The Definitive Guide to Linux System Calls - Packagecloud Blog

笔记本:	内核
创建时间:	2021/4/30 14:56
URL:	https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-syst...

The Definitive Guide to Linux System Calls

Apr 5, 2016 • packagecloud

Tags: linux

TL;DR

This blog post explains how Linux programs call functions in the Linux kernel.

It will outline several different methods of making systems calls, how to handcraft your own assembly to make system calls (examples included), kernel entry points into system calls, kernel exit points from system calls, glibc wrappers, bugs, and much, much more.

What is a system call?

When you run a program which calls `open`, `fork`, `read`, `write` (and many others) you are making a system call.

System calls are how a program enters the kernel to perform some task. Programs use system calls to perform a variety of operations such as: creating processes, doing network and file IO, and much more.

You can find a list of system calls by checking the man page for syscalls(2).

There are several different ways for user programs to make system calls and the low-level instructions for making a system call vary among CPU architectures.

As an application developer, you don't typically need to think about how exactly a system call is made. You simply include the appropriate header file and make the call as if it were a normal function.

`glibc` provides wrapper code which abstracts you away from the underlying code which arranges the arguments you've passed and enters the kernel.

Before we can dive into the details of how system calls are made, we'll need to define some terms and examine some core ideas that will appear later.

Prerequisite information

Hardware and software

This blog post makes the following assumptions that:

- You are using a 32-bit or 64-bit Intel or AMD CPU. The discussion about the methods may be useful for people using other systems, but the code samples below contain CPU-specific code.

- You are interested in the Linux kernel, version 3.13.0. Other kernel versions will be similar, but the exact line numbers, organization of code, and file paths will vary. Links to the 3.13.0 kernel source tree on GitHub are provided.
- You are interested in `glibc` or `glibc` derived libc implementations (e.g., `eglibc`).

x86-64 in this blog post will refer to 64bit Intel and AMD CPUs that are based on the x86 architecture.

User programs, the kernel, and CPU privilege levels

User programs (like your editor, terminal, ssh daemon, etc) need to interact with the Linux kernel so that the kernel can perform a set of operations on behalf of your user programs that they can't perform themselves.

For example, if a user program needs to do some sort of IO (`open`, `read`, `write`, etc) or modify its address space (`mmap`, `sbrk`, etc) it must trigger the kernel to run to complete those actions on its behalf.

What prevents user programs from performing these actions themselves?

It turns out that the x86-64 CPUs have a concept called privilege levels. Privilege levels are a complex topic suitable for their own blog post. For the purposes of this post, we can (greatly) simplify the concept of privilege levels by saying:

1. Privilege levels are a means of access control. The current privilege level determines which CPU instructions and IO may be performed.
2. The kernel runs at the most privileged level, called “Ring 0”. User programs run at a lesser level, typically “Ring 3”.

In order for a user program to perform some privileged operation, it must cause a privilege level change (from “Ring 3” to “Ring 0”) so that the kernel can execute.

There are several ways to cause a privilege level change and trigger the kernel to perform some action.

Let’s start with a common way to cause the kernel to execute: interrupts.

Interrupts

You can think of an interrupt as an event that is generated (or “raised”) by hardware or software.

A hardware interrupt is raised by a hardware device to notify the kernel that a particular event has occurred. A common example of this type of interrupt is an interrupt generated when a NIC receives a packet.

A software interrupt is raised by executing a piece of code. On x86-64 systems, a software interrupt can be raised by executing the `int` instruction.

Interrupts usually have numbers assigned to them. Some of these interrupt numbers have a special meaning.

You can imagine an array that lives in memory on the CPU. Each entry in this array maps to an interrupt number. Each entry contains the address of a function that the CPU will begin executing when that interrupt is received along with some options, like what privilege level the interrupt handler function should be executed in.

Here’s a photo from the Intel CPU manual showing the layout of an entry in this array:

