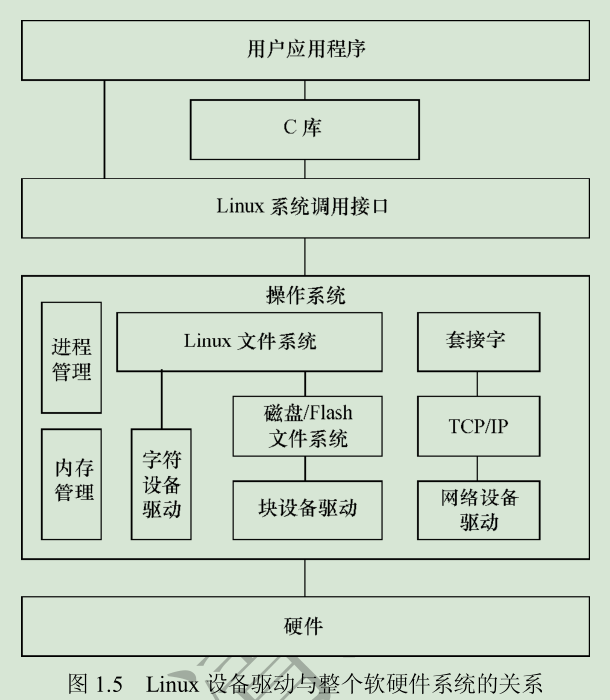
# 1、Linux 将存储器和外设分为 3 个基础大类：

字符设备；

块设备；

网络设备。

字符设备指那些必须以串行顺序依次进行访问的设备，如触摸屏、磁带驱动器、鼠标等。块设备可以用任意顺序进行访问，以块为单位进行操作，如硬盘、软驱等。字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。但是，字符设备和块设备并没有明显的界限，如 Flash 设备符合块设备的特点，但是我们仍然可以把它作为一个字符设备来访问。字符设备和块设备的驱动设计呈现出很大的差异，但是对于用户而言，他们都使用文件系统的操作接口 open()、close()、read()、write()等函数进行访问。在 Linux 系统中，网络设备面向数据包的接收和发送而设计，它并不对应于文件系统的节点。内核与网络设备的通信和内核与字符设备、块设备的通信方式完全不同。另外，TTY 驱动、I 2 C 驱动、USB 驱动、PCI 驱动、LCD 驱动等本身大体可归纳入 3 个基础大类，但是对于这些复杂的设备，Linux 系统还定义了独特的驱动体系结构。



**Linux 设备驱动的学习是一项浩大的工程，读者需要首先掌握以下基础。**

1.编写 Linux 设备驱动要求工程师具有良好的硬件基础，懂得 SRAM、Flash、SDRAM、磁盘的读写方式，UART、I 2 C、USB 等设备的接口，轮询、中断、DMA 的原理，PCI 总线的工作方式以及 CPU 的内存管理单元（MMU）等。

2.编写 Linux 设备驱动要求工程师具有良好的 C 语言基础，能灵活地运用 C 语言的结构体、指针、函数指针及内存动态申请和释放等。

3.编写 Linux 设备驱动要求工程师具有一定的 Linux 内核基础，虽然并不要求工程师对内核各个部分有深入的研究，但至少要了解设备驱动与内核的接口，尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备。

