

Elements de Sistemes Operatius

Capítol 2

La frontera

J. Fornés M. Farreras Y. Becerra

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Grau en Enginyeria Informàtica — Sistemes Operatius

Versió de prova per a lectors · 23 de setembre de 2026

© Universitat Politècnica de Catalunya

Aquesta obra està subjecta a la llicència de Creative Commons Reconeixement-NoComercial-CompartirIgual 4.0 Internacional (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.ca>

Aquest text s'ha escrit amb l'ajuda de Claude, el model de llenguatge d'Anthropic.
Tots els errors que quedin són dels autors.

2 La frontera

Al capítol anterior hem deixat en Ken Thompson escrivint un sistema per a ell i tres companys més. Aquest capítol pregunta què és, exactament, això que va escriure —i per què el programa que teniu a mig fer no pot funcionar sense.

2.1 Un sistema és maquinari i programari

Quan dieu «el sistema va lent» no us referiu al processador. Us referiu al conjunt: la màquina i el programari que la governa, que per a vosaltres és una sola cosa perquè mai no en veieu una sense l'altra.

El **sistema operatiu** és el programari que administra els recursos del maquinari i fa d'intermediari entre les aplicacions i la màquina. Cap endins defineix estructures de dades per gestionar el maquinari i algorismes per decidir com repartir-lo; cap enfora ofereix un conjunt de funcions per demanar-li aquesta gestió.

Les dues meitats de la definició són les dues meitats del llibre. La de cap endins —taules, cues, planificadors, inodes— és el que anirà sortint capítol rere capítol. La de cap enfora és aquest conjunt de funcions, que tenen nom propi: les **crides al sistema**, i que en aquest llibre van sempre **en vermell**. A partir d'aquí, cada cop que en un llistat veieu una paraula vermella, allà el programa deixa de ser seu.

Fixeu-vos que la definició no diu «programa». El sistema operatiu no és un programa que s'executa mentre vosaltres espereu: és codi que s'executa *dins* de la vostra execució, quan li toca, i que la resta del temps no fa res. Aquesta és la primera cosa que costa de veure i és on anem.

2.2 Sense sistema operatiu

Imaginem-nos la màquina sense ell, perquè és com eren totes fins que algú va decidir el contrari.

Un programa sol, carregat a la memòria, amb la màquina sencera per a ell. No hi ha ningú a qui demanar res: si voleu llegir del disc, escriviu vosaltres als registres del controlador. No hi ha fitxers, hi ha sectors. No hi ha processos, hi ha *el* programa. I no hi ha cap protecció, perquè no hi ha de qui protegir-se.

Això existeix i té nom, *bare metal*, i encara és el que fa el microcontrolador d'un rentaplats. No és una antigalla: és la resposta correcta quan la màquina fa una sola cosa i sempre la mateixa.

El sistema operatiu no és una comoditat: és la resposta a **compartir** la màquina. Quan hi ha un sol programa no cal repartir res, no cal protegir res i no cal arbitrar res.

En traieu tres coses, i el llibre les persegueix totes tres:

- un entorn **usable**, que us estalvia saber com és el controlador de disc d'aquesta màquina en concret;
- un entorn **segur**, que impedeix que un programa toqui el maquinari com vulgui o la memòria d'un altre;
- un entorn **eficient**, que reparteix una màquina entre molts i fa que cadascú tingui la sensació de tenir-la sencera.

De la segona i la tercera se'n deriva tot el que ve ara. Perquè si un programa no pot tocar el maquinari, però el necessita, algú l'hi ha de tocar per ell —i cal una manera de demanar-ho.

2.3 Els dos modes d'execució

La pregunta és com s'impedeix que un programa faci el que no ha de fer. La resposta no és del sistema operatiu: és del maquinari.

El processador té, com a mínim, dos modes d'execució.*

- En **mode usuari** hi ha instruccions de llenguatge màquina que el processador es nega a executar i adreces de memòria que es nega a llegir. No és que estigui prohibit: és que *no funciona*.
- En **mode kernel** no hi ha cap d'aquestes restriccions.

