

Elements de Sistemes Operatius

Capítol 7

Signals

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.

7 Signals

Fins ara, quan un procés havia d'esperar alguna cosa, l'esperava: es bloquejava en un `waitpid()` o en un `read()` i el kernel el despertava quan allò arribava. Tot passava per un lloc que el procés havia triat.

Els signals són l'altra manera. No hi ha cap lloc on esperar-los, perquè poden arribar en qualsevol moment i en qualsevol punt del codi.

7.1 Una interrupció software

Ja sabeu què fa una interrupció hardware: el processador està executant una cosa, arriba un senyal elèctric d'un dispositiu, i el processador abandona el que feia, executa una rutina de servei i torna on era. Un signal és exactament això, un pis més amunt.

Un **signal** és una **interrupció software** que el kernel envia a un procés per informar-lo que ha passat alguna cosa. El procés interromp el que estigués fent, executa la rutina que aquell signal tingui associada i — si hi sobreviu — continua on era.

La paraula que hi importa és *asíncron*. Un esdeveniment síncron passa cada vegada que s'executa una instrucció determinada; un signal pot caure entre dues instruccions qualssevol, i el programa no en té cap manera de saber quines. Aquesta és la font de gairebé tota la dificultat del capítol, i també de tot el que en podreu fer.

7.2 D'on venen

Els signals són a Unix des de les primeres versions, però no s'hi van posar per fer el que ara en fem. Servien per a situacions excepcionals — que un usuari intentés matar el procés d'un altre, que el maquinari fallés —, i en cap moment es van dissenyar com un mecanisme de comunicació entre processos [2].

Es nota. Els primers signals eren fàcils de perdre i difícils de gestionar. Tant BSD com el System V Release 3 d'AT&T van canviar el model per aconseguir signals *reliable*, però ho van fer de dues maneres incompatibles, i durant uns quants anys programar signals volia dir programar-los dos cops. Qui va posar-hi ordre va ser POSIX.1*, i és el model de POSIX el que explicarem aquí.

Compte

Encara trobareu codi escrit amb els esquemes antics, i encara funciona. La crida `signal()`, sense el prefix `sig`, n'és la peça central, i la trobareu a molts llibres i a molt codi antic. En aquest llibre no la farem servir mai: fa coses diferents segons el sistema i té una carrera pròpia que ningú no vol. El que sí que farem servir és `sigaction()`, que és la seva substituta i la que POSIX fixa. Barrejar els dos models, en canvi, no és mai bona idea.

7.3 Enviar-ne un

Un signal pot arribar-vos de tres llocs.

D'un altre procés, que executa la crida al sistema `kill()`. El nom és desafortunat: la crida no mata res, envia el signal que li digueu, i que això mati el destinatari o no depèn de quin signal sigui i de què hi tingui programat.

*També conegut com IEEE Std 1003.1-1988.

Crida al sistema `kill`

```
int kill(pid_t pid, int signum);
```

Per enviar-ne un heu de saber el PID del destinatari, i heu de tenir-hi dret: només podeu enviar signals a processos del vostre usuari, per la regla del capítol 5.

Llistat 7.1. Enviar un signal són dues línies

```

1  int main(int argc, char *argv[])
2  {
3      pid_t desti;
4
5      if (argc != 2)
6          Usage(argv[0]);
7
8      desti = (pid_t)atoi(argv[1]);
9      if (kill(desti, SIGUSR1) < 0)
10         error_i_exit("kill", 1);
11     return 0;
12 }
```

- ❶ L'única crida al sistema del programa. Torna `-1` si el procés no existeix o si no és vostre.

Del terminal, que és el mateix per un altre camí. Quan premeu **Ctrl-C**, el terminal envia un `SIGINT` a tots els processos que hi tingueu en primer pla; **Ctrl-Z** n'envia un `SIGTSTP`, que els atura. I des del *shell* teniu la comanda `kill`, que fa exactament el mateix que el llistat 7.1:

```

$ kill -USR1 1234      # envia un SIGUSR1 al procés 1234
$ kill -9 1234         # el número també val: 9 és SIGKILL
$ kill -l              # quins signals hi ha en aquesta màquina
$ man 7 signal         # i què vol dir cadascun
```

I del kernel mateix, que n'envia sense que ningú no els demani: quan un fill vostre acaba, quan se us esgota un temporitzador, o quan el vostre programa fa una cosa que el maquinari no li permet — tocar una adreça que no és seva, dividir per zero.