4.编写 Linux 设备驱动要求工程师具有良好的多任务并发控制和同步的基础，因为在设备驱动中会大量使用自旋锁、互斥、信号量、等待队列等并发与同步机制。

## 嵌入式微控制器一般由一个 CPU 核和多个外围电路集成，目前主流的 CPU 核如下：

1．Advanced RISC Machines 公司的 ARM

ARM 内核的设计技术被授权给数百家半导体厂商，做成不同的 SoC 芯片。ARM

的功耗很低，在当今最活跃的无线局域网、3G、手机终端、手持设备、有线网络通信

设备等中应用非常广泛。

《Linux 设备驱动开发详解》——第 2 章、驱动设计的硬件基础

嵌入式学院—华清远见旗下品牌：www.embedu.org

2．Silicon Graphics 公司的 MIPS

两个最重要的 MIPS 芯片厂商为 PMC 和 IDT。PMC-Sierra 公司的 MIPS 处理器被

CISCO 公司大量采用在高端路由器上。IDT 公司在 MIPS 核上集成 PCI 接口，广泛用

于以太网交换，另外也尝试增加了 HDLC、Ethernet、串口、SDRAM 控制器、片选、

DMA 控制器等外设接口，以用于低端通信产品。

3．IBM 和 Motorola 的 PowerPC

PowerPC 处理器是通信和工控领域应用最广泛的处理器，国内包括华为、中兴在

内的通信公司都大量使用 PowerPC，MPC860 和 MPC8260 是其最经典的两款。

4．Motorola 公司独有的内核 68K/COLDFIRE

68K 内核是最早在嵌入式领域广泛应用的内核，其最著名的代表芯片是 68360。

Coldfire 则继承了 68K 的特点并对其保持了兼容。Coldfire 内核被用于 DSP 模块、CAN

总线模块以及一般嵌入式处理器所集成的外设模块，在工业控制、机器人研究、家电

控制等领域被广泛采用。

Motorola 的半导体部已经独立为飞思卡尔半导体公司(Freescale Semiconductor

Inc.)，因为历史原因，上文仍然使用 Motorola。

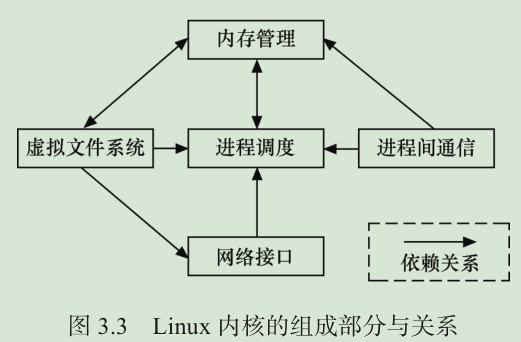
中央处理器的体系架构可以分为两类，一类为冯·诺伊曼结构，一类为哈佛结构。冯·诺伊曼结构也称普林斯顿结构，是一种将程序指令存储器和数据存储器合并在一起的存储器结构。程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同。而哈佛结构将程序指令和数据分开存储，指令和数据可以有不同的数据宽度。此外，哈佛结构还采用了独立的程序总线和数据总线，分别作为 CPU 与每个存储器之间的专用通信路径，具有较高的执行效率。

从指令集的角度来讲，中央处理器也可以分为两类，即 RISC（精简指令集计算机）和 CISC（复杂指令集计算机）。CSIC 强调增强指令的能力、减少目标代码的数量，但是指令复杂，指令周期长；而 RISC 强调尽可能减少指令集、指令单周期执行，但是目标代码会更大。ARM、MIPS、PowerPC 等 CPU 内核都采用了 RISC 指令集。

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。

## Linux 内核的组成部分

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。



**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。

**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。

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。

**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。

**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。

### 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。

## GNU C

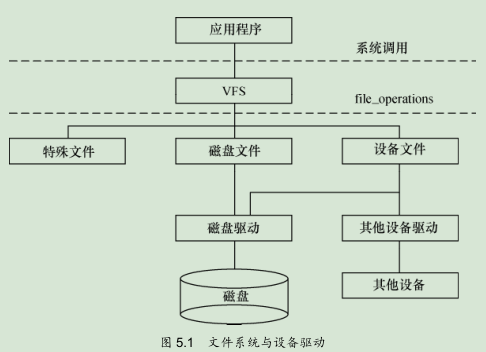
Linux 上可用的 C 编译器是 GNU C 编译器，它建立在自由软件基金会的编程许可证的基础上，因此可以自由发布。GNU C 对标准 C 进行一系列扩展，以增强标准 C 的功能。

在 Linux 内核中，经常会看到 do {} while(0) 这样的语句，其实 do {} while(0) 的用法主要用于**宏定义**中。

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。

# Linux 内核模块

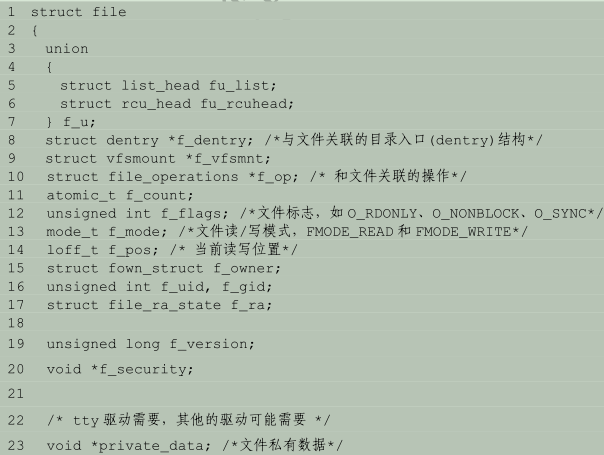
# Linux文件系统与设备文件系统

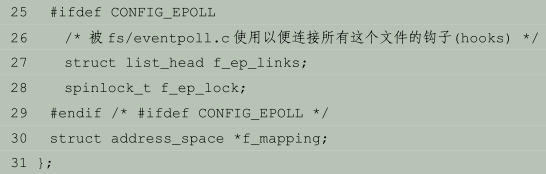


在设备驱动程序的设计中，一般而言，会关心file和inode这两个结构体。

### file结构体

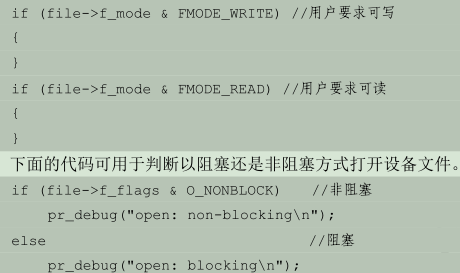
文件结构体代表一个打开的文件（设备对应于设备文件），系统中每个打开的文件在内核空间都有一个关联的 struct file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动源代码中，struct file 的指针通常被命名为 file 或filp（即 file pointer）。代码清单 5.3 给出了文件结构体的定义。





