Broadview
www.broadview.com.cn

多年开发笔记融入其中，精研Linux内核铸就鸿篇。

LINUX

Linux 内核分析及编程

倪继利 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Linux
内核分析及编程

倪继利 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书作者在整理自己多年研发笔记的基础上，以精心挑选的典型开发实例，向读者详细讲述了 Linux 内核源代码的各部分结构、原理及组成框架，主要分析了 Linux 最新版本（2.6.11）的内核源代码，帮助读者深入理解 Linux 内核，精通 Linux 内核编程。全书分为 20 章，内容包括进程管理、进程间通信、内存管理、文件系统、I/O 接口及资源管理、内核的编译及调试原理、网络通信、内核安全、USB 驱动程序等。

对于想了解 Linux 开发，以及从事 Linux 内核编程的开发人员来说，本书是一本集大成之作，它既有讲解透彻的原理，也有详细实用的示例，更有作者多年从事实际开发工作的心得。本书主要针对从事 Linux 内核编程的中高级读者及软件工程师，也很合适作为大学教材和参考书。

图书在版编目（CIP）数据

Linux 内核分析及编程 / 倪继利著. —北京：电子工业出版社，2005.9
ISBN 7-121-01518-8

I. L… II. 倪… III. Linux 操作系统—程序设计 IV. TP316.89

中国版本图书馆 CIP 数据核字（2005）第 075076 号

策划编辑：郭 立
责任编辑：孙学瑛
印 刷：北京天宇星印刷厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
经 销：各地新华书店
开 本：850×1168 1/16 印张：52.5 字数：1 402 千字
印 次：2005 年 9 月第 1 次印刷
印 数：5000 册 定价：88.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前　言

Linux 是开放的源代码，它具备了 UNIX 的全部特征，与 POSIX 标准兼容。Linux 操作系统，如 Redhat Linux 9，不仅被广泛地应用于 PC、服务器，还广泛地用于手机、PDA 等高端嵌入设备。由于 Linux 综合了 UNIX 主要派生系统（包括 SysV、BSD）的先进技术，所以，Linux 操作系统上能运行原 UNIX 系统的各种应用程序，同时，还存在大量的应用程序开放源代码供开发者使用。而且，许多著名公司均发布了自行开发的 Linux 程序源代码。所有这些因素导致了 Linux 在嵌入系统中的大量应用。

为什么写作本书

如今，Linux 内核代码几乎是每个软件工程师必读的，即便内核代码复杂难懂。作为一名 Linux 编程者，我一向颇为留意内核编程方面的图书，我一直期望能有一本实践性很强的书，是真正从事 Linux 内核开发的人士写作的。

由于我多年从事 Linux 内核开发工作，并一直记有相关笔记，后来，我便想到，如果我从一名研发者的角度来写作这样一本书，把自己的笔记加以整理，那么，对读者的实践应该会有不小的帮助。在整理自己的开发笔记的基础上，我还查阅了大量相关资料，加强研究，力求融会贯通，费时两年，写成这本书。

Linux 2.6 版内核改写了以前版本内核的绝大部分，本书主要针对目前 Linux 的最新版本 2.6.11 版，对以后的新版本也具有普适性。本书的主要目的是帮助软件工程师读懂 Linux 2.6.11 版本内核，并能开发各种驱动程序、编写内核模块。

在这本书里，我将平常编程中遇到的重点、难点进行分析，并给予充分的论述，相信其中许多问题是其他内核编程者也会遇到的。对于软件工程师来说，本书有助于他们少走弯路，更快地掌握 Linux 2.6.11 内核源代码及编程技巧。

关于本书作者

笔者从在清华大学电子系读研究生起，就开始从事 Linux 内核编程，后又一直在外国著名公司从事 Linux 内核编程工作，先后从事过 Linux 内核的移植、USB 驱动程序编写、内核安全程序的编写等。目前，Linux 被广泛地应用于手机、PDA 等高端嵌入设备，笔者在这方面有丰富的开发经验。

