第 6 章 信号量，中断和时间

信号量（Signal）是进程间通讯（IPC）的一种形式——是一个进程给另一个进程发送信息的方法。但是信息不可能很多——一个信号量不可能携带详细的信息，即使是传送者的身份也不能被传递；唯一能够确定的事实是信号量的确被发送了。（然而和经典信号量不同，POSIX 实时信号量允许传送稍微多一点的信息。）实际上，信号量对于双向通讯是没有用处的。还有，根据某些限定，信号量的接受者不必以任何方式作出响应，甚至可以直接忽略大部分信号量。

虽然有这么多的限制，然而信号量仍然是一种功能强大的十分有用的机制——勿庸置疑，这是 Unix IPC 中使用最频繁的机制。每当进程退出或者废弃一个空指针时，每当使用 Ctrl+C 键终止程序运行时，都要传递信号量。

第 9 章会更详细的讨论 IPC 机制。对于本章的讨论来说，信号量的内容就足够讨论了。

正如在 Linux 内核本身的代码注释中所说明的一样，中断（Interrupt）对于内核来说和信号量是类似的。中断一般都是从磁盘之类的硬件设备送往内核，用以提示内核该设备需要加以注意。一个重要的硬件中断源就是定时器设备，它周期性地通知内核已经通过的时间。如同第 5 章中阐述的一样，中断也可以由用户进程通过软件产生。

在本章中，我们首先讨论一下 Linux 中信号量和中断的实现，最后再浏览一下 Linux 的时间处理方式。

虽然内核对代码的要求标准非常严格，本章所涉及的代码仍然特别清晰明白。本章使用的一般方法是首先介绍相关的数据结构和它们之间的关系，接下来讨论操纵和查询它们的函数。

锁的概述

锁的基本思想是限制对共享资源的访问——共享资源包括共享的文件，共享的内存片，以及在一次只能由一个 CPU 执行的代码段。概括的说，在单处理器上运行的 Linux 内核并不需要锁，这是因为在编写 Linux 内核时就已经注意到要尽量避免各种可能需要锁的情况了。但是，在多处理器机器上，一个处理器有时需要防止其它处理器对它的有害的介入。

include/asm-i386/spinlock.h 文件（从 12582 行开始）并不使用难看的**#ifdef** 把所有对锁函数的调用封装起来，它包含一系列对单处理器平台（UP）基本为空的宏，然而在多处理器平台（SMP）上这些宏将展开成为实际代码。因而内核的其它代码对 UP 和 SMP（当涉及到这种特性时）都是相同的，但是它们两个的效果却是迥然不同的。

第 10 章中涉及 SMP 的部分会对锁做深入的介绍。但是，由于你在代码中将到处都能够看到对锁宏的调用，特别是在本章所讨论到的代码中这一点尤为明显，所以你应该首先对宏的用途有初步了解——以及为什么现在在大多数情况下我们都可以安全地将其忽略（我们将在讨论的过程中对其中的异常情况进行说明）。

信号量

Linux 内核将信号量分为两类：

- 非实时的（Nonrealtime）——大部分是些传统的信号量，例如 **SIGSEGV**，**SIGHUP**

和 **SIGKILL**。

- 实时的（realtime）——由 POSIX 1003.1b 标准规定，它们同非实时信号量有细微的区别。特别是实时信号量具有进程可以配置的意义——就像是非实时信号量 **SIGUSR1** 和 **SIGUSR2** 一样——额外的信息能够和这些信号量一起传送。它们也会排队，因此如果在第一个信号量处理完成之前有多个信号量实例到达，所有的信号量都能够被正确传送；这对于非实时信号量则是不可能的。

在第 7 章中我们将会对实时性对于 Linux 内核的意义进行更详细的介绍——特别是实时性所不能够说明的内容。

信号量数目的宏定义从 12048 行开始。实时信号量的数目在 **SIGRTMIN** 和 **SIGRTMAX**（分别在 12087 行和 12088 行）所定义的范围之内。

数据结构

本节讨论信号量代码使用的最重要的数据结构。

sigset_t

12035：**sigset_t** 表示信号量的集合。根据使用地点的不同，它的意思也不同——例如，它可能记录着正在等待某一个进程的信号量（如 16425 行 **struct task_struct** 的 **signal** 成员）的集合，也可能是某个进程已经请求阻塞了的信号量（如同一行中定义的同一结构的 **blocked** 成员）的集合。随着本书的进行，我们会逐渐看到这些类似的应用。

12036：**sigset_t** 的唯一一个组成部分是一组 **unsigned long**（无符号长整型数），其中的每一位都代表一个信号量。注意到无符号长整型类型在整个内核代码中是作为一个字来处理的，这和你所希望的可能有所出入——即使是在当前 x86 CPU 的讨论中，有时候字也被用于说明 16 位类型。由于 Linux 是一个真 32 位操作系统，将 32 位看作是一个字在绝大多数情况下是正确的。（将 Linux 称为真 32 位操作系统也有一些不准确，因为在 64 位 CPU 上它也是一个真 64 位操作系统。）

这个数组的大小 **_NSIG_WORDS** 在 12031 行直接计算。（**_NSIG_BPW** 中的‘ BPW ”是“ bits per word（每字位数）”的缩写。）在不同的平台上，**_NSIG_WORDS** 的大小从 1（Alpha 平台中）到 4（MIPS 平台中）不等。如你所见，在 x86 平台中，该值正好是 2，这意味着在 x86 平台上 2 个无符号数就可以包含足够的位数来代表所有 Linux 使用的信号量。

struct sigaction

12165：**struct sigaction** 代表信号量到达时进程应该执行的动作。它被封装在 **struct k_sigaction**（12172 行）结构中，而该结构又是被封装在 **struct signal_struct** 结构中的，后者是 **struct task_struct** 结构的 **sig** 成员所指向的一个实例（16424 行）。如果这个指针为空，进程就会退出而不必接受任何信号量。否则，每个进程对于每个信号量数目都需要若干 **_NSIG struct sigaction** 结构和一个 **struct sigaction** 结构。

12166：**sa_handler**（**__sighandler_t** 类型——一个在 12148 行定义的函数指针类型）描述了进程希望处理信号量的方式。其值可以是下面中的一个：

- **SIG_DFL**（12151 行）申请处理信号量的缺省操作，不管该操作是什么——这是由信号量所决定的。注意它和 **NULL** 是等同的。
- **SIG_IGN**（12153 行）意味着信号量应该忽略。但是，并不是所有的信号量都可以被忽略的。
- 所有的其它值都是在信号量到达时所需要调用的用户空间函数的地址。

12167：**sa_flags** 进一步调整信号量处理代码所完成的工作。可能的标志集合从 12108 行开始定义。这些标志允许用户代码在信号量实例发送以后（或者保留用户定制的操作时）请求恢复缺省操作，等等。这一点在宏定义块前面的标签注释中已经说明了。

12168：**sa_restorer** 是本书中所没有涉及的一些信号量处理代码细节所使用的。

12169：**sa_mask** 是一系列其它信号量的集合，进程在处理这些信号量的过程中可能需要进行锁定。例如，如果一个进程在处理 **SIGCHLD** 的时候希望锁定 **SIGHUP** 和 **SIGINT**，进程的第 **SIGCHLD** 个 **sa_mask** 就会对与 **SIGHUP** 和 **SIGINT** 相关的位进行置位。

siginfo_t

11851：**struct siginfo**（也称为 **siginfo_t**）是伴随着信号量，特别是在实时信号量，所传递的额外信息。

11852：勿庸置疑，**si_signo** 是信号量的数目。

11853：**si_errno** 应该是信号量传递时传送者的 **errno** 的值，这样接收者就可以对它进行检测。内核本身并不关心这个值；当在某些情况下需要设置这个值时，内核将其设置为 0。我推测如果这样，即使调用者没有设置这个值，它们仍然会发现 **si_error** 的值被设为已知状态。

11854：**si_code** 记录了信号量的来源（不是发送者的进程 ID 号，也就是 PID——它在别处记录）。有效的信号量来源在 11915 行及其随后部分使用宏进行了定义。

11856：该结构的最后一部分是 **union** 类型的；该 **union** 类型依赖于 **si_code** 的值。

11857：**union** 的第一部分是**_pad**，它将 **siginfo_t** 的长度扩展填充为 128***sizeof**（**int**）字节（在 x86 平台上一共是 512 个字节）。留意一下这个数组的大小，也就是 **SI_PAD_SIZE**（11849 行），代表了该结构的前三个成员——如果增加了更多的成员，**SI_PAD_SIZE** 就需要进行相应修改。

struct signal_queue

17132：**struct signal_queue** 结构用来确保所有的实时信号量都被正确传送了，如果可能，每一个都包含着额外信息（**siginfo_t**）。如同后面你将会看到的一样，内核会为每个进程都设置一个队列，用来存放该进程的挂起的实时信号量。这个队列类型本身很小，仅仅由一个指向下一个节点的指针和 **siginfo_t** 本身组成。

应用函数

有关信号量的一个最重要的数据结构是 **sigset_t**，它是由一系列在 include/linux/signal.h 文件中定义的简单函数所操纵的，这些函数的定义从 17123 行开始。在 x86 平台上，这些相同的函数可以——而且已经——使用汇编语言更加有效的实现了；这些更高效的版本从 12204 行开始。（m68k 端口是唯一一个例外的端口，它使用体系结构特有的代码实现。）由于平台无关的版本和 x86 特有的版本都很重要，我们会对两者都加以介绍。

平台无关的 sigset_t 函数

配合 **sigset_t** 使用的平台无关的函数在 include/linux/sigal.h 文件中，从 17123 行开始。称为“ bitops ”（位级的操作）的函数将在后面介绍。

sigaddset

17145：**sigaddset** 把一个信号量加入集合——也就是说，它修改了集合中的一位。
17147：为了便于位操作，将基于 0 的信号量转化为基于 1 的信号量。
17149：如果信号量中填入一个无符号长整型数，恰当的位就会被设置。
17151：否则，**sigaddset** 就需要绕很多弯路，首先装入恰当的数组元素，接着设置该元素中相关位。

17148 行的代码和该文件中后面的其它代码一样，第一次见到时可能会令人感到有些困惑。在内核代码中，速度是压倒一切的因素。从而，也许你并不会看到类似于下面的运行期间进行决定的代码：

```
if (_NSIG_WORDS == 1)
    set->sig[0] |=1UL << sig;
else
    set->sig[sig / NSIGBPW] |= 1UL << (sig % NSIGBPW);
```

而你看到的是类似于下面的在编译期间决定的代码：

```
#if (_NSIG_WORDS == 1)
    set->sig[0] |=1UL << sig;
#else
    set->sig[sig / NSIGBPW] |= 1UL << (sig % NSIGBPW);
#endif
```

难道这样不会运行的更快些吗？不要忘了，**if** 条件是能够在编译期间进行计算的，因此预处理器可以使系统没有必要在运行期间执行检测工作。

当你认识到优化工作的实现方式时，这也就没有什么神秘的了。gcc 的优化器的敏锐程度足以注意到 **if** 表达式只有一个出口，因此它可以把那些不必要的代码移走。作为内核“运行期间”版本的结果代码和“编译期间”的版本是等同的。

但是在我们使用优化器很糟糕的编译器时，基于预处理器的版本还会更好吗？这一点并不确定。问题之一是，基于预处理器的（编译期间的）版本有一点更难懂。当代码的复杂程度比前面的简单例子要高时，可读性的差别就会明显的显示出来。例如，让我们考虑一下 **sigemptyse**t 中的从 17264 行开始的 **switch**。现在的 **switch** 类似于这样：

```
swithc (_NSIG_WORDS) {
default:
   memset(set, 0, sizeof(sigset_t));
   break;
case 2:   set->sig[1] = 0;
case 1:   set->sig[0] = 0;
   break;
}
```

