《Linux 设备驱动开发详解》（第 2 版）

作者：华清远见

第 3 章 Linux 内核及内核编程

本章目标

本章为读者打下 Linux 驱动编程的软件基础。由于 Linux 驱动编程本质属于 Linux 内核编程，因此我们有必要熟悉 Linux 内核及内核编程的基础知识。

3.1～3.2 节讲解了 Linux 内核的演变及新版 Linux 2.6 内核的特点。

3.3 节分析了 Linux 内核源代码目录结构和 Linux 内核的组成部分及其关系，并对 Linux 的用户空间和内核空间进行了说明。

3.4 节讲述了 Linux 2.6 内核的编译及内核引导过程。除此之外，还描述了在 Linux 内核中新增程序的方法，驱动工程师编写的设备驱动也应该以此方式被添加。

3.5 节阐述了 Linux 下 C 编程的命名习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。

3.1 Linux 内核的发展与演变

Linux 操作系统是 UNIX 操作系统的一种克隆系统，诞生于 1991 年 10 月 5 日（第一次正式向外公布的时间）。Linux 操作系统的诞生、发展和成长过程依赖着 5 个重要支柱：UNIX 操作系统、Minix 操作系统、GNU 计划、Posix 标准和 Internet。

1．UNIX 操作系统

UNIX 操作系统是美国贝尔实验室的 Ken. Thompson 和 Dennis Ritchie 于 1969 年夏在 DEC PDP-7 小型计算机上开发的一个分时操作系统。Linux 操作系统可看作 UNIX 操作系统的一个克隆版本。

2．Minix 操作系统

Minix 操作系统也是 UNIX 的一种克隆系统，它于 1987 年由著名计算机教授 Andrew S. Tanenbaum 开发完成。开放源代码 Minix 系统的出现在全世界的大学中刮起了学习 UNIX 系统的旋风。Linux 刚开始就是参照 Minix 系统于 1991 年才开始开发。

3．GNU 计划

GNU 计划和自由软件基金会（FSF）是由 Richard M. Stallman 于 1984 年创办的，GNU 是“GNU's Not UNIX”的缩写。到 20 世纪 90 年代初，GNU 项目已经开发出许多高质量的免费软件，其中包括 emacs 编辑系统、bash shell 程序、gcc 系列编译程序、gdb 调试程序等。这些软件为 Linux 操作系统的开发创造了一个合适的环境，是 Linux 诞生的基础之一。没有 GNU 软件环境，Linux 将寸步难行。因此，严格而言，“Linux”应该被称为“GNU/Linux”系统。

4．Posix 标准

Posix（Portable Operating System Interface for Computing Systems，可移植的操作系统接口）是由 IEEE 和 ISO/IEC 开发的一组标准。该标准基于现有的 UNIX 实践和经验完成，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植。该标准在推动 Linux 操作系统朝着正规化发展起着重要的作用，是 Linux 前进的灯塔。

5．Internet

如果没有 Intenet，没有遍布全世界的无数计算机骇客的无私奉献，那么 Linux 最多只能发展到 0.13（0.95）版的水平。从 0.95 版开始，对内核的许多改进和扩充均以其他人为主了，而 Linus 以及其他 maintainer 的主要任务开始变成对内核的维护和决定是否采用某个补丁程序。

表 3.1 描述了 Linux 操作系统重要版本的变迁历史及各版本的主要特点。

表 3.1　　Linux 操作系统版本历史

版　本	时　间	特　点
0.1	1991.10	最初的原型
1.0	1994.3	包含了 386 的官方支持，仅支持单 CPU 系统
1.2	1995.3	第一个包含多平台（Alpha、Sparc、MIPS 等）支持的官方版本
2.0	1996.6	包含很多新的平台支持，最重要的是，它是第一个支持 SMP（对称多处理器）体系的内核版本
2.2	1999.1	极大提升 SMP 系统上 Linux 的性能，并支持更多的硬件
2.4	2001.1	进一步地提升了 SMP 系统的扩展性，同时也集成了很多用于支持桌面系统的特性：USB、PC 卡（PCMCIA）的支持，内置的即插即用等
2.6	2003.12	无论是对于企业服务器还是对于嵌入式系统，Linux 2.6 都是一个巨大的进步。对高端的机器来说，新特性针对的是性能改进、可扩展性、吞吐率，以及对 SMP 机器 NUMA 的支持。对于嵌入式领域，添加了新的体系结构和处理器类型。包括对那些没有硬件控制的内存管理方案的 MMU-less 系统的支持。同样地，为了满足桌面用户群的需要，添加了一整套新的音频和多媒体驱动程序

从表 3.1 可以看出，Linux 的开发一直朝着支持更多的 CPU、硬件体系结构和外部设备，支持更广泛领域的应用，提供更好的性能 3 个方向发展。

除了 Linux 内核本身可提供免费下载以外，一些厂商封装了 Linux 内核和大量有用的软件包，制定了相应的 Linux 发布版，如 Red Hat Linux、TurboLinux、Debian、SuSe、Ubuntu，国内的 RedFlag 和 xteam 等。

再者，针对嵌入式系统的应用，一些改进内核的 Linux 被开发出来，如改进实时性的 Hard Hat Linux 和 RTLinux、支持不含 MMU CPU 的μClinux（目前 Linux mainline 已经支持 MMU-less 系统）、面向数字相机和 MP3 等微型嵌入式设备的 ThinLinux 和以及颇有商业背景的 MontaVista 等。

3.2 Linux 2.6 内核的特点

本书基于的是 Linux 2.6 内核，LDD6410 开发板内核的完整版本号为 2.6.28.6。Linux 2.6 内核是 Linux 开发者群落一个寄予厚望的版本，从 2003 年 12 月 Linux 2.6.0 发布至今，一直还处于开发之中，并还将稳定较长一段时间。Linux 2.6 相对于 Linux 2.4 有相当大的改进，主要体现在如下几个方面：

1．新的调度器

2.6 版本的 Linux 内核使用了新的进程调度算法，它在高负载的情况下执行得极其出色，并且当有很多处理器时也可以很好地扩展。

2．内核抢占

在 2.6 版本的 Linux 内核中，一个内核任务可以被抢占，从而提高系统的实时性。这样做最主要的优势在于，可以极大地增强系统的用户交互性，用户将会觉得鼠标单击和击键的事件得到了更快速的响应。

3．改进的线程模型

2.6 版本的 Linux 中线程操作速度得以提高，可以处理任意数目的线程，最大可以到 20 亿。

4．虚拟内存的变化

从虚拟内存的角度来看，新内核融合了 r-map（反向映射）技术，显著改善虚拟内存在一定程度负载下的性能。

5．文件系统

2.6 版内核增加了对日志文件系统功能的支持，解决了 2.4 版在这方面的不足。2.6 版内核在文件系统上的关键变化还包括对扩展属性及 Posix 标准访问控制的支持。ext2/ext3 作为大多数 Linux 系统缺省安装的文件系统，在 2.6 版内核中增加了对扩展属性的支持，可以给指定的文件在文件系统中嵌入元数据。

6．音频

新的 Linux 音频体系结构 ALSA（Advanced Linux Sound Architecture）取代了缺陷很多的旧的 OSS（Open Sound System）。新的声音体系结构支持 USB 音频和 MIDI 设备，并支持全双工重放等功能。

7．总线

SCSI/IDE 子系统经过大幅度的重写，解决和改善了以前的一些问题。比如 2.6 版内核可以直接通过 IDE 驱动程序来支持 IDE CD/RW 设备，而不必像以前一样要使用一个特别的 SCSI 模拟驱动程序。

8．电源管理

支持 ACPI（高级电源配置管理界面，Advanced Configuration and Power Interface），用于调整 CPU 在不同的负载下工作于不同的时钟频率以降低功耗。

9．联网和 IPSec

2.6 内核中加入了对 IPSec 的支持，删除了原来内核内置的 HTTP 服务器 khttpd，加入了对新的 NFSv4（网络文件系统）客户机/服务器的支持，并改进了对 IPv6 的支持。