7.3.1 Quins signals hi ha

Cada esdeveniment té un signal assignat, i la llista la fixa el kernel: no en podeu inventar de nous. Són números, però tenen noms i s'escriuen sempre pel nom. Quants n'hi ha depèn del sistema*; aquests són els que sortiran en aquest capítol:

Nom	Per defecte	Quan arriba
SIGCHLD	s'ignora	un fill ha acabat o s'ha aturat
SIGCONT	continuar	continua, si estava aturat
SIGSTOP	aturar	atura el procés
SIGINT	acabar	interrupció del teclat (Ctrl-C)
SIGALRM	acabar	s'ha esgotat l' alarm()
SIGKILL	acabar	matar el procés
SIGSEGV	acabar i <i>core</i>	referència a memòria no vàlida
SIGUSR1	acabar	el que vulgueu
SIGUSR2	acabar	el que vulgueu

Els dos últims són el regal de Unix al programador: no signifiquen res, no els envia mai ningú si no els envieu vosaltres, i estan justament perquè els feu servir per al que us convingui. En aquest capítol els farem servir per sincronitzar processos.

Compte

Fixeu-vos en la primera fila. El pare rep un SIGCHLD cada cop que se li mor un fill — això que a tot arreu es diu «el pare és avisat» —, però l'acció per defecte és **ignorar-lo**. Si no feu res, no en notareu res.

*POSIX.1-2018 n'exigeix 29. Un `kill -1` a Linux en dóna 64; a macOS, 31.

7.4 Els tres tractaments

Quan un signal arriba a un procés, el kernel mira què hi té associat, i només hi ha tres possibilitats:

- **SIG_DFL**: l'acció per defecte, que és la de la taula de dalt. Hhabitualment acabar el procés, però n'hi ha que s'ignoren, un que l'atura i un que el continua.
- **SIG_IGN**: ignorar-lo. El signal arriba i no passa res, com si no s'hagués enviat.
- **Una funció vostra**, que en direm la **RAS** — rutina d'atenció al signal, el *signal handler* de la documentació. El kernel la crida quan el signal arriba, i en tornar-ne el procés continua on era.

Quan un programa comença, cada signal té **SIG_DFL** o **SIG_IGN**. Associar-hi una funció és el que en direm **capturar** el signal, i és la feina de la secció següent.

Compte

Dos signals no es poden ni capturar ni ignorar ni bloquejar: **SIGKILL** i **SIGSTOP**. I no és cap descuit. Si es poguessin capturar, un procés podria escriure una RAS buida per a tots dos i quedar-se la màquina per sempre: ni l'administrador el podria aturar. El `kill -9` és l'últim recurs del sistema perquè el kernel garanteix que ningú no s'hi pot resistir.

7.5 Capturar-lo: `sigaction()`

Crida al sistema `sigaction`

```
int sigaction(int signum, const struct sigaction *nou,
              struct sigaction *antic);
```

La crida consulta o canvia el tractament d'un signal, el `signum`. Si **nou** no és `NULL`, hi posa el tractament que li digueu; si **antic** no és `NULL`, hi deixa el que hi havia abans, cosa que és útil si el voleu restaurar més tard. Un cop feta, el tractament es queda com l'heu deixat fins que torneu a cridar `sigaction()`: això, que sembla obvi, és una de les coses que les implementacions antigues no garantien.

L'estructura no és igual a tot arreu*, però els tres camps que ens interessin hi són sempre:

```
struct sigaction {
    void      (*sa_handler)(int); /* què s'ha de fer      */
    sigset_t   sa_mask;          /* què es bloqueja mentre */
    int        sa_flags;          /* variants             */
};
```

sa_handler pot valer `SIG_DFL`, `SIG_IGN` o l'adreça d'una funció vostra amb la capçalera `void nom(int)`[†]. L'enter que rep la funció és el signal que ha arribat, de manera que podeu associar la mateixa funció a diversos signals i distingir-los a dins.

sa_mask és un conjunt de signals que es bloquejaren *mentre* s'executi la RAS, i que es desbloquejaren en sortir-ne. Serveix per protegir la RAS d'ella mateixa i de les seves veïnes.

sa_flags canvia detalls del comportament. Amb `SA_RESETHAND`, després de tractar el signal el tractament torna a ser el de per defecte; amb `SA_RESTART`, una crida al sistema bloquejada que hagi estat interrompuda pel signal es reprèn en comptes de tornar error. Amb 0 no hi ha cap variant, i és el que farem servir. Vol dir, però, una cosa que cal tenir present: sense `SA_RESTART`, una crida bloquejada que el signal interrompi **no es reprèn**, sinó que torna `-1` amb l'`errno` a `EINTR`. La funció `signal` de la

*En alguns sistemes els dos tractaments possibles són una `union`, i en d'altres dos camps separats. En tot cas només se n'omple un.

[†]Hi ha una segona forma, `sa_sigaction`, amb tres paràmetres, que dona a la RAS molta més informació sobre qui li ha enviat el signal i per què. En aquest llibre no la farem servir.

llibreria, que trobareu a molts exemples, fa just el contrari sense dir-ho; la secció 9.12 ho ensenyarà amb un programa.

Compte

`trat` és una variable local: si no n'ompliu els tres camps, els que deixeu duen les escombraries que hi hagi a la pila. El programa funcionarà — fins al dia que no. **Ompliu sempre els tres**, encara que dos siguin buits.

