crime and `strace`

follows six bytes from Python through libc, x86-64 syscall entry, linux's write path, and back

2022-11-16 · 17 min · 3999 words

https://tramasys.mov/posts/crime-and-strace/

> [!] **updated · august 2026**
>
> I revisited this post against current linux 7.1, glibc 2.44, CPython, Rust, and strace 7.0 sources. The kernel sections now follow the current x86 entry code and generated syscall dispatch.

the offense

I wanted the smallest program which still makes the whole trip into the kernel. Python's `print` is slightly too helpful. It performs text encoding, uses a buffered stream, and may postpone or combine writes.

`os.write` gives us a cleaner suspect:

```
import os

os.write(1, b"crime\n")
```

The arguments are a file descriptor, a bytes object, and an implied length. File descriptor `1` is conventionally standard output. The byte string is six bytes long, including the newline.

Running it under `strace` gives the whole public version of events:

```
$ strace -e trace=write -s 32 \
    python3 -c 'import os; os.write(1, b"crime\n")' >/dev/null
write(1, "crime\n", 6)                 = 6
+++ exited with 0 +++
```

The `6` on the right is not a second copy of the argument. It is the return value. `write(2)` reports how many bytes it accepted, and that may be less than the requested count. Six bytes went in and six came back on this run.

The compact route is:

```
Python os.write
  -> CPython _Py_write
  -> glibc write
  -> x86-64 syscall instruction
  -> linux entry_SYSCALL_64
  -> do_syscall_64
  -> __x64_sys_write
  -> ksys_write
  -> vfs_write
  -> the file operation behind fd 1
  -> return through SYSRETQ or IRETQ
```

`strace` watches from the side. It is not one of the arrows in the untraced route, although tracing adds stops to that route later.

through cpython

The public `os.write` method is generated by Argument Clinic, but its current implementation in `Modules/posixmodule.c` is pleasantly small:

```c
static Py_ssize_t
os_write_impl(PyObject *module, int fd, Py_buffer *data)
{
    return _Py_write(fd, data->buf, data->len);
}
```

Argument Clinic has already converted the Python integer to a C `int` and acquired a `Py_buffer` view of the bytes object. `data->buf` is now an address in the Python process, and `data->len` is `6`.

`_Py_write` leads to `_Py_write_impl`. With the Windows branches removed, the important part looks like this:

```c
do {
    Py_BEGIN_ALLOW_THREADS
    errno = 0;
    n = write(fd, buf, count);
    err = errno;
    Py_END_ALLOW_THREADS
} while (n < 0 && err == EINTR &&
         !(async_err = PyErr_CheckSignals()));
```

CPython releases the GIL because a write can block. It also retries after `EINTR` unless the Python signal handler raised an exception. None of this is the syscall yet. The lowercase `write` in the middle is the C library function.

The call also preserves the short-write behavior. `os.write` returns whatever count libc returned, so a caller which needs to send the whole buffer must loop. Our six bytes happen to fit almost everywhere.

rust takes a slightly longer route

The equivalent ordinary Rust program is:

```rust
use std::io::{self, Write};

fn main() -> io::Result<()> {
    io::stdout().write_all(b"crime\n")
}
```

This time `write_all` promises to keep trying after short writes. Standard output is protected by a reentrant lock and wrapped in a `LineWriter`. The newline flushes that line, so the experiment still produces one call:

```
$ strace -e trace=write ./crime-rs >/dev/null
write(1, "crime\n", 6)                 = 6
+++ exited with 0 +++
```

The buffering machinery lives in `std::io::stdio`. At the bottom, the Unix-specific `Stdout` constructs a `FileDesc` for `STDOUT_FILENO`, and `FileDesc::write` does this:

```rust
let ret = cvt(unsafe {
    libc::write(
        self.as_raw_fd(),
        buf.as_ptr() as *const libc::c_void,
        cmp::min(buf.len(), READ_LIMIT),
    )
})?;
```

The Python and Rust policies differ above this point. Their final Unix call does not. Both enter glibc with the normal C function-call convention.

