The vDSO on arm64

Kevin Brodsky
ARM

Linux Plumbers Conference — Android Microconference
November 3, 2016

Non-Confidential © ARM 2016
Outline

The vDSO
  What is a vDSO?
  “Virtual” syscalls
  Why a DSO?

Implementation and plumbing
  Kernel and userspace setup
  Anatomy of the vDSO on arm64

Adding a 32-bit vDSO to arm64
  Compat processes and vDSO
  Main parts of the 32-bit vDSO implementation
  Problems and solutions
  Some figures

Conclusion
The vDSO
What is a vDSO?

vDSO: virtual DSO (Dynamic Shared Object)

- A full-blown DSO (shared library), provided by the kernel
- Mapped by the kernel into all user processes
- Linked like a normal .so shared library
  - The one gdb used to complain about! (warning: Could not load shared library symbols for linux-vdso.so.1)
- Mainly meant for providing “syscalls in userspace” (virtual syscalls)
“Virtual” syscalls

It’s all about speed!

- Certain syscalls are fast to process and the syscall itself (kernel enter/exit) is a *significant overhead*.
- Certain syscalls do not require much privilege to process.

Solution: provide some code to userspace that “emulates” the syscall

- Possibly using some data made available by the kernel
- Outside of the kernel, but strongly tied to it

Typical candidates: time-related syscalls

- For instance, a “virtual” gettimeofday() can be up to 10 times faster than the normal syscall!
Why a DSO?

A significant improvement over the old vsyscall page:

- **More flexible**: no fixed offset within the vDSO
- **Cleaner**: appears like a regular library to userspace → improved debugging
- **Harder to exploit**: takes advantage of ASLR

Now included in most major architectures, deprecating (or completely replacing) the vsyscall page

### vsyscall by arch
- x86_64 2.5.6 2002 [Initial arch impl.]
- i386 2.5.53 2002

### vDSO by arch
- ppc64 2.6.12 2005
- i386 2.6.18 2006
- x86_64 2.6.23 2007
- mips 2.6.34 2010
- arm64 3.7 2012 [Initial arch impl.]
- arm 4.1 2015
Implementation and plumbing
Kernel and userspace setup

execve()

ELF loader
- Maps vDSO pages (code + data)
- Sets AT_SYSINFO0_EHDR in the auxiliary vector

Dynamic linker
- Looks up AT_SYSINFO0_EHDR in the auxiliary vector
- If set, links the vDSO (→ [vdso])

libc init
- Looks up function symbols (e.g. __vdso_gettimeofday) in [vdso]
- If found, sets global function pointers
Anatomy of the vDSO on arm64

- \texttt{__kernel_gettimeofday()}
- \texttt{__kernel_clock_gettime()}
- \texttt{__kernel_clock_getres()}
- \texttt{__kernel_rt_sigreturn()}

[vvar]

[vdso]

AT\_SYSINFO\_EHDR

 USERSPACE | KERNEL

vdso\_data

4K page

4K page
Anatomy of the vDSO on arm64

- **Function call**: Read  
  - **Write**: Syscall

**vdso_data**

- **gettimeofday()**  
  - **__kernel_gettimeofday()**  
    - **sys_gettimeofday()**

- **clock_gettime()**  
  - **__kernel_clock_gettime()**  
    - **sys_clock_gettime()**

- **clock_getres()**  
  - **__kernel_clock_getres()**  
    - **sys_clock_getres()**

- **[signal handler returning]**  
  - **__kernel_rt_sigreturn()**  
    - **sys_rt_sigreturn()**

- **update_vsyscall()**  
- **timekeeping_update()**  
  - **update_vsyscall_tz()**  
    - **settimeofday()**

10/21  Non-Confidential © ARM 2016
Adding a 32-bit vDSO to arm64
Compat processes and vDSO

- **COMPAT**: running 32-bit processes under a 64-bit kernel
  - Present on x86, arm64, mips, powerpc, ...
- Requires dedicated vDSO support
  - Present on x86, mips, powerpc, ... but not arm64
  - Partly due to arm only having a vDSO since 4.1 (glibc support only added in 2.22)
- Why bother about the performance of 32-bit processes on arm64?
  - Very little use on arm64 servers, but...
  - Still widespread on Android (apps shipped with 32-bit libraries)
  - arm64 Chromebooks run a fully 32-bit userspace (for now)
  - Vendors started implementing their own 32-bit vDSO!