Amb això ja podeu reprogramar el Ctrl-C:

Llistat 7.2. Capturar un signal

```

1  /* La rutina d'atenció al signal. La crida el kernel, no ←
   nosaltres. */
2  static void ras(int quin)                                ❶
3  {
4      static const char adeu[] = "\nSIGINT rebut; plego jo\n";
5
6      (void)quin;
7
8      write(1, adeu, sizeof(adeu) - 1);                    ❷
9      _exit(0);                                             ❸
10 }
11
12 int main(void)
13 {
14     struct sigaction trat;
15     char buf[80];
16
17     sigemptyset(&trat.sa_mask);                            ❹
18     trat.sa_flags = 0;
19     trat.sa_handler = ras;
20
21     if (sigaction(SIGINT, &trat, NULL) < 0)                ❺
22         error_i_exit("sigaction", 1);
23
24     for (;;) {
25         sprintf(buf, "fent la meva feina...\n");
26         write(1, buf, strlen(buf));

```

```

27         sleep(1);
28     }
29     return 0;
30 }

```

- ❶ La RAS. Fixeu-vos que no la crida ningú del programa: la crida el kernel.
- ❷ `write()` i no `sprintf`: dins d'una RAS no s'hi pot cridar qualsevol cosa. En parlem a la penúltima secció.
- ❸ `_exit()` i no `exit()`, pel mateix motiu.
- ❹ Els tres camps, sempre.
- ❺ A partir d'aquí, un `SIGINT` ja no mata el procés.

```

alumne@labS0:~/S4$ ./sig-captura
fent la meva feina...
fent la meva feina...
^C
SIGINT rebut; plego jo

```

7.6 Bloquejar-lo: conjunts i `sigprocmask()`

Capturar un signal diu *què* s'ha de fer quan arribi. Bloquejar-lo diu *quan* voleu que arribi, i és una cosa ben diferent.

Cada procés té un conjunt de signals bloquejats — la *signal mask*. Mentre un signal hi és, si algú us l'envia no us arriba: el kernel se'l guarda com a **pendent** i us el donarà quan el desbloquegeu. Serveix per protegir els trossos de codi on rebre un signal faria mal.

Per manipular conjunts hi ha cinc funcions:

```

int sigemptyset(sigset_t *conj);    /* conjunt buit          */
int sigfillset(sigset_t *conj);    /* conjunt amb tots       */
int sigaddset(sigset_t *conj, int signum);
int sigdelset(sigset_t *conj, int signum);
int sigismember(const sigset_t *conj, int signum);

```

Aquestes cinc **no són crides al sistema**: són funcions de la llibreria de C, i les trobareu a la secció 3 del manual, no a la 2. No surten del vostre procés; només toquen una variable vostra. La que sí que és una crida al sistema és la que ve ara, que és l'única que canvia res dins del kernel.

Crida al sistema `sigprocmask`

```
int sigprocmask(int com, const sigset_t *conj,
                sigset_t *antic);
```

El primer paràmetre diu què s'ha de fer amb el conjunt:

SIG_BLOCK afegeix els signals de `conj` als bloquejats.

SIG_UNBLOCK els en treu.

SIG_SETMASK substitueix la màscara sencera per `conj`.

Si `conj` és `NULL` no es canvia res, i amb `antic` podeu llegir la màscara que hi havia: és la manera de preguntar què teniu bloquejat en un moment donat.

Bloquejar un signal, doncs, són quatre línies que veureu sempre juntes:

```
sigset_t bloc;

sigemptyset(&bloc);
sigaddset(&bloc, SIGUSR1);
sigprocmask(SIG_BLOCK, &bloc, NULL);
```

7.6.1 La vida d'un signal

Amb el bloqueig sobre la taula, ja es poden dir les paraules que la documentació fa servir i que no s'han de confondre.

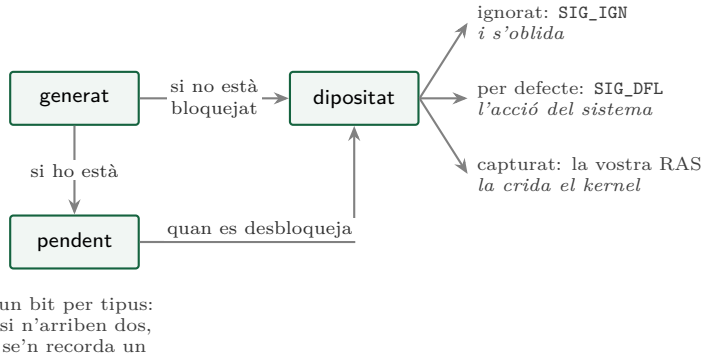


Figura 7.1. Un senyal es *genera*, potser espera un temps *pendent*, i es *diposita* quan el procés finalment el rep.

Un senyal es **genera** quan passa allò que el provoca, i es **diposita** quan el procés el rep de debò i s'executa el seu tractament. Entre les dues coses hi pot haver estona: si el senyal està bloquejat, queda **pendent** fins que el desbloquegeu. El temps que passa entre generar-se i dipositar-se és el seu temps de vida.