（请注意经周密考虑的 case 2 随 case 1 连续执行的情况。）为了更好的利用预处理器而将其重写，它就可能类似于：

```
#if ((_NSIG_WORDS != 2)) && \ (_NSIG_WORDS != 1)
   memset(set, 0, sizeof(sigset_t));
#else /* (_NSIG_WORDS is 2 or 1). */
#if (_NSIG_WORDS == 2)
   set->sig[1] = 0;
#endif
```

```
    set->sog[0] = 0;
#endif /* _NSIG_WORDS test. */
```

gcc 的优化器为两者产生的目标代码是相同的。你更希望读哪一种版本的源程序代码呢？

另外，即使编译器的优化器并没有这么好——这种优化实在相当简单——那么编译器就不可能生成很好的代码。不管我们提供多少帮助都注定是不够的，因此我们可能要编写一些更容易读、更容易维护的代码——这是又一项工程技术的权衡。最后，就象我们在前面的内容中已经看到而且还要不断看到的那样，使用除 gcc 之外的编译器编译内核本身就是个挑战——增加一段 gcc 特有代码不会引起更多问题的。

sigdelset

17154：这些代码和 **sigaddset** 非常类似；区别在于这里从集合中删去了一位——就是把相应的位设置为关。

sigismember

17163：这些代码和 **sigaddset** 也非常类似；这里是要测试某一位是否被设置。注意到 17167 行可能和下面的这种写法有同样的好处：

```
return set->sig[0] & (1UL << sig);
```

这种写法与 17169 行非常相似。虽然这样能够和其它函数的编写风格更加一致，但是这并不是什么改进。

这些修改将对函数的行为方式稍有改动：它现在返回 0 或 1，经过这种修改，就可以在一个位被设置时返回其它的非 0 值。但是，这种改变不会终止没有退出的代码，因为其调用者只关心返回值是否为 0（它们并不特别在意是否为 1）。

sigfindinword

17172：这个函数返回 **word** 中设置的第一个位的位置。函数 **ffz**（在本书中没有涉及）返回其参数中第一个 0 位的位置。在将位求补的字中的第一个 0 的位置——这正是这个函数搜寻的内容——显然是原始顺序中的第一个 1 的位置。它从最小位 0 开始计算。

sigmask

17177：最后，这个有用的 **sigmask** 宏简单的把信号量数目通过一个相应的位集合转化为一个位掩码。

平台相关的 sigset_t 函数

即使平台无关的版本已经使用了简单有效的 C 代码，它也可以通过使用 x86 CPU 家族的方便而功能强大的位集指令在 x86 平台上更加有效地实现，。这些函数中的大部分都可以减少为单独的机器指令，因此这里的讨论也都很精简。

在 x86 平台（例如 m68k）上平台无关的函数对于编译器甚至是不可见的。17126 行包含进了 asm/signal.h 文件，在 x86 上这个文件被分解为 include/asm-i386/signal.h，这都应该归功于设置文件所建立的符号链接。12202 行定义了预处理器符号 **__HAVE_ARCH_SIG_BITOPS**，它消除了这些平台无关的函数的定义（请参看 17140 行）。

sigaddset

12204：x86 特有的使用 **btsl** 指令的 **sigaddset** 实现，它仅对操作数的一个位进行设置。

sigdelset

12210：同样，这是 x86 特有的使用 **btrl** 指令的 **sigdelset** 实现，它对操作数的一个位进行重置（清除）。

sigismember

12233：**sigismember** 根据其 **sig** 参数是否是一个编译期常量表达式来选择实现方法。文档中所没有说明的 gcc 编译器的强大的特殊参数**__builtin_constant_p** 是一个编译期操作符（就象 **sizeof** 一样），它能够报告是否可以在编译期间计算其参数值。
如果可以，**sigismember** 使用**__const_sigismember** 函数（12216 行）完成这项工作，因为它的大部分表达式都可以在编译期间计算。否则就使用更为普遍的版本**__gen_sigismember** 函数（12224 行）来代替。更普遍的版本中使用的是 x86 的 **btl** 指令，它需要测试其操作数中的某一位。
注意到在编译期的常量合并和死锁代码消除通常意味着这样的完整测试只能在编译期间执行——关键是 **sigismember** 要根据需要使用**__const_sigismember** 或者**__gen_sigismember** 替换，在作为结果的目标代码中甚至完全看不出来根本就没有对另一部分进行考虑。这样相当精简，难道不是吗？

sigmask

12238：x86 特有的 **sigmask** 的实现，这与平台无关的版本是等同的。

sigfindinword

12240：最后，x86 特有的 **sigfindinword** 实现只使用了 x86 的 **bsfl** 指令，它在自己的操作数中寻找一个设置位。

设置函数

除了前面的那一组函数之外，还有一组对 **sigset_t** 执行设置操作的函数和宏。和前面一组类似，这些函数使用**__HAVE_ARCH_SIG_SETOPS** 预处理器符号保护起来。然而现在没有一种体系结构能够提供自己独有的这些函数的实现，正因为如此，体系结构无关的版本是现存的唯一版本。

_SIG_SET_BINOP

17184：我们希望定义的全部三个二进制操作——**sigorsets**，**sigandsets** 和 **signandsets**——的实现方式从本质上来说是相同的。这个宏简单的把这三个函数的共同代码分解出来，从而只给它们提供一个操作和一个名字。当然，这同 C++模版函数类似，不过这样我们不得不自己处理一些记录工作，而不能完全信任编译器——这是我们使用 C 工作所付出的一部分代价。

17191：程序开始在 **sigset_t** 中全部四个字节的无符号长整型数的循环，同时对这些操作数进行应用。这个循环是为了速度的原因而展开的——通过减少循环控制开销来提高速度，这是很出名的一种增加速度的方法。然而，大多数情况下，这个循环根本就不执行。例如，在 x86 平台上，编译器可以在运行期间就证实不会执行该循环体，因为截断取整以后，**_NSIG_WORDS**/4 的结果是 0。（回忆一下**_NSIG_WORDS** 在 x86 平台上的值为 2。）

17201：**switch** 从循环末尾处理剩余工作的这行开始。如果在某些平台上**_NSIG_WORDS** 正好为 6，那么该循环就可以执行一次，而且 **switch** 的情况 2 也可以被执行。在 x86

平台上，循环永远不会执行；只有 **switch** 的情况 2 才可能执行。
顺便说一下，我并不清楚为什么 **switch** 不和其类似的**_SOG_SET_OP** 一样使用直接流程的方式实现。通常情况下，现存的版本可以更充分的利用缓存（如果你试图重新编写它，那么你就可以清楚的认识到这一点）——但是如果实际原因的确如此，那么**_SIG_SET_OP** 也应该使用相同的参数。

_SIG_SET_OP

17238：**_SIG_SET_OP** 和**_SIG_SET_BINOP** 类似，但是它使用的是一元操作而不是二元操作，因此我们并不需要详细地介绍它。但是你应该注意的是，这只能使用一次——在 17257 行生成 **signotset**——这和**_SIG_SET_BINOP** 不同。因此，在某种程度上这是不需要的——其实现者可以直接编写 **signotset**，而不必借助**_SIG_SET_OP**，这并没有产生任何重复代码。然而，二者生成的目标代码是相同的，这样如果我们以后选择增加一元操作，意义也就不大了。

sigemptyset

17262：**sigemptyset** 清空所提供的集合——要把其中的每一位都清空。（下面一个函数 **sigfillset** 和这个函数功能相同，不过它要设置所有的位而不是清除所有的位，因此我们就不再详细介绍了。）

17265：普通情况下使用 **memset** 把集合中的每一位都置为 0。

17268：对于**_NSIG_WORDS** 的一些比较小的值来说，简单的直接设置 **sigset_t** 的一两个元素可能速度更快。在这里采用的就是这种直接流程实现。

sigaddsetmask

17292：该函数和下面的几个函数是更简单快速设置和读取最低的 32 位(或者根据字的大小)信号量的一系列函数。**sigaddsetmask** 简单地把 **mask** 所指定的位置位，而不对剩余的位进行任何处理——这是一个集合的联合操作。

siginitset

17310：根据提供的掩码对最低 32 位（或者是别的）置位，并将其它位设置为 0。下面一个函数 **siginitsetinv**（17323 行）正好相反：它根据掩码的补数设置最低 32 位（或者别的），并对其余的位置位。

传送信号量

从用户的观点来看，传送信号量相当简单：调用系统调用 **kill**，该调用只需要进程 ID 号和一个信号量。但是，正如本节中所显示的那样，其实现要复杂得多。

sys_kill

28768：**sys_kill** 是系统调用 **kill** 的一个具有欺骗性的实现样例；真正的实际工作是在 **kill_somethig-info** 中实现的，我们随后就将对这个方面进行研究。**sys_kill** 的参数是要传递的信号量 **sig** 和信号量的目的 **pid**。就象你将看到的那样，参数 **pid** 并不仅是进程 ID。（PID 和进程的其它概念都在第 7 章中详细介绍。）

28770：根据提供给 **sys_kill** 的信息声明并填充 **struct siginfo** 结构。特别要注意的是 **si_code** 是 **SI_USER**（因为只有用户进程才可以调用该系统调用；内核本身是不会调用系统调用的，它更倾向于使用低层函数）。

28778：传递这些信息给 **kill_something_info**，该函数处理实际的工作。

kill_something_info

28484：该函数的参数和 **sys_kill** 类似，但是增加了一项 **siginfo** 结构的指针。

28487：如果 **pid** 为 0，就意味着当前进程希望把信号量传递给整个进程组，该工作由 **kill_pg_info**（28408 行）完成。

28489：如果 **pid** 是-1，信号量（几乎）被送往系统中的每一个进程，这在下面的段落中介绍。

28494：使用 **for_ech_tas**k 宏（在 16898 行宏定义，第 7 章中详细介绍）开始循环处理现存进程列表的每一项。

28496：如果这不是 idle 进程（或 init），就使用 **send_sig_info**（28218 行，后面将会讨论）传递信号量。每次发现合适的任务时 **count** 的值都会增加，虽然 **kill_something_info** 并不关心 **count** 的实际值，而是在意是否能够发现合适的进程。如果所有试图发送信号量的努力都失败了，将记录失败的过程以使得 **kill_something_info** 可以在 28503 行返回错误代码；如果发生了多次错误，则只返回最后一次失败的情况。

28503：如果发现了合适的候选进程，**kill_something_info** 就返回最近失败的错误代码，或者成功就返回 0。如果没有发现任何合适的候选进程，就返回 **ESRCH** 错误。

28504：其它负的 **pid**（是负值，但不是-1）定义了接收信号量的进程组；**pid** 的绝对值是进程组号。和前面一样，**kill_pg_info** 的使用就是出于这种目的。

28506：其它的所有可能性都已经进行了说明；**pid** 必须为正数。在这种情况下，它是信号量传送的目的进程的 PID。这由 **kill_proc_info** 实现（28463 行，很快就会讨论）。

kill_pg_info

28408：这个函数给进程组中的每一个进程发送一个信号量和一个 **struct siginfo** 结构。其函数体和前面介绍的 **kill_something_info** 类似，因此我只是简单介绍一下。

28417：开始循环处理系统中的所有进程。

28418：如果进程在正确的进程组中，那么信号量就发送给它。

28427：如果信号量成功发送给任何进程，**retval** 就设置为 0，从而在 28430 行成功返回。如果信号量不能被发往任何进程，那么要么是所给的进程组中没有进程，在这种情况下，**reval** 仍然会在 28415 行赋值为**-ESRCH**；或者 **kill_pg_info** 发送信号量给一个或多个进程，但是每次都失败了，在这种情况下 **retval** 值为从 **send_sig_info** 得到的最近错误代码。注意它和 **kill_something_info** 的细微区别，后者如果发送信号量失败时就返回错误。但是这里的 **kill_pg_info** 即使在某些情况下出错了，只要信号能成功地传递给任意进程，就会返回成功信息。

28430：在 28410 行中，如果进程组号无效，**retval** 或者是如前所述的赋值，或者就是**-EINVAL**。

kill_proc_info

28463：**kill_proc_info** 是一个相当简单的函数，它把信号量和 **struct siginfo** 结构传递给由 PID 定义的单个进程。

28469：通过所提供的 PID 查找相应的进程；如果成功 **find_task_by_pid**（16570 行）返回一个指向该进程的指针，如果没有找到该进程就返回 **NULL**。

28472：如果找到匹配进程，就使用 **send_sig_info** 把信号量传送给目的进程。

28474：返回错误指示，或者是在 28470 行由于没有发现匹配进程而返回**-ESRCH**，或者是其它情况下从 **send_sig_info** 中返回的值。

send_sig_info

28218：我们最后看的几个函数中最重要的显然是 **send_sig_info**。这个函数使用不同的方法装载进程并处理实际的工作。现在应该了解一下实际的工作是如何完成的。**send_sig_info** 将使用 **info** 指针（该指针也可能为 **NULL**）指向额外信息的信号量 **sig** 传送给 **t** 指针（调用者应该保证 **t** 不会为 **NULL**）指向的进程。

