arm

Unified Virtual Dynamic Shared Object (vDSO)

An Unexpected Journey

Vincenzo Frascino
<vincenzo.frascino@arm.com>

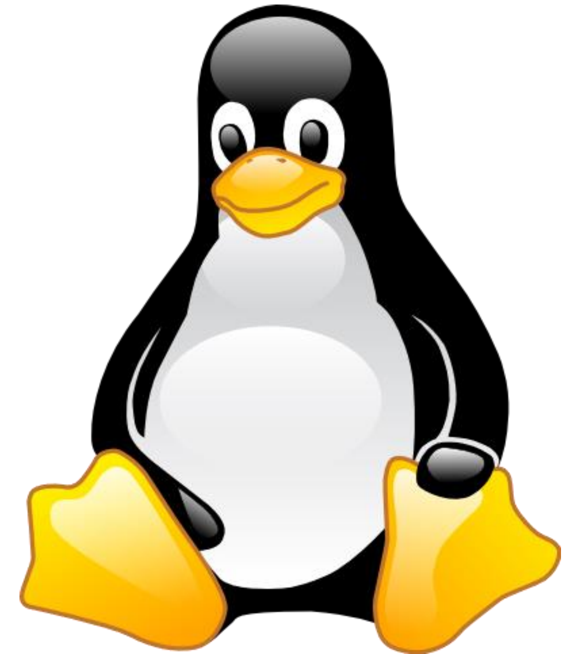
Linux Plumbers 2020 – linux/arch/* Microconference

August, 25 2020

© 2020 Arm Limited (or its affiliates)

Summary

- vDSO: Introduction
- vDSO: State of the Art
- vDSO: What’s Next?



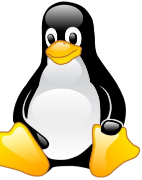
© 2020 Arm Limited (or its affiliates)

arm

vDSO: Introduction

© 2020 Arm Limited (or its affiliates)

What is a vDSO?



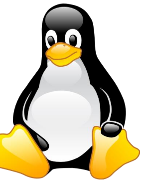
vDSO: virtual Dynamic Shared Object

- A shared object mainly intended to provide virtual “syscalls in userspace”.
- It is mapped by the kernel in all the userspace processes.
- It is linked during the kernel compilation process as a shared object (.so).

arch	version	year
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
arm	4.1	2015
…	…	…

© 2020 Arm Limited (or its affiliates)
arm

vDSO Implementation (1/2)



libc init

- Looks up for the function symbols in [vdso]
- If the symbols are present sets the function pointer (O/W libc provides fallbacks on syscalls)

dynamic linker

- Looks for AT_SYSINFO_EHDR in AUXV
- If it is set, links the vDSO shared object

USERSPACE

KERNEL

execve()

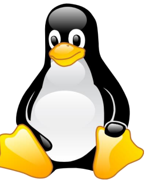
elf loader

- Maps the vDSO code and data pages
- Sets AT_SYSINFO_EHDR in AUXV

© 2020 Arm Limited (or its affiliates)

arm

vDSO Implementation (2/2)



timekeeping_update()

update_vsyscall()
update_vsyscall_tz()

settimeofday()

KERNEL | USERSPACE

PAGE_SIZE

vdso_data

AT_SYSINFO_EHDR

__vdso_gettimeofday()
__vdso_clock_gettime()
__vdso_clock_getres()
…

PAGE_SIZE

gettimeofday_fallback()

gettimeofday()

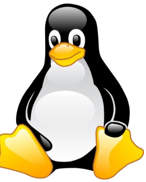
© 2020 Arm Limited (or its affiliates)

arm

vDSO: State of the Art

© 2020 Arm Limited (or its affiliates)

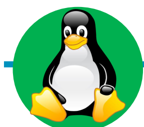
Unified vDSO



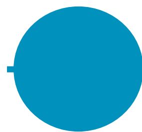
Prior to the introduction of the Unified vDSO, every architecture implemented their own vDSO library in the architectural code.

Scope: Identify the commonalities in between the architectures and try to consolidate the common code paths.

