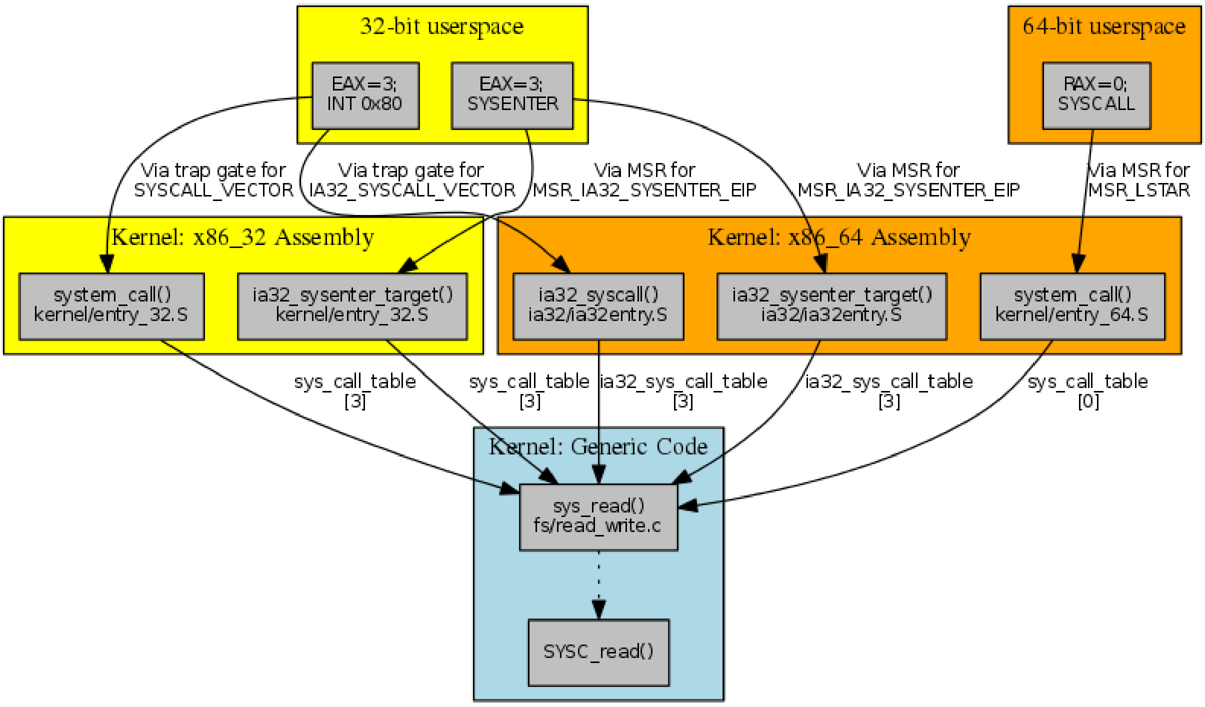
Anatomy of a system call, part 2

lwn.net/Articles/604515/

July 16, 2014

This article was contributed by David Drysdale

The previous article explored the kernel implementation of system calls (syscalls) in its most vanilla form: a normal syscall, on the most common architecture: x86_64. We complete our look at syscalls with variations on that basic theme, covering other x86 architectures and other syscall mechanisms. We start by exploring the various 32-bit x86 architecture variants, for which a map of the territory involved may be helpful. The map is clickable on the filenames and arrow labels to link to the code referenced:



x86_32 syscall invocation via `SYSENTER`

The normal invocation of a system call on a 32-bit x86_32 system is closely analogous to the mechanism for x86_64 systems that was described in the previous article. Here, the `arch/x86/syscalls/syscall_32.tbl` table has an entry for `sys_read`:

```
3 i386 read    sys_read
```

This entry indicates that `read()` for x86_32 has syscall number 3, with entry point `sys_read()` and an `i386` calling convention. The table post-processor will emit a `__SYSCALL_I386(3, sys_read, sys_read)` macro call into the generated arch/x86/include/generated/asm/syscalls_32.h file. This, in turn, is used to build the syscall table, sys_call_table, as before.

Moving outward, sys_call_table is accessed from the ia32_sysenter_target entry point of arch/x86/kernel/entry_32.S. However, here the SAVE_ALL macro actually pushes a different set of registers (EBX/ECX/EDX/ESI/EDI/EBP rather than RDI/RSI/RDX/R10/R8/R9) onto the stack, reflecting the different syscall ABI convention for this platform.