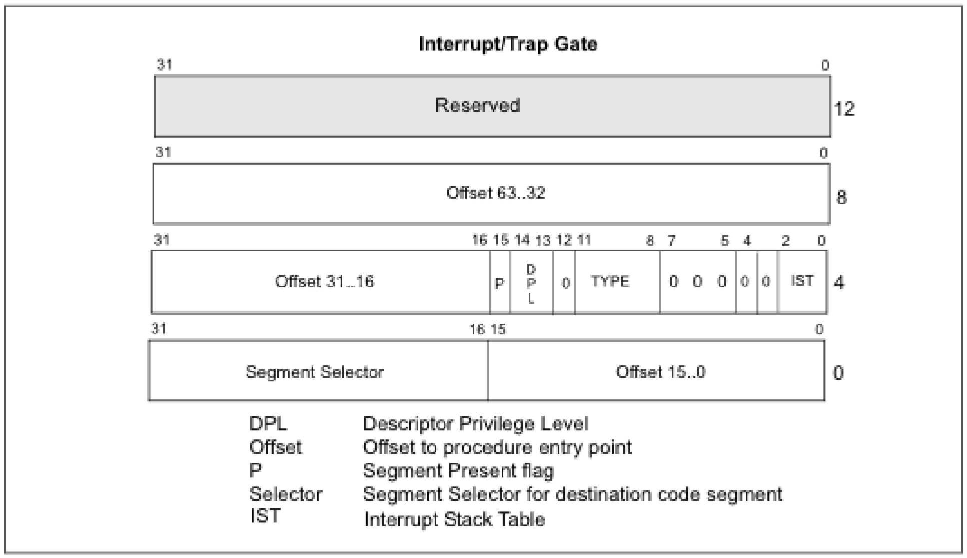


Figure 5-7. 64-Bit IDT Gate Descriptors

If you look closely at the diagram, you can see a 2-bit field labeled DPL (Descriptor Privilege Level). The value in this field determines the minimum privilege level the CPU will be in when the handler function is executed.

This is how the CPU knows which address it should execute when a particular type of event is received and what privilege level the handler for that event should execute in.

In practice, there are lots of different ways to deal with interrupts on x86-64 systems. If you are interested in learning more read about the 8259 Programmable Interrupt Controller, Advanced Interrupt Controllers, and IO Advanced Interrupt Controllers.

There are other complexities involved with dealing with both hardware and software interrupts, such as interrupt number collisions and remapping.

We don't need to concern ourselves with these details for this discussion about system calls.

Model Specific Registers (MSRs)

Model Specific Registers (also known as MSRs) are control registers that have a specific purpose to control certain features of the CPU. The CPU documentation lists the addresses of each of the MSRs.

You can use the CPU instructions `rdmsr` to `wrmsr` to read and write MSRs, respectively.

There are also command line tools which allow you to read and write MSRs, but doing this is *not recommended* as changing these values (especially while a system is running) is dangerous unless you are really careful.

If you don't mind potentially destabilizing your system or irreversibly corrupting your data, you can read and write MSRs by installing `msr-tools` and loading the `msr` kernel module:

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

Some of the system call methods we'll see later make use of MSRs, as we'll see soon.

Calling system calls with assembly is a bad idea

It’s not a great idea to call system calls by writing your own assembly code.

One big reason for this is that some system calls have additional code that runs in glibc before or after the system call runs.

In the examples below, we’ll be using the `exit` system call. It turns out that you can register functions to run when `exit` is called by a program by using [`atexit`](atexit) .

Those functions are called from glibc, not the kernel. So, if you write your own assembly to call `exit` as we show below, your registered handler functions won’t be executed since you are bypassing glibc.

Nevertheless, manually making system calls with assembly is a good learning experience.

Legacy system calls

Create a package repository

in less than 10 seconds, free.

Using our prerequisite knowledge we know two things:

1. We know that we can trigger the kernel to execute by generating a software interrupt.
2. We can generate a software interrupt with the `int` assembly instruction.

Combining these two concepts leads us to the legacy system call interface on Linux.

The Linux kernel sets aside a specific software interrupt number that can be used by user space programs to enter the kernel and execute a system call.

The Linux kernel registers an interrupt handler named `ia32_syscall` for the interrupt number: 128 (0x80). Let's take a look at the code that actually does this.

From the `trap_init` function in the kernel 3.13.0 source in `arch/x86/kernel/traps.c` :

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

        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32
_syscall);
```

Where `IA32_SYSCALL_VECTOR` is a defined as `0x80` in `arch/x86/include/asm/irq_vectors.h` .

But, if the kernel reserves a single software interrupt that userland programs can raise to trigger the kernel, how does the kernel know which of the many system calls it should execute?

The userland program is expected to put the system call number in the `eax` register. The arguments for the syscall itself are to be placed in the remaining general purpose registers.

One place this is documented is in a comment in `arch/x86/ia32/ia32entry.S` :

```
 * 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, s
hould not be touched]
 *
```

Now that we know how to make a system call and where the arguments should live, let's try to make one by writing some inline assembly.

Using legacy system calls with your own assembly

To make a legacy system call, you can write a small bit of inline assembly. While this is interesting from a learning perspective, I encourage readers to never make system calls by crafting their own assembly.

In this example, we'll try calling the `exit` system call, which takes a single argument: the exit status.

First, we need to find the system call number for `exit`. The Linux kernel includes a file which lists each system call in a table. This file is processed by various scripts at build time to generate header files which can be used by user programs.

Let's look at the table found in `arch/x86/syscalls/syscall_32.tbl`:

```
1 i386   exit      sys_exit
```

The `exit` syscall is number `1`. According to the interface described above, we just need to move the syscall number into the `eax` register and the first argument (the exit status) into `ebx`.

Here's a piece of C code with some inline assembly that does this. Let's set the exit status to "42":

(This example can be simplified, but I thought it would be interesting to make it a bit more wordy than necessary so that anyone who hasn't seen

GCC inline assembly before can use this as an example or reference.)