本书主要内容

本书共包括 20 章，每章的主要内容如下：

第 1 章“数据类型及链表”介绍了数据类型占用的空间及用户空间输出的数据类型，说明了内核通用链表的原理，以及行内汇编语言的语法。

第 2 章“进程及进程调度”分析了进程结构及进程调度算法。

第 3 章“内核同步机制”介绍了内核的互斥机制：自旋锁、原子操作和信号量。还说明了 RCU 读写机制，及内核与用户空间进行通信的机制。

第 4 章“内存管理”介绍了虚拟内存及映射，还分析了物理内存的管理（它包括缓存的分配及回收，请页机制，交换空间等），还说明了内存缓冲池和大块内存的管理机制。

第 5 章“虚拟文件系统”介绍了虚拟文件系统的结构和实现文件操作的机制，还介绍了底层各种缓存的管理。

第 6 章“EXT2 文件系统”介绍了 EXT2 文件系统逻辑分区的结构、节点和 dentry 的管理，说明了读写系统调用的具体实现过程。

第 7 章“其他文件系统”介绍了常用的一些文件系统：ramfs、proc、vfat、devfs 和 sysfs。

第 8 章“I/O 端口资源管理”介绍了中断处理、DMA 及电源管理的实现机制。

第 9 章“模块机制”介绍了内核模块的实现机制，还分析了内核空间的保护机制。

第 10 章“设备驱动程序”介绍了字符设备及块设备驱动程序的工作原理，分析了通用硬盘及块层的机制，还说明了如何编写字符设备与块设备驱动程序。

第 11 章“Flash 闪存及 SD/MMC 卡设备驱动程序”分析了 MTD 设备驱动程序和 MMC/SD 驱动程序，它们分别驱动 Flash 闪存和存储卡，是嵌入设备的主要存储设备。

第 12 章“Linux 系统初始化”阐述了在 i386 机器上的 BootLoader 和嵌入设备上的 Blob，分析了 Linux 系统的启动过程。

第 13 章“系统调用”分析了系统调用的实现机制。

第 14 章“Linux 网络系统分层结构”介绍了数据包的传递过程、与应用层的接口、与底层的接口以及网络驱动程序的编写。

第 15 章“执行文件的运行过程”阐述了动态链接与静态链接的概念，并说明了动态链接中函数定位的原理。然后分析了 ELF 文件格式，以及 ELF 文件在内核中是如何加载运行的。

第 16 章“进程间通信”分析了进程间通信机制在内核中的实现原理。

第 17 章“Linux 的安全策略”介绍了 selinux 的安全策略配置语言、selinux 模块的实现，以及病毒防火墙的原理。

第 18 章“内核配置与编译”说明了内核的配置、配置语言的语法，还分析了 makefile 是如何进行内核编译的。

第 19 章“Linux 内核调试”分析了内核调试的方法，控制台驱动程序，以及如何将打印信息显示在控制台上，阐述了日志系统是如何工作的，还说明了 ptrace 调试跟踪的原理。

第 20 章“USB 总线驱动程序”分析了 USB 总线接口驱动程序（包括 USB 总线驱动程序的结构、编写方法和 USB 接口的 U 盘设备驱动程序）。

附录“Linux 系统调用”列表说明了系统调用的功能，供读者快速查询。

如何阅读本书

这是一本大厚书，读者应该怎样利用这本书呢？

在阅读此书前，读者应当学过操作系统原理、数据结构、计算机结构及组成、汇编语言及 C 语言等课程或具备这方面的知识。这本书章节的安排是依据读者对内核学习循序渐进的顺序而设立的，建议初学者从前至后阅读。由于 Linux 内核复杂难懂，我建议读者分几遍阅读本书。

- 第一遍先将书通读，主要弄清楚概念，程序代码部分可以只是浏览一下。当对概念有初步认识时，再尝试编译安装内核，安装一个驱动程序模块。
- 第二遍再对照源代码详细看一个驱动程序，如：USB 驱动程序，理解驱动程序模块是如何调用

