第 3 章
Linux 内核及内核编程

本章导读

本章有助于读者打下 Linux 驱动编程的软件基础。由于 Linux 驱动编程的本质属于 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 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 内核的编译及内核的引导过程。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。

3.6 节讲解了 Linux 的工具链以及工具链对浮点的支持情况。

3.7 节介绍了公司或学校的实验室建设情况。

3.8 节介绍了 Linux 下的串口工具。

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，是一种类 UNIX 操作系统，诞生于 1991 年 10 月 5 日（第一次正式向外公布的时间），起初的作者是 Linus Torvalds。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

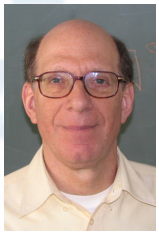
S. Tanenbaum 开发完成。有开放源代码的 Minix 系统的出现在全世界的大学中刮起了学习 UNIX 系统的旋风。Linux 刚开始就是参照 Minix 系统于 1991 年开发的。

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，严格来说，“Linux”应该称为“GNU/Linux”系统。

下面从左到右依次为前文所提到的 5 位大师 Linus Torvalds、Dennis Ritchie、Ken. Thompson、Andrew S. Tanenbaum、Richard M. Stallman。但愿我们能够追随大师的足迹，让自己不断地成长与进步。Linus Torvalds 的一番话甚为有道理：“Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.”技术成长的源动力应该是兴趣而非其他，只有兴趣才可以支撑一个人持续不断地十年如一日地努力与学习。Linus Torvalds 本人，虽然已经是一代大师，仍然在不断地管理和合并 Linux 内核的代码。这点，在国内浮躁的学术氛围之下，几乎是不可思议的。我想，中国梦至少包含每个码农都可以因为技术成长而得到人生出彩的机会。




4. POSIX 标准

POSIX（Portable Operating System Interface，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，用于保证编写的应用程序可以在源代码级上在多种操作系统中移植。该标准在推动 Linux 操作系统朝着正规化发展，是 Linux 前进的灯塔。

5. 互联网

如果没有互联网，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 Linux 0.13（0.95）版本的水平。从 Linux 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 以及其他维护者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 操作系统版本的历史及特点

版　本	时　间	特　点
Linux 0.1	1991 年 10 月	最初的原型
Linux 1.0	1994 年 3 月	包含了 386 的官方支持，仅支持单 CPU 系统
Linux 1.2	1995 年 3 月	第一个包含多平台（Alpha、Sparc、MIPS 等）支持的官方版本
Linux 2.0	1996 年 6 月	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
Linux 2.2	1999 年 1 月	极大提升 SMP 系统上 Linux 的性能，并支持更多的硬件
Linux 2.4	2001 年 1 月	进一步提升了 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
Linux 2.6.0 ~ 2.6.39	2003 年 12 月 ~ 2011 年 5 月	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无 MMU 系统的支持。同样，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
Linux 3.0 ~ 3.19、Linux 4.0-rc1 至今	2011 年 7 月至今	开发热点聚焦于虚拟化、新文件系统、Android、新体系结构支持以及性能优化等

Linux 内核通常以 2 ~ 3 个月为周期更新一次大的版本号，如 Linux 2.6.34 是在 2010 年 5 月发布的，Linux 2.6.35 的发布时间则为 2010 年 8 月。Linux 2.6 的最后一个版本是 Linux 2.6.39，之后 Linux 内核过渡到 Linux 3.0 版本，同样以 2 ~ 3 个月为周期更新小数点后第一位。因此，内核 Linux 3.x 时代，Linux 3 和 Linux 2.6 的地位对等，因此，Linux 2.6 时代的版本变更是 Linux 2.6.*N* ~ 2.6.*N*+1 以 2 ~ 3 个月为周期递进，而 Linux 3.x 时代后，则是 Linux 3.*N* ~ 3.*N*+1 以 2 ~ 3 个月为周期递进。Linux 3.x 的最后一个版本是 Linux 3.19。

在 Linux 内核版本发布后，还可以进行一个修复 bug 或者少量特性的反向移植（Backport，即把新版本中才有的补丁移植到已经发布的老版本中）的工作，这样的版本以小数点后最后一位的形式发布，如 Linux 2.6.35.1、Linux 2.6.35.2、Linux 3.10.1 和 Linux 3.10.2 等。此类已经发布的版本的维护版本通常是由 Greg Kroah-Hartman 等人进行管理的。Greg Kroah-Hartman 是名著 LDD3（《Linux 设备驱动（第 3 版）》的作者之一。

关于 Linux 内核从 Linux 2.6.39 变更为 Linux 3.0 的变化，按照 Linus Torvalds 的解释，并没有什么大的改变：“NOTHING. Absolutely nothing. Sure, we have the usual two thirds driver changes, and a lot of random fixes, but the point is that 3.0 is *just* about renumbering, we are very much *not* doing a KDE-4 or a Gnome-3 here. No breakage, no special scary new features, nothing at all like that.”因此，简单来说，版本号变更为“3.x”的原因就是“我喜欢”。

关于 Linux 内核每一个版本具体的变更，可以参考网页 http://kernelnewbies.org/LinuxVersions，比如 Linux 3.15 针对 Linux 3.14 的变更归纳在：http://kernelnewbies.org/Linux_3.15。

就在本书写作的过程中，2015 年 2 月 23 日，也迎来了 Linux 4.0-rc1 的诞生，而理由仍然是那么“无厘头”：

> .. after extensive statistical analysis of my G+ polling, I've come to
> the inescapable conclusion that internet polls are bad.
>
> Big surprise.
>
> But "Hurr durr I'ma sheep" trounced "I like online polls" by a 62-to-38%
> margin, in a poll that people weren't even supposed to participate in.
> Who can argue with solid numbers like that? 5,796 votes from people who
> can't even follow the most basic directions?
>
> In contrast, "v4.0" beat out "v3.20" by a slimmer margin of 56-to-44%,
> but with a total of 29,110 votes right now.
>
> Now, arguably, that vote spread is only about 3,200 votes, which is less
> than the almost six thousand votes that the "please ignore" poll got, so
> it could be considered noise.
>
> But hey, I asked, so I'll honor the votes.

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，提供更好的性能这 3 个方向发展。按照现在的状况，Linux 内核本身基本没有大的路线图，完全是根据使用 Linux 内核的企业和个人的需求，被相应的企业和个人开发出来并贡献给 Linux 产品线的。简单地说，Linux 内核是一个演变而不是一个设计。关于 Linux 的近期热点和走向，可以参考位于 http://www.linuxfoundation.org/news-media/lwf 的《Linux Weather Forecast》。

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、中间件、桌面环境和应用程序，制定了针对桌面 PC 和服务器的 Linux 发行版（Distro），如 Ubuntu、Red Hat、Fedora、Debian、SuSe、Gentoo 等，国内的红旗 Linux 开发商中科红旗则已经宣布倒闭。

再者，针对嵌入式系统的应用，一些集成和优化内核、开发工具、中间件和 UI 框架的嵌入式 Linux 被开发出来了，例如 MontaVista Linux、Mentor Embedded Linux、MeeGo、Tizen、Firefox OS 等。

Android 采用 Linux 内核，但是在内核里加入了一系列补丁，如 Binder、ashmem、wakelock、low memory killer、RAM_CONSOLE 等，目前，这些补丁中的绝大多数已经进入

Linux 的产品线。

图 3.1 显示了 Linux 2.6.13 以来每个内核版本参与的人、组织的情况以及每次版本演进的时候被改变的代码行数和补丁的数量。目前每次版本升级，都有分布于 200 多个组织超过 1000 人提交代码，被改变的代码行数超过 100 万行，补丁数量达 1 万个。