D'aquest segon en sentireu dir també **mode sistema** i **mode privilegiat**, i els tres noms volen dir el mateix; en aquest llibre s'usen indistintament. El primer, en canvi, es diu sempre **mode usuari** i prou.

El vostre programa s'executa sempre en mode usuari. El codi del sistema operatiu s'executa en mode kernel. I el maquinari sap en tot moment en quin dels dos està, perquè ho duu en un bit d'un registre seu que el mode usuari no pot escriure.

Aquí convé separar dues paraules que el llibre farà servir contínuament. El **kernel** és la part del sistema operatiu que s'executa en mode kernel —d'aquí el nom del mode—: el codi que administra el maquinari i atén les crides al sistema. El **sistema operatiu** és més ampli: hi ha també la llibreria de C, el shell i les comandes, que s'executen en mode usuari com el vostre programa. Quan diem «el kernel», volem dir aquella part i prou.

Compte

Qui us atura no és el sistema operatiu: és el processador. Si des d'un programa d'usuari executeu una instrucció privilegiada, no hi ha cap comprovació de ningú —el maquinari genera una excepció i us treu el control. El sistema operatiu ni tan sols hi era, fins que aquesta excepció l'ha despertat.

2.3.1 Quan s'executa, doncs, el kernel?

Si el sistema operatiu no s'està executant mentre vosaltres us executeu, cal respondre quan ho fa. Només hi ha tres portes, i totes tres les vau

*Com a mínim. L'x86 en va definir quatre, els «anells», i a la pràctica els sistemes en fan servir dos. La virtualització n'hi ha afegit un altre per sota.

veure a Estructura de Computadors:

- una **interrupció** de maquinari —el teclat, el rellotge, un disc que ha acabat—, que és *asíncrona*: arriba entre dues instruccions qualssevol i no té res a veure amb el que fèieu;
- una **excepció** —una divisió per zero, una fallada de pàgina—, que és *síncrona*: la provoca la instrucció que s’executava i sempre la mateixa;
- una **crida al sistema**, que també és síncrona i que a més és *voluntària*: la provoqueu vosaltres, amb una instrucció posada a posta per demanar un servei.

Des del punt de vista del maquinari les tres són gairebé la mateixa cosa: guardar on érem, canviar de mode i saltar a una adreça que el sistema operatiu va deixar preparada en arrencar. La diferència és qui les provoca i per què.

i Compte

Si cap de les tres no passés mai, el sistema operatiu no s’executaria mai més i un programa podria quedar-se la màquina per sempre. Per això el kernel, en arrencar, programa la **interrupció de rellotge** perquè salti sola cada pocs mil·lisegons. No és un detall d’implementació: és l’única raó per la qual el sistema no perd el control, i és el que fa possible el capítol 6.

2.4 Dues interfícies: el shell i les syscalls

Ara ja es pot dir de què va el llibre. Si el sistema és maquinari més programari, i només s’hi pot entrar per la porta, hi ha exactament dues portes segons qui truqui:

- la **interfície d’usuari**, per a una persona asseguda davant d’un terminal: és el shell, i és el capítol 3;
- la **interfície de programació**, per a un programa: són les crides al sistema, i són tota la resta del llibre.

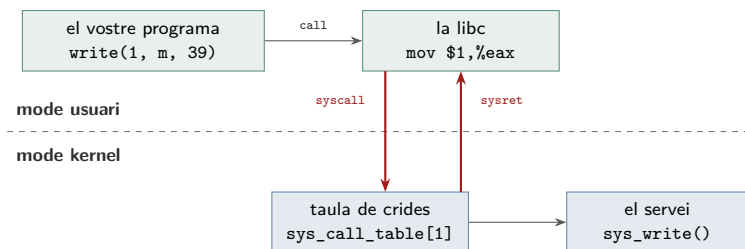
I aquí hi ha la broma que cal entendre de seguida: el shell no és el sistema operatiu. És un programa d'usuari com el vostre, que s'executa en mode usuari com el vostre, i que quan escriviu `ls` fa exactament el que faríeu vosaltres —crides al sistema. Per això el capítol 1 podia explicar que treure l'interpret de comandes del kernel va ser una idea, i una de bona.

