What is linux-gate.so.1?

When you use the ldd utility on a reasonably recent Linux system you'll frequently see a reference to an ethereal entity known as linux-gate.so.1:

```
ldd /bin/sh

        linux-gate.so.1 =>  (0xffffe000)

        libdl.so.2 => /lib/libdl.so.2 (0xb7fb2000)

        libc.so.6 => /lib/libc.so.6 (0xb7e7c000)

        /lib/ld-linux.so.2 (0xb7fba000)
```

What's so strange about that? It's just a dynamically loaded library, right?

Sort of, for sufficiently generous definitions of dynamically loaded library. The lack of file name in the output indicates that ldd was unable to locate a file by that name. Indeed, any attempt to find the corresponding file – whether manually or by software designed to automatically load and analyze such libraries – will be unsuccessful.

From time to time this is a cause of befuddlement and frustration for users as they go searching for a non-existent system file. You can confidently tell users on this futile quest that there's not supposed to be a linux-gate.so.1 file present anywhere on the file system; it's a virtual DSO, a shared object exposed by the kernel at a fixed address in every process' memory:

```
cat /proc/self/maps

08048000-0804c000 r-xp 00000000 08:03 7971106    /bin/cat

0804c000-0804d000 rwxp 00003000 08:03 7971106    /bin/cat

0804d000-0806e000 rwxp 0804d000 00:00 0          [heap]

b7e88000-b7e89000 rwxp b7e88000 00:00 0

b7e89000-b7fb8000 r-xp 00000000 08:03 8856588    /lib/libc-2.3.5.so
```

```
b7fb8000-b7fb9000 r-xp 0012e000 08:03 8856588    /lib/libc-2.3.5.so

b7fb9000-b7fbc000 rwxp 0012f000 08:03 8856588    /lib/libc-2.3.5.so

b7fbc000-b7fbe000 rwxp b7fbc000 00:00 0

b7fc2000-b7fd9000 r-xp 00000000 08:03 8856915    /lib/ld-2.3.5.so

b7fd9000-b7fdb000 rwxp 00016000 08:03 8856915    /lib/ld-2.3.5.so

bfac3000-bfad9000 rw-p bfac3000 00:00 0          [stack]

ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

Here cat prints its own memory map. The line marked [vdso] is the linux-gate.so.1 object in that process, a single memory page mapped at address ffffe000. A program can determine the location of the shared object in memory by examining an AT_SYSINFO entry in the ELF auxiliary vector. The auxiliary vector (auxv) is an array of pointers passed to new processes in the same way program arguments (argv) and environment variables (envp) are.

The sample output above come from an x86 box where processes live in plain old 32-bit address spaces divided into pages of 4096 bytes, making ffffe000 the penultimate page. The very last page is reserved to catch accesses through invalid pointers, e.g. dereferencing a decremented NULL pointer or a MAP_FAILED pointer returned from mmap.

I should note here that while linux-gate.so.1 is always mapped at this fixed location on that machine, the address used can differ between systems and even be randomly chosen per process as a security measure. On a system that does the latter, getting to the object is somewhat trickier and the following demonstration will not work.

When all processes share the same object at the same location it's easy to extract a copy of it if we want to take a closer look at it. For example, we can simply ask dd to dump the page from its own memory (carefully choosing an output name different from linux-gate.so.1 to avoid creating a file that's not supposed to exist):

```
dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1
```

```
1+0 records in
1+0 records out
```

We skip 1048574 because there are 220 = 1048576 pages in total and we want to extract the next to last page. The result looks like any other shared ELF object file:

```
file -b linux-gate.dso

ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), stripped
```

```
objdump -T linux-gate.dso

linux-gate.dso:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
ffffe400 l    d  .text  00000000
ffffe460 l    d  .eh_frame_hdr  00000000
ffffe484 l    d  .eh_frame      00000000
ffffe608 l    d  .useless       00000000
ffffe400 g    DF .text  00000014  LINUX_2.5   __kernel_vsyscall
00000000 g    DO *ABS*  00000000  LINUX_2.5   LINUX_2.5
ffffe440 g    DF .text  00000007  LINUX_2.5   __kernel_rt_sigreturn
ffffe420 g    DF .text  00000008  LINUX_2.5   __kernel_sigreturn
```

These symbols are entry points for the rt_sigreturn/sigreturn functions and for making virtual system calls. On the x86 platform linux-gate.so.1 was initially called linux-vsyscall.so.1, but this was changed during development to get a common name accurately reflecting its purpose across platforms: to act as a gateway between user and kernel space. Not all platforms need virtual syscalls, but they must be fairly important for x86 to warrant this elaborate mechanism.

Traditionally, x86 system calls have been done with interrupts. You may remember that the way to request operating system functions was via interrupt 33 (21h) back in the bad old MS-DOS days. Windows system calls are buried beneath layers of user-mode APIs, but at some point they too boil down to int 0x2e. Similarly, syscall implementations in Linux and other *nix kernels have been using int 0x80.

It turns out, though, that system calls invoked via interrupts are remarkably slow on the more recent members of the x86 processor family. An int 0x80 system call can be as much as an order of magnitude slower on a 2 GHz Pentium 4 than on an 850 MHz Pentium III. The impact on performance resulting from this could easily be significant, at least for applications that do a lot of system calls.

Intel recognized this problem early on and introduced a more efficient system call interface in the form of sysenter and sysexit instructions. This fast system call feature first appeared in the Pentium Pro processor, but due to hardware bugs it's actually broken in most of the early CPUs. That's why you may see claims that sysenter was introduced with Pentium II or even Pentium III.

The hardware problems also help explain why it took quite some time before operating systems started supporting fast system calls. If we ignore earlier experimental patches, Linux support for sysenter appeared in December 2002 during kernel 2.5 development. That's ten years after the instruction was defined! Microsoft started using sysenter only slightly earlier, in Windows XP.

You can find out if your Linux machine is using the sysenter instruction for system calls by disassembling __kernel_vsyscall:

```
objdump -d --start-address=0xffffe400 --stop-address=0xffffe414 linux-gate.dso
```

```
linux-gate.dso:     file format elf32-i386
```

```
Disassembly of section .text:

ffffe400 <__kernel_vsyscall>:
ffffe400:   51        push   %ecx
ffffe401:   52        push   %edx
ffffe402:   55        push   %ebp
ffffe403:   89 e5     mov    %esp,%ebp
ffffe405:   0f 34     sysenter
ffffe407:   90        nop
ffffe408:   90        nop
ffffe409:   90        nop
ffffe40a:   90        nop
ffffe40b:   90        nop
ffffe40c:   90        nop
ffffe40d:   90        nop
ffffe40e:   eb f3     jmp    ffffe403 <__kernel_vsyscall+0x3>
ffffe410:   5d        pop    %ebp
ffffe411:   5a        pop    %edx
ffffe412:   59        pop    %ecx
ffffe413:   c3        ret
```

The preferred way of invoking a system call is determined by the kernel at boot time, and evidently this box uses sysenter. On an older machine you may see int 0x80 being used instead. In case you are struggling to make sense of that jump (like I was the first time I saw it) you might be interested to learn that it's there because Linus Torvalds is a disgusting pig and proud of it. (It's a trick to handle restarting of system calls with six parameters).