Linux 系统调用权威指南

2016 年 4 月 5 日, • packagecloud （于朝阳 译）

标签:

- linux
- kernel
- syscall

简介

这篇博文描述了 Linux 程序如何调用内核方法。

文中概述了几种不同的执行系统调用的方法，同时举例讲述了如何通过写汇编代码来执行系统调用，以及系统调用的内核入口/出口，glibc 封装，代码错误，和其它相关信息。

什么是系统调用？

当一个应用程序调用如 `open`，`fork`，`read`，`write`（或者很多其它类似方法）时，就会用到系统调用。

系统调用就是应用程序进入内核中去完成某些任务的途径。应用程序通过系统调用来完成一系列操作，比如：创过新进程，执行网络/文件 IO，等等。

在这个网页 man page for syscalls(2)里，可以找到所有系统调用的列表。应用程序可以用几种不同的方法来完成系统调用，而具体的机器码则与 CPU 体系结构相关。

从应用程序开发人员的角度来看，我们大可不必关心系统调用是如何被执行的。因为我们只需要引用合适的头文件，然后就像使用普通函数那样去调用库函数就可以了。

`glibc` 提供的封装代码可以让调用者不必去关心底层实现，这些封装代码会使用调用时传入的参数来执行内核调用。

在详细解释系统调用如何执行之前，我们需要定义一些后文中出现的术语和观点。

背景知识

硬件和软件

在这篇文章中，我们有以下假设：

- 你用的是一个 32 位或 64 位的 Intel 或 AMD CPU 的系统。文中描述的理论对于使用其它系统的人来说是基本相同，但是具体的代码都是以 32 位或 64 位 Intel/AMD CPU 为例的。
- 你对 Linux 内核版本 3.13.0 感兴趣。其它版本的代码可能是类似的，但是行号或者代码组织方式，文件路径可能会有所不同。GitHub 上有 3.13.0 版本的内核代码可供参考。
- 你对 `glibc` 或者基于 `glibc` 的其它 libc 实现（比如 `eglibc`）感兴趣。

在这篇文章中，x86-64 指基于 x86 架构的 64 位 Intel/AMD CPU。

应用程序，内核与 CPU 特权级

应用程序（比如文本编辑器，终端程序，ssh 守护程序，等等）需要与 Linux 内核进行交互，内核才能帮助这些应用程序完成某些应用程序自己不能做的一系 等列操作。

比如，一个应用程序需要完成某些 IO 操作（`open`，`read`，`write` 等）或者修改它的地址空间（`mmap`，`sbrk` 等）。应用程序必须触发内核来代表它完成这些操作。

为什么应用程序不能自己做这些操作呢？

这主要是由于 x86-64 CPU 中一个叫做 特权级的概念。特权级是一个复杂的话题，我们可以把它放在另外一篇文章中单独来讲。对于这篇文章而言，我们（很大程度地）简化了特权级的概念，如下：

1. 特权级是一种访问控制的方法。当前的特权级决定了哪些 CPU 指令和 IO 操作允许被执行。
2. 内核运行在最高特权级，称作“Ring 0”特权级。应用程序运行在一个较低的特权级上，通常是“Ring 3”特权级。

为了能够执行某些特权操作，应用程序必须进行特权级转换（从“Ring 3”到“Ring 0”），内核才能完成操作。

有多种方法可以完成一次特权级转换并触发内核执行某些操作。

让我们先来看一种最常见的方式：中断。

中断

你可以认为中断是一个由硬件或软件生成（或者叫“引起”）的事件。

硬件中断由硬件设备发起，它用来通知内核一个特殊事件的发生。一个常见的这类中断的例子就是当网卡接收到一个包时产生的中断。

软件中断是由执行一段代码而触发的。在 x86-64 的系统中，软件中断可以由执行 `int` 指令来触发。

中断通常是有编号的，某些中断号具有特殊的意义。

可以这样想象，在内存中有一个数组，数组中的每一项都保存着一个函数地址，并且与一个中断号相对应。每当接收到某个编号的中断和相关信息时，比如这个中断处理函数所需要的特权级，CPU 就从数组中保存的这个函数地址开始执行。