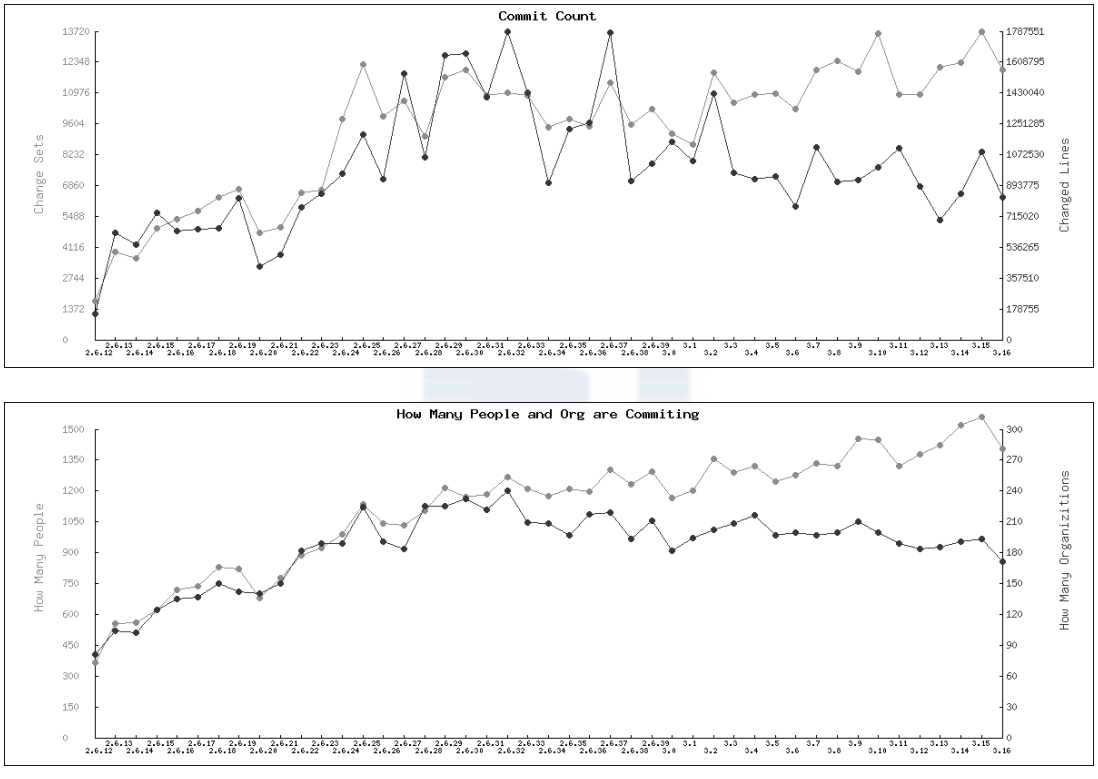


图 3.1　Linux 内核开发人员和补丁情况

3.2　Linux 2.6 后的内核特点

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 月直至 2011 年 7 月，内核重新进行了版本的编号，从而过渡到 Linux 3.x 版本直到成书时的 Linux 4.0-rc1。

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

Linux 2.6 以后版本的 Linux 内核使用了新的进程调度算法，它在高负载的情况下有极其出色的性能，并且当有很多处理器时也可以很好地扩展。在 Linux 内核 2.6 的早期采用了 O(1) 算法，之后转移到 CFS（Completely Fair Scheduler，完全公平调度）算法。在 Linux 3.14 中，也增加了一个新的调度类：SCHED_DEADLINE，它实现了 EDF（Earliest Deadline

First，最早截止期限优先）调度算法。

2. 内核抢占

在 Linux 2.6 以后版本的 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。Linux 2.6 以后的内核版本还是存在一些不可抢占的区间，如中断上下文、软中断上下文和自旋锁锁住的区间，如果给 Linux 内核打上 RT-Preempt 补丁，则中断和软中断都被线程化了，自旋锁也被互斥体替换，Linux 内核变得能支持硬实时。

如图 3.2 所示，左侧是 Linux 2.4，右侧是 Linux 2.6 以后的内核。在 Linux 2.4 的内核中，在 IRQ1 的中断服务程序唤醒 RT（实时）任务后，必须要等待前面一个 Normal（普通）任务的系统调用完成，返回用户空间的时候，RT 任务才能切入；而在 Linux 2.6 内核中，Normal 任务的关键部分（如自旋锁）结束的时候，RT 任务就从内核切入了。不过也可以看出，Linux 2.6 以后的内核仍然存在中断、软中断、自旋锁等原子上下文进程无法抢占执行的情况，这是 Linux 内核本身只提供软实时能力的原因。

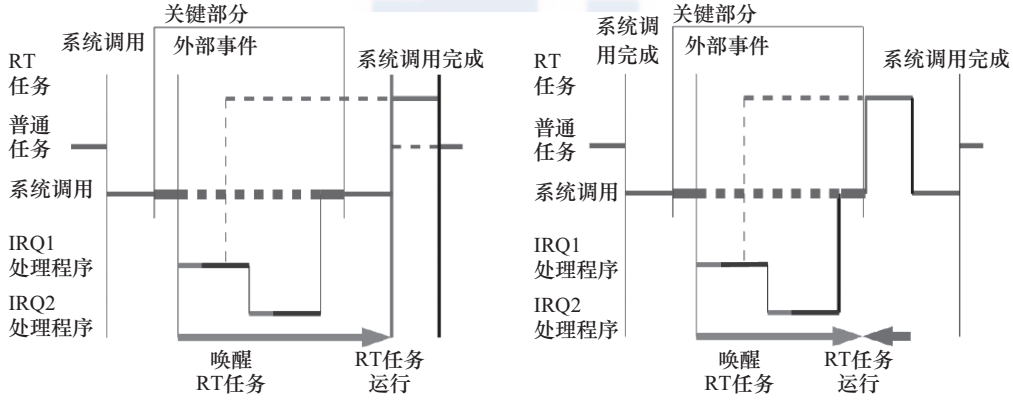


图 3.2　Linux 2.4 和 2.6 以后的内核在抢占上的区别

3. 改进的线程模型

Linux 2.6 以后版本中的线程采用 NPTL（Native POSIX Thread Library，本地 POSIX 线程库）模型，操作速度得以极大提高，相比于 Linux 2.4 内核时代的 LinuxThreads 模型，它也更加遵循 POSIX 规范的要求。NPTL 没有使用 LinuxThreads 模型中采用的管理线程，内核本身也增加了 FUTEX（Fast Userspace Mutex，快速用户态互斥体），从而减小多线程的通信开销。

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大小负载下的性能。在 Linux 2.4 中，要回收页时，内核的做法是遍历每个进程的所有 PTE 以判断该 PTE 是否与该页建立了映射，如果建立了，则取消该映射，最后无 PTE 与该

页相关联后才回收该页。在 Linux 2.6 后，则建立反向映射，可以通过页结构体快速寻找到页面的映射。

5. 文件系统

Linux 2.6 版内核增加了对日志文件系统功能的支持，解决了 Linux 2.4 版本在这方面的不足。Linux 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/ext4 作为大多数 Linux 系统默认安装的文件系统，在 Linux 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。

在文件系统方面，当前的研究热点是基于 B 树的 Btrfs，Btrfs 称为是下一代 Linux 文件系统，它在扩展性、数据一致性、多设备管理和针对 SSD 的优化等方面都优于 ext4。

