第 1 章
嵌入式系统概述

在数字信息技术和网络技术高速发展的后PC时代，嵌入式系统因其体积小、可靠性高、功能强、灵活方便等优点，已经渗透到工业、农业、教育、国防、科研以及日常生活等各个领域，对各行各业的技术改造、产品更新换代、加速自动化进程、提高生产率等起到了极其重要的推动作用。同时，嵌入式Linux操作系统以其开放源代码、易于开发、功能强大、稳定、成本低等优势，迅速跻身于主流嵌入式开发平台。基于嵌入式Linux操作系统的研究和应用具有巨大的学术和商业价值。在嵌入式Linux系统的开发中，嵌入式设备种类繁多的特点，决定了不同的嵌入式产品在开发时都必须设计和开发自己的设备驱动程序。因此，嵌入式Linux设备驱动程序的开发在整个嵌入式系统开发工作中占有举足轻重的地位。

本章对嵌入式Linux系统的基本概念做一个阐述，让读者对嵌入式Linux系统有一个基本的感性认识。

1.1 嵌入式系统

嵌入式系统是目前发展最快的应用领域之一。嵌入式系统用在一些特定的专用设备上，通常这些设备的硬件资源非常有限，并且对成本很敏感，对实时性应用要求很高，特别是对于消费家电的智能化来说，嵌入式显得尤为重要。而当今又是智能设备和家电设备的大融合时代，嵌入式的发展也是日新月异。嵌入式的发展不仅需要大批量专业人才的加入，更需要整个嵌入式开发过程的更新和进步。

根据电气工程协会（IEEE）的定义，嵌入式系统是用来控制、监控或者辅助操作机器、装置、工厂等大规模系统的设备，这仅是功能上的定义，而更具一般性的定义是：嵌入式系统以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。正是由于嵌入式系统的这种可剪裁和可定制性，才使得嵌入式硬件不具备通用性，从而导致嵌入式软件的开发依赖于硬件。

从上述定义可以看出，嵌入式系统是针对特定应用的软硬件综合体，其一般具备以下几方面的特征：

（1）嵌入式系统通常是面向用户、面向产品、面向特定应用的。

（2）嵌入式系统是先进的计算机技术、半导体技术以及电子技术与各个行业的具体应用相结合的产物。

（3）嵌入式系统必须根据应用需求对软硬件进行裁剪，满足应用系统的功能、可靠性、成本、体积等要求。

（4）嵌入式系统开发需要专门的开发工具和环境。

（5）为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在储存器芯片或单片机中。

嵌入式系统目前已经在许多领域被广泛使用，例如日常使用的电视机、冰箱、手机、平板电脑等，此外还有汽车、航空航天领域，如刹车系统、月球探测器、空间望远镜等，都需要嵌入式应用的支持。

1.2　Linux 操作系统

Linux操作系统核心最早是由芬兰的Linus Torvalds于1991年8月在芬兰赫尔辛基大学上学时发布的（Linux 0.11版），后来经过众多世界顶尖的软件工程师不断修改和完善，Linux得以在全球普及开来，在服务器领域及个人桌面领域得到越来越多的应用。

Linux是在GNU公共许可权限下免费获得的，是一款符合POSIX标准的多用户、多任务、支持多线程和多CPU的类UNIX操作系统。Linux以其高效性和灵活性著称。Linux模块化的设计结构使得它能够在价格昂贵的工作站上运行，也能够在廉价的计算机上实现全部的UNIX特性。Linux当前有很多发行版本，较流行的有Red Hat Linux、Debian Linux、RedFlag Linux等。

1.3　Linux 作为嵌入式操作系统的优势

在后PC时代，由于集成电路技术的飞速发展，嵌入式系统的开发从单片机时代进入一个“系统”开发的阶段，嵌入式操作系统也逐渐走上了历史舞台。从国内和国外来看，嵌入式操作系统主要有Windows CE、VxWorks、pSOS、Palm OS等。Linux是一个成熟、稳定的操作系统，由于其在嵌入式开发方面具有其他操作系统无可比拟的优势，经过这几年的发展，已迅速跻身主流嵌入式开发平台，并以每年100%的用户递增数量显示其强大的力量。Linux作为嵌入式操作系统的优势在于：

（1）Linux是开放源代码的免费软件。只要遵守GPL的规定，就可以免费获得Linux内核和其他自由软件的源代码，采用Linux操作系统构建嵌入式系统，可以大大降低开发成本和周期。此外，由于拥有源代码，开发人员可以针对特定的应用来修改Linux内核和软件的源代码，更好地进行开发工作。

（2）Linux具有完善的文档和广泛的技术支持。Linux是互联网充分发展的产物，在网上能够找到许多关于Linux的文档以及强大的技术支持。

（3）Linux内核功能强大，性能高效、稳定。Linux的内核非常稳定，它的高效和稳定性已经在各个领域，尤其是在网络服务器领域得到了事实上的验证。

（4）Linux能够支持多种体系结构，是支持微处理器种类最多的操作系统。目前，Linux已经被移植到数十种硬件平台上，几乎所有主流的硬件平台，如X86、ARM、PPC、MIPS、ALPHA、SPARC等，Linux都支持。

（5）Linux拥有强大的网络功能。随着嵌入式系统的发展，嵌入式系统与Internet结合得越来越紧密。与其他操作系统相比，Linux在网络方面具有较大的优势，基本上所有的网络协议和网络接口都可以在Linux上找到。

（6）Linux大小和功能可定制。Linux继承了UNIX的优秀设计思想，内核与用户界面完全独立，各部分的可定制性很强，可以按照需求进行定制和配置，这对于硬件资源有限的嵌入式系统来说是一个理想的选择。

总之，嵌入式Linux操作系统非常适合用于构建嵌入式系统，但由于Linux是一种通用的操作系统，而不是一个真正的实时操作系统，内核不支持事件优先级和抢占实时特性（2003年年底推出的Linux 2.6内核实现了一定程度上的可抢占性），因此也存在着硬实时性、体积庞大等问题。

由于Linux具有免费开源的特性和嵌入式Linux广阔的市场前景，因此针对上述问题的研究具有巨大的学术和商业价值，国内外不少大学、研究机构和公司都纷纷加入嵌入式Linux的研究开发中，目前国际上对嵌入式Linux的研究开发主要集中在以下几个方面：

- 实时性：在数据采集、控制、音/视频等设备中，对操作系统的实时性有比较高的要求。Linux 并不是一个实时操作系统，因而必须提高实时性以满足这些设备的要求。
- 内核裁减：嵌入式设备资源有限，对软件的体积有比较苛刻的要求。由于 Linux 是单一模块结构，体积较大，不适合直接在嵌入式设备中应用。许多厂商致力于开发符合原 Linux 接口标准的小体积 Linux 内核，并加强其可裁减性和可配置性。
- 集成开发环境：提供完整的集成开发环境是每一个嵌入式系统开发人员所期待的。一个完整的嵌入式系统的集成开发环境一般由编译器、连接器、内核调试/跟踪器和集成图形界面开发平台组成。目前嵌入式 Linux 还没有比较完善的集成开发环境，特别是基于图形界面的特定系统定制平台的研究上，与 Windows 操作系统相比还存在差距。因此，要使嵌入式 Linux 在嵌入式操作系统领域中的优势更加明显，整体集成开发环境还有待提高和完善。

从国家战略目标与产业发展来看，嵌入式软件特别是嵌入式操作系统是实现传统制造业转型与提升的关键技术，它对整体提升我国制造业的竞争能力、大幅度地增强我国软件自主创新能力意义重大。但是，我国当前嵌入式系统研究和开发的整体水平不高，与国际领先水平还有较大的差距，国内的嵌入式设备生产商大多数还是采用国外的商用嵌入式操作系统。而免费开源的Linux在嵌入式领域的发展为我国发展自己的嵌入式操作系统，扭转PC机软件市场的被动局面提供了难得的机遇，也为振兴国内软件行业找到了最佳的突破口。