2.5 Què passa de debò quan feu un `write()`

Anem al gra. Teniu això:

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

i sembla una crida a funció. N'és una, i no ho és, i les dues coses alhora. Anem a mirar-ho.



La fletxa de dalt no surt del mode usuari: el programa i la llibreria són a la mateixa banda. Les dues verticals són l'única cosa que travessa, i cadascuna és una sola instrucció. El codi de sota no forma part del vostre executable.

Figura 2.1. El viatge d'un `write`. La ratlla del mig no la travessa cap salt normal.

El que passa són cinc passos:

1. El vostre programa fa una crida a funció **normal i corrent**, un `call`, a una funció que es diu `write` i que és a la llibreria de C. Encara sou en mode usuari i no ha passat res d'especial.

2. Aquesta funció —quatre instruccions, no més— posa els arguments als registres que el kernel espera i el **número** de la crida en un registre a part.
3. Executa **la** instrucció. Una. Que canvia de mode i salta, i que fa les dues coses alhora i sense que res es pugui esmunyir pel mig.
4. Ja en mode kernel, el sistema agafa aquell número, l'usa per indexar una taula i crida el servei corresponent. Aquest codi **no forma part del vostre executable**: hi era abans que arrenqués.
5. Torna, es restaura el mode usuari, i la funció de la libc mira el resultat i us el dona.

La paraula per al pas 3 és **trap**, i és el nom genèric: cada arquitectura té la seva instrucció i no es diuen igual enlloc.

La llibreria de C *no és* el sistema operatiu. S'executa en mode usuari, forma part del vostre executable i el podeu desassemblar. L'única cosa que fa és saber quins registres toca omplir en aquesta màquina.

Un avís abans del primer **strace**, perquè no us espanti el que hi veureu. Aquest llibre va de Unix, i la norma de Unix és POSIX: la llista de crides al sistema que estudiareu és la que diu POSIX, i cada sistema l'implementa com li sembla. Linux, que és el que teniu al laboratori, sovint prefereix oferir-ne una de sola, més general, i deixar que la llibreria en faci les que POSIX demana. Per això el nom que veureu passar per la frontera no sempre serà el que heu escrit: un **fork()** hi surt com un **clone()**; un **open()**, com un **openat()**; un **stat()**, com un **newfstatat()**; un **pipe()**, com un **pipe2()**. Cada cop que passi, el llibre ho dirà en una nota a peu. Això no canvia què és una crida al sistema —el **fork()** ho és— ni canvia el dibuix d'aquí sobre: hi ha una funció de la llibreria, hi ha un trap, i hi ha codi del kernel a l'altra banda.

2.5.1 Per què el trap i no un call

Podríeu preguntar per què tanta cerimònia, si al capdavall es tracta de saltar a un tros de codi. Hi ha tres raons i les tres són bones:

- Un **call** no canvia de mode. Continuaríeu en mode usuari i el codi del kernel no podria fer res que vosaltres no poguéssiu fer ja.
- Un **call** salta a una *adreça*, i l'adreça del codi del kernel canvia amb cada versió. El vostre programa compilat fa tres anys hauria de saber on és avui.
- Un **call** salta on li digueu, i això vol dir saltar al mig d'una rutina del kernel saltant-se les comprovacions. La instrucció de trap només sap saltar a un lloc, el que el sistema va fixar en arrencar.

El segon problema és el que resol el número. Una crida al sistema no s'identifica amb una adreça sinó amb un **número**, que indexa una taula que el kernel manté i que ha de continuar significant el mateix entre versions. És l'única cosa que un binari compilat pot saber per endavant.

