Introducció a la programació de Unix

(Versió 1.32)

Miquel Nicolau i Vila

Índex

1 Introducció

En aquest document s'introduiran els conceptes bàsics de la programació amb crides a sistema de Unix. En cap moment, però, vol ser un catàleg complet de totes les crides a sistema i del seu ús, sinó que vol presentar les funcions més significatives de l'entorn Unix relacionades amb la gestió dels processos, l'entrada/sortida i la comunicació entre processos. L'objectiu final és oferir les eines bàsiques per entendre els trets més característics de Unix i poder així començar a desenvolupar aplicacions sobre aquest entorn.

El document s'estructura en quatre capítols. Aquest primer capítol és una introducció que presenta, de forma general, alguns conceptes que ajudaran a entendre els fonaments de la programació sobre Unix. Els altres tres capítols tracten tres àmbits fonamentals de tot sistema operatiu: la gestió de processos (capítol 2), l'entrada/sortida (capítol 3) i la comunicació i sincronització entre processos (capítol 4).

L'esquema dels capítols 2, 3 i 4 és molt similar. En cadascun d'ells s'introdueix, primer de tot, els conceptes relacionats amb la temàtica tractada, tot seguit es presenten les crides a sistema fonamentals i s'acaba amb un conjunt d'exemples que ajudaran a clarificat tot el que s'ha tractat.

La presentació de cada crida a sistema s'estructura en cinc apartats:

1. *Sintaxi*: es presenta la sintaxi en llenguatge C de cada crida a sistema i es descriuen els fitxers de definicions (*include*) necessaris.
2. *Descripció*: s'introdueix amb detall el funcionament de la crida a sistema i les accions que es duen a terme.
3. *Paràmetres*: s'expliquen un a un els paràmetres utilitzats i els possibles valors.
4. *Valor retornat*: es descriuen els possibles valors retornats i el seu significat.
5. *Errors*: es presenten els errors més significatius que pot produir cada crida a sistema.

1.1 El manual de Unix

Unix ofereix un manual d'usuari estructurat en 8 seccions on es descriuen totes les seves característiques. Aquest manual està disponible en el propi sistema operatiu (*man pages*) i s'hi pot accedir a través de la comanda **`man`** de l'intèrpret de comandes. Les seccions tracten els temes següents:

Secció 1: comandes d'usuari disponibles des de l'intèrpret (*shell*)
Secció 2: crides a sistema
Secció 3: funcions de biblioteca (*library*) del llenguatge C
Secció 4: dispositius
Secció 5: formats dels arxius
Secció 6: jocs
Secció 7: entorns, taules i macros
Secció 8: manteniment del sistema

La descripció de cada concepte del manual s'acompanya de l'identificador de la secció a la qual pertany. Per exemple, la comanda *ls* que permet llistar el contingut d'un directori apareixerà com *ls (1)* ja que és una comanda de l'intèrpret i, per tant, es

descriurà a la secció 1. Totes les crides a sistema estaran explicades a la secció 2 i les referències ho indicaran així: *fork (2)*.

El manual en línia de Unix és una eina molt important de suport en l'ús del sistema operatiu i, en concret, per al bon coneixement de les crides a sistema.

Algunes exemples de comandes:

- `$ man man` (obtindrem ajuda del propi manual)
- `$ man ls` (obtindrem ajuda sobre la comanda *ls*)
- `$ man 2 write` (obtindrem ajuda de la crida a sistema *write*, en ella s'ha especificat la secció 2 –amb el paràmetre 2– per distingir-la de la comanda de l'intèrpret *write (1)*.

1.2 El control d'errors

La majoria de crides a sistema de Unix retornen un valor enter en acabar la seva execució. Un valor 0 o positiu indica un acabament correcte de la crida. Un valor negatiu (-1) indica un error en la seva execució.

Cada crida a sistema pot produir errors diferents i variats. Per poder saber l'error que s'ha produït, cada procés disposa d'una variable global anomenada *errno* que descriu l'error produït després de cada crida a sistema. Per poder controlar correctament el comportament d'un procés, que utilitza crides a sistema, és del tot necessari fer un seguiment detallat de l'execució de cada crida. Per aquesta raó, és del tot aconsellable verificar el correcte acabament de les crides i, en cas contrari, detectar l'error que s'ha produït.

L'estructura general de programació de qualsevol crida a sistema seria la següent:

```
#include <errno.h>

int p[2];

...

if (pipe(p) < 0){

      /* Escriu l'error pel canal estàndard d'errors (canal 2) */

      write(2, "Error pipe\n", strlen("Error pipe\n"));
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}
```

En el codi anterior s'inclou el fitxer *errno.h*, on es descriu la variable global *errno* necessària per identificar l'error produït. El procés executa la crida a sistema *pipe* que retornarà el valor -1 en cas d'error. Si es produeix un error, el codi escriurà (*write*) el tipus d'error (variable *errno*) i acabarà l'execució del procés (crida a sistema *exit*).

2 La gestió de processos

En aquest capítol es descriuran les característiques fonamentals dels processos, els seus atributs, la creació i destrucció, juntament amb les crides bàsiques per a la seva gestió.

2.1 Els processos

La gestió de processos és l'eina fonamental que permet la creació i destrucció de nous processos dins del sistema operatiu. A Unix existeix una jerarquia de processos encapçalada per un procés inicial a partir del qual es genera la resta de processos del sistema. Aquest procés (el procés *init*) té l'identificador 1 i tindrà un paper molt important al llarg de la vida del sistema tal i com es veurà més endavant.

Els processos de Unix tenen un conjunt d'atributs, que cal conèixer per poder gestionar-los correctament, dels quals cal destacar els següents:

- Identificador del procés (*PID*)
- Identificador del procés pare (*parent process ID*)
- Identificador del grup de processos (*process group ID*)
- Identificador d'usuari (*UID*)
- Identificador del grup d'usuaris (*GID*)
- Estat del procés

2.1.1 Identificador del procés (*PID*)

L'identificador del procés és un enter positiu únic que s'associa a cada procés en el moment de la seva creació i que es manté fins a la seva desaparició. Aquest identificador permetrà gestionar el procés al llarg de la seva vida i fer-ne el seguiment de la seva execució.

2.1.2 Identificador del procés pare (*parent process ID*)

Els processos de Unix mantenen la relació jeràrquica del sistema mitjançant l'identificador del seu procés pare (*parent process ID*). Aquest identificador permet saber qui ha creat el procés.

2.1.3 Identificador del grup de processos (*process group ID*)

L'identificador de grup indica el grup de processos al qual pertany el procés. Un grup de processos és una agrupació de processos que facilita la gestió conjunta d'algunes funcions com l'enviament de senyals (crida a sistema *signal*). El procés líder del grup és el que defineix el valor de l'identificador del grup, que serà el mateix que el seu identificador de procés (*PID*). Els grups de processos existeixen mentre existeixi algun procés del grup. Un nou procés manté el grup del procés pare mentre no es canviï de grup mitjançant la crida a sistema *setpgid*.

2.1.4 Identificador d'usuari (*UID*)

Tot procés pertany a l'usuari que l'ha creat i tot usuari de Unix posseeix un identificador únic que el representa. L'identificador d'usuari s'assignarà al procés en el moment de la seva creació i li oferirà un conjunt de drets sobre els recursos del sistema.

Tot procés disposarà de dos identificadors d'usuari:

- l'identificador real d'usuari (*real user ID*)
- l'identificador efectiu d'usuari (*effective user ID*)

L'identificador real d'usuari (*real user ID*) no es modifica mai en tota la vida del procés i correspon a l'identificador de l'usuari que ha creat el procés.

L'identificador efectiu d'usuari (*effective user ID*) és el que utilitza el sistema per verificar els drets del procés sobre els diferents recursos del sistema. En el moment de la creació d'un procés, el seu identificador efectiu d'usuari coincideix amb l'identificador real d'usuari, però l'identificador efectiu d'usuari sí que es pot modificar de forma controlada al llarg de l'execució del procés. La modificació de l'identificador efectiu d'usuari ofereix una eina important per a l'accés restringit dels processos d'usuari a recursos protegits del sistema.

El canvi d'identificador efectiu d'usuari el pot provocar l'execució (crida a sistema *exec*) d'un fitxer executable que tingui actiu el bit anomenat *setUID* (*set-user-ID*). Si el bit *set-user-ID* està actiu, l'identificador efectiu d'usuari del procés que ha executat (*exec*) el fitxer, prendrà com a valor l'identificador del propietari del fitxer executable. L'identificador real d'usuari no és modificarà.

Un exemple clar de la utilització del *setUID* es pot trobar en la comanda *passwd*. Aquesta comanda permet que qualsevol usuari pugui modificar la seva clau d'accés al sistema. Aquesta clau està emmagatzemada en un fitxer (*/etc/passwd*) que és propietat de l'usuari *root* o *super-user* i que només aquest usuari pot modificar. El fitxer executable amb la comanda *passwd* pertany també a l'usuari *root* i té el bit *setUID* actiu. Per tant, qualsevol procés que executi aquesta comanda canviarà el seu identificador efectiu d'usuari que prendrà per valor l'identificador d'usuari *root* i, per tant, podrà accedir al fitxer de claus (*/etc/passwd*) i modificar-lo.

2.1.5 Identificador del grup d'usuaris (*GID*)

Tot procés pertany al grup d'usuaris de l'usuari que l'ha creat. L'identificador del grup d'usuaris s'assignarà al procés en el moment de la seva creació i li oferirà un conjunt de drets sobre els recursos del sistema.

Tot procés disposarà de dos identificadors del grup d'usuaris:

- l'identificador real del grup d'usuaris (*real group ID*)
- l'identificador efectiu del grup d'usuaris (*effective group ID*)

L'identificador real del grup d'usuaris (*real group ID*) no es modifica mai en tota la vida del procés i correspon a l'identificador del grup d'usuaris de l'usuari que ha creat el procés.

L'identificador efectiu del grup d'usuaris (*effective group ID*) és el que utilitza el sistema per verificar els drets del procés sobre els diferents recursos del sistema. En el moment de la creació d'un procés, el seu identificador efectiu del grup d'usuaris coincideix amb l'identificador real del grup d'usuaris, però l'identificador efectiu del grup d'usuaris sí que es pot modificar de forma controlada al llarg de l'execució del procés. La modificació de l'identificador efectiu del grup d'usuaris ofereix una eina important per a l'accés restringit dels processos d'usuari a recursos protegits del sistema.

El canvi d'identificador efectiu del grup d'usuaris el pot provocar l'execució (crida a sistema *exec*) d'un fitxer executable que tingui actiu el bit anomenat *setGID* (*set-group-ID*). Si el bit *set-group-ID* està actiu, l'identificador efectiu del grup d'usuaris del procés que ha executat (*exec*) el fitxer, prendrà com a valor l'identificador del grup d'usuaris del propietari del fitxer executable. L'identificador real del grup d'usuaris no es modificarà.

2.1.6 Estat del procés

Com en tot sistema operatiu, els processos poden estar en diferents estats: en execució, bloquejats, preparats, *zombie*, ...

En Unix cal destacar un estat particular que tenen alguns processos després de la seva destrucció: l'estat *zombie*. Aquest estat correspon a un procés que ja no pot tornar a executar-se perquè ja ha finalitzat, però que encara està present en el sistema perquè no ha pogut alliberar tots els seus recursos. La raó de l'estat *zombie* té a veure amb la jerarquia de processos de Unix que s'origina en la creació de processos i es manté fins a la seva desaparició.

Cada procés és fill del seu procés pare que és el responsable d'alliberar els recursos dels seus processos fill en el moment de la seva finalització. Per alliberar els processos i eliminar-los del tot del sistema, és necessari que el procés pare se sincronitzi (crida *wait*) amb la finalització (*exit*) dels seus processos fill. En aquest procés de sincronització, el procés fill informa al seu pare de la causa de la seva mort al mateix temps que allibera tots els seus recursos i desapareix del sistema. Si un procés finalitza sense aquesta sincronització pare-fill, el procés passa a l'estat *zombie* i es quedarà en aquest estat fins que pugui alliberar els recursos pendents. Si el procés pare desaparegués sense haver realitzat la sincronització (*wait*) amb els seus processos fill, els processos fill passaran a ser adoptats pel procés *init* (procés 1) i aquest procés primogènit els alliberarà de l'estat *zombie* i desapareixeran del sistema.

És important tenir en compte aquesta característica dels processos i evitar, sempre que sigui possible, que no quedi cap procés en estat *zombie* ja que ocuparia inútilment recursos del sistema. Per aquesta raó, els processos esperaran amb la crida a sistema *wait* la finalització dels seus fills i, d'aquesta manera, faran desaparèixer del sistema els processos que ja han finalitzat.

2.2 La creació de nous processos

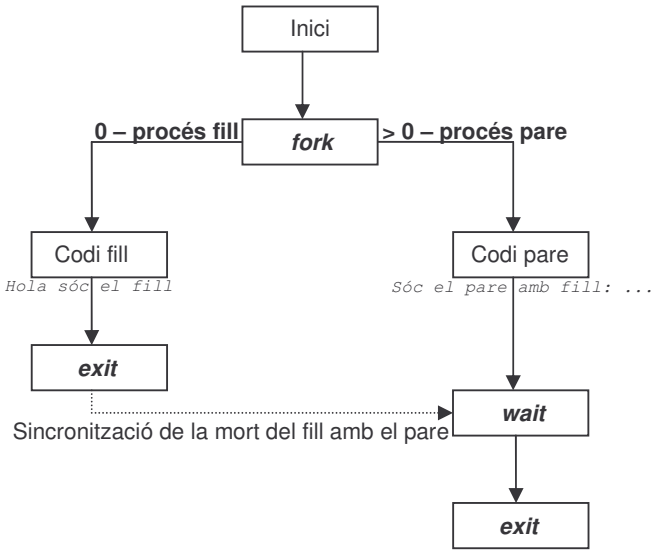
La creació de nous processos és una de les accions més importants que permet que es puguin realitzar noves accions per part dels diferents usuaris. Cada nou procés s'executarà de forma concurrent amb els altres processos i haurà de compartir els recursos (processador, memòria, ...) del sistema. A diferència d'altres sistemes operatius, la creació de processos a Unix es realitza d'una manera molt senzilla,

mitjançant una crida a sistema anomenada *fork* que no utilitza cap paràmetre. El resultat de l'execució de la crida *fork* és l'aparició d'un nou procés, fill del procés creador i que s'executarà concurrentment amb la resta de processos del sistema, que hereta un gran nombre de característiques del seu pare. Cal destacar, per exemple, que el nou procés té el mateix codi, les mateixes dades, la mateixa pila i el mateix valor del comptador de programa que el procés creador. Això no vol dir que comparteixi codi i dades, sinó que es crea un nou procés on es copia el codi i les dades del procés que l'ha creat. Només hi ha una dada que és diferent: el valor retornat per la crida *fork*.