内核函数、注册驱动程序等，再反过来，仔细看这些内核函数的实现，就明白了内核为什么要写这些函数了。

- 第三遍再根据需要对照源程序看相关章节，例如：对照源程序看“内存管理”一章中 vmolloc 函数是如何实现的。这种阅读法使读者能从本书中获得最大收益，是学习内核的好方法。

当然，如果你是一名内核精通者，也可以根据需要直接跳读到相关章节，查阅你需要的内容。

阅读内核是一个反复又枯燥的过程，读者只有在反复的研读中，才能逐渐使自己的内核知识条理化，在此基础上，你还需要去应用这些知识，比如，你可以尝试写一个驱动程序、系统调用或文件系统，在实践的过程中再反复查阅参考书及源代码，这样才能达到掌握内核知识的目的。

致谢

我首先要特别感谢我的妻子邓咏秋，我记录开发笔记的初衷只是总结自己的经验和收获，供自己使用，或与朋友分享，是她促成了这本书的出版，使这本书能与成千上万读者分享。她在撰写博士学位论文的同时，热情而高效率地充当着我的“经纪人”，帮我承担了与出版社联系的大部分工作，使我能够全身心投入到书稿写作中去。

我还要真诚地感谢三家优秀的计算机图书出版社：电子工业出版社、清华大学出版社和人民邮电出版社，他们都愿意出版我的这本书。在此，我非常感谢他们对这本书的认可和兴趣。

电子工业出版社计算机图书事业部主任郭立女士的热情推动最终促成了我们与电子工业出版社的合作。在此，非常感谢郭立女士和电子工业出版社对本书的重视，以及他们为本书出版所做的一切。

由于作者水平有限，书中不足及错误之处在所难免，敬请各位专家和读者批评指正。

谨以此书献给正在从事和将要从事 Linux 内核编程的人们。

倪继利

2005 年 8 月

目　录

第 1 章　数据类型及链表

本章介绍了数据类型占用的空间，对用户空间输出的数据类型，说明了内核通用链表的原理，还介绍了行内汇编语言的语法。

1.1　数据类型所占空间

当 Linux 内核在体系结构差异较大的平台之间移植时，会产生与数据类型相关的问题。在编译内核时使用 -Wall -Wstrict-prototypes 选项，可以避免很多错误的发生。

内核使用的基本数据类型主要有：

- int　　标准 C 语言整数类型；
- u32　　32 位整数类型；
- pid_t　　特定内核对象 pid 的类型。

在不同的 CPU 体系结构上，C 语言的数据类型所占空间不一样。下面是在 x86 下数据类型所占的字节数：

arch	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i686	1	2	4	4	4	8	1	2	4	8

下面是在其他平台上的数据类型所占的字节数：

arch	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386	1	2	4	4	4	8	1	2	4	8
alpha	1	2	4	8	8	8	1	2	4	8
armv4l	1	2	4	4	4	8	1	2	4	8
ia64	1	2	4	8	8	8	1	2	4	8
m68k	1	2	4	4	4	8	1	2	4	8
mips	1	2	4	4	4	8	1	2	4	8
ppc	1	2	4	4	4	8	1	2	4	8
sparc	1	2	4	4	4	8	1	2	4	8
sparc64	1	2	4	4	4	8	1	2	4	8

