

Elements de Sistemes Operatius

Capítol 5

Programes versus processos

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 · 8 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.

5 Programes versus processos

Hi ha dues paraules que tothom fa servir com si fossin la mateixa i que no ho són. Un programa és una cosa; un procés, una altra. La diferència no és terminològica: és la diferència entre un full de partitura i un concert.

5.1 Un fitxer i un verb

Un programa és una seqüència d'instruccions i de dades desades en un fitxer ordinari amb permís d'execució. Aquest permís és l'única cosa que el fa executable: recordeu del capítol 3 que a Unix el nom del fitxer no vol dir res*.

Els fitxers executables surten de dos llocs. D'un compilador, com al capítol 4: el `gcc` en fa un fitxer amb el codi màquina i unes capçaleres que diuen al sistema on comença tot. O d'un editor de text, si el que escriviu és un guió d'*shell*: un fitxer de text amb comandes i un `chmod u+x` al damunt. El primer és ràpid i pot fer qualsevol cosa; el segon s'escriu en dos minuts i només pot fer el que ja saben fer les comandes.

En tots dos casos, però, el que teniu al disc és una cosa quieta. Un fitxer no fa res.

Un **procés** és la representació que el sistema operatiu fa d'un programa en execució. El programa és el que hi ha al disc; el procés és el que passa a la màquina mentre s'executa, amb la seva memòria, els seus registres, els seus fitxers oberts i la seva posició dins del codi.

*Un fitxer que es digui `programa.exe` sense permís d'execució no s'executa. Un que es digui `apunts.txt` amb permís d'execució, sí — i el sistema mirarà què hi ha a dins per saber com fer-ho.

D'aquí surt la primera conseqüència, i no és menor: del mateix programa en podeu tenir en marxa tants processos com vulgueu, i cadascun és independent dels altres. Tres finestres amb el mateix editor són un programa i tres processos. El que els distingeix no és el codi — que és idèntic — sinó tot allò que el sistema hi ha afegit.

5.2 Què cal, per posar un programa en marxa

Executar un programa vol dir, com a mínim:

- **Donar-li memòria** per al codi, per a les dades i per a la pila.
- **Preparar els registres** de la CPU, i en particular el comptador de programa, perquè comenci a executar-se per on toca.
- **Donar-li accés als dispositius**, que el procés no pot tocar directament perquè viuen en mode kernel: només hi arribarà a través de crides al sistema.
- **Recordar tot això**, perquè d'aquí a deu mil·lisegons el procés deixarà la CPU i caldrà poder-lo tornar a posar exactament on era.

Aquest últim punt és el que obliga a tenir una estructura de dades.

5.2.1 El PCB

Tot el que el sistema necessita saber d'un procés viu en una estructura anomenada *Process Control Block*, o PCB. Què hi ha exactament depèn del sistema i de la màquina, però sempre s'hi poden distingir tres grups:

- **L'espai d'adreces:** la descripció de les regions de memòria del procés — codi, dades, pila — i on són de debò. D'això en parla el capítol 8.
- **El context de programari:** el PID, la informació que necessita el planificador, quins dispositius té oberts, les estadístiques d'ús.

- **El context de maquinari:** el contingut dels registres i el que calgui de la màquina — la taula de pàgines, el comptador de programa — perquè el procés pugui continuar exactament on era.

Compte

El PCB és del kernel, no vostre. No hi ha cap manera que un procés el llegeixi o l'escrigui directament; el que hi ha són crides al sistema que en consulten o en canvien trossos concrets, i totes comproven que tingueu dret a fer-ho. Aquesta frontera és, literalment, el sistema operatiu.

5.3 Concurrencia i paral·lelisme

En una màquina normal hi ha desenes de processos alhora, i no és per capritx. Mentre un procés espera que el disc li porti dades, la CPU no té res a fer; si hi ha un altre procés preparat, se li dona a ell. Amb prou processos, la màquina no s'atura mai, i cada usuari té la impressió que la té per a ell tot sol.

Ara bé, «alhora» vol dir dues coses diferents, i convé no confondre-les.

Concurrencia: una CPU, repartida



Paral·lelisme: tres CPU, alhora

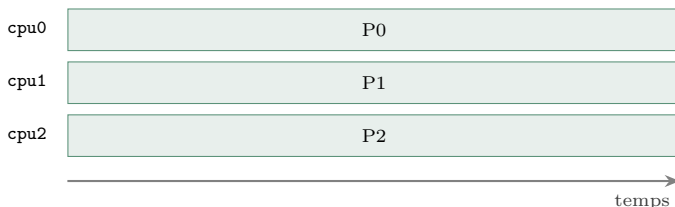


Figura 5.1. Els mateixos tres processos, vistos amb una CPU i amb tres.
A dalt s'alternen; a baix avancen de debò al mateix temps.

Diversos processos són **concurrents** si tenen la capacitat d'executar-se en paral·lel, tant si la màquina els ho permet com si no. Hi ha **paral·lelisme** quan de debò s'executen al mateix temps, cosa que demana més d'un core.

La concurrència és una propietat del programa: la teniu o no la teniu segons com l'hàgiu escrit. El paral·lelisme és una propietat de l'execució, i depèn de la màquina, del sistema operatiu i de què més s'hi estigui executant en aquell moment. El mateix programa concurrent, en una màquina d'un sol core, no en tindrà gens.

