第十七章 最新进展

Linux 一直在迅速地发展着，开发人员总是迫切希望改善核心内部，它们并不考虑向后兼容性。这种自由开发导致了不同版本核心提供的设备驱动程序接口之间一定程度的不兼容。不过，在应用级还保持着兼容，除了个别需要与核心特征进行低级交互的应用（象 *ps*）。
另一方面，设备驱动程序是直接链接到核心映象上的，因此必须与数据结构、全局变量、以及由内核系统引出的函数发生的改变保持一致。在开发过程中，随着新特征的加入，内部被修改；新的实现取代了就的实现，因为实践证明它们更快，更清晰。尽管不兼容性要求程序员在写模块时要做一些额外的工作，我认为连续的开发是 Linux 社区的成功点：严格的先后兼容性最终证明是有害的。
这一章讲述 2.0.*x* 和 2.1.43 之间的不同，这些将会与即将推出的 2.2 发布类似。Linus 在前几个 2.1 版本中引入了最重要的改变，这样核心就可以多经历几个 2.1 版本，使得驱动程序的作者有足够的时间在开发被锁定以发布稳定的 2.2 之前来稳定驱动程序。下面的小节介绍驱动程序是如何处理 2.0 和 2.1.43 之间的不同的。我已经修改了本书介绍的所有示例代码，使得它们可以同时在 2.0 和 2.1.43 上编译和运行，以及这之间的大多数版本。
驱动程序的新版本可以从 O'Reilly 的 FTP 站点上在线例子的 v2.1 目录下得到。2.0 和 2.1 之间的兼容性通过头文件 *sysdep-2.1.h* 获得，它可以与你自己的模块集成。我选择不把兼容性扩展到 1.2 避免了给 C 代码加载太多的条件，而且 1.2-2.0 的不同已经在前面的章节解释过了。在我将要写完这本书时，我了解到从 2.1.43 起又引入了一些小的不兼容性；我不打算对之加以评述，因为我不能保证对这些最新版本的完全支持。
注意在本章我不会讲述 2.1 开发系列引入的所有新东西。我要做的只是移植 2.0 模块，使之可以在 2.0 和 2.1 核心上运行。利用 2.1 的特征意味着放弃对不具有这些特征的 2.0 发布的支持。2.0 版本仍是本书的重点。
在写 *sysdep-2.1.h* 时，我已努力使你熟悉新的 API，我引入的宏用来使 2.1 的代码可以在 2.0 上跑，而不是相反。
本章以重要性逐渐降低的顺序介绍不兼容性；最重要的不同首先被介绍，次要的细节则在后面介绍。

模块化

在 Linux 社区中，模块化变的越来越重要，开发人员决定用一个更清晰的实现取代旧的。头文件<linux/module.h>在 2.1.18 中完全重写了，一个新的 API 被引入。如你所期望的，新的实现比旧的要容易使用。
为了加载你的模块，你将需要包 *modutils-2.1.34* 甚至更新版本（细节见 *Documentation/Changes*）。当与旧的核心一起使用时，这个包可以回到兼容模式，因此你可以用这个新包替换 *modules-2.0.0*，即使你经常在 2.0 和 2.1 之间切换。

引出符号

符号表的新接口比以前的要容易多了，它依赖于下面的宏：
EXPORT_NO_SYMBOLS；

这个宏与 register_symtab(NULL)等价；它可以出现在一个函数的内部或外部，因为它只是指导汇编器，而不产生实际代码。如果你想在 Linux2.0 上编译模块，这个宏应该在 *init_module* 中被使用。

EXPORT_SYMTAB；

如果你打算引出一些符号，那么模块必须在包含<linux/module.h>之前定义这个宏。

EXPORT_SYMBOL(name)；

这个宏表明你想引出这个符号名。它必须在任何函数之外使用。

EXPORT_SYMBOL_NOVERS(name)

使用这个宏而不是 EXPORT_SYMBOL()强制丢弃版本信息，即使是编译带有版本支持的代码。这对避免一些不必要的重编译很有用。例如，*memset* 函数将总以同样的方式工作；引出符号而不带版本信息允许开发者改变实现（甚至使用的数据类型）而不需 *insmod* 标出不兼容性。在模块化的代码中不大可能需要这个宏。

如果这些宏都没有在你的源码中使用，那么所有的非静态符号都被引出；这与在 2.0 中一样。如果这个模块是从几个源文件生成的，你可以从任何源文件引出符号，而且还可以在模块的范围中共享任何符号。

