另一种是通用的DSP操作系统，它们支持广泛的DSP芯片类型，在其上开发的软件可移植性好，但是这种操作系统的费用昂贵，如风河（WIND DIVER）公司的VSPWoks、

Enea OSE System AB公司的OSE操作系统。

众所周知，Linux是一种免费的操作系统。它开放源代码，内核功能强大（支持多任务、多线程），大小和功能都可定制，具有良好的开发环境，软件资源丰富，系统实时性较好，非常适合嵌入式的应用。将Linux移植到DSP芯片上，形成嵌入式的DSP操作系统，是一种低成本、高效率的DSP开发解决方案。

5.5.1 Linux 内核的层次

Linux内核的层次十分清晰，如图5.4所示。内核的五个子系统都通过硬件抽象层（hardware abstract layer ）访问硬件，这使得Linux移植到不同的硬件平台非常容易，只需改动硬件抽象层中与硬件相关的代码即可。

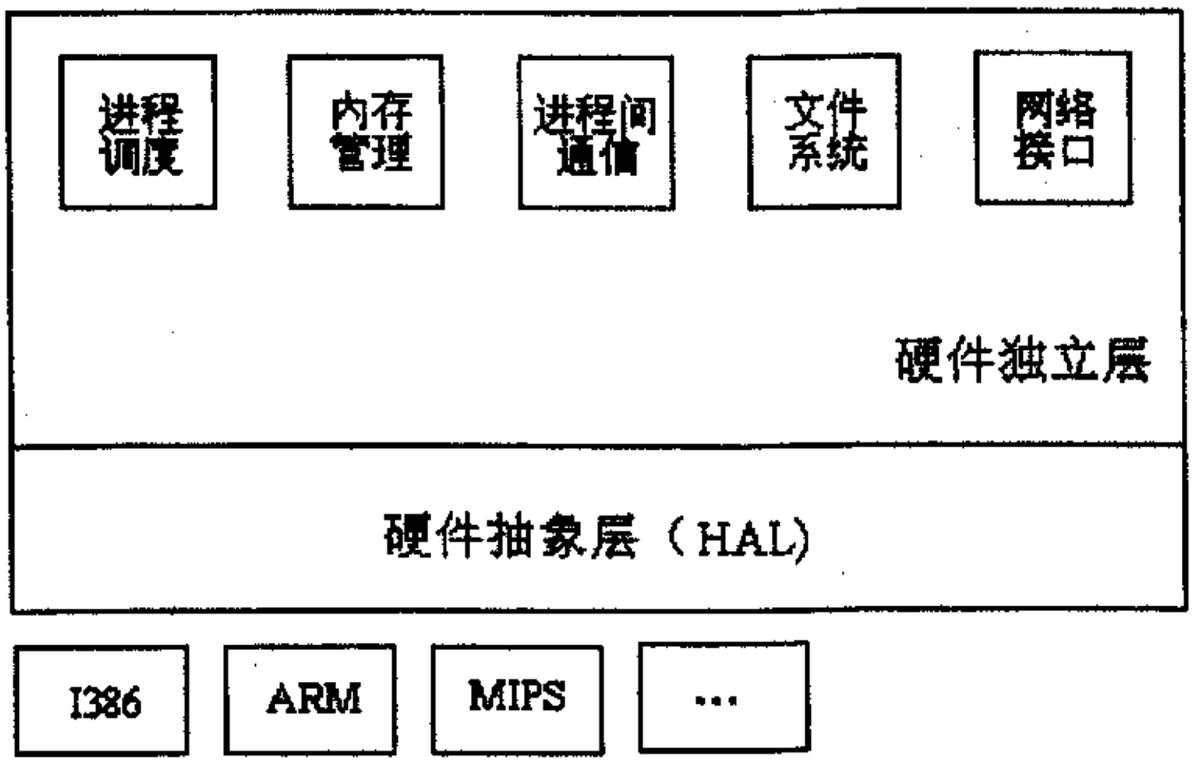

图5.4 Linux内核的硬件抽象层

5.5.2 移植Linux 系统至 DSP 平台

Linux的内核不是针对DSP平台设计的，目前也不支持DSP芯片。移植的工作主要包括两方面：

（1）对内核进行裁减、改造，使其适合DSP应用；

（2）重写内核中与平台相关的代码。

㈠ 要考虑的因素

*DSP芯片主要应用在实时性强、数据量大、计算密集的领域［1］，它不适合处理跳转指令较多的程序；

*DSP应用采用的存储设备种类少，一般为ROM / RAM，存储空间要求低；

*Linux内核本身不支持DSP平台；

*Linux内核采用单块结构，进程调度算法采用基于优先级的调度算法，可以保证对实时任务较好的支持，但它不能应用于硬实时任务[37]；

* 对内核结构和运行体制的改动要考虑到兼容性的问题。

㈡ 总体方案

Linux内核最初并不是为嵌入式应用而设计的，它采用单块结构，在移植时须对内核进行相应的裁减、改造以适应DSP应用。改造主要涉及到进程调度、内存管理、文件系统等方面。

进程调度是Linux内核的核心，它使用基于简单优先权的调度算法来选择下一个占用CPU的进程。优先权的确定和多个因素有关，其中一个重要的因素就是进程是否为实时进程，实时进程的优先权高于普通进程。同为实时进程，采用FIFO（先来先服务调度）和RR（时间片轮转法）进行调度。虽然调度程序保证了实时进程可以优先获得CPU资源，甚至抢先运行（preempt），但是它不能满足硬实时任务的需要。RT_Linux提供了一种改进方案，它建立了一个直接面对处理器的小内核，此内核独立于原Linux的标准内核，并且拥有自己的调度程序[38]。原Linux内核在这个小内核的基础上以最低的优先权与其它实时任务分享处理器，由此保证对硬实时任务的支持。

