undo

LINUX KERNEL/USER ABI

Greg Law

C++ now

2025

How the Linux User/Kernel ABI Really Works

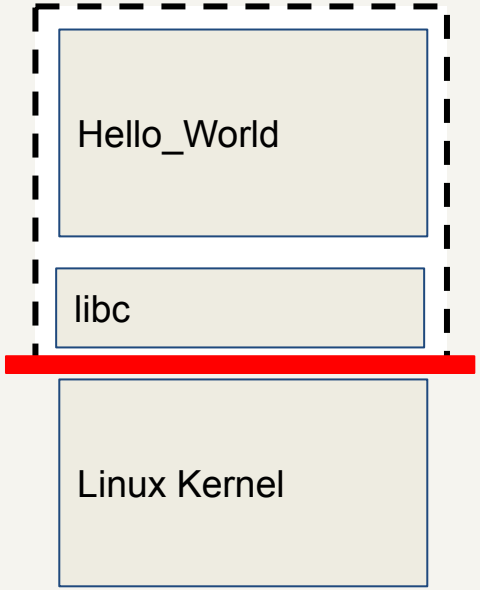
Driving the Linux Kernel Down at the Metal

Greg Law

AGENDA

- Process boundary, libc, kernel.
  - How to issue a system call?
  - What exactly is the kernel anyway?
  - User-mode, kernel mode and system calls.
- VSDO (Virtual Dynamic Shared Object)
- errno
- Signals
- ptrace
- /proc

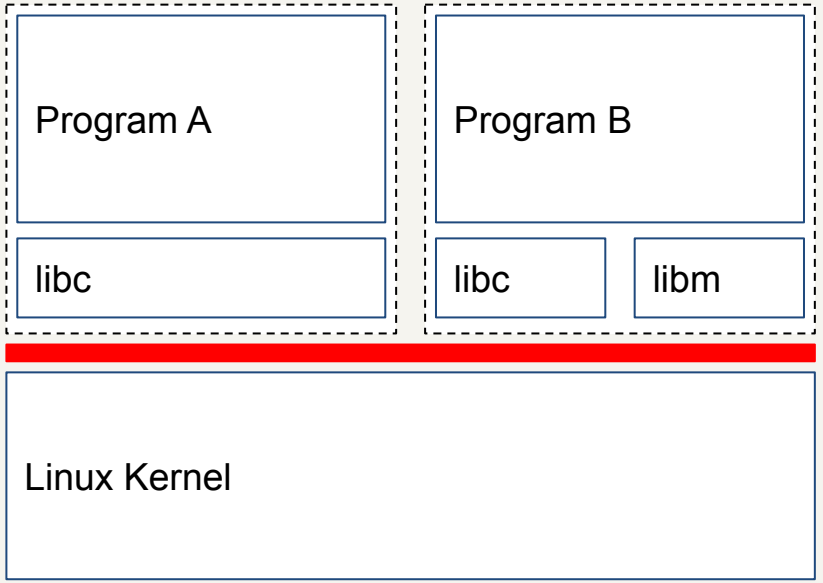
MOST PROGRAMS TALK TO LIBC (NOT THE ~~KERNEL~~)



Note:
The Linux kernel does not present POSIX API glibc does.

OPERATING SYSTEM KERNELS

- Protection
  - Program A cannot access B's data.
  - Program A cannot interfere with B's jobs.
- Security
  - Policy defining who is allowed to do what.
- Services
  - Network, file system, devices, etc



undo

USER MODE, KERNEL MODE, AND SYSTEM CALLS

Kernel mode

All instructions are available.

User mode

Certain instructions are unavailable.

Sometimes some of the virtual address space is inaccessible.

