

Lecture 9

System Calls
the only door into the kernel

Md Mahbub E Noor

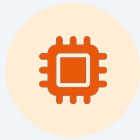
Assistant Professor, Department of CSE
University of Barishal

```
● ● ● strace ./rawhello (real)
execve("./rawhello", ...) = 0
write(1, "hello from the kernel,
        no libc\n", 31) = 31
exit(0)                               = ?
# no C library at all:
# 2 system calls, by hand
```

What you will learn today



Tell a library function from a system call, and predict which C calls enter the kernel.



Make a system call by hand: the syscall instruction and its registers.



Follow a system call through the kernel: entry, dispatch, SYSCALL_DEFINE, return.



Explain errno, the cost of a system call, and how the vDSO avoids it.

Today's plan (60 min)

0-5

Recap of 8.2

5-15

Library or kernel?

15-27

The syscall instruction

27-42

Inside the kernel

42-50

Errors and errno

50-56

Cost and vDSO

56-60

Checkpoint

Lab 9: a C program with no C library.

Quick recap of Lecture 8.2

Q1

nice 0 and nice 5 share one CPU. Who gets what?

1024 : 335, so 75 % and 25 %. Measured: 75.4 % and 24.6 % (Homework 8.2 had nice 3: 66.1 %).

Q2

Where does the auxiliary vector start on a new stack?

Right after the NULL that ends envp. AT_SYSINFO_EHDR in it points at the vDSO.

Q3

/proc/self/maps shows [vdso]. No file holds it. Who put it there?

The kernel, at execve (binfmt_elf.c 1237). Today: what it is for.

PART 1

Library function or system call?

printf is not a system call. Which C calls really enter the kernel?

A system call is a request; a library function is just code

your code

```
printf("n=%d\n", n);
```

C library (libc)

formats the text into a buffer; later calls write()

the door

the syscall instruction, number 1 in rax

the kernel

sys_write → the terminal driver

user mode, ring 3

kernel mode, ring 0

man section 3: library

printf, fopen, fread

malloc, free

strlen, qsort, sin

time, gettimeofday

system, popen

man section 2: system call

write, open, read

brk, mmap, munmap

(none: pure computation)

(none: the vDSO, Part 5)

fork, execve, wait4

SAME AS C

Looks like a C function

write(1, buf, n) is a normal C function in libc: same call, same return value. Inside, it is three or four instructions around `syscall`.

DIFFERENT FROM C

But it leaves your program

A C function runs your code with your rights. A system call runs **kernel** code with **all** rights, so the kernel checks every argument.

Book: Kerrisk, TLPI §3.1; Love, LKD Ch. 5

Quiz: which of these lines enter the kernel?

● ● ● libsys.c (main, shortened)

```
size_t n = strlen("hello");      /* 1 */
char *p = malloc(100);           /* 2 */
char *q = malloc(200000);        /* 3 */
time_t t = time(NULL);           /* 4 */
pid_t me = getpid();             /* 5 */
printf("n=%zu me=%d ...", ...);  /* 6 */
printf("second line\n");         /* 7 */
free(q);                         /* 8 */
free(p);                         /* 9 */
```

Pure computation: strlen, the second free: no kernel at all.

libc decides: malloc asks for memory in big pieces (brk; mmap above 128 KiB). printf buffers, then one write per line on a terminal.

● ● ● strace ./libsys (on a terminal, real)

```
# 1 strlen:  nothing
getrandom(..., 8, GRND_NONBLOCK) = 8      # 2
brk(NULL)                        = 0x564c5da38000
brk(0x564c5da59000)              = 0x564c5da59000
mmap(NULL, 200704, ...)          = 0x7f0260132000 # 3
# 4 time:  nothing!
getpid()                         = 361      # 5
fstat(1, {st_mode=S_IFCHR|0600, ...}) = 0      # 6
write(1, "n=5 me=361 t>0:1\n", 17) = 17
write(1, "second line\n", 12)     = 12      # 7
munmap(0x7f0260132000, 200704)    = 0        # 8
# 9 free(p): nothing
```

Surprise: time() made no system call: the vDSO (Part 5).

strace: watching the door

```
● ● ● strace -c -S calls ls -l /usr/bin (real, top lines)
```

% time	seconds	usecs/call	calls	errors	syscall
28.59	0.015423	10	1468	1468	lgetxattr
40.93	0.022080	15	1468		statx
16.13	0.008703	9	912		listxattr
13.21	0.007127	12	556		readlink
0.50	0.000272	10	26		write
...					
100.00	0.053948	11	4552	1477	total

1467 files, 4552 system calls: about 3 per file (statx for the size and date, lgetxattr and listxattr for security labels). Every one crossed the door.

