

Elements of Operating Systems

Chapter 2

The Boundary

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · September 28, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

2 The Boundary

In the previous chapter we left Ken Thompson writing a system for himself and three colleagues. This chapter asks what, exactly, it was that he wrote —and why the program you have half-written cannot work without it.

2.1 A system is hardware and software

When you say “the system is slow” you don’t mean the processor. You mean the whole thing: the machine and the software that runs it, which to you are a single thing because you never see one without the other.

The **operating system** is the software that manages the hardware’s resources and acts as the go-between for applications and the machine. Inwards, it defines data structures to manage the hardware and algorithms to decide how to share it out; outwards, it offers a set of functions for asking it to do that managing.

The two halves of the definition are the two halves of the book. The inward one —tables, queues, schedulers, inodes— is what will keep turning up chapter after chapter. The outward one is that set of functions, which have a name of their own: **system calls**, and in this book they are always **in red**. From here on, every time you see a red word in a listing, that is where the program stops being its own.

Notice that the definition does not say “program”. The operating system is not a program that runs while you wait: it is code that runs *inside* your execution, when its turn comes, and that does nothing the rest of the time. That is the first thing that is hard to see, and it is

where we are heading.

2.2 Without an operating system

Let's picture the machine without one, because that is how every machine was until somebody decided otherwise.

One program on its own, loaded into memory, with the whole machine to itself. There is nobody to ask for anything: if you want to read from the disk, you write to the controller's registers yourself. There are no files, there are sectors. There are no processes, there is *the* program. And there is no protection at all, because there is nobody to be protected from.

This exists and has a name, *bare metal*, and it is still what the microcontroller in a dishwasher does. It is not a relic: it is the right answer when the machine does one thing, always the same one.

The operating system is not a convenience: it is the answer to **sharing** the machine. When there is only one program there is nothing to share out, nothing to protect and nothing to referee.

You get three things out of it, and the book chases all three:

- a **usable** environment, which spares you from knowing what the disk controller of this particular machine looks like;
- a **safe** environment, which stops a program from touching the hardware as it pleases, or another program's memory;
- an **efficient** environment, which shares one machine among many and makes each of them feel it has the whole machine.

Everything that follows comes from the second and the third. Because if a program cannot touch the hardware, but needs it, somebody has to touch it on the program's behalf —and there has to be a way to ask.

2.3 The two execution modes

The question is how a program is stopped from doing what it must not do. The answer does not come from the operating system: it comes from the hardware.

The processor has at least two execution modes.*

- In **user mode** there are machine-language instructions that the processor refuses to execute and memory addresses it refuses to read. It isn't that they are forbidden: they *don't work*.
- In **kernel mode** there are none of these restrictions.

You will also hear the second one called **system mode** and **privileged mode**; all three names mean the same thing, and this book uses them interchangeably. The first, on the other hand, is always just **user mode**.

Your program always runs in user mode. The operating system's code runs in kernel mode. And the hardware knows at every moment which of the two it is in, because it keeps it in a bit of one of its registers that user mode cannot write.

Here we need to separate two words that the book will use all the time. The **kernel** is the part of the operating system that runs in kernel mode—hence the mode's name—: the code that manages the hardware and serves the system calls. The **operating system** is broader: it also includes the C library, the shell and the commands, which run in user mode like your program. When we say “the kernel”, we mean that part and nothing else.

Careful

It isn't the operating system that stops you: it is the processor. If you execute a privileged instruction from a user program, nobody checks anything—the hardware raises an exception and takes control away from you. The operating system wasn't even there until that exception woke it up.

*At least. The x86 defined four, the “rings”, and in practice systems use two. Virtualisation has added another one underneath.

2.3.1 So when does the kernel run?

If the operating system is not running while you are, we have to say when it does. There are only three doors, and you saw all three in Computer Structure:

- a hardware **interrupt** —the keyboard, the clock, a disk that has finished—, which is *asynchronous*: it arrives between any two instructions and has nothing to do with what you were doing;
- an **exception** —a division by zero, a page fault—, which is *synchronous*: it is caused by the instruction being executed, and always by the same one;
- a **system call**, which is also synchronous and, what is more, *voluntary*: you cause it, with an instruction put there on purpose to ask for a service.

From the hardware's point of view the three are almost the same thing: save where we were, switch mode and jump to an address that the operating system prepared at boot time. The difference is who causes them and why.

i Careful

If none of the three ever happened, the operating system would never run again and a program could keep the machine forever. That is why the kernel, when it boots, programs the **clock interrupt** to fire on its own every few milliseconds. This is not an implementation detail: it is the only reason the system does not lose control, and it is what makes Chapter 6 possible.