10．用户界面层

2.6 内核重写了帧缓冲/控制台层，人机界面层还加入了对近乎所有接口设备的支持（从触摸屏到盲人用的设备和各种各样的鼠标）。

在设备驱动程序的方面，Linux 2.6 相对于 Linux 2.4 也有较大的改动，这主要表现在内核 API 中增加了不少新功能（例如内存池）、sysfs 文件系统、内核模块从.o 变为.ko、驱动模块编译方式、模块使用计数、模块加载和卸载函数的定义等方面。

3.3 Linux 内核的组成

3.3.1 Linux 内核源代码目录结构

本书范例程序所基于的 Linux 2.6.28.6 内核源代码包含如下目录。

- arch：包含和硬件体系结构相关的代码，每种平台占一个相应的目录，如 i386、arm、powerpc、mips 等。
- block：块设备驱动程序 I/O 调度。
- crypto：常用加密和散列算法（如 AES、SHA 等），还有一些压缩和 CRC 校验算法。
- Documentation：内核各部分的通用解释和注释。
- drivers：设备驱动程序，每个不同的驱动占用一个子目录，如 char、block、net、mtd、i2c 等。
- fs：支持的各种文件系统，如 EXT、FAT、NTFS、JFFS2 等。
- include：头文件，与系统相关的头文件被放置在 include/linux 子目录下。
- init：内核初始化代码。
- ipc：进程间通信的代码。
- kernel：内核的最核心部分，包括进程调度、定时器等，而和平台相关的一部分代码放在 arch/*/kernel 目录下。
- lib：库文件代码。
- mm：内存管理代码，和平台相关的一部分代码放在 arch/*/mm 目录下。
- net：网络相关代码，实现了各种常见的网络协议。
- scripts：用于配置内核的脚本文件。
- security：主要是一个 SELinux 的模块。
- sound：ALSA、OSS 音频设备的驱动核心代码和常用设备驱动。
- usr：实现了用于打包和压缩的 cpio 等。

3.3.2 Linux 内核的组成部分

如图 3.1 所示，Linux 内核主要由进程调度（SCHED）、内存管理（MM）、虚拟文件系统（VFS）、网络接口（NET）和进程间通信（IPC）5 个子系统组成。

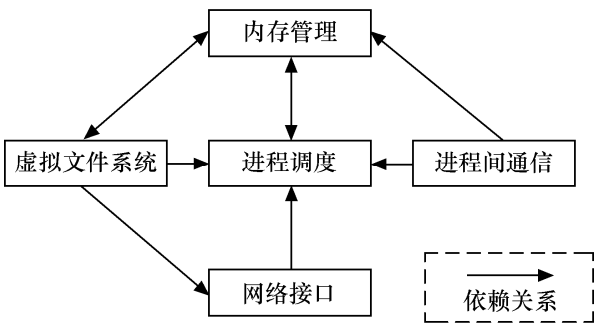

图 3.1 Linux 内核的组成部分与关系

1．进程调度

进程调度控制系统中的多个进程对 CPU 的访问，使得多个进程能在 CPU 中“微观串行，宏观并行”地执行。进程调度处于系统的中心位置，内核中其他的子系统都依赖它，因为每个子系统都需要挂起或恢复进程。

如图 3.2 所示，Linux 的进程在几个状态间进行切换。在设备驱动编程中，当请求的资源不能得到满足时，驱动一般会调度其他进程执行，并使本进程进入睡眠状态，直到它请求的资源被释放，才会被唤醒而进入就绪态。睡眠分成可被打断的睡眠和不可被打断的睡眠，两者的区别在于可被打断的睡眠在收到信号的时候会醒。

在设备驱动编程中，当请求的资源不能得到满足时，驱动一般会调度其他进程执行，其对应进程进入睡眠状态，直到它请求的资源被释放，才会被唤醒而进入就绪态。

设备驱动中，如果需要几个并发执行的任务，可以启动内核线程，启动内核线程的函数为：

```
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags);
```

2．内存管理

内存管理的主要作用是控制多个进程安全地共享主内存区域。当 CPU 提供内存管理单元（MMU）时，Linux 内存管理完成为每个进程进行虚拟内存到物理内存的转换。Linux 2.6 引入了对无 MMU CPU 的支持。

如图 3.3 所示，一般而言，Linux 的每个进程享有 4GB 的内存空间，0～3GB 属于用户空间，3～4GB 属于内核空间，内核空间对常规内存、I/O 设备内存以及高端内存存在不同的处理方式。

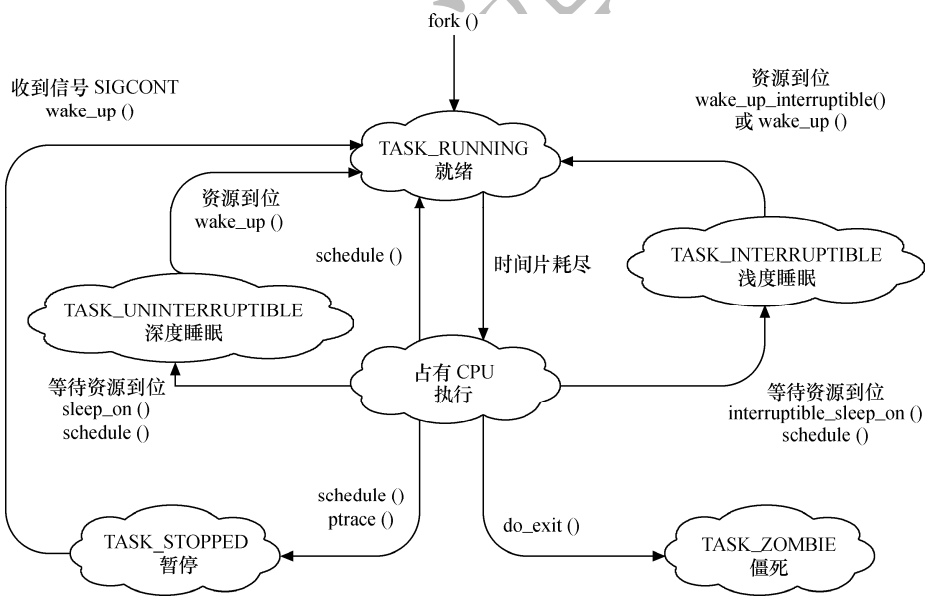

图 3.2　Linux 进程状态转换

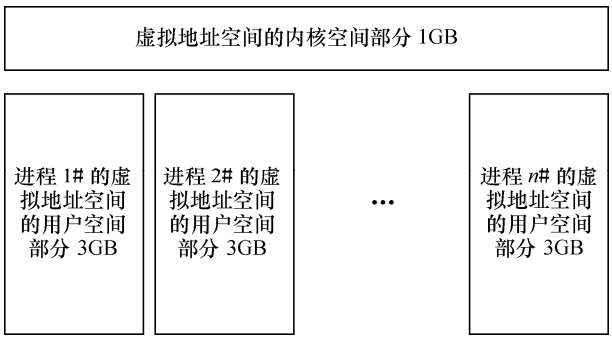

图 3.3　Linux 进程地址空间

3．虚拟文件系统

如图 3.4 所示，Linux 虚拟文件系统（VFS）隐藏各种了硬件的具体细节，为所有的设备提供了统一的接口。而且，它独立于各个具体的文件系统，是对各种文件系统的一个抽象，它使用超

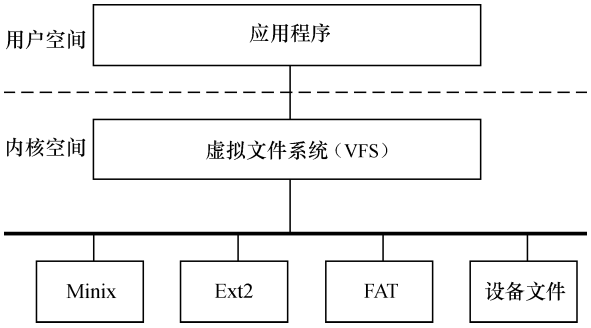

图 3.4　Linux 文件系统

级块 super block 存放文件系统相关信息，使用索引节点 inode 存放文件的物理信息，使用目录项 dentry 存放文件的逻辑信息。

4．网络接口

网络接口提供了对各种网络标准的存取和各种网络硬件的支持。如图 3.5 所示，在 Linux 中网络接口可分为网络协议和网络驱动程序，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通信，每一种可能的硬件设备都有相应的设备驱动程序。

