# Elements de Sistemes Operatius

Capítol 8

## On viu un procés

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 · 16 d'octubre de 2026*

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© 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.

# 8 On viu un procés

Al capítol 5 vam dir que el `fork()` li dóna al fill una còpia de la memòria del pare, i una nota al peu prometia que de debò no se'n fa cap. Vam dir també que el PCB conté la descripció de l'espai d'adreces del procés, sense dir què vol dir això. I al capítol 4 vam veure que el C us deixa sumar 1 a una adreça i escriure on no toca, sense explicar què impedeix que allò s'endugui la màquina per davant.

Tot això és el mateix tema: on és, de debò, la memòria d'un procés.

## 8.1 Dues menes d'adreça

Comencem pel problema, que es veu millor mirant un executable. Al capítol 4 el `gcc` ens va fer un fitxer; el `objdump` ens diu què hi ha a dins*:

```
alumne@labSO:~/S5$ objdump -d prova
0000000000401116 <lafuncio>:
  401116:  55                push   %rbp
  401117:  48 89 e5          mov    %rsp,%rbp
  40111a:  90                nop
  40111b:  5d                pop    %rbp
  40111c:  c3                ret

000000000040111d <main>:
  40111d:  55                push   %rbp
  40111e:  48 89 e5          mov    %rsp,%rbp
  401121:  e8 f0 ff ff ff    call   401116 <lafuncio>
  ...
```

*Totes les sessions d'aquest capítol són d'una aula amb openSUSE, que són la majoria i les que fareu servir a aquesta assignatura. A l'aula amb Ubuntu les adreces no comencen igual; l'avís de la pàgina 8 diu per què.

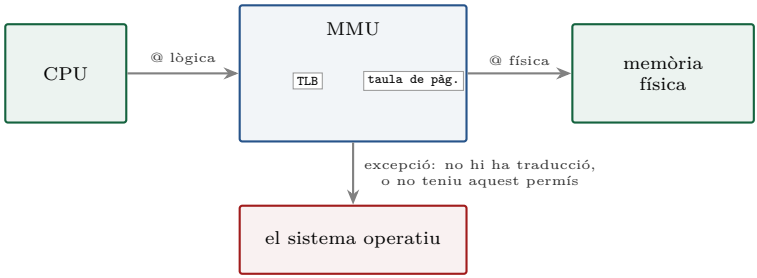
Fixeu-vos en la línia del `call`. El compilador ha escrit dins del programa l'adreça `401116`: la de la funció a la qual s'ha de saltar. Ha hagut de decidir un número i posar-lo allà.

I aquí ve la pregunta incòmoda. El programa es pot executar tres vegades alhora, en una màquina on hi ha altres cinquanta processos i on la memòria lliure és diferent cada cop. Com pot ser que el compilador sàpiga, mesos abans, quina posició de memòria estarà lliure?

No ho sap.

> Les adreces que el compilador escriu al programa són **adreces lògiques**: un sistema de numeració privat de cada procés, que comença sempre igual i que no diu res sobre on és la memòria de debò. Les posicions reals de la memòria són les **adreces físiques**, i qui les assigna és el sistema operatiu, en el moment de posar el programa en marxa.

Entre les unes i les altres hi ha d'haver algú que tradueixi, i ha de ser prou ràpid per fer-ho a cada accés a memòria — és a dir, unes quantes vegades per instrucció. Això no ho pot fer programari: ho fa el maquinari, i la peça es diu **MMU**, *memory management unit*.




L'única adreça que el programa coneix és la lògica. Qui sap on és de debò és la MMU, i qui l'hi ha dit és el sistema operatiu, quan va assignar la memòria.

**Figura 8.1.** Qui pregunta, qui respon, i què passa quan no es pot respondre.

Del *com* ho fa la MMU no en parlarem gaire, perquè ja ho heu

vist: l'espai lògic es parteix en *pàgines* de mida fixa, la memòria física en *marcs* de la mateixa mida, i una taula de pàgines per procés diu quin marc correspon a cada pàgina, amb quins permisos i si hi és. Els processadors hi afegeixen una memòria associativa, la TLB, per no haver de consultar la taula a cada accés. El que ens interessa aquí és una altra cosa: què passa quan la MMU **no** pot traduir.

> Quan la MMU no pot completar una traducció — perquè l'adreça no és del procés, perquè no en té el permís, o perquè encara no hi ha cap marc assignat — genera una excepció i el sistema operatiu l'ha de resoldre. De vegades matant el procés, i de vegades fent la feina que faltava.

Aquesta frase és la clau de tot el capítol. La meitat de les coses que el kernel fa amb la memòria consisteixen a no fer-les fins que la MMU es queixa.

