第 5 章 系统调用

大部分介绍 Unix 内核的书籍都没有仔细说明系统调用，我认为这是一个失误。实际上，我们实际需要的系统调用现在已经十分完美。因此，从某种意义上来说，研究系统调用的实现是无意义的——如果你想为 Linux 内核的改进贡献自己的力量，还有其它许多方面更值得投入精力。

然而，对于我们来说，仔细研究少量系统调用是十分值得的。这样就有机会初步了解一些概念，这些概念将随本书发展而进行详细介绍，例如进程处理和内存。这使得你可以趁机详细了解一下 Linux 内核编程的特点。这包括一些和你过去在学校里（或工作中）所学的内容不同的方法。和其它编程任务相比，Linux 内核编程的一个显著特点是它不断同三个成见进行斗争——这三个成见就是速度、正确和清晰——我们不可能同时获取这三个方面…至少并不总是能够。

什么是系统调用

系统调用发生在用户进程（比如 emacs）通过调用特殊函数（例如 **open**）以请求内核提供服务的时候。在这里，用户进程被暂时挂起。内核检验用户请求，尝试执行，并把结果反馈给用户进程，接着用户进程重新启动，随后我们就将详细讨论这种机制。

系统调用负责保护对内核所管理的资源的访问，系统调用中的几个大类主要有：处理 I/O 请求（**open**，**close**，**read**，**write**，**poll** 等等），进程（**fork**，**execve**，**kill**，等等），时间（**time**，**settimeofday** 等等）以及内存（**mmap**，**brk**，等等）的系统调用。几乎所有的系统调用都可以归入这几类中。

然而，从根本上来说，系统调用可能和它表面上有所不同。首先，在 Linux 中，C 库中对于一些系统调用的实现是建立在其它系统调用的基础之上的。例如，**waitpid** 是通过简单调用 **wait4** 实现的，但是它们两个都是作为独立的系统调用说明的。其它的传统系统调用，如 **sigmask** 和 **ftime** 是由 C 库而不是由 Linux 内核本身实现的；即使不是全部，至少大部分是如此。

当然，从技巧的一面来看这是无害的——从应用程序的观点来看，系统调用就和其它的函数调用一样。只要结果符合预计的情况，应用程序就不能确定是否真正使用到了内核。（这种处理方式还有一个潜在的优点：用户可以直接触发的内核代码越少，出现安全漏洞的机会也就越少。）但是，由于使用这种技巧所引起的困扰将会使我们的讨论更为困难。实际上，系统调用这一术语通常被演讲者用来说明在第一个 Unix 版本中的任何对系统的调用。但是在本章中我们只对“ 真正 ”的系统调用感兴趣——真正的系统调用至少包括用户进程对部分内核代码的调用。

系统调用必须返回 **int** 的值，并且也只能返回 **int** 的值。为了方便起见，返回值如果为零或者为正，就说明调用成功；为负则说明发生了错误。就像老练的 C 程序员所知道的一样，当标准 C 库中的函数发生错误时会通过设置全局整型变量 **errno** 指明发生错误的属性，系统调用的原理和它相同。然而，仅仅研究内核源程序代码并不能够获得这种系统调用方式的全部意义。如果发生了错误，系统调用简单返回自己所期望的负数错误号，其余部分则由标准 C 库实现。（正常情况下，用户代码并不直接调用内核系统函数，而是要通过标准 C 库中专门负责翻译的一个小层次（thin layer）实现。）我们随便举一个例子，27825 行（**sys_nanosleep** 的一部分）返回**-EINVAL** 指明所提供的值越界了。标准 C 库中实际处理

sys_nanosleep 的代码会注意到返回的负值，从而设置 **errno** 和 **EINVAL**，并且自己返回-1 给原始的调用者。

在最近的内核版本中，系统调用返回负值偶尔也不一定表示错误了。在目前的几个系统调用中（例如 **lseek**），即使结果正确也会返回一个很大的负值。最近，错误返回值是在-1 到-4095 范围之内。现在，标准 C 库实现能够以更加成熟和高级的方式解释系统调用的返回值；当返回值为负时，内核本身就不用再做任何特殊的处理了。

中断、内核空间和用户空间

我们将在第 6 章中介绍中断和在第 8 章中介绍内存时再次明确这些概念。但是在本章中，我们只需要粗略地了解一些术语。

第一个术语是中断（interrupt），它来源于两个方面：硬件中断，例如磁盘指明其中存放一些数据（这与本章无关）；和软件中断，一种等价的软件机制。在 x86 系列 CPU 中，软件中断是用户进程通知内核需要触发系统调用的基本方法（出于这种目的使用的中断号是 0x80，对于 Intel 芯片的研究者来说更为熟悉的是 INT 80）。内核通过 **system_call**（171 行）函数响应中断，这一点我们马上就会介绍。