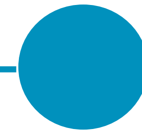
- Every architecture defines the arch specific hooks in a header in "asm/vdso/".
- The generic implementation includes the arch specific one and lives in "lib/vdso".
- The arch specific code for gettimeofday lives in "<arch path>/vdso/gettimeofday.c" and includes the generic code only.
- The generic implementation of update_vsyscall and update_vsyscall_tz lives in kernel/vdso and provides the bindings that can be implemented by each architecture.
- Each architecture provides its implementation of the bindings in "asm/vdso/vsyscall.h".
- This approach makes it possible to consolidate the common code in a single place with the benefit of avoiding code duplication.
- CLOCK_BOOTTIME and CLOCK_TAI introduced for all the supported platforms.



Unified vDSO



vDSO Headers

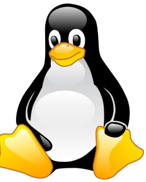


Recent activities

© 2020 Arm Limited (or its affiliates)

arm

vDSO Headers



Scope: The vDSO code runs entirely in userspace, so it should not be relying on any kernel header.

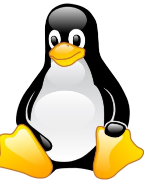
- Extract from include/linux/ the vDSO required kernel interface and place it in include/vdso/
- Make sure that where meaningful the kernel includes "vdso" headers.
- Limit the vDSO library to include headers coming only from UAPI and "vdso" (with 2 exceptions compiler.h for barriers and param.h for HZ).
- Adapt all the architectures that support the unified vDSO library to use "vdso" headers.

Back in July last year we started having a problem in building compat vDSOs on arm64 [1] [2] that was not present when the arm64 porting to the Unified vDSO was done. In particular when the compat vDSO on such architecture is built with gcc it generates the warning below:

```
In file included from ./arch/arm64/include/asm/thread_info.h:17:0,
                 from ./include/linux/thread_info.h:38,
                 from ./arch/arm64/include/asm/preempt.h:5,
                 from ./include/linux/preempt.h:78,
                 from ./include/linux/spinlock.h:51,
                 from ./include/linux/seqlock.h:36,
                 from ./include/linux/time.h:6,
                 from ./lib/vdso/gettimeofday.c:7,
                 from <command-line>:0:
./arch/arm64/include/asm/memory.h: In function '__tag_set':
./arch/arm64/include/asm/memory.h:233:15: warning: cast from pointer
                to integer of different size [-Wpointer-to-int-cast]
  u64 __addr = (u64)addr & ~__tag_shifted(0xff);
               ^
…
```

© 2020 Arm Limited (or its affiliates)
arm

Recent Activities



- riscv architecture introduced vDSO support based on Unified vDSOs.
- PowerPC and s390 portings to the Unified vDSO library are on the way.
- Time namespaces were introduced for the kernel leveraging the work done for Unified vDSOs.
- x86_64 and arm64 (5.9-rc1) portings of Time namespaces has been merged.

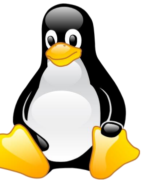
© 2020 Arm Limited (or its affiliates)
arm

arm

vDSO: What's Next?

© 2020 Arm Limited (or its affiliates)

Extend the Unified vDSOs to more Syscalls (1/2)



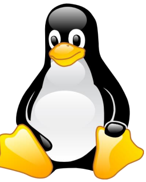
"*vDSO Principle*": Do not enter the kernel if we do not have to because it is costly.

- Libc already helps because it might not do a syscall at all if it can be avoided.
- It takes advantage of the vDSO library or caches information (e.g. getpid()).
- In doing so, the libc adds some complexity to its implementation (e.g. to cache getpid(), libc has to understand concepts like fork()/clone() so that the caching is correct).
- Since the vDSO library is already there, we could move to a model in which the cached information is provided via the library in an efficient way.
- This approach would centralize the information, simplify the implementation and improve the memory utilization (no caching required).
- But extending the vDSO library to encompass new syscalls raises some questions.

© 2020 Arm Limited (or its affiliates)

arm

Extend the Unified vDSOs to more Syscalls (2/2)



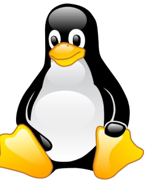
Extending the Unified vDSOs to more syscalls raises some questions:

- Which information do we want to make accessible via vDSO?
- Are those private to a process?
- Is there a risk if another process accesses them?
  - Exposing private information requires a per process page instead of a global one
- If we keep global vDSO data pages what is the best approach a per-thread one or a per-cpu one?
- Shall we make the vDSO data pages writable?
  - Should the kernel trust a writable vDSO data pages?
  - Might expose the kernel to “time of check – time of use” (TOCTOU) attacks.
  - Might require to encrypt the datapage in a way that can be decrypted/accessed only by the vDSO library.