Efectivament, la crida fork retorna dues vegades, una vegada en el procés que l'ha invocat i una altra, en el nou procés que, com ja s'ha dit, té el mateix codi que el procés creador i, per tant, començarà la seva execució just després d'haver acabat la funció *fork*. En el procés pare, la crida *fork* retorna l'identificador del procés fill i en el procés fill retorna el valor 0. És gràcies als valors diferents de retorn que es pot distingir entre pare i fill i, per tant, es pot modificar el comportament dels dos processos. Tot seguit es mostra un petit programa que crea un nou procés:

```
main()
{
      int process, st, pid;
      char s[80];

      switch (process = fork())
      {
            case -1:

                  /* En cas d'error el procés acaba */

                  sprintf(s, "Error del fork\n");
                  write(2, s, strlen(s));
                  exit(1);

            case 0:

                  /* Procés fill - Escriu i acaba */

                  sprintf(s, "Hola sóc el fill\n");
                  write(1, s, strlen(s));
                  exit(0);

            default:

                  /* Procés pare - Espera acabament fill i escriu */

                  sprintf(s, "Sóc el pare amb fill: %d\n", process);
                  write(1, s, strlen(s));
                  pid = wait(&st);
                  exit(0);
      }
}
```

L'evolució dels dos processos es mostra en el diagrama següent:



En el codi anterior es poden distingir clarament els possibles valors retornats per la crida *fork*:

- Valor -1: representa un error i que no s'ha pogut crear el nou procés
- Valor 0: és el valor que es retorna al nou procés (procés fill)
- Valor > 0: és el valor que es retorna al procés creador (*pid* del nou procés)

En el cas que la crida *fork* no hagi produït cap error apareixerà un nou procés que tindrà replicat (no compartit) el mateix codi i les mateixes dades que el procés creador i amb el mateix valor del comptador de programa. Per tant, començarà la seva execució en el punt on el procés creador l'ha creat, és a dir, just retornar de la crida *fork*. Per tant, el nou procés començarà la seva execució pel *case 0* del *switch*, escriurà un missatge i acabarà (*exit*):

```
case 0:

      /* Procés fill - Escriu i acaba */

      sprintf(s, "Hola sóc el fill\n");
      write(1, s, strlen(s));
      exit(0);
```

En canvi el pare, després d'haver creat el fill seguirà la seva execució pel *case default* del *switch*, escriurà l'identificador del fill que li ha retornat la crida *fork* (variable *process*), esperarà l'acabament del fill (*wait*) i acabarà (*exit*):

```
default:

      /* Procés pare - Espera acabament fill i escriu */

      sprintf(s, "Sóc el pare amb fill: %d\n", process);
      write(1, s, strlen(s));
      pid = wait(&st);
      exit(0);
```

La sortida dels processos pare i fill es pot donar en qualsevol ordre ja que l'execució és concurrent. Pot ser que escrigui primer el pare o que escrigui primer el fill:

```
Sóc el pare amb fill: ...
Hola sóc el fill
```

o

```
Hola sóc el fill
Sóc el pare amb fill: ...
```

Es important remarcar la importància de sincronitzar-se amb la desaparició dels fills (*wait*). Tal i com s'ha comentat abans, si el procés creador no espera la finalització dels seus fills, els processos que es moren no poden alliberar tots els seus recursos i queden en estat *zombie*. Si en l'exemple anterior el procés creador hagués acabat sense executar la crida a sistema *wait*, el procés fill hauria quedat en estat *zombie* fins que el procés creador hagués mort i el procés *init* hagués adoptat el nou procés i hagués alliberat els seus recursos.

En altres apartats s'aniran comentant altres aspectes importants de l'herència entre processos relacionats amb l'entrada/sortida i amb la comunicació entre processos.

2.3 La transformació d'un procés (exec)

Com s'ha vist a l'apartat anterior la creació de processos de Unix es limita a crear un nou procés idèntic amb el procés pare. En realitat, però, quan es crea un nou procés normalment es vol que faci coses molt diferents de les que fa el seu creador. Per aquesta raó, cal disposar d'una funció que ens permeti canviar del tot un determinat procés.

La crida a sistema *exec* permet canviar tot el codi, les dades i la pila d'un procés i carregar un nou codi i unes noves dades emmagatzemades en un fitxer executable. Aquesta crida és la que realment permet executar programes després d'haver creat un nou procés i, habitualment, s'utilitzarà immediatament després de la crida *fork*. Un cop acabada la crida *exec*, el codi del procés que l'ha invocat haurà desaparegut del tot i, per tant, la crida no retornarà mai, és a dir, les sentències del programa que puguin haver-hi després de la crida mai no s'executaran.

El codi que es mostra tot seguit crea un nou procés i, immediatament després, el procés fill executa (crida a sistema *execlp*) el fitxer executable *ls*. Per tant, el nou procés ja no manté el mateix codi ni les mateixes dades que el procés creador, sinó que ha transformat totalment el seu codi i les seves dades segons el contingut del fitxer executable *ls*. Per aquesta raó, el procés no executarà mai les sentències de després de la crida, excepte en el cas que *exec* produeixi un error i, per tant, el canvi d'imatge no es pugui realitzar. Aquesta *mutació* només afecta a les dades i al codi, en canvi el procés segueix essent el mateix amb el mateix identificador (*pid*) i els mateixos identificadors d'usuari, llevat que el fitxer executable tingués actiu el bit *setUID* o el bit *setGID*, tal i com s'ha explicat prèviament.

```
main()
{
      int st;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En cas d'error el procés acaba */

                  sprintf(s, "Error del fork\n");
                  write(2, s, strlen(s));
                  exit(1);

            case 0:

                  /* Procés fill - Executarà ls */
                  /* Carrega el codi de ls */

                  execlp("ls", "ls", (char *)0);

                  /* Si arriba aquí, hi ha hagut error i acaba */

                  sprintf(s, "Error exec\n");
                  write(2, s, strlen(s));
                  exit(1);

            default:

                  /* Procés pare - Esperem acabament fill i acabem */

                  wait(&st);
                  exit(0);
      }
}
```

2.4 Crides a sistema de gestió de processos

En aquest apartat es presenten les crides següents relacionades amb la gestió de processos:

- *fork*: crea un nou procés
- *exec*: reemplaça la imatge del procés amb una nova imatge
- *exit*: finalitza un procés
- *wait*: espera la finalització d'un procés fill
- *getpid*: obté l'identificador del procés
- *getppid*: obté l'identificador del procés pare
- *getpgrp*: obté l'identificador del grup de processos del procés
- *setpgid*: canvia l'identificador del grup de processos d'un procés

2.4.1 fork

Sintaxi

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Descripció

La crida *fork* no té cap paràmetre i la seva funció és crear un nou procés. El nou procés (procés fill) és una còpia exacta del procés creador (procés pare). El procés fill hereta tot l'entorn del procés pare. Entre els atributs heretats cal destacar:

- Identificadors d'usuari i de grup, tant real com efectiu.
- Canals (*file descriptors*) oberts.
- Programació dels *signals.*
- Segments de memòria compartida.
- Bit de *set-user-ID.*
- Bit de *set-group-ID.*
- Identificador de grup (*process group Id*).
- Directori de treball (*current working directory*).
- Directori arrel (*root directory*).
- Màscara de creació de fitxers (*umask*).

El procés fill es distingeix del procés pare en els aspectes següents:

- L'identificador de procés del fill és diferent de l'identificador del pare.
- El procés fill té un identificador de procés pare diferent (és l'identificador del procés que l'ha creat).
- El procés fill té la seva pròpia còpia dels canals (*file descriptors*) del pare. Cada canal del fill comparteix amb el canal del pare el mateix punter al fitxer (punter de lectura/escriptura).
- El conjunt de *signals* pendents està buit.
- No s'hereta cap operació d'entrada/sortida asíncrona.

Valor retornat

Si el fork s'executa correctament retornarà el valor 0 al procés fill i retornarà l'identificador del procés fill (PID) al procés pare. En cas d'error, la crida retornarà el valor -1 al procés pare, no es crearà cap procés i la variable *errno* indicarà l'error produït.

Errors

La crida *fork* fallarà si:

- *EAGAIN*: s'ha superat el nombre màxim de processos possibles per usuari o la quantitat total de memòria del sistema disponible és insuficient per duplicar el procés.
- *ENOMEM*: no hi ha espai de *swap* suficient.

2.4.2 exec

La crida a sistema exec s'ofereix amb diferents sintaxis que faciliten la seva utilització en funció dels paràmetres utilitzats: execl, execv, execle, execve, execlp, execvp.

Sintaxi

```
#include <unistd.h>

int execl(char *path, char *arg0, ..., char *argn, char * /*NULL*/);

int execv(char *path, char *argv[]);

int execle(char *path, char *arg0, ..., char *argn, char * /*NULL*/, char *envp[]);

int execve(char *path, char *argv[], char * envp[]);

int execlp(char *file, char *arg0, ..., char *argn, char * /*NULL*/);

int execvp(char *file, char *argv[]);
```

Descripció

Cadascuna de les funcions de la família *exec* reemplaça la imatge (codi i dades) del procés que la invoca amb una nova imatge. La nova imatge es construeix a partir d'un fitxer executable que es passa com a paràmetre. No es retorna cap valor en el cas que la crida s'executi correctament, ja que la imatge del procés que la invoca és eliminada per la nova imatge.

Els canals (*file descriptors*) oberts del procés que invoca la crida romanen oberts després del canvi d'imatge, excepte aquells que tenen el flag *close-on-exec* actiu.

Els *signals* definits, en el procés que invoca la crida, amb acció per defecte o que han de ser ignorats es mantenen igual després del canvi d'imatge. Els *signals* programats amb alguna funció en el procés que invoca la crida canvien la seva programació a l'acció per defecte després del canvi d'imatge, ja que el codi de les funcions amb què havien estat programats desapareix després de la crida *exec*.

Si el bit *set-user-ID* està actiu, l'identificador efectiu d'usuari del procés amb la nova imatge prendrà com a valor l'identificador del propietari del fitxer d'imatge. Igualment si el bit *set-group-ID* està actiu, l'identificador efectiu de grup del procés amb la nova imatge prendrà com a valor l'identificador del grup del fitxer d'imatge. Els identificadors reals d'usuari i de grup es mantindran.

Els segments de memòria compartida del procés que invoca la crida es desassignaran de la nova imatge.

La nova imatge del procés mantindrà, entre d'altres, els atributs següents:

- L'identificador del procés (*PID*).
- L'identificador del procés pare (*parent process ID*).
- L'identificador del grup (*process group ID*)
- L'identificador real d'usuari i de grup (*real user ID i real group ID*)

- El directori de treball (*current working directory*).
- El directori arrel (*root directory*).
- La màscara de creació de fitxers (*umask*).
- Els *signals* pendents.

Paràmetres

- *path* apunta al nom complet que identifica el nou fitxer d'imatge.
- *file* s'utilitza per construir el nom complet que identifica el nou fitxer d'imatge. Si el paràmetre *file* conté una barra inclinada ('/'), aleshores s'utilitza com a nom complet del nou fitxer d'imatge. En cas contrari, el camí complet del nom del fitxer s'obté mitjançant una cerca en els directoris inclosos a la variable d'entorn *PATH*.
- Els paràmetres representats per *arg()...* són punters a cadenes de caràcters. Aquests paràmetres són la llista d'arguments que es passaran a la nova imatge. La llista s'acaba amb un punter nul. El paràmetre *arg0* indicarà el nom que s'associarà amb el procés iniciat per la funció *exec*.
- *argv* és un punter a una taula de cadenes de caràcters. L'última entrada de la taula ha de ser un punter nul. Els elements de la taula són la llista d'arguments que es passaran a la nova imatge. El valor de l'entrada *argv[0]* indicarà el nom que s'associarà amb el procés iniciat per la funció *exec*.
- *envp* és un punter a una taula de cadenes de caràcters. L'última entrada de la taula ha de ser un punter nul. Els valors d'aquest paràmetre constitueixen l'entorn per a la nova imatge.

Valor retornat

La crida retornarà el valor -1 en el cas d'un error en la seva execució i la variable *errno* indicarà l'error produït. No es retorna cap valor en el cas que la crida s'executi correctament, ja que la imatge del procés que la invoca és eliminada per la nova imatge.

Errors

La crida *exec* fallarà, entre d'altres raons, si:

- *EACCES*: no es té permís per cercar en un del directoris que apareixen en el nom complet del fitxer d'imatge, o el fitxer amb la nova imatge no és un fitxer regular, o el fitxer amb la nova imatge no es pot executar.
- *EAGAIN*: la quantitat de memòria disponible del sistema en el moment de llegir el fitxer imatge és insuficient.
- *EFAULT*: algun dels paràmetres apunta a una adreça il·legal.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *exec*.
- *ELOOP*: s'han trobat massa enllaços simbòlics durant la traducció del paràmetre *path* o *file*.
- *ENAMETOOLONG*: la longitud del paràmetre *path* o *file* excedeix la mida permesa (PATH_MAX).
- *ENOENT*: algun component del paràmetre *path* o *file* no existeix o està buit.
- *ENOMEM*: la nova imatge del procés necessita més memòria de la permesa.
- *ENOTDIR*: algun component del prefix del paràmetre *path* o *file* no és un directori.