如你所看到的，引出符号表的新方法解决了一些问题，但这个创新也引入了一个重要的不兼容性：一个引出了一些符号的模块，如果想同时在 2.0 和 2.1 上编译运行，则必须用条件编译来包含两个实现。下面是 *export* 模块（*v2.1/misc-modules/export.c*）如何处理这个问题的：

（代码 384 #1）

上面的代码依赖于下面 *sysdep-2.1.h* 中的行：

（代码 384 #2）

当使用 2.1.18 或更新的核心时，REGISTER_SYMTAB 扩展为什么都不做，因为 *init_module* 中没有什么需要做的；在函数外使用 EXPORT_SYMBOL 是引出模块符号唯一需要做的。

声明参数

核心模块的新的实现利用了 ELF 二进制格式的特征以获得更好的灵活性。更特别地，当构造一个 ELF 目标文件时，你可以声明除“正文”、“数据”和“bss”之外的节。一个“节”是一个连续的数据区域，与“段”的概念类似。

对于 2.1，核心模块必须使用 ELF 二进制格式编译。事实上，2.1 核心利用了 ELF 的节（见“处理核心空间错误”），只能编译为 ELF。因此模块的限制并不是个真正限制。使用 ELF 允许信息域被存在目标文件中。好奇的读者可以使用 *objdump –section-headers* 来观察节头，用 *objdump –section=.modinfo –full-contents* 来查看模块特定的信息。实际上，.modinfo 一节是用来存储模块信息的节，包含被称做“参数”的值，可以在加载时修改。

当在 2.1 上编译时，一个参数可以用宏如下声明：

MODULE_PARM(variable, type-description)；

当你在源文件中使用这个宏时，编译器被告知在目标文件中插入一个描述串；这个描述表明 variable 是个参数，它的类型对应于 type-description。*insmod* 和 *modprobe* 查看目标文件，保证你被允许修改 variable，同时检查参数的实际类型。类型检查对防止不愉快的错误非常重要，例如用一个串覆盖了一个整数，或错把长整数当成了短整数。

按我的观点，讲述宏的最好办法时给出几行示例代码。下面的代码属于一个想象的网卡：

（代码 385）

type-description 串在头文件<linux/module.h>中被非常详细地介绍，并且为了你的方便，它可以在整个核心源码中找到。

值得给出的一个技巧是如何参数化一个数组的长度，象上面的 io。例如，设想网络驱动程序支持的外围板子的数目有宏 MAX_DEVICES 表示，而不是硬写入的数字 4。出于这个目的，头文件<linux/module.h>定义了一个宏（__MODULE_STRING），它用 C 预处理器将一个宏

“字符串化”。这个宏可以如下使用：

```
int io[MAX_DEVICES+1]={0,};
MODULE_PARM(io, “1-” __MODULE_STRING(MAX_DEVICES) “i”);
```

在前一行中，被“字符串化”的值与其他串接在一起构成目标文件中有意义的串。

scull 示例模块也用 MODULE_PARM 来声明它的参数（scull_major 和其他整数变量）。这在 Linux2.0 上编译时可能会出问题，那里这个宏未定义。我选择的简单的修正是在 *sysdep-2.1.h* 中定义 MODULE_PARM，这样在与 2.0 头文件编译时，它扩展为空语句。

其它有意义的值可以象 MODULE_AUTHOR()一样 存在模块的.*modinfo* 一节，但它们目前没有使用。请参考<linux/module.h>以获得更多的信息。

/proc/modules

/proc/modules 的格式在 2.1.18 中略有改变，而所有的模块化代码都被重写了。尽管这个改变并不影响源码，你可能对其细节不感兴趣，因为*/proc/modules* 在模块开发时经常被检查。

新格式和旧的一样是面向行的，每行包含下面的域：

模块名

这个域与 Linux2.0 相同。

模块大小

这是个十进制数，以字节为单位（而不是内存页）报告长度。

这个模块的使用计数

如果模块没有使用计数，这个计数报告-1。这是和新的模块化代码一道引入的新特征；你可以写一个模块，它的去除可以有一个函数控制而不是使用计数。这个函数判断模块是否能够被卸载。例如，*ipv6* 模块就使用这个特征。

可选标志

标志是文本串，每个都由括号包含，并由空格分隔。

参考本模块的模块列表

这个列表整体被包含在方括号内，表中的单个名字由空格隔开。

下面是*/proc/modules* 在 2.1.43 中的可能内容：

```
morgana% cat /proc/modules
ipv6                   57164   -1
netlink                 3180    0  [ipv6]
floppy                 45960    1    (autoclean)
monitor                  516    0    (unused)
```