下面这幅来自 Intel CPU 手册中的图展示了这个数组中每一项的结构：

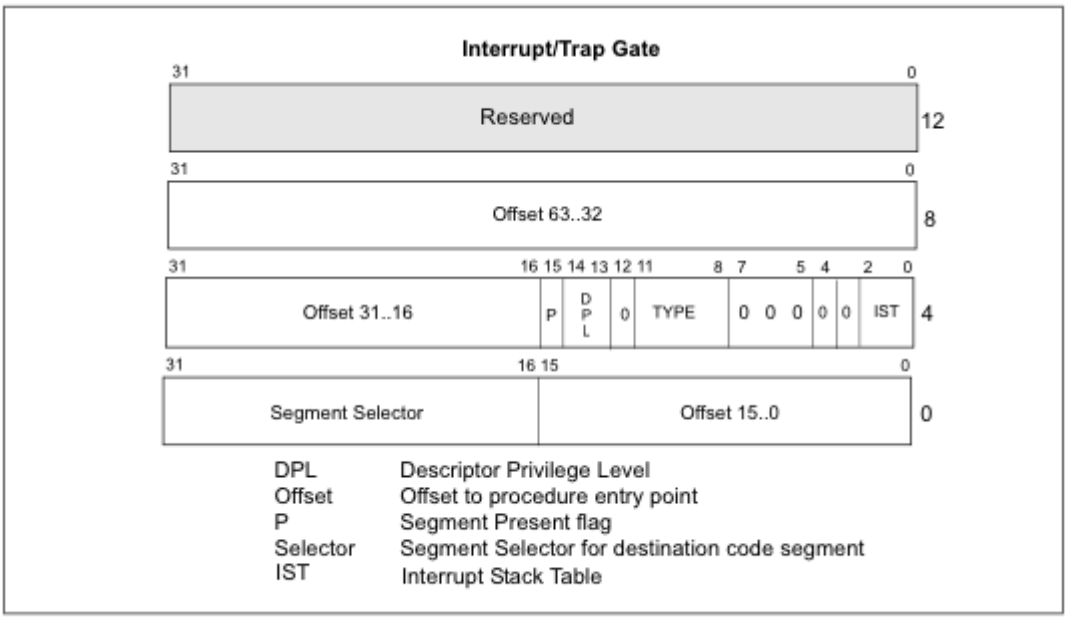


Figure 5-7. 64-Bit IDT Gate Descriptors

仔细观察这张图，你会发现有一个标记为 DPL（描述符特权级）的 2-bit 位域。这个 2-bit 位域的值是当前中断处理函数被允许执行的最低特权级。

当某一个事件发生时，CPU 就是通过这种方式获取到中断处理函数的地址和执行这个中断处理函数所需要的特权级的。

实际上，x86-64 系统有多种不同的方式来处理中断。如果你有兴趣可以阅读下面这些内容 8259 Programmable Interrupt Controller，Advanced Interrupt Controllers 和 IO Advanced Interrupt Controllers。

在处理硬件中断和软件中断时还有很多复杂的问题，比如中断号冲突和重映射。在这篇文章中，我们暂时不考虑这些问题。

特殊模块寄存器（MSRs）

特殊模块寄存器（也称为 MSRs）是一些控制 CPU 某些特定功能的寄存器。CPU 的文档列出了每一个 MSR 的地址。

我们可以用 CPU 指令 `rdmsr` 和 `wrmsr` 来分别读/写 MSRs。

还有一些命令行工具可以用来读/写 MSRs，但是我们不建议修改它们。因为这样做（特别是当系统正在运行时）是非常危险的，除非你做的时候非常非常小心谨慎。

当然，如果你不在意这样可能会让你的系统变得不稳定或者造成数据丢失，你可以安装 `msr-tools` 并加载 `msr` 内核模块来读/写 MSRs。

```
% sudo apt-get install msr-tools
% sudo modprobe msr
% sudo rdmsr
```

我们稍后也会讲到，某些系统调用方法用到了 MSRs。

通过汇编语言来执行系统调用，这不是个好主意

通过自己编写汇编代码来执行系统调用不是个好办法。

一个最主要的原因是某些系统调用在 glibc 中执行前/后需要运行一些额外的代码。

在下面这个例子中，我们将使用 `exit` 系统调用。通过 `atexit` 我们可以注册某些函数，这些函数会在 `exit` 被调用时执行。

这些函数都是在 glibc 中被调用的，而不是在内核中被调用的。所以，如果你编写了自己的汇编代码去调用 `exit`，那些注册的处理函数就不会被执行，因为我们绕过了 glibc。

但是，动手编写汇编代码来完成系统调用仍不失为一个好的学习手段。

传统系统调用

在背景知识介绍中我们讲到了两点：

1. 我们可以通过产生一个软件中断来触发内核执行。

2. 我们可以通过 `int` 汇编指令来产生一个软件中断。

结合这两点，我们可以得到 Linux 上传统的系统调用接口。

Linux 内核预留了一个特殊的软件中断号，应用程序可以使用这个中断来进入内核来执行系统调用。

Linux 内核对中断号 128（0x80）注册了一个中断处理函数 `ia32_syscall` 。让我们来看看这段代码。在内核 3.13.0 源代码 `arch/x86/kernel/traps.c` 中的 `trap_init` 函数中：

```
void __init trap_init(void)
{
        /* ..... other code ... */

        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
```

在头文件 `arch/x86/include/asm/irq_vectors.h` 中，宏 `IA32_SYSCALL_VECTOR` 定义为 `0x80`。

问题来了，如果内核只给应用程序预留了一个软件中断号，内核如何知道它应该执行哪个系统调用呢？

答案是，应用程序会把系统调用号放在 `eax` 寄存器中。系统调用的参数会放在其它的通用寄存器当中。

这种用法被记录在源文件 `arch/x86/ia32/ia32entry.S` 中的一段注释中：

```
 * Emulated IA32 system calls via int 0x80.
 *
 * Arguments:
 * %eax System call number.
 * %ebx Arg1
 * %ecx Arg2
 * %edx Arg3
 * %esi Arg4
 * %edi Arg5
 * %ebp Arg6    [note: not saved in the stack frame, should not be
touched]
 *
```

现在我们知道了如何执行一个系统调用，以及参数保存的位置。让我们来试着写一段汇编代码来完成系统调用。

自己写汇编代码来执行系统调用

我们可以通过编写一段汇编代码来执行一个系统调用。尽管这是一个有趣的学习过程，但是仍要说明我们不鼓励读者自己编写汇编代码去执行系统调用。

在这个例子中，我们试着去调用 `exit` 系统调用，它只有一个参数：退出状态。

首先，我们需要找到 `exit` 的系统调用号。Linux 内核有一个文件列出了所有的系统调用。这个文件在编译时被多个脚本用来生成应用程序可用的头文件。让我们来看看 `arch/x86/syscalls/syscall_32.tbl` 文件中的系统调用列表。

```
1 i386 exit sys_exit
```

可以看到，`exit` 系统调用的编号是 1。根据上面描述的接口，我们只需要把系统调用号放到 `eax` 寄存器中，并且把第一个参数（退出状态）放入 `ebx`。

下面的 C 代码和一段内联的汇编代码就完成了这些操作。我们把退出状态设为“42”：

（这个例子其实还可以更简化，但是我想把它写得可读性更强一些，这样既使是不了解 GCC 内联汇编的读者也可以看懂它，或者把它作为一个示例或参考。）

```
int
main(int argc, char *argv[])
{
  unsigned int syscall_nr = 1;
  int exit_status = 42;

  asm ("movl %0, %%eax\n"
       "movl %1, %%ebx\n"
       "int $0x80"
    : /* output parameters, we aren't outputting anything, no none
 */
      /* (none) */
    : /* input parameters mapped to %0 and %1, repsectively */
      "m" (syscall_nr), "m" (exit_status)
    : /* registers that we are "clobbering", unneeded since we are
 calling exit */
      "eax", "ebx");
}
```