© 2020 Arm Limited (or its affiliates)

arm

Analysing a real world scenario



Booting Ubuntu 18.04.5 (*SYSCALL_DEFINE0* class)

Number of syscalls

```
{
        static int i = 0;
        printk("%s(): %d\n", __func__, i);
        i++;
}
```

Y axis: 0, 100, 200, 300, 400, 500, 600

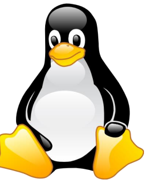
X axis: getpid(), getegid(), getgid(), geteuid(), getuid(), gettid(), getppid()

Legend: Number of syscalls

© 2020 Arm Limited (or its affiliates)

arm

vDSO Per CPU Investigation



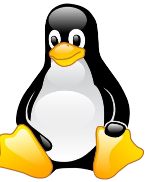
The vdso_per_cpu_data[] offset is exposed by the kernel in an architecture specific way. The user should be only able to read it.


```
#define VDSO_PAGES_SIZE PAGE_ALIGN(sizeof(vdso_data) + NR_CPUS * sizeof(vdso_per_cpu_data))
```


© 2020 Arm Limited (or its affiliates)

arm

vDSO Per THREAD Investigation



The vdso_per_thread_data[] offset is exposed by the kernel in an architecture specific way. The user should be only able to read it.

VDSO_PAGES are private to the process.

KERNEL | USERSPACE

offset

VDSO_PAGES_SIZE

vdso_data

vdso_per_thread_data[]

PAGE_SIZE

...

getpid()

...

AT_SYSINFO_EHDR

System wide NR_THREADS is bounded as [20, FUTEX_TID_MASK] (Linux 4.1). It can be modified by the user via:

/proc/sys/kernel/threads-max

The written value is checked against the occupation of the thread struct in RAM (< 1/8).

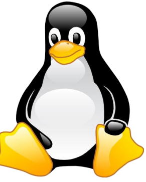
There is no per process MAX_THREADS concept.

```
#define VDSO_PAGES_SIZE PAGE_ALIGN(sizeof(vdso_data) + NR_THREADS * sizeof(vdso_per_thread_data))
```


© 2020 Arm Limited (or its affiliates)

arm

Per THREAD vs Per CPU vDSO



Per THREAD

Concept: Each process has its own private vDSO page. Can embed per thread structs for a better coverage.

PRO: Can expose process private data.

PRO: More flexible,allows to implement more system calls.

CON: Requires more memory.

CON: Implementation is more complex.

Per CPU

Concept: At a given point in time on each CPU runs a single thread.

PRO: Can reuse global vDSO pages (simpler to implement, less invasive).

PRO: Better memory impact.

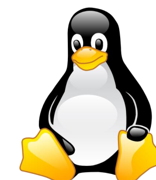
CON: Cannot expose process private data.

CON: Less flexible, allows to implement a more restricted number of system calls.

© 2020 Arm Limited (or its affiliates)

arm

Per CPU Implementation (Datapage)



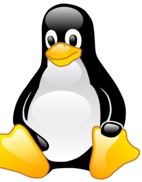
```
struct vdso_per_cpu_data {
        /* getcpu() data */
        u32 cpu;
        u32 node;
        /* getpid() data */
        s32 pid;
        /* gettid() data */
        s32 tid;
        /* getuid() data */
        u32 uid;
        /* geteuid() data */
        u32 euid;
        /* getgid() data */
        u32 gid;
        /* getegid() data */
        u32 egid;
        /* getppid() data */
        s32 ppid;
};

struct vdso_vsys_data {
        /* Note: vdso_data must be kept as the first
         * member of this structure */
        struct vdso_data data[CS_BASES];
        /* Data for per cpu information */
        struct vdso_per_cpu_data cpu_data[];
};
```

- The datapage has been extended to include vdso_per_cpu_data.
- Each CPU has its own data allocated at the offset defined as smp_processor_id() * sizeof(struct vdso_per_cpu_data).
- The number of data pages that an architecture needs reserve depends on NR_CPUS and the size of vdso_per_cpu_data.
- The way in which the offset is passed to the userspace is architecture specific.