User mode code can invoke functions in the kernel via system calls

	i386	**x86-64**	**arm32**	**arm64**
Instruction	int $0x80	syscall	svc #0	svc #0
syscall No.	eax	rax	r0	x0
arg1	ebx	rdi	r1	x1
arg2	ecx	rsi	r2	x2
arg3	edx	rdx	r3	x3

undo

DIFFERENT WAYS OF DOING SYSCALLS

Any 32-bit or 64-bit x86 CPU: **int $0x80**

- Simple.
- Always available.
- Slow.

“Newer” (i686) x86 CPUs: **sysenter**

- Slightly weird (fixed return address, must use “flat address space”).
- Not always available (e.g. 80486, 32-bit mode AMD x86-64).

X86-64: **syscall**

- Simple.
- Always available.
- Fast.

32-bit user-space on AMD x86-64: **syscall**.

32-bit user-space on Intel x86-64: **sysenter**.

VDSO: VIRTUAL DYNAMIC SHARED OBJECT

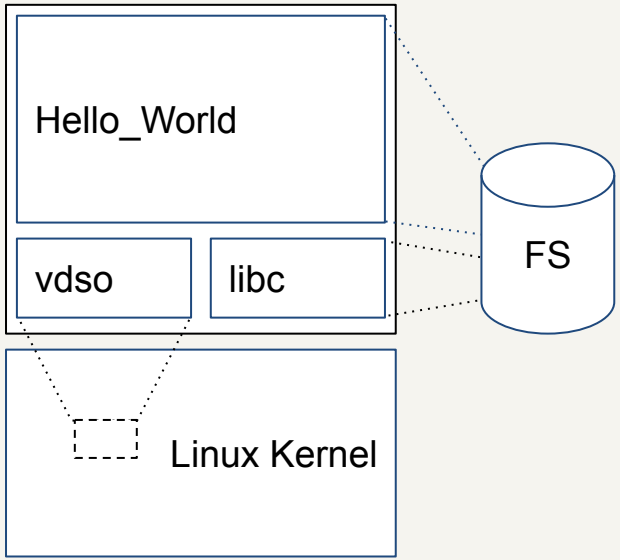
Magical library, injected by kernel into user-space process.

Allows the same binary application and even same file-system to w stems.

Beware strace!

Allows some calls to shortcut the kernel entirely.

See `/proc/sys/abi/vsyscall32` contents - set to 1 if VSDO mapped on x86-32



undo

OTHER SPECIAL MAPS

`[heap]`	– modified by `brk()`
`[vdso]`	– kernel-supplied code
`[vvar]`	– kernel-supplied data
`[stack]`	– `MAP_GROWSDOWN`
`[vsyscall]`	– legacy

`errno` IS NOT A REGULAR VARIABLE

Stored in Thread Local Storage (TLS)

A pointer to **`errno`** is returned via libc's **`__errno_location()`** function

On i686 the TLS block is referenced by **`%gs`** segment register; on x86-64 via **`%fs`**.

All syscalls return between -4095 and -1 on failure.

Even the **`syscall()`** function sets **`errno`**!

undo

PROCESSES AND THREADS

No real distinction between threads and processes - they're all tasks.

A POSIX process maps on to a Linux thread-group.

Thread-group leader is task with tid == pid.

`pthread_create()` results in a clone() with `CLONE_VM` set.

Pthread mutex built on futex.

LIBC `getpid()` CACHE?

Libc 2.3.4 to 2.24 would cache the pid.

Since libc 2.25 it no longer does.

Could we store it in vDSO/vvar? Or even in a **`MADV_WIPEONFORK`** area?

- Yes, but see Linus's rant about no sane application needing to do this.
- Also, tid is available through TLS.

/proc

States:

R	Running or runnable (on CPU or waiting to run)
S	Sleeping (interruptible sleep, waiting for an event)
D	Uninterruptible sleep (usually I/O)
Z	Zombie (process terminated, but parent hasn't collected it)
T	Stopped (by a job control signal) or Traced (being debugged) (see TracerPid)
t	A different stopped state — stopped by a debugger

SIGNALS

Old-style via the **`sigaction`** system call (do not use).