接下来编译，执行，检查退出状态：

```
$ gcc -o test test.c
$ ./test
```

```
$ echo $?
42
```

成功啦！我们通过产生一个软件中断的方式完成了 `exit` 系统调用。

内核部分：`int $0x80` 调用入口

到目前为止，我们已经看到了怎样在应用程序中实现一个系统调用。下面让我们来看内核怎样使用系统调用号来执行系统调用的代码。

还记得上一节中内核注册了一个系统调用处理函数 `ia32_syscall`。这个函数在源文件 `arch/x86/ia32/ia32entry.S` 中是用汇编代码实现的。我们可以看到这个函数做了很多操作，但其中最重要的一步就是调用真正的系统调用函数：

```
ia32_do_call:
        IA32_ARG_FIXUP
        call *ia32_sys_call_table(,%rax,8) # xxx: rip relative
```

`IA32_ARG_FIXUP` 是一个宏，它重新组织了传入的参数，这样系统调用层就能正确的读取和使用这些参数。

`ia32_sys_call_table` 指针指向中断处理向量表，这张表定义在源文件 `arch/x86/ia32/syscall_ia32.c` 中。请注意这段代码末尾的`#include` 那行：

```
const sys_call_ptr_t ia32_sys_call_table[__NR_ia32_syscall_max+1]
= {
        /*
         * Smells like a compiler bug -- it doesn't work
         * when the & below is removed.
         */
        [0 ... __NR_ia32_syscall_max] = &compat_ni_syscall,
#include <asm/syscalls_32.h>
};
```

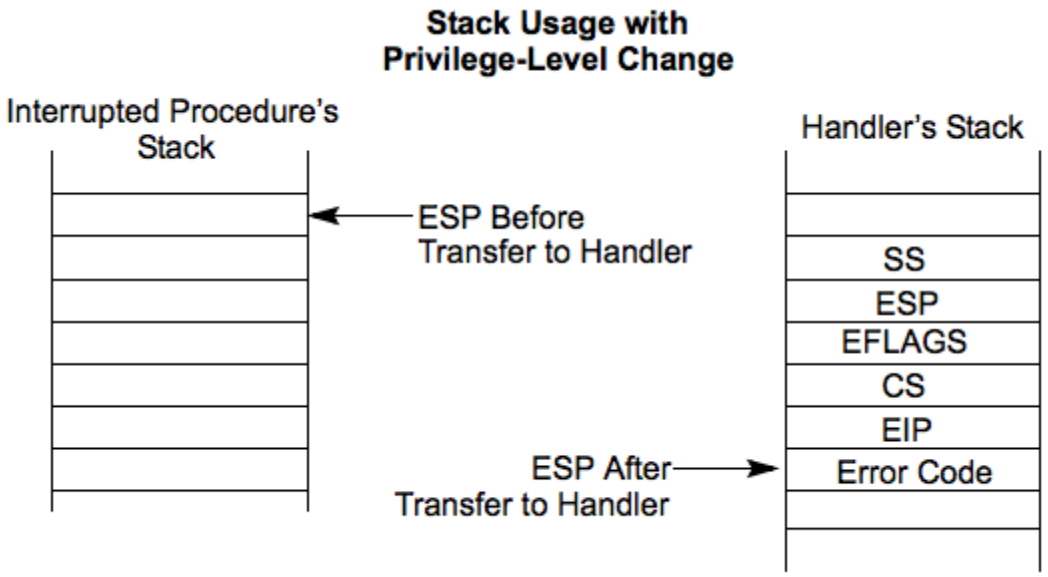
之前我们讲过，这个系统调用表的实际内容定义在源文件 `arch/x86/syscalls/syscall_32.tbl` 中，而且还有一些脚本会在编译时用这个系统调用表来生成头文件 `syscalls_32.h`。这个头文件实际上就是一些 C 代码，它只是简单地将所有的中断处理函数地址和对应的系统调用号复制到 `ia32_sys_call_table` 数组中。

这就是在执行系统调用的时候，从中断进入内核时所做的事情。

从系统调用中返回 `iret`

我们现在已经了解了怎样通过软件中断进入内核，接下来，内核在执行完成后是怎样返回到应用程序并且退回低特权级的呢？

在 Intel 软件开发人员手册（注意：这是一个非常大的 PDF 文件）中，我们可以找到一幅图，它很好地解释了当发生特权级转换时，内存中的调用栈是如何组织的。让我们来看看下面这幅图：



当应用程序触发了软件中断时，系统执行从用户代码转移到了内核函数 `ia32_syscall`，这时就发生了特权级转换。当执行进入到 `ia32_syscall` 函数中时，我们可以看到调用栈就变成了上图右边的样子。

这就是说，在 `ia32_syscall` 函数真正执行之前，CPU 的相关信息，包括特权级，返回地址和其它所有的状态都被保存在调用栈里了。

所以，为了返回应用程序继续执行，内核只需要简单地把保存的这些信息从栈里再复制到对应的寄存器里，这样应用程序就可以继续执行了。

下面我们来看具体是怎么执行的。

同样我们也有很多种方法来完成这个过程，但是有一个比较简便的办法就是使用 `iret` 指令。

在 Intel 的指令集手册里是这样解释 `iret` 指令的。`iret` 指令从栈里取出返回地址和其它寄存器值，取出的顺序跟保存时压栈的顺序相同：

> 从实地址模式的中断中返回时，IRET 指令从栈中弹出返回指令指针，代码段选择子和 EFLAGS，分别放到 EIP、CS、和 EFLAGS 寄存器中，然后系统从应用程序或过程的中断处继续执行。

想在 Linux 内核源代码中找到这段代码还是有点困难的，因为它被隐藏在几个宏定义之下，而且里面有很多处理信号和 ptrace 系统调用退出追踪的代码。

在内核相关的宏展开之后，我们就可以看到 `iret`，它会使代码执行从系统调用中返回到应用程序中去。

在源文件 `arch/x86/kernel/entry_64.S` 中的 `irq_return` 行：

```
irq_return:
  INTERRUPT_RETURN
```