libc's last words

The current glibc source defines `write` around one cancellation-aware macro:

```
ssize_t
__libc_write (int fd, const void *buf, size_t nbytes)
{
  return SYSCALL_CANCEL (write, fd, buf, nbytes);
}
```

POSIX makes `write` a thread cancellation point. In a multithreaded program, glibc may therefore do cancellation bookkeeping around the kernel call. The installed glibc 2.44 binary on this machine has that machinery in `__write`, but its direct path eventually becomes:

```
mov    eax, 1
syscall
cmp    rax, -4096
ja     error
```

The first instruction selects syscall number `1`. The arguments did not need to move because a three-argument C function and a three-argument x86-64 linux syscall both receive them in `rdi`, `rsi`, and `rdx`.

The comparison handles linux's error convention. Inside the kernel, failures are negative values such as `-EBADF` or `-EFAULT`. glibc recognizes the range `-4095` through `-1`, negates the value into thread-local `errno`, and returns `-1` to C. A successful byte

count passes through unchanged.

For syscalls with four or more arguments, glibc has one extra job. The normal System V function ABI puts argument four in `rcx`, while the syscall ABI puts it in `r10`. The x86-64 `DO_CALL` macro moves `rcx` to `r10` before `syscall`. `write` only has three arguments, so we avoid that piece of paperwork.

removing libc

The library is useful, but not required. A complete static program can issue the call directly:

```
.intel_syntax noprefix
.global _start

.section .rodata
message:
    .ascii "crime\n"
.set message_len, . - message

.section .text
_start:
    mov eax, 1
    mov edi, 1
    lea rsi, [rip + message]
    mov edx, message_len
    syscall

    xor edi, edi
    mov eax, 60
    syscall
```

This is GNU assembler syntax in Intel mode. Building it without a C runtime leaves exactly two system calls:

```
$ cc -nostdlib -static crime.s -o crime
$ strace -e trace=write,exit ./crime >/dev/null
write(1, "crime\n", 6)                  = 6
exit(0)                                 = ?
+++ exited with 0 +++
```

The second call terminates the process. A raw `_start` cannot return to a caller because there is no caller.

Immediately before the first `syscall`, the registers mean:

register	value	meaning
`rax`	`1`	syscall number for `write`
`rdi`	`1`	file descriptor
`rsi`	address	first byte of `"crime\n"`
`rdx`	`6`	byte count
`r10`	unused	fourth argument
`r8`	unused	fifth argument
`r9`	unused	sixth argument

The table in `arch/x86/entry/syscalls/syscall_64.tbl` is the source of the first number:

```
1    common    write    sys_write
60   common    exit     sys_exit
```

`syscall` is not a function call. It has no encoded target address and does not look at the Procedure Linkage Table. The CPU already knows where the kernel asked it to go.

GDB makes the boundary visible, but it cannot follow the instruction like an ordinary call:

```
(gdb) set disassembly-flavor intel
(gdb) break *0x401016
(gdb) run
(gdb) info registers rax rdi rsi rdx rcx r11
(gdb) stepi
0x0000000000401018 in _start ()
```

One `stepi` appears to jump from the `syscall` instruction to the following userspace instruction. The CPU did execute the kernel path in between. A userspace debugger gets control again when the traced task is ready to resume in userspace. Stepping through `entry_SYSCALL_64` itself needs kernel-aware debugging, commonly a VM under QEMU with GDB attached to the guest kernel.

how the cpu knows where linux lives

During boot, linux programs model-specific registers on every CPU. On the traditional x86-64 entry path, current `syscall_init` eventually does the equivalent of:

```
wrmsrq(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
wrmsrq(MSR_SYSCALL_MASK, flags_to_clear);
wrmsr(MSR_STAR, 0, user_and_kernel_code_selectors);
```

These MSRs are privileged and local to each logical CPU. Userspace cannot redirect its own next syscall by writing a new `LSTAR` value.