Compte

Els pendents es guarden en un bit per tipus de senyal, no en un comptador. Si mentre teniu bloquejat el `SIGUSR1` algú us n'envia cinc, en desbloquejar-lo n'executareu la RAS **una vegada**. No és un error d'implementació: els senyals no són un mecanisme per comptar coses. Si necessiteu comptar-les, els senyals us serviran per avisar i haureu de comptar-les vosaltres.

7.7 Esperar-lo bé

Ara ja podem plantejar el problema de debò: el procés A envia un senyal al procés B en algun moment, i B l'ha d'esperar.

La primera idea és l'espera activa: una variable global que la RAS posa a 1 i un bucle al programa principal que la va mirant.

```
while (!arribat)
    ;                               /* i mentrestant, cremar CPU */
```

Funciona i no és sempre una mala idea: si l'espera ha de ser molt curta, el cost de bloquejar el procés, planificar-ne un altre i tornar (el canvi de context del capítol 6) pot ser més gran que el de donar unes quantes voltes. Però per a esperes llargues és malbaratar la màquina: mentre doneu voltes, els altres processos — i el que espereu, que potser és un dels vostres — no avancen.

La segona idea és bloquejar-se, i és aquí on comença el problema.

7.7.1 La carrera del `pause()`

Hi ha una crida, `pause()`, que adorm el procés fins que arribi un signal qualsevol. Sembla que amb això n'hi hagi prou:

Llistat 7.3. Això no s'ha de fer

```
1  static volatile sig_atomic_t arribat = 0;           ❶
2
3  static void ras(int quin)
4  {
5      (void)quin;
6      arribat = 1;
7  }
8
9  int main(void)
10 {
11     struct sigaction trat;
12     char buf[80];
13
14     sigemptyset(&trat.sa_mask);
15     trat.sa_flags = 0;
16     trat.sa_handler = ras;
17     if (sigaction(SIGUSR1, &trat, NULL) < 0)
18         error_i_exit("sigaction", 1);
19
20     sprintf(buf, "esperant un SIGUSR1...\n");
```

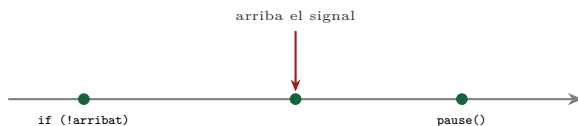
```

21     write(1, buf, strlen(buf));
22
23     if (!arribat)                                ❷
24         pause();                                  ❸
25
26     sprintf(buf, "arribat\n");
27     write(1, buf, strlen(buf));
28     return 0;
29 }

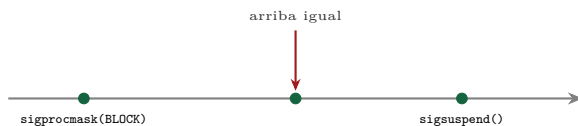
```

- ❶ `volatile sig_atomic_t` i no `int`: la variable la toquen dues coses alhora. Ho expliquem a la secció de la RAS.
- ❷ Mirem si ja ha arribat...
- ❸ ...i si no, ens adormim.

Entre el globus ❷ i el globus ❸ hi ha una escletxa. Són dues línies de C, i per tant unes quantes instruccions de màquina. Si el signal arriba justament allà, la RAS s'executa, posa `arribat` a 1, torna... i el programa continua on era, que és fent el `pause()`. El signal que esperàvem ja ha passat, i el `pause()` espera el següent, que no arribarà mai.

Amb pause()

La RAS s'executa i posa arribat a 1, però la condició ja s'havia mirat. El `pause()` espera el *següent* signal: el procés hi queda per sempre.

Amb sigsuspend()

Com que està bloquejat, no s'entrega: queda *pendent*. El `sigsuspend()` el desbloqueja i s'adorm en una sola operació, de manera que el rep tot seguit.

Figura 7.2. El mateix signal, arribant al mateix moment. A dalt es perd; a baix queda pendent i s'entrega tot seguit.

El programa del llistat 7.3 funcionarà les nou vegades de cada deu que el proveu, i és exactament per això que és perillós.

7.7.2 sigsuspend()

La solució ha de fer el desbloqueig i l'adormida en una sola operació, sense esclatxa entremig. Això és el que és `sigsuspend()`:

Crida al sistema `sigsuspend`

```
int sigsuspend(const sigset_t *conj);
```

La crida canvia la màscara del procés per `conj` i adorm el procés, tot alhora, de manera indivisible. El procés es desperta quan arriba un signal que `conj` no bloqueja; en tornar de la RAS, la màscara torna a ser la que era.

L'esquema és sempre el mateix: **bloquejar el signal abans de preparar res**, i passar a `sigsuspend()` un conjunt que el deixi passar. Així, si el signal arriba abans que estigueu a punt, no es perd: queda pendent, i el `sigsuspend()` el rep immediatament.