宏 `INTERRUPT_RETURN` 在源文件 `arch/x86/include/asm/irqflags.h` 中被定义为 `iretq`。

好了，到这里为止，你就已经知道了系统调用是如何工作的了。

快速系统调用

上面讲的系统调用方法似乎已经足够好了，但是我们还有些新方法来执行一个系统调用。这些新方法不必采用软件中断，因此他们比原先的中断方式执行起来更快。

这两种快速系统调用方法，每一种都有两条指令。一条指令进入内核，另一条指令离开内核。这两种方法在 Intel CPU 文档中都被描述为“快速系统调用”。

不幸的是，Intel 和 AMD 在实现上有一点没有达成一致，即 CPU 处于 32 位或 64 位模式时，到底哪种方法是可用的。

为了在 Intel 和 AMD 处理器上达到最大的兼容性：

- 32 位系统上使用：`sysenter` 和 `sysexit`。
- 64 位系统上使用：`syscall` 和 `sysret`。

32 位快速系统调用

sysenter/sysexit

用 sysenter 来完成一次系统调用比原先的中断方式更复杂一些，而且需要应用程序（通过 glibc 库）和内核之间有更多的协作。

让我们一步一步来详细讲解。首先，来看看 Intel 指令集手册（注意，又是一个非常大的 PDF 文件）里是怎么描述 sysenter 指令以及它的用法的：

> 在执行 SYSENTER 指令之前，应用程序必须将特权级 0 的代码段选择子和代码入口指针，以及特权级 0 的堆栈段选择子和栈指针放到下面指定的 MSRs 寄存器中：
>
> • IA32_SYSENTER_CS（MSR 地址 174H）——这个寄存器的低 16 位用来保存特权级 0 代码段的段选择子。这个值也决定了特权级 0 的堆栈段选择子（参考“操作”一节）。这个值不能为空值 NULL。
>
> • IA32_SYSENTER_EIP（MSR 地址 176H）——这个寄存器中的值将会被加载到 RIP 中（因此，这个值指向了处理函数的首地址）。在保护模式下，只有 0~31 位会被使用。
>
> • IA32_SYSENTER_ESP（MSR 地址 175H）——这个寄存器中的值将会被加载到 RSP 中（因此，这个值是特权级 0 的栈指针）。这个值必须指向一个合法的地址空间。在保护模式下，只有 0~31 位会被使用。

换句话说，为了使内核能够接收处理 sysenter 系统调用，内核必须设置这三个特殊模块寄存器 MSRs。在这三个寄存器中，我们最感兴趣的是 IA32_SYSENTER_EIP（地址 0x176）。当应用程序执行 sysenter 指令时，这个寄存器中存放的就是内核将要执行的处理函数的首地址。

我们可以在 Linux 内核源文件 arch/x86/vdso/vdso32-setup.c 中找到写 MSR 的代码：

```
void enable_sep_cpu(void)
{
        /* ... other code ... */

        wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long) ia32_sysenter
_target, 0);
```

其中，宏 MSR_IA32_SYSENTER_EIP 在源文件 arch/x86/include/uapi/asm/msr-index.h 中定义为 0x00000176。

与软件中断方式的系统调用类似，用 `sysenter` 来完成系统调用时也有一些约定。

在源代码 `arch/x86/ia32/ia32entry.S` 中有一处注释说明了它的用法：

```
* 32bit SYSENTER instruction entry.
 *
 * Arguments:
 * %eax System call number.
 * %ebx Arg1
 * %ecx Arg2
 * %edx Arg3
 * %esi Arg4
 * %edi Arg5
 * %ebp user stack
 * 0(%ebp) Arg6
```

之前我们讲的中断方式的系统调用，有一个返回被中断的应用程序的机制：`iret` 指令。

若采用类似的方式来处理 `sysenter` 会比较复杂，因为与软件中断不同的是，`sysenter` 并不保存返回地址。

确切地说，内核在执行 `sysenter` 指令之前所做的事情并非一成不变（确实会有变化，在后面的“错误”一节我们会讲到）。

为了保证这些变化不会产生影响，应用程序需要调用一个函数`__kernel_vsyscall`。尽管这个函数是在内核里实现的，但是它在用户进程启动的时候被映射到用户进程的地址空间里。

这听起来好像稍微有点奇怪，这段代码是内核代码，但却运行在用户模式下。

实际上，`__kernel_vsyscall` 是一个叫做虚拟动态共享对象（vDSO）的一部分。这个 vDSO 主要作用就是允许应用程序能够在用户模式下执行内核代码。

在这篇文章的后面，我们会深入解释 vDSO 是什么，它做了些什么，以及是如何完成的。

现在，让我们先来看看`__kernel_vsyscall` 的内部实现。

`__kernel_vsyscall` 解析

`__kernel_vsyscall` 函数封装了 `sysenter` 调用，源代码在 `arch/x86/vdso/vdso32/sysenter.S` 中：

```
__kernel_vsyscall:
.LSTART_vsyscall:
        push %ecx
.Lpush_ecx:
        push %edx
.Lpush_edx:
        push %ebp
.Lenter_kernel:
        movl %esp,%ebp
        sysenter
```

`__kernel_vsyscall` 是动态共享对象（也称为共享库）的一部分。应用程序如何在运行时找到这个函数地址呢？

`__kernel_vsyscall` 函数的地址保存在一个 ELF 辅助向量中，因此应用程序或者其它库（通常是 `glibc`）就能找到，并且调用它。

查询 ELF 辅助向量有几种方法：

1. 调用 `getauxval` 时，使用 `AT_SYSINFO` 参数。
2. 遍历环境变量，从内存中解析出来。

第一种方法是最简单的，但是在 `glibc` 2.16 版本之前无法使用它。下面的代码给出了第二种方法的示例。

之前我们讲过，`__kernel_vsyscall` 会在执行 `sysenter` 之前做一些记录操作。

所以，我们要通过 `sysenter` 的方式进入内核，只需要做以下操作：

- 在 ELF 辅助向量里通过 `AT_SYSINFO` 来查找`__kernel_vsyscall` 的地址。
- 与中断方式系统调用类似，我们需要将系统调用号和参数放到相应的寄存器当中。
- 调用 `__kernel_vsyscall` 函数。