2.6 La taula, i sense cap «depèn»

Ara la comparació, que és el cor del capítol. Primer per conceptes:

	Crida a funció	Crida al sistema
qui n'executa el codi	el vostre procés	el kernel
mode d'execució	usuari tota l'estona	canvia a kernel i torna
on és el codi	al vostre executable	ja hi era abans
com se sap on saltar	l'enllaçador ho va resoldre en compilar	un número que indexa una taula del kernel
qui hi pot dir que no com diu que ha fallat amb què es depura què costa	ningú com vulgui el gdb 2 ns	el kernel, per permisos sempre -1 i l' errno l' strace de 100 a 400 ns

I ara el mateix amb els registres i les instruccions de debò. Hi ha tres columnes i no dues, i el perquè és tot el moll de l'os: n'hi ha *dues* d'ARM64, que són el mateix maquinari amb dos sistemes operatius diferents.

	x86-64 Linux	ARM64 Linux	ARM64 macOS
<i>Crida a funció</i>			
arguments	rdi rsi rdx rcx r8 r9	x0...x7	x0...x7
instrucció	call	bl	bl
resultat	rax	x0	x0
per tornar	ret	ret	ret
<i>Crida al sistema</i>			
arguments	rdi rsi rdx r10 r8 r9	x0...x5	x0...x5
el número va a	rax	x8	x16
el trap és	syscall	svc #0	svc #0x80
resultat	rax	x0	x0
ha fallat si	rax és entre -1 i -4095	x0 és negatiu	el <i>flag de carry</i> està encès
el write és	1	64	4
i el getpid	39	172	20

Aquesta taula es llegeix de dues maneres, i totes dues ensenyen alguna cosa.

Compareu les dues primeres columnes i veureu què decideix l'arquitectura: els noms dels registres, quants n'hi ha i com es diu la instrucció. Tot canvia, i no podia ser d'altra manera, perquè són dos jocs d'instruccions diferents.

Compareu les dues últimes i veureu què decideix el sistema operatiu, perquè aquí el maquinari és exactament el mateix xip. A la meitat de dalt no canvia *res*: els arguments, la instrucció i el retorn són idèntics. A la meitat de baix canvia gairebé tot —el registre del número, el valor del trap, els números de cada crida i fins i tot com es diu que ha fallat.

El conveni de crida a funció el fixa **l'arquitectura**; el de crida al sistema el fixa **el sistema operatiu**. Per això «per a Apple Silicon» no vol dir res mentre no digueu també quin sistema hi corre.

I encara hi ha dues coses amagades a la taula.

La primera: dins de cada columna, **els arguments van als mateixos registres** tant si és una crida a funció com si és una crida al sistema. No és casualitat: està fet a posta perquè la funció de la llibreria no hagi de moure res de lloc.

La segona: a l'x86-64 la crida al sistema fa servir **r10** allà on la crida a funció fa servir **rcx**. Sembla un caprici i no ho és, i és l'excepció que confirma la regla —la instrucció **syscall** esclafa el **rcx**, perquè hi desa l'adreça de tornada, i per tant allà no hi pot haver cap argument.

2.6.1 🕒 Mirem-ho

Res d'això no cal creure-s'ho. Aquí teniu el mateix **write** escrit sense la llibreria, amb una branca per cada columna de la taula:

Llistat 2.1. *frontera-trap*: la crida al sistema a mà

```

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"                                ❸
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;                    ❷
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"                        ❸
32                       "cset x3, cs"                          ❹
33                       : "+r" (x0), "=r" (x3)
34                       : "r" (x16), "r" (x1), "r" (x2)
35                       : "cc", "memory");
36     return x3 ? -x0 : x0;                                    ❺
37
38 #else
39 # error "escrit per a x86-64 i ARM64; per a res més, ←
40     mireu-vos-ho abans"
41 #endif
42
43 int main(void)
44 {
45     const char *m = "he travessat la frontera sense la libc\n";
46     long r;
47
48     r = trap_write(1, m, strlen(m));
49     if (r < 0) {                                              ❻
50         errno = -r;
51         error_i_exit("write", 1);
52     }
53     return 0;
54 }

```

- ❶ La instrucció de l'Intel. Una, i les de sobre només omplen registres.
- ❷ El número del `write` a Linux, que va al `rax`.
- ❸ Els arguments, als mateixos registres que en una crida a funció.