Llistat 7.4. Això sí

```

1  int main(void)
2  {
3      struct sigaction trat;
4      sigset_t bloc, espera;
5      char buf[80];
6
7      sigemptyset(&bloc);
8      sigaddset(&bloc, SIGUSR1);
9      if (sigprocmask(SIG_BLOCK, &bloc, NULL) < 0)
10         error_i_exit("sigprocmask", 1);
11
12     sigemptyset(&trat.sa_mask);
13     trat.sa_flags = 0;
14     trat.sa_handler = ras;
15     if (sigaction(SIGUSR1, &trat, NULL) < 0)
16         error_i_exit("sigaction", 1);
17
18     sigfillset(&espera);
19     sigdelset(&espera, SIGUSR1);
20
21     sprintf(buf, "esperant un SIGUSR1...\n");
22     write(1, buf, strlen(buf));
23     while (!arribat)
24         sigsuspend(&espera);
25
26     if (sigprocmask(SIG_UNBLOCK, &bloc, NULL) < 0)
27         error_i_exit("sigprocmask", 1);
28     sprintf(buf, "arribat\n");
29     write(1, buf, strlen(buf));
30     return 0;
31 }
```

❶ Bloquejar primer. A partir d'aquí, cap SIGUSR1 no es perd.

❷ El conjunt de l'espera: tot bloquejat menys el que esperem. Així sabem

exactament què ens pot despertar.

- ❸ El bucle no és decoratiu: podríem despertar-nos per un altre signal, i llavors hem de tornar a dormir.
- ❹ No es comprova l'error perquè `sigsuspend()` torna *sempre* `-1`, amb `errno` a `EINTR`. És l'única crida del llibre que ho fa.
- ❺ I es deixa la màscara com estava.

```
alumne@labS0:~/S4$ ./sig-espera &
[1] 2402
esperant un SIGUSR1...
alumne@labS0:~/S4$ ./sig-envia 2402
arribat
[1]+  Done                  ./sig-espera
```

⚠️ Qüestió 1

Al llistat 7.4, el conjunt del globus ❷ es fa amb `sigfillset` i un `sigdelset`. Hi ha qui, en comptes d'això, guarda la màscara antiga amb el tercer paràmetre del `sigprocmask()` i la passa al `sigsuspend()`. Fan el mateix?

💡 Resposta

Gairebé, i tots dos són correctes, però no diuen el mateix. Passar la màscara antiga vol dir «torna a deixar entrar tot el que entrava abans»; `sigfillset` més `sigdelset` vol dir «no deixis entrar res excepte això». La segona és més restrictiva i deixa el programa més fàcil de raonar: hi ha una sola cosa que us pugui despertar. En aquest llibre farem sempre la segona.

7.8 El rellotge: `alarm()`

Crida al sistema `alarm`

```
unsigned alarm(unsigned segons);
```

Demana al kernel que us envii un `SIGALRM` d'aquí a tants segons. Torna els segons que quedaven de l'alarma anterior, i amb `alarm(0)` la cancel·leu.

Compte

Cada procés té **un** temporitzador i prou. Si programeu una alarma quan ja n'hi havia una, la primera desapareix; no se n'encuen.

Amb això i el `sigsuspend()` ja podeu fer una cosa periòdica sense cremar ni un cicle de CPU esperant. Aquest programa llança un `ps` cada dos segons:

Llistat 7.5. Fer una cosa cada dos segons

```
1  #define VOLTES 3
2
3  /* No ha de fer res: només ens interessa que ens desperti. */
4  static void ras(int quin) ❗
5  {
6      (void)quin;
7  }
8
9  static void llanca_ps(void)
10 {
11     pid_t ret = fork();
12
13     if (ret < 0)
14         error_i_exit("fork", 1);
15     if (ret == 0) {
16         execlp("ps", "ps", (char *)NULL);
17         error_i_exit("execlp", 1);
```

```

18     }
19     waitpid(ret, NULL, 0);
20 }
21
22 int main(void)
23 {
24     struct sigaction trat;
25     sigset_t bloc, espera;
26     int i;
27
28     sigemptyset(&bloc);
29     sigaddset(&bloc, SIGALRM);
30     sigprocmask(SIG_BLOCK, &bloc, NULL);
31
32     sigemptyset(&trat.sa_mask);
33     trat.sa_flags = 0;
34     trat.sa_handler = ras;
35     if (sigaction(SIGALRM, &trat, NULL) < 0)
36         error_i_exit("sigaction", 1);
37
38     sigfillset(&espera);
39     sigdelset(&espera, SIGALRM);
40
41     for (i = 0; i < VOLTES; i++) {
42         alarm(2);
43         sigsuspend(&espera);
44         llanca_ps();
45     }
46     return 0;
47 }

```

- ❶ La RAS és buida a posta. No ha de fer res: només ens interessa que el signal ens tregui del `sigsuspend()`.
- ❷ Es programa l'alarma...
- ❸ ...i s'espera. Aquí el procés està en estat BLOCKED: mireu-vos el graf del capítol 5.
- ❹ I la feina es fa després de despertar-se, no dins de la RAS.