6. 音频

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

7. 总线、设备和驱动模型

在 Linux 2.6 以后的内核中，总线、设备、驱动三者之间因为一定的联系性而实现对设备的控制。总线是三者联系起来的基础，通过一种总线类型，将设备和驱动联系起来。总线类型中的 match() 函数用来匹配设备和驱动，当匹配操作完成之后就会执行驱动程序中的 probe() 函数。

8. 电源管理

支持高级配置和电源接口（Advanced Configuration and Power Interface，ACPI），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。目前，Linux 内核的电源管理（PM）相对比较完善了，包括 CPUFreq、CPUIdle、CPU 热插拔、设备运行时（runtime）PM、Linux 系统挂起到内存和挂起到硬盘等全套的支持，在 ARM 上的支持也较完备。

9. 联网和 IPSec

Linux 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

Linux 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。

11. Linux 3.0 后 ARM 架构的变更

Linus Torvalds 在 2011 年 3 月 17 日的 ARM Linux 邮件列表中宣称“this whole ARM thing is a f*cking pain in the ass”，这引发了 ARM Linux 社区的地震，随后 ARM 社区进行

了一系列重大修正。社区必须改变这种局面，于是 PowerPC 等其他体系结构下已经使用的 FDT（Flattened Device Tree）进入到了 ARM 社区的视野。

此外，ARM Linux 的代码在时钟、DMA、pinmux、计时器刻度等诸多方面都进行了优化和调整，也删除了 arch/arm/mach-xxx/include/mach 头文件目录，以至于 Linux 3.7 以后的内核可以支持多平台，即用同一份内核镜像运行于多家 SoC 公司的多个芯片，实现“一个 Linux 可适用于所有的 ARM 系统”。

3.3　Linux 内核的组成

3.3.1　Linux 内核源代码的目录结构

Linux 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、arm64、powerpc、mips 等。Linux 内核目前已经支持 30 种左右的体系结构。在 arch 目录下，存放的是各个平台以及各个平台的芯片对 Linux 内核进程调度、内存管理、中断等的支持，以及每个具体的 SoC 和电路板的板级支持代码。
- 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：所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放置在 include/linux 子目录下。
- init：内核初始化代码。著名的 start_kernel() 就位于 init/main.c 文件中。
- ipc：进程间通信的代码。
- kernel：内核最核心的部分，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：实现用于打包和压缩的 cpio 等。
- include：内核 API 级别头文件。

内核一般要做到 drivers 与 arch 的软件架构分离，驱动中不包含板级信息，让驱动跨平台。同时内核的通用部分（如 kernel、fs、ipc、net 等）则与具体的硬件（arch 和 drivers）剥离。

3.3.2 Linux 内核的组成部分

如图 3.3 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。

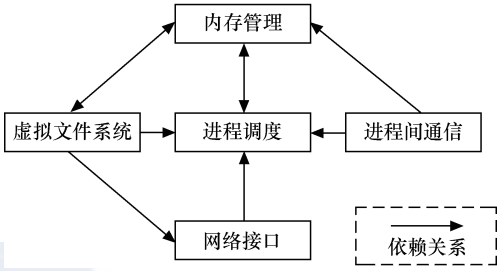


图 3.3 Linux 内核的组成部分与关系

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 3.4 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状态。睡眠分成可中断的睡眠和不可中断的睡眠，两者的区别在于可中断的睡眠在收到信号的时候会醒。

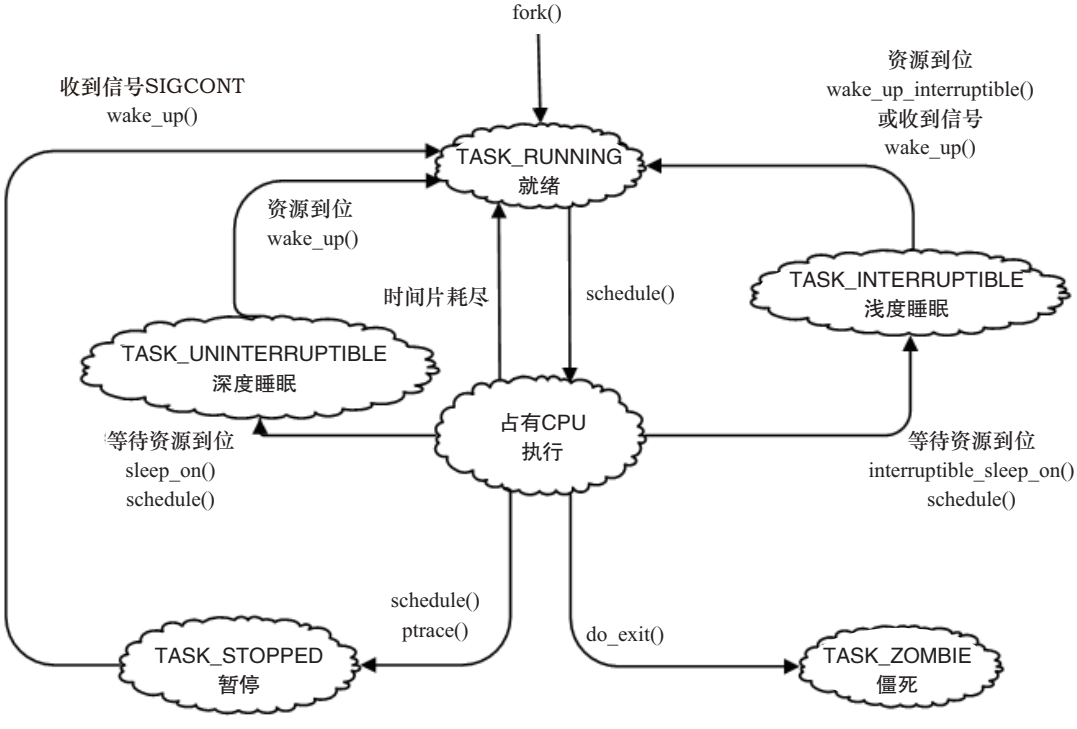


图 3.4 Linux 进程状态转换

完全处于 TASK_UNINTERRUPTIBLE 状态的进程甚至都无法被“杀死”，所以 Linux 2.6.26 之后的内核也存在一种 TASK_KILLABLE 的状态，它等于“TASK_WAKEKILL | TASK_UNINTERRUPTIBLE”，可以响应致命信号。

在 Linux 内核中，使用 task_struct 结构体来描述进程，该结构体中包含描述该进程内存资源、文件系统资源、文件资源、tty 资源、信号处理等的指针。Linux 的线程采用轻量级进程模型来实现，在用户空间通过 pthread_create() API 创建线程的时候，本质上内核只是创建了一个新的 task_struct，并将新 task_struct 的所有资源指针都指向创建它的那个 task_struct 的资源指针。

绝大多数进程（以及进程中的多个线程）是由用户空间的应用创建的，当它们存在底层资源和硬件访问的需求时，会通过系统调用进入内核空间。有时候，在内核编程中，如果需要几个并发执行的任务，可以启动内核线程，这些线程没有用户空间。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 内存管理对于每个进程完成从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 3.5 所示，一般而言，32 位处理器的 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有不同的处理方式。当然，内核空间和用户空间的具体界限是可以调整的，在内核配置选项 Kernel Features → Memory split 下，可以设置界限为 2GB 或者 3GB。