```
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 anyth
ing, no none */
       /* (none) */
     : /* input parameters mapped to %0 and %1, repsect
ively */
       "m" (syscall_nr), "m" (exit_status)
     : /* registers that we are "clobbering", unneeded
 since we are calling exit */
       "eax", "ebx");
}
```

Next, compile, execute, and check the exit status:

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

Success! We called the `exit` system call using the legacy system call method by raising a software interrupt.

Kernel-side: `int $0x80` entry point

So now that we’ve seen how to trigger a system call from a userland program, let’s see how the kernel uses the system call number to execute the system call code.

Recall from the previous section that the kernel registered a syscall handler function called `ia32_syscall`.

This function is implemented in assembly in [`arch/x86/ia32/ia32entry.S`]() and we can see several things happening in this function, the most important of which is the call to the actual syscall itself:

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

`IA32_ARG_FIXUP` is a macro which rearranges the legacy arguments so that they may be properly understood by the current system call layer.

The `ia32_sys_call_table` identifier refers to a table which is defined in [`arch/x86/ia32/syscall_ia32.c`]() . Note the `#include` line toward the end of the code:

```
const sys_call_ptr_t ia32_sys_call_table[__NR_ia32_sys
call_max+1] = {
        /*
         * Smells like a compiler bug -- it doesn't wo
rk
         * when the & below is removed.
        */
        [0 ... __NR_ia32_syscall_max] = &compat_ni_sys
call,
#include <asm/syscalls_32.h>
};
```

Recall earlier we saw the syscall table defined in `arch/x86/syscalls/syscall_32.tbl` .

There are a few scripts which run at compile time which take this table and generate the `syscalls_32.h` file from it. The generated header file is comprised of valid C code, which is simply inserted with the `#include` shown above to fill in `ia32_sys_call_table` with function addresses indexed by system call number.

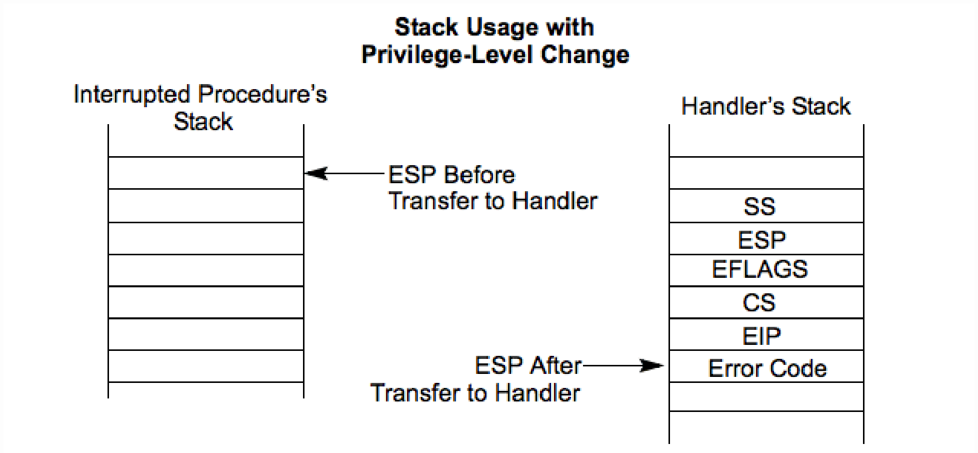
And this is how you enter the kernel via a legacy system call.

Returning from a legacy system call with `iret`

We've seen how to enter the kernel with a software interrupt, but how does the kernel return back to the user program and drop the privilege level after it has finished running?

If we turn to the (warning: large PDF) Intel Software Developer's Manual we can find a helpful diagram that illustrates how the program stack will be arranged when a privilege level change occurs.

Let's take a look:


Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
ESP After Transfer to Handler
Error Code

When execution is transferred to the kernel function `ia32_syscall` via the execution of a software interrupt from a user program, a privilege level change occurs. The result is that the stack when `ia32_syscall` is entered will look like the diagram above.

This means that the return address and the CPU flags which encode the privilege level (and other stuff), and more are all saved on the program stack before `ia32_syscall` executes.

So, in order to resume execution the kernel just needs to copy these values from the program stack back into the registers where they belong and execution will resume back in userland.

OK, so how do you do that?

There's a few ways to do that, but one of the easiest ways is to the use the `iret` instruction.

The Intel instruction set manual explains that the `iret` instruction pops the return address and saved register values from the stack in the order they were prepared:

> As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

Finding this code in the Linux kernel is a bit difficult as it is hidden beneath several macros and there is extensive care taken to deal with things like signals and ptrace system call exit tracking.

Eventually all the macros in the assembly stubs in the kernel reveal the `iret` which returns from a system call back to a user program.

From `irq_return` in `arch/x86/kernel/entry_64.S`:

```
irq_return:
  INTERRUPT_RETURN
```

Where `INTERRUPT_RETURN` is defined in `arch/x86/include/asm/irqflags.h` as `iretq`.

And now you know how legacy system calls work.

Fast system calls

The legacy method seems pretty reasonable, but there are newer ways to trigger a system call which don't involve a software interrupt and are much faster than using a software interrupt.

Each of the two faster methods is comprised of two instructions. One to enter the kernel and one to leave. Both methods are described in the Intel CPU documentation as "Fast System Call".

Unfortunately, Intel and AMD implementations have some disagreement on which method is valid when a CPU is in 32bit or 64bit mode.

In order to maximize compatibility across both Intel and AMD CPUs:

- On 32bit systems use: `sysenter` and `sysexit`.
- On 64bit systems use: `syscall` and `sysret`.

in less than 10 seconds, free.

32-bit fast system calls

`sysenter`/`sysexit`

Using `sysenter` to make a system call is more complicated than using the legacy interrupt method and involves more coordination between the user program (via `glibc`) and the kernel.

Let's take it one step at a time and sort out the details. First, let's see what the documentation in the Intel Instruction Set Reference (warning very large PDF) says about the `sysenter` and how to use it.

Let's take a look:

> Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:
>
> • IA32_SYSENTER_CS (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.

> • IA32_SYSENTER_EIP (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
>
> • IA32_SYSENTER_ESP (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

In other words: in order for the kernel to receive incoming system calls with `sysenter`, the kernel must set 3 Model Specific Registers (MSRs). The most interesting MSR in our case is `IA32_SYSENTER_EIP` (which has the address 0x176). This MSR is where the kernel should specify the address of the function that will execute when a `sysenter` instruction is executed by a user program.

We can find the code in the Linux kernel which writes to the MSR in [`arch/x86/vdso/vdso32-setup.c`]:

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

        wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long) i
a32_sysenter_target, 0);
```

Where `MSR_IA32_SYSENTER_EIP` is defined as a `0x00000176` [`arch/x86/include/uapi/asm/msr-`

`index.h` .

Much like the legacy software interrupt syscalls, there is a defined convention for making system calls with `sysenter`.

One place this is documented is in a comment in `arch/x86/ia32/ia32entry.S` :

```
* 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
```

Recall that the legacy system call method includes a mechanism for returning back to the userland program which was interrupted: the `iret` instruction.

Capturing the logic needed to make `sysenter` work properly is complicated because unlike software interrupts, `sysenter` does not store the return address.

How, exactly, the kernel does this and other bookkeeping prior to executing a `sysenter` instruction can change over time (and it has changed, as you will see in the Bugs section below).

In order to protect against future changes, user programs are intended to use a function called `__kernel_vsyscall` which is implemented in the kernel, but mapped into each user process when the process is started.

This is a bit odd; it's code that comes with the kernel, but runs in userland.

It turns out that `__kernel_vsyscall` is part of something called a virtual Dynamic Shared Object (vDSO) which exists to allow programs to execute kernel code in userland.

We'll examine what the vDSO is, what it does, and how it works in depth later.

For now, let's examine the `__kernel_vsyscall` internals.

`__kernel_vsyscall` internals

The `__kernel_vsyscall` function that encapulates the `sysenter` calling convention can be found in [`arch/x86/vdso/vdso32/sysenter.S`](arch/x86/vdso/vdso32/sysenter.S) :

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

`__kernel_vsyscall` is part of a Dynamic Shared Object (also known as a shared library) how does a user program locate the address of that function at runtime?

The address of the `__kernel_vsyscall` function is written into an ELF auxilliary vector where a user program or library (typically `glibc`) can find it and use it.