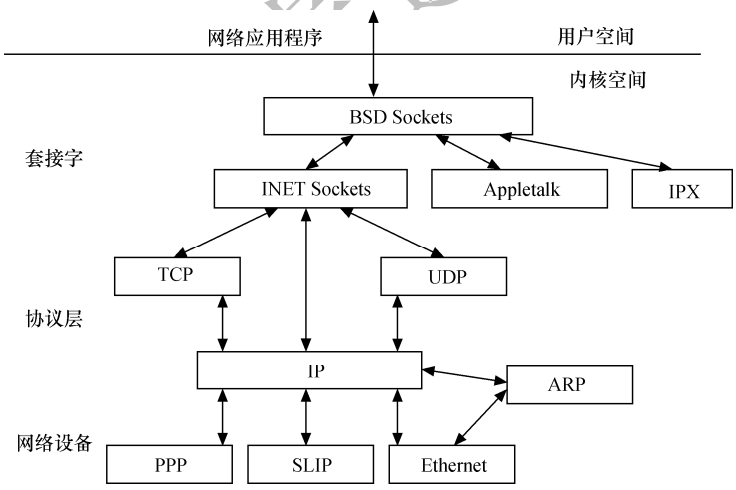

图 3.5　Linux 网络体系结构

5．进程通信

进程通信支持提供进程之间的通信，Linux 支持进程间的多种通信机制，包含信号量、共享内存、管道等，这些机制可协助多个进程、多资源的互斥访问、进程间的同步和消息传递。

Linux 内核的 5 个组成部分之间的依赖关系如下。

- 进程调度与内存管理之间的关系：这两个子系统互相依赖。在多道程序环境下，程序要运行必须为之创建进程，而创建进程的第一件事情，就是将程序和数据装入内存。
- 进程间通信与内存管理的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，这种机制允许两个进程除了拥有自己的私有空间，还可以存取共同的内存区域。

- 虚拟文件系统与网络接口之间的关系：虚拟文件系统利用网络接口支持网络文件系统(NFS)，也利用内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程（swapd）定期由调度程序调度，这也是内存管理依赖于进程调度的惟一原因。当一个进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前正在运行的进程。

除了这些依赖关系外，内核中的所有子系统还要依赖于一些共同的资源。这些资源包括所有子系统都用到的例程，如分配和释放内存空间的函数、打印警告或错误信息的函数及系统提供的调试例程等。

3.3.3 Linux 内核空间与用户空间

现代 CPU 内部往往实现了不同的操作模式（级别），不同的模式有不同的功能，高层程序往往不能访问低级功能，而必须以某种方式切换到低级模式。

例如，ARM 处理器分为 7 种工作模式。

- 用户模式（usr）：大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。
- 快速中断模式（fiq）：用于高速数据传输或通道处理。
- 外部中断模式（irq）：用于通用的中断处理。
- 管理模式（svc）：操作系统使用的保护模式。
- 数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM Linux 的系统调用实现原理是采用 swi 软中断从用户态 usr 模式陷入内核态 svc 模式。

又如，X86 处理器包含 4 个不同的特权级，称为 Ring 0～Ring 3。Ring0 下，可以执行特权级指令，对任何 I/O 设备都有访问权等，而 Ring3 则被限制很多操作。

Linux 系统充分利用 CPU 的这一硬件特性，但它只使用了两级。在 Linux 系统中，内核可进行任何操作，而应用程序则被禁止对硬件的直接访问和对内存的未授权访问。例如，若使用 X86 处理器，则用户代码运行在特权级 3，而系统内核代码则运行在特权级 0。

内核空间和用户空间这两个名词被用来区分程序执行的这两种不同状态，它们使用不同的地址空间。Linux 只能通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。

3.4 Linux 内核的编译及加载

3.4.1 Linux 内核的编译

Linux 驱动工程师需要牢固地掌握 Linux 内核的编译方法以为嵌入式系统构建可运行的 Linux 操作系统映像。在编译 LDD6410 的内核时，需要配置内核，可以使用下面命令中的　　一个：

```
#make config（基于文本的最为传统的配置界面，不推荐使用）
#make menuconfig（基于文本菜单的配置界面）
#make xconfig（要求 QT 被安装）
#make gconfig（要求 GTK+被安装）
```

在配置 Linux 2.6 内核所使用的 make config、make menuconfig、make xconfig 和 make gconfig 这 4 种方式中，最值得推荐的是 make menuconfig，它不依赖于 QT 或 GTK+，且非常直观，对 LDD6410 的 Linux 2.6.28 内核运行 make menuconfig 后的界面如图 3.6。

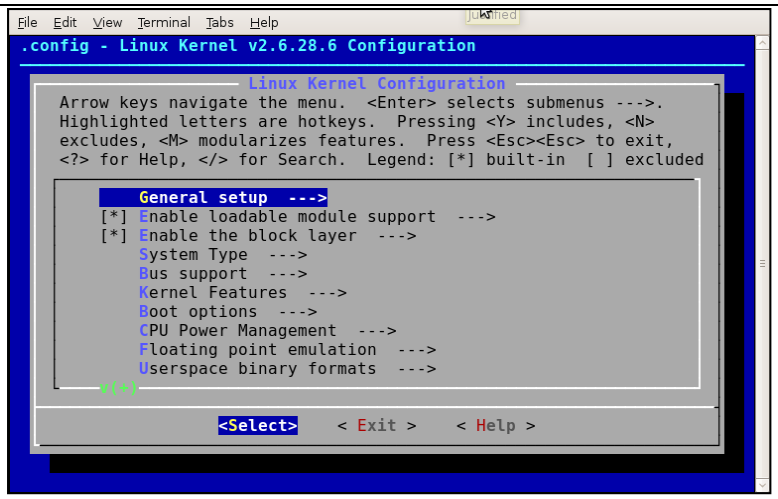

图 3.6 Linux 内核编译配置

内核配置包含的项目相当多，arch/arm/configs/ldd6410lcd_defconfig 文件包含了 LDD6410 的默认配置，因此，只需要运行 make ldd6410lcd_defconfig 就可以为 LDD6410 开发板配置内核。

编译内核和模块的方法是：

```
make zImage
make modules
```

执行完上述命令后，在源代码的根目录下会得到未压缩的内核映像 vmlinux 和内核符号表文件 System.map，在 arch/arm/boot/目录会得到压缩的内核映像 zImage，在内核各对应目录得到选中的内核模块。

Linux 2.6 内核的配置系统由以下 3 个部分组成。

- Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则。
- 配置文件（Kconfig）：给用户提供配置选择的功能。
- 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面和图形界面）。这些配置工具都是使用脚本语言，如 Tcl/TK、Perl 等编写。

使用 make config、make menuconfig 等命令后，会生成一个.config 配置文件，记录哪些部分被编译入内核、哪些部分被编译为内核模块。

运行 make menuconfig 等时，配置工具首先分析与体系结构对应的/arch/xxx/Kconfig 文件（xxx 即为传入的 ARCH 参数），/arch/xxx/Kconfig 文件中除本身包含一些与体系结构相关的配置项和配置菜单以外，还通过 source 语句引入了一系列 Kconfig 文件，而这些 Kconfig 又可能再次通过 source 引入下一层的 Kconfig，配置工具依据这些 Kconfig 包含的菜单和项目即可描绘出一个如图 3.6 所示的分层结构。例如，/arch/arm/Kconfig 文件的结构如下：

```
 mainmenu "Linux Kernel Configuration"

 config ARM
bool
default y
select HAVE_AOUT
select HAVE_IDE
select RTC_LIB
select SYS_SUPPORTS_APM_EMULATION
select HAVE_OPROFILE
select HAVE_ARCH_KGDB
select HAVE_KPROBES if (!XIP_KERNEL)
select HAVE_KRETPROBES if (HAVE_KPROBES)
select HAVE_FUNCTION_TRACER if (!XIP_KERNEL)
select HAVE_GENERIC_DMA_COHERENT
help
```

```
    The ARM series is a line of low-power-consumption RISC chip designs
    licensed by ARM Ltd and targeted at embedded applications and
    handhelds such as the Compaq IPAQ.  ARM-based PCs are no longer
    manufactured, but legacy ARM-based PC hardware remains popular in
    Europe.  There is an ARM Linux project with a web page at
    <http://www.arm.linux.org.uk/>.

...
config MMU
     bool
     default y

 ...
 config ARCH_S3C64XX
bool "Samsung S3C64XX"
select GENERIC_GPIO
select HAVE_CLK
help
    Samsung S3C64XX series based systems
...
if ARCH_S3C64XX
source "arch/arm/mach-s3c6400/Kconfig"
source "arch/arm/mach-s3c6410/Kconfig"
endif

...
```