© 2020 Arm Limited (or its affiliates)

arm

Per CPU Implementation (kernel/vdso.c)



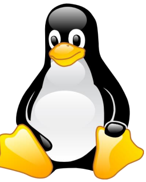
```
static __always_inline
void vdso_per_cpu_update(struct task_struct *tsk, size_t offset)
{
    struct vdso_vsys_data *data =
              (struct vdso_vsys_data *)vdso_per_cpu_data;
    unsigned int cpu;

    if (offset) {
        cpu = smp_processor_id();

        /* getpid() data update */
        data->cpu_data[cpu].pid = task_tgid_vnr(tsk);
        /* gettid() data update */
        data->cpu_data[cpu].tid = task_pid_vnr(tsk);
        /* getuid() data update */
        data->cpu_data[cpu].uid =
            from_kuid_munged(task_user_ns(tsk), task_uid(tsk));
        /* geteuid() data update */
        data->cpu_data[cpu].euid =
            from_kuid_munged(task_user_ns(tsk), task_euid(tsk));
        /* getgid() data update */
        data->cpu_data[cpu].gid =
            from_kgid_munged(task_user_ns(tsk), task_gid(tsk));
        /* getegid() data update */
        data->cpu_data[cpu].egid =
            from_kgid_munged(task_user_ns(tsk), task_gid(tsk));
        /* getppid() data update */
        data->cpu_data[cpu].ppid = vdso_getppid(tsk);
    }
}
```

- vdso_per_cpu_update() is invoked during start_thread() and switch_thread().
- Its logic is similar in behaviour to what happens in update_vsyscall().
- It currently uses only generic kernel functions, which makes the implementation of most of the newly introduced functions architecture independent.
- The porting to other architectures relies only on providing a mechanism to update/access the correct offsets and the fallbacks for the various system calls.

© 2020 Arm Limited (or its affiliates)

arm

Per CPU Implementation (vDSO data update)



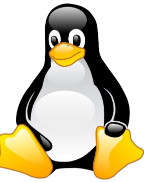
```
void vdso_per_cpu_start_thread(struct task_struct *tsk)
{
        size_t offset = vdso_per_cpu_offset();

        vdso_per_cpu_update(tsk, offset);

        arch_notify_vdso_per_cpu_offset(offset);
}

void vdso_per_cpu_switch_thread(struct task_struct *tsk)
{
        size_t offset = vdso_per_cpu_offset();

        vdso_per_cpu_update(tsk, offset);

        arch_notify_vdso_per_cpu_offset(offset);
}

void vdso_per_cpu_flush_thread(void)
{
        arch_notify_vdso_per_cpu_offset(0);
}
```

start
switch
ready
wait
sleep
run
block
dead

© 2020 Arm Limited (or its affiliates)

arm

Per CPU Implementation (lib/vdso)



```
static int __cvdso_getpid(void)
{
        struct vdso_per_cpu_data *cpu_data =
                        __arch_get_vdso_per_cpu_data();
        int pid = 0;

        if (cpu_data) {
                pid = cpu_data->pid;

                return pid;
        }

        return getpid_fallback();
}
```

- With this implementation the generic vDSO library code looks very simple:
  - It accesses the per cpu data at the correct offset.
  - Returns the value to the userspace.
  - Note: the ABI consistency with the system calls will be evaluated at a later stage, before the series will be sent out for review.
- With this logic, work has been done to implement:
  - getpid()
  - gettid()
  - getgid()
  - getegid()
  - getuid()
  - geteuid()
  - getppid()
  - getcpu()
- The implementation has been tested on arm64.

© 2020 Arm Limited (or its affiliates)

Open points and discussion

Git Repo: git://linux-arm.org/linux-vf.git vdso/lpc2020

- What is the best way to implement the offset mechanism on other architectures?
  - On arm64 we are using the TPIDRRO_EL0 register to store the offset.
- Are there syscalls that would definitely benefit from this approach and should be considered first?
- Which naming convention should we adopt for the extended library?
  - vsyscall is already taken for other purposes.
- How do we make sure that we do not duplicate the code once the extended vDSOs are in place?
- In future, where possible, should we try to go vDSO first when we introduce new syscalls?

- The prototype for arm64 is currently based on 5.8. The rebase on 5.9-rc1 requires it to work with time namespaces which rely already on multiple data pages.

© 2020 Arm Limited (or its affiliates)

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה

© 2020 Arm Limited (or its affiliates)