“Newer”-style **`rt_sigaction`** system call.

Note that libc rt_sigaction is a wrapper around the system call and uses different sigaction structure

```
struct sys_rt_sigaction
{
    union
    {
        k_sighandler_t k_sa_handler;
        k_sigaction_t k_sa_action;
    };
    long                sa_flags;
    void                (*sa_restorer) (void);
    uint64_t          sa_mask;
};
```

```
struct sigaction
  {
    /* Signal handler.  */
    __sighandler_t sa_handler;

    /* Additional set of signals to be blocked.
    __sigset_t sa_mask;

    /* Special flags.  */
    int sa_flags;

    /* Restore handler.  */
    void (*sa_restorer) (void);
  };
```

SIGNAL STACK

```
struct ucontext {
    unsigned long       uc_flags;
    struct ucontext     *uc_link;
    stack_t             uc_stack;
    struct sigcontext   uc_mcontext;
    sigset_t            uc_sigmask; /* mask last for extensibility */
};
```

SYSCALL INTERRUPTION AND RESTART

System calls return `-EINTR` if interrupted by a signal handler.

Except those that don't!

`futex`, `sigwait`, `sigwaitinfo`, and others.

If signal causes read or write to return short.

Signal handler wit `SA_RESTART` flag.

SYSCALLS INTERRUPTED EVEN WITH ~~SA_RESTART~~!

Kernel implements by hacking return address from signal handler.

From the kernel's **`errno.h`**:

```
/*
 * These should never be seen by user programs.  To return one of ERESTART*
 * codes, signal_pending() MUST be set.  Note that ptrace can observe these
 * at syscall exit tracing, but they will never be left for the debugged user
 * process to see.
 */
#define ERESTARTSYS             512
#define ERESTARTNOINTR          513
#define ERESTARTNOHAND          514  /* restart if no handler.. */
#define ENOIOCTLCMD             515  /* No ioctl command */
#define ERESTART_RESTARTBLOCK 516  /* restart by calling sys_restart_syscall */
```

SYSCALLS INTERRUPTED EVEN WITH ~~SA_~~RESTART

Kernel implements by hacking return address from signal handler.

From the kernel's **`errno.h`**:

```
/*
 * These should never be seen by user programs.  To return one of ERESTART*
 * codes, signal_pending() MUST be set.  Note that ptrace can observe these
 * at syscall exit tracing, but they will never be left for the debugged user
 * process to see.
 */
#define ERESTARTSYS             512
#define ERESTARTNOINTR          513
#define ERESTARTNOHAND          514  /* restart if no handler.. */
#define ENOIOCTLCMD             515  /* No ioctl command */
#define ERESTART_RESTARTBLOCK 516  /* restart by calling sys_restart_syscall */
```

```
static int
handle_signal(unsigned long sig, siginfo_t *info, struct k_sigaction *ka,
              sigset_t *oldset, struct pt_regs *regs)
{
     /* Are we from a system call? */
     if ((long)regs->orig_rax >= 0) {
          /* If so, check system call restarting.. */
          switch (regs->rax) {
               case -ERESTART_RESTARTBLOCK:
               case -ERESTARTNOHAND:
                    regs->rax = -EINTR;
                    break;
               case -ERESTARTSYS:
                    if (!(ka->sa.sa_flags & SA_RESTART)) {
                         regs->rax = -EINTR;
                         break;
                    }
                    /* fallthrough */
               case -ERESTARTNOINTR:
                    regs->rax = regs->orig_rax;
                    regs->rip -= 2;
                    break;
          }
     }
     int ret = setup_rt_frame(sig, ka, info, oldset, regs);
```