3.4.2 Kconfig 和 Makefile

在 Linux 内核中增加程序需要完成以下 3 项工作。

- 将编写的源代码拷入 Linux 内核源代码的相应目录。
- 在目录的 Kconfig 文件中增加关于新源代码对应项目的编译配置选项。
- 在目录的 Makefile 文件中增加对新源代码的编译条目。

1．实例引导：S3C6410 处理器的 RTC 驱动配置

在讲解 Kconfig 和 Makefile 的语法之前，我们先利用两个简单的实例引导读者建立初步的认识。

首先，在 linux-2.6.28-samsung/drivers/rtc 目录中包含了 S3C6410 处理器的 RTC 设备驱动源代码 rtc-s3c.c。而在该目录的 Kconfig 文件中包含关于 RTC_DRV_S3C 的配置项目：

```
config RTC_DRV_S3C
 tristate "Samsung S3C series SoC RTC"
 depends on ARCH_S3C2410 || ARCH_S3C64XX || ARCH_S5PC1XX || ARCH_S5P64XX
 help
    RTC (Realtime Clock) driver for the clock inbuilt into the
    Samsung S3C24XX series of SoCs. This can provide periodic
    interrupt rates from 1Hz to 64Hz for user programs, and
    wakeup from Alarm.

    The driver currently supports the common features on all the
    S3C24XX range, such as the S3C2410, S3C2412, S3C2413, S3C2440
```

```
        and S3C2442.
        This driver can also be build as a module. If so, the module
        will be called rtc-s3c.
```

上述 Kconfig 文件的这段脚本意味着只有在 ARCH_S3C2410、ARCH_S3C64XX、ARCH_ S5PC1XX 或 ARCH_S5P64XX 项目之一被配置的情况下，才会出现 RTC_DRV_S3C 配置项目，这个配置项目为三态（可编译入内核，可不编译，也可编译为内核模块，选项分别为“Y”、“N”和“M”），菜单上显示的字符串为“Samsung S3C series SoC RTC”，“help”后面的内容为帮助信息。图 3.7 显示了 RTC_DRV_S3C 菜单以及其 help 在运行 make menuconfig 时的情况。

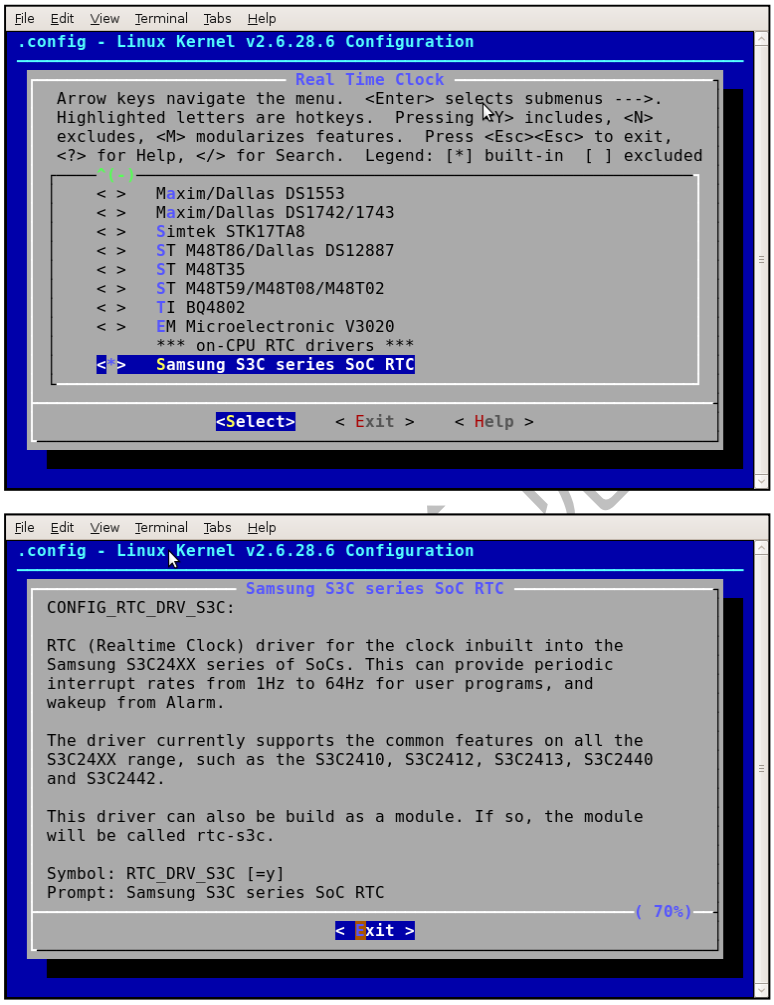

图 3.7 Kconfig 菜单项目与帮助信息

除了布尔型的配置项目外，还存在一种布尔型（bool）配置选项，它意味着要么编译入内核，要么不编译，选项为“Y”或“N”。

在目录的 Makefile 中关于 RTC_DRV_S3C 的编译脚本为：

```
obj-$(CONFIG_RTC_DRV_S3C)   += rtc-s3c.o
```

上述脚本意味着如果 RTC_DRV_S3C 配置选项被选择为“Y”或“M”，即 obj-$(CONFIG_RTC_ DRV_S3C)等同于 obj-y 或 obj-m 时，则编译 rtc-s3c.c，选“Y”的情况直接会将生成的目标代码直接连接到内核，为“M”的情况则会生成模块 rtc-s3c.ko；如果 RTC_DRV_S3C 配置选项被选择为“N”，即 obj-$(CONFIG_RTC_DRV_S3C)等同于 obj-n 时，则不编译 rtc-s3c.c。

一般而言，驱动工程师只会在内核源代码的 drivers 目录的相应子目录中增加新设备驱动的源代码，并增加或修改 Kconfig 配置脚本和 Makefile 脚本，完全仿照上述过程执行即可。

2. Makefile

这里主要对内核源代码各级子目录中的 kbuild（内核的编译系统）Makefile 进行简单介绍，这部分是内核模块或设备驱动的开发者最常接触到的。

Makefile 的语法包括如下几个方面。

（1）目标定义。

目标定义就是用来定义哪些内容要作为模块编译，哪些要编译并连接进内核。

例如：

```
obj-y += foo.o
```

表示要由 foo.c 或者 foo.s 文件编译得到 foo.o 并连接进内核，而 obj-m 则表示该文件要作为模块编译。除了 y、m 以外的 obj-x 形式的目标都不会被编译。

而更常见的做法是根据.config 文件的 CONFIG_变量来决定文件的编译方式，如：

```
obj-$(CONFIG_ISDN) += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

除了 obj-形式的目标以外，还有 lib-y library 库，hostprogs-y 主机程序等目标，但是基本都应用在特定的目录和场合下。

（2）多文件模块的定义。

最简单的 Makefile 如上一节一句话的形式就够了，如果一个模块由多个文件组成，会稍微复杂一些，这时候应采用模块名加-y 或-objs 后缀的形式来定义模块的组成文件。如以下例子：

```
#
# Makefile for the linux ext2-filesystem routines.
#
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o dir.o file.o fsync.o ialloc.o inode.o \
          ioctl.o namei.o super.o symlink.o
ext2-$(CONFIG_EXT2_FS_XATTR)     += xattr.o xattr_user.o xattr_trusted.o
ext2-$(CONFIG_EXT2_FS_POSIX_ACL) += acl.o
ext2-$(CONFIG_EXT2_FS_SECURITY)  += xattr_security.o
ext2-$(CONFIG_EXT2_FS_XIP)   += xip.o
```

模块的名字为 ext2，由 balloc.o、dir.o、file.o 等多个目标文件最终链接生成 ext2.o 直至 ext2.ko 文件，并且是否包括 xattr.o、acl.o 等则取决于内核配置文件的配置情况，例如，如果 CONFIG_ EXT2_FS_POSIX_ACL 被选择，则编译 acl.c 得到 acl.o 并最终链接进 ext2。

（3）目录层次的迭代。

如下例：