设备驱动程序在Linux内核中扮演着特殊的角色，它是进入Linux内核世界的大门。对嵌入式Linux设备驱动的研究有助于深入理解嵌入式Linux内核代码。此外，硬件必须有配套的驱动程序才能正常工作。由于嵌入式设备种类繁多的特点，决定了不同的嵌入式产品在开发时都必

须设计自己的设备驱动程序，使得设备驱动程序的开发在整个嵌入式系统开发工作中占有举足轻重的地位。因此，对嵌入式Linux设备驱动的研究是一个很好的技术方向，具有重要的社会和商业价值。

1.4　嵌入式系统的开发流程

目前嵌入式开发从开始立项到最终结束，大致可以分为如下几个部分：需求分析，硬件设计，驱动开发，应用程序开发、整合和调试。在这个流程中，每一步都依赖于之前的步骤，硬件的开发进度很大程度上影响软件的开发。嵌入式开发作为开发过程中比较靠后的位置，其进度严重地依赖软件开发之前的步骤。这种情况不仅影响嵌入式系统的开发周期，对于整个嵌入式项目的管理也是非常困难的事情。

自从集成电路技术出现以来，嵌入式设备得到了迅速的应用，特别是最近几年来，手机和平板电脑得到了长足的发展，这使得嵌入式技术的发展吸引了大量的开发者参与其中。嵌入式系统可以分为有操作系统和无操作系统两大类。嵌入式系统中的操作系统（简称嵌入式操作系统）由于其源码可修改、可定制的特性，使其在嵌入式开发中应用得越来越广泛。嵌入式操作系统为嵌入式软件开发提供了一个统一的平台，在一定程度上缓解了嵌入式应用程序开发中应用程序对底层硬件的依赖。嵌入式操作系统对底层硬件进行一定的抽象，并为上层的应用程序提供了统一的接口，让千差万别的硬件在应用程序看来都是类似的设备。在操作系统中，完成这一艰巨任务的就是硬件抽象层（Hardware Abstract Layer，HAL）。

硬件抽象层是Linux中用于抽象底层硬件，并向上层应用程序提供统一接口的软件。硬件抽象层对同一类硬件进行抽象，向上层提供统一的接口，使应用程序开发人员不需要学习底层的硬件知识，就可以通过统一的接口实现和硬件的交互。这一层的好处是减少了程序员的学习成本，提高了开发效率。

Linux操作系统和硬件抽象层协同工作，屏蔽硬件细节，提供统一接口，使得应用程序不需要修改，就可以把通用设备上开发的应用程序，轻松移植到嵌入式设备上，这也是Linux能成为最流行的嵌入式操作系统的一个重要原因。由于操作系统和硬件抽象层的出现，使得应用程序和硬件设计同时进行成为可能。但是，在实际应用中，很多嵌入式设备的硬件都是定制的，很多硬件在通用的计算机上并不存在。所以，在没有硬件的情况下，虽然操作系统能够提供统一的接口，但由于没有硬件的支持，因此程序写出来无法进行调试。另一方面，嵌入式开发基本上都是以交叉开发来完成的，在没有硬件的情况下，即使程序能够编译成功，也不能看到运行结果，这对于软件的开发是不利的。

因此，本书不少实例使用虚拟驱动来解决这个问题，用虚拟驱动来模拟嵌入式的硬件，使得软件开发可以在没有硬件支持的情况下进行编写和调试，并提供了一定的扩展性，程序员可以添加一些自定义的新的虚拟设备。此外，本书还将详细介绍基于虚拟化平台QEMU的嵌入式开发，本质上也可以解决这个问题。

学习嵌入式开发不是一件容易的事情，不像学C语言那样，一上来就进入实战环节，这不现实。所以我们用一章的篇幅介绍理论，让读者有一个“感性”的认识，知道这个领域要学哪些内容。

1.5　嵌入式 Linux 系统的体系结构

嵌入式Linux系统有两层含义：狭义的嵌入式Linux系统指的是嵌入式Linux操作系统，广义的嵌入式Linux系统指的是基于嵌入式Linux操作系统构建的嵌入式系统。嵌入式系统主要分为两大部分：嵌入式硬件和嵌入式软件。嵌入式系统体系结构如图1-1所示。

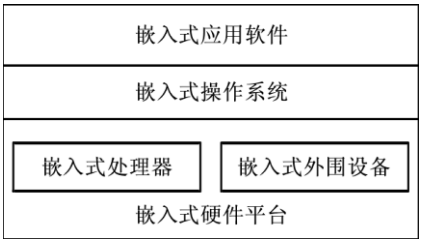

图 1-1

嵌入式硬件部分主要由嵌入式处理器、储存器、I/O端口和外围设备构成；嵌入式软件部分主要由嵌入式操作系统、设备驱动和嵌入式应用软件构成。

1.5.1　嵌入式处理器

嵌入式系统的核心是各种类型的嵌入式处理器，嵌入式处理器与通用处理器最大的不同点在于，嵌入式处理器大多工作在为特定用户群所设计的系统中，它将许多板卡上的接口电路集成到芯片内部，从而有利于嵌入式系统趋于小型化，同时还具有很高的效率和可靠性。嵌入式处理器可以分为以下几类：

（1）嵌入式微处理器（Embedded Microprocessor Unit，EMU）。嵌入式微处理器的基础是通用计算机中的CPU。在应用中，将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的母板功能，这样可以大幅度减小系统体积和功耗。嵌入式微处理器目前主要有ARM、Power PC、MIPS、Am186/88、386EX、68000等系列。

（2）嵌入式微控制器（Micro Controller Unit，MCU）。嵌入式微控制器一般以某一种微处理器内核为核心，芯片内部集成ROM/EPROM、RAM、I/O、串口、脉宽调制输出、A/D、D/A、Flash RAM等各种必要功能和外设。嵌入式微控制器目前主要有8051、P51XA、MCS-96/196/296、C166/167、MC68HC05/11/12/16等系列。

（3）数字信号处理器（Digital Signal Processor，DSP）。数字信号处理器对系统结构和指令进行了特殊设计，使其适合执行DSP算法，编译效率较高，指令执行速度也较快，在数字滤波、快速傅里叶变换、频谱分析等方面得到了大量应用。嵌入式数字信号处理器比较有代表性的产品是TI的TMS320系列和Motorola 的DSP56000系列。

（4）嵌入式片上系统（System On Chip）。嵌入式片上系统指的是在单个芯片上集成一个完整的系统。所谓完整的系统，一般包括中央处理器、存储器以及外围电路等。通用的SOC系列包括Infineon的TriCore、Motorola的M-Core以及Echelon和Motorola联合研制的Neuron芯片等。

1.5.2　嵌入式外围硬件设备

嵌入式外围设备是指在一个嵌入硬件系统中，除中心控制部件（EMU、MCU、DSP、SOC）外的完成存储、通信、保护、调试、显示等辅助功能的其他部件。根据外围设备的功能，主要可分为以下三类。

（1）存储设备类：静态易失型存储器（RAM、SRAM）、动态存储器（DRAM）、非易失型存储器（ROM、EPROM、EEPROM、FLASH）。其中，FLASH以可擦写次数多、存储速度快、容量大、价格便宜等优点在嵌入式领域中有着广泛的应用。

（2）通信设备类：目前存在的大多数通信设备在嵌入式领域中都有着广泛的应用，其中RS232接口（串行通信接口）、IrDA接口（红外线接口）、I^2C总线接口（现场总线接口）、USB接口（通用串行总线接口）和Ethernet接口（以太网接口）应用较广泛。