另外两个术语是内核空间（kernel space）和用户空间（user space），它们分别对应内核保留的内存和用户进程保留的内存。当然，多用户进程也经常同时运行，而且各个进程之间通常不会共享它们的内存，但是，任何一个用户进程使用的内存都称为用户空间。内核在某一个时刻通常只和一个用户进程交互，因此实际上不会引起任何混乱。

由于这些内存空间是相互独立的，用户进程根本不能直接访问内核空间，内核也只能通过 **put_user**（13278 行）和 **get_user**（13254 行）宏和类似的宏才可以访问用户空间。因为系统调用是进程和进程所运行的操作系统之间的接口，所以系统调用需要频繁地和用户空间交互，因此这些宏也就会不时的在系统调用中出现。在通过数值传递参数的情况下并不需要它们，但是当用户把指针——内核通过这个指针进行读写——传递给系统调用时，就需要这些宏了。

如何激活系统调用

系统调用的的激活有两种方法：**system_call** 函数和 **lcall7** 调用门（call gate）（请参看 135 行）。（你可能听说过还有一种机制，**syscall** 函数，是通过调用 **lcall7** 实现的——至少在 x86 平台上是如此——因此，它并不是一个特有的方法。）本节将细致地讨论一下这两种机制。

在阅读的过程中请注意系统调用本身并不关心它们是由 **system_call** 还是由 **lcall7** 激活的。这种把系统调用和其实现方式区别开来的方法是十分精巧的。这样，如果出于某种原因我们不得不增加一种激活系统调用的方法，我们也不必修改系统调用本身来支持这种方法。

在你浏览这些汇编代码之前要注意这些机器指令中操作数的顺序和普通 Intel 的次序相反。虽然还有一些其它的语法区别，但是操作数反序是最令人迷惑的。如果你还记得 Intel 的语法：

```
mov eax, 0
```

（本句代码的意思是把常数 0 传送到寄存器 EAX 中）在这里应该写作：

```
mov1 $0, %eax
```

这样你就能够正确通过。（内核使用的语法是 AT&T 的汇编语法。在 GNU 汇编文档中有更多资料。）

system_call

system_call（171 行）是所有系统调用的入口点（这是对于内部代码来说的；**lcall7** 用来支持 iBCS2，这一点我们很快就会讨论）。正如前面标题注释中说明的一样，目的是为普通情况简单地实现直接的流程，不采用跳转，因此函数的各个部分都是离散的——整体的流量控制已经因为要避免普通情况下的多分支而变得非常复杂。（分支的避免是十分值得的，因为它们引起的代价非常昂贵。它们可以清空 CPU 管道，使现存 CPU 的并行加速机制失效。）

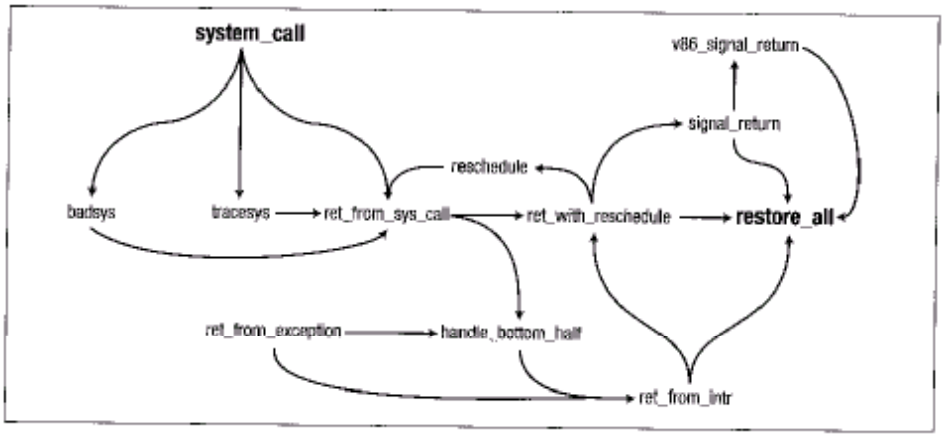

图 5.1　**system_call** 的流程控制

图 5.1 显示了作为 **system_call** 的一部分出现的分支目标标签以及它们之间的流程控制方向，该图可以在你阅读本部分讨论内容时提供很大的帮助。图中 **system_call** 和 **restore_all** 两个标签比其它标签都要大，因为这两处是该函数正常的出口点和入口点；然而，还有另外两个入口点，这一点在本章的后续内容中很快就可以看到。

system_call 是由标准 C 库激活的，该标准 C 库会把自己希望传递的参数装载到 CPU 寄存器中，并触发 0x80 软件中断。（**system_call** 在这里是一个中断处理程序。）内核记录了软件中断和 6828 行的 **system_call** 函数的联系（**SYSCALL_VECTOR** 是在 1713 行宏定义为 0x80 的）。

system_call

172：　**system_call** 的第一个参数是所希望激活的系统调用的数目；它存储在 EAX 寄存器中。**system_call** 还允许有多达四个的参数和系统调用一起传送。在一些极其罕见的情况下使用四个参数的限制是负担繁重的，通常可以建立一个指向结构的指针参数来巧妙地完成同样功能，指针指向的结构中可以包含你所需要的一切信息。
随后可能需要 EAX 值的一个额外拷贝，因此通过将其压栈而保存起来；这个值就是 218 行的 **ORIG_EAX**（%**esp**）表达式的值。