28229：确保 **sig** 在范围之内。注意使用的是如下的测试

sig > _NSIG

而不是你可能预期的

sig >= _NSIG

这是因为信号量的计数是从 1 开始的，而不是从 0 开始的。因此虽然不存在对这个信号量编号的定义，有效信号量的编号的标识符**_NSIG** 本身也是有效的信号量编号。

28233：这是另外一个严密性检查——实际上包含多个检验。基本的思想是检测信号量的传送是否合法。虽然内核本身可以给任何进程传送信号量，但是除了在涉及 **SIGCONT** 的情况之外，除 root 之外的用户都不能给其它用户的进程传送信号量。总之，这个长长的 **if** 条件说明了如下问题：

- （28233 行）如果不存在补充信息，或者虽然存在补充信息，但是信号量来源于用户而不是内核，并且…
- （28235 行）…信号量不是 **SIGCONT**，或者虽然信号量是 **SIGCONT**，但是并不是传送给同一会话过程中的其它进程，并且…
- （28237 行和 28238 行）…发送者有效的用户 ID 既不是已经存储了的目标进程的用户 ID，也不是目标进程的当前用户 ID，并且…
- （28239 行和 28240 行）…发送者的当前用户 ID 既不是已经存储了的目标进程的用户 ID，也不是目标进程的当前用户 ID，并且…
- （28241 行）…此处不会允许用户超越普通许可（例如，由于用户是 root）…那么就不应该发送信号量了；可以跳过这段发送信号量的代码。

对于前面的 **if** 条件必须明白两点。首先，当将 **info** 映射为无符号长整型数的时候，如果它为 1，这就不是一个实际指向 **struct siginfo** 结构的指针。相反的，它是说明信号量来自于内核的特殊值，但是并没有进一步的附加信息可供使用。内核本身在最低的页（内存页在第 8 章中讨论）中并不分配空间，因此在 4,096 之下的任何地址（除了 0，**NULL** 之外）都可以作为这种特殊值使用。

其次，在这几种条件的情况中，位 XOR 运算操作符（^）比不等运算操作符（**!=**）使用得更为普遍。在这些情况下，两个操作符意义相同，因为如果两个相比较的正数之间有一位不同，在 XOR 运算的结构中就至少有一位被置位，所以结果非空，逻辑值为真。推测起来，cc 的早期版本为^生成的代码比为**!=**生成的代码更为有效，但是在现在的编译器版本中就不是这样了。

28248：忽略信号量 0 并拒绝将信号量传送给僵进程（已经退出但是还尚未从系统的数据结构中移走的进程；请参看第 7 章的“进程状态”一节，它讨论了函数 **exit**）。

28252：对于一些信号量，在实际发送之前必须进行一些额外的工作。这些工作是在这里的 **switch** 中处理的。

28253：如果正在发送 **SIGKLL** 或者 **SIGCONT**，**send_sig_info** 就唤醒进程（也就是说，如果当前被停止了就允许它再次运行）。

28257：设置进程的返回代码为 0——如果进程已经使用 **SIGSTOP** 停止了，返回代码域就被用来在停止等待的信号量和其祖先间建立通讯。

28258：取消任何挂起的 **SIGSTOP**（被调试器阻塞），**SIGSTP**（由键盘输入的 Ctrl+Z 终止），

SIGTTIN（试图从 TTY 中读取信息的后台运行进程），**SIGTTOU**（试图向 TTY 中写入信息的后台运行的进程）；这些是所有可能中断进程的条件，也是 **SIGCONT** 或者 **SIGKILL** 最可能作为响应出现的情况。

28263：在一些信号量被删除之后，调用 **recalc_sigpending**（16654 行，将在后面讨论）来判断是否还有信号量仍然处于挂起状态以等待进程。

28266：在前面的情况中，如果 **SIGCONT** 或者 **SIGKILL** 到达了，这四个信号量就会都被取消。但是看起来有些不太对称，如果这四个信号量有一个到达了，任何挂起等待的 **SIGCONT** 都会被取消。然而 **SIGKILL** 却不会被取消，这遵循 **SIGKILL** 永远不会被锁定或者取消的规律。

28281：如果目标进程希望忽略信号量并且允许不接收信号量，那么就跳过了信号量的接收过程。

28284：非实时信号量并不排队等待，这就意味着如果在进程处理第一个信号量实例之前，同一信号量的第二个实例就到达了，那么第二个实例就会被忽略。这一点就是在这里确保的（回想一下 **struct task_struct** 结构的 **singnal** 成员中保存着一个进程的当前正在挂起等待的信号量的集合）。

28304：在限制条件的控制下，实时信号量需要排队等待。最重要的限制是可以同时排队等待的实时信号量总数的可配置限制；这一限制值为 **max_queued_signal**，它是在 28007 行定义的，而且可以使用 Linux 的系统控制特性加以修改。如果有空间来容纳更多的信号量，就分配 **struct signal_queue** 结构来容纳排队等待的信息。

为什么所要首先限制排队等待的信号量的数目呢？这是为了防止服务拒绝的攻击：如果没有这个限制，用户可以持续发送实时信号量直到内核内存溢出，这样就会阻碍内核为其它进程提供该服务及其它服务。

28310：如果一个队列节点已经被分配，现在 **send_sig_info** 就必须使有关这个信号量的信息进入队列。

28311：把信息加入队列是很直接的：**send_sig_info** 把挂起等待的信号量数量（全局变量）增加 1，接着把新的节点增加到目标进程的信号量队列中。

28315：根据提供给 **send_sig_info** 的 **info** 参数填充队列节点的 **info** 成员。

28316：0（**NULL**）意味着信号量是从用户发送而来的，而且可能使用了从 28513 行到 28544 行定义的向后兼容的信号量发送函数。目标 **siginfo_t** 使用相对比较明确的值来填写。

28323：值 1 是指示信号量来源于内核的一个特殊值——再一次的使用了向后兼容的函数。和前面的情况一样，目标 **siginfo_t** 使用相对比较明确的值来填写。

28331：正常情况下，**send_sig_info** 得到一个实际的 **siginfo_t**，它可以简单地将其拷贝到队列节点中。

28334：没有分配队列节点——或者因为系统内存溢出而造成 **kmem_cache_alloc** 在 28306 行返回 **NULL**；或者因为已经达到了信号量队列的最大值，**send_sig_info** 根本就没有试图分配节点。不管怎样，**send_sig_info** 所处理的内容是相同的：除非该信号量是通过内核或者老式的信号量函数（例如 **kill**）发出的，否则 **send_sig_info** 就返回 **EAGAIN** 错误，通知调用者现在信号量不能排队等待，但是后来调用者应该可以再次使用相同的参数成功执行调用。否则，**send_sig_info** 就传送该信号量但并不将其排入队列中。

28345：最后，**send_sig_info** 从实际上准备好发送信号量。首先，信号量进入该进程的挂起等待的信号量的集合中。注意即使信号量被锁定了这个过程也要执行，这可能有点奇怪。但是这样处理是有原因的：内核必须提供 **sys_sigpending**（28981 行，本章中后面部分将讨论），它允许进程查询在锁定时传送进来什么信号量。

28346：如果信号量没有被锁定，那么进程应该被通知有信号量到达了。相应的，其 **sigpendig** 标志被置位。

28370：如果进程正在等待信号量的到达并且有信号量也正在等待它，那么这个进程就被唤醒（使用 **wake_up_process**，26356 行）来处理信号量。

force_sig_info

28386：这个函数被内核用来保证不管进程是否需要，它都确实接收了一个信号量。例如，在进程释放未用指针时，可以使用这个函数来确保该进程接收了 **SIGSEGV**（请参看 7070 行——实际上是调用了向后兼容的函数 **force_sig**，但是 **force_sig** 完全是按照 **force_sig_info** 实现的）。**force_sig_info** 的参数和 **send_sig_info** 的参数相同，两者的意义也相同。

28392：如果目标进程是僵进程，即使是内核也不应该给它发送任何信号量；所进行的尝试将被拒绝。

28397：如果进程将要忽略这个信号量，**force_sig_info** 将通过强制它执行缺省操作的方式进行纠正。实际上它并不像外表所表现出来的那样无害：在内核使用该函数的情况下，对这个信号量的缺省操作是杀死进程。

28399：把信号量从 **t** 所锁定的集合中移走。

28402：**force_sig_info** 现在已经建立了一些条件使得 **t** 必须接收信号量，因此该信号量就可以使用 **send_sig_info** 进行发送。如果 **send_sig_info** 的实现改变了，这仍然可能造成信号量不能发送，因此这两个函数必须保持同步。

recalc_sigpending

16654：这个函数重新计算进程的 **sigpending** 标志；当进程的 **signal** 或 **blocked** 集合改变时就调用该函数。

16676：在最简单的情况中，**recalc_sigpending** 将信号量和锁定集合求补的结果执行位 AND 操作。（对锁定集合求补就是允许的集合。）其它的情况仅仅是这种情况的泛化。

16679：如果前面操作中的任何一个在 **ready** 中遗留下了任何一位，那么挂起等待的信号量集合中最少有一个信号量现在还被锁定；因此 **recalc_sigpending** 将增加 **sigpending** 标志的值。

由于 **recalc_sigpending** 所实际需要了解的全部内容只是是否至少有一个信号量在挂起等待——例如，如果不止一个，也并不需要知道有多少信号量在挂起等待——非平凡情况下的代码只要发现 **ready** 的值被置为非 0 值就应该停止对其进行修改（例如，前面 16662 行通过中断循环）。但是，任何可能来对此优化所产生的效率增进都必须要同为此而进行的额外测试进行权衡。正是由于这个原因，又加上 **_NSIG_WORDS** 很小（在实际中无论如何都是如此），改进的版本可能要比标准情况快一点。

ignored_signal

28183：**ignored_signal** 有助于 **send_sig_info** 决定是否给一个进程发送信号量。

28189：如果进程正被其祖先跟踪（可能是调试器），或者信号量是在进程锁定的集合中，那么它就不能被忽略。第二种情况可能是我们过去所没有考虑过的；如果信号量被锁定了，**send_sig_info**（还有 **ignored_signal**）难道不应该将其忽略吗？如果情况的确如此，还真不应该忽略。这个函数通过信号量是否应该被忽略表明了进程的信号量的 **signal** 集合的相应位是否应该被置位。如同前面我们已经看到的那样，对 **sigpending** 系统调用的支持要求如果在锁定过程中有信号量到达，内核就应该设置相应的位。因此，被锁定信号量不能简单地忽略。

28194：如果进程是一个僵进程，信号量就应该被忽略。这种测试是不必要的，因为这种情况甚至在 28248 行的 **ignored_signal** 调用之前就会被发现。

28199：在大多数情况下，**SIG_DFL**（缺省的）操作是处理信号量而不是将其忽略。你所能看到的例外是 **SIGCONT**，**SIGWINCH**，**SIGCHLD**，和 **SIGURG**。

28207：进程允许忽略大部分信号量，但是不能忽略 **SIGCHLD**。对于 **SIGCHLD**，POSIX 赋予 **SIG_IGN** 一种特殊的意义，这一点在 28831 行将会说明。这里所提到的“automatic child reaping”（自动子进程空间回收）将在 3426 行执行。

28211：在缺省的情况下，可以假定 **ignored_signal** 有一个实际的函数指针，而不是 **SIG_DFL** 或者 **SIG_IGN** 两个伪值的一个。这样，信号量就和用户定义的处理句柄联系起来，这意味着进程希望处理这个信号量。它通过返回 0 来指明信号量不应该被忽略。

do_signal

3364： **do_signal** 在信号量到达进程时使用。这个函数在内核中被调用的地方不止一次——如我们在第 5 章中看到的从 203 行和 211 行，还有从 2797 行和 2827 行。通常所有这些情况都是当前进程希望处理挂起等待的信号量（如果有的话）。

3375： 如果非空，**oldset** 用来返回当前进程锁定的信号量集合。由于 **do_signal** 不会修改锁定的集合，它可以简单的返回一个指向现有锁定集合的指针。

3378： 进入几乎扩展到该函数末尾（3478 行）的循环。退出该循环的方法只有两种：把所有可能的信号量都处理了，或者处理唯一一个信号量。

3382： 使用 **dequeue_signal** 使信号量出队列（28060，后面将会介绍）。**dequeue_signal** 或者返回 0，或者返回需要处理的信号量的编号，并且它还会填充 **info** 中的附加信息。

3385： 如果没有信号量处于等待状态，将在这里中断循环。正常情况下，它在循环第一次执行过程中是不会发生的。

3388： 如果当前进程正在被其祖先跟踪（可能是调试器），而且信号量也并不是不可锁定的 **SIGKILL**，那么在信号量到达之前，进程的祖先就必须已经得到通知了。