## 8.2 El mapa d'un procés

L'espai lògic d'un procés no és una massa uniforme: està repartit en **regions**, cadascuna amb una mida, un origen i uns permisos.

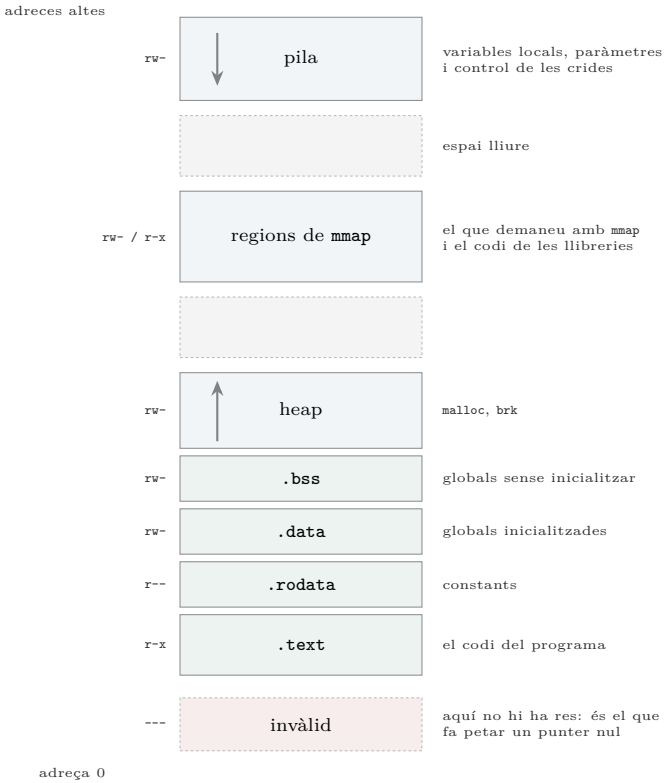



**Figura 8.2.** L'espai lògic d'un procés. Les regions de baix vénen de l'executable; les de dalt es fabriquen en temps d'execució.

Aquest dibuix diu quatre coses.

**Les regions de baix vénen del fitxer.** Un executable no és un bloc: és un format — a Unix, l'ELF — amb **seccions** declarades a la capçalera. La `.text` porta el codi; la `.data`, les variables globals amb valor inicial; la `.rodata`, les constants. La `.bss` és el cas curiós: són les globals *sense* inicialitzar, i com que totes valen zero, al fitxer només se'n guarda la mida. Amb un `objdump -h` les veureu totes.

**Les de dalt no hi són, al fitxer.** La pila i el heap es fabriquen quan el programa s'executa i canvien de mida mentre dura. Creixen l'una cap a l'altra: la pila cap a adreces baixes, cada vegada que crideu

una funció; el heap cap a altes, cada vegada que demaneu memòria.

**Cada regió té permisos.** El codi és de lectura i execució però no d'escriptura, i per això no us podeu reescriure el programa a mitja execució. Les constants són de només lectura, i per això escriure sobre una cadena literal peta. La MMU els fa complir pàgina a pàgina, i és la mateixa MMU que fa la traducció.

**I l'adreça 0 no és de ningú.** La primera pàgina es deixa sense assignar a posta, de manera que qualsevol accés a través d'un punter nul provoca l'excepció i el procés mor amb un `SIGSEGV`* en comptes de trepitjar el codi.

### 8.2.1 Mirar-s'ho

Res d'això és teoria: el kernel us ensenya el mapa si l'hi demaneu. Aquest programa escriu l'adreça de mitja dotzena de coses seves i es queda quiet, perquè el pugueu comparar amb el mapa de debò:

**Llistat 8.1.** On viu cada cosa

```
1  int global_iniciada = 5;            /* secció .data   */
2  int global_buida;                   /* secció .bss    */
3  const char constant[] = "text";     /* secció .rodata */
4
5  static void on(const char *que, const void *adreca)
6  {
7      char buf[96];
8
9      sprintf(buf, "%-24s %p\n", que, adreca);
10     write(1, buf, strlen(buf));
11 }
12
13 int main(void)
14 {
15     int local = 1;                                  ❶
16     int *dinamica = malloc(sizeof(int));            ❷
17     char buf[80];
18
```