173：　**SAVE_ALL** 宏是在 85 行定义的；它把所有寄存器的值压入 CPU 的堆栈。随后，就在 **system_call** 返回之前，使用 **RESTALL_ALL**（100 行）把栈中的值弹出。在这中间，**system_call** 可以根据需要自由使用寄存器的值。更重要的是，任何它所调用的 C 函数都可以从栈中查找到所希望的参数，因为 **SAVE_ALL** 已经把所有寄存器的值都压入栈中了。
结果栈的结构从 26 行开始描述。象 **0（%esp）**和 **4（%esp）**一样的表达式指明了堆栈指针（ESP 寄存器）的一种替换形式——分别表示 ESP 上的 0 字节，ESP 上的 4 字节，等等。特别要注意的是在前面一行中压入堆栈的 EAX 的拷贝已经变成本标

题注释作为 **orig_eax** 所描述的内容；它们是由 **SAVE_ALL** 压入寄存器之上的堆栈的（**orig_eax** 之上的寄存器在这里早已就绪了）。

还需注意：这可能有点令人迷惑——由于我们调用 **orig_eax** 时 EAX 的拷贝已经压入了堆栈，它是否有可能在其它寄存器下面而不是在其它寄存器上面呢？答案既是肯定的，也是否定的。x86 的堆栈指针寄存器 ESP 在有数据压入堆栈时会减少——堆栈会向内存低地址发展。因此，**orig_eax** 逻辑上是在其它值的下面，但是物理上却是在其它值的上面。

从 51 行开始的一系列宏有助于使这些替换更容易理解。例如，**EAX（%esp）**就和 **18（%esp）**相同——然而前一种方法通过表达式引用存储在堆栈中的 EAX 寄存器副本的决定可以使整个过程更加简单。

174： 从 EBX 寄存器中取得指向当前任务的指针。完成这个工作的宏 **GET_CURRENT**（131 行）对于在大部分代码中使用的 C 函数 **get_current**（10277 行）来说是一个无限循环。

此后，当看到类似于 **foo（%ebx）**或者 **foo（%esp）**的表达式时，这意味着这些的代码正在引用代表当前进程的结构的字段——16325 行的 **struct task_struct**——这在第 7 章中将对它进行更详细的介绍。（更确切的描述是， **%ebx** 的置换在 **struct task_struct** 中，**%esp** 的置换在与 **struct task_struct** 相关联的 **struct pt_regs** 结构中。但是这些细节在这里都并不重要。）

175： 检查（EAX 中的）系统调用的数目是否超过系统调用的最大数量。（此处 EAX 为一个无符号数，因此不可能为负值。）如果的确超过了，就向前跳转到 **badsys**（223 行）。

177： 检测系统调用是否正被跟踪。如 strace 之类的程序为有兴趣的人提供了系统调用的跟踪工具，或者额外的调试信息：如果能够监测到正在执行的系统调用，那么你就可以了解到当前程序正在处理的内容。如果系统调用正被跟踪，控制流程就向前跳转到 **tracesys**（215 行）。

179： 调用系统函数。此处有很多工作需要处理。首先，**SYSMOL_NAME** 宏不处理任何工作，只是简单的为参数文本所替换，因此可以将其忽略。**sys_call_table** 是在当前文件（arch/i386/kernel/entry.S）的末尾从 373 行开始定义的。这是一张由指向实现各种系统调用的内核函数的函数指针组成的表。

本行中第二对圆括号中包含了三个使用逗号分割开的参数（第一个参数为空）；这里就是实现数组索引的地方。当然，这个数组是以 **sys_call_table** 作为索引的，这称为偏移（displacement）。这三个参数是数组的基地址、索引（EAX，系统调用的数目）和大小，或者每个数组元素中的字节数——在这里就是 4。由于数组基地址为空，就将其当作 0——但是它要和偏移地址，**sys_call_table，**相加，简单的说就是 **sys_call_table** 被当作数组的基地址。本行基本上等同于如下的 C 表达式：

```
/* Call a function in an array of functions. */
(sys_call_table[eax])();
```

然而， C 当然还要处理许多繁重的工作，例如为你记录数组元素的大小。不要忘记，系统调用的参数早已经存储在堆栈中了，这主要由调用者提供给 **system_call** 并使用 **SAVE_ALL** 把它们压栈。

180： 系统调用已经返回。它在 EAX 寄存器中的返回值（这个值同时也是 **system_call** 的返回值）被存储起来。返回值被存储在堆栈中的 EAX 内，以使得 **RESTORE_ALL** 可以迅速地恢复实际的 EAX 寄存器以及其它寄存器的值。