`IA32_LSTAR` contains the virtual address of the 64-bit entry point. `IA32_STAR` supplies the code-segment selectors used for the privilege transition and return. Segments are mostly flat in 64-bit mode, but the privilege bits in `CS` still determine the current privilege level. `IA32_FMASK` says which `RFLAGS` bits should be cleared in the live flags on entry.

linux includes `IF`, `TF`, `DF`, and `AC` in that mask. Clearing `IF` keeps ordinary maskable interrupts out until the entry code has a safe stack. Clearing `TF` prevents userspace single-step state from walking into the entry trampoline. Clearing `DF` gives C code its expected forward string direction. Clearing `AC` makes sure userspace cannot carry its SMAP override into ring 0. The original flags are still preserved in `r11` for the return.

The `SYSCALL` instruction itself performs a small, fixed set of operations:

- it saves the address of the following userspace instruction in `rcx`
- it saves userspace `RFLAGS` in `r11`
- it loads the kernel instruction pointer from `IA32_LSTAR`
- it loads ring 0 `CS` and `SS` values derived from `IA32_STAR`
- it masks flags using `IA32_FMASK`

That changes the current privilege level from ring 3 to ring 0. It is not a scheduler context switch. We are still the same task on the same logical CPU. No scheduler ran, and no register file belonging to another task was loaded.

This also explains two odd pieces of the syscall ABI. `rcx` cannot carry argument four because the CPU needs it for the return address, so linux uses `r10` instead. Both `rcx` and `r11` are declared clobbered around inline syscall assembly because hardware overwrites them even when the kernel handler does nothing.

There is an important omission: `SYSCALL` does not save anything on a stack, and it does not change `rsp`. For the first few kernel instructions, `rsp` still points at the untrusted userspace stack. linux has to arrange the real kernel stack itself. Unlike an interrupt or exception which changes privilege, this path does not ask the task-state segment to load its ring 0 stack.

That omission is intentional. `SYSCALL` and `SYSRETQ` were designed as a small, programmable fast path. Hardware performs the minimum privilege transition and leaves operating-system policy, including the stack layout, to software.

The split between work done by the CPU and work done by linux is easier to see in one picture. This is the traditional non-FRED path used on the machine from these notes:

```
+-------------------------- userspace / ring 3 -----------------+
| CPython os.write(1, b"crime\n")                               |
|             |                                                 |
|             v                                                 |
| libc write(fd=1, buf, 6)                                      |
| rax=1  rdi=1  rsi=buf  rdx=6  rsp=user stack                  |
+-------------|-------------------------------------------------+
              |  0f 05
              v
+------------------------ CPU / SYSCALL ----------------------+
| rcx <- next user rip       r11 <- user rflags                 |
| rip <- IA32_LSTAR          rflags &= ~IA32_FMASK              |
| CS/SS <- IA32_STAR         privilege: ring 3 -> ring 0        |
|                                                               |
| rsp is unchanged. It still points at the userspace stack.     |
+-------------|-------------------------------------------------+
              v
+--------------------------- linux / ring 0 --------------------+
| entry_SYSCALL_64                                              |
|   -> swapgs                  select kernel per-CPU data       |
|   -> save user rsp             TSS scratch field              |
|   -> switch CR3                kernel page tables, if KPTI    |
|   -> load task kernel stack                                   |
|   -> build struct pt_regs                                     |
|   -> do_syscall_64                                            |
|        -> __x64_sys_write                                     |
|             -> ksys_write -> vfs_write -> file operation -> TTY |
|                                                               |
| result: rax=6                                                 |
|                                                               |
| exit preparation                                              |
|   -> validate saved rip and rflags                            |
|   -> restore most registers                                   |
|   -> switch to trampoline stack                               |
|   -> switch to user CR3                                       |
|   -> restore rdi and user rsp                                 |
|   -> swapgs                                                   |
+-------------|-------------------------------------------------+
              v
+------------------------ CPU / SYSRETQ ----------------------+
| rip <- rcx       rflags <- r11       privilege: ring 0 -> 3   |
+-------------|-------------------------------------------------+
              v
+-------------------------- userspace / ring 3 -----------------+
| libc returns 6 -> CPython returns 6                           |
+---------------------------------------------------------------+
```

The CPU changes privilege at `SYSCALL`. linux changes page tables and stacks several instructions later. Those are separate operations, which is the part most short syscall diagrams quietly skip.