在这个屏幕快照中，ipv6 没有使用计数，并依赖于 netlink；floppy 已经被 *kerneld* 加载，由“autoclean”标志给出，monitor 是我的一个小工具，控制一些状态灯，并在系统终止时关掉我的计算机。如你所看到的，它是“unused”，我并不关心它的使用计数。

文件操作

有几个文件操作在 2.1 里与 2.0 有不同的原型。这主要是出于处理大小不能放入 32 位的文件的需要。其不同由头文件 *sysdep-2.1.h* 处理，它根据使用的核心版本定义了几个伪类型。文件操作中引入的仅有的显著创新是 *poll* 方法，它用完全不同的实现代替了 *select* 方法。

原型的不同

四个文件操作表征一个新的原型；它们是：

```
long long (*llseek) (struct inode *, struct file *, long long, int);
long (*read) (struct inode *, struct fle *, char *, unsigned long);
long (*write) (struct inode *, struct file *, const char *, unsigned long);
int (*release) (struct inode *, struct file *);
```

它们在 2.0 中的对应者是：

```
int (*lseek) (struct inode *, struct file *, off_t, int);
int (*read) (struct inode *, struct file *, char *, int);
int (*write) (struct inode *, struct file *, const char *, int);
void (*release) (struct inode *, struct file *);
```

如你所见的，其不同在于它们的返回值（它允许了更大的范围），还有 count 和 offset 参数。头文件 *sysdep-2.1.h* 通过定义下面的宏处理这些不同：

read_write_t

这个宏扩展为参数 count 的类型以及 *read* 和 *write* 的返回值。

lssek_t

这个宏扩展为 *llseek* 的返回值类型。方法名字的改变（从 *lseek* 到 *llseek*）并不是个问题，因为你一般在 file_operations 中并不用名字对域赋值，而是声明一个静态结构。

lseek_off_t

lseek 的 offset 参数。

release_t

release 方法的返回值；或为 void 或为 int;

release_return(int return_value)

这个宏可以用来从 *release* 方法返回。它的参数用来返回一个错误代码：0 表示成功，负值表示失败。在比 2.1.31 老的核心中，这个宏扩展为 return，因为这个方法返回 void。

用前面的宏，一个可移植的驱动程序原型是：

```
lseek_t my_lseek(struct inode *, struct file *, lseek_off_t, int);
read_write_t my_read(struct inode *, struct file *, char *, count_t);
read_write_t my_write(struct inode *, struct file *, const char *, count_t);
release_t my_release(struct inode *, struct file *);
```

poll 方法

2.1.23 引入了 *poll* 系统调用，它是 system V 中 *select* 的对应者（由 BSD Unix 引入）。不幸的是不可能在 *select* 设备方法之上实现 *poll* 的功能，所以整个实现用不同的一个代替，它作为 *select* 和 *poll* 的后端。

在当前版本的核心中，file_operations 中的设备方法也叫 *poll*，与系统调用类似，因为其内部模仿这个系统调用。这个方法的原型是：

```
unsigned int (*poll) (struct file *, poll_table *);
```

驱动程序中设备特定的实现主要完成两个任务：

- 在一个可能在将来唤醒它的等待队列中将当前进程排队。通常，这意味着同时在输入和输出队列中对进程排队。函数 *poll_wait* 被用于这个目的，其工作方式与 *select_wait* 非常类似（细节请看第五章“增强的字符设备驱动程序操作”中“*select*”一节）。
- 构造一个位掩码描述设备的状态，并将其返回给调用者。这些位的值是平台特定的，在<linux/poll.h>中定义，它必须被包含在驱动程序中。

在讲述位掩码的每一位前，我想给出一个典型的实现。下面的函数是 *v2.1/scull/pipe.c* 的一部分，是*/dev/scullpipe* 的 *poll* 方法的实现。*scullpipe* 的内部在第五章介绍过。
（代码 389）
如你所看到的，这个代码相当简单。它比对应的 *select* 方法要容易。至于 *select*，状态位计为“可读”、“可写”，或“发生例外”（这是 *select* 的第三个条件）。
poll 各位的完全列表在下面给出。“输入”位列在前面，然后是“输出”，一个“例外”位列在最后。

POLLIN
: 如果设备可以被无阻塞地读，那么这个位必须被设置。

POLLRDNORM
: 如果“一般”数据可以被读，这个位必须被设。一个可读设备返回（POLLIN | POLLRDNORM）。

POLLRDBAND
: 在目前的核心源码中这个位不被使用。Unix System V 使用这个位报告非 0 优先级的数据可读。数据优先级的概念与“Streams”包相关。