Option	What it does
-C	count calls and time per call
-f	follow children (fork, clone)
-e trace=write,openat	only these calls
-T / -tt	time in each call / time stamps
-o file	write the trace to a file

How strace works: it uses another system call, `ptrace`. The kernel stops the traced process at every system call entry and exit and lets strace read its registers: the number in rax, the arguments, the result.

PART 2

The syscall instruction

One instruction, six registers, one number.

The register contract (x86-64 Linux)

	system call (syscall)	C function call (call)
what to run	rax = the number (1 = write)	the address of the function
argument 1, 2, 3	rdi, rsi, rdx	rdi, rsi, rdx
argument 4	r10	rcx
argument 5, 6	r8, r9	r8, r9 (more on the stack)
result	rax (-4095 ... -1 = error)	rax
destroyed	rcx, r11 (by the CPU)	rax, rcx, rdx, rsi, rdi, r8-r11

Why r10, not rcx? The syscall instruction itself writes the return address into rcx and the flags into r11. So argument 4 must travel in another register, and the kernel moves it back to rcx before calling C code.

● ● ● entry_64.S, Linux 6.8 (real)

```
68 * Registers on entry:
69 * rax  system call number
70 * rcx  return address
71 * r11  saved rflags
72 * rdi  arg0
73 * rsi  arg1
74 * rdx  arg2
75 * r10  arg3 (needs to be moved
           to rcx to conform to C ABI)
```

C link: a system call is a function call by **number**, not by address. You cannot jump into the kernel's code; you can only ask for entry number 1.

Book: Bryant & O'Hallaron, CS:APP §8.1.3 (Linux/x86-64 system calls)

Hello world with two system calls and nothing else

```
● ● ● rawhello.s (gcc -nostdlib -static -o rawhello rawhello.s)
```

```
_start:
    mov    $1, %rax          # number 1 = write
    mov    $1, %rdi          # fd 1: the screen
    lea    msg(%rip), %rsi   # address of the bytes
    mov    $len, %rdx        # how many bytes
    syscall                 # enter the kernel
    mov    $60, %rax         # number 60 = exit
    mov    $0, %rdi          # exit status 0
    syscall                 # never returns
msg: .ascii "hello from the kernel, no libc\n"
```

```
● ● ● ./rawhello ; strace ./rawhello (real)
```

```
hello from the kernel, no libc
execve("./rawhello", ...) = 0
write(1, "hello from the ...", 31) = 31
exit(0)                          = ?
```

Program	System calls	Who makes them
rawhello	2 (+ execve)	our 9 instructions
hello.c, gcc -static	16	libc's start-up code
hello.c, normal	34	ld.so loads libc first

Where do 1 and 60 come from? /usr/include/x86_64-linux-gnu/asm/unistd_64.h: `#define __NR_write 1, __NR_exit 60`. The same numbers are in the kernel's table, arch/x86/entry/syscalls/syscall_64.tbl. This is the whole interface: a number and six registers.

Inside libc: the wrappers are tiny

```
gdb: disassemble getpid (glibc 2.39, real)
```

```
<+0>: endbr64
<+4>: mov    $0x27,%eax    # 39 = getpid
<+9>: syscall
<+11>: ret
```

getpid

Three instructions: put 39 in `eax`, `syscall`, return. `getpid` cannot fail, so there is no error check.

```
gdb: disassemble write (fast path, real)
```

```
<+13>: mov    $0x1,%eax    # 1 = write
<+18>: syscall
<+20>: cmp    $0xffffffffffff000,%rax
<+26>: ja     <+112>        # an error?
<+28>: ret
<+119>: neg    %eax           # -9 -> 9
<+121>: mov    %eax,%fs:(%rdx) # errno = 9
<+124>: mov    $0xffffffffffffffff,%rax # -1
```

write

Same, plus one comparison: a result between `-4095` and `-1` is an error. Then `libc` flips the sign, stores it in `errno` and returns `-1` (Part 4).