El contrari de concurrent és **seqüencial**: els processos s'executen l'un darrere l'altre perquè el programador ho ha forçat, sincronitzant-los. Ho veurem d'aquí a poques pàgines, i la manera de fer-ho serà posar un `waitpid()` al lloc adequat.

5.4 Els estats d'un procés

Si el sistema ha de decidir a qui dóna la CPU, li cal saber qui la vol. I no tots la volen: un procés que espera que arribi una tecla no serveix de res al planificador. Per això el kernel té els processos classificats segons què estan fent, i això és el que s'anomena l'**estat** del procés.

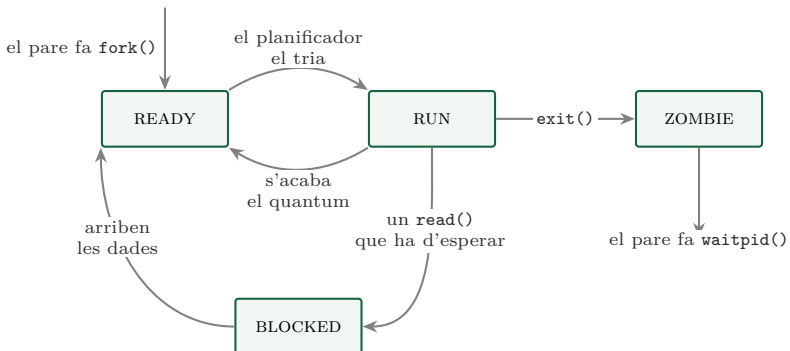


Figura 5.2. Els estats d'un procés a Linux, simplificats. Cada fletxa duu l'esdeveniment que la provoca.

- RUN: el procés té una CPU assignada i s'hi està executant.
- READY: està a punt per executar-se i només espera que li toqui.
- BLOCKED: no pot avançar, perquè espera dades d'un dispositiu o l'arribada d'un esdeveniment. Aquí no consumeix CPU, i és important que no en consumeixi.
- ZOMBIE: ha acabat, però encara no ha desaparegut de les estructures del kernel. De per què fa falta aquest estat en parlarem quan arribem a l'`exit()`.

Aquest graf no és universal: és una conseqüència de com és el sistema. Un sistema que no fos multiprogramat no necessitaria distingir RUN de READY; un que no tingués entrada/sortida bloquejant no necessitaria BLOCKED. Quan veieu un graf d'estats, el que esteu veient és la llista de decisions de disseny d'aquell sistema.

Qüestió 1

El pas de RUN a READY és l'única fletxa del dibuix que el procés no demana. Qui la provoca, i com pot ser que el kernel executi res si la CPU la té el procés?

Resposta

La provoca el planificador, i s'hi arriba per la interrupció de rellotge. Cada pocs mil·lisegons el maquinari interromp el que s'estigui executant i passa el control al kernel, que aprofita per mirar si aquell procés ja ha gastat el seu torn. Sense aquesta interrupció, un bucle infinit en un programa d'usuari deixaria la màquina penjada per sempre, que és exactament el que passava als sistemes que no en tenien. Al capítol 6 hi tornem.

5.5 Identitat, entorn i context

Un procés no és només el programa que executa: és tot allò que distingeix una execució d'aquell programa de la del costat. A Linux, aquestes propietats s'agrupen en tres.

La **identitat** diu qui és el procés i, per tant, què pot fer. Hi ha el *process id* o PID, un número que el kernel assigna a cada procés nou i que l'identifica en tot el sistema: és el que escriureu quan vulgueu enviar-li alguna cosa. I hi ha les credencials — l'usuari i els grups a què pertany el procés —, que són les que decideixen a quins fitxers i a quins recursos podrà arribar. De les credencials en tornem a parlar a l'última secció del capítol.

L'**entorn** és el que li heu donat en néixer: els paràmetres de la línia de comandes, que li arriben com l'**argv** del capítol 4, i les variables d'entorn del capítol 3, que hereta de qui l'ha creat.

El **context** és tota la resta: en quin estat està, quins fitxers té oberts, quanta CPU porta consumida, què hi ha als seus registres. És el que canvia a cada instant, i és el que ocupa la major part del PCB.

```
alumne@labS0:~$ ps -o pid,ppid,user,stat,cmd
  PID   PPID  USER   STAT  CMD
  2841   2839 alumne   Ss    -bash
  3112   2841 alumne   R+    ps -o pid,ppid,user,stat,cmd
```

Mireu-vos aquesta sortida amb calma, perquè hi és tot. El **ps** té el PID 3112 i el seu pare és el 2841, que és el *shell* des d'on l'heu llançat. Tots dos són vostres. I la columna **STAT** diu que el *shell* està adormit (S) esperant que el **ps** acabi, mentre que el **ps** s'està executant (R). Tot el que ve a continuació és la manera de provocar això des d'un programa.

5.6 Quatre crides i prou

Unix resol la gestió de processos amb un joc de crides sorprenentment petit:

Vull...	Crida
crear un procés	<code>fork()</code>
canviar-ne l'executable	<code>execlp()</code>
acabar	<code>exit()</code>
esperar que acabi un fill	<code>waitpid()</code>
saber el meu PID	<code>getpid()</code>
saber el PID del meu pare	<code>getppid()</code>

D'aquestes, el `waitpid()` és **bloquejant**: pot fer que el procés deixi l'estat RUN i passi a un estat des del qual no es pot executar, fins que passi allò que espera. No és cap desgràcia — al contrari, és el que permet que la CPU faci una altra cosa mentre tant —, però convé saber quines crides ho són, perquè són les que us canvien el dibuix de la pàgina anterior.