There are a few methods for searching ELF auxilliary vectors:

1. By using `getauxval` with the `AT_SYSINFO` argument.
2. By iterating to the end of the environment variables and parsing them from memory.

Option 1 is the simplest option, but does not exist on `glibc` prior to 2.16. The example code shown below illustrates option 2.

As we can see in the code above, `__kernel_vsyscall` does some bookkeeping before executing `sysenter`.

So, all we need to do to manually enter the kernel with `sysenter` is:

- Search the ELF auxilliary vectors for `AT_SYSINFO` where the address of `__kernel_vsyscall` is written.
- Put the system call number and arguments into the registers as we would normally for legacy system calls
- Call the `__kernel_vsyscall` function

You should absolutely never write your own `sysenter` wrapper function as the convention the kernel uses to enter and leave system calls with `sysenter` can change and your code will break.

You should *always* start a `sysenter` system call by calling through `__kernel_vsyscall`.

So, lets do that.

Using `sysenter` system calls with your own assembly

Keeping with our legacy system call example from earlier, we'll call `exit` with an exit status of `42`.

The `exit` syscall is number `1`. According to the interface described above, we just need to move the syscall number into the `eax` register and the first argument (the exit status) into `ebx`.

(This example can be simplified, but I thought it would be interesting to make it a bit more wordy than necessary so that anyone who hasn't seen GCC inline assembly before can use this as an example or reference.)

```
#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 t
he environment
   * variables
   *
   * check this helpful diagram: https://static.lwn.ne
t/images/2012/auxvec.png
   */
  while(*envp++ != NULL);

  /* envp is now pointed at the auxilliary vectors, si
nce we've iterated
   * 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 th
e 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 any
thing, no none */
        /* (none) */
      : /* input parameters mapped to %0 and %1, repse
ctively */
        "m" (syscall_nr), "m" (exit_status), "m" (auxv
->a_un.a_val)
      : /* registers that we are "clobbering", unneede
d since we are calling exit */
        "eax", "ebx");
}
```

Next, compile, execute, and check the exit status:

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

Success! We called the `exit` system call using the legacy sysenter method without raising a software interrupt.

Kernel-side: sysenter entry point

So now that we've seen how to trigger a system call from a userland program with `sysenter` via `__kernel_vsyscall`, let's see how the kernel uses the system call number to execute the system call code.

Recall from the previous section that the kernel registered a syscall handler function called `ia32_sysenter_target`.

This function is implemented in assembly in `arch/x86/ia32/ia32entry.S`. Let’s take a look at where the value in the eax register is used to execute the system call:

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

This is identical code as we saw in the legacy system call mode: a table named `ia32_sys_call_table` which is indexed into with the system call number.

After all the needed bookkeeping is done both the legacy system call model and the `sysenter` system call model use the same mechanism and system call table for dispatching system calls.

Refer to the `int $0x80` entry point section to learn where the `ia32_sys_call_table` is defined and how it is constructed.

And this is how you enter the kernel via a `sysenter` system call.

Returning from a `sysenter` system call with `sysexit`

The kernel can use the `sysexit` instruction to resume execution back to the user program.

Using this instruction is not as straight forward as using `iret`. The caller is expected to put the address to return to into the `rdx` register, and to put the pointer to the program stack to use in the `rcx` register.

This means that your software must compute the address where execution should be resumed, preserve that value, and restore it prior to calling `sysexit`.

We can find the code which does this in: arch/x86/ia32/ia32entry.S:

```
sysexit_from_sys_call:
        andl    $~TS_COMPAT,TI_status+THREAD_INFO(%rs
p,RIP-ARGOFFSET)
        /* clear IF, that popfq doesn't enable interru
pts early */
        andl  $~0x200,EFLAGS-R11(%rsp)
        movl    RIP-R11(%rsp),%edx              /* Use
r %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                           /* Use
r %esp */
        CFI_REGISTER rsp,rcx
        TRACE_IRQS_ON
        ENABLE_INTERRUPTS_SYSEXIT32
```

`ENABLE_INTERRUPTS_SYSEXIT32` is a macro which is defined in arch/x86/include/asm/irqflags.h which contains the `sysexit` instruction.

And now you know how 32-bit fast system calls work.

64-bit fast system calls

Next up on our journey are 64-bit fast system calls. These system calls use the instructions `syscall` and `sysret` to enter and return from a system call, respectively.

`syscall`/`sysret`

The documentation in the Intel Instruction Set Reference (very large PDF) explains how the `syscall` instruction works:

> SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX).

In other words: for the kernel to receive incoming system calls, it must register the address of the code that will execute when a system call occurs by writing its address to the `IA32_LSTAR` MSR.