2.4 Two interfaces: the shell and the syscalls

Now we can say what the book is about. If the system is hardware plus software, and the only way in is through the door, there are exactly two doors depending on who knocks:

- the **user interface**, for a person sitting at a terminal: it is the shell, and it is Chapter 3;
- the **programming interface**, for a program: it is the system calls, and it is the whole rest of the book.

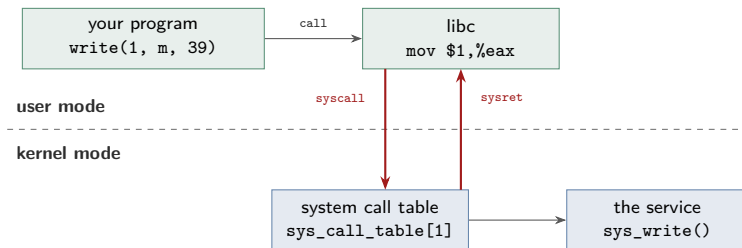
And here is the joke you need to get straight away: the shell is not the operating system. It is a user program like yours, which runs in user mode like yours, and which, when you type `ls`, does exactly what you would do—system calls. That is why Chapter 1 could say that taking the command interpreter out of the kernel was an idea, and a good one.

2.5 What really happens when you call `write()`

Let's get down to it. You have this:

```
write(1, m, 39);
```

and it looks like a function call. It is one, and it isn't, and both at the same time. Let's take a look.



The top arrow never leaves user mode: the program and the library are on the same side. The two vertical arrows are the only thing that crosses, and each one is a single instruction. The code below is not part of your executable.

Figure 2.1. The journey of a `write`. No ordinary jump crosses the line in the middle.

Five steps take place:

1. Your program makes a perfectly **ordinary** function call, a **call**, to a function named **write** that lives in the C library. You are still in user mode and nothing special has happened.
2. That function —four instructions, no more— puts the arguments in the registers the kernel expects, and the call’s **number** in a register of its own.
3. It executes **the** instruction. **One**. It switches mode and jumps, and it does both at once, with no chance for anything to slip in between.
4. Now in kernel mode, the system takes that number, uses it to index a table and calls the corresponding service. That code **is not part of your executable**: it was already there before you started.
5. It returns, user mode is restored, and the libc function looks at the result and hands it to you.

The word for step 3 is **trap**, and it is the generic name: each architecture has its own instruction, and no two call it the same thing.

The C library *is not* the operating system. It runs in user mode, it is part of your executable and you can disassemble it. The only thing it does is know which registers have to be filled on this machine.

A warning before your first **strace**, so that what you see there does not throw you. This book is about Unix, and the rulebook for Unix is POSIX: the list of system calls you will study is the one POSIX gives, and each system implements it as it sees fit. Linux, which is what you have in the lab, often prefers to offer a single, more general call and let the library build the ones POSIX asks for. That is why the name you see crossing the boundary will not always be the one you wrote: a **fork()** shows up there as a **clone()**; an **open()**, as an **openat()**; a

`stat()`, as a `newfstatat()`; a `pipe()`, as a `pipe2()`. Whenever that happens, the book will say so in a footnote. None of this changes what a system call is —`fork()` is one— nor the picture above: there is a library function, there is a trap, and there is kernel code on the other side.

2.5.1 Why the trap and not a call

You might ask why all the ceremony, if in the end it is just a matter of jumping to a piece of code. There are three reasons, and all three are good ones:

- A `call` does not switch mode. You would still be in user mode, and the kernel's code could not do anything you couldn't already do.
- A `call` jumps to an *address*, and the address of the kernel's code changes with every version. Your program, compiled three years ago, would have to know where it is today.
- A `call` jumps wherever you tell it, and that means jumping into the middle of a kernel routine and skipping its checks. The trap instruction can only jump to one place, the one the system fixed at boot time.

The number solves the second problem. A system call is identified not by an address but by a **number**, which indexes a table the kernel keeps and which has to go on meaning the same thing from one version to the next. It is the only thing a compiled binary can know in advance.