*Del capítol 7. Si no captureu el signal, el tractament per defecte és matar el procés i deixar-ne un fitxer `core`.

```
19      if (dinamica == NULL)
20          error_i_exit("malloc", 1);
21
22      on("codi (main)", (void *)main);                ❸
23      on("constant (.rodata)", constant);
24      on("global iniciada (.data)", &global_iniciada);
25      on("global buida (.bss)", &global_buida);
26      on("heap (malloc)", dinamica);
27      on("final del heap", sbrk(0));                  ❹
28      on("pila (local)", &local);
29
30      sprintf(buf, "\nsóc el %d: mireu /proc/%d/maps\n",
31              (int)getpid(), (int)getpid());
32      write(1, buf, strlen(buf));
33      sleep(30);
34
35      free(dinamica);
36      return 0;
37  }
```

❶ Una variable local: viu a la pila.
❷ `malloc` torna una adreça del heap. D'això en parlem a la secció següent.
❸ El nom d'una funció és l'adreça de la seva primera instrucció: la mateixa que ens ha ensenyat l'`objdump`.
❹ `sbrk()` amb zero no demana res: només diu on s'acaba el heap ara mateix.

```
alumne@labSO:~/S5$ ./mem-mapa &
codi (main)               0x401217
constant (.rodata)        0x402008
global iniciada (.data)   0x404060
global buida (.bss)       0x404068
heap (malloc)             0x39bfa010
final del heap            0x39c1b000
pila (local)              0x7ffec08e7b74

sóc el 8385: mireu /proc/8385/maps
alumne@labSO:~/S5$ cat /proc/8385/maps
00400000-00401000 r--p ...  /home/alumne/S5/mem-mapa
00401000-00402000 r-xp ...  /home/alumne/S5/mem-mapa
00402000-00403000 r--p ...  /home/alumne/S5/mem-mapa
00403000-00404000 r--p ...  /home/alumne/S5/mem-mapa
```

```
00404000-00405000 rw-p ...  /home/alumne/S5/mem-mapa
39bfa000-39c1b000 rw-p ...  [heap]
7f5f4e400000-7f5f4e428000 r--p ...  /usr/lib64/libc.so.6
7f5f4e428000-7f5f4e59e000 r-xp ...  /usr/lib64/libc.so.6
...
7ffec08c8000-7ffec08ea000 rw-p ...  [stack]
```

Ara aparelleu-ho. El `main` és a `0x401217`, que cau dins de la línia `r-xp` de l'executable: lectura i execució, sense escriptura, com tocava. La constant cau a la línia `r--p` de sota. Les dues globals cauen totes dues a l'única línia `rw-p` del programa. El `malloc` cau dins de `[heap]`, i el final del heap coincideix exactament amb el final d'aquella línia. I la variable local és a `[stack]`, a més de cent terabytes de distància de tota la resta.

**Qüestió 1**

Al mapa hi ha cinc línies de l'executable i al programa només n'hi ha quatre menes de cosa. I pel mig hi surt la `libc`, que nosaltres no hem demanat enlloc. D'on ha sortit?

**Resposta**

De les llibreries dinàmiques, que veurem més avall, als quatre trucs. El vostre executable no porta el codi de la `libc` a dins: porta una nota que diu que la necessita, i és en arrencar que algú la mapeja a l'espai d'adreces del procés. Aquell «algú» també és un programa, i també el veureu al mapa: `ld-linux.so`.

## 8.3 Què passa dins d'un `exec()`

Amb això ja podem completar el que el capítol 5 va deixar a mitges. Quan un procés fa `execlp()`, el kernel:

1. **Interpreta el fitxer:** llegeix la capçalera ELF i mira quines seccions té, de quina mida i on són dins del fitxer.

2. **Defineix l'espai lògic:** decideix quines adreces lògiques ocupa cada regió i amb quins permisos, i s'ho apunta al PCB.

3. **Assigna memòria física** i configura la MMU perquè sàpiga traduir.

4. **Hi porta el contingut:** llegeix del disc les seccions i les escriu a la memòria assignada.

5. **Posa el comptador de programa** a la primera instrucció.

I aquí el procés continua, amb un altre programa a dins. Del pas 4, més endavant en direm que és mentida: no es llegeix tot, i gairebé mai.

**Compte**

Les adreces inicials de les regions no són sempre les mateixes. Si compareu dues execucions del `mem-mapa` veureu que alguns números ballen: el sistema hi aplica un desplaçament aleatori a posta, perquè un atacant que vulgui saltar a una adreça concreta no la pugui endevinar. A les aules amb openSUSE el codi i les globals no es mouen —el compilador hi fa executables que s'han de carregar sempre a `0x400000`—, però el heap, les llibreries i la pila sí. A l'aula amb Ubuntu, la de SO2 i CASO, el compilador fa executables *PIE*, que es poden carregar a qualsevol lloc: allà els números comencen per `0x55` i es mou tot, inclòs el codi. Per això, i per la resta del capítol, l'única cosa que podeu donar per segura d'una adreça és a quina regió cau.