We can find that code in the kernel in [`arch/x86/kernel/cpu/common.c`]:

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

Where `MSR_LSTAR` is defined as `0xc0000082` in [`arch/x86/include/uapi/asm/msr-index.h`].

Much like the legacy software interrupt syscalls, there is a defined convention for making system calls with `syscall`.

The userland program is expected to put the system call number to be in the `rax` register. The arguments to the syscall are expected to be placed in a subset of the general purpose registers.

This is documented in the x86-64 ABI in section A.2.1:

> 1. User-level applications use as integer registers for passing the sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9. The kernel interface uses %rdi, %rsi, %rdx, %r10, %r8 and %r9.
> 2. A system-call is done via the syscall instruction. The kernel destroys registers %rcx and %r11.

> 3. The number of the syscall has to be passed in register %rax.
> 4. System-calls are limited to six arguments,no argument is passed directly on the stack.
> 5. Returning from the syscall, register %rax contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is -errno.
> 6. Only values of class INTEGER or class MEMORY are passed to the kernel.

This is also documented in a comment in `arch/x86/kernel/entry_64.S` .

Now that we know how to make a system call and where the arguments should live, let's try to make one by writing some inline assembly.

Using `syscall` system calls with your own assembly

Building on the previous example, let's build a small C program with inline assembly which executes the exit system call passing the exit status of 42.

First, we need to find the system call number for `exit`. In this case we need to read the table found in `arch/x86/syscalls/syscall_64.tbl` :

```
60      common  exit                    sys_exit
```

The `exit` syscall is number `60`. According to the interface described above, we just need to move `60` into the `rax` register and the first argument (the exit status) into `rdi`.

Here’s a piece of C code with some inline assembly that does this. Like the previous example, this example is more wordy than necessary in the interest of clarity:

```
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 anyth
ing, no none */
      /* (none) */
    : /* input parameters mapped to %0 and %1, repsect
ively */
      "m" (syscall_nr), "m" (exit_status)
    : /* registers that we are "clobbering", unneeded
 since we are calling exit */
      "rax", "rdi");
}
```

Next, compile, execute, and check the exit status:

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

Success! We called the `exit` system call using the `syscall` system call method. We avoided raising a software interrupt and (if we were timing a micro-benchmark) it executes much faster.

Kernel-side: syscall entry point

Now we've seen how to trigger a system call from a userland program, let's see how the kernel uses the system call number to execute the system call code.

Recall from the previous section we saw the address of a function named `system_call` get written to the `LSTAR` MSR.

Let's take a look at the code for this function and see how it uses `rax` to actually hand off execution to the system call, from [`arch/x86/kernel/entry_64.S`]:

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

Much like the legacy system call method, `sys_call_table` is a table defined in a C file that uses `#include` to pull in C code generated by a script.

From [`arch/x86/kernel/syscall_64.c`], note the `#include` at the bottom:

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

Earlier we saw the syscall table defined in `arch/x86/syscalls/syscall_64.tbl` . Exactly like the legacy interrupt mode, a script runs at kernel compile time and generates the `syscalls_64.h` file from the table in `syscall_64.tbl`.

The code above simply includes the generated C code producing an array of function pointers indexed by system call number.

And this is how you enter the kernel via a `syscall` system call.

Returning from a `syscall` system call with `sysret`

The kernel can use the `sysret` instruction to resume execution back to where execution left off when the user program used `syscall`.

`sysret` is simpler than `sysexit` because the address to where execution should be resume is copied into the `rcx` register when `syscall` is used.

As long as you preserve that value somewhere and restore it to `rcx` before calling `sysret`, execution will resume where it left off before the call to `syscall`.

This is convenient because `sysenter` requires that you compute this address yourself in addition to clobbering an additional register.

We can find the code which does this in [`arch/x86/kernel/entry_64.S`]:

```
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
```

`USERGS_SYSRET64` is a macro which is defined in [`arch/x86/include/asm/irqflags.h`] which contains the `sysret` instruction.

And now you know how 64-bit fast system calls work.

Calling a syscall semi-manually with

syscall(2)

Great, we've seen how to call system calls manually by crafting assembly for a few different system call methods.

Usually, you don't need to write your own assembly. Wrapper functions are provided by glibc that handle all of the assembly code for you.

There are some system calls, however, for which no glibc wrapper exists. One example of a system call like this is `futex`, the fast userspace locking system call.

But, wait, why does no system call wrapper exist for `futex`?

`futex` is intended only to be called by libraries, not application code, and thus in order to call `futex` you must do it by:

1. Generating assembly stubs for every platform you want to support
2. Using the `syscall` wrapper provided by glibc

If you find yourself in the situation of needing to call a system call for which no wrapper exists, you should definitely choose option 2: use the function `syscall` from glibc.