2.6 The table, with no “it depends”

Now the comparison, which is the heart of the chapter. First, by concept:

	Function call	System call
who runs the code	your process	the kernel
execution mode	user, all the way	switches to kernel and back
where the code is	in your executable	it was already there
how it knows where to jump	the linker resolved it at build time	a number that indexes a kernel table
who can say no	nobody	the kernel, on permissions
how it reports failure	however it likes	always <code>-1</code> and <code>errno</code>
debugged with	<code>gdb</code>	<code>strace</code>
what it costs	2 ns	100 to 400 ns

And now the same with the real registers and instructions. There are three columns, not two, and the reason is the crux of the matter: there are *two* ARM64 columns, which are the same hardware with two different operating systems.

	x86-64 Linux	ARM64 Linux	ARM64 macOS
<i>Function call</i>			
arguments	<code>rdi rsi rdx rcx r8 r9</code>	<code>x0...x7</code>	<code>x0...x7</code>
instruction	<code>call</code>	<code>b1</code>	<code>b1</code>
result	<code>rax</code>	<code>x0</code>	<code>x0</code>
to return	<code>ret</code>	<code>ret</code>	<code>ret</code>
<i>System call</i>			
arguments	<code>rdi rsi rdx r10 r8 r9</code>	<code>x0...x5</code>	<code>x0...x5</code>
the number goes in	<code>rax</code>	<code>x8</code>	<code>x16</code>
the trap is	<code>syscall</code>	<code>svc #0</code>	<code>svc #0x80</code>
result	<code>rax</code>	<code>x0</code>	<code>x0</code>
it failed if	<code>rax</code> is between <code>-1</code> and <code>-4095</code>	<code>x0</code> is negative	the <i>carry flag</i> is set
write is	1	64	4
and <code>getpid</code>	39	172	20

This table can be read in two ways, and both teach you something.

Compare the first two columns and you will see what the architecture decides: the names of the registers, how many there are and what the instruction is called. Everything changes, and it could not be otherwise, because they are two different instruction sets.

Compare the last two and you will see what the operating system decides, because here the hardware is exactly the same chip. In the top half *nothing* changes: the arguments, the instruction and the return value are identical. In the bottom half almost everything changes—the number’s register, the trap’s operand, the number of each call and even how failure is reported.

The function-call convention is set by **the architecture**; the system-call convention is set by **the operating system**. That is why “for Apple Silicon” means nothing until you also say which system runs on it.

And there are two more things hidden in the table.

The first: within each column, **the arguments go in the same registers** whether it is a function call or a system call. That is no coincidence: it is done on purpose so that the library function does not have to move anything around.

The second: on x86-64 the system call uses `r10` where the function call uses `rcx`. It looks like a whim and it isn’t, and it is the exception that proves the rule—the `syscall` instruction *clobbers rcx*, because it stores the return address there, so no argument can live in it.

2.6.1 Let’s look

None of this has to be taken on trust. Here is the same `write`, written without the library, with one branch for each column of the table:

Listing 2.1. boundary-trap: the system call by hand

```

1 static long trap_write(int fd, const char *buf, unsigned ↵
   long n)
2 {
3     #if defined(__x86_64__) && !defined(__APPLE__)
4         long r;
5         __asm__ volatile ("syscall"                               ❶
6                             : "=a" (r)

```

```

7             : "a" (1L),                ❷
8             "D" ((long) fd),          ❸
9             "S" ((long) buf),
10            "d" ((long) n)
11            : "rcx", "r11", "memory");  ❹
12    return r;
13
14    #elif defined(__aarch64__) && !defined(__APPLE__)
15        register long x8 __asm__("x8") = 64;          ❺
16        register long x0 __asm__("x0") = fd;
17        register long x1 __asm__("x1") = (long) buf;
18        register long x2 __asm__("x2") = n;
19        __asm__ volatile ("svc #0"                  ❻
20                           : "+r" (x0)
21                           : "r" (x8), "r" (x1), "r" (x2)
22                           : "memory");
23    return x0;
24
25    #elif defined(__aarch64__) && defined(__APPLE__)
26        register long x16 __asm__("x16") = 4;          ❼
27        register long x0 __asm__("x0") = fd;
28        register long x1 __asm__("x1") = (long) buf;
29        register long x2 __asm__("x2") = n;
30        register long x3 __asm__("x3");
31        __asm__ volatile ("svc #0x80\n\t"            ❽
32                           "cset x3, cs"             ❾
33                           : "+r" (x0), "=r" (x3)
34                           : "r" (x16), "r" (x1), "r" (x2)
35                           : "cc", "memory");
36    return x3 ? -x0 : x0;                      ❿
37
38    #else
39    # error "written for x86-64 and ARM64; for anything else, ↵
40        check it first"
41    #endif
42    }
43
44    int main(void)
45    {
46        const char *m = "I've crossed the boundary with no libc\n";
47        long r;
48
49        r = trap_write(1, m, strlen(m));
50        if (r < 0) {                               ❾
            errno = -r;