你之前肯定没有写过你自己的 `sysenter` 封装函数，因为内核进入和退出系统调用 `sysenter` 的方法不是一成不变的，所以你的代码很可能会失效。

总之，若要使用 `sysenter` 系统调用，必须通过`__kernel_vsyscall` 函数。

让我们来试试。

自己写汇编代码来执行 `sysenter` 系统调用

与我们之前的例子一样，我们退出时调用 `exit`，并且置退出状态为 42。

`exit` 系统调用号是 1。根据之前对接口的描述，我们只需要把系统调用号放入到 `eax` 寄存器中，并且把第一个参数（退出状态）放到 `ebx` 中就可以了。

（同样，这个例子还可以更简化一些，但是我想把它写得可读性更强，这样既使是不了解 GCC 内联汇编的读者也可以看懂它，或者把它作为一个示例或参考。）

```
#include <stdlib.h>
#include <elf.h>

int
main(int argc, char* argv[], char* envp[])
{
  unsigned int syscall_nr = 1;
  int exit_status = 42;
  Elf32_auxv_t *auxv;

  /* auxilliary vectors are located after the end of the environme
nt
   * variables
   *
   * check this helpful diagram: https://static.lwn.net/images/201
2/auxvec.png
   */
  while(*envp++ != NULL);

  /* envp is now pointed at the auxilliary vectors, since we've it
erated
   * through the environment variables.
   */
  for (auxv = (Elf32_auxv_t *)envp; auxv->a_type != AT_NULL; auxv+
+)
  {
    if( auxv->a_type == AT_SYSINFO) {
      break;
    }
  }

  /* NOTE: in glibc 2.16 and higher you can replace the above code
 with
   * a call to getauxval(3):  getauxval(AT_SYSINFO)
   */

  asm(
      "movl %0,  %%eax    \n"
      "movl %1, %%ebx    \n"
      "call *%2          \n"
```

```
        : /* output parameters, we aren't outputting anything, no no
ne */
          /* (none) */
        : /* input parameters mapped to %0 and %1, repsectively */
          "m" (syscall_nr), "m" (exit_status), "m" (auxv->a_un.a_va
l)
        : /* registers that we are "clobbering", unneeded since we a
re calling exit */
          "eax", "ebx");
}
```

接下来编译，执行，查看退出状态：

```
$ gcc -m32 -o test test.c
$ ./test
$ echo $?
42
```

成功！我们可以不使用软件中断的方式，而是采用 sysenter 方法调用了 `exit` 系统调用。

内核部分：`sysenter` 调用入口

现在，我们已经知道了应用程序怎样通过`__kernel_vsyscall` 和 `sysenter` 来触发系统调用。下面让我们来看看内核是怎样使用系统调用号来执行系统调用代码的。

之前我们讲到过，内核注册了一个系统调用处理函数 `ia32_sysenter_target` 函数。

这个函数在源文件 `arch/x86/ia32/ia32entry.S` 中，以汇编代码实现的。让我们来看一下放在 eax 寄存器中的值是在哪里被用来执行系统调用的：

```
sysenter_dispatch:
        call    *ia32_sys_call_table(,%rax,8)
```

这一步的代码与中断方式执行系统调用的代码相同：以系统调用号为顺序的 `ia32_sys_call_table` 中断向量表。

完成这些操作之后，中断方式和 `sysenter` 方式的系统调用都采用相同的机制和系统调用表来完成后续的操作。

请参考 `int $0x80` 调用入口这一节的内容，它讲了中断向量表 `ia32_sys_call_table` 是怎么定义和生成的。

以上内容，就是通过 `sysenter` 系统调用来进入内核的方法。

`sysexit` 从系统调用 `sysenter` 中返回

内核可以通过 `sysexit` 指令返回被中断的应用程序中继续执行。

使用这条指令时不像用 `iret` 时那样直接，调用者需要将返回地址放到 `rdx` 寄存器中，而且需要将返回后的程序栈指针放到 `rcx` 寄存器中。

这意味着应用程序必须计算好返回地址并保存起来，以便在调用 `sysexit` 之前能够把它原样恢复。

我们可以在源文件 `arch/x86/ia32/ia32entry.S` 中找到这段代码：

```
sysexit_from_sys_call:
        andl    $~TS_COMPAT,TI_status+THREAD_INFO(%rsp,RIP-ARGOFFS
ET)
        /* clear IF, that popfq doesn't enable interrupts early */
        andl  $~0x200,EFLAGS-R11(%rsp)
        movl    RIP-R11(%rsp),%edx              /* User %eip */
        CFI_REGISTER rip,rdx
        RESTORE_ARGS 0,24,0,0,0,0
        xorq    %r8,%r8
        xorq    %r9,%r9
        xorq    %r10,%r10
        xorq    %r11,%r11
        popfq_cfi
        /*CFI_RESTORE rflags*/
        popq_cfi %rcx                           /* User %esp */
        CFI_REGISTER rsp,rcx
        TRACE_IRQS_ON
        ENABLE_INTERRUPTS_SYSEXIT32
```

其中宏 `ENABLE_INTERRUPTS_SYSEXIT32` 定义在源文件 `arch/x86/include/asm/irqflags.h` 中，它包含了 `sysexit` 指令。

到此为止，我们已经讲完了 32 位快速系统调用是如何工作的。

64 位快速系统调用

接下来我们要讲得是 64 位快速系统调用。在这部分，我们将使用 `syscall` 和 `sysret` 来分别完成进入系统调用以及从系统调用中返回。

`syscall`/`sysret`

在 Intel 指令集手册（非常大的 PDF）中是这样描述 `syscall` 的工作方式的：

> SYSCALL 会在特权级 0 上调用一个操作系统的系统调用处理函数。它会将 IA32_LSTAR MSR 寄存器中的值加载到 RIP 寄存器中（在将下一条指令地址保存在 RCX 寄存器之后）。