```
obj-$(CONFIG_EXT2_FS) += ext2/
```

当 CONFIG_EXT2_FS 的值为 y 或 m 时，kbuild 将会把 ext2 目录列入向下迭代的目标中。

3．Kconfig

内核配置脚本文件的语法也比较简单，主要包括如下几个方面。

（1）菜单入口。

大多数的内核配置选项都对应 Kconfig 中的一个菜单入口：

```
config MODVERSIONS
      bool "Module versioning support"
      help
         Usually, you have to use modules compiled with your kernel.
         Saying Y here makes it ...
```

“config”关键字定义新的配置选项，之后的几行定义了该配置选项的属性。配置选项的属性包括类型、数据范围、输入提示、依赖关系、选择关系及帮助信息和默认值等。

每个配置选项都必须指定类型，类型包括 bool、tristate、string、hex 和 int，其中 tristate 和 string 是两种基本的类型，其他类型都基于这两种基本类型。类型定义后可以紧跟输入提示，下面的两段脚本是等价的：

```
bool "Networking support"
```

和

```
bool
prompt "Networking support"
```

输入提示的一般格式为：

```
prompt <prompt> [if <expr>]
```

其中可选的 if 用来表示该提示的依赖关系。

默认值的格式为：

```
default <expr> [if <expr>]
```

一个配置选项可以存在任意多个默认值，这种情况下，只有第一个被定义的值是可用的。如果用户不设置对应的选项，配置选项的值就是默认值。

依赖关系的格式为：

```
depends on（或者 requires） <expr>
```

如果定义了多重依赖关系，它们之间用“&&”间隔。依赖关系也可以应用到该菜单中所有的其他选项(同样接受 if 表达式)，下面的两段脚本是等价的：

```
bool "foo" if BAR
default y if BAR
```

和

```
depends on BAR
bool "foo"
default y
```

选择关系（也称为反向依赖关系）的格式为：

```
select <symbol> [if <expr>]
```

A 如果选择了 B，则在 A 被选中的情况下，B 自动被选中。

kbuild Makefile 中的 expr（表达式）定义为：

```
<expr> ::= <symbol>
              <symbol> '=' <symbol>
              <symbol> '!=' <symbol>
             '(' <expr> ')'
             '!' <expr>
              <expr> '&&' <expr>
              <expr> '||' <expr>
```

也就是说 expr 是由 symbol、两个 symbol 相等、两个 symbol 不等以及 expr 的赋值、非、与或运算构成。而 symbol 分为两类，一类是由菜单入口定义配置选项定义的非常数 symbol，另一类是作为 expr 组成部分的常数 symbol。

数据范围的格式为：

```
range <symbol> <symbol> [if <expr>]
```

为 int 和 hex 类型的选项设置可以接受输入值范围，用户只能输入大于等于第一个 symbol，小于等于第二个 symbol 的值。

帮助信息的格式为：

```
help（或---help---）
   开始
   …
   结束
```

帮助信息完全靠文本缩进识别结束。“---help---”和“help”在作用上没有区别，设计“---help---”的初衷在于将文件中的配置逻辑与给开发人员的提示分开。

menuconfig 关键字的作用与 config 类似，但它在 config 的基础上要求所有的子选项作为独立的行显示。

（2）菜单结构。

菜单入口在菜单树结构中的位置可由两种方法决定。第一种方式为：

```
menu "Network device support"
    depends on NET
config NETDEVICES
    …
endmenu
```

所有处于“menu”和“endmenu”之间的菜单入口都会成为“Network device support”的子菜单。而且，所有子菜单选项都会继承父菜单的依赖关系，比如，“Network device support”对“NET”的依赖会被加到了配置选项 NETDEVICES 的依赖列表中。

注意 menu 后面跟的“Network device support”项目仅仅是 1 个菜单，没有对应真实的配置选项，也不具备 3 种不同的状态。这是它和 config 的区别。

另一种方式是通过分析依赖关系生成菜单结构。如果菜单选项在一定程度上依赖于前面的选项，它就能成为该选项的子菜单。如果父选项为“N”，子选项不可见；如果父选项可见，子选项才能可见。例如：

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES
comment "module support disabled"
    depends on !MODULES
```

MODVERSIONS 直接依赖 MODULES，只有 MODULES 不为“n”时，该选项才可见。

除此之外，Kconfig 中还可能使用“choices … endchoice”、“comment”、“if…endif”这样的语法结构。其中“choices … endchoice”的结构为：

```
choice
<choice options>
<choice block>
endchoice"
```

它定义一个选择群，其接受的选项（choice options）可以是前面描述的任何属性，例如 LDD6410 的 VGA 输出分辨率可以是 1 024×768 或者 800×600，在 drivers/video/samsung/Kconfig 就定义了如下的 choice：

```
choice
depends on FB_S3C_VGA
prompt "Select VGA Resolution for S3C Framebuffer"
default FB_S3C_VGA_1024_768
config FB_S3C_VGA_1024_768
     bool "1 024*768@60Hz"
     ---help---
     TBA
config FB_S3C_VGA_640_480
     bool "640*480@60Hz"
     ---help---
     TBA
endchoice
```

Kconfig 配置脚本和 Makefile 脚本编写的更详细信息，可以分别参看内核文档 Documentation 目录的 kbuild 子目录下的 Kconfig-language.txt 和 Makefiles.txt 文件。

4．应用实例：在内核中新增驱动代码目录和子目录

下面来看一个综合实例，假设我们要在内核源代码 drivers 目录下为 ARM 体系结构新增如下用于 test driver 的树型目录：

```
|--test
   |-- cpu
       | -- cpu.c
   |-- test.c
   |-- test_client.c
   |-- test_ioctl.c
   |-- test_proc.c
   |-- test_queue.c
```

在内核中增加目录和子目录，我们需为相应的新增目录创建 Makefile 和 Kconfig 文件，而新增目录的父目录中的 Kconfig 和 Makefile 也需修改，以便新增的 Kconfig 和 Makefile 能被引用。

在新增的 test 目录下，应该包含如下 Kconfig 文件：

```
#
# TEST driver configuration
#
menu "TEST Driver "
comment " TEST Driver"

config CONFIG_TEST
    bool "TEST support "

config CONFIG_TEST_USER
    tristate "TEST user-space interface"
    depends on CONFIG_TEST

endmenu
```

由于 test driver 对于内核来说是新的功能，所以需首先创建一个菜单 TEST Driver。然后，显示“TEST support”，等待用户选择；接下来判断用户是否选择了 TEST Driver，如果是（CONFIG_TEST=y），则进一步显示子功能：用户接口与 CPU 功能支持；由于用户接口功能可以被编译成内核模块，所以这里的询问语句使用了 tristate。

为了使这个 Kconfig 能起作用，修改 arch/arm/Kconfig 文件，增加：

```
source "drivers/test/Kconfig"
```

脚本中的 source 意味着引用新的 Kconfig 文件。

在新增的 test 目录下，应该包含如下 Makefile 文件：

```
# drivers/test/Makefile
#
# Makefile for the TEST.
#
obj-$(CONFIG_TEST) += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER) += test_ioctl.o
obj-$(CONFIG_PROC_FS) += test_proc.o

obj-$(CONFIG_TEST_CPU) += cpu/
```

该脚本根据配置变量的取值，构建 obj-*列表。由于 test 目录中包含一个子目录 cpu，当 CONFIG_TEST_CPU=y 时，需要将 cpu 目录加入列表。

test 目录中的 cpu 子目录也需包含如下的 Makefile：

```
# drivers/test/test/Makefile
#
# Makefile for the TEST CPU
#
obj-$(CONFIG_TEST_CPU) += cpu.o
```

为了使得整个 test 目录能够被编译命令作用到，test 目录父目录中的 Makefile 也需新增如下脚本：

```
obj-$(CONFIG_TEST) += test/
```

在 drivers/Makefile 中加入 obj-$(CONFIG_TEST) += test/，使得在用户在进行内核编译时能够进入 test 目录。

增加了 Kconfig 和 Makefile 之后的新的 test 树型目录为：

```
|--test
    |-- cpu
        | -- cpu.c
        | -- Makefile
    |-- test.c
    |-- test_client.c
    |-- test_ioctl.c
    |-- test_proc.c
    |-- test_queue.c
    |-- Makefile
    |-- Kconfig
```