```

```

51     error_and_exit("write", 1);
52 }
53 return 0;
54 }

```

- ❶ The Intel instruction. Just one; the lines above only fill registers.
- ❷ The number of `write` on Linux, which goes in `rax`.
- ❸ The arguments, in the same registers as in a function call.
- ❹ `rcx` and `r11` are clobbered by the instruction. We have to tell the compiler so that it doesn't leave anything it needs there.
- ❺ The same number, on Linux on ARM, is not 1 but 64. And the register is not `rax` but `x8`.
- ❻ And the instruction is a different one.
- ❼ The same chip, another system: now the number is 4 and the register is `x16`.
- ❽ And the trap carries an operand that Linux doesn't have.
- ❾ Here there is no negative result to look at. `cset` copies the carry flag into a register, which is the only way to tell, from C, whether things went well.
- ❿ And that is why the function has to manufacture the negative value itself, so that it can return the same thing on all three.
- ⓫ The error arrives as a negative number, not as a `-1`. Turning it into `errno` is what the library does for you.

❗ Careful

Look at the `#ifs` carefully, because there is a trap in them, and an expensive one. On a Mac with Apple Silicon the macro `__aarch64__` *is defined*, so if the branches only looked at the architecture, a Mac would take the Linux branch —and compile it without a single complaint. When you run it, this:

```

$ make boundary-trap && ./boundary-trap
cc -Wall -Wextra -O0 -g -o boundary-trap boundary-trap.c
tools.c
zsh: invalid system call ./boundary-trap

```

The chip executed the `svc` perfectly. What didn't understand what the 64 meant was the operating system, which answered

with a SIGSYS. The same source code and the same silicon, and it dies because the one in charge is different: the table a few pages back describes exactly this, and these two lines prove it. That is why the branches have to look at **the architecture and the system**, which is what the `&& !defined(__APPLE__)` does.

Compile it and have a look at it. So that you don't have to hunt for the function among all of `objdump`'s output, you can write yourself a small shell function that shows just one:

```
$ dis() { objdump -d --no-show-raw-insn --disassemble=$1 $2 \
> | sed -n "/<$1>:/,/^$/p"; }
$ gcc -O2 -o boundary-trap boundary-trap.c tools.c
$ dis main boundary-trap
0000000000401070 <main>:
401070:      mov     $0x1,%eax
401075:      mov     $0x402008,%esi
40107a:      mov     $0x27,%edx
40107f:      mov     %rax,%rdi
401082:      syscall
401084:      test    %rax,%rax
401087:      js      40108c <main+0x1c>
401089:      xor     %eax,%eax
40108b:      ret
40108c:      sub     $0x18,%rsp
401090:      mov     %rax,0x8(%rsp)
401095:      call    401030 <__errno_location@plt>
40109a:      mov     $0x1,%esi
40109f:      mov     $0x402030,%edi
4010a4:      mov     %rax,%rcx
4010a7:      mov     0x8(%rsp),%eax
4010ab:      neg     %eax
4010ad:      mov     %eax,(%rcx)
4010af:      call    401060 <error_and_exit>
4010b4:      xor     %eax,%eax
4010b6:      add     $0x18,%rsp
4010ba:      ret
4010bb:      nopl     0x0(%rax,%rax,1)
```

With `-O2` the compiler has pulled the function into `main`, and this output is complete. The first seven lines are the whole call: `$0x1` is the number of `write`; `$0x402008` is the address of the message; `$0x27` is the 39 from `strlen`, which the compiler managed to count by itself; `mov %rax,%rdi` puts file descriptor 1 in the first argument, taking advantage of already having it; the `syscall` is the boundary, and the `test` and the `js` check whether the result is negative. If it isn't, the program ends two lines further down.

All the rest is the error path. It asks the library where `errno` is, flips the sign of the result (`neg`), stores it there, and calls `error_and_exit`. That is balloon ① of Listing 2.1, and in a page you will find it again inside the C library.