Fixeu-vos en una conseqüència del bloqueig que us estalviarà maldecaps. Mentre el programa està al globus ❹ llançant el `ps` — i esperant que acabi — el `SIGALRM` continua bloquejat. Si l'alarma salta enmig d'aquella feina, no es perd: queda pendent, i el `sigsuspend()` de la volta següent la recull tot seguit.

7.9 Els fills que moren

Al capítol 5 el pare es bloquejava en un `waitpid()` fins que els fills anaven acabant, i vam dir que convenia fer-ho per no deixar zombies. Però un pare que ha de fer feina no es pot passar la vida esperant.

El `SIGCHLD` és la sortida. El kernel l'envia al pare cada cop que un fill acaba; si el captureu, podeu recollir-los des de la RAS i no bloquejar-vos mai.

Llistat 7.6. Recollir els fills sense esperar-los

```

1  #define NFILLS 3
2
3  static volatile sig_atomic_t vius = NFILLS;
4
5  static void ras(int quin)
6  {
7      (void)quin;
8      while (waitpid(-1, NULL, WNOHANG) > 0)      ❶
9          vius--;
10 }
11
12 int main(void)
13 {
14     struct sigaction trat;
15     sigset_t bloc, espera;
16     pid_t ret;
17     char buf[80];
18     int i;
19
20     sigemptyset(&bloc);
21     sigaddset(&bloc, SIGCHLD);
22     sigprocmask(SIG_BLOCK, &bloc, NULL);        ❷
23
24     sigemptyset(&trat.sa_mask);
25     trat.sa_flags = 0;
26     trat.sa_handler = ras;
27     if (sigaction(SIGCHLD, &trat, NULL) < 0)
28         error_i_exit("sigaction", 1);
29
30     for (i = 0; i < NFILLS; i++) {
31         if ((ret = fork()) < 0)

```

```

32         error_i_exit("fork", 1);
33     if (ret == 0) {
34         sleep(i + 1);
35         exit(0);
36     }
37 }
38
39 sigfillset(&espera);
40 sigdelset(&espera, SIGCHLD);
41 while (vius > 0)
42     sigsuspend(&espera);
43
44 sprintf(buf, "tots recollits\n");
45 write(1, buf, strlen(buf));
46 return 0;
47 }

```

- ❶ WNOHANG vol dir «si no n'hi ha cap per recollir, no m'esperis». Dins d'una RAS, bloquejar-se seria un pecat.
- ❷ Bloquejat des del principi, com sempre: els fills poden morir abans que estiguem a punt.
- ❸ Acaben en ordre, però el programa no en depèn gens.
- ❹ I aquí el pare no espera cap fill concret: espera que el comptador arribi a zero.

⚠️ Qüestió 2

Al globus ❶ hi ha un bucle, i no una sola crida a `waitpid()`. Per què? Si arriba un `SIGCHLD` per cada fill que mor, no n'hi hauria d'haver prou amb recollir-ne un a cada RAS?

💡 Resposta

No, perquè els pendents es guarden en un bit i no en un comptador. Si dos fills moren mentre el `SIGCHLD` està bloquejat — o mentre s'executa la RAS, que el bloqueja implícitament — només en queda constància d'un. La RAS s'executarà una sola vegada i hi hauria dos fills per recollir. El bucle amb `WNOHANG` els recull tots els que hi hagi i se'n va quan no en queda cap.

7.10 Què travessa un `fork()` i un `exec()`

Els signals són l'exemple més clar de la distinció del capítol 5 entre el que és del programa i el que és del procés.

Amb un `fork()`, el fill **hereta** la taula de tractaments — té una còpia del codi del pare, i per tant les RAS existeixen a la seva memòria — i hereta també la màscara de bloquejats. No hereta, en canvi, els signals pendents ni el temporitzador: aquells anaven adreçats al pare, i el fill és un procés nou amb un PID nou.

Amb un `exec()` passa gairebé el contrari. Els tractaments que havíeu programat es perden i tornen a ser els de per defecte, cosa que és inevitable: el codi de les RAS ja no hi és*. Però el procés no canvia, i per tant *sí* que conserva la màscara de bloquejats i els signals que tingués pendents.

Llistat 7.7. El comptador no s'hereta; la RAS, sí

```

1  int main(void)
2  {
3      struct sigaction trat;
4      sigset_t bloc, espera;
5      char buf[96];
6      const char *qui;
7      unsigned resta;
8      pid_t ret;
9
10     sigemptyset(&bloc);
11     sigaddset(&bloc, SIGALRM);
12     sigprocmask(SIG_BLOCK, &bloc, NULL);
13     sigemptyset(&trat.sa_mask);
14     trat.sa_flags = 0;
15     trat.sa_handler = ras;
16     if (sigaction(SIGALRM, &trat, NULL) < 0)
17         error_i_exit("sigaction", 1);
18     sigfillset(&espera);
19     sigdelset(&espera, SIGALRM);
20
21     alarm(5);
```

*Els que estaven a SIG_IGN continuen ignorant-se: per ignorar un signal no fa falta codi.