The exact architectural rules are in volume 2 of the `Intel 64 and IA-32 manuals` under `SYSCALL`. AMD64 implements the same linux-facing contract.

linux catches the instruction

`IA32_LSTAR` points at `entry_SYSCALL_64`. The beginning of the current linux 7.1 path is:

```
swapgs
mov    QWORD PTR gs:[tss_sp2], rsp
SWITCH_TO_KERNEL_CR3 scratch_reg=rsp
mov    rsp, QWORD PTR gs:[current_top_of_stack]
```

The source uses AT&T syntax. I have written the excerpts here in Intel syntax to match the userspace disassembly.

`swapgs` exchanges the bases held in `IA32_GS_BASE` and `IA32_KERNEL_GS_BASE`. In ring 3, the active base may belong entirely to the program. After `swapgs`, a `gs:` memory operand reaches linux's per-CPU area. The instruction has to be paired exactly once on entry and once on exit. Doing it twice would put the userspace base back while still in the kernel.

The syscall path uses that newly available per-CPU memory immediately. It stores the userspace `rsp` in the `sp2` field of the per-CPU TSS. The CPU did not perform a TSS stack switch here. linux is simply borrowing a reliably mapped field as scratch space before the task's kernel stack is available.

`SWITCH_TO_KERNEL_CR3` changes page tables when Kernel Page Table Isolation is active. The userspace page table contains the process mappings and only the small CPU entry area needed to get in and out. The kernel copy contains the full kernel mapping as well as the userspace mappings needed by guarded access helpers such as `copy_from_user`. The entry trampoline therefore has enough code and per-CPU data mapped to reach the CR3 switch, but not the rest of the kernel.

Writing `CR3` selects a different top-level page-table root in the MMU. That normally disturbs cached translations in the TLB. When the CPU supports PCID, linux can tag translations from the user and kernel views and avoid flushing everything on every crossing. It does not make the switch free, but it removes some of the original KPTI cost. The macro is patched or compiled down according to the CPU, boot options, and kernel configuration. The `PTI documentation` describes the two page-table copies and the shared entry area.

Only then does linux load the top of this task's kernel stack into `rsp`. Each thread has a kernel stack which userspace cannot address. Calls, local variables, interrupts, and saved state below this point belong there.

At this point ordinary interrupts are still disabled because `IF` was removed through `IA32_FMASK`. That closes the window in which an interrupt might try to use a half-constructed frame. NMIs cannot be masked this way, which is one reason x86 entry code contains considerably more defensive machinery than the happy path shown here.

making the frame hardware did not make

linux next constructs a `struct pt_regs` on that stack:

```
push   __USER_DS
push   QWORD PTR gs:[tss_sp2]  # userspace rsp
push   r11                     # userspace rflags
push   __USER_CS
push   rcx                     # userspace rip
push   rax                     # original syscall number
PUSH_AND_CLEAR_REGS rax=-ENOSYS
```

The remaining macro saves the general-purpose registers and clears selected state. It also gives the return-value slot an initial `-ENOSYS`. If dispatch does not find a syscall, "not implemented" is therefore already waiting.

The saved object has everything needed by tracing, signals, seccomp, audit, the syscall wrapper, and eventually the return to userspace. The entry code then passes two C arguments:

```
mov    rdi, rsp       # struct pt_regs *
movsxd rsi, eax       # syscall number
call   do_syscall_64
```

Current source also places `IBRS_ENTER`, return-branch untraining, and branch history clearing between the frame construction and that call. These are Spectre-era speculation mitigations selected through linux's alternatives machinery. They are not part of the definition of a syscall, and the emitted instructions depend on the CPU and boot configuration. They are now part of the practical cost on affected systems.

the newer hardware footnote

linux 7.1 also supports Intel FRED, Flexible Return and Event Delivery. On a machine with FRED enabled, `SYSCALL` arrives through a common FRED userspace entry, hardware supplies a richer event frame, and linux returns with `ERETU`. The current `FRED_entry_code` and `fred_other_dispatch` still lead to `do_syscall_64`, so the `write` investigation below converges again.