POLLHUP
: 当一个读设备的进程看到文件结尾时，驱动程序必须设置 POLLHUP(挂起)。一个调用 *select* 的进程将被告知设备可读，这由 *select* 的功能说明。

POLLERR
: 设备上发生了一个错误条件。当 *poll* 被 *select* 系统调用调用时，设备被报告为既可读又可写，因为 *read* 或 *write* 将无阻塞地返回一个错误代码。

POLLOUT
: 如果设备可以被无阻塞地写，这个位在返回值中被设置。

POLLWRNORM
: 这个位与 POLLOUT 有相同的含义，有时甚至的确为同一个数。一个可写的设备返回（POLLOUT | POLLWRNORM）。

POLLWRBAND
: 与 POLLRDBAND 类似，这个位意味着非 0 优先级的数据可以被写到设备。只有 *poll* 的“数据报”实现用到着位，因为一个数据报可以传送“无团队数据（out-of-band data）”。*select* 报告设备是可写的。

POLLPRI
: 高优先级的数据（“无团队的”）可以被无阻塞地读取。这个位导致 *select* 报告文件上发生了一个例外条件，因为 *select* 将无团队包作为一个例外条件报告。

poll 的主要问题是它与 2.0 核心所使用的 *select* 方法没有任何关系。因此，处理这个不同的最好方法是使用条件编译来编译合适的函数，而且同时将它们都包含在源文件中。
如果当前版本支持 *select* 而不是 *poll*，那么头文件 *sysdep-2.1.h* 定义符号 __USE_OLD_SELECT__。这将你从在源码中必须引用 LINUX_VERSION_CODE 中解脱出来。v2.1 目录下的示例驱动程序使用了与下面类似的代码：
（代码 390）
（代码 391）
这两个函数用同样的名字调用，因为在结构 sample_fops 中 sample_poll 被引用，那里 *poll* 文件操作代替了 *select* 方法。

访问用户空间

核心的第一个 2.1 版引入了一种从核心代码访问用户空间的新（更好）方法。这个改变修正了一个长期存在的错误行为并增强了系统的性能。

当你位核心 2.1 编译代码，并需要访问用户空间时，你需要包含<asm/uaccess.h>，而不是<asm/segment.h>。你还必须使用一个与 2.0 不同的函数集。不用说，头文件 *sysdep-2.1.h* 尽可能地照顾了这些不同，允许你在 2.0 上编译时使用 2.1 的语义。

在用户访问中最令人注意的不同时 *verify_area* 没有了，因为多数验证都由 CPU 完成了。关于这个主题的细节见本章后面的“处理核心空间错误”。

可被用来访问用户空间的新的函数集是：

int access_ok(int type, unsigned long addr, unsigned long size);

如果当前进程被允许访问地址 addr 处的内存，函数返回真（1），否则为假（0）。这个函数取代 *verify_area*，尽管它进行较少的检查。和老的 *verify_area* 接收一样的参数，但是要快的多。在你复引用一个用户空间地址之前，这个函数应该被调用对之进行检查；如果你没有检查，用户有可能会访问和修改核心内存。本章后面的“虚拟内存”一节更细致地解释了这个问题。幸运的是，下面描述的大多数函数都替你进行了这个检查，因此你实际上并不需要调用 *access_ok*，除非你选择这样做。

int get_user(lvalue, address);

在 2.1 核心中使用的宏*get_user*与我们在 2.0 中使用的并不相同。其返回值在成功时为 0，否则为一个负的错误代码（总是-EFAULT）。这个函数的净效果是将从地址address取得的数据赋给*lvalue*。在通常的C语言含义中，这个宏的第一个参数必须是一个*lvalue*[*]。与 2.0 版中的这个函数类似，数据项的实际大小依赖于address参数类型。这个函数在内部调用*access_ok*。

int __get_user(lvalue, address);

这个函数完全类似 *get_user*，但它不内部调用 *access_ok*。当你访问一个已经从同一核心函数内部检查过的用户地址时，你应该调用*__get_user*。

get_user_ret(lvalue, address, retval);

这个宏是调用 *get_user* 的快捷方式，如果函数失败则返回 retval。

int put_user(expression, address);

int __put_user(expression, address);

put_user_ret(expression, address, retval);

这些函数与它们的 *get_*对应者非常类似，只是它们是向用户空间写，而不是读。成功时，值 expression 被写到地址 address。

unsigned long copy_from_user(unsigned long to, unsigned long from, unsigned long len);