3391： 将传递给子孙进程的信号量编号传送到祖先进程中对应子孙进程的 **exit_code** 域；祖先使用 **sys_wait4** 收集这些信息（23327 行，在第 7 章中介绍）。**do_signal** 停止子孙进程的运行，然后使用 **notify_parent**（3393 行）给祖先进程发送 **SIGCHLD** 信号量，接着调用调度函数 **schedule**（26686 行，第 7 章中介绍），给其它进程——尤其是其祖先进程——运行的机会。**schedule** 会把 CPU 分配给其它进程，因此直到内核跳转回这个进程才会返回。

3397： 如果调试器取消了信号量，**do_signal** 在这里就不应该处理它；循环继续进行。

3402： **SIGSTOP** 可能只是由于进程正在被跟踪而产生。这里没有必要处理它；循环继续进行。

3406： 如果调试器修改了 **do_signal** 要处理的信号量编号，**do_signal** 将根据新的信息填充 **info**。

3415： 正如注释中所说明的一样，如果新的信号量被锁定了，就需要重新排队，循环继续进行。否则，控制流程将直接执行下面的代码。

3421： 在这里，或者进程未被跟踪，或者进程正被跟踪但是得到了一个 **SIGKILL** 信号量，或者控制流程直接从前面的代码块中执行下来。在任何一种情况中，**do_signal** 都有一个应该现在处理的信号量。它从获取 **struct k_sigaction** 结构开始，这个结构指明了怎样处理这个信号量编号。

3423： 如果进程试图忽略信号量，那么除非这个信号量是 **SIGCHLD**，否则 **do_signal** 就继续执行循环从而忽略该信号量。为什么这个测试不能同时保证该进程不会忽略掉 **SIGKILL** 这个注定不可忽略也不可锁定的信号量呢？答案在于和 **SIGKILL** 相关的

操作永远不会是 **SIG_IGN** 的，实际上也不会是除 **SIG_DFL** 之外的任何操作——28807 行就保证了这一点（在 **do_sigaction** 函数中）。这样，如果操作是 **SIG_IGN**，那么信号量编号就不可能是 **SIGKILL**。

3426：如同在从 28820 行开始的标题注释中说明的一样，POSIX 标准说明了忽略 **SIGCHLD** 的操作实际上意味着自动回收其子孙进程。子孙进程是通过使用 **sys_wait4** 来回收的（23327 行，在第 7 章中介绍），此后循环继续运行。

3435： 进程为这个信号量采用缺省操作。专用的初始化进程接收到全部信号量所对应的缺省操作是把信号量整个忽略掉。

3439： 对信号量 **SIGCONT**、**SIGCHLD** 和 **SIGWINCH** 所采取的缺省操作是不加处理，只是简单地继续执行循环。

3442： 对于信号量 **SIGSTP**，**SIGTTIN** 和 **SIGTTOU**，缺省的操作各自不同。如果该进程所归属的进程组是孤立的——简单的说就是没有连接到 TTY 上——那么 POSIX 规定对于这些基于终端的信号量的缺省操作是将其忽略。如果进程的进程组不是孤立的，缺省的操作是停止进程的运行——这和 **SIGSTOP** 的情况是相同的，在这种情况下控制流程直接向下运行。

3447： 在对 **SIGSTOP** 的响应中（或者是从前面情况中直接执行下来），**do_signal** 终止了进程。另外，除非祖先进程已经规定对其子孙进程的终止运行不加理会，否则祖先进程将会在其子孙进程退出时被通知。和 3394 行一样，调用 **schedule** 交出 CPU 给其它某一进程。当内核把 CPU 再次分配给当前进程的时候，该循环继续运行以处理队列中的另外一个信号量。

这不是我们希望的——我认为当 **schedule** 返回时，循环应该退出，因为信号量已经处理完了。其原理在于如果进程被终止了，唤醒进程的最可能的原因是进程又得到了信号量，可能是 **SIGCONT**，因此该进程现在就可以检测并处理信号量了。

3456： 对于其它信号量的缺省操作是退出进程。它们中的一些将使进程首先清空内核（详细的介绍请参看第 8 章），这些信号量就是 **SIGQUIT**，**SIGILL**，**SIGTRRAP**，**SIGABRT**，**SIGFPE** 和 **SIGSEGV**。如果此二进制格式（详细的介绍请参看第 7 章）知道如何清空内核并且成功地清空了内核，那么在进程的返回代码中就会有一位被设置来指明进程在退出之前就已经清空了内核。接着流程按照 **default** 的情况继续执行，终止进程的运行。注意 **do_exit**（23267 行，在第 7 章中也会有介绍）是从来不会返回的——因而在 3471 行中会有“ NOTREACHED ”注释。

3476： 此处，**do_signal** 从队列中取出一个信号量，该信号量既不和 **SIG_IGN** 的操作有关，也不和 **SIG_DFL** 的操作有关。唯一的另外一种可能性是这是用户定义的信号量处理函数。**do_signal** 调用 **handle_signal**（3314 行，本章随后将会更为详细地讨论）来触发这个信号量处理函数，接着返回 1 向调用者声明这个信号量已经处理过了。

3481： 此处，**do_signal** 不能为当前进程从队列中取出信号量。（只有从 3386 行的 **break** 退出时才能执行到本行。）如果在系统调用的处理过程中间被中断了，**do_signal** 就要调整寄存器，从而系统调用将可以重新执行。

3490：返回 0 以通知调用者 **do_signal** 没有处理任何信号量。

dequeue_signal

28060：**dequeue_signal** 将信号量从进程信号量队列中移出，同时忽略那些由掩码说明的信号量。它返回信号量的编号并使用指针参数 **info** 返回相关的 **siginfo_t**。

28071：为了避免重复参照而建立一些别名：**s** 是进程的挂起等待的信号量的集合（记住其中可能包括了一些锁定的信号量），**m** 是掩码的集合。特别要注意的是*s 表达式，在该函数中这个表达式出现了不止一次，但是它只是 **current->signal.sig[0]**的一种简

单写法。

28073：在这个 **switch** 条件分支中，**sig** 被设置为第一个挂起等待的信号量。从最简单的情况入手最容易理解；其它的情况只是这种情况的泛化。

28091：最简单的情况是：它把挂起等待的信号量和掩码求补后的结果进行位 AND 运算，结果被存储在临时变量 **x** 中；**x** 现在就是掩码不能忽略的挂起等待的信号量的集合。如果 **x** 不为 0，那么就存在挂起等待的信号量（**x** 至少有一位被置位）；**dequeue_signal** 使用 **ffz**（本书中没有涉及）取得相应的信号量编号，并将其转化为从 1 开始计数的信号量编号，将结果存储在 **sig** 中。正如前面所说明的一样，其它情况只是这种情况的泛化；最重要的结果是 **sig** 被置位，如果可能的话在每种情况下都是如此——此后其它变量（**i**，**s**，**m** 或者 **x**）的状态就不难理解了。如果在 **switch** 之后的 **sig** 是 0，掩码中就没有传递挂起等待的信号量。

28097：如果一个信号量正在挂起等待，那么 **dequeue_signal** 应该试图将其从队列中释放出来。**reset** 跟踪 **dequeue_signal** 以判定是否应该把信号量从进程的挂起等待的信号量队列中删除。将 **reset** 初始化为 1 仅仅是由于在函数处理过程中它可能会改变的假定。

28107：对于非实时信号量，内核不会保持原始的 **siginfo_t**（如果曾经有过的话），因此 **dequeue_signal** 应该尽可能的重新组织有关的信息。不幸的是，当前实现方法中并没有多少信息——只有信号量编号自身而已。**info** 的其它成员都简单地被设置为 0。

28118：在另一种情况，也就是实时信号量情况下，**siginfo_t** 只是一种点缀。**dequeue_signal** 会在进程的 sigqueue 中进行扫描以确定其值。

28122：如果找到了 **siginfo_t**，**dequeue_signal** 现在就使其出队列，将 **siginfo_t** 的内容拷贝到 **info** 中，并释放为这个队列节点分配的内存。

28129：如果队列中没有这个信号量的更多实例，那么信号量就不会在挂起等待了。但是为了弄清楚队列中是否还有信号量的实例，**dequeue_signal** 需要遍历整个队列。因此，该函数需要扫描这个队列的其余元素来查询是否存在相同信号量的其它实例。如果发现了实例，**dequeue_signal** 就清空 **reset** 标志——只有在这种独特的情况下才会进行的操作。

28142：正在出队的信号量是实时信号量，但是在进程的挂起等待的实时信号量队列中却没有发现它，其原因在代码中已经进行了阐述。现在，**dequeue_signal** 和它有非实时信号量的情况相同了——它知道信号量是可以访问的，但是没有方法可以访问其原始值——并且其响应过程处理的工作也完全相同，仅仅使用信号量编号来填充 **info**，而没有其它属性值。

28150 除非 **reset** 标志被清空了——也就是说除非这是一个实时信号量并且同一个信号量的其它实例仍然在挂起等待队列中——该信号量已经被处理过；它应该从进程的挂起等待集合中删除。

28152：信号量脱离队列，因此 **dequeue_signal** 应该重新计算进程的 **sigpending** 标志。我认为这里有一个可以进行少量优化的机会：只用当 **reset** 为真的时候 **dequeue_signal** 才需要这样处理。**recalc_sigpending** 从进程的锁定集合和挂起等待集合中计算结果；锁定的集合没有改变，因此只有当挂起等待的集合发生改变时，**dequeue_signal** 才需要调用 **recalc_sigpending**。如果 **reset** 为假，挂起等待的集合就不会改变，因此对于 **recalc_sigpending** 的调用就是不必要的。

28163：**switch** 没有发现信号量，因此没有信号量正在挂起等待。作为内部正确性的检测，**dequeue_signal** 确保内核不会认为有信号量正在为某任务挂起等待。

28174：返回出队的信号量编号，或者如果没有信号量出队，就返回 0。

notify_parent

28548：**notify_parent** 寻找进程的祖先进程并通知它其子孙进程的状态发生了改变——通常情况是其子孙进程或者被终止了，或者被杀死了。

28553：使用有关信号量发生的上下文的信息填充局部变量 **info**。

28564：如果子孙进程已经退出，**why** 被赋以适当的值以指明其原因是因为它清空了内核，或者被某信号量将其杀死，或者因为执行了非法操作。

28572：类似地，如果使用信号量终止了进程，对 **why** 赋值以说明发生的情况。

28578：前面的情况几乎覆盖了所有的可能性。如果没有，函数打印出警告信息并继续运行；在这种情况下，系统会在 28562 行将 **why** 的值赋为 **SI_KERNEL**。

28586：给进程的祖先进程发送信号量。下面一行唤醒任何等待这个子孙进程的进程并为其提供 **CPU**。

handle_signal

3314： **handle_signal** 在需要调用用户定义的信号量处理程序时由 **do_signal** 调用。

3338： 建立一个用户处理程序可以在其中运行的堆栈帧。如果进程已经请求了内核所拥有的有关信号量的原始值和其上下文的附加信息，那么堆栈帧就使用 **setup_rt_frame**（3231 行）构建起来；否则就使用 **setup_frame**（3161 行）构建。这两种方法都可以实现构建工作，这样控制流程会返回信号量处理程序。当它返回时，实际返回的是信号量到达的时候正在执行的代码。

3343： 如果 **SA_ONESHOT** 标志被设置，则信号量处理程序应该只执行一次。（注意 **sys_signal** 是 **signal** 系统调用的实现，它使用 **SA_ONESHOT** 类型的信号量处理程序——请参看 29063 行。）在这种情况中，缺省的操作是立即将其恢复。

3346： **SA_NODEFER** 意味着在执行这个信号量的处理程序时，不应该有其它信号量被锁定。如果位没有设置，其它的位现在就会被加入进程的锁定的集合中。

其它有关信号量的函数

其它一些有关信号量处理的函数。

sys_sigpending

28981：这个简短的系统调用允许进程询问在信号量锁定期间是否有非实时信号量到达。通过所提供的指针，该函数返回一个位集以指明它们是哪些信号量。

28987：这个函数的核心是进程的 **blocked** 集合和 **signal** 集合间的简单位 AND 操作。它只对最低 32 位感兴趣，这些都是非实时信号量。