（3）显示设备类：CRT、LCD/LED和触摸屏等外围显示设备。

1.5.3　嵌入式操作系统

嵌入式操作系统是一种用途广泛的系统软件，负责嵌入式系统的全部软、硬件资源的分配，调度、控制和协调各部件的工作。嵌入式操作系统与一般操作系统相比，在系统的实时高效性、硬件的相关依赖性、软件固化以及应用的专用性等方面具有较为突出的特点。一般情况下，嵌入式操作系统可以分为两类：一类是面向控制、通信等领域的实时操作系统，如VxWorks、pSOS、QNX等；另一类是面向个人数字助理（Personal Digital Assistant，PDA）、移动电话、机顶盒等消费电子产品的非实时操作系统，如Windows CE、嵌入式Linux、Palm OS等。

1.5.4　设备驱动

在Linux内核中，设备驱动程序是一个个独立的“黑盒子”，使某个特定硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。用户的操作通过一组标准化的调用执行，设备驱动负责将这些调用映射到作用于实际硬件设备特有的操作上。

1.5.5　嵌入式应用软件

嵌入式应用软件是针对特定应用领域，基于某一固定的硬件平台，用来达到用户预期目标的计算机软件。嵌入式应用软件和普通应用软件有一定的区别，不仅要求其准确性、安全性和稳定性等方面能够满足实际应用的需要，还要尽可能地进行优化，以减少对系统资源的消耗，降低硬件成本。

1.6　嵌入式 Linux 系统的设计与实现

嵌入式系统的开发涉及两个方面：硬件部分与软件部分。硬件部分提供整个系统开发可见的或可触摸的“实体”，而软件部分则提供这个“实体”内部的功能逻辑，本章我们的目标集中在嵌入式Linux系统的软件开发上面。嵌入式Linux系统从软件的角度来看通常可以分为4个层次：

（1）引导加载程序，包括固化在固件（Firmware）中的启动代码（可选）和Bootloader（引导加载程序）两大部分。

（2）嵌入式Linux内核，包括特定于嵌入式开发板的定制内核以及控制内核引导系统的参数。

（3）文件系统，包括根文件系统（RamDisk）和建立于Flash之上的文件系统。

（4）用户应用程序，特定于用户的应用程序，有时还包括一个嵌入式图形用户界面。

针对以上4个软件层次，构建一个嵌入式Linux系统的基本步骤如下：

（1）搭建嵌入式Linux开发环境。

（2）根据开发板的硬件配置编写Bootloader。

（3）裁减和编译嵌入式Linux内核。

（4）编写设备驱动程序和嵌入式Linux应用程序。

（5）构建嵌入式Linux根文件系统。

1.7　Linux 操作系统内核

Linux源代码的开放性为修改和更新标准Linux内核代码，开发适合目标平台的嵌入式Linux内核以及驱动程序提供了良好的机会，但这一切都必须建立在熟悉Linux内核结构和工作原理的基础之上。

1.7.1　Linux内核的组成

Linux是采用模块化程序设计方法开发的单内核结构的操作系统，可将Linux内核按功能划分为5个部分：进程管理、内存管理、文件系统、设备控制和网络，如图1-2所示。

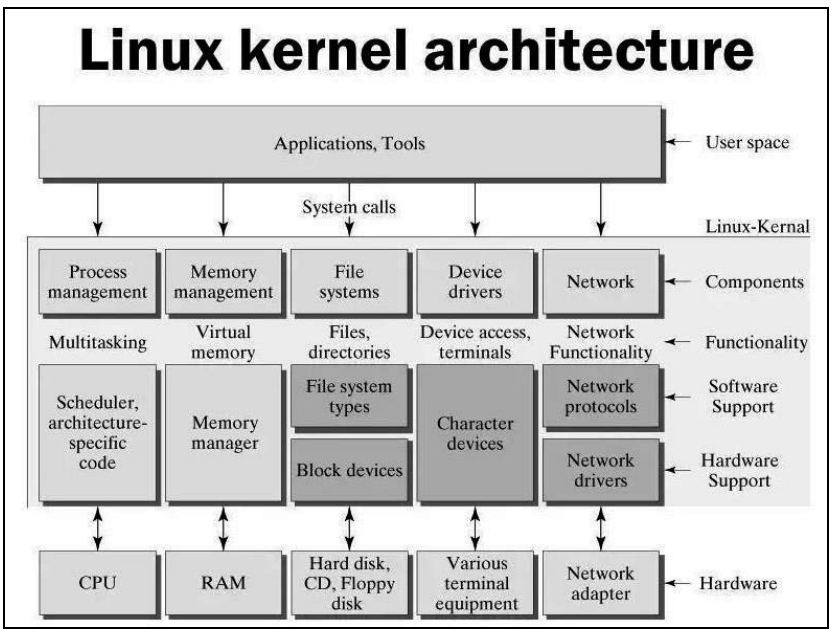

图 1-2

这里，笔者故意画了一幅英文图，笔者觉得，学Linux，常用的Linux相关的英文单词必须知道，因为你以后查国外技术资料时，这些单词都是经常会碰到的。

1. 进程管理

进程管理（Process Management）模块负责创建和销毁进程，并且采用合适的调度策略对进程进行调度，控制进程对CPU的访问，使得各个进程能够公平合理地访问CPU，同时保证内核能够及时地执行硬件操作。除此之外，进程管理还支持进程间各种通信机制。

2. 内存管理

内存管理（Memory Management）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，将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，这样使得Linux的进程可以使用比实际内存空间更多的内存容量。内存管理从逻辑上分为硬件无关部分和硬件相关部分。硬件无关部分提供了进程的映射和逻辑内存的对换，硬件相关的部分为内存管理硬件提供了虚拟接口。

3. 文件系统

文件系统（File System）管理模块用于管理挂接在系统上的文件系统以及文件。Linux虚拟文件系统（Virtual File System，VFS）通过向所有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，从而提供并支持多达数十种不同的文件系统。

4. 设备驱动（Device Driver）

几乎每一个系统操作最终都会映射到物理设备上。除CPU、内存以及其他有限的几个对象外，所有设备控制操作都由与被控制相关的代码（驱动程序）来完成。内核通过为系统中的每个外设嵌入相应的驱动程序来对硬盘驱动器、键盘和鼠标等设备提供支持。

5. 网络接口

网络接口（Network Interface）提供了对各种网络标准协议的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

1.7.2　Linux内核各部分的工作机制

由于嵌入式Linux内核是通过对标准Linux进行裁减和修改后得到的，因此其支持的功能一般来说是标准Linux的子集，但作为一个操作系统，对于进程管理、内存管理和文件系统管理的支持是必不可少的。

1. 进程管理

进程是一个执行中的程序实例，在Linux中每个进程都会经历一个从创建到执行，再到消亡的生命周期。在进程的生命周期中，Linux内核是通过一个名为task_struct的数据结构来描述

进程的。task_struct也称为进程控制块（Process Control Block，PCB）或进程描述符（Process Descriptor，PD），其中保存着用于控制和管理进程的所有信息（进程调度信息、进程间通信信息、文件系统信息、虚拟内存信息等）。

Linux内核可使用系统调用fork()、clone()和vfork()来创建一个进程，这三个系统调用都会以不同的参数调用do_fork()。do_fork()完成了进程创建中的大部分工作，其运行的大致流程如下：

（1）调用alloc_task_struct()为新进程分配两个连续的物理页面，用作task_struct和系统空间堆栈，然后将父进程的task_struct复制到新分配的空间中。