这个函数从用户空间复制数据到核心空间。它代替旧的 *memcpy_tofs* 调用。这个函数内部调用 *access_ok*。返回值是未能传送的字节数。这样，如果发生错误，返回值必然大于 0；在那种情况下，驱动程序返回-EFAULT，因为错误是由错误的内存访问引起的。

unsigned long __copy_from_user(unsigned long to, unsigned long from, unsigned long len);

这个函数与 *copy_from_user* 一样，但它不内部调用 *access_ok*。

caopy_from_user_ret(to, from, len, retval);

这个宏是内部调用 *copy_from_user* 的快捷方式；如果失败，则从当前函数返回。

unsigned long copy_to_user(unsigned long to, unsigned long from, unsigned long len);

---

[*] 一个*lvalue*是一个可以作为赋值的左操作数的表达式。例如，count, v[34+check()], 和*((prt+offset)->field)是lvalue；i++,32, 和cli()都不是。

unsigned long __copy_to_user(unsigned long to, unsigned long from, unsigned long len);
copy_to_user(to, from, len, retval);

这些函数被用来将数据复制到用户空间，它们的行为非常类似于它们的 *copy_from* 的对应者。

2.1 版核心还定义了其它访问用户空间的函数：*clear_user*，*strncpy_from_user*，和 *strlen_user*。我不打算讨论它们了，因为 Linux2.0 中没有这些函数，并且驱动程序的代码也很少用到它们。有兴趣的读者可以看看<asm/access.h>。

使用新的接口

访问用户空间的新的函数集初看起来可能有点令人失望，但它们的确使程序员的日子好过的多了。在 Linux2.1 上，不再需要显式地检查用户空间；*access_ok* 一般不需要调用。使用新接口的代码可以直接进行数据传送。_ret 函数在实现系统调用时证明是相当有用的，因为一个用户空间的失败通常导致系统调用的一个返回-EFAULT 的失败。

因此，一个典型的 *read* 实现，看起来如下：

```
long new_read(struct inode *inode, struct file *filp, char *buf, unsigned long count);
{
/* identify your data (device-specific code) */
if (__copy_to_user(buf, new_data, count))
    return -EFAULT;
return count;
}
```

注意使用不进行检查的*__copy_to_user* 是因为调用者在把数据传输分派到文件操作之前已经检查了用户空间。这就象 2.0，*read* 和 *write* 不需要调用 *verify_area*。

类似地，典型的 *ioctl* 实现看起来如下：

```
int new_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
{
/* device-specific checks, if needed */
switch(cmd){
    case NEW_GETVALUE:
        put_user_ret(new_value, (int *)arg, -EFAULT);
        break;
    case NEW_SETVALUE:
        get_user_ret(new_value, (int *)arg, -EFAULT);
    default:
        return –EINVAL;
    }
return 0;
}
```

于版本 2.0 的对应者不同的是，这个函数在 switch 语句之前并不需要检查参数，因为每个 *get_user* 或 *put_user* 会进行检查。另一种实现方式如下：

（代码 394 #2）

另一方面，当你想写可以同时在 2.0 和 2.1 上编译的代码时，问题变得稍微复杂一些，因为在老的核心上，你不能用 C 预处理器伪装新的行为。你不能简单地#define 一个接收两个参数的 *get_user* 宏，因为实际的 *get_user* 实现在 2.0 中已经是个宏。

我在写既可移植有高效率的代码的选择是设置 *sysdep-2.1.h* 以提供具有下列函数的源码。下面只列出了读取数据的函数；写数据的函数行为完全一样。

int access_ok(type, address, size);

当在 2.0 上编译时，这个函数以 *verify_area* 的名义实现。

int verify_area_20(type, address, size);

通常，当为 Linux2.1 写代码时，你不需调用 *access_ok*。另一方面，当在 Linux2.0 上编译时，则必须调用 *verify_area*。这个函数就是要填平这个不同：当为 Linux2.1 编译时，它扩展为空；而为 2.0 编译时，它扩展为原来的 *verify_area*。这个函数不能被称做 *verify_area*，因为 2.1 已经有一个宏叫这个名字了。在 2.1 中定义的 *verify_area* 宏实现了 a*ccess_ok* 的老的语义，它的存在是为了简化源码从 2.0 到 2.1 的转换。（从理论上说，你可以在你的模块中留下 *verify_area*，只是将函数名改一下；这种简单移植技巧的缺点是新版本不能在 2.0 上编译。）

int GET_USER(var, add);

int __GET_USER(var, add);

GET_USER_RET(var, add, ret);