5.6.1 `fork()`: un procés que en fa un altre

Crida al sistema `fork`

```
pid_t fork(void);
```

El `fork()` crea un procés nou que és un clon del que l'ha cridat*. S'estableix una relació que en direm **pare** i **fill**, i com que el fill també pot cridar `fork()`, el que es genera és un arbre.

El fill rep una còpia privada de la memòria del pare: el mateix codi, les mateixes dades amb els mateixos valors, la mateixa pila. La memòria física és nova[†], però el contingut és idèntic. I el comptador de programa també: el fill comença a executar-se exactament per on era el pare, és a dir, tot just sortint del `fork()`.

*Literalment, i el nom no és casual: a Linux qui fa la feina és el `clone()`, amb la bandera `SIGCHLD`. El kernel té una sola crida per fer fils d'execució nous —la veurem al capítol 12— i el `fork` n'és el cas de «copia-ho tot i que en surti un procés». Per això a l'`strace` hi llegireu `clone`.

[†]Almenys conceptualment. Fer-ne una còpia de debò seria un malbaratament, i al capítol 8 veurem amb quin truc s'ho estalvien tots els Unix d'avui.

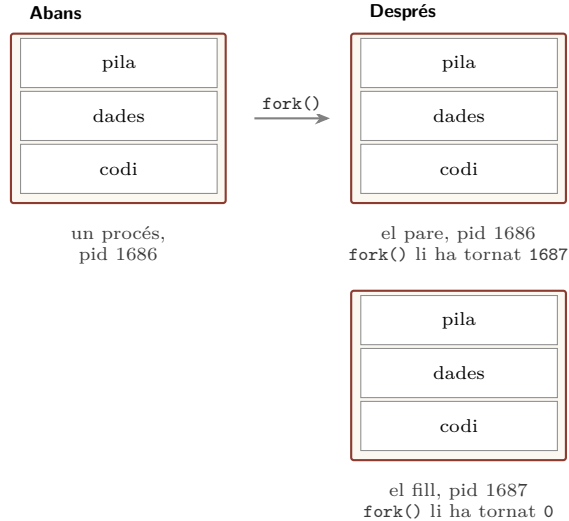


Figura 5.3. Un `fork()` vist des de la memòria. Tot és igual als dos costats menys una cosa: el valor que la crida els ha tornat.

I aquí ve l'única raresa del mecanisme, que és també la que el fa funcionar. Si els dos processos són idèntics i tots dos continuen per la mateixa línia, com poden fer coses diferents? Doncs perquè la crida no els torna el mateix:

- al **pare** li torna el PID del fill;
- al **fill** li torna **0**;
- i si ha fallat, torna **-1** i no hi ha cap fill.

`fork()` es crida una vegada i torna dues. Aquesta frase sembla una broma i és la descripció literal del que passa.

El programa més curt que ho ensenya és aquest:

Llistat 5.1. Es crida un cop i escriu dues línies

```

1  int main(void)
2  {
3      char buf[128];
4      pid_t jo, ret;
5
6      jo = getpid();                                ❶
7
8      ret = fork();                                  ❷
9      if (ret < 0)
10         error_i_exit("fork", 1);
11
12     sprintf(buf, "era %d; sóc %d; pare %d; fork() torna %d\n",
13             (int)jo, (int)getpid(), (int)getppid(), (int)ret);
14     write(1, buf, strlen(buf));                    ❸
15     return 0;
16 }

```

- ❶ El PID d'abans de la crida. Ens el guardem per poder-lo comparar.
- ❷ A partir d'aquí hi ha dos processos executant aquesta mateixa línia de codi.
- ❸ I per tant aquest `write()` s'executa dues vegades: un cop a cada procés.

```

alumne@labS0:~/S3$ ./proc-fork
era 1686; sóc 1686; pare 1607; fork() torna 1687
era 1686; sóc 1687; pare 1686; fork() torna 0

```

La segona línia és la del fill, i s'hi veu tot: el seu PID és nou, el seu pare és el procés original, i la variable `jo` — que va omplir el pare abans del `fork()` — encara duu el valor antic, perquè el fill n'ha heretat una còpia.

 Per saber-ne més

Una funció de C no pot tornar dos valors. El truc és que el `fork()` no en torna dos: en torna *un a cada procés*, i els dos processos no comparteixen registres.

Quan un programa entra al kernel, el kernel desa els registres del programa per poder-los-hi tornar en sortir, i els desa amb el procés —és el context de maquinari del PCB. El valor de retorn d'una crida tampoc no va enlloc més: el kernel l'escriu en aquella còpia, al registre del resultat de la taula del capítol 2 (el **rax** d'Intel, el **x0** d'ARM), i és en sortir que passa al registre de debò.

El **fork()** fa el fill copiant, entre altres coses, aquesta còpia dels registres del pare. I llavors hi fa l'única diferència: a la còpia del pare hi deixa el PID del fill, i a la del fill hi posa un zero. A Linux és una línia de la funció **copy_thread**, dins del codi de cada arquitectura.

El fill no s'executa encara. Quan el planificador el tria per primer cop, el kernel li restaura els registres com a qualsevol altre procés, el zero arriba al registre de veritat, i el programa del fill continua just després de la crida amb un **fork()** que ha tornat 0. Ningú no ha executat mai el **fork()** dins del fill: el fill neix a mitja sortida.