→ There is a need for a 32-bit vDSO on arm64
Main parts of the 32-bit vDSO implementation

- The 32-bit vDSO (userspace library) itself
  
  `kernel/vdso32/vgettimeofday.c` Time-related syscalls (`gettimeofday()` and `clock_gettime()`)  
  
  `kernel/vdso32/sigreturn.S` sigreturn trampolines

- Install the vDSO mappings in compat user processes (and set `mm->context.vdso`)  
  
  `kernel/vdso.c` `aarch32_setup_additional_pages()`

- Tell `fs/compat_binfmt_elf.c` to set `AT_SYSINFO_EHDR`  
  
  `include/asm/elf.h` `COMPAT_ARCH_DLINFO::AT_SYSINFO_EHDR = mm->context.vdso`

- Use the sigreturn trampolines  
  
  `kernel/signal32.c` `compat_setup_return()`

For more information, have a look at the patch series: [RFC PATCH v2 0/8] arm64: Add a compat vDSO
Problems and solutions

- Some redundancy with the [vectors] page
  - Remove it (so long, kuser helpers!)
  - Move the sigreturn trampolines to [vdso]

- The arm64 vDSO is implemented in assembly → cannot be reused
  - Reuse and adapt the arm vDSO (modified to share the same data page)

- Compiling arm code: we need a 32-bit toolchain!
  - Compat vDSO only built if CROSS_COMPILE_ARM32 is set
  - Pass a clever mixture of flags to the 32-bit compiler

- Kernel support is pointless without support in libc + dynamic linker
  - Support added to glibc in 2.22 (August 2015)
  - Support added to bionic in July 2016 — but it didn’t make it into Android N 😞
Some figures

- Very simple benchmark, run on Juno R0 with 4.8-rc1 + compat vDSO
- Using glibc 2.23 compiled for arm
- Biggest gain on coarse clocks (very fast to read → maximal syscall overhead)
- Slightly lower gain in 32-bit — probably because it is not written in assembly 😊
Conclusion

- The vDSO: a useful and flexible mechanism
  - To avoid the overhead of a syscall, by doing the work in userspace
  - To provide any kind of data or code to userspace (e.g. sigreturn trampolines)

- Kernel-side implementation completely arch-specific (in practice, always more or less similar)

- libc + dynamic linker support essential!

- Proposed addition of a 32-bit vDSO to arm64
  - Very relevant for Android and Chrome OS
  - Better to have it available in mainline than implemented by each vendor
  - Closely linked to the arm vDSO
  - Patch series: [RFC PATCH v2 0/8] arm64: Add a compat vDSO
Questions
Appendices
Full benchmarks

vDSO call vs direct syscall, 64-bit and 32-bit

Time (ns)

<table>
<thead>
<tr>
<th>Function</th>
<th>64/Syscall</th>
<th>64/vDSO</th>
<th>32/Syscall</th>
<th>32/vDSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_gettime REALTIME</td>
<td>972</td>
<td>1,120</td>
<td>1,076</td>
<td>1,068</td>
</tr>
<tr>
<td>clock_gettime MONOTONIC</td>
<td>963</td>
<td>1,076</td>
<td>948</td>
<td>918</td>
</tr>
<tr>
<td>clock_gettime MONOTONIC_RAW</td>
<td>837</td>
<td>974</td>
<td>1,007</td>
<td>1,007</td>
</tr>
<tr>
<td>clock_gettime REALTIME_COARSE</td>
<td>918</td>
<td>961</td>
<td>954</td>
<td>954</td>
</tr>
<tr>
<td>clock_gettime MONOTONIC_COARSE</td>
<td>128</td>
<td>143</td>
<td>133</td>
<td>135</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>128</td>
<td>182</td>
<td>143</td>
<td>133</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>641</td>
<td>1,142</td>
<td>1,120</td>
<td>1,120</td>
</tr>
</tbody>
</table>

Time (ns)
vDSO hacking/debugging

Debugging the vDSO is a bit tricky, due to it being used by default (no easy way to opt out)

Quick hacks to ease debugging:

- Create a shared library with the libc functions you want to override and use LD_PRELOAD
- More global: modify your libc so that it only considers the vDSO if an environment variable is set
vDSO data page ([vvar])

```c
struct vdso_data {
    __u64 cs_cycle_last;         /* Timebase at clocksource init */
    __u64 raw_time_sec;         /* Raw time */
    __u64 raw_time_nsec;
    __u64 xtime_clock_sec;      /* Kernel time */
    __u64 xtime_clock_nsec;
    __u64 xtime_coarse_sec;     /* Coarse time */
    __u64 xtime_coarse_nsec;
    __u64 wtm_clock_sec;        /* Wall to monotonic time */
    __u64 wtm_clock_nsec;
    __u32 tb_seq_count;         /* Timebase sequence counter */
    __u32 cs_mono_mult;         /* NTP-adj usted clocksource multiplier */
    __u32 cs_shift;             /* Clocksource shift (mono = raw) */
    __u32 cs_raw_mult;          /* Raw clocksource multiplier */
    __u32 tz_minuteswest;       /* Whacky timezone stuff */
    __u32 tz_dsttime;
    __u32 use_syscall;
};
```