- ④ El `rcx` i l'`r11` els esclafa la instrucció. Ho hem de dir al compilador perquè no hi deixi res que li faci falta.
- ⑤ El mateix número, a l'ARM de Linux, no és 1 sinó 64. I el registre no és el `rax` sinó el `x8`.
- ⑥ I la instrucció és una altra.
- ⑦ El mateix xip, un altre sistema: ara el número és el 4 i el registre és el `x16`.
- ⑧ I el trap duu un argument que a Linux no hi és.
- ⑨ Aquí no hi ha cap resultat negatiu que mirar. El `cset` copia el flag de carry a un registre, que és l'única manera de saber, des del C, si la cosa ha anat bé.
- ⑩ I per això la funció ha de fabricar-lo, el negatiu, per poder tornar el mateix a les tres bandes.
- ⑪ L'error arriba com un negatiu, no com un `-1`. Convertir-lo a `errno` és el que la llibreria fa per vosaltres.

❗ Compte

Mireu-vos els `#if` amb calma, perquè hi ha una trampa i és cara. En un Mac amb Apple Silicon, la macro `__aarch64__` està definida, de manera que si les branques només miressin l'arquitectura, un Mac es quedaria la branca de Linux —i compilaria sense una sola queixa. En executar-lo, això:

```
$ make frontera-trap && ./frontera-trap
cc -Wall -Wextra -O0 -g -o frontera-trap frontera-trap.c
eines.c
zsh: invalid system call ./frontera-trap
```

El xip va executar el `svc` perfectament. Qui no va entendre què volia dir el 64 va ser el sistema operatiu, que va contestar amb un SIGSYS. El mateix codi font i el mateix silici, i mor perquè canvia qui hi mana: la taula de la pàgina anterior descriu això, i aquestes dues línies ho demostren. Per això les branques han de mirar l'**arquitectura i el sistema**, que és el que fa el `&& !defined(__APPLE__)`.

Compileu-lo i mireu-vos-el. Per no haver de buscar la funció entre tota la sortida de l'`objdump`, us podeu fer una petita funció de shell

que només n'ensenyi una:

```
$ dis() { objdump -d --no-show-raw-insn --disassemble=$1 $2 \
> | sed -n "/<$1>:/,/^$/p"; }
$ gcc -O2 -o frontera-trap frontera-trap.c eines.c
$ dis main frontera-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_i_exit>
4010b4:      xor     %eax,%eax
4010b6:      add     $0x18,%rsp
4010ba:      ret
4010bb:      nopl    0x0(%rax,%rax,1)
```

Amb `-O2` el compilador s'ha endut la funció dins del `main`, i aquesta sortida és sencera. Les set primeres línies són tota la crida: l'`$0x1` és el número del `write`; el `$0x402008` és l'adreça del missatge; el `$0x27` és el 39 del `strlen`, que el compilador ha sabut comptar sol; el `mov %rax,%rdi` posa al primer argument el canal 1, aprofitant que ja el tenia; el `syscall` és la frontera, i el `test` i el `js` miren si el resultat és negatiu. Si no ho és, el programa acaba dues línies més avall.

Tota la resta és el camí de l'error. Demana a la llibreria on és l'`errno`, li canvia el signe al resultat (`neg`), l'hi desa, i crida l'`error_i_exit`.

És el globus  del llistat 2.1, i d'aquí a una pàgina el tornareu a trobar dins de la llibreria de C.

I ara la mateixa cosa a l'ARM, amb uns altres noms i el mateix esquelet:

```
$ dis main frontera-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>
```

El 0x40 és el 64 de la taula de Linux i el `svc` és la frontera. La del final, `tbzn x0, #63`, diu «si el bit 63 està encès»: és a dir, si el resultat és negatiu. La mateixa comprovació que a l'altra, dita d'una altra manera.