The location of the ia32_sysenter_target entry point gets written to a model-specific register (MSR) at kernel start (in enable_sep_cpu()); in this case, the MSR in question is MSR_IA32_SYSENTER_EIP (0x176), which is used for handling the SYSENTER instruction.

This shows the invocation path from userspace. The standard modern ABI for how x86_32 programs invoke a system call is to put the system call number (3 for read()) into the EAX register, and the other parameters into specific registers (EBX, ECX, and EDX for the first 3 parameters), then invoke the SYSENTER instruction.

This instruction causes the processor to transition to ring 0 and invoke the code referenced by the MSR_IA32_SYSENTER_EIP model-specific register — namely ia32_sysenter_target. That code pushes the registers onto the (kernel) stack, and calls the function pointer at entry EAX in sys_call_table — namely sys_read(), which is a thin, asmlinkage wrapper for the real implementation in SYSC_read().

x86_32 syscall invocation via INT 0x80

The sys_call_table table is also accessed in arch/x86/kernel/entry_32.S from the system_call assembly entry point. Again, this entry point saves registers to the stack, then uses the EAX register to pick the relevant entry in sys_call_table and call it. This time, the location of the system_call entry point is used by trap_init():

```
#ifdef CONFIG_X86_32
 set_system_trap_gate(SYSCALL_VECTOR, &system_call);
 set_bit(SYSCALL_VECTOR, used_vectors);
#endif
```

This sets up the handler for the SYSCALL_VECTOR trap to be system_call; that is, it sets it up to be the recipient of the venerable INT 0x80 software interrupt method for invoking system calls.

This is the original user-space invocation path for system calls, which is now generally avoided because, on modern processors, it's slower than the instructions that are specifically designed for system call invocation (SYSCALL and SYSENTER).

With this older ABI, programs invoke a system call by putting the system call number into the EAX register, and the other parameters into specific registers (EBX, ECX, and EDX for the first 3 parameters), then invoking the INT 0x80 instruction. This instruction causes the processor to transition to ring 0 and invoke the trap handler for software interrupt 0x80 —

namely `system_call`. The `system_call` code pushes the registers onto the (kernel) stack, and calls the function pointer at entry EAX in the `sys_call_table` table — namely `sys_read()`, the thin, `asmlinkage` wrapper for the real implementation in `SYSC_read()`. Much of that should seem familiar, as it is the same as for using `SYSENTER`.

x86 syscall invocation mechanisms

For reference, the different user-space syscall invocation mechanisms on x86 we've seen so far are as follows:

- 64-bit programs use the `SYSCALL` instruction. This instruction was originally introduced by AMD, but was subsequently implemented on Intel 64-bit processors and so is the best choice for cross-platform compatibility.
- Modern 32-bit programs use the `SYSENTER` instruction, which has been present since Intel introduced the IA-32 architecture.
- Ancient 32-bit programs use the `INT 0x80` instruction to trigger a software interrupt handler, but this is much slower than `SYSENTER` on modern processors.

x86_32 syscall invocation on x86_64

Now for a more complicated case: what happens if we are running a 32-bit binary on our x86_64 system? From the user-space perspective, nothing is different; in fact, nothing can be different, because the user code being run is exactly the same.

For the `SYSENTER` case, an x86_64 kernel registers a different function as the handler in the `MSR_IA32_SYSENTER_EIP` model-specific register. This function has the same name (`ia32_sysenter_target`) as the x86_32 code but a different definition (in `arch/x86/ia32/ia32entry.S`). In particular, it pushes the old-style registers but uses a different syscall table, `ia32_sys_call_table`. This table is built from the 32-bit table of entries; in particular, it will have entry 3 (as used on 32-bit systems), rather than 0 (which is the syscall number for `read()` on 64-bit systems), mapping to `sys_read()`.