Linux内核的存储器管理子系统是针对有内存管理单元的处理器设计的，它采用了基于分页存储管理的虚拟存储器技术。虚拟地址被送到内存管理单元（MMU），把虚拟地址映射为物理地址。DSP芯片一般不包含内存管理单元，而且DSP应用对存储空间的要求也不高，采用虚拟存储技术既复杂效率又低。考虑到兼容性问题，保留Linux的分页存储管理方案。精简后的存储器管理子系统采用分页存储管理方案，不支持虚拟内存，不提供对进程存储空间的保护。程序执行时，必须一次将其全部页面调入内存，程序设计人员必须要考虑到程序所需存储空间的大小及实际存储空间的大小，避免出现存储空间不足的情况。

DSP应用的存储设备一般采用ROM / RAM / FLASH，存储空间也不大（一般在1兆以内），数据管理的复杂度小。在DSP平台上采用VFS，占用存储空间大（约230KB左右），执行效率低。改进的方法有两种。其一是采用嵌入式文件系统，它本身占用的存储空间较小，

另外内核支持该文件系统所需的代码更少，可以大大减小Linux内核文件系统占用的存储空间。如ROMFS;，一种非常小的只读文件系统；RAMFS文件系统,用于将文件暂时保持在主存中；JFFS文件系统，用于将文件保存在RAMDISK中。另一种方法是将文件系统作为一个存储设备的驱动程序来实现。进行数据读/写时，通过设备驱动程序直接访问存储设备中的数据。较前一种方法，该方法所需的存储空间更小，适用于数据管理简单的应用。

除了以上谈到的三方面外，还有一些问题需要考虑。为了在目标系统上引导Linux内核，必须要有一个引导程序。内核引导成功后，要加载系统初始化进程。另外，内核中还必须包含相应的设备驱动程序，如DSP芯片自带的串口的驱动程序，它们和具体的DSP平台相关。一个完整的嵌入式Linux系统至少包括以下部分：

* 引导程序：用于在目标系统上引导Linux内核；
* Linux内核：由内存管理、进程管理和定时服务构成；
* 初始化进程；
* 硬件驱动程序；
* 一个或多个进程，提供所需功能。

在某些应用中，你可能需要加上：

A. 一个文件系统；

B. TCP/IP网络堆栈。

㈢ 代码移植

Linux内核不支持DSP平台，要完成内核移植，必须修改、编写内核中与平台相关的代码。Linux内核中与平台相关的代码都放在arch目录下，arch又依据不同平台，分为不同目录（以后简称为arch/%%%目录）。如目录arch/arm表示内核源代码中与arm平台相关的代码。arch/%%%目录又至少包含三个子目录：kernel，lib,和mm。kernel目录存放内核中与平台相关的代码，如信号量的实现等；lib,目录存放与平台相关的通用函数的实现；mm目录中存放与平台相关的内存管理代码。除了这三个目录外，arch/%%%目录下还包括其它一些目录，如,boot目录存放在该平台上启动内核所实用的程序的源代码。从arch/%%%目录下，可以找出Linux内核中与特定的平台相关的代码，从而在向DSP平台移植时，就可以参照着写出与DSP平台相关的Linux内核源代码。

㈣ 移植所需环境

Linux内核移植环境至少包括：

* 安装有Linux操作系统的PC机一台；

* 一套DSP目标板；

*一个交叉平台的编译器，用于在PC机上编译生成目标板上的可执行代码。目前，Linux平台上最好的交叉平台编译器是GCC，其最新版本已经支持TI公司的TMS320C3x、TMS320C4x浮点运算DSP处理器；

* 目标平台上的编译链接库。

由于Linux内核本身不支持DSP平台，故移植工作量很大。内核中与平台相关的代码：包括约2万行的C语言代码和2千行的汇编语句，都必须针对DSP平台重写。若只保留Linux内核的进程调度、进程间通信，内存分配采用分页存储管理，文件系统作为一个驱动程序实现，内核大小可以控制在200KB以下。移植成功后，可以有效利用GNU Linux免费的软件资源，减小DSP系统的开发难度，从而大幅度地降低开发成本。

6 嵌入式 Linux 系统设备驱动程序开发

开发一个完整的嵌入式实用系统，就需要为系统外设编写自己的驱动程序。而在进行设备驱动程序开发时，通常需要编写相当的设备相关代码、以及管理 I/O 接口和处理硬件数据等的操作函数；通过对驱动开发的探究，本文利用虚拟字符设备来最大限度地封装硬件设备驱动的具体细节和其特定信息模式，以降低嵌入式 Linux 系统程序开发调试的难度。最后，总结了两种虚拟字符设备用以实际的嵌入式开发。

在开发一个成功的嵌入式系统过程中，工作量最大、占用时间相对较长也是开发的一个难点就是对设备驱动的编写。linux 将设备驱动程序引入内核增强了设备的驱动能力，系统定义了简洁完善的驱动程序界面，方法是使用数据结构 struct file_operations[39]，这个数据结构中包括许多操作函数的指针，如打开 (),结束 (),读 () 写 () 等，每个驱动程序按照自己的需要做独立实现，把自己提供的功能和服务隐藏在这个统一界面下。但由于硬件芯片集成度的提高，嵌入式系统也延伸到各种专业领域，相对驱动程序的设计对性能有更高的要求，而且驱动程序的代码量正在逐步增加。过去的这种封装远远难以满足要求。为此，我们要为每个具体的硬件编写相当的底层操作函数和设备相关代码，加大了嵌入式应用系统的开发难度。 我们希望发现一些共性，来解决这个问题。