I si desassembleu la llibreria de C, hi trobareu això mateix, però no pas on l'esperaríeu. Al laboratori, el `__write` de la llibreria no conté cap `syscall`*:

```
$ 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
```

*Amb la `glibc` 2.43, la del laboratori. Amb la 2.39, la d'Ubuntu 24.04, sí que el conté, i la resta d'aquesta secció la trobareu tota dins del `__write`. Vegeu «D'on surten les sortides», al prefaci.

```

112ae3:    xor    %ecx,%ecx
112ae5:    call   97e1e <__syscall_cancel>
112aea:    add    $0x18,%rsp
112aee:    ret

```

Posa el número del `write` (el `push $0x1`) i crida una funció comuna, la mateixa per a totes les crides al sistema que es poden quedar esperant. El motiu és dels threads del capítol 12: un thread que espera dins d'un `write()` ha de poder ser cancel·lat per un altre, i la llibreria ho vigila en un sol lloc en comptes de fer-ho a cada crida. Les versions més antigues de la llibreria no ho feien així, i a l'`__write` hi havia el `syscall` directament; mireu quina teniu.

La funció comuna és la que té la resposta a «com se sap que ha fallat»:

```

$ 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    $0xffffffffffffffff000,%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

```

El `syscall` de debò és un pis més avall, a la funció que crida el

`call`, que abans de travessar mira si algú ha demanat cancel·lar el thread:

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

Les instruccions de després del `call` són tot el misteri de l'`errno`. El kernel torna el codi d'error com un número negatiu petit; la llibreria compara amb `-4096` per veure si hi cau, i si hi cau li canvia el signe (`neg`), el desa on el procés guarda l'`errno` i retorna `-1` —que és el `$0xffffffffffffffff` de la penúltima línia. La regla «totes les crides al sistema tornen `-1`» no és del kernel: és d'aquí. I és el mateix que el compilador havia fet a mà al `frontera-trap`.

2.6.2 🕒 I la tercera columna

Queda la que no podeu mirar des del laboratori, i que és d'on surt el SIGSYS de fa dues pàgines. La llibreria de sistema de macOS es diu `libsystem_kernel.dylib` i el desassemblador, `otool`:

```
$ otool -tV /usr/lib/system/libsystem_kernel.dylib \
  | grep -A 4 '^_write:'
_write:
000000000000482c  mov     x16, #0x4
0000000000004830  svc     #0x80
0000000000004834  b.lo    0x4854
0000000000004838  pacibsp
```

Quatre instruccions, com les altres dues, i les tres diferències de la taula totes juntes: el número va al `x16` i és el 4, i el trap és `svc #0x80`. La tercera és la més bonica. On el Linux comprovava el signe

del resultat, aquí hi ha un **b.lo**, que en ARM vol dir «salta si el flag de carry està *apagat*» —o sigui, salta si **no** hi ha hagut error, i deixa caure cap a la rutina d'error si n'hi ha hagut. El mateix ofici, un altre conveni.

I els números de la taula són literalment això:

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

🔗 Per saber-ne més

Tot això és una demostració, no una tècnica, i els dos sistemes hi diuen coses contràries. A macOS els números de crida **no són una interfície estable**: poden canviar entre versions i l'única cosa que Apple promet és que la llibreria continuarà anant-hi bé. A Linux passa exactament al revés, i és una promesa famosa d'en Torvalds —la taula no es toca mai—, per això hi ha llenguatges que se salten la libc i fan els traps ells mateixos. On cada sistema decideix posar la seva frontera de compatibilitat diu més d'ell que qualsevol detall de registres: Apple la posa a la llibreria i Linux al kernel.