28992：使用所提供的指针把挂起等待的集合拷贝回用户空间。如果失败就返回**-EFAULT**，如果成功就返回 0。注意是否有信号量正在挂起等待——也就是说，返回值是否为空——并不是成功的判据之一。

do_sigaction

28801：**do_sigaction** 实现了系统调用 **sigaction** 有意义的部分。（其余部分在 2833 行的 **sys_sigaction** 中。）**sigaction** 是 POSIX 中等价于 ISO C 的函数 **signal**——它把信号量和操作关联起来，这样进程接收到信号量时就能够执行相应的操作。

28806：健全性检测：确保 **sig** 在范围之内并且进程没有试图把 **SIGKILL** 或者 **SIGCONT** 和某种操作相关联。进程被简单地剥夺了覆盖这两个信号量的缺省操作的权力。然而，

和 **signal** 实现的处理程序不同，使用 **sigaction** 实现的处理程序不是 **SA_ONESHOT** 类型的，因此在处理程序被调用的时候就不用每次都将其重新装载。

28811：获取和这个信号量相关的指向 **k_sigaction** 结构的指针。

28813：**sigaction** 可以通过一个过去所提供的指针返回旧有的操作。这在以堆栈方式存在的处理程序中是很有用的，在这里处理程序被临时覆盖，以后再恢复出来。如果 **oact** 指针非空，旧有的操作就会被拷贝到其中。（但是这并不会把信息拷贝会用户空间；调用者必须执行这样的处理。）

28815：如果 **do_sigaction** 被赋予一个需要同信号量相关联的操作，那么二者现在就相互关联起来。**SIGKILL** 和 **SIGSTOP** 也必须被从操作的掩码中删除，为了确保这些信号量不会被锁定或者覆盖。

28836：正如在 28820 行开始的标题中注释的一样，为了遵守 POSIX 标准，下面的几行代码必须要经过一定变形，并且在必要情况下还会舍弃某些信号量。对于这些细节情况，我们即使跳过也不会有什么损失。

sys_rt_sigtimedwait

28694：**sys_rt_sigtimedwait** 等待信号量的到达，它可能在经过一段特定的时间间隔以后超时退出。并不是所有的信号量都会接收；指针 **uthese** 所指明的 **sigset_t** 说明了调用者所感兴趣的信号量。

28714：**uthese**（它已经被拷贝到局部变量 **these** 中了）是允许的信号量的集合，于是内核元语只知道如何锁定信号量。但这样也没有关系：对允许的信号量集合进行求补运算就得到了应该锁定的信号量，所得到的结果就可以直接使用了。

28717：如果调用者提供了超时时间，该超时时间就将被拷贝到用户空间中，而且其值也必须经过健全检测。

28726：检查是否已经有信号量正在等待了——如果有，就没有必要为其等待了。否则，调用者必须等待。

28731：保存原来的锁定信号量集合，然后阻塞由 **these** 定义的所有信号量。

28737：如果用户没有提供超时时间，那么超时时间会是 **MAX_SCHEDULE_TIMEOUT**（宏定义为 **LONG_MAX**，或者是 2^{31}-1，16228 行）。但是并不永远都是如此——超时时间是以瞬间（jiffy）计数的，它的系统时钟以每秒 100 次的速度跳动着，因此大约有 248 天，超时时间就耗尽了。（在 64 位机器中，这大约需要三十亿年。）

28739：如果用户确实提供了超时时间，就将其转化为以瞬间计算的值。“ + ”后面的表达式是对下一个瞬间进行向上舍入的明智方法——其思想是 **timespec_to_jiffies** 可能已经向下舍入了，但是内核必须是上舍入的，因为它必须等够用户请求的瞬间个数。它虽然可以检测 **timespec_to_jiffies**（18357 行）是否是下舍入的，但是下面这种方法更为简单：如果用户提供的超时时间不是 0 就为其增加一个瞬间，并且认为是对它进行了调整。毕竟 Linux 不是一个真正的实时操作系统——当你指定了超时时间时，Linux 只能保证至少等待如此长的时间。

28742：设置当前用户的状态为 **TASK_INTERRUPTIBLE**（请参看第 7 章）。**schedule_timeout**（26577 行）用来让出 CPU；在指定的时间用完以后或者其它事件到达并唤醒进程（比如接收了一个信号量）时，该进程才可以继续运行。

28746：进程希望被信号量唤醒。**sys_rt_sigtimedwait** 再次尝试从进程的等待信号量队列中取出信号量并恢复原来锁定的集合。

28752：此处，该函数仍然不知道信号量是否已经到达了——它可能无需等待就可以得到一个信号量，或者在等待期间可能有另一个信号量到达，也或者该函数一直在等待但是没有信号量到达。

28753：如果信号量到达，该函数就给用户进程传递信息并且返回信号量编号。

28759：否则，虽然进行了等待，但是没有信号量到达。在这种情况下，该函数或者返回**-EAGAIN**（说明用户进程可以再次使用相同的参数尝试），或者返回**-EINTR**（说明其等待过程被由于某些原因而不能传递的信号量中断了）。

内核如何区分实时信号量和非实时信号量

简单的说，答案并不复杂。我几乎掩盖了其中的绝大部分区别，这是有一定原因的：退出语句不多。现在，为了使这一点更加清楚，让我们来看一下系统调用 **sigprocmask** 的两个版本，它允许进程处理自己的锁定信号量的集合——增加，删除，或者简单地对信号量集合进行设置。

sys_sigprocmask

28931：**sys_sigprocmask** 是这个函数的原始版本，这一版本并不知道或者是不关心实时信号量。参数 **how** 指明了要执行的操作；如果 **set** 不为 **NULL**，就是这个操作的操作数；如果 **oset** 是非空的，那么 **oset** 返回的就是原始的锁定集合。

28937：如果 **set** 为空，那么 **how** 的值就没有什么用处了：该操作就没有操作数了，因此该函数不会处理有关的内容。否则，就继续执行该操作。

28939：在新的锁定集合中的拷贝，其中删除了不可锁定的 **SIGKILL** 和 **SIGSTOP**。

28944：为了处理以后将当前锁定集合拷贝回用户空间的需要，在 **old_set** 中存储当前锁定集合的一个备份。由于当前锁定集合在以后的代码中可能会被修改，因此在它改变之前必须对其值进行存储。

28948：当然是忽略无效的操作。

28951：**SIG_BLOCK** 操作符指明 **new_set** 应该解释为要锁定的附加信号量的集合。这些信号量将被加入该锁定集合中。

28954：**SIG_UNBLOCK** 操作符指明 **new_set** 应该解释为要从锁定的信号量的集合移出的信号量集合。这些信号量现在被移出锁定集合。

28957：**SIG_SETMASK** 操作符指明 **new_set** 应该解释为新的锁定集合，简单覆盖该锁定集合原有的值。因此，**sys_sigprocmask** 正是实现这一点的。注意它只设置了 **bloched.set** 数组的最低的元素——这个元素包含低 32 位非实时信号量，这是该函数所关心的内容。

28966：如果调用者已经请求查询锁定的集合的原来的值，执行流程就向前跳到 **set_old** 标号（28970 行）。

28968：如果 **set** 为空，意味着调用者没有请求对锁定集合进行修改，但是调用者可能仍然希望了解锁定集合的当前值。

28972：**oset** 非空（**set** 也可能为非空）。不管哪一种情况，**old_set** 都包含一个原来锁定的集合的备份，在返回之前 **sys_sigprocmask** 会试图将其拷贝回用户空间。

sys_rt_sigprocmask

28612：**sys_rt_sigprocmask** 和 **sys_sigprocmask** 非常类似，但是它也能够处理新的实时信号量。由于这两者之间的相似性，在这里我仅仅介绍一下它们之间有趣的区别。

28638：与如下代码不相类似的是

```
/* how sys_sigprocmask does SIG_BLOCK.*/
   new_set =    *set ;          /* line 28938 */
```

```
    blocked |= new_set ;       /* line 28952 */
```

（作为一个例子采用 **SIG_BLOCK** 的情况），实际代码类似如下代码：

```
/* how sys_rt_sigprocmask does SIG_BLOCK.*/
    new_set =    *set ;          /* line 28625 */
    new_set |= old_set ;       /* line 28639 */
    blocked |= new_set ;       /* line 28648 */
```

我不明白为什么 **sys_rt_sigprocmask** 不使用和 **sys_sigprocmask** 相同的方式实现，而且这样还可以节约一点效率。

中断

中断的名字十分形象，因为它们终止了系统正常的处理过程。在前面第 5 章中你就已经看到了中断的一个例子：提供系统调用基本机制的软件中断。在本章中，我们来了解一下硬件中断。

和系统调用中断一样，硬件中断也可能转化为内核模式运行然后返回。如果用户进程运行时发生了中断，系统就转化为内核模式，并且内核要对中断做出响应。接着，内核将控制返回给用户进程，用户进程能够从当时离开的位置继续运行。

同系统调用中断的另一个区别是硬件中断可能在内核已经在内核模式下运行时发生。这在系统调用中很少发生——通常内核不会麻烦地触发系统调用中断，因为它可以直接调用目标内核函数。如果中断发生时系统处于内核模式，结果就同在用户模式下的机制相一致——唯一的区别是内核自身所特有的执行过程而不是用户进程的执行过程暂时地被中断。

如果内核在一段时期内不希望被中断，那么就可以使用 **cli** 和 **sti** 函数（13105 行和 13104 行是 UP 版本；1216 行和 1229 行是 SMP 版本）屏蔽和开启中断。这些函数根据底层的 x86 指令命名：**cli** 代表“清除中断标志”，**sti** 代表“设置中断标志”。其工作方式和其名称类似：CPU 有一个“中断允许”标志，如果对其置位就允许中断，如果将其清空就禁止中断。因此，你可以使用 **cli** 清空这个标志从而禁止中断，也可以使用 **sti** 设置这个标志从而允许中断。在 UP 代码中，你可以选择调用两个等价的宏**__cli** 和**__sti**——分别见 13105 行和 13104 行。

当然，把内核移植到非 x86 平台上会使用不同的底层指令——在这些体系结构中 **cli** 和 **sti** 函数的实现都不相同。

IRQs

IRQ，或者中断请求，是从硬件设备发往 CPU 的中断信号。作为对 IRQ 的响应，CPU 跳转到某个地址——中断服务例行程序（***ISR***），更普通的情况是调用中断处理程序——内核在前面已经对这些处理程序进行了登记。中断处理程序是内核执行的为中断服务的函数；从中断处理程序中返回就继续执行中断前所在位置的代码。

IRQ 是有编号的，每一个硬件设备在系统中都对应一个 IRQ 号码。例如在 IBM PC 体系结构中，IRQ 0 就关联着一个每秒产生 100 次中断的定时器。把 IRQ 号码和设备关联起来，使得 CPU 可以区分每个中断是哪个设备产生的，从而允许它跳转到正确的中断处理程序。（在某些情况中，在一个系统中一个 IRQ 号可以被多个设备所共用，当然这不是非常普遍的情况。）

Bottom Halves

中断处理程序的下半部分（bottom half）是无须立即执行的部分。在某些中断之后，你甚至可能根本就不需要执行它。

给定的中断处理程序从概念上可以被分为上半部分（top half）和下半部分（bottom half）；在中断发生时上半部分的处理过程立即执行，但是下半部分（如果有的话）却推迟执行。这是通过把上半部分和下半部分处理为独立的函数并对其区别对待实现的。总之，上半部分要决定其相关的下半部分是否需要执行。不能推迟的部分显然不会属于下半部分，但是可以推迟的部分只是可能属于下半部分。

你也许会很奇怪为什么 Linux 会辛苦地把它们区分开——为什么要延迟呢？一个原因是要把中断的总延迟时间最小化。Linux 内核定义了两种类型的中断，快速的和慢速的，这两者之间的一个区别是慢速中断自身还可以被中断，而快速中断则不能。因此，当处理快速中断时，如果有其它中断到达——不管是快速中断还是慢速中断——它们都必须等待。为了尽可能快地处理这些其它的中断，内核就需要尽可能地将处理延迟到下半部分执行。

另外一个原因是，在最低层，当内核执行上半部分时，中断控制芯片将被告知禁止正在服务的这个特殊 IRQ（这和 CPU 级别的中断禁止不同，它把快速中断和慢速中断区别开来）。我们并不希望这种状态会持续地比需要的时间还长，因此只有上半部分中时间最为关键的部分才被处理，但是下半部分中其它的工作就要延迟处理了。

区分上下部分还有一个原因是处理程序的下半部分包含有一些中断所不一定非要处理的操作，只要内核可以在一系列设备中断之后可以从某些地方得到。在这种情况下，执行对于每个中断的下半部分的处理完全是一种浪费，它可以稍稍延迟并在后来只执行一次。