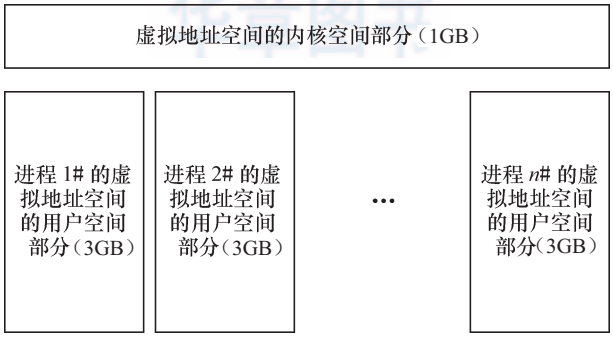


图 3.5　Linux 进程地址空间

如图 3.6 所示，Linux 内核的内存管理总体比较庞大，包含底层的 Buddy 算法，它用于管理每个页的占用情况，内核空间的 slab 以及用户空间的 C 库的二次管理。另外，内核也提供了页缓存的支持，用内存来缓存磁盘，per-BDI flusher 线程用于刷回脏的页缓存到磁盘。Kswapd（交换进程）则是 Linux 中用于页面回收（包括 file-backed 的页和匿名页）的内核

线程，它采用最近最少使用（LRU）算法进行内存回收。

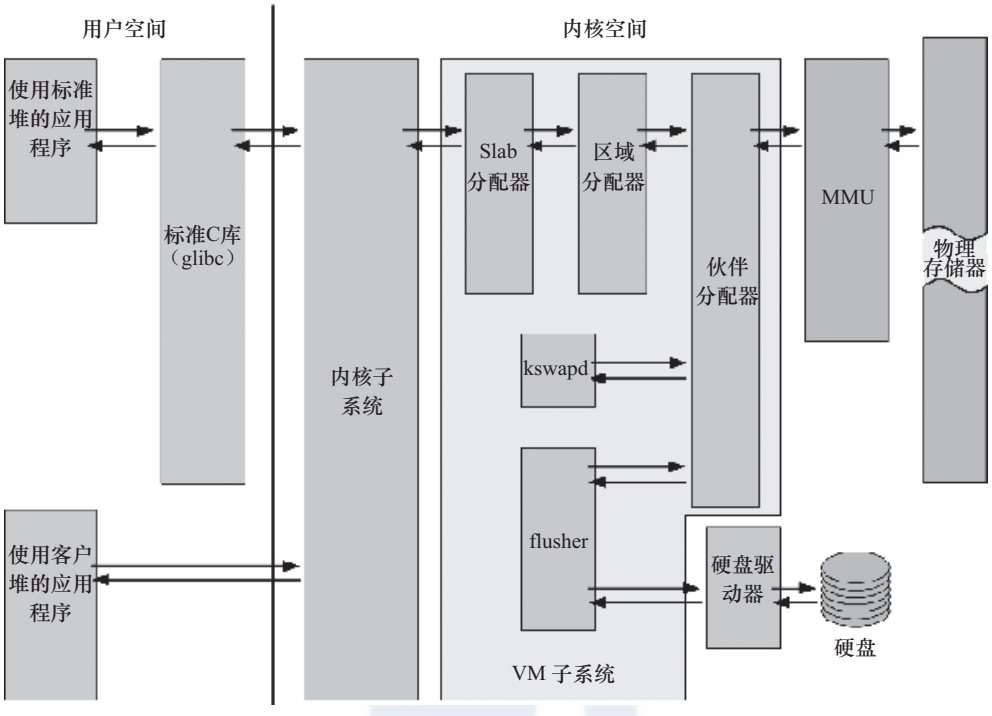


图 3.6　Linux 内存管理

3. 虚拟文件系统

如图 3.7 所示，Linux 虚拟文件系统隐藏了各种硬件的具体细节，为所有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。它为上层的应用程序提供了统一的 vfs_read()、vfs_write() 等接口，并调用具体底层文件系统或者设备驱动中实现的 file_operations 结构体的成员函数。

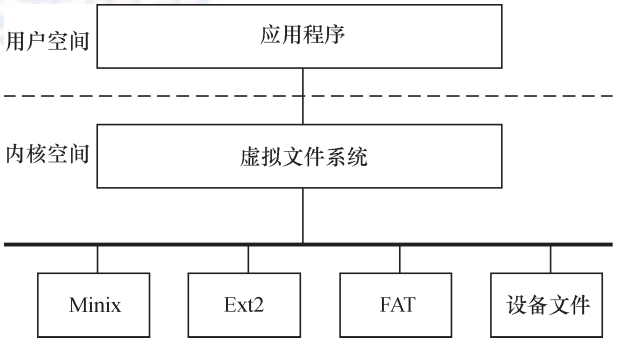


图 3.7　Linux 虚拟文件系统

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 3.8 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。

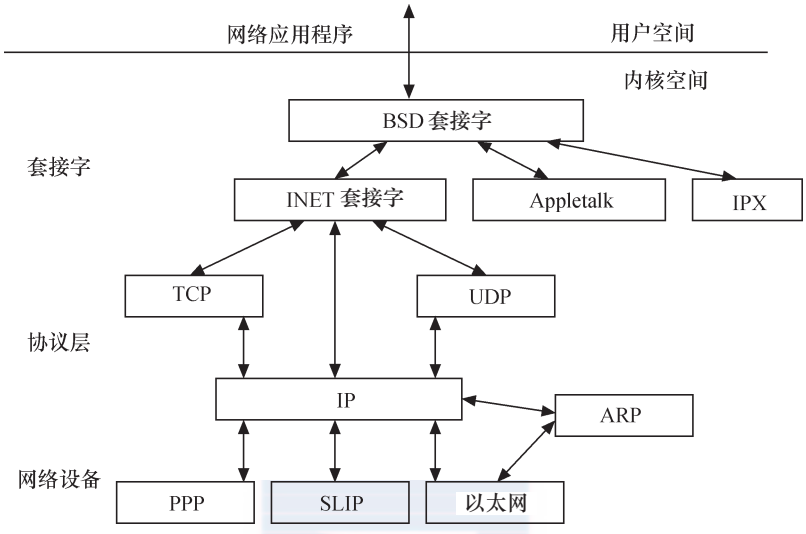


图 3.8　Linux 网络体系结构

Linux 内核支持的协议栈种类较多，如 Internet、UNIX、CAN、NFC、Bluetooth、WiMAX、IrDA 等，上层的应用程序统一使用套接字接口。

5. 进程间通信

进程间通信支持进程之间的通信，Linux 支持进程间的多种通信机制，包含信号量、共享内存、消息队列、管道、UNIX 域套接字等，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。在实际的 Linux 应用中，人们更多地趋向于使用 UNIX 域套接字，而不是 System V IPC 中的消息队列等机制。Android 内核则新增了 Binder 进程间通信方式。

Linux 内核 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。在多程序环境下，程序要运行，则必须为之创建进程，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之外，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，交换进程定期由调度程序调度，这也是内存管理依赖于进程调度的原因。当一个进程存取的内存映射被换出时，内存管理向虚拟文件系统发出请求，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 API，如分配和释放内存空间的函数、输出警告或错误消息的函数及系统提供的调试接口等。

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操作模式（级别），不同模式有不同功能，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。
- 数据访问中止模式（abt）：当数据或指令预取中止时进入该模式，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（usr）模式陷入管理模式（svc）。

又如，x86 处理器包含 4 个不同的特权级，称为 Ring 0 ~ Ring 3。在 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 系统可充分利用 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用来区分程序执行的两种不同状态，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 驱动开发者需要牢固地掌握 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）
```

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 Kconfig 包含的菜单和条目即可描绘出一个如图 3.9 所示的分层结构。

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。

- 将编写的源代码复制到 Linux 内核源代码的相应目录中。
- 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：TTY_PRINTK 字符设备

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对其建立对具初步的认识。

首先，在 drivers/char 目录中包含了 TTY_PRINTK 设备驱动的源代码 drivers/char/ttyprintk.c。而在该目录的 Kconfig 文件中包含关于 TTY_PRINTK 的配置项：

```
config TTY_PRINTK
       tristate "TTY driver to output user messages via printk"
       depends on EXPERT && TTY
       default n
       ---help---
        If you say Y here, the support for writing user messages (i.e.
        console messages) via printk is available.