当在 2.1 上编译时，这些宏扩展为实际的 *get_user* 函数，即上面解释过的那些。当在 2.0 上编译时，*get_user* 的 2.0 实现被用来实现与 2.1 中同样的功能。

int copy_from_user(to, from, size);

int __copy_from_user(to, from, size);

copy_from_user_ret(to, from, size);

当在 2.0 上编译时，这些扩展为 *memcpy_fromfs*；而在 2.1 上，则使用本身的函数。*_ret* 一类在 2.0 上从不会返回，因为复制函数不会失败。

我个人比较喜欢这种实现兼容性的方法，但这并不是唯一的方法。在我的示例代码中，任何用户空间的访问（除了用来 *read* 或 *write* 的缓冲区，它们已经事先检查过了）之前，*verify_area_20* 必须被调用。另一种方法更加忠实于 2.1 的语义，即当用 2.0 时，在每个 *get_user* 和 *copy_from_user* 之前自动生成一个 *verify_area*。这个选择在源码级要更清晰一些，但在版本 2.0 上编译时效率相当低，包括代码大小和执行时间。

可以同时在 2.0 和 2.1 上编译的示例代码，如 *scull* 模块，可以在目录 *v2.1/scull* 中找到。我不觉得这个代码足够有趣，因此不在这里给出。

任务队列

从 2.1.30 开始的 Linux 版本不再定义函数 *queue_task_irq* 和 *queue_task_irq_off*，因为在 *queue_task* 上的实际加速不值得花精力维护两个独立的函数。当新机制被加到核心时，这就变得明显了。

在源码级，这是 2.0 和 2.1 之间唯一的区别；头文件定义了消失的函数简化了从 2.0 移植驱动程序。感兴趣的读者可以查看<asm/spinlock.h>以获得更多的细节。

中断管理

在 2.1 的开发中，有些 Linux 内部被修改了。新核心提供了对内部锁的很好的管理；通过使用几个细粒度的锁，而不是全局的锁，竞争条件被避免了，这样也获得了更好的性能----特别是 SMP 配置下。

更细的锁机制的一个结果是 intr_count 不再存在了。2.1.34 抛弃了这个全局变量，而布尔函数 *in_interrupt* 可以取而代之（这个函数从 2.1.30 开始存在）。目前，*in_interrupt* 是在头文件 <asm/hardirq.h>中声明的宏，这个头文件又包含在<linux/interrupt.h>中。头文件 *sysdep-2.1.h* 用 intr_count 的名义定义了 *in_interrupt* 以获得对 2.0 的向后兼容性。
注意虽然 *in_interrupt* 是个整数，intr_count 却是个 unsigned long，因此，如果你想打印这个值，并在 2.0 和 2.1 间可移植，你必须强制将这个值转换为一个显式的类型，并在调用 *printk* 时指定一个合适的格式。
在 2.1.37 中中断管理又引入了一个不同：快和慢中断处理程序不再存在了。SA_INTERRUPT 不被新版本的 *request_irq* 使用，但它在处理程序执行以前仍然控制着中断是否被打开。如果几个处理程序共享一个中断线，每个可以是个不同的“类型”。中断打开与否依赖于第一个被调用的处理程序。当中断处理程序存在时，下半部总是执行。

位操作

2.1.37 稍微改变了在<asm/bitops.h>中定义的位操作的作用。现在函数 *set_bit* 及其相关者返回 void，而新的类似 *test_and_set_bit* 的函数已被引入。新的函数集有如下原型：

```
void set_bit(int nr, volatile void * addr);
void clear_bit(int nr, volatile void * addr);
void change_bit(int nr, volatile void * addr);
int test_and_set_bit(int nr, volatile void * addr);
int test_and_clear_bit(int nr, volatile void * addr);
int test_and_change_bit(int nr, volatile void * addr);
int test_bit(nr, addr);
```

如果你想获得与 2.0 的后向兼容性，你可以在你的模块中包含 *sysdep-2.1.h*，并使用新的原型。

转换函数

版本 2.1.10 引入了一个新的转换函数，在<asm/byteorder.h>中声明。这些函数可以用来访问多字节值，只要这个值已知是小印地安字节序或大印地安字节序。因为这些函数为写驱动程序代码提供了很好的快捷方式，头文件 *sysdep-2.1.h* 在较早的版本就已经定义了它们。
2.1 核心源码提供的本身实现比 *sysdep-2.1.h* 提供的可移植的实现要快，因为它可以利用体系相关的功能。
新函数对应下面的原型，其中 le 表示小印地安字节序，be 表示大印地安字节序。注意编译器并不强制严格的数据类型化，因为大多数函数都是预处理宏；下面给出的类型仅供参考。

```
__u16 cpu_to_le16(__u16 cpu_val);
__u32 cpu_to_le32(__u32 cpu_val);
__u16 cpu_to_be16(__u16 cpu_val);
__u32 cpu_to_be32(__u32 cpu_val);
__u16 le16_to_cpu(__u16 le_val);
__u32 le32_to_cpu(__u32 le_val);
__u16 be16_to_cpu(__u16 be_val);
__u32 be32_to_cpu(__u32 be_val);
```