（2）将新的子进程状态设置为TASK_UNINTERRUPTIBLE，以保证它不会投入运行。

（3）调用copy_flags()以更新task_struct的flags成员。

（4）调用getpid()为新进程获取一个有效的PID，再初始化task_struct相应的字段。

（5）根据参数标志，调用copy_files()、copy_fs()、copy_sighand()和copy_mm()来复制或共享打开的文件、文件系统信息、信号处理函数和进程地址空间等。

（6）将父进程的task_struct结构中的counter值（进程的运行时间配额）分成两半，父子进程各一半。

（7）再调用SET_LINKS()将子进程的task_struct结构链入内核的进程队列，然后通过hash_pid()将其挂入可执行进程等待调度。

（8）新的子进程执行后一般会调用exec()函数执行一个新的程序。

Linux中的进程被创建后就进入了执行阶段，进程的执行状态及其转换如图1-3所示。

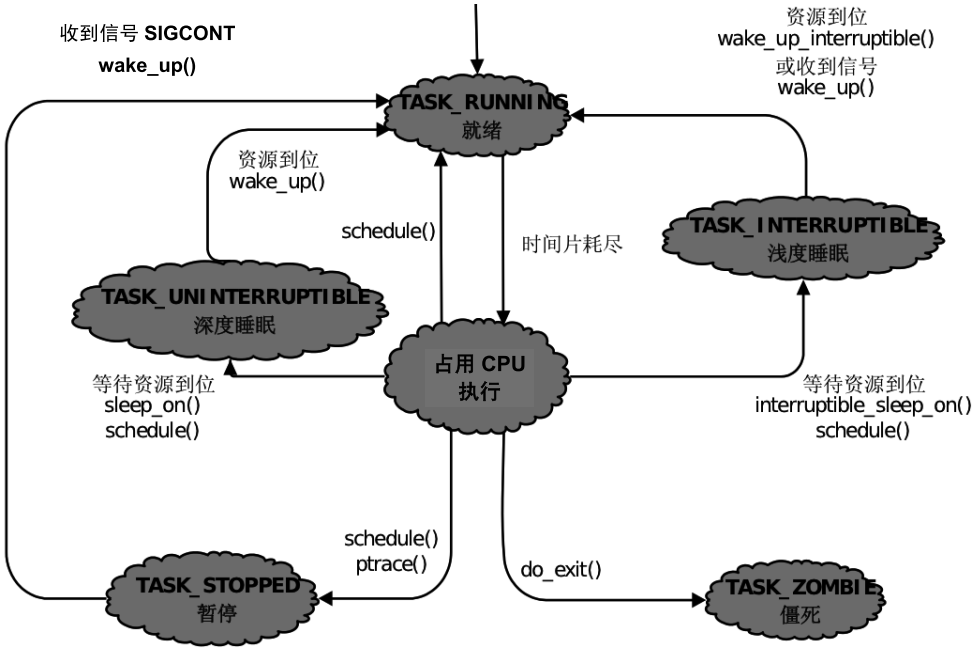

图 1-3

在Linux中，进程状态可以分为R（正在运行，或在就绪队列中的进程）、D（不可中断）、S（处于休眠状态）、T（停止或被追踪）、Z（僵尸进程）、X（死掉的进程）这5个状态。

- R 状态：R 是 Runnable 或 Running 的缩写，表示进程在 CPU 的就绪队列中，正在运行或者正在等待运行，对应图 1-3 中的“就绪”和“占用 CPU 执行”。也就是说，R 状态并不意味着进程一定在运行中（并不一定在使用 CPU），它表明进程要么是在运行中，要么在运行队列中。
- D 状态: 磁盘休眠(Disk Sleep)状态有时候也叫不可中断睡眠状态(Uninterruptible Sleep)，在这个状态的进程通常会等待 IO 结束。该状态是一种深度睡眠，一般表示进程正在跟硬件交互，并且交互过程不允许被其他进程或中断打断。比如往磁盘中写数据，不可以随便终止进程，进程需要接受磁盘返回的信息，对应图 1-3 中的“深度睡眠”。
- S 状态：S 是 Sleeping 的缩写，也就是睡眠状态，意味着进程在等待事件完成，这里的睡眠有时候也叫作可中断睡眠（Interruptible Sleep），也就是可以随时被终止，接收中断信号，是一种浅度睡眠，比如调用 sleep 函数可以随时被终止，对应图 1-3 中的浅度睡眠。
- T 状态：T 表示 Stopped，也就是停止状态，可以通过发送 SIGSTOP 信号给进程来停止进程。这个被暂停的进程可以通过发送 SIGCONT 信号让进程继续运行，对应图 1-3 中的暂停。
- I 状态：I 是 Idle 的缩写，也就是空闲状态，用在不可中断睡眠的内核线程上。硬件交互导致的不可中断进程用 D 表示，但对某些内核线程来说，它们有可能实际上并没有任何负载，用 Idle 正是为了区分这种情况。要注意，D 状态的进程会导致平均负载升高，I 状态的进程却不会。
- Z 状态：Z 是 Zombie 的缩写，它表示僵尸进程，也就是进程实际上已经结束了，但是父进程还没有回收它的资源（比如进程的描述符、PID 等），对应图 1-3 中的僵死。

我们在Linux下可以用命令ps -aux来查看进程状态，命令结果中的列STAT表示进程状态，如图1-4所示。

```
root@bush-virtual-machine:~# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2 167528 11372 ?        Ss   10:18   0:04 /sbin/init sp
root           2  0.0  0.0      0     0 ?        S    10:18   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   10:18   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   10:18   0:00 [rcu_par_gp]
```

图 1-4

进程状态后面可以跟如下几个修饰符：<（高优先级）、N（低优先级）、L（有些页被锁进内存）、s（包含子进程）、+（位于后台的进程组）、l（多线程，克隆线程）。

进程之所以有5种执行状态，是因为Linux操作系统利用分时技术支持多个进程同时运行。分时技术的原理是把CPU的运行时间划分成一个个规定长度的时间片，让每个进程在一个时间片内运行，当进程的时间片用完时，系统就利用调度程序切换到另一个进程来运行。Linux内核通过调度程序schedule()来分时调度运行各个进程。在进程调度时，schedule()函数完成的主要工作是：

（1）判断进程的active_mm是否为0，检查进程是否在中断或低半部程序中，如果是，则会出错。

（2）调用signal_pending()判断是否有信号发送给处于等待状态的当前进程，如果有，则将其唤醒，如果没有，则将其从运行队列中删去。

（3）遍历可执行队列中的各个进程，为每个进程调用goodness()，计算出其当前具有的权值，最后找到队列中权值最高的那个进程，若该进程的权值不为0 则将获得CPU，若该进程的权值为0，则要重新计算所有进程的时间配额。

（4）选出权值最高的进程后，schedule()最后会调用switch_to()对进程进行切换。switch_to()主要通过堆栈的切换和程序执行的切换来完成进程的切换。

当一个进程完成自己的使命后，会通过调用exit()或者从某个程序的主函数返回这两种方式进入消亡阶段，内核必须释放它所占用的资源并通知其父进程，大部分的工作都是由do_exit()来完成的，其完成的工作主要有：

（1）将task_struct中的标志成员设置为PF_EXITING，调用acct_process()来输出统计信息。

（2）调用__exit_files()、__exit_fs()、__exit_sighand()和__exit_mm()分别递减文件描述符、文件系统数据、信号处理函数和进程地址空间的引用计数。如果其中某些引用计数降为0，则代表没有进程在使用相应的资源，可以彻底释放。