6.1 驱动开发探究

6.1.1 Linux 的硬件设备驱动

Linux 下硬件设备的功能都被文件系统接口或 socket 接口等高层的内核功能屏蔽。实际上，这些接口是 Linux 操作系统利用了“系统门”这个硬件界面（通过调用 int $0x80 机器指令），构造出的形形色色的系统调用作为访问硬件资源的软件界面；只是这些系统调用的软件界面又被内核进行了标准封装，隐藏了有关计算机硬件实现的底层细节，最终统一定义挂接在虚拟文件系统下形成了三种标准的驱动 I/O 接口[40]，它们是：字符设备接口、块设备接口、和网络接口；字符接口提供对底层硬件设备的非结构化访

问，字符设备只能使用它们可能使用的或多或少的字节大小进行操作。而块设备接口只能以块为单位接受输入和返回输出（块的大小根据设备的不同而不同）；同时，块设备接口对于请求有缓冲区，因此它们可以选择以什么顺序进行响应，对于存储设备而言这一点是很重要的，因为在读写连续的扇区时比远远的分离的扇区更快；块设备可以通过它们的设备文件存取，但通常是通过文件系统存取，只有块设备支持挂接的文件系统。网络接口，包括网卡和协议栈等复杂的网络输入输出服务，网络设备在文件系统中并不出现，他们只能通过 socket 接口访问[41,42]。

要编写设备驱动程序就不可避免地要涉及到特殊设备文件，设备文件的原始目的是允许进程和内核中的设备驱动程序通信以通过它们和物理设备通信（调制解调器，终端，等等）。随着 Linux 内核技术的完善和其 I/O 效率的提高，虽然数据已不再是那种没有格式的字符流模式，但设备文件的引用被保留下来并得以相应的扩展，内核提供了一整套的设备注册和设备访问的数据结构，这些数据结构提供了记录设备相关信息的场所及用户进程对设备进行访问的机制，进程访问设备文件的模式如图 6.1 所示。通过这些数据结构建立起了核心代码和驱动程序模块之间的联系。