D'aquí surt l'esquelet de tot el que ve després:

```
ret = fork();
if (ret < 0)                /* no s'ha pogut: no hi ha fill */
    error_i_exit("fork", 1);
if (ret == 0) {
    /* això només ho executa el fill */
} else {
    /* i això només el pare */
}
/* i això, tots dos */
```

5.6.2 Què s'hereta i què no

El fill no és una còpia de tot.

Hereta — és a dir, en rep una còpia privada — l'espai d'adreces sencer, els dispositius que el pare tenia oberts, les credencials d'usuari i de grup, les variables d'entorn i la taula que diu què s'ha de fer amb cada signal (capítol 7).

No hereta el que per definició és seu i de ningú més: el PID i el PID del pare, els comptadors d'ús de CPU — que comencen a zero — i

els signals i les alarmes que el pare tinguéss pendents. Un esdeveniment adreçat al pare va adreçat al pare; el fill és un altre procés.

i Compte

Que el fill hereti els *dispositius oberts* i no la *memòria* és la distinció que costa més, i és la que fa possible el capítol 10. De la memòria en rep una còpia: si el pare canvia una variable, el fill no se n'assabenta. Dels canals oberts no en rep una còpia: en comparteixen la posició de lectura, de manera que si un llegeix, l'altre continua on l'ha deixat el primer.

5.6.3 `exit()` i `waitpid()`: acabar, i assabentar-se'n

Crida al sistema `exit` i `waitpid`

```
void exit(int estat);  
pid_t waitpid(pid_t pid, int *estat, int opcions);
```

Un procés pot acabar voluntàriament, cridant `exit()`, o involuntàriament, si el mata un signal*. En tots dos casos el sistema li allibera la memòria, li tanca els dispositius i recupera tot el que li havia donat.

Tot menys una cosa. Perquè el procés que acaba té alguna cosa a dir — l'estat de sortida que passa a l'`exit()`, i que per conveni val 0 si tot ha anat bé —, i aquesta informació s'ha de guardar en algun lloc fins que algú la reculli. El lloc és el PCB, i qui la recull és el pare, amb `waitpid()`.

Un procés que ha acabat però del qual el pare encara no ha recollit l'estat de sortida és un **zombie**. No té memòria, ni CPU, ni res:

*A l'`strace` tampoc no hi trobareu aquests dos noms. L'`exit` encara fa feina abans de plegar —executa el que hàgiu deixat dit amb l'`atexit` i buida els *buffers*— i acaba en un `exit_group()`, que a Linux és qui mata el procés sencer; el `waitpid`, en un `wait4()`.

només queda el PCB, esperant.

Els zombies fan gràcia però ocupen entrades d'una taula del kernel que és finita, i per això convé fer sempre el `waitpid()` dels fills que creueu. Si un procés mor deixant fills vius o zombies, el sistema els assigna un pare nou — l'`init`, el procés número 1 —, que es dedica, entre altres coses, a recollir els morts que ningú no ha reclamat.

Els dos usos habituals del `waitpid()` són aquests:

```
waitpid(-1, NULL, 0);    /* qualsevol fill      */
waitpid(fill, NULL, 0);  /* aquell fill i cap més */
```

I si en comptes de `NULL` li passeu l'adreça d'un enter, hi deixarà l'estat de sortida. Compte, però, que aquell enter no és el número que el fill ha passat a l'`exit()`: hi ha empaquetades dues coses diferents — com ha acabat i amb quin valor —, i s'han de desempaquetar amb macros:

```
if (WIFEXITED(estat))    /* ha acabat ell sol? */
    codi = WEXITSTATUS(estat); /* amb quin valor */
else
    quin = WTERMSIG(estat); /* quin signal l'ha mort */
```

5.6.4 `execlp()`: canviar de programa sense canviar de procés

Amb el `fork()` sol no aneu enlloc: tot el que podeu fer és executar més vegades el codi que ja teníeu. Per executar un altre programa cal la crida que li falta:

Crida al sistema `execlp`

```
int execlp(const char *fitxer, const char *arg0, ...);
```

L'última cosa que se li passa ha de ser un `(char *)NULL`, perquè és així com sap on s'acaba la llista.

El `execlp()` substitueix l'executable del procés per un altre*. Fixeu-vos bé en el que *no* diu: no crea cap procés. El procés és el mateix, amb el mateix PID i el mateix pare; el que canvia és el programa que hi ha a dins.

Canvia, doncs, tot l'espai d'adreces — codi, dades i pila del programa nou — i el comptador de programa, que torna a la primera instrucció. No canvia res del que és la identitat del procés: el PID, les credencials, els comptadors d'ús, els signals pendents. I la taula de signals es posa per defecte, cosa que té tot el sentit: el codi que la va programar ja no hi és.

Compte

Si el `execlp()` funciona, **no torna**. No hi ha res a què tornar: la instrucció següent era del programa vell, que ja no existeix. Per tant, l'única línia que pot venir després d'un `execlp()` és el tractament de l'error. Si s'executa, és que ha fallat.