最后一段的一个暗示是值得说明的：没有必要每次中断都调用下半部分。相反，是上半部分（或者也可能是其它代码）简单地标记下半部分，通过设置某一位来指明下半部分必须执行。如果下半部分已经标记过需要执行了，现在又再次标记，那么内核就简单地保持这个标记；当情况允许的时候，内核就对它进行处理。如果在内核有机会运行其下半部分之前给定的设备就已经发生了 100 次中断，那么内核的上半部分就运行 100 次，下半部分运行 1 次。

下半部分在内核中有时候被认为是“软 IRQ”或者“软中断处理程序”，这有助于你理解今后要遇到的一些文件名和术语。

在本节的剩余内容中，我们将保持下半部分概念的抽象。下一节深入介绍定时器中断，包括其下半部分的处理，并展示了下半部分概念的一个有趣的滥用现象——我的意思是一个有趣的变种。

数据结构

同对信号量的处理一样，我们首先介绍一下中断和下半部分使用的重要的数据结构。图 6.1 阐述了这些数据类型之间的关系。

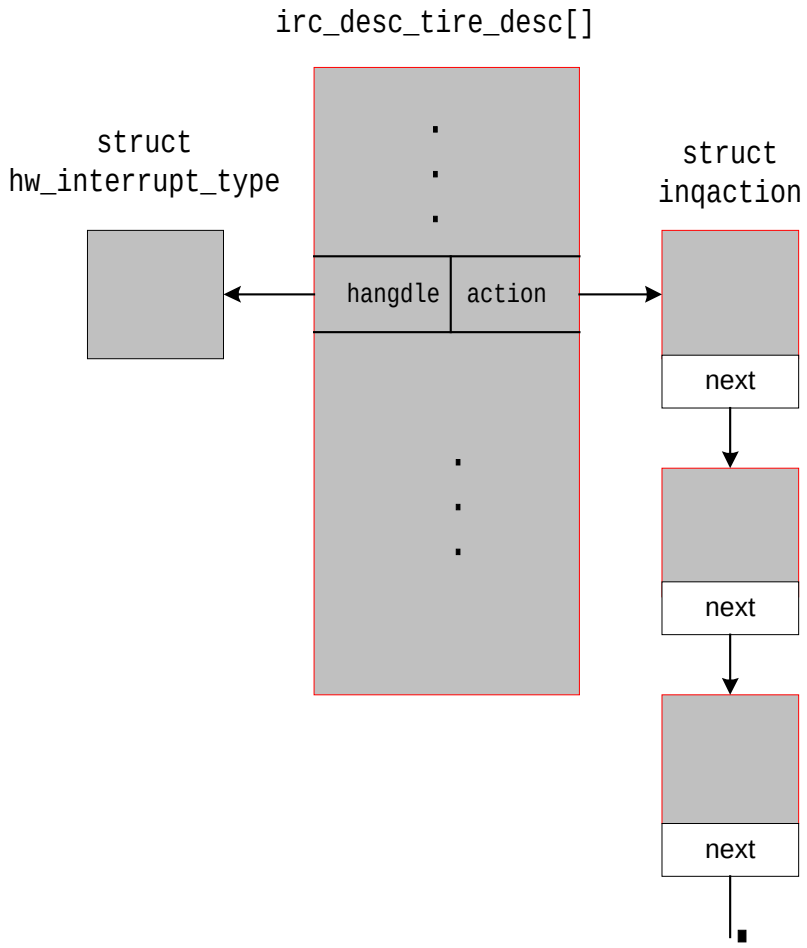

图 6.1 有关中断的数据结构

我们从这里开始，体系结构无关的头文件 linux/interrupt.h 定义了 **struct irqaction** 结构（14844 行），它代表了内核接收到特定 IRQ 之后应该采取的操作（在本章后面的部分中你将看到 **struct irqaction** 结构是如何与 IRQ 关联的）。其成员如下：

- **handler**——指向某一函数的指针，该函数是作为对中断的响应所执行的操作。
- **flags**——从与前面已经介绍过了的 **sa_flags** 相同的集合中提取出来；这个集合从 12108 行开始。该集合中仅仅为此目的而出现的值只有 **SA_INTERRUPT**（使用另外一个中断来中断这个中断也是可以的），**SA_SAMPLE_RANDOM**（考虑到这个中断也是源于物理随机性），和 **SA_SHIRQ**（这个 IRQ 和其它 **struct irqaction** 共享）。
- **mask**——在 x86 或者体系结构无关的代码中不会使用（除非将其设置为 0）；看起来只有在 SPARC64 的移植版本中要跟踪有关软盘的信息时才会使用它。
- **name**——生成中断的硬件设备的名字。由于不止一个硬件可以共享一个 IRQ，这在打印人工阅读程序时就有助于区分它们。
- **dev_id**——标识硬件类型的一个唯一的 ID——Linux 支持的所有硬件设备的每一种类型都有一个由制造厂商定义的在此成员中记录的设备 ID。其所有的可能值都是从一个巨大的集合中抽取出来的，这个集合在本书中没有介绍，因为它包含的内容是十分繁琐的，而且都是重复的——它仅仅是结构上类似于下面一小段代码的巨大宏定义块。

```
#define PCI_DEVICE_ID_S3_868 0x8880
#define PCI_DEVICE_ID_S3_928 0x88b0
#define PCI_DEVICE_ID_S3_864_1 0x88c0
#define PCI_DEVICE_ID_S3_864_2 0x88c2
```

在你看完这些的一部分之后，也就相当于将其完整的看了一下。可能你已经发现了，摘录的这部分内容是从包含针对基于 S3 的 PCI 显卡的设备 ID 的文件中选取的。虽然 **dev_id** 是一个指针，可它并不指向任何内容，但若将其解除参照就会引起错误。能够说明问题的是它的位结构模式。

- **next**——如果 IRQ 是共享的，那么这就是指向队列中下一个 **struct irqaction** 结构的指针。通常情况下，IRQ 不是共享的，因此这个成员就为空。

接下来我们感兴趣的两个数据结构存在于体系结构相关的文件 arch/i386/kernel/irq.h 中。第一个是 **struct hw_interrupt_type** 结构（1673 行），它是一个抽象的中断控制器。这是一系列的指向函数的指针，这些函数处理控制器特有的操作：

- **typename**——赋给控制器的人工可读的名字。
- **startup**——允许从给定的控制器的 IRQ 所产生的事件。
- **shutdown**——禁止从给定的控制器的 IRQ 所产生的事件。
- **handle**——根据提供给该函数的 IRQ 处理唯一的中断。
- **enable** 和 **disable**——这两个函数基本上和 **startup** 和 **shutdown** 相同；存在的差异对于本书中涉及的代码都不很重要。（实际上，对于本书中包含的所有代码来说，**enable**/**disable** 函数对和 **startup**/**shutdown** 函数对都是相同的。）

这个文件中我们感兴趣的另外一个数据结构是 **irq_desc_t**（1698 行），它具有如下成员：

- **status**——一个整数，它的位或者为 0，或者对应从 1685 行到 1689 行定义的集合中抽取出的标志。这些标志的集合代表了 IRQ 的状态——IRQ 是否被禁止了，有关 IRQ 的设备当前是否正被自动检测，等等。
- **handler**——指向 **hw_interrupt_type** 的指针。
- **action**——指向 **irqaction** 结构组成的队列的头。如同前面说明的一样，正常情况下每个 IRQ 只有一个操作，因此链接列表的正常长度是 1（或者 0）。但是，如果 IRQ 被两个或者多个设备所共享，那么这个队列中就有多个操作了。
- **depth**——**irq_desc_t** 的当前用户的个数。主要是用来保证事件正在处理的过程中 IRQ 不会被禁止。

irq_desc_t 是在 **irq_desc** 数组中（733 行）积聚起来的。对于每一个 IRQ 都有一个数组入口，因此数组把每一个 IRQ 映射到和它相关的处理程序和 **irq_desc_t** 中的其它信息上。

最后一个需要说明的数据结构集合从 29094 行开始，这些都与前面所讨论的下半部分有关：

- **bh_mask_count**（29094 行）——跟踪为每个下半部分提出的 **enable**/**disable** 请求嵌套对的数组。这些请求通过调用 **enable_bh**（12575 行）和 **disable_bh**（12568 行）实现。每个禁止请求都增加计数器；每个使能请求都减小计数器。当计数器达到 0 时，所有未完成的禁止语句都已经被使能语句所匹配了，因此下半部分最终被重新使能。
- **bh_mask** 和 **bh_active**（14856 行和 14857 行）——它们共同控制下半部分是否运行。它们两个都有 32 位，而每一个下半部分都占用一位。当一个上半部分（或者一些其它代码）决定其下半部分需要运行时，就通过设置 **bh_active**（12498 行中使用 **mark_bh**）中的一位来标记下半部分。不管是否经过了这样的标记，下半部

分可能会通过清空 **bh_mask** 中的相关位来整个跳过——通过调整 **bh_mask_count** 入口，**enable_bh** 和 **disable_bh** 完成了这个功能。
因此，对 **bh_mask** 和 **bh_active** 进行位 AND 运算就能够表明应该运行哪一个下半部分。特别是如果位与运算的结果是 0，就没有下半部分需要运行。这种技术在内核中多次使用，例如在宏 **get_active_bhs**（12480 行）中就使用了这种技术

- **bh_base**（14858 行）——这是一组简单的指向下半部分函数的指针。
- 未命名的 **enum**——从 14866 行开始的未命名的 **enum** 为内核使用的每一个下半部分指定了一个符号名称。例如，为了把计数器的下半部分标记为活动的，你可以这样的语句：
  mark_bh（TIME_BH）；
  27450 行的确就是这样处理的。

操作和 IRQ

一个经过仔细选择的小型函数集合处理了操作和 IRQ 之间的链接和解除链接。本节就是要讨论这些函数，以及那些从整体上对 IRQ 系统进行初始化的函数。

init_ IRQ

1597： **init_ IRQ** 初始化 IRQ 的处理。

1601： 符号 **CONFIG_X86_ISWS_APIC** 是为 SGI 虚拟工作站以及 SGI 的基于 x86 的工作站流水线而设置的。虽然同样基于 x86 的 CPU，虚拟工作站不能和基于 IBM PC 的体系结构共享很多其它特性——特别是如同你看到的，它们的中断处理有些不同。我们以后将忽略虚拟工作站所特有的代码。

1609： 建立中断描述符表，给 32 项到 95 项（十进制）赋缺省值。在这个过程中使用了 **set_nitr_gate**（6647 行），该函数很快就会介绍到。

1651： 建立 IRQ 2（级联中断）和 IRQ13（为 FPU 使用——请参看 955 行）。和这两个 IRQ 有关的 **irqaction** 结构分别是 **irq2**（979 行）和 **irq13**（974 行）。

init_ISA_irqs

1578： 该函数填充 **irq_desc** 数组，为 ISA 总线类型的机器（也就是所有标准 PC）初始化所有 IRQ。虽然该函数没有声明为 **static** 类型的，也没有使用**__initfunc** 标签标记，但是它只会被 **init_ IRQ** 调用。因此，只有在内核初始化过程中这个函数才是必要的。

1583： 对 **irq_desc** 中的每一个元素，系统为 **status**，**action** 和 **depth** 成员赋与了不会惹人反对的，也不会使人吃惊的的缺省值。

1589： 原来的（在 PCI 之前）IRQ 使用 **i8259A_irq_type**（723 行）处理。

1592： 编号比较高的 IRQ 初始化为 **no_irq_type**（701 行），这是一个必要的空处理程序。后来它们可能会改变——实际上，如果你使用了 PCI 卡，就确实会改变，就象现在的大多数 PC 一样。

set_intr_gate

6647： **set_intr_gate** 在 x86CPU 的中断描述符表（IDT）中建立一个项。在基于 x86 的系统中发生的每一个软件中断和硬件中断都有一个编号，这个编号被 CPU 用作是对这个表的索引。（包括系统调用中断——编号为 0x80——在第 5 章中我们已经介绍过了。）表中相关的项是中断发生时（内核）函数需要跳转到的地址。

setup_x86_irq

1388：**setup_x86_irq** 给指定的 IRQ 增加了一个操作（一个 **struct irqaction** 结构）。例如，在 6088 行使用它来记录定时器的中断。它还可以通过 **request_irq**（1439 行）使用，这在下一节介绍。