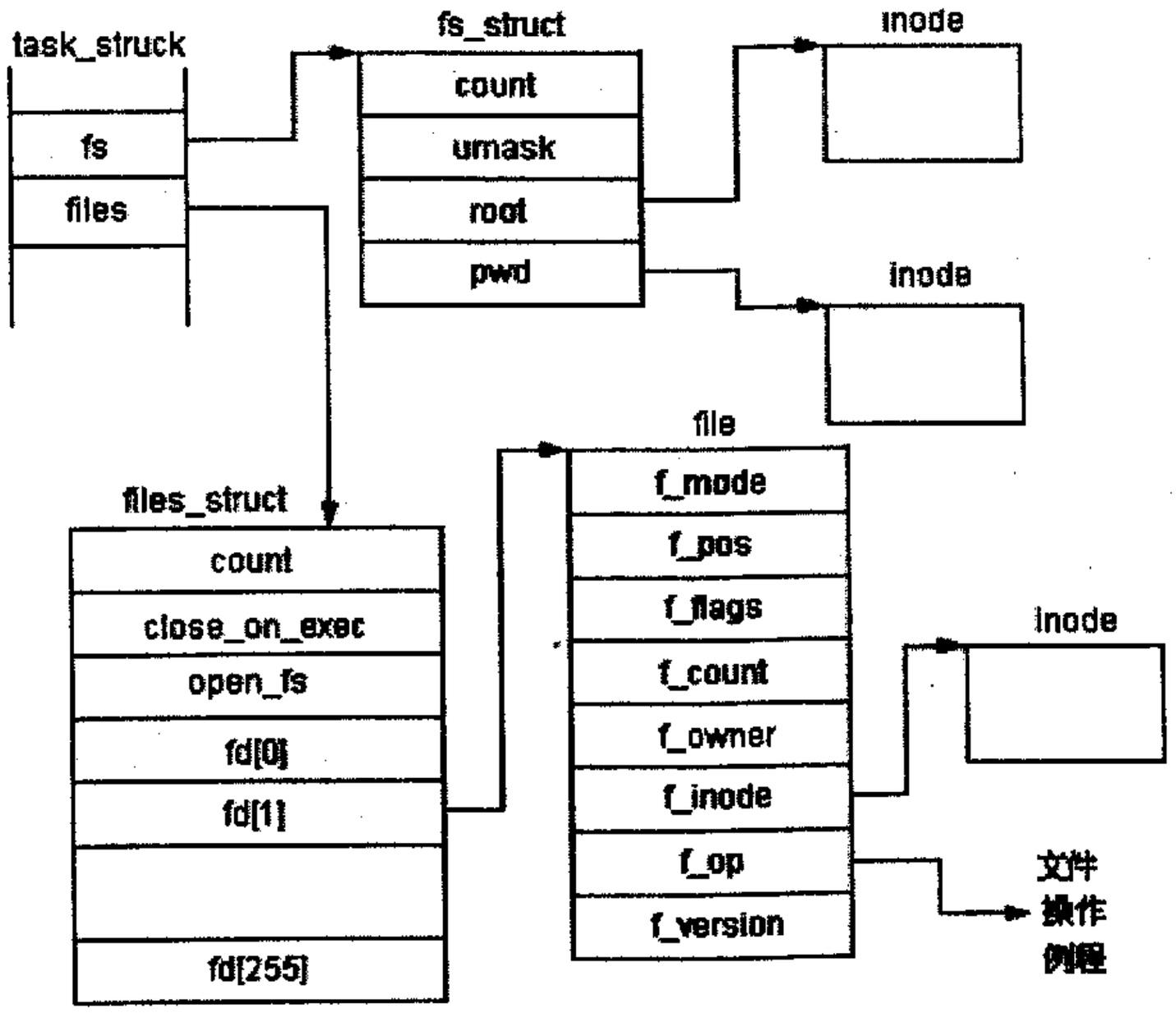

图 6.1　进程的设备文件访问

每个设备驱动程序负责某种硬件，驱动硬件的过程就是在内核机制下实现进程和设备文件的对话过程。因此，设备文件应该表现物理设备，其传输的数据大小和排列格式都要受到其驱动硬件的限制。具体的驱动程序要在以上三个通用设备驱动程序的基本结构种添加相当的必须要使用到它们各自特性的特定设备驱动代码。这些特定代码包括：设备状态管理、I/O 内存缓存管理、I/O 同步、重定位、数据在内核中的移动模式、以及文件访问的指针移动等，同时也要求基于硬件记录格式的相关的底层处理。

6.1.2 嵌入式 Linux 下的硬件设备驱动

输入/输出是操作系统设计中最凌乱的部分，Linux 操作系统根据自己的硬件体系结构特性，力争在提高效率的前提下，希望能用一种统一的方式处理所有的设备。而在一个具体实用的嵌入式 Linux 系统中，原有的通用性作用不大，构造一个成功的应用产品，除了裁减定制内核外，工作量几乎都落在重写与日俱增的驱动代码上；而且应用环境的简单，也使得某些高效的硬件无法使用。所以人们在嵌入式 Linux 的驱动开发中应更着重于输入/输出的通用功能结构上。则最好在原有通用接口的基础上再做进一步地封装，即对硬件特性做最大限度地抽象。

分析各种驱动程序的特定驱动代码部分发现，对于每一类设备是有一定的规律的。它的许多操作也是相似的，许多数据结构也是可以公用的。如：声音设备驱动要使用直接内存存取通道，显示设备要提供对显存的操作，硬盘驱动要处理复杂的缓冲区结构等。可以采用更高一层的封装方法，为同一类设备定义一组文件层次 file_operations 结构中的接口指针函数，在其中处理大多数底层相关操作，如各种缓冲区的申请和释放等，而仅将具体操作底层硬件的一小部分留给驱动开发人员，用以降低驱动开发难度。所以可以为驱动程序提供另外一个文件层到底层驱动的接口，通常为一个结构体，其中包含成员变量和函数指针等，不同的设备驱动有着不同的数据结构。为了方便设备管理，这个数据结构根据硬件设备的具体特性编写，采用合适的成员变量科学仿真记录各种设备的全部信息，包括设备的设置参数、状态、以及操作函数指针等。这样，一方面，保证了文件层 I/O 接口 file_operations 的一致性；另一方面，驱动的开发人员也不必了解设备驱动的太多细节，只需关注硬件相关的 I/O 操作即可。

但为了便于用户进程与这个结构体的直接交互，可以把此结构体进一步抽象成一个

虚拟字符设备，即为设备提供了一个非结构化的接口。管理对设备的直接 I/O 操作和处理 I/O 同步问题。也正是由于字符设备的非结构化和原始他才可依据设备要求，定义操作对象的大小和特性，把底层硬件的细节信息与驱动程序在内核的其他部分分开。底层硬件的所有其它限制也都通过字符接口传递给用户，因此用户进程必须要遵守底层硬件的限制。如，字符设备文件的偏移量和传输数据的大小必须是设备要求之块大小的整数倍。同时，由于嵌入式系统资源紧张，如内存的限制；有些字符接口在把用户数据放到一个 I/O 队列之前，不会将他们先放到内核的缓冲区内。而实际上，字符接口管理那些直接对用户地址空间进行的 I/O 操作。物理设备会限制传输的数据大小和排列情况。然而，传输的数据大小并不受到系统缓冲区中内部缓冲区大小的限制，因为我们并没有使用这些缓冲区。而这些对于资源相对紧张的嵌入式系统又是最有意义的。

6.2 虚拟字符设备的实现

根据硬件传输数据量的大小和驱动速度要求的不同，将虚拟字符设备分为两种，一种是使用内存映像，可以直接对用户的地址空间进行 I/O 操作；另一种是内核中有自己的缓冲区叫”buffered”方式，即在系统中占用内核缓冲区[43]。以下就对这两种字符设备接口的实现原理、特性及编写等分别加以讲述。

6.2.1 使用内存映像。

㈠实现原理及其特性

Linux 通过内存映像机制来提供用户程序对内存直接访问的能力。内存映像的意思是把内核中特定部分的内存空间映射到用户级程序的内存空间去[44]。也就是说，用户空间和内核空间共享一块相同的内存。这样做的直观效果显而易见：内核在这块地址内存储变更的任何数据，用户可以立即发现和使用，根本无须数据拷贝。而在使用系统调用交互信息时，在整个操作过程中必须有一步数据拷贝的工作——或者是把内核数据拷贝到用户缓冲区，或只是把用户数据拷贝到内核缓冲区——这对于许多数据传输量大、时间要求高的应用，这无疑是致命的一击：许多应用根本就无法忍受数据拷贝所耗费的时间和资源。

曾经为一块高速采样设备开发过驱动程序，该设备要求在 20 兆采样率下以 1KHz 的重复频率进行 16 位实时采样，每毫秒需要采样、DMA 和处理的数据量惊人，如果要使用数据拷贝的方法，根本无法达成要求。此时，内存映像成为唯一的选择：我们在内存中保留了一块空间，将其配置成环形队列供采样设备 DMA 输出数据。再把这块内存空间映射到在用户空间运行的数据处理程序上，于是，采样设备刚刚得到并传送到主机上的数据，马上就可以被用户空间的程序处理。