182： 接下来的代码仍然是 **system_call** 的一部分，它是一个也可以命名为 **ret_from_sys_call** 和 **ret_from_intr** 的独立入口点。它们偶尔会被 C 直接调用，也可

以从 **system_call** 的其它部分跳转过来。

185： 接下来的几行检测“下半部分（bottom half）”是否激活；如果激活了，就跳转到 **handle_bottom_half** 标号（242 行）并立即开始处理。下半部分是中断进程的一部分，将在下一章中讨论。

189： 检查该进程是否为再次调度做了标记（记住表达式**$0** 就是常量 0 的系统简单表示）。如果的确如此，就跳转到 **reschedule** 标号（247 行）。

191： 检测是否还有挂起的信号量，如果有的话，下一行就向前跳转到 **signal_return**（197 行）。

193： **restore_all** 标号是 **system_call** 的退出点。其主体就是简单的 **RESTORE_ALL** 宏（100 行），该宏将恢复早先由 **SAVE_ALL** 存储的参数并返回给 **system_call** 的调用者。

197： 当 **system_call** 从系统调用返回前，如果它检测到需要将信号量传送给当前的进程时，才会执行到 **signal_return**。它通过使中断再次可用开始执行，有关内容将在第 6 章中介绍。

199： 如果返回虚拟 8086 模式（这不是本书的主题），就向前跳转到 **v86_signal_return**（207 行）。

202： **system_call** 要调用 C 函数 **do_signal**（3364 行，在第 6 章中讨论）来释放信号量。**do_signal** 需要两个参数，这两个参数都是通过寄存器传递的；第一个是 EAX 寄存器，另一个是 EDX 寄存器。**system_call**（在 200 行）早已把第一个参数的值赋给了 EAX；现在，就把 EDX 寄存器和寄存器本身进行 XOR 操作，从而将其清 0，这样 **do_signal** 就认为这是一个空指针。

203： 调用 **do_signal** 传递信号量，并且跳回到 **restore_all**（193 行）结束。

207： 由于虚拟 8086 模式不是本书的主题，我们将忽略大部分 **v86_signal_return**。然而，它和 **signal_return** 的情况非常类似。

215： 如果当前进程的系统调用正由其祖先跟踪，就像 strace 程序中那样，那么就可以执行到 **tracesys** 标号。这一部分的基本思想如同 179 行一样是通过 **syscall_table** 调用系统函数，但是这里把该调用和对 **syscall_trace** 函数的调用捆绑在一起。后面的这个函数在本书中并没有涉及到，它能够中止当前进程并通知其祖先注意当前进程将要激活一个系统调用。

EAX 操作和这些代码的交错使用最初可能容易令人产生困惑。**system_call** 把存储在堆栈中的 EAX 拷贝赋给**-ENOSYS**，调用 **syscall_trace**，在 172 行再从所做的备份中恢复 EAX 的值，调用实际的系统调用，把系统调用的返回值置入堆栈中 EAX 的位置，再次调用 **syscall_trace**。

这种方式背后的原因是 **syscall_trace**（或者更准确的说是它所要使用到的跟踪程序）需要知道在它是在实际系统调用之前还是之后被调用的。**-ENOSYS** 的值能够用来指示它是在实际系统调用执行之前被调用的，因为实际中所有实现的系统调用的执行都不会返回**-ENOSYS**。因此，EAX 在堆栈中的备份在第一次调用 **syscall_trace** 之前是**-ENOSYS**，但是在第二次调用 **syscall_trace** 之前就不再是了（除非是调用 **sys_ni_syscall** 的时候，在这种情况下，我们并不关心是怎样跟踪的）。218 行和 219 行中 EAX 的作用只是找出要调用的系统调用，这和无须跟踪的情况是一致的。

222： 被跟踪的系统调用已经返回；流程控制跳转回 **ret_from_sys_call**（184 行）并以与普通的无须跟踪的情况相同的方式结束。

223： 当系统调用的数目越界时，就可以执行到 **badsys** 标号。在这种情况下，**system_call** 必须返回**-ENOSYS**（**ENOSYS** 在 82 行将它赋值为 38）。正如前面提到的一样，调用者会识别出这是一个错误，因为返回值在-1 到-4,095 之间。

228： 在诸如除零错误（请参看 279 行）之类的 CPU 异常中断情况下将执行到 **ret_from_exception** 标号；但是 **system_call** 内部的所有代码都不会执行到这个标号。如果有下半部分是激活的，现在就是它在起作用了。

233： 处理完下半部分之后或者从上面的情况简单的执行下来（虽然没有下半部分是激活的，但是同样也触发了 CPU 异常），就执行到了 **ret_from_intr** 标号。这是一个全局符号变量，因此可能在内核的其它部分也会有对它的调用。