        The feature is useful to inline user messages with kernel
        messages.
        In order to use this feature, you should output user messages
        to /dev/ttyprintk or redirect console to this TTY.

        If unsure, say N.
```

上述 Kconfig 文件的这段脚本意味着只有在 EXPERT 和 TTY 被配置的情况下，才会出现 TTY_PRINTK 配置项，这个配置项为三态（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“TTY driver to output user messages via printk”，“help”后面的内容为帮助信息。图 3.10 显示了 TTY_PRINTK 菜单以及 help 在运行 make menuconfig 时的情况。

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

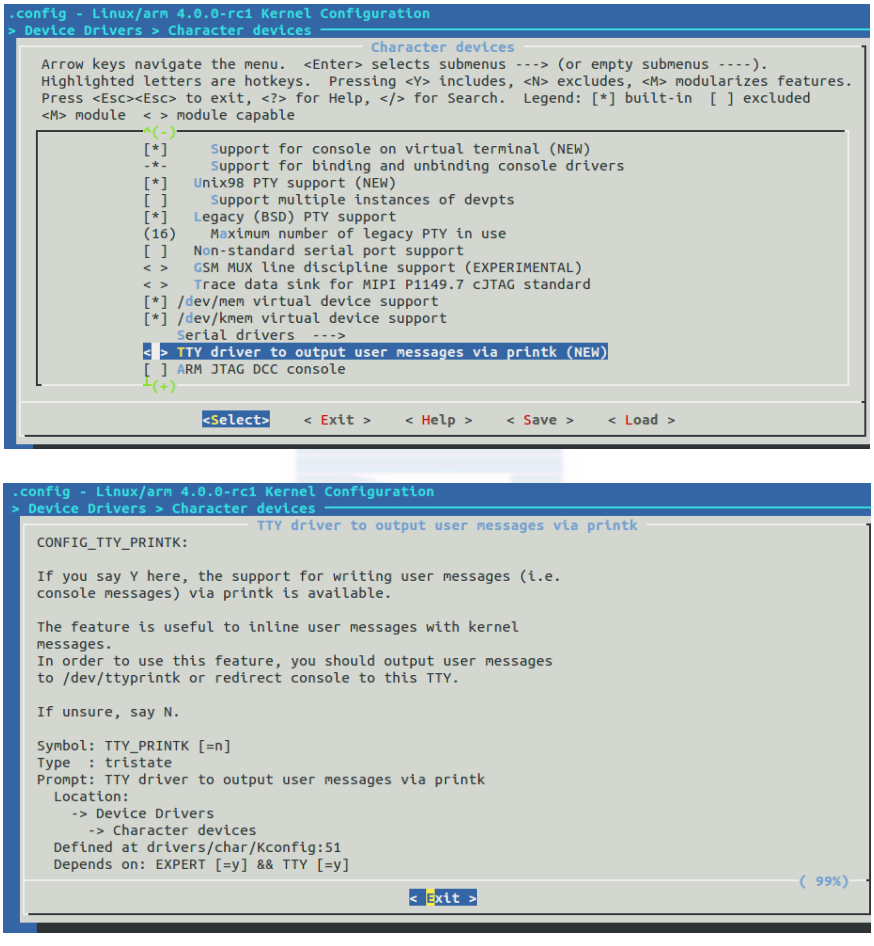


图 3.10　Kconfig 菜单项与 help 信息

在目录的 Makefile 中关于 TTY_PRINTK 的编译项为：

```
obj-$(CONFiG_TTY_PRINTK)          += ttyprintk.o
```

上述脚本意味着如果 TTY_PRINTK 配置选项被选择为“Y”或“M”，即 obj-$(CONFIG_TTY_PRINTK) 等同于 obj-y 或 obj-m，则编译 ttyprintk.c，选“Y”时会直接将生成的目标代码连接到内核，选“M”时则会生成模块 ttyprintk.ko；如果 TTY_PRINTK 配置选项被选择为“N”，即 obj-$(CONFIG_TTY_PRINTK) 等同于 obj-n，则不编译 ttyprintk.c。

一般而言，驱动开发者会在内核源代码的 drivers 目录内的相应子目录中增加新设备驱动的源代码或者在 arch/arm/mach-xxx 下新增加板级支持的代码，同时增加或修改 Kconfig 配置脚本和 Makefile 脚本，具体执行完全仿照上述过程即可。

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，哪些要编译并链接进内核。

例如：

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

表示要由 foo.c 或者 foo.s 文件编译得到 foo.o 并链接进内核（无条件编译，所以不需要 Kconfig 配置选项），而 obj-m 则表示该文件要作为模块编译。obj-n 形式的目标不会被编译。

更常见的做法是根据 make menuconfig 后生成的 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
```

除了具有 obj- 形式的目标以外，还有 lib-y library 库、hostprogs-y 主机程序等目标，但是这两类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 仅需一行代码就够了。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，如下：

```
#

# 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）配置选项

大多数内核配置选项都对应Kconfig中的一个配置选项（config）：

```
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，之后的几行代码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、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>]
```

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）内，下面两段脚本是等价的：

```
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。

- 数据范围的格式为：

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

- Kconfig 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。比如，SHDMA_R8A73A4 是一个布尔配置选项，表达式“ARCH_R8A73A4 && SH_DMAE != n”暗示只有当 ARCH_R8A73A4 被选中且 SH_DMAE 没有被选中的时候，才可能出现这个 SHDMA_R8A73A4。

```
config SHDMA_R8A73A4
       def_bool y
       depends on ARCH_R8A73A4 && SH_DMAE != n
```

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

（2）菜单结构

配置选项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”之间的配置选项都会成为“Network device support”的子菜单，而且，所有子菜单（config）选项都会继承父菜单（menu）的依赖关系，比如，“Network device support”对“NET”的依赖会被加到配置选项 NETDEVICES 的依赖列表中。

注意：menu 后面跟的“Network device support”项仅仅是 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 中就定义了如下 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 "1024*768@60Hz"
```

```
        ---help---
        TBA
config FB_S3C_VGA_640_480
        bool "640*480@60Hz"
        ---help---
        TBA
endchoice
```

上述例子中，prompt 配合 choice 起到提示作用。

用 Kconfig 配置脚本和 Makefile 脚本编写的更详细信息，可以分别参见内核文档 Documentation 目录内的 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 的树形目录：

```
|--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时，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，如果选择