实际上，内存映射方式通常也正是应用在那些内核和用户空间需要快速大量交互数据的情况下，特别是那些对实时性要求较强的应用。嵌入式 Linux 下那些需要进行大量的数据交换和快速处理数据的硬件驱动程序，如显示设备就可典型地使用内存映像方式。这样可使我们可以直接利用用户空间处理的结果，相对于 lseek/write 系统调用来说，将图形显示内存直接映射到用户空间可以显著提高效能。而建立一个基于虚拟内存的字符设备接口则又可从根本上减少此类驱动开发的工作量，这都是嵌入式 Linux 应用系统开发运行所需要的。

㈡ 编写一个样本驱动程序

现我们只要定义一个结构体、编写相应的初始化函数和此结构体中定义的底层操作函数并配以一个标准的操作界面即可。就以LCD显示器底层驱动的字符设备接口—— 帧缓冲驱动为代表介绍。

·定义结构体

由于 LCD 显示设备的特殊性，在驱动层的接口不但包括底层函数， 还要有一页记录当前设备独立状态的数据，则为帧缓冲设备定义的驱动层接口为 struct fb_info 结构。这个结构是唯一在内核空间可见的。代码如下：

```
struct fb_info {
    ......
    char modename[40];              /* default video mode */
    kdev_t node;
    int open;                       /* Has this been open already ? */
    struct fb_var_screeninfo var;   /* Current var */
    struct fb_fix_screeninfo fix;   /* Current fix */
```

```
    struct fb_ops *fbops;
    char *screen_base;                    /* Virtual address */
    struct display *disp;          /* initial display variable */
      ......
};
```

数据结构中记录了帧缓冲设备的全部信息，包括设备的设置参数、状态、以及操作函数指针等。其成员变量 modename 为设备名称，fontname 为显示字体，screen_base 为屏幕缓冲区的基地址，struct display 用于设置屏幕的现实参数，如分辨率、显示颜色、屏幕大小等。bops 为指向底层操作函数的指针，这些函数需要驱动开发人员根据硬件设备的具体特性编写。另外两个记录设备状态的数据结构 struct fb_var_screeninfo 和 struct fb_fix_screeninfo ，他们分别记录可设置信息和不可设置的设备信息。第一个结构是用来描述图形卡的特性的。通常是被用户设置的。第二个结构定义了图形卡的硬件特性， 是不能改变的，用户选定了哪一个图形卡，那么它的硬件特性也就定下来了。

• 编写初始化函数：

初始化函数首先初始化 LCD 控制器，通过写寄存器设置显示模式和显示颜色，然后分配 LCD 显示缓冲区,在 linux 中可以用 vmalloc（）函数分配一段连续的空间[45]。缓冲区的大小为：

点阵行数*点阵列述用于表示一个像素的比特数/8。缓冲区通常分配在大容量的片外 SDRAM 中，起始地址保存在 LCD 控制寄存器中。

然后初始化一个 fb_info 结构，填充其成员变量，并调用 register_framebuffer(&fb_info)，将 fb_info 登记入内核。部分代码如下：

```
int __init vfb_init(void)
{
    ......
 if (!(videomemory = (u_long)vmalloc(videomemorysize)))
return -ENOMEM;
 strcpy(fb_info.modename, vfb_name);
```

```
        fb_info.changevar = NULL;
        fb_info.node = -1;
        fb_info.fbops = &vfb_ops;
        fb_info.disp = &disp;
        vfb_set_var(&vfb_default, -1, &fb_info);
    ......
        if (register_framebuffer(&fb_info) < 0) {
        vfree((void *)videomemory);
        return -EINVAL;
        }
        printk(KERN_INFO "fb%d: Virtual frame buffer device, using %ldK of video
memory\n",
            GET_FB_IDX(fb_info.node), videomemorysize>>10);
        return 0;
    }
```

·编写成员函数

编写 fb_info 中函数指针 fb_ops 对应的成员函数，对于嵌入式系统比较简单，只需实现以下 3 个函数即可：

```
    struct fb_ops {
        int (*vfb_get_fix)(struct fb_fix_screeninfo *fix, int con,
                struct fb_info *info);
        /* get settable parameters */
        int (*vfb_get_var)(struct fb_var_screeninfo *var, int con,
                struct fb_info *info);
    /* set settable parameters */
    int (*vfb_set_var)(struct fb_var_screeninfo *var, int con,
            struct fb_info *info);
     };
```

struct fb_ops 在中定义。这些函数都是用来获取/设置 fb_info 中的成员变量的，当应用程序调用 ioctl 操作时会调用他们。

• 操作界面

此虚拟字符设备的文件层接口 file_operations 定义如下：

```
static struct file_operations vfb_fops = {
    owner:      THIS_MODULE,
    read:       fb_read,
    write:      fb_write,
    ioctl:      fb_ioctl,
    mmap:       fb_mmap,
    open:       fb_open,
    release:    fb_release,
};
```

其中，fb_read、fb_write 等句柄我们都作了标准定义，处理了许多与显示设备相关的操作，如显存的申请、释放和操作，对寄存器进行设置，读/写屏幕缓冲区等。而映射操作 fb_mmap 则可以将文件的内容映射到用户空间。对于本虚拟字符设备，将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读写这段地址访问屏幕缓冲区，在屏幕上绘图了。

6.2.2 使用内核缓冲区

㈠ 实现原理及其特性

共享内存的方式免去了大批量数据拷贝的时间以及系统资源耗费，但 mmap 完全是基于共享内存的观念了，也正因为此，它能提供额外的便利，但也特别难以控制。最大的困难是没有专门的同步机制可以让用户程序和内核程序共享，所以在读取和写入数据时要有非常谨慎的设计以保证不会产生干扰，所以基于共享内存的字符接口不好解决同步问题。而且，并不是任何类型的应用都适合 mmap，比如像串口和鼠标这些基于流数据的字符设备，mmap 就没有太大的用武之地。