## 8.4 Memòria dinàmica

Fins ara, tota la memòria que hem fet servir estava decidida abans d'executar: les globals les dimensiona el compilador i les locals les dimensiona la crida a la funció. Però hi ha coses que no se saben fins que el programa s'executa. Si el programa rep un número per la línia de comandes i n'ha de fer aquell nombre de fills, quants `int` li fan falta?

Fixar-ho en compilar té dues respostes i totes dues són dolentes: o en reserveu massa i el malbarateu, o en reserveu pocs i peta. La bona és demanar-ne quan se sap.

### 8.4.1 `brk()` i `sbrk()`: moure el límit

La manera clàssica és moure el límit superior del heap.

**Crida al sistema** `sbrk`

```
void *sbrk(intptr_t bytes);
```

`sbrk()` desplaça el límit del heap tants bytes com li digueu i us torna *on era abans*, que és precisament l'adreça del tros que acabeu de guanyar*. Amb un valor negatiu el fa recular; amb zero no el mou i només us diu on és.

**Llistat 8.2.** El límit del heap, a mà

```
1  static void limit(const char *quan)
2  {
3      char buf[96];
4
5      sprintf(buf, "%-26s %p\n", quan, sbrk(0));      ❶
6      write(1, buf, strlen(buf));
7  }
8
9  int main(void)
10 {
11     char buf[96];
12     void *meva;
13
14     limit("a l'inici");
```

*El `sbrk` és l'excepció d'aquest llibre: no és a POSIX —hi constava com a LEGACY i en va acabar sortint— i a Linux tampoc no és una crida. La crida és el `brk()`, que vol l'adreça nova del límit, sencera; el `sbrk` és la comoditat de demanar-ho en bytes i que us diguin on era. Surt al llibre perquè és el que trobareu escrit a tot arreu, i perquè s'explica sol.

```
15
16      meva = sbrk(4096);                                  ❷
17      if (meva == (void *)-1)
18          error_i_exit("sbrk", 1);
19      sprintf(buf, "%-26s %p\n", "sbrk(4096) m'ha tornat", meva);
20      write(1, buf, strlen(buf));
21      limit("un cop fet el sbrk");
22
23      if (sbrk(-4096) == (void *)-1)                      ❸
24          error_i_exit("sbrk", 1);
25      limit("un cop tornats");
26
27      free(malloc(1));                                    ❹
28      limit("un cop fet malloc(1)");
29      return 0;
30  }
```

❶ `sbrk()` amb zero és la manera de preguntar sense demanar.
❷ Demanem una pàgina. El que torna és l'adreça del principi.
❸ I la tornem.
❹ Aquest `malloc` d'un sol byte és el més interessant del programa: mireu on acaba deixant el límit.

```
alumne@labSO:~/S5$ ./mem-sbrk
a l'inici                   0x326e7000
sbrk(4096) m'ha tornat      0x326e7000
un cop fet el sbrk          0x326e8000
un cop tornats              0x326e7000
un cop fet malloc(1)        0x32708000
```

Un `malloc` d'un byte ha mogut el límit 132 kB. No és cap error: la llibreria en demana molts de cop perquè no li calgui tornar a molestar el kernel a la propera.

**Compte**

Aquest programa barreja `sbrk()` i `malloc` a posta, per ensenyar que tots dos toquen el mateix límit. En un programa de debò

no ho feu mai: la llibreria dóna per fet que el heap és seu, i moure'l per sota seu és una manera excellent de tenir errors que no s'entenen.

`sbrk()` és honest i incòmode. Us dóna bytes crus, no sap què hi guardeu, i tota la comptabilitat — què està lliure, què es pot tornar, quan — queda de la vostra banda. Per això gairebé ningú no el fa servir directament.

### 8.4.2 `malloc`: l'excepció

**Funció de llibreria** `malloc i free`

```
void *malloc(size_t bytes);
void  free(void *regio);
```

`malloc` i `free` **no són crides al sistema**: són funcions de la llibreria de C, i és per això que en aquest llibre no surten en vermell. El que fan és la comptabilitat que el `sbrk()` no fa.

Quan li demaneu memòria, `malloc` mira si al heap ja n'hi ha un tros lliure prou gran; si n'hi ha, us el marca com a ocupat i us en torna l'adreça, sense cap crida al sistema pel mig. Si no n'hi ha, ha de demanar-ne al kernel, i aquí fa una cosa que heu de saber: **tria entre dues crides al sistema segons la mida**. Per a les peticions normals mou el límit del heap amb `brk()` — i en demana de més, per no haver de tornar a molestar el kernel de seguida. Per a les grosses, a partir d'un llindar que el sistema fixa*, ni tan sols toca el heap: en demana una regió a part amb `mmap()`. I el `free` tampoc no fa sempre el mateix: el que ve del heap es marca com a lliure a les estructures de la llibreria i no es torna al sistema — ja es reaprofitarà —, mentre que el que venia d'un `mmap()` es torna de debò amb `munmap()`.