（3）调用exit_notify()向父进程发送信号，通知父进程处理相关事务，并且将进程状态设置为TASK_ZOMBIE。

（4）调用schedule()切换到其他的进程，由于进程的状态为TASK_ZOMBIE，因此进程将永远不会从schedule()中返回。父进程被唤醒后会遍历其子进程的状态，若发现有状态为TASK_ZOMBIE的子进程，则调用release_task()释放子进程残存的资源。

2. 内存管理

内存是计算机系统中最重要的资源之一，内存管理子系统是Linux内核的重要组成部分。Linux的内存管理主要包括物理内存的管理和虚拟内存的管理。

为了有效地使用机器中的物理内存，Linux内核将内存划分成几个功能区域，如图1-5所示。

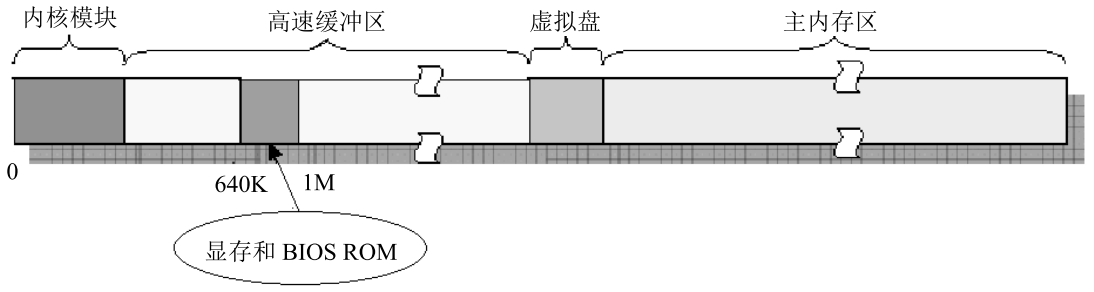

图 1-5

其中，Linux内核程序占据在物理内存的开始部分，接下来是用于供硬盘等块设备使用的高速缓冲区部分。当有一个进程需要读取块设备中的数据时，系统会首先将数据读到高速缓冲区中；当有数据需要写到块设备上去时，系统也是先将数据放到高速缓冲区中，然后由块设备的驱动程序写到设备上。最后的部分是供所有进程可以随时申请使用的主内存区。所有程序在使用主内存区之前，都要首先向内核的内存管理子系统提出申请。

Linux内核支持多个进程同时运行，但内存是一种很有限的资源，它通常容纳不下系统中的全部活动进程。内存管理子系统利用了虚拟内存技术来解决这个问题，以满足活动进程所需

的内存空间。虚拟内存通过使用磁盘作为RAM的扩展，而使系统看起来有多于实际内存的内存容量。Linux内核利用地址映射机制、内存分配回收机制、缓存和刷新机制、请求页机制、交换机制和内存共享机制来实现虚拟内存。内存管理程序通过映射机制把用户程序的逻辑地址映射到物理地址。当用户程序运行时，如果发现程序中要用的虚地址没有对应的物理内存，就发出了请求页要求。如果有空闲的内存可供分配，就请求分配内存，并把正在使用的物理页记录在缓存中。如果没有足够的内存可供分配，那么就调用交换机制，腾出一部分内存。另外，在地址映射中要通过翻译后援存储器（Translation Lookaside Buffer，TLB）来寻找物理页；交换机制中也要用到交换缓存，并且把物理页内容交换到交换文件中，也要修改页表来映射文件地址。

3. 文件系统管理

Linux内核利用虚拟文件系统支持很多不同的逻辑文件系统和很多不同的硬件设备。虚拟文件系统对用户隐去了各种不同文件系统的实现细节，为用户程序提供了一个统一的、抽象的、虚拟的文件系统界面，此界面的主体为file_operation、dentry_operations、inode_operations和super_operations结构，它们是代表着文件系统的逻辑文件操作、目录操作、物理文件操作和文件系统超级块操作的函数跳转表，用来指向具体文件系统实现的入口函数。文件管理子系统向用户提供系统调用mount()和umount()来实现文件系统的安装和拆卸。mount()负责将一个可访问 的块设备安装到一个可访问的节点上，而umount()则负责把已经安装的设备拆卸下来。

1.8　Linux 设备驱动程序

1.8.1　Linux设备驱动概述

现在的处理器都具有保护系统软件不受应用程序破坏的功能，实现这个功能的方法是在处理器中实现不同的操作级别，不同的级别具有不同的功能，Linux使用处理器的两种级别，应用程序运行在最低级别，即用户空间（或用户态），内核运行在最高级别，即内核空间（或内核态）。

任何一个嵌入式系统都是由硬件系统和软件系统组成的。硬件系统和软件系统的协同保证了嵌入式系统的运行。硬件系统是由集成电路和电子元器件构成的，是所有软件得以运行的平台，程序代码的功能最终靠硬件系统上的组合逻辑和时序逻辑电路实现。操作系统是介于应用程序和机器硬件之间的一个系统软件，它掩盖了系统硬件之间的差别，为用户提供了一个统一的应用编程接口。操作系统和应用程序构成了软件系统，应用程序通过应用编程接口使用操作系统提供的服务。而设备驱动程序是操作系统和硬件之间的接口，它为应用程序屏蔽了硬件的细节，使得应用程序只需要调用操作系统的应用编程接口就可以让硬件来完成要求的工作。

不同种类、型号和厂家的设备都有自己的特性。要支持某种设备，就必须提供这种设备的控制代码，如果要把所有设备的驱动程序都写在操作系统内核中，就必然会使内核变得过分庞大，这不利于内核的开发和维护。解决这个问题需要两步：第一步把驱动程序从内核中分离；第二步是在内核和驱动程序之间定义一个统一的接口，双方通过这个接口通信。

因此，对Linux内核来说，驱动程序是一个设备的代表。当Linux内核需要与某个设备通信

时，内核首先找到该设备的驱动程序，然后通过标准的接口调用驱动程序的相应函数完成与设备的通信。

由于定义了内核和驱动程序之间的接口，因此驱动程序的开发变得相对容易。设备驱动程序就是对内核和驱动程序之间的接口函数的实现。硬件厂商和第三方用户都可以开发自己的驱动程序。设备驱动程序由一些私有数据和一组函数组成，它是Linux内核的一部分，设备通过驱动程序与内核其他部分交互。

Linux内核中有许多不同的设备驱动程序，而且其数量还在不断增长，所有的驱动程序都有一些共同的特点：

（1）设备驱动程序运行在内核态，是内核的一部分。如果驱动程序发生错误，就会导致很严重的后果，甚至系统崩溃。

（2）实现标准的内核接口，设备驱动程序必须向Linux内核或它所在的系统提供一组函数来实现这些标准的内核接口函数。

（3）使用标准的内核服务。虽然驱动程序运行在内核态，但是它不能使用所有的内核服务，可以使用的内核服务也必须定义在一个接口函数中。

（4）可装载、可配置。这样可以减少内核的大小，节约系统资源。

初始化配置程序、I/O程序和中断服务程序是Linux下的设备驱动程序的三个主要组成部分。初始化配置程序检测硬件存在与否，能否正常工作，在可以正常工作的前提下初始化硬件。用户空间函数通过I/O程序完成数据的通信。应用程序进行的系统调用是该部分程序完成的。Linux内核负责用户态与内核态的切换。中断服务程序是硬件产生中断后被内核调用的一段程序代码，因为中断产生时Linux内核有正在运行的进程，所以中断服务程序禁止依赖进程的上下文。

1.8.2　设备驱动的功能

设备驱动程序是应用程序和实际设备之间的一个软件层，它向下负责和硬件设备的交互，向上通过一个通用的接口挂接到文件系统上，从而使用户或应用程序可以按操作普通文件的方式访问控制硬件设备。作为Linux内核的重要组成部分，设备驱动程序主要完成以下功能：