%fs

`errno` lives in thread-local storage: every thread has its own `errno` (Lecture 8.1's threads).

The number is the interface, and it differs per architecture

Call	x86-64 (syscall)	i386 (int \$0x80)	arm64, RISC-V (asm-generic)
read	0	3	63
write	1	4	64
getpid	39	20	172
exit	60	1	93

from arch/x86/entry/syscalls/syscall_64.tbl, syscall_32.tbl and include/uapi/asm-generic/unistd.h (Linux 6.8)

```
● ● ● syscall_64.tbl (real lines)
# <number> <abi> <name> <entry point>
0    common read    sys_read
1    common write   sys_write
39   common getpid  sys_getpid
57   common fork    sys_fork
59   64    execve   sys_execve
60   common exit    sys_exit
231  common exit_group ...
```

373 system calls in Linux 6.8 for x86-64, numbered 0 to 461: numbers 335 to 423 are skipped so that newer calls have the **same number on every architecture**. A number, once given, never changes: "we do not break user space". Old programs keep working on new kernels.

C link: like the values of an **enum** that other people's compiled programs already use. You may add new names at the end; you may never renumber the old ones.

PART 3

Inside the kernel

`entry_SYSCALL_64`, the dispatch, and `SYSCALL_DEFINE`: Linux 6.8 source.

From the syscall instruction to C code

1

At boot: `syscall_init()` writes the address of `entry_SYSCALL_64` into a CPU register, `MSR_LSTAR` (`cpu/common.c` 2074).

2

The CPU (syscall): `rcx` = return address, `r11` = flags, switch to ring 0, jump to `MSR_LSTAR`. No stack change.

3

swaps, save the user `rsp`, load **this task's kernel stack** (lines 91–95).

4

Push the user registers: struct `pt_regs`, on the kernel stack (101–109).

5

call `do_syscall_64`(regs, nr) (120): C code from here on.

6

Return: restore registers, `sysretq`: `rip` = `rcx`, flags = `r11`, ring 3 (165).

```
entry_64.S, Linux 6.8 (real lines)

87  SYM_CODE_START(entry_SYSCALL_64)
91  swaps
93  movq %rsp, PER_CPU_VAR(...TSS_sp2)
94  SWITCH_TO_KERNEL_CR3 ...
95  movq PER_CPU_VAR(...top_of_stack), %rsp
   /* Construct struct pt_regs on stack */
101 pushq $__USER_DS      /* pt_regs->ss */
102 pushq ...TSS_sp2      /* pt_regs->sp */
103 pushq %r11            /* pt_regs->flags */
105 pushq %rcx            /* pt_regs->ip */
107 pushq %rax            /* orig_ax */
109 PUSH_AND_CLEAR_REGS rax=$-ENOSYS
120 call do_syscall_64
   ...
165 sysretq
```

Book: Love, LKD Ch. 5; Bovet & Cesati Ch. 10 (system call handler)

Watch one write() go in and come back

One system call: write(1, msg, 31)

SET UP SYSCALL DISPATCH WRITE RETURN

user space: rawhello (ring 3)

```
401000  mov  $1,%rax
401007  mov  $1,%rdi
40100e  lea  msg,%rsi
401015  mov  $31,%rdx
40101c  syscall
40101e  mov  $60,%rax
401025  mov  $0,%rdi
40102c  syscall
```

msg at 0x402000: "hello from the kernel, no libc\n"

CPU registers

rax 1

rdi 1

rsi 0x402000

rdx 0x1f

rcx 0x40101e

r11 0x202

rip kernel

rsp kernel stack

kernel mode, ring 0

kernel space (Linux 6.8)

entry_SYSCALL_64
entry_64.S 87

kernel stack: pt_regs (part)

ss	user
sp	...8af0
flags	0x202
cs	user
ip	0x40101e
orig_ax	1
di	1
si	0x402000
dx	31
ax	-ENOSYS

Step 4: entry_SYSCALL_64 switches to this task's own kernel stack (the user stack cannot be trusted) and pushes every user register there: struct pt_regs. The user's registers are now safe.