1398： Linux 使用了几种物理的随机源——例如中断——把一系列不可预知的值提供给设备/dev/random，这是一个有限却具有很高随机性的数据源，还有/dev/urandom，这是对应/dev/random 的无限的但是随机性较小的对应版本。随机系统作为一个整体在本书中并没有涉及，但是如果你不知道这个概念，这一大部分代码就会显得十分神秘。

1412： 如果现存的操作列表非空，**setup_x86_irq** 必须保证现存的操作和新的操作可以共享这个 IRQ。

1414：验证这个 IRQ 可以和其上现存的 **struct irqaction** 结构共享。这种测试是十分有效的，它部分是基于我们的一些认识：没有必要遍历执行队列中的所有操作，也没有必要检测它们可能共享的所有情况。除非这两个操作和第一个操作都允许共享 IRQ，否则不会允许第一个操作后的所有操作都进入队列。因此，如果第一个操作可以共享 IRQ，那么队列中的其它操作也就可以共享 IRQ；如果第一个操作不能共享，那么队列中的其它任何操作也都不能共享 IRQ。

1420： IRQ 正在被共享。**setup_x86_irq** 利用 **p** 向前执行操作队列直到末尾，离开时 **p** 指向队列的最后一个元素的 **next** 域。它也会增加 **shared** 标志的值，这将会在 1429 行中被使用。

1427： **p** 现在指向队列中的最后一个元素的 **next** 域，如果要共享 IRQ，或者 **p** 在不共享的情况下指向 **irq_desc[irq].action**——指向队列的头节点的指针。不管怎样，指针现在被设置为新的元素了。

1429： 如果还没有操作和这个 IRQ 关联，**irq_desc[irq]**的其它部分也就还没有设置，在这里就需要对其初始化了。特别要注意 1433 行中为这个 IRQ 调用了 **startup** 函数。

request_irq

1439：**request_irq** 从提供的值中创建一个 **struct irqaction** 结构，并将其加入对应给定的 IRQ 的 **struct irqaction** 列表中。（如果你对 C++和 Java 比较熟悉，可以把它当作是操作的构造函数。）它的实现非常简单明了。

1448： 对一对输入值进行健全性检测。注意没有必要测试 **irq** 是否小于 0，因为它是一个无符号数。

1453： 动态分配新的 **struct irqaction** 结构。为此目的使用的函数 **kmalloc** 在第 8 章中简单介绍。

1458： 填充新的操作并使用 **setup_x86_irq** 将其加入操作列表。

free_irq

1472： **free_irq** 是 **request_irq** 的补数（inverse）。如果 **request_irq** 类似于操作的构造函数，那么这就是操作的析构函数。

1481： 在确保 **irq** 在范围内以后，**free_irq** 找到有关的 **irq_desc** 项并且开始遍历操作列表。

1483： 除非它有正确的设备 ID，否则就忽略这个队列元素。

1487： 把这个元素从队列中分离出来并且释放其所占用的内存。

1489： 如果现在操作队列为空——也就是如果队列中只有唯一一个元素没有被链接——设备就会被关闭。

1495： 如果控制流程执行到这里，就意味着 **free_irq** 处理了整个操作列表而没有发现匹配

的 **dev_id**。如果发现了匹配对象，1493 行的 **goto** 语句就已经跳过了本行。因此，这个试图释放 IRQ 操作的努力是错误的；在这种情况下 **free_irq** 会打印出一条警告信息对当前状况进行描述。

prove_irq_on

1506： **probe_irq_on** 实现了内核 IRQ 自动探测的重要的一部分。阅读 14889 行开始的标题注释就得到了对整个进程的描述。根据描述我们知道这里要作的工作（只）是执行步骤三：暂时使能所有没有定义的 IRQ，以使得 **probe_irq_on** 的调用者可以检测它们。

1514： 对于除 IRQ 0 之外的每一个 IRQ，如果这个 IRQ 还没有与之相关的操作，**probe_irq_on** 会记录下这个 IRQ 正在自动探测的事实并启动关联设备。顺便说明一下，我不认为有任何的原因使这个循环向后执行。

1524： 忙等待约十分之一秒时间以允许生成伪中断的设备取消自己。

1530： 循环再次遍历所有的 IRQ，这一次要过滤出所有生成伪中断的设备。这个循环每次重复执行都是从 1 开始而不是从 0 开始，这是因为不需要自动检测的 IRQ 都被忽略掉了，而 IRQ0 是从来都不会自动检测的。速度在这里也是一个问题；在十分之一秒的延时之后——这是很长的一段时间，即使是从慢速的 CPU 的观点来看也是如此——一个循环或多或少都是有些不合理的。

1537： 如果设备在 1524 行的等待过程中触发了中断，这个中断可能就是伪中断：在此期间系统应该还没有和设备通讯过，因此设备也应该还没有和系统通讯过。因此自动探测位将被清空，处理程序再次关闭。

1544： 返回特殊数字 0x12345678，其原因将在下面的讨论中进行说明。

prove_irq_off

1547： **prove_irq_off** 实现了 IRQ 自动探测的另外一部分重要的内容。这里的工作是决定对探测到的哪一个 IRQ 做出响应，并返回其中的一个 IRQ。

1551： 检测名字很容易让人误解的参数 **unused** 和 **probe_irq_on** 返回的特殊数字是否相同。调用者假定象下面这样处理：

```
magic = prove_irq_on（）
/* … */
probe_irq_off（magic）;
```

如果偶然使用了其它的方法调用了 **probe_irq_off**（例如，如果由于其它一些逻辑调用者偶尔跳过了对 **probe_irq_on** 的调用），那么提供的参数可能不会包含正确的值。可能更重要的是这个参数给正在编写代码使用这个函数的程序员提供了一些信息：在研究其参数应该是什么的时候，你会发现在调用它的时候一直遵守的规则。这种规则很容易就被过度使用。

通过对紧随的错误消息的严格调整，似乎该函数的早期版本中可能已经在其参数中采用了调用者的地址。如果的确如此，这个测试就具有了第三种目的：把仍然不正确使用这个函数的调用者检测出来。

1557： 循环遍历所有的 IRQ，搜寻响应调用者探测的所有设备。这个循环也可以从 1 开始循环，这和前面讨论的 **probe_irq_on** 的原因是相同的。

1560： 内核没有试图自动检测这个 IRQ 上的任何内容；它跳到了下一个 IRQ。

1563： **IRQ_INPROGRESS** 标志指明了该 IRQ 的一个中断已经到达。由于 **probe_irq_on** 可能捕获所有的伪中断，假定这是对探测的真实响应。成功地自动检测到 IRQ 的数量因此而增一，同时保存第一次的数字。

1568： 不管是否成功自动探测到 IRQ，自动探测标志都要减少，并且再次结束处理程序。
1573： 如果不止一个 IRQ 被成功地自动探测到，就通过否定的 **irq_found** 来通知调用者。
1575： 返回 **irq_found**——0，或者（可能是经过求反的）第一个成功地自动探测到的 IRQ 号。注意如果发现了设备，则返回值决不会是 0，因为内核不会试图自动检测 IRQ 0。因此当没有自动检测到 IRQ 时，**probe_irq_off** 就返回 0。

硬件中断处理程序和下半部分

x86 系列的实际中断处理程序是微不足道的；在最低的层次上，这是通过反复使用 **BUILD_IRQ** 宏（1886 行）建立了一系列小汇编函数而实现的。**BUILD_ IRQ** 自己被 **BI** 宏调用（866 行），这个宏又顺次被 **BUILD_16_IRQS** 宏（869 行）调用，该宏在 878 行到 895 行的代码中用来建立汇编程序。这一连串的宏调用的目的仅仅是试图减少必须编写的代码数量和复杂度——我们应该使用 256 次对 **BUILD_IRQ** 的调用，而不是 16 次对 **BUILD_16_IRQS** 的调用。

汇编程序和如下代码相类似：

```
IRQ0x00_interrupt:
  push1 0x00-256
  jmp common_interrupt
```

也就是每一次都简单地把它的 IRQ 号（减去 256，原因在 1897 行有论述）压入堆栈并且跳转到正常的中断程序。

正常的中断处理程序是调用 **common_interrupt**，该函数也十分简短。它是使用 **BUILD_COMMON_IRQ** 宏（1871 行）建立的，在为 **do_IRQ** 进行安排之后简单调用 **do_IRQ** 返回给 **from_intr**（233 行）——这是第 5 章中介绍的系统调用的一部分。随后将要介绍的 **do_ IRQ**（1362 行）负责查看中断是否已经被处理了。

在介绍这些代码之前，从总体上观察一下在处理单个中断时这些部分如何组织在一起是很有帮助的：

1. CPU 跳转到 **IRQ0x*NN*_interrupt** 程序（其中的 *NN* 是中断号），它将其唯一的中断号压入堆栈并跳转到 **common_interrupt**。
2. **common_interrupt** 调用 **do_IRQ** 并保证当 **do_IRQ** 返回时控制流程能够转向 **ret_from_intr**。
3. **do_IRQ** 调用中断处理器芯片独有的代码——直接和芯片通讯的代码，如果需要就用它来处理中断。对于 PC 体系结构中流行的 8259A 控制器芯片，处理函数是 **do_8259A_IRQ**，在这里提到它仅仅是为了举一个例子而已。
4. **do_8259A_IRQ** 暂时禁止正在处理的特殊 IRQ，调用 **handle_IRQ_event**，接着重新使能这个 IRQ。
5. **handle_IRQ_event** 为慢速 IRQ 使能中断，或者为处理快速 IRQ 而将这些中断保持在禁止状态。接着遍历一个已经和这个 IRQ 建立联系的函数队列，并依次调用这些函数。由于中断为慢速 IRQ 而使能，这里就是慢速 IRQ 的处理程序可能被其它中断所中断的地方。在执行完队列中的所有函数之后，**handle_IRQ_event** 禁止中断并返回控制器所特有的处理函数，而该函数将返回到 **do_IRQ**。
6. **do_IRQ** 处理所有挂起等待的下半部分，接着返回。如同你已经知道的那样，它要返回到 **ret_from_intr**。第 5 章中介绍了从此之后的处理内容。

do_IRQ

1375： 更新内核的一些统计数字并调用与该 IRQ 相关的处理函数。对于一些老式 PC 上的标号较小的 IRQ 来说，其处理程序是 **handle_IRQ_event**。

1383： 如果下半部分是激活的，内核现在就使用 **do_bottom_half**（29126 行）对它们进行处理。

do_IRQ_event

1292：**do_IRQ_event** 为 **do_8259A_IRQ**（821 行）负担了大半重要的工作。本书中没有涉及到的其它一些代码也会调用这个函数。

1302：与其描述（12094 行）正好相反，**SA_INTERRUPT** 标志并不是一个 no_op。如果没有设置该标志，在接下来的代码中就允许中断。这是快速中断和慢速中断之间历史上遗留下来的区别，这一点我们已经讨论过了。（处理这两种类型中断的代码通常有很多差异，但是结果是相同的——代码已经被处理得更加出色了。）恰当的说，这个标志似乎是大多数情况下都为慢速设备使用——顾名思义，就是软盘设备。

1305： 通过调用每一个的处理函数来遍历执行这个 IRQ 的操作队列（队列头是由调用者提供的）。

1310： 这里的中断触发是用来为/dev/random 和/dev/urandom 增加一些随机信息——从大体上看来，大部分中断都是随机发生的。

1312： 禁止中断（当条件具备时调用者可以再次允许这些中断）。

do_bottom_half

29126：Linux 代码中有三处调用了 **do_bottom_half**：26707 行，243 行，1384 行。（你会发现，其中的两个是在体系结构特有的文件中的；在非 x86 平台体系结构特有的文件中对应部分也会调用这个函数。）因此，下半部分是用来处理如下三种情况的：

- 当决定随后哪一个进程应该获得 CPU 时。
- 当从系统调用中返回时。
- 在从 **do_IRQ** 中返回之前——也就是说，在每个中断之后。代码中的注释暗示了在内核的未来版本中不一定总在这里运行下半部分。

29130：下半部分的一个理想特性是在某一时刻只能有一个下半部分处于运行状态。这种特性在这里，也就是锁定（locking）在 UP 代码中体现出重要意义的位置之一，得到了强化。首先调用 **soft_trylock**（UP 版本在 12559 行——第 10 章中可以查看所有的 SMP 版本），只有在 **local_bh_count[cpu]**原来为 0 时这个函数才把 **local_bh_count [cpu]**设置为 1 并返回真值。根据 17479 行，对于 UP 来说 **cpu** 总是 0，而且你应该注意到 **softirq_trylock** 自己是不能被中断的，因为在这里中断已经被禁止了。**softirq_trylock** 和对应的 **softirq_endlock**（12561 行），就是仅仅为了以下目的而退出的，而不存在其它原因：协助保证这个下半部分不会被其它下半部分中断（虽然它们可以被上半部分所中断）。