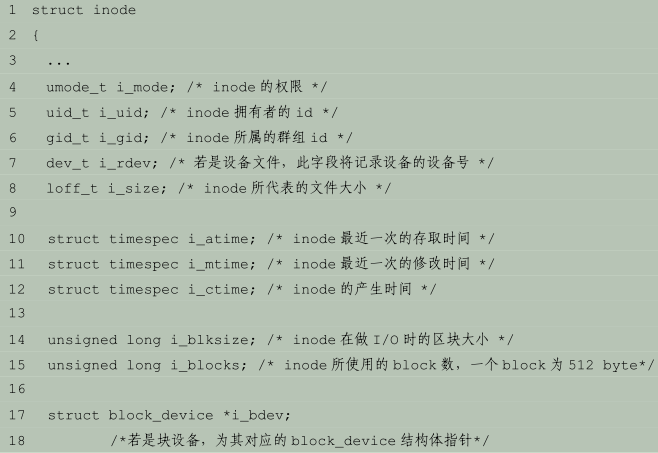
文件读/写模式 mode、标志 f\_flags 都是设备驱动关心的内容，而私有数据指针 private\_data 在设备驱动中被广泛应用，大多被指向设备驱动自定义用于描述设备的结构体。

驱动程序中经常会使用如下类似的代码来检测用户打开文件的读写方式。



### inode 结构体

VFS inode 包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁，inode 结构体的定义如代码清单 5.4 所示。





对于表示设备文件的 inode 结构，i\_rdev 字段包含设备编号。Linux 2.6 设备编号分为主设备编号和次设备编号，前者为 dev\_t 的高 12 位，后者为 dev\_t 的低 20 位。下列操作用于从一个 inode 中获得主设备号和次设备号：

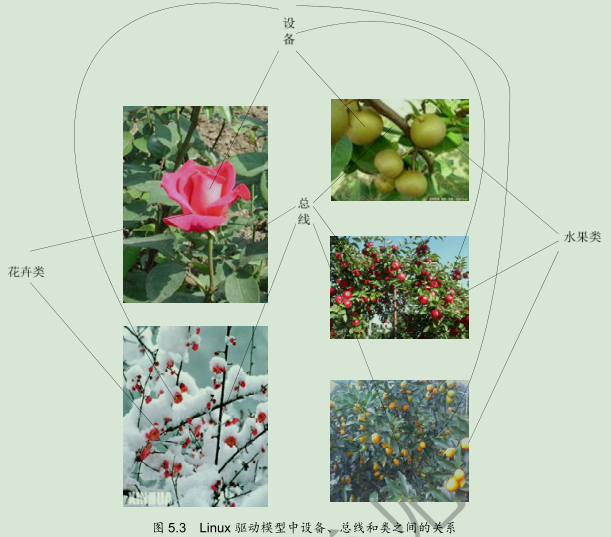


查看/proc/devices 文件可以获知系统中注册的设备，第 1 列为主设备号，第 2 列为设备名

杳看/dev 目录可以获知系统中包含的设备文件，日期的前两列给出了对应设备的主设备号和次设备号

主设备号是与驱动对应的概念，同一类设备一般使用相同的主设备号，不同类的设备一般使用不同的主设备号（但是也不排除在同一主设备号下包含有一定差异的设备）。因为同一驱动可支持多个同类设备，因此用次设备号来描述使用该驱动的设备的序号，序号一般从 0 开始。

内核 Documents 目录下的 devices.txt 文件描述了 Linux 设备号的分配情况，它由 LANANA（The LinuxAssigned Names And Numbers Authority，网址：http://www.lanana.org/）组织维护，Torben Mathiasen 是其中的主要维护者。需要注意的是，LANANA 给出的设备号标准并不是硬性规定，在具体的设备驱动程序中，尽管一般会遵循 LANANA，但是也可以有例外。



在嵌入式系统中，也可以用用udev的轻量级版本mdev，mdev集成于busybox中。在编译busybox的时候，选中mdev相关项目即可。

Android也没有采用udev，它采用的是vold。Vold的机制和udev是一样的，理解了udev，也理解了vold。Android的源代码NetlinkManager.cpp同样是监听基于netlink的套接字，并解析收到的信息。

Linux 系统用户空间的文件编程有两种方法，即通过 Linux API 和通过 C 库函数访问文件。用户空间看不到设备驱动，能看到的只有设备对应的文件，因此文件编程即是用户空间的设备编程。

Linux 系统按照功能对文件系统的目录结构进行了良好的规划。/dev 是设备文件的存放目录，devfs 和udev 分别是 Linux 2.4 设备和 Linux 2.6 设备生成设备文件节点的方法，前者运行于内核空间，后者运行于用户空间。

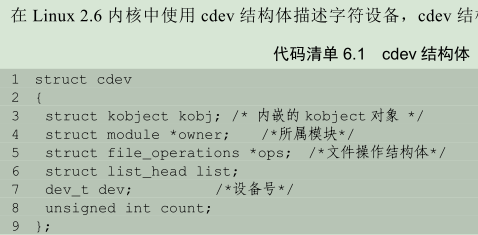
Linux 2.6 设备通过一系列数据结构定义了设备模型，设备模型与 sysfs 文件系统中的目录和文件存在一种对应关系，

设备与驱动分离，并通过总线进行匹配。

udev可以利用内核通过netlink（套接字）发出的uevent信息动态创建设备文件节点。

# Linux 字符设备驱动

## cdev 结构体



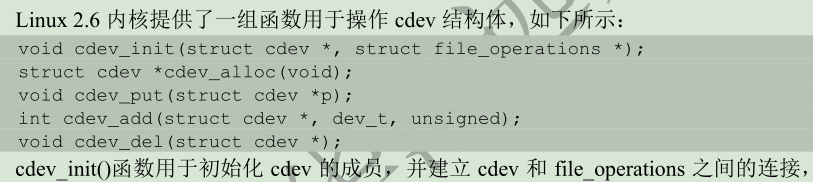
cdev 结构体的 dev\_t 成员定义了设备号，为 32 位，其中高 12 位为主设备号，低 20 位为次设备号。使用下列宏可以从 dev\_t 获得主设备号和次设备号。