2.4.3 exit

Sintaxi

```
#include <stdlib.h>

void exit(int status);
```

Descripció

La crida *exit* té com a funció finalitzar el procés que la invoca amb les conseqüències següents:

- Es tanquen tots els canals (*file descriptors*) del procés.
- Si el procés pare està executant la crida *wait* se li notifica la finalització del fill i se li passa el valor dels vuit bits de menys pes del paràmetre *status*. Si el procés pare no està executant la crida *wait*, l'estat del fill (valor del paràmetre *status*) se li passarà en el moment que l'executi.
- Si el procés pare no està executant la crida *wait* en el moment que el procés invoca la crida *exit*, aleshores el procés que invoca *exit* es transforma en un procés *zombie*. Un procés *zombie* és un procés inactiu que només ocupa una entrada de la taula de processos i s'eliminarà completament en el moment que el seu procés pare executi la crida *wait*.
- S'envia un *signal* SIGCHLD al seu pare.
- S'allibera tota la memòria assignada al procés.
- Es desassignen els segments de memòria compartida.

Paràmetres

El paràmetre *status* emmagatzema, en els seus 8 bits (bits 0377) de menys pes, el valor que el procés fill passarà al seu pare quan el procés pare executi la crida *wait*.

Valor retornat

La crida *exit* no retorna mai al procés que la invoca.

Errors

No hi ha cap error definit.

2.4.4 wait

Sintaxi

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

Descripció

La crida *wait* atura l'execució del procés que la invoca fins que està disponible la informació de l'estat d'acabament d'algun dels seus fills, o fins que arriba un *signal* que té una funció programada o que provoca la finalització del procés. Si la informació de l'estat d'acabament d'algun fill està disponible abans d'invocar la crida, *wait* retornarà immediatament.

Si *wait* finalitza perquè l'estat d'acabament d'algun fill està disponible, aleshores retornarà l'identificador del fill que ha acabat.

Si un procés acaba sense haver esperat (*wait*) la finalització dels seus fills, l'identificador del procés pare de tots els seus fills prendrà el valor 1. És a dir, els seus processos fill són heretats pel procés d'inicialització (*init*) que es converteix en el seu pare.

Paràmetres

El paràmetre *stat_loc* és un punter on s'emmagatzemarà l'estat del procés fill que ha acabat.

Si la crida ha retornat perquè estava disponible l'estat d'acabament d'algun fill, aleshores l'estat del procés fill que ha acabat s'emmagatzemarà a l'adreça apuntada per *stat_loc* de la manera següent:

- Si el procés fill acaba per l'execució de la crida *exit*, els vuit bits de menys pes de la variable apuntada per *stat_loc* valdran 0 i els vuit bits de més pes contindran el valor dels vuit bits de menys pes de l'argument passat a la crida *exit*.
- Si el procés fill acaba a causa d'un *signal*, els vuit bits de més pes de la variable apuntada per *stat_loc* valdran 0 i els vuit bits de menys pes contindran el número del *signal* que ha causat la finalització del procés.

Valor retornat

Si *wait* retorna a causa de la finalització d'un procés fill, aleshores la crida retorna l'identificador del procés que ha acabat. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *wait* fallarà si:

- *ECHILD*: el procés que invoca la crida no té cap fill viu.

- *EINTR*: la crida ha estat interrompuda per un *signal*.

2.4.5 getpid

Sintaxi

```
#include <unistd.h>

pid_t getpid(void);
```

Descripció

La crida *getpid* no té cap paràmetre i retorna l'identificador del procés que la invoca.

Valor retornat

La crida retornarà l'identificador del procés que la invoca. No hi haurà cap cas que produeixi error.

Errors

No hi ha cap error definit.

2.4.6 getppid

Sintaxi

```
#include <unistd.h>

pid_t getppid(void);
```

Descripció

La crida *getppid* no té cap paràmetre i retorna l'identificador del procés pare del procés que la invoca.

Valor retornat

La crida retornarà l'identificador del procés pare del procés que la invoca. No hi haurà cap cas que produeixi error.

Errors

No hi ha cap error definit.

2.4.7 getpgrp

Sintaxi

```
#include <unistd.h>

pid_t getpgrp(void);
```

Descripció

La crida *getpgrp* no té cap paràmetre i retorna l'identificador del grup de processos (*process group ID*) del procés que la invoca.

Valor retornat

La crida retornarà l'identificador del grup de processos del procés que la invoca. No hi haurà cap cas que produeixi error.

Errors

No hi ha cap error definit.

2.4.8 setpgid

Sintaxi

```
#include <sys/types.h>
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Descripció

La crida *setpgid* assigna el valor *pgid* a l'identificador del grup de processos (*process group ID*) del procés amb identificador *pid*.

Si *pgid* és igual a *pid*, el procés amb identificador *pid* esdevindrà el líder del grup. Altrament el procés amb identificador *pid* esdevindrà un membre d'un grup ja existent.

Si *pid* és igual a 0 s'utilitza com a *pid* l'identificador de procés del procés que ha realitzat la crida.

Si *pgid* és igual a 0, el procés amb identificador *pid* esdevindrà el líder del grup de processos.

Paràmetres

- *pid* és l'identificador del procés al qual se li vol canviar el seu identificador del grup de processos.
- *pgid* és el valor de l'identificador del grup de processos que es vol assignar al procés.

Valor retornat

Si la crida s'executa correctament retornarà el valor 0. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *setpgid* fallarà, entre d'altres raons, si:

- *EACCES*: el *pid* correspon a l'identificador d'un fill del procés que invoca la crida i que ha executat una crida a sistema *exec*.
- *EINVAL*: el paràmetre *pgid* té un valor negatiu o superior al màxim permès.
- *ESRCH*: el paràmetre *pid* no coincideix amb l'identificador del procés que invoca la crida ni amb l'identificador de cap dels seus fills.

2.5 Exemples de les crides de gestió de processos

2.5.1 Exemple 1

Aquest exemple mostra la creació d'un nou procés (*fork*) i la sincronització del procés pare amb la finalització del procés fill (*exit* i *wait*). El nou procés s'executarà concurrentment amb el procés pare.

El procés pare realitzarà les accions següents:

a) Crea el procés fill (*fork*)
b) Escriu el seu identificador de procés (*write* i *getpid*)
c) Espera la finalització del procés fill (*wait*)
d) Escriu l'identificador del fill que ha mort (*write*)
e) Acaba (*exit*)

El procés fill, per la seva banda, realitzarà concurrentment amb el procés pare les accions següents:

a) Escriu el seu identificador de procés (*write* i *getpid*)
b) Escriu l'identificador de procés del seu pare (*write* i *getppid*)
c) Acaba (*exit*)

L'escriptura b) del pare i les escriptures a) i b) del fill poden aparèixer en qualsevol ordre en funció de l'execució concurrent dels dos processos:

```
Hola sóc el fill: ...          {sortida a) del fill}
El meu pare és: ...            {sortida b) del fill}
Hola sóc el pare: ...          {sortida b) del pare}
```

o

```
Hola sóc el pare: ...          {sortida b) del pare}
Hola sóc el fill: ...          {sortida a) del fill}
El meu pare és: ...            {sortida b) del fill}
```

o

```
Hola sóc el fill: ...          {sortida a) del fill}
Hola sóc el pare: ...          {sortida b) del pare}
El meu pare és: ...            {sortida b) del fill}
```

En canvi, l'escriptura d) del pare sempre sortirà al final de tot ja que el pare, abans d'escriure, espera la finalització del fill. En la darrera escriptura el pare retornarà l'identificador del fill que ha mort i que li ha estat retornat per la crida a sistema *wait*. Per tant, l'última escriptura serà:

```
El fill finalitzat és: ...     {sortida d) del pare}
```

```
#include <errno.h>


void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int st, pid;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En cas d'error el procés acaba */

                  error("Fork");

            case 0:

                  /* Procés fill - Escriu i acaba */

                  sprintf(s, "Hola sóc el fill: %d\n", getpid());
                  write(1, s, strlen(s));
                  sprintf(s, "El meu pare és: %d\n", getppid());
                  write(1, s, strlen(s));
                  exit(0);

            default:

                  /* Procés pare - Escriu i espera acabament fill */

                  sprintf(s, "Hola sóc el pare: %d\n", getpid());
                  write(1, s, strlen(s));
                  pid = wait(&st);

                  /* Escriu i acaba */

                  sprintf(s, "El fill finalitzat és: %d\n", pid);
                  write(1, s, strlen(s));
                  exit(0);
      }
}
```

2.5.2 Exemple 2

Aquest exemple mostra la creació d'un nou procés (*fork*), el canvi d'imatge (*exec*) i la sincronització del pare amb la finalització del fill (*exit* i *wait*). El procés fill executa (*exec*) la comanda `ls -l /usr/bin` i acaba (*exit*). El procés pare espera la finalització del seu fill (*wait*) i acaba (*exit*).

```
#include <errno.h>

void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int st;

      switch (fork())
      {
            case -1:

                  /* En cas d'error el procés acaba */

                  error("Fork");

            case 0:

                  /* Procés fill - Executarà ls */
                  /* Carrega el codi de ls */
                  /* Rep com a paràmetres -l i /usr/bin */

                  execlp("ls", "ls", "-l", "/usr/bin", (char *)0);

                  /* Si arriba aquí, execlp ha fallat */

                  error("Executant ls");

            default:

                  /* Procés pare */
                  /* Espera que acabi el fill */

                  wait(&st);
                  exit(0);
      }
}
```

3 L'entrada/sortida

Al llarg d'aquest capítol es descriurà l'estructura general del sistema de fitxers, els diferents tipus de fitxers, la independència de dispositius i la redirecció de l'entrada/sortida, així com les principals crides per gestionar les entrades i sortides.

3.1 L'entrada/sortida i el sistema de fitxers

L'entrada/sortida de Unix i el seu sistema de fitxers estan íntimament relacionats tant pel conjunt de crides a sistema uniformes i idèntiques per accedir a qualsevol dispositiu o fitxer, com també per l'existència de diversos tipus de fitxers que inclouen els propis dispositius. Tot dispositiu de Unix es reconeix en el sistema com un fitxer de tipus dispositiu, que pot ser utilitzat amb les mateixes crides a sistema (*open, close, read, write, ...*) que per als fitxers regulars que contenen informació dels usuaris.

Tipus de fitxers

El sistema de fitxers de Unix inclou diversos tipus de fitxers que representen tant els fitxers convencionals, que contenen informació dels usuaris, com els propis dispositius d'entrada/sortida i altres recursos del sistema. Els tipus de fitxers més destacats de Unix són:

- *Directori*: fitxer que conté referències a d'altres fitxers i que constitueix l'element fonamental de la jerarquia de fitxers de Unix. Dins dels fitxers tipus directoris hi ha entrades que aparellen el número d'un *inode* (l'element que descriu cada fitxer) amb el nom (*link*) que se li dóna al fitxer en aquell directori.
- *Fitxer regular*: fitxer que conté informació general sense cap estructura en particular.
- *Dispositiu*: fitxer especial que representa cadascun dels dispositius del sistema. Hi ha dos tipus de dispositiu: els dispositius de caràcter (per exemple terminals) i els dispositius de bloc (per exemple discs).
- *Enllaç simbòlic* (*soft link*): fitxer que conté el nom d'un altre fitxer al qual representa, de tal manera que l'accés a l'enllaç simbòlic és com accedir al fitxer que enllaça.
- *Named pipes*: dispositiu de comunicació entre processos.

Els noms dels fitxers (*hard links* i *soft links*)

Una característica del sistema de fitxers de Unix és la possibilitat que els fitxers puguin tenir més d'un nom (*hard links*). Fins i tot, poden existir fitxers que no tinguin nom i que siguin accessibles i utilitzables pels processos que el tenen obert. Gràcies a aquesta característica, es pot fer visible i accessible un fitxer des de diversos punts del sistema de fitxers. I això és possible per la separació existent entre la informació del fitxer (*inode*) i el seu nom.

Els *inodes* són unes estructures que contenen tota la informació d'un fitxer excepte el nom, o noms del fitxer, i les seves dades, que estan emmagatzemades en els blocs de dades del sistema de fitxers. Dins de l'*inode* s'hi pot trobar, entre d'altres, les informacions següents:

- Identificador del propietari del fitxer
- Identificador del grup d'usuaris del fitxer
- Mida del fitxer
- Data de creació, data de l'últim accés i data de l'última modificació
- Tipus de fitxer
- Permisos d'accés per a l'usuari, per al grup i per a la resta d'usuaris
- Nombre de noms (*hard links*) del fitxer
- Punters als blocs de dades

Els *inodes* estan numerats de l'1 al nombre màxim d'*inodes* d'un determinat sistema de fitxers.

El nom del fitxer està emmagatzemat dins dels directoris. Cada entrada a un directori és una associació entre un número d'*inode* i un nom. D'aquesta manera es poden tenir tants noms com es vulgui de cada fitxer. Només cal associar el mateix número d'*inode* (que representa el fitxer) amb diferents noms dins dels directoris que es vulgui.

Entrada d'un directori

Número d'*inode*	Nom del fitxer

El nombre de noms (*hard links*) d'un fitxer es guardarà en l'*inode* corresponent i servirà perquè el sistema operatiu pugui detectar quan un fitxer ja no té cap nom i, per tant, quan un fitxer es pot eliminar del sistema. Tanmateix, que un fitxer no tingui cap nom no és una condició suficient perquè desaparegui el seu *inode* i la informació que conté. Per poder esborrar del tot un fitxer cal, a més de no tenir cap nom, que no hi hagi cap procés que l'estigui utilitzant (fitxer obert). Per tant, a Unix és possible que un procés tingui un fitxer obert, n'esborri el darrer nom i segueixi treballant amb ell sense que cap altre procés hi pugui accedir, ja que no disposa de cap nom visible en el sistema de fitxers. Aquesta manera de treballar és usual entre els programes que utilitzen fitxers temporers durant la seva execució. En aquest cas, el fitxer desapareix quan el tanca el darrer procés que el tenia obert (vegeu l'exemple 2 d'aquest capítol).

Unix utilitza la paraula *link* (enllaç) per referir-se als noms d'un fitxer i distingeix entre els *hard links* (que són els que s'acaben d'explicar) que estan comptabilitzats dins de l'*inode* de cada fitxer i els *soft links* (enllaços simbòlics). Els *soft links* són un tipus especial de fitxer que s'utilitza per poder accedir a un altre fitxer des de qualsevol lloc del sistema de fitxers. Els *soft links* contenen el nom del fitxer al qual apunten i tenen un conjunt de característiques que els diferencien dels *hard links*:

1. Els *soft links* són fitxers, en canvi els *hard links* no són res més que una entrada en un directori que associa un nom amb un número d'*inode*.
2. Els *soft links* no són coneguts pel fitxer que representen. A diferència dels *hard links* que són comptabilitzats dins de l'inode de cada fitxer, els *soft links* es creen i es destrueixen sense que el fitxer apuntat en tingui cap constància. Per aquesta raó, és possible que puguin existir *soft links* que apuntin a fitxers inexistents, perquè els fitxers apuntats hagin desaparegut del sistema després de la creació del *soft link*.
3. Els *soft links* es poden situar en un sistema de fitxers diferent al del fitxer que representen. En canvi els *hard links* (les parelles "*inode* – nom" dins dels directoris) han d'estar situades totes dins del mateix sistema de fitxers,

per evitar així les ambigüitats amb els números d'*inode* que coincideixen entre sistemes de fitxers diferents.

3.2 La independència de dispositius i la redirecció de l'entrada/sortida

Per garantir la independència de dispositius de les aplicacions és necessari disposar d'un conjunt de crides homogènies d'entrada/sortida i de dispositius virtuals (canals) que puguin ser associats amb qualsevol dispositiu real o fitxer. Unix ofereix un conjunt de crides idèntiques per gestionar l'entrada/sortida independentment del dispositiu o fitxer que s'estigui utilitzant. Aquestes crides utilitzen dispositius virtuals que poden associar-se a qualsevol dispositiu real o fitxer.

Els dispositius virtuals (o canals) de Unix anomenats *file descriptors* s'agrupen en una taula independent per a cada procés. Cada dispositiu virtual s'identifica amb el valor d'entrada a la taula, de tal manera que el primer dispositiu virtual serà el canal 0 i així successivament. Les crides a sistema de Unix de lectura (*read*) i escriptura (*write*) utilitzen exclusivament els dispositius virtuals i, d'aquesta manera, s'independitzen dels dispositius reals o fitxers que estiguin associats a cada *file descriptor*. Per associar els dispositius virtuals amb un determinat dispositiu o fitxer s'utilitza la crida *open* (obrir). Per alliberar un determinat dispositiu s'utilitza la crida a sistema *close* (tancar). Per tant, abans de poder utilitzar un determinat dispositiu o fitxer caldrà obrir-lo (*open*) per així associar un determinat dispositiu virtual amb el fitxer o dispositiu real. Un cop tinguem el dispositiu virtual associat ja podrem escriure (write) o llegir (read) sobre aquest dispositiu virtual. Un cop haguem finalitzat amb l'ús del dispositiu real o fitxer podrem alliberar-lo juntament amb el canal virtual utilitzant la crida *close*.

A la pàgina següent es mostra el codi d'un procés que obre un fitxer existent i llegeix tot el seu contingut caràcter a caràcter. Un cop ha acabat la lectura, és tanca el canal i s'alliberen els recursos. Cal fixar-se que la crida *open* retorna el valor del primer dispositiu virtual lliure de la taula de canals del procés que, en aquest cas, segurament serà el canal 3 tal i com s'explica tot seguit.

Per conveni, Unix considera que els canals 0, 1 i 2 s'utilitzaran com a canals estàndard d'entrada/sortida:

- 0: entrada estàndard (*stdin*)
- 1: sortida estàndard (*stdout*)
- 2: sortida d'error estàndard (*stderr*)

A l'exemple de la pàgina següent es pot veure que la funció *error* escriu els missatges d'error pel canal de sortida d'error estàndard (dispositiu virtual 2 – *stderr*). En canvi, el contingut del fitxer s'escriu caràcter a caràcter pel canal estàndard de sortida (dispositiu virtual 1 – *stdout*).

Tots els processos que s'executen des de l'intèrpret de comandes de Unix tenen oberts els canals 0, 1 i 2, ja que els nous processos hereten dels seus pares una taula de canals amb els mateixos canals oberts que el procés creador. L'intèrpret de comandes té oberts els canals estàndards i, per tant, tots els seus fills també els tindran oberts i associats amb els mateixos dispositius en el moment de la seva creació. Assumint aquest fet, es podria afirmar que la crida *open* de l'exemple retornaria el canal 3 (primera entrada lliure del procés) si fos l'intèrpret de comandes qui creés aquell procés.

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ... */
#include <errno.h>


void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int fd, n;
      char c;

      if ((fd = open("Datafile.dat", O_RDONLY)) < 0)
            error("Obertura del fitxer");

      /* Llegeix del fitxer mentre hi hagi informació   */
      /* i escriu el contingut per la sortida estàndard */

      while ((n=read(fd, &c, 1)) > 0)
            write(1, &c, 1);

      if (n<0)
            error("Lectura del fitxer");

      close (fd);
}
```

Anem a suposar que l'exemple anterior està emmagatzemat en un fitxer executable anomenat *testprogram* i que volem executar-lo però amb la sortida estàndard redireccionada cap un altre fitxer anomenat *output.dat*. Des de l'intèrpret de comandes caldria invocar la línia següent:

```
$ testprogram > output.dat
```

El símbol > representa la redirecció de la sortida estàndard del procés cap al fitxer indicat. Per realitzar aquesta redirecció des del programa es podria fer amb el codi següent:

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ...         */
#include <errno.h>

void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int st;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En cas d'error el procés acaba */

                  error("Error del fork");

            case 0:

                  /* Es tanca el canal estàndard de sortida */

                  close (1);

                  /* Es crea el fitxer que s'assignarà al dispositiu */
                  /* virtual 1 (stdin) que és el primer lliure       */

                  if (open("output.dat", O_WRONLY|O_CREAT, 0600) < 0)
                        error("Obertura del fitxer");

                  /* A partir d'aquest moment l'stdout del nou   */
                  /* procés ja està redireccionada cap al fitxer */

                  /* Procés fill - Executa testprogram */

                  execlp("testprogram", "testprogram", (char *)0);

                  /* Si arriba aquí, execlp ha fallat */

                  error("Executant testprogram");

            default:

                  /* Procés pare - Espera acabament fill i acaba */

                  wait(&st);
                  exit(0);
      }
}
```

En el programa anterior, per redireccionar la sortida estàndard del nou procés es procedeix de la forma següent:

1. Es tanca (*close*) el canal de sortida estàndard (*file descriptor* 1) a fi que quedi lliure i, per tant, es pugui assignar en el moment de fer una crida *open*.

2. Amb la crida *open* es crea el fitxer cap on es vol redireccionar la sortida estàndard. Aquesta crida buscarà el primer canal (dispositiu virtual) lliure de la taula de canals del procés. El canal 1 s'ha tancat prèviament i, per tant, serà el primer canal lliure. A partir d'aquest moment, doncs, el canal estàndard de sortida estarà associat al fitxer *output.dat* i tot el que s'escrigui per la sortida estàndard del procés (*file descriptor* 1), s'escriurà sobre aquest fitxer.
3. Un cop redireccionada la sortida estàndard del nou procés, s'invoca la crida a sistema *exec* amb el fitxer *testprogram* com a paràmetre i el procés canviarà el seu codi pel codi definit en el fitxer. La crida *exec* no produeix cap modificació en la taula de canals del procés i, per tant, el procés llegirà tot el fitxer *Datafile.dat* i l'escriurà caràcter a caràcter per la seva sortida estàndard que, en aquest cas, està redireccionada cap el fitxer *output.dat*.

3.3 La taula de fitxers oberts i el punter de lectura/escriptura

A l'apartat anterior s'ha explicat la manera de redireccionar l'entrada/sortida d'un procés gràcies a l'existència dels dispositius virtuals i d'un conjunt de crides homogènies. També s'ha vist que la taula de canals o dispositius virtuals s'hereta entre pares i fills, de tal manera que els canals oberts en el procés pare també estan oberts i apuntant als mateixos dispositius en el procés fill.

Cada canal està associat a un dispositiu o a un fitxer a través d'una entrada a una taula anomenada taula de fitxers oberts. Aquesta taula és global per a tot el sistema i, per tant, és la mateixa per a tots els processos. Dins d'aquesta taula de fitxers oberts s'hi emmagatzema, entre d'altres coses, el punter de lectura/escriptura dels fitxers oberts.

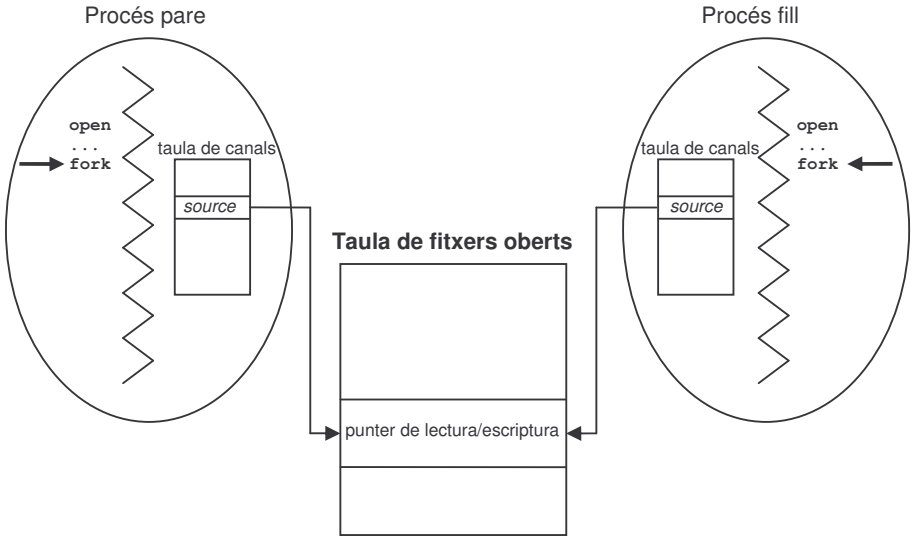
El punter de lectura/escriptura indica la posició on es farà la següent lectura o escriptura sobre el fitxer corresponent. En el moment d'obrir un fitxer, el punter de lectura/escriptura se situa, per defecte, a l'inici del fitxer. A mesura que es llegeix o s'escriu en un fitxer, el punter de lectura/escriptura va avançant automàticament tantes posicions com bytes s'hagin llegit o escrit. També es pot modificar el punter de lectura/escriptura amb una crida a sistema específica per a aquest propòsit (*lseek*).

Cada cop que s'obre o es crea un fitxer, a més d'ocupar una entrada de la taula de canals (*file descriptors*) del procés corresponent, també s'ocupa una nova entrada de la taula de fitxers oberts que s'associa amb el canal obert. Dins de l'entrada de la taula de fitxers oberts s'hi guardarà el punter de lectura/escriptura situat a l'inici del fitxer obert, llevat que s'indiqui una altra opció.

En la creació de processos, el nou procés no només hereta el contingut de la taula de canals del procés pare, sinó que els seus canals oberts apunten també a les mateixes entrades de la taula de fitxers oberts que el pare. Per tant, pare i fill compartiran el mateix punter de lectura/escriptura dels fitxers oberts que el fill hagi heretat en el moment de la seva creació.

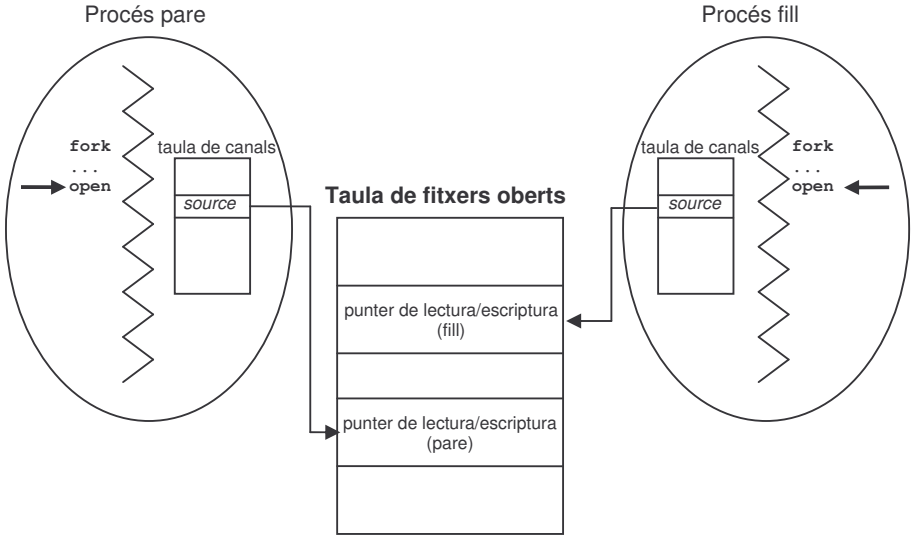
A l'exemple 4 i a l'exemple 5 d'aquest capítol es poden veure dos programes que mostren la diferència entre compartir un fitxer ja obert entre procés pare i fill (exemple 4), o obrir dues vegades i de forma independent el mateix fitxer entre pare i fill (exemple 5).

A l'exemple 4, el procés pare obre un fitxer abans de la creació del fill (canal *source*). Per tant, quan es crea el nou procés, els dos processos comparteixen el mateix punter de lectura/escriptura.



A la figura anterior es pot veure l'estat de la taula de canals dels dos processos i l'estat de la taula de fitxers oberts després de la creació del procés fill. El procés fill hereta la mateixa taula de canals que el pare i cada canal heretat apunta a la mateixa entrada compartida de la taula de fitxers oberts. Per aquesta raó, els punters de lectura/escriptura dels fitxers heretats són compartits entre pare i fill.

A l'exemple 5, el procés pare obre un fitxer (canal *source*) després d'haver creat el procés fill. El procés fill obre també el mateix fitxer. El resultat és que els dos processos accedeixen al mateix fitxer amb punters de lectura/escriptura independents:



A la figura anterior es pot veure l'estat de la taula de canals dels dos processos i l'estat de la taula de fitxers oberts, després de la creació del procés fill i després que els dos processos hagin obert el mateix fitxer de forma independent. En aquest cas, procés pare i fill tenen una entrada a la taula de fitxers oberts diferent i, per tant, no comparteixen el punter de lectura/escriptura en l'accés al fitxer.

3.4 Crides a sistema d'entrada/sortida

En aquest apartat es presenten les crides següents relacionades amb l'entrada/sortida i el sistema de fitxers:

- *creat*: crea un nou fitxer
- *open*: obre un fitxer
- *close*: tanca un canal (*file descriptor*)
- *read*: llegeix d'un fitxer
- *write*: escriu a un fitxer
- *lseek*: situa el punter de lectura/escriptura
- *dup*: duplica un canal obert
- *unlink*: esborra una entrada (*link*) d'un directori

3.4.1 creat

Sintaxi

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Descripció

La crida *creat* crea un nou fitxer regular o rescriu un fitxer existent.

Si el fitxer existeix, la seva longitud passa a ser 0 i no es modifica el seu propietari. Si el fitxer no existeix, l'identificador del propietari del nou fitxer serà l'identificador d'usuari efectiu del procés que invoca la crida i l'identificador de grup del fitxer serà l'identificador efectiu de grup del procés.

Els valors dels bits corresponents als permisos d'accés seran els indicats al paràmetre *mode* i modificats segons la màscara de creació (*umask*): es farà una *AND* bit a bit amb la màscara de creació complementada i, per tant, tots els bits amb valor 1, en la màscara de creació del procés, prendran el valor 0 en la màscara de permisos.

Si la crida acaba correctament, es retorna un canal (*file descriptor*) de només escriptura amb el punter de lectura/escriptura situat a l'inici del fitxer i el fitxer queda obert per a escriptura, encara que el paràmetre *mode* no ho permeti.

Aquesta crida és equivalent a:

open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)

Paràmetres

- *path* apunta al nom complet del fitxer.
- *mode* representa una màscara de bits que descriu els permisos amb què es crearà el fitxer, un cop modificada segons el valor de la màscara de creació (*umask*).

Valor retornat

Si la crida acaba correctament es retorna un valor enter no negatiu que correspon al primer canal (*file descriptor*) lliure disponible del procés. Altrament es retorna el valor negatiu -1, no es crea ni modifica cap fitxer i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *creat* fallarà, entre d'altres raons, si:

- *EACCES*: el procés no té permís per cercar en un dels components directori del nom del fitxer, el fitxer no existeix i el directori on s'ha de crear el nou fitxer no permet escriure o el fitxer existeix però no es tenen permisos d'escriptura.

- *EDQUOT*: no es pot crear el fitxer perquè l'usuari no disposa de més quota d'espai de blocs de disc o no disposa de més quota d'*inodes* en el sistema de fitxers.
- *EFAULT*: el paràmetre *path* apunta a una adreça il·legal.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *creat*.
- *EISDIR*: existeix un directori amb el mateix nom que el fitxer que es vol crear.
- *EMFILE*: el procés té massa fitxers oberts.
- *ENOENT*: algun component directori del paràmetre *path* no existeix o el paràmetre està buit.
- *ENOTDIR*: algun component del prefix del paràmetre *path* no és un directori.

3.4.2 open

Sintaxi

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, /* mode_t mode */...);
```