237： 被保存的 CPU 的 EFLAGS 和 CS 寄存器在此已经被并入 EAX，因而高 24 位的值（其中恰好包含了一位在 70 行定义的非常有用的 **VM_MASK**）来源于 EFLAGS，其它低 8 位的值来源于 CS。该行隐式的同时对这两部分进行测试以判断进程到底返回虚拟 8086 模式（这是 **VM_MASK** 的部分）还是用户模式（这是 3 的部分——用户模式的优先等级是 3）。下面是近似的等价 C 代码：

```
/* Mix eflags and cs in eax. */
eax = eflags & ~0xff;
eax |= cs & ~0xff
/* Simultaneously test lower 2 bits
* and VM_MASK bit. */
if    (eax & (VM_MASK | 3))
    goto ret_with_reschedule;
goto restore_all;
```

238： 如果这些条件中有一个能得到满足，流程控制就跳转到 **ret_with_reschedule**（188 行）标号来测试在 **system_call** 返回之前进程是否需要再次调度。否则，调用者就是一个内核任务，因此 **system_call** 通过跳转到 **restore_all** (193 行)来跳过重新调度的内容。

242： 无论何时 **system_call** 使用一个下半部分服务时都可以执行到 **handle_bottom_half** 标号。它简单的调用第 6 章中介绍的 C 函数 **bottom_half**（29126 行），然后跳回到 **ret_from_intrr**（233 行）。

248： **system_call** 的最后一个部分在 **reschedule** 标号之下。当产生系统调用的进程已经被标记为需要进行重新调度时，就可以执行到这个标号；典型地，这是因为进程的时间片已经用完了——也就是说，进程到目前为止已经尽可能的拥有 CPU 了，应该给其它进程一个机会来运行了。因此，在必要的情况下就可以调用 C 函数 **schedule**（26686 行）交出 CPU，同时流程控制转回 249 行。CPU 调度是第 7 章中讨论的一个主题。

lcall7

Linux 支持 Intel 二进制兼容规范标准的版本 2（iBCS2）。（iBCS2 中的小写字母 i 显然是有意的，但是该标准却没有对此进行解释；这样看来似乎和现实的 Intel 系列的 CPU 例如 i386，i486 等等是一致的。）iBCS2 的规范中规定了所有基于 x86 的 Unix 系统的应用程序的标准内核接口，这些系统不仅包括 Linux，而且还包括其它自由的 x86 Unix（例如 FreeBSD），也还包括 Solaris/x86，SCO Unix 等等。这些标准接口使得为其它 Unix 系统开发的二进制商业软件在 Linux 系统中能够直接运行，反之亦然（而且，近期新开发软件向其它 Unix 移植的情况越来越多）。例如，Corel 公司的 WordPerfect 的 SCO Unix 的二进制代码在还没有 Linux 的本地版本的 WordPerfect 之前就可以使用 iBCS2 在 Linux 上良好地运行。

iBCS2 标准有很多组成部分，但是我们现在关心的是这些系统调用如何协调一致来适应这些迥然不同的 Unix 系统。这是通过 **lcall7** 调用门实现的。它是一个相当简单的汇编函数

（尤其是和 **system_call** 相比而言更是如此），仅仅定位并全权委托一个 C 函数来处理细节。（调用门是 x86 CPU 的一种特性，通过这种特性用户任务可以在安全受控的模式下调用内核代码。）这种调用门在 6802 行进行设定。

lcall7

136： 前面的几行将通过调整处理器堆栈以使堆栈的内容和 **system_call** 预期的相同——**system_call** 中的一些代码将会完成清理工作，这样所有的内容都可以连续存放了。

145： 基于同样的思想，**lcall7** 把指向当前任务的指针置入 EBX 寄存器，这一点和 **system_call** 的情况是相同的。但是，它的执行方式却与 **system_call** 不同，这就比较奇怪了。这三行可以等价地按如下形式书写：

push1 %esp

GET_CURRENT（%ebx）

这种实现的执行速度并不比原有的更快；在将宏展开以后，实际上这还是同样的三条指令以不同的次序组合在一起而已。这样做的优点是可以和文件中的其它代码更为一致，而且代码也许会更清晰一些。

148： 取得指向当前任务 **exec_domain** 域的指针，使用这个域以获取指向其 **lcall7** 处理程序的指针，接着调用这个处理程序。

本书中并没有对执行域（execution domains）进行详细说明——但是简单说来，内核使用执行域实现了部分 iBCS2 标准。在 15977 行你可以找到 **struct exec_domain** 结构。**default_exec_domain**（22807 行）是缺省的执行域，它拥有一个缺省的 **lcall7** 处理程序。它就是 **no_lcall7**（22820 行）。其基本的执行方式类似于 SVR4 风格的 Unix，如果调用进程没有成功，就传送一个分段违例信号量（segmentation violation signal）给调用的进程，。

152： 跳转到 **ret_from_sys_call** 标号（184 行——注意这是在 **system_call** 内部的）清除并返回，就像是正常的系统调用一样。