而使用下列宏则可以通过主设备号和设备号生成 dev\_t。

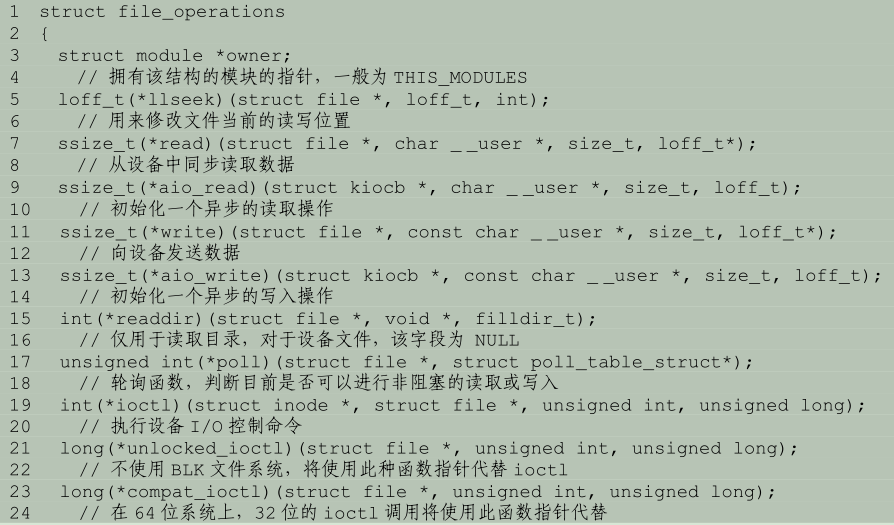


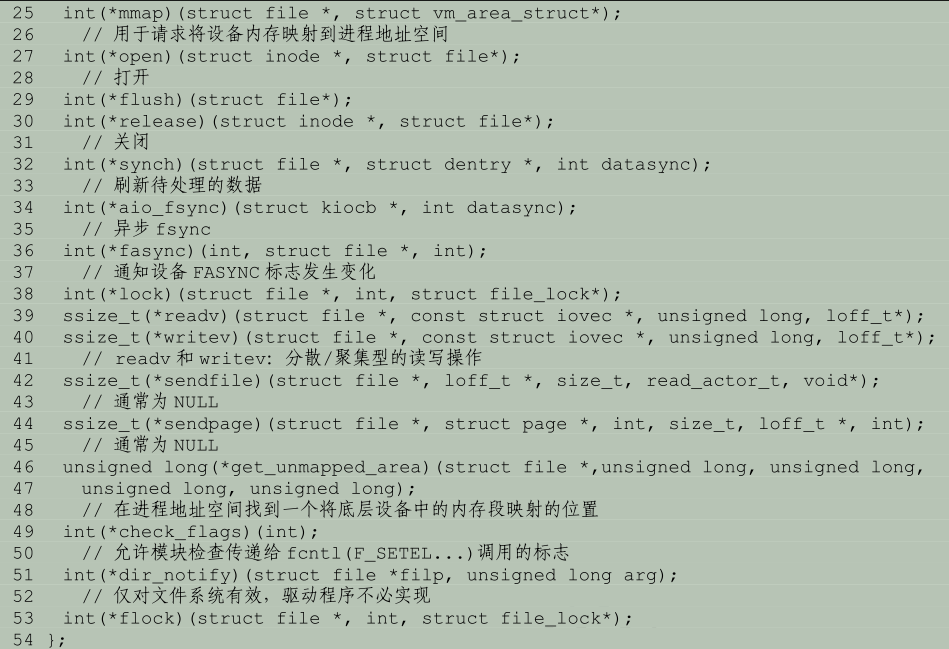
cdev 结构体的另一个重要成员 file\_operations 定义了字符设备驱动提供给虚拟文件系统的接口函数。



## file\_operations 结构体

file\_operations 结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际会在应用程序进行 Linux 的 open()、write()、read()、close()等系统调用时最终被调用。file\_operations 结构体目前已经比较庞大，它的定义如代码清单 6.4 所示。





下面对 file\_operations 结构体中的主要成员进行讲解。

llseek()函数用来修改一个文件的当前读写位置，并将新位置返回，在出错时，这个函数返回一个负值。

read()函数用来从设备中读取数据，成功时函数返回读取的字节数，出错时返回一个负值。

write()函数向设备发送数据，成功时该函数返回写入的字节数。如果此函数未被实现，当用户进行 write()系统调用时，将得到-EINVAL 返回值。

readdir()函数仅用于目录，设备节点不需要实现它。

ioctl()提供设备相关控制命令的实现（既不是读操作也不是写操作），当调用成功时，返回给调用程序一个非负值。内核本身识别部分控制命令，而不必调用设备驱动中的 ioctl()。如果设备不提供 ioctl()函数，对于内核不能识别的命令，用户进行 ioctl()系统调用时将获得-EINVAL 返回值。

mmap()函数将设备内存映射到进程内存中，如果设备驱动未实现此函数，用户进行 mmap()系统调用时将获得-ENODEV 返回值。这个函数对于帧缓冲等设备特别有意义。