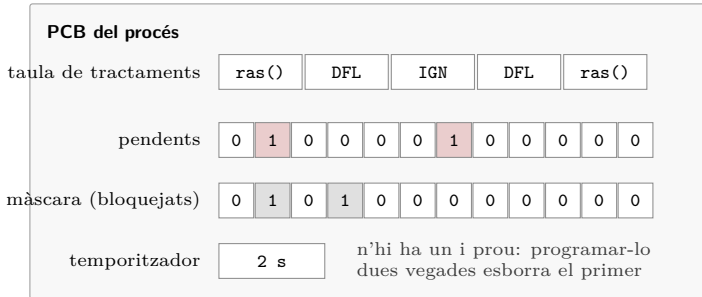
```
22     if ((ret = fork()) < 0)                                ❷
23         error_i_exit("fork", 1);
24     qui = (ret == 0) ? " el fill" : "el pare";
25
26     sleep(1);
27     resta = alarm(0);                                     ❸
28     sprintf(buf, "%s: em quedaven %u segons\n", qui, resta);
29     write(1, buf, strlen(buf));
30
31     alarm(1);                                              ❹
32     sigsuspend(&espera);
33     sprintf(buf, "%s: he executat la meva RAS\n", qui);
34     write(1, buf, strlen(buf));
35
36     if (ret > 0)
37         waitpid(ret, NULL, 0);
38     return 0;
39 }
```

- ❶ L'alarma la programa el pare, abans de clonar-se.
- ❷ I aquí ja són dos.
- ❸ `alarm(0)` cancel·la l'alarma i torna els segons que quedaven. És la manera de preguntar-ho sense esperar.
- ❹ Ara cadascú es programa la seva, i tots dos executen la RAS que el pare va instal·lar abans del `fork()`.

```
alumne@labS0:~/S4$ ./sig-herencia
el pare: em quedaven 4 segons
  el fill: em quedaven 0 segons
  el fill: he executat la meva RAS
el pare: he executat la meva RAS
```

7.11 Què en guarda el kernel

Tot el que hem anat dient viu al PCB, l'estructura del capítol 5: la gestió dels signals és per procés, i cada procés té la seva.



El signal 1 està pendent i bloquejat: no s'entregarà fins que s'esborri el seu bit de la màscara. El 6 està pendent i no bloquejat: s'entrega de seguida.

Figura 7.3. Els signals, tal com el kernel se'ls mira: una taula, dues tires de bits i un comptador.

Són quatre coses. Una **taula de tractaments** amb una entrada per signal, que diu si és SIG_DFL, SIG_IGN o l'adreça d'una RAS. Un **bitmap de pendents**, un bit per signal — d'aquí ve que no se'n puguin comptar. Una **màscara** de bloquejats, de la mateixa mida. I un **temporitzador**, un de sol, que és el de l'**alarm()**.

El kernel tracta els signals dins del context del procés que els rep, i això vol dir que el procés s'ha d'estar executant per rebre'n cap. Un signal enviat a un procés que està a la cua de READY no es diposita fins que el planificador li torni la CPU. És per això que quan mireu la sortida d'un programa amb signals no en podeu deduir l'ordre exacte de res.

7.12 Una RAS que no us traeixi

Una RAS s'executa entre dues instruccions qualssevol del vostre programa. Això vol dir que pot enxampar el programa principal a mig fer qualsevol cosa — a mig actualitzar una estructura, a mig escriure en un buffer de la llibreria —, i que si la RAS toca allò mateix, el que quedi no serà ni una cosa ni l'altra.

Els consells són pocs, i s'han de seguir tots [1]:

Curta. Com més curta, menys ocasions de fer mal. La feina de debò

es fa al programa principal, com al llistat 7.5.

Només funcions segures. Dins d'una RAS només hi podeu cridar funcions *async-signal-safe*: les que es poden interrompre i tornar a començar sense mal. `write()` ho és; `printf`, `sprintf` i `malloc`, no. La llista sencera és a `man 7 signal-safety`, i n'hi ha una discussió llarga a l'Stevens [3].

Deseu l'errno. La RAS pot fer una crida al sistema que canviï l'`errno` enmig d'una comprovació del programa principal. Deseu-lo al principi i restaureu-lo al final.

Protegiu les variables compartides. Si el programa principal i la RAS toquen la mateixa variable, teniu una regió crítica. Bloquegeu el signal mentre la toqueu.

Declareu les volatile. Una variable que la RAS canvia canvia d'esquena al compilador, que altrament és lliure de guardar-la en un registre i no tornar-la a llegir mai. El tipus per a això és `volatile sig_atomic_t`.