Descripció

La crida *open* obre un fitxer mitjançant l'establiment d'una connexió entre el fitxer (*path*) i un canal (*file descriptor*) del procés.

Es crea una nova entrada a la taula de fitxers oberts que representa al fitxer obert i el nou canal (*file descriptor*) apunta a aquesta entrada.

Si la crida acaba correctament es retorna un canal (*file descriptor*) amb el punter de lectura/escriptura situat a l'inici del fitxer. La modalitat d'accés del fitxer s'estableix segons el valor del paràmetre *oflag*. El paràmetre *mode* només s'utilitza si el paràmetre *oflag* inclou el valor O_CREAT.

El valor del paràmetre *oflag* es construeix amb una *OR* bit a bit de tots els valors inclosos com a paràmetres.

Paràmetres

- *path* apunta al nom complet d'un fitxer.
- *oflag* indicarà la modalitat d'accés del fitxer obert i pot prendre, entre d'altres, els valors següents:
  - Ha de tenir sempre únicament un d'aquests tres valors:
    - *O_RDONLY*: fitxer obert només per a lectura.
    - *O_WRONLY*: fitxer obert només per a escriptura
    - *O_RDWR*: fitxer obert per a lectura i escriptura.
  - Pot utilitzar-se qualsevol combinació dels valors següents:
    - *O_APPEND*: el punter de lectura/escriptura se situarà al final del fitxer
    - *O_CREAT*: crea el fitxer si no existeix. Si s'utilitza aquest valor cal afegir el paràmetre *mode* a la crida.
    - *O_EXCL*: si s'inclouen els valors *O_CREAT* i *O_EXCL*, la crida *open* fallarà si el fitxer ja existeix.
    - *O_NONBLOCK o O_NDELAY*: si s'inclou algun d'aquests dos valors les lectures (*read*) i escriptures (*write*) posteriors no provocaran bloqueig. Si hi ha els dos valors *O_NONBLOCK* té preferència.
    - *O_TRUNC*: si el fitxer existeix, és un fitxer regular i s'obre amb *O_RDWR* o *O_WRONLY*, aleshores la seva longitud passa a ser 0. El resultat d'utilitzar *O_TRUNC* amb *O_RDONLY* no està definit.
- *mode* representa una màscara de bits que descriu els permisos amb què s'obrirà el fitxer, un cop modificada segons el valor de la màscara de creació (*umask*).

Valor retornat

Si la crida acaba correctament es retorna un valor enter no negatiu que correspon al primer canal (*file descriptor*) lliure disponible del procés. Altrament es retorna el valor negatiu -1, no es crea ni modifica cap fitxer i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *open* fallarà, entre d'altres raons, si:

- *EACCES*: el procés no té permís per cercar en un dels components directori del nom del fitxer, o el fitxer existeix i els permisos especificats per *oflag* no estan permesos, o el fitxer no existeix i no es permet escriure en el directori on ha de crear-se, o s'ha especificat *O_TRUNC* i no es tenen permisos d'escriptura.
- *EDQUOT*: el fitxer no existeix, s'ha especificat *O_CREAT* i no es pot crear perquè l'usuari no disposa de més quota d'espai de blocs de disc, o no disposa de més quota d'*inodes* en el sistema de fitxers.
- *EEXIST*: s'han inclòs els valors *O_CREAT* i *O_EXCL* i el fitxer ja existeix.
- *EFAULT*: el paràmetre *path* apunta a una adreça il·legal.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *open*.
- *EISDIR*: el fitxer és un directori i s'hi intenta accedir amb *O_WRONLY* o *O_RDWR*.
- *EMFILE*: el procés té massa fitxers oberts.
- *ENOENT*: no s'ha inclòs el valor *O_CREAT* i el fitxer no existeix, o el valor *O_CREAT* s'ha inclòs però algun component directori del paràmetre *path* no existeix o el paràmetre està buit.
- *ENOTDIR*: algun component del prefix del paràmetre *path* no és un directori.

3.4.3 close

Sintaxi

```
#include <unistd.h>

int close(int fildes);
```

Descripció

La crida *close* tanca el canal (*file descriptor*) indicat en el paràmetre *fildes*. Tancar el canal significa fer-lo disponible perquè pugui ser assignat posteriorment per altres crides com *open*.

Un cop tancats tots els canals associats a una *pipe*, les dades que encara poguessin estar a la *pipe* són eliminades. Un cop tancats tots els canals associats a un fitxer, si el nombre d'enllaços (*links*) del fitxer és zero, l'espai ocupat pel fitxer en el sistema de fitxers s'allibera i el fitxer ja no serà accessible novament.

En el moment que es tanquen tots els canals associats a una entrada de la taula de fitxers oberts, aquesta entrada s'alliberarà.

Paràmetres

fildes indica el canal (*file descriptor*) que es vol tancar.

Valor retornat

Si la crida acaba correctament retorna el valor 0. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *close* fallarà, entre d'altres raons, si:

- *EBADF*: el paràmetre *fildes* no és un canal (*file descriptor*) vàlid.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *close*.
- *EIO*: s'ha produït un error d'entrada/sortida mentre es llegia o s'escrivia al sistema de fitxers.