Watch for

1. **rax** holds the number, not a name.
2. The CPU fills **rcx** and **r11** itself.
3. The kernel saves everything in **pt_regs**.
4. A **switch** picks the function.
5. The answer comes back in **rax**: 31.

Click the video to play or pause. About 1.5 minutes, no sound.

Dispatch: an array of function pointers, or a switch?

```
● ● ● arch/x86/entry/common.c, Linux v6.8 (real)

42 static bool do_syscall_x64(struct pt_regs *regs, int nr)
48     unsigned int unr = nr;
50     if (likely(unr < NR_syscalls)) {
51         unr = array_index_nospec(unr, NR_syscalls);
52         regs->ax = sys_call_table[unr](regs);
    /* the table: syscall_64.c 16 */
```

```
● ● ● grep ... /proc/kallsyms (QEMU guest, Ubuntu 6.8)
```

```
fffffffffa1c05930 T x64_sys_call
ffffffffffa1d26c10 T __x64_sys_getpid
ffffffffffa20f86e0 T __x64_sys_write
ffffffffffa2e53e30 T do_syscall_64
ffffffffffa3000080 T entry_SYSCALL_64
ffffffffffa3200320 D sys_call_table
```

```
● ● ● Linux v6.8.12 (stable) and Ubuntu 6.8 (real)
```

```
52         regs->ax = x64_sys_call(regs, unr);
    /* syscall_64.c */
27 long x64_sys_call(const struct pt_regs *regs,
    unsigned int nr) {
29     switch (nr) {
    case 0: return __x64_sys_read(regs);
    case 1: return __x64_sys_write(regs); ...
31     default: return __x64_sys_ni_syscall(regs);
```

SAME AS C

Both are plain C

An array of function pointers indexed by a number, or a switch statement: the two ways you learned to pick code by number.

Why the change (2024)? A CPU attack (Spectre-BHI) could misuse the indirect call through the table. Direct calls in a switch cannot be steered. Same table, generated two ways.

SYSCALL_DEFINE3(write): the kernel side of write()

fs/read_write.c, Linux 6.8 (real lines)

```
652 SYSCALL_DEFINE3(write, unsigned int, fd,  
    const char __user *, buf, size_t, count)  
654 {  
655     return ksys_write(fd, buf, count);  
}  
632 ssize_t ksys_write(unsigned int fd, ...)  
634     struct fd f = fdget_pos(fd);  
635     ssize_t ret = -EBADF;  
637     if (f.file) {  
643         ret = vfs_write(f.file, buf, count, ppos);  
570 ssize_t vfs_write(struct file *file, ...)  
574     if (!(file->f_mode & FMODE_WRITE)) return -EBADF;  
578     if (unlikely(!access_ok(buf, count))) return -EFAULT;  
588     ret = file->f_op->write(file, buf, count, pos);
```

The macro makes	Its job
__x64_sys_write(regs)	the entry the switch calls
__se_sys_write(regs->di, regs->si, regs->dx)	takes the arguments out of pt_regs
__do_sys_write(fd, buf, count)	your code: the body after the macro

__user

a mark for checking tools: this pointer belongs to the user; never read it directly.

f_op->write

a function pointer again: the terminal, a pipe or a disk file each bring their own write (Lecture 12).

Book: Love, LKD Ch. 5 (system call implementation)

The kernel never trusts a pointer from user space

● ● ● errno_demo, on a terminal (real)

```
$ ./errno_demo
libc: write(1, NULL, 5)      returns -1  errno = 14  Bad address
libc: write(1, kernel address, 5) returns -1  errno = 14  Bad address
      kernel address = 0xffffffff81000000
```

SAME AS C

A bad pointer is a bug

In C, *p with p = NULL kills your program: SIGSEGV (Lecture 8.1, segv.c).

DIFFERENT FROM C

Here: an error code

The kernel cannot crash because you were careless. It checks, returns EFAULT, and your program goes on.

access_ok(buf, count)

Is the whole range below the user/kernel border? A kernel address fails here: line 578.

copy_from_user(dst, buf, n)

Copy with a safety net: a missing page (like NULL) becomes "bytes not copied", then EFAULT, not a kernel crash.