And now the same thing on ARM, with other names and the same skeleton:

```
$ dis main boundary-trap
0000000000000780 <main>:
780:  adrp    x1, 0
784:  mov     x8, #0x40      // #64
788:  add     x1, x1, #0x9c0
78c:  mov     x0, #0x1       // #1
790:  mov     x2, #0x27      // #39
794:  svc     #0x0
798:  tbnz    x0, #63, 7a4 <main+0x24>
```

The `0x40` is the 64 from the Linux table and the `svc` is the boundary. The last one, `tbznz x0, #63`, says “if bit 63 is set”: that is, if the result is negative. The same check as in the other one, said differently.

And if you disassemble the C library you will find this very thing, but not where you would expect it. In the lab, the library's `__write` contains no `syscall` at all*:

*With `glibc` 2.43, the one in the lab. With 2.39, Ubuntu 24.04's, it does contain it, and you will find the rest of this section all inside `__write`.

```

$ L=$(ldd /bin/ls | awk '/libc.so/ {print $3}')
$ dis __write $L
0000000000112ace <__write>:
  112ace:      endbr64
  112ad2:      nop
  112ad3:      nop
  112ad4:      sub     $0x10,%rsp
  112ad8:      movslq  %edi,%rdi
  112adb:      xor     %r9d,%r9d
  112ade:      xor     %r8d,%r8d
  112ae1:      push    $0x1
  112ae3:      xor     %ecx,%ecx
  112ae5:      call    97e1e <__syscall_cancel>
  112aea:      add     $0x18,%rsp
  112aee:      ret

```

It pushes the number of `write` (the `push $0x1`) and calls a common function, the same one for every system call that may be left waiting. The reason lies with the threads of Chapter 12: a thread waiting inside a `write()` must be cancellable by another one, and the library keeps watch over that in a single place instead of in every call. Older versions of the library did not do it this way, and `__write` contained the `syscall` directly; check which one you have.

The common function is the one that holds the answer to “how do we know it failed”:

```

$ dis __syscall_cancel $L
0000000000097e1e <__syscall_cancel>:
  97e1e:      endbr64
  97e22:      nop
  97e23:      nop
  97e24:      sub     $0x10,%rsp
  97e28:      push    0x18(%rsp)
  97e2c:      call    97d7e <__internal_syscall_cancel>
  97e31:      pop     %rdx
  97e32:      pop     %rcx
  97e33:      cmp     $0xffffffffffff000,%rax
  97e39:      ja      97e40 <__syscall_cancel+0x22>
  97e3b:      add     $0x8,%rsp

```

```

97e3f:      ret
97e40:      mov     0x15cfa1(%rip),%rdx      # 1f4de8
      <_GLOBAL_OFFSET_TABLE_+0x268>
97e47:      neg     %eax
97e49:      mov     %eax,%fs:(%rdx)
97e4c:      mov     $0xffffffffffffffff,%rax
97e53:      add     $0x8,%rsp
97e57:      ret

```

The real `syscall` is one floor further down, in the function the `call` calls, which checks whether anyone has asked to cancel the thread before it crosses:

```

$ dis __internal_syscall_cancel $L | grep -B1 'syscall$'
97e03:      mov     0x20(%rsp),%rax
97e08:      syscall

```

The instructions after the `call` are the whole mystery of `errno`. The kernel returns the error code as a small negative number; the library compares against `-4096` to see whether it falls in that range, and if it does, it flips its sign (`neg`), stores it where the process keeps `errno` and returns `-1` —which is the `$0xffffffffffffffff` in the last line but one. The rule “every system call returns `-1`” does not come from the kernel: it comes from here. And it is the same thing the compiler had done by hand in `boundary-trap`.

2.6.2 🕒 And the third column

That leaves the one you cannot look at from the lab, which is where the SIGSYS of a couple of pages ago comes from. The macOS system library is called `libsystem_kernel.dylib`, and the disassembler, `otool`:

```
$ otool -tV /usr/lib/system/libsystem_kernel.dylib \
    | grep -A 4 '^_write:'
_write:
0000000000000482c    mov     x16, #0x4
00000000000004830    svc     #0x80
00000000000004834    b.lo    0x4854
00000000000004838    pacibsp
```

Four instructions, like the other two, and the table’s three differences all together: the number goes in `x16` and is 4, and the trap is `svc #0x80`. The third is the prettiest. Where Linux checked the sign of the result, here there is a `b.lo`, which on ARM means “branch if the carry flag is *clear*” —in other words, branch if there has **not** been an error, and fall through to the error routine if there has. The same job, a different convention.