这些函数在处理二进制数据流时很有用（例如文件系统数据或存在接口板中的信息），版本 2.1.43 又增加了两个新的转换函数集。这些集允许你用指针获取一个值，或是对参数指定的

一个值进行就地转换。对应与 16 位小印地安字节序的函数又如下的原型；类似的函数对其它类型的整数也存在，导致一共 16 个函数。

__u16 cpu_to_le16p(__u16 *addr)
__u16 le16_to_cpup(__u16 *addr)
void cpu_to_le16s(__u16 *addr)
void le16_to_cpus(__u16 *addr)

“p”函数类似与指针的复引用，但在需要时转换这个值；“s”函数可以在原地转换一个值的印地安字节序（例如，cpu_to_le16s(addr) 和 addr=cpu_to_le16(*addr)完成的工作是一样的）。这些函数也在 *sysdep-2.1.h* 中定义了。为了避免双重解释的副作用，这个头文件用线入函数，而不是预处理宏。

vremap

在第七章“把握内存”中“*vmalloc* 和朋友们”一节描述的 *vremap* 函数在版本 2.1 中得到一个新名字。新函数 *ioremap* 只是名字变了，它与旧的 *remap* 取同样的参数。响应的释放函数是 *iounmap*，它代替 *vfree* 来释放被重映射的地址。

这个改变是为了明确这个函数的实际作用：将 I/O 空间重映射到核心空间的一个虚地址。头文件 *sysdep-2.1.h* 强化了这种新规则，当在 2.0 版本编译时，它#define 了 *ioremap* 和 *iounmap* 到它们 2.0 的对应者。

虚拟内存

在核心的版本 2.1，Linux 的 Intel 移植对虚拟内存有了一个成熟的视图。早些的版本的内存管理一直使用“分段”的方法，这是从核心生命期的开始时期继承下来的。这个改变并不影响驱动程序代码，但不管怎样，还是值得一说的。

新的规则与 Linux 的其它移植匹配的起来。虚拟地址空间被构造成核心居于非常高的地址（从 3GB 往上），而用户地址空间在 0-3GB 范围。当一个进程运行在“管态”时，它可以访问两个空间。另一方面，当它运行在“用户态”时，它不能访问核心空间，因为属于核心的页被标记为“管理员”页，处理器阻止了对它们的访问。

这种内存布局有助于取消旧的 *memcpy_to_fs* 一类的函数，因为已经没有 FS 段了。核心空间和用户空间使用同一个“段”，其区别在于 CPU 所在的优先级。

处理核心空间错误

Linux 核心的 2.1 版本对从核心空间处理段错误的能力有一个极大的增强。本章里，我准备对其原则给一个快速的概述。新机制对源码的影响在“访问用户空间”中已经描述过。

如前面所提到过的，核心的最近版本充分利用了 ELF 二进制格式，特别是考虑到它的在编译的文件中定义用户定义的节的能力。编译器和链接器保证属于同一节的代码段在可执行文件中一定是连续的，因此当文件被装载时，在内存中也是连续的。

例外处理是通过在核心可执行映象(*vmlinux*)中定义两个新节实现的。每次当源码通过 *copy_to_user*, *put_user*, 或其读取的对应者访问用户空间时，一些代码被加到这两个节中。尽管这看起来是不可忽略的开销，这个新机制的一个结果是不再需要使用一个昂贵的 *verify_area*。而且，如果使用的用户地址是正确的，计算流将不会有一个跳转。

当被访问的用户地址是无效的时，硬件发出一个页面错。错误处理程序（在体系结构特定的

源码树中的 *do_page_fault*）确认这个错误是一个“不正确的地址”错（与“页不存在”相对），并使用下面的 ELF 节进行适当的动作：

__ex_table

这节是个指针对的表。每对的第一个指针指向一个可能因错误的用户空间地址而失败的指令，第二个值指向一个地址，处理器将在那里找到几条的指令来处理这个错误。

.fixup

这节包含指令，处理在__ex_table 中描述的所有可能的错误。这个表中每对的第二个指针指向居于.fixup 中的代码。

头文件<asm/uaccess.h>负责构造所需的 ELF 节。访问用户空间的每个函数（如 *put_user*）扩展为汇编指令，它将指针加到__ex_table 并处理.*fixup* 中的错。