*A la `glibc` el llindar és configurable i, per defecte, és de l'ordre dels 128 kB. Es pot consultar i canviar amb `mallopt(M_MMAP_THRESHOLD, ...)`.

El heap que veieu al `/proc/PID/maps` no és el que el vostre programa està fent servir: és el que la llibreria té demanat. Entre les dues coses hi ha tot el que `malloc` s'ha guardat de reserva.

### 8.4.3 Per què el `malloc` sí i el `printf` no

Aquesta pregunta l'heu de tenir a la punta de la llengua. A tot el llibre hem escrit amb `write()` i mai amb `printf`, amb l'argument que `printf` és una funció de llibreria i que ens amaga el sistema; i ara resulta que fem servir el `malloc`, que també ho és.

No és cap incoherència: és la mateixa regla, aplicada dues vegades.

El `printf` **substitueix** la crida al sistema que volem ensenyar: hi posa un buffer al mig que decideix pel seu compte quan s'escriu de debò, i aquell buffer amaga precisament allò que estudiem — en quin ordre passen les coses quan hi ha més d'un procés escrivint. El `malloc` no substitueix res: **és el client** de les crides que volem ensenyar. Per sota fa `brk()` o `mmap()` — l'una o l'altra segons la mida que li demaneu —, i el que hi afegeix, que és agrupar peticions i recordar què està lliure, és feina que el kernel no fa i que algú ha de fer igualment.
La regla, doncs: **evitem el que ens amaga el mecanisme i acceptem el que s'hi recolza.**

El `sprintf` és al segon grup, i per això el fem servir sense problemes: formata text en una taula de `char` i s'atura allà. El `write()` que ve després continua sent nostre, i és on es veu el sistema.

### 8.4.4 `mmap()`: la manera moderna

**Crida al sistema** `mmap i munmap`

```
void *mmap(void *addr, size_t bytes, int prot,
           int flags, int fd, off_t offset);
int   munmap(void *addr, size_t bytes);
```

`mmap()` no mou cap límit: crea una regió nova, on el sistema vulgui, de la mida que digueu i amb els permisos que digueu. Té sis paràmetres i fa por, però tres són zero gairebé sempre.

**`addr`** on la voleu. Amb `NULL` ho decideix el sistema, que és el que voleu el 99 % de les vegades.

**`bytes`** quanta.

**`prot`** els permisos: `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, o `PROT_NONE`. Són els mateixos que després llegireu al `maps`.

**`flags`** de què s'ompla i qui la veu. `MAP_ANONYMOUS` vol dir «memòria en blanc»; sense ell, la regió s'omple amb el contingut d'un fitxer. I `MAP_PRIVATE` o `MAP_SHARED` decideixen si els canvis els veu algú més.

**`fd` i `offset`** quin fitxer i des d'on. Amb `MAP_ANONYMOUS` s'ignoren i s'hi posa $-1$ i 0.

Per demanar memòria i prou, doncs, la fórmula és sempre aquesta:

**Llistat 8.3.** Memòria fora del heap

```
1 /* Els BSD, i per tant el macOS, en diuen MAP_ANON. La ↩
     plataforma de
2    referència d'aquest llibre és Linux (vegeu ↩
     codi/PORTABILITAT.md),
3    però aquesta línia és gratis. */
4 #ifndef MAP_ANONYMOUS
```

```c
#define MAP_ANONYMOUS MAP_ANON
#endif

#define QUANTS 4096
int main(void)
{
    char buf[96];
    int *regio;

    regio = mmap(NULL, QUANTS,                          ❶
                 PROT_READ | PROT_WRITE,                ❷
                 MAP_ANONYMOUS | MAP_PRIVATE,           ❸
                 -1, 0);
    if (regio == MAP_FAILED)                            ❹
        error_i_exit("mmap", 1);

    regio[0] = 42;
    sprintf(buf, "regió a %p, hi he desat un %d\n",
            (void *)regio, regio[0]);
    write(1, buf, strlen(buf));
    sprintf(buf, "el heap continua acabant a %p\n", sbrk(0));
    write(1, buf, strlen(buf));

    if (munmap(regio, QUANTS) < 0)                      ❺
        error_i_exit("munmap", 1);
    return 0;
}
```

❶ `NULL`: decideix-ho tu.
❷ Lectura i escriptura. Si demanéssiu només `PROT_READ`, la línia que ve després us mataria el procés.
❸ Memòria en blanc i per a mi tot sol.
❹ `mmap()` no torna `NULL` quan falla, sinó `MAP_FAILED`. És una de les poques crides que no segueixen el conveni del $-1$.
❺ I es torna quan ja no cal. A diferència del `free`, això sí que ho torna al sistema de debò.

```
alumne@labSO:~/S5$ ./mem-mmap
regió a 0x7f3fe4d7d000, hi he desat un 42
el heap continua acabant a 0x39e17000
```

Compareu els dos números amb el dibuix de la figura 8.2: la regió nova no és al heap, és molt més amunt, entre el heap i la pila, al costat de les llibreries.