undo

ARGV, ENVP, and AUXV

```
position            content                     size (bytes) + comment
  ------------------------------------------------------------------------
  stack pointer ->  [ argc = number of args ]     4
                    [ argv[0] (pointer) ]         4   (program name)
                    [ argv[1] (pointer) ]         4
                    [ argv[..] (pointer) ]        4 * x
                    [ argv[n] (pointer) ]         4   (= NULL)

                    [ envp[0] (pointer) ]         4
                    [ envp[1] (pointer) ]         4
                    [ envp[..] (pointer) ]        4
                    [ envp[term] (pointer) ]      4   (= NULL)

                    [ auxv[0] (Elf32_auxv_t) ]    8
                    [ auxv[1] (Elf32_auxv_t) ]    8
                    [ auxv[..] (Elf32_auxv_t) ]   8
                    [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                    [ padding ]                   0 - 16

                    [ argument ASCIIZ strings ]   >= 0
                    [ environment ASCIIZ str. ]   >= 0

  (0xbffffffc)      [ end marker ]                4   (= NULL)

  (0xc0000000)      < bottom of stack >           0   (virtual)
```

Credit: Manu Garg
http://articles.manugarg.com/aboutelfauxiliaryvectors

undo

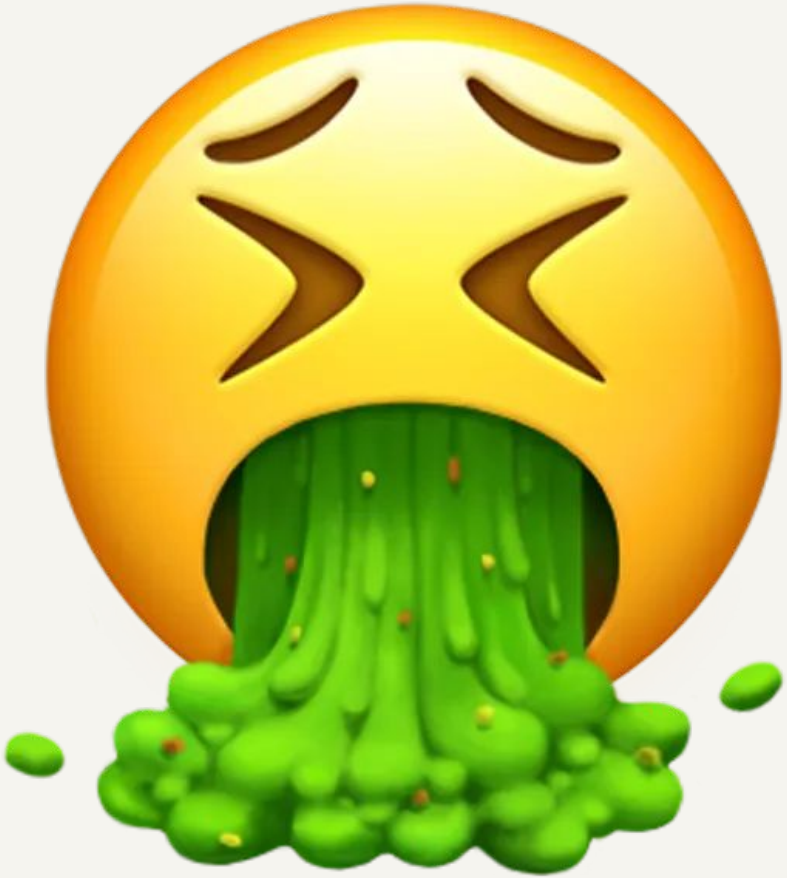
YEAH BUT WHAT ACTUALLY IS AUXV?

Just a list of key-value pairs, passed from the OS to the process.

See `man getauxval` for details.