29131：如果成功获得了锁，那么该函数就尝试另一个函数，10736 行的 **hardirq_trylock**。它只报告当前执行进程是否位于 **hardirq_enter/hardirq_exit** 对（10739 行和 10740 行）之间。对于 UP，这两者的定义是和 **irq_enter** 与 **irq_exit**（1810 行和 1811 行）两者的定义相同的；后两个函数在 **handle_IRQ_event** 中使用，当然在其它我们所没有讨论到的地方也对它们有所引用。这些宏协同工作来保证**__cli** 和**__sti** 对能够正确的进行嵌套——由于 CPU 不会嵌套使用这些宏，我们必须保证不会使用**__sti** 处理其它的**__cli**，而这也不是我们所希望的。

29132：没有其它下半部分在运行，而且 **do_bottom_half** 有权使能硬件中断。因此，它就使能硬件中断，运行下半部分，接着再次禁止中断。

29135：释放该函数已经获取的锁并返回。

run_bottom_halves

29110：现在内核能够运行挂起等待的下半部分。

29115：存储当前局部变量 **active** 中活动的——也就是被标记过的——下半部分的集合，并使用 **clear_active_bh** 宏（12481 行）清空全局变量 **bh_active** 中的设置位。对 **bh_active** 中这些位的清除将同时取消对所有下半部分的标记。

现在你就应该可以看到下半部分有时候是批量处理的，这一点在前面已经进行了没有论证的说明。此处中断是被使能的，因此如果在 **run_bottom_halves** 把 **bh_active** 拷贝到 **active** 之前就有中断触发并标记已经标记过的下半部分，那么上半部分就已经运行了两次，然而下半部分只运行了一次。还有，由于这个中断可以被自己中断，在下半部分运行一次的时候上半部分就已经运行了三次，等等。但是随着递归调用数量的增长，这很快就会变得不再可能了。

代码有可能忽略某个下半部分吗？假定中断在最坏的时刻发生：在 29115 行和 29116 行之间——也就是在拷贝 **bh_active** 之后，但是在清空其中的设置位之前。下面是三种可能的情况：

- *新的中断没有标记下半部分*。这种情况显然不会引起什么问题——中断之后处理的下半部分集合和前面的集合相同，因此 **run_bottom_halves** 仍将运行所有它应该运行的下半部分。
- *新的中断标记了一个已经标记过了的下半部分*。这种情况也不会引起问题——**run_bottom_halves** 不管怎样都要运行这个下半部分，而且在新中断返回之后就运行它。
- *新的中断标记了一个前面没有标记过的下半部分*。在这种情况下，当 **run_bottom_halves** 遍历执行所有的下半部分时，**active** 就不再和 **bh_active** 匹配了。然而，由于 **clear_active_bhs** 只会清空 **active** 集合中设置的位，所以 29116 行不会清空 **bh_active** 中新近标记的位。**clear_active_bhs** 使用 **atomic_clear_mask**（10262 行），后者简单的对 **active** 集合中的设置位进行位 AND 运算，而并不处理其余部分。因此，当 **run_bottom_halves** 执行循环时，就不会立刻对新近标记的下半部分进行处理；但是由于它的位仍然在 **bh_active** 中设置，**run_bottom_halves** 就仍然会在最后对它进行处理。更为准确的说法是，通过这种方法跳过的下半部分会在处理随后的一个定时器中断的过程中一起处理，这只是一个瞬间的延迟而已——或者如果有其它中断首先发生了，那么这段时间延迟会更短。因此，迷途的下半部分通常的等待时间不会超过百分之一秒，而且根据定义，下半部分毕竟不是时效性要求非常高的，这种少量延时不会引起任何问题。

29118：同时遍历执行 **bh_base** 数组和 **active** 中的位。如果 **active** 中最低的位被设置了，就调用相关的下半部分；接着循环推进到下一个 **bh_base** 项和 **active** 中的下一位继续执行。

因为这是一个 **do/while** 循环，所以它最少执行一次。部分原因是由于在调用 **do_bottom_half** 之前，调用者都要检测是否有下半部分需要处理，所以这个循环最少要执行一次。在一些情况下这种检测会执行到 **do_bottom_half** 本身中，但是如果没有下半部分需要运行，在调用之前执行测试就能够节省函数调用的开销。不管怎样，我们很容易就可以看出即使没有下半部分需要运行，这个循环也可以正确执行；虽然这会浪费一些时间，但是不这样就会引起错误。

29123：当 **active** 中没有任何位被设置时，循环就终止退出。由于在循环执行的过程中 **active** 是不断移位的，这样就同时测试了其余的位，而没有必要对它们的每一个都进行循环处理。

时间

本节通过观察一个中断的例子——定时器中断——的工作方式来使你能够将中断和下半部分的知识融会贯通起来。

定时器中断函数 **timer_interrupt** 是和 6086 行的 IRQ 0 相关的。此处使用的 **irq0** 变量是在 5937 行定义的。27972 行通过使用 **init_bh**（12484 行）把 **timer_bh** 函数注册为定时器的下半部分。

当触发 IRQ 0 时，**timer_interrupt** 从 CPU 时间戳计数器中读取一些属性值，如果 CPU 中有值（这在本书中没有涉及到的一些代码中使用），就调用 **do_timer_interrupt**（5758 行）。除了其它一些工作之外，它会调用 **do_timer**，这是定时器中断非常有趣的一部分。

do_timer

27446：从我们的出发点来看，这是我们感兴趣的定时器的上半部分。

27448：更新全局变量 **jiffies**，这个值记录了机器启动以来系统时钟跳动的次数。（显然，这实际上记录的是从定时器中断装载以来已经经过的定时器跳动的次数，定时器中断在系统启动的瞬间是不会发生的。）

27449：递增丢失的定时器跳动的数目——也就是那些没有被下半部分处理的定时器的跳动。很快你就会看到有定时器的下半部分是怎样使用这个变量的。

27450：上半部分已经运行了，因此其下半部分被标记为只要可能就运行。

27451：除了要记录从上一次定时器的下半部分运行以来定时器跳动发生的次数之外，我们还要知道有多少这种跳动发生在系统模式下。很快我们会再次看到为什么下半部分需要它；但是在目前，如果进程运行在系统（内核）模式下而不是用户模式下，那么只需要递增 **lost_ticks_system** 的值。

27453：如果定时器队列中有任务在等待，定时器队列的下半部分被标记为准备好运行（我们很快就会对定时器队列进行讨论）。而这就是整个定时器中断。虽然这看起来十分简单，但是这很大程度上是由于主要的工作都适当的延迟到下半部分处理了。

timer_bh

27439：这是定时器的下半部分。它调用函数为进程和内核本身更新有关时间的统计数字，并同时对老式内核定时器和新式内核定时器进行处理。

update_times

27412：这个函数主要是更新统计数字：计算系统的平均负载，更新记录当前时间的全局变量，并更新内核的当前进程使用的 CPU 时间的估计值。

27422：取得从上次下半部分运行以来发生的定时器跳动的数目，并重置计数器。

27427：如果数字非空——正常情况下都是这样——**update_times** 会找出有多少这种跳动发生在系统模式下。

27429：调用 **calc_load**（27135 行，马上就介绍）更新内核有关系统负载要素的估计值。

27430：调用 **calc_wall_time**（27311 行，后面会讨论）更新 **xtime**，它记录了当前的 wall_clock 时间。

27433：调用 **update_process_times**（27382 行），该函数同一个辅助函数 **update_one_process**（27371 行）共同作用，更新内核中有关当前进程已经运行的时间长度的统计。这些统计数字可以使用诸如 time，top，和 ps 之类的普通程序得到。现在你就可以看出，这些统计资料未必一定要正确：如果一个进程能够做到在所有时钟中断触发的时候都不处于运行状态，那么它就可以偷偷的使用大部分 CPU 而且还不会被认为使用了任何 CPU 资源。然而，还是比较难以理解为什么（或者怎样）恶意的进程要试图执行它，对于意图良好的进程来说，统计资料自然要进行平均——它们要为一些自己几乎不能使用的 CPU 定时器跳动负责，但是却不用为那些自己几乎在整个定时器跳动期间都在使用 CPU 的另一种情况负责，但是这些最终都不存在了。

update_wall_time

27313：为每一个必须要处理的跳动调用 **update_wall_time_one_tick**（27271 行）。它更新了全局变量 **xtime**，如果可能，就通过遵守网络时间协议（Network Time Protocol）努力将其和实际时间保持同步。

27318：将 **xtime** 标准化，使得微秒数在 0 到 999,999 的范围内。在极端的情况下可能会丢失多于一秒的定时器跳动，那么 **xtime** 的 **tv_usec** 部分就可能会超出 2 百万，这段代码就可能在标准化 **xtime** 时失败。但是随后的调用可以把 **xtime** 完全标准化。

calc_load

27135：**calc_load** 从定时器的下半部分中调用以更新内核对于当前系统负载的估计值。虽然对于它的介绍有点离题，但是对于每个对这个随处可见的数字是如何计算的感到疑惑的人都会对这个函数很感兴趣，因此它还是值得一看的。

27138：静态变量 **count** 记录了从上次计算平均负载以来已经经过了多少时间。它被初始化为 **LOAD_FREQ**，这在 16164 行中进行宏定义以代表相当于 5 秒时间的定时器间隔。对于这个函数的每一个调用都要递减遗留到下一次计算负载的定时器跳动数量。

27142：当剩余的跳动数小于 0 时，就应该重新计算了。（我倒是希望这个数是小于或者等于 0，而不是仅仅小于 0，但是实际上这两个值差不了多少。）

27143：为了能在另一 5 秒内可以再次触发，重新初始化 **count**，此后，**count_active_tasks**（27119 行）被用来监视系统中当前有多少任务。现在 **count_active_tasks** 的实现可能已经不是什么奇妙的事情了，但是你所有有关它的问题在阅读完第 7 章后都应该得到解答。

27144：在 16169 行定义的 **CALC_LOAD** 宏用来更新 **avenrun** 数组（27116 行）的三个项，它的三个元素分别记录了前面 5 秒、10 秒、15 秒的系统负载。与前一章中说明的一样，内核中尽量避免浮点数运算，因此这些计算都是在固定点进行的。

run_old_timers

27077：内核提供了类似于下半部分的已经不再使用了的技巧，通过这种技巧内核函数可以被登记到表中，和超时时间建立联系，并在指定时间到达时调用。这个函数根据需要调用其它函数。由于现在使用这个函数的唯一目的是为了支持原来的代码，所以我就不仔细地介绍它了。它简单的遍历处理 **timer_table** 数组（27109 行）的列表项，如果定时器已经触发就调用相关函数。

定时器队列

定时器队列最初的背景思想是与上半部分相关的下半部分并不一定非要与中断处理有关。相反的，它也可能是我们所需要周期性处理的任何内容。将一个函数定义为定时器中断处理程序下半部分的一部分能够保证这个函数每秒钟大约被调用 100 次。（如果每秒运行 100 次太频繁了，那么这个函数可以保持一个计数器并每 10 次调用都简单返回 9 次，例如——我们看到 **calc_load** 就是这样处理的。）结果就像是创建了一个进程，它的 **main** 部分在无穷循环中调用这个函数，于是它没有占用通常的和进程相关的开销。

但是，下半部分是一种有限的资源——只存在 32 个，这是因为我们希望 **bh_mask** 和 **bh_active** 各自都匹配一个无符号长整型数。我们可以通过使用系统中类似于实现信号量的方法来扩展下半部分的数量，但是这样仅仅能增加静态可用的下半部分的数量——它并不能使我们能够动态地扩展下半部分列表。

从本质上说，这是定时器队列所提供的内容：一个动态的可增长的下半部分的列表，所有项都和定时器中断有关。这里有一个独立的下半部分以处理这种情况——**TQUEUE_BH**——如同你看到的一样，如果定时器队列中有任务，它就和 **TIMER_BH** 一起被标记。因此，定时器中断有两个下半部分。

定时器队列实际上只是更普通的内核特性——任务队列的一个实例。根据说明文档，任务队列在内核本身中是相当自由的——请参看文件 include/linux/tqueue.h 从 18527 行开始的部分。因此，接下来我们就不再介绍它们了。但是，它们是很重要的内核服务，而且那个简短的文件也很值得一读。