其中基于 sparc64 平台的 Linux 用户空间可以运行 32 位代码，用户空间指针是 32 位宽的，但内核空间是 64 位的。

内核中的地址是 unsigned long 类型，指针大小和 long 类型相同。

内核提供下列数据类型。所有类型在头文件<asm/types.h>中声明，这个文件又被头文件<Linux/types.h>所包含。下面是 include/asm/types.h 文件。

```
#ifndef _I386_TYPES_H
#define _I386_TYPES_H

#ifndef __ASSEMBLY__
```

```
typedef unsigned short umode_t;

// 下面__xx类型不会损害POSIX 名字空间，在头文件使用它们，可以输出给用户空间
typedef __signed__ char __s8;
typedef unsigned char __u8;

typedef __signed__ short __s16;
typedef unsigned short __u16;

typedef __signed__ int __s32;
typedef unsigned int __u32;

#if defined(__GNUC__) && !defined(__STRICT_ANSI__)
typedef __signed__ long long __s64;
typedef unsigned long long __u64;
#endif

#endif /* __ASSEMBLY__ */

//下面的类型只用在内核中，否则会产生名字空间崩溃
#ifdef __KERNEL__

#define BITS_PER_LONG 32

#ifndef __ASSEMBLY__

#include <Linux/config.h>

typedef signed char s8;
typedef unsigned char u8;

typedef signed short s16;
typedef unsigned short u16;

typedef signed int s32;
typedef unsigned int u32;

typedef signed long long s64;
typedef unsigned long long u64;

/* DMA addresses come in generic and 64-bit flavours.  */
#ifdef CONFIG_HIGHMEM64G
typedef u64 dma_addr_t;
#else
typedef u32 dma_addr_t;
#endif
typedef u64 dma64_addr_t;

#ifdef CONFIG_LBD
typedef u64 sector_t;
#define HAVE_SECTOR_T
#endif

typedef unsigned short kmem_bufctl_t;

#endif /* __ASSEMBLY__ */
#endif /* __KERNEL__ */
#endif
```

下面是 Linux/types.h 的部分定义。

```
#ifndef _LINUX_TYPES_H
#define _LINUX_TYPES_H
```

```
#ifdef    __KERNEL__//这个宏定义必须在#include < asm /types.h>之前
#include <Linux/config.h>

#define BITS_TO_LONGS(bits) \
     (((bits)+BITS_PER_LONG-1)/BITS_PER_LONG)
#define DECLARE_BITMAP(name,bits) \
     unsigned long name[BITS_TO_LONGS(bits)]
#endif

#include <Linux/posix_types.h>
#include <asm/types.h>

u8;   /* unsigned byte (8 bits) */
u16;  /* unsigned word (16 bits) */
u32;  /* unsigned 32-bit value */
u64;  /* unsigned 64-bit value */
```

使用有前缀的类型用于将变量显露给用户空间，如_ _u8 类型。例如：一个驱动程序通过 ioctl 函数与运行在用户空间的程序交换数据，应该用_ _u32 来声明 32 位的数据类型。

有时内核使用 C 语言的类型，如 unsigned int，这通常用于大小独立于体系结构的数据项。

内核中许多数据类型由 typedef 声明，这样方便移植。如使用 pid_t 类型作为进程标志符（pid）的类型，而不是 int 类型，pid_t 屏蔽了在不同平台上的实际数据类型的差异。

如果不容易选择合适的类型，就将其强制转换成最可能的类型（long 或 unsigned long）。

1.2　有关移植性的其他问题

1.2.1　时间间隔

对于 1s 的时间间隔，不能用 100 个 jiffy。因为不同的平台可能设置不一，所以，应该用 Hz（每秒定时器中断的次数）来衡量。

1.2.2　页面大小

内存页的大小为 PAGE_SIZE 字节，而不是 4 KB。在不同的平台上，页大小范围可以是 4 KB 到 64 KB。PAGE_SHIFT 的作用是通过对地址右移 PAGE_SHIFT 得到一个地址所在页的页号。对于用户空间，可以使用 getpagesize 函数来得到页的大小。

例如，使用 get_free_pages 函数申请 16 KB 空闲空间（即 2^{14}），先将 16 KB 转换成 2^{order} 空闲页数。在 x86 下定义 PAGE_SHIFT 为 12，即 2^{12} 为 4 KB。

```
int order = (14 - PAGE_SHIFT > 0) ? 14 - PAGE_SHIFT : 0;
buf = get_free_pages(GFP_KERNEL, order);
```