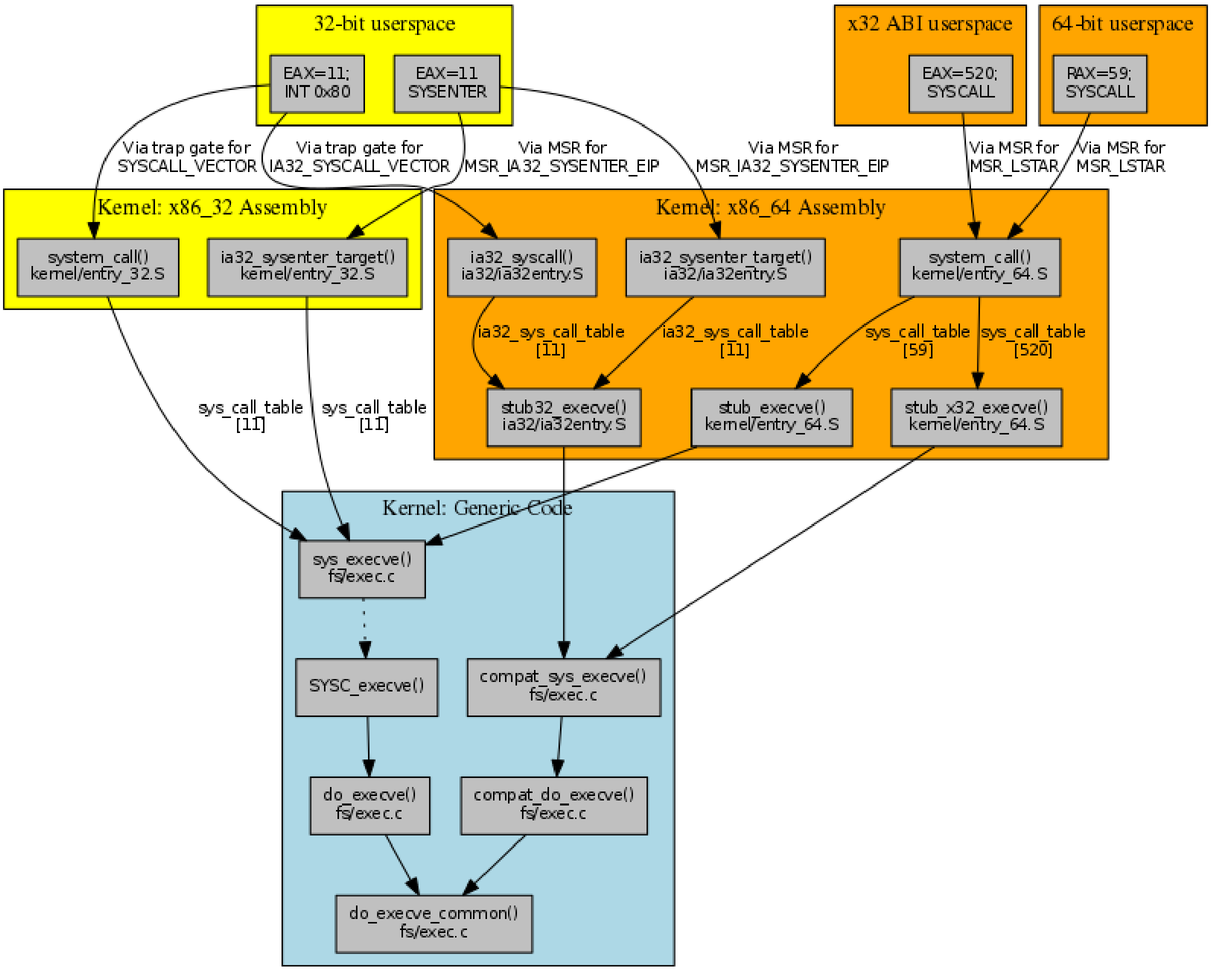
For the `INT 0x80` case, the `trap_init()` code on x86_64 instead invokes:

```
#ifdef CONFIG_IA32_EMULATION
 set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
 set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif
```

This maps the `IA32_SYSCALL_VECTOR` (which is still 0x80) to `ia32_syscall`. This assembly entry point (in `arch/x86/ia32/ia32entry.S`) uses `ia32_sys_call_table` rather than the 64-bit `sys_call_table`.

A more complex example: `execve` and 32-bit compatibility handling

Now let's look at a system call that involves other complications: `execve()`. We'll again work outward from the kernel implementation of the system call, and explore the differences from the simpler `read()` call along the way. Again, a clickable map of the territory we're going to explore might help things along:



The `execve()` definition in [`fs/exec.c`]() looks similar to that for `read()`, but there is an interesting additional function defined right after it (at least when [`CONFIG_COMPAT`]() is defined):

```
SYSCALL_DEFINE3(execve,
  const char __user *, filename,
      const char __user *const __user *, argv,
      const char __user *const __user *, envp)
{
     return do_execve(getname(filename), argv, envp);
}
#ifdef CONFIG_COMPAT
asmlinkage long compat_sys_execve(const char __user * filename,
     const compat_uptr_t __user * argv,
     const compat_uptr_t __user * envp)
{
     return compat_do_execve(getname(filename), argv, envp);
}
#endif
```

Following the processing path, these two implementations converge on `do_execve_common()` to perform the real work (`sys_execve()` → `do_execve()` → `do_execve_common()` versus `compat_sys_execve()` → `compat_do_execve()` → `do_execve_common()`), setting up [`user_arg_ptr`]() structures along the way. These structures hold those syscall arguments that are pointers-to-pointers, together with an indication of whether they come from a 32-bit compatibility ABI; if so, the value being pointed to is a 32-bit user-space address, not a 64-bit value, and the code to copy the argument values from user space needs to allow for that.