了（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，因此当 CONFIG_TEST_CPU=y 时，需要将 cpu 目录加入列表中。

test 目录中的 cpu 子目录也需包含如下 Makefile：

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

为了使得编译命令作用到能够整个 test 目录，test 目录的父目录中 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 test 树形目录为：

```
|--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 ARM Linux 为例来进行讲解（见图 3.11）。一般的 SoC 内嵌入了 bootrom，上电时 bootrom 运行。对于 CPU0 而言，bootrom 会去引导 bootloader，而其他 CPU 则判断自己是不是 CPU0，进入 WFI 的状态等待 CPU0 来唤醒它。CPU0 引导 bootloader，bootloader 引导 Linux 内核，在内核启动阶段，CPU0 会发中断唤醒 CPU1，之后 CPU0 和 CPU1 都投入运行。CPU0 导致用户空间的 init 程序被调用，init 程序再派生其他进程，派生出来的进程再派生其他进程。CPU0 和 CPU1 共担这些负载，进行负载均衡。

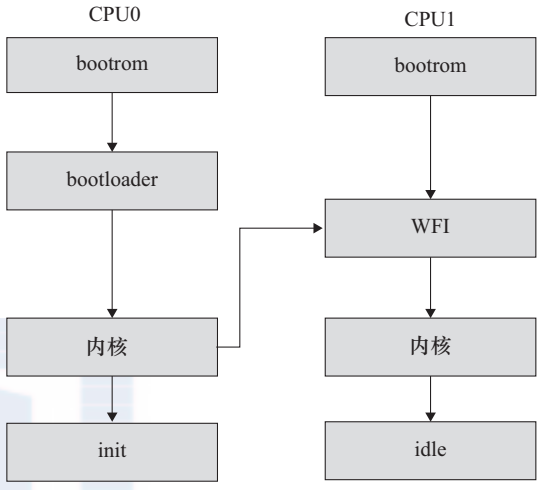


图 3.11 ARM 上的 Linux 引导流程

bootrom 是各个 SoC 厂家根据自身情况编写的，目前的 SoC 一般都具有从 SD、eMMC、NAND、USB 等介质启动的能力，这证明这些 bootrom 内部的代码具备读 SD、NAND 等能力。

嵌入式 Linux 领域最著名的 bootloader 是 U-Boot，其代码仓库位于 http://git.denx.de/u-boot.git/。早前，bootloader 需要将启动信息以 ATAG 的形式封装，并且把 ATAG 的地址填充在 r2 寄存器中，机型号填充在 r1 寄存器中，详见内核文档 Documentation/arm/booting。在 ARM Linux 支持设备树（Device Tree）后，bootloader 则需要把 dtb 的地址放入 r2 寄存器中。当然，ARM Linux 也支持直接把 dtb 和 zImage 绑定在一起的模式（内核 ARM_APPENDED_DTB 选项“Use appended device tree blob to zImage”），这样 r2 寄存器就不再需要填充 dtb 地址了。

类似 zImage 的内核镜像实际上是由没有压缩的解压算法和被压缩的内核组成，所以在 bootloader 跳入 zImage 以后，它自身的解压缩逻辑就把内核的镜像解压缩出来了。关于内核启动，与我们关系比较大的部分是每个平台的设备回调函数和设备属性信息，它们通常包装在 DT_MACHINE_START 和 MACHINE_END 之间，包含 reserve()、map_io()、init_machine()、init_late()、smp 等回调函数或者属性。这些回调函数会在内核启动过程中被调用。后续章节会进一步介绍。

用户空间的 init 程序常用的有 busybox init、SysVinit、systemd 等，它们的职责类似，把整个系统启动，最后形成一个进程树，比如 Ubuntu 上运行的 pstree:

```
init─┬─NetworkManager─┬─dhclient
     │                └─2*[{NetworkManager}]
     ├─VBoxSVC─┬─VirtualBox───29*[{VirtualBox}]
```

```
│           └─11*[{VBoxSVC}]
├─VBoxXPCOMIPCD
├─accounts-daemon ─── {accounts-daemon}
├─acpid
├─apache2 ─── 5*[apache2]
├─at-spi-bus-laun ─── 2*[{at-spi-bus-laun}]
├─atd
├─avahi-daemon ─── avahi-daemon
├─bluetoothd
├─cgrulesengd
├─colord ─── 2*[{colord}]
├─console-kit-dae ─── 64*[{console-kit-dae}]
├─cpufreqd ─── {cpufreqd}
├─cron
├─cupsd
├─2*[dbus-daemon]
├─dbus-launch
├─dconf-service ─── 2*[{dconf-service}]
├─dnsmasq
```

3.5 Linux 下的 C 编程特点

3.5.1 Linux 编码风格

Linux 有独特的编码风格，在内核源代码下存在一个文件 Documentation/CodingStyle，进行了比较详细的描述。

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 3.1415926      /* 用大写字母代表宏 */
int minValue, maxValue;   /* 变量：第一个单词全小写，其后单词的第一个字母大写 */
void SendData(void);      /* 函数：所有单词第一个字母都大写 */
```

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，许多领域的程序开发都遵照此习惯。

但是 Linux 不以这种习惯命名，对于上面的一段程序，在 Linux 中它会被命名为：

```
#define PI 3.1415926
int min_value, max_value;
void send_data(void);
```

在上述命名方式中，下划线大行其道，不按照 Windows 所采用的用首字母大写来区分单

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，代码风格理应与 Linux 开发社区保持一致。

Linux 的代码缩进使用“TAB”。

Linux 中代码括号“{”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++)␣{”语句中的“␣”都是空格。

在工程阶段，一般可以在 SCM 软件的服务器端使能 pre-commit hook，自动检查工程师提交的代码是否符合 Linux 的编码风格，如果不符合，则自动拦截。git 的 pre-commit hook 可以运行在本地代码仓库中，如 Ben Dooks 完成的一个版本：

```
#!/bin/sh
#
# pre-commit hook to run check-patch on the output and stop any commits
# that do not pass. Note, only for git-commit, and not for any of the
# other scenarios
#
# Copyright 2010 Ben Dooks, <ben-linux@fluff.org>

if git rev-parse --verify HEAD 2>/dev/null >/dev/null
then
    against=HEAD
else
    # Initial commit: diff against an empty tree object
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi

git diff --cached $against -- | ./scripts/checkpatch.pl --no-signoff -
```

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 紧接着的内存区域中，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] 中的数都会满足这个 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成是一个表达式，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，所以用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) 会展开为 ((++ia) < (++ib) ? (++ia): (++ib))，传入宏的“参数”增加两次。

4. typeof 关键字

typeof(x) 语句可以获得 x 的类型，因此，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) 那个宏那样把 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，可以有零个或多个参数，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，如下列代码：

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

会被扩展为：

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

使用“##”是为了处理 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，下列代码：

```
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，下面的代码定义了一个数组，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,
    .ioct        = 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 预定义了两个标识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(void)
{
   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，以便手动优化代码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齐方式，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 字节对齐。

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齐，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齐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；否则，CPU 需要两个总线周期才能读取 32 位。

9. 内建函数

GNU C 提供了大量内建函数，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。

例如，下面的代码可检测第 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：

```
#define test_bit(nr,addr) \
```

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

- 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。

Linux 内核编程时常用的 likely() 和 unlikely() 底层调用的 likely_notrace()、unlikely_notrace() 就是基于 __builtin_expect(EXP, C) 实现的。

```
#define likely_notrace(x)       __builtin_expect(!!(x), 1)
#define unlikely_notrace(x)     __builtin_expect(!!(x), 0)
```

若代码中出现分支，则即可能中断流水线，我们可以通过 likely() 和 unlikely() 暗示分支容易成立还是不容易成立，例如：

```
if (likely(!IN_DEV_ROUTE_LOCALNET(in_dev)))
    if (ipv4_is_loopback(saddr))
        goto e_inval;
```

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) 语句

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，在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) 语句，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 */
```