当用户空间调用 Linux API 函数 open()打开设备文件时，设备驱动的 open()函数最终被调用。驱动程序可以不实现这个函数，在这种情况下，设备的打开操作永远成功。与 open()函数对应的是 release()函数。

poll()函数一般用于询问设备是否可被非阻塞地立即读写。当询问的条件未触发时，用户空间进行select()和 poll()系统调用将引起进程的阻塞。

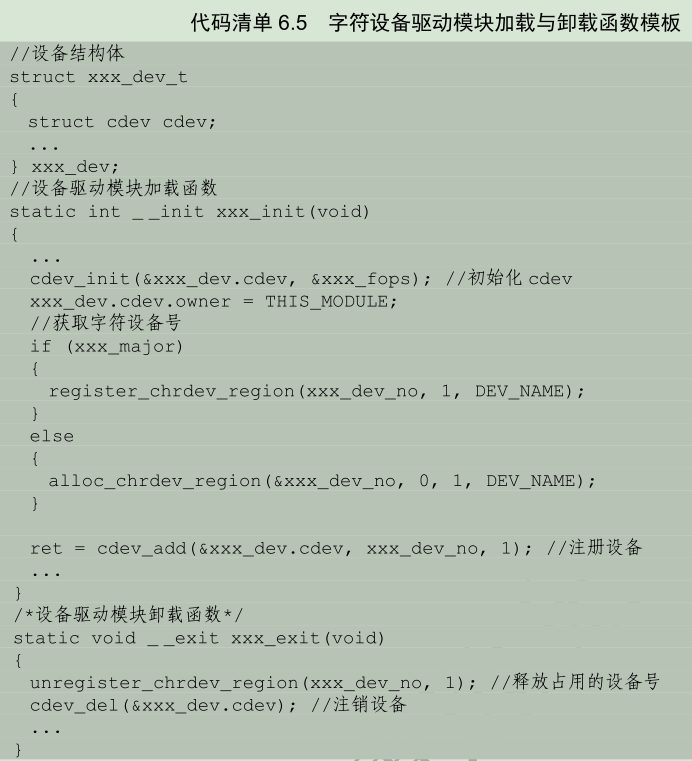
aio\_read()和 aio\_write()函数分别对与文件描述符对应的设备进行异步读、写操作。设备实现这两个函数后，用户空间可以对该设备文件描述符调用 aio\_read()、aio\_write()等系统调用进行读写。

## Linux 字符设备驱动的组成

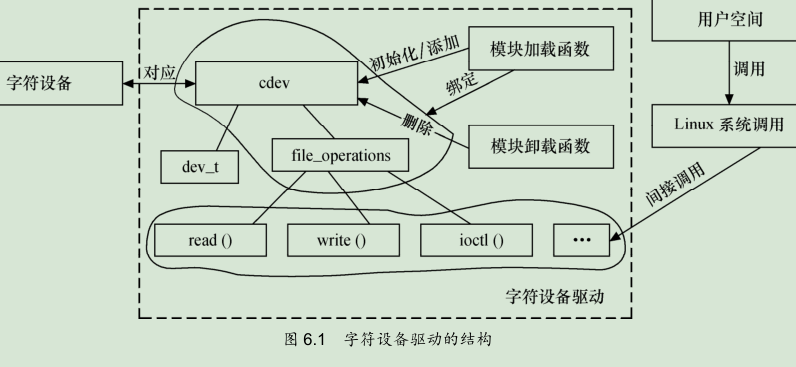
### 1．字符设备驱动模块加载与卸载函数

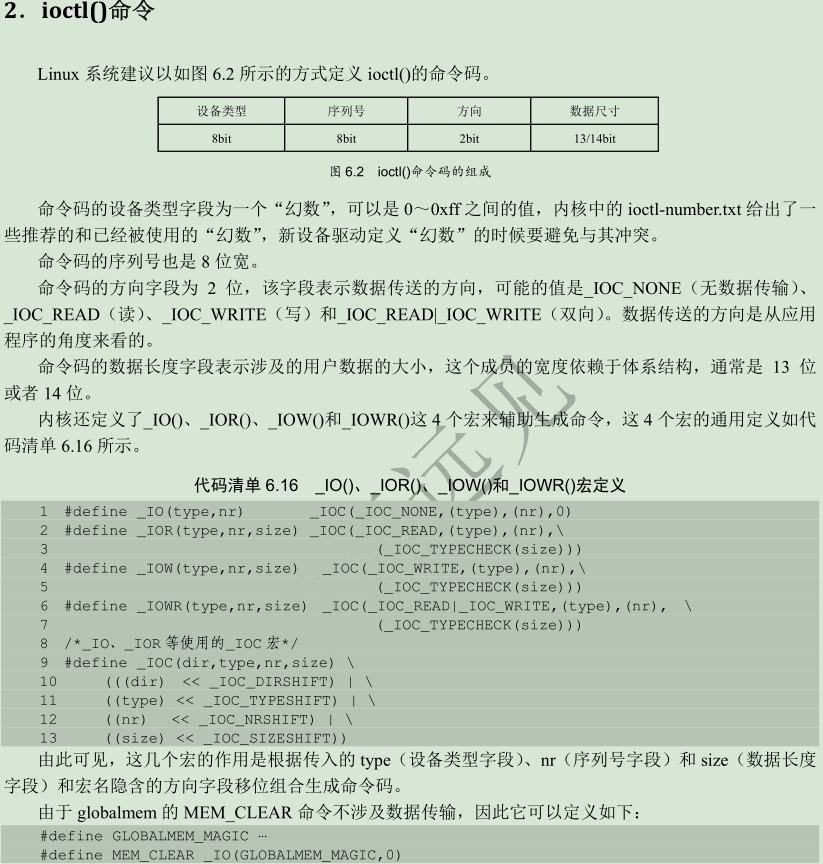
在字符设备驱动模块加载函数中应该实现设备号的申请和 cdev 的注册，而在卸载函数中应实现设备号的释放和 cdev 的注销。