Let's use `syscall` from glibc to call `exit` with exit status of `42`:

```
#include <unistd.h>

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

  syscall(syscall_nr, exit_status);
}
```

Next, compile, execute, and check the exit status:

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

Success! We called the `exit` system call using the `syscall` wrapper from glibc.

glibc syscall wrapper internals

Let's take a look at the `syscall` wrapper function we used in the previous example to see how it works in glibc.

From `sysdeps/unix/sysv/linux/x86_64/syscall.S` :

```
/* Usage: long syscall (syscall_number, arg1, arg2, ar
g3, arg4, arg5, arg6)
   We need to do some arg shifting, the syscall_number
will be in
   rax.  */


        .text
ENTRY (syscall)
        movq %rdi, %rax         /* Syscall number -> r
ax.  */
        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 stac
k.  */
        syscall                 /* Do the system call.
  */
        cmpq $-4095, %rax       /* Check %rax for erro
r.  */
        jae SYSCALL_ERROR_LABEL /* Jump to error handl
er if error.  */
L(pseudo_end):
        ret                     /* Return to caller.
*/
```

Earlier we showed an excerpt from the x86_64 ABI document that describes both userland and kernel calling conventions.

This assembly stub is cool because it shows *both* calling conventions. The arguments passed into this function follow the userland calling convention, but are then moved to a different set of registers to obey the kernel calling convention prior to entering the kernel with `syscall`.

This is how the glibc syscall wrapper works when you use it to call system calls that do not come with a wrapper by default.

Virtual system calls

We've now covered all the methods of making a system call by entering the kernel and shown how you can make those calls manually (or semi-manually) to transition the system from userland to the kernel.

What if programs could call certain system calls without entering the kernel at all?

That's precisely why the Linux virtual Dynamic Shared Object (vDSO) exists. The Linux vDSO is a set of code that is part of the kernel, but is mapped into the address space of a user program to be run in userland.

The idea is that some system calls can be used without entering the kernel. One such call is: `gettimeofday`.

Programs calling the `gettimeofday` system call do not actually enter the kernel. They instead make a simple function call to a piece of code that was *provided* by the kernel, but is run in userland.

No software interrupt is raised, no complicated `sysenter` or `syscall` bookkeeping is required. `gettimeofday` is just a normal function call.

You can see the vDSO listed as the first entry when you use `ldd`:

```
$ ldd `which bash`
  linux-vdso.so.1 =>  (0x00007fff667ff000)
  libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5
 (0x00007f623df7d000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00
007f623dd79000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000
7f623d9ba000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

Let's see how the vDSO is setup in the kernel.

vDSO in the kernel

You can find the vDSO source in `arch/x86/vdso/` . There are a few assembly and C source files along with a linker script.

The linker script is a cool thing to take a look at.

From `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: *;
        };
}
```

Linker scripts are pretty useful, but not particularly very well known. This linker script arranges the symbols that are going to be exported in the vDSO.

We can see that vDSO exports 4 different functions, each with two names. You can find the source for these functions in the C files in this directory.

For example, the source for `gettimeofday` found in `arch/x86/vdso/vclock_gettime.c` :

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

This is defining `gettimeofday` to be a weak alias for `__vdso_gettimeofday`.

The `__vdso_gettimeofday` function in the same file contains the actual source which will be executed *in user land* when a user program calls the `gettimeofday` system call.

Locating the vDSO in memory

Due to address space layout randomization the vDSO will be loaded at a random address when a program is started.

How can user programs find the vDSO if its loaded at a random address?

If you recall earlier when examining the `sysenter` system call method we saw that user programs should call `__kernel_vsyscall` instead of writing their own `sysenter` assembly code themselves.

This function is part of the vDSO, as well.

The sample code provided located `__kernel_vsyscall` by searching the ELF auxilliary headers to find a header with type `AT_SYSINFO` which contained the address of `__kernel_vsyscall`.

Similarly, to locate the vDSO, a user program can search for an ELF auxilliary header of type `AT_SYSINFO_EHDR`. It will contain the address of the start of the ELF header for the vDSO that was generated by a linker script.

In both cases, the kernel writes the address in to the ELF header when the program is loaded. That's how the correct addresses always end up in `AT_SYSINFO_EHDR` and `AT_SYSINFO`.

Once that header is located, user programs can parse the ELF object (perhaps using libelf) and call the functions in the ELF object as needed.

This is nice because this means that the vDSO can take advantage of some useful ELF features like symbol versioning.