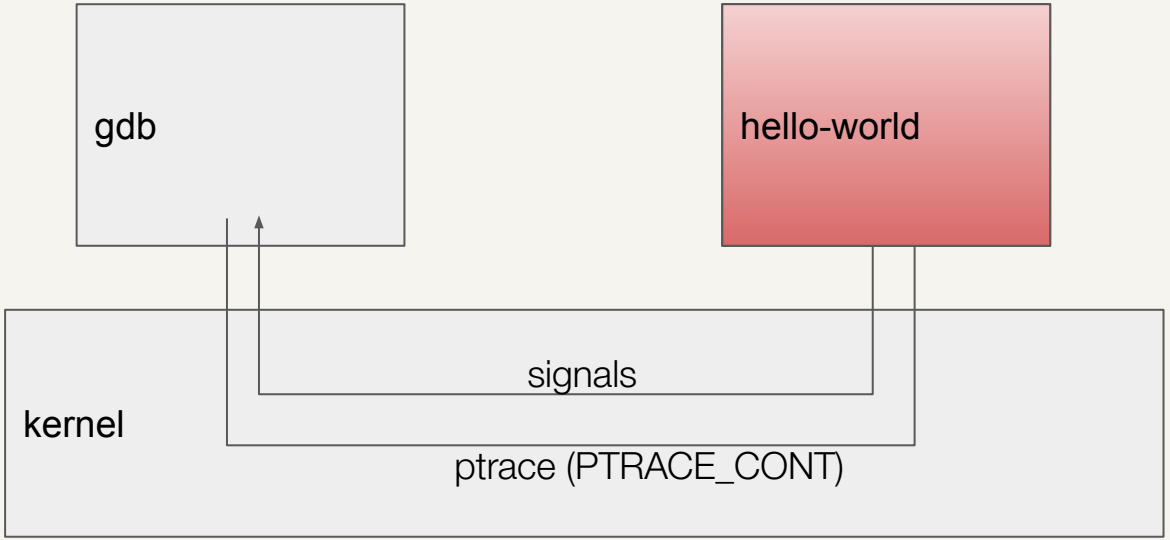
(Most of what you want probably better from /proc though, e.g. /proc/cpuinfo.)

PTRACE



undo

RUNNING A PROGRAM UNDER PTRACE



PTRACE AND SIGNALS

Signals only reach the tracee via **`PTRACE_CONT`**

e.g. **`ptrace(PTRACE_CONT, pid, NULL, (void*)SIGALRM)`**

if blocked by tracee the handler will run when the signal becomes unblocked

if ignored by tracee the signal is discarded

Breakpoints and single-step are **`SIGTRAP`**s

^C is **`SIGINT`**

SIGNAL DELIVERY ALGORITHM

```
if sig == SIGKILL: kill process
else if sig == SIGSTOP: suspend process
else if traced: suspend process (tracing stop)
else if blocked(sig): mark pending
else if ignored: ignore
else if handler: run handler
else: terminate process
```

SIGNAL DELIVERY ALGORITHM

```
if fault(sig) and (blocked(sig) or ignored(sig)):
    unblock(sig) and set_handler(sig, SIG_DFL)
if sig == SIGKILL: kill process
else if sig == SIGSTOP: suspend process
else if traced: suspend process (tracing stop)
else if blocked(sig): mark pending
else if ignored: ignore
else if handler: run handler
else: terminate process
```