换句话说，当系统调用发生时，为了让内核能够接收处理，必须将待执行的代码地址放到 `IA32_LSTAR` MSR 寄存器中。

我们可以在内核源文件 `arch/x86/kernel/cpu/common.c` 中找到这段代码：

```
void syscall_init(void)
{
        /* ... other code ... */
        wrmsrl(MSR_LSTAR, system_call);
```

其中 `MSR_LSTAR` 的值为 `0xc0000082`，定义在源文件 `arch/x86/include/uapi/asm/msr-index.h` 中。

与软件中断方式的系统调用类似，通过 `syscall` 执行系统调用时，也有一些约定。

应用程序需要将系统调用号放到 `rax` 寄存器中，其它的参数需要放到一系列通用寄存器中。

这些都在 x86-64 ABI 文档的 A.2.1 节中有定义：

1. 应用程序使用整型寄存器来传递参数序列%rdi，%rsi，%rdx，%rcx，%r8 and %r9。内核接口使用 %rdi，%rsi，%rdx，%r10，%r8 and %r9。
2. 系统调用通过 syscall 指令来完成。内核将会使用覆盖%rcx 和%r11 两个寄存器。
3. 系统调用号必须通过%rax 寄存器来传递。
4. 系统调用的参数最多 6 个，不能通过栈空间来传递参数。
5. 从系统调用中返回时，寄存器%rax 中保存的是系统调用的返回值。如果返回值在-4095 到-1 这个区间，说明系统调用出错，错误码为返回值再取负。
6. 只有 INTEGER 和 MEMORY 类型的值可以传递到内核。

这些内容在源文件 `arch/x86/kernel/entry_64.S` 中的一段注释中也有记录。

现在我们已经知道了如何执行系统调用以及参数存放的地方，下面让我们来试着自己写一段内嵌汇编代码来完成一次系统调用。

自己写汇编代码来执行 `syscall` 系统调用

在前面那个示例的基础上，让我们来写一段 C 程序和内嵌的汇编代码来完成退出系统调用并返回退出状态 42。

首先，我们需要找到 `exit` 对应的系统调用号。这里，我们需要查看定义在源文件 `arch/x86/syscalls/syscall_64.tbl` 中的一张表：

```
60 common exit sys_exit
```

可以看到，`exit` 的系统调用号为 `60`。根据之前描述的接口，我们只需要把 `60` 这个值放到 `rax` 寄存器中，并且把第一个参数（退出状态）放到 `rdi` 寄存器中。

下面这段代码就是内嵌了汇编代码的一段 C 程序。与之前的例子一样，这段示例代码为了可读性，写得稍微复杂了些，并不是最简单的形式：

```
int
main(int argc, char *argv[])
{
  unsigned long syscall_nr = 60;
  long exit_status = 42;

  asm ("movq %0, %%rax\n"
       "movq %1, %%rdi\n"
       "syscall"
    : /* output parameters, we aren't outputting anything, no none
 */
      /* (none) */
    : /* input parameters mapped to %0 and %1, repsectively */
      "m" (syscall_nr), "m" (exit_status)
    : /* registers that we are "clobbering", unneeded since we are
 calling exit */
      "rax", "rdi");
}
```

接下来编译，执行，查看退出状态：

```
$ gcc -o test test.c
$ ./test
$ echo $?
42
```

又成功啦！我们通过 `syscall` 完成了调用 `exit` 系统调用的过程。这个过程中，我们没有使用软件中断的方式，因此它执行得更快（如果我们使用一个精确时钟就可以观察到）。

内核部分：`syscall` 调用入口

现在，我们已经知道了怎样在应用程序中触发一次系统调用。下面让我们来看看内核是如何用系统调用号来执行系统调用的代码的。

之前我们讲到，我们把 `system_call` 函数的地址放到了 `LSTAR` MSR 寄存器中。

让我们来看看这个函数的代码，看它如何使用 `rax` 寄存器将实际执行权转交给系统调用。在源文件 `arch/x86/kernel/entry_64.S` 中：

```
call *sys_call_table(,%rax,8)  # XXX:    rip relative
```

与中断方式系统调用类似，`sys_call_table` 表定义在一个 C 文件中，用 `#include` 把脚本生成的 C 代码定义在这个表中。

在源文件 `arch/x86/kernel/syscall_64.c` 中，注意最下面的`#include` 那一行：

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1]
 = {
        /*
         * Smells like a compiler bug -- it doesn't work
         * when the & below is removed.
         */
        [0 ... __NR_syscall_max] = &sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

之前，我们看到系统调用表定义在源文件 `arch/x86/syscalls/syscall_64.tbl` 中。与中断方式类似，一个脚本在内核编译时，根据 `syscall_64.tbl` 中定义的系统调用表来生成 `syscalls_64.h` 头文件。

上面的代码只是简单地引入了以系统调用号为顺序的函数指针数组。

以上就是如何通过 `syscall` 系统调用的方式进入内核。

`sysret` 从系统调用 `syscall` 中返回

内核可以用 `sysret` 指令来返回到应用程序执行 `syscall` 时进入系统调用的地方，并继续执行。

`sysret` 比 `sysexit` 更简单一些，因为返回地址已经在调用 `syscall` 时保存在 `rcx` 寄存器中了。

因此，只需要将这个值保存在某个地方，并在调用 `sysret` 之前，将这个值恢复到 `rcx` 寄存器中就可以了。

这个要求并不复杂，因为 `sysenter` 指令要求应用程序自己计算这个地址，并且需要用到另一个寄存器。

我们可以在源文件 `arch/x86/kernel/entry_64.S` 中找到这段代码：

```
movq RIP-ARGOFFSET(%rsp),%rcx
CFI_REGISTER     rip,rcx
RESTORE_ARGS 1,-ARG_SKIP,0
/*CFI_REGISTER  rflags,r11*/
movq     PER_CPU_VAR(old_rsp), %rsp
USERGS_SYSRET64
```

宏 `USERGS_SYSRET64` 定义在源文件 `arch/x86/include/asm/irqflags.h` 中，其中包含了 `sysret` 指令。

好了，现在我们已经讲完了 64 位快速系统调用是如何工作的。

半手动方式调用 syscall(2)

我们已经知道了几种不同的方法，采用汇编代码完成系统调用。

通常情况下，我们并不需要自己写汇编代码，因为 glibc 提供的封装函数就可以帮助我们完成这些事情。

但是，有一些系统调用，glibc 并没有提供封装函数。其中一个例子就是 `futex`，它是一个快速锁住用户空间的系统调用。

等等，为什么 glibc 对 `futex` 不提供封装函数?