（1）对设备初始化和释放。
（2）把数据从内核传送到硬件和从硬件读取数据。
（3）读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
（4）检测错误和处理中断。

1.8.3　设备的分类

Linux内核对设备进行分类管理，共有三类：字符设备（Character Device）、块设备（Block Device）和网络设备（Network Device也称网络接口（Network Interface））。每类设备驱动程序都向内核提供通用接口，内核使用这些通用接口与设备进行通信。

- 字符设备：字符设备只能顺序存储或者传输不定长数据。一些字符设备不使用缓存技术并以字节为单位处理数据。另一些在内部缓冲数据，一次可以传输多字节数据。Linux 内核把字符设备看成是可顺序访问的字节流。常见的字符设备有串口、键盘、触摸屏等。
- 块设备：按可寻址的块为单位（大小从 512B 到 32KB）进行数据处理的设备。块设备使用缓冲技术并允许随机访问。常见的块设备有硬盘、光盘驱动器等。
- 网络设备：网络设备是通过套接口访问的设备，它负责数据包的发送和接收。网络设备不对应任何设备文件，内核设备按照 UNIX 标准给网络设备分配一个名字（如 eth0）。

1.8.4　驱动的分类

前面提到，Linux内核把设备分为三类，相应地，设备驱动程序也分为三类，分别是字符设备驱动程序、块设备驱动程序和网络设备驱动程序。字符设备是以字节为单位逐个进行I/O操作的设备，在对字符设备发出读写请求时，实际的硬件I/O紧接着就发生了，一般来说字符设备中的缓存是可有可无的，而且也不支持随机访问。块设备主要是针对磁盘等慢速设备设计的，其目的是避免耗费过多的CPU时间来等待操作的完成。它利用一块系统内存作为缓冲区，当用户进程对设备进行读写请求时，驱动程序先查看缓冲区中的内容，如果缓冲区中的数据能满足用户的要求就返回相应的数据，否则调用相应的请求函数来进行实际的I/O操作。网络设备是一个能够和其他主机交换数据的设备，它通常是个物理设备，但也可能是个软件设备，如回环（Loopback）设备。网络设备驱动程序负责驱动设备发送和接收数据包。除设备类型外，内核还使用了一个主设备号和一个次设备号来唯一标识设备，主设备号标识了设备对应的驱动程序，而次设备号仅由驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O请求所涉及的那个设备。

Linux操作系统分为用户态和内核态，驱动程序作为应用软件访问硬件的接口部件，具有连接用户态和内核态的功能，并能利用Linux提供的通信技术帮助应用程序和内核代码进行通信。在Linux操作系统中，最常见的用户态访问内核态的方式是系统调用，Linux操作系统向应用程序提供了一系列的函数接口用于内核态访问。

在Linux的世界里，所有东西都被看作文件，当然这也包括设备。Linux通过HAL的抽象把所有的硬件设备都看作文件，对硬件设备的操作也被抽象成对文件的读写，每个设备都被抽象成一个设备文件。Linux支持多种文件系统，设备文件系统（devfs）是专门处理设备文件而产生的，Linux为了给用户统一的文件接口，又对文件系统进行抽象，把所有的文件操作抽象出来，形成虚拟文件系统（Virtual File System，VFS），虚拟文件系统给用户提供了统一的文件操作接口。

在Linux操作系统中，添加内核代码有两种方式，一种是把代码直接编译到内核中，这样每次启动内核，这部分代码都会被加载到内存中；另一种方法是以模块的方式动态地加载到内核中：把代码编译成模块的形式，在需要的时候才加载到内核中，在不需要的时候卸载出来，这样的设计能够保证内核代码的纯洁性，由于模块可以单独编译，这就使得内核可以快速进行编译，此外，由于可以在不重新启动内核的情况下把一个模块加载、卸载多次，因此对模块进行调试也比对内核进行调试效率要高得多。

Linux为常见的硬件都提供了框架，在框架的基础上编写驱动程序可以大大减少编写程序的工作量。Linux把同类硬件设备操作上的共性提取出来，形成了设备驱动框架，设备驱动框架屏蔽了大量重复的编码工作，驱动开发者只需要针对每个硬件之间不同的部分编写代码就可以实现对硬件的控制。如下所示，把一个灯点亮的语句为：

```
echo 1 > /sys/class/leds/red_1/brightness
```

以上语句是向名为/sys/class/leds/red_1/brightness的文件中写入数值1，这条语句会导致下面一系列的过程调用。

（1）系统调用，向文件中写入1的shell命令，会被转换成对brightness文件的open的系统调用，并通过write系统调用向该文件中写入1。这个过程和在Linux编程中向普通的文件中写入数据十分类似。这个过程是用户态程序通过系统调用完成对文件的写操作，其中使用了系统调用技术。

（2）虚拟文件系统，shell程序通过open、write对文件进行操作时，open和write都是Linux的库函数，是用户态程序，而在库函数中最终会调用sys_open、sys_write等系统调用。Linux为文件操作提供了一套系统调用的接口，其中包括sys_open、sys_write、sys_close、sys_write等。这些sys_xxx就是由虚拟文件系统提供的文件访问接口。

Linux驱动框架如图1-6所示。

（3）文件系统，sys_write函数最终要把数据写到具体的文件系统中，而前面所提到的brightness所在的文件系统名为sysfs，这种文件系统和devfs类似，是Linux中比较新的设备文件系统。虚拟文件系统中的sys_write函数最终会调用挂载在其上的sysfs文件系统中的函数。

（4）具体的文件节点，write函数会查找文件/sys/class/leds_1/brightness，并向这个文件中写入数据。

（5）硬件抽象层（Hardware Abstraction Layer，HAL），向sysfs中的brightness文件写入数据，实际上就相当于向硬件抽象层中写入数据，在这里，硬件抽象层把点亮LED灯的操作抽象成向代表LED灯的brightness文件写入数据1，而把熄灭LED的操作抽象成向LED灯的brightness文件写入数据0，这样就屏蔽了底层的硬件细节，对于LED灯这样简单的硬件来说，底层的实现细节也可能是不同的，比如，可以通过GPIO来控制灯，也可以通过脉冲宽度调制（Pulse Width Modulation，PWM）来控制灯，这完全取决于硬件是如何设计的。有些高级的控制也可能会用到，例如通过brightness_set和brightness_get文件可以控制灯的亮度。

（6）设备驱动，设备驱动链接硬件抽象层和具体的硬件。设备驱动需要针对不同的硬件进行开发，每种硬件的控制是不同的。对于LED设备驱动来说，硬件抽象层规定，设备驱动需要实现brightness_set和brightness_get这两个程序接口，硬件抽象层通过回调机制在适当的时候调用这两个函数实现对硬件设备的控制。