采用 “buffered” 方式，先在内核空间开一个缓冲区，然后用特定的输入输出

方式将数据从：I/O 空间传输到内核缓冲区，然后再调用一个 copy_to_user()函数通过 read() 的系统调用把数据从内核缓冲区传递到用户空间；或再调用一个 copy_from_user()函数通过 write() 的系统调用把数据从用户空间传递到内核缓冲区，从而最终实现用户程序与物理设备的信息通讯。这样，读写仍在系统调用界面下完成，而它们之间只要通过一个内核信号变量，就可解决同步问题，同时，也加强了内核空间和用户空间的相互作用；特别是实现了内核向用户程序直接提交数据和内核对用户进程的直接调用。但这种建立在内核缓冲区的字符接口所传输的数据量就要受到一定的限制可能。

设备驱动程序使用数据结构来保持和它所控制的设备的联系，这些数据结构现在就以设备文件的形式，作为驱动程序的一部分来静态地分配，使用内核中没有分页的内存来存储它们的数据。当应用程序打开字符设备文件时，从进程的文件引用接口 files 指向的 files_struct 中（如图 6.1 进程的设备文件访问所示），找到当前打开文件描述符 fd 来定位文件结构 file ；而 file 结构中的 private_data:字段是跨系统调用保存状态信息的一个非常有用的资源， 就用它来引向一个数据结构，建立起描述此字符设备文件的文件数据结构。同时用 kmalloc 来分配的一片对应大小的内核缓冲区，从而构建了此类虚拟字符设备。

㈡ 编写一个样本驱动程序

我们说网络设备很难找到一个字符接口，但可以利用共享内核缓冲区机制实现，虽然实现的功能简单，数据传输量小，但在嵌入式简单应用环境下仍有一定的实用性。下就以虚拟的网络字符接口为例介绍。

• 定义结构体

```
struct vir_device{
  ...
  char name[8];
  int busy;
  unsigned char *buffer;
  int mtu;
  spinlock_t lock;
```

```
    ...
    ssize_t (*net_write)(const char *buffer, size_t length, int buffer_size);
};
```

这个驱动层的接口不但包括底层函数，还记录了一般网络设备的基本信息，包括设备状态、有关网络数据包参数、内核缓冲区等。其中 name[8] 为设备名称； mtu 保存当前可发送的网络数据包最大传输单位，以字节为单位；buffer 指向内核缓存区，用来存放发送和接收的网络数据；lock 的类型是自旋锁类型 spinlock_t ，它实际以一个整数域作为锁，在同一时刻对同一字符设备，只能有一个操作以防止不同的用户对设备同时访问，所以使用内核锁机制保护防止数据污染；busy 字段用来说明字符设备是否是处于忙状态。以及一个模拟的网络数据包发送函数 (*net_write)， 但我们也可以根据需要，用一个底层操作函数的指针扩展函数定义，实现更复杂的网络功能。

用这个数据结构描述了两个字符设备 sending device 和 receiving device 分别代替网络发送接口和接收接口，定义字符设备 struct vir_device vir[2]，并分配不同的主设备号。而实际上它们只是一个内核共享缓冲区。这两个字符设备 sending device 和 receiving device 从用户空间角度看事实上又是内核空间和用户空间交换数据的缓存区，编写字符设备驱动实际上就是编写用户空间读写字符设备所需要的内核设备操作函数。

- 初始化字符设备

初始化定义没有什么特别之处，只是字符设备数组各个变量共同赋以初值，并为它们的 buffer 成员申请内核缓冲区。

- 设备操作函数

字符设备的主要操作例程 device_open()、device_release()、device_read()、devie_write()。以上函数的函数指针的原形为：

```
device_open:  int(*open)(struct inode *, struct file *)
device_release: int (*release) (struct inode *, struct file *);
device_read:  ssize_t (*read) (struct file *, char *, size_t, loff_t *);
device_write: ssize_t (*write) (struct file *, const char *, size_t, loff_t
*);
```

• open（）、read（）函数

如果用户应用程序需要执行文件操作时，则内核根据驱动程序的注册信息调用file_operations 结构中定义的对应函数。

open（）函数首先通过*inode 结构来找主设备号，据主设备号找到保存当前设备信息的结构指针，并将其保存在系统地 file 结构的 private_data:字段中，以方便其他函数如：read、write 等访问，如下语句：

```
int Device_Major;
    struct ed_device *edp;
    Device_Major = inode->i_rdev >> 8;
    if (Device_Major == MAJOR_NUM_REC )
    {
 file->private_data = &ed[ED_REC_DEVICE]; //如果主设备号一致就为private_data 设值指向定义的数据结构
      ......
    }
    else
    if (Device_Major == MAJOR_NUM_TX){
        file->private_data = &ed[ED_TX_DEVICE];
         ......
    }
    else
        return -NODEV;
    edp = (struct ed_device *)file->private_data; // 把一个定为此数据结构的指针指向 private_data
    ......
```

read（）函数则可从 open（）函数中保存的 file 结构的 private_data:字段中取得保存当前设备信息的结构指针。这样，它就可以通过这个指针对当前设备进行操作了。可以用以下语句实现：

```
struct ed_device *edp;
edp = (struct ed_device *)file->private_data;
```

以上两个虚拟字符接口各有优缺点，可以根据驱动程序特点使用。使用这种封装方式，确实减少了代码量的编写，最大限度地避免了一些不必要的重复工作；也增强了软件的移植性和系统的可配置性。 甚至有时还可取代物理硬件实现相应的功能，对今后的嵌入式开发具有一定的参考价值。