2.7 Quant costa travessar-la

Queda saber si tot això es nota. Tres coses que des del C s'escriuen igual:

Llistat 2.2. frontera-cost: què val de travessar

```
1 int main(int argc, char *argv[])
2 {
3     long n = (argc > 1) ? atol(argv[1]) : REPETICIONS;
4     struct timespec t;
5     double t0;
6     char buf[128];
```

```

7     long i;
8
9     if (n <= 0)
10         n = REPETICIONS;
11
12     t0 = ara();
13     for (i = 0; i < n; i++)
14         escombraries = suma(i, 1);
15     sprintf(buf, "crida a funcio ..... %6.1f ns\n",
16             (ara() - t0) * 1e9 / n);
17     write(1, buf, strlen(buf));
18
19     t0 = ara();
20     for (i = 0; i < n; i++)
21         clock_gettime(CLOCK_MONOTONIC, &t);
22     sprintf(buf, "clock_gettime (vDSO) . %6.1f ns\n",
23             (ara() - t0) * 1e9 / n);
24     write(1, buf, strlen(buf));
25
26     t0 = ara();
27     for (i = 0; i < n; i++)
28         escombraries = getpid();
29     sprintf(buf, "getpid (travessa) .... %6.1f ns\n",
30             (ara() - t0) * 1e9 / n);
31     write(1, buf, strlen(buf));
32
33     return 0;
34 }

```

- ❶ Tres bucles iguals, i l'única cosa que canvia és què hi ha a dins. El primer crida una *suma* nostra de dues línies, que no surt del procés.
- ❷ Aquesta *sembla* una crida al sistema i no ho és.
- ❸ I aquesta ho és, i és de les barates: el kernel ja té el número a mà. O sigui que el que es mesura és el viatge, no el servei.

```

$ ./frontera-cost
crida a funcio .....    1.8 ns
clock_gettime (vDSO) .   13.3 ns
getpid (travessa) ....  410.1 ns

```

Més de dues-centes vegades. I `getpid` és la crida al sistema més barata que hi ha: no fa res, només va i torna. Qualsevol de les que fan alguna cosa costa molt més, i el capítol 9 en tornarà a parlar quan compari quatre-centes mil crides amb cent.

El número de la darrera línia depèn molt de la màquina, i la comparació la podeu fer vosaltres. En dues aules del laboratori, amb la **mateixa** distribució i el mateix kernel, el mateix programa dona 410 ns en una i 73 ns en l'altra. La diferència no és la velocitat del processador —el segon és només una mica més nou—: és que el primer és dels afectats pel Meltdown i el kernel, per protegir-lo, fa més feina a cada viatge. Ho diuen les línies **Vulnerability** del `lscpu`: on una màquina diu **Meltdown: Mitigation: PTI**, l'altra diu **Not affected**. Quan una frontera és insegura, el que s'encareix és travessar-la.

Compte

El del mig és el cas interessant. El `clock_gettime()` no travessa la frontera: el kernel ha deixat una còpia del codi i de l'hora dins de la memòria de cada procés —el *vDSO*— justament perquè és una crida tan freqüent que no es pot permetre el viatge. O sigui que hi ha coses que sembla que siguin crides al sistema i no ho són. Al llibre les trobareu en vermell igualment, perquè POSIX diu que ho són; el que canvia és per on passen.

2.8 Exercicis

1. Compileu un programa que faci un `printf` i un altre que faci un `write()`, i passeu-los tots dos per l'`strace`. Quantes crides al sistema fa cadascun? Per què no és el mateix número?
2. Desassembleu la llibreria de C de la vostra màquina i busqueu-hi el `getpid`. Quantes instruccions té? Quina és el trap?
3. Escriviu un programa d'una línia que executi una instrucció privilegiada: `__asm__("hlt")` si sou en un Intel, `__asm__("msr daifset, #2")` si sou en un ARM. Compila? Executa? Amb

quin senyal mor, i per què no és el mateix als dos? Mireu també el **dmmsg**: qui us ha aturat hi ha deixat una nota.

4. Al llistat 2.1 hi ha un número, el del **write**. Busqueu on és escrit al vostre sistema i comproveu que és el que diem. A Linux, mireu `/usr/include/asm/unistd_64.h`.
5. Feu-hi passar un descriptor que no existeixi, per exemple el 99. Què torna la funció? I què hauria tornat el **write** de la llibreria?
6. Executeu el **frontera-cost** a la màquina del laboratori i a la vostra. Els tres números canvien igual, o n'hi ha un que canvia més? Què en dedueu.