## globalmem 设备驱动



所示为字符设备驱动的结构、字符设备驱动与字符设备以及字符设备驱动与用户空间访问该设备的程序之间的关系。







container\_of()的作用是通过结构体成员的指针找到对应结构体的指针，这个技

巧在 Linux 内核编程中十分常用。

## 总结

字符设备是3 大类设备（字符设备、块设备和网络设备）中较简单的一类设备，其驱动程序中完成的主要工作是初始化、添加和删除cdev 结构体，申请和释放设备号，以及填充file\_operations 结构体中的操作函数，实现file\_operations结构体中的read()、write()和ioctl()等函数是驱动设计的主体工作。

# 第七章Linux 设备驱动中的并发控制

## 7.1并发与竞态

并发（concurrency）指的是多个执行单元同时、并行被执行，而并发的执行单元对共享资源（硬件资源和软件上的全局变量、静态变量等）的访问则很容易导致竞态（race conditions）。

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。

访问共享资源的代码区域称为临界区（critical sections），临界区需要以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁和信号量等是 Linux 设备驱动中可采用的互斥途径。

## 7.2 编译乱序和执行乱序

理解Linux内核的锁机制，还需要理解编译器和处理器的特点。

Linux不太喜欢volatile

## 7.3 中断屏蔽

由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也就得以避免了。

由于 Linux 系统的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。

## 7.4 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

Linux 内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不被打断。位和整型变量原子操作都依赖底层 CPU 的原子操作来实现，因此所有这些函数都与 CPU 架构密切相关。

## 7.4 自旋锁

自旋锁（spin lock）是一种对临界资源进行互斥手访问的典型手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。

如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。

Linux 系统中与自旋锁相关的操作主要有如下 4 种。

定义自旋锁



定义自旋锁



获得自旋锁

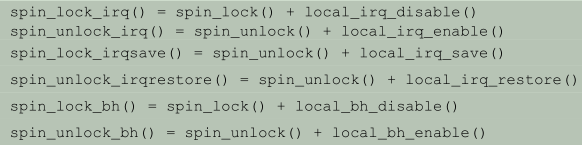
（会原地打转）

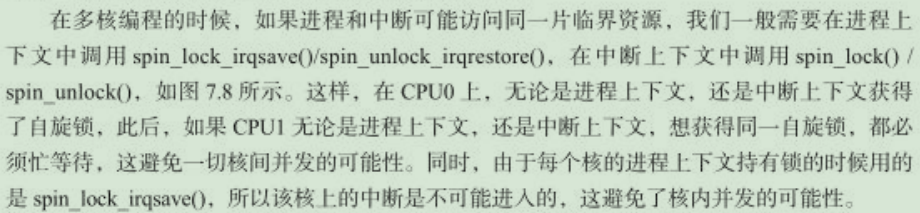
（不会原地打转，false）

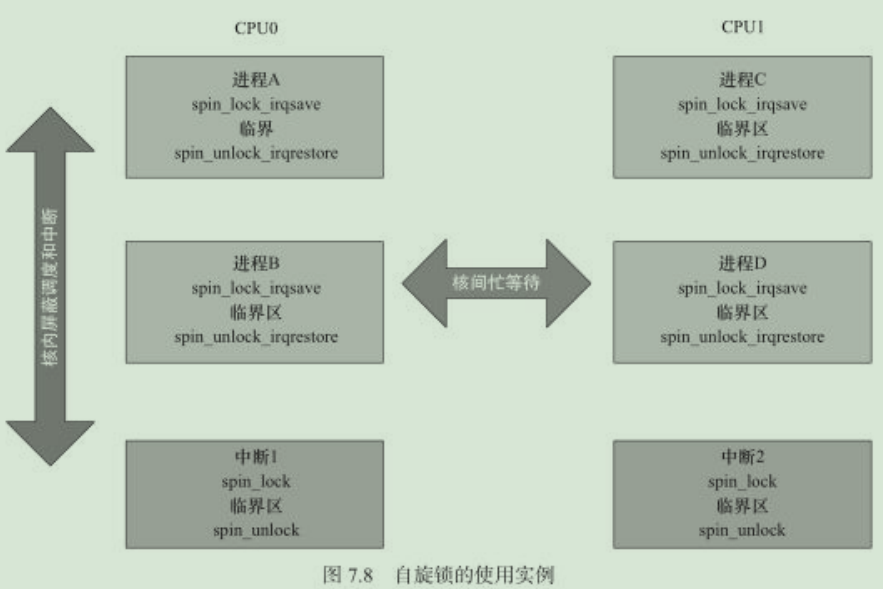
释放自旋锁



尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部（BH）的影响。为了防止这种影响，就需要用到自旋锁的衍生 。 spin\_lock()/spin\_unlock() 是 自 旋 锁 机 制 的 基 础 ， 它 们 和 关 中 断 local\_irq\_ disable()/ 开 中 断local\_irq\_enable()、关底半部 local\_bh\_disable()/开底半部 local\_bh\_enable()、关中断并保存状态字local\_irq\_save()/开中断并恢复状态 local\_irq\_restore()结合就形成了整套自旋锁机制，关系如下所示：







驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题：

1. 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。
2. 自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。
3. 在自旋锁锁定器件不能调用可能引起进程调度的函数。如果进程获得自旋锁之后再阻塞，也有可能导致死锁的发生。copy\_from\_user()、copy\_to\_user()和 kmalloc()等函数都有可能引起阻塞，因此在自旋锁的占用期间不能调用这些函数。
4. 在单核情况下编程的时候，也应该认为自己的CPU是多核的，驱动特别强调跨平台的概念。比如，在单CPU的情况下，若中断和进程可能访问同一临界区，进程里调用spin\_lock\_irqsave()是安全的，在中断里其实不调用spin\_lock()也没有问题，因为spin\_lock\_irqsave()可以保证这个CPU的中断服务程序不可能执行。但是，若CPU变成多核，spin\_lock\_irqsave()不能屏蔽另外一个核的中断，所以另外一个核就可能造成并发问题。因此，无论如何，我们在中断服务程序里也应该spin\_lock()。



### 读写自旋锁

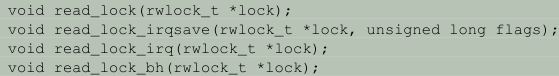
自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生锁读写自旋锁（rwlock）可允许读的并发。

读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有一个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。读写自旋锁涉及的操作如下所示。

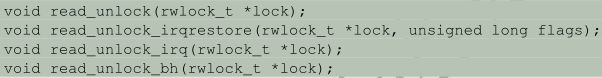
1 ．定义和初始化读写自旋锁



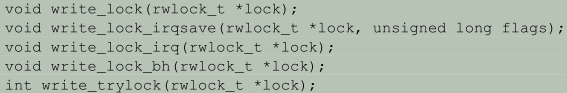
2 ．读锁定



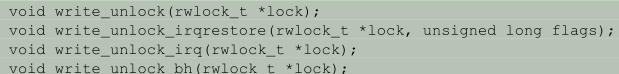
3 ．读解锁

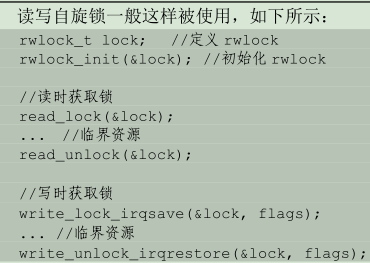


4 ．写锁定



5 ．写解锁





### 顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元可以在写执行单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。

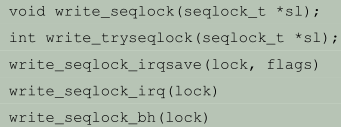
但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。

如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。这种锁在读写同时进行的概率比较小时，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

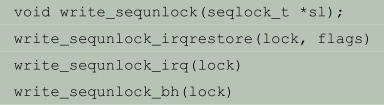
顺序锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致 Oops。