Per què preferim aquesta i no el `brk()`? Perquè el `brk()` només sap fer una cosa — moure un límit — i el `mmap()` en sap fer quatre. Us deixa triar els permisos regió a regió; us deixa alliberar del mig sense haver d'esperar que s'alliberi tot el que hi ha per sobre; us deixa omplir la regió amb un fitxer, cosa que veurem al capítol 9; i us deixa compartir-la, que és la secció següent. El `brk()` no era una mala idea el 1975; simplement ha quedat petit.

## 8.5 Compartir memòria

Per defecte, un procés no pot tocar la memòria d'un altre. És la protecció més bàsica que hi ha, i és la MMU qui la fa complir: si una adreça no és a la vostra taula de pàgines, no existeix per a vosaltres.

Però de vegades justament el que voleu és compartir-la, i llavors ho heu de demanar. El flag és el que ho diu:

**Llistat 8.4.** Un flag de diferència

```
1  #ifndef MAP_ANONYMOUS
2  #define MAP_ANONYMOUS MAP_ANON
3  #endif
4
5  int main(int argc, char *argv[])
6  {
7      char buf[96];
8      int *valor, flag;
9      pid_t ret;
10
11     if (argc != 2)
12         Usage(argv[0]);
13     if (strcmp(argv[1], "shared") == 0)
14         flag = MAP_SHARED;                          ❶
15     else if (strcmp(argv[1], "private") == 0)
16         flag = MAP_PRIVATE;
17     else
```

```
18          Usage(argv[0]);
19
20      valor = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
21                   MAP_ANONYMOUS | flag, -1, 0);
22      if (valor == MAP_FAILED)
23          error_i_exit("mmap", 1);
24
25      *valor = 100;                                          ❷
26
27      if ((ret = fork()) < 0)
28          error_i_exit("fork", 1);
29      if (ret == 0) {
30          *valor = 42;                                       ❸
31          exit(0);
32      }
33      waitpid(ret, NULL, 0);
34
35      sprintf(buf, "%s: el pare llegeix %d\n", argv[1], *valor);
36      write(1, buf, strlen(buf));                            ❹
37      munmap(valor, sizeof(int));
38      return 0;
39  }
```

❶ L'única línia que canvia entre els dos casos.
❷ El pare hi escriu un 100 abans de tenir cap fill.
❸ El fill hi escriu un 42.
❹ I el pare mira què hi ha.

```
alumne@labSO:~/S5$ ./mem-compartida private
private: el pare llegeix 100
alumne@labSO:~/S5$ ./mem-compartida shared
shared: el pare llegeix 42
```

Amb `MAP_PRIVATE`, la regió es comporta com la resta de la memòria del procés: el `fork()` en fa una còpia per al fill, i cadascú escriu a la seva. Amb `MAP_SHARED`, les dues taules de pàgines apunten al mateix marc, i el que hi escriu un el veu l'altre immediatament.

Això ja és comunicació entre processos, i és la més ràpida que hi ha: no hi ha còpia, ni crida al sistema, ni kernel pel mig. Escriure a la memòria compartida és escriure a memòria.
El preu és que no hi ha res que sincronitzi res. Si tots dos hi escriuen alhora, el resultat és el que sigui. Amb les pipes del capítol 10 això no passa, perquè el kernel hi és; aquí no hi és, i us el jugueu vosaltres.

## 8.6 Els quatre trucs

Tot el que hem explicat fins ara ja funcionaria. El que ve ara no cal per a res, tècnicament: són quatre optimitzacions. Passa que sense elles la màquina que teniu al davant no aguantaria ni deu minuts, i totes quatre surten de la mateixa idea, que és la de la primera secció: *no facis res fins que la MMU no es queixi.*

### 8.6.1 Càrrega sota demanda

Vam dir que l'`exec()` llegeix les seccions del disc i les escriu a la memòria assignada. No ho fa.