3.4.3 Linux 内核的引导

引导 Linux 系统的过程包括很多阶段，这里将以引导 X86 PC 为例来进行讲解。引导 X86 PC 上的 Linux 的过程和引导嵌入式系统上的 Linux 的过程基本类似。不过在 X86 PC 上有一个从 BIOS（基本输入/输出系统）转移到 Bootloader 的过程，而嵌入式系统往往复位后就直接运行 Bootloader。

图 3.8 所示为 X86 PC 上从上电/复位到运行 Linux 用户空间初始进程的流程。在进入与 Linux 相关代码之间，会经历如下阶段。

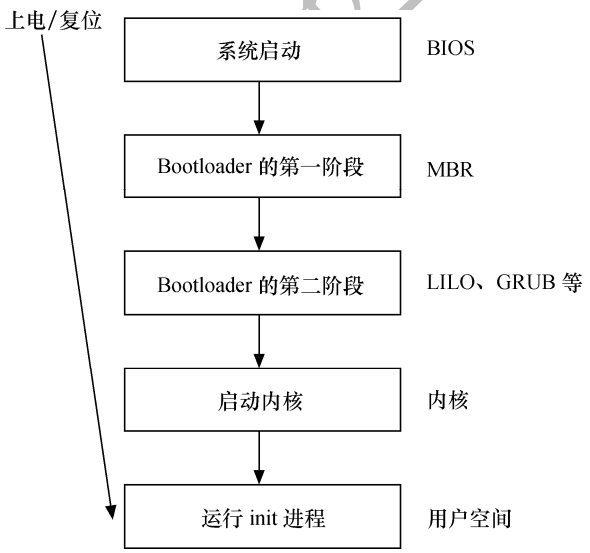

图 3.8 X86 PC 上的 Linux 引导流程

（1）当系统上电或复位时，CPU 会将 PC 指针赋值为一个特定的地址 0xFFFF0 并执行该地址处的指令。在 PC 机中，该地址位于 BIOS 中，它保存在主板上的 ROM 或 Flash 中。

（2）BIOS 运行时按照 CMOS 的设置定义的启动设备顺序来搜索处于活动状态并且可以引导的设备。若从硬盘启动，BIOS 会将硬盘 MBR（主引导记录）中的内容加载到 RAM。MBR 是一个 512 字节大小的扇区，位于磁盘上的第一个扇区中（0 道 0 柱面 1 扇区）。当 MBR 被加载到 RAM 中之后，BIOS 就会将控制权交给 MBR。

（3）主引导加载程序查找并加载次引导加载程序。它在分区表中查找活动分区，当找到一个活动分区时，扫描分区表中的其他分区，以确保它们都不是活动的。当这个过程验证完成之后，就将活动分区的引导记录从这个设备中读入 RAM 中并执行它。

（4）次引导加载程序加载 Linux 内核和可选的初始 RAM 磁盘，将控制权交给 Linux 内核源代码。

（5）运行被加载的内核，并启动用户空间应用程序。

嵌入式系统中 Linux 的引导过程与之类似，但一般更加简洁。不论具体以怎样的方式实现，只要具备如下特征就可以称其为 Bootloader。

- 可以在系统上电或复位的时候以某种方式执行，这些方式包括被 BIOS 引导执行、直接在 NOR Flash 中执行、NAND Flash 中的代码被 MCU 自动拷入内部或外部 RAM 执行等。
- 能将 U 盘、磁盘、光盘、NOR/NAND Flash、ROM、SD 卡等存储介质，甚或网口、串口中的操作系统加载到 RAM 并把控制权交给操作系统源代码执行。

完成上述功能的 Bootloader 的实现方式非常多样化，甚至本身也可以是一个简化版的操作系统。著名的 Linux Bootloader 包括应用于 PC 的 LILO 和 GRUB，应用于嵌入式系统的 U-Boot、RedBoot 等。

相比较于 LILO，GRUB 本身能理解 EXT2、EXT3 文件系统，因此可在文件系统中加载 Linux，而 LILO 只能识别“裸扇区”。

U-Boot 的定位为“Universal Bootloader”，其功能比较强大，涵盖了包括 PowerPC、ARM、MIPS 和 X86 在内的绝大部分处理器构架，提供网卡、串口、Flash 等外设驱动，提供必要的网络协议（BOOTP、DHCP、TFTP），能识别多种文件系统（cramfs、fat、jffs2 和 registerfs 等），并附带了调试、脚本、引导等工具，应用十分广泛。

Redboot 是 Redhat 公司随 eCos 发布的 Bootloader 开源项目，除了包含 U-Boot 类似的强大功能外，它还包含 GDB stub（插桩），因此能通过串口或网口与 GDB 进行通信，调试 GCC 产生的任何程序（包括内核）。

我们有必要对上述流程的第 5 个阶段进行更详细的分析，它完成启动内核并运行用户空间的 init 进程。

当内核映像被加载到 RAM 之后，Bootloader 的控制权被释放，内核阶段就开始了。内核映像并不是完全可直接执行的目标代码，而是一个压缩过的 zImage（小内核）或 bzImage（大内核，bzImage 中的 b 是“big”的意思）。

但是，并非 zImage 和 bzImage 映像中的一切都被压缩了，否则 Bootloader 把控制权交给这个内核映像它就“傻”了。实际上，映像中包含未被压缩的部分，这部分中包含解压缩程序，解压缩程序会解压映像中被压缩的部分。zImage 和 bzImage 都是用 gzip 压缩的，它们不仅是一个压缩文件，而且在这两个文件的开头部分内嵌有 gzip 解压缩代码。

如图 3.9 所示，当 bzImage（用于 i386 映像）被调用时，它从/arch/i386/boot/head.S 的 start 汇编例程开始执行。这个程序执行一些基本的硬件设置，并调用/arch/i386/boot/compressed/head.S 中的 startup_32 例程。startup_32 程序设置一些基本的运行环境（如堆栈）后，清除 BSS 段，调用/arch/ i386/boot/compressed/misc.c 中的 decompress_kernel() C 函数解压内核。内核被解压到内存中之后，会再调用/arch/i386/kernel/head.S 文件中的 startup_32 例程，这个新的 startup_32 例程（称为清除

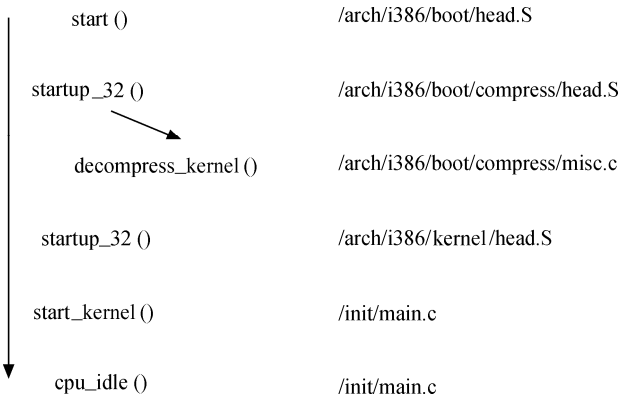

图 3.9　X86 PC 上的 Linux 内核初始化

程序或进程 0）会初始化页表，并启用内存分页机制，接着为任何可选的浮点单元（FPU）检测 CPU 的类型，并将其存储起来供以后使用。这些都做完之后，/init/main.c 中的 start_kernel()函数被调用，进入与体系结构无关的 Linux 内核部分。

start_kernel()会调用一系列初始化函数来设置中断，执行进一步的内存配置。之后，/arch/i386/kernel/process.c 中 kernel_thread()被调用以启动第一个核心线程，该线程执行 init()函数，而原执行序列会调用 cpu_idle()等待调度。

作为核心线程的 init()函数完成外设及其驱动程序的加载和初始化，挂接根文件系统。init()打开/dev/console 设备，重定向 stdin、stdout 和 stderr 到控制台。之后，它搜索文件系统中的 init 程序（也可以由“init=”命令行参数指定 init 程序），并使用 execve()系统调用执行 init 程序。搜索 init 程序的顺序为：/sbin/init、/etc/init、/bin/init 和/bin/sh。在嵌入式系统中，多数情况下，可以给内核传入一个简单的 shell 脚本来启动必需的嵌入式应用程序。