7 结论与建议

7.1 本文结论

随着嵌入式系统的不断发展，嵌入式操作系统在嵌入式系统中所起的作用越来越大。嵌入式操作系统内核是嵌入式操作系统的重中之重，研究嵌入式操作系统内核的裁剪与定制，使之具备高度的可配置性、可配置性和动态性是嵌入式操作系统剪裁与定制的核心内容。同时开放源码运动给我们带来了丰富的源代码资源，本文所分析的 Linux 内核是开放源代码运动中的杰出代表，应用本文所阐述的裁剪方法对 Linux 内核进行剪裁，使其适用于嵌入式系统具有普遍意义，在我国经济水平比较落后，嵌入式系统需求很大的条件下，具有重大的经济和社会价值。通过一系列分析和研究，初步得出以下结论：

（1）Linux 是个单内核操作系统，但它与传统的单一内核 UNIX 操作系统不同。在普通的单一内核系统中，所有内核代码都是被静态编译连入的。而在 Linux 中，可以动态装入和卸载内核中的部分代码。这一特性使得 Linux 内核具有微内核的灵活性，并能在嵌入式领域得到广泛的应用。

（2）Linux 是一个实用主义的操作系统，以代码执行效率为自己操作系统的第一要务，并没有进行过一个系统的设计工作。但它的特性并不是杂乱无章的，又有一定的严格的层次和规律性，如它分为五个子系统，而每个子系统又都含有同样的两个接口，两个接口为：①.用户进程的系统调用接口。②.对于其他内核子系统的接口。从中我们发现构件的理论是可以办到的，那也就是说总有一个最小的自己是不可改变和替换的部分，即在任何体系结构上都可利用本文的基于调用图的裁剪方法分离出 Linux 操作系统内核中必要部分和可裁减部分，来最大限度地降低使用硬件的限制、减小系统开销、提高移植性。

（3）实时操作在 Linux 下很少受到重视，通过对内核进程管理子系统、内存管理子系统以及文件子系统，做出一定的修改，就可达到嵌入式系统的实时要求，并同时可以进一步减小内核体积与系统开销。

（4）传统的驱动程序框架抽象层次太高，如果设备操作接口 file_operations 中

的函数都由驱动开发来实现，其工作量相当大。而对于每一类设备，其中不仅许多操作相似，而且许多数据结构也是可以公用的，我们可以利用虚拟字符设备来封装硬件设备细节，进一步降低传统的驱动程序框架抽象度，来减少系统开发人员的工作量。

（5）将 Linux 内核移植到 DSP 平台说明 Linux 内核强大的资源管理功能，只要能够把那些常见的，独立性极强的模块而且集中处理系统底层细节的代码装成基本模块，然后在这个基础上搭建操作系统，就会很方便地移植到各个嵌入式硬件平台。

7.2 存在的问题与建议

（1）目前世界上已开发了几百种嵌入式操作系统，这是走向面向应用特定操作系统（Application Specific Operating System,ASOS）的开始。ASOS 概念是硬件中 ASIC（Application Specific Integrated Circcuits）概念在软件中的对偶或对称，相信 ASOS 对于嵌入式软件系统的发展，也将起到像 ASIC 在处理其硬件发展中所起的作用。必须指出的是，当前的许多所谓专用操作系统，其实只是面向特定应用的操作系统，并不是 ASOS。从面向特定应用到面向应用特定，也就是从实践到理论、零散到系统、艺术到工程的变化过程。因而，嵌入式系统的设计理念也应有一个变化，它应具有系统级应用开发技术的特征，将“软件和硬件统一”，突破以往设计中将软硬件看成两个设计领域的局限。

（2）对系统的裁剪定制本文只完成了内核部分，而整个操作系统的改造还有很多工作要做，如对于库的裁定，载入连接方式的选定这些都是今后要继续研发的。

（3）目前，嵌入式系统的实时一般仅限于较简单的有实时要求的串/并口数据采集、浮点数据计算等，而像实时网络这样实时系统的高级应用还需进一步发展。向上扩充 Linux 内核，从功能上扩充 Linux 的实时处理和控制系统，这将对裁剪和定制工作提出更高的要求。

参考文献

[1] 刘艺平，陈宏林.嵌入式操作系统的航向[J].微电脑世界，2002，1（2）：98

[2] 嵌入式系统发展趋势：http://www.ccidnet.com/tech/os/2001/08/23/58 3079.html

[3] 于海东，嵌入式操作系统一览[J].嵌入式系统，2001，8（3）：56～57

[4] 倪光南.Linux 在中国的背景[J].河北大学学报,2001,21(1):1～2

[5] 孙永杰.引人注目的嵌入式 Linux[J].微电脑世界,2000,(35):15

[6] 范志坚. Linux 在嵌入式系统中的应用[J].计算机与现代化,2000,(6):22～25

[7] Rick Lehrbaum, Snapshot of the Embeddedlinux market,May, 2003, see also http://www.linuxdevices.com/articles/AT7301151332.html

[8] Jacob Lehrbaum, William Weinberg, Embeddedlinux operating system and development environment, Operating Systems, February 2002.