El que fa és definir les regions i deixar-les sense marcs assignats: la MMU no en té traducció. La primera vegada que el programa toca una pàgina, salta l'excepció; el kernel mira al PCB, veu que aquella adreça és vàlida però que no està carregada, li assigna un marc, hi porta el tros de fitxer que toca, actualitza la MMU i **reprèn la instrucció que havia fallat**, que aquest cop funciona. El programa no s'assabenta de res.

I el guany és doble: arrencar és molt més ràpid, perquè no s'espera a llegir un executable sencer, i d'un programa de dos megabytes del qual només se n'executa una funció, només se'n carrega aquella funció.

### 8.6.2 Llibreries dinàmiques

Penseu quants programes de la vostra màquina fan servir la `libc`. Si cadascun en portés una còpia dins de l'executable, seria un disc ple de còpies idèntiques, i una memòria plena de còpies idèntiques.

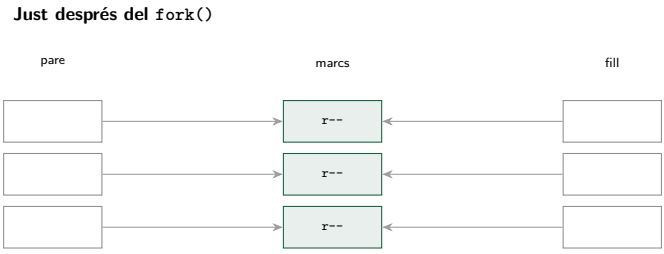
Per això, per defecte, el codi de les llibreries no forma part de l'executable: hi ha només una nota que diu quines funcions en necessita. En arrencar, el kernel llança un altre programa — l'enllaçador dinàmic, que al mapa surt com `ld-linux.so` — que mapeja les llibreries a l'espai d'adreces del procés amb `mmap()`, i com que aquell codi és de només lectura, tots els processos que la facin servir en comparteixen els mateixos marcs.

Una còpia al disc, una còpia a la memòria, i actualitzar-la no obliga a recompilar res. Amb `gcc -static` obtindríeu el comportament antic; feu-ho una vegada i compareu la mida dels dos executables.

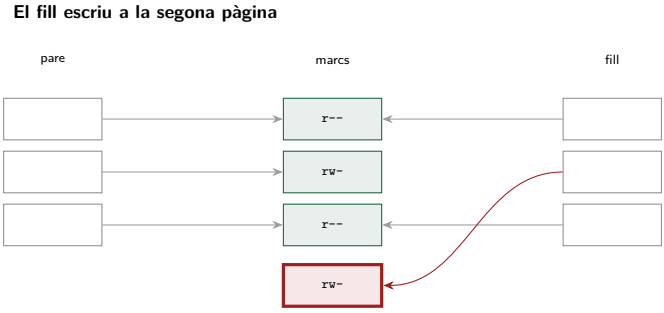
### 8.6.3 Copy on write

Ara ja podem pagar el deute del capítol 5. Vam dir que el fill d'un `fork()` rep una còpia de la memòria del pare. És veritat des de dins del programa, i és fals des de dins del kernel.

Copiar l'espai d'adreces sencer d'un procés seria caríssim, i seria malbaratat gairebé sempre: al capítol 5, la meitat dels nostres exemples feien un `fork()` i tot seguit un `execlp()`, que llença tot allò que s'acabava de copiar.




Cap còpia. Les dues taules de pàgines apunten als mateixos marcs, i el kernel els ha marcat tots de *només lectura*, també al pare.




La MMU no el deixa escriure i avisa el kernel, que copia *aquella* pàgina i prou, torna els permisos i reprèn la instrucció. Les altres dues segueixen compartides.

**Figura 8.3.** El `fork()` no copia res. La còpia es fa pàgina a pàgina, i només de les que algú toca.

El mecanisme és el mateix de sempre. En comptes de copiar, el kernel fa que les dues taules de pàgines apuntin als mateixos marcs i els marca tots dos de **només lectura** — també els del pare, que fins ara hi podia escriure. Mentre tots dos només llegeixin, no passa res i hi ha una sola còpia. Quan un dels dos escriu, la MMU no el deixa i avisa el kernel; el kernel mira les seves estructures, veu que allò és una pàgina de copy-on-write, copia *aquella* pàgina, torna els permisos d'escriptura — als dos processos, i només d'aquella pàgina — i reprèn la instrucció.

Linux ho aplica també a la memòria dinàmica que demaneu i encara no heu tocat: totes les pàgines noves apunten a un mateix marc ple de zeros, de només lectura, fins que hi escriviu.

**Qüestió 2**

Un procés ocupa set pàgines: tres de codi, dues de dades, una de pila i una de heap. Fa un `fork()` i, tot seguit, el fill fa un `execlp()` d'un programa que n'ocupa deu. Quantes pàgines de memòria física s'han arribat a gastar de més, amb copy-on-write i sense?