至此，漫长的 Linux 内核引导和启动过程就此结束，而 init()对应的这个由 start_kernel()创建的第一个线程也进入用户模式。

3.5 Linux 下的 C 编程特点

3.5.1 Linux 编码风格

Linux 程序的命名习惯和 Windows 程序的命名习惯及著名的匈牙利命名法有很大的不同。

在 Windows 程序中，习惯以如下方式命名宏、变量和函数：

```
#define PI 3.141 592 6  /*用大写字母代表宏*/
int minValue, maxValue; /*变量：第一个单词全写，其后的单词第一个字母小写*/
void SendData(void);  /*函数：所有单词第一个字母都大写定义*/
```

这种命名方式在程序员中非常盛行，意思表达清晰且避免了匈牙利法的臃肿，单词之间通过首字母大写来区分。通过第 1 个单词的首字母是否大写可以区分名称属于变量还是属于函数，而看到整串的大写字母可以断定为宏。实际上，Windows 的命名习惯并非仅限于 Windows 编程，大多数领域的程序开发都遵照此习惯。

但是 Linux 不以这种习惯命名，对应于上面的一段程序，在 Linux 中会被命名为：

```
#define PI 3.141 592 6
int min_value, max_value;
void send_data(void);
```

上述命名方式中，下划线大行其道，不依照 Windows 所采用的首字母大写以区分单词的方式。Linux 的命名习惯与 Windows 命名习惯各有千秋，但是既然本书和本书的读者立足于编写 Linux 程序，代码风格理应保持与 Linux 开发社区的一致性。

Linux 的代码缩进使用“TAB”（8 个字符）。

Linux 的代码括号“{”和“}”的使用原则如下。

（1）对于结构体、if/for/while/switch 语句，“{”不另起一行，例如：

```
struct var_data {
    int len;
    char data[0];
};

if (a == b) {
     a = c;
     d = a;
}
for (i = 0; i < 10; i++) {
     a = c;
     d = a;
```

```
}
```

（2）如果 if、for 循环后只有 1 行，不要加“{”和“}”，例如：

```
for (i = 0; i < 10; i++) {
      a = c;
}
```

应该改为：

```
for (i = 0; i < 10; i++)
      a = c;
```

（3）if 和 else 混用的情况下，else 语句不另起一行，例如：

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
   ...
}
```

（4）对于函数，“{”另起一行，譬如：

```
int add(int a, int b)
{
     return a + b;
}
```

在 switch/case 语句方面，Linux 建议 switch 和 case 对齐，例如：

```
switch (suffix) {
case 'G':
case 'g':
      mem <<= 30;
      break;
case 'M':
case 'm':
      mem <<= 20;
      break;
case 'K':
case 'k':
      mem <<= 10;
      /* fall through */
default:
      break;
}
```

内核下的 Documentation/CodingStyle 描述了 Linux 内核对编码风格的要求，内核下的 scripts/checkpatch.pl 提供了 1 个检查代码风格的脚本。如果我们使用 scripts/checkpatch.pl 检查包含如下代码块的源程序：

```
for (i = 0; i < 10; i++) {
      a = c;
}
```

就会产生“WARNING: braces {} are not necessary for single statement blocks”的警告。

另外，请注意代码中空格的应用，譬如“for␣(i␣=␣0;␣i␣<␣10;␣i++)␣{”语句中“␣”都是空格。

3.5.2 GNU C 与 ANSI C

Linux 上可用的 C 编译器是 GNU C 编译器，它建立在自由软件基金会的编程许可证的基础上，因此可以自由发布。GNU C 对标准 C 进行一系列扩展，以增强标准 C 的功能。

1．零长度和变量长度数组

GNU C 允许使用零长度数组，在定义变长对象的头结构时，这个特性非常有用。例如：

```
struct var_data {
    int len;
    char data[0];
};
```

char data[0]仅仅意味着程序中通过 var_data 结构体实例的 data[index]成员可以访问 len 之后的第 index 个地址，它并没有为 data[]数组分配内存，因此 sizeof(struct var_data)=sizeof(int)。

假设 struct var_data 的数据域就保存在 struct var_data 紧接着的内存区域，则通过如下代码可以遍历这些数据：

```
struct var_data s;
...
for (i = 0; i < s.len; i++)
    printf("%02x", s.data[i]);
```

GNU C 中也可以使用 1 个变量定义数组，例如如下代码中定义的“double x[n]”：

```
int main (int argc, char *argv[])
{
    int i, n = argc;
    double x[n];

    for (i = 0; i < n; i++)
      x[i] = i;
    return 0;
}
```

2．case 范围

GNU C 支持 case x…y 这样的语法，区间[x,y]的数都会满足这个 case 的条件，请看下面的代码：

```
switch (ch) {
case '0'... '9': c -= '0';
   break;
case 'a'... 'f': c -= 'a' - 10;
   break;
case 'A'... 'F': c -= 'A' - 10;
   break;
}
```

代码中的 case '0'... '9'等价于标准 C 中的：

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
```

3．语句表达式

GNU C 把包含在括号中的复合语句看做是一个表达式，称为语句表达式，它可以出现在任何允许表达式的地方。我们可以在语句表达式中使用原本只能在复合语句中使用的循环、局部变量等，例如：

```
#define min_t(type,x,y) \
({ type __x = (x); type __y = (y); __x < __y ? __x: __y; })
int ia, ib, mini;
float fa, fb, minf;
mini = min_t(int, ia, ib);
minf = min_t(float, fa, fb);
```

因为重新定义了__xx 和__y 这两个局部变量，所以以上述方式定义的宏将不会有副作用。在标准 C 中，对应的如下宏则会产生副作用：

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

代码 min(++ia,++ib)会被展开为((++ia) < (++ib) ? (++ia): (++ib))，传入宏的“参数”被增加 2 次。

4．typeof 关键字

typeof(x)语句可以获得 x 的类型，因此，我们可以借助 typeof 重新定义 min 这个宏：

```
#define min(x,y) ({ \
 const typeof(x) _x = (x);       \
 const typeof(y) _y = (y);       \
 (void) (&_x == &_y);                \
 _x < _y ? _x : _y; })
```

我们不需要像 min_t(type,x,y)这个宏那样把 type 传入，因为通过 typeof(x)、typeof(y)可以获得 type。代码行(void) (&_x == &_y)的作用是检查_x 和_y 的类型是否一致。

5．可变参数宏

标准 C 就支持可变参数函数，意味着函数的参数是不固定的，例如 printf()函数的原型为：

```
int printf( const char *format [, argument]... );
```

而在 GNU C 中，宏也可以接受可变数目的参数，例如：

```
#define pr_debug(fmt,arg...) \
            printk(fmt,##arg)
```

这里 arg 表示其余的参数，可以是零个或多个，这些参数以及参数之间的逗号构成 arg 的值，在宏扩展时替换 arg，例如下列代码：

```
pr_debug("%s:%d",filename,line)
```

会被扩展为：

```
printk("%s:%d", filename, line)
```

使用“##”的原因是处理 arg 不代表任何参数的情况，这时候，前面的逗号就变得多余了。使用“##”之后，GNU C 预处理器会丢弃前面的逗号，这样，代码：

```
pr_debug("success!\n")
```

会被正确地扩展为：

```
printk("success!\n")
```

而不是：

```
printk("success!\n",)
```

这正是我们希望看到的。

6．标号元素

标准 C 要求数组或结构体的初始化值必须以固定的顺序出现，在 GNU C 中，通过指定索引或结构体成员名，允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前添加“[INDEX] =”，当然也可以用“[FIRST ... LAST] =”的形式指定一个范围。例如，下面的代码定义一个数组，并把其中的所有元素赋值为 0：

```
unsigned char data[MAX] = { [0 ... MAX-1] = 0 };
```

下面的代码借助结构体成员名初始化结构体：

```
struct file_operations ext2_file_operations = {
    llseek: generic_file_llseek,
    read: generic_file_read,
    write: generic_file_write,
    ioctl: ext2_ioctl,
    mmap: generic_file_mmap,
```

```
    open: generic_file_open,
    release: ext2_release_file,
    fsync: ext2_sync_file,
};
```