3.4.4 read

Sintaxi

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
```

Descripció

La crida *read* llegeix d'un fitxer. La crida *read* intenta llegir el nombre de bytes indicats al paràmetre *nbyte* del fitxer associat al canal obert (*file descriptor*) *fildes*. Els bytes llegits s'emmagatzemen al buffer apuntat pel paràmetre *buf*.

Si el paràmetre *nbyte* és 0, la crida retornarà el valor 0 i no tindrà cap altre efecte.

En els fitxers on es permet situar (*lseek*) el punter de lectura/escriptura (per exemple els fitxers regulars), la crida *read* començarà la lectura en la posició indicada pel desplaçament del punter de lectura/escriptura associat al canal *fildes*. Al final de la lectura, el desplaçament del punter de lectura/escriptura del fitxer s'incrementarà el nombre de bytes que s'hagin llegit.

En els fitxers on no es permet situar el punter de lectura/escriptura (per exemple els terminals), la crida *read* sempre llegirà de la posició actual. En aquest cas el desplaçament del punter de lectura/escriptura associat al fitxer estarà indefinit.

Si s'intenta llegir del final de fitxer (*end-of-file*) o més enllà del final de fitxer, no es produirà cap lectura.

La lectura d'una *pipe* buida produirà els efectes següents:

- Si no hi ha cap procés que tingui la *pipe* oberta per escriptura, la crida *read* retornarà el valor 0 per indicar final de fitxer (*end-of-file*).
- Si algun procés té la *pipe* oberta per escriptura i el *flag O_NDELAY* de la *pipe* està actiu (*open*), la crida *read* retornarà el valor 0.
- Si algun procés té la *pipe* oberta per escriptura i el *flag O_NONBLOCK* de la *pipe* està actiu (*open*), la crida *read* retornarà el valor -1 i la variable *errno* valdrà *EAGAIN*.
- Si els *flags O_NDELAY* i *O_NONBLOCK* de la *pipe* no estan actius (*open*), la crida *read* es bloqueja fins que algun procés hi escrigui dades o fins que tots els processos que la tenen oberta per escriptura la tanquin.

La lectura d'un fitxer associat amb un terminal que no tingui dades disponibles produirà els efectes següents:

- Si el *flag O_NDELAY* del terminal està actiu (*open*), la crida *read* retornarà el valor 0.
- Si el *flag O_NONBLOCK* del terminal està actiu (*open*), la crida *read* retornarà el valor -1 i la variable *errno* valdrà *EAGAIN*.
- Si els *flags O_NDELAY* i *O_NONBLOCK* del terminal no estan actius (*open*), la crida *read* es bloquejarà fins que hi hagi dades disponibles.

La crida *read* llegeix dades que s'han escrit prèviament. Tanmateix, si es llegeix alguna part d'un fitxer anterior al final de fitxer (*end-of-file*) que no ha estat mai escrita, *read* retornarà bytes amb el valor 0. Aquesta situació pot passar en situar (*lseek*) el punter de lectura/escriptura més enllà del final de fitxer i escriure-hi alguna informació. L'espai situat entremig retornarà, en ser llegit, bytes amb el valor 0 fins que s'hi escrigui alguna cosa explícitament.

Si la crida acaba correctament i el paràmetre *nbyte* és més gran que 0, aleshores retornarà el nombre de bytes llegits. Aquest nombre mai no serà superior al valor del paràmetre *nbyte,* però sí que pot ser inferior si no hi ha tants bytes disponibles per ser llegits o la crida ha estat interrompuda per un *signal*.

Si la crida *read* és interrompuda per un *signal* abans que hagi llegit res, retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Si la crida *read* és interrompuda per un *signal* després d'haver llegit alguna cosa, retornarà el nombre de bytes llegits.

Paràmetres

- *fildes* canal obert associat al fitxer del qual es vol llegir.
- *buf* punter a un espai de memòria on es guardaran els bytes llegits.
- *nbyte* nombre de bytes que es volen llegir.

Valor retornat

Si la crida *read* acaba correctament retornarà un valor no negatiu que indicarà el nombre de bytes realment llegits del fitxer associat amb el canal (*file descriptor*) *fildes*. Aquest nombre mai no serà superior al valor del paràmetre *nbytes*. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *read* fallarà, entre d'altres raons, si:

- *EBADF*: el paràmetre *fildes* no és un canal obert (*file descriptor*) vàlid per a lectura.
- *EFAULT*: el paràmetre *buf* apunta a una adreça il·legal.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *read* i no hi ha hagut cap lectura.
- *EIO*: s'ha produït un error del dispositiu físic d'entrada/sortida.

3.4.5 write

Sintaxi

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Descripció

La crida *write* escriu a un fitxer. La crida *write* intenta escriure el nombre de bytes indicats al paràmetre *nbyte*, que estan emmagatzemats al buffer apuntat pel paràmetre *buf*, al fitxer associat al canal obert (*file descriptor*) *fildes*.

Si el paràmetre *nbyte* és 0, la crida retornarà el valor 0 i no tindrà cap altre efecte.

En els fitxers on es permet situar (*lseek*) el punter de lectura/escriptura (per exemple els fitxers regulars), la crida *write* començarà l'escriptura en la posició indicada pel desplaçament del punter de lectura/escriptura associat al canal *fildes*. Al final de l'escriptura, el desplaçament del punter de lectura/escriptura del fitxer s'incrementarà el nombre de bytes que s'hagin escrit.

En els fitxers on no es permet situar el punter de lectura/escriptura (per exemple els terminals), la crida *write* sempre escriurà a la posició actual. En aquest cas el desplaçament del punter de lectura/escriptura associat al fitxer estarà indefinit.

Si el *flag O_APPEND* (*open*) està actiu, el punter de lectura/escriptura se situarà al final de fitxer abans de cada escriptura.

En el cas que no es puguin escriure el nombre de bytes indicat al paràmetre *nbyte*, perquè no hi ha espai disponible o s'ha superat algun límit indicat pel sistema, la crida *write* només escriurà el nombre de bytes que sigui possible i en retornarà aquest nombre.

Si la crida *write* és interrompuda per un *signal* abans que hagi escrit res, retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Si la crida *write* és interrompuda per un *signal* després d'haver escrit alguna cosa, retornarà el nombre de bytes llegits.

L'execució correcta de la crida *write* sobre un fitxer regular tindrà les conseqüències següents:

- Si es fa un lectura (*read*) de les posicions del fitxer modificades per la crida *write*, retornarà les dades escrites per la crida *write* en aquestes posicions.
- Si es fan escriptures sobre posicions prèviament ja escrites, els valors de la darrera escriptura substituiran els valors existents.

Les escriptures a *pipes* tenen les mateixes conseqüències que als fitxers regulars llevat de les excepcions següents:

- No hi ha cap punter de lectura/escriptura associat amb la *pipe* i, per tant, totes les escriptures s'afegeixen al final de la *pipe*.

- L'escriptura a una *pipe* és indivisible i, per tant, garanteix que les dades de cap altra escriptura realitzada per altres processos, a la mateixa *pipe*, pugui mesclar-se amb les dades escrites per la crida *write*.
- Si els *flags O_NDELAY* i *O_NONBLOCK* de la *pipe* no estan actius (*open*), la crida *write* pot bloquejar el procés (per exemple si la *pipe* està plena), però al final de l'escriptura haurà escrit tots els bytes indicats i, per tant, retornarà el valor *nbyte*.
- Si els *flags O_NDELAY* i *O_NONBLOCK* de la *pipe* estan actius (*open*), la crida *write* no bloquejarà mai al procés. Si la crida pot escriure els bytes indicats, retornarà *nbyte*. Altrament, si el *flag O_NONBLOCK* està actiu, retornarà el valor -1 i la variable *errno* valdrà *EAGAIN*, o si el *flag O_NDELAY* està actiu, retornarà el valor 0.

Paràmetres

- *fildes* canal obert associat al fitxer al qual es vol escriure.
- *buf* punter a un espai de memòria on es guarden els bytes que es volen escriure.
- *nbyte* nombre de bytes que es volen escriure.

Valor retornat

Si la crida *write* acaba correctament retornarà un valor no negatiu que indicarà el nombre de bytes realment escrits al fitxer associat amb el canal (*file descriptor*) *fildes*. Aquest nombre mai no serà superior al valor del paràmetre *nbytes*. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *write* fallarà, entre d'altres raons, si:

- *EBADF*: el paràmetre *fildes* no és un canal obert (*file descriptor*) vàlid per a escriptura.
- *EDQUOT*: la quota de blocs de disc de l'usuari ha estat superada.
- *EFAULT*: el paràmetre *buf* apunta a una adreça il·legal.
- *EFBIG*: s'intenta escriure a un fitxer que supera la mida màxima de fitxer permesa al procés, o que supera la mida màxima de fitxer del sistema.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *write* i no hi ha hagut cap escriptura.
- *EPIPE*: s'intenta escriure a una *pipe* que ja no està oberta per a lectura per cap procés. Un *signal SIGPIPE* s'enviarà al procés en generar-se aquest error.

3.4.6 lseek

Sintaxi

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

Descripció

La crida *lseek* situa, en una posició determinada, el punter de lectura/escriptura del fitxer especificat pel canal (*file descriptor*) obert *fildes*. La posició final del punter dependrà del valor dels paràmetres *offset* i *whence* de la manera següent:

- Si *whence* val *SEEK_SET*, el punter se situa a la posició del valor d'*offset*.
- Si *whence* val *SEEK_CUR*, el punter se situa a la posició resultant de sumar el valor *offset* amb la posició actual del punter.
- Si *whence* val *SEEK_END*, el punter se situa a la posició resultant de sumar el valor *offset* al final del fitxer.

La crida *lseek* permet situar el punter de lectura/escriptura més enllà del final del fitxer. Si s'escriu en aquesta posició, les lectures que es facin en les posicions on no hi ha dades retornaran el valor zero fins que s'hi escrigui alguna cosa diferent.

Paràmetres

- *fildes* indica un canal (*file descriptor*) obert d'un fitxer.
- *offset* indica el desplaçament relatiu que es vol aplicar al punter de lectura/escriptura.
- *whence* indica la posició a partir de la qual s'afegirà el desplaçament.

Valor retornat

Si la crida acaba correctament es retornarà l'*offset* resultant, mesurat en bytes, del punter de lectura/escriptura respecte l'origen del fitxer. Altrament es retornarà el valor negatiu -1, l'*offset* no es modificarà i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *lseek* fallarà, entre d'altres raons, si:

- *EBADF*: el paràmetre *fildes* no és un canal (*file descriptor*) obert.
- *ENVAL*: el paràmetre *whence* no és *SEEK_SET*, *SEEK_CUR* o *SEEK_END*.
- *ESPIPE*: el paràmetre *fildes* està associat a una *pipe*.

3.4.7 dup

Sintaxi

```
#include <unistd.h>

int dup(int fildes);
```

Descripció

La crida *dup* retorna un nou canal (*file descriptor*) que té les característiques comunes següents amb el canal indicat al paràmetre *fildes*:

- Mateix fitxer obert o pipe.
- Mateix punter de lectura/escriptura, és a dir, els dos canals compartiran un sol punter.
- Mateixa modalitat d'accés (lectura, escriptura o lectura/escriptura).

Paràmetres

fildes indica el canal (*file descriptor*) que es vol duplicar.

Valor retornat

Si la crida acaba correctament es retornarà un valor enter no negatiu que correspondrà al primer canal (*file descriptor*) lliure disponible del procés. Altrament es retornarà el valor negatiu -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *dup* fallarà, entre d'altres raons, si:

- *EBADF*: el paràmetre *fildes* no és un canal (*file descriptor*) vàlid.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *dup*.
- *EMFILE*: el procés té massa fitxers oberts.

3.4.8 unlink

Sintaxi

```
#include <unistd.h>

int unlink(const char *path);
```

Descripció

La crida *unlink* elimina un nom (*link)* d'un fitxer i redueix el comptador de noms (*links*) del fitxer referenciat. Si el paràmetre *path* representa un enllaç simbòlic (*symbolic link*), la crida esborra l'enllaç simbòlic però no afecta al fitxer o al directori apuntat per l'enllaç.

Si el comptador d'enllaços d'un fitxer arriba a zero i cap procés té el fitxer obert, aleshores l'espai ocupat pel fitxer s'allibera i el fitxer ja no podrà ser accessible novament. Si algun procés té el fitxer obert quan s'esborra el darrer nom (*link*), aleshores els continguts del fitxer no s'eliminaran fins que tots els processos hagin tancat el fitxer.

Paràmetres

path apunta al nom complet de l'entrada (*link*) que es vol eliminar.

Valor retornat

Si la crida acaba correctament retornarà el valor 0. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *unlink* fallarà, entre d'altres raons, si:

- *EACCES*: no està permesa la cerca en algun dels components directori del *path*, no està permesa l'escriptura en el directori que conté el nom (*link*) que es vol esborrar, o l'usuari no és el propietari del directori que conté el nom (*link*) ni és el propietari del fitxer.
- *EFAULT*: el paràmetre *path* apunta a una adreça il·legal.
- *EINTR*: ha arribat un *signal* durant l'execució de la crida *unlink*.
- *ENOENT*: no existeix el nom del fitxer (*link*) o el paràmetre *path* està buit.
- *ENOTDIR*: algun component del prefix del paràmetre *path* no és un directori.
- *EROFS*: l'entrada que es vol esborrar forma part d'un sistema de fitxers de només lectura.

3.5 Exemples de les crides d'entrada/sortida

3.5.1 Exemple 1

Aquest exemple copia, caràcter a caràcter, un fitxer font en un altre fitxer destí. Els noms dels fitxers es passen com a paràmetres del programa. El programa obre el fitxer font (*open*), crea el fitxer destí (*open*) i va llegint (*read*) del fitxer font caràcter a caràcter i el va escrivint (*write*) al fitxer destí.

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ...         */
#include <errno.h>

void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main(int argc, char *argv[])
{
      int source, dest, n;
      char c;

      if (argc != 3)
            error("Nombre arguments erroni");

      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Obertura del fitxer font");

      /* Si el fitxer destí existeix, el sobreescriu (O_TRUNC)   */
      /* Si el fitxer destí no existeix, el crea (O_CREAT, 0600) */

      if ((dest = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0600)) < 0)
            error("Creació del fitxer destí");

      /* Llegeix del fitxer font mentre hi hagi informació */
      /* Escriu al fitxer destí tot el que ha llegit       */

      while ((n=read(source, &c, 1)) > 0)
            if (write(dest, &c, 1) < 0)
                  error("Escriptura al fitxer destí");

      if (n<0)
            error("Lectura del fitxer font");

      close (source);
      close (dest);
}
```

3.5.2 Exemple 2

Aquest exemple mostra la creació d'un fitxer temporer que no apareix en el sistema de fitxers mentre és utilitzat. Per crear-lo s'utilitza la crida a sistema *creat* i immediatament després s'esborra el nom del fitxer (*unlink*). A partir d'aquest moment, el fitxer segueix existint però sense accés per part de cap altre procés. En el moment que el procés tanca el canal obert associat al fitxer (*close*), el fitxer desapareix físicament del sistema de fitxers.

```
#include <fcntl.h>
#include <errno.h>

void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int fildes;

      /* Es crea el fitxer temp.dat amb permisos per al propietari */

      if ((fildes = creat("temp.dat", 0700)) < 0)
            error("Error en la creació del fitxer");

      /* S'esborra el nom (link) del fitxer creat */

      if (unlink("temp.dat") < 0)
            error("Error esborrant el nom (link) del fitxer");

      /* A partir d'aquest moment, el fitxer només és          */
      /* accessible mitjançant el canal fildes d'aquest procés */



      /* ... */



      /* Un cop tancat el canal fildes, el fitxer desapareix */

      close(fildes);
      exit(0);
}
```

3.5.3 Exemple 3

Aquest exemple escriu el contingut d'un fitxer en sentit invers, és a dir el primer caràcter d'un fitxer (*source*) passa a ser l'últim caràcter d'un altre fitxer (*dest*). Obre el fitxer origen (*source*) passat com a primer paràmetre (*open*) i crea el fitxer destí (*dest*) passat com a segon paràmetre (*open*). Llegeix el fitxer origen (*read*) en ordre invers (*lseek*) i escriu les dades llegides (*write*) en el fitxer destí.

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ...         */
#include <unistd.h> /* definicions de SEEK_SET, SEEK_CUR, SEEK_END */
#include <errno.h>

void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, dest;
      long index;
      char c;

      if (argc != 3)
            error("Arguments insuficients");
      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Obrir el fitxer origen");
      if ((dest = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0600)) < 0)
            error("Crear el fitxer destí");

      /* Es llegeix el fitxer origen des del final cap a l'inici */

      /* Es calcula la mida del fitxer amb el retorn de la       */
      /* crida lseek que posiciona el punter al final del fitxer */

      if ((index = lseek(source, 0, SEEK_END)) < 0)
            error("Càlcul mida fitxer amb lseek");
      index--;
      while (index >= 0) {

            if (lseek(source, index, SEEK_SET) < 0)
                  error("Posició amb lseek");
            if (read(source, &c, 1) < 0)
                  error("Lectura del fitxer origen");
            if (write(dest, &c, 1) < 0)
                  error("Escriptura del fitxer destí");
            index--;

      }

      close(source);
      close(dest);
}
```

3.5.4 Exemple 4

Aquest exemple mostra la compartició del punter de lectura/escriptura entre un procés pare i el seu procés fill que ha heretat la taula de canals (*file descriptors*) del pare. El procés pare obre el fitxer (*open*) abans de crear el procés fill, crea (*fork*) el procés fill que hereta la taula de canals i, per tant, comparteix el punter de lectura/escriptura associat al canal del fitxer. Els dos processos van llegint (*read*) del fitxer i escrivint (*write*) per la sortida estàndard fins arribar al final de fitxer. La sortida mostrarà el fitxer un sol cop gràcies a la compartició del punter de lectura/escriptura.

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ...         */
#include <errno.h>

void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, n, st;
      char c;

      if (argc != 2) error("Falten arguments");
      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Obrir el fitxer origen");

      switch (fork())
      {
            case -1: /* En cas d'error el procés acaba */
                  error("Fork");

            case 0: /* Procés fill - llegeix del fitxer i escriu */
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escriptura sortida estàndard");
                  if (n<0)
                        error("Lectura del fitxer font");
                  close(source);
                  exit(0);

            default: /* Procés pare - llegeix del fitxer i escriu */
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escriptura sortida estàndard");
                  if (n<0)
                        error("Lectura del fitxer font");
                  close(source);
                  wait(&st);
                  exit(0);
      }
}
```

3.5.5 Exemple 5

Si agafem el mateix codi que a l'exemple 5 però movem l'obertura (*open*) del fitxer i la posem després de la creació (*fork*) del procés fill, aleshores tant el procés pare com el procés fill escriuran el fitxer complet i, per tant, el contingut apareixerà dues vegades. En aquest programa el fill no ha heretat el canal obert del fitxer i, per tant, cada procés crea el seu propi canal amb la seva pròpia entrada a la taula de fitxers i amb punters de lectura/escriptura independents per a cada procés.

```
#include <fcntl.h>  /* definicions O_RDONLY, O_WRONLY, ...         */
#include <errno.h>

void error(char *m)
{

      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, n, st;
      char c;

      if (argc != 2) error("Falten arguments");

      switch (fork())
      {
            case -1: /* En cas d'error el procés acaba */
                  error("Fork");

            case 0: /* Procés fill - obre, llegeix i escriu */
                  if ((source = open(argv[1], O_RDONLY)) < 0)
                        error("Obrir el fitxer origen");
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escriptura sortida estàndard");
                  if (n<0)
                        error("Lectura del fitxer font");
                  close(source);
                  exit(0);

            default: /* Procés pare - obre, llegeix i escriu */
                  if ((source = open(argv[1], O_RDONLY)) < 0)
                        error("Obrir el fitxer origen");
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escriptura sortida estàndard");
                  if (n<0)
                        error("Lectura del fitxer font");
                  close(source);
                  wait(&st);
                  exit(0);
      }
}
```

4 La comunicació i sincronització entre processos

Tots els sistemes operatius que permeten l'execució concurrent de processos és indispensable que ofereixin eines de comunicació i sincronització. Unix ofereix diversos mecanismes de comunicació i sincronització i en aquest capítol en presentarem dos: els *signals* i les *pipes*. Els *signals* són senyals que permeten notificar un esdeveniment concret, però que no inclouen contingut a part del propi senyal. Les *pipes*, en canvi, permeten enviar missatges amb qualsevol tipus de contingut entre processos que mantinguin una relació de parentiu (pare/fill).