Per saber-ne més

Les lletres del final no són decoració. L'`exec()` és una família de sis funcions que fan totes la mateixa cosa i només es diferencien en com els doneu la feina:

*De crides d'`exec`, POSIX en defineix sis; el kernel de Linux, una: l'`execve()`. Les altres cinc són llibreria i hi acaben totes. Quines són i en què es diferencien, unes quantes pàgines més avall.

lletra	què vol dir
l	els arguments d'un en un, acabats amb un <code>NULL</code> (<i>list</i>): és el cas del codi d'aquest capítol
v	els arguments en un vector (<i>vector</i>): és el que us cal quan no sabeu quants n'hi haurà fins que no us ho digui l'usuari
p	busca l'executable pel <code>PATH</code> en comptes d'exigir-vos el camí sencer (<i>path</i>)
e	li passeu vosaltres l'entorn en comptes d'heretar-lo (<i>environment</i>)

Combinant-les surten l'`execl()`, l'`execv()`, l'`execlp()`, l'`execvp()`, l'`execle()` i l'`execve()`. I totes sis acaben en la mateixa: la darrera, l'`execve()`, és l'única que el kernel arriba a veure, i les altres cinc li col·loquen els arguments com toca abans de cridar-la. Ho podeu comprovar vosaltres amb l'`strace`, i l'apèndix A ho fa.

De fills també n'hi ha més d'una manera de fer. El `vfork()` és un `fork()` que no copia res i que deixa el pare aturat fins que el fill faci l'`exec()`: venia de l'època en què copiar l'espai d'adreces era car de debò, i avui, amb la còpia mandrosa del capítol 8, ja no compensa el que costa de fer servir bé. I el `clone()` és la generalització de tots dos, que és el tema del capítol 12.

5.6.5 I amb això ja tenim un *shell*

Al capítol 3 vam definir el *shell* com un bucle que llegeix línies i executa programes, i vam prometre que hi cabria en trenta línies de C. Ja teniu les peces per veure com ho fa.

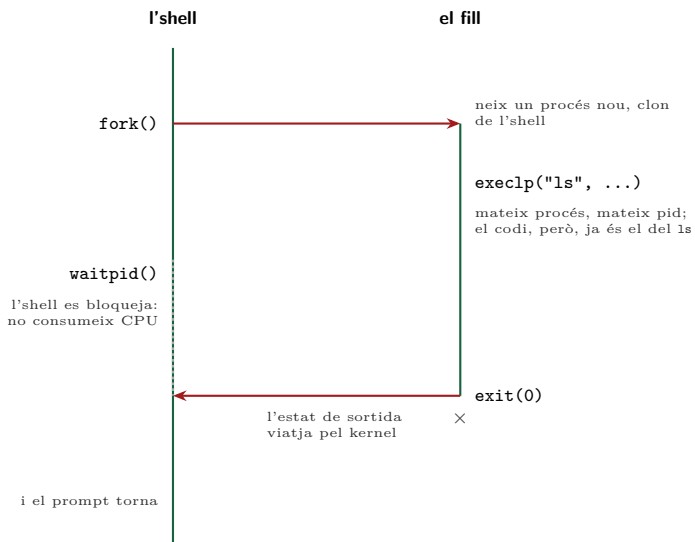


Figura 5.4. Què passa de debò quan escrivim `ls -l`: el *shell* es clona, el clon es converteix en el *ls*, i l'original espera.

Llistat 5.2. El *shell*, reduït a l'os

```

1  int main(void)
2  {
3      pid_t ret;
4      char buf[80];
5
6      ret = fork();
7      if (ret < 0)
8          error_i_exit("fork", 1);
9
10     if (ret == 0) {
11         execvp("ls", "ls", "-l", (char *)NULL);    ❶
12         error_i_exit("execvp", 1);                 ❷
13     }
14
15     waitpid(ret, NULL, 0);                          ❸
16     sprintf(buf, "i el prompt torna\n");

```

```

17     write(1, buf, strlen(buf));
18     return 0;
19 }

```

- ❶ Aquesta línia no torna mai, si va bé. A partir d'aquí el procés fill ja no executa aquest programa sinó el `ls`.
- ❷ I per això aquí no hi va cap `exit()`: si el flux hi arriba, és perquè el `execlp()` ha fallat, i el que cal és dir-ho.
- ❸ El pare espera aquell fill concret. Sense aquesta línia, el prompt tornaria abans que el `ls` hagués escrit res.

⚠️ Qüestió 2

Al llistat 5.2, què passaria si tragués el `if` i el `execlp()` el fessin tots dos processos?

💡 Resposta

Que el pare també es convertiria en un `ls`, i el `waitpid()` de la línia següent no s'executaria mai perquè aquella línia ja no existiria. Veuríeu dos llistats i cap prompt: el *shell* hauria desaparegut, substituït pel `ls`. Provar-ho des d'una finestra que no us importi és instructiu.

5.7 Dos patrons, i la línia que els separa

Amb aquestes quatre crides ja podeu escriure les dues maneres de repartir feina entre processos. Compareu-les: el codi és el mateix, i l'única diferència és on va el `waitpid()`.

Llistat 5.3. Seqüencial: d'un en un

```

1  int main(void)
2  {
3      pid_t ret;
4      int i;

```

```

5
6     for (i = 0; i < NFILLS; i++) {
7         if ((ret = fork()) < 0)
8             error_i_exit("fork", 1);
9         if (ret == 0) {                                ❶
10             tasca(i);
11             exit(0);                                  ❷
12         }
13         waitpid(-1, NULL, 0);                          ❸
14     }
15     return 0;
16 }

```

- ❶ El fill entra per aquí i el pare no.
- ❷ Sense aquest `exit()` el fill continuaria el bucle i faria fills seus. En parlem a la secció següent.
- ❸ Aquí, dins del bucle: el pare no crea el fill següent fins que l'anterior no ha acabat.