And the numbers in the table are literally these:

```
$ grep -w 'SYS_write\|SYS_getpid' \
    $(xcrun --show-sdk-path)/usr/include/sys/syscall.h
#define SYS_write      4
#define SYS_getpid     20
```

Going further

All of this is a demonstration, not a technique, and the two systems say opposite things about it. On macOS the call numbers **are not a stable interface**: they may change between versions, and the only thing Apple promises is that the library will keep working with them. On Linux it is exactly the other way round, and it is a famous promise of Torvalds’ —the table is never touched—, which is why there are languages that skip `libc` and do the traps themselves. Where each system decides to put its compatibility boundary says more about it than any detail of its registers: Apple puts it in the library and Linux in the kernel.

2.7 What crossing it costs

It remains to see whether any of this is noticeable. Three things that look the same from C:

Listing 2.2. boundary-cost: what crossing costs

```
1  int main(int argc, char *argv[])
2  {
3      long n = (argc > 1) ? atol(argv[1]) : REPETITIONS;
4      struct timespec t;
5      double t0;
6      char buf[128];
7      long i;
8
9      if (n <= 0)
10         n = REPETITIONS;
11
12     t0 = now();
13     for (i = 0; i < n; i++)
14         sink = sum(i, 1);
15     sprintf(buf, "function call ..... %6.1f ns\n",
16             (now() - t0) * 1e9 / n);
17     write(1, buf, strlen(buf));
18
19     t0 = now();
20     for (i = 0; i < n; i++)
21         clock_gettime(CLOCK_MONOTONIC, &t);
22     sprintf(buf, "clock_gettime (vDSO) . %6.1f ns\n",
23             (now() - t0) * 1e9 / n);
24     write(1, buf, strlen(buf));
25
26     t0 = now();
27     for (i = 0; i < n; i++)
28         sink = getpid();
29     sprintf(buf, "getpid (crosses) ..... %6.1f ns\n",
30             (now() - t0) * 1e9 / n);
31     write(1, buf, strlen(buf));
32
33     return 0;
34 }
```

- ❶ Three identical loops, and the only thing that changes is what is inside. The first calls a two-line `sum` of our own, which never leaves the process.
- ❷ This one *looks* like a system call and isn't one.
- ❸ And this one is, and a cheap one at that: the kernel already has the number at hand. So what is measured is the trip, not the service.

```
$ ./boundary-cost
function call ..... 1.8 ns
clock_gettime (vDSO) . 13.3 ns
getpid (crosses) ..... 410.1 ns
```

More than two hundred times. And `getpid` is the cheapest system call there is: it does nothing, it just goes and comes back. Any of the ones that do something costs a lot more, and Chapter 9 will come back to this when it compares four hundred thousand calls with a hundred.

The number on the last line depends a lot on the machine, and you can make the comparison yourselves. In two lab rooms, with the **same** distribution and the same kernel, the same program gives 410 ns in one and 73 ns in the other. The difference is not the speed of the processor—the second one is only slightly newer—it is that the first is one of those affected by Meltdown, and the kernel, to protect it, does more work on every trip. The **Vulnerability** lines of `lscpu` say so: where one machine says **Meltdown: Mitigation; PTI**, the other says **Not affected**. When a boundary is unsafe, what gets more expensive is crossing it.

❗ Careful

The one in the middle is the interesting case. `clock_gettime()` does *not* cross the boundary: the kernel has left a copy of the code and of the time inside every process's memory—the *vDSO*—precisely because it is such a frequent call that it cannot afford the trip. So there are things that look like system calls and aren't. In this book you will find them in red all the same, because POSIX

says they are system calls; what changes is the way they go.

2.8 Exercises

1. Compile one program that does a `printf` and another that does a `write()`, and run both through `strace`. How many system calls does each make? Why isn't it the same number?
2. Disassemble the C library on your machine and look for `getpid` in it. How many instructions does it have? Which one is the trap?
3. Write a one-line program that executes a privileged instruction: `__asm__("hlt")` if you are on Intel, `__asm__("msr daifset, #2")` if you are on ARM. Does it compile? Does it run? Which signal does it die with, and why isn't it the same on both? Look at `dmesg` too: whoever stopped you has left a note there.
4. In Listing 2.1 there is a number, the one for `write`. Find where it is written down on your system and check that it is the one we give. On Linux, look at `/usr/include/asm/unistd_64.h`.
5. Pass it an invalid file descriptor, say 99. What does the function return? And what would the library's `write` have returned?
6. Run `boundary-cost` on the lab machine and on your own. Do the three numbers change in the same way, or does one change more? What do you conclude from that?