当代码运行时，实际的执行路径有一下步骤组成：用于函数“返回值”的处理器寄存器被初始化为 0（也就是没有错误），数据被传送，返回知被传回调用者。一般的操作的确非常快。如果一个异常发生，*do_page_fault* 打印一条消息，查看__ex_table，跳转到.*fixup*，这里设置返回值为-EFAULT，然后跳转到访问用户空间的指令后位置。

新的行为可以用 *faulty*（在 *v2.1/misc-modules* 目录）模块来检验。*faulty* 在第四章“调试技巧”中“调试系统错误”一节描述。*faulty* 的设备结点通过读取一个短缓冲区界外来传送数据到用户空间，这样当读取一个在模块页以上的地址时，会导致一个页面错。有趣的是注意到这个错误依赖于使用核心空间中的一个不正确地址，而大多数情况下异常是有出错的用户空间地址造成的。

当在 PC 上用 *cat* 命令读 *faulty* 时，下面的消息被打印在控制台上：

```
read: inode c1188348, file c16decf0, buf 0804cbd0, count 4096
cat: Exception at [<c2807b7>](c2807115)
```

前一行是由 *faulty* 的 *read* 方法打印的，而后者是由错误处理程序打印的。第一个数字是错误指令的地址，而第二个是修正代码（在.*fixup* 节中）的地址。

其它改变

在 2.0 和 2.1.43 之间还有其它一些不同。以我的观点，它们不需要给予特别的关注，因此我将迅速地概述一下。

proc_register_dynamic 在 2.1.29 中消失了。最近的核心对每个*/proc* 文件使用 *proc_register* 接口；如果结构 proc_dir_entry 的 low_ino 域是 0，那么会被分配一个动态的 inode 号。当为 2.1.29 或更新的核心编译时，头文件 *sysdep-2.1.h* 象 *proc_register* 一样定义 *proc_register_dynamic*；这个在注册的 proc_dir_entry 结构以 0 为 inode 号时是可行的。

在网络接口驱动程序领域，*rebuild_header* 设备方法从 2.1.15 起有一个新的原型。如果你只开发以太网驱动程序，你不会关心这个不同，因为以太网驱动程序不实现它们自己的方法；它们依赖于通用的以太网实现。当旧的实现需要时，头文件 sysdep-2.1 定义了宏__USE_OLD_REBUILD_HEADER__。示例模块 *snull* 显示了如何使用这个宏，但每必要在这里给出。

网络代码的另一个改变影响了结构 enet_statistics，它从 2.1.25 起不再存在。代替它的是一个新结构 net_device_stats，它在<linux/netdevice.h>中定义，而不是<linux/if_ether.h>。新结构与旧结构类似，但是多了两个域存储字节计数器：unsigned long rx_bytes, tx_bytes；一个全特征的网络接口驱动程序应该与 rx_packets 和 tx_packets 一道增加这些计数器，尽管一个快速的计划可能要抛弃这些计数器。核心头文件将 enet_statistics（老结构的名字）定义为

net_device_stats（新结构的名字）以方便已有驱动程序的可移植性。

最后，我需要指出 current 不再是个全局变量 x86, Alpha,以及 Sparc 的核心移植使用了聪明的技巧将 current 存在处理器中。这样核心的开发者努力又挤出了几个 CPU 周期。这个技巧避免了大量的内存访问，有时还能释放一个通用目的寄存器；编译器经常分配处理器寄存器来高速缓存几个经常访问的内存位置，而 current 是经常访问的。在不同的移植中使用了不同的技巧以优化访问。Alpha 和 Sparc 版本使用一个处理器寄存器（编译器优化不使用的一个）来存储 current。而 Intel 处理器有有限数目的寄存器，编译器可以使用它们所有；在这种情况下技巧包括将结构 task_struct 和核心栈页存储在连续的虚存页内。这允许 current 指针被“编码”在栈指针中。对每个 Linux 支持的平台，头文件<asm/current.h>给出了实际选择的实现。

象所有重要的软件一样，Linux 一直在改变着。如果你想为这个最新的、最伟大的核心写驱动程序，你需要保持跟上核心的发展。尽管处理不兼容性看起来可能很困难，我们发现两点特性：首先，主要的程序设计技巧一直在那里，不太可能改变（至少不常）；第二，每次改变都变得更好了，经常使你在将来的开发中需要的工作越来越少。