4.1 Els signals

Els *signals* són senyals asíncrons que poden ser enviats als processos per les causes següents:

- Un error en l'execució del procés: per exemple executar una instrucció il·legal (*signal SIGILL)*, intentar accedir a una adreça de memòria invàlida (*signal SIGSEGV*), ...
- Un avís del sistema operatiu sobre algun recurs relacionat amb el procés: indicació d'un canvi en l'estat d'algun fill (*signal SIGCHLD*), avís d'escriure a una *pipe* que no té lectors (*signal SIGPIPE*), ...
- Un avís d'un altre procés o del mateix procés per algun esdeveniment que el procés considera adient: senyal de l'alarma del propi procés (*signal SIGALRM*), senyal d'algun esdeveniment definit pel propi procés o conjunt de processos (*signal SIGUSR1*), ... Aquest enviament explícit de *signals* entre processos es fa mitjançant la crida a sistema *kill*.

Cada procés pot programar la seva resposta a cadascun dels *signals*. L'acció a prendre en rebre un *signal* determinat dependrà d'aquesta programació i es poden prendre les accions següents:

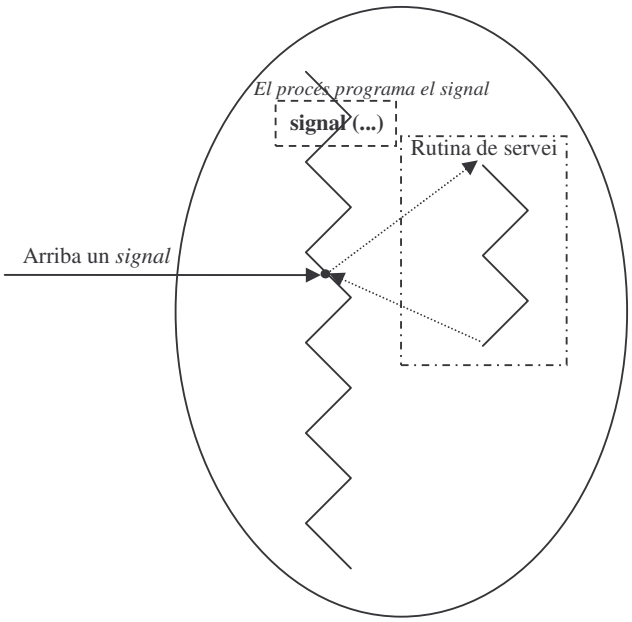
- Acció per defecte (*SIG_DFL)*: en la majoria dels *signals* l'acció per defecte serà la mort del procés que rep el *signal*. Aquesta acció és la que es pren si el procés no ha heretat cap altra programació, ni ha programat el *signal*.
- Ignorar el *signal* (*SIG_IGN)*: en aquest cas el *signal* no arribarà al procés i, per tant, no es prendrà cap acció.
- Programar una rutina de servei del *signal* (*signal handler*): en el moment de rebre el *signal*, l'execució del procés s'interromprà i s'executarà la rutina de servei programada. Un cop acabada l'execució de la rutina de servei es tornarà a l'execució del procés en el punt on s'havia interromput. Aquesta manera d'actuar s'assimila a la gestió d'interrupcions. Algunes versions de Unix reprogramen els *signals* a l'acció per defecte un cop es serveix un *signal*. Per aquesta raó, és convenient reprogramar novament el *signal* dins de la rutina de servei.

Alguns *signals* no són reprogramables i, en rebre'ls, sempre es pren l'acció per defecte: per exemple el *signal SIGKILL* (sempre mata el procés) i el *signal SIGSTOP* (sempre atura el procés). Una excepció és el *signal SIGCHLD* que per defecte s'ignora.

La programació dels *signals* s'hereta (després d'un *fork*) de pares a fills. En el cas que un procés faci un canvi d'imatge (crida a sistema *exec*) només es mantindran els *signals* programats per ser ignorats (*SIG_IGN)* o per prendre l'acció per defecte

(*SIG_DFL)*. En canvi els *signals* que s'han programat amb una rutina de servei (*signal handler*) passaran a la programació per defecte, ja que amb la nova imatge desapareixerà la rutina de servei programada. La programació dels *signals* es realitza amb la crida a sistema del mateix nom (*signal*).

Tot seguit es mostra l'execució d'un procés que rep un *signal* programat (amb la crida a sistema *signal*) amb una rutina de servei. En el moment que el *signal* arriba, s'interromp l'execució del codi del procés i s'executa la rutina de servei. Un cop la rutina de servei acaba, es torna al punt d'execució on s'havia interromput prèviament:



Taula de *signals* segons l'estàndard *POSIX*:

Senyal	**Significat del senyal**	**Número**
SIGABRT	Acabament anormal d'un procés	6
SIGALRM	Senyal produït per l'alarma del procés	14
SIGFPE	Excepció aritmètica	8
SIGHUP	Tall de comunicació amb la línia (terminal, mòdem ...)	1
SIGILL	Instrucció il·legal	4
SIGINT	Interrupció produïda des del terminal (combinació de tecles)	2
SIGQUIT	Acabament produït des del terminal (combinació de tecles)	3
SIGKILL	Matar el procés (no es pot programar ni ignorar)	9
SIGPIPE	S'ha escrit a una *pipe* sense lectors	13
SIGSEGV	Accés a memòria invàlid	11
SIGTERM	Senyal d'acabament d'un procés	15
SIGUSR1	*Signal* 1 definit per l'usuari	10
SIGUSR2	*Signal* 2 definit per l'usuari	12
SIGCHLD	L'estat d'un fill ha canviat	17
SIGCONT	Senyal de continuació si el procés està aturat	18
SIGSTOP	Senyal d'*stop* (no és pot programar ni ignorar)	19
SIGTSTP	Senyal d'*stop* produït des del terminal (combinació de tecles)	20
SIGTTIN	Procés en segon pla que intenta llegir del terminal	21
SIGTTOU	Procés en segon pla que intenta escriure al terminal	22

Tots els *signals* de l'1 al 15 provoquen, per defecte, la mort del procés. Cal destacar que el *signal SIGKILL* no es pot programar ni ignorar i s'utilitza per matar definitivament els processos.

Els *signals* 17 a 22 s'oferiran només en el cas que el control de treballs (*job control*) estigui definit en la versió del sistema. Per defecte *SIGCHLD* s'ignora, en canvi *SICONT* provoca la continuació d'un procés aturat i la resta (*SIGSTOP*, *SIGTSTP*, *SIGTTIN*, *SIGTTOU*) provoquen l'aturada del procés que els rep.

4.2 Les pipes

Les *pipes* són dispositius lògics que permeten la comunicació entre processos amb una relació de parentiu entre ells, és a dir, entre processos que comparteixin aquest dispositiu per herència. Les *pipes* són canals unidireccionals de comunicació i, en ser creades (crida a sistema *pipe*), retornen dos *file descriptors* dins de la taula de canals del procés, un per a lectura i un altre per a escriptura:

```
int p[2];

...

/* Es crea una pipe i es rebem els dos canals:   */
/* el de lectura i el d'escriptura (p[0] i p[1]) */

if (pipe(p) < 0) error("Creació pipe");

...
```

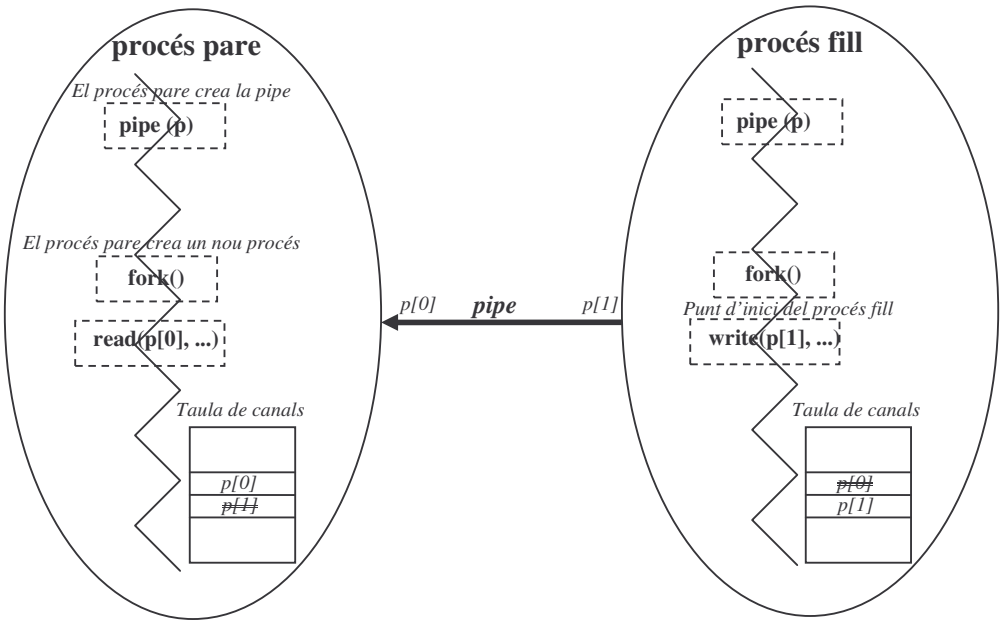
En el codi anterior el procés crea un *pipe* que és accessible a través dels canals (*file descriptors*) *p[0]* i *p[1]*. El canal *p[0]* és de lectura i el canal *p[1]* és d'escriptura.

Per poder utilitzar la *pipe* per comunicar-se amb un altre procés, cal que els dos processos la comparteixin en les seves respectives taules de canals. Per això, és necessari que el procés que ha creat la *pipe* creï un nou procés que hereti els canals de la *pipe*, a fi que pugui comunicar-se amb el procés creador.

En la pròxima figura es mostren dos processos (pare i fill) que es comuniquen mitjançant una *pipe*. El procés pare crea la *pipe* (crida a sistema *pipe*) i tot seguit crea el procés fill (*fork*).

Després de la creació de la *pipe*, a la taula de canals del procés pare apareixen dues noves entrades*, p[0]* i *p[1]*, que corresponen als canals de lectura i d'escriptura de la *pipe*. De moment, però, la *pipe* només la pot utilitzar el procés que l'ha creada, ja que la *pipe* només existeix en la seva taula de canals. Després de la creació del nou procés, el procés fill dispos d'una taula de canals idèntica a la del pare i, per tant, comparteix els dos canals d'accés a la *pipe*. A partir d'aquest moment, pare i fill podran comunicar-se mitjançant la mateixa *pipe*. En l'exemple presentat el fill escriu un missatge a la *pipe* mitjançant el seu canal d'escriptura a la *pipe p[1]*. Aquest missatge és llegit pel pare mitjançant el seu canal de lectura de la *pipe p[0]*.

A fi de poder aprofitar totes les característiques de les *pipes*, com s'explicarà tot seguit, els processos haurien de tancar els canals que no utilitzin abans d'iniciar la comunicació. En l'exemple presentat, el procés pare només utilitzarà el canal de lectura de la *pipe* (*p[0]*) i, per tant, tancarà el seu canal d'escriptura (*p[1]*). Per la seva banda, el procés fill només utilitzarà el canal d'escriptura de la *pipe* (*p[1]*) i, per tant, tancarà el seu canal de lectura (*p[0]*).



El codi presentat tot seguit mostra les accions que faran pare i fill per poder comunicar-se mitjançant una *pipe* segons l'esquema anterior.

```
#include <errno.h>

void error(char *m)
{
      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int p[2];
      int st;
      char buff[20];

      /* Es crea una pipe i es reben els dos canals:   */
      /* el de lectura i el d'escriptura (p[0] i p[1]) */

      if (pipe(p) < 0) error("Creació pipe");

      switch (fork())
      {
            case -1: error("Fork 1");

            case 0: /* Procés fill */

                  /* Tanca el canal de la pipe que no utilitza */
```

```
                close(p[0]);

                /* Escriu un missatge a la pipe, la tanca i acaba */

                write(p[1],"Hola\n", 5);
                close(p[1]);
                exit(0);

          default: /* Procés pare */
                break;
    }

    /* El pare tanca el canal de la pipe que no utilitza */

    close(p[1]);

    /* Llegeix el missatge de la pipe i l'escriu per stdout */

    read(p[0], buff, 5);
    write(1, buff, 5);

    /* Espera que acabi el fill i finalitza la seva execució */

    wait(&st);
    exit(0);
}
```

La lectura d'una *pipe* provoca el bloqueig del procés que intenta llegir fins que la *pipe* disposi del total de bytes que es volen llegir. Així mateix, l'escriptura d'una *pipe* escriu el missatge i retorna immediatament llevat que la *pipe* estigui plena. En aquest cas, el procés que intenta escriure quedarà bloquejat fins que hi hagi espai suficient a la *pipe* per escriure tot el missatge. Aquest és el comportament habitual de les *pipes* en la lectura i escriptura, sempre i quan hi hagi algun procés que tingui obert algun canal de lectura i algun canal d'escriptura sobre la *pipe*. Altrament, el comportament és força diferent:

- Si no hi ha cap procés que tingui un canal d'escriptura obert d'una *pipe*, és a dir, si ja no és possible que ningú escrigui mai més res a la *pipe*, aleshores la lectura de la *pipe* mai no bloquejarà el procés i retornarà immediatament amb el nombre de bytes llegits disponibles a la *pipe*. I si la *pipe* està buida la lectura retornarà 0 bytes llegits.