[9] Darrick Addison ,Embeddedlinux applications: An overview, August 1, 2001

[10] [美]William Standings.魏迎梅等译.操作系统---内核与设计原理[M].北京：电子工业出版社，2001，5[4]:23 ～25

[11] Yaghmour K.Building Embedded Linux Systems[M]. CA:O'Reilly & Associates,Inc.2003:66

[12] Bovet D P,Cesati M.Understanding the Linux knernel[M].CA:O'Reilly & Associates,Inc.2000:42

[13] 李善平等.内核2.4版源代码分析大全[M]. 北京：机械工业出版社，2002,1：411

[14] 天泽. 嵌入式系统开发与应用实验教程[M].北京：北京航空航天大学出版社，2004，6[1]:1 ～3

[15] [美]Frank Valid,Tony Givargis.骆丽译.嵌入式系统设计[M].北京：北京航空航天大学出版社,2004,9[1]:221

[16] 金西,黄汪,李尧. Linux 小型化技术[J].计算机工程,2001,27(1):3～4

[17] Ori Pomerantz，“Linux Kernel Module Programming Guide”，1998

[18] Andrew S.Tanenbaum,Albert S.Woodhull, "Operating Systems:Design and Implementation 2nd ED."London:Prentic Hall Int'1,1997

[19] SWITZER,R.W: "Operating Systems,A Practical Approach,London: Prentic Hall Int'1",1997

[20] 夏一民等. 实时 Linux 操作系统初探[J]. 计算机应用研究, 2001, 18(1):45～48

[22] 杨朝军等. Linux 嵌入式系统的优化[J]. 重庆邮电学院学报, 2001, 12(1):54～56

[21] DALAL V,RAVIKUMAR C P.Software Power Optimizations in an Embedded System[C].VKSI Design,2001.Fourteenth International Conference on,2001:254～259

[23] David Pitts. Redhat Linux 大全[M]. 北京: 电子工业出版社, 1998:23～43

[24] David Rusling. 路丝林等译. Linux 编程白皮书[M]. 北京机械工业出版社, 2000:55

[25] B.Sribuvasan,S.Pather,T.Hill,F.Ansari,and D.Niehaus, "A Firm Real_Time System Implementation UsingCommercial Off-The-Shelf Hardware and free Software",In proceedings of IEEE Real_Time Technology and Applications Symposium,June 1998

[26] M.Barabanov and V.Yodaiken, " Introducing Real-Time Linux" Joural,Issue34,February1997

[27] 刘云新, 张尧学. 一个基于 Linux 的嵌入式实时操作系统[J]. 计算机工程与应用, 2001, 37(7)64～66

[28] 阎凯等. 386 微型计算机硬件设计原理分析和维修(下)[M]. 北京: 科学出版社, 1994:339～343

[29] 袁静波. Linux虚拟内存管理分析[J]. 开放系统世界, 2002, (6):57～59

[30] 胡晓明, 王强. Linux核心源代码分析[M]. 北京:人民邮电出版社, 2000:50～100

[31] 毛德操, 胡希明. LINUX内核源代码情景分析（上册）[M]. 杭州: 浙江大学出版社, 2001, 5(1):66～68

[32] Andrews. Tanenbaum. 操作系统设计与实现[M]. 北京:清华大学出版社, 1997: 30～55

[33] 沈玉利. Linux的虚拟文件系统中数据结构的研究[J]. 湛江海洋大学学报, 2001, 21(3):61～62

[34] 杨益，郭庆平.Linux 虚拟文件系统实现技术剖析[J]. 交通与计机,2001,19(101):46～49

[35] 刘章仪.Linux文件系统分析[J]. 贵州工业大学学报(自然科学版),2002,31(4):135～137

[36] Rosenblum.m.,and Ousterhout,J.K.: "The Design and Implementation of a Log-Structured File System,"Proc.Thirteenth Symp. On Operating System Principles,ACM,PP.1-15,1991

[37] 王念旭. DSP基础与应用系统设计[M].北京：北京航空航天大学出版社，2001,3(1):82

[38] [英]Neil Matthew.叶小虎等译.linux高级编程.北京：机械工业出版社，2002,1：89

[39] 张海峰.嵌入式 linux 内核及驱动开发[硕士学位论文]. 厦门：厦门大学，2003:28

[40] 纪纯杰，贺晓能.LINUX 内核分析及常见问题解答[M]. 北京：人民邮电出版社 ，2002，7[1]:65～67

[41] Marshall Kirk Mckusick,Keith Bostic. 李善平，刘文峰译. 4.4BSD 操作系统设计与实现[M]. 北京：中国电力出版社 2003，7[1]:33～35

[42] 马季兰等.操作系统原理与 LINUX 系统[M].北京：人民邮电出版社 ，1999，10[1]:72～73

[43] Michael K. Johnson，"Writing Linux Device Drivers"，DECUS '95 in Washington，1995

[44] 毛德操，胡希明.LINUX内核源代码情景分析（下册）[M].杭州：浙江大出版社,2001,5(1):326

[45] Michael K. Johnson and Others，"Linux Kernel Hackers' Guide"，1998

攻读硕士期间发表的论文及所取得的研究成果

1 发表的论文

（1） 《嵌入式 Linux 的驱动程序开发研究》，已被《计算机与现代化》录用。

（2） 《虚拟字符设备在嵌入式 Linux 中的应用》，已被《高性能计算技术》录用。

（3） 《嵌入式 Linux 应用系统的驱动程序开发研究》，已被《计算机与信息技术》录用。

致 谢

值此论文完成之际，谨向我的导师刘福岩教授致以最诚挚的谢意。感谢他将我引入计算机科学的前沿领域——嵌入式 Linux。这一领域方兴未艾，尤其是内核的开发和研究方面，国家正急需大量攻关人才，以迎头追赶欧美等国的先进水平，这无疑为我们提供了难得的机遇。

特别感谢韩燮教授、张咏梅老师、杨秋祥老师、马礼老师、邵坚婷老师及其他诸位老师给我的关心、指导和帮助。

感谢井超、杨晓雯、张瑞萍等同学在学习期间给我的帮助。

感谢在攻读硕士期间所有关心和帮助我的老师、同学和朋友。