`futex` 通常只会被库调用，而应用程序不会使用这个系统调用。因此，如果要调用 `futex`，我们有下面两个选择：

1. 为每一个需要的平台编写一套汇编代码。
2. 使用 glibc 提供的 `syscall` 封装函数。

如果我们必须调用一个没有封装的系统调用，最好用第二个选择：使用 glibc 的 `syscall` 函数。

下面让我们用 glibc 的 `syscall` 来调用一次退出状态为 42 的 `exit` 系统调用：

```
#include <unistd.h>

int
main(int argc, char *argv[])
{
  unsigned long syscall_nr = 60;
  long exit_status = 42;

  syscall(syscall_nr, exit_status);
}
```

接下来编译，执行，查看退出状态：

```
$ gcc -o test test.c
$ ./test
$ echo $?
42
```

成功！我们通过 glibc 提供的 `syscall` 封装函数又一次完成了 `exit` 系统调用。

glibc `syscall` 封装解析

让我们来看看上一节示例中用到的 `syscall` 封装函数是如何工作的。

在源文件 `sysdeps/unix/sysv/linux/x86_64/syscall.S` 中：

```
/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, ar
g5, arg6)
   We need to do some arg shifting, the syscall_number will be in
   rax.  */


        .text
ENTRY (syscall)
        movq %rdi, %rax         /* Syscall number -> rax.  */
        movq %rsi, %rdi         /* shift arg1 - arg5.  */
        movq %rdx, %rsi
```

```
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                 /* Do the system call.  */
        cmpq $-4095, %rax       /* Check %rax for error.  */
        jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.
  */
L(pseudo_end):
        ret                     /* Return to caller.  */
```

之前我们引用了 x86_64 ABI 文档中描述的用户模式和内核模式的调用约定。

这个汇编清楚地展示了应用程序和内核两部分的调用约定。传入的参数遵从应用程序调用约定，但是紧接着在调用 `syscall` 进入内核之前，这些参数被重新组织到一组遵从内核调用约定的寄存器中。

这就是 glibc 系统调用的封装函数对于没有默认封装的系统调用的处理方式。

虚拟系统调用

我们现在已经讲完了进入内核执行系统调用的所有方法，并且展示了如何通过手动（或者半手动）的方式来执行系统调用，使得系统从用户态迁移动内核态。

有没有可能应用程序根本不需要进入内核就能够执行某些系统调用呢？

这正是 Linux 虚拟动态共享对象（vDSO）存在的原因。Linux vDSO 是内核部分代码中的一段，但是被映射到应用程序的地址空间中，因此可以在用户态下执行。

这件事情的核心意义在于，某些系统调用可以在不进入内核态的情况下被调用。其中一个例子便是：`gettimeofday`。

应用程序调用 `gettimeofday` 系统调用时，并不需要真正进入内核。这时，它只是在用户态下，执行了由内核提供的一小段代码。

这里不需要软件中断，也没有复杂的 `sysenter` 和 `syscall` 调用约定。`gettimeofday` 只是一个普通的函数调用。

当你使用 `ldd` 命令时，vDSO 就显示在结果的第一行：

```
$ ldd `which bash`
  linux-vdso.so.1 =>  (0x00007fff667ff000)
```

```
  libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f62
3df7d000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f623dd790
00)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f623d9ba00
0)
  /lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

让我们来看看 vDSO 是如何在内核中建立起来的。

内核 vDSO

vDSO 的源代码在 `arch/x86/vdso/` 目录中。同时，还有一些汇编和 C 源文件以及一个链接器脚本文件。

我们来看看这个有点酷的链接器脚本文件 `arch/x86/vdso/vdso.lds.S`：

```
/*
 * This controls what userland symbols we export from the vDSO.
 */
VERSION {
        LINUX_2.6 {
        global:
                clock_gettime;
                __vdso_clock_gettime;
                gettimeofday;
                __vdso_gettimeofday;
                getcpu;
                __vdso_getcpu;
                time;
                __vdso_time;
        local: *;
        };
}
```

链接器脚本虽然非常有用，但却鲜为人知。这个链接器脚本将需要在 vDSO 中导出的符号组织在一起。

我们可以看到 vDSO 导出了 4 组不同的函数，每一组有两个名字。在这个目录中的 C 文件中可以找到这些函数的源代码。

例如，函数 `gettimeofday` 的代码定义在 `arch/x86/vdso/vclock_gettime.c` 文件里：

```
int gettimeofday(struct timeval *, struct timezone *)
        __attribute__((weak, alias("__vdso_gettimeofday")));
```

这里 `gettimeofday` 被定义为`__vdso_gettimeofday`的一个弱别名。

在同一个源文件里，函数`__vdso_gettimeofday` 包含了实际的源代码。在应用程序调用 `gettimeofday` 系统调用时，这些代码将在用户态下执行。

在内存中定位 vDSO

因为地址空间随机布局，当应用程序启动时，vDSO 会被加载到一个随机的地址中。

那么应用程序要如何才能找到被加载到随机地址中的 vDSO 呢？

回忆一下我们之前讲的 `sysenter` 系统调用方法，我们知道应用程序需要调用 `__kernel_vsyscall`，而不是执行自己写的 `sysenter` 汇编代码。

这个函数也是 vDSO 的一部分。

之前的示例代码通过在 ELF 辅助向量查找类型为 `AT_SYSINFO` 的数据来找到 `__kernel_vsyscall` 的地址。

类似地，为了找到 vDSO，应用程序可以在 ELF 辅助向量中查找类型为 `AT_SYSINFO_EHDR` 的数据结构。这个数据结构中就包含了 vDSO 的 ELF 头的起始地址，该起始地址是由链接器脚本产生的。

对于这两种类型，都是在应用程序加载时，由内核把对应的函数地址写入到 ELF 头部的。这就是为什么我们查到 `AT_SYSINFO_EHDR` 和 `AT_SYSINFO` 时能得到正确函数地址的原因。

一旦找到了 ELF 头，应用程序就可以解析这个 ELF 对象（可以用 libelf），并且调用 ELF 对象中的函数了。

这种方式的优势在于 vDSO 可以利用一些 ELF 提供的有效功能，例如多版本符号。

在内核文档 `Documentation/vDSO/` 中有一个例子，展示了如何解析和调用 vDSO 中的函数。

glibc 中的 vDSO

大多数情况下，我们可能都感知不到，我们正在访问 vDSO。因为 `glibc` 很好地把这些细节抽象成了之前我们描述过的接口。

当应用程序加载时，动态链接器和加载器会加载那些应用程序所代赖的 DSO，其中就包括 vDSO。

`glibc` 在应用程序加载时解析 ELF 头，并将 vDSO 中的一些地址信息保存起来。其中就包括那些在执行系统调用之前需要查找 vDSO 的一些符号。

例如 `glibc` 中的 `gettimeofday` 函数，在源文件 `sysdeps/unix/sysv/linux/x86_64/gettimeofday.c` 中：