PTRACE

PTRACE_CONT

PTRACE_SINGLESTEP

PTRACE_SYSCALL

PTRACE_GETREGS

PTRACE_SETREGS

PTRACE_PEEKTEXT/PTRACE_PEEKDATA

PTRACE_POKETEXT/PTRACE_POKEDATA

PTRACE_PEEKUSER

PTRACE_POKEUSER

OMG MORE PTRACE

PTRACE_GETSIGINFO / PTRACE_SETSIGINFO

PTRACE_GETSIGMASK / PTRACE_SETSIGMASK

PTRACE_PEEKSIGINFO

PTRACE_GETFPREGS / PTRACE_SETFPREGS

PTRACE_GETREGSET / PTRACE_SETREGSET

PTRACE_ATTACH / PTRACE_DETACH

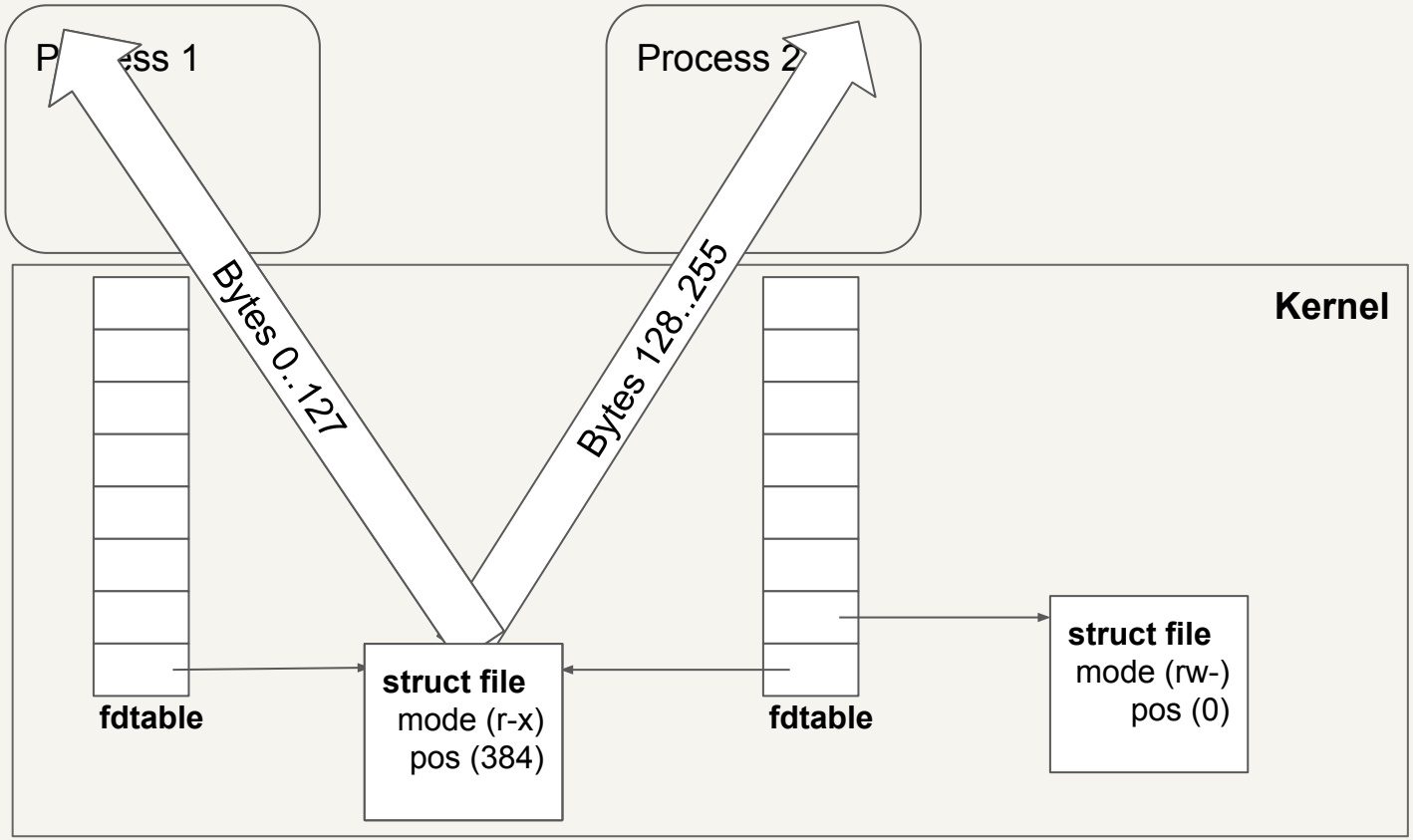
PTRACE_SETOPTIONS

PTRACE_GETEVENTMSG

PTRACE_SEIZE

PTRACE_INTERRUPT

FILE DESCRIPTOR TABLES



VISIT UNDO.IO TO LEARN MORE