Llistat 5.4. Concurrent: tots alhora

```

1  int main(void)
2  {
3      pid_t ret;
4      int i;
5
6      for (i = 0; i < NFILLS; i++) {
7          if ((ret = fork()) < 0)
8              error_i_exit("fork", 1);
9          if (ret == 0) {
10             tasca(i);
11             exit(0);
12         }
13     }
14     while (waitpid(-1, NULL, 0) > 0)                    ❶
15         ;
16     return 0;
17 }

```

- ❶ La mateixa espera, però fora del bucle i repetida fins que no queda cap fill. El `waitpid()` torna `-1` quan ja no n'hi ha, i el bucle s'atura.

```

alumne@labS0:~/S3$ time ./proc-sequencial
    fill 0 (pid 1689) treballant
    fill 1 (pid 1690) treballant
    fill 2 (pid 1691) treballant
real    0m3,004s
alumne@labS0:~/S3$ time ./proc-concurrent
    fill 0 (pid 1693) treballant
    fill 1 (pid 1694) treballant
    fill 2 (pid 1695) treballant
real    0m1,003s

```

Cada fill dorm un segon. Tres segons contra un: aquesta és tota la diferència entre les dues versions, i és la raó per la qual cal entendre on va la línia.

5.8 L'arbre que surt d'un bucle

Torneu al llistat 5.3 i mireu-vos el globus ②. Aquell `exit()` sembla una formalitat i no ho és. Considereu aquest bucle, que és el mateix sense ell:

```

for (i = 0; i < 3; i++) {
    ret = fork();
    if (ret == 0)
        ferTasca();          /* ...i si aquí dins no hi ha exit()? */
}

```

Si `ferTasca` acaba amb un `exit()`, el fill mor allà mateix i el pare crea tres fills: un per iteració. Si no hi és, el fill torna del `ferTasca`, es troba dins del bucle, i continua les iteracions que li queden al pare — fent fills seus, que al seu torn faran el mateix.

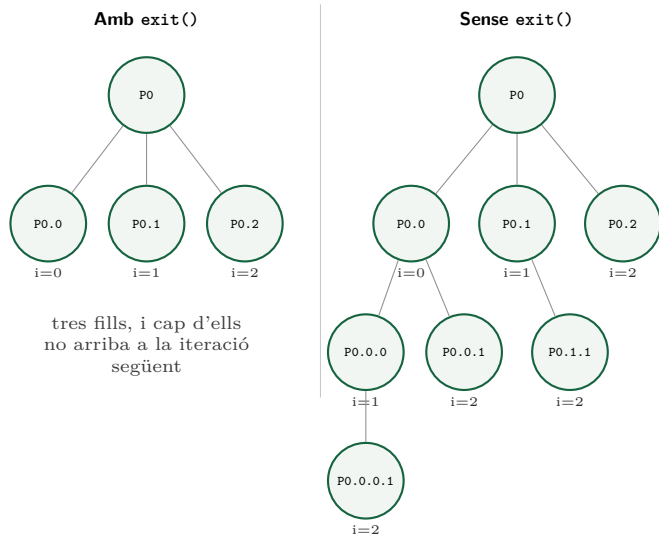


Figura 5.5. El mateix bucle de tres `fork()`, amb l'`exit()` del fill i sense. A l'esquerra hi ha quatre processos; a la dreta, vuit.

⚠️ Qüestió 3

A la dreta hi ha vuit processos, i el bucle en fa tres, de voltes. D'on surt el vuit? I quants n'hi hauria amb un bucle de deu?

💡 Resposta

De 2^3 . A cada iteració, *tots* els processos que hi ha en aquell moment executen el `fork()` — els fills inclosos, que hi han arribat per la volta següent a la que els ha creat — i per tant se'n dupliquen. Amb un bucle de deu en sortirien $2^{10} = 1024$, que és més del que l'administrador de la màquina us permetrà tenir.

5.9 Un *shell* d'una sola línia

Ajuntem-ho tot. Aquest programa rep una llista de comandes, les executa totes alhora i va informant de com acaba cadascuna.

Llistat 5.5. `fork + exec + waitpid`, tot junt

```

1  static void llanca(char *comanda)                                ❶
2  {
3      pid_t ret = fork();
4
5      if (ret < 0)
6          error_i_exit("fork", 1);
7      if (ret == 0) {
8          execlp(comanda, comanda, (char *)NULL);
9          error_i_exit("execlp", 1);
10     }
11 }
12
13 static void informa(pid_t pid, int estat)                          ❷
14 {
15     char buf[128];
16
17     if (WIFEXITED(estat))
18         sprintf(buf, "el procés %d ha acabat amb estat %d\n",
19                 (int)pid, WEXITSTATUS(estat));
20     else
21         sprintf(buf, "el procés %d l'ha mort el signal %d\n",
22                 (int)pid, WTERMSIG(estat));
23     write(1, buf, strlen(buf));
24 }
25
26 int main(int argc, char *argv[])
27 {
28     pid_t pid;
29     int i, estat;
30
31     if (argc < 2)
32         Usage(argv[0]);
33
34     for (i = 1; i < argc; i++)
35         llanca(argv[i]);
36

```

```

37     while ((pid = waitpid(-1, &estat, 0)) > 0)      ❸
38         informa(pid, estat);
39     return 0;
40 }