Why so strict?

If write() would read any address you give, any program could print the kernel's secrets to the screen.

PART 4

Errors: `-errno` and `errno`

How the kernel says no, and what libc turns it into.

The kernel returns `-errno`; libc returns `-1` and sets `errno`

```
./errno_demo (real)
raw syscall: write(42, "x", 1)    returns -9
libc: write(42, "x", 1)          returns -1  errno = 9  Bad file descriptor
libc: syscall(1000)              returns -1  errno = 38  Function not implemented
raw syscall: syscall(1000)       returns -38
libc: write(1, "ok\n", 3)        returns 3
$ strace ./errno_demo
write(42, "x", 1) = -1 EBADF (Bad file descriptor)
syscall_0x3e8(...) = -1 ENOSYS (Function not implemented)
```

-errno	the result
-4095 ... -1: error	0 or more (or an address): success

```
include/linux/err.h 18: #define MAX_ERRNO 4095
```

Why -4095 ... -1 only? `mmap` returns an address, and an address can look like a big "negative" number. The top 4095 values are reserved for errors.

1 **Check the return value first:** `errno` is only meaningful after a call **failed**. A successful call may leave any old value in it.

2 **Save `errno` at once** (`int e = errno;`): `printf` and other calls can change it before you print it.

C link: `perror("open")` and `strerror(errno)` turn the number into the text you saw in Lecture 8.1's mysh: "No such file or directory" is `errno 2` (`ENOENT`).

PART 5

What a system call costs, and how to avoid one

Measured: about 40 C function calls. The vDSO skips the door for the clock.

The price of the door (syscost.c, median of 7 runs)

2 000 000 times each	ns per call	≈ C calls
a C function call	2.4	1
getpid, raw syscall instruction	101	42
getpid() from libc	102	43
write(-1, ...): fails at once	101	42
clock_gettime as a real system call	149	62
clock_gettime() from libc: the vDSO	26	11

our test machine: a cloud virtual machine, Intel Xeon 2.1 GHz. Numbers vary $\pm 20\%$ between runs.

Same work, two prices: clock_gettime through the door costs 149 ns, without the door 26 ns: almost 6 times less. How? Next slide.

Where do 100 ns go?

Switch stack and page tables, save and clear registers, CPU-attack mitigations on entry and exit, then the real work (for getpid: almost none).

The lesson

Make **few, big** calls: one write of 4096 bytes, not 4096 writes of 1 byte. This is why stdio has a buffer (Lecture 1, Homework 2).

The vDSO: kernel code that runs in user mode

```
● ● ● QEMU guest, Ubuntu 6.8 (real)

$ grep -E 'vdso|vvar' /proc/self/maps
7ffdedca5000-7ffdedca9000 r--p ... [vvar]
7ffdedca9000-7ffdedcab000 r-xp ... [vdso]
$ objdump -T vdso.so (copied out by vdso_dump)
__vdso_clock_gettime __vdso_gettimeofday __vdso_time
__vdso_getcpu __vdso_clock_getres (+ sgx_enter_enclave)
```

What it is

A small shared library (an ELF file, 8 KiB) that the kernel maps into every process at `execve`. No file on disk holds it.

How it works

The kernel writes the current time into the read-only [vvar] page at every tick. `clock_gettime` in the vDSO just reads it: no ring change at all.

Who finds it

`ld.so`, through `AT_SYSINFO_EHDR` in the auxiliary vector (Lecture 8.2), and binds `libc`'s `clock_gettime` to it.

```
● ● ● strace -c ./timeloop (our machine, real)

# 1000 x clock_gettime + 1000 x gettimeofday
calls  syscall
      8  mmap
      3  brk ...
    34 total    # not one clock call
```

Book: Love, LKD Ch. 5; the Linux manual: man 7 vdso

LAB 9

A C program with no C library

Write your own syscall wrapper, then measure the door.