经过以上步骤的分析，说明要写一个高质量的设备驱动，不仅需要了解系统底层硬件信息，还需要熟悉从虚拟文件系统到文件系统以及硬件抽象层等多个部件的原理。

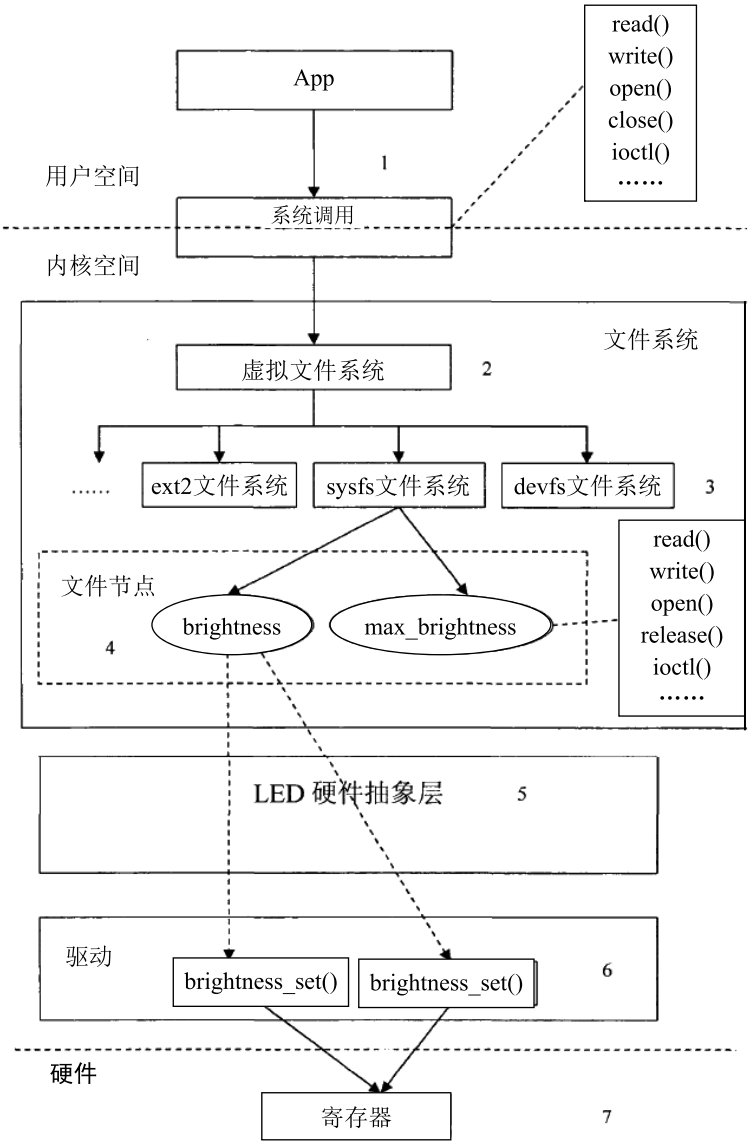

图 1-6

1.8.5　设备驱动与内核的关系

在Linux中将驱动程序嵌入内核有两种方式：一种是静态编译链接进内核；另一种是先编译成模块，再动态加载进内核。如果采用静态编译链接方式，就需要把驱动程序的源代码放在内核源代码目录“Drivers/”下，并修改Makefile文件。在编译链接内核的时候，驱动程序会作为内核的一部分链接到内核镜像文件中。这种方式会增加内核的大小，还要改动内核的源文件，而且不利于调试，不如模块方式方便灵活。模块（Module）在Linux中是一种已经编译好的目标文件，它可以被链接进内核，从而生成可执行的机器代码。如果采用模块加载的方式，驱动程序首先会被编译成未链接的目标文件，具有root权限的用户在需要时可利用insmod命令将其

动态地加载到内核中，而在不需要的时候可利用rmmod命令卸载该模块。

驱动程序加载到Linux内核中后，会利用虚拟文件系统提供的一个统一的接口（file_operations数据结构）挂载到虚拟文件系统下，二者的关系如图1-7所示。

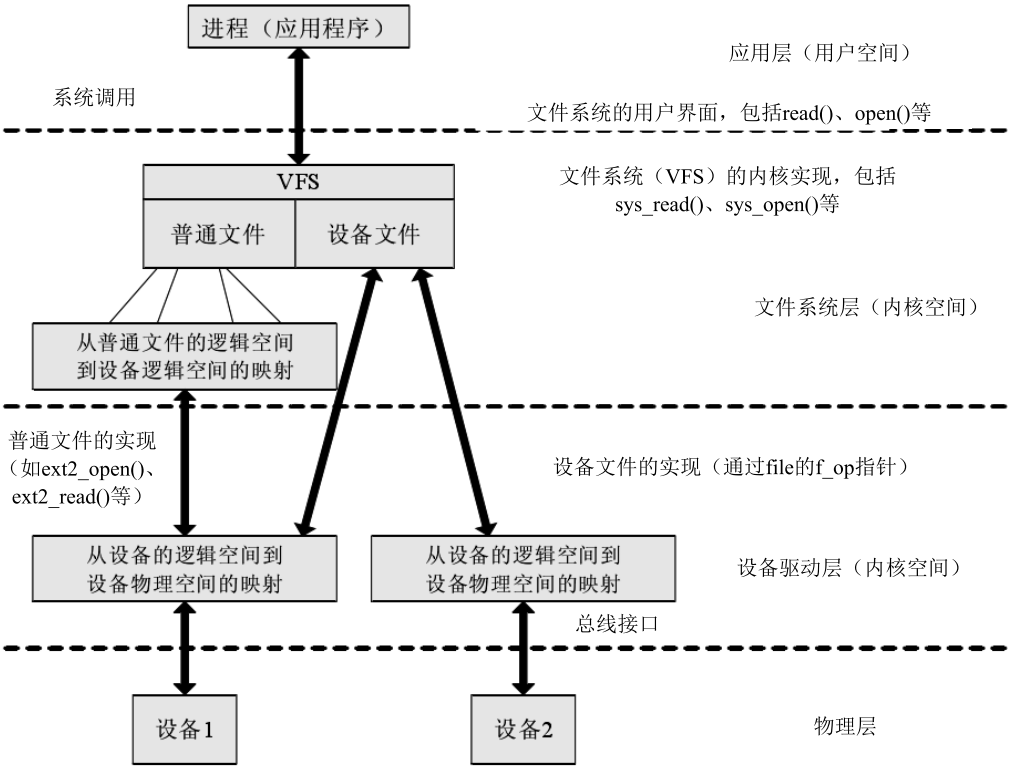

图 1-7

从图1-7可以看出，一个用户应用程序要对一个设备文件进行操作，它的流程大致如下：

（1）用户程序通过Linux所提供的系统调用（如open()、read()等）进入内核。这些函数总共有几百个，提供了几乎所有应用程序中可能需要进入内核运行的操作。

（2）内核中对应这些系统调用的函数是sys_open()、sys_read()等，函数的参数与系统调用函数的参数相同，它们由内核空间中的虚拟文件系统层实现。虚拟文件系统是一个从普通文件和设备文件抽象出来的文件系统层，完成了进入具体的设备文件的操作之前的准备工作。

（3）Linux内核将通过file_operations结构进入具体的设备文件的操作函数，这部分的函数是由设备驱动程序提供的。这些函数负责对设备硬件进行各种操作。

1.8.6　设备驱动的结构

Linux的设备驱动程序按实现的功能大致可以分为如下几个部分：驱动程序的注册与注销、设备的打开与释放、设备的读写操作、设备的控制操作、设备的中断和轮询处理。

1. 驱动程序的注册与注销

向系统增加一个驱动程序意味着要赋予它一个主设备号，这可以通过在驱动程序的初始化过程中调用register_chrdev()或者register_blkdev()来完成。而在关闭字符设备或者块设备时，则需要通过调用unregister_chrdev()或unregister_blkdev()从内核中注销设备，同时释放占用的主设备号。

2. 设备的打开与释放

打开设备是通过调用file_operations结构中的函数open()来完成的，它是驱动程序用来为今后的操作完成初始化准备工作的。在大部分驱动程序中，open()通常需要完成以下工作：

（1）检查设备相关错误，如设备尚未准备好等。
（2）如果是第一次打开，则初始化硬件设备。
（3）识别次设备号，如果有必要，则更新读写操作的当前位置指针f_ops。
（4）分配和填写要放在file->private_data中的数据结构。
（5）使计数增1。

释放设备是通过调用file_operations结构中的函数release()来完成的，这个设备方法有时也被称为close()，它的作用正好与open()相反，通常要完成以下工作：