```
void *gettimeofday_ifunc (void) __asm__ ("__gettimeofday");

void *
gettimeofday_ifunc (void)
{
  PREPARE_VERSION (linux26, "LINUX_2.6", 61765110);

  /* If the vDSO is not available we fall back on the old vsyscal
l.  */
  return (_dl_vdso_vsym ("gettimeofday", &linux26)
          ?: (void *) VSYSCALL_ADDR_vgettimeofday);
}
__asm (".type __gettimeofday, %gnu_indirect_function");
```

这段代码在 vDSO 中找到 `gettimeofday` 函数，并返回它的函数地址。它被很好地封装在一个间接函数中。

这就是应用程序如何在调用 `gettimeofday` 时，通过 `glibc` 执行到 vDSO 中，而不需要切换到内核态，也不需要触发特权级转换以及软件中断的原因。

这样，我们就可以推导出在 32 位或 64 位 Intel 和 AMD CPU 上的 Linux 系统中的每一个系统调用的方法了。

`glibc` 系统调用封装

我们一直在讨论系统调用，因此有必要简单地说一下 `glibc` 库是如何处理系统调用的。

对大多数系统调用来说，`glibc` 只是简单地用一个封装函数把参数放到合适的寄存器里，并执行 `syscall` 或 `int $0x80` 指令，或者直接调用 `__kernel_vsyscall`。

glibc 根据定义在文本文件中的一系列表来处理系统调用，这些文本文件经过脚本处理后会生成相应的 C 代码。

例如，在文本文件 `sysdeps/unix/syscalls.list` 中描述了一些常见的系统调用：

```
access - access i:si __access access
acct - acct i:S acct
chdir - chdir i:s __chdir chdir
chmod - chmod i:si __chmod chmod
```

如果想要了解关于每一列的含义，可以参考处理该文件的脚本 `sysdeps/unix/make-syscalls.sh` 中的注释。

更复杂一些的系统调用，比如 `exit` ，它们需要调用一些用 C 或者汇编实现的处理函数。因此，这类系统需要调用不会出现在上面所说的那个格式化的文本文件中。

后续的文章将会对一些有趣的系统调用，深入分析 `glibc` 以及 Linux 内核的实现。

与系统调用相关的有趣问题

讲到这儿，如果不提一下与系统调用相关的两个绝妙 bug，实在是太浪费这个机会了。

下面让我们来看看吧。

CVE-2010-3301

这个安全漏洞可能使本地用户获取 root 权限。

问题的原因在于汇编代码中的一个小 bug，它使得应用程序可以在 x86-64 系统上执行中断方式的系统调用。

攻击代码写得相当聪明：它用 mmap 在指定的位置映射生成了一段内存，并利用一个整型溢出来使得下面这句代码：

（还记得这是软件中断方式系统调用中用的到吧。）

```
call *ia32_sys_call_table(,%rax,8)
```

把系统执行跳转到一个任意的地址上，而且是运行在内核模式下。这样，用户就可以把当前进程提升为 root。

安卓 `sysenter` ABI 问题

还记得我们之前讲过，不要在应用程序中硬编码 `sysenter` 的 ABI 调用约定吧？

不幸的是，安卓 x86 的开发人员就犯了这个错误。当内核 ABI 发生变化时，安卓 x86 突然就出问题了。

内核开发人员后来不得不恢复了原来的 `sysenter` ABI 调用约定，来避免已经大量使用硬编码 `sysenter` ABI 的安卓设备无法使用。

这是提交到 Linux 内核的解决方案。在它的提交注释中，我们可以找到引入这个错误的那次安卓代码的提交链接。

一定要记住：千成不要自己写 `sysenter` 的汇编代码。如果你万不得已必须这样做，至少要像之前的例子那样调用 `__kernel_vsyscall`。

结论

Linux 内核的系统调用复杂地难以想象吧。不但有好多种不同的方式来执行系统调用，而且每一种方式都有自己的优点和缺点。

通常来说，自己写汇编代码来执行系统调用绝对不是一个好主意，因为 ABI 的变化可能随时会毁了你的代码，让它们无法正常工作。一般来说，系统的内核和 libc 的实现都会用最快的方式来完成系统调用。

如果你不能使用 `glibc` 提供的封装函数（或者这个系统调用没有封装函数），你应该至少选择使用 `syscall` 封装函数，或者试着调用 vDSO 提供的 `__kernel_vsyscall`。

后续我们还会有文章来专门研究某些系统调用及其实现。

相关阅读

如果你喜欢这篇文章，你可能还会喜欢下面这些底层技术细节的文章：

- `strace` 是如何工作的？
- `ltrace` 是如何工作的？
- APT 哈希和不匹配
- HOWTO: GPG 签名和校验 deb 包以及 APT 库
- HOWTO: GPG 签名和校验 RPM 包以及 yum 库