而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 语句

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;
```

这种将 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　工具链

在 Linux 的编程中，通常使用 GNU 工具链编译 Bootloader、内核和应用程序。 GNU 组织维护了 GCC、GDB、glibc、Binutils 等，分别见于 https://gcc.gnu.org/，https://www.gnu.org/software/gdb/, https://www.gnu.org/software/libc/、https://www.gnu.org/software/binutils/。

建立交叉工具链的过程相当烦琐，一般可以通过类似 crosstool-ng 这样的工具来做。

crosstool-ng 也采用了与内核相似的 menuconfig 配置方法。在官网 http://www.crosstool-ng.org/ 上下载 crosstool-ng 的源代码并编译安装后，运行 ct-ng menuconfig，会出现如图 3.12 的配置菜单。在里面我们可以选择目标机处理器型号，支持的内核版本号等。

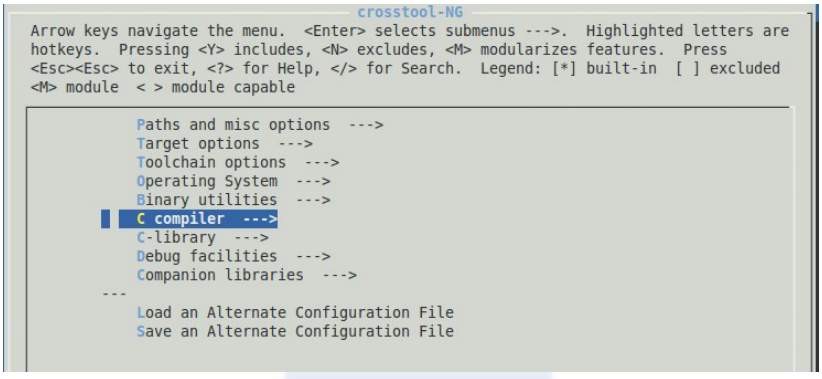


图 3.12 crosstool-ng 的配置菜单

当然，也可以直接下载第三方编译好的、开放的、针对目标处理器的交叉工具链，如在 http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/ 上可以下载针对 ARM、MIPS、高通 Hexagon、Altera Nios II、Intel、AMD64 等处理器的工具链，在 http://www.linaro.org/downloads/ 可以下载针对 ARM 的工具链。

目前，在 ARM Linux 的开发中，人们趋向于使用 Linaro（http://www.linaro.org/）工具链团队维护的 ARM 工具链，它以每月一次的形式发布新的版本，编译好的可执行文件可从网址 http://www.linaro.org/downloads/ 下载。Linaro 是 ARM Linux 领域中最著名最具技术成就的开源组织，其会员包括 ARM、Broadcom、Samsung、TI、Qualcomm 等，国内的海思、中兴、全志和中国台湾的 MediaTek 也是它的会员。

一个典型的 ARM Linux 工具链包含 arm-linux-gnueabihf-gcc（后续工具省略前缀）、strip、gcc、objdump、ld、gprof、nm、readelf、addr2line 等。用 strip 可以删除可执行文件中的符号表和调试信息等来实现缩减程序体积的目的。gprof 在编译过程中在函数入口处插入计数器以收集每个函数的被调用情况和被调用次数，检查程序计数器并在分析时找出与程序计数器对应的函数来统计函数占用的时间。objdump 是反汇编工具。nm 则用于显示关于对象文件、可执行文件以及对象文件库里的符号信息。其中，前缀中的“hf”显示该工具链是完全的硬浮点，由于目前主流的 ARM 芯片都自带 VFP 或者 NEON 等浮点处理单元（FPU），所以对硬浮点的需求就更加强烈。Linux 的浮点处理可以采用完全软浮点，也可以采用与软浮点兼容，但是使用 FPU 硬件的 softfp，以及完全硬浮点。具体的 ABI（Application Binary Interface，应用程序二进制接口）通过 -mfloat-abi= 参数指定，3 种情况下的参数分别是 -mfloat-abi=soft/softfp/hard。

在以前，主流的工具链采用“与软浮点兼容，但是使用 FPU 硬件的 softfp”。 softfp 使

用了硬件的 FPU，但是函数的参数仍然使用整型寄存器来传递，完全硬浮点则直接使用 FPU 的寄存器传递参数。

下面一段程序：

```
float mul(float a, float b)
{
        return a * b;
}
void main(void)
{
        printf("1.1 * 2.3 = %f\n", mul(1.1, 2.3));
}
```

对其使用 arm-linux-gnueabihf-gcc 编译并反汇编的结果是：

```
000 08394 <mul>:
     8394:  b480        push  {r7}
     8396:  b083        sub  sp, #12
     8398:  af00        add  r7, sp, #0
     839a:  ed87 0a01   vstr  s0, [r7, #4](null)
     839e:  edc7 0a00   vstr  s1, [r7]
     83a2:  ed97 7a01   vldr  s14, [r7, #4](null)
     83a6:  edd7 7a00   vldr  s15, [r7]
     83aa:  ee67 7a27   vmul.f32  s15, s14, s15
     83ae:  eeb0 0a67   vmov.f32  s0, s15
     83b2:  f107 070c   add.w  r7, r7, #12
     83b6:  46bd        mov  sp, r7
     83b8:  bc80        pop  {r7}
     83ba:  4770        bx       lr

0000 83bc <main>:
     83bc:  b580        push  {r7, lr}
     83be:  af00        add  r7, sp, #0
     83c0:  ed9f 0a09   vldr  s0, [pc, #36](null)   ; 83e8 <main+0x2c>
     83c4:  eddf 0a09   vldr  s1, [pc, #36](null)   ; 83ec <main+0x30>
     83c8:  f7ff ffe4   b8394 <mul>
     83cc:  eef0 7a40   vmov.f32  s15, s0
     83d0:  eeb7 7ae7   vcvt.f64.f32  d7, s15
     83d4:  f248 4044   movw  r0, #33860  ; 0x8444
     83d8:  f2c0 0000   movt  r0, #0
     83dc:  ec53 2b17   vmov  r2, r3, d7
     83e0:  f7ff ef82   blx  82e8 <_init+0x20>
     83e4:  bd80        pop  {r7, pc}
     83e6:  bf00        nop
```

而使用没有“hf”前缀的 arm-linux-gnueabi-gcc 编译并反汇编的结果则为

```
000 0838c <mul>:
     838c:  b480        push  {r7}
     838e:  b083        sub  sp, #12
```

```
      8390:  af00         add  r7, sp, #0
      8392:  6078         str  r0, [r7, #4]
      8394:  6039         str  r1, [r7, #0]
      8396:  ed97 7a01    vldr  s14, [r7, #4](null)
      839a:  edd7 7a00    vldr  s15, [r7]
      839e:  ee67 7a27    vmul.f32  s15, s14, s15
      83a2:  ee17 3a90    vmov  r3, s15
      83a6:  4618         mov  r0, r3
      83a8:  f107 070c    add.w  r7, r7, #12
      83ac:  46bd         mov  sp, r7
      83ae:  bc80         pop  {r7}
      83b0:  4770         bx       lr
      83b2:  bf00         nop