但是，Linux 2.6 推荐类似的代码应该尽量采用标准 C 的方式：

```
struct file_operations ext2_file_operations = {
     .llseek     = generic_file_llseek,
     .read       = generic_file_read,
     .write      = generic_file_write,
     .aio_read   = generic_file_aio_read,
     .aio_write  = generic_file_aio_write,
     .ioctl      = ext2_ioctl,
     .mmap       = generic_file_mmap,
     .open       = generic_file_open,
     .release    = ext2_release_file,
     .fsync      = ext2_sync_file,
     .readv      = generic_file_readv,
     .writev     = generic_file_writev,
     .sendfile   = generic_file_sendfile,
};
```

7．当前函数名

GNU C 预定义了两个标志符保存当前函数的名字，__FUNCTION__保存函数在源码中的名字，__PRETTY_FUNCTION__保存带语言特色的名字。在 C 函数中，这两个名字是相同的。

```
void example()
{
   printf("This is function:%s", __FUNCTION__);
}
```

代码中的__FUNCTION__意味着字符串“example”。C99 已经支持__func__宏，因此建议在 Linux 编程中不再使用__FUNCTION__，而转而使用__func__：

```
void example()
{
   printf("This is function:%s", __func__);
}
```

8．特殊属性声明

GNU C 允许声明函数、变量和类型的特殊属性，以便进行手工的代码优化和定制代码检查的方法。要指定一个声明的属性，只需要在声明后添加__attribute__ ((ATTRIBUTE))。其中 ATTRIBUTE 为属性说明，如果存在多个属性，则以逗号分隔。GNU C 支持 noreturn、format、section、aligned、packed 等十多个属性。

noreturn 属性作用于函数，表示该函数从不返回。这会让编译器优化代码，并消除不必要的警告信息。例如：

```
# define ATTRIB_NORET  __attribute__((noreturn)) ....
asmlinkage NORET_TYPE void do_exit(long error_code) ATTRIB_NORET;
```

format 属性也用于函数，表示该函数使用 printf、scanf 或 strftime 风格的参数，指定 format 属性可以让编译器根据格式串检查参数类型。例如：

```
asmlinkage int printk(const char * fmt, ...) __attribute__ ((format (printf, 1, 2)));
```

上述代码中的第 1 个参数是格式串，从第 2 个参数开始都会根据 printf()函数的格式串规则检查参数。

unused 属性作用于函数和变量，表示该函数或变量可能不会被用到，这个属性可以避免编译器产生警告信息。

aligned 属性用于变量、结构体或联合体，指定变量、结构体或联合体的对界方式，以字节为单位，例如：

```
struct example_struct {
    char a;
    int b;
    long c;
} __attribute__((aligned(4)));
```

表示该结构类型的变量以 4 字节对界。

packed 属性作用于变量和类型，用于变量或结构体成员时表示使用最小可能的对界，用于枚举、结构体或联合体类型时表示该类型使用最小的内存。例如：

```
struct example_struct {
    char a;
    int b;
    long c  __attribute__((packed));
};
```

编译器对结构体成员及变量对界的目的是为了更快地访问结构体成员及变量占据的内存。例如，对于一个 32 位的整型变量，若以 4 字节方式存放（即低两位地址为 00），则 CPU 在一个总线周期内就可以读取 32 位；若不然，CPU 需要两次总线周期才能组合为一个 32 位整型。

9．内建函数

GNU C 提供了大量的内建函数，其中大部分是标准 C 库函数的 GNU C 编译器内建版本，例如 memcpy()等，它们与对应的标准 C 库函数功能相同。

不属于库函数的其他内建函数的命名通常以__builtin 开始，如下所示。

- 内建函数__builtin_return_address (LEVEL)返回当前函数或其调用者的返回地址，参数 LEVEL 指定调用栈的级数，如 0 表示当前函数的返回地址，1 表示当前函数的调用者的返回地址。
- 内建函数__builtin_constant_p(EXP)用于判断一个值是否为编译时常数，如果参数 EXP 的值是常数，函数返回 1，否则返回 0。
- 内建函数__builtin_expect(EXP, C)用于为编译器提供分支预测信息，其返回值是整数表达式 EXP 的值，C 的值必须是编译时常数。

例如，下面的代码检测第 1 个参数是否为编译时常数以确定采用参数版本还是非参数版本的代码：

```
#define test_bit(nr,addr) \
(__builtin_constant_p(nr) ? \
constant_test_bit((nr),(addr)) : \
variable_test_bit((nr),(addr)))
```

在使用 gcc 编译 C 程序的时候，如果使用“-ansi –pedantic”编译选项，则会告诉编译器不使用 GNU 扩展语法。例如对于如下 C 程序 test.c：

```
struct var_data {
   int len;
   char data[0];
};
struct var_data a;
```

直接编译可以通过：

```
gcc -c test.c
```

如果使用“-ansi –pedantic”编译选项，编译会报警：

```
gcc -ansi -pedantic -c test.c
test.c:3: warning: ISO C forbids zero-size array ‘data’
```

3.5.3 do { } while(0)

在 Linux 内核中，经常会看到 do {} while(0)这样的语句，许多人开始都会疑惑，认为 do {} while(0)毫无意义，因为它只会执行一次，加不加 do {} while(0)效果是完全一样的，其实 do {} while(0)的用法主要用于宏定义中。

这里用一个简单点的宏来演示：

```
#define SAFE_FREE(p) do{ free(p); p = NULL;} while(0)
```

假设这里去掉 do...while(0)，即定义 SAFE_DELETE 为：

```
#define SAFE_FREE(p) free(p); p = NULL;
```

那么以下代码

```
if(NULL != p)
   SAFE_DELETE(p)
else
   .../* do something */
```

会被展开为：

```
if(NULL != p)
   free(p); p = NULL;
else
   .../* do something */
```

展开的代码中存在两个问题。

（1）因为 if 分支后有两个语句，导致 else 分支没有对应的 if，编译失败。

（2）假设没有 else 分支，则 SAFE_FREE 中的第二个语句无论 if 测试是否通过，都会执行。

的确，将 SAFE_FREE 的定义加上{}就可以解决上述问题了，即：

```
#define SAFE_FREE(p) { free(p); p = NULL;}
```

这样，代码：

```
if(NULL != p)
   SAFE_DELETE(p)
else
   ... /* do something */
```

会被展开为：

```
if(NULL != p)
  { free(p); p = NULL; }
else
   ... /* do something */
```

但是，在 C 程序中，每个语句后面加分号是一种约定俗成的习惯，那么，如下代码：

```
if(NULL != p)
   SAFE_DELETE(p);
else
   ... /* do something */
```

将被扩展为：

```
if(NULL != p)
  { free(p); p = NULL; };
else
   ... /* do something */
```

这样，else 分支就又没有对应的 if 了，编译将无法通过。假设用了 do {} while(0)，情况就不一样了，同样的代码会被展开为：

```
if(NULL != p)
  do{ free(p); p = NULL;} while(0);
else
```

```
    ... /* do something */
```

不会再出现编译问题。do while(0)的使用完全是为了保证宏定义的使用者能无编译错误地使用宏，它不对其使用者做任何假设。

3.5.4 goto

用不用 goto 一直是一个著名的争议话题，Linux 内核源代码中对 goto 的应用非常广泛，但是一般只限于错误处理中，其结构如：

```
if(register_a()!=0)
     goto err;
if(register_b()!=0)
     goto err1;
if(register_c()!=0)
     goto err2;
if(register_d()!=0)
     goto err3;
...
err3:
   unregister_c();
err2:
   unregister_b();
err1:
   unregister_a();
err:
   return ret;
```

这种 goto 用于错误处理的用法实在是简单而高效，只需保证在错误处理时注销、资源释放等与正常的注册、资源申请顺序相反。

3.6 总结

本章主要讲解了 Linux 内核和 Linux 内核编程的基础知识，为进行 Linux 驱动开发打下软件基础。

在 Linux 内核方面，主要介绍了 Linux 内核的发展史、组成、特点、源代码结构、内核编译方法及内核引导过程。

由于 Linux 驱动编程本质属于内核编程，因此掌握内核编程的基础知识显得尤为重要。本章在这方面主要讲解了在内核中新增程序及目录和编写 Kconfig 和 Makefile 的方法，并分析了 Linux 下 C 编程习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067