- Si no hi ha cap procés que tingui un canal de lectura obert d'una *pipe*, és a dir, si ja no és possible que ningú llegeixi mai més res de la *pipe*, aleshores un intent d'escriptura sobre la *pipe* provocarà que el procés rebi un *signal SIGPIPE* i que no s'escrigui res a la *pipe*. El *signal SIGPIPE* avisarà, al procés que intenta escriure, que ja no és possible que cap procés llegeixi de la *pipe*.

Aquest comportament especial de les *pipes* que no disposen dels dos canals (el d'escriptura i el de lectura) permet un control detallat del final de la comunicació entre els processos implicats. Per aquesta raó és molt important que sempre es tanquin tots els canals que no s'utilitzin de les *pipes*. Altrament la finalització d'algun dels processos, que es comuniquen amb la *pipe*, no seria detectat per l'altre procés i provocaria un bloqueig permanent del procés, ja sigui en l'intent de llegir d'una *pipe* buida o en l'intent d'escriure a una *pipe* plena.

4.3 Crides a sistema de comunicació i sincronització entre processos

En aquest apartat es presenten les crides següents relacionades amb la comunicació i sincronització entre processos:

- *pipe*: crea una *pipe* de comunicació i retorna els canals (*file descriptors*) de lectura i escriptura
- *signal*: programa l'acció que es prendrà en arribar un *signal* determinat
- *kill*: envia un *signal* a un procés o a un grup de processos
- *alarm*: programa el rellotge del procés perquè li enviï un *signal SIGALRM* al cap d'un cert temps
- *pause*: bloqueja el procés fins que arriba un *signal*

4.3.1 pipe

Sintaxi

```
#include <unistd.h>

int pipe(int fildes[2]);
```

Descripció

La crida *pipe* crea una mecanisme de comunicació entre processos anomenat *pipe* i retorna dos canals oberts (*file descriptors*) que correspondran al canal de lectura sobre la *pipe* (*fildes[0]*) i al canal d'escriptura sobre la *pipe* (*fildes[1]*).

La lectura de la *pipe* pel canal *fildes[0*] accedeix a les dades escrites a la *pipe* pel canal *fildes[1]* seguint una estructura *FIFO* (*first-in-first-out*).

Paràmetres

fildes es una taula on la crida retornarà els dos canals (*file descriptors*) d'accés a la *pipe*.

Valor retornat

Si la crida acaba correctament retornarà el valor 0. Altrament retornarà el valor -1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *pipe* fallarà si:

- *EMFILE*: el nombre de canals oberts supera el màxim permès al procés.
- *ENFILE*: no hi ha espai per a una nova entrada a la taula de fitxers.

4.3.2 signal

Sintaxi

```
#include <signal.h>

void (*signal (int sig, void (*disp)(int)))(int);
```

Descripció

Aquesta crida permet programar l'acció que prendrà el procés davant la rebuda d'un *signal*. Els signals es poden programar de tres formes diferents:

- *SIG_DFL*: es prendrà l'acció per defecte definida per al *signal* indicat al paràmetre *sig*.
- *SIG_IGN*: s'ignorarà la rebuda del *signal* indicat al paràmetre *sig*.
- L'adreça de la rutina de servei (*signal handler*): cada cop que arribi un *signal* del tipus definit al paràmetre *sig* s'executarà la rutina de servei indicada.

El sistema garanteix que si s'envia més d'un *signal* del mateix tipus a un procés, el procés rebrà almenys un d'aquests *signals*. No es garanteix, però, la recepció de cadascun dels *signals* enviats.

Paràmetres

- *sig* indica el *signal* que es vol programar i no pot valer ni *SIGKILL* ni *SIGSTOP*.
- *disp* indica la programació del *signal* que pot ser *SIG_DFL*, *SIG_IGN* o l'adreça de la rutina de servei del *signal* (*signal handler*).

Valor retornat

Si la crida *signal* acaba correctament, retornarà la programació prèvia del *signal* corresponent. Altrament retornarà *SIG_ERR* i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *signal* fallarà si:

- *EINTR*: ha arribat un *signal* durant l'execució de la crida *signal*.
- *EINVAL*: el valor del paràmetre *sig* no és un valor de *signal* vàlid o és igual a *SIGKILL* o *SIGSTOP*.

4.3.3 kill

Sintaxi

```
#include <sys/types.h>
#include <signal.h>

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

Descripció

La crida *kill* envia un *signal* a un procés o a un grup de processos (*process group*). El procés o el grup de processos s'especifica al paràmetre *pid* i el *signal* enviat s'especifica al paràmetre *sig*. Si el paràmetre *sig* val 0 (*null signal*), aleshores la crida *kill* verifica la validesa de l'identificador indicat al *pid* però no s'envia cap *signal*.

L'identificador d'usuari real o efectiu (*real* o *effective user ID*) del procés que envia el *signal* ha de coincidir amb l'identificador real del procés que rep el *signal*, excepte en el cas que el procés que envia el *signal* pertanyi al *super-user*.

Si el paràmetre *pid* és més gran que 0, el *signal* s'enviarà al procés que tingui aquest identificador.

Si el paràmetre *pid* és negatiu i diferent de -1, el *signal* s'enviarà a tots els processos que pertanyin al grup amb identificador igual al valor absolut de *pid* i per als quals es tingui permís per enviar un *signal*.

Si el paràmetre *pid* val 0, el *signal* s'enviarà a tots els processos que pertanyin al mateix grup que el procés que envia el *signal*.

Si el paràmetre *pid* val -1 i l'usuari efectiu (*effective user*) del procés que envia el *signal* no és *super-user*, aleshores el *signal* s'enviarà a tots els processos que tinguin un identificador d'usuari real (*real user ID*) igual a l'identificador d'usuari efectiu (*effective user ID*) del procés que envia el *signal*.

Si el paràmetre *pid* val -1 i l'usuari efectiu (*effective user*) del procés que envia el *signal* és *super-user*, aleshores el *signal* s'enviarà a tots els processos

El sistema garanteix que si és s'envia més d'un *signal* del mateix tipus a un procés, el procés rebrà almenys un d'aquests *signals*. No es garanteix, però, la recepció de cadascun dels *signals* enviats.

Paràmetres

- *pid* indica l'identificador del procés o del grup de processos (*process group*) a qui es vol enviar el *signal*.
- *sig* indica el *signal* que es vol enviar.

Valor retornat

Si la crida acaba correctament retornarà el valor 0. Altrament retornarà el valor -1, no s'enviarà cap *signal* i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *kill* fallarà si:

- *EINVAL*: el valor del paràmetre *sig* no és un valor de *signal* vàlid.
- *EPERM*: si no es tenen permisos per enviar un *signal* a un determinat procés.
- *ESRCH*: no existeix cap procés ni cap grup de processos (*process group*) amb l'identificador indicat en el paràmetre *pid*.

4.3.4 alarm

Sintaxi

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

Descripció

La crida *alarm* programa el rellotge del procés que la invoca a fi que enviï al propi procés un *signal SIGALRM* després d'un determinat nombre de segons especificats al paràmetre *sec*.

Les peticions fetes amb la crida *alarm* per un mateix procés no s'acumulen, sinó que una nova crida anul·la l'anterior.

Si el paràmetre *sec* val 0, es cancel·la qualsevol petició d'alarma feta prèviament.

La crida *fork* anul·la en el fill la programació de l'alarma que tingui el procés pare. En canvi la crida *exec* manté l'alarma que s'hagi programat abans de la seva invocació.

Paràmetres

sec indica el nombre de segons a partir dels quals s'enviarà un *signal SIGALRM* al procés que ha invocat la crida.

Valor retornat

La crida *alarm* retorna el temps que encara falta perquè arribi la proper alarma del procés.

Errors

No hi ha cap error definit.

4.3.5 pause

Sintaxi

```
#include <unistd.h>

int pause(void);
```

Descripció

La crida *pause* suspèn l'execució del procés que l'ha invocat fins que el procés rep un *signal* que no hagi estat programat per ser ignorat.

Si el *signal* provoca la finalització del procés, aleshores la crida *pause* ja no retorna.

Valor retornat

La crida *pause* sempre retorna error amb el valor –1 i la variable *errno* indicarà l'error que s'ha produït.

Errors

La crida *pause* fallarà si:

- *EINTR*: ha arribat un *signal* durant l'execució de la crida *pause*.

4.4 Exemples de les crides de comunicació i sincronització entre processos

4.4.1 Exemple 1

Aquest exemple mostra la programació (*signal*) d'una alarma (*alarm*) que s'envia al procés cada 3 segons. El procés es bloqueja (*pause*) i espera l'arribada de l'alarma.

```
#include <signal.h>

#define TEMPS 3


/* Rutina d'atenció al SIGALRM */

void rutina_alarma(int foo)
{
      char s[20];

      /* Redefineix la programació del signal SIGLARM */

      signal(SIGALRM, rutina_alarma);

      /* Escriu un missatge cada cop que arriba l'alarma */

      write(1, "Alarma 3 segons\n", strlen("Alarma 3 segons\n"));

      /* Sol·licita un signal SIGLARM d'aquí a TEMPS segons */

      alarm(TEMPS);
}


main()
{
      /* Defineix la programació del signal SIGALRM) */

      signal(SIGALRM, rutina_alarma);

      /* Sol·licita un SIGLARM d'aquí a TEMPS segons */

      alarm(TEMPS);

      /* Bucle infinit en espera de l'alarma*/

      while(1)
      {
            /* Bloqueja al procés fins que arriba el signal */

            pause();
      }
}
```

4.4.2 Exemple 2

Aquest exemple mostra la creació (*fork*) de dos processos fills que es comuniquen amb una *pipe*. El procés pare crea la *pipe* (*pipe*) i tot seguit crea (*fork*) el primer procés fill. Aquest primer procés fill hereta la *pipe* del pare, redirecciona (*close* i *dup*) la seva sortida estàndard cap a la *pipe* i tanca (*close*) els canals que no utilitza de la *pipe*. Tot seguit executa (*exec*) el fitxer executable *ls*. El pare crea (*fork*) el segon procés fill. El segon procés fill hereta la *pipe* del pare, redirecciona (*close* i *dup*) la seva entrada estàndard cap a la *pipe*, tanca (*close*) els canals que no utilitza de la *pipe* i executa (*exec*) el fitxer executable *grep*. El pare tanca (*close*) els canals de la *pipe* que no utilitza, espera (*wait*) la finalització dels dos fills i acaba (*exit*). L'execució dels dos fills equival a la comanda: `ls -l /urs/bin | grep cat`

```
#include <errno.h>

void error(char *m)
{
      /* Escriu els errors pel canal estàndard d'errors (canal 2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int p[2];
      int st1, st2;

      /* Es crea una pipe i es reben els dos canals:   */
      /* el de lectura i el d'escriptura (p[0] i p[1]) */

      if (pipe(p) < 0) error("Creació pipe");

      switch (fork())
      {
            case -1: error("Fork 1");

            case 0: /* Fill 1 - Redirecciona, tanca i executa */

                  /* Redirecciona sortida estàndard cap a la pipe */

                  close(1); dup(p[1]);

                  /* Tanca els canals de la pipe que no utilitza */

                  close(p[0]); close(p[1]);

                  /* Carrega el codi de ls amb dos paràmetres*/

                  execlp("ls", "ls", "-l", "/usr/bin", (char *)0);
                  error("Execució ls");

            default: /* Procés pare */
                  break;
      }
```

```
        switch (fork())
        {
              case -1: error("Fork 2");

              case 0: /* Fill 2 - Redirecciona, tanca i executa */

                    /* Redirecciona entrada estàndard cap a la pipe */

                    close(0); dup(p[0]);

                    /* Tanca els canals de la pipe que no utilitza */

                    close(p[0]); close(p[1]);

                    /* Carrega el codi de grep amb un paràmetre */

                    execlp("grep", "grep", "cat", (char *)0);
                    error("Execució grep");

              default: /* Procés pare */
                    break;
        }

        /* El pare tanca els canals que no utilitza */

        close(p[0]); close(p[1]);

        /* Espera que acabin els dos fills */

        wait(&st1);
        wait(&st2);

        /* Finalitza la seva execució */

        exit(0);
}
```

Si algun dels processos no tanqués els canals de la *pipe* que no utilitza la finalització dels processos no seria correcta. Per exemple, si el pare no tanca els dos canals de la *pipe* abans d'esperar la finalització dels fills, provocarà que el procés *grep* es quedi bloquejat indefinidament llegint (*read*) de la seva entrada estàndard un cop s'hagi acabat el procés *ls*. Conseqüentment el procés pare tampoc no acabarà perquè es quedarà bloquejat esperant (*wait*) la finalització del seu segon fill.

5 Bibliografia

- Manual en línia de Unix: comanda `man`.
- Kernighan, Brian W. i Pike, Rob. *El entorno de programación UNIX*. Prentice-Hall Hispanoamericana. México, 1987.
- Robbins, Kay A. i Robbins, Steven. UNIX programación práctica. Prentice-Hall Hispanoamericana. México, 1997.
- Stevens W.Richard. *Advanced Programming in the Unix Environment*. Reading, Mass.: Addison-Wesley, 2001.