000 083b4 <main>:
      83b4:  b580         push  {r7, lr}
      83b6:  af00         add  r7, sp, #0
      83b8:  4808         ldr  r0, [pc, #32]  ; (83dc <main+0x28>)
      83ba:  4909         ldr  r1, [pc, #36]  ; (83e0 <main+0x2c>)
      83bc:  f7ff ffe6    bl  838c <mul>
      83c0:  ee07 0a90    vmov  s15, r0
      83c4:  eeb7 7ae7    vcvt.f64.f32  d7, s15
      83c8:  f248 4038    movwr0, #33848   ; 0x8438
      83cc:  f2c0 0000    movtr0, #0
      83d0:  ec53 2b17    vmovr2, r3, d7
      83d4:  f7ff ef84    blx  82e0 <_init+0x20>
      83d8:  bd80         pop  {r7, pc}
      83da:  bf00         nop
```

关注其中加粗的行，可以看出前面的汇编使用s0和s1传递参数，后者则仍然使用ARM的r0和r1。测试显示一个含有浮点运算的程序若使用hard ABI会比softfp ABI快5% ~ 40%，如果浮点负载重，结果可能会快200%以上。

3.7 实验室建设

在公司或学校的实验室中，PC的性能一般来说不会太高，用PC来编译Linux内核和模块的速度总会受限。相反，服务器的资源相对比较充分，CPU以及磁盘性能都较高，因此在服务器上进行内核、驱动及应用程序的编译开发将更加快捷，而且使用服务器更有利于统一管理实验室内的所有开发者。图3.13所示为一种常见的小型Linux实验室环境。

在Linux服务器上启动了Samba、NFS和sshd进程，各工程师在自己的Linux或Windows客户机上通过SSH用自己的用户名和密码登录服务器，便可以使用服务器上的GCC、GDB等软件。

在Windows下，常用的SSH客户端软件是SSH Secure Shell，而配套的SSH Secure File Transfer则可用在客户端和服务器端复制文件。在Linux下可以通过在终端下运行ssh命令连接服务器，并通过scp命令在服务器和本地之间复制文件。

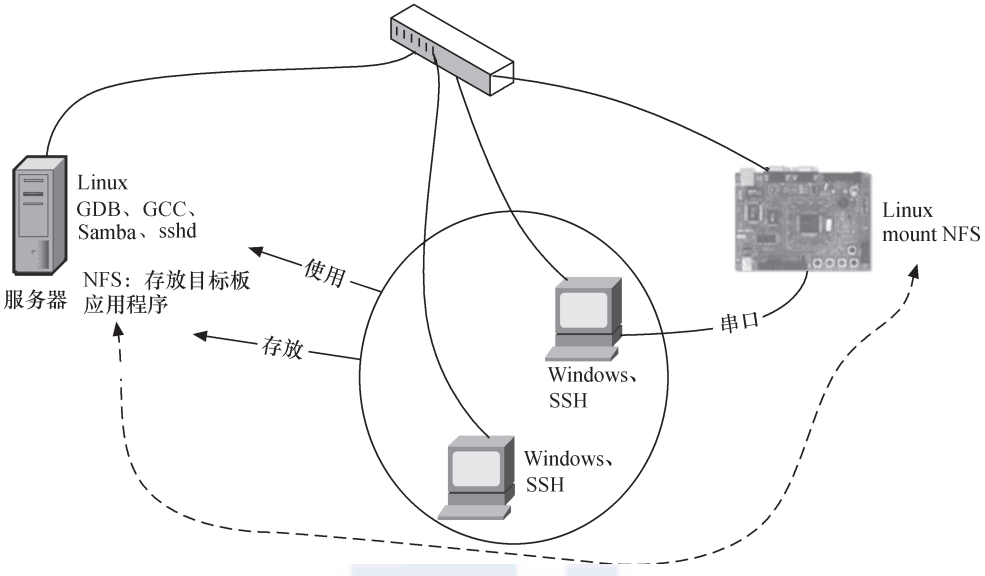


图 3.13　小型 Linux 实验室环境

目标板、服务器和客户端全部通过交换机连接，同时客户端连接目标板的串口作为控制台。在调试 Linux 应用程序时，为了方便在目标板和开发环境间共享文件，有时候可以使用 NFS 文件系统。编写完成的应用程序或内核模块可直接存放在服务器的 NFS 服务目录内，而该目录可被目标板上的 Linux 系统装载到本身的一个目录内。

3.8　串口工具

在嵌入式 Linux 的调试过程中，目标机往往会提供给主机一个串口控制台，驱动工程师在 80% 以上的情况下都是通过串口与目标机通信。因此，好用的串口工具将大大提高工程师的生产效率。

在 Windows 环境下，其附件内自带了超级终端，超级终端包括了对 VT100、ANSI 等终端仿真功能以及对 xmodem、ymodem、zmodem 等协议的支持。

在调试过程中，经常需要保存串口打印信息的历史记录，这时候可以使用“传送”菜单下的“捕获文字”功能来实现。

SecureCRT 是比超级终端更强大且更方便的工具，它将 SSH 的安全登录、数据传送性能和 Windows 终端仿真提供的可靠性、可用性和可配置性结合在一起。鉴于 SecureCRT 具备比超级终端更强大且好用的功能，建议直接用 SecureCRT 替代超级终端。

在开发过程中，为执行自动化的串口发送操作，可以使用 SecureCRT 的 VBScript 脚本功能，让其运行一段脚本，自动捕获接收到的串口信息并向串口上发送指定的数据或文件。下

面的脚本设置了 SecureCRT 等待至接收到“CCC”字符串后通过 xmodem 协议发送 file.bin 文件，接着，当接收到“y/n”时，选择“y”。

```
#$language = "VBScript"
#$interface = "1.0"

Sub main
   Dir = "d:\baohua\"
   ' turn on synchronous mode so we don't miss any data
   crt.Screen.Synchronous = True
   'wait "CCC" string then send file
   crt.Screen.WaitForString "CCC"
  crt.FileTransfer.SendXmodem Dir & "file.bin"
  'wait "y/n" string then send "y"
  crt.Screen.WaitForString "y/n"
  crt.Screen.Send "y" & VbCr
End Sub
```

另外，在 Windows 环境下，也可以选用 PuTTY 工具，该工具非常小巧，而功能很强大，可支持串口、Telnet 和 SSH 等，其官方网址为 http://www.chiark.greenend.org.uk/~sgtatham/putty/。

Minicom 是 Linux 系统下常用的类似于 Windows 下超级终端的工具，当要发送文件或设置串口时，需先按下“Ctrl+A”键，紧接着按下“Z”键激活菜单，如图 3.14 所示。

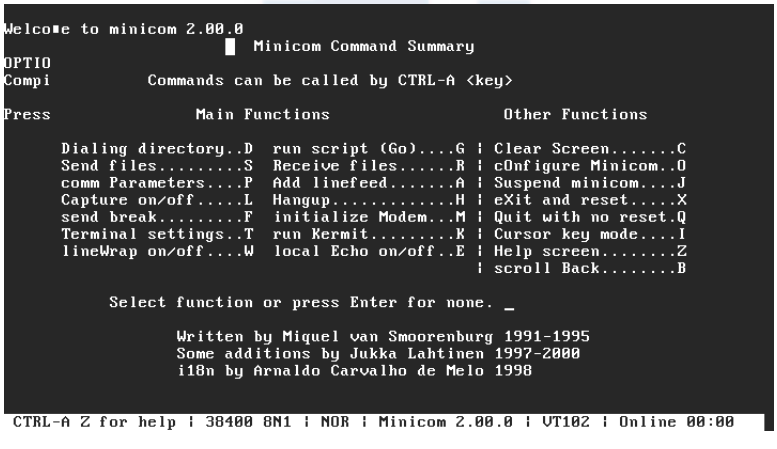


图 3.14 Minicom

除了 Minicom 以外，在 Linux 系统下，也可以直接使用 C-Kermit。运行 kermit 命令即可启动 C-Kermit。在使用 C-Kermit 连接目标板之前，需先进行串口设置，如下所示：

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
```

```
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

之后，使用以下命令就可以将 kermit 连接到目标板：

```
connect
```

在 kermit 的使用过程中，会涉及串口控制台和 kermit 功能模式之间的切换，从串口控制台切换到 kermit 的方法是按下“Ctrl + \”键，然后再按下“C”键。

假设我们在串口控制台上敲入命令，使得目标板进入文件接收等待状态，此后可按下“Ctrl + \”键，再按“C”键，切换到 kermit，运行“send /file_name”命令传输文件。文件传输结束后，再运行“c”命令，将进入串口控制台。

3.9　总结

本章主要讲解了 Linux 内核和 Linux 内核编程的基础知识，为读者在进行 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、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。