Contribution ID: **97** Type: **not specified**

# Moving the Linux ABI to userspace

*Wednesday, 11 September 2019 10:00 (45 minutes)*

The ABI between Linux and user software mostly sits at the user/privileged boundary, although many architectures extend this with a small amount of special-case code that sits in userspace, such as in special pages or shared libraries (vDSOs) mapped into each user process [1] that user code can call into.

The reasons for this are a bit arbitrary: system interface libraries such as glibc and Bionic are maintained as separate projects from the kernel, by different people. The privileged/unprivileged boundary is the de facto demarcation point between projects, because by design only kernel code can run privileged.

Because Linux's user/privileged boundary and ABI are welded together in this way though, the Linux ABI is forced to evolve (or prevented from doing so) for reasons that have little to do with functionality, such as backwards compatibility for superseded interfaces, and optimisations (e.g., vDSO gettimeofday(), getcpu() etc.).

Moving implementation of pieces of kernel functionality between privileged space and userspace is currently hard due to the resulting ABI breaks, yet moving functionality into userspace (e.g., into the vDSO) has some interesting potential use cases, such as:

- Allowing the user/privileged boundary to evolve independently of the kernel ABI.
- Providing a way to push obsolete, deprecated, redundant and/or regrettable syscalls out of the kernel proper.
- Making it easier for userspace to refine its own ABI personality: so things like libc, fakeroot etc., can catch and reimplement syscalls in a transparent way.
- Migrating to a unified library-style ABI instead of relying on a patchwork of bare syscalls, vDSO etc., but without the risk of competing or incompatible implementations.

Migrating a vDSO function to be implemented in privileged space is straightforward: a stub function can be left in the vDSO for old userspace callers to use: the stub just makes the appropriate syscall.

The converse is harder, and requires syscall trapping or filtering mechanisms such as BPF or ptrace.

This presentation will describe some approaches to reflecting syscalls back to userspace, and how feasible they look.

Things I aim to cover:

- What mechanisms can be used?
- How expensive are they, and what breaks?
- What's the likely overhead of doing *all* syscalls through a vDSO or similar?

[1] e.g.,
Documentation/ABI/stable/vdso
Documentation/arm/kernel_user_helpers.txt

## I agree to abide by the anti-harassment policy

Yes

**Primary author:** MARTIN, Dave (ARM Limited)

**Presenter:** MARTIN, Dave (ARM Limited)

**Session Classification:** Kernel Summit Track

**Track Classification:** Kernel Summit talk