The machine used for these notes does not advertise FRED, so `entry_SYSCALL_64` and `SYSRETQ` are the path I actually disassembled. FRED is worth mentioning because "every current x86-64 syscall lands at LSTAR" has stopped being universally true.

dispatch, current edition

Many descriptions now say that linux uses the syscall number as an index into `sys_call_table` and calls the resulting function pointer. That was a useful model, but it is stale for the current native x86 path.

linux 7.1 says this directly in `arch/x86/entry/syscall_64.c`:

```
/*
 * The sys_call_table[] is no longer used for system calls, but
 * kernel/trace/trace_syscalls.c still wants to know the system
 * call address.
 */

#define __SYSCALL(nr, sym) case nr: return __x64_##sym(regs);

long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
    switch (nr) {
    #include <asm/syscalls_64.h>
    default: return __x64_sys_ni_syscall(regs);
    }
}
```

The generated include turns syscall number `1` into a switch case which calls `__x64_sys_write(regs)`. The compiler can generate direct calls instead of an indirect function-pointer dispatch. A `sys_call_table` is still emitted for tracing metadata, which is why finding that symbol does not prove that the hot path uses it.

Before the switch, `do_syscall_64` calls `syscall_enter_from_user_mode`. That is where entry work such as ptrace, seccomp, audit, and syscall user dispatch can inspect, rewrite, skip, or reject the request. It then bounds-checks the number and applies `array_index_nospec` before calling `x64_sys_call`.

The name `__x64_sys_write` does not appear as an ordinary function in `fs/read_write.c`. It is generated by the x86 `SYSCALL_DEFINE` macros. For a native 64-bit call, the wrapper receives only `struct pt_regs *` and decodes:

```
fd    = regs->di;
buf   = regs->si;
count = regs->dx;
```

It type-checks and converts those values before reaching the body written in the source as `SYSCALL_DEFINE3(write, ...)`.

inside write

The public handler in current `fs/read_write.c` is small:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
    return ksys_write(fd, buf, count);
}
```

`__user` is a sparse annotation, not a C address-space feature. It warns kernel developers and static analysis that `buf` came from an untrusted userspace address. The kernel must not simply dereference it.

`ksys_write` resolves the descriptor in the calling process's file table:

```
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
    CLASS(fd_pos, f)(fd);
    ssize_t ret = -EBADF;

    if (!fd_empty(f)) {
        loff_t pos, *ppos = file_ppos(fd_file(f));
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(fd_file(f), buf, count, ppos);
        if (ret >= 0 && ppos)
            fd_file(f)->f_pos = pos;
    }
    return ret;
}
```

The integer `1` becomes a referenced `struct file`. That object decides what "standard output" actually means. It might be a regular file, a pipe, a socket, `/dev/null`, or a terminal. The syscall does not contain a special case for words appearing on a screen.

`vfs_write` checks that the file is writable, validates the userspace address range with `access_ok`, applies limits and permissions, and chooses the file's operation:

```
if (file->f_op->write)
    ret = file->f_op->write(file, buf, count, pos);
else if (file->f_op->write_iter)
    ret = new_sync_write(file, buf, count, pos);
else
    ret = -EINVAL;
```

For a modern `write_iter` implementation, `new_sync_write` wraps the userspace address in an `iov_iter` and calls the operation supplied by the filesystem or device. This is the point where one generic syscall turns into many possible I/O paths.

where the six bytes actually go

In an interactive terminal, this usually shows a pseudoterminal slave:

```
$ readlink /proc/$$/fd/1
/dev/pts/3
```

For that case, the current `tty_write` path receives the `iov_iter`. `iterate_tty_write` finally crosses the memory boundary:

```
if (copy_from_iter(tty->write_buf, size, from) != size)
    break;

ret = ld->ops->write(tty, file, tty->write_buf, size);
```

`copy_from_iter` uses the architecture's guarded user-copy machinery. A bad pointer becomes `-EFAULT` instead of letting a kernel page fault crash the machine. On x86, hardened and accelerated copy implementations may be chosen according to CPU features and kernel configuration.

The normal `N_TTY` line discipline may process output settings such as turning newline into carriage-return plus newline. The PTY driver then places the bytes in the master side's buffer. A terminal emulator in userspace reads that master descriptor and draws glyphs in a window. The kernel got the bytes to a queue. It did not choose a font.

Redirecting the same program changes everything after `vfs_write`:

```
$ ./crime >evidence.txt
```

Now descriptor `1` names a regular file. Its filesystem's `write_iter` implementation may update the page cache, allocate blocks, journal metadata, or submit I/O. The syscall entry and dispatch are identical. Following the bytes all the way to a storage controller would be a separate post and a much larger crime scene.

returning the count

Once the selected file operation reports success, the value `6` travels back through `vfs_write`, `ksys_write`, and `__x64_sys_write`. Dispatch stores it in `regs->ax`.

`syscall_exit_to_user_mode` performs pending work before the return. That may include ptrace's syscall-exit stop, signal delivery, rescheduling, audit work, and restart handling. A system call can therefore return through a signal handler or after another task has run, even though none of that happened in our small example.

For a clean 64-bit register frame, `do_syscall_64` permits the fast return. The assembly first restores the general-purpose registers from `pt_regs`, apart from `rdi` and `rsp`. It cannot simply switch to the userspace page table while continuing to use the ordinary kernel stack. Under KPTI that stack will no longer be mapped.

Instead, linux moves the final values onto the per-CPU trampoline stack in the CPU entry area. That small area is mapped in both page-table views. From there it can switch to the userspace `CR3`, restore `rdi`, and finally restore the userspace `rsp`. After that point the kernel must avoid normal stack accesses. The last instructions are:

```
swapgs
CLEAR_CPU_BUFFERS
sysretq
```

`SYSRETQ` uses `rcx` as the userspace instruction pointer and `r11` as the userspace flags. Like `SYSCALL`, it does not load `rsp`. linux already restored that by hand. Hardware derives the userspace code and stack selectors from `IA32_STAR`, changes the privilege level back to ring 3, and continues at the instruction after the original `SYSCALL`.

The fast instruction has sharp edges, so linux only chooses it after checking the software frame. `rcx` must still match the requested return `rip`, `r11` must match the requested flags, and `CS` and `SS` must be the normal 64-bit userspace selectors. The target must lie below `TASK_SIZE_MAX`, and the resume and trap flags must be clear. Signals, ptrace, or seccomp may have changed any of these values while the syscall was running.

The address check is not cosmetic. On Intel CPUs, `SYSRETQ` with a non-canonical `rcx` raises a general-protection fault while the CPU is still at ring 0 and the userspace stack pointer is already live. The slower `IRETQ` path consumes a complete interrupt frame and handles unusual state safely.

`CLEAR_CPU_BUFFERS` is another alternatives-site mitigation. On CPUs which need it, linux clears vulnerable microarchitectural buffers before returning to a less privileged context. Like the branch mitigations on entry, the macro may become different instructions or nothing at all on a particular machine.

Back in glibc, `rax = 6` is not an error. It returns to `_Py_write_impl`, CPython reacquires the GIL, and `os.write` creates the Python integer `6`. The userspace instruction after `syscall` continues as if a very unusual function call had just returned.

what strace did to the scene

`strace` is another userspace process. It cannot see privileged register state by staring harder. It asks the kernel to trace the target with `ptrace`.

When `strace` launches a program, the child stops before `execve` and the parent establishes the tracing relationship. When attaching to an existing process it uses the attach or seize forms instead. The main ingredients are then:

1. set `PTRACE_O_TRACESYSGOOD` so syscall stops are distinguishable from an ordinary `SIGTRAP`
2. resume the tracee with `PTRACE_SYSCALL`
3. wait for the kernel to stop it at syscall entry
4. read the number and arguments, then resume it with `PTRACE_SYSCALL` again
5. wait for the syscall-exit stop and read the return value

The option turns the stop signal into `SIGTRAP | 0x80`. Current `strace.c` defines exactly that value and uses `PTRACE_SYSCALL` as its normal restart operation.

On a recent kernel, strace can request a `struct ptrace_syscall_info` with `PTRACE_GET_SYSCALL_INFO`. The entry record includes the syscall number and six argument slots. Its `x86 fallback` reads the register set and decodes the same ABI we used above:

```
tcp->u_arg[0] = x86_64_regs.rdi;
tcp->u_arg[1] = x86_64_regs.rsi;
tcp->u_arg[2] = x86_64_regs.rdx;
tcp->u_arg[3] = x86_64_regs.r10;
tcp->u_arg[4] = x86_64_regs.r8;
tcp->u_arg[5] = x86_64_regs.r9;
```

That gives strace `1`, a pointer, and `6`. The text `"crime\n"` is not stored in a register, so strace must read memory from the stopped process. Current `ucopy.c` prefers `process_vm_readv` and falls back to repeated `PTRACE_PEEKDATA` operations. It then escapes the bytes and prints them as a C string.

At the exit stop, `rax` contains `6`. strace appends `= 6`, writes the line to its own output, and resumes the tracee. Its own log output naturally requires more syscalls, but the default trace only reports calls made by the tracee.

This also explains the cost. A normally fast syscall now stops twice and wakes the tracer twice. `strace` is excellent for investigation and a poor benchmarking environment.

The observer can also touch the evidence. strace's injection support can replace the result before the tracee continues:

```
$ strace -e trace=write -e inject=write:error=EIO:when=1 ./crime
write(1, "crime\n", 6) = -1 EIO (Input/output error) (INJECTED)
+++ exited with 0 +++
```

The raw assembly program ignores `rax` and exits successfully, which is its own bug. CPython would turn the injected error into an `OSError`. ptrace is an active control interface, even when strace normally uses it only to observe.

The entry stop happens after the CPU has already entered the kernel. ptrace work in `syscall_enter_from_user_mode` notices the trace flag and reports the stop before `__x64_sys_write` runs. The exit stop happens during the return work after the handler has produced its result. strace does not trap the userspace `syscall` opcode directly.

It also cannot show an operation which never becomes a syscall. A successful `clock_gettime` through the vDSO, for example, stays in userspace and is invisible to a syscall tracer. That is the useful negative evidence from the vDSO investigation.

back at the top

The final path for this run was:

```
os.write(1, b"crime\n")
  CPython turns the bytes object into pointer + length
  glibc prepares cancellation and error handling
  rax=1, rdi=1, rsi=buffer, rdx=6
  SYSCALL changes privilege and jumps to linux entry code
  linux switches page tables and stacks, then builds pt_regs
  generated x86 dispatch selects __x64_sys_write
  ksys_write resolves fd 1 and vfs_write selects its operation
  the endpoint copies or consumes the six userspace bytes
  rax=6 returns through SYSRETQ
  glibc and CPython turn it back into a normal return value
```

The one-line strace record is a very good summary. It is just compressed enough to hide the CPU contract, two stacks, optional page-table switch, generated wrapper, VFS dispatch, guarded memory copy, and two ptrace stops underneath it.

source trail

- CPython `os.write` implementation
- CPython `_Py_write` implementation
- Rust standard output buffering
- Rust Unix `FileDesc::write`
- glibc linux `write` wrapper
- glibc x86-64 syscall ABI macros
- linux 7.1 x86-64 syscall table
- linux 7.1 syscall MSR setup
- linux 7.1 x86-64 entry assembly
- linux 7.1 x86 entry notes
- linux 7.1 page-table isolation notes
- linux 7.1 generated syscall dispatch
- linux 7.1 x86 syscall wrappers
- linux 7.1 `write` and VFS path
- linux 7.1 TTY write path
- strace 7.0 main tracing loop
- strace 7.0 syscall decoding
- strace 7.0 tracee memory copying
- Intel 64 and IA-32 software developer manuals