An example of parsing and calling functions in the vDSO is provided in the kernel documentation in `Documentation/vDSO/`.

vDSO in glibc

Most of the time, people access the vDSO without knowing it because `glibc` abstracts this away from them by using the interface described in the previous section.

When a program is loaded, the dynamic linker and loader loads the DSOs that the program depends on, including the vDSO.

`glibc` stores some data about the location of the vDSO when it parses the ELF headers of the program that is being loaded. It also includes short stub functions that will search the vDSO for a symbol name prior to making an actual system call.

For example, the `gettimeofday` function in `glibc`, from `sysdeps/unix/sysv/linux/x86_64/gettimeofday.c`:

```
void *gettimeofday_ifunc (void) __asm__ ("__gettimeofd
ay");

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

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

This code in `glibc` searches the vDSO for the `gettimeofday` function and returns the address. This is wrapped up nicely with an indirect function.

That's how programs calling `gettimeofday` pass through `glibc` and hit the vDSO all without switching into kernel mode, incurring a privilege level change, or raising a software interrupt.

And, that concludes the showcase of every single system call method available on Linux for 32-bit and 64-bit Intel and AMD CPUs.

glibc system call wrappers

While we're talking about system calls ;) it makes sense to briefly mention how `glibc` deals with system calls.

For many system calls, `glibc` simply needs a wrapper function where it moves arguments into the proper registers and then executes the `syscall` or `int $0x80` instructions, or calls `__kernel_vsyscall`.

It does this by using a series of tables defined in text files that are processed with scripts and output C code.

For example, the `sysdeps/unix/syscalls.list` file describes some common system calls:

```
access          -       access          i:si    __acce
ss         access
acct            -       acct            i:S     acct
chdir           -       chdir           i:s     __chdi
r          chdir
chmod           -       chmod           i:si    __chmo
d          chmod
```

To learn more about each column, check the comments in the script which processes this file: `sysdeps/unix/make-syscalls.sh` .

More complex system calls, like `exit` which invokes handlers have actual implementations in C or assembly code and will not be found in a templated text file like this.

Future blog posts will explore the implementation in `glibc` and the linux kernel for interesting system calls.

Interesting syscall related bugs

It would be unfortunate not to take this opportunity to mention two fabulous bugs related to system calls in Linux.

So, let’s take a look!

CVE-2010-3301

This security exploit allows local users to gain root access.

The cause is a small bug in the assembly code which allows user programs to make legacy system calls on x86-64 systems.

The exploit code is pretty clever: it generates a region of memory with `mmap` at a particular address and uses an integer overflow to cause this code:

(Remember this code from the legacy interrupts section above?)

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

to hand execution off to an arbitrary address which runs as kernel code and can escalate the running process to root.

Android sysenter ABI breakage

Remember the part about not hardcoding the `sysenter` ABI in your application code?

Unfortunately, the android-x86 folks made this mistake. The kernel ABI changed and suddenly android-x86 stopped working.

The kernel folks ended up restoring the old `sysenter` ABI to avoid breaking the Android devices in the wild with stale hardcoded `sysenter` sequences.

Here's the fix that was added to the Linux kernel. You can find a link to the offending commit in the android source in the commit message.

Remember: never write your own `sysenter` assembly code. If you have to implement it directly for some reason, use a piece of code like the example above and go through `__kernel_vsyscall` at the very least.

Conclusion

The system call infrastructure in the Linux kernel is incredibly complex. There are many different methods for making system calls each with their own advantages and disadvantages.

Calling system calls by crafting your own assembly is generally a bad idea as the ABI may break underneath you. Your kernel and libc implementation will (probably) choose the fastest method for making system calls on your system.

If you can't use the `glibc` provided wrappers (or if one doesn't exist), you should at the very least use the `syscall` wrapper function, or try to go through the vDSO provided `__kernel_vsyscall`.

Stay tuned for future blog posts investigating individual system calls and their implementations.

Related Posts

If you enjoyed this post, you may also enjoy other

Features	Info	HOWTOs	Guides	Docs
Travis CI	Pricing	NPM/NodeJS HOWTO	Maven Guide	General Docs
Jenkins	Private NPM registry	Maven HOWTO	Debian Guide	API Docs
Buildkite	Private DEB repository	Java HOWTO	RPM Guide	Comman Line Interface
Public Package Repository	Private RPM repository	Debian HOWTO	RubyGem Guide	
Private Package Repository	Private RubyGem server	RPM HOWTO	Python Guide	
GPG Signatures	Private PyPI server	RubyGem HOWTO	Linux Guide	
	Private Maven repository	Python HOWTO		
		Linux HOWTO		