1.2.3　字节存储顺序

字节存储顺序有两种：低字节优先（little-endian）或称为小端字节序，高字节优先（big-endian）或称大端字节序。低字节优先的方式是在存储多字节数值时，低字节在前面，高字节在后面。高字节优先则正好相反。现代的处理器大部分工作在 big-endian 模式下。Linux 内核定义了一组宏，用于在处理器字节序数据和特殊字节序数据之间进行转换。这组宏如下(在 Linux/byteorder/big_endian.h 中)：

```
u32 _ _cpu_to_le32 (u32); //将一个 CPU 使用的值的字节序转换成一个小端字节序的无符号值
```

```
//32位 little-endian数
u32 _ _le32_to_cpu (u32);  //与上操作正好相反，将小端字节序的32位数转换为cpu使用的字节序。
```

使用这些宏，可以使编写可移植代码的工作变得更加容易。这个头文件还有一些类似的转换，如__be64_to_cpu，__le16_to_cpus 和__cpu_to_le32p 函数。

1.2.4 数据对齐

例如，读取一个存储在非 4 字节倍数的地址中的 4 字节值，这就存在数据对齐的问题。如果需要访问未对齐的数据，则应该使用下面的宏：

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是与类型无关的，对各种数据项，不管它是 1 字节、2 字节、4 字节，还是 8 字节，这些宏都有效。所有版本的内核都定义了这些宏。

1.3 内核通用链表

操作系统内核经常需要维护数据结构。内核有标准的循环链表、双向链表的实现。在<Linux/list.h>文件中定义了一个 list_head 类型简单结构：

```
struct list_head {
  struct list_head *next, *prev;
};
```

通用链表的常用用途是将某一个数据结构本身串成链表，或将某些链表与一个数据结构联系起来，这两种情况实质上都是由结构 list_head 组成链表，只是 list_head 所“背负”的负载不一样。下面分别举例说明这两种用途。

以下示例说明了如何将某一个数据结构本身串成链表，并对链表进行操作，同时还说明 list_head 结构的实现与使用。

示例：将某一个数据结构本身串成链表。

（1）加入 list_head 结构成员。

假设有一个 example_struct 结构需连接成链表，因而在其结构里面加上 list_head 成员，就组成了结构链表，如下：

```
struct example_struct {
  struct list_head list;
  int priority;
  ……//其他成员
};
```

在 example_struct 结构中的 list 成员，用来将 example_struct 结构串成链表。可理解为 list_head“背负”的负载是 example_struct 结构。

（2）创建 list_head 结构。

使用前必须申请链表头并用 INIT_LIST_HEAD 宏来初始化链表头。可使用两种方法。

方法 1：

```
struct list_head example_list;
INIT_LIST_HEAD(&example_list);
```

方法 2：

```
LIST_HEAD(example_list);
```

其中，这两个宏在 include/Linux/list.h 中定义如下：

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

宏定义 INIT_LIST_HEAD 初始化了链表头，即向前、向后的指针都指向链表头。这样，就已初始化了一个 example_list 的链表头，以后就可以向链表中增加链表元素了。