```

- ❶ Crear i mutar, en cinc línies. Fixeu-vos que el pare no espera: torna de seguida a llançar la comanda següent.
- ❷ Desempaquetar l'estat de sortida. Un programa que acaba bé i un que l'han mort no s'expliquen igual.
- ❸ L'espera, un cop llançades totes. L'ordre en què acaben no té res a veure amb l'ordre en què els heu escrit.

```

alumne@labS0:~/S3$ ./proc-llancador date whoami
alumne
Wed Sep  2 17:14:49 UTC 2026
el procés 1701 ha acabat amb estat 0
el procés 1700 ha acabat amb estat 0

```

Mireu l'ordre de les dues primeres línies: el **whoami** ha escrit abans que el **date**, tot i haver-se llançat després. No hi ha cap error. Són dos processos concurrents, i qui escrigui primer depèn del planificador, no de vosaltres. Si el vostre programa necessita que sigui d'una altra manera, ho ha de dir; i dir-ho és sincronitzar-los, que és de què va el capítol 7.

Per saber-ne més

Aquí sempre surt la mateixa pregunta, i és bona: **per què dues crides?** Si el que vol tothom és engegar un programa, per què no una de sola que ho faci tot?

N'hi ha una, i es diu **posix_spawn()**. Rep el programa, els arguments i l'entorn, i us torna el PID del fill ja engegat.

Però mireu què hi ha entre el **fork()** i l'**exec()** dels programes d'aquest capítol: hi cap *codi vostre*, executant-se ja dins del fill i encara amb el vostre programa a la memòria. Aquelles poques línies són el lloc on el *shell* tanca canals, en redirigeix, canvia de directori, posa el *nice* o deixa

anar privilegis —i el capítol 9 sencer viu en aquesta escletxa. Amb una sola crida no hi ha escletxa: tot el que vulgueu fer entremig ho heu de saber demanar per paràmetre, i el `posix_spawn()` en té una llista llarga justament per això, que mai no serà tan llarga com «qualsevol cosa que se us acudeixi».

O sigui que la separació no és un descuit dels anys setanta: és el que fa que el fill sigui programable. El preu és la còpia de l'espai d'adreces, i ja heu vist que avui és mandrosa i quasi no costa.

Totes dues són funcions de POSIX: l'estàndard fixa la interfície i no diu mai res de com s'implementa. Per això en aquest llibre totes dues són crides al sistema. A sota, però, cada kernel ha decidit per ell mateix quina es mereix una entrada a la taula de crides:

És una crida al sistema de debò?

	Linux	macOS
<code>fork()</code>	no : la libc fa un <code>clone()</code> amb <code>SIGCHLD</code>	sí : la número 2
<code>posix_spawn()</code>	no : la libc fa un <code>clone()</code> i un <code>execve()</code>	sí : la número 244

I aquesta diferència també explica una cosa. Al macOS, on un procés és una task de Mach i fer-ne una és car, el camí ràpid és el de la crida única; a Linux, on el `fork()` ja era barat, no calia. El mateix POSIX a sobre, dues decisions oposades a sota.

I encara hi ha un tercer camí, que avui fa anar mig món i que tampoc no ha calgut inventar. El mateix `clone()` que hi ha sota del `fork()` té una segona família de banderes que no diuen què es *comparteix* amb el pare, sinó què es *deixa de veure*: la taula de processos, l'arbre de fitxers, la xarxa. Un programa engegat així continua essent un `fork()` i un `exec()` ben corrents, i tothom en diu un contenidor. La secció 14.6 n'explica les dues meitats.

5.10 Què fa el kernel, mentrestant

Fins aquí, el mecanisme vist des de fora. Ara toca veure, encara que sigui a grans trets, què passa a l'altra banda de cada crida.

`fork()` busca un PCB lliure i el reserva; hi inicialitza el que és nou (el

PID, els comptadors a zero); aplica la política de memòria per donar-li un espai d'adreces amb una còpia del del pare; actualitza les estructures d'entrada/sortida perquè el fill tingui els mateixos canals oberts; i afegeix el procés a la cua dels que volen CPU.

execlp() substitueix l'espai d'adreces pel del programa nou, i reinicialitza els camps del PCB que hi estan lligats: la taula de signals, els registres, el comptador de programa. Ni toca el PCB com a entrada de la taula ni el treu de cap cua: el procés és el mateix i continua volent CPU.

exit() allibera tots els recursos — memòria, dispositius —, desa l'estat de sortida al PCB i treu el procés de la cua dels executables, de manera que ja no se li tornarà a donar CPU. El PCB, això sí, es queda. Després crida el planificador, perquè la CPU acaba de quedar lliure.

waitpid() busca el fill que li han demanat. Si ja era zombie, en recull l'estat de sortida, allibera el PCB i torna de seguida. Si encara viu, bloqueja el pare — de RUN a BLOCKED, la fletxa de la figura 5.2 — i crida el planificador.

Hi surten tres coses que aquest capítol ha anat deixant per a més endavant: les cues, el planificador i la política de memòria. Les dues primeres són el capítol 6; la tercera, el capítol 8.

5.11 Qui pot fer què

Queda una peça de la identitat del procés que hem deixat penjada, i és la que decideix si totes aquestes crides us deixaran fer res.