系统调用样例

现在你已经知道了系统调用是如何激活的，接下来我们将通过几个系统调用例子的剖析来了解一下它们的工作方式。注意系统调用 **foo** 几乎都是使用名为 **sys_foo** 的内核函数实现的，但是在某些情况下该函数也会使用一个名为 **do_foo** 的辅助函数。

sys_ni_syscall

29185：**sys_ni_syscall** 的确是最简单的系统调用；它只是简单的返回 **ENOSYS** 错误。最初的时候这可能显得没有什么作用，但是它的确是有用的。实际上，**sys_ni_syscall** 在 **sys_call_table** 中占据了很多位置——而且其原因并不只有一个。开始的时候，**sys_ni_syscall** 在位置 0（374 行），因为如果漏洞百出的代码错误地调用了 **system_call**——例如，没有初始化作为参数传递给 **system_call** 的变量——在这种偶然的变量定义中，0 是最可能的值。如果我们能够避免这种情况，那么在错误发生时就不用采取象杀掉进程一样的剧烈措施。（当然，只要允许有用工作的进行，就不可能防止所有的错误。）这种使用表的元素 0 作为抵御错误的手段在内核中被作为良好的经验而广泛使用。

而且，你还会发现 **sys_ni_syscall** 在表中明显出现的地方就多达十几处。这些条目代表了那些已经从内核中移出的系统调用——例如在 418 行，就代替了已经废弃了的

prof 系统调用。我们不能简单地把另外的实际系统调用放在这里，因为老的二进制代码可能还会使用到这些已经废弃了的系统调用号。如果一个程序试图调用这些老的系统调用，但是结果却与预期的完全不同，例如打开了一个文件，这会比较令人感到奇怪的。

最后，**sys_ni_syscall** 将占据表尾部所有未用的空间；这一点是在从 572 行到 574 行的代码实现的，它根据需要重复使用这些项来填充表。由于 **sys_ni_syscall** 只是简单返回 **ENOSYS** 错误号，对它的调用和跳转到 **system_call** 中的 **badsys** 标号作用是相同的——也就是说，使用指向这些表项的系统调用号和在表外对整个表进行全部索引具有相同的作用。因此，我们不用改变 **NR_syscalls** 就可以在表中增加（或者删除）系统调用，但是其效果与我们真的对 **NR_syscalls** 进行了修改一样（不管怎样，这都是由 **NR_syscalls** 所建立的限制条件所决定的）。

到现在也许你已经猜到了，**sys_ni_syscall** 中的“ ni ”并不是指 Monty Python 的“ 说‘ Ni ’ 的骑士 ”；而是指“ not implemented（没有实现）”这一相较而言并不太诙谐的短语。

对于这个简单的函数我们需要研究的另外一个问题是 **asmlinkage** 标签。这是为一些 gcc 功能定义的一个宏，它告诉编译器该函数不希望从寄存器中（这是一种普通的优化 ）取得任何参数，而希望仅仅从 CPU 堆栈中取得参数。回忆一下我们前面提到过 **system_call** 使用第一个参数作为系统调用的数目，同时还允许另外四个参数和系统调用一起传递。**system_call** 通过把其它参数（这些参数是通过寄存器传递过来的）滞留在堆栈中的方法简单的实现了这种技巧。所有的系统调用都使用 **asmlinkage** 标签作了标记，因此它们都要查找堆栈以获得参数。当然，在 **sys_ni_syscall** 的情况下这并没有任何区别，因为 **sys_ni_syscall** 并不需要任何参数。但是对于其它大部分系统调用来说这就是个问题了。并且，由于在其它很多函数前面都有 **asmlinkage** 标签，我想你也应该对它有些了解。

sys_time

31394：**sys_time** 是包含几个重要概念的简单系统调用。它实现了系统调用 **time**，返回值是从某个特定的时间点（1970 年 1 月 1 日午夜 UTC）以来经过的秒数。这个数字被作为全局变量 **xtime**（请参看 26095 行；它被声明为 **volatile** 型的变量，因为它可以通过中断加以修改，这一点我们在第 6 章中就会看到）的一部分，通过 **CURRENT_TIME** 宏（请参看 16598 行）可以访问它。

31400：该函数非常直接的实现了它的简单定义。当前时间首先被存储在局部变量 **i** 中。

31402：如果所提供的指针 **tloc** 是非空的，返回值也将被拷贝到指针指向的位置。该函数的一个微妙之处就在于此；它把 i 拷贝到用户空间中而不是使用 **CURRENT_TIME** 宏来重新对其进行计算，这基于两个原因：

- **CURRENT_TIME** 宏的定义以后可能会改变，新的实现方法可能会由于某种原因而速度比较慢，但是对于 i 的访问至少应该和 **CURRENT_TIME** 宏展开的速度同样快。
- 使用这种方式处理，确保结果的一致性：如果代码刚好执行到 31400 行和 31402 行之间时时间发生了改变，**sys_time** 可能把一个值拷贝到***tloc** 中，但是在结束之后却返回另一个值。