So, unlike `read()`, where the syscall implementation didn't need to distinguish between 32-bit and 64-bit callers because the arguments were pointers-to-values, `execve()` does need to distinguish, because it has arguments that are pointers-to-pointers. This turns out to be a common theme — other `compat_sys_`*`name`*`()` entry points are there to cope with pointer-to-pointer arguments (or pointer-to-struct-containing-pointer arguments, for example `struct iovec` or `struct aiocb`).

x32 ABI support

The complication of having two variant implementations of `execve()` spreads outward from the code to the system call tables. For x86_64, the 64-bit table has two distinct entries for `execve()`:

```
59 64 execve   stub_execve
...
520 x32 execve   stub_x32_execve
```

The additional entry in the 64-bit table at syscall number 520 is for **x32** ABI programs, which run on x86_64 processors but use 32-bit pointers. As a result of the `64` and `x32` ABI indicators, we will end up with `stub_execve` as entry 59 in `sys_call_table`, and `stub_x32_execve` as entry 520.

Although this is our first mention of the x32 ABI, it turns out that our previous `read()` example did quietly include x32 ABI compatibility. As no pointer-to-pointer address translation was needed, the syscall invocation path (and syscall number) could simply be shared with the 64-bit version.

Both `stub_execve` and `stub_x32_execve` are defined in `arch/x86/kernel/entry_64.S`. These entry points call on `sys_execve()` and `compat_sys_execve()`, but also save additional registers (R12-R15, RBX, and RBP) to the kernel stack. Similar `stub_*` wrappers are also present in `arch/x86/kernel/entry_64.S` for other syscalls (`rt_sigreturn()`, `clone()`, `fork()`, and `vfork()`) that may potentially need to restart user-space execution at a different address and/or with a different user stack than when the syscall was invoked.

For x86_32, the 32-bit table has an entry for `execve()` that's slightly different in format from that for `read()`:

```
11 i386 execve   sys_execve   stub32_execve
```

First of all, this tells us that `execve()` has syscall number of 11 on 32-bit systems, as compared to number 59 (or 520) on 64-bit systems. More interesting to observe is the presence of an extra field in the 32-bit table, holding a compatibility entry point `stub32_execve`. For a native 32-bit build of the kernel, this extra field is ignored and the `sys_call_table` holds `sys_execve()` as entry 11, as usual.

However, for a 64-bit build of the kernel, the IA-32 compatibility code inserts the `stub32_execve()` entry point into `ia32_sys_call_table` as entry 11. This entry point is defined in `arch/x86/ia32/ia32entry.S` as:

```
PTREGSCALL stub32_execve, compat_sys_execve
```

The `PTREGSCALL` macro sets up the `stub32_execve` entry point to call on to `compat_sys_execve()` (by putting its address into RAX), and saves additional registers (R12-R15, RBX, and RBP) to the kernel stack (like `stub_execve()` above).

`gettimeofday()`: vDSO

Some system calls just read a small amount of information from the kernel, and for these, the full machinery of a ring transition is a lot of overhead. The vDSO (Virtual Dynamically-linked Shared Object) mechanism speeds up some of these read-only syscalls by mapping the page containing the relevant information (and code to read it) into user space, read-only. In particular, the page is set up in the format of an ELF shared-library, so it can be straightforwardly linked into user programs.

Running `ldd` on a normal glibc-using binary shows the vDSO as a dependency on `linux-vdso.so.1` or `linux-gate.so.1` (which `ldd` obviously can't find a file to back); it also shows up in the memory map of a running process (`[vdso]` in `cat /proc/`*`PID`*`/maps`).

Historically, vsyscall was an earlier mechanism to do something similar, which is now deprecated due to security concerns. This older article by Johan Petersson describes how vsyscall's page appears as an ELF object (at a fixed position) to user space.