（3）链表与用户结构连接。

list_entry 宏将 exmplelist 链表与 exmple_struct 结构类型连接起来，其原理如图 1.1 所示。

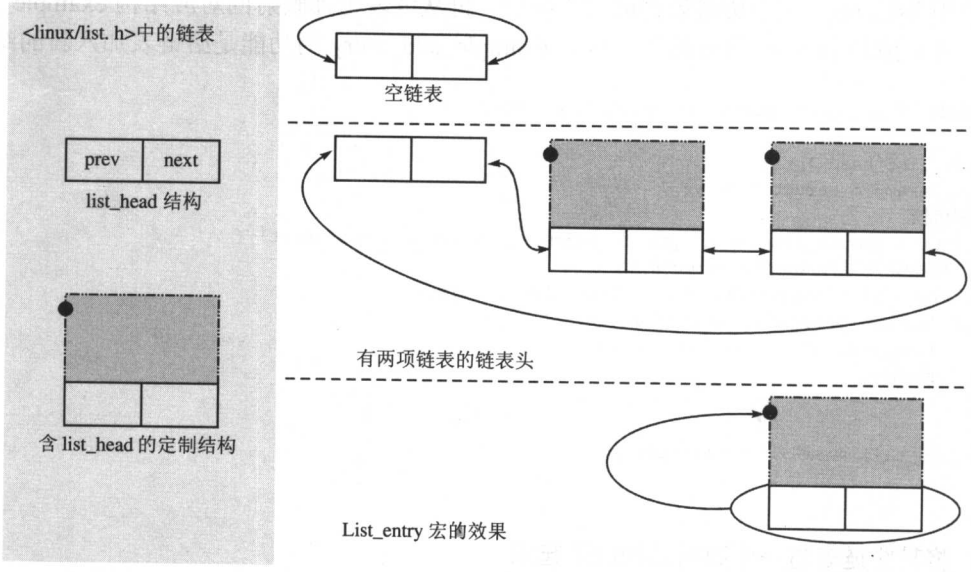

图 1.1　list_entry 宏及 struct list_head 链表的关系图

下面这个代码行就是从 examplelist 链表中得到节点对应的 example_struct 结构指针，其中 ptr 是 examplelist 链表中的指针，如 ptr = examplelist->next。

```
struct example_struct *node =
  list_entry(ptr, struct example_struct, list);
```

在上面代码行中的宏定义 list_entry 将一个 list_head 结构指针映射回一个指向结构 example_struct 的指针，即得到 list_head 的宿主结构。下面分析这个宏定义（在 include/Linux/list.h 中）：

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

list_entry 的功能是得到链表中节点的结构，它的参数含义为：

- ptr 是链表中的一个 struct list_head 结构元素指针。
- type 是用户定义的结构类型，其中，包含 struct list_head 结构成员。
- member 用户定义结构中的 struct list_head 结构成员名字。

在 include/Linux/kernel.h 中有 container_of 的定义，参数含义与 list_entry 中一致，container_of 得到 list 的容器结构，即含有 list 成员的结构 type。container_of 的定义如下：

```
#define container_of(ptr, type, member) ({              \
      //将链表中的元素ptr转换成结构type中成员member的类型
      const typeof( ((type *)0)->member ) *__mptr = (ptr);\
      //__mptr减去member成员偏移地址正好是type结构地址
      (type *)( (char *)__mptr - offsetof(type,member) );})
```

在 include/Linux/stddef.h 中有宏 offsetof 的定义，列出如下：

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

offsetof 宏对于上述示例的展开及分析是：&((struct example_struct *)0)->list 表示当结构 example_struct 正好在地址 0 上时其成员 list 的地址，即成员位移。

（4）遍历链表

下面使用 list_entry 宏遍历链表得到链表指针，再从链表指针映射回对应结构 example_struct 的指针。然后，对其成员 priority 进行操作。函数 example_add_entry 的功能是给链表加入新的结构成员。

```
void example_add_entry(struct example_struct *new)
{
  struct list_head *ptr;
  struct example_struct *entry;
   //遍历链表
  for (ptr = exmple_list.next; ptr != &exmple_list; ptr = ptr->next) {
     //映射回对应结构example_struct的指针
     entry = list_entry(ptr, struct todo_struct, list);
     if (entry->priority < new->priority) {
        list_add_tail(&new->list, ptr);
        return;
     }
  }
  list_add_tail(&new->list, &exmple_struct)
}
```

示例：将某些链表与一个数据结构联系起来。

函数 new_inode 为给定的超级块分配一个新的节点，并将新的节点加到链表 inode_in_use 和 sb->s_inodes 中，从而在两个链表中链接了新的节点。一个是以 inode_in_use 为链表头的全局的节点链表；一个是超级块结构为链表头的节点链表。其中，超级块与节点链表连接如图 1.2 所示。

```
fs/inode.c
extern struct list_head inode_in_use;
struct inode *new_inode(struct super_block *sb)
{
     static unsigned long last_ino;
     struct inode * inode;

     spin_lock_prefetch(&inode_lock);

     inode = alloc_inode(sb);
     if (inode) {
          spin_lock(&inode_lock);
          inodes_stat.nr_inodes++;
```