另外还有一个小的方面需要注意，此处的代码不使用**&&**来编写而是使用两个 **if**，这可能有一点令人奇怪。内核中采用这些看起来非常特殊的代码的一般原因都是由于速度的要求，但是 gcc 为**&&**版本和两个 **if** 版本的代码生成的代码是等同的，因此这里的原因就不可能是速度的要求——除非这些代码是在早期 gcc 版本下开发的，

这样才有些意义。

31403：如果 **sys_time** 不能访问所提供的位置（一般都是因为 **tloc** 无效），它就把**-EFAULT** 的值赋给 **i**，从而在 31405 行返回错误代码。

31405：为调用者返回的 **i** 或者是当前时间，或者是**-EFAULT**。

sys_reboot

29298：内核中其他地方可能都没有 **sys_reboot** 的实现方法这样先进。其原因是可以理解为：根据调用的名字我们就可以知道，**reboot** 系统调用可以用来重新启动机器。根据所提供的参数，它还能够挂起机器，关闭电源，允许或者禁止使用 Ctrl+Alt_Del 组合键来重启机器。如果你要使用这个函数编写代码，需要特别注意它上面的注释标题的警告：首先同步磁盘，否则磁盘缓冲区中的数据可能会丢失。

由于它可能为系统引发的潜在后果，**sys_reboot** 需要几个特殊参数，这一点马上就会讨论。

29305：如果调用者不具有 **CAP_SYS_BOOT**（14096 行）权能（capability），系统就会返回 **EPERM** 错误。权能在第 7 章中会详细讨论。现在，简单的说就是：权能是检测用户是否具有特定权限的方法。

29309：在这里，这种偏执的思想充分发挥了作用。**syst_reboot** 根据从 16002 到 16005 行定义的特殊数字检测参数 **magic1** 和 **magic2**。这种思想是如果 **sys_reboot** 在某种程度上是被偶然调用的，那么就不太可能再从由 **magic1** 和 **magic2** 组成的小集合中同时提取值。注意这并不意味着这是一个防止粗心的安全措施。

顺便说一下，这些特殊数字并不是随机选取的。第一个参数的关系是十分明显的，它是“ 感受死亡（feel dead）”的双关语。后面的三个参数要用十六进制才能了解它们全部的意思：它们分别是 0x28121969，0x5121996，0x16041998。这似乎代表 Linus 的妻子（或者就是 Linus 自己）和他两个女儿的生日。由此推论，当 Linus 和他的妻子养育了更多儿女的时候，重启动需要的特殊参数可能在某种程度上会增加。不过我想在他们用尽 32 位可能空间之前，他的妻子就会制止他的行为了。

29315：请求内核锁，这样能保证这段代码在某一时间只能由一个处理器执行。使用 **lock_kernel/unlock_kernel** 函数对所保护起来的任何其它代码对其它 CPU 都同样是不可访问的。在单处理器的机器中，这只是一个 no-op（不处理任何事情）；而详细讨论它在多处理器上的作用则是第 10 章的内容。

29317：在 **LINUX_REBOOT_CMD_RESTART** 的情况中，**sys_reboot** 调用一系列基于 **reboot_notifier_list** 的函数来通知它们系统正在重新启动。正常情况下，这些函数都是操作系统关闭时需要清除的模块的一部分。这个列表函数似乎并不在内核中的其它地方使用——至少在标准内核发行版本中是这样，也许此外的其它模块可能使用这个列表。不管怎样，这个列表的存在可以方便其他人使用。

LINUX_REBOOT_CMD_RESTART 和其它 **cmd** 识别出的值从 16023 行开始通过 **#define** 进行宏定义。这些值并没有潜在的意义，选用它们的简单原因是它们一般不会发生意外并且相互之间各不相同。（有趣的是，**LINUX_REBOOT_CMD_OFF** 是零，这是在意外情况下最不可能出现的一个值。但是，由于 **LINUX_REBOOT_CMD_OFF** 简单的禁止用户使用 Ctrl+Alt+Del 重新启动机器，它就是一种“ 安全 ”的意外了。）

29321：打印警告信息以后，**sys_reboot** 调用 **machine_restart**（2185 行）重启机器。正如你从 2298 行中所看到的一样，**machine_restart** 函数从来不会返回。但是不管怎样，对于 **machine_restart** 的调用后面都跟着一个 **break** 语句。

这仅仅是经典的良好的编程风格吗？的确如此，但是却又不仅仅如此。文件

kernel/sys.c 的代码是属于体系结构无关部分的。但是 **machine_restart**，它显然是体系结构所特有的，属于代码的体系结构特有的部分（arch/i386/kernel/process.c）。因而对于不同的移植版本也有所不同。我们并不清楚以后内核的每个移植版本的实现都不会返回——例如，它可能调度底层硬件重启但是本身要仍然持续运行几分钟，这就需要首先从函数中返回。或者更为确切的说法是，由于某些特定的原因，系统可能并不总是能够重启；或许某些软件所控制的硬件根本就不能重启。在这种平台上，**machine_restart** 就应该可以返回，因此体系结构无关的代码应该对这种可能性有所准备。