Lab 9 tasks, step by step

1 **TODO 1** in `nolibc_starter.c`: write `my_syscall3` with inline assembly (`syscall`; `rax`, `rdi`, `rsi`, `rdx`; `rcx` and `r11` change)

2 **TODO 2**: the numbers of `write`, `getpid`, `exit` from `unistd_64.h`

3 **TODO 3**: `put_num`: print a number without `printf`. **TODO 4**: `argc` and `argv` from the stack

4 Run `strace ./nolibc hello`; compare with the `libsys` quiz: why so many writes?

5 Run `./syscost` on the lab PC; fill the table; explain the `vDSO` line

Before TODO 1: the starter prints nothing and dies with "Illegal instruction" (exit status 132 = 128 + 4, SIGILL): exit never reached the kernel. `strace` shows every call as `syscall_0x3e8 ... = -1 ENOSYS`.

the finished `nolibc` (real)

```
$ ./nolibc hello world; echo $?
my PID is 417
argc = 3
argv: hello
argv: world
write(42, ...) returned -9 (the
kernel returns -errno; 9 = EBADF)
3
$ strace ./nolibc hello
execve("./nolibc", ...) = 0
write(1, "my PID is ", 10) = 10
getpid()                    = 390
write(1, "390", 3)           = 3 ...
write(42, "x", 1)            = -1 EBADF
exit(2)                      = ?
```

Checkpoint: test yourself

Q1

Which register holds the system call number on x86-64? And the 4th argument?

→ **rax; r10** (not rcx: the CPU overwrites rcx)

Q2

Does `strlen()` enter the kernel? Does `malloc(100)`?

→ **strlen never; malloc only when it needs more memory (brk or mmap)**

Q3

The kernel returned `-2` for `open()`. What does your C code see?

→ **-1, and `errno = 2` (ENOENT)**

Q4

Why does the kernel switch to its own stack at entry?

→ **the user `rsp` cannot be trusted; each task has a kernel stack**

Q5

Why is `clock_gettime` 6× cheaper than `getpid`?

→ **the vDSO: it runs in user mode and reads `[vvar]`**

Key ideas from today



A system call is a request by number; libc's wrappers put it in rax and run syscall.



Arguments in rdi, rsi, rdx, r10, r8, r9; result in rax; the CPU saves rip in rcx, flags in r11.



entry_SYSCALL_64 → pt_regs on the kernel stack → do_syscall_64 → switch on the number → sys_write.



The kernel checks every argument: bad fd → EBADF, bad pointer → EFAULT, never a crash.



Errors come back as -1 ... -4095; libc returns -1 and sets errno (per thread).



≈ 100 ns per call (≈ 40 function calls). Batch your work; the vDSO avoids the door for time.

Homework 9 (due before Lecture 10.1)

1. `int $0x80`

Read `int80.c` (demo folder): the same message is written with `int $0x80` and `eax = 4`, then with `syscall` and `rax = 4`. Predict both results, then build with `-static` and run. Explain.

2. One byte

`bytes.c` writes 1 MiB one byte at a time, with `write()` or with `putchar()`. Count the system calls of each (`strace -c`) and time both. Explain the ratio.

3. Numbers

System call number 39 is `getpid` on x86-64. What is 39 on i386 and on arm64? Use the three tables in the lab folder (`tables/`).

4. Which?

A program calls `time`, `gettimeofday`, `clock_gettime`, `sched_getcpu`, `getpid` three times, `getppid`. How many system calls does `strace` show? Check with `which.c`.

5. `/dev/null`

`write(1, NULL, 5)` gives `EFAULT` on a terminal. Run `./errno_demo > /dev/null` under `strace`: what does that write return now? Explain with `vfs_write`.

Next lecture and reading

NEXT: LECTURE 10.1

Inter-process communication

Processes share nothing (Lecture 8.1). How do they talk? Pipes, FIFOs, shared memory: each one a few system calls you now know how to read in strace.

Main text

R. Love, **Linux Kernel Development**, 3rd ed., Ch. 5 (today). M. Kerrisk, **TLPI**, Ch. 3 (today) and Ch. 44 (pipes and FIFOs) for 10.1.

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §8.1.3 (system calls) and §8.3 (system call error handling).

Source

Linux 6.8: arch/x86/entry/entry_64.S, common.c, syscall_64.tbl, fs/read_write.c, kernel/sys.c (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

lab9_files.zip: nolibc_starter.c, syscost.c and the demo programs.