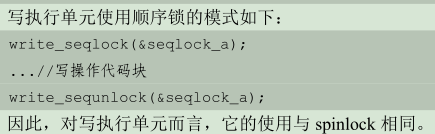
在 Linux 内核中，写执行单元涉及如下顺序锁操作。

1.获得顺序锁

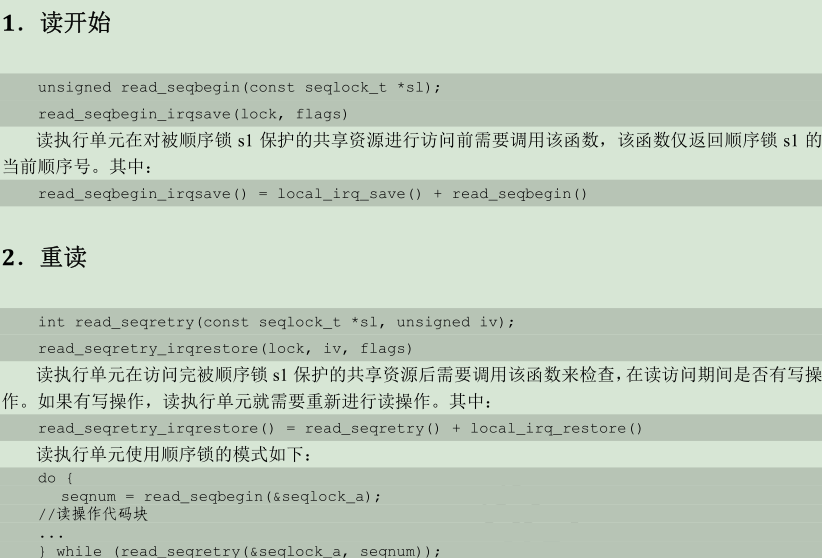


2.释放顺序锁





读执行单元涉及如下顺序锁操作。

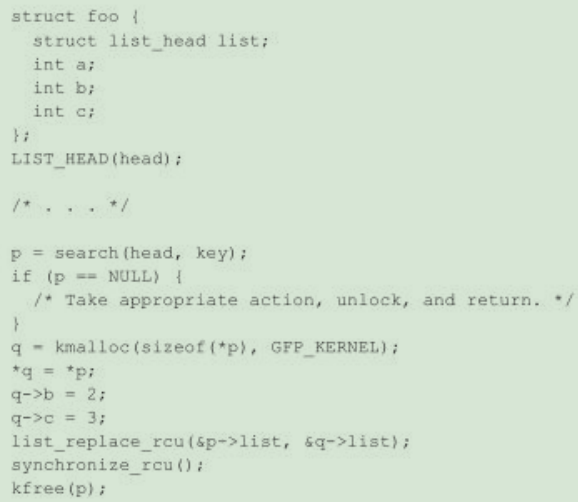


### 读－拷贝－更新

RCU（Read-Copy Update，读－拷贝－更新），它是基于其原理命名的。RCU 并不是新的锁机制，

使用 RCU 的写执行单元在访问它前需首先复制一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的CPU 都退出对共享数据的操作的时候。读执行单元没有任何同步开销，而写执行单元的同步开销则取决于使用的写执行单元间的同步机制。

用代码来描述这个逻辑就是：



RCU 可以看做读写锁的高性能版本，相比读写锁，RCU 的优点在于既允许多个读执行单元同时访问被保护的数据，又允许多个读执行单元和多个写执行单元同时访问被保护的数据。但是，RCU 不能替代读写锁，因为如果写比较多时，对读执行单元的性能提高不能弥补写执行单元导致的损失。因为使用 RCU 时，写执行单元之间的同步开销会比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其他写执行单元的修改操作。

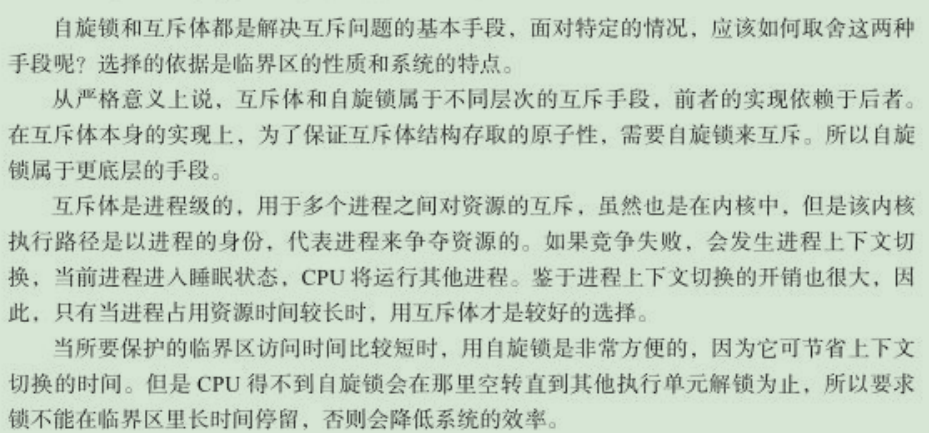
## 7.6 信号量

信号量（semaphore）是用于保护临界区的一种常用方法，它的使用方式和自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。

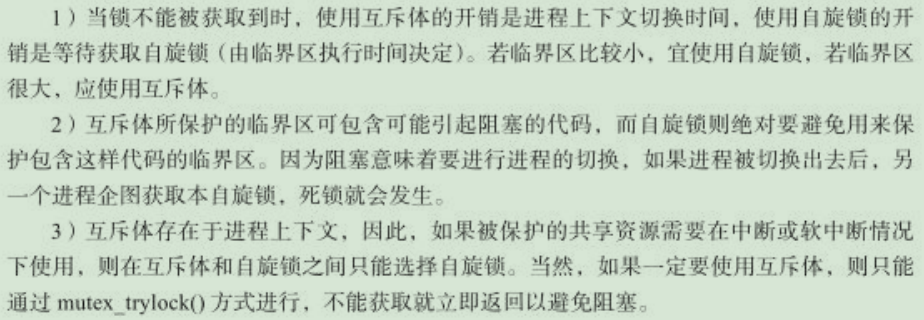
由于新的Linux内核倾向于直接使用mutex（互斥锁）作为互斥手段，信号量用作互斥不再被推荐使用。

信号量也可以用于同步，一个进程执行down()等待信号量，另外一个进程B执行up()释放信号量，这样进程A就同步地等待了进程B。

## 7.7 互斥体（锁）



由此，可以终结出自旋锁和互斥体选用的3项原则。



## 7.8 完成量

Linux提供来了完成量（Completion，关于这个名字，至今没有好的翻译），它用于一个执行单元等待另一个执行单元来执行完成某事。

可以用来同步

## 总结

并发和竞态广泛存在，中断屏蔽、原子操作、自旋锁和互斥体都是解决并发问题的机制。中断屏蔽很少单独使用，原子操作只能针对整数进行，因此自旋锁和互斥体应用最为广泛。

自旋锁会导致死循环，锁定期间不允许阻塞，因此要求锁定的临界区小。互斥体允许临界区阻塞，可以适用于临界区大的情况。