Compte

El `sprintf` i el `write()` que fem servir a tot el llibre no valen dins d'una RAS: el `write()` sí, el `sprintf` no. Per això les RAS d'aquest capítol o no escriuen res, o escriuen una cadena constant amb `write()` directament.

7.13 Terminologia

El vocabulari dels signals és petit, i és el mateix que fareu servir tota la vida: el `man`, el codi font del kernel i qualsevol llibre que consulteu us el diran en anglès. Per això aquí van els dos costats de cada paraula. No és una llista de traduccions: són les vuit paraules amb què es descriu tot el que li pot passar a un signal, i confondre'n dues és confondre el mecanisme.

Anglès	Català	Què vol dir
generate	generar	produir-se l'esdeveniment que provoca el signal
deliver	dipositar	arribar al procés i executar-se'n el tractament
lifetime	temps de vida	el que passa entre generar-se i dipositar-se
pending	pendent	generat i encara no dipositat, perquè està bloquejat
catch	capturar	haver-hi associat una RAS pròpia amb <code>sigaction()</code>
ignore	ignorar	rebre'l i no fer-ne res; és un tractament, no l'absència de tractament
block	bloquejar	retardar-ne el dipòsit; un signal bloquejat no s'ha perdut, espera
signal handler	RAS	la rutina d'atenció al signal: la funció que s'executa en dipositar-se

7.13.1 Capturat no vol dir dipositat

D'aquestes vuit paraules n'hi ha dues que es confonen sistemàticament, i cal aturar-s'hi, perquè la confusió no és de vocabulari: és de model.

Capturar és una propietat del *tipus* de signal, i es fa una sola vegada. **Dipositar** és un esdeveniment que li passa a *un* signal concret, i passa cada cop que n'arriba un.

Quan feu `sigaction()` sobre el `SIGUSR1`, heu capturat el `SIGUSR1` — tots els `SIGUSR1` que us arribin d'ara endavant, i no cap en particular. No n'ha arribat cap encara. El programa del llistat 7.2 captura el `SIGINT` a la tercera línia del `main` i es passa hores sense que se'n dipositi cap.

I a l'inrevés: un signal es pot dipositar sense estar capturat. Si no heu tocat res, el `SIGINT` que arriba amb el `Ctrl-C` es diposita igualment — s'executa l'acció per defecte i el procés mor. El signal ha

arribat, s'ha tractat, s'ha dipositat. Simplement no l'havíeu capturat.

i Compte

D'aquí surt l'error de laboratori més freqüent d'aquest capítol: «he capturat el signal però no em passa res». Capturar-lo no fa que arribi. Si no arriba, o no us l'envia ningú, o el teniu bloquejat, o el destinatari és un altre procés.

I encara dues confusions més, més fàcils de desfer:

- **Ignorar i bloquejar** no són el mateix. Un signal ignorat arriba i es llença — s'ha dipositat, i el tractament ha estat no fer res. Un de bloquejat *encara* no ha arribat, i arribarà.
- **Generar i dipositar** tampoc. Entre les dues coses hi pot haver una estona llarga, i és exactament l'estona que dura la figura 7.1.

7.14 Exercicis

1. Executeu el **sig-carrera** en segon pla i envieu-li el SIGUSR1 de seguida, sense esperar. Ara feu-ho posant un **sleep 1** pel mig. Quantes vegades de deu us queda penjat en cada cas?
2. Modifiqueu el **sig-espera** perquè esperi un SIGUSR1 o un SIGUSR2 i digui quin ha rebut. Amb una sola RAS n'hi ha prou: mireu-vos el paràmetre que rep.
3. Al **sig-captura**, canvieu SIGINT per SIGKILL i proveu-ho. Què us diu el **sigaction()**? Per què?
4. Escriviu una funció que digui quins signals teniu bloquejats en aquell moment. (Pista: **sigprocmask()** amb el segon paràmetre a NULL, i **sigismember** en un bucle.) Compte a fer-la servir des d'una RAS: què hi podeu cridar i què no?
5. Al **sig-fills**, traieu el bucle de la RAS i deixeu-hi un sol **waitpid()**. Feu que els tres fills acabin alhora — traieu-ne el

`sleep` — i mireu si el programa acaba. Amb el `ps` podreu veure què ha quedat.

6. Feu un programa que, cada segon i durant deu, escrigui quant de temps li queda, i que en acabar plegui. Sense cremar CPU i sense fer servir `sleep`.

Bibliografia

- [1] Randal E. Bryant i David O'Hallaron. *Computer systems : a programmer's perspective*. eng. 3rd ed. Essex: Pearson, 2016. ISBN: 9781292101767. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711.
- [2] Marshall Kirk McKusick i George V. Neville-Neil. *The design and implementation of the FreeBSD operating system*. eng. Boston: Addison-Wesley, 2005. ISBN: 0201702452. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/8e3cvp/alma991003102659706711.
- [3] W. Richard. Stevens i Stephen A. Rago. *Advanced programming in the UNIX environment*. eng. 3a ed. Addison-Wesley professional computing series. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 0321637739. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004011309706711.