Cada procés porta un usuari i uns grups, heretats de qui l'ha creat, i el kernel els fa servir per contestar cada vegada la mateixa pregunta: aquest procés pot tocar aquest objecte? Per als fitxers, la resposta surt dels permisos que vam llegir al capítol 3 i que el capítol 11 explica a fons. Per als processos, val la mateixa regla: només podeu enviar signals o mirar per dins els processos del vostre usuari.

L'excepció és el **root**, que hi arriba a tot i que pot executar les operacions privilegiades del sistema. D'aquí ve el problema pràctic: hi

ha coses que un usuari normal ha de poder fer i que impliquen tocar fitxers que no són seus. Canviar-se la contrasenya, per exemple, vol dir escriure a `/etc/shadow`, que és del `root` i que no us convé gens que sigui de ningú més.

Unix ho resol amb un bit al permís del fitxer executable, el *setuid*: quan el porta, el procés que en surt no s'executa amb les credencials de qui l'ha llançat sinó amb les del propietari del programa.

```
alumni@lab50:~$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root shadow 89184 May  5 12:38 /usr/bin/passwd
```

Aquella `s` on esperàveu la `x` és tot el mecanisme*. El `passwd` és del `root`, i per tant, mentre s'executa, el vostre procés és el `root` i pot escriure on calgui†.

Compte

Un programa *setuid* és, per definició, un forat controlat a la protecció del sistema. Si té un error, l'error s'executa amb els privilegis del propietari, no amb els vostres. Bona part dels atacs clàssics contra Unix són exactament això: trobar un programa *setuid* del `root` que es pugui fer fallar d'una manera convenient. Per això n'hi ha d'haver pocs, i han de ser curts.

Per saber-ne més

El bit *setuid* no és màgia: el que fa és donar al procés **dos números en comptes d'un**. L'*UID real* és qui sou —qui ha llançat el programa—

*El punt que hi ha just després dels permisos no hi té res a veure: diu que el fitxer porta una etiqueta de seguretat addicional (de SELinux), que en aquest llibre no farem servir.

†Al laboratori no us servirà per canviar-vos la contrasenya. El `/etc/passwd` i el `/etc/shadow` d'aquelles màquines només tenen els usuaris locals: els vostres comptes es validen en un altre servidor, i el disc es torna a instal·lar cada vegada que s'engega l'ordinador. El mecanisme és el mateix; el fitxer on escriuria, no.

i l'*UID efectiu* és qui el sistema fa veure que sou a l'hora de decidir si us deixa tocar un fitxer. Normalment són el mateix número; en un programa *setuid*, no. El `getuid()` us diu el primer i el `geteuid()` el segon:

Crida al sistema `getuid` i `geteuid`

```
uid_t getuid(void);      /* qui sou */
uid_t geteuid(void);     /* qui podeu ser */
```

I n'hi ha una tercera, el `setuid()`, que serveix per a la cosa més important que fa un programa privilegiat ben escrit: **deixar anar els privilegis quan ja no els necessita**. El codi/`proc-credencials.c` és un programa de vuit línies que els imprimeix tots dos, els deixa anar i els torna a imprimir. Per provar-lo cal ser administrador de la màquina, cosa que al laboratori no sou; aquesta sessió és d'una màquina on sí:

```
$ sudo chown root proc-credencials
$ sudo chmod u+s proc-credencials
$ ./proc-credencials
abans:   real=1000  efectiu=0
després: real=1000  efectiu=1000
```

L'efectiu de la primera línia és un zero: mentre dura, aquell procés és el *root*. El de la segona ja no, i **no hi ha camí de tornada**: quan l'efectiu val zero, el `setuid()` canvia els tres UID —real, efectiu i guardat— i tanca la porta. Això és la virtut, no la limitació: un *passwd* ben escrit obre el `/etc/shadow` de seguida i torna a ser vós abans de mirar-se res del que heu teclejat, de manera que l'error que tingui més avall ja no s'executa com a *root*.

La protecció, en tot cas, és un problema intern: es tracta de decidir qui pot tocar què dins d'una màquina que funciona. La seguretat és una altra cosa, i és més gran: hi entren el lloc físic on és la màquina, les persones que hi tenen accés, el sistema operatiu i, avui sobretot, la

xarxa. Aquest llibre només us donarà la primera.

5.12 Exercicis

1. Executeu el **proc-fork** unes quantes vegades seguides. Els PID canvien; canvia sempre la diferència entre el del pare i el del fill? I l'ordre en què surten les dues línies?
2. Quants processos crea aquest fragment? Dibuixeu-ne l'arbre.

```
fork();  
fork();  
fork();
```

3. Modifiqueu el **proc-fork** perquè el fill escrigui **getppid()** després d'un **sleep(2)** i el pare no faci res més que acabar immediatament. Què us contesta el **getppid** del fill si l'espereu prou? (Pista: mireu el **ps** mentre passa.)
4. Traieu el **waitpid()** del **proc-sequencial** i deixeu-hi l'**exit()**. Quin dels dos programes obteniu? I si en traieu l'**exit()** i hi deixeu el **waitpid()**?
5. Feu que un fill del **proc-llancador** acabi amb un estat que no sigui 0 — per exemple, demanant-li que executi una comanda inexistent — i mireu què n'informa. Després mateu-ne un des d'un altre terminal amb **kill** i compareu.
6. Escriviu un programa que creï N processos en cadena: cada fill crea un fill i espera que acabi, fins a profunditat N . Quantes línies de diferència hi ha respecte del **proc-sequencial**?