**Resposta**

Sense copy-on-write, en el moment del `fork()` se'n copien set, i tot seguit l'`execlp()` les llença totes i en demana deu: set pàgines de feina i de memòria completament perdudes.
Amb copy-on-write, entre el `fork()` i l'`execlp()` no se n'ha copiat cap. El fill acaba amb les seves deu i el pare conserva les seves set, que és on s'havia d'arribar. Aquest cas — `fork()` i `execlp()` seguits — és el més freqüent de tots: és el que fa el *shell* cada vegada que hi escriviu una comanda.

### 8.6.4 Memòria virtual

L'últim truc va en direcció contrària: no és portar coses a memòria quan calen, sinó **treure'n** les que no calen.

Si la memòria física s'acaba, el kernel tria una pàgina que li sembla que no es farà servir aviat, la desa en una zona de disc — l'àrea de *swap* — i cedeix el marc a qui el necessita. Si algú torna a tocar aquella pàgina, la MMU no la pot traduir, i ja sabeu com continua: excepció, el kernel busca un marc lliure, hi torna a portar el contingut des del disc, actualitza la MMU i reprèn la instrucció. Això és una **fallada de pàgina**.

La conseqüència és grossa: la suma dels espais lògics de tots els processos pot ser més gran que la memòria de la màquina, i fins i tot un sol procés hi pot tenir un espai més gran. La memòria deixa de ser un límit dur i passa a ser una qüestió de velocitat.

**Compte**

De velocitat, però, molta. Un accés a una pàgina resident són nanosegons; un a una que és al disc són sis ordres de magnitud més. Si el sistema es passa i no aconsegueix mantenir en memòria ni el mínim que cada procés necessita per avançar, entra en *thrashing*: es passa el dia intercanviant pàgines i no executa res. La màquina no s'ha penjat, però és com si. La sortida és treure-hi processos, i és el mateix sistema qui ho ha de detectar mirant-se la taxa de fallades de pàgina.

Perquè això passi poques vegades, el kernel tria bé les víctimes — aproximant l'algorisme que treu la pàgina que fa més temps que ningú no toca — i fa *prefetch*: si veu que aneu llegint pàgines seguides, en porta unes quantes per endavant.

## 8.7 Què hi ha, doncs, a Linux

Recollint-ho tot, i quedant-nos amb el que s'ha de recordar:

- Memòria virtual basada en **paginació***, amb un algorisme de reemplaçament que aproxima el «fa més temps que no es fa servir» i amb prefetch seqüencial.
- **Càrrega sota demanda** de tot: de l'executable i de les llibreries.
- **Copy-on-write** a nivell de pàgina, tant al `fork()` com a la memòria dinàmica acabada de demanar.
- **Llibreries compartides** per defecte.
- I, al PCB, l'adreça base de la taula de pàgines del procés. Al canvi de context del capítol 6 s'hi carrega la del procés que entra i s'invalida la TLB, que és una part gens petita del que aquell canvi costa.

*Sobre Intel el maquinari obliga a passar per la segmentació, però Linux la configura de manera que no fa res i treballa només amb pàgines.

## 8.8 Exercicis

1. Executeu el `mem-mapa` dues vegades i compareu les adreces. Quines canvien i quines no? Ara torneu-ho a fer amb `setarch -R ./mem-mapa`, que desactiva l'aleatorietat.

2. Al `mem-mapa`, afegiu-hi una escriptura sobre la constant: `((char *)constant)[0] = 'X';`. Compila? S'executa? Quin signal rebeu, i per què el rebeu en comptes de veure la constant canviada?

3. Feu que el `mem-mmap` demani la regió amb `PROT_READ` tot sol i intenteu escriure-hi. Compareu el que passa amb l'exercici anterior.

4. Compileu un programa qualsevol amb `gcc` i amb `gcc -static`, i compareu les mides dels dos executables i els dos mapes de `/proc/PID/maps`. On ha anat a parar la diferència?

5. Escriviu un programa que demani deu megabytes amb `mmap()` i no els toqui, i mireu-vos el camp `VmRSS` de `/proc/self/status`*. Ara escriviu un byte a cada pàgina i torneu-hi. De quin dels trucs d'aquest capítol acabeu de veure l'efecte?

6. Al `mem-compartida`, traieu el `waitpid()`. Continua sortint 42 amb `shared`? Sempre? Per què això és una pregunta del capítol 7 i no d'aquest?

*`VmSize` és la mida de l'espai lògic; `VmRSS`, la memòria física que de debò té assignada.