There's a Linux Journal article that discusses vDSO setup in some detail (although it is now slightly out of date), so we'll just describe the basics here, as applied to the `gettimeofday()` syscall.

First, `gettimeofday()` needs to access data. To allow this, the relevant `vsyscall_gtod_data` structure is exported into a special data section called `.vvar_vsyscall_gtod_data`. Linker instructions then ensure that this `.vvar_vsyscall_gtod_data` section is linked into the kernel in the `__vvar_page` section, and at kernel startup the `setup_arch()` function calls `map_vsyscall()` to set up a fixed mapping for that `__vvar_page`.

The code that provides the core vDSO implementation of `gettimeofday()` is in `__vdso_gettimeofday()`. It's marked as `notrace` to prevent the compiler from ever adding function profiling, and also gets a weak alias as `gettimeofday()`. To ensure that the resulting page looks like an ELF shared object, the `vdso.lds.S` file pulls in `vdso-layout.lds.S` and exports both `gettimeofday()` and `__vdso_gettimeofday()` into the page.

To make the vDSO page accessible to a new user-space program, the code in `setup_additional_pages()` sets the vDSO page location to a random address chosen by `vdso_addr()` at process start time. Using a random address mitigates the security problems found with the earlier vsyscall implementation, but does mean that the user program needs a way to find the location of the vDSO page. The location is exposed to user space as an ELF auxiliary value: the binary loader for ELF format programs (`load_elf_binary()`) uses the `ARCH_DLINFO` macro to set the `AT_SYSINFO_EHDR` auxiliary value. The user-space program can then find the page using the getauxval() function to retrieve the relevant auxiliary value (although in practice the `libc` library usually takes care of this under the covers).

For completeness, we should also mention that the vDSO mechanism is used for another important syscall-related feature for 32-bit programs. At boot time, the kernel determines which of the possible x86_32 syscall invocation mechanisms is best, and puts the appropriate implementation wrapper (`SYSENTER`, `INT 0x80`, or even `SYSCALL` for an AMD 64-bit processor) into the `__kernel_vsyscall` function. User-space programs can then invoke this wrapper and be sure of getting the fastest way into the kernel for their syscalls; see Petersson's article for more details.

`ptrace()`: syscall tracing

The `ptrace()` system call is implemented in the normal manner, but it's particularly relevant here because it can cause system calls of the traced kernel task to behave differently. Specifically, the `PTRACE_SYSCALL` request aims to "arrange for the tracee to be stopped at the next entry to or exit from a system call".

Requesting `PTRACE_SYSCALL` causes the `TIF_SYSCALL_TRACE` thread information flag to be set in thread-specific data (`struct thread_info.flags`). The effect of this is architecture-specific; we'll describe the x86_64 implementation.

Looking more closely at the assembly for syscall entry (in all of x86_32, x86_64, and ia32) we see a detail that we skipped over previously: if the thread flags have any of the `_TIF_WORK_SYSCALL_ENTRY` flags (which include `TIF_SYSCALL_TRACE`) set, the syscall implementation code follows a different path to invoke `syscall_trace_enter()` instead (x86_32, x86_64, ia32). The `syscall_trace_enter()` function then performs a variety of different functions that are associated with the various per-thread flag values that were checked for with `_TIF_WORK_SYSCALL_ENTRY`:

- `TIF_SINGLESTEP`: single stepping of instructions for ptrace
- `TIF_SECCOMP`: perform secure computing checks on syscall entry
- `TIF_SYSCALL_EMU`: perform syscall emulation
- `TIF_SYSCALL_TRACE`: syscall tracing for ptrace
- `TIF_SYSCALL_TRACEPOINT`: syscall tracing for ftrace
- `TIF_SYSCALL_AUDIT`: generation of syscall audit records

In other words, `syscall_trace_enter` is the control point for a whole collection of different per-syscall interception functionality — including `TIF_SYSCALL_TRACE` syscall tracing. It ends up calling `ptrace_stop()` with `why=CLD_TRAPPED`, which notifies the tracing program (via `SIGCHLD`) that the tracee has been stopped on entry to a syscall.

Epilogue

System calls have been the standard method for user-space programs to interact with Unix kernels for decades and, consequently, the Linux kernel includes a set of facilities to make it easy to define them and to efficiently use them. Despite the invocation variations across architectures and occasional special cases, system calls also remain a remarkably homogeneous mechanism — this stability and homogeneity allows all sorts of useful tools, from `strace` to seccomp-bpf, to work in a generic way.

(Log in to post comments)