针对这个问题，正式的发行版本中都至少包含一个退出端口，使 **machine_restart** 函数可以从这个端口返回：m68k 端口。不同的基于 m68k 的机器支持的代码也各不相同，由于本书主要是针对 x86 的，我不希望花费过多的时间来解析所有的细节。但是这的确是可能的。（在其它情况下，**machine_restart** 简单进入一个无限循环——既不重新启动机器，也不返回。但是这里我们担心的是需要返回的情况。）

因此，我们毕竟是需要 **break** 的。前面看起来只是简单的习惯甚至是偏执的思想在这里为了内核的移植性已经变成必须的了。

29324：接下来的两种情况分别允许和禁止臭名卓著的 Ctrl+Alt+Del 组合键（这三个组合键也被称为“ Vulcan 神经收缩（Vulcan nerve pinch）”，“ 黑客之手（hacker’s claw）”，“ 三指之礼（three-fingered salute）”，我最喜欢后面这个）。这些只是简单的设置全局 **C_A_D** 标志（在 29160 行定义，在 29378 行检测）。

29332：这种情况和 **LINUX_REBOOT_CMD_RESTART** 类似，但只是暂停系统而不是将其重新启动。两者之间的一个区别是它调用 **machine_halt**（2304 行）——这是 x86 上的一条 no-op 指令，但是在其它平台上却要完成关闭系统的实际工作——而不是调用 **machine_restart**。并且它会把 **machine_halt** 不能使之暂停的机器转入低功耗模式运行。它使用 **do_exit**（23267 行）杀死内核本身。

29340：到现在为止，这已经是一种比较熟悉的模式了。这里，**sys_reboot** 关闭机器电源，除了为可以使用软件自行关闭电源的系统调用 **machine_power_off**（2307 行）之外，其它的应该和暂停机器情况完全相同。

29348：**LINUX_REBOOT_CMD_RESTART**2 的情况是已建立主题的一个变种。它接收命令，将其作为 ASCII 字符串传递，该字符串说明了机器应该如何关闭。字符串不会由 **sys_reboot** 本身来解释，而是使用 **machine_restart** 函数来解释；因而这种模式的意义，如果有的话，就是这些代码是平台相关的。（我使用“ 如果有 ”的原因是启动机器——特别是在 x86 中——一般只有一种方法，因此其它的信息都可以被 **machine_restart** 忽略。）

29365：调用者传递了一个无法识别的命令。**sys_reboot** 不作任何处理，仅仅返回一个错误。因此，即使由 **magic1** 和 **magic2** 传递给 **sys_reboot** 正确的 magic 数值，它也无须处理任何内容。

29369：一个可识别的命令被传递给 **sys_reboot**。如果流程执行到这里，它可能就是两个设置 **C_A_D** 的命令之一，因为其它情况通常都是停止或者重新启动机器。在任何情况下，**sys_reboot** 都简单把内核解锁并返回 0 以表示成功。

sys_sysinfo

24142：一个只能返回一个整型值的系统调用。如果需要返回更多的信息，我们只需要使用类似于在系统调用中传递多于四个参数时所使用的技巧就可以了：我们通过一个指向结构的指针将结果返回。收集系统资源使用情况的 **sysinfo** 系统调用就是这种函数的一个样例。

24144：分配并清空一个 **struct sysinfo** 结构（15004 行）以暂时存储返回值。**sys_sysinfo** 可以把结构中的每个域都独立地拷贝出来，但是这样会速度很慢、很不方便，而且必然不容易阅读。

24148：禁止中断。这在第 6 章中会有详细的介绍；作为目前来说，我们只要说明这种模式在使用的过程中能够确保 **sys_sysinfo** 正在使用的值不会改变就足够了。

24149：**struct sysinfo** 结构的 **uptime** 域用来指明系统已经启动并运行了的秒数。这个值是使用 **jiffies**（26146 行）和 **HZ** 来计算的。**jiffies** 计算了系统运行过程中时钟的滴答次数；**HZ** 是系统相关的一个参数，它十分简单，就是每秒内部时钟滴答的次数。

24151：数组 **avenrun**（27116 行）记录了运行队列的平均长度——也就是等待 CPU 的平均进程数——在最后的 1 秒钟，5 秒钟和 15 秒钟。**calc_load**（27135 行）周期性的重复计算它的值。由于内核中是要严格禁止浮点数运算的，所以只能通过计算变化的次数这一修正值来计算。

24155：同样记录系统中当前运行的进程数。

24158：**si_meminfo**（07635 行）写入这个结构中的内存相关成员，**si_swapinfo**（38544 行）写入与虚拟内存相关的部分。

24161：现在整个结构都已经全部填充了。**sysinfo** 试图将其拷贝回用户空间，如果失败就返回 **EFAULT**，如果成功就返回 0。