（1）使计数减1。内核对每个file结构维护其被使用多少次的计数器，无论创建进程还是复制进程，都不会创建新的file数据结构，数据结构只能由open创建，它们只是增加已有结构中的计数，只有在file结构的计数归0时，close这个系统调用才会执行release方法。
（2）释放在file->private_data中分配的内存。
（3）如果使计数归为0，则关闭设备。

3. 设备的读写操作

字符设备的读写操作相对比较简单，直接使用函数read()和write()就可以了。但如果是块设备，则需要调用函数block_read()和block_write()来进行数据读写，这两个函数将在设备请求表中增加读写请求，以便Linux内核可以对请求顺序进行优化。由于是对内存缓冲区而不是直接对设备进行操作的，因此能够很大程度上加快读写速度。如果内存缓冲区中没有所要读入的数据，或者需要执行写操作将数据写入设备，那么就要执行真正的数据传输，这是通过调用数据结构blk_dev_struct中的函数request_fn()来完成的。

4. 设备的控制操作

除读写操作外，应用程序有时还需要对设备进行控制，这可以通过设备驱动程序中的函数ioctl()来完成。ioctl()的用法与具体设备密切关联，因此需要根据设备的实际情况进行具体分析。

5. 设备的中断和轮询处理

对于不支持中断的硬件设备，读写时需要轮流查询设备状态，以便决定是否继续进行数据传输。如果设备支持中断，则可以按中断方式进行操作。

1.8.7　设备驱动的设计和实现步骤

由于模块方式要比静态编译链接方式更加方便灵活，因此在Linux内核基础上二次开发的设备驱动程序一般是按照模块方式实现的。模块化驱动程序的设计和实现流程主要有编写模块化编程子程序、编写自动配置和初始化子程序、编写服务于I/O请求的子程序和编写中断服务子程序4个步骤。

1. 模块化编程子程序

模块化编程子程序主要包括init_module()函数和cleanup_module()函数。init_module()函数在模块被加载到内核时调用，其主要负责向内核注册模块所提供的任何新功能。新功能可能是一个完整的驱动程序，或者只是一个新的软件抽象。对于每种新功能，init_module()函数都会调用与其相对应的内核函数完成注册。字符设备驱动程序对应的内核函数的函数原型为：

```
    int register_chrdev(unsigned int major, const char *name, struct file_operations
*fops);
```

其中，major是为设备驱动程序向系统申请的主设备号，如果为0，则系统为此驱动程序动态地分配一个主设备号。name是设备名，fops是指向与设备驱动对应的file_operations结构的指针。如果register_chrdev()操作成功，驱动程序就会注册到内核中，设备名也会出现在/proc/devices文件中。在成功向系统注册了设备驱动程序后，就可以使用mknod命令来将设备映射为一个设备文件，其他程序使用这个设备的时候，只要对此设备文件进行操作就行了。

cleanup_module()函数在模块被卸载的时候调用，其主要负责从内核中注销模块所提供的各种功能。cleanup_module()函数会调用与其相对应的内核函数完成注销，字符驱动程序对应的内核函数为unregister_chrdev()。

2. 自动配置和初始化子程序

自动配置和初始化子程序一般在设备接入系统或者加载设备驱动时调用，其主要负责检测所要驱动的硬件设备是否存在或是否能正常工作。如果该设备正常，则对这个设备及其相关的驱动程序需要的软件状态进行初始化。自动配置和初始化子程序还负责为驱动程序申请包括内存、中断、时钟、I/O端口等资源，这些资源也可以在xxx_open()函数中申请。

3. 服务于 I/O 请求的子程序

服务于I/O请求的子程序又称为驱动程序的上半部分，调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为和调用的进程属于同一个进程，只是进程的运行状态由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境。驱动程序所提供的与设备的打开、释放、读写和控制操作相对应的入口点函数都属于服务于I/O请求的子程序，并且通过file_operations结构向系统进行说明。file_operations结构的定义如下：

```
    struct file_operations {
      loff_t (*llseek) (struct file *, loff_t, int); //修改文件的当前读写位置
       ssize_t (*read) (struct file *, char *, size_t, loff_t *); //从设备读取数据
```

```
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);  //向设备发送
数据
        int (*readdir) (struct file *, const char * size_t, loff_t *); //读目录，设备驱
动中应设为NULL
     //询问设备是否可读可写，若为NULL，则表明设备总是可读写的
     unsigned int (*poll) (struct file *, struct poll_table_struct *);
      //调用设备控制相关命令
      int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
      int (*mmap) (struct file *, struct vm_area_struct *); //将设备内存映射到进程内存中
      int (*open) (struct inode *, struct file *);          //打开设备
      int (*flush) (struct file *);                         //将缓冲中的数据写回设备
      int (*release) (struct inode *, struct file *);       //关闭设备
      int (*fsync) (struct file *, struct dentry *);        //刷新设备
      int (*fasync) (int , struct file *, int);  //用于异步触发，通知设备FASYNC标志的变化
      int (*lock) (struct file *, int, struct file_lock *);   //给文件上锁
      //将读入的数据按同样的顺序散布到缓冲区中
      ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
      //将多个数据存储在一起
      ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
     };
```

从file_operations结构的定义可以看出，该结构是一个函数指针表，这个表中的每一项都指向由驱动程序实现的、处理相应请求的函数。在编写设备驱动程序时，根据具体设备提供的功能分别编写相应的函数，函数的名称可任意定义，但要符合函数名称的基本要求，然后按照其实现的功能填入file_operations结构相应的位置中。对于驱动程序来说，常用的方法指针主要有open()、release()、read()、write()和ioctl()等，它们对应着设备的打开、释放、读写和控制等基本设备操作。

4. 中断服务子程序

中断服务子程序又称为驱动程序的下半部分。在Linux系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由Linux系统接收硬件中断，再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境相关的函数。

由于对中断的处理属于系统核心的部分，因此如果设备与系统之间以中断方式进行数据交换的话，就必须把该设备的驱动程序作为系统核心的一部分。设备驱动程序在设备第一次打开且硬件被告知产生中断之前，调用request_irq()函数来申请中断，在最后一次关闭且硬件设备被告知不用再进行中断处理后，调用free_irq()函数来释放中断。它们的定义为：

```
     int request_irq(unsigned int irq,  void (*handler) (int irq, void dev_id, struct
pt_regs *regs), unsigned long flags, const char *device, void *dev_id);
     void free_irq(unsigned int irq, void *dev_id);
```

参数irq表示所要申请的硬件中断号，handler为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数irq为中断号，dev_id为申请时告诉系统的设备标识，regs为中断发生时寄存器的内容。device为设备名，将会出现在/proc/interrupts文件中。flags是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是中断处理程序是快速中断处理程序

（flags中设置了SA_INTERRUPT）还是慢速中断处理程序。快速中断处理程序运行时，所有中断都被屏蔽；而慢速中断处理程序运行时，除正在处理的中断外，其他中断都没有被屏蔽。dev_id用来区分共享同一中断的不同中断处理程序（flags中设置了SA_SHIRQ），如果中断由某个处理程序独占，则dev_id可以为NULL。

Linux中断处理程序一般分为两个半部：上半部（tophalf）和下半部（bottomhalf）。上半部的功能是“登记中断”，当一个中断发生时，它首先会调用相应的硬件读写函数，将设备的“中断挂起”位清除，让设备可以继续产生中断，然后把中断例程的下半部挂到该设备的下半部执行队列中去。因此，上半部执行的速度就会很快，可以服务更多的中断请求，下半部则完成中断处理的绝大多数工作。上半部和下半部最大的不同是：上半部不可中断，而下半部